Data Structures (15B11CI311)

Odd Semester 2020

3rd Semester , Computer Science and Engineering

Jaypee Institute Of Information Technology (JIIT), Noida

Outline

- Review of Stack and Queue without using STL
- Implementation of Stack and Queue using STL, Recursion removal using stack

Review of Stack and Queue without using STL

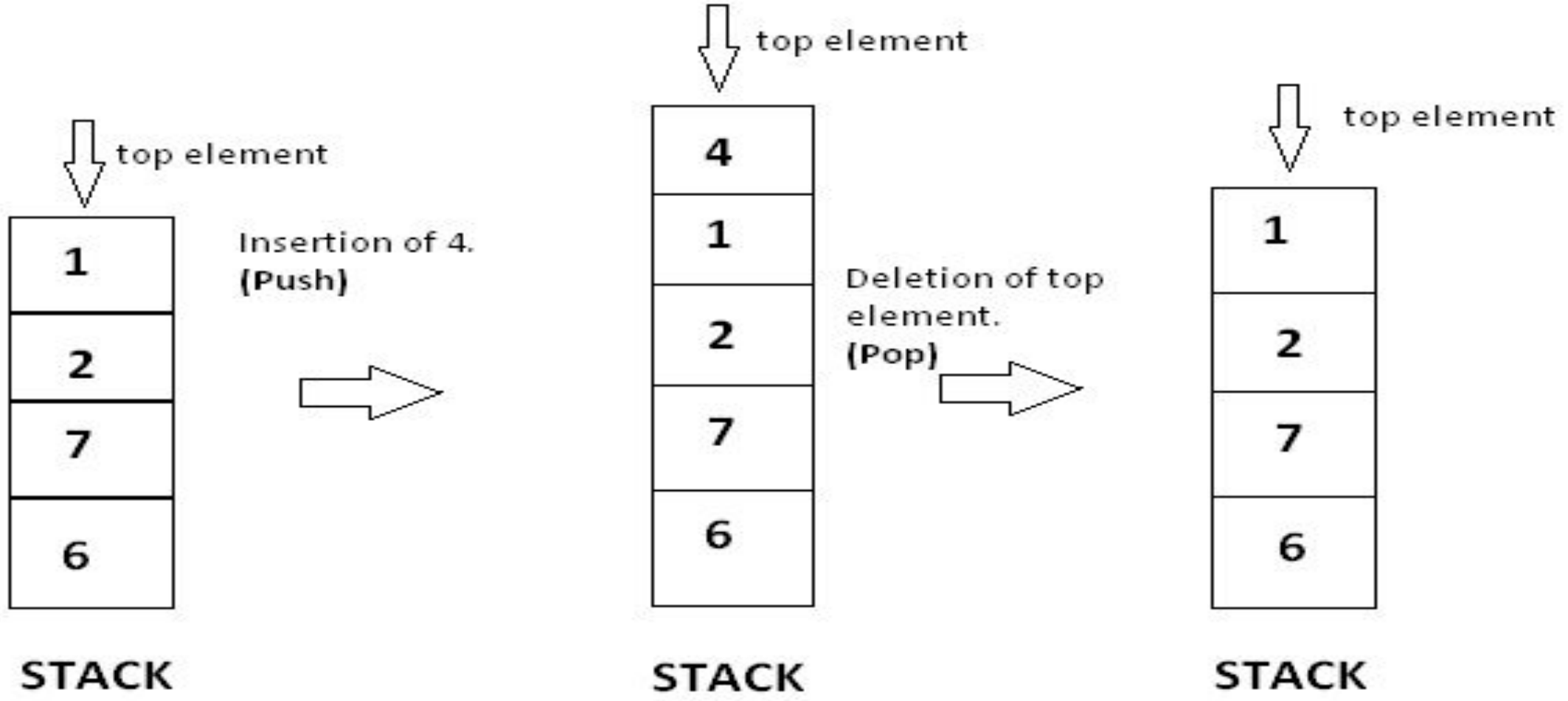
Stacks: Stack is collection of elements, that follows the LIFO order.

- **LIFO stands for Last In First Out**, which means element which is inserted most recently will be removed first.

Imagine a stack of tray on the table –

- When we put a tray there we put it at top, and when you remove it, we also remove it from top.

- A stack has a restriction that insertion and deletion of element can only be done from only one end of stack and we call that position as **top**.
- The element at top position is called **top element**.
- Insertion of an element is called **PUSH** and deletion is called **POP**.


top element
1
2
7
6
Insertion of 4.
(Push)
STACK
top element
4
1
2
7
6
Deletion of top
element.
(Pop)
STACK
top element
1
2
7
6
STACK

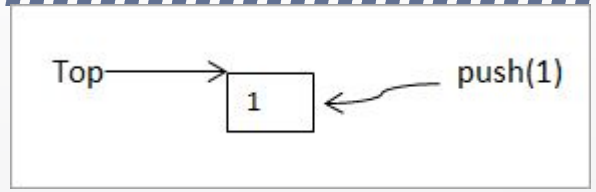
Stack Operations

push:

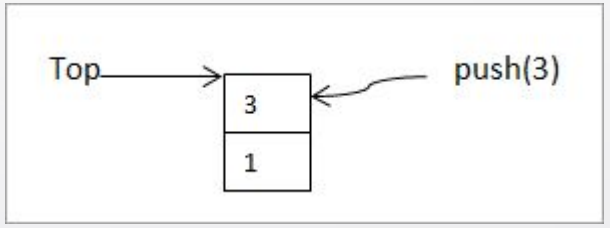
- push operation is used to insert an element in the stack.
- This operation always adds elements at the top of the stack.
- **Consider an empty stack mystack of type integer**.

Push(x)

- add element 1 to the stack



- Then, add element 3 to the stack.



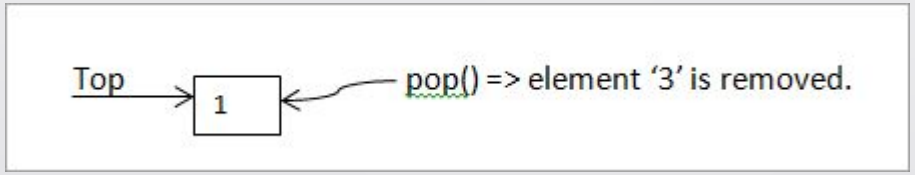
- After applying push operation, an element is added at the top of the stack.
- After every push operation, the size of the stack is increased by 1.

push(x) : insert element x at the top of stack.

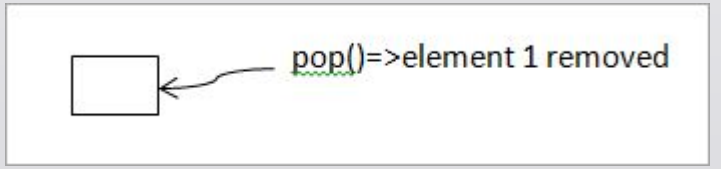
```
void push (int stack[ ] , int x , int n) {
    if ( top == n-1 ) {              //if top position is the last of position of stack, means stack is full .
        cout << "Stack is full. Overflow condition!" ;
    }
    else{
        top = top +1 ;               //incrementing top position
        stack[ top ] = x ;           //inserting element on incremented position .
}
}
```

Pop()

- pop operation is used to remove an element from the stack.
- The element removed is the one that is pointed to by the top of the stack.
- After applying pop operation, the stack size is reduced by 1.
- **Let us see how the pop operation looks like:**
- When the function pop()call is executed, the element at the top of the stack is removed and the 'Top' points to the next element as shown below.



- If pop() is called again , then the next element (in this case 1) will be removed thereby resulting in an empty stack.



pop( ) : removes element from the top of stack.

```
void pop (int stack[ ] ,int n )
{
   if( isEmpty ( ) )
   {
      cout << “Stack is empty . Underflow condition! ” << endl ;
   }
   else
   {
       top = top - 1 ; //decrementing top’s position will detach last element from stack .
   }
}
```

Other stack operations:

top:

- Returns the topmost element of the stack.

empty:

- Checks if the stack is empty or not.

size:

- Returns the size of the stack i.e. the number of elements in the stack.

topElement ( ) : access the top element of stack.

```
int topElement ( )
{
   return stack[ top ];
}
```

isEmpty ( ) : check whether the stack is empty or not.

```
bool isEmpty ( )
{
   if ( top == -1 )  //stack is empty .
   return true ;
   else
   return false;
}
```

size ( ) : tells the current size of stack

```
int size ( )
{
   return top + 1;
}
```

Stack Implementation:

```
#include <iostream>
using namespace std;
int top = -1; //globally defining the value of top ,as the stack is empty
.
void push (int stack[ ] , int x , int n)
{
if ( top == n-1 )          //if top position is the last of position of stack,
means stack is full .
   fvd{
cout << "Stack is full. Overflow condition!" ;
}
   else
   {
        top = top +1 ;          //incrementing top position
        stack[ top ] = x ;       //inserting element on incremented
position .
   }
}
bool isEmpty ( )
{
   if ( top == -1 )  //stack is empty .
      return true ;
   else
      return false;
}
```

```
void pop (int stack[ ] ,int n )
{

   if( isEmpty ( ) )
   {
      cout << "Stack is empty . Underflow condition! " << endl ;
   }
   else
   {
       top = top - 1 ; //decrementing top's position
 will detach last element from stack .
   }
}
int size ( )
{
   return top + 1;
}
int topElement ( )
{
   return stack[ top ];
}
// Now lets implementing these functions on the above stack
```

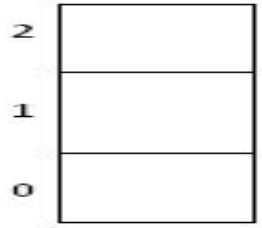
Implementation:

```
int main( )
{
   int stack[ 3 ];        // pushing element 5 in the stack .
   push(stack , 5 , 3 ) ;
   cout << "Current size of stack is " << size ( ) << endl ;
   push(stack , 10 , 3);
   push (stack , 24 , 3) ;

   cout << "Current size of stack is " << size( ) << endl ;
   //As now the stack is full ,further pushing  will show overflow
condition .
   push(stack , 12 , 3) ;
   //Accessing the top element .
   cout << "The current top element in stack is " << topElement( ) <<
endl;
   //now removing all the elements from stack .
   for(int i = 0 ; i < 3;i++ )
      pop( );
   cout << "Current size of stack is " << size( ) << endl ;

   //as stack is empty , now further popping will show
underflow condition .
   pop ( );

}
```

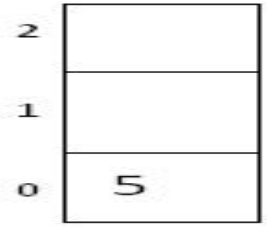
Output :

```
Current size of stack is 1
Current size of stack is 3
The current top element in stack is 24
Stack is full. Overflow condition!
Current size of stack is 0
Stack is empty . Underflow condition!
```

1) Initially **stack** is empty.
top = -1.

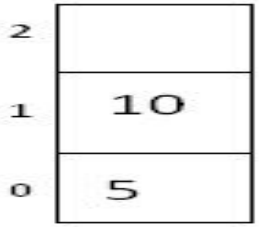


2) push(stack, 5, 3)



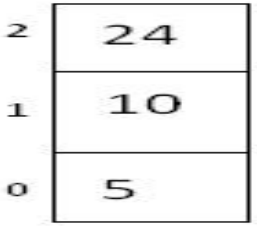
top element
top = 0

3) push(stack, 10, 3)



top element
top = 1

4) push(stack, 24, 3)



top element
top = 2

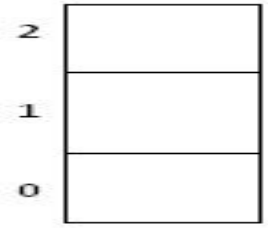
5) As top = 2, current size of stack is top+1 , i.e 3 .
Now stack is full,as 3 is maximum size of stack

6) push(stack, 12, 3)

As ,stack is full ,it will show
OVERFLOW CONDTION!

7) Deleting all elements from stack.

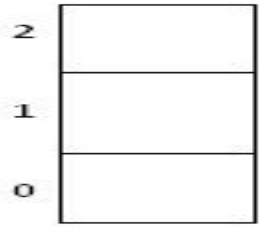
pop(stack, 3)
pop(stack, 3)
pop(stack, 3)



EMPTY STACK !!

top = -1

8) pop(stack, 3)



As **stack** is empty, further deleting will cause

UNDERFLOW CONDITION!

Queue

- Queue is a data structure that follows the **FIFO** principle.
- FIFO means **First In First Out** i.e the element added first in the queue will be the one to be removed first.
- Elements are always added to the back and removed from the front.
- Think of it as a line of people waiting for a bus at the bus stand.
- The person who will come first will be the first one to enter the bus.

Queue operations:

Enqueue:

- Adds an element to the back of the queue if the queue is not full otherwise it will print “OverFlow”.

```
void enqueue(int queue[], int element, int& rear, int arraySize) {
    if(rear == arraySize)           // Queue is Full
        printf(“OverFlow\n”);
    else {
        queue[rear] = element;    // Add the element to the back
        rear++;
    }
}
```

Queue operations:

Dequeue:

- Removes the element from the front of the queue if the queue is not empty otherwise it will print “UnderFlow”.

```
void dequeue(int queue[], int& front, int rear) {
    if(front == rear)           // Queue is empty
        printf(“UnderFlow\n”);
    else {
        queue[front] = 0;       // Delete the front element
        front++;
    }
}
```

Queue operations:

Front:

- Return the front element of the queue

```
int Front(int queue[], int front) {
    return queue[front];
}
```

Size:

- Returns the size of the queue or the number of elements in the queue.

```
int size(int front, int rear) {
    return (rear - front);
}
```

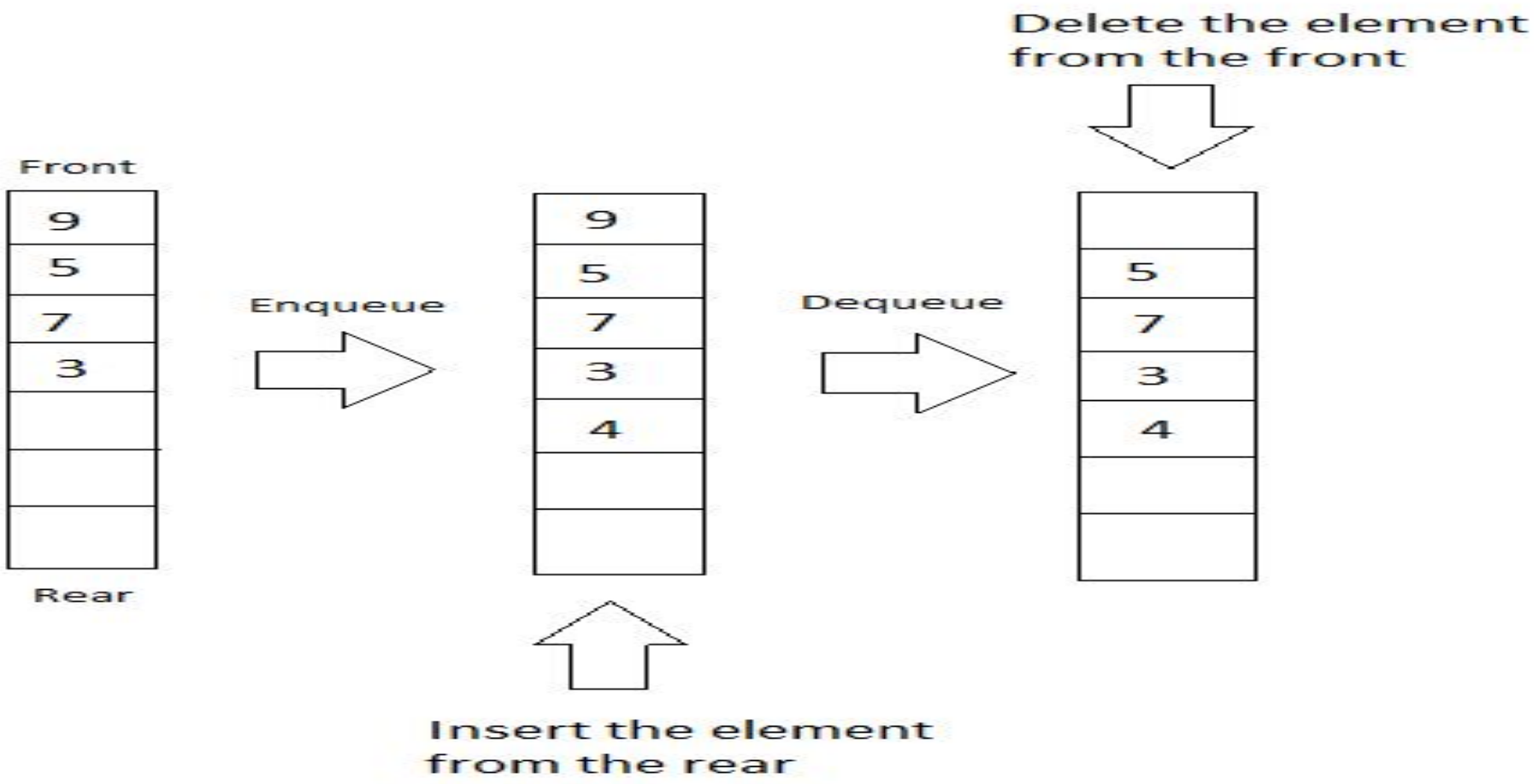
Queue operations:

IsEmpty:

- Returns true if the queue is empty otherwise returns false.

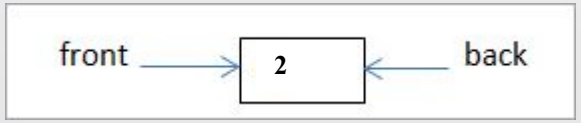
```
bool isEmpty(int front, int rear) {
    return (front == rear)
}
```

Queue Implementation



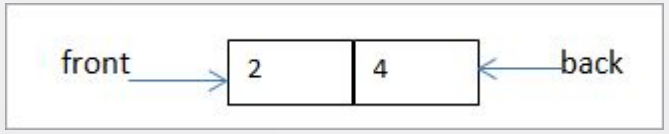
Example :

- Now we push an even number 2 in the queue ‘myqueue’ with the operation
- myqueue.push(2);
- **Now the queue will look like:**



Example :

- Next, we add ‘4’ to the queue with call ‘myqueue.push(4)’.
- **Now the queue looks as shown below:**



- As seen above, the elements are pushed into the queue from the rear end or back.
- **Now let us pop operation on myqueue.**
- myqueue.pop();

Example :

```
#include <iostream>
using namespace std;
int queue[100], n = 100, front = - 1, rear = - 1;
void Insert() {
  int val;
  if (rear == n - 1)
  cout<<"Queue Overflow"<<endl;
  else {
    if (front == - 1)
    front = 0;
    cout<<"Insert the element in queue : "<<endl;
    cin>>val;
    rear++;
    queue[rear] = val;
  }
}
```

```
void Delete() {
  if (front == - 1 || front > rear) {
    cout<<"Queue Underflow ";
    return ;
  } else {
    cout<<"Element deleted from queue is : "<< queue[front] <<endl;
    front++;;
  }
}
void Display() {
  if (front == - 1)
  cout<<"Queue is empty"<<endl;
  else {
    cout<<"Queue elements are : ";
    for (int i = front; i <= rear; i++)
    cout<<queue[i]<<" ";
      cout<<endl;
  }
}
```

Example :

```
int main() {
   int ch;
   cout<<"1) Insert element to queue"<<endl;
   cout<<"2) Delete element from queue"<<endl;
   cout<<"3) Display all the elements of
queue"<<endl;
   cout<<"4) Exit"<<endl;
   do {
      cout<<"Enter your choice : "<<endl;
      cin<<ch;
      switch (ch) {
         case 1: Insert();
         break;
         case 2: Delete();
         break;
         case 3: Display();
         break;
         case 4: cout<<"Exit"<<endl;
         break;
         default: cout<<"Invalid choice"<<endl;
      }
   } while(ch!=4);
   return 0;
}
```

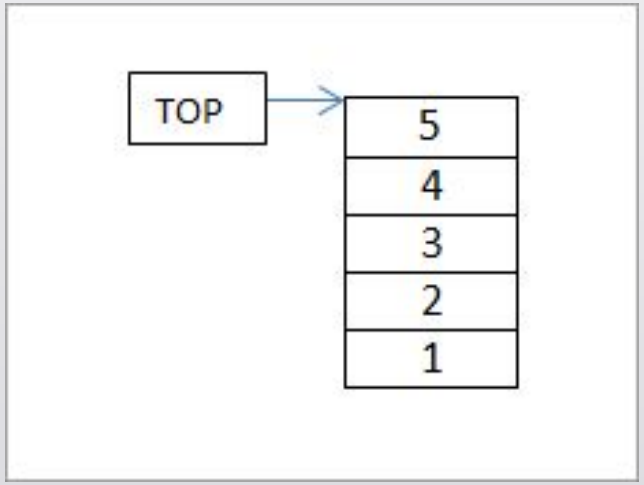
Output :

```
1) Insert element to queue
2) Delete element from queue
3) Display all the elements of queue
4) Exit
Enter your choice : 1
Insert the element in queue : 4
Enter your choice : 1
Insert the element in queue : 3
Enter your choice : 1
Insert the element in queue : 5
Enter your choice : 2
Element deleted from queue is : 4
Enter your choice : 3
Queue elements are : 3 5
Enter your choice : 7
Invalid choice
Enter your choice : 4
Exit
```

Implementation of Stack using STL

Stacks

- Stack container in STL is a type of container adaptors.
- It is used to replicate a stack data structure in C++.
- Stack container is a set of elements in which the elements are inserted at one end and are also deleted at the same end.
- This common point of addition and deletion is known as “Top of the stack”.
- **The pictorial representation of stack is shown below.**



Stacks

- Since addition and deletion happen at the same end, we can say that stack container is LIFO (last in, first out) type of work.

- This means, that the element added first will be the last one to be deleted.
- **To implement stack container, the header <stack> is included in our program.**

#include<stack>

- **The general declaration syntax for stack container is:**

stack<objectType> stackName;

Stack

Functionalities served by Stack are as follows:

- ✔ **stack.push(element):** It inserts an element to the top of the stack.
- ✔ **stack.pop():** It deletes an element present at the top of the stack.
- ✔ **stack.empty():** It checks whether the stack is empty or not.
- ✔ **stack.top():** It returns the element present at the top of the stack.

EXAMPLE :

```
#include <bits/stdc++.h>
using namespace std;
 void display(stack <int> S)
{
    while (!S.empty())
    {
        cout << '\t' << S.top();
        S.pop();
    }
    cout << '\n';
}
int main ()
{
    stack <int> s;
    s.push(1);
    s.push(0);
    s.push(2);
    s.push(6);
    cout << "The stack is : ";
    display(s);
    cout << "\nThe top element of the stack:\n" << s.top();
    cout << "\nStack after removing the top element from the stack:\n";
    s.pop();
    display(s);
     return 0;
}
```

Output:

The stack is : 6 2 0 1

The top element of the stack:

6

Stack after removing the top element from the stack:

2 0 1

Example:

```
#include <iostream>
#include <stack>
using namespace std;
void printStack(stack <int> stk)
{
   while (!stk.empty())
      {
         cout << '\t' << stk.top();
         stk.pop();
      }
   cout << '\n';
}
```

```
int main ()
{
   stack <int> oddstk;
   oddstk.push(1);
   oddstk.push(3);
   oddstk.push(5);
   oddstk.push(7);
   oddstk.push(9);
   cout << "The stack is : ";
   printStack(oddstk);
   cout << "\nSize of stack: " << oddstk.size();
   cout << "\nTop of stack: " << oddstk.top();
   cout << "\noddstk.pop() : ";
   oddstk.pop();
   printStack(oddstk);
   cout<<"\nAnother pop(): ";
   oddstk.pop();
   printStack(oddstk);
   return 0;
}
```

Output:

The Stack is : 9 7 5 3 1

Size of Stack: 5

Top of Stack: 9

Oddstk.pop(): 7 5 3 1

Another pop(): 5 3 1

Implementation of Queue using STL

Queue

- The queue is yet another container in STL which is very simple and useful too. Queue container is a replica of the queue data structure in C++.
- Unlike stack, in the queue container, there are two ends, i.e. front, and rear(back).
- Elements are added to the queue at the back while deleted from the front of the queue.
- In general, queue uses FIFO (First in, First Out) type of arrangement.
- To implement a queue container in a program, we have to include a header <queue> in the code.

Queue

- **To implement queue container, the header <queue> is included in our program.**

- **#include <queue>**
- The general syntax for declaration of the queue is:

queue<objectType> queue_name;

- We declare the queue container as follows:

Queue<int> myqueue;

Queue

Some of the commonly used functions offered by generic Queue:

- ✔ **queue.empty():** Checks whether the queue is empty or not.
- ✔ **queue.push(element):** This functions adds an element to the end of the queue.
- ✔ **queue.pop():** It deletes the first element of the queue.
- ✔ **queue.front():** It returns an iterator element which points to the first element of the queue.
- ✔ **queue.back():** It returns an iterator element which points to the last element of the queue.

Example:

```
#include <iostream>
#include <queue>

using namespace std;

void display(queue <int> Q1)
{
   queue <int> Q = Q1;
   while (!Q.empty())
   {
      cout << '\t' << Q.front();
      Q.pop();
   }
   cout << '\n';
}
```

```
int main()
{
   int i=1;
   queue <int> qd;
   while (i<5)
   {
      qd.push(i);
      i++;
   }

    cout << "Queue:\n";
   display(qd);
   cout<<"Popping an element from the queue..\n";
   qd.pop();
   display(qd);
   return 0;
}
```

Output

Queue:

1 2 3 4

Popping an element from the queue..

2 3 4

Example:

```
#include <iostream>
#include <queue>
using namespace std;
void printQueue(queue <int> myqueue)
{
   queue <int> secqueue = myqueue;
   while (!secqueue.empty())
   {
      cout << '\t' << secqueue.front();
      secqueue.pop();
   }
   cout << '\n';
}
```

```
int main()
  {
     queue <int> myqueue;
     myqueue.push(2);
     myqueue.push(4);
     myqueue.push(6);
     myqueue.push(8);
     cout << "The queue myqueue is : ";
     printQueue(myqueue);
     cout << "\nmyqueue.size() : " << myqueue.size();
     cout << "\nmyqueue.front() : " << myqueue.front();
     cout << "\nmyqueue.back() : " << myqueue.back();
     cout << "\nmyqueue.pop() : ";
     myqueue.pop();
     printQueue(myqueue);
     return 0;
  }
```

Output:

The queue myqueue is : 2 4 6 8

myqueue.size() : 4

myqueue.front() : 2

myqueue.back() : 8

myqueue.pop() : 4 6 8

Priority queue

In priority queue, the elements are placed in decreasing order of their values and the first element represents the largest of all the inserted elements.

Syntax:

priority_queue <data_type> priority_queue_name;

Some of the functions offered by Priority queue:

- ✔ **priority_queue.empty():** Checks whether the queue is empty or not.
- ✔ **priority_queue.top():** It returns the top element from the queue.
- ✔ **priority_queue.pop():** It deletes the first element from the queue
- ✔ **priority_queue.push(element):** It inserts the element at the end of the queue.
- ✔ **priority_queue.swap():** It swaps the elements of one priority queue with another of the similar data type and size.
- ✔ **priority_queue.size():** It returns the number of elements present in the priority queue.

Example:

```
#include <iostream>
#include <queue>
 using namespace std;
void display(priority_queue <int> PQ)
{
   priority_queue <int> p = PQ;
   while (!p.empty())
   {
      cout << '\t' << p.top();
      p.pop();
   }
   cout << '\n';
}
int main ()
{    int i=1;
   priority_queue <int> PQ;
   while(i<5)
   {
      PQ.push(i*10);
      i++;
   }
        cout << "The priority queue:\n";
   display(PQ);

   cout << "\nThe priority_queue.top() function:\n" << PQ.top();

   cout << "\nThe priority_queue.pop() function:\n";
   PQ.pop();
   display(PQ);
    return 0;
}
```

Output:

The priority queue:

40 30 20 10

The priority_queue.top() function:

40

The priority_queue.pop() function:

30 20 10

References

- https://www.hackerearth.com/practice/notes/stacks-and-queues/

- https://www.softwaretestinghelp.com/stacks-and-queues-in-stl/