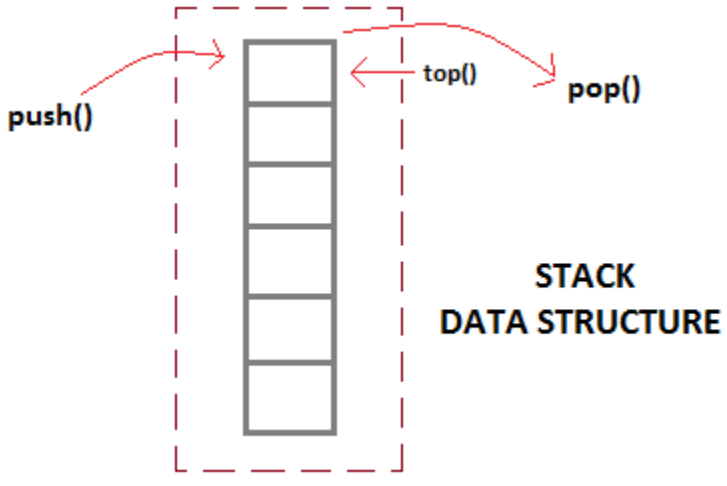
What is Stack Data Structure?

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



Basic features of Stack

1. Stack is an **ordered list** of **similar data type**.
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. `push()` function is used to insert new elements into the Stack and `pop()` function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Applications of Stack

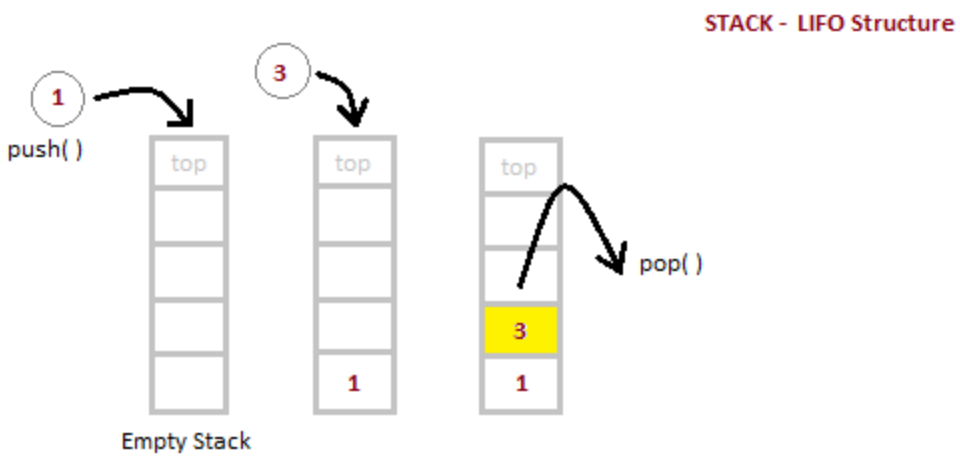
The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like:

1. Parsing
2. Expression Conversion(Infix to Postfix, Postfix to Prefix etc)

Implementation of Stack Data Structure

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



In a Stack, all operations take place at the "**top**" of the stack. The "**push**" operation adds an item to the top of the Stack.
The "**pop**" operation removes the item on top of the stack.

Algorithm for PUSH operation

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element.

Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.

Below we have a simple C++ program implementing stack data structure while following the object oriented programming concepts.

If you are not familiar with C++ programming concepts, you can learn it form here.

```
/*  Below program is written in C++ language  */


# include<iostream>


using namespace std;


class Stack
{
    int top;
    public:
    int a[10];  //Maximum size of Stack
    Stack()
```

```
    {
        top = -1;
    }

    // declaring all the function
    void push(int x);
    int pop();
    void isEmpty();
};

// function to insert data into stack
void Stack::push(int x)
{
    if(top >= 10)
    {
        cout << "Stack Overflow \n";
    }
    else
    {
        a[++top] = x;
        cout << "Element Inserted \n";
    }
}

// function to remove data from the top of the stack
int Stack::pop()
{
    if(top < 0)
    {
        cout << "Stack Underflow \n";
```

```
        return 0;

    }

    else

    {

        int d = a[top--];

        return d;

    }

}


// function to check if stack is empty

void Stack::isEmpty()

{

    if(top < 0)

    {

        cout << "Stack is empty \n";

    }

    else

    {

        cout << "Stack is not empty \n";

    }

}


// main function

int main() {


    Stack s1;

    s1.push(10);

    s1.push(100);

    /*

        preform whatever operation you want on the stack
```

```
    */

}
```

Copy

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Analysis of Stack Operations

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** : O(1)
- **Pop Operation** : O(1)
- **Top Operation** : O(1)
- **Search Operation** : O(n)

The time complexities for `push()` and `pop()` functions are `O(1)` because we always have to insert or remove the data from the **top** of the stack, which is a one step process.

Now that we have learned about the Stack in Data Structure, you can also check out these topics:

- Queue Data Structure
- Queue using stack

What is a Queue Data Structure?

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**).

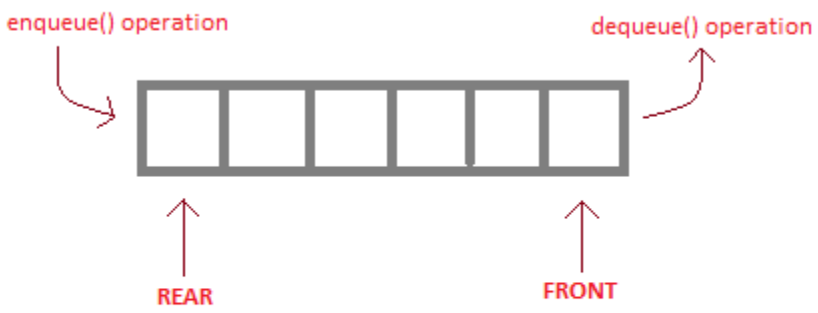
Before you continue reading about queue data structure, check these topics before to understand it clearly:

- Data Structures and Algorithms
- Stack Data Structure

This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.

Which is exactly how queue system works in real world. If you go to a ticket counter to buy movie tickets, and are first in the queue, then you will be the first one to get the tickets. Right? Same is the case with Queue data structure. Data inserted first, will leave the queue first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like stack, queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. `peek( )` function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

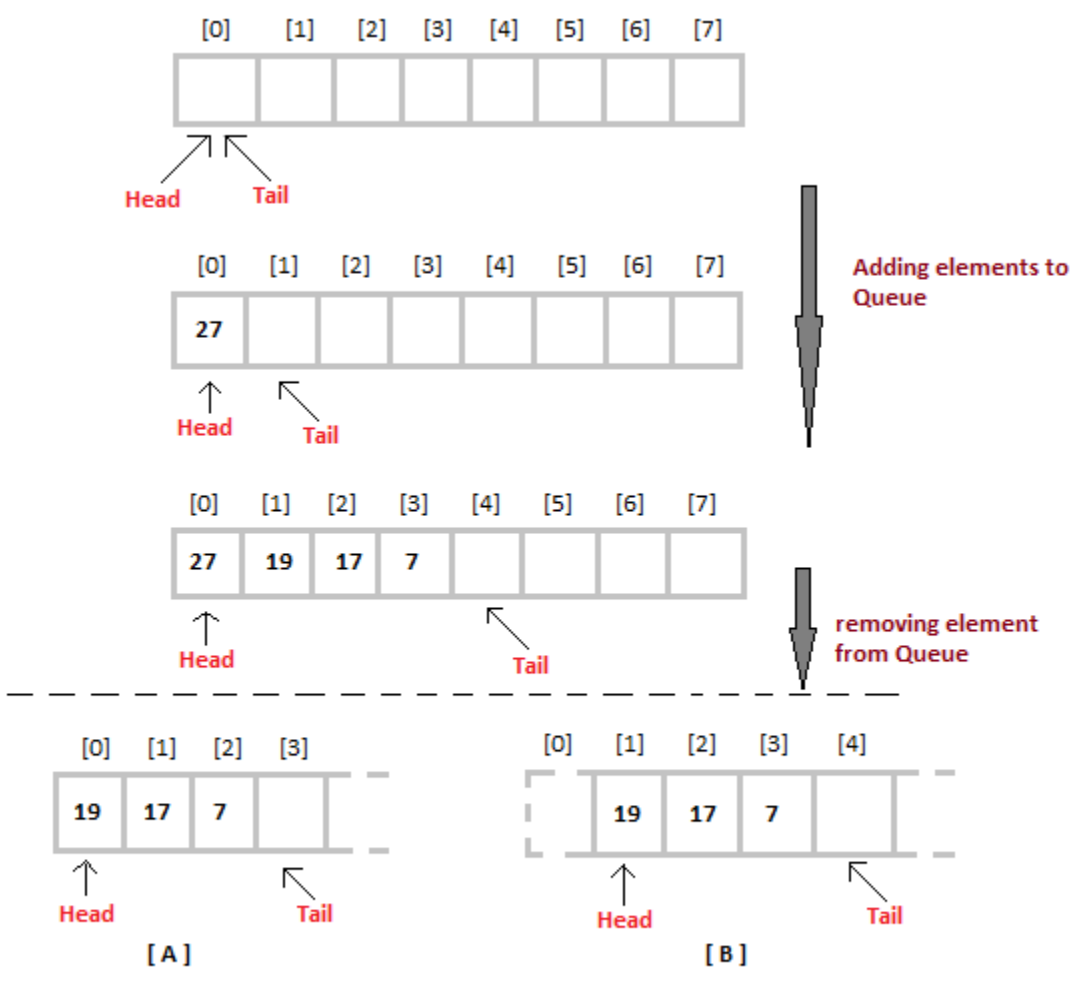
1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the **tail** keeps on moving ahead, always pointing to the position where the next element will be inserted, while the **head** remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an **overhead of shifting the elements one position forward** every time we remove the first element.

In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the **size on Queue is reduced by one space** each time.

Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then print overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then print underflow error and exit the program.
3. If the queue is not empty, then print the element at the head and increment the head.

```
/* Below program is written in C++ language */

#include<iostream>

using namespace std;

#define SIZE 10

class Queue
{
    int a[SIZE];
    int rear;   //same as tail
    int front;  //same as head

public:
Queue()
{
    rear = front = -1;
}

//declaring enqueue, dequeue and display functions
void enqueue(int x);
int dequeue();
```

```
    void display();
};


// function enqueue - to add data to queue
void Queue :: enqueue(int x)
{
    if(front == -1) {
        front++;
    }
    if( rear == SIZE-1)
    {
        cout << "Queue is full";
    }
    else
    {
        a[++rear] = x;
    }
}


// function dequeue - to remove data from queue
int Queue :: dequeue()
{
    return a[++front];  // following approach [B], explained above
}


// function to display the queue elements
void Queue :: display()
{
    int i;
    for( i = front; i <= rear; i++)
```

```
    {
        cout << a[i] << endl;
    }
}


// the main function
int main()
{
    Queue q;
    q.enqueue(10);
    q.enqueue(100);
    q.enqueue(1000);
    q.enqueue(1001);
    q.enqueue(1002);
    q.dequeue();
    q.enqueue(1003);
    q.dequeue();
    q.dequeue();
    q.enqueue(1004);


    q.display();



    return 0;
}
```

Copy

To implement approach [A], you simply need to change the `dequeue` method, and include a `for` loop which will shift all the remaining elements by one position.

```
return a[0];    //returning first element
for (i = 0; i < tail-1; i++)    //shifting all other elements
{
```

```
    a[i] = a[i+1];

    tail--;

}
```

Copy

Complexity Analysis of Queue Operations

Just like Stack, in case of a Queue too, we know exactly, on which position new element will be added and from where an element will be removed, hence both these operations requires a single step.

- Enqueue: **O(1)**
- Dequeue: **O(1)**
- Size: **O(1)**

Implement Queue using Stacks

A Queue is defined by its property of **FIFO**, which means First in First Out, i.e the element which is added first is taken out first. This behaviour defines a queue, whereas data is actually stored in an **array** or a **list** in the background.

What we mean here is that no matter how and where the data is getting stored, if the first element added is the first element being removed and we have implementation of the functions `enqueue()` and `dequeue()` to enable this behaviour, we can say that we have implemented a Queue data structure.

In our previous tutorial, we used a simple **array** to store the data elements, but in this tutorial we will be using **Stack data structure** for storing the data.

While implementing a queue data structure using stacks, we will have to consider the natural behaviour of stack too, which is **First in Last Out**.

For performing **enqueue** we require only **one stack** as we can directly **push** data onto the stack, but to perform **dequeue** we will require **two Stacks**, because we need to follow queue's FIFO property and if we directly **pop** any data element out of Stack, it will follow LIFO approach(Last in First Out).

Implementation of Queue using Stacks

In all we will require two Stacks to implement a queue, we will call them `S1` and `S2`.

```
class Queue {

    public:

    Stack S1, S2;



    //declaring enqueue method

    void enqueue(int x);



    //declaring dequeue method

    int dequeue();

}
```

Copy

In the code above, we have simply defined a class `Queue`, with two variables `S1` and `S2` of type `Stack`.

We know that, Stack is a data structure, in which data can be added using `push()` method and data can be removed using `pop()` method.

You can find the code for `Stack` class in the Stack data structure tutorial.

To implement a queue, we can follow two approaches:

1. **By making the `enqueue` operation costly**
2. **By making the `dequeue` operation costly**

---

1. Making the Enqueue operation costly

In this approach, we make sure that the oldest element added to the queue stays at the **top** of the stack, the second oldest below it and so on.

To achieve this, we will need two stacks. Following steps will be involved while enqueuing a new element to the queue.

NOTE: First stack(`S1`) is the main stack being used to store the data, while the second stack(`S2`) is to assist and store data temporarily during various operations.

1. If the queue is empty(means `S1` is empty), directly **push** the first element onto the stack `S1`.
2. If the queue is not empty, move all the elements present in the first stack(`S1`) to the second stack(`S2`), one by one. Then add the new element to the first stack, then move back all the elements from the second stack back to the first stack.
3. Doing so will always maintain the right order of the elements in the stack, with the 1st data element staying always at the **top**, with 2nd data element right below it and the new data element will be added to the bottom.

This makes removing an element from the queue very simple, all we have to do is call the `pop()` method for stack `S1`.

---

2. Making the Dequeue operation costly

In this approach, we insert a new element onto the stack `S1` simply by calling the `push()` function, but doing so will will push our first element towards the bottom of the stack, as we insert more elements to the stack.

But we want the first element to be removed first. Hence in the `dequeue` operation, we will have to use the second stack `S2`.

We will have to follow the following steps for `dequeue` operation:

1. If the queue is empty(means `S1` is empty), then we return an error message saying, the queue is empty.
2. If the queue is not empty, move all the elements present in the first stack(`S1`) to the second stack(`S2`), one by one. Then remove the element at the **top** from the second stack, and then move back all the elements from the second stack to the first stack.
3. The purpose of moving all the elements present in the first stack to the second stack is to reverse the order of the elements, because of which the first element inserted into the queue, is positioned at the top of the second stack, and all we have to do is call the `pop()` function on the second stack to remove the element.

NOTE: *We will be implementing the **second approach**, where we will make the `dequeue()` method costly.*

Adding Data to Queue - `enqueue()`

As our Queue has a Stack for data storage instead of arrays, hence we will be adding data to Stack, which can be done using the `push()` method, therefore `enqueue()` method will look like:

```
void Queue :: enqueue(int x)

{

    S1.push(x);

}
```

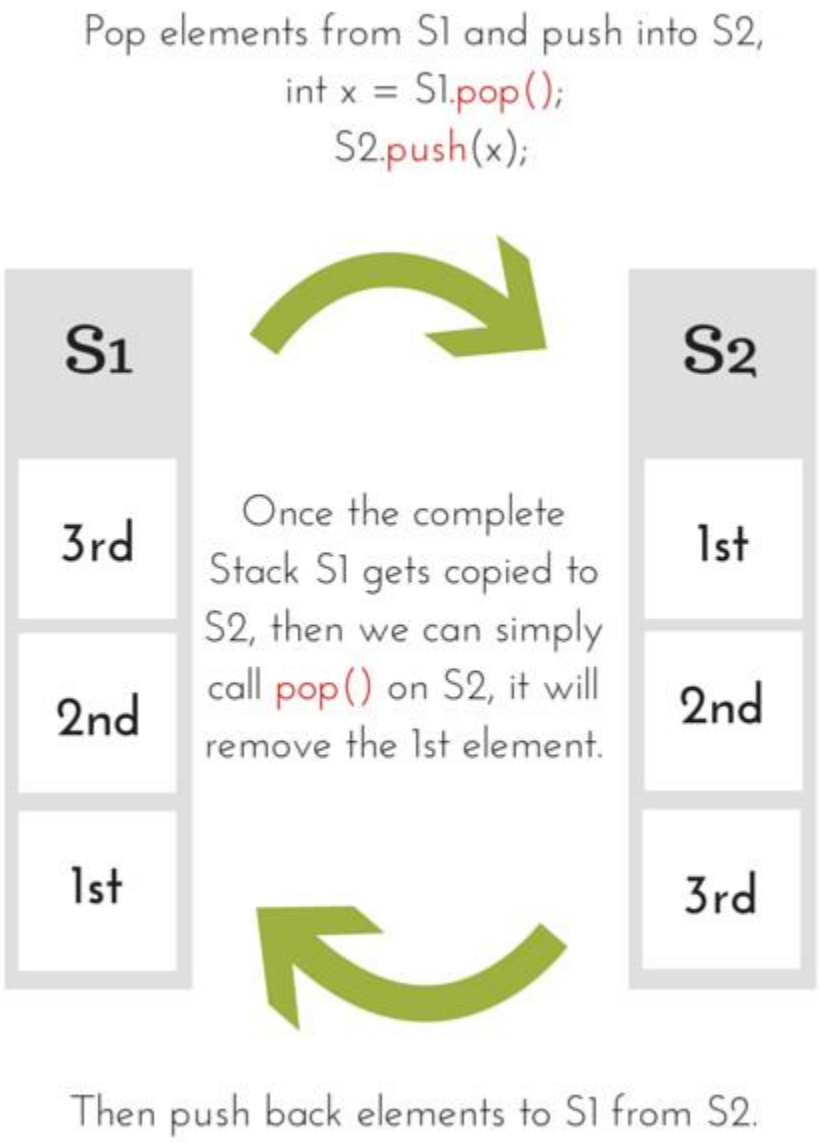
Copy

That's it, new data element is enqueued and stored in our queue.

Removing Data from Queue - `dequeue()`

When we say remove data from Queue, it always means taking out the element which was inserted first into the queue, then second and so on, as we have to follow the **FIFO approach**.

But if we simply perform `S1.pop()` in our **dequeue** method, then it will remove the Last element inserted into the queue first. So what to do now?



As you can see in the diagram above, we will move all the elements present in the first stack to the second stack, and then remove the **top** element, after that we will move back the elements to the first stack.

```
int Queue :: dequeue()

{

    int x, y;

    while(S1.isEmpty())

    {

        // take an element out of first stack

        x = S1.pop();

        // insert it into the second stack

        S2.push();

    }


    // removing the element

    y = S2.pop();


    // moving back the elements to the first stack

    while(!S2.isEmpty())

    {

        x = S2.pop();

        S1.push(x);

    }


    return y;

}
```

Copy

Now that we know the implementation of `enqueue()` and `dequeue()` operations, let's write a complete program to implement a queue using stacks.

Implementation in C++(OOPS)

We will not follow the traditional approach of using pointers, instead we will define proper classes, just like we did in the Stack tutorial.

```
/*  Below program is written in C++ language  */
```

```
# include<iostream>

using namespace std;

// implementing the stack class
class Stack
{
    int top;
    public:
    int a[10];  //Maximum size of Stack
    Stack()
    {
        top = -1;
    }

    // declaring all the function
    void push(int x);
    int pop();
    bool isEmpty();
};

// function to insert data into stack
void Stack::push(int x)
{
    if(top >= 10)
    {
        cout << "Stack Overflow \n";
    }
    else
```

```
    {
        a[++top] = x;
        cout << "Element Inserted into Stack\n";
    }
}

// function to remove data from the top of the stack
int Stack::pop()
{
    if(top < 0)
    {
        cout << "Stack Underflow \n";
        return 0;
    }
    else
    {
        int d = a[top--];
        return d;
    }
}

// function to check if stack is empty
bool Stack::isEmpty()
{
    if(top < 0)
    {
        return true;
    }
    else
    {
```

```
        return false;
    }
}


// implementing the queue class
class Queue {
    public:
    Stack S1, S2;

    //declaring enqueue method
    void enqueue(int x);

    //declaring dequeue method
    int dequeue();
};


// enqueue function
void Queue :: enqueue(int x)
{
    S1.push(x);
    cout << "Element Inserted into Queue\n";
}


// dequeue function
int Queue :: dequeue()
{
    int x, y;
    while(!S1.isEmpty())
    {
        // take an element out of first stack
```

```
        x = S1.pop();

        // insert it into the second stack

        S2.push(x);

    }


    // removing the element

    y = S2.pop();


    // moving back the elements to the first stack

    while(!S2.isEmpty())

    {

        x = S2.pop();

        S1.push(x);

    }


    return y;

}


// main function

int main()

{

    Queue q;

    q.enqueue(10);

    q.enqueue(100);

    q.enqueue(1000);

    cout << "Removing element from queue" << q.dequeue();


    return 0;

}
```

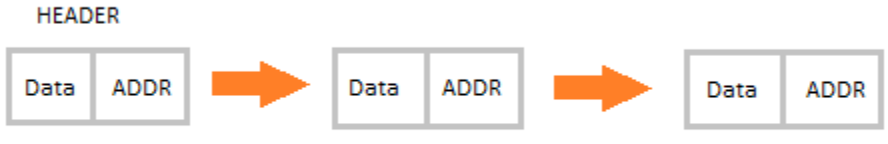
Copy

Introduction to Linked Lists

Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.

Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.

Linked Lists are used to create trees and graphs.



Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

There are 3 different implementations of Linked List available, they are:

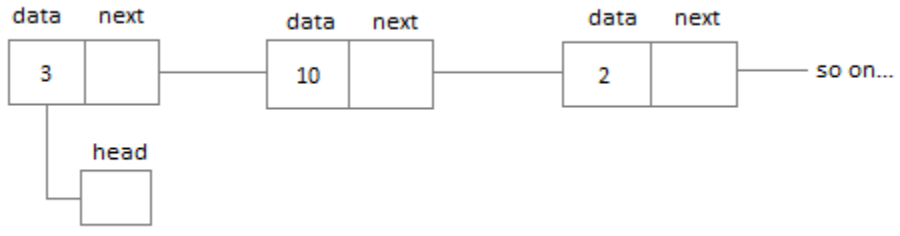
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Let's know more about them and how they are different from each other.

Singly Linked List

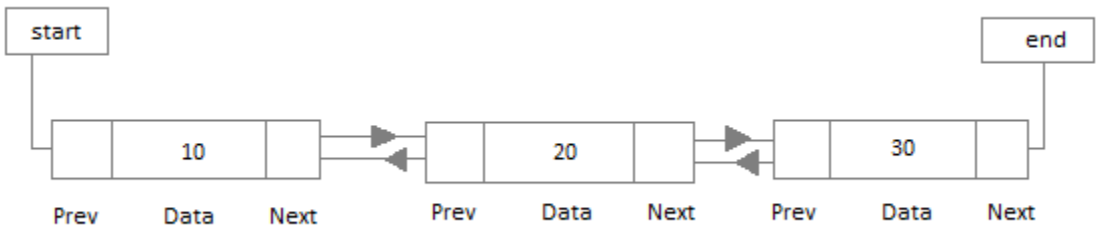
Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. `next`, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



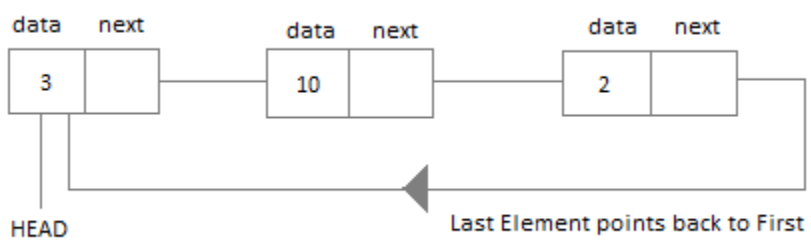
Doubly Linked List

In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.



Circular Linked List

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.



We will learn about all the 3 types of linked list, one by one, in the next tutorials. So click on **Next** button, let's learn more about linked lists.

Difference between Array and Linked List

Both Linked List and Array are used to store linear data of similar type, but an array consumes contiguous memory locations allocated at compile time, i.e. at the time of declaration of array, while for a linked list, memory is assigned as and when data is added to it, which means at runtime.

Before we proceed further with the differences between Array and Linked List, if you are not familiar with Array or Linked list or both, you can check these topics first:

- Array in C
- Linked List

This is the basic and the most important difference between a linked list and an array. In the section below, we will discuss this in details along with highlighting other differences.

Linked List vs. Array

Array is a datatype which is widely implemented as a default type, in almost all the modern programming languages, and is used to store data of similar type.

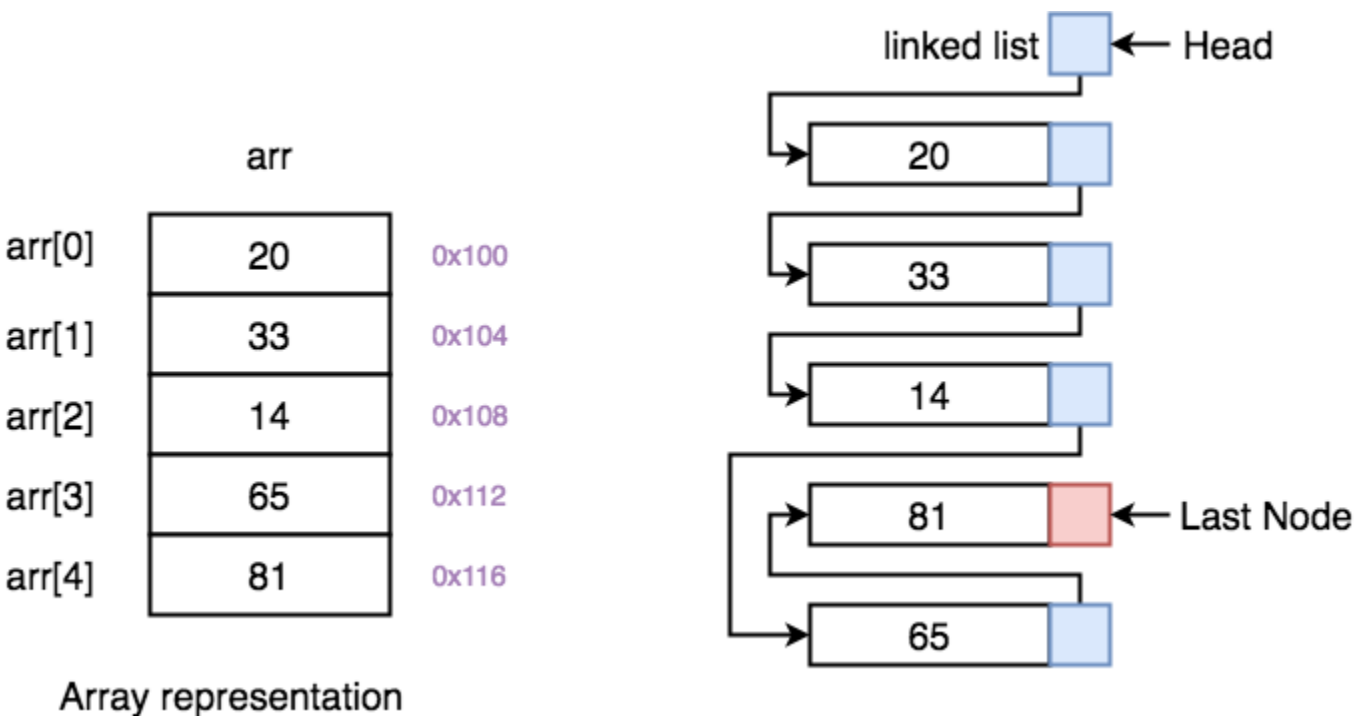
But there are many usecases, like the one where we don't know the quantity of data to be stored, for which advanced data structures are required, and one such data structure is **linked list**.

Let's understand how array is different from Linked list.

ARRAY	LINKED LIST
Array is a collection of elements of similar data type.	Linked List is an ordered collection of elements of same type, which are connected to each other using pointers.
Array supports **Random Access**, which means elements can be accessed directly using their index, like `arr[0]` for 1st element, `arr[6]` for 7th element etc. Hence, accessing elements in an array is **fast** with a constant time complexity of `O(1)`.	Linked List supports **Sequential Access**, which means to access any element/node in a linked list, we have to sequentially traverse the complete linked list, upto that element. To access **nth** element of a linked list, time complexity is `O(n)`.
In an array, elements are stored in **contiguous memory location** or consecutive manner in the memory.	In a linked list, new elements can be stored anywhere in the memory. Address of the memory location allocated to the new element is stored in the previous node of linked list, hence formaing a link between the two nodes/elements.

In array, **Insertion and Deletion** operation takes more time, as the memory locations are consecutive and fixed.	In case of linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list. Insertion and Deletion operations are **fast** in linked list.
Memory is allocated as soon as the array is declared, at **compile time**. It's also known as **Static Memory Allocation**.	Memory is allocated at **runtime**, as and when a new node is added. It's also known as **Dynamic Memory Allocation**.
In array, each element is independent and can be accessed using it's index value.	In case of a linked list, each node/element points to the next, previous, or maybe both nodes.
Array can be **single dimensional**, **two dimensional** or **multidimensional**	Linked list can be **Linear(Singly) linked list**, **Doubly linked list** or **Circular linked list** linked list.
Size of the array must be specified at time of array decalaration.	Size of a Linked list is variable. It grows at runtime, as more nodes are added to it.
Array gets memory allocated in the **Stack** section.	Whereas, linked list gets memory allocated in **Heap** section.

Below we have a pictorial representation showing how consecutive memory locations are allocated for array, while in case of linked list random memory locations are assigned to nodes, but each node is connected to its next node using **pointer**.



On the left, we have **Array** and on the right, we have **Linked List**.

Why we need pointers in Linked List? [Deep Dive]

In case of array, memory is allocated in contiguous manner, hence array elements get stored in consecutive memory locations. So when you have to access any array element, all we have to do is use the array index, for example `arr[4]` will directly access the 5th memory location, returning the data stored there.

But in case of linked list, data elements are allocated memory at runtime, hence the memory location can be anywhere. Therefore to be able to access every node of the linked list, address of every node is stored in the previous node, hence forming a link between every node.

We need this additional **pointer** because without it, the data stored at random memory locations will be lost. We need to store somewhere all the memory locations where elements are getting stored.

Yes, this requires an additional memory space with each node, which means an additional space of $O(n)$ for every n node linked list.

Linear Linked List

Linear Linked list is the default linked list and a linear data structure in which data is not stored in contiguous memory locations but each data node is connected to the next data node via a pointer, hence forming a chain.

The element in such a linked list can be inserted in 2 ways:

- Insertion at beginning of the list.
- Insertion at the end of the list.

Hence while writing the code for Linked List we will include methods to insert or add new data elements to the linked list, both, at the beginning of the list and at the end of the list.

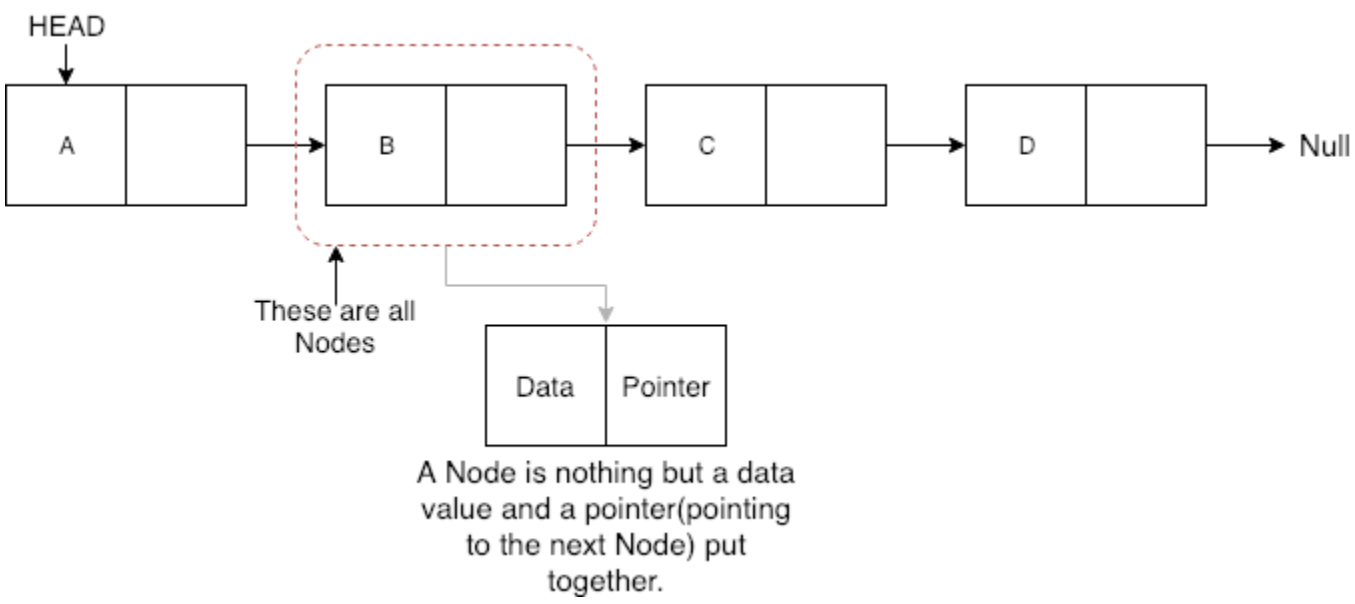
We will also be adding some other useful methods like:

- Checking whether Linked List is empty or not.
- Searching any data element in the Linked List
- Deleting a particular Node(data element) from the List

Before learning how we insert data and create a linked list, we must understand the components forming a linked list, and the main component is the **Node**.

What is a Node?

A Node in a linked list holds the data value and the pointer which points to the location of the next node in the linked list.



A Node is nothing but a data value and a pointer(pointing to the next Node) put together.

In the picture above we have a linked list, containing 4 nodes, each node has some data(A, B, C and D) and a pointer which stores the location of the next node.

You must be wondering **why we need to store the location of the next node**. Well, because the memory locations allocated to these nodes are not contiguous hence each node should know where the next node is stored.

As the node is a combination of multiple information, hence we will be defining a class for `Node` which will have a variable to store **data** and another variable to store the **pointer**. In C language, we create a structure using the `struct` keyword.

```
class Node
```

```
{
    public:
    // our linked list will only hold int data
    int data;
    //pointer to the next node
    node* next;

    // default constructor
    Node()
    {
        data = 0;
        next = NULL;
    }

    // parameterised constructor
    Node(int x)
    {
        data = x;
        next = NULL;
    }
}
```

Copy

We can also make the `Node` class properties `data` and `next` as **private**, in that case we will need to add the getter and setter methods to access them(don't know what getter and setter methods are: Inline Functions in C++). You can add the getter and setter functions to the `Node` class like this:

```
class Node
{
    // our linked list will only hold int data
    int data;
    //pointer to the next node
    node* next;
```

```
    // default constructor same as above


    // parameterised constructor same as above


    /* getters and setters */
    // get the value of data
    int getData()
    {
        return data;
    }


    // to set the value for data
    void setData(int x)
    {
        this.data = x;
    }
    // get the value of next pointer
    node* getNext()
    {
        return next;
    }
    // to set the value for pointer
    void setNext(node *n)
    {
        this.next = n;
    }
}
```

Copy

The `Node` class basically creates a node for the data to be included into the Linked List. Once the object for the class `Node` is created, we use various functions to fit in that node into the Linked List.

Linked List class

As we are following the complete OOPS methodology, hence we will create a separate class for **Linked List**, which will have all the methods like insertion, search, deletion etc. Also, the linked list class will have a pointer called `head` to store the location of the first node which will be added to the linked list.

```
class LinkedList
{
    public:
    node *head;
    //declaring the functions


    //function to add Node at front
    int addAtFront(node *n);
    //function to check whether Linked list is empty
    int isEmpty();
    //function to add Node at the End of list
    int addAtEnd(node *n);
    //function to search a value
    node* search(int k);
    //function to delete any Node
    node* deleteNode(int x);


    LinkedList()
    {
        head = NULL;
    }
}
```

Copy

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int LinkedList :: addAtFront(node *n) {
  int i = 0;
  //making the next of the new Node point to Head
  n->next = head;
  //making the new Node as Head
  head = n;
  i++;
  //returning the position where Node is added
  return i;
}
```

Copy

Inserting at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, hence making the new node the last Node.

```
int LinkedList :: addAtEnd(node *n) {
  //If list is empty
  if(head == NULL) {
    //making the new Node as Head
```

```
    head = n;

    //making the next pointe of the new Node as Null

    n->next = NULL;

  }

  else {

    //getting the last node

    node *n2 = getLastNode();

    n2->next = n;

  }

}



node* LinkedList :: getLastNode() {

  //creating a pointer pointing to Head

  node* ptr = head;

  //Iterating over the list till the node whose Next pointer points to
null

  //Return that node, because that will be the last node.

  while(ptr->next!=NULL) {

    //if Next is not Null, take the pointer one step forward

    ptr = ptr->next;

  }

  return ptr;

}
```

Copy

Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```


node* LinkedList :: search(int x) {

  node *ptr = head;
```

```
  while(ptr != NULL && ptr->data != x) {

    //until we reach the end or we find a Node with data x, we keep
moving

    ptr = ptr->next;

  }

  return ptr;

}
```

Copy

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

```
node* LinkedList :: deleteNode(int x) {

  //searching the Node with data x

  node *n = search(x);

  node *ptr = head;

  if(ptr == n) {

    ptr->next = n->next;

    return n;

  }

  else {

    while(ptr->next != n) {

      ptr = ptr->next;

    }

    ptr->next = n->next;

    return n;
```

```
  }

}
```

Copy

Checking whether the List is empty or not

We just need to check whether the **Head** of the List is `NULL` or not.

```
int LinkedList :: isEmpty() {

  if(head == NULL) {

    return 1;

  }

  else { return 0; }

}
```

Copy

Now you know a lot about how to handle List, how to traverse it, how to search an element. You can yourself try to write new methods around the List.

If you are still figuring out, how to call all these methods, then below is how your `main()` method will look like. As we have followed OOP standards, we will create the objects of **LinkedList** class to initialize our List and then we will create objects of **Node** class whenever we want to add any new node to the List.

```
int main() {

  LinkedList L;

  //We will ask value from user, read the value and add the value to
our Node

  int x;

  cout << "Please enter an integer value : ";

  cin >> x;

  Node *n1;

  //Creating a new node with data as x

  n1 = new Node(x);

  //Adding the node to the list

  L.addAtFront(n1);
```

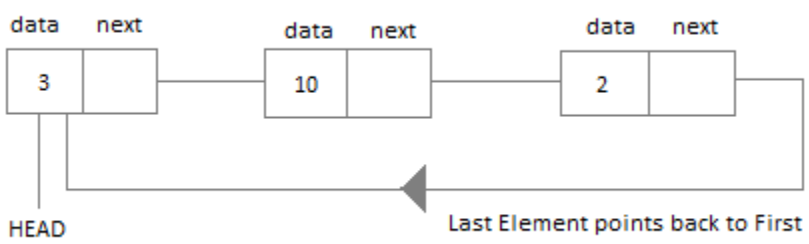
```
}
```

Copy

Similarly you can call any of the functions of the LinkedList class, add as many Nodes you want to your List.

Circular Linked List

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last **Node** will have it's **next** point to the **Head** of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.

So this will be oue Node class, as we have already studied in the lesson, it will be used to form the List.

```
class Node {
  public:
  int data;
  //pointer to the next node
  node* next;

  node() {
```

```
    data = 0;

    next = NULL;

  }



  node(int x) {

    data = x;

    next = NULL;

  }

}
```

Copy

Circular Linked List

Circular Linked List class will be almost same as the Linked List class that we studied in the previous lesson, with a few difference in the implementation of class methods.

```
class CircularLinkedList {

  public:

  node *head;

  //declaring the functions



  //function to add Node at front

  int addAtFront(node *n);

//function to check whether Linked list is empty

int isEmpty();

//function to add Node at the End of list

int addAtEnd(node *n);

//function to search a value

node* search(int k);

//function to delete any Node

node* deleteNode(int x);
```

```
  CircularLinkedList() {

    head = NULL;

  }

}
```

Copy

Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the fisrt Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```
int CircularLinkedList :: addAtFront(node *n) {

  int i = 0;

  /* If the list is empty */

  if(head == NULL) {

    n->next = head;

    //making the new Node as Head

    head = n;

    i++;

  }

  else {

    n->next = head;

    //get the Last Node and make its next point to new Node

    Node* last = getLastNode();

    last->next = n;

    //also make the head point to the new first Node
```

```
    head = n;

    i++;

  }

  //returning the position where Node is added

  return i;

}
```

Copy

Insertion at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, and make the next of the Newly added Node point to the Head of the List.

```
int CircularLinkedList :: addAtEnd(node *n) {

  //If list is empty

  if(head == NULL) {

    //making the new Node as Head

    head = n;

    //making the next pointer of the new Node as Null

    n->next = NULL;

  }

  else {

    //getting the last node

    node *last = getLastNode();

    last->next = n;

    //making the next pointer of new node point to head

    n->next = head;

  }

}
```

Copy

Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```
node* CircularLinkedList :: search(int x) {

  node *ptr = head;

  while(ptr != NULL && ptr->data != x) {

    //until we reach the end or we find a Node with data x, we keep
moving

    ptr = ptr->next;

  }

  return ptr;

}
```

Copy

Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted. And update the next pointer of the Last Node as well.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.
- If the Node is at the end, then remove it and make the new last node point to the head.

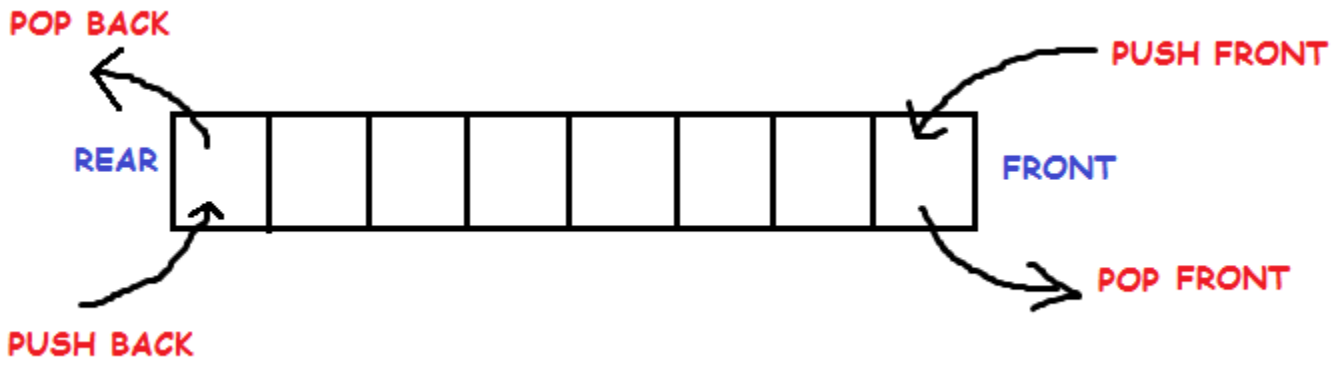
```
node* CircularLinkedList :: deleteNode(int x) {

  //searching the Node with data x

  node *n = search(x);

  node *ptr = head;

  if(ptr == NULL) {

    cout << "List is empty";
```

```
    return NULL;
  }
  else if(ptr == n) {
    ptr->next = n->next;
    return n;
  }
  else {
    while(ptr->next != n) {
      ptr = ptr->next;
    }
    ptr->next = n->next;
    return n;
  }
}
```

Copy

Double Ended Queue

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



Implementation of Double ended Queue

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

- **push_back :** inserts element at back
- **push_front :** inserts element at front
- **pop_back :** removes last element
- **pop_front :** removes first element
- **get_back :** returns last element
- **get_front :** returns first element
- **empty :** returns true if queue is empty
- **full :** returns true if queue is full

```
// Maximum size of array or Dequeue



#define SIZE 5



class Dequeue

{

    //front and rear to store the head and tail pointers

    int  *arr;

    int front, rear;



    public :
```

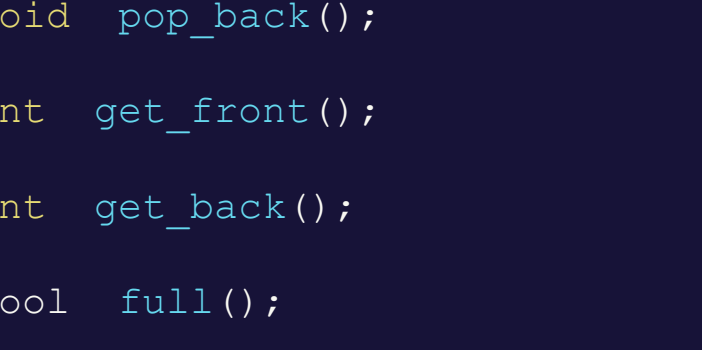
```
    Dequeue()

    {

     //Create the array

        arr = new int[SIZE];



        //Initialize front and rear with -1

        front = -1;

        rear = -1;

    }



    // Operations on Deque

    void  push_front(int );

    void  push_back(int );

    void  pop_front();

    void  pop_back();

    int  get_front();

    int  get_back();

    bool  full();

    bool  empty();

};
```

Copy

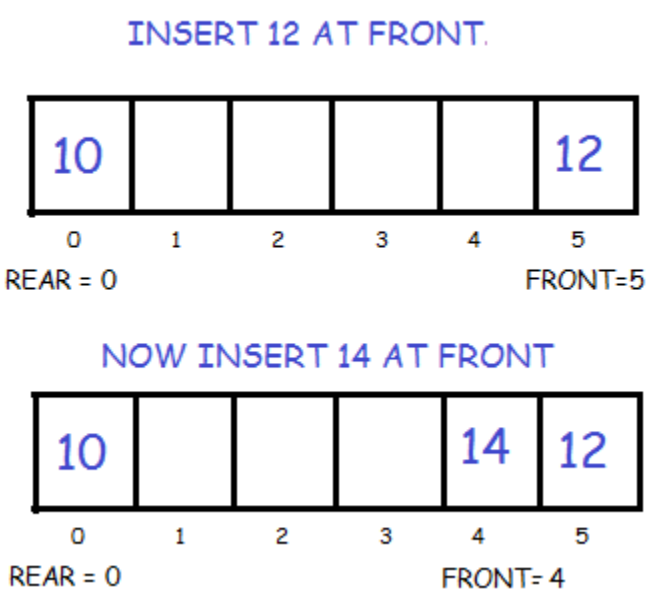
Insert Elements at Front

First we check if the queue is full. If its not full we insert an element at front end by following the given conditions :

- If the queue is empty then intialize front and rear to 0. Both will point to the first element.



- Else we decrement front and insert the element. Since we are using circular array, we have to keep in mind that if front is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.




```
void Dequeue :: push_front(int key)

{

    if(full())

    {

        cout << "OVERFLOW\n";

    }

    else

    {

     //If queue is empty

     if(front == -1)

          front = rear = 0;


   //If front points to the first position element

      else if(front == 0)

          front = SIZE-1;


      else

        --front;


      arr[front] = key;

  }
```

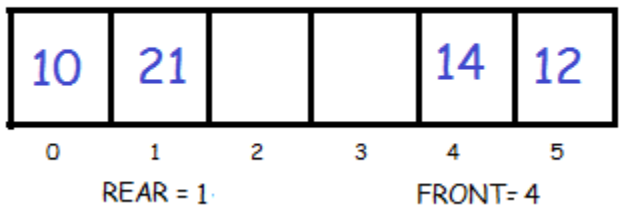
```
}
```

Copy

Insert Elements at back

Again we check if the queue is full. If its not full we insert an element at back by following the given conditions:

- If the queue is empty then intialize front and rear to 0. Both will point to the first element.
- Else we increment rear and insert the element. Since we are using circular array, we have to keep in mind that if rear is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.




```
void Dequeue :: push_back(int key)

{

    if(full())

    {

        cout << "OVERFLOW\n";

    }

    else

    {

        //If queue is empty

    if(front == -1)

        front = rear = 0;


    //If rear points to the last element

    else if(rear == SIZE-1)

        rear = 0;


    else

      ++rear;
```

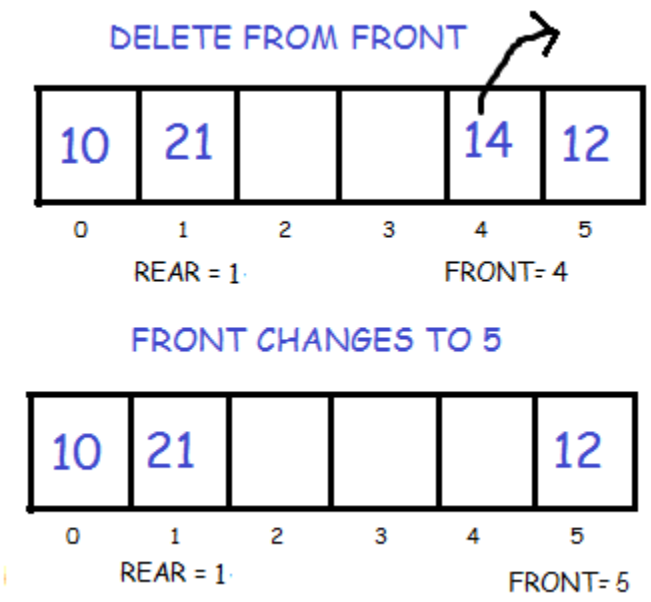
```
        arr[rear] = key;

    }

}
```

Copy

Delete First Element

In order to do this, we first check if the queue is empty. If its not then delete the front element by following the given conditions :

- If only one element is present we once again make front and rear equal to -1.
- Else we increment front. But we have to keep in mind that if front is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.




```
void Dequeue :: pop_front()

{

    if(empty())

    {

        cout << "UNDERFLOW\n";

    }

    else

    {

     //If only one element is present

        if(front == rear)

          front = rear = -1;
```

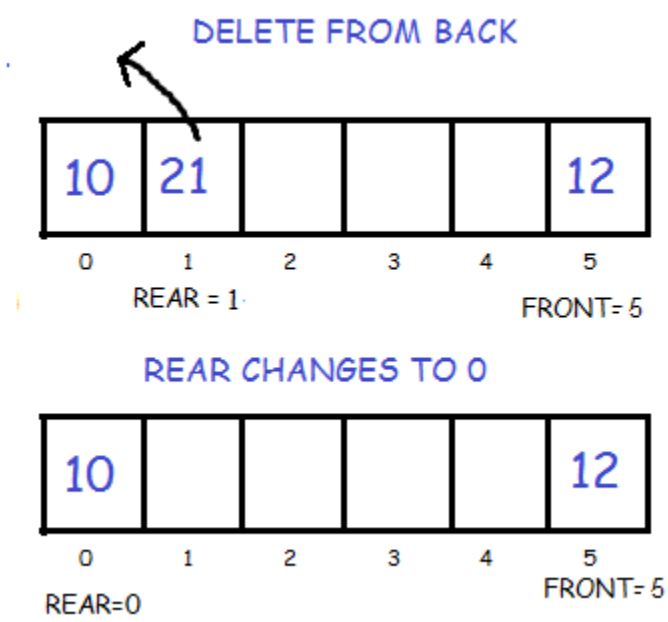
```
        //If front points to the last element
        else if(front == SIZE-1)
          front = 0;


        else
          ++front;
    }
}
```

Copy

Delete Last Element

Inorder to do this, we again first check if the queue is empty. If its not then we delete the last element by following the given conditions :

- If only one element is present we make front and rear equal to -1.
- Else we decrement rear. But we have to keep in mind that if rear is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.




```
void Dequeue :: pop_back()
{
    if(empty())
    {
        cout << "UNDERFLOW\n";
    }
    else
```

```
    {
     //If only one element is present
        if(front == rear)
          front = rear = -1;

        //If rear points to the first position element
        else if(rear == 0)
          rear = SIZE-1;

        else
            --rear;
    }
}
```

Copy

Check if Queue is empty

It can be simply checked by looking where front points to. If front is still intialized with -1, the queue is empty.

```
bool Dequeue :: empty()
{
    if(front == -1)
     return true;
    else
     return false;
}
```

Copy

Check if Queue is full

Since we are using circular array, we check for following conditions as shown in code to check if queue is full.

```
bool Dequeue :: full()
```

```
{

    if((front == 0 && rear == SIZE-1)   ||

     (front == rear + 1))

        return true;

    else

        return false;

}
```

Copy

Return First Element

If the queue is not empty then we simply return the value stored in the position which front points.

```
int Dequeue :: get_front()

{

    if(empty())

    {cout << "f=" <<front << endl;

        cout << "UNDERFLOW\n";

        return -1;

    }

    else

    {

        return arr[front];

    }

}
```

Copy

Return Last Element

If the queue is not empty then we simply return the value stored in the position which rear points.

```
int Dequeue :: get_back()

{
```

```
    if(empty())
    {
        cout << "UNDERFLOW\n";
        return -1;
    }
    else
    {
        return arr[rear];
    }
}
```

Copy

Stack using Queue

A Stack is a **Last In First Out(LIFO)** structure, i.e, the element that is added last in the stack is taken out first. Our goal is to implement a Stack using Queue for which will be using two queues and design them in such a way that **pop** operation is same as dequeue but the **push** operation will be a little complex and more expensive too.

Implementation of Stack using Queue

Assuming we already have a class implemented for Queue, we first design the class for Stack. It will have the methods `push()` and `pop()` and two queues.

```
class Stack

{

    public:

    // two queue

    Queue Q1, Q2;

    // push method to add data element

    void push(int);

    // pop method to remove data element

    void pop();

};
```

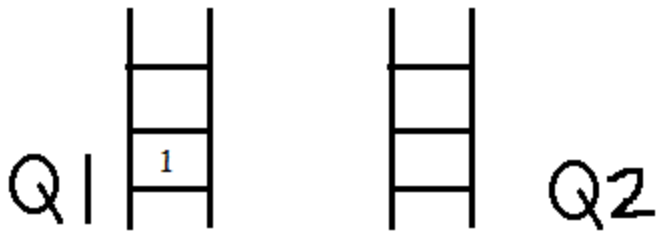
Copy

Inserting Data in Stack

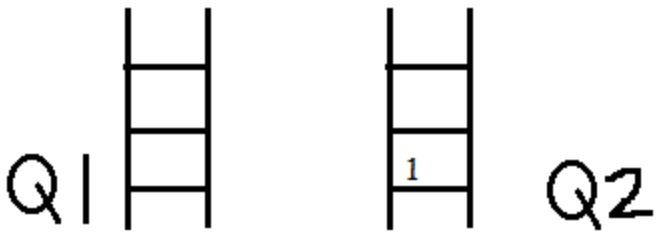
Since we are using Queue which is **First In First Out(FIFO)** structure , i.e, the element which is added first is taken out first, so we will implement the **push** operation in such a way that whenever there is a **pop** operation, the stack always pops out the last element added.

In order to do so, we will require two queues, `Q1` and `Q2`. Whenever the **push** operation is invoked, we will enqueue(move in this case) all the elements of `Q1` to `Q2` and then enqueue the new element to `Q1`. After this we will enqueue(move in this case) all the elements from `Q2` back to `Q1`.

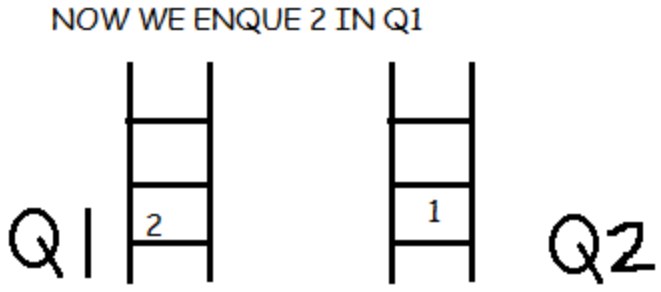
INITIALLY BOTH THE QUEUES ARE EMPTY.
WHEN WE GET A QUERY PUSH(1), SINCE Q1 IS EMPTY,
WE DIRECTLY INSERT 1 TO Q1.



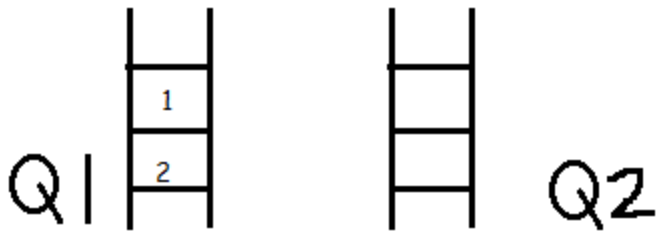
NOW IF WE GET A QUERY PUSH(2), WE WILL
FIRST ENQUE ALL ELEMENTS FROM Q1 TO Q2



NOW WE ENQUE 2 IN Q1



FINALLY WE DEQUE ALL THE ELEMENTS
FROM Q2 AND ENQUE THEM IN Q1



HENCE WE SEE THAT THE LAST ELEMENT ADDED , I.E., 2
IS IN THE FRONT OF Q1. SO IF WE GET A POP QUERY,
JUST USING THE DEQUE OPERATION IN Q1 WILL POP
OUT THE LAST ELEMENT ADDED IN THE STACK.

So let's implement this in our code,

```
void Stack :: push(int x)

{

     // move all elements in Q1 to Q2

     while(!Q1.isEmpty())

     {

          int temp = Q1.deque();

          Q2.enque(temp);

     }



     // add the element which is pushed into Stack

     Q1.enque(x);
```

```
        // move back all elements back to Q1 from Q2
        while(!Q2.isEmpty())
        {
              int temp = Q2.deque();
              Q1.enque(temp);
        }
}
```

Copy

It must be clear to you now, why we are using two queues. Actually the queue `Q2` is just for the purpose of keeping the data temporarily while operations are executed.

In this way we can ensure that whenever the **pop** operation is invoked, the stack always pops out the last element added in `Q1` queue.

Removing Data from Stack

Like we have discussed above, we just need to use the dequeue operation on our queue `Q1`. This will give us the last element added in Stack.

```
int Stack :: pop()
{
      return Q1.deque();
}
```

Copy

Time Complexity Analysis

When we implement Stack using a Queue the **push** operation becomes expensive.

- Push operation: O(n)
- Pop operation: O(1)

Conclusion

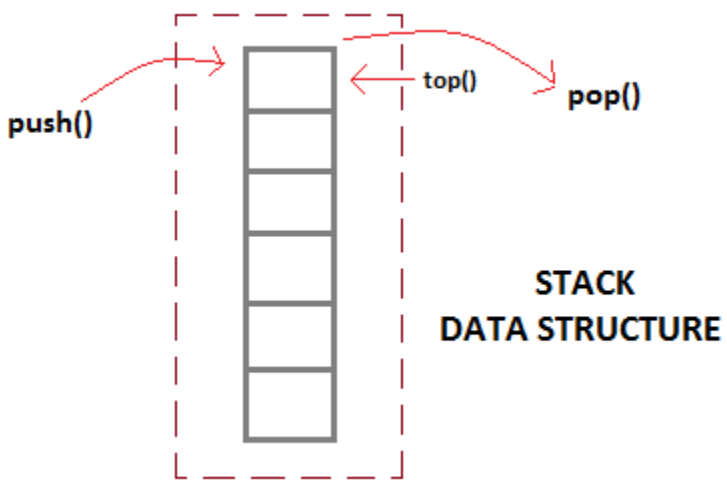
When we say "implementing Stack using Queue", we mean how we can make a Queue behave like a Stack, after all they are all logical entities. So for any data structure to act as a Stack, it should have `push()` method to add data on top and `pop()` method to remove data from top. Which

is exactly what we did and hence accomplished to make a Queue(in this case two Queues) behave as a Stack.

Stack using Linked List

Stack as we know is a **Last In First Out(LIFO)** data structure. It has the following operations :

- **push:** push an element into the stack
- **pop:** remove the last element added
- **top:** returns the element at top of stack



Implementation of Stack using Linked List

Stacks can be easily implemented using a linked list. Stack is a data structure to which a data can be added using the `push()` method and data can be removed from it using the `pop()` method. With Linked list, the **push** operation can be replaced by the `addAtFront()` method of linked list and **pop** operation can be replaced by a function which deletes the front node of the linked list.

In this way our Linked list will virtually become a Stack with `push()` and `pop()` methods.

First we create a class **node**. This is our Linked list node class which will have **data** in it and a **node pointer** to store the address of the next node element.

```
class node

{

     int data;

     node *next;

};
```

Copy

Then we define our stack class,

```
class Stack

{

     node *front;  // points to the head of list

     public:

     Stack()
```

```
	{
		front = NULL;
	}
	// push method to add data element
	void push(int);
	// pop method to remove data element
	void pop();
	// top method to return top data element
	int top();
};
```

Copy

Inserting Data in Stack (Linked List)

In order to insert an element into the stack, we will create a node and place it in front of the list.

```
void Stack :: push(int d)
{
	// creating a new node
	node *temp;
	temp = new node();
	// setting data to it
	temp->data = d;

	// add the node in front of list
	if(front == NULL)
	{
		temp->next = NULL;
	}
	else
	{
		temp->next = front;
```

```
	}

	front = temp;

}
```

Copy

Now whenever we will call the `push()` function a new node will get added to our list in the front, which is exactly how a stack behaves.

Removing Element from Stack (Linked List)

In order to do this, we will simply delete the first node, and make the second node, the head of the list.

```
void Stack :: pop()

{

	// if empty

	if(front == NULL)

		cout << "UNDERFLOW\n";


	// delete the first element

	else

	{

		node *temp = front;

		front = front->next;

		delete(temp);

	}

}
```

Copy

Return Top of Stack (Linked List)

In this, we simply return the data stored in the head of the list.

```
int Stack :: top()

{

	return front->data;
```

```
}
```

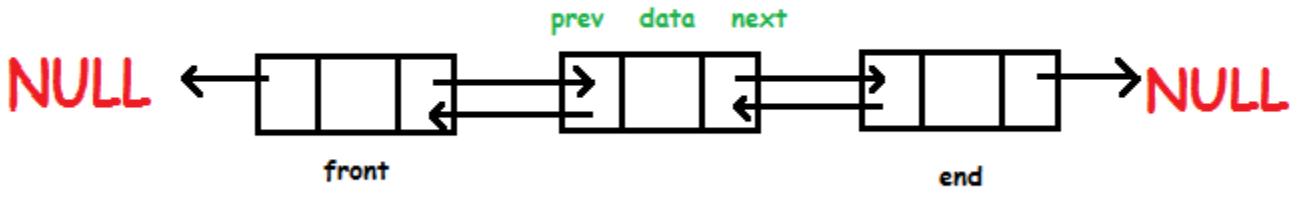
Copy

Conclusion

When we say "implementing Stack using Linked List", we mean how we can make a Linked List behave like a Stack, after all they are all logical entities. So for any data structure to act as a Stack, it should have `push()` method to add data on top and `pop()` method to remove data from top. Which is exactly what we did and hence accomplished to make a Linked List behave as a Stack.

Doubly Linked List

Doubly linked list is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list. The first node of the list has its previous link pointing to NULL similarly the last node of the list has its next node pointing to NULL.



The two links help us to traverse the list in both backward and forward direction. But storing an extra link requires some extra space.

Implementation of Doubly Linked List

First we define the node.

```
struct node

{

     int data;       // Data

     node *prev;     // A reference to the previous node

     node *next;     // A reference to the next node

};
```

Copy

Now we define our class Doubly Linked List. It has the following methods:

- **add_front:** Adds a new node in the beginning of list
- **add_after:** Adds a new node after another node
- **add_before:** Adds a new node before another node
- **add_end:** Adds a new node in the end of list
- **delete:** Removes the node
- **forward_traverse:** Traverse the list in forward direction
- **backward_traverse:** Traverse the list in backward direction

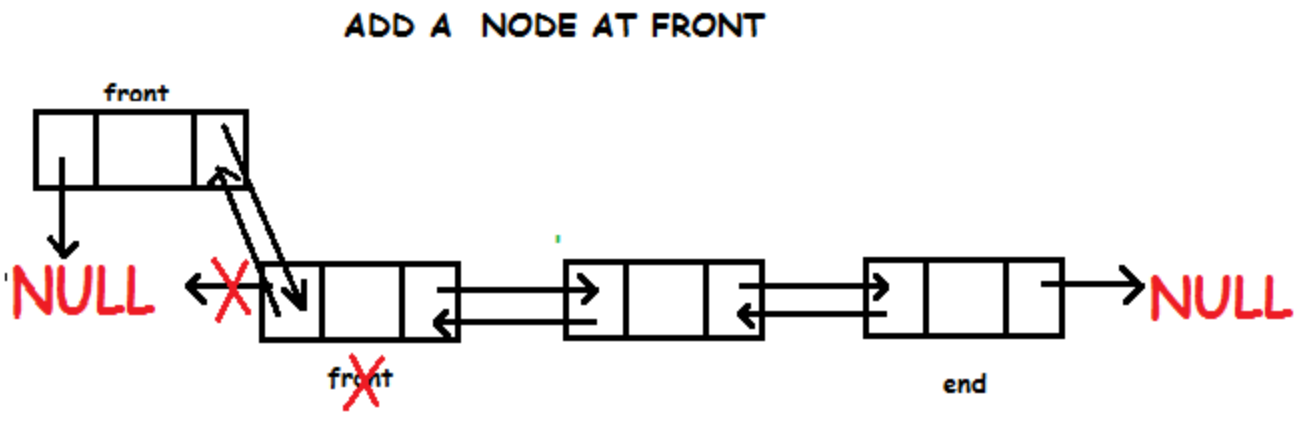
```
class Doubly_Linked_List

{

     node *front;    // points to first node of list
```

```
    node *end;      // points to first las of list
    public:
    Doubly_Linked_List()
    {
        front = NULL;
        end = NULL;
    }
    void add_front(int );
    void add_after(node* , int );
    void add_before(node* , int );
    void add_end(int );
    void delete_node(node*);
    void forward_traverse();
    void backward_traverse();
};
```

Copy

Insert Data in the beginning

1. The **prev** pointer of first node will always be NULL and **next** will point to **front**.
2. If the node is inserted is the first node of the list then we make **front** and **end** point to this node.
3. Else we only make **front** point to this node.



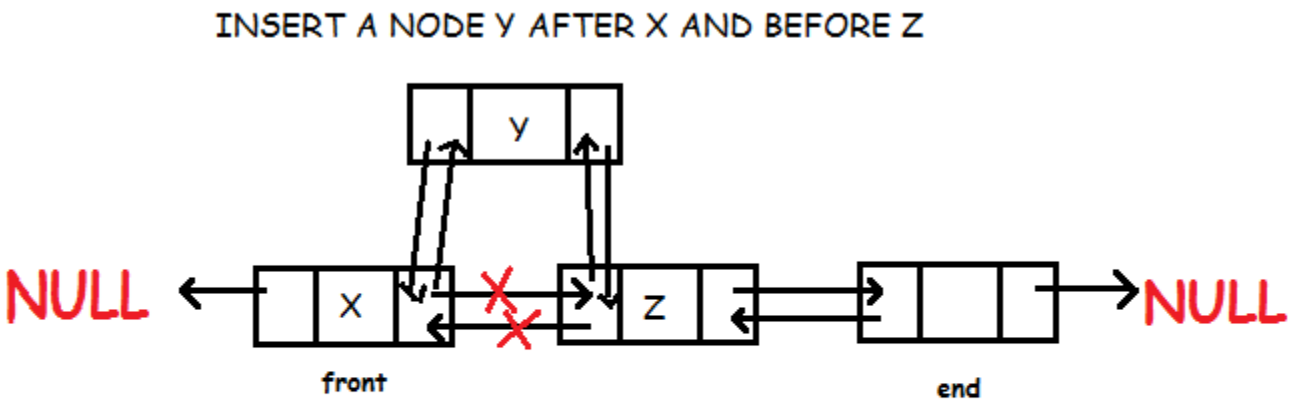
1. WE MAKE THE CURRENT FRONT NODE'S PREV POINT TO NEW NODE.
2. MAKE NEXT OF NEW NODE POINT TO CURRENT FRONT AND PREV OF NEW NODE POINT TO NULL.
3. WE CHANGE FRONT NODE TO THE NEW NODE.

```
void Doubly_Linked_List :: add_front(int d)

{

     // Creating new node

     node *temp;

     temp = new node();

     temp->data = d;

     temp->prev = NULL;

     temp->next = front;



     // List is empty

     if(front == NULL)

          end = temp;


     else

          front->prev = temp;


     front = temp;

}
```

Copy

Insert Data before a Node

Let’s say we are inserting node X before Y. Then X’s next pointer will point to Y and X’s prev pointer will point the node Y’s prev pointer is pointing. And Y’s prev pointer will now point to X. We need to make sure that if Y is the first node of list then after adding X we make front point to X.




```
void Doubly_Linked_List :: add_before(node *n, int d)

{

     node *temp;

     temp = new node();

     temp->data = d;

     temp->next = n;

     temp->prev = n->prev;

     n->prev = temp;



     //if node is to be inserted before first node

     if(n->prev == NULL)

          front = temp;

}
```

Copy

Insert Data after a Node

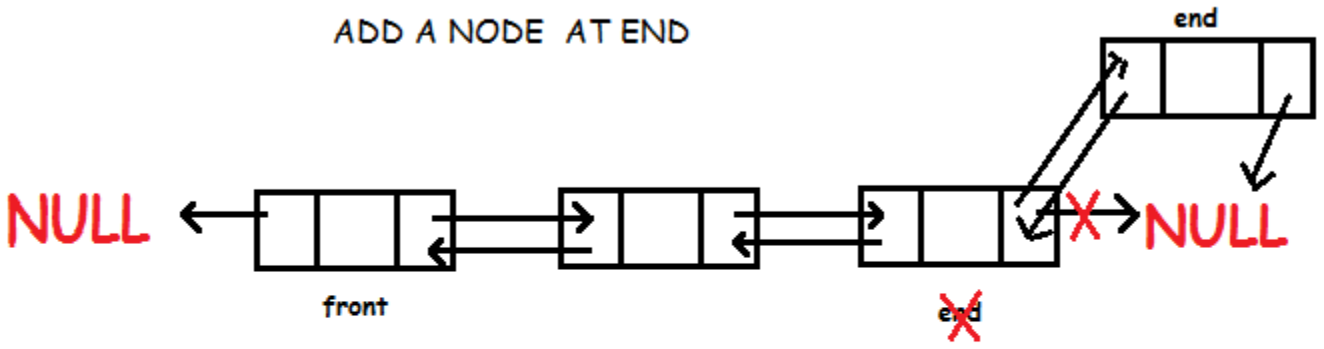
Let’s say we are inserting node Y after X. Then Y’s prev pointer will point to X and Y’s next pointer will point the node X’s next pointer is pointing. And X’s next pointer will now point to Y. We need to make sure that if X is the last node of list then after adding Y we make end point to Y.

```
void Doubly_Linked_List :: add_after(node *n, int d)

{

     node *temp;

     temp = new node();

     temp->data = d;

     temp->prev = n;

     temp->next = n->next;

     n->next = temp;



     //if node is to be inserted after last node

     if(n->next == NULL)

          end = temp;

}
```

Copy

Insert Data in the end

1. The **next** pointer of last node will always be NULL and **prev** will point to end.
2. If the node is inserted is the first node of the list then we make front and end point to this node.
3. Else we only make end point to this node.



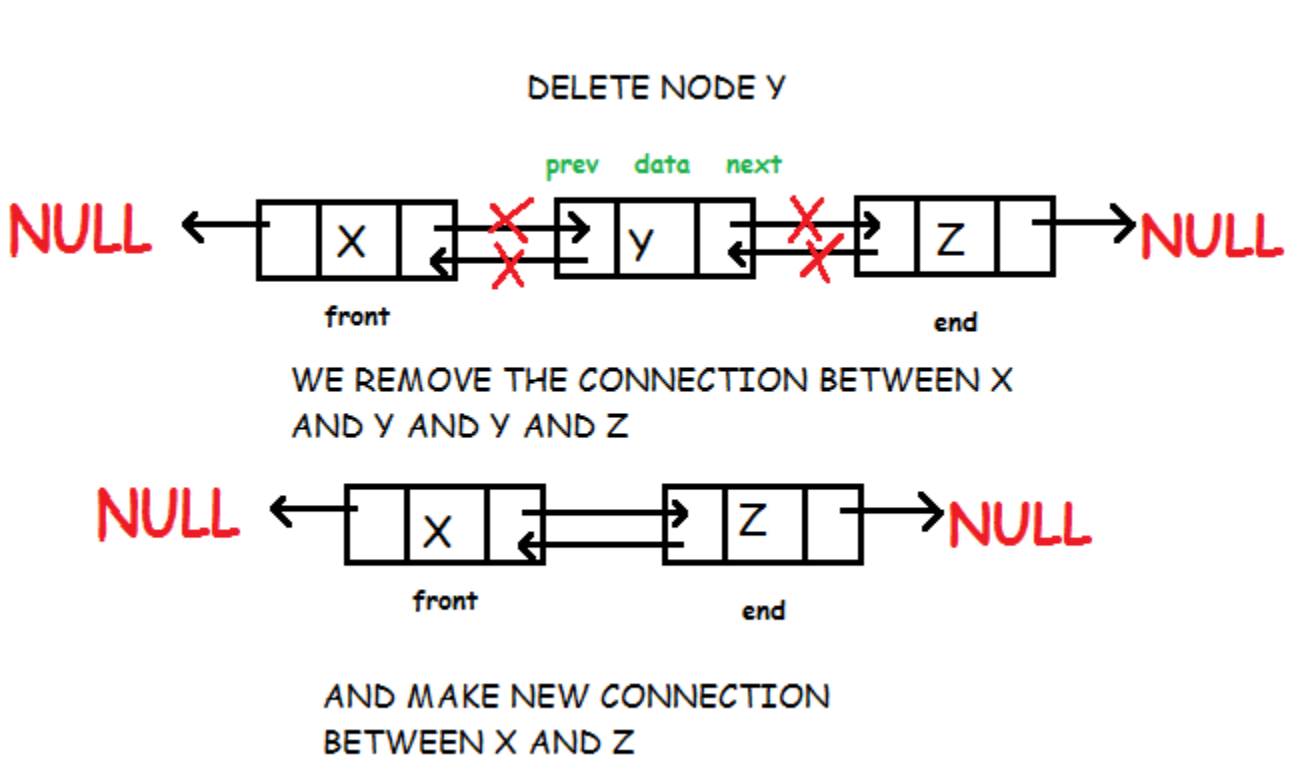

```
void Doubly_Linked_List :: add_end(int d)

{
```

```
    // create new node

    node *temp;

    temp = new node();

    temp->data = d;

    temp->prev = end;

    temp->next = NULL;


    // if list is empty

    if(end == NULL)

         front = temp;

    else

         end->next = temp;

    end = temp;

}
```

Copy

Remove a Node

Removal of a node is quite easy in Doubly linked list but requires special handling if the node to be deleted is first or last element of the list. Unlike singly linked list where we require the previous node, here only the node to be deleted is needed. We simply make the next of the previous node point to next of current node (node to be deleted) and prev of next node point to prev of current node. Look code for more details.




```
void Doubly_Linked_List :: delete_node(node *n)
```

```
{
     // if node to be deleted is first node of list
     if(n->prev == NULL)
     {
          front = n->next; //the next node will be front of list
          front->prev = NULL;
     }
     // if node to be deleted is last node of list
     else if(n->next == NULL)
     {
          end = n->prev;   // the previous node will be last of list
          end->next = NULL;
     }
     else
     {
          //previous node's next will point to current node's next
          n->prev->next = n->next;
          //next node's prev will point to current node's prev
          n->next->prev = n->prev;
     }
     //delete node
          delete(n);
}
```

Copy

Forward Traversal

Start with the front node and visit all the nodes untill the node becomes NULL.

```
void Doubly_Linked_List :: forward_traverse()
{
     node *trav;
```

```
      trav = front;

      while(trav != NULL)

      {

            cout<<trav->data<<endl;

            trav = trav->next;

      }

}
```

Copy

Backward Traversal

Start with the end node and visit all the nodes until the node becomes NULL.

```
void Doubly_Linked_List :: backward_traverse()

{

      node *trav;

      trav = end;

      while(trav != NULL)

      {

            cout<<trav->data<<endl;

            trav = trav->prev;

      }

}
```

Copy

Now that we have learned about the Doubly Linked List, you can also check out the other types of Linked list as well:

- Linear Linked List
- Circular Linked List

Introduction To Binary Trees

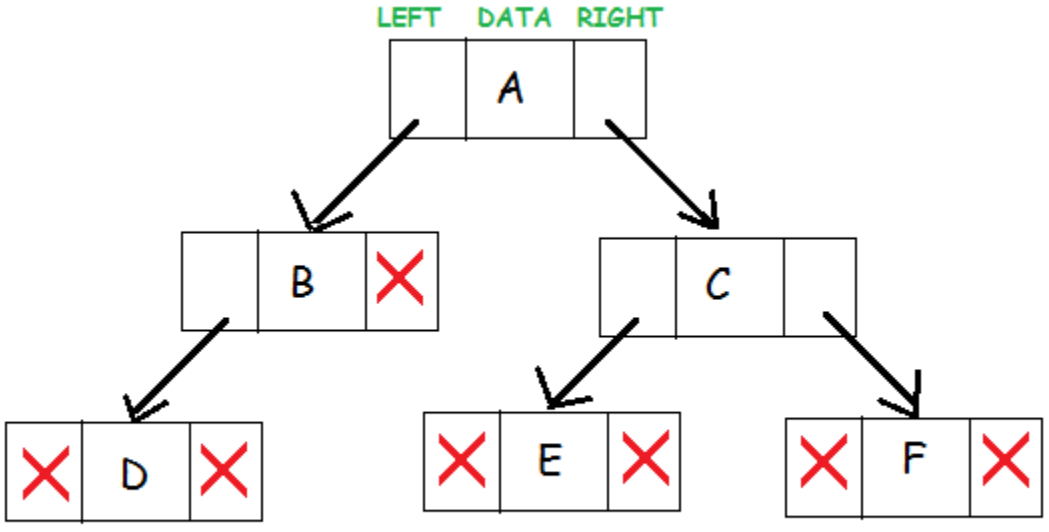
A binary tree is a hierarchical data structure in which each node has at most two children generally referred as left child and right child.

Each node contains three components:

1. Pointer to left subtree
2. Pointer to right subtree
3. Data element

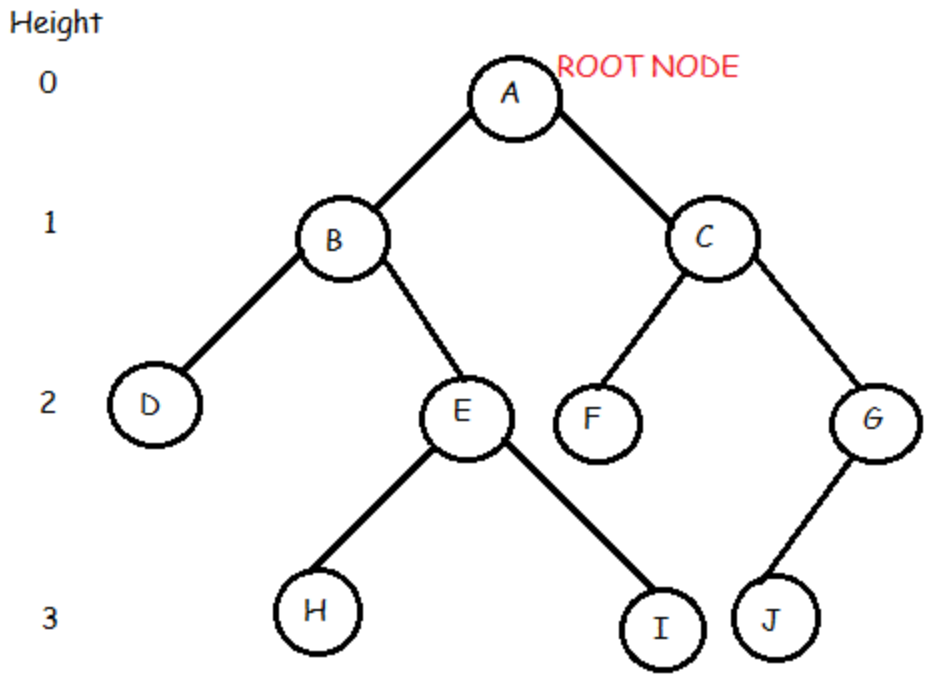
The topmost node in the tree is called the root. An empty tree is represented by **NULL** pointer.

A representation of binary tree is shown:



Binary Tree: Common Terminologies

- **Root:** Topmost node in a tree.
- **Parent:** Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent.
- **Child:** A node directly connected to another node when moving away from the root.
- **Leaf/External node:** Node with no children.
- **Internal node:** Node with atleast one children.
- **Depth of a node:** Number of edges from root to the node.
- **Height of a node:** Number of edges from the node to the deepest leaf. Height of the tree is the height of the root.



In the above binary tree we see that root node is **A**. The tree has 10 nodes with 5 internal nodes, i.e, **A,B,C,E,G** and 5 external nodes, i.e, **D,F,H,I,J**. The height of the tree is 3. **B** is the parent of **D** and **E** while **D** and **E** are children of **B**.

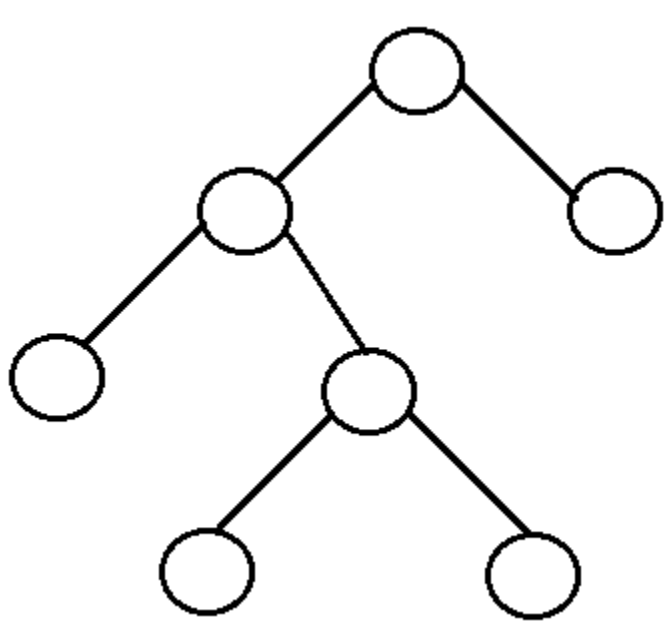
Advantages of Trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data.
- Trees are used to represent hierarchies.
- Trees provide an efficient insertion and searching.
- Trees are very flexible data, allowing to move subtrees around with minumum effort.

Types of Binary Trees (Based on Structure)

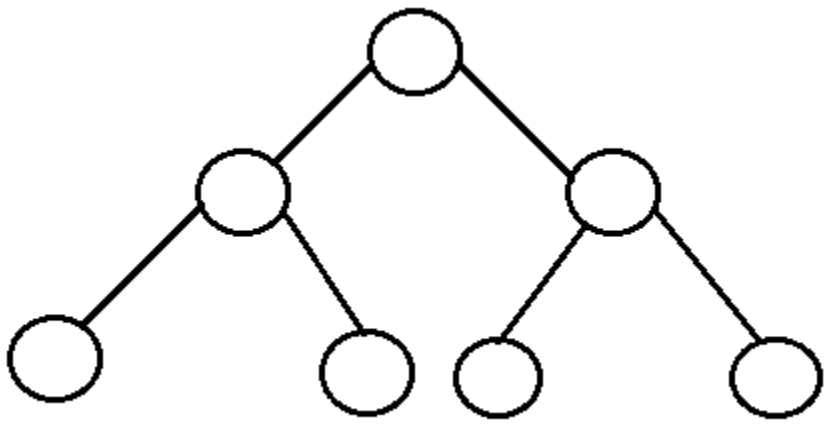
- **Rooted binary tree:** It has a root node and every node has atmost two children.
- **Full binary tree:** It is a tree in which every node in the tree has either 0 or 2 children.



  - The number of nodes, *n*, in a full binary tree is atleast n = 2h – 1, and atmost $n = 2^{h+1} - 1$, where *h* is the height of the tree.

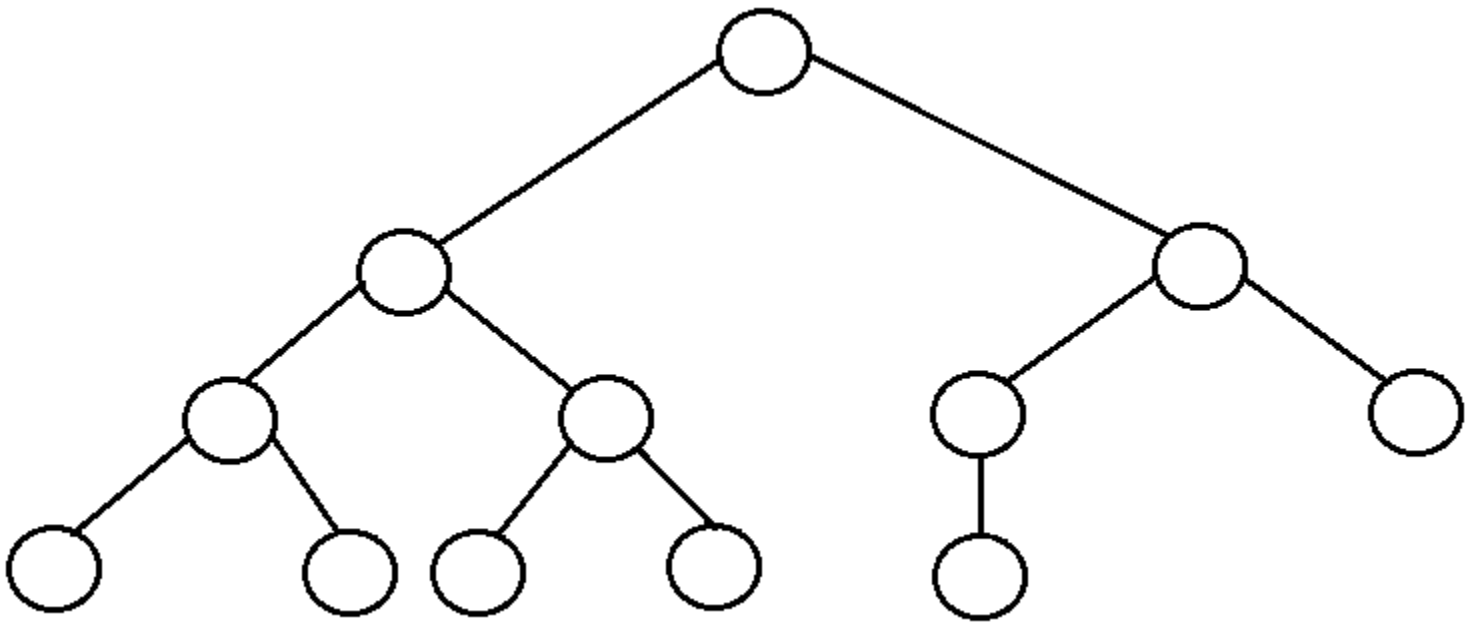
    - The number of leaf nodes *l*, in a full binary tree is number, *L* of internal nodes + 1, i.e, *l = L+1*.

- **Perfect binary tree:** It is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



    - A perfect binary tree with *l* leaves has *n = 2l-1* nodes.
    - In perfect full binary tree, *l = 2h* and *n = 2h+1 - 1* where, *n* is number of nodes, *h* is height of tree and *l* is number of leaf nodes

- **Complete binary tree:** It is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

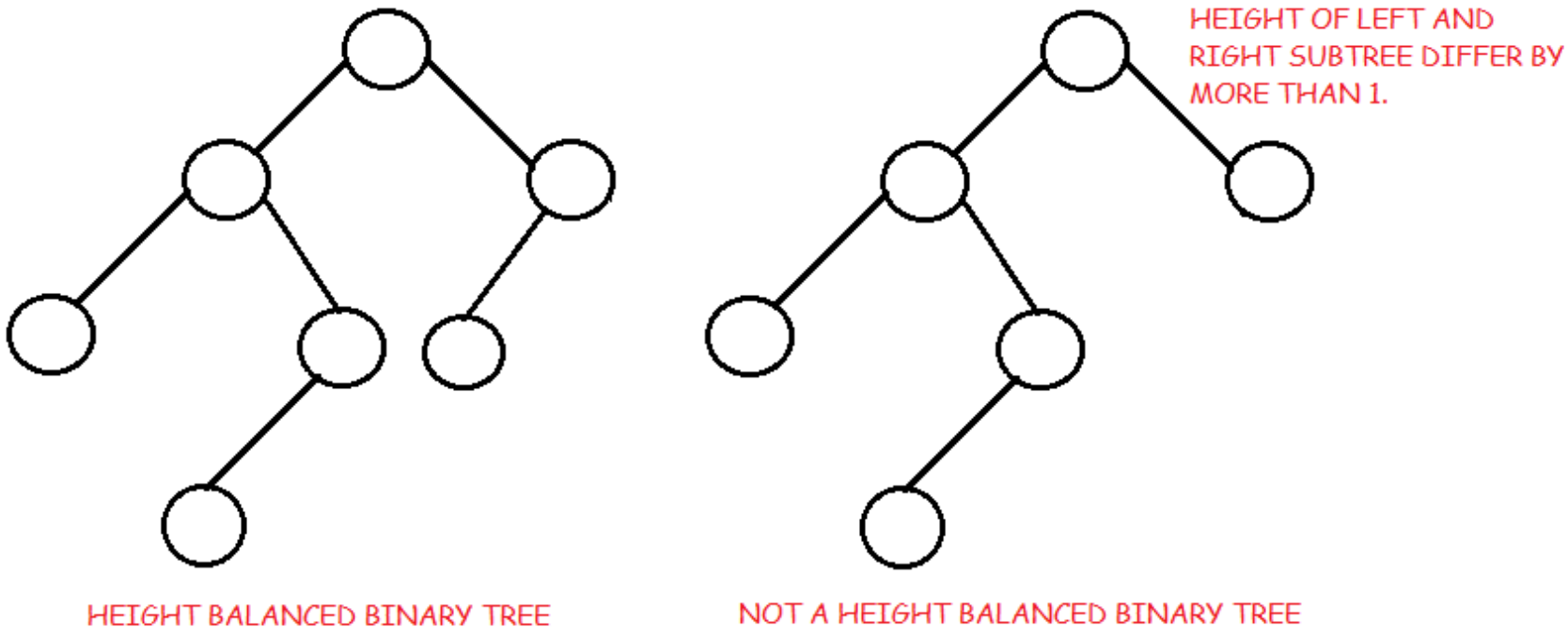


    - The number of internal nodes in a complete binary tree of n nodes is *floor(n/2)*.

- **Balanced binary tree:** A binary tree is height balanced if it satisfies the following constraints:

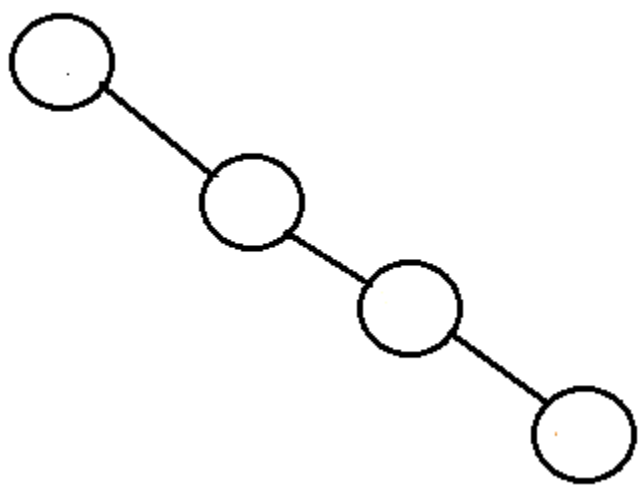
    1. The left and right subtrees' heights differ by at most one, AND
    2. The left subtree is balanced, AND
    3. The right subtree is balanced

    An empty tree is height balanced.



- The height of a balanced binary tree is O(*Log n*) where *n* is number of nodes.

- **Degenarate tree:** It is a tree is where each parent node has only one child node. It behaves like a linked list.



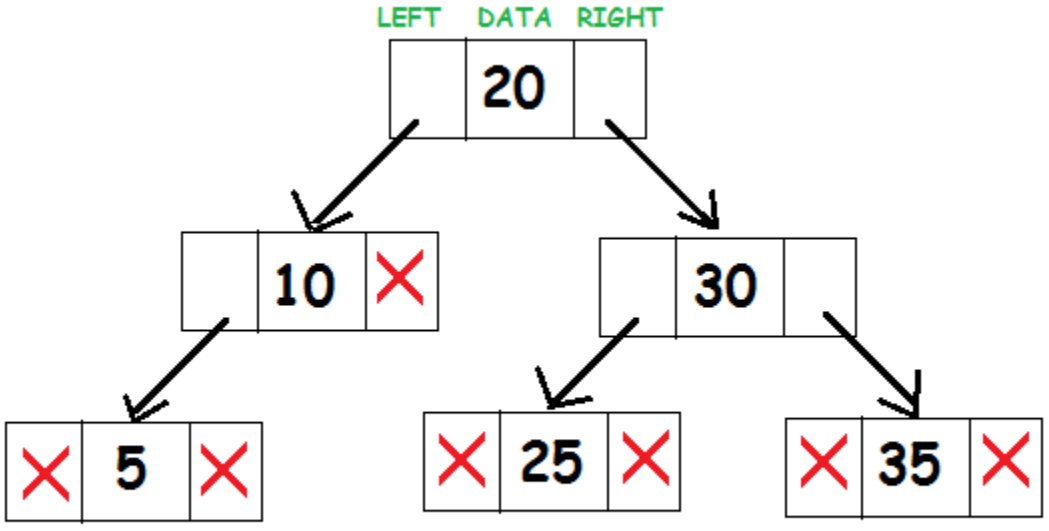
Binary Search Tree

A binary search tree is a useful data structure for fast addition and removal of data.

It is composed of nodes, which stores data and also links to upto two other child nodes. It is the relationship between the leaves linked to and the linking leaf, also known as the parent node, which makes the binary tree such an efficient data structure.

For a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be less than the data of the root. The data of all the nodes in the right subtree of the root node should be greater than equal to the data of the root. As a result, the leaves on the farthest left of the tree have the lowest values, whereas the leaves on the right of the tree have the greatest values.

A representation of binary search tree looks like the following:



Consider the root node 20. All elements to the left of subtree(10, 5) are less than 20 and all elements to the right of subtree(25, 30, 35) are greater than 20.

Implementation of BST

First, define a struct as `tree_node`. It will store the data and pointers to left and right subtree.

```
struct tree_node

{

     int data;

     tree_node *left, *right;

};
```

Copy

The node itself is very similar to the node in a linked list. A basic knowledge of the code for a linked list will be very helpful in understanding the techniques of binary trees.

It is most logical to create a binary search tree class to encapsulate the workings of the tree into a single area, and also making it reusable. The class will contain functions to insert data into the tree, search if the data is present and methods for traversing the tree.

```
class BST

{

     tree_node *root;

     void insert(tree_node* , int );

     bool search(int , tree_node* );

     void inorder(tree_node* );

     void preorder(tree_node* );

     void postorder(tree_node* );



     public:

     BST()

     {

          root = NULL;

     }

     void insert(int );

     bool search(int key);

    void inorder();

    void preorder();

    void postorder();

};
```

Copy

It is necessary to initialize root to NULL for the later functions to be able to recognize that it does not exist.

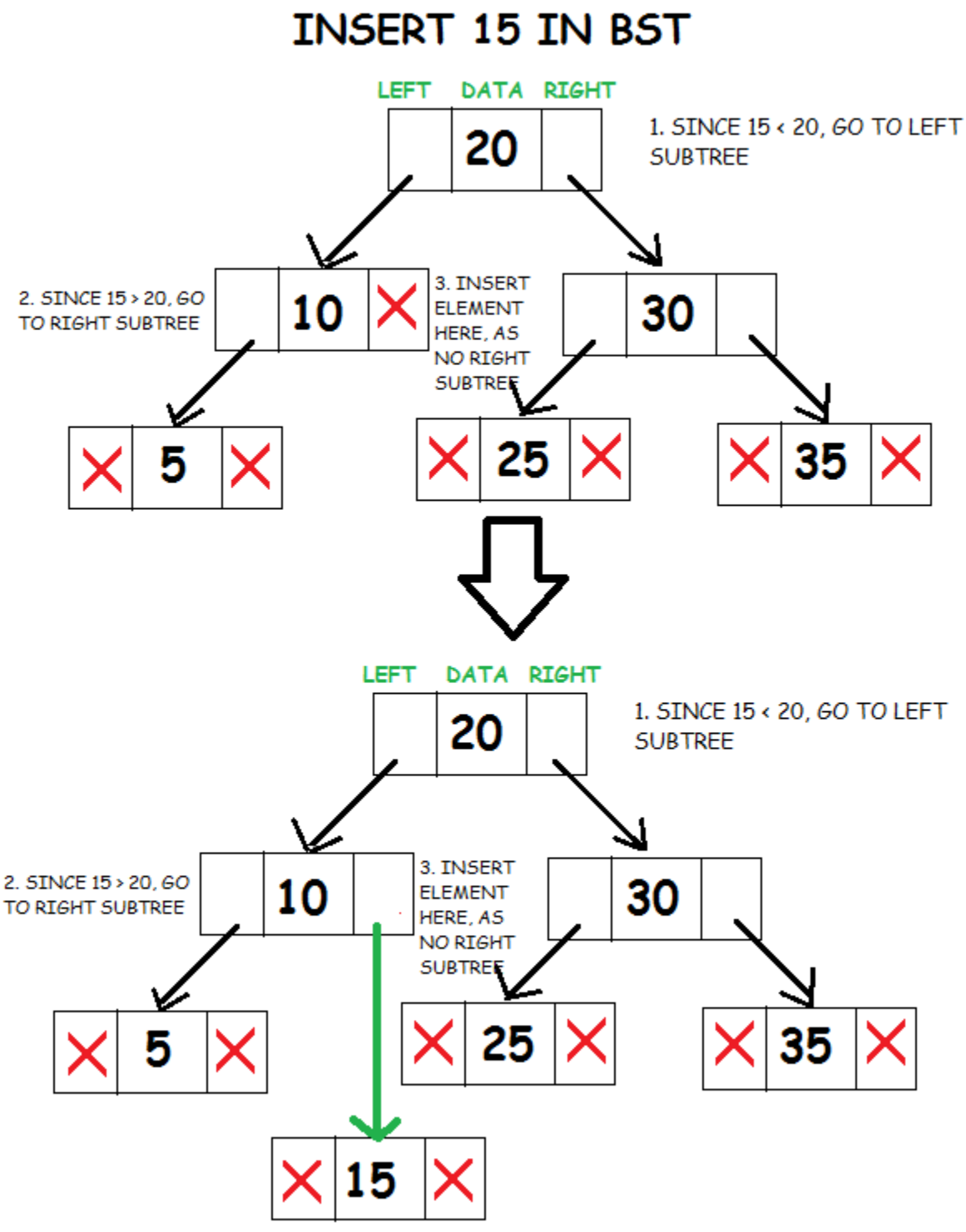
All the public members of the class are designed to allow the user of the class to use the class without dealing with the underlying design. The functions which will be called recursively are the ones which are private, allowing them to travel down the tree.

Insertion in a BST

To insert data into a binary tree involves a function searching for an unused node in the proper position in the tree in which to insert the key value. The insert function is generally a recursive function that continues moving down the levels of a binary tree until there is an unused leaf in a position which follows the following rules of placing nodes.

- Compare data of the root node and element to be inserted.

- If the data of the root node is greater, and if a left subtree exists, then repeat step 1 with **root = root of left subtree**. Else,
- Insert element as left child of current root.
- If the data of the root node is greater, and if a right subtree exists, then repeat step 1 with **root = root of right subtree**.
- Else, insert element as right child of current root.




```
void BST :: insert(tree_node *node, int d)

{

     // element to be inserted is lesser than node's data

     if(d < node->data)

     {

          // if left subtree is present

          if(node->left != NULL)

               insert(node->left, d);
```

```
            // create new node
            else
            {
                  node->left = new tree_node;
                  node->left->data = d;
                  node->left->left = NULL;
                  node->left->right = NULL;
            }
      }

      // element to be inserted is greater than node's data
      else if(d >= node->data)
      {
            // if left subtree is present
            if(node->right != NULL)
                  insert(node->right, d);

            // create new node
            else
            {
                  node->right = new tree_node;
                  node->right->data = d;
                  node->right->left = NULL;
                  node->right->right = NULL;
            }
      }

}
```

Copy

Since the root node is a private member, we also write a public member function which is available to non-members of the class. It calls the private recursive function to insert an element and also takes care of the case when root node is NULL.

```
void BST::insert(int d)

{

     if(root!=NULL)

                insert(root, d);

     else

    {

        root = new tree_node;

        root->data = d;

        root->left = NULL;

        root->right = NULL;

    }

}
```

Copy

Searching in a BST

The search function works in a similar fashion as insert. It will check if the key value of the current node is the value to be searched. If not, it should check to see if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node, or if it is greater than the value of the node, it should be recursively called on the right child node.

- Compare data of the root node and the value to be searched.
- If the data of the root node is greater, and if a left subtree exists, then repeat step 1 with **root = root of left subtree**. Else,
- If the data of the root node is greater, and if a right subtree exists, then repeat step 1 with **root = root of right subtree**. Else,
- If the value to be searched is equal to the data of root node, return true.
- Else, return false.

```
bool BST::search(int d, tree_node *node)

{

     bool ans = false;



     // node is not present
```

```
     if(node == NULL)
          return false;


     // Node’s data is equal to value searched
    if(d == node->data)
          return true;


     // Node’s data is greater than value searched
    else if(d < node->data)
          ans = search(d, node->left);


      // Node’s data is lesser than value searched
    else
          ans = search(d, node->right);


    return ans;
}
```

Copy

Since the root node is a private member, we also write a public member function which is available to non-members of the class. It calls the private recursive function to search an element and also takes care of the case when root node is NULL.

```
bool BST::search(int d)
{
     if(root ==  NULL)
          return false;
     else
          return  search(d, root);
}
```

Copy

Traversing in a BST

There are mainly three types of tree traversals:

1. Pre-order Traversal:
In this technique, we do the following :

- Process data of root node.
- First, traverse left subtree completely.
- Then, traverse right subtree.

```
void BST :: preorder(tree_node *node)

{

     if(node !=  NULL)

    {

           cout<<node->data<<endl;

           preorder(node->left);

           preorder(node->right);

    }

}
```

Copy

2. Post-order Traversal
In this traversal technique we do the following:

- First traverse left subtree completely.
- Then, traverse right subtree completely.
- Then, process data of node.

```
void BST :: postorder(tree_node *node)

{

     if(node !=  NULL)

     {

        postorder(node->left);

        postorder(node->right);

         cout<<node->data<<endl;

     }

}
```

Copy

3. In-order Traversal

In in-order traversal, we do the following:

- First process left subtree.
- Then, process current root node.
- Then, process right subtree.

```
void BST :: inorder(tree_node *node)

{

     if(node !=  NULL)

     {

        inorder(node->left);

         cout<<node->data<<endl;

        inorder(node->right);

     }

}
```

Copy

The in-order traversal of a binary search tree gives a sorted ordering of the data elements that are present in the binary search tree. This is an important property of a binary search tree.

Since the root node is a private member, we also write public member functions which is available to non-members of the class. It calls the private recursive function to traverse the tree and also takes care of the case when root node is NULL.

```
void BST :: preorder()

{

     if(root ==  NULL)

          cout<<"TREE IS  EMPTY\n";

     else

          preorder(root);

}


void BST :: postorder()

{

     if(root ==  NULL)

          cout<<"TREE IS  EMPTY\n";
```

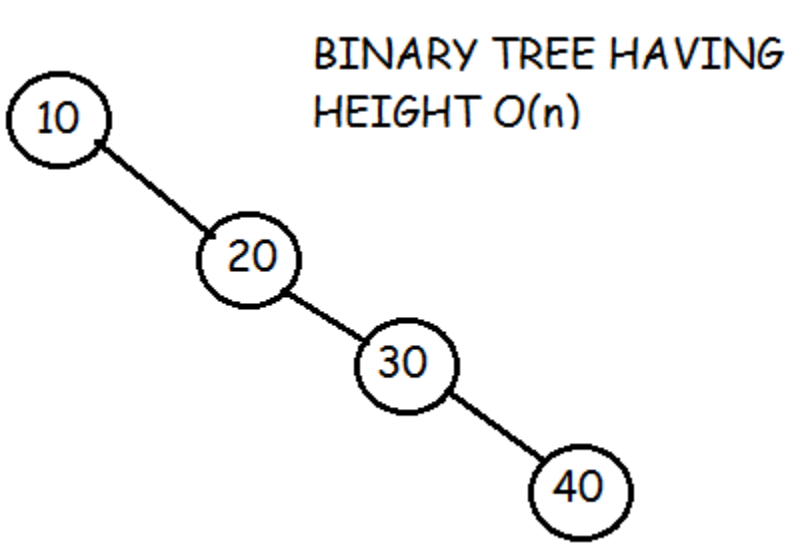
```
     else
         postorder(root);
}


void BST :: inorder()
{
     if(root == NULL)
          cout<<"TREE IS EMPTY\n";
     else
          inorder(root);
}
```

Copy

Complexity Analysis

ALGORITHM	AVERAGE CASE	WORST CASE
Space	O(n)	O(n)
Insert	O(log n)	O(n)
Search	O(log n)	O(n)
Traverse	O(n)	O(n)

The time complexity of search and insert rely on the height of the tree. On average, binary search trees with n nodes have **O(log n)** height. However in the worst case the tree can have a height of **O(n)** when the unbalanced tree resembles a linked list. For example in this case :



Traversal requires **O(n)** time, since every node must be visited.