STACKS

1.1 What is a Stack?

A stack is a simple linear data structure used for storing data. In stack, the order in which the data arrives is important. The pile of books is a good example of stack. We usually pick the top most books from the pile. Similarly, the top most one(i.e. the item which made the entry last) the first one to be picked up/removed from the stack.

Definition

A stack is an ordered list in which a data insertion and deletion are done at one end. Where, the end is called as top. The last elements are inserted is the first one to be deleted. Hence, it is called Last in First out (LIFO) or First in Last out (FIFO).

In general, the special names are given to the two operations that can be made to a stack. Inserting an element to a stack is called as PUSH and when an element removed from the stack is called as POP. Trying to pop out an element from empty stack is called as under flow and trying to push an element into a full stack is called as over flow.

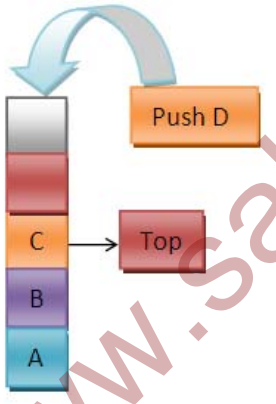


Fig1

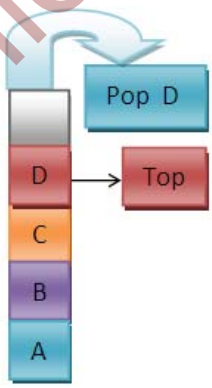


Fig2

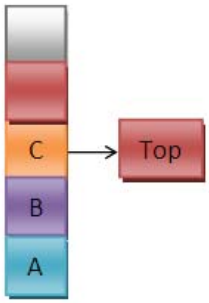


Fig3

From the above figures:

Fig1:

- The address location holding C element indicates the top of the stock or the stack pointer points to this location holding C element.
- The address location holding A is the bottom or starting point of stack.
- Now data D is pushed in stack from bottom.

Fig 2:

- In the fig 1 after pushing D in to stack, the address location holding D becomes the top the stock, meaning that the last added element indicates the top of the stack and stack keeps growing on with the data pushing in to stack.

- In the next step, the element D is popped out of the stock.

Fig 3:

- In Fig 2, the element D is popped out, then stack pointer automatically points to the last element address location i.e., C.

1.2 How stacks are used?

Stacks are mostly used in computer applications. Their most notable usage is in system software such as, compilers, operating systems etc. For example, when one C function calls another function, and passes some parameters, those parameters are passed using stack. Even, many compilers store local variables in stack.

1.3 Stack Operations:

Main Stack Operations:

- Void Push(int item): Inserts data onto stack
- Int Pop(): Removes and returns the last element from the stack.

Auxiliary Stack Operations:

- int Top(): This function returns the last inserted.
- void show(): Returns the number of elements stored in stack in stack.
- void is Empty(): Which returns whether stack empty or not.

1.4 Implementation

There exists many ways to implement stack and the most commonly used methods are:

1. Simple array based implementation
2. Based on Dynamic Memory Management Techniques:
   - Using Linked list

Example: Let's see how the new items get added to the end of the stack, and also how the elements get removed from there.

We assume that initially the stack is empty.

Operation	Contents of the stack after the operation
Push(A)	A
Push(B)	A B
Pop	A
Push(C)	A C
Push(D)	A C D
Push(E)	A C D E
Pop	A C D
Pop	A C
Pop	A
Pop	Empty

1.4.1 Simple Array Implementation

The implementation of stack uses an array.

- Here we add elements from left to right.
- Use a variable *Top* to keep track of the index of the top element and it moves up and down dynamically depending on push (inserting new item) and pop (removes existing element) operations performed.
- Here we have to define size of an array. The array storing an element into stack may become full.
  - If we try to push a new element into full stack it displays stack overflow.
  - If we try to pop an element from stack it displays stack underflow.

C Program:

```
/* Implement stack using an array*/
#define MAX 5 //define any number to limit your stack size
#include<stdio.h>
#include<conio.h>

int stack[MAX];
int top=0;//indicates the top of the stack
int stack_flag=0;//not empty

//insert an item to the stack
//While inserting an elemnt we are adding element and incrementing the top.
void push()
{
        int item;
        // Before pushing an element into stack it checks stack full or not
        if(top>=MAX){
                printf("Stack is overflow. we cant add more items\n");
                return;
        }
        else{
                printf("\nEnter an item to insert into stack: ");
```

```
                scanf("%d",&item);
                stack[top]=item;
                top++;
        }
}

//Read and delete the top most item from the stack
//while reading weare decrementing the top and popping the element.
int pop()
{
   top--;
        //before pop out an elemnt from stack it checks for stack empty
        if(top<0){
                printf("Stack is underflow.");
                stack_flag=1;
                return;
        }

        return stack[top];
}
// Display the elemnts in the stack
void show()
{
        int i;
        //before displaying the elemnts it checks for stack empty or not
        if(top==0){
                printf("\nStack is empty.");
                return;
         }
        else{
                printf("\nThe Stack elements are:");
                for(i=0;i<top;i++)
                {
                  printf("%d",stack[i]);
                printf(" ");
                }
        }
}
// main starts here
void main()
{
int choice, item;
    printf("1.Insert");
    printf("\n2.Delete");
    printf("\n3.show or display");
printf("\n4.Quit");
while(1)
    {
        printf("\nEnter your choice for the operation: ");
        scanf("%d",&choice);
switch(choice)
```

```
      {
case 1:
                push();
                show();
                break;
case 2:
item=pop();
                if(stack_flag==1){
                        printf("No items to delete\n");
                        stack_flag=0;
                        top=0;
                        break;
                }
                else{
                        printf("\nThe token deleted is %d",item);
                        show();
                        break;
        }
case 3:
                show();
                break;
       case 4:
return;
     }//switch ends here

   }//while ends here

}//main end here
```

Output:

```
1.Insert
2.Delete
3.show or display
4.Quit
Enter your choice for the operation: 1

Enter an item to insert into stack: 10

The Stack elements are:10
Enter your choice for the operation: 1

Enter an item to insert into stack: 20

The Stack elements are:10 20
Enter your choice for the operation: 1

Enter an item to insert into stack: 30

The Stack elements are:10 20 30
Enter your choice for the operation: 2

The token deleted is 30
The Stack elements are:10 20
Enter your choice for the operation: 4
```

Performance & Limitations

Performance

Let be define the number of elements in the stack are n. The complexities for each stack operations given as (for this type of implementation).

Space complexity (For n push operations)	O(n)
Time complexity of Push()	O(1)
Time complexity of Pop()	O(1)

Limitations: Here, the maximum size of the stack must be defined in prior to implementation and it cannot be changed.

1.4.2Dynamic Memory Management techniques:

Linked List Implementation

The other way of stack implementation is by using linked list. Here, push operation is implemented by inserting element at the beginning of the list and pop operation is implemented by deleting the node from the beginning (the head/ top).

*Here top node always points to beginning of the list.

```
/* C Program to Implement a Stack using Linked List */

#include<stdio.h>
#include<stdlib.h>

struct node
{
int item;
struct node *next;
}*top,*temp;

void push(int data);
void pop();
int Top();
voidisEmpty();
void show();
void DeleteStack();
void stack_count();
void CreateStack();

int count = 0;

void main()
{
int data, ch, e,n;
```

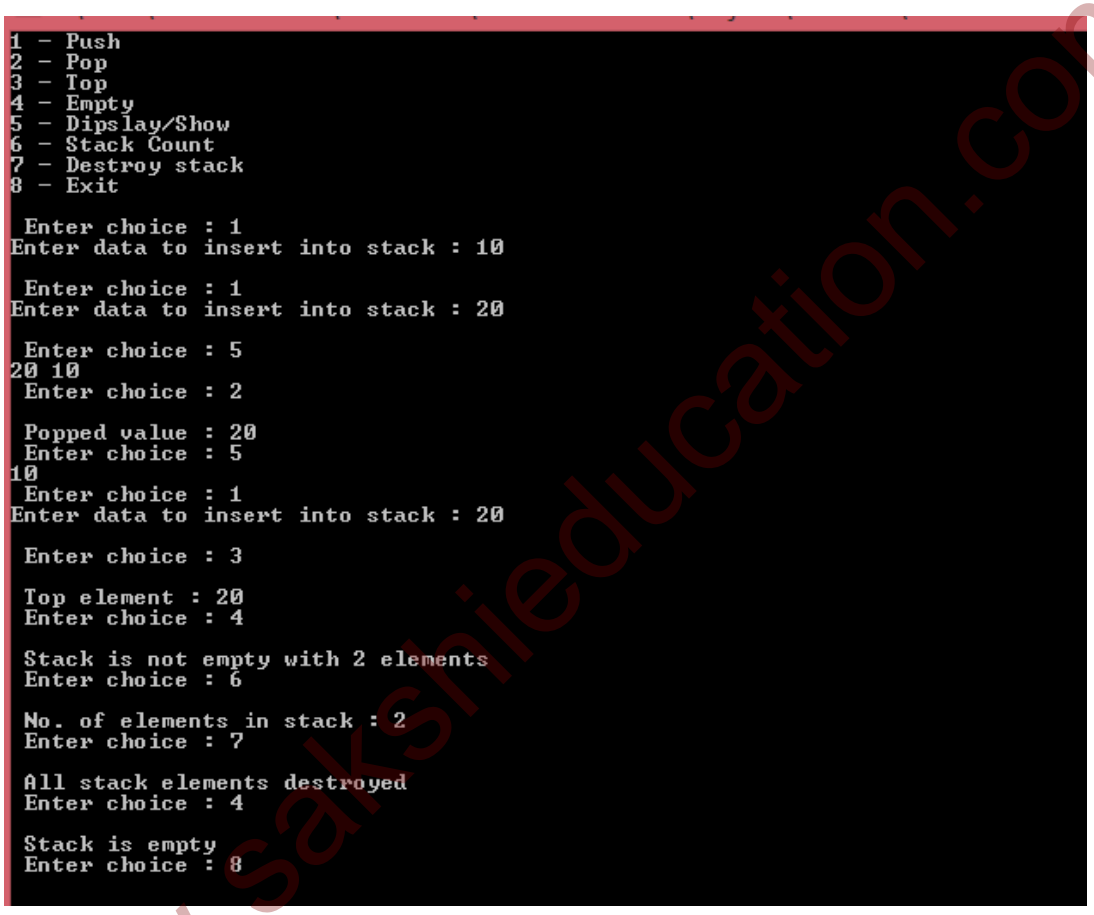
```
   printf("1 - Push\n");
   printf("2 - Pop\n");
   printf("3 - Top\n");
   printf("4 - Empty\n");
   printf("5 - Dipslay/Show\n");
   printf("6 - Stack Count\n");
   printf("7 - Destroy stack\n");
         printf("8 - Exit\n");

   CreateStack();

while (1)
   {
      printf("\n Enter choice : ");
      scanf("%d", &ch);

switch (ch)
      {
case 1:
         printf("Enter data to insert into stack : ");
         scanf("%d", &data);
         push(data);
break;
case 2:
         pop();
                        break;
case 3:
if (top == NULL)
            printf("No elements in stack");
else
         {
            e = Top();
            printf("\n Top element : %d", e);
         }
break;
case 4:
isEmpty();
break;
case 5:
         show();
break;
case 6:
         stack_count();
break;
case 7:
         DeleteStack();
break;
                  case 8:
         exit(0);
default :
```

```
            printf(" Wrong choice, Please enter correct choice  ");
break;
        }
    }
}

/* Create empty stack */
void CreateStack()
{
   top = NULL;
}

/* Count stack elements */
void stack_count()
{
   printf("\n No. of elements in stack : %d", count);
}

/* Push data into stack */
void push(int data)
{
if (top == NULL)
   {
      top =(struct node *)malloc(sizeof(struct node));
      top->next = NULL;
      top->item = data;
   }
else
   {
      temp =(struct node *)malloc(sizeof(struct node));
      temp->next = top;
      temp->item = data;
      top = temp;
   }
   count++;
}

/* Display stack elements */
void show()
{
struct node* p;
      p = top;

if (p == NULL)
   {
      printf("Stack is empty");
return;
   }

while (p != NULL)
   {
```

```
        printf("%d ", p->item);
        p = p->next;
    }
 }

/* Pop Operation on stack */
void pop()
{
struct node* n;
        n = top;

if (n == NULL)
    {
        printf("\n Error : Trying to pop from empty stack");
                return;
    }
else
        n = n->next;
            printf("\n Popped value : %d", top->item);
    free(top);
    top = n;
    count--;

}

/* Return top element */
int Top()
{
return(top->item);
}

/* Check if stack is empty or not */
voidisEmpty()
{
if (top == NULL)
        printf("\n Stack is empty");
else
        printf("\n Stack is not empty with %d elements", count);
}

/* Destroy entire stack */
void DeleteStack()
{
struct node* t;
        t = top;

while (t != NULL)
    {
        t = top->next;
        free(top);
        top = t;
```

```
    t = t->next;
  }
  free(t);
  top = NULL;

  printf("\n All stack elements destroyed");
  count = 0;
}
```

Output:



Performance

Let be define the number of elements in the stack are n. The complexities for each stack operations given as (for this type of implementation).

Space complexity (For n push operations)	O(n)
Time complexity of CreateStack()	O(1)
Time complexity of Push()	O(1)
Time complexity of Pop()	O(1)
Time complexity of Top()	O(1)
Time complexity of isEmpty()	O(1)
Time complexity of DeleteStack()	O(1)

Comparing Array Implementation and Linked List Implementation

Array Implementation:

- Operations take constant time.
- Any sequence of n operations (starting from empty stack) is “amortized” takes time proportional to n.

Linked List Implementation:

- Every operations take constant time O(1).
- Grows and shrinks gracefully.
- Every operation uses extra space and time to deal with reference.

1.5 Applications

Following are some of the important applications, where stack plays an important role.

- Infix-to-postfix conversion
- Implementation of function calls (including recursion)
- To evaluate postfix expression
- Syntax Parsing
- Backtracking
- Runtime Memory Management

Let’s consider an arithmetic expression a+b*c. Here, the addition operation is not evaluated first, because the operations are evaluated in the order of their precedence in the expression. Before evaluating an expression, the entire expression examined to determine whether there is any operator with higher precedence. After examining the expression, back tracking is done to evaluate the first operator to obtain the result. The back tracking operation can be best implemented by the stack operations.

The various stack oriented notations are:

- Infix
- Prefix
- Postfix

Infix:

An ordinary mathematical expression is called infix notation. When using an expression as an infix type notation, the operands are placed between the operands. To represent an expression in infix notation, we use parenthesis to specify the order in which the operands to be performed. Otherwise, the precedence rules will be followed to eliminate ambiguous result of the expression.

Example: (A+B)+(C-D)

Prefix:

In this type of notation, the operators are placed before the operands in a mathematical expression. This type of can also called as polish notation.

Example: The prefix notation for A+B+C-D expression is ++AB-CD.

Postfix:

This is also known as Reverse Polish Notation (RPN). In this type of notation, the operators are placed after the operands, i.e. the operators are preceded by the operands.

Example: The postfix notation for A+B+C-D expression is AB+CD-+.

- The advantage of using prefix and postfix notations in a mathematical expression is that to avoid the use of parenthesis and operator precedence rules completely.
- Time taken to evaluate a postfix and prefix expression is O(n), where n is the number of elements in an array.

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B-C	-+AB	AB+C-
(A+B)*C-D	-*+ABCD	AB+C*D-

Conversion from Infix to Prefix/Postfix Notations

In infix expressions, the operator precedence is implicit unless we use parenthesis. Before the infix to prefix or postfix conversion algorithm we have to define the operator precedence inside the algorithm. The table below shows the precedence and their order of evaluation among the operators.

Token	**Operator**	**Precedence (Priority)**	**Associatively (order of evaluation)**
( ) [ ] -> .	function call array element member selection via pointer member selection via object name	17	left- to- right
-- ++	Postfix decrement, Increment	16	left- to- right
-- ++ ! ~	Prefix decrement, Increment logical not bitwise complement	15	right-to-left

- + &* sizeof	unary minus or plus address or indirection size (in bytes)		
(type)	cast (convert value to temporary of type)	14	right-to-left
* / %	Multiplication/ division/ modulus	13	left- to- right
+ -	Binary add or subtract	12	left- to- right
<<>>	Bitwise shift left, shift right	11	left- to- right
>>= <<=	relational	10	left- to- right
== !=	equality	9	left- to- right
&	bitwise and	8	left- to- right
^	bitwise exclusive or	7	left- to- right
|	bitwise or	6	left- to- right
&&	logical and	5	left- to- right
||	logical or	4	left- to- right
?:	conditional	3	right-to-left
= += -+ *= /= %= &= ^= |= <<= >>=	Assignment addition/subtraction assignment multiplication/division assignment modulus/bitwise AND assignment bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	2	right-to-left
,	Comma (separate expressions)	1	left- to- right

1.6 Problems on Stack:

Problem1: Discuss the Infix to prefix notation Algorithm?

The below algorithm is used to convert infix expression to prefix expression.

Algorithm:

1. First, create an empty stack and reverse the given input string.
2. For each character t in the input stream, if the input t is an operand, then place it in the output buffer.
3. If the input t is an operator, push t into the stack.
4. If the operator in a stack has equal or higher precedence than input operator t, then pop the operator present in stack and add it to output buffer.

5. If the input is a right brace (close brace), push it into the stack.
6. If the input is a left brace (open brace), pop elements in stack one by one until we encounter left brace. Discard braces while writing to output buffer.

After completion of all mentioned above steps reverse the data in output buffer and the result will be prefix notation of given expression.

Example: Convert given infix expression (a+b)*(c-d) into prefix expression.

Given infix expression: (a+b)*(c-d)

Step1: Reverse the given expression:) d-c(*)b+a(

Step2: **Input:**)

Input is a right brace, so place it into a stack.

Output:

Input:d

d is an operand place it into stack

Output: **d**

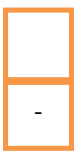
Input:-

Operator place it into a stack

Output: **d**

Input: c

Place the operand c into output buffer.



Output: **dc**

Input: (
Pop all – and)

Output: **dc-**

Input: *
Push the operator into stack

Output: **dc-**

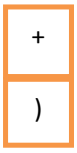
Input:)
Push the operator into stack

Output: **dc-**

Input: b
Place the operand in the output buffer

Output: **dc-b**

Input: +
Push the operator into stack



Output: **dc-b**

Input: a

Place the operand in the output buffer

Output: **dc-ba**

Input: (

Pop all operators till)

Output: **dc-ba+**

No more inputs to parse, pop all remaining elements in stack.

Output: **dc-ba+***

Reverse the output string:*+ba-cd

Prefix Notation: **+*ab-cd**

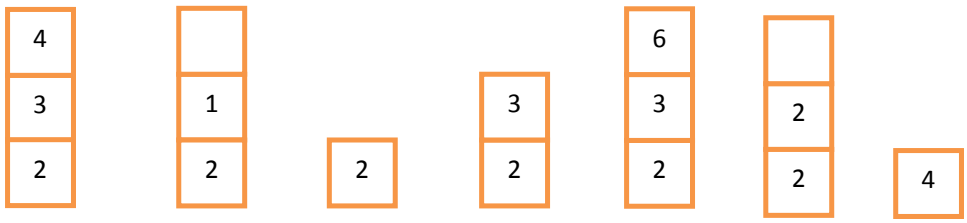
Problem2: Evaluate prefix expression using stack.

Stacks can also be used to evaluate a prefix notation. The following steps are considered to evaluate a prefix expression.

- Reverse the given input string
- For each character in string, if the character is operand push it into stack.
- If the character is operator, then the pop the first two operands in the stack and evaluate using this operator then place the result into the stack.

Example: +/63*-432

Reverse the given input string: 234-*36/+



Problem 3: Discuss the Infix to postfix notation Algorithm?

Before discussing the conversion algorithm we should know some important properties.

1. Let's consider an infix expression a+b*c and its postfix equivalent abc*+. Here notice that between infix and postfix the order of operands is unchanged i.e. a b c in both cases. But, the order of operators * and + is effected in the two expressions.
2. So, only one stack is enough to convert an infix to postfix expression. The stack we are going to used change the order of operators from infix to postfix. The stack we use will only contain operators and the open parenthesis symbol '('. Postfix expression don't contain parentheses. We shall not output the parentheses in the postfix output.

Algorithm:

1. Create an empty stack
2. For each character in a given input string, if the input is an operand, then place it into the output buffer.
3. If the input is an operator, push it into the stack.
4. If the operator in stack has equal or higher precedence than input operator, then pop the operator present in stack and add it to output buffer.
5. If the input is an left (open) brace, push it into the stack
6. If the input is a right (close) brace, pop elements in stack one by one until we encounter close brace. Discard braces while writing to output buffer.

Example: Convert given infix expression (a+b)*(c-d) into postfix expression.

Given infix expression: (a+b)*(c-d)

Step2: **Input:** (

Input is a left brace, so place it into a stack.

Output:

Input:a

a is an operand place it into stack

Output: **a**

Input:+

Operator place it into a stack

Output: **a**

Input: b

Place the operand c into output buffer.

Output: **ab**

Input:)

Pop all + and)

Output: **ab+**

Input: *

Push the operator into stack

Output: **ab+**

Input: (

Push the operator into stack

Output: **ab+**

Input: c

Place the operand in the output buffer

Output: **ab+c**

Input: -

Push the operator into stack

Output: **ab+c**

Input: d

Place the operand in the output buffer

Output: **ab+cd**

Input:)

Pop all operators till (

Output: **ab+cd-**

No more inputs to parse pop all remaining elements in stack.

Output: **ab+cd-***

Postfix Notation: **ab+cd-***

Problem2: Evaluate prefix expression using stack.

Stacks can also be used to evaluate a postfix notation. The following steps are considered to evaluate a postfix expression.

For each character in string, if the character is operand push it into stack.

If the character is operator, then the pop the first two operands in the stack and evaluate using this operator then place the result into the stack.

Example: evaluate postfix expression 68+92-/