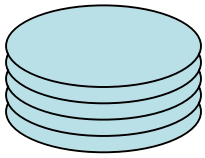
Chapter 6: Stacks

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:

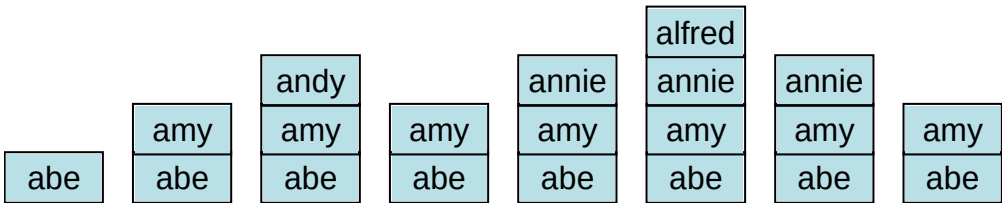
	Java class Stack	C++ stack adapter	Python list
push	push(value)	push(value)	lst.append(value)
pop	pop()	pop	Del lst[-1]
top	peek()	top()	lst[-1]
isEmpty	empty()	empty()	len(lst) == 0

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.

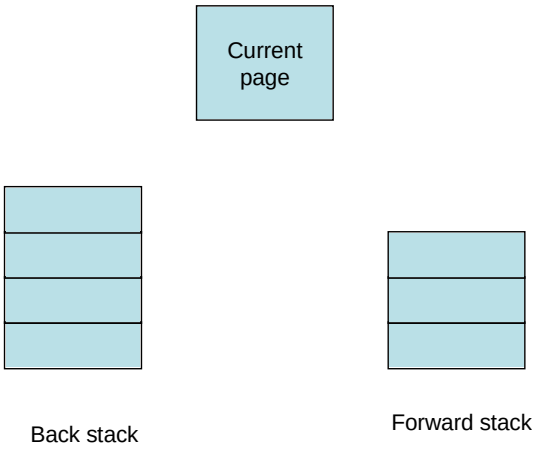


Applications of Stacks

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?

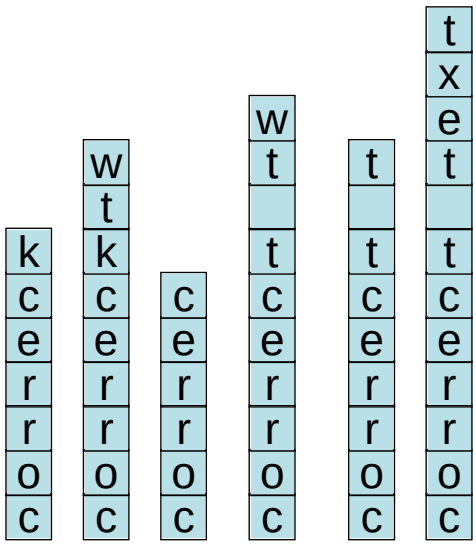
Question: Web browsers often provide a *history* feature, which records all web pages accessed in the recent past. How is this different from the back stack? Describe how the history should change when each of the three following conditions occurs: (a) when the user moves to a new page by pressing a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



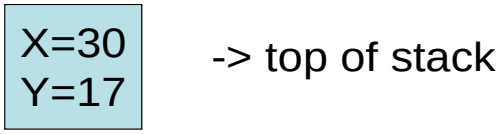
Question: What should be the effect if the user enters a backspace key and there are no characters in the input?

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)
```

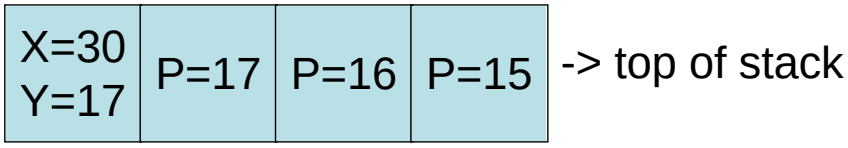
When the function a is invoked the activation record looks something like the following:



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.

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.

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 + ?

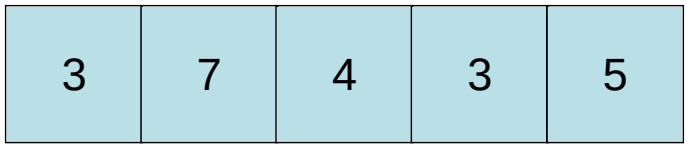
Stack Implementation Techniques

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

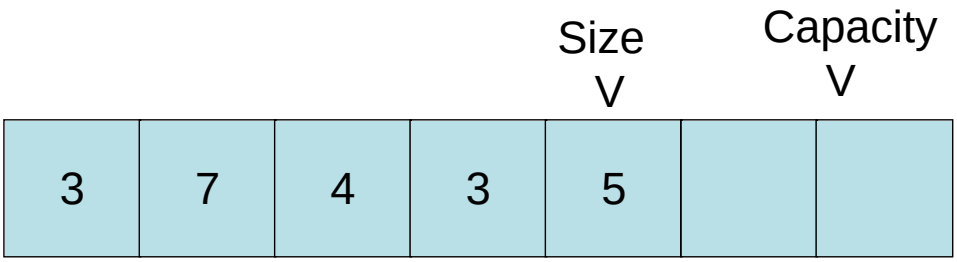
Building a Stack using a Dynamic Array

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

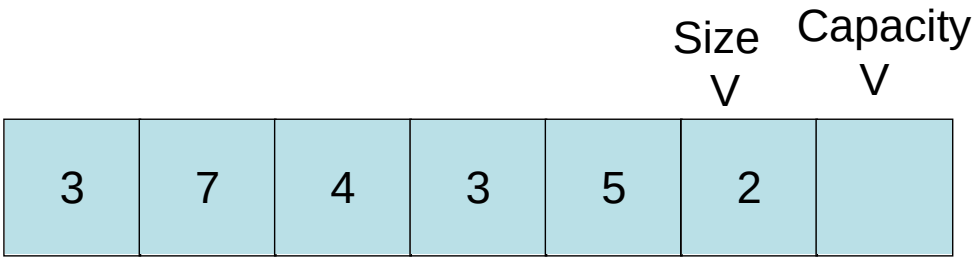


One problem with an array is that memory is allocated as a block. The size of this block is fixed when the array is created. If the size of the block corresponds directly to the number of elements in the collection, then adding a new value requires creating an entirely new block, and copying the values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

ERROR: undefinedresult
OFFENDING COMMAND: itransform

STACK:

2177.1
2889.9