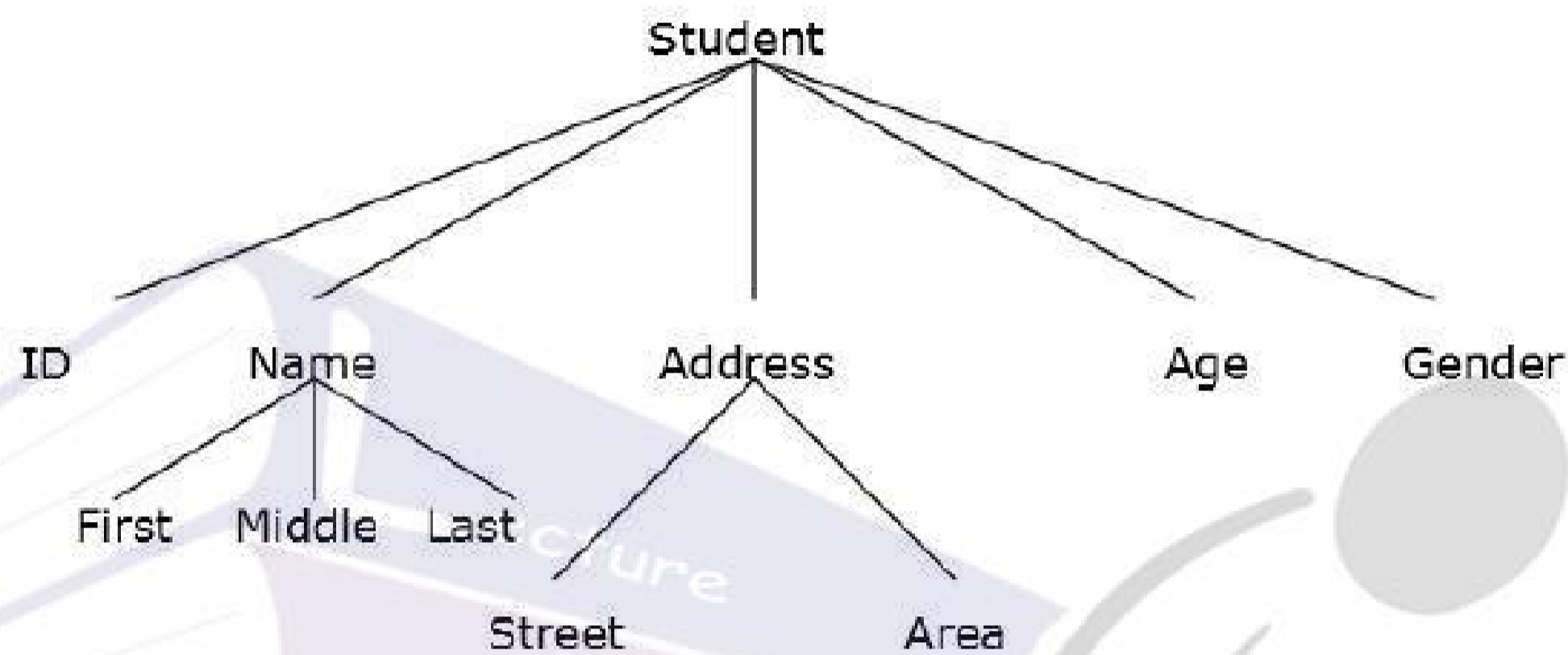


Introduction to Data Structures

Data:

Data are simply collection of facts and figures. Data are values or set of values. A data item refers to a single unit of values. Data items that are divided into sub items are group items; those that are not are called elementary items. For example, a student's name may be divided into three sub items – [first name, middle name and last name] but the ID of a student would normally be treated as a single item.



In the above example (ID, Age, Gender, First, Middle, Last, Street, Area) are elementary data items, whereas (Name, Address) are group data items.

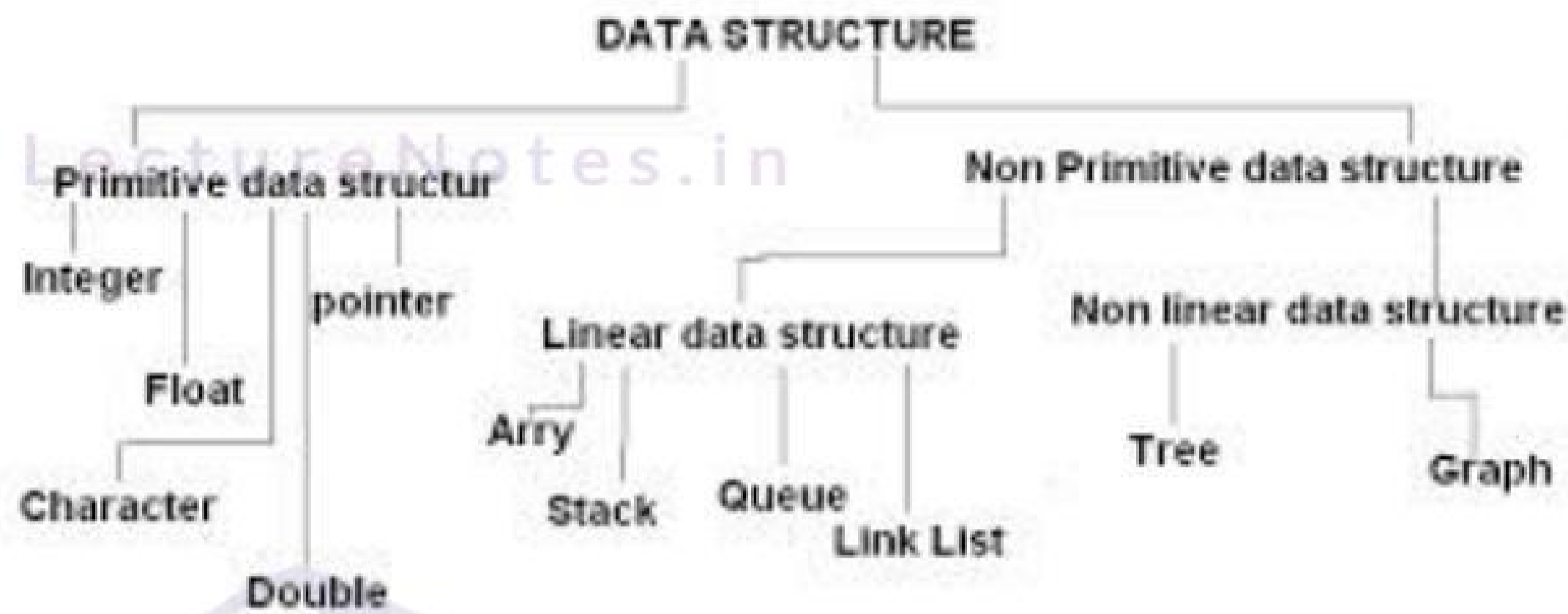
Abstract Data Type(ADT)

Abstract data types or ADTs are a mathematical specification of a set of data and the set of operations that can be performed on the data.

Data Structure

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage.

Classification of data structure



Data structure are Divided into two main categories.

1. **Primitive data structure**
2. **Non-primitive data structure**

Primitive data structure:

The primitive data structure can be manipulated or operated by the machine instruction. There is general, have different representations an different computers.

For example- the integers, floating-point numbers, pointers, string constants, characters etc. are some of the different primitive data structure in java language, the different primitive data structure are defined using the data type such as

- Int
- Char
- Float
- Double
- pointer

etc.

Non-primitive data structure

The non-primitive data structures are data structure that cannot be manipulated or operated directly by the machine instructions. These are more sophisticated data structures.

These are derived from the primitive data structure.

For example-Arrays, structure, stack, queues, linked list etc.

The Non-Primitive data structure is classified into two categories

- **Linear** Data Structure and
- **Non Linear** Data Structure.

Linear data structure:

Collection of nodes which are logically adjacent in which logical adjacency is maintained by pointers

(or)

Linear data structures can be constructed as a continuous arrangement of data elements in the memory. It can be constructed by using array data type. In the linear Data Structures the relationship of adjacency is maintained between the Data elements.

Operations applied on linear data structure:

The following list of operations applied on linear data structures

1. Add an element
2. Delete an element
3. Traverse
4. Sort the list of elements
5. Search for a data element

By applying one or more functionalities we can create different types of linear data structures.

For example Stack, Queue, Tables, List, and Linked Lists.

Non-linear data structure:

Non-linear data structure can be constructed as a collection of randomly distributed set of data item joined together by using a special pointer (tag). In non-linear Data structure the relationship of adjacency is not maintained between the Data items.

Operations applied on non-linear data structures:

The following list of operations applied on non-linear data structures.

1. Add elements
2. Delete elements
3. Display the elements
4. Sort the list of elements
5. Search for a data element

By applying one or more these functionalities we can create different types of data structures.

For example Tree, Decision tree, Graph and Forest.

Array ADT

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element** – Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.
- **Base address**-the starting address of the array
- **Size and type**

Array Representation

Arrays can be declared in various ways in different languages. For example, let's take Java array declaration.



LectureNotes.in

As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 12 which means it can store 12 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 7.

Operations on array

1. **Traversing:** means to visit all the elements of the array in an operation is called traversing.
2. **Insertion:** means to put values into an array
3. **Deletion / Remove:** to delete a value from an array.
4. **Sorting:** Re-arrangement of values in an array in a specific order (Ascending / Descending) is called sorting.
5. **Searching:** The process of finding the location of a particular element in an array is called searching. There are two popular searching techniques/mechanisms : Linear search and binary search

a. Traversing in Linear Array:

It means processing or visiting each element in the array exactly once; Let 'A' is an array stored in the computer's memory. If we want to display the contents of 'A', it has to be traversed i.e. by accessing and processing each element of 'A' exactly once.

The following is a simple java program to traverse elements of an array.

```
/* Traverse_Array.java */
import java.util.Scanner;
class Traverse_Array
```

```
{
public static void main(String[] args)
{
    int n;
    Scanner s = new Scanner(System.in);
    System.out.print("Enter no. of elements you want in array:");
    n = s.nextInt();
    int a[] = new int[n];
    System.out.println("Enter all the elements:");
    for(int i = 0; i < n; i++)
    {
        a[i] = s.nextInt();
    }
    System.out.print("The array elements are:");
    for(int i = 0; i < n; i++)
    {
        System.out.print(a[i] + "\t");
    }
}
}
```

Output:

```

C:\Program Files (x86)\EditPlus 2\launcher.exe
Enter no. of elements you want in array:5
Enter all the elements:
10
20
30
40
50
The array elements are:10 20 30 40 50
Press any key to continue...

```

b. Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

10	20	30	40	50	60	70	80	90	100
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Original Array

Now we need to insert an element at 4th position.

10	20	30	40		50	60	70	80	90	100
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

➔ Elements should be moved to right hand side

10	20	30	40	45	50	60	70	80	90	100
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]

Array after insertion

The following program illustrates the insertion of an element in an array within the specified position.

```

import java.util.Scanner;

class Insert_Array

```

```
{
public static void main(String[] args)
{
    int n, pos, x;
    Scanner s = new Scanner(System.in);
    System.out.print("Enter no. of elements you want in array:");
    n = s.nextInt();
    int a[] = new int[n+1];
    System.out.println("Enter all the elements:");
    for(int i = 0; i < n; i++)
    {
        a[i] = s.nextInt();
    }
    System.out.print("Enter the position where you want to insert element:");
    pos = s.nextInt();
    System.out.print("Enter the element you want to insert:");
    x = s.nextInt();
    for(int i = (n-1); i >= (pos-1); i--)
    {
        a[i+1] = a[i];
    }
    a[pos-1] = x;
    System.out.print("After inserting the array elements:");
    for(int i = 0; i <= n; i++)
    {
```



```

        System.out.print(a[i] + "\t");
    }
}
}

```

Output LectureNotes.in

```

Command Prompt
E:\Naresh\DS Programs\Arrays>javac Insert_Array.java
E:\Naresh\DS Programs\Arrays>java Insert_Array
Enter no. of elements you want in array:4
Enter all the elements:
10
20
30
40
Enter the position where you want to insert element:2
Enter the element you want to insert:15
After inserting the array elements:10 15 20 30 40
E:\Naresh\DS Programs\Arrays>

```

c. Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

The following program illustrates the deletion of an element from an array.

```
/* Java Program Example - delete Element from Array */
```

```

import java.util.Scanner;

class Delete_Array

{

    public static void main(String args[])

    {

        int size, i, del, count=0;

```

```
int arr[];

Scanner scan = new Scanner(System.in);

System.out.print("Enter Array Size : ");

size = scan.nextInt();

arr = new int[size];

System.out.print("Enter Array Elements : ");

for(i=0; i<size; i++)

{

arr[i] = scan.nextInt();

}

System.out.print("Enter Element to be Delete : ");

del = scan.nextInt();

for(i=0; i<size; i++)

{

if(arr[i] == del)

{

for(int j=i; j<(size-1); j++)

{

arr[j] = arr[j+1];

}

count++;

break;

}

}

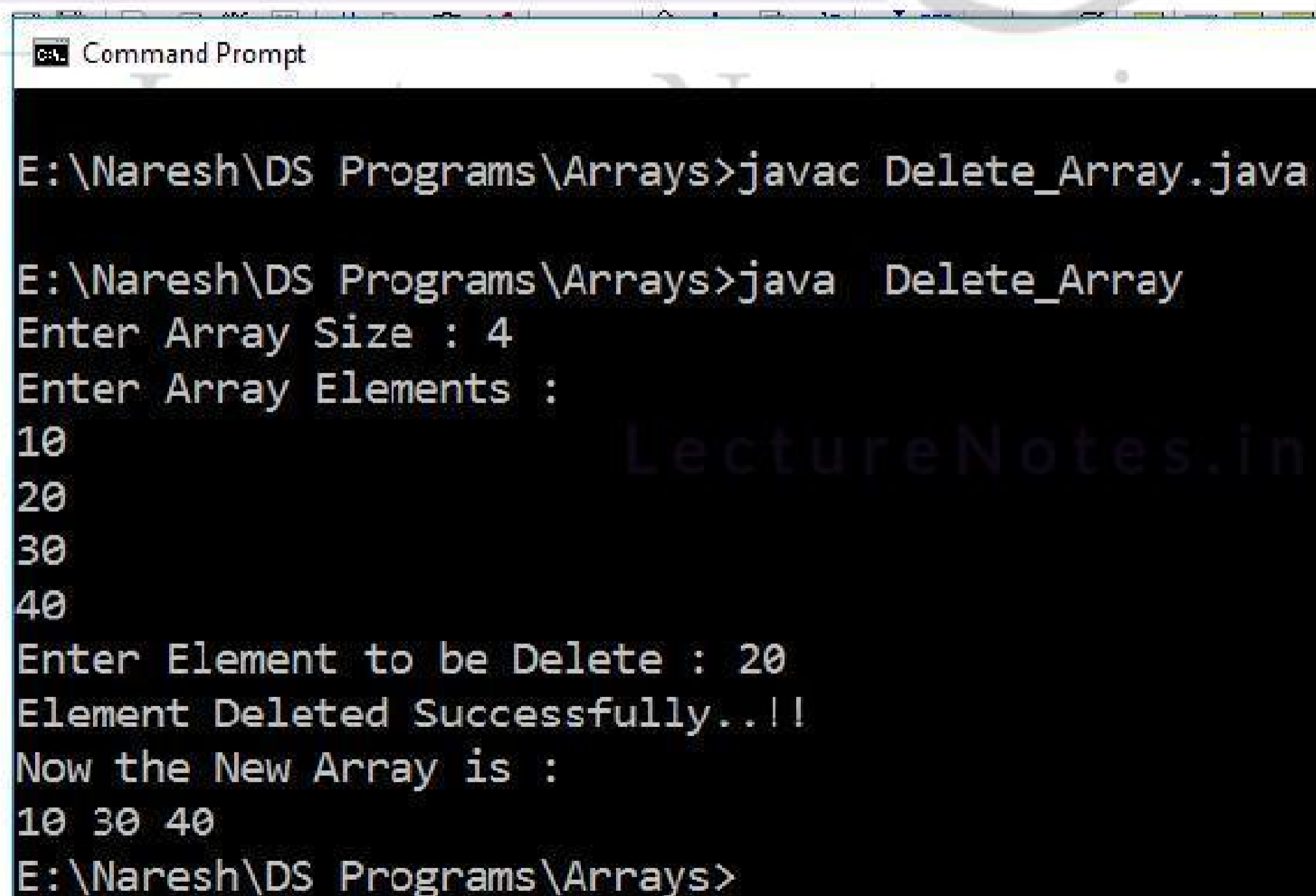
if(count==0)
```

```

    {
        System.out.print("Element Not Found..!!");
    }
else
    {
        System.out.print("Element Deleted Successfully..!!");
        System.out.print("\nNow the New Array is :\n");
        for(i=0; i<(size-1); i++)
        {
            System.out.print(arr[i]+ " ");
        }
    }
}
}

```

Output:



```

Command Prompt
E:\Naresh\DS Programs\Arrays>javac Delete_Array.java
E:\Naresh\DS Programs\Arrays>java Delete_Array
Enter Array Size : 4
Enter Array Elements :
10
20
30
40
Enter Element to be Delete : 20
Element Deleted Successfully..!!
Now the New Array is :
10 30 40
E:\Naresh\DS Programs\Arrays>

```

Searching:

```
import java.util.Scanner;
```

```
class T_Array
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        int n,x,i=0,count=0;
```

```
        Scanner s = new Scanner(System.in);
```

```
        System.out.print("Enter no. of elements you want in array:");
```

```
        n = s.nextInt();
```

```
        System.out.print("enter the element to search");
```

```
        x = s.nextInt();
```

```
        int a[] = new int[n];
```

```
        System.out.println("Enter all the elements:");
```

```
        for( i = 0; i < n; i++)
```

```
        {
```

```
            a[i] = s.nextInt();
```

```
        }
```

```
        for(i = 0; i < n; i++)
```

```
        {
```

```
            if(a[i]==x)
```

```
                count++;
```

```
}  
if(count==1)  
    System.out.print("The element exist in the array ");  
    else  
        System.out.print("The element not exist in the array ");  
}  
}
```

Search operation is used to find whether the element is present in the array or not.

Output:

```
C:\Users\meenakshi\Desktop>javac T_Array.java
```

```
C:\Users\meenakshi\Desktop>java T_Array
```

```
Enter no. of elements you want in array:5
```

```
enter the element to search8
```

```
Enter all the elements:
```

```
2
```

```
5
```

```
9
```

```
4
```

```
7
```

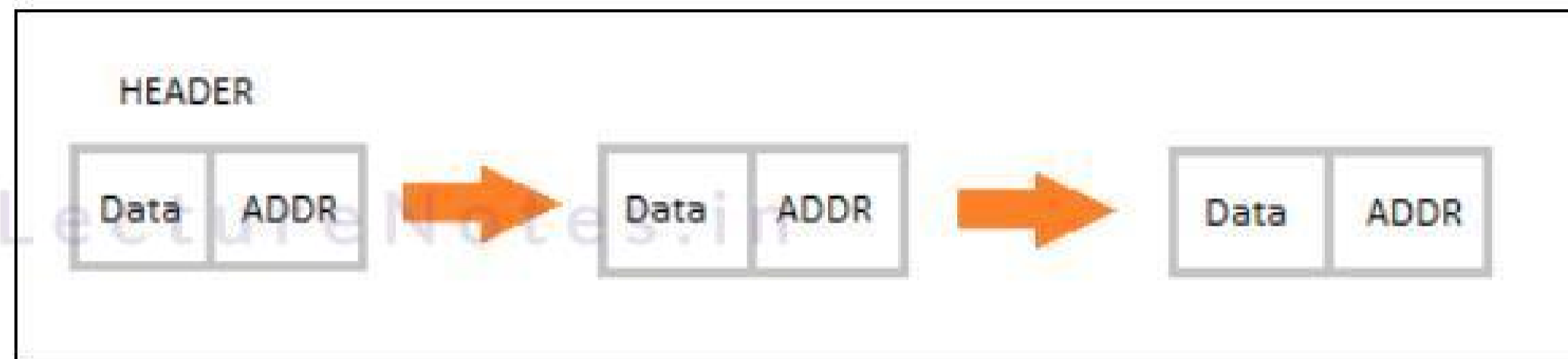
```
The element not exist in the array
```

Linked Lists

Introduction to Linked Lists

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs, stacks and queues.

The general structure of linked list is shown in below.



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

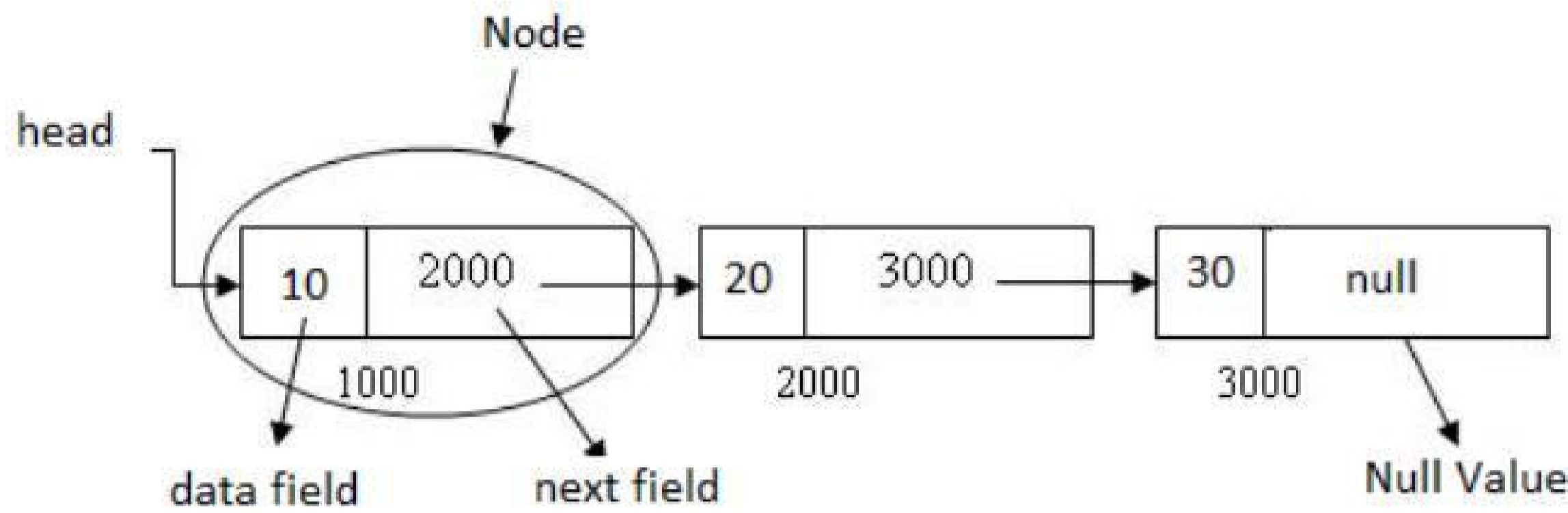
- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

Singly Linked List:

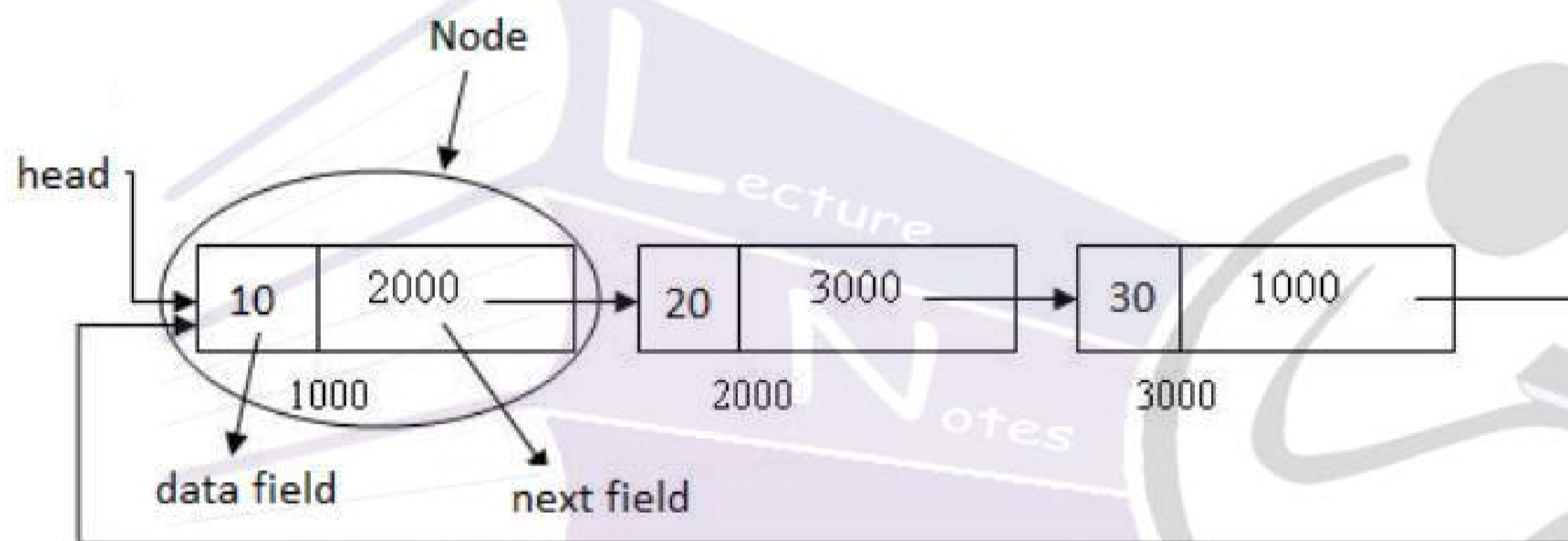
- It is linear collection of data elements which are called 'Nodes'.
- The elements may or may not be stored in consecutive memory locations. So pointers are used maintain linear order.
- Each node is divided into two parts.

- The NEXT Field of last node contains NULL Value which indicates that it is the end of linked list.
- It is shown below:



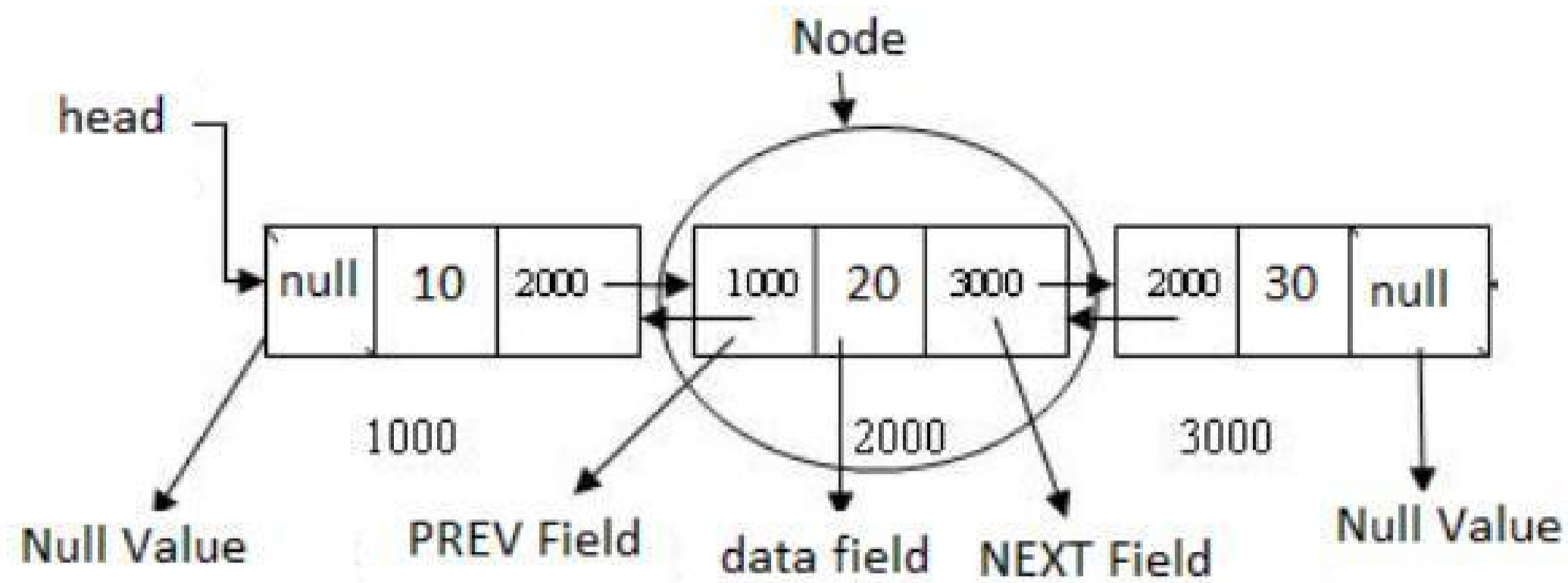
2. Circular Linked List

- It is singly linked list in which the next part of the last node contains address of the first node instead of null. Thus it makes the single linked list as circular singly linked list.
- It is shown below:



3. Doubly Linked List or Two-Way Linked List or Two-Way Chain:-

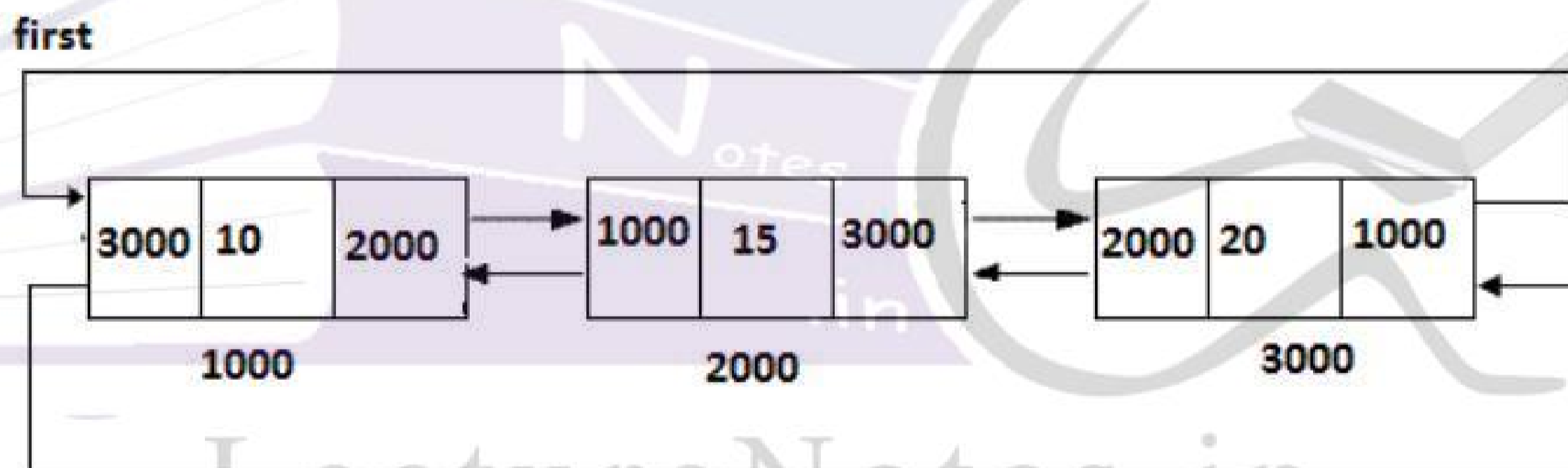
- In it each node is divided into three parts:
 - The first part is PREV part. It is previous pointer field. It contains the address of the node which is before the current node.
 - The second part is the DATA part. It contains the information of the element.
 - The third part is NEXT part. It is next pointer field. It contains the address of the node which is after the current node.
- The HEAD contains the address of the first node in linked list.
- The PREV field of first node and the NEXT field of last node contain NULL value. This shows the end of list on both sides.
- This list can be traversed in both directions that is forward and backward.
- It is shown below:



LectureNotes.in

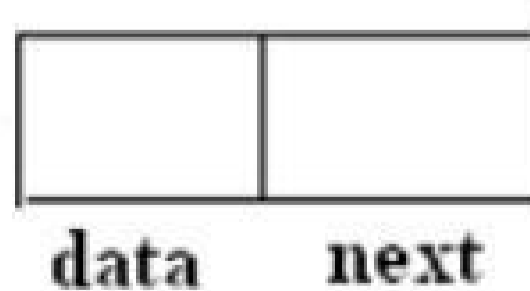
4. Circular Double Linked List

- It is a type of doubled linked list in which the next part of the last node contains address of the first node instead of null and the previous part of the first node contains the address of the last node instead of null. Thus it makes the doubled linked list as circular doubled linked list.
- It is shown below:



Single Linked List

- It is a Linear Data Structure.
- In which each and every element is called as a node and each node contains two parts.
- The first part of the node is called as data part and the second part is address part (next part), this is shown in below.



- The data part contains the actual data element and the address part contains the address part of its next node.
- The general structure of a Single Linked List is as follows.



We can identify the first node of a Single Linked List by using the head pointer. In this, the head pointer always points to the first node of a Single Linked List. If there are no elements in a Single Linked List the head pointer points to a null value.

We can also identify the last node of a Single Linked List by using its address part i.e., the address part of the last node will always be a null value.

In general, we can able to perform the following operations on a Single Linked List.

- Insertion of a node at beginning.
- Insertion of a node at end.
- Insertion of a node at middle.
- Deletion of a node.
- Display.

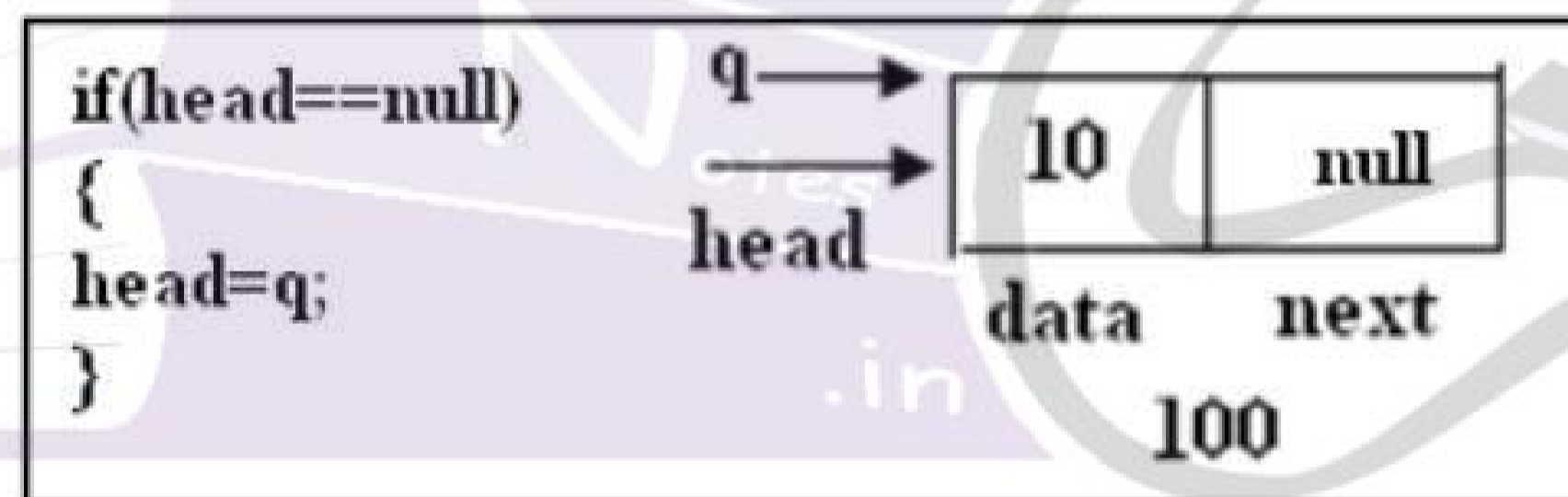
These are explained below.....

Assume that the newly inserted node is q.

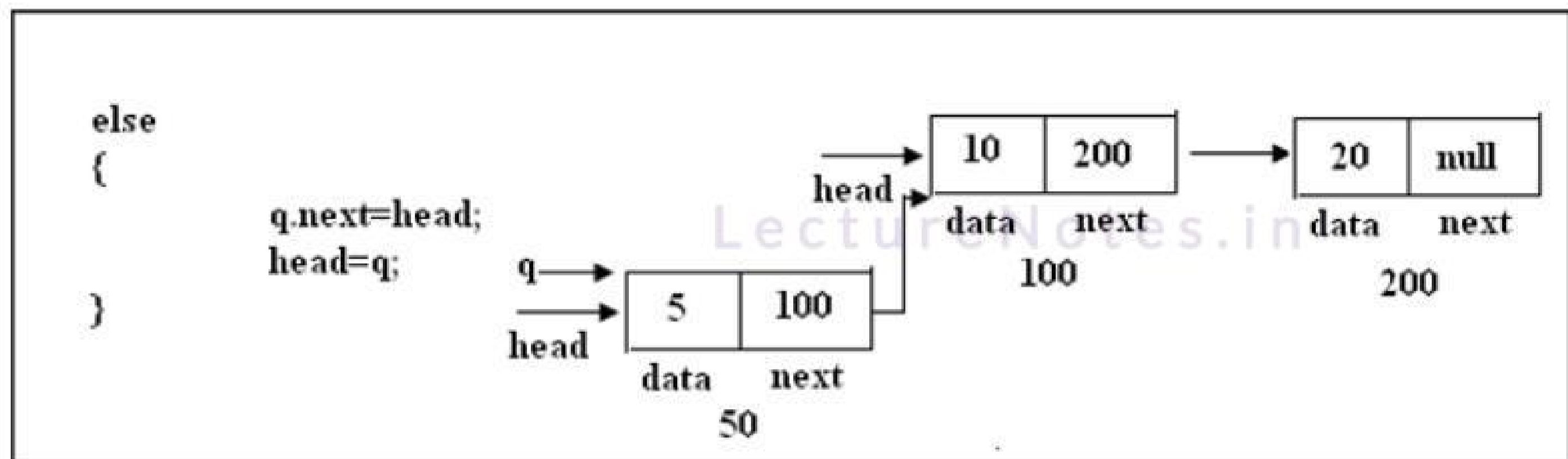
node q=new node();

Inserting a node at beginning

- If the Linked List is empty, then the inserted element will be the starting and the last element. We can check the following condition to know either the Linked List is empty or not.

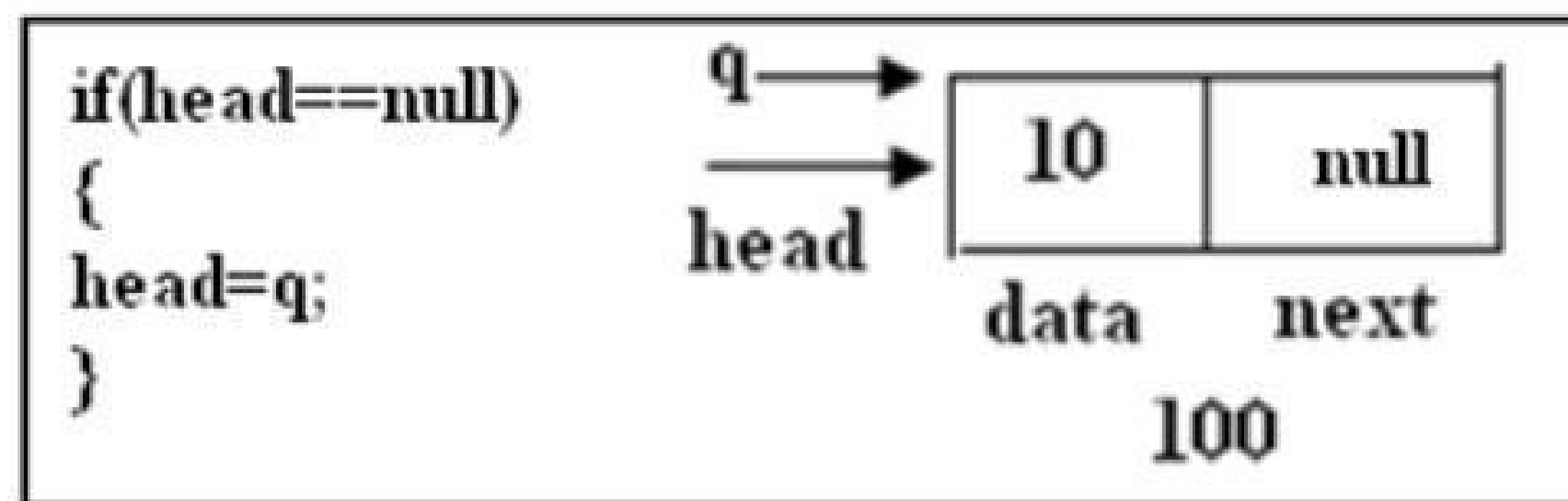


Otherwise we can insert the element at beginning of the Linked List as follows....

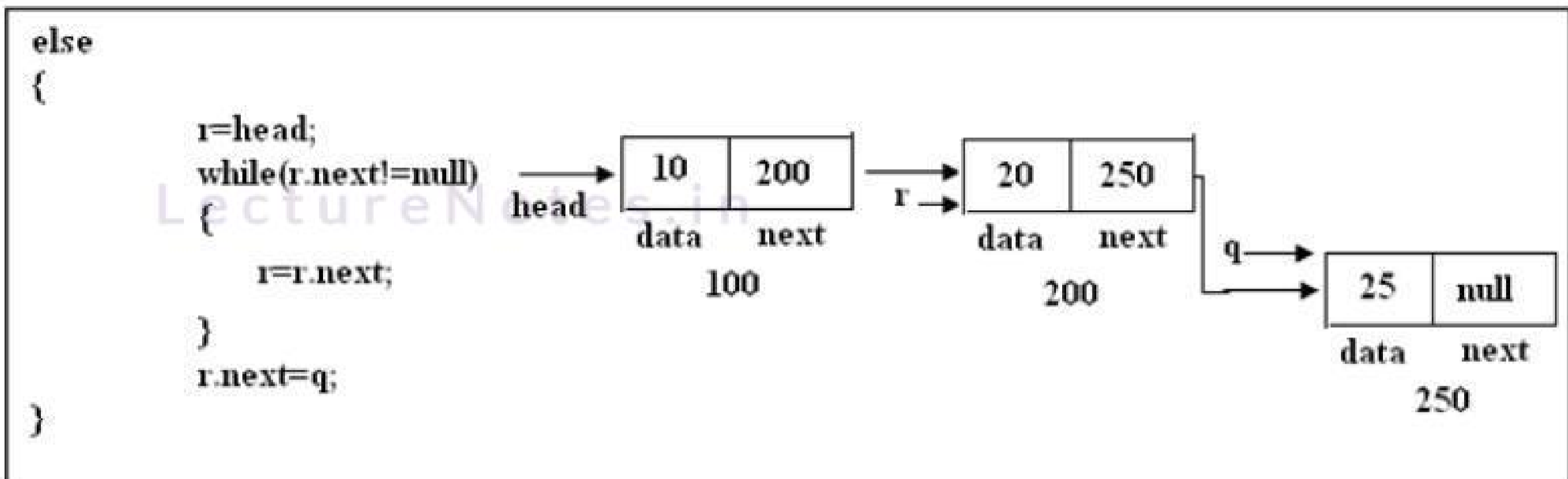


Insertion of a node at end

- If the Linked List is empty, then the inserted element will be the starting and the last element. We can check the following condition to know either the Linked List is empty or not.



Otherwise we can move to the last node and we can insert the newly created node at end.



Insertion of a node at Middle

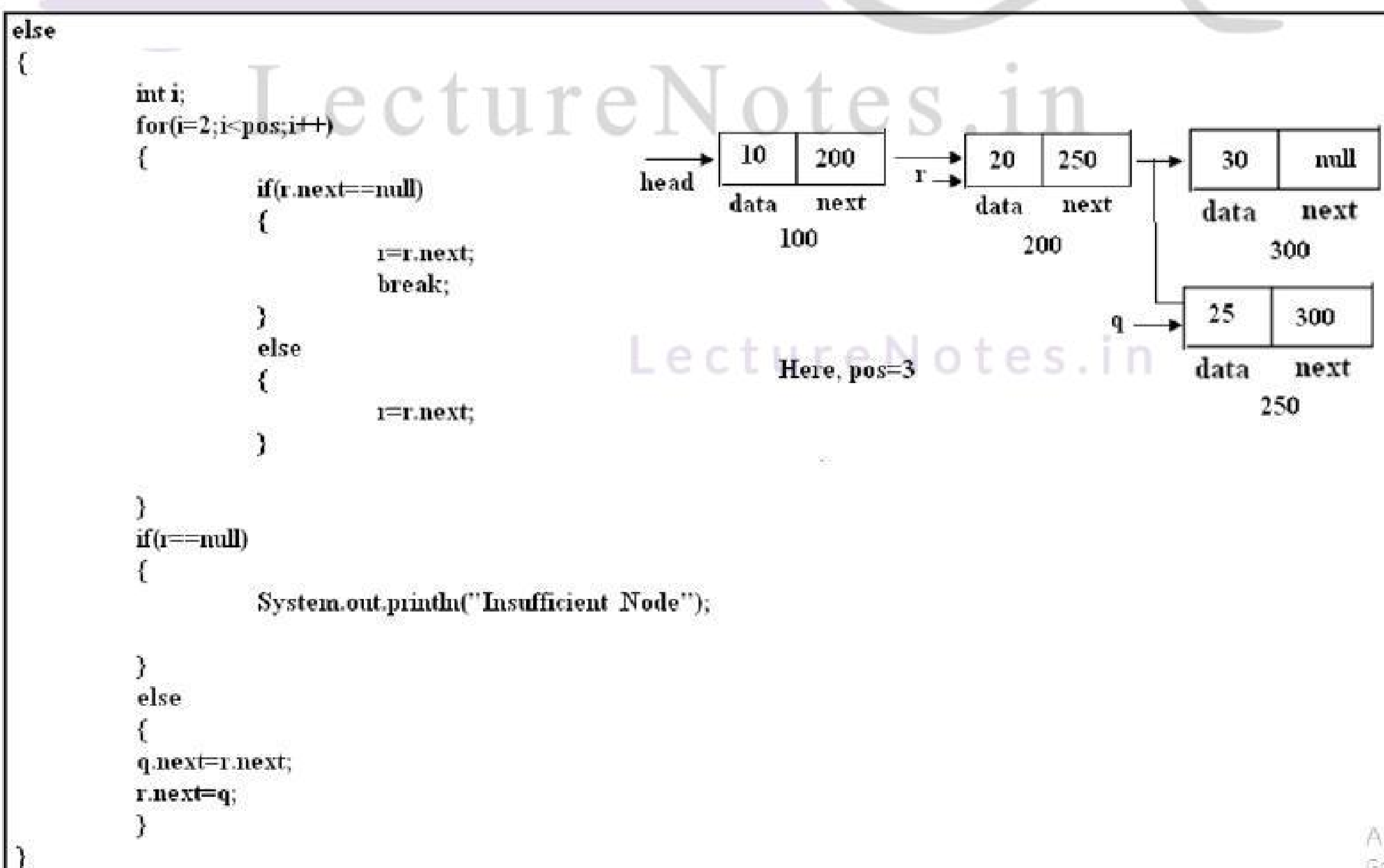
- If the Linked List is empty, then the insertion of an element at middle is not possible. We can check the following condition to know either the Linked List is empty or not.

```

if(head==null)
{
System.out.println("Insertion is not possible");
}

```

Otherwise we can move to the previous node of the position value and we can insert it its specified position as follows.



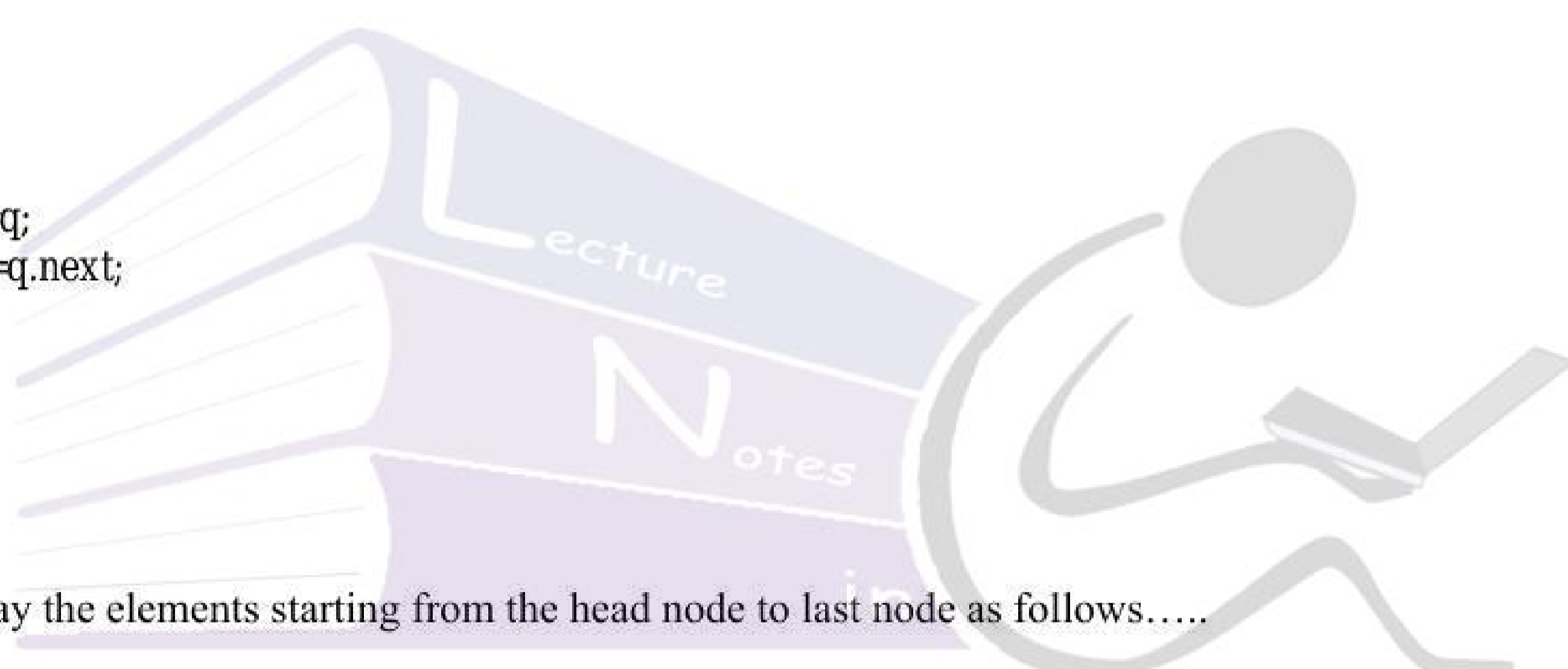
We can delete an element from the linked list depending on the element. If the element was not found we can display an error message. We can delete an element as follows...

```
int del(int x)
{
    q=head;
    r=head;
    while(q!=null)
    {
        if(q.data==x)
        {
            if(head==q)
            {
                head=head.next;
                return x;
            }
            else
            {
                r.next=q.next;
                return x;
            }
        }
        else
        {
            r=q;
            q=q.next;
        }
    }
    return 0;
}
```

Display

We can display the elements starting from the head node to last node as follows.....

```
void display()
{
    node r;
    r=head;
    if(r==null)
    {
        System.out.println("Insufficient nodes");
    }
    else
    {
        System.out.println("Elements in the list are....");
        while(r!=null)
        {
            System.out.println("\t" + r.data);
            r=r.next;
        }
    }
}
```

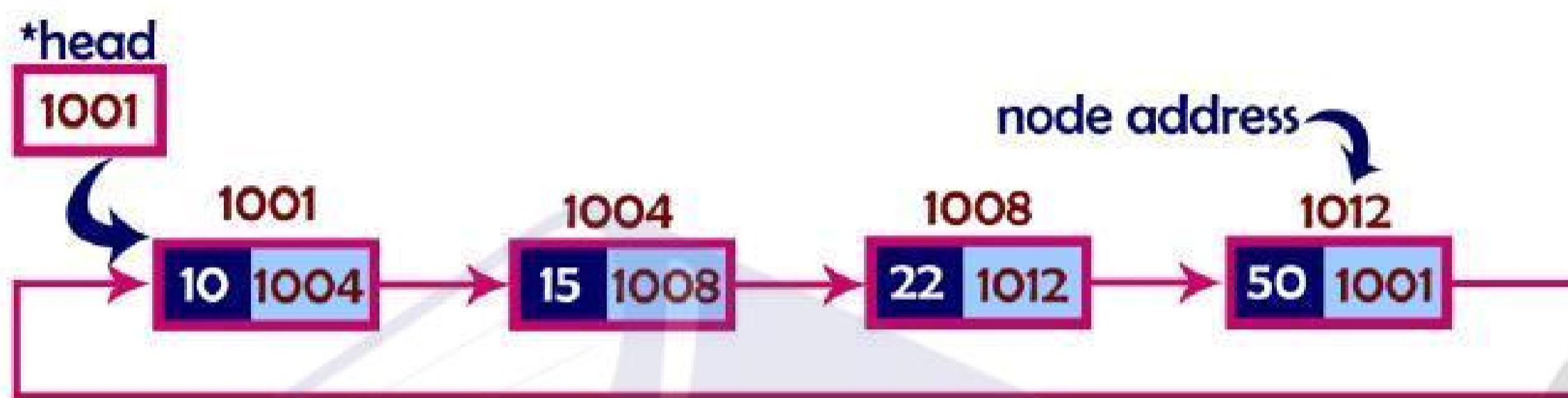


LectureNotes.in

LectureNotes.in

Circular Linked List

In single linked list, every node points to its next node in the sequence and the last node points NULL. But in circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.



Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

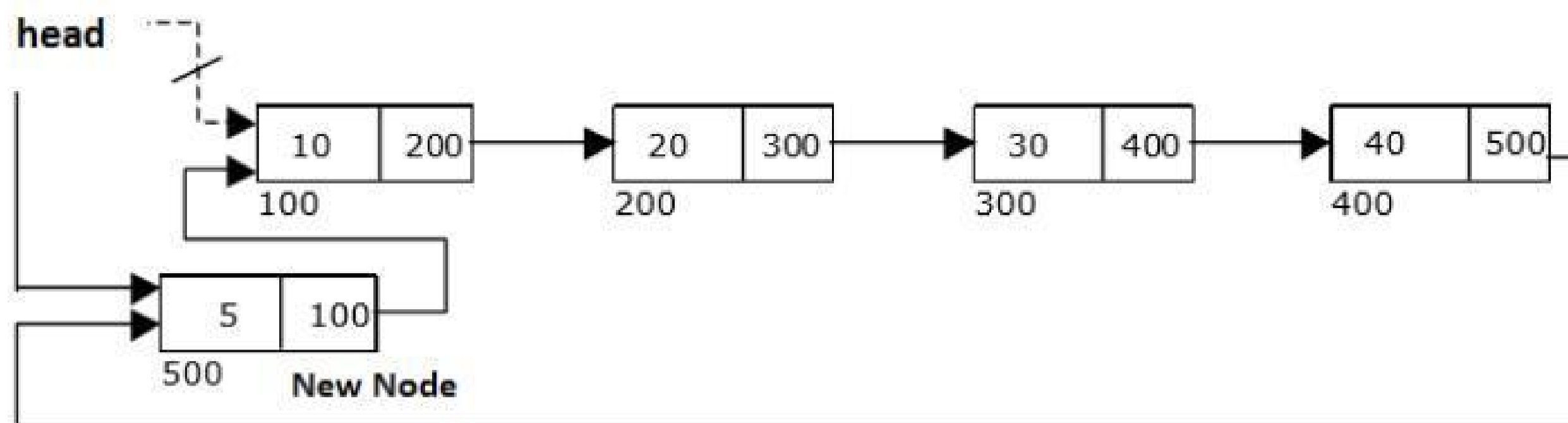
1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then, set **head = newNode** and **newNode.next = head** .
- **Step 4:** If it is **Not Empty** then, define a Node pointer '**r**' and initialize with '**head**'.
- **Step 5:** Keep moving the '**r**' to its next node until it reaches to the last node (until '**r.next == head**').
- **Step 6:** Set '**newNode.next = head**', '**head = newNode**' and '**r.next = head**'.

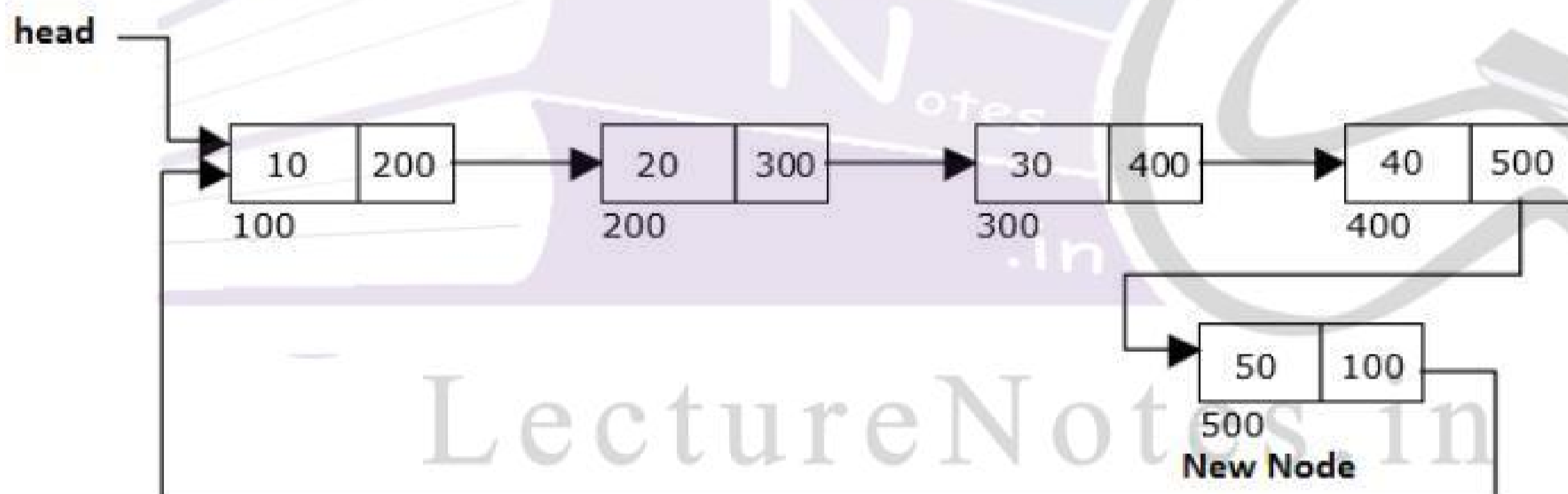
This can be shown in below.



Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**).
- **Step 3:** If it is **Empty** then, set **head = newNode** and **newNode.next = head**.
- **Step 4:** If it is **Not Empty** then, define a node pointer '**r**' and initialize with **head**.
- **Step 5:** Keep moving the '**r**' to its next node until it reaches to the last node in the list (until **r.next == head**).
- **Step 6:** Set **r.next = newNode** and **newNode.next = head**.

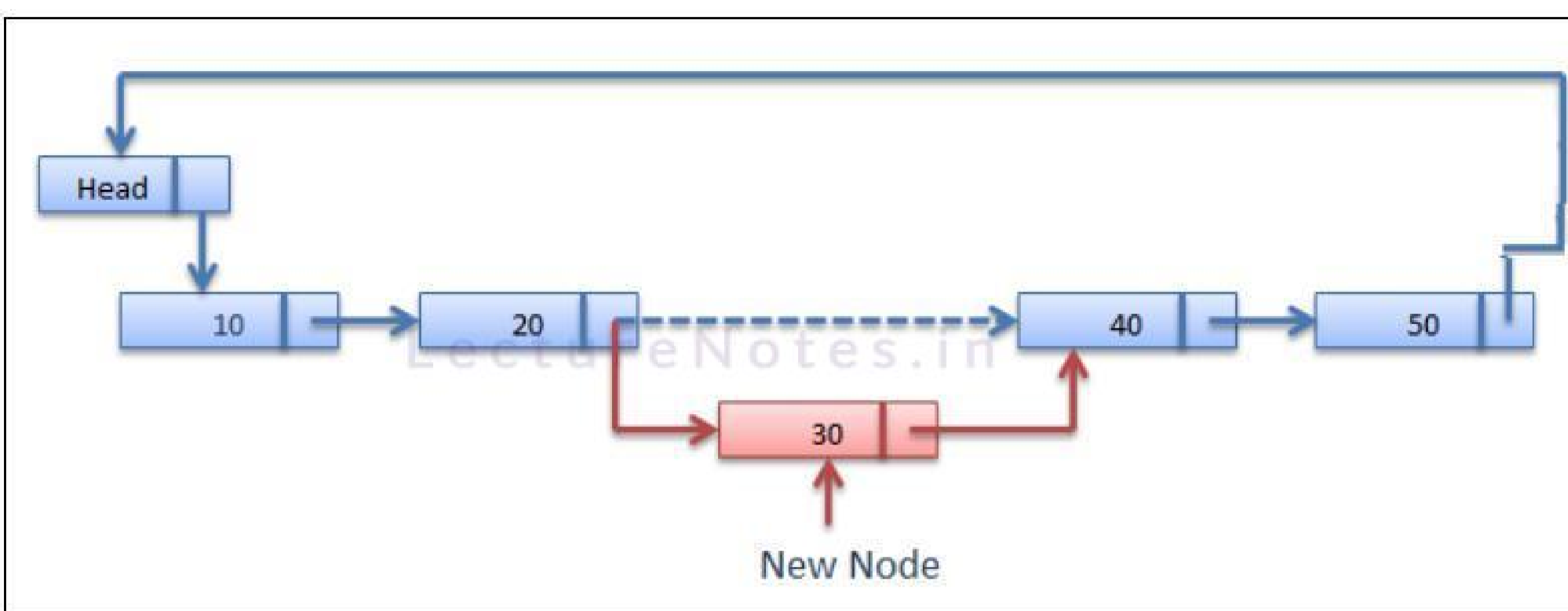


Inserting At Specific location in the list (Before a Node)

We can use the following steps to insert a new node before a node in the circular linked list...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether list is **Empty** (**head == NULL**)
- **Step 3:** If it is **Empty** then, set **head = newNode** and **newNode.next = head**.
- **Step 4:** If it is **Not Empty** then, define a node pointer '**r**' and initialize with **head**.
- **Step 5:** Keep moving the '**r**' to its next node until it reaches to the node before which we want to insert the **newNode**
- **Step 6:** Every time check whether '**r**' is reached to the last node or not. If it is reached to last node then

- **Step 7:** If 'r' is reached to the exact node after which we want to insert the newNode then check whether it is last node (r.next == head).
- **Step 8:** If 'r' is last node then set **r.next = newNode** and **newNode.next = head**.
- **Step 8:** If 'r' is not last node then set **newNode.next = r.next** and **r.next = newNode**.

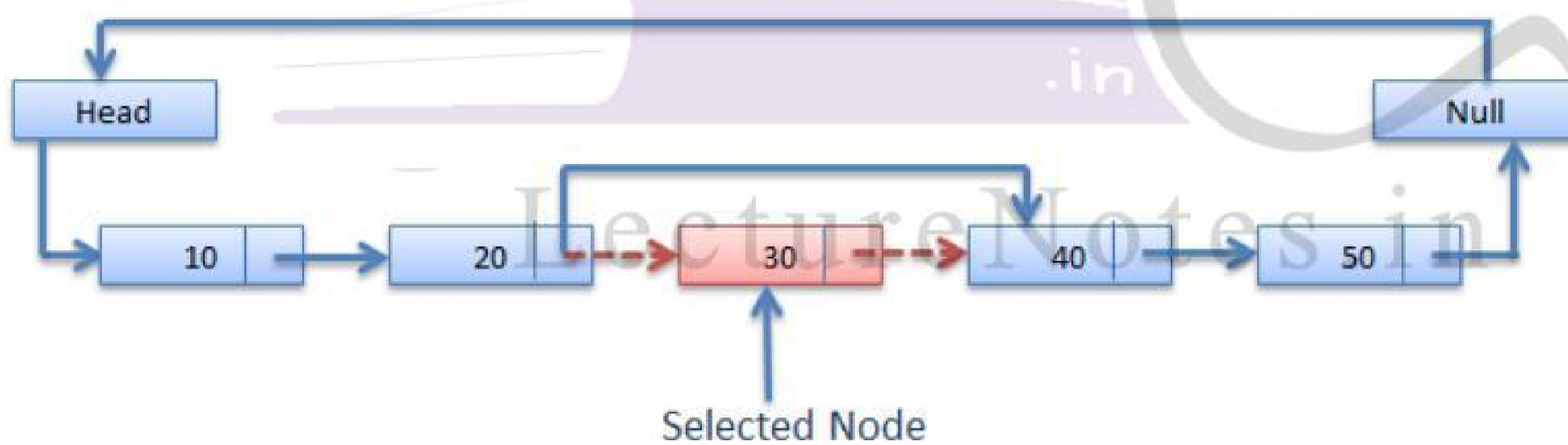


Deleting a node based on data part

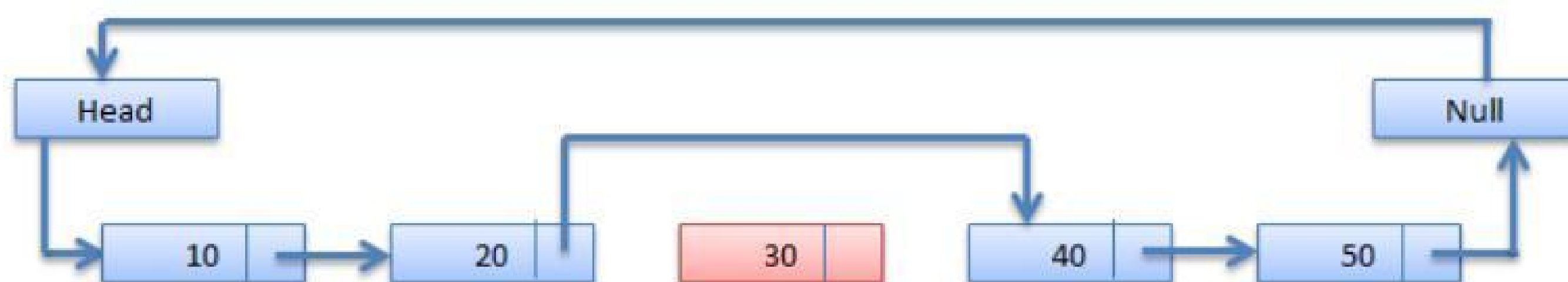
We can delete a node in Circular Singled Linked List depends on the data part of the node. If the node found in the list we simply delete that node otherwise we print that the node not found.

The following is the logic to delete an element within circular singled linked list.

For example consider the following circular singled linked list and we need to delete the element 30 from the list.



After deletion the list is as follows.



*****Insert Program Here*****

Doubly Linked List (DLL)

- A pointer to previous node called as left.
- Data part.
- A pointer to next node called as right.

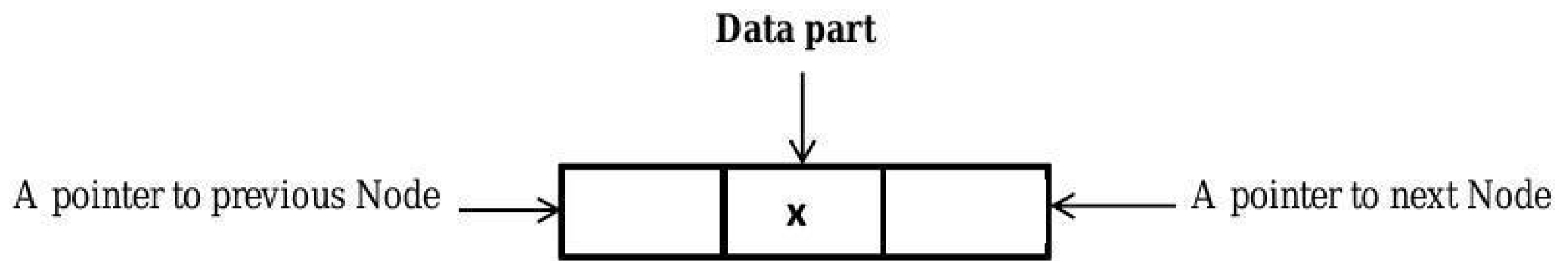
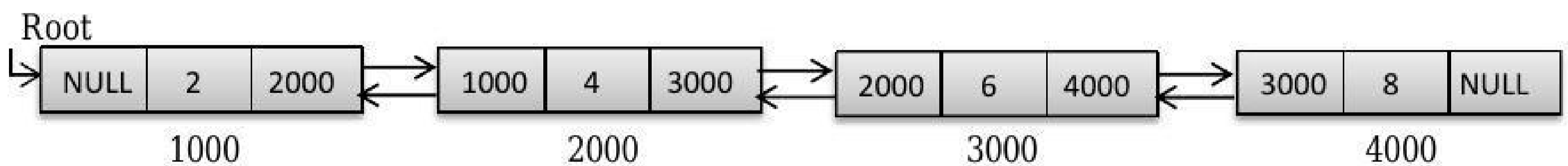


Fig: Node

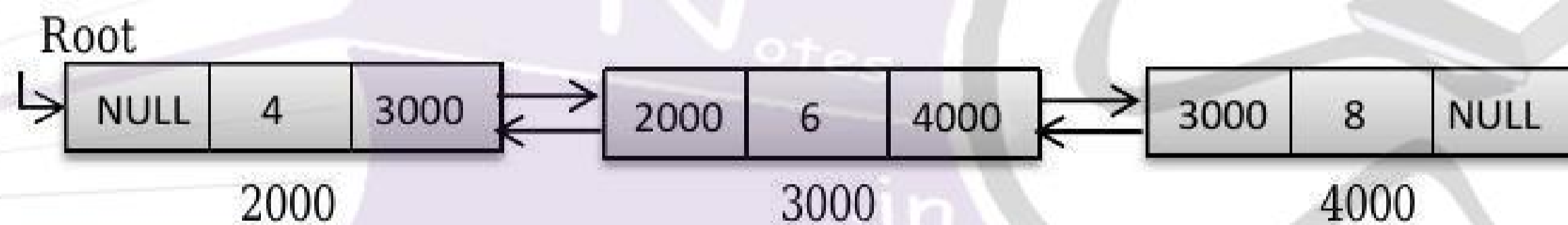
DLL contains external pointer to point the first node address and also first node left pointer and last node right pointer must be set to NULL. This is shown in the following diagram.



If the DLL contains only one node then its left and right pointers must be set to NULL.

Operations of DLL

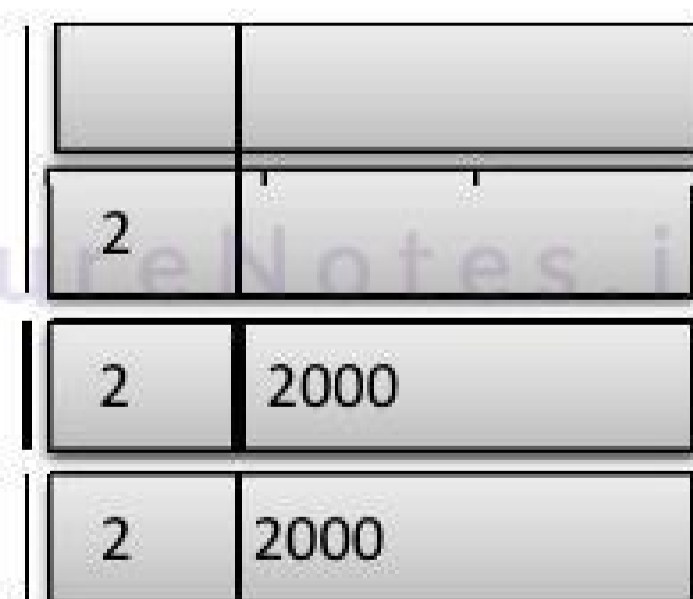
a) Insertion of node at Beginning



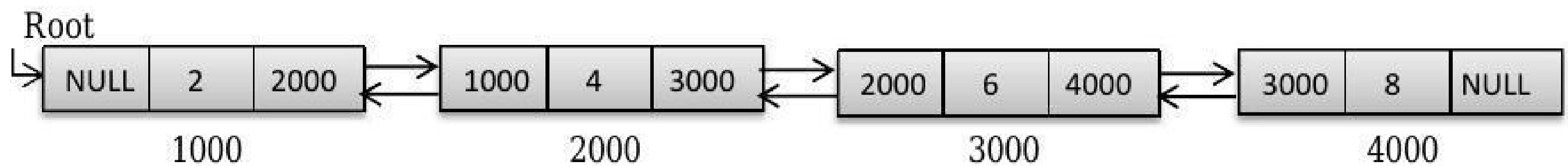
Consider the following DLL

Insertion of node p

- Step - 1: Create a node (p) -----
- Step - 2: Insert data '2' in 'p'.-----
- Step - 3: Assign right (p) to root-----
- Step - 4: Assign Left (p) to NULL-----
- Step - 5: Assign left (root) to p
- Step - 6 : Assign root to p-----

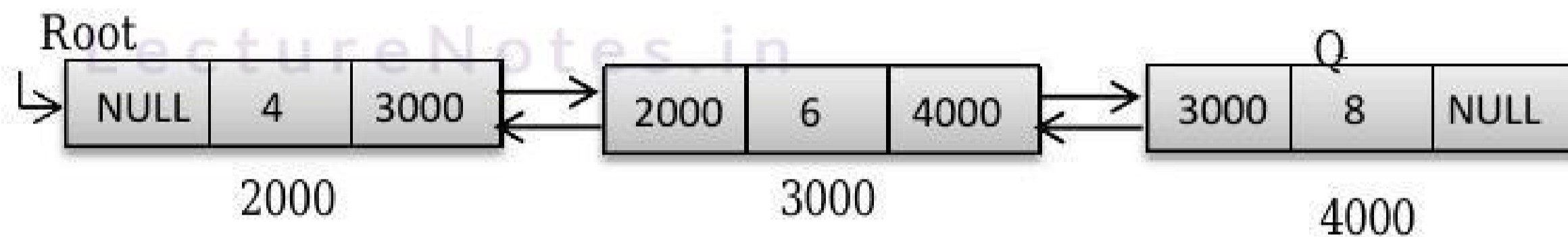


After insertion node at beginning the DLL is;



b) Insertion of node at end

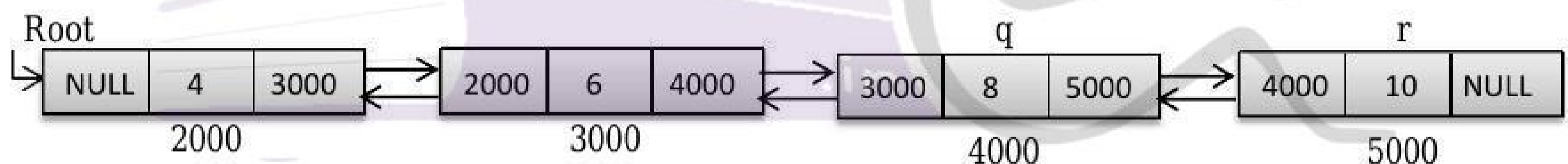
Consider the following DLL



Insertion of node r at end

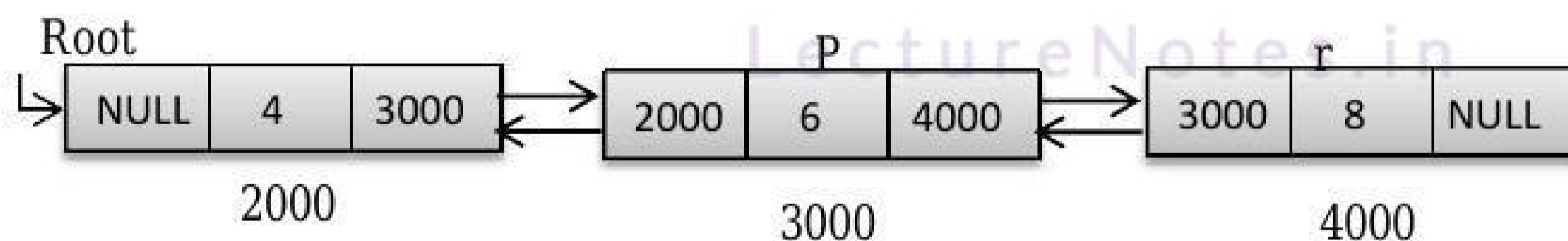
- ✦ Step - 1: Create a node (r)
- ✦ Step - 2: Insert data '10' in 'r'.
- ✦ Step - 3: Assign right (r) to NULL
- ✦ Step - 4: Assign right (q) to r
- ✦ Step - 5 assign left(r) to q

After inserting node at end the DLL is



c) Insertion of node at middle

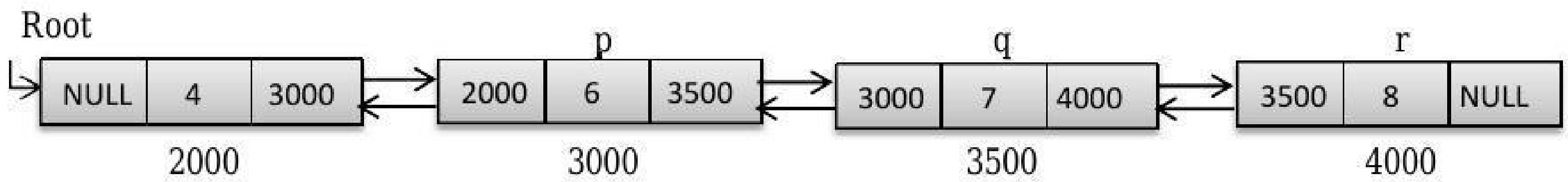
Consider the following DLL



Insertion of node q in between p and r

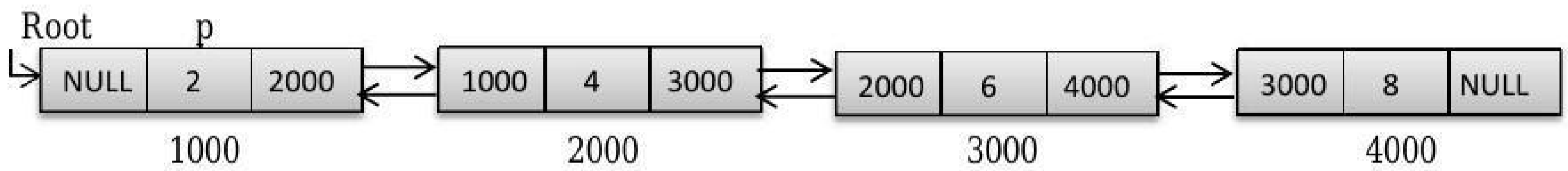
- ✦ Step - 1: Create a node (q)
- ✦ Step - 2: Insert data '7' in 'q'.
- ✦ Step - 3: Assign right (p) to q
- ✦ Step - 4: Assign left (q) to p
- ✦ Step - 5: assign right (q) to r
- ✦ Step - 6: Assign left (r) to q

After inserting node at end the DLL is;



d) Deletion of node at Beginning

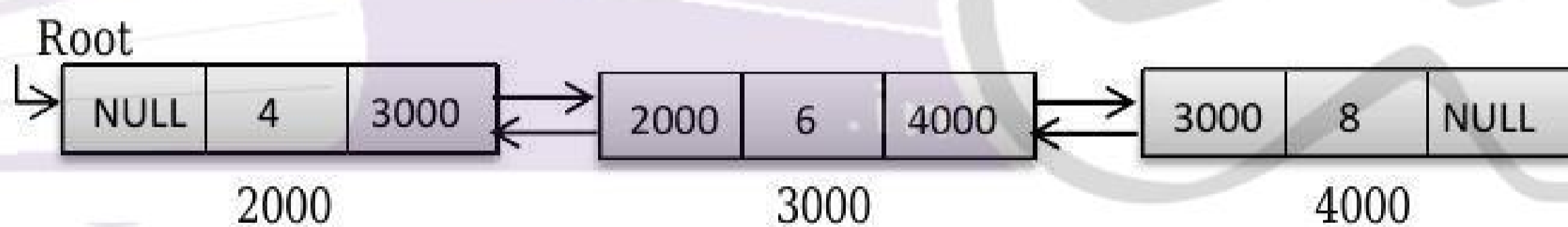
Consider the following DLL



Deletion of node p

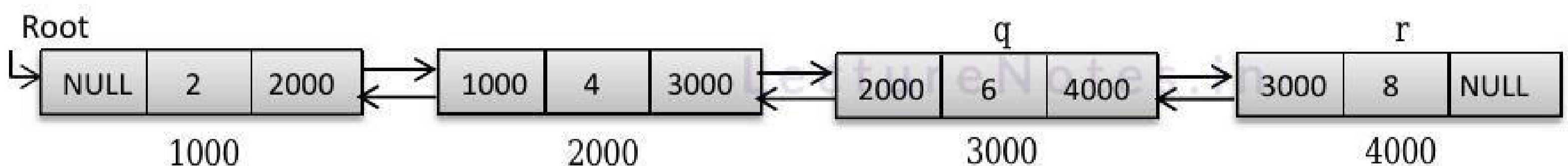
- ✚ Step - 1: delete 2 from p
- ✚ Step - 2: Assign root to next (p)
- ✚ Step - 3: Assign left (root) NULL
- ✚ Step - 4: remove node 'p'.

After Deletion of node at beginning the DLL is;



e) Deletion of node at End

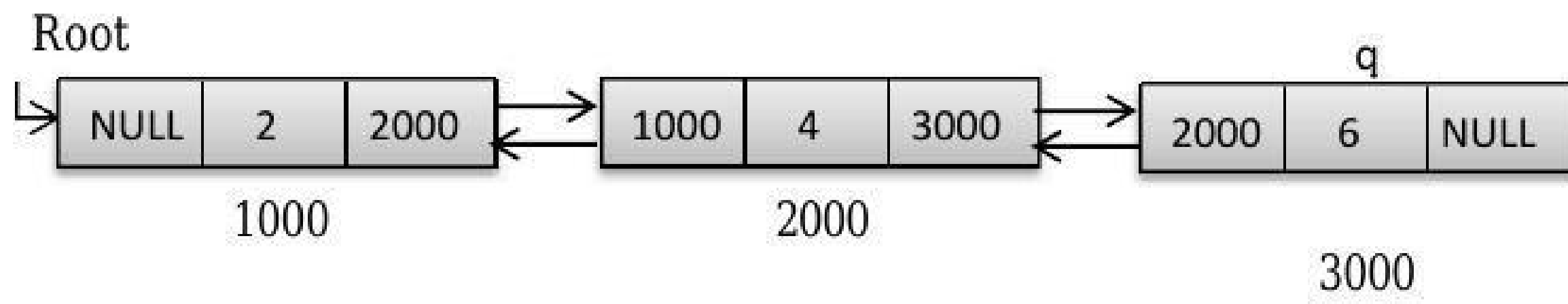
Consider the following DLL



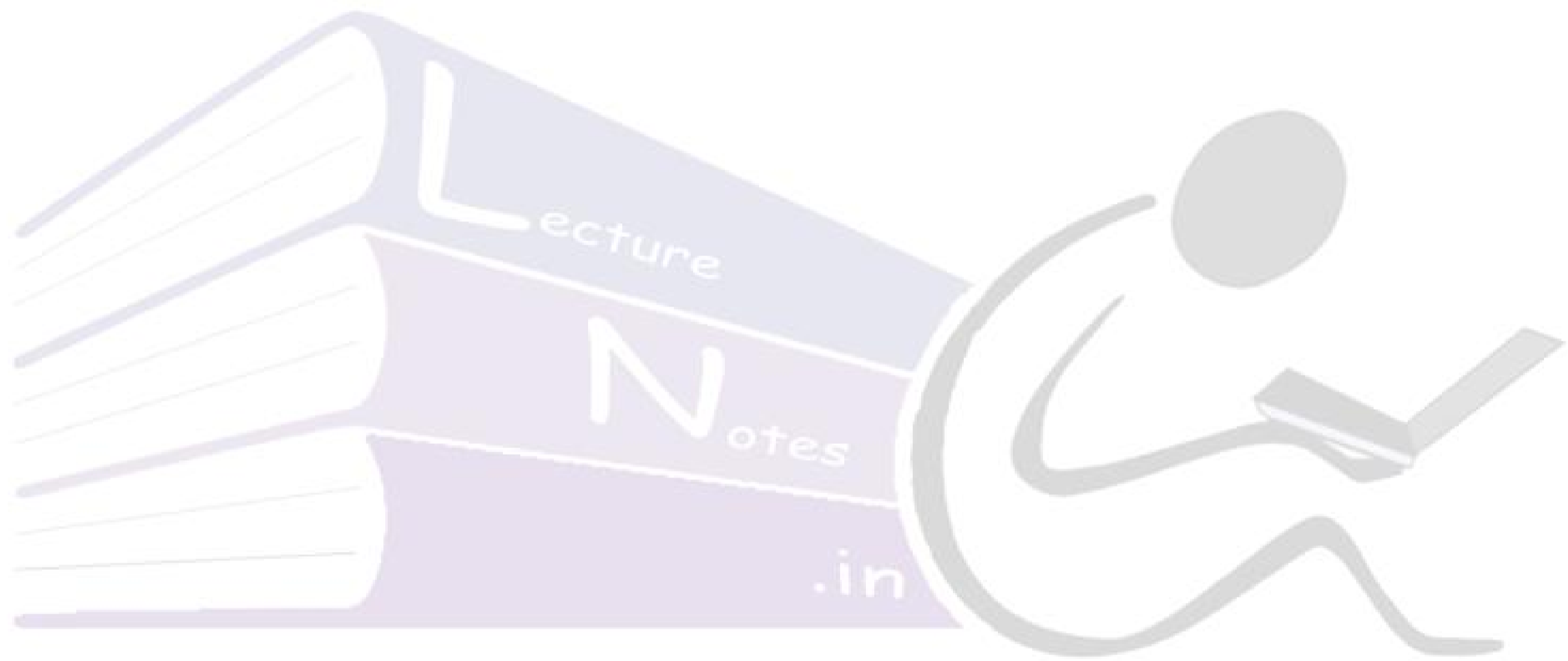
Deletion of node r

- ✚ Step - 1: delete 8 from r
- ✚ Step - 2: Assign right (q) to NULL
- ✚ Step - 3: remove node 'r'.

After Deletion of node at end the DLL is;



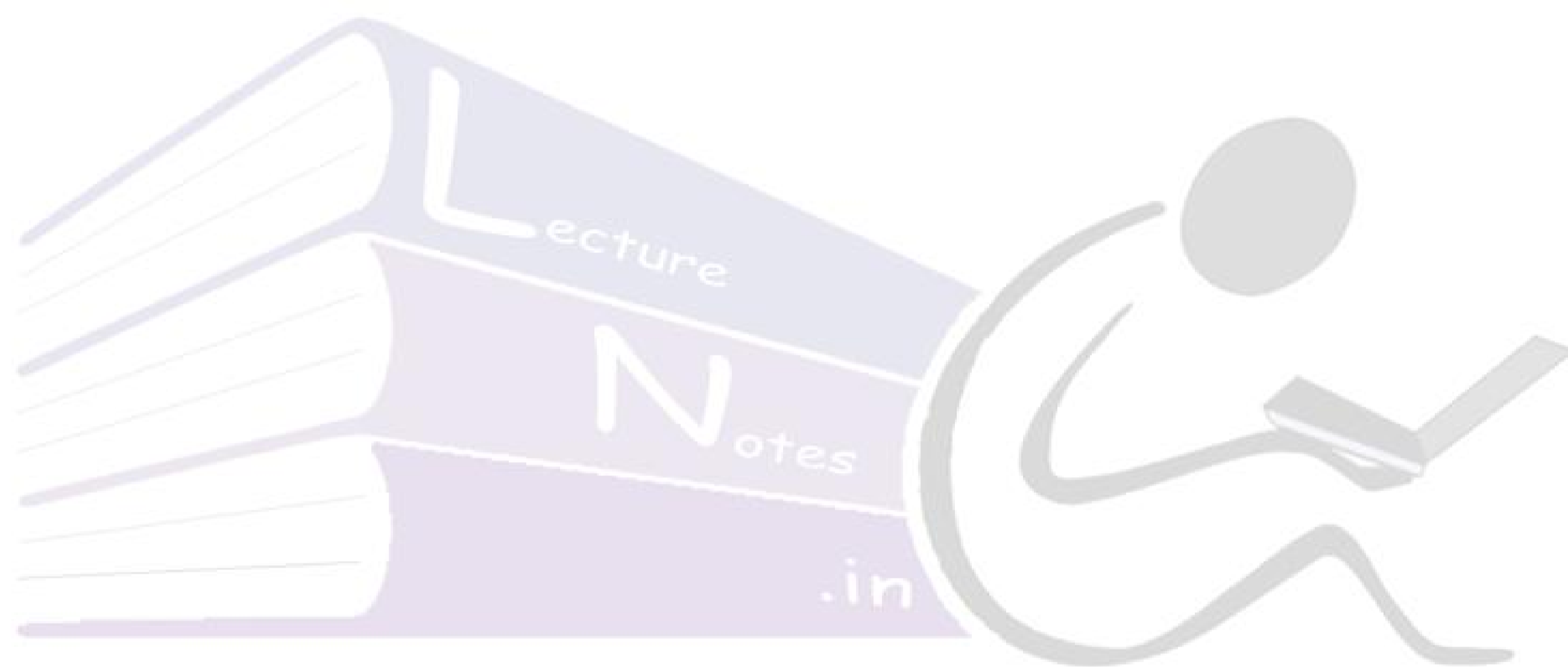
LectureNotes.in



LectureNotes.in

LectureNotes.in

LectureNotes.in



LectureNotes.in

LectureNotes.in

Stack using Linked List (OR) Linked Stack

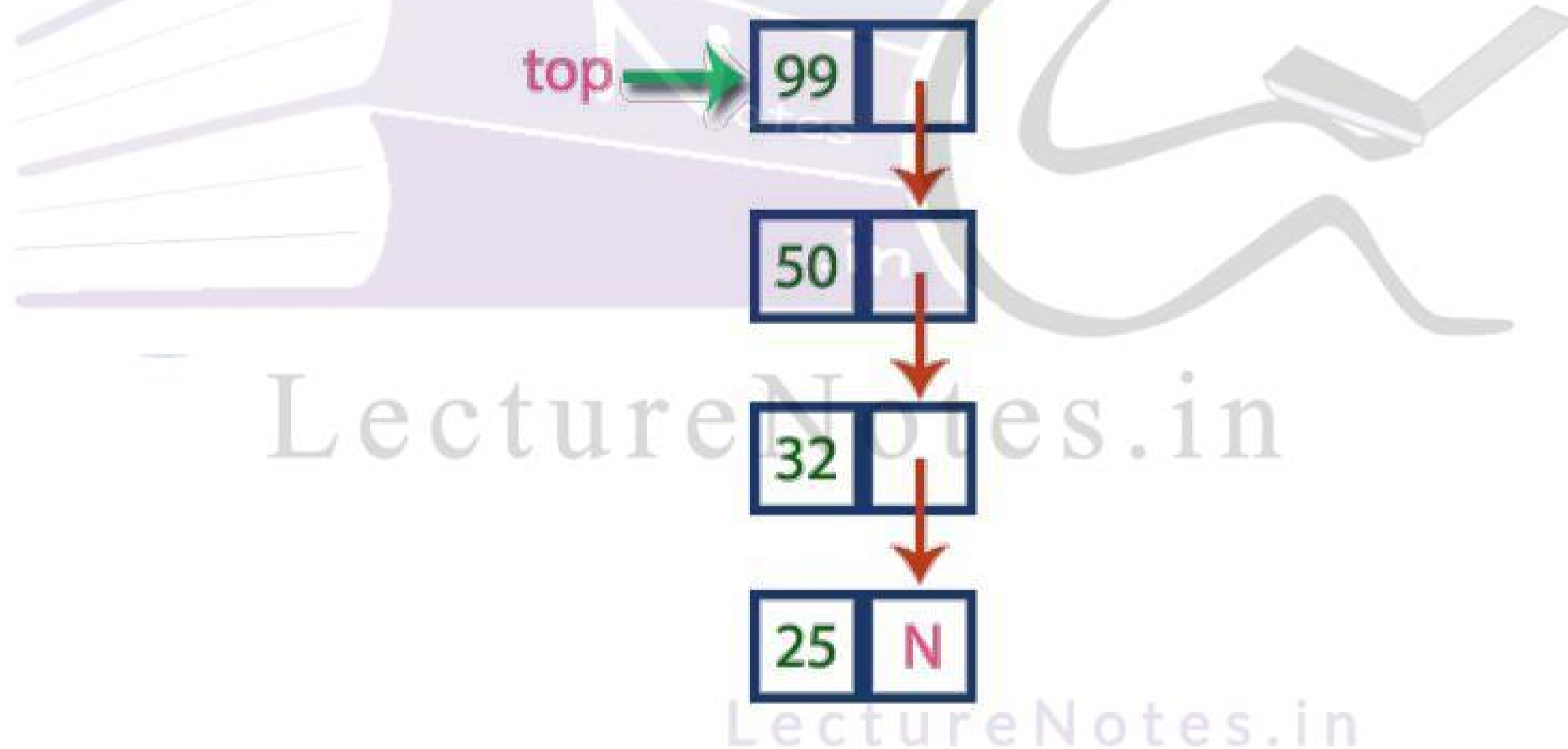
The major problem with the stack implemented using array is, it works only for fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using array is not suitable, when we don't know the size of data which we are going to use.

A stack data structure can be implemented by using linked list data structure. The stack implemented using linked list can work for unlimited number of values. That means stack implemented using linked list works for variable size of data.

So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as **'top'** element. That means every newly inserted element is pointed by **'top'**. Whenever we want to remove an element from the stack, simply remove the node which is pointed by **'top'** by moving **'top'** to its next node in the list. The **next** field of the first element must be always **NULL**.

Example



In above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32, 50 and 99.

Operations

To implement stack using linked list, we need to set the following things before implementing actual operations.

- **Step 1:** Import all the required packages which are used in the program.
- **Step 2:** Define a 'Node' class with two member's **data** and **next**.
- **Step 3:** Define a **Node** pointer '**top**' and set it to **NULL**.
- **Step 4:** Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main** method.

push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

- **Step 1:** Create a **newNode** with given value.
- **Step 2:** Check whether stack is **Empty** (**top == NULL**)
- **Step 3:** If it is **Empty**, then set **newNode.next = NULL**.
- **Step 4:** If it is **Not Empty**, then set **newNode.next = top**.
- **Step 5:** Finally, set **top = newNode**.

pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

- **Step 1:** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2:** If it is **Empty**, then display "**Stack is Empty!!! Deletion is not possible!!!**" and terminate the function
- **Step 3:** If it is **Not Empty**, then set '**top = top.next**'.

display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

- **Step 1:** Check whether stack is **Empty** (**top == NULL**).
- **Step 2:** If it is **Empty**, then display '**Stack is Empty!!!**' and terminate the function.
- **Step 3:** If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **top**.

- **Step 4:** Display '**temp.data**' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack (**temp.next != NULL**).

The following is the java program that implements the stack using linked list.

Aim: To write a java program that implements a stack data structure using singled linked list.

Program: LinkedStack.java

```
import java.util.*;
class node
{
    int data;
    node next;
}
class LinkedStack
{
    node top;
    LinkedStack()
    {
        top=null;
    }
    void push(int ele)
    {
        node q=new node();
        q.data=ele;
        if(top==null)
        {
            q.next=null;
            top=q;
        }
        else
        {
            q.next=top;
            top=q;
        }
        System.out.println(ele + " inserted successfully");
        System.out.println("\n*****");
    }
    int pop()
    {
        if(top==null)
        {
            return 0;
        }
        else
        {
```

```

        int d=top.data;
        top=top.next;
        return d;
    }
}
void display()
{
    if(top==null)
        System.out.println("Linked Stack is empty");
        System.out.println("\n*****");
    }
    else
    {
        node r;
        r=top;
        System.out.println("The elements in Linked Stack are...\n");
        while(r!=null)
        {
            System.out.println(r.data);
            r=r.next;
        }
        System.out.println("\n*****");
    }
}
}
class LinkedStackDemo
{
    public static void main(String[] args)
    {
        int ele,ch;
        Scanner sc=new Scanner(System.in);
        LinkedStack ls=new LinkedStack();
        System.out.println("\n1.PUSH");
        System.out.println("\n2.POP");
        System.out.println("\n3.DISPLAY");
        System.out.println("\n4.EXIT");
        do
        {
            System.out.println("\nEnter your choice:");
            ch=sc.nextInt();
            System.out.println("\n*****");
            switch(ch)
            {
                case 1: System.out.println("Enter element to insert:");
                        ele=sc.nextInt();

```

```

        ls.push(ele);
        break;
    case 2: ele=ls.pop();
        if(ele==0)
        {
            System.out.println("Linked Stack is Underflow");

            System.out.println("\n*****");
        }
        else
        {
            System.out.println("The popped element is:" + ele);

            System.out.println("\n*****");
        }
        break;
    case 3: ls.display();
        break;
    case 4: System.exit(0);
        }
    }while(ch!=4);
}

```

Output:

```
1.PUSH
```

```
2.POP
```

```
3.DISPLAY
```

```
4.EXIT
```

```
Enter your choice:
```

```
1
```

```
*****
```

```
Enter element to insert:
```

```
10
```

```
10 inserted successfully
```

```
*****
```

```
Enter your choice:
```

```
1
```

```
*****
```

```
Enter element to insert:
```

```
20
```

```
20 inserted successfully
```

```
*****
```

```
Enter your choice:
```

```
3
```

```
*****
```

```
The elements in Linked Stack are...
```

```
20
```

```
10
```

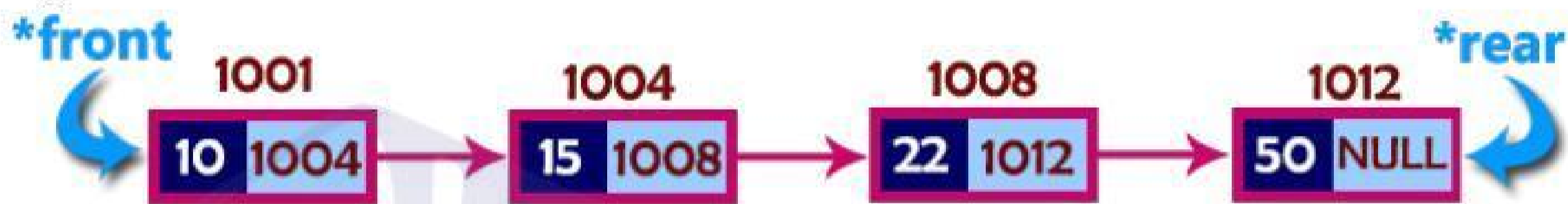

Queue using Linked List (OR) Linked Queue

The major problem with the queue implemented using array is, It will work for only fixed number of data. That means, the amount of data must be specified in the beginning itself. Queue using array is not suitable when we don't know the size of data which we are going to use.

A queue data structure can be implemented using linked list data structure. The queue which is implemented using linked list can work for unlimited number of values. That means, queue using linked list can work for variable size of data (No need to fix the size at beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

Example



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

Operations

To implement queue using linked list, we need to set the following things before implementing actual operations.

- **Step 1:** Import all the required packages which are used in the program.
- **Step 2:** Define a '**Node**' class with two member's **data** and **next**.
- **Step 3:** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4:** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

We can use the following steps to insert a new node into the queue...

- **Step 1:** Create a **newNode** with given value and set '**newNode.next**' to **NULL**.
- **Step 2:** Check whether queue is **Empty** (**rear == NULL**)
- **Step 3:** If it is **Empty** then, set **front = newNode** and **rear = newNode**.
- **Step 4:** If it is **Not Empty** then, set **rear.next = newNode** and **rear = newNode**.

deQueue() - Deleting an Element from Queue

We can use the following steps to delete a node from the queue...

- **Step 1:** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2:** If it is **Empty**, then display "**Queue is Empty!!! Deletion is not possible!!!**" and terminate from the function
- **Step 3:** If it is **Not Empty** then, delete the element at front position and set '**front = front.next**'.

display() - Displaying the elements of Queue

We can use the following steps to display the elements (nodes) of a queue...

- **Step 1:** Check whether queue is **Empty** (**front == NULL**).
- **Step 2:** If it is **Empty** then, display '**Queue is Empty!!!**' and terminate the function.
- **Step 3:** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with **front**.

- **Step 4:** Display '**temp.data**' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp.next != NULL**).

The following is the java program that implements the queue using linked list.

Aim: To write a java program that implements a queue data structure using singled linked list.

Program: LinkedQueue.java

```

import java.util.*;
class node
{
    int data;
    node next;
}
class LinkedQueue
{
    node front,rear;
    LinkedQueue()
    {
        front=null;
        rear=null;
    }
    void insert(int ele)
    {
        node q=new node();
        q.data=ele;
        q.next=null;
        if(front==null && rear==null)
        {
            rear=q;
            front=q;
        }
        else
        {
            rear.next=q;
            rear=q;
        }
        System.out.println(ele + " inserted into Queue Succussfully");
        System.out.println("*****");
    }
    int delete()
    {
        if(front==null && rear==null)
        {

```

```

        return 0;
    }
    else
    {
        int d=front.data;
        front=front.next;
        if(front==null)
        {
            rear=null;
        }
        return d;
    }
}
void display()
{
    if(rear==null)
    {
        System.out.println("Linked Queue is empty");
        System.out.println("*****");
    }
    else
    {
        node r;
        r=front;
        System.out.println("The Linked Queue Elements are....");
        while(r!=null)
        {
            System.out.print(r.data + "\t");
            r=r.next;
        }
        System.out.println("\n*****");
    }
}
}
}
}
class LinkedQueueDemo
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        int ele,ch;
        LinkedQueue lq=new LinkedQueue();

        System.out.println("1.INSERT");
        System.out.println("2.DELETE");
        System.out.println("3.DISPLAY");
    }
}

```

```

System.out.println("4.EXIT");
System.out.println("*****");
do
{
    System.out.println("Enter your choice:");
    ch=sc.nextInt();

    switch(ch)
    {
        case 1: System.out.println("Enter element to insert:");
                ele=sc.nextInt();
                lq.insert(ele);
                break;
        case 2: ele=lq.delete();
                if(ele==0)
                {
                    System.out.println("Linked Queue is Underflow");
                    System.out.println("*****");
                }
                else
                {
                    System.out.println("The deleted element is:" +ele);
                    System.out.println("*****");
                }
                break;
        case 3: lq.display();
                break;
        case 4: System.exit(0);
    }
}while(ch!=4);
}
}

```

Output:

```

1.INSERT
2.DELETE
3.DISPLAY
4.EXIT
*****
Enter your choice:
1
Enter element to insert:
10
10 inserted into Queue Succussfully
*****

```

```

Enter your choice:
1
Enter element to insert:
20
20 inserted into Queue Succussfully
*****
Enter your choice:
3
The Linked Queue Elements are....
10    20
*****
Enter your choice:

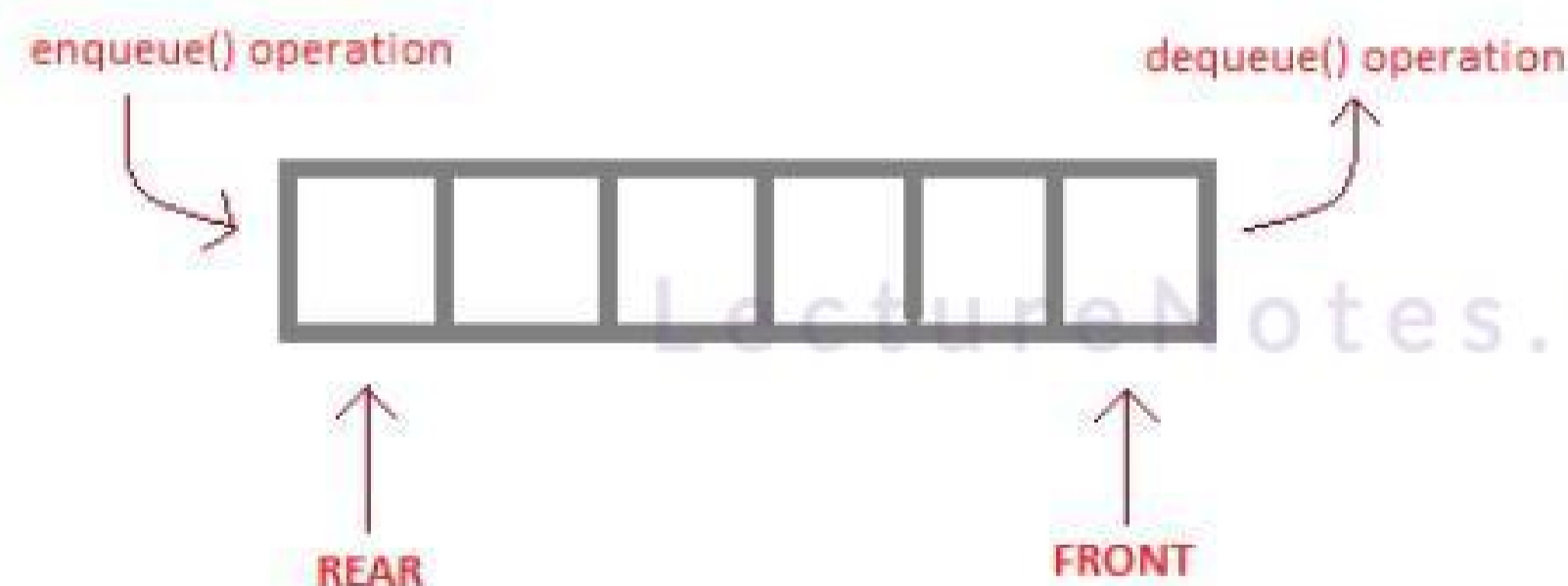
```

QUEUE DATA STRUCTURE

Queue



- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.
- The end which is used to insert the elements into the queue is called as rear end.
- The end which is used to delete the elements from the queue is called as front end.
- Initially the front and rear of queue are set to -1.
- In queue data structure, the insertion and deletion operations are performed based on **FIFO (First In First Out)** principle.
- The process of inserting elements into the queue is called as enqueue and this can be done by the function enqueue().
- The process of deleting elements into the queue is called as dequeue and this can be done by the function dequeue().
- The general structure of a Queue data structure is shown in below.



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

QUEUE DATA STRUCTURE

Applications:

Typical uses of queues are in simulations and operating systems.

- Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
- Computer systems must often provide a “holding area” for messages between two processes, two programs, or even two systems. This holding area is usually called a “buffer” and is often implemented as a queue.

Operations on a Queue

The following operations are performed on a queue data structure...

1. **enQueue(value)** - (To insert an element into the queue)
2. **deQueue()** - (To delete an element from the queue)
3. **display()** - (To display the elements of the queue)

Queue data structure can be implemented in two ways. They are as follows...

1. **Using Array**
2. **Using Linked List**

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

Implementation of Queue Using Array

A queue data structure can be implemented using one dimensional array. But, queue implemented using array can store only fixed number of data values.

The implementation of queue data structure using array is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables '**front**' and '**rear**'. Initially both '**front**' and '**rear**' are set to -1.

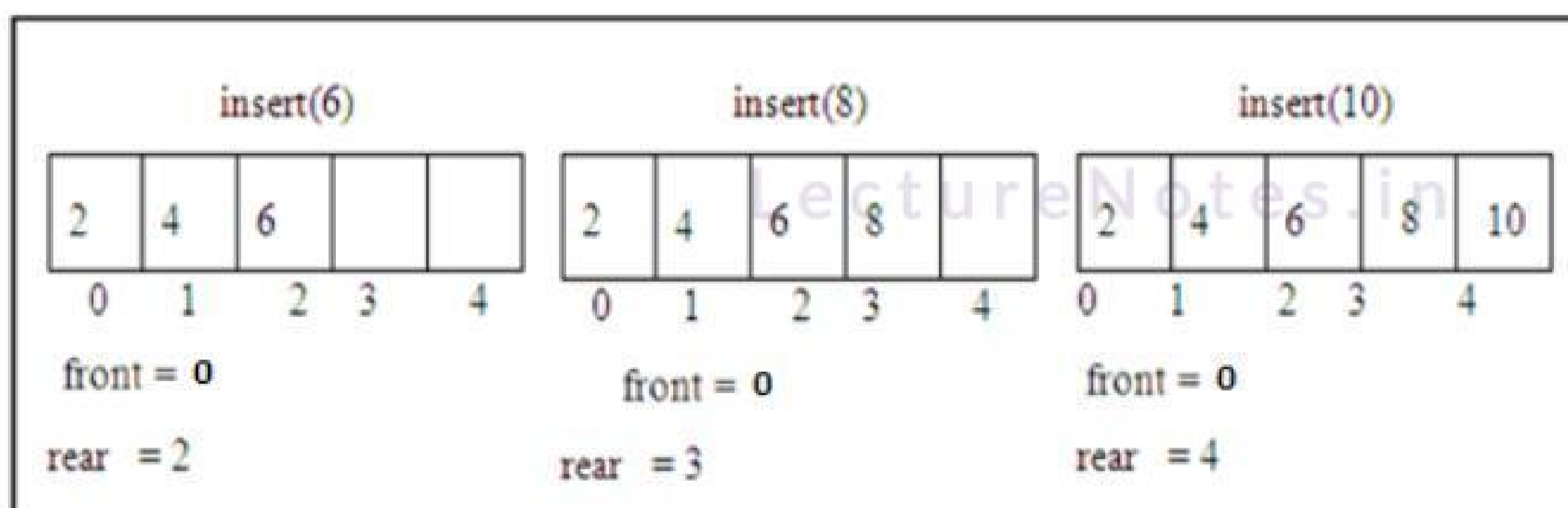
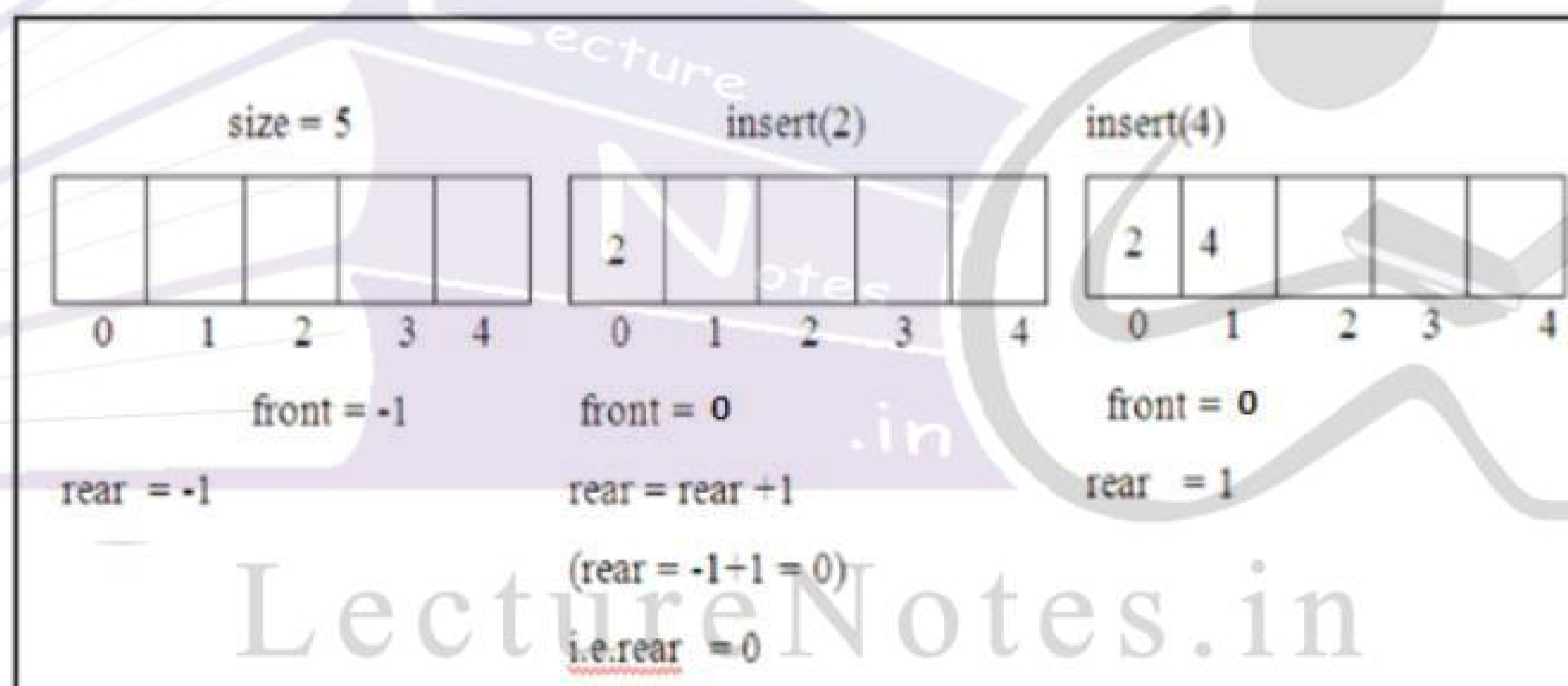
Queue Operations using Array

enQueue(value) - Inserting value into the queue

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

- **Step 1:** Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- **Step 2:** If it is **FULL**, then display "**Queue is FULL!!! Insertion is not possible!!!**" and terminate the function.
- **Step 3:** If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **que[rear] = value**.

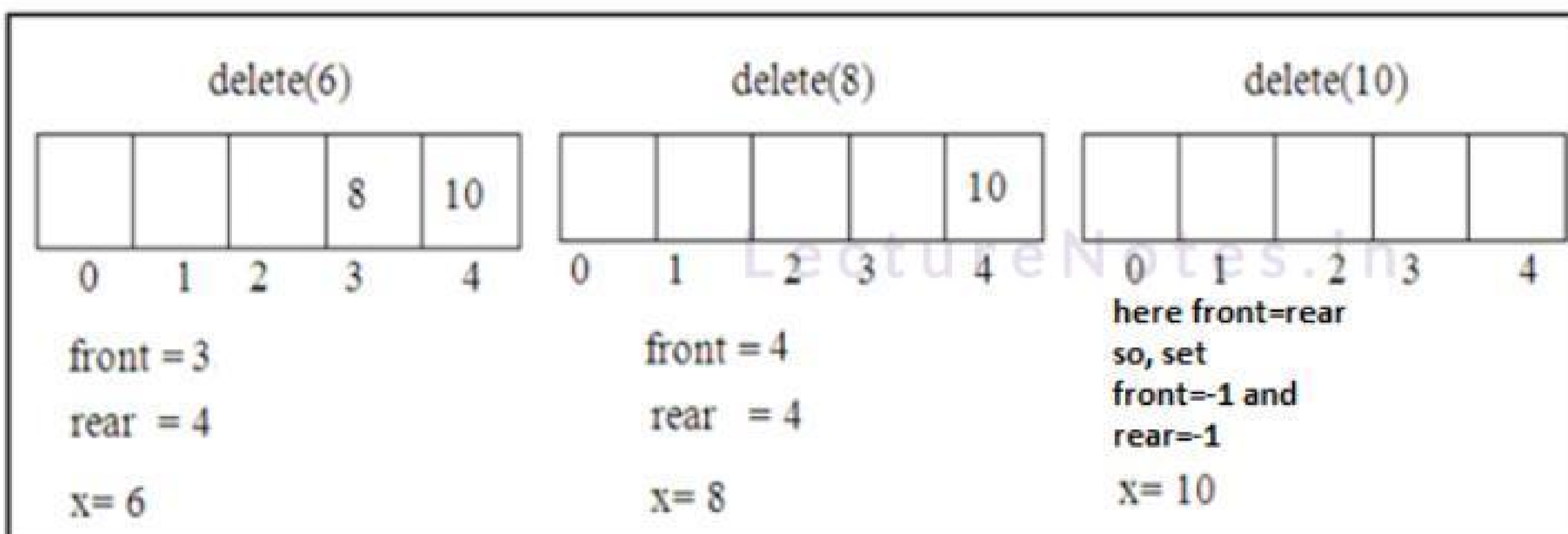
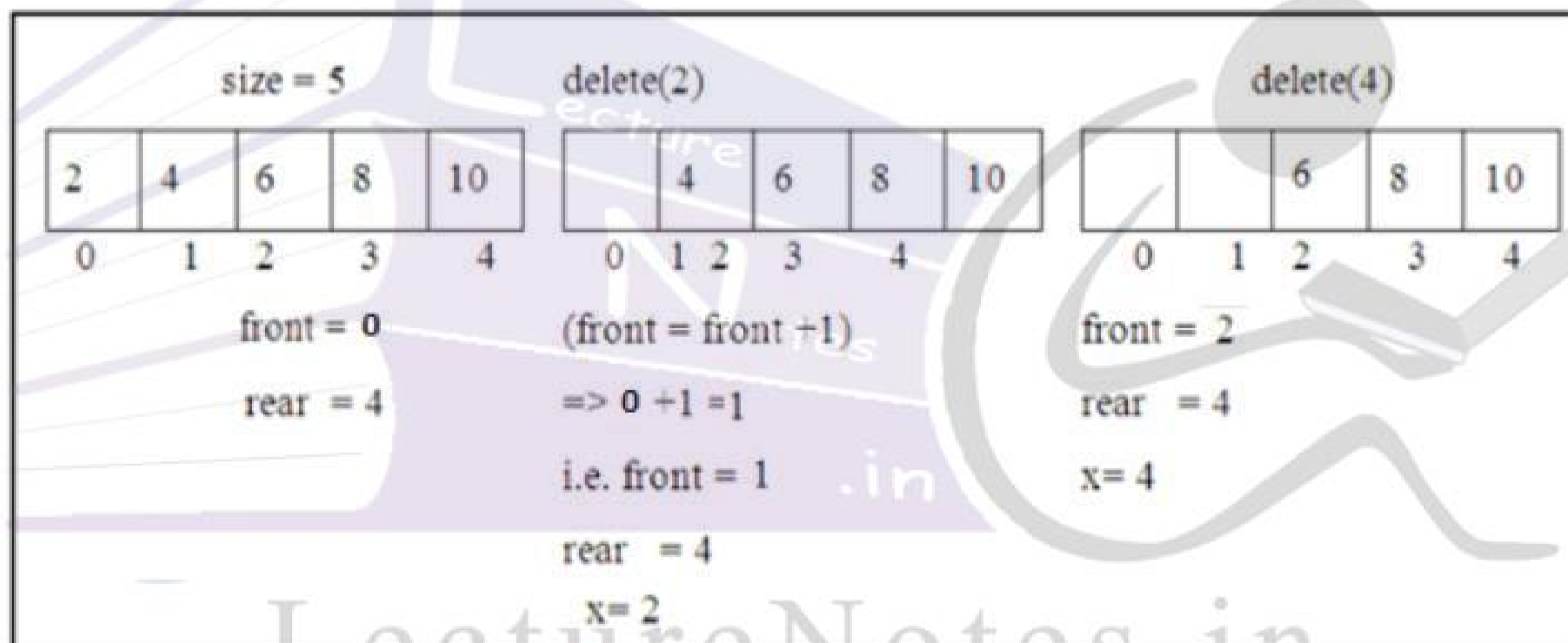
Example: Initially front and rear are set to '-1'.



deQueue() - Deleting a value from the Queue

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

- **Step 1:** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2:** If it is **EMPTY**, then display "**Queue is EMPTY!!! Deletion is not possible!!!**" and terminate the function.
- **Step 3:** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and then increment the **front** value by one (**front++**).. Then check whether both **front** and **rear** are equal (**front == rear**), if it is **TRUE**, then set both **front** and **rear** to **'-1'** (**front = rear = -1**).



display() - Displays the elements of a Queue

We can use the following steps to display the elements of a queue...

- **Step 1:** Check whether **queue** is **EMPTY** . (**front == rear**)
- **Step 2:** If it is **EMPTY** , then display "**Queue is EMPTY!!!**" and terminate the function.
- **Step 3:** If it is **NOT EMPTY** , then define an integer variable '**i**' and set '**i = front+1**'.
- **Step 3:** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value is equal to **rear** (**i <= rear**)

Program to implement Queue using Array

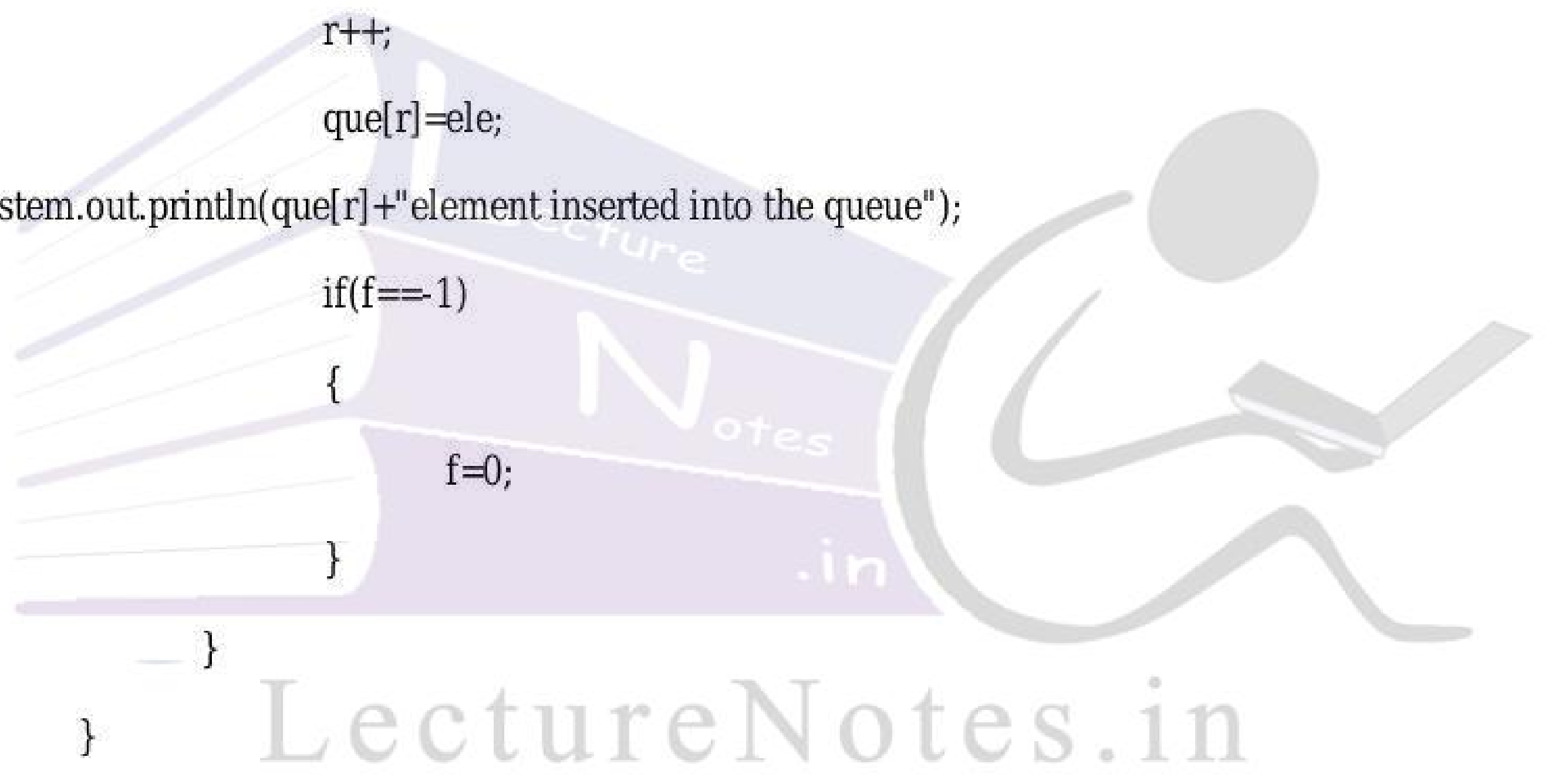
The following is a java program that implements a queue data structure by using arrays.

Aim: To write a java program that implements a queue data structure by using arrays.

Program: QueueDemo.java

```
import java.io.*;
import java.util.*;
class Queue
{
    int que[],max,f,r;
    Queue(int size)
    {
        max=size;
        f=1;
        r=1;
        que=new int[max];
    }
}
```

```
void insert(int ele)
{
    if(r==max-1)
    {
        System.out.println("Queue Overflow");
    }
    else
    {
        r++;
        que[r]=ele;
        System.out.println(que[r]+" element inserted into the queue");
        if(f==1)
        {
            f=0;
        }
    }
}
```

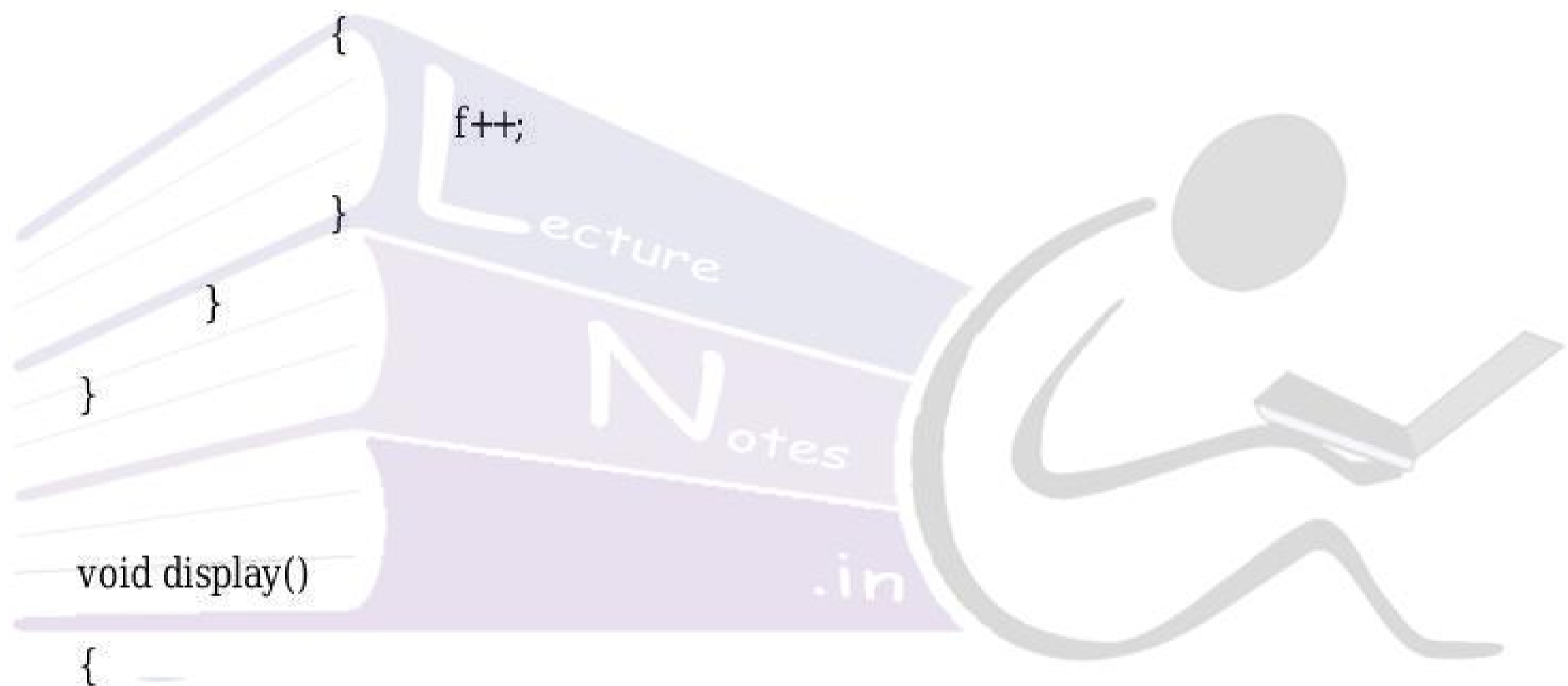


```
void delete()
{
    int ele;
    if(f==1&&r==1)
        System.out.println("Queue is underflow");
    else
    {
```

```

ele=que[f];
System.out.println(ele+"element deleted form the Queue ");
if(f==r)
{
    f=1;
    r=1;
}
else

```



```

void display()
{

```

```

    if(f==1)
    {
        System.out.println("Queue is empty");
    }
    else
    {
        System.out.println("Elements in the queue are....");
        for(int i=f;i<=r;i++)
        {

```

```
        System.out.print("\t" + que[i]);
    }
    System.out.println("");
}
} LectureNotes.in
}
```

```
class QueueDemo
```

```
{
    public static void main(String args[])
    {
        int ch,ele,n;
        Scanner d=new Scanner(System.in);
        System.out.println("Enter size of the queue:");
        n= d.nextInt();
        Queue q=new Queue(n);
        System.out.println("\n1.Insert\n");
        System.out.println("2.Delete\n");
        System.out.println("3.Display\n");
        System.out.println("4.Exit\n");
        do
        {
            System.out.println("Enter your choice...");
            ch=d.nextInt();
```

```
switch(ch)
{
    case 1:
        System.out.println("Enter an element to insert");
        ele=d.nextInt();
        q.insert(ele);
        break;

    case 2:
        q.delete();
        break;

    case 3:
        q.display();
        break;

    case 4:
        System.out.println("Exiting from the queue");
        break;
}
}while(ch!=4);
}
```

Output:

C:\Program Files (x86)\EditPlus 2\launcher.exe

Enter size of the queue:

5

1.Insert

2.Delete

3.Display

4.Exit

Enter your choice....

1

Enter an element to insert

10

Enter your choice....

1

Enter an element to insert

20

Enter your choice....

3

Elements in the queue are....

10 20

Enter your choice....

2

10 deleted from the queue

Enter your choice....

4

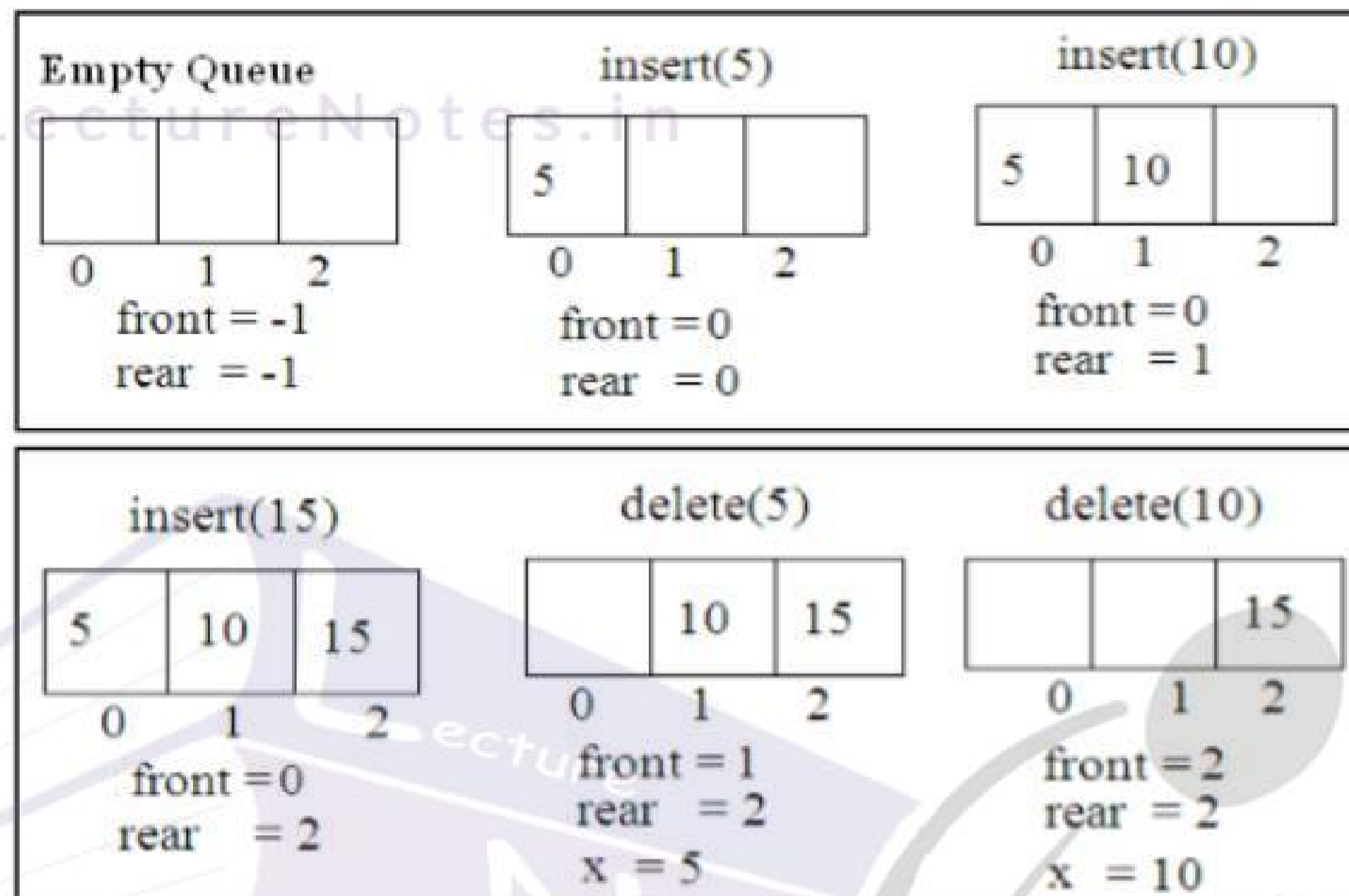
Exiting from the queue

Press any key to continue...

Circular Queue

Problem in Queue

For example take a queue with size =3.

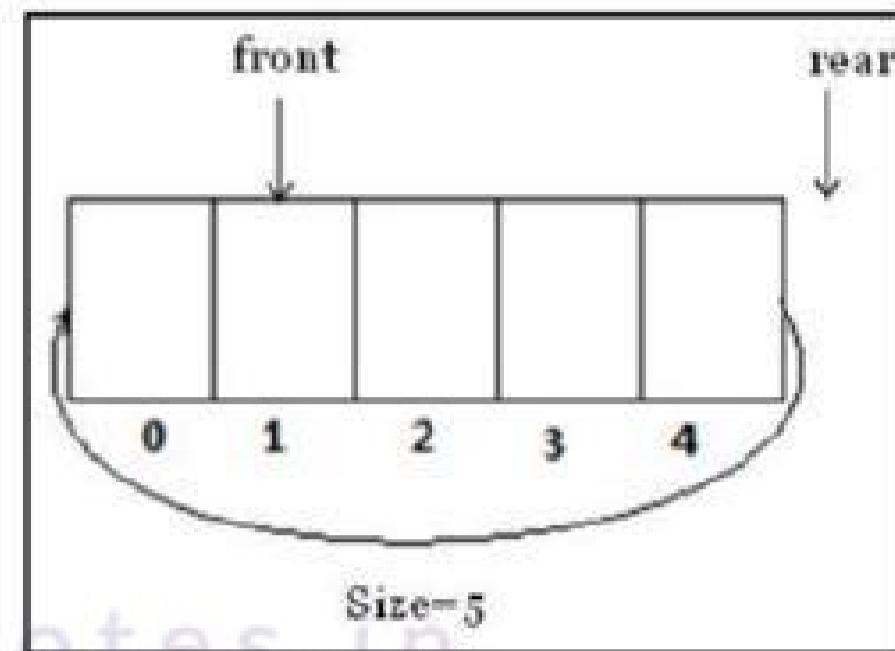


Here queue contains two empty locations but rear = size -1. i.e. queue is full and therefore insertion is not possible here. This problem is avoided by Circular Queue.

Circular Queue:

In circular Queue we insert the data element after rear = size-1. Here we move circularly to insert or delete data elements. It is simply called as circular array. In circular queue front and rear initially set to zero.

Here, we consider another variable called count to represent the total number of elements within the Circular Queue.



Initially count=0

Operations of Circular Queue:

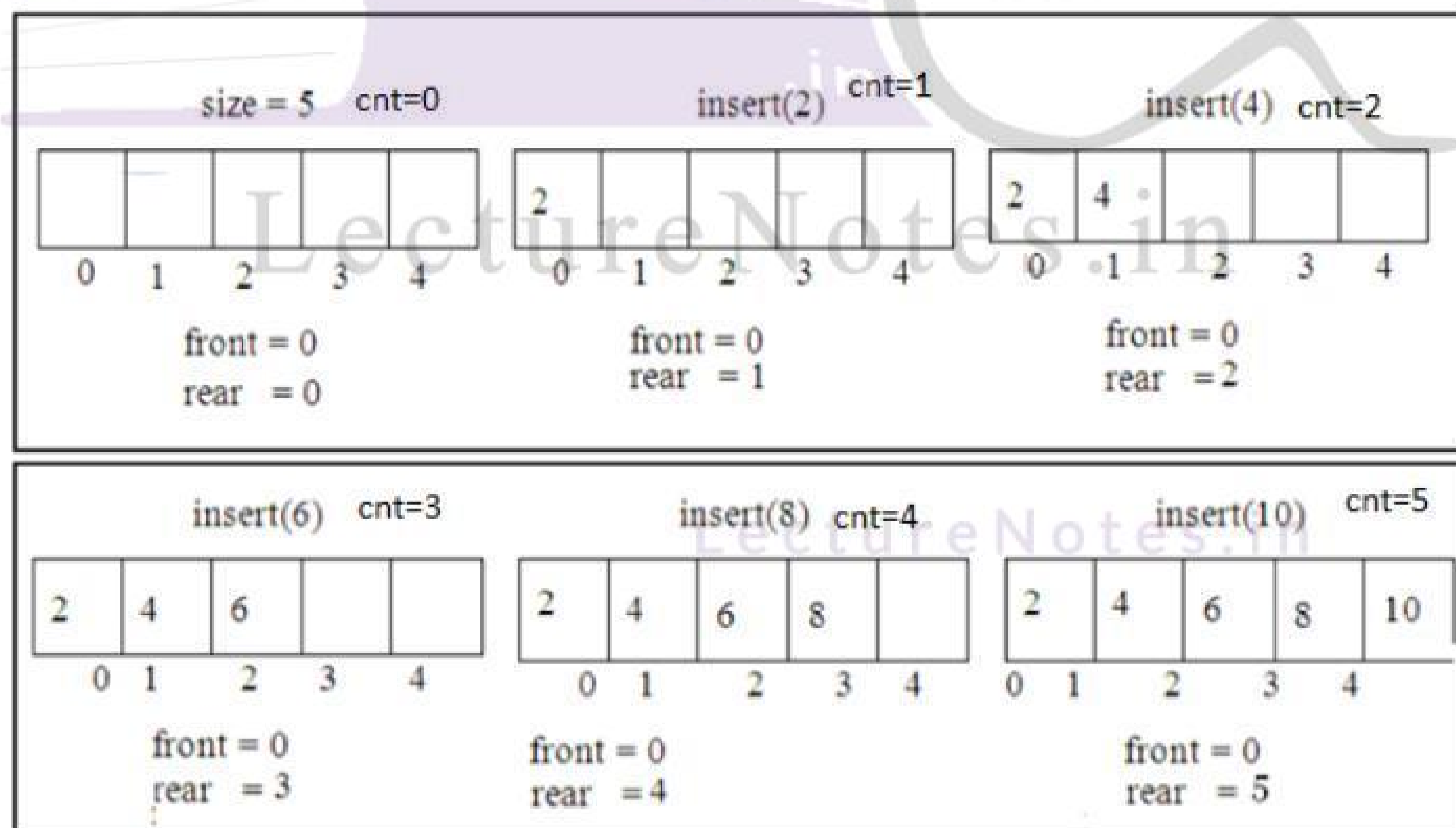
Create:

It is used to create empty circular queue.

Insert:

Before inserting the data element in Queue first of check the condition ($\text{count} == \text{size}$), if it is true the Queue is overflow (full). Otherwise insert the data element in that location and increment the rear. When an element is added to Circular Queue we increment the value of cnt by 1.

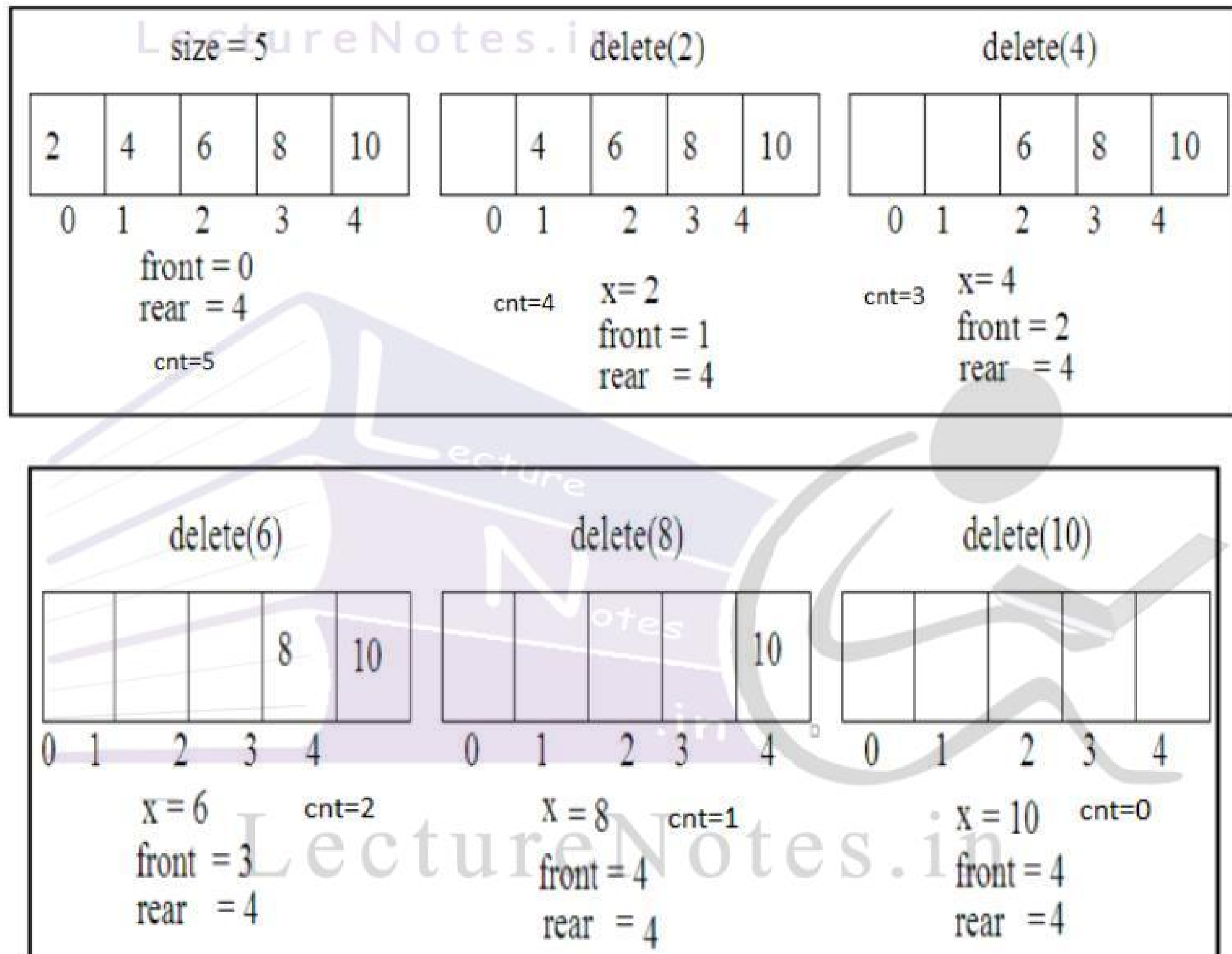
Example: Initially front and rear are set to '0'.



After inserting all the elements, $\text{cnt} = 5$ it means the queue is full.

Delete:

To delete data element from the circular queue first of all check the condition ($cnt == 0$), if it is true circular queue is underflow (empty), otherwise delete the data element in that location and increment the front. When an element is removed from Circular Queue we decrease the value of cnt by 1.

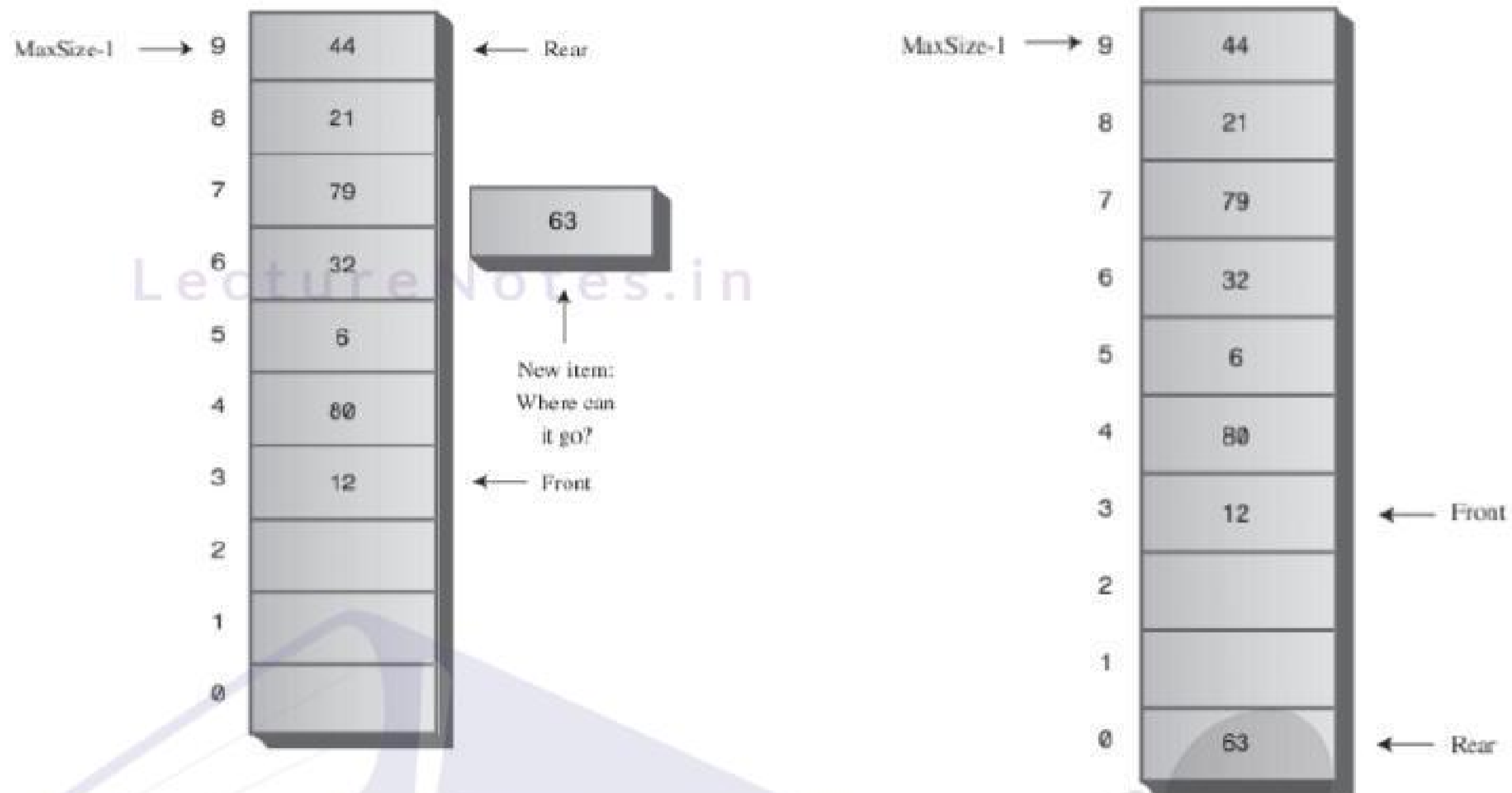


After deleting the entire elements $cnt == 0$; it means that queue is empty.

LectureNotes.in

Display: It is used to display the elements in the queue.

The following diagrams show how to insert data element in circular queue from the front end.



The following program illustrates the implementation of Circular Queue.

Aim: To write a java program to implement Circular Queue Data Structure.

Program: CQueueDemo.java

```
import java.io.*;
class CQueue
{
    DataInputStream d=new DataInputStream(System.in);
    int a[];
    int i,front,rear,max,ele,count;
    CQueue(int size)
    {
        max=size;
        front=0;
        rear=0;
        count=0;
        a=new int[max];
    }
    void insert()
    {
        if(count<max)
        {
            System.out.println("Enter the element to be added:");
```

```
        ele= Integer.parseInt(d.readLine());
        a[rear]=ele;
        rear++;
        count++;
    }
    else
    {
        System.out.println("QUEUE IS FULL");
    }
    if(front==max)
    {
        front=0;
    }
}

void delete()
{
    int ele;
    if(count!=0)
    {
        ele=a[front];
        System.out.println("The item deleted is:" + ele);
        front++;
        count--;
    }
    else
    {
        System.out.println("QUEUE IS EMPTY ");
    }
    if(rear==max)
    {
        rear=0;
    }
    if(front==max)
    {
        front=0;
    }
}

void display()
{
    int m=0;
    if(count==0)
    {
        System.out.println("QUEUE IS EMPTY ");
    }
}
```

```

        else
        {
            for(i=front;m<count;i++,m++)
                System.out.println(" "+a[i%max]);
        }
    }
}
class CQueueDemo
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int ele,ch,n;
        System.out.println("Enter the size of the queue...");
        n=d. Integer.parseInt(d.readLine());
        CQueue cq=new CQueue(n);
        do
        {
            System.out.println(" 1.Enqueue \n\n 2.Dequeue \n\n 3.Display \n\n 4.Exit");
            System.out.println("Enter the choice");
            ch= Integer.parseInt(d.readLine());
            switch (ch)
            {
                case 1:
                    cq.insert();
                    break;
                case 2:
                    cq.delete();
                    break;
                case 3:
                    cq.display();
                    break;
                case 4:
                    System.out.println("Exit from the CQueue");
                    break;
            }
        }while(ch!=4);
    }
}

```

Output:

C:\Program Files (x86)\EditPlus 2\launcher.exe

Enter the size of the queue...

3

1.Enqueue

2.Dequeue

3.Display

4.Exit

Enter the choice

1

Enter the element to be added:

10

1.Enqueue

2.Dequeue

3.Display

4.Exit

Enter the choice

1

Enter the element to be added:

20

1.Enqueue

2.Dequeue

3.Display

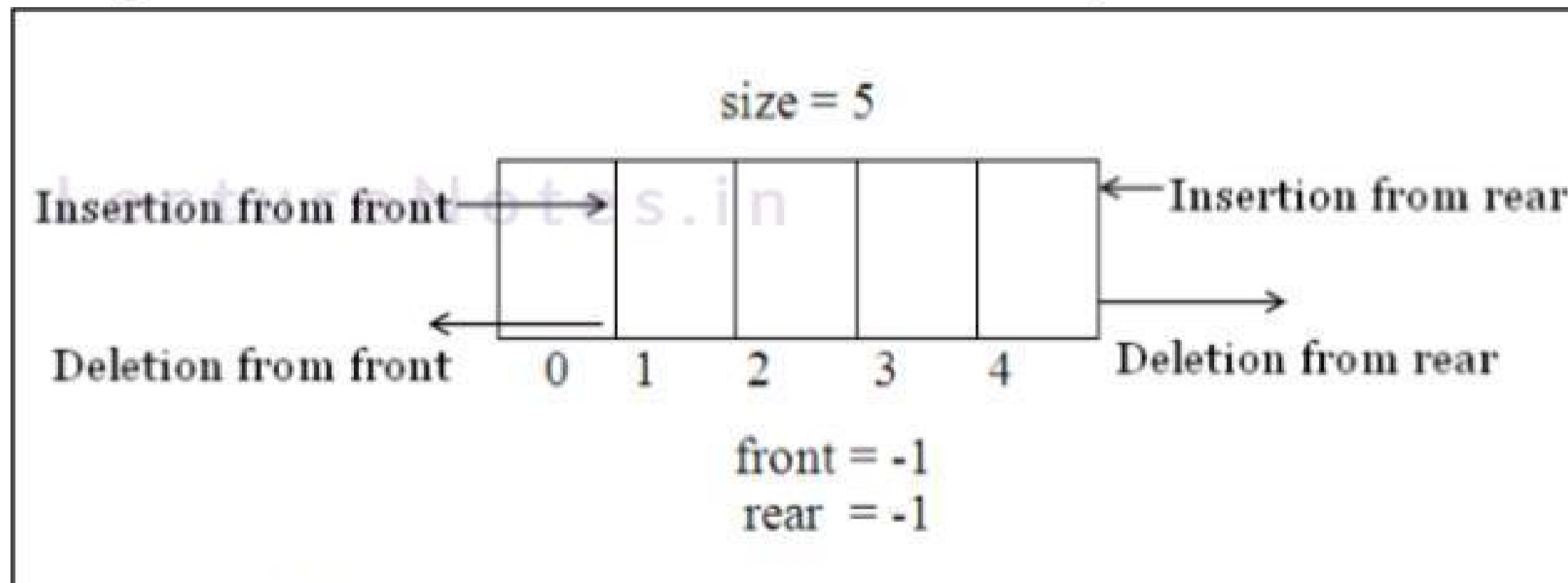
4.Exit

Enter the choice

1_

Double Ended Queue (Deque)

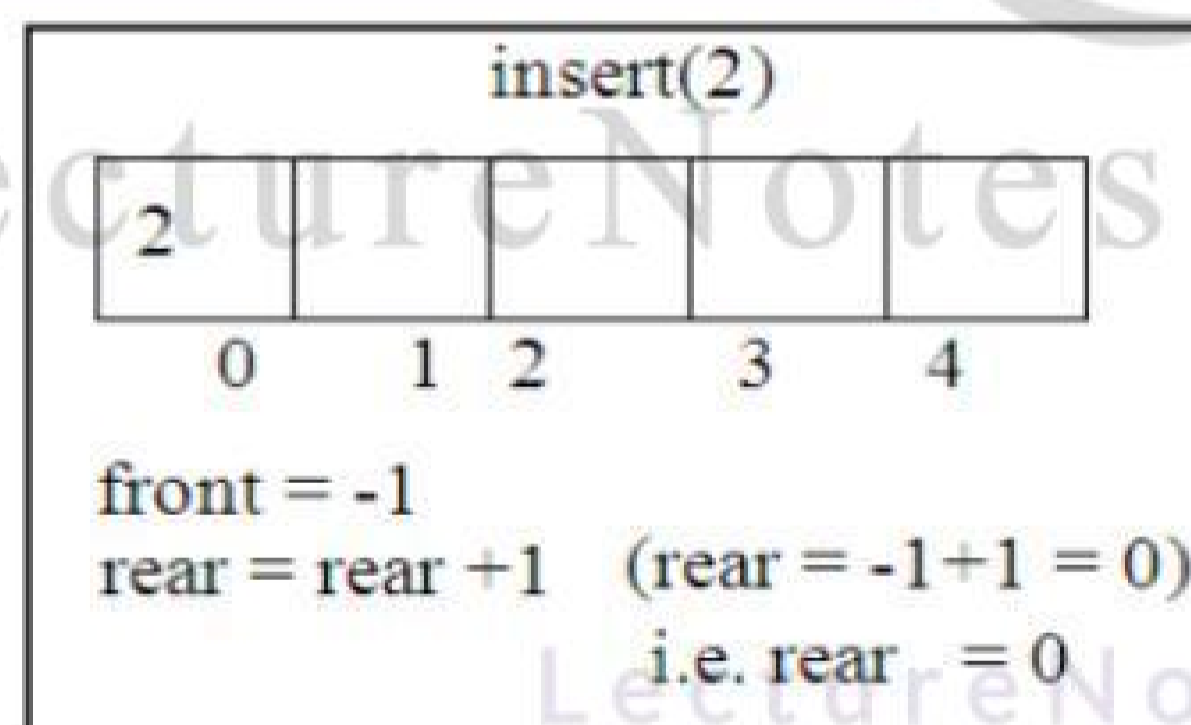
Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

**Operations of De-Queue:**

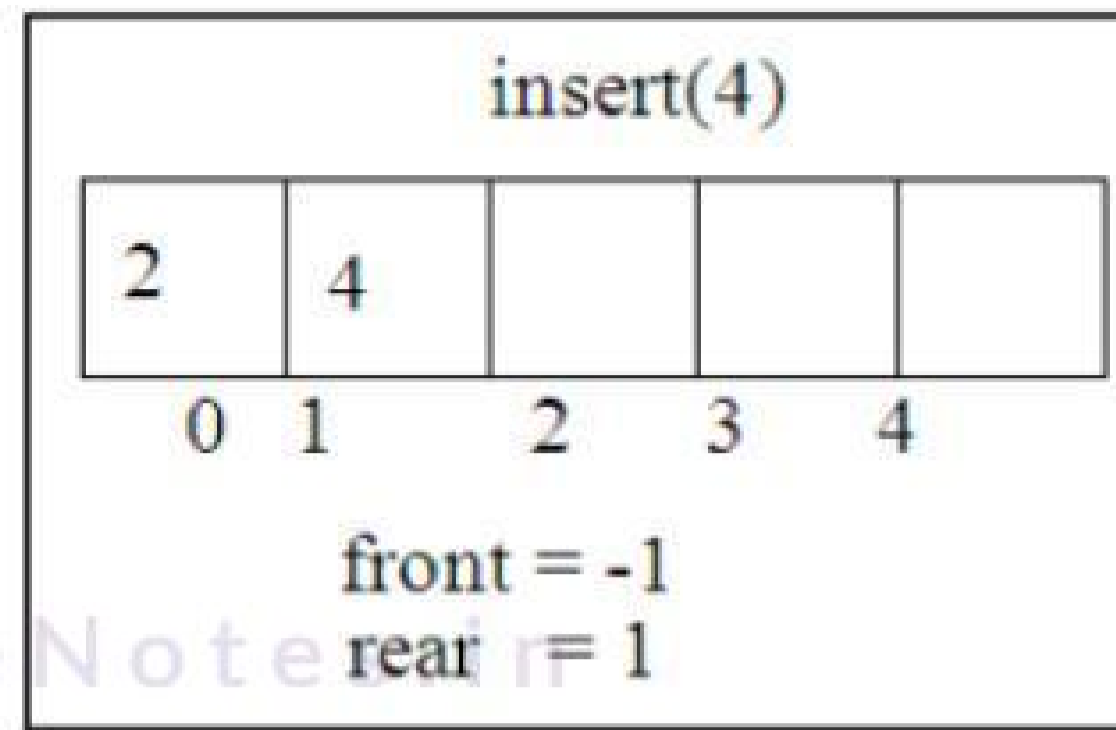
- ✚ Insertion from front
- ✚ Insertion from rear
- ✚ Deletion from front
- ✚ Deletion from rear

a). Insertion from front

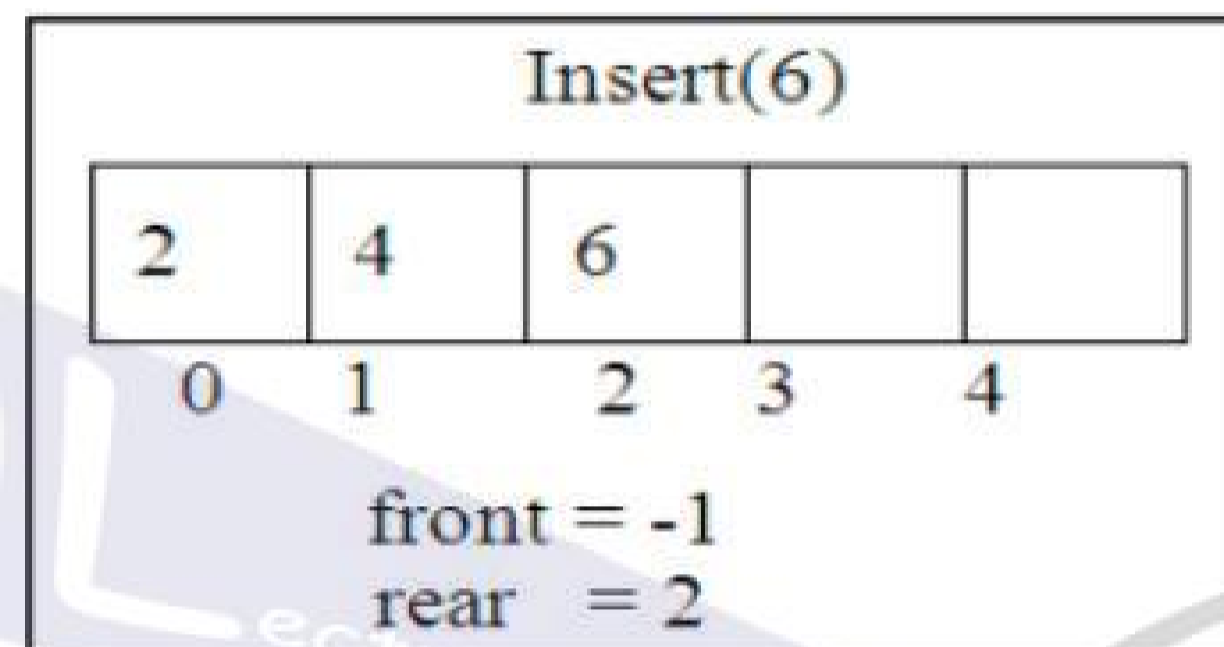
For example consider a queue with size = 5. Here queue is empty. Therefore insertion of front is identical to insertion of rear. For example the data element is inserting from rear end. i.e increment the rear and insert the data element in that location.

Insertion of (2)**Insertion of (4)**

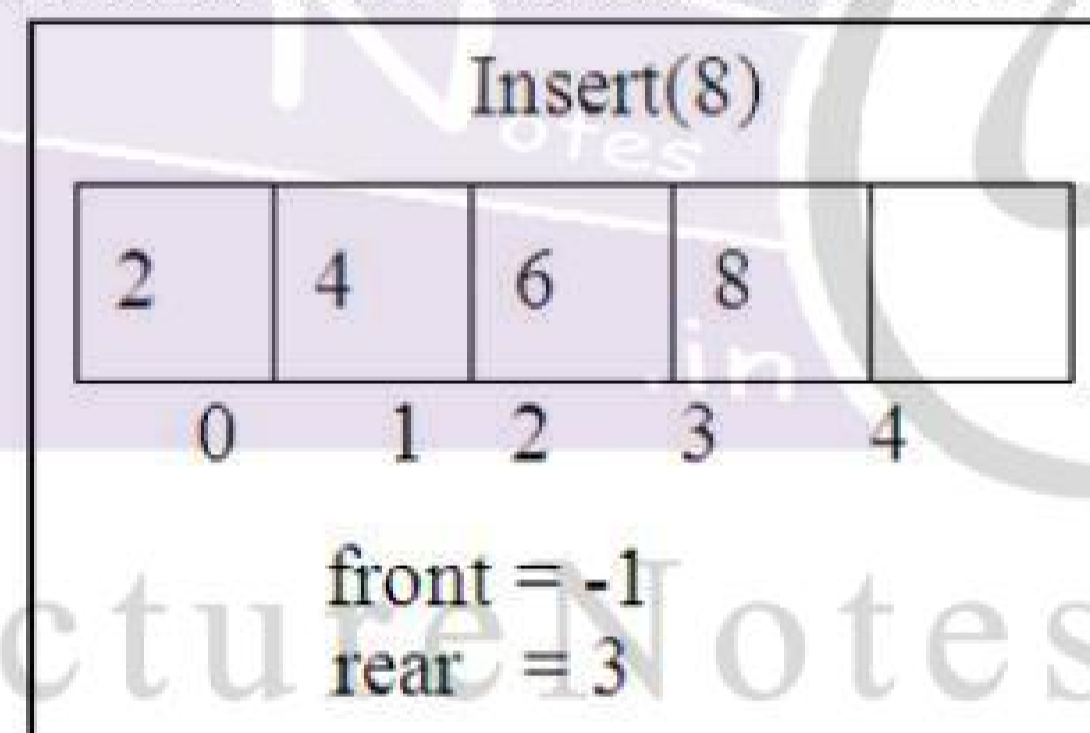
Here queue is not empty and front == -1. Therefore, insertion from front end is not possible. So insert the data element from the rear end. i.e. increment the rear and insert the data element in that location.

**Insertion of (6)**

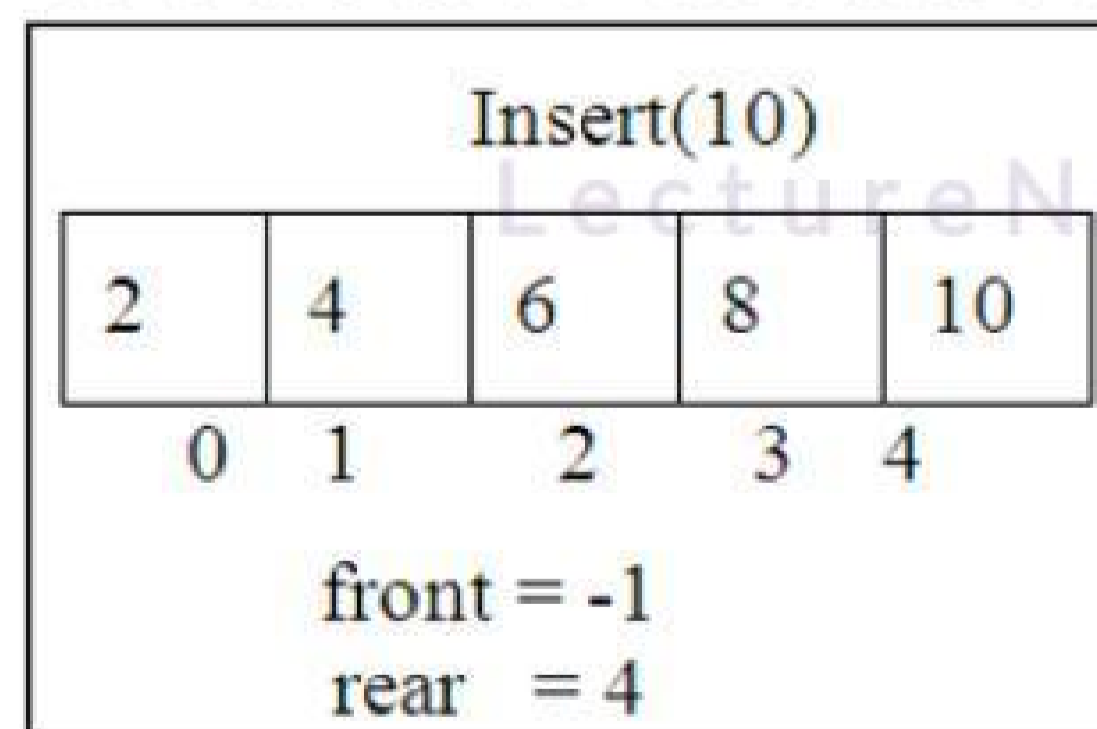
Here $\text{rear} \neq \text{size} - 1$. Increment the rear and insert the data element in that location.

**Insertion of (8)**

Here $\text{rear} \neq \text{size} - 1$. Increment the rear and insert the data element in that location.

**Insertion of (10)**

Here $\text{rear} \neq \text{size} - 1$. Increment the rear the insert the data element in that location.

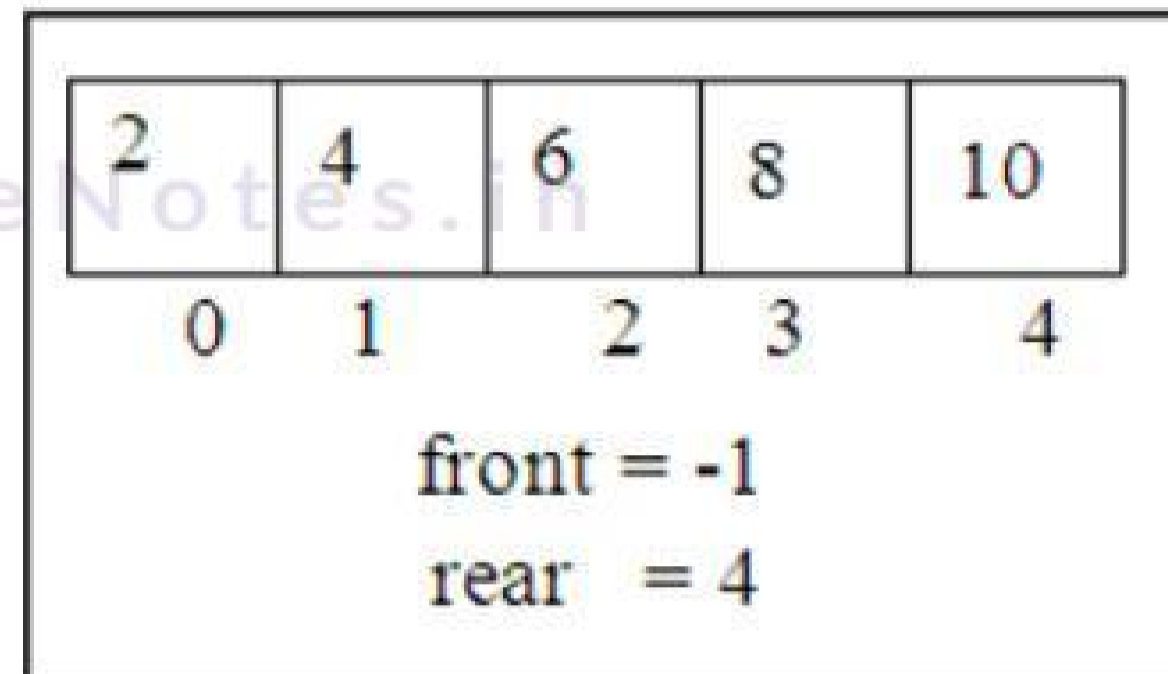


Insertion of (12)

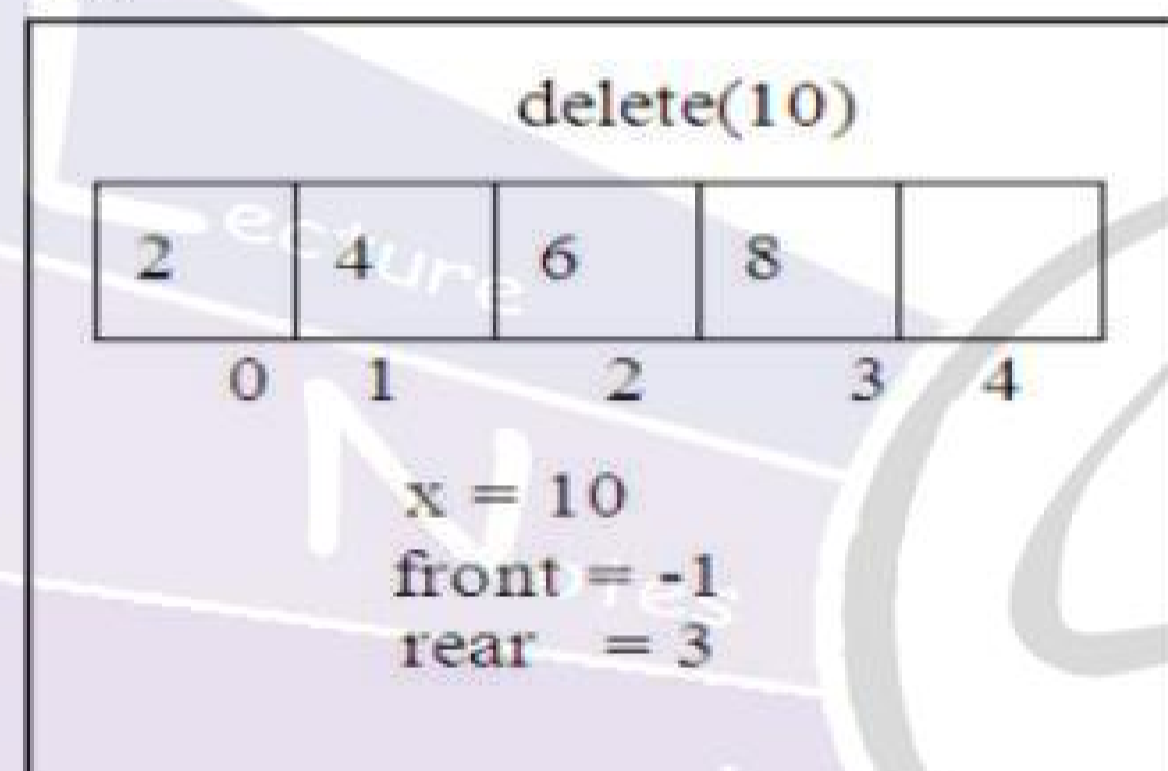
Here, $\text{rear} = \text{size} - 1$. Therefore De-Queue is overflow and also insertion from the front is not possible because queue is not empty and $\text{front} = -1$

Deletion from Rear:

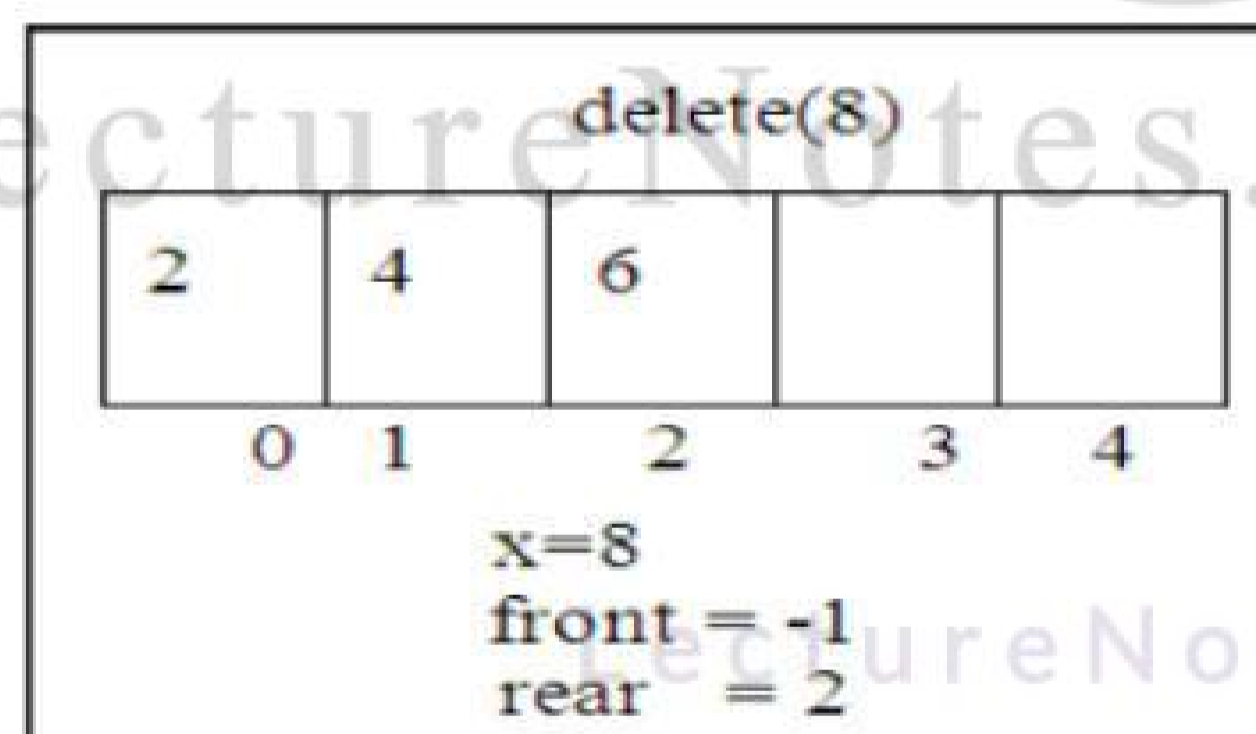
Consider a queue as follows;



Here $\text{front} \neq \text{rear}$. Deletion is possible from rear end. Therefore delete the data element '10' and decrement rear by '1'.

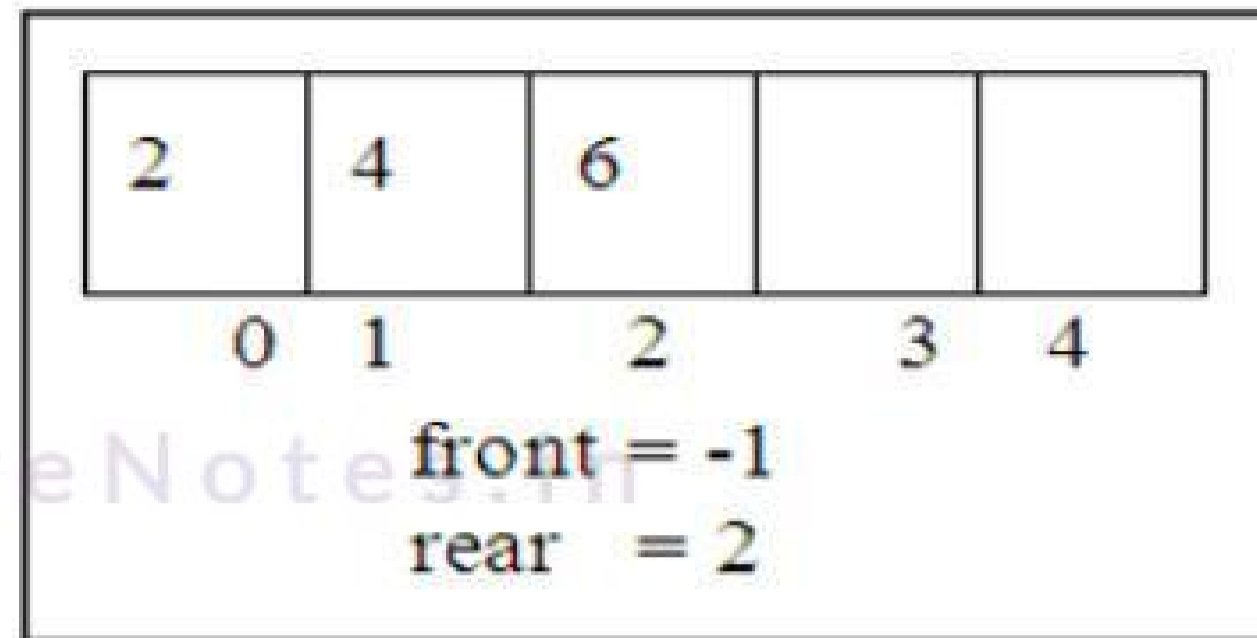


Again $\text{front} \neq \text{rear}$, delete the data element and decrement rear by '1'.

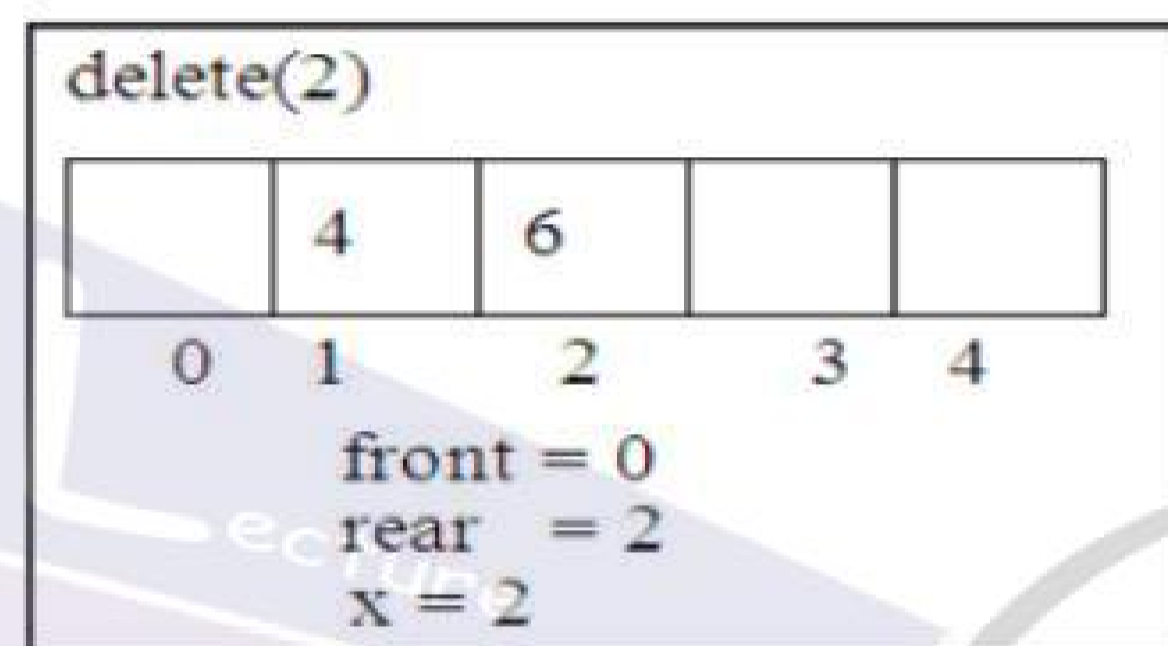


Deletion from front end:

Consider Queue as follows

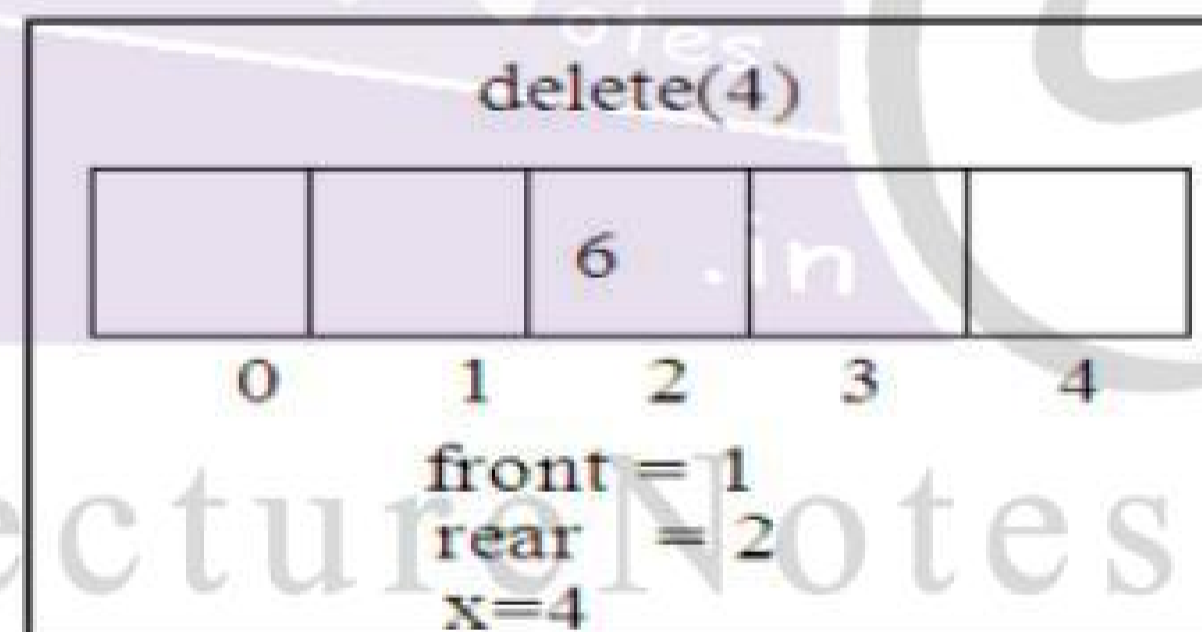


Here $\text{front} \neq \text{rear}$, deletion is possible from front end. So increment the front and delete the data element in that location.



Again $\text{front} \neq \text{rear}$

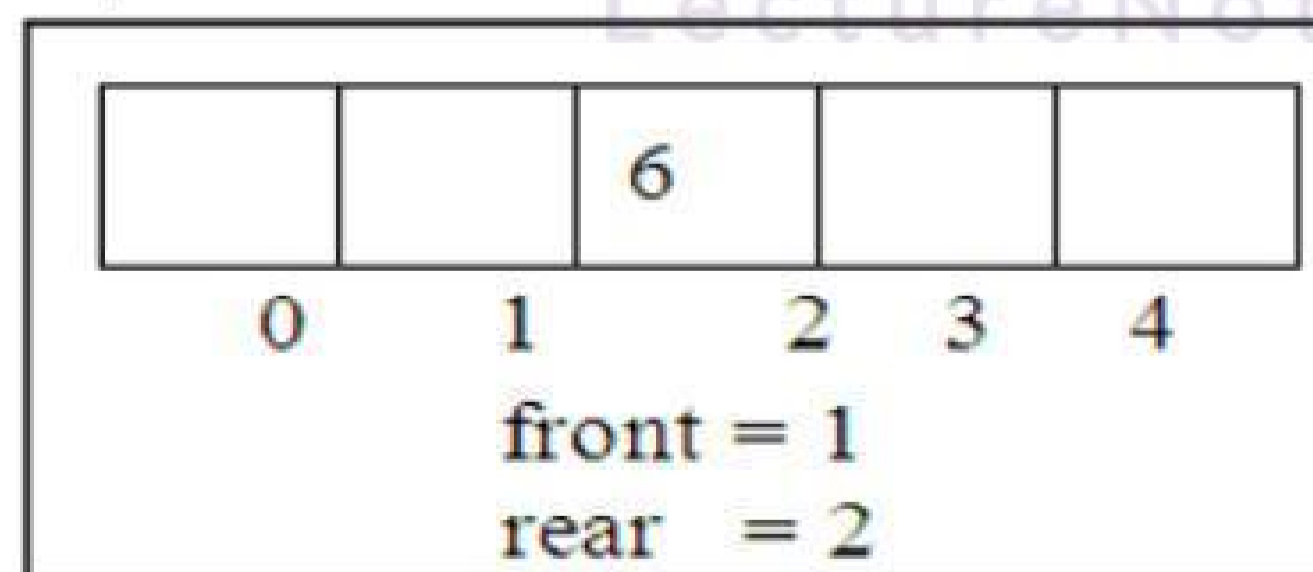
Increment the front and delete the data element in that location.



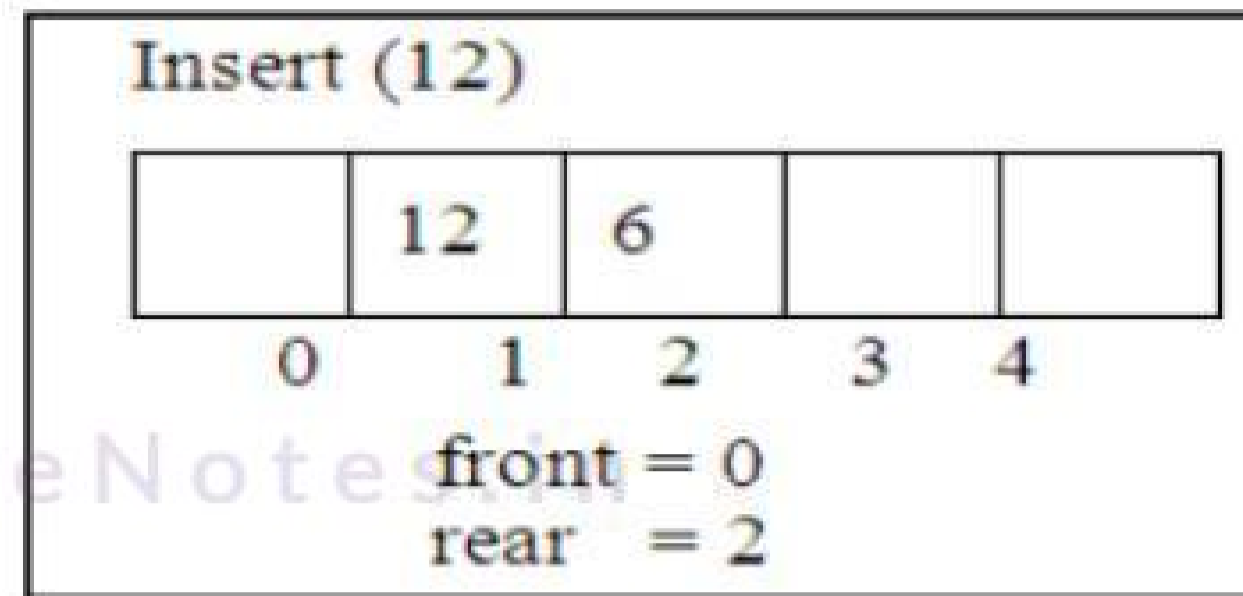
Here queue contains only one element and is deleted by using either end.

Insertion from front:

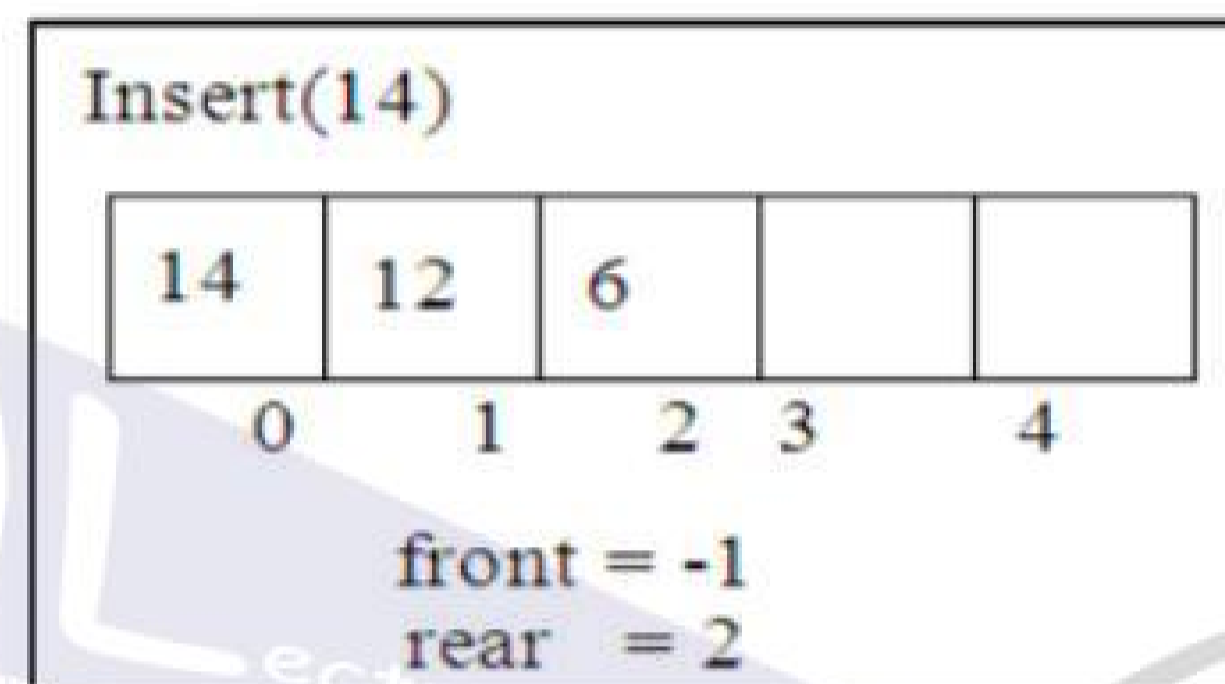
Consider a queue as follows;



Here queue is not empty and front is not equal to '-1'. So insertion from front is possible. Insert the element then decrement the front by '1'.



Again queue is not empty and front \neq -1. Insert the data element and decrement front by '1'.



Now further insertion is not possible from the front because front == -1.

Note:

Insertion from front is possible only in two cases.

1. If front \neq -1
2. We have to delete at least one element from the front end.

The following program illustrates the implementation of Double Ended Queue.

Aim: To write a java program to implement Double Ended Queue.

Program: DequeueDemo.java

```
import java.io.*;
class Dequeue
{
    int front,rear,max;
    int dque[];
    Dequeue(int size)
    {
        max=size;
        dque=new int[max];
        front=-1;
        rear=-1;
    }
    void insert_rear(int ele)
    {
```

```
    if(rear==max-1)
    {
        System.out.println("Insertion not posible\n");
    }
    else
    {
        rear++;
        dque[rear]=ele;
        System.out.println("Element " + ele + " inserted succusfully\n");
    }
}
void insert_front(int ele)
{
    if(front==1)
    {
        System.out.println("Insertion not posible\n");
    }
    else
    {
        dque[front]=ele;
        front--;
        System.out.println("Element " + ele + " inserted succesfully\n");
    }
}
void delete_front()
{
    int ele;
    if(front==rear)
    {
        System.out.println("Deletion is not possible");
    }
    else
    {
        front++;
        d=dque[front];
        System.out.println(ele+"element Deleted through front end from
        dequeue");
        if(front==rear)
            front=rear=1;
    }
}
void delete_rear()
{
    int ele;
    if(front==rear)
```

```

    {
        System.out.println("Deletion from rear end is not possible");
    }
    else
    {
        ele=dque[rear];
        rear--;
        System.out.println(ele+"element deleted through rear end from dequeue");
        if(front==rear)
            front=rear= -1;
    }
    return d;
}
void display()
{
    if(front==1 && rear==1)
    {
        System.out.println("\nDequeue is empty");
    }
    else
    {
        System.out.println("\nThe elements of dequeue are....");
        for(int i=front+1;i<=rear;i++)
        {
            System.out.print(dque[i] + "\t");
        }
    }
}
}
class DequeueDemo
{
    public static void main(String[] args)
    {
        DataInputStream d=new DataInputStream(System.in);
        int n,ch,ele;
        System.out.println("Enter size of the dequeue:");
        n=Integer.parseInt(d.readLine());
        Dequeue dq=new Dequeue(n);

        System.out.println("\n1.Insertion from front");
        System.out.println("\n2.Insertion from rear");
        System.out.println("\n3.Deletion from front");
        System.out.println("\n4.Deletion from rear");
        System.out.println("\n5.Display");
        System.out.println("\n6.Exit");
    }
}

```

```
do
{
    System.out.println("Enter your choice:");
    ch= Integer.parseInt(d.readLine());
    switch(ch)
    {
        case 1: System.out.println("Enter element to insert at front:");
                ele= Integer.parseInt(d.readLine());
                dq.insert_front(ele);
                break;
        case 2: System.out.println("Enter element to insert at rear:");
                ele= Integer.parseInt(d.readLine());
                dq.insert_rear(ele);
                break;
        case 3: dq.delete_front();
                break;
        case 4: dq.delete_rear();
                break;
        case 5: dq.display();
                break;
        case 6: System.exit("Exit from Dequeue");
                break;
    }
}while(ch!=6);
}
```

Output:

LectureNotes.in

```

C:\Program Files (x86)\EditPlus 2\launcher.exe
Enter size of the dequeue:
1
1.Insertion from front
2.Insertion from rear
3.Deletion from front
4.Deletion from rear
5.Display
6.Exit
Enter your choice:
1
Enter element to insert at front:
10
Insertion not posible
Enter your choice:
2
Enter element to insert at rear:
10
Element 10 inserted succusfully
Enter your choice:

```

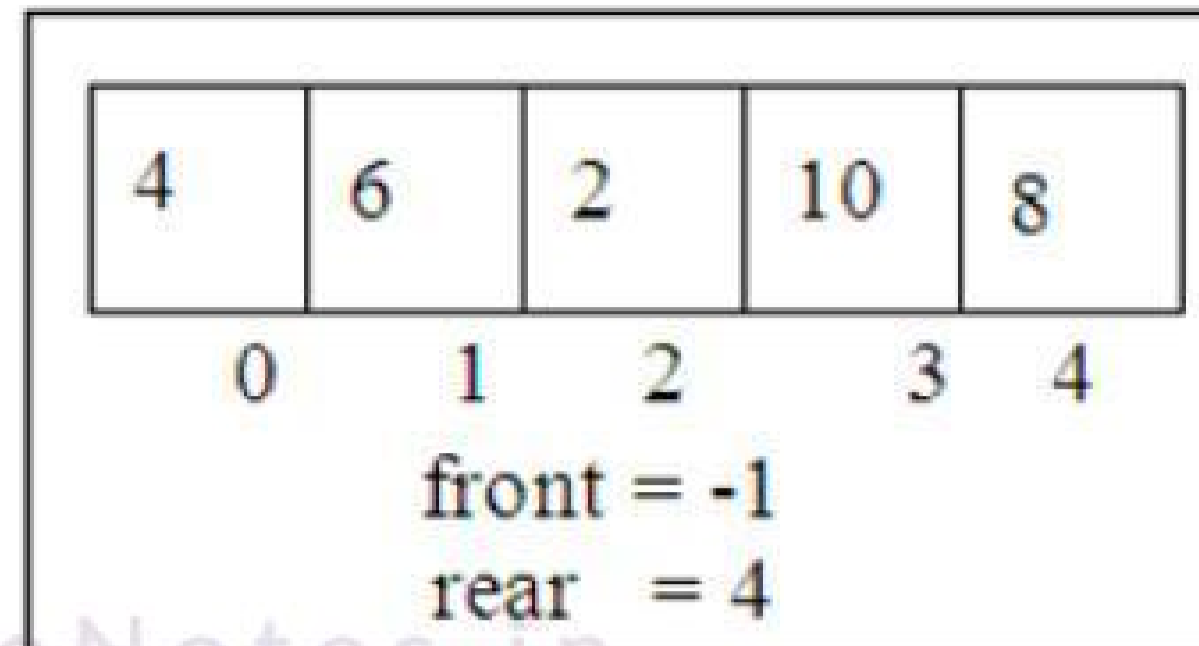
Priority Queue

Priority Queue is a queue, in this insertions are done as a normal queue and deletions are done according to some priority.

For example use the following two operations:

pq_mindele 0:-

Using this operation we find the minimum element in the queue and delete it from queue. For example queue size is 5 and initially front and rear set to '-1' and the following 5 elements are inserted;

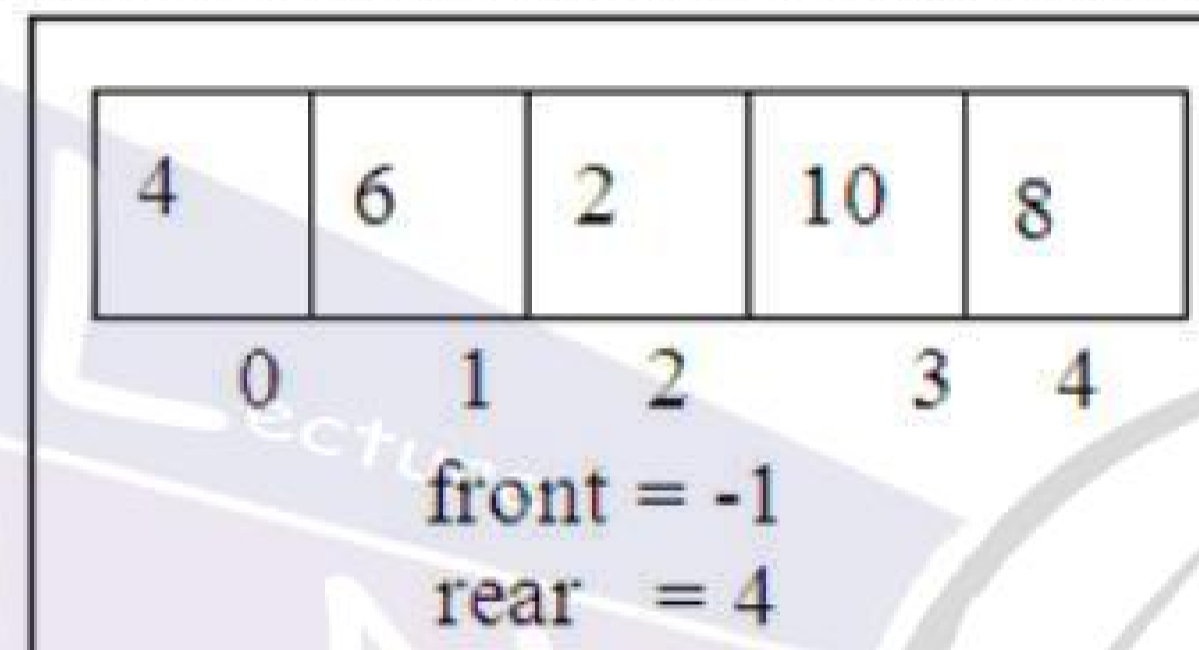


Now in this operation data element '2' is deleted and will produce the output as 4 6 10

8.

pq_maxdel ():-

Using this operation we find the maximum element in the queue and deleted it from the queue. For example consider queue size is 5 and the following 5 elements are inserted;



Now in this operation data element '10' is deleted and will produce the output as 4 6 2

8.

pq_min or pq_max contains two major issues.

- To identify minimum or maximum element.
- Shuffle the array.

After shuffling the array, the succeeding elements must be shifted to one position left and also the rear must be decremented by '1'.

The following is the java program that illustrates the Priority Queue Data Structure.

Aim: To write a java program that implements a Priority Queue Data Structure.

Program: PQueueDemo.java

```
import java.io.*;
class PQueue
{
    int que[],max,f,r;
    PQueue(int size)
    {
        max=size;
        f=1;
        r=1;
        que=new int[max];
    }
}
```

```
}
int find_min()
{
    int min;
    min=que[f];
    for(int i=f+1;i<=r;i++)
    {
        if(min>que[i])
        {
            min=que[i];
        }
    }
    return min;
}
int find_max()
{
    int max;
    max=que[f];
    for(int i=f+1;i<=r;i++)
    {
        if(max<que[i])
        {
            max=que[i];
        }
    }
    return max;
}
void insert(int ele)
{
    if(r==max-1)
    {
        System.out.println("Queue Overflow");
    }
    else
    {
        r++;
        que[r]=ele;
        System.out.println(ele+"element inserted successfully");
        if(f==1)
        {
            f=0;
        }
    }
}
void delete(int x)
{
```



```

int d;
if(f==1)
{
    System.out.println("Queue is Empty");
}
else
{
    if(x==2)
    {
        d=find_min();
        System.out.println("Minimum element is"+d);
    }
    else
    {
        d=find_max();
        System.out.println("Maximum element is"+d);
    }
    for(int i=f; i<r; i++)
    {
        if(que[i] == d)
        {
            for(int j=i; j<=r-1; j++)
            {
                que[j] = que[j+1];
            }
            break;
        }
    }
}

if(f==r)
{
    f=r-1;
}
else
{
    r--;
}

return d;
}
void display()
{
    if(f==1)

```

```

    {
        System.out.println("Queue is empty");
    }
    else
    {
        System.out.println("Elements in the queue are.....");
        for(int i=f;i<=r;i++)
        {
            System.out.print("\t" + que[i]);
        }
        System.out.println("");
    }
}
}
}
class PQueueDemo
{
    public static void main(String args[])throws IOException
    {
        int ch,ele,size;
        DataInputStream d=new DataInputStream(System.in);
        System.out.println("Enter size of the queue:");
        size= Integer.parseInt(d.readLine());
        PQueue p=new PQueueDemo(size);

        do
        {
            System.out.println("\n1.Insert\n");
            System.out.println("2.Delete Minimum\n");
            System.out.println("3.Delete Maximum\n");
            System.out.println("4.Display\n");
            System.out.println("5.Exit\n");

            System.out.println("Enter your choice....");
            ch=sc.nextInt();

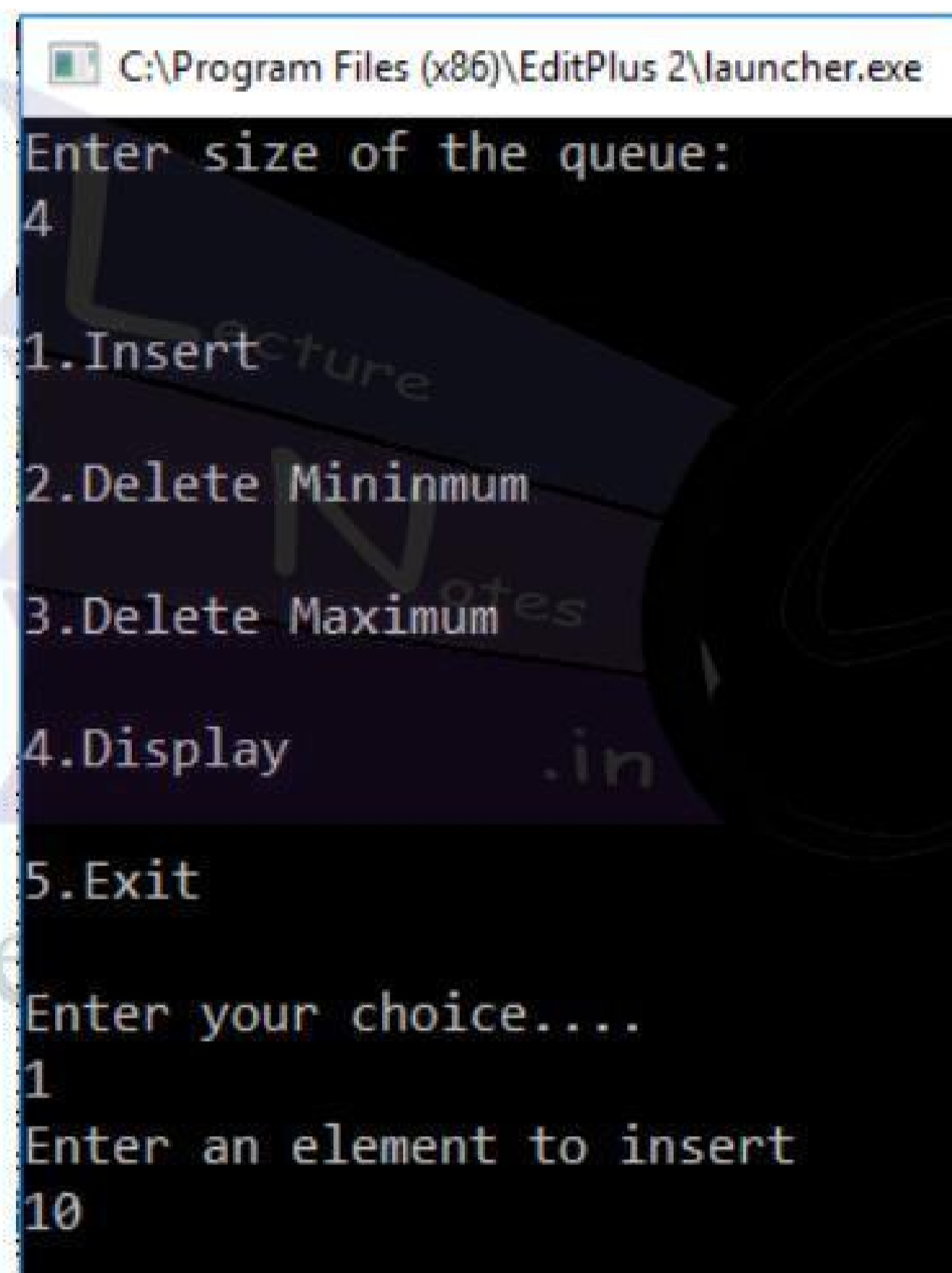
            switch(ch)
            {
                case 1:
                    System.out.println("Enter an element to insert");
                    ele= Integer.parseInt(d.readLine());
                    p.insert(ele);
                    break;

                case 2:
                    p.delete(ch);
                    break;

                case 3:

```

```
        p.delete(ch);
        break;
    case 4:
        p.display();
        break;
    case 5:
        System.out.println("Exiting from the queue");
        break;
    }
}while(ch!=5);
}
```

Output:

```
C:\Program Files (x86)\EditPlus 2\launcher.exe
Enter size of the queue:
4
1.Insert
2.Delete Mininum
3.Delete Maximum
4.Display
5.Exit
Enter your choice....
1
Enter an element to insert
10
```

SEARCHING'S

What is Search

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear Search Algorithm (Sequential Search Algorithm)

Linear search algorithm finds given element in a list of elements with $O(n)$ time complexity where n is total number of elements in the list.

Linear search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the first element in the list.
- **Step 3:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 4:** If both are not matching, then compare search element with the next element in the list.
- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Example

Consider the following list of element and search element...

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

search element **12**

Step 1:

search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

LectureNotes.in
12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

LectureNotes.in 12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are matching. So we stop comparing and display element found at index 5.

The following is a java program that implements Linear Search algorithm.

Aim: To write a java program that implements Linear Search algorithm.

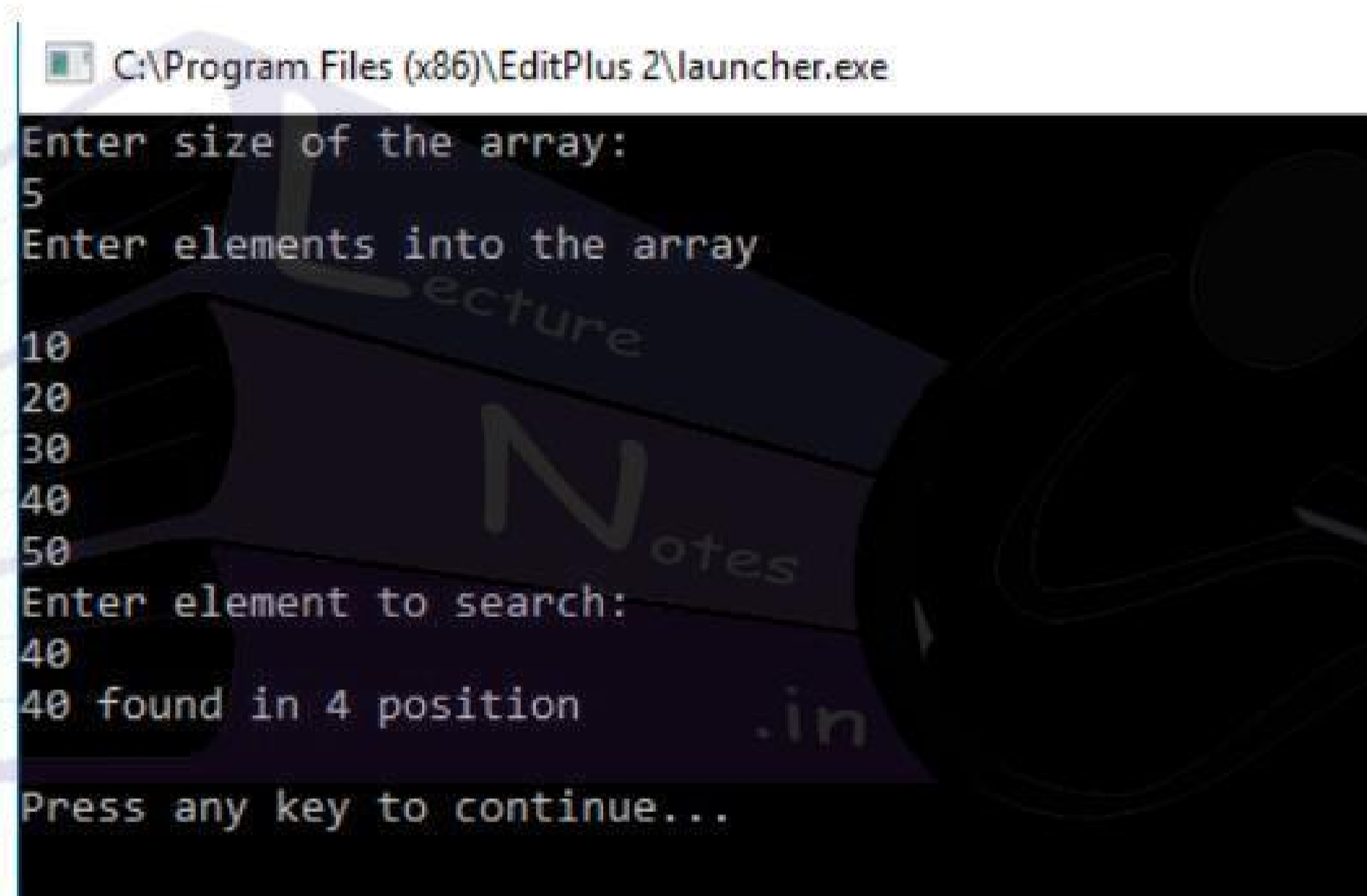
Program: SequentialSearch.java

```
import java.io.*;
class SequentialSearch
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int n,i,flag=0,key;
        System.out.println("Enter size of the array");
        n=Integer.parseInt(d.readLine());
```

```
int a[]=new int[n];  
System.out.println("Enter elements into the array");  
for(i=0;i<n;i++)  
{  
    a[i]=Integer.parseInt(d.readLine());  
}  
System.out.println("the array elements are....");  
for(i=0;i<n;i++)  
{  
    System.out.println(a[i]);  
}  
System.out.println("Enter element to search");  
key=Integer.parseInt(d.readLine());  
for(i=0;i<n;i++)  
{  
    if(key==a[i])  
    {  
        flag=1;  
        break;  
    }  
}  
if(flag==1)  
{  
    System.out.println(key+" found in" +(i+1)+"position");  
}
```

```
    else
    {
        System.out.println(key+" not found");
    }
}
}
}
```

Output:



```
C:\Program Files (x86)\EditPlus 2\launcher.exe
Enter size of the array:
5
Enter elements into the array
10
20
30
40
50
Enter element to search:
40
40 found in 4 position
Press any key to continue...
```

Binary Search Algorithm

Binary search algorithm finds given element in a list of elements with $O(\log n)$ time complexity where n is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in an order. The binary search cannot be used for list of element which are in random order.

Binary search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Find the middle element in the sorted list

- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 5:** If both are not matching, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sub list of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sub list of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Example

Consider the following list of element and search element...

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

search element **12**

LectureNotes.in

Step 1:

search element (12) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
					12				

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (12) is compared with middle element (12)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
		12							

Both are matching. So the result is "Element found at index 1"

Now consider another search element 80

Step 1:

search element (80) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

LectureNotes.in 80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (80) is compared with middle element (65)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 3:

search element (80) is compared with middle element (80)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

80

Both are not matching. So the result is "Element found at index 7"

The following is the java program that implements binary search algorithm.

Aim: To write a java program that implements Binary Search algorithm.

Program: BinarySearch.java

```
import java.io.*;

class Binarysearch
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int n,i,key,low,high,mid;
        System.out.println("Enter size of the array");
        n=Integer.parseInt(d.readLine());
        int a[]=new int[n];
        System.out.println("Enter elements into the array");
        for(i=0;i<n;i++)
        {
            a[i]=Integer.parseInt(d.readLine());
        }
        System.out.println("the array elements are");
        for(i=0;i<n;i++)
        {
            System.out.println(a[i]);
        }
        System.out.println("Enter element to search");
        key=Integer.parseInt(d.readLine());
```

```
low=0;
high=n-1;
mid=(low+high)/2;
while(low<high)
{
    if(key==a[mid])
        break;
    else if(key>a[mid])
        low=mid+1;
    else
        high=mid-1;
    mid=(low+high)/2;
}
if(key==a[mid])
{
    System.out.println(" found at" +(mid+1)+"position");
}
else
{
    System.out.println("element not found");
}
}
```

Output:

```
C:\Program Files (x86)\EditPlus 2\launcher.exe
Enter size of the array:
5
Enter elements into the array
10
20
30
40
50
Enter element to search:
50
50 found at 5 Position
Press any key to continue...
-
```



LectureNotes.in

LectureNotes.in

UNIT - V

Sorting and Searching: Selection, Insertion, Bubble, Merge, Quick, Heap sort, Sequential and Binary Searching.

LectureNotes.in

SORTING S

Definition of Sorting:

Sorting is to organize a collection of data elements based on the order of a comparable property of each element.

There are three concepts in this definition:

Data element: A unit of information.

Comparable property: A property in each element that can be used to compare element A with element B. The comparison must give one of three possible results:

1. A is greater than B;
2. A is equal to B;
3. A is less than B.

Collection: Sorting works on a collection of data elements, not on a single data element.

Bubble Sort

Bubble sort is also known as exchange sort. Bubble sort is a simplest sorting algorithm. In bubble sort algorithm array is traversed from 0 to the length-1 index of the array. In this we compared one element to the next element, if the first element is greater than the second element we simply swap them. In other words, bubble sorting algorithm compare two values and put the largest value at largest index.

The algorithm follow the same steps repeatedly until the values of array is sorted. In worst-case the complexity of bubble sort is $O(n^2)$ and in best-case the complexity of bubble sort is $\Omega(n)$.

In our example we are taking the following array values

12 9 4 99 120 1 3 10

The basic steps followed by algorithm:-

In the first step compare first two values 12 and 9.

12 **9** 4 99 120 1 3 10

As $12 > 9$ then we have to swap these values

Then the new sequence will be

9 **12** 4 99 120 1 3 10

In next step take next two values 12 and 4

9 **12** **4** 99 120 1 3 10

Compare these two values .As $12 > 4$ then we have to swap these values.

Then the new sequence will be

9 4 **12** **99** 120 1 3 10

We have to follow similar steps up to end of array. e.g.

9 4 12 **99** **120** 1 3 10

9 4 12 99 **120** 1 3 10

9 4 12 99 1 **120** **3** 10

9 4 12 99 1 3 **120** **10**

9 4 12 99 1 3 10 120

After completing the first iteration the first largest element i.e., 120 will be placed in its proper position. Similarly, in each iteration the next highest value will be placed in its proper position.

When we reached at last index, then restart same steps until the data is sorted.

The output of this example will be:

1 3 4 9 10 12 99 120

The following is the program illustrates the bubble sort.

Aim: To write a java program to implement the bubble sort algorithm.

//BubbleSort.java

```
import java.io.*;

class BubbleSort
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int n,i,j,temp;
        System.out.println("enter the size of an array");
        n=Integer.parseInt(d.readLine());
        int a[]=new int[n];
        System.out.println("enter the array elements");
        for(i=0;i<n;i++)
        {
            a[i]=Integer.parseInt(d.readLine());
        }
        System.out.println("before sorting the array elements are");
        for(i=0;i<n;i++)
        {
            System.out.println(a[i]);
        }
    }
}
```

```
for(i=n-1;i>0;i--)  
{  
    for(j=0;j<i;j++)  
    {  
        if(a[j]>a[j+1])  
        {  
            temp=a[j];  
            a[j]=a[j+1];  
            a[j+1]=temp;  
        }  
    }  
}  
System.out.println("after sorting the array elements are");  
for(i=0;i<n;i++)  
{  
    System.out.println(a[i]);  
}  
}
```

Output:

```

Command Prompt

E:\Naresh\DS Programs\Sortings>javac BubbleSort.java

E:\Naresh\DS Programs\Sortings>java BubbleSort
Enter size of the array:
5
Enter elements into the array

50
40
30
20
10

as before sorting are....

50    40    30    20    10

as after sorting are...

10    20    30    40    50

E:\Naresh\DS Programs\Sortings>

```

Selection Sort

In selection sorting algorithm, find the minimum value in the array then swap it first position. In next step leave the first value and find the minimum value within remaining values. Then swap it with the value of minimum index position. Sort the remaining values by using same steps.

The complexity of selection sort algorithm is in worst-case, average-case, and best case run-time of $\Theta(n^2)$, assuming that comparisons can be done in constant time.

The following is an example.....

Consider the array 'a' of size 5 with elements as follows.....

```
a[5]={50,40,30,20,10};
```

First Pass

At beginning the array elements are

50 **40** 30 20 10

Compare 50 and 40, i.e., $50 > 40$ if it true swap these two. After swapping the array elements are as follows....

40 50 **30** 20 10

Now compare 40 and 30, i.e., $40 > 30$. Swap these two. After swapping the array is as follows

30 50 40 **20** 10

Now compare 30 and 20, i.e., $30 > 20$. Swap these two. After swapping the array is as follows

20 50 40 30 **10**

Now compare 20 and 10, i.e., $20 > 10$. Swap these two. After swapping the array is as follows

10 50 40 30 20

Therefore after completing the first pass we are getting the array elements

10 50 40 30 20

By observing the above elements, it is cleared that the smallest element i.e., 10 will be placed in its proper position. So, similarly in every pass one smallest element will be placed in its proper position.

After completing the second pass the array elements are as follows.....

10 20 50 40 30

After completing the third pass the array elements are as follows.....

10 20 30 50 40

After completing the fourth pass the array elements are as follows.....

10 20 30 40 50

The following program illustrates the concept of implementation of Selection sort in java.

Aim: To write a java program to implement the selection sort algorithm in java.

Program://SelectionSort.java

```
import java.io.*;
class SelectionSort
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int n,i,j,temp;
        System.out.println("enter the size of an array");
        n=Integer.parseInt(d.readLine());
        int a[]=new int[n];
        System.out.println("enter the array elements");
        for(i=0;i<n;i++)
        {
            a[i]=Integer.parseInt(d.readLine());
        }
        System.out.println("before sorting the array elements are");
        for(i=0;i<n;i++)
        {
            System.out.println(a[i]);
        }
        for(i=0;i<n;i++)
```

```
{
    for(j=i+1;j<n;j++)
    {
        if(a[i]>a[j])
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }
    System.out.println("after sorting the array elements are");
    for(i=0;i<n;i++)
    {
        System.out.println(a[i]);
    }
}
}
```

Output:

```
E:\Naresh\DS Programs\Sortings>javac SelectionSort.java
```

```
E:\Naresh\DS Programs\Sortings>java SelectionSort
```

```
Enter size of the array:
```

```
5
```

```
Enter elements into the array
```

```
50
```

```
40
```

```
30
```

```
20
```

```
10
```

```
The elements before the sorting are....
```

```
50 40 30 20 10
```

```
The elements after the sorting are....
```

```
10 20 30 40 50
```

```
E:\Naresh\DS Programs\Sortings>
```

Insertion Sort

Insertion sorting algorithm is similar to bubble sort. But insertion sort is more efficient than bubble sort because in insertion sort the elements comparisons are less as compare to bubble sort.

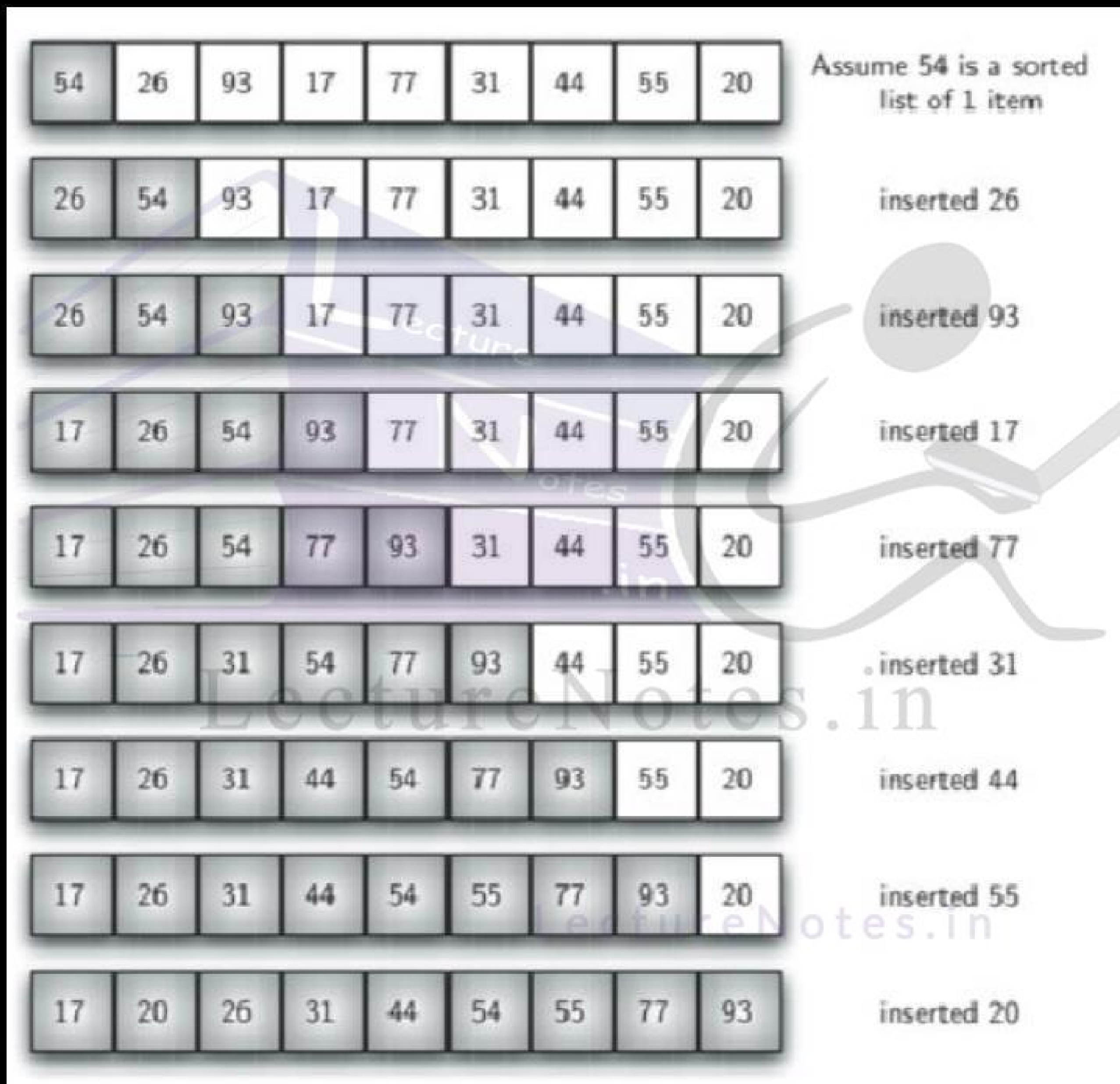
- This is implemented by inserting a particular element at the appropriate position.
- In this method the first iteration starts with the comparison of 1st element with 0th element.
- In the second iteration, 2nd element is compared with 0th and 1st elements.
- In general, in every iteration the element is compared with all the elements present before it.

During the comparison if it is found that the element can be inserted at a suitable position then space is created for it by shifting the other elements one position to the right and insertion the left element at the suitable position.

Positive feature of insertion sorting:

1. It is simple to implement
2. It is efficient on (quite) small data values
3. It is efficient on data sets which are already nearly sorted.

The complexity of insertion sorting is $O(n)$ at best case of an already sorted array and $O(n^2)$ at worst case .



The following program illustrates the implementation of Insertion sort in java.

Aim: To write a java program to implement the Insertion Sort algorithm.

Program:// InsertionSort.java

```
import java.io.*;

class InsertionSort
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int n,i,j,temp;
        System.out.println("enter the size of an array");
        n=Integer.parseInt(d.readLine());
        int a[]=new int[n];
        System.out.println("enter the array elements");
        for(i=0;i<n;i++)
        {
            a[i]=Integer.parseInt(d.readLine());
        }
        System.out.println("before sorting the array elements are");
        for(i=0;i<n;i++)
        {
            System.out.println(a[i]);
        }
    }
}
```

```
for(i=1;i<n;i++)
{
    j=i-1;
    while((j>=0)&&(a[j]>a[j+1]))
    {
        temp=a[j];
        a[j]=a[j+1];
        a[j+1]=temp;
        j=j-1;
    }
}
System.out.println("after sorting the array elements are");
for(i=0;i<n;i++)
{
    System.out.println(a[i]);
}
}
```

Output:

```

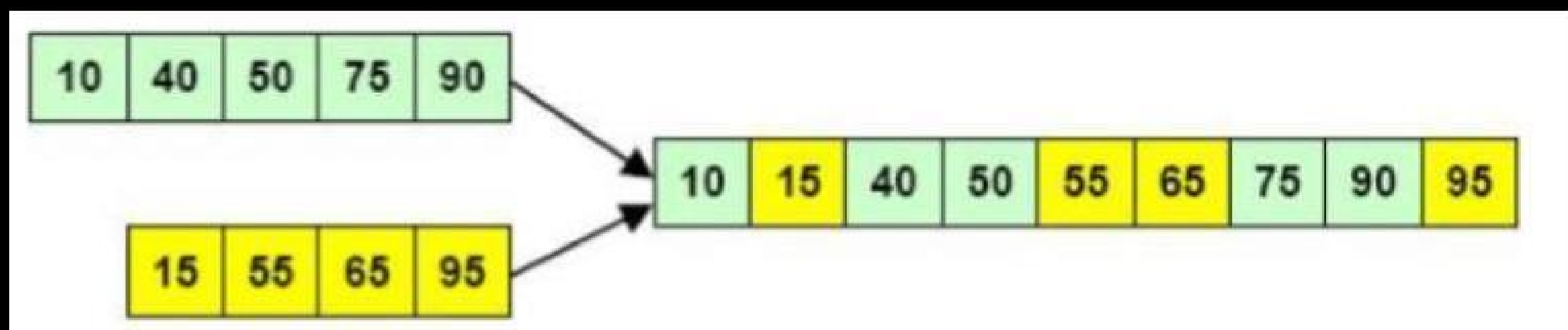
E:\Naresh\DS Programs\Sortings>javac InsertionSort.java
E:\Naresh\DS Programs\Sortings>java InsertionSort
Enter size of the array:
5
Enter elements into the array
50
40
30
20
10

as before sorting are....
50 40 30 20 10
Elements after asorting are...
10 20 30 40 50
E:\Naresh\DS Programs\Sortings>

```

Merge Sort

Merging is the combination of two or more sorted arrays into a single sorted array. Following figure illustrates the basic, two-way merge operation. In a two-way merge, two sorted sequences are merged into one.



The time complexity for merge sort is $O(n \log n)$.

The following program illustrates the implementation of Merge Sort in java.

Aim: To write a java program to implement the Merge Sort Algorithm.

Program://MergeSort.java

```
import java.io.*;

class Mergesort
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int m,n,i,j,k,temp;

        System.out.println("enter the size of the first array");
        m=Integer.parseInt(d.readLine());
        int a[]=new int[m];
        System.out.println("enter the elements into the first array");
        for(i=0;i<m;i++)
        {
            a[i]=Integer.parseInt(d.readLine());
        }
        System.out.println("enter the size of the second array");
        n=Integer.parseInt(d.readLine());
        int b[]=new int[n];
        System.out.println("enter the elements into the second array");
        for(j=0;j<n;j++)
```

```
{
    b[j]=Integer.parseInt(d.readLine());
}
System.out.println("A pplying sorting on first array");
for(i=0;i<m;i++)
{
    for(j=i+1;j<m;j++)
    {
        if(a[i]>a[j])
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }
}
System.out.println("A pplying sorting on second array");
for(i=0;i<n;i++)
{
    for(j=i+1;j<n;j++)
    {
        if(b[i]>b[j])
        {
            temp=b[i];
            b[i]=b[j];
            b[j]=temp;
        }
    }
}
```

```
                b[j]=temp;
            }
        }
    }
    System.out.println("the first array elements are");
    for(i=0;i<m;i++)
    {
        System.out.println(a[i]);
    }
    System.out.println("the second array elements are");
    for(j=0;j<n;j++)
    {
        System.out.println(b[j]);
    }
    i=0;j=0;k=0;
    int c[]=new int[m+n];
    while((i<m)&&(j<n))
    {
        if(a[i]<b[j])
        {
            c[k]=a[i];
            k++;
            i++;
        }
        if(a[i]>b[j])
```

```
    {  
        c[k]=b[j];  
        k++;  
        j++;  
    }
```

LectureNotes.in

```
    if(a[i]==a[j])  
    {
```

```
        c[k]=a[i];
```

```
        k++;
```

```
        c[k]=b[j];
```

```
        k++;
```

```
        i++;
```

```
        j++;
```

```
    }
```

```
}
```

```
while(i<m)
```

```
{
```

```
    c[k]=a[i];
```

```
    k++;
```

```
    i++;
```

```
}
```

```
while(j<n)
```

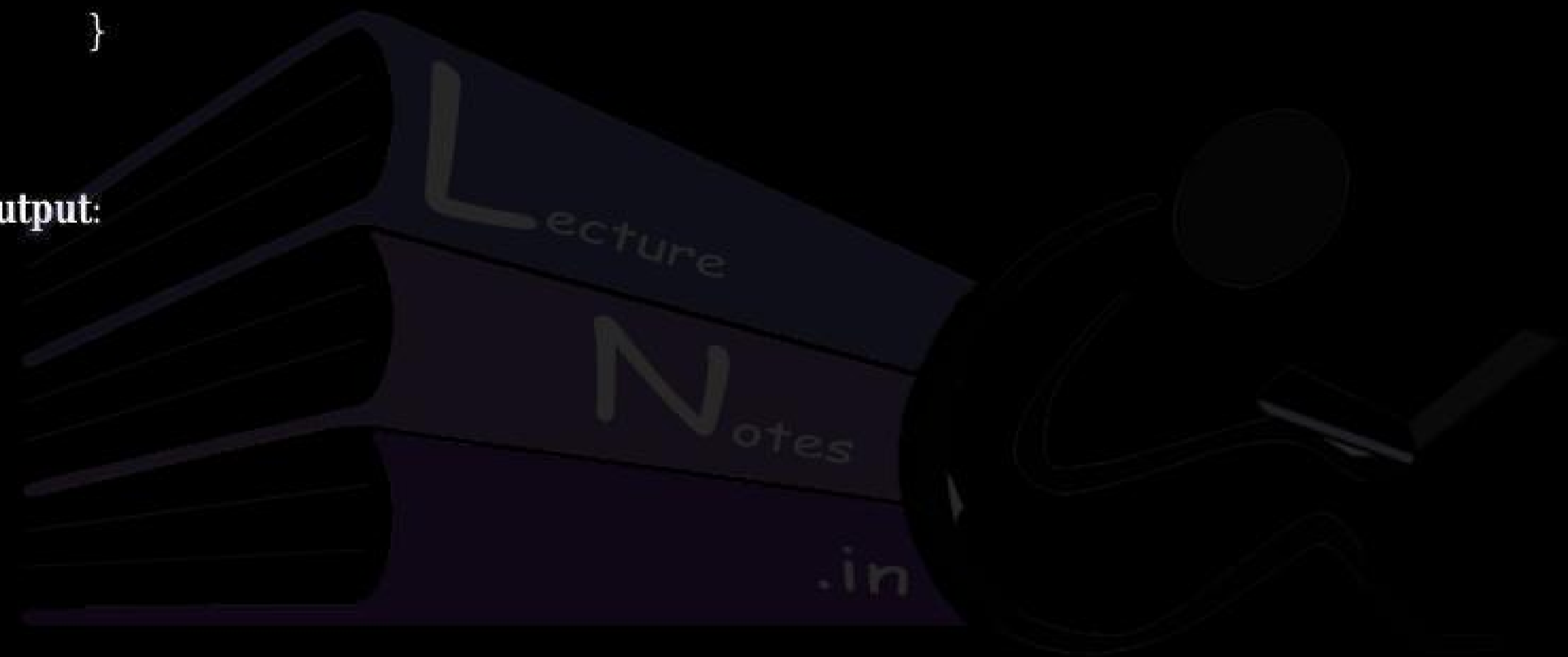
```
{
```

```
    c[k]=b[j];
```

```
    k++;
```

```
        j++;  
    }  
  
    System.out.println("After sorting the elements in the resultant array are.....");  
    for(k=0;k<m+n;k++)  
    {  
        System.out.println(c[k]);  
    }  
}  
}
```

Output:



LectureNotes.in


```

Command Prompt

E:\Naresh\DS Programs\Sortings>java MergeSort
Enter size of the first array:
3
Enter elements into the first array

2
8
6
Enter size of the second array:
4
Enter elements into the second array

12
5
3
9

Elements of first array are....

    2    8    6
Elements of second array are....

    12    5    3    9
Array elements after the sorting are....

    2    3    5    6    8    9    12
E:\Naresh\DS Programs\Sortings>

```

Quick Sort

Quick sort is a divide-and-conquer sorting algorithm in which division is dynamically carried out (as opposed to static division in Merge sort).

The time complexity for the quick sort is $O(n \log n)$. Where 'n' represents the number of elements.

The three steps of Quick sort are as follows:

Divide:

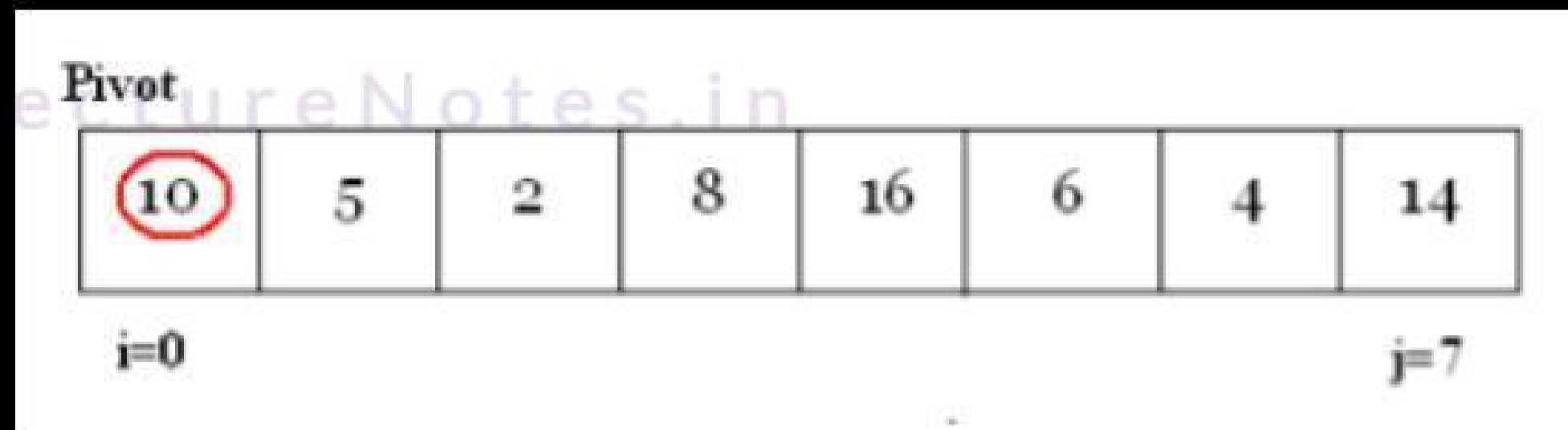
Rearrange the elements and split the array into two sub arrays by placing an element in between these two sub arrays so that each element in the left sub array is less than or equal the middle element and each element in the right sub array is greater than the middle element.

Conquer: Recursively sort the two sub arrays.

Combine: Since the sub arrays are sorted in place, no work is needed to combining them: the entire array S is now sorted.

Example for Quick Sort

Let us consider the following elements, $n=8$.



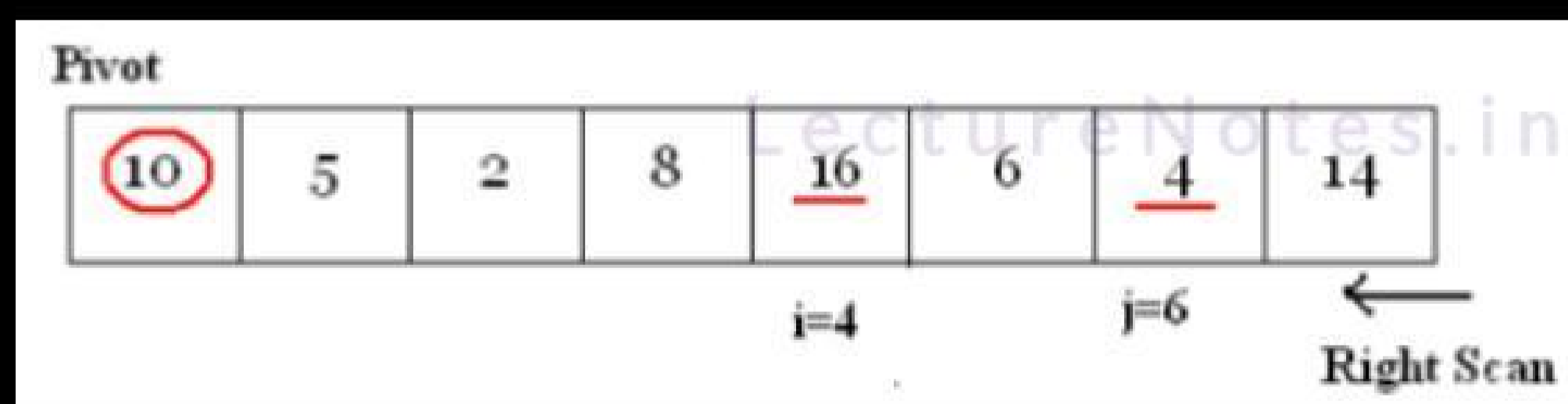
Start Left Scan,

Here our aim is to find the element greater than the key i.e., we increment the i value as long as we get an element greater than the key.

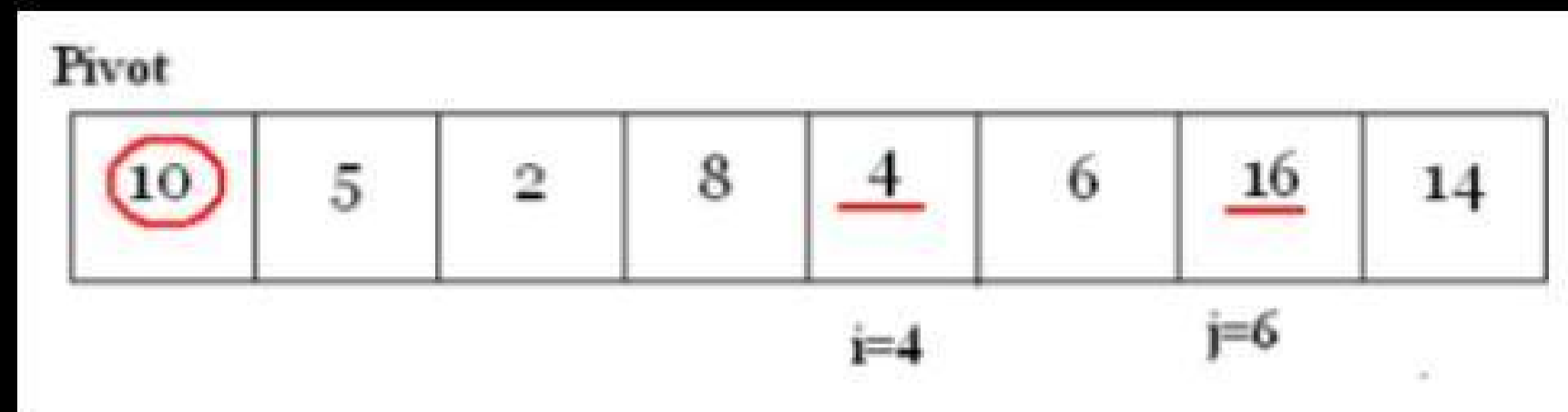


Now start the right scan,

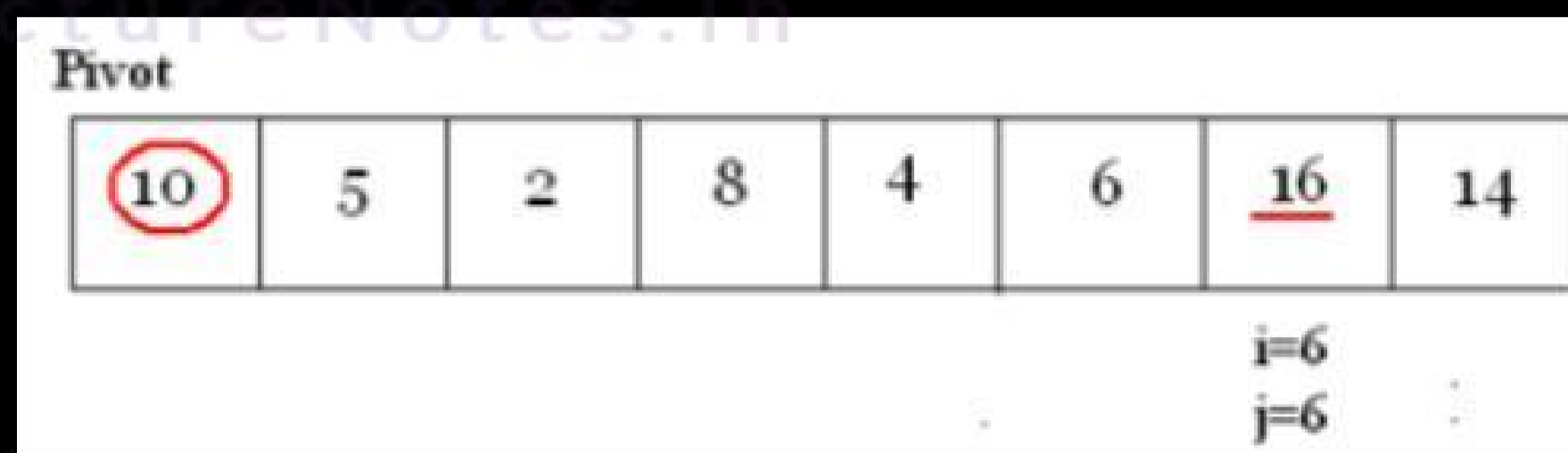
Here our aim is to find the element smaller than the key i.e., we decrement j value until we find an element less than the key.



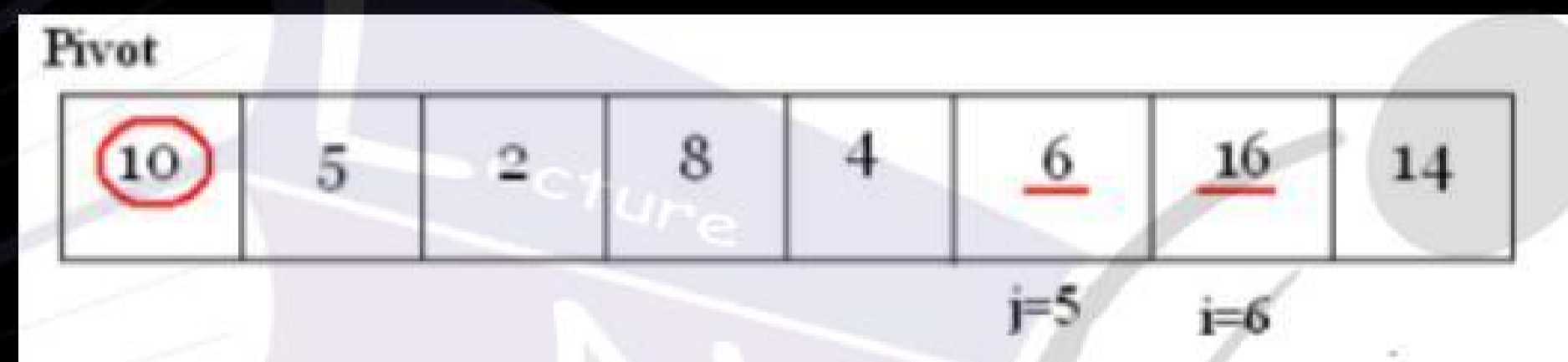
Here $i < j$, then swap $a[i]$, $a[j]$ and continue left scan and right scan.



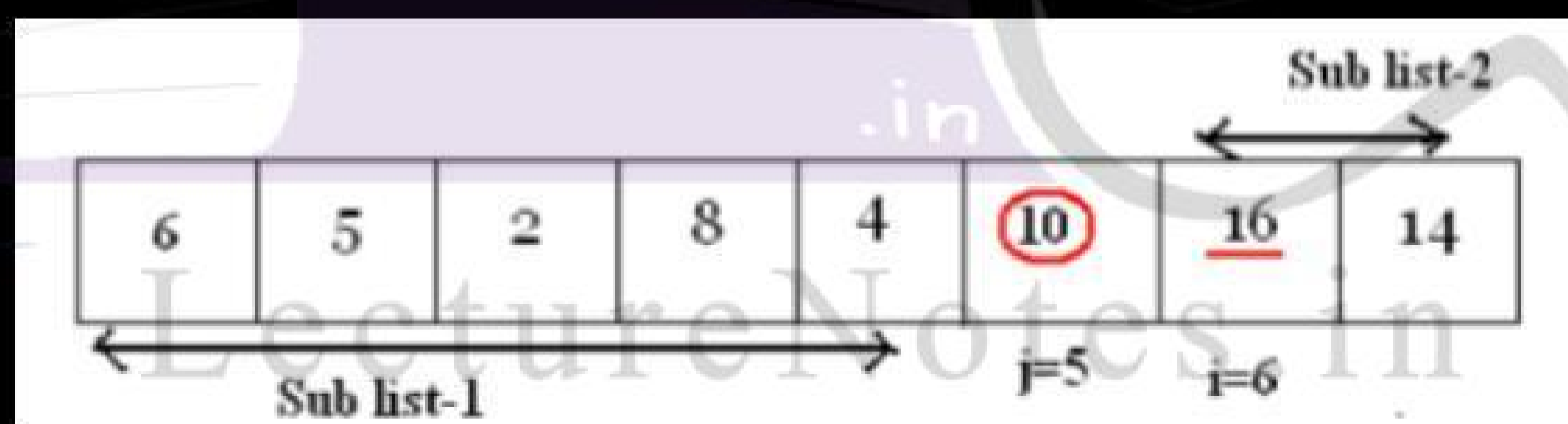
Now apply left scan,



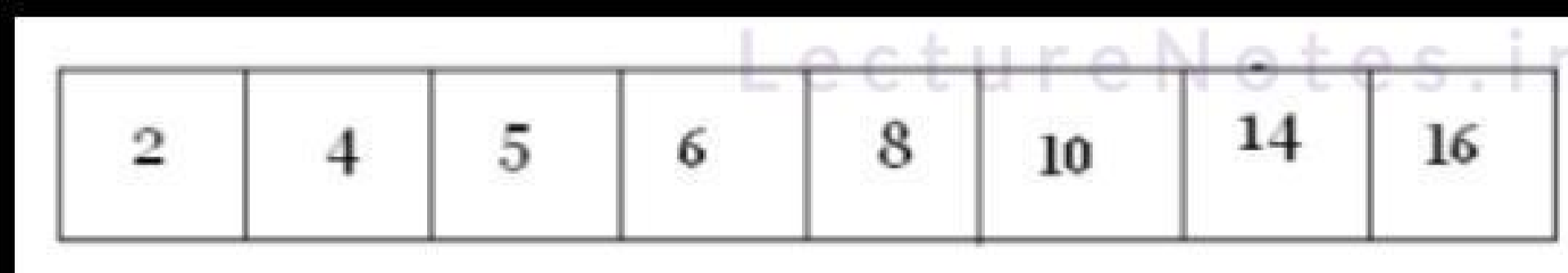
Now apply right scan,



Here $i > j$ therefore we swap $a[j]$ and key element.



Here we again apply the same process for sub list-1 and sub list-2 until each sub list contains only one element. Finally we get



The following program illustrates the implementation of Quick Sort in java.

Aim: To write a java program to implement the Quick Sort Algorithm.

Program://QuickSort.java

```
import java.io.*;
class QuickSort
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int n,i;
        System.out.println("enter the size of the array");
        n=Integer.parseInt(d.readLine());
        int a[]=new int[n];
        System.out.println("enter the array elements");
        for(i=0;i<n;i++)
        {
            a[i]=Integer.parseInt(d.readLine());
        }
        System.out.println("before sorting the array elements are");

        for(i=0;i<n;i++)
        {
            System.out.println(a[i]);
        }
        qsort(a,0,n-1);
        System.out.println("after sorting array elements are");
        for(i=0;i<n;i++)
```

```
        {  
            System.out.println(a[i]);  
        }  
    }  
  
    public static void qsort(int a[],int first,int last)  
    {  
        int i,j,pivot,temp;  
        if(first<last)  
        {  
            pivot=first;  
            i=first;  
            j=last;  
            while(i<j)  
            {  
                while((a[i]<=a[pivot])&&(i<last))  
                    i++;  
                while(a[j]>a[pivot])  
                    j--;  
                if(i<j)  
                {  
                    temp=a[i];  
                    a[i]=a[j];  
                    a[j]=temp;  
                }  
            }  
        }  
    }  
}
```

```

        temp=a[pivot];
        a[pivot]=a[j];
        a[j]=temp;
        qsort(a,first,j-1);
        qsort(a,j+1,last);
    }
}
}

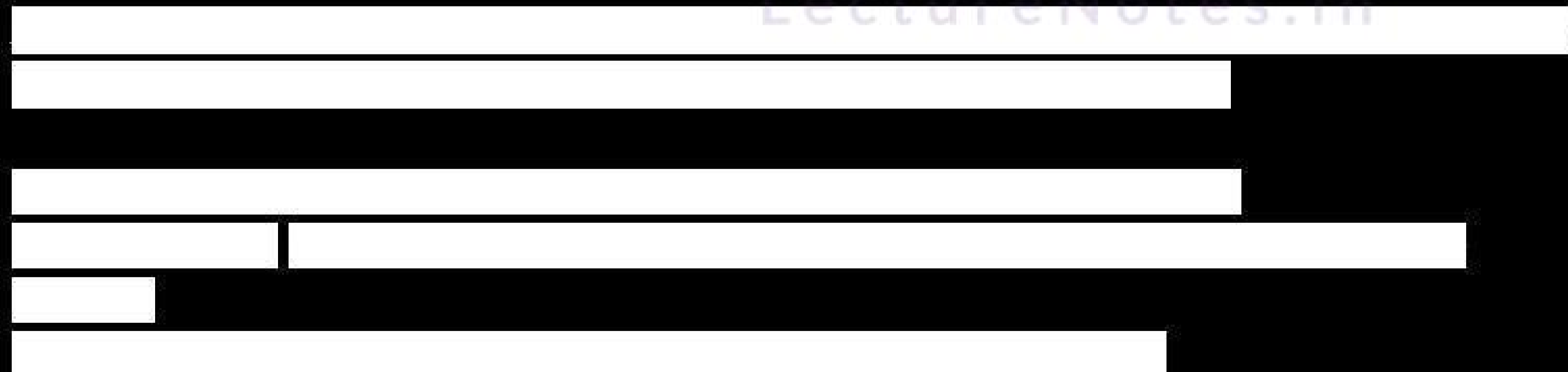
```

Output:

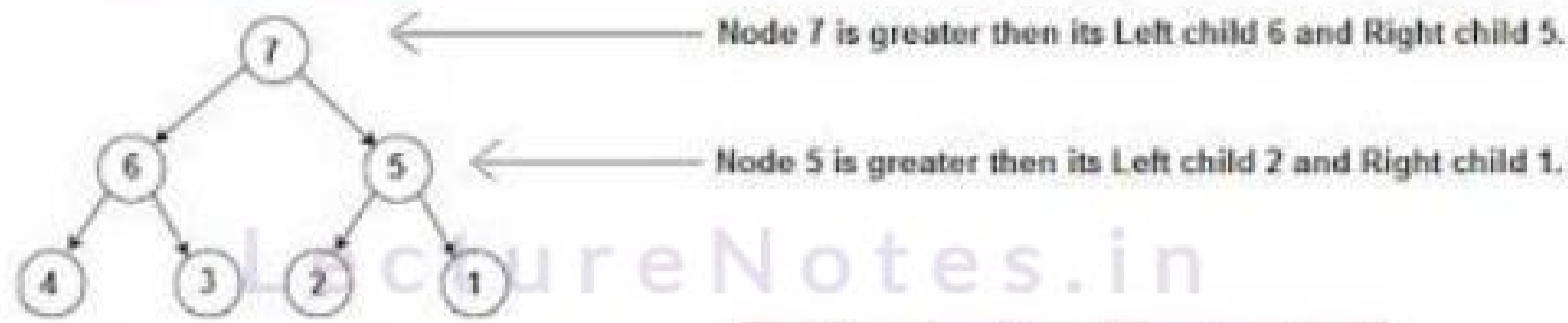
```

E:\Maresh\DS Programs\Sortings>javac QuickSort1.java
E:\Maresh\DS Programs\Sortings>java QuickSort1
Enter size of the array:
8
Enter elements into the array
10
5
2
8
16
6
4
14
Elements after sorting
2 4 5 6 8 10 14 16
E:\Maresh\DS Programs\Sortings>

```

Heap Sort:

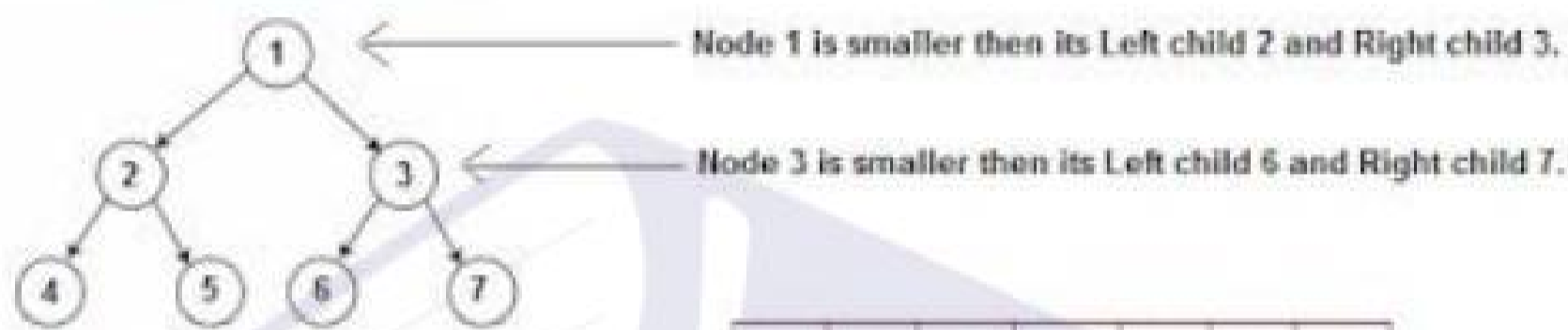
Max Heap Binary Tree



Array representation of above binary Tree:

7	6	5	4	3	2	1
---	---	---	---	---	---	---

Min Heap Binary Tree



Array representation of above binary Tree:

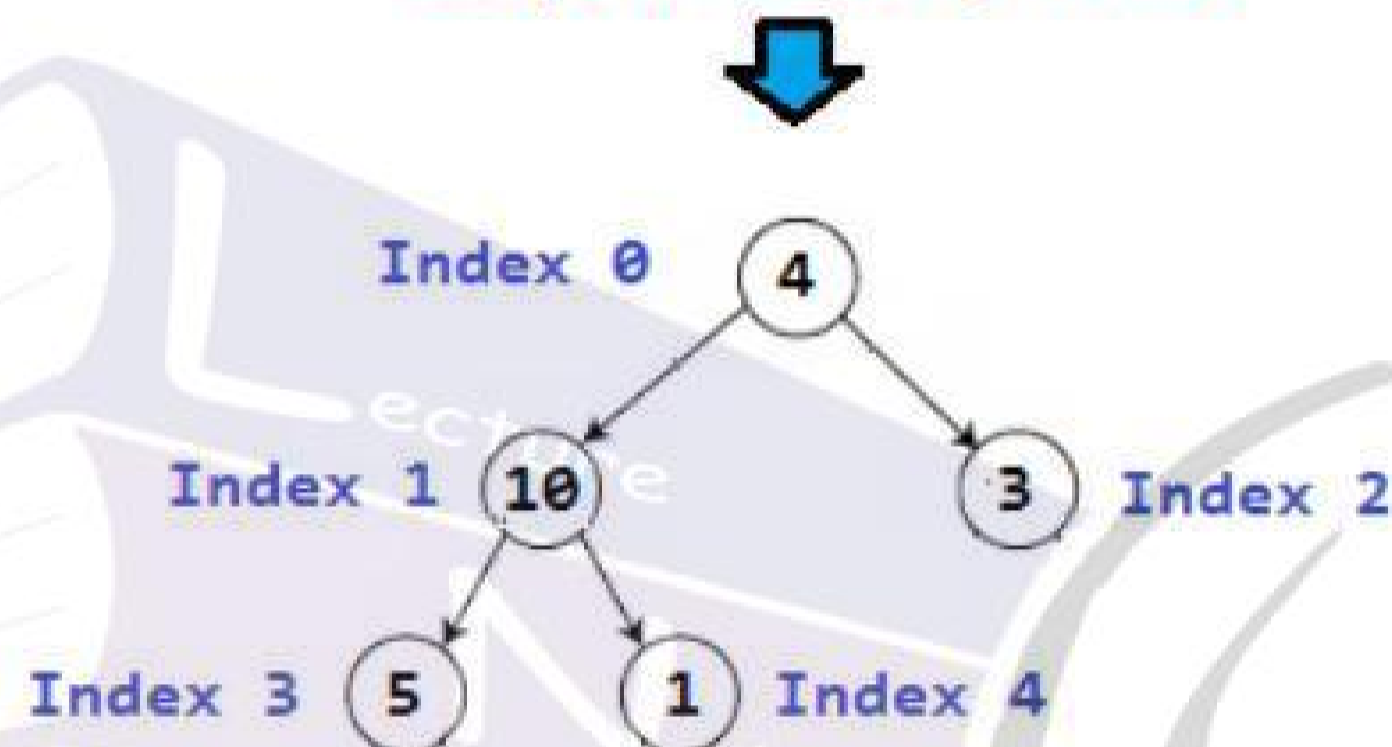
1	2	3	4	5	6	7
---	---	---	---	---	---	---

Heapify Process with Example

Heapify Process

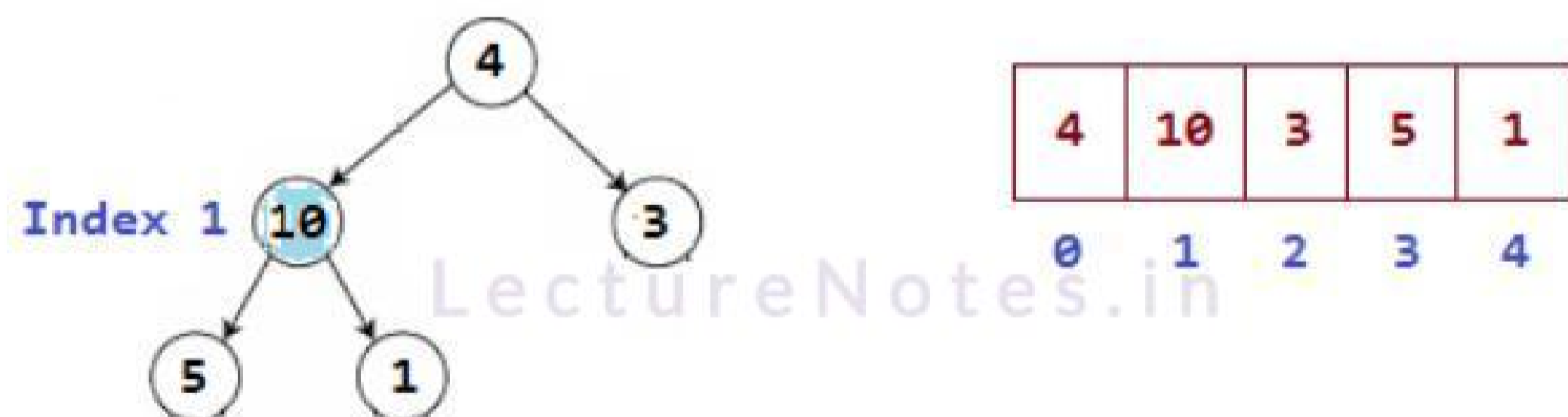
4	10	3	5	1
0	1	2	3	4

Represent array into Complete Binary Tree.

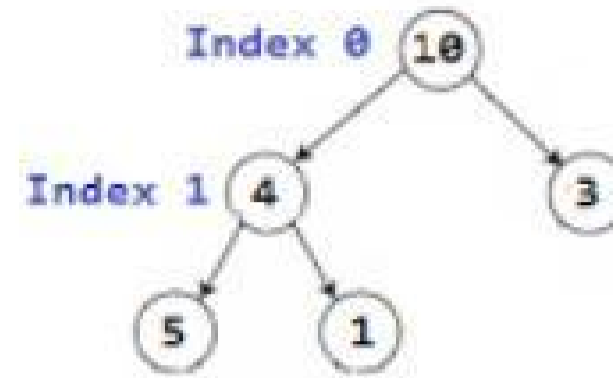
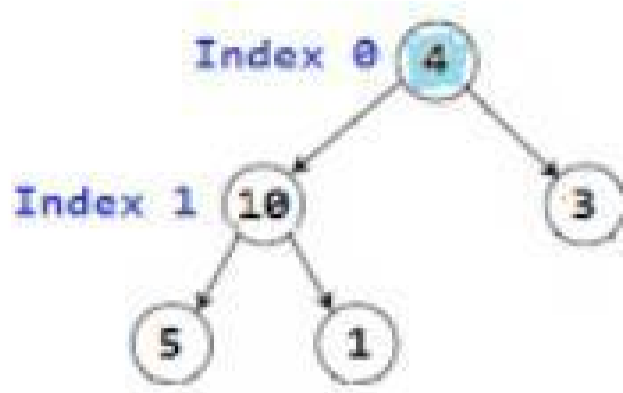


We need to start heapify process from Node at index
 $= (\text{array size}/2) - 1 = (5/2) - 1 = 2 - 1 = 1$

Check parent Node 10 is greater than Left child 5 and Right child 1.
YES. Node at index 1 satisfy max-heap property.



Check parent Node 4 is greater then Left child 10 and Right child 3.
 NO. Replace 4 with maximum from (4, 10, 3),

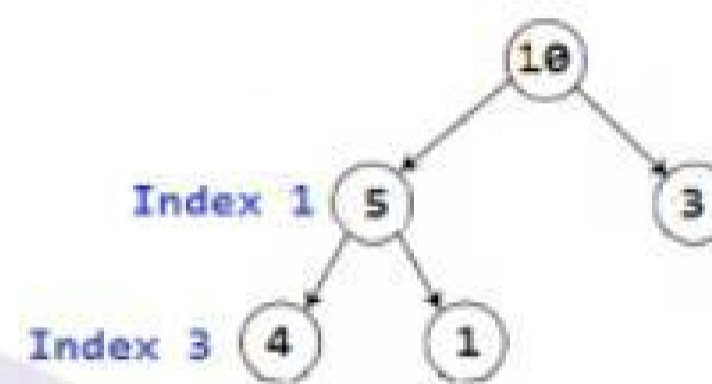
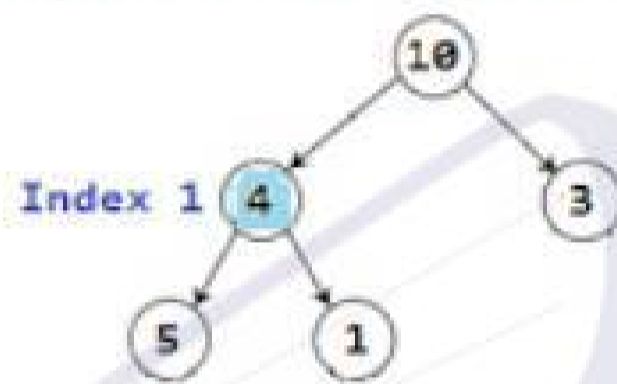


10	4	3	5	1
0	1	2	3	4

LectureNotes

Node at index 1 is replaced with Node at index 0.
 Now, Node 0 is satisfying max-heap property but Node at index 1 is disturbed and not satisfying max-heap property,
 So apply heapify process on Node at index 1

Check parent Node 4 is greater then Left child 5 and Right child 1.
 NO. Replace 4 with maximum from (4, 5, 1),



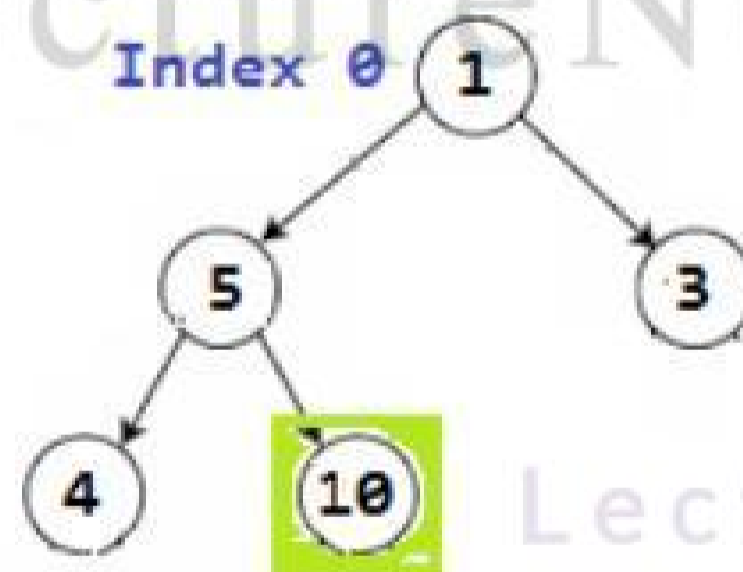
10	5	3	4	1
0	1	2	3	4

Node at index 1 is replaced with Node at index 3

All Nodes satisfy heapify property now. Root node contains item with maximum value 10.
 So place it at its proper place and remove it from further processing.
 Replace root node 10 with last node 1, So that item with maximum value is at last and in its proper place.

1	5	3	4	10
0	1	2	3	4

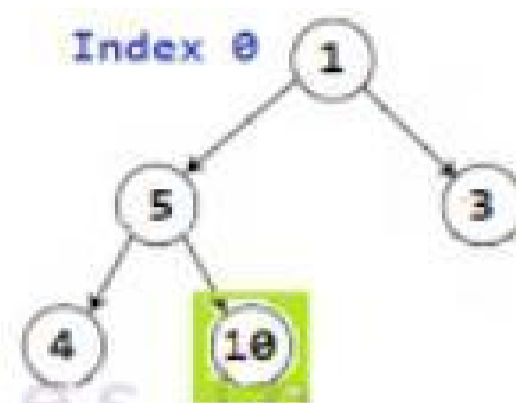
LectureNotes.in



LectureNotes.in

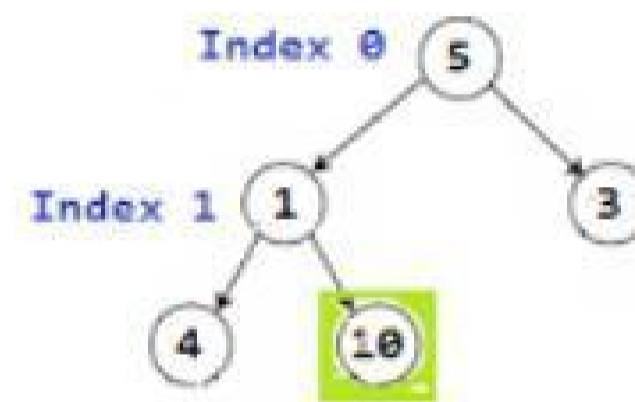
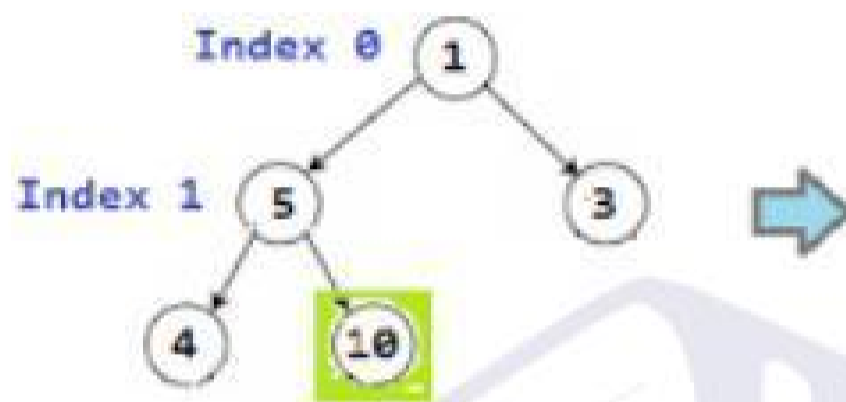
Node 10 is at proper place, So it will not be part of further heapify process.
 Node 1 when placed at the root, again disturbed heapify process at index 0.
 Repeat Heapify process at index 0.

1	5	3	4	10
0	1	2	3	4



LectureNotes.in

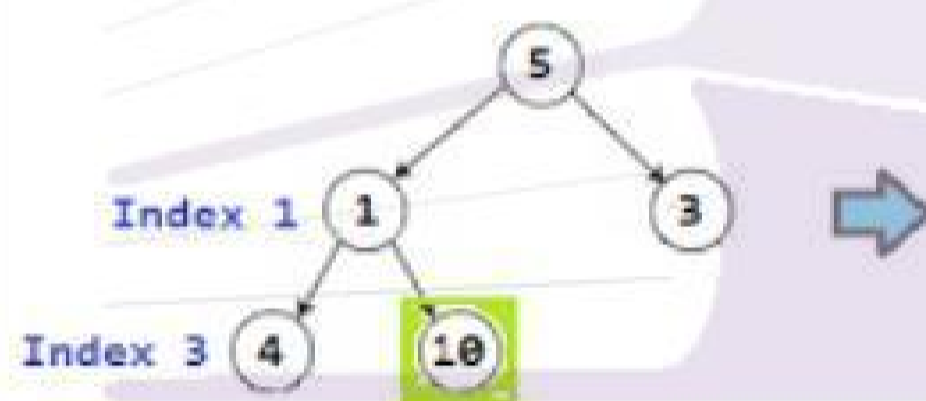
Check parent Node 1 is greater then Left child 5 and Right child 3.
 NO. Replace 1 with maximum from (1, 5, 3).



5	1	3	4	10
0	1	2	3	4

Node at index 1 is replaced with Node at index 0,
 Now, Node 0 is satisfying max-heap property but Node at
 index 1 is disturbed and not satisfying max-heap property.
 So apply heapify process on Node at index 1

Check parent Node 1 is greater then Left child 4.
 NO. Replace 4 with maximum from (1,4)



5	4	3	1	10
0	1	2	3	4

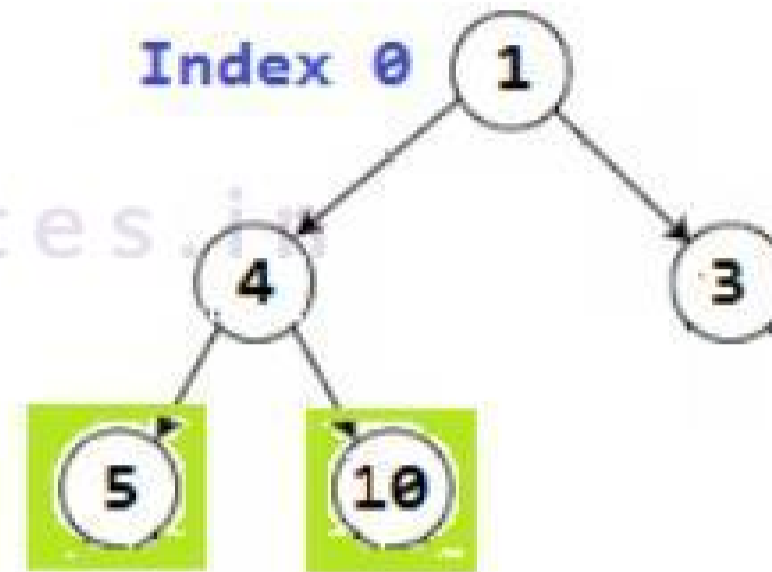
Node at index 1 is replaced with Node at index 3

All Nodes satisfy heapify property now. Root node contains item with maximum value 5.
 So place it at its proper place and remove it from further processing.
 Replace root node 5 with last node 1, So that item with maximum value is at last and in its proper place.

LectureNotes.in

LectureNotes.in

1	4	3	5	10
0	1	2	3	4

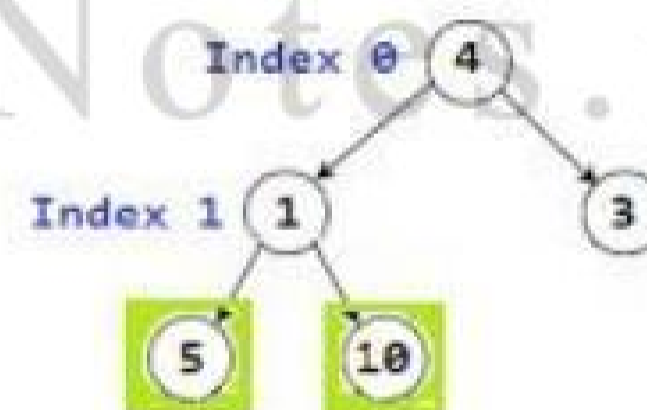
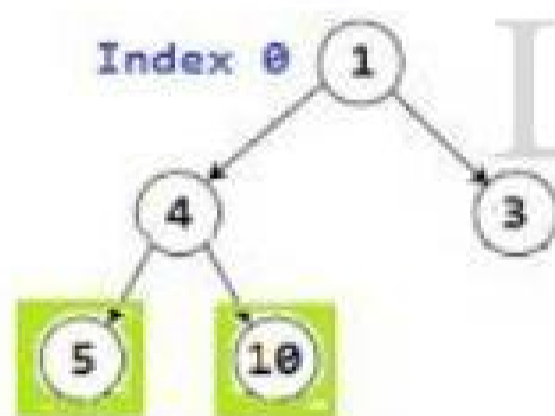


Node 5 is at proper place, So it will not be part of further heapify process.
 Node 1 when placed at the root, again disturbed heapify process at index 0.
 Repeat Heapify process at index 0.

1	4	3	5	10
0	1	2	3	4



Check parent Node 1 is greater then Left child 4 and Right child 3.
 NO. Replace 1 with maximum from (1, 4, 3),

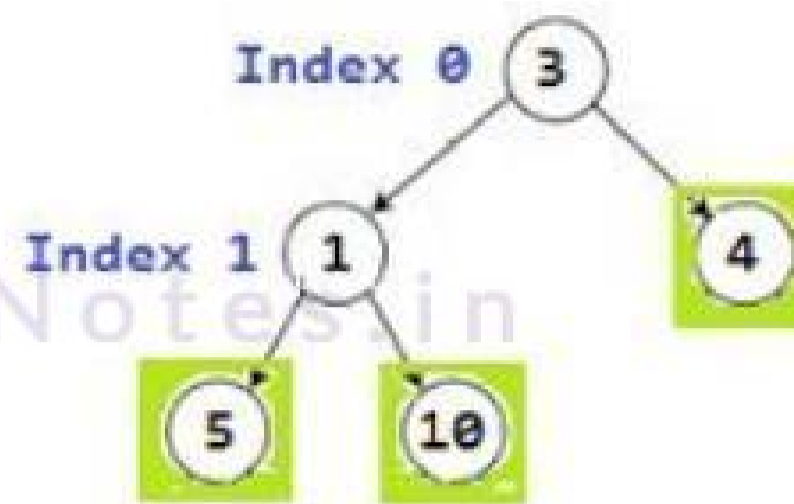


4	1	3	5	10
0	1	2	3	4

Node at index 1 is replaced with Node at index 0.

All Nodes satisfy heapify property now. Root node contains item with maximum value 4.
 So place it at its proper place and remove it from further processing.
 Replace root node 4 with last node 3, So that item with maximum value is at last and in its proper place.

3	1	4	5	10
0	1	2	3	4

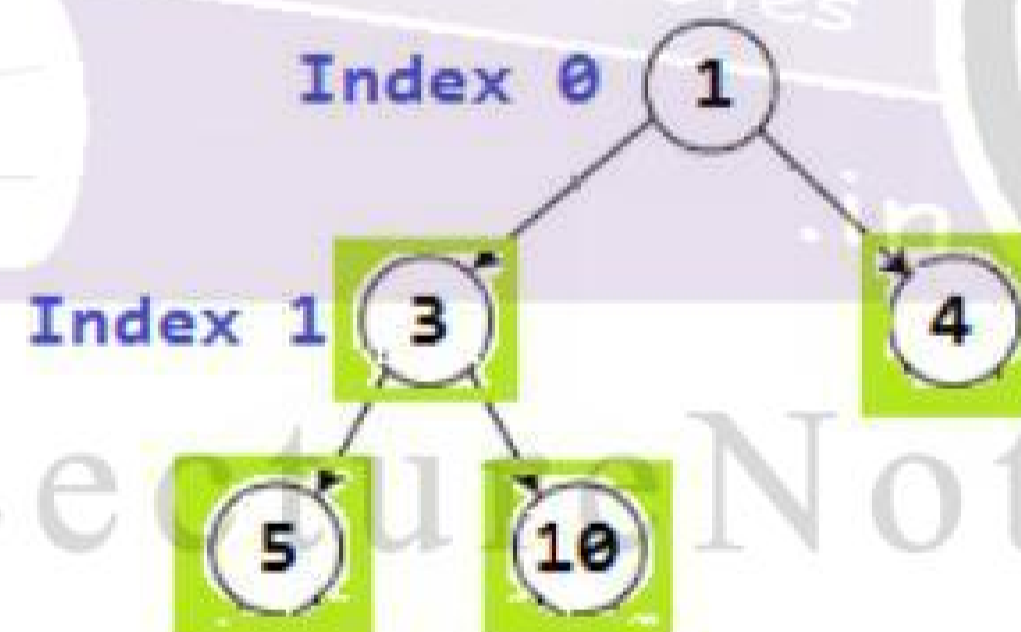


Check parent Node 3 is greater than Left child 1.
YES. Node at index 0 satisfy max-heap property.

All Nodes satisfy heapify property now. Root node contains item with maximum value 3.
 So place it at its proper place and remove it from further processing.

Replace root node 3 with last node 1, So that item with maximum value is at last and in its proper place.

1	3	4	5	10
0	1	2	3	4



Now, only 1 item is remaining which doesn't have any child and is leaf node, Node 1 which satisfies heap property and is already placed at its proper place.

Stop HEAPIFY process.
All elements are sorted now.

1	3	4	5	10
0	1	2	3	4

The following program illustrates the implementation of Heap Sort in java.

Aim: To write a java program to implement the Heap Sort Algorithm.

Program://HeapSort.java

```
import java.io.*;

class HeapSort
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int n,i,t;
        System.out.println("enter the size of an array");
        n=Integer.parseInt(d.readLine());
        int a[]=new int[n+1];
        System.out.println("enter the elements into the array");
        for(i=1;i<=n;i++)
        {
            a[i]=Integer.parseInt(d.readLine());
        }
        System.out.println("before sorting array elements are");
        for(i=1;i<=n;i++)
        {
            System.out.println(a[i]);
        }

        for(i=n;i>0;i--)
```

```
        {
            hsort(a,i);
            t=a[i];
            a[i]=a[1];
            a[1]=t;
        }
        System.out.println("after sorting the array elements are");
        for(i=1;i<=n;i++)
        {
            System.out.println(a[i]);
        }
    }
    public static void hsort(int a[],int n)
    {
        int j,temp;
        for(j=2;j<=n;j++)
        {
            if(a[j]>a[j/2])
            {
                temp=a[j];
                a[j]=a[j/2];
                a[j/2]=temp;
                if(j/2>1)
                    hsort(a,j/2);
            }
        }
    }
}
```

```
    }  
  }  
}
```

LectureNotes.in

OUTPUT:

```
enter the size of an array  
5  
enter the elements into the array  
34  
1  
67  
29  
41  
before sorting array elements are  
34  
1  
67  
29  
41  
after sorting the array elements are  
1  
29  
34  
41  
67
```

LectureNotes.in

What is Sparse Matrix

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a mXn matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Sparse matrix is a matrix which contains very few non-zero elements.

Sparse Matrix Representations

A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation
2. Linked Representation

Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.

For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

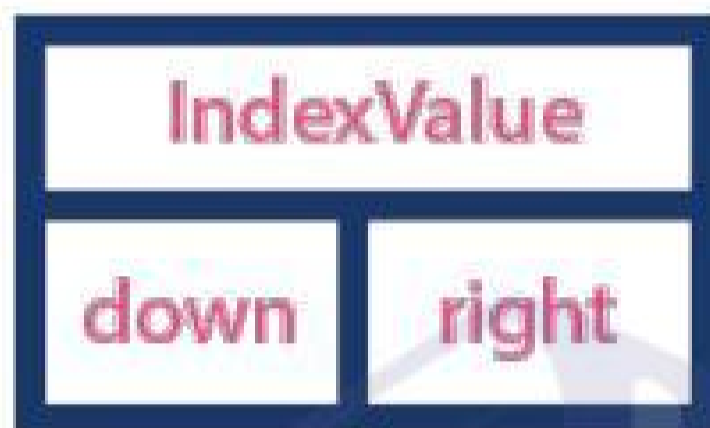
In above example matrix, there are only 6 non-zero elements (those are 9, 8, 4, 2, 5 & 2) and matrix size is 5 X 6. We represent this matrix as shown in the above image. Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values. Second row is filled with 0, 4, & 9 which indicates

the value in the matrix at 0th row, 4th column is 9. In the same way the remaining non-zero values also follows the similar pattern.

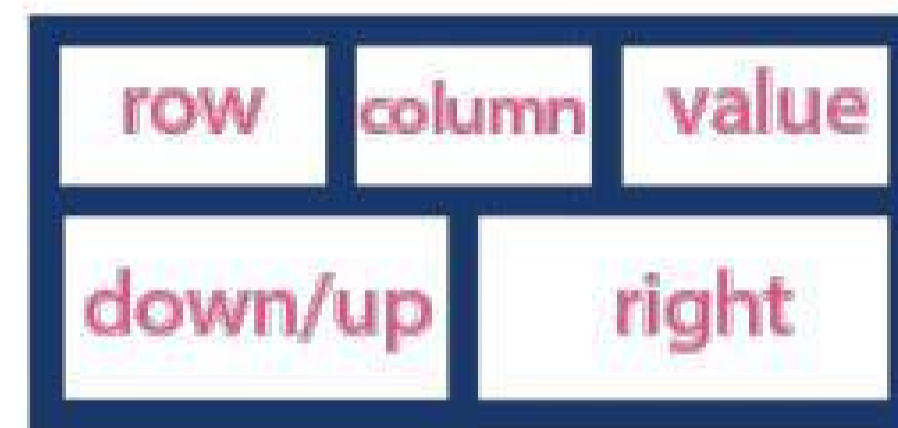
Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely **header node** and **element node**. Header node consists of three fields and element node consists of five fields as shown in the image...

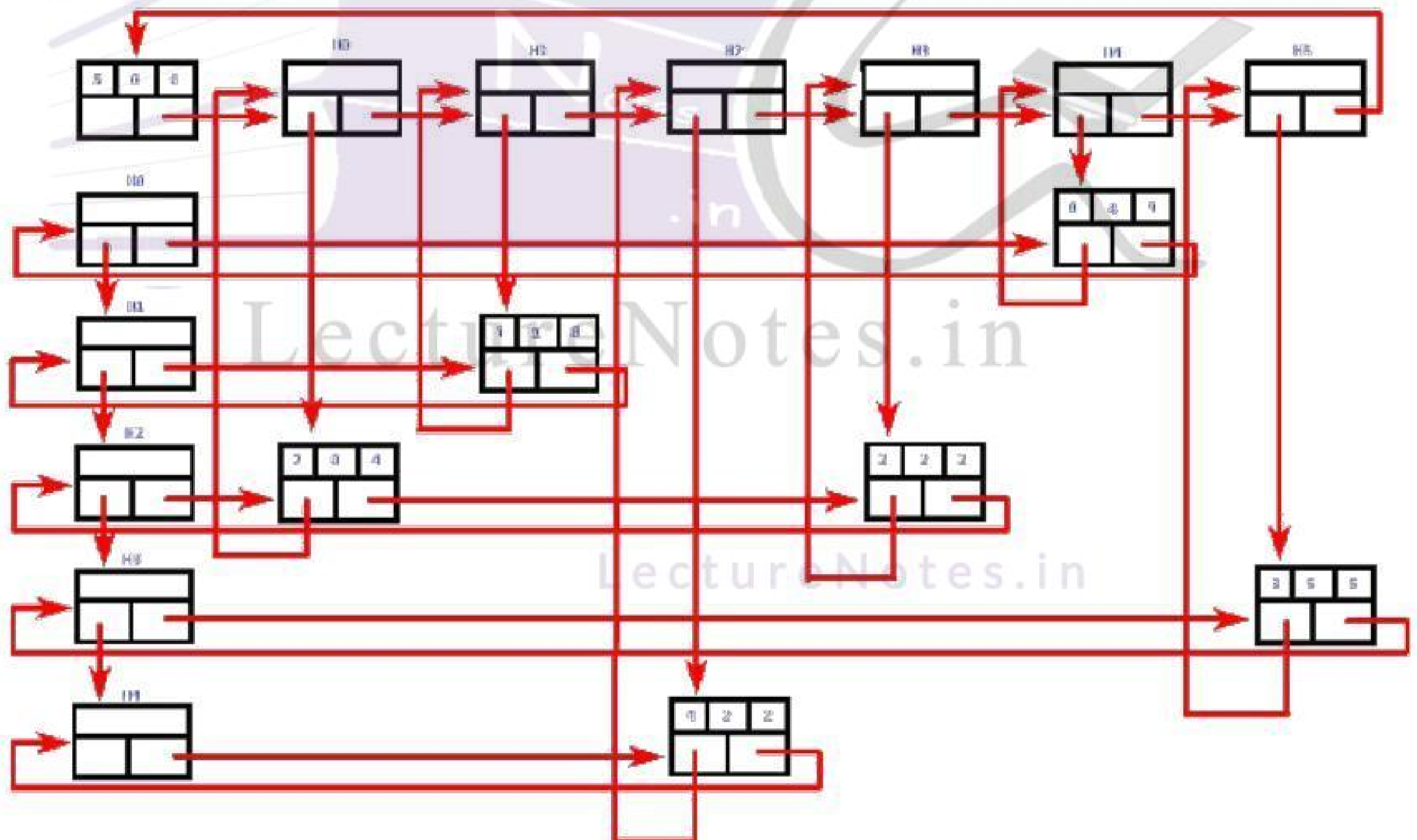
Header Node



Element Node



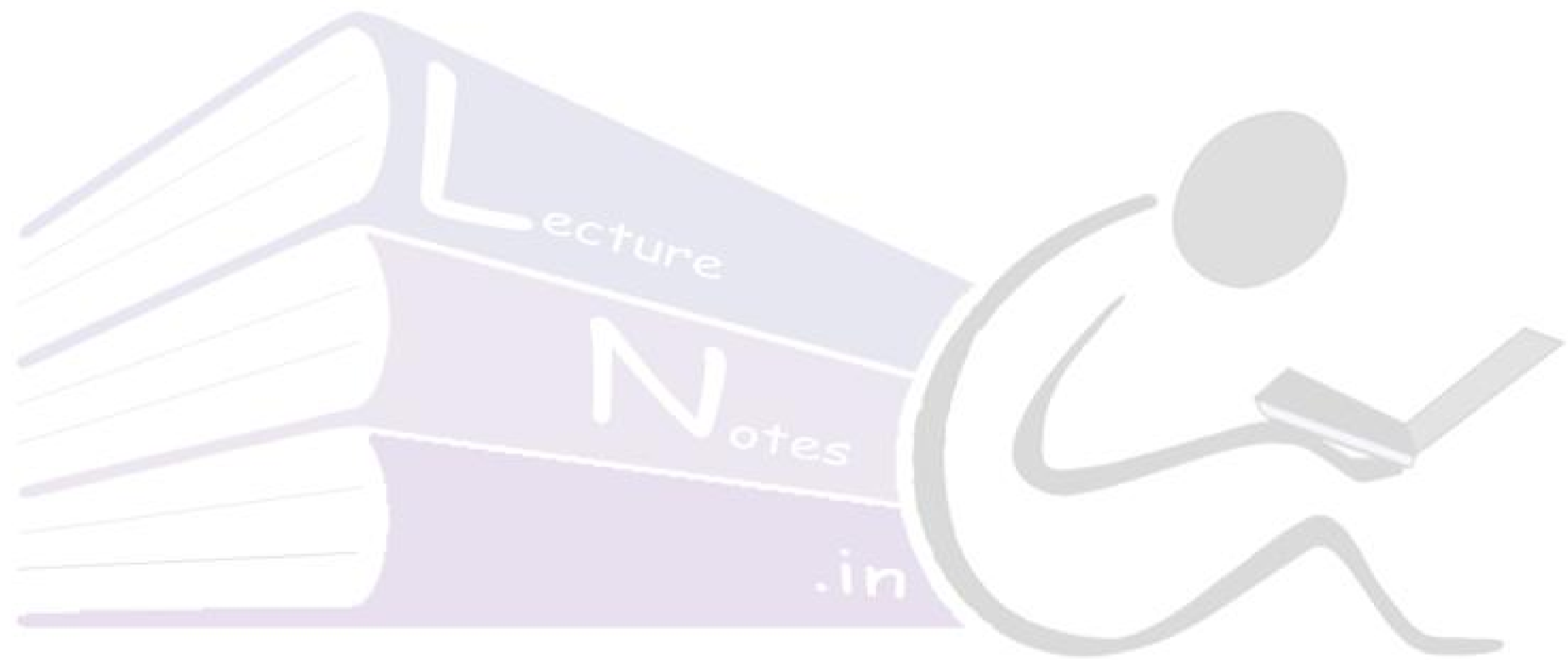
Consider the above same sparse matrix used in the Triplet representation. This sparse matrix can be represented using linked representation as shown in the below image...



In above representation, H0, H1,...,H5 indicates the header nodes which are used to represent indexes. Remaining nodes are used to represent non-zero elements in the matrix, except the very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

In this representation, in each row and column, the last node right field points to it's respective header node.

LectureNotes.in

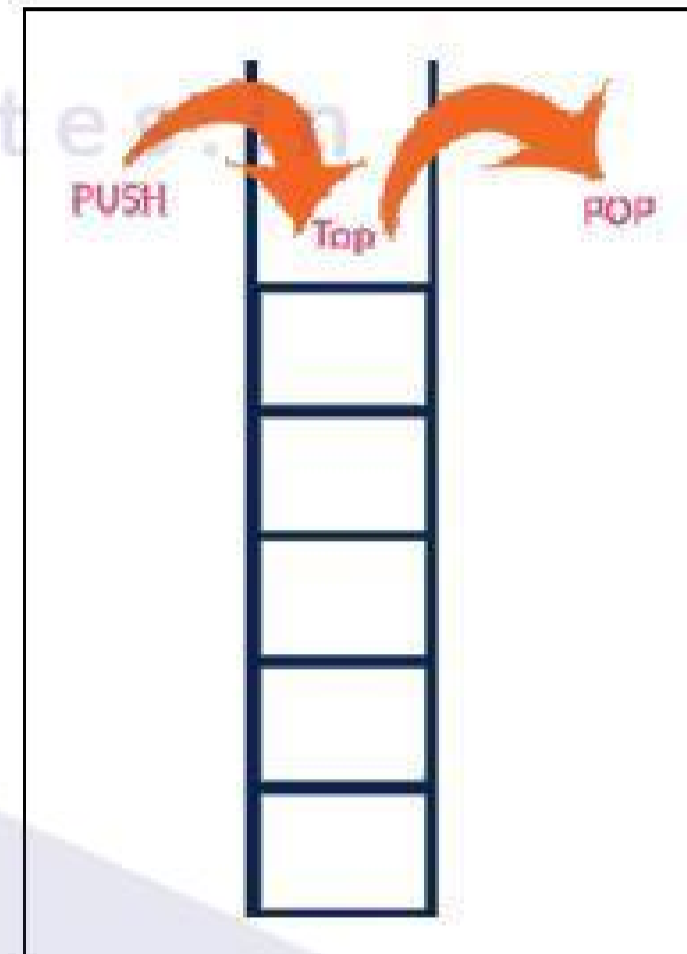


LectureNotes.in

LectureNotes.in

STACK AND ITS APPLICATIONS

- Stack is a linear data structure in which the insertion and deletion operations are performed at only one end known as **top** end.
- In stack, the insertion and deletion operations are performed based on **LIFO** (Last In First Out) principle.



- The process of inserting elements into the stack is called "**push**" and the process of deleting elements from the stack is called "**pop**".
- Initially the top is set to -1.

Finally, the stack data structure can be defined as follows...

Stack is a linear data structure in which the operations are performed based on LIFO principle.

Operation on Stack

The following fundamental operations on stack can be carried out through static implementation called array implementation. We will perform the following operations on the stack:

- **PUSH:** Inserting an element into the stack
- **POP:** Deleting an element from the stack
- **Display:** Displaying all the elements of stack

Implementation of Stack

Stack can be implemented in two ways:

1. Array Implementation of stack (or static implementation)
2. Linked list implementation of stack (or dynamic)

Array (static) implementation of a stack

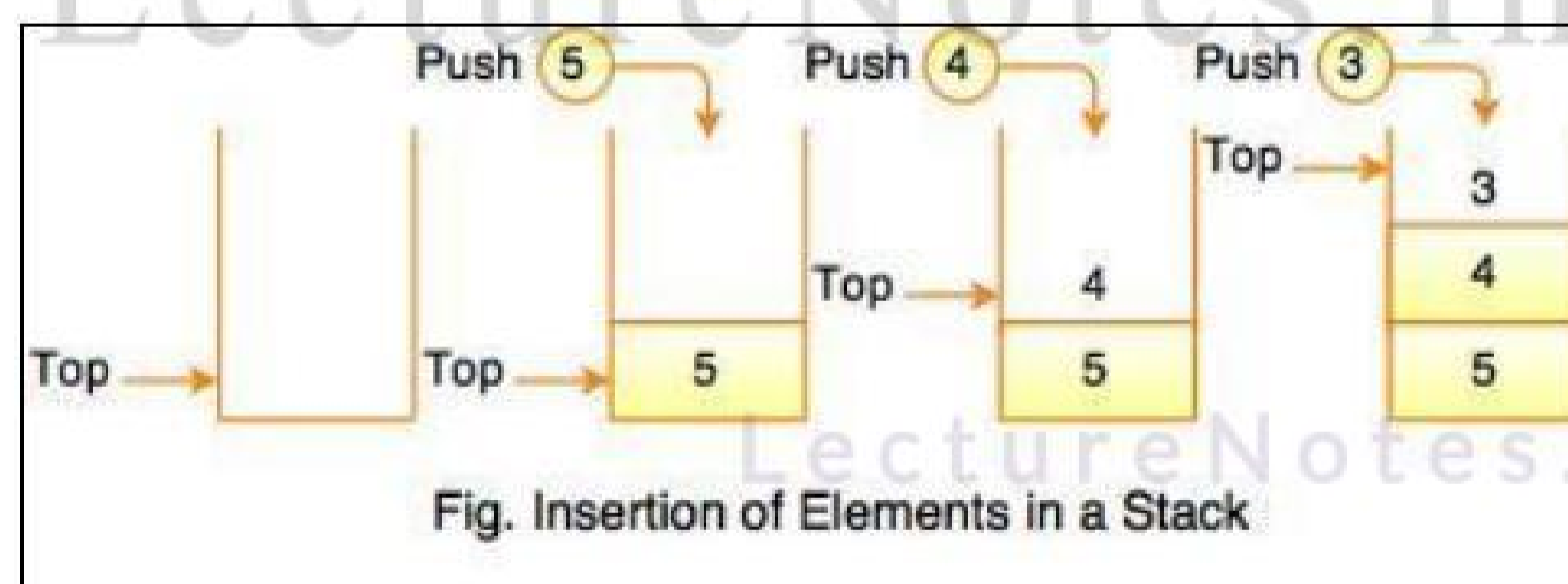
It is one of two ways to implement a stack that uses a one dimensional array to store the data. In this implementation top is an integer value (an index of an array) that indicates the top position of a stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of current top of the stack. By convention, in Java implementation the empty stack is indicated by setting the value of top to -1 (top = -1).

Algorithm for Push operation on stack

The insertion of element onto the stack is called as "push" i.e. when an item is added to the stack the operation is called "push".

To insert an element into the stack first we need to check the following condition.

```
if(top==size-1)
{
    Stack is overflow;
}
else
{
    top++;
    stk[top]=ele;
}
```



Stack Overflow

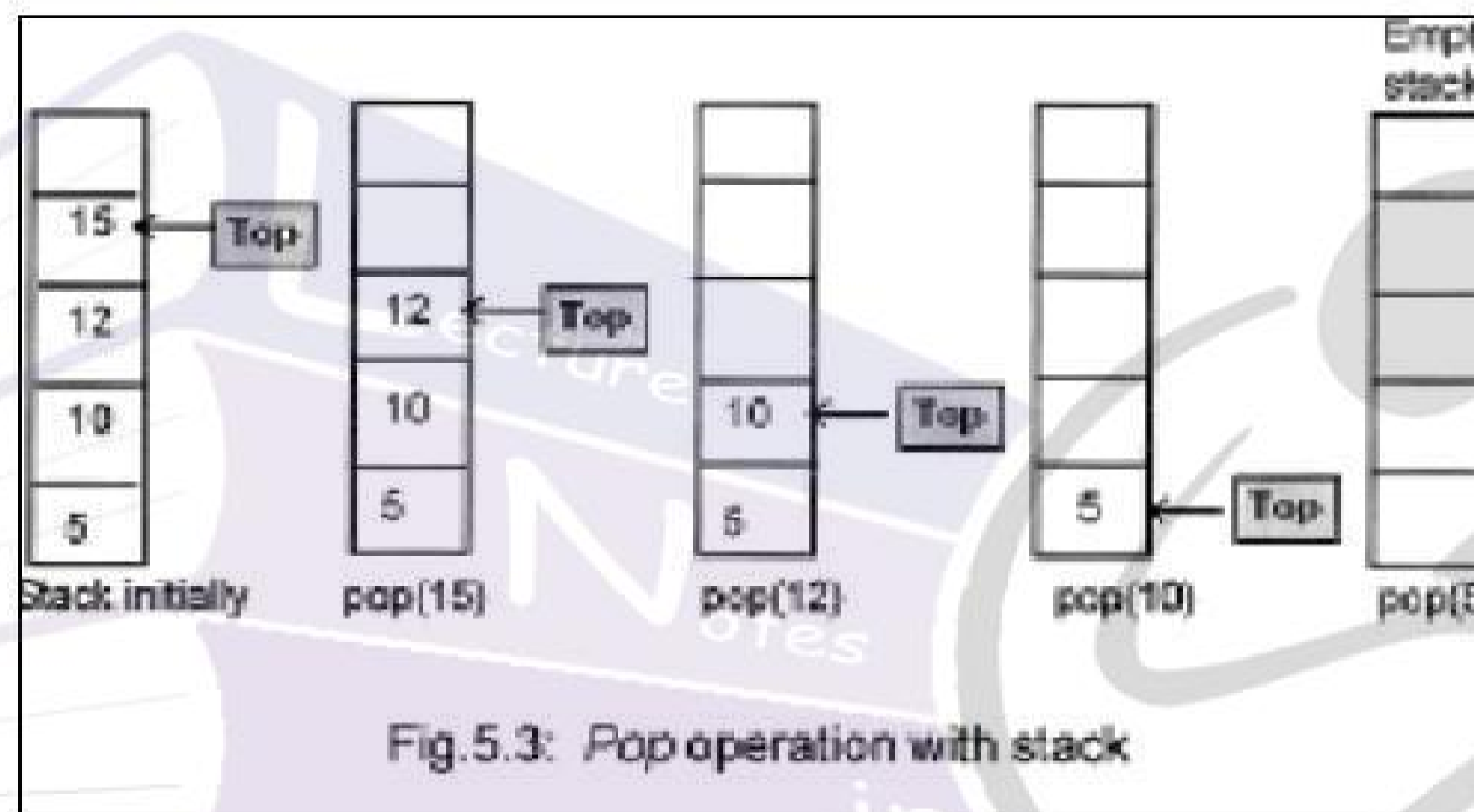
Any attempt to insert a new element in already full stack results into Stack Overflow.

Algorithm for POP operation on stack

The deletion operation is called "pop i.e., when an item is removed from the stack the operation is called "pop".

To POP an element from the stack first we need to check the following condition.

```
if(top==1)
{
    Stack is underflow;
}
else
{
    d=stk[top];
    top--;
}
```



Stack Underflow

Any attempt to delete an element from already empty stack results into Stack Underflow.

The following program illustrates the implementation of Stack data structure using arrays.

Aim: To write a java program to implement the operations of stack data structure using arrays.

Program: StackDemo.java

```
import java.io.*;
import java.util.*;
import java.lang.*;

class Stack
{
```

```
int s[],top,max;
Stack(int size)
{
    max=size;
    top=-1;
    s=new int[max];
}
void push(int ele)
{
    if(top==max-1)
    {
        System.out.println("Stack is Overflow");
    }
    else
    {
        top++;
        s[top]=ele;
        System.out.println(s[top]+" element inserted into the stack");
    }
}
void pop()
{
    int ele;
    if(top==1)
    {
        System.out.println("stack is underflow");
    }
}
```

```
    }  
    else  
    {  
        ele=s[top];  
        System.out.println(s[top]+" element deleted from the stack");  
        top--;  
    }  
}
```

```
void display()  
{  
    if(top==1)  
    {  
        System.out.println("Stack is empty");  
    }  
    else  
    {  
        System.out.println(" the stack elements are ");  
        int i;  
        for(i=top;i>=0;i--)  
        {  
            System.out.println(s[i]);  
        }  
    }  
}
```

```
}  
class StackDem  
{  
    public static void main(String args[])  
    {  
        Scanner d=new Scanner(System.in);  
        System.out.println("Enter size of the stack:");  
        int n=d.nextInt();  
        Stack s=new Stack(n);  
        int ch;  
        do  
        {  
            System.out.println("\n\n");  
            System.out.println("1.Push");  
            System.out.println("2.Pop");  
            System.out.println("3.Display");  
            System.out.println("4.Exit");  
  
            System.out.println("Enter your choice");  
            ch= d.nextInt();  
            switch(ch)  
            {  
                case 1:  
                    System.out.println("Enter an element");  
                    int x1= d.nextInt();  
                    s.push(x1);
```



```
break;
```

```
case 2:
```

```
s.pop();
```

```
break;
```

```
case 3:
```

```
s.display();
```

```
break;
```

```
case 4:
```

```
System.out.println("Exit from the stack");
```

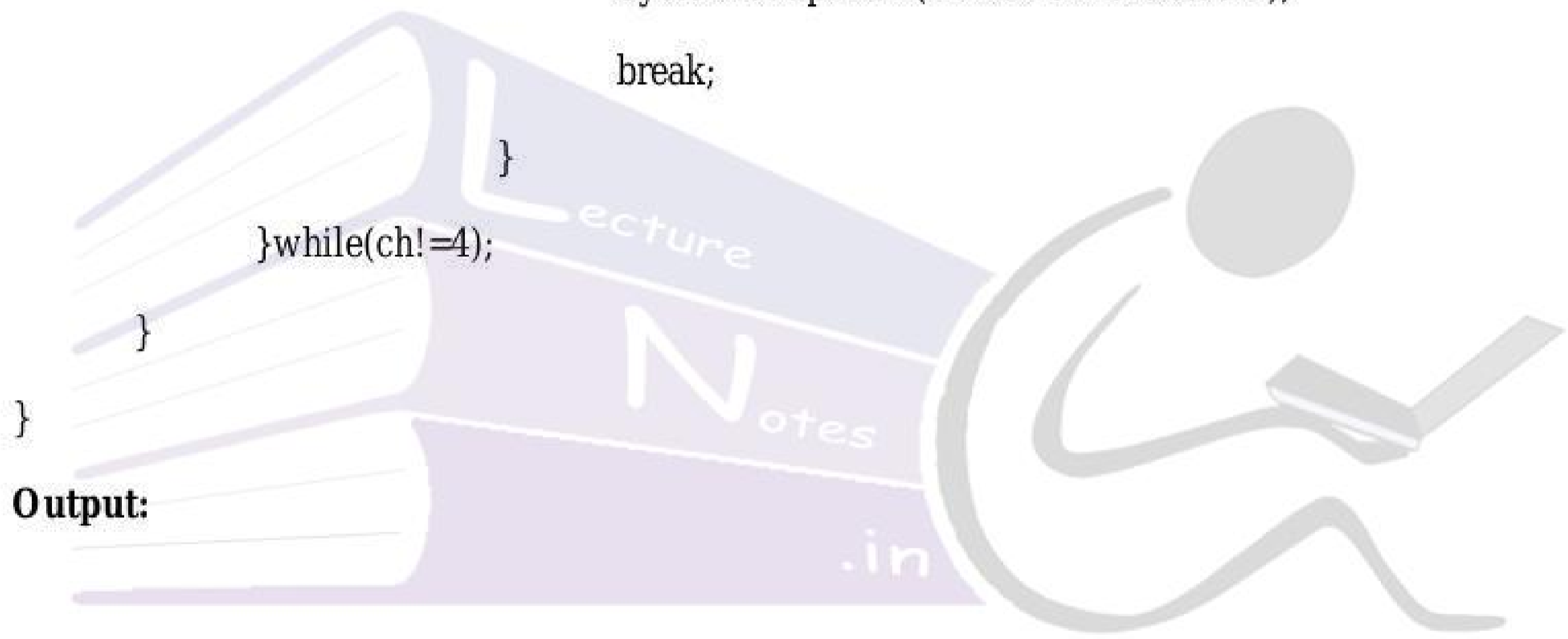
```
break;
```

```
}
```

```
}while(ch!=4);
```

```
}
```

Output:



LectureNotes.in

LectureNotes.in

C:\Program Files (x86)\EditPlus 2\launc

```
Enter size of the stack:  
5
```

```
1.Push  
2.Pop  
3.Display  
4.Exit  
Enter your choice
```

```
1  
Enter an element  
10
```

```
1.Push  
2.Pop  
3.Display  
4.Exit  
Enter your choice
```

```
1  
Enter an element  
20
```

```
1.Push  
2.Pop  
3.Display  
4.Exit  
Enter your choice
```

```
1  
Enter an element  
30
```

```
1.Push  
2.Pop  
3.Display  
4.Exit  
Enter your choice
```

```
1  
Enter an element  
40
```

Applications of Stack

Stack is used directly and indirectly in the following fields:

- To evaluate the expressions (postfix, prefix)
- To keep the page-visited history in a Web browser
- To perform the undo sequence in a text editor

- Used in recursion
- To pass the parameters between the functions.
- Can be used as an auxiliary data structure for implementing algorithms
- Can be used as a component of other data structures

Recursion

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result. In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and second, the problem statement must include a stopping condition.

Finding factorial of a number in Java using Recursion

The factorial of a number be found using recursion also. The base case can be taken as the factorial of the number 0 or 1, both of which are 1. The factorial of some number n is that number multiplied by the factorial of (n-1). Mathematically,

$$\begin{aligned} \text{factorial} (0) &= 1 \\ \text{factorial} (n) &= n * \text{factorial} (n - 1) \end{aligned}$$

Given below is a program which calculates the factorial of a given number using recursion.

Aim: To write a java program to find the factorial of a given number using recursion.

Program: Factorial.java

```
import java.util.Scanner;
class Factorial
{
    public static void main(String[] args)
    {
        int n, fact;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter any integer:");
        n = s.nextInt();
        if(n<0)
        {
            System.out.println("Factorial not posible");
        }
        else
        {
            fact = findFact(n);
            System.out.println("Factorial of "+n+" :"+fact);
        }
    }
    static int findFact(int x)
    {
```

```

    if(x > 1)
    {
        return(x * findFact(x - 1));
    }
    return 1;
}
}

```

Output:



Expressions

An expression can be defined as follows...

An expression is a collection of operators and operands that represents a specific value.

In above definition, **operator** is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,

Operands are the values on which the operators can perform the task. Here operand can be a direct value or variable or address of memory location.

Expression Types

Based on the operator position, expressions are divided into THREE types. They are as follows...

1. **Infix Expression**
2. **Postfix Expression**
3. **Prefix Expression**

Infix Expression

In infix expression, operator is used in between operands.

The general structure of an Infix expression is as follows...

Operand1 Operator Operand2

Example



LectureNotes.in

Postfix Expression

In postfix expression, operator is used after operands. We can say that "**Operator follows the Operands**".

The general structure of Postfix expression is as follows...

Operand1 Operand2 Operator

Example



Prefix Expression

In prefix expression, operator is used before operands. We can say that "**Operands follows the Operator**".

The general structure of Prefix expression is as follows...

Operator Operand1 Operand2

Example



Any expression can be represented using the above three different types of expressions. And we can convert an expression from one form to another form like **Infix to Postfix**, **Infix to Prefix**, **Prefix to Postfix** and vice versa.

Expression Conversion

Any expression can be represented using three types of expressions (Infix, Postfix and Prefix). We can also convert one type of expression to another type of expression like Infix to Postfix, Infix to Prefix, Postfix to Prefix and vice versa.

Infix to Postfix Conversion using Stack Data Structure

To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps...

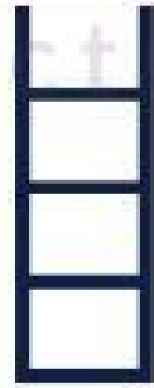

1. Read all the symbols one by one from left to right in the given Infix Expression.
2. If the reading symbol is operand, then directly print it to the result (Output).
3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.
5. If the reading symbol is operator (+, -, *, / etc.), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.




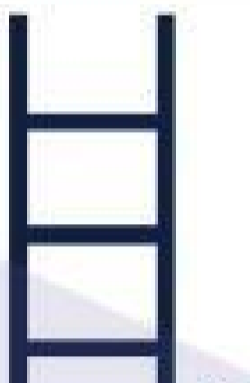



Example


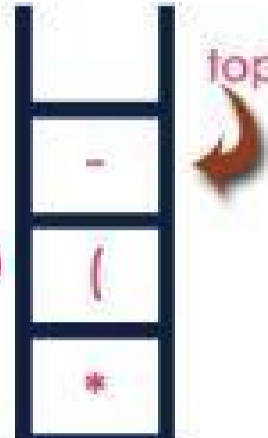

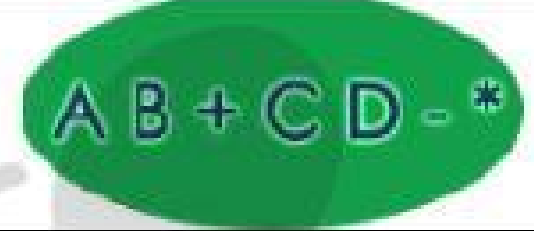
Consider the following Infix Expression...

$(A + B) * (C - D)$

The given infix expression can be converted into postfix expression using Stack data Structure as follows...

Reading character	Stack	Postfix expression
Initially	Stack is EMPTY 	Empty
(Push '(' 	Empty

A	<p>No operation Since 'A' is OPERAND</p> 	A
+ LectureNotes.in	<p>'+' has low priority than '(' so, PUSH '+'</p> 	A
B	<p>No operation Since 'B' is OPERAND</p> 	AB
)	<p>POP all elements till we reach '('</p> <p>POP '+' POP '('</p> 	AB+
* LectureNotes.in	<p>Stack is EMPTY & '*' is Operator PUSH '*'</p> 	AB+
(<p>PUSH '('</p> 	AB+
C	<p>No operation Since 'C' is OPERAND</p> 	AB+C

-	<p>'-' has low priority than '(' so, PUSH '-'</p> 	AB+C
D	<p>No operation Since 'D' is OPERAND</p> 	AB+CD
)	<p>POP all elements till we reach '('</p> <p>POP '-' POP '('</p> 	AB+CD-
End of expression	<p>POP all elements till Stack becomes Empty</p>	

The final Postfix Expression is as follows...

AB+CD-*

The following is the java program to convert an Infix expression into a Postfix expression using stack data structure.....

Aim: To write a java program to convert an infix expression into postfix expression using stack data structure.

```

/*InToPost.java*/
import java.io.*;
class stack
{
    char s[];
    int top;
    stack(int len)
    {
        S[]=new char[len];
        top=-1;
    }
}

```



```

void push(char ch)
{
    top++;
    s[top]=ch;
}
char pop()
{
    char ch;
    ch=s[top];
    top--;
    return ch;
}
int pre(char ch)
{
    switch(ch)
    {
        case '=':return 0;
        case '-':return 1;
        case '+':return 1;
        case '*':return 2;
        case '/':return 2;
    }
    return 0;
}
boolean operator(char ch)
{
    if(ch=='/' || ch=='*' || ch=='+' || ch=='-' || ch=='=')
        return true;
    else
        return false;
}

```

```

boolean isOperand(char ch)
{
    if(ch>='a' && ch<='z' || ch>='A' && ch<='Z' || ch>='0' && ch<='9')
        return true;
    else
        return false;
}

```

```

void postfix(String str)
{
    char output[]=new char[str.length()+1];
    char ch;
    int p=0,i;
    for(i=0;i<str.length();i++)
    {
        ch=str.charAt(i);
        if(ch=='(')
        {

```

```

        push(ch);
    }
    else if(isOperand(ch))
    {
        output[p++]=ch;
    }
    else if(operator(ch))
    {
        if(top==-1 || (pre(ch)>=pre(stack1[top])) || stack1[top]!='(')
        {
            push(ch);
        }
        else if(pre(ch)<pre(stack1[top]))
        {
            output[p++]=pop();
            push(ch);
        }
    }
    else if(ch=='(')
    {
        while((ch=pop())!='(')
        {
            output[p++]=ch;
        }
    }
}
while(top!=-1)
{
    char c=pop();
    if(c!='(' && c!=')')
        output[p++]=c;
}
for(int j=0;j<str.length();j++)
{
    System.out.print(output[j]);
}
}
}
class InToPost
{
    public static void main(String[] args)throws Exception
    {
        DataInputStream d=new DataInputStream(System.in);
        String s;
        System.out.println("Enter input string");
        s=d.readLine();
        int len=s.length();
        stack b=new stack(len);

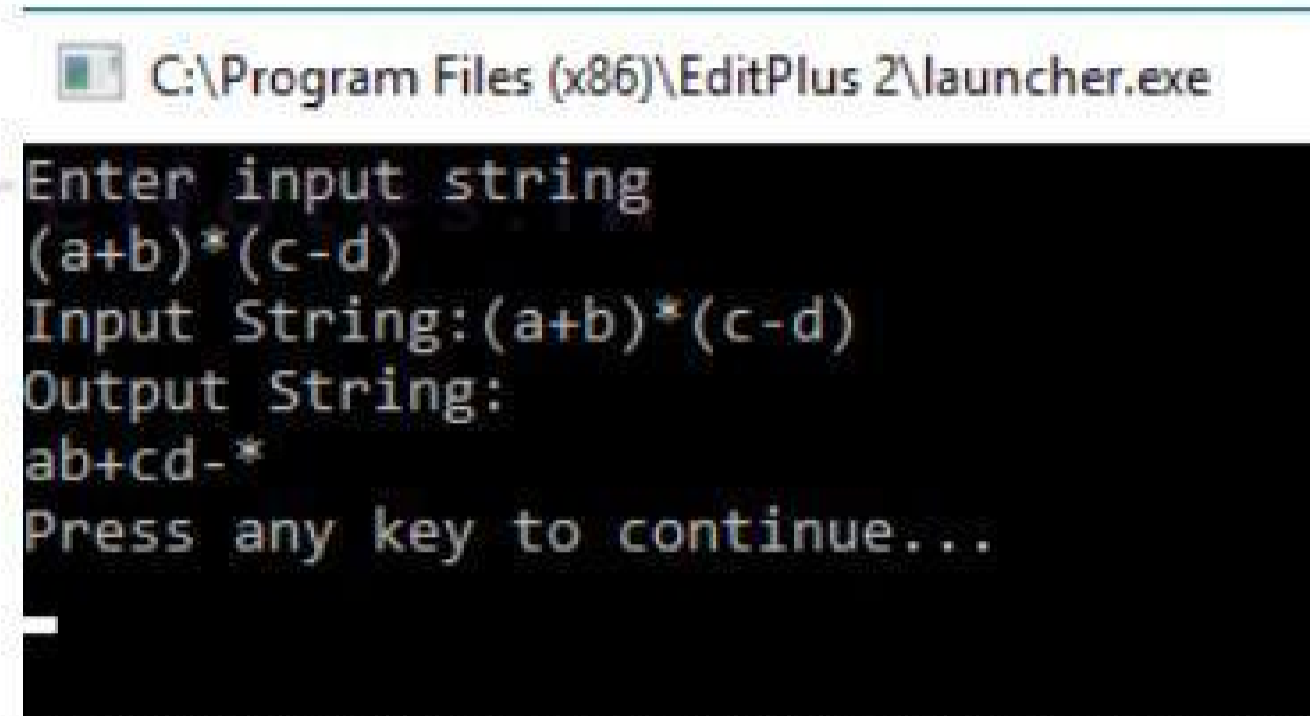
```

```

        System.out.println("InString:"+s);
        System.out.println("Output String:");
        b.postfix(s);
    }
}

```

Output:



Postfix Expression Evaluation

A postfix expression is a collection of operators and operands in which the operator is placed after the operands. That means, in a postfix expression the operator follows the operands.

Postfix Expression has following general structure...

Operand1 Operand2 Operator

Example



Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+, -, *, / etc.), then perform TWO pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.

Example

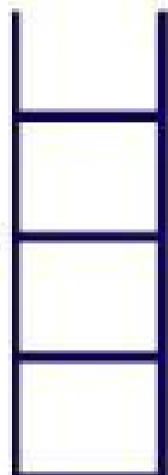
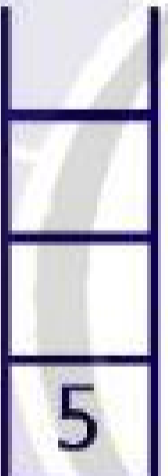
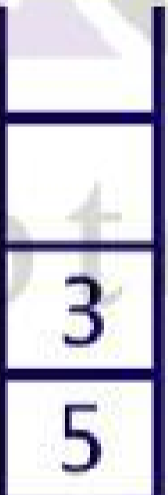
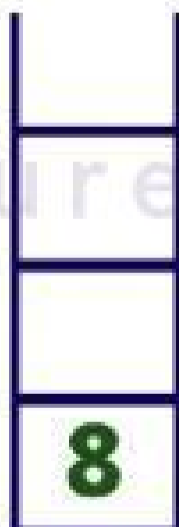
Consider the following Expression...

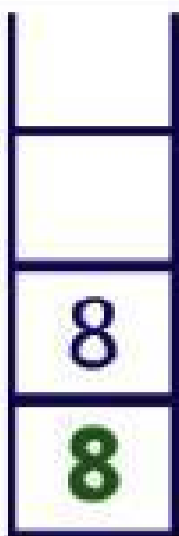
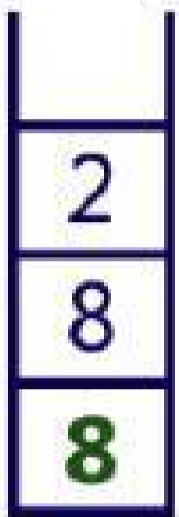

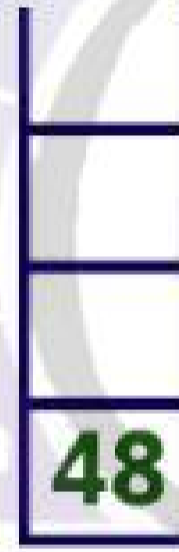
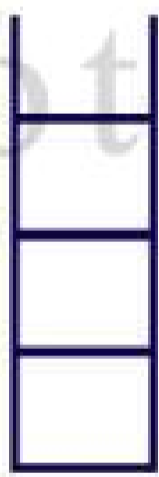
Infix Expression **$(5 + 3) * (8 - 2)$**

Postfix Expression **$5 3 + 8 2 - *$**

LectureNotes.in

The above postfix expression can be evaluated by using stack data structure as follows

Reading Symbol	Stack Operations	Evaluated Part of Expression
Initially	Stack is Empty 	Nothing
5	push(5) 	Nothing
3	push(3) 	Nothing
+	<pre>value1 = pop() value2 = pop() result = value2 + value1 push(result)</pre> 	<pre>value1 = pop(); // 3 value2 = pop(); // 5 result = 5 + 3; // 8 Push(8)</pre> $(5 + 3)$

8	push(8)		(5 + 3)
2	push(2)		(5 + 3)
-	<pre>value1 = pop() value2 = pop() result = value2 - value1 push(result)</pre>		<pre>value1 = pop(); // 2 value2 = pop(); // 8 result = 8 - 2; // 6 Push(6)</pre> <p>(8 - 2) (5 + 3) , (8 - 2)</p>
*	<pre>value1 = pop() value2 = pop() result = value2 * value1 push(result)</pre>		<pre>value1 = pop(); // 6 value2 = pop(); // 8 result = 8 * 6; // 48 Push(48)</pre> <p>(6 * 8) (5 + 3) * (8 - 2)</p>
\$ End of Expression	result = pop()		Display (result) 48 As final result

LectureNotes.in

Infix Expression $(5 + 3) * (8 - 2) = 48$

Postfix Expression $5 3 + 8 2 - *$ value is **48**

The following is the java program to evaluate a Postfix expression using stack data structure.....

Aim: To write a java program to evaluate a Postfix expression using stack data structure.

```
/*PostfixEvaluation.java*/
```

```
import java.io.*;
class Stack
{
    int a[],top,max;
    Stack(int size)
    {
        max=size;
        a=new int[max];
        top=-1;
    }
    void push(int ele)
    {
        top++;
        a[top]=ele;
    }
    int pop()
    {
        int ele;
        ele=a[top];
        top--;
        return(ele);
    }
}
class Evaluation
{
    int calculate(String s)throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        int n,r=0;
        n=s.length();
        Stack a=new Stack(n);
        for(int i=0;i<n;i++)
        {
            char ch=s.charAt(i);
            if(ch>='0'&&ch<='9')
                a.push((int)(ch-'0'));
            else if((ch>='a'&& ch<='z') || (ch>='A'&&ch<='Z'))
            {
                System.out.println("Enter value for " + ch + ":");
                int op=Integer.parseInt(d.readLine());
                a.push(op);
            }
            else
            {
                int x=a.pop();
                int y=a.pop();
            }
        }
    }
}
```

```
switch(ch)
{
    case '+':r=x+y;
        break;
    case '-':r=y-x;
        break;
    case '*':r=x*y;
        break;
    case '/':r=y/x;
        break;
    default:r=0;
}
a.push(r);
}
}
r=a.pop();
return(r);
}
```

```
class PostfixEvaluation
{
    public static void main(String args[])throws IOException
    {
        DataInputStream d=new DataInputStream(System.in);
        String str;
        while(true)
        {
            System.out.println("Enter the postfix expresion");
            str=d.readLine();
            if(str.equals("stop"))
                break;
            Evaluation e=new Evaluation();
            System.out.println("Result:- "+e.calculate(input));
        }
    }
}
```

Output:

C:\Program Files (x86)\EditPlus 2\launcher.exe

Enter the postfix expresion

ab+23-*

Enter value for a:

12

Enter value for b:

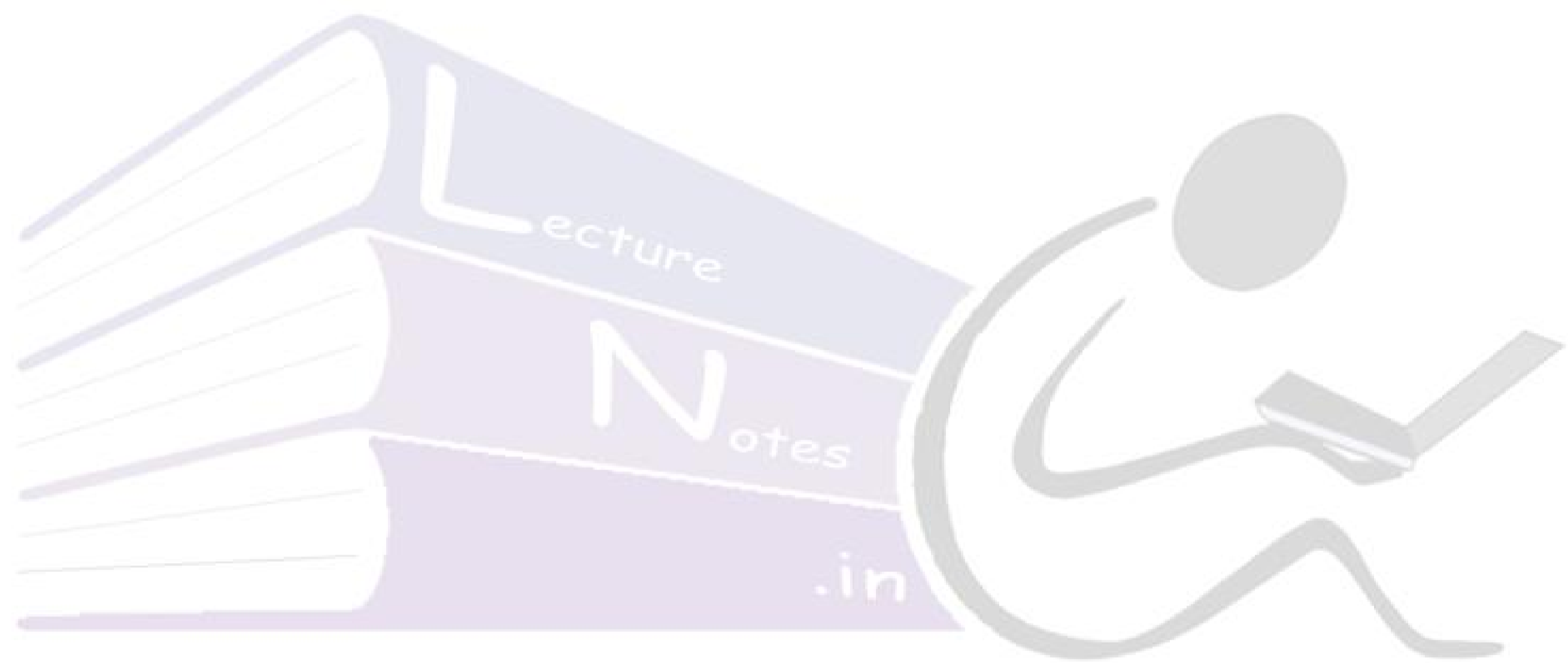
5

Result:- -17

Enter the postfix expresion

stop

Press any key to continue...



LectureNotes.in

LectureNotes.in

UNIT: III

TREE DATA STRUCTURE

Trees: Binary Tree, Definition, Properties, ADT, Array and Linked representations, Implementations and Applications.

Binary Search Trees (BST): Definition, ADT, Operations and Implementations, BST Applications. Threaded Binary Trees, Heap trees.

Introduction:

In linear data structure, data is organized in sequential order and in non-linear data structure; data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

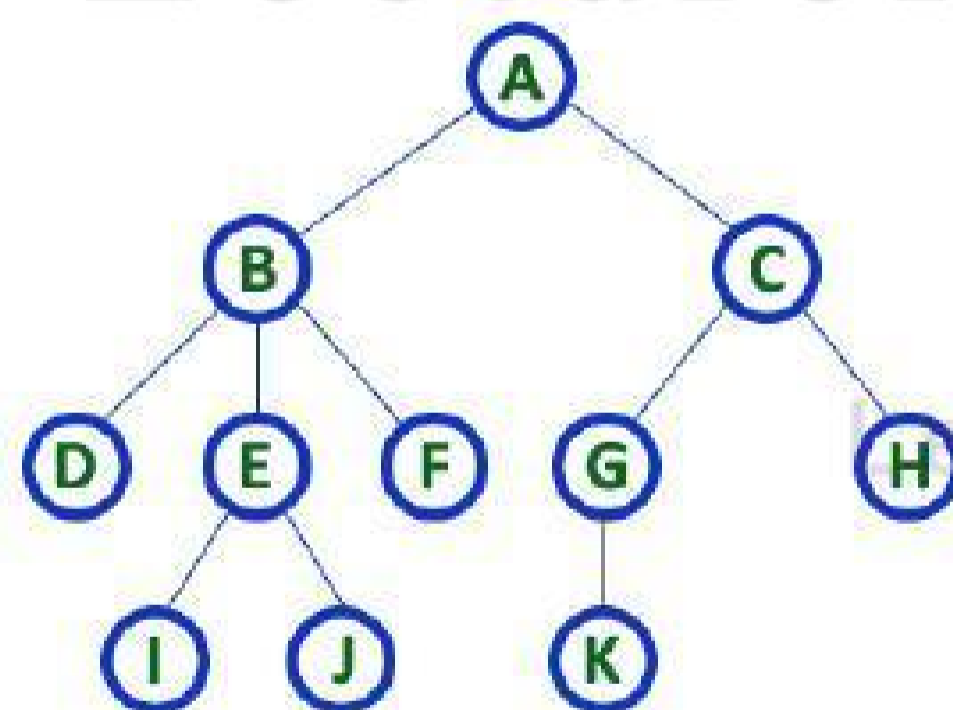
A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition.

In tree data structure, every individual element is called as **Node**. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have N number of nodes then we can have a maximum of $N-1$ number of links.

Example



TREE with 11 nodes and 10 edges

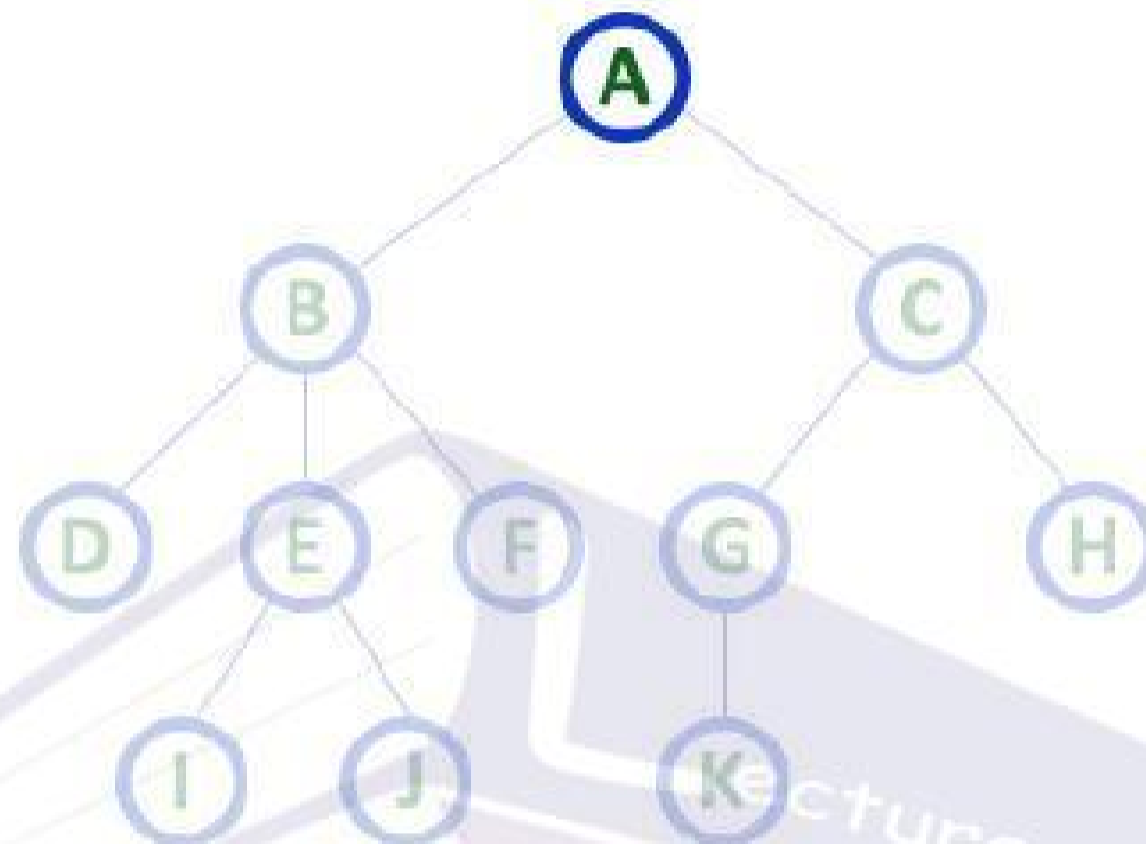
- In any tree with ' N ' nodes there will be maximum of ' $N-1$ ' edges
- In a tree every individual element is called as '**NODE**'

Tree Terminology

In a tree data structure, we use the following terminology...

1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

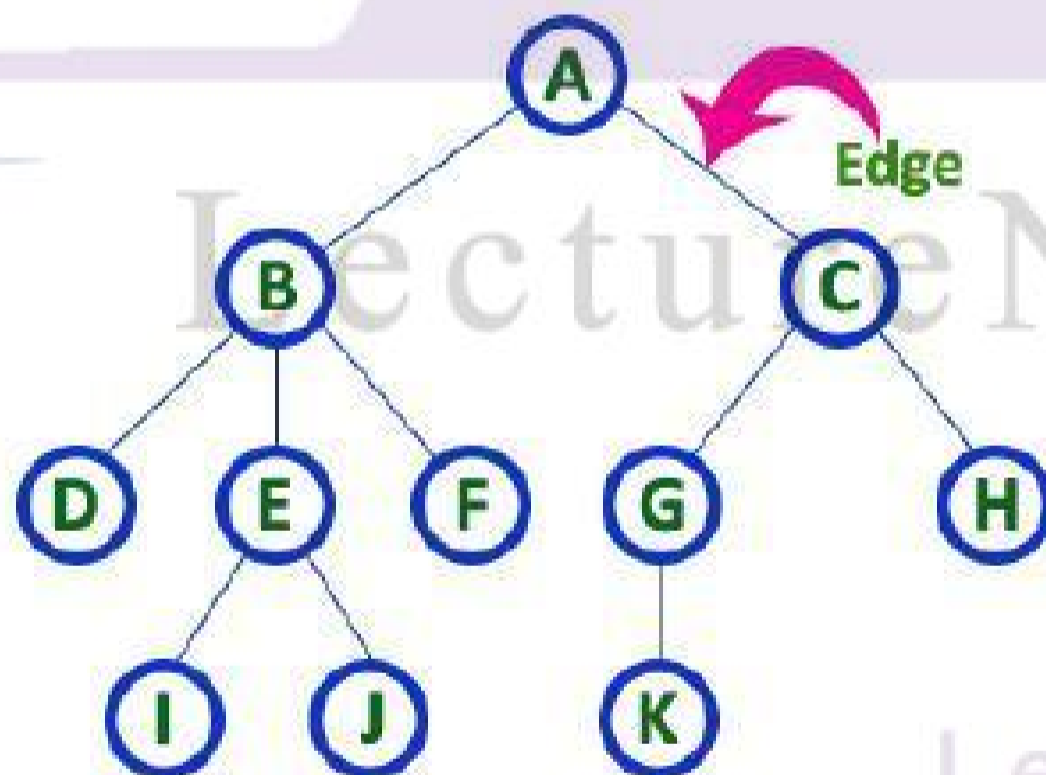


Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node

2. Edge

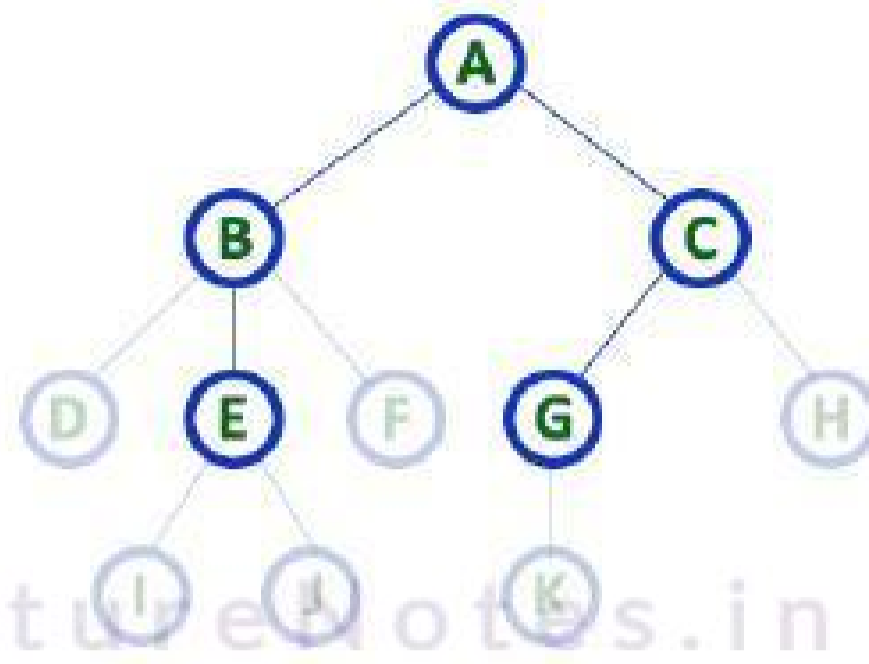
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".



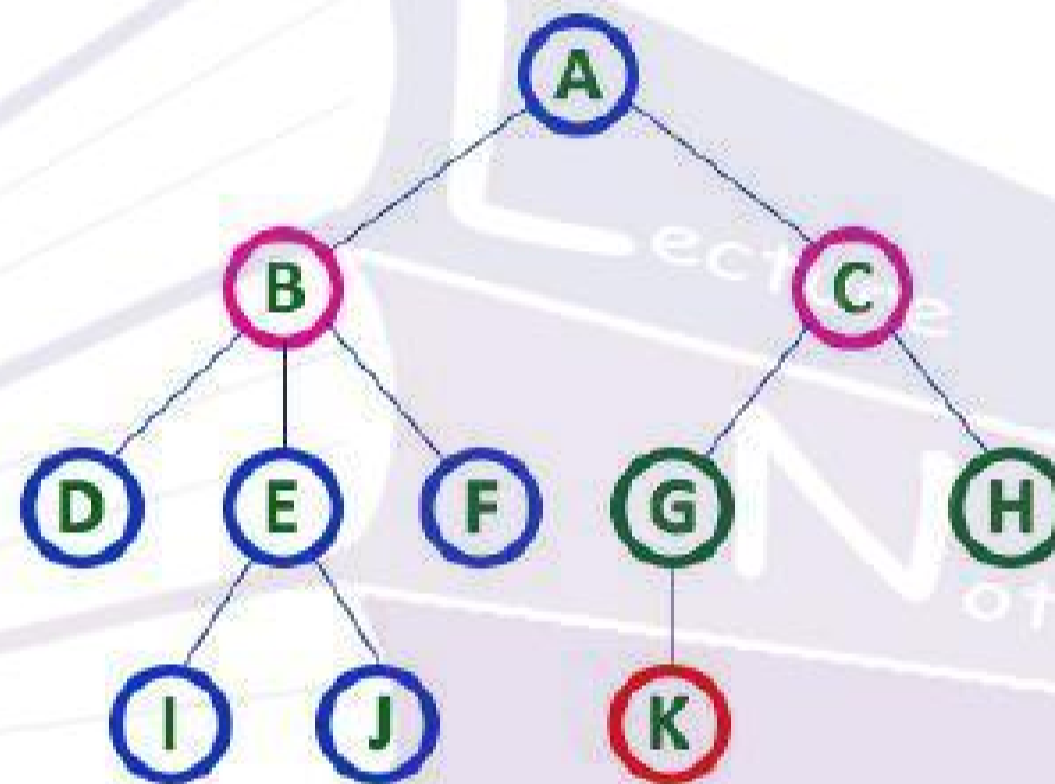
Here A, B, C, E & G are **Parent** nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

LectureNotes.in

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are **Children** of A

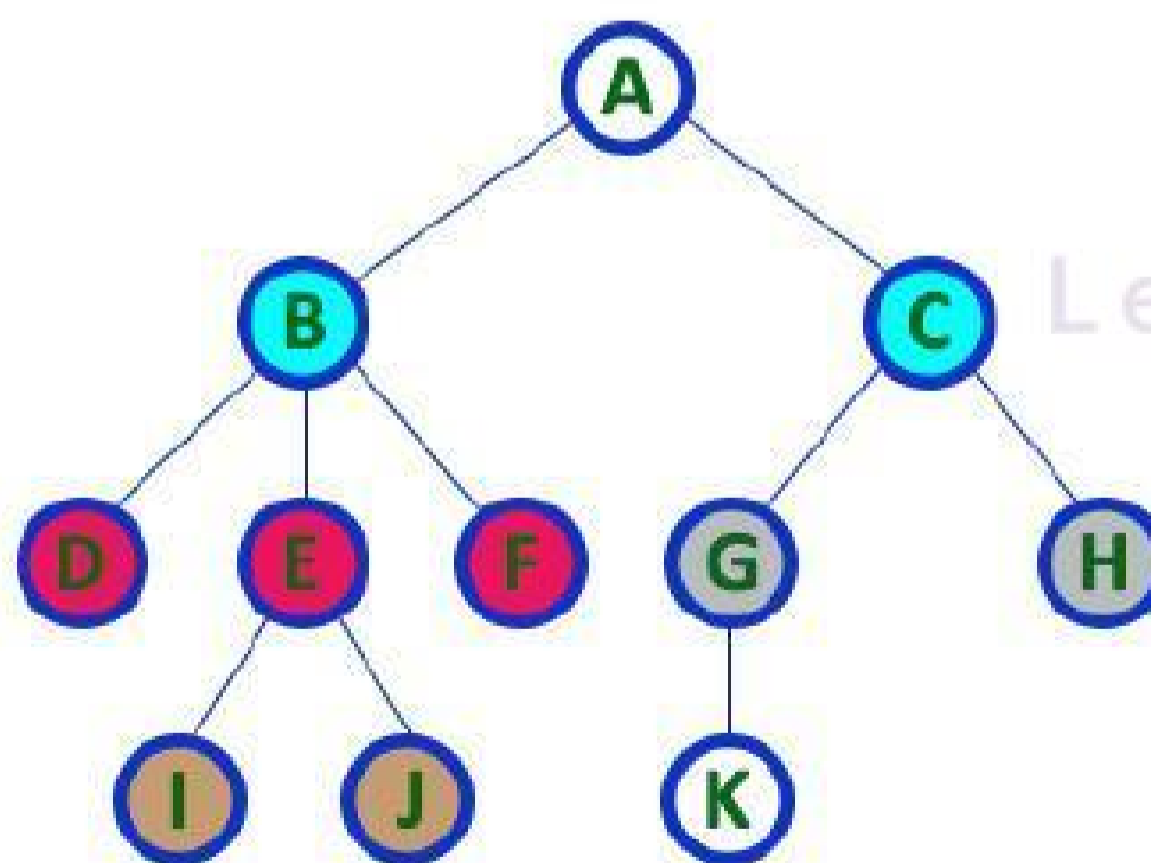
Here G & H are **Children** of C

Here K is **Child** of G

- descendant of any node is called as **CHILD Node**

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



Here B & C are **Siblings**

Here D E & F are **Siblings**

Here G & H are **Siblings**

Here I & J are **Siblings**

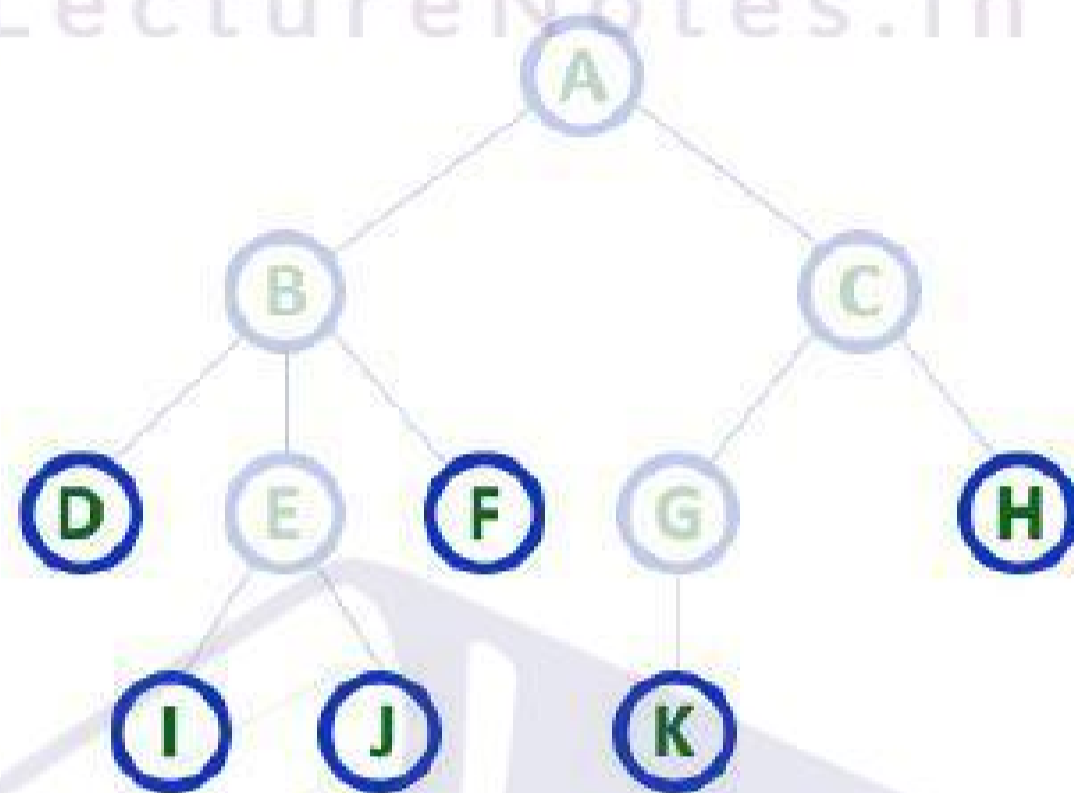
- In any tree the nodes which has same Parent are called '**Siblings**'
- The children of a Parent are called '**Siblings**'

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.

LectureNotes.in



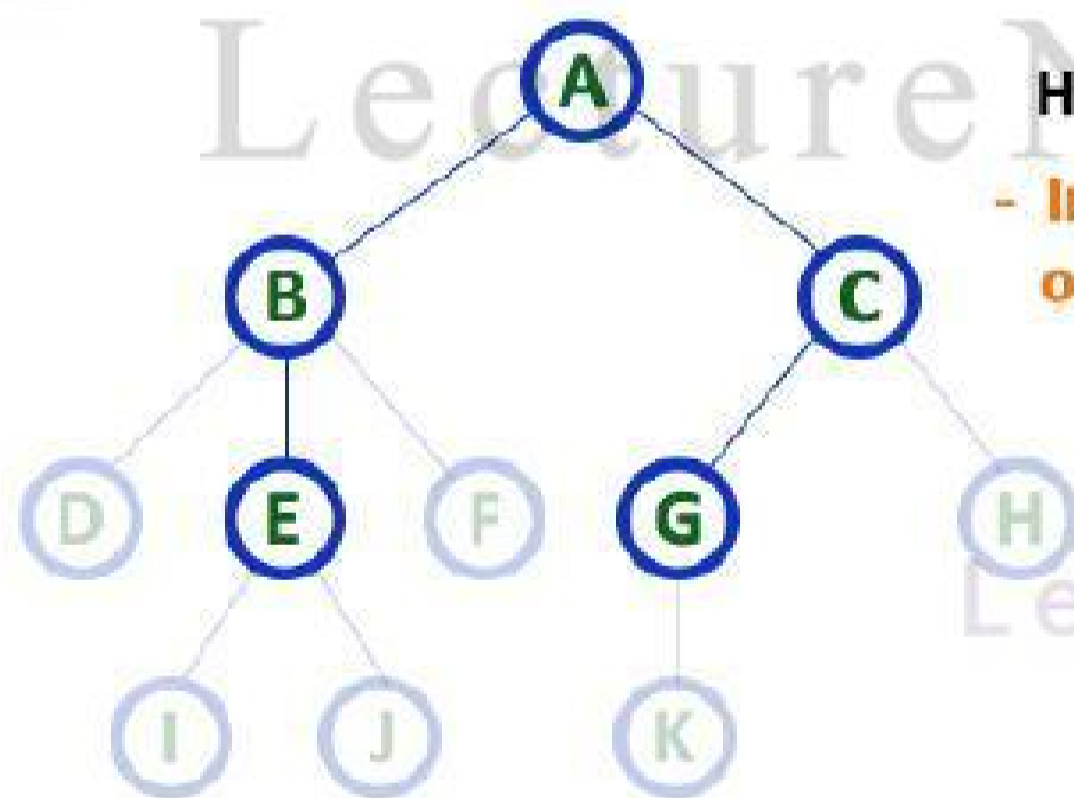
Here D, I, J, F, K & H are **Leaf** nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.



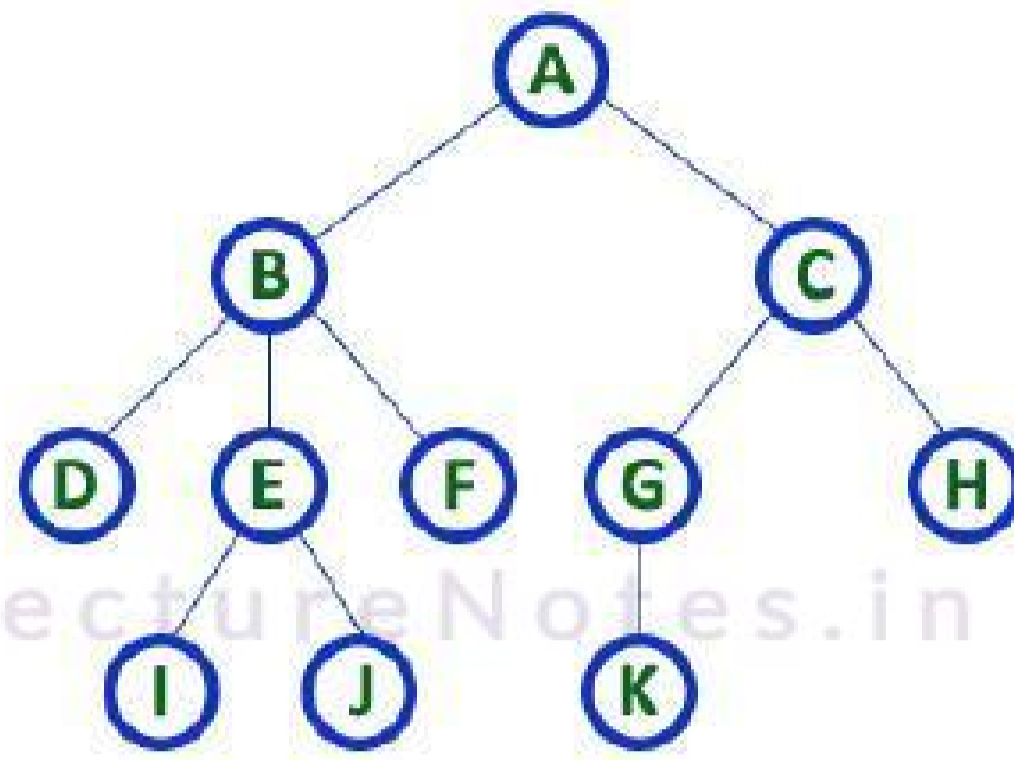
Here A, B, C, E & G are **Internal** nodes

- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

LectureNotes.in

8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



Here Degree of B is 3

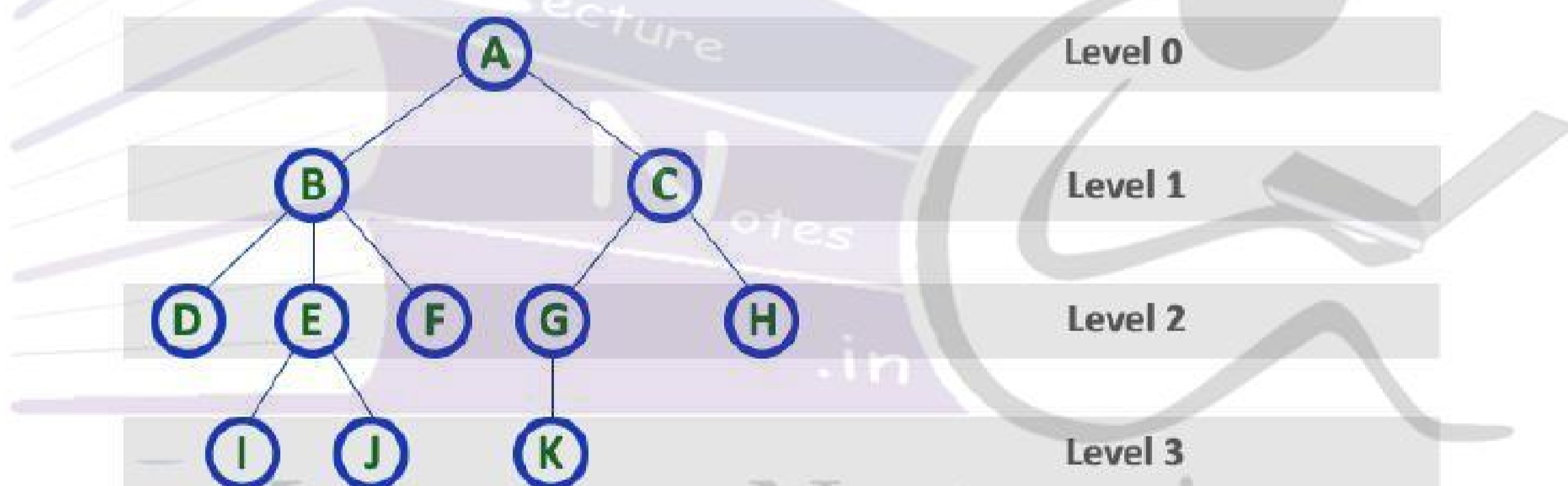
Here Degree of A is 2

Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

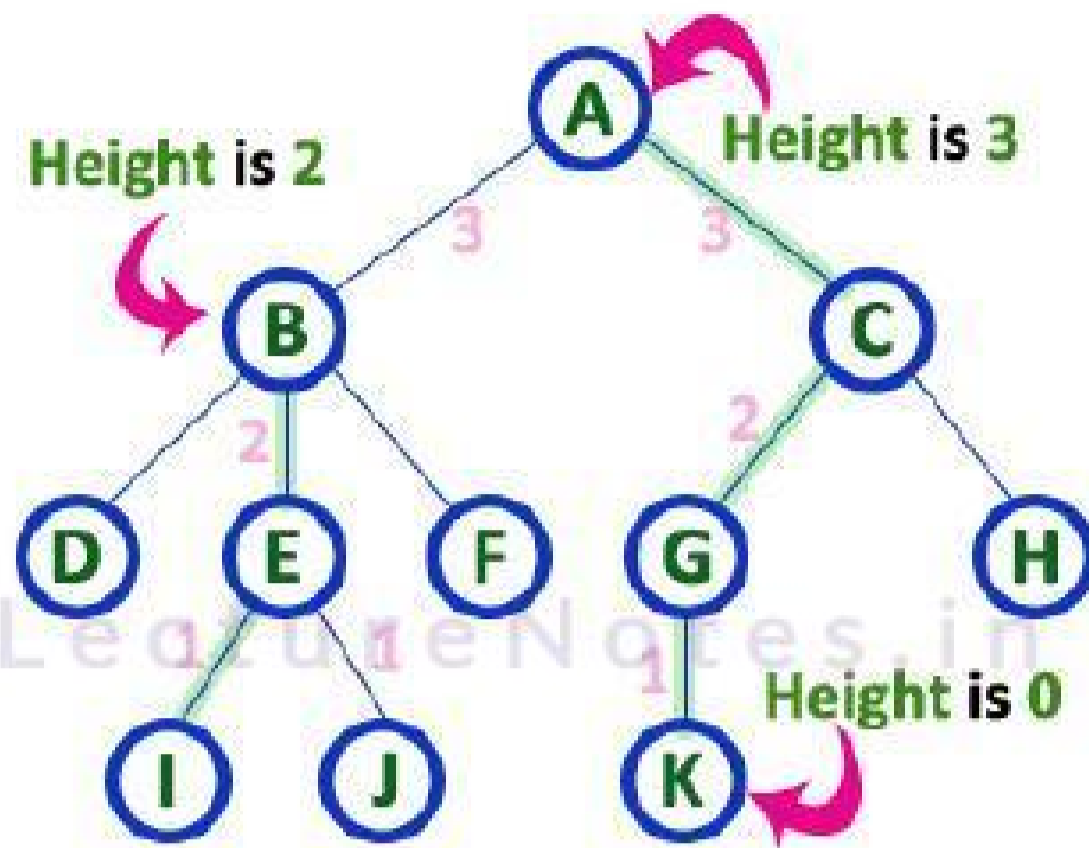
9. Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'**.

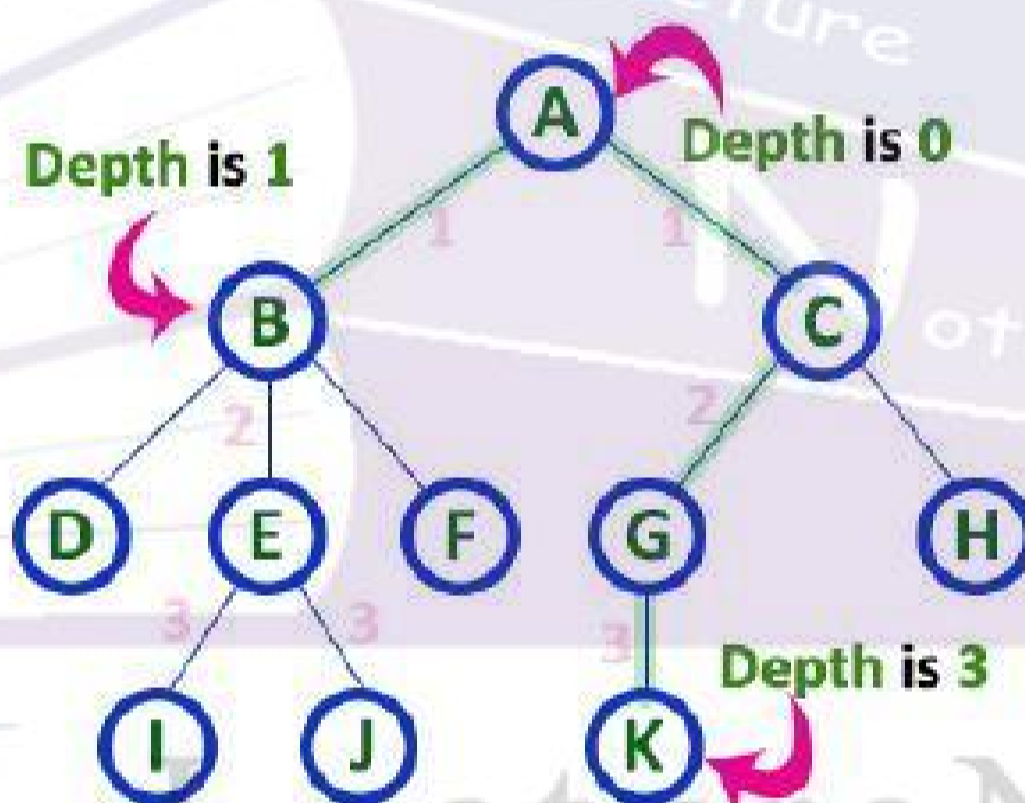


Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'**.

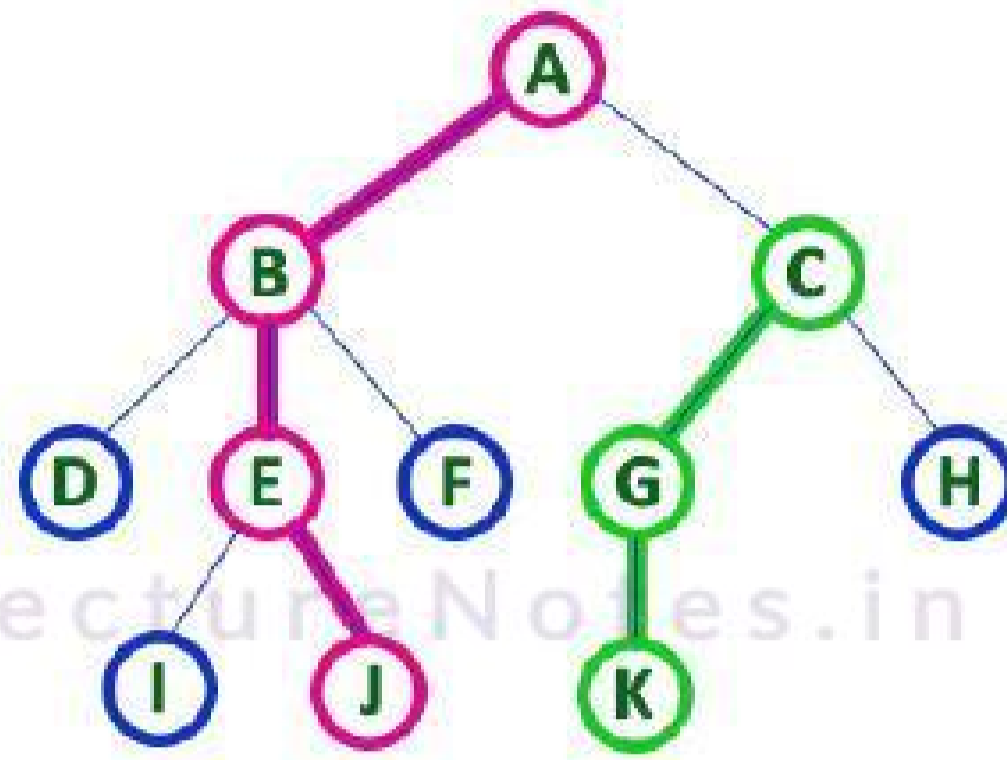


Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4.**



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

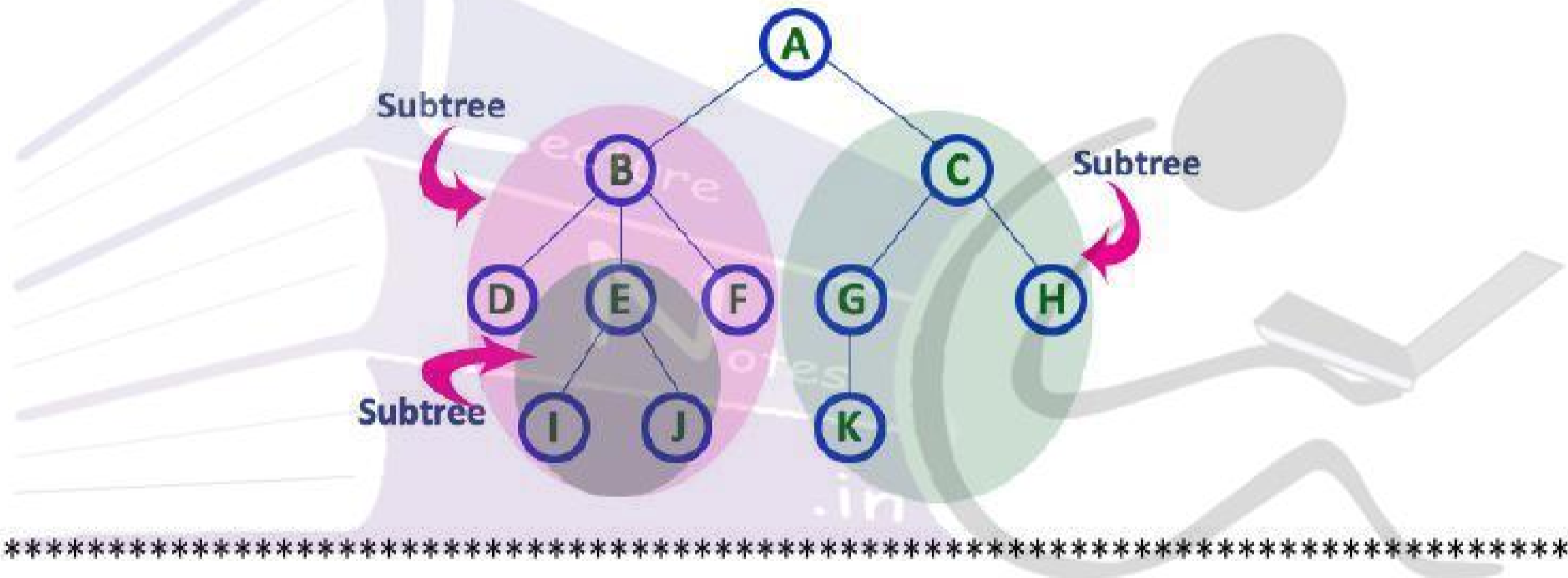
A - B - E - J

Here, 'Path' between C & K is

C - G - K

13. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



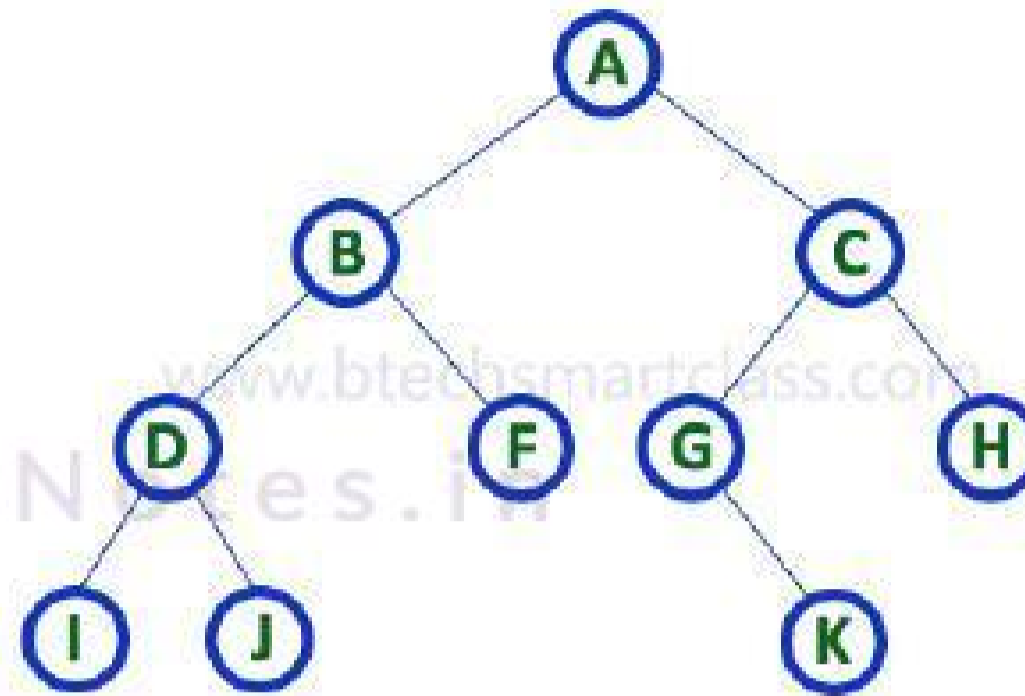
Binary Tree

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as left child and the other is known as right child.

A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example



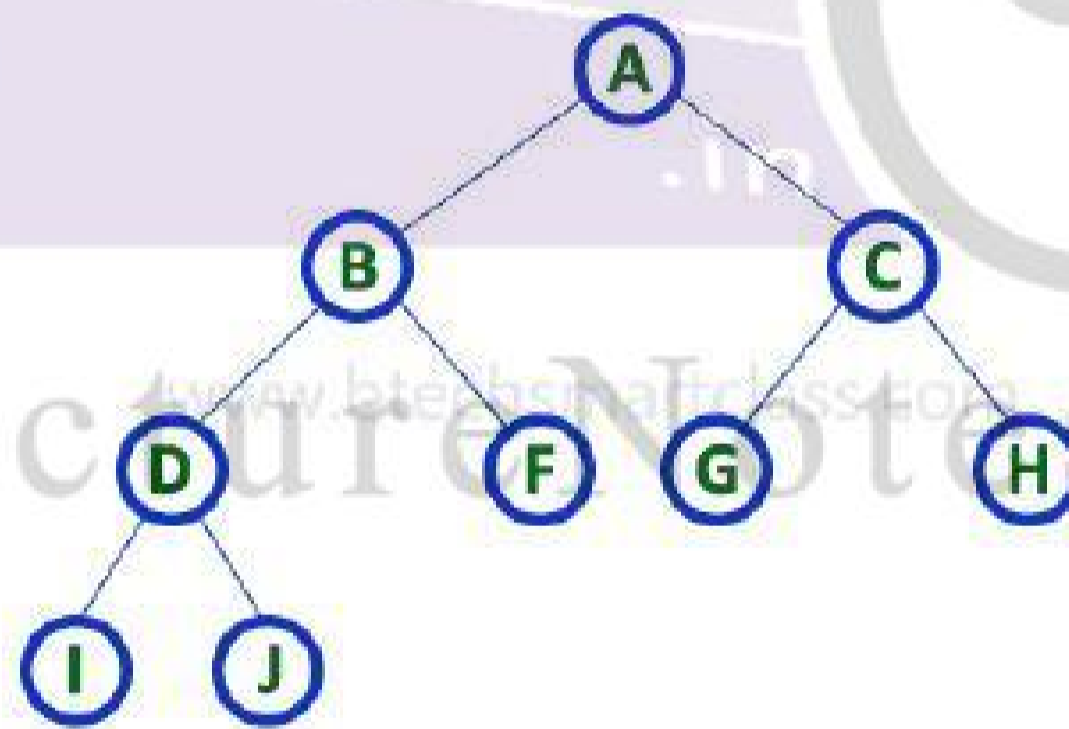
There are different types of binary trees and they are...

1. Strictly Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

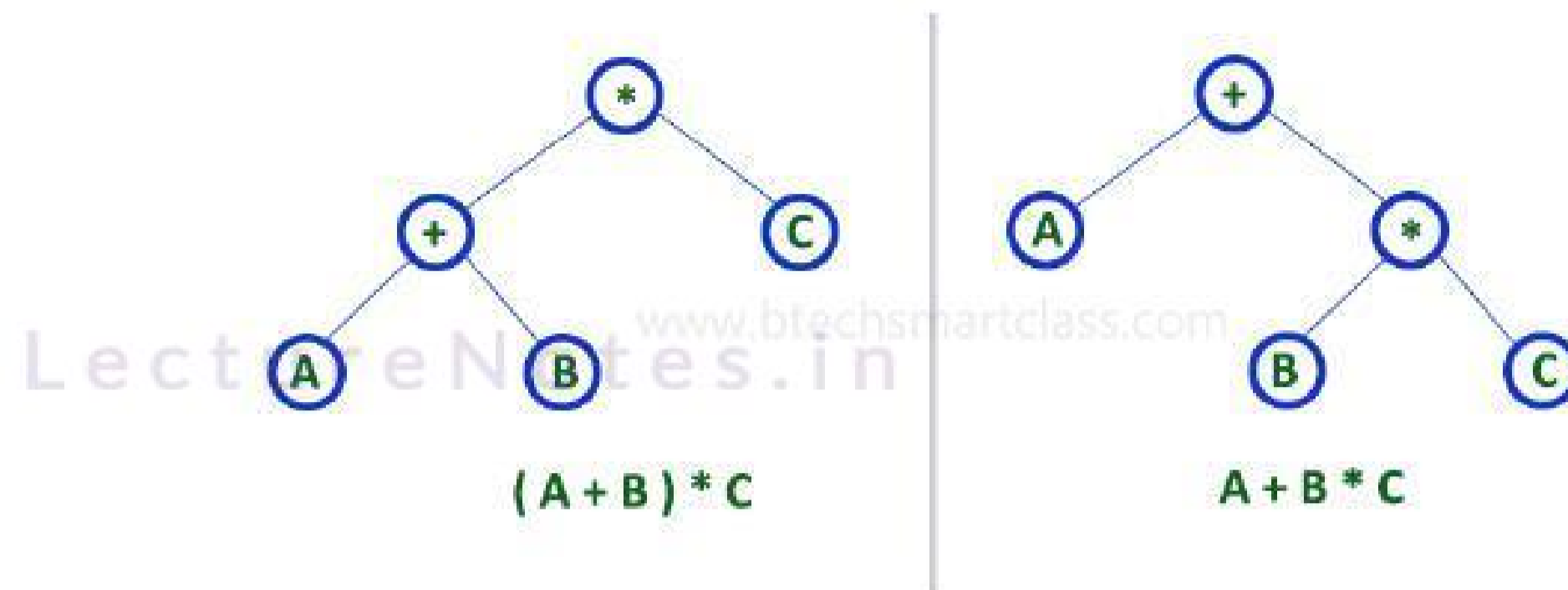
A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree**



Strictly binary tree data structure is used to represent mathematical expressions.

Example

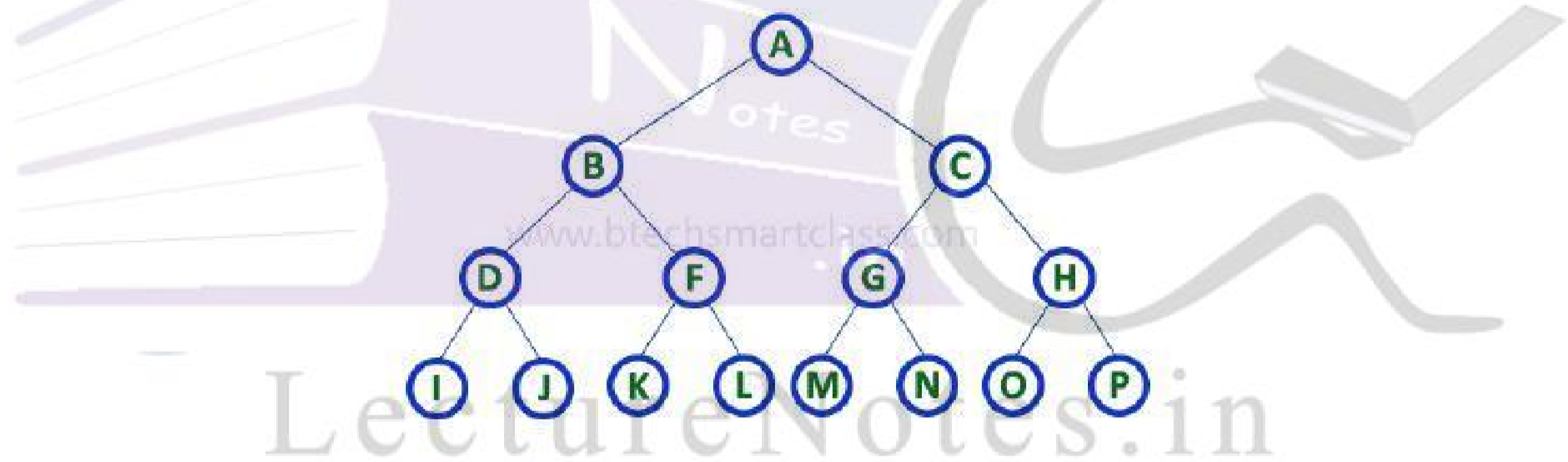


2. Complete Binary Tree

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

At every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.

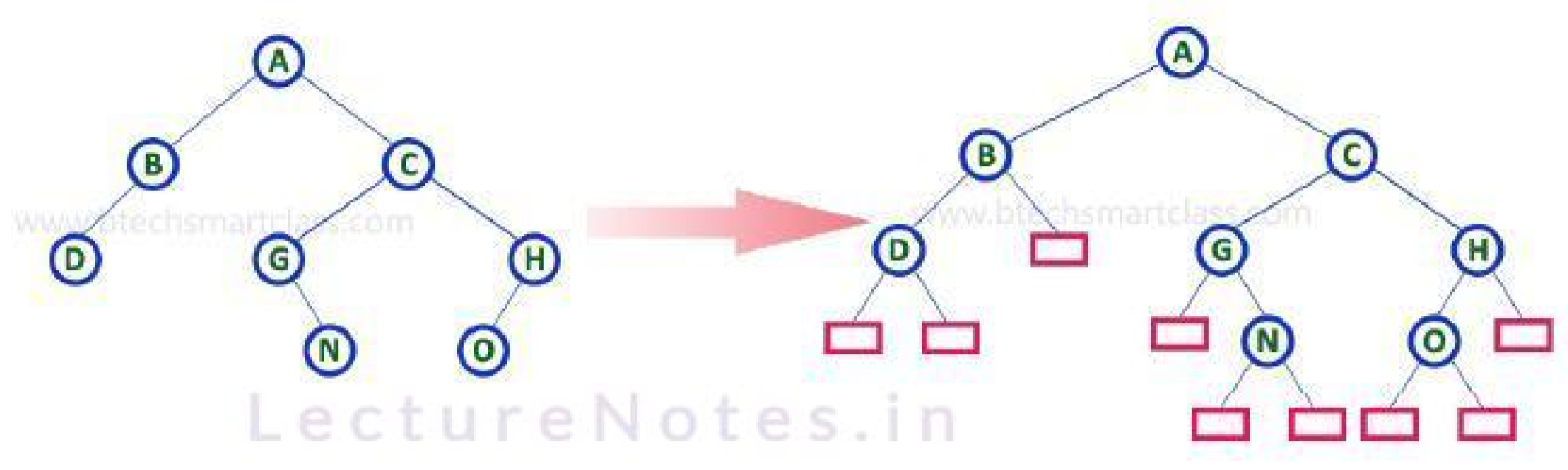
Complete binary tree is also called as **Perfect Binary Tree**



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

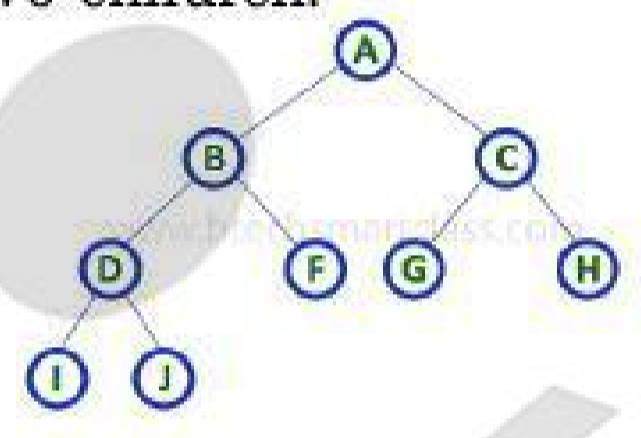
The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



In above figure, a normal binary tree is converted into full binary tree by adding dummy nodes (In pink color).

Binary tree Properties: Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children.

1. A binary tree of n elements has n-1 edges.
2. A binary tree of height h has at least h elements.
3. Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$.
4. The maximum number of nodes at level 'l' of a binary tree is 2^l .
5. In a Binary Tree with N nodes, minimum possible height or minimum number of levels is $\lceil \log_2(N+1) \rceil$.
6. A Binary Tree with L leaves has at least $\lceil \log_2 L \rceil + 1$ levels.
7. In Binary tree, number of leaf nodes is always one more than nodes with two children.

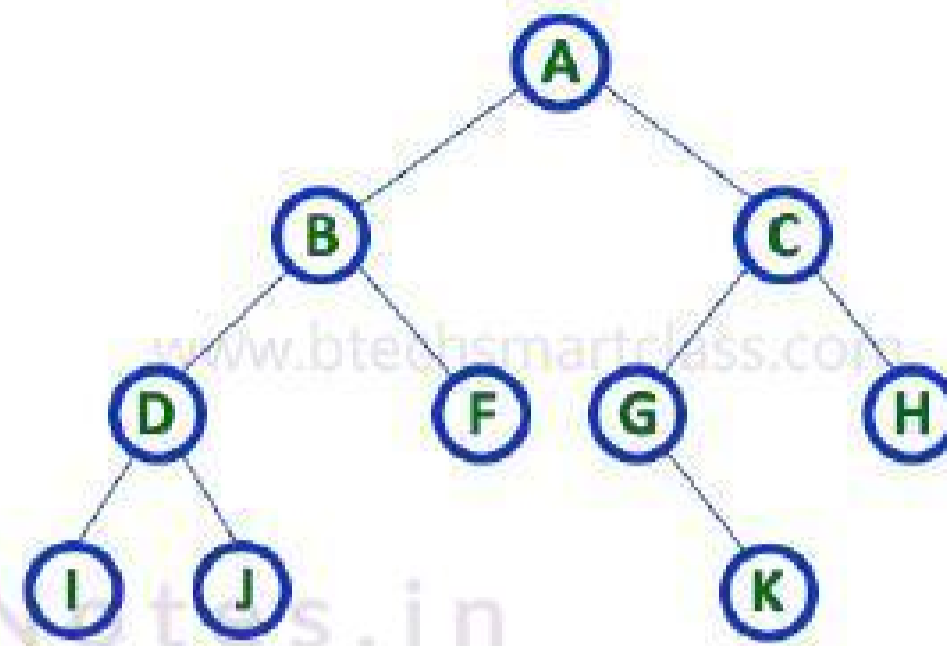


Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

Consider the following binary tree...



LectureNotes.in

1. Array Representation

In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree.

In the array approach, the nodes are stored in an array and are not linked by references. The position of the node in the array corresponds to its position in the tree. The node at index 0 is the root, the node at index 1 is the root's left child, and so on, progressing from left to right along each level of the tree.

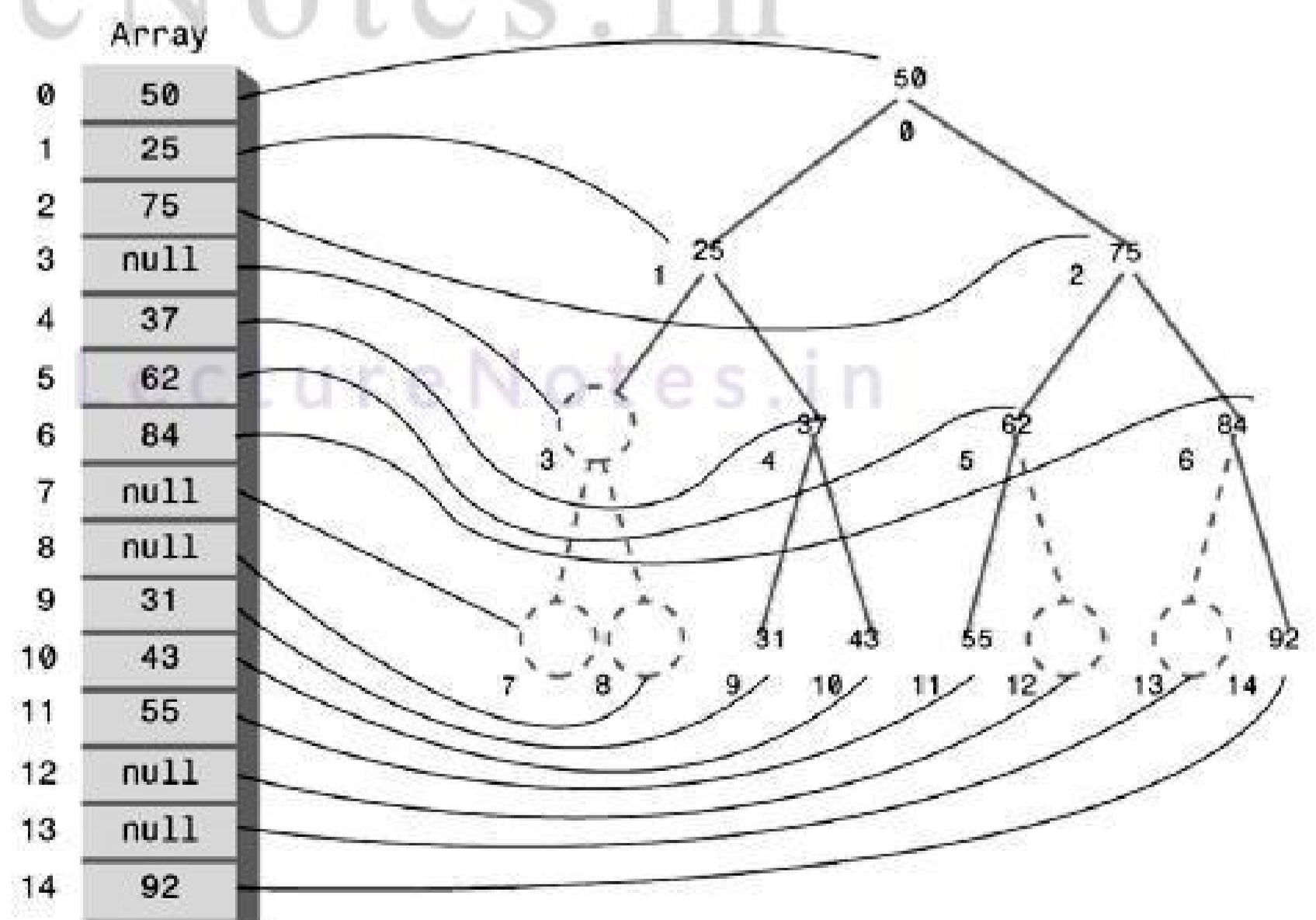
Every position in the tree whether it represents an existing node or not, corresponds to a cell in the array. Adding a node at a given position in the tree means inserting the node into the equivalent cell in the array. Cells representing tree positions with no nodes are filled with zero or null.

Consider the above example of binary tree and it is represented as follows...



A node's children and parent can be found by applying some simple arithmetic to the node's index number in the array. If a node's index number is $index$, then this node's left child is $2 * index + 1$ its right child is $2 * index + 2$ and its parent is $(index - 1) / 2$.

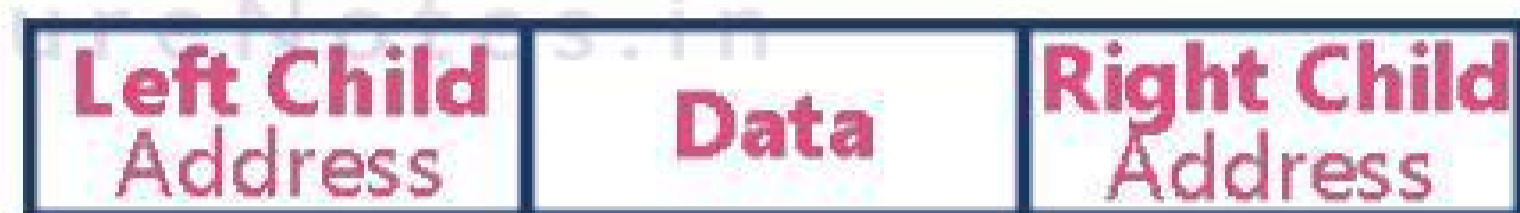
To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.



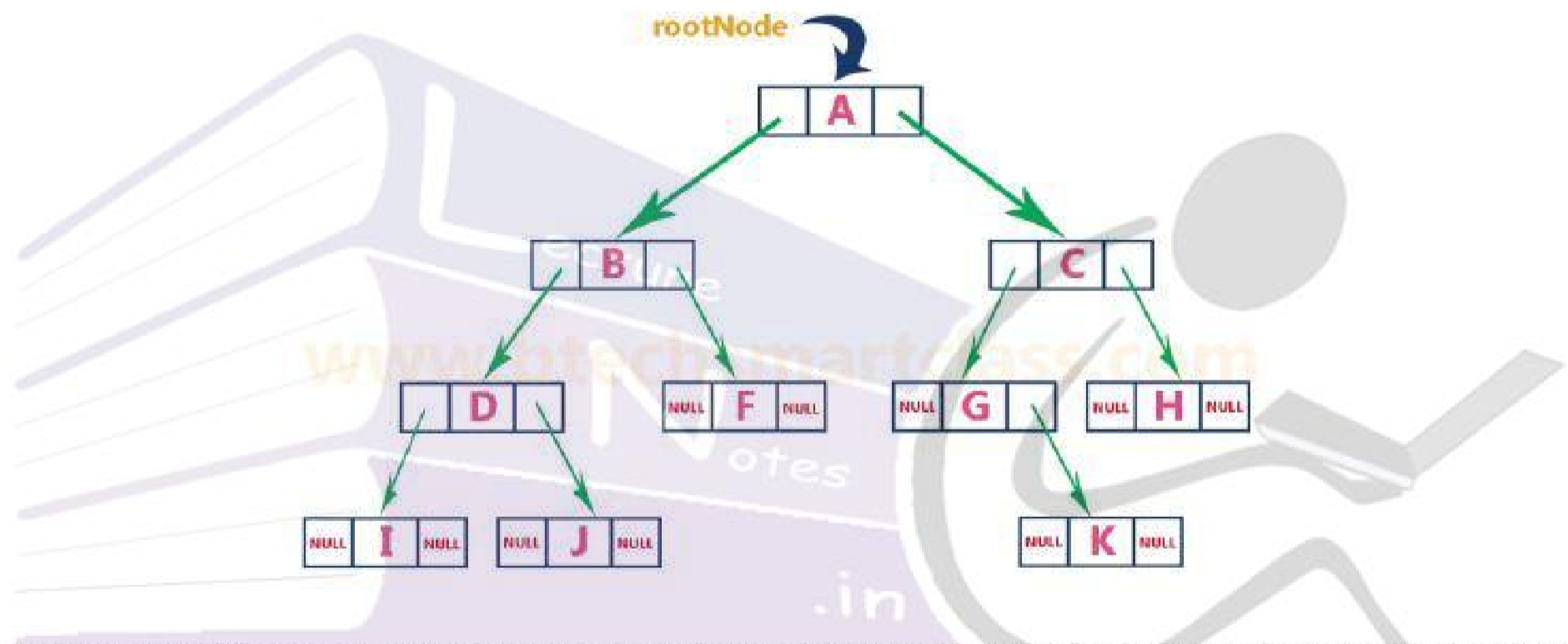
2. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...



The above example of binary tree represented using Linked list representation is shown as follows...



Applications of Binary Tree:

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children.

1. Implementing routing table in router.
2. Implementation of Expression parsers and expression solvers
3. To solve database problem such as indexing.
4. Expression evaluation.
5. Binary trees are used in Huffman coding, which are used as a compression code.
6. Binary trees are used in Binary search trees, which are useful for maintaining records of data without much extra space.
7. One of the most important applications of binary trees is **balanced** binary search trees like Red-Black trees, AVL trees.

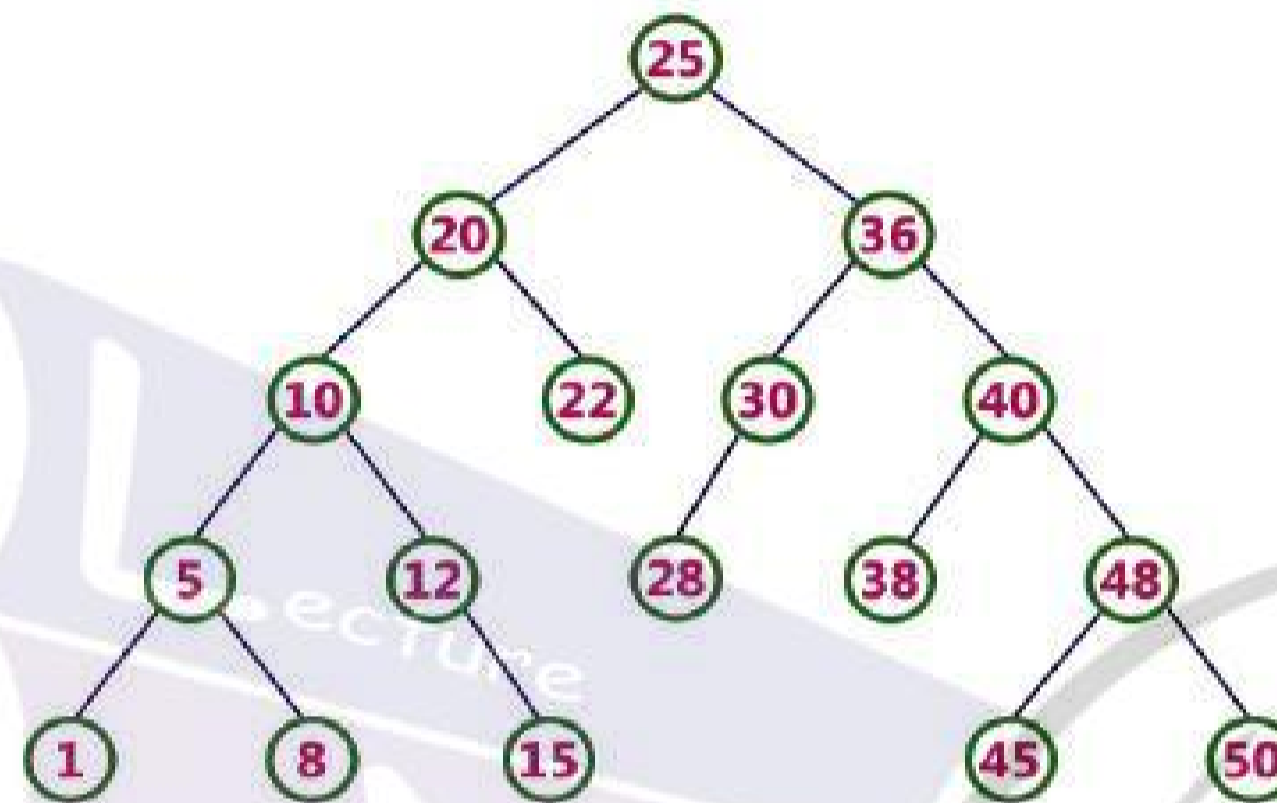
8. Hash trees, similar to hash tables;
9. Abstract syntax trees for compilation of computer languages

Binary Search Tree

Binary Search tree exhibits a special behavior.

- A node's left child must have a value less than its parent's value.
- The node's right child must have a value greater than its parent value.

Example



BST Basic Operations

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Delete** – Deletes an element in a tree.
- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **In order Traversal** – Traverses a tree in an in-order manner.
- **Post order Traversal** – Traverses a tree in a post-order manner.

Insert Operation:

Adding a value to BST can be divided into two stages:

- Search for a place to put a new element;
- Insert the new element to this place.

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left sub tree and insert the data. Otherwise, search for the empty location in the right sub tree and insert the data.

Algorithm

```
If root is NULL
  then create root node
  return
Else If root exists then
  begin
    compare the data with node.data until insertion position is located
    If data is greater than node.data
      goto right subtree
    else
      goto left subtree
  end
end insert data
end If
```

Example:

5, 2, 8, 4, 1

Step 1: If root is NULL then

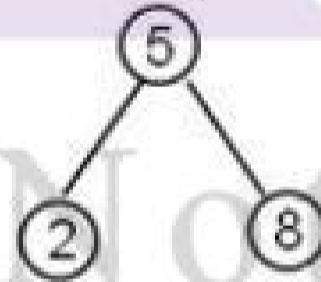
Create root node

5

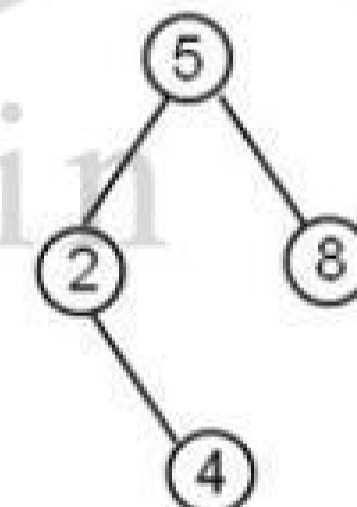
Step 2: compare 2 with 5, $2 < 5$ so insert into the left of 5

2

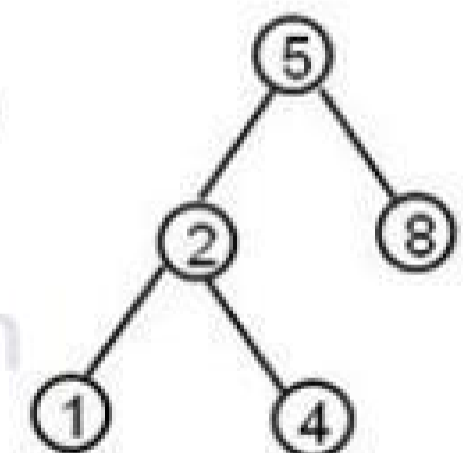
Step 3: $5 < 8$ so insert to the right of 5



Step 4: next element is 4, $4 < 2$ and $4 < 5$ so 4 is add to the right of 2.



Step 5: next element is 1, $1 < 5$, $1 < 2$ so is added to the left of the 2



Search Operation:

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, searches for the element in the left sub tree. Otherwise, search for the element in the right sub tree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
    return root
else
    while data not found
        If data is greater than node.data
            goto right subtree
        else
            goto left subtree
        If data found
            return node
    endwhile
    return data not found
end if
```

Deleting a node from a binary search tree

There are three cases for deleting a node from a binary search tree.

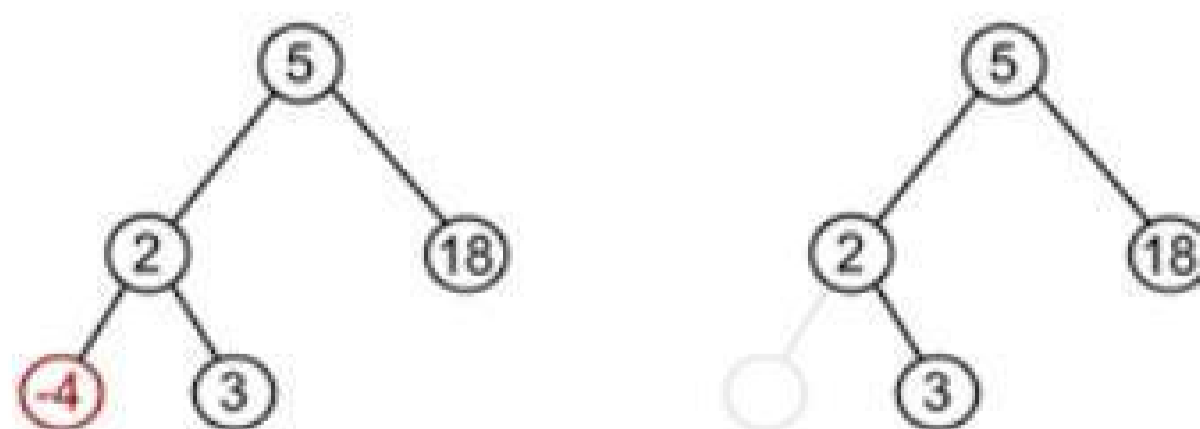
Case 1: Delete a leaf node

Case 2: Delete a node with one child

Case 3: Delete a node with two children

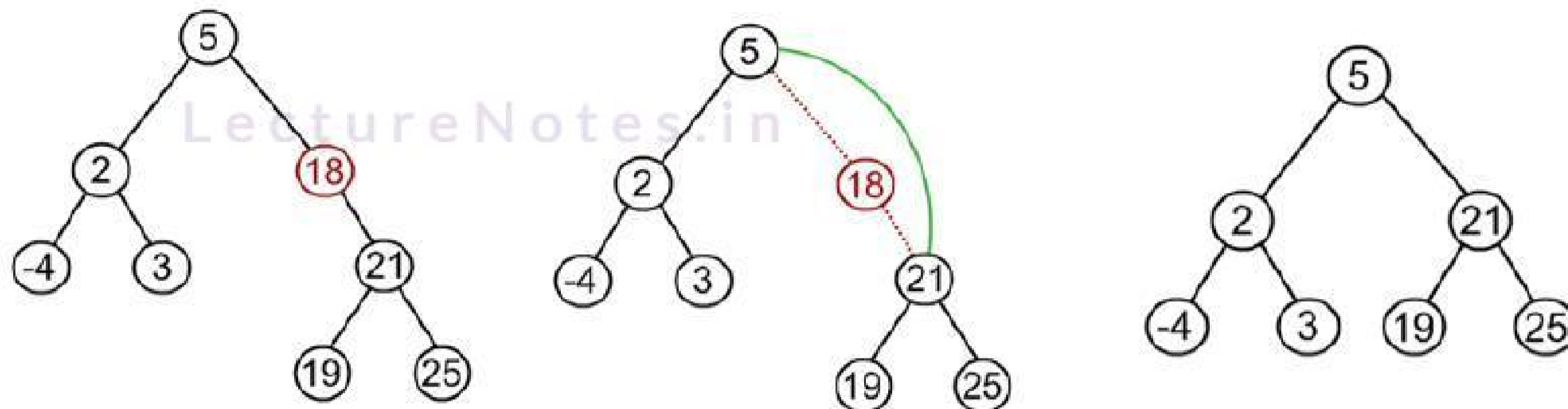
Case 1: Delete a leaf node

- **Step 1:** Find the node to be deleted using **search operation**
- **Step 2:** Delete the node using **free** function (If it is a leaf) and terminate the function



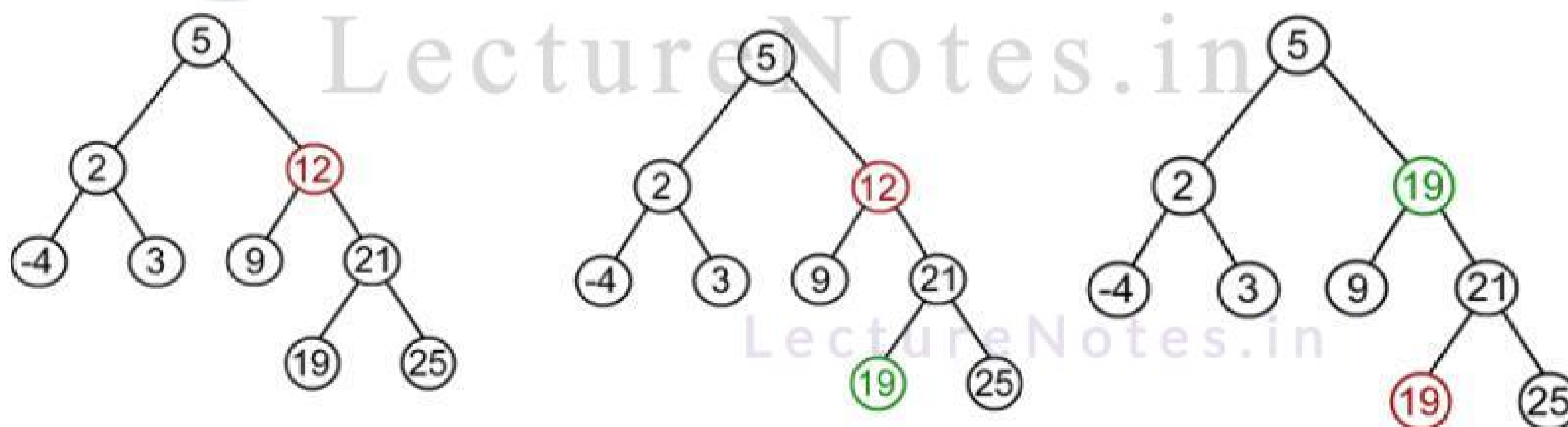
Case 2: Delete a node with one child:

- **Step 1: Find** the node to be deleted using **search operation**
- **Step 2:** If it has only one child, then create a link between its parent and child nodes.
- **Step 3:** Delete the node using **free** function and terminate the function.



Case 3: Delete a node with two children:

- **Step 1: Find** the node to be deleted using **search operation**
- **Step 2:** If it has two children, then find the **largest** node in its **left subtree** (OR) the **smallest** node in its **right subtree**.
- **Step 3: Swap** both **deleting node** and node which found in above step.
- **Step 4:** Then, check whether deleting node came to **case 1** or **case 2** else goto steps 2
- **Step 5:** If it comes to **case 1**, then delete using case 1 logic.
- **Step 6:** If it comes to **case 2**, then delete using case 2 logic.
- **Step 7:** Repeat the same process until node is deleted from the tree.



Find minimum element in the right subtree of the node to be removed. In current example it is 19.

Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.

Tree Traversing Techniques

Tree traversal is a process of moving through a tree in a specified order to process each of the nodes. Each of the nodes is processed only once (although it may be visited more than once). Usually, the traversal process is used to print out the tree.

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

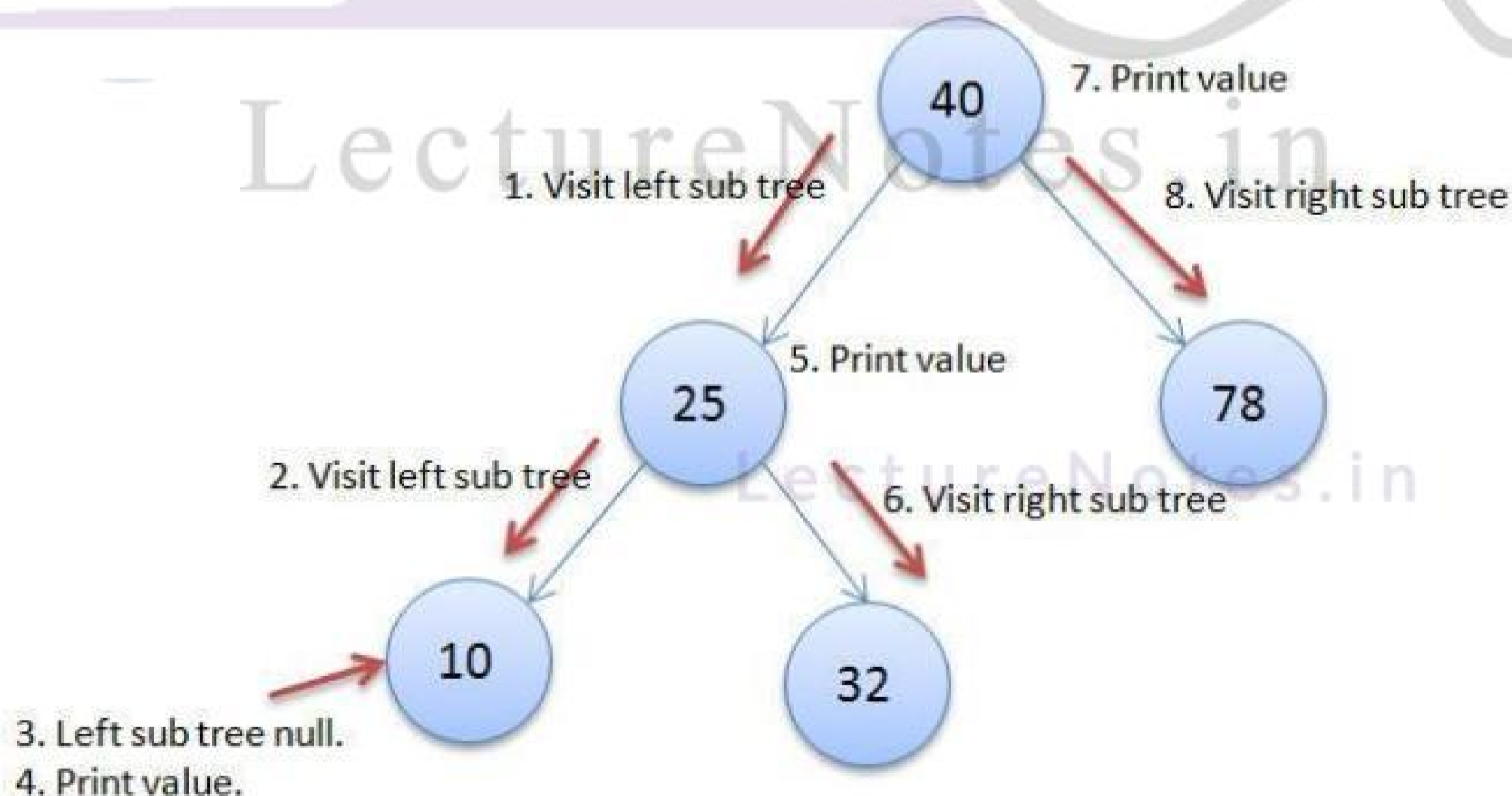
1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

Inorder Traversal

Algorithm Inorder (tree)

1. Traverse the left subtree, i.e., call Inorder (left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder (right-subtree)

In this traversal the left sub tree of the given node is visited first, then the value at the given node is printed and then the right sub tree of the given node is visited. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



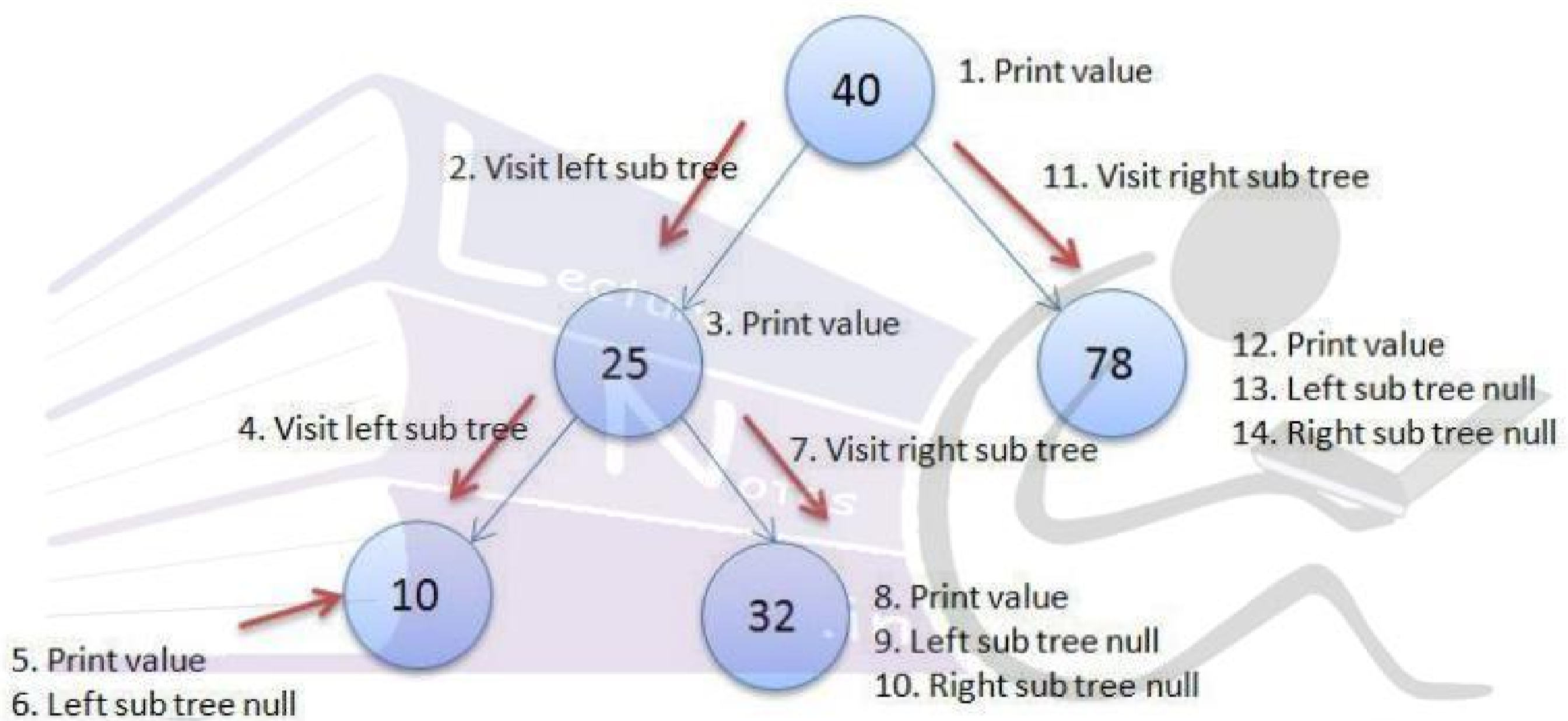
The above INORDER traversal gives: **10, 25, 32, 40, 78**

Preorder traversal

Algorithm Preorder (tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

In this traversal the value at the given node is printed first and then the left sub tree of the given node is visited and then the right sub tree of the given node is visited. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



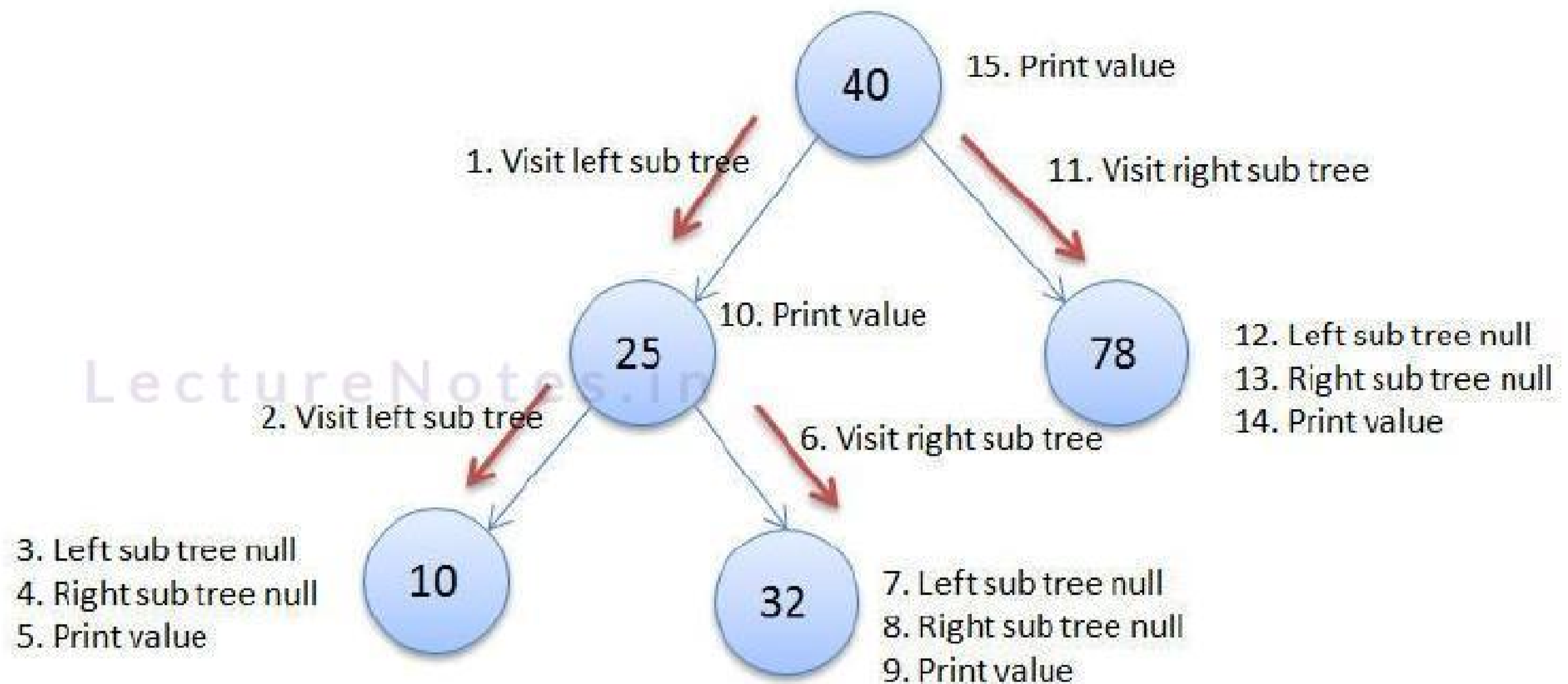
The above PREORDER traversal gives: **40, 25, 10, 32, 78**

Postorder Traversal

Algorithm Postorder (tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

In this traversal the left sub tree of the given node is traversed first, then the right sub tree of the given node is traversed and then the value at the given node is printed. This process is applied recursively all the node in the tree until either the left sub tree is empty or the right sub tree is empty.



The above POSTORDER traversal gives: **10, 32, 25, 78, 40**

Threaded Binary Tree

A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position.

In any binary tree linked list representation, there are more number of NULL pointer than actual pointers.

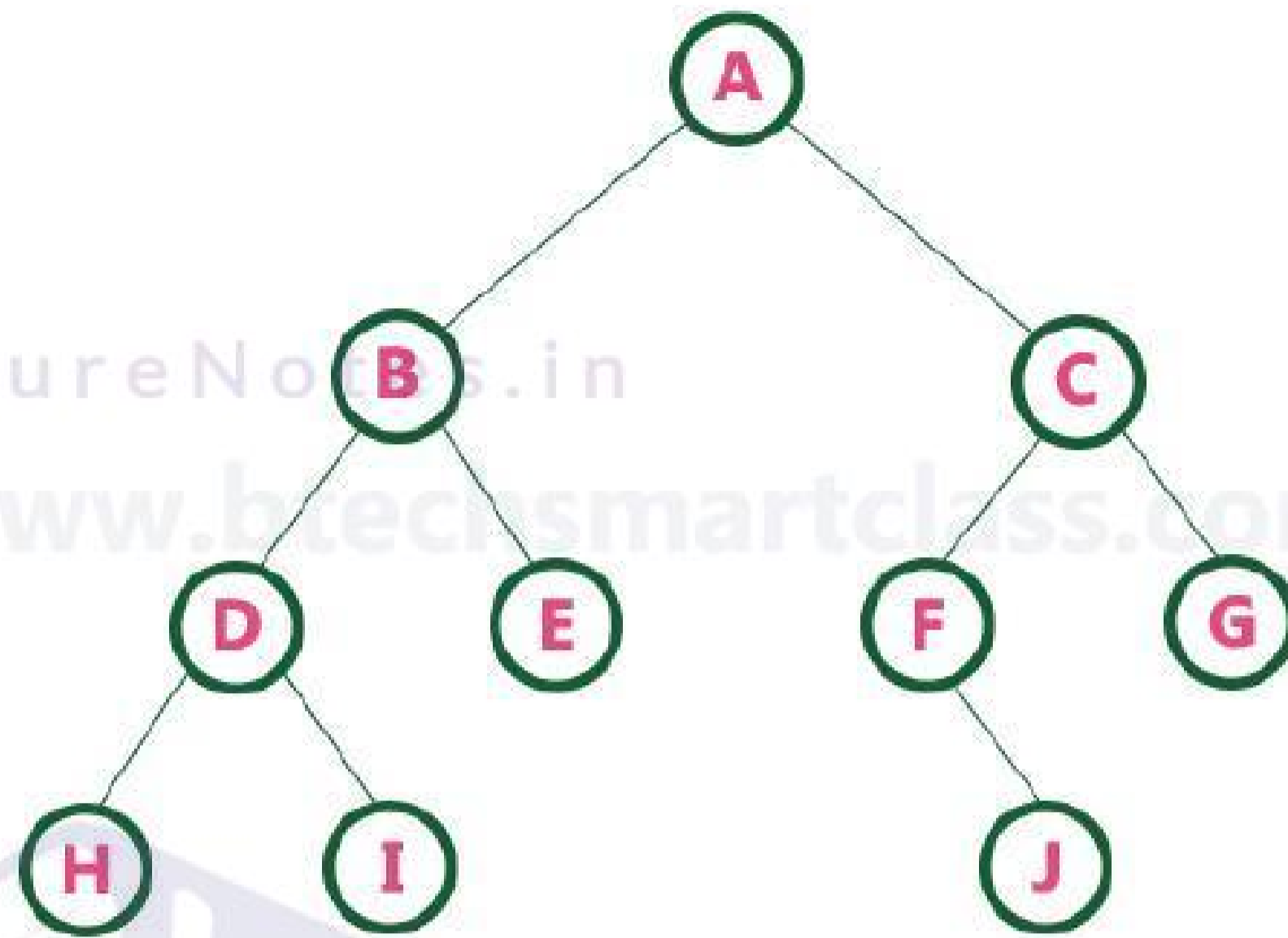
Generally, in any binary tree linked list representation, if there are $2N$ number of reference fields, then $N+1$ number of reference fields are filled with NULL ($N+1$ are NULL out of $2N$). This NULL pointer does not play any role except indicating there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "**Threaded Binary Tree**", which make use of NULL pointer to improve its traversal processes. In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called **threads**

Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.

If there is no in-order predecessor or in-order successor, then it point to root node.

Consider the following binary tree...



To convert above binary tree into threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...

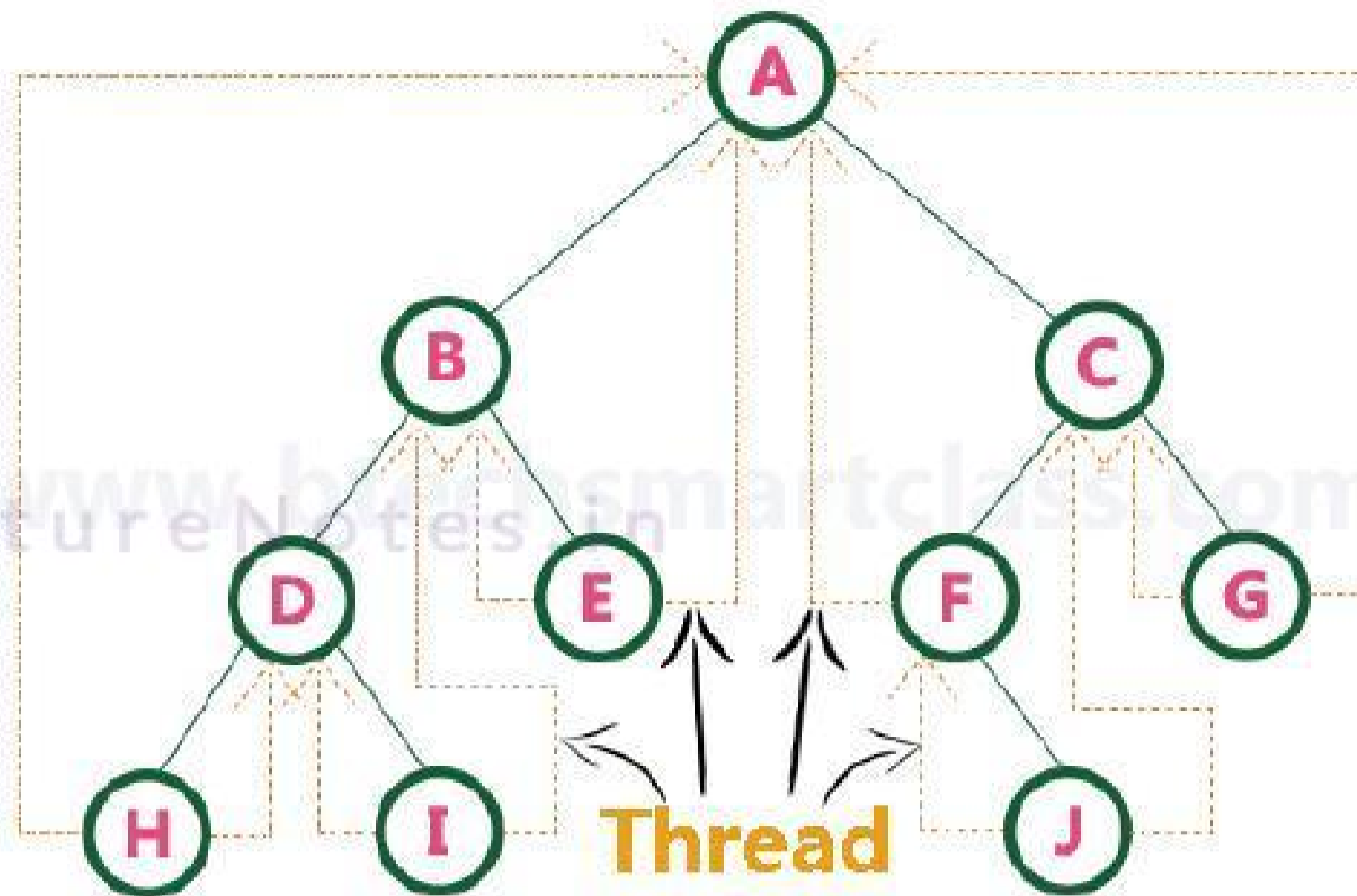
H - D - I - B - E - A - F - J - C - G

When we represent above binary tree using linked list representation, nodes **H, I, E, F, J** and **G** left child pointers are NULL.

This NULL is replaced by address of its in-order predecessor, respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A.

And nodes **H, I, E, J** and **G** right child pointers are NULL. This NULL pointers are replaced by address of its in-order successor, respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

Above example binary tree become as follows after converting into threaded binary tree.



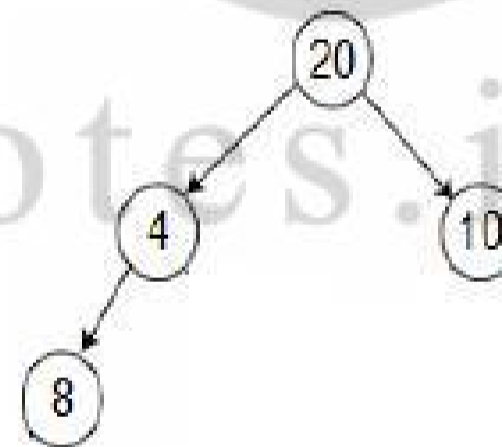
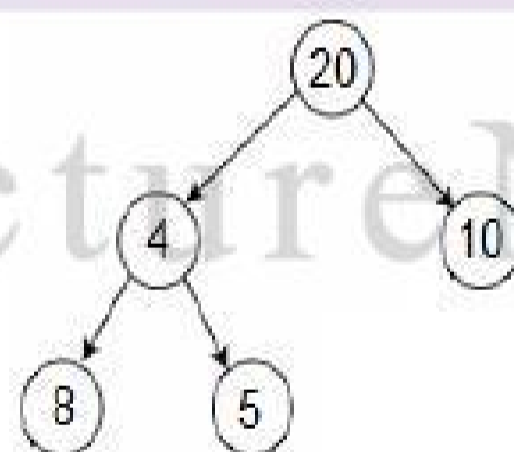
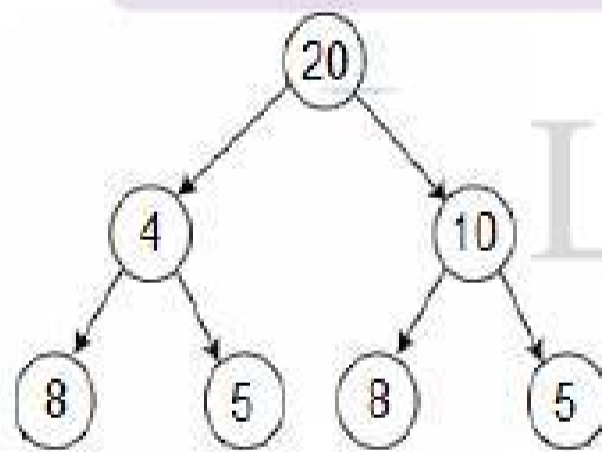
In above figure threads are indicated with dotted links.

Heap Tree

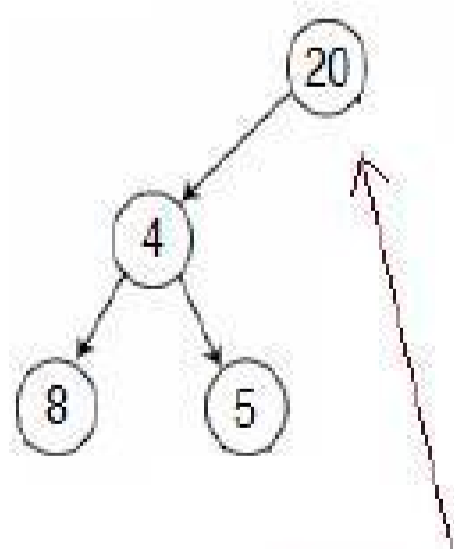
A Complete binary heap tree is a binary tree in which every node other than the leaves has two children. In complete binary tree at every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Complete Binary Tree

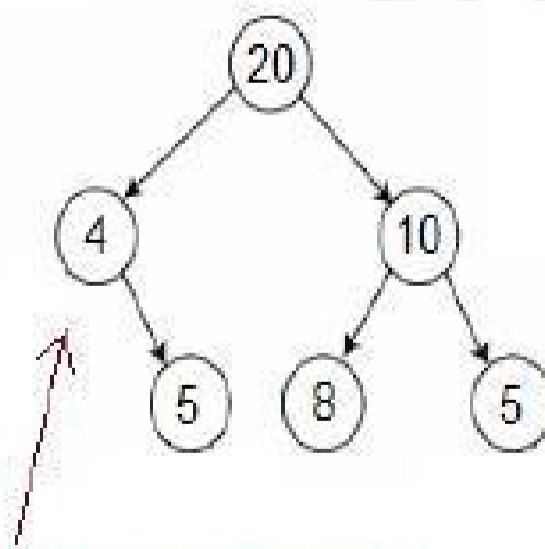
A binary tree is complete if it follows the sequence of 1. Root Node, 2. Left Node, 3. Right Node. It cannot break the ordering. A Tree is still called complete if node has no Right child, left child. If Node has no left child but has right child present then it is not called complete because it has not followed order sequence.



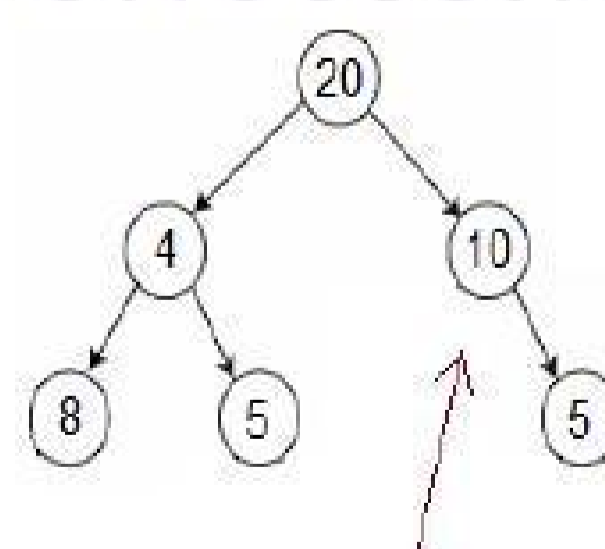
Not Complete Binary Tree



Childs of Node 4 is present without right child of Node 20.



Left child of Node 4 is not present while right child of Node 4 is present.



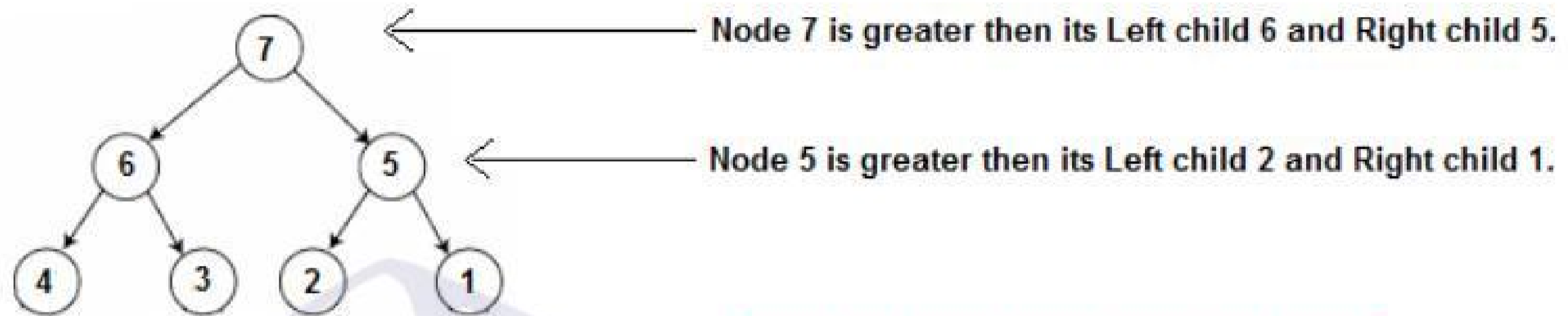
Left child of Node 10 is not present while right child is present.

Depending on the ordering, a heap is called a max-heap or a min-heap.

In a Max-heap, the keys of parent nodes are always greater than or equal to those of the children. In max-heap, Largest element of the Tree is always at top(Root Node).

In a Min-heap, the keys of parent nodes are less than or equal to those of the children. In min-heap, Smallest element of the Tree is always at top(Root Node).

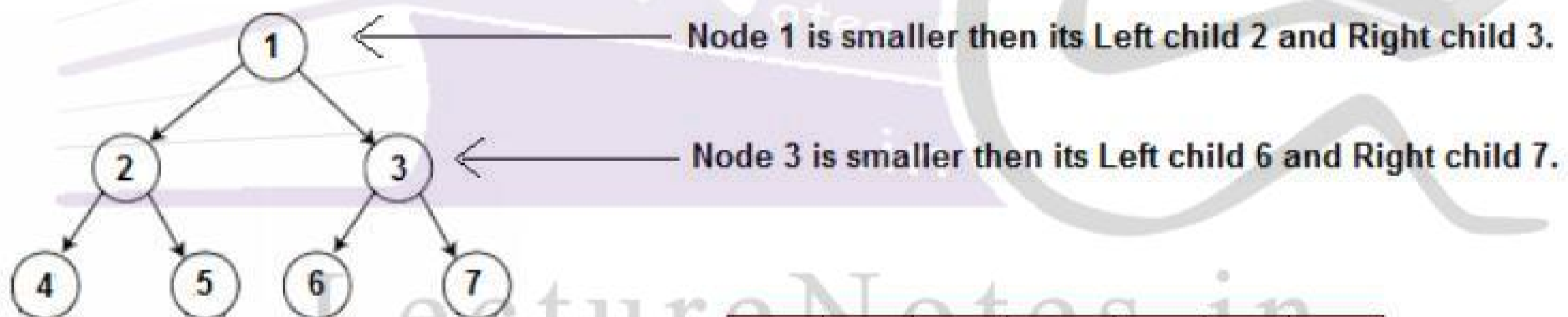
Max Heap Binary Tree



Array representation of above binary Tree:

7	6	5	4	3	2	1
---	---	---	---	---	---	---

Min Heap Binary Tree



Array representation of above binary Tree:

1	2	3	4	5	6	7
---	---	---	---	---	---	---

LectureNotes.in

Implementation of Binary Search Tree Algorithm

```
import java.util.*;
// Represents a node in the Binary Search Tree.
class Node
{
    public int value;
    public Node left;
    public Node right;
    public Node(int value)
    {
        this.value = value;
    }
}
// Represents the Binary Search Tree.
class BinarySearchTree
{
    public Node root;
    public BinarySearchTree insert(int value)
    {
        Node node = new Node(value);
        if (root == null)
        {
            root = node;
            return this;
        }
        insertRec(root, node);
        return this;
    }
    private void insertRec(Node latestRoot, Node node)
    {
        if (latestRoot.value > node.value)
```

```

        {
            if (latestRoot.left == null)
            {
                latestRoot.left = node;
            }
            else
            {
                insertRec(latestRoot.left, node);
            }
        }
        else
        {
            if (latestRoot.right == null)
            {
                latestRoot.right = node;
            }
            else
            {
                insertRec(latestRoot.right, node);
            }
        }
    }
}

```

```

// Printing the contents of the tree in an inorder way
public void printInorder()
{
    printInOrderRec(root);
    System.out.println("");
}
//Helper method to recursively print the contents in an inorder way
private void printInOrderRec(Node currRoot)
{

```



```

        if (currRoot == null)
        {
            return;
        }
        printInOrderRec(currRoot.left);
        System.out.print(currRoot.value + ", ");
        printInOrderRec(currRoot.right);
    }
    //Printing the contents of the tree in a Preorder way.
    public void printPreorder()
    {
        printPreOrderRec(root);
        System.out.println("");
    }
    // Helper method to recursively print the contents in a Preorder way
    private void printPreOrderRec(Node currRoot)
    {
        if (currRoot == null)
        {
            return;
        }
        System.out.print(currRoot.value + ", ");
        printPreOrderRec(currRoot.left);
        printPreOrderRec(currRoot.right);
    }
    // Printing the contents of the tree in a Postorder way.
    public void printPostorder()
    {
        printPostOrderRec(root);
        System.out.println("");
    }

```

```

// Helper method to recursively print the contents in a Postorder way
private void printPostOrderRec(Node currRoot)
{
    if (currRoot == null)
    {
        return;
    }
    printPostOrderRec(currRoot.left);
    printPostOrderRec(currRoot.right);
    System.out.print(currRoot.value + ", ");
}
}

public class BinarySearchTreeDemo
{
    public static void main(String[] args)
    {
        int ch,ele;
        BinarySearchTree bst = new BinarySearchTree();
        Scanner sc=new Scanner(System.in);
        do
        {
            System.out.println("1.Insert");
            System.out.println("2.Inorder");
            System.out.println("3.Preorder");
            System.out.println("4.Postorder");
            System.out.println("5. Exit");
            System.out.println("Enter your choice:");
            ch=sc.nextInt();
            switch(ch)
            {
                case 1: System.out.println("Enter element to insert:");

```

```

        ele=sc.nextInt();
        bst.insert(ele);
        break;
    case 2: System.out.println("Inorder traversal");
            bst.printInorder();
            break;
    case 3: System.out.println("Preorder Traversal");
            bst.printPreorder();
            break;
    case 4: System.out.println("Postorder Traversal");
            bst.printPostorder();
            break;
    case 5: System.exit(0);
            }
        }while(ch!=5);
    }
}

```

Output:

```

1.Insert
2.Inorder
3.Preorder
4.Postorder
5. Exit
Enter your choice:
1
Enter element to insert:
10
1.Insert
2.Inorder
3.Preorder
4.Postorder
5. Exit
Enter your choice:
1
Enter element to insert:
20

```

```

1.Insert
2.Inorder
3.Preorder
4.Postorder
5. Exit
Enter your choice:
2
Inorder traversal
10, 20,
1.Insert
2.Inorder
3.Preorder
4.Postorder
5. Exit
Enter your choice:

```

UNIT: IV

GRAPHS

Graphs – Graph and its Representation, Graph Traversals, Connected Components, Basic Searching Techniques, Minimal Spanning Trees

Graph

Graph is a non linear data structure; it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

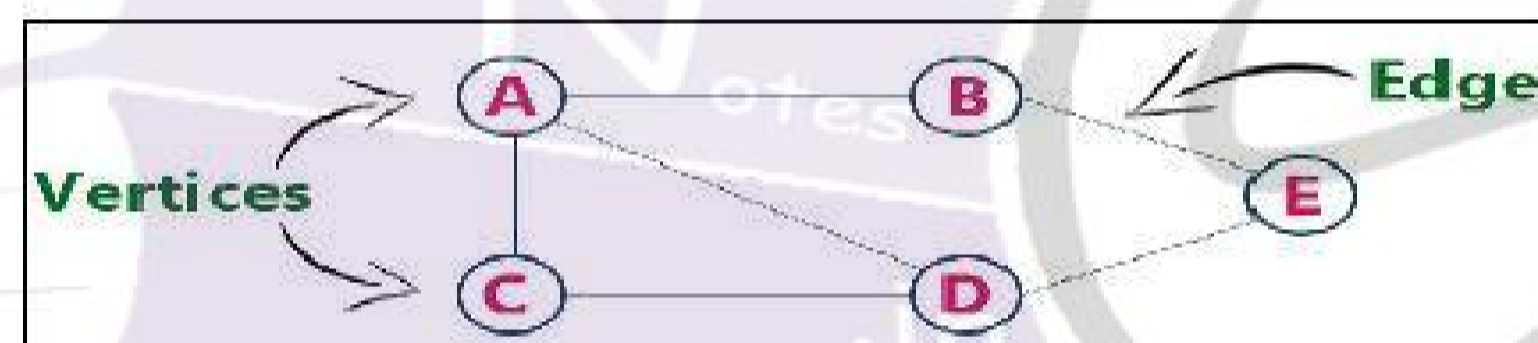
Generally, a graph G is represented as $G = (V, E)$, where V is set of vertices and E is set of edges.

The following is a graph with 5 vertices and 6 edges. This graph G can be defined as

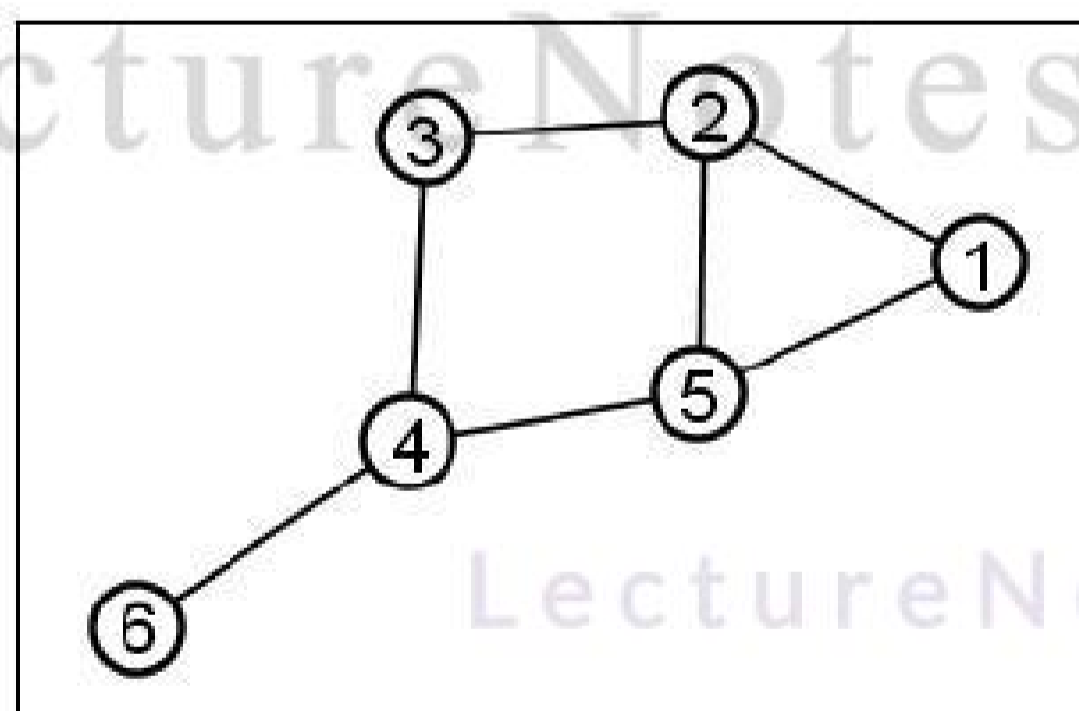
$$G = (V, E)$$

Where $V = \{A, B, C, D, E\}$ and

$$E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}.$$



Example:



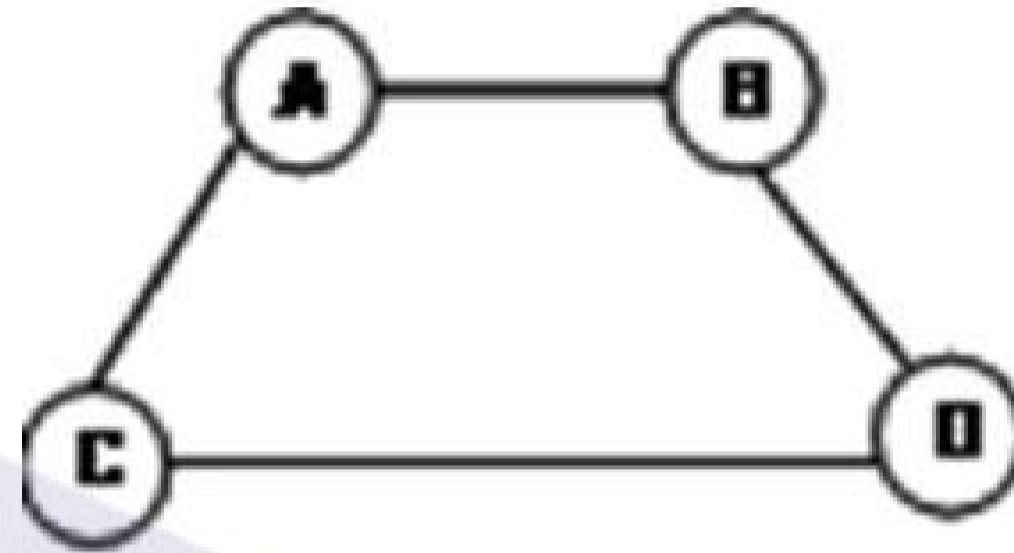
The picture above represents the following graph:

- $V = \{1, 2, 3, 4, 5, 6\}$
- $E = \{\{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 5\}, \{3, 4\}, \{4, 5\}, \{4, 6\}\}$
- $G = (V, E)$

Graph Terminology

Vertex and Edge

- Each node of the graph is represented as a vertex. The labeled circle represents vertices. Thus, A to D are vertices.
- **Edge** – Edge represents a path between two vertices or a line between two vertices.



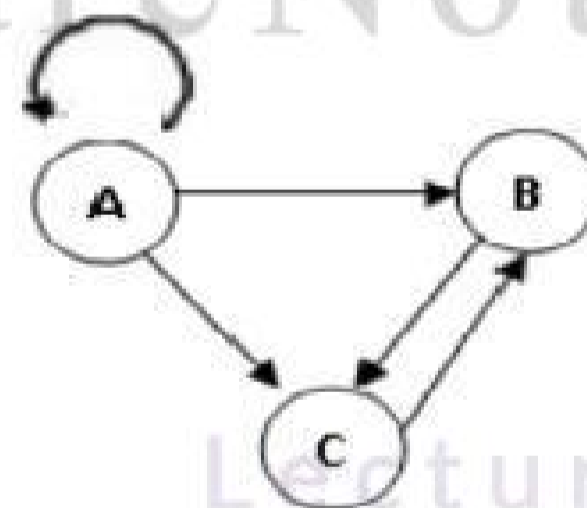
Adjacency nodes

- Two nodes or vertices are adjacent if they are connected to each other through an edge. In the example, B is adjacent to A, C is adjacent to A, and so on.



Loop:

- In a directed graph some edges may connect to itself then such type of edges are called as a loop. This is shown in below.



- In the above figure, (A, A) is a loop.

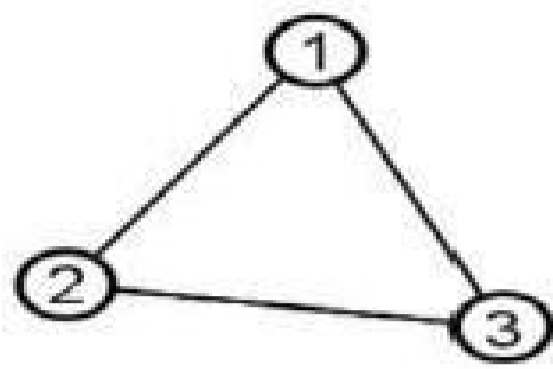
Undirected Graph and Directed Graph

- **Undirected Graph**

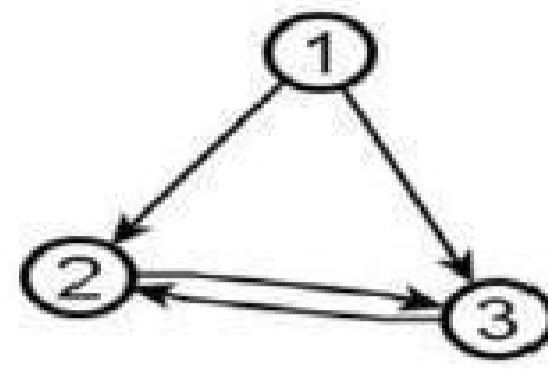
A Graph that does not contain any direction is called as undirected graph.

- **Directed Graph**

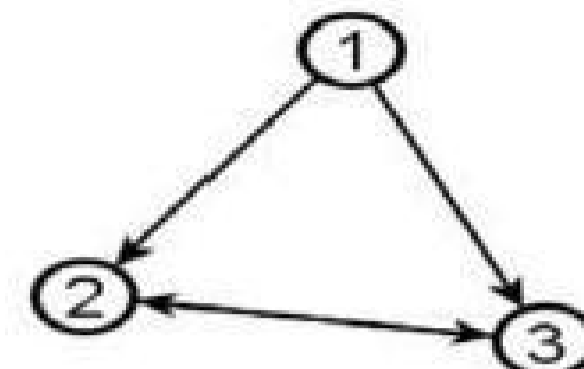
A pair of vertices between the edges must be ordered then that graph is known as directed graph.



Undirected graph (V_1, E_1)
 $V_1 = \{1, 2, 3\}$
 $E_1 = \{(1, 2), \{2, 3\}, \{3, 1\}\}$



Directed graph (V_2, E_2)
 $V_2 = \{1, 2, 3\}$
 $E_2 = \{(1, 2), (2, 3), (3, 2), (1, 3)\}$



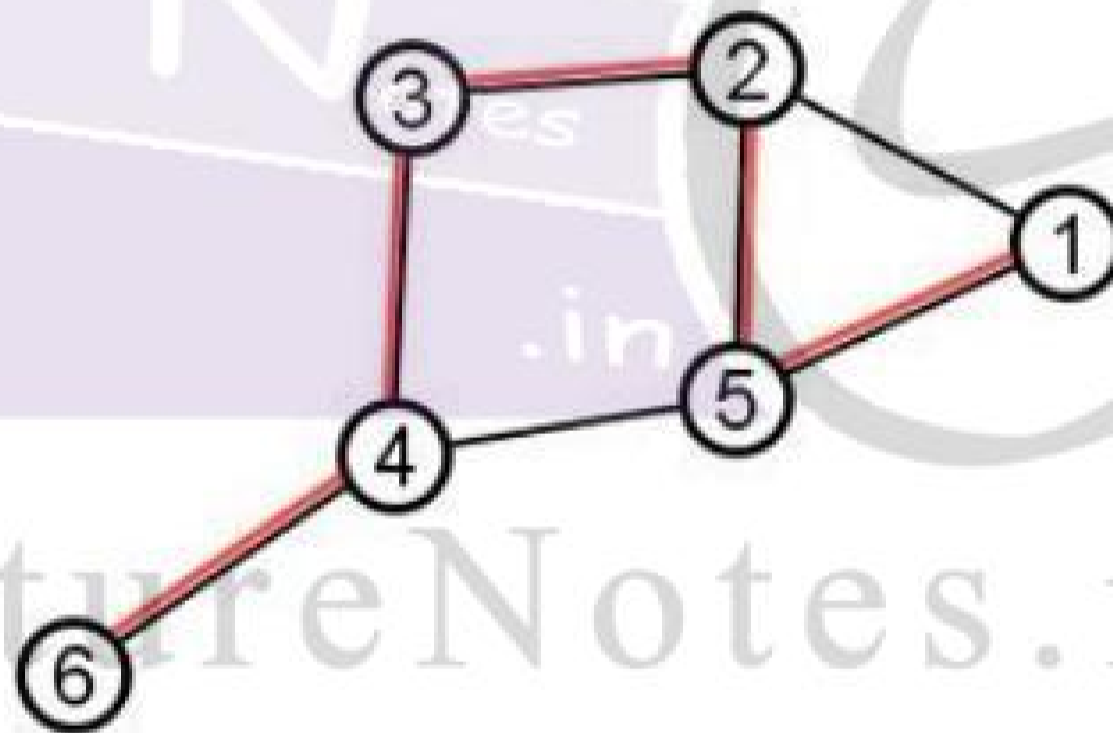
Easier way to draw directed graph (V_2, E_2)

Path

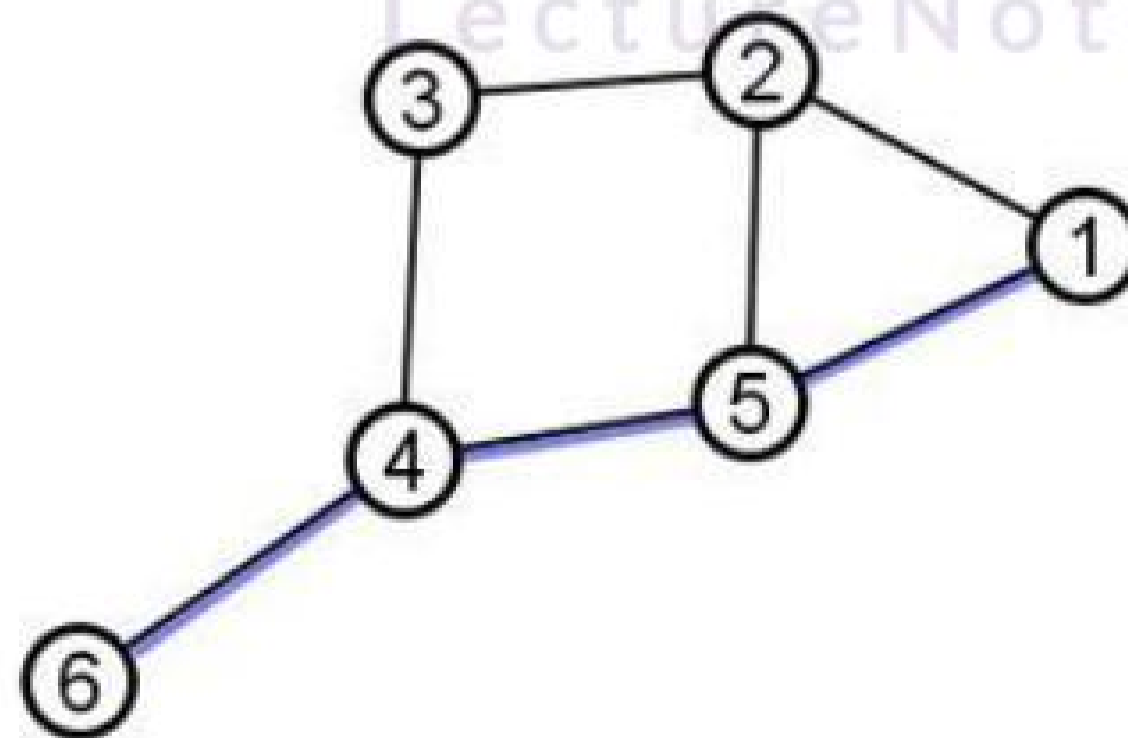
- A path in a graph represents a way to get from an origin to a destination by traversing edges in the graph. For example, in the undirected graph $G=(V,E)$ drawn below, there are many paths from node 6 to node 1.

Examples:

- The red path above is $((6,4), (4,3), (3,2), (2,5), (5,1))$; it is a path in G from node 6 to node 1

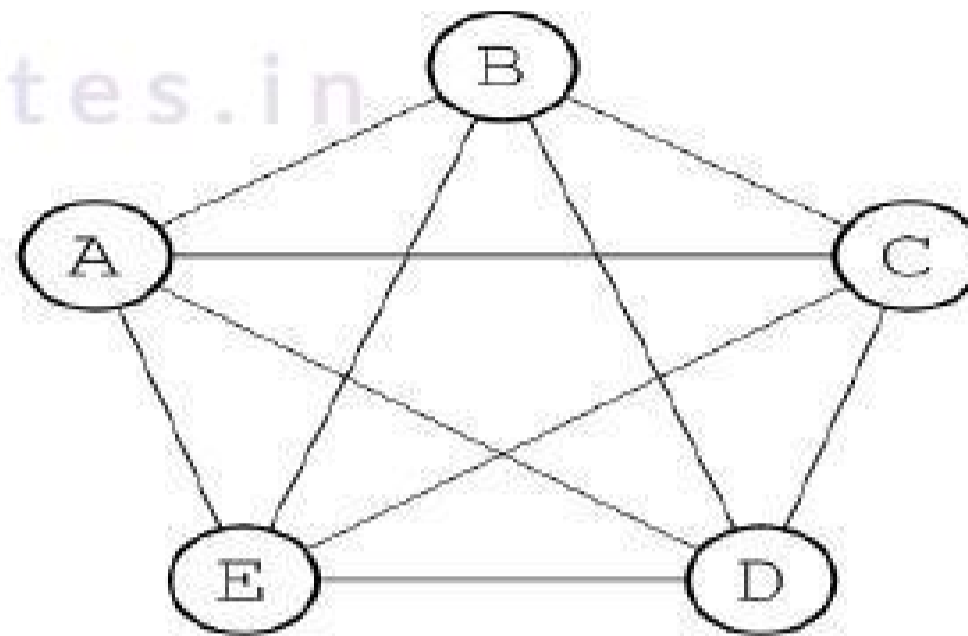


- The blue path below is $((6,4), (4,5), (5,1))$; it is also a path in G from node 6 to node 1



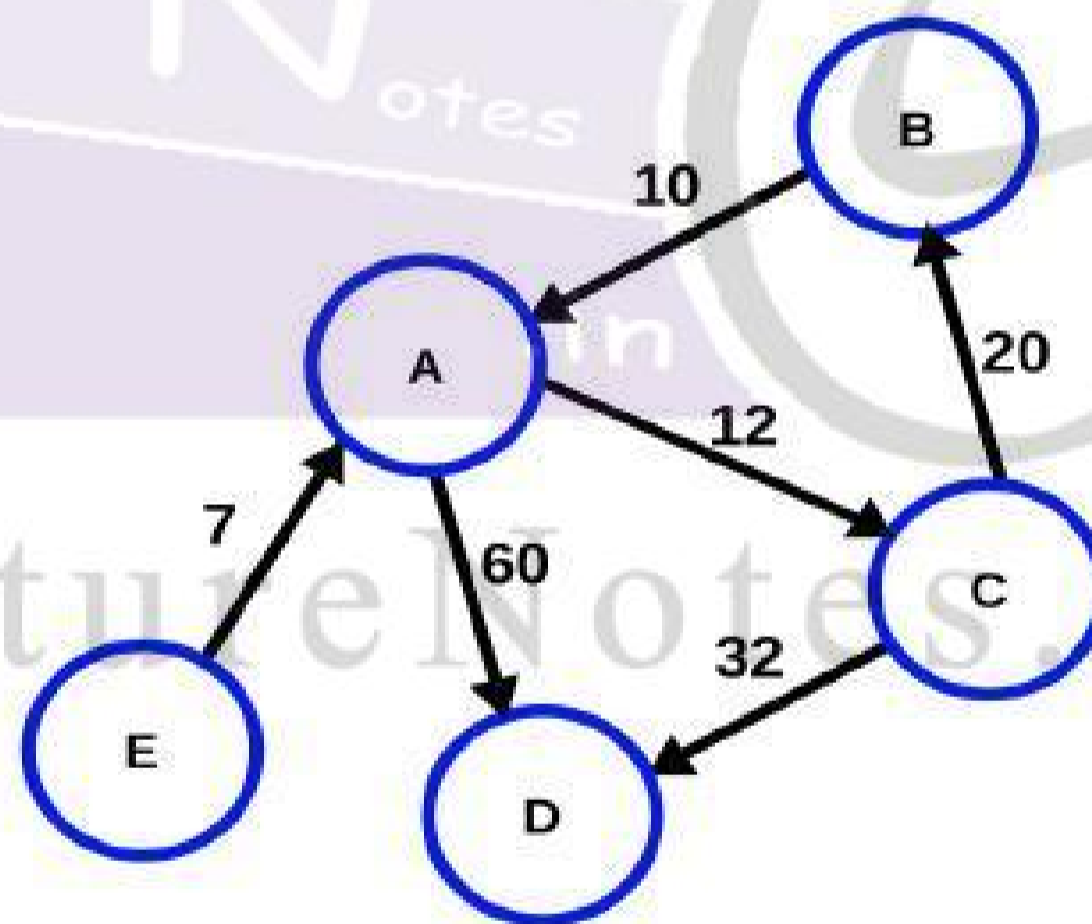
Complete graph

- A graph in which every vertex is directly connected to every other vertex.



Weighted Graph

- Some graphs contain weights on their edges, such type of graphs is called as weighted graph and the number associated with an edge is called as weight.



- The weight of an edge is often referred to as the "cost" of the edge.
- In applications, the weight may be a measure of the length of a route, the capacity of a line, the energy required to move between locations along a route, etc.

In-degree and Out-degree

In-degree of a node:

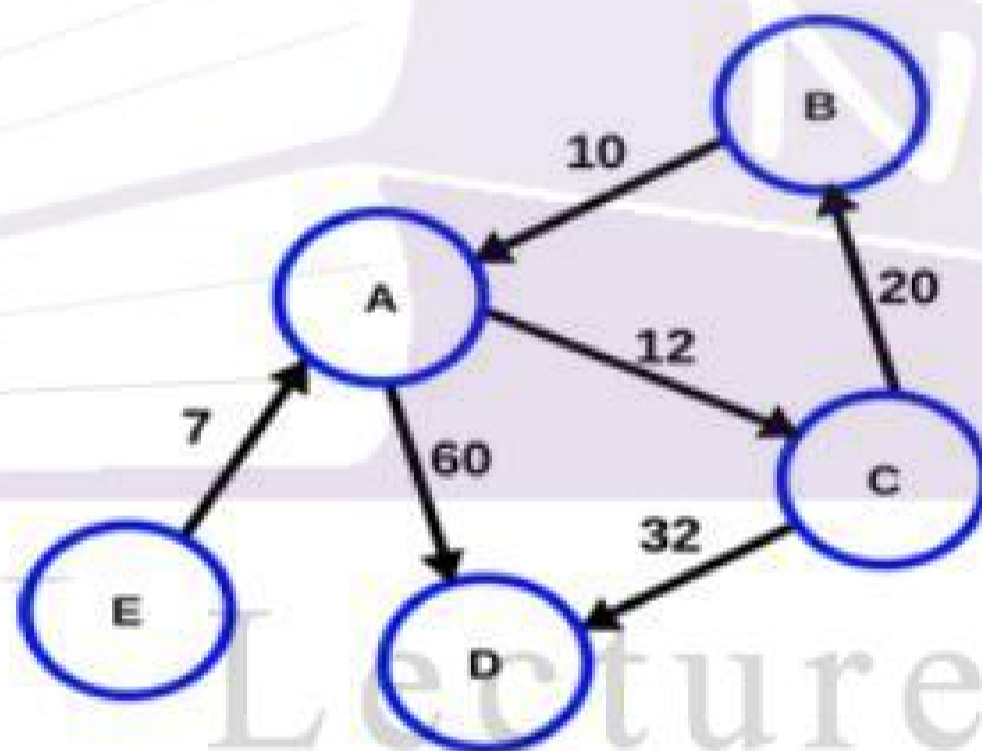
The In-degree of a node n is nothing but the number of edge coming to that node.

LectureNotes.in

Out-Degree of a node:

The out-degree of a node n is nothing but the number of edges that are move away from that node.

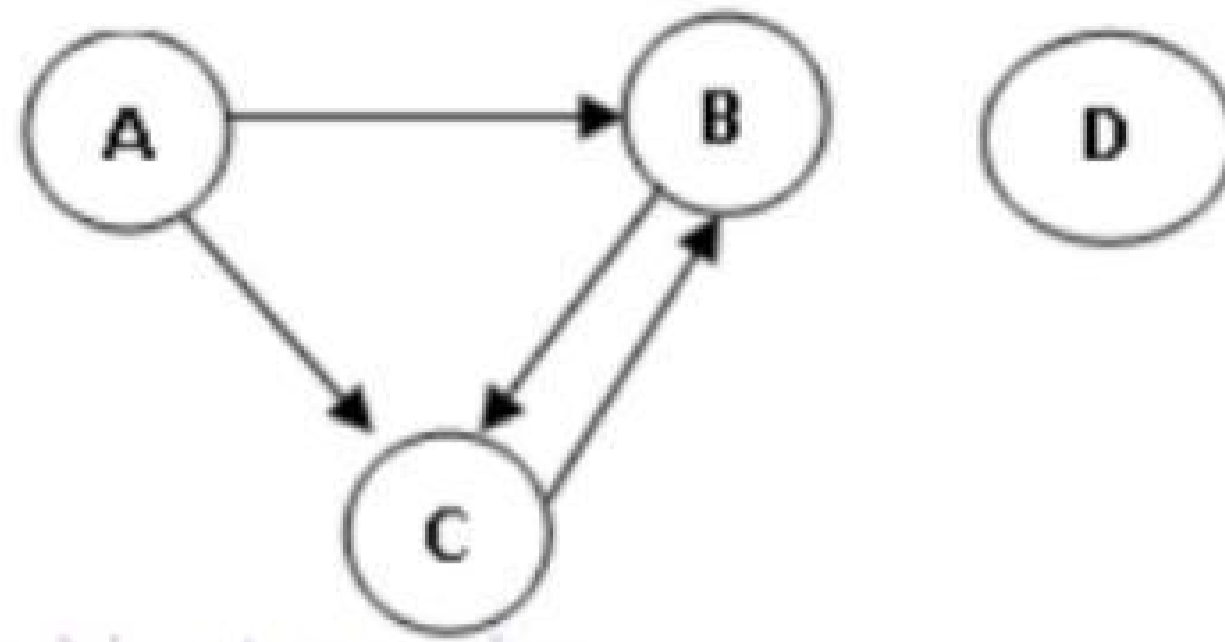
Example:



Vertex	In-Degree	Out-Degree
A	2	2
B	1	1
C	1	2
D	2	0
E	0	1

Isolated Node

- In a graph if a node that does not adjacent to any other node then that node is called as an isolated node.

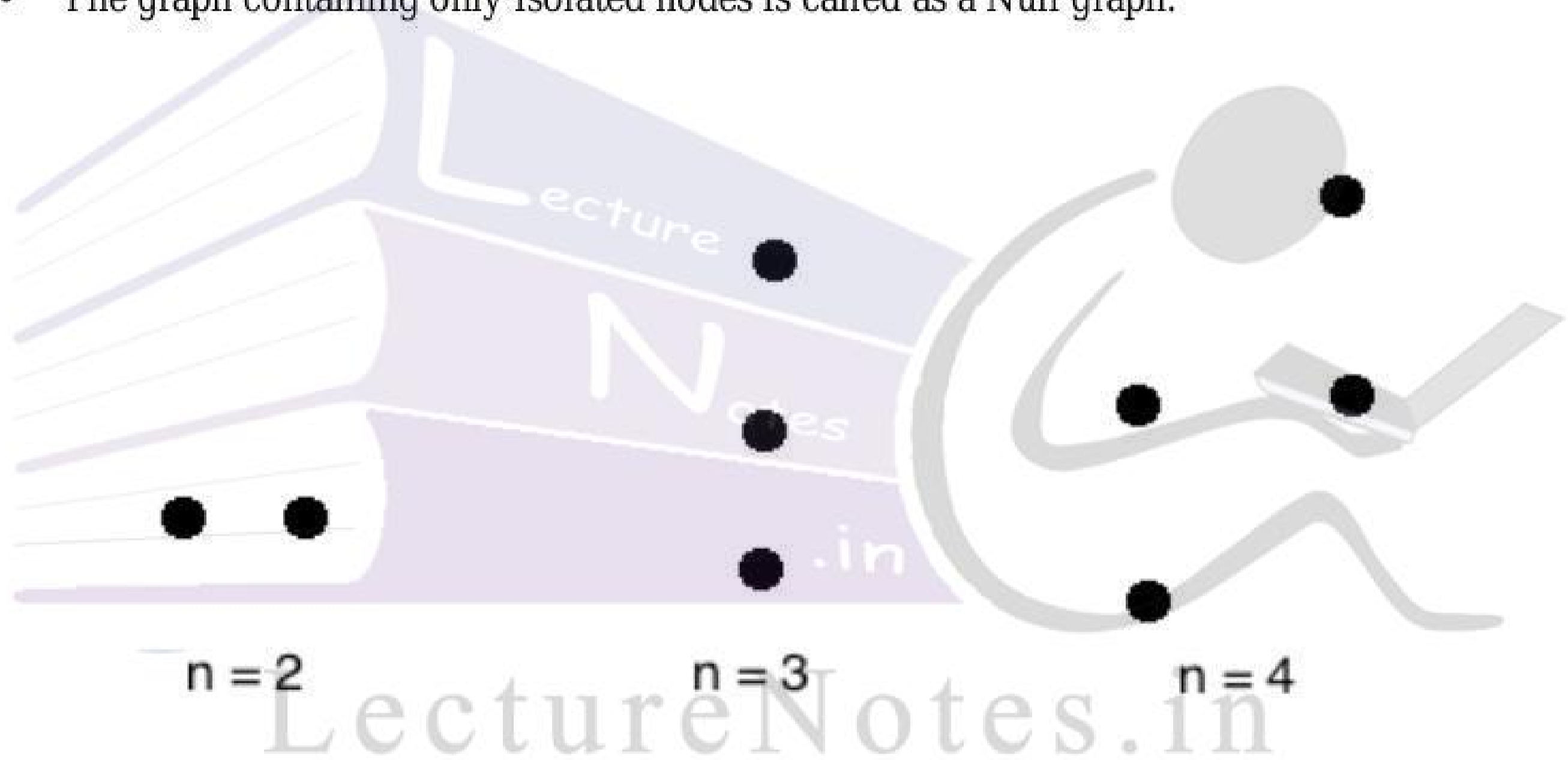


LectureNotes.in

- In the above figure, the node D is an Isolated node.

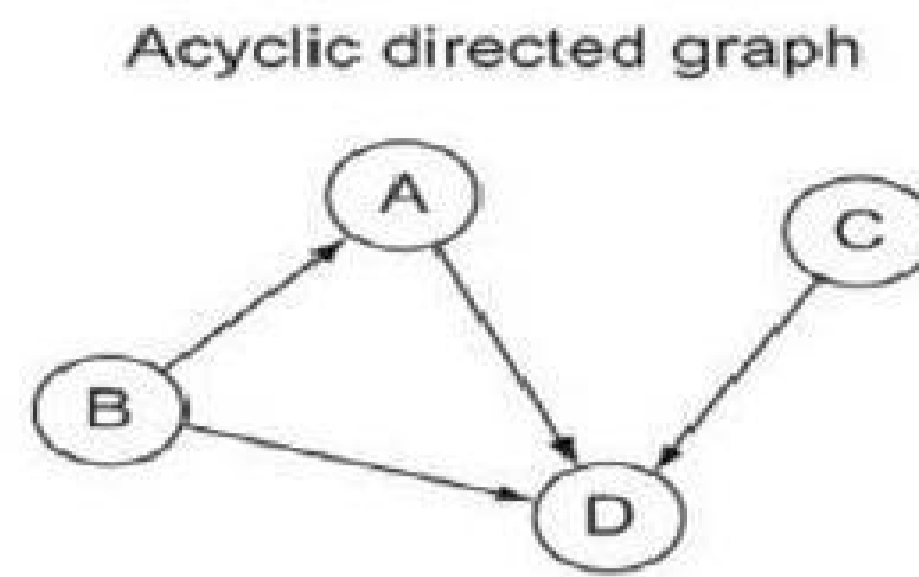
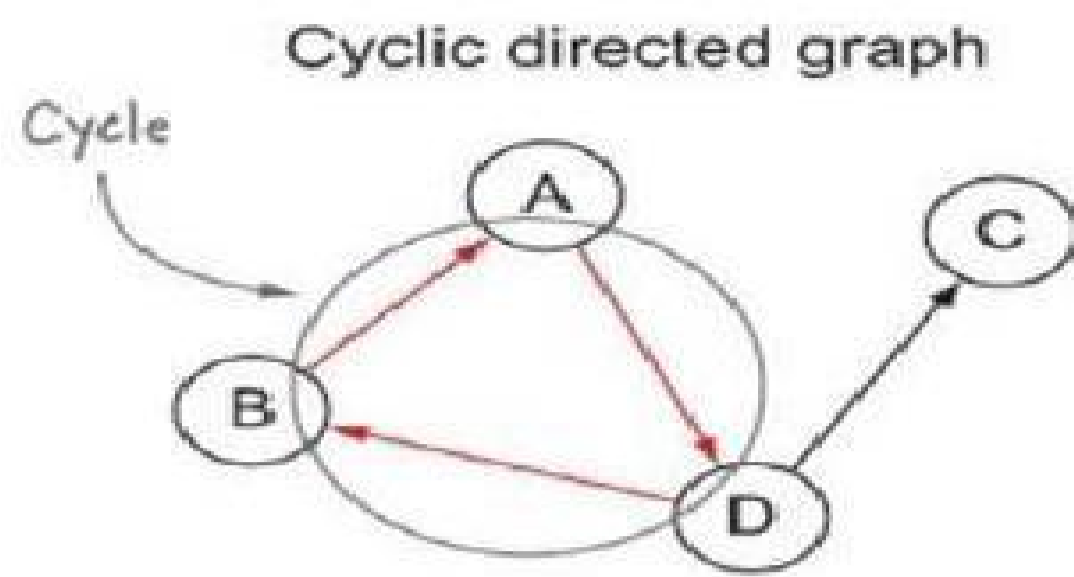
Null Graph

- The graph containing only isolated nodes is called as a Null graph.



Cyclic Graph and Acyclic Graph

- A path from a node to itself is called as a cyclic graph. In other words, a graph containing a cycle in it is called as a cyclic graph.
- The graph does not contain the cycle is called as an acyclic graph and the directed graph that does not contain any cycle is called as directed acyclic graph.



LectureNotes.in

Graph Representations

Graph data structure is represented using following representations...

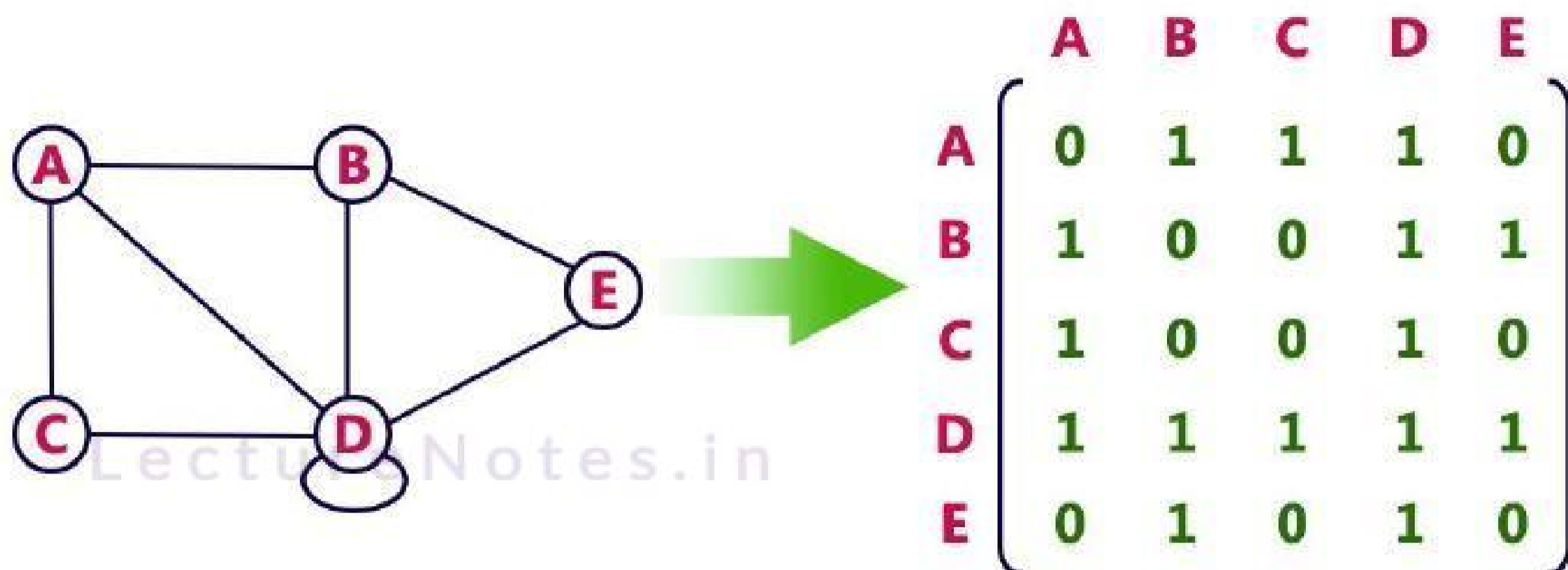
1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

Adjacency Matrix

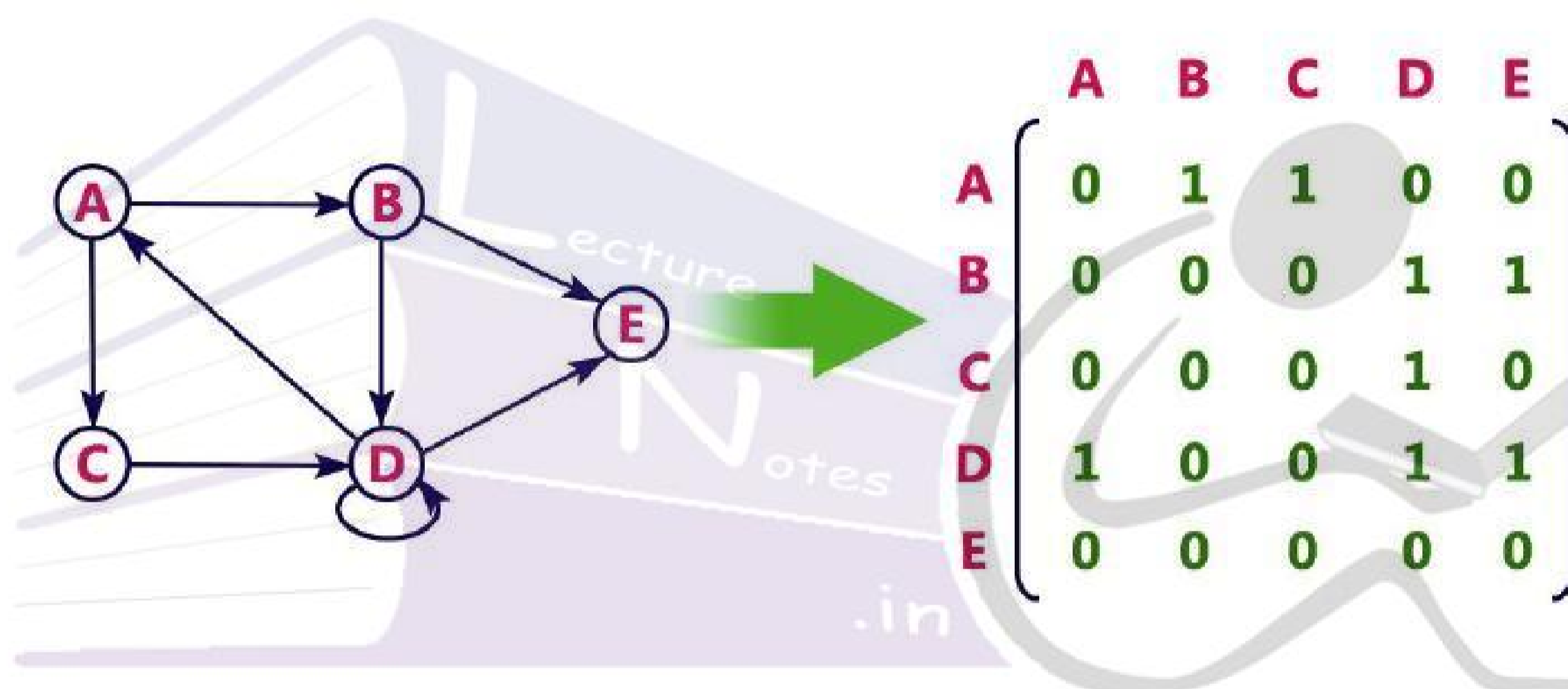
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...

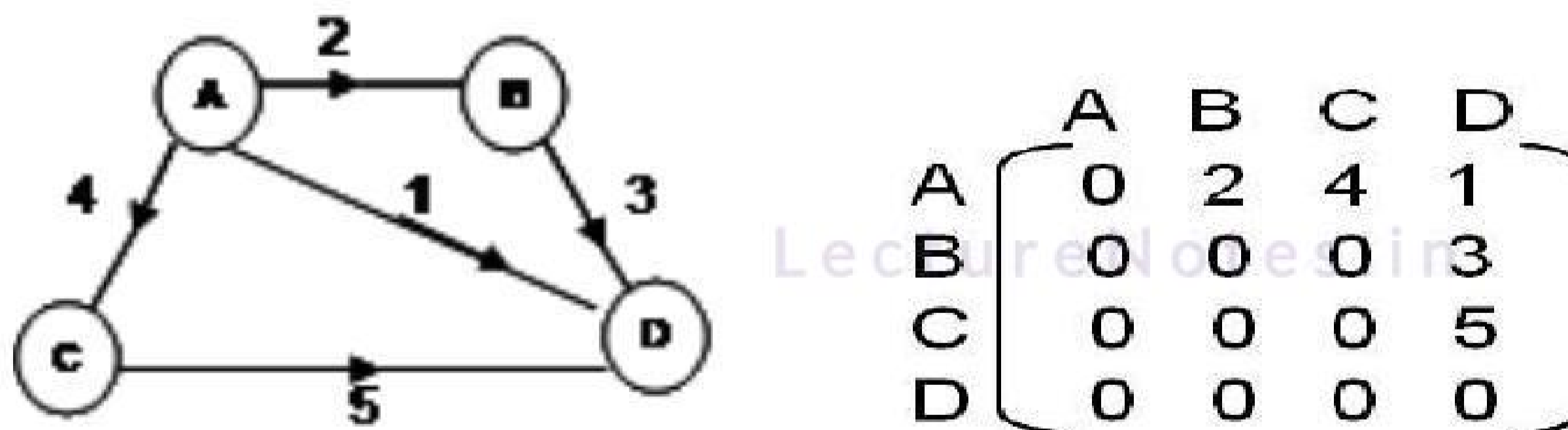
LectureNotes.in



Directed graph representation...



- A **weighted graph** may be represented using the weight as the entry.

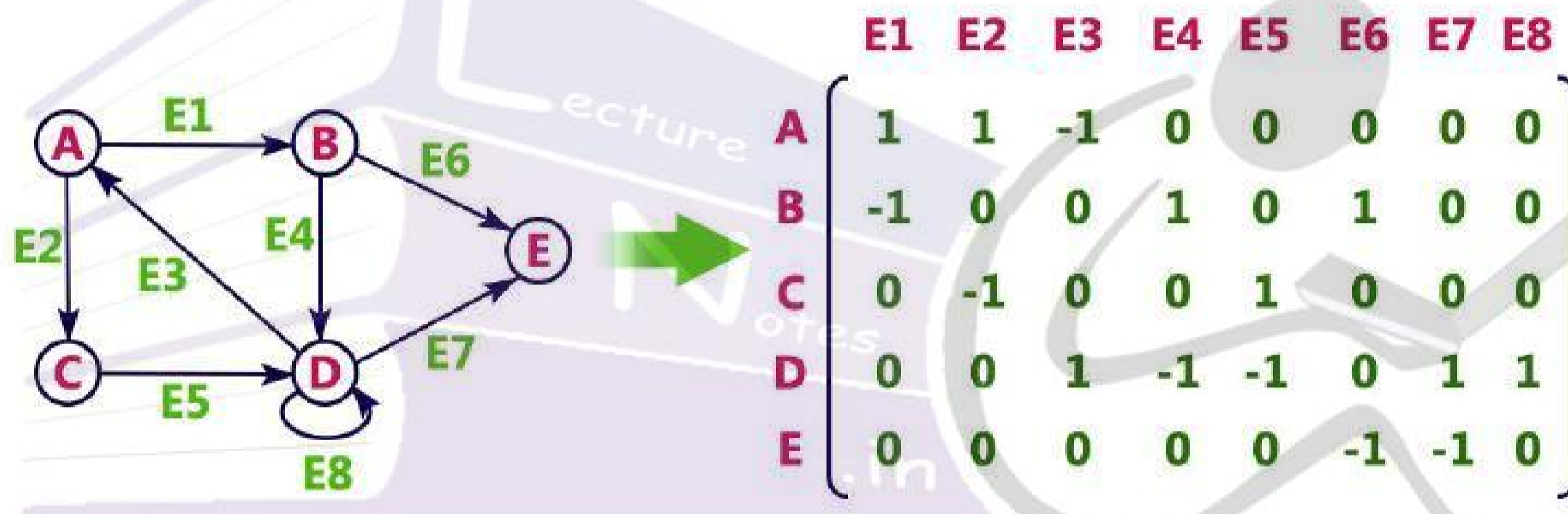


Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges.

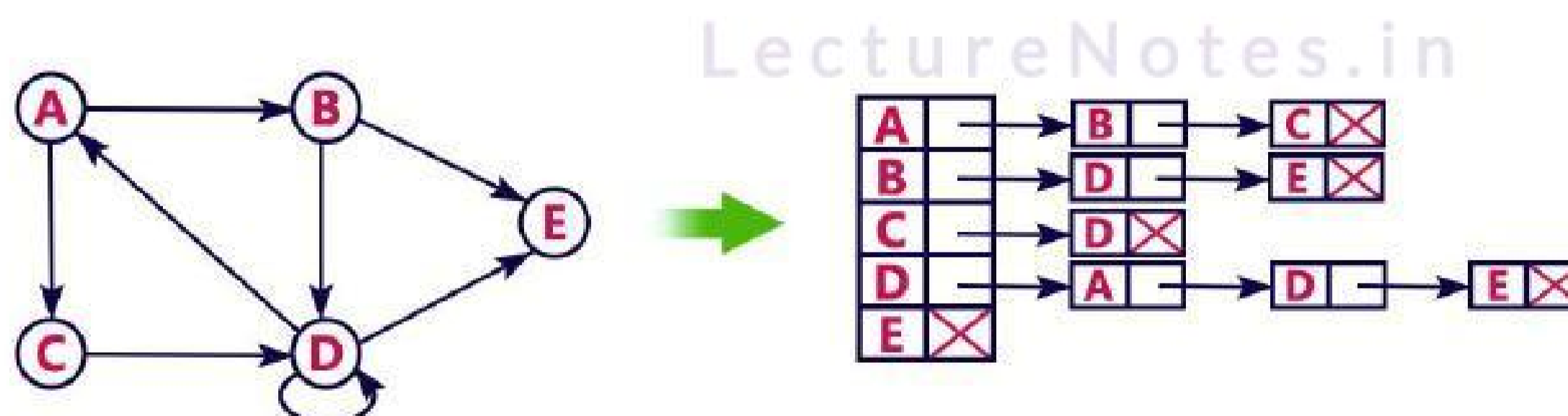
This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

For example, consider the following directed graph representation...

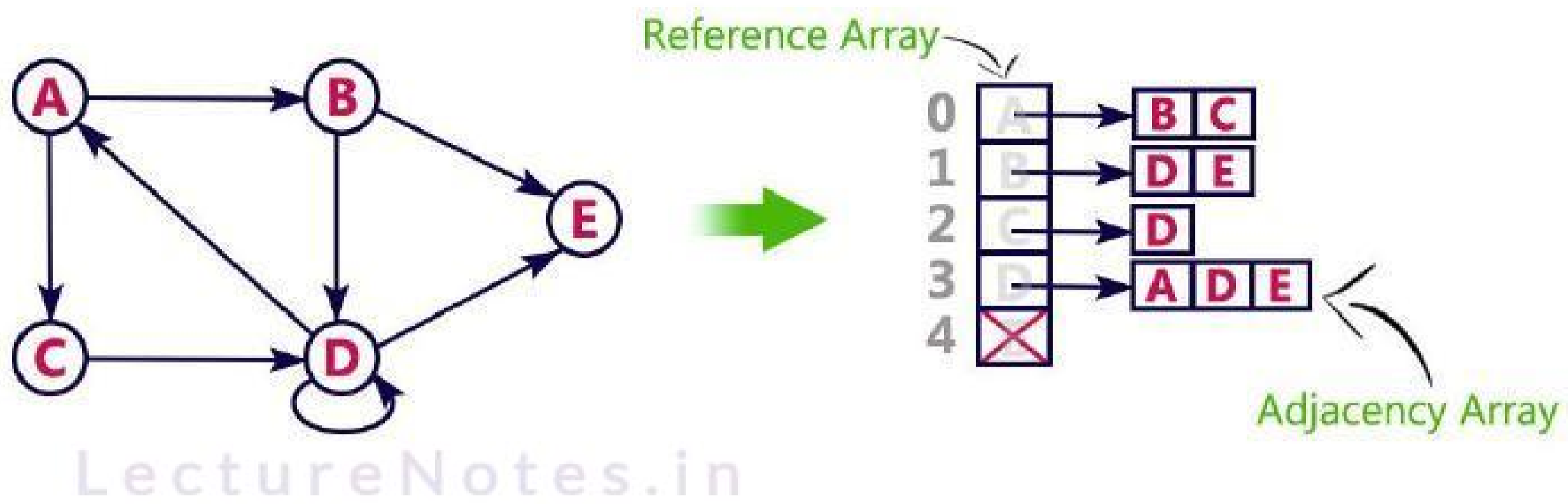


Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. For example, consider the following directed graph representation implemented using linked list...

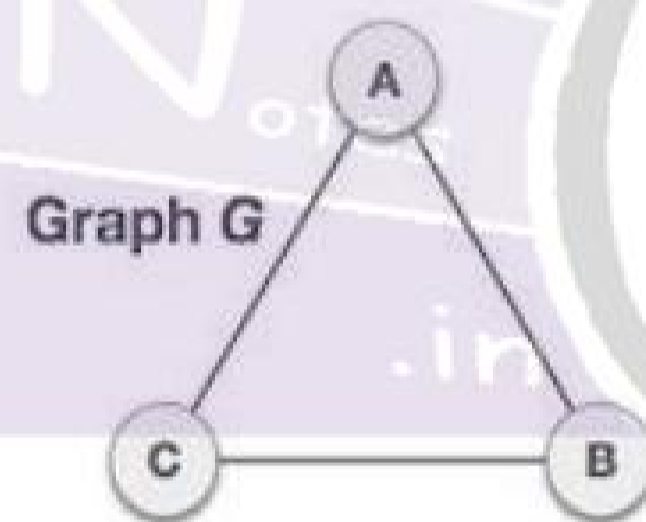


This representation can also be implemented using array as follows..



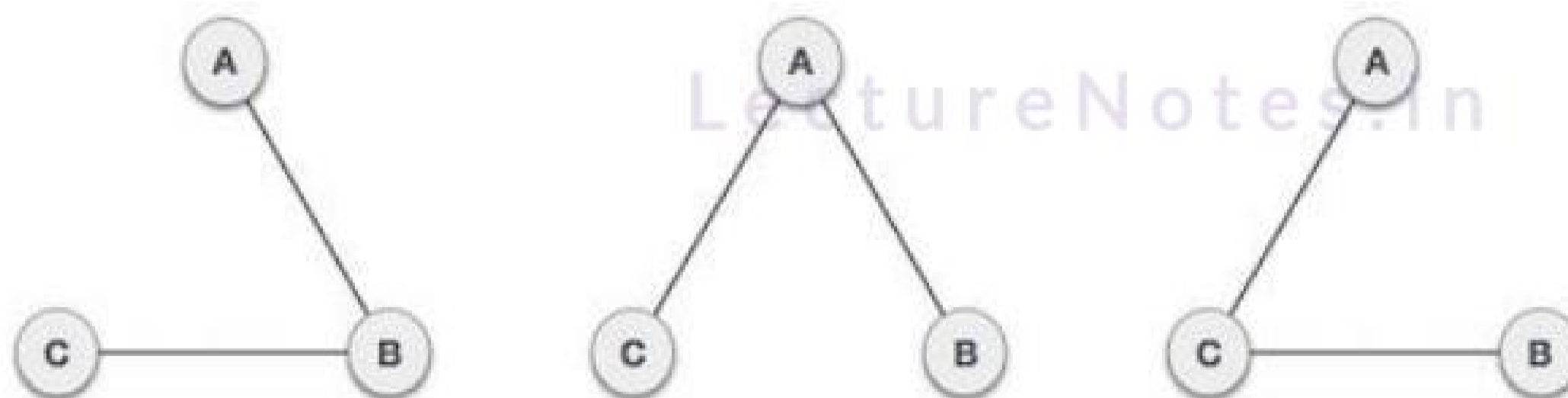
Spanning Tree

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.



LectureNotes.in

Spanning Trees



Graph Traversals

Graph traversal is technique used for searching a vertex in a graph. The graph traversal is also used to decide the order of vertices to be visit in the search process. A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. **DFS (Depth First Search)**
2. **BFS (Breadth First Search)**

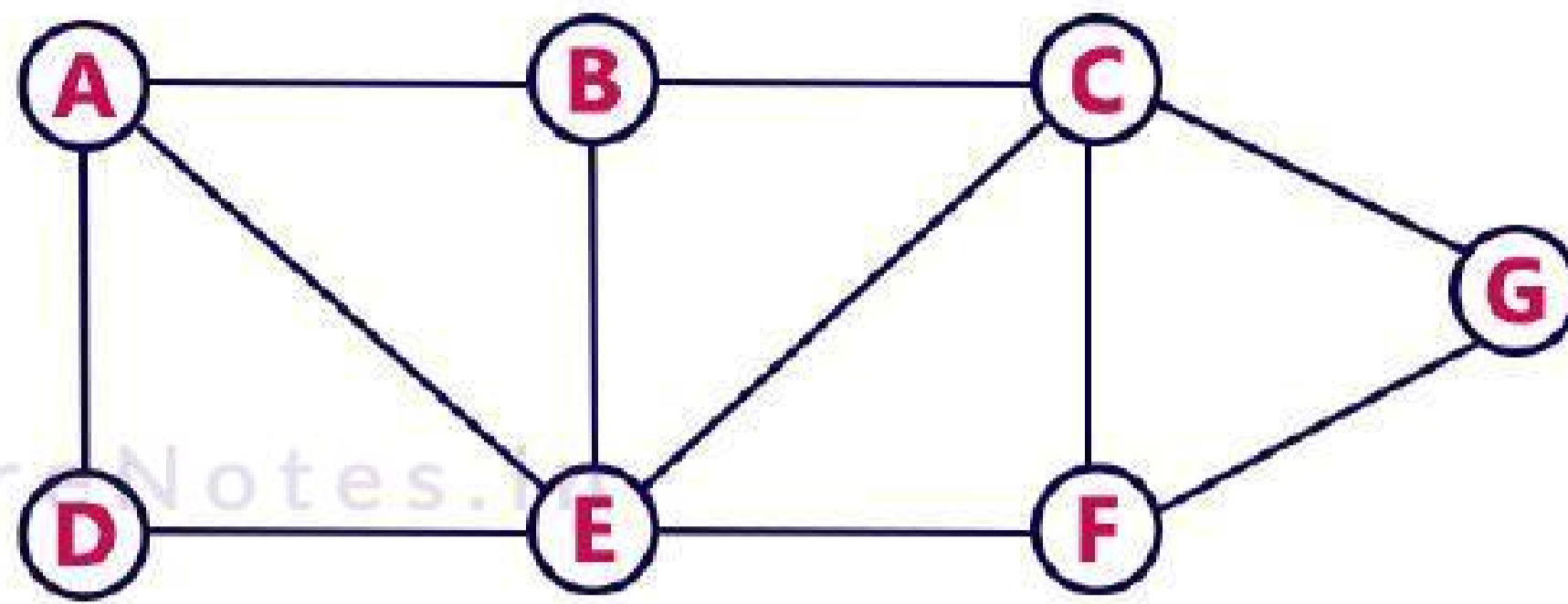
DFS (Depth First Search)

DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of a graph. We use the following steps to implement DFS traversal...

- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

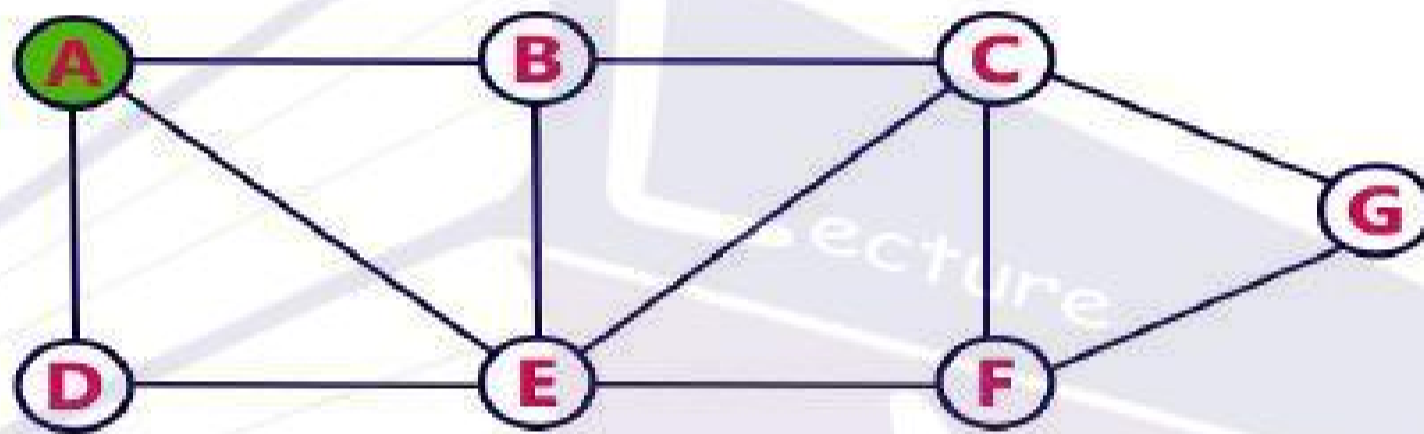
Back tracking is coming back to the vertex from which we came to current vertex.

Consider the following example graph to perform DFS traversal



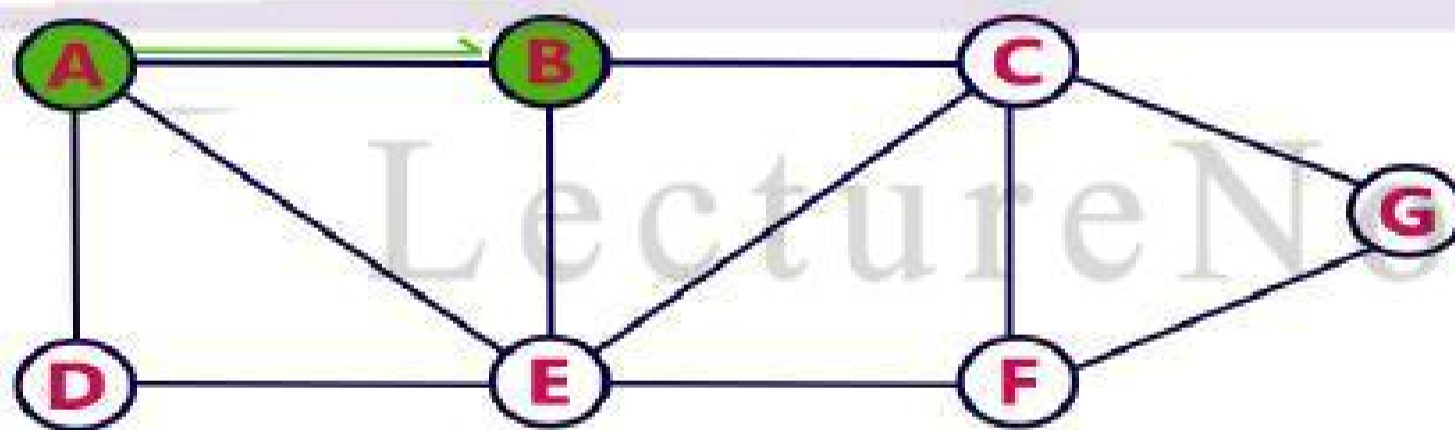
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



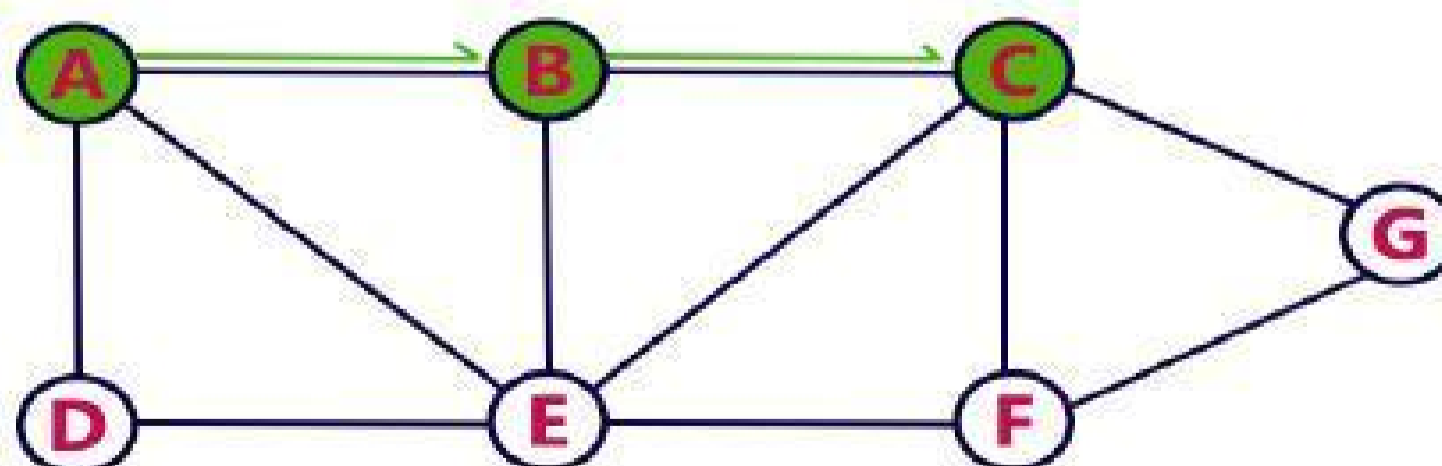
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex **B** on to the Stack.



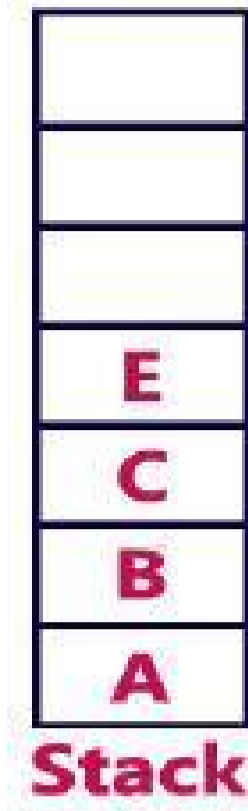
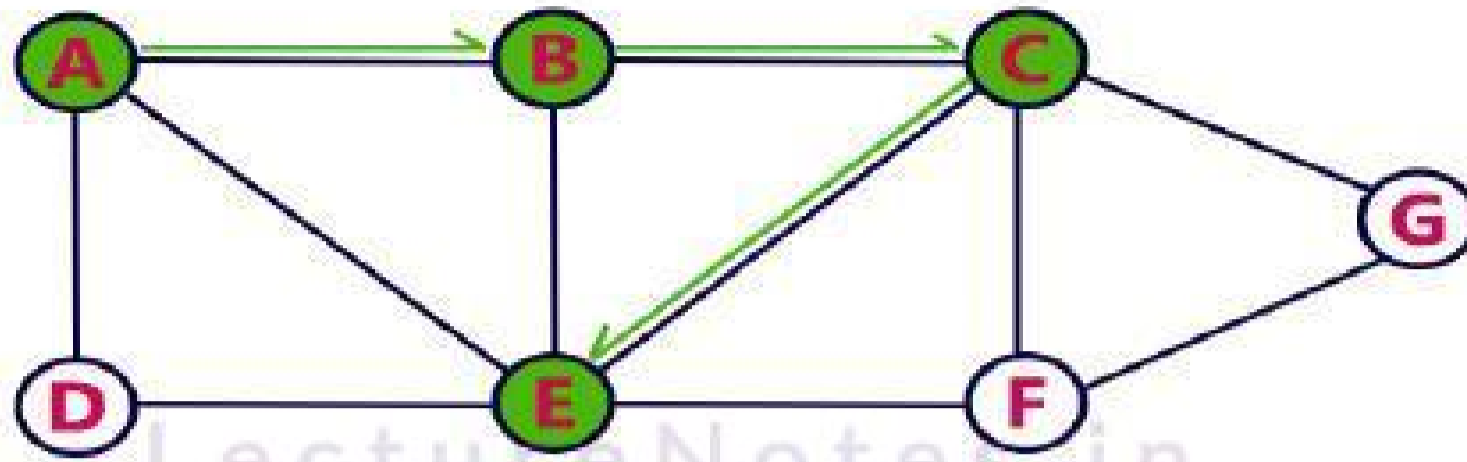
Step 3:

- Visit any adjacent vertex of **B** which is not visited (**C**).
- Push **C** on to the Stack.



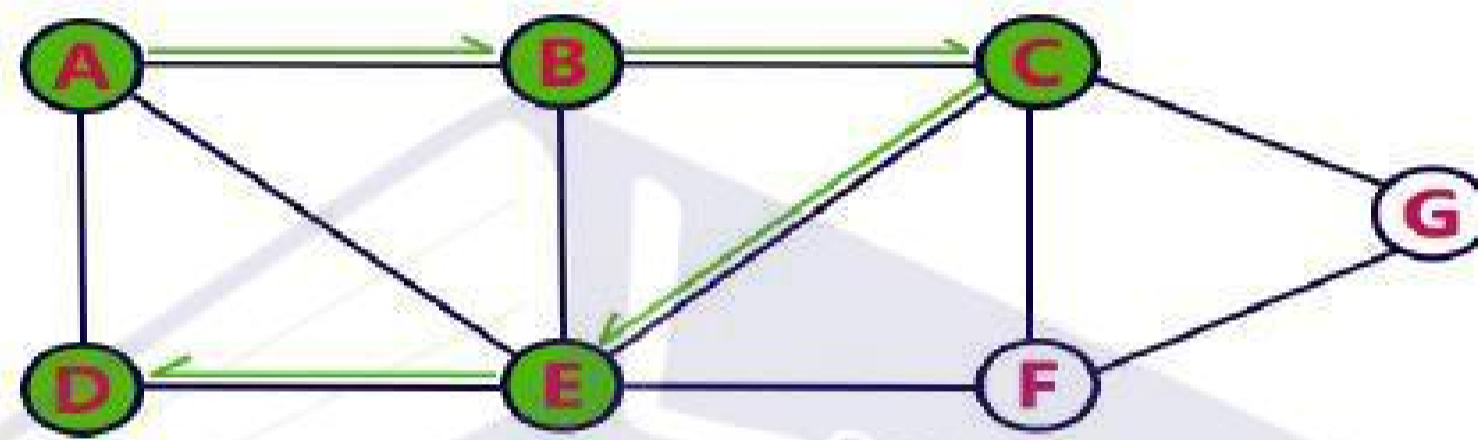
Step 4:

- Visit any adjacent vertex of **C** which is not visited (**E**).
- Push E on to the Stack



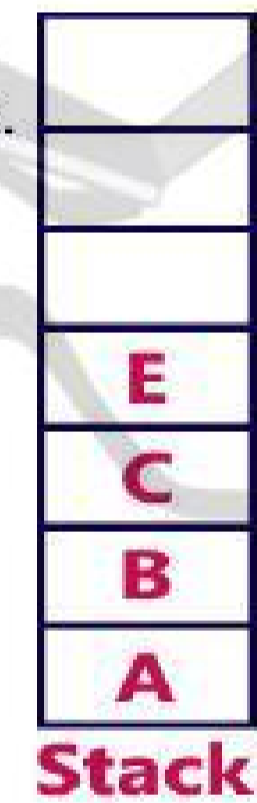
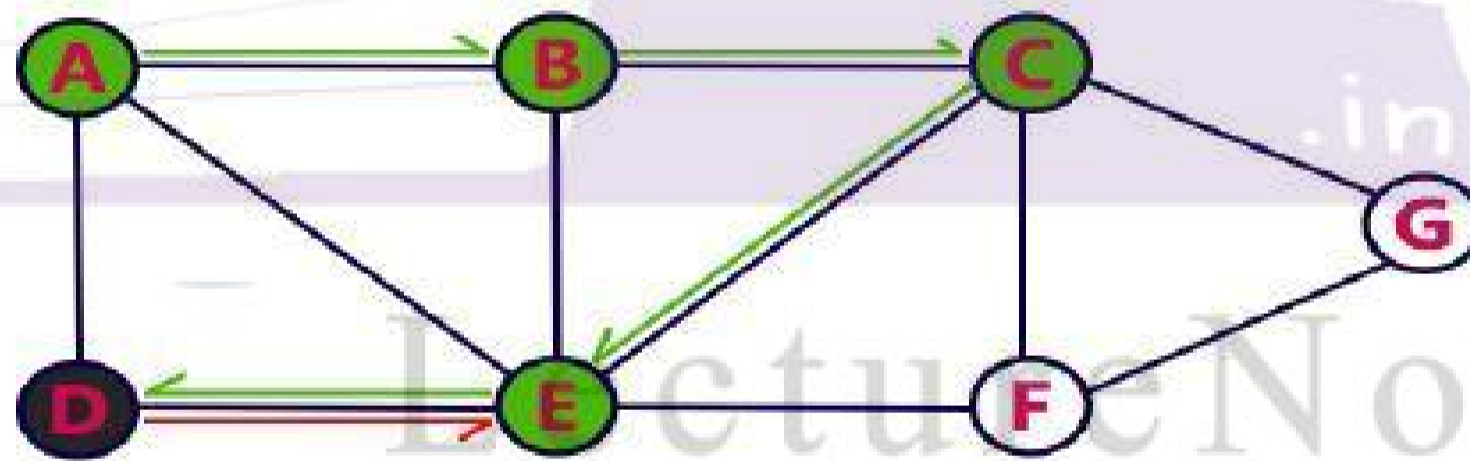
Step 5:

- Visit any adjacent vertex of **E** which is not visited (**D**).
- Push D on to the Stack



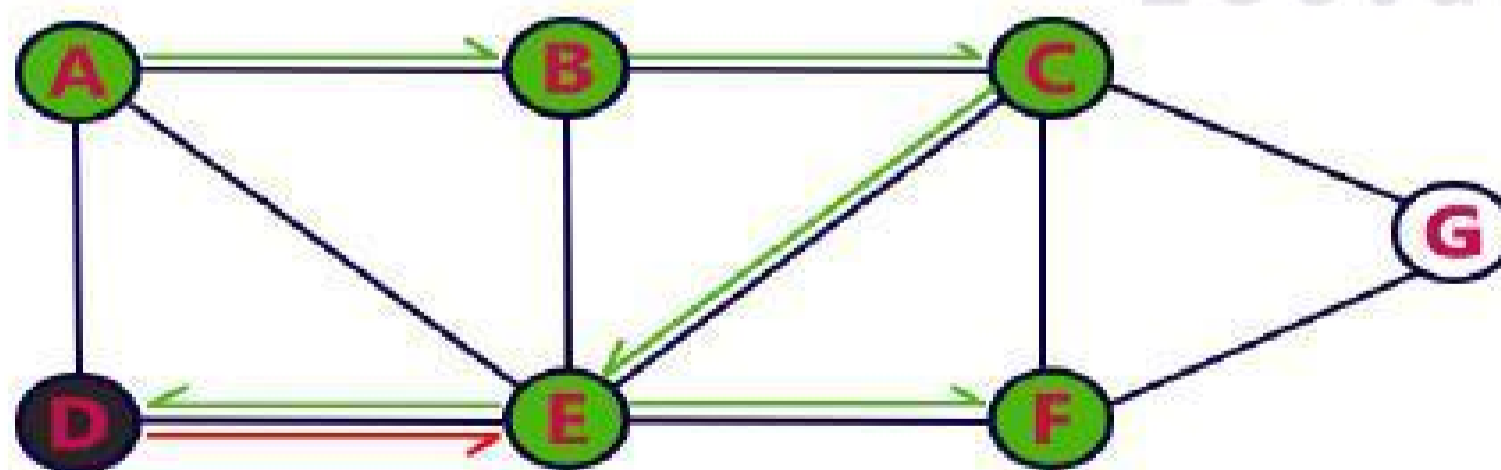
Step 6:

- There is no new vertex to be visited from D. So use back track.
- Pop D from the Stack.



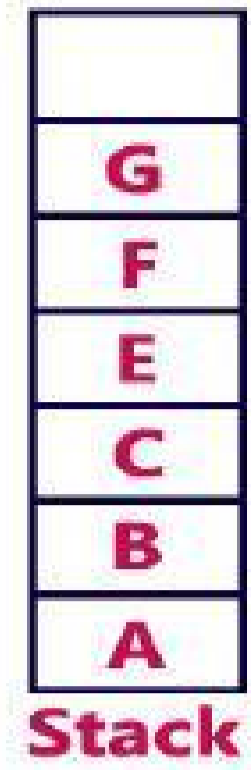
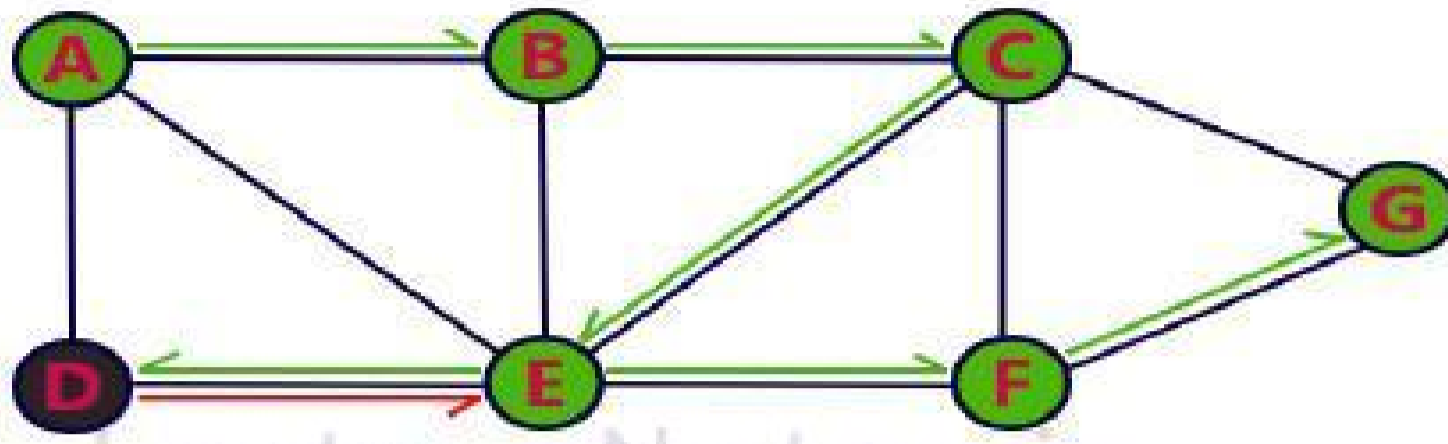
Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



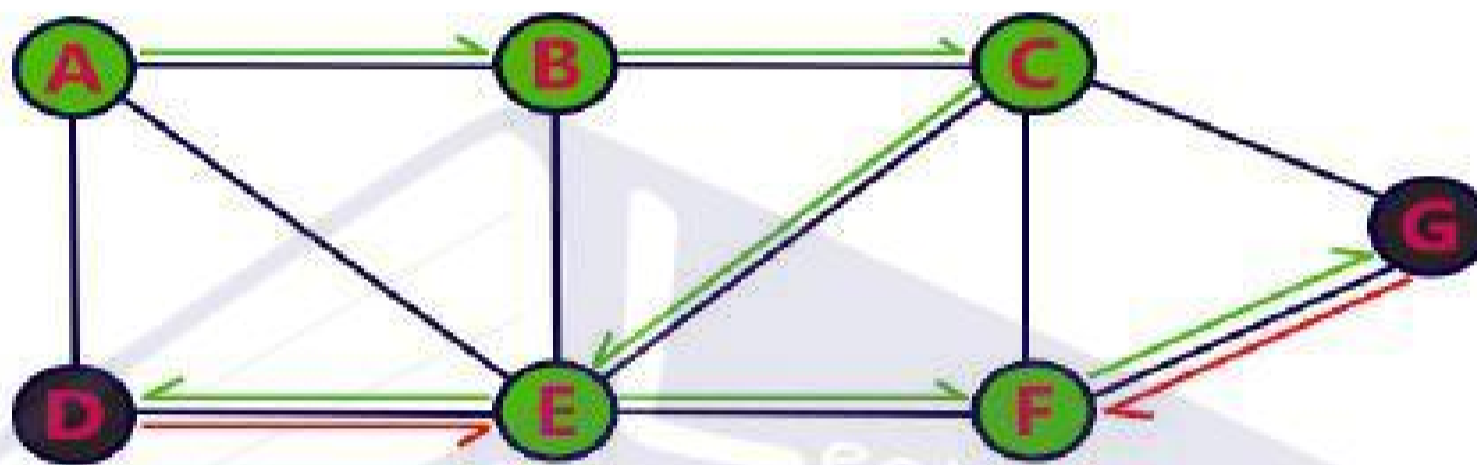
Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.



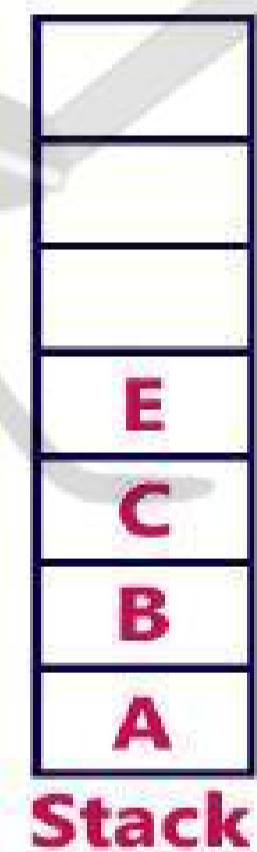
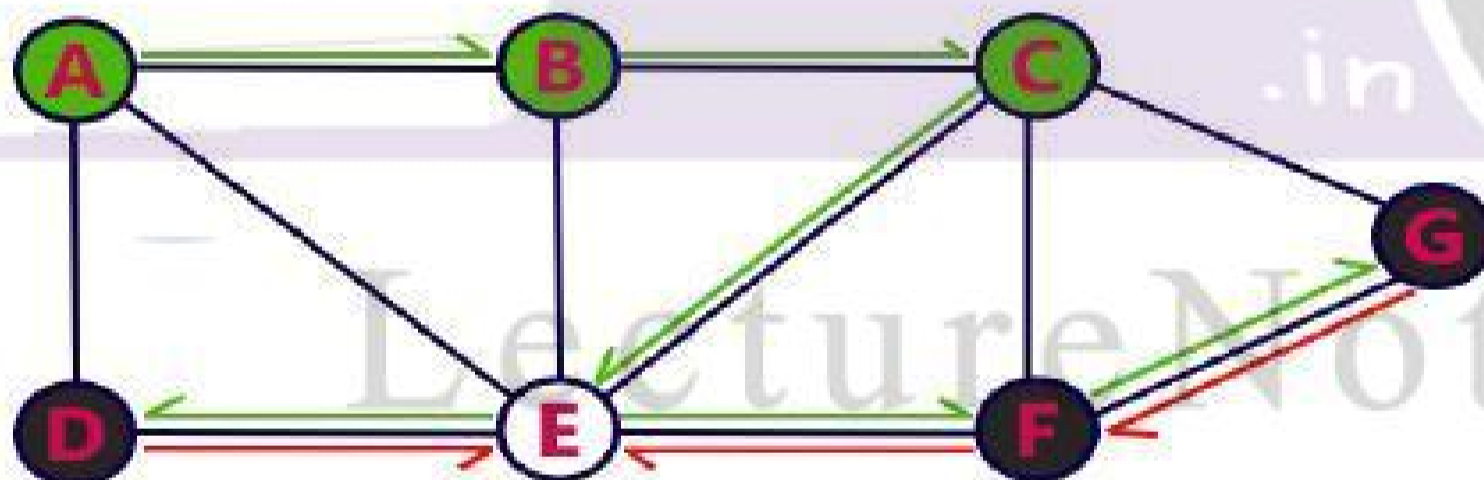
Step 9:

- There is no new vertex to be visited from **G**. So use back track.
- Pop **G** from the Stack.



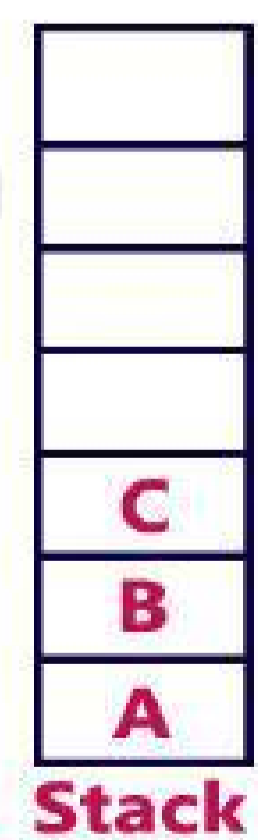
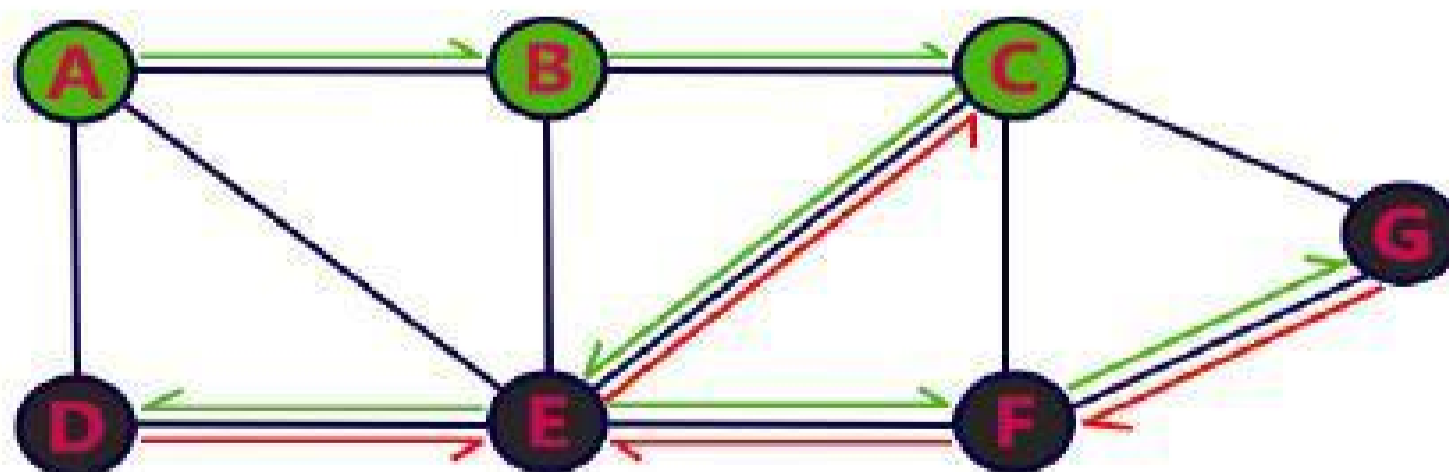
Step 10:

- There is no new vertex to be visited from **F**. So use back track.
- Pop **F** from the Stack.



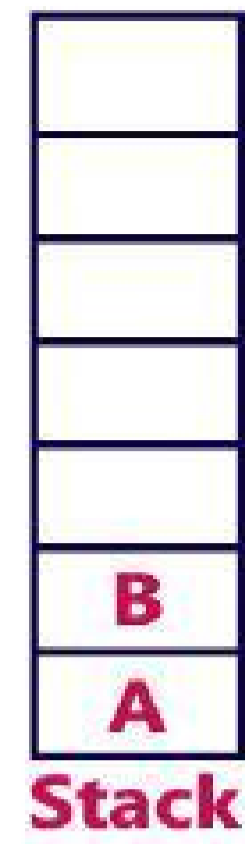
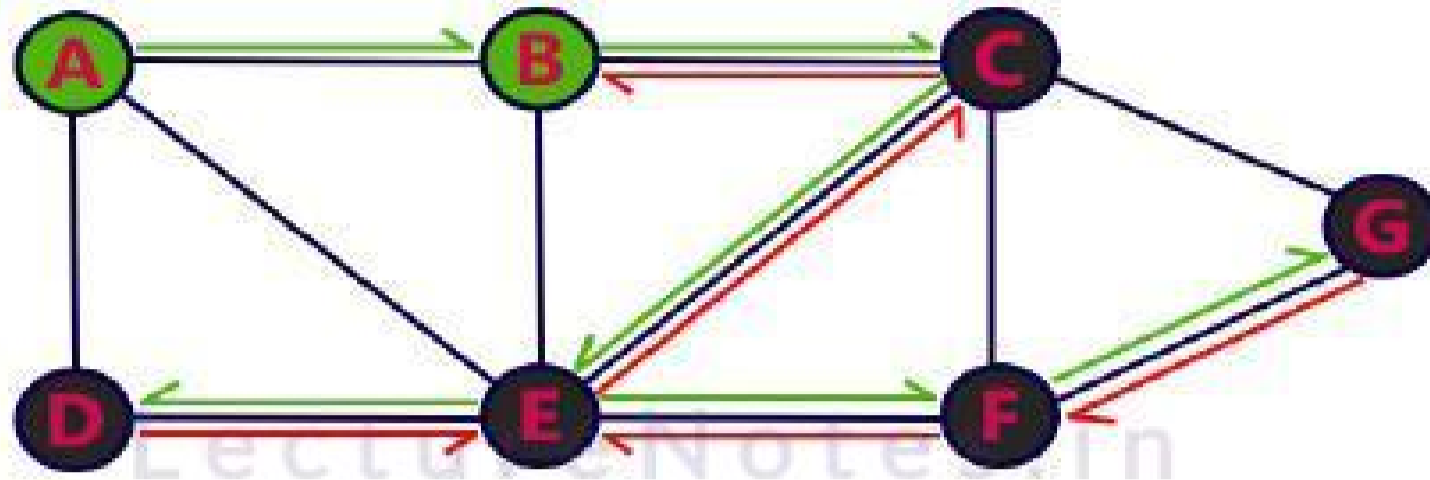
Step 11:

- There is no new vertex to be visited from **E**. So use back track.
- Pop **E** from the Stack.



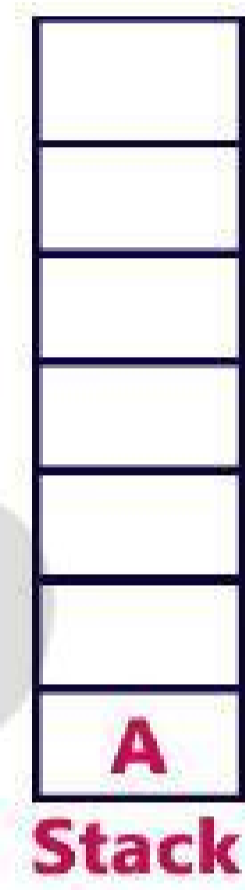
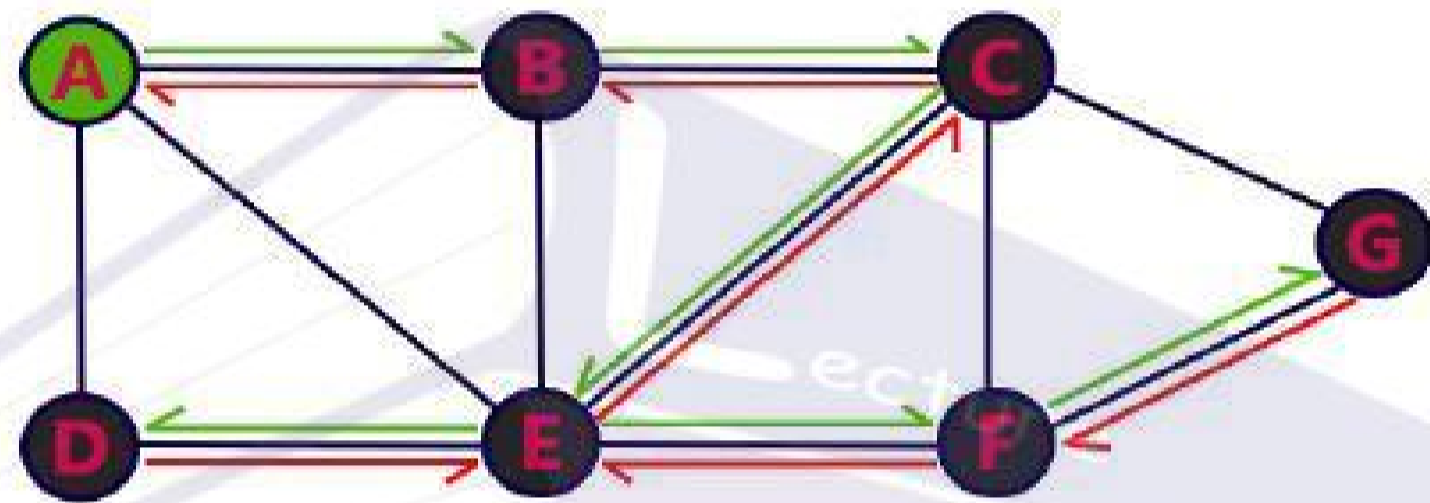
Step 12:

- There is no new vertex to be visited from C. So use back track.
- Pop C from the Stack.



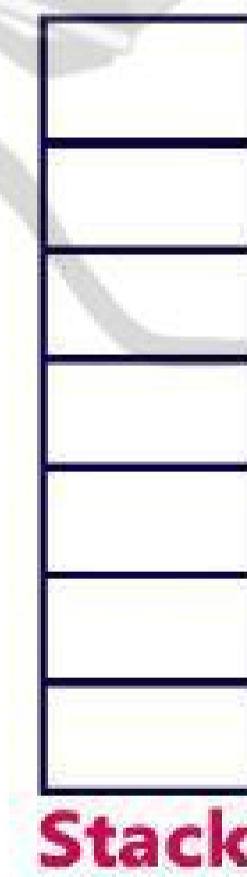
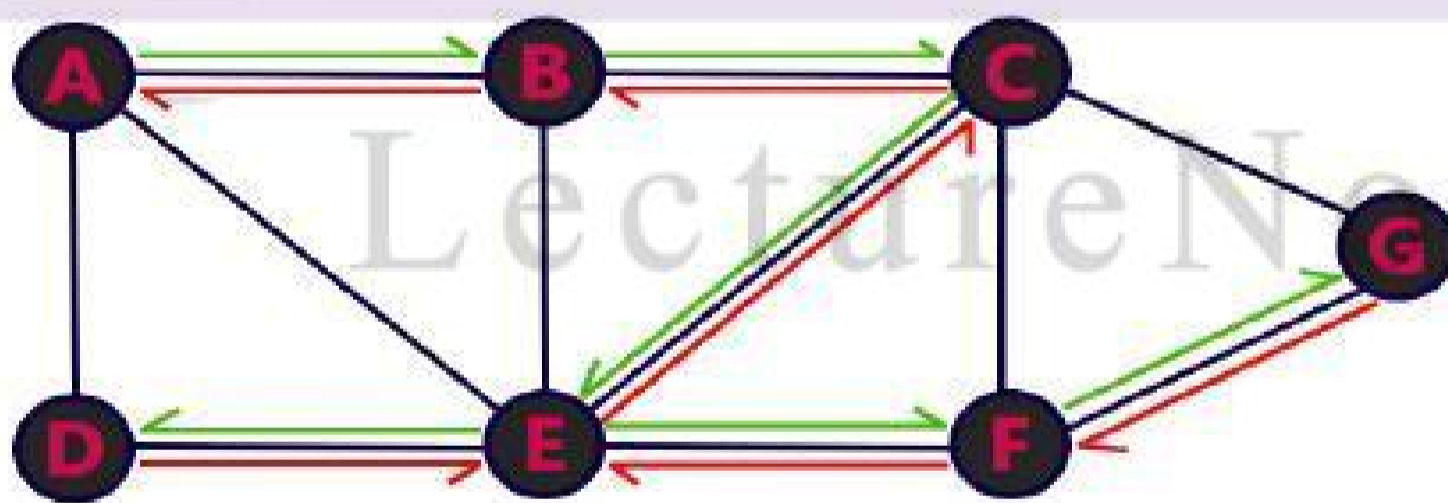
Step 13:

- There is no new vertex to be visited from B. So use back track.
- Pop B from the Stack.

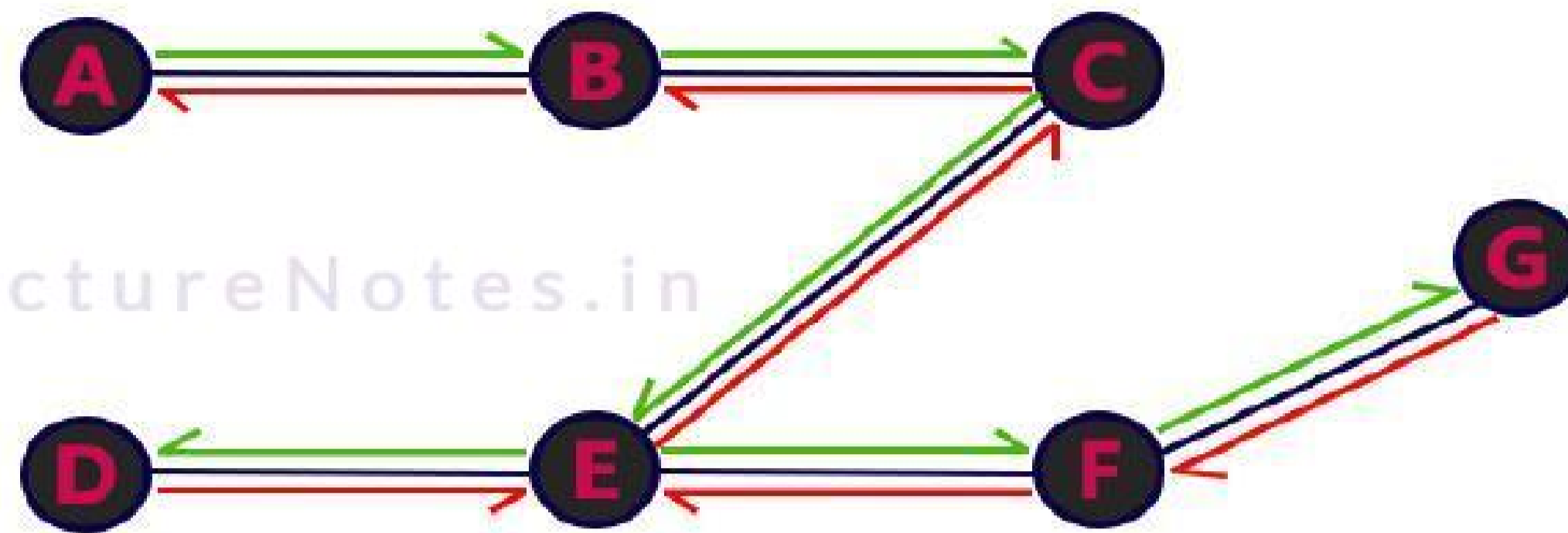


Step 14:

- There is no new vertex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Traversal.
- Final result of DFS traversal is following spanning tree.



BFS (Breadth First Search)

BFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

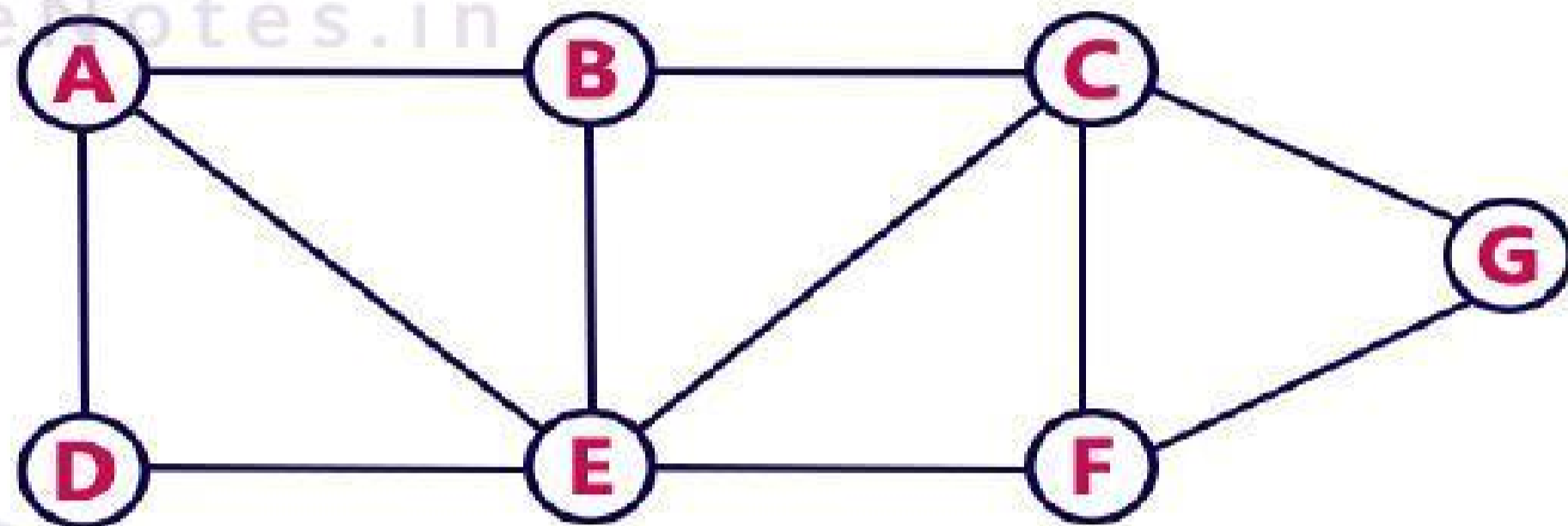
We use the following steps to implement BFS traversal...

- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the **adjacent** vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.

- **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

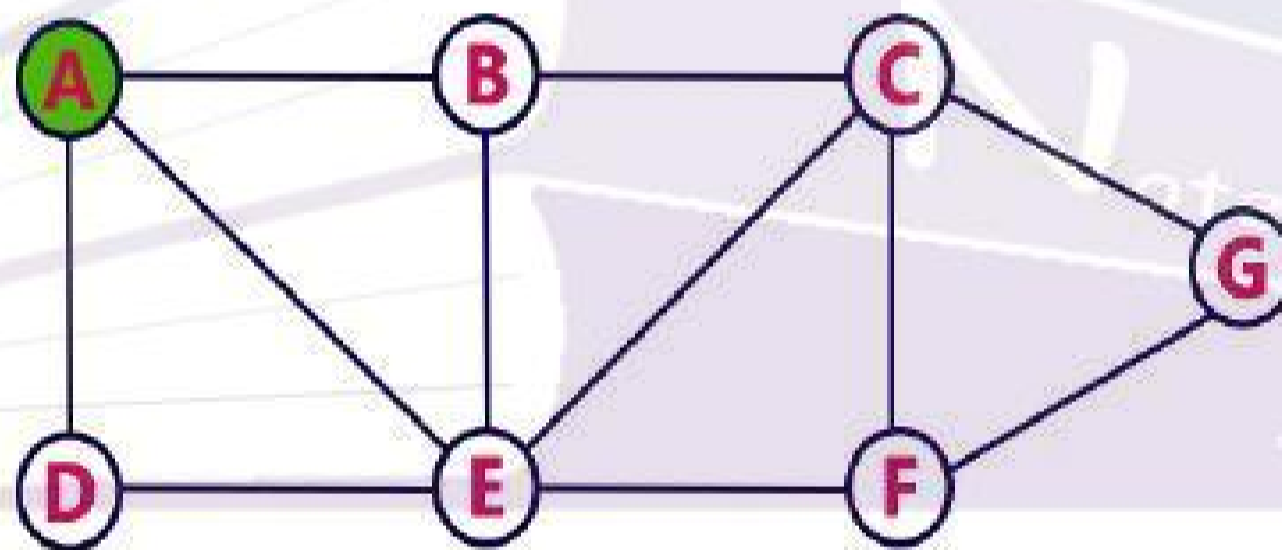
Example

Consider the following example graph to perform BFS traversal

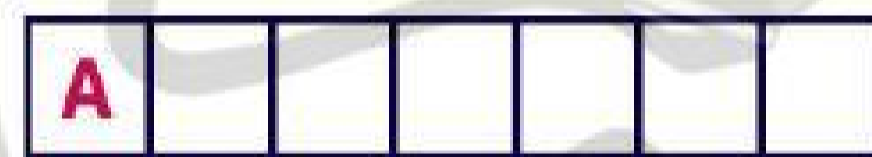


Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

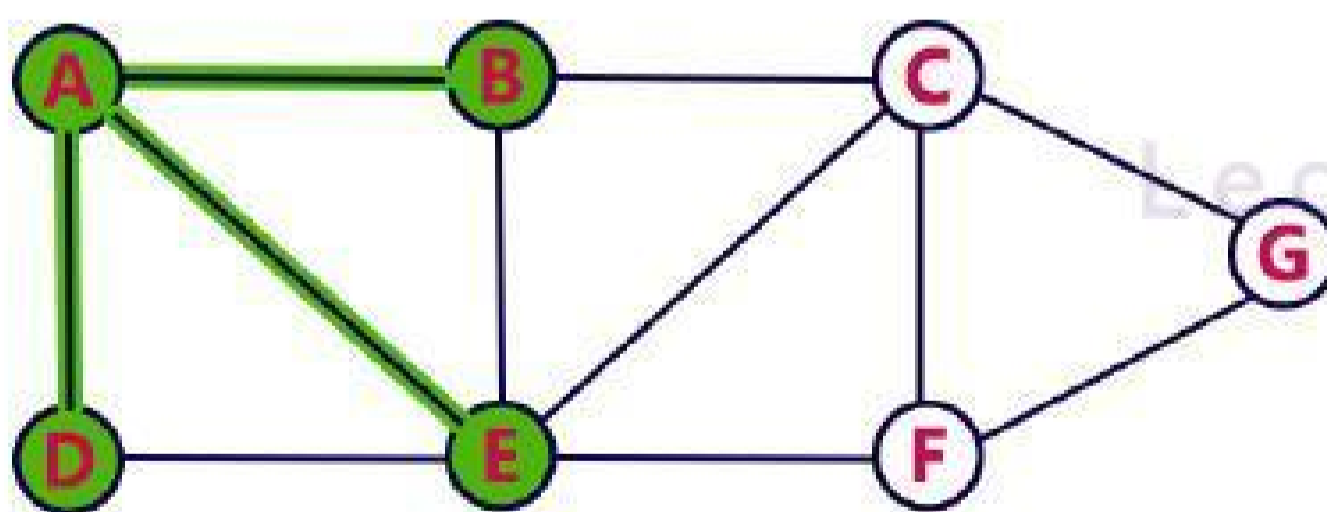


Queue



Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete **A** from the Queue..

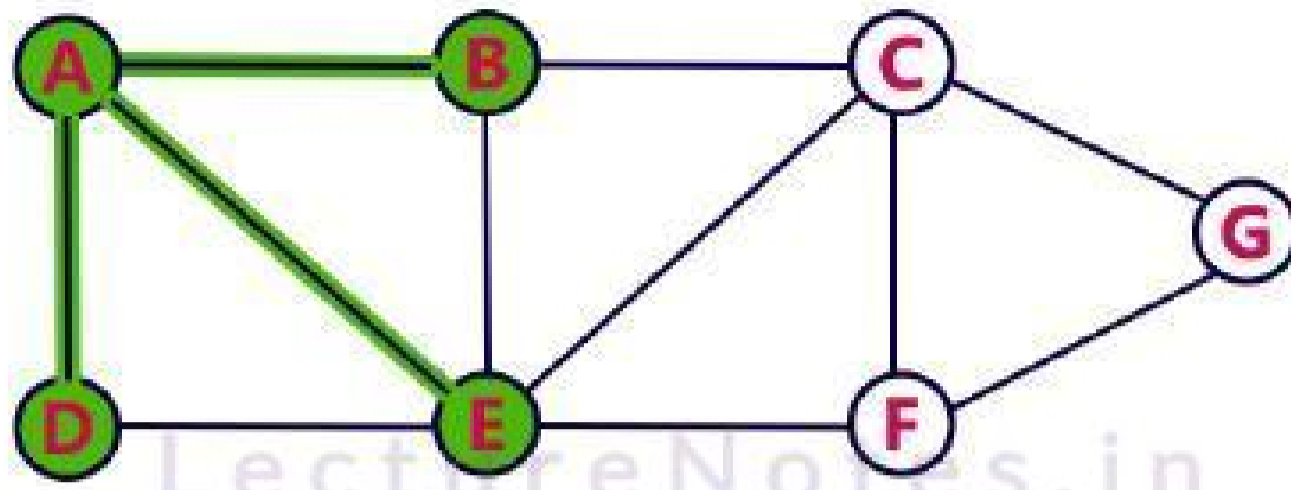


Queue

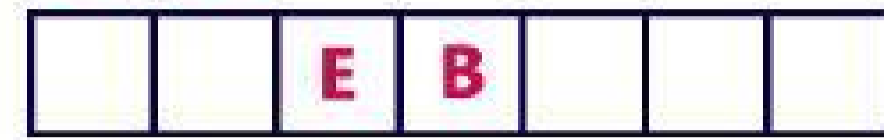


Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

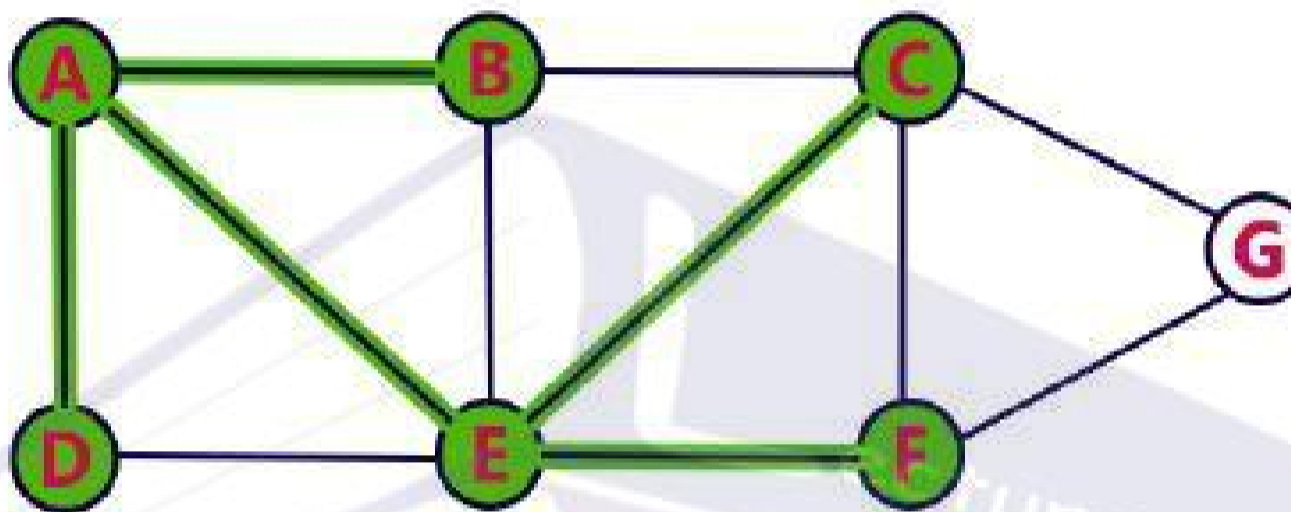


Queue

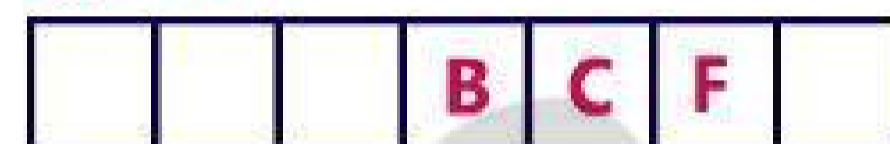


Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C, F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

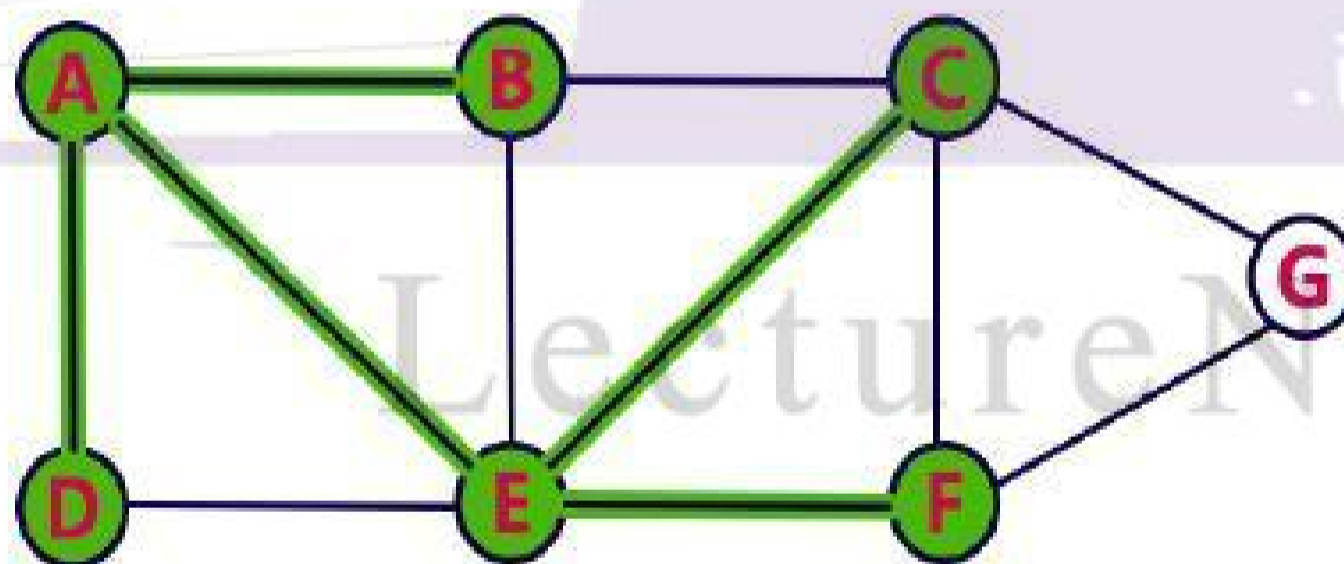


Queue

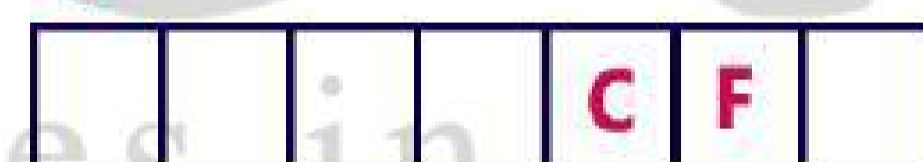


Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

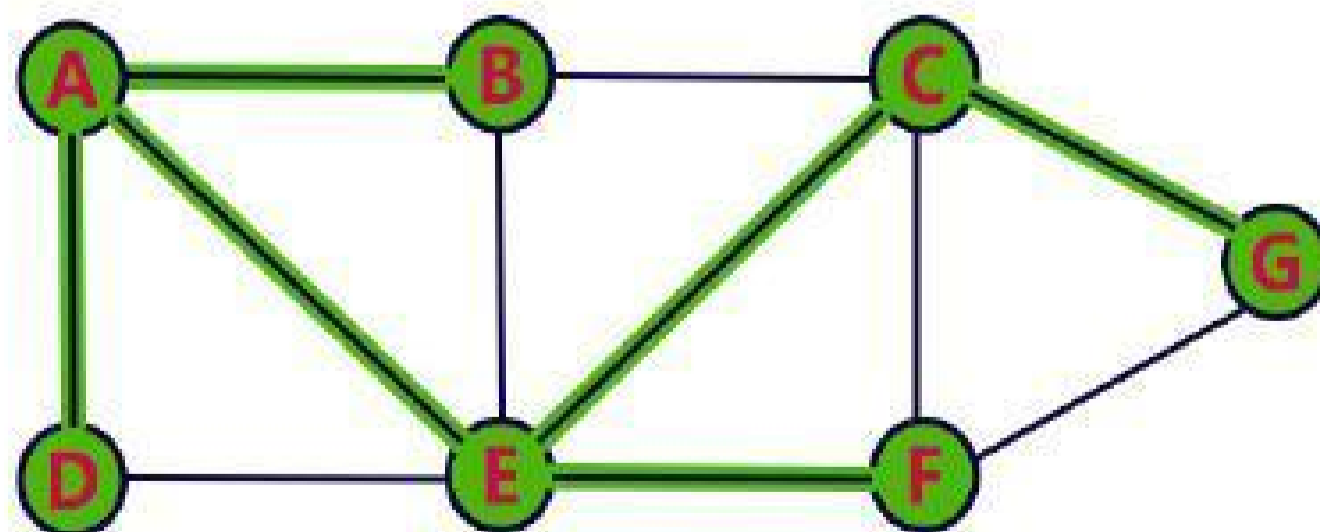


Queue

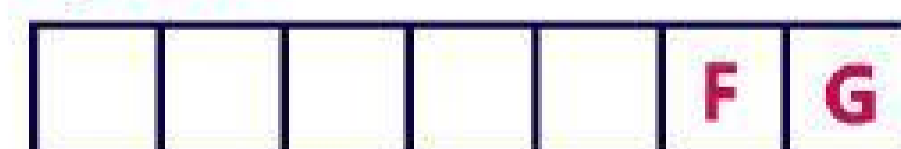


Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

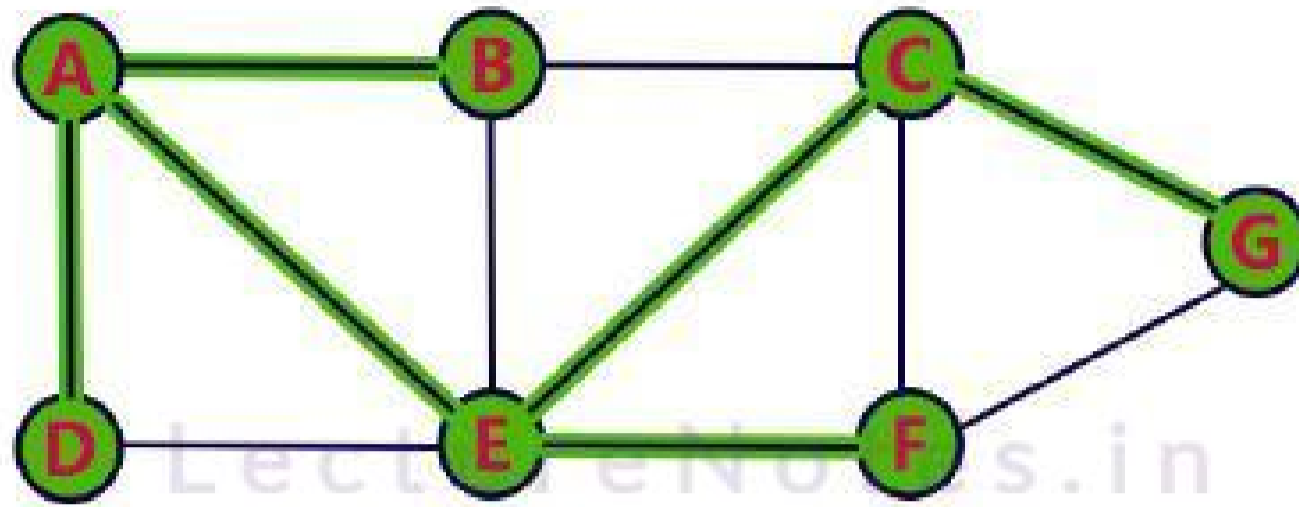


Queue

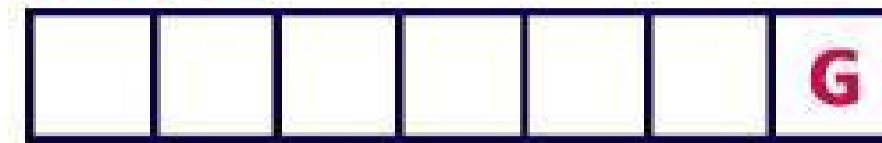


Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

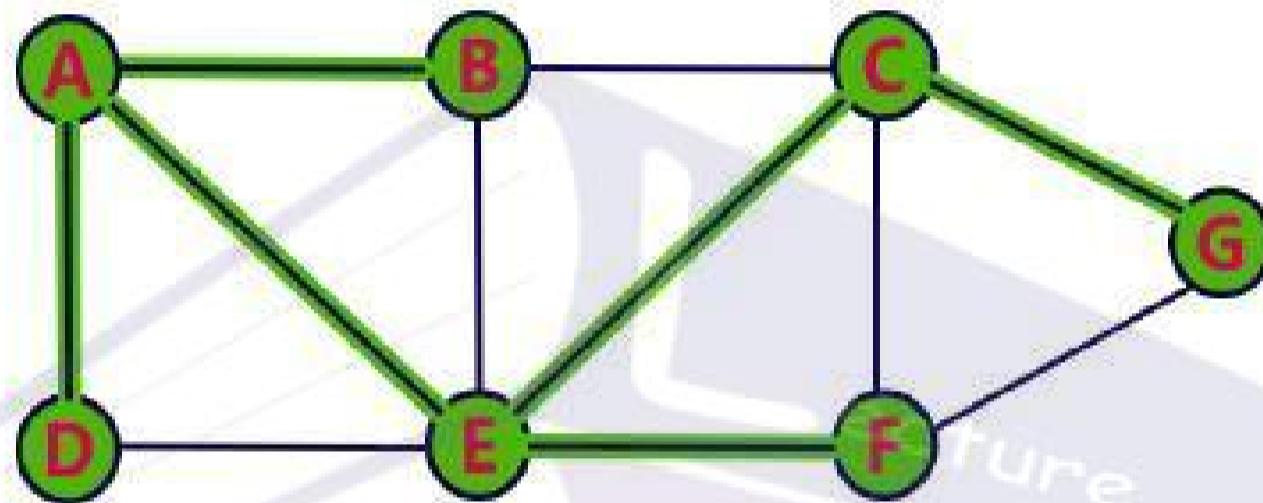


Queue

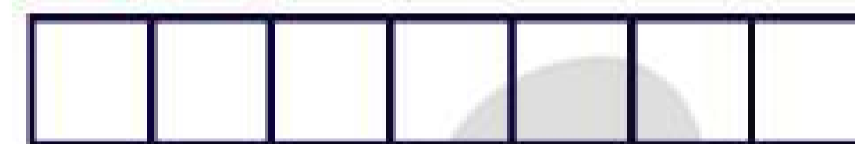


Step 8:

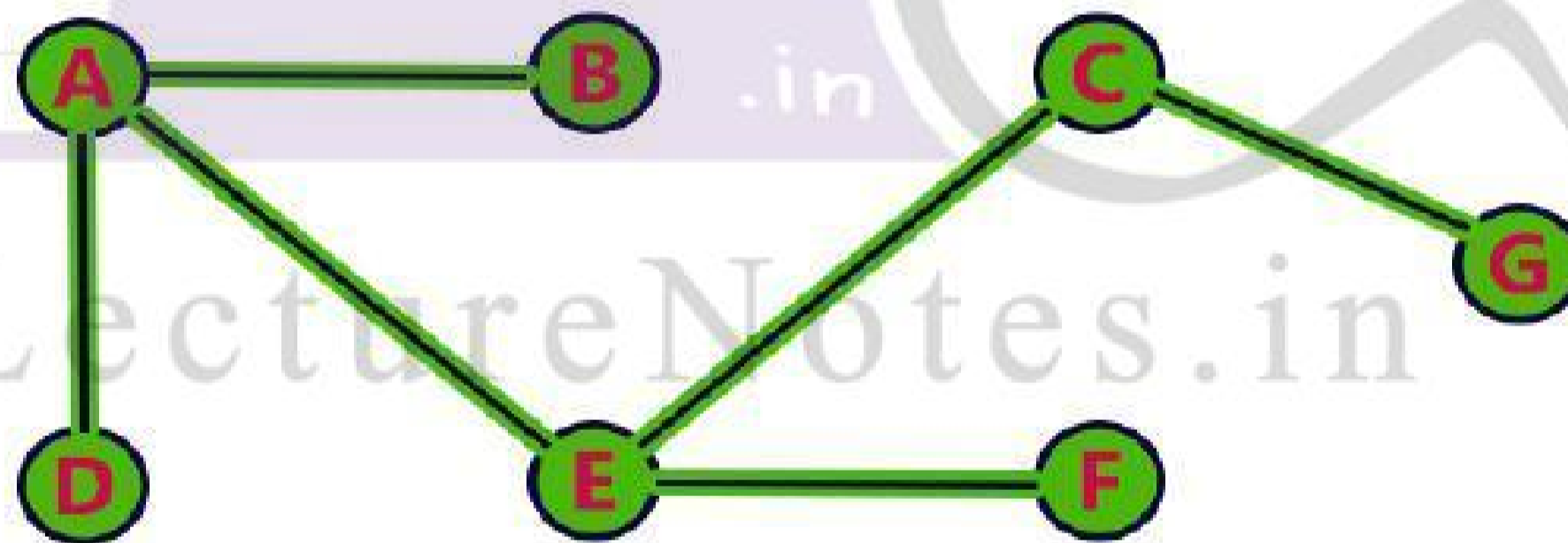
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



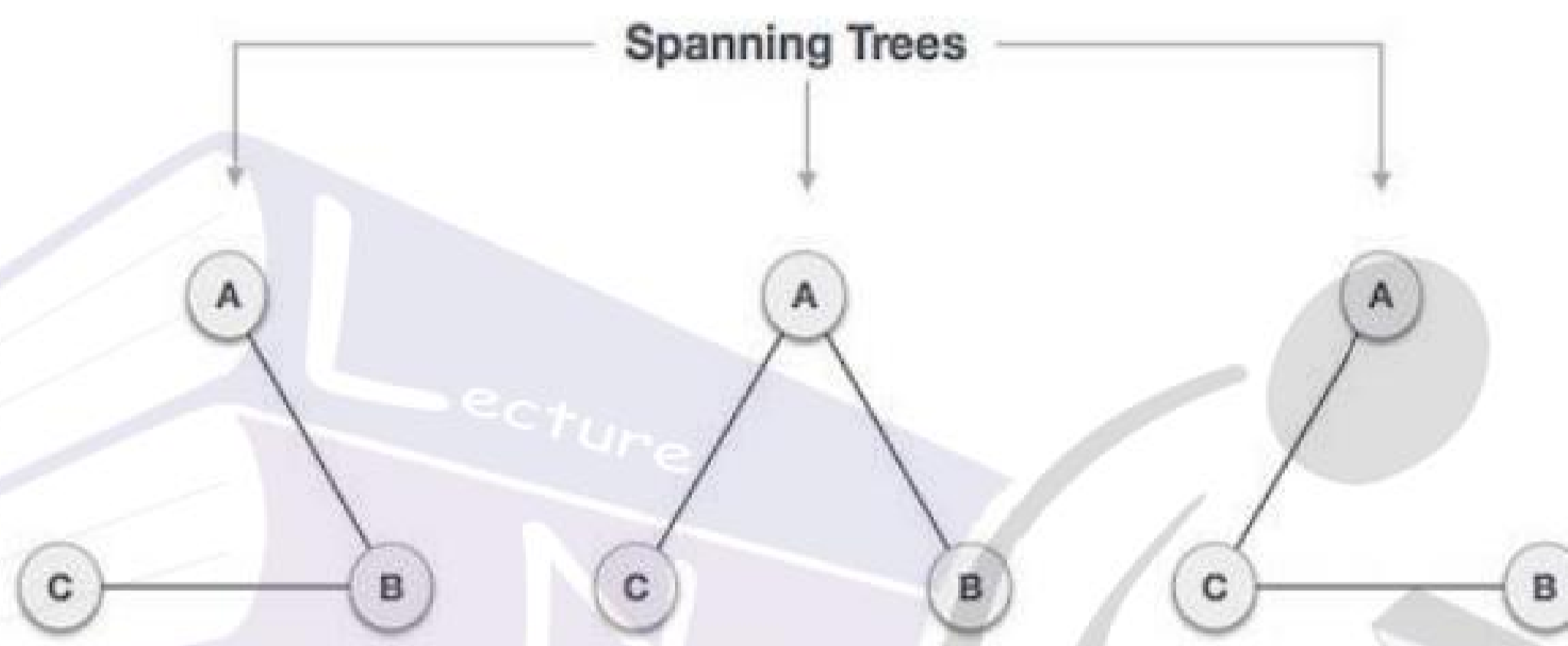
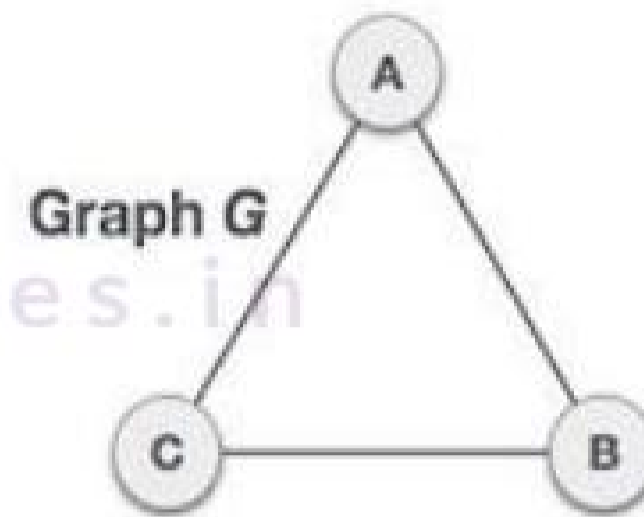
- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Define Spanning Tree and explain minimum spanning tree algorithms

Spanning Tree

- A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes. In the above example, $3^{3-2} = 3$ spanning trees are possible.

Properties of Spanning Tree:

1. A connected graph G can have more than one spanning tree.
2. Each pair of nodes in the spanning trees should possess single path.
3. All possible spanning trees of graph G have the same number of edges and vertices.
4. Spanning tree has $n-1$ edges, where n is the number of nodes (vertices).
5. From a complete graph, by removing maximum $e - n + 1$ edge, we can construct a spanning tree.
6. A complete graph can have maximum n^{n-2} number of spanning trees.
7. The spanning tree does not have any cycle (loops).

8. Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
9. Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

Application of Spanning Tree

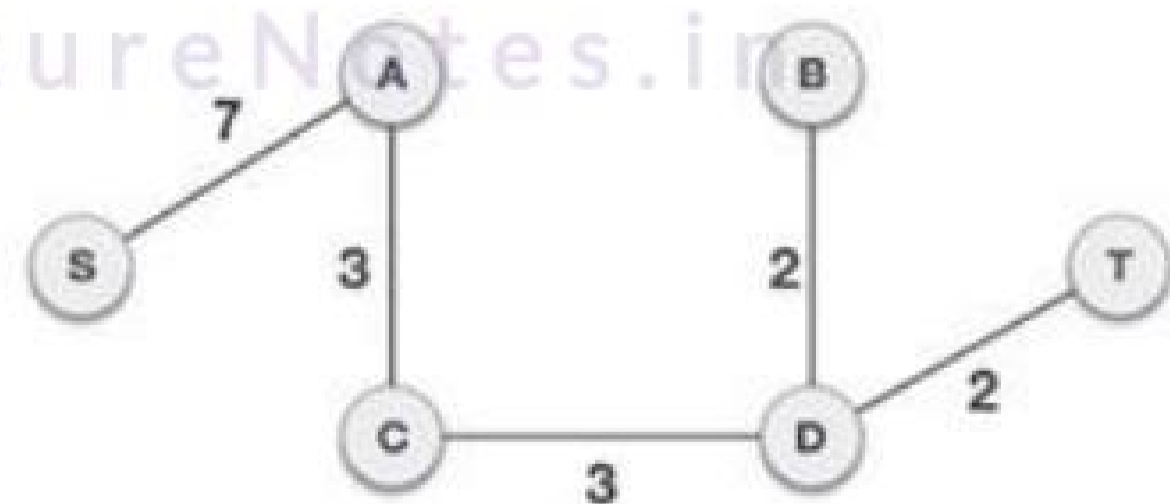
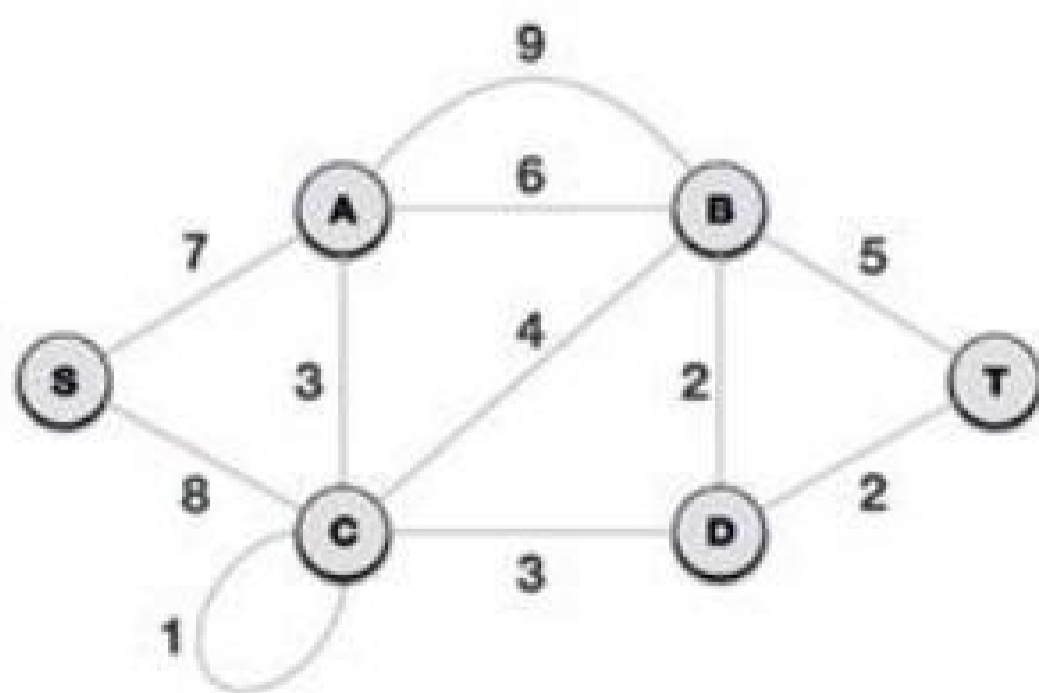
Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common applications of spanning trees are

- **Civil Network Planning**
- **Computer Network Routing Protocol**
- **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

Minimum Spanning Tree (MST):

- In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.
- In real-world situations, this weight can be measured as distance, traffic load or any arbitrary value denoted to the edges.



Minimum Spanning-Tree Algorithm

There are two famous algorithms for finding the Minimum Spanning Tree:

- Kruskal's Algorithm
- Prim's Algorithm

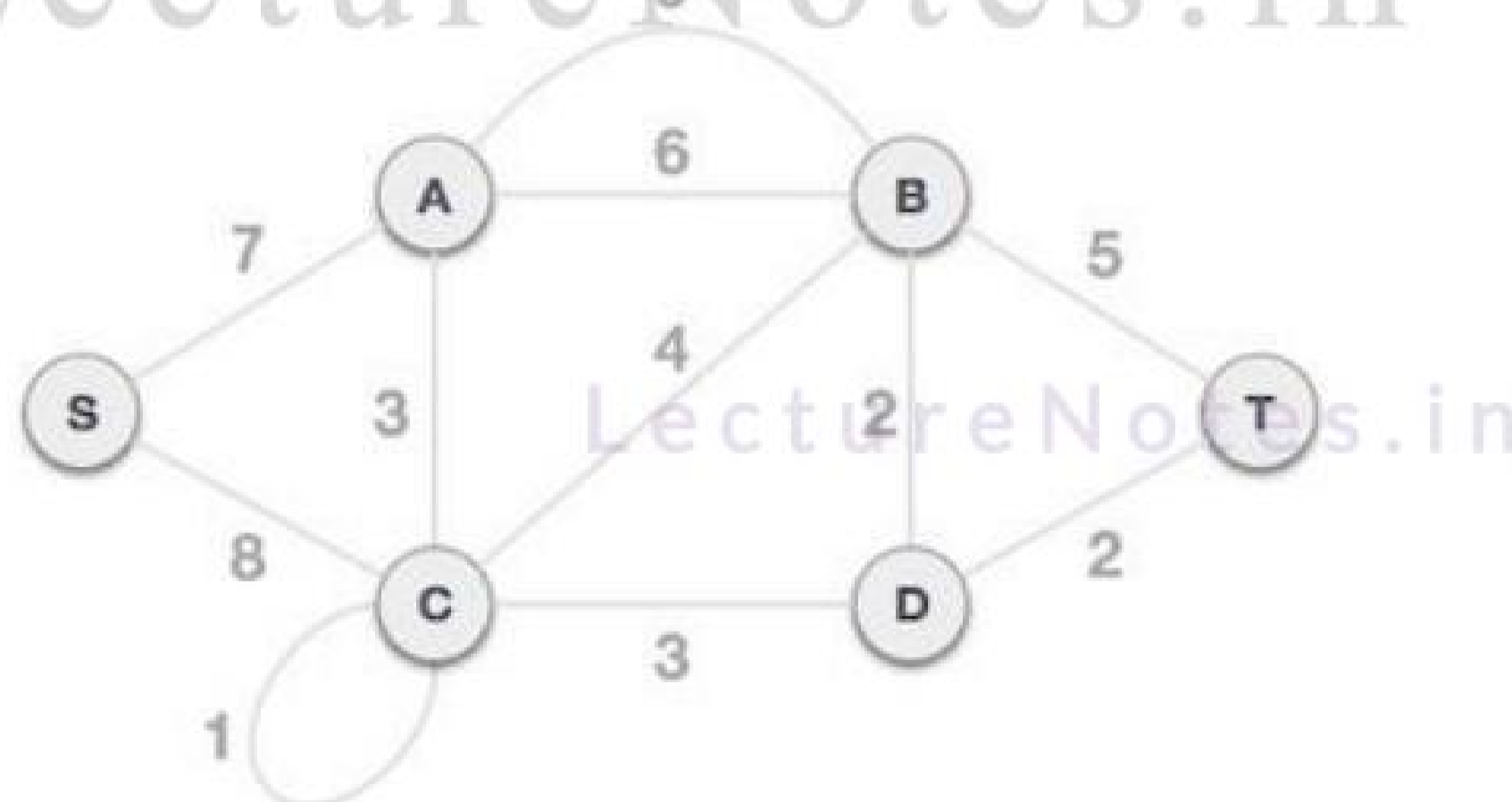
Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

Algorithm Steps:

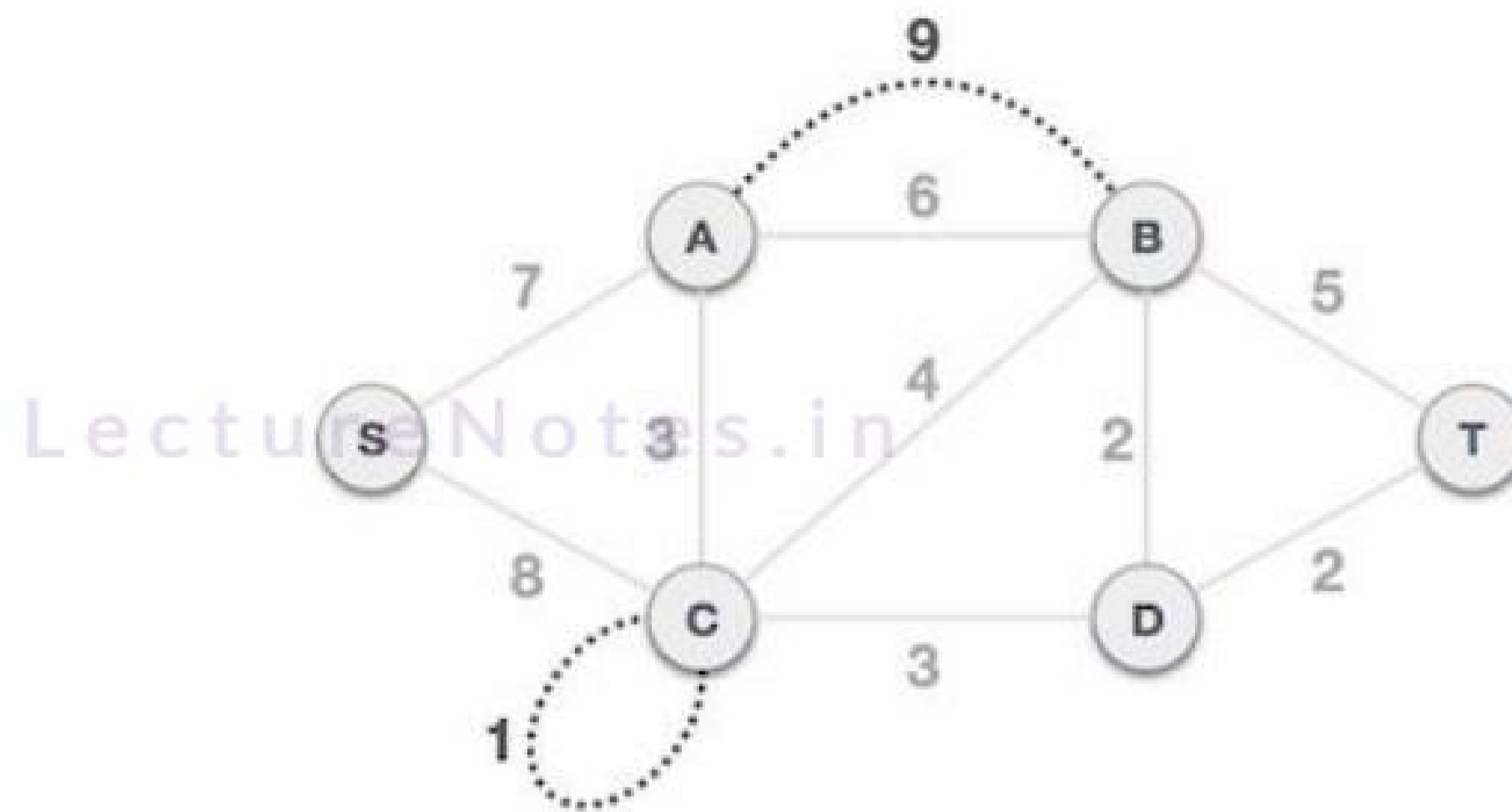
- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

To understand Kruskal's algorithm let us consider the following example –

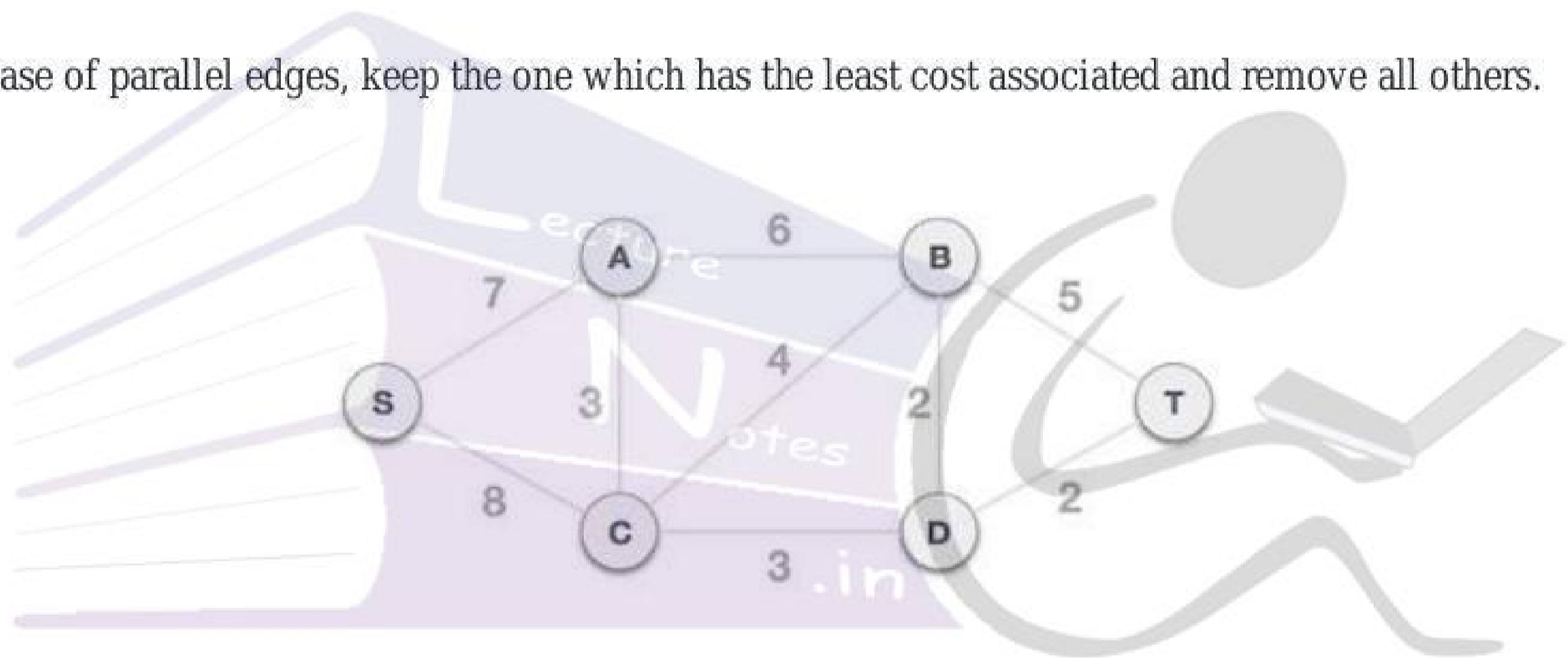


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



Step 2 - Arrange all edges in their increasing order of weight

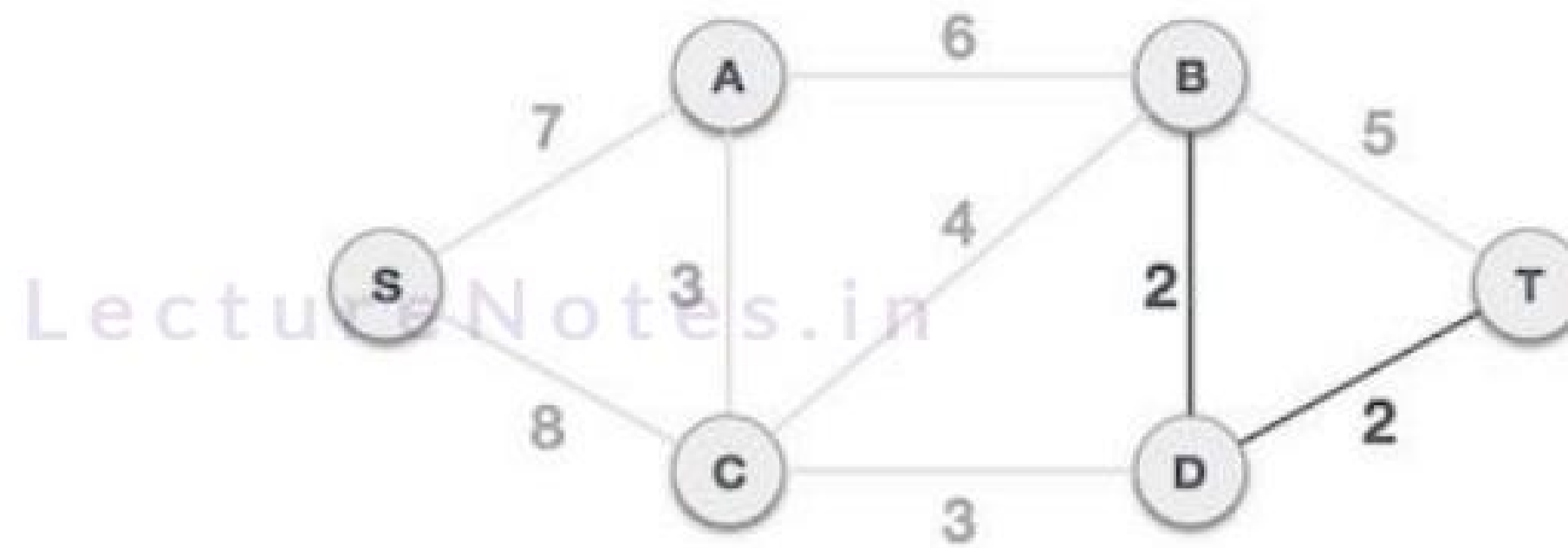
The next step is to create a set of edges and weight, and arrange them in an ascending order of weight (cost).

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weightage

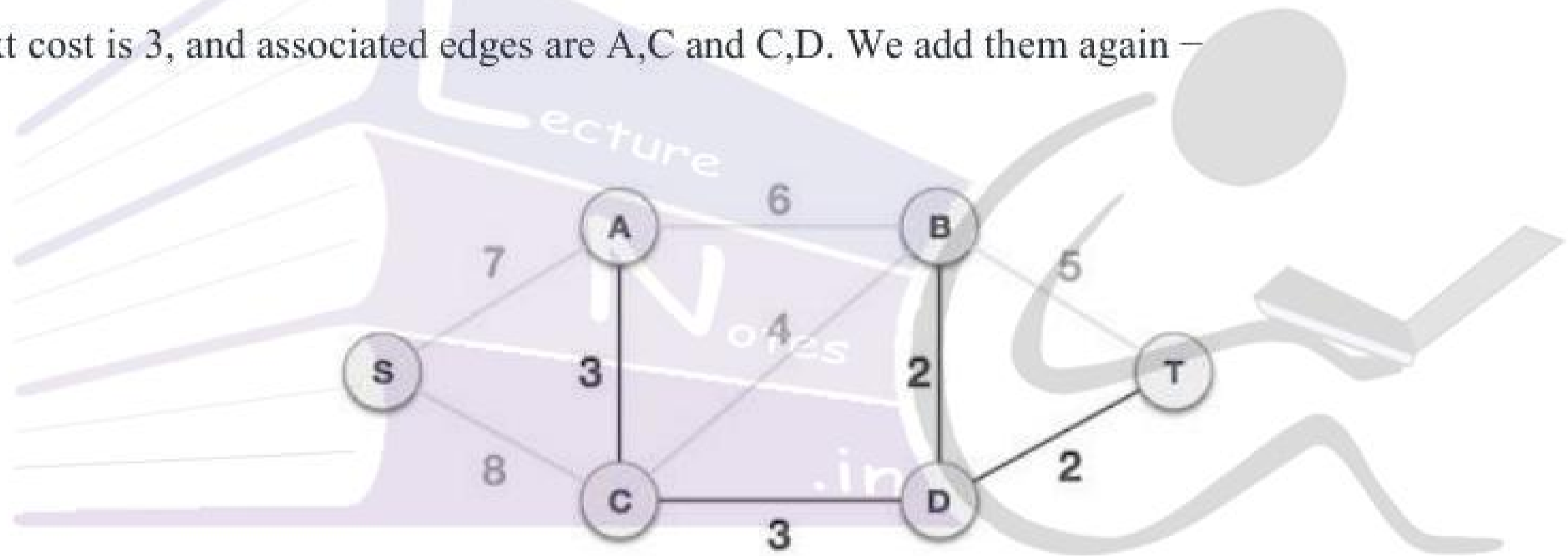
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding

one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

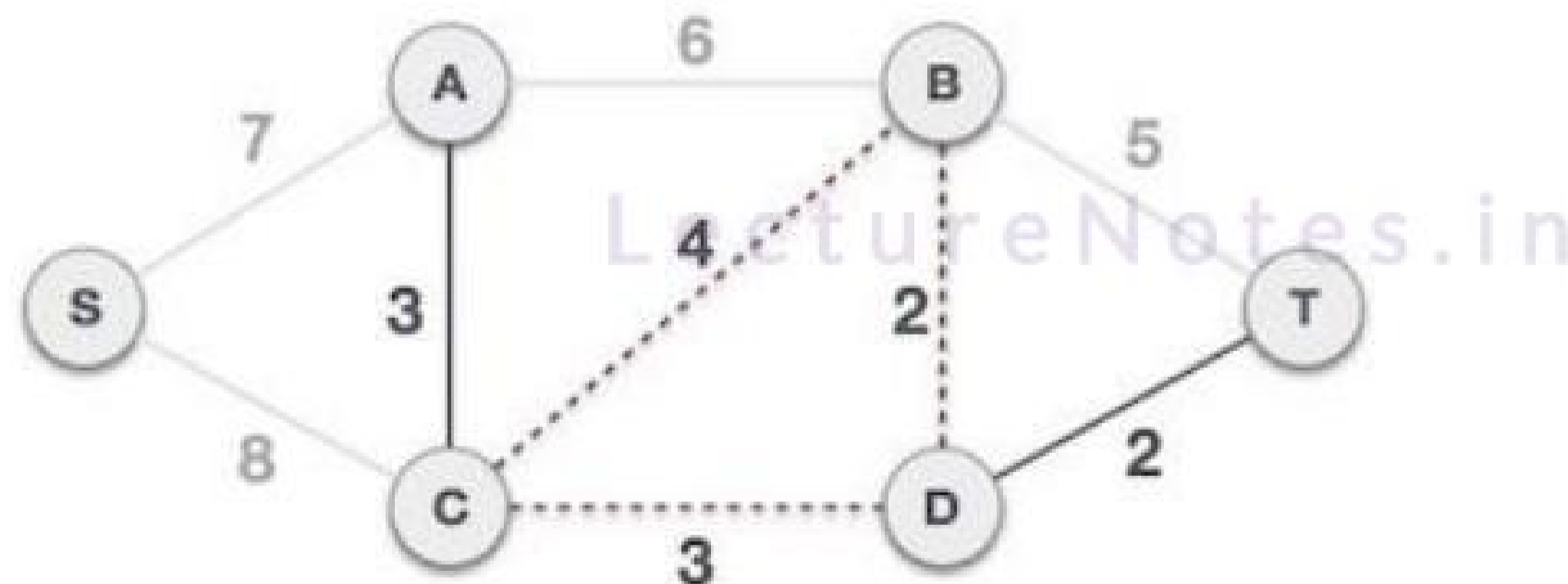


The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

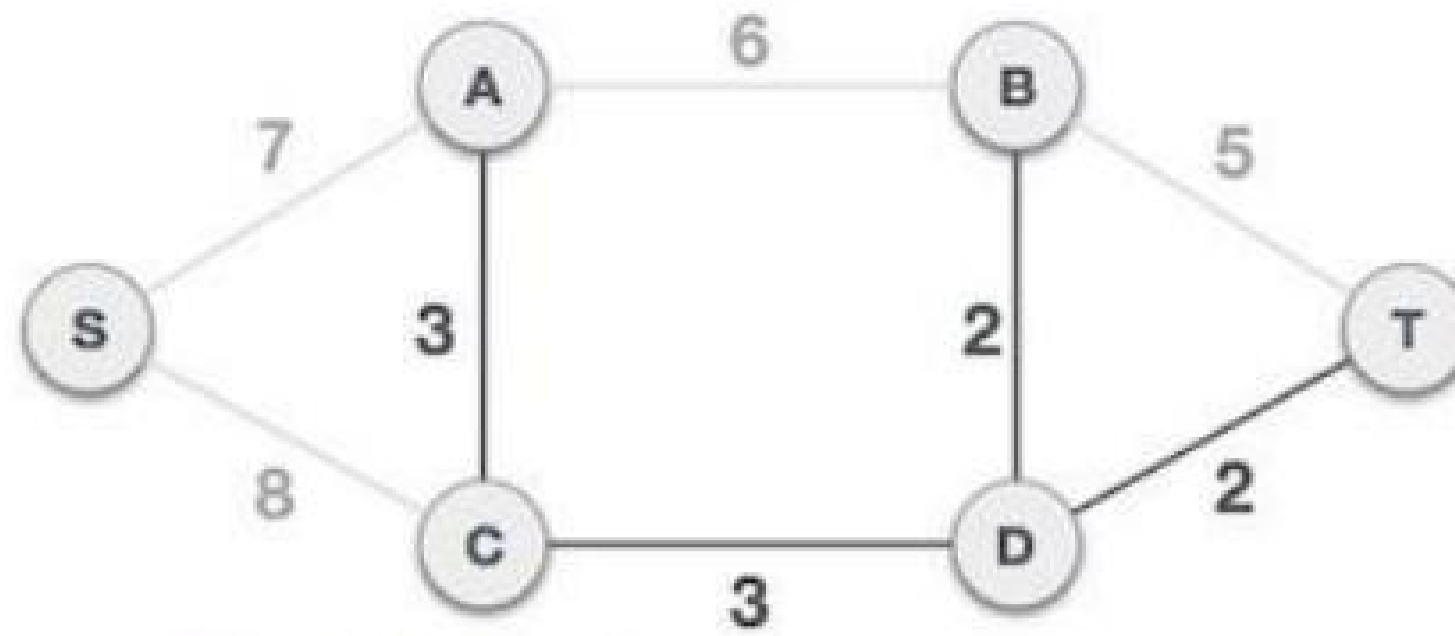
Next cost is 3, and associated edges are A,C and C,D. We add them again –



Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –

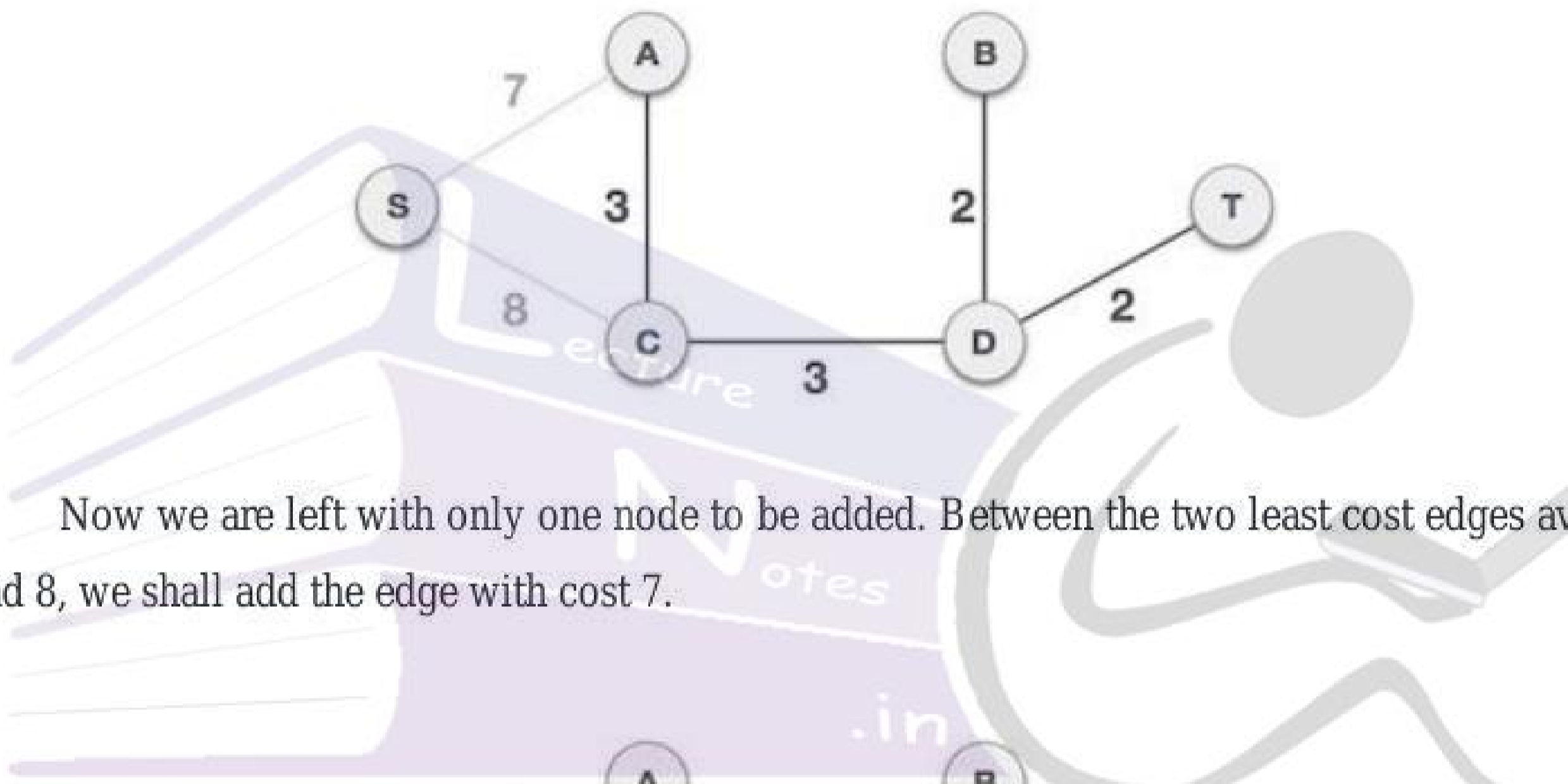


We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

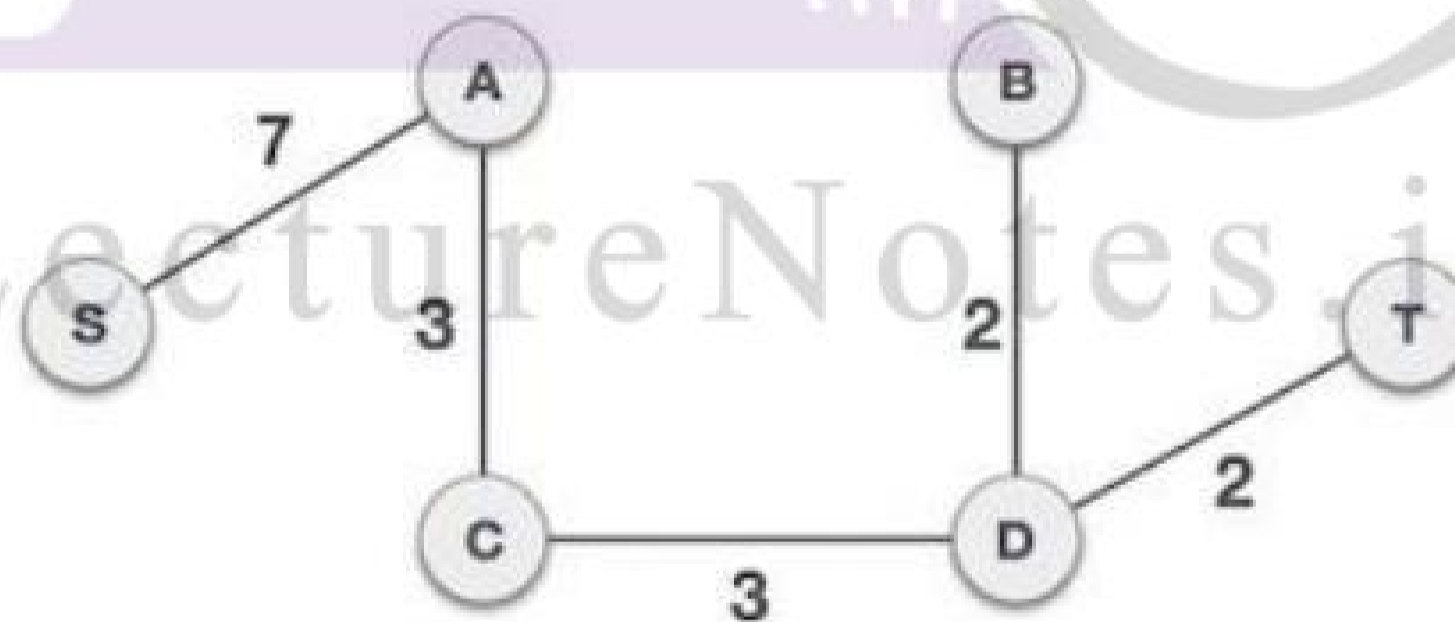


LectureNotes.in

We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Prim's algorithm

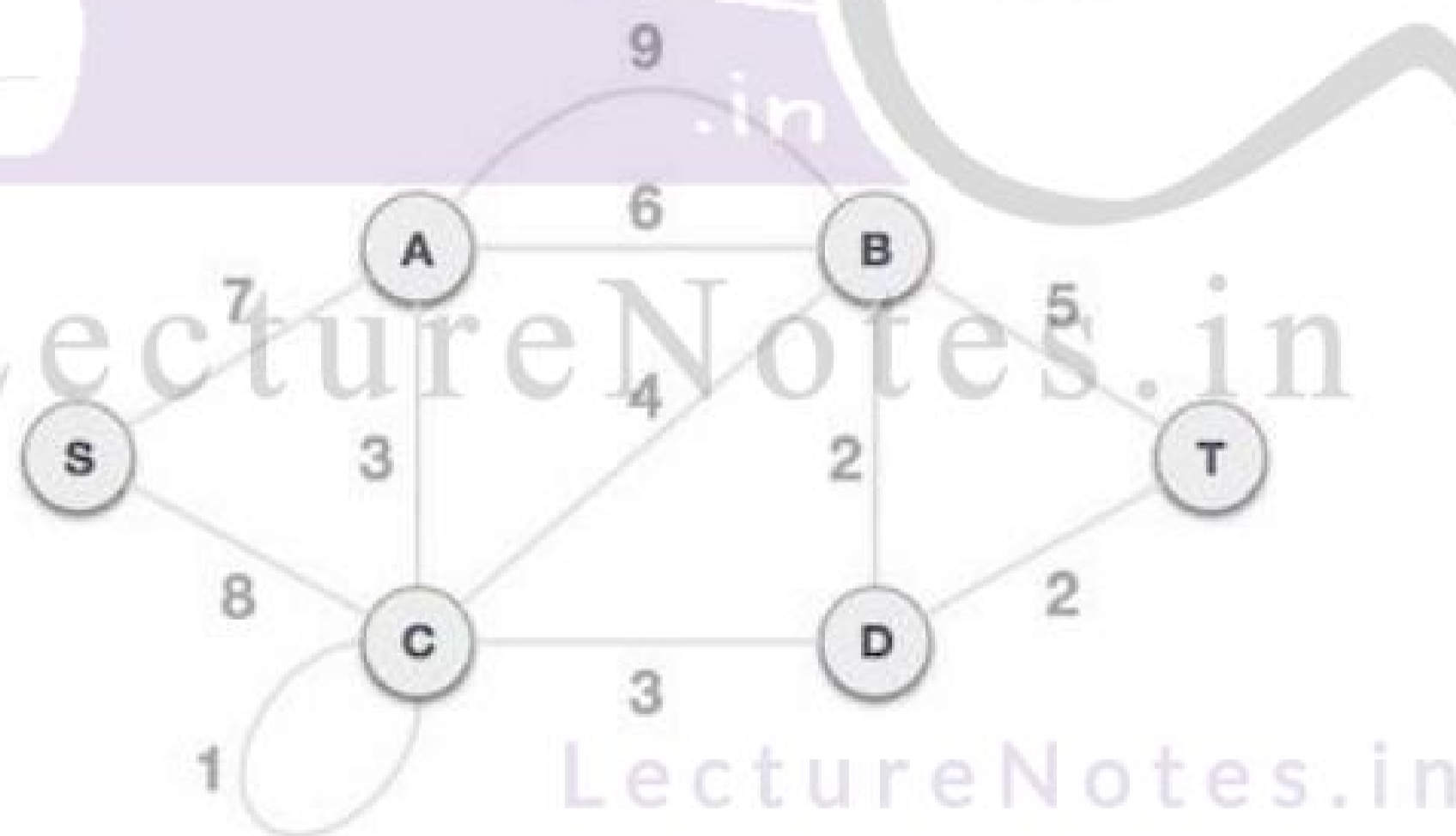
Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included.

At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

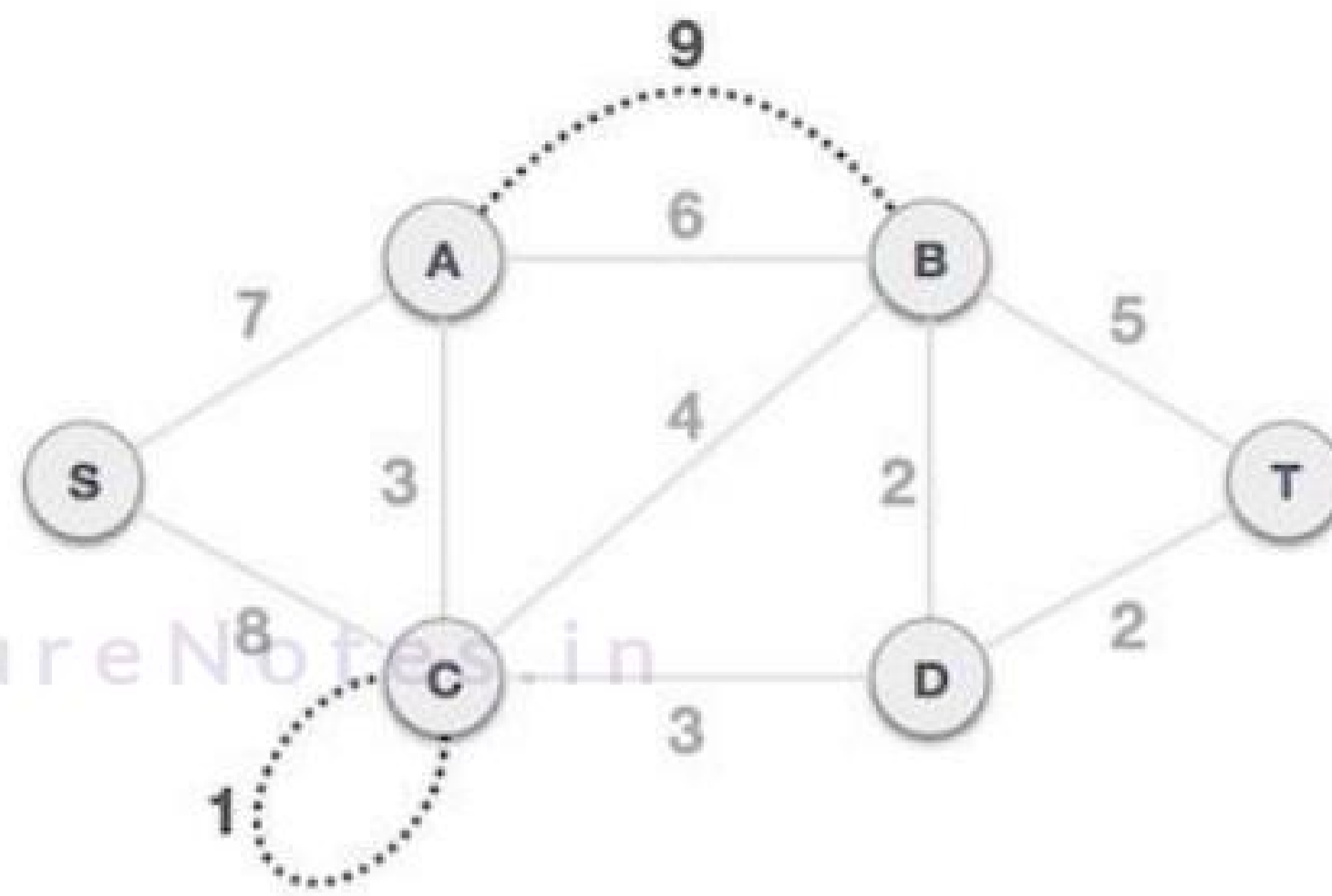
Prim's Procedure

- Initialize the min priority queue Q to contain all the vertices.
- Set the key of each vertex to ∞ and root's key is set to zero
- Set the parent of root to NIL
- If weight of vertex is less than key value of the vertex, connect the graph.
- Repeat the process till all vertex are used.

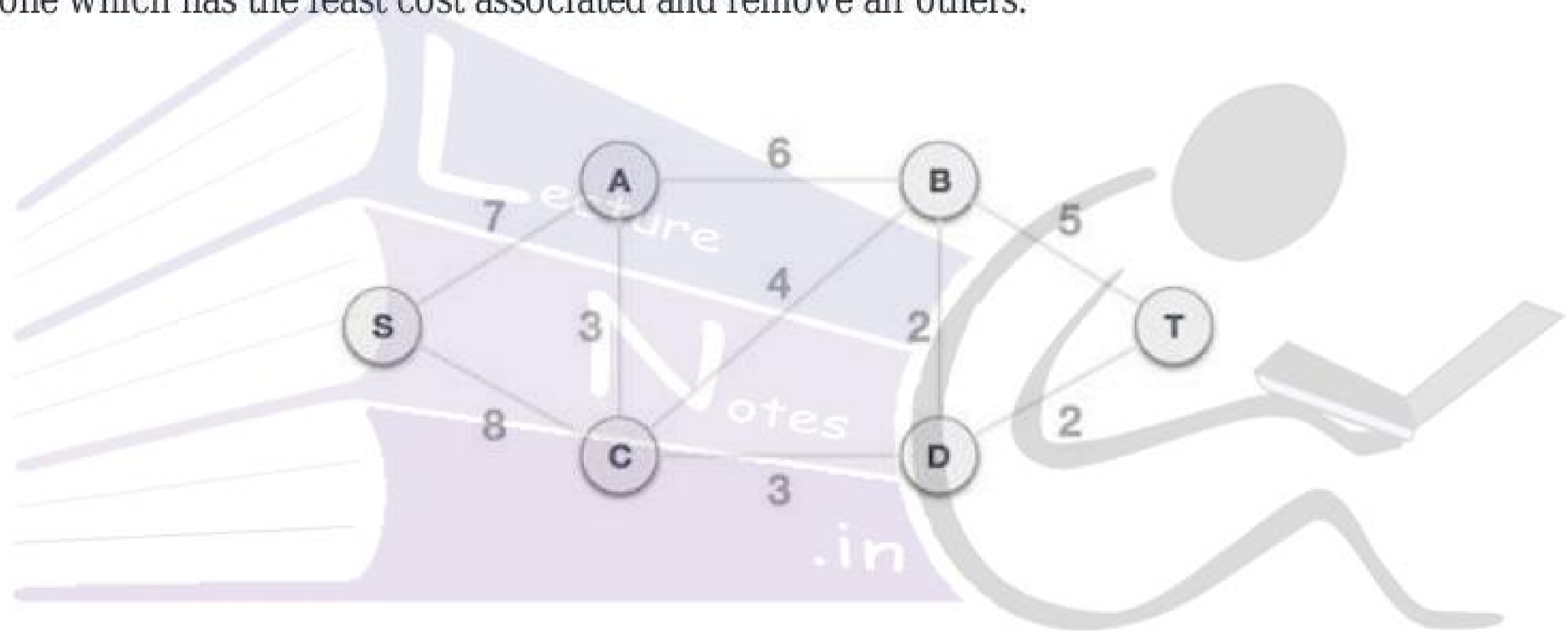
To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same example –



Step 1 - Remove all loops and parallel edges



Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

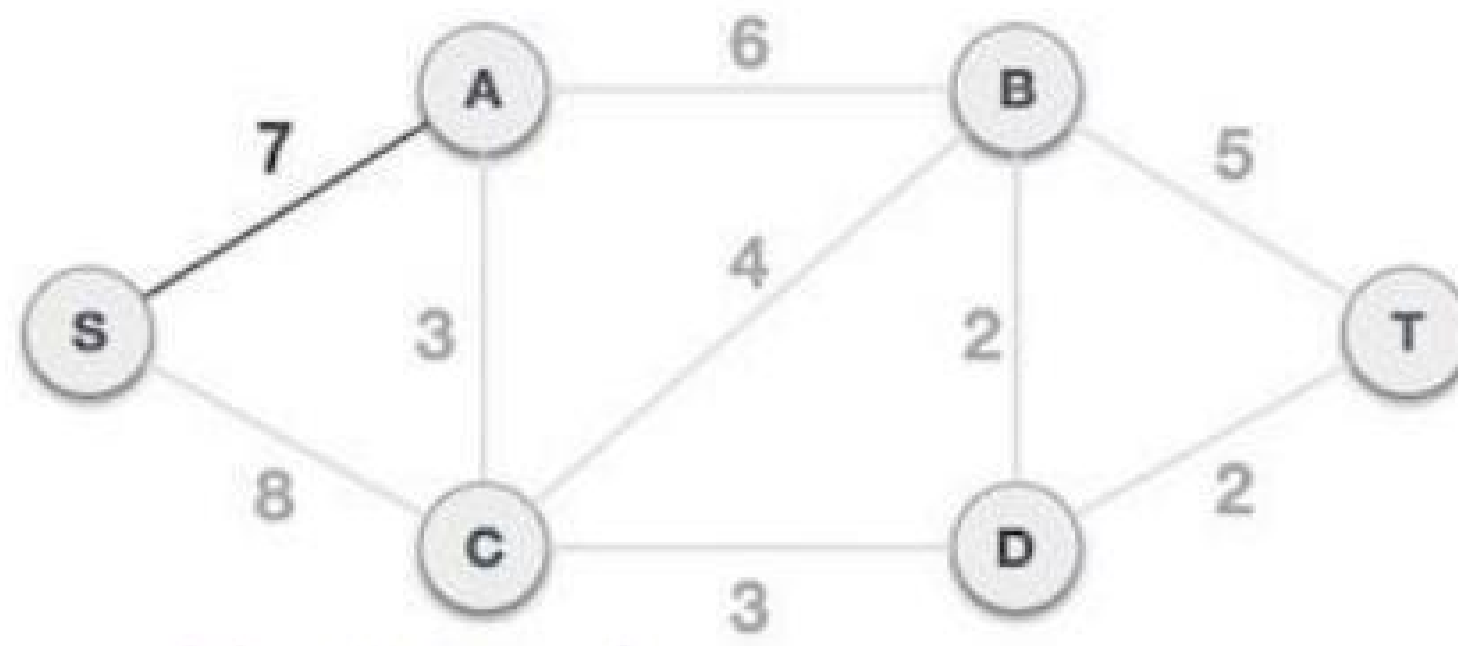


Step 2 - Choose any arbitrary node as root node

In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node.

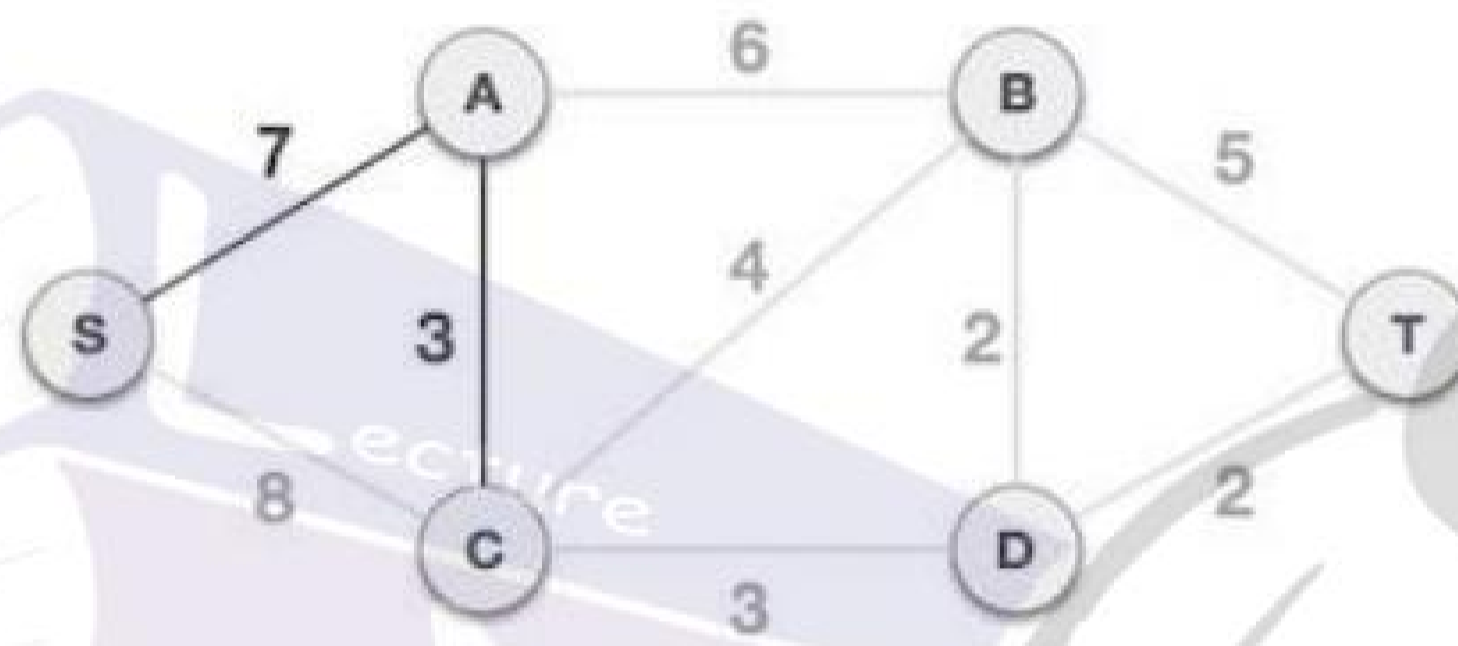
Step 3 - Check outgoing edges and select the one with less cost

After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

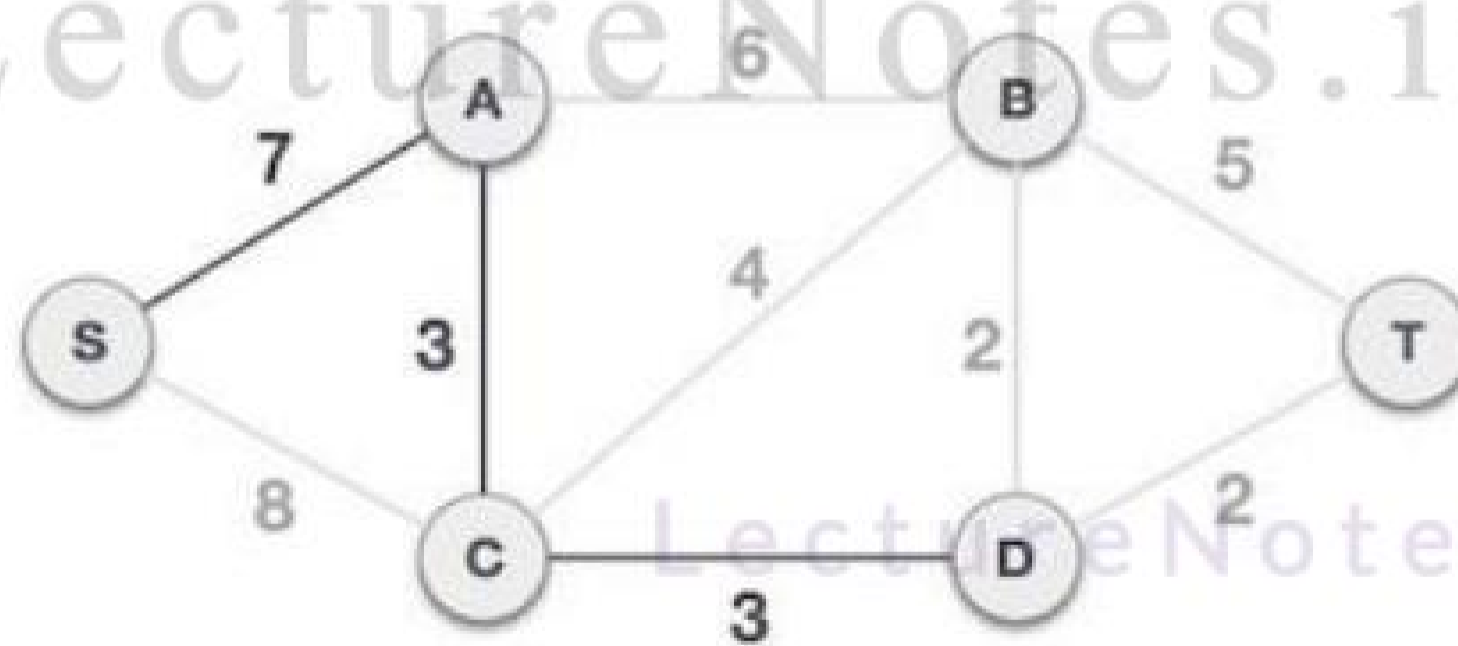


LectureNotes.in

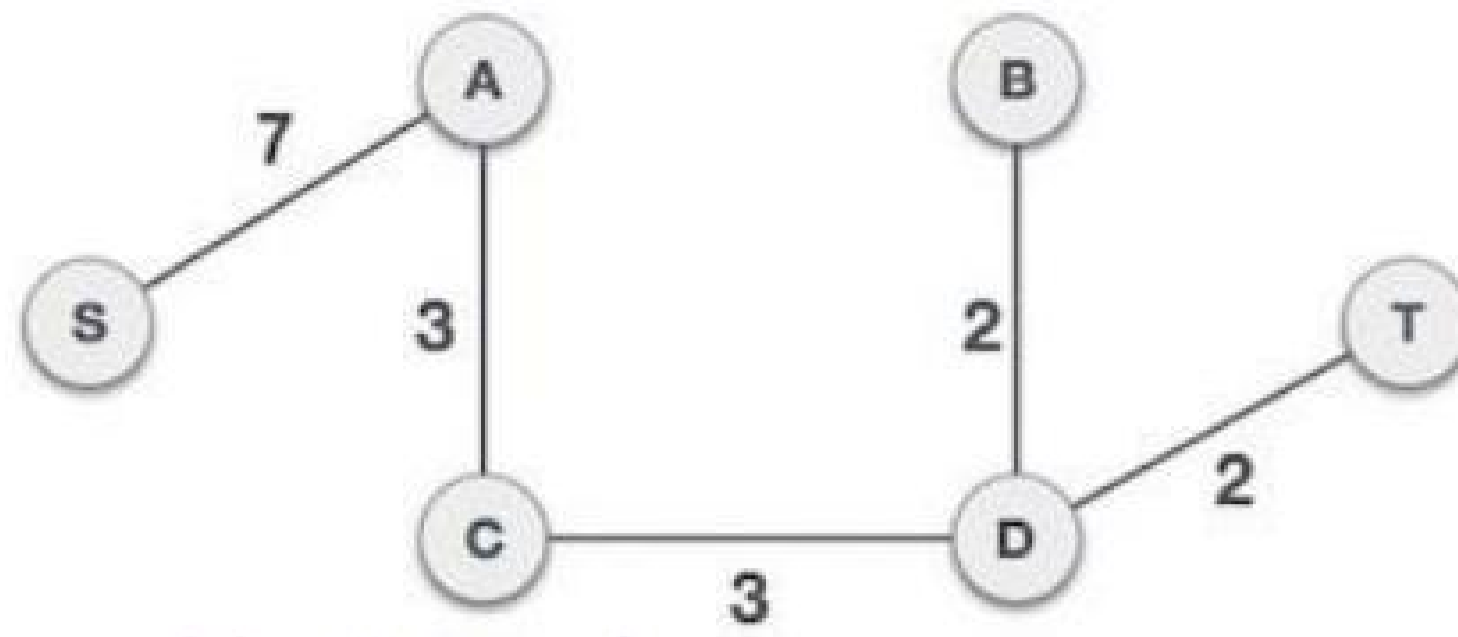
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.

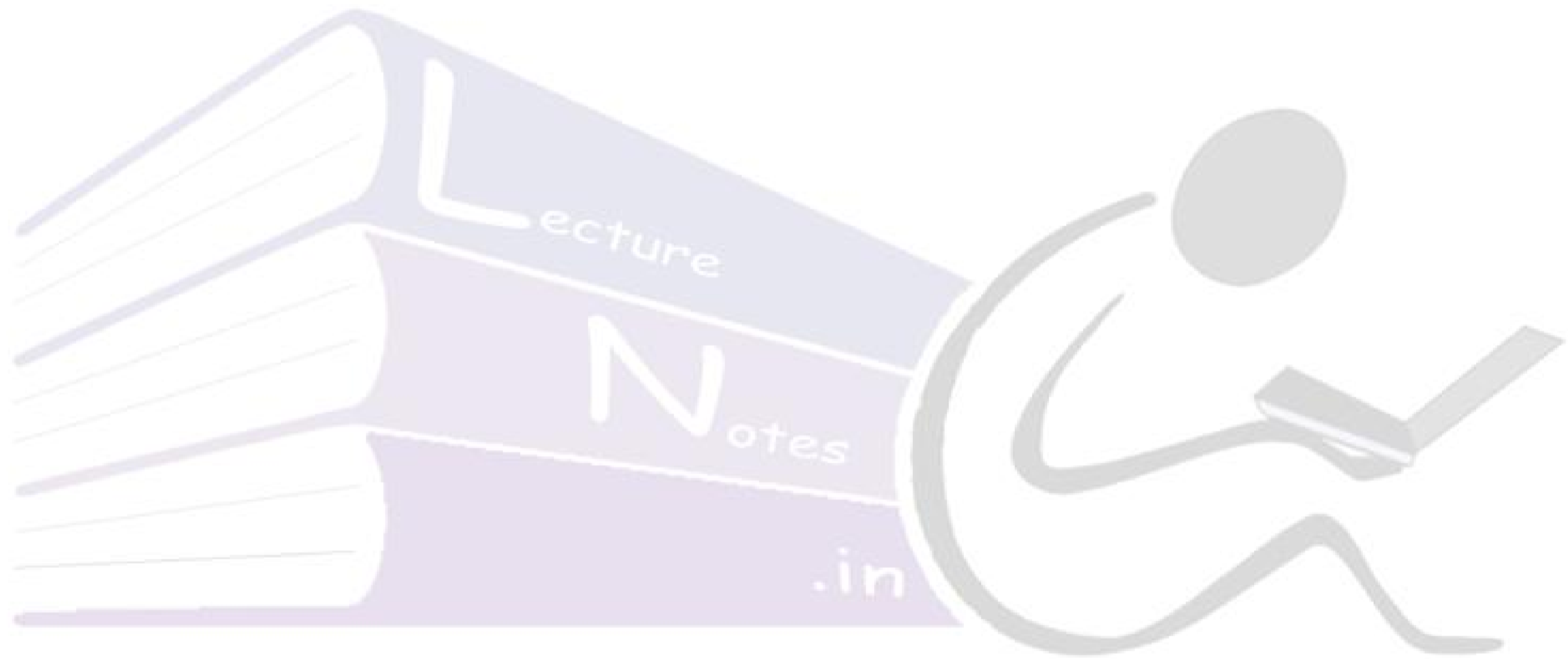


After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



LectureNotes.in

We may find that the output spanning tree of the same graph using two different algorithms is same.



LectureNotes.in

LectureNotes.in