



HARISH CHANDRA P.G. COLLEGE, VARANASI

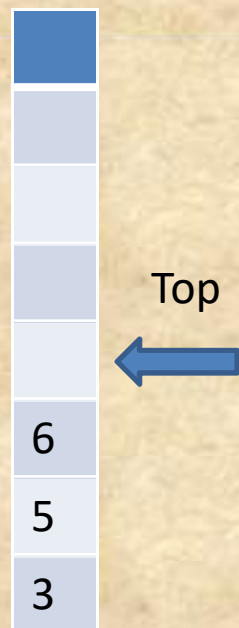
Subject:- *Data Structure using C & C++*

Class:- *BCA 3rd Semester*

Topic : *Stack And Queues*

Sub-Topic:- *Stack and queue operation Introduction*

Key Words : *Stack , infix ,prefix,postfix,Queue,Dqueue,priority Queue*



Stack



Queue

Name :- *Alok Kumar*

Department of *BCA*

Harish Chandra P G College ,Varanasi.

Mobile no *9696019403*

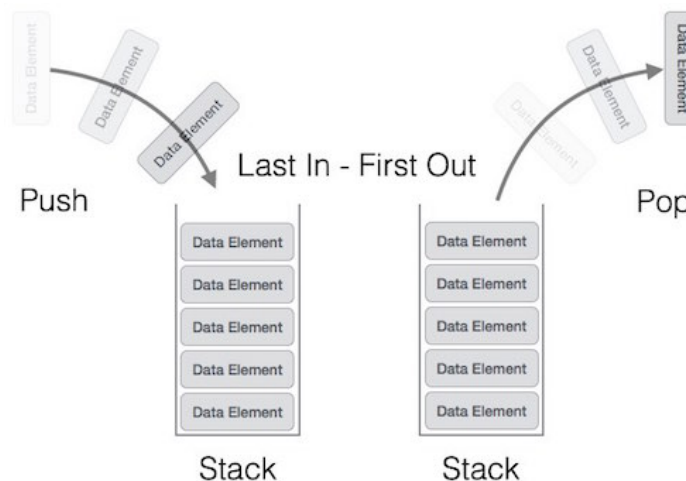
Email :- *alok.seth4@gmail.com*

Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.



Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

push() – Pushing (storing) an element on the stack.

pop() – Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

peek() – get the top data element of the stack, without removing it.

isFull() – check if stack is full.

isEmpty() – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

Program to illustrate stack operation

```
#include<stdio.h>  
#include<conio.h>  
int stk[7];  
int top=0;  
int max=7;  
int min=0;  
void push(int value)  
{  
if(top==max)  
{  
printf("stack is full");  
}  
else  
{  
stk[top]=value;  
top++;  
}  
}
```

```
int pop()  
{  
int vl;  
if(top==min)  
{  
printf("stack is empty");  
}  
else  
{  
vl=stk[--top];  
}  
return vl;  
}  
void display()  
{  
int i;  
for(i=0;i<top;i++)  
{  
printf("%d \t",stk[i]);  
}  
printf("\n");  
}
```

```
void main()
{
int i;
push(6);
push(5);
push(9);
push(11);
display();
i=pop();
display();
printf("your popped value is %d",i);
getch();
}
```

Infix to post fix using Stack

The Correct Way

Infix to Postfix Conversion

Expression = $A + B * C / D - F + A \wedge E$

Scanned Symbol	Stack	Output	Reason
A		A	Step 2
+	+	A	Step 3.1
B	+	AB	Step 2
*	+	AB	Step 3.1
C	+	ABC	Step 2
/	+/	ABC*	Step 3.2 / prec. is equal to * so not higher, so going to step 3.2
D	+/	ABC*D	Step 2
-	-	ABC*D/+	Step 3.2 / will be popped, added to o/p & then + popped & o/p. - will be pushed
F	-	ABC*D/+F	Step 2
+	+	ABC*D/+F-	Step 3.2 - will be popped, added to o/p and then + to stack
A	+	ABC*D/+F-A	Step 2
^	+	ABC*D/+F-A	Step 2
E	+	ABC*D/+F-AE	Step 2
(empty)		ABC*D/+F-AE^+	Step 8

Program to convert infix to postfix expression

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// A structure to represent a stack
typedef struct node {
    int top;
    int maxSize;
    // we are storing string in integer array, this will not give error
    // as values will be stored in ASCII and returned in ASCII thus, returned as string again
    int* array;
}Stack;

Stack* create(int max)
{
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->maxSize = max;
    stack->top = -1;
    stack->array = (int*)malloc(stack->maxSize * sizeof(int));
    return stack;
}

// Checking with this function is stack is full or not
// Will return true is stack is full else false
//Stack is full when top is equal to the last index
int isFull(Stack* stack)
{
    if(stack->top == stack->maxSize - 1){
        printf("Will not be able to push maxSize reached\n");
    }
}
```

```
// Since array starts from 0, and maxSize starts from 1
return stack->top == stack->maxSize - 1;
}
```

```
// By definition the Stack is empty when top is equal to -1
// Will return true if top is -1
int isEmpty(Stack* stack)
{
return stack->top == -1;
}
```

```
// Push function here, inserts value in stack and increments stack top by 1
void push(Stack* stack, int item)
{
if (isFull(stack))
return;
stack->array[++stack->top] = item;
}
```

```
// Function to remove an item from stack. It decreases top by 1
int pop(Stack* stack)
{
if (isEmpty(stack))
return INT_MIN;
return stack->array[stack->top--];
}
```

// Function to return the top from stack without removing it

```
int peek(Stack* stack)
{
    if (isEmpty(stack))
        return INT_MIN;
    return stack->array[stack->top];
}
```

// A utility function to check if the given character is operand

```
int checkIfOperand(char ch)
{
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}
```

// Function to compare precedence

// If we return larger value means higher precedence

```
int precedence(char ch)
{
    switch (ch)
    {
        case '+':
        case '-':
            return 1;

        case '*':
        case '/':
            return 2;
```

```

return 2;case '^':
return 3;
}
return -1;
}

```

```

// The driver function for infix to postfix conversion
int covertInfixToPostfix(char* expression)

```

```

{
int i, j;

```

```

// Stack size should be equal to expression size for safety
Stack* stack = create(strlen(expression));
if(!stack) // just checking is stack was created or not
return -1 ;

```

```

for (i = 0, j = -1; expression[i]; ++i)
{
// Here we are checking is the character we scanned is operand or not
// and this adding to to output.
if (checkIfOperand(expression[i]))
expression[++j] = expression[i];

```

```

// Here, if we scan character '(', we need push it to the stack.
else if (expression[i] == '(')
push(stack, expression[i]);

```

```

// Here, if we scan character is an ')', we need to pop and print from the stack
// do this until an '(' is encountered in the stack.
else if (expression[i] == ')')
{

```

```

while (!isEmpty(stack) && peek(stack) != '(')
expression[++j] = pop(stack);
if (!isEmpty(stack) && peek(stack) != '(')
return -1; // invalid expression
else
pop(stack);
}
else // if an operator
{
while (!isEmpty(stack) && precedence(expression[i]) <= precedence(peek(stack)))
expression[++j] = pop(stack);
push(stack, expression[i]);
}

}

// Once all initial expression characters are traversed
// adding all left elements from stack to exp
while (!isEmpty(stack))
expression[++j] = pop(stack);

expression[++j] = '\0';
strrev(expression);
printf( "\n%s", expression);
return 1;
}

```

```
int main()
{
char expression[] = "a+b*c-d";
strrev(expression);
convertInfixToPostfix(expression);
getch();
return 0;
}
```

Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



Program to demonstrate Queue operation

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
typedef struct node
{
    int que[MAX];
    int front;
    int rear;
}Queue;

void init(Queue *q)
{
    q->front = q->rear = 0;
}

void enqueue(Queue * q ,int value)
{
    if(q->rear == MAX)
        printf("queue is full");
    else
    {
        q->que[q->rear]=value;
        q->rear++;
    }
}

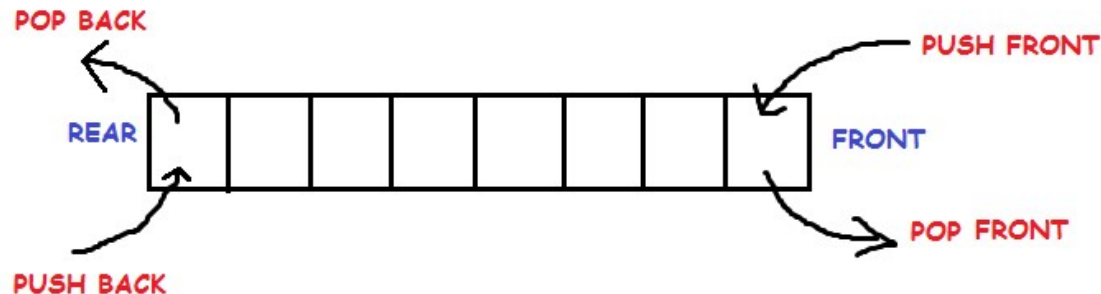
int dequeue(Queue * q)
{
    int v;
    if(q->front == q->rear)
        printf("queue is empty");
    else
```

```
else
v=q->que[q->front++];
return v;
}
void display(Queue *q)
{
int i;
for(i=q->front ; i<q->rear ; i++)
printf("%d \t",q->que[i]);
printf("\n");
}

void main()
{
Queue q;
int i;
init(&q);
enqueue(&q,6);
enqueue(&q,7);
enqueue(&q,2);
enqueue(&q,12);
enqueue(&q,9);
display(&q);
i=dequeue(&q);
display(&q);
printf("your popped value is %d",i);
getch();
}
```

Double Ended Queue

Double ended queue is a more generalized form of queue data structure which allows insertion and removal of elements from both the ends, i.e , front and back.



Implementation of Double ended Queue

Here we will implement a double ended queue using a circular array. It will have the following methods:

push_back : inserts element at back

push_front : inserts element at front

pop_back : removes last element

pop_front : removes first element

get_back : returns last element

get front : returns first element

empty : returns true if queue is empty

full : returns true if queue is full

Priority Queue

Priority Queue is an extension of queue with following properties.


Every item has a priority associated with it.

An element with high priority is dequeued before an element with low priority.

If two elements have the same priority, they are served according to their order in the queue.

In the below priority queue, element with maximum ASCII value will have the highest priority.

Priority Queue		
Initial Queue = { }		
Operation	Return value	Queue Content
insert (C)		C
insert (O)		C O
insert (D)		C O D
remove max	O	C D
insert (I)		C D I
insert (N)		C D I N
remove max	N	C D I
insert (G)		C D I G



A typical priority queue supports following operations.

insert(item, priority): Inserts an item with given priority.

getHighestPriority(): Returns the highest priority item.

deleteHighestPriority(): Removes the highest priority item.

Thank you