

The background of the slide is a dense, colorful pattern of small circles or dots in various colors including green, blue, yellow, orange, red, and purple. A dark, semi-transparent rectangular overlay covers the left side of the image, containing the text. The text is white and centered within the overlay.

Algorithms

**Linear data structures –
Stacks & Queues**



Stack Data structure



Introduction

A stack is a linear data structure that follows the principle of **Last In First Out (LIFO)**. This means **the last element inserted inside the stack is removed first**.

You can think of the stack data structure as the pile of plates on top of another.





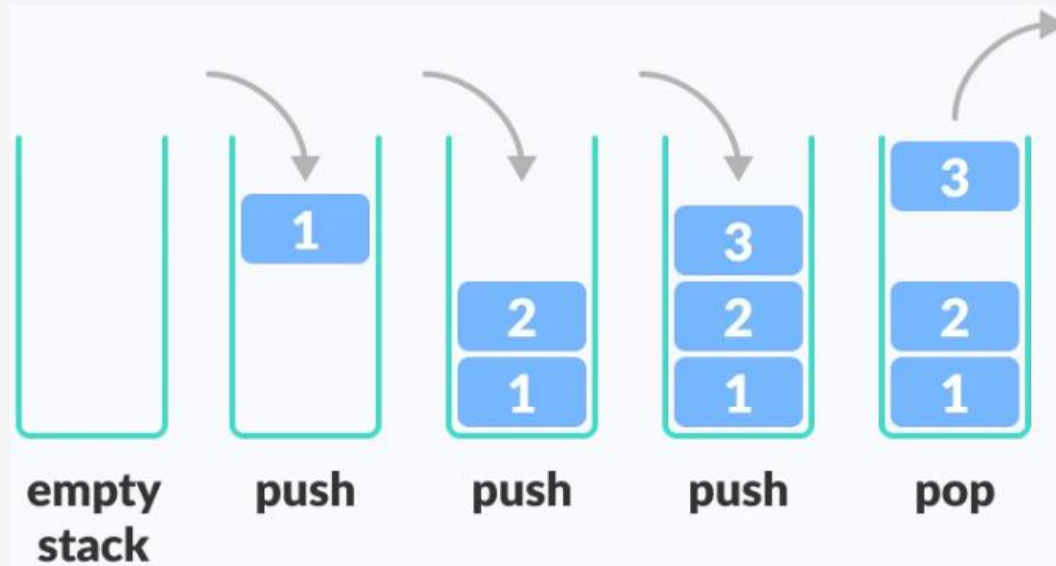
Here, you can:

- **Put a new plate on top**
- **Remove the top plate**

And, if you want the plate at the bottom, you must first remove all the plates on top. This is exactly how the stack data structure works.

LIFO Principle of Stack

In programming terms, putting an item on top of the stack is called **push** and removing an item is called **pop**.



In this example, although item 3 was kept last, it was removed first. This is exactly how the **LIFO** (Last In First Out) Principle works.

Basic operations of Stack

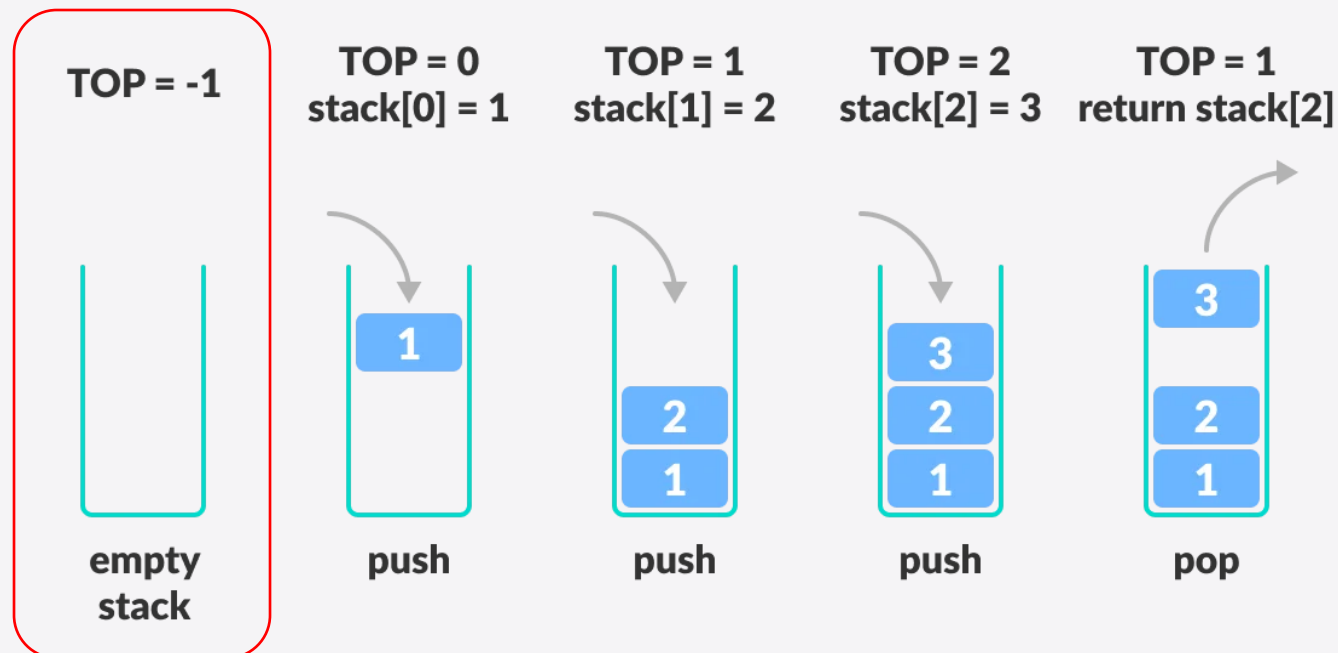
There are some basic operations that allow us to perform different actions on a stack.

- **Push:** Add an element to the top of a stack
- **Pop:** Remove an element from the top of a stack
- **IsEmpty:** Check if the stack is empty
- **IsFull:** Check if the stack is full
- **Peek:** Get the value of the top element without removing it

Working of Stack Data Structure

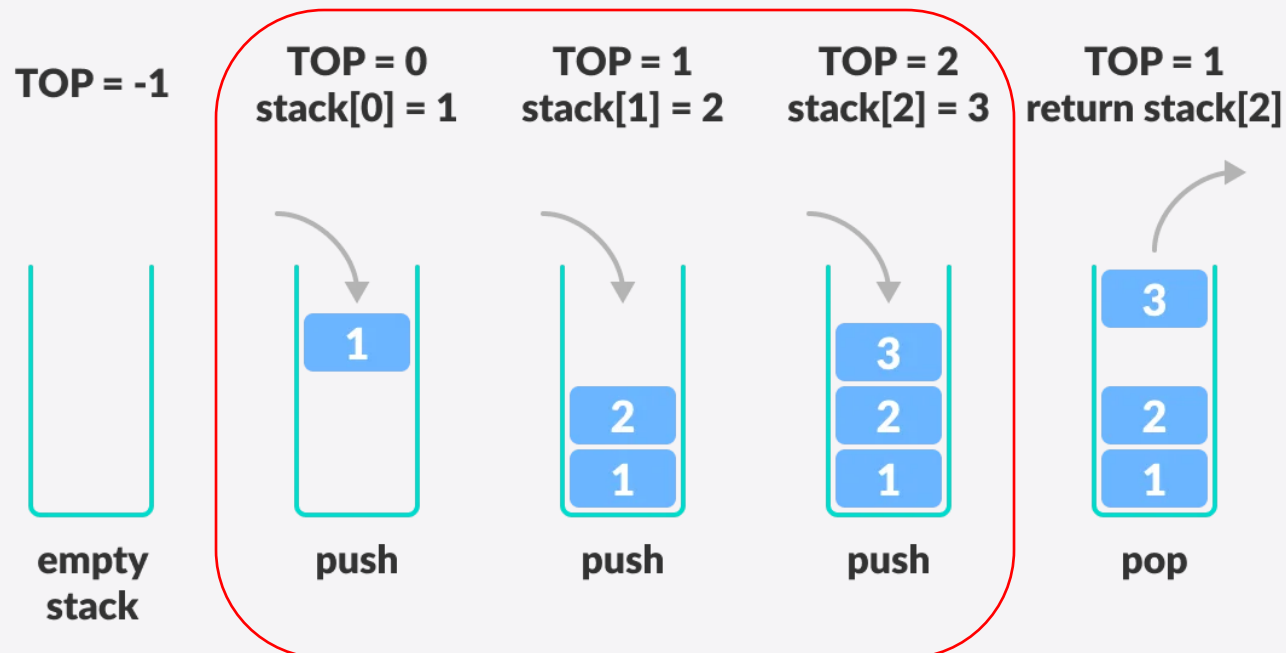
The operations work as follows:

- A pointer called **TOP** is used to keep track of the top element in the stack.
- When initializing the stack, we set its value to **-1** so that we can check if the stack is **empty** by comparing **TOP == -1**.



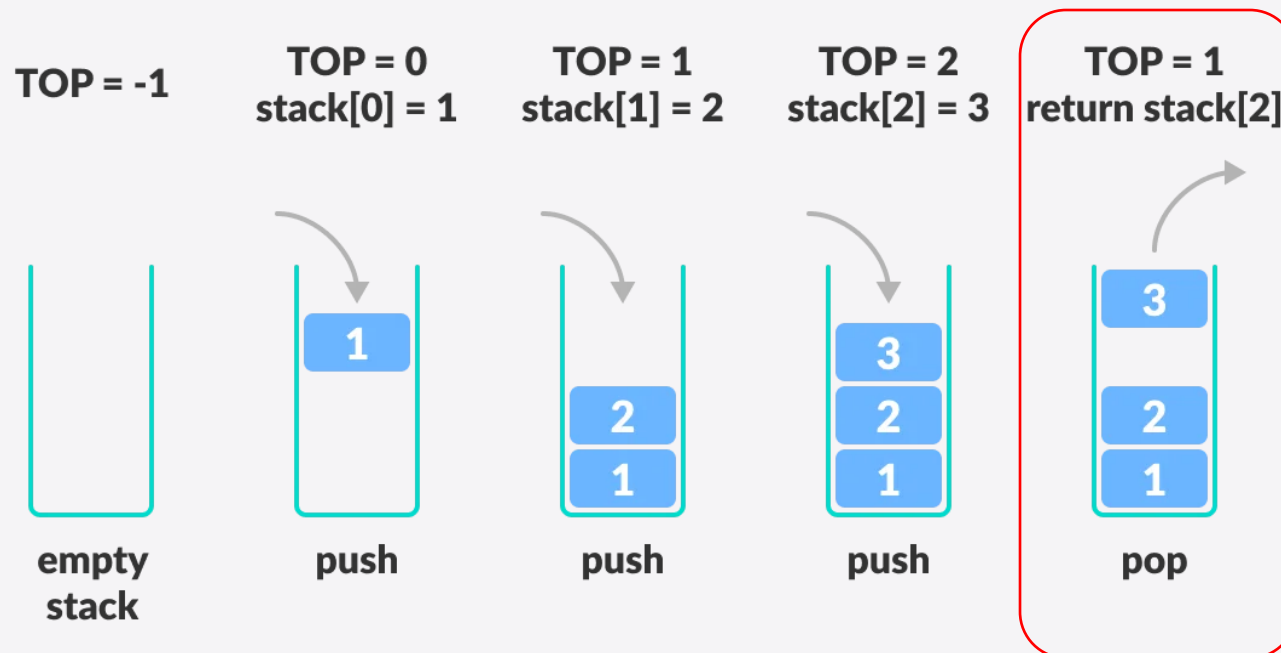
Working of Stack Data Structure

- On **pushing** an element, we **increase** the value of **TOP** and place the new element in the position pointed to by **TOP**.
- Before pushing, we **check if the stack is already full**.



Working of Stack Data Structure

- On **popping** an element, we return the element pointed to by **TOP** and **reduce** its value.
- Before popping, we **check if the stack is already empty**.



```

struct Node
{
    int num;
};

class Stack
{
private:
    int N, Top;
    Node *st;
public:
    Stack(int size)
    {
        Top = -1;
        N = size;

        st = new struct Node[N];
    }
    ~Stack()
    {
        cout << "Destructor: Erase Stack" << endl;
        delete[] st;
    }
    void Push(int);
    struct Node *Pop();
    int *Peek();
    bool IsEmpty();
};

```

Initialization

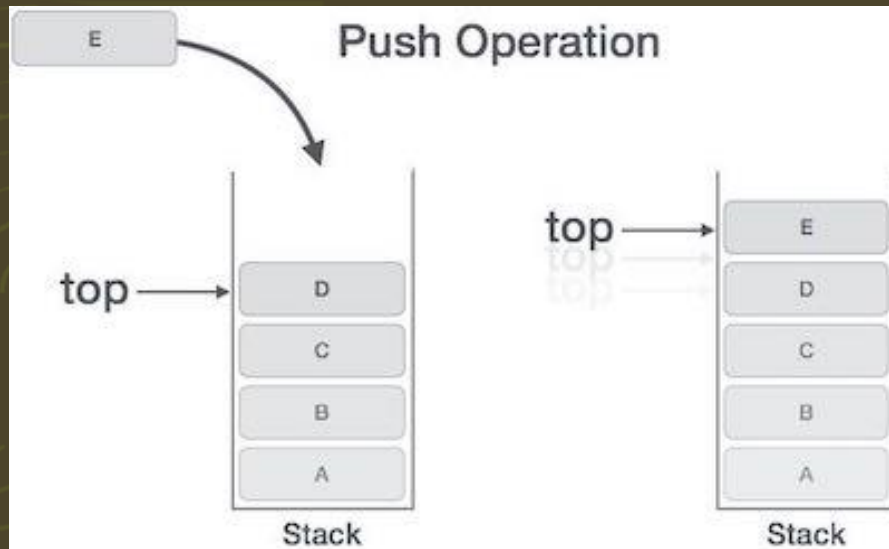
- We create a stack of elements of type **Node**. We start by initializing the variable **Top = -1** (in charge of keeping track of the top element) and we ask for the **stack size (N)**, the maximum number of Node elements to store.
- The pointer named as **st** stores the created array of nodes in dynamic memory.

```
void Stack::Push(int NUM)
{
    if(Top == N-1)
    {
        cout << "The stack is full (OverFlow)." << endl;
    }
    else
    {
        Top++;
        st[Top].num = NUM;
    }
}
```

Push

Push an element in a stack can be performed in the following ways,

- Make sure the stack is not full.
- If not, increase the value of Top and place the new element in the position pointed by Top.



What is the time complexity?

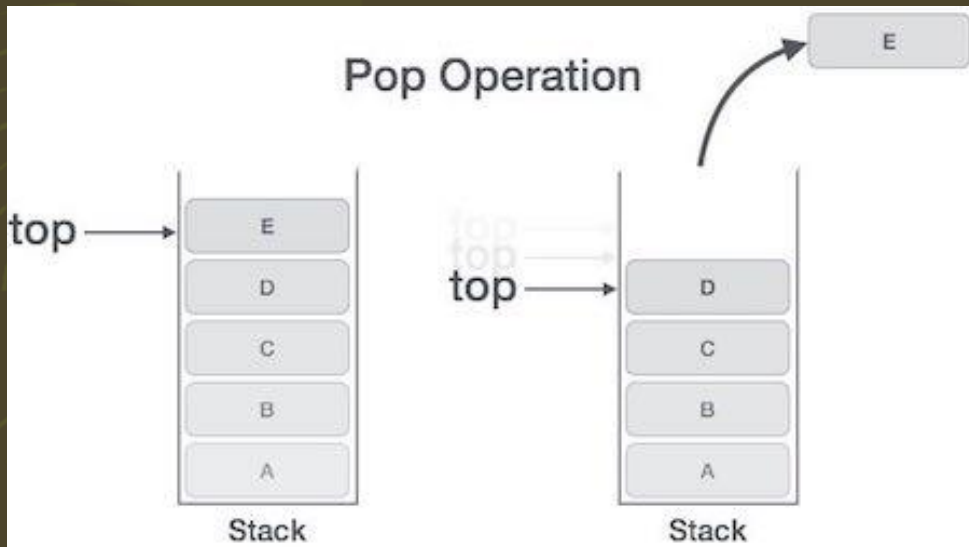
Pop

Pop an element of the stack can be performed in the following ways,

- Make sure the stack is not empty.
- If not, decrease the value of Top, but be sure to return the value on top of the stack.

```
struct Node *Stack::Pop(void)
{
    if(Top < 0)
    {
        cout << "The stack is empty (UnderFlow)." << endl;
        return NULL;
    }

    Top--;
    return &st[Top + 1];
}
```



What is the time complexity?

IsEmpty

The function checks whether the stack is empty or not.

```
bool Stack::IsEmpty()
{
    if(Top < 0)
        return true;
    return false;
}
```

- If the stack is empty (**Top = -1**), return **true**.
- Otherwise, return **false**.

What is the time complexity?

Peek

The function checks whether the stack is empty or not.

```
int *Stack::Peek(void)
{
    if(Top < 0)
        return NULL;
    return &(st[Top].num);
}
```

- If the stack is empty (**Top = -1**), return a **NULL** value.
- Otherwise, return the value at the top of the stack without pop the value of the stack.

What is the time complexity?

Program body

```
int main(void)
{
    int size;
    cout << "Insert size of the stack: ";
    cin >> size;

    Stack P(size);

    int x;
    for (int i = 0; i < size; i++)
    {
        cout << "Enter Value: ";
        cin >> x;
        P.Push(x);
    }
    P.Push(100);

    int *v;
    v = P.Peek();
    cout << "The value at top is: " << *v << endl;

    while(!P.IsEmpty())
    {
        cout << P.Pop()->num << " " << endl;
    }

    return (0);
}
```

Result:

```
Insert size of the stack: 3
Enter Value: 7
Enter Value: 96
Enter Value: 8
The stack is full (OverFlow).
The value at top is: 8
8
96
7
Destructor: Erase Stack
```

Applications of Stack Data Structure

Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

- **To reverse a word** - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.
- **In compilers** - Compilers use the stack to calculate the value of expressions like $2 + 4 / 5 * (7 - 9)$ by converting the expression to prefix or postfix form.
- **In browsers** - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.



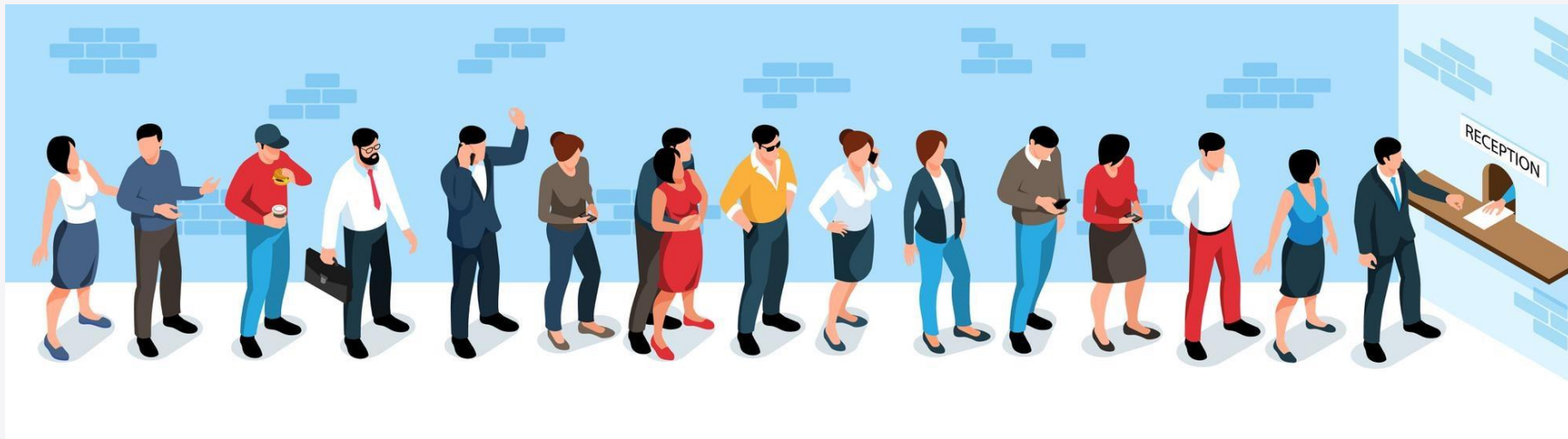
Queue Data structure



Introduction

A queue is a useful data structure in programming. It is like the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the **First In First Out (FIFO)** rule - **the item that goes in first is the item that comes out first.**



FIFO Principle of Queues



Here it is exemplified, since 1 was kept in the queue before 2, it is the first to be removed from the queue as well. It follows the **FIFO** rule.

In programming terms, putting items in the queue is called **enqueue**, and removing items from the queue is called **dequeue**.

Basic operations of Queue

A queue is an object (an abstract data structure - **ADT**) that allows the following operations:

- **Enqueue:** Add an element to the end of the queue.
- **Dequeue:** Remove an element from the front of the queue.
- **IsEmpty:** Check if the queue is empty.
- **IsFull:** Check if the queue is full.
- **Peek:** Get the value of the front of the queue without removing it.

Working of Queue Data Structure

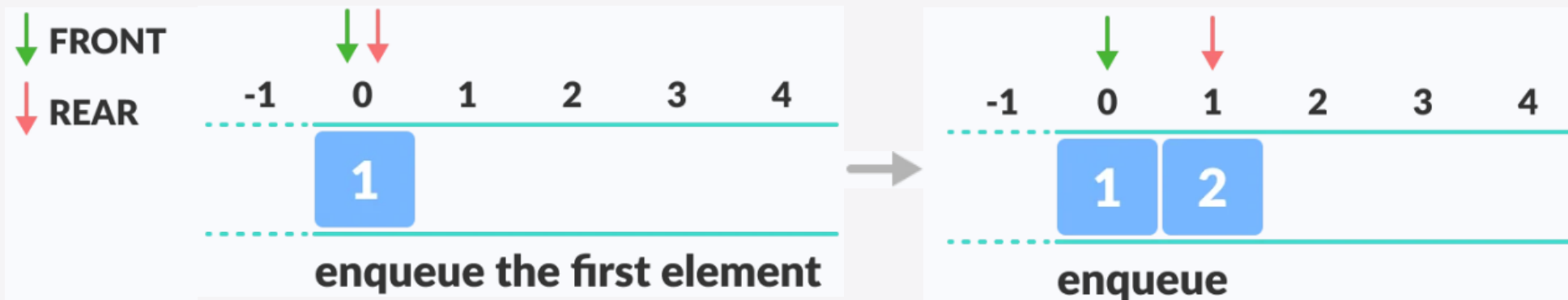
The operations work as follows:

- Two pointers **FRONT** and **REAR** are used to keep track of the elements in the Queue.
- When initializing the queue, we set their value to **-1** so that we can check if the queue is **full** by comparing **FRONT==0** and **REAR==N-1**.



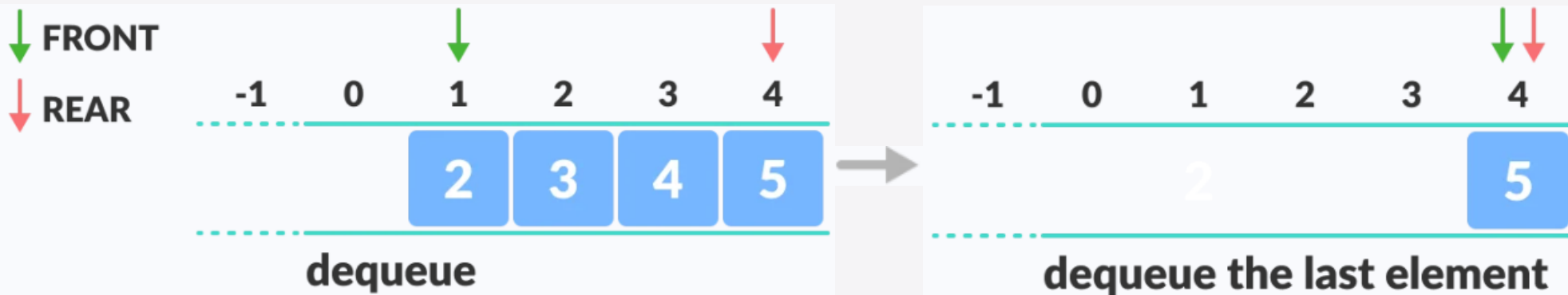
Working of Stack Data Structure

- On **Enqueue** an element, we **set** the value of **FRONT** to **0** and increase the **REAR** index by **1**, the new element is placed in the position pointed to by **REAR**.
- Before pushing, we **check if the queue is already full**.



Working of Stack Data Structure

- On **Dequeue** an element, we return the element pointed to by **FRONT** and **increase** its value by **1**. For the last element, we reset the values of **FRONT** and **REAR** to **-1**.
- Before popping, we **check if the queue is already empty**.



```

struct Node
{
    int value;
};

class Queue
{
private:
    Node *Q;
    int front,rear,N;
public:
    Queue(int size)
    {
        N = size;
        Q = new Node[size];
        front = -1;
        rear = -1;
    }
    ~Queue()
    {
        delete[] Q;
    }

    void Enqueue(int);
    Node *Dequeue();
    void Display();
    bool IsEmpty();
};

```

Initialization

- We create a queue of elements of type **Node**. We start by initializing the variables **front = -1** (keep track of the first element) and **rear = -1** (keep track of the last element) and we receive the **queue size** (N), if apply.
- The pointer named as **Q** stores the created array of nodes in dynamic memory.

Enqueue

Enqueue an element in a queue can be performed in the following ways,

- Make sure the queue is not full.
- For the first element, set the value of **front** to **0**.
- Increase the value of **rear** by **1**
- Add the new element in the position pointed to by **rear**.

What is the time complexity?

```
void Queue::Enqueue(int n)
{
    if(front == 0 && rear == N-1)
        cout << "Queue is full" << endl;
    else
    {
        if(front == -1)
            front = 0;

        rear++;

        Q[rear].value = n;
    }
}
```

```

Node *Queue::Dequeue()
{
    if(front == -1)
    {
        cout << "Queue is empty" << endl;
        return NULL;
    }

    Node *Aux = &Q[front];

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

    return Aux;
}

```

Dequeue

Dequeue an element of the queue can be performed in the following ways,

- Make sure the queue is not empty.
- Return the value pointed by **front**.
- increase the **front** index by **1**.
- for the last element, reset the values of **front** and **rear** to **-1**.

What is the time complexity?

IsEmpty

The function checks whether the queue is empty or not.

- If the queue is empty (**front = -1**), return **true**.
- Otherwise, return **false**.

What is the time complexity?

```
bool Queue::IsEmpty()
{
    if(front == -1)
        return true;
    return false;
}
```

```
void Queue::Display()
{
    if (IsEmpty())
    {
        cout << "Empty Queue" << endl;
    }
    else
    {
        cout << endl << "Front -> ";

        for (int i = front; i <= rear; i++)
            cout << Q[i].value << " ";

        cout << "<- Rear" << endl << endl;
    }
}
```

Display

The function displays all the element in the queue.

- Check If the queue is empty.
- If not, shows the values between the two pointers, **front** to **rear**.

What is the time complexity?

Program body

```
int main()
{
    Queue C(3);

    int x;
    for (int i = 0; i < 4; i++)
    {
        cout << "Enter Value: ";
        cin >> x;
        C.Enqueue(x);
    }

    C.Display();

    while(!C.IsEmpty())
    {
        cout << C.Dequeue()->value << " " << endl;
    }

    return 0;
}
```

Result:

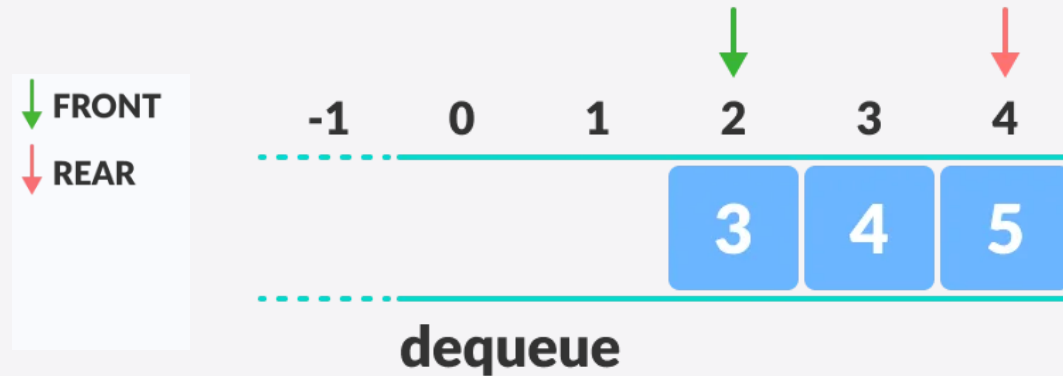
```
Enter Value: 5
Enter Value: 7
Enter Value: 2
Enter Value: 3
Queue is full
```

Front -> 5 7 2 <- Rear

```
5
7
2
```

Limitations of Queue

After a bit of enqueueing and dequeuing, the size of the queue has been reduced.



And **we can only add indexes 0 and 1 only when the queue is reset** (when all the elements have been dequeued).

After **rear** reaches the last index, if we can store extra elements in the empty spaces (0 and 1), we can make use of the empty spaces. This is implemented by a modified queue called the **circular queue**.

Applications of Queue Data Structure

- CPU scheduling, Disk Scheduling
- When data is transferred asynchronously between two processes. The queue is used for synchronization. For example: IO Buffers, pipes, file IO, etc
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people calling them in order.



Types of Queues

Introduction

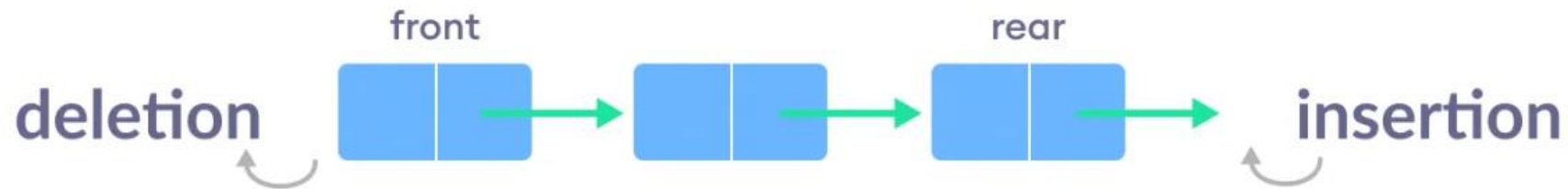
A **queue** is a useful data structure in programming. It is like the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

There are four different types of queues:

- **Simple Queue**
- **Circular Queue**
- **Priority Queue**
- **Double Ended Queue**

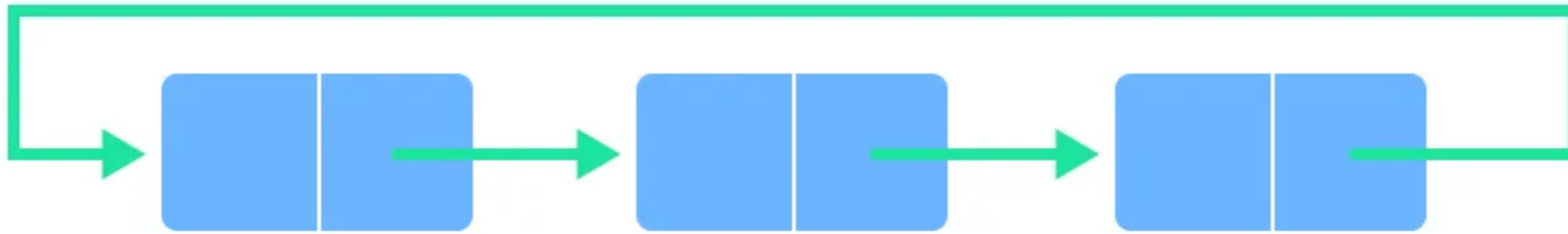
Simple Queue

In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the **FIFO** (First in First out) rule.



Circular Queue

In a circular queue, the **last element points to the first element** making a circular link.

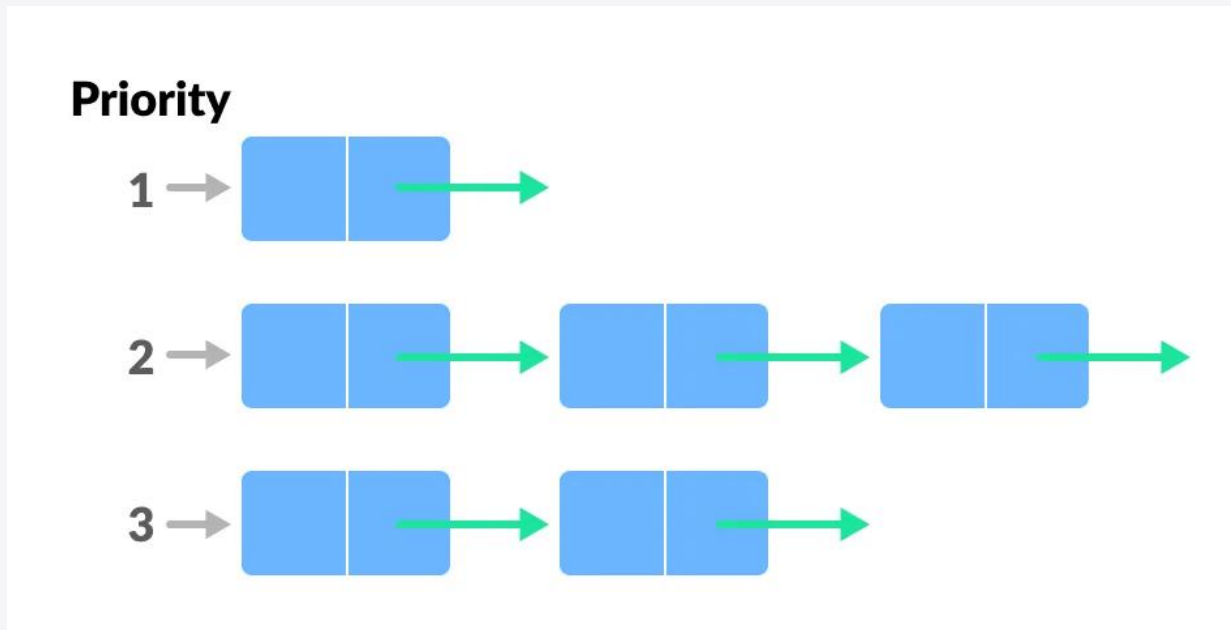


The main advantage of a circular queue over a simple queue is **better memory utilization**. If the last position is full and the first position is empty, we can insert an element in the first position. This action is not possible in a simple queue.

Priority Queue

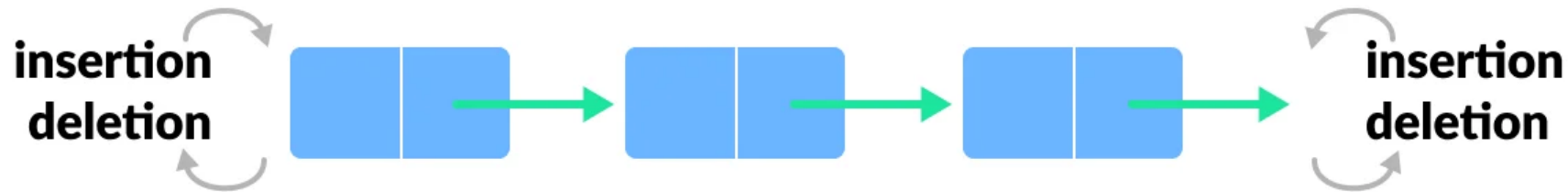
A priority queue is a special type of queue in which each element is associated with a **priority** and is served according to its priority. If elements with the **same priority occur**, they are served according to their order in the queue.

Insertion occurs based on the arrival of the values and **removal occurs based on priority**.



Deque (Double Ended Queue)

In a double ended queue, **insertion** and **removal** of elements can be performed from **either from the front or rear**. Thus, **it does not follow the FIFO** (First In First Out) rule.





Priority Queue

Data structure

Introduction

A priority queue is a special type of queue in which **each element is associated with a priority value**. And elements are served based on their **priority**. That is, higher priority elements are served first.

However, **if elements with the same priority occur, they are served according to their order in the queue**.

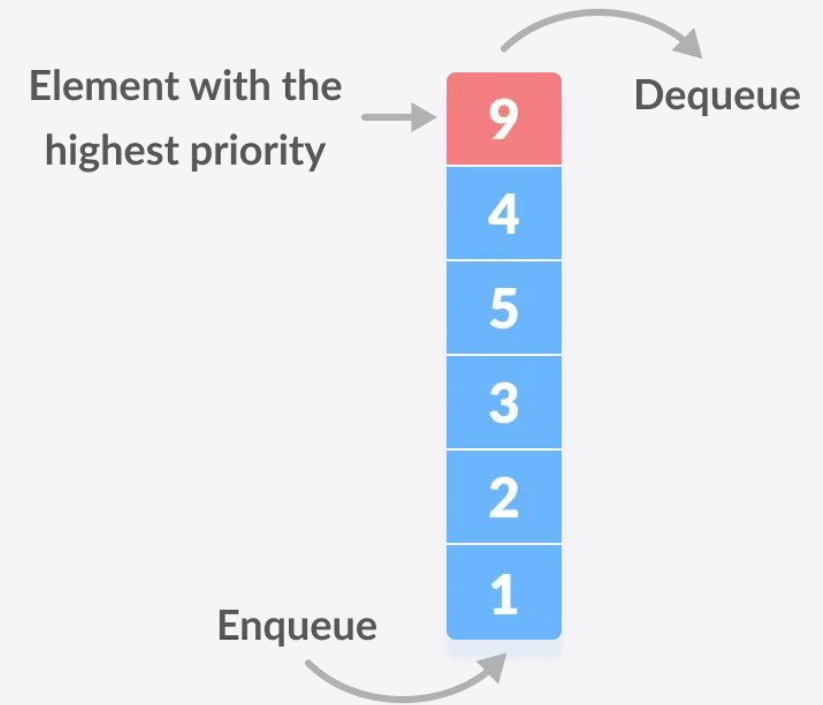


Assigning Priority Value

Generally, the value of the element itself is considered for assigning the priority. For example, depending on the case, the element with **the highest value is considered the highest priority element** or **vice versa**. We can also set priorities according to our needs.

- **Difference between Priority Queue and Normal Queue**

In a queue, the **first-in-first-out rule** is implemented whereas, in a **priority queue**, the values are removed **based on priority**. The element with the highest priority is removed first.



Implementation of Priority Queue

Priority queue can be implemented using an **array**, a **linked list**, a **heap data structure**, or a **binary search tree**.

Among these data structures, **heap data structure** provides an efficient implementation of priority queues.

Since we have not seen the more advanced data structures, we will use the **array** to implement the priority queue in this lesson.

Basic operations of Priority Queue

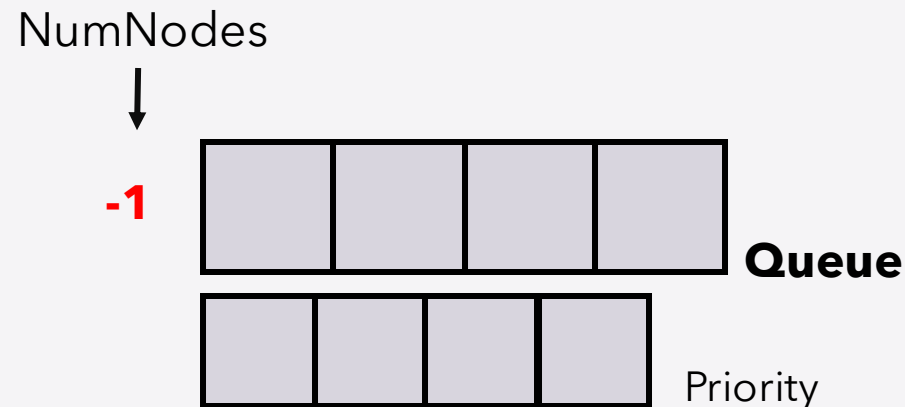
A priority queue is an object that allows the following operations:

- **Enqueue:** Add an element to the end of the queue.
- **Dequeue:** Remove the element with the highest priority (Here, **highest value = highest priority**) using **Peek()**.
- **Peek:** Get the **index** of the element with the highest priority, in case of a tie, the element with the largest **data** value is chosen.

Working of Priority Queue

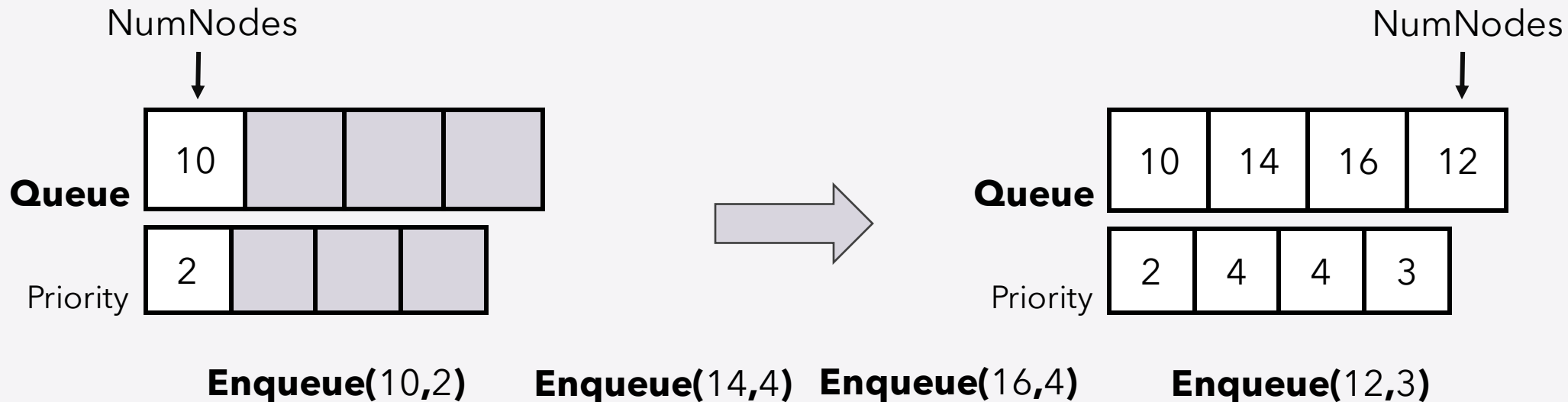
The operations work as follows:

- A pointers **NumNodes** is used to keep track of the elements in the Queue.
- When initializing the queue, we set the value to **-1** so that we can check if the queue is **full** by comparing **NumNodes == N-1**.



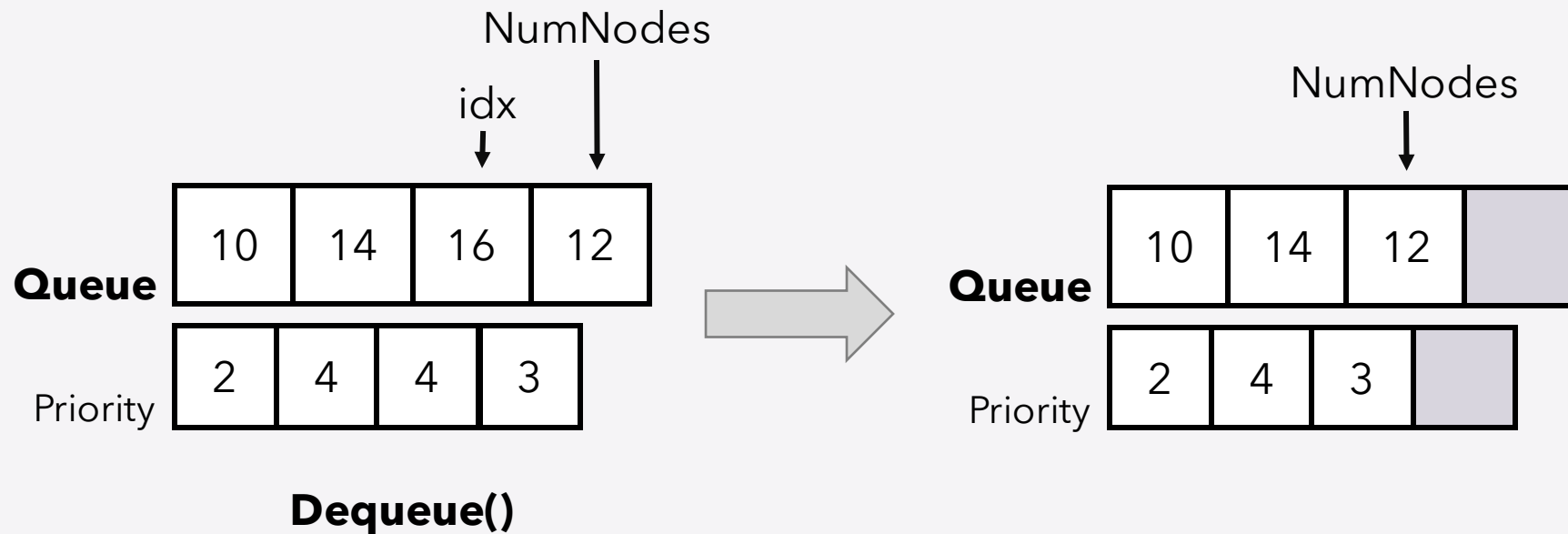
Working of Priority Queue

- Before enqueue, we **check if the queue is already full**.
- On **Enqueue** an element, we increase the value of **NumNodes** by **1**, the new element is place in the position pointed to by **NumNodes**.



Working of Priority Queue

- Before dequeue, we **check if the queue is already empty**.
- On **Dequeue** an element, we look for the element with more priority (**idx**) and save it. Rearrange the elements from **idx** to **NumNodes**. **Decrease** the **NumNodes** value by **1**. Return the saved element.



```

struct Node
{
    int value;
    int priority;
};

class PriorityQueue
{
private:
    Node *PQ;
    int N, NumNodes;
public:
    PriorityQueue(int size)
    {
        N = size;
        PQ = new Node[size];
        NumNodes = -1;
    }
    ~PriorityQueue()
    {
        delete[] PQ;
    }

    void Enqueue(int, int);
    Node *Dequeue();
    int Peek();
    void Display(int);
    void Display();
};

```

Initialization

- We create a queue of elements of type **Node**. We start by initializing the pointer **NumNodes = -1** (keep track of the elements) and we receive the **queue size** (N), if apply.
- The pointer named as **PQ** stores the created array of nodes in dynamic memory.

Enqueue

Enqueue an element in a queue can be performed in the following ways,

- Make sure the queue is not full.
- Increase the value of **NumNodes** by **1**
- Add the new element in the position pointed to by **NumNodes**.

```
void PriorQueue::Enqueue(int value, int priority)
{
    if(NumNodes == N-1)
    {
        cout << "Priority Queue is full" << endl;
        return;
    }
    // Increase the size
    NumNodes++;

    // Insert the element
    PQ[NumNodes].value = value;
    PQ[NumNodes].priority = priority;
}
```

What is the time complexity?

```

Node *PriorityQueue::Dequeue()
{
    if(NumNodes == -1)
    {
        cout << "Priority Queue is empty" << endl;
        return NULL;
    }

    int ind = Peek();

    Node *Aux = &PQ[ind];

    for (int i = ind; i < NumNodes; i++)
    {
        PQ[i] = PQ[i + 1];
    }

    NumNodes--;

    return Aux;
}

```

Dequeue

Dequeue an element of the queue can be performed in the following ways,

- Make sure the queue is not empty.
- Find the position of the element with the highest priority.
- Shift the position of all elements after the position of the searched element one index before.
- Decrease the value of **NumNodes** by **1**

What is the time complexity?


```

int PriorityQueue::Peek()
{
    int highestPriority = INT_MIN;
    int ind = -1;

    for (int i = 0; i <= NumNodes; i++)
    {
        if (highestPriority < PQ[i].priority)
        {
            highestPriority = PQ[i].priority;
            ind = i;
        }
        else if(highestPriority == PQ[i].priority
        && PQ[i].value > PQ[ind].value)
        {
            highestPriority = PQ[i].priority;
            ind = i;
        }
    }

    return ind;
}

```

Peek

The function finds the position of the element with the highest priority.

- If priority is same, choose the element with the highest value.
- Return position of the element (**ind**).

What is the time complexity?

Display (Bonus)

```
void PriorityQueue::Display(int idx)
{
    cout << PQ[idx].value << endl << endl;
}

void PriorityQueue::Display()
{
    cout << "Elements: ";

    for (int i=0; i <= NumNodes; i++)
    {
        cout << PQ[i].value << " ";
    }
    cout << endl << endl;
}
```

- One function displays all the elements in the priority queue.
- The other print only the element in the **idx** index.

Program body

```
int main()
{
    PriorityQueue P(5);

    P.Enqueue(10, 2);
    P.Enqueue(14, 4);
    P.Enqueue(16, 4);
    P.Enqueue(12, 3);
    P.Display();

    int indx = P.Peek();
    cout << "Higher priority element: ";
    P.Display(indx);

    P.Dequeue();
    P.Display();

    indx = P.Peek();
    cout << "Higher priority element: ";
    P.Display(indx);

    P.Dequeue();
    P.Display();

    indx = P.Peek();
    cout << "Higher priority element: ";
    P.Display(indx);

    P.Dequeue();
    P.Display();

    P.Dequeue();
    P.Display();

    P.Dequeue();

    return 0;
}
```

Result:

Elements: 10 14 16 12

Higher priority element: 16

Elements: 10 14 12

Higher priority element: 14

Elements: 10 12

Higher priority element: 12

Elements: 10

Elements:

Priority Queue is empty

Implementations of priority Queue

A comparative analysis of different implementations of priority queue is given below.

Operations	peek	insert	delete
Linked List	$O(1)$	$O(n)$	$O(1)$
Binary Heap	$O(1)$	$O(\log n)$	$O(\log n)$
Binary Search Tree	$O(1)$	$O(\log n)$	$O(\log n)$

Applications of Priority Queue Data Structure

Some applications may not apply with this array implementation. In general, some of the applications of a priority queue are:

- Dijkstra's algorithm
- for implementing stack
- for load balancing and interrupt handling in an operating system
- for data compression in Huffman code



Deque Data structure



Introduction

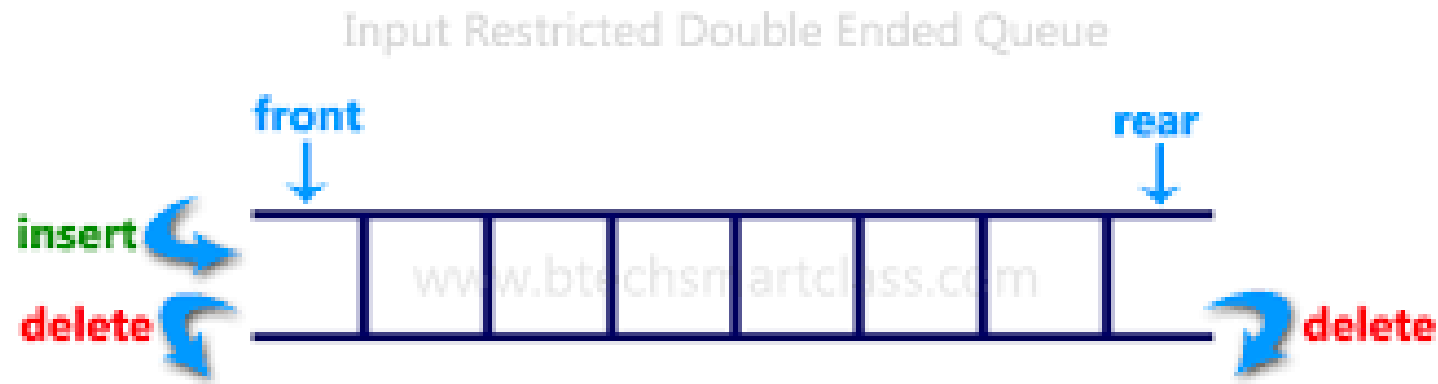
Deque or **Double Ended Queue** is a type of queue in which insertion and removal of elements can either be performed from the **front** or the **rear**. Thus, **it does not follow FIFO rule** (First In First Out).



Types of Deque (1/2)

- **Input Restricted Deque**

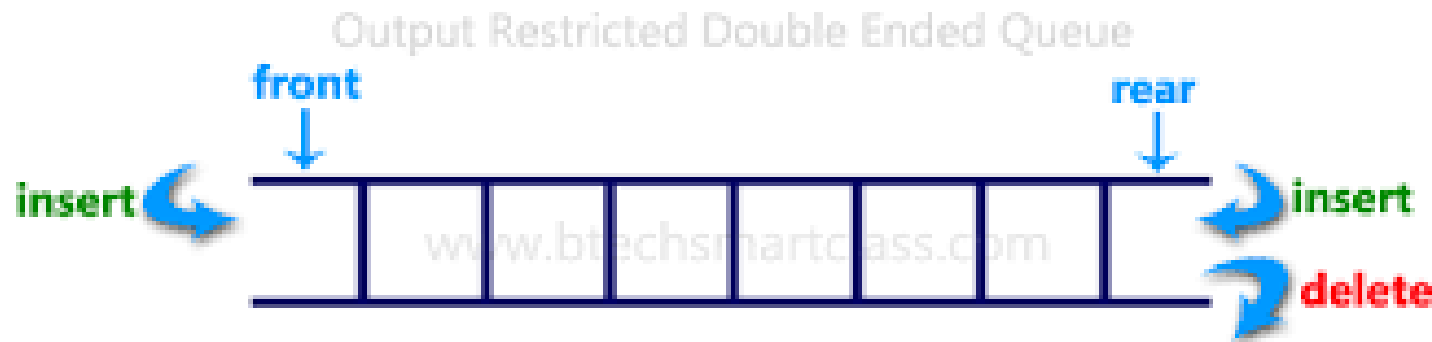
In this deque, **insert** is restricted at a **single end** but allows **deletion** at **both the ends**.



Types of Deque (2/2)

- **Output Restricted Deque**

In this deque, **delete** is restricted at a **single end** but allows **insertion** at **both the ends**.



Basic operations of Deque

A Deque is an object (an abstract data structure - **ADT**) that allows the following operations:

- **Insert at the Front:** Add an element at the front.
- **Insert at the Rear:** Add an element at the rear.
- **Delete from the Front:** Delete an element from the front.
- **Delete from the Rear:** Delete an element from the rear.

Working of Deque Data Structure

The operations work as follows:

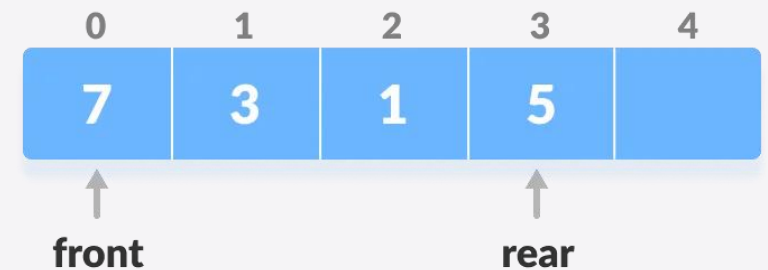
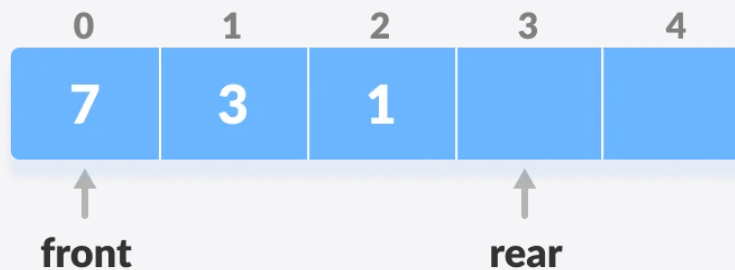
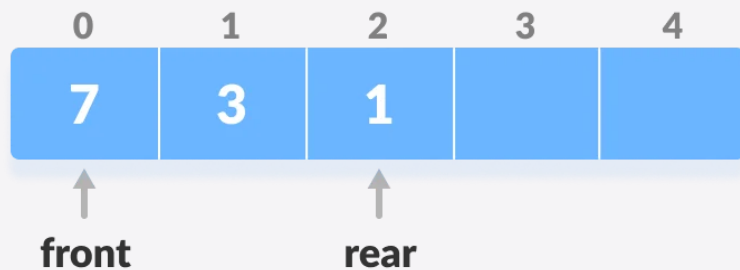
- Take an array (deque) of size **N**.
- Set two pointers at the first position and set **FRONT = -1** and **REAR = -1**.





Working of Deque Data structure

- Before **insert at the rear** (the standard operation in a queue), we **check if the queue is already full**.
- We increase the value of **rear** by **1**, the new element is place in the position pointed to by **rear**.
- If it is the first element, put **front = 0**.

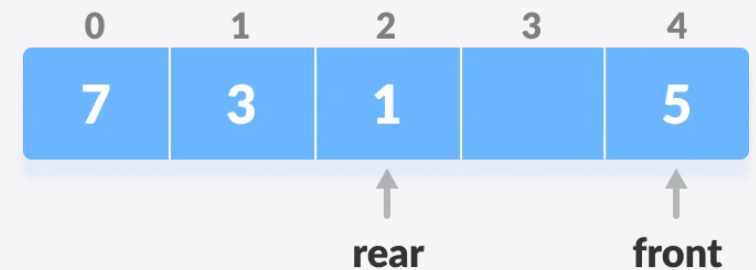
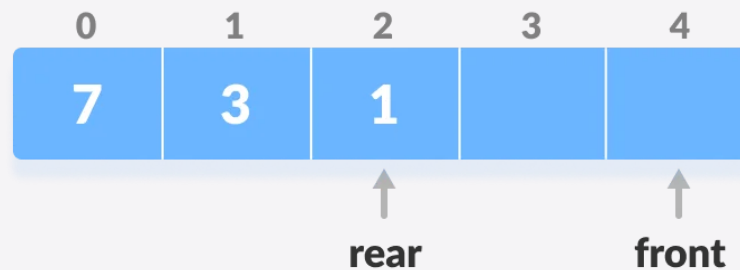
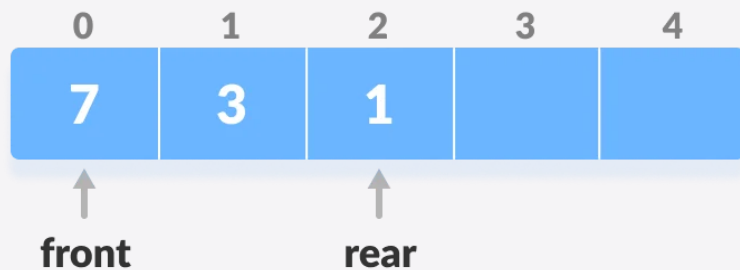


Push_back(5)



Working of Deque Data structure

- Before **insert at the front**, we **check if the queue is already full**.
- If **front < 1**, reinitialize **front = N-1** (last index). Else, decrease **front** by 1.
- Add the new element into **array[front]**.

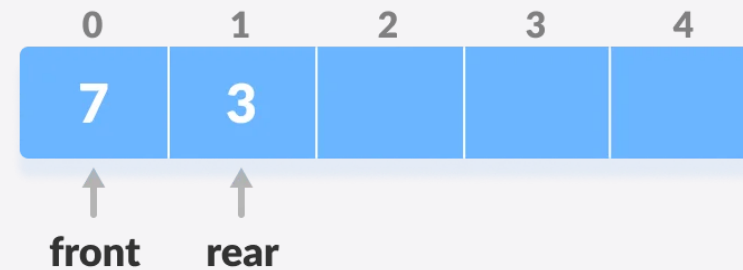
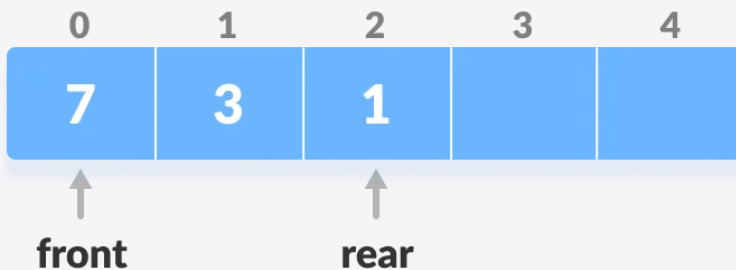


Push_front(5)



Working of Deque Data structure

- Before **delete from the rear** (the standard operation in a stack), we **check if the queue is empty**. If it is empty (**front = -1**), deletion cannot be done.
- Save the element to return, **array[rear]**.
- If the deque has only one element (**front = rear**), set **front = -1** and set **rear = -1**.
- Else if **rear** is at the front (**rear = 0**), set **rear = N-1**.
- Else, **rear = rear - 1**.

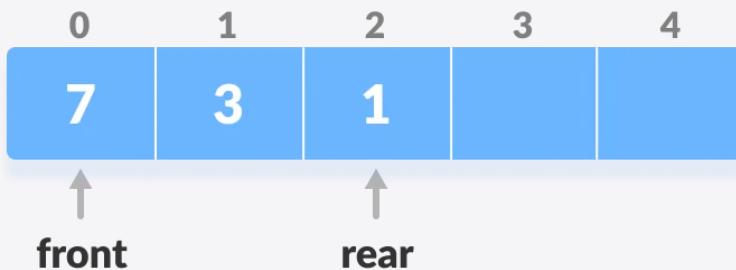


Pop_rear()



Working of Deque Data structure

- Before **delete from the front** (standard operation in a queue), we **check if the queue is empty**. If it is empty (**front = -1**), deletion cannot be done.
- Save the element to return, **array[front]**.
- If the deque has only one element (**front = rear**), set **front = -1** and set **rear = -1**.
- Else if **front** is at the end (**front = N-1**), set **front = 0**.
- Else, **front = front + 1**.



Pop_front()

Working of Deque Data structure

- The **deque is full** if **front = 0** and **rear = N-1** OR **front = rear + 1**.
- The **deque is empty** if **front = -1** (which also imply that **rear = -1**).


```

struct Node
{
    int data;
};

class Deque
{
private:
    Node *D;
    int N;
    int front, rear;
public:
    Deque(int size)
    {
        N = size;
        D = new Node[size];
        front = -1;
        rear = -1;
    }
    ~Deque()
    {
        delete[] D;
    }

    void Push_front(int);
    void Push_back(int);
    Node *Pop_front();
    Node *Pop_back();
    void Display();
};

```

Initialization

- We create a queue of elements of type **Node**. We start by initializing the pointers **front = -1** and **rear = -1** (keep track of the elements) and we receive the **deque size** (N), if apply.
- The pointer named as **D** stores the created array of nodes in dynamic memory.

Insert at the rear

The operation of insert an element at the rear can be performed in the following ways,

- Make sure the queue is not full.
- If it is the first element, set **front** at 0.
- Increase **rear** by 1.
- Add the new element in the position pointed to by **rear**.

```
void Deque::Push_back(int value)
{
    if( (front==0 && rear==N-1) || front==rear+1 )
    {
        cout << "The Deque is full" << endl;
        return;
    }

    if(front == -1) // first element
        front = 0;

    rear++;

    D[rear].data = value;
}
```

What is the time complexity?

Insert at the front

The operation of insert an element at the front can be performed in the following ways,

- Make sure the queue is not full.
- If **front** is at the beginning, set **front** at the end. Else, decrease **front** by 1.
- Add the new element in the position pointed to by **front**.

```
void Deque::Push_front(int value)
{
    if( (front==0 && rear==N-1) || front==rear+1 )
    {
        cout << "The Deque is full" << endl;
        return;
    }

    if (front < 1)
        front = N-1;
    else
        front--;

    D[front].data = value;
}
```

What is the time complexity?

```

Node *Deque::Pop_back()
{
    if(front == -1)
    {
        cout << "The Deque is empty" << endl;
        return NULL;
    }

    Node* Aux = &D[rear];

    if(front == rear) //only one element
    {
        front = -1;
        rear = -1;
    }
    else if(rear == 0) //it's at the front (rear < front)
    {
        rear = N-1;
    }
    else //front < rear
    {
        rear--;
    }

    return Aux;
}

```

Pop from the rear

The operation of pop from the rear can be performed in the following ways,

- Make sure the queue is not empty.
- Save the element to return.
- Redefine the position of **rear** depending on the position of the popped element (only element, front, second to last).
- Return the element.

What is the time complexity?

```

Node *Deque::Pop_front()
{
    if(front == -1)
    {
        cout << "The Deque is empty" << endl;
        return NULL;
    }

    Node* Aux = &D[front];

    if(front == rear) //only one element
    {
        front = -1;
        rear = -1;
    }
    else if(front == N-1) //front is at the end
    {
        front = 0;
    }
    else //0 <= front < rear
    {
        front++;
    }

    return Aux;
}

```

Pop from the front

The operation of pop from the front can be performed in the following ways,

- Make sure the queue is not empty.
- Save the element to return.
- Redefine the position of **front** depending on the position of the popped element (only element, end, first to last-1).
- Return the element.

What is the time complexity?

```

void Deque::Display()
{
    if(front == -1)
        cout << "The Deque is empty" << endl;

    cout << "Elements: ";

    int end = (front > rear)? (N-1):rear;
    for (int i = front; i <= end; i++)
    {
        cout << D[i].data << " ";
    }


    int ini = (front > rear)? 0:front;
    for (int i = ini; i <= rear; i++)
    {
        cout << D[i].data << " ";
    }
    cout << endl;
}

```

Display (Bonus)

- First, display all the elements from the pointer **front** to the end of the deque.
- Second, display all the elements from the beginning of the deque to the pointer **rear**.

Regardless of whether the pointer is at the beginning of the deque or on the other side, the result would be the same. If **front** \leq **rear**, only the first '**for**' loop will be executed; otherwise, both '**for**' loops will be executed.



Program body

```
int main()
{
    Deque DQ(4);

    DQ.Push_back(8);
    DQ.Push_back(5);
    DQ.Push_front(7);
    DQ.Push_front(10);
    DQ.Display();

    DQ.Push_back(11);


    DQ.Pop_back();
    DQ.Pop_front();
    DQ.Display();

    DQ.Push_front(55);
    DQ.Push_back(45);
    DQ.Display();

    return 0;
}
```

Result:

Elements: 10 7 8 5
The Deque is full
Elements: 7 8
Elements: 55 7 8 45



Applications of Deque Data Structure

Some of the applications of a priority queue are:

- In undo operations on software.
- To store history in browsers.
- For implementing both **stacks** and **queues**.

