Concise Notes on Data Structures and Algorithms

Ruby Edition

Christopher Fox
James Madison University

2011

Contents

1: Introduction

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 what we are really studying is 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.

Here are some examples of ADTs:

> *Boolean*—The carrier set of the Boolean ADT is the set { true, false }. The operations on these values are 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 operations on these values are 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 value 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). Operations on string values include 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: { λ, 0, 1, 00, 01, 10, 11, 000, ... }. Operations on bit strings include 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 set and its operations are mathematical entities, like numbers or geometric objects; 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 implementation 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 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 Ruby.

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 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 operations of the type really means finding algorithms that use the data structures for the carrier set to implement the operations of the ADT. 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 its operations.

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 computing. 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 an ADT 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.

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 references) to implement more complicated ADTs. Along the way we will construct a classification of complex ADTs that will serve as the basis for a class 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.

The Ruby 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 Ruby programming language.

Ruby is an interpreted, purely object-oriented language with many powerful features, such as garbage collection, dynamic arrays, hash tables, and rich string processing facilities. We use Ruby because it is a fairly popular, full-featured object-oriented language, but it can be learned well enough to write substantial programs fairly quickly. Thus we will be able to use a powerful language and still have time to concentrate on data structures and algorithms, which is what we are really interested in. Also, it is free.

Ruby is weakly typed, does not support design-by-contract, and has a somewhat frugal collection of features for object-oriented programming. Although this makes the language easier to learn and use, it also opens up many opportunities for errors. Careful attention to types, fully understanding preconditions for executing methods, and thoughtful use of class hierarchies are important for novice programmers, so we will pay close attention to these matters in our discussion of data structures and algorithms, and we will, when possible, incorporate this material into Ruby code. This often results in code that does not conform to the style prevalent in the Ruby community. However, programmers must understand and appreciate these matters so that they can handle data structures in more strongly typed languages such as Java, C++, or C#.

Review Questions

1. What would be 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?

Exercises

1. Describe the carrier sets and some operations 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 of the ADT might be implemented.

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 the 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 strings 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 fist 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

Simple and Structured Types

Virtually all programming languages have implementations of several ADTs built into them, thus providing the set of types provided by the language. We can distinguish two sorts 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 integers, Booleans, floating point numbers, enumerations, and characters. Some languages also provide strings as built-in types.
>
> **Structured types:** The values of the carrier set are not atomic, consisting instead of several atomic values arranged in some way. Common examples of structured types are arrays, records, classes, and sets. Some languages treat strings as structured types.

Note that both simple and structured types are implementations of ADTs, it 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 some Ruby simple and structured types to illustrate these ideas.

Types in Ruby

Ruby is a pure object-oriented language, meaning that all types in Ruby are classes, and every value in a Ruby program is an instance of a class. This has several consequences for the way values can be manipulated that may seem odd to programmers familiar with languages that have values that are not objects. For example, values in Ruby respond to method calls: The expressions `142.even?` and `"Hello".empty?` are perfectly legitimate (the first expression is `true` and the second is `false`).

The fact that all types in Ruby are classes has consequences for the way data structures are implemented as well, as we will see later on.

Ruby has many built-in types because it has many built-in classes. Here we only consider a few Ruby types to illustrate how they realize ADTs.

Symbol: A Simple Type in Ruby

Ruby has many simple types, including numeric classes such as `Integer`, `Fixnum`, `Bignum`, `Float`, `BigDecimal`, `Rational`, and `Complex`, textual classes such as `String`, `Symbol`, and `Regexp`, and many more. One unusual and interesting simple type is `Symbol`, which we consider in more detail to illustrate how a type in a programming language realizes an ADT.

Ruby has a `String` class whose instances are mutable sequences of Unicode characters. `Symbol` class instances are character sequences that are not mutable, and consequently the `Symbol` class has far fewer operations than the `String` class. Ruby in effect has

this is the symbol that we like and respect :symbol!

implementations of two String ADTs—we consider the simpler one, calling it the *Symbol* ADT for purposes of this discussion.

The carrier set of the Symbol ADT is the set of all finite sequences of characters over the Unicode characters set (Unicode is a standard character set of over 109,000 characters from 93 scripts). Hence this carrier set includes the string of zero characters (the empty string), all strings of one character, all strings of two characters, and so forth. This carrier set is infinite.

The operations of the Symbol ADT are the following.

a==*b*—returns true if and only if symbols *a* and *b* are identical.

a<=b—returns true if and only if either symbols *a* and *b* are identical, or symbol *a* precedes symbol *b* in Unicode collating sequence order.

a<*b*—returns true if and only if symbol *a* precedes symbol *b* in Unicode collating sequence order.

empty?(*a*)—returns true if and only if symbol *a* is the empty symbol.

a=~*b*—returns the index of the first character of the first portion of symbol *a* that matches the regular expression *b*. If there is not match, the result is undefined.

caseCompare(*a*,*b*)—compares symbols *a* and *b*, ignoring case, and returns -1 if *a*<*b*, 0 if *a*==*b*, and 1 otherwise.

length(*a*)—returns the number of characters in symbol *a.*

capitalize(*a*)—returns the symbol generated from *a* by making its first character uppercase and making its remaining characters lowercase.

downcase(*a*)—returns the symbol generated from *a* by making all characters in *a* lowercase.

upcase(*a*)—returns the symbol generated from *a* by making all characters in *a* uppercase.

swapcase(*a*)—returns the symbol generated from *a* by making all lowercase characters in *a* uppercase and all uppercase characters in *a* lowercase.

charAt(*a*,*b*)—returns the one character symbol consisting of the character of symbol *a* at index *b* (counting from 0); the result is undefined if *b* is less than 0 or greater than or equal to the length of *a*.

charAt(*a*,*b*,*c*)—returns the substring of symbol *a* beginning at index *b* (counting from 0), and continuing for *c* characters; the result is undefined if *b* is less than 0 or greater than or equal to the length of *a*, or if *c* is negative. If *a*+*b* is greater than the length of *a*, the result is the suffix of symbol *a* beginning at index *b*.

succ(*a*)—returns the symbol that is the successor of symbol *a*. If *a* contains characters or letters, the successor of *a* is found by incrementing the right-most letter or digit according to the Unicode collating sequence, carrying leftward if necessary when the last digit or letter in the collating sequence is encountered.

If *a* has no letters or digits, then the right-most character of *a* is incremented, with carries to the left as necessary.

toString(*a*)—returns a string whose characters correspond to the characters of symbol *a*.

toSymbol(*a*)—returns a symbol whose characters correspond to the characters of string *a*.

The Symbol ADT has no concatenation operations, but assuming we have a full-featured String ADT, symbols can be concatenated by converting them to strings, concatenating the strings, then converting the result back to a symbol. Similarly, String ADT operations can be used to do other manipulations. This explains whey the Symbol ADT has a rather odd mix of operations: The Symbol ADT models the `Symbol` class in Ruby, and this class only has operations often used for Symbols, with most string operations appearing in the `String` class.

The Ruby implementation of the Symbol ADT, as mentioned, hinges on making `Symbol` class instances immutable, which corresponds to the relative lack of operations in the Symbol ADT. `Symbol` values are stored in the Ruby interpreter's symbol table, which guarantees that they cannot be changed. This also guarantees that only a single `Symbol` instance will exist corresponding to any sequence of characters, which is an important characteristic of the Ruby `Symbol` class that is not required by the Symbol ADT, and distinguishes is from the `String` class.

All Symbol ADT operations listed above are implemented in the `Symbol` class, except *toSymbol*(), which is implemented in classes (such as `String`), that can generate a `Symbol` instance. When a result is undefined in the ADT, the result of the corresponding `Symbol` class method is `nil`. The names are sometimes different, following Ruby conventions; for example, *toString*() in the ADT becomes `to_s()` in Ruby, and *charAt*() in the ADT is `[]()` in Ruby.

Ruby is written in C, so carrier set members (that is, individual symbols) are implemented as fixed-size arrays of characters (which is how C represents strings) inside the `Symbol` class. The empty symbol is an array of length 0, symbols of length one are arrays with a single element, symbols of length two are arrays with two elements, and so forth. `Symbol` class operations are either written to use these arrays directly, or to generate a `String` instance, do the operation on the string, and convert the result back into a `Symbol` instance.

Range: A Structured Type in Ruby

Ruby has a several structured types, including arrays, hashes, sets, classes, streams, and ranges. In this section we will only discuss ranges briefly as an example of a structured type.

The *Range of T* ADT represents a set of values of type *T* (called the *base type*) between two extremes. The *start value* is a value of type *T* that sets the lower bound of a range, and the *end value* is a value of type *T* that sets the upper bound of a range. The range itself is the set of values of type *T* between the lower and upper bounds. For example, the Range of Integers from 1 to 10 inclusive is the set of values {1, 2, 3, ..., 10}.

A range can be *inclusive*, meaning that it includes the end value, or *exclusive*, meaning that it does not include the end value. Inclusive ranges are written with two dots between the extremes, and exclusive ranges with three dots. Hence the Range of Integers from 1 to 10 exclusive is the set {1, 2, 3, ..., 9}.

A type can be a range base type only if it supports order comparisons. For example, the Integer, Real, and String types support order comparisons and so may be range base types, but Sets and Arrays do not, so they cannot be range base types.

The carrier set of a Range of T is the set of all sets of values $v \in T$ such that for some start and end values $s \in T$ and $e \in T$, either $s \leq v$ and $v \leq e$ (the inclusive ranges), or $s \leq v$ and $v < s$ (the exclusive ranges), plus the empty set. For example, the carrier set of the Range of Integer is the set of all sequences of contiguous integers. The carrier set of the Range of Real is the set of all sets of real number greater than or equal to a given number, and either less than or equal to another, or less than another. These sets are called *intervals* in mathematics.

The operations of the Range of T ADT includes the following, where $a, b \in T$ and r is a value of Range of T:

> *a..b*—returns a range value (an element of the carrier set) consisting of all v $\in T$ such that $a \leq v$ and $v \leq b$.
>
> *a...b*—returns a range value (an element of the carrier set) consisting of all v $\in T$ such that $a \leq v$ and $v < b$.
>
> *a==b*—returns true if and only if *a* and *b* are identical.
>
> *min*(*r*)—returns the smallest value in *r*. The result is undefined if *r* is the empty set.
>
> *max*(*r*)—returns the largest value in *r*. The result is undefined if *r* has no largest value (for example, the Range of Real 0...3 has no largest value because there is no largest Real number less than 3).
>
> *cover?*(*r*, *x*)—returns true if and only if $x \in r$.

The Range of T ADT is a structured type because the values in its carrier set are composed of values of some other type, in this case, sets of value of the base type T.

Ruby implements the Range of T ADT in its `Range` class. Elements of the carrier set are represented in `Range` instances by recording the type, start, and end values of the range, along with an indication of whether the range is inclusive or exclusive. Ruby implements all the operations above, returning `nil` when the ADT operations are undefined. It is quite easy to see how to implement these operations given the representation elements of the carrier set. In addition, the `Range` class provides operations for accessing the begin and end values defining the range, which are easily accessible because they are recorded. Finally, the Range class has an `include?()` operation that tests range membership by stepping through the values of the range from start value to end value when the range is non-numeric. This gives slightly different results from `cover?()` in some cases (such as with `String` instances).

Review Questions

1. What is the difference between a simple and a structured type?
2. What is a pure object-oriented language?
3. Name two ways that `Symbol` instances differ from `String` instances in Ruby.
4. Is `String` a simple or structured type in Ruby? Explain.
5. List the carrier set of Range of {1, 2, 3}. In this type, what values are 1..1, 2..1, and 1...3? What is max(1...3)?

Exercises

1. Choose a language that you know well and list its simple and structures types.
2. Choose a language that you know well and compare its simple and structured types to those of Ruby. Does one language have a type that is simple while the corresponding type in the other language is structured? Which language has more simple types or more structured types?
3. Every Ruby type is a class, and every Ruby value is an instance of a class. What advantage and disadvantages do you see with this approach?
4. Write pseudocode to implement the `cover?()` operation for the Range class in Ruby.
5. Give an example of a Ruby `String` range `r` and `String` instance `v` such that `r.cover?(v)` and `r.include?(v)` differ.

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 `Integer` type in Ruby cannot be broken into parts, while the values of a `Range` in Ruby can be (the parts are the individual elements of the ranges).
2. A pure object-oriented language is one whose types are all classes. Java and C++, for example, are not pure object-oriented languages because they include primitive data types, such as `int`, `float`, and `char`, that are not classes. Smalltalk and Ruby are pure object-oriented languages because they have no such types.
3. `Symbol` instances in Ruby are immutable while `String` instances are mutable. `Symbol` instances consisting of a particular sequence of characters are unique, while there may be arbitrarily many `String` instances with the same sequence of characters.
4. `String` is a simple type in Ruby because strings are not composed of other values—in Ruby there is no character type, so a `String` value cannot be broken down into parts composed of characters. If `s` is a `String` instance, then `s[0]` is not a character, but another `String` instance.
5. The carrier set of Range of {1, 2, 3} is { {}, {1}, {2}, {3}, {1, 2}, {2, 3}, {1, 2, 3} }. The value 1..1 is {1}, the value 2..1 is {}, and the value 1...3 is {1, 2}, and max(1...3) is 2.

3: Arrays

Introduction

A structured type of fundamental importance in almost every procedural 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 it starts at 0. In the example above *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 that (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* × *m*,) where *m* is the size of an array element, and *b* is the base location of the array *a*. This computation is obviously 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, and abstract data types abstract from all details of implementation on a computer. Nonetheless, we can discuss arrays in a "semi-abstract" fashion that abstracts some implementation details. The definition above abstracts away the 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 will treat arrays as implementation mechanisms and not as ADTs.

Varieties of Arrays

In some languages, the size of an array must be established once and for all at program design time and cannot change during execution. Such arrays are called **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, such as 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 minimize its frequency of use. 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 needed. The hope is that providing a lot of extra space will avoid many expensive reallocation procedures if the array expands slowly over time.

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.

Arrays in Ruby

Ruby arrays are dynamic arrays that expand automatically whenever a value is stored in a location beyond the current end of the array. To the programmer, it is as if arrays are unbounded and as many locations as are needed are available. Locations not assigned a value in an expanded array are initialized to `nil` by default. Ruby also has an interesting indexing mechanism for arrays. Array indices begin at 0 (as in many other languages) so, for example, `a[13]` is the value in the 14th position of the array. Negative numbers are the indices of elements counting from the current end of the array, so `a[-1]` is the last element, `a[-2]` is the second to last element, and so forth. Array references that use an out-of-bound index return `nil`. These features combine to make it difficult to write an array reference that causes an indexing error. This is apparently a great convenience to the programmer, but actually it is not because it makes it so hard to find bugs: many unintended and erroneous array references are legal.

The ability to assign arbitrary values to arrays that automatically grow arbitrarily large makes Ruby arrays behave more like lists than arrays in other languages. We will discuss the List ADT later on.

Another interesting feature of Ruby arrays has to do with the fact that it is a pure object-oriented language. This means (in part) that every value in Ruby is an object, and hence

every value in Ruby is an instance of `Object`, the super-class of all classes, or one of its sub-classes. Arrays hold `Object` values, so any value can be stored in any array! For example, an array can store some strings, some integers, some floats, and so forth. This appears to be a big advantage for programmers, but again this freedom has a price: it much harder to find bugs. For example, in Java, mistakenly assigning a string value to an array holding integers is flagged by the compiler as an error, but in Ruby, the interpreter does not complain.

Ruby arrays have many interesting and powerful methods. Besides indexing operations that go well beyond those discussed above, arrays have operations based on set operations (membership, intersection, union, and relative complement), string operations (concatenation, searching, and replacement), stack operations (push and pop), and queue operations (shift and append), as well as more traditional array-based operations (sorting, reversing, removing duplicates, and so forth). Arrays are also tightly bound up with Ruby's iteration mechanism, which will be discussed later.

Review Questions

1. If an array holds integers, each of which is four bytes long, how many bytes from the base location of the array is the location of the fifth element?
2. Is the formula for finding the location of an element in a dynamic array different from the formula for finding the location of an element in a static array?
3. When a dynamic array expands, why can't the existing elements be left in place and extra memory simply be allocated at the end of the existing memory allocation?
4. If a Ruby array `a` has `n` elements, which element is `a[n-1]`? Which is element `a[-1]`?

Exercises

1. Suppose a dynamic integer array `a` with indices beginning at 0 has 1000 elements and the line of code `a[1000] = a[5]` is executed. How many array values must be moved from one memory location to another to complete this assignment statement?
2. Memory could be freed when a dynamic array shrinks. What advantages or disadvantages might this have?
3. To use a static array, data must be recorded about the base location of the array, the size of the elements (for indexing), and the number of elements in the array (to check that indexing is within bounds). What information must be recorded to use a dynamic array?
4. State a formula to determine how far the from base location of a Ruby array an element with index *i* is when *i* is a negative number.
5. Give an example of a Ruby array reference that will cause an indexing error at run time.
6. Suppose the Ruby assignment `a=(1 .. 100).to_a` is executed. What are the values of the following Ruby expressions? Hint: You can check your answers with the Ruby interpreter.
   (a) `a[5..10]`
   (b) `a[5...10]`

(c) `a[5, 4]`
(d) `a[-5, 4]`
(e) `a[100..105]`
(f) `a[5..-5]`
(g) `a[0, 3] + a[-3, 3]`

7. Suppose that the following Ruby statement are executed in order. What is the value of array `a` after each statement? Hint: You can check your answers with the Ruby interpreter.
   (a) `a = Array.new(5, 0)`
   (b) `a[1..2] = []`
   (c) `a[10] = 10`
   (d) `a[3, 7] = [1, 2, 3, 4, 5, 6, 7]`
   (e) `a[0, 2] = 5`
   (f) `a[0, 2] = 6, 7`
   (g) `a[0..-2] = (1..3).to_a`

Review Question Answers

1. If an array holds integers, each of which is four bytes long, then the fifth element is 20 bytes past the base location of the array.
2. The formula for finding the location of an element in a dynamic array is the same as the formula for finding the location of an element in a static array. The only difference is what happens when a location is beyond the end of the array. For a static array, trying to access or assign to an element beyond the end of the array is an indexing error. For a dynamic array, it may mean that the array needs to be expanded, depending on the language. In Ruby, for example, accessing the value of `a[i]` when `i` ≥ `a.size` produces `nil`, while assigning a value to `a[i]` when `i` ≥ `a.size` causes the array `a` to expand to size `i`+1.
3. The memory allocated for an array almost always has memory allocated for other data structures after it, so it cannot simply be increased without clobbering other data structures. A new chunk of memory must be allocated from the free memory store sufficient to hold the expanded array, and the old memory returned to the free memory store so it can be used later for other (smaller) data structures.
4. If a Ruby array `a` has `n` elements, then element `a[n-1]` and element `a[-1]` are both the last element in the array. In general, `a[n-i]` and `a[-i]` are the same elements.

4: Assertions

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.

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 arguments. 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 attributes and relationships between the attributes 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` attributes. 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 operation, but it must be true between executions of public operations. For example, an operation to add something to a container must increase the size and decrease the space left attributes, and for a moment during execution of this operation their sum might not be correct, but when the operation 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 every 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.

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 always be true of the values or the operations in the type. ADT assertions can describe many things about an ADT, but usually they are used to 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 statement about the properties of operations, often showing how operations are related to one another. For examples, 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 the result is between zero and the modulus less one. An axiom of this ADT is that for all natural numbers a, b, and $m > 0$,

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

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.

Using Assertions

When writing code, programmer should state pre- and post conditions for public operations of a class or module, state invariants for classes, and insert unreachable code assertions and loop invariants wherever appropriate.

Some languages have facilities to directly support assertions 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 class 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 exported operation. For this reasons, it is often not practical to do this. Unreachable code assertions occur relatively infrequently and they 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 of using a linear 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.

Assertions in Ruby

Ruby provides no support for assertions whatever. Furthermore, because it is weakly typed, Ruby does not even enforce rudimentary type checking on operation parameters and return values, which amount to failing to check type pre- and post conditions. This puts the burden of assertion checking firmly on the Ruby programmer.

Because it is so burdensome, it is not reasonable to expect programmers to perform type checking on operation parameters and return values. Programmers can easily document pre- and post conditions and class invariants, however, and insert code to check most value preconditions, and some post conditions and class invariants. Checks can also be inserted easily to check that supposedly unreachable code is never executed. Assertion checks can raise appropriate exceptions when they fail, thus halting erroneous programs.

For example, suppose that the operation `f(x)` must have a non-zero argument and return a positive value. We can document these pre- and post conditions in comments and incorporate a check of the precondition in this function's definition, as shown below.

```
# Explanation of what f(x) computes
# @pre: x != 0
# @post: @result > 0
def f(x)
  raise ArgumentError if x == 0
  ...
end
```

Figure 1: Checking a Precondition in Ruby

Ruby has many predefined exceptions classes (such as `ArgumentError`) and new ones can be created easily by sub-classing `StandardError`, so it is easy to raise appropriate exceptions.

We will use assertions frequently when discussing ADTs and the data structures and algorithms that implement them, and we will put checks in Ruby code to do assertion checking. We will also state pre- and post-conditions and class invariants in comments using the following symbols.

Use	Symbol
Mark precondition	`@pre:`
Mark post condition	`@post:`
Mark class invariant	`@inv:`
Return value	`@result`
Value of `@attr` after operation	`@attr`
Value of `@attr` before operation	`old.@attr`

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. Should a program attempt to catch assertion exceptions?
5. Can assertion checking be turned off easily in Ruby programs as it can in Eiffel or D programs?

Exercises

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

   Consider the following fragment of a class declaration in Ruby.

```
NUM_LOCKERS = 138

class Storage

  # Set up the locker room data structures
  def initialize
    @lockerIsRented = new Array(NUM_LOCKERS, false)
    @numLockersAvailable = NUM_LOCKERS
  end

  # Find an empty locker, mark it rented, return its number
  def rentLocker
    ...
  end

  # Mark a locker as no longer rented
  def releaseLocker(lockerNumber)
    ...
  end

  # Say whether a locker is for rent
  def isFree(lockerNumber)
    ...
  end

  # Say whether any lockers are left to rent
  def isFull?
    ...
  end
end
```

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 comment for the `Storage` class.
4. Write precondition comments and Ruby code for all the operations that need them in the `Storage` class. The precondition code may use other operations in the class.
5. Write post condition comments for all operations that need them in the `Storage` class. The post-condition comments may use other operations in the class.
6. Implement the operations in `Storage` class in Ruby.

Review Question Answers

1. A precondition is an assertion that must be true when an operation begins to execute. A post condition is an assertion that must be true when an operation completes execution. A class invariant in an assertion that must be true between executions of the operations 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 lop 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. Programs should not attempt to catch assertion exceptions because they indicate errors in the design and implementation of the program, so it is best that the program fail than that it continue to execute and produce possibly incorrect results.
5. Assertion checking cannot be turned off easily in Ruby programs because it is completely under the control of the programmer. Assertion checking can be turned of easily in Eiffel and D because assertions are part of the programming language, and so the compiler knows what they are. In Ruby, assertions are not supported, so all checking is (as far as the compiler is concerned) merely code.

5: Containers

Introduction

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

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 unstructured containers that allows their elements to be accessed using keys.

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 a 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), or membership operations.

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. 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. In object-oriented terms, a Container is an interface, not a class. These considerations lead to the UML class diagram in Figure 1 below.

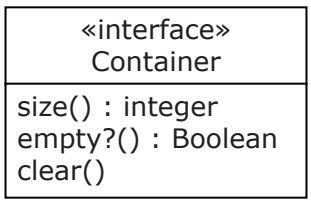


Figure 1: The Container Interface

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 addition, our container hierarchy appears in Figure 2.

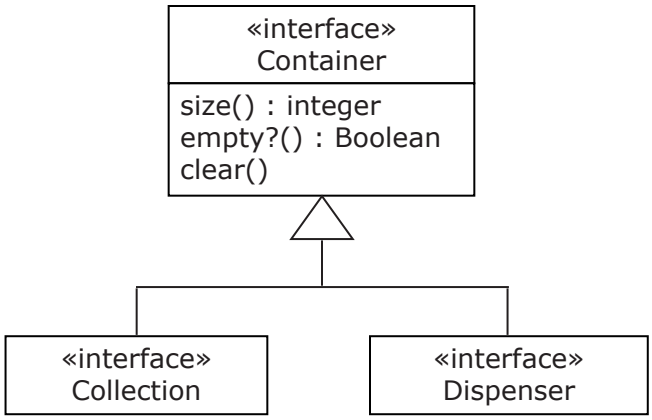


Figure 2: Top of the Container Taxonomy

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 these non-traversable containers.

Interfaces in Ruby

Recall that in object-oriented programming, and interface is a collection of abstract operations that cannot be instantiated. Although Ruby is object-oriented, it does not have interfaces. Interface-like classes can be constructed in Ruby by creating classes whose operations have empty bodies. These *pseudo-interface* classes can be used as super-classes of classes that implement the interface. Pseudo-interface classes can still be instantiated, and their operations can still be called without overriding them, so they are only a bit like real interfaces. One way to make them more like interfaces, however, is to implement the

operations in the pseudo-interface so that they raise exceptions if they are called. This forces sub-classes to override the operations in these pseudo-interface classes before they can be used.

This is the approach we will take in implementing interfaces in Ruby: we will make super-classes whose operations raise `NotImplementedError` exceptions. Classes implementing a pseudo-interface will then have to inherit from it and override its operations.

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.empty?() return? What does c.size() return?
3. What would happen if you tried to instantiate a pseudo-interface class implemented as suggested above in Ruby?

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?

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.empty?()` returns true, and `c.size()` returns 0.
3. A pseudo-interface class implemented in Ruby with operations that raise `NotImplementedError` exceptions when they are called could be instantiated without any problem, but such objects could not be used for anything useful. The only thing that such a class would be good for would be as a super-class of classes that override its operations, which is pretty much what an interface is good for as well.

6: Stacks

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.

The Stack ADT

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 essential operations of the type are the following (where *s* is a stack of *T*, and *e* is a *T* value).

> *push*(*s*,*e*)—Return a new stack just like *s* except that *e* has been added at the top of *s*.
>
> *pop*(*s*)—Remove the top element of *s* and return the resulting (shorter) stack. Attempting to pop an empty stack gives an undefined result. Thus, a precondition of the *pop()* operation is that the stack *s* is not empty.
>
> *empty?*(*s*)—Return the Boolean value true just in case *s* is empty.
>
> *top*(*s*)—Return the top element of *s* without removing it. Like *pop()*, this operation has the precondition that the stack *s* is not empty.

Besides the precondition assertions mentioned in the explanation of the stack ADT operations above, we can also state some axioms to help us understand the stack ADT. For example, consider the following axioms.

> For any stack *s* and element *e*, *pop*(*push*(*s*,*e*)) = *s*.
>
> For any stack *s* and element *e*, *top*(*push*(*s*,*e*)) = *e*.
>
> For any stack *s*, and element *e*, *empty?*(*push*(*s*,*e*)) = *false*.

The first axiom tells us that the *pop*() operation undoes what the *push*() operation achieves. The second tells us that when an element is pushed on a stack, it becomes the top element on the stack. The third tells us that pushing an element on an empty stack can never make it empty. With the right axioms, we can completely characterize the behavior of ADT operations without having to describe them informally in English (the problem is to know when we have the right axioms). Generally, we will not pursue such an axiomatic

specification of ADTs, though we may use some axioms from time to time to explain some ADT operations.

The Stack Interface

The stack ADT operations map stacks into one another, and this is done by having stacks as operation arguments and results. This is what one would expect from mathematical functions that map values from the carrier set of an ADT to one another. However, when implementing stacks in an object-oriented language that allows us to create a stack class with stack instances, there is no need to pass or return stack values—the stack instances hold values that are transformed into other values by the stack operations. Consequently, when we specify an object-oriented implementation of the stack ADT, all stack arguments and return values vanish. The same thing occurs as we study other ADTs and their implementations throughout this book: ADT carrier set arguments and return values will vanish from operation signatures in ADT implementation classes, and instead instances will be transformed from one carrier set value to another as operations are executed.

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

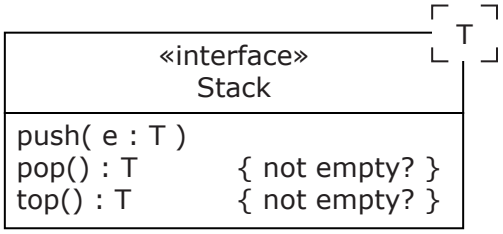


Figure 1: The `Stack` Interface

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 class icon. Note also that the preconditions of operations that need them are shown as UML properties enclosed in curly brackets to the right of the operation signatures. Finally, because `pop()` no longer needs to return a stack, it can return nothing, or as a convenience, it can return the element that is popped from the stack, which is what the operation in the `Stack` interface is designed to do.

Using Stacks—An Example

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 are to 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 pseudocode in Figure 2 describing the activities of the print spooler.

```
def printCollated(j : Job)
  Stack stack = Stack.new
  for each Page p in j do stack.push(p)
  while !stack.empty?
    print(stack.pop)
  end
end
```

Figure 2: 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.

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; it 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.

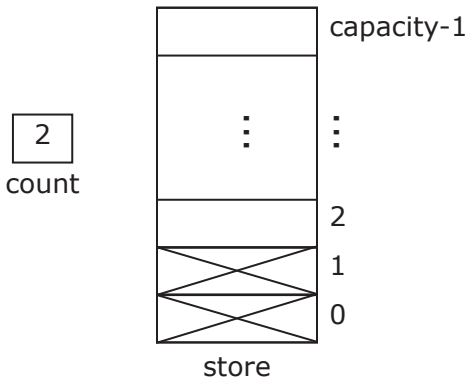


Figure 3: Implementing a Stack With an Array

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 3 below illustrates this data structure.

The marker is called count because it keeps track of the array location where the top element will go on the next push, and the array grows from index 0 upward, so the marker also records how many elements are in the stack.

The store array is allocated to hold capacity elements; this is the maximum number of items that can be placed on the stack before it must 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 also 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 must be expanded when count equals capacity and another element is pushed on the stack.

Implementing the operations of the stack ADT using this data structure is quite 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, the store is expanded. Then the pushed value is assigned to location store[count] and then count is incremented.

A class realizing this implementation might be called ArrayStack(T). It would implement the Stack(T) interface and have the store array and the count variable as private attributes and the stack operations as public methods. Its constructor would create an empty stack. The capacity variable could be maintained by the ArrayStack class or be part of the implementation of dynamic arrays in the implementation language.

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 review the basics of linked data structures.

> **Node**: An aggregate variable composed of data and link 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 needed 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 4 illustrates this data structure. The reference to the head of the list is called `topNode`. Each node has an `item` field (for values of type *T*) and a `link` field (for the references). The figure shows a stack with

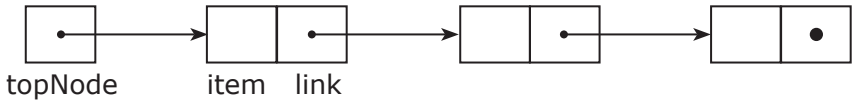


Figure 4: Implementing a Stack With a Singly Linked List

three elements; the top of the stack, of course, is the first node on the list. The stack is empty when `topNode` is null; the stack never becomes full (unless memory is exhausted).

Implementing the operations of the stack ADT using a linked data structure is quite simple. For example, to implement the `push()` operation, a new node is created with its `item` field set to the new top element and its `link` field set to the current value of `topNode`. The `topNode` variable is then assigned a reference to the new node.

A class realizing this implementation might be called `LinkedStack(T)`. It would implement the `Stack(T)` interface and have `topNode` as a private attribute and the stack operations as public methods. It might also have a private inner `Node` class for node instances. Its constructor would create an empty stack. As new nodes are needed when values are pushed onto the stack, new `Node` instances would be instantiated and used in the list. When values are popped off the stack, `Node` instances would be freed and their space reclaimed. A `LinkedStack` would thus use only as much space as was needed for the elements it holds.

Summary and Conclusion

Both ways of implementing 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.

Review Questions

1. The *pop*() operation in the stack ADT returns a stack, while the `pop()` operation in the `Stack` interface returns a value of type *T*. Why are these so different?

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?

Exercises

1. State three more axioms about the stack of *T* ADT.
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. What should happen if a precondition of a Stack operation is violated?
4. How can a programmer who is using an ArrayStack or a LinkedStack make sure that his code will not fail because it violates a precondition?
5. Should a LinkedStack have a count attribute? Explain why or why not.
6. Suppose a LinkedStack has a count attribute. State a class invariant relating the count and topNode attributes.
7. Could the top element be stored at the tail of a LinkedStack? What consequences would this have for the implementation?
8. A LinkedStack could be implemented using a doubly-linked list. What are the advantages or disadvantages of this approach?
9. As noted before, every Ruby value is an object, so every Ruby value has the same type (in a broad sense). What consequences does this have in implementing the stack of *T* ADT?
10. Implement the Stack interface and the ArrayStack and LinkedStack classes in Ruby.

Review Question Answers

1. The stack ADT *pop*() operation is a mathematical function that shows how elements of the carrier set of the ADT are related to one another, specifically, it returns the stack that is the result of removing the top element of the stack that is its argument. The pop() operation of the Stack interface is an operation that alters the data stored in a stack container by removing the top element stored; the new value of the ADT is represented by the data in the container, so it need not be returned as a result of the operation. For convenience, pop() returns the value that is removes.
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 unspecified capacity (such as a resizable ArrayStack or 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 reference, 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 far more space than it uses at any given moment to store data—usually there will be at least as many unused elements of the store array than used elements. This is because the ArrayStack must have enough capacity to accommodate the largest number of elements that could ever be pushed on the stack, but most of the time relatively little of this space is actually in use. On balance, then, ArrayStacks will typically use more space than LinkedStacks.

7: Queues

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 a operating system.

The Queue ADT

Queues are containers, and they hold 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 essential operations of the type are the following (*q* is a queue of *T* and *e* is a *T* value).

> *enter*(*q*,*e*)—Return a new queue just like *q* except that *e* has been added at the rear of *q*.
>
> *leave*(*q*)—Remove the front element of *q* and return the resulting (shorter) queue. Attempting to remove an element from an empty queue gives an undefined result; a precondition of the *leave()* operation is that *q* is not empty.
>
> *empty?*(*q*)—Return the Boolean value true just in case *q* is empty.
>
> *front*(*q*)—Return the front element of *q* without removing it. Like *leave*(), this operation has the precondition that *q* is not empty.

The queue ADT operations thus transform queues into one another. When implemented, a single data structure stores the values in the queue, so there is generally no need to return queues once they have been changed. Hence the operations of a queue type usually have slightly different parameters and return types, as we will see next.

The Queue Interface

A `Queue` interface is a sub-interface of `Dispenser`, which is a sub-interface of `Container`, so it already contains an `empty?()` operation that is has 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.

Note that a generic or template is used to generalize the interface for any element type. Note also that preconditions have been added for the operations that need them.

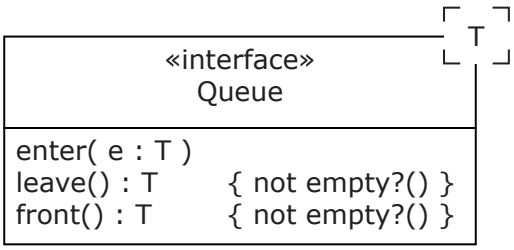


Figure 1: The Queue Interface

Using Queues—An Example

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( Job.new(d) );
}

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

Figure 2: Using A Queue to Spool Pages for Printing

The print spooler has a job 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.

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 (if any) would always be stored at location 0, and record the size of the queue, implying that the rear element would be at location size-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.

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

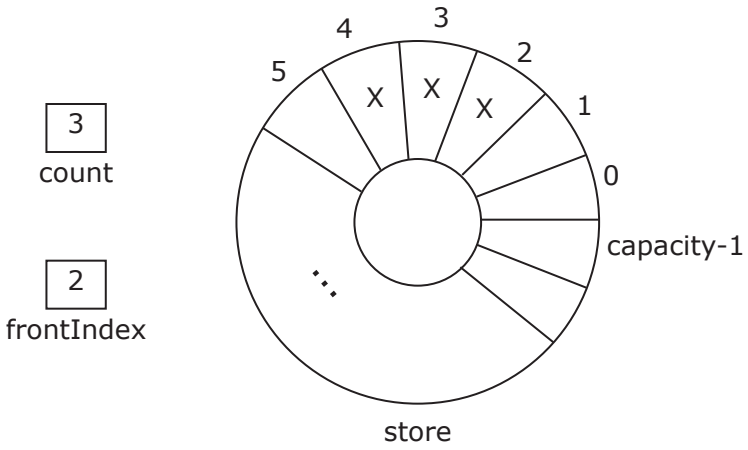


Figure 3: Implementing a Queue With a Circular Array

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) mod 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. Usually 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 is acquired at 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) mod capacity, count is decremented, and the value stored in the temporary variable is returned.

A class realizing this implementation might be called ArrayQueue(T). It would implement the Queue(T) interface and have the store array and the frontIndex and count variables as private attributes. The queue operations would be public methods. Its constructor would create an empty queue. The value of capacity could be a constant defined at compile time, or a constructor parameter (if dynamic arrays are available).

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 reference 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 frontNode and rearPNode references to keep track of both ends of the list.

If rearNode refers to the head of the list and frontNode 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 frontNode refer to the head of the list, and rearPtr to its tail. Figure 4 illustrates this data

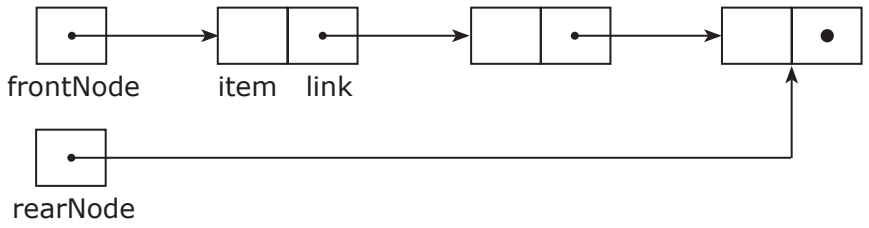


Figure 4: Implementing a Queue With a Linked List

structure. Each node has data field (for values of type *T*) and a link field (for the references). The figure shows a queue with three elements. The queue is empty when the frontNode and rearNode references are null; the queue never becomes full (unless memory is exhausted).

Implementing the operations of the Queue ADT using a linked data structure is quite simple, though some care is needed to keep the two references synchronized. For example, to implement the leave() operation, a check is first made that the queue is not empty. If not, then the value field of the front node is assigned to a temporary variable. The frontNode variable is then assigned the link field of the first node, which removes the first node from

the list. If frontNode is null, then the list has become empty, so rearNode must also be assigned null. Finally, the value saved in the temporary variable is returned.

A class realizing this implementation might be called LinkedQueue(T). It would implement the Queue(T) interface and have frontNode and rearNode as private attributes and the queue operations as public methods. It might also have a private inner Node class for node instances. 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.

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 nodes to store the values actually kept in the queue.

Overall, then, the linked implementation of queues seems slightly better than the contiguous implementation.

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?

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. 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 Ruby code for the enter() operation that includes arranging the data where it needs to be in the newly allocated array.
3. Write a class invariant for a LinkedQueue class whose attributes are frontNode, rearNode, and count.
4. Write a version of the LinkedQueue class that does not have a count attribute.
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 and write Ruby code for the enter() and leave() operations to illustrate this technique.

6. A LinkedQueue could be implemented using a doubly-linked list. What are the advantages or disadvantages of this approach?
7. Implement the Queue interface and the ArrayQueue and LinkedQueue classes in Ruby.

Review Question Answers

1. The *leave*() and *front*() operations both have as their precondition that the queue *q* 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.

8: Stacks and Recursion

Introduction

Before moving on from discussing dispensers to discussing collections, we should briefly discuss 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 "unwound" 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.

Balanced Brackets

Because of its simplicity, we begin with an example that doesn't really need a stack or recursion to solve, 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 as follows:

1. The empty string and the string "[]" are both strings 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, written in Ruby, checks whether a string of brackets is balanced.

```
def recursive_balanced?(string)
  return true if string.empty?
  source = StringEnumerator.new(string)
  check_balanced?(source) && source.empty?
end

def check_balanced?(source)
  return false unless source.current == '['
  loop do
    if source.next == '['
      return false if !check_balanced?(source)
    end
    return false unless source.current == ']'
    break if source.next != '['
  end
  true
end
```

Figure 1: Recursive Algorithm For Checking String of Balanced Brackets

The `recursive_balanced?()` operation has a string argument containing only brackets. It does some initial and final processing, but most of the work is done by the recursive helper function `check_balanced?()`. Note that a `StringEnumerator` is created from the string argument and passed to the `check_balanced?()` operation. We will discuss enumerators later, but for now it suffices to say that a `StringEnumerator` is a class that provides characters from a string one by one when asked for them. It also indicates when all the characters have been provided, signalling the end of the string.

The algorithm is based on the recursive definition of balanced brackets. If the string is empty, then it is a string of balanced brackets. This is checked right away by `recursive_balanced?()`. If the string is not empty, then `check_balanced?()` is

called to check the string. It first checks the current character to see whether it is a left bracket, and returns false if it is not. It then considers the next character. If it is another left bracket, then there is a nested string of balanced brackets, which is checked by a recursive call. In any case, a check is then made for the right bracket matching the initial left bracket, which takes care of the other basis case in the recursive definition. The loop is present to take care of the case of a sequence of balanced brackets, as allowed by the recursive definition. Finally, when `check_balanced?()` returns its result to `recursive_balanced?()`, the latter checks to make sure that the string has all been consumed, which guarantees that there are no stray brackets at the end of the string.

This same job could be done just as well with a non-recursive algorithm using a stack. In the code in Figure 2 below, again written in Ruby, 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).

```
def stack_balanced?(string)
  stack = LinkedStack.new
  string.chars do | ch |
    case
    when ch == '['
      stack.push(ch)
    when ch == ']'
      return false if stack.empty?
      stack.pop
    else
      return false
    end
  end
  stack.empty?
end
```

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

In this case the recursive algorithm is about as complicated as the stack-based algorithm. In the examples below, we will see that sometimes the recursive algorithm is simpler, and sometimes the stack-based algorithm is simpler, depending on the problem.

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 all binary. All the expressions on 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 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 Ruby to evaluate prefix expressions recursively appears in Figure 3 below. The main operation `recursive_eval_prefix()` accepts a string as an argument. Its job is to create a `StringEnumeration` object to pass along to the recursive helper function, and to make sure that the string has all been read (if not, then there are extra characters at the end of the expression). The real work is done by the `eval_prefix()` operation, which is recursive.

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*.

The `eval_prefix()` operation first checks to see whether the string is exhausted and throws an exception if it is (because the empty string is not a prefix expression). Otherwise, it fetches the current character and advances to the next character to prepare for later processing. If the current character is a digit, 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, the 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. 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. The `evaluate()` operation is a helper function that simply applies the operation indicated in its `op` argument to its `left_arg` and `right_arg` values.

```
 def recursive_eval_prefix(string)
   source = StringEnumerator.new(string)
   result = eval_prefix(source)
   raise "Too many arguments" unless source.empty?
   result
 end

def eval_prefix(source)
   raise "Missing argument" if source.empty?
   ch = source.current
   source.next
   if ch =~ /\d/
     return ch.to_i
   else
     left_arg = eval_prefix(source)
     right_arg = eval_prefix(source)
     return evaluate(ch,left_arg, right_arg)
   end
 end
```

Figure 3: Recursive Algorithm to Evaluate Prefix Expressions

This recursive algorithm is extremely simple, yet is does a potentially very complicated job. In contrast, algorithms to evaluate prefix expressions using a stack are quite a bit more complicated. One such an algorithm is shown below in Figure 4. This algorithm has two stacks: one for (integer) left arguments, and one for (character) operators.

```
def stack_eval_prefix(string)
  raise "Missing expression" if string.empty?
  op_stack = LinkedStack.new
  left_arg_stack = LinkedStack.new
  left_arg = right_arg = nil
  string.chars do | ch |
    case
    when ch =~ /\d/
      if left_arg == nil
        left_arg = ch.to_i
      else
        right_arg = ch.to_i
        loop do
          raise "Missing operator" if op_stack.empty?
          right_arg = evaluate(op_stack.pop,left_arg,right_arg)
          if left_arg_stack.empty?
            left_arg = right_arg
            break
          else
            left_arg = left_arg_stack.pop
          end
        end
      end
    when ch =~ /[+\-*\/%]/
      op_stack.push(ch)
      if left_arg != nil
        left_arg_stack.push(left_arg)
        left_arg = nil
      end
    end
  end
  raise "Missing argument" if !op_stack.empty?
  raise "Too many arguments" unless left_arg_stack.empty?
  raise "Missing expression" unless left_arg
  left_arg
end
```

Figure 4: Stack-Based Algorithm to Evaluate Prefix Expressions

The strategy of this algorithm is to process each character from the string in turn, pushing operators on the operator stack as they are encountered and left arguments on the left argument stack as necessary. When a right argument is encountered, as many operations are applied as possible until arguments run out. Once the string is exhausted, the result

value should be stored in the `left_arg` variable, and the left argument and operator stacks should both be empty—if not, then there are either too many arguments or too many operators.

Clearly, this stack-based evaluation algorithm is much 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.

```
def stack_eval_postfix(string)
  stack = LinkedStack.new
  string.chars do | ch |
    case
    when ch =~ /\d/
      stack.push(ch.to_i)
    when ch =~ /[+\-*\/%]/
      raise "Missing argument" if stack.empty?
      right_arg = stack.pop
      raise "Missing argument" if stack.empty?
      left_arg = stack.pop
      stack.push( evaluate(ch, left_arg, right_arg) )
    end
  end
  raise "Missing expresion" if stack.empty?
  raise "Too many arguments" unless stack.size == 1
  stack.pop
end
```

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 input string. 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 string 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 input string is exhausted, then the input expression is not well-formed.

The recursive algorithm for evaluating postfix expressions is quite complicated. The strategy is the 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, and in other cases a stack-based algorithm 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.

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 need to use a stack. If a recursive algorithm is such that at most 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 compute the factorial function, like the one shown in Figure 6.

```
def recursive_factorial(n)
  raise ArgumentError if n < 0
  (n <= 1) ? 1 : n * recursive_factorial(n-1)
end
```

Figure 6: A Recursive Factorial Algorithm

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

```
def factorial(n)
  raise ArgumentError if n < 0
  product = 1
  n.downto(1).each { | i | product *=  i }
  product
end
```

Figure 7: A Non-Recursive Factorial Algorithm

Algorithms that only call themselves at most once as the final step in every execution of their bodies, like the factorial 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.

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.

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.

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.

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 one 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, 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).

9: Collections

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 Ruby. Based on this discussion, we will decide how to support collection iteration in our container hierarchy, and how to add collections to the hierarchy.

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 a function or a block), 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 control iteration over a collection, the iteration is said to be *external*. In this case, a collection must provide 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 for where the iteration mechanism resides.

> **In the language**—An iteration mechanism may be built into a language. For example, Java, Visual Basic, and Ruby have special looping control structures that provide means for external iteration over collections. Ruby has a special control structure for yielding control to a block passed to an operation, which provides support for internal iteration.
>
> **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 object**—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 object, and the collection usually has an operation to create iterators for it.

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 object, and external iteration residing in the language, in the collection, or in an iterator object. 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 iterators (this is known as the Iterator design pattern). Nowadays many languages provide external iteration in control structures, as mentioned above. Ruby provides five of the six alternatives! We will now consider the Iterator design pattern, and then at iteration in Ruby.

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 examples, a particular 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 design pattern that specifies the composition and interactions of a few 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 complete.
>
> *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.

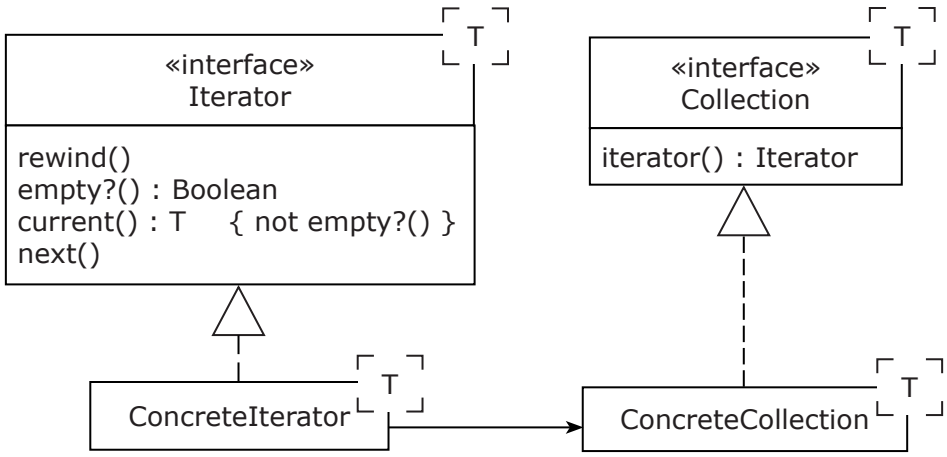


Figure 1: The Iterator Pattern

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 asking a ConcreteCollection for one by calling its iterator() operation. The client can then rewind the iterator and use a while loop to access its elements. The pseudocode below in Figure 2 illustrates how this is done.

```
c = ConcreteCollection.new
...
i = c.iterator
i.rewind
while !i.empty?
  element = i.current
  // process element
  i.next
end
```

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.

Iteration in Ruby

As mentioned, Ruby supports five of the six alternatives for doing iteration: there is no support in Ruby for external iteration residing in collections. This is probably because there are so many other ways to iterate over collections that there is no need for this alternative. Lets now consider the five different ways of doing iteration in Ruby.

Internal iteration supported by the language. A block of code may be passed as a special parameter to any Ruby operation, and this operation may then yield control (along with data arguments) to the block. This mechanism, which is not tied to collections but can be used anywhere in the language, can be considered a special form of internal iteration with control residing in the language itself. This is also the way that internal iterators are implemented for collections defined by programmers.

External iteration provided by the language. Ruby has a `for/in` loop that can be used with any collection. Each element of the collection is assigned in turn to a loop variable, and the body of the loop can then access and process the elements of the collection through this loop variable.

Internal iteration in a collection. This is the preferred collection traversal mechanism in Ruby. `Enumerable` is a special mixin module that provides over twenty internal iteration operations. All built-in collections mix in this module (and user-defined collections should as well). The internal iteration operations all accept parameterized blocks as the packaged process and apply the block to each element of the collection.

Internal and external iterator objects. In Ruby, an *enumerator* is an entity whose job is to iterate over a collection; in other words, an enumerator is an iterator object. Enumerators are created from any `Enumerable` object by calling `to_enum()` or `enum_for()`. All enumerators mix in the `Enumerable` module, so they include all the internal collection iterator operations. Hence they are internal iteration objects. Furthermore, they also include operations for external iteration: `rewind()` initializes an enumerator in preparation for iteration, and `next()` fetches the next object in the iteration (thus combining the actions of fetching the current element and advancing to the next element). If `next()` is called after iteration is complete, it raises a `StopIteration` exception, which is how iteration is terminated. The `loop` control structure catches this exception and terminates automatically, so explicitly handling exceptions is often not necessary when using an enumerator as an external iterator.

Although Ruby offers so many alternative forms of iteration, it is clearly designed to favor internal iteration. Usually, this is not a problem, but occasionally external iterators are needed to increase control over iteration. Ironically, the external iteration mechanisms in Ruby do not provide much additional flexibility. The `for/in` construct is quite rigid, and enumerators lack a non-exception based means of determining iteration termination. We can, if we like, enrich our container hierarchy by adding better external iteration facilities.

Collections, Iterators, and Containers

There are many sorts of collections, including simple linear sequences (lists), unordered aggregates (sets), and aggregates with keyed access (maps). There are not many operations common to this wide 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.

Two operations do come to mind, however: we may ask of a collection whether it contains some element, and we may compare collections to see whether they are equal. Although the Ruby `Object` class includes equality operations, adding one to the Collection interface will force concrete collections to implement it. We also add a collection containment query operation to the Collection interface.

In the spirit of Ruby, it seems wise to mix the Ruby `Enumerable` module into the Collection interface. This provides a rich variety of internal iterators. Given the drawbacks of internal iteration, it might be advisable to include external iterators based on the Iterator design pattern as well. Hence we include an `iterator()` factory method in the Collection interface.

Figure 3 shows the final Collection interface and its place in the Container hierarchy, along with the Ruby `Enumerable` mixin, which is indicated in UML using a dependency arrow.

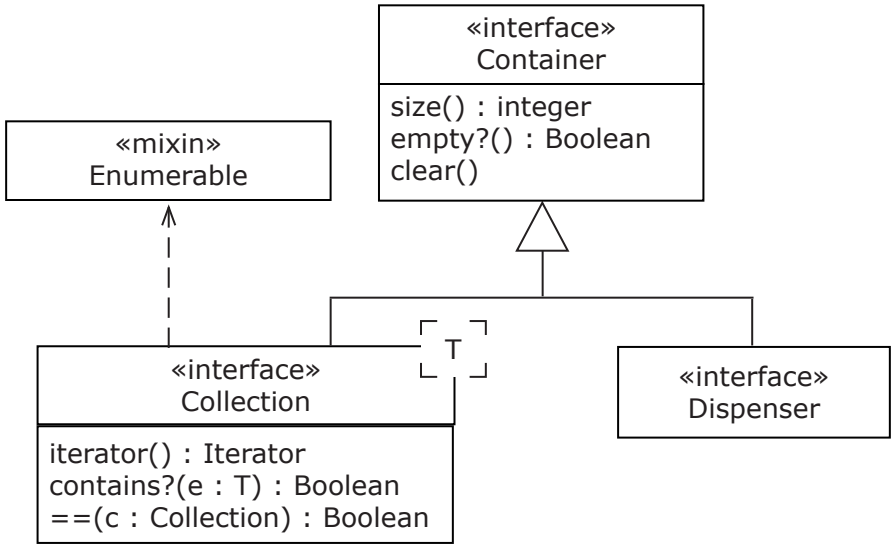


Figure 3: The Collection Interface in the Container Hierarchy

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 powerful possibility, but Ruby favors and supports internal iteration. We will incorporate both Ruby internal iterators and external iterator objects from the Iterator design pattern in our Collection interface.

Collections should also include an equality operation that indicates whether two collections are the same, along with a collection containment operation. Both of these operations appear in our Collection interface.

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. Which of the six iteration design alternatives does Ruby not support?
7. What is the `Enumerable` module in Ruby?
8. What does the contains?() operation return when its Collection object is empty?

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 an iterators, but the Java Iterator interface does not have a rewind() operation. Why not?
4. The Java Iterator interface includes a hasNext() operation corresponding to the empty?() operation in the Iterator pattern, but only a next() operation rather than both next() and current() operations. How do you explain this difference?
5. Would it be possible to have an Iterator interface with only a single operation? If so, how could the four Iterator functions be realized?
6. How might external iterators residing in collections be added to Ruby?
7. Write a generic Ruby implementation of the Collection contains?() operation using
   (a) an internal collection iterator
   (b) an external iterator object
   (c) an internal iterator object
   (d) the `for/in` loop construct
8. What happens to an iterator (any iterator) when its associated collection changes?
9. Consider the problem of checking whether a collection contains duplicates. Can this problem be solved using collection iterators? Can it be solved using iterator objects?

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 object (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 (based on a yield statement), (2) external control residing in a language (based on a for/in loop), (3) internal control residing in the collection, (4) external control residing in the collection, (5) internal control residing in an iterator object, (5) external control residing in an iterator object.
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 object 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. Ruby does not support external iteration residing in the collection.
7. The `Enumerable` module is used to mix in operations for internal iteration to other classes. It is used to add internal iteration facilities to collections and to enumerators, which are iterator objects.
8. If a Collection is empty, then it contains nothing, so the contains?() operation returns false no matter what its argument.

10: Lists

Introduction

Lists are simple 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 really is 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 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.

The List ADT

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), the lists with three elements of type *T* (ordered triples), 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 operations that may be included in a list ADT; the following operations are common (*s* and *t* are lists of *T, i* is an index, and *e* is a *T* value, *n* is a length, and the value *nil* is a special value that indicates that a result is undefined).

size(*t*)—Return the length of list *t*.

insert(*t*,*i*,*e*)—Return a list just like *t* except that *e* has been inserted 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: -*size*(*t*) ≤ *i* ≤ *size*(*t*). When *i* is negative, elements are counted backwards from the end of the list starting at -1, and *e* is inserted after the element; when *i* is 0, *e* is inserted at the front of the list; and when *i* is *size*(*t*), *e* is appended to the end of the list.

delete_at(*t*,*i*)—Remove the element at index *i* of *t* and return the resulting (shorter) list. The precondition of this operation is that *i* be a valid index position, with negative indices indicating a location relative to the end of the list: -*size*(*t*) ≤ *i* < *size*(*t*).

t[*i*]—Return the value at index *i* of *t*. Its precondition is that *i* be a valid index position: -*size*(*t*) ≤ *i* < *size*(*t*).

t[*i*]=*e*—Replace the element at index *i* of *t* and return the resulting (same sized) list. The precondition is that *i* be a valid index position: -*size*(*t*) ≤ *i* < *size*(*t*).

index(*t*,*e*)—Return the index of the first occurrence of *e* in *t*. If *e* is not in *t*, return *nil*.

slice(*t*, *i*, *n*)—Return a new list that is a sub-list of *t* whose first element is the value at index *i* of *t* and whose length is *n*. The precondition of this operation is that *i* is valid

and n is non-negative: $-size(t) \le i < size(t)$ and $0 \le n$. The sub-list does not extend past the end of t no matter how large n is.

t==s—Return true if and only if s and t have the same elements in the same order.

As with the ADTs we have studied before, an object-oriented implementation of these operations as instance methods will include the list as an implicit parameter, so the signatures of these operations will vary somewhat when they are implemented.

The List Interface

A List interface is a sub-interface of Collection, which is a sub-interface of Container, so it has several operations that it has inherited from its ancestors. The diagram in Figure 1 shows the List interface.

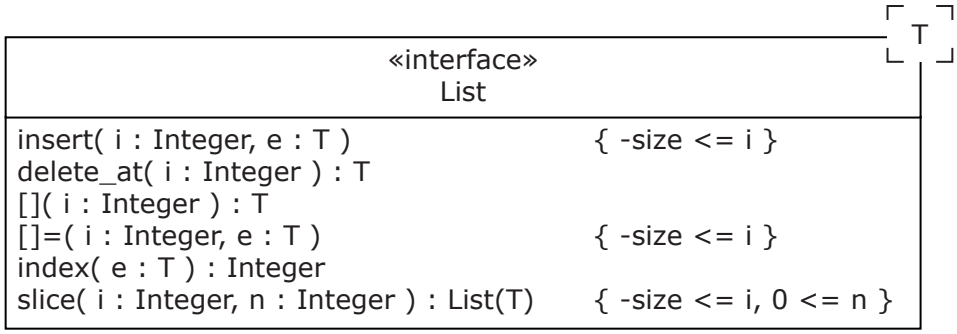


Figure 1: The List Interface

Note that a template parameter is used to generalize the interface for any element type. Note also that preconditions have been added, though they differ slightly from those in the ADT. In particular, when querying a list, if an index is out of bounds, then nil is returned. When inserting or replacing an element in a list, if the index is beyond the upper bound, then the list is extended with nil values until it is large enough, and then the operation is performed. If delete_at() is given an out of bounds index, nothing is changed.

Using Lists—An Example

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 to-do lists. Iteration over a List to display its contents is easy with an internal or external iterator. The insert() and delete_at() operations allow items to be inserted into the list, removed from it, or moved around in it. The [] operation can be used to obtain a list element for display during a mouse-over event, and the []= 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.

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 object 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 `[]` and `[]=` list 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 of lists where elements ore often accessed but not too often added or removed, the contiguous implementation of lists will be very efficient; applications that have the opposite behavior, will be much less efficient, especially if the lists are long.

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 reference, traditionally called `head`, holding null when the list is empty, and a reference to the first node in the list otherwise. The length of 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 by walking 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. In any case, if lists are long and there are many changes towards the end of the list, much processing will be done simply finding the right spots in the list to do things.

This difficulty can be alleviated by keeping a special cursor consisting of an index number and a pointer 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 be useful to maintain a pointer to the end of the list. Then operations at the end of the list can be done quickly in exchange for the slight extra effort of maintaining the extra pointer. 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 reference to both its successor and its predecessor nodes. Figure 2 below illustrates this setup. 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 pointer 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 back of the list, or the cursor, which is often somewhere in the middle of the list.

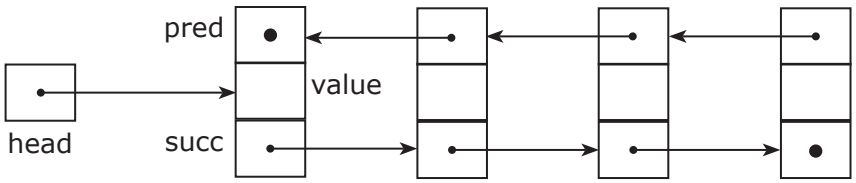


Figure 2: A Doubly Linked List

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

Iterators for the linked implementation of lists must obtain a pointer to the head of the list, which can be passed to the instance constructor. Then it is merely a question of maintaining a cursor and walking down the list as the `Iterator.next()` operation is called.

Modifying linked implementations of lists 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. A linked implementation is a good choice when a variety of list operations are needed.

Implementing Lists in Ruby

The Ruby array is already very similar to a contiguously implemented list; it lacks only `contains?()` and `iterator()` operations. Hence the simplest way to implement an `ArrayList` in Ruby is to make the `ArrayList` a sub-class of the built-in Ruby `Array` class and add the two missing operations. Of course, the `iterator()` factory method must return an instance of an `Iterator` that traverses the `ArrayList`, so an `ArrayListIterator` class must also be written.

A `LinkedList` can be implemented in Ruby by making linked lists using a `Node` class containing data and reference fields and writing operations to perform list operations as discussed above. In this case `LinkedList` is a sub-class of `List`, an interface that is a sub-class of `Collection`, which in turn is a sub-class of `Container`.

Summary and Conclusion

The contiguous implementation of lists is very easy to program and quite efficient for element access, but quite slow for element addition and removal. The linked implementation is considerably more complex 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. 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.

Review Questions

1. Does it matter where list element numbering begins?
2. What does the list ADT *index*(*t*,*e*) operation return when *e* is not in list *t*?
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?

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 do the delete_at() and []() operations not have preconditions while the insert(), []=(), and slice() operations do have preconditions?
5. An Iterator must have a reference to the Collection with which it is associated. How is such a connection made if a concrete iterator class is separate from its associated collection class? For example, suppose the ArrayListIterator is a class entirely separate from the ArrayList class. How can an ArrayListIterator instance obtain a connection to the particular ArrayList instance it is supposed to traverse?
6. 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.
7. Write ArrayList and ArrayListListIterator implementations in Ruby. Note that the List interface is very almost entirely implemented by the Ruby `Array` class: all that is missing is the iterator() and contains?() operations.
8. Write an ArrayListListIterator to go along with the implementation in the last exercise.
9. Write a LinkedList implementation in Ruby that uses a singly-linked list, no reference to the end of the list, and a cursor. Write a LinkedListIterator to go along with it.
10. Write a LinkedListListIterator to go along with the LinkedList class in the previous exercise.
11. Write a LinkedList implementation in Ruby that uses a doubly-linked circular list and a cursor. Write a LinkedListIterator to along with it.
12. Write a LinkedListListIterator to go along with the LinkedList class in the previous exercise.

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*(*t*,*e*) operation cannot return an index when *e* is not in list *t*. It should return a value indicating that the result is undefined, namely *nil*.
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 it is changed near its beginning often. 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 around than in a singly linked list, which can speed up the most expensive part of linked list operations: finding the node 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 to 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 operation, then it can be used to get to a node in the middle of the list very 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 the end of the list).

11: Analyzing Algorithms

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 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 many 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.

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 also interfere with program timings.
- Program must be written by programmers, some of whom write very fast code and other of whom 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 program 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 values are subtracted and the result written down 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, we will 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.*

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.

Which Operations to Count

In most cases, certain operations are done far more often than others in 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 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 incrementing and comparison operations), does not affect the overall efficiency classification of an algorithm.

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 certain value.

```
def find(key, array)
  array.each do | element |
    return true if key == element
  end
  return false;
end
```

Figure 1: An Array 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 value of 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 and then goes on to derive 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, 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. So 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 the index of the key in the list, and -1 or nil if the key is not present—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.

```
def max(array)
  return nil if array.empty?
  result = array[0]
  1.upto(array.size-1).each do | index |
    result = array[index] if result < array[index]
  end
  return result
end
```

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; we will use $C(n)$ to designate the complexity of such an algorithm on an input of size n, so for the maximum-finding algorithm, $C(n) = n-1$.

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.

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.

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?

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 Ruby sequential search method that finds the index of a key in an array.
3. Consider the Ruby code below.

```
def max_char_sequence(string)
  return 0 if string.empty?
  max_len = 0
  this_len = 1
  last_char = nil
  string.each_char do | this_char |
    if this_char == last_char
      this_len += 1
    else
      max_len = this_len if max_len < this_len
      this_len = 1
    end
    last_char = this_char
  end
  return (max_len < this_len) ? this_len : max_len
end
```

   (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.

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 analyzer 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 analyzer 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, and average case behavior are about differences in behavior given input of a certain size.

12: Function Growth Rates

Introduction

We have set up an approach for determining the amount of work done by algorithms 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 $C(n) = 2n-3$ 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.

Definitions and Notation

Our goal is the classify functions into groups that such all the functions in a group grow no faster than some reference function for the group. Informally, $O(f)$ (read big-oh of f)is the set of functions that grows no faster than $f(n)$ (that is, those that grow more slowly than $f(n)$ or at the same rate as $f(n)$).

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
>
> $$g(n) \leq c \cdot f(n) \text{ for all } n \geq n_0$$

In other words, g is in $O(f)$ if at some point $g(n)$ is never greater than some multiple of $f(n)$. The following are examples.

$$8n+5 \in O(n)$$

$$8n+5 \in O(n^2)$$

$$6n^2+23n-14 \in O(4n^2-18n+65)$$

$$n^k \in O(n^p) \text{ for all } k \leq p$$

$$\log n \in O(n)$$

$$2^n \in O(n!)$$

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 the universe, even on the fastest possible computers.

Establishing the Order of Growth of a Function

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

> **Theorem**: $g \in O(f)$ if the $\lim_{n\to\infty} g(n)/f(n) = c$, for $c \geq 0$.

For example, to show that $3n^2+2n-1 \in O(n^2)$ we need to consider $\lim_{n\to\infty} (3n^2+2n-1)/n^2$:

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

Because this limit is not infinite, $3n^2+2n-1 \in O(n^2)$.

Another theorem that is also very useful in solving limit problems is L'Hôpital's Rule:

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

To illustrate the use of L'Hôpital's Rule, lets determine whether $n^2 \in O(n \lg n)$. First note that $\lim_{n\to\infty} n^2 = \lim_{n\to\infty} n \lg n = \infty$, so L'Hôpital's Rule applies.

$$\lim_{n\to\infty} n^2/(n \lg n) = \lim_{n\to\infty} n/(\lg n)$$
$$= \text{(using L'Hôpital's Rule) } \lim_{n\to\infty} 1/((\lg e)/n)$$
$$= \lim_{n\to\infty} n/(\lg e) = \infty$$

Because this limit is infinite, we know that $n^2 \notin O(n \lg n)$, that is, we know that n^2 grows faster than $n \lg n$. It is easy to use L'Hôpital's Rule to show that $n \lg n \in O(n^2)$, however.

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. Similarly, we determined that for the maximum-finding algorithm, $C(n) = n-1$. This function is also in $O(n)$, so this is also a linear algorithm. We will soon see algorithms whose complexity is in sets with higher orders of growth.

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.

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?

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 $2^n \in O(n)$.
4. Show that n lg $n \in O(n^2)$.
5. Show that if $a, b \geq 0$ and $a \leq b$, then $n^a \in O(n^b)$.

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 $\lim_{n->\infty}(\lg n)^2/n$ is. Using L'Hôpital's Rule twice to differentiate the numerators and denominators, we get

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

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

13: Basic Sorting Algorithms

Introduction

Sorting is one of the most fundamental and important data processing tasks.

> **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 (arrays in the sample code), and the basic operation counted is comparison of list elements (also called *keys*).

Bubble Sort

One of the oldest sorting algorithms is Bubble sort. The idea behind the sort 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 Ruby implementation of Bubble sort appears in Figure 1.

```
def bubble_sort(array)
  (array.size-1).downto(1).each do | j |
    1.upto(j).each do | i |
      if array[i] < array[i-1]
        array[i], array[i-1] = array[i-1], array[i]
      end
    end
  end
  return array
end
```

Figure 1: Bubble Sort

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

On the first pass through the data, every element in the array but the first is compared with its predecessor, 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, $n(n\text{-}1)/2 \in O(n^2)$.

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 $O(n^2)$, however, so this is not much of an improvement.

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. Thus, the algorithm works by repeatedly "selecting" the item that goes at the end of the unsorted portion of the list.

A Ruby implementation of Selection sort appears in Figure 2.

```
def selection_sort(array)
  0.upto(array.size-2).each do | j |
    min_index = j
    (j+1).upto(array.size-1).each do | i |
      min_index = i if array[i] < array[min_index]
    end
    array[j], array[min_index] = array[min_index], array[j]
  end
  return array
end
```

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 basic 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

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

As noted before, $n(n\text{-}1)/2 \in O(n^2)$.

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.

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 Ruby implementation of Insertion sort appears in Figure 3.

```
def insertion_sort(array)
  1.upto(array.size-1).each do | j |
    element = array[j]
    i = j
    while 0 < i && element < array[i-1]
      array[i] = array[i-1]
      i -= 1
    end
    array[i] = element
  end
  return array
end
```

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 where the search is done to find the right place to make the insertion. Once it 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

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

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.

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

The quantity $j/(j+1)$ is always less than one, and 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:

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

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 $O(n^2)$, 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 $O(n^2)$ 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. 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 $O(n)$, that is, Insertion sort always sorts a nearly sorted list in linear time, which is very fast indeed.

Although our analysis of Insertion sort only counted comparisons of keys, there are also many comparisons of indices. We can lessen this overhead cost and speed up the algorithm quite a bit by placing a sentinel value at the start of the list.

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

Extreme values can sometimes be found to use for sentinels, but often using the minimum or maximum value in a data structure, as we do here, works well.

In this case the sentinel value needed is the smallest value in the list. By placing it in the first location, the algorithm need not check the index marching backwards through the list to make sure that it does not go beyond the start of the list—comparison with the sentinel value will always fail, obviating the index check. Ruby code for this revised version of Insertion sort appears in Figure 4.

```
def sentinel_insertion_sort(array)
   # first put the minimum value in location 0
   min_index = 0;
   1.upto(array.size-1).each do | index |
     min_index = index if array[index] < array[min_index]
   end
   array[0], array[min_index] = array[min_index], array[0]

   # now insert elements into the sorted portion
   2.upto(array.size-1).each do | j |
     element = array[j]
     i = j
     while element < array[i-1]
       array[i] = array[i-1]
       i -= 1
     end
     array[i] = element
   end
   return array
 end
```

Figure 4: Insertion Sort with a Sentinel Value

Insertion sort with a sentinel value still has average and worst case complexity in $O(n^2)$, best case complexity in $O(n)$, and it still sorts nearly sorted lists in linear time, but it is perhaps 40% faster than the unmodified Insertion sort algorithm most of the time.

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 Ruby appears in Figure 5.

```
def shell_sort(array)
  # compute the starting value of h
  h = 1;
  h = 3*h + 1 while h < a.size/9

  # insertion sort using decreasing values of h
  while 0 < h do
    h.upto(array.size-1).each do | j |
      element = array[j]
      i = j
      while 0 < i && element < array[i-h]
        array[i] = array[i-h]
        i -= h
      end
      array[i] = element
    end
    h /= 3
  end
  return array
end
```

Figure 5: 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.

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.

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. What is a sentinel value?
4. Would a sentinel value be useful in Shell sort?

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 $W(n) = n(n\text{-}1)$. Why is this result different from $W(n) = n(n\text{-}1)/2$, which we calculated elsewhere?
5. Rewrite Shell sort using a sentinel value to avoid having to check to make sure that invalid array accesses are not attempted.
6. The sorting algorithms presented in this chapter are written for ease of analysis and do not take advantage of all the features of Ruby. Rewrite the sorting algorithms using as many features of Ruby as possible to shorten the algorithms or make them faster.
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?

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 swap one element into place on each pass. Bubble sort makes $O(n^2)$ swaps on average, while Selection sort $O(n)$ swaps in all cases; had we chosen swaps as a basic operation, this difference would have been reflected in our analysis.
3. A sentinel value is a special data value placed in a data structure to mark a boundary.
4. Because Shell sort is abased on Insertion sort, and a sentinel value slightly speeds up Insertion sort, it is reasonable to assume that using a sentinel value in Shell sort would speed it up a bit. Unfortunately, sentinels must be found for each of the sub-lists, which requires many passes over the data, and actually slows down Shell sort slightly.

14: Recurrences

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.

Setting Up Recurrences

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

```
def reverse(s)
  return s if s.size <= 1
  return reverse(s[1..-1]) + s[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 parameter. 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 basic 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.

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

Lets consider a slightly more complex example. The Towers of Hanoi puzzle is a famous game in which one must transfer a pyramidal stack 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.

```
def move_tower(src, dst, aux, n)
  if n == 1
    move_disk(src,dst)
  else {
    move_tower(src, aux, dst, n-1)
    move_disk(src, dst)
    move_tower(aux, dst, src, n-1)
  end
end
```

Figure 2: Towers of Hanoi Algorithm

The last parameter of the `move_tower()` operation is the number of disks to move from the `src` to the `dst` tower. To solve the problem, one calls

```
move_tower(src, dst, aux, src.size)
```

Our measure of the size of the input is the size of the source tower. The algorithm always does the same thing, of course, for a given value of n, so we compute basic 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 pegs from the auxiliary to the destination peg. This gives the following recurrence.

$$C(1) = 1 \qquad \text{for } n = 1$$
$$C(n) = 1 + 2 \cdot C(n\text{-}1) \qquad \text{for } n > 1$$

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.

Solving Recurrences

There are several ways to solve recurrences, but we will consider only one; it is 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 recurrence relation.
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()` operation 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:

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

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

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 disk moves made to solve the Towers of Hanoi puzzle for a source tower of size n is $2^n - 1$, which is obviously in $O(2^n)$.

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. Of course, 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.

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?

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) $C(n) = n + C(n/2)$, $C(1) = 1$

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(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 recurrence relation, (4) substitute the value for i and simplify to obtain a closed form for the recurrence.

15: Merge sort and Quicksort

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.

Merge Sort

Merge sort is a classic example of 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 less), so this algorithm performs well. A Ruby implementation of merge sort appears in Figure 1 below.

```
def merge_sort(array)
  merge_into(array.dup, array, 0, array.size)
  return array
end

def merge_into(src, dst, lo, hi)
  return if hi-lo < 2
  m = (lo+hi)/2
  merge_into(dst, src, lo, m)
  merge_into(dst, src, m, hi)
  j = lo; k = m
  (lo..hi-1).each do | i |
    if j < m && k < hi
      if src[j] < src[k]
        dst[i] = src[j]; j += 1
      else
        dst[i] = src[k]; k += 1
      end
    elsif j < m
      dst[i] = src[j]; j += 1
    else # k < lo
      dst[i] = src[k]; k += 1
    end
  end
end
```

Figure 1: Merge Sort

Merging requires a place to store the result of merging two lists, and duplicating the original list provides space for merging. Hence this algorithm duplicates the original list and passes the duplicate to `merge_into()`. This operation recursively sorts the two halves of the auxiliary list and then merges them back into the original list. 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 $O(n \lg n)$.

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:

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

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

$$
\begin{aligned}
W(n) &= n\text{-}1 + n\text{-}2 + n\text{-}4 + \ldots + n\text{-}2^{i-1} + 2^i \cdot W(n/2^i) \\
&= n\text{-}1 + n\text{-}2 + n\text{-}4 + \ldots + n\text{-}2^{\lg n - 1} \\
&= \sum_{j = 0 \text{ to } \lg n - 1} n - 2^j \\
&= \sum_{j = 0 \text{ to } \lg n - 1} n - \sum_{j = 0 \text{ to } \lg n - 1} 2^j \\
&= n \sum_{j = 0 \text{ to } \lg n - 1} 1 - (2^{\lg n - 1 + 1} - 1) \\
&= n \lg n - n + 1
\end{aligned}
$$

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

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, the elements chosen from one list will be about the same as the elements chosen 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 $O(n \lg n)$ time, which means that it is significantly faster than the other sorts we have seen so far. Its major drawback is that it uses $O(n)$ extra memory locations to do its work.

Quicksort

The most widely studied, widely used (by professionals), and fastest of all algorithms that sort by comparison of keys is quicksort, invented by C. A. R. Hoare in 1960. A Ruby implementation of quicksort appears in Figure 2 below.

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 less, 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.

```
def quick(array, lb, ub)
  return if ub <= lb
  pivot = array[ub]
  i, j = lb-1, ub
  loop do
    loop do i += 1; break if pivot <= array[i]; end
    loop do j -= 1; break if j <= lb || array[j] <= pivot; end
    array[i], array[j] = array[j], array[i]
    break if j <= i
  end
  array[j], array[i], array[ub] = array[i], pivot, array[j]
  quick(array,lb,i-1)
  quick(array,i+1,ub)
end

def quicksort(array)
  quick(array, 0, array.size-1)
  return array
end
```

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:

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

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

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

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

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 $O(n^2)$!

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

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, but perhaps the best 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 below. 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 as well. From now on, we will assume that quicksort includes 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.

Even without these further refinements, empirical studies have shown that quicksort is, on average, about twice as fast as any other $O(n \lg n)$ sorting algorithm. This is because there is minimal overhead in the partitioning algorithm loops, and because quicksort does relatively few swaps (look at all the assignments that merge sort does by comparison, for example).

```
def quick_m3(array, lb, ub)
  return if ub <= lb

  # find sentinels and the median for the pivot
  m = (lb+ub)/2
  array[lb],array[m]=array[m],array[lb] if array[m] < array[lb]
  array[m],array[ub]=array[ub],array[m] if array[ub] < array[m]
  array[lb],array[m]=array[m],array[lb] if array[m] < array[lb]

  # if the sub-array is size 3 or less, it is now sorted
  return if ub-lb < 3

  # put the median just shy of the end of the list
  array[ub-1], array[m] = array[m], array[ub-1]

  pivot = array[ub-1]
  i, j = lb, ub-1
  loop do
    loop do i += 1; break if pivot <= array[i]; end
    loop do j -= 1; break if j <= lb || array[j] <= pivot; end
    array[i], array[j] = array[j], array[i]
    break if j <= i
  end
  array[j], array[i], array[ub-1] = array[i], pivot, array[j]
  quick_m3(array,lb,i-1)
  quick_m3(array,i+1,ub)
end
private :quick_m3

def quicksort_m3(array)
  quick_m3(array, 0, array.size-1)
  return array
end
```

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

Summary and Conclusion

Merge sort is a fast sorting algorithm whose best, worst, and average case complexity are all in $O(n \lg n)$, but unfortunately it uses $O(n)$ extra space to do its work. Quicksort has best and average case complexity in $O(n \lg n)$, but unfortunately its worst case complexity is in $O(n^2)$. The median-of-three improvement makes quicksort's worst case behavior extremely unlikely, however, and quicksort's unmatched speed in practice make it the sorting algorithm of choice when sorting by comparison of keys.

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 so widely?

Exercises

1. Explain why the merge sort algorithm first copies the original list 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. The merge sort algorithm presented above does not take advantage of all the features of Ruby. Write a version of merge sort that uses features of Ruby to make it simpler or faster than the version in the text. Time your algorithm against the one in the text to determine which is faster.
4. 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.
5. 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.
6. The quicksort algorithm presented above does not take advantage of all the features of Ruby. Write a version of quicksort that takes advantage of Ruby's features to make the algorithm shorter or faster. Time your version against the one in the text to determine which one is faster.
7. Write the fastest quicksort you can. Time your implementation against the unmodified quicksort and summarize the results.

8. The Ruby `Array` class has a `sort()` method that uses a version of the quicksort algorithm. If you time any of the quicksort algorithms we have presented, or one you write yourself, against the `Array.sort()` method, you will find that the latter is much faster. Why is this so?

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 widely 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. Innovations like Introspective sort (see exercise 6) can even deal effectively with quicksort's worst case behavior without sacrificing its amazing speed.

16: Trees, Heaps, and Heapsort

Introduction

Trees are the basis for several important data types and data structures. There are also 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.

Basic Terminology

A trees is a special type of graph.

> **Graph**: A collection of *vertices*, or *nodes*, and *edges* connecting the nodes. An edge may be thought of 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 nodes and edges descending below it. Figure 1 illustrates a tree.

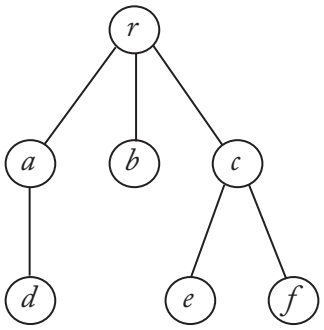


Figure 1: A Tree

Node *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 *descendents* of a node are all the nodes of which it is an ancestor. Thus node *f* has ancestors *f, c*, and *r*, and *c* has descendents *c*, *e*, and *f*. A vertex without children is a *terminal node* or a *leaf*; those with children are *non-terminal* nodes or *internal* nodes. The tree in Figure 1 has three internal nodes (*r*, *a*, and *c*), and four leaf nodes (*b*, *d*, *e*, and *f*). The graph consisting of a vertex in a tree, all its descendents, and the edges connecting them, is a *subtree* of the tree.

A graph consisting of several trees is a *forest*. The *level* of a node in a tree is the number of nodes in the path from the node to the root, not including itself. In Figure 1, node *r* is at level 0, nodes *a*, *b*, and *c* are at level 1, and nodes *d*, *e*, and *f* are at level 2. The *height* of a tree is the maximum level in the tree. The height of the tree in Figure 1 is 2.

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

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 subtree whose root is the left (right) child of a vertex is the *left (right) subtree* of that vertex.

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

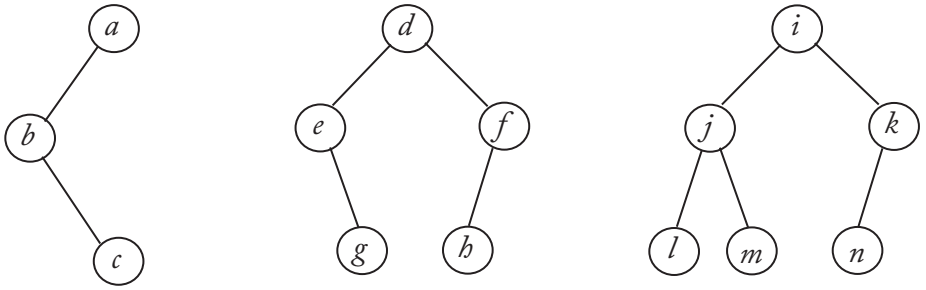


Figure 2: Binary Trees

The trees in Figure 2 are binary trees. In the tree on the left, node *a* has a left child, node *b* has a right child, and node *c* has no children. This tree is neither full nor complete. The middle tree is full but not complete, and the right tree is complete.

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

- A tree with n nodes has n-1 edges.
- A complete binary tree with n internal nodes has either n or n+1 leaves.
- The height of a full binary tree with n nodes is floor(lg n).

Heaps

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

> **Heap**: A complete binary tree whose every node has the heap-order property.

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

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

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

It is fairly efficient to make complete binary trees into heaps because each subtree 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 floor(lg *n*), 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. Consider the numbers assigned to the nodes 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.

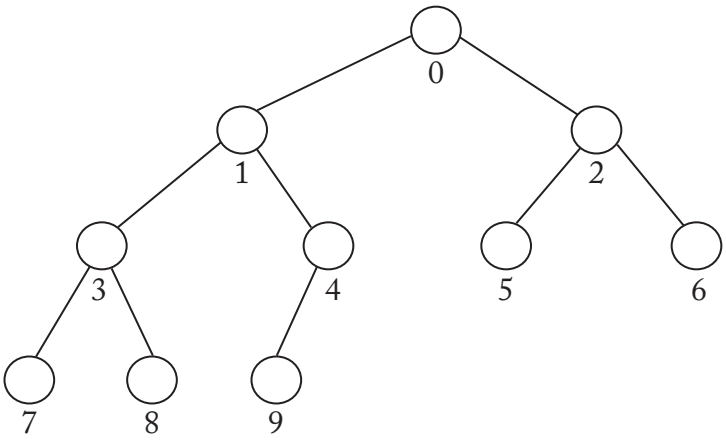


Figure 3: Numbering Nodes for Contiguous Storage

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

- The left child of node *k* is node 2*k*+1.
- The right child of node *k* is node 2*k*+2.
- The parent node k is node floor((*k*-1)/2).
- If there are *n* nodes in the tree, the last one with a child is node floor(*n*/2) - 1.

Now, if we let these node numbers be array indices, then each array location is associated with a node in the tree, and we can store the values at the nodes of the tree in the array: the value of node *k* is stored in array location *k*. The correspondence between array indices and node 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 node, so no space is unused in the array.

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 be 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 the process again and again until the entire array is sorted.

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

```
def heapify(array, i, max_index)
  tmp = array[i]
  j = 2*i + 1
  while j <= max_index
    j += 1 if j < max_index && array[j] < array[j+1]
    break if array[j] <= tmp
    array[i] = array[j]
    i, j = j, 2*i + 1
  end
  array[i] = tmp
end

def heap_sort(array)
  # make the entire array into a heap
  max_index = array.size-1
  ((max_index-1)/2).downto(0).each do | i |
    heapify(array,i,max_index)
  end

  # repeatedly remove the root and remake the heap
  loop do
    array[0],array[max_index] = array[max_index],array[0]
    max_index -= 1
    break if max_index <= 0
    heapify(array, 0, max_index)
  end
  return array
end
```

Figure 4: Heapsort

A somewhat 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 $O(n \lg n)$. Thus heapsort joins 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 merge sort, with the advantage that it does not use any extra space, and it does not have bad worst case complexity.

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 $O(n \lg n)$ time. This algorithm is called heapsort, and is a good algorithm to use when space is at a premium and respectable worst case complexity is required.

Review Questions

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

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 nodes.
3. Draw a complete binary tree with 12 nodes, placing arbitrary values at the nodes. 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 the node is less than or equal to the values stored at its descendents. 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 `heapify()` operation is applied to nodes starting at `max_index-1`. Why does the algorithm not start at `max_index`?
6. Draw a complete binary tree with 12 node, placing arbitrary values at the nodes. 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), it switches to heapsort to sort sub-lists. This algorithm does $O(n \lg n)$ comparisons even in the worst case, sorts in place, and usually runs almost as fast as quicksort on average. Write an introspective sort operation, time your implementation against standard quicksort, and summarize the results.

Review Question Answers

1. In Figure 1 the descendents of *r* are all the nodes in the tree. Node *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. Not every full binary tree is complete because all the leaves of a tree might be on two levels, making it full, but some of the missing leaves at the bottom level might not be on the right, meaning that it is not complete. Every complete binary tree must be a full binary tree, 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 node 27 is node (2*27)+1= 55, the right children of node 27 is node (2*27)+2 = 56, and the parent of node 27 is node floor((27-1)/2) = 13.
6. The worst, best, and average case behavior of heapsort is in $O(n \lg n)$.

17: Binary Trees

Introduction

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

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 nodes hold a value of type *T*. The carrier set thus includes the empty tree, the trees with only a root holding a value of type *T*, the trees with a root and a left child, the trees with a root and a right child, and so forth. Operations in this ADT include the following.

size(*t*)—Return the number of nodes in the tree *t*.

height(*t*)—Return the height of tree *t*.

empty?(*t*)—Return true just in case *t* is the empty tree.

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

buildTree(v,t_l,t_r)—Create and return a new binary tree whose root holds the value *v* and whose left and right subtrees are t_l and t_r.

emptyTree()—Return the empty binary tree.

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

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

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

This ADT allows us to create arbitrary binary trees and examine them. For example, consider the binary tree in Figure 1.

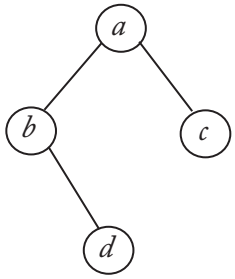


Figure 1: A Binary Tree

This tree can be constructed using the expression below.

buildTree(*a*,
    buildTree(*b*,
        emptyTree(),
        buildTree(*d*,
            emptyTree(),
            emptyTree()))),
    buildTree(*c*,
        emptyTree(),
        emptyTree()))

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

rootValue(*rightSubtree*(*leftSubtree*(*t*)))

As with the ADTs we have studied before, an object-oriented implementation of these operations as instance methods will include the tree as an implicit parameter, so the signatures of these operations vary somewhat when they are implemented. Furthermore, there are several operations that are very useful for a binary tree implementation that are not present in the ADT, and several operations in the ADT that are not needed (more about this below).

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 collections, not as collections in their own right. Usually clients are interested in using basic Collection 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 in the container hierarchy to make both contiguous and linked implementations for all interfaces would create a class 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 does not implement the Collection interface, though it is convenient for it to have several of standard collection operations. It also includes operations for creating and traversing trees in various ways, as well as several kinds of iterators. The Binary Tree class is pictured in Figure 2.

Note that there is no *buildTree*() operation and no *emptyTree*() operation 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.

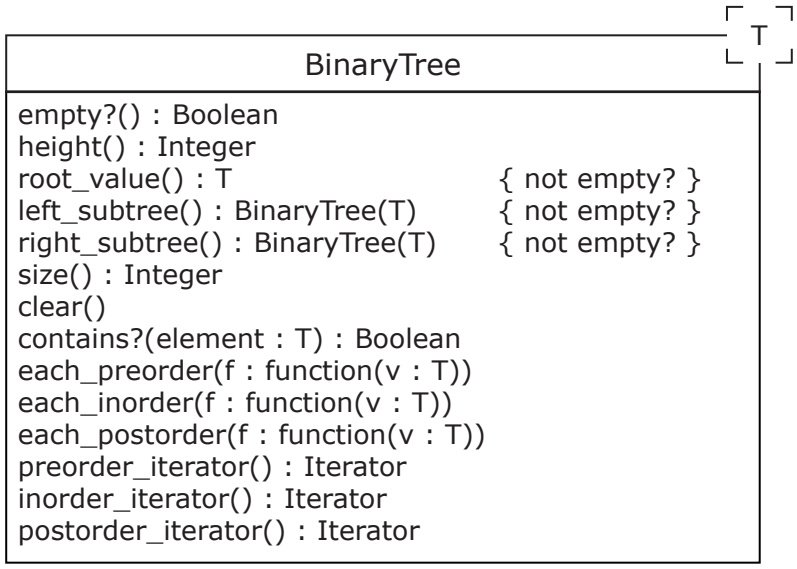


Figure 2: The BinaryTree Class

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

> **Preorder**: When the nodes of a binary tree are visited in preorder, the root node 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 nodes of a binary tree are visited inorder, the left sub-tree (if any) is visited inorder, then the root node is visited, then the right sub-tree is visited inorder.
>
> **Postorder**: When the nodes 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 node 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 nodes in the order *m, b, p, k, t, d, a, g, c, f, h*. A preorder traversal visits the nodes in the order *d, b, m, k, p, t, c, a, g, f, h*. A postorder traversal visits the nodes in the order *m, p, t, k, b, g, a, h, f, c, d*.

The BinaryTree class has internal iterators for visiting the nodes of the tree in the three orders listed above and applying the function passed in as an argument to each node of the tree. For examples, suppose that a print(v : T) operation prints the value v. If t is a binary tree, then the call t.each_inorder(print) will cause the values in the tree t to be printed out inorder, the call t.each_preorder(print) will cause them to be printed in preorder, and the call t.each_postorder(print) will cause them to be printed in postorder.

In addition, the BinaryTree class has three operations that return external iterators that provide access to the values in the tree in each of the three orders above.

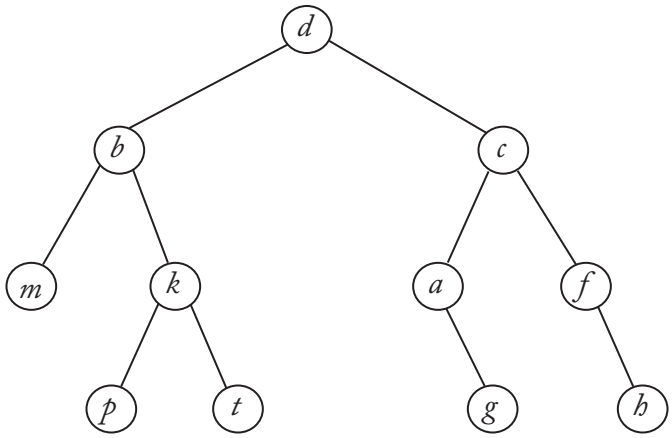


Figure 3: A Binary Tree

When implementing the BinaryTree class in Ruby, it will mix in `Enumerator`, and the `Enumerator.each()` operation will be an alias for the each_inorder(), which is the most common way to visit the nodes of a binary tree.

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 pointers, 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 full, there is still the problem of having to predict the size of the tree ahead of time so that an array could be allocated that is big enough to hold all the tree nodes. 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 in our container hierarchy, or even to build one as an implementation mechanism used to implement other collections. Instead we will implement our BinaryTree class as a linked data structure, and use it as the linked structure implementation mechanism for several of the collections we will add to our container hierarchy later on.

Linked Implementation of Binary Trees

A linked implementation of binary trees resembles implementations of other ADTs using linked structures. Binary tree nodes are represented by objects of a BinaryTreeNode class that has attributes for the data held at the node and references to the left and right subtrees. Each BinaryTree object holds a reference of type BinaryTreeNode to the tree's root node. Empty trees are represented as null references. Figure 4 illustrates this approach.

Trees are inherently recursive structures, so it is natural to write many BinaryTree class operations recursively. For example, to implement the size() operation, the BinaryTree can call an internal size(r : BinaryTreeNode) operation on the root node. This operation returns zero if its parameter is null, and one plus the sum of recursive calls on the left and right sub-

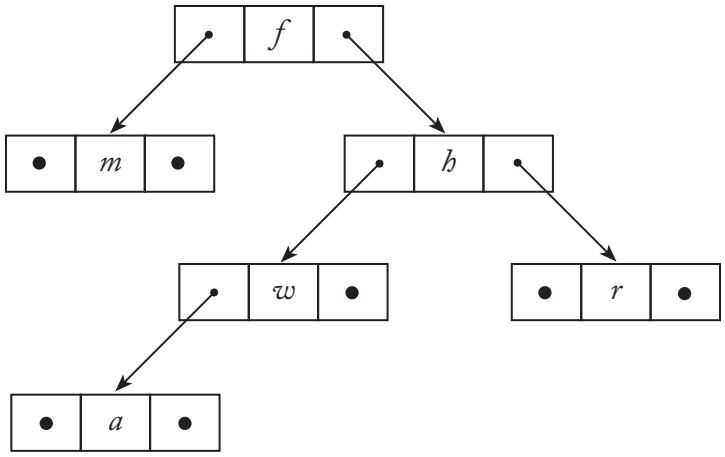


Figure 4: A Linked Representation of a Binary Tree

trees of its parameter node. Many other operations, and particularly the traversal operations that apply functions to the data held at each node, can be implemented just as easily.

Implementing iterators is more challenging, however. The problem is that 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 data structure (like 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 user recursion to implement iterators: use a stack instead.

The second alternative, though harder to do, it clearly better because it uses much less space.

Summary and Conclusion

The binary tree ADT describes basic operations for building and examining binary trees whose nodes hold values of type *T*. A `BinaryTree` class has several operations not in the ADT, in particular, visitor operations for traversing the nodes of the tree and applying a function to the data stored in each node. Iterators are also made available by this class.

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 iterators. The `BinaryTree` class implemented using a linked structure will be used as an implementation mechanism for container classes to come.

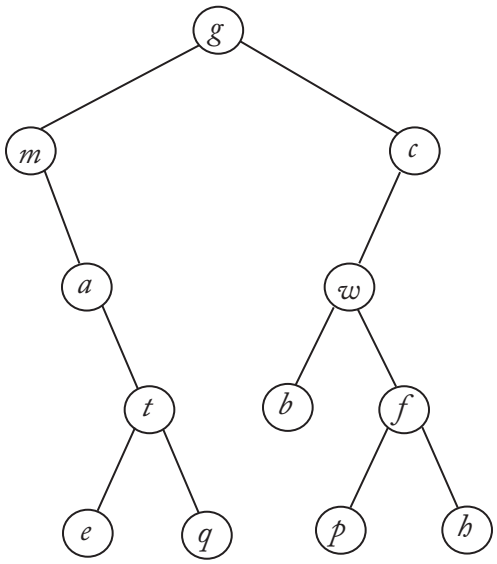
Review Questions

1. Where does the `BinaryTree` class fit in the `Container` class hierarchy?
2. Why does the `BinaryTree` class not include a `buildTree()` operation?

3. Why is the contiguous implementation of binary trees not very useful?
4. What is the relationship between the BinaryTree and BinaryTreeNode classes?

Exercises

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



2. Write the size() operation for BinaryTree class in Ruby.
3. Write the height() operation for the BinaryTree class in Ruby.
4. Write the each_preorder(), each_inorder(), and each_postorder() operations for the BinaryTree class in internal iterator operations in Ruby.
5. Write a PreorderIterator class whose instances are iterators over for the BinaryTree class. The PreorderIterator class will need a stack attribute to hold the nodes that have not yet been visited, with the node holding the current node during iteration at the top of the stack.

Review Question Answers

1. We have decided not to include the BinaryTree class in the Container class 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 buildTree() operation because it may have a constructor that does the very same job.
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. The BinaryTree class has an attribute that stores a reference to the root of the tree, which is a BinaryTreeNode instance. The root node (if any) stores references to the left and right sub-trees of the root, which are also references to instances of the BinaryTreeNode class. Although the BinaryTree and BinaryTreeNode classes are not related by inheritance, they are intimately connected, just as the LinkedList class is closely connected to the LinkedListNode class.

18: Binary Search and Binary Search Trees

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.

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 value 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 coded in Ruby below.

```
def rb_search(array, key)
  return nil if array.empty?
  m = array.size/2;
  return m if key == array[m]
  return rb_search(array[0...m],key) if key < array[m]
  index = rb_search(array[m+1..-1],key)
  index ? m+1+index : nil
end
```

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; in Ruby we have a special value for undefined results, so we return `nil` if the key is not in the array. 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.

The recursion stops when the array is empty and the key has not been found. Otherwise, the element at index `m` in the middle 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 list before or after `m` depending on whether the key is less than or greater than `array[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 made 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 Ruby code in Figure 2 below shows.

```
def binary_search(array, key)
  lb, ub = 0, array.size-1
  while (lb <= ub)
    m = (ub+lb)/2;
    return m if key == array[m]
    if key < array[m]
      ub = m-1
    else
      lb = m+1
    end
  end
  return nil
end
```

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 may take 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 at the middle of the list, doing only 1 comparison. In the worst, case, the key is not in the list, or is found when the sub-list 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 nodes of binary tree are filled 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 out of the list <a, b, c, d, e, f, g, h, i, j, k, l> in the way just described.

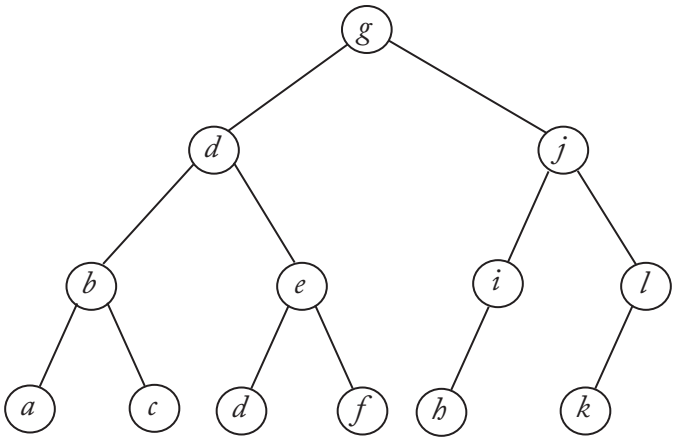


Figure 3: A Binary Tree Made from a List

A tree built this way has the following interesting properties:

- It is a full binary tree, so its height is always floor(lg *n*).
- For every node, every element in its left sub-tree (if any) is less than or equal to the element at the node, and every element in its right sub-tree (if any) is greater than or equal to the element at the node.
- If we traverse the tree inorder, we visit the node elements 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 node from the root to a leaf in the worst case. The number of nodes on these paths is the height of the tree plus one, so $W(n)$ = floor(lg n) + 1. We can also calculate the average case by considering each node equally likely to be the target of a binary search, and figuring out the average length of the path to each node. 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 O(lg n) comparisons, which is very good.

Binary Search Trees

The essential characteristic of the binary tree we looked at above is the relationship between the values at a node 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 node is such that the value at each node 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 data type that retains the property that traversing them in order visits the values in the nodes in sorted order. However, a binary search tree may not be full, so its height may be greater than floor(lg n). In fact, a binary search tree whose every node but one has only a single child will have height n-1.

Binary search trees are interesting because it is fast both to insert elements into them and fast to search them (provided they are not too long and skinny). This contrasts with most

collections, which are usually fast for insertions but slow for searches, or vise versa. For example, elements can be inserted into an (unsorted) LinkedList quickly, but searching a LinkedList is slow, while a (sorted) ArrayList can be searched quickly with binary search, but inserting elements into 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 nodes hold a value of type *T*. It is thus a subset of the carrier set of the binary tree of *T* ADT. The operations in this ADT includes all the operations of the binary tree ADT, with the addition of a precondition on *buildTree()*, shown below. The list below also includes two operations added to the binary search tree ADT.

buildTree(v,t_l,t_r)—Create and return a new binary tree whose root holds the value v and whose left and right subtrees are t_l and t_r. Its precondition is that v is greater than any value held in t_l and less than any value held in t_r.

add(t,v)—Put v into a new node added as a leaf to t, preserving the binary search tree property, and return the resulting binary search tree. If v is already in t, then t is unchanged.

remove(t,v)—Remove the node holding v from t, if any, while preserving the result as a binary search tree, and return the resulting binary search tree.

This ADT is the basis for a BinarySearchTree class.

The Binary Search Tree Class

A BinarySearchTree is a kind of BinaryTree, so the BinarySearchTree class is a sub-class of BinaryTree. Its constructor needs a precondition to make sure that trees are constructed properly. It can also override the contains() operation to be more efficient. Otherwise, it only needs to implement the three operations pictured in Figure 4 below.

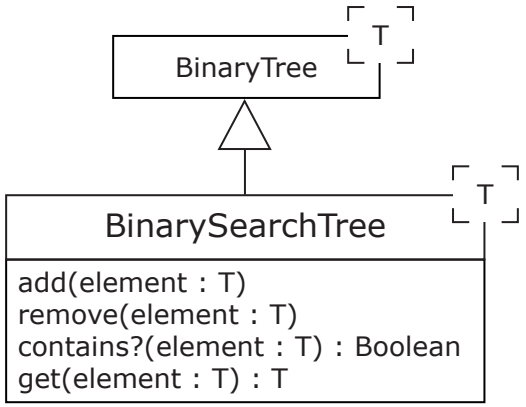


Figure 4: The BinarySearchTree Class

The add() operation puts an element into the tree by making a new child node at a spot that preserves the binary search tree's integrity. If the element is already in the tree, 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 `get()` operation returns the value stored in the tree that is "equal" to the element sent in. 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).

The `contains()` and `get()` operations both search the tree by starting at its root and moving down the tree, mimicking a binary search. If the desired value is at the root (the best case), this requires only one comparison, so $B(n)$ is in $O(1)$. In the worst case, when the tree is effectively a list, this requires $O(n)$ comparisons. Empirical studies have shown that when binary search trees are built by a series of insertions of random data, they are more or less bushy, and their height is not too much more than lg n, so the number of comparisons is in $O(\lg n)$ on average.

The `add()` 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. This again requires $O(1)$ operations in the best case, $O(\lg n)$ comparisons in the average case, and $O(n)$ comparisons in the worst case. Finally, the `remove()` operation must first find the deleted element, and then manipulate the tree to remove the node holding the element in such a way that it is preserved as a binary search tree. This operation also takes $O(1)$ time in the best case, $O(\lg n)$ time in the average case, and $O(n)$ time in the worst case.

Binary search trees thus 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 worst case behavior. We will not have time to study balanced binary search trees.

Summary and Conclusion

Binary search is a very efficient algorithm for searching ordered lists, with average and worst case complexity in $O(\lg n)$. 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.

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 a list 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`?

Exercises

1. A precondition of binary search is that the array searched 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 binary trees?
6. The `BinarySearchTree add()` 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 the `add()` operation for the `BinarySearchTree` class using the strategy explained in the last exercise.
8. Write the `remove()` operation for the `BinarySearchTree` class. This operation 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 rule 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 `remove()` operation using the algorithm above.

Review Question Answers

1. Recursion 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 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 floor(lg 100) = 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, floor(lg 1000000)+1 = 20 comparison would be made. In the average case, roughly floor(lg 1000000) = 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: Sets

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.

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):

$e \in s$—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.

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 the contains?() operation), so they don't appear explicitly in the Set interface.

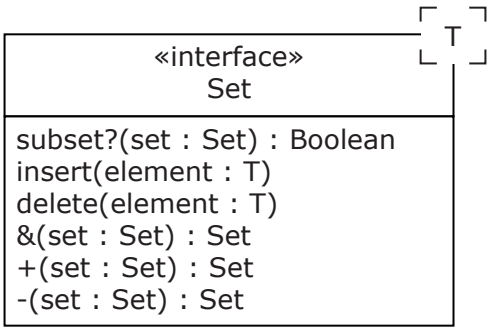


Figure 1: The Set Interface

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 gap for the inserted or deleted element.

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 be used to represents 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.

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 a data type that provides fast insertion, deletion, and membership checking (at least in the best and average cases): binary search trees. Recall that a binary search tree (if fairly well balanced) can be searched in $O(\lg n)$ time, and elements can be inserted and deleted in $O(\lg n)$ time.

An implementation of sets using binary search trees is called a TreeSet. TreeSets are very efficient implementations of sets provided some care is taken to keep them from becoming too unbalanced. TreeSets have the further advantage of allowing iteration over the elements in the set in sorted order, which can be very useful in some applications. As you will see when you do the exercises, all the hard work creating the BinarySearchTree class will not pay off by making it very easy to implement a TreeSet class.

Summary and Conclusion

Sets are useful ADTs that can be implemented efficiently using binary search trees. Figure 2 below shows the portion of the container hierarchy culminating in the TreeSet class, including the Enumerable mixin and the BinarySearchTree class, to illustrate how all these interfaces and classes fit together.

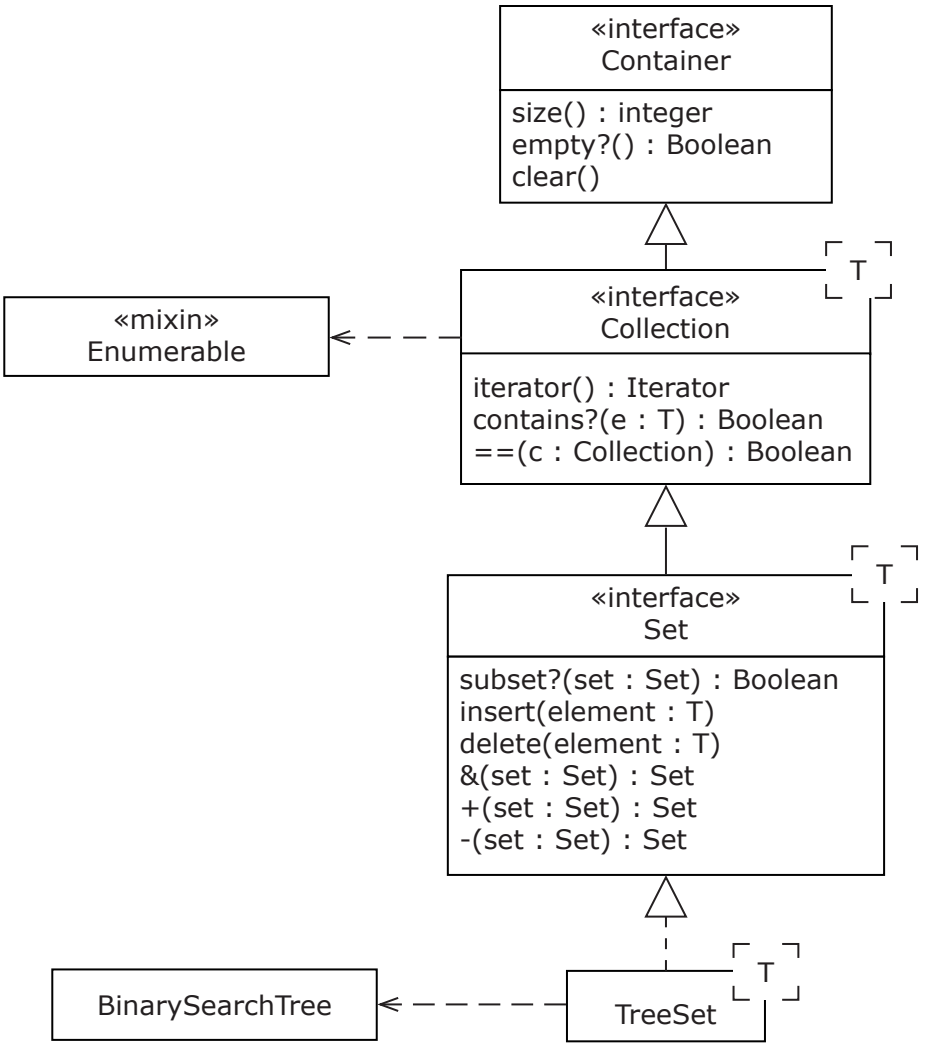


Figure 2: Sets and TreeSets in the Container Hierarchy

Review Questions

1. Which set operations appear in the set ADT but do 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 binary search tree a good data structure for implementing the set ADT?

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. Write the beginnings of a TreeSet class in Ruby that includes its attributes, invariant, constructor, and the operations inherited from the Collection interface.
4. Continue the implementation begun in exercise 3 by writing the subset?() operation for TreeSet.
5. Continue the implementation begun in exercise 3 by writing the insert() and delete() operations for TreeSet.
6. Continue the implementation begun in exercise 3 by writing the +() (union) operation for TreeSet. Note that you need to create and return a brand new TreeSet.
7. Continue the implementation begun in exercise 3 by writing the &() (intersection) operation for TreeSet. Note that you need to create and return a brand new TreeSet.
8. Continue the implementation begun in exercise 3 by writing the -() (relative complement) operation for TreeSet. Note that you need to create and return a brand new TreeSet.

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 me 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 linear 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 binary search tree is a good data structure for implementing the set ADT because a binary search tree allows insertion, deletion, and membership testing to all be done quickly, in $O(\lg n)$ time (provided the tree is kept fairly balanced).

20: Maps

Introduction

A very useful sort of 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 map or 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.

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 various values of types *K* and *T*, the maps with two ordered pairs of values of types *K* and *T*, and so forth.

The essential operations of maps, in addition to those common to all collections, are those for inserting, deleting, searching and retrieving keys and values.

empty?(*m*)—Return true if and only if *m* is the empty map.

size(*m*)—Return the length of map *m*.

has_key?(*m*, *k*)—Return true if and only if map *m* contains an ordered pair whose first element is *k*.

has_value?(*m*, *v*)—Return true if and only if map *m* contains an ordered pair whose second element is *v*.

m[*k*]=*v*—Return a map just like *m* except that the ordered pair <*k*, *v*> has been added to *m*. If *m* already contains an ordered pair whose first element is *k*, then this ordered pair is replaced with the new one.

delete(*m, k*)—Return a map just like *m* except that any ordered pair whose first element is *k* has been removed from *m*. If no such ordered pair is in *m*, then the result is *m* (in other words, if there is no pair with key *k* in *m*, then this operation has no effect).

m[*k*]—Return the second element in the ordered pair whose first element is *k*. Its precondition is that *m* contains an ordered pair whose first value is *k*.

There is considerable similarity between these operations and the operations for lists and sets. For example, the *delete_at*() operation for lists takes a list and an index and removes the element at the designated index, while the map operation takes a map and a key and

removes the key-element pair matching the key argument. 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 element argument that is not in the set.

The Map Interface

The diagram below shows the Map interface, which is a sub-interface of Collection. It includes all the operations of the map of (*K*, *T*) ADT. As usual, the operation parameters are a bit different from those in the ADT because the map is an implicit parameter of all operations. The Map interface is show in Figure 1.

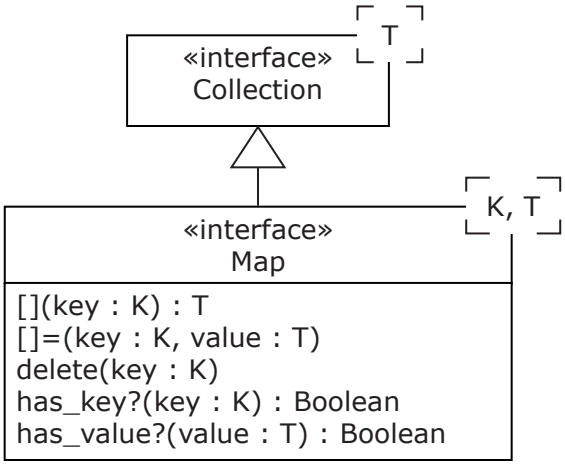


Figure 1: The Map Interface

The contains?() operation inherited from Collection is a synonym for has_value?(). As a Collection, a Map has an associated Iterator (returned by the Collection iterator() operation). In Ruby, it is convenient to have this Iterator traverse the Map and returns its key-value pairs as an array with two elements, the first being the key and the second being the value.

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 addition, 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, and we will discuss how this works later on.

Linked Implementation of the Map ADT

As with sets, using linked lists to store map elements is not much better than using an array. But again as with sets, binary search trees can store map elements and provide fast insertion, deletion, and search operations on keys. Furthermore, using binary 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 TreeMap is thus an excellent implementation of the Map interface.

The trick to using binary search tree to store map elements is to create a class to hold key-value pairs, redefining its relational operators to compare keys, and using this as the datum stored in nodes of the binary search tree. Dummy class instances with the correct key field can then be used to search the tree and to retrieve key-value pairs.

Summary and Conclusion

Maps are extremely important kinds of 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 later). Binary search trees, however, provide very efficient implementations, so a TreeMap class is a good realization of the Map interface. Figure 2 shows how Maps and TreeMaps fit into the container hierarchy.

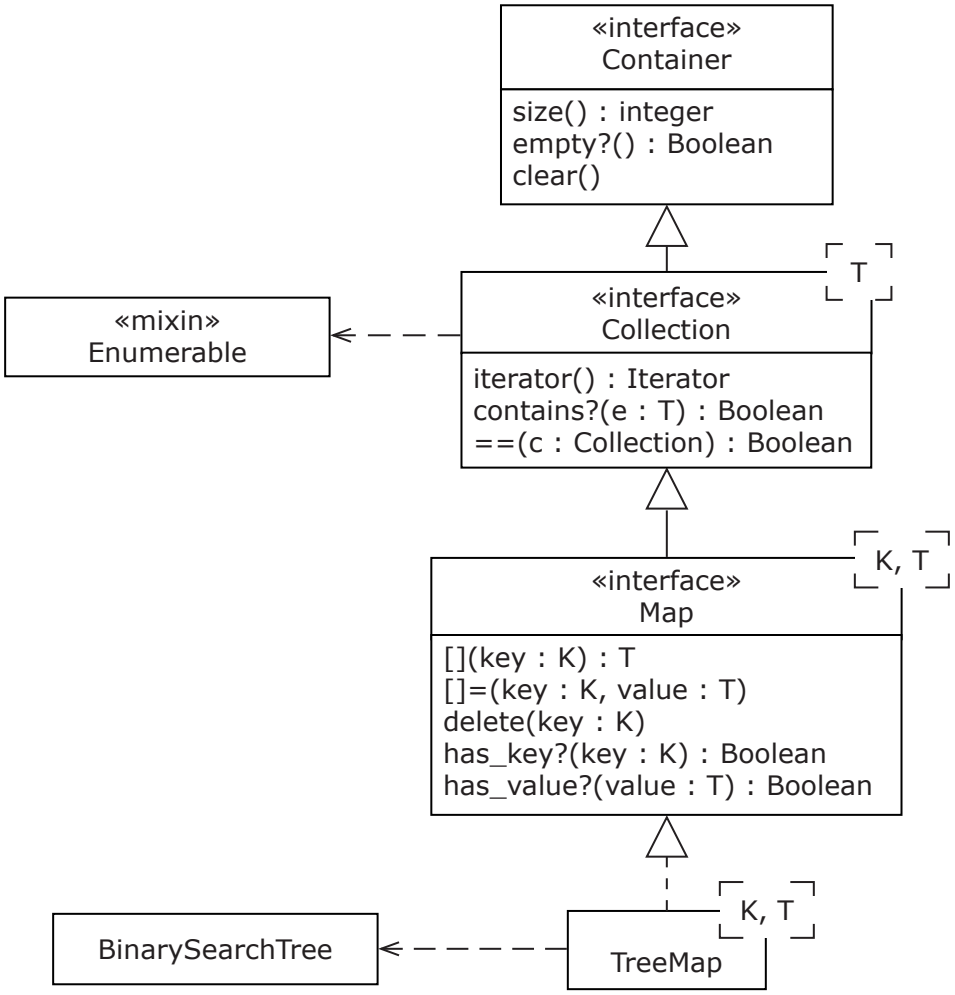


Figure 2: Maps **and** TreeMaps **in the Container Hierarchy**

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 binary search tree a good data structure for implementing the map ADT?

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 values would be the keys and which the elements?
2. Represent the map described in the previous exercise as a set of ordered pairs. If this map is *m*, then also represent as a set of ordered pairs the map that results when the operation *remove*(*m*, Michigan) is applied.
3. Is an iterator required for maps? How does this compare with the situation for lists?
4. To make a TreeMap class that uses the BinarySearchTree class discussed in Chapter 18, you will need to make a class to hold key-value pairs with comparison operations that work on the keys. Write such a Pair class in Ruby.
5. Write the beginnings of a TreeMap class in Ruby that includes its attributes, invariant, constructor, and the operations inherited from the Collection interface. You will need to make use of the Pair class from the previous exercise.
6. Continue the implementation begun in exercise 5 by writing the `[]=`, `[]`, and delete() operations for TreeMap.
7. Continue the implementation begun in exercise 5 by writing the `Enumerable.each()` operation for TreeMap. This operation should produce each key-value pair, yielding the key and value as two elements, just as this operation does for the Ruby `Hash` class. Also write `each_key()` and `each_value()` operations, again modeled on the Ruby `Hash` class.
8. Continue the implementation begun in exercise 5 by writing the iterator() operation (you will need a TreeMapIterator class for TreeMap). The Iterator.current() operation should return a key-value pair as an array with two elements: the key and the value.

Review Question Answers

1. A map associates keys and elements such that each key is associated with at most one element. 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 of 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, and finding them when they need to be removed, is slow because we 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, and finding them when they need to be removed, are slow operations because the LinkedList must be traversed node by node.
5. A binary search tree a good data structure for implementing the map ADT because (assuming that the tree remains fairly balanced), adding key-value pairs, searching for them by key, and removing them by key, are all done very quickly. Furthermore, if the nodes in the tree are traversed in order, then the key-value pairs are accessed in key-order.

21: Hashing

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.

Of course, there are a few details to work out.

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.

Hash Functions

A hash function must transform a key into an integer in the range 0...t, 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.

$$hash(k) = k \bmod t$$

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 pseudocode illustrates a way to hash a string.

```
def hash_function(string, table_size)
     result = 0
     string.each_byte do |byte|
       result = (result * 151 + byte) % table_size
     end
     return result
  end
```

Figure 1: Hash Function for Strings

Thus, making hash functions is not too onerous. A good rule of thumb is 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.

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 elements that go along with them (or, if you like, the key and the value are the same).

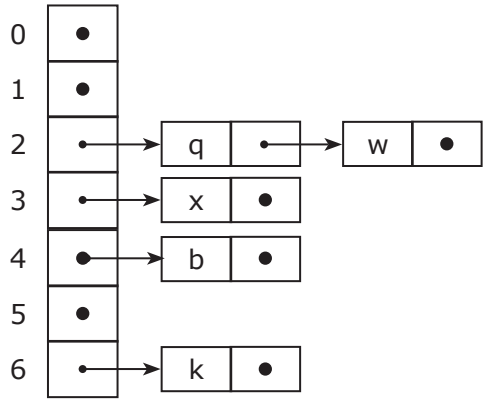


Figure 2: Hash Table with Chaining to Resolve Collisions

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.

When using chaining, the average chain length is λ. If the chain is unordered, successful searches required about 1+ λ/2 comparisons on average, and unsuccessful searches λ comparisons on average. If the chain is ordered, both successful and unsuccessful searches take about 1+ λ/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 his 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.

Theoretical studies of random probe sequences have determined ideal levels of performance for open addressing. Performance declines sharply as the load factor approaches one. For example, with a load factor of 0.9, the number of comparisons for a successful search is about 2.6, and for an unsuccessful search is 20. 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 hashed 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 to use in the probe sequence using a second hash function on 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 $f(x) = x \bmod 7$, and the hash function used to generate a constant for the probe sequence is $g(x) = (x \bmod 5)+1$. The values 8, 12, 9, 6, 25, and 22 are hashed into the table.

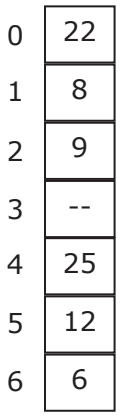


Figure 3: Hash Table with Open Addressing and Double Hashing to Resolve Collisions

The first five keys do not collide. But 22 mod 7 is 1, so 22 collides with 8. The probe constant for double hashing is (22 mod 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) mod 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. Hashing mechanisms that use open addressing must have a way to expand the table, thus lowering the load factor and improving 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.

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, and expanding the table is very expensive.

No matter how hashing is implemented, however, average performance for insertions, deletions, and searches is $O(1)$. 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.

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?

Exercises

1. Why does the example of open addressing and double hashing in the text use the hash function $g(x) = (x \bmod 5)+1$ rather than $g(x) = x \bmod 5$ to generate probe sequences?
2. Suppose a hash table has 11 locations and the simple division method hash function $f(x) = x \bmod 11$ 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?
3. Suppose a hash table has 11 locations, keys are placed in the table using the hash function $f(x) = x \bmod 11$, 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. Suppose a hash table has 11 locations, keys are mapped into the table using the hash function $f(x) = x \bmod 11$, 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.
5. Suppose a hash table has 11 locations, keys are mapped into the table using the hash function $f(x) = x \bmod 11$, and collisions are resolved using double hashing with the hash function $g(x) = (x \bmod 3)+1$ 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.

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 has 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 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 an ordered list of table location that are 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.

22: Hashed Collections

Introduction

As we have seen, hashing provides very fast (effectively $O(1)$) algorithms for inserting, deleting, and searching collections of key-value pairs. It is therefore an excellent way to implement maps. But it also provides a very efficient way to implement sets.

Hash Tablets

A hash table is designed to store key-value pairs by the hash value of the key. But suppose we wanted to store keys alone, with no accompanying value. We can then simply store keys in the hash table by the hash value of the key. We call such a "degenerate" form of hash table a *hash tablet*. Note that several of the hash table examples in the previous chapter about hashing are really hash tablets. Figure 1 shows the features of a HashTablet class.

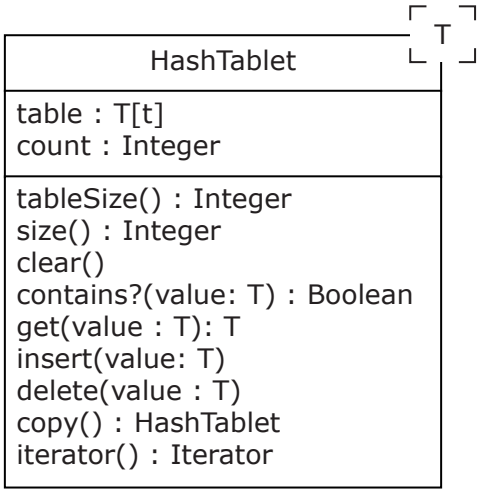


Figure 1: A HashTablet Class

The HashTablet holds a table array into which it hashes values of type T (this table can actually have some other type—for example, if chaining is used, it might be a reference to a node or a node type). The HashTablet allows values to be added, removed, and searched for. The insert() operation replaces the value if it is already in the table. The delete() operation does nothing if the value is not present in the HashTablet. The get() operation returns the value in the table that matches the value argument. This operation seems pointless, but if T is a key-value pair and values are found by comparing keys, the argument to get() may have a dummy value, while the pair returned from the table may contain a legitimate value. This is how values can be retrieved by their keys.

A HashTablet also provides an operation to make a shallow copy of the entire instance (very useful for implementing sets), and it provides an iterator.

HashSets

Hash tablets 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 value we wish to store, but what is the value in the key-value pair? There is none—we must simply ignore this field in the records stored in the hash table, wasting the space. On the other hand, if we use a hash tablet then there is no value field, so no space is wasted. Hash tablets are thus a good way to implement hash sets.

A HashSet implements the Set interface and holds a HashTablet as a private attribute. The basic Collection operations are easily realized using HashTablet operations; for example, a HashSet is empty if its HashTablet's size is 0; clearing a HashSet is done by clearing its HashTablet; iterating over the HashSet is done by iterating over the HashTablet, and so forth.

HashSet union and complement operations may be implemented by copying the host HashSet (and its HashTablet) and iterating over the argument HashSet, adding (for union) or removing (for complement) all its values; intersection is done by iterating over one HashSet and adding to the result HashSet only those values that the other contains.

HashMaps

Hash tablets can be used to implement other collections as well. Consider maps. We can define a Pair class with key and value attributes, and comparison and hash operations that use only the key attribute. We can then implement a map using a hash tablet that stores instances of this Pair class. The hash tablet operations will store, delete, and retrieve instances via the key because the Pair class's operations have been defined to work this way, and the retrieved instances can be queried to obtain the value paired with a key, so all map operations can be implemented. Most of the work will be done by the hash tablet, so it is easy to code the hash map.

A HashMap implements the Map interface and holds a HashTablet as a private attribute. The HashTablet stores instances of a Pair class with a key attribute of type K and a value attribute of type T, and it has hash and comparison operations that work on the key attribute.

Implementing Hashed Collections in Ruby

The Ruby `Object` class includes a `hash()` operation that produces a `Fixnum` value for every object. This value can be modified using the division method for any hash table. Also, the `hash()` function can be overridden to provide appropriate hash values for a `Pair` class, for example.

Ruby has a built-in HashMap class called `Hash`. This class already implements the entire Map interface except the contains?() and iterator() operations. The former is a synonym for `has_value()`, which is in `Hash`, and the latter is easily written. Thus the easiest way to implement a HashMap class in Ruby is to sub-class `Hash`. A HashSet could be implemented in Ruby using a HashTablet as discussed above, or using the built-in `Hash` class. The former

approach will use less space, but the latter will probably be faster because the built-in type is implemented much more efficiently.

Summary and Conclusion

Hashing is an efficient way to implement sets as well as maps. A degenerate form of hash table that we call a hash tablet is a useful implementation data structure that makes implementing hashed collections quite easy.

Review Questions

1. What is the difference between a hash table and a hash tablet?
2. What sorts of collision resolution techniques can be used in a hash tablet?
3. Does a HashSet inherit from HashTablet?
4. Why is a Pair class needed when a HashTablet is used to implement a HashMap?
5. What does the hash_function() of a Pair class do?

Exercises

1. Begin writing a `HashTablet` class in Ruby by writing the attributes, invariant, and `initialize()` operation for the class. This will require that you decide on a collision resolution strategy.
2. Continue writing a `HashTablet` class in Ruby by implementing the `size()`, `tableSize()`, `clear()`, `copy()`, and `iterator()` operations.
3. Finish implementing the `HashTablet` class in Ruby by implementing the remaining operations. Note that values stored in the table must have a hash function that `HashTablet` can use.
4. Assume that a `HashTablet` class is available, and begin implementing a `HashSet` class in Ruby by writing the attributes, invariant, and `initialize()` operation for this class.
5. Continue implementing a `HashSet` class in Ruby by writing the `contains?()`, `[]=`, and `delete()` operations of this class.
6. Using a `HashTablet` to implement a `HashMap` requires that a `Pair` class storing a key and a value be created. Write such a `Pair` class in Ruby, and include a hashing function and comparison operators for the class.
7. Assume that a `HashTablet` class is available, and begin implementing a `HashMap` class in Ruby by writing the attributes, invariant, and `initialize()` operation for this class. You will need to use the `Pair` class from the last exercise.
8. Continue implementing a `HashMap` class by writing the `contains?()`, `[]=`, and `[]` operations of this class.
9. When discussing ArrayList in Chapter 10 we suggested that implementing an ArrayList in Ruby could be done quite simply by sub-classing the built-in Ruby `Array` class. Implement the HashMap class by sub-classing the built-in `Hash` class.

Review Question Answers

1. A hash table stores key-value pairs by hashing the key. A hash tablet stores keys only using hashing. A hash tablet is thus a degenerate or simplified version of a hash table.
2. Any sort of collision resolution techniques can be used in a hash tablet.
3. A `HashSet` does not inherit from `HashTablet`, but it contains a `HashTablet` attribute, and the `HashTablet` ends up doing most of the work when the `HashSet` is implemented. The `HashSet` thus *delegates* work to the `HashTablet`.
4. A `Pair` class is needed when a `HashTablet` is used to implement a `HashMap` because a `HashTablet` only stores one value, not a key-value pair. The `HashTablet` can be made to work as if it stored pairs by defining a `Pair` class and storing instances of the `Pair` class in the `HashTablet`. Thus a degenerate hash table can be made to work like a full-fledged hash table quite simply.
5. The `hash_function()` of a `Pair` class computes a hash table value using the `key` attribute of the `Pair` class. This allows the `HashTablet` to store and retrieve the `Pair` based on the `key`.

Glossary

abstract data type (ADT)—a set of values (the **carrier set**), and operations on those values.

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 many resources (such as time and memory) an algorithm consumes when it executes.

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.

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.

bag—an n unordered collection in which an entity may appear more than once; also called a **multiset.**

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 node is such that the value at the node 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 *right child*. The subtree whose root is the left (right) child of a vertex is the *left (right) subtree* of that vertex.

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 full binary tree whose missing nodes are all at the right of its bottom level.

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.

container—an entity that holds finitely many other entities.

cursor—a value marking a location in a data structure.

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.

dequeue—a dispenser whose elements can be accessed, inserted, or removed only at its ends.

dereferencing—the operation of following the address held by a pointer to obtain a value of its associated type.

dictionary—an unordered collection that allows insertion, deletion, and test of membership.

dispenser—a non-traversable container.

doubly linked list—a linked structure whose nodes each have two pointer fields used to form the nodes into a sequence. Each node but the first has a predecessor link field containing a pointer to the previous node in the list, and each node but the last has a successor link containing a 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).

factory method—an operation of a class that returns a new instance of some class.

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 nodes. An edge may be thought of 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.

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-element pairs whose locations are determined by a hash function.

heap—a complete binary tree whose every node 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.

infix expression—an expression in which the operators appear between their operands.

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.

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 collection.

map—an unordered collection whose elements (called *values*) are accessible using a key; also called a **table**, **dictionary**, or **associative array**.

node—an aggregate variable composed of data and link or pointer fields.

pointer—a derivative type whose carrier set is addresses of values of some associated 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.

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.

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**.

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.

sentinel value—a special value placed in a data structure to mark a boundary.

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 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 node link but the last contains a pointer to the next node in the list; the link filed of the last node contains null.

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 key fields in each record.

stack—a dispenser holding a sequence of elements that can be accessed, inserted, or removed at only one end, called the **top**.

static array—an array whose size is established at program design time and cannot change during execution.

string—a finite sequence of characters drawn from some alphabet.

structured type—a type whose carrier set values are composed of some arrangement of atomic values.

tail recursive—an algorithm is tail recursive when at most one recursive call is made as the last step of each execution of the algorithms 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.

unreachable code assertion—an assertion that is placed at a point in a program that should not be executed under any circumstances.

vector—a finite ordered collection of values of the same type.

worst case complexity $W(n)$—the maximum number of basic operations performed by an algorithm for any input of size n.