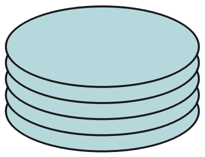
Chapter 6: Stacks



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:

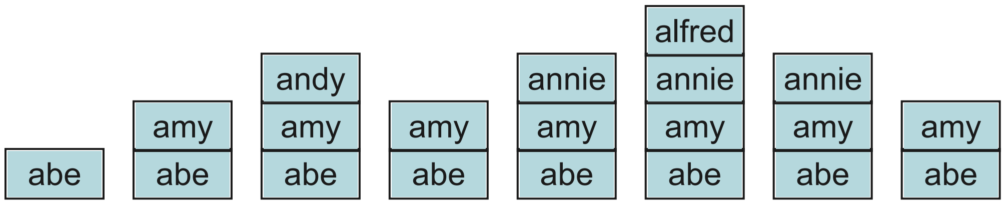
	Java class Stack	C++ stack adapter	Python list
push	push(value)	push(value)	lst.append(value)
pop	pop()	pop	Del lst[-1]
top	peek()	top()	lst[-1]
isEmpty	empty()	empty()	len(lst) == 0

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.

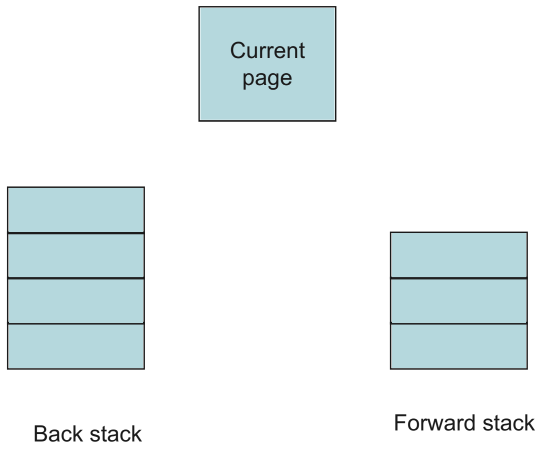


Applications of Stacks

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?

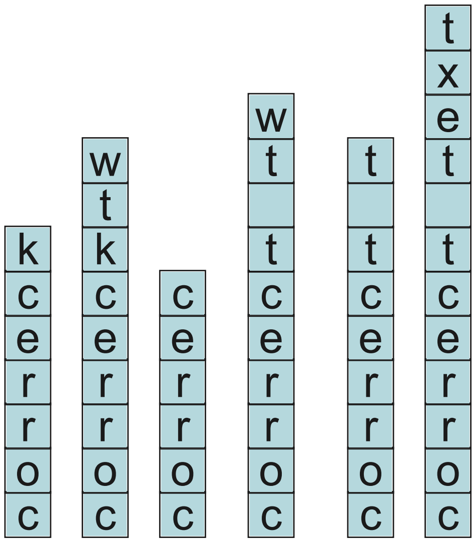
Question: Web browsers often provide a *history* feature, which records all web pages accessed in the recent past. How is this different from the back stack? Describe how the history should change when each of the three following conditions occurs: (a) when the user moves to a new page by pressing a 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 read in a stack-like fashion. Each non-backspace character is simply pushed on the stack. When a backspace is typed, the topmost character is popped from the stack and erased.



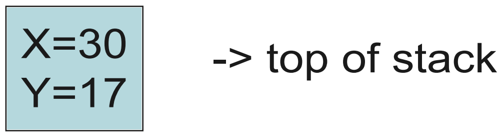
Question: What should be the effect if the user enters a backspace key and there are no characters in the input?

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)
```

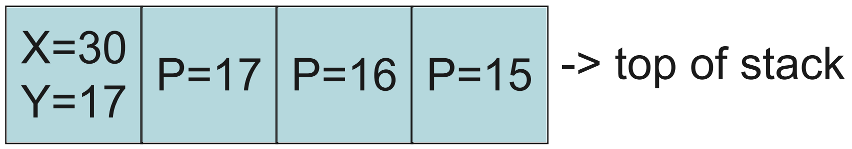
When the function a is invoked the activation record looks something like the following:



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.

The following illustrates the state of the stack at various points during the processing of the expression a (b { d e [f] g { h } I } j k) l m.

picture

The following illustrates the detection of an error when a closing delimiter fails to match the correct opening character:

picture

Another error occurs when there are opening delimiters but no closing character:

picture

Question: Show the state of the stack after each character is read in the following expression: (a b { c } d ([e [f] g]) (j))

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 is 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.

The following diagram illustrates the state of the stack and the output as different characters in the input are read:

picture

Question: Using this algorithm, show the state of the stack and the output for each of the following expressions:

example

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.

The following illustrates the state of the stack during the evaluation of the expression

picture

Question: Using this algorithm, show the state of the stack and the output for each of the following expressions.

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 + 4 5 + ?

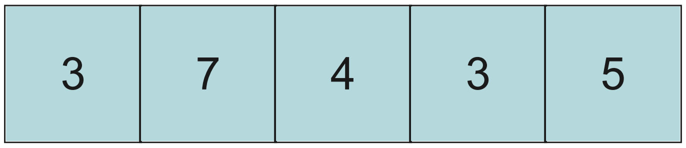
Stack Implementation Techniques

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 16	Introduction to the Dynamic Array
Worksheet 17	Dynamic Array Stack
Worksheet 18	Introduction to Linked List, Linked List Stack

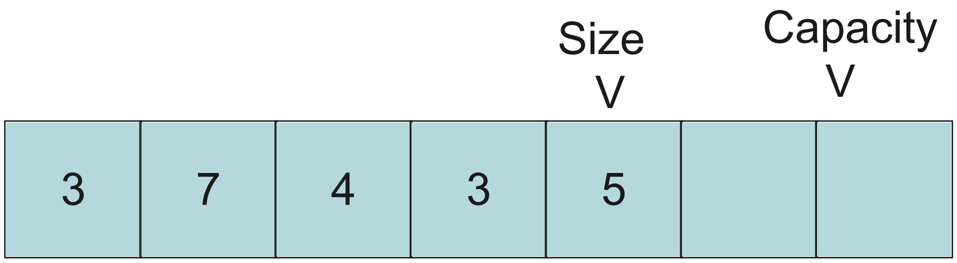
Building a Stack using a Dynamic Array

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

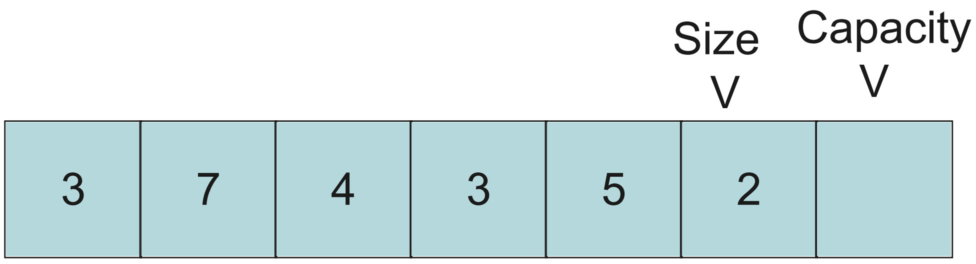


One problem with an array is that memory is allocated as a block. The size of this block is fixed when the array is created. If the size of the block corresponds directly to the number of elements in the collection, then adding a new value requires creating an entirely new block, and copying the values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

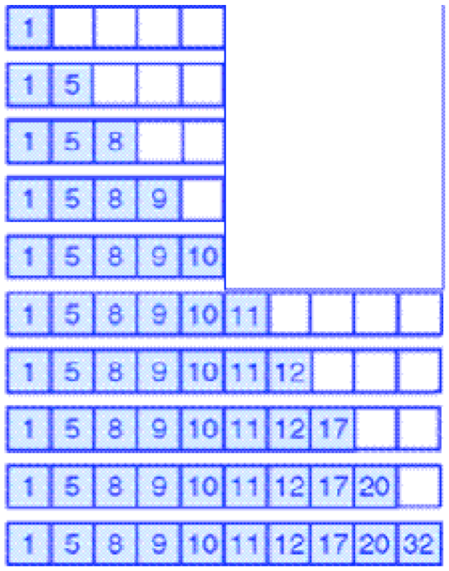


If the size is less than the capacity, then adding a new element to the stack is easy. It is simply a matter of incrementing the count on the size, and copying the new element into the correct location. Similarly, removing an element is simply a matter of setting the topmost location to null (thereby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

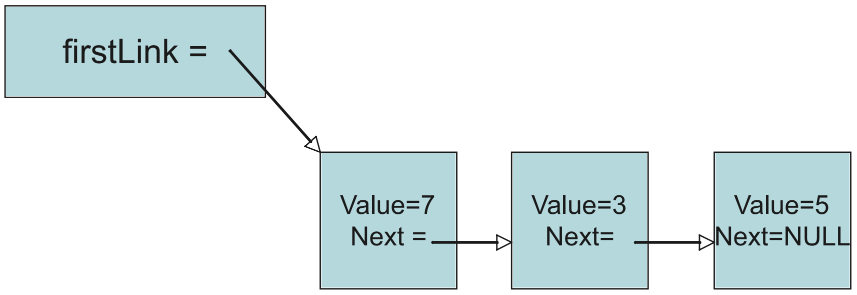
The second exceptional condition is more difficult. When a push instruction is requested but the size is equal to the capacity, there is no space for the new element. In this case a new array must be created. Typically, the size of the new array is twice the size of the current. Once the new array is 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 16 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.



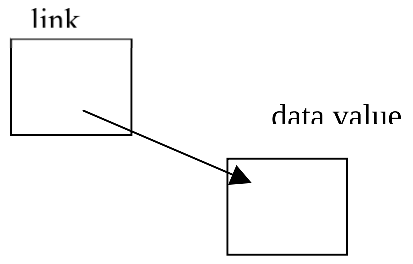
The advantage of the linked list is that the collection can grow as large as necessary, and each new addition to the chain of links requires only a constant amount of work. Because there are no big blocks of memory, it is never necessary to copy an entire block from place to place.

Worksheet 17 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 is the big-Oh complexity of pushing a value on the front of a linked list stack? Of popping?

11. What would eventually happen if the pop function did not free the first link in the list?

12. 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.

picture

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.

picture

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
```

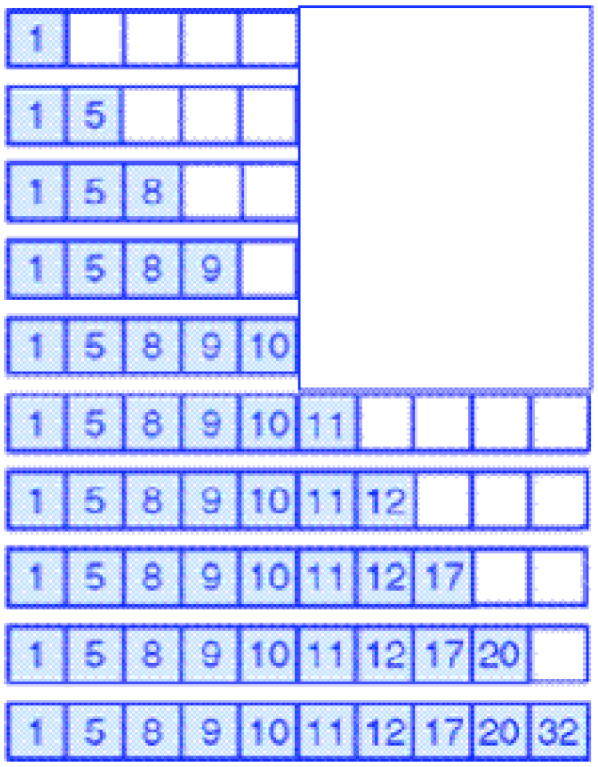
8. Using a ListStack as the implementation structure do the same analysis as in previous question.

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 indicating 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 consequitive 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

1. 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.

2. 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.

3. 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?

4. 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 5 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 remove, 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.