UNIT-4

STACKS & QUEUES

STACKS

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a ***stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.***

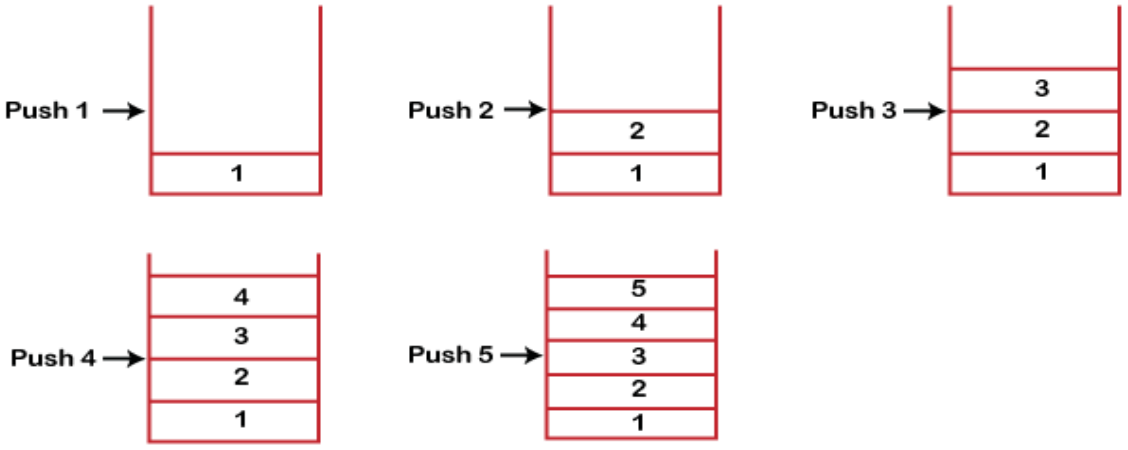
Some key points related to stack

- ✓ It is called as stack because it behaves like a real-world stack, piles of books, etc.
- ✓ A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- ✓ It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were

entering the new element in the stack. The stack gets filled up from the bottom to the top.

When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

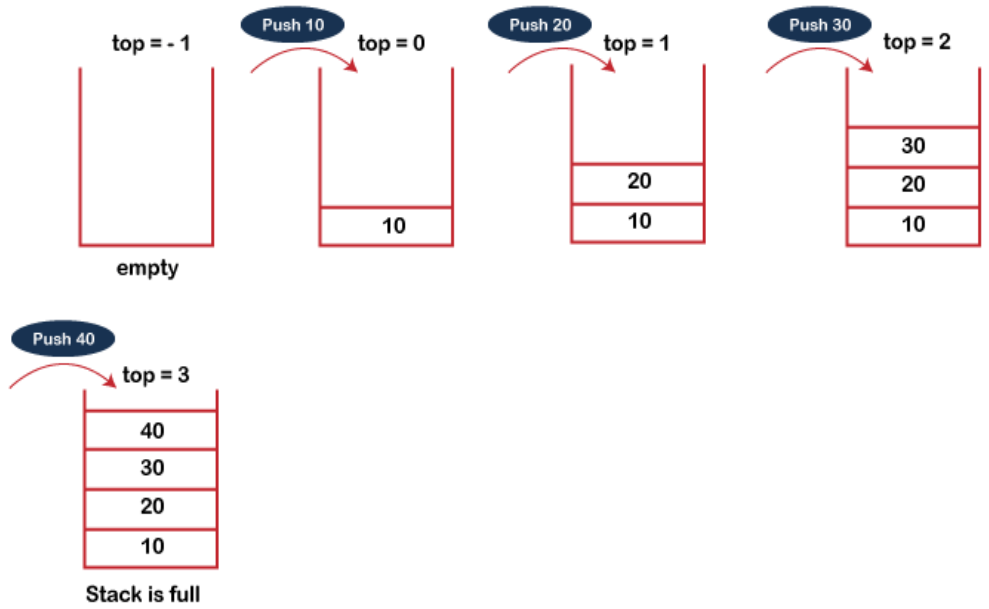
The following are some common operations implemented on the stack:

- ✓ **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- ✓ **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- ✓ **isEmpty():** It determines whether the stack is empty or not.
- ✓ **isFull():** It determines whether the stack is full or not.'
- ✓ **peek():** It returns the element at the given position.
- ✓ **count():** It returns the total number of elements available in a stack.
- ✓ **change():** It changes the element at the given position.
- ✓ **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

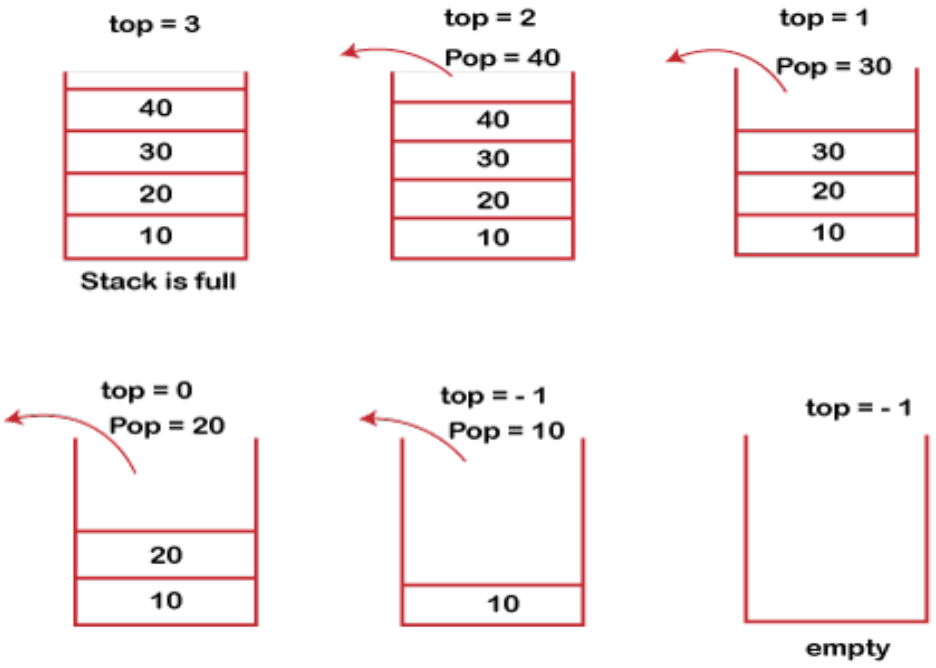
- ✓ Before inserting an element in a stack, we check whether the stack is full.
- ✓ If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- ✓ When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- ✓ When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1**, and the element will be placed at the new position of the **top**.
- ✓ The elements will be inserted until we reach the ***max*** size of the stack.



POP operation

The steps involved in the POP operation is given below:

- ✓ Before deleting the element from the stack, we check whether the stack is empty.
- ✓ If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- ✓ If the stack is not empty, we first access the element which is pointed by the ***top***
- ✓ Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



Applications of Stack

The following are the applications of the stack:

- **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:

As we know, each program has *an opening* and *closing* braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- ✓ **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a **"AdityaITM"** string, so we can achieve this with the help of a stack.
  First, we push all the characters of the string in a stack until we reach the null character.
  After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- ✓ **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- ✓ **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- ✓ **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
- ✓ **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- ✓ **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
  - ✓ Infix to prefix

- ✓ Infix to postfix
- ✓ Prefix to infix
- ✓ Prefix to postfix
- ✓ Postfix to infix

✓ **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Array implementation of Stack

program that implements a stack using array is given as follows.

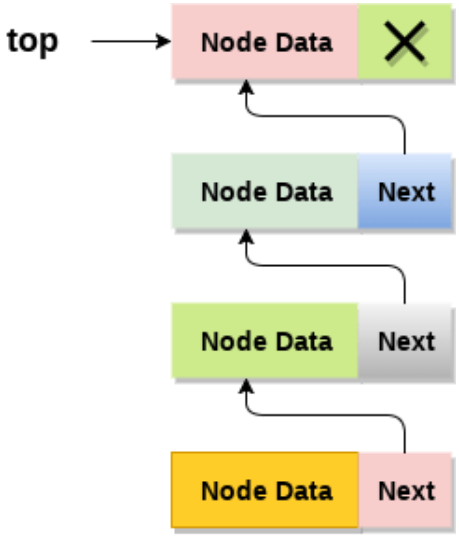
```
#include <iostream>
using namespace std;
int stack[100], n=100, top=-1;

void push(int val) {
   if(top>=n-1)
   cout<<"Stack Overflow"<<endl;
   else {
      top++;
      stack[top]=val;
   }
}
void pop() {
   if(top<=-1)
   cout<<"Stack Underflow"<<endl;
   else {
      cout<<"The popped element is "<< stack[top] <<endl;
      top--;
   }
}
void display() {
   if(top>=0) {
      cout<<"Stack elements are:";
      for(int i=top; i>=0; i--)
      cout<<stack[i]<<" ";
      cout<<endl;
   } else
   cout<<"Stack is empty";
}
int main() {
   int ch, val;
   cout<<"1) Push in stack"<<endl;
   cout<<"2) Pop from stack"<<endl;
   cout<<"3) Display stack"<<endl;
   cout<<"4) Exit"<<endl;
```

```
    do {
        cout<<"Enter choice: "<<endl;
        cin>>ch;
        switch(ch) {
            case 1: {
                cout<<"Enter value to be pushed:"<<endl;
                cin>>val;
                push(val);
                break;
            }
            case 2: {
                pop();
                break;
            }
            case 3: {
                display();
                break;
            }
            case 4: {
                cout<<"Exit"<<endl;
                break;
            }
            default: {
                cout<<"Invalid Choice"<<endl;
            }
        }
    }while(ch!=4);
    return 0;
}
```

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



Stack

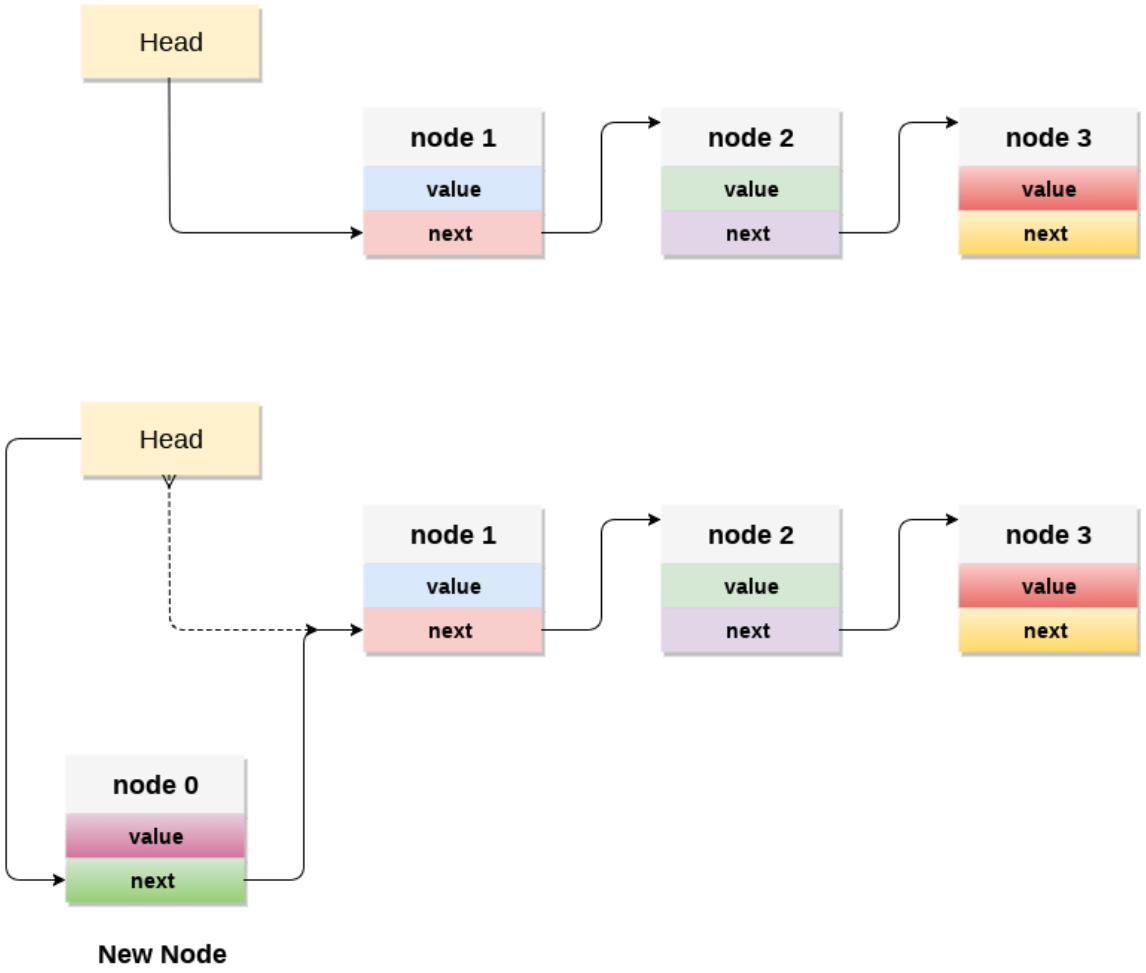
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

- ✓ Create a node first and allocate memory to it.
- ✓ If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
- ✓ If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

Time Complexity : o(1)



C++ implementation :

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    struct Node *next;
};
struct Node* top = NULL;
void push(int val) {
    struct Node* newnode = (struct Node*) malloc(sizeof(struct Node));
    newnode->data = val;
    newnode->next = top;
    top = newnode;
}
void pop() {
    if(top==NULL)
    cout<<"Stack Underflow"<<endl;
    else {
        cout<<"The popped element is "<< top->data <<endl;
        top = top->next;
    }
}
void display() {
    struct Node* ptr;
    if(top==NULL)
    cout<<"stack is empty";
```

```
    else {
        ptr = top;
        cout<<"Stack elements are: ";
        while (ptr != NULL) {
            cout<< ptr->data <<" ";
            ptr = ptr->next;
        }
    }
    cout<<endl;
}
int main() {
    int ch, val;
    cout<<"1) Push in stack"<<endl;
    cout<<"2) Pop from stack"<<endl;
    cout<<"3) Display stack"<<endl;
    cout<<"4) Exit"<<endl;
    do {
        cout<<"Enter choice: "<<endl;
        cin>>ch;
        switch(ch) {
            case 1: {
                cout<<"Enter value to be pushed:"<<endl;
                cin>>val;
                push(val);
                break;
            }
            case 2: {
                pop();
                break;
            }
            case 3: {
                display();
                break;
            }
            case 4: {
                cout<<"Exit"<<endl;
                break;
            }
            default: {
                cout<<"Invalid Choice"<<endl;
            }
        }
    }while(ch!=4);
    return 0;
}
```

Queue

Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Queue can also be defined as the list or collection in which the insertion is done from one end known as the **rear end** or the **tail** of the queue, whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last. Similar approach is followed in the queue in data structure.

- ✓ A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT.**
- ✓ Queue is referred to be as **First In First Out** list.
- ✓ For example, people waiting in line for a rail ticket form a queue.

The representation of the queue is shown in the below image -

Applications of Queue

- ✓ Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.
- ✓ Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- ✓ Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
- ✓ Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
- ✓ Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
- ✓ Queues are used in operating systems for handling interrupts.

Complexity:

Data Structure	**Time Complexity**								**Space Compleity**
		Average			**Worst**				**Worst**
	Acce ss	Sear ch	Inse rtio n	Dele tion	Acce ss	Sear ch	Inse rtio n	Dele tion	

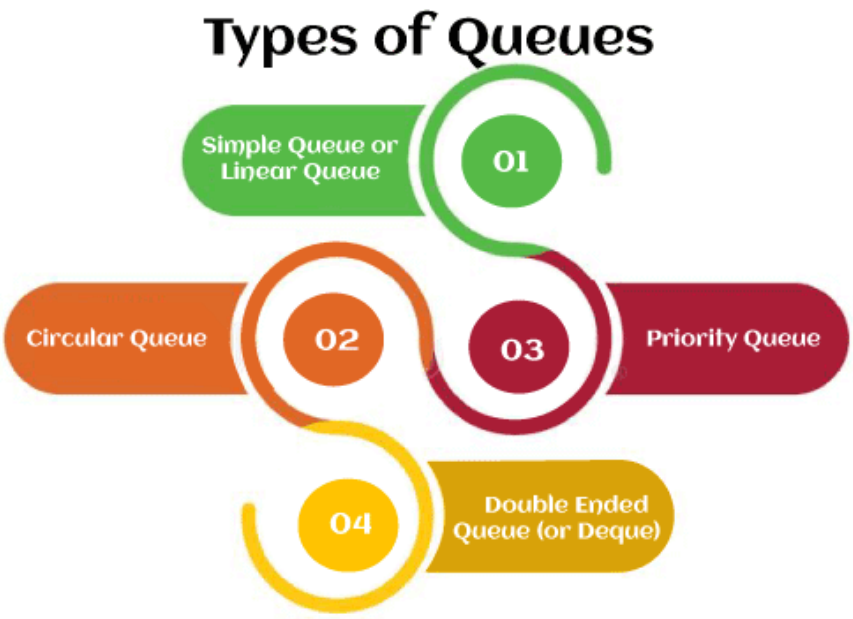
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Types of Queue

There are four different types of queue that are listed as follows –

- ✓ Simple Queue or Linear Queue
- ✓ Circular Queue
- ✓ Priority Queue
- ✓ Double Ended Queue (or Deque)

Let's discuss each of the type of queue.



Simple Queue or Linear Queue

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.

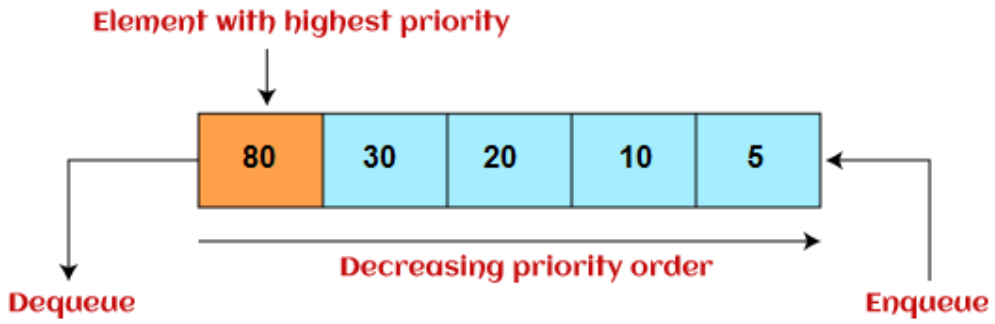
The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a

Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

The drawback that occurs in a linear queue is overcome by using the

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows -

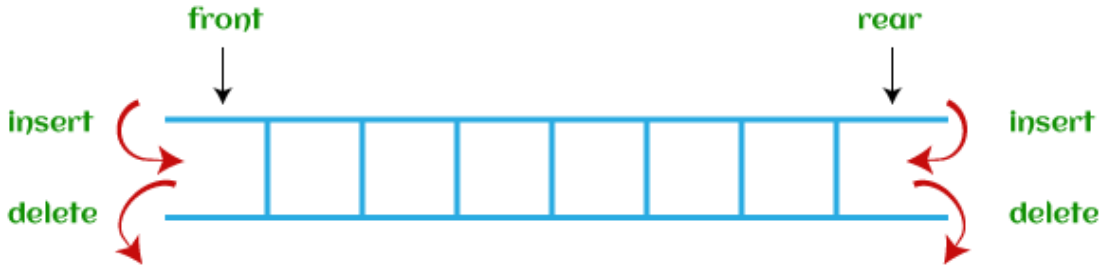
- ✓ **Ascending priority queue -** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
- ✓ **Descending priority queue -** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the

queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

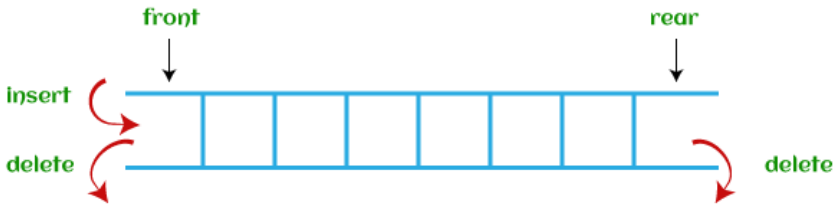
Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.



double ended queue

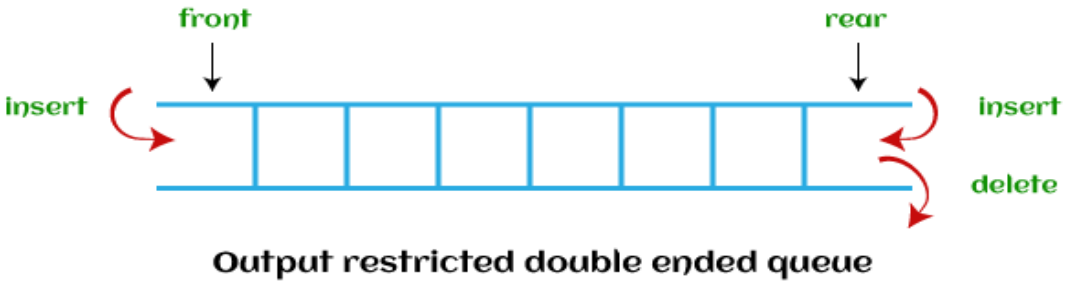
There are two types of deque that are discussed as follows -

- **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

- **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Now, let's see the operations performed on the queue.

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- ✓ **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- ✓ **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
- ✓ **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- ✓ **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
- ✓ **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

Now, let's see the ways to implement the queue.

Ways to implement the queue

There are two ways of implementing the Queue:

- o **Implementation using array:** The sequential allocation in a Queue can be implemented using an array. For more details,
- o **Implementation using Linked list:** The linked list allocation in a Queue can be implemented using a linked list. For more details,

Implementation using array in CPP programming language:

```
#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;
   }
}
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;}
```

Implementation using Linked List in CPP programming language:

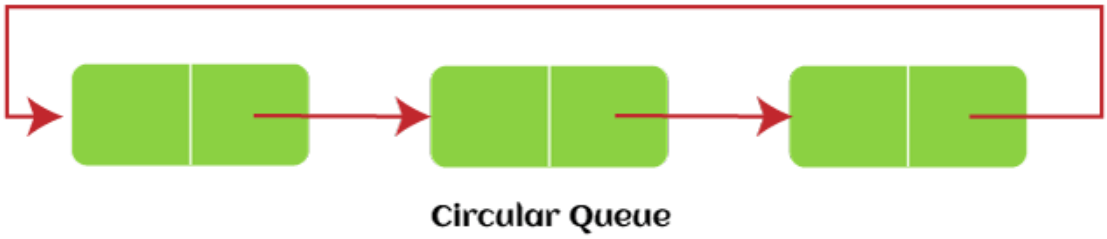
```
#include <iostream>
using namespace std;
struct node {
 int data;
 struct node *next;
};
struct node* front = NULL;
struct node* rear = NULL;
struct node* temp;
void Insert() {
 int val;
 cout<<"Insert the element in queue : "<<endl;
 cin>>val;
 if (rear == NULL) {
 rear = (struct node *)malloc(sizeof(struct node));
 rear->next = NULL;
 rear->data = val;
 front = rear;
 } else {
```

```
 temp=(struct node *)malloc(sizeof(struct node));
 rear->next = temp;
 temp->data = val;
 temp->next = NULL;
 rear = temp;
 }
}
void Delete() {
 temp = front;
 if (front == NULL) {
 cout<<"Underflow"<<endl;
 return;
 }
 else
 if (temp->next != NULL) {
 temp = temp->next;
 cout<<"Element deleted from queue is : "<<front->data<<endl;
 free(front);
 front = temp;
 } else {
 cout<<"Element deleted from queue is : "<<front->data<<endl;
 free(front);
 front = NULL;
 rear = NULL;
 }
}
void Display() {
 temp = front;
 if ((front == NULL) && (rear == NULL)) {
 cout<<"Queue is empty"<<endl;
 return;
 }
 cout<<"Queue elements are: ";
 while (temp != NULL) {
 cout<<temp->data<<" ";
 temp = temp->next;
 }
 cout<<endl;
}
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;
```

```
}
```

Circular Queue

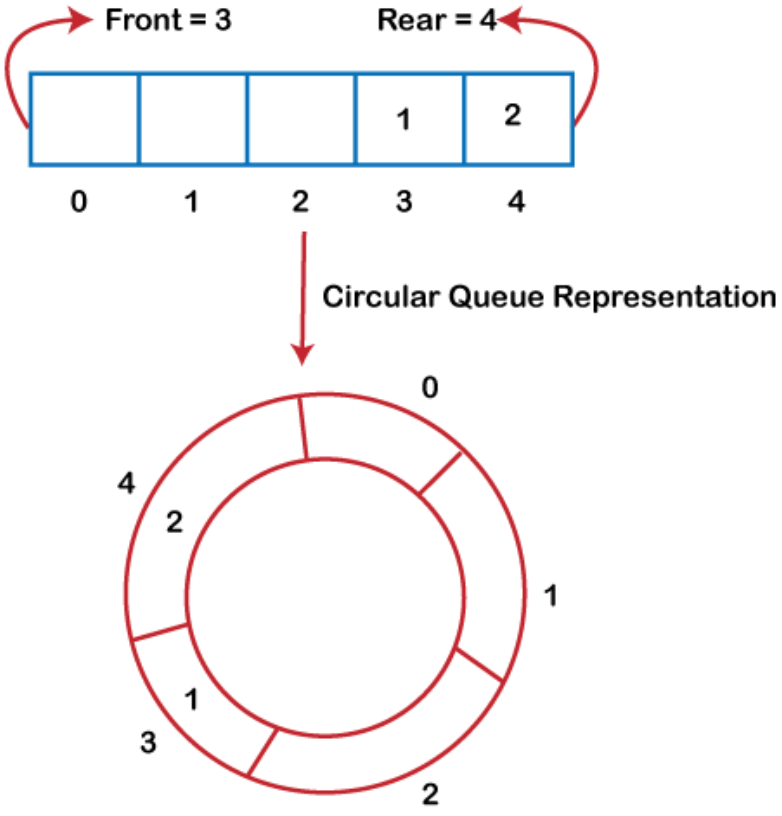
Why was the concept of the circular queue introduced?



Circular Queue

There was one limitation in the array implementation of Queue.

If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0^{th} position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

The main advantage of using the circular queue is better memory utilization

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- ✓ **Front:** It is used to get the front element from the Queue.
- ✓ **Rear:** It is used to get the rear element from the Queue.
- ✓ **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- ✓ **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- ✓ **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- ✓ **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
- ✓ **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

The steps of enqueue operation are given below:

- ✓ First, we will check whether the Queue is full or not.
- ✓ Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.

- ✓ When we insert a new element, the rear gets incremented, i.e., ***rear=rear+1***.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- ✓ **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- ✓ **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

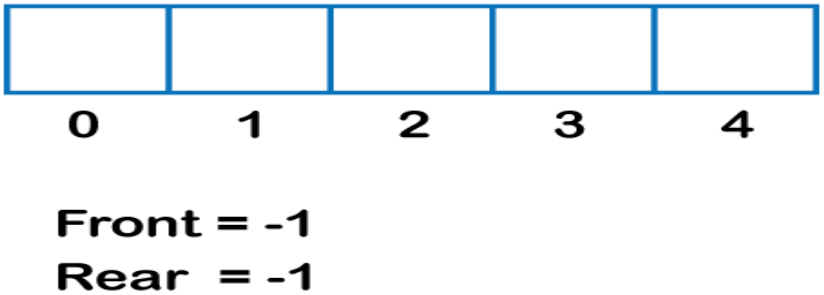
- ✓ When **front ==0** && **rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- ✓ front== rear + 1;

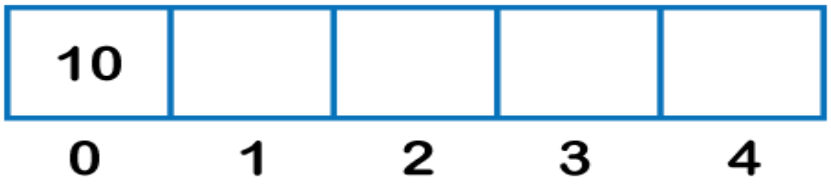
Dequeue Operation

The steps of dequeue operation are given below:

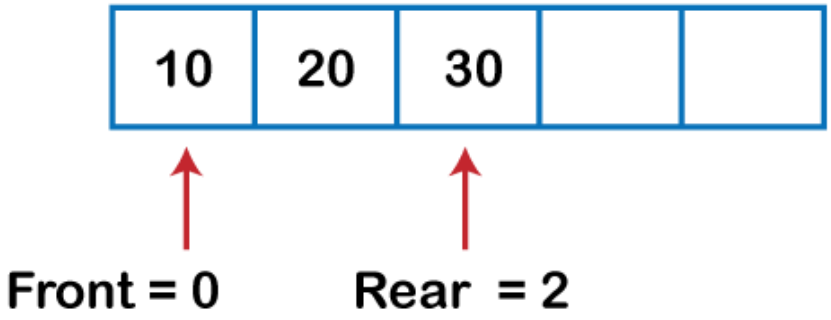
- ✓ First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- ✓ When the element is deleted, the value of front gets decremented by 1.
- ✓ If there is only one element left which is to be deleted, then the front and rear are reset to -1.

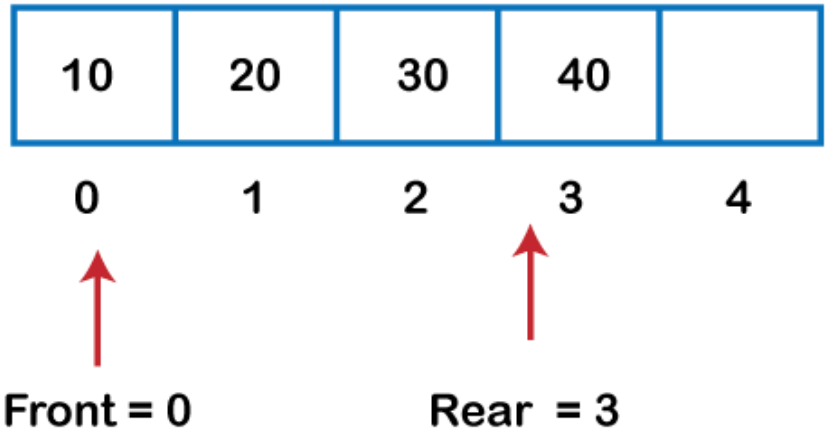
Let's understand the enqueue and dequeue operation through the diagrammatic representation.

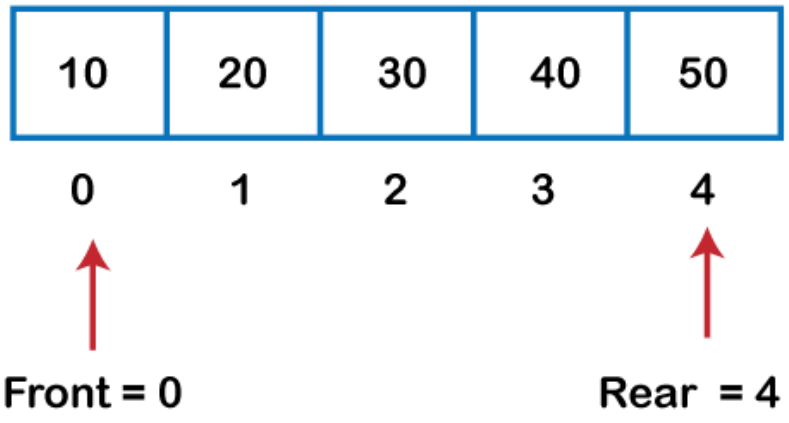


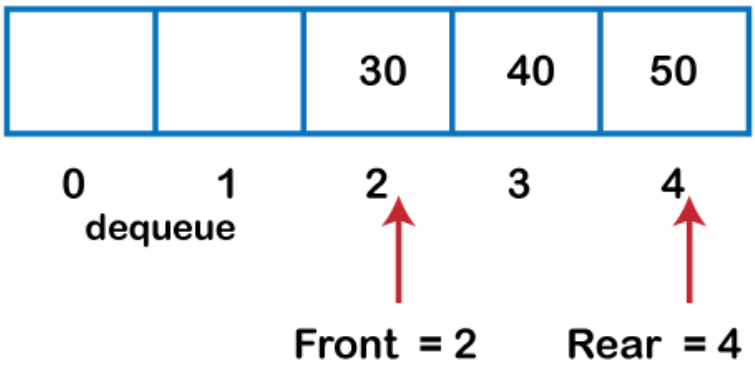


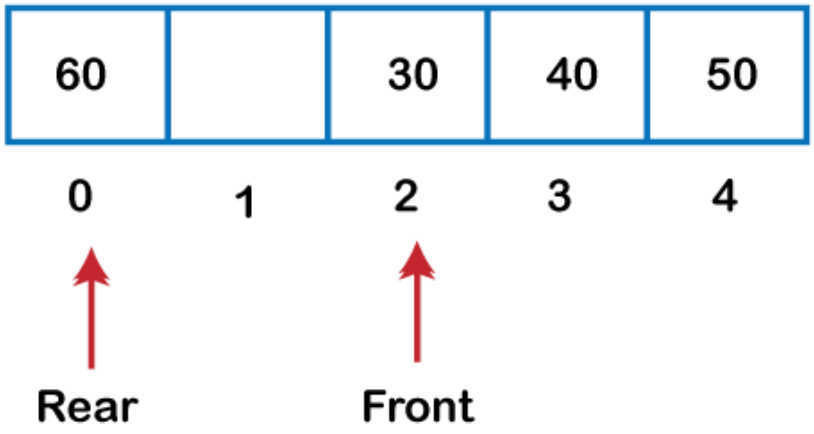
Front = 0
Rear = 0

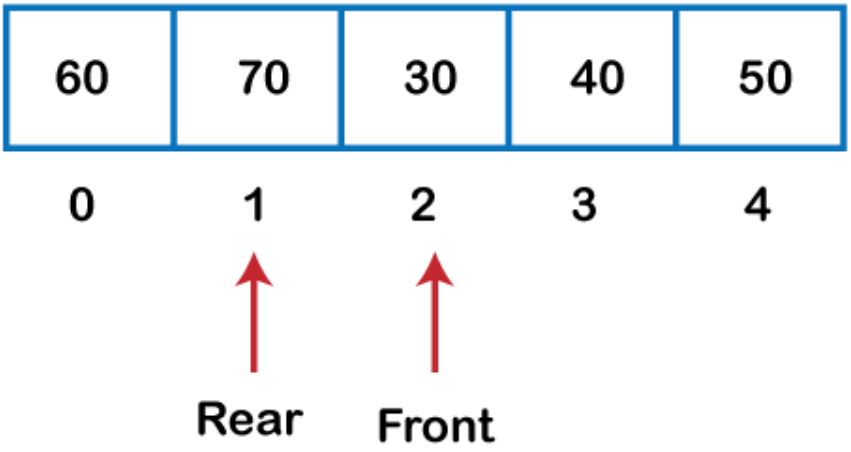












Implementation of circular queue using Array

```
#include <iostream>
using namespace std;

int cqueue[5];
int front = -1, rear = -1, n=5;

void insertCQ(int val) {
   if ((front == 0 && rear == n-1) || (front == rear+1)) {
      cout<<"Queue Overflow \n";
```

```
        return;
    }
    if (front == -1) {
        front = 0;
        rear = 0;
    }
        else
        rear = (rear + 1)%n;

    cqueue[rear] = val ;
}
void deleteCQ() {
    if (front == -1) {
        cout<<"Queue Underflow\n";
        return ;
    }
    cout<<"Element deleted from queue is : "<<cqueue[front]<<endl;

    if (front == rear) {
        front = -1;
        rear = -1;
    }
        else
        front = (front + 1)%n;

}
void displayCQ() {
    int f = front, r = rear;
    if (front == -1) {
        cout<<"Queue is empty"<<endl;
        return;
    }
    cout<<"Queue elements are :\n";
    if (f <= r) {
        while (f <= r){
            cout<<cqueue[f]<<" ";
            f++;
        }
    } else {
        while (f <= n - 1) {
            cout<<cqueue[f]<<" ";
            f++;
        }
        f = 0;
        while (f <= r) {
            cout<<cqueue[f]<<" ";
            f++;
        }
    }
    cout<<endl;
}
int main() {

    int ch, val;
    cout<<"1)Insert\n";
    cout<<"2)Delete\n";
```

```
        cout<<"3)Display\n";
        cout<<"4)Exit\n";
        do {
           cout<<"Enter choice : "<<endl;
           cin>>ch;
           switch(ch) {
              case 1:
              cout<<"Input for insertion: "<<endl;
              cin>>val;
              insertCQ(val);
              break;

              case 2:
              deleteCQ();
              break;

              case 3:
              displayCQ();
              break;

              case 4:
              cout<<"Exit\n";
              break;
              default: cout<<"Incorrect!\n";
           }
        } while(ch != 4);
        return 0;
     }
```

What is a Deque (or double-ended queue)

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -

Representation of deque

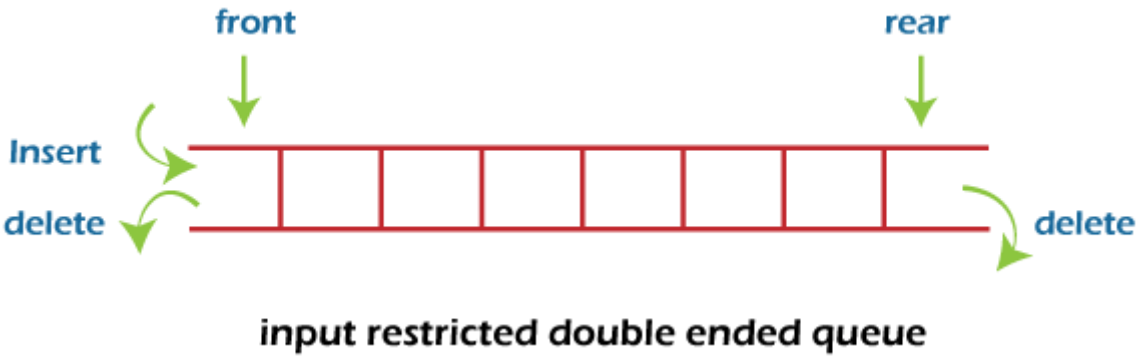
Types of deque

There are two types of deque -

- ✓ Input restricted queue
- ✓ Output restricted queue

Input restricted Queue

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.

Output restricted double ended queue

Operations performed on deque

There are the following operations that can be applied on a deque

- ✓ Insertion at front
- ✓ Insertion at rear
- ✓ Deletion at front
- ✓ Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque -

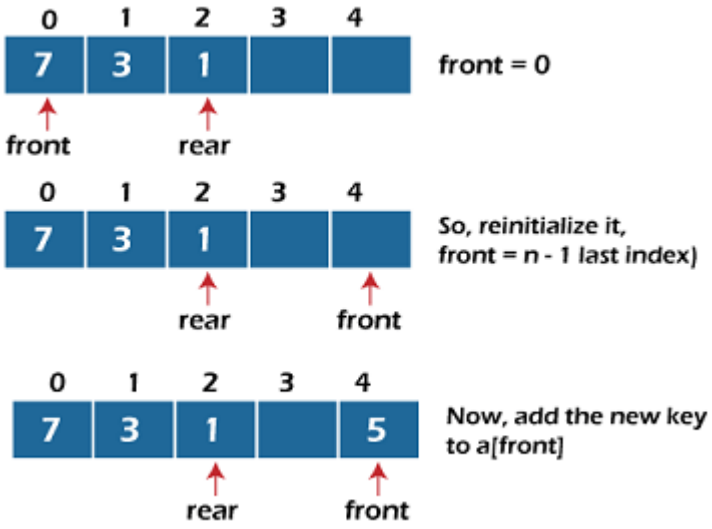
- ✓ Get the front item from the deque
- ✓ Get the rear item from the deque
- ✓ Check whether the deque is full or not
- ✓ Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

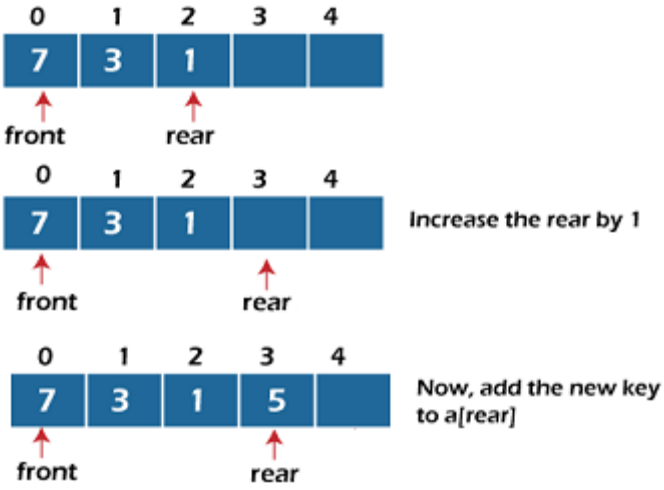
- ✓ If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- ✓ Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -

- ✓ If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- ✓ Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



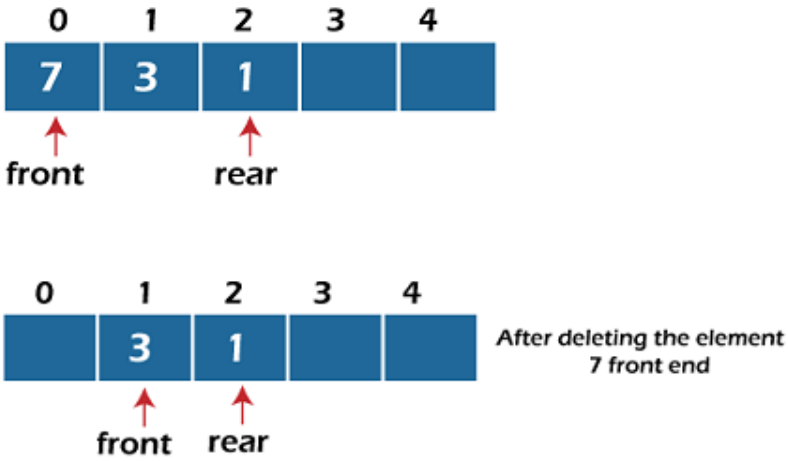
Deletion at the front end

In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

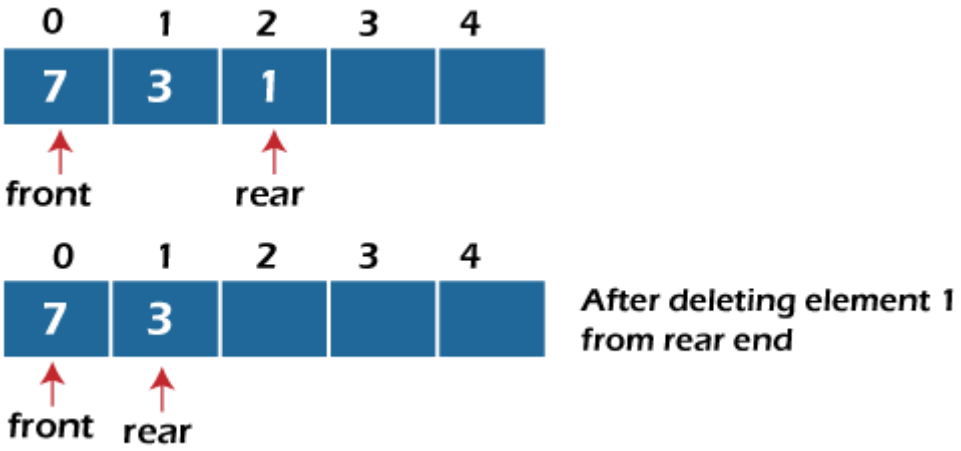
If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).

the below conditions -



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Applications of deque

- ✓ Deque can be used as both stack and queue, as it supports both operations.
- ✓ Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Implementation of deque

```
#include<iostream>
using namespace std;
#define SIZE 10
class dequeue {
   int a[20],f,r;
   public:
      dequeue();
      void insert_at_beg(int);
      void insert_at_end(int);
      void delete_fr_front();
```

```
        void delete_fr_rear();
        void show();
};
dequeue::dequeue() {
    f=-1;
    r=-1;
}
void dequeue::insert_at_end(int i) {
    if(r>=SIZE-1) {
        cout<<"\n insertion is not possible, overflow!!!!";
    } else {
        if(f==-1) {
            f++;
            r++;
        } else {
            r=r+1;
        }
        a[r]=i;
        cout<<"\nInserted item is"<<a[r];
    }
}
void dequeue::insert_at_beg(int i) {
    if(f==-1) {
        f=0;
        a[++r]=i;
        cout<<"\n inserted element is:"<<i;
    } else if(f!=0) {
        a[--f]=i;
        cout<<"\n inserted element is:"<<i;
    } else {
        cout<<"\n insertion is not possible, overflow!!!";
    }
}
void dequeue::delete_fr_front() {
    if(f==-1) {
        cout<<"deletion is not possible::dequeue is empty";
        return;
    }
    else {
        cout<<"the deleted element is:"<<a[f];
        if(f==r) {
            f=r=-1;
            return;
        } else
            f=f+1;
        }
    }
    void dequeue::delete_fr_rear() {
        if(f==-1) {
            cout<<"deletion is not possible::dequeue is empty";
            return;
        }
        else {
            cout<<"the deleted element is:"<<a[r];
            if(f==r) {
                f=r=-1;
```

```
        } else
            r=r-1;
      }
   }
   void dequeue::show() {
      if(f==-1) {
         cout<<"Dequeue is empty";
      } else {
         for(int i=f;i<=r;i++) {
            cout<<a[i]<<" ";
         }
      }
   }
   int main() {
      int c,i;
      dequeue d;
      Do//perform switch opeartion {
      cout<<"\n 1.insert at beginning";
      cout<<"\n 2.insert at end";
      cout<<"\n 3.show";
      cout<<"\n 4.deletion from front";
      cout<<"\n 5.deletion from rear";
      cout<<"\n 6.exit";
      cout<<"\n enter your choice:";
      cin>>c;
      switch(c) {
         case 1:
            cout<<"enter the element to be inserted";
            cin>>i;
            d.insert_at_beg(i);
         break;
         case 2:
            cout<<"enter the element to be inserted";
            cin>>i;
            d.insert_at_end(i);
         break;
         case 3:
            d.show();
         break;
         case 4:
            d.delete_fr_front();
         break;
         case 5:
            d.delete_fr_rear();
         break;
         case 6:
            exit(1);
         break;
         default:
            cout<<"invalid choice";
         break;
      }
   } while(c!=7);
}
```

Implementation of Deque

```
#include<iostream>
using namespace std;

#define MAX 100

class Deque
{
    int  arr[MAX];
    int  front;
    int  rear;
    int  size;
public :
    Deque(int size)
    {
        front = -1;
        rear = 0;
        this->size = size;
    }


    void  insertfront(int key);
    void  insertrear(int key);
    void  deletefront();
    void  deleterear();
    bool  isFull();
    bool  isEmpty();
    int  getFront();
    int  getRear();
};

bool Deque::isFull()
{
    return ((front == 0 && rear == size-1)||
            front == rear+1);
}

bool Deque::isEmpty ()
{
    return (front == -1);
}

void Deque::insertfront(int key)
{

    if (isFull())
    {
        cout << "Overflow\n" << endl;
        return;
    }


    if (front == -1)
    {
        front = 0;
```

```
        rear = 0;
    }

    else if (front == 0)
        front = size - 1 ;

    else
        front = front-1;

    arr[front] = key ;
}

void Deque ::insertrear(int key)
{
    if (isFull())
    {
        cout << " Overflow\n " << endl;
        return;
    }


    if (front == -1)
    {
        front = 0;
        rear = 0;
    }


    else if (rear == size-1)
        rear = 0;


    else
        rear = rear+1;


    arr[rear] = key ;
}


void Deque ::deletefront()
{

    if (isEmpty())
    {
        cout << "Queue Underflow\n" << endl;
        return ;
    }


    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else
```

```
        if (front == size -1)
            front = 0;

        else
            front = front+1;
}


void Deque::deleterear()
{
    if (isEmpty())
    {
        cout << " Underflow\n" << endl ;
        return ;
    }


    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else if (rear == 0)
        rear = size-1;
    else
        rear = rear-1;
}


int Deque::getFront()
{

    if (isEmpty())
    {
        cout << " Underflow\n" << endl;
        return -1 ;
    }
    return arr[front];
}


int Deque::getRear()
{

    if(isEmpty() || rear < 0)
    {
        cout << " Underflow\n" << endl;
        return -1 ;
    }
    return arr[rear];
}


int main()
{
```

```
    Deque dq(5);
    cout << "Insert element at rear end  : 5 \n";
    dq.insertrear(5);

    cout << "insert element at rear end : 10 \n";
    dq.insertrear(10);

    cout << "get rear element " << " "
         << dq.getRear() << endl;

    dq.deleterear();
    cout << "After delete rear element new rear"
         << " become " << dq.getRear() << endl;

    cout << "inserting element at front end \n";
    dq.insertfront(15);

    cout << "get front element " << " "
         << dq.getFront() << endl;

    dq.deletefront();

    cout << "After delete front element new "
       << "front become " << dq.getFront() << endl;
    return 0;

}
```

***Classic problems using Stacks and Queues

Balancing Symbols

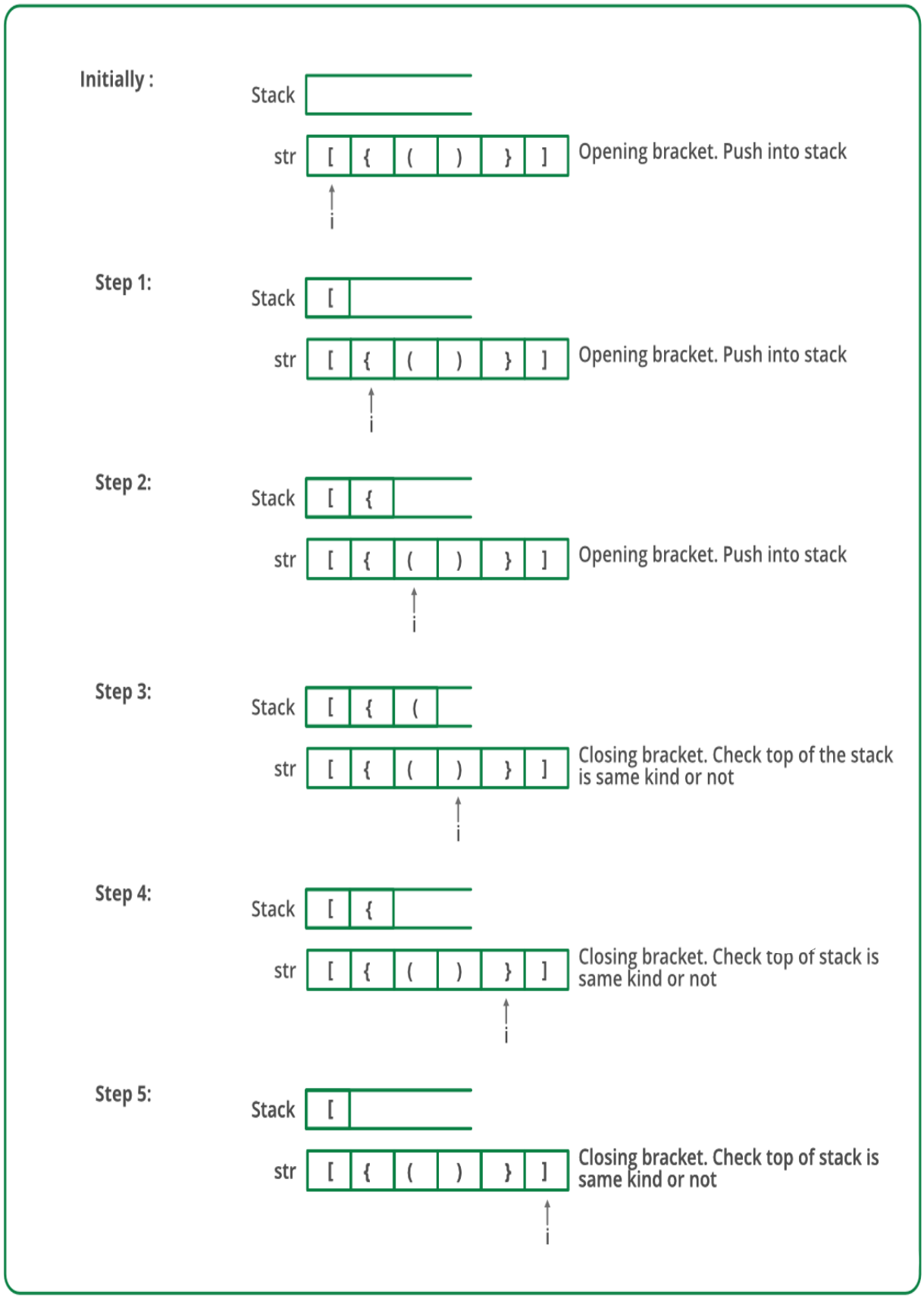
Approach:

The idea is to put all the opening brackets in the stack and whenever you hit a closing bracket. Search if the top of the stack is the opening bracket of the same nature. If this holds then pop the stack and continue the iteration , in the end if the stack is empty, it means all brackets are well-formed and return Balanced , else return Not Balanced.

Illustration:

Below is the illustration of above approach.
Follow the steps mentioned below to implement the idea:

- ✓ Declare a character stack **temp**.
- ✓ Now traverse the string exp.
  - ✓ If the current character is a starting bracket (**(or { or [**) then push it to stack.
  - ✓ If the current character is a closing bracket (**) or } or]**) then pop from stack and if the popped character is the matching starting bracket then fine else brackets are **Not Balanced**.
- ✓ After complete traversal, if there is some starting bracket left in stack then **Not balanced** , else **Balanced**.



Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;


bool areBracketsBalanced(string expr)
```

```
{

    stack<char> temp;
    for (int i = 0; i < expr.length(); i++) {
        if (temp.empty()) {

            temp.push(expr[i]);
        }
        else if ((temp.top() == '(' && expr[i] == ')')
                 || (temp.top() == '{' && expr[i] == '}')
                 || (temp.top() == '[' && expr[i] == ']')) {

            temp.pop();
        }
        else {
            temp.push(expr[i]);
        }
    }
    if (temp.empty()) {

        return true;
    }
    return false;
}


int main()
{
    string expr = "{()}[]";


    if (areBracketsBalanced(expr))
        cout << "Balanced";
    else
        cout << "Not Balanced";
    return 0;
}
```

Reverse a string using Stack

Approach:
The idea is to create an empty stack and push all the characters from the string into it. Then pop each character one by one from the stack and put them back into the input string starting from the 0'th index. As we all know, stacks work on the principle of first in, last out. After popping all the elements and placing them back to string, the formed string would be reversed.
Follow the steps given below to reverse a string using stack.

- Create an empty stack.

- One by one push all characters of string to stack.
- One by one pop all characters from stack and put them back to string.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;

class Stack {
public:
    int top;
    unsigned capacity;
    char* array;
};

Stack* createStack(unsigned capacity)
{
    Stack* stack = new Stack();
    stack->capacity = capacity;
    stack->top = -1;
    stack->array
        = new char[(stack->capacity * sizeof(char))];
    return stack;
}

int isFull(Stack* stack)
{
    return stack->top == stack->capacity - 1;
}


int isEmpty(Stack* stack) { return stack->top == -1; }

void push(Stack* stack, char item)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = item;
}

char pop(Stack* stack)
{
    if (isEmpty(stack))
        return -1;
    return stack->array[stack->top--];
}


void reverse(char str[])
{

    int n = strlen(str);
    Stack* stack = createStack(n);
```

```
    int i;
    for (i = 0; i < n; i++)
        push(stack, str[i]);

    for (i = 0; i < n; i++)
        str[i] = pop(stack);
}

int main()
{
    char str[] = "AdityaInstitute";

    reverse(str);
    cout << "Reversed string is " << str;

    return 0;
}
```

Implement Undo and Redo features of a Text Editor

Approach: The problem can be solved using Stack. Follow the steps below to solve the problem:

- ✓ Initialize two stacks, say **Undo** and **Redo**.
- ✓ Traverse the array of strings, **Q**, and perform the following operations:
- ✓ If **"WRITE"** string is encountered, push the character to **Undo** stack
- ✓ If **"UNDO"** string is encountered, pop the top element from **Undo** stack and push it to **Redo** stack.
- ✓ If **"REDO"** string is encountered, pop the top element of **Redo** stack and push it into the **Undo** stack.
- ✓ If **"READ"** string is encountered, print all the elements of the **Undo** stack in reverse order.

C++ Program to implement

```
#include <bits/stdc++.h>
using namespace std;

void WRITE(stack<char>& Undo,
           char X)
{

    Undo.push(X);
}

void UNDO(stack<char>& Undo,
          stack<char>& Redo)
{
    char X = Undo.top();


    Undo.pop();
     Redo.push(X);
```

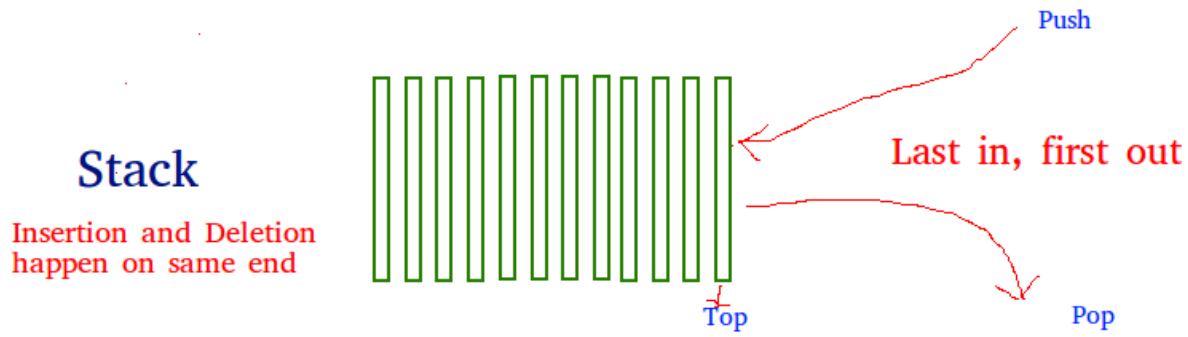
```
}


void REDO(stack<char>& Undo,
          stack<char>& Redo)
{

    char X = Redo.top();


    Redo.pop();
     Undo.push(X);
}

void READ(stack<char> Undo)
{
    stack<char> revOrder;


    while (!Undo.empty()) {

        revOrder.push(Undo.top());

        Undo.pop();
    }

    while (!revOrder.empty()) {

        cout << revOrder.top();
      Undo.push(revOrder.top());

        revOrder.pop();
    }

    cout << " ";
}

void QUERY(vector<string> Q)
{

    stack<char> Undo;

    stack<char> Redo;

    int N = Q.size();

    for (int i = 0; i < N; i++) {
        if (Q[i] == "UNDO") {
            UNDO(Undo, Redo);
        }
        else if (Q[i] == "REDO") {
            REDO(Undo, Redo);
        }
        else if (Q[i] == "READ") {
            READ(Undo);
        }
```

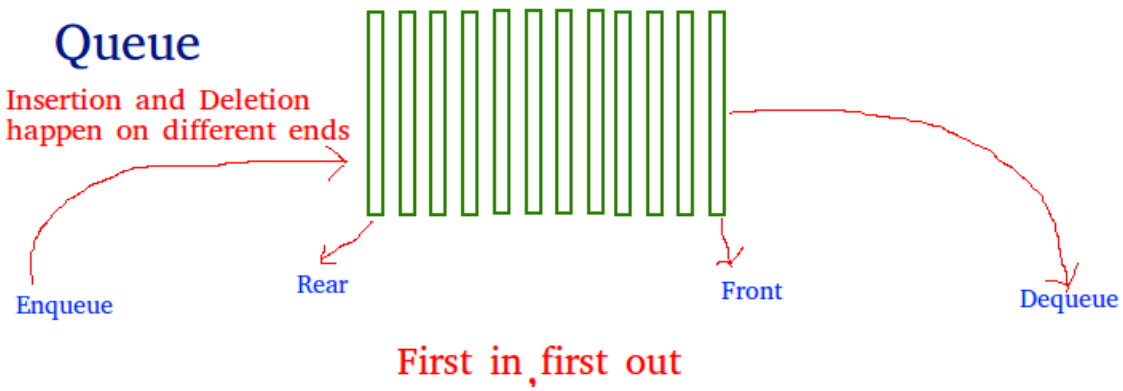
```
        else {
            WRITE(Undo, Q[i][6]);
        }
    }
}


int main()
{

    vector<string> Q = { "WRITE A", "WRITE B",
                         "WRITE C", "UNDO",
                         "READ", "REDO", "READ" };
    QUERY(Q);
    return 0;
}
```

Time Complexity:
UNDO: O(1)
REDO: O(1)
WRITE: O(1)
READ: O(N), where N denotes the size of the Undo stack
Auxiliary Space: O(N)

Queue using Stacks

We are given a stack data structure with push and pop operations, the task is to implement a queue using instances of stack data structure and operations on them.





A queue can be implemented using two stacks. Let queue to be implemented be q and stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

Method 1 (By making enQueue operation costly): This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

enQueue(q, x):

- ✓ *While stack1 is not empty, push everything from stack1 to stack2.*
- ✓ *Push x to stack1 (assuming size of stacks is unlimited).*
- ✓ *Push everything back to stack1.*

Here time complexity will be O(n)

deQueue(q):

- ✓ *If stack1 is empty then error*
- ✓ *Pop an item from stack1 and return it*

Here time complexity will be O(1)

Below is the implementation of the above approach:

```
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s1, s2;

    void enQueue(int x)
    {
        while (!s1.empty()) {
            s2.push(s1.top());
            s1.pop();
        }

        s1.push(x);

        while (!s2.empty()) {
            s1.push(s2.top());
            s2.pop();
        }
    }

    int deQueue()
    {
        if (s1.empty()) {
            cout << "Q is Empty";
            exit(0);
        }


        int x = s1.top();
        s1.pop();
        return x;
    }
};
```

```
int main()
{
    Queue q;
    q.enQueue(1);
    q.enQueue(2);
    q.enQueue(3);

    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';

    return 0;
}
```

Output:

1

2

3

Complexity Analysis:

- ✓ **Time Complexity:**
  - ✓ **Push operation:** O(N).
    In the worst case we have empty whole of stack 1 into stack 2.
  - ✓ **Pop operation:** O(1).
    Same as pop operation in stack.
- ✓ **Auxiliary Space:** O(N).
  Use of stack for storing values.

Method 2 (By making deQueue operation costly): In this method, in enqueue operation, the new element is entered at the top of stack1. In dequeue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enQueue(q, x)

1) Push x to stack1 (assuming size of stacks is unlimited).

Here time complexity will be O(1)

deQueue(q)

1) If both stacks are empty then error.

2) If stack2 is empty

While stack1 is not empty, push everything from stack1 to stack2.

3) Pop the element from stack2 and return it.

Here time complexity will be O(n)

Method 2 is definitely better than method 1.

Method 1 moves all the elements twice in enQueue operation, while method 2 (in deQueue operation) moves the elements once and moves elements only

if stack2 empty. So, the amortized complexity of the dequeue operation becomes

Implementation of method 2:

```
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s1, s2;

    void enQueue(int x)
    {
        s1.push(x);
    }


    int deQueue()
    {

        if (s1.empty() && s2.empty()) {
            cout << "Q is empty";
            exit(0);
        }


        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }


        int x = s2.top();
        s2.pop();
        return x;
    }
};

int main()
{
    Queue q;
    q.enQueue(1);
    q.enQueue(2);
    q.enQueue(3);

    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';

    return 0;
}
```

Output:
1 2 3

Complexity Analysis:

- ✓ **Time Complexity:**
  - ✓ **Push operation:** O(1).
    Same as pop operation in stack.
  - ✓ **Pop operation:** O(N) in general and O(1) amortized time complexity.
    In the worst case we have to empty the whole of stack 1 into stack 2 so its O(N).
- ✓ **Auxiliary Space:** O(N). Use of stack for storing values.

Queue can also be implemented using one user stack and one Function Call Stack.

Below is modified Method 2 where recursion (or Function Call Stack) is used to implement queue using only one user defined stack.

enQueue(x)

- ✓ Push *x* to *stack1*.

deQueue:

- ✓ 1) If *stack1* is empty then error.
- ✓ 2) If *stack1* has only one element then return it.
- ✓ 3) Recursively pop everything from the stack1, store the popped item in a variable *res*, push the *res* back to stack1 and return *res*
- ✓ The step 3 makes sure that the last popped item is always returned and since the recursion stops when there is only one item in *stack1* (step 2), we get the last element of *stack1* in deQueue() and all other items are pushed back in step

Implementation

```
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s;


    void enQueue(int x)
    {
        s.push(x);
    }


    int deQueue()
    {
        if (s.empty()) {
            cout << "Q is empty";
            exit(0);
        }
```

```
        int x = s.top();
        s.pop();


        if (s.empty())
            return x;


        int item = deQueue();

         s.push(x);

        return item;
    }
};

int main()
{
    Queue q;
    q.enQueue(1);
    q.enQueue(2);
    q.enQueue(3);

    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';
    cout << q.deQueue() << '\n';

    return 0;
}
```

Output: 1 2 3

Complexity Analysis:

- ✓ **Time Complexity:**
  - ✓ **Push operation :** O(1).
    Same as pop operation in stack.
  - ✓ **Pop operation :** O(N).
    The difference from above method is that in this method element is returned and all elements are restored back in a single call.
- ✓ **Auxiliary Space:** O(N).
  Use of stack for storing values.

Implement Stack using Queues

A stack can be implemented using two queues. Let stack to be implemented by 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

Method 1 (By making push operation costly):
This method ensures that the newly entered element is always at the front of 'q1' so that pop operation dequeues from 'q1'. 'q2' is used to put every new element in front of 'q1'.

- ✓ **push(s, x)** operation's steps are described below:
  - Enqueue x to q2
  - One by one dequeue everything from q1 and enqueue to q2.
  - Swap the names of q1 and q2
- ✓ **pop(s)** operation's function is described below:
  - Dequeue an item from q1 and return it.

Below is the implementation

```
#include <bits/stdc++.h>

using namespace std;

class Stack {

    queue<int> q1, q2;

public:
    void push(int x)
    {

        q2.push(x);

        while (!q1.empty()) {
            q2.push(q1.front());
            q1.pop();
        }


        queue<int> q = q1;
        q1 = q2;
        q2 = q;
    }

    void pop()
    {

        if (q1.empty())
            return;
        q1.pop();
    }

    int top()
    {
        if (q1.empty())
            return -1;
        return q1.front();
    }

    int size()
    {
        return q1.size();
    }
};
```

```
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);

    cout << "current size: " << s.size()
         << endl;
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;

    cout << "current size: " << s.size()
         << endl;
    return 0;}
```

Output

```
current size: 3

3

2

1

current size: 1
```

Complexity Analysis:

- **Time Complexity:**
  - **Push operation:** O(N), As all the elements need to be popped out from the Queue (q1) and push them back to Queue (q2).
  - **Pop operation:** O(1), As we need to remove the front element from the Queue.
- **Auxiliary Space:** O(N), As we use two queues for the implementation of a stack.

Method 2 (By making pop operation costly):
In a push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally, the last element is dequeued from q1 and returned.

1. **push(s, x)** operation:
   - Enqueue x to q1 (assuming size of q1 is unlimited).
2. **pop(s)** operation:
   - One by one dequeue everything except the last element from q1 and enqueue to q2.
   - Dequeue the last item of q1, the dequeued item is result, store it.
   - Swap the names of q1 and q2
   - Return the item stored in step 2.

Below is the implementation

```
#include <bits/stdc++.h>
using namespace std;
```

```
class Stack {
    queue<int> q1, q2;

public:
    void pop()
    {
        if (q1.empty())
            return;

        while (q1.size() != 1) {
            q2.push(q1.front());
            q1.pop();
        }

        q1.pop();

        queue<int> q = q1;
        q1 = q2;
        q2 = q;
    }

    void push(int x)
    {
        q1.push(x);
    }

    int top()
    {
        if (q1.empty())
            return -1;

        while (q1.size() != 1) {
            q2.push(q1.front());
            q1.pop();
        }


        int temp = q1.front();

        q1.pop();

        q2.push(temp);

        queue<int> q = q1;
        q1 = q2;
        q2 = q;
        return temp;
    }

    int size()
    {
        return q1.size();
    }
};
```

```
int main()
{
    Stack s;
    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);

    cout << "current size: " << s.size()
         << endl;
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    s.pop();
    cout << s.top() << endl;
    cout << "current size: " << s.size()
         << endl;
    return 0;
}
```

Output

```
current size: 4
4
3
2
current size: 2
```

Complexity Analysis:

- ✓ **Time Complexity:**
  - ✓ **Push operation:** O(1), As, on each push operation the new element is added at the end of the Queue.
  - ✓ **Pop operation:** O(N), As, on each pop operation, all the elements are popped out from the Queue (q1) except the last element and pushed into the Queue (q2).
- ✓ **Auxiliary Space:** O(N) since 2 queues are used.

Method 3 (Implement Stack using only 1 queue):
In this method, we will be using only one queue and make the queue act as a stack by using following steps:

- ✓ The idea behind this approach is to make one queue and push the first element in it.
- ✓ After the first element, we push the next element and then push the first element again and finally pop the first element.
- ✓ So, according to the FIFO rule of the queue, the second element that was inserted will be at the front and then the first element as it was pushed again later and its first copy was popped out.
- ✓ So, this acts as a stack and we do this at every step i.e. from the initial element to the second last element, and the last

element will be the one which we are inserting and since we will be pushing the initial elements after pushing the last element, our last element becomes the first element.

Below is the implementation

```
#include <bits/stdc++.h>
using namespace std;

class Stack {

    queue<int> q;

public:
    void push(int data);
    void pop();
    int top();
    bool empty();
};


void Stack::push(int data)
{
       int s = q.size();

    q.push(data);


    for (int i = 0; i < s; i++) {

        q.push(q.front());

        q.pop();
    }
}

void Stack::pop()
{
    if (q.empty())
        cout << "No elements\n";
    else
        q.pop();
}

int Stack::top() { return (q.empty()) ? -1 : q.front();
}

bool Stack::empty() { return (q.empty()); }

int main()
{
    Stack st;
    st.push(40);
    st.push(50);
    st.push(70);
    cout << st.top() << "\n";
```

```
    st.pop();
    cout << st.top() << "\n";
    st.pop();
    cout << st.top() << "\n";
    st.push(80);
    st.push(90);
    st.push(100);
    cout << st.top() << "\n";
    st.pop();
    cout << st.top() << "\n";
    st.pop();
    cout << st.top() << "\n";
    return 0;
}
```

Output

```
70

50

40

100

90

80
```

Complexity Analysis:

- ✓ **Time Complexity:**
  - **Push operation:** O(N)
  - **Pop operation:** O(1)
- ✓ **Space Complexity:** O(N)