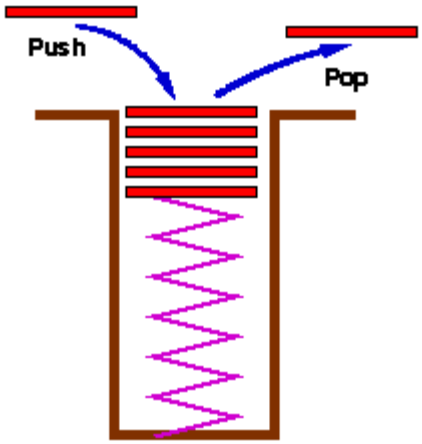
CHAPTER FOUR
4. DATA STRUCTURES AND APPLICATIONS : STACKS

4.1. Introduction

The stack is a list-like structure in which elements may be inserted or removed from only one end. While this restriction makes stacks less flexible than lists, it also makes stacks both efficient (for those operations they can do) and easy to implement. Many applications require only the limited form of insert and remove operations that stacks provide. In such cases, it is more efficient to use the simpler stack data structure rather than the generic list.

Stack is a simple data structure, in which insertion and deletion occur at the same end. It is a LIFO (Last In First Out) structure. The accessible element of the stack is called the **top** element. Elements are not said to be inserted, they are *pushed* onto the stack. When removed, an element is said to be *popped* from the stack.



A Stack is an ordered collection of data items, into which new items may be inserted and from which items may be deleted at only one end; and that end is called Top of the Stack.

A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.

4.2. Fundamental operations on stack

The fundamental operations on a stack are *push*, which is equivalent to an insert, and *pop*, which deletes the most recently inserted element. The most recently inserted element can be examined prior to performing a *pop* by use of the *top* routine. A *pop* or *top* on an empty stack is generally considered an error in the stack ADT. On the other hand, running out of space when performing a push is an implementation limit.

Formally, a stack is an abstract data type (ADT) that supports the following operations:

- **Push (x):** Insert the element *x* at the top of the stack.
- **Pop ( ):** Remove the top element from the stack; an error occurs if the stack is empty.

- **Empty ( ):** Check to see if the stack is empty or not. Return true if the stack is empty and false otherwise.
- **Full ( ):** Check to see if the stack is full or not.
- **top ( )**: Return the top most element in the stack without removing it; an error occurs if the stack is empty.
- **Size ( )**: Return the number of elements in the stack.

A series of push and pop operations is shown in the following figure. After pushing number 10 onto an empty stack, the stack contains only this number. After pushing 5 on the stack, the number is placed on top of 10 so that, when the popping operation is executed, 5 is removed from the stack, because it arrived after 10, and 10 is left on the stack. After pushing 15 and then 7, the topmost element is 7, and this number is removed when executing the popping operation, after which the stack contains 10 at the bottom and 15 above it.

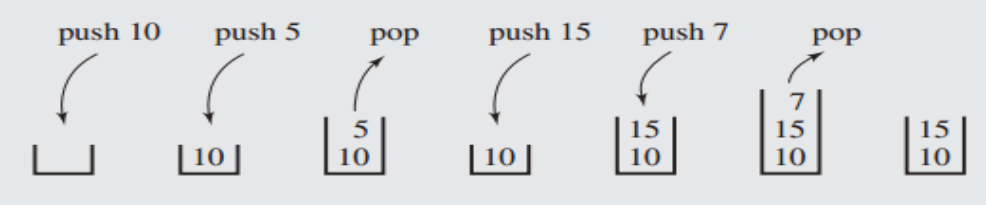


Figure 4.1: A series of operations executed on a stack.

The following table shows a series of stack operations and their effects on an initially empty stack of integers.

Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	*5*	(5)
pop()	–	()
pop()	*"error"*	()
top()	*"error"*	()
empty()	*true*	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	*4*	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	*3*	(9,7,3)

4.3. Implementation of Stacks

Stacks can be implemented both as an array (contiguous list) and as a linked list. We want a set of operations that will work with either type of implementation: i.e. the method of implementation is hidden and can be changed without affecting the programs that use them.

As with lists, there are many variations on stack implementation. The two approaches presented here are array-based and linked stacks, which are analogous to array-based and linked lists, respectively.

For the implementation of stack using array, *top* is defined to be the array index of the first free position in the stack. Thus, an empty stack has *top* set to 0, the first available free position in the array. (Alternatively, *top* could have been defined to be the index for the *top* element in the stack, rather than the first free position. If this had been done, the empty list would initialize *top* as -1). Methods *push* and *pop* simply place an element into, or remove an element from, the array position indicated by *top*. Because *top* is assumed to be at the first free position, *push* first inserts its value into the *top* position and then increments *top*, while *pop* first decrements *top* and then removes the *top* element.

4.3.1. Array Implementation of Stacks

Although an array cannot be a stack it can be the home of the stack i.e., an array can be declared with a range that is large enough for the maximum size of the stack. During the course of program execution the stack will grow and shrink within the space reserved for it. One end of the array will be fixed as bottom of the stack, while the top of the stack will constantly shift as items are popped and pushed. Thus another field is needed to keep track of the current position of the top of the stack.

i). The Push operation

To remind you, addition of an element is known as the push operation. So, if an array is given to you, which is supposed to act as a stack, you know that it has to be a static stack; meaning, data will overflow if you cross the upper limit of the array. So, keep this in mind.

The idea of push operation is the following

```
Push ( )
     {
      if there is room
        put an item on the top of the stack
      else
        give an error message
      }
```

Push operation is similar to inserting an element at the end of the list (ordinary not linked list).

Algorithm:

Step-1: Increment the Stack TOP by 1. Check whether it is always less than the Upper Limit of the stack. If it is less than the Upper Limit go to step-2 else report -"Stack Overflow"
Step-2: Put the new element at the position pointed by the TOP

Implementation:

```
static int stack[UPPERLIMIT];
int top= -1; /*stack is empty*/
void push(int item)
  {
    top = top + 1;
    if(top < UPPERLIMIT)
         stack[top] = item;
    else
         cout<<"Stack Overflow";
   }
```

Note: - In array implementation, we have taken TOP = -1 to signify the empty stack, as this simplifies the implementation. While pushing an element, top value will be incremented by '1' and one element will be added to stack.

ii). The Pop operation

POP is the synonym for delete when it comes to Stack. So, if you're taking an array as the stack, remember that you'll return an error message, "Stack underflow", if an attempt is made to Pop an item from an empty Stack. OK!

The idea of pop operation is the following.

```
Pop ( )
      {
        if stack not empty
         {
          return the value of the top item
          remove the top item from the stack
         }
        else
            give an error message
      }
```

Pop operation is similar to deleting the end element of an ordinary list.

Algorithm

Step-1: If the Stack is empty then give the alert "Stack underflow" and quit; or else go to step-2
Step-2:

a) Hold the value for the element pointed by the TOP
b) Put a NULL value instead
c) Decrement the TOP by 1

Implementation:

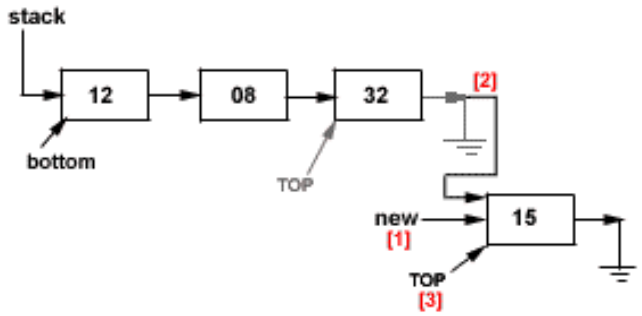
```
static int stack[UPPPERLIMIT];
int top = -1;
int pop( )
 {
   int del_val = 0;
   if(top = = -1)
   cout<<"Stack underflow"; /*step-1*/
    else
    {
   del_val = stack[top];  /*step-2*/
   stack[top] = NULL;
   top = top -1;
    }
   return(del_val);
 }
```

Note: - Step-2: (b) signifies that the respective element has been deleted. While popping an element, top value will be decremented by '1' and one element will be deleted from the stack.

4.3.2. Linked List Implementation of Stacks

i). The PUSH operation

It's very similar to the insertion operation in a dynamic singly linked list. The only difference is that here you'll add the new element only at the end of the list, which means addition can happen only from the TOP. Since a dynamic list is used for the stack, the Stack is also dynamic, means it has no prior upper limit set. So, we don't have to check for the Overflow condition at all!



In Step [1] we create the new element to be pushed to the Stack.
In Step [2] the TOP most element is made to point to our newly created element.
In Step [3] the TOP is moved and made to point to the last element in the stack, which is our newly added element.

Algorithm

Step-1: If the Stack is empty go to step-2 or else go to step-3
Step-2: Create the new element and make your "stack" and "top" pointers point to it and quit.
Step-3: Create the new element and make the last (top most) element of the stack to point to it
Step-4: Make that new element your TOP most element by making the "top" pointer point to it.

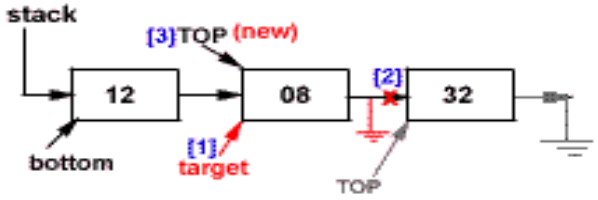
Implementation:

```
struct node
  {
   int item;
   struct node *next;
  };
struct node *stack = NULL; /*stack is initially empty*/
struct node *top = stack;


   void push(int value)
   {
      if(stack = = NULL)  /*step-1*/
        {
         temp = new node  /*step-2*/
         temp -> item = value;
         temp -> next = NULL;
         stack = temp;
         top = stack; /*step-4*/
        }
      else
        {
         temp = new node; /*step-3*/
         temp -> item = value;
         temp -> next = NULL;
          top -> next = temp;
          top = temp;   /*step-4*/
        }
   }
```

ii). The POP operation

This is again very similar to the deletion operation in any Linked List, but you can only delete from the end of the list and only one at a time; and that makes it a stack. Here, we'll have a list pointer, "target", which will be pointing to the last but one element in the List (stack). Every time we POP, the TOP most element will be deleted and "target" will be made as the TOP most element.



In step [1] we got the "target" pointing to the last but one node. In step [2] we freed the TOP most element. In step [3] we made the "target" node as our TOP most element.

Supposing you have only one element left in the Stack, then we won't make use of "target" rather we'll take help of our "bottom" pointer.

Algorithm

Step-1: If the Stack is empty then give an alert message "Stack Underflow" and quit; or else proceed
Step-2: If there is only one element left go to step-3 or else step-4
Step-3: Free that element and make the "stack", "top" and "bottom" pointers point to NULL and quit
Step-4: Make "target" point to just one element before the TOP; free the TOP most element; make "target" as your TOP most element

Implementation:

```
struct node
 {
   int item;
   struct node *next;
 };
struct node *stack = NULL; /*stack is initially empty*/
struct node *top = stack;

int pop( )
{   int pop_val = 0;
    struct node *target = stack;
        if(stack = = NULL)  /*step-1*/
           cout<<"Stack Underflow";
        else
          {   if(top = = bottom)  /*step-2*/
              {  pop_val = top -> item;   /*step-3*/
                 delete top;
                 stack = NULL;
                 top = bottom = stack;
               }
             else   /*step-4*/
              {
                 while(target -> next ! = top)
                     target = target -> next;
                 pop_val = top -> item;
                 delete top;
                 top = target;
                 target -> next = NULL;
               }
          }
    return(pop_val);
}
```

4.4. Applications of Stacks

4.4.1. Evaluation of Algebraic Expressions

E.g. **4 + 5 * 5**

Simple calculator: 45
Scientific calculator: 29 (correct)

Question:

Can we develop a method of evaluating arithmetic expressions without having to 'look ahead' or 'look back'? i.e. consider the quadratic formula:

x = (-b+(b^2-4*a*c)^0.5)/(2*a)

where **^** is the power operator, or, as you may remember it :

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

In its current form we cannot solve the formula without considering the ordering of the parentheses. i.e. we solve the innermost parenthesis first and then work outwards also considering operator precedence. Although we do this naturally, consider developing an algorithm to do the same, possible but complex and inefficient. Instead re-express the expression.

Re-expressing the Expression

Computers solve arithmetic expressions by restructuring them so the order of each calculation is embedded in the expression. Once converted an expression can then be solved in one pass.

Types of Expression

The normal (or human) way of expressing mathematical expressions is called infix form, e.g. **4+5*5**. However, there are other ways of representing the same expression, either by writing all operators before their operands or after them,

e.g.: **4 5 5 * +**

+ 4 * 5 5

This method is called Polish Notation (because this method was discovered by the Polish mathematician Jan Lukasiewicz).

When the operators are written before their operands, it is called the **prefix** form

e.g. **+ 4 * 5 5**

When the operators come after their operands, it is called **postfix** form (**suffix** form or **reverse polish notation**)

e.g. **4 5 5 * +**

Advantages of RPN (Reverse Polish Notation or postfix)

- Parentheses are unnecessary
- Easy for a computer (compiler) to evaluate an arithmetic expression Postfix (Reverse Polish Notation)

Postfix notation arises from the concept of post-order traversal of an expression tree (this concept will be covered when we look at trees).

For now, consider postfix notation as a way of redistributing operators in an expression so that their operation is delayed until the correct time.

Consider again the quadratic formula:

x = (-b+(b^2-4*a*c)^0.5)/(2*a)

In postfix form the formula becomes:

x b @ b 2 ^ 4 a * c * - 0.5 ^ + 2 a * / =

where **@** represents the unary - operator.

Notice the order of the operands remain the same but the operands are redistributed in a non-obvious way (an algorithm to convert infix to postfix can be derived).

The reason for using postfix notation is that a fairly simple algorithm exists to evaluate such expressions based on using a stack.

Postfix Evaluation

Consider the postfix expression: **6 5 2 3 + 8 * + 3 + ***

Algorithm:

```
initialize stack to empty;
while (not end of postfix expression)
{
  get next postfix item;
  if(item is value)
    push it onto the stack;
  else if(item is binary operator) {
    pop the stack to x;
    pop the stack to y;
    perform y operator x;
    push the results onto the stack;
  } else if (item is unary operator) {
    pop the stack to x;
    perform operator(x);
    push the results onto the stack
  }
}
```

The single value on the stack is the desired result.

Binary operators: +, -, *, /, etc.,
Unary operators: **unary minus**, **square root**, **sin**, **cos**, **exp**, etc.,

So for **6 5 2 3 + 8 * + 3 + ***
the first item is a value (6) so it is pushed onto the stack
the next item is a value (5) so it is pushed onto the stack
the next item is a value (2) so it is pushed onto the stack
the next item is a value (3) so it is pushed onto the stack
and the stack becomes

the remaining items are now: + 8 * + 3 + *

So next a '+' is read (a binary operator), so 3 and 2 are popped from the stack and their sum '5' is pushed onto the stack, and the stack becomes:

Next 8 is pushed and the next item is the operator *:

(8, 5 popped, 40 pushed)

Next the operator + followed by 3:

(40, 5 popped, 45 pushed, 3 pushed)

Next is operator +, so 3 and 45 are popped and 45 + 3 = 48 is pushed, the stack becomes:

Next is operator *, so 48 and 6 are popped, and 6 * 48 = 288 is pushed, the stack becomes:

Now there are no more items and there is a single value on the stack, representing the final answer 288.

Note the answer was found with a single traversal of the postfix expression, with the stack being used as a kind of memory storing values that are waiting for their operands.

4.4.2. Infix to Postfix (RPN) Conversion

Of course postfix notation is of little use unless there is an easy method to convert standard (infix) expressions to postfix. Again a simple algorithm exists that uses a stack:

Algorithm

```
initialize stack and postfix output to empty;
while(not end of infix expression)
{    get next infix item
    if(item is value) append item to postfix o/p
    else if(item = = ‘(‘) push item onto stack
    else if(item = = ‘)’) {
       pop stack to x
     while(x ! = ‘(‘)
       app. x to pfix o/p & pop stack to x
  }
else {
while(precedence(stack top) > = precedence(item))
       pop stack to x & app. x to pfix o/p
     push item onto stack
  }
}
while(stack not empty)
  pop stack to x and append x to pfix o/p
```

Operator Precedence (for this algorithm):

4 : '**(**' - only popped if a matching '**)**' is found
3 : All unary operators
2 : **/ ***
1 : + -

The algorithm immediately passes values (operands) to the postfix expression, but remembers (saves) operators on the stack until their right-hand operands are fully translated.

Eg., consider the infix expression **a+b*c+(d*e+f)*g**

Stack	Output
+	**ab**
* +	**abc**
+	**abc*+**
* (+	**abc*+de**
+ (+	**abc*+de*f**
+	**abc*+de*f+**
* +	**abc*+de*f+g**
empty	**abc*+de*f+g*+**