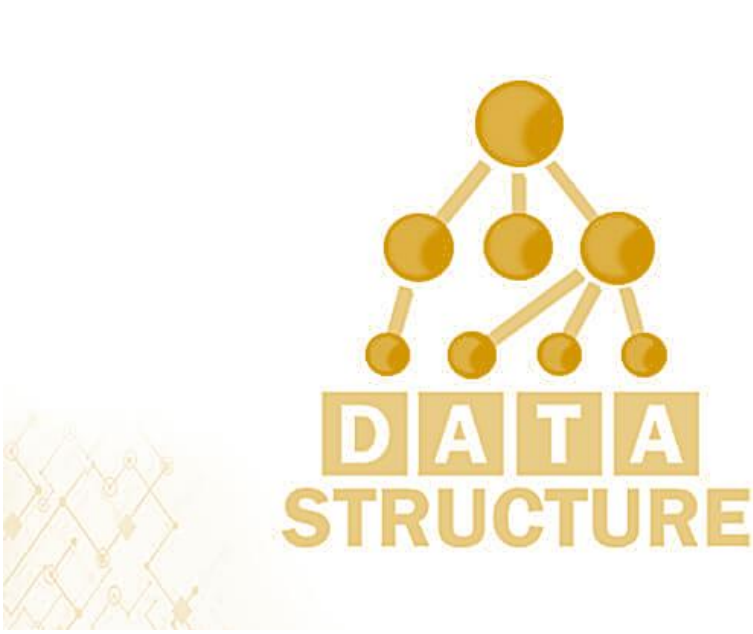
كلية الحاسبات والذكاء الاصطناعي
جامعة سوهاج



LECTURES ON DATA STRUCTURES

Dr.-Ing. Samy Sadek

2022-2023

Chapter -1

Abstraction and Abstract Data Types

Abstraction and Abstract Data Types

Abstraction is the process of trying to identify the most important or inherent qualities of an object or model, and ignoring or omitting the unimportant aspects. It brings to the forefront or highlights certain features, and hides other elements. In computer science we term this process *information hiding*.

Abstraction is used in all sorts of human endeavors. Think of an atlas. If you open an atlas, you will often first see a map of the world. This map will show only the most significant features. For example, it may show the various mountain ranges, the ocean currents, and other extremely large structures. But small features will almost certainly be omitted.

A subsequent map will cover a smaller geographical region, and will typically possess more detail. For example, a map of a single continent (such as South America) may now include political boundaries, and perhaps the major cities. A map over an even smaller region, such as a country, might include towns as well as cities, and smaller geographical features, such as the names of individual mountains. A map

of an individual large city might include the most important roads leading into and out of the city. Maps of smaller regions might even represent individual buildings.

Notice how, at each level, certain information has been included, and certain information has been purposely omitted. There is simply no way to represent all the details when an artifact is viewed at a higher level of abstraction. And even if all the detail could be described (using tiny writing, for example) there is no way that people could assimilate or process such a large amount of information. Hence details are simply left out.

Abstraction is an important means of controlling complexity. When something is viewed at an abstract level only the most important features are being emphasized. The details that are omitted need not be remembered or even recognized.

Another term that we often use in computer science for this process is *encapsulation*. An encapsulation is a packaging, placing items into a unit, or capsule. The key consequence of this process is that the encapsulation can be viewed in two ways, from the inside and from the outside. The outside view is often a description of the task being performed, while the

inside view includes the implementation of the task.

An example of the benefits of abstraction can be seen by imagining calling the function used to compute the square root of a double precision number.

```
double sqrt (double n)
{
  double result = n/2;
  while (… ) {
  …
}
 return result;
}
```

The only information you typically need to know is the name of the function (say, **`sqrt`**), the argument types, and perhaps what it will do in exceptional conditions (say, if you pass it a negative number). The computation of the square root is actually a nontrivial process. The function will probably use some sort of approximation technique, such as Newtons iterative method. But the details of how the result is produced have been abstracted away, or encapsulated within the function boundary, leaving you only the need to understand the description of the desired result.

Programming languages have various different

techniques for encapsulation. The previous paragraph described how functions can be viewed as one approach. The function cleanly separates the outside, which is concerned with the "what" – what is the task to be performed, from the inside, the "how" – how the function produces its result. But there are many other mechanisms that serve similar purposes.

Some languages (but not C++) include the concept of an *interface*. An interface is typically a collection of functions that are united in serving a common purpose. Once again, the interface shows only the function names and argument types (this is termed the function *signature*), and not the bodies, or implementation of these actions.

```
public interface Stack {
    public void push (Object a);
    public Object top ();
    public void pop ();
    public boolean isEmpty ();
};
```

In fact, there might be more than one implementation for a single interface. At a higher level, some languages include features such as modules, or packages. Here, too, the intent is to provide an encapsulation mechanism, so that code that is

outside the package need only know very limited details from the internal code that implements the package.

Interface Files

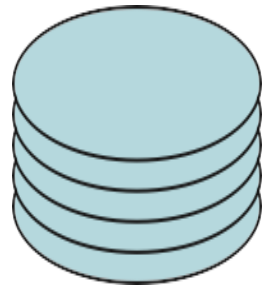
The C language, which we use in this book, has an older and more primitive facility. Programs are typically divided into two types of files. Interface files, which traditionally end with a .h file extension, contain only function prototypes, interface descriptions for individual files. These are matched with an implementation file, which traditionally end with a .c file extension.

Implementation files contain, as the name suggests, implementations of the functions described in the interface files, as well as any supporting functions that are required, but are not part of the public interface. Interface files are also used to describe standard libraries.

Abstract Data Types

The study of data structures is concerned largely with the need to maintain *collections* of values. These are sometimes termed *containers*. Even without discussing how these collections can be implemented, a number of different types of containers can be identified purely by their purpose or

behavior. This type of description is termed an *abstract data type*.



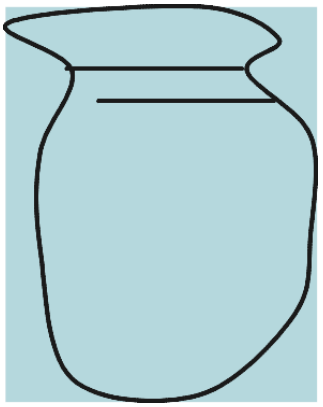
A simple example will illustrate this idea. A *stack* is a collection in which the order that elements are inserted is critically important. A metaphor, such as a stack of plates, helps in envisioning the idea. Only the topmost item in the stack (the topmost plate, for example), is accessible. To second element in the stack can only be accessed by first removing the topmost item. Similarly, when a new item is placed into the collection (a new plate placed on the stack, for example), the former top of the stack is now inaccessible, until the new top is removed.

Notice several aspects of this description. The first is the important part played by *metaphor*. The characteristics of the collection are described by appealing to a common experience with non-computer related examples. The second is that it is the *behavior* that is important in defining the type of collection, not the particular names given to the operations. Eventually the operations will be named, but the names selected (for example, push, add, or insert for placing an item on to the stack) are not what makes the collection into a stack. Finally, in order to be useful, there must eventually be a concrete

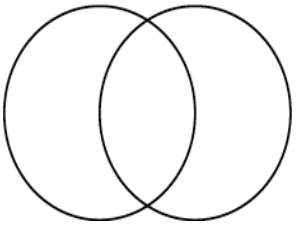
realization, what we term an *implementation*, of the stack behavior. The implementation will, of course, use specific names for the operations that it provides.

The Classic Abstract Data Types

There are several abstract data types that are so common that the study of these collection types is considered to be the heart of a fundamental understanding of computer science. These can be described as follows:



A *bag* is the simplest type of abstraction. A good metaphor is a bag of marbles. Operations on a bag include adding a value to the collection, asking if a specific value is or is not part of the collection, and removing a value from the collection.



A *set* is an extension of a bag. In addition to bag operations the set makes the restriction that no element may appear more than once, and also defines several functions that work with entire sets. An example would be set intersection, which constructs a new set consisting of values that appear in two argument sets. A *Venn* diagram is a

good metaphor for this type of collection.

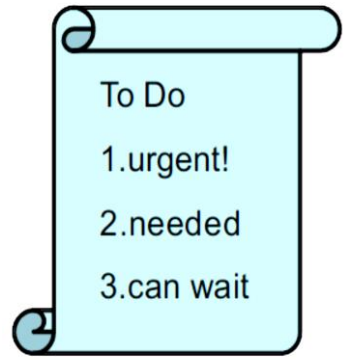
The order that elements are placed into a bag is completely unimportant. That is not true for the next three abstractions. For this reason, these are sometimes termed *linear* collections. The simplest of these is the *stack*. The stack abstraction was described earlier. The defining characteristic of the stack is that it remembers the order that values were placed into the container. Values must be removed in a strict LIFO order (last-in, first-out). A stack of plates is the classic metaphor.

A *queue*, on the other hand, removes values in exactly the same order that they were inserted. This is termed FIFO order (first-in, first-out). A queue of people waiting in line to enter a theater is a useful metaphor.

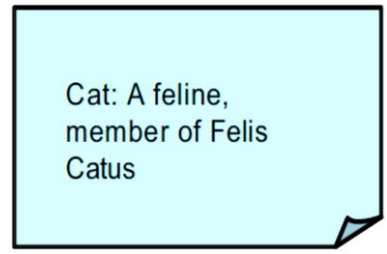
The *deque* combines features of the stack and queue. Elements can be inserted at either end, and removed from either end, but only from the ends. A good mental image of a deque might be placing peas in a straw. They can be inserted at either end, or removed from either end, but it is

not possible to access the peas in the middle without first removing values from the end.

A *priority queue* maintains values in order of importance. A metaphor for a priority queue is a to-do list of tasks waiting to be performed, or a list of patients waiting for an operating room in a hospital. The key feature is that you want to be able to quickly find the most important item, the value with highest priority.



A *map*, or *dictionary*, maintains pairs of elements. Each key is matched to a corresponding value. They keys must be unique. A good metaphor is a dictionary of word/definition pairs.



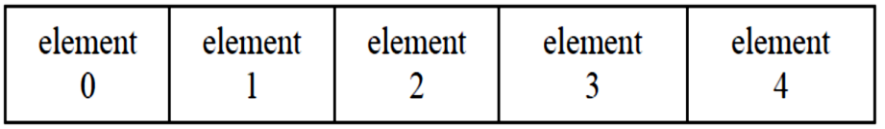
Each of these abstractions will be explored in subsequent chapters, and you will develop several implementations for all of them.

Implementations

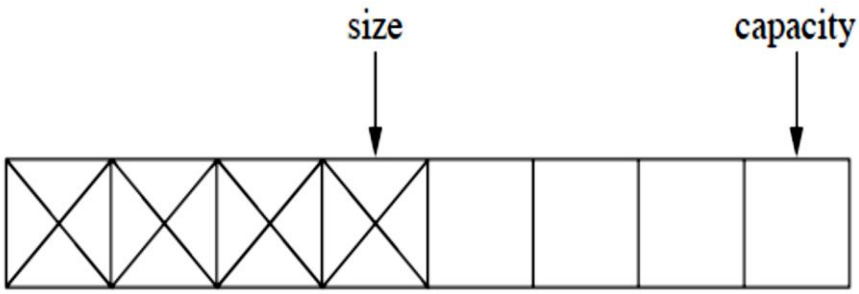
Before a container can be used in a running program it must be matched by an *implementation*. The majority of this book will be devoted to explaining different implementations

techniques for the most common data abstractions. Just as there are only a few classic abstract data types, with many small variations on a common theme, there are only a handful of classic implementation techniques, again with many small variations.

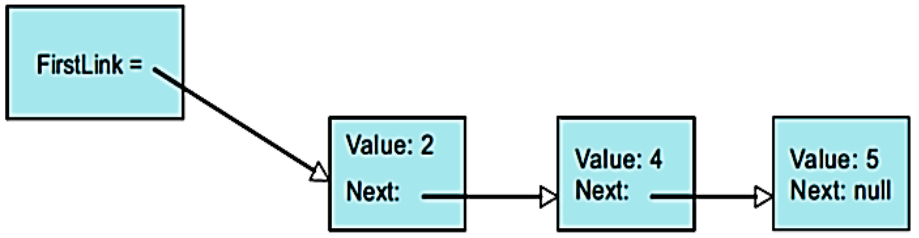
The most basic way to store a collection of values is an *array*. An array is nothing more than a fixed size block of memory, with adjacent cells in memory holding each element in the collection:



A disadvantage of the array is the fixed size, which typically cannot be changed during the lifetime of the container. To overcome this, we can place one level of indirection between the user and the storage. A *dynamic array* stores the size and capacity of a container, and a pointer to an array in which the actual elements are stored. If necessary, the internal array can be increased during the course of execution to allow more elements to be stored. This increase can occur without knowledge of the user. Dynamic arrays are introduced in Worksheet 1, and used in many subsequent worksheets.

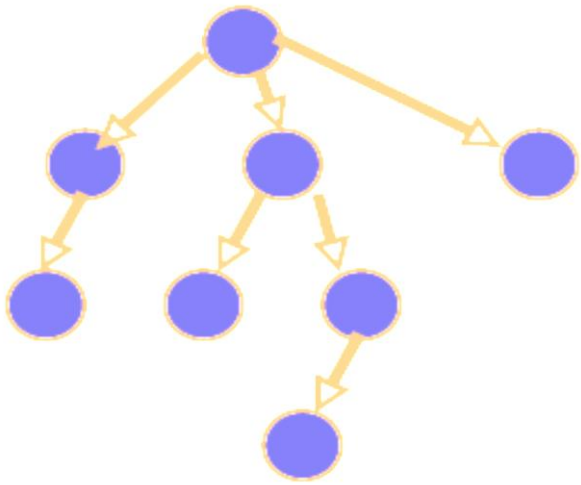


The fact that elements in both the array and the dynamic array are stored in a single block is both an advantage and a disadvantage. When collections remain roughly the same size during their lifetime the array uses only a small amount of memory. However, if collection changes size dramatically then the block can end up being largely unused. An alternative is a *linked list*. In a linked list each element refers to (points to) the next in sequence, and are not necessarily stored in adjacent memory locations.

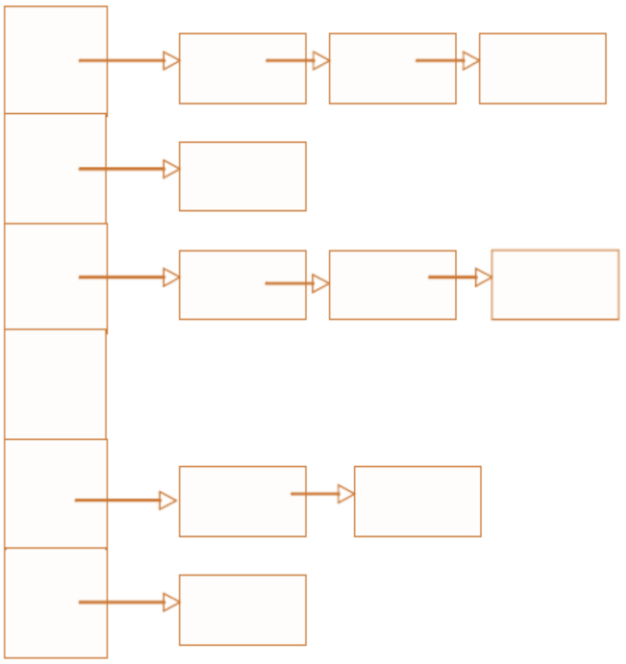


Both the array and the linked list suffer from the fact that they are linear organizations. To search for an element, for example, you examine each value one after another. This can be very slow. One way to speed things up is to use a tree,

specifically a *binary tree*. A search in a binary tree can be performed by moving from the top (the root) to the leaf (the bottom) and can be much faster than looking at each element in turn.



There are even more complicated ways to organize information. A *hash table*, for example, is basically a combination of an array and a linked list. Elements are assigned positions in the array, termed their *bucket*. Each bucket holds a linked list of values. Because each list is relatively small, operations on a hash table can be performed very quickly.



Many more variations on these themes, such as skip lists (a randomized tree structure imposed on a simple linked list), or heaps (a binary tree organized as a priority queue) will be presented as we explore this topic.

Some Words Concerning C

In the first part of this book, we have made little reference to the type of values being held in a collection. Where we have used a data structure, such as the array used in sorting algorithms, the element type has normally been a simple floating-point number (a `double`). In the development that follows we want to generalize our containers

so that they can maintain values of many different types. Unfortunately, C provides only very primitive facilities for doing so.

A major tool we will use is symbolic name replacement provided by the C preprocessor. This facility allows us to define a name and value pair. Prior to compilation, the C preprocessor will systematically replace each occurrence of the name with a value. For example, we will define our element type as follows:

```
# define EleType double
```

Following this definition, we can use the name EleType to represent the type of value our container will hold. This way, the user need only change the one definition in order to modify the type of value a collection can maintain.

Another feature of the preprocessor allows us to make this even easier. The statement #ifndef informs the preprocessor that any text between the statement and a matching #endef statement should only be included if the argument to the ifndef is *n*ot already *def*ined. The definition of TYPE in the interface file will be written as follows

```
# ifndef TYPE
# define TYPE double
# endif
```

These statements tell the preprocessor to only define the name EleType if it has not already been defined. In effect, it makes double into our default value, but allows the user to provide an alternative, by preceding the definition with an alternative definition. If the user wants to define the element type as an integer, for example, they simple precede the above with a line:

```
# define TYPE integer
```

A second feature of C that we make extensive use of in the following chapters is the equivalence between arrays and pointers. In particular, when an array must be allocated dynamically (that is, at run time), it is stored in a pointer variable. The function used to allocate memory is termed malloc. The malloc function takes as argument an integer representing the number of bytes to allocate. The computation of this quantity is made easier by another function, sizeof, which computes the size of its argument type.

You saw an example of the use of malloc and sizeof in the merge sort algorithm described in a basic algorithms course. There the malloc was used to create a temporary array. Here is another example. The following bit of code takes an integer value stored in the variable n, and allocates an array that can hold n elements of whatever type is represented by

EleType. Because the malloc function can return zero if there is not enough memory for the request, the result should always be checked. Because malloc returns an indetermined pointer type, the result must be cast to the correct form.

```
int n;
TYPE * data;
…
n = 42; /* n is given some value */
…
data = (TYPE *) malloc(n * sizeof(TYPE)); /* array of size n is allocated */
assert (data != 0); /* check that allocation worked */
  …
free (data);
```

Dynamically allocated memory must be returned to the memory manager using the free operation. You should always make sure that any memory that is allocated is eventually freed.

This is an idiom you will see repeatedly starting in Worksheet 1. We will be making extensive use of pointers, but treating them as if they were arrays. Pointers in C can be indexed, exactly as if were arrays.

Study questions

1. What is abstraction? Give three examples of abstraction from real life.
2. What is information hiding? How is it related to abstraction?
3. How is encapsulation related to abstraction?
4. Explain how a function can be viewed as a type of encapsulation. What information is being hidden or abstracted away?
5. What makes an ADT description abstract? How is it different from a function signature or an interface?
6. Come up with another example from everyday life that illustrates the behavior of each of the six classic abstractions (bag, stack, queue, deque, priority queue, map).
7. For each of the following situations, describe what type of collection seems most appropriate, and why. Is order important? Is time of insertion important?

    i. The names of students enrolled in a class.
    ii. Files being held until they can be printed on a printer.
    iii. URLs for recently visited web pages in a browser.

iv. Names of patients waiting for an operating room in a hospital emergency ward.

v. Names and associated Employee records in a company database.

8. In what ways is a set similar to a bag? In what ways are they different?
9. In what ways is a priority queue similar to a queue? In what ways are they different?
10. Once you have completed Worksheet 1, answer this question and the ones that follow. What is a dynamic array?
11. What does the term capacity refer to in a dynamic array?
12. What does the term size refer to in a dynamic array?
13. Can you describe the set of legitimate subscript positions in a dynamic array? Is this different from the set of legal positions in an ordinary array?
14. How does the add function in a dynamic array respond if the user enters more values than the current capacity of the array?
15. Suppose the user enters 17 numbers into a dynamic array that was initialized with a capacity of 5? What will be the final length of the data array? How many times will it have

been expanded?

16. What is the algorithmic execution time for the `_dyArrayDoubleCapacity`, if n represents the size of the final data array?

Analysis Exercises

1. Explain, in your own words, how calling a function illustrates the ideas of abstraction and information hiding. Can you think of other programming language features that can also be explained using these ideas?

2. Even without knowing the implementation, you can say that it would be an error to try to perform a *pop* operation on a stack if there has not been a preceding *push*. For each of the classic abstractions describe one or more sequences of actions that should always produce an error.

3. This question builds on the work you began with the preceding question. Without even looking at the code some test cases can be identified from a specification alone, independent of the implementation. As you probably know, this is termed *black box testing*. For example, if you push an item on to a stack, then perform a pop, the item you just pushed should be returned. For each of the classic data structures come up with a set of test cases derived

simply from the description.

4. Contrast an interface description of a container with the ADT description of behavior. In what ways is one more precise than the other? In what ways is it less precise?

5. Once you have completed Worksheet 1 you should be able to answer the following. When using a partially filled array, why do you think the data array is doubled in size when the size exceeds the capacity? Why not simply increase the size of the array by 1, so it can accommodate the single new element? Think about what would happen if, using this alternative approach, the user entered 17 data values. How many times would the array be copied?

Programming Projects

1. In Worksheet 2 you explore why a dynamic array doubles the size of its internal array value when the size must be increased. In this project you can add empirical evidence to support the claim that this is a good idea. Take the dynamic array functions you developed in the worksheets 1 and 2 and add an additional variable to hold the "unit cost". As we described in Worksheet 2, add 1 each time an element is added without reallocation, and add the size of the new array each time a reallocation occurs, plus 1 for

the addition of the new element. Then write a main program that will perform 200 additions, and print the average cost after each insertion. Do the values remain relatively constant?

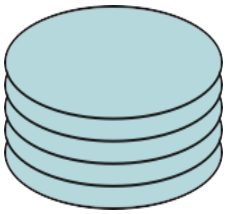
On the Web

Wikipedia (http://en.wikipedia.org/wiki/Main_Page) has a good explanation of the concept of abstract data types. Links from that page explore most of the common ADTs. Another definition of ADT, as well as definitions of various forms of ADTs, can be found on DADS (http://www.nist.gov/dads/) the *Dictionary of Algorithms and Data Structures* maintained by the National Institute of Standards and Technology. Wikipedia has entries for many common C functions, such as malloc. There are many on-line tutorials for the C programming language. A very complete tutorial has been written in Brian Brown, and is mirrored at many sites. You can find this by googling the terms “Brian Brown C programming”.

Chapter -2

Stacks

STACKS



You are familiar with the concept of a *stack* from many everyday examples. For example, you have seen a stack of books on a desk, or a stack of plates in a cafeteria. The common characteristic of these examples is that among the items in the collection, the easiest element to access is the topmost value. In the stack of plates, for instance, the first available plate is the topmost one. In a true stack abstraction that is the *only* item you are allowed to access.

Furthermore, stack operations obey the *last-in, first-out* principle, or LIFO. If you add a new plate to the stack, the previous topmost plate is now inaccessible. It is only after the newly added plate is removed that the previous top of the stack once more becomes available. If you remove all the items from a stack, you will access them in reverse chronological order – the first item you remove will be the item placed on the stack most recently, and the last item will be the value that has been held in the stack for the longest period of time.

Stacks are used in many different types of computer

applications. One example you have probably seen is in a web browser. Almost all web browsers have *Back* and *Forward* buttons that allow the user to move backwards and forwards through a series of web pages. The Back button returns the browser to the previous web page. Click the back button once more, and you return to the page before that, and so on. This works because the browser is maintaining a stack containing links to web pages. Each time you click the back button it removes one link from this stack and displays the indicated page.

The Stack Concept and ADT specification

Suppose we wish to characterize the stack metaphor as an abstract data type. The classic definition includes the following four operations:

`Push (newEntry)`	Place a new element into the collection. The value provided becomes the new topmost item in the collection. Usually there is no output associated with this operation.
`Pop ()`	Remove the topmost item from the stack.
`Top ()`	Returns, but does not remove, the topmost item from the stack.

`isEmpty ()`	Determines whether the stack is empty

Note that the names of the operations do not specify the most important characteristic of a stack, namely the LIFO property that links how elements are added and removed. Furthermore, the names can be changed without destroying the stack-edness of an abstraction. For example, a programmer might choose to use the names add or insert rather than push, or use the names peek or inspect rather than top. Other variations are also common. For example, some implementations of the stack concept combine the pop and top operations by having the pop method return the value that has been removed from the stack. Other implementations keep these two tasks separate, so that the only access to the topmost element is through the function named top. As long as the fundamental LIFO behavior is retained, all these variations can still legitimately be termed a stack.

Finally, there is the question of what to do if a user attempts to apply the stack operations incorrectly. For example, what should be the result if the user tries to pop a value from an empty stack? Any useful implementation must provide some well-defined behavior in this situation. The

most common implementation technique is to throw an exception or an assertion error when this occurs, which is what we will assume. However, some designers choose to return a special value, such as null. Again, this design decision is a secondary issue in the development of the stack abstraction, and whichever design choice is used will not change whether or not the collection is considered to be a stack, as long as the essential LIFO property of the collection is preserved. The following table illustrates stack operations in several common languages:

	Java class Stack	C++ stack adapter	Python list
push	**push(value)**	**push(value)**	**lst.append(value)**
pop	**pop()**	**pop**	**Del lst[-1]**
top	**peek()**	**top()**	**lst[-1]**
isEmpty	**empty()**	**empty()**	**len(lst) == 0**

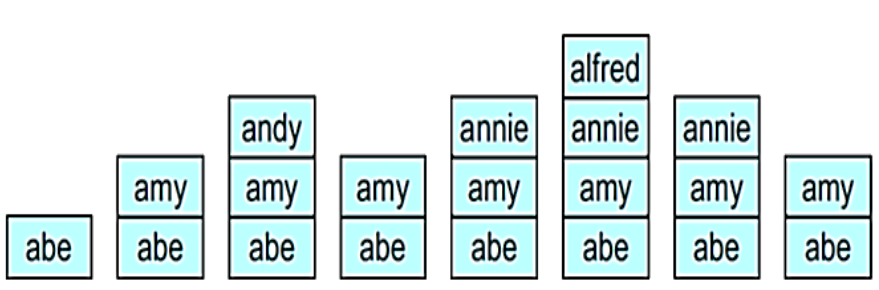
In a pure stack abstraction, the only access is to the topmost element. An item stored deeper in the stack can only be obtained by repeatedly removing the topmost element until the value in question rises to the top. But as we will see in the discussion of implementation alternatives, often a stack is combined with other abstractions, such as a dynamic

array. In this situation the data structure allows other operations, such as a search or direct access to elements. Whether or not this is a good design decision is a topic explored in one of the lessons described later in this chapter.

To illustrate the workings of a stack, consider the following sequence of operations:

```
push("abe")
push("amy")
push("andy")
pop()
push("anne")
push("alfred")
pop()
pop()
```

The following diagram illustrates the state of the stack after each of the eight operations.



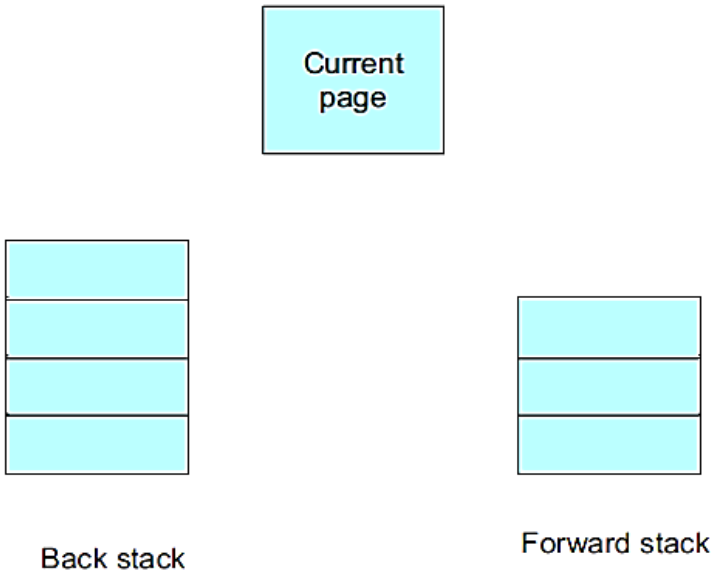
Applications of Stacks

Back and Forward Buttons in a Web Browser

In the beginning of this chapter, we noted how a stack might be used to implement the Back button in a web browser. Each time the user moves to a new web page, the current web page is stored on a stack. Pressing the back button causes the topmost element of this stack to be popped, and the associated web page is displayed.

However, that explanation really provided only half the story. To allow the user to move both forward and backward two stacks are employed. When the user presses the back button, the link to the current web page is stored on a separate stack for the forward button. As the user moved backward through previous pages, the link to each page is moved in turn from the back to the forward stack.

When the user pushes the forward button, the action is the reverse of the back button. Now the item from the forward stack is popped, and becomes the current web page. The previous web page is pushed on the back stack.



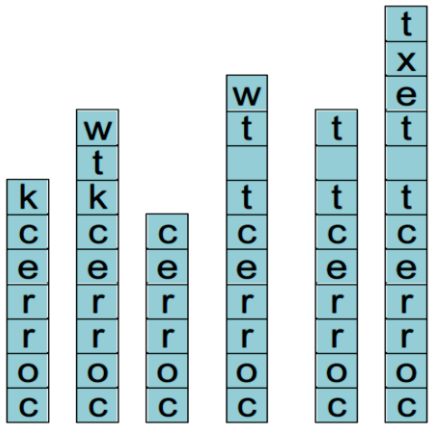
Question: The user of a web browser can also move to a new page by selecting a hyperlink. In fact, this is probably more common than using either the back or forward buttons. When this happens how should the contents of the back and forward stacks be changed?

Question: Web browsers often provide a *history* feature, which records all web pages accessed in the recent past. How is this different from the back stack? Describe how the history should change when each of the three following conditions occurs: (a) when the user moves to a new page by pressing a hyperlink, (b) when the user restores an old page by pressing the back button, and (c) when the user moves forward by pressing the forward button.

Buffered Character Input

An operating system uses a stack in order to correctly process backspace keys in lines of input typed at a keyboard. Imagine that you enter several keys and then discover a mistake. You press the backspace key to move backward over a previously entered character. Several backspaces may be used in turn to erase more than one character. If we use < to represent the backspace character, imagine that you typed the following:

```
correcktw<<<<t tw<ext
```



The operating system function that is handling character input will arrive at the correct text because it stores the characters as they are non-backspace character is simply backspace is typed, the topmost character is popped from the stack and erased.

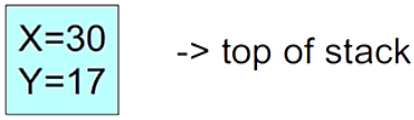
Question: What should be the effect if the user enters a backspace key and there are no characters in the input?

Activation Record Stack

Another example of a stack that we discussed briefly in an earlier chapter is the activation record stack. This term describes the spaced used by a running program to store parameters and local variables. Each time a function or method is invoked, space is set aside for these values. This space is termed an activation record. For example, suppose we execute the following function

```
void a (int x)
int y
y = x - 23;
y = b (y)
```

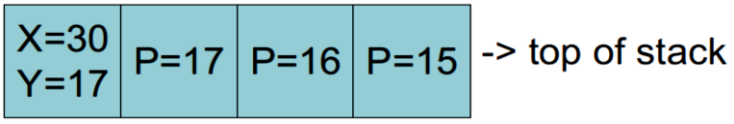
When the function a is invoked the activation record looks something like the following:



Imagine that b has the following recursive definition

```
int b (int p)
if (p < 15) return 1;
else return 1 + b(p-1)
```

Each time the function b is invoked a new activation record is created. New local variables and parameters are stored in this record. Thus, there may be many copies of a local variable stored in the stack, one for each current activation of the recursive procedure.



Functions, whether recursive or not, have a very simple execution sequence. If function a calls function b, the execution of function a is suspended while function b is active.

Function b must return before function a can resume. If function b calls another function, say c, then this same pattern will follow. Thus, function calls work in a strict stack-like fashion. This makes the operation of the activation record stack particularly easy. Each time a function is called new area is created on the activation record stack. Each time a function returns the space on the activation record stack is popped, and the recovered space can be reused in the next function call.

Question: What should (or what does) happen if there is no

available space in memory for a new activation record? What condition does this most likely represent?

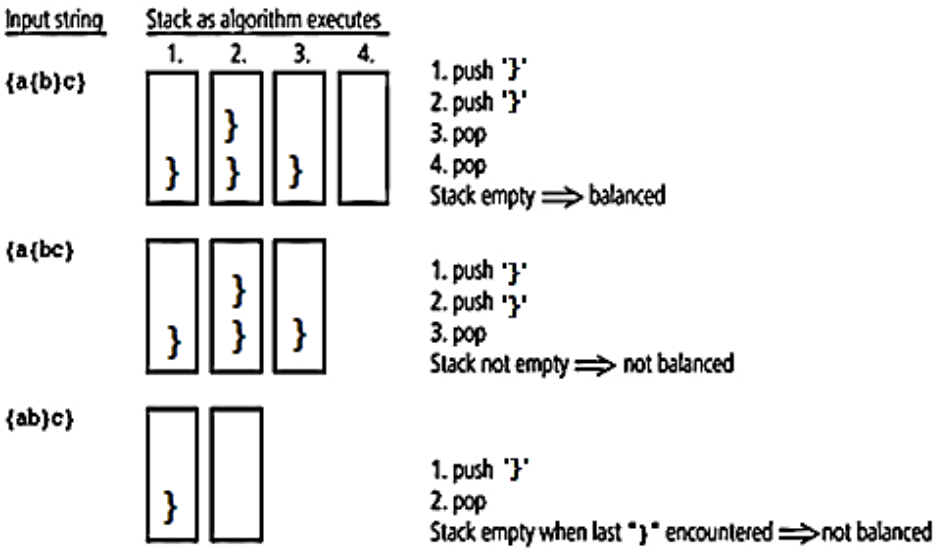
Checking Balanced Parenthesis

A simple application that will illustrate the use of the stack operations is a program to check for balanced parenthesis and brackets. By balanced we mean that every open parenthesis is matched with a corresponding close parenthesis, and parenthesis are properly nested. We will make the problem slightly more interesting by considering both parenthesis and brackets. All other characters are simply ignored. So, for example, the inputs (x(y)(z)) and a({(b)}c) are balanced, while the inputs w)(x) and p({(q)r)} are not.

To discover whether a string is balanced, each character is read in turn. The character is categorized as either an opening parenthesis, a closing parenthesis, or another type of character. Values of the third category are ignored. When a value of the first category is encountered, the corresponding close parenthesis is stored on the stack. For example, when a "(" is read, the character ")" is pushed on the stack. When a "{" is encountered, the character pushed is "}". The topmost element of the stack is therefore the closing value we expect to see in a well-balanced expression. When a closing

character is encountered, it is compared to the topmost item in the stack. If they match, the top of the stack is popped and execution continues with the next character. If they do not match, an error is reported. An error is also reported if a closing character is read and the stack is empty. If the stack is empty when the end of the expression is reached, then the expression is well-balanced.

Example-1: The following illustrates the state of the stack at various points during the processing of each of the given expressions.

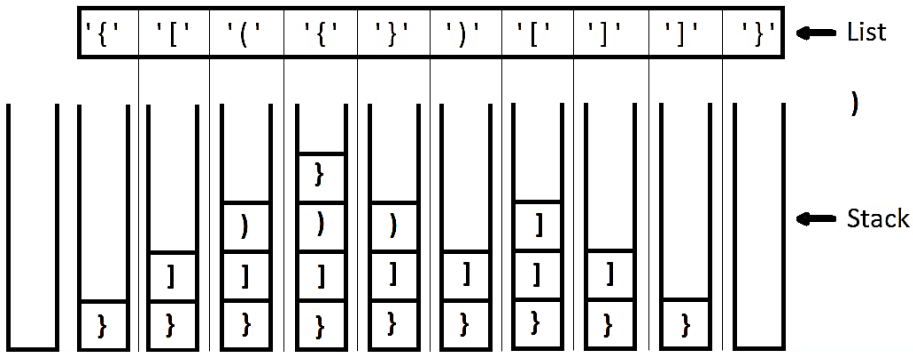


Note that an error occurs, when there are opening delimiters but no closing character. Moreover, another error can be detected, when a closing delimiter fails to match the

correct opening character.

Example-2: The following illustrates the state of the stack at various points during the processing of the expression:

a{b[cd(e{f}g)[h]j]k}



Question: Show the state of the stack after each character is read in the following expressions:

1. (a b { c } d ([e [f] g]) (j))

2. a (b { d e [f] g { h } I } j k) l m.

Evaluating Expressions

Two standard examples that illustrate the utility of the stack expression involve the evaluation of an arithmetic expression. Normally we are used to writing arithmetic expressions in what is termed *infix form*. Here a binary operator is written between two arguments, as in 2 + 3 * 7. Precedence rules are used to determine which operations

should be performed first, for example multiplication typically takes precedence over addition. Associativity rules apply when two operations of the same precedence occur one right after the other, as in 6 – 3 – 2. For addition, we normally perform the left most operation first, yielding in this case 3, and then the second operation, which yields the final result 1. If instead the associativity rule specified right to left evaluation, we would have first performed the calculation 3 – 2, yielding 1, and then subtracted this from 6, yielding the final value 5. Parenthesis can be used to override either precedence or associativity rules when desired. For example, we could explicitly have written 6 – (3 – 2).

The evaluation of infix expressions is not always easy, and so an alternative notion, termed *postfix notation*, is sometimes employed. In postfix notation the operator is written after the operands. The following are some examples:

Infix	**2 + 3**	**2 + 3 * 4**	**(2 + 3) * 4**	**2 + 3 + 4**	**2 - (3 – 4)**
Postfix	**2 3 +**	**2 3 4 * +**	**2 3 + 4 ***	**2 3 + 4 +**	**2 3 4 - -**

Notice that the need for parenthesis in the postfix form is avoided, as are any rules for precedence and associativity. We can divide the task of evaluating infix expressions into two separate steps, each of which makes use of a stack.

These steps are the conversion of an infix expression into postfix, and the evaluation of a postfix expression.

Conversion of infix to postfix

To convert an infix expression into postfix, we scan the value from left to right and divide the tokens into four categories. This is similar to the balanced parenthesis example. The categories are left and right parenthesis, operands (such as numbers or names) and operators. The actions for three of these four categories are simple:

Left parenthesis	Push on to stack
Operand	Write to output
Right parenthesis	Pop stack until corresponding left parenthesis is found. If stack becomes empty, report error. Otherwise, write each operator to output as it is popped from stack

The action for an operator is more complex. If the stack is empty or the current top of stack is a left parenthesis, then the operator is simply pushed on the stack. If neither of these conditions is true, then we know that the top of stack is an operator. The precedence of the current operator is compared to the top of the stack. If the operator

on the stack has higher precedence, then it is removed from the stack and written to the output, and the current operator is pushed on the stack. If the precedence of the operator on the stack is lower than the current operator, then the current operator is simply pushed on the stack. If they have the same precedence then if the operator associates left to right the actions are as in the higher precedence case, and if association is right to left, the actions are as in the lower precedence case.

Example-3: Show the state of the stack and the output as different characters in the following infix expression are read and find the corresponding postfix expression.

(A/B−C) *D+E)

Symbol Scanned	Stack	Output
(	(	-
A	(	A
/	(/	A
(	(/(	A
B	(/(	AB
-	(/(-	AB
C	(/(-	ABC
)	(/	ABC-
*	(*	ABC-/
D	(*	ABC-/D
+	(+	ABC-/D*
E	(+	ABC-/D*E
)	Empty	ABC-/D*E+

Postfix Expression: ABC-/D*E+

Question: Using the above algorithm, show the state of the stack and the output for each of the following infix expressions and find the corresponding postfix expressions.

```
1) (A+B*D)/(E-F)+G

2) A-(B+C)*D+E/F

3) X+Y*Z+(P*Q+R)*S

4) (A-B)*(C+D)-E

5) A+B*(C-D)-E/F*G+H

6) (A+B)/(C-D)-E/F*G+H
```

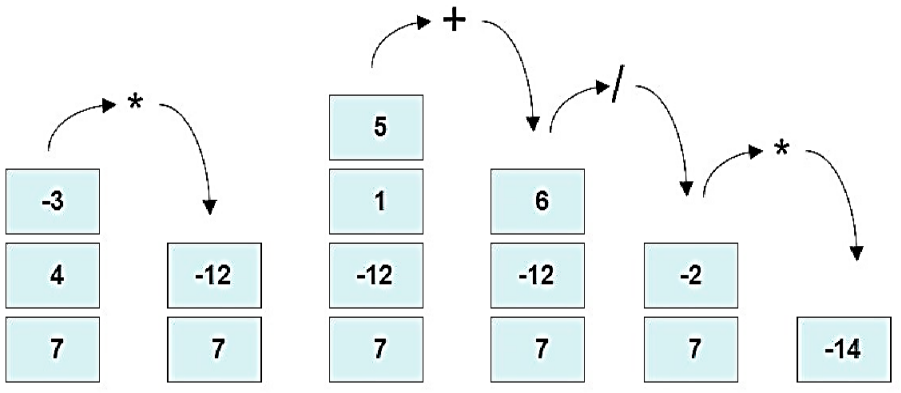
Evaluation of a postfix expression

The advantage of postfix notation is that there are no rules for operator precedence and no parenthesis. This makes evaluating postfix expressions particularly easy. As before, the postfix expression is evaluated left to right. Operands (such as numbers) are pushed on the stack. As each operator is encountered the top two elements on the stack are removed, the operation is performed, and the result is pushed back on the stack. Once all the input has been scanned the final result is left sitting in the stack.

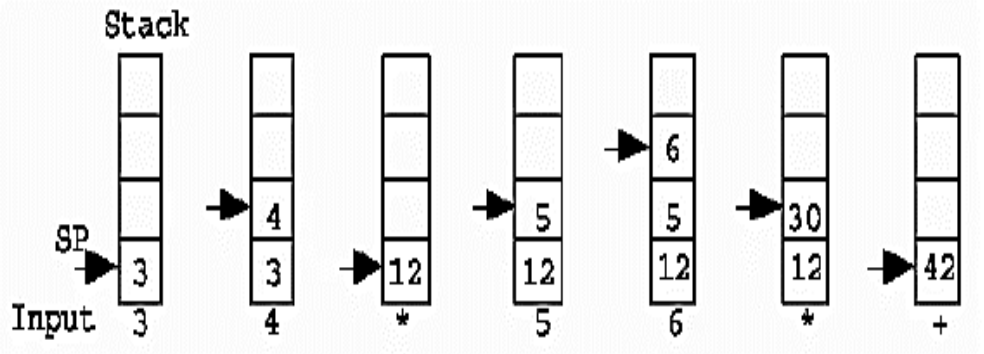
Example-4: The following illustrates the state of the stack

during the evaluation of the following postfix expression.

- Expression = 7 4 -3 * 1 5 + / *



Example-5: Show the state of the stack and the output for the following postfix expression: `3 4 * 5 6 * +`



Question: What error conditions can arise if the input is not a correctly formed postfix expression? What happens for the expression 3 4 + + ? How about 3 4 + 5 6 + ?

Stack Implementation Techniques

In the worksheets we will discuss two of the major techniques that are typically used to create stacks. These are the use of a dynamic array, and the use of a linked list. Study questions that accompany each worksheet help you explore some of the design tradeoffs a programmer must consider in evaluating each choice. The self-study questions given at the end of this chapter are intended to help you measure your own understanding of the material. Exercises and programming assignments that follow the self-study questions will explore the concept in more detail.

Worksheet 3	Introduction to the Dynamic Array
Worksheet 4	Dynamic Array Stack
Worksheet 18	Introduction to Linked List, Linked List Stack

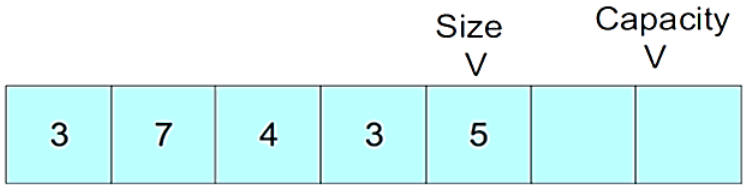
Building a Stack using a Dynamic Array

An array is a simple way to store a collection of values:

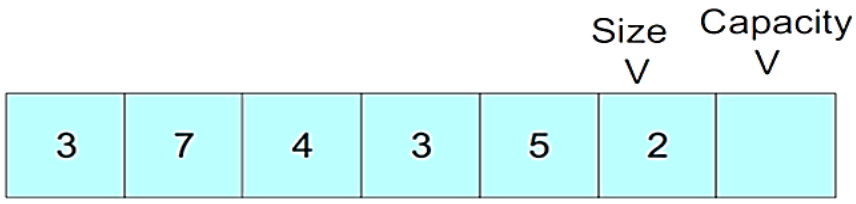
One problem with an array is that memory is allocated as a block. The size of this block is fixed when the array is created. If the size of the block corresponds directly to the number of elements in the collection, then adding a new value requires creating an entirely new block, and copying the

values from the old collection into the new.

This can be avoided by purposely making the array larger than necessary. The values for the collection are stored at the bottom of the array. A counter keeps track of how many elements are currently being stored in the array. This is termed the *size* of the stack. The size must not be confused with the actual size of the block, which is termed the *capacity*.

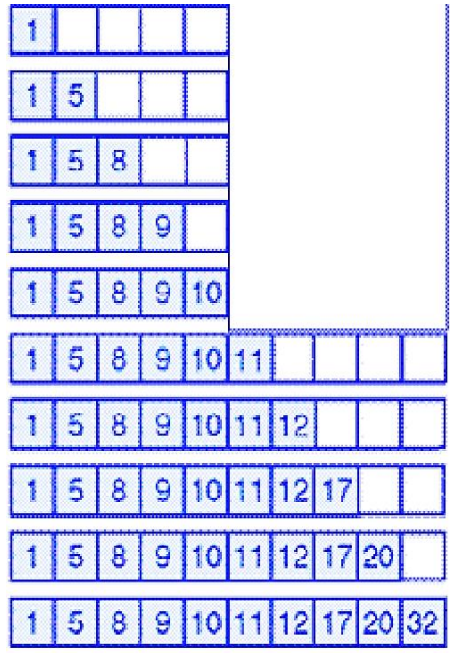


If the size is less than the capacity, then adding a new element to the stack is easy. It is simply a matter of incrementing the count on the size, and copying the new element into the correct location. Similarly, removing an element is simply a matter of setting the topmost location to null (thereby allowing the garbage collection system to recover the old value), and reducing the size.



Because the number of elements held in the collection can easily grow and shrink during run-time, this is termed a *dynamic array*. There are two exceptional conditions that must be handled. The first occurs when an attempt is made to remove a value from an empty stack. In this situation you should throw a StackUnderflow exception.

The second exceptional condition is more difficult. When a push instruction is requested but the size is equal to the capacity, there is no space for the new element. In this case a new array must be created. Typically, the size of the new array is twice the size of the current. Once the new array is created, the values are copied from existing array to the new array, and the new array replaces the current array. Since there is now enough room for the new element, it can be inserted.

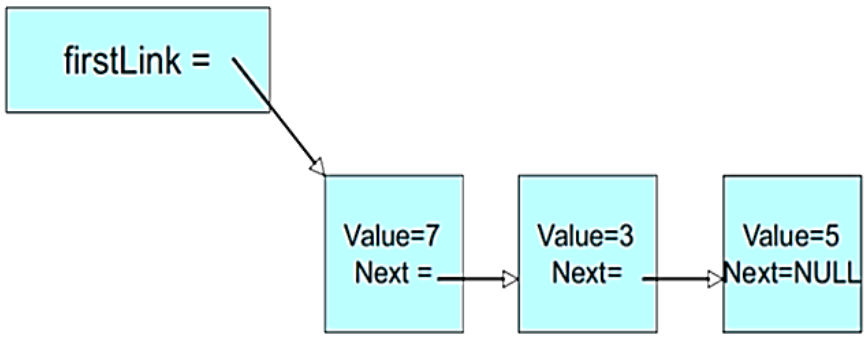


Worksheet 3 explores the implementation of a dynamic array stack. In the exercises at the end of the chapter you will

explore the idea that while the worst-case execution time for push is relatively slow, the worst case occurs relatively infrequently. Hence, the expectation is that in the average execution of push will be quite fast. We describe this situation by saying that the method push has constant *amortized* execution time.

Linked List Implementation of Stack

An alternative implementation approach is to use a linked list. Here, the container abstraction maintains a reference to a collection of elements of type Link. Each Link maintains two data fields, a value and a reference to another link. The last link in the sequence stores a null value in its link.



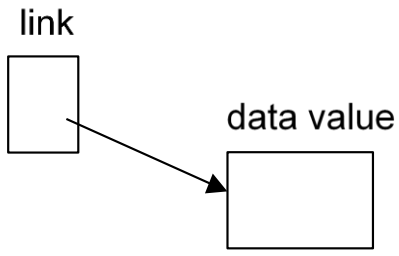
The advantage of the linked list is that the collection can grow as large as necessary, and each new addition to the chain of links requires only a constant amount of work. Because there are no big blocks of memory, it is never necessary to

copy an entire block from place to place. Worksheet 4 will introduce the idea of a linked list, and explore how a linked list can be used to implement a stack.

Memory Management

The linked list and the Dynamic Array data structures take different approaches to the problem of memory management. The Dynamic Array uses a large block of memory. This means that memory allocation is much less common, but when it occurs much more work must be performed. The linked list allocates a new link every time a new element is added. This makes memory allocation more frequent, but as the memory blocks are small less work is performed on each allocation.

If, as is often the case, a linked list is used to store pointers to a dynamically allocated value, then there are two dynamically allocated spaces to manage, the link and the data field itself:



An important principle of good memory management is "everybody must clean up their own mess". (This is sometimes termed the *kindergarten principle*). The linked list allocates space for the link, and so must ensure that the space is freed by calling the associated pop routine. The user of the list allocates space for the data value, and must therefore ensure that the field is freed when no longer needed. Whenever you create a dynamically allocated value you need to think about how and when it will be freed.

Earlier we pointed out the problem involved in placing a new element into the middle of a Dynamic Array (namely, that the following elements must then be moved). Linked lists will help solve this problem, although we have not yet demonstrated that in this chapter.

For the Dynamic Array we created a single general-purpose data structure, and then showed how to use that data structure in a variety of ways. In examining the linked list, we will take a different approach. Rather than making a single data abstraction, we will examine the *idea* of the linked list in a variety of different forms. In subsequent lessons we will examine a number of variations on this idea, such as header or sentinel links, single versus double links, maintaining a

pointer to the last as well as the first link, and more.

Occasionally you will find links placed directly into a data object. For example, suppose you were creating a card game, and needed a list of cards. One way to do this would be the following:

Each card can then be used as a link in a linked list.

```
struct Card {
int suit;
int rank;
   /* link to next card */
   struct Card * next;
};
```

Although this approach is easy to implement, it should be avoided, for several reasons. It confuses two issues, the management of cards, and the manipulation of the list. These problems should be dealt with independently. It makes your code very rigid; for example, you cannot move the Card abstraction to another program in which cards are not on a list, or must be placed in two different lists at the same time. And finally, you end up duplicating code that you can more easily write once and reuse by using a standard container.

Self-Study Questions

1. What are the defining characteristics of the stack abstraction?
2. Explain the meaning of the term LIFO. Explain why FILO might have been equally appropriate.
3. Give some examples of stacks found in real life.
4. Explain how the use of an activation record stack simplifies the allocation of memory for program variables. Explain how an activation record stack make it possible to perform memory allocation for recursive procedures.
5. Evaluate the following postfix polish expressions:
   a) 2 3 + 5 9 - *
   b) 2 3 5 9 + - *
6. How is the memory representation of a linked list different from that of a Dynamic Array?
7. What information is stored in a link?
8. How do you access the first element in a linked list? How would you get to the second element?
9. In the last link, what value is stored in the next field?
10. What would eventually happen if the pop function did not free the first link in the list?
11. Suppose you wanted to test your linked list class. What would be some boundary value test cases? Develop a test

harness program and execute your code with these test cases.

Short Exercises

1. Describe the state of an initially empty stack after each of the following sequence of operations. Indicate the values held by any variables that are declared, and also indicate any errors that may occur:

a)
```
que.addLast(new Integer(3));
Object a = que.getLast();
que.addLast(new Integer(5));
que.removeLast();
```

b)
```
que.addLast(new Integer(3));
que.addLast(new Integer(4));
que.removeLast();
que.removeLast();
Object a = que.getLast();
```

2. What is the Polish notation representation of the following expression?

(a `* (b + c)) + (b / d) * a`

3. One problem with polish notation is that you cannot use the same symbol for both unary and binary operators. Illustrate this by assuming that the minus sign is used for both unary and binary negation, and explain the two alternative meanings for the following postfix polish expression:

7 5 – –

4. Write an algorithm to translate a postfix polish expression into an infix expression that uses parenthesis only where necessary.

5. Phil Parker runs a parking lot where cars are stored in six stacks holding at most three cars each. Patrons leave the keys in their cars so that they can be moved if necessary.

Figure:

Assuming that no other parking space is available, should Phil allow his parking space to become entirely full? Assuming that Phil cannot predict the time at which patrons will return for their cars, how many spaces must he leave empty to ensure that he can reach any possible car?

6. Imagine a railroad switching circuit such as the one below. Railroad cars are given unique numbers, from 1 onwards. Cars come in from the right in a random order, and the goal is to assemble the cars in numeric order on the left.

Figure:

For example, to assemble the cars in the sequence shown (2, 1, 3) the car numbered 2 would be switched on to the siding. Then car numbered 1 would be moved on to the siding and then to the left. Next car numbered 2 would be moved up from the siding and assembled behind car 1. Finally, car numbered 3 would be moved from the right to the left, via the siding. Notice that the cars in the siding work as a stack, since if a car is moved on to a nonempty siding the existing cars cannot be accessed until the new car is removed.

Can you describe an algorithm that will rearrange the cars in sequential order regardless of the order in which they appear on the left? What is the complexity of your algorithm as a function of N, the number of cars entering on the right?

7. Some people prefer to define the Stack data type by means of a series of *axioms*. An example axiom might be that if a push operation is followed by a top, the value returned will be the same as the value inserted. We can write this in a code like fashion as follows:

```
Object val = new Integer(7);
stk.push(val);
boolean test = (val == stk.top()); // test must
```

```
always be true
```

Trace the ArrayStack implementation of the stack and verify the following axioms:

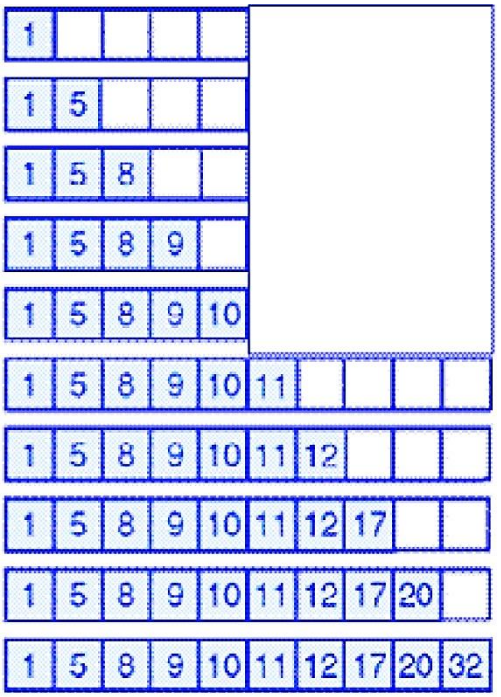
```
Stack stk = new Stack();

boolean test = stk.isEmpty(); // should always be
true

stk.push(new Integer(2));

boolean test = stk.isEmpty(); // should always be
false

Stack stk = new Stack();

boolean test = stk.pop(); // should always raise
error
```

8. Using a ListStack as the implementation structure do the same analysis as in previous question.

9. Does an ArrayStack or a ListStack use less memory? Assume for this question that a data value requires 1 unit of memory, and each memory reference (such as the next field in a Link, or the firstLink field in the ListStack, or the data field in the ArrayStack) also requires 1 unit of memory. How much memory is required to store a stack of 100 values in an ArrayStack? How much memory in a ListStack?

Analysis Exercises

1. When you developed the ArrayStack you were asked to determine the algorithmic execution time for the push operation. When the capacity was less than the size, the execution time was constant. But when a reallocation became necessary, execution time slowed to O(n).



This might at first seem like a very negative result, since it means that the worst-case execution time for pushing an item on to the stack is O(n). But the reality is not nearly so bleak. Look again at the picture that described the internal array as new elements were added to the collection.

Notice that the costly reallocation of a new array occurred only once during the time that ten elements were added to the collection. If we compute the *average* cost, rather than the *worst-case* cost, we will see that the ArrayStack is still a relatively efficient container.

To compute the average, count 1 “unit” of cost each time a value is added to the stack without requiring a reallocation. When the reallocation occurs, count ten “units” of cost for the assignments performed as part of the reallocation process, plus one more for placing the new element into the newly enlarged array. How many “units” are spent in the entire process of inserting these ten elements? What is the average “unit” cost for an insertion?

When we can bound an “average” cost of an operation in this fashion, but not bound the worst-case execution time, we call it *amortized constant* execution time, or *average* execution time. Amortized constant execution time is often written as O(1)+, the plus sign indicates it is not a guaranteed execution time bound.

Do a similar analysis for 25 consecutive add operations, assuming that the internal array begins with 5 elements (as shown). What is the cost when averaged over this range?

This analysis can be made into a programming assignment. Rewrite the ArrayStack class to keep track of the “unit cost” associated with each instruction, adding 1 to the cost for each simple insertion, and n for each time an array of n elements is copied. Then print out a table showing 200 consecutive insertions into a stack, and the value of the unit

cost at each step.

2. The Java standard library contains a number of classes that are implemented using techniques similar to those you developed in the programming lessons described earlier. The classes Vector and ArrayList use the dynamic array approach, while the class LinkedList uses the idea of a linked list. One difference is that the names for stack operations are different from the names we have used here:

Stack	Vector	ArrayList	LinkedList
Push(newValue)	Add(newValue)	Add(newValue)	addFirst(newObject)
Pop()	Remove(size()-1)	Remove(size()-1)	removeFirst(ewObject)
Top()	lastElement()	Get(size()-1)	getFirst()
IsEmpty()	Size() == 0	isEmpty()	isEmpty()

Another difference is that the standard library classes are designed for many more tasks than simply representing stacks, and hence have a much larger interface.

An important principle of modern software development is an emphasis on software reuse. Whenever possible you should leverage existing software, rather than rewriting new code that matches existing components. But there are various different techniques that can be used to achieve software reuse. In this exercise you will investigate

some of these, and explore the advantages and disadvantages of each. All of these techniques leverage an existing software component in order to simplify the creation of something new.

Imagine that you are a developer and are given the task of implementing a stack in Java. Part of the specifications insist that stack operations must use the push/pop/top convention. There are at least three different approaches you could use that

a) If you had access to the source code for the classes in the standard library, you could simply add new methods for these operations. The implementation of these methods can be pretty trivial, since they need do nothing more than invoke existing functions using different names.

b) You could create a new class using inheritance, and subclass from the existing class.

```
class Stack extends Vector {
    …
}
```

Inheritance implies that all the functionality of the parent class is available automatically for the child

class. Once more, the implementation of the methods for your stack can be very simple, since you can simply invoke the functions in the parent class.

3. The third alternative is to use composition rather than inheritance. You can create a class that maintains an internal data field of type Vector (alternatively, ArrayList or LinkedList). Again, the implementation of the methods for stack operations is very simple, since you can use methods for the vector to do most of the work.

```
class Stack<T> {
    private Vector<T> data;
     …
}
```

Write the implementation of each of these. (For the first, just write the methods for the stack operations, not the other vector code). Then compare and contrast the three designs. Issues to consider in your analysis include readability/usability and encapsulation. By readability or usability, we mean the following: how much information must be conveyed to a user of your new class before they can do their job. By encapsulation we mean: How good a job does your design do in guaranteeing the safety of the data? That is, making sure

that the stack is accessed using only valid stack instructions. On the other hand, there may be reasons why you might want to allow the stack to be accessed using non-stack instructions. A common example is allowing access to all elements of the stack, not just the first. Which design makes this easier?

If you are the developer for a collection class library (such as the developer for the Java collection library), do you think it is a better design choice to have a large number of classes with very small interfaces, or a very small number of classes that can each be used in a number of ways, and hence have very large interfaces? Describe the advantages and disadvantages of both approaches.

The bottom line is that all three design choices have their uses. You, as a programmer, need to be aware of the design choices you make, and the reasons for selecting one alternative over another.

4. In the previous question you explored various alternative designs for the Stack abstraction when built on top of an existing class. In some of those there was the potential that the end user could manipulate the stack using commands that were not part of the stack abstraction. One way to avoid

this problem is to define a stack interface. An *interface* is similar to a class, but only describes the signatures for operations, not their implementations. An interface for the Stack abstraction can be given as follows:

```
interface Stack<T> {

    public void push (T newValue);

    public void pop ();

    public T top ();

    public Boolean isEmpty();

}
```

Rewrite your two stack implementations (ArrayStack and ListStack) using the stack interface. Verify that you can now declare a variable of type Stack, and assign it a value of either ArrayStack or ListStack. Rewrite the implementations from the previous analysis lesson so that they use the stack interface. Verify that even when the stack is formed using inheritance from ArrayList or Vector, that a variable declared as type Stack cannot use any operations except those specified by the interface.

Can you think of a reason why the designer of the Java collection classes did not elect to define interfaces for the common container types?

5. Whenever you have two different implementations with the same interface, the first question you should ask is whether the algorithmic execution times are the same. If not, then select the implementation with the better algorithmic time. If, on the other hand, you have similar algorithmic times, then a valid comparison is to examine actual clock times (termed a benchmark). You can determine the current time using the function named clock that is defined in the <time.h> interface file:

```
# include <time.h>

double getMilliseconds() {

return 1000.0 * clock() / CLOCKS_PER_SEC;

}
```

As the name suggests, this function returns the current time in milliseconds. If you subtract an earlier time from a later time, you can determine the amount of time spent in performing an action.

In the first experiment, try inserting and removing ten values from a stack, doing these operations *n* times, for various values of *n*. Plot your results in a graph that compares the execution time to the value of *n* (where n ranges, say, from 1000 to 10,000 in increments of 1000).

This first experiment can be criticized because once

the vector has reached its maximum size it is never enlarged. This might tend to favor the vector over the linked list. An alternative exercise would be to insert and remove *n* values. This would force the vector to continually increase in size. Perform this experiment and compare the resulting execution times.

Programming Assignments

1. Complete the implementation of a program that will read an infix expression from the user, and print out the corresponding postfix expression.
2. Complete the implementation of a program that will read a postfix expression as a string, break the expression into parts, evaluate the expression and print the result.
3. Combine parts 1 and 2 to create a program that will read an infix expression from the user, convert the infix expression into an equivalent postfix expression, then evaluate the postfix expression.
4. Add a graphical user interface (GUI) to the calculator program. The GUI will consist of a table of buttons for numbers and operators. By means of these, the user can enter an expression in infix format. When the calculate

button is pushed, the infix expression is converted to postfix and evaluated. Once evaluated, the result is displayed.

5. Here is a technique that employs two stacks in order to determine if a phrase is a palindrome, that is, reads the same forward and backward (for example, the word "rotator"" is a palindrome, as is the string "rats live on no evil star").
   - Read the characters one by one and transfer them into a stack. The characters in the stack will then represent the reversed word.
   - Once all characters have been read, transfer half the characters from the first stack into a second stack. Thus, the order of the words will have been restored.
   - If there were an odd number of characters, remove one further character from the original stack.
   - Finally, test the two stacks for equality, element by element. If they are the same, then the word is a palindrome.
   - Write a procedure that takes a String argument and tests to see if it is a palindrome using this algorithm.
6. In Chapter 1 you learned that a Bag was a data structure

characterized by the following operations: add an element to the collection, test to see if an element is in the collection, and remove an element from the collection. We can build a bag using the same linked list techniques you used for the linked list stack. The add operation is the same as the stack. To test an element, simply loop over the links, examining each in turn. The only difficult operation is removed, since to remove a link, you need access to the immediately preceding link. To implement this, one approach is to loop over the links using a pair of pointers. One pointer will reference the current link, while the second will always reference the immediate predecessor (or null, if the current link is the first element). That way, when you find the link to remove, you simply update the predecessor link. Implement the three bag operations using this approach.

On the Web

The wikipedia entry for "Stack (data structure)" provides another good introduction to the concept. Other informative wikipedia entries include LIFO, call stack, and stack-based memory allocation. Wikipedia also provides a bibliographical sketch of Friedrich L. Bauer, the computer scientist who first

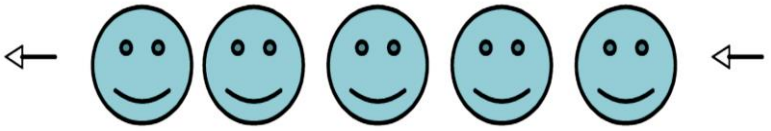
proposed the use of stack for evaluating arithmetic expressions. The NIST *Dictionary of Algorithms and Data Structures* also has a simple description of the stack data type. (http://www.nist.gov/dads/HTML/stack.html). If you google "Stack in Java" (or C++, or C, or any other language) you will find many good examples.

Chapter -3

Queues and Deques

QUEUES AND DEQUES

After the stack, the next simplest data abstraction is the *queue*. As with the stack, the queue can be visualized with many examples you are already familiar with from everyday life. A simple illustration is a line of people waiting to enter a theater. The fundamental property of the queue is that items are inserted at one end (the rear of the line) and removed from the other (the door to the theater). This means that the order that items are removed matches the order that they are inserted. Just as a stack was described as a LIFO (last-in, first-out) container, this means a queue can be described as FIFO (first in, first out).



A variation is termed the *deque*, pronounced "deck", which stands for *d*ouble-*e*nded *que*ue. In a deque values can be inserted at *either* the front or the back, and similarly the deck allows values to be removed from either the front or the back. A collection of peas in a straw is a useful mental image. (Or, to update the visual image, a series of tapioca buds in a bubble-tea straw).

Queues and deques are used in a number of ways in computer applications. A printer, for example, can only print one job at a time. During the time it is printing, there may be many different requests for other output to be printed. To handle these the printer will maintain a queue of pending print tasks. Since you want the results to be produced in the order that they are received, a queue is the appropriate data structure.

The Queue ADT specification

The classic definition of the queue abstraction as an ADT includes the following operations:

addBack (newElement)	Insert a value into the queue
front()	Return the front (first) element in queue
removeFront()	Remove the front (first) element in queue
isEmpty()	Determine whether the queue has any elements

The deque will add the following:

addFront(newElement)	Insert a value at front of deque
back()	Return the last element in the queue
removeBack()	Return the last element in the queue

Notice that the queue is really just a special case of the deque. Any implementation of the deque will also work as an implementation of the queue. (A deque can also be used to implement a stack, a topic we will explore in the exercises).

As with the stack, it is the FIFO (first in, first out) property that is, in the end, the fundamental defining characteristic of the queue, and not the names of the operations. Some designers choose to use names such as “add”, “push” or “insert”, leaving the location unspecified. Similarly, some implementations elect to make a single operation that will both return and remove an element, while other implementations separate these two tasks, as well do here. And finally, as in the stack, there is the question of what the effect should be if an attempt is made to access or remove an element from an empty collection. The most common solutions are to throw an exception or an assertion error (which is what we will do), or to return a special value, such as Null.

For a deque, the defining property is that elements can only be added or removed from the end points. It is not possible to add or remove values from the middle of the collection.

The following table shows the names of deque and queue operations in various programming languages:

Operation:	C++	Java	Perl	Python
insert at front	push_front	addFirst	unshift	appendleft
Insert at back	Push_back	addLast	Push	append
remove last	pop_back	removeLast	pop	pop
remove first	pop_front	removeFirst	shift	popleft
examine last	back	getLast	$_[-1]	deque[-1]
examine first	front	getFirst	$_[0]	deque[0]

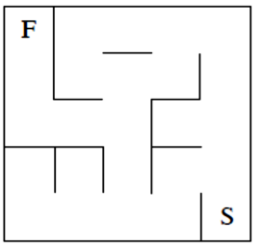
Applications of Queues

Given that queues occur frequently in real life, it is not surprising that they are also frequently used in simulations. To model customers using a bank, for example, you could use a queue of waiting patrons. A simulation might want to ask questions such as the how the average waiting time would change if you added or removed a teller position.

Queues are also used in any time collection where time of insertion is important. We have noted, for example, that a printer might want to keep the collection of pending jobs in a queue, so that they will be printed in the same order that they were submitted.

Depth-first and Breadth-first search

Imagine you are searching a maze, such as the one shown at right. The goal of the search is to move from the square marked S, to the square marked F.



A simple algorithm would be the following:

How to search a maze:

- Keep a list of squares you have visited, initially empty.
- Keep a stack of squares you have yet to visit. Put the starting square in this stack
- While the stack is not empty:

  Remove an element from the stack

  If it is the finish, then we are done

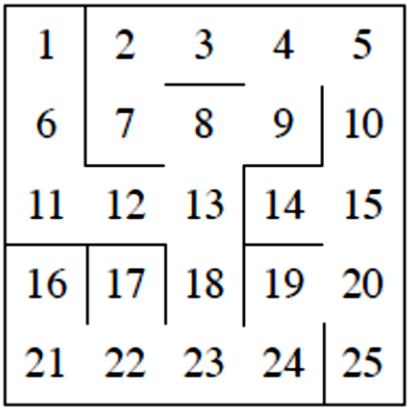
  Otherwise, if you have already visited this square, ignore it. Otherwise, mark the square on your list of visited positions, and add all the neighbors of this square to your stack.

- If you eventually reach an empty stack and have not found the start, there is no solution

To see the working of this algorithm in action, let us number of states of our maze from 1 to 25, as shown. There is only one cell reachable from the starting position, and thus after the first step the queue contains only one element:

the deque

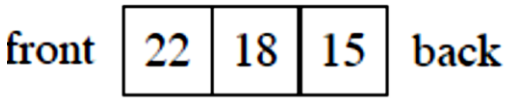
front | 20 | back



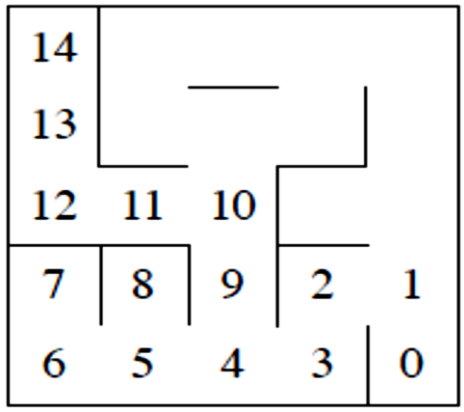
This value is pulled from the stack, and the neighbors of the cell are inserted back. This time there are two neighbors, and so the stack will have two entries.

front | 19 | 15 | back

Only one position can be explored at any time, and so the first element is removed from the stack, leaving the second waiting for later exploration. Two steps later we again have a choice, and both neighbors are inserted into the stack. At this point, the stack has the following contents:



The solution is ultimately found in fifteen steps. The following shows the path to the solution, with the cells numbered in the order in which they were considered.



The strategy embodied in this code doggedly pursues a single path until it either reaches a dead end or until the solution is found. When a dead end is encountered, the most recent alternative path is reactivated, and the search continues. This approach is called a ***depth- first search,*** because it moves deeply into the structure before examining alternatives. A depth-first search is the type of search a single individual might perform in walking through a maze.

Suppose, on the other hand, that there were a group of people walking together. When a choice of alternatives was

encountered, the group might decide to split itself into smaller groups, and explore each alternative simultaneously. In this fashion all potential paths are investigated at the same time. Such a strategy is known as a ***breadth-first* search**.

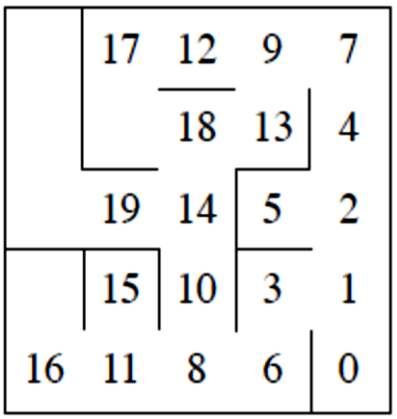
What is intriguing about the maze-searching algorithm is that the exact same algorithm can be used for both, changing only the underlying data structure. Imagine that we change the stack in the algorithm to a queue:

How to search a maze using breadth-first search:

- Keep a list of squares you have visited, initially empty.
- Keep a queue of squares you have yet to visit. Put the starting square in this queue
- While the queue is not empty:
  - Remove an element from the queue
  - If it is the finish, then we are done
  - Otherwise, if you have already visited this square, ignore it
  - Otherwise, mark the square on your list of visited positions, and add all the neighbors of this queue to your stack.

– If you eventually reach an empty queue and have not found the start, there is no solution

As you might expect, a breadth-first search is more thorough, but may require more time than a depth-first search. While the depth-first search was able to find the solution in 15 steps, the breadth-first search is still looking after 20.



The diagram at right shows the search at this point. Trace with your finger the sequence of steps as they are visited. Notice how the search jumps all over the maze, exploring a number of different alternatives at the same time. Another way to imagine a breadth-first first is as what would happen if ink were poured into the maze at the starting location, and slowly permeates every path until the solution is reached.

We will return to a discussion of depth-first and breadth-first search in a later chapter, after we have developed a number of other data structures, such as graphs, that are useful in the representation of this problem.

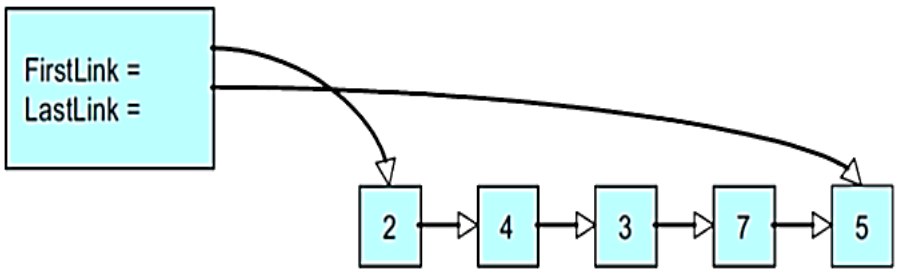
Queue Implementation Techniques

As with the stack, the two most common implementation techniques for a queue are to use a linked list or to use an array. In the worksheets you will explore both of these alternatives.

Worksheet 5	Linked List Queue, Introduction to Sentinels
Worksheet 6	Linked List Deque
Worksheet 20	Dynamic Array Queue

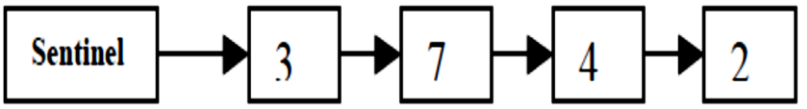
Building a Linked List Queue

A stack only needs to maintain a link to one end of the chain of values, since both insertions and removals occur on the same side. A queue, on the other hand, performs insertions on one side and removals from the other. Therefore, it is necessary to maintain a links to both the front and the back of the collection.



We will add another variation to our container. A *sentinel* is a special link, one that does not contain a value. The

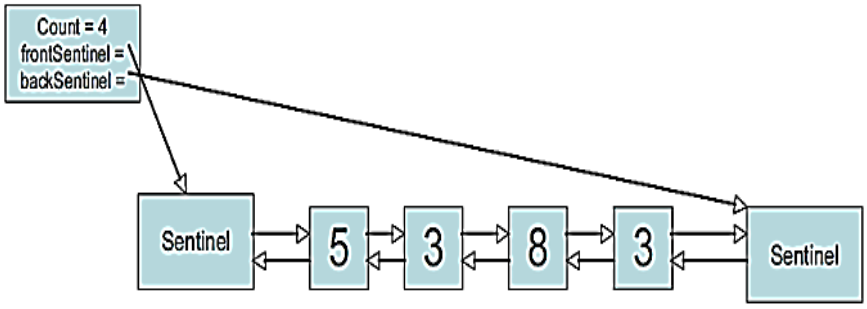
sentinel is used to mark either the beginning or end of a chain of links. In our case we will use a sentinel at the front. This is sometimes termed a *list header*. The presence of the sentinel makes it easier to handle special cases. For example, the list of links is never actually empty, even when it is logically empty, since there is always at least one link (namely, the sentinel). A new value is inserted after the end, after the element pointed to by the last link field. Afterwards, the last link is updated to refer to the newly inserted value.



Values are removed from the front, as with the stack. But because of the sentinel, these will be the element right after the sentinel. You will explore the implementation of the queue using linked list techniques in Worksheet 5.

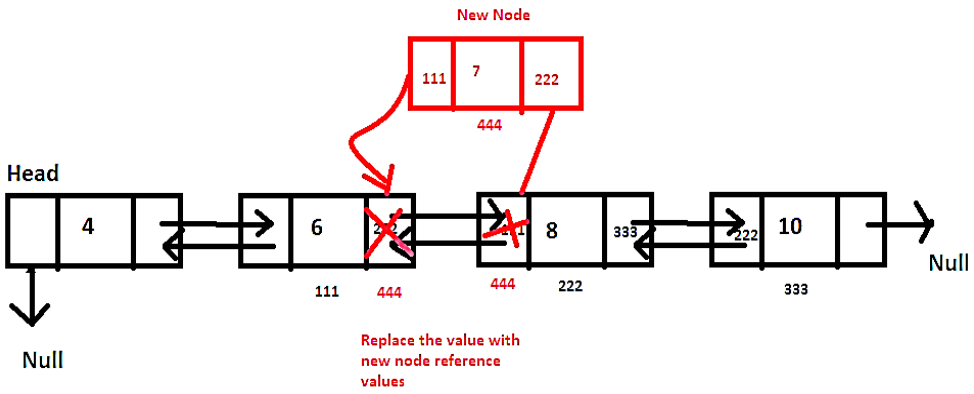
A Linked List Deque – using Double Links

You may have noticed that removal from the end of a ListQueue is difficult because with only a single link it is difficult to "back up". That is, while you have a pointer to the end sentinel, you do not have an easy way to back up and find the link immediately preceding the sentinel.

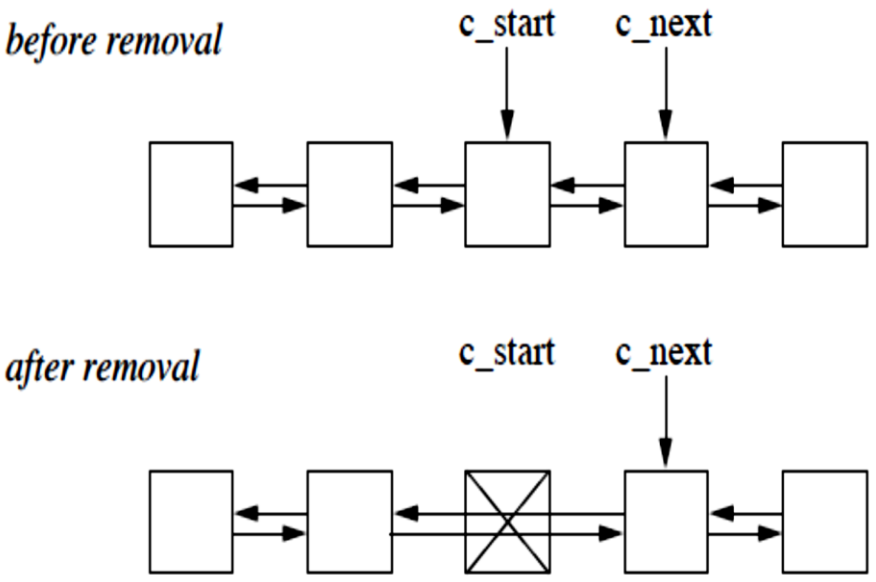


One solution to this problem is to use a *doubly-linked* list. In a doubly-linked list, each link maintains two pointers. One link, the forward link, points forward toward the next link in the chain. The other link, the prev link, points backwards towards the previous element. Anticipating future applications, we now also keep a count of the number of elements in the list.

With this picture, it is now easy to move either forward or backwards from any link. We will use this new ability to create a linked list deque. In order to simplify the implementation, we will this time include sentinels at both the beginning and the end of the chain of links. Because of the sentinels, both adding to the front and adding to the end of a collection can be viewed as special cases of a more general "add to the middle of a list" operation. That is, perform an insertion such as the following:



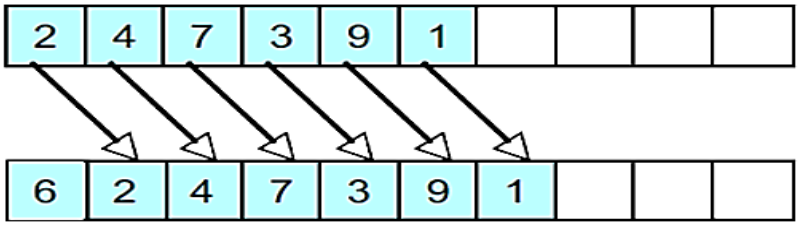
Similarly, removing a value (from other the front or the back) is a special case of a more general remove operation:



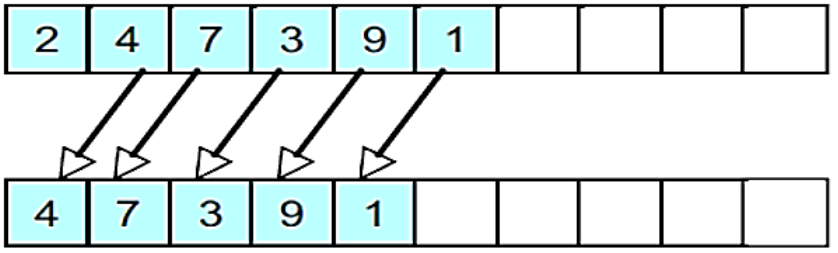
In Worksheet 6 you will complete the implementation of the list deque based on these ideas.

A Dynamic Array Deque

The dynamic array that we used in implementing the ArrayStack will not work for either a queue or a deque. The reason is that inserting an element at the front of the array requires moving every element over by one position. A loop to perform this movement would require O(n) steps, far too slow for our purposes.

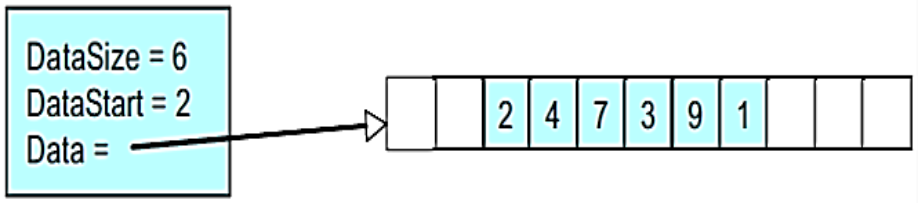


If we tried adding elements to the end (as we did with the stack) then the problem is reversed. Now it is the remove operation that is slow, since it requires moving all values down by one position

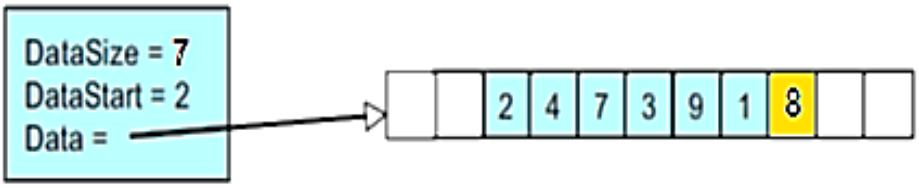


The root of the problem is that we have fixed the start of the collection at array index position zero. If we simply loosen that requirement, then things become much easier. Now in

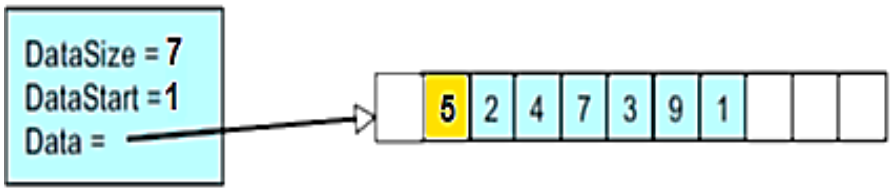
addition to the data array our collection will maintain two integer data fields; a size (as before) and a starting location.



Adding a value to the end is similar to the ArrayStack. Simply increase the size, and place the new element at the end.

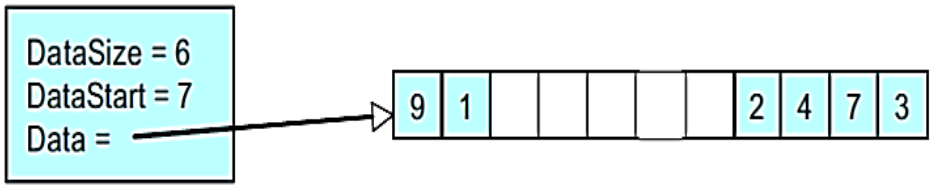


But adding a value to the front is also simple. Simply decrement the starting point by one, then add the new element to the front:



Removing elements undo these operations. As before, if an insertion is performed when the size is equal to the capacity, the array must be doubled in capacity, and the values copied into the new array.

There is just one problem. Nothing prevents the data values from wrapping around from the upper part of the array to the lower:



To accommodate index values must be computed carefully. When an index wraps around the end of the array, it must be altered to point to the start of the array. This is easily done by subtracting the capacity of the array. That is, suppose we try to index the fifth element in the picture above. We start by adding the index, 5, to the starting location, 7. The resulting sum is 12. But there are only eleven values in the collection. Subtracting 11 from 12 yields 1. This is the index for the value we seek.

In worksheet 20, you will explore the implementation of the dynamic array deque constructed using these ideas. A deque implemented in this fashion is sometimes termed a *circular buffer*, since the right-hand side circles around to begin again on the left.

Self-Study Questions

1. What are the defining characteristics of the queue ADT?
2. What do the letters in FIFO represent? How does this describe the queue?
3. What does the term deque stand for?
4. How is the deque ADT different from the queue abstraction?
5. What will happen if an attempt is made to remove a value from an empty queue?
6. What does it mean to perform a depth-first search? What data structure is used in performing a depth-first search?
7. How is a breadth-first search different from a depth-first search? What data structure is used in performing a breadth-first search?
8. What is a sentinel in a linked list?
9. What does it mean to say that a list is singly-linked?
10. How is a doubly-linked list different from a singly-linked list? What new ability does the doubly-linked feature allow?
11. Why is it difficult to implement a deque using the same dynamic array implementation that was used in the dynamic array stack?

Analysis Exercises

1. A deque can be used as either a stack or a queue. Do you think that it is faster or slower in execution time than either a dynamic array stack or a linked list stack? Can you design an exercise to test your hypothesis? In using a Deque as a stack there are two choices; you can either add and remove from the front, or add and remove from the back. Is there a measurable difference in execution time between these two alternatives?

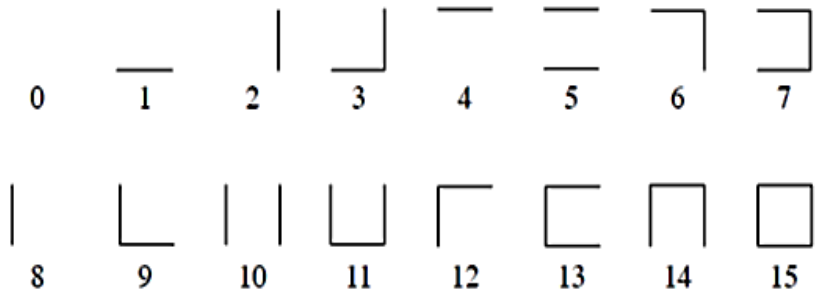
Programming Assignments

1. Many computer applications present practice drills on a subject, such as arithmetic addition. Often such systems will present the user with a series of problems and keep track of those the user answered incorrectly. At the end of the session, the incorrect problems will then be presented to the user a second time. Implement a practice drill system for addition problems having this behavior.

2. FollowMe is a popular video game. The computer displays a sequence of values, and then asks the player to reproduce the same sequence. Points are scores if the entire sequence was produced in order. In

implementing this game, we can use a queue to store the sequence for the period of time between generation by the computer and response by the player.

3. To create a maze-searching program you first need some way to represent the maze. A simple approach is to use a two-dimensional integer array. The values in this array can represent the type of room, as shown below. The values in this array can then tell you the way in which it is legal to move. Positions in the maze can be represented by (i,j) pairs. Use this technique to write a program that reads a maze description, and prints a sequence of moves that are explored in either a depth-first or breadth-first search of the maze.

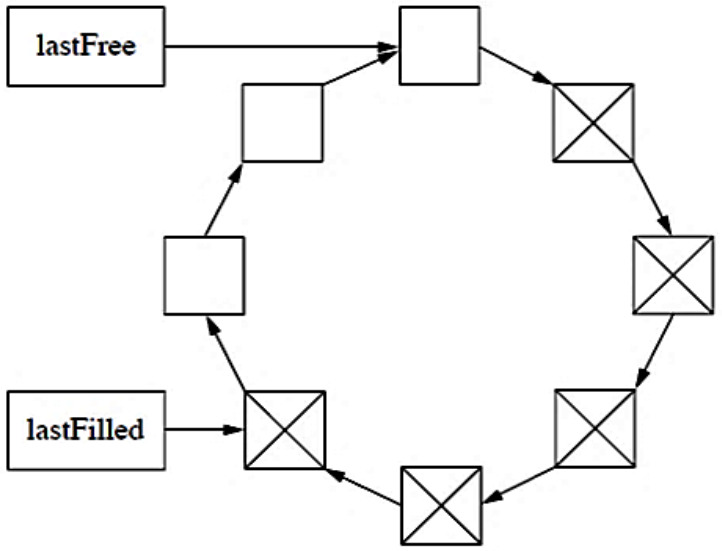


4. In Chapter 1, you learned that a *Bag* was a data structure characterized by the following operations: add an element to the collection, test to see if an element is in the collection, and remove an element

from the collection. We can build a bag using the same linked list techniques you used for the linked list queue. The add operation is the same as the queue. To test an element, simply loop over the links, examining each in turn. The only difficult operation is removed, since to remove a link, you need access to the immediately preceding link. To implement this, one approach is to loop over the links using a pair of pointers. One pointer will reference the current link, while the second will always reference the immediate predecessor (or the sentinel, if the current link is the first element). That way, when you find the link to remove, you simply update the predecessor link. Implement the three bag operations using this approach. Does the use of the sentinel make this code easier to understand than the equivalent code was for the stack version?

5. Another implementation technique for a linked list queue is to link the end of the queue to the front. This is termed a circular buffer. Two pointers then provide access to the last filled position, and the last free position. To place an element into the queue, the last filled pointer is advanced, and the value stored. To remove a value the last free pointer is advanced, and

the value returned. The following picture shows this structure. How do you know when the queue is completely filled? What action should you take to increase the size of the structure in this situation? Implement a circular buffer queue based on these ideas.



On The Web

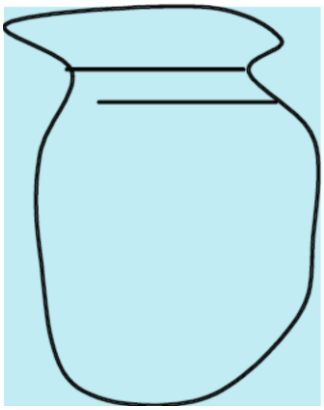
Wikipedia has well written entries for queue, deque, FIFO, as well as on the implementation structures dynamic array, linked list and sentinel node. The NITS *Dictionary of Algorithms and Data Structures* also has a good explanation.

Chapter-4

Bags and Sets

Bags and Sets

In the stack and the queue abstractions, the order that elements are placed into the container is important, because the order elements are removed is related to the order in which they are inserted. For the *Bag*, the order of insertion is completely irrelevant. Elements can be inserted and removed entirely at random.

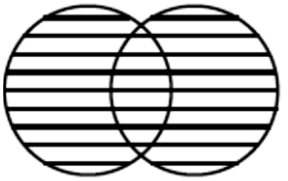


By using the name *Bag* to describe this abstract data type, the intent is to once again to suggest examples of collection that will be familiar to the user from their everyday experience. A bag of marbles is a good mental image.

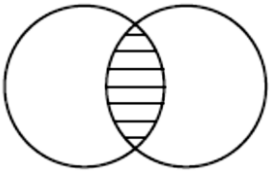
Operations you can do with a bag include inserting a new value, removing a value, testing to see if a value is held in the collection, and determining the number of elements in the collection. In addition, many problems require the ability to loop over the elements in the container. However, we want to be able to do this without exposing details about how the collection is organized (for example, whether it uses an array

or a linked list). Later in this chapter, we will see how to do this using a concept termed an *iterator*.

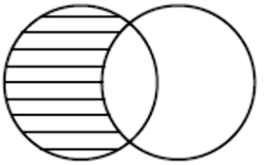
A *Set* extends the bag in two important ways. First, the elements in a set must be unique; adding an element to a set when it is already contained in the collection will have no effect. Second, the set adds a number of operations that combine two sets to produce a new set. For example, the set *union* is the set of values that are present in either collection.



The *intersection* is the set of values that appear in both collections.



A set *difference* includes values found in one set but not the other.



Finally, the subset test is used to determine if all the values found in one collection are also found in the second. Some implementations of a set allow elements to be repeated more than once. This is usually termed a *multiset*.

The Bag and Set ADT specifications

The traditional definition of the Bag abstraction includes the following operations:

Add (newElement)	Place a value into the bag
Remove (element)	Remove the value
Contains (element)	Return true if element is in collection
Size ()	Return number of values in collection
Iterator ()	Return an iterator used to loop over collection

As with the earlier containers, the names attached to these operations in other implementations of the ADT need not exactly match those shown here. Some authors prefer "insert" to "add", or "test" to "contains". Similarly, there are differences in the exact meaning of the operation "remove". What should be the effect if the element is not found in the collection? Our implementation will silently do nothing. Other authors prefer that the collection throw an exception in this situation. Either decision can still legitimately be termed a bag type of collection.

The following table gives the names for bag-like containers in several programming languages.

Operation	Java Collection	C++ vector	Python
Add	Add(element)	Push_back(element)	Lst.append(element)
remove	Remove(element)	Erase(iterator)	Lst.remove(element)
contains	Contains(element)	Count(iterator)	Lst.count(element)

The set abstraction includes, in addition to all the bag operations, several functions that work on two sets. These include forming the intersection, union or difference of two sets, or testing whether one set is a subset of another. Not all programming languages include set abstractions. The following table shows a few that do:

operation	Java Set	C++ set	Python list comprehensions
intersection	retainAll	Set_intersection	[x for x in a if x in b]
union	addAll	Set_union	[x if (x in b) or (x in a)]
difference	removeAll	Set_difference	[x for x in a if x not in b]
subset	containsAll	includes	Len([x for x in a if x not in b]) != 0

Python list comprehensions (modeled after similar facilities in the programming languages ML and SETL) are a particularly elegant way of manipulating set abstractions.

Applications of Bags and Sets

The bag is the most basic of collection data structures, and hence almost any application that does not require remembering the order that elements are inserted will use a variation on a bag. Take, for example, a spelling checker. An on-line checker would place a dictionary of correctly spelled words into a bag. Each word in the file is then tested against

the words in the bag, and if not found it is flagged. An off-line checker could use set operations. The correctly spelled words could be placed into one bag, the words in the document placed into a second, and the difference between the two computed. Words found in the document but not the dictionary could then be printed.

Bag and Set Implementation Techniques

For a Bag, we have a much wider range of possible implementation techniques than we had for stacks and queues. So many possibilities, in fact, that we cannot easily cover them in contiguous worksheets. The early worksheets describe how to construct a bag using the techniques you have seen, the dynamic array and the linked list. Both of these require the use of an additional data abstraction, the iterator. Later, more complex data structures, such as the skip list, avl tree, or hash table, can also be used to implement bag-like containers.

Another thread that weaves through the discussion of implementation techniques for the bag is the advantages that can be found by maintaining elements in order. In the simplest there is the sorted dynamic array, which allows the use of binary search to locate elements quickly. A skip list

uses an ordered linked list in a more subtle and complex fashion. AVL trees and similarly balanced binary trees use ordering in an entirely different way to achieve fast performance.

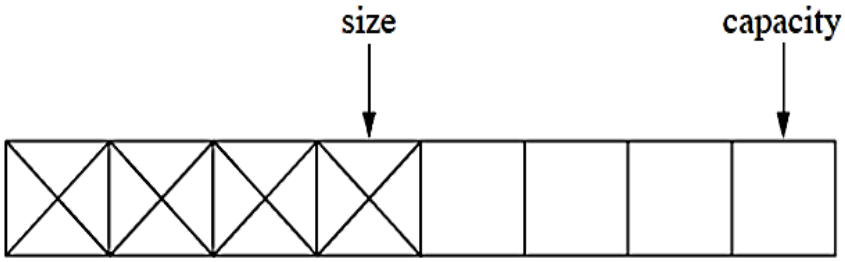
The following worksheets describe containers that implement the bag interface. Those involving trees should be delayed until you have read the chapter on trees.

Worksheet 8	Dynamic Array Bag
Worksheet 9	Linked List Bag
Worksheet 10	Introduction to the Iterator
Worksheet 11	Linked List Iterator
Worksheet 13	Sorted Array Bag
Worksheet 15	Skip list bag
Worksheet 16	Balanaced Binary Search Trees
Worksheet 18	AVL trees
Worksheet 24	Hash tables

Building a Bag using a Dynamic Array

For the Bag abstraction, we will start from the simpler dynamic array stack described in Chapter 2, and not the more complicated deque variation you implemented in Chapter 3. Recall that the Container maintained two data fields. The first was a reference to an array of objects. The number of positions in this array was termed the *capacity* of the

container. The second value was an integer that represented the number of elements held in the container. This was termed the *size* of the collection. The size must always be smaller than or equal to the capacity.



As new elements are inserted, the size is increased. If the size reaches the capacity, then a new array is created with twice the capacity, and the values are copied from the old array into the new. This process of reallocating the new array is an issue you have already solved back in Chapter 2. In fact, the function *add* can have exactly the same behavior as the function *push* you wrote for the dynamic array stack. That is, add simply inserts the new element at the end of the array.

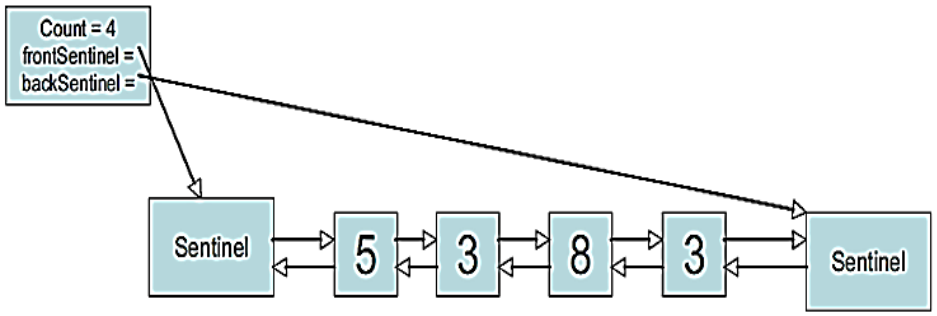
The *contains* function is also relatively simple. It simply uses a loop to cycle over the index values, examining each element in turn. If it finds a value that matches the argument, it returns true. If it reaches the end of the collection without finding any value, it returns false.

The *remove* function is the most complicated of the Bag

abstraction. To simplify this task, we will divide it into two distinct steps. The *remove* function, like the *contains* function, will loop over each position, examining the elements in the collection. If it finds one that matches the desired value, it will invoke a separate function, *removeAt*, that removes the value held at a specific location. You will complete this implementation in Worksheet 8.

Constructing a Bag using a Linked List

To construct a Bag using the idea of a Linked List, we will begin with the list deque abstraction you developed in Chapter 3. Recall that this implementation used a sentinel at both ends and double links.



The *contains* function must use a loop to cycle over the chain of links. Each element is tested against the argument. If any are equal, then the Boolean value true is returned. Otherwise, if the loop terminates without finding any matching element, the value False is returned.

The *remove* function uses a similar loop. However, this time, if a matching value is found, then the function *removeLink* is invoked. The remove function then terminates, without examining the rest of the collection. (As a consequence, only the first occurrence of a value is removed. Repeated values may still be in the collection. A question at the end of this chapter asks you to consider different implementation techniques for the *removeAll* function.)

Introduction to the Iterator

As we noted in Chapter 1, one of the primary design principles for collection classes is *encapsulation*. The internal details concerning how an implementation works are hidden behind a simple and easy to remember interface. To use a Bag, for example, all you need know is the basic operations are *add*, *collect* and *remove*. The inner workings of the implementation for the bag are effectively hidden.

When using collections, a common requirement is the need to loop over all the elements in the collection, for example to print them to a window. Once again it is important that this process be performed without any knowledge of how the collection is represented in memory. For this reason, the conventional solution is to use a mechanism termed an *Iterator*.

```
/* conceptual interface */
 Boolean (or int) hasNext ( );
 TYPE next ( );
 void remove ( );
```

Each collection will be matched with a set of functions that implement this interface. The functions *next* and *hasNext* are used in combination to write a simple loop that will cycle over the values in the collection.

The iterator loop exposes nothing regarding the structure of the container class. The function *remove* can be used to delete from the collection the value most recently returned by next.

```
LinkedListIterator itr;
TYPE current;
...
ListIteratorInit (aList, itr);
while (ListIteratorHasNext(itr)) {
  current = ListIteratorNext(itr);
        ...
      /* do something with current */
    }
```

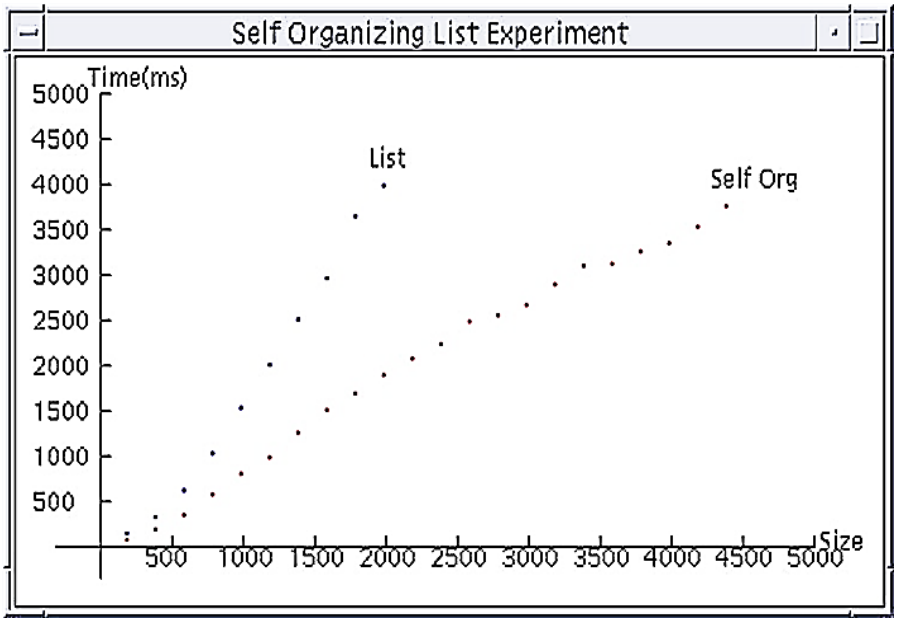
Notice that an iterator is an object that is separate from

the collection itself. The iterator is a *facilitator object*, that provides access to the container values. In worksheets 10 and 24, you complete the implementation of iterators for the dynamic array and for the linked list.

Self-Organizing Lists

We have treated all list operations as if they were equally likely, but this is not always true in practice. Often an analysis of the frequency of operations will suggest ways that a data structure can be modified in order to improve performance. For example, one common situation is that a successful search will frequently be followed relatively soon by a search for the same value. One way to handle this would be for a successful search to remove the value from the list and reinsert it at the front. By doing so, the subsequent search will be much faster.

A data structure that tries to optimize future performance based on the frequency of past operations is called *self-organizing*. We will subsequently encounter a number of other self-organizing data structures. Given the right circumstances self-organization can be very effective. The following chart shows the results of one simple experiment using this technique.



Simple Set Operations

Any bag can be used to construct a set. Among the basic functions, the only change is in the addition operation, which must check that the value is not already in the collection. Operations such as set union, intersection and difference can all be implemented with simple loops. The following pseudo-code shows set union:

```
setUnion (one, two, three) /* assume set
three is initially empty */
for each element in one
if value is found in two
add to set three
```

The algorithmic execution time for this operation depends upon a combination of the time to search the second set, and the time to add into the third set. If we use a simple dynamic array or linked list bag, then both of these operations are O(n) and so, since the outer loop cycles over all the elements in the first bag, the complete union operation is $O(n^2)$.

Simple algorithms for set intersection, difference, and subset are similar, with similar performance. These are analyzed in questions at the end of this chapter. Faster set algorithms require adding more structure to the collection, such as keeping elements in sorted order.

The Bit Set

A specialized type of set is used to represent positive integers from a small range, such as a set drawn from the values between 0 and 75. Because only integer values are allowed, this can be represented very efficiently as binary values, and is therefore termed a *bit set*. Worksheet 12 explores the implementation of the bit set.

Advanced Bag Implementation Techniques

Several other bag implementation techniques use radically different approaches. Indeed, three of the following chapters will be devoted in some fashion or another to bag data

structures. Chapter 5 explores the saving possible when elements are maintained in order. Chapter 6 introduces a new type of list, the skip list, as well as an entirely different form of organization, the binary tree. Finally, Chapter 7 introduces yet another technique, hashing, which produces some of the fastest implementations of bag operations.

Self-Study Questions

1. What features characterize the Bag data type?
2. How is a set different from a bag?
3. When a dynamic array is used as a bag, how is the add operation different from operations you have already implemented in the dynamic array stack?
4. When a dynamic array is used as a set, how is the add operation different from operations you have already implemented in the dynamic array stack?
5. Using a dynamic array as the implementation technique, what is the big-oh algorithmic execution time for each of the bag operations? Is this different if you use a linked list as the implementation technique?
6. What is an iterator? What purpose does the iterator address?
7. Any bag can be used as the basis for a set. When using a dynamic array or linked list as the bag, what is the

algorithmic execution time for the set operations?

8. What is a bit set?

Short Exercises

1. Assume you wanted to define an equality operator for bags. A bag is equal to another bag if they have the same number of elements, and each element occurs in the other bag the same number of times. Notice that the order of the elements is unimportant. If you implement bags using a dynamic array, how would you implement the equality- testing operator? What is the algorithmic complexity of your operation? Can you think of any changes to the representation that would make this operation faster?

Analysis Exercises

1. There are two simple approaches to implementing bag structures; using dynamic arrays or linked list. The only difference in algorithmic execution time performance between these two data structures is that the linked list has guaranteed constant time insertion, while the dynamic array only has amortized constant time insertion. Can you design an experiment that will determine if there is any measurable difference caused

by this? Does your experiment yield different results if a test for inclusion is performed more frequently that insertions?

2. What should the remove function for a bag do if no matching value is identified? One approach is to silently do nothing. Another possibility is to return a Boolean flag indicating whether or not removal was performed. A third possibility is to throw an exception or assertion error. Compare these three possibilities and give advantages and disadvantages of each.
3. Finish describing the naïve set algorithms that are built on top of bag abstractions. Then, using invariants, provide an informal proof of correctness for each of these algorithms.
4. Assume that you are assigned to test a bag implementation. Knowing only the bag interface, what would be some good example test cases? What are some of the boundary values identified in the specification?
5. The entry for Bag in the *Dictionary of Algorithms and Data Structures* suggests a number of axioms that can be used to characterize a bag. Can you define test cases

that would exercise each of these axioms? How do your test cases here differ from those suggested in the previous question?

6. Assume that you are assigned to test a set implementation. Knowing only the set interface, what would be some good example test cases? What are some of the boundary values identified in the specification?

7. The bag data type allows a value to appear more than once in the collection. The remove operation only removes the first matching value it finds, potentially leaving other occurrences of the value remaining in the collection. Suppose you wanted to implement a removeAll operation, which removed all occurrences of the argument. If you did so by making repeated calls on remove what would be the algorithmic complexity of this operation? Can you think of a better approach if you are using a dynamic array as the underlying container? What about if you are using a linked list?

Programming Projects

1. Implement an experiment designed to test the two simple bag implementations, as described in analysis exercise 1.
2. Using either of the bag implementations implement the simple set algorithms and empirically verify that they are $O(n^2)$. Do this by performing set unions for two sets with n elements each, of various values of n. Verify that as n increases the time to perform the union increases as a square. Plot your results to see the quadratic behavior.

On the Web

The wikipedia has (at the time of writing) no entry for the bag data structure, but does have an entry for the related data type termed the *multiset*. Other associated entries include bitset (termed a bit array or bitmap). The *Dictionary of Algorithms and Data Structures* does have an entry that describes the bag.

Chapter-5

Searching, Ordered Collections

SEARCHING, ORDERED COLLECTIONS

In Chapter 4, we said that a *bag* was a data structure characterized by three primary operations. These operations were inserting a new value into the bag, testing a bag to see if it contained a particular value, and removing an element from the bag. In our initial description, we treated all three operations as having equal importance.

In many applications, however, one or another of the three operations may occur much more frequently than the others. In this situation it may be useful to consider alternative implementation techniques. Most commonly the favored operation is searching.

We can illustrate an example with the questions examined previously in previous chapters. Why do dictionaries or telephone books list their entries in sorted order? To understand the reason, we pose the following two questions. Suppose you were given a telephone book and asked to find the number for an individual named Chris Smith. Now suppose you were asked to find the name of the person who has telephone number 543- 7352. Which task is easier?

```
Abby Smith   954-9845
Chris Smith  234-7832
Fred Smith   435-3545
Jaimie Smith 845-2395
Robin Smith  436-9834
```

The difference in the two tasks represents the difference between a *sequential*, or *linear* search and a *binary search*. A linear search is basically the approach used in our Bag abstraction, and the approach that you by necessity need to perform if all you have is an unsorted list. Simply compare the element you seek to each value in the collection, element by element, until either you find the value you want, or exhaust the collection.

A binary search, on the other hand, is much faster, but works only for ordered lists. You start by comparing the test value to the element in the middle of the collection. In one step you can eliminate half the collection, continuing the search with either the first half or the last half of the list. If you repeat this halving idea, you can search the entire list in O (log n) steps.

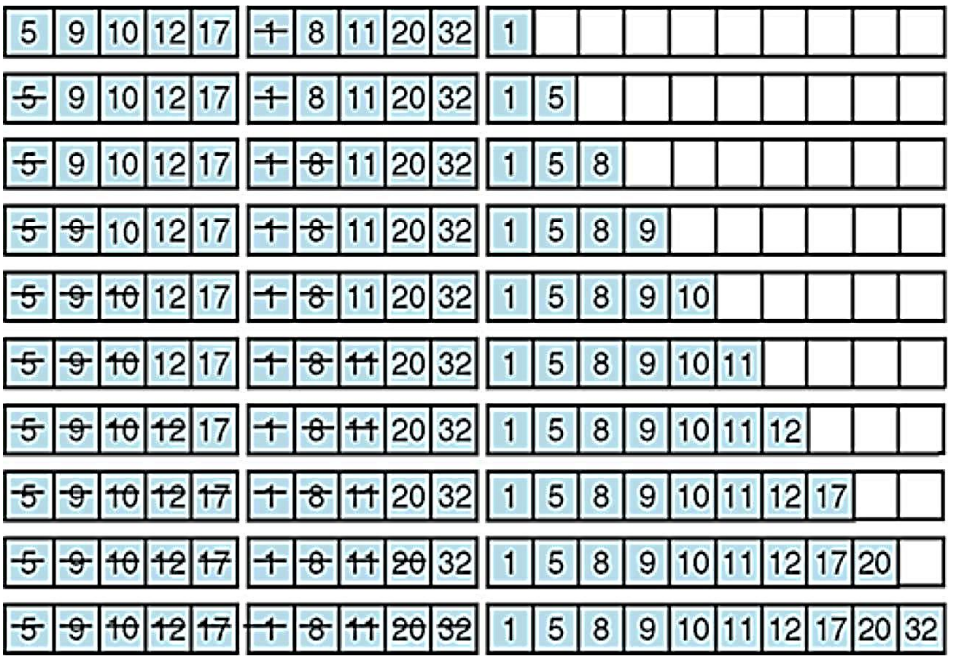
As the thought experiment with the telephone book shows, binary search is much faster than linear search. An ordered array of one billion elements can be searched using

no more than twenty comparisons using a binary search.

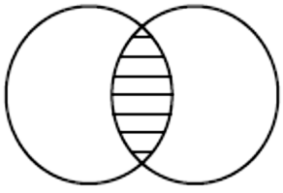
Before a collection can be searched using a binary search it must be placed in order. There are two ways this could occur. Elements can be inserted, one by one, and ordered as they are entered. The second approach is to take an unordered collection and rearrange the values so that they become ordered. This process is termed *sorting*, and you have seen several sorting algorithms already in earlier chapters. As new data structures are introduced in later chapters, we will also explore other sorting algorithms.

Fast Set Operations

Keeping a dynamic array in sorted order allows a fast search by means of the binary search algorithms. But there is another reason why you might want to keep a dynamic array, or even a linked list, in sorted order. This is that two sorted collections can be *merged* very quickly into a new, also sorted collection. Simply walk down the two collections in order, maintaining an index into each one. At each step select the smallest value, and copy this value into the new collection, advancing the index. When one of the pointers reaches the end of the collection, all the values from the remaining collection are copied.



The merge operation by itself is sometimes a good enough reason to keep a sorted collection. However, more often this technique is used as a way to provide fast set operations. Recall that the *set* abstraction is similar to a bag, with two important differences. First, elements in a set are unique, never repeated. Second, a set supports a number of collections with collection operations, such as set union, set intersection, set difference, and set subset.



All of the set operations can be viewed as a variation on the idea of merging two ordered collections. Take,

for example, set intersection. To form the intersection simply walk down the two collections in order. If the element in the first is smaller than that in the second, advance the pointer to the first. If the element in the second is smaller than that in the first, advance the pointer to the second. Only if the elements are both equal is the value copied into the set intersection.

This approach to set operations can be implemented using either ordered dynamic arrays, or ordered linked lists. In practice ordered arrays are more often encountered, since an ordered array can also be quickly searched using the binary search algorithm.

Implementation of Ordered Bag using a Sorted Array

In an earlier chapter you encountered the *binary search* algorithm. The version shown below takes as argument the value being tested, and returns in O(log n) steps either the location at which the value is found, *or if it is not in the collection* the location the value can be inserted and still preserve order.

```
int binarySearch (TYPE * data, int size,
                  TYPE testValue) {
```

```
int low = 0;
 int high = size;
 int mid;
 while (low < high) {
    mid = (low + high) / 2;
    if (LT(data[mid], testValue)) low = mid + 1;
     else high = mid;
 }
 return low;
}
```

Consider the following array, and trace the execution of the binary search algorithm as it searches for the element 7:

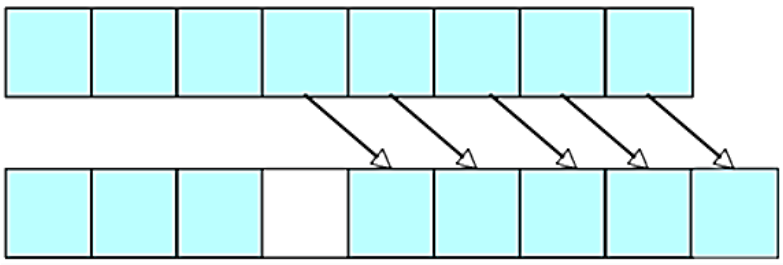
2	4	5	7	8	12	24	37	40	41	42	50	68	69	72

Do the same for the values 1, 25, 37 and 76. Notice that the value returned by this function need not be a legal index. If the test value is larger than all the elements in the array, the only position where an insertion could be performed and still preserve order is the next index at the top. Thus, the binary search algorithm might return a value equal to size, which is not a legal index.

If we used a dynamic array as the underlying container, and if we kept the elements in sorted order, then we could use the binary search algorithm to perform a very

rapid contains test. Simply call the binary search algorithm, and save the resulting index position. Test that the index is legal; if it is, then test the value of the element stored at the position. If it matches the test argument, then return true. Otherwise return false. Since the binary search algorithm is O(log n), and all other operations are constant time, this means that the contains test is O(log n), which is much faster than either of the implementations you developed in the preceding chapter.

Inserting a new value into the ordered array is not quite as easy. True, we can discover the position where the insertion should be made by invoking the binary search algorithm. But then what? Because the values are stored in a block, the problem is in many ways the opposite of the one you examined in Chapter 4. Now, instead of moving values down to delete an entry, we must here move values up to make a “hole” into which the new element can be placed:



As we did with *remove*, we will divide this into two steps. The add function will find the correct location at which to insert a value, then call another function that will insert an element at a given location:

```
void orderedArrayAdd (struct dyArray *dy,
                      TYPE newElement){
    int index = binarySearch(dy->data, dy->size,
    newElement);
   dyArrayAddAt (dy, index, newElement);
}
```

The method addAt must check that the size is less than the capacity, calling setCapacity if not, loop over the elements in the array in order to open up a hole for the new value, insert the element into the hole, and finally update the variable count so that it correctly reflects the number of values in the container.

```
void dyArrayAddAt (struct dyArray *dy, int index,
                   TYPE newElement) {
      int i;
      assert(index > 0 && index <= dy->size);
      if (dy->size >= dy->capacity)
      _dyArraySetCapacity(dy, 2 * dy->capacity);
      …
```

```
        /* you get to fill this in */
    }
```

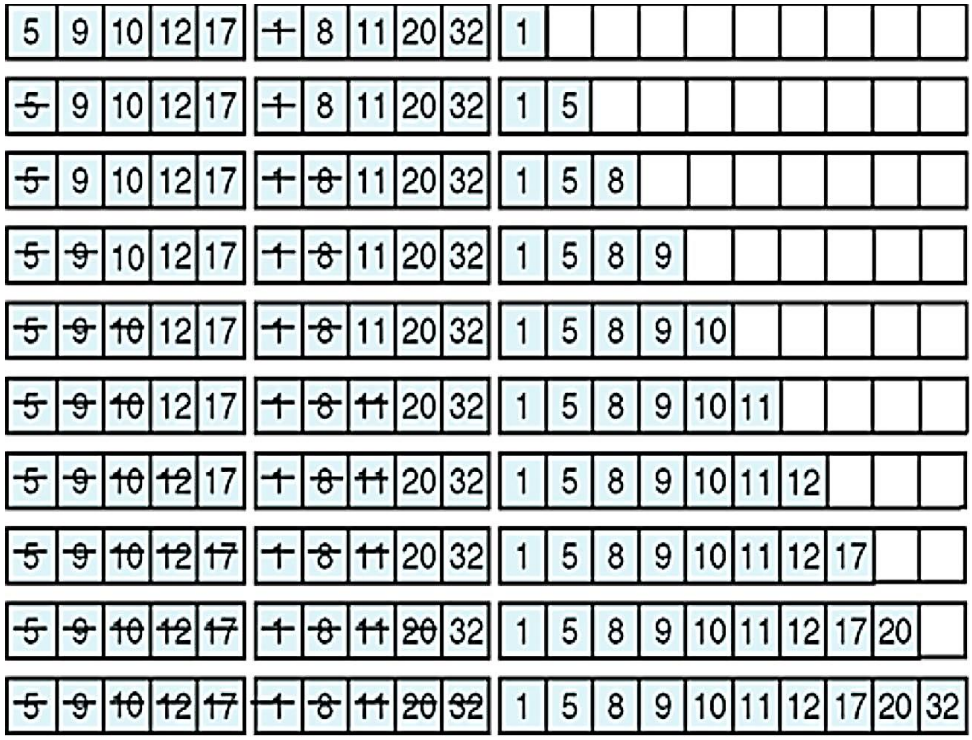
The function *remove* could use the same implementation as you developed in Chapter 4. However, whereas before we used a linear search to find the position of the value to be deleted, we can here use a binary search. If the index returned by binary search is a legal position, then invoke the method *removeAt* that you wrote in Chapter 4 to remove the value at the indicated position.

In Worksheet 13 you will complete the implementation of a bag data structure based on these ideas.

Fast Set Operations for Ordered Arrays

Two sorted arrays can be easily merged into a new array. Simply walk down both input arrays in parallel, selecting the smallest element to copy into the new array shown below:

The set operations of union, intersection, difference and subset are each very similar to a merge. That is, each algorithm can be expressed as a parallel walk down the two input collections, advancing the index into the collection with the smallest value, and copying values into a new vector.




```
void arraySetIntersect (struct dyArray *left,
  struct dyArray *right, struct dyArray *to) {
   int i = 0;
   int j = 0;
   while ((i < dyArraySize(left)) && (j <
   dyArraySize(right))) {
if (LT(dyArrayGet(left, i), dyArrayGet(right, j))
{
  i++;
}
else if (EQ(dyArrayGet(left, i),
dyArrayGet(right, j))) {
  dyArrayAdd(to, dyArrayGet(left, i));
  i++;
```

```
  j++;
}
 else {
 j++;
}
}
}
```

Take, for example, set intersection. The intersection copies a value when it is found in both collections. Notice that in this abstraction it is more convenient to have the set operations create a new set, rather than modifying the arguments. Union copies the smaller element when they are unequal, and when they are equal copies only one value and advances both pointers (remember that in a set all elements are unique, each value appears only once). The difference copies values from the first collection when it is smaller than the current element in the second, and ignores elements that are found in both collections. Finally, there is the subset test. Unlike the others this operation does not produce a new set, but instead returns false if there are any values in the first collection that are not found in the second. But this is the same as returning false if the element from the left set is ever the smallest value (indicating it is not found in the other set).

Question: The parallel walk halts when one or the other array reaches the end. In the merge algorithm there was an additional loop after the parallel walk needed to copy the remaining values from the remaining array. This additional step is necessary in some but not all of the set operations. Can you determine which operations need this step?

In Worksheet 14, you will complete the implementation of the sorted array set based on these ideas. In Chapter 4, you have developed *set* algorithms that made no assumptions concerning the ordering of elements. Those algorithms each have $O(n^2)$ behavior, where n represents the number of elements in the resulting array. What will be the algorithmic execution times for the new algorithms?

	Dynamic Array Set	Sorted Array Set
unionWith	$O(n^2)$	O(
intersectionWith	$O(n^2)$	O(
differenceWith	$O(n^2)$	O(
subset	$O(n^2)$	O(

Set operations Using Ordered Linked Lists

We haven't seen ordered linked lists yet for the simple reason that there usually isn't much reason to maintain linked lists in order. Searching a linked list, even an ordered one, is

still a sequential operation and is therefore O(n). Adding a new element to such a list still requires a search to find the appropriate location, and there therefore also O(n). And removing an element involves finding it first, and is also O(n). Since none of these are any better than an ordinary unordered linked list, why bother?

One reason is the same as the motivation for maintaining a dynamic array in sorted order. We can quickly merge two ordered linked lists to produce a new list that is also sorted. And, as we discovered in the last worksheet, the set operations of intersection, union, difference and subset can all be thought of as simple variations on the idea of a merge. Thus, we can quickly make fast implementations of all these operations as well.

The motivation for keeping a list sorted is not the same as it was for keeping a vector sorted. With a vector one could use binary search quickly find whether a collection contained a specific value. The sequential nature of a linked list prevents the use of the binary search algorithm (but not entirely, as we will see in a later chapter).

Self-Study Questions

1. What two reasons are identified in this chapter for keeping the elements of a collection in sorted order?
2. What is the algorithmic execution time for a binary search? What is the time for a linear search?
3. If an array contains n values, what is the range of results that the binary search algorithm might return?
4. The function dyArrayGet produced an assertion error if the index value was larger than or equal to the array size. The function dyArrayAddAt, on the other hand, allows the index value to be equal to the size. Explain why. (Hint: How many locations might a new value be inserted into an array).
5. Explain why the binary search algorithm speeds the test operation, but not additions and removals.
6. Compare the algorithm execution times of an ordered array to an unordered array. What bag operations are faster in the ordered array? What operations are slower?
7. Explain why merging two ordered arrays can be performed very quickly.
8. Explain how the set operations of union and intersection can be viewed as variations on the idea of

merging two sets.

Short Exercises

1. Show the sequence of index values examined when a binary search is performed on the following array, seeking the value 5. Do the same for the values 14, 41, 70 and 73.

Analysis Exercises

1. The binary search algorithm as presented here continues to search until the range of possible insertion points is reduced to one. If you are searching for a specific value, however, you might be lucky and discover it very quickly, before the values low and high meet. Rewrite the binary search algorithm so that it halts if the element examined at position mid is equal to the value being searched for. What is the algorithmic complexity of your new algorithm? Perform an experiment where you search for random values in a given range. Is the new algorithm faster or slower than the original?
2. Provide invariants that could be used to produce a proof of correctness for the binary search algorithm. Then provide the arguments used to join the

invariants into a proof.

3. Provide invariants that could be used to produce a proof of correctness for the set intersection algorithm. Then provide the arguments used to join the invariants into a proof.
4. Do the same for the intersection algorithm.
5. Two sets are equal if they have exactly the same elements. Show how to test equality in O(n) steps if the values are stored in a sorted array.
6. A set is considered a subset of another set if all values from the first are found in the second. Show how to test the subset condition in O(n) steps if the values are stored in a sorted array.
7. The binary search algorithm presented here finds the midpoint using the formula (low+ high) / 2. Recently Google reported finding an error that was traced to this formula, but occurred only when numbers were close to the maximum integer size. Explain what error can occur in this situation. This problem is easily fixed by using the alternative formula low + (high – low) / 2. Verify that the values that cause problems with the first formula now work with the second.

Programming Projects

1. Rewrite the binary search algorithm so that it halts if it finds an element that is equal to the test value. Create a test harness to test your new algorithm, and experimentally compare the execution time to the original algorithm.
2. Write the function to determine if two sets are equal, as described in an analysis exercise above. Write the similar function to determine if one set is a subset of the second.
3. Create a test harness for the sorted dynamic array bag data structure. Then create a set of test cases to exercise boundary conditions. What are some good test cases for this data structure?

On the Web

Wikipedia contains a good explanation of binary search, as well as several variations on binary search. Binary search is also explained in the *Dictionary of Algorithms and Data Structures.*

Chapter-6

Efficient Collections (skip lists, trees)

EFFICIENT COLLECTIONS (SKIP LISTS, TREES)

If you performed the analysis exercises in Chapter 5, you discovered that selecting a bag- like container required a detailed understanding of the tasks the container will be expected to perform. Consider the following chart:

Function	Dynamic array	Linked list	Ordered
add	O(1)+	O(1)	O(n)
contains	O(n)	O(n)	O(log n)
remove	O(n)	O(n)	O(n)

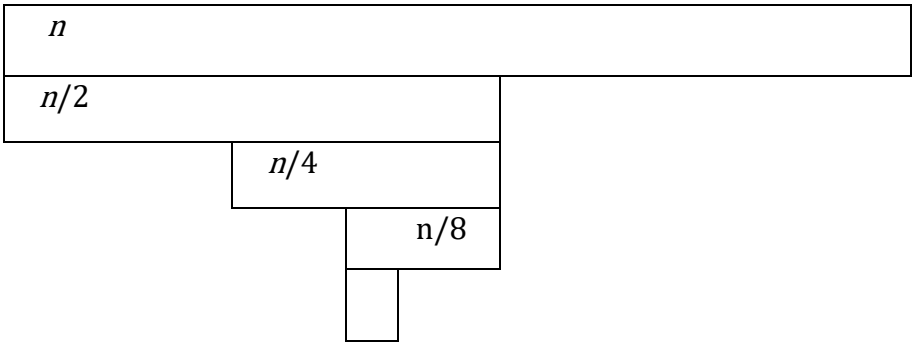
If we are simply considering the cost to insert a new value into the collection, then nothing can beat the constant time performance of a simple dynamic array or linked list. But if searching or removals are common, then the O (log n) cost of searching an ordered list may more than make up for the slower cost to perform an insertion.

Imagine, for example, an on-line telephone directory. There might be several million search requests before it becomes necessary to add or remove an entry. The benefit of being able to perform a binary search more than makes up for the cost of a slow insertion or removal.

What if all three bag operations are more-or-less equal? Are there techniques that can be used to speed up all three operations? Are arrays and linked lists the only ways of organizing a data for a bag? Indeed, they are not. In this chapter we will examine two very different implementation techniques for the Bag data structure. In the end they both have the same effect, which is providing O (log n) execution time for all three bag operations. However, they go about this task in two very different ways.

Achieving Logarithmic Execution

In previous chapters, we discussed the log function. There we noted that one way to think about the log was that it represented "the number of times a collection of n elements could be cut in half". It is this principle that is used in binary search. You start with n elements, and in one step you cut it in half, leaving n/2 elements. With another test you have reduced the problem to n/4 elements, and in another you have n/8. After approximately log n steps you will have only one remaining value.



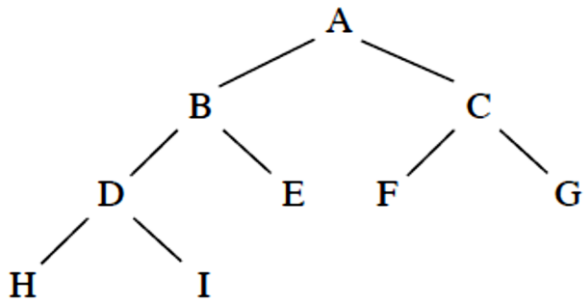
There is another way to use the same principle. Imagine that we have an organization based on layers. At the bottom layer there are n elements. Above each layer is another that has approximately half the number of elements in the one below. So that, the next to the bottom layer has approximately $n/2$ elements, the one above that approximately $n/4$ elements, and so on. It will take approximately $\log n$ layers before we reach a single element.

How many elements are there altogether? One way to answer this question is to note that the sum is a finite approximation to the series $n + n/2 + n/4 + \ldots$. If you factor out the common term n, then this is $1 + 1/2 + 1/4 + \ldots$ This well-known series has a limiting value of 2. This tells us that the structure we have described has approximately 2n elements in it.

The *skip list* and the *binary tree* use these observations in very different ways. The first, the skip list, makes use of

non-determinism. Non-determinism means using random chance, like flipping a coin. If you flip a coin once, you have no way to predict whether it will come up heads or tails. But if you flip a coin one thousand times, then you can confidently predict that about 500 times it will be heads, and about 500 times it will be tails. Put another way, randomness in the small is unpredictable, but in large numbers randomness can be very predictable. It is this principle that casinos rely on to ensure they can always win.

The second technique makes use of a data organization technique we have not yet seen, the binary tree. A binary tree uses nodes, which are very much like the links in a linked list. Each node can have zero, one, or two children.



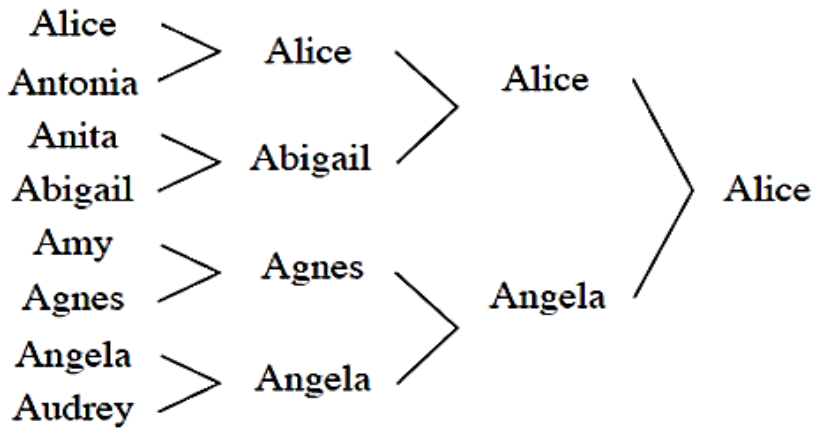
Compare this picture to the earlier one. Look at the tree in levels. At the first level (termed the *root*) there is a single node. At the next level there can be at most two, and the next level there can be at most four, and so on. If a tree is relatively "full" (a more precise definition will be given later), then if it has n

nodes the height is approximately log n.

Tree Introduction

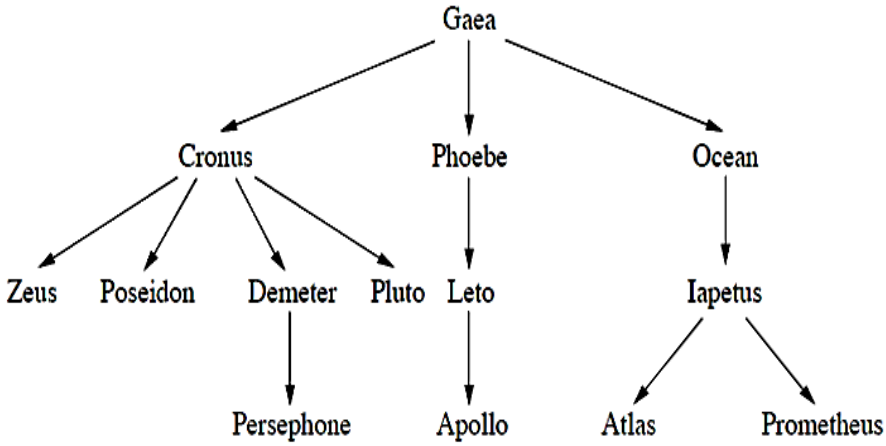
Trees are the third most used data structure in computer science, after arrays (including dynamic arrays and other array variations) and linked lists. They are ubiquitous, found everywhere in computer algorithms.

Just as the intuitive concepts of a stack or a queue can be based on everyday experience with similar structures, the idea of a tree can also be found in everyday life, and not just the arboreal variety. For example, sport events are often organized using binary trees as shown below, with each node representing a pairing of contestants, the winner of each round advancing to the next level.

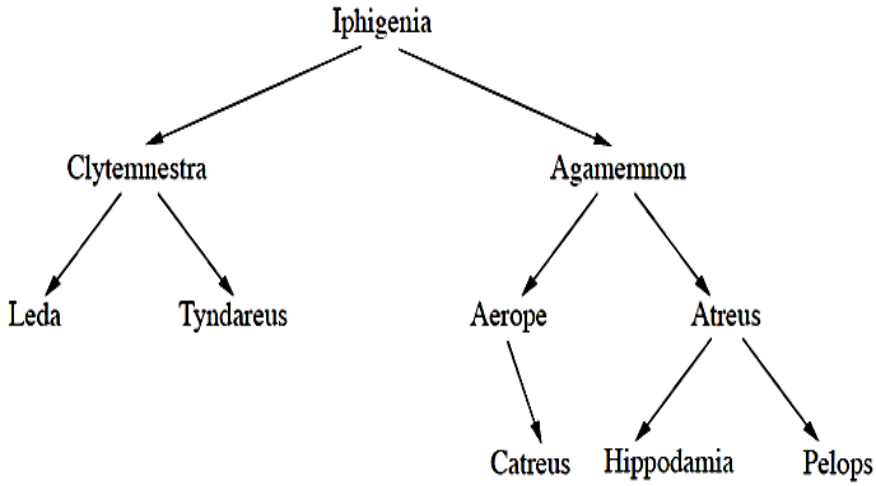


Information about ancestors and descendants is often organized into a tree structure. The following is a typical family

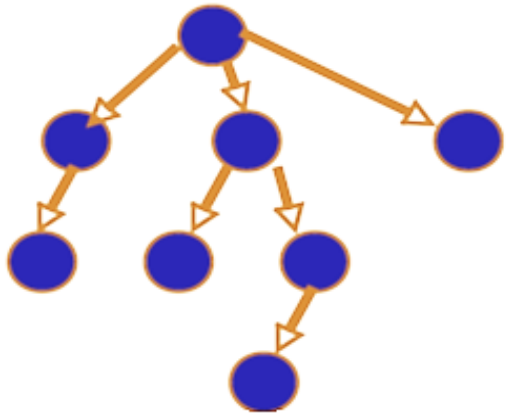
tree.



In a sense, the inverse of a family tree is an ancestor tree. While a family tree traces the descendants from a single individual, an ancestor tree records the ancestors. An example is the following. We could infer from this tree, for example, that Iphigenia is the child of Clytemnestra and Agamemnon, and Clytemnestra is in turn the child of Leda and Tyndareus.



The general characteristics of trees can be illustrated by these examples. A tree consists of a collection of *nodes* connected by directed *arcs*. A tree has a single *root node*. A node that points to other nodes is termed the *parent* of those nodes while the nodes pointed to are the *children*. Every node except the root has exactly one parent. Nodes with no children are termed *leaf nodes*, while nodes with children are termed *interior nodes*. Identify the root, children of the root, and leaf nodes in the following tree.

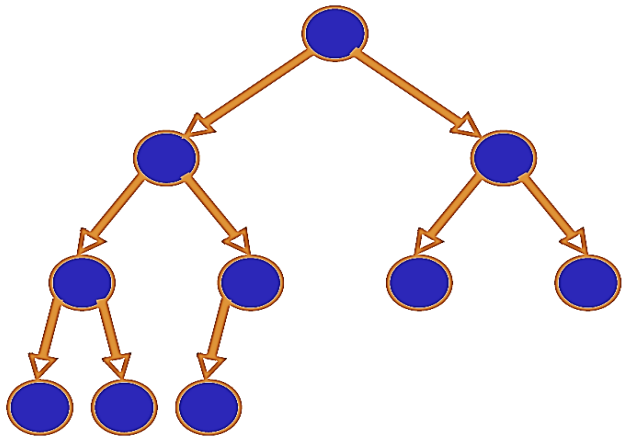


There is a single unique path from the root to any node; that is, arcs don't join together. A path's length is equal to the number of arcs traversed. A node's height is equal to the maximum path length from that node to a left node. A leaf node has height 0. The height of a tree is the height of the root. A nodes depth is equal to the path length from root to that node. The root has depth 0.

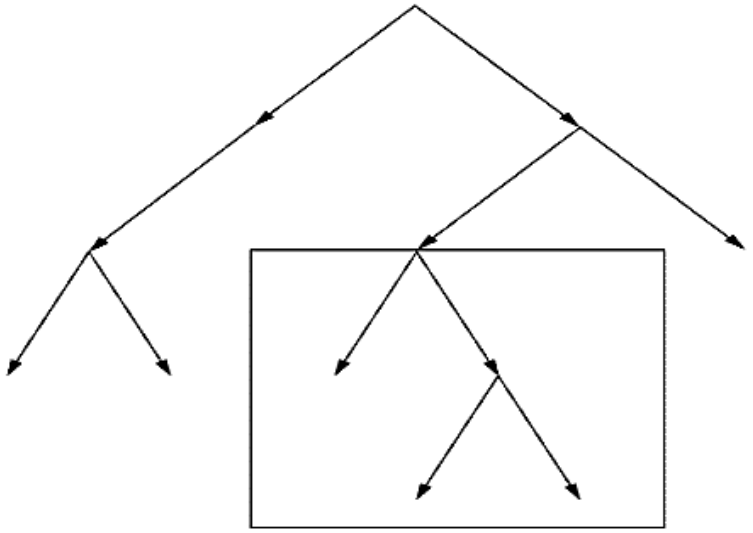
A *binary tree* is a special type of tree. Each node has at most two children. Children are either left or right.

Question: Which of the trees given previously represent a binary tree?

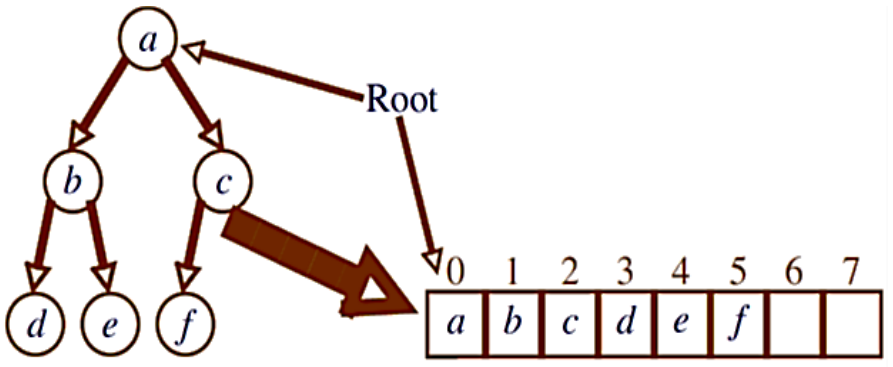
A *full* binary tree is a binary tree in which every leaf is at the same depth. Every internal node has exactly 2 children. Trees with a height of n will have $2^{n+1} - 1$ nodes, and will have 2^n leaves. A *complete* binary tree is a full tree except for the bottom level, which is filled from left to right. How many nodes can there be in a complete binary tree of height n? If we flip this around, what is the height of a complete binary tree with n nodes?



Trees are a good example of a *recursive* data structure. Any node in a tree can be considered to be a root of its own tree, consisting of the values below the node.



A binary tree can be efficiently stored in an array. The *root* is stored at position 0, and the children of the node stored at position i are stored in positions 2i+1 and 2i+2. The parent of the node stored at position i is found at position floor((i-1)/2), as in the example shown below:

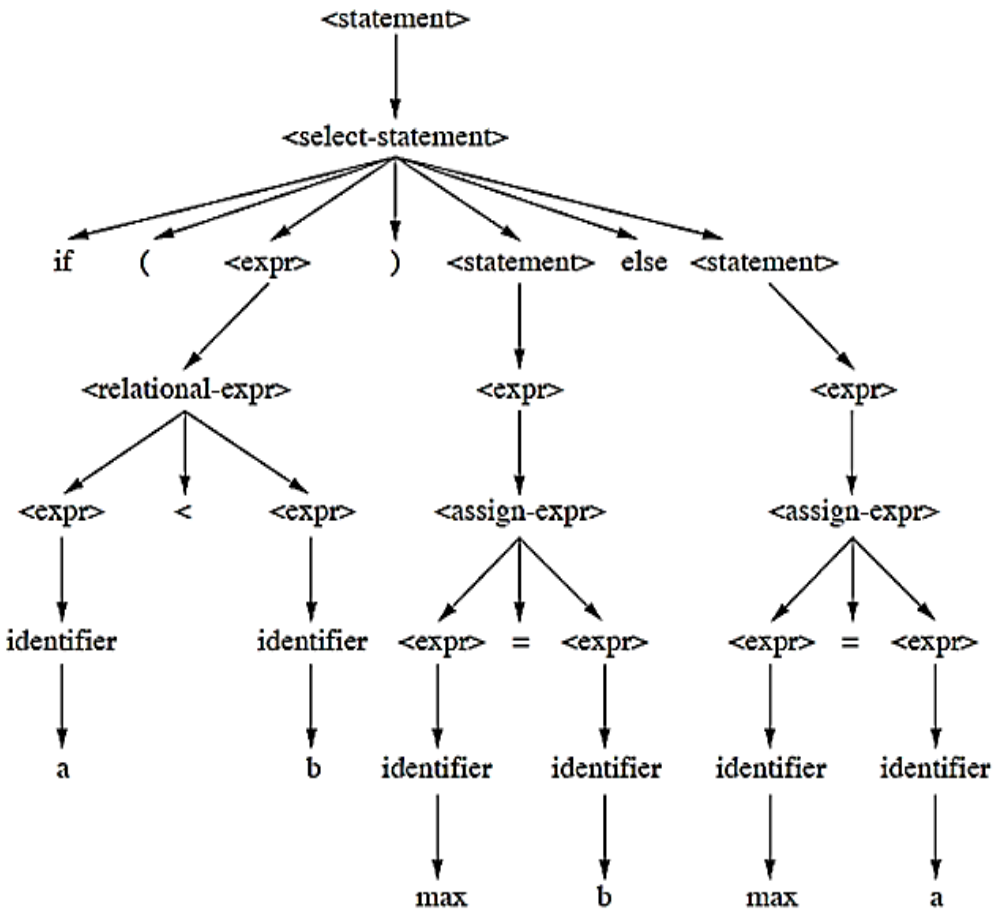


Question: What can you say about a complete binary tree stored in this representation? What will happen if the tree is not complete?

More commonly we will store our binary trees in instances of class Node. This is very similar to the Link class we used in the linked list. Each node will have a value, and a left and right child.

```
struct node {
  EleType value;
  struct node * left;
  struct node * right;
}
```

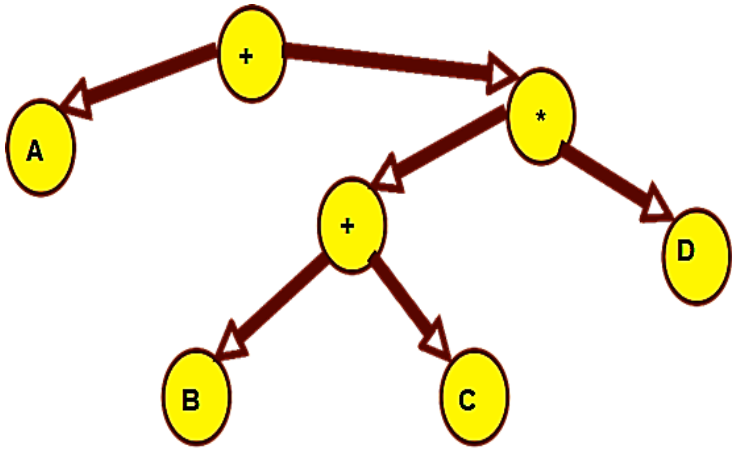
Trees appear in computer science in a surprising large number of varieties and forms. A common example is a parse tree. Computer languages, such as C, are defined in part using a grammar. Grammars provide rules that explain how the tokens of a language can be put together. A statement of the language is then constructed using a tree, such as the one shown below. In this tree, leaf nodes represent the tokens, or symbols, used in the statement, whereas interior nodes represent syntactic categories.



Syntax Trees and Polish Notation

A tree is a common way to represent an arithmetic expression. For example, the following tree represents the expression A + (B + C) * D. As you learned in Chapter 1, Polish notation is a way of representing expressions that avoids the need for parentheses by writing the operator for an expression first. The polish notation form for this expression

is + A * + B C D. Reverse polish writes the operator after an expression, such as A B C + D * +. Describe the results of each of the following three traversal algorithms on the following tree, and give their relationship to polish notation.



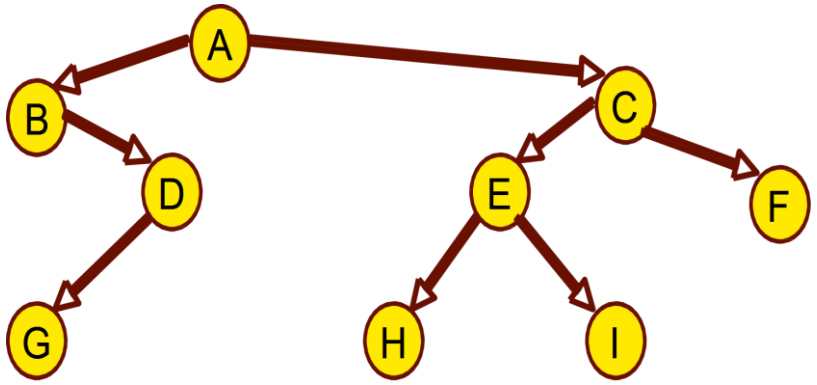
Tree Traversals

Just as with a list, it is often useful to examine every node in a tree in sequence. This is a termed a *traversal*. There are four common traversals:

- ***Preorder***: Examine a node first, then left children, then right children
- ***Inorder***: Examine left children, then a node, then right children
- ***Postorder***: Examine left children, then right children, then a node

- ***Levelorder***: Examine all nodes of depth n first, then nodes depth n+1, etc

Question: Using the following tree, describe the order that nodes would be visited in each of the traversals.



In practice, the *inorder* traversal is the most useful. As you might have guessed when you simulated the inorder traversal on the above tree, the algorithm makes use of an internal stack. This stack represents the current path that has been traversed from root to left. Notice that the first node visited in an inorder traversal is the leftmost child of the root. A useful function for this purpose is slideLeft.

```
slideLeft (node n)
while n is not null
add n to stack
    n = left child of n
```

Using the slide left routine, you should verify that the following algorithm will produce a traversal of a binary search tree. The algorithm is given in the form of an iterator, consisting of tree parts: initialization, test for completion, and returning the next element:

```
initialization:
create an empty stack
has next:
 if stack is empty
 perform slide left on root
 otherwise
  let n be top of stack.
  Pop topmost element
  slide left on right child of n
return true if stack is not empty
current element:
 return value of top of stack
```

You should simulate the traversal on the tree given earlier to convince yourself it is visiting nodes in the correct order. Although the inorder traversal is the most useful in practice, there are other tree traversal algorithms. Simulate each of the following, and verify that they produce the desired

traversals. For each algorithm characterize (that is, describe) what values are being held in the stack or queue.

```
PreorderTraversal:
initialize an empty stack
has next
if stack is empty
then push the root on to the stack
otherwise
pop the topmost element of the stack,
and push the children from left to right
return true if stack is not empty
current element:
return the value of the current top of stack
PostorderTraversal:
intialize a stack by performing a slideLeft
from the root
has Next
if top of stack has a right child
 perform slide left from right child
return true if stack is not empty
current element
pop stack and return value of node
```

```
LevelorderTraversal:
initialize an empty queue
has Next
if queue is empty then push root in to the queue
otherwise
 pop the front element from the queue
 and push the children from right to left in to the queue return true if queue is not empty
current element
return the value of the current front of the queue
```

Euler Tours

The three common tree-traversal algorithms can be unified into a single algorithm by an approach that visits every node three times. A node will be visited before its left children (if any) are explored, after all left children have been explored but before any right child, and after all right children have been explored. This traversal of a tree is termed an *Euler tour*. An Euler tour is like a walk around the perimeter of a binary tree.

```
void EulerTour (Node n ) {
```

```
 beforeLeft(n);
if (n.left != null) EulerTour (n.left);
inBetween (n);
if (n.right != null) EulerTour (n.right);
afterRight(n);
}
void beforeLeft (Node n) { … }
void inBetween (Node n) { … }
void afterRight (Node n) { … }
```

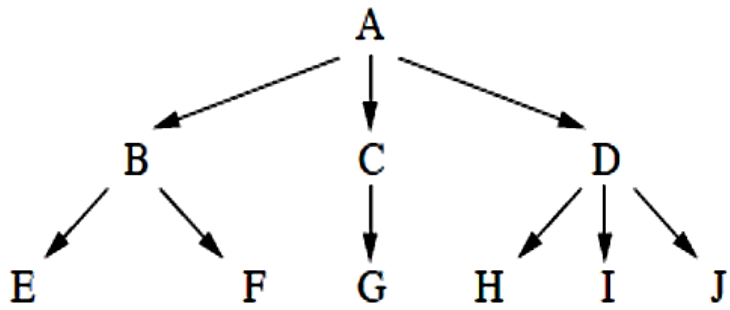
The user constructs an Euler tour by providing implementations of the functions beforeLeft, inBetweeen and afterRight. To invoke the tour the root node is passed to the function EulerTour. For example, if a tree represents an arithmetic expression the following could be used to print the representation.

```
void beforeLeft (Node n) { print("("); }
void inBetween (Node n) { printl(n.value); }
void afterRight (Node n) { print(")"); }
```

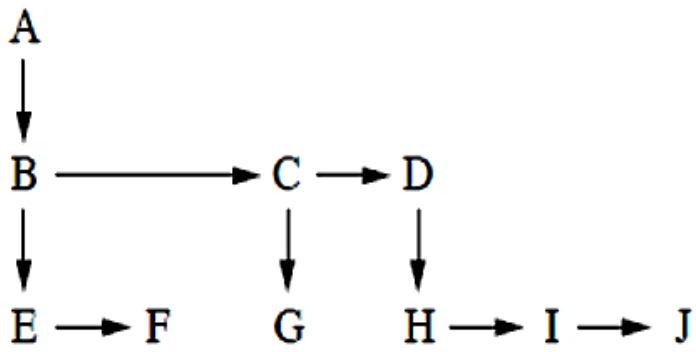
Binary Tree Representation of General Trees

The binary tree is actually more general than you might first imagine. For instance, a binary tree can

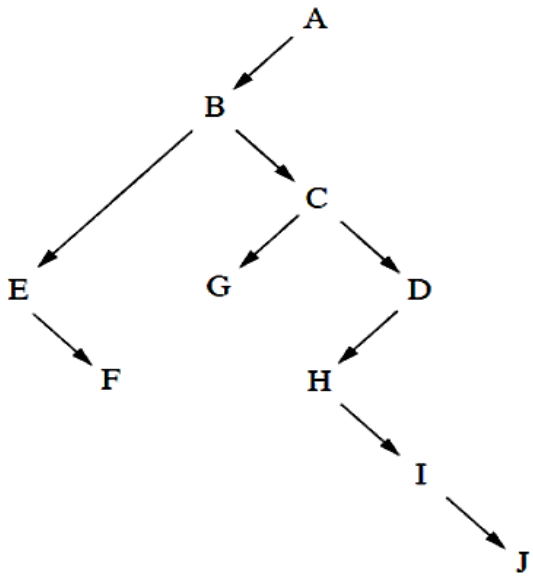
actually represent any tree structure. To illustrate this, consider an example tree such as shown below.



To represent this tree using binary nodes, use the left pointer on each node to indicate the first child of the current node, then use the right pointer to indicate a "sibling", a child with the same parents as the current node. The tree would thus be represented as follows:



Turning the tree 45 degrees makes the representation look more like the binary trees we have been examining in earlier parts of this chapter:



Question: Try each of the tree traversal techniques described earlier on this resulting tree. Which algorithms correspond to a traversal of the original tree?

Efficient Logarithmic Implementation Techniques

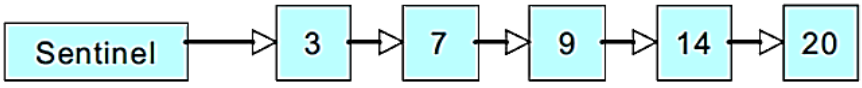
In the worksheets, you will explore two different efficient (that is, logarithmic) implementation techniques. These are the *skip list* and the *balanced binary tree*.

The Skip List

The SkipList is a more complex data structure than we have seen up to this point, and so we will spend more time in development and walking you through the implementation. To motivate the need for the skip list, consider that ordinary linked lists and dynamic arrays have fast (O(1)) addition of

new elements, but a slow time for search and removal. A sorted array has a fast O (log n) search time, but is still slow in addition of new elements and in removal. A skip list is fast O (log n) in all three operations.

We begin the development of the skip list by first considering a simple ordered list, with a sentinel on the left, and single links:



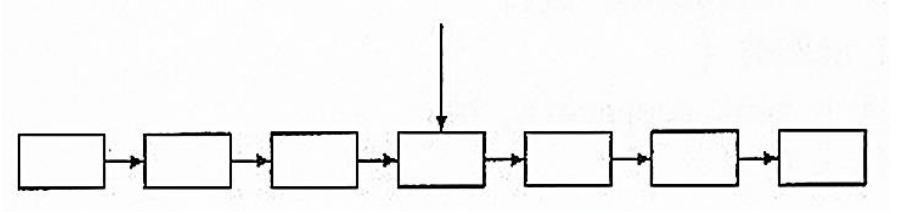
To add a new value to such a list you find the correct location, then set the value. Similarly, to see if the list contains an element, you find the correct location, and see if it is what you want. And to remove an element, you find the correct location, and remove it. Each of these three has in common the idea of finding the location at which the operation will take place. We can generalize this by writing a common routine, named **slideRight**. This routine will move to the right as long as the next element is smaller than the value being considered.

```
slide right (node n, TYPE test) {
while (n-> next != null and n->next->value
        < test)  n = n->next;
return n;
}
```

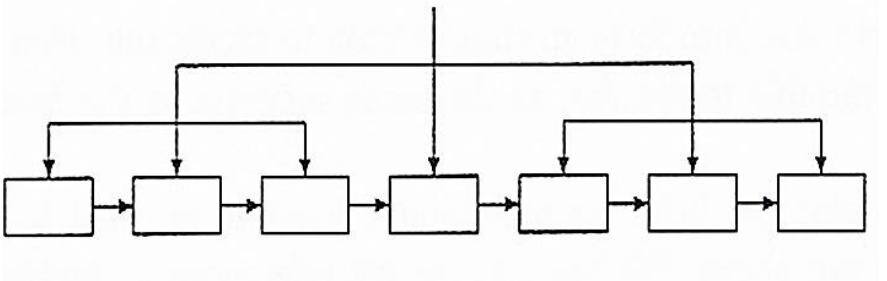
Try simulating some of the operations on this structure using the list shown until you understand what the function slideRight is doing. What happens when you insert the value 10? Search for 14? Insert 25? Remove the value 9?

By itself, this new data structure is not particularly useful or interesting. Each of the three basic operations still loop over the entire collection, and are therefore $O(n)$, which is no better than an ordinary dynamic array or linked list.

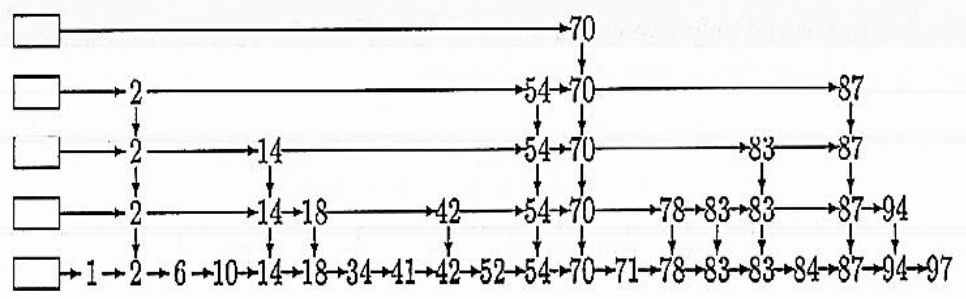
Why can't one do a binary search on a linked list? The answer is because you cannot easily reach the middle of a list. But one could imagine keeping a pointer into the middle of a list:



And then another, an another, until you have a tree of links.



In theory this would work, but the effort and cost to maintain the pointers would almost certainly dwarf the gains in search time. But what if we didn't try to be precise, and instead maintained links *probabistically*? We could do this by maintaining a stack of ordered lists, and links that could point down to the lower link. We can arrange this so that each level has approximately half the links of the next lower. There would therefore be approximately log n levels for a list with n elements.



This is the basic idea of the skip list. Because each level has half the elements as the one below, the height is approximately log n. Because operations will end up being proportional to the height of the structure, rather than the number of elements, they will also be O (log n). Worksheet 15 will lead through the implementation of the skip list.

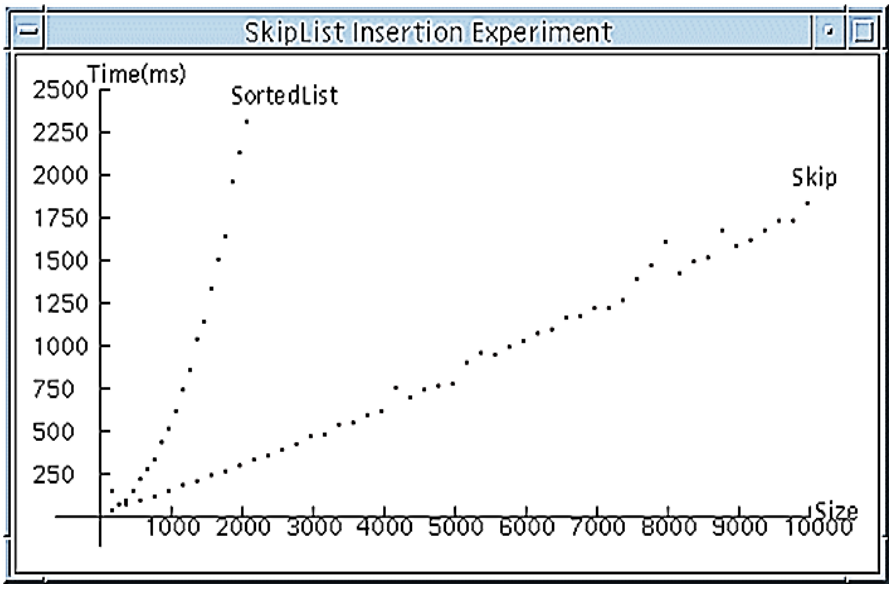
Other data structures have provided fast implementations of one or more operations. An ordinary dynamic array had fast insertion; a sorted array provided a fast

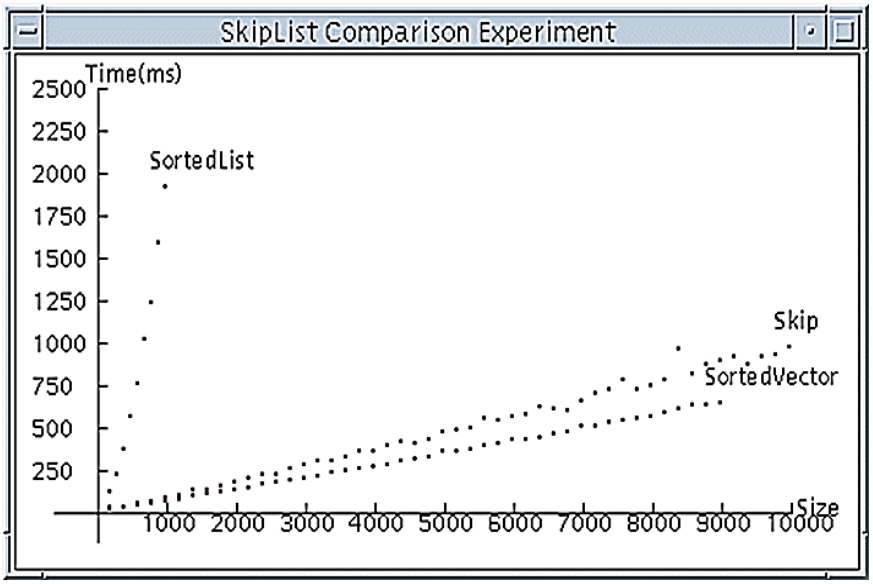
search. The skip list is the first data structure we have seen that provides a fast implementation of all three of the Bag operations. This makes it a very good general purpose data structure. The one disadvantage of the skip list is that it uses about twice as much memory as needed by the bottommost linked list. In later lessons we will encounter other data structures that also have fast execution time, and use less memory.

The skip list is the first data structure we have seen that uses *probability*, or *random chance*, as a technique to ensure efficient performance. The use of a coin toss makes the class non-deterministic. If you twice insert the same values into a skip list, you may end up with very different internal links. Random chance used in the right way can be a very powerful tool.

In one experiment to test the execution time of the skip list we compared the insertion time to that of an ordered list. This was done by adding n random integers into both collections. The results were as shown in the first graph below. As expected, insertion into the skip list was much faster than insertion into a sorted list. However, when comparing two operations with similar time the results are more complicated. For example, the second graph compares

searching for elements in a sorted array versus in a skip list. Both are O (log n) operations. The sorted array may be slightly faster, however this will in practice be offset by the slower time to perform insertions and removals.



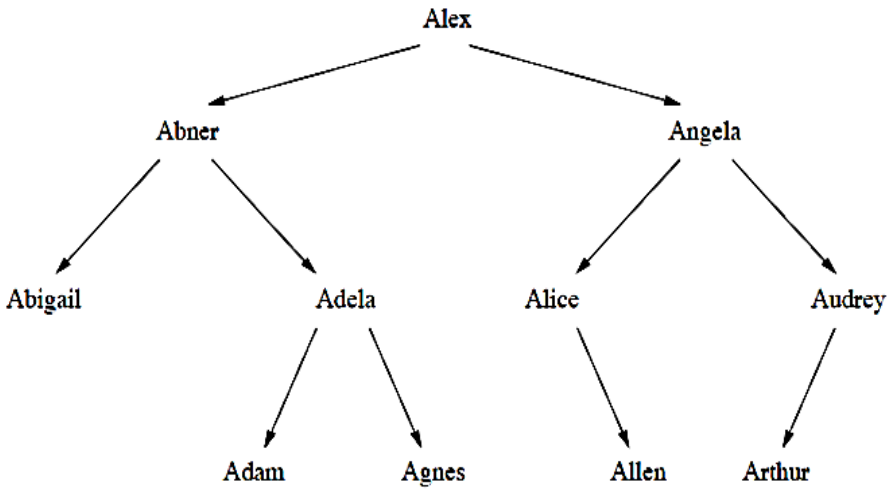


The bottom line says that to select an appropriate container one must consider the entire mix of operations a task will require. If a task requires a mix of operations the skip list is a good overall choice. If insertion is the dominant operation, then a simple dynamic array or list might be preferable. If searching is more frequent than insertions then a sorted array is preferable.

The Binary Search Tree

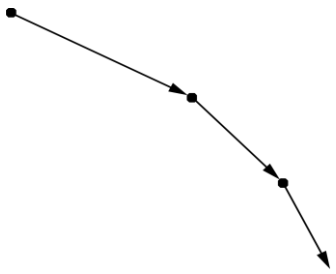
As we noted earlier in this chapter, another approach to finding fast performance is based on the idea of a binary tree. A binary tree consists of a collection of nodes. Each node can have zero, one or two children. No node can be pointed to by more than one other node. The node that points to another node is known as the parent node.

To build a collection out of a tree we construct what is termed a binary search tree. A *binary search tree* is a binary tree that has the following additional property: for each node, the values in all descendants to the left of the node are less than or equal to the value of the node, and the values in all descendants to the right are greater than or equal. The following is an example binary search tree:



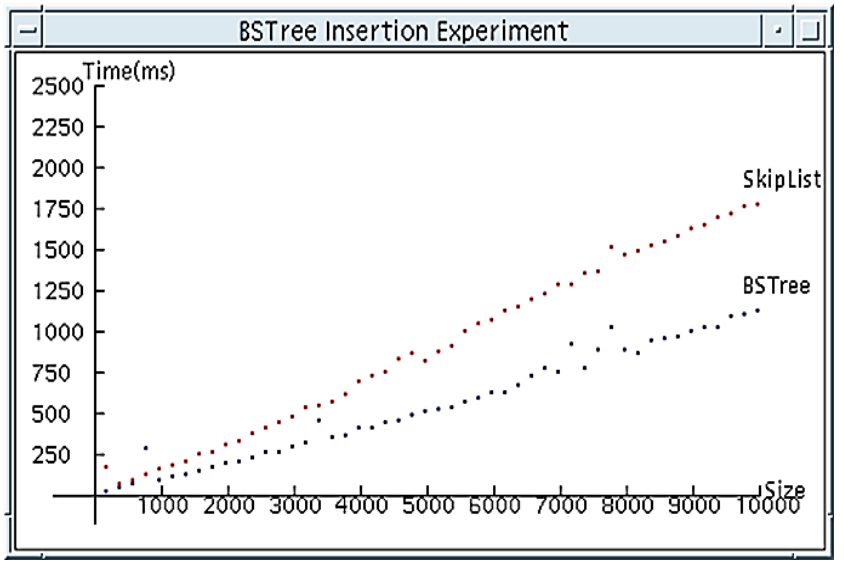
Notice that an inorder traversal of a BST will list the elements in sorted order. The most important feature of a binary search tree is that operations can be performed by walking the tree from the top (the root) to the bottom (the leaf). This means that a BST can be used to produce a fast bag implementation. For example, suppose you wish to find out if the name “Agnes” is found in the tree shown. You simply compare the value to the root (Alex). Since Agnes comes before Alex, you travel down the left child. Next you compare “Agnes” to “Abner”. Since it is larger, you travel down the right. Finally, you find a node that matches the value you are searching, and so you know it is in the collection. If you find a null pointer along the path, as you would if you were searching for “Sam”, you would know the value was not in the collection.

Adding a value to a binary search tree is easy. You simply perform the same type of traversal as described above, and when you find a null value, you insert a new node. Try inserting the value “Amina”. Then try inserting “Sam”.



The development of a bag abstraction based on these ideas occurs in two worksheets. In Worksheet 16 you explore the basic algorithms. Unfortunately, bad luck in the order in which values are inserted into the bag can lead to very poor performance. For example, if elements are inserted in order, then the resulting tree is nothing more than a simple linked list. In Worksheet 17 you explore the AVL tree, which rebalances the tree as values are inserted in order to preserve efficient performance.

As long as the tree remains relatively well balanced, the addition of values to a binary search tree is very fast. This can be seen in the execution timings shown below. Here the time required to place n random values into a collection is compared to a Skip List, which had the fastest execution times we have seen so far.



Functional versus State Change Data Structures

In developing the AVL tree, you create a number of functions that operate differently than those we have seen earlier. Rather than making a change to a data structure, such as modifying a child field in an existing node, these functions leave the current value unchanged, and instead create a new value (such as a new subtree) in which the desired modification has been made (for example, in which a new value has been inserted). The methods **add**, **removeLeftmostChild**, and **remove** illustrate this approach to the manipulation of data structures. This technique is often termed the *functional* approach, since it is common in functional programming languages, such as ML and Haskell. In many situations it is easier to describe how to build a new

value than it is to describe how to change an existing value. Both approaches have their advantages, and you should add this new way of thinking about a task to your toolbox of techniques and remember it when you are faced with new problems.

Self-Study Questions

1. In what operations is a simple dynamic array faster than an ordered array? In what operations is the ordered array faster?
2. What key concept is necessary to achieve logarithmic performance in a data structure?
3. What are the two basic parts of a tree?
4. What is a root note? What is a leaf node? What is an interior node?
5. What is the height of a binary tree?
6. If a tree contains a node that is both a root and a leaf, what can you say about the height of the tree?
7. What are the characteristics of a binary tree?
8. What is a full binary tree? What is a complete binary tree?
9. What are the most common traversals of a binary tree?
10. How are tree traversals related to polish notation?
11. What key insight allows a skip list to achieve efficient

performance?

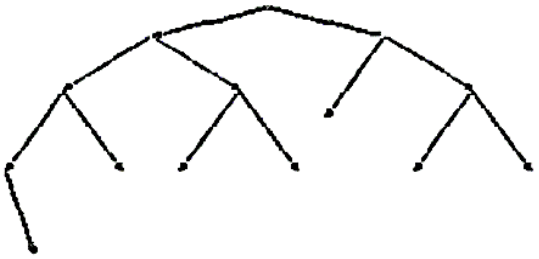
12. What are the features of a binary search tree?
13. Explain the operation of the function slideRight in the skip list implementation. What can you say about the link that this method returns?
14. How are the links in a skip list different from the links in previous linked list containers?
15. Explain how the insertion of a new element into a skip list uses random chance.
16. How do you know that the number of levels in a skip list is approximately log n?

Analysis Exercises

1. Would the operations of the skip list be faster or slower if we added a new level one- third of the time, rather than one-half of the time? Would you expect there to be more or fewer levels? What about if the probability were higher, say two-thirds of the time? Design an experiment to discover the effect of these changes.
2. Imagine you implemented the naïve set algorithms described in Lesson 24, but used a skip list rather than a vector. What would the resulting execution times be?
3. Prove that a complete binary tree of height n will have

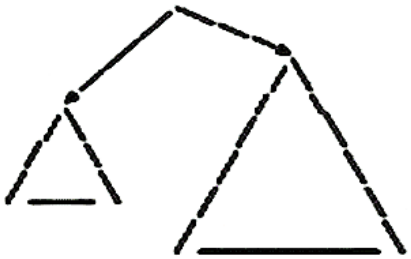
2^n leaves. (Easy to prove by induction).

4. Prove that the number of nodes in a complete binary tree of height n is $2^{n+1} - 1$.
5. Prove that a binary tree containing n nodes must have at least one path from root to leaf of length floor (log n).
6. Prove that in a complete binary tree containing n nodes, the longest path from root to leaf traverses no more than ceil (log n) nodes.
7. So how close to being well balanced is an AVL tree? Recall that the definition asserts the difference in height between any two children is no more than one. This property is termed a *height-balanced tree*. Height balance assures that locally, at each node, the balance is roughly maintained, although globally over the entire tree differences in path lengths can be somewhat larger. The following shows an example height-balanced binary tree.



A complete binary tree is also height balanced. Thus,

the largest number of nodes in a balanced binary tree of height h is 2^{h+1}-1. An interesting question is to discover the *smallest* number of nodes in a height-balanced binary tree. For height zero there is only one tree. For height 1 there are three trees, the smallest of which has two nodes. In general, for a tree of height h the smallest number of nodes is found by connecting the smallest tree of height h-1 and h-2.



If we let M_h represent the function yielding the minimum number of nodes for a height balanced tree of height h, we obtain the following equations:

$$M_0 = 1$$

$$M_1 = 2$$

$$M_{h+1} = M_{h-1} + M_h + 1$$

These equations are very similar to the famous *Fibonacci numbers* defined by the formula $f_0 = 0$, $f_1 = 1$, $f_{n+1} = f_{n-1} + f_n$. An induction argument can be used to show that $M_h = f_{h+3} - 1$. It is easy to show using induction that we can

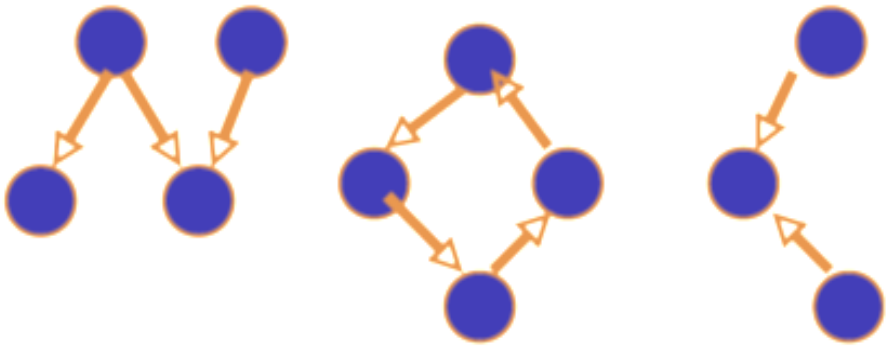
bound the Fibonacci numbers by 2^n. In fact, it is possible to establish an even tighter bounding value. Although the details need not concern us here, the Fibonacci numbers have a closed form solution; that is, a solution defined without using recursion. The value F_h is approximately $\frac{\emptyset^i}{\sqrt{5}}$, where $\emptyset$ is the golden mean value $\frac{1+\sqrt{5}}{2}$, or approximately 1.618. Using this information, we can show that the function M_h also has an approximate closed form solution:

$$M_h \approx \frac{\emptyset^{h+3}}{\sqrt{6}} - 1$$

By taking the logarithm of both sides and discarding all but the most significant terms we obtain the result that h is approximately 1.44 log M_h. This tells us that the longest path in a height-balanced binary tree with n nodes is at worst only 44 percent larger than the log n minimum length. Hence algorithms on height-balanced binary trees that run in time proportional to the length of the path are still O(log n). More importantly, preserving the height balanced property is considerably easier than maintaining a completely balanced tree.

Because AVL trees are fast on all three bag operations

they are a good general purpose data structure useful in many different types of applications. The Java standard library has a collection type, TreeSet, that is based on a data type very similar to the AVL trees described here. Explain why the following are not legal trees.



Programming Projects

1. The bottom row of a skip list is a simple ordered list. You already know from Lesson 36 how to make an iterator for ordered lists. Using this technique, create an iterator for the skip list abstraction. How do you handle the remove operation?

2. The smallest value in a skip list will always be the first element. Assuming you have implemented the iterator described in question 6, you have a way of accessing this value. Show how to use a skip list to implement a priority queue. A priority queue, you will recall,

provides fast access to the smallest element in a collection. What will be the algorithmic execution time for each of the priority queue operations?

```
void skipPQaddElement (struct skipList *,
                       TYPE newElement);
TYPE skipPQsmallestElement (struct skipList *);
void skipPQremoveSmallest (struct skipList *);
```

On the Web

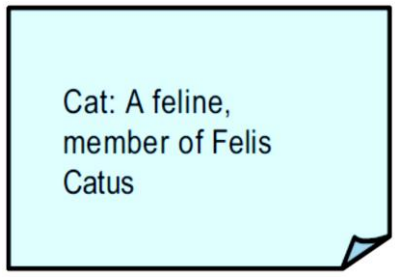
The wikipedia has a good explanation of various efficient data structures. These include entries on skip lists, AVL trees and other self-balancing binary search trees. Two forms of tree deserve special note. Red/Black trees are more complex than AVL trees, but require only one bit of additional information (whether a node is red or black), and are in practice slightly faster than AVL trees. For this reason, they are used in many data structure libraries. B-trees (such as 2-3 B trees) store a larger number of values in a block, similar to the dynamic array block. They are frequently used when the actual data is stored externally, rather than in memory. When a value is accessed, the entire block is moved back into memory.

CHAPTER-7

DICTIONARY (HASH TABLES)

DICTIONARY (HASH TABLES)

In the containers we have examined up to now, the emphasis has been on the values themselves. A bag, for example, is used to hold a collection of elements. You can add a new value to a bag, test to see whether or not a value is found in the bag, and remove a value from the bag. In a *dictionary*, on the other hand, we separate the data into two parts. Each item stored in a dictionary is represented by a key/value pair. The key is used to access the item. With the key you can access the value, which typically has more information.



The name itself suggests the metaphor used to understand this abstract data type. Think of a dictionary of the English language. Here a word (the key) is matched with a definition (the value). You use the key to find the definition. But the connection is one way. You typically cannot search a dictionary using a definition to find the matching word.

A keyed container, such as a dictionary, can be contrasted with an indexed container, such as an array.

Index values are a form of key; however, they are restricted to being integers and must be drawn from a limited range: usually zero to one less than the number of elements. Because of this restriction the elements in an array can be stored in a contiguous block; and the index can be used as part of the array offset calculation. In an array, the index need not be stored explicitly in the container. In a dictionary, on the other hand, a key can be any type of object. For this reason, the container must maintain both the key and its associated value.

The Dictionary ADT

The abstract data type that corresponds to the dictionary metaphor is known by several names. Other terms for keyed containers include the names *map*, *table*, *search table*, *associative array*, or *hash*. Whatever it is called, the idea is a data structure optimized for a very specific type of search. Elements are placed into the dictionary in key/value pairs. To do a retrieval, the user supplies a key, and the container returns the associated value. Each key identifies one entry; that is, each key is unique. However, nothing prevents two different keys from referencing the same value. The contains test is in the dictionary replaced by a test to see if a given key is legal.

Finally, data is removed from a dictionary by specifying the key for the data value to be deleted.

As an ADT, the dictionary is represented by the following operations:

`get(key)`	Retrieve the value associated with the given key.
`put(key, value)`	Place the key and value association into the dictionary
`containsKey(key)`	Return true if key is found in dictionary
`removeKey(key)`	Remove key from association
`keys()`	Return iterator for keys in dictionary
`size()`	Return number of elements in dictionary

The operation we are calling put is sometimes named set, insertAt, or atPut. The get operation is sometimes termed at. In our containers a get with an invalid key will produce an assertion error. In some variations on the container this operation will raise an exception, or return a special value, such as null. We include an iterator for the key set as part of the specification, but will leave the implementation of this feature as an exercise for the reader. The following illustrates some of the implementations of the

dictionary abstraction found in various programming languages.

Operation	C++ Map<keytype, valuetype>	Java HashMap<keytype, valuetype>	C# hashtable
`get`	`Map[key]`	`Get(key)`	`Hash[key]`
`put`	`Insert(key, value)`	`Put(key, value)`	`Add(key, value)`
`containsKey`	`Count(key)`	`containsKey(key)`	
`removeKey`	`Erase(key)`	`Remove(key)`	

Applications of the Dictionary data type

Maps or dictionaries are useful in any type of application where further information is associated with a given key. Obvious examples include dictionaries of word/definition pairs, telephone books of name/number pairs, or calendars of date/event-description pairs. Dictionaries are used frequently in analysis of printed text. For example, a concordance examines each word in a text, and constructs a list indicating on which line each word appears.

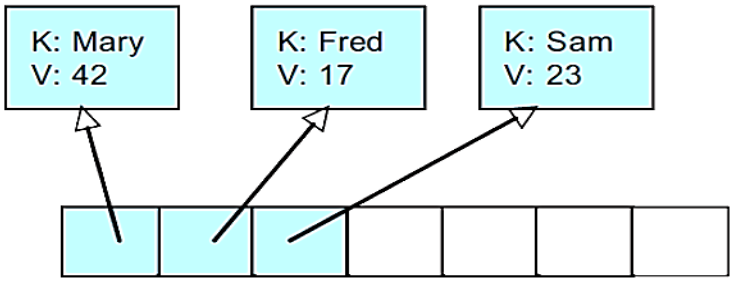
Implementations of the Dictionary ADT

We will describe several different approaches to implementing the dictionary data type. The first has the advantage of being easy to describe, however it is relatively slow. The remaining implementation techniques will introduce a new way of organizing a collection of data values. This technique is termed hashing, and the container built using this technique is called a hash table.

The concept of hashing that we describe here is useful in implementing both Bag and Dictionary type collections. Much of our presentation will use the Bag as an example, as dealing with one value is easier than dealing with two. However, the generalization to a Dictionary rather than a Bag is straightforward, and will be handled in programming projects described at the end of the chapter.

A Dictionary Built on top of a Bag

The basic idea of the first implementation approach is to treat a dictionary as simply a bag of key/value pairs as shown below. We will introduce a new name for this pair, calling it an *Association*. An Association is in some ways similar to a link. Instances of the class Association maintain a key and value field.



But we introduce a subtle twist to the definition of the type Association. When one association is compared to another, the result of the comparison is determined solely by the key. If two keys are equal, then the associations are considered equal. If one key is less than another, then the first association is considered to be smaller than the second.

With the assistance of an association, the implementation of the dictionary data type is straightforward. To see if the dictionary contains an element, a new association is created. The underlying bag is then searched. If the search returns true, it means that there is an entry in the bag that matches the key for the collection. Similarly, the remove method first constructs a new association. By invoking the remove method for the underlying bag, any element that matches the new association will be deleted. But we have purposely defined the association so that it tests only the key. Therefore, any element that matches the key will be removed. This type of

Map can be built on top of any of our Bag abstractions.

Hashing Background

We have seen how containers such as the skip list and the AVL tree can reduce the time to perform operations from O(n) to O(log n). But can we do better? Would it be possible to create a container in which the average time to perform an operation was O(1)? The answer is both yes and no. To illustrate how, consider the following story. Six friends; Alfred, Alessia, Amina, Amy, Andy and Anne, have a club. Amy is in charge of writing a program to do bookkeeping. Dues are paid each time a member attends a meeting, but not all members attend all meetings. To help with the programming Amy uses a six-element array to store the amount each member has paid in dues.

Alfred	$2.65	F = 5 % 6 = 5
Alessia	$6.75	E = 4 % 6 = 4
Amina	$5.50	I = 8 % 6 = 2
Amy	$10.50	Y = 24 % 6 = 0
Andy	$2.25	D = 3 % 6 = 3
Anne	$0.75	N = 13 % 6 = 1

Amy uses an interesting fact. If she selects the third letter of each name, treating the letter as a number from 0 to 25, and then divides the number by 6, each name yields a

different number. So in O(1) time Amy can change a name into an integer index value, then use this value to index into a table. This is faster than an ordered data structure, indeed almost as fast as a subscript calculation.

What Amy has discovered is called a *perfect hash function*. A *hash function* is a function that takes as input an element and returns an integer value. Almost always the index used by a hash algorithm is the remainder after dividing this value by the hash table size. So, for example, Amy's hash function returns values from 0 to 25. She divided by the table size (6) in order to get an index.

The idea of *hashing* can be used to create a variety of different data structures. Of course, Amy's system falls apart when the set of names is different. Suppose Alan wishes to join the club. Amy's calculation for Alan will yield 0, the same value as Amy. Two values that have the same hash are said to have *collided*. The way in which collisions are handed is what separates different hash table techniques.

Almost any process that converts a value into an integer can be used as a hash function. Strings can interpret characters as integers (as in Amy's club), doubles can use a portion of their numeric value, structures can use one or more fields. Hash

functions are only required to return a value that is integer, not necessarily positive. So it is common to surround the calculation with abs ( ) to ensure a positive value.

Open Address Hashing

There are several ways we can use the idea of hashing to help construct a container abstraction. The first technique you will explore is termed *open-address hashing*. (Curiously, also sometimes called *closed hashing*). We explain this technique by first constructing a Bag, and then using the Bag as the source for a Dictionary, as described in the first section. When open-address hashing is used all elements are stored in a single large table. Positions that are not yet filled are given a **null** value. An eight-element table using Amy's algorithm would look like the following:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred		Aspen

Notice that the table size is different, and so the index values are also different. The letters at the top show characters that hash into the indicated locations. If Anne now joins the club, we will find that the hash value (namely, 5) is the same as for Alfred. So, to find a location to store the

value Anne we *probe* for the next free location. This means to simply move forward, position by position, until an empty location is found. In this example the next free location is at position 6.

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina			Andy	Alessia	Alfred	Anne	Aspen

No suppose Agnes wishes to join the club. Her hash value, 6, is already filled. The probe moves forward to the next position, and when the end of the array is reached it continues with the first element, in a fashion similar to the dynamic array deque you examined in Chapter 7. Eventually the probing halts, finding position 1:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina	Agnes		Andy	Alessia	Alfred	Anne	Aspen

Finally, suppose Alan wishes to join the club. He finds that his hash location, 0, is filled by Amina. The next free location is not until position 2:

0-aiqy	1-bjrz	2-cks	3-dlt	4-emu	5-fnv	6-gow	7-hpx
Amina	Agnes	Alan	Andy	Alessia	Alfred	Anne	Aspen

We now have as many elements as can fit into this table. The ratio of the number of elements to the table size is known as the *load factor*, written λ. For open address hashing the load factor is never larger than 1. Just as a dynamic array was doubled in size when necessary, a common solution to a full hash table is to move all values into a new and larger table when the load factor becomes larger than some threshold, such as 0.75. To do so a new table is created, and every entry in the old table is rehashed, this time dividing by the new table size to find the index to place into the new table.

To see if a value is contained in a hash table the test value is first hashed. But just because the value is not found at the given location doesn't mean that it is not in the table. Think about searching the table above for the value Alan, for example. Instead of immediately halting, an unsuccessful test must continue to probe, moving forward until either the value is found or an empty location is encountered.

Removing an element from an open hash table is problematic. We cannot simply replace the location with a null entry, as this might interfere with subsequent search operations. Imagine that we replaced Agnes with a null value

in the table given above, and then once more performed a search for Alan. What would happen?

λ	(1/(1-λ))
0.25	1.3
0.5	2.0
0.6	2.5
0.75	4.0
0.85	6.6
0.95	19.0

One solution to this problem is to not allow removals. This is the technique we will use. The second solution is to create a special type of marker termed a *tombstone*. A tombstone replaces a deleted value, can be replaced by another newly inserted value, but does not halt the search.

How fast are hash table operations? The analysis depends upon several factors. We assume that the time it takes to compute the hash value itself is constant. But what about distribution of the integers returned by the hash function? It would be perfectly legal for a hash function to always return the value zero – legal, but not very useful.

The best case occurs when the hash function returns values that are uniformly distributed among all possible index values; that is, for any input value each index is equally likely. In this situation one can show that the number of elements that will be examined in performing an addition, removal or test will be roughly $1/(1 - \lambda)$. For a small load factor this is acceptable, but degrades quickly as the load

factor increases. This is why hash tables typically increase the size of the table if the load factor becomes too large. Worksheet 14 explores the implementation of a bag using open hash table techniques.

Caching

Indexing into a hash table is extremely fast, even faster than searching a skip list or an AVL tree. There are many different ways to exploit this speed. A *cache* is a data structure that uses two levels of storage. One level is simply an ordinary collection class, such as a bag dictionary. The second level is a hash table, used for its speed. The cache makes no attempt to handle the problem of collisions within the hash table. When a search request is received, the cache will examine the hash table. If the value is found in the cache, it is simply returned. If it is not found, then the original data structure is examined. If it is found there, the retrieved item *replaces* the value stored in the cache. Because the new item is now in the cache, a subsequent search for the same value will be very fast.

The concept of a cache is generally associated with computer memory, where the underlying container represents paged memory from a relatively slow device (such

as a disk), and the cache holds memory pages that have been recently accessed. However, the concept of a two-level memory system is applicable in any situation where searching is a more common operation than addition or removal, and where a search for one value means that it is very likely the value will again be requested in the near future.

For example, a cache is used in the interpreter for the language Smalltalk to match a message (function applied to an object) with a method (code to be executed). The cache stores the name and class for the most recent message. If a message is sent to the same class of object, the code is found very quickly. If not, then a much slower search of the class hierarchy is performed. However, after this slower search the code is placed into the cache. A subsequent execution of the same message (which, it turns out, is very likely) will then be much faster.

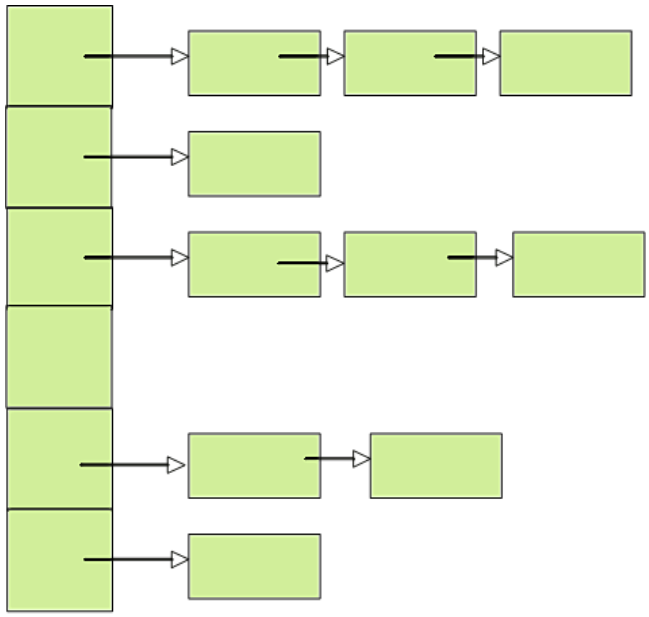
Like the self-organizing linked list (Chapter 4) and the skew heap (worksheet 22), a cache is a self-organizing data structure. That is, the cache tries to improve future performance based on current behavior.

Hash Table with Buckets

In earlier sections you learned about the concept of

hashing, and how it was used in an open address hash table. An entirely different approach to dealing with collisions is the idea of hash tables using buckets.

A hash table that uses buckets is really a combination of an array and a linked list. Each element in the array (the hash table) is a header for a linked list. All elements that hash into the same location will be stored in the list. For the Bag type abstraction, the link stores only a value and a pointer to the next link. For a dictionary type abstraction, such as we will construct, the link stores the key, the value associated with the key, and the pointer to the next link.



Each operation on the hash table divides into two steps. First, the element is hashed and the remainder taken

after dividing by the table size. This yields a table index. Next, linked list indicated by the table index is examined. The algorithms for the latter are very similar to those used in the linked list.

As with open address hash tables, the load factor (λ) is defined as the number of elements divided by the table size. In this structure the load factor can be larger than one, and represents the average number of elements stored in each list, assuming that the hash function distributes elements uniformly over all positions. Since the running time of the contains test and removal is proportional to the length of the list, they are $O(\lambda)$. Therefore, the execution time for hash tables is fast only if the load factor remains small. A typical technique is to resize the table (doubling the size, as with the vector and the open address hash table) if the load factor becomes larger than 10. Hash tables with buckets are explored in worksheet 26.

Bit Set Spell Checker

In Chapter 4 you learned about the bit set abstraction. Also, in that chapter you read how a set can be used as part of a spell checker. A completely different technique for creating a spelling checker is based on using a BitSet in the fashion of

a hash table. Begin with a BitSet large enough to hold t elements. We first take the dictionary of correctly spelled words, hash each word to yield a value h, then set the position h % t. After processing all the words, we will have a large hash table of zero/one values. Now to test any particular word we perform the same process; hash the word, take the remainder after dividing by the table size, and test the entry. If the value in the bit set is zero, then the word is misspelled. Unfortunately, if the value in the table is 1, it may still be misspelled since two words can easily hash into the same location.

To correct this problem, we can enlarge our bit set, and try using more than one hash function. An easy way to do this is to select a table size t, and select some number of prime numbers, for example five, that are larger than t. Call these p_1, p_2, p_3, p_4 and p_5. Now rather than just using the hash value of the word as our starting point, we first take the remainder of the hash value when divided by each of the prime numbers, yielding five different values. Then we map each of these into the table. Thus, each word may set five different bits. This following illustrates this with a table of size 21, using the values 113, 181, 211, 229 and 283 as our prime numbers:

Word	Hashcode	H%113	H%181	H%211	H%229	H%283
This	6022124	15(15)	73(10)	184(16)	111(6)	167(20)
Text	6018573	80(17)	142(16)	9(9)	224(14)	12(12)
Example	1359142016	51(9)	165(18)	75(12)	223(13)	273(0)

1 0 0 0 0 0 1 0 0 1 1 0 1 1 1 1 1 1 1 0 1

The key assumption is that different words may map into the same locations for one or two positions, but not for all five. Thus, we reduce the chances that a misspelled word will score a false positive hit.

Analysis shows that this is a very reliable technique as long as the final bit vector has roughly the same number of zeros as ones. Performance can be improved by using more prime numbers.

Chapter Summary

Whereas bags, stacks, queues and the other data abstractions examined up to this point emphasize the collection of individual values, a dictionary is a data abstraction designed to maintain *pairs* consisting of a key and a value. Entries are placed into the dictionary using both key and value. To recover or delete a value, the user provides only the key.

In this chapter we have explored several implementation techniques that can be used with the dictionary abstraction. Most importantly, we have introduced the idea of a hash table.

To *hash* a value means simply to apply a function that transforms a possibly non-integer key into an integer value. This simple idea is the basis for a very powerful data structuring technique. If it is possible to discover a function that transforms a set of keys via a one-to-one mapping on to a set of integer index values, then it is possible to construct a vector using non-integer keys. More commonly, several key values will map into the same integer index. Two keys that map into the same value are said to have *collided*.

The problem of collisions can be handled in a number of different ways. We have in this chapter explored three possibilities. Using open-address-hashing we probe, or search, for the next free unused location in a table. Using caching, the hash table holds the most recent value requested for the given index. Collisions are stored in a slower dictionary, which is searched only when necessary. Using hash tables with buckets, collisions are stored in a linked list of associations.

The state of a hash table can be described in part by the load factor, which is the number of elements in a table divided by the size of the hash table. For open address hashing, the load factor will always be less than 1, as there can be no more elements than the table size. For hash tables that use buckets, the load factor can be larger than 1, and can be interpreted as the average number of elements in each bucket.

The process of hashing permits access and testing operations that potentially are the fastest of any data structure we have considered. Unfortunately, this potential depends upon the wise choice of a hash function, and luck with the key set. A good hash function must uniformly distribute key values over each of the different buckets. Discovering a good hash function is often the most difficult part of using the hash table technique.

Key concepts

- Map
- Association
- Hashing
- Hash functions
- collisions
- Hash Tables
- Open address hashing

Self-Study Questions

1. In what ways is a dictionary similar to an array? In what ways are they different?
2. What does it mean to hash a value?
3. What is a hash function?
4. What is a perfect hash function?
5. What is a collision of two values?
6. What does it mean to probe for a free location in an open address hash table?
7. What is the load factor for a hash table?
8. Why do you not want the load factor to become too large?
9. In searching for a good hash function over the set of integer elements, one student thought he could use the following:

   ```
   int hash = (int)Math.sin(value);
   ```

   explain why this was a poor choice.

Short Exercises

1. In the dynamic array implementation of the dictionary, the **put** operation removes any prior association with the given key before insertion a new association. An alternative would be to search the list of associations, and if one is found simply replace the

value field. If no association is found, then insert a new association. Write the **put** method using this approach. Which is easier to understand? Which is likely to be faster?

2. When Alan wishes to join the circle of six friends, why can't Amy simply increase the size of the vector to seven?
3. Amy's club has grown, and now includes the following members:

Abel	Abigail	Abraham	Ada
Adam	Adrian	Adrienne	Agnes
Albert	Alex	Alfred	Alice
Amanda	Amy	Andrew	Andy
Angela	Anita	Anne	Antonia
Arnold	Arthur	Audrey	

Find what value would be computed by Amy's hash function for each member of the group.

4. Assume we use Amy's hash function and assign each member to a bucket by simply dividing the hash value by the number of buckets. Determine how many elements would be assigned to each bucket for a hash table of size 5. Do the same for a hash table of size 11.
5. In searching for a good hash function over the set of integer values, one student thought he could use the

following:

```
int index = (int) Math.sin(value);
```

What was wrong with this choice?

6. Can you come up with a perfect hash function for the names of the week? The names of the months? The names of the planets?
7. Examine a set of twelve or more telephone numbers, for example the numbers belonging to your friends. Suppose we want to hash into seven different buckets. What would be a good hash function for your set of telephone numbers? Will your function continue to work for new telephone numbers?
8. Experimentally test the birthday paradox. Gather a group of 24 or more people, and see if any two have the same birthday.
9. Most people find the birthday paradox surprising because they confuse it with the following: if there are *n* people in a room, what is the probability that somebody else shares *your* birthday. Ignoring leap years, give the formula for this expression, and evaluate this when *n* is 24.
10. The function containsKey can be used to see if a dictionary contains a given key. How could you

determine if a dictionary contains a given value? What is the complexity of your procedure?

Analysis Exercises

1. A variation on a map, termed a multi-map, allows multiple entries to be associated with the same key. Explain how the multimap abstraction complicates the operations of search and removal.
2. (The birthday Paradox) The frequency of collisions when performing hashing is related to a well-known mathematical puzzle. How many randomly chosen people need be in a room before it becomes likely that two people will have the same birth date? Most people would guess the answer would be in the hundreds, since there are 365 possible birthdays (excluding leap years). In fact, the answer is only 24 people.

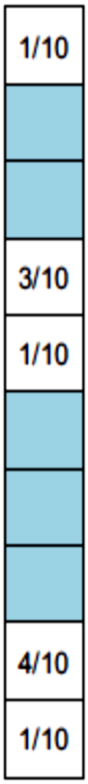
   To see why, consider the opposite question. With n randomly chosen people in a room, what is the probability that no two have the same birth date? Imagine we take a calendar, and mark off each individual's birth date in turn. The probability that the second person has a different birthday from the first is 364/365, since there are 364 different possibilities

not already marked. Similarly, the probability that the third person has a different birthday from the first two is 363/365. Since these two probabilities are independent of each other, the probably that they are *both* true is their product. If we continue in this fashion, if we have n-1 people all with different birthdays, the probability that individual n has a different birthday is: 364/365 * 363/365 * 363/365 * … * 365-n+1/365. When n >= 24 this expression becomes less than 0.5. This means that if 24 or more people are gathered in a room the odds are better than even that two individuals have the same birthday.

The implication of the birthday paradox for hashing is to tell us that for any problem of reasonable size we are almost certain to have some collisions. Functions that avoid duplicate indices are surprisingly rare, even with a relatively large table.

3. **(Clustering)** Imagine that the colored squares in the ten-element table at right indicate values in a hash table that have already been filled. Now assume that the next value will, with equal probability, be any of the ten values. What is the probability that each of the free squares will be filled? Fill in the remaining

squares with the correct probability.



Here is a hint: since both positions 1 and 2 are filled, any value that maps into these locations must go into the next free location, which if 3. So, the probability that square 3 will be filled is the sum of the probabilities that the next item will map into position 1 (1/10) plus the probability that the next item will map into position 2 (which is 1/10) plus the probability that the next item will map into position 3 (also 1/10). So what is the final probability that position 3 will be filled? Continue with this type of analysis for the rest of the squares.

This phenomenon, where the larger a block of filled cells becomes, the more likely it is to become even larger, is known as clustering. (A similar phenomenon explains why groups of cars on a freeway tend to become larger).

Clustering is just one reason why it is important to keep the load factor of hash tables low.

Simply moving to the next free location is known as linear probing. Many alternatives to linear probing have been studied, however as open address hash

tables are relatively rare, we will not examine these alternatives here.

Show that probing by any constant amount will not reduce the problem caused by clustering, although it may make it more difficult to observe since clusters are not adjacent. To do this, assume that elements are uniformly inserted into a seven-element hash table, with a linear probe value of 3.

Having inserted one element, compute the probability that any of the remaining empty slots will be filled. (You can do this by simply testing the values 0 to 6, and observing which locations they will hash into. If only one element will hash into a location, then the probability is 1/7, if two the probability is 2/7, and so on). Explain why the empty locations do not all have equal probability. Place a value in the location with highest probability, and again compute the likelihood that any of the remaining empty slots will be filled. Extrapolate from these observations and explain how clustering will manifest itself in a hash table formed using this technique.

4. You can experimentally explore the effect of the load factor on the efficiency of a hash table. First, using an open address hash table, allow the load factor to reach 0.9 before you reallocate the table. Next, perform the same experiment, but reallocate as soon as the table reaches 0.7. Compare the execution times for various sets of operations. What is the practical effect? You can do the same for the hash table with bucket abstraction, using values larger than 1 for the load factor. For example, compare using the limit of 5 before reallocation to the same table where you allow the lists to grow to length 20 before reallocation.

5. To be truly robust, a hash table cannot perform the conversion of hash value into index using integer operations. To see why, try executing using the standard abs function, and compute and print the absolute value of the smallest integer number. Can you explain why the absolute value of this particular integer is not positive? What happens if you negate the value v? What will happen if you try inserting the value v into the hash table containers you have created in this chapter?

To avoid this problem the conversion of hash value

into index must be performed first as a long integer, and then converted back into an integer.

```
Long longhash = abs((Long) hash(v));

int hashindex = (int) longhash;
```

Verify that this calculation results in a positive integer. Explain how it avoids the problem described earlier.

Programming Projects

1. All our Bag abstractions have the ability to return an iterator. When used in the fashion of Worksheet D1, these iterators will yield a value of type Association. Show how to create a new inner class that takes an iterator as argument, and when requested returns only the key portion of the association. What should this iterator do with the remove operation?
2. Although we have described hash tables in the chapter on dictionaries, we noted that the technique can equally well be applied to create a Bag like container. Rewrite the Hashtable class as a Hashbag that implements Bag operations, rather than dictionary operations.
3. An iterator for the hash table is more complex than an iterator for a simple bag. This is because the iterator

must cycle over two types of collections: the array of buckets, and the linked lists found in each bucket. To do this, the iterator will maintain two values, an integer indicating the current bucket, and a reference to the link representing the current element being returned. Each time hasMore is called the current link is advanced. If there are more elements, the function returns true. If not, then the iterator must advance to the next bucket, and look for a value there. Only when all buckets have been exhausted should the function hasNext return false. Implement an iterator for your hash table implementation.

4. If you have access to a large online dictionary of English words (on unix systems such a dictionary can sometimes be found at /usr/lib/words) perform the following experiment. Add all the words into an open address hash table. What is the resulting size of the hash table? What is the resulting load factor? Change your implementation so that it will keep track of the number of probes used to locate a value. Try searching for a few words. What is the average probe length?

5. Another approach to implementing the Dictionary is to use a pair of parallel dynamic arrays. One array will

maintain the keys, and the other one will maintain the values, which are stored in the corresponding positions.

If the key array is sorted, in the fashion of the SortedArrayBag, then binary search can be used to quickly locate an index position in the key array. Develop a data structure based on these ideas.

6. Individuals unfamiliar with a foreign language will often translate a sentence from one language to another using a dictionary and word-for-word substitution. While this does not produce the most elegant translation, it is usually adequate for short sentences, such as ``Where is the train station?'' Write a program that will read from two files. The first file contains a series of word-for-word pairs for a pair of languages. The second file contains text written in the first language. Examine each word in the text, and output the corresponding value of the dictionary entry. Words not in the dictionary can be printed in the output surrounded by square brackets, as are [these] [words].

7. Implement the bit set spell checker as described earlier in this chapter.

On the Web

The dictionary data structure is described in wikipedia under the entry “Associative Array”. A large section of this entry describes how dictionaries are written in a variety of languages. A separate entry describes the Hash Table data structure. Hash tables with buckets are described in the entry “Coalesced hashing”.

REFERENCES

[1] G. H. Gonnet and R. Baeza-Yates, Handbook of Algorithms and Data Structures in Pascal and C. Reading, Mass.: Addison-Wesley, 1991.

[2] E.M. Reingold and W. J. Hensen, Data Structures in C++, Little Brown and Company, Boston, MA, 2015.

[3] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, Data Structures and Algorithms. Reading, MA: Addison-Wesley, 1983.

[4] R. Bayer, "Symmetric binary B-trees: Data structure and maintenance," Acta Informatica, vol. 1, no. 4, pp. 290–306, 1972.

[5] K. Mehlhorn, Data Structures and Algorithms 1: Sorting and Searching, vol. 1 of EATCS Monographs on Theoretical Computer Science. Heidelberg, Germany: Springer-Verlag, 1984.

[6] H. Samet, The Design and Analysis of Spatial Data Structures. Reading, MA: Addison-Wesley, 1990.

[7] J. S. Vitter and P. Flajolet, "Average-case analysis of algorithms and data structures," in Algorithms and Complexity (J. van Leeuwen, ed.), vol. A of Handbook of Theoretical Computer Science, pp. 431–524, Amsterdam: Elsevier, 1990.

[8] D. Wood, Data Structures, Algorithms, and Performance. Reading,Mass.: Addison-Wesley, 1993.

[9] D.S. Malik, C++ Programming: From Problem Analysis to Program Design, 4th ed., Course Technology, Boston, MA, 2009.

[10] E. Horowitz, S. Sahni, and S. Rajasekaran, Computer Algorithms C++, Computer Science Press, New York, 2004.

[11] D. S. Malik and M.K. Sen, Discrete Mathematical Structures, Theory and Applications, Course Technology, Boston, MA, 2004.

[12] N.M. Josuttis, The C++ Standard Library: A Tutorial and Reference, Addison-Wesley, Reading, MA, 1999.

[13] R. Sedgewick, Algorithms in C, 3rd ed., Addison-Wesley, Boston, MA, Parts 1–4, 1998; Part 5, 2002.

[14] D.E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd ed., Addison-Wesley, Reading, MA, 1997.

[15] D.E. Knuth, The Art of Computer Programming, Volume 3: Searching and Sorting, 2nd ed., Addison-Wesley, Reading, MA, 1998.