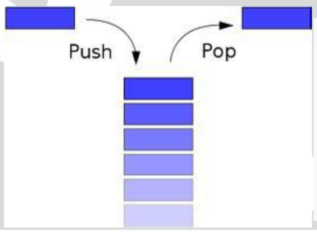
Chapter 4

Stack

A stack is an ordered collection of items where new items must be inserted or deleted only from one end, called the top of the stack. A stack is a data structure that keeps objects in Last- In-First-Out (LIFO) order, Objects are added to the top of the stack, and only the top of the stack can be accessed

$$\left.\begin{array}{l} L \to \text{Last} \\ I \to \text{In} \\ F \to \text{First} \\ O \to \text{Out} \end{array}\right\} = \left\{\begin{array}{l} F \to \text{First} \\ I \to \text{In} \\ L \to \text{Last} \\ O \to \text{Out} \end{array}\right.$$

Initially stack is empty. So, top = –1



Implementing stack:
There are two kinds of implementation of stack:
(1) Static implementation (2) Dynamic link list (discussed in Linked list)
(1) Static implementation :
Static implementation is done using array.
Information needed when implementing stack by using array are
(1) size of array (elements of stack will be stored using array.
(2) top variable.

One way to implement the stack is to have a data structure where a variable called top keeps the location of the elements in the stack (array). An array is used to store the elements in the stack

The following structure can be used to define the stack data structure:

```
#define Max 50
typedef struct{
int top;
 int items[Max]; /*stack can contain up to 50 integers*/
 }stack;
```

BASIC STACK OPERATIONS

(A) Initialize the Stack. :- You can initialize the stack by assigning -1 to the top pointer to indicate that the array based stack is empty (initialized) as follows: you can use the following function:

```
void StackInitialize(stack *S) { S→top = –1;  }
```

(B) Checking Empty Stack:

```
Boolean IsEmpty(stack *S){
if(S→top == –1)return true;
else return false;
}
```

(C) Checking FULL Stack

```
Boolean IsFull(stack *S){
If(S->top==Max-1) return true;
else return false;
}
```

(D) Push an item (insert an item)

```
void push (stack *S, int data) /*pushes data */
{
if(S->top == Max-1){
printf("Stack is full\n");
return; /*return back to main function*/
}
else
S->items[++s →top]= data;
}
```

(E) Pop an item(delete an item)

```
int pop(stack *S)
      {
     if(S->top==-1){
   printf("Stack is empty");
    return;
        }
      else
 return (S->element[S->top—] );
```

(F) Display ( )

```
void Display (stack * S)
        { int i;
        if (S →top = = –1)
                {
                printf ("stack is empty");
                return;
                }
                else
                for (i = top ; i ≥ 0; i – –)
                        {
                        printf ("%d", S →i);
                        }
                return;
        }
}
```

(G) Peek ( )

```
int Peek (stack *S, int I)
{
        if (S →top – I + 1 < = –1)
                {
                printf ("stack is underflow on Peek");
                return;
                }
                else
```

```
                    return (S $\rightarrow$ top – I + 1)
        }
```

Application of stack:

(A) Check for balanced parentheses in an expression

Given an expression string exp, write a program to examine whether the pairs and the orders of "{","}","(",")","[","]" are correct in exp. For example, the program should print true for exp = "[()]{}{[()()]()}" and false for exp = "[(])"

Algorithm:

(1) Declare a character stack S.

(2) Now scan the expression string exp from left to right.

(a) If the current character is a starting bracket ('(' or '{' or '[') then push it to stack.

(b) If the current character is a closing bracket (')' or '}' or ']') same as the top elements, then pop from stack

(3) After complete traversal, if there is some starting bracket left in stack then "not balanced"

Alternative Method: Valid only for few cases.

Algorithm

(1) Declare a variable counter and initialize it zero i.e counter1=0 ,counter2=0,counter3=0

Counter1 for (and)

Counter2 for { and }

Counter3 for [and]

(2) Traverse the string till null

(a) When ever you encounter an open brace increment
 counter1 if it is '('
 counter2 if it is '{'
 counter3 if it is '['

(b) When ever you encounter a closed brace decrement
 counter1 if it is ')'
 counter2 if it is '}'
 counter3 if it is ']'

(3) After the string traversal if all the three counters are zero then it is balanced otherwise unbalanced.

(B) Infix, Postfix and Prefix interconversion

Infix notation: (X+Y).

Operators are written in-between their operands. An expression such as `A* ( B + C ) / D` is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

Postfix notation : (also known as "Reverse Polish notation"): X Y +

Operators are written after their operands. The infix expression given above is equivalent to `A B C + * D/`

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order.

Prefix notation (also known as "Polish notation"): + X Y

Operators are written before their operands. The expressions given above are equivalent to `/ * A + B C D` As for Postfix, operators are evaluated left-to-right and brackets .Operators act on the two nearest values on the right

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D,add the results
A * (B + C)/D	A B C + * D /	/ * A + B C D	add B and C, multiply by A,divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

Infix to Postfix Conversion:

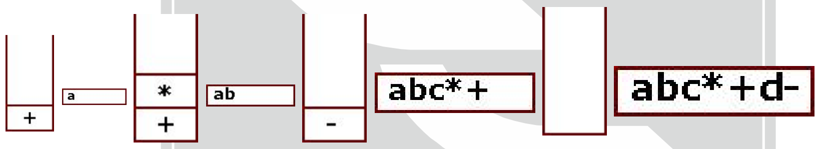
The algorithm for the conversion is as follows :

- Scan the Infix string from left to right.
- Initialize an empty stack.
- If the scannned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty. Add it to the postfix string
- If the scanned character is an Operand and the stack is not empty, compare the precedence of the character with the element on top of the stack(topStack). If topStack has higher precedence over the scanned character Pop of the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.

Repeat this step till all the characters are scanned.

- (After all characters are scanned, we have to add any character that the stack may have to the Postfix string.) If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.
- Return the Postfix string.

Example : Infix String : a+b*c-d



End result:

- Infix String : a+b*c-d
- Postfix String : abc*+d-

Postfix Evalutaion: Scan the given postfix expression from left to right.

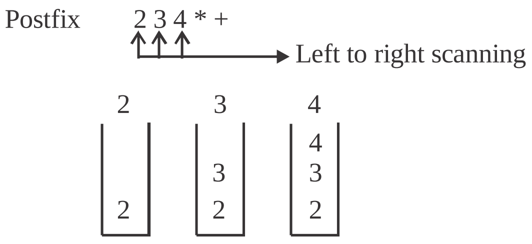
(1) When we encounter an operand and push it into d stack.

(2) When we encounter an operator pop two symbols (operands) from the stack apply the operator and put the result back in the stack.

(3) Repeat the above steps until the whole postfix expression is scanned.

(4) Pop out the result from the stack.

e.g. 2 + 3 * 4 = 14



POP 2 consicutive operand and the first operand go to the right side of operator.

$3 \times 4 = 12 \rightarrow$ again push it in stack