Java Data Structures and Algorithms

Christopher Fox

bookboon
The eBook company

CHRISTOPHER FOX

JAVA DATA STRUCTURES AND ALGORITHMS

Java Data Structures and Algorithms
1st edition

ISBN 978-87-403-2554-6
Peer review by

CONTENTS

PREFACE

Typical algorithms and data structures textbooks are seven or eight hundred pages long, include chapters about software engineering and the programming language used in the book, and include appendices with yet more information about the programming language. Often they include lengthy case studies with tens of pages of specifications and code. Frequently they are hardcover books printed in two colors; sometimes they have sidebars with various sorts of supplementary material. All of these characteristics make these textbooks very expensive (and very heavy), but in my experience, relatively few students take advantage of the bulk of this material and few appreciate these books' many features: much of the time and money lavished on these texts is wasted on their readers.

Students seem to prefer dealing with only essential material compressed into the fewest number of pages. Perhaps this is attributable to habits formed by life on the Internet, or perhaps it is due to extreme pragmatism. But whatever the reason, it seems very difficult to persuade most computer science students to engage with long texts, large examples, and extra material, no matter how well it is presented and illustrated. This text is a response to this tendency.

This text covers the usual topics in an introductory survey of algorithms and data structures, but it does so in under 200 pages. There are relatively few examples and no large case studies. Code is presented in Java (more about this in a moment), but the book does not teach Java and it does not include reference material about the language. There are no sidebars and the book is in black and white. The book does include features of pedagogical value: every chapter has review questions with answers, and a set of exercises. There is also a glossary at the end. Versions of this book using Java and other programming languages have been used successfully for several years to teach introductory algorithms and data structures at James Madison University. Many students have commented appreciatively regarding its brevity, clarity, and low cost.

Ideally, a language for algorithms and data structures would be easy to learn (so as to leave time for learning algorithms and data structures), support data abstraction well, provide a good development environment, and engage students. Although Java is large and therefore a challenge to learn, a subset of it adequate for the needs of an introductory algorithms and data structures course can be learned with some effort. Java is an object-oriented language with interfaces and generics, so it does an excellent job supporting data abstraction. It is also statically typed and provides assertions, so it provides lots of help to novice programmers. Finally, it is fully supported by Eclipse, including the industry standard JUnit testing framework, which help students a great deal. Thanks to its popularity, many students learn Java in high school or on their own, and those who do not know it want to learn it. Overall, Java is an excellent choice for teaching algorithms and data structures.

Very few algorithms and data structures books do a good job teaching programming, and there is no reason for them to try. An algorithms and data structures book should concentrate on its main topic; students can learn the programming language from any of the many excellent books devoted to teaching it. Especially when an algorithms and data structures text is either free or only costs a few dollars, it is quite reasonable to expect students to also obtain a programming language tutorial. There are a plethora of introductory Java programming texts, any of which would be a useful companion to this book.

This text is informed by several themes:

Abstract data typing—All types are presented as implementations of abstract data types, and strong connections are made between the data structures used to represent values and the carrier set of the ADT, and the algorithms used to manipulate data and the method set of the ADT.

Contiguous versus linked representations—Two implementation strategies are considered for every ADT: one using contiguous memory locations (arrays), and one using linked structures.

Container hierarchy—A container hierarchy is built using the data structures studied in the text. Although modest, this hierarchy introduces the notion of a container library like the ones that students will encounter in many languages, and it shows how various containers are related to one another.

Assertions—Preconditions, post-conditions, class invariants, and unreachable-code assertions are stated wherever appropriate, and exceptions are raised when assertions are violated.

All the code appearing in the book has been written and tested under Java version 1.8. Code implementing the container hierarchy and the searching and sorting algorithms covered in the book is downloadable from Github at github.com/foxcjmu/javaDataStructures.

I thank Nathan Sprague for many constructive criticisms and error corrections that have improved the book. I also thank my students for suggestions and corrections, and for being willing to test the book for me.

Christopher Fox
July, 2018

1 INTRODUCTION

1.1 WHAT ARE DATA STRUCTURES AND ALGORITHMS?

If this book is about data structures and algorithms, then perhaps we should start by defining these terms. We begin with a definition for "algorithm."

> **Algorithm**: A finite sequence of steps for accomplishing some computational task. An algorithm must

- Have steps that are simple and definite enough to be done by a computer, and
- Terminate after finitely many steps.

This definition of an algorithm is similar to others you may have seen in prior computer science courses. Notice that an algorithm is a sequence of steps, not a program. You might use the same algorithm in different programs, or express the same algorithm in different languages, because an algorithm is an entity that is abstracted from implementation details. Part of the point of this course is to introduce you to algorithms that you can use no matter what language you program in. We will write programs in a particular language, but we are really studying the algorithms, not their implementations.

The definition of a data structure is a bit more involved. We begin with the notion of an abstract data type.

> **Abstract data type (ADT)**: A set of values (the **carrier set**), and operations on those values (the **method set**).

Here are some examples of ADTs:

> *Boolean*—The carrier set of the Boolean ADT is the set {true, false}. The method set includes negation, conjunction, disjunction, conditional, is equal to, and perhaps some others.
>
> *Integer*—The carrier set of the Integer ADT is the set {..., -2, -1, 0, 1, 2, ...}, and the method set includes addition, subtraction, multiplication, division, remainder, is equal to, is less than, is greater than, and so on. Note that although some of these operations yield other Integer values, some yield values from other ADTs (like true and false), but all have at least one Integer argument.

String—The carrier set of the String ADT is the set of all finite sequences of characters from some alphabet, including the empty sequence (the empty string). The method set includes concatenation, length of, substring, index of, and so forth.

Bit String—The carrier set of the Bit String ADT is the set of all finite sequences of bits, including the empty strings of bits, which we denote λ. This set is {λ, 0, 1, 00, 01, 10, 11, 000, ...}. The method set of the Bit String ADT includes complement (which reverses all the bits), shifts (which rotates a bit string left or right), conjunction and disjunction (which combine bits at corresponding locations in the strings), and concatenation and truncation.

The thing that makes an abstract data type *abstract* is that its carrier and method sets hold mathematical entities, like numbers or geometric objects, and functions on them; all details of implementation on a computer are ignored. This makes it easier to reason about them and to understand what they are. For example, we can decide how *div* and *mod* should work for negative numbers in the Integer ADT without having to worry about how to make this work on real computers. Then we can deal with implementations of our decisions as a separate problem.

Once an abstract data type is implemented on a computer, we call it a data type.

Data type: An implementation of an abstract data type on a computer.

Thus, for example, the Boolean ADT is implemented as the `boolean` type in Java, and the `bool` type in Go and C++; the Integer ADT is realized as the `int` and `long` types in Java, and the `Integer` class in Ruby; the String ADT is implemented as the `String` class in Java and the `string` type in C++.

Abstract data types are very useful for helping us understand the mathematical objects that we use in our computations but, of course, we cannot use them directly in our programs. To use ADTs in programming, we must figure out how to implement them on a computer. Implementing an ADT requires two things:

- Representing the values in the carrier set of the ADT by data stored in computer memory, and
- Realizing computational mechanisms for the operations in the method set of the ADT.

Finding ways to represent carrier set values in a computer's memory requires that we determine how to arrange data (ultimately bits) in memory locations so that each value of the carrier set has a unique representation. Such things are data structures.

> **Data structure**: An arrangement of data in memory locations to represent values of the carrier set of an abstract data type.

Realizing computational mechanisms for performing the operations in the method set of the type really means finding algorithms that use the data structures for the carrier set to implement the operations in the method set. And now it should be clear why we study data structures and algorithms together: to implement an ADT, we must find data structures to represent the values of its carrier set and algorithms to work with these data structures to implement the operations in its method set.

A course in data structures and algorithms is thus a course in implementing abstract data types. It may seem that we are paying a lot of attention to a minor topic, but abstract data types are really the foundation of everything we do in programming. Our computations work on data. This data must represent things and be manipulated according to rules. These things and the rules for their manipulation amount to abstract data types.

Usually there are many ways to implement an ADT. A large part of the study of data structures and algorithms is learning about alternative ways to implement ADTs and evaluating the alternatives to determine their advantages and disadvantages. Typically some alternatives will be better for certain applications and other alternatives will be better for other applications. Knowing how to do such evaluations to make good design decisions is an essential part of becoming an expert programmer.

As we have noted, a data type in a programming languages is an implementation of an abstract data type. Thus the set of values in a data type is a set of *representations* of the values in the carrier set of the corresponding ADT. But we are usually more interested in the values represented than the representations themselves, so when we refer to the **carrier set of a data type**, we will (usually) mean the set of values from the corresponding ADT represented in the data type. Similarly, a data type has implementations of algorithms in its method set, but when we refer to the **method set of a data type** we will (usually) mean the operations of the corresponding ADT. For example, the carrier set of the `int` data type in Java is the set of integers between -2147483648 and 2147483647, and the method set of this data type includes addition, subtraction, multiplication, and so on.

1.2 STRUCTURE OF THE BOOK

In this book we will begin by studying fundamental data types that are usually implemented for us in programming languages. Then we will consider how to use these fundamental types and other programming language features (such as classes and interfaces) to implement more

complicated ADTs. Along the way we will construct a classification of complex ADTs that will serve as the basis for a library of implementations. We will also learn how to measure an algorithm's efficiency and use this skill to study algorithms for searching and sorting, which are very important in making our programs efficient when they must process large data sets.

1.3 THE JAVA PROGRAMMING LANGUAGE

Although the data structures and algorithms we study are not tied to any program or programming language, we need to write particular programs in particular languages to practice implementing and using the data structures and algorithms that we learn. In this book, we will use the Java programming language.

Java is an object-oriented language designed for general purpose programming that is widely used in education, industry, and government. This book uses Java under the assumption that the reader already knows it. We will use Java language features in our discussion and examples without explaining them, allowing us to concentrate on data structures and algorithms, which is what we are really interested in. Furthermore, Java's rich class, interface, and generic features allow us to implement a powerful library quite easily.

1.4 REVIEW QUESTIONS

1. What are the carrier set and some operations of the Character ADT?
2. How might the Bit String ADT carrier set be represented on a computer in some high level language?
3. How might the concatenation operation of the Bit String ADT be realized using the carrier set representation you devised for question two above?
4. What do your answers to questions two and three above have to do with data structures and algorithms?

1.5 EXERCISES

1. Describe the carrier and method sets for the following ADTs:
   a) The Real numbers
   b) The Rational numbers
   c) The Complex numbers
   d) Ordered pairs of Integers
   e) Sets of Characters
   f) Grades (the letters A, B, C, D, and F)

2. For each of the ADTs in exercise one, either indicate how the ADT is realized in some programming language, or describe how the values in the carrier set might be realized using the facilities of some programming language, and sketch how the operations in the method set might be implemented.

1.6 REVIEW QUESTION ANSWERS

1. We must first choose a character set; suppose we use the ASCII characters. Then the carrier set of the Character ADT is the set of ASCII characters. Some operations of this ADT might be those to change character case from lower to upper and the reverse, classification operations to determine whether a character is a letter, a digit, whitespace, punctuation, a printable character, and so forth, and operations to convert between integers and characters.
2. Bit String ADT values could be represented in many ways. For example, bit strings might be represented in character strings of "0"s and "1"s. They might be represented by arrays or lists of characters, Booleans, or integers.
3. If bit strings are represented as characters strings, then the bit string concatenation operation is realized by the character string concatenation operation. If bit strings

are represented by arrays or lists, then the concatenation of two bit strings is a new array or list whose size is the sum of the sizes of the argument data structures consisting of the bits from the first bit string copied into the initial portion of the result array or list, followed by the bits from the second bit string copied into the remaining portion.

4. The carrier set representations described in the answer to question two are data structures, and the implementations of the concatenation operation described in the answer to question three are (sketches of) algorithms.

2 BUILT-IN TYPES

2.1 SIMPLE AND STRUCTURED TYPES

Virtually every programming language has implementations of several ADTs built into it. We distinguish two kinds of built-in types:

> **Simple types:** The values of the carrier set are atomic, that is, they cannot be divided into parts. Common examples of simple types are integer, Boolean, character, floating point, and enumerations. Some languages also provide string as a built-in simple type.
>
> **Structured types**: The values of the carrier set are not atomic, consisting instead of several atomic or structured values arranged in some way. Common examples of structured types are arrays, records, classes, and sets. Some languages treat string as a built-in structured type.

Note that both simple and structured types are implementations of ADTs; the distinction is simply a question of how the programming language treats the values of the carrier set of the ADT in its implementation. The remainder of this chapter considers simple and structured types and uses Java to illustrate these ideas.

2.2 SIMPLE TYPES IN JAVA

Java has a typical collection of simple types similar to those of many other popular imperative programming languages. The Java language specification calls these *primitive types*. The simplest primitive type is `boolean`, which (as noted above) implements the Boolean ADT. The `char` type has a subset of the Unicode characters as its carrier set (Unicode is a standard for representing the characters used in most languages in the world). The `char` type method set is accessible through the `Character` class and includes operations for testing properties of a character (such as whether it is a digit or whether it is uppercase), and converting characters to other types or other characters (such as making a character uppercase).

Java has four integer types (`byte`, `short`, `int`, and `long`) that differ in the number of bits they use to represent carrier set values, and therefore in the size of their carrier sets. Their method sets are the same. Java has two floating point types (`float` and `double`) which again differ only in the number of bits they use to represent carrier set values, which determines the range and the precision with which they can represent real numbers. The carrier sets of both types include five special values: positive and negative infinity, positive

and negative zero, and not-a-number (`NaN`). Besides all the operations one would expect for numeric types (addition, subtraction, multiplication, various sorts of division, comparisons, and integer increment and decrement operations), Java provides bitwise operations (such as shifts and bitwise and, bitwise or, and bitwise complement), and implicit and explicit type conversions between numeric types.

Strings are not simple types in Java; we discuss Java strings in more detail later.

2.3 STRUCTURED TYPES IN JAVA

Java has only three structured types.

> *Array*—A fixed length, ordered collection of values of the same type (called the *element type*) stored in contiguous memory locations.
>
> *Class*—A collection of named fields and methods that may access these fields. Classes are the central type in Java and are used for a variety of purposes, including grouping fields and methods, acting as a library-like container for methods, and controlling access to fields and methods.
>
> *Interface*—A collection of method signatures. A **method signature** is the name, number and types of parameters, and return type of a method. Interfaces are discussed in more detail in the next chapter.

These built-in structured types can be combined to make more complex structured types. For example, an array can have elements that are classes some whose fields are classes and others of which are arrays. Thus programmers can build arbitrarily complicated structured types.

An important distinction between types in a language is whether a type is value or a reference type.

> **Value type**: A type whose variables hold data structures representing the values of the carrier set of the type.
>
> **Reference type**: A type whose variables hold references to locations where data structures representing the values of the carrier set of the type are stored.

To illustrate this distinction, note that in Java the `byte` type is a value type and array types are reference types. If `v` is a `byte` variable storing the value five, for example, then the memory location associated with `v` holds the representation of five (usually 00000101). If

`r` is a `byte` array variable holding a ten-`byte` array, then the memory location associated with `r` holds a reference to another memory location where the elements and other data of the array is stored. This has important consequences for assignments, parameter passing, and comparisons. For example, the assignment `byte x = v` will copy the value five from `v` into `x`, and further modifications of `x` or `v` will not affect the other. In contrast, the assignment `byte[] b = r` will copy the reference to the `byte` array from `r` into `b`. Both `r` and `b` will thus hold a reference to a *single* `byte` array. Changes to the byte array through `r` or `b` (such as `b[0] = 5`) will be reflected in the other (so `r[0]` will be five). In such cases we say that `r` and `b` are *aliases*.

In Java all primitive types are value types and all structured types are reference types (many languages do not make such a clean distinction).

Java's structured types, and in particular Java classes, are central in implementing ADTs in Java, and the next chapter is devoted to discussing this topic. The remainder of this chapter explores Java arrays, characters, and strings in greater detail.

2.4 ARRAYS

A structured type of fundamental importance in almost every imperative programming language is the array.

> **Array**: A fixed length, ordered collection of values of the same type stored in contiguous memory locations; the collection may be ordered in several dimensions.

The values stored in an array are called **elements**. Elements are accessed by *indexing* into the array: an integer value is used to indicate the ordinal value of the element. For example, if `a` is an array with 20 elements, then `a[6]` is the element of `a` with ordinal value 6. Indexing may start at any number, but generally (and in Java in particular), it starts at 0. Thus `a[6]` is the seventh value in `a` when indexing start at 0.

Arrays are important because they allow many values to be stored in a single data structure while providing very fast access to each value. This is made possible by the fact that (a) all values in an array are the same type, and hence require the same amount of memory to store, and (b) elements are stored in contiguous memory locations. Accessing element `a[i]` requires finding the location where the element is stored. This is done by computing $b + (i \times m,)$ where m is the size of an array element, and b is the base address of array `a`. This computation is very fast. Furthermore, access to all the elements of the array can be done by starting a counter at b and incrementing it by m, thus yielding the location of each element in turn, which is also very fast.

Arrays are not abstract data types because their arrangement in the physical memory of a computer is an essential feature of their definition, while abstract data types abstract from all details of implementation on a computer. Nonetheless, we can discuss arrays in a "semi-abstract" fashion that ignores some implementation details. The definition above omits details about how elements are stored in contiguous locations (which indeed does vary somewhat among languages). Also, arrays are typically types in procedural programming languages, so they are treated like realizations of abstract data types even though they are really not. In this book, we treat arrays as implementation mechanisms and not as ADTs.

In some languages, the size of an array must be established once and for all when storage for the array is allocated and cannot change thereafter. Such arrays are called **fixed** or **static arrays**. A chunk of memory big enough to hold all the values in the array is allocated when the array is created, and thereafter elements are accessed using the fixed base location of the array. Static arrays are the fundamental array type in most older procedural languages, such as Fortran, Basic, and C, and in many newer object-oriented languages as well, including Java.

Some languages provide arrays whose sizes are established at run-time and can change during execution. These **dynamic arrays** have an initial size used as the basis for allocating a segment of memory for element storage. Thereafter the array may shrink or grow. If the array shrinks during execution, then only an initial portion of allocated memory is used. But if the array grows beyond the space allocated for it, a more complex *reallocation procedure* must occur, as follows:

1. A new segment of memory large enough to store the elements of the expanded array is allocated.
2. All elements of the original (unexpanded) array are copied into the new memory segment.
3. The memory used initially to store array values is freed and the newly allocated memory is associated with the array variable or reference.

This reallocation procedure is computationally expensive, so systems are usually designed to do it as infrequently as possible. For example, when an array expands beyond its memory allocation, its memory allocation might be doubled even if space for only a single additional element is requested. We will see in a later chapter that this is a wise strategy.

Dynamic arrays are convenient for programmers because they can never be too small—whenever more space is needed in a dynamic array, it can simply be expanded. One drawback of dynamic arrays is that implementing language support for them is more work for the compiler or interpreter writer. A potentially more serious drawback is that the expansion procedure is expensive, so there are circumstances when using a dynamic array can be dangerous. For example, if an application must respond in real time to events in its environment, and a dynamic array must be expanded when the application is in the midst of a response, then the response may be delayed too long, causing problems.

As mentioned, Java has static arrays. However, much of the convenience of dynamic arrays is achieved with the library class `ArrayList`. An `ArrayList` stores data internally in fixed-size arrays, but presents an interface to its clients (that is, programmers who use `ArrayList`) that makes an `ArrayList` work as if it were a dynamic array. In particular, an `ArrayList` can grow or shrink as a programmer directs, or grow automatically as elements are added. This requires that each `ArrayList` manage its memory as indicated above, allocating new fixed arrays when the capacity of the `ArrayList` grows.

The `ArrayList` class illustrates how a data type can be added to a language and by so doing provide useful features not built into the language. The `ArrayList` class is widely used because it provides such useful features but is still reliable and relatively efficient. The ability to write reliable and efficient classes like this is one of the skills be will develop as we learn about data structures and algorithms.

2.5 JAVA CHARACTERS AND STRINGS

Characters and strings in Java provide an interesting illustration of the relationships between abstract data types and their implementations using data structures and algorithms; we will here be focussing in particular on the way ADT carrier sets are realized in data structures.

The *character* ADT has a set of symbols as its carrier set and operations for classifying symbols (for example, as white space characters or as digits) and mapping them from one to another (for example, an operation to change a letter to uppercase) in its method set. The character ADT implemented in Java was originally intended to have as its carrier set the set of all symbols in the world's languages, plus mathematical symbols and various decorative symbols as described in the *Unicode Standard*. This standard is an international agreement about representing symbols in computers that is essentially an assignment of a number, called a *Unicode code point*, to every symbol included in the standard (see www.unicode.org).

When Java was originally released in 1995, the Unicode Standard included only characters from modern written languages, which numbered less than 2^{14}. The Java `char` type was designed to use a two byte (16-bit) representation for Unicode characters, which was more than sufficient at the time. The `String` class also used this representation, and it was an easy matter to incorporate `char` values into `String` objects. However, in 1996 the Unicode consortium decided to expand Unicode to include rarely used and historic characters. Currently the Unicode standard specifies over 120,000 characters and code points. Unicode code points range from 0 to 1,114,111. This latter number is 10FFFF in hexadecimal, so it requires 18 bits when represented in base two.

This change presented a challenge to the designers of Java. A 16-bit format was no longer adequate to represent all Unicode characters, so either the goal of handling all Unicode characters in Java had to be abandoned, or the language would have to be modified. The latter alternative presents serious problems. The `char` type is specified in the language as having a 16-bit representation, and Java Unicode escape characters are specified as consisting of `\u` followed by four hexadecimal characters (specifying 16 bits of data). Changing these specifications would break many existing Java programs. On the other hand, the `String` and `Character` classes are defined by their interfaces, that is, the constants and methods that they provide. The hidden implementation of Java strings could be changed, and these classes could be supplemented with new methods.

In the end, the Java designers decided to keep the specification of the `char` type and Unicode escape sequences unchanged, so the `char` carrier set does not include all Unicode characters. However, the designers were able to allow string constants to contain all Unicode characters by designating characters with code points beyond 2^{16} with two Unicode escapes in a row (using the UTF-16 encoding scheme). The internal representations of strings in the

`String` class incorporated this change as well. This change to the `String` class allows its carrier set to include strings over all Unicode characters, avoiding the restriction that had to be made to the `char` type's carrier set.

Furthermore, methods were added to the `String` and `Character` classes to handle Unicode code points (as `int` values) and pairs of `char` values representing Unicode characters with code points larger than 2^{16}. These extra methods allow Java to process all Unicode strings, though with some degree of awkwardness. Thus Java can still handle strings composed of any Unicode characters, even though its `char` type cannot represent all Unicode characters.

This story illustrates several points relevant to data structures and algorithms. First, when making decisions about implementing data types or choosing algorithms, advantages and disadvantages must be traded-off against one another to arrive at a good solution. The Java language designers had to either give up being able to handle all Unicode characters, or accept a restriction in the `char` type; they chose the latter while mitigating the restriction by making other backwards compatible changes to the language and its libraries, though at the expense of a messier design.

Second, because Java provides the ability to separate the interface from the implementation of classes, and the library writers had taken advantage of this ability, it was possible to alter the implementations of the `Character` and `String` classes without disturbing all the software written using the old implementations. Separating interfaces from implementations is an important characteristic of well-written software that we will strive for in the code we produce in this book.

2.6 ADTS AND RECEIVERS

ADTs as we have defined them consist of sets of values (carrier sets) and sets of functions (method sets). This description matches Java's built-in simple types quite nicely. For example, the Java `int` type has as its carrier set the integer values between -2147483648 and 2147483647, and its method set includes the functions invoked by the standard operators +, *, -, /, and so on. For example, if `a` and `b` are `int` variables, then `a+b` applies the addition function to the values of `a` and `b` and returns the resulting `int` value, as one would expect.

The situation is not quite so obvious with Java's built-in reference types, a point we glossed over above. For examples, consider array types. The array type includes indexing and length functions in its method set. The indexing function takes an array and an index and returns the value stored in the array at that index. For example, suppose that `a` is an array of type `int[10]` and we wish to find that value at location `i`. The indexing operation has the special syntax `a[i]`, which, though odd, still has the appearance of a function applied to `a` and `i` to return as its result the i^{th} value of `a`. Now consider `a.length()`. This expression applies the length function to the array `a` and returns the number of elements in the array. But the length function takes an array as its argument, so shouldn't this expression be `length(a)`? In fact, it is, but generally in object-oriented languages the object to which a method is applied (which is always a parameter) is conventionally shown as the receiver of the method call; typically the dot operator is used to indicate the receiver. Java uses this notation for all reference types, not just classes, and hence we must use the expression `a.length()` to invoke the length function in the array method set.

Using the dot operator to identify a receiver, though it obscures an argument of method set functions, has an advantage. To illustrate, lets consider a very simple ADT and its Java implementation. The *cyclic n-counter* ADT has as its carrier set the integers from 0 to *n*-1. Its method set consists of the following functions.

set(*i*)—*i* is an integer and the result is the cyclic *n*-counter value (*i* % *n* + *n*) % *n*.

add(*c*, *i*)—*c* is a cyclic *n*-counter value (that is, a value between 0 and *n*-1), *i* is an integer, and the result is the cyclic *n*-counter value (*c* + *set*(*i*)) % *n*.

In other words, a cyclic *n*-counter counts from 0 to *n*-1 and then cycles back to 0. The function *set*() converts an integer to a cyclic *n*-counter value, and the function *add*() uses a cyclic *n*-counter to count, taking one as input and returning an another as output.

Notice that the *add*() function returns a cyclic *n*-counter value. If we implement this ADT as a class, then the cyclic *n*-counter argument to the `add()` method will be the object receiver of the method. Furthermore, we will modify this receiver, and so there is no reason to return it as the result of the method. Hence the implementation can simplify the ADT function to have (apparently) one fewer argument and no return value. Or, even better, the `add()` method can return the integer value of the cyclic *n*-counter after the addition, which clients may find convenient, thus improving the implementation. This is the advantage of using the receiver syntax: it apparently reduces the number of method arguments and often frees up a method return value for another purpose.

The box below shows a Java implementation of this ADT.

```
public class CyclicCounter {
 int count;   // current value of the counter
 final int n;  // modulus for this counter

 public CyclicCounter(int n) {
  this.n = n;
  count = 0;
 }

 public int set(int i) {
  count = (i % n + n) % n;
  return count;
 }

 public int add(int i) {
  count = ((count + i) % n + n) % n;
  return count;
 }

 public int value() { return count; }
}
```

Figure 1: An Implementation of the Cyclic n-Counter ADT

The constructor sets the limit of the cyclic *n*-counter (the modulus). Both the `set()` and `add()` methods return the integer value of the counter once it is set or modified, which again can be useful for clients of this class. Finally, this implementation includes a `value()` method, which may also be useful for clients.

To summarize from this simple example, we see that the ADT specification, which is stated in terms of sets of values and functions on those values, differs somewhat from the object-oriented implementation. This is because (a) object-oriented languages usually use a receiver syntax for methods, (b) methods often take advantage of receiver syntax to return a different value from the corresponding ADT function for reasons of convenience, and (c) additional methods may be added to the class, again for the convenience of clients.

Consequently, when we specify ADTs in future chapters, we will generally present them by describing their carrier sets but presenting their methods sets as functions with an implicit receiver parameter that can be changed and (implicitly) returned. This is done to lessen the burden on the reader of converting from sets of functions on carrier set values to methods of a class. To illustrate, consider the redefined cyclic *n*-counter method set below, which assumes an implicit, changeable cyclic *n*-counter supplied to each function.

set(*i*)—assign the cyclic *n*-counter the value (*i* % *n* + *n*) % *n*. The return value is the cyclic *n*-counter value as an integer.

add(*i*)—modify the cyclic *n*-counter by adding *i* to it, and return the resulting value as an integer; more precisely, if *c* is the implicit cyclic *n*-counter, then after this function is called the value of *c* is ((*c* + *i*) % *n* + *n*) % *n*, which is also returned as an integer value.

Note that the functions in these **implicit-receiver method** sets (as we will call them) more closely correspond to the methods in interfaces and classes implementing them. However, you should understand that ADTs really are carrier sets and methods sets as we have defined them earlier, and we can always convert from implicit-receiver method sets to plain method sets if we need to.

2.7 REVIEW QUESTIONS

1. What is the difference between a simple and a structured type?
2. List the built-in simple and structured types of Java.
3. Explain the difference between fixed and dynamic arrays.
4. Explain the difference between value and reference types.
5. Explain why the Java `char` type does not include all Unicode characters.
6. What advantages do implicit receivers provide in defining functions?

2.8 EXERCISES

1. Choose a language besides Java and list its simple and structured types.
2. Java was created as a successor to the C++ programming language; compare the simple and structured types of Java and C++.
3. Find a language with a simple type that is a reference type, and language with a structured type that is a value type. Name the languages and the types in these categories.
4. Write a small Java program to illustrate how reference types differ from value types with respect to parameter passing.
5. Write a small Java program to illustrate how reference types differ from value types with respect to equality testing.
6. Write a Java program to print the Unicode characters between 0 and 1023. Each line should display the value as a Unicode code point in the format U+*hhhh* (where *h* is a hexadecimal symbol), followed by the character as a glyph. Hint: investigate `printf()` in the `PrintStream` class.

7. The YFI ADT records non-negative distances in yards, feet, and inches, to the nearest inch. The carrier set of the YFI ADT is the set of all triples (*y*, *f*, *i*) where *y*, *f*, and *i* are natural numbers such that $f < 3$ and $i < 12$. The YFI ADT has the following functions in its method set.
   set(*y*, *f*, *i*)—Return a YFI value representing a distance of *y* yards, *f* feet, and *i* inches. If $36*y + 12*f + i < 0$, then the result of this function is undefined.
   m + *n*—Return a new YFI value that represents a distance that is the sum of the distances represented by *m* and *n*.
   m == *n*—Return true iff *m* and *n* represent the same distance.
   Implement the YFI ADT as the Java class `YFI`. Record distances as inches. Implement the + function as the method `add(YFI n)` that adds `n` to the receiver object. Implement the == function as the method `equals(FYI n)`. Finally, implement `getYards()`, `getFeet()`, and `getInches()` methods such that if y == m.getYards(),
   `f == m.getFeet()`, and `i = m.getInches()`, then `f` < 3, `i` < 12, and `m.equals(n.set(y,f,i))`.

2.9 REVIEW QUESTION ANSWERS

1. The values of a simple type cannot be divided into parts, while the values of a structured type can be. For example, the values of the Java `int` type cannot be broken into parts, while the values of a Java array can be (the parts are the elements of the array).
2. The built-in simple types of Java are `byte`, `char`, `short`, `int`, `long`, `float`, `double`, and `boolean`. The built-in structured types of Java are arrays, classes, and interfaces.
3. A fixed array is an array whose size is determined when the array is created and cannot be changed thereafter. A dynamic array is an array whose size can be changed at any time during execution. Java has fixed arrays.
4. A value type is a type whose values are stored in memory locations associated with variables of the type. For example, the Java `boolean` type is a value type. If `b` is a `boolean` variable, then the memory location associated with `b` stores the actual rune value (`true` of `false`). A reference type is a type whose values are stored in some memory location not associated with variables of the type; memory locations associated with variables of the type instead store references (that is, addresses) of the memory location where the data is stored. For example, in Java the `String` type is a reference type because it is a class type. If `s` is a `String` variable, then the memory location associated with `s` stores a reference to the memory location where and data for an instance of the `String` class is stored.

5. The Java `char` type by definition contains a 16-bit representation of a Unicode character. However, there are more than 2^{16} Unicode characters, so `char` type values cannot possibly represent all Unicode characters. The Java `char` type values are the Unicode characters with code points between 0 and 2^{16}. This set includes the vast majority of characters in everyday use.
6. Functions with implicit receivers no longer need to have an explicit receiver argument, which simplifies the function's parameters. Furthermore, the function need not return a modified receiver value, allowing the result of the function to be some other value, which is potentially another advantage. Finally, defining functions this way more closely reflects the implementation of abstract data types in object-oriented programming languages, making it easier to see the connection between ADTs and code to implement them.

3 CONTAINERS

3.1 INTRODUCTION

Simple abstract data types are useful for manipulating simple sets of values, like Integers or Booleans, but more complex abstract data types are crucial for most applications. A category of complex ADTs that has proven particularly important are containers.

> **Container**: An entity that holds finitely many other entities.

Just as containers like boxes, baskets, bags, pails, cans, drawers, and so forth are important in everyday life, containers such as lists, stacks, and queues are important in programming.

3.2 VARIETIES OF CONTAINERS

Various containers have become standard in programming over the years; these are distinguished by three properties:

> *Structure*—Some containers hold elements in some sort of structure, and some do not. Containers with no structure include sets and bags. Containers with linear structure include stacks, queues, and lists. Containers with more complex structures include multidimensional matrices.
>
> *Access Restrictions*—Structured containers with access restrictions only allow clients to add, remove, and examine elements at certain locations in their structure. For example, a stack only allows element addition, removal, and examination at one end, while lists allow access at any point. A container that allows client access to all its elements is called **traversable**, **enumerable**, or **iterable**
>
> *Keyed Access*—A collection may allow its elements to be accessed by keys. For example, maps are unordered containers that allows their elements to be accessed using keys.

3.3 A CONTAINER TAXONOMY

It is useful to place containers in a taxonomy to help understand their relationships to one another and as a basis for implementation using a class hierarchy. The root of the taxonomy is `Container`. A `Container` may be structured or not, so it cannot make

assumptions about element location (for example, there may not be a first or last element in the container). A `Container` may or may not be accessible by keys, so it cannot make assumptions about element retrieval methods (for example, it cannot have a key-based search method). Finally, a `Container` may or may not have access restrictions, so it cannot have addition and removal operations (for example, only stacks have a `push()` operation).

The only things we can say about `Containers` is that they have some number of elements. Thus a `Container` can have a `size()` operation. We can also ask (somewhat redundantly) whether a `Container` is empty, so there should be an `isEmpty()` operation. And although a `Container` cannot have specific addition and removal operations, it can have an operation for emptying it completely, which we call `clear()`.

A `Container` is a broad category whose instances are all more specific things; there is never anything that is just a `Container`. This is perhaps most evident in the fact that a `Container` is characterized by its operations alone. In object-oriented terms, a `Container` is an interface, not a class.

UML, the *Unified Modeling Language*, is a notation developed for object-oriented modeling. In UML, the icon for an interface is a compartmentalized rectangle with the interface name in the top compartment, along with the stereotype keyword *interface* placed between guillemet symbols. The operations in the interface in the second compartment. A UML diagram for the `Container` interface is shown in Figure 1 below.

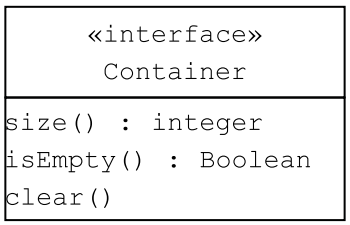


Figure 1: The `Container` Interface

There are many ways that we could construct our container taxonomy from here; one way that works well is to make a fundamental distinction between traversable and non-traversable containers:

Collection: A traversable container.

Dispenser: A non-traversable container.

`Collections` include lists, sets, and maps; dispensers include stacks and queues. With this additiown, our container hierarchy appears in Figure 2.

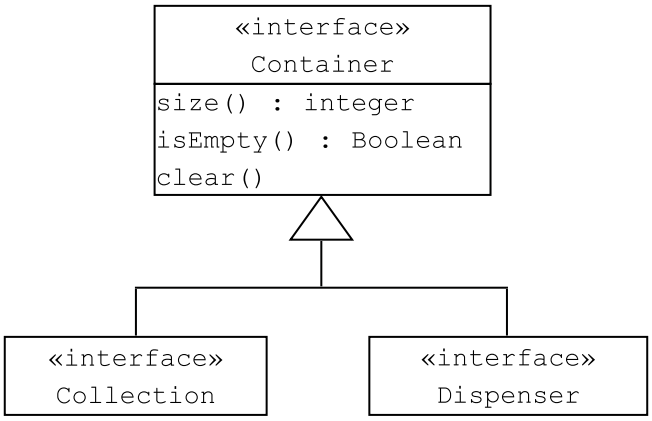


Figure 2: Top of the Container Hierarchy

In UML the sub-interface symbol is a hollow rectangle, so this diagram says that the `Collection` and `Dispenser` interfaces are sub-interfaces of `Container`. We will late consider what operations, if any, belong in the `Collection` and `Dispenser` sub-interfaces, so for now we leave the method compartment out of the icons for these interfaces in our diagram.

`Dispensers` are linearly ordered and have access restrictions. As noted, `Dispensers` include stacks and queues. We turn in the next chapter to detailed consideration of non-traversable containers.

3.4 A JAVA CONTAINER HIERARCHY

One of the best ways to understand various sorts of data structures is to use them to implement a simple container hierarchy. This is also a good way to improve your programming skills. Consequently we ask you to implement a container hierarchy in Java. Of course, Java already has a rich set of containers in its library, called the collection classes. We have no intention of replacing this hierarchy with our own. Rather, the hope is that by writing a simple container hierarchy, you will better understand the Java collection classes (and the containers you will find in the libraries of other languages) and how to use them effectively.

You should place your container hierarchy in a package called `containers`. You can also put tests and other material in this package since you will not be distributing it to anyone. We will provide some code for you to use as we go along, but you will write most of it yourself. If you like, you can look at Java library class source code, or find code on the web, but you will learn best if you use other code as a model or a guide rather than simply cutting and pasting it without understanding how it works. Again, the goal is not to write a commercial-grade container library, but to understand how data structures work.

3.5 REVIEW QUESTIONS

1. Are sets structured? Do sets have access restrictions? Do sets have keyed access?
2. If `c` is a `Container` and `c.clear()` is called, what does `c.isEmpty()` return? What does `c.size()` return?

3.6 EXERCISES

1. Consider a kind of `Container` called a `Log` that is an archive for summaries of transactions. Summaries can be added to the end of a `Log`, but once appended, they cannot be deleted or changed. When a summary is appended to a `Log`, it is time-stamped, and summaries can be retrieved from a `Log` by their time stamps. The summaries in a `Log` can also be examined in arbitrary order.
   a) Is a `Log` structured? If so, what kind of structure does a `Log` have?
   b) Does a `Log` have access restrictions?
   c) Does a `Log` provide keyed access? If so, what is the key?
   d) In the container hierarchy, would a `Log` be a `Collection` or a `Dispenser`?

2. Consider a kind of `Container` called a `Shoe` used in an automated Baccarat program. When a `Shoe` instance is created, it contains eight decks of `Cards` in random order. `Cards` can be removed one at a time from the front of a `Shoe`. `Cards` cannot be placed in a `Shoe`, modified, or removed from any other spot. No `Cards` in a `Shoe` can be examined.
   a) Is a `Shoe` structured? If so, what kind of structure does a `Shoe` have?
   b) Does a `Shoe` have access restrictions?
   c) Does a `Shoe` provide keyed access? If so, what is the key?
   d) In the container hierarchy, would a `Shoe` be a `Collection` or a `Dispenser`?

3. Consider a kind of `Container` called a `Randomizer` used to route packets in an anonymizer. Packets go into the `Randomizer` at a single input port, and come out randomly at one of *n* output ports, each of which sends packets to different routers. Packets can only go into a `Randomizer` at the single input port, and can only come out one of the *n* output ports. Packets come out of a single output port in the order they enter a `Randomizer`. Packets cannot be accessed when they are inside a `Randomizer`.
   a) Is a `Randomizer` structured? If so, what kind of structure does a `Randomizer` have?
   b) Does a `Randomizer` have access restrictions?
   c) Does a `Randomizer` provide keyed access? If so, what is the key?
   d) In the container hierarchy, would a `Randomizer` be a `Collection` or a `Dispenser`?

4. You will implement the `Container` hierarchy in Java as we progress through the book. Begin this task by making a Java package called `container`. Inside this package write a `Container` interface in accord with the UML diagram in Figure 1.

3.7 REVIEW QUESTION ANSWERS

1. Sets are not structured—elements appear in sets or not, they do not have a position or location in the set. Sets do not have access restrictions: elements can be added or removed arbitrarily. Elements in sets do not have keys (they are simply values), so there is no keyed access to elements of a set.
2. When a `Container c` is cleared, it contains no values, so `c.isEmpty()` returns true, and `c.size()` returns 0.

4 ASSERTIONS

4.1 INTRODUCTION

At each point in a program, there are usually constraints on the computational state that must hold for the program to be correct. For example, if a certain variable is supposed to record a count of how many changes have been made to a file, this variable should never be negative. It helps human readers to know about these constraints. Furthermore, if a program checks these constraints as it executes, it may find errors almost as soon as they occur. For both these reasons, it is advisable to record constraints about program state in assertions.

> **Assertion**: A statement that must be true at a designated point in a program.

4.2 TYPES OF ASSERTIONS

There are three sorts of assertions that are particularly useful:

> *Preconditions*—A **precondition** is an assertion that must be true at the initiation of an operation. For example, a square root operation cannot accept a negative argument, so a precondition of this operation is that its argument be non-negative. Preconditions most often specify restrictions on parameters, but they may also specify that other conditions have been established, such as a file having been created or a device having been initialized. Often an operation has no preconditions, meaning that it can be executed under any circumstances.
>
> *Post conditions*—A **post condition** is an assertion that must be true at the completion of an operation. For example, a post condition of the square root operation is that its result, when squared, is within a small amount of its argument. Post conditions usually specify relationships between the arguments and the result, or restrictions on the results. Sometimes they may specify that the arguments do not change, or that they change in certain ways. Finally, a post condition may specify what happens when a precondition is violated (for example, that an exception will be thrown).
>
> *Class invariants*—A **class invariant** is an assertion that must be true of any class instance before and after calls of its exported operations. Usually class invariants specify properties of fields and relationships between the fields in a class. For example, suppose a Bin class models containers of discrete items, like apples or nails. The `Bin` class might have `currentSize`, `spaceLeft`, and `capacity` fields. One of its class invariants is that `currentSize` and `spaceLeft` must always be between zero and `capacity`; another is that `currentSize` + `spaceLeft` = `capacity`.

A class invariant may not be true *during* execution of a public method, but it must be true *between* executions of public methods. For example, a method to add something to a container must increase the `currentSize` and decrease the `spaceLeft` fields, and for a moment during execution of this operation their sum might not be correct, but when the method is done, their sum must be the `capacity` of the container.

Other sorts of assertions may be used in various circumstances. An **unreachable code** assertion is an assertion that is placed at a point in a program that should not be executed under any circumstances. For example, the cases in a switch statement often exhaust the possible values of the switch expression, so execution should never reach the default case. An unreachable code assertion can be placed at the default case; if it is ever executed, then the program is in an erroneous state. A **loop invariant** is an assertion that must be true at the start of a loop on each of its iterations. Loop invariants are used to prove program correctness. They can also help programmers write loops correctly, and understand loops that someone else has written.

4.3 ASSERTIONS AND ABSTRACT DATA TYPES

Although we have defined assertions in terms of programs, the notion can be extended to abstract data types (which are mathematical entities). An **ADT assertion** is a statement that must be true of the carrier set values or the method set operations of the ADT. ADT assertions can describe many things about an ADT, but usually they help describe the operations of the ADT. Especially helpful in this regard are operation **preconditions**, which usually constrain the parameters of operations, operation **post conditions**, which define the results of the operations, and **axioms**, which make statements about the properties of operations, often showing how operations are related to one another. For example, consider the Natural ADT whose carrier set is the set of non-negative integers and whose operations are the usual arithmetic operations. A precondition of the mod (%) operation is that the modulus not be zero; if it is zero, the result of the operation is undefined. A post condition of the mod operation is that its result is between zero and one less than the modulus. An axiom of this ADT is that for all natural numbers a, b, and $m > 0$,

$$(a+b) \% m = ((a \% m) + b) \% m$$

This axiom shows how the addition and mod operations are related.

We will often use ADT assertions, and especially preconditions, in specifying ADTs. Usually, ADT assertions translate into assertions about the data types that implement the ADTs, which helps insure that our ADT implementations are correct.

4.4 USING ASSERTIONS

As a rule, when writing code, programmers should state pre- and subtle post conditions for public methods, state class invariants, and insert unreachable code assertions and loop invariants wherever appropriate.

Some languages have facilities to support assertions directly and some do not. If a language does not directly support assertions, then the programmer can mimic their effect. For example, the first statements in a method can test the preconditions of the method and throw an exception if they are violated. Post conditions can be checked in a similar manner. Class invariants are more awkward to check because code for them must be inserted at the start and end of every public method. This is usually not practical. Unreachable code assertions occur relatively infrequently and are easy to insert, so they should always be used. Loop invariants are mainly for documenting and proving code, so they can be stated in comments at the tops of loops.

Often efficiency issues arise. For example, the precondition of a binary search is that the array searched is sorted, but checking this precondition is so expensive that one would be better off using a sequential search. Similar problems often occur with post conditions.

Hence many assertions are stated in comments and are not checked in code, or are checked during development and then removed or disabled when the code is compiled for release.

Languages that support assertions often provide different levels of support. For example, Java has an `assert` statement that takes a `boolean` argument; it throws an exception if the argument is not true. Assertion checking can be turned off with a compiler switch. Programmers can use the `assert` statement to write checks for pre- and post conditions, class invariants, and unreachable code, but this is all up to the programmer..

The languages Eiffel and D provide constructs in the language for invariants and pre- and post conditions that are compiled into the code and are propagated down the inheritance hierarchy. Thus Eiffel and D make it easy to incorporate these checks into programs. A compiler switch can disable assertion checking for released programs..

4.5 ASSERTIONS IN JAVA

Java provides good support for assertions. It has an `assert` statement, and exceptions can also be used as well.

The `assert` statement has two forms, as follows.

```
assert <booleanExpression>
 assert <booleanExpression> : <messageExpression>
```

In the first form, the `<booleanExpression>` is evaluated, and if `false`, an `AssertionError` is thrown. Programs should not try to catch these errors, so as a rule this will cause the program to fail and print a message and a stack trace. The second form of the statement does the same as the first, except that the `<messageExpression>` is evaluated and this value is printed as part of the error message when the program fails.

Java `assert` statements are disabled by default at runtime, which means they are treated as if they where not there. They are activated using the `-ea` or `-enableassertions` switch on the `java` interpreter.

The `assert` statement is used to terminate programs when some unexpected situation arises. This is appropriate in some cases, such as when unreachable code is executed, a post condition is violated, or a precondition of a private method is violated. But in other cases, it is better to throw an exception. This is especially true for preconditions of public methods. Java has many built-in exceptions that can be thrown in such cases (such as `NullPointerException`, `IllegalArgumentException`, or `IllegalStateException`). Programmers can

define their own exceptions as well. Throwing exceptions instead of using `assert` statements allows clients to catch and handle calls that violate preconditions, providing more flexibility.

As noted before, many assertions are too expensive to check during execution and should instead be noted in comments. This practice should be followed in Java programs as well. *Javadoc* is program that comes with the Java Software Development Kit (SDK). It generates HTML/CSS documentation from Java source code. Because pre- and post conditions often help clients understand how to use methods, pre- and post conditions should be part of the Javadoc comments for public methods. Programmers associate Javadoc comments with methods by preceding a method declaration with a Javadoc comment that begins `/**` at the start of a line. Within this comment, special Javadoc *tags* have special meaning. Each `@param` tag is followed by a parameter name and a short description of the parameter. This description can state parameter preconditions. A `@return` tag is followed by a description of the result of the method (if any); it can state post conditions. Finally, a `@throws` tag is followed by the name of an exception class and a description. This tag can be used to document what happens when an assertion fails.

To illustrate, consider the Java code fragment below.

```
/**
 * Compute a square root.
 *
 * @param r value whose square is found; r must be at least 0
 * @result the square root, which will be at least 0
 * @throws IllegalArgumentException if r is less than 0
 */
public double sqrt(double r) { ... }
```

Note that the precondition that `r` must be non-negative is stated in the `@parameter` description, the post condition that the result must be non-negative is stated in the `@result` description, and the result of violating the precondition is stated in the `@throws` description.

Other assertions not checked in code, such as class or loop invariants, can easily be noted in comments at the top of the class or loop they describe.

In summary then, when using assertions in Java,

- Document public method pre- and post conditions in Javadoc comments;
- Only code assertions that are computationally feasible and note the rest in comments;
- Throw exceptions when pre-conditions of public methods are violated;
- Use the Java `assert` statement for post conditions, unreachable code assertions, and non-public method pre-conditions.

4.6 REVIEW QUESTIONS

1. Name and define in your own words three kinds of assertions.
2. What is an axiom?
3. How can programmers check preconditions of an operation in a language that does not support assertions?
4. In Java, should a class invariant violation raise an exception or use the `assert` statement?

4.7 EXERCISES

1. Consider the Integer ADT with the method set { +, -, *, div, mod, = }. Write preconditions for those methods that need them, post conditions for all methods, and at least four axioms.
2. Consider the Real ADT with the method set { +, -, *, /, $\sqrt[n]{x}$, x^n }, where x is a real number and n is an integer. Write preconditions for those methods that need them, post conditions for all methods, and at least four axioms.

Consider the following fragment of a class declaration in Java.

```
public class Storage {
 public final int NUM_LOCKERS = 138;
 private boolean[] lockerIsRented;
 private int numLockersAvailable;

 /**
  * Find an empty locker; mark it rented; return its number.
  */
 public int rentLocker() { ... }

 /** Mark a locker as no longer rented.
  */
 public void releaseLocker(int lockerNumber) { ... }

 /** Say whether a locker is for rent.
  */
 public boolean isFree(int lockerNumber) { ... }

 /** Say whether any lockers are left to rent.
  */
 public boolean isFull() { ... }
}
```

This class keeps track of the lockers in a storage facility at an airport. Lockers have numbers that range from 0 to 137. The Boolean array keeps track of whether a locker is rented. Use this class for the following exercises.

3. Write a class invariant for the `Storage` class.
4. Write preconditions in Java using Javadoc tags for all the methods that need them in the `Storage` class.
5. Write post conditions in Java using Javadoc tags for all methods that need them in the `Storage` class.
6. Write code to check preconditions for all the methods in the `Storage` class that need them.
7. Implement the methods in `Storage` class in Java.
8. Augment your implementation with `assert` statements that check class invariants for the `Storage` class.

4.8 REVIEW QUESTION ANSWERS

1. A precondition is an assertion that must be true when a method begins to execute. A post condition is an assertion that must be true when a method completes execution. A class invariant in an assertion that must be true between executions of the methods that a class makes available to clients. An unreachable code assertion is an assertion stating that execution should never reach the place where it occurs. A loop invariant is an assertion true whenever execution reaches the top of the loop where it occurs.
2. An axiom is a statement about the operations of an abstract data type that must always be true. For example, in the Integer ADT, it must be true that for all Integers n, $n * 1 = n$, and $n + 0 = n$, in other words, that 1 is the multiplicative identity and 0 is the additive identity in this ADT.
3. Preconditions can be checked in a language that does not support assertions by using conditionals to check the preconditions, and then throwing an exception, returning an error code, calling a special error or exception operation to deal with the precondition violation, or halting execution of the program when preconditions are not met.
4. Often it is not practical to check class invariants because this would need to be done at the end of every public method. However, if we did do this, then we would usually be checking relationships between (private) class fields. The private fields of a class and their relationships are implementation details hidden from clients, so no exception we raise regarding them could make sense to clients, or be something that clients could handle. Consequently it only make sense to raise an `AssertionError`, in other words, to use the `assert` statement, when checking class invariants.

5 STACKS

5.1 INTRODUCTION

Stacks have many physical metaphors: shirts in a drawer, plates in a plate holder, box-cars in a dead-end siding, and so forth. The essential features of a stack are that it is ordered and that access to it is restricted to one end.

> **Stack**: A dispenser holding a sequence of elements that can be accessed, inserted, or removed at only one end, called the **top**.

Stacks are also called last-in-first-out (LIFO) lists. Stacks are important in computing because of their applications in recursive processing, such as language parsing, expression evaluation, runtime function call management, and so forth.

5.2 THE STACK ADT AND INTERFACE

Stacks are containers, and as such they hold values of some type. We must therefore speak of the ADT *stack of T*, where *T* is the type of the elements held in the stack. The carrier set of this type is the set of all stacks holding elements of type *T*. The carrier set thus includes the empty stack, the stacks with one element of type *T*, the stacks with two elements of type *T*, and so forth. The stack of *T* ADT implicit-receiver method set is shown below, where *e* is a *T* value. If a function's precondition is not satisfied, its result is undefined.

> *push*(*e*)—Add *e* at the top of the stack.
>
> *pop*()—Remove and return the top element of the stack. The precondition of the *pop()* operation is that the stack is not empty.
>
> *isEmpty*()—Return the Boolean value true just in case the stack is empty.
>
> *top*()—Return the top element of the stack without removing it. Like *pop()*, this operation has the precondition that the stack is not empty.

The *pop*() operation undoes what the *push*() operation achieves. When an element is pushed on a stack, it becomes the top element on the stack. No element is accessible except the one at the top of the stack.

A `Stack` interface is a sub-interface of `Dispenser`, which is a sub-interface of `Container`, so it already contains an `isEmpty()` operation that it has inherited from `Container`. The `Stack` interface need only add methods for pushing elements, popping elements, and peeking at the top element of the stack. The diagram in Figure 1 shows the `Stack` interface.

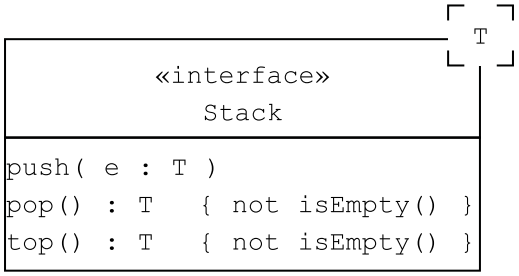


Figure 1: The Stack Interface

Note that a generic or type parameter is used to generalize the interface for any element type, shown in UML as a dashed box in the upper right-hand corner of the interface icon. Note also that the preconditions of methods that need them are shown as UML properties enclosed in curly brackets to the right of the operation signatures.

The UML specification of the `Stack` interface can be coded in Java quite directly. To illustrate, consider the code in Figure 2 below.

```
public interface Stack<T> extends Container {
 /**
  * Add a value to the top of the stack.
  * @param item the value added
  */
 void push(T item);

 /**
  * Return the top value from the stack without removing it.
  * The stack cannot be empty.
  * @result the top value, which is not removed
  * @throws IllegalStateException if the stack is empty.
  */
 T top() throws IllegalStateException;

 /**
  * Remove and return the top value on the stack.
  * The stack cannot be empty.
  *
  * @result the top value, which is removed
  * @throws IllegalStateException if the stack is empty.
  */
 T pop() throws IllegalStateException;
}
```

Figure 2: Java Stack Interface

The code in Figure 2 shows how the type parameter in UML becomes a generic interface parameter in Java. The preconditions stated in the UML diagram are reflected in the comments. The `throw` declarations result from precondition checking and are documented in Javadoc comments in accord with the policy we stated regarding precondition assertions.

5.4 AN EXAMPLE USING STACKS

When sending a document to a printer, one common option is to collate the output, in other words, to print the pages so that they come out in the proper order. Generally, this means printing the last page first, the next to last next, and so forth. Suppose a program sends several pages to a print spooler (a program that manages the input to a printer) with the instruction that they be collated. Assuming that the first page arrives at the print spooler first, the second next, and so forth, the print spooler must keep the pages in a container

until they all arrive, so that it can send them to the printer in reverse order. One way to do this is with a `Stack`. Consider the Java-like pseudocode in Figure 3 describing the activities of the print spooler.

```
printCollated( Job j ) {
 Stack(Page) stack;
 for each Page p in j stack.push(p);
 while ( !stack.isEmpty )
  printer.print( stack.pop() );
}
```

Figure 3: Using A Stack to Collate Pages for Printing

A `Stack` is the perfect container for this job because it naturally reverses the order of the data placed into it.

5.5 CONTIGUOUS IMPLEMENTATION OF THE STACK ADT

There are two approaches to implementing the carrier set for the stack ADT: a contiguous implementation using arrays, and a linked implementation using singly linked lists; we consider each in turn.

Implementing stacks of elements of type *T* using arrays requires a *T* array to hold the contents of the stack, and a marker to keep track of the top of the stack. The marker can record the location of the top element, or the location where the top element would go on the next `push()` operation. Which of these alternative a programmer chooses does not matter as long as the programmer is clear what the marker denotes and writes code accordingly.

If a static (fixed-size) array is used, then the stack can become full; if a dynamic (resizable) array is used, then the stack is essentially unbounded. Usually, resizing an array is an expensive operation because new space must be allocated, the contents of the old array copied to the new, and the old space deallocated, so this flexibility is acquired at a cost.

We will use a dynamic array called `store` to hold stack elements, and a marker called `count` to keep track of the top of the stack. Figure 4 below illustrates this data structure.

The marker is called `count` because it keeps track of the number of elements currently in the stack. When array indices begin at zero, this also happens to be the array location where the top element will go the next time `push()` is executed.

The `store` array holds `capacity` elements; this is the maximum number of elements that can be placed on the stack before it has to be expanded. The diagram shows two elements in the stack, designated by the cross-hatched array locations, and the value of the `count` variable. Note how the `count` variable indicates the array location where the next element will be stored when it is pushed onto the stack. The stack is empty when `count` is 0, and it must be expanded when `count` equals `capacity` and another element is pushed on the stack.

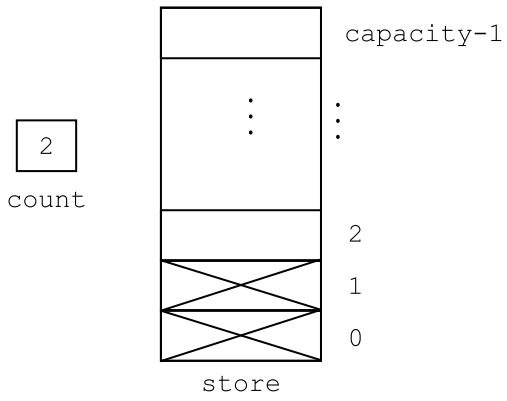


Figure 4: Implementing a Stack With an Array

Implementing the operations of the stack ADT using this data structure is straightforward. For example, to implement the `push()` operation, a check is first made to see whether the `store` must be expanded by testing whether `count` equals `capacity`. If so, then the `store` is expanded. Then the pushed value is assigned to location `store[count]`, and then `count` is incremented.

Lets call our Java class realizing this implementation `ArrayStack<T>`. We will not often display code that completely implements a container, but we do so in this case for two reasons: first, this is the first container we are considering, so it may help to see how to approach such a problem, and second, an `ArrayStack` provides a simple example of how to reallocate static arrays to provide a dynamic contiguous store Consequently the code for the `ArrayStack` type appears in Figure 5.

There are several things to note about the code in Figure 5. First, most comments have been removed to save space; normally we would include standard javadoc comments. Second, the Java generic mechanism has been employed to create a typed `ArrayStack` class. Third, there is a constructor that takes no parameter and constructs an empty `ArrayStack` with a default initial capacity of 8 elements, and a constructor that constructs an empty `ArrayStack` with a specified initial capacity of at least one. This is not part of the the `ArrayStack` specification, but it seems like a reasonable thing to do. We will often make (somewhat arbitrary) decisions like this for unspecified aspects of our implementations.

The implementations of the `top()` and `pop()` methods first check that the `ArrayStack` is not empty, and throw an `IllegalStateException` if they are. This accords with our policy of checking preconditions and indicating when they are violated.

The rest of the implementation is quite straightforward, except perhaps for the code in the `push()` method that expands the `store` array. At the start of the`push()` method, a check is made to see whether the current `store` array is full. If it is, then a new store is allocated that is twice the size of the existing store (we will discuss why this is a good idea in a later chapter). Then a for loop copies all the data from the old store to the new store, and finally the `store` variables is assigned the new store array, thus completing the reallocation procedure. The method finishes by adding `item` to the top of the stack (the end of `store`).

```
public class ArrayStack<T> implements Stack<T> {
 public static final int INITIAL_SIZE = 8;

 private T[] store;  // contents of the stack
 private int count;  // top is at store[count-1]

 public ArrayStack() { this(INITIAL_SIZE); }

 public ArrayStack(int initialSize) {
  if (initialSize < 1) initialSize = INITIAL_SIZE;
  store = (T[]) new Object[initialSize];
  count = 0;
 }

 public int size() { return count; }
 public boolean isEmpty() { return count == 0; }
 public void clear() { count = 0; }

 public void push(T item) {
  if (count == store.length) {
   Object[] newStore = new Object[2*store.length];
   for (int i = 0; i < store.length; i++) newStore[i] = store[i];
   store = (T[])newStore;
  }
  store[count++] = item;
 }

 public T top() throws IllegalStateException {
  if (count == 0)
   throw new IllegalStateException("top of an empty stack");
  return store[count-1];
 }

 public T pop() throws IllegalStateException {
  if (count == 0)
   throw new IllegalStateException("pop of an empty stack");
  return store[--count];
 }
}
```

Figure 5: Implementation of a Contiguous Stack

We could have used an `ArrayList` to implement an `ArrayStack` with less work, but it is useful to see how to manage static arrays to realize a dynamic array.

5.6 LINKED IMPLEMENTATION OF THE STACK ADT

A linked implementation of a stack ADT uses a linked data structure to represent values of the ADT carrier set. Lets consider the basics of linked data structures.

Node: An aggregate variable with data and link (pointer or reference) fields.

Linked (data) structure: A collection of nodes formed into a whole through its constituent node link fields.

Nodes may contain one or more data and link fields depending on the need, and the references may form a collection of nodes into linked data structures of arbitrary shapes and sizes. Among the most important linked data structures are the following.

Singly linked list: A linked data structure whose nodes each have a single link field used to form the nodes into a sequence. Each node link but the last contains a reference to the next node in the list; the link field of the last node contains null (a special reference value that does not refer to anything).

Doubly linked list: A linked structure whose nodes each have two link fields used to form the nodes into a sequence. Each node but the first has a predecessor link field containing a reference to the previous node in the list, and each node but the last has a successor link containing a reference to the next node in the list.

Linked tree: A linked structure whose nodes form a tree.

The linked structure for a stack is very simple, requiring only a singly linked list, so list nodes need only contain a value of type *T* and a reference to the next node. The top element of the stack is stored at the head of the list, so the only data that the stack data structure must keep track of is the reference to the head of the list. Figure 6 illustrates this data structure.

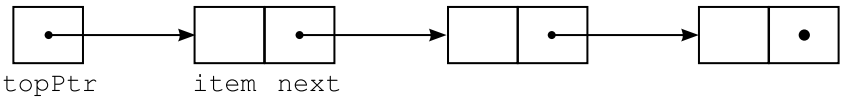


Figure 6: Implementing a Stack With a Singly Linked List

The reference to the head of the list is called `topPtr`. Each node has an `item` field holding a value of type *T* and a `next` field holding a link. The figure shows a stack with three elements; the top of the stack, of course, is the first node on the list. The stack is empty when `topPtr` is null; the stack never becomes full (unless memory is exhausted).

Again we show the entire implementation of a `LinkedStack` to demonstrate the similarities to and differences from an `ArrayStack`. We begin with code for the list nodes in Figure 7.

```
public class LinkedStack<T> implements Stack<T> {

...

 private class Node {
  T item;      // the value stored at this node
  Node next;   // link to the next node
  Node(T value, Node link) {
   item = value;
   next = link;
  }
 }
}
```

Figure 7: Singly Linked List Node Class

The `Node` class is a private inner class inside the `LinkedStack` class (the remainder of whose code is shown in Figure 8). The `Node` class has an `item` field for the value of generic type `T` stored in the node, and a field called `next`. The `next` field's type is `Node` because Java stores references to objects in object variables. Hence the value of `next` is either `null` or the node succeeding the current node in the list. This is how we can connect `Node` instances together to form a linked structure in Java. The constructor makes it easy to create new `Node` instances with their value and link fields set.

Now let us consider the rest of code in the `LinkedStack` class in Figure 8. Once again we remove comments to save space, and use the Java generic type facility to make a `LinkedStack` holding elements of type `T`.

Although the size of the stack can be computed from the linked list, it is convenient to have a `count` field to keep track of the size of the stack. The constructor makes an empty stack, setting the fields as expected. Also as before, the `pop()` and `top()` methods check that the stack is not empty and throw an `IllegalStateException` if it is. Note how a new node is added to the front of the list in the `push()` method and removed from the front of the list in the `pop()` method by rearranging references. You should draw pictures and execute this code by hand a few times to make sure you understand how it works.

As new nodes are needed when values are pushed onto the stack, new `Node` instances are instantiated and used in the list. When values are popped off the stack, `Node` instances are discarded and reclaimed by the system. Thus a `LinkedStack` uses only as much space as it needs for the elements it holds.

```
public class LinkedStack<T> implements Stack<T> {
 private int count;     // how many items in the stack
 private Node topPtr;  // head of the singly linked list

 public LinkedStack() {
  count = 0;
  topPtr = null;
 }

 public int size() { return count; }
 public boolean isEmpty() { return topPtr == null; }

 public void clear() {
  topPtr = null;
  count = 0;
 }

 public void push(T item) {
  topPtr = new Node(item, topPtr);
  count++;
 }

 public T top() throws IllegalStateException {
  if (topPtr == null)
   throw new IllegalStateException("top of an empty stack");
  return topPtr.item;
 }

 public T pop() throws IllegalStateException {
  if (topPtr == null)
   throw new IllegalStateException("pop of an empty stack");
  T result = topPtr.item;
  topPtr = topPtr.next;
  return result;
 }
}
```

Figure 8: Implementation of a Linked Stack

5.8 SUMMARY AND CONCLUSION

Both contiguous and linked implementations of stacks are simple and efficient, but the contiguous implementation either places a size restriction on the stack or uses an expensive reallocation technique if a stack grows too large. If contiguously implemented stacks are made extra large to make sure that they don't overflow, then space may be wasted.

A linked implementation is essentially unbounded, so the stack never becomes full. It is also very efficient in its use of space because it only allocates enough nodes to store the values actually kept in the stack. Overall, then, the linked implementation of stacks seems slightly better than the contiguous implementation.

5.8 REVIEW QUESTIONS

1. If the `ArrayStack` implementation did not reallocate space when the store array becomes full, what might happen? How could this be dealt with in the interface the `ArrayStack` class?
2. Should the `size()` operation from the `Container` interface return the capacity of a `Stack` or the number of elements currently in a `Stack`? What value should be returned by this operation in terms of the `ArrayStack` implementation?
3. The nodes in a `LinkedStack` hold a reference for every stack element, increasing the space needed to store data. Does this fact invalidate the claim that a `LinkedStack` uses space more efficiently than an `ArrayStack`?

5.9 EXERCISES

1. Restate the operations of the stack of *T* ADT so that they are mathematical functions from carrier set values to carrier set values (when a stack is changed) or from carrier set values to values of type *T* (when a stack is accessed).
2. Suppose that an `ArrayStack` is implemented so that the top elements is always stored at `store[0]`. What are the advantages or disadvantages of this approach?
3. How can a programmer who is us.ing an `ArrayStack` or a `LinkedStack` make sure that her code will not fail because it violates a precondition?
4. How could the `LinkedStack` implementation be altered to execute correctly without a `count` field?
5. State a class invariant relating the `count` and `topNode` attributes of a `LinkedStack`.
6. Could the top element be stored at the tail of a `LinkedStack` list? What consequences would this have for the implementation?

7. A `LinkedStack` could be implemented using a doubly-linked list. What are the advantages or disadvantages of this approach?
8. Write the `Stack<T>` interface in Java. Include Javadoc comments documenting appropriate assertions. Methods whose precondition is that the stack not be empty should be declared to throw an `IllegalStateException` when this precondition is violated.
9. Modify the `ArrayStack` implementation to use a Java `ArrayList`?

5.10 REVIEW QUESTION ANSWERS

1. If the `ArrayStack` implementation did not reallocate space when the store array becomes full, it might overflow the store array, meaning that the `push()` method could fail. This could be handled by adding a precondition to the `ArrayStack` `push()` method asserting that the stack is not full. Then the `push()` method could check this precondition and throw an exception if it was not true. It would probably be a good idea to add a method to `ArrayStack` allowing clients to determine whether it is safe to call `push();` an `isFull()` method would do the trick.
2. The `Container size()` operation, which is inherited by the `Stack` interface and must thus be implemented in all `Stacks`, should return the number of elements currently stored in a `Stack`. If a `Stack` has an arbitrary capacity (such as a `LinkedStack`), then the capacity of the `Stack` is not even well defined, so it would not make sense for the `size()` operation to return the capacity.
3. Each node in a `LinkedStack` contains both an element and a link, so a `LinkedStack` does use more space (perhaps twice as much space) as an `ArrayStack` to store a single element. On the other hand, an `ArrayStack` typically allocates more space than it uses at any given moment to store data—often there will be at least as many unused elements of the store array as there are used elements. This is because the `ArrayStack` must have enough capacity to accommodate the largest number of elements ever pushed on the stack, even when many elements have subsequently been popped from the stack. On balance, then, an `ArrayStack` will typically use more space than a `LinkedStack`.

6 QUEUES

6.1 INTRODUCTION

Queues are what we usually refer to as lines, as in "please get in line for a free lunch." The essential features of a queue are that it is ordered and that access to it is restricted to its ends: things can enter a queue only at the rear and leave the queue only at the front.

> **Queue**: A dispenser holding a sequence of elements that allows insertions only at one end, called the back or rear, and deletions and access to elements at the other end, called the front.

Queues are also called first-in-first-out, or FIFO, lists. Queues are important in computing because of the many cases where resources provide service on a first-come-first-served basis, such as jobs sent to a printer, or processes waiting for the CPU in an operating system.

6.2 THE QUEUE ADT AND INTERFACE

Queues are containers holding values of some type. We must therefore speak of the ADT *queue of T*, where *T* is the type of the elements held in the queue. The carrier set of this type is the set of all queues holding elements of type *T*. The carrier set thus includes the empty queue, the queues with one element of type *T*, the queues with two elements of type *T*, and so forth. The implicit-receiver method set of the type is the following, where *e* is a *T* value.

> *enter*(*e*)—Add *e* to the rear of the queue.
>
> *leave*()—Remove and return the front element of the queue. The precondition of the *leave()* operation is that the queue is not empty.
>
> *isEmpty*()—Return the Boolean value true just in case the queue is empty.
>
> *front*()—Return the front element of the queue without removing it. Like *leave*(), this operation has the precondition that the queue is not empty.

The `Queue` interface is a sub-interface of `Dispenser`, which is a sub-interface of `Container`, so it already contains an `isEmpty()` operation inherited from `Container`. The `Queue` interface need only add operations for entering elements, removing elements, and peeking at the front element of the queue. The diagram in Figure 1 shows the `Queue` interface.

As with stacks, a generic or template is used to generalize the interface for any element type, and that preconditions have been added for the operations that need them.

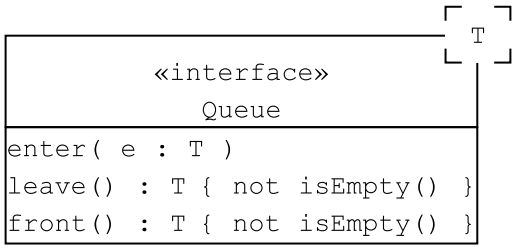


Figure 1: The Queue Interface

6.3 AN EXAMPLE USING QUEUES

When sending a document to a printer, it may have to be held until the printer finishes whatever job it is working on. Holding jobs until a printer is free is generally the responsibility of a print spooler (a program that manages the input to a printer). Print spoolers hold jobs in a `Queue` until the printer is free. This provides fair access to the printer and guarantees that no print job will be held forever. The pseudocode in Figure 2 describes the main activities of a print spooler.

```
Queue(Job) queue;

spool( Document d ) {
 queue.enter( new Job(d) );
}

run() {
 while ( true ) {
  if ( printer.isFree && !queue.isEmpty )
   printer.print( queue.leave );
 }
}
```

Figure 2: Using A Queue to Spool Pages for Printing

The print spooler has a job queue stored in the variable `queue`. A client can ask the spooler to print a document for it using the `spool()` operation and the spooler will add the document to its job `queue`. Meanwhile, the spooler is continuously checking the printer and its job `queue`; whenever the printer is free and there is a job in the `queue`, it will remove the job from the `queue` and send it to the printer.

6.5 CONTIGUOUS IMPLEMENTATION OF THE QUEUE ADT

There are two approaches to implementing the carrier set for the queue ADT: a contiguous implementation using arrays, and a linked implementation using singly linked lists; we consider each in turn.

Implementing queues of elements of type T using arrays requires a T array to hold the contents of the queue and some way to keep track of the front and the rear of the queue. We might, for example, decide that the front element of the queue would always be stored in an array at location 0, and record the `count` of the elements in the queue, implying that the rear element would be at location `count`-1. This approach requires that the data be moved forward in the array every time an element leaves, which is not very efficient.

A clever solution to this problem is to allow the data in the array to "float" upwards as elements enter and leave, and then wrap around to the start of the array when necessary. It is as if the locations in the array are in a circular rather than a linear arrangement. Figure 3 illustrates this solution. Queue elements are held in the `store` array. The variable `frontIndex` keeps track of the array location holding the element at the front of the queue, and `count` holds the number of elements in the queue. The `capacity` is the size of the array and hence the number of elements that can be stored in the queue.

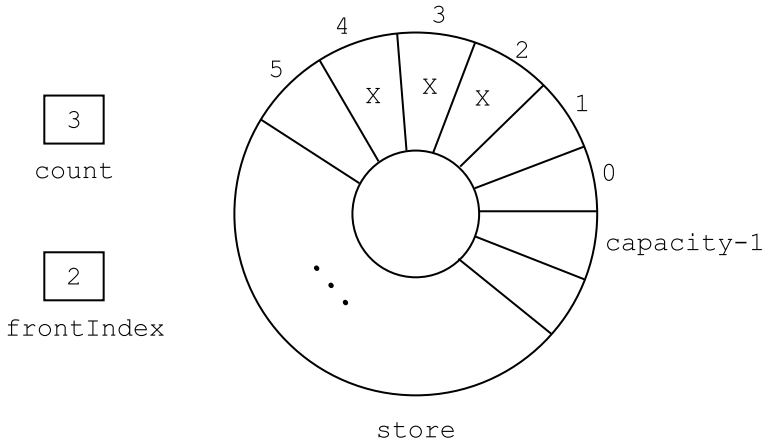


Figure 3: Implementing a Queue With a Circular Array

In Figure 3, data occupies the regions with an `X` in them: there are three elements in the queue, with the front element at `store[2]` (as indicated by the `frontIndex` variable) and the rear at `store[4]` (because `frontIndex+count-1` is 4). The next element entering the queue would be placed at `store[5]`; in general, elements enter the queue at

```
store[(frontIndex+count) % capacity]
```

The modular division is what makes the queue values wrap around the end of the array to its beginning. This trick of using a circular array is the standard approach to implementing queues in contiguous locations.

If a static array is used, then the queue can become full; if a dynamic array is used, then the queue is essentially unbounded. As we have mentioned before, resizing an array is an expensive operation because new space must be allocated, the contents of the array copied, and the old space deallocated, so this flexibility has a cost. Care must also be taken to move elements properly to the expanded array—remember that the front of the queue may be somewhere in the middle of the full array, with elements wrapping around to the front.

Implementing the operations of the queue ADT using this data structure is quite straightforward. For example, to implement the `leave()` operation, a check is first made that the precondition of the operation (that the queue is not empty) is not violated by testing whether `count` equals 0. If not, then the value at `store[frontIndex]` is saved in a temporary variable, `frontIndex` is set to `(frontIndex+1) % capacity`, `count` is decremented, and the value stored in the temporary variable is returned.

A Java class realizing this implementation might be called `ArrayQueue<T>`. It would implement the `Queue<T>` interface (which extends the `Container` interface) and have a `store` array and integer `frontIndex` and `count` variables as private fields. The queue operations would be public methods. Its constructor would create an empty queue. The capacity would be the length of the `store` array, which Java keeps track of.

6.6 LINKED IMPLEMENTATION OF THE QUEUE ADT

A linked implementation of a queue ADT uses a linked data structure to represent values of the ADT carrier set. A singly linked list is all that is required, so list nodes need only contain a value of type *T* and a link to the next node. We could keep a reference to only the head of the list, but this would require moving down the list from its head to its tail whenever an operation required manipulation of the other end of the queue, so it is more efficient to keep a reference to each end of the list. We will thus use both `frontPtr` and `rearPtr` references to keep track of both ends of the list.

If `rearPtr` refers to the head of the list and `frontPtr` to its tail, it will be impossible to remove elements from the queue without walking down the list from its head; in other words, we will have gained nothing by using an extra reference. Thus we must have `frontPtr` refer to the head of the list, and `rearPtr` to its tail. Figure 4 illustrates this data structure. Each node has an `item` field (for values of type *T*) and a `next` field (for the links). The figure shows a queue with three elements. The queue is empty when the `frontPtr` and `rearPtr` references are null; the queue never becomes full (unless memory is exhausted).

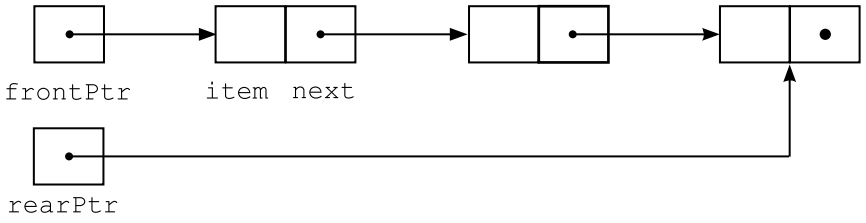


Figure 4: Implementing a Queue With a Linked List

Implementing the operations of the Queue ADT using a linked data structure is quite simple, though care must be taken to keep `frontPtr` and `rearPtr` synchronized. For example, to implement the `leave()` operation, a check is first made that the queue is not empty—if it is then an exception is raised. If not, then the `item` field of the front node is assigned to a temporary variable. The `frontPtr` field is then assigned the `next` field of the first node, which removes the first node from the list. If `frontPtr` is null, then the list has become empty, so `rearPtr` must also be assigned null. Finally, the value saved in the temporary variable is returned.

A Java class realizing this implementation might be called `LinkedQueue<T>`. It would implement the `Queue<T>` interface and have `frontPtr` and `rearPtr` private fields and the queue operations as public methods. It might also have a private inner `Node` class for node instances, and perhaps a `count` field to keep track of the number of items in the queue. Its constructor would create an empty queue. As new nodes are needed when values enter the queue, new `Node` instances would be allocated and used in the list.

6.7 SUMMARY AND CONCLUSION

Both queue implementations are simple and efficient, but the contiguous implementation either places a size restriction on the queue or uses an expensive reallocation technique if a queue grows too large. If contiguously implemented queues are made extra large to make sure that they don't overflow, then space may be wasted.

A linked implementation is essentially unbounded, so the queue never becomes full. It is also very efficient in its use of space because it only allocates enough memory to store the values actually kept in the queue. Overall, the linked implementation of queues seems slightly better than the contiguous implementation.

6.8 REVIEW QUESTIONS

1. Which operations of the queue ADT have preconditions? Do these preconditions translate to the `Queue` interface?
2. Why should storage be thought of as a circular rather than a linear arrangement of storage locations when implementing a queue using contiguous memory locations?
3. Why is there a reference to both ends of the linked list used to store the elements of a queue?

6.9 EXERCISES

1. In the contiguous storage implementation of a queue, is it possible to keep track of only the location of the front element (using a variable `frontIndex`) and the rear element (using a variable `rearIndex`), with no `count` variable? If so, explain how this would work.
2. Write the `Queue<T>` interface in Java.
3. Suppose that an `ArrayQueue` is implemented so that the array is reallocated when a client attempts to `enter()` an element when the array is full. Assume that the reallocated array is twice as large as the full array, and write Java code for the `enter()` operation that includes arranging the data where it needs to be in the newly allocated array.
4. Write a class invariant for a `LinkedQueue` class whose fields are `frontPtr`, `rearPtr`, and `count`.
5. A **circular singly linked list** is a singly linked list in which the last node in the list holds a references to the first element rather than null. It is possible to implement a `LinkedQueue` efficiently using only a single reference into a circular singly linked list rather than two references into a (non-circular) singly linked list as we did in the text. Explain how this works.
6. A `LinkedQueue` could be implemented using a doubly-linked list. What are the advantages or disadvantages of this approach?
7. Write the `ArrayQueue<T>` class in Java so that the queue never becomes full. If the `store` array fills up, then create a new `store` array that is twice the size (remember to move the elements starting at `frontIndex`). Make two constructors, the first with no parameters that creates an empty queue with room for eight values in its `store`, and the other with a parameter indicating the initial size of the `store` (if this argument is less than one, adjust it to eight).
8. Implement the `LinkedQueue<T>` class in Java. You class should have `frontPtr`, `rearPtr`, and `count` fields.
9. Implement the `LinkedQueue<T>` class without a `count` field.

6.10 REVIEW QUESTION ANSWERS

1. The *leave*() and *front*() operations both have as their precondition that the queue not be empty. This translates directly into the precondition of the `leave()` and `front()` operations of the `Queue` interface that the queue not be empty.
2. If storage is linear, then the data in a queue will "bump up" against the end of the array as the queue grows, even if there is space at the beginning of the array for queue elements. This problem can be solved by copying queue elements downwards in the array (inefficient), or by allowing the queue elements to wrap around to the beginning of the array, which effectively treats the array as a circular rather than a linear arrangement of memory locations.
3. In a queue, alterations are made to both ends of the container. It is not efficient to walk down the entire linked list from its beginning to get to the far end when an alteration must be made there. Keeping a reference to the far end of the list obviates this inefficiency and makes all queue operations very fast.

7 STACKS AND RECURSION

7.1 INTRODUCTION

Before moving on from discussing dispensers to discussing collections, we must explore the strong connection between stacks and recursion. Recall that recursion involves operations that call themselves.

> **Recursive operation**: An operation that either calls itself directly, or calls other operations that call it.

Recursion and stacks are intimately related in the following ways:

- Every recursive operation (or group of mutually recursive operations) can be rewritten without recursion using a stack.
- Every algorithm that uses a stack can be rewritten without a stack using one or more recursive operations.

To establish the first point, note that computers do not support recursion at the machine level—most processors can move data around, do a few simple arithmetic and logical operations, and compare values and branch to different points in a program based on the result, but that is all. Yet many programming language support recursion. How is this possible? At runtime, compiled programs use a stack that stores data about the current state of execution of a sub-program, called an *activation record*. When a sub-program is called, a new activation record is pushed on the stack to hold the sub-program's arguments, local variables, return address, and other book-keeping information. The activation record stays on the stack until the sub-program returns, when it is popped off the stack. Because every call of a sub-program causes a new activation record to be pushed on the stack, this mechanism supports recursion: every recursive call of a sub-program has its own activation record, so the data associated with a particular sub-program call is not confused with that of other calls of the sub-program. Thus the recursive calls can be recorded onto the stack, and a non-recursive machine can implement recursion.

The second point is not quite so easy to establish, but the argument goes like this: when an algorithm would push data on a stack, a recursive operation can preserve the data that would go on the stack in local variables and then call itself to continue processing. The recursive call returns just when it is time to pop the data off the stack, so processing can continue with the data in the local variables just as it would if the data had been popped off the stack.

In some sense, then, recursion and stacks are equivalent. It turns out that some algorithms are easier to write with recursion, some are easier to write with stacks, and some are just as easy (or hard) one way as the other. But in any case, there is always a way to write algorithms either entirely recursively without any stacks, or without any recursion using stacks.

In the remainder of this chapter we will illustrate the theme of the equivalence of stacks and recursion by considering a few examples of algorithms that need either stacks or recursion, and we will look at how to implement these algorithms both ways.

7.2 BALANCED BRACKETS

Because of its simplicity, we begin with an example that doesn't really need a stack or recursion, but illustrates how both can be used with equal facility: determining whether a string of brackets is balanced or not. The strings of balanced brackets are defined recursively as follows:

1. The empty string is a string of balanced brackets.
2. If A is a string of balanced brackets, then so it "[”A“]".
3. If A and B are strings of balanced brackets, then so is AB.

So, for example, [[[]]][]] is a string of balanced brackets, but [[[]][]][]] is not.

The recursive algorithm in Figure 1 (in Java) checks whether a string of brackets is balanced.

```
boolean isBalancedRecursive() {
 return isBalanced() && ch == EOS;
}

boolean isBalanced() {
 try {
  if (ch == EOS) return true;
  if (ch != '[') return false;
  ch = source.read();
  if (ch == '[')
   if (!isBalanced()) return false;
  if (ch != ']') return false;
  ch = source.read();
  if(ch == '[') return isBalanced();
  return true;
 } catch (IOException e) {
  return false;
 }
}
```

Figure 1: Recursive Algorithm For Checking String of Balanced Brackets

This code assumes that there is a `java.io.Reader` object called `source` and that `ch` is initialized with the first character from `source` before `isBalancedRecursive()` is called. The constant `EOS` is the value returned by `read()` (namely -1) when a `Reader` is at the end of its input stream.

The `isBalancedRecursive()` method merely calls the recursive `isBalanced()` method to do most of the work, and checks that the `source` input stream is exhausted when the helper function returns. This check takes care of the case of extra characters at the end of an otherwise legitimate string of balanced brackets.

The `isBalanced()` method is based on the recursive definition of balanced brackets. If the input stream is empty, corresponding to the first clause of the recursive definition, then the string is balanced. If it is not empty, and the first character is a left bracket, then the left bracket is consumed. If the next character is also a left bracket, then `isBalanced()` is called to check for nested balanced brackets. If the recursive call returns `true`, then a check is made for the right bracket matching the initial left bracket. This takes care of the second clause in the definition of balanced brackets. If the right bracket matches a left bracket,

then the right bracket is consumed. The final check to see whether the next character is a left bracket, and a call of `IsBalanced()` if it is, accounts for the case of a sequence of balanced brackets, as allowed by the third clause in the definition.

This same job could be done as easily with a non-recursive algorithm using a stack. In the code in Figure 2 below, a stack is used to hold left brackets as they are encountered. If a right bracket is found for every left bracket on the stack, then the string of brackets is balanced. Note that the stack must be checked to make sure it is not empty as we go along (which would mean too many right brackets), and that it is empty when the entire string is processed (which would mean too many left brackets).

```
boolean isBalancedStack() {
 Stack<Integer> s = new LinkedStack<Integer>();
 try {
  while (ch != EOS) {
   switch (ch) {
   case '[':
    s.push(ch);
    break;
   case ']':
    if (s.isEmpty()) return false;
    s.pop();
    break;
   default: return false;
   }
  ch = source.read();
  }
  return s.isEmpty();
 } catch (IOException e) {
  return false;
 }
}
```

Figure 2: Non-Recursive Algorithm For Checking Strings of Balanced Brackets

These algorithms are about equally complicated, so there is no particular advantage to using either a stack or recursion in this example. In the examples below, either the recursive algorithm or the algorithm that uses a stack is easier, depending on the problem.

7.3 INFIX, PREFIX, AND POSTFIX EXPRESSIONS

The arithmetic expressions we learned in grade school are infix expressions, but other kinds of expressions, called prefix or postfix expressions, might also be used.

Infix expression: An expression in which the operators appear between their operands.

Prefix expression: An expression in which the operators appear before their operands.

Postfix expression: An expression in which the operators appear after their operands.

In a prefix expression, the operands of an operator appear immediately to its right, while in a postfix expression, they appear immediately to its left. For example, the infix expression (4 + 5) * 9 can be rewritten in prefix form as * + 4 5 9 and in postfix form as 4 5 + 9 *. An advantage of pre- and postfix expressions over infix expressions is that the latter don't need parentheses.

Many students are confused by prefix and postfix expressions the first time they encounter them, so lets consider a few more examples. In the expressions in the table below, all numbers are one digit long and the operators are the usual binary integer operations. All the expressions in a row are equivalent.

Infix	Prefix	Postfix
(2 + 8) * (7 % 3)	* + 2 8 % 7 3	2 8 + 7 3 % *
((2 * 3) + 5) % 4	% + * 2 3 5 4	2 3 * 5 + 4 %
((2 * 5) % (6 / 4)) + (2 * 3)	+ % * 2 5 / 6 4 * 2 3	2 5 * 6 4 / % 2 3 * +
1 + (2 + (3 + 4))	+ 1 + 2 + 3 4	1 2 3 4 + + +
((1 + 2) + 3) + 4	+ + + 1 2 3 4	1 2 + 3 + 4 +

Note that all the expressions have the digits in the same order. This is necessary because order matters for the subtraction and division operators. Also notice that the order of the operators in a prefix expression is not necessarily the reverse of its order in a postfix expression; sometimes operators are in the opposite order in these expressions, but not always. The systematic relationship between the operators is that the main operator always appears within the fewest number of parentheses in the infix expression, is first in the prefix expression, and is last in the postfix expression. Finally, in every expression, the number of constant arguments (digits) is always one more than the number of operators.

Let's consider the problem of evaluating prefix and postfix expressions. It turns out that sometimes it is much easier to write a recursive evaluation algorithm, and sometimes it is much easier to write a stack-based evaluation algorithm. In particular,

- It is very easy to write a recursive prefix expression evaluation algorithm, but somewhat harder to write this algorithm with a stack.
- It is very easy to write a stack-based postfix expression evaluation algorithm, but very hard to write this algorithm recursively.

To establish these claims, we will consider a few of the algorithms. An algorithm in Java to evaluate prefix expressions recursively appears in Figure 3 below. These methods assume a stream of characters called `source`, and that the variable `ch` has been initialized with the first character from `source`. The `evalPrefixRecursive()` method calls the recursive `evalPrefix()` method to do most of the work, merely ensuring that the stream is empty before returning the result supplied by `evalPrefix()`. (Extra characters in `source` indicate too many arguments to an operator, so this raises an exception.)

```
public int evalPrefixRecursive() throws Exception {
 int result = evalPrefix();
 if (ch == EOS) return result;
 throw new IllegalArgumentException();
}

private int evalPrefix() throws Exception {
 if (ch == EOS)
  throw new IllegalArgumentException();
 else if (Character.isDigit(ch)) {
  int num = ch - '0';
  ch = source.read();
  return num;
 } else {
  int op = ch;
  ch = source.read();
  return applyOp(op, evalPrefix(), evalPrefix());
 }
}
```

Figure 3: Recursive Algorithm to Evaluate Prefix Expressions

It helps to consider the recursive definition of a prefix expression to understand this algorithm:

A prefix expression is either a digit, or if *A* and *B* are prefix expressions and *op* is an operator, then an expression of the form *op A B*.

As noted, `evalPrefix()` does the real work. If first checks to see whether the stream is exhausted and throws an exception if it is (because the empty string is not a prefix expression). Otherwise, it reads the next character and checks to see whether it is a digit. If so, this is the basis case of the recursive definition of a prefix expression, so it simply returns the integer value of the digit. Otherwise, the current character is an operator. According to the recursive definition, an operator should be followed by two prefix expressions, so the algorithm applies this operator to the result of recursively evaluating the following left and right arguments (this is done by the helper method `applyOp()`). If these arguments are not there, or are ill-formed, then one of these recursive calls will throw an exception that is propagated to the caller.

```
int evalPrefixStack() throws Exception {
 Stack<Integer> opStack = new LinkedStack<Integer>();
 Stack<Integer> argStack = new LinkedStack<Integer>();
 while (ch != EOS) {
  if (Character.isDigit(ch)) {
   int num = ch - '0';
   argStack.push(num);
   while (!opStack.isEmpty() && opStack.top() == 'v') {
    opStack.pop();
    if (opStack.isEmpty())
     throw new IllegalArgumentException();
    int op = opStack.pop();
    if (argStack.isEmpty())
     throw new IllegalArgumentException();
    int arg2 = argStack.pop();
    if (argStack.isEmpty())
     throw new IllegalArgumentException();
    int arg1 = argStack.pop();
    argStack.push( applyOp(op, arg1, arg2) );
   }
   if (!opStack.isEmpty() && opStack.top() != 'v')
    opStack.push((int)'v');
  } else {
   opStack.push(ch);
  }
  ch = source.read();
 }
 if (!opStack.isEmpty()) throw new IllegalArgumentException();
 if (argStack.isEmpty()) throw new IllegalArgumentException();
 int result = argStack.pop();
 if (!argStack.isEmpty()) throw new IllegalArgumentException();
 return result;
}
```

Figure 4: Stack-Based Algorithm to Evaluate Prefix Expressions

The recursive algorithm is simple, yet is does a potentially very complicated job. In contrast, consider the code to evaluate a prefix expression using stack shown in Figure 4. This algorithm has two stacks: one for operators (`opStack`) and one for values (`argStack`). Its strategy is to process each character from the source stream in turn, pushing operators on the `opStack` as they are encountered and values on the `argStack` to retain left arguments. Left argument placement is marked on the `opStack` by pushing on it the character 'v'. Operators are applied whenever both left and right arguments are available, and the result is pushed back on the `argStack` and marked on the `opStack`. Once the string is exhausted, the result value should be the only thing in the `argStack` and the `opStack` should be empty. The best way to understand how this algorithm works is to run through a few examples by hand.

Clearly, this stack-based evaluation algorithm is more complicated than the recursive evaluation algorithm. In contrast, a stack-based evaluation algorithm for postfix expressions is quite simple, while a recursive algorithm is quite complicated. To illustrate, consider the stack-based postfix expression evaluation algorithm in Figure 5 below.

```
public int evalPostfixStack() throws Exception {
 Stack<Integer> stack = new LinkedStack<Integer>();
 while (ch != EOS) {
  if (Character.isDigit(ch)) stack.push(ch-'0');
  else {
   if (stack.isEmpty())
    throw new IllegalArgumentException();
   int arg2 = stack.pop();
   if (stack.isEmpty())
    throw new IllegalArgumentException();
   int arg1 = stack.pop();
   stack.push( applyOp(ch, arg1, arg2) );
  }
  ch = source.read();
 }
 if (stack.isEmpty()) throw new IllegalArgumentException();
 int result = stack.pop();
 if (!stack.isEmpty()) throw new IllegalArgumentException();
 return result;
}
```

Figure 5: Stack-Based Algorithm to Evaluate Postfix Expressions

The strategy of this algorithm is quite simple: there is a single stack that holds arguments, and values are pushed on the stack whenever they are encountered in the source stream. Whenever an operator is encountered, the top two values are popped of the stack, the operator is applied to them, and the result is pushed back on the stack. This continues until the stream is exhausted, at which point the final value should be on the stack. If the stack becomes empty along the way, or there is more than one value on the stack when the source stream is exhausted, then the input expression is not well-formed.

As noted, the recursive algorithm for evaluating postfix expressions is quite complicated. The strategy is to remember arguments in local variables, making recursive calls as necessary until an operator is encountered. We leave this algorithm as a challenging exercise. The lesson of all these examples is that although it is always possible to write an algorithm using either recursion or stacks, in some cases a recursive algorithm is easier to develop, in other cases a stack-based algorithm is easier5, and in some cases neither approach is easier. Each problem should be explored by sketching out both sorts of algorithms, and then choosing the one that appears easiest for detailed development.

7.4 TAIL RECURSIVE ALGORITHMS

We have claimed that every recursive algorithms can be replaced with a non-recursive algorithm using a stack. This is true, but it overstates the case: sometimes a recursive algorithm can be replaced with a non-recursive algorithm that does not even use a stack. If a recursive algorithm is such that only one recursive call is made as the final step in each execution of the algorithm's body, then the recursion can be replaced with a loop. No stack is needed because data for additional recursive calls is not needed—there are no additional recursive calls. A very simple example is a recursive algorithm to search an array, like the one shown in Figure 6.

```
public static boolean recursiveSearch(
                        int[] array, int key, int start) {
 if (array[start] == key) return true;
 int next = start+1;
 if (array.length == next) return false;
 return recursiveSearch(array, key, next);
}
```

Figure 6: A Recursive Array Search Algorithm

The recursion in this algorithm can be replaced with a simple loop as shown in Figure 7.

```
public static boolean search(int[] array, int key) {
 for (int v : array) {
  if (v == key) return true;
 }
 return false;
}
```

Figure 7: A Non-Recursive Array Search Algorithm

Algorithms that only call themselves once as the final step in every execution of their bodies, like this search algorithm, are called *tail-recursive*.

> **Tail recursive algorithm**: A recursive algorithm that calls itself at most once as the last step in every execution of its body.

Recursion can always be removed from tail-recursive algorithms without using a stack. We will see another example of tail recursion when we consider binary search.

7.5 SUMMARY AND CONCLUSION

Algorithms that use recursion can always be replaced by algorithms that use a stack, and vice versa, so stacks and recursion are in some sense equivalent. However, some algorithms are much easier to write using recursion, while others are easier to write using a stack. Which is which depends on the problem. Programmers should evaluate both alternatives when deciding how to solve individual problems.

7.6 REVIEW QUESTIONS

1. Which of the algorithms for determining whether a string of brackets is balanced is easiest to for you to understand?
2. What characteristics do prefix, postfix, and infix expressions share?
3. Which is easier: evaluating a prefix expression with a stack or using recursion?
4. Which is easier: evaluating a postfix expression with a stack or using recursion?
5. Is the recursive algorithm to determine whether a string of brackets is balanced tail recursive? Explain why or why not.

7.7 EXERCISES

1. We can slightly change the definition of strings of balanced brackets to exclude the empty string. Restate the recursive definition and modify the algorithms to check strings of brackets to see whether they are balanced to incorporate this change.
2. Fill in the following table with equivalent expressions in each row.

Infix	Prefix	Postfix
(((2 * 3) - 4) * (8 / 3)) + 2		
	% + 8 * 2 6 - 8 4	
		8 2 - 3 * 4 5 + 8 % /

3. Write a recursive algorithm to evaluate postfix expressions as discussed in this chapter.
4. Write a recursive algorithm to evaluate infix expressions. Assume that operators have equal precedence and are left-associative so that, without parentheses, operations are evaluated from left to right. Parentheses alter the order of evaluation in the usual way.
5. Write a stack-based algorithm to evaluate infix expressions as defined in the last exercise.
6. Which of the algorithms for evaluating infix expressions is easier to develop?
7. Write a non-recursive algorithm that does not use a stack to determine whether a string of brackets is balanced. Hint: count brackets.

7.8 REVIEW QUESTION ANSWERS

1. This answer depends on the individual, but most people probably find the stack-based algorithm a bit easier to understand because its strategy is so simple.
2. Prefix, postfix, and infix expressions list their arguments in the same order. The number of operators in each is always one less than the number of constant arguments. The main operator in each expression and sub-expression is easy to find: the main operator in an infix expression is the left-most operator inside the fewest number of parentheses; the main operator of a prefix expression is the first operator; the main operator of a postfix expression is the last operator.
3. Evaluating a prefix expression recursively is much easier than evaluating it with a stack.
4. Evaluating a postfix expression with a stack is much easier than evaluating it recursively.
5. The recursive algorithm to determine whether a string of brackets is balanced calls itself at most once on each activation, so it is tail recursive activation, but the recursive call is not the last step in the execution of the body of the algorithm—there must be a check for the closing right bracket after the recursive call. Hence this operation is not tail recursive and it cannot be implemented without a stack. (There is a non-recursive algorithm to check for balanced brackets without using a stack, but it uses a completely different approach from the recursive algorithms—see exercise 7).

8 COLLECTIONS AND ITERATORS

8.1 INTRODUCTION

Recall that we have defined a collection as a type of container that is traversable, that is, a container that allows access to all its elements. The process of accessing all the elements of a collection is also called iteration. Iteration over a collection may be supported in several ways depending on the agent that controls the iteration and where the iteration mechanism resides. In this chapter we examine iteration design alternatives and discuss how collections and iteration work in Java. Based on this discussion, we will decide how to support collection iteration in our container hierarchy, and how to add collections to the hierarchy.

8.2 ITERATION DESIGN ALTERNATIVES

There are two ways that iteration may be controlled.

> **Internal iteration**—When a collection controls iteration over its elements, then iteration is said to be *internal*. A client wishing to process each element of a collection packages the process in some way (typically in an operation), and passes it to the collection, perhaps with instruction about how iteration is to be done. The collection then applies the processing to each of its elements. This mode of control makes it easier for the client to iterate over a collection, but with less flexibility in dealing with issues that may arise during iteration.
>
> **External iteration**—When a client controls iteration over a collection, the iteration is said to be *external*. In this case, the client must be provided with operations that allow an iteration to be initialized, to obtain the current element from the collection, to move on to the next element in the collection, and to determine when iteration is complete. This mode of control imposes a burden on the client in return for more flexibility in dealing with the iteration.

In addition to issues of control, there are also alternatives concerning where the iteration mechanism resides.

> **In the language**—An iteration mechanism may be built into a language. For example, Java, Ruby, and Go (to name just a few examples) have special looping control structures that provide means for external iteration over collections.

In the collection—An iteration mechanism may reside in a collection. In the case of a collection with an external iteration mechanism, the collection must provide operations to initialize iteration, return the current element, advance to the next element, and indicate when iteration is complete. In the case of a collection with an internal iteration mechanism, the collection must provide an operation that accepts a packaged process and applies it to each of its elements.

In an iterator—An iteration mechanism may reside in a separate entity whose job is to iterate over an associated collection. In this case the operations mentioned above to support internal or external iteration are in the iterator and the collection usually has an operation to create iterators.

Combining these design alternatives gives six ways that iteration can be done: internal iteration residing in the language, in the collection, or in an iterator, and external iteration residing in the language, in the collection, or in an iterator. Each of these alternatives has advantages and disadvantages, and various languages and systems have incorporated one or more of them. For example, most object-oriented languages have external iteration residing in iterator objects (this is known as the Iterator design pattern). Nowadays many languages provide external iteration in control structures, as mentioned above. We will now consider the Iterator design pattern, and then internal and external iteration in Java.

8.3 THE ITERATOR DESIGN PATTERN

A software design pattern is an accepted solution to a common design problem that is proposed as a model for solving similar problems.

> **Software design pattern**: A model proposed for imitation in solving a software design problem.

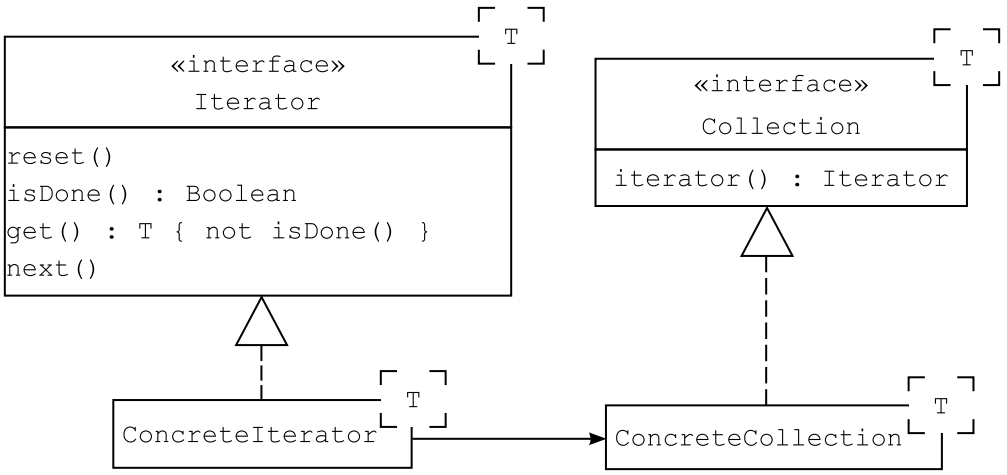
Design patterns occur at many levels of abstraction. For example, an algorithm or data structure is a low-level design pattern, and the overall structure of a very large program (such as a client-server structure) is a high-level design pattern. The Iterator pattern is a mid-level object-oriented design pattern that specifies the composition and interactions of several classes and interfaces.

The Iterator pattern consists of an `Iterator` class whose instances are created by an associated collection and provided to clients. The `Iterator` instances house an external iteration mechanism. Although `Iterator` class functionality can be packaged in various ways, `Iterator` classes must provide the following functionality.

> *Initialization*—Prepare the `Iterator` object to traverse its associated collection. This operation will set the current element (if there is one).
>
> *Completion Test*—Indicate whether traversal by this `Iterator` is finished.
>
> *Current Element Access*—Provide the current collection element to the client. The precondition for this operation is that iteration is not complete.
>
> *Current Element Advance*—Make the next element in the collection the current element. This operation has no effect once iteration is complete. However, iteration may become complete when it is invoked—in other words, if the current item is the last, executing this operation completes the iteration, and calling it again does nothing.

The class diagram in Figure 1 below presents the static structure of the Iterator pattern. The four operations in the Iterator interface correspond to the four functions listed above. The `iterator()` operation in the `Collection` interface creates and returns a new concrete iterator for the particular collection in which it occurs; this is called a factory method because it manufactures a class instance.

The interfaces and classes in this pattern are templated with the type of the elements held in the collection. The arrow from the `ConcreteIterator` to the `ConcreteCollection` indicates that the `ConcreteIterator` must have some sort of reference to the collection with which it is associated so that it can access its elements.



A client uses an iterator by requesting one from a `ConcreteCollection` by calling its `iterator()` operation. The client can then reset the iterator and use a while loop to access its elements. The Java code below in Figure 2 illustrates how this is done.

```
Collection c = new ConcreteCollection<T>();
...
Iterator<T> i = c.iterator();
while ( !i.isDone ) {
 T element = i.get;
 // process element
 i.next();
}
```

Figure 2: Using an Iterator

Note that if the programmer decided to switch from one `ConcreteCollection` to another, only one line of this code would have to be changed: the first. Because of the use of interfaces, the code would still work even though a different `ConcreteIterator` would be used to access the elements of the collection.

8.4 COLLECTIONS AND ITERATION IN JAVA

Lets now consider how Java provides collection iteration. The Java platform includes an entire group of collection classes and interfaces called the *Java collections framework*. In this framework, `java.util.Collection<T>` is the top interface for collections. It is implemented by classes providing stacks, queues, dequeues (double-ended queues), lists, and sets. There is also a `java.util.Map<K,E>` interface that is implemented by

various kinds of maps. We will study these collections (including maps) in this book. For now, you should appreciate that the Java collections framework provides a rich variety of collections with sophisticated methods supporting many computational needs. Although we are building our own container hierarchy to learn about containers, the Java collections framework is what you should use for containers once this course is over.

Java has a `java.lang.Iterable<T>` interface whose only method is `iterator()`, a factory method that returns a class that implements the `java.util.Iterator<T>` interface; `Collection<T>` extends the `Iterable<T>` interface. In other words, the Java collections framework uses the Iterator pattern to provide external iterators. Furthermore, any class that implements the `Iterable` interface can be used as a target of the for-each statement. This is how Java supports external iteration as part of the language. Hence Java supports two of the six iteration design alternatives mentioned above. (Java 8 also supports internal iteration, but we will not discuss it here.)

Java's `Iterator` interface is slightly different from the one we have discussed as part of the Iterator design pattern.

- The Java `Iterator` interface has no `reset()` method. When an `Iterator` object is created, it is also initialized. Once iteration is complete, the `Iterator` object is "used up" and a new one must be created.
- The Java `Iterator` interface method for testing whether iteration is complete is called `hasNext()` rather than `isDone()`, and it returns the opposite boolean value.
- The Java `Iterator` interface method `next()` combines the `get()` and `next()` methods of the Iterator design pattern: it not only moves to the next element in the collection, but returns the current element as well. It throws a `NoSuchElementException` if it is called when the collection is exhausted.
- Finally, the Java `Iterator` interface has a `remove()` method for removing the element returned by `next()` from the collection. Modifying a collection during iteration is not always a well-defined operation. Supplying this method in an iterator (which is optional in Java), provides a safe way to modify the collection. There is no method in the interface for safely inserting elements into a collection during iteration, however.

One nice feature of Java is that any class that implements the `Iterable` interface can be traversed using the for-each loop. Thus by defining an iterator for an arbitrary collection class, it automatically becomes traversable using this control structure.

In our container hierarchy we will use the methods from the Java `Iterator` interface to implement external iterators. We will also assume that collections are never altered during iteration, so we will not bother with the optional `Iterator remove()` method. Finally, our collections will implement the Java `Iterable` interface. This means that we will be able to use the for-each statement on our collections.

8.5 COLLECTIONS AND ITERATORS IN THE CONTAINER HIERARCHY

When discussing the Java collections framework we mentioned several kinds of collections, including lists (simple linear sequences), sets (unordered aggregates), and maps (aggregates with keyed access). There are not many methods common to this variety of collections that should be included in the `Collection` interface. For example, although one must be able to add elements to every collection, how elements are added varies. Adding an element to an unordered collection simply involves the element added. Adding to a list requires specifying where the element is to be added, and adding to a map requires that the access key be specified. Only one common method comes to mind: we may ask of any collection whether it contains some particular element. Consequently, we add a collection containment query method to the `Collection` interface.

Figure 3 is a UML diagram showing the `Collection` interface. Of course it extends the `Container` interface. As mentioned, it also extends the `Iterable` interface because it must include an `iterator()` method.

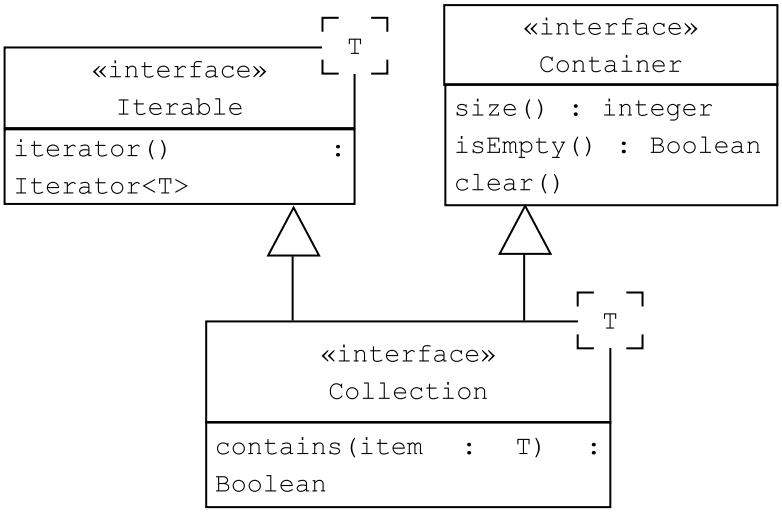


Figure 3: Collection in the Container Hierarchy

8.6 SUMMARY AND CONCLUSION

Collections are traversable containers and hence require some sort of iteration facility. There are many alternative designs for such a facility. The Iterator design pattern, a way to use external iterator objects, is a good model for external iterators. We have used the variation of this pattern in Java to specify our `Collection` interface. Collections should also include a collection containment query operation, so this also appears in our `Collection` interface.

8.7 REVIEW QUESTIONS

1. What are the alternatives for controlling iteration?
2. Where might iteration mechanisms reside?
3. What are the six alternatives for designing collection iteration facilities?
4. What is a software design pattern?
5. What functions must an external `Iterator` object provide in the Iterator pattern?
6. What sort of support does Java provide for collection iteration?
7. What does the `contains()` operation return when a `Collection` is empty?

8.8 EXERCISES

1. What is the point of an iterator when each element of a list can already be accessed one by one using indices?
2. Explain why a `Dispenser` does not have an associated iterator.
3. Java has iterators, but the Java `Iterator` interface does not have a `reset()` operation. Why not?
4. Would it be possible to have an `Iterator` interface with only a single operation? If so, how could the four `Iterator` functions be realized?
5. How might external iterators residing in collections be added to the Java collections framework?
6. How might internal iteration be added to the Java collections framework?
7. Write a Java implementation of the `Collection contains()` operation.
8. What happens to an iterator (any iterator) when its associated collection changes during iteration?
9. Consider the problem of checking whether two collections contain the same values. Can this problem be solved using collection internal iterators? Can it be solved using external iterators?

8.9 REVIEW QUESTION ANSWERS

1. There are two alternatives for controlling iteration: the collection may control it (internal iteration) or the client may control it (external iteration).
2. Iteration mechanisms can reside in three places: in the language (in the form of control structures), in the collection (as a set of operations), or in a separate iterator (with certain operations).
3. The six alternatives for designing collection iteration facilities are generated by combining control alternatives with residential alternatives, yielding the following six possibilities: (1) internal control residing in the language, (2) external control residing in a language, (3) internal control residing in the collection, (4) external control residing in the collection, (5) internal control residing in an iterator, (6) external control residing in an iterator.
4. A software pattern is model proposed for imitation in solving a software design problem. In other words, a pattern is a way of solving a design or implementation problem that has been found to be successful, and that can serve as a template for solving similar problems.
5. An `Iterator` must provide four functions: a way to initialize the `Iterator` to prepare to traverse its associated `Collection`, a way to fetch the current element of the `Collection`, a way to advance to the next element of the `Collection`, and a way to indicate that all elements have been accessed.
6. Java provides support for external iteration over classes that implement the `Iterable` interface by making them traversable by the built-in for-each loop. Furthermore, such classes have an `iterator()` factory method that returns an instance of a class that implements the `Iterator` interface, meaning that the Iterator pattern is built into Java, so an `Iterator` can also be used in a while loop to traverse collections.
7. If a `Collection` is empty, then it contains nothing so the `contains()` operation returns false no matter what its argument.

9 LISTS

9.1 INTRODUCTION

Lists are linearly ordered collections. Some things we refer to in everyday life as lists, such as shopping lists or laundry lists, are really sets because their order doesn't matter. Order matters in lists. A to-do list is really a list if the tasks to be done are in the order in which they are supposed to be completed (or some other order).

List: An ordered linear collection.

Because order matters in lists, we must specify a location, or index, of elements when we modify the list. Indices can start at any number, but we will follow convention and give the first element of a list index 0, the second index 1, and so forth.

9.2 THE LIST ADT AND INTERFACE

Lists are collections of values of some type, so the ADT is *list of T*, where *T* is the type of the elements in the list. The carrier set of this type is the set of all sequences or ordered tuples of elements of type *T*. The carrier set thus includes the empty list, the lists with one element of type *T* (one-tuples), the lists with two elements of type *T* (ordered pairs), and so forth. Hence the carrier set of this ADT is the set of all tuples of type *T*, including the empty tuple.

There are many functions that may be included in a list ADT; the following is a typical list ADT implicit-receiver method set. The result of a function is undefined if its precondition is violated.

size()—Return the length of the list.

insert(*i*,*e*)—Place *e* into the list at index *i*, moving elements with larger indices up in the list, if necessary. The precondition of this operation is that *i* be a valid index position: $0 \le i \le size()$. When *i* is 0, *e* is inserted at the front of the list, and when *i* is *size*(), *e* is appended to the end of the list.

delete(*i*)—Remove the element at index *i* from the list and return the deleted element. The precondition of this operation is that *i* be a valid index position: $0 \le i < size()$.

get(*i*)—Return the value at index *i* of the list. Its precondition is that *i* be a valid index position: $0 \le i < size()$.

put(*i*,*e*)—Replace the element at index *i* of the list with *e*. The precondition is that *i* be a valid index position: $0 \leq i < size()$.

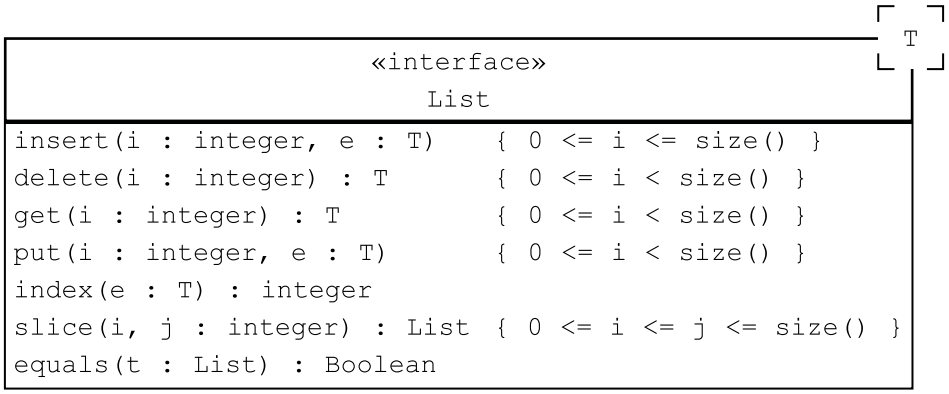
index(*e*)—Return the index of the first occurrence of *e* in the list. The precondition of this operation is that *e* is in the list.

slice(*i*, *j*)—Return a new list that is a slice of the list whose first element is the value at index *i* of the list and whose last value is at index *j*-1 of the list. The precondition of this operation is that *i* and *j* are are valid: $0 \leq i \leq j \leq size()$. Note that the slice may be empty if $i = j$.

isEqual(*s*)—Return true if and only if list *s* has the same elements in the same order as the (receiver) list.

As with the ADTs we have studied before, an implementation of these operations as methods of a class has the class as the host of the methods, so the signatures of these operations will vary somewhat when they are implemented in Java.

A `List` interface is a sub-interface of `Collection`, which is a sub-interface of `Container`, so it has several operations that it inherits from its ancestors. The UML diagram in Figure 1 shows the `List` interface. As usual, a template parameter is used to generalize the interface for any element type. Preconditions are listed when they are needed; exceptions may be thrown when these are violated. The `index()` method returns -1 if its parameter does not appear in the list.



9.3 AN EXAMPLE USING LISTS

Suppose a calendar program has a to-do list whose elements are ordered by precedence, so that the first element on the list must be done first, the second next, and so forth. The items in the to-do list are all visible in a scrollable display; items can be added, removed, or moved around in the list freely. Mousing over list items displays details about the item, which can be changed. Users can ask to see or print sub-lists (like the last ten items on the list), or they can ask for details about a list item with a certain precedence (like the fifth element).

Clearly, a `List` is the right sort of container for ordered to-do lists. Iteration over a `List` to display its contents is easy with an internal or external iterator. The `insert()` and `delete()` operations allow items to be inserted into the list, removed from it, or moved around in it. The `get()` operation can be used to obtain a list element for display during a mouse-over event, and the `put()` operation can replace a to-do item's record if it is changed. The `slice()` operation produces portions of the list for display or printing, and the `index()` operation can determine where an item lies in the list.

9.4 CONTIGUOUS IMPLEMENTATION OF THE LIST ADT

Lists are very easy to implement with arrays. Static arrays impose a maximum list size, while dynamic arrays allow lists to grow without limit. The implementation is similar in either case. An array is allocated to hold the contents of the list, with the elements placed into the array in order so that the element at index i of the list is at index i of the array. A counter maintains the current size of the list. Elements added at index i require that the elements in the slice from i to the end of the list be moved upwards in the array to make a "hole" into which the inserted value is placed. When element i is removed, the slice from i+1 to the end of the list is copied down to close the hole left when the value at index i is removed.

Static arrays have their size set at compile time so an implementation using a static array cannot accommodate lists of arbitrary size. In contrast, an implementation using a dynamic array can allocate a larger array if a list exceeds the capacity of the current array during execution. Reallocating the array is an expensive operation because the new, larger array must be created, the contents of the old, smaller array must be copied into the new array, and the old array must be deallocated. To avoid this expense, the number of array reallocations should be kept to a minimum. One popular approach is to double the size of the array whenever it needs to be made larger. For examples, suppose a list begins with a capacity of 10. As it expands, its capacity is changed to 20, then 40, then 80, and so forth. The array never becomes smaller.

Iterators for lists implemented with an array are also very easy to code. The iterator need merely keep a reference to the list and the current index during iteration, which acts as a cursor marking the current element during iteration.

> **Cursor**: A variable marking a location in a data structure.

Accessing the element at index i of an array is almost instantaneous, so the `get()` and `put()` operations are very fast using a contiguous implementation. But adding and removing elements requires moving slices of the list up or down in the array, which can be very slow. Hence for applications where list elements are often accessed but not too often added or removed, the contiguous implementation will be very efficient; applications that have the opposite behavior will be much less efficient, especially if the lists are long.

9.5 LINKED IMPLEMENTATION OF THE LIST ADT

A linked implementation of the list ADT uses a linked data structure to represent values of the ADT carrier set. A singly- or multiply-linked list may be used, depending on the needs of clients. We will consider using singly- or doubly-linked lists to illustrate implementation alteratives.

Suppose a singly-linked list is used to hold list elements. It consists of a variable, often called `head`, holding a reference to the first node in the list. This first node may contain data for the first element in the list, or it may be a dummy node (thought of as being at location -1) that is always present, even when the list is empty. Using a dummy node wastes a bit of space, but it greatly simplifies various list manipulation algorithms, so we will use one in our algorithms. In any case, when the list is empty, either `head` or the link field of the dummy node is null. The length of the list is also typically recorded.

Most list operations take an index *i* as an argument, so most algorithms to implement these operations will have to begin from `head` and walk from node to node down the list to locate the node at position *i* or position *i*-1. (Why *i*-1? Because for addition and removal, the link field in the node preceding node *i* will have to be changed.) If lists are long and there are many changes towards the end of the list, much processing must be done simply finding the right spot in the list to do things.

This difficulty can be alleviated by keeping a cursor consisting of an index number and a reference into the list. The cursor is used to find the node that some operation needs to do its job. The next time an operation is called, it may be able to use the existing value of the cursor, or use it with slight changes, thus saving time. For example, suppose that a value is added at the end of the list. The cursor is used to walk down to the end of the list and make the addition; when this task is done, the cursor marks the node at, let us say, location `size()`-2 in the list. If another addition is made, the cursor only needs to be moved forward one node to the end of the list to mark the node whose link field must be changed—the walk down the list from its beginning has been avoided.

It may also be useful to maintain a reference to the end of the list. Then operations at the end of the list can be done quickly in exchange for the slight additional effort of maintaining the extra reference. If a client does many operations at the end of a list, the extra work will be justified.

Another way to make list operations faster is to store elements in a doubly-linked list in which each node (except those at the ends) has a link to both its successor and its predecessor nodes. A dummy first node may be used as well. Figure 2 below illustrates this setup (with a dummy first node).

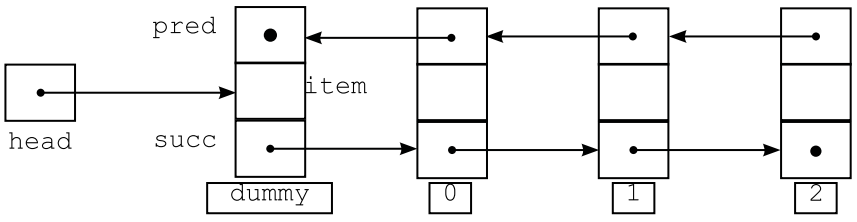


Figure 2: A Doubly-Linked List

Using a cursor with a doubly-linked list can speed things up considerably because the links make it possible to move both backwards and forwards in the list. If an operation needs to get to node *i* and the cursor marks node *j*, which is closer to node *i* than node *i* is to the head of the list, following links from the cursor can get to node *i* more quickly than following links from the head of the list. Keeping a reference to the end of the list makes things faster still: it is possible to start walking toward a node from three points: the front of the list, the end of the list, or the cursor, which is often somewhere in the middle of the list.

Another trick is to make the list circular: have the `pred` link of the first node refer to the last node rather than containing null, and have the `succ` link of the last node refer to the first node rather than containing null. Then there is no need for a separate reference to the end of the list: the `head` reference can be used to get to both the front and the rear of the list. This obviates the need for a pointer to the end of the list.

To illustrate, consider the code in Figure 3 below for setting the cursor in a doubly-linked circular list with a dummy node. Setting the cursor is done in almost every list operation, so it is important to make this code efficient.

```
private void setCursor(int index) {
 if (index <= count/2) {
  if (index+1 < Math.abs(index-cursorIndex)) {
   cursorIndex = -1;
   cursorPtr = head;
  }
 } else {
  if ((count-index) < Math.abs(index-cursorIndex)) {
   cursorIndex = count;
   cursorPtr = head;
  }
 }
 while (cursorIndex < index) {
  cursorPtr = cursorPtr.succ;
  cursorIndex++;
 }
 while (index < cursorIndex) {
  cursorPtr = cursorPtr.pred;
  cursorIndex--;
 }
}
```

Figure 3: Setting the Cursor in a Doubly-Linked Circular List with a Dummy Node

The conditionals determine whether the current cursor position (marked jointly by `cursorPtr` and `cursorIndex`), the front of the list, or the end of the list is closer to the target `index`, and either leaves the cursor alone or sets it to the front or the end of the list. Note that the dummy node (pointed to by `head`) is both just before the front of the list (at location -1) *and* just after the end of the list (at location `count`) because the list is circular. The while loops then either march the cursor forward or backward to the target position. Note that only one of these while loops executes in a call of this method. This code is somewhat tricky, so it may help to work through a few examples to see exactly how it works.

Iterators for the linked implementation of lists must obtain a reference to the head of the list; this can be passed to the new `Iterator` object by the factory function that creates it. Then it is merely a question of maintaining a cursor and walking down the list whenever the `Iterator.next()` operation is called.

Modifying lists with a linked implementation is very fast once the nodes to operate on have been found, and using doubly-linked lists with cursors can make node finding fairly fast. As a rule, a linked implementation of a list allows for faster list modifications than a contiguous implementation, but slower list element access than a contiguous implementation. Hence a linked implementation will generally be a better choice when a list is modified frequently but accessed relatively infrequently.

9.6 EXAMPLE: MODIFYING A DOUBLY-LINKED CIRCULAR LIST

Doubly-linked lists have many links, and sometimes it is confusing to see how to modify them when performing list operations. Consequently Figure 4 shows code for inserting and deleting in a doubly-linked circular list with a dummy node.

```
public void insert(int i, T item) throws IndexOutOfBoundsException {
 if (i < 0 || count < i) throw new IndexOutOfBoundsException();
 setCursor(i);
 Node newNode = new Node(item, cursorPtr, cursorPtr.pred);
 cursorPtr.pred = cursorPtr.pred.succ = newNode;
 cursorPtr = newNode;
 count++;
}

public T delete(int i) throws IndexOutOfBoundsException {
 if (i < 0 || count <= i) throw new IndexOutOfBoundsException();
 setCursor(i);
 T result = cursorPtr.item;
 cursorPtr.succ.pred = cursorPtr.pred;
 cursorPtr.pred.succ = cursorPtr.succ;
 cursorPtr = cursorPtr.succ;
 count--;
 return result;
}
```

Figure 4: Modifying a Doubly-Linked Circular List with a Dummy Node

Both `insert()` and `delete()` begin by ensuring that the index at which the operation is to take place are in range. Then both methods set the cursor to the node where the insertion or deletion is to occur.

For an insertion, the node at which `cursorPtr` points will be the successor of the inserted node. Consequently a new node is created whose successor is the cursor node (pointed to by `cursorPtr`), and whose predecessor is the predecessor of the cursor node (pointed to by `cursorPtr.pred`). Then the cursor node's predecessor pointer and the cursor node's predecessor's successor pointer are both set to the new node. Finally, the cursor is updated to point at the current node, and the list size is incremented. You should execute this code by hand on a few example lists to see how it works; it works best to draw a picture of the list like the one in Figure 2 and modify it as you run through the code. Be sure to try it on an empty list to see how the dummy node works in this case (note that when the list is empty, the dummy node pointers point to the dummy node itself, `cursorIndex` is -1 and `cursorPtr` points at the dummy node).

After setting the cursor to the deleted node, the `delete()` method saves its `item` value to be returned later. Removing the deleted node requires setting the successor node's predecessor pointer to the cursor node's predecessor, and setting the predecessor node's successor pointer to the cursor node's successor. To finish up, the method sets `cursorPtr` to what will now be the i^{th} node, decrements the list size, and returns the value stored in the deleted node. Again, you will understand this code better if you run by hand on a few examples.

9.7 SUMMARY AND CONCLUSION

The contiguous implementation of lists is easy to program and efficient for element access, but slow for element insertion and removal. The linked implementation is considerably more difficult to program, and can be slow if operations must always locate nodes by walking down the list from its head, but if double links and a cursor are used, list operations can be quite fast across the board. Using a dummy node simplifies the code. The contiguous implementation is preferable for lists that don't change often but must be accessed quickly, while the linked implementation is better when these characteristics don't apply.

9.8 REVIEW QUESTIONS

1. Does it matter where list element numbering begins?
2. What does the list ADT *index*(*e*) function return when *e* is not in the list? What does the `index()` method in the `List` interface do in a such a situation?
3. What is a cursor?
4. Under what conditions does the contiguous implementation of the list ADT not perform well?
5. What advantage does a doubly-linked list provide over a singly-linked list?
6. What advantage does a circular doubly-linked list provide over a non-circular list?
7. What advantage does a cursor provide in the linked implementation of lists?
8. In an application where long lists are changed infrequently but access to the middle of the lists are common, would a contiguous or linked implementation be better?

9.9 EXERCISES

1. Do we really need iterators for lists? Explain why or why not.
2. Would it be worthwhile to maintain a cursor for a contiguously implemented list? Explain why or why not.
3. What should happen if a precondition of a `List` operation is violated?
4. In the `List` interface, why does the precondition for the `insert()` operation differ from the preconditions for the `delete()`, `get()`, and `put()` operations?
5. A `ListIterator` is a kind of `Iterator` that (a) allows a client to start iteration at the end of list and go through it backwards, (b) change the direction of iteration during traversal, and (c) obtain the index as well as the value of the current element. Write an interface for `ListIterator` that is a sub-interface of `Iterator`.
6. Write a `List<T>` interface in Java. Specify that its methods throw either an `IllegalStateException` or an `IndexOutOfBoundsException` where appropriate.
7. Write an `ArrayList<T>` class in Java implementing the `List<T>` interface from the previous exercise. Use a dynamic array so that the list never becomes full. Note that you will also have to create an `ArrayListIterator<T>` class that implements the `Iterator<T>` interface that is returned by the `iterator()` method in the `Collection<T>` interface.
8. Write an `ArrayListListIterator<T>` class in Java to go along with the code in the previous exercise. Add a `listIterator()` method to the `ArrayList<T>` class that creates and returns a new `ArrayListListIterator<T>` instance.
9. Write a `LinkedList<T>` class in Java implementing the `List<T>` interface that uses a singly-linked list, no reference to the end of the list, a dummy first node, and a cursor.

10. Write a `LinkedListListIterator<T>` class to go along with the `LinkedList<T>` type in the previous exercise.
11. Write a `DoublyLinkedList<T>` class in Java implementing the `List<T>` interface that uses a doubly-linked circular list, no reference to the end of the list, no dummy first node, and a cursor.

9.10 REVIEW QUESTION ANSWERS

1. It does not matter where list element numbering begins: it may begin at any value. However, it is usual in computing to start at zero, and in everyday life to start at one, so one of these values is preferable.
2. The list ADT *index*(*e*) operation cannot return an index when *e* is not in the list. Its result is undefined in this case. The `index()` operation in the `List` interface returns -1 to indicate that `e` is not in the collection.
3. A cursor is a variable marking a location in a data structure. In the case of a `List`, a cursor is a data structure marking a particular element in the `List`. For an `ArrayList`, a cursor might be simply an index. For a `LinkedList`, it is helpful for the cursor to hold both the index and a reference to the node where the item is stored.
4. A contiguous implementation of the list ADT does not perform well when the list is long and is often changed near its beginning. Every change near the beginning of a long contiguously implemented list requires that many list elements be copied up or down the list, which is expensive.
5. A doubly-linked list makes it faster to move from node to node than in a singly-linked list, which can speed up the most expensive part of linked list operations: finding the nodes in the list where the operation must do its job.
6. A circular doubly-linked list makes it possible to follow links quickly from the list head to the end of the list. This can only be done in a non-circular list if a reference to the end of the list is maintained.
7. A cursor helps speed up linked list operations by often making it faster to get to the nodes where the operations must do their work. Even in a circular doubly-linked list, it is expensive to get to the middle of the list. If a cursor is present and it ends up near the middle of a list after some list operations, then it can be used to get to a node in the middle of the list more quickly.
8. In an application where long lists are changed infrequently but are accessed near their middle often, a contiguous implementation will likely be better than a linked implementation because access to the middle of a contiguously implemented list (no matter how long) is instantaneous, while access to the middle of a linked list will almost always be slower, and could be extremely slow (if link following must begin at one end of the list).

10 ANALYZING ALGORITHMS

10.1 INTRODUCTION

We have so far been developing algorithms in implementing ADTs without worrying too much about how good the algorithms are, except perhaps to point out in a vague way that certain algorithms will be more or less efficient in certain circumstances than others. We have not considered in any rigorous and careful way how efficient our algorithms are in terms of how much work they need to do and how much memory they consume; we have not done a careful algorithm analysis.

> **Algorithm analysis**: The process of determining, as precisely as possible, how much of various resources (such as time and memory) an algorithm consumes when it executes.

In this chapter we will lay out an approach for analyzing algorithms and demonstrate how to use it on several simple algorithms. We will mainly be concerned with analyzing the amount of work done by algorithms; occasionally we will consider how much memory they consume as well.

10.2 MEASURING THE AMOUNT OF WORK DONE

An obvious measure of the amount of work done by an algorithm is the amount of time the algorithm takes to do some task. Before we get out our stopwatches, however, we need to consider several problems with this approach.

To measure how much time an algorithm takes to run, we must code it up in a program. This introduces the following difficulties:

- A program must be written in a programming language. How can we know that the language or its compiler or interpreter have not introduced some factors that artificially increase or decrease the running time of the algorithm?
- The program must run on a machine under the control of an operating system. Machines differ in their speed and capacity, and operating systems may introduce delays; other processes running on the machine may interfere with program timings.
- Programs must be written by programmers; some programmers write very fast code and others write slower code.

Without finding some way to eliminate these confounding factors, we cannot have trustworthy measurements of the amount of work done by algorithms—we will only have measurements of the running times of various programs written by particular programmers in particular languages run on certain machines with certain operating systems supporting particular loads.

In response to these difficulties, we begin by abandoning direct time measurements of algorithms altogether, instead focussing on algorithms in abstraction from their realization in programs written by programmers to run on particular machines running certain operating systems. This immediately eliminates most of the problems we have considered, but it leads to the question: if we can't measure time, what can we measure?

Another way to think about the amount of work done by an algorithm is to consider how many operations the algorithm executes. For example, consider the subtraction algorithm that elementary children learn. The input comes in the form of two numbers written one above the other. The algorithm begins by checking whether the value in the units column of the bottom number is greater than the value in the units column of the top number (a comparison operation). If the bottom number is greater, a borrow is made from the tens column of the top number (a borrow operation). Then the bottom value is subtracted from the top values and the result written beneath the bottom number (a subtraction operation). These steps are repeated for the tens column, then the hundreds column, and so forth, until the entire top number has been processed. For example, subtracting 284 from 305 requires three comparisons, one borrow, and three subtractions, for a total of seven operations.

In counting the number of operations required to do this task, you probably noticed that the number of operations is related to the size of the problem: subtracting three digit numbers requires between six and eight operations (three comparison, three subtractions, and zero to two borrows), while subtracting nine digit numbers requires between 18 and 26 operations (nine comparisons, nine subtractions, and zero to eight borrows). In general, for n digit numbers, between $2n$ and $3n$-1 operations are required.

How did the algorithm analysis we just did work? We simply figured out how many operations were done in terms of the size of the input to the algorithm. We will adopt this general approach for deriving measure of work done by an algorithm:

> *To analyze the amount of work done by an algorithm, produce measures that express a count of the operations done by an algorithm as a function of the size of the input to the algorithm.*

10.3 THE SIZE OF THE INPUT

How to specify the size of the input to an algorithm is usually fairly obvious. For example, the size of the input to an algorithm that searches a list will be the size of the list, because it is obvious that the size of the list, as opposed to the type of its contents, or some other characteristic, is what determines how much work an algorithm to search it will do. Likewise for algorithms to sort a list. An algorithm to raise b to the power k (for some constant b) obviously depends on k for the amount of work it will do.

10.4 WHICH OPERATIONS TO COUNT

In most cases, certain operations are done far more often than others by an algorithm. For example, in searching and sorting algorithms, although some initial assignment and arithmetic operations are done, the operations that are done by far the most often are loop control variable increments, loop control variable comparisons, and key comparisons. These are (usually) each done approximately the same number of times, so we can simply count key comparisons as a stand-in for the others. Thus counts of key comparisons are traditionally used as the measure of work done by searching and sorting algorithms.

This technique is also part of the standard approach to analyzing algorithms: one or perhaps two *basic operations* are identified and counted as a measure of the amount of work done.

> **Basic operation**: An operation fundamental to an algorithm used to measure the amount of work done by the algorithm.

As we will see when we consider function growth rates, not counting initialization and bookkeeping operations (like loop control variable comparison and incrementing operations), does not affect the overall efficiency classification of an algorithm.

10.5 BEST, WORST, AND AVERAGE CASE COMPLEXITY

Algorithms don't always do the same number of operations on every input of a certain size. For example, consider the following algorithm to search an array for a value.

```
boolean find(int key, int[] array) {
 for (int i = 0; i < array.length; i++) {
  if (key == array[i]) return true;
 }
 return false;
}
```

Figure 1: An Arra Searching Algorithm

The measure of the size of the input is the array size, which we will label n. Let us count the number of comparisons between the key and the array elements made in the body of the loop. If the key is the very first element of the array, then the number of comparisons is only one; this is the *best case complexity*. We use $B(n)$ to designate the best case complexity of an algorithm on input of size n, so in this case $B(n) = 1$.

In contrast, suppose that the key is not present in the array at all, or is the last element in the array. Then exactly n comparisons will be made; this is the *worst case complexity*, which we designate $W(n)$, so for this algorithm, $W(n) = n$.

Sometimes the key will be in the array, and sometimes it will not. When it is in the array, it may be at any of its n locations. The number of operations done by the algorithm depends on which of these possibilities obtains. Often we would like to characterize the behavior of an algorithm over a wide range of possible inputs, thus producing a measure of its *average case complexity*, which we designate $A(n)$. The difficulty is that it is often not clear what constitutes an "average" case. Generally an algorithm analyst makes some reasonable assumptions about the probabilities of various inputs and then derives a measure for the average case complexity. For example, suppose we assume that the key is in the array, and that it is equally likely to be at any of the n array locations. Then the probability that it is in position i, for $0 \le i < n$, is $1/n$. If the key is at location zero, then the number of comparisons is one; if it is at location one, then the number of comparisons is two; in general, if the key is at position i, then the number of comparisons is $i+1$. Hence the average number of comparisons is given by the following equation.

$$A(n) = \sum_{i=0 \text{ to } n-1} 1/n \cdot (i+1) = 1/n \cdot \sum_{i=1 \text{ to } n} i$$

You may recall from discrete mathematics that the sum of the first n natural numbers is $n(n+1)/2$, so $A(n) = (n+1)/2$. In other words, if the key is in the array and is equally likely to be in any location, then on average the algorithm looks at about half the array elements before finding it, which makes sense.

Lets consider what happens when we alter our assumptions about the average case. Suppose that the key is not in the array half the time, but when it is in the array, it is equally likely to be at any location. Then the probability that the key is at location i is $1/2 \cdot 1/n = 1/2n$. In this case, our equation for $A(n)$ is the sum of the probability that the key is not in the list (1/2) times the number of comparisons made when the key is not in the list (n), and the sum of the product of the probability that the key is in location i times the number of comparisons made when it is in location i:

$$A(n) = n/2 + \sum_{i=0 \text{ to } n-1} 1/2n \cdot (i+1) = n/2 + 1/2n \cdot \sum_{i=1 \text{ to } n} i = n/2 + (n+1)/4 = (3n+1)/4$$

In other words, if the key is not in the array half the time, but when it is in the array it is equally likely to be in any location, then the algorithm looks about three-quarters of the way through the array on average. Said another way, it looks all the way through the array half the time (when the key is absent), and half way through the array half the time (when the key is present), so overall it looks about three quarters of the way through the array. This makes sense too.

We have now completed an analysis of the algorithm above, which is called sequential search.

> **Sequential search**: An algorithm that looks through a list from beginning to end for a key, stopping when it finds the key.

Sometimes a sequential search returns both an indication of whether the key is in the list and, if its is, its index as well—the `index()` operation in our `List` interface is intended to embody such a version of the sequential search algorithm.

Not every algorithm has behavior that differs based on the content of its inputs—some algorithms behave the same on inputs of size n in all cases. For example, consider the algorithm in Figure 2.

```
int max(int[] array) {
 if (array.length == 0)
  throw new IllegalArgumentException();
 int m = array[0];
 for (index = 1; index < array.length; index++)
  if (m < array[index]) m = array[index];
 return m
}
```

Figure 2: Maximum-Finding Algorithm

This algorithm, the *maximum-finding algorithm*, always examines every element of the array after the first (as it must, because the maximum value could be in any location). Hence on an input of size n (the array size), it always makes n-1 comparisons (the basic operation we are counting). The worst, best, and average case complexity of this algorithm are all the same. The *every-case complexity* of an algorithm is a the number of basic operations performed by the algorithm when it does the same number of basic operations on all inputs of size n. We will use $C(n)$ to designate every-case complexity, so for the maximum-finding algorithm, $C(n) = n$-1.

10.6 SUMMARY AND CONCLUSION

We define the various kinds of complexity we have discussed as follows.

Computational complexity: The time (and perhaps the space) requirements of an algorithm.

Every-case complexity *C(n)*: The number of basic operations performed by an algorithm as a function of the size of its input n when this value is the same for any input of size n.

Worst case complexity *W(n)*: The maximum number of basic operations performed by an algorithm for any input of size n.

Best case complexity *B(n)*: The minimum number of basic operations performed by an algorithm for any input of size n.

Average case complexity *A(n)*: The average number of basic operations performed by an algorithm for all inputs of size n, given assumptions about the characteristics of inputs of size n.

We can summarize the process for analyzing an algorithm as follows:

1. Choose a measure for the size of the input.
2. Choose a basic operation to count.
3. Determine whether the algorithm has different complexity for various inputs of size n; if so, then derive measures for $B(n)$, $W(n)$, and $A(n)$ as functions of the size of the input; if not, then derive a measure for $C(n)$ as a function of the size of the input.

We will consider how to do step 3 in more detail later.

10.7 REVIEW QUESTIONS

1. Give three reasons why timing programs is insufficient to determine how much work an algorithm does.
2. How is a measure of the size of the input to an algorithm determined?
3. How are basic operations chosen?
4. Why is it sometimes necessary to distinguish the best, worst and average case complexities of algorithms?
5. Does best case complexity have anything to do with applying an algorithm to smaller inputs?

10.8 EXERCISES

1. Determine measures of the size of the input and suggest basic operations for analyzing algorithms to do the following tasks.
   a) Finding the average value in a list of numbers.
   b) Finding the number of 0s in a matrix.
   c) Searching a text for a string.
   d) Finding the shortest path between two nodes in a network
   e) Finding a way to color the countries in a map so that no adjacent countries are the same color.
2. Write a Java sequential search method that returns the index of the key if it is present, and -1 otherwise.
3. Consider the Java code below.

```
int maxCharSequence(String s) {
  if (s.length() == 0) return 0;
  int maxLength = 0;
  int thisLength = 1;
  int lastChar = s.charAt(0);
  for (int thisIdx = 1; thisIdx < s.length(); thisIdx++) {
     if (s.charAt(thisIdx) == lastChar) thisLength++;
     else {
       if (maxLength < thisLength) maxLength = thisLength;
       thisLength = 1;
     }
     lastChar = s.charAt(thisIdx);
  }
  if (maxLength < thisLength) maxLength = thisLength;
  return maxLength;
}
```

a) What does this algorithm do?
b) In analyzing this algorithm, what would be a good measure of input size?
c) What would be a good choice of basic operation?
d) Does this algorithm behave differently for different inputs of size *n*?
e) What are the best and worst case complexities of this algorithm?

4. Compute the average case complexity of sequential search under the assumption that the likelihood that the key is in the list is *p* (and hence the likelihood that it is not in the list is 1-*p*), and that if in the list, the key is equally likely to be at any location.

10.9 REVIEW QUESTION ANSWERS

1. Timing depends on actual programs running on actual machines. The speed of a real program depends on the skill of the programmer, the language the program is written in, the efficiency of the code generated by the compiler or the efficiency of program interpretation, the speed of the hardware, and the ability of the operating system to accurately measure the CPU time consumed by the program. All of these are confounding factors that make it very difficult to evaluate algorithms by timing real programs.
2. The algorithm analyst must choose a measure that reflects aspects of the input that most influence the behavior of the algorithm. Fortunately, this is usually not hard to do.

3. The algorithm analyst must choose one or more operations that are done most often during execution of the algorithm. Generally, basic operations will be those used repeatedly in inner loops. Often several operations will be done roughly the same number of times; in such cases, only one operation need be counted (for reasons to be explained in the next chapter about function growth rates).
4. Algorithms that behave differently depending on the composition of inputs of size n can do dramatically different amounts of work, as we saw in the example of sequential search. In such cases, a single value is not sufficient to characterize an algorithm's behavior, and so we distinguish best, worst, and average case complexities to reflect these differences.
5. Best case complexity has to do with the behavior of an algorithm for inputs of a given size, not with behavior as the size of the input varies. The complexity functions we produce to count basic operations are already functions of the size of the input. Best, worst, average, and every-case behavior are about differences in behavior given input of a certain size.

11 FUNCTION GROWTH RATES

11.1 INTRODUCTION

We have set up an approach for determining the amount of work done by an algorithm based on formulating functions expressing counts of basic operations in terms of the size of the input to the algorithm. By concentrating on basic operations, our analysis framework introduces a certain amount of imprecision. For example, an algorithm whose complexity is $C(n) = 2n\text{-}3$ may actually run slower than an algorithm whose complexity is $C(n) = 12n+5$, because uncounted operations in the former may slow its actual execution time. Nevertheless, both of these algorithms would surely run much more quickly than an algorithm whose complexity is $C(n) = n^2$ as n becomes large; the running times of the first two algorithms are much closer to each other than they are to the third algorithm.

In comparing the efficiency of algorithms, we are more interested in big differences that manifest themselves as the size of the input becomes large than we are in small differences in running times that vary by a constant or a multiple for inputs of all sizes. The theory of the *asymptotic growth rate* of functions, also called the *order of growth* of functions, provides a basis for partitioning algorithms into groups with equivalent efficiency, as we will now see.

11.2 DEFINITIONS AND NOTATION

Our goal is to classify functions into groups by growth rates. Our notation describes groups of functions using a reference function (which may be any function). The three groups we distinguish are the following.

> $O(f)$ (pronounced *big-oh of f*)—The set of functions that grow no faster than $f(n)$ (that is, those that grow more slowly than $f(n)$ or at the same rate as $f(n)$). For example, $8n+5 \in O(n)$, $8n+5 \in O(n^2)$, and $6n^2+23n\text{-}14 \in O(4n^2\text{-}18n+65)$.
>
> $\Omega(f)$ (pronounced *big-omega of f*)—The set of functions that grow at least as fast as $f(n)$ (that is, those that grow more quickly than $f(n)$ or at the same rate as $f(n)$). For example, $8n+5 \in \Omega(n)$, $n^2 \in \Omega(8n+5)$, and $6n^2+23n\text{-}14 \in \Omega(4n^2\text{-}18n+65)$.
>
> $\Theta(f)$ (pronounced *big-theta of f*)—The set of functions that grow at the same rate as $f(n)$. For example, $8n+5 \in \Omega(n)$, $n^2 \in \Theta(7n^2\text{-}987)$, and $6n^2+23n\text{-}14 \in \Theta(4n^2\text{-}18n+65)$.

These three groups overlap: the intersection of $O(f)$ and $\Omega(f)$ is $\Theta(f)$.

Formally, let $f(n)$ and $g(n)$ be functions from the natural numbers to the non-negative real numbers.

> **Definition**: The function g is in the set $O(f)$, denoted $g \in O(f)$, if there exist some positive constant c and non-negative integer n_0 such that
> $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$

> **Definition**: The function g is in the set $\Omega(f)$, denoted $g \in \Omega(f)$, if there exist some positive constant c and non-negative integer n_0 such that
> $g(n) \geq c \cdot f(n)$ for all $n \geq n_0$

> **Definition**: The function g is in the set $\Theta(f)$, denoted $g \in \Theta(f)$, if both $g \in O(f)$ and $g \in \Omega(f)$.

In other words, g is in $O(f)$ if at some point $g(n)$ is never greater than some multiple of $f(n)$; in this sense f is a sort of upper bound for g. Similarly, g is in $g \in \Omega(f)$ if at some point $g(n)$ is never less than some multiple of $f(n)$; hence f is a sort of lower bound for g. Finally, if $g \in \Theta(f)$, then no matter how large the argument to these functions grow, their values will always be within a constant factor of each other—this is the sense in which they grow at the same rate.

It is important to realize the huge difference between the growth rates of functions in sets with different orders of growth. The table below shows the values of functions in sets with increasing growth rates. Blank spots in the table indicate absolutely enormous numbers. (The function lg n is $\log_2 n$.)

n	lg n	n	n lg n	n^2	n^3	2^n	n!
10	3.3	10	33	100	1000	1024	3,628,800
100	6.6	100	660	10,000	1,000,000	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
1000	10	1000	10,000	1,000,000	10^9		
10,000	13	10,000	130,000	10^8	10^{12}		
100,000	17	100,000	1,700,000	10^{10}	10^{15}		
1,000,000	20	1,000,000	$2 \cdot 10^7$	10^{12}	10^{18}		

Table 1: Values of Functions of Different Orders of Growth

As this table suggests, algorithms whose complexity is characterized by functions in the first several columns are quite efficient, and we can expect them to complete execution quickly for even quite large inputs. Algorithms whose complexity is characterized by functions in the last several columns must do enormous amounts of work even for fairly small inputs, and for large inputs, they simply will not be able to finish execution before the end of time, even on the fastest possible computers.

11.3 ESTABLISHING THE ORDER OF GROWTH OF A FUNCTION

When confronted with the question of whether some function g is in $O(f)$, $\Omega(f)$, or $\Theta(f)$, we can use the definitions directly to decide, but there is an easier way embodied in the following theorem.

> Theorem: (1) $g \in O(f)$ iff $\lim_{n\to\infty} g(n)/f(n) = c$, for $c \geq 0$;
> (2) $g \in \Omega(f)$ iff $\lim_{n\to\infty} g(n)/f(n) = c$, for $c > 0$, or $\lim_{n\to\infty} g(n)/f(n) = \infty$; and
> (3) $g \in \Theta(f)$ iff $\lim_{n\to\infty} g(n)/f(n) = c$, for $c > 0$.

For example, to show that $3n^2+2n-1$ is in $O(n^2)$, $\Omega(n^2)$,or $\Theta(n^2)$,we need only determine $\lim_{n\to\infty} (3n^2+2n-1)/n^2$:

$$\begin{aligned}\lim_{n\to\infty} (3n^2+2n-1)/n^2 &= \lim_{n\to\infty} 3n^2/n^2 + \lim_{n\to\infty} 2n/n^2 - \lim_{n\to\infty} 1/n^2 \\ &= \lim_{n\to\infty} 3 + \lim_{n\to\infty} 2/n - \lim_{n\to\infty} 1/n^2 = 3\end{aligned}$$

Because this limit is a constant greater than 0, we can conclude that $3n^2+2n-1 \in O(n^2)$ and that $3n^2+2n-1 \in \Theta(n^2)$.

A theorem that is very useful in solving limit problems is L'HÔpital's Rule:

Theorem: If $\lim_{n->\infty} f(n) = \lim_{n->\infty} g(n) = \infty$, and the derivatives f' and g' exist, then $\lim_{n->\infty} f(n)/g(n) = \lim_{n->\infty} f'(n)/g'(n)$.

To illustrate the use of L'HÔpital's Rule, lets determine the relative orders of groth of n^2 and $n \lg n$. First note that $\lim_{n->\infty} n^2 = \lim_{n->\infty} n \lg n = \infty$, and that the first derivatives of both of these functions exist, and L'HÔpital's Rule applies.

$$\begin{aligned} \lim_{n->\infty} n^2/(n \lg n) &= \lim_{n->\infty} n/(\lg n) \\ &= \lim_{n->\infty} 1/((\lg e)/n) \text{ (using L'HÔpital's Rule)} \\ &= \lim_{n->\infty} n/(\lg e) = \infty \end{aligned}$$

Because this limit is infinite, we know that $n^2 \in \Omega(n \lg n)$ but that $n^2 \notin \Theta(n \lg n)$; in other words, we know that n^2 grows faster than $n \lg n$.

11.4 APPLYING ORDERS OF GROWTH

In our discussion of complexity we determined that for sequential search, $W(n) = (n+1)/2$, $B(n) = 1$, and $A(n) = (3n+ 1)/4$. Clearly, these functions are all in $O(n)$; we say that sequential search is a *linear* algorithm because its worst case is in $\Theta(n)$. Similarly, we determined that for the maximum-finding algorithm, $C(n) = n$-1. This function is also in $\Theta(n)$, so this is also a linear algorithm. We will soon see algorithms whose complexity is in sets with higher orders of growth.

11.5 SUMMARY AND CONCLUSION

Our algorithm analysis approach has three steps:

1. Choose a measure for the size of the input.
2. Choose a basic operation to count.
3. Determine whether the algorithm has different complexity for various inputs of size n; if so, then derive measures for $B(n)$, $W(n)$, and $A(n)$ as functions of the size of the input; if not, then derive a measure for $C(n)$ as a function of the size of the input.

We now add a fourth step:

4. Determine the order of growth of the complexity measures for the algorithm.

Usually this last step is quite simple. In evaluating an algorithm, we are often most interested in the order of its worst case complexity or (if there is no worst case) basic complexity, because this places an upper bound on the behavior of the algorithm: though it may perform better, we know it cannot perform worse than this. Sometimes we are also interested in average case complexity, though the assumptions under which such analyses are done may sometimes not be very plausible.

11.6 REVIEW QUESTIONS

1. Why is the order of growth of functions pertinent to algorithm analysis?
2. If a function g is in $O(f)$, can f also be in $O(g)$?
3. What function is lg n?
4. Why is L'HÔpital's Rule important for analyzing algorithms?

11.7 EXERCISES

1. Some algorithms have complexity lg lg n (that is lg (lg n)). Make a table like Table 1 above showing the rate of growth of lg lg n as n becomes larger.
2. Show that $n^3+n-4 \notin O(2n^2-3)$.
3. Show that lg $2^n \in \Theta(n)$.
4. Show that n lg $n \in O(n^2)$ and that $n^2 \in \Omega(n \text{ lg } n)$.
5. Show that if $a,\ b \geq 0$ and $a \leq b$, then $n^a \in O(n^b)$.

11.8 REVIEW QUESTION ANSWERS

1. The order of growth of functions is pertinent to algorithm analysis because the amount of work done by algorithms whose complexity functions have the same order of growth is not very different, while the amount of work done by algorithms whose complexity functions have different orders of growth is dramatically different. The theory of the order of growth of functions provides a theoretical framework for determining significant differences in the amount of work done by algorithms.
2. If g and f grow at the same rate, then $g \in O(f)$ because g grows no faster than f, and $f \in O(g)$ because f grows no faster than g. For any functions f and g with the same order of growth, $f \in O(g)$ and $g \in O(f)$.
3. The function lg n is $\log_2 n$. that is, the logarithm base two of n.

4. L'HÔpital's Rule is important for analyzing algorithms because if makes it easier to compute the limit of the ratio of two functions of n as n goes to infinity, which is the basis for determining their comparative growth rates. For example, it is not clear what the value of $\lim_{n\to\infty}(\lg n)^2/n$ is. Using L'HÔpital's Rule twice to differentiate the numerators and denominators, we get

$$\lim_{n\to\infty}(\lg n)^2/n = \lim_{n\to\infty}(2 \lg e \cdot \lg n)/n = \lim_{n\to\infty}(2 (\lg e)^2)/n = 0.$$

This shows that $(\lg n)^2 \in O(n)$ and that $(\lg n)^2 \notin \Theta(n)$.

12 AMORTIZED ANALYSIS

12.1 INTRODUCTION

The sort of algorithm analysis we have discussed so far will be applied to many algorithms in the next several chapters. There is another sort of analysis that we will touch on briefly in this chapter and apply to a few examples, but will not pursue further in the remainder of the book. Rather than looking at the time or space efficiency of a particular algorithm, we can examine the behavior of the operations in a data type to understand the average time or space required to use the data type. Note that this sort of analysis considers several operations (hence several algorithms) jointly rather than individually as we have been doing. This is important because it is not uncommon for some operations in a data type to be quite fast while others are quite slow. If we draw conclusions about the worst case behavior of the operations by considering only the slowest operations in our analysis, then we may be misled about the true cost of using the data type. *Amortized analysis* looks at the overall performance of the operations in a data type by considering arbitrary sequences of operations that may include fast and slow operations. This often leads to much different conclusions about the cost of using the data type.

> **Amortized analysis**: Determining the average time of operations in the worst case by considering the time needed to perform an arbitrary sequence of n operations, and then dividing this total by n.

There are several techniques for doing amortized analysis; here we consider the most basic technique, called *aggregate analysis*.

12.2 A STACK DATA TYPE

To illustrate amortized analysis, suppose we have a stack data type whose implicit-receiver method set consists of the operations *push*(), *pop*(), *size*(), and *multiPop*(), where *multiPop*(k) pops k elements from the stack, or empties the stack if it has fewer than k elements. In analyzing this data type, suppose we take *push*() and *pop*() as basic operations. Then *multiPop*(k) consumes *minimum*(k, *size*()) basic operations (because it does this many *pop*() operations). The complexity, $C(n)$, of both *pop*() and *push*() is 1, and the worst case complexity, $W(n)$, of *multipPop*(k) is m, where m is *minimum*(k, *size*()). If we consider a sequence of n operations from this data type in the worst case, we may have a stack size of n (because we may have n *push*() operations). The worst case of a single operation is thus n (because it may be *multiPop*(k) with $k \geq n$). But a sequence of n operations with worst case n has worst case n^2, so on average a single operation in this sequence has complexity n. Hence the average worst case complexity of the operations in this data type is $O(n)$.

When we use this data type in a program, not every operation call can be *multiPop*(), so this estimate is too pessimistic. How can we characterize the overall complexity of using this data type in a program more accurately? We proceed as follows using amortized analysis.

Consider an arbitrary sequence of n operations that begins with an empty stack and only contains operations whose preconditions are met (in other words, we never try to apply *pop*() or *multiPop*() to an empty stack). Note first that if the sequence contains only *push*() and *pop*() operations, then it must have exactly n basic operations. Second, we can replace any *multiPop*(k) operation with *minimum*(k, *size*()) *pop*() operations to achieve the same result, thus expanding the original sequence of n operations to some longer sequence of m *push*() and *pop*() operations. How large can m be? A little reflection should reveal that the maximum value of m occurs when n-1 *push*() operations are followed by a single *multiPop*(n-1) operation. This results in $m = n\text{-}1 + n\text{-}1 = 2n\text{-}2 < 2n$ basic operations. Hence the worst-case time for any sequence of n operations of this data type less than $2n$, and therefore the average time for each operation is less than 2. Hence the average cost of the operations in this data type is less than 2, which is $\Theta(1)$. This reveals that despite the worst case complexity of one of the operations in the data type being $O(n)$, using the operations in this data type will incur only a constant cost on average in the worst case.

12.3 DYNAMIC ARRAYS

We have discussed how a dynamic arrays expands= as necessary to accommodate arbitrarily many insertions. Implementing dynamic arrays must be done using statically allocated blocks of memory. When a dynamic array is expanded, a new, larger array must be allocated, the contents of the old array must be copied into the new array, the space for the old array must be freed, and finally the new value must be added to the now expanded array. How efficient is the use of dynamic arrays?

We begin by specifying a dynamic array type. Suppose dynamic array values begin as empty arrays (with size 0) and have the following implicit-receiver method set.

> *size*()—Return the number of values stored in the array.
> *get*(i)—Return the value stored at location i. The precondition of this operation is that $0 \le i < size()$.
> *put*(i,v)—Replace the value at location i with value v. The precondition of this operation is that $0 \le i \le size()$. Note that if $i = size()$, then the array is expanded.

Suppose our array expansion policy is to increase the number of slots in the array by just as much as is needed, which with this type means just one array slot at a time. Let us do an amortized analysis to see how efficient this policy is.

We will take the basic operation that we count to be a value copy operation. Hence the complexity of *size*() is 0, the complexity of *get*() is 1, and the complexity of *put*() is either 1 (if $0 \le i < size()$), or *size*()+1 (if $i = size()$).

Consider an arbitrary sequence of n operations of this data type that starts with an empty dynamic array. The worst case sequence consists of only *put*() operations, each at location *size*(). This results in $1+2+3+ \ldots + n = n(n+1)/2$ copy operations. Dividing by n to get the average cost, we arrive at $(n+1)/2$, which is $\Theta(n)$. Thus, on average, using the operations of this data type is $\Theta(n)$ in the worst case, where n is the number of operations invoked. This is not a very attractive alternative because all three array operations for a static array are $\Theta(1)$. Can we do better?

Let us change the dynamic array expansion policy to double the capacity of the array when it is expanded—note that we must now distinguish between size (the number of values stored) and capacity (the number of slots in the array). The array begins with capacity 0, and

when expanded its capacity becomes 1, then 2, then 4, then 8, and so on. Thus the array only needs to be expanded when a *put*() occurs with $i = size()$ when *size*() is a power of 2.

Let us consider an arbitrary sequence of n dynamic array operations. Once again the worst case is a sequence of *put*() operations with $i = size()$. However in this case the number of copy operations is quite different. There is always a copy for the inserted value v, yielding n copies for the sequence. But the number of copies required when the array is expanded is much less: $1+2+4+ \ldots + 2^{\lfloor \lg (n-1) \rfloor} = 2^{\lfloor \lg (n-1) \rfloor + 1} - 1 \leq 2(n-1) - 1 < 2n$. Combining the copies for the new value and those for expansion, we see that the sequence requires fewer than $3n$ copies for the sequence. Hence each operation requires less than 3 copy operations on average in the worst case, which is in $\Theta(1)$.

This result compares favorably with the complexity of static arrays. Amortized analysis reveals that the policy of expanding the array only just as much as necessary is not efficient, but that doubling the capacity of the array when it is expanded provides good average performance, even in the worst case.

12.4 SUMMARY AND CONCLUSION

Amortized analysis is a technique for analyzing the complexity of the operations of a data type together, thus providing results about the average complexity of operations in the worst case. Amortized analysis often provides less pessimistic, and hence more accurate, assessments of the cost of using some data type in a program. It also may provide insights into algorithm design alternatives, allowing us to pick the most effective implementations of data type operations. Although we will not discuss it further in this book, amortized analysis is a powerful technique often used in more advanced data structures and algorithms texts.

12.5 REVIEW QUESTIONS

1. In the stack data type example, why does a sequence of n *push*() and *pop*() operations have complexity n?
2. When expanding a dynamic array with the policy that only one slot is added with each expansion, why is it not necessary to distinguish size and capacity?
3. Why do all three array operations have complexity $\Theta(1)$ for static arrays?
4. For the second dynamic array expansion policy, why is the number of copies performed $1+2+4+ \ldots + 2^{\lfloor \lg (n-1) \rfloor}$?

12.6 EXERCISES

1. Write pseudocode to implement the *put*(*i*,*v*) operation using the first expansion policy for dynamic arrays discussed in the text. Include code for doing the expansion.
2. Write pseudocode to implement the *put*(*i*,*v*) operation using the second expansion policy for dynamic arrays discussed in the text. Include code for doing the expansion.
3. Make a table showing the number of expansion copies made under the second expansion policy for dynamic arrays when *n put*(*size*(),*v*) operations are performed. The columns of your table should be the value of n, the capacity before the operation, the capacity after the operation, and the number of copies does during the operation. Your table should show values of n from 1 to 10.
4. Suppose we add a mechanism to shrink dynamic arrays using an operation *delete*() that removes the last value in an array (the one at location *size*()-1). Suppose further than when the size of an array reaches half its capacity when a value is removed, then its capacity is halved. Halving the capacity involves allocating a new array half the size of the old array, copying the old values into it, and freeing the old, larger array. Starting with an empty array. Determine a sequence of operations, starting with an empty array, that cause the average operation to be $O(n)$ in the worst case.

12.7 REVIEW QUESTION ANSWERS

1. In the stack data type example, the basic operations we have chosen to count are *push*() and *pop*(). Hence any sequence of n *push*() and *pop*() operations must have complexity n.
2. When expanding a dynamic array with the policy that only one slot is added in each expansion, the size and capacity are always identical. Hence it is not necessary (and perhaps is confusing) to distinguish size and capacity. However, when the second expansion policy is used, the number of slots in the array doubles when the number of values in the array increases by only one, so we must distinguish between capacity (the number of slots), and size (the number of values in the array).
3. In a static array, the only times values are copied are when a value is returned from the array (a *get*() operation), which requires one value copy, or when a value is placed into a slot in the array (a *put*() operation), which also requires one value copy. The *size*() operation does not copy any data, so its complexity is 0. Because all these operations perform a constant number of copies, they all have complexity $\Theta(1)$.
4. For the second dynamic array expansion policy, the capacity of the array is doubled at every expansion. The array begins empty, so on the first expansion its capacity becomes one. No copies from the old array are made in this case because the old array is empty. The next time it is expanded, its capacity goes from one to two; in this case one value is copied from the old array. The next time the array is expanded, its capacity goes from two to four, and the two values from the old array are copied. The next expansion goes from four to eight, and four values are copied from the old array in this case. Clearly, the number of values copied across the whole sequence of *put*() operations is the sum of powers of two. Figuring out how many powers of two are in this summation is a bit tricky. In a sequence of n *put*() operations, expansions occur at every $(i+1)^{\text{th}}$ operation, where i is a power of two, so we add a power of two to this sequence when i-1 is a power of two. The last power of two less than or equal to n-1 is $\lfloor \lg (n\text{-}1) \rfloor$, and hence the last element in the sum of the powers of two is $\lfloor \lg (n\text{-}1) \rfloor$.

13 BASIC SORTING ALGORITHMS

13.1 INTRODUCTION

Sorting is a fundamental and important data processing task.

> **Sorting algorithm**: An algorithm that rearranges records in lists so that they follow some well-defined ordering relation on values of keys in each record.

An *internal* sorting algorithm works on lists in main memory, while an *external* sorting algorithm works on lists stored in files. Some sorting algorithms work much better as internal sorts than external sorts, but some work well in both contexts. A sorting algorithm is *stable* if it preserves the original order of records with equal keys.

Many sorting algorithms have been invented; in this chapter we will consider the simplest sorting algorithms. In our discussion in this chapter, all measures of input size are the length of the sorted lists (slices in the sample code), and the basic operation counted is comparison of list elements (also called *keys*). Our implementations are in Java, and to keep the algorithms simple, they are all framed as algorithms to sort arrays of `int`s. We could generalize the algorithms to sort arrays of other types, but this would distract attention from the essentials of sorting. You can generalize these algorithms for other array types as an exercise.

13.2 BUBBLE SORT

Bubble sort is one of the oldest sorting algorithms. The idea behind it is to make repeated passes through the list from beginning to end, comparing adjacent elements and swapping any that are out of order. After the first pass, the largest element will have been moved to the end of the list; after the second pass, the second largest will have been moved to the penultimate position; and so forth. The idea is that large values "bubble up" to the top of the list on each pass.

A Java implementation of bubble sort appears in Figure 1.

```
void bubbleSort(int[] a) {
 for (int j = a.length-1; 0 < j; j--) {
  for (int i = 0; i < j; i++) {
   if (a[i+1] < a[i]) {
    int tmp = a[i];
    a[i] = a[i+1];
    a[i+1] = tmp;
   }
  }
 }
}
```

Figure 1: Bubble Sort

It should be clear that the algorithm does exactly the same key comparisons no matter what the contents of the array, so we need only consider its every-case complexity.

On the first pass through the data, every element in the array but the last is compared with its successor, so n-1 comparisons are made. On the next pass, one less comparison is made, so n-2 comparisons are made. This continues until the last pass, where only one comparison is made. The total number of comparisons is thus given by the following summation.

$$C(n) = \sum_{i=1 \text{ to } n-1} i = n(n\text{-}1)/2$$

Clearly, $n(n\text{-}1)/2 \in \Theta(n^2)$.

Bubble sort is not very fast. Various suggestions have been made to improve it. For example, a Boolean variable can be set to false at the beginning of each pass through the list and set to true whenever a swap is made. If the flag is false when the pass is completed, then no swaps were done and the array is sorted, so the algorithm can halt. This gives exactly the same worst case complexity, but a best case complexity of only n. The average case complexity is still in $\Theta(n^2)$, however, so this is not much of an improvement.

13.3 SELECTION SORT

The idea behind selection sort is to make repeated passes through the list, each time finding the largest (or smallest) value in the unsorted portion of the list, and placing it at the end (or beginning) of the unsorted portion, thus shrinking the unsorted portion and growing the sorted portion. The algorithm works by repeatedly "selecting" the item that goes at one end of the unsorted portion of the list.

A Java implementation of selection sort appears in Figure 2.

```
void selectionSort(int[] a) {
 for (int j = 0; j < a.length-1; j++) {
  int minIndex = j;
  for (int i = j+1; i < a.length; i++)
   if (a[i] < a[minIndex]) minIndex = i;
  int tmp = a[j];
  a[j] = a[minIndex];
  a[minIndex] = tmp;
 }
}
```

Figure 2: Selection Sort

This algorithm finds the minimum value in the unsorted portion of the list n-1 times and puts it where it belongs. Like bubble sort, it does exactly the same thing no matter what the contents of the array, so we need only consider its every-case complexity.

On the first pass through the list, selection sort makes n-1 comparison; on the next pass, it makes n-2 comparisons; on the third, it makes n-3 comparisons, and so forth. It makes n-1 passes altogether, so its complexity is

$$C(n) = \sum_{i=1 \text{ to } n-1} i = n(n-1)/2$$

As noted before, $n(n-1)/2 \in \Theta(n^2)$.

Although the number of comparisons that selection sort makes is identical to the number that bubble sort makes, selection sort usually runs considerable faster. This is because bubble sort typically makes many swaps on every pass through the list, while selection sort makes only one. Nevertheless, neither of these sorts is particularly fast.

13.4 INSERTION SORT

Insertion sort works by repeatedly taking an element from the unsorted portion of a list and inserting it into the sorted portion of the list until every element has been inserted. This algorithm is the one usually used by people when sorting piles of papers.

A Java implementation of insertion sort appears in Figure 3.

```
void insertionSort(int[] a) {
 for (int j = 1; j < a.length; j++) {
  int element = a[j];
  int i;
  for (i = j; 0 < i && element < a[i-1]; i--)
   a[i] = a[i-1];
  a[i] = element;
 }
}
```

Figure 3: Insertion Sort

A list with only one element is already sorted, so the elements inserted begin with the second element in the array. The inserted element is held in the `element` variable and values in the sorted portion of the array are moved up to make room for the inserted element in the same loop that finds the right place to make the insertion. Once that spot is found, the loop ends and the inserted element is placed into the sorted portion of the array.

Insertion sort does different things depending on the contents of the list, so we must consider its worst, best, and average case behavior. If the list is already sorted, one comparison is made for each of n-1 elements as they are "inserted" into their current locations. So the best case behavior of insertion sort is

$$B(n) = n\text{-}1$$

The worst case occurs when every inserted element must be placed at the beginning of the already sorted portion of the list; this happens when the list is in reverse order. In this case, the first element inserted requires one comparison, the second two, the third three, and so forth, and n-1 elements must be inserted. Hence

$$W(n) = \sum_{i=1 \text{ to } n\text{-}1} i = n(n\text{-}1)/2$$

To compute the average case complexity, let's suppose that the inserted element is equally likely to end up at any location in the sorted portion of the list, as well as the position it initially occupies. When inserting the element with index j, there are j+1 locations where the element may be inserted, so the probability of inserting into each location is $1/(j+1)$. Hence the average number of comparison to insert the element with index j is given by the following expression.

$$
\begin{aligned}
&1/(j+1) + 2/(j+1) + 3/(j+1) + \ldots + j/(j+1) + j/(j+1) \\
&= 1/(j+1) \cdot \sum_{i=1 \text{ to } j} i + j/(j+1) \\
&= 1/(j+1) \cdot j(j+1)/2 + j/(j+1) \\
&= j/2 + j/(j+1) \\
&\approx j/2 + 1
\end{aligned}
$$

The quantity $j/(j+1)$ is always less than one and it is very close to one for large values of j, so we simplify the expression as noted above to produce a close upper bound for the count of the average number of comparisons done when inserting the element with index j. We will use this simpler expression in our further computations because we know that the result will always be a close upper bound on the number of comparisons.

We see that when inserting an element into the sorted portion of the list we have to make comparisons with about half the elements in that portion of the list, which makes sense.

Armed with this fact, we can now write down an equation for the approximate average case complexity:

$$
\begin{aligned}
&A(n) \approx \sum_{j=1 \text{ to } n-1} (j/2 + 1) \\
&\approx \tfrac{1}{2} \sum_{j=1 \text{ to } n-1} j + \sum_{j=1 \text{ to } n-1} 1 \\
&\approx \tfrac{1}{2} (n(n-1)/2) + (n-1) \\
&\approx (n^2 + 3n - 4)/4 \\
&\approx (n+4)(n-1)/4
\end{aligned}
$$

In the average case, insertion sort makes about half as many comparisons as it does in the worst case. Unfortunately, both these functions are in $\Theta(n^2)$, so insertion sort is not a great sort. Nevertheless, insertion sort is quite a bit better than bubble and selection sort on average and in the best case, so it is the best of the three $\Theta(n^2)$ sorting algorithms.

Insertion sort has one more interesting property to recommend it: it sorts nearly sorted lists very fast. A *k-nearly sorted list* is a list all of whose elements are no more than k positions from their final locations in the sorted list. Inserting any element into the already sorted portion of the list requires at most k comparisons in a k-nearly sorted list. A close upper bound on the worst case complexity of insertion sort on a k-nearly sorted list is:

$$W(n) = \sum_{i=1 \text{ to } n-1} k = k \cdot (n-1)$$

Because k is a constant, $W(n)$ is in $\Theta(n)$, that is, insertion sort always sorts a nearly sorted list in linear time, which is very fast indeed.

13.5 SHELL SORT

Shell sort is an interesting variation of insertion sort invented by Donald Shell in 1959. It works by insertion sorting the elements in a list that are h positions apart for some h, then decreasing h and doing the same thing over again until $h = 1$.

A version of Shell sort in Java appears in Figure 4.

```
func ShellSort(a []int) {
 h := 1
 for h < len(a)/9 {
  h = 3*h + 1
 }

 for 0 < h {
  for j := h; j < len(a); j++ {
   element := a[j]
   var i int
   for i = j; h <= i && element < a[i-h]; i -= h {
    a[i] = a[i-h]
   }
   a[i] = element
  }
  h /= 3
 }
}
```

Figure 4: Shell Sort

Although Shell sort has received much attention over many years, no one has been able to analyze it yet! It has been established that for many sequences of values of h (including those used in the code above), Shell sort never does more than $n^{1.5}$ comparisons in the worst case. Empirical studies have shown that it is quite fast on most lists. Hence Shell sort is the fastest sorting algorithm we have considered so far.

13.6 SUMMARY AND CONCLUSION

For small lists of less than a few hundred elements, any of the algorithms we have considered in this chapter are adequate. For larger lists, Shell sort is usually the best choice, except in a few special cases:

- If a list is nearly sorted, use insertion sort;
- If a list contains large records that are very expensive to move, use selection sort because it does the fewest number of data moves (of course, the fast algorithms we study in a later chapter are even better).

Never use bubble sort: it makes as many comparisons as any other sort, and usually moves more data than any other sort, so it is generally the slowest of all.

13.7 REVIEW QUESTIONS

1. What is the difference between internal and external sorts?
2. The complexity of bubble sort and selection sort is exactly the same. Does this mean that there is no reason to prefer one over the other?
3. Does Shell sort have a best case different from its worst case?

13.8 EXERCISES

1. Rewrite the bubble sort algorithm to incorporate a check to see whether the array is sorted after each pass, and to stop processing when this occurs.
2. An alternative to bubble sort is the Cocktail Shaker sort, which uses swaps to move the largest value to the top of the unsorted portion, then the smallest value to the bottom of the unsorted portion, then the largest value to the top of the unsorted portion, and so forth, until the array is sorted.
   a) Write code for the Cocktail Shaker sort.
   b) What is the complexity of the Cocktail Shaker sort?
   c) Does the Cocktail Shaker sort have anything to recommend it (besides its name)?

3. Adjust the selection sort algorithm presented above to sort using the maximum rather than the minimum element in the unsorted portion of the array.
4. Every list of length n is n-nearly sorted. Using the formula for the worst case complexity of insertion sort on a k-nearly sorted list with $k = n$, we get $W(n) = n(n-1)$. Why is this result different from $W(n) = n(n-1)/2$, which we calculated elsewhere?
5. Shell sort is a modified insertion sort, and insertion sort is very fast for nearly sorted lists. Do you think Shell sort would sort nearly sorted lists even faster than insertion sort? Explain why or why not.
6. The sorting algorithms presented in this chapter are written for ease of analysis and so are only suitable for sorting `int` arrays. Rewrite the sorting algorithms so that they can be used for arrays of arbitrary classes that implement the `Comparable<T>` interface. Hint: an example header for an insertion sort method would be `public static <T extends Comparable<T>> void insertionSort(T[] a)`.
7. A certain data collection program collects data from seven remote stations that it contacts over the Internet. Every minute, the program sends a message to the remote stations prompting each of them to collect and return a data sample. Each sample is time stamped by the remote stations. Because of transmission delays, the seven samples do not arrive at the data collection program in time stamp order. The data collection program stores the samples in an array in the order in which it receives them. Every 24 hours, the program sorts the samples by time stamp and stores them in a database. Which sorting algorithm should the program use to sort samples before they are stored: bubble, selection, insertion, or Shell sort? Why?

13.9 REVIEW QUESTION ANSWERS

1. An internal list processes lists stored in main memory, while an external sorts processes lists stored in files.
2. Although the complexity of bubble sort and selection sort is exactly the same, in practice they behave differently. Bubble sort tends to be significantly slower than selection sort, especially when list elements are large entities, because bubble sort moves elements into place in the list by swapping them one location at a time while selection sort merely swaps one element into place on each pass. Bubble sort makes $\Theta(n^2)$ swaps on average, while selection sort $\Theta(n)$ swaps in all cases. Had we chosen swaps as a basic operation, this difference would have been reflected in our analysis.
3. 21Shell sort does different things when the data in the list is different so it has best, worst, and average case behaviors that differ from one another. For example, it will clearly do the least amount of work when the list is already sorted, as is the case for insertion sort.

14 RECURRENCES

14.1 INTRODUCTION

It is relatively easy to set up equations, typically using summations, for counting the basic operations performed in a non-recursive algorithm. But this won't work for recursive algorithms in which much computation is done in recursive calls rather than in loops. How are basic operations to be counted in recursive algorithms?

A different mathematical techniques must be used; specifically, a recurrence relation must be set up to reflect the recursive structure of the algorithm.

> **Recurrence relation**: An equation that expresses the value of a function in terms of its value at another point.

For example, consider the recurrence relation $F(n) = n \cdot F(n\text{-}1)$, where the domain of F is the non-negative integers (all our recurrence relations will have domains that are either the non-negative integers or the positive integers). This recurrence relation says that the value of F is its value at another point times n. Ultimately, our goal will be to solve recurrence relations like this one by removing recursion, but as it stands, it has infinitely many solutions. To pin down the solution to a unique function, we need to indicate the value of the function at some particular point or points. Such specifications are called **initial conditions.** For example, suppose the initial condition for function F is $F(0) = 1$. Then we have the following values of the function.

$$
\begin{aligned}
&F(0) = 1\\
&F(1) = 1 \cdot F(0) = 1\\
&F(2) = 2 \cdot F(1) = 2\\
&F(3) = 3 \cdot F(2) = 6\\
&\ldots
\end{aligned}
$$

We thus recognize F as the factorial function. A recurrence relation plus one or more initial conditions form a recurrence.

> **Recurrence**: a recurrence relation plus one or more initial conditions that together recursively define a function.

14.2 SETTING UP RECURRENCES

Lets consider a few recursive algorithms to illustrate how to use recurrences to analyze them. The Java code in Figure 1 implements a recursive algorithm to reverse a string.

```
String reverse(String s) {
 if (s.length() <= 1) return s;
 return reverse(s.substring(1)) + s.charAt(0);
}
```

Figure 1: Recursively Reversing a String

If the string parameter `s` is a single character or the empty string, then it is its own reversal and it is returned. Otherwise, the first character of `s` is concatenated to the end of the result of reversing `s` with its first character removed.

The size of the input to this algorithm is the length n of the string argument. We will count string concatenation operations. This algorithm always does the same thing no matter the contents of the string, so we need only derive its every-case complexity $C(n)$. If n is 0 or 1, that is, if the string parameter s of `reverse()` is empty or only a single character, then no concatenations are done, so $C(0) = C(1) = 0$. If $n > 1$, then the number of concatenations is one plus however many are done during the recursive call on the substring, which has length n-1, giving us the recurrence relation

$$C(n) = 1 + C(n\text{-}1)$$

Putting these facts together, we have the following recurrence for this algorithm.

$$\begin{aligned} C(n) &= 0 && \text{for } n = 0 \text{ or } n = 1 \\ C(n) &= 1 + C(n\text{-}1) && \text{for } n > 1 \end{aligned}$$

Lets consider a slightly more complex example. The Towers of Hanoi puzzle is a famous game in which one must transfer a pyramidal tower of disks from one peg to another using a third as auxiliary, under the constraint that no disk can be placed on a smaller disk. The algorithm in Figure 2 solves this puzzle in the least number of steps.

```
void moveTower(Peg src, Peg dst, Peg aux, int n) {
 if (n == 1) moveDisk(src, dst);
 else {
  moveTower(src, aux, dst, n-1);
  moveDisk(src, dst);
  moveTower(aux, dst, src, n-1);
 }
}
```

Figure 2: Towers of Hanoi Algorithm

This method appears in a `HanoiState` class that keeps track of the disks on each of three pegs (`Peg` is an enumeration with values `A`, `B`, and `C`) and also includes the `moveDisk()` method. The last parameter of the `moveTower()` method is the number of disks to move from the `src` peg to the `dst` peg. To solve the problem, one creates a `HanoiState s` with `n` disks, which are all placed on peg `A`, (and hence with no disks on pegs `B` or `C`). Then a call to `s.moveTower(A, C, B, n)` transfers all the disks from peg `A` to peg `C` (with `B` as an auxiliary) in the fewest number of steps.

Our measure of the size of the input is n, the number of disks on the source peg. The algorithm always does the same thing for a given value of n, so we compute the every-case complexity $C(n)$. When $n = 1$, only one disk is moved, so $C(1) = 1$. When n is greater than one, then the number of disks moved is the number moved to shift the top n-1 disks to the auxiliary peg, plus one move to put the bottom disk on the destination peg, plus the number moved to shift n-1 disks from the auxiliary peg to the destination peg. This gives the following recurrence.

$$\begin{aligned} C(1) &= 1 && \text{for } n = 1 \\ C(n) &= 1 + 2 \cdot C(n\text{-}1) && \text{for } n > 1 \end{aligned}$$

Although recurrences are nice, they don't tell us in a closed form the complexity of our algorithms—in other words, the only way to calculate the value of a recurrence for n is to start with the initial conditions and work our way up to the value for n using the recurrence relation, which can be a lot of work. We would like to come up with solutions to recurrences that don't use recursion so that we can compute them easily.

14.3 SOLVING RECURRENCES

There are several ways to solve recurrences but we will consider only one called the method of backward substitution. This method has the following steps.

1. Expand the recurrence relation by substituting it into itself several times until a pattern emerges.
2. Characterize the pattern by expressing the recurrence relation in terms of n and an arbitrary term i.
3. Substitute for i an expression that will remove the recursion from the equation.
4. Manipulate the result to achieve a final closed form for the defined function.

To illustrate this technique, we will solve the recurrences above, starting with the one for the string reversal algorithm. Steps one and two for this recurrence appear below.

$$\begin{aligned} C(n) &= 1 + C(n\text{-}1) \\ &= 1 + (1 + C(n\text{-}2)) = 2 + C(n\text{-}2) \\ &= 2 + (1 + C(n\text{-}3)) = 3 + C(n\text{-}3) \\ &= \ldots \\ &= i + C(n\text{-}i) \end{aligned}$$

The last expression characterizes the recurrence relation for an arbitrary term i. $C(n\text{-}i)$ is equal to an initial condition for $i = n\text{-}1$. We can substitute this into the equation as follows.

$$\begin{aligned} C(n) &= i + C(n\text{-}i) \\ &= n\text{-}1 + C(n - (n\text{-}1)) \\ &= n\text{-}1 + C(1) \\ &= n\text{-}1 + 0 \\ &= n\text{-}1 \end{aligned}$$

This solves the recurrence: the number of concatenations done by the `Reverse()` function on a string of length n is n-1 (which makes sense if you think about it).

Now lets do the same thing for the recurrence we generated for the Towers of Hanoi algorithm:

$$\begin{aligned} C(n) &= 1 + 2 \cdot C(n\text{-}1) \\ &= 1 + 2 \cdot (1 + 2 \cdot C(n\text{-}2)) \qquad = 1 + 2 + 4 \cdot C(n\text{-}2) \\ &= 1 + 2 + 4 \cdot (1 + 2 \cdot C(n\text{-}3)) \qquad = 1 + 2 + 4 + 8 \cdot C(n\text{-}3) \\ &= \ldots \\ &= 1 + 2 + 4 + \ldots + 2^i \cdot C(n\text{-}i) \end{aligned}$$

The initial condition for the Towers of Hanoi problem is $C(1) = 1$, and if we set i to n-1, then we can achieve this initial condition and thus remove the recursion:

$$\begin{aligned} C(n) &= 1 + 2 + 4 + \ldots + 2^i \cdot C(n\text{-}i) \\ &= 1 + 2 + 4 + \ldots + 2^{n-1} \cdot C(n\text{-}(n\text{-}1)) \\ &= 1 + 2 + 4 + \ldots + 2^{n-1} \cdot C(1) \\ &= 2^n - 1 \end{aligned}$$

Thus the number of moves made to solve the Towers of Hanoi puzzle for n disks is $2^n - 1$, which is obviously in $\Theta(2^n)$.

14.4 SUMMARY AND CONCLUSION

Recurrences provide the technique we need to analyze recursive algorithms. Together with the summations technique we use with non-recursive algorithms, we are now in a position to analyze any algorithm we write. Often the analysis is mathematically difficult, so we may not always succeed in our analysis efforts. But at least we have techniques that we can use.

14.5 REVIEW QUESTIONS

1. Use different initial conditions to show how the recurrence equation $F(n) = n \cdot F(n\text{-}1)$ has infinitely many solutions.
2. Consider the recurrence relation $F(n) = F(n\text{-}1) + F(n\text{-}2)$ for $n > 1$ with initial conditions $F(0) = F(1) = 1$. What well-known sequence of values is generated by this recurrence?
3. What are the four steps of the method of backward substitution?

14.6 EXERCISES

1. Write the values of the following recurrences for $n = 0$ to 4.
   a) $C(n) = 2 \cdot C(n\text{-}1)$, $C(0) = 1$
   b) $C(n) = 1 + 2 \cdot C(n\text{-}1)$, $C(0) = 0$
   c) $C(n) = b \cdot C(n\text{-}1)$, $C(0) = 1$ (b is some constant)
   d) $C(n) = n + C(n\text{-}1)$, $C(0) = 0$

2. Write the values of the following recurrences for n = 1, 2, 4, and 8.
   a) $C(n) = 2 \cdot C(n/2)$, $C(1) = 1$
   b) $C(n) = 1 + C(n/2)$, $C(1) = 0$
   c) $C(n) = n + 2 \cdot C(n/2)$, $C(1) = 0$
   d) $C(n) = n + C(n/2)$, $C(1) = 1$

3. Use the method of backward substitution to solve the following recurrences.
   a) $C(n) = 2 \cdot C(n\text{-}1)$, $C(0) = 1$
   b) $C(n) = 1 + 2 \cdot C(n\text{-}1)$, $C(0) = 0$
   c) $C(n) = b \cdot C(n\text{-}1)$, $C(0) = 1$ (b is some constant)
   d) $C(n) = n + C(n\text{-}1)$, $C(0) = 0$

4. Use the method of backward substitution to solve the following recurrences. Assume that $n = 2^k$ to solve these equations.
   a) $C(n) = 2 \cdot C(n/2)$, $C(1) = 1$
   b) $C(n) = 1 + C(n/2)$, $C(1) = 0$
   c) $C(n) = n + 2 \cdot C(n/2)$, $C(1) = 0$
   d) $C(n) = n + C(n/2)$, $C(1) = 1$

5. Write a complete Java program to solve the towers of Hanoi problem as outlined in the text. Have the `moveDisk()` method write the state of the game after it moves a disk, so that when towers are moved the sequence of steps can be traced. Number the disks from 1 for the smallest to `n` for the largest. The `HanoiState` class will

need private attributes to keep track of the contents of the towers `A`, `B`, and `C` (what would be a good data structures for this task?), and a `toString()` method to create a `String` representation of the three towers. The main program can create an instance of `HanoiState` with four disks and then call `moveTower(Peg.A, Peg.C, Peg.B, 4)` to move them.

14.7 REVIEW QUESTION ANSWERS

1. To see that $F(n) = n \cdot F(n\text{-}1)$ has infinitely many solutions, consider the sequence of initial conditions $F(0) = 0$, $F(0) = 1$, $F(0) = 2$, and so on. For initial condition $F(0) = 0$, $F(n) = 0$ for all n. For $F(0) = 1$, $F(n)$ is the factorial function. For $F(0) = 2$, $F(\text{n})$ is twice the factorial function, and in general for $F(0) = k$, $F(n) = k \cdot n!$ Hence infinitely many functions are generated by choosing different initial conditions.
2. This recurrence defines the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, ...
3. The four steps of the method of backward substitution are (1) expand the recurrence relation several times until a pattern is detected, (2) express the pattern in terms of n and some index variable i, (3) Find a value for i that uses initial conditions to remove the recursion from the equation, (4) substitute the value for i and simplify to obtain a closed form for the recurrence.

15 MERGE SORT AND QUICKSORT

15.1 INTRODUCTION

The sorting algorithms we have looked at so far are not very fast, except for Shell sort and Insertion sort on nearly-sorted lists. In this chapter we consider two of the fastest sorting algorithms known: merge sort and quicksort.

15.2 MERGE SORT

Merge sort is a divide and conquer algorithm that solves a large problem by dividing it into parts, solving the resulting smaller problems, and then combining these solutions into a solution to the original problem. The strategy of merge sort is to sort halves of a list (recursively) then merge the results into the final sorted list. Merging is a pretty fast operation, and breaking a problem in half repeatedly quickly gets down to lists that are already sorted (lists of length one or zero), so this algorithm performs well. A Java implementation of merge sort appears in Figure 1 below.

```
void mergeSort(int[] a) {
 int[] auxiliary = Arrays.copyOf(a, a.length);
 mergeInto(a, auxiliary, 0, a.length);
}

private static void mergeInto(int[] dst, int[] src,
                  int lo, int length) {
 if (length < 2) return;
 int loLength = length/2;
 int hiLength = length-loLength;
 int hi = lo+loLength;
 mergeInto(src, dst, lo, loLength);
 mergeInto(src, dst, hi, hiLength);
 int j = lo;
 int k = hi;
 for (int i = lo; i < lo+length; i++) {
  if (j < hi && k < hi+hiLength) {
   if (src[j] < src[k]) dst[i] = src[j++];
   else dst[i] = src[k++];
  }
  else if (j < hi) dst[i] = src[j++];
  else dst[i] = src[k++];
 }
}
```

Figure 1: Merge Sort

Merging requires a place to store the result of merging two lists: duplicating the original list provides space for merging. Hence this algorithm duplicates the original list and passes the duplicate to `mergeInto()`. This operation recursively sorts the two halves of the auxiliary list and then merges them back into the original list. Note that `mergeInto()` uses parallel segments of the two arrays and merges them back and forth into each other, which is tricky, but works. Although it is possible to sort and merge in place, or to merge using a list only half the size of the original, the algorithms to do merge sort this way are complicated and have a lot of overhead—it is simpler and faster to use an auxiliary list the size of the original, even though it requires a lot of extra space.

In analyzing this algorithm, the measure of the size of the input is, of course, the length of the list sorted, and the operations counted are key comparisons. Key comparison occurs in the merging step: the smallest items in the merged sub-lists are compared, and the smallest is moved into the target list. This step is repeated until one of the sub-lists is exhausted, in which case the remainder of the other sub-list is copied into the target list.

Merging does not always take the same amount of effort: it depends on the contents of the sub-lists. In the best case, the largest element in one sub-list is always smaller than the smallest element in the other (which occurs, for example, when the input list is already sorted). If we make this assumption, along with the simplifying assumption that $n = 2^k$, then the recurrence relation for the number of comparisons in the best case is

$$\begin{aligned} B(n) &= n/2 + 2 \cdot B(n/2) \\ &= n/2 + 2 \cdot (n/4 + 2 \cdot B(n/4)) = 2 \cdot n/2 + 4 \cdot B(n/4) \\ &= 2 \cdot n/2 + 4 \cdot (n/8 + 2 \cdot B(n/8) = 3 \cdot n/2 + 8 \cdot B(n/8) \\ &= \ldots \\ &= i \cdot n/2 + 2^i \cdot B(n/2^i) \end{aligned}$$

The initial condition for the best case occurs when n is one or zero, in which case no comparisons are made. If we let $n/2^i = 1$, then $i = k = \lg n$. Substituting this into the equation above, we have

$$B(n) = \lg n \cdot n/2 + n \cdot B(1) = (n \lg n)/2$$

Thus, in the best case, merge sort makes only about $(n \lg n)/2$ key comparisons, which is quite fast. It is also obviously in $\Theta(n \lg n)$.

In the worst case, making the most comparisons occurs when merging two sub-lists such that one is exhausted when there is only one element left in the other. In this case, every merge operation for a target list of size n requires n-1 comparisons. We thus have the following recurrence relation:

$$
\begin{aligned}
W(n) &= n\text{-}1 + 2 \cdot W(n/2) \\
&= n\text{-}1 + 2 \cdot (n/2\text{-}1 + 2 \cdot W(n/4)) = 2n - 3 + 4 \cdot W(n/4) \\
&= 2n - 3 + 4 \cdot (n/4\text{-}1 + 2 \cdot W(n/8) = 3n - 7 + 8 \cdot W(n/8) \\
&= \ldots \\
&= i \cdot n - (2^i - 1) + 2^i \cdot W(n/2^i)
\end{aligned}
$$

The initial conditions are the same as before, so we may again let $i = \lg n$ to solve this recurrence.

$$
\begin{aligned}
W(n) &= i \cdot n - (2^i - 1) + 2^i \cdot W(n/2^i) \\
&= n \lg n - (2^{\lg n} - 1) + 2^{\lg n} \cdot W(1) \\
&= n \lg n - (n - 1) \\
&= n \lg n - n + 1
\end{aligned}
$$

The worst case behavior of merge sort is thus also in $\Theta(n \lg n)$.

As an average case, lets suppose that each comparison of keys from the two sub-lists is equally likely to result in an element from one sub-list being moved into the target list as from the other. This is like flipping coins: it is as likely that the element moved from one sub-list will win the comparison as an element from the other. And like flipping coins, we expect that in the long run, about the same number of elements will be chosen from one list as from the other, so that the sub-lists will run out at about the same time. This situation is about the same as the worst case behavior, so on average, merge sort will make about the same number of comparisons as in the worst case.

Thus in all cases, merge sort runs in $\Theta(n \lg n)$ time, which means that it is significantly faster than the other sorts we have seen so far. Its major drawback is that it uses $\Theta(n)$ extra memory locations to do its work.

15.3 QUICKSORT

Quicksort was invented by C. A. R. Hoare in 1960, and it is still the fastest known algorithm for sorting random data by comparison of keys.

Quicksort is a divide and conquer algorithm. It works by selecting a single element in the list, called the *pivot element*, and rearranging the list so that all elements less than or equal to the pivot are to its left, and all elements greater than or equal to it are to its right. This operation is called *partitioning*. Once a list is partitioned, the algorithm calls itself recursively to sort the sub-lists left and right of the pivot. Eventually, the sub-lists have length one or zero, at which point they are sorted, ending the recursion.

The heart of quicksort is the partitioning algorithm. This algorithm must choose a pivot element and then rearrange the list as quickly as possible so that the pivot element is in its final position, all values greater than the pivot are to its right, and all values less than it are to its left. Although there are many variations of this algorithm, the general approach is to choose an arbitrary element as the pivot, scan from the left until a value greater than the pivot is found, and from the right until a value less than the pivot is found. These values are then swapped, and the scans resume. The pivot element belongs in the position where the scans meet. Although it seems very simple, the quicksort partitioning algorithm is quite subtle and hard to get right. For this reason, it is generally a good idea to copy it from a source that has tested it extensively.

A Java implementation appears in Figure 2. A recursive helper function does the real work while the `quicksort()` method provides a simple interface.

```
void quicksort(int[] a) {
 quicksortSublist(a, 0, a.length-1);
}

quicksortSublist(int[] a, int lb, int ub) {
 if (ub <= lb) return;
 int pivot = a[ub];
 int i = lb-1;
 int j = ub;
 while (i < j) {
  do { i++; } while (a[i] < pivot);
  do { j--; } while (lb < j && pivot < a[j]);
  int tmp = a[i]; a[i] = a[j]; a[j] = tmp;
 }
 int tmp = a[i];
 a[ub] = a[j];
 a[j] = tmp;
 a[i] = pivot;
 quicksortSublist(a, lb, i-1);
 quicksortSublist(a, i+1, ub);
}
```

Figure 2: Quicksort

We analyze this algorithm using the list size as the measure of the size of the input, and using comparisons as the basic operation. Quicksort behaves very differently depending on the contents of the list it sorts. In the best case, the pivot always ends up right in the middle of the partitioned sub-lists. We assume, for simplicity, that the original list has 2^k-1 elements. The partitioning algorithm compares the pivot value to every other value, so it makes n-1 comparisons on a list of size n. This means that the recurrence relation for the number of comparison is

$$B(n) = n\text{-}1 + 2 \cdot B((n\text{-}1)/2)$$

The initial condition is $B(n) = 0$ for $n = 0$ or 1 because no comparisons are made on lists of size one or empty lists. We may solve this recurrence as follows:

$$\begin{aligned}
B(n) &= n\text{-}1 + 2 \cdot B((n\text{-}1)/2) \\
&= n\text{-}1 + 2 \cdot ((n\text{-}1)/2 - 1 + 2 \cdot B(((n\text{-}1)/2 - 1)/2)) \\
&= n\text{-}1 + n\text{-}3 + 4 \cdot B((n\text{-}3)/4) \\
&= n\text{-}1 + n\text{-}3 + 4 \cdot ((n\text{-}3)/4 - 1 + 2 \cdot B(((n\text{-}3)/4 - 1)/2)) \\
&= n\text{-}1 + n\text{-}3 + n\text{-}7 + 8 \cdot B((n\text{-}7)/8) \\
&= \ldots \\
&= n\text{-}1 + n\text{-}3 + n\text{-}7 + \ldots + (n\text{-}(2^i\text{-}1) + 2^i \cdot B((n\text{-}(2^i\text{-}1))/2^i)
\end{aligned}$$

If we let $(n\text{-}(2^i\text{-}1))/2^i = 1$ and solve for i, we get $i = k\text{-}1$. Substituting, we have

$$\begin{aligned}
B(n) &= n\text{-}1 + n\text{-}3 + n\text{-}7 + \ldots + n\text{-}(2^{k\text{-}1}\text{-}1) \\
&= \sum_{j = 1 \text{ to } k\text{-}1} n - (2^j - 1) \\
&= \sum_{j = 1 \text{ to } k\text{-}1} n\text{+}1 - \sum_{j = 1 \text{ to } k\text{-}1} 2^j \\
&= (k\text{-}1) \cdot (n\text{+}1) + 1 - \sum_{j = 0 \text{ to } k\text{-}1} 2^j \\
&= k \cdot (n\text{+}1) - n - (2^k - 1) \\
&= (n\text{+}1) \lg (n\text{+}1) - 2n
\end{aligned}$$

Thus the best case complexity of quicksort is in $\Theta(n \lg n)$.

Quicksort's worst case behavior occurs when the pivot element always ends up at one end of the sub-list, meaning that sub-lists are not divided in half when they are partitioned, but instead one sub-list is empty and the other has one less element than the sub-list before partitioning. If the first or last value in the list is used as the pivot, this occurs when the original list is already in order or in reverse order. In this case the recurrence relation is

$$W(n) = n\text{-}1 + W(n\text{-}1)$$

This recurrence relation is easily solved and turns out to be $W(n) = n(n\text{-}1)/2$, which of course we know to be in $\Theta(n^2)$!

The average case complexity of quicksort involves a recurrence that is somewhat hard to solve, so we simply present the solution: $A(n) = 2(n\text{+}1) \cdot \ln 2 \cdot \lg n \approx 1.39\ (n\text{+}1) \lg n$. This is not far from quicksort's best case complexity. So in the best and average cases, quicksort is very fast, performing $\Theta(n \lg n)$ comparisons; but in the worst case, quicksort is very slow, performing $\Theta(n^2)$ comparisons.

15.4 IMPROVEMENTS TO QUICKSORT

Quicksort's worst case behavior is abysmal, and because it occurs for sorted or nearly sorted lists, which are often encountered, this is a big problem. Many solutions to this problem have been proposed; one of the simplest is called the *median-of-three improvement*, and it consists of using the median of the first, last, and middle values in each sub-list as the pivot element. Except in rare cases, this technique produces a pivot value that ends up near the middle of the sub-list when it is partitioned, especially if the sub-list is sorted or nearly sorted.

A version of quicksort with the median-of-three improvement appears in Figure 3. In this code, once the first, middle, and last elements of a sub-list are put in order (and assuming there are more than three elements in the sub-list), the median is swapped into the next-to-last location and used as the pivot; otherwise, the algorithm is unchanged.

The median-finding process also allows sentinel values to be placed at the ends of the sub-list, which speeds up the partitioning algorithm a little bit because array indices do not need to be checked.

Sentinel value: a special value placed in a data structure to mark a boundary.

From now on, we will assume that quicksort includes the median-of-three improvement.

```
void qsort(int[] a) {
 qsortSublist(a, 0, a.length-1);
}

void qsortSublist(int[] a, int lb, int ub) {
 if (ub <= lb) return;
 int m = (ub+lb)/2;
 if (a[m]  < a[lb])
  { int tmp = a[m]; a[m] = a[lb]; a[lb] = tmp; }
 if (a[ub] < a[m])
  { int tmp = a[ub]; a[ub] = a[m]; a[m] = tmp; }
 if (a[m]  < a[lb])
  { int tmp = a[m]; a[m] = a[lb]; a[lb] = tmp; }
 if (ub-lb < 3) return;
 int pivot = a[m];
 int tmp = a[ub-1]; a[ub-1] = a[m]; a[m] = tmp;
 int i = lb;
 int j = ub-1;
 while (i < j) {
  do { i++; } while (a[i] < pivot);
  do { j--; } while (lb < j && pivot < a[j]);
  tmp = a[i]; a[i] = a[j]; a[j] = tmp;
 }
 tmp = a[i];
  a[ub-1] = a[j];
 a[j] = tmp;
 a[i] = pivot;
 qsortSublist(a, lb, i-1);
 qsortSublist(a, i+1, ub);
}
```

Figure 3: Quicksort with the Median-of-Three Improvement

Other improvement to quicksort have been proposed, and each speeds it up slightly at the expense of making it a bit more complicated. Among the suggested improvement are the following:

- Use Insertion sort for small sub-lists (ten to fifteen elements). This eliminates a lot of recursive calls on small sub-lists and takes advantage of Insertion sort's linear behavior on nearly sorted lists. Generally this is implemented by having quicksort stop when it gets down to lists of less than ten or fifteen elements and then Insertion sorting the whole list at the end.
- Remove recursion and use a stack to keep track of sub-lists yet to be sorted. This removes function calling overhead.
- Partition smaller sub-lists first, which keeps the stack a little smaller.

Despite all these improvement, there are still many cases where quicksort does poorly on data that has some degree of order, which is characteristic of much real-world data. Recent efforts to find sorting algorithms that take advantage of order in the data have produced algorithms (usually based on merge sort) that use little extra space and are far faster than quicksort on data with some order. Such algorithms are considerably faster than quicksort in many real-world cases.

15.5 SUMMARY AND CONCLUSION

Merge sort is a fast sorting algorithm whose best, worst, and average case complexity are all in $\Theta(n \lg n)$, but unfortunately it uses $\Theta(n)$ extra space to do its work. Quicksort has best and average case complexity in $\Theta(n \lg n)$, but unfortunately its worst case complexity is in $\Theta(n^2)$. The median-of-three improvement makes quicksort's worst case behavior less likely, but it is still vulnerable to poor performance on data with some order. Nevertheless, quicksort is still the fastest algorithm known for sorting random data by comparison of keys.

15.6 REVIEW QUESTIONS

1. Why does merge sort need extra space?
2. What stops recursion in merge sort and quicksort?
3. What is a pivot value in quicksort?
4. What changes have been suggested to improve quicksort?
5. If quicksort has such bad worst case behavior, why is it still used?

15.7 EXERCISES

1. Explain why the merge sort algorithm first copies the original array into the auxiliary array.
2. Write a non-recursive version of merge sort that uses a stack to keep track of sub-lists that have yet to be sorted. Time your implementation against the unmodified merge sort algorithm and summarize the results.
3. Modify the quicksort algorithm with the median-of-three improvement so that it does not sort lists smaller than a dozen elements and calls Insertion sort to finish sorting at the end. Time your implementation against the unmodified quicksort with the median-of-three improvement and summarize the results.
4. Modify the quicksort algorithm with the median-of-three improvement so that it uses a stack rather than recursion and works on smaller sub-lists first. Time your implementation against the unmodified quicksort with the median-of-three improvement and summarize the results.
5. Write the fastest quicksort you can. Time your implementation against the unmodified quicksort and summarize the results.
6. The algorithms presented above in Java sort `int` arrays. Modify these algorithms so that they can sort slices of arbitrary types.

15.8 REVIEW QUESTION ANSWERS

1. Merge sort uses extra space because it is awkward and slow to merge lists without using extra space.
2. Recursion in merge sort and quicksort stops when the sub-lists being sorted are either empty or of size one—such lists are already sorted, so no work needs to be done on them.
3. A pivot value in quicksort is an element of the list being sorted that is chosen as the basis for rearranging (partitioning) the list: all elements less than the pivot are placed to the left of it, and all elements greater than the pivot are placed to the right of it. (equal values may be placed either left or right of the pivot, and different partitioning algorithms may make different choices).

4. Among the changes that have been suggested to improve quicksort are (a) using the median of the first, last, and middle elements in the list as the pivot value, (b) using Insertion sort for small sub-lists, (c) removing recursion in favor of a stack, and (d) sorting small sub-lists first to reduce the depth of recursion (or the size of the stack).
5. Quicksort is still used because its performance is so good on average: quicksort usually runs in about half the time of other sorting algorithms, especially when it has been improved in the ways discussed in the chapter. Its worst case behavior is relatively rare if it incorporates the median-of-three improvement.

16 TREES, HEAPS, AND HEAPSORT

16.1 INTRODUCTION

Trees are the basis for several important data types and data structures. There are several sorting algorithms based on trees. One of these algorithms is heapsort, which uses a complete binary tree represented in an array for fast in-place sorting.

16.2 BASIC TERMINOLOGY

A tree is a special type of graph.

Graph: A collection of *vertices* (or *nodes*) and *edges* connecting the vertices. An edge may be thought of as a pair of vertices. Formally, a graph is an ordered pair <*V*,*E*> where *V* is a set of vertices and *E* is a set of pairs of elements of *V*.

Simple path: A list of distinct vertices such that successive vertices are connected by edges.

Tree: A graph with a distinguished vertex *r*, called the *root*, such that there is exactly one simple path between each vertex in the tree and *r*.

We usually draw trees with the root at the top and the vertices and edges descending below it. Figure 1 illustrates a tree.

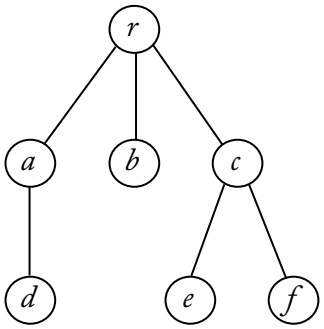


Figure 1: A Tree

Vertex *r* is the root. The root has three *children*: *a*, *b*, and *c*. The root is the *parent* of these vertices. These vertices are also *siblings* of one another because they have the same parent Vertex *a* has child *d* and vertex *c* has children *e* and *f*. The ancestors of a vertex are the vertices on the path between it and the root; the *descendants* of a vertex are all the vertices of which it is an ancestor. Thus vertex *f* has ancestors *f*, *c*, and *r*, and *c* has descendants

c, *e*, and *f*. A vertex without children is a *terminal vertex* or a *leaf*; those with children are *non-terminal* vertices or *internal* vertices. The tree in Figure 1 has three internal vertices (*r*, *a*, and *c*) and four leaf vertices (*b*, *d*, *e*, and *f*). The graph consisting of a vertex in a tree, all its descendants, and the edges connecting them, is a *sub-tree* of the tree.

A graph consisting of several trees is a *forest*. The *level* of a vertex in a tree is the number of edges in the path from the vertex to the root. In Figure 1, vertex *r* is at level zero, vertices *a*, *b*, and *c* are at level one, and vertices *d*, *e*, and *f* are at level two. The *height* of a tree is the maximum level in the tree. The height of the tree in Figure 1 is two.

An *ordered tree* is one in which the order of the children of each vertex is specified. Ordered trees are not drawn in any special way—some other mechanism must be used to specify whether a tree is ordered.

16.3 BINARY TREES

Binary trees are especially important for making data structures.

> **Binary tree:** An ordered tree whose vertices have at most two children. The children are distinguished as the *left child* and *right child*. The sub-tree whose root is the left (right) child of a vertex is the *left (right) sub-tree* of that vertex.

A *complete binary tree* is one whose every level is full except possibly the last, and only the right-most leaves at the bottom level are missing. Figure 2 illustrates these notions.

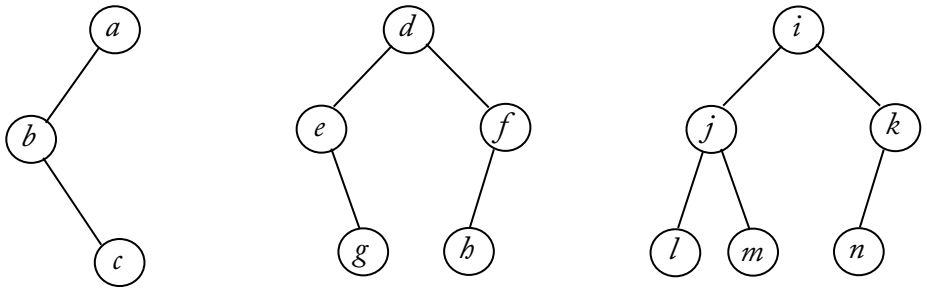


Figure 2: Binary Trees

The trees in Figure 2 are binary trees. In the tree on the left, vertex *a* has a left child, vertex *b* has a right child, and vertex *c* has no children. This tree is not complete. The middle tree is not complete because although every level but the last is full, the missing leaves at the bottom level are not right-most. The right tree is complete.

Trees have several interesting and important properties, the following among them.

- A tree with n vertices has n-1 edges.
- A complete binary tree with n internal vertices has either n or n+1 leaves.
- The height of a complete binary tree with n vertices is $\lfloor \lg n \rfloor$.

16.4 HEAPS

A vertex in a binary tree has the *heap-order property* if the value stored at the vertex is greater than or equal to the values stored at its descendants.

Heap: A complete binary tree whose every vertex has the heap-order property.

An arbitrary complete binary tree can be made into a heap as follows:

- Every leaf already has the heap-order property, so the sub-trees whose roots are leaves are heaps.
- Starting with the right-most internal vertex v at the next-to-last level, and working left across levels and upwards in the tree, do the following: if vertex v does not have

the heap-order property, swap its value with the largest of its children, then do the same with the modified vertex, until the sub-tree rooted at v is a heap.

It is fairly efficient to make complete binary trees into heaps because each sub-tree is made into a heap by swapping its root downwards in the tree as far as necessary. The height of a complete binary tree is $\lfloor \lg n \rfloor$, so this operation cannot take very long.

Heaps can be implemented in a variety of ways, but the fact that they are complete binary trees makes it possible to store them very efficiently in contiguous memory locations.

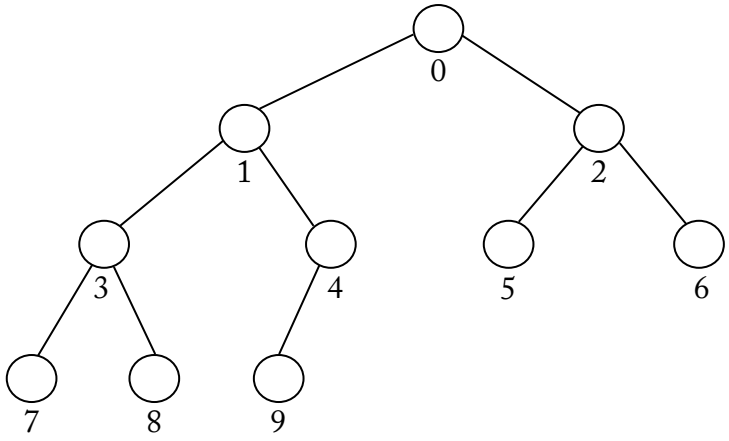


Figure 3: Numbering Vertices for Contiguous Storage

Consider the numbers assigned to the vertices of the complete binary tree in Figure 3. Note that numbers are assigned left to right across levels, and from top to bottom of the tree.

This numbering scheme can be used to identify each vertex in a complete binary tree: vertex zero is the root, vertex one is the left child of the root, vertex two is the right child of the root, and so forth. Note in particular that

- The left child of vertex k is vertex $2k+1$.
- The right child of vertex k is vertex $2k+2$.
- The parent vertex k is vertex $\lfloor (k-1)/2 \rfloor$.
- If there are n vertices in the tree, the last one with a child is vertex $\lfloor n/2 \rfloor - 1$.

If we let these vertex numbers be array indices, then each array location is associated with a vertex in the tree, and we can store the values at the vertices of the tree in the array: the value of vertex k is stored in array location k. The correspondence between array indices and vertex locations thus makes it possible to represent complete binary trees in arrays. The fact that the binary tree is complete means that every array location stores the value at a vertex, so no space is unused in the array.

16.5 HEAPSORT

We now have all the pieces we need to for an efficient and interesting sorting algorithm based on heaps. Suppose we have an array to be sorted. We can consider it to be a complete binary tree stored in an array as explained above. Then we can

- Make the tree into a heap as explained above.
- The largest value in a heap is at the root, which is always at array location zero. We can swap this value with the value at the end of the array and pretend the array is one element shorter. Then we have a complete binary tree that is almost a heap—we just need to sift the root value down the tree as far as necessary to make it one. Once we do, the tree will once again be a heap.
- We can then repeat this process again and again until the entire array is sorted.

This sorting algorithm, called *heapsort*, is shown in the Java code in Figure 4 below.

```
void heapsort(int[] a) {
 if (a.length < 2) return;
 int maxIndex = a.length-1;
 for (int i = (maxIndex-1)/2; 0 <= i; i--)
  siftDown(a, i, maxIndex);
 while (true) {
  int tmp = a[0];
  a[0] = a[maxIndex];
  a[maxIndex] = tmp;
  maxIndex--;
  if (maxIndex <= 0) break;
  siftDown(a, 0, maxIndex);
 }
}

private static void siftDown(int[] a, int i,
                 int maxIndex) {
 int tmp = a[i];
 for (int j = 2*i+1; j <= maxIndex; j = 2*i+1) {
  if (j < maxIndex && a[j] < a[j+1]) j++;
  if (a[j] <= tmp) break;
  a[i] = a[j];
  i = j;
 }
 a[i] = tmp;
}
```

Figure 4: Heapsort

A complex analysis that we will not reproduce here shows that the number of comparisons done by heapsort in both the best, worst, and average cases are all in $\Theta(n \lg n)$. Thus heapsort joins Shell sort, merge sort, and quicksort in our repertoire of fast sorting algorithms. Empirical studies have shown that while heapsort is not as fast as quicksort, it is not much slower than Shell sort and merge sort, with the advantage that it does not use any extra space like merge sort, and it does not have bad worst case complexity like quicksort.

16.6 SUMMARY AND CONCLUSION

A tree is a special sort of graph that is important in computing. One application of trees is for sorting: an array can be treated as a complete binary tree and then transformed into a heap. The heap can then be manipulated to sort the array in place in $\Theta(n \lg n)$ time. This algorithm is called heapsort and it is a good algorithm to use when space is at a premium and respectable worst case complexity is required.

16.7 REVIEW QUESTIONS

1. In Figure 1, what are the descendants of *r*? What are the ancestors of *r*?
2. How can you tell from a diagram whether a tree is ordered?
3. Is every binary tree all of whose levels are full a complete binary tree? Does every complete binary tree have every level full?
4. Where is the largest value in a heap?
5. Using the heap data structure numbering scheme, which vertices are the left and right children of vertex 27? Which vertex is the parent of vertex 27?
6. What is the worst case behavior of heapsort?

16.8 EXERCISES

1. Represent the three trees in Figure 2 as sets of ordered pairs according to the definition of a graph.
2. Could the graph in Figure 1 still be a tree if *b* was its root? If so, redraw the tree in the usual way (that is, with the root at the top) to make clear the relationships between the vertices.
3. Draw a complete binary tree with 12 vertices, placing arbitrary values at the vertices. Use the algorithm discussed in the chapter to transform the tree into a heap, redrawing the tree at each step.
4. Suppose that we change the definition of the heap-order property to say that the value stored at a vertex is less than or equal to the values stored at its descendants. If we use the heapsort algorithm on trees that are heaps according to this definition, what will be the result?
5. In the heapsort algorithm in Figure 4, the `siftDown()` operation is applied to vertices starting at `maxIndex-1`. Why does the algorithm not start at `maxIndex`?
6. Draw a complete binary tree with 12 vertex, placing arbitrary values at the vertices. Use the heapsort algorithm to sort the tree, redrawing the tree at each step, and placing removed values into a list representing the sorted array as they are removed from the tree.
7. Write a program to sort arrays of various sizes using heapsort, merge sort, and quicksort. Time your implementations and summarize the results.
8. Introspective sort is a quicksort-based algorithm recently devised by David Musser. Introspective sort works like quicksort except that it keeps track of the depth of recursion (or of the stack), and when recursion gets too deep (about $2 \cdot \lg n$ recursive calls or stack elements), it switches to heapsort to sort sub-lists. This algorithm does $\Theta(n \lg n)$ comparisons even in the worst case, sorts in place, and usually runs almost as fast as quicksort on average. Write an introspective sort method, time your implementation against standard quicksort, and summarize the results.

16.9 REVIEW QUESTION ANSWERS

1. In Figure 1 the descendants of r are all the vertices in the tree. Vertex r has no ancestor except itself.
2. You can't tell from a diagram whether a tree is ordered; there must be some other notation to indicate that this is the case.
3. Every tree whose every level is full is a complete binary tree because it has no missing leaves on its bottom level, and hence it is not the case that there is a missing leaf on the bottom level that is not right-most. Not every complete binary tree has all its levels full, however.
4. The largest value in a heap is always at the root.
5. Using the heap data structure numbering scheme, the left child of vertex 27 is vertex (2*27)+1= 55, the right child of vertex 27 is vertex (2*27)+2 = 56, and the parent of vertex 27 is vertex $\lfloor(27\text{-}1)/2\rfloor$ = 13.
6. The worst, best, and average case behavior of heapsort is in $\Theta(n \lg n)$.

17 BINARY TREES

17.1 INTRODUCTION

As mentioned in the last chapter, binary trees are ordered trees whose vertices have at most two children, a left child and a right child. Although other kinds of ordered trees arise in computing, binary trees are especially common and have been very well studied. In this chapter we discuss the binary tree abstract data type and binary trees as an implementation mechanism.

17.2 THE BINARY TREE ADT

Binary trees hold values of some type, so the ADT is *binary tree of T*, where *T* is the type of the elements in the tree. The carrier set of this type is the set of all binary trees whose vertices hold a value of type *T*. The carrier set thus includes the empty tree, trees with only a root with a value of type *T*, trees with a root and a left child, trees with a root and a right child, and so forth. Operations in the implicit-receiver method set include the following.

size()—Return the number of vertices in the tree.

height()—Return the height of the tree.

isEmpty()—Return true just in case the tree is the empty tree.

contains(*v*)—Return true just in case the value *v* is present in the tree.

rootValue()—Return the value of type *T* stored at the root of the tree. Its precondition is that the tree is not the empty tree.

leftSubtree()—Return the tree whose root is the left child of the root of the tree. Its precondition is that the tree is not the empty tree.

rightSubtree()—Return the tree whose root is the right child of the root of the tree. Its precondition is that the tree is not the empty tree.

Suppose we have the following tree constructors.

newEmptyTree()—Create and return a new empty tree.

newTree(v, t_1, t_2)—Create and return a new tree with v at its root, left child t_1 and right child t_2.

For example, consider the binary tree in Figure 1.

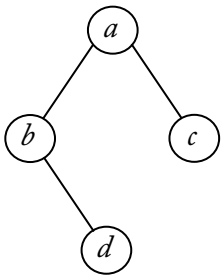


Figure 1: A Binary Tree

This tree can be constructed using the expression below.

```
newTree(a,
        newTree(b,
                newEmptyTree(),
                newTree(d,
                        newEmptyTree(),
                        newEmptyTree())),
        newTree(c,
                newEmptyTree(),
                newEmptyTree()))
```

To extract a value from the tree, such as the bottom-most vertex *d*, we could use the following expression, where *t* is the tree in Figure 1.

```
t.leftSubtree().rightSubtree().rootValue()
```

17.3 THE BINARY TREE CLASS

We could treat binary trees as a kind of collection, adding it to our container hierarchy, but we won't do this for two reasons:

- In practice, binary trees are used to implement other containers, not as containers in their own right. Usually clients are interested in using basic `Container` operations, not in the intricacies of building and traversing trees. Adding binary trees to the container hierarchy would complicate the hierarchy with a container that not many clients would use.

- Although binary trees have a contiguous implementation (discussed below), it is not useful except for heaps. Providing such an implementation in line with our practice of always providing both contiguous and linked implementations of all containers would create an entity without much use.

We will make a `BinaryTree` class but its role will be to provide an implementation mechanism for other collections. Thus the `BinaryTree` class is not part of the container hierarchy, though it includes several standard `Container` operations. It also includes operations for accessing and traversing trees in various ways, as well as several kinds of iterators. The `BinaryTree` class is pictured in Figure 2.

Note that there is no `newTree()` method and no `newEmptyTree()` method in the `BinaryTree` class, though there is one in the ADT. The `BinaryTree` class constructor does the job of these two operations, so they are not needed as separate operations in the class.

To *visit* or *enumerate* the vertices of a binary tree is to traverse or iterate over them one at a time, processing the values held in each vertex. This requires that the vertices be traversed in some order. There are three fundamental orders for traversing a binary tree. All are most naturally described in recursive terms.

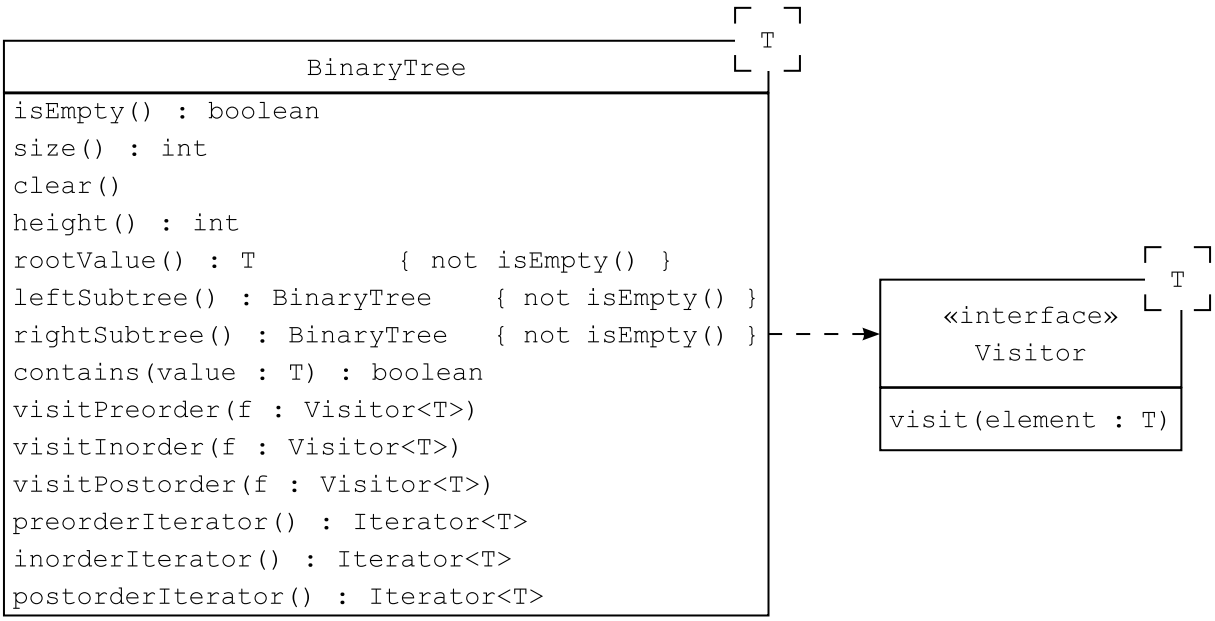


Figure 2: The BinaryTree Class

Preorder: When the vertices of a binary tree are visited in preorder, the root vertex of the tree is visited first, then the left sub-tree (if any) is visited in preorder, then the right sub-tree (if any) is visited in preorder.

Inorder: When the vertices of a binary tree are visited inorder, the left sub-tree (if any) is visited inorder, then the root vertex is visited, then the right sub-tree is visited inorder.

Postorder: When the vertices of a binary tree are visited in postorder, the left sub-tree is visited in postorder, then the right sub-tree is visited in postorder, and then the root vertex is visited.

To illustrate these traversals, consider the binary tree in Figure 3 below. An inorder traversal of the tree in Figure 3 visits the vertices in the order *m*, *b*, *p*, *k*, *t*, *d*, *a*, *g*, *c*, *f*, *h*. A preorder traversal visits the vertices in the order *d*, *b*, *m*, *k*, *p*, *t*, *c*, *a*, *g*, *f*, *h*. A postorder traversal visits the vertices in the order *m*, *p*, *t*, *k*, *b*, *g*, *a*, *h*, *f*, *c*, *d*.

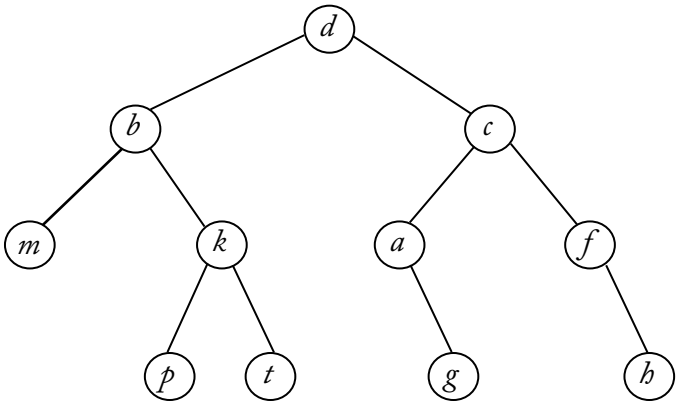


Figure 3: A Binary Tree

Recall from our discussion of iteration in Chapter 8 that this can be done with internal iteration (control is in the collection) or external iteration (control is in an iterator). The `BinaryTree` class has methods for creating external iterators and for internal iteration. Internal iteration is done by packaging the processing that a client wants to do in some way, and passing the packaged process to the collection for application to its elements. In this case, clients write a class implementing the `Visitor` interface, with the processing done by a `visit()` method. An instance of this class is then passed to `visitPreorder()`, `visitInorder()`, or `visitPostorder()` (depending on the order in which vertices are to be visited,) which apply the `visit()` method to every value in the binary tree.

17.4 CONTIGUOUS IMPLEMENTATION OF BINARY TREES

We have already considered how to implement binary trees using an array when we learned about heapsort. The contiguous implementation is excellent for complete or even full binary trees because it wastes no space on references and it provides a quick and easy way to navigate in the tree. Unfortunately, in most applications binary trees are far from complete, so many array locations are never used, which wastes a lot of space. Even if our binary trees were always complete, there is still the problem of having to predict the size of the tree ahead of time to allocate an array big enough to hold all the tree vertices. The array could be reallocated if the tree becomes too large, but this is an expensive operation.

This is why it is not particularly useful to have a contiguous implementation of binary trees. The `BinaryTree` class will use a linked data structure to represent binary trees, and this class will be the basis for other collection types that use binary trees to hold collection data.

17.5 LINKED IMPLEMENTATION OF BINARY TREES

A linked implementation of binary trees resembles implementations of other ADTs using linked structures. Binary tree nodes require three fields: one for the data held at the node and two for references to the nodes that are the roots of the left and right sub-trees. In addition, it is useful to have an object acting as a host for the graph formed by the linked nodes. This host object has a reference to the tree's root node and other fields as needed. For example, a `count` field might be useful to keep track of the number of nodes in the tree.

Figure 4 shows how this works for a small example. There is a single instance of the host structure with a `count` field and a `root` field. The reference in the `root` field points to the root node of the tree. The fields in the nodes are called `value` (for the data at the node), `left`, and `right` (for the references to the sub-trees). The reference fields in these structures are null when the trees to which they refer are empty.

Trees are inherently recursive structures so it is natural to write many `BinaryTree` methods recursively. For example, the height of a binary tree that has one or less vertices is zero, and otherwise it is one plus the maximum of the heights of its two sub-trees. Thus in a Java implementation we may write a `height()` method that calls a private `heightOfTree(BinaryTreeNode r)` method with the root node of the tree as its argument. This method returns zero if its argument node is null or has empty subtrees, and otherwise returns one plus the maximum of the results of recursive calls on the left and right sub-trees of its argument node. Many other `BinaryTree` methods, and particularly the internal iterator methods, can be implemented easily using recursion.

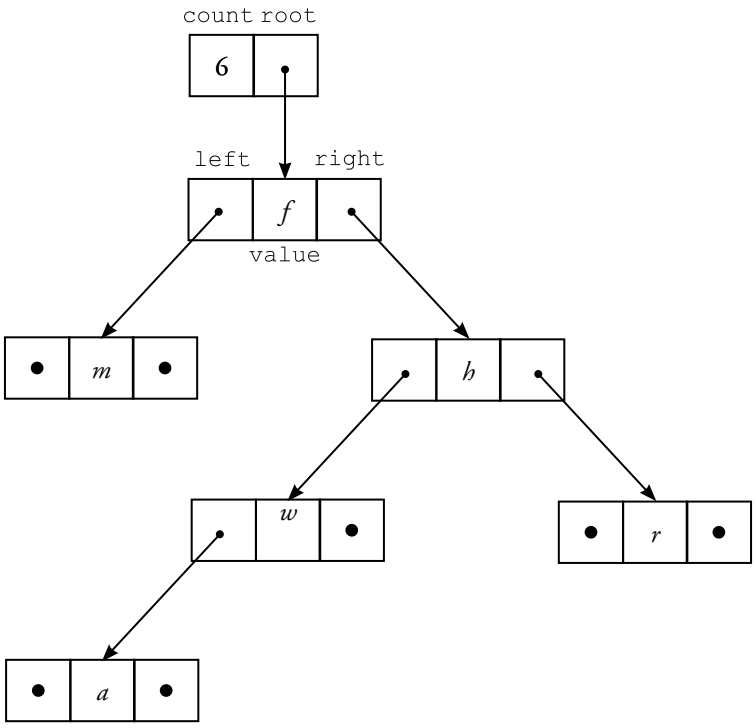


Figure 4: Linked Representation of a Binary Tree

Implementing external iterators is more challenging, however. The problem is that external iterators cannot be written recursively because they have to be able to stop every time a new node is visited to deliver the value at the node to the client. There are two ways to solve this problem:

- Write a recursive operation to copy node values into a queue in the correct order and then extract items from the data structure one at a time as the client requests them.
- Don't use recursion to implement iterators: use a stack instead.

The second alternative, though a little harder, is better because it uses much less space.

17.6 SUMMARY AND CONCLUSION

The binary tree ADT describes basic operations for building and examining binary trees whose vertices hold values of type *T*. A `BinaryTree` class has several methods not in the ADT, in particular, internal iterator methods that apply a method (in a `Visitor` object) to every value in the tree in some order, and external iterator factory methods.

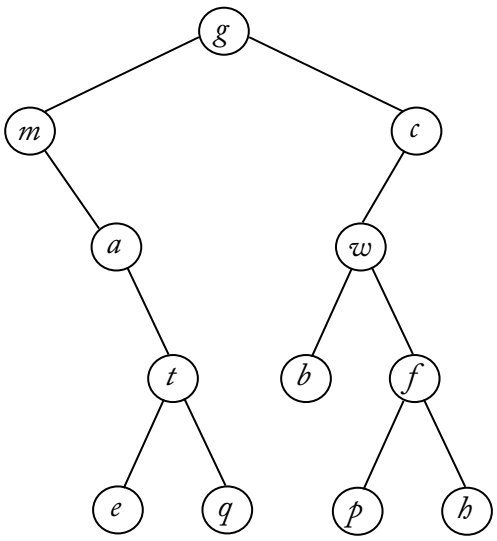
Contiguous implementations of the binary tree ADT are possible and useful in some special circumstances, such as in heapsort, but the main technique for implementing the binary tree ADT uses a linked representation. Recursion is a very useful tool for implementing most `BinaryTree` operations but it cannot be used as easily for implementing external iterators. A `BinaryTree` realization with a linked structure will be used as an implementation mechanism for container classes to come.

17.7 REVIEW QUESTIONS

1. Where does the `BinaryTree` class fit in the container hierarchy?
2. Why does the `BinaryTree` class not include a `newTree()` method?
3. Why is the contiguous implementation of binary trees not very useful?
4. Why is recursion important in writing `BinaryTree` methods?

17.8 EXERCISES

1. Write the values of the vertices in the following tree in the order they are visited when the tree is traversed inorder, in preorder, and in postorder.



2. Write the `Visitor` interface in Java.
3. Start writing the `BinaryTree` class in Java by creating its fields and constructors and a `BinaryTreeNode` class.
4. Continue writing the `BinaryTree` class by adding `isEmpty()`, `size()`, `clear()`, `rootValue()`, `leftSubtree()`, and `rightSubtree()` methods.

5. Continue writing the `BinaryTree` class by adding `height()` and `contains()` methods. You may need private recursive helper functions.
6. Add `visitPreorder()`, `visitInorder()`, and `visitPostorder()` to the `BinaryTree` class. You will also need to create a `Visitor` interface.
7. Add `preorderIterator()`, `inorderIterator()`, and `postorderIterator()` to the `BinaryTree` class. You will need to create iterator classes that use a stack to keep track of their state during iteration.

17.9 REVIEW QUESTION ANSWERS

1. We have decided not to include the `BinaryTree` class in the container hierarchy because it is usually not used as a container in its own right, but rather as an implementation mechanism for other containers.
2. The `BinaryTree` class does not include a `newTree()` method because this operation is a factory function that creates `BinaryTree` instance. But this job can be done using a `BinaryTree` constructor.
3. The contiguous implementation of binary trees is not very useful because it only uses space efficiently if the binary tree is at least full, and ideally complete. In practice, this is rarely the case so the linked implementation uses space more efficiently.
4. Recursion is important in writing `BinaryTree` methods because binary trees are defined recursively and many of the properties and characteristics of binary trees are too. Hence, methods to process binary trees or determine their properties and characteristics are easily written recursively by modelling them on these recursive definitions

18 BINARY SEARCH AND BINARY SEARCH TREES

18.1 INTRODUCTION

Binary search is a much faster alternative to sequential search for sorted lists. Binary search is closely related to binary search trees, which are a special kind of binary tree. We will look at these two topics in this chapter, studying the complexity of binary search, and eventually arriving at a specification for a `BinarySearchTree` class.

18.2 BINARY SEARCH

When people search for something in an ordered list (like a dictionary or a phone book), they do not start at the first element and march through the list one element at a time. They jump into the middle of the list, see where they are relative to what they are looking for, and then jump either forward or backward and look again, continuing in this way until they find what they are looking for, or determine that it is not in the list.

Binary search takes the same tack in searching for a key in a sorted list: the key is compared with the middle element in the list. If it is the key, the search is done. If the key is less than the middle element, then the process is repeated for the first half of the list. If the key is greater than the middle element, then the process is repeated for the second half of the list. Eventually, either the key is found in the list, or the list is reduced to nothing (the empty list), at which point we know that the key is not present in the list.

This approach naturally lends itself to a recursive algorithm, which we show in Java below.

```
int binarySearchRecursive(int[] a, int key) {
 return binSearch(key, a, 0, a.length-1);
}

int binSearch(int key, int[] a, int lo, int hi) {
 if (hi < lo) return -1;
 int m = (lo+hi) / 2;
 if (key == a[m]) return m;
 if (key < a[m]) return binSearch(key, a, lo, m-1);
 return binSearch(key, a, m+1, hi);
}
```

Figure 1: Recursive Binary Search

Search algorithms traditionally return the index of the key in the list or -1 if the key is not found, so that is what happens here. Note also that although the algorithm has the important precondition that the array is sorted, checking this would take far too much time, so it is not checked. A helper function is used because we need to keep track of the bounds of the unsearched portion of the array, but we don't want clients to worry about (or even know about) this detail.

Recursion stops when the unsearched portion of the array is empty. Otherwise, the element at index `m` in the middle of the unsearched portion of the array is checked. If it is the key, the search is done and index `m` is returned; otherwise, a recursive call is made to search the portion of the array before or after `m` depending on whether the key is less than or greater than `a[m]`.

Although binary search is naturally recursive, it is also tail recursive. Recall that a tail recursive algorithm is one in which at most one recursive call is the last action in each activation of the algorithm, and that tail recursive algorithms can always be converted to non-recursive algorithms using only a loop and no stack. This is always more efficient and often simpler as well. In the case of binary search, the non-recursive algorithm is about equally complicated, as the Java code in Figure 2 below shows.

```
int binarySearch(int[] a, int key) {
 int lo = 0;
 int hi = a.length-1;
 while (lo <= hi) {
  int m = (lo+hi) / 2;
  if (key == a[m]) return m;
  if (key < a[m]) hi = m-1;
  else lo = m+1;
 }
 return -1;
}
```

Figure 2: Non-Recursive Binary Search

To analyze binary search, we will consider its behavior on lists of size n and count the number of comparisons between list elements and the search key. Traditionally, the determination of whether the key is equal to, less than, or greater than a list element is counted as a single comparison even though it requires two comparisons in most programming languages.

Binary search does not do the same thing on every input of size n. In the best case, it finds the key in the middle of the array, doing only a single comparison. In the worst case, the key is not in the array or is found when the portion of the array being searched has only one element. We can easily generate a recurrence relation and initial conditions to find the worst case complexity of binary search, but we will instead use a binary search tree to figure this out.

Suppose that we construct a binary tree from a sorted list as follows: the root of the tree is the element in the middle of the list; the left child of the root is the element in the middle of the first half of the list; the right child of the root is the element in the middle of the second half of the list, and so on. In other words, the vertices of the binary tree are set according to the order in which the values would be encountered during a binary search of the list. To illustrate, consider the binary tree in Figure 3 made from the list $<a, b, c, d, e, f, g, h, i, j, k, l, m>$ in the way just described.

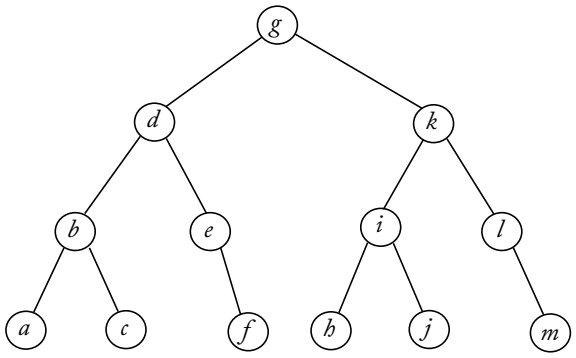


Figure 3: A Binary Tree Made from a List

A tree built this way has the following interesting properties:

- All levels but possibly the last are full, so its height is always $\lfloor \lg n \rfloor$.
- For every vertex, every element in its left sub-tree (if any) is less than or equal to the element at the vertex, and every element in its right sub-tree (if any) is greater than or equal to the element at the vertex.
- If we traverse the tree inorder, we visit the vertices in the order of the original list, that is, in sorted order.

The first property tells us the worst case performance of binary search because a binary search will visit each vertex from the root to a leaf in the worst case. The number of vertices on these paths is the height of the tree plus one, so $W(n) = \lfloor \lg n \rfloor + 1$. We can also calculate the average case by considering each vertex equally likely to be the target of a binary search, and figuring out the average length of the path to each vertex. This turns out to be approximately lg n for both successful and unsuccessful searches. Hence, on average and in the worst case, binary search makes $\Theta(\lg n)$ comparisons, which is very good.

18.3 BINARY SEARCH TREES

The essential characteristic of the binary tree we looked at above is the relationship between the value at a vertex and the values in its left and right sub-trees. This is the basis for the definition of binary search trees.

Binary search tree: A binary tree whose every vertex is such that the value at each vertex is greater than the values in its left sub-tree, and less than the values in its right sub-tree.

Binary search trees are an important kind of graph that retains the property that traversing them in order visits the values in the vertices in sorted order. However, a binary search tree may not be balanced, so its height may be greater than $\lfloor \lg n \rfloor$. In fact, a binary search tree whose every vertex but one has only a single child will have height n-1.

Binary search trees are interesting because it is fast to insert elements into them, fast to delete elements from them, and fast to search them (provided they are not too tall and skinny). This contrasts with most collections, which are usually fast for one of these operations but slow for the other two. For example, elements can be inserted into an (unsorted) linked list quickly, but searching or deleting an element from a linked list is slow, while a (sorted) contiguous list can be searched quickly with binary search, but inserting into and deleting elements from it to keep it sorted is slow.

The *binary search tree of T* ADT has as its carrier set the set of all binary search trees whose vertices hold a value of type *T*. It is thus a subset of the carrier set of the binary tree of *T* ADT. The implicit-receiver method set of this ADT includes the implicit-receiver method set of the binary tree ADT. All binary search trees except the empty tree are formed from others using the *insert*() and *delete*() operations described below.

insert(*v*)—Put *v* into a new vertex added as a leaf to the tree, preserving the binary search tree property. If *v* is already in the tree, then do nothing.

delete(*v*)—Remove the vertex holding *v* from the tree while preserving it as a binary search tree. If *v* is not present in the tree, do nothing.

This ADT is the basis for a `BinarySearchTree` class.

18.9 THE BINARY SEARCH TREE CLASS

Every function in the `BinarySearchTree` class is also in the `BinaryTree` class, so `BinarySearchTree` is a sub-class of `BinaryTree` Binary search trees require that values be compared when they are added or removed, so operations specific to `BinarySearchTrees` must be given values that can be compared to one another. This requirement is captured by an interface modelled on the Java `Comparable` interface that contains the `compareTo()` method. The `compareTo(v)` method returns 0 if the instance and `v` have the same value, a negative `int` if the instance is less than `v`, and a positive `int` otherwise. The `Comparable` interface and `BinarySearchTree` class appear in Figure 4 below.

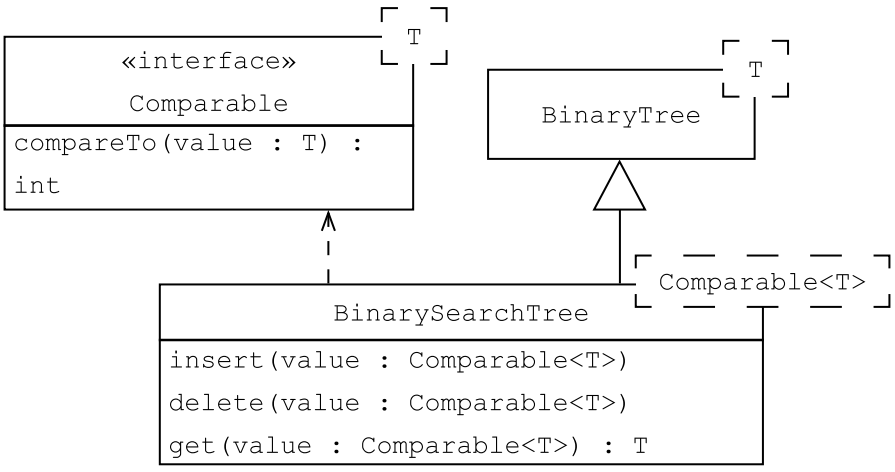


Figure 4: The BinarySearchTree Class

The `insert()` operation puts an element into the tree by making a new child node at the bottom of the tree in a way that preserves the binary search tree's integrity. If the element is already in the tree (as determined by the `compareTo()` method), then the element passed in replaces the value currently stored in the tree. In this way a new record can replace an old one with the same key (more about this in later chapters). The `delete()` operation deletes an element from the tree while preserving the tree's integrity. If the element is not in the tree, then no action is taken. The `get()` operation returns the value stored in the tree that is equal to the element sent in as determined by the `compareTo()` method,

or null if there is no such element. It is intended to fetch a record from the tree with the same key as a dummy record supplied as an argument, thus providing a retrieval mechanism (again, we will discuss this more later).

All of these methods search the tree by starting at its root and moving down the tree, mimicking a binary search. The `insert()` operation takes a path down the tree to the spot where the new element would be found during a search and adds a new leaf node to hold it. In the best case, the sub-tree of the root node where the new element belongs is empty, so only one comparison is required. In the worst case, the new element belongs at the end of a string of n nodes, requiring n comparisons. Empirical studies have shown that when binary search trees are built by a series of insertions and deletions of random data, they are more or less bushy and their height is not too much more than lg n, so on average the number of comparisons done by the `insert()` operation is in $\Theta(\lg n)$.

The `delete()` operation must first find the element to be deleted and then manipulate the tree to remove the node holding the element in such a way that the tree is preserved as a binary search tree. This operation makes $\Theta(1)$ comparisons in the best case, $\Theta(\lg n)$ comparisons in the average case, and $\Theta(n)$ comparisons in the worst case. Finally, the `get()` operation is essentially a search, so it also takes $\Theta(1)$ time in the best case, $\Theta(n)$ time in the worst case, and $\Theta(\lg n)$ time on average.

Note also that even though it is not necessary, the `BinaryTree contains()` operation should be overridden for binary search trees to take advantage of the structure of the tree. This results in faster execution than the exhaustive search conducted by the version of `contains()` implemented for binary trees.

Binary search trees provide very efficient operations except in the worst case. There are several kinds of balanced binary search trees whose insertion and deletion operations keep the tree bushy rather than long and skinny, thus eliminating the poor worst case behavior. We will study several kinds of balanced binary search trees in the next two chapters.

18.10 SUMMARY AND CONCLUSION

Binary search is a very efficient algorithm for searching ordered lists, with average and worst case complexity in $\Theta(\lg n)$. We can represent the workings of binary search in a binary tree to produce a full binary search tree. Binary search trees have several interesting properties and provide a kind of collection that features excellent performance for addition, deletion, and search, except in the worst case. We can also traverse binary search trees inorder to access the elements of the collection in sorted order.

18.11 REVIEW QUESTIONS

1. Why can recursion be removed from the binary search algorithm without using a stack?
2. If a binary tree is made from an ordered list of 100 names by placing them into the tree to mimic a binary search as discussed in the text, what is the height of the resulting tree?
3. Approximately how many comparisons would be made by binary search when searching an array of one million elements in the best, worst, and average cases?
4. What advantage does a binary search tree have over collections like `ArrayList` and `LinkedList`?

18.12 EXERCISES

1. A precondition of binary search is that the searched array is sorted. What is the complexity of an algorithm to check this precondition?
2. Write and solve a recurrence relation for the worst case complexity of the binary search algorithm.
3. *Interpolation search* is like binary search except that it uses information about the distribution of keys in the array to choose a spot to check for the key. For example,

suppose that numeric keys are uniformly distributed in an array and interpolation search is looking for the value *k*. If the first element in the array is *a* and the last is *z*, then interpolation search would check for *k* at location (*k*-*a*)/(*z*-*a*) * (*array.length*-1). Write a non-recursive linear interpolation search using this strategy.

4. Construct a binary search tree based on the order in which elements of a list containing the numbers one to 15 would be examined during a binary search, as discussed in the text.
5. Draw all the binary search tree that can be formed using the values *a*, *b*, and *c*. How many are full at every level but the last?
6. The `BinarySearchTree insert()` operation does not attempt to keep the tree balanced. It simply work its way down the tree until it either finds the node containing the added element or finds where such a node should be added at the bottom of the tree. Draw the binary search tree that results when values are added to the tree in this manner in the order *m*, *w*, *a*, *c*, *b*, *z*, *g*, *f*, *r*, *p*, *v*.
7. Write Java code to implement the `Comparable` interface and the skeleton of the `BinarySearchTree` class.
8. Rewrite the `contains()` method from `BinaryTree` to take advantage of the structure of a binary search tree and add it as a method to `BinarySearchTree`.
9. Write the `BinarySearchTree insert()` method in Java using the strategy explained in Exercise 6 above.
10. Write the `BinarySearchTree get()` method in Java.
11. Write the `BinarySearchTree delete()` method must preserve the essential property of a binary search tree, namely that the value at each node is greater than or equal to the values at the nodes in its left sub-tree, and less than or equal to the values at the nodes in its right sub-tree. In deleting a value, three cases can arise:
    - The node holding the deleted value has no children; in this case, the node can simply be removed.
    - The node holding the deleted value has one child; in this case, the node can be removed and the child of the removed node can be made the child of the removed node's parent.
    - The node holding the deleted value has two children; this case is more difficult. First, find the node holding the successor of the deleted value. This node will always be the left-most descendent of the right child of the node holding the deleted value. Note that this node has no left child, so it has at most one child. Copy the successor value over the deleted value in the node where it resides, and remove the redundant node holding the successor value using the rules for removing a node with no children or only one child above.

    a) Use this algorithm to remove the values *v*, *a*, and *c* from the tree constructed in exercise 6 above.

    b) Write the `delete()` method in Java using the algorithm above.

18.13 REVIEW QUESTION ANSWERS

1. Recursion can be removed from the binary search algorithm without using a stack because the binary search algorithm is tail recursive, that is, it only calls itself once as its last action on each activation.
2. If a binary tree is made from an ordered list of 100 names by placing them into the tree to mimic a binary search as discussed in the text, the height of the resulting tree is $\lfloor \lg 100 \rfloor = 6$.
3. When searching a list of one million elements in the best case, the very first element checked would be the key, so only one comparison would be made. In the worst case, $\lfloor \lg 1000000 \rfloor + 1 = 20$ comparison would be made. In the average case, roughly $\lfloor \lg 1000000 \rfloor = 19$ comparison would be made.
4. An `ArrayList` and a `LinkedList` allow rapid insertion but slow deletion and search, or rapid search (in the case of an ordered `ArrayList`) but slow insertion and deletion. A binary search tree allows rapid insertion, deletion, and search.

19 AVL TREES

19.1 INTRODUCTION

Binary search trees are an excellent data structure because insertion, deletion, and search can all be done very quickly, *provided* the tree does not become too long and skinny. We would have an even better data structure if we could ensure that our binary search trees could never get long, thus avoiding worst case behaviors. Trees with this characteristic are called balanced.

> **Balanced tree**: a tree such that for every node, the height of its sub-trees differ by at most some constant value.

There are several sorts of balanced trees. We will consider one kind of balanced binary tree (AVL trees) in this chapter and one kind of balanced non-binary tree (2-3 trees) in the next.

19.2 BALANCE IN AVL TREES

AVL trees were introduced in 1962 by G. M. Adelson-Velsky and E. M. Landis (hence their name). They are defined as follows.

> The **balance factor** of a node is the height of the left sub-tree minus the height of the right sub-tree, with the height of the empty tree defined as -1.
>
> An **AVL tree** is a binary search tree in which the balance factor at each node is -1, 0, or 1. The empty tree is an AVL tree.

AVL trees all of whose nodes have balance factors of 0 are perfectly balanced. But some somewhat lop-sided trees are still AVL trees, and some trees that may appear balanced are not AVL trees. Consider the examples in Figure 1 (balance factors are shown above the nodes). Both are binary search trees, but only the tree on the left is an AVL tree. (In the tree on the right, the node holding 10 has a balance factor of -2.)

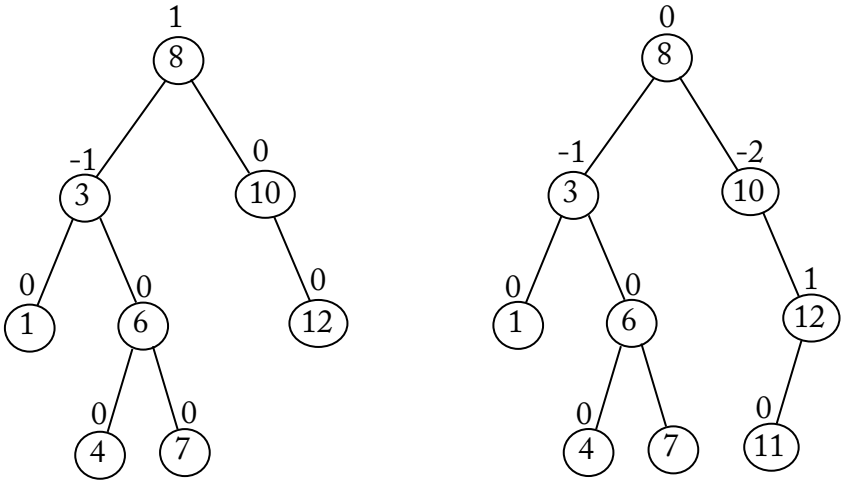


Figure 1: Two Binary Search Trees; One AVL Tree

Despite the look of some AVL trees, the constraint on balance factors ensures every AVL tree with at least two nodes has height less than 1.5 lg n, where n is the number of nodes in the tree.

19.3 INSERTION IN AVL TREES

Inserting a value into an AVL tree begins the way that binary search tree insertion does: starting at the root, the key is compared to node values to follow a path down the tree until arriving at the bottom, where the inserted value is placed in a new leaf node. The new leaf node will make a path to the root longer, altering at least one node's balance factor, and perhaps all nodes' balance factors on the path back to the root. If all balance factors are still within constraints, then balance factors are adjusted and insertion is complete. However, some nodes' balance factors may become 2 or -2. In this case, the tree must be rebalanced by rearranging some of its nodes. It turns out that this rearrangement, made at the node with an illegal balance factor closest to the inserted node, results in a rebalanced tree with the same height as before.

Trees are rearranged by applying one of four *rotations*. The rotations applied for a balance factor of 2 are depicted in Figure 2. The rows show the three possibilities when an insertion results in some node having a balance factor of 2. The sub-tree where the insertion took place, denoted t_i, has a double baseline; balance factors are shown above nodes; and heights of sub-trees are shown beside them.

In the first row, the left sub-tree of the node with balance factor 2 has balance factor 1, which can only occur if the insertion was made in its left sub-tree. In this case an *R* rotation is used to rebalance the tree. Note that the balance factors in the sub-trees t_i, t_1, and t_2 are not changed, nor are the balance factors above the root node (because after the rotation, the height of the sub-tree is what it was before the insertion). Thus these changes are restricted to a small portion of the tree and can be made relatively easily.

The second and third rows show the cases when the left child of the root has a balance factor of -1. Then the insertion must have been made in the right sub-tree of the left child of the root, though it might have been made in this sub-tree's left sub-tree (row 2) or its right sub-tree (row 3). In both cases an *LR* rotation is used to restore balance. Note again that the balance factors in sub-trees t_i, t_1, t_2, and t_3 are unaffected by the change, as are the balance factors above the root (again because the sub-tree's height is returned to what is was before the insertion). So again these changes occur in a small region of the tree and can be made quite easily.

When a node has a balance factor of -2, the other two rotations, *L* and *RL*, are used. These rotations are mirror images of the *R* and *LR* rotations.

Insertion requires modification of balance factors on the path from the new leaf to the root, and possibly a rotation to rebalance the tree. The number of operations needed for this is proportional to the height of the tree, which as mentioned above less than 1.5 lg n, where n is the number of nodes in the tree. On the other hand, insertion always takes place at the bottom of the tree, and it turns out that the shortest path in an AVL tree always has length proportional to lg n. Hence the every case complexity of insertion is in $\Theta(\lg n)$.

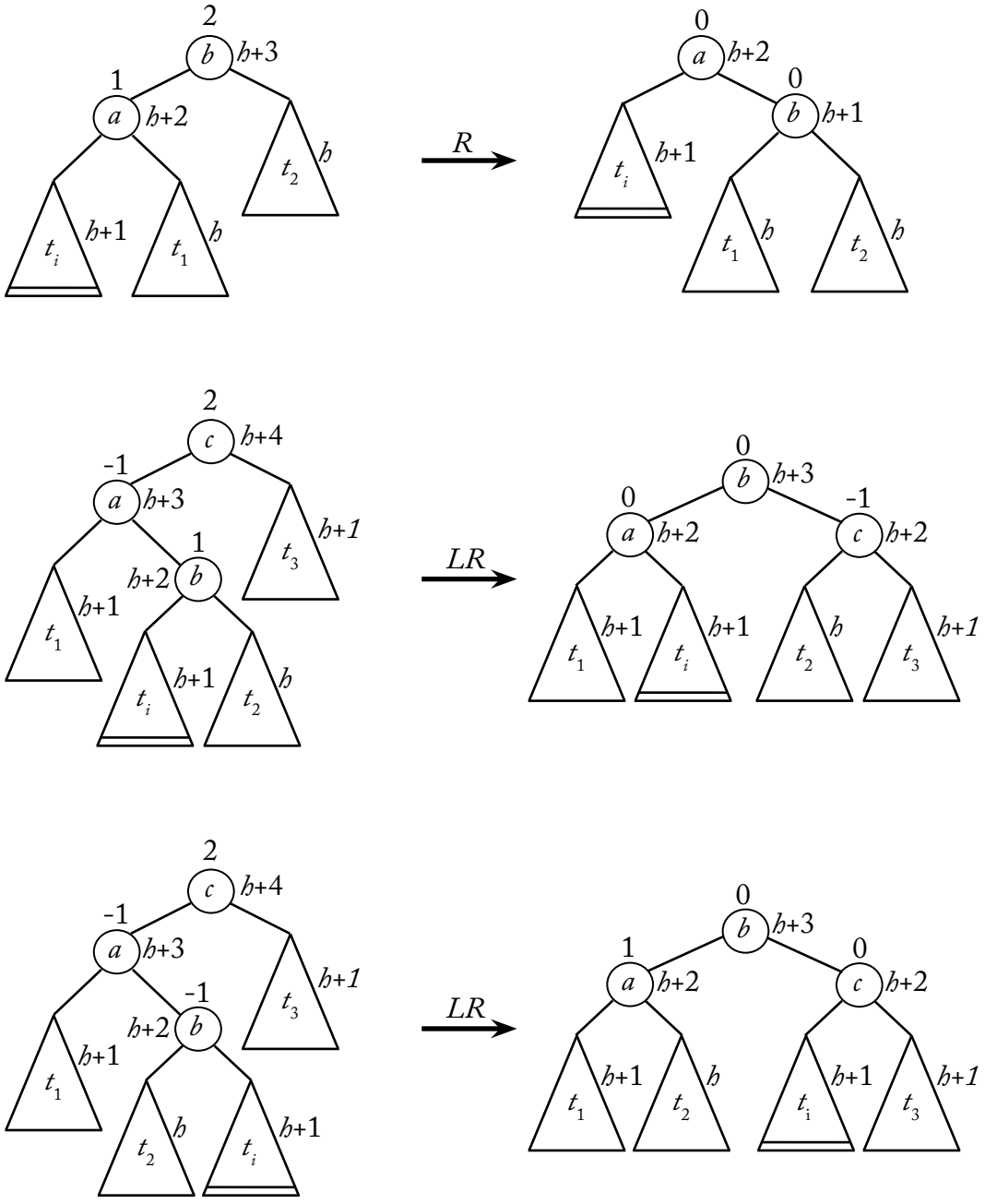


Figure 2: AVL Tree 2 Rotations

19.4 DELETION IN AVL TREES

Deletion in AVL trees resembles deletion in ordinary binary search trees: a search is made from the root down the tree to locate the node holding the deleted value, the *target* node. If this node has at most one child, then it can be deleted and its child, if any, attached to its parent in its place. If the target node has two children, then its successor node can be found by descending one node to the right then all the way down the tree to the left. This successor node as only a right child, so it can be deleted and its value copied over the value in the target node, thus removing the deleted value and preserving the tree as a binary search tree.

In an AVL tree, after a node is removed, the tree may become unbalanced. As with insertion, the path back to the root must be retraced, adjusting balance factors and perhaps rearranging the tree to rebalance it.

Rebalancing is done using the same four rotations as for insertion. During insertion, if a node has a balance factor of 2 (or -2), then its left (or right) child must have a balance factor of 1 or -1, and this determines which rotation to use. During deletion, the child of a node with a balance factor of 2 (or -2) may have a left (or right) child with a balance factor of 0. In such cases an *R* (or *L*) rotation is used to rebalance the tree, but otherwise deciding which rotation to use is the same as for insertion.

Figure 3 shows several operations on an AVL tree illustrating insertions and deletions that force rebalancing. The first insertion makes the node holding *c* have a balance factor of 2 and a left node with a balance factor of 1, calling for an *R* rotation. Notice that after this rotation the subtree is perfectly balanced. After inserting *k* the node holding *i* has a balance factor of -2 and its right child has a balance factor of 1. This calls for an *RL* rotation, which again brings the subtree into perfect balance. Deleting *e* causes the node holding the successor of *e* (namely *f*) to be removed, and the successor to be copied into the node holding *e*. This makes the node now holding *f* have a balance factor of 2 with a left child whose balance factor is 0. This means an *R* rotation should be applied. Afterwards, the subtree is not perfectly balanced, but all balance factors are again between -1 and 1.

This example was designed to show how various operations cause rotations to occur, but very often operations do not cause any rotations. For examples, deleting any node but *a* in the last tree in Figure 3 would not cause the tree to need to be rebalanced.

19.5 THE EFFICIENCY OF AVL OPERATIONS

During insertion, at most one rebalancing operation is needed along the path to the root. In deletion, a rebalancing operation may be needed for many nodes on the path from the deleted node to the root. Nevertheless, all node balance factor adjustments and rotations are made only on nodes in the path from the bottom of the tree where a node is removed to the root, which contains fewer than 1.5 lg n nodes, as noted before. As with insertion, all deletions occur at a leaf, and the shortest path from the root is proportional to lg n, so at least $\Theta(\lg n)$ nodes must be processed. Hence the every case complexity of deletion is in $\Theta(\lg n)$.

AVL trees are binary search trees, and searching them works exactly as it does in ordinary binary search trees. The best case complexity of this operation is $\Theta(1)$ (when the desired value is at the root). The worst case complexity of searching an ordinary binary search tree is n because the tree may have height n. But AVL trees have height bounded by 1.5 lg n, so the worst case complexity of searching an AVL tree is in $\Theta(\lg n)$. It turns out that the average case complexity for searching an AVL tree is also in $\Theta(\lg n)$. We thus see that insertion, deletion, and searching in AVL trees all take $O(\lg n)$ time, so AVL trees are in ideal data structure for applications where all three of these operations must be efficient.

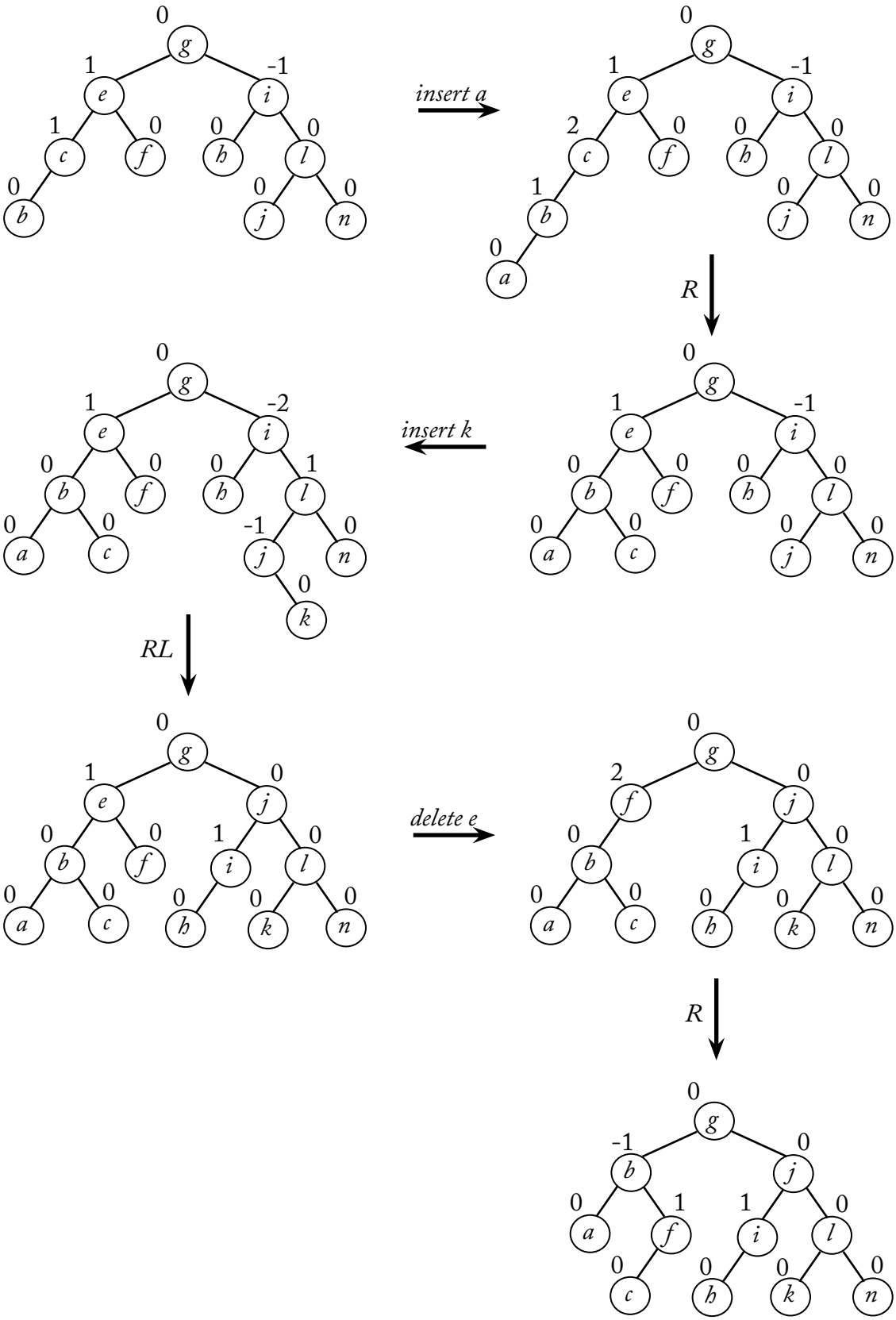


Figure 3: Insertion, Deletion, and Rotations

19.6 THE AVL TREE CLASS

An AVL tree is a kind of binary search tree, so the `AVLTree` class should be a sub-class of the `BinarySearchTree` class. AVL tree nodes need an extra field for monitoring node balance factors. This field could be the balance factor itself, but it is simpler to add a `height` field to each `BinarySearchTree` node and calculate each node's balance factor from its children's `height` fields when it is needed.

The `AVLTree` constructor should create only empty AVL trees. The `insert()` and `delete()` methods must of course be overridden to incorporate node height adjustment and rebalancing into these algorithms. The `height()` method (inherited through `BinarySearchTree` from `BinaryTree`) should also be overridden to take advantage of the fact that each node maintains its height. This changes this method from an $\Theta(n)$ to an $\Theta(1)$ time method.

Both the `insert()` and `delete()` methods must make adjustments to nodes along the path from the bottom of the tree to its root. This path can be maintained in a stack, but it is probably easier to implement these methods using recursion.

19.7 SUMMARY AND CONCLUSION

AVL trees are balanced binary search trees guaranteeing that insertion, deletion, and search take $O(\lg n)$ time. Each node of an AVL tree has a balance factor, which is the difference between the heights of its sub-trees. The absolute value of the balance factor at each node of an AVL tree never exceeds one, meaning that the difference in height between any two sub-trees of a node is never more than one. This entails that the height of an AVL tree with more than two nodes is less than $1.5 \lg n$, where n is the number of nodes in the tree.

Insertion, deletion, and searching work as they do in binary search trees, except that after a node is added or removed the tree may need to be rebalanced. Rebalancing occurs at nodes along the path from the inserted or deleted node to the root, and is done by local transformations called *rotations*. There are four rotations applied based on the balance factors of the child nodes of the node where the imbalance occurs.

An `AVLTree` class can be implemented as a sub-class of `BinarySearchTree`, with only three methods needing to be overridden.

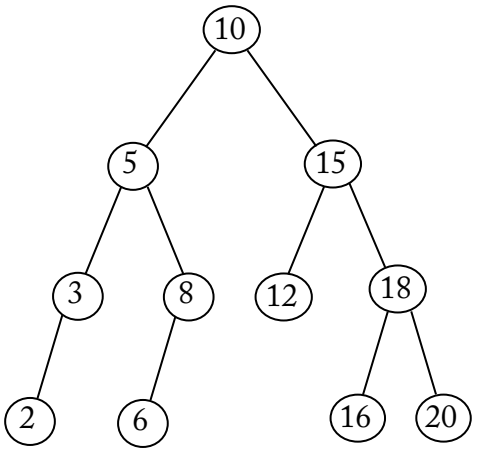
19.8 REVIEW QUESTIONS

1. What would an AVL tree look like if all its nodes have a balance factor of 0?
2. Approximately how many comparisons would be made in the worst case when searching an AVL tree holding one million values?
3. What advantage does an AVL tree have over collections like `ArrayList` and `LinkedList`?

19.9 EXERCISES

1. Draw all the AVL trees with one, two, three, and four nodes.
2. Draw an AVL tree of height four with as few nodes as possible. How many nodes did you need? Is it true that four is less than 1.5 lg n, where n is the number of nodes in your tree?
3. What is the maximum number of nodes in an AVL tree of height four? What is the maximum number of nodes in an AVL tree of height h?
4. Using the diagram in Figure 2 as a guide, draw diagrams illustrating the L and RL rotations.
5. Make concrete example trees illustrating the application of the four AVL tree rotations. Your four examples should show the trees before and after the rotations, with all node balance factors included.

6. Draw a series of AVL trees illustrating the result of adding the following numbers in order to the empty tree: 3, 2, 1, 10, 11, 7, 5, 8, 12, 9.
7. Starting with the tree below, draw a series of AVL trees illustrating the result of removing the following numbers in order: 2, 15, 3, 10, 18, 16, 20, 5.



8. Write Java code for the skeleton of the `AVLTree` class as a sub-class of the `BinarySearchTree` class. Make an AVL node class as a sub-class of the binary tree node class as well.
9. Override the `height()` method from `BinaryTree` in the `AVLTree` class.
10. Write rotation methods in the AVL node class, along with methods to set the height of a node, get the balance factor at a node, and rebalance a node.
11. Override the `BinarySearchTree insert()` method in the `AVLTree` class. You may want to write a recursive helper function to do the real work.
12. Override the `BinarySearchTree delete()` method in the `AVLTree` class. Recursion is especially helpful for this method.

19.10 REVIEW QUESTION ANSWERS

1. An AVL tree whose every node has balance factor 0 must be such that every node has zero or two children. This is the definition of a full binary tree. A full binary tree has every level completely full and is perfectly balanced.
2. An AVL tree holding one million elements has height less than 1.5 lg 1000000 ≈ 29.9. Hence fewer than 30 comparisons would be made in searching this AVL tree.
3. An `ArrayList` and a `LinkedList` allow rapid insertion ($\Theta(1)$) but slow deletion and search ($O(n)$), or rapid search (in the case of an ordered `ArrayList`, $O(\lg n)$) but slow insertion and deletion ($O(n)$). An AVL tree allows rapid insertion, deletion, and search (all $O(\lg n)$) even in the worst case.

20 2-3 TREES

20.1 INTRODUCTION

AVL trees use a rotation technique to maintain balance in binary search trees. Another approach is to maintain perfect balance in search trees by relaxing the constraint on the number of children (and keys) that a tree node can have. In general, a tree node might be allowed to have zero or two to m children; this produces a data structures called a *B-tree*. We will focus in this chapter on the case when $m = 3$, producing trees called *2-3 trees*.

Because 2-3 trees are perfectly balanced, modifying them is fast, and because they maintain the search tree property among the values at their nodes, searching them is also fast.

20.2 PROPERTIES OF 2-3 TREES

2-3 trees were invented in 1970 by John Hopcroft. They are defined as follows.

> A tree is **perfectly balanced** if all its leaves are on the same level; that is, the path from the root to any leaf is always the height of the tree.
>
> **2-3 tree**: a perfectly balanced tree whose every node is either a *2-node* with one value v and zero or two children, such that every value in its left sub-tree is less than v and every value in its right sub-tree is greater than v, or a *3-node* with two values v_1 and v_2 and zero or three children such that every value in its left-most sub-tree is less than v_1, every value in its middle sub-tree is greater than v_1 and less than v_2, and every value in its right-most sub-tree is greater than v_2.

Figure 1 shows an example of a 2-3 tree.

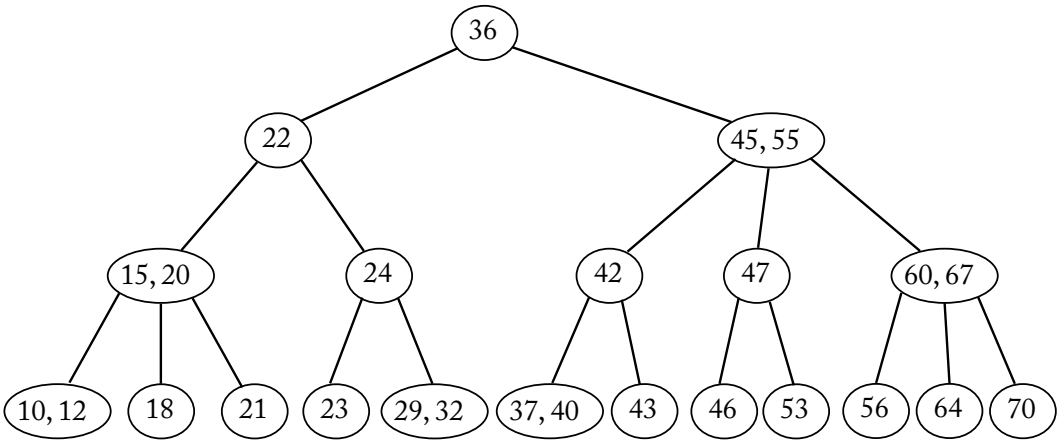


Figure 1: A 2-3 Tree

The shortest 2-3 tree holding the most values would be one that has only 3-nodes. In a tree with only 3-nodes, there is one node at the root, three 3-nodes at level one, nine 3-nodes at level two, and in general 3^h 3-nodes at level h. There are two values stored in each node. Hence the number of values stored in the entire tree is

$$2 \cdot \sum_{i=0}^{h} 3^i = 2 \cdot (3^{h+1}\text{-}1)/2 = 3^{h+1}\text{-}1.$$

On the other hand, the tallest tree holding the fewest values would have only 2-nodes. A tree with only 2-nodes has one 2-node at the root, two 2-nodes at level one, four 2-nodes at level two, and in general, 2^h 2-nodes at level h. There is one value stored in each node, so the number of values stored in the entire tree is

$$\sum_{i=0}^{h} 2^i = 2^{h+1}-1.$$

Hence we have the following relationship between the n values in a 2-3 tree and its height h.

$$2^{h+1}-1 \le n \text{ and } n \le 3^{h+1}-1$$
$$2^{h+1} \le n + 1 \text{ and } n + 1 \le 3^{h+1}$$
$$h + 1 \le log_2 (n+1) \text{ and } log_3 (n+1) \le h + 1$$
$$log_3 (n+1) \le h + 1 \le log_2 (n+1)$$
$$log_3 (n+1) - 1 \le h \le log_2 (n+1) - 1$$

In other words, the height of a tree with n values must be (roughly) between $log_3\ n$ and $log_2\ n$, which means the height grows no faster than lg n. This implies that the height of a 2-3 tree is in Θ(lg n). Therefore any operation that takes time proportional to the height of a 2-3 tree with n values is in Θ(lg n).

Lets consider search in a 2-3 tree. At each node, at most two values must be compared to the search value, and the search either halts successfully or continues in one of the sub-trees of the node. This continues until either the search value is found or the bottom of the tree is reached. The number of comparisons is thus at most $2 \cdot (h+1)$ which we know from the argument above is in O(lg n).

We will soon see that insertion and deletion in 2-3 trees are also done in time proportional to the height of the tree, and are thus also in Θ(lg n).

20.3 INSERTION IN 2-3 TREES

Inserting a value into a 2-3 tree begins the way that binary search tree and AVL tree insertion does: starting at the root, the key is compared to node values to follow a path down the tree until arriving at the bottom, where the inserted value is placed in a leaf node. If the leaf node is a 2-node, then the leaf becomes a 3-node and the operation is complete. However, if the node is 3-node, then inserting a value into it makes it into a 4-node, which of course is not allowed in a 2-3 tree. In this case, the 4-node (which has two values and four empty sub-trees), is split into three 2-nodes: the middle value of the 4-node becomes a new 2-node whose left sub-tree is a 2-node holding the left-most value in the 4-node, and whose right sub-tree is 2-node holding the right-most value in the 4-node. This transformation is shown below in Figure 2.

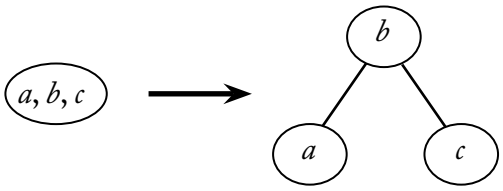


Figure 2: Splitting a Leaf 4-Node

After this transformation, the problem is that the leaf level of the tree now contains a sub-tree of height one, so the tree is unbalanced. Balance is restored by incorporating the root of this sub-tree into the level above. If the parent node of the newly created 2-node is a 2-node, then the new 2-node is added to its parent 2-node, making the parent into a 3-node and balancing the tree. However, if the parent of the newly created 2-node is a 3-node, then folding the new 2-node into its parent creates a 4-node, and the problem we had before recurs. We can solve it the same way: make the 4-node into a 2-node with two children, then rebalance. Figure 3 shows how to split a 4-node when it has sub-trees.

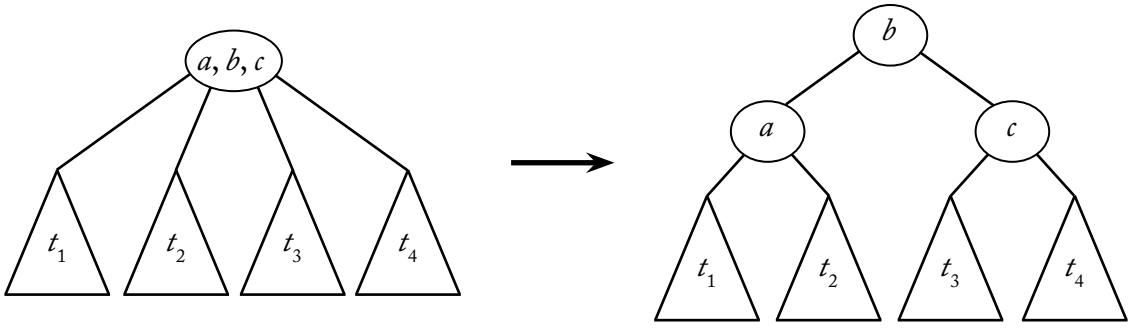


Figure 3: Splitting a 4-Node With Children

This technique of propagating split 4-nodes up the tree continues as far as necessary. If it reaches the root, then the new 2-node created from splitting a 4-node becomes the new root of the entire tree. This is the only way that the height of a 2-3 tree can increase and it is the key to keeping it balanced: the tree increases its height only at its root. If the tree were to grow at the bottom, it would be very difficult to keep it perfectly balanced, but because it only grows at the root it stays perfectly balanced without any further manipulations.

Figure 4 shows an example to illustrate this process at its most transformative: a value inserted at a leaf causes 4-nodes to split all the way up the tree to its root, increasing the height of the tree.

Insertion requires a search down the tree to find the leaf where the inserted value is placed; as noted above, this requires $\Theta(\lg n)$ operations. If splits of 4-nodes are necessary, these must occur along the path from the leaf where the insertion takes place back to the root, a path with at most $h+1$ nodes that must be split. Each split requires at most some fixed number of operations, hence this transformation requires $O(\lg n)$ operations as well. The time complexity for insertion is thus in $\Theta(\lg n)$.

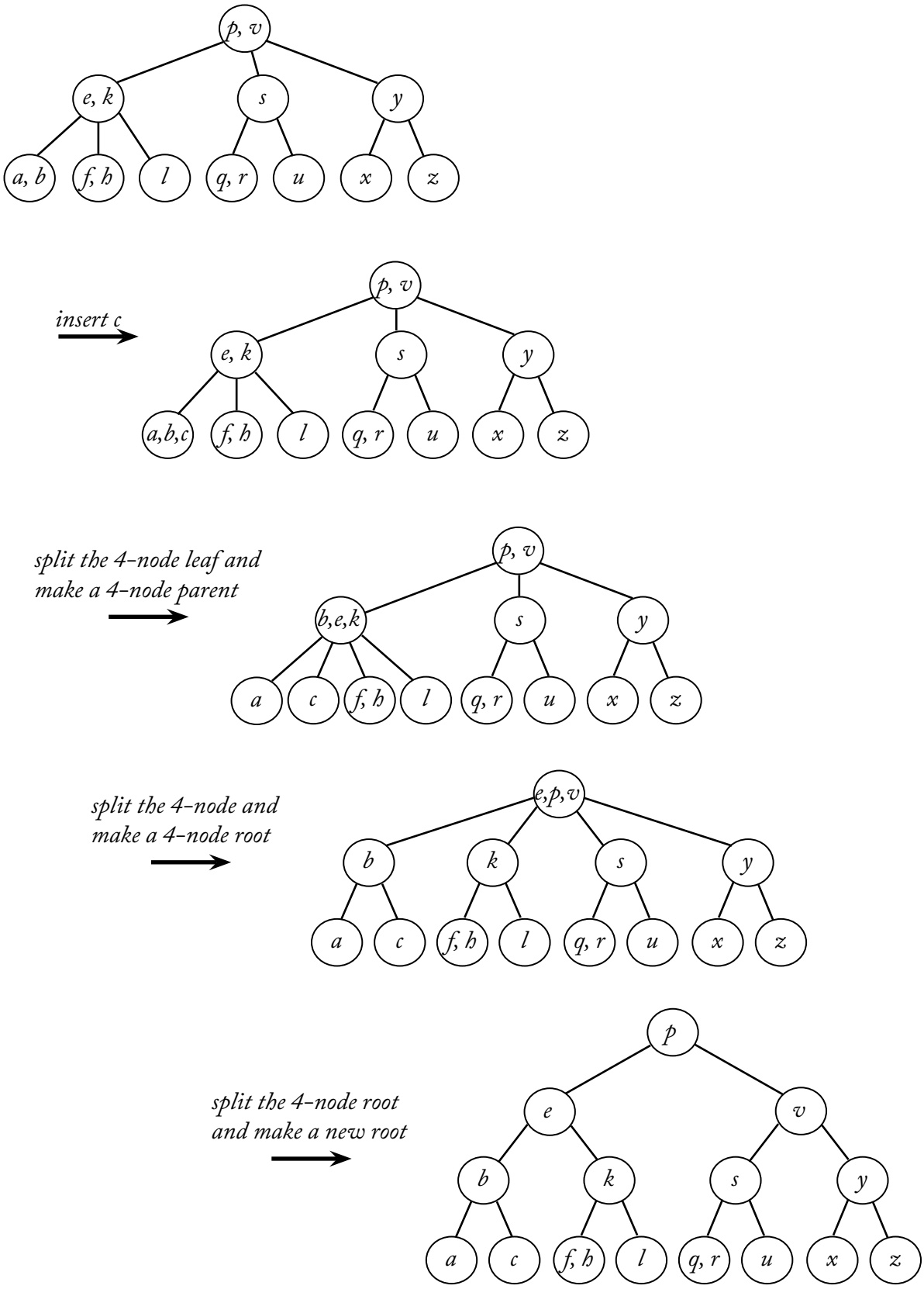


Figure 4: Inserting At a Leaf and Propagating 4-Node Splits Upwards

20.4 DELETION IN 2-3 TREES

Deletion in 2-3 trees resembles deletion in ordinary binary search trees and AVL trees: a search is made from the root down the tree to locate the node holding the deleted value, the *target* node. If this node is not a leaf, then the node with the deleted value's successor is found, the successor value is copied into the target node over the deleted value, and the duplicate successor is removed from the leaf. Thus values are always removed from leaves.

Akin to other search trees, the successor of a value in a 2-3 tree is be found by descending one node into the right sub-tree of the value (either the middle or the right-most sub-tree, depending on whether the value is the left-most or right-most), then all the way down the sub-tree to the left. This successor node must be a leaf with the successor as its left-most value.

When a value is removed from a leaf, then if the node was a 3-node, it becomes a 2-node and the deletion is complete. If the leaf-node was a 2-node, then deleting its value makes it into a 1-node, which we define to be a node with no values and possibly one sub-tree (which we may suppose is the left-most sub-tree). Of course there can be no 1-nodes in a 2-3 tree, so this problem must be fixed. The process of handling 1-nodes is the same for internal nodes as for leaves, so we consider it as a general problem.

If a node has a child that is a 1-node, then it is handled by the first of the following cases that obtains.

- If the 1-node has a sibling that is a 3-node, then the 1-node borrows a value and a sub-tree from its 3-node sibling, with needed alterations in the parent value(s) as well. This case is illustrated in Figure 5 below. This figure only shows the case where there is one adjacent sibling and it is is 3-node. Similar transformations are made when there are two siblings and the 3-node is not adjacent.

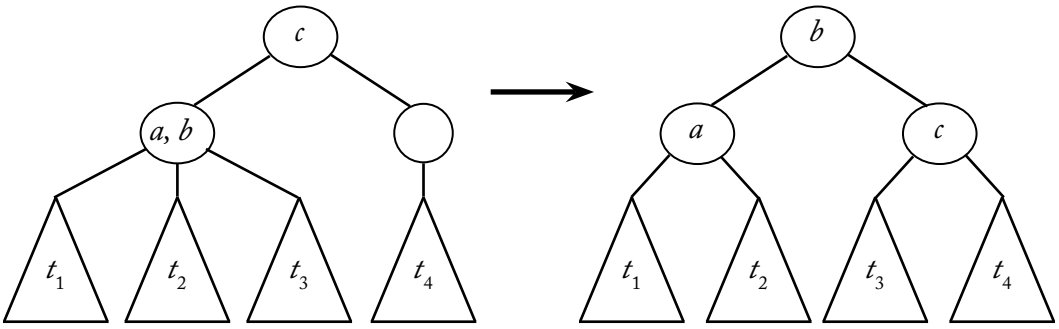


Figure 5: Borrowing from a Sibling to Transform a 1-Node

- If the 1-node's parent is a 3-node, then the 1-node borrows a value and a sub-tree from its parent, as illustrated in Figure 6. Again, other cases with the 1-node in a different location are similar.

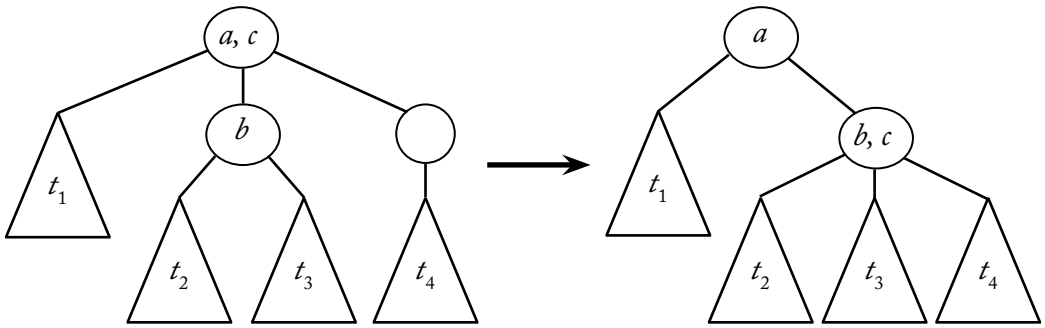


Figure 6: Borrowing from a Parent to Transform a 1-Node

- If a 1-node has a 2-node parent and a 2-node sibling, then it combines with its sibling, borrowing the value from its parent. This makes its parent into a 1-node with a single 3-node child. The new 1-node must be handled by its parent. This case is shown in Figure 7. Again, the 1-node could be on the other side.

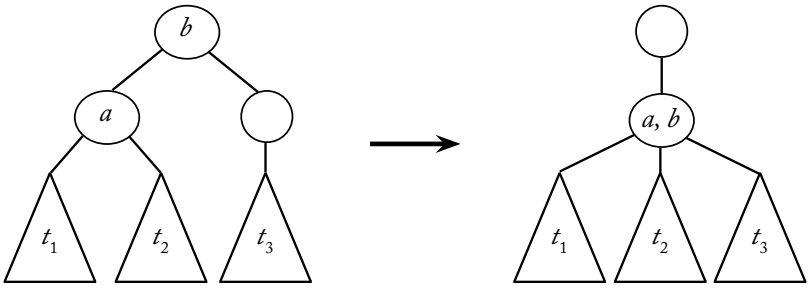


Figure 6: Borrowing from a Parent to Transform a 1-Node

- Finally, if changes propagate to the root and it becomes a 1-node, then the new root of the tree becomes the child of the 1-node. In other words, the tree becomes shorter. As with insertion, the tree only shrinks at its root, guaranteeing that it remains completely balanced during deletions.

Figure 8 illustrates a few deletions and how a 2-3 tree is transformed when 1-nodes appear. This diagram shows two transformations. In the first, a 1-node (the leaf where the deletion occurs) has a 2-node sibling and 2-node parent, so it must borrow from its parent and combine with its sibling to make a 3-node with a 1-node parent. This 1-node parent has a 3-node sibling, so it can borrow from its sibling.

In a deletion, a search must progress from the root to the bottom of the tree, which takes $\Theta(\lg n)$ time. If the tree must be transformed to deal with 1-nodes, this occurs in a constant number of operations for each of at most h nodes (where h is the height of the tree) back along the path to the root. Hence these modifications take $O(\lg n)$ time. We thus see that deletion in 2-3 trees, like search and insertion, is done in $\Theta(\lg n)$ time.

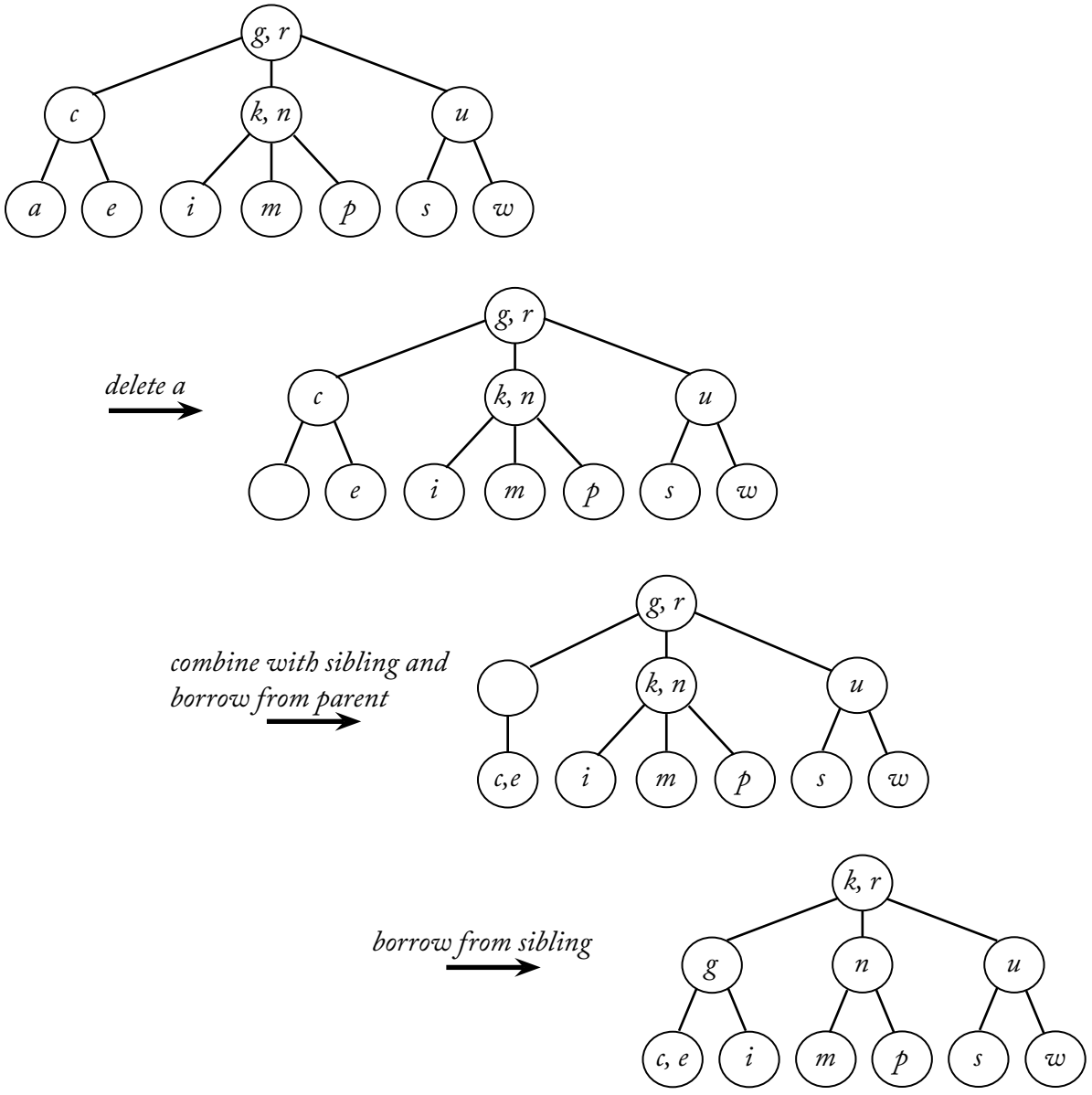


Figure 8: Transformation Resulting from a Deletion

20.5 THE TWO-THREE TREE CLASS

A 2-3 tree is not a kind of binary tree, so the a `TwoThreeTree` class is not a sub-class of the `BinaryTree` class. However, since 2-3 trees are search trees, the `TwoThreeTree` class has almost the same operations as the `BinarySearchTree` or `AVLTree` classes. Figure 9 shows the `TwoThreeTree` class.

One difference between the `TwoThreeTree` class and the `BinaryTree`-descended search tree classes is that the former lacks methods and iterators for traversing the tree in preorder or post order. Although these traversal orders are defined for 2-3 trees and we could add methods to perform them, they are somewhat less useful than they are for binary trees, so we have left them out.

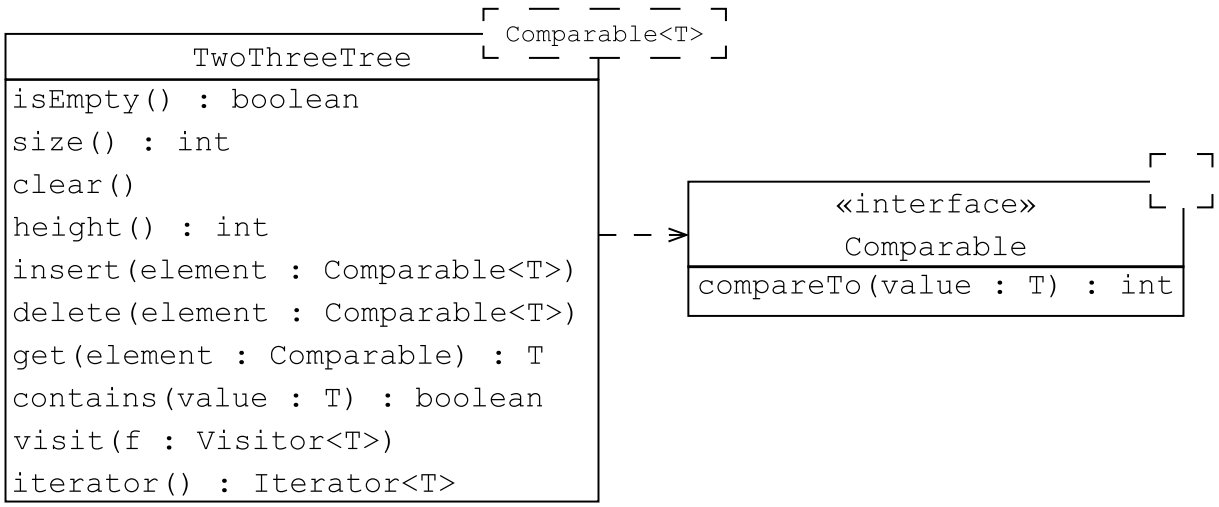


Figure 9: The TwoThreeTree Class

20.6 SUMMARY AND CONCLUSION

Two-three trees are perfectly balanced search trees guaranteeing that insertion, deletion, and search take $O(\lg n)$ time. Each node of a 2-3 tree has either one value and two subtrees (a 2-node), or two values and three sub-trees (a 3-node). In either case, a value in a node is greater than any value in its left subtree and less than any value in its right subtree, which is what makes 2-3 trees search trees. All 2-3 trees are perfectly balanced, meaning that the length of any path from a leaf to the root is the same: the height of the tree.

Insertion, deletion, and searching work much as they do in binary search trees, except that 2-3 trees grow and shrink only at their root, ensuring that they remain perfectly balanced.

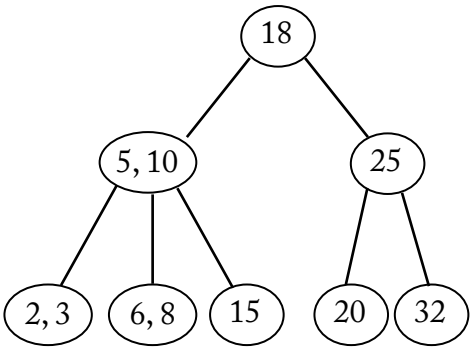
The `TwoThreeTree` class is not a sub-class of `BinaryTree` because 2-3 trees are not binary trees. Nevertheless, the methods in the `TwoThreeTree` class are similar to those in the `BinarySearchTree` or `AVLTree` classes.

20.7 REVIEW QUESTIONS

1. Can a 2-3 tree have a root that is a 2-node?
2. Approximately how many comparisons would be made in the worst case when searching a 2-3 tree holding one million values? How many comparisons would be made in the best case, assuming that the value searched for is not in the tree?
3. What advantage does a 2-3 tree have over a plain binary search tree?

20.8 EXERCISES

1. Draw examples of the shortest and tallest possible 2-3 trees with eight values.
2. Draw a schematic (a tree with dots instead of actual values) of a 2-3 tree of height three with as few values as possible. How many values are in this tree? Draw a schematic of a 2-3 tree of height three with as many values as possible. How many values are in this tree?
3. Using the diagram in Figure 5 as a guide, draw a diagram illustrating how to deal with a 1-node that is the first child of a 3-node parent whose second child is a 2-node and whose third child is a 3-node.
4. Using the diagram is Figure 6 as a guide, draw a diagram illustrating how to deal with a 1-node that is the middle child of a 3-node parent whose other two children ar 2-nodes.
5. Draw a series of 2-3 trees illustrating the result of adding the following numbers in order to the empty tree: 3, 2, 1, 10, 11, 7, 5, 8, 12, 9.
6. Starting with the tree below, draw a series of AVL trees illustrating the result of removing the following numbers in order: 2, 15, 3, 10, 18, 16, 22, 5, 20, 25, 30.



7. Write Java code for the skeleton of the `TwoThreeTree` class. You will need a `TreeNode` class whose instances can be 1-nodes, 2-nodes, or 3-nodes as well.
8. Write the `isEmpty()`, `clear()`, `size()`, and `height()` methods in the `TwoThreeTree` class.

9. Write the `insert()` method in the `TwoThreeTree` class. You may want to write a recursive helper method in `TreeNode` (along with other helper methods in this class) to do the real work.
10. Write the `contains()` and `get()` methods in the `TwoThreeTree` class. You may want to write a recursive helper method in `TreeNode` (along with other helper methods in this class) to do the real work.
11. Write the `delete()` method in the `TwoThreeTree` class.
12. Write the `visit()` and `iterator()` methods in the `TwoThreeTree` class.

20.9 REVIEW QUESTION ANSWERS

1. The root of a 2-3 tree can be either a 2-node or a 3-node. In fact it is always the case that the first two values added to an empty 2-3 tree will first produce a 2-node, then a 3-node.
2. A 2-3 tree holding one million elements with 2-nodes for all but some of its leaves (the tallest 2-3 tree with a million elements) will have height $\lfloor \lg 1000001 \rfloor - 1 = 19$, so there will be about 20 comparisons in the worst case. A 2-3 tree with 3-nodes for all but some of its leaves (the shortest 2-3 tree with a million elements) will have height $\lfloor \log_3 1000001 \rfloor - 1 = 11$, so there would be about 12 comparisons in the best case when searching for a value not in the tree.
3. A plain binary search tree can become completely unbalanced, resulting in $O(n)$ worst case behavior for insertion, deletion, and searching. A 2-3 tree is always completely balanced, so its worst case behavior for insertion, deletion, and searching is always in $\Theta(\lg n)$.

21 SETS

21.1 INTRODUCTION

Lists have a linear structure and trees have a two-dimensional structure. We now turn to unstructured collections. The simplest unstructured collection is the set.

> **Set**: An unordered collection in which an element may appear at most once.

We first review the set ADT and then discuss ADT implementation.

21.2 THE SET ADT

The *set of T* abstract data type is the ADT of sets of elements of type *T*. Its carrier set is the set of all sets of *T*. This ADT is the abstract data type of sets that we all learned about starting in grade school. Its operations are exactly those we would expect (and more could be included as well). Note this is not an implicit-receiver method set.

$e \in s$—Return true if e is a member of the set s.
$s \subseteq t$—Return true if every element of s is also an element of t.
$s \cap t$—Return the set of elements that are in both s and t.
$s \cup t$—Return the the set of elements that are in either s or t.
$s - t$—Return the set of elements of s that are not in t.
$s == t$—Return true if and only if s and t contain the same elements.

The set ADT is so familiar that we hardly need discuss it. Instead, we can turn immediately to the set interface that all implementation of the set ADT will use.

21.3 THE SET INTERFACE

The `Set` interface appears in Figure 1. The `Set` interface is a sub-interface of the `Collection` interface, so it inherits all the operations of `Collection` and `Container`. Some of the set ADT operations are included in these super-interfaces (such as `contains()`), so they don't appear explicitly in the `Set` interface.

21.4 CONTIGUOUS IMPLEMENTATION OF SETS

The elements of a set can be stored in an array or an `ArrayList` but this approach is not very efficient. To see this, lets consider how the most important set operations—insertion, deletion, and membership check—would be implemented using an array. If the elements are not kept in order, then inserting them is very fast, but deleting them is slow (because a sequential search must be done to find the deleted element), and the membership check is slow (because it also requires a sequential search). If the elements are kept in order, then the membership check is fast (because binary search can be used), but insertion and deletion are very slow because on average half the elements must be moved to open or close a hole for the inserted or deleted element.

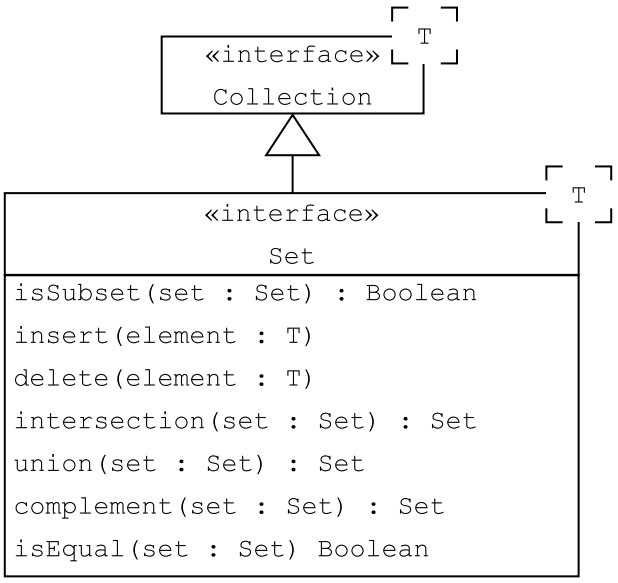


Figure 1: The `Set` Interface

There is one way to implement sets using contiguous storage that is very time efficient, though it may not be space efficient. A boolean array, called a *characteristic function*, can represent a set. The characteristic function is indexed by set elements so that the value of the characteristic function at index x is true if and only if x is in the set. Insertion, deletion, and the membership check can all be done in constant time. The problem, of course, is that each characteristic function array must have an element for every possible value that could be in the set, and these values must be able to index the array. If a set holds values from a small sub-range of an integral type, such as the ASCII characters, or integers from 0 to 50, then this technique is feasible (though it still may waste a lot of space). But for large sub-ranges or for non-integral set elements, this technique is no longer possible.

There is one more contiguous implementation technique that is very fast for sets: hashing. We will discuss using hashing to implement sets later on.

21.5 LINKED IMPLEMENTATION OF SETS

The problem with the contiguous implementation of sets is that insertion, deletion, and membership checking cannot all be done efficiently. The same holds true of linked lists. We have, however, encountered data structures guaranteed to provide fast insertion, deletion, and membership checking: balanced search trees. Recall that a balanced search tree can be searched in $O(\lg n)$ time, and elements can be inserted and deleted in $\Theta(\lg n)$ time.

An implementation of sets using search trees is called a *tree set*. Tree sets are very efficient implementations of sets provided care is taken to keep them balanced. Tree sets have the further advantage of allowing iteration over the elements in the set in sorted order, which can be very useful in some applications.

Based on what we have learned in the previous chapters, we can make tree sets from binary search trees, AVL trees, or 2-3 trees. Binary search trees are not guaranteed to stay balanced, so we won't use those. There is no particular reason to prefer AVL trees over 2-3 trees, or vice-versa, so either may be used.

21.6 SETS AND TREE SETS IN JAVA

Tree sets are implemented using search trees, which in our Java implementations require that stored values implement the `Comparable` interface. Ideally, values of arbitrary types should be allowed in any set, including values of types like `int`, which is a primitive type that does not implement any interfaces. However, it is quite easy to convert values into types that do implement the `Comparable` interface to get around this limitation. For example, in Java `int` values are easily convertible (usually automatically) into `Integer` values, and the `Integer` class implements the comparable interface.

It is very easy to implement tree sets using balanced search trees because all the hard work is done by the balanced search tree class. For example, consider the Java code in Figure 2 below that shows part of a `TreeSet` class that uses AVL trees to implement sets.

```
public class TreeSet<T extends Comparable<T>> implements Set<T> {
 private AVLTree<T> tree;

  public TreeSet() { tree = new AVLTree<>(); }

  public boolean contains(T item) { return tree.contains(item); }

 ...

  public boolean isSubset(Set<T> set) {
   for (T element : this)
    if (!set.contains(element)) return false;
   return true;
  }

 ...

} // TreeSet
```

Figure 2: Using Search Trees to Implement Tree Sets

The `TreeSet` class has a private `AVLTree` field for storing the elements of the set. Most tree set methods simply delegate to the AVL tree, as illustrated by the `contains()` method. Some methods need to do a little work of their own. For example, the `isSubset()` method iterates through the elements in the host set and checks to ensure that they are all in the argument set. Behind the scenes, an inorder traversal of the host set's search tree is done to iterate over the elements, and the search tree `contains()` operation is used to quickly determine whether a value is in the argument set.

21.7 SUMMARY AND CONCLUSION

Sets are useful ADTs that can be implemented efficiently using balanced search trees. Although sets in principle hold values of any type, using search trees to implement them forces us to store only `Comparable` values in tree sets. We will consider another data structure for realizing sets that imposes a different restriction in a later chapter.

21.8 REVIEW QUESTIONS

1. Which set operation appears in the set ADT but does not appear explicitly in the `Set` interface? Why is it not present?
2. What is a characteristic function?
3. Why is an array or an `ArrayList` not a good data structure for implementing the set ADT?
4. Why is a `LinkedList` not a good data structure to implement the set ADT?
5. Why is a balanced search tree a good data structure for implementing the set ADT?

21.9 EXERCISES

1. What other operation do you think might be useful to add to the set of *T* ADT?
2. Is an iterator required for sets? How does this compare with the situation for lists?
3. Which search tree iteration order would be best as the iteration order for sets? Why did you choose as you did?
4. Using the code in Figure 2 as a staring point, write the `Set` interface. Start writing a `TreeSet` class by implementing all the `Collection` interface methods. You may use a 2-3 tree if you prefer.
5. Continue the implementation begun in the previous exercise by writing the `insert()` and `delete()` methods.
6. Continue the implementation from exercise 4 by implementing the remaining methods from the `Set` interface in the `TreeSet` class.

21.10 REVIEW QUESTION ANSWERS

1. The membership operation in the set ADT does not appear explicitly in the `Set` interface because the `contains()` operation from the `Collection` interface already provides this functionality.
2. A characteristic function is a function created for a particular set that takes a value as an argument and returns true just in case that value is a member of the set. For example, the characteristic function $f(x)$ for the set $\{a, b, c\}$ would have the value true for $f(a)$, $f(b)$, and $f(c)$, but false for any other argument.
3. If an array or an `ArrayList` is used to implement the set ADT, then either the insertion, deletion, or set membership operations will be slow. If the array or `ArrayList` is kept in order, then the set membership operation will be fast (because binary search can be used), but insertion and deletion will be slow because elements will have to be moved to keep the array in order. If the array or `ArrayList` is not kept in order, then insertions will be fast, but deletions and set membership operations will require sequential searches, which are slow.
4. If a `LinkedList` is used to implement the set ADT, then deletion and membership testing will be slow because a linear search will be required. If the elements of the list are kept in order to speed up searching (which only helps a little because a sequential search must still be used), then insertion is made slower.
5. A balanced search tree is a good data structure for implementing the set ADT because a balanced search tree allows insertion, deletion, and membership testing to all be done quickly, in $O(\lg n)$ time.

22 MAPS

22.1 INTRODUCTION

A very useful collection is one that is a hybrid of lists and sets, called a *map*, *table*, *dictionary*, or *associative array*. A map (as we will call it), is a collection whose elements (which we will refer to as *values*) are unordered, like a set, but whose values are accessible via a *key*, akin to the way that list elements are accessible by indices.

Map: An unordered collection whose values are accessible using a key.

Another way to think of a map is as a function that maps keys to values (hence the name), like a function in mathematics. As such, a map is a set of ordered pairs of keys and values such that each key is paired with a single value (though a value may be paired with several keys).

22.2 THE MAP ADT

Maps store values of arbitrary type with keys of arbitrary type, so the ADT is *map of* (K, T), where K is the type of the keys and T is the type of the values in the map. The carrier set of this type is the set of all ordered pairs whose first element is of type K and whose second element is of type T. The carrier set thus includes the empty map, the maps with one ordered pair of values of types K and T, the maps with two ordered pairs of values of types K and T, and so forth.

The essential implicit-receiver operations of maps, in addition to those common to all collections, are those for inserting, deleting, searching and retrieving keys and values.

isEmpty()—Return true if and only if the map is empty.

size()—Return the number of pairs in the map.

hasKey(*k*)—Return true if and only if the map contains an ordered pair whose first element is *k*.

hasValue(*v*)—Return true if and only if the map contains an ordered pair whose second element is *v*.

insert(*k*, *v*)—Add the ordered pair <*k*, *v*> to the map. If the map already contains an ordered pair whose first element is *k*, then this ordered pair is replaced with the new one.

delete(*k*)—Remove from the map the ordered pair whose first element is *k*. If the map does not contain such an ordered pair, do nothing.

get(*k*)—Return the second element in the ordered pair whose first element is *k*. This operation's precondition is that the map contains an ordered pair whose first value is *k*.

equal(*m*)—Return true iff the receiver map and map *m* have the same key-value pairs.

There is considerable similarity between these operations and the operations for lists and sets. For example, the *delete*() operation for lists takes an index and removes the element at the designated index, while the map operation takes a key and removes the key-value pair matching the key. On the other hand, when the list index is out of range, there is a precondition violation, while if the key is not present in the map, the map is unchanged. This latter behavior is the same as what happens with sets when the set *delete*() operation is called with an argument that is not in the set.

22.3 THE MAP INTERFACE

The diagram below in Figure 1 shows the `Map` interface, which is a sub-interface of `Collection`. It includes all the operations of the map of (*K*, *T*) ADT, except *hasValue*`()`, whose functionality is accomplished by `contains()` from the `Collection` interface.

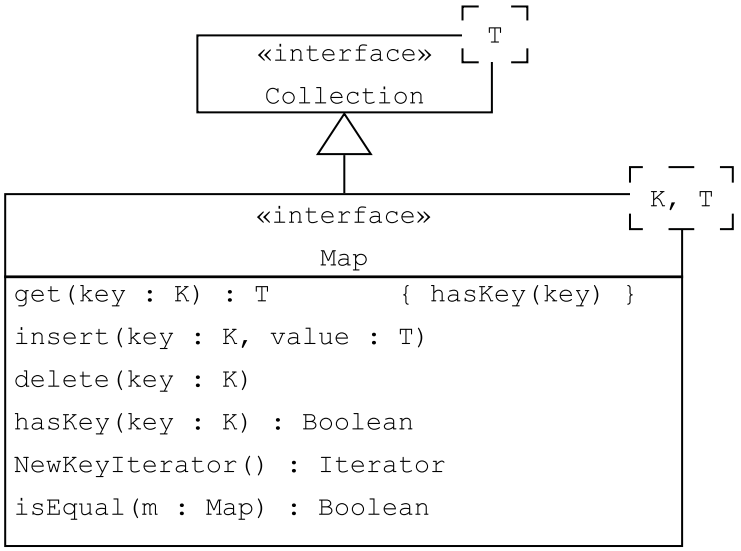


Figure 1: The `Map` Interface

As a `Collection`, a `Map` has an associated `Iterator` (returned by the `Collection iterator()` operation) that iterates over the values in the `Map`. The `NewKeyIterator()` operation returns an `Iterator` that iterates over the keys in the `Map`.

The `Map` interface lacks an equivalent of the *hasValue*() operation from the ADT. This method would generally be very inefficient, so its use is discouraged by not including it in the interface. Furthermore, its effect can be achieved using an iterator, so it is not essential.

22.4 CONTIGUOUS IMPLEMENTATION OF THE MAP ADT

As with sets, using an array or `ArrayList` to store the ordered pairs of a map is not very efficient because only one of the three main operations of insertion, deletion, and search can be done quickly. Also as with sets, a characteristic function can be used if the key set is a small integral type (or a small sub-range of an integral type), but this situation is rare. Finally, as with sets, hashing provides a very efficient contiguous implementation of maps that we will discuss later on.

22.5 LINKED IMPLEMENTATION OF THE MAP ADT

As with sets, using linked lists to store map elements is not much better than using arrays. But again as with sets, balanced search trees can store map elements and provide fast insertion, deletion, and search operations on keys. Furthermore, using search trees to store map elements allows the elements in the map to be traversed in sorted key order, which is sometimes very useful. A *tree map* is thus an excellent implementation of the `Map` interface.

The trick to using search trees to store map elements is to create a class implementing the `Comparable` interface holding key-value pairs that defines the `Comparable` operations in terms of comparison of keys. These key-value pair classes (or references to them) are stored in search tree nodes. Dummy values with the correct key fields but arbitrary value fields can then be used to search the tree and retrieve key-value pairs, or delete key-value pairs. To illustrate, consider the Java code in Figure 2.

```
public class TreeMap<K extends Comparable<K>, T>
             implements Map<K, T> {
 private AVLTree<Pair> tree;
 ...
 public T get(K key) {
  Pair dummy = new Pair(key,null);
  Pair result = tree.get(dummy);
  return result == null ? null : result.value;
 }
 ...
 private class Pair implements Comparable<Pair> {
  K key; // the key in the pair
  T value; // the value in the pair

  public Pair(K k, T v) { key = k; value = v; }

  public int compareTo(Pair other) {
   return key.compareTo(other.key);
  }
 }
 ...
}
```

Figure 2: Using a Pair Class to Implement a Tree Map in Java

This code shows the somewhat tricky declarations needed for the generic parameters of the `TreeMap` and `Pair` classes, as well as the contents of the `Pair` class and how to use it in the `get()` method, which is fairly representative of other methods in this class. This class uses an AVL tree to store the pairs, but some other sort of balanced search tree could be used instead. Note how the `get()` method creates a `dummy Pair` with the desired key, then uses it to search the AVL tree for the `Pair` instance stored there. The result is extracted from the retrieved `Pair` instance (if it exists).

22.6 SUMMARY AND CONCLUSION

Maps are extremely important collections because they allow values to be stored using keys, a very common need in programming. The map of (*K*, *T*) ADT specifies the essential features of the type, and the `Map` interface captures these features and places maps in the `Container` hierarchy. Contiguous implementations are not well suited for maps (except hash tables, which we discuss in the next chapter). Balanced search trees, however, provide very efficient implementations, so a tree map is a good realization of the `Map` interface. The crucial technique for making this work is to store key-value pairs in a class that implements the `Comparable` interface in such a way that keys are compared. These pairs can then be stored in the balanced search tree by their keys.

22.7 REVIEW QUESTIONS

1. Why is a map called a map?
2. The `Collection` interface has a generic parameter `T`, and the `Map` interface has generic parameters `K` and `T`. What is the relationship between them?
3. Why is an array or an `ArrayList` not a good data structure for implementing the map ADT?
4. Why is a `LinkedList` not a good data structure to implement the map ADT?
5. Why is a balanced search tree a good data structure for implementing the map ADT?

22.8 EXERCISES

1. Make a function mapping the states California, Virginia, Michigan, Florida, and Oregon to their capitals. If you wanted to store this function in a map ADT, which would be the keys and which the values?
2. Is an iterator required for maps? How does this compare with the situation for lists?
3. Draw a UML class diagram showing the entire `Container` interface hierarchy including all the collections we have considered up to this point. You need not include operations in your diagram. Do include the `Iterator` interface.
4. 35Assuming that all `Collection` and `Map` operations are implemented in a `TreeMap` using a balanced search tree, make a table showing the time complexity of every operation in a `TreeMap`.
5. Using the Java code in Figure 2 as a starting point, write code for the `Map` interface and the `TreeMap` class, including the class constructor.

6. Continue the Java implementation you began in the last exercise by writing code to implement all the operations the `Map` interface inherits from `Container` and `Collection` except those for iteration.
7. Continue the Java implementation from the last exercise by writing code for the iterators in a `TreeMap`.
8. Complete the Java implementation from the last exercise by writing all remaining `Map` interface methods.

22.9 REVIEW QUESTION ANSWERS

1. A map associates keys and values such that each key is associated with at most one value. This is the definition of a function from keys to values. Functions are also called *maps*, and we take the name of the collection from this meaning of the word.
2. The `Collection` interface generic parameter `T` is the same as the `Map` interface generic parameters `T`: the elements of a `Collection` are also the values of a `Map`. But `Maps` have an additional data item—the key—whose type is `K`.
3. An array or an `ArrayList` is not a good data structure for implementing the map ADT because the key-value pairs would have to be stored in the array or `ArrayList` in order or not in order. If they are stored in order, then finding a key-value pair by its key is fast (because we can use binary search), but adding and removing pairs is slow. If pairs are not stored in order, then they can be inserted quickly by appending them at the end of the collection, but searching for them or finding them when they need to be removed are slow operations because they must use sequential search.
4. A `LinkedList` is not a good data structure to implement the map ADT because although key-value pairs can be inserted quickly into a `LinkedList`, searching for pairs or finding them when they need to be removed are slow operations because the `LinkedList` must be traversed node by node.
5. A balanced search tree is a good data structure for implementing the map ADT because adding key-value pairs, searching for them by key, and removing them by key, are all done very quickly (in $O(\lg n)$ time). Furthermore, if the nodes in the tree are traversed in order, then the key-value pairs are accessed in key-order.

23 HASHING

23.1 INTRODUCTION

In an ideal world, retrieving a value from a map would be done instantly by just examining the value's key. That is the goal of hashing, which uses a hash function to transform a key into an array index, thereby providing instantaneous access to the value stored in an array holding the key-value pairs in the map. This array is called a hash table.

> **Hash function**: A function that transforms a key into a value in the range of indices of a hash table.
>
> **Hash table**: An array holding key-value pairs whose locations are determined by a hash function applied to keys.

Of course, there are a few details to work out.

23.2 THE HASHING PROBLEM

If a set of key-value pairs is small and we can allocate an array big enough to hold them all, we can always find a hash function that transforms keys to unique locations in a hash table. For example, in some old programming languages, identifiers consisted of an upper-case letter possibly followed by a digit. Suppose these are our keys. There are 286 of them, and it is not too hard to come up with a function that maps each key of this form to a unique value in the range 0..285. But usually the set of keys is too big to make a table to hold all the possibilities. For example, older versions of FORTRAN had identifiers that started with an upper-case letter, followed by up to five additional upper-case letters or digits. The number of such identifiers is 1,617,038,306, which is clearly too big for a hash table if we were to use these as keys.

A smaller table holding keys with a large range of values will inevitably require that the function transform several keys to the same table location. When two or more keys are mapped to the same table location by a hash function we have a collision. Mechanisms for dealing with them are called *collision resolution schemes*.

> **Collision**: The event that occurs when two or more keys are transformed to the same hash table location.

How serious is the collision problem? After all, if we have a fairly large table and a hash function that spreads keys out evenly across the table, collisions may be rare. In fact, however, collisions occur surprisingly often. To see why, lets consider the *birthday problem*, a famous problem from probability theory: what is the chance that a least two people in a group of k people have the same birthday? This turns out to be p =1-(365!/k!/365^k). Table 1 below lists some values for this expression. Surprisingly, in a group of only 23 people there is better than an even chance that two of them have the same birthday!

If we imagine that a hash table has 365 locations, and that these probabilities are the likelihoods that a hash function transforms two values to the same location (a collision), then we can see that we are almost certain to have a collision when the table holds 100 values, and very likely to have a collision with only about 40 values in the table. Forty is only about 11% of 365, so we see that collisions are very likely indeed. Collision resolution schemes are thus an essential part of making hashing work in practice.

k	p
5	0.027
10	0.117
15	0.253
20	0.411
22	0.476
23	0.507
25	0.569
30	0.706
40	0.891
50	0.970
60	0.994
100	0.9999997

Table 1: Probabilities in the Birthday Problem

An implementation of hashing thus requires two things:

- A hash function for transforming keys to hash table locations, ideally one that makes collisions less likely.
- A collision resolution scheme to deal with the collisions that are bound to occur.

We discuss each of these in turn.

23.3 HASH FUNCTIONS

A hash function must transform a key into an integer in the range 0 to t-1, where t is the size of the hash table. A hash function should distribute the keys in the table as uniformly as possible to minimize collisions. Although many hash functions have been proposed and investigated, the best hash functions use the division method, which for numeric keys is the following.

$$hash(k) = k \% t$$

This function is simple, fast, and spreads out keys uniformly in the table. It works best when t is a prime number not close to a power of two. For this reason, hash table sizes should always be chosen to be a prime number not close to a power of two.

For non-numeric keys, there is usually a fairly simple way to convert the value to a number and then use the division method on it. For example, the following Java function illustrates a way to hash a string using the division method.

```
int hash(String s, int tableSize) {
 int result = 0;
 for (int i = 0; i < s.length(); i++)
  result = (result * 151 + s.charAt(i)) % tableSize;
 return result;
}
```

Figure 1: A Java Hash Function for Strings

Making hash functions is not too onerous. Good rules of thumb are to use prime numbers whenever a constant is needed, and to test the function on a representative set of keys to ensure that it spreads them out evenly across the hash table.

23.4 COLLISION RESOLUTION SCHEMES

There are two main kinds of collision resolution schemes, with many variations: chaining and open addressing. In each scheme, an important value to consider is the load factor, $\lambda = n/t$, where n is the number of elements in the hash table and t is the table size.

Chaining

In chaining (or separate chaining) records whose keys collide are formed into a linked list or chain whose head is in the hash table array. Figure 2 below shows a hash table with collisions resolved using chaining. For simplicity, only the keys are listed and not the values that go along with them (or, if you like, the key and the value are the same).

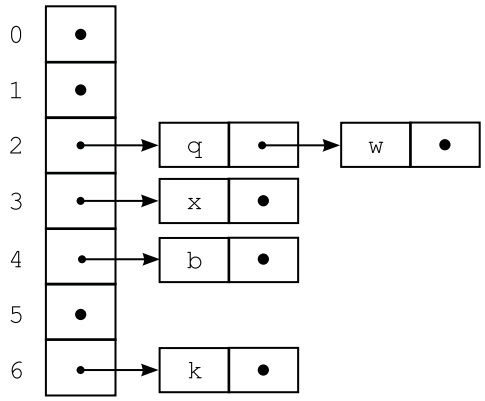


Figure 2: Hash Table with Chaining to Resolve Collisions

In this example, the table has seven locations. Two keys, *q* and *w*, collide at location two and they are placed in a linked list whose head is at location two. The keys *x*, *b*, and *k* are hashed to locations three, four, and six, respectively. This example uses an array of list heads, but an array of list nodes could have been used as well, with some special value in the data field to indicate when a node is unused.

The average chain length is λ. If the chain is unordered, on average successful searches require about $1+ \lambda/2$ comparisons and unsuccessful searches about λ comparisons. If the chain is ordered, both successful and unsuccessful searches take about $1+ \lambda/2$ comparisons, but insertions take longer. In the worse case, which occurs when all keys map to a single table location and the search key is at the end of the linked list or not in the list, searches require n comparisons. But this case is extremely unlikely.

More complex linked structures (like binary search trees), don't generally improve performance much, particularly if λ is kept fairly small. As a rule of thumb, λ should be kept less than about 10. But performance only degrades gradually as the number of items in the table grows, so hash tables that use chaining to resolve collisions can perform well on wide ranges of values of n.

Open Addressing

In the open addressing collision resolution scheme, records with colliding keys are stored at other free locations in the hash table found by *probing* the table for open locations. Different kinds of open addressing schemes use different *probe sequences*. In all cases, however, the table can only hold as many items as it has locations, that is, $n \leq t$ and λ cannot exceed one; this constraint is not present for chaining.

Open addressing has been modelled in theoretical studies using random probe sequences. In these studies, the number of probes for unsuccessful searches is about $\frac{1}{2}(1+ 1/(1-\lambda))$ and for unsuccessful searches it is about $\frac{1}{2}(1+ 1/(1-\lambda)2)$. These values shoot up as λ approaches one. For example, with a load factor of 0.95, the expected number of comparisons for a successful search is 10.5, and for an unsuccessful search it is 200.5. Real open addressing schemes do not do even as well as this, so load factors must generally be kept below about 0.75.

Another point to understand about open addressing is that when a collision occurs, the algorithm proceeds through the probe sequence until either (a) the desired key is found, (b) an open location is found, or (c) the entire table is traversed. But this only works when a marker is left in slots where an element was deleted to indicate that the location may not have been empty before, and so the algorithm should proceed with the probe sequence. In a highly dynamic table there will be many markers and few empty slots, so the algorithm will need to follow long probe sequences, especially for unsuccessful searches, even when the load factor is low.

> **Linear probing** is using a probe sequence that begins with the hash table index and increments it by a constant value modulo the table size. If the table size and increment are relatively prime, every slot in the table will appear in the probe sequence. Linear probing performance degrades sharply when load factors exceed 0.8. Linear probing is also subject to *primary clustering*, which occurs when clumps of filled locations accumulate around a location where a collision first occurs. Primary clustering increases the chances of collisions and greatly degrades performance.
>
> **Double hashing** works by generating an increment for the probe sequence by applying a second hash function to the key. The second hash function should generate values quite different from the first so that two keys that collide will be mapped to different values by the second hash function, making the probe sequences for the two keys different. Double hashing eliminates primary clustering. The second hash function must always generate a number that is relatively prime to the table size. This is easy if the table size is a prime number. Double hashing works so well that its performances approximates that of a truly random probe sequence. It is thus the method of choice for generating probe sequences.

Figure 3 below shows an example of open addressing with double hashing. As before, the example only uses keys for simplicity, not key-value pairs. The main hash function is $f(x) = x \% 7$, and the hash function used to generate a constant for the probe sequence is $g(x) = (x \% 5)+1$. The values 8, 12, 9, 6, 25, and 22 are hashed into the table.

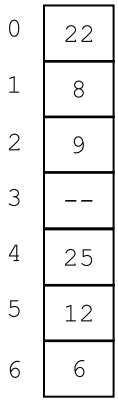


Figure 3: Hash Table with Open Addressing and Double Hashing to Resolve Collisions

The first five keys do not collide. But 22 % 7 is 1, so 22 collides with 8. The probe constant for double hashing is (22 % 5)+1 = 3. We add 3 to location 1, where the collision occurs, to obtain location 4. But 25 is already at this location, so we add 3 again to obtain location 0 (we wrap around to the start of the array using the table size: (4+3) % 7 = 0). Location 0 is not occupied, so that is where 22 is placed.

Note that some sort of special value must be placed in the unoccupied locations—in this example we used a double dash. A different value must be used when a value is removed from the table to indicate that the location is free, but that it was not before, so that searches must continue past this value when it is encountered during a probe sequence.

We have noted that when using open addressing to resolve collisions, performance degrades considerably as the load factor approaches one. This in effect means that hashing mechanisms that use open addressing must have a way to expand the table to lower the load factor and improve performance. A new, larger table can be created and filled by traversing the old table and inserting all records into the new table. Note that this involves hashing every key again because the hash function will generally use the table size, which has now changed. Consequently, this is a very expensive operation.

Some table expansion schemes work incrementally by keeping the old table around and making all insertions in the new table, all deletions from the old table, and perhaps moving records gradually from the old table to the new in the course of doing other operations. Eventually the old table becomes empty and can be discarded.

23.5 SUMMARY AND CONCLUSION

Hashing uses a hash function to transform a key into a hash table location, thus providing almost instantaneous access to values though their keys. Unfortunately, it is inevitable that more than one key will be hashed to each table location, causing a collision and requiring some way to store more than one value associated with a single table location.

The two main approaches to collision resolution are chaining and open addressing. Chaining uses linked lists of key-value pairs that start in hash table locations. Open addressing uses probe sequences to look through the table for an open spot to store a key-value pair and then later to find it again. Chaining is very robust and has good performance for a wide range of load factors, but it requires extra space for the links in the list nodes. Open addressing uses space efficiently, but its performance degrades quickly as the load factor approaches one; expanding the table is very expensive.

No matter how hashing is implemented, however, average performance for insertions, deletions, and searches is $\Theta(1)$. Worst case performance is $O(n)$ for chaining collision resolution, but this only occurs in the very unlikely event that the keys are hashed to only a few table locations. Worst case performance for open addressing is a function of the load factor that gets very large when λ is near one, but if λ is kept below about 0.8, $W(n)$ is less than 10.

23.6 REVIEW QUESTIONS

1. What happens when two or more keys are mapped to the same location in a hash table?
2. If a hash table has 365 locations and 50 records are placed in the table at random, what is the probability that there will be at least one collision?
3. What is a good size for hash tables when a hash function using the division method is used?
4. What is a load factor? Under what conditions can a load actor exceed one?
5. What is a probe sequence? Which is better: linear probing or double hashing?

23.7 EXERCISES

1. Why does the example of open addressing and double hashing in the text use the hash function $g(x) = (x \% 5)+1$ rather than $g(x) = x \% 5$ to generate probe sequences?
2. Suppose a hash table has 11 locations and the simple division method hash function $f(x) = x \% 11$ is used to map keys into the table. Compute the locations where the following keys would be stored: 0, 12, 42, 18, 6, 22, 8, 105, 97. Do any of these keys collide? What is the load factor of this table if all the keys are placed into it?
3. Suppose a hash table has 11 locations, keys are placed in the table using the hash function $f(x) = x \% 11$, and linear chaining is used to resolve collisions. Draw a picture similar to Figure 2 of the result of storing the following keys in the table: 0, 12, 42, 18, 6, 22, 8, 105, 97.
4. Modify with the diagram you made for Exercise 3 to show what happens when 18 and 42 are removed from the hash table.
5. Suppose a hash table has 11 locations, keys are mapped into the table using the hash function $f(x) = x \% 11$, and collisions are resolved using open addressing and linear probing with a constant of three to generate the probe sequence. Draw a picture of the result of storing the following keys in the table: 0, 12, 42, 18, 6, 22, 8, 105, 97.
6. List the probe sequence (the table indices) used to search for 97 in the diagram you drew for Exercise 5. List the probe sequence used when searching for 75.
7. Starting with the diagram you made for Exercise 5, show the result of removing 18 from the table. List the probe sequence used to search for 97. How do you guarantee that 97 is found even though 18 is no longer encountered in the probe sequence?
8. Suppose a hash table has 11 locations, keys are mapped into the table using the hash function $f(x) = x \% 11$, and collisions are resolved using double hashing with the hash function $g(x) = (x \% 3)+1$ to generate the probe sequence. Draw a picture of the result of storing the following keys in the table: 0, 12, 42, 18, 6, 22, 8, 105, 97.
9. List the probe sequences used to search for 97 and 75 using the diagram you drew for Exercise 8. In what way are these sequences different from the probe sequences generated in Exercise 6?

23.8 REVIEW QUESTION ANSWERS

1. When two or more keys are mapped to the same location in a hash table, they are said to *collide*, and some action must be taken, called *collision resolution*, so that records containing colliding keys can be stored in the table.
2. If 50 values are added at random to a hash table with 365 locations, the probability that there will be at least one collision is 0.97, according to the Table 1.
3. A good size for hash tables when a division method hash function is used is a prime number not close to a power of two.
4. The load factor of a hash table is the ratio of the number of key-value pairs in the table to the table size. In open addressing, the load factor cannot exceed one, but with chaining, because in effect more than one key-value pair can be stored in each location, the load factor can exceed one.
5. A probe sequence is a list of table locations checked when elements are stored or retrieved from a hash table that resolves collisions with open addressing. Linear probing is subject to primary clustering, which decreases performance, but double hashing as been shown to be as good as choosing increments for probe sequences at random.

24 HASHED COLLECTIONS

24.1 INTRODUCTION

As we have seen, hashing provides very fast (effectively $\Theta(1)$) algorithms for inserting, deleting, and searching collections of key-value pairs. It is therefore an excellent way to implement maps. It also provides a very efficient way to implement sets.

24.2 HASH TABLE CLASS

A hash table is designed to store key-value pairs by the hash value of the key. A hash table class must therefore be generic in the type of both keys and values. Furthermore, hash tables must be have operations sufficient to implement maps. Figure 1 shows a UML diagram for a `HashTable` class meeting these requirements.

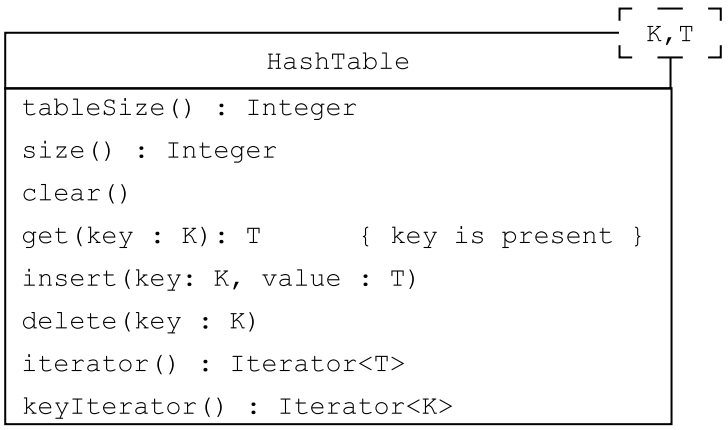


Figure 1: A `HashTable` Class

The type `K` must have values that can be hashed and compared for identity. There are no restrictions on the value type `T`. The `HashTable` class must allow key-value pairs to be inserted, deleted, modified, searched, and iterated over in various ways. The `insert()` method is used not only to add new key-value pairs to the hash table but to modify them as well: if a key is already in the table, then calling `insert()` with that key should replace the existing value with the new value passed to the method. The `iterator()` method enumerates the values in the table and the `keyIterator()` method enumerates the keys. Neither keys nor values are enumerated in order, of course. The `get()` method has as its precondition that the key is present in the table. The `tableSize()` method returns the number of slots in the table, and the `size()` method returns the number of pairs currently in the table.

Constructors for a `HashTable` class usually provide a default size (a small prime number like 13), or accept a parameter for the table size. In the latter case, the constructor should adjust the parameter to make sure that it is not too small and that it is a prime number (suggesting that the smallest hash table size is two).

Note that this `HashTable` model does not distinguish between collision resolution strategies; this is an implementation detail hidden by this interface. When open addressing is used, the `insert()` method should monitor the load factor and expand the table if it gets close to one. Expanding a hash table requires that a new, larger table be created, and that all the current key-value pairs in the old table be inserted in the new table, so this is an expensive operation. If chaining is used, the load factor can be arbitrarily large, but of course performance will degrade when it gets too big, so a sophisticated implementation should also monitor the load factor when insertions occur and expand the table if its gets too big.

Finally, hash tables, like trees, are not containers that clients are expected to use; rather, they are implementation data structures used to implement containers in the `Container` hierarchy.

In Java, a `HashTable` class should be generic in types `K` and `T`, with `K` extending `Comparable<K>` so that equality comparisons can be made between keys. The `Object` class has a `hashCode()` method that can be used to hash keys. This method may return

any `int` value, so its result will have to be modified to use as a hash table index. The hash table itself will have to hold (references to) instances of a class holding key-value pairs. This class will be generic in `K` and `T`, and Java prohibits making arrays of such generic classes. Consequently the hash table array must be an array of `Objects` whose elements must be cast to the key-value pair class.

24.3 HASHMAPS

The `HashTable` class is of course used to implement maps, resulting in a collection called a *hash map*. Hash maps, provided their backing hash tables do not become too full, provide constant time performance for all operations except the `contains()` collection method (which takes linear time). This beats search trees, so hash maps are generally faster than tree maps.

A `HashMap` class implements the `Map` interface and holds a hash table in a private field. The hash table stores the key-value pairs in the map, and most `Map` operations are delegated directly to the hash table. A few `Map` operations (like `contains()` and `isEqual()`), require a few lines of code for iterating through the hash table and making some comparisons.

24.4 HASHSETS

Hash tables are a convenient implementation data structure because they can be used to implement hashed collections of various kinds, not just maps. For example, suppose we wish to use hashing to implement sets. If we use a hash table to do this, then the key is the set element we wish to store, but what is the value in the key-value pair? There is none, so we can store the key in the value field as well, or simply leave the value field blank (by storing null there, for example). This wastes space, but provides the constant-time performance of hash tables. Hash tables are thus a good way to implement hash sets.

A *hash set* implements the `Set` interface and holds a hash table in a private attribute. As with hash maps, the basic `Collection` operations are easily realized using `HashTable` operations. To make this efficient, all operations should use keys. For example, the `contains() Collection` method should use the `HashTable get()` method to search for a set element as a key (using hashing) rather than iterating through the table looking at values.

Hash set union may be implemented by copying the host hash set (and its hash table) and iterating over the argument set, adding all its values. Set complement is done by making a new result set and iterating over the host set, adding to the result all elements not in

the argument set. Finally, intersection is done by iterating over one set and adding to the result hash set only those values that the other contains. All these operations are linear in the size of the sets operated on.

24.5 SUMMARY AND CONCLUSION

Hashing is an efficient way to implement sets as well as maps. A hash table class can provide operations that make set and map implementations fast using either open addressing or chaining to resolve collisions. In either case, it may be wise to monitor the load factor and expand the table if the load factor gets too high.

24.6 REVIEW QUESTIONS

1. What sorts of collision resolution techniques can be used in a `HashTable` class?
2. What happens when the `insert()` method is called in a `HashTable` class that uses open addressing with a load factor of one?
3. Sets do not have keys and values so how can hash tables be used to store sets?
4. Why is the `contains()` method slower than the `get()` method in a `HashMap` class?

24.7 EXERCISES

1. Write the `HashTable` class in Java. Your class may use open addressing or chaining to resolve collisions. Note that you will have to write a private inner class to hold key-value pairs.
2. Write a `HashMap` class in Java using the `HashTable` you wrote in the previous exercise.
3. Write a `HashSet` class in Java using the `HashTable` you wrote in the first exercise. You may use set elements or `null` as the value in key-value pairs.
4. Hash tables waste space when implementing hash sets. To avoid this, write a `HashTablet` class in Java that uses hashing to store single values of a type `T`. These values extend `Comparable<T>` so you can compare them. Then use a `HashTablet` to implement a `HashSet` class.
5. Use the `HashTablet` class you wrote in the last exercise to implement a `HashMap` class. In this case, you will have to make a `Pair` class to store key-value pairs that extends `Comparable<Pair>` in such a way that its `compareTo()` method compares keys. Also, its `hashCode()` method must hash keys. Then `Pair` class instances can be stored in the `HashTablet` to complete the implementation of `HashMap`.

24.8 REVIEW QUESTION ANSWERS

1. Any sort of collision resolution techniques can be used in a `HashTable` class.
2. Suppose that a `HashTable` class implemented with open addressing has a load factor of one. In this case, we know that all the key-value pairs are stored in the table array (because open addressing is used to resolve collisions), and that all the array locations are in use (because the load factor is one). If the `insert()` method is called, then we are attempting to add yet another key-value pair to the table. But there is not room for it. This method must either fail (perhaps raising an exception), or it must create a new, larger table into which it inserts all the old entries before it can insert the new key-value pair.
3. Sets only store single elements, not key-value pairs, but hash tables store key-value pairs. We can use a hash table to store single elements in one of two ways. We can treat set elements as element-element pairs (in other words, make both the key and the value the same element), or we can store the element as the key and null or some arbitrary entity as the value. Then all set operations can be done in terms of the keys in the hash table, which will be fast (because the keys are hashed) achieving good performance.
4. In a `HashMap` key-value pairs are stored by hashing the key. To find a particular value, one must iterate through the key-value pairs and examine each value in the pair, which is slow. This is what must be done to implement the `contains()` method, which asks about a particular value. On the other hand, the `get()` method uses a key to find its associated value. This is fast because the key is hashed to find the key-value pair, and then the value is extracted from the pair.

25 GRAPHS

25.1 INTRODUCTION

One of the most important modeling tools in computing is the *graph*, which is informally understood as a collection of points connected by lines. Graphs are used to model networks, processes, relationships between entities, and so on—almost every picture we draw in computer science is a graph of one sort or another. In this chapter we present the graph abstract data type and consider two data structures for representing graphs. In the next chapter we study a few graph algorithms.

25.2 DIRECTED AND UNDIRECTED GRAPHS

We defined a graph in the course of discussing trees in Chapter 16.

> **Graph**: A collection of *vertices* (or *nodes*) and *edges* connecting the nodes. An edge may be thought of as a pair of vertices. Formally, a graph is an ordered pair <*V*, *E*> where *V* is a set of vertices and *E* is a set of pairs of elements of *V*.
>
> **Undirected graph**: A graph in which the edges are sets of two vertices. In this case the edges have no direction and are represented by line segments when we draw pictures of them.
>
> **Directed graph** or **digraph**: A graph in which the edges are ordered pairs of vertices. In this case the edges have direction and are represented by arrows in pictures of them.

To illustrate these definitions, consider the images of graphs in Figure 1.

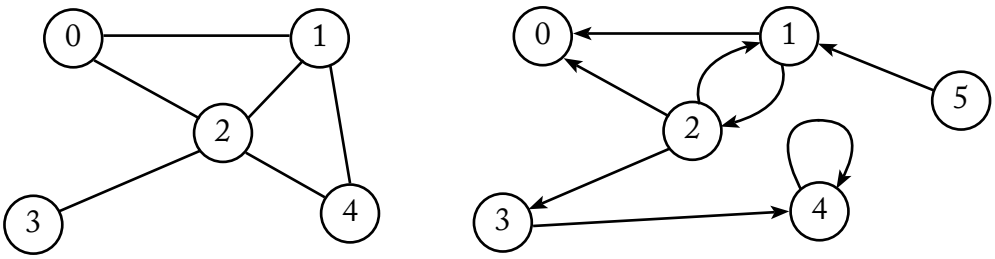


Figure 1: Two Graphs

In these images the vertices are identified by circled numbers. In general we may use any symbol to identify vertices, but as a rule we will use an initial set of natural numbers (that is, any set {0, 1, 2, ... *n*}, where $n \geq 0$). The graph on the left is an undirected graph so its edges have no arrows. In its set representation, this graph is

<{0, 1, 2, 3, 4}, {{0,1}, {0,2}, {1,2}, {1,4}, {2,4}, {2,3}}>.

The graph on the right is a directed graph, so it has arrows on its edges. Note that this allows edges from a node to itself (such as the edge from 4 to itself), and two distinct edges between a pair of vertices (such as the two edges connecting vertices 1 and 2); neither of these can occur in an undirected graph. The set representation of the right-hand graph is

<{0, 1, 2, 3, 4}, {<1,0>, <1,2>, <2,0>, <2,1>, <2,3>, <3,4>, <4,4>, <5,1>}>.

Note that the edge set in the left-hand graph is a set of sets while the edge set in the right-hand graph is a set of ordered pairs.

Although a graph is really an ordered pair of sets, representing graphs this way is awkward and hard to read, as the examples above illustrate. Consequently we will almost always represent graphs as pictures.

Both undirected and directed graphs are very important and widely applicable in computer science, but we will focus for the remainder of our discussion on undirected graphs. From now on we will refer to undirected graphs as simply *graphs*.

25.3 BASIC TERMINOLOGY

There are several additional terms that must be learned to talk about graphs.

Adjacency: Vertices v_1 and v_2 in a graph $G=<V, E>$ such that $\{v_1, v_2\} \in E$.

In Figure 2 below, vertices 0 and 1 and vertices 7 and 4 are adjacent, for example, but vertices 0 and 4 and vertices 1 and 3 are not adjacent.

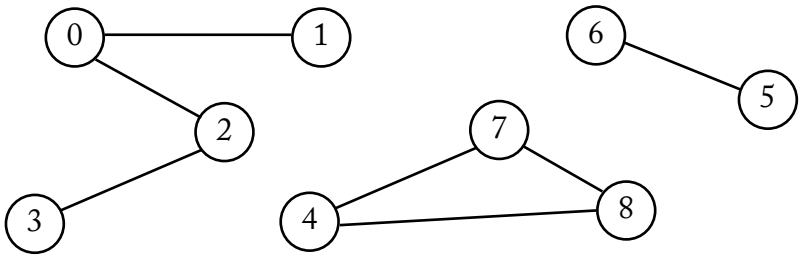


Figure 2: A Graph

Path: A sequence of vertices $p = <v_1, v_2, ..., v_n>$ in a graph where $n \geq 2$ and every pair of vertices v_i and v_{i+1} in p are adjacent.

Path length: The number of edges in a path.

In Figure 2, the paths <0, 2, 3> and <4, 7, 8> both have length two.

Cycle: A path $<v_1, v_2, ..., v_n>$ in which $v_1 = v_n$ but no other vertices are repeated.

Cyclic graph: A graph with a cycle of length at least three.

In Figure 2, the paths <0, 1, 0> and <4, 7, 8, 4> are cycles. Because the second has length three, the graph in Figure 2 is cyclic.

Sub-graph: A graph $H=<W, F>$ is a sub-graph of graph $G=<V, E>$ if $W \subseteq V$ and $F \subseteq E$.

Connected vertices: Two vertices with a path between them.

Connected graph: A graph with a path between every pair of vertices. A graph that is not connected consists of a set of connected components that are sub-graphs of the graph.

Figure 2 shows a single graph with three connected components. Each of these components is a sub-graph of the whole graph.

Acyclic graph: A graph that is not cyclic.

The graph in Figure 2 is cyclic, as noted before, but the sub-graph consisting of the vertices 0, 1, 2, and 3 and the edges that connect them, and the sub-graph consisting of vertices 5 and 6 and the edge that connects them, are acyclic graphs.

Tree: An acyclic connected graph.

Forest: A set of trees with no vertices in common.

Spanning tree: Any sub-graph of a connected graph *G* that is a tree and contains every vertex of *G*.

Figure 3 shows a graph with a spanning tree marked with double lines.

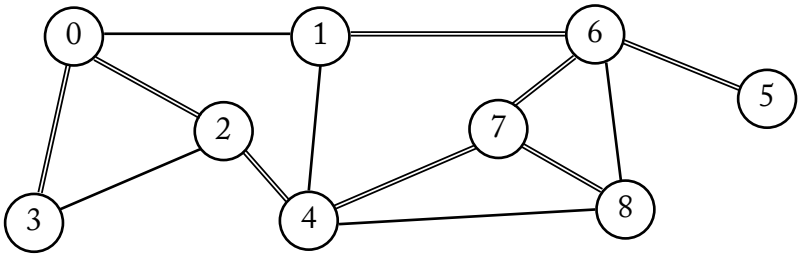


Figure 3: A Graph and A Spanning Tree

25.4 THE GRAPH ADT

A graph is a mathematical entity consisting of an ordered pair of sets. Vertices can be anything; for example, they could be numbers. Hence we can specify the carrier set of the graph ADT as the set of all sets that meet the definition of a graph stated above, with initial sets of natural numbers acting as vertices. The implicit-receiver method set of the graph ADT consists of a few basic operations for constructing and querying graphs. We could include operations for adding and deleting vertices and deleting edges, as well as several other query operations, but they are not necessary for the applications we will consider.

newGraph(*n*)—Return a graph with *n* vertices and no edges. The precondition of this operation is that $n \geq 0$.

edges()—Return the number of edges in the graph.

vertices()—Return the number of vertices in the graph.

addEdge(*v*,*w*)—Augment the graph with an edge connecting *v* and *w*. The precondition is that *v* and *w* are distinct vertices in *g*.

isEdge(*v*,*w*)—Return true if and only if there is an edge between vertices *v* and *w* in the graph.

25.5 THE GRAPH INTERFACE

A graph is not a collection so the `Graph` interface is not a sub-interface of any other interface in our container hierarchy. It is very convenient to be able to iterate over the vertices adjacent to a given vertex so the `Graph` interface depends on the `Iterator` interface (though this is not shown in the UML diagram). The `Graph` interface appears in Figure 4.

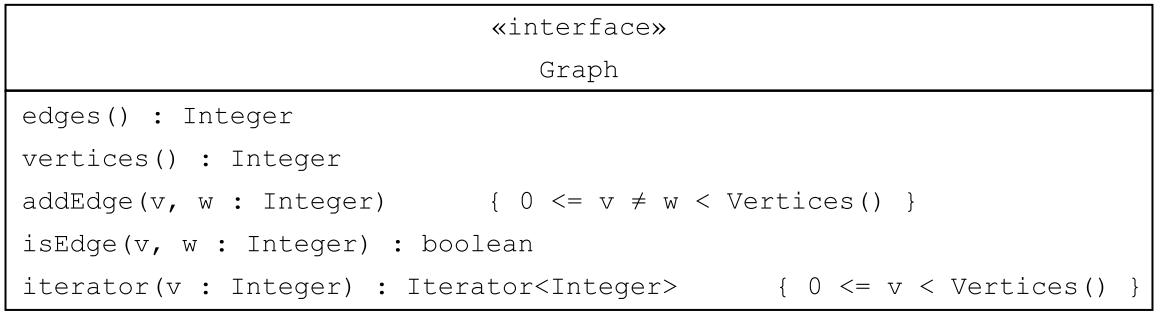


Figure 4: The `Graph` Interface

Most of these functions work just as one would expect from the graph ADT. The `addEdge()` and `iterator()` operations do nothing if their preconditions are violated. There is no need for an iterator over the vertices in the graph because we know that they are (represented by) the integers between 0 and `vertices()`-1.

25.6 CONTIGUOUS IMPLEMENTATION OF THE GRAPH ADT

A contiguous implementation of the graph ADT represents a graph using an array. An initial set of natural numbers already represents vertices, so the only thing left to represent is the set of edges. An adjacency matrix m is an $n \times n$ Boolean matrix that represents a graph with n vertices by storing true at location $m[v,w]$ if and only if there is an edge between v and w. Notice that this means that every edge is represented twice in the matrix.

This scheme is very simple and very fast: adding an edge to a graph or detecting whether an edge exists between two vertices are both $\Theta(1)$ operations, and iterating over the vertices adjacent to a vertex v takes time proportional to the number of vertices in the graph. Unfortunately, this speed comes at great cost because the matrix requires n^2 storage locations. Most graphs are *sparse*, meaning they have far fewer than n^2 edges, so often most of this storage space is wasted. Even if a graph is *dense* (the opposite of sparse), space can be saved by storing the edges that are *not* in the graph rather than in the graph. In either case, the adjacency matrix representation does make efficient use of space.

25.7 LINKED IMPLEMENTATION OF THE GRAPH ADT

A linked implementation of the graph ADT represents graphs by using space only for the edges in the graph. An adjacency list is a linked list of vertices adjacent to a given vertex. An array of adjacency lists holds all the edges in a graph. The diagram in Figure 5 shows the adjacency lists representation of the graph in Figure 2. Note that the array holds list headers and the adjacency lists are singly-linked.

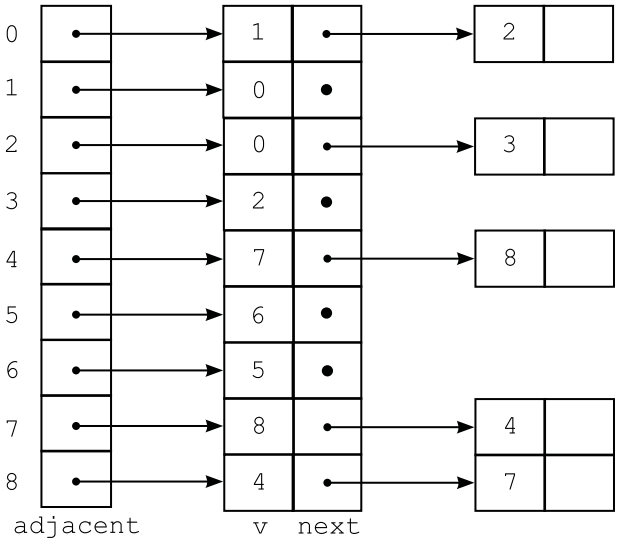


Figure 5: An Adjacency Lists Representation of a Graph

Notice that the adjacency lists representation, like the adjacency matrix representation, records every edge $\{v, w\}$ twice: once on the list for the edges adjacent to v and once on the list for the edges adjacent to w.

The adjacency lists data structure uses space proportional to the sum of the number of vertices and edges (the array has space for every vertex, and there are twice as many nodes in the linked lists as there are edges in the graph). This is typically much less than the space required for the adjacency matrix representation. Also, adding an edge takes $\Theta(1)$ time and determining whether there is an edge between two vertices takes time proportional to the number of edges emanating from (one of) the vertices; this is $O(e)$ (where e is the number of edges in the graph) in the worst case, but typically it is much less. Thus the adjacency lists representation uses relatively little space but is still quite efficient.

In implementing adjacency lists it is convenient to use a the linked list from our `Container` hierarchy to realize each adjacency list. The `adjacent` array then holds linked lists implementing the `List` interface rather than references to nodes.

25.8 SUMMARY AND CONCLUSION

Graphs are an important modeling tool in computing. The graph ADT provides a few operations for building and querying a graph and this is carried over into the `Graph` interface. The adjacency matrix technique for representing graphs makes operations efficient but uses a great deal of space. The adjacency lists approach is nearly as fast but uses much less space. As a rule, unless a graph is dense or it has a small number of nodes (say, less than a few hundred), the adjacency lists representation is preferable.

25.9 REVIEW QUESTIONS

1. List several elements of the graph ADT carrier set.
2. What is the result of applying the graph ADT operation *addEdge*() twice with the same vertices? In other words, if *g* is a graph and *v* and *w* are vertices, what is the result of *g.addEdge*(*v*,*w*); *g.addEdge(v*,*w*)?
3. How could you iterate over every vertex in a graph?
4. Why is every edge in a graph represented twice in both the adjacency matrix and adjacency lists representations?

25.10 EXERCISES

1. Can a vertex be adjacent to itself in an undirected graph?
2. If a graph has no edges, can it have any paths? If a graph has edges, does it have a longest path?
3. How many vertices must there be in the smallest cycle in an undirected graph? How many in the smallest cycle in a directed graph?
4. In Chapter 16 a tree was defined as a graph with a distinguished vertex *r*, called the *root*, such that there is exactly one simple path between each vertex in the tree and *r*. Show that this definition is equivalent to the definition stated in this chapter.
5. Can a graph have more than one spanning tree? Explain.
6. Use the operations of the graph ADT to construct the graph in Figure 2.
7. Represent the graph in Figure 2 using an adjacency matrix.
8. Represent the graph in Figure 2 using adjacency lists. Draw a picture like the one in Figure 5.
9. Write the `Graph` interface in Java.
10. Create an `ArrayGraph` class in Java to represent graphs using an adjacency matrix. The `ArrayGraph` constructor must take `n`, the number of nodes in the graph, as a parameter.
11. Create a `LinkedGraph` class in Java to represent graphs using adjacency lists. The `LinkedGraph` constructor must take `n`, the number of nodes in the graph, as a parameter. The `LinkedGraph` class should have a field that is an array of `Objects` that will hold `LinkedList<Integer>` objects.

25.11 REVIEW QUESTION ANSWERS

1. The graph ADT carrier set includes the empty graph, which as no vertices and no edges: <∅, ∅>. The next largest graph has a single vertex and no edges: <{0}, ∅>. The next largest graphs have two vertices and either no edges or one edge: <{0,1}, ∅>, and <{0,1}, {{0,1}}>. There are several graphs with three vertices: <{0, 1, 2}, ∅>, <{0, 1, 2}, {{0, 1}}>, <{0, 1, 2}, {{0, 2}}>, <{0, 1, 2}, {{1, 2}}>, <{0, 1, 2}, {{0, 1}, {0, 2}}>, <{0, 1, 2}, {{0, 1}, {1, 2}}>, <{0, 1, 2}, {{0, 2}, {1, 2}}>, <{0, 1, 2}, {{0, 1}, {0, 2}, {1, 2}}>.
2. If g is a graph and v and w are vertices, the result of $g.addEdge(v,w)$ is g with the edge $\{v, w\}$ added to its edge set. The result of calling $g.addEdge(v,w)$ again will be g with the edge $\{v, w\}$ added to its edge set. But this edge was already in the edge set of g, so the second call does not change g. Hence applying *addEdge*() to a graph with the same vertices several times does nothing after the first call.
3. Iterating over every vertex in a graph g simply requires looping over every integer from 0 to $vertices(g)-1$.

4. Every edge in a graph is represented twice in both the adjacency matrix and adjacency lists representations because in each case the representation "indexes" edges by their vertices. Because each edge has two vertices, each appears twice. Note that we could easily come up with representations in which an edge appears only once. For example, we could only put the smaller of the two vertices of an edge into the adjacency array in both the adjacency matrix and adjacency lists representations. This would save about half the space, but it would make iterating over the vertices adjacent to a given vertex (which turns out to be a very important operation) very slow: we would have to search the entire representation to find all the vertices adjacent to a given vertex. So in this case space is traded for time, and we use more space to get much faster performance in an essential operation.

26 GRAPH ALGORITHMS

26.1 INTRODUCTION

There are many important algorithms on graphs. In this chapter we examine a few of the most fundamental and widely used. In particular, we consider graph search algorithms and several algorithms based on them.

26.2 SEARCHING GRAPHS

Many problems involving graphs require a systematic traversal of the graph along its edges—this is termed a graph search. We have already seen graph search algorithms in the form of tree traversals, but these only apply to trees, not graphs in general.

As an example of a problem illustrating the need to search a general graph, suppose that we wish to solve a maze. A maze can be represented by a graph as follows: map the entrance, exit, and intersections in the maze to graph vertices and map the paths between the entrance,

the exit, and intersections in the maze to graph edges. Moving through a maze from the entry to the exit corresponds to the problem of searching a graph representing the maze to find a path from the entry vertex to the exit vertex.

There are two main approaches to graph searching.

> **Depth-first search**: A search that begins by visiting vertex *v*, and then recursively searches the unvisited vertices adjacent to *v*.
>
> **Breadth-first search**: A search that begins by visiting vertex *v*, then visits the vertices adjacent to *v*, then visits the vertices adjacent to each of *v*'s adjacent vertices, and so on.

Vertices in a depth-first search are visited in an order that follows paths leading away from the starting vertex until the paths can be followed no further; the algorithm then backs up and follows other paths. Vertices deep in the graph (relative to the starting vertex) are visited sooner than shallow vertices. In contrast, during a breadth-first search the vertices closest to the starting vertex are visited first, then those a bit further away, and so on. Some problems are solved with one kind of search, some with the other, and in some cases it does not matter which kind of search is used.

26.3. DEPTH-FIRST SEARCH

Given its recursive characterization, it is not surprising that depth-first search is easily implemented using recursion. Consider the Java code in Figure 1. The `DFS()` method is called with a graph to be searched, an origin vertex to start from, and an instance of an `EdgeVisitor` class containing a `visit()` method that is applied to edges. It may seem peculiar to use and edge visitor to process vertices, but it turns out that there are several algorithms that we can implement if our depth-first and breadth-first algorithms use edge visitors rather than vertex visitors. If we need to just visit a vertex rather than an edge, we will use an edge visitor method that processes only the second vertex in the edge, thus achieving the same thing while preserving flexibility in our visitor methods.

The `DFS()` method sets up a boolean array to keep track of the vertices that have been visited and then calls a private `dfs()` method to actually do the work. The recursive `dfs()` method takes the graph, the `EdgeVisitor` instance, the boolean array, and a source vertex `v` as its arguments. It iterates though the vertices adjacent to `v`, and for each adjacent vertex `w` not yet visited, it applies the visit method to `w`, marks `w` as visited, and calls itself on `w`.

```
public static void DFS(Graph g, int v, EdgeVisitor edgeVisitor) {
 boolean[] isVisited = new boolean[g.vertices()];
 edgeVisitor.visit(g, -1, v);
 isVisited[v] = tarue;
 dfs(v, g, edgeVisitor, isVisited);
}

private static void dfs(Graph g, int v, EdgeVisitor
          edgeVisitor, boolean[] isVisited) {
 Iterator<Integer> iter = g.iterator(v);
 while (iter.hasNext()) {
  int w = iter.next();
  if (isVisited[w]) continue;
  edgeVisitor.visit(g, v, w);
  isVisited[w] = true;
  dfs(g, w, edgeVisitor, isVisited);
 }
}
```

Figure 1: Recursive Depth-First Search

Just as with a binary tree traversal, we can also write a depth-first search using a stack rather than recursion. Figure 2 shows this algorithm.

The stack-based algorithm uses the same general strategy as the recursive algorithm: it keeps track of unvisited vertices and only processes those that have not yet been visited. Where the recursive algorithm makes recursive calls on unvisited adjacent vertices after a vertex has been visited, this algorithm places the unvisited adjacent vertices in a stack. However, since we are using an edge visitor, we must have a stack of edges rather than just vertices.

Both versions of the depth-first search algorithm visit each vertex at most once and process each edge leaving every visited vertex exactly once. Each edge has two vertices so each edge is processed at most twice. Hence depth-first search runs in $O(v+e)$ time in the worst case, where v is the number of vertices and e is the number of edges in the graph.

26.4 BREADTH-FIRST SEARCH

The stack-based version of depth-first search places vertices adjacent to the current vertex in a stack, then it processes the vertex on top of the stack, placing its adjacent vertices on the stack, and so forth. The effect of this strategy is that vertices adjacent to the initial vertex are at the bottom of the stack and therefore get processed after vertices further away. If we use a queue instead of a stack, then adjacent vertices will be processed first, then those adjacent to those next, and so on. In short, we can make a breadth-first search algorithm

from the stack-based depth-first search algorithm by simply replacing the stack with a queue. Such an algorithm is shown in Figure 3.

```
public static void stackDFS(Graph g, int v0, EdgeVisitor visitor)
{
 boolean[] isVisited = new boolean[g.vertices()];
 Stack<Edge> stack = new LinkedStack<Edge>();
  stack.push(new Edge(-1, v0));
 while (!stack.isEmpty()) {
  Edge edge = stack.pop();
  if (isVisited[edge.w]) continue;
  visitor.visit(g, edge.v, edge.w);
  isVisited[w] = true;
  Iterator<Integer> iter = g.iterator(edge.w);
  while (iter.hasNext()) {
   int x = iter.next();
   if (!isVisited[x]) stack.push(new Edge(edge.w,x));
  }
 }
}
```

Figure 3: Queue-Based Breadth-First Search

This algorithm visits each vertex exactly once and follows each edge at most twice, so its performance is in $O(e+v)$, just like its stack-based peer. The only difference is that is visits vertices in a different order.

26.5 PATHS IN A GRAPH

We can use graph searching algorithms to determine graph properties. Recall that two vertices are connected if and only if there is a path between them. We can determine this using either depth-first or breadth-first search by starting a search at one vertex and checking whether the search ever reaches the other vertex. Java code for such a function appears in Figure 4.

```
public static boolean isPath(Graph g, int v, int w) {
 if (v < 0 || g.vertices() <= v) return false;
 if (w < 0 || g.vertices() <= w) return false;
 MarkTarget target = new MarkTarget(w);
 DFS(g, v, target);
 return target.isReached();
}
```

Figure 4: Determining Whether Two Vertices are Connected

This function first checks whether the argument vertices are even in the graph—if one is not, then there is no path between them. It then uses depth-first search from the first vertex with an edge visitor class (`MarkTarget`) that watches for a target vertex and remembers whether it has been reached.

If two vertices are connected, there may be more than one path between them, and often it is useful to know the shortest path (the one with the fewest edges). The function in Figure 5 finds the shortest path between two vertices.

The shortest-path function first makes sure that there is a path between the argument vertices and returns `null` if there is not. The core of the algorithm is an `EdgeVisitor` that, for some source vertex `w`, constructs an array that contains, for each vertex except `w`, the vertex next on the shortest path back to `w`. This array, called `toEdge` in Figure 5, must be constructed using a breadth-first search. A vertex `x` adjacent to `w` has its `toEdge` entry set to `w` because the shortest path from `x` back to `w` is obviously the edge to `w`. The vertices adjacent to `x` have their `toEdge` entries set to `x` because the shortest path back to `w` goes first to `x` and then to `w` (a shorter path could only exist if these edges were adjacent to `w`). The `toEdge` entries are constructed in like fashion for vertices further from `w`. Clearly, a breadth-first search is needed to visit vertices in the order necessary to make this work.

Once the `toEdge` array is constructed, it contains the next vertices on the shortest path from any vertex to `w`. It is then an easy task to traverse it from the source vertex `v` to the target vertex `w` to generate a shortest path between the two.

```
public static List<Integer> shortestPath(Graph g, int v, int w) {
 if (!isPath(g,v,w)) return null;
 int [] toEdge = new int[g.vertices()];
 PathVisitor visitor = new PathVisitor(toEdge);
 BFS(g,w,visitor);
 List<Integer> result = new ArrayList<Integer>();
 int x = v;
 while (x != w) {
  result.insert(result.size(),x);
  x = toEdge[x];
 }
 result.insert(result.size(),x);
 return result;
}

private static class PathVisitor implements EdgeVisitor {
 private int[] toEdge;

 public PathVisitor(int[] toEdge) { this.toEdge = toEdge; }

 public void visit(Graph g, int v, int w) { toEdge[w] = v; }

}
```

Figure 5: Finding the Shortest Path Between Connected Vertices

26.6 CONNECTED GRAPHS AND SPANNING TREES

Besides determining whether two vertices are connected, we can also determine whether an entire graph is connected, and in much the same way. In this case, we can do a depth-first or breadth-first search from any vertex in the graph and check whether every other vertex is visited. The graph is connected if and only if every other vertex is visited by either a breadth-first or depth-first search starting at a source vertex. We leave the code for this algorithm as an exercise.

Recall that a spanning tree is a sub-graph of a graph g that is a tree and contains every vertex of g. If we visit every vertex in a connected graph g from a source vertex and add the edge along which each vertex is visited to a new graph h, then h will be a spanning

tree for *g* when we are done. This is the strategy used to construct a spanning tree in the Java code in Figure 6.

```
public static Graph spanningTree(Graph g) {
 if (!isConnected(g)) return null;
 Graph result = new LinkedGraph(g.vertices());
 EdgeVisitor visitor = new TreeBuilder(result);
 DFS(g, 0, visitor);
 return result;
}

private static class TreeBuilder implements EdgeVisitor {
 private Graph tree;
  public TreeBuilder(Graph g) { tree = g; }
   public void visit(Graph g, int v, int w) {
  tree.addEdge(v, w);
 }
}
```

Figure 6: Making a Spanning Tree for a Connected Graph

This function returns `null` when the argument graph is not connected.

26.7 SUMMARY AND CONCLUSION

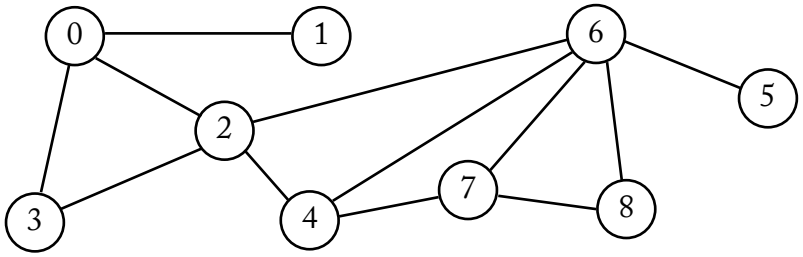
Depth-first and breadth-first search are two ways to visit the vertices of a graph by following its edges in an organized way. Both algorithms work in time proportional to the number of edges and vertices in the graph. Depth-first and breadth-first search are the basis for many algorithms that process graphs in various ways, including determining whether two vertices are connected, determining whether a graph is connected, finding the shortest path between two vertices, and finding a spanning tree for a connected graph.

26.8 REVIEW QUESTIONS

1. What is the relationship between graph search and graph traversal?
2. How are graph searching algorithms related to stacks and queues?
3. Under what circumstances is one graph searching algorithm preferable to the other?
4. Does it matter whether we use depth-first or breadth-first search to find the shortest path between two vertices?
5. Does it matter whether we use depth-first or breadth-first search to generate a spanning tree for a connected graph?

26.9 EXERCISES

Use the graph below to do the exercises.



1. List the order in which the vertices are visited in a depth-first search from vertex 0 in the graph above. Assume that adjacent vertices are visited in order from smallest to largest.
2. List the order in which the vertices are visited in a breadth-first search from vertex 0 in the graph above. Assume that adjacent vertices are visited in order from smallest to largest.
3. Trace the execution of the shortest-path function in generating a shortest path between vertices 0 and 8 in the graph above. What is the path?

4. Trace the execution of the spanning tree function in generating a spanning tree for the graph above. What is the spanning tree?
5. Write a Java function `boolean isConnected(Graph g)` to determine whether graph `g` is connected.
6. Write a Java function `int componentCount(Graph g)` to count the number of connected components in graph `g`.
7. The *degree* of a vertex is the number of edges connected to it. Write a Java function `int maxDegree(Graph g)` to find the maximum degree of the vertices in graph `g`.
8. Our graph algorithms are written to apply visit functions to graphs using either depth-first or breadth-first search. But we could also write the functions directly using graph search without a visit function. Rewrite the shortest path and spanning tree algorithms directly so that they do not call the depth-first or breadth-first search functions.
9. The following problems can be solved by modeling the problem with a graph and then applying graph functions to the model. Explain how to solve these problems using graphs and graph algorithms.
   a) A path through a maze.
   b) The smallest set of phone calls that must be made to transmit a message amongst a group of friends.
   c) Determining whether it is possible to get from point A to point B using only main highways.
   d) Finding the degree of separation between two people in a community (the *degree of separation* between two people who know each other is one; the degree of separation between two people who do not know each other but have a mutual friend is two, and so on).

26.10 REVIEW QUESTION ANSWERS

1. Graph search is another name for graph traversal.
2. The depth-first search algorithm uses a stack (or recursion), while the breadth-first search algorithm uses a queue. Otherwise, they are virtually identical.
3. The depth-first and breadth-first search algorithms run in the same amount of time and use the same amount of space, so there is no basis for preferring one over the other in general. However, some algorithms require one or the other to work properly.
4. The shortest path algorithm is an example of an algorithm that requires one of the search algorithms to work properly; in particular, the shortest path algorithm requires a breadth-first search to work properly.
5. The spanning tree algorithm is an example of an algorithm where is does not matter which search algorithm is used: either depth-first or breadth-first search can be used to generate a spanning tree.

27 GLOSSARY

abstract data type (ADT)—a set of values (the **carrier set**), and operations on those values (the **method set**).

abstract method—a method with a signature but no body.

acyclic graph—a graph with no cycles.

adjacency—vertices v_1 and v_2 are **adjacent** in an undirected graph $G=<V, E>$ if $\{v_1, v_2\}$ is a member of E.

adjacency list—a linked list of vertices adjacent to a given vertex.

adjacency matrix—an $n \times n$ Boolean matrix m that represents a graph with n vertices by storing true at location $m[v,w]$ if and only if there is an edge between v and w.

ADT assertion—a statement that must be true of the carrier set values or method set operations of the type.

algorithm—a finite sequence of steps for accomplishing some computational task. An algorithm must have steps that are simple and definite enough to be done by a computer and terminate after finitely many steps.

algorithm analysis—the process of determining, as precisely as possible, how much of various resources (such as time and memory) an algorithm consumes when it executes.

amortized analysis—determining the average time of operations in the worst case by considering the time needed to perform an arbitrary sequence of n operations, and then dividing this total by n.

array—a fixed length, ordered collection of values of the same type stored in contiguous memory locations; the collection may be ordered in several dimensions.

assertion—a statement that must be true at a designated point in a program.

associative array—see **map**.

average case complexity $A(n)$—the average number of basic operations performed by an algorithm for all inputs of size n given assumptions about the characteristics of inputs of size n.

AVL tree—a binary search tree in which the balance factor at each node is -1, 0, or 1; the empty tree is an AVL tree.

balance factor of a tree node—the height of the left sub-tree of the node minus the height of the right sub-tree of the node, with the height of the empty tree defined as -1.

balanced tree—a tree such that for every node, the height of its sub-trees differ by at most some constant value.

basic operation—an operation fundamental to an algorithm used to measure the amount of work done by the algorithm.

best case complexity $B(n)$—the minimum number of basic operations performed by an algorithm for any input of size n.

binary search tree—a binary tree whose every vertex is such that the value at the vertex is greater than the values in its left sub-tree and less than the values in its right sub-tree.

binary tree—an ordered tree whose vertices have at most two children. The children are distinguished as the *left child* and the *right child*. The sub-tree whose root is the left (right) child of a vertex is the *left (right) sub-tree* of that vertex.

breadth-first search—a search that begins by visiting vertex *v*, then visits the vertices adjacent to *v*, then visits the vertices adjacent to each of *v*'s adjacent vertices, and so on.

built-in type—a data type that is provided directly by a programming language.

carrier set—in an abstract data type or a data type, the set of values of the abstract data type.

chaining—a hashing collision resolution scheme in which key-value pairs whose keys collide are formed into a linked list or chain whose head is in the hash table.

circular doubly linked list—a doubly linked list in which the last node in the list holds a references to the first element rather than null, and the first node in the list holds a reference to the last element rather than null.

circular singly linked list—a singly linked list in which the last node in the list holds a references to the first element rather than null.

class invariant—an assertion that must be true of any class instance before and after calls of its exported operation.

collection—a traversable container.

collision—the event that occurs when two or more keys are transformed to the same hash table location.

complete binary tree—a tree whose every level is full except possibly the last, and only the right-most leaves at the bottom level are missing.

complexity *C*(*n*)—the number of basic operations performed by an algorithm as a function of the size of its input *n* when this value is the same for any input of size *n*.

computational complexity—the time (and perhaps the space) requirements of an algorithm.

connected components—in a graph that is not connected, the sub-graphs that are connected.

connected graph—a graph in which every pair of vertices is connected.

connected vertices—two vertices in a graph with a path between them.

container—an entity that holds finitely many other entities.

cursor—a value marking a location in a data structure.

cycle—a path $<v_1, v_2, ..., v_n>$ in a graph in which $v_1 = v_n$ but no other vertices are repeated.

cyclic graph—a graph with a cycle of length at least three.

data structure—an arrangement of data in memory locations to represent values of the carrier set of an abstract data type.

data type—an implementation of an abstract data type on a computer.

depth-first search—a search that begins by visiting vertex v, and then recursively searches the unvisited vertices adjacent to v.

dequeue—a dispenser whose elements can be accessed, inserted, or removed only at its ends.

dictionary—see **map**.

digraph—a directed graph.

directed graph—a graph in which the edges are ordered pairs of vertices; the edges have direction and are represented by arrows.

dispenser—a non-traversable container.

divide and conquer algorithm—an algorithm that solves a large problem by dividing it into parts, solving the resulting smaller problems, and then combining these solutions into a solution to the original problem.

double hashing—in open addressing collision resolution, a probe sequence using an increment generated by applying a second hash function to the key.

doubly linked list—a linked structure whose nodes each have two reference or pointer fields used to form the nodes into a sequence. Each node but the first has a predecessor link field containing a reference or pointer to the previous node in the list, and each node but the last has a successor link containing a reference or pointer to the next node in the list.

dynamic array—an array whose size is established at run-time and can change during execution.

element—a value stored in an array or a traversable container (a collection).

enumerable—see **traversable**.

every-case complexity—the number of basic operations performed by an algorithm as a function of the size of its input n when this value is the same for any input of size n.

external iteration—collection iteration under the control of a separate entity, called an iterator.

factory method—a non-constructor operation that returns a new instance of a class.

fixed array—an array whose size is established when space for the array is allocated and cannot change thereafter.

forest—a set of trees with no vertices in common.

full binary tree—a binary tree whose every level is full except possibly the last.

graph—a collection of *vertices* (or *nodes*) and *edges* connecting the vertices. An edge may be thought of as a pair of vertices. Formally, a graph is an ordered pair $<V,E>$ where V is a set of vertices and E is a set of pairs of elements of V.

graph search—a systematic traversal of a graph along its edges.

hash function—a function that transforms a key into a value in the range of indices of a hash table.

hash table—an array holding key-value pairs whose locations are determined by a hash function applied to keys.

heap—a complete binary tree whose every vertex has the heap-order property.

heap-order property—a vertex has the heap order property when the value stored at the vertex is at least as large as the values stored at its descendents.

height of a tree—the maximum level of any node in the tree.

implicit-receiver method set—a method set containing functions with an implicit receiver argument.

infix expression—an expression in which the operators appear between their operands.

inorder traversal—a tree traversal in which, at every node, and for every key at a node from left to right, the child to the left of a node key node is visited first, followed by the node key, followed by the child to the right of the node key.

internal iteration—collection iteration under the control of the collection.

iterable—see **traversable**.

iteration—the process of accessing each element of a collection in turn; the process of traversing a collection.

iterator—an entity that provides serial access to each member of an associated collection.

iterator pattern—an object-oriented software design pattern specifying the use of an external iterator.

level of a node—in a tree, the number of edges in the path from the root to the node.

linear probing—in open addressing collision resolution, a probe sequence that begins with the hash table index and increments it by a constant value modulo the table size.

linked (data) structure—a collection of nodes formed into a whole through its constituent node link fields.

linked tree—a linked structure whose nodes form a tree.

list—an ordered linear collection.

load factor—in hashing, the value $\lambda = n/t$, where n is the number of elements in the hash table and t is the table size.

loop invariant—an assertion that must be true at the start of every execution of the body of a loop.

map—an unordered collection of elements, which are values of an *element type*, accessible using a key, which is a value of a *key type*; also called a **table**, **dictionary**, or **associative array**.

method set—in an abstract data type, or a data type the set of operations of the abstract data type.

method signature—the name, number and types of parameters, and return type of a method.

node—an aggregate variable with data and link (reference or pointer) fields; in a graph, a vertex.

open addressing—a hashing collision resolution scheme in which key-value pairs with colliding keys are stored at other free locations in the hash table found by probing the table for open locations.

order—the function g is in the set $O(f)$, denoted $g \in O(f)$, if there exist some positive constant c and non-negative integer n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$; informally, when g is in $O(f)$ we say that g has the same order (of growth) as f.

ordered tree—a tree in which the order of the children of each node is specified.

path—in a graph, a sequence $p = \langle v_1, v_2, \ldots, v_n \rangle$, where $n \geq 2$, such that every pair of vertices v_i and v_{i+1} in p are adjacent.

path length—the number of edges in a path.

perfectly balanced tree—a tree all whose leaves are on the same level; that is, the path from the root to any leaf is always the height of the tree.

pointer—a type whose carrier set contains addresses of values of an associated base type.

post condition—an assertion that must be true at the completion of an operation.

postfix expression—an expression in which the operators appear after their operands.

postorder traversal—a tree traversal in which, at every node, the children of the node are visited in order from left to right, followed by the node.

precondition—an assertion that must be true at the initiation of an operation.

prefix expression—an expression in which the operators appear before their operands.

preorder traversal—a tree traversal in which, at every node, the node is visited first, followed by the children of the node in order from left to right.

priority queue—a queue whose elements each have a non-negative integer **priority** used to order the elements of the priority queue such that the highest priority elements are at the front and the lowest priority elements are at the back.

queue—a dispenser holding a sequence of elements that allows insertions only at one end, called the **back** or **rear**, and deletions and access to elements at the other end, called the **front**.

receiver—when an operation with a receiver is called, the receiver is an implicit argument of the operation and may also be an implicit result of the operation.

record—a finite collection of named values of arbitrary type called **fields** or **members**; a record is also called a **struct**.

recurrence—a recurrence relation plus one or more initial conditions that together recursively define a function.

recurrence relation—an equation that expresses the value of a function in terms of its value at another point.

recursive operation—an operation that either calls itself directly, or calls other operations that call it.

reference type—a type whose variables hold references to locations where data structures representing the values of the carrier set of the type are stored; compare to **value type**.

sentinel value—a special value placed in a data structure to mark a boundary.

separate chaining—see **chaining**.

sequential search—an algorithm that looks through a list from beginning to end for a key, stopping when it finds the key.

set—an unordered collection in which an element may appear at most once.

simple cycle—a cycle in a graph with no repeated edges or vertices (except the first and the last vertices).

simple path—a list of distinct vertices such that successive vertices are connected by edges.

simple type—a type in which the values of the carrier set are atomic, that is, they cannot be divided into parts.

singly linked list—a linked data structure whose nodes each have a single link field used to form the nodes into a sequence. Each link but the last contains a reference or pointer to the next node in the list; the link field of the last node contains null.

slice—a reference to a contiguous segment of an associated array.

software design pattern—a model proposed for imitation in solving a software design problem.

sorting algorithm—an algorithm that rearranges records in lists so that they follow some well-defined ordering relation on values of keys in each record.

spanning tree—any sub-graph of a connected graph G that is a tree and contains every vertex of G.

stack—a dispenser holding a sequence of elements that can be accessed, inserted, or removed at only one end, called the **top**.

static array—see **fixed array**.

string—a finite sequence of characters drawn from some alphabet.

struct—a record consisting of named fields of various types.

structured type—a type whose carrier set values are composed of some arrangement of atomic values.

sub-graph—a graph $H=<W, F>$ is a **sub-graph** of graph $G=<V, E>$ if $W \subseteq V$ and $F \subseteq E$.

table—see **map**.

tail recursive algorithm—an algorithm in which at most one recursive call is made as the last step of each execution of the algorithm's body.

traversable—a container is traversable iff all the elements it holds are accessible to clients.

tree—a graph with a distinguished vertex r, called the *root*, such that there is exactly one simple path between each vertex in the tree and r; alternatively, an acyclic connected graph.

two-three tree—a perfectly balanced tree whose every node is either a *2-node* with one value v and zero or two children, such that every value in its left sub-tree is less than v and every value in its right sub-tree is greater than v, or a *3-node* with two values v_1 and v_2 and zero or three children such that every value in its left-most sub-tree is less than v_1, every value in its middle sub-tree is greater than v_1 and less than v_2, and every value in its right-most sub-tree is greater than v_2

undirected graph—a graph in which the edges are sets of two vertices; the edges have no direction and are represented by line segments in pictures.

unreachable code assertion—an assertion that is placed at a point in a program where execution should not occur under any circumstances.

value type—a type whose variables hold data structures representing the values of the carrier set of the type; compare to **references type**.

variable-length encoding—a representation of a set of values that uses bit strings of different lengths to save space: more frequently occurring values are represented by shorter bit strings and less frequently occurring values by longer bit strings.

worst case complexity $W(n)$—the maximum number of basic operations performed by an algorithm for any input of size n.