



# Data Structure Lab

[Document subtitle]



**Engr. Khuram Shahzad**  
**CL20001 – Data Structure-Lab**



Array



Linked List



Stack



Queue



Binary Tree



Hash Table



Matrix



Sparse Matrix



Heap



OCTOBER 28, 2022

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES, FAST- PESHAWAR CAMPUS  
Department of Computer Science & Software Engineering

**Instructor: Engr. Khuram Shahzad, FAST NUCES PWR**



## Contents

<b>INTRODUCTION .....</b>	<b>5</b>
<b>COURSE INTRODUCTION.....</b>	<b>5</b>
<b>LAB-01.....</b>	<b>8</b>
<b>OBJECTIVE.....</b>	<b>8</b>
PRE-LAB READING ASSIGNMENT .....	8
Tools/ Apparatus: .....	8
LAB RELATED CONTENT .....	8
Experiment 01: Write a program to perform following operations in arrays (Insert an element, Find an element, delete an element) .....	8
Pointers.....	11
Pointers and Arrays .....	18
<b>String in C++.....</b>	<b>23</b>
Example 2: C++ String to read a line of text .....	24
<b>LAB-02.....</b>	<b>29</b>
<b>OBJECTIVE.....</b>	<b>29</b>
PRE-LAB READING ASSIGNMENT .....	29
Tools/ Apparatus: .....	29
LAB RELATED CONTENT .....	29
Experiment 01: Write a program to perform following operations in arrays (Insert an element, Find an element, delete an element) .....	29
Experiment 02: Write a program to find an element using binary search. ....	32
<b>LAB-03.....</b>	<b>34</b>
<b>OBJECTIVE.....</b>	<b>34</b>
PRE-LAB READING ASSIGNMENT .....	34
Tools/ Apparatus: .....	34
LAB RELATED CONTENT .....	34
<b>C++ Dynamic Memory.....</b>	<b>34</b>
delete operator.....	37
Dynamic Memory Allocation for Objects .....	38
C++ Dynamic Memory Allocation Example Arrays .....	42



C++ Dynamic Memory Allocation Example Multi-Dimensional Arrays .....	42
Dynamic memory allocation in C++ for 2D and 3D array .....	45
1. Single Dimensional Array .....	45
2. 2-Dimensional Array .....	46
1. Using Single Pointer .....	46
2. Using array of Pointers .....	47
3. 3-Dimensional Array .....	49
1. Using Single Pointer .....	49
As seen for the 2D array, we allocate memory of size $X \times Y \times Z$ dynamically and assign it to a pointer. Then we use pointer arithmetic to index the 3D array. ....	49
2. Using Triple Pointer .....	50
C++ program for array implementation of List ADT .....	52
<b>LAB-04</b> .....	56
OBJECTIVE .....	56
PRE-LAB READING ASSIGNMENT .....	56
Apparatus/Tools .....	56
Lab Related Contents:.....	56
<b>LAB-05</b> .....	62
OBJECTIVE .....	62
PRE-LAB READING ASSIGNMENT .....	62
Apparatus/Tools .....	62
Lab Related Contents:.....	62
<b>Doubly Linked List  </b> .....	62
Advantages of DLL over the singly linked list:.....	62
Disadvantages of DLL over the singly linked list: .....	62
Insertion in DLL: .....	63
<b>LAB-06</b> .....	70
OBJECTIVE .....	70
PRE-LAB READING ASSIGNMENT .....	70
LAB RELATED CONTENT .....	70
Circular Linked List .....	70
<b>LAB-07</b> .....	73



OBJECTIVE .....	73
PRE-LAB READING ASSIGNMENT .....	73
Apparatus/ Tools .....	73
LAB RELATED CONTENT .....	73
<b>LAB-08.....</b>	<b>78</b>
OBJECTIVE .....	78
PRE-LAB READING ASSIGNMENT .....	78
Tools/ Apparatus: .....	78
LAB RELATED CONTENT .....	78
<b>LAB-09.....</b>	<b>85</b>
<b>OBJECTIVE.....</b>	<b>85</b>
<b>PRE-LAB READING ASSIGNMENT.....</b>	<b>85</b>
<b>Tools/ Apparatus:.....</b>	<b>85</b>
<b>LAB RELATED CONTENT .....</b>	<b>85</b>
<b>LAB-10.....</b>	<b>90</b>
<b>OBJECTIVE.....</b>	<b>90</b>
<b>PRE-LAB READING ASSIGNMENT.....</b>	<b>90</b>
<b>Tools/ Apparatus:.....</b>	<b>90</b>
<b>LAB RELATED CONTENT .....</b>	<b>90</b>
<b>LAB-11.....</b>	<b>98</b>
<b>OBJECTIVE.....</b>	<b>98</b>
<b>PRE-LAB READING ASSIGNMENT.....</b>	<b>98</b>
<b>Tools/ Apparatus:.....</b>	<b>98</b>
<b>LAB RELATED CONTENT .....</b>	<b>98</b>
AVL Tree.....	98
Why AVL Trees? .....	99
Insertion in AVL Tree:.....	99
<b>LAB-12.....</b>	<b>111</b>
<b>OBJECTIVE.....</b>	<b>111</b>
<b>PRE-LAB READING ASSIGNMENT.....</b>	<b>111</b>
<b>Tools/ Apparatus:.....</b>	<b>111</b>



<b>LAB RELATED CONTENT .....</b>	<b>111</b>
<b>LAB-13.....</b>	<b>116</b>
<b>OBJECTIVE.....</b>	<b>116</b>
<b>PRE-LAB READING ASSIGNMENT .....</b>	<b>116</b>
<b>Tools/ Apparatus:.....</b>	<b>116</b>
<b>LAB-14.....</b>	<b>119</b>
<b>OBJECTIVE.....</b>	<b>119</b>
<b>PRE-LAB READING ASSIGNMENT .....</b>	<b>119</b>
<b>Tools/ Apparatus:.....</b>	<b>119</b>
<b>LAB RELATED CONTENT .....</b>	<b>119</b>
Whats Is A Graph In C++? .....	119
Applications Of Graphs .....	132
<b>LAB-15.....</b>	<b>134</b>
<b>OBJECTIVE.....</b>	<b>134</b>
<b>PRE-LAB READING ASSIGNMENT .....</b>	<b>134</b>
<b>Tools/ Apparatus:.....</b>	<b>134</b>
<b>LAB RELATED CONTENT .....</b>	<b>134</b>
Depth First Search (DFS) In C++ .....	134
BFS Vs DFS .....	141
Applications Of DFS .....	141
Conclusion .....	141
Breadth First Search (BFS) C++ Program To Traverse A Graph Or Tre .....	142
Breadth First Search (BFS) Technique In C++ .....	142
Applications Of BFS Traversal.....	147
Conclusion .....	148



## INTRODUCTION

### COURSE INTRODUCTION

<b>Course Title</b>	Data Structures -Lab
<b>Course Code</b>	CL2001
<b>Credit Hours</b>	1
<b>Lab. Instructor</b>	Engr. Khuram Shahzad

#### Uses of data structures:

##### **What Is a Data structures?**

- It is a logical way of storing data and it also define mechanism of retrieve data.
- A data Structure Requires:
  - Space for each data item it stores.
  - Time to perform each basic operation
  - Programming effort

**Example:** Arrays, Stacks etc.

##### **Types of Data structures:**

There are two types of data structures

##### **1. Linear Data Structure**

A data structure is said to be linear if the elements form a sequence.

Two ways of representation:

- by means of Sequential memory Locations
- by means of Links
  - Array
  - Stack
  - Queue
  - Linked List

##### **2. Non-Linear Data structure**

Don't form a direct sequence among elements.

- Tree
- Graph

##### **Operations on Data structures:**

- **Traversing:** Accessing each record exactly once so that certain item in the record may be processed.
- **Searching:** finding the location of the record with a given key value.
- **Insertion:** add a new record to the structure

- **Deletion:** removing a record from the structure

### **Arrays:**

An array is a collection of homogeneous type of data elements. An array is consisting of a collection of elements.

All above mentioned operations plus sorting and merging can be performed on arrays.

### **Limitations of arrays •**

Size of array is fixed.

- Inserting/Deleting an element is expensive **Linked List:**

A Linked list is a linear collection of data elements. It has two part one is info and other is link part. info part gives information and link part is address of next node.

### **Advantage:**

- Dynamic Size
- Ease of Insertion and Deletion

### **Drawbacks:**

- Random access is not allowed

### **STACKS:**

A Stack is a list of elements in which an element may be inserted or deleted at one end which is known as TOP of the stack.

### **Operation on LIFO:**

- **Push:** add an element in stack
- **Pop:** remove an element in stack

**Example:** Operating System stacks for addresses of programs.

### **QUEUES:**

A queue is defined as a special type of data structure where the elements are inserted from one end and elements are deleted from the other end.

- The end from where the elements are inserted is called *REAR end*.
- The end from where the elements are deleted is called *FRONT end*.

Queue is organized as **First In First Out (FIFO)** Data Structure with following operations:

- Insertion: add a new element in queue(Enque)
- Deletion: Removing an element in queue(Deque)

**Examples:** Queues in Operating Systems

### **Trees:**



A tree is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

## Types of Trees

- Binary Trees
- BST Trees
- Heaps
- 2-3-4 Trees
- B Trees **Graphs:**

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Connections/relations in social networking sites, Routing ,networks of communication, data organization etc.

**Hash tables:** Hash Table - used for fast data lookup - symbol table for compilers, database indexing, caches, Unique data representation.



## LAB-01

### OBJECTIVE

After this lab students will be able to understand

- Insertion in Arrays
- Deletion in Arrays
- Traversing
- Pointers
- Binary Search

### PRE-LAB READING ASSIGNMENT

NONE

Tools/ Apparatus:

Dev C++, VS Code

### LAB RELATED CONTENT

#### Arrays:

The array is the most commonly used data storage structure; it's built into most programming languages. Because arrays are so well known, they offer a convenient jumping off place for introducing data structures and for seeing how object-oriented programming and data structures relate to one another. Access File Extensions

---

Experiment 01: Write a program to perform following operations in arrays (Insert an element, Find an element, delete an element)

---

```
#include<iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class Person
{
    private:
    string lastName;
    string firstName;
    int age;
    //-----
    public:

    Person()
    {
    }
    //-----
    void displayPerson()
    {
        cout << "    First name:" << firstName << endl;
```

```
        cout << "    Last name:" << lastName << endl;
        cout << "    Age:" << age << endl;
    }

    void setFirstName(string firstName)
    {
        this->firstName = firstName;
    }
    void setLastName(string lastName)
    {
        this->lastName = lastName;
    }
    void setAge(int age)
    {
        this->age = age;
    }

    //-----
    // get last name
    string getLast()
    {
        return lastName;
    }
}; // end class Person

class DataArray
{
private:
    Person* a;           // reference to array
    int nElems;
    int size;
    int index;           // number of data items
public:

    DataArray(int max)    // constructor
    {
        a = new Person[max];
        size = max;       // create the array  nElems = 0;
        nElems = 0;
        index = -1;       // no items yet
    }

    void find(string searchName)
    {
        Person obj;
        // find specified value
        int j; for (j = 0; j < nElems; j++)           // for each element,
            if (a[j].getLast() == (searchName))      // found item?
                break;                               // exit loop before end
        if (j == nElems)                             // gone to end?
            cout << "yes, can't find it " << endl;    //
        else
        {
            cout << "yes, found " << endl;
            a[j].displayPerson();                     // no, found it
        }
    }
};
```

```
    }
}
// end find()
//-----
// put person into array public
void insert(string last, string first, int age)
{
    if (nElems <= size - 1)
    {
        index++;
        a[index].setFirstName(first);
        a[index].setLastName(last);
        a[index].setAge(age);
        nElems++;
        // displayA();
    }
    else
    {
        cout << " Array Full" << endl;
    }
    // increment size
}
//----- public
bool deletePerson(string searchName)
{
    // delete person from array
    int j; for (j = 0; j < nElems; j++) // look for it
        if (a[j].getLast() == (searchName))
            break;
    if (j == nElems) // can't find it
        return false;
    else // found it
    {
        for (int k = j; k < nElems; k++) // shift down
            a[k] = a[k + 1];
        nElems--;
        index--; // decrement size
        return true;
    }
}
// end delete()
//----- public
void displayA() // displays array contents
{
    for (int j = 0; j < nElems; j++) // for each element,
        a[j].displayPerson(); // display it
}
//-----
}; // end class ClassDataArray
//////////////////////////////////// class
int main()
{
    int maxSize = 100; // array size
    DataArray arr(maxSize); // reference to array
    // arr = new ClassDataArray(maxSize); //
    create the array
    // insert 10 items
    arr.insert("Evans", "Patty", 24);
}
```

```
arr.insert("Smith", "Lorraine", 37);
arr.insert("Yee", "Tom", 43);
arr.insert("Adams", "Henry", 63);
arr.insert("Hashimoto", "Sato", 21);
arr.insert("Stimson", "Henry", 29);
arr.insert("Velasquez", "Jose", 72);
arr.insert("Lamarque", "Henry", 54);
//arr.insert("Vang", "Minh", 22);
//arr.insert("Creswell", "Lucinda", 18);
arr.displayA(); // display items
string searchKey = "Stimson"; // search for item
arr.find(searchKey);

cout << "Deleting Smith, Yee, and Creswell";
arr.deletePerson("Smith"); // delete 3 items arr.delete("Yee");
arr.delete("Creswell"); arr.displayA(); // display items again
} // end main()
```

## Pointers

Pointers are the most powerful feature of C and C++. These are used to create and manipulate data structures such as linked lists, queues, stacks, trees etc. The virtual functions also require the use of pointers. These are used in advanced programming techniques. To understand the use of pointers, the knowledge of memory locations, memory addresses and storage of variables in memory is required.

### Memory addresses & Variables

Computer memory is divided into various locations. Each location consists of 1 byte. Each byte has a unique address.

When a program is executed, it is loaded into the memory from the disk. It occupies a certain range of these memory locations. Similarly, each variable defined in the program occupies certain memory locations. For example, an int type variable occupies two bytes and float type variable occupies four bytes.

When a variable is created in the memory, three properties are associated with it. These are:

- Type of the variable
- Name of the variable
- Memory address assigned to the variable.

For example, an integer type variable xyz is declared as shown below

```
int xyz = 6760;
```

**int** represents the data type of the variable.

**xyz** represents the name of the variable.

xyz(1011)

6760
------

When variable is declared, a memory location is assigned to it. Suppose, the memory address assigned to the above variable xyz is 1011. The attribute or properties of this variable can be shown as below:

int

The box indicates the storage location in the memory for the variable xyz. The value of the variable is accessed by referencing its name. Thus, to print the contents of variable xyz on the computer screen, the statement is written as:

```
cout<<xyz
```

The memory address where the contents of a specific variable are stored can also be accessed. The **address operator (&)** is used with the variable name to access its memory address. The address operator (&) is used before the variable name.

For example, to print the memory address of the variable xyz, the statement is written as:

```
cout<<&xyz
```

The memory address is printed in hexadecimal format.

### Pointer Variables

The variables that is used to hold the memory address of another variable is called a pointer variable or simply pointer.

The data type of the variable (whose address a pointer is to hold) and the pointer variable must be the same. A pointer variable is declared by placing an asterisk (\*) after data type or before the variable name in the data type statement.

For example, if a pointer variable “**p**” is to hold memory address of an integer variable, it is declared as:

```
int* p;
```

Similarly, if a pointer variable “**rep**” is to hold memory address of a floating-point variable, it is declared as:

```
float* rep;
```

The above statements indicate that both “**p**” and “**rep**” variable are pointer variables and they can hold memory address of integer and floating-point variable respectively. Although the asterisk is written after the data type, is usually more convenient to place the asterisk before the pointer variable. i.e. **float \*rep;**

```
#include <iostream>
using namespace std;
int main() {

int a, b;
int *x, *y;
a = 33;
```

```
b = 66;
x = &a;
y = &b;

cout<<"Memory address of variable a= "<<x<<endl;
cout<<"Memory address of variable b= "<<y<<endl;
cout<<"Memory address of pointer x= "<<&x<<endl;
cout<<"Memory address of pointer y= "<<&y<<endl;
cout<<"Memory address of variable x= "<<*&x<<endl;
cout<<"Memory address of variable y= "<<*&y<<endl;
return 0;
}
```

### Output:

Memory address of variable a= 0x7bfe0c

Memory address of variable b= 0x7bfe08

A pointer variable can also be used to access data of memory location to which it points. In the above program, **x** and **y** are two pointer variables. They hold memory addresses of variables **a** and **b**. To access the contents of the memory addresses of a pointer variable, an asterisk (\*) is used before the pointer variable.

For example, to access the contents of **a** and **b** through pointer variable **x** and **y**, an asterisk is used before the pointer variable. For example,

```
cout<<"Value in memory address x = "<<*&x<<endl;
cout<<"Value in memory address y = "<<*&y<<endl;
```

### Program

Write a program to assign a value to a variable using its pointer variable. Print out the value using the variable name and also print out the memory address of the variable using pointer variable.

```
#include <iostream>
using namespace std;
int main () {
int *p;
int a;
p=&a;
cout<<"Enter data value? ";
cin>>*p;
cout<<"Value of variable  = "<<a<<endl;
cout<<"Memory Address of variable= "<<p<<endl;
return 0;
```

```
}
```

## Output:

Enter data value? 44

Value of variable =44

Memory Address of variable= 0x7bfe14

## The “void” Type Pointers

Usually type of variable and type of pointer variable that holds memory address of the variable must be the same. But the “**void**” type pointer variable can hold memory address of variables of any type. A void type pointer is declared by using keyword “**void**”. The asterisk is used before pointer variable name.

Syntax for declaring void type pointer variable is:

```
void *p;
```

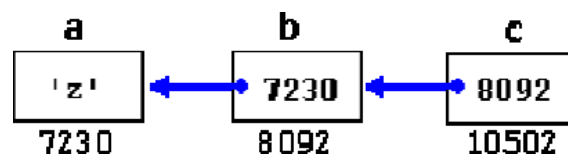
The pointer variable “**p**” can hold the memory address of variables of any data type.

## Pointers to Pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (\*) for each level of reference in their declarations:

```
char a;  
char *b;  
char  
**c;  
a = 'z';  
b = &a;  
c = &b;
```

This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could be represented as:



The value of each variable is written inside each cell; under the cells are their respective addresses in memory. The new thing in this example is variable c, which can be used in three different levels of indirection, each one of them would

correspond to a different value:

```
#include <iostream>
using namespace std;
int main ()
{
    int a;
    int *b;
    int **c;
    int ***d;
    a = 7;
    b = &a;
    c = &b;
    d = &c;

    cout<<"The Address of the Vairiable a is: "<<b<<endl;
    cout<<"The Address of the Vairiable b is: "<<c<<endl;
    cout<<"The Address of the Vairiable c is: "<<d<<endl;
    return 0;
}
```

**Output:**

The Address of the Vairiable a is: 0x7bfe14  
The Address of the Vairiable b is: 0x7bfe08  
The Address of the Vairiable d is: 0x7bfe00

**The Reference (Address Of) Operator (&)**

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be - Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases we may be interested in knowing the address where our variable is being stored during runtime in order to operate with relative positions to it.

The address that locates a variable within memory is what we call a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example:

```
ted = &andy;
```



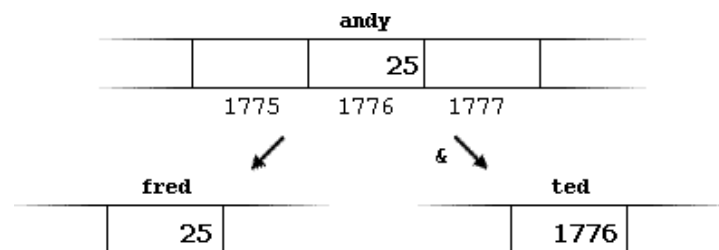
This would assign to ted the address of variable **andy**, since when preceding the name of the variable **andy** with the reference operator (&) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

From now on we are going to assume that andy is placed during runtime in the memory address 1776. This number (1776) is just an arbitrary assumption we are inventing right now in order to help clarify some concepts in this tutorial, but in reality, we cannot know before runtime the real value the address of a variable will have in memory.

Consider the following code fragment:

```
andy = 25;  
fred = andy;  
ted = &andy;
```

The values contained in each variable after the execution of this, are shown in the following diagram:



First, we have assigned the value 25 to andy (a variable whose address in memory we have assumed to be 1776). The second statement copied to fred the content of variable andy (which is 25). This is a standard assignment operation, as we have done so many times before.

Finally, the third statement copies to ted not the value contained in andy but a reference to it (i.e., its address, which we have assumed to be 1776). The reason is that in this third assignment operation we have preceded the identifier andy with the reference operator (&), so we were no longer referring to the value of andy but to its reference (its address in memory).

The variable that stores the reference to another variable (like ted in the previous example) is what we call a pointer. Pointers are a very powerful feature of the C++ language that has many uses in advanced programming. Farther ahead, we will see how this type of variable is used and declared.

## The Dereferencing Operator (\*)

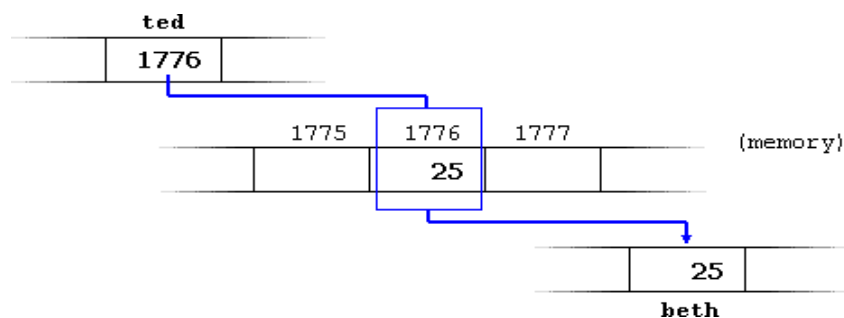
We have just seen that a variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store.

Using a pointer, we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (\*), which acts as dereference operator and that can be literally translated to "value pointed by".

Therefore, following with the values of the previous example, if we write:

```
beth = *ted;
```

(that we could read as: "beth equal to value pointed by ted") beth would take the value 25, since ted is 1776, and the value pointed by 1776 is 25.



You must clearly differentiate that the expression `ted` refers to the value 1776, while `*ted` (with an asterisk \* preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the dereference operator (I have included an explanatory commentary of how each of these two expressions could be read):

```
beth = ted; // beth equal to ted ( 1776 )
beth = *ted; // beth equal to value pointed by ted ( 25 )
```

Notice the difference between the **reference** and **dereference** operators:

- **&** is the **reference operator** and can be read as "**address of**"
  - **\*** is the **dereference operator** and can be read as "**value pointed by**"
- Thus, they have complementary (or opposite) meanings. A variable referenced with **&** can be dereferenced with **\***.

Earlier we performed the following two assignment operations:

```
andy = 25;
ted = &andy;
```

Right after these two statements, all of the following expressions would give true as result:

```
andy == 25
&andy == 1776
ted == 1776
*ted == 25
```

The first expression is quite clear considering that the assignment operation performed on `andy` was `andy=25`. The second one uses the reference operator (`&`), which returns the address of variable `andy`, which we assumed it to have a value of 1776. The third one is somewhat obvious since the second expression was true and the assignment operation performed on `ted` was `ted=&andy`. The fourth expression uses the dereference operator (`*`) that, as we have just seen, can be read as "value pointed by", and the value pointed by `ted` is indeed 25.

So, after all that, you may also infer that for as long as the address pointed by `ted` remains unchanged the following expression will also be true:

```
*ted == andy
```

## Pointers and Arrays

There is a close relationship between pointers and arrays. In Advanced programming, arrays are accessed using pointers.

Arrays consist of consecutive locations in the computer memory. To access an array, the memory location of the first element of the array is accessed using the pointer variable. The pointer is then incremented to access other elements of the array. The pointer is increased in the value according to the size of the elements of the array.

When an array is declared, the array name points to the starting address of the array. For example, consider the following example.

```
int x[5];
int *p;
```

The array "**x**" is of type **int** and "**p**" is a pointer variable of type **int**.

To store the starting address of array "**x**" (or the address of first element), the following statement is used.

```
p = x;
```

The address operator (`&`) is not used when only the array name is used. If an element of the array is used, the `&` operator is used. For example, if memory address of first element of the array is to be assigned to a pointer, the statement is written as:

```
p = &x[0];
```

when integer value 1 is added to or subtracted from the pointer variable “p”, the content of pointer variable “p” is incremented or decremented by (1 x size of the object or element), it is incremented by 1 and multiplied with the size of the object or element to which the pointer refers.

For example, the memory size of various data types is shown below:

- The array of **int** type has its object or element size of **2 bytes**. It is **4 bytes** in Xenix System.
- The array of type **float** has its object or element size of **4 bytes**.
- The array of type **double** has its object or element size of **8 bytes**.
- The array of type **char** has its object or element size of **1 byte**.

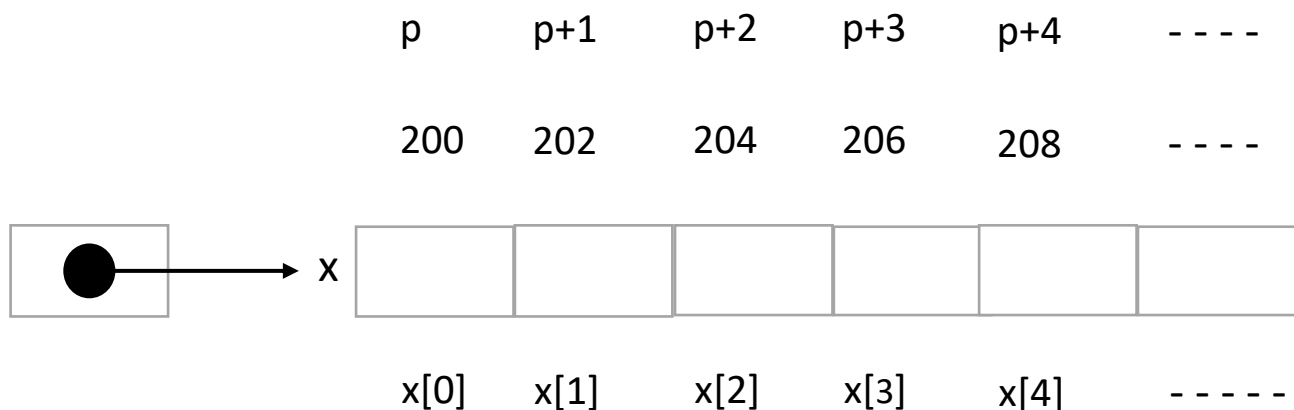
Suppose the location of first element in memory is 200. i.e., the value of pointer variable “p” is 200, and it refers to an integer variable.

When the following statement is executed,

`p=p+1;`

the newly value of “p” will be  $200+(1*2)=202$ . All elements of an array can be accessed by using this technique.

The logical diagram of an integer type array “x” and pointer variable “p” is that refers to the elements of the array “x” is given below:



## Program

Write a Program to input data into an array and then to print on the computer screen by using pointer notation.

```
using namespace std;
```

```
#include <iostream>
```

```
int main()
{
    int arr[5], *pp, i;
    pp = arr;
    //int size = arr / arr[0];
    cout << "Enter Values into an array(index): " << endl;
    for (int i = 0; i <= 4; i++)
    {
        cin >> arr[i];
    }

    pp = arr;
    cout << "Enter Values into an array(pinter): " << endl;
    for (int i = 0; i <= 4; i++)
    {
        cin >> *pp;
        pp++;
    }
    cout << "Values from array Using Pointer notation: " << endl;
    pp = arr;
    for (int i = 0; i <= 4; i++)
    {
        cout << *pp << "\t";
        pp++;
    }
    cout << "\nUsing While Loop: " << endl;
    int x = 5;
    pp = arr;
    while (x!=0) {
        cout << *pp++ << "\t";
        x--;
    }
    return 0;
}
```

```
}
```

## Output:

Enter Values into an array:

4  
5  
3  
55  
88

Values from array Using Pointer notation:

4   5   3   55   88

## Passing Pointers as Arguments to Functions

The pointer variables can also be passed to functions as arguments. When pointer variable is passed to a function, the address of the variable is passed to the function. Thus, a variable is passed to a function not by its value but by its reference.

```
#include <iostream>
using namespace std;
void temp(int *, int *);
int main () {
    int a, b;
    a=10;
    b=20;
    temp(&a, &b); // function calling
    cout<<"Value of a= "<<a<<endl;
    cout<<"Value of b= "<<b<<endl;
    cout<<"OK"<<endl;
    return 0;
}
void temp(int *x, int *y)
{
    *x = *x+100;
    *y = *y+100;
}
```

**Output:** Value of a= 110

Value of b= 120

OK

## Program Explanation

In the above program, the function “**temp**” has two parameters which are pointers and are of **int** type. When the function “**temp**” is called, the addresses of variables “**a**” and “**b**” are passed to the function.

In the function, a value 100 is added to both variables “**a**” and “**b**” through their pointers. That is, the previous values of variables “**a**” and “**b**” are increased by 100. When the control returns to the program, the values of variable **a** is 110 and that of variable **b** is 120.

## Passing Pointers to a Function

```
#include <string>
#include <iostream>
using namespace std;
void abc(int *a)
{
    *a=*a * *a - *a;
}
int main()
{
    int x=5;
    int *p;
```

```
p=&x;  
abc(p); // calling function  
cout<<"Value of p is changed by the function passed as parameter.: "<<*p<<endl;  
}
```

**Output:**

Value of p is changed by the function passed as parameter.: 20

**Program**

Write a program to swap two values by passing pointers as arguments to the function.

```
#include <iostream>  
using namespace std;  
void swap(int*, int*); // Function prototype  
int main () {  
    int a, b;  
    cout<<"Enter 1st Value for a ? ";  
    cin>>a;  
    cout<<"Enter 2nd Value for b? ";  
    cin>>b;  
    swap(&a, &b); // Function Calling  
    cout<<"Values after exchange = "<<endl;  
    cout<<"Value of a = "<<a<<endl;  
    cout<<"Value of b = "<<b<<endl;  
    return 0;  
}  
void swap(int *x, int *y) // Function Definition  
{  
    int t;  
    t = *x;  
    *x = *y;  
    *y = t;  
}
```

**Output:**

Enter 1st Value for a? 3  
Enter 2nd Value for b? 7  
Values after exchange =  
Value of a= 7  
Value of b= 3

**Returning Pointers from Function**

```
using namespace std;  
  
#include <string>  
#include <iostream>  
using namespace std;  
  
int* abc(int &a)  
{
```

```
int *p;
p=&a;
*p = (*p + *p) * *p - *p**p;
return p;
}
int main()
{
int x=3;
int *p;
p=abc(x);
cout<<"Value of p is changed by the function returned.: "<<*p<<endl;
}
```

**Output:**

Value of p is changed by the function returned.: 9

## String in C++

String is a collection of characters. There are two types of strings commonly used in C++ programming language:

- Strings that are objects of string class (The Standard C++ Library string class).
- C-strings (C-style Strings).

### C-strings

In C programming, the collection of characters is stored in the form of arrays. This is also supported in C++ programming. Hence, it's called C-strings.

C-strings are arrays of type `char` terminated with null character, that is, `\0` (ASCII value of null character is 0).

#### How to define a C-string?

```
char str[ ] = "C++";
```

In the above code, `str` is a string and it holds 4 characters.

Although, `"C++"` has 3 character, the null character `\0` is added to the end of the string automatically.

```
cout<<str;           // will print C++
cout<<str[0];        // will print only "C"
```

#### Alternative ways of defining a string

```
char str[4] = "C++";
char str[] = {'C','+','+','\0'};
char str[4] = {'C','+','+','\0'};
```

Like arrays, it is not necessary to use all the space allocated for the string. For example:



```
char str[100] = "C++";
```

**Example 1: C++ String to read a word**  
**C++ program to display a string entered by user.**

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

int main()
{
    char str[100];

    cout << "Enter a string: ";
    cin >> str;
    cout << "You entered: " << str << endl;

    cout << "\nEnter another string: ";
    cin >> str;
    cout << "You entered: " << str << endl;

    return 0;
}
```

### Output

```
Enter a string: C++
You entered: C++

Enter another string: Programming is fun.
You entered: Programming
```

Notice that, in the second example only "Programming" is displayed instead of "Programming is fun".

This is because the extraction operator >> works as `scanf()` in C and considers a space " " has a **terminating character**.

**Example 2: C++ String to read a line of text**

```
#include <iostream>
```

```
using namespace std;

int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin.get(str, 100);

    cout << "You entered: " << str << endl;
    return 0;
}
```

## Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

To read the text containing blank space, `cin.get` function can be used. This function takes two arguments.

First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.

In the above program, `str` is the name of the string and `100` is the maximum size of the array.

### string Object

In C++, you can also create a string object for holding strings.

Unlike using char arrays, string objects has no fixed length, and can be extended as per your requirement.

### Example 3: C++ string using string data type

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

int main()
{
    // Declaring a string object
    string str;
    cout << "Enter a string: ";
    getline(cin, str);
}
```

```
cout << "You entered: " << str << endl;  
return 0;  
}
```

## Output

Enter a string: Programming is fun.  
You entered: Programming is fun.

In this program, a string `str` is declared. Then the string is asked from the user. Instead of using `cin>>` or `cin.get()` function, you can get the entered line of text using `getline()`.

`getline()` function takes the input stream as the first parameter which is `cin` and `str` as the location of the line to be stored.

### Passing String to a Function

Strings are passed to a function in a similar way arrays are passed to a function.

```
#include <iostream>  
using namespace std;  
  
void display(char *);  
void display(string);  
  
int main()  
{  
    string str1;  
    char str[100];  
  
    cout << "Enter a string: ";  
    getline(cin, str1);  
  
    cout << "Enter another string: ";  
    cin.get(str, 100, '\n');  
  
    display(str1);  
    display(str);  
    return 0;  
}
```

```
void display(char s[])
{
    cout << "Entered char array is: " << s << endl;
}

void display(string s)
{
    cout << "Entered string is: " << s << endl;
}
```

### Output

```
Enter a string: Programming is fun.
Enter another string: Really?
Entered string is: Programming is fun.
Entered char array is: Really?
```

In the above program, two strings are asked to enter. These are stored in `str` and `str1` respectively, where `str` is a `char` array and `str1` is a `string` object.

Then, we have two functions `display()` that outputs the string onto the string.

The only difference between the two functions is the parameter. The first `display()` function takes char array as a parameter, while the second takes string as a parameter. **This process is known as function overloading.**

### Pointers and Strings

A String is a sequence of characters. A string type variable is declared in the same manner as an array type variable is declared. This is because a string is an array of characters type variables.

Since a string is like an array, pointer variables can also be used to access it. For example:

```
char st1[] = "Pakistan";
char *st2 = "Pakistan"
```

In the above statements, two string variables “`st1`” and “`st2`” are declared. The variable “`st1`” is an array of character type. The variable “`st2`” is a pointer also of character type. These two variables are equivalent. The difference between string variables “`st1`” and “`st2`” is that:

- string variable “`st1`” represents a pointer constant. Since a string is an array of character type, the “`st1`” is the name of the array. Also, the name of the array

represents its address its which is a constant. Therefore, **st1** represents a pointer constant.

- string variable “**st2**” represents a pointer variable.

In the following program example, a string is printed by printing its character on by one.

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

void ppp(char *); // Function Prototype

int main () {
char st[] = "Pakistan";

ppp(st); // calling function
cout<<"OK";
}
// function definition
void ppp(char *sss)
{
    //loop iterating string using pointer
    while (*sss != '\0')
    {
        cout<<*sss<<endl;
        *sss++;
    }
}
```

**Output:**

```
P
a
k
i
s
t
a
n
OK
```

## LAB-02

### OBJECTIVE

After this lab students will be able to understand

- Insertion in Arrays
- Deletion in Arrays
- Traversing
- Binary Search

### PRE-LAB READING ASSIGNMENT

NONE

### Tools/ Apparatus:

Dev C++, VS Code

### LAB RELATED CONTENT

#### Arrays:

The array is the most commonly used data storage structure; it's built into most programming languages. Because arrays are so well known, they offer a convenient jumping off place for introducing data structures and for seeing how object-oriented programming and data structures relate to one another. Access File Extensions

---

Experiment 01: Write a program to perform following operations in arrays (Insert an element, Find an element, delete an element)

---

```
#include<iostream>
#include <cstring>
#include <stdlib>
using namespace std;
class Person
{
    private:
    string lastName;
    string firstName;
    int age;
    //-----
    public:

    Person()
    {
    }
    //-----
    void displayPerson()
    {
```

```
    cout << "    First name:" << firstName << endl;
    cout << "    Last name:" << lastName << endl;
    cout << "    Age:" << age << endl;
}

void setFirstName(string firstName)
{
    this->firstName = firstName;
}
void setLastName(string lastName)
{
    this->lastName = lastName;
}
void setAge(int age)
{
    this->age = age;
}

//-----
// get last name
string getLast()
{
    return lastName;
}
}; // end class Person

class DataArray
{
private:
    Person* a;           // reference to array
    int nElems;
    int size;
    int index;           // number of data items
public:

    DataArray(int max)    // constructor
    {
        a = new Person[max];
        size = max;           // create the array  nElems = 0;
        nElems = 0;
        index = -1;           // no items yet
    }

    void find(string searchName)
    {
        Person obj;
        // find specified value
        int j; for (j = 0; j < nElems; j++)           // for each element,
            if (a[j].getLast() == (searchName)) // found item?
                break;                               // exit loop before end
        if (j == nElems)                             // gone to end?
            cout << "yes, can't find it " << endl;    //
        else
        {
            cout << "yes, found " << endl;
            a[j].displayPerson();                     // no, found it
        }
    }
}
```

```

}
// end find()
//-----
// put person into array public
void insert(string last, string first, int age)
{
    if (nElems <= size - 1)
    {
        index++;
        a[index].setFirstName(first);
        a[index].setLastName(last);
        a[index].setAge(age);
        nElems++;
        // displayA();
    }
    else
    {
        cout << " Array Full" << endl;
        // increment size
    }
}
//----- public
bool deletePerson(string searchName)
{
    // delete person from array
    int j; for (j = 0; j < nElems; j++) // look for it
        if (a[j].getLast() == (searchName))
            break;
    if (j == nElems) // can't find it
        return false;
    else // found it
    {
        for (int k = j; k < nElems; k++) // shift down
            a[k] = a[k + 1];
        nElems--;
        index--; // decrement size
        return true;
    }
}
// end delete()
//----- public
void displayA() // displays array contents
{
    for (int j = 0; j < nElems; j++) // for each element,
        a[j].displayPerson(); // display it
}
//-----
}; // end class ClassDataArray
//////////////////////////////////// class
int main()
{
    int maxSize = 100; // array size
    DataArray arr(maxSize); // reference to array
    // arr = new ClassDataArray(maxSize); //
create the array
    // insert 10 items

```



```
arr.insert("Evans", "Patty", 24);
arr.insert("Smith", "Lorraine", 37);
arr.insert("Yee", "Tom", 43);
arr.insert("Adams", "Henry", 63);
arr.insert("Hashimoto", "Sato", 21);
arr.insert("Stimson", "Henry", 29);
arr.insert("Velasquez", "Jose", 72);
arr.insert("Lamarque", "Henry", 54);
//arr.insert("Vang", "Minh", 22);
//arr.insert("Creswell", "Lucinda", 18);
arr.displayA(); // display items
string searchKey = "Stimson"; // search for item
arr.find(searchKey);

cout << "Deleting Smith, Yee, and Creswell";
arr.deletePerson("Smith"); // delete 3 items arr.delete("Yee");
arr.delete("Creswell"); arr.displayA(); // display items again
} // end main()
```

## Binary Search

Binary search works on sorted arrays. Binary search begins by comparing the middle element of the array with the target value. If the target value matches the middle element, its position in the array is returned. If the target value is less than or greater than the middle element, the search continues in the lower or upper half of the array, respectively, eliminating the other half from consideration.

Ordered arrays are therefore useful in situations in which searches are frequent, but insertions and deletions are not. An ordered array might be appropriate for a database of company employees, for example. Hiring new employees and laying off existing ones would probably be infrequent occurrences compared with accessing an existing employee's record for information, or updating it to reflect changes in salary, address, and so on.

---

Experiment 02: Write a program to find an element using binary search.

---

```
#include
using namespace std;
int binarySearch(int arr[], int p, int r, int num)
{
    if (p <= r)
    {
        int mid = (p + r) / 2;
        if (arr[mid] == num)
            return mid;
        if (arr[mid] > num)
```

```
        return binarySearch(arr, p, mid - 1, num);
    if (arr[mid] < num)
        return binarySearch(arr, mid + 1, r, num);
    }
    return -1;
}
int main(void)
{
    int arr[] = { 1, 3, 7, 15, 18, 20, 25, 33, 36, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int num;
    cout << "Enter the number to search:
";
    cin >> num;
    int index = binarySearch(arr, 0, n - 1, num);
    if (index == -1)
    {
        cout << num << " is not present in the array";
    }
    else
    {
        cout << num << " is present at index " << index << " in the array";
    }
    return 0;
}
```

## LAB-03

### OBJECTIVE

After completing this experiment, you will be able to:

- Pointers
- DMA
- 2D Arrays ,3 D Arrays
- C++ Dynamic Memory Allocation Example Arrays
- C++ Dynamic Memory Allocation Example Multi-Dimensional Arrays

### PRE-LAB READING ASSIGNMENT

Tools/ Apparatus:

Dev C++, Visual Studio

### LAB RELATED CONTENT

#### C++ Dynamic Memory

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts –

- **The stack** – All variables declared inside the function will take up memory from the stack.
- **The heap** – This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory that was previously allocated by new operator.

#### **new and delete operators in C++ for dynamic memory**

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack (Refer Memory Layout C Programs for details).

What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except variable length arrays.
- The most important use is flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need and whenever we don't need anymore. There are many cases where this flexibility helps. Examples of such cases are Linked List, Tree, etc.

How is it different from memory allocated to normal variables?

For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and deallocated. For dynamically allocated memory like "int \*p = new int[10]", it is programmers responsibility to deallocate memory when no longer needed. If programmer doesn't deallocate memory, it causes memory leak (memory is not deallocated until program terminates).

How is memory allocated/deallocated in C++?

C uses malloc() and calloc() function to allocate memory dynamically at run time and uses free() function to free dynamically allocated memory. C++ supports these functions and also has two operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.

This is all about new and delete operators.

### **new operator**

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

- Syntax to use new operator: To allocate memory of any data type, the syntax is:

**pointer-variable = new data-type;**

- Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example:

```
// Then request memory for the variable
int* p = NULL;
p = new int;
```

OR

```
// Combine declaration of pointer
// and their assignment
int* p = new int;
```

**Initialize memory:** We can also initialize the memory for built-in data types using new operator. For custom data types a constructor is required (with the data-type as input) for initializing the value. Here's an example for the initialization of both data types :  
**pointer-variable = new data-type(value);**

**Example:**

```
int* p = new int(25);
float* q = new float(75.25);

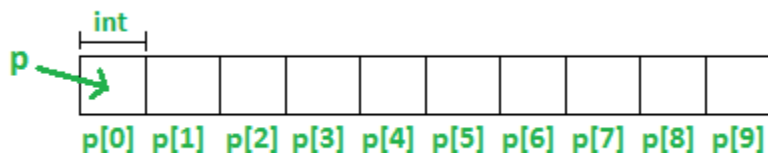
// Custom data type
struct cust
{
    int p;
    cust(int q) : p(q) { }
};

cust* var1 = new cust;    // Works fine, doesn't require
                          // constructor
OR
cust* var1 = new cust();  // Works fine, doesn't require
                          // constructor

cust* var = new cust(25)  // Notice error if you comment
                          // this line
```

**int \*p = new int[10]**

- Dynamically allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer). p[0] refers to first element, p[1] refers to second element and so on.



Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, normal arrays are deallocated by compiler (If array is local, then deallocated when function returns or completes). However, dynamically allocated arrays always remain there until either they are

deallocated by programmer or program terminates.

### What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type `std::bad_alloc`, unless “nothrow” is used with the new operator, in which case it returns a NULL pointer (scroll to section “Exception handling of new operator” in this article). Therefore, it may be good idea

```
int* p = new (nothrow) int;
if (!p)
{
    cout << "Memory allocation failed\n";
}
```

Since it is programmer’s responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

Syntax:

// Release memory pointed by pointer-variable

delete pointer-variable;

Here, pointer-variable is the pointer that points to the data object created by new.

Examples:

**delete p;**

**delete q;**

To free the dynamically allocated array pointed by pointer-variable, use following form of delete:

```
// Release block of memory
// pointed by pointer-variable
Delete [] pointer-variable;
```

Example:

```
Example:
// It will free the entire array
// pointed by p.
Delete [] p;
```

```
// C++ program to illustrate dynamic allocation
// and deallocation of memory using new and delete
#include <iostream>
using namespace std;
int main()
{
    // Pointer initialization to null
    int* p = NULL;

    // Request memory for the variable
    // using new operator
    p = new (nothrow) int;
    if (!p)
        cout << "allocation of memory failed\n";
    else
    {
        // Store value at allocated address
        *p = 29;
        cout << "Value of p: " << *p << endl;
    }

    // Request block of memory
    // using new operator
    float* r = new float(75.25);

    cout << "Value of r: " << *r << endl;
    // Request block of memory of size n
    int n = 5;
    int* q = new (nothrow) int[n];

    if (!q)
        cout << "allocation of memory failed\n";
    else
    {
        for (int i = 0; i < n; i++)
            q[i] = i + 1;
        cout << "Value store in block of memory using index: ";
        for (int i = 0; i < n; i++)
            cout << q[i] << " ";
        cout << endl << "Value store in block of memory using ptr: ";

        for (int i = 0; i < n; i++)
        {
            cout << *q << " ";
            q = q + 1;
        }
    }

    // freed the allocated memory
    delete p;
```

```
#include <iostream>
using namespace std;
class Box
{
    public:
        Box()
        {
            cout << "Constructor called!" << endl;
        }
        ~Box()
        {
            cout << "Destructor called!" << endl;
        }
};
int main()
{
    Box* myBoxArray = new Box[4];
    delete[] myBoxArray; // Delete array

    return 0;
}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times. If we compile and run above code, this would produce the following

result –

```
Constructor called!
Constructor called!
Constructor called!
Constructor called!
Destructor called!
Destructor called!
Destructor called!
Destructor called!
```



## C++ new and delete Operator for Arrays

```
// C++ Program to store GPA of n number of students and display it
// where n is the number of students entered by the user
#include <iostream>
using namespace std;
int main()
{
    int num;
    cout << "Enter total number of students: ";
    cin >> num;
    float* ptr;
    // memory allocation of num number of floats
    ptr = new float[num];

    cout << "Enter GPA of students." << endl;
    for (int i = 0; i < num; ++i)
    {
        cout << "Student" << i + 1 << ": ";
        cin >> *(ptr + i);
    }
    cout << "\nDisplaying GPA of students." << endl;
    for (int i = 0; i < num; ++i)
    {
        cout << "Student" << i + 1 << " : " << *(ptr + i) << endl;
    }
    // ptr memory is released
    delete[] ptr;
    return 0;
}
```

```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9
Displaying GPA of students.
Student1 :3.6
Student2 :3.1
Student3 :3.9
Student4 :2.9
```

### C++ new and delete Operator for Objects

```
#include <iostream>
using namespace std;
class Student
{
    int age;
public:
    // constructor initializes age to 12
    Student() : age(12) { }

    void getAge()
    {
        cout << "Age = " << age << endl;
    }
};

int main()
{
    // dynamically declare Student object
    Student* ptr = new Student();
    // call getAge() function
    ptr->getAge();
    // ptr memory is released
    delete ptr;
    return 0;
}
```

### Output

Age = 12

In this program, we have created a Student class that has a private variable age. We have initialized age to 12 in the default constructor Student() and print its value with the function getAge().

In main(), we have created a Student object using the new operator and use the pointer ptr to point to its address.

The moment the object is created, the Student() constructor initializes age to 12.

We then call the getAge() function using the code:

## C++ Dynamic Memory Allocation Example Arrays

```
/* C++ Dynamic Memory Allocation Example Program */
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
int* rollno;    // declares an integer pointer
float* marks;   // declares a float pointer

int main()
{
    int size, i;
    cout << "How many elements for the array ? ";
    cin >> size;
    rollno = new (nothrow)int[size];    // dynamically allocate rollno array
    marks = new (nothrow)float[size];    // dynamically allocate marks array
    // first check, whether the memory is available or not
    if ((!rollno) || (!marks))    // if rollno or marks is null pointer
    {
        cout << "Out of Memory...!!..Aborting...!!\n";
        cout << "Press any key to exit..";
        // getch();
        exit(1);
    }
    int* roll; float *mark;
    roll = rollno;
    mark = marks;
    // read values in the array elements
    for (i = 0; i < size; i++)
    {
        cout << "Enter rollno and marks for student "
              << (i + 1) << "\n";
        //cin >> rollno[i] >> marks[i];
        cin >> *roll >> *mark;
        roll++;
        mark++;
    }

    // now display the array contents
    cout << "\nRollNo\t\tMarks\n";
    roll = rollno;
    mark = marks;
    for (i = 0; i < size; i++)
    {
        // cout << rollno[i] << "\t\t" << marks[i] << "\n";
        cout << *roll++ << "\t\t" << *mark++ << "\n";
    }
    delete[] rollno;    // deallocating rollno array
    delete[] marks;    // deallocating marks array
}
```

## C++ Dynamic Memory Allocation Example Multi-Dimensional Arrays

```
/* C++ Dynamic Memory Allocation Example Program
```

- \* This is the same program as above, but this
- \* program uses two-dimensional array to demonstrates
- \* dynamic memory allocation in C++. This C++ program
- \* also displays the rowsum and the colsum of the array \*/

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
int main()
{

    int* val, *rows, *cols;
    int maxr, maxc, i, j;
    cout << "Enter the dimension of the array (row col): ";
    cin >> maxr >> maxc;

    val = new int[maxr * maxc];
    rows = new int[maxr];
    cols = new int[maxc];

    for (i = 0; i < maxr; i++)
    {
        cout << "\nEnter elements for row " << i + 1 << " : ";
        rows[i] = 0;
        for (j = 0; j < maxc; j++)
        {
            cin >> val[i * maxc + j];

            rows[i] = rows[i] + val[i * maxc + j];
        }
    }

    for (j = 0; j < maxc; j++)
    {
        cols[j] = 0;
        for (i = 0; i < maxr; i++)
        {
            cols[j] = cols[j] + val[i * maxc + j];
        }
    }
}
```

```
cout << "\nThe given array in 2 x 2 dimensional (alongwith
rowsum and colsum) is :\n";
for (i = 0; i < maxr; i++)
{
    for (j = 0; j < maxc; j++)
    {
        cout << val[i * maxc + j] << "\t";
    }
    cout << rows[i] << "\n";
}

for (j = 0; j < maxc; j++)
{
    cout << cols[j] << "\t";
}
cout << "\n";
delete[] val;
delete[] rows;
delete[] cols;
getch();
return 0;
}
```

```
}  
D:\Anjum FAST\DS Manuals\lab 2\DMA3.exe  
Enter the dimension of the array (row col): 4  
4  
Enter elements for row 1 : 1  
2  
3  
4  
Enter elements for row 2 : 1  
2  
3  
4  
Enter elements for row 3 : 1  
2  
3  
4  
Enter elements for row 4 : 1  
2  
3  
4  
The given array in 4 x 4 dimensional (alongwith rowsum and colsum) is :  
1      2      3      4      10  
1      2      3      4      10  
1      2      3      4      10  
1      2      3      4      10  
4      8      12     16
```

## Dynamic memory allocation in C++ for 2D and 3D array

This post will discuss dynamic memory allocation in C++ for multidimensional arrays.

### 1. Single Dimensional Array

```
#include <iostream>  
#define N 10  
  
// Dynamically allocate memory for 1D Array in C++  
int main()  
{  
    // dynamically allocate memory of size `N`  
    int* A = new int[N];
```

```
// assign values to the allocated memory
for (int i = 0; i < N; i++)
{
    A[i] = i + 1;
}

// print the 1D array
for (int i = 0; i < N; i++)
{
    std::cout << A[i] << " ";    // or *(A + i)
}

// deallocate memory
delete[] A;

return 0;
}
```

## 2. 2-Dimensional Array

### 1. Using Single Pointer

In this approach, we simply allocate one large block of memory of size  $M \times N$  dynamically and assign it to the pointer. Then we can use pointer arithmetic to index the 2D array.

```
#include <iostream>
```

```
// `M Ã– N` matrix
```

```
#define M 4
```

```
#define N 5
```

```
// Dynamically allocate memory for 2D Array in C++
```

```
int main()
```

```
{
```

```
    // dynamically allocate memory of size `M Ã– N`
```

```
    int* A = new int[M * N];
```

```
    // assign values to the allocated memory
```

```
    for (int i = 0; i < M; i++)
```

```
    {
```

```
        for (int j = 0; j < N; j++)
```

```
        {
```

```
            *(A + i * N + j) = rand() % 100;
```

```
        }
```

```
    }
```

```
    // print the 2D array
```

```
    for (int i = 0; i < M; i++)
```

```
{
    for (int j = 0; j < N; j++)
    {
        std::cout << *(A + i * N + j) << " ";    // or (A +
i*N)[j])
    }
    std::cout << std::endl;
}

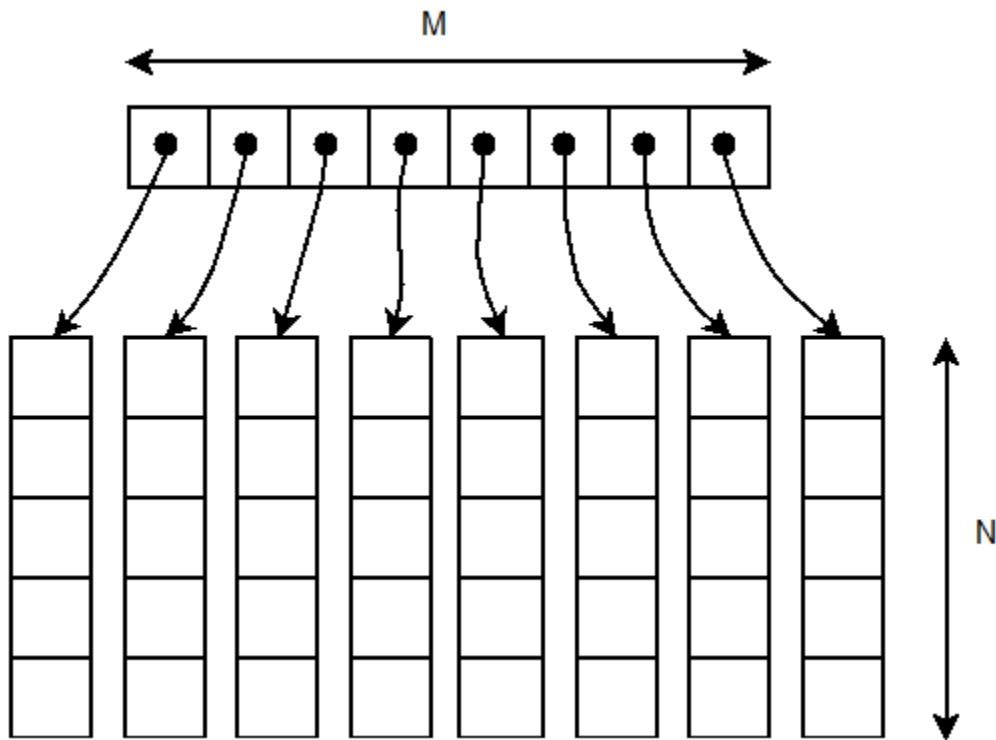
// deallocate memory
delete[] A;

return 0;
}
```

## 2. Using array of Pointers

We can dynamically create an array of pointers of size `M` and then dynamically allocate memory of size `N` for each row, as shown below:





```
#include <iostream>
```

```
// `M Ã– N` matrix
```

```
#define M 4
```

```
#define N 5
```

```
// Dynamic Memory Allocation in C++ for 2D Array
```

```
int main()
```

```
{
```

```
    // dynamically create an array of pointers of size `M`
```

```
    int** A = new int*[M];
```

```
    // dynamically allocate memory of size `N` for each row
```

```
    for (int i = 0; i < M; i++)
```

```
    {
```

```
        A[i] = new int[N];
```

```
    }
```

```
    // assign values to the allocated memory
```

```
    for (int i = 0; i < M; i++)
```

```
    {
```

```
        for (int j = 0; j < N; j++)
```

```
        {
```

```
        A[i][j] = rand() % 100;
    }
}

// print the 2D array
for (int i = 0; i < M; i++)
{
    for (int j = 0; j < N; j++)
    {
        std::cout << A[i][j] << " ";
    }
    std::cout << std::endl;
}

// deallocate memory using the delete[] operator
for (int i = 0; i < M; i++)
{
    delete[] A[i];
}
delete[] A;

return 0;
}
```

### 3. 3-Dimensional Array

#### 1. Using Single Pointer

As seen for the 2D array, we allocate memory of size  $X \times Y \times Z$  dynamically and assign it to a pointer. Then we use pointer arithmetic to index the 3D array.

```
#include <iostream>
// `X Ã– Y Ã– Z` matrix
#define X 2
#define Y 3
#define Z 4

// Dynamic Memory Allocation in C++ for 3D Array
int main()
{
    // dynamically allocate memory of size `X Ã– Y Ã– Z`
    int* A = new int[X * Y * Z];

    // assign values to the allocated memory
    for (int i = 0; i < X; i++)
    {
```

```
    for (int j = 0; j < Y; j++)
    {
        for (int k = 0; k < Z; k++)
        {
            *(A + i * Y * Z + j * Z + k) = rand() % 100;
        }
    }

    // print the 3D array
    for (int i = 0; i < X; i++)
    {
        for (int j = 0; j < Y; j++)
        {
            for (int k = 0; k < Z; k++)
            {
                std::cout << *(A + i * Y * Z + j * Z + k) << " ";
            }
            std::cout << std::endl;
        }
        std::cout << std::endl;
    }
    // deallocate memory
    delete[] A;

    return 0;
}
```

## 2. Using Triple Pointer

```
#include <iostream>
// `X Ã– Y Ã– Z` matrix
#define X 2
#define Y 3
#define Z 4
// Dynamically allocate memory for 3D Array in C++
int main()
{
    int*** A = new int**[X];

    for (int i = 0; i < X; i++)
    {
        A[i] = new int*[Y];
        for (int j = 0; j < Y; j++)
        {
```

```
        A[i][j] = new int[Z];
    }
}

// assign values to the allocated memory
for (int i = 0; i < X; i++)
{
    for (int j = 0; j < Y; j++)
    {
        for (int k = 0; k < Z; k++)
        {
            A[i][j][k] = rand() % 100;
        }
    }
}

// print the 3D array
for (int i = 0; i < X; i++)
{
    for (int j = 0; j < Y; j++)
    {
        for (int k = 0; k < Z; k++)
        {
            std::cout << A[i][j][k] << " ";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

// deallocate memory
for (int i = 0; i < X; i++)
{
    for (int j = 0; j < Y; j++)
    {
        delete[] A[i][j];
    }
    delete[] A[i];
}

delete[] A;

return 0;
```

}

#### C++ program for array implementation of List ADT

**Concept:** A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – each link of a linked list can store a data called an element.
- **Next** – each link of a linked list contains a link to the next link called Next.
- **Linked List** – A Linked List contains the connection link to the first link called First.

```
#include<iostream>
#include<conio.h>
#include<process.h>

int length = 0;
void create();
void insert();
void deletion();
void search();
void display();
int size();
int a, b[20], n, d, e, f, i;
using namespace std;
int main()
{
    int c;

    cout << "\n Main Menu";
    cout << "\n 1.Create \n 2.Delete \n 3.Search \n 4.insert \n
5.Display \n 6.Exit";
    do
    {
        cout << "\n enter your choice:";
        cin >> c;
```

```
switch (c)
{
    case 1:
        create();
        break;
    case 2:
        deletion();
        break;
    case 3:
        search();
        break;
    case 4:
        insert();
        break;
    case 5:
        display();
        break;
    case 6:
        exit(0);
        break;
    default:
        cout << "The given number is not between 1-5\n";
}
} while (c <= 6);
getch();
}
void create()
{
    cout << "\n Enter the number of elements you want to create: ";
    cin >> n;
    length = n;
    cout << "\nenter the elements\n";
    for (i = 0; i < n; i++)
    {
        cin >> b[i];
    }
}

void deletion()
{
    if (length == 0)
        cout << "Array Empty, Please Initilized it first.";
    else
```

```
{

    cout << "Enter the number u want to delete \n";
    cin >> d;
    for (i = 0; i < n; i++)
    {
        if (b[i] == d)
        {
            b[i] = 0;
            cout << d << " deleted";
            length--;
        }
    }
}

void search()
{
    cout << "Enter the number \n";
    cin >> e;
    for (i = 0; i < n; i++)
    {
        if (b[i] == e)
        {
            cout << "Value found the position\n" << i + 1;
        }
        else
        {
            cout << "The Value " << e << " found the position\n";
        }
    }
}

void insert()
{
    cout << "\nenter how many number u want to insert: ";
    cin >> f;
    cout << "\nEnter the elements\n";
    for (i = 0; i < f; i++)
    {
        cin >> b[n++];
        length++;
    }
}

void display()
{
```



```
    for (i = 0; i < n; i++)
    {
        cout << "\n" << b[i];
    }
}

int size()
{
    return length;
}
```



## LAB-04

### OBJECTIVE

#### Implementation of Singly Linked list and perform following operations

- Insertion
- Traversing all nodes in the Linked List
- Searching a specific item in the Linked List
- Deletion's operations

### PRE-LAB READING ASSIGNMENT

Understanding of Algorithms of Linked List Discussed in theory class

### Apparatus/Tools

C++

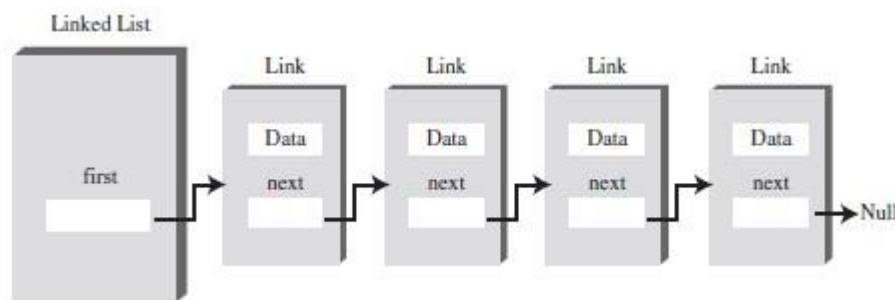
### Lab Related Contents:

#### **Linked List:**

The linked list is a versatile mechanism suitable for use in many kinds of general-purpose databases. It can also replace an array as the basis for other storage structures such as stacks and queues. In fact, you can use a linked list in many cases in which you use an array, unless you need frequent random access to individual items using an index.

#### **Links:**

In a linked list, each data item is embedded in a link. A link is an object of a class called something like Link. Because there are many similar links in a list, it makes sense to use a separate class for them, distinct from the linked list itself. Each Link object contains a reference (usually called next) to the next link in the list. A field in the list itself contains a reference to the first link.



*Figure 1: Links in Linked List*

#### **A Simple Linked List**

Our first example program, linkList.C++, demonstrates a simple linked list. The only operations allowed in this version of a list are:

- Inserting an item at the beginning of the list
- Deleting the item at the beginning of the list
- Iterating through the list to display its contents

---

### Experiment 01: Implement a Single linked List and perform basic operations

---

```
//Single Linklist implementation
#include<iostream>
using namespace std;
class node
{
    public:
        int data;
        node* next;
};
node* head = new node();
node* curr = new node();
int length = 0;
void GoToHead()
{ // set curr pointer to head node;
    curr = head;
}
void insertNodeAtEnd(int val)
{ // This function will insert new node at the end.
    GoToHead();
    node* t = new node();
    while (curr->next != NULL)
        curr = curr->next;
    t->data = val;
    t->next = NULL;
    curr->next = t;
    length++;
}
void AddNodeBeforeHead(int val)
{ // This function will insert new node as a head.
    GoToHead();
    node* t = new node();
    t->data = val;
    t->next = curr;
    head = t;
    length++;
}
void InsertAfterSpecificKey(int val, int key)
{
    node* t = new node();
    GoToHead();
    while (curr != NULL)
    {
```

```
        if (curr->data == key)
        {
            t->data = val;
            t->next = NULL;
            t->next = curr->next;
            curr->next = t;
            length++;
            break;
        }
        curr = curr->next;
    }
}

void InsertBeforeSpecificKey(int val, int key)
{
    node* ptr = NULL;
    GoToHead();
    while (curr != NULL)
    {
        if (curr->data == key)
        {
            node* t = new node();
            t->data = val;
            t->next = NULL;
            t->next = curr;
            ptr->next = t;
            length++;
            break;
        }
        ptr = curr;
        curr = curr->next;
    }
}

void printLinklist()
{
    GoToHead();
    while (curr != NULL)
    {
        cout << curr->data << "\t";
        curr = curr->next;
    }
}

void DeleteNodeUsingKey(int key)
{
    GoToHead();
    node* prenode = new node();
    if (curr->data == key)
    {
        head = curr->next;
        delete curr;
        length--;
        return;
    }
    else
        while (curr != NULL)
        {
            if (curr->data == key)
```

```
        {
            prenode->next = curr->next;
            delete curr;
            length--;
            break;
        }
        prenode = curr;
        curr = curr->next;
    }
}

void DeleteNodeUsingPos(int pos)
{
    GoToHead();
    node* prenode = new node();
    if (pos > length)
    {
        cout << "This Position dosenot exist" << endl;
        return;
    }
    else if (pos == 1)
    { // if we want to delet head node
        prenode = curr;
        head = curr->next;
        delete prenode;
        length--;
    }
    else
    {
        for (int x = 1; x < pos; x++)
        {
            prenode = curr;
            curr = curr->next;
        }
        prenode->next = curr->next;
        delete curr;
        length--;
    }
}

void InsertNodeUsingKey(int val, int key, bool isBefore)
{
    if (isBefore)
        InsertBeforeSpecificKey(val, key);
    else
        InsertAfterSpecificKey(val, key);
}

void InsertNodeUsingPos(int val, int pos, bool isBefore)
{
    GoToHead();
    if (pos > length)
    {
        cout << "This Position dosenot exist" << endl;
        return;
    }
}
```

```
else if (pos == 1 && isBefore)
{ // if we want to insert before head
    AddNodeBeforeHead(val);
}
else
{
    node* prenode = new node();
    for (int x = 1; x < pos; x++)
    {
        prenode = curr;
        curr = curr->next;
    }
    if (isBefore)
    {
        node* t = new node();
        t->data = val;
        t->next = NULL;
        t->next = curr;
        prenode->next = t;
    }
    else
    {
        node* t = new node();
        t->data = val;
        t->next = NULL;
        t->next = curr->next;
        curr->next = t;
    }
}

}

int main()
{
    head->data = 1;
    head->next = NULL;
    insertNodeAtEnd(2);
    insertNodeAtEnd(3);
    insertNodeAtEnd(4);
    printLinklist();
    cout << endl;
    InsertAfterSpecificKey(99, 2);
    printLinklist();
    cout << endl;
    DeleteNodeUsingKey(99);
    printLinklist();
    cout << endl;
    InsertBeforeSpecificKey(99, 2);
    printLinklist();
    cout << endl;
    InsertNodeUsingPos(88, 1, true);
    printLinklist();
    cout << endl;
    DeleteNodeUsingPos(1);
    DeleteNodeUsingPos(2);
    printLinklist();
}
```



```
cout << endl;  
return 0;  
}
```

### Output

D:\Anjum FAST Labs\Data Structure -Lab\lab 4\SingleLinkedList.exe

```
1      2      3      4  
1      2      99     3      4  
1      2      3      4  
1      99     2      3      4  
88     1      99     2      3      4  
1      2      3      4
```

## LAB-05

### OBJECTIVE

#### Implementation of Doubly Linked list and perform following operations

- Traversing all nodes in the Linked List
- Searching a specific item in the Linked List
- Delete etc.

### PRE-LAB READING ASSIGNMENT

Understanding of Algorithms of Linked List Discussed in theory class

### Apparatus/Tools

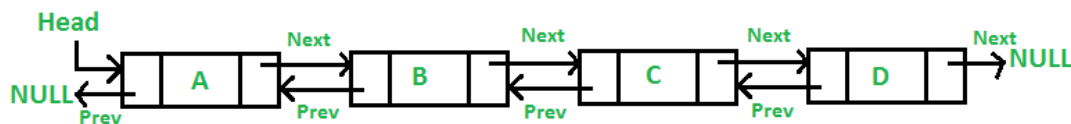
C++

### Lab Related Contents:

#### Doubly Linked List |

A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

**Prerequisites:** Linked List Introduction, Inserting a node in Singly Linked List



Advantages of DLL over the singly linked list:

- A DLL can be traversed in both forward and backward directions.
- The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

Disadvantages of DLL over the singly linked list:

- Every node of DLL Requires extra space for a previous pointer. It is possible to implement DLL with a single pointer though (See this and this).
- All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with the next pointers. For example in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

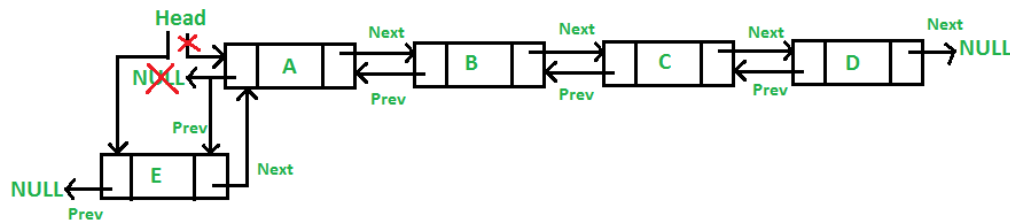
Insertion in DLL:

A node can be added in four ways:

- At the front of the DLL
- After a given node.
- At the end of the DLL
- Before a given node.

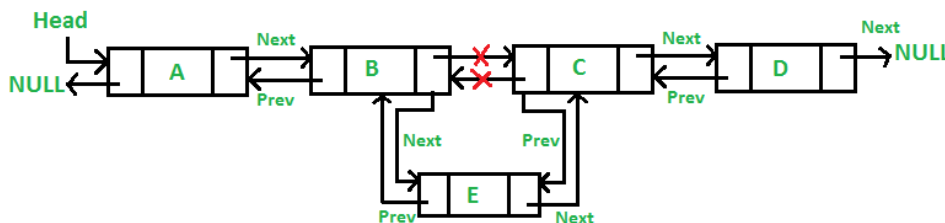
### 1) Add a node at the front:

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example, if the given Linked List is **1->0->1->5** and we add an item **5** at the front, then the Linked List becomes **5->1->0->1->5**. Let us call the function that adds at the front of the list push(). The push() must receive a pointer to the head pointer because the push must change the head pointer to point to the new node (See this)



**Note:** Four steps of the above five steps are the same as the 4 steps used for inserting at the front in the singly linked list. The only extra step is to change the previous head.

We are given a pointer to a node as prev\_node, and the new node is inserted after the given node.



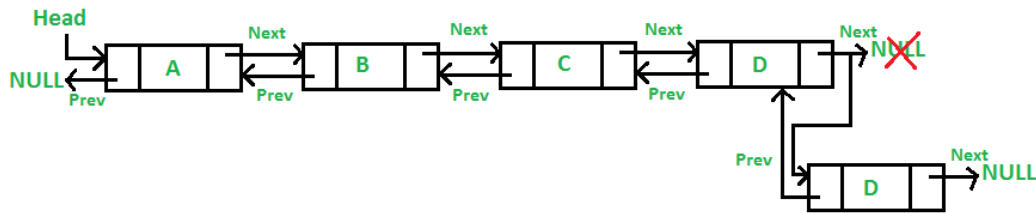
**Note:** Five of the above steps step process are the same as the 5 steps used for inserting after a given node in the singly linked list. The two extra steps are needed to change the previous pointer of the new node and the previous pointer of the new node's next node.

### 3) Add a node at the end:

The new node is always added after the last node of the given Linked List. For example, if the given DLL is **5->1->0->1->5->2** and we add item **30** at the end, then the DLL becomes **5->1->0-**



>1->5->2->30. Since a Linked List is typically represented by its head of it, we have to traverse the list till the end and then change the next of last node to the new node.



Below is the implementation of the 7 steps to insert a node at the end of the linked list:

- Inserting an item at the beginning of the list
- Deleting the item at the beginning of the list
- Iterating through the list to display its contents

---

### Experiment 01: Implement a Single linked List and perform basic operations

---

```

/*
 * C++ Program to Implement Doubly Linked List
 */
#include<iostream>
#include<cstdio>
#include<cstdlib>
/*
 * Node Declaration
 */
using namespace std;
class node
{
public:
    int info;
    class node *next;
    node* prev;
}

;
node* start;
/*
 * Class Declaration
 */
class double_llist
{
public:
    void create_list(int value);
    void add_begin(int value);
}
    
```

```
void add_after(int value, int position);
void delete_element(int value);
void search_element(int value);
void display_dlist();
void count();
void reverse();
double_llist()
{
    start = NULL;
}
};

/*
 * Main: Conatins Menu
 */
int main()
{
    int choice, element, position;
    double_llist dl;
    while (1)
    {
        cout << endl << "-----" << endl;
        cout << endl << "Operations on Doubly linked list" << endl;
        cout << endl << "-----" << endl;
        cout << "1.Create Node" << endl;
        cout << "2.Add at begining" << endl;
        cout << "3.Add after position" << endl;
        cout << "4.Delete" << endl;
        cout << "5.Display" << endl;
        cout << "6.Count" << endl;
        cout << "7.Reverse" << endl;
        cout << "8.Quit" << endl;
        cout << "Enter your choice : ";
        cin >> choice;
        switch (choice)
        {
            case 1:
                cout << "Enter the element: ";
                cin >> element;
                dl.create_list(element);
                cout << endl;
                break;
            case 2:
                cout << "Enter the element: ";
                cin >> element;
                dl.add_begin(element);
                cout << endl;
                break;
            case 3:
                cout << "Enter the element: ";
                cin >> element;
                cout << "Insert Element after postion: ";
                cin >> position;
                dl.add_after(element, position);
                cout << endl;
                break;
            case 4:
```

```
        if (start == NULL)
        {
            cout << "List empty,nothing to delete" << endl;
            break;
        }
        cout << "Enter the element for deletion: ";
        cin >> element;
        dl.delete_element(element);
        cout << endl;
        break;
    case 5:
        dl.display_dlist();
        cout << endl;
        break;
    case 6:
        dl.count();
        break;
    case 7:
        if (start == NULL)
        {
            cout << "List empty,nothing to reverse" << endl;
            break;
        }
        dl.reverse();
        cout << endl;
        break;
    case 8:
        exit(1);
    default:
        cout << "Wrong choice" << endl;
    }
}
return 0;
}

/*
 * Create Double Link List
 */
void double_llist::create_list(int value)
{
    node* s, *temp;
    temp = new(node);
    temp->info = value;
    temp->next = NULL;
    if (start == NULL)
    {
        temp->prev = NULL;
        start = temp;
    }
    else
    {
        s = start;
        while (s->next != NULL)
            s = s->next;
        s->next = temp;
        temp->prev = s;
    }
}
```

```
}

/*
 * Insertion at the beginning
 */
void double_llist::add_begin(int value)
{
    if (start == NULL)
    {
        cout << "First Create the list." << endl;
        return;
    }
    node* temp;
    temp = new(node);
    temp->prev = NULL;
    temp->info = value;
    temp->next = start;
    start->prev = temp;
    start = temp;
    cout << "Element Inserted" << endl;
}

/*
 * Insertion of element at a particular position
 */
void double_llist::add_after(int value, int pos)
{
    if (start == NULL)
    {
        cout << "First Create the list." << endl;
        return;
    }
    node* tmp, *q;
    int i;
    q = start;
    for (i = 0; i < pos - 1; i++)
    {
        q = q->next;
        if (q == NULL)
        {
            cout << "There are less than ";
            cout << pos << " elements." << endl;
            return;
        }
    }
    tmp = new(node);
    tmp->info = value;
    if (q->next == NULL)
    {
        q->next = tmp;
        tmp->next = NULL;
        tmp->prev = q;
    }
    else
    {
        tmp->next = q->next;
        tmp->next->prev = tmp;
    }
}
```

```
        q->next = tmp;
        tmp->prev = q;
    }
    cout << "Element Inserted" << endl;
}

/*
 * Deletion of element from the list
 */
void double_llist::delete_element(int value)
{
    node* tmp, *q;
    /*first element deletion*/
    if (start->info == value)
    {
        tmp = start;
        start = start->next;
        start->prev = NULL;
        cout << "Element Deleted" << endl;
        free(tmp);
        return;
    }
    q = start;
    while (q->next->next != NULL)
    {
        /*Element deleted in between*/
        if (q->next->info == value)
        {
            tmp = q->next;
            q->next = tmp->next;
            tmp->next->prev = q;
            cout << "Element Deleted" << endl;
            free(tmp);
            return;
        }
        q = q->next;
    }
    /*last element deleted*/
    if (q->next->info == value)
    {
        tmp = q->next;
        free(tmp);
        q->next = NULL;
        cout << "Element Deleted" << endl;
        return;
    }
    cout << "Element " << value << " not found" << endl;
}

/*
 * Display elements of Doubly Link List
 */
void double_llist::display_dlist()
{
    node* q;
    if (start == NULL)
    {
```

```
        cout << "List empty,nothing to display" << endl;
        return;
    }
    q = start;
    cout << "The Doubly Link List is :" << endl;
    while (q != NULL)
    {
        cout << q->info << " <-> ";
        q = q->next;
    }
    cout << "NULL" << endl;
}

/*
 * Number of elements in Doubly Link List
 */
void double_llist::count()
{
    node* q = start;
    int cnt = 0;
    while (q != NULL)
    {
        q = q->next;
        cnt++;
    }
    cout << "Number of elements are: " << cnt << endl;
}

/*
 * Reverse Doubly Link List
 */
void double_llist::reverse()
{
    node* p1, *p2;
    p1 = start;
    p2 = p1->next;
    p1->next = NULL;
    p1->prev = p2;
    while (p2 != NULL)
    {
        p2->prev = p2->next;
        p2->next = p1;
        p1 = p2;
        p2 = p2->prev;
    }
    start = p1;
    cout << "List Reversed" << endl;
}
```

## LAB-06

### OBJECTIVE

After this lab students, will be able to implement. Circular linked list

### PRE-LAB READING ASSIGNMENT

Understanding concepts of Circular linked list discussed in theory class.

### LAB RELATED CONTENT

#### Circular Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

---

### Experiment 01: Implement Linked list Basic operations

---

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

struct Node
{
    int data;
    struct Node* next;
};

struct Node* addToEmpty(struct Node* last, int data)
{
    // This function is only for empty list
    if (last != NULL)
        return last;

    // Creating a node dynamically.
    struct Node* temp
        = (struct Node*)malloc(sizeof(struct Node));

    // Assigning the data.
    temp->data = data;
    last = temp;

    // Creating the link.
    last->next = last;
    return last;
}

struct Node* addBegin(struct Node*last, int data)
{
    if (last == NULL)
```

```
    return addToEmpty(last, data);

    struct Node*temp
        = (struct Node*)malloc(sizeof(struct Node));
temp->data = data;
temp->next = last->next;
last->next = temp;
return last;
}

struct Node*addEnd(struct Node*last, int data)
{
    if (last == NULL)
        return addToEmpty(last, data);

    struct Node*temp
        = (struct Node*)malloc(sizeof(struct Node));

temp->data = data;
temp->next = last->next;
last->next = temp;
last = temp;

return last;
}

struct Node*addAfter(struct Node*last, int data, int item)
{
    if (last == NULL)
        return NULL;

    struct Node *temp, *p;
p = last->next;

do
{
    if (p->data == item)
    {
        temp
            = (struct Node*)malloc(sizeof(struct Node));
temp->data = data;
temp->next = p->next;
p->next = temp;
if (p == last)
    last = temp;
return last;
    }
    p = p->next;
} while (p != last->next) ;

cout << item << " not present in the list." << endl;
return last;
}
```



```
void traverse(struct Node*last)
{
    struct Node*p;

    // If list is empty, return.
    if (last == NULL)
    {
        cout << "List is empty." << endl;
        return;
    }

    // Pointing to first Node of the list.
    p = last->next;

    // Traversing the list.
    do
    {
        cout << p->data << " ";
        p = p->next;
    } while (p != last->next);
}

// Driver code
int main()
{
    struct Node*last = NULL;
    last = addToEmpty(last, 6);
    last = addBegin(last, 4);
    last = addBegin(last, 2);
    last = addEnd(last, 8);
    last = addEnd(last, 12);
    last = addAfter(last, 10, 8);

    // Function call
    traverse(last);
    return 0;
}
```

## LAB-07

### OBJECTIVE

In this Lab Student will be able to perform following Operations on Stack Data Structure

- Implement stack using Arrays and linked list and perform PUSH & POP operations
- Read an input Expression and implement algorithms to convert it into Postfix
- Evaluation of Postfix Expression

### PRE-LAB READING ASSIGNMENT

Understanding of Stack, PUSH and POP

### Apparatus/ Tools

C++

### LAB RELATED CONTENT

**Stacks can be implemented using arrays and linked list.**

### STACKS using Arrays:

---

Experiment 01: This is a sample program to demonstrate push and pop functionality in Stack in C++ using Arrays.

---

```
// C++ program for the above methods
#include<iostream>
using namespace std;
class Stack
{
    private:
        int* arr;
        int size;
        int noOfElements;
        int top; // Index
    public:
        Stack(int size)
        {
            this->size = size;
            arr = new int[size];
            top = -1;
        }
        ~Stack()
        {
            delete[] arr;
        }
        void push(int val)
        {
            if (top == size - 1)
            {
                cout << "Stack OverFlow" << endl;
            }
        }
    };
};
```

```
        return;
    }
    arr[++top] = val;
    noOfElements++;
}
void push1(int val)
{
    if (top == size - 1)
    {
        cout << "Stack Overflow" << endl;
        return;
    }
    Stack temp(noOfElements);

    for (int x = 1; x <= noOfElements; x++)
    {
        if (temp.top == temp.size - 1)
        {
            cout << "Stack Overflow" << endl;
        }
        else
        {
            temp.arr[++temp.top] = pop();
            temp.noOfElements++;
        }
    }
    top = 0;
    arr[top] = val;
    for (int x = 1; x <= noOfElements; x++)
    {
        arr[++top] = temp.pop();
    }
    noOfElements++;
}
int pop()
{
    if (top == -1)
    {
        cout << "Stack UnderFlow" << endl;
        return 0;
    }
    else
        return arr[top--];
}
int peek()
{
    if (top == -1)
    {
        cout << "Stack UnderFlow" << endl;
        return 0;
    }
    return arr[top];
}
void display()
{
    int x = 0;
```

```
        for (int i = top; i >= 0; i--)
            cout << "Item: " << ++x << " : " << arr[i] << endl;
    }
};
int main()
{
    Stack stack1(5);
    stack1.push1(1);
    stack1.push1(2);
    stack1.push1(3);
    stack1.push1(4);
    stack1.push1(5);
    stack1.push1(6);
    // stack1.pop();
    stack1.display();
    return 0;
}
```

## STACKS using LinkedList:

```
// Stack USING sINGLE Linked List Implementation
//KS.
#include<iostream>
using namespace std;
class node
{
public:
    node* previous;
    int data;
    node(int val)
    {
        data = val;
        previous = NULL;
    }
};
class Stack
{
private:
    node* top;
    int noOfElements;
    int size;
public:
    Stack()
    {
        top = NULL;
        noOfElements = 0;
        size = 10;
    }
    void push(int val);
    // Display
    void display();
    // Remove
    int pop();
    int peek()
    {
        return top->data;
    }
}
```

```
    }
    bool isEmpty();
    int getLength();
    void setSize(int size)
    {
        this->size = size;
    }
};

int main()
{
    Stack stack1, stack2;
    stack1.push(1);
    stack1.push(2);
    stack1.push(3);
    stack1.push(4);
    stack1.push(5);
    stack1.push(6);
    stack1.pop();
    stack1.display();
    cout << endl << " Stack size: " << stack1.getLength() << endl;
    return 0;
}

int Stack :: getLength()
{
    return this->noOfElements;
}

bool Stack :: isEmpty()
{
    if (noOfElements == 0) // If (top==NULL)
        return true;
    else
        false;
}

int Stack:: pop()
{
    if (top == NULL)
    {
        cout << "List is Empty: Stack Under Flow, return -1" << endl;
        return -1;
    }
    else
    {
        node* curr = top;
        int data = curr->data;
        top = curr->previous;
        delete curr;
        noOfElements--;
        return data;
    }
}

void Stack::push(int val)
{

```

```
if (noOfElements < size)
{
    node* t = new node(val);
    if (top == NULL)
    { // Stack is Empty
        top = t;
        noOfElements++;
    }
    else
    {
        t->previous = top;
        top = t;
        noOfElements++;
    }
}
else
    cout << "Stack Full: Over Flow" << endl;
}
void Stack::display()
{
    node* curr = top;
    while (curr != NULL)
    {
        cout << "Node: " << curr->data << endl;
        curr = curr->previous;
    }
}
```

## LAB-08

### OBJECTIVE

**In this Lab Student, will be able to perform following Operations on Stack Data Structure**

- Evaluation of Postfix Expression
- Tower of Hanoi using recursion

### PRE-LAB READING ASSIGNMENT

NONE

### Tools/ Apparatus:

Dev C++, VS Code, Python Tutor

### LAB RELATED CONTENT

#### Evaluation of Postfix Expression

One of the most useful characteristics of postfix expression is that the value of postfix expression can be computed easily with the aid of a stack. The components of a postfix expression are processed from left to right as follows:

<i>Item scanned from Postfix Expression</i>	<i>Action</i>
Operand	Push operand onto the stack
Operator	Pop the top two operands from the stack, apply the operator to them, and evaluate it. Push this result onto the stack.

---

**Experiment 01: Write a program to Evaluate a postfix expression**

---

Code:

```
#include<iostream>
#include<stack>
```

```
using namespace std;

//class Stack{
// defines the Boolean function for operator, operand, equalOrhigher precedence and
the string conversion function.

bool IsOperator(char);
bool IsOperand(char);
bool eqlOrhigher(char, char);
string convert(string);

//};

int main()
{
    string infix_expression, postfix_expression;
    int ch;
    do
    {
        cout << " Enter an infix expression: ";
        cin >> infix_expression;
        postfix_expression = convert(infix_expression);
        cout << "\n Your Infix expression is: " << infix_expression;
        cout << "\n Postfix expression is: " << postfix_expression;
        cout << "\n \t Do you want to enter infix expression (1/ 0)?";
        cin >> ch;
        //cin.ignore();
    } while (ch == 1);
    return 0;
}

// define the IsOperator() function to validate whether any symbol is operator.
/* If the symbol is operator, it returns true, otherwise false. */
bool IsOperator(char c)
{
    if (c == '+' || c == '-' || c == '*' || c == '/' || c == '^')
        return true;
    return false;
}

// IsOperand() function is used to validate whether the character is operand.
bool IsOperand(char c)
{
    if (c >= 'A' && c <= 'Z') /* Define the character in between A to Z. If not, it
returns False.*/
        return true;
    if (c >= 'a' && c <= 'z') // Define the character in between a to z. If not, it
returns False. */
        return true;
    if (c >= '0' && c <= '9') // Define the character in between 0 to 9. If not,
it returns False. */
        return true;
    return false;
}

// here, precedence() function is used to define the precedence to the operator.
int precedence(char op)
{

```



```
    if (op == '+' || op == '-')                /* it defines the lowest
precedence */
        return 1;
    if (op == '*' || op == '/')
        return 2;
    if (op == '^')                            /* exponent operator has the
highest precedence */
        return 3;
    return 0;
}
/* The eql0rhigher() function is used to check the higher or equal precedence of the
two operators in infix expression. */
bool eql0rhigher(char op1, char op2)
{
    int p1 = precedence(op1);
    int p2 = precedence(op2);
    if (p1 == p2)
    {
        if (op1 == '^')
            return false;
        return true;
    }
    return (p1 > p2 ? true : false);
}

/* string convert() function is used to convert the infix expression to the postfix
expression of the Stack */
string convert(string infix)
{
    stack<char> S;
    string postfix = "";
    char ch;

    S.push('(');
    infix += ')';

    for (int i = 0; i < infix.length(); i++)
    {
        ch = infix[i];

        if (ch == ' ')
            continue;
        else if (ch == '(')
            S.push(ch);
        else if (IsOperand(ch))
            postfix += ch;
        else if (IsOperator(ch))
        {
            while (!S.empty() && eql0rhigher(S.top(), ch))
            {
                postfix += S.top();
                S.pop();
            }
            S.push(ch);
        }
        else if (ch == ')')
        {
            while (!S.empty())
            {
                postfix += S.top();
                S.pop();
            }
        }
    }
    return postfix;
}
```

```
        while (!S.empty() && S.top() != '(')
        {
            postfix += S.top();
            S.pop();
        }
        S.pop();
    }
}
return postfix;
}
```

#### Output

```
Enter postfix: 345+*612+/-
3 Stack (bottom-->top):
4 Stack (bottom-->top): 3
5 Stack (bottom-->top): 3 4
+ Stack (bottom-->top): 3 4 5
* Stack (bottom-->top): 3 9
6 Stack (bottom-->top): 27
1 Stack (bottom-->top): 27 6
2 Stack (bottom-->top): 27 6 1
+ Stack (bottom-->top): 27 6 1 2
/ Stack (bottom-->top): 27 6 3
- Stack (bottom-->top): 27 2
Evaluates to 25
```

#### Experiment 02: Write a program to Evaluate a prefix expression

```
#include<iostream>
using namespace std;
class Stack
{
private:
    char* arr;
    int size;
    int top; // Index
    int length;
public:

    Stack(int size)
    {
        this->size = size;
        arr = new char[size];
        top = -1;
        length = 0;
    }
    ~Stack()
    {
        delete[] arr;
    }
}
```

```
}
void push(char val);
int pop();
int peek();
bool IsEmpty();

bool eqlOrHigher(char op1, char op2);
int precedence(char op);
bool IsOperand(char c);
bool IsOperator(char c);

string postFix(string infix);
string preFix(string infix);

void display();
};

int main()
{
    Stack S(100);
    string infix_expression, postfix_expression;
    int ch;
    do
    {
        cout << " Enter an infix expression: ";
        cin >> infix_expression;
        postfix_expression = S.postFix(infix_expression);
        cout << "\n Your Infix expression is: " << infix_expression;
        cout << "\n Postfix expression is: " << postfix_expression;
        cout << "\n Prefix expression is: " << S.preFix(infix_expression);
        cout << "\n \t Do you want to enter infix expression (1/ 0)? ";
        cin >> ch;
        //cin.ignore();
    } while (ch == 1);
    return 0;
}

void Stack:: push(char val)
{
    if (top == size - 1)
    {
        cout << "Stack OverFlow" << endl;
        return;
    }
    arr[++top] = val;
    length++;
}

int Stack:: pop()
{
    if (top == -1)
    {
        cout << "Stack UnderFlow" << endl;
        return 0;
    }
    else
        return arr[top--];
}
```

```
void Stack:: display()
{
    for (int i = 0; i <= top; i++)
        cout << "Item: " << i + 1 << " : " << arr[i] << endl;
}
int Stack:: peek()
{
    if (top == -1)
    {
        cout << "Stack UnderFlow" << endl;
        return 0;
    }
    return arr[top];
}

bool Stack:: IsEmpty()
{
    if (length == 0)
        return true;
    return false;
}

string Stack:: postFix(string infix)
{
    string postfix = "";
    char ch;

    push('(');
    infix += ')';

    for (int i = 0; i < infix.length(); i++)
    {
        ch = infix[i];

        if (ch == ' ')
            continue;
        else if (ch == '(')
            push(ch);
        else if (IsOperand(ch))
            postfix += ch;
        else if (IsOperator(ch))
        {
            while (!IsEmpty() && eqLorHigher(peek(), ch))
            {
                postfix += peek();
                pop();
            }
            push(ch);
        }
        else if (ch == ')')
        {
            while (!IsEmpty() && peek() != '(')
            {
                postfix += peek();
                pop();
            }
        }
    }
}
```

```
        pop();
    }
}
return postfix;
}

string Stack:: preFix(string infix)
{
    string postfix = postFix(infix);
    Stack s1(postfix.length());
    for (int i = 0; i < postfix.length(); i++)
    {
        s1.push(postfix[i]);
    }

    string prefix = "";
    for (int i = 0; i < postfix.length(); i++)
    {
        prefix += s1.pop();
    }
    return prefix;
}

bool Stack:: IsOperator(char c)
{
    if (c == '+' || c == '-' || c == '*' || c == '/' || c == '^')
        return true;
    return false;
}

// IsOperand() function is used to validate whether the character is operand.
bool Stack:: IsOperand(char c)
{
    if (c >= 'A' && c <= 'Z') /* Define the character in between A to Z. If not, it
returns False.*/
        return true;
    if (c >= 'a' && c <= 'z') // Define the character in between a to z. If not, it
returns False. */
        return true;
    if (c >= '0' && c <= '9') // Define the character in between 0 to 9. If not,
it returns False. */
        return true;
    return false;
}

// here, precedence() function is used to define the precedence to the operator.
int Stack:: precedence(char op)
{
    if (op == '+' || op == '-') /* it defines the lowest
precedence */
        return 1;
    if (op == '*' || op == '/')
        return 2;
    if (op == '^') /* exponent operator has the
highest precedence */
        return 3;
    return 0;
}
```

/\* The eqlOrHigher() function is used to check the higher or equal precedence of the two operators in infix expression. \*/

```
bool Stack:: eqlOrHigher(char op1, char op2)
{
    int p1 = precedence(op1);
    int p2 = precedence(op2);
    if (p1 == p2)
    {
        if (op1 == '^')
            return false;
        return true;
    }
    return (p1 > p2 ? true : false);
}
```

## LAB-09

### OBJECTIVE

In this lab students, will perform following Operations in Queues

- Insertion
- Deletion
- Implement a Priority queue.
- Implement a double ended queue ADT using
  - i) array and
  - ii) doubly linked list respectively.

### PRE-LAB READING ASSIGNMENT

Read about Queues and its functionality.

### Tools/ Apparatus:

C++

### LAB RELATED CONTENT

#### QUEUES(FIFO)

---

### Experiment 01: Implement queues basic operation using arrays

---

#### Sample code:

```
#include<iostream>
using namespace std;
class node
{
    public:
        string name;
        string rollno;
        float cgpa;
        node(string name, string rollno, cgpa)
        {
```

```
        this->
    }

};
class Queue
{
    private:
        int* arr;
        int size;
        int rear, front; // Index
        int noOfElemets;

    public:
        Queue(int size)
        {
            this->size = size;
            arr = new int[size];
            rear = -1, front = -1, noOfElemets = 0;
        }
        ~Queue()
        {
            delete[] arr;
        }
        void enqueue(int val)
        {
            if (rear == size - 1)
            {
                cout << "Queue OverFlow" << endl;
                return;
            }
            if (front == -1)
                front = 0;
            arr[++rear] = val;
            noOfElemets++;
        }

        int dequeue()
        {
            if (front == -1)
            {
                cout << "Queue Underflow ";
                return -1;
            }
            else
            {
                cout << "Element deleted from queue is : " << arr[front] << endl;
                return arr[front++];
            }
        }
        void display()
        {
            if (front == -1)
                cout << "Queue is empty" << endl;
            else
            {
                cout << "Queue elements are : ";
            }
        }
    };
};
```

```
        for (int i = front; i <= rear; i++)
            cout << arr[i] << " ";
        cout << endl;
    }
}
};
int main()
{
    Queue q1(5);
    q1.enqueue(1);
    q1.enqueue(2);
    q1.enqueue(3);
    q1.enqueue(4);
    q1.enqueue(5);
    q1.display();
    q1.dequeue();
    q1.dequeue();
    q1.display();
    return 0;
}
```

---

```
Queue: Kim(1) Nitin(2) Laxmi(2) Babu(3) Jimmy(3)
delete(): Kim(1)
Queue: Nitin(2) Laxmi(2) Babu(3) Jimmy(3)
Queue: Anu(1) Nitin(2) Laxmi(2) Scot(2) Babu(3) Jimmy(3) Lehar(4)
```

---

### Experiment 02: Implement queues basic operation using arrays

```
// Queue USING SINGLE Linked List Implementation
//KS.
#include<iostream>
using namespace std;
class node
{
public:
    node* next;
    int data;
    node(int val)
    {
        data = val;
        next = NULL;
    }
};
class Queue
{
private:
    node* rear;
    node* front;
    int noOfElements;
```



```
public:
    Queue()
    {
        front = rear = NULL;
        noOfElements = 0;
    }
    // Insertion in Queue
    void enqueue(int val);
    // Display
    void display();
    // Remove
    int dequeue();
    bool isEmpty();
    int size();
};

int main()
{
    Queue Queue1;
    Queue1.enqueue(1);
    Queue1.enqueue(2);
    Queue1.enqueue(3);
    Queue1.enqueue(4);
    Queue1.enqueue(5);
    Queue1.enqueue(6);
    Queue1.dequeue();
    Queue1.display();
    cout << endl << " Queue size: " << Queue1.size() << endl;
    return 0;
}

int Queue:: size()
{
    return noOfElements;
}

bool Queue:: isEmpty()
{
    if (noOfElements == 0)
        return true;
    else
        false;
}

int Queue:: dequeue()
{
    node* curr = front;
    if (curr == NULL)
    {
        cout << "List is Empty: Queue Under Flow, return -1" << endl;
        return -1;
    }
    else
    {
        node* curr = front;
        int data = curr->data;
        front = curr->next;
        delete curr;
        noOfElements--;
        return data;
    }
}
```

```
}  
void Queue::enqueue(int val)  
{  
    node* t = new node(val);  
    if (front == NULL)  
    { // Queue is Empty  
        front = rear = t;  
        noOfElements++;  
    }  
    else  
    {  
        rear->next = t;  
        rear = rear->next;  
        noOfElements++;  
    }  
}  
void Queue::display()  
{  
    node* curr = front;  
    while (curr != NULL)  
    {  
        cout << "Node: " << curr->data << endl;  
        curr = curr->next;  
    }  
}
```

## STUDENT TASK-01

Write programs to implement the deque (double ended queue) ADT using

- (a) Array
- (b) Doubly linked list.

## LAB-10

### OBJECTIVE

After this lab students will be able to work with following operations in Binary trees:

Inserting a node in binary search trees

Traversing in trees

- In order Traversing of a binary tree
- Preorder Traversing of a binary tree
- Post-order Traversing of a binary tree

Finding a node in Binary search trees

Deleting a node in binary tree(3- variations)

- Deleting a node with zero child
- Deleting a node having one child
- Deleting a node having two child

### PRE-LAB READING ASSIGNMENT

Read and understand algorithms discussed in class.

### Tools/ Apparatus:

Dev C++, VS Code, Python Tutor

### LAB RELATED CONTENT

#### Binary Search Trees

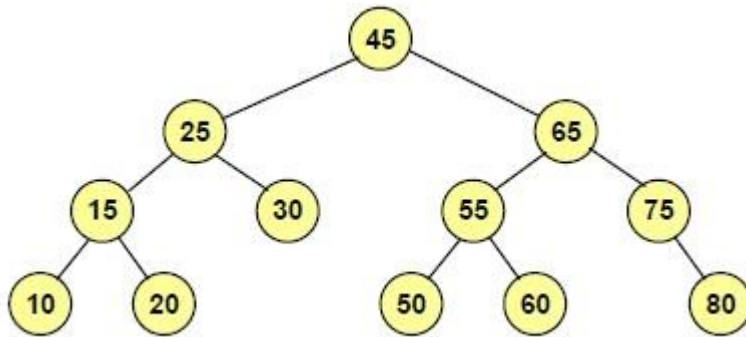
A binary tree is a special form of a tree. A binary tree is more important and frequently used in various applications of computer science. When binary trees are in sorted form, they facilitate quick search, insertion and deletion.

A binary tree is either empty, or it consists of a node called the root together with two binary trees called the left subtree or the right subtree of the root. This definition is that of a mathematical structure. To specify binary trees as an abstract data type, we must state what operations can be performed on binary trees.

A binary search tree is a binary tree that is either empty or in which every node contains a key and satisfies the conditions:

1. The key in the left child of a node (if it exists) is less than the key in its parent node.
2. The key in the right child of a node (if it exists) is greater than the key in its parent node.
3. The left and right subtrees of the root are again binary search trees.

The first two properties describe the ordering relative to the key in the root node, and that the third property extends them to all nodes in the tree; hence we can continue to use the recursive structure of the binary tree.



---

**Experiment 01: Write a C++ program to perform the following operations:**

- (a) Insert an element into a binary search tree.
- (b) Delete an element from a binary search tree.
- (c) Search for a key element in a binary search tree.
- (d) Traverse

---

Code:

```
// Binary Search Tree Implementation..
#include<iostream>
using namespace std;
class node
{
public:
    int data;
    node* left, *right;
    node(int data)
    {
        this->data = data;
        left = right = NULL;
    }
};

class BinarySearchTree
{
private:
    node* Insert(node* root, int val);
    node* DeleteNodeInBST(node* root, int data);
    node* InOrderTraversal(node* root);
    node* PreOrderTraversal(node* root);
    node* PostOrderTraversal(node* root);
    node* Merge(node* r1, node* r2);
    node* FindMax(node* root);
    int leafCount(node* root);
    int treeHeight(node* root);

public:
    node* root;
    BinarySearchTree()
    {
        root = NULL;
    }
};
```

```
{
    root = NULL;
}
int count = 0;

node* countNode(node* r)
{
    if (r == NULL)
        return r;
    count++;
    Count(r->left);
    Count(r->right);
}

int x = 1;
int y = 1;
node* Odd(node* r, int count)
{
    if (r == NULL)
        return r;
    Odd(r->left, count);
    if (x == count)
    {
        cout << "Mode: " << r->data;
    }
    x++;
    Odd(r->right, count);
}

int sum = 0;
node* Even(node* r, int count)
{
    if (r == NULL)
        return r;
    Even(r->left, count);

    if (y == count)
    {
        sum = sum + r->data;
    }

    if (y == count + 1)
    {
        sum = sum + r->data;
        cout << "Mode: " << sum / 2;
    }
    y++;
    Even(r->right, count);
}

void Mode()
{
    countNode(this->root);
}
```

```
        if (count % 2 == 0)
        {
            Even(this->root, count / 2);
        }
        else
        {
            Odd(this->root, (count + 1) / 2);
        }
    }

    void Insert(int val)
    {
        Insert(this->root, val);
    }

    void Delete(int val)
    {
        DeleteNodeInBST(this->root, val);
    }

    void InOrderTraversal()
    {
        InOrderTraversal(this->root);
    }
    void PreOrderTraversal()
    {
        PreOrderTraversal(this->root);
    }
    void PostOrderTraversal()
    {
        PostOrderTraversal(this->root);
    }

    void Merge(BinarySearchTree bst1, BinarySearchTree bst2)
    {
        Merge(bst1.root, bst2.root);
    }
    void FindMax()
    {
        FindMax(this->root);
    }
    int leafCount()
    {
        return leafCount(this->root);
    }
    int treeHeight()
    {
        return treeHeight(this->root);
    }
    int getCount()
    {
        return count;
    }
};
```

```
int main()
{
    BinarySearchTree tree1, tree2;

    tree1.Insert(6);
    tree1.Insert(3);
    tree1.Insert(4);
    tree1.Insert(8);
    tree1.Insert(9);
    // tree1.Insert(2);
    tree1.Mode();

    return 0;
}

node* BinarySearchTree::FindMax(node* r)
{
    while (r->right != NULL)
    {
        r = r->right;
    }
    return r;
}

node* BinarySearchTree::Insert(node* r, int val)
{
    if (r == NULL)
    {
        node* t = new node(val);

        if (r == root)
            root = r = t;
        else
            r = t;

        return r;
    }
    // else if (r->data == val){
    //     //cout<<"Duplicate Record  "<<val;
    //     return r;
    // }
    else if (val < r->data)
    {
        r->left = Insert(r->left, val);
        return r;
    }

    else if (val > r->data)
    {
        r->right = Insert(r->right, val);
        return r;
    }
}
```

```
node* BinarySearchTree::DeleteNodeInBST(node* root, int data)
{
    if (root == NULL)
        return root;
    else if (data < root->data)
        root->left = DeleteNodeInBST(root->left, data);
    else if (data > root->data)
        root->right = DeleteNodeInBST(root->right, data);
    else
    {
        //No child
        if (root->right == NULL && root->left == NULL)
        {
            delete root;
            root = NULL;
            return root;
        }
        //One child on left
        else if (root->right == NULL)
        {
            node* temp = root;
            root = root->left;
            delete temp;
        }
        //One child on right
        else if (root->left == NULL)
        {
            node* temp = root;
            root = root->right;
            delete temp;
        }
        //two child
        else
        {
            node* temp = FindMax(root->left);
            root->data = temp->data;
            root->left = DeleteNodeInBST(root->left, temp->data);
        }
    }
    return root;
}
```

```
node* BinarySearchTree::InOrderTraversal(node* r)
{
    if (r == NULL)
        return NULL;
    /* first recur on left child */
    InOrderTraversal(r->left);
    /* then print the data of node */
    cout << " " << r->data << " -> ";
    /* now recur on right child */
    InOrderTraversal(r->right);
}
```



```
node* BinarySearchTree::PreOrderTraversal(node* r)
{
    if (r == NULL)
        return NULL;

    cout << " " << r->data << " -> ";
    PreOrderTraversal(r->left);
    PreOrderTraversal(r->right);
}

node* BinarySearchTree::PostOrderTraversal(node* r)
{
    if (r == NULL)
        return NULL;
    PostOrderTraversal(r->left);
    PostOrderTraversal(r->right);
    cout << " " << r->data << " -> ";
}

int BinarySearchTree::leafCount(node* r)
{
    int static count= 0;
    if (r == NULL)
        return 0;
    else if (r->left == NULL && r->right == NULL)
        return 1;

    return count + leafCount(r->left) + leafCount(r->right);
}

int BinarySearchTree::treeHeight(node* root)
{
    int static l_height= 0;
    int static r_height= 0;
    if (root == NULL)
        return -1;
    else
    {
        l_height = treeHeight(root->left);
        r_height = treeHeight(root->right);
        if (l_height > r_height)
            return (l_height + 1);
        else
            return (r_height + 1);
    }
}

// This method will merge tree1 into tree2
node* BinarySearchTree::Merge(node* r1, node* r2)
{
    if (r1 == NULL)
        return NULL;
    /* first recur on left child */
    Merge(r1->left, r2);

    Insert(r2, r1->data);
    /* now recur on right child */
    Merge(r1->right, r2);
}
```



}

```
Select D:\Anjum FAST Labs\Data Structure -Lab\lab 9\BinarySearchTree.exe
In Order Print (left--Root--Right)
6 -> 8 -> 9 -> 10 -> 14 -> 15 -> 20 ->
-----
Pre Order Print (Root--left--Right)
10 -> 8 -> 6 -> 9 -> 15 -> 14 -> 20 ->
-----
Post Order Print (left--Right--Root)
6 -> 9 -> 8 -> 14 -> 20 -> 15 -> 10 ->

The total leaf node in tree are: 4

The height of root node is : 2

After Merging
In Order Print (left--Root--Right)
6 -> 7 -> 8 -> 9 -> 10 -> 14 -> 15 -> 20 -> 33 ->
```

## LAB-11

### OBJECTIVE

After this lab students, will be able to AVL Tree coding

### PRE-LAB READING ASSIGNMENT

Read and understand AVL Tree discussed in class.

### Tools/ Apparatus:

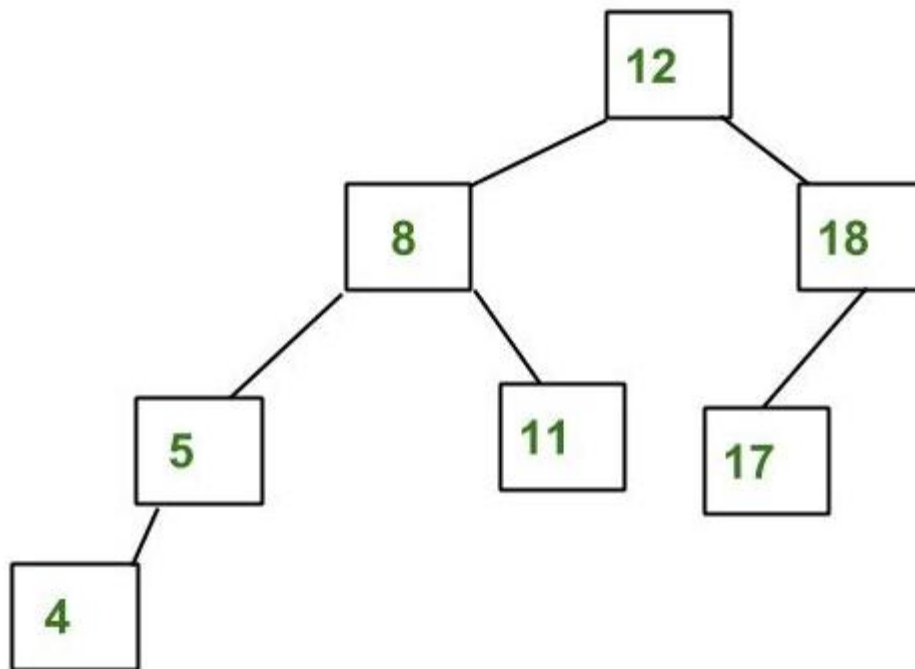
C++

### LAB RELATED CONTENT

#### AVL Tree:

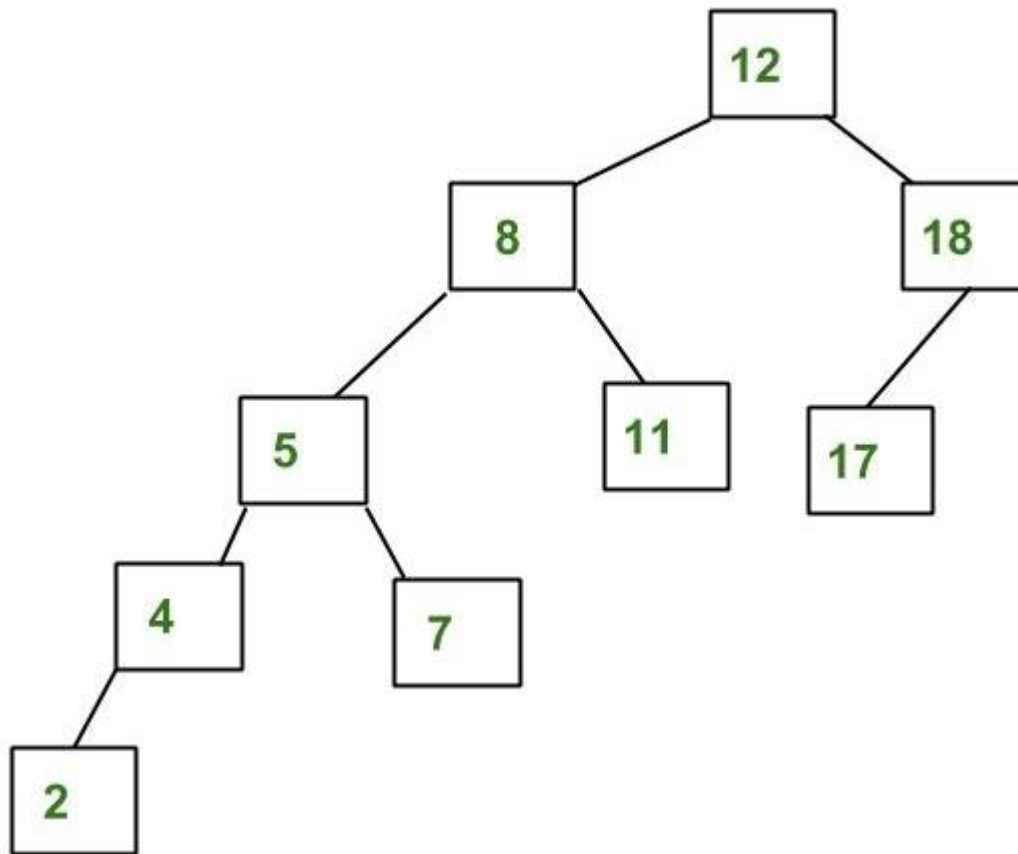
AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

#### Example of AVL Tree:



The above tree is AVL because the differences between heights of left and right subtrees for every node are less than or equal to 1.

Example of a Tree that is NOT an AVL Tree:



The above tree is not AVL because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take  $O(h)$  time where  $h$  is the height of the BST. The cost of these operations may become  $O(n)$  for a **skewed Binary tree**. If we make sure that the height of the tree remains  $O(\log(n))$  after every insertion and deletion, then we can guarantee an upper bound of  $O(\log(n))$  for all these operations. The height of an AVL tree is always  $O(\log(n))$  where  $n$  is the number of nodes in the tree.

Insertion in AVL Tree:

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing.

Following are two basic operations that can be performed to balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).

- Left Rotation
- Right Rotation

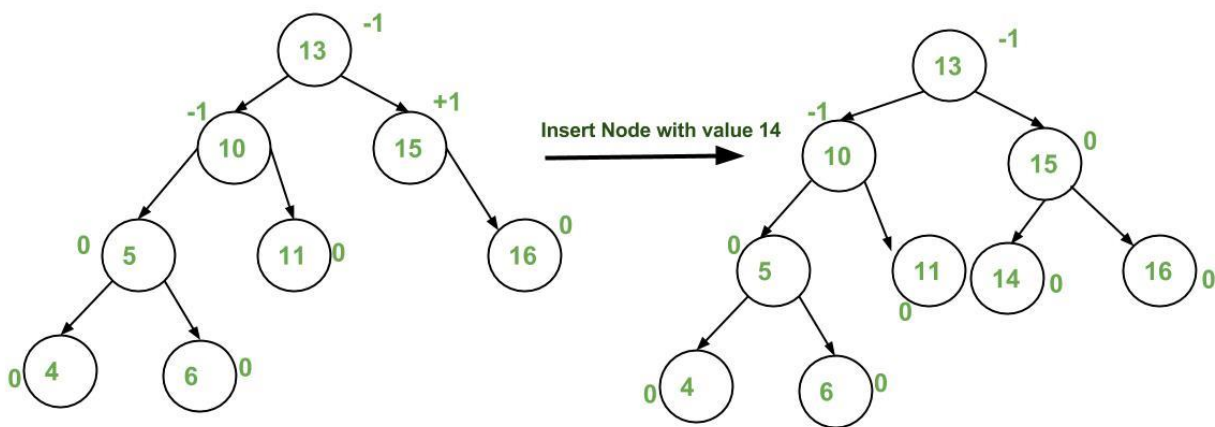
T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)

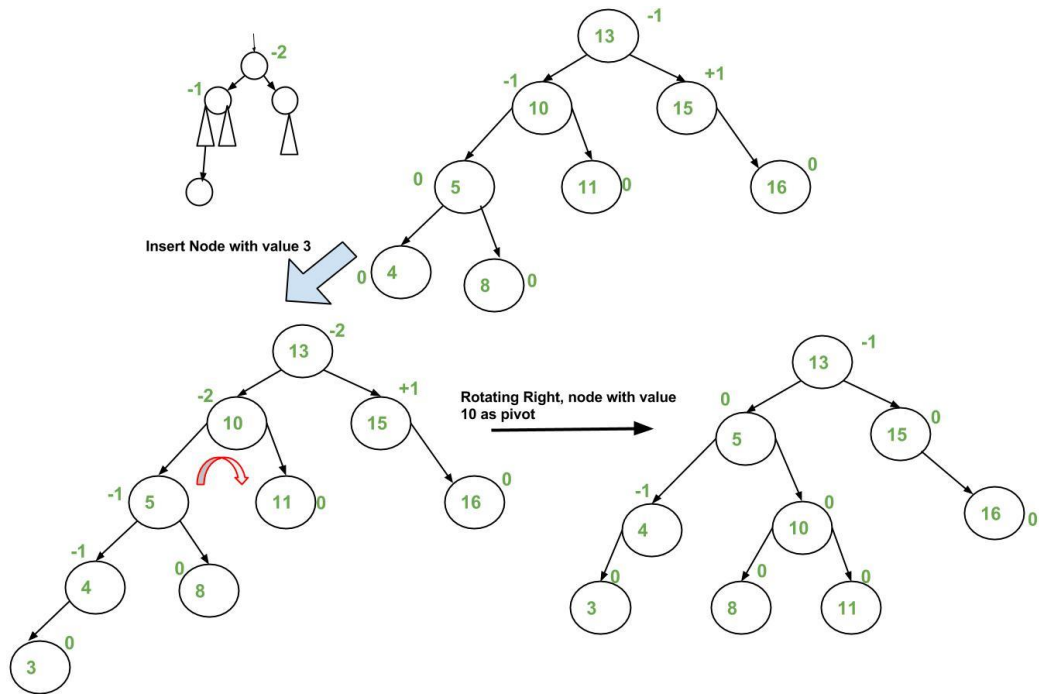
Let the newly inserted node be **w**

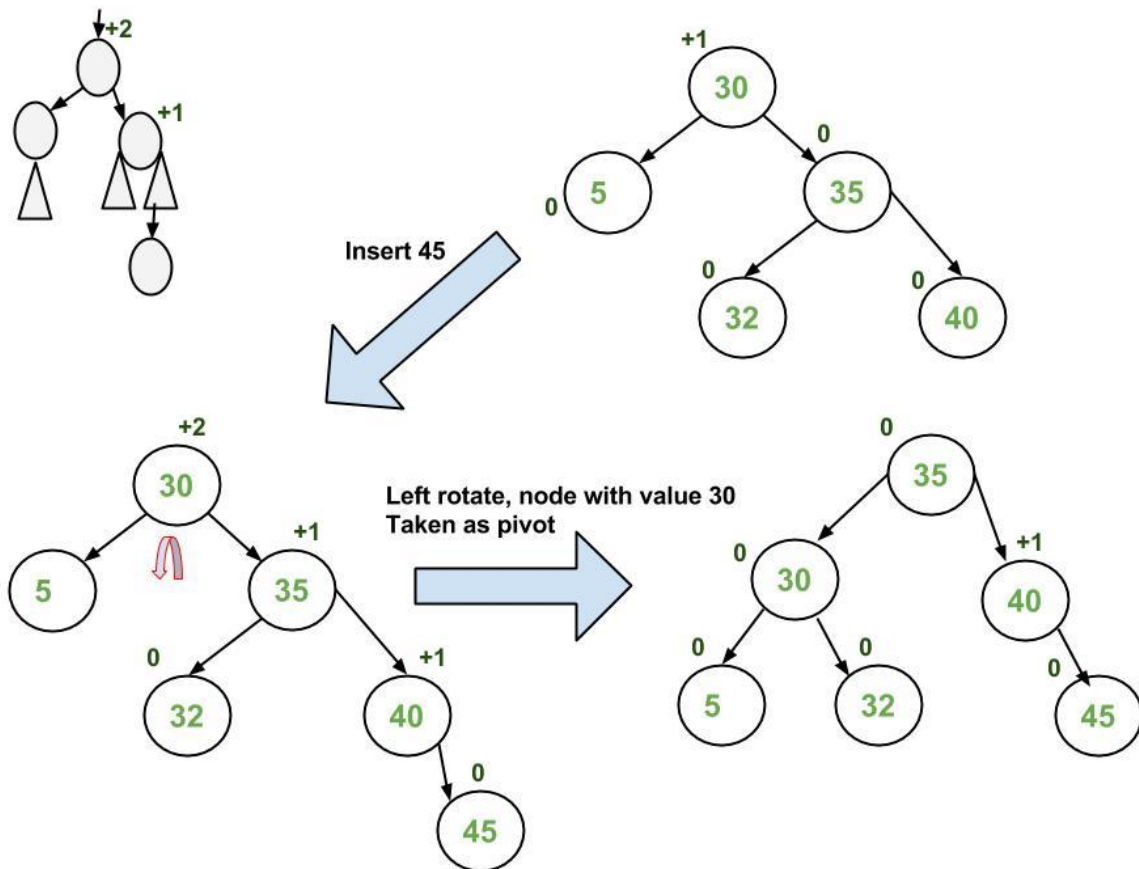
- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z**.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z**. There can be 4 possible cases that need to be handled as **x**, **y** and **z** can be arranged in 4 ways.
- Following are the possible 4 arrangements:
  - y is the left child of z and x is the left child of y (Left Left Case)
  - y is the left child of z and x is the right child of y (Left Right Case)
  - y is the right child of z and x is the right child of y (Right Right Case)
  - y is the right child of z and x is the left child of y (Right Left Case)

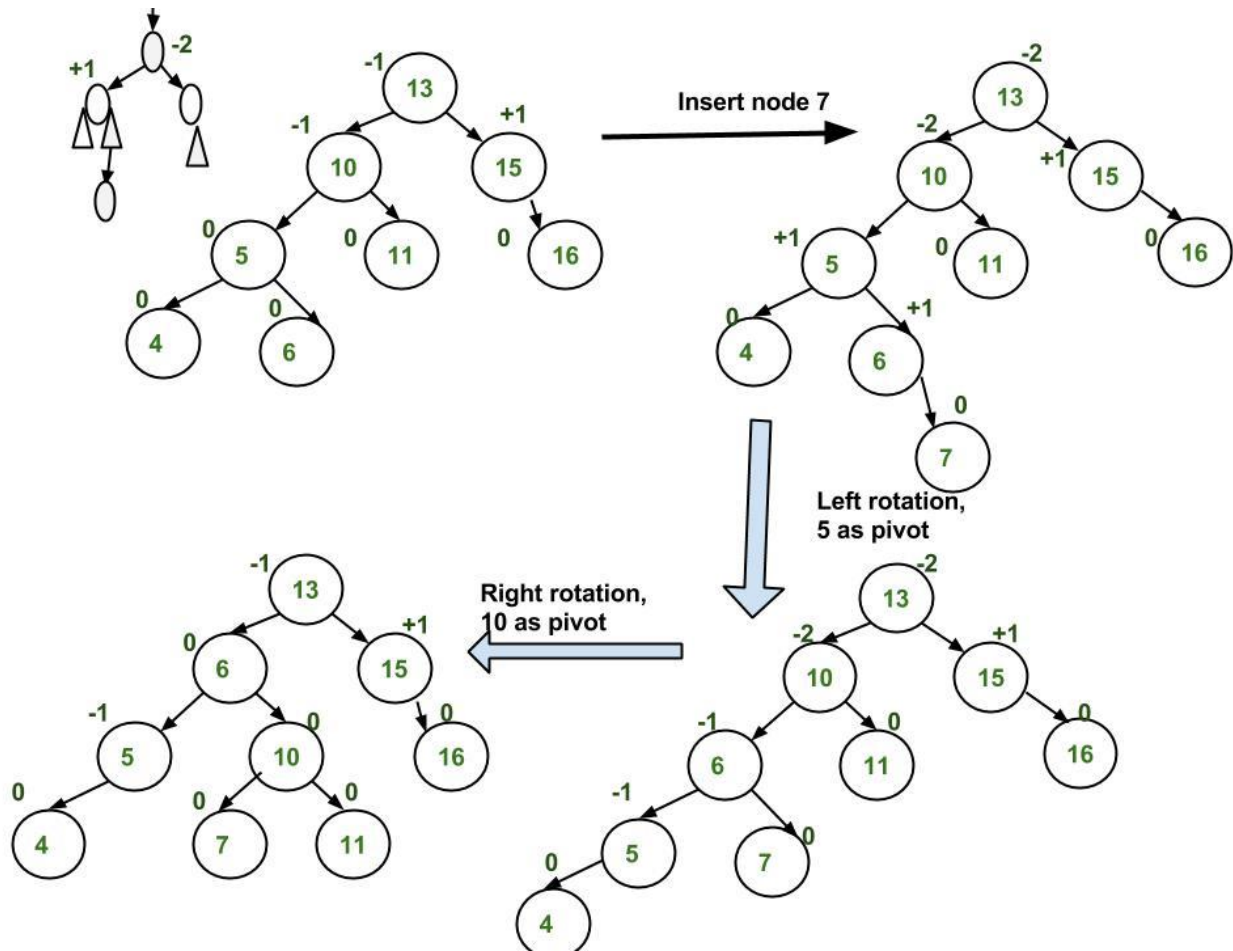
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to **re-balance** the subtree rooted with **z** and the complete tree becomes balanced as the height of the subtree (After appropriate rotations) rooted with **z** becomes the same as it was before insertion.

### Illustration of Insertion at AVL Tree

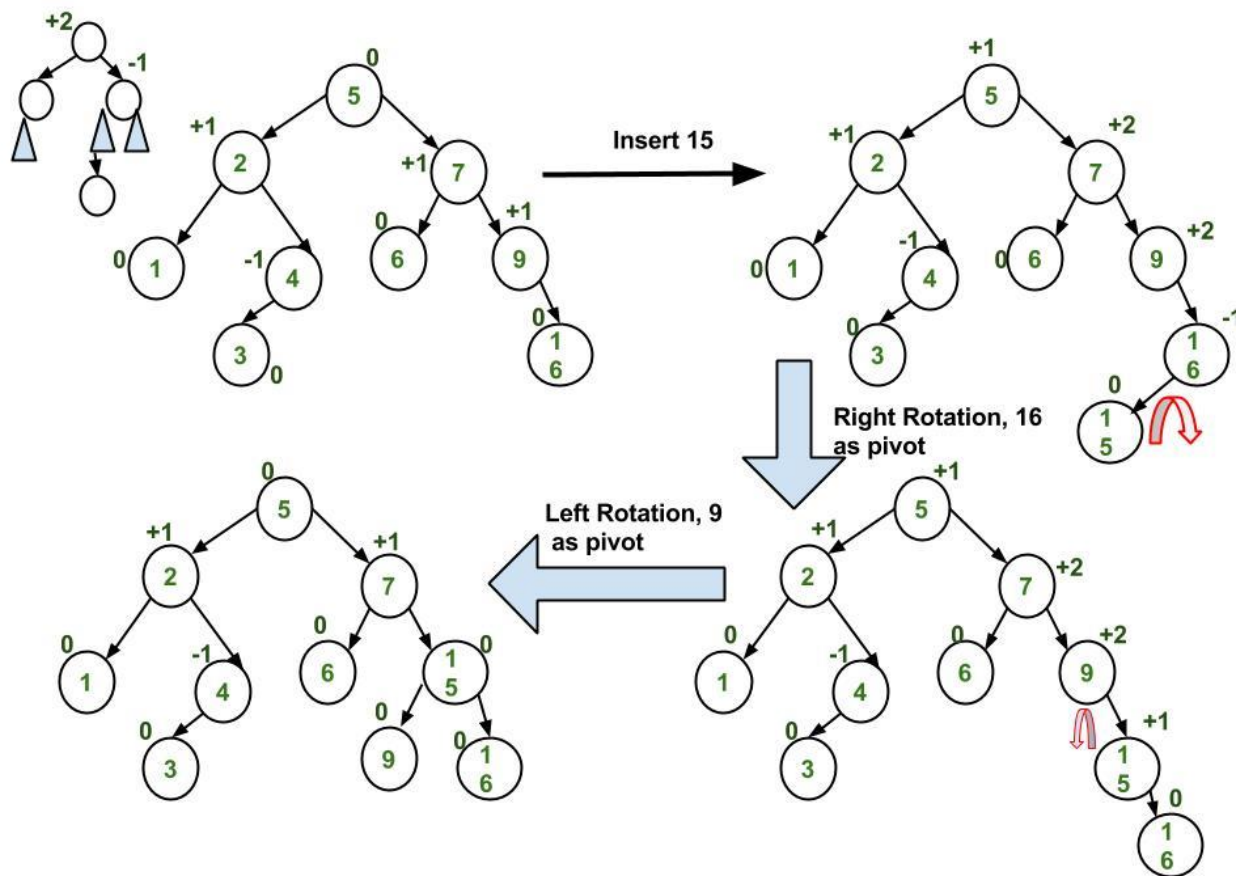












### Approach:

The idea is to use recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

Follow the steps mentioned below to implement the idea:

- Perform the normal BST insertion.
- The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.
- Get the balance factor (**left subtree height – right subtree height**) of the current node.
- If the balance factor is greater than **1**, then the current node is unbalanced and we are either in the **Left Left** case or **left Right** case. To check whether it is **left left** case or not, compare the newly inserted key with the key in the **left subtree root**.
- If the balance factor is less than **-1**, then the current node is unbalanced and we are either in the **Right Right** case or **Right-Left** case. To check whether it is the **Right Right** case or not, compare the newly inserted key with the key in the **right subtree root**.

Below is the implementation of the above approach:

```
/* AVL Tree Implementation in C++ */
/* @KS */
#include<iostream>
using namespace std;
class node
{
public:
    int data;
    node* left;
    node* right;
    int height;

    node(int data)
    {
        this->data = data;
        left = right = NULL;
        height = 0;
    }
};

class AVLTree
{
//private:

    node* root;
    void makeEmpty(node* t);
    node* insert(int x, node* t);

// Rotation
    node* singleRightRotate(node* &t);
    node* singleLeftRotate(node* &t);
    node* doubleRightLeftRotate(node* &t);
    node* doubleLeftRightRotate(node* &t);
    node* FindMax(node* r);

    int height(node* t)
    {
        return (t == NULL ? -1 : t->height);
    }
    node* findMin(node* t);
    node* findMax(node* t);
    node* remove(int x, node* t);

    int getBalance(node* t);
    void inorder(node* t);
    int treeHeight(node* root)
    {
        int static l_height= 0;
        int static r_height= 0;
        if (root == NULL)
            return -1;
        else
        {
            l_height = treeHeight(root->left);
```

```
        r_height = treeHeight(root->right);
        if (l_height > r_height)
            return (l_height + 1);
        else
            return (r_height + 1);
    }

}

public:
AVLTree()
{
    root = NULL;
}

void insert(int x)
{
    root = insert(x, root);
}

void remove(int x)
{
    root = remove(x, root);
}

void display()
{
    inorder(root);
    cout << endl;
}

};

int main()
{
    AVLTree tree1;
    tree1.insert(30);
    tree1.insert(20);
    tree1.insert(40);
    tree1.insert(15);
    tree1.remove(40);
    // tree1.insert(15);
    // tree1.insert(14);
    // tree1.insert(20);
    //// tree.insert(3);
    //// tree.insert(15);
    //// tree.insert(14);
    //// tree.insert(20);
    //// tree.insert(22);
    //// tree.insert(30);
    //// tree.insert(4);
    //// tree.insert(5);
    tree1.display();
    // tree.height();
}

void AVLTree::makeEmpty(node* t)
```

```
{
    if (t == NULL)
        return;
    makeEmpty(t->left);
    makeEmpty(t->right);
    delete t;
}

node* AVLTree:: insert(int x, node* t)
{
    if (t == NULL)
    {
        t = new node(x);
    }
    else if (x < t->data)
    {
        t->left = insert(x, t->left);
        int bf = height(t->left) - height(t->right);
        if (bf == 2)
        {
            if (x < t->left->data)
                t = singleRightRotate(t);
            else
                t = doubleLeftRightRotate(t);
        }
    }
    else if (x > t->data)
    {
        t->right = insert(x, t->right);
        int bf = height(t->right) - height(t->left);
        if (bf == 2)
        {
            if (x > t->right->data)
                t = singleLeftRotate(t);
            else
                t = doubleRightLeftRotate(t);
        }
    }

    t->height = max(height(t->left), height(t->right)) + 1;
    return t;
}

node* AVLTree::singleRightRotate(node* &t)
{
    node* u = t->left;
    t->left = u->right;
    u->right = t;
    t->height = max(height(t->left), height(t->right)) + 1;
    u->height = max(height(u->left), t->height) + 1;
    return u;
}

node* AVLTree::singleLeftRotate(node* &t)
{
    node* u = t->right;
    t->right = u->left;
```

```
    u->left = t;
    t->height = max(height(t->left), height(t->right)) + 1;
    u->height = max(height(u->right), t->height) + 1;
    return u;
}

node* AVLTree::doubleRightLeftRotate(node* &t)
{
    t->right = singleRightRotate(t->right);
    return singleLeftRotate(t);
}

node* AVLTree::doubleLeftRightRotate(node* &t)
{
    t->left = singleLeftRotate(t->left);
    return singleRightRotate(t);
}

node* AVLTree::findMin(node* t)
{
    if (t == NULL)
        return NULL;
    else if (t->left == NULL)
        return t;
    else
        return findMin(t->left);
}

node* AVLTree::findMax(node* t)
{
    if (t == NULL)
        return NULL;
    else if (t->right == NULL)
        return t;
    else
        return findMax(t->right);
}

node* AVLTree::FindMax(node* r)
{
    while (r->right != NULL)
    {
        r = r->right;
    }
    return r;
}

node* AVLTree::remove(int data, node* r)
{
    // node * r= root1;
    if (r == NULL)
        return r;
    else if (data < r->data)
        r->left = remove(data, r->left);
    else if (data > r->data)
        r->right = remove(data, r->right);
    else
    {
        //No child
    }
}
```

```
if (r->right == NULL && r->left == NULL)
{
    delete r;
    r = NULL;
    return r;
}
//One child on left
else if (r->right == NULL)
{
    node* temp = r;
    r = r->left;
    delete temp;
}
//One child on right
else if (r->left == NULL)
{
    node* temp = r;
    r = r->right;
    delete temp;
}
//two child
else
{
    node* temp = FindMax(r->right);
    int x = r->data; //
    r->data = temp->data;
    temp->data = x;
    r->left = remove(temp->data, r->left);
}
}
if (r == NULL)
    return r;

r->height = max(height(r->left), height(r->right)) + 1;
int bal = height(r->left) - height(r->right);
if (bal > 1)
{
    if (height(r->left) >= height(r->right))
    {
        return singleRightRotate(r);
    }
    else
    {
        r->left = singleLeftRotate(r->left);
        return singleRightRotate(r);
    }
}
else if (bal < -1)
{
    if (height(r->right) >= height(r->left))
    {
        return singleLeftRotate(r);
    }
    else
    {
        r->right = singleRightRotate(r->right);
        return singleLeftRotate(r);
    }
}
```

```
    }  
}  
  
    return r;  
}  
  
//    int AVLTree::height(node* t)  
//    {  
//        return ( t == NULL ? -1 : t->height);  
//    }  
  
int AVLTree::getBalance(node* t)  
{  
    if (t == NULL)  
        return 0;  
    else  
        return height(t->left) - height(t->right);  
}  
  
void AVLTree::inorder(node* t)  
{  
  
    if (t == NULL)  
        return;  
  
    cout << t->data << " -> ";  
    inorder(t->left);  
    inorder(t->right);  
  
}  
  
//    void AVLTree::preorder(node* t)  
//    {  
//        if(t == NULL)  
//            return;  
//        cout << t->data << " -> ";  
//        inorder(t->left);  
//        inorder(t->right);  
//    }
```

## LAB-12

### OBJECTIVE

After this lab students, will be able to understand and implement Following

- Heaps
- Heapsort

### PRE-LAB READING ASSIGNMENT

Read and understand algorithm of Heapsort discussed in class.

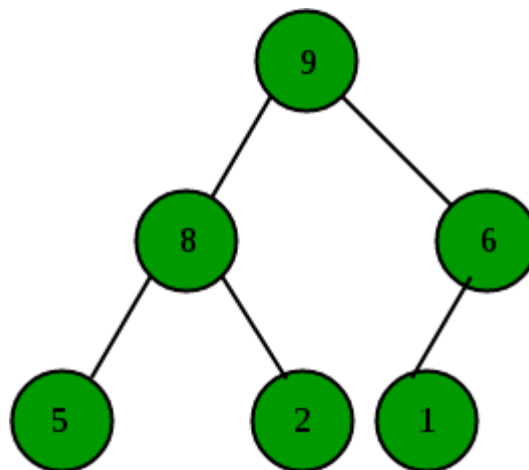
### Tools/ Apparatus:

Dev C++, VS Code, Python Tutor

### LAB RELATED CONTENT

A max-heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node. Mapping the elements of a heap into an array is trivial: if a node is stored an index  $k$ , then its left child is stored at index  $2k + 1$  and its right child at index  $2k + 2$ .

**Illustration:** Max Heap



### How is Max Heap represented?

A-Max Heap is a Complete Binary Tree. A-Max heap is typically represented as an array. The root element will be at  $Arr[0]$ . Below table shows indexes of other nodes for the  $i$ th node, i.e.,  $Arr[i]$ :

$Arr[(i-1)/2]$  Returns the parent node.

$Arr[2*i+1]$  Returns the left child node.

$Arr[2*i+2]$  Returns the right child node.

### Operations on Max Heap are as follows:

- **getMax():** It returns the root element of Max Heap. The Time Complexity of this operation is  $O(1)$ .
- **extractMax():** Removes the maximum element from **MaxHeap**. The Time Complexity of this Operation is  $O(\log n)$  as this operation needs to maintain the heap property by calling the [heapify\(\) method](#) after removing the root.
- **insert():** Inserting a new key takes  $O(\log n)$  time. We add a new key at the end of the tree. If the new key is smaller than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

*Note: In the below implementation, we do indexing from index 1 to simplify the implementation.*



## Methods:

There are 2 methods by which we can achieve the goal as listed:

1. Basic approach by creating **maxHeapify()** method
2. Using [Collections.reverseOrder\(\)](#) method via library Functions

**Method 1:** Basic approach by creating **maxHeapify()** method

We will be creating a method assuming that the left and right subtrees are already heapified, we only need to fix the root.

## Example

```
#include <assert.h>
// #include <algorithm>
#include <iostream>
using namespace std;
// Why do we use assert H in C++?
// Answer: An assert in C++ is a predefined macro using which we can test certain
assumptions that are set in the program.
// When the conditional expression in an assert statement is set to true, the program
continues normally. But when the expression is
// false, an error message is issued and the program is terminated.
class MaxHeap
{
    private:
        // Can be node class
        struct Node
        {
            int key;
        };
        Node* arr;
        int capacity; // Size of Heap
        int totalItems; //

        void doubleCapacity() // Increase the size
        {
            if (this->arr == NULL)
            {
                this->arr = new Node[1];
                this->capacity = 1;
                return;
            }

            int newCapacity = capacity * 2;
            Node* newArr = new Node[newCapacity];
            for (int i = 0; i < this->totalItems; i++)
            {
                newArr[i] = this->arr[i];
            }
            // if (this->arr != NULL)
            delete this->arr;
            this->capacity = newCapacity;
            this->arr = newArr;
        }
        void shiftUp(int index)
        {
            if (index < 1)
                return;
```

```
int parent = (index - 1) / 2;
if (this->arr[index].key > this->arr[parent].key)
{
    swap(this->arr[index], this->arr[parent]);
    shiftUp(parent);
}
return;
}
void shiftDown(int index)
{
    int maxIndex = -1;
    int lChildIndex = index * 2 + 1;
    int rChildIndex = (index * 2) + 2;
    if (lChildIndex < totalItems)
    {
        if (arr[index].key < arr[lChildIndex].key)
        {
            maxIndex = lChildIndex;
        }
    }
    if (rChildIndex < totalItems)
    {
        int newIndex = (maxIndex == -1 ? index : maxIndex);
        if (arr[newIndex].key < arr[rChildIndex].key)
        {
            maxIndex = rChildIndex;
        }
    }
    if (maxIndex == -1)
        return;
    swap(arr[index], arr[maxIndex]);
    shiftDown(maxIndex);
}
public:
    MaxHeap()
    {
        this->arr = NULL;
        this->capacity = 0;
        this->totalItems = 0;
    }
    MaxHeap(int _capacity)
    {
        assert(_capacity >= 1);
        this->arr = new Node[_capacity];
        this->capacity = _capacity;
        this->totalItems = 0;
    }
    void insert(int key)
    {
        if (this->totalItems == this->capacity)
        {
            doubleCapacity();
        }
        this->arr[totalItems].key = key;
        shiftUp(totalItems);
        this->totalItems++;
    }
}
```

```
void getMax(int & key)
{
    assert(totalItems != 0);
    key = this->arr[0].key;
}
// void getValue(int & key, int index)
// {
//     assert(totalItems != 0);
//     key = this->arr[index].key;
// }
void deleteMax()
{
    assert(totalItems != 0);
    swap(arr[0], arr[this->totalItems - 1]);
    totalItems--;
    shiftDown(0);
}
bool isEmpty() const
{
    return (totalItems == 0);
}
void deleteAll()
{
    if (this->arr != NULL)
    {
        delete[] arr;
        arr = NULL;
        this->capacity = 0;
        this->totalItems = 0;
    }
}
void InOrder(int index)
{
    if (index >= totalItems)
        return;
    InOrder(index * 2 + 1);
    cout << this->arr[index].key << " < ";
    InOrder(index * 2 + 2);
}
void PreOrder(int index)
{
    if (index > totalItems)
        return;
    cout << this->arr[index].key << " < ";
    PreOrder(index * 2 + 1);
    PreOrder(index * 2 + 2);
}
void PostOrder(int index)
{
    PostOrder(index * 2 + 1);
    PostOrder(index * 2 + 2);
    cout << this->arr[index].key << " < ";
    if (index > totalItems)
        return;
}
```

```
}
~MaxHeap()
{
    deleteAll();
}
};
int main()
{
    MaxHeap heap;
    // for (int i = 1; i <= 200; i++)
    //     heap.insert(i);
    // heap.deleteAll();
    //
    for (int i = 201; i <= 205; i++)
        heap.insert(i);
    cout << endl << "Pre Order" << endl;
    heap.PreOrder(0);

    // cout<<endl<<"In Order"<<endl;
    // heap.InOrder(0);
    //
    //
    // cout<<endl<<"Post Order"<<endl;
    // heap.PostOrder(0);

    // heap.deleteMax();
    while (!heap.isEmpty())
    {
        int s;
        heap.getMax(s);
        cout << s << endl;
        heap.deleteMax();
    }
}
```

## LAB-13

### OBJECTIVE

- Hash

### PRE-LAB READING ASSIGNMENT

#### Tools/ Apparatus:

C++

### LAB RELATED CONTENT

```
// Implementing hash table in C++
//Open Hashing (Separate Chaining)
#include <iostream>
#include <list>
using namespace std;
class HashTable
{
    int capacity;
    list<string>* table;

public:
    HashTable(int size);
    void insertItem(int rollNo, string name);
    void deleteItem(int rollNo, string name);
    void searchItem(int rollNo, string name);
    void displayHash();
    int checkPrime(int n)
    {
        int i;
        if (n == 1 || n == 0)
        {
            return 0;
        }

        for (i = 2; i < n / 2; i++)
        {
            if (n % i == 0)
            {
                return 0;
            }
        }
        return 1;
    }

    int getPrime(int n)
    {
        if (n % 2 == 0)
        {
            n++;
        }
        while (!checkPrime(n))
        {
            n += 2;
        }
    }
}
```

```
    }
    return n;
}

int hashFunction(int rollNo)
{
    return (rollNo % capacity); //[0--cap-1]
}

};

HashTable::HashTable(int c)
{
    int size = getPrime(c);
    this->capacity = size;
    table = new list<string>[capacity];
}

void HashTable::insertItem(int rollNo, string name)
{
    int index = hashFunction(rollNo);
    table[index].push_back(name);
}

void HashTable::deleteItem(int key, string data)
{
    int index = hashFunction(key);
    list<string>::iterator i;

    for (i = table[index].begin(); i != table[index].end(); i++)
    {
        if (*i == data)
            break;
    }
    if (i != table[index].end())
        table[index].erase(i);
}

void HashTable::searchItem(int key, string data)
{
    int index = hashFunction(key);
    list<string>::iterator i;
    int x = 0;
    bool found = false;
    for (i = table[index].begin(); i != table[index].end(); i++)
    {
        if (*i == data)
        {
            cout << "Found this data at table ["
            << index << "]-[" << x << "]" --> " << data << endl;
            found = true;
            break;
        }
        x++;
    }
}
```

```
    if (!found)
    {
        cout << "Record dosenot exist: " << data << endl;
    }
}
void HashTable::displayHash()
{
    for (int i = 0; i < capacity; i++)
    {
        cout << "table[" << i << " ]";
        for (auto x : table[i])
            cout << " --> " << x;
        cout << endl;
    }
}
int main()
{
    int rollNo[] = { 231, 321, 212, 321, 433, 262 };
    string data[] = {"Anjum", "Asif", "Asad",
                    "Aslam", "Majid", "Ajaz"};

    int size = sizeof(rollNo) / sizeof(rollNo[0]);

    HashTable h(size*2);
    for (int i = 0; i < size; i++)
        h.insertItem(rollNo[i], data[i]);

    h.displayHash();
    cout << endl;
    h.searchItem(321, "Ajmil");
}
```

## STUDENT TASK-01

Use as many as you can queries from quick reference.

## LAB-14

### OBJECTIVE

- Graph
- Adjacency Matrix
- Adjacency List

### PRE-LAB READING ASSIGNMENT

None

### Tools/ Apparatus:

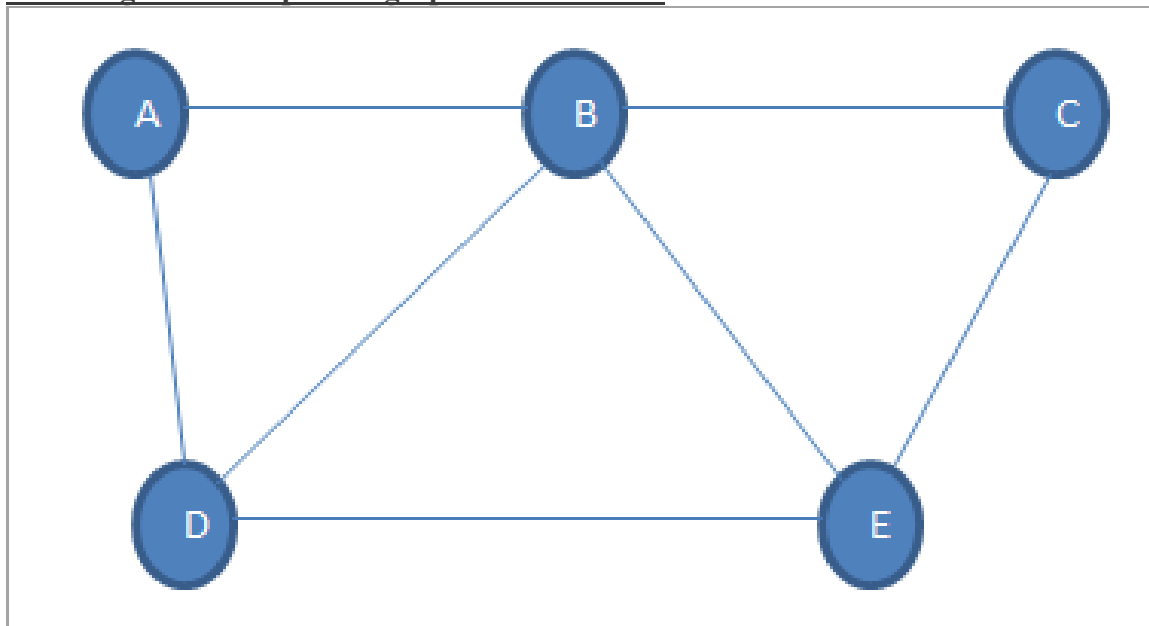
Dev C++, VS Code, Python Tutor

### LAB RELATED CONTENT

## What's Is A Graph In C++?

As stated above, a graph in C++ is a non-linear data structure defined as a collection of vertices and edges.

Following is an example of a graph data structure.



Given above is an example graph G. Graph G is a set of vertices {A,B,C,D,E} and a set of edges {(A,B),(B,C),(A,D),(D,E),(E,C),(B,E),(B,D)}.

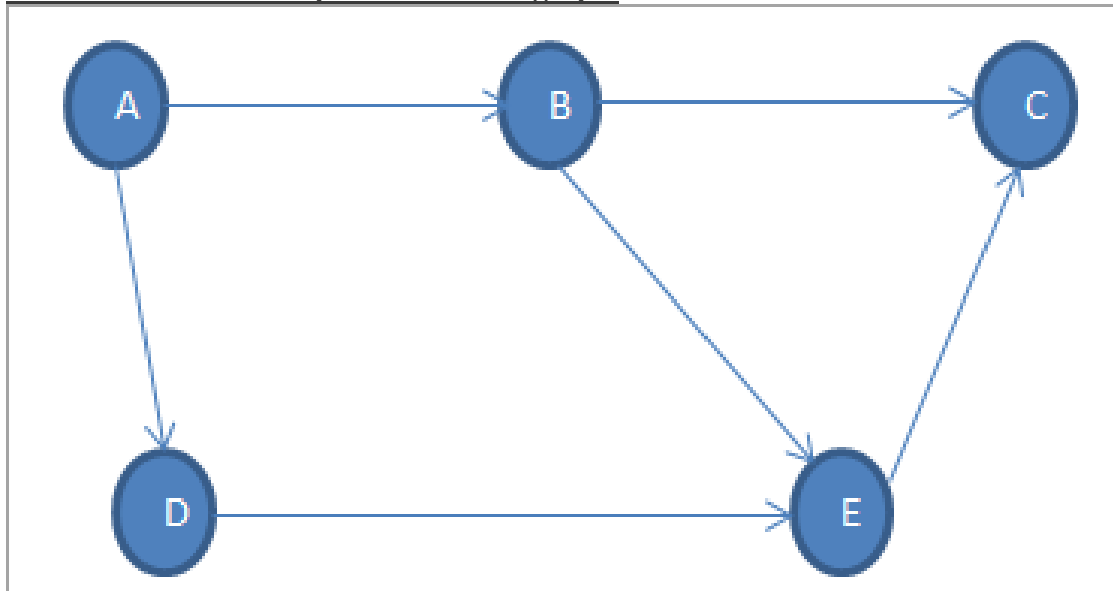
## Types of Graphs – Directed And Undirected Graph

A graph in which the edges do not have directions is called the Undirected graph. The graph shown above is an undirected graph.



A graph in which the edges have directions associated with them is called a Directed graph.

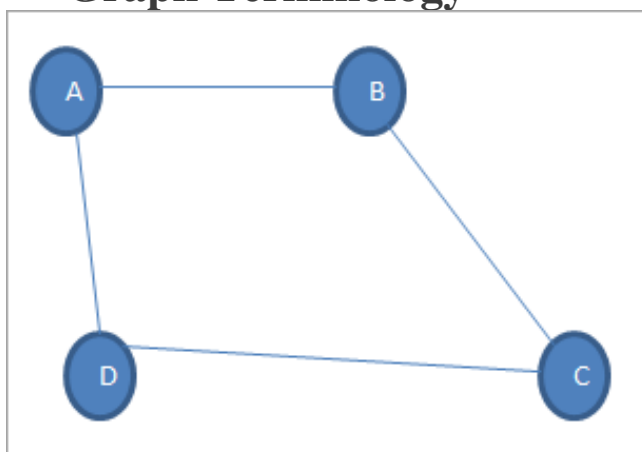
Given below is an example of a directed graph.



In the directed graph shown above, edges form an ordered pair wherein each edge represents a specific path from one vertex to another vertex. The vertex from which the path initiates is called “**Initial Node**” while the vertex into which the path terminates is called the “**Terminal Node**”. Thus in above graph, the set of vertices is {A, B, C, D, E} and the set of edges is {(A,B),(A,D),(B,C),(B,E),(D,E),(E,C)}.

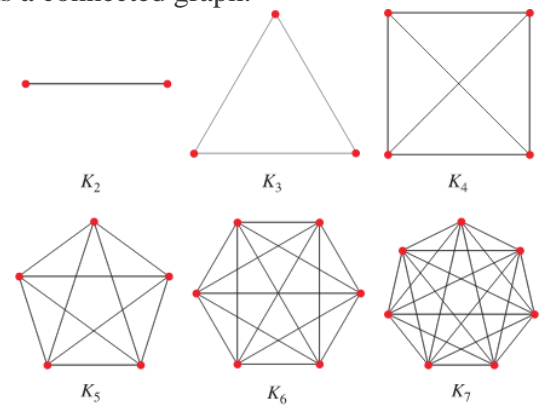
We will discuss the graph terminology or the common terms used in relation to the graph below.

## Graph Terminology

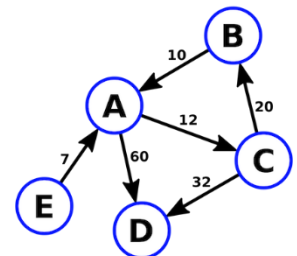


1. **Vertex:** Each node of the graph is called a vertex. In the above graph, A, B, C, and D are the vertices of the graph.
2. **Edge:** The link or path between two vertices is called an edge. It connects two or more vertices. The different edges in the above graph are AB, BC, AD, and DC.

3. **Adjacent node:** In a graph, if two nodes are connected by an edge then they are called adjacent nodes or neighbors. In the above graph, vertices A and B are connected by edge AB. Thus A and B are adjacent nodes.
4. **Degree of the node:** The number of edges that are connected to a particular node is called the degree of the node. In the above graph, node A has a degree 2.
5. **Path:** The sequence of nodes that we need to follow when we have to travel from one vertex to another in a graph is called the path. In our example graph, if we need to go from node A to C, then the path would be A->B->C.
6. **Closed path:** If the initial node is the same as a terminal node, then that path is termed as the closed path.
7. **Simple path:** A closed path in which all the other nodes are distinct is called a simple path.
8. **Cycle:** A path in which there are no repeated edges or vertices and the first and last vertices are the same is called a cycle. In the above graph, A->B->C->D->A is a cycle.
9. **Connected Graph:** A connected graph is the one in which there is a path between each of the vertices. This means that there is not a single vertex which is isolated or without a connecting edge. The graph shown above is a connected graph.
10. **Complete Graph:** A graph in which each node is connected to another is called the Complete graph. If N is the total number of nodes in a graph, then the complete graph contains  $N(N-1)/2$  number of edges.



11. **Weighted graph:** A positive value assigned to each edge indicating its length (distance between the vertices connected by an edge) is called weight. The graph containing weighted edges is called a weighted graph. The weight of an edge e is denoted by  $w(e)$  and it indicates the cost of traversing an edge.



12. **Diagraph:** A digraph is a graph in which every edge is associated with a specific direction and the traversal can be done in specified direction only.

## Graph Representation

The way in which graph data structure is stored in memory is called “representation”. The graph can be stored as a sequential representation or as a linked representation.

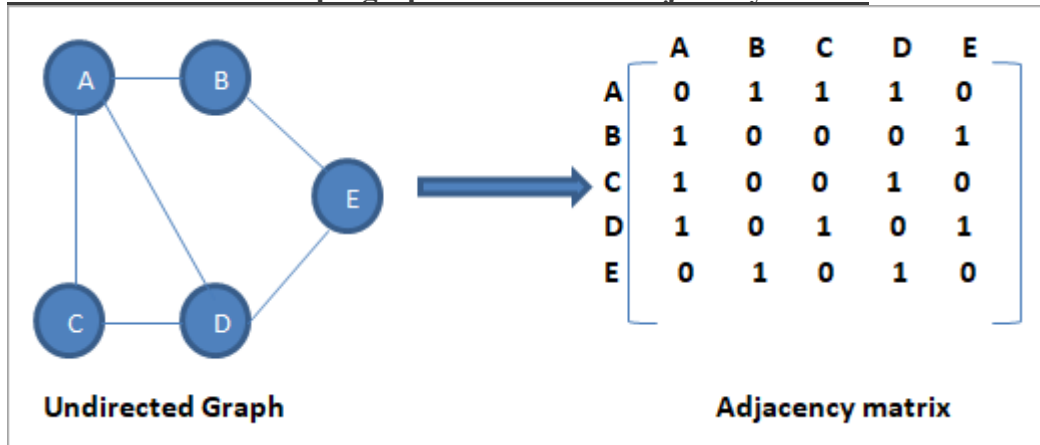
Both these types are described below.

### Sequential Representation

In the sequential representation of graphs, we use the adjacency matrix. An adjacency matrix is a matrix of size  $n \times n$  where  $n$  is the number of vertices in the graph.

The rows and columns of the adjacency matrix represent the vertices in a graph. The matrix element is set to 1 when there is an edge present between the vertices. If the edge is not present then the element is set to 0.

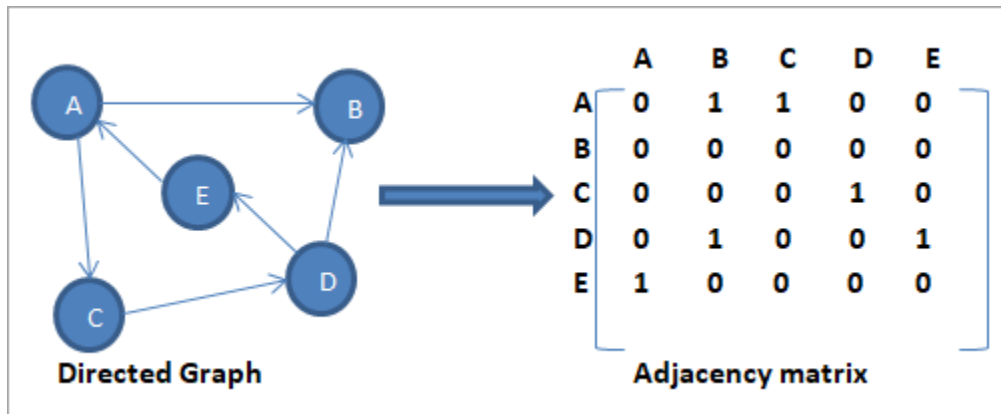
**Given below is an example graph that shows its adjacency matrix.**



We have seen the adjacency matrix for the above graph. Note that since this is an undirected graph, and we can say that the edge is present in both directions. **For Example**, as edge AB is present, we can conclude that edge BA is also present.

In the adjacency matrix, we can see the interactions of the vertices which are matrix elements that are set to 1 whenever the edge is present and to 0 when the edge is absent.

**Now let us see the adjacency matrix of a directed graph.**

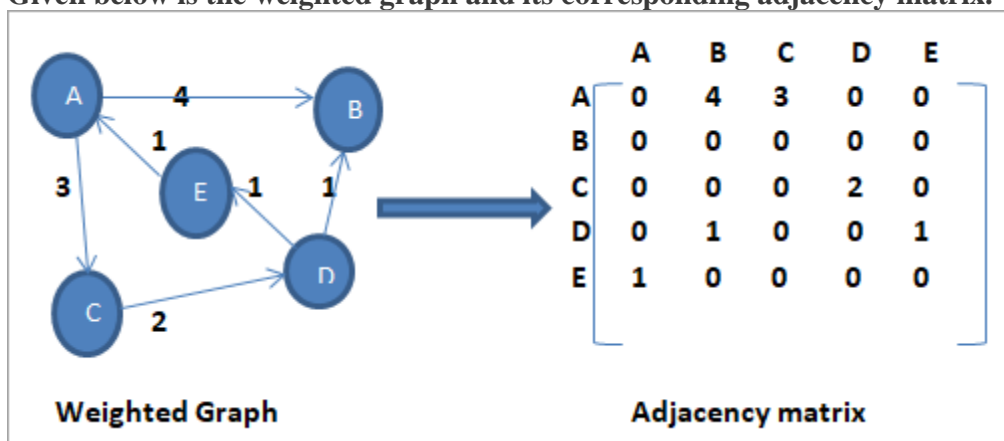


As shown above, the intersection element in the adjacency matrix will be 1 if and only if there is an edge directed from one vertex to another.

In the above graph, we have two edges from vertex A. One edge terminates into vertex B while the second one terminates into vertex C. Thus, in adjacency matrix the intersection of A & B is set to 1 as the intersection of A & C.

Next, we will see the sequential representation for the weighted graph.

**Given below is the weighted graph and its corresponding adjacency matrix.**



We can see that the sequential representation of a weighted graph is different from the other types of graphs. Here, the non-zero values in the adjacency matrix are replaced by the actual weight of the edge.

The edge AB has weight = 4, thus in the adjacency matrix, we set the intersection of A and B to 4. Similarly, all the other non-zero values are changed to their respective weights.

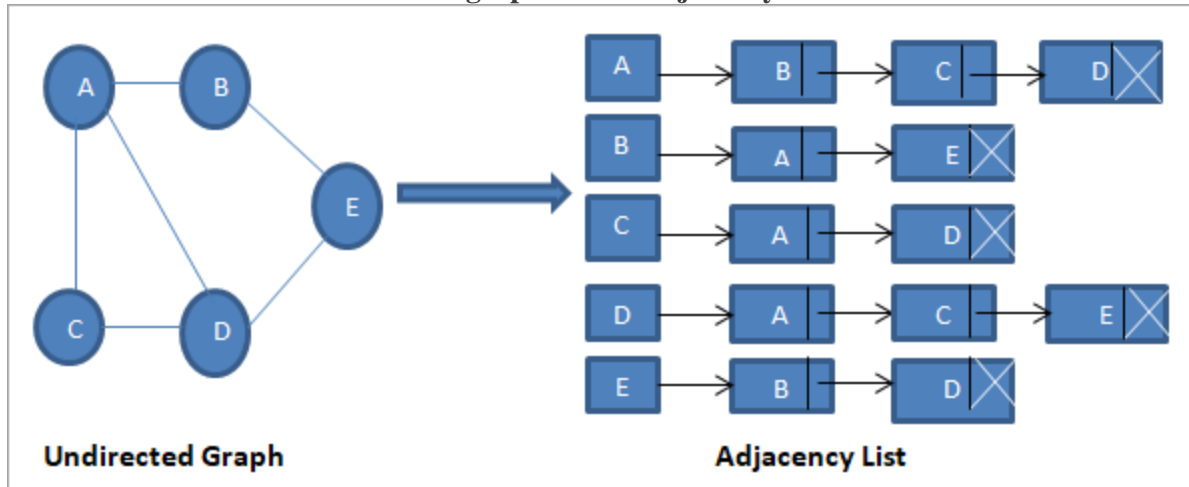
The adjacency list is easier to implement and follow. Traversal i.e. to check if there is an edge from one vertex to another **takes O (1)** time and removing an edge also **takes O (1)**.

Whether the graph is sparse (fewer edges) or dense, it always takes more amount of space.

## Linked Representation

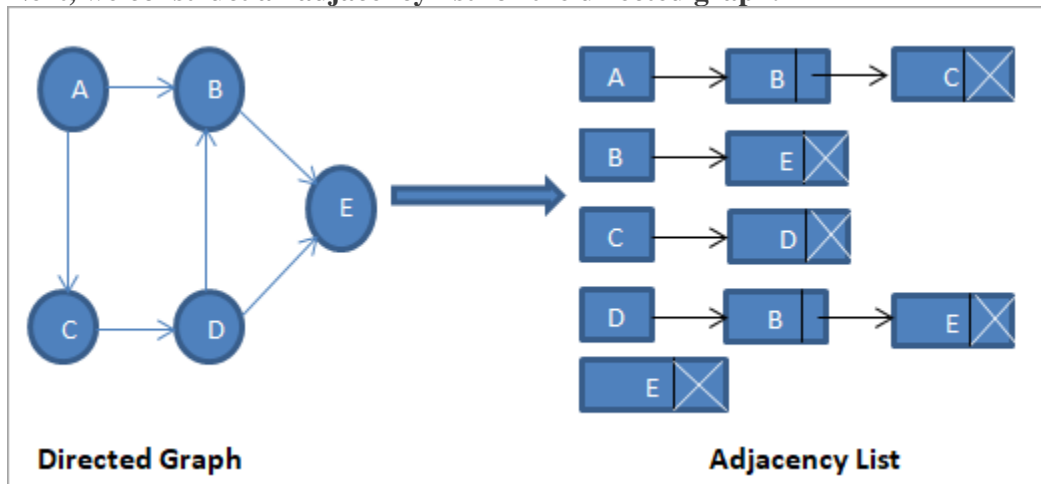
We use the adjacency list for the linked representation of the graph. The adjacency list representation maintains each node of the graph and a link to the nodes that are adjacent to this node. When we traverse all the adjacent nodes, we set the next pointer to null at the end of the list.

Let us first consider an undirected graph and its adjacency list.



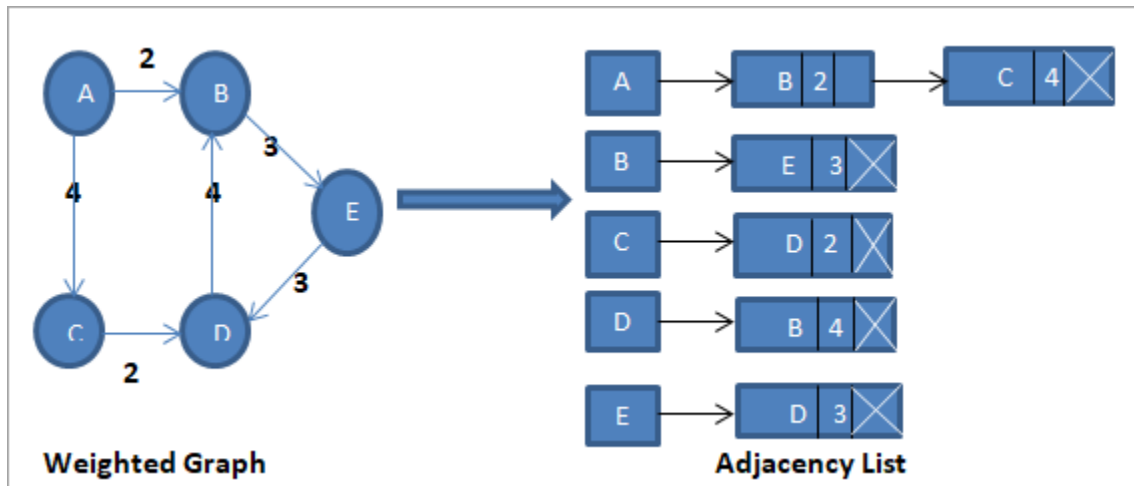
As shown above, we have a linked list (adjacency list) for each node. From vertex A, we have edges to vertices B, C and D. Thus these nodes are linked to node A in the corresponding adjacency list.

Next, we construct an adjacency list for the directed graph.



In the above-directed graph, we see that there are no edges originating from vertex E. Hence the adjacency list for vertex E is empty.

Now let us construct the adjacency list for the weighted graph.



For a weighted graph, we add an extra field in the adjacency list node to denote the weight of the edge as shown above.

Adding vertex in the adjacency list is easier. It also saves space due to the linked list implementation. When we need to find out if there is an edge between one vertex to another, the operation is not efficient.

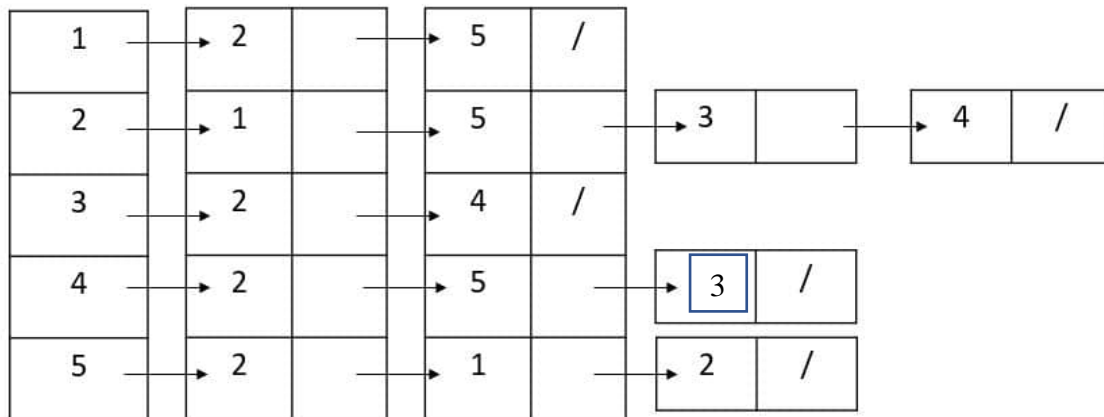
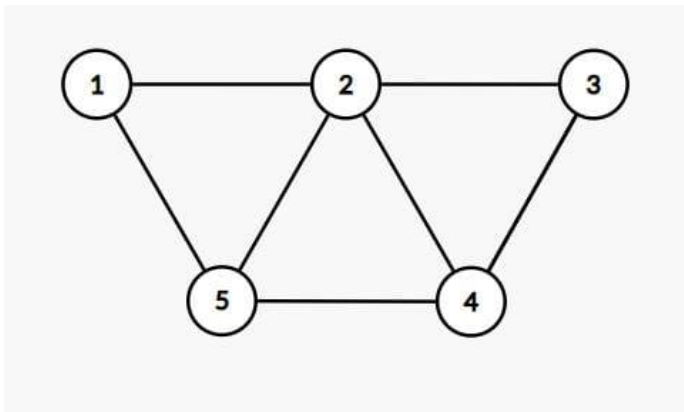
## Basic Operations for Graphs

Following are the basic operations that we can perform on the graph data structure:

- **Add a vertex:** Adds vertex to the graph.
- **Add an edge:** Adds an edge between the two vertices of a graph.
- **Display the graph vertices:** Display the vertices of a graph.

## C++ Graph Implementation Using Adjacency List Example 1

- G is an undirected graph with 5 vertices and 7 edges.
- Adjacency-list representation of G.



- The adjacency-matrix representation of G.

Graph	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

### Code for Adjacency list representation of a graph

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

class Graph
{
public:
    list<int>* adjlist;
    int n; //No of nodes/vertex

    Graph(int v)
    {
        adjlist = new list<int>[v];
        n = v;
    }
    void addEdge(int startEdge, int endEdge, bool bi)
    {
        adjlist[startEdge].push_back(endEdge);
        if (bi)
            adjlist[endEdge].push_back(startEdge);
    }
    void print()
    {
        for (int i = 0; i < n; i++)
        {
            cout << i << "-->";
        }
    }
}
```



```

        for (auto it:adjlist[i])
            cout << it << " ";
        cout << endl;
    }
};

int main()
{
    Graph g(6);
    g.addEdge(1, 2, true);
    g.addEdge(1, 5, true);
    g.addEdge(2, 3, true);
    g.addEdge(2, 4, true);
    g.addEdge(2, 5, true);
    g.addEdge(3, 4, true);
    g.addEdge(5, 4, true);
    g.print();
}

```

D:\Anjum FAST Labs\Data Structure -Lab\lab 14\Graph2.e

```

1-->2 5
2-->1 3 4 5
3-->2 4
4-->2 3 5
5-->1 2 4

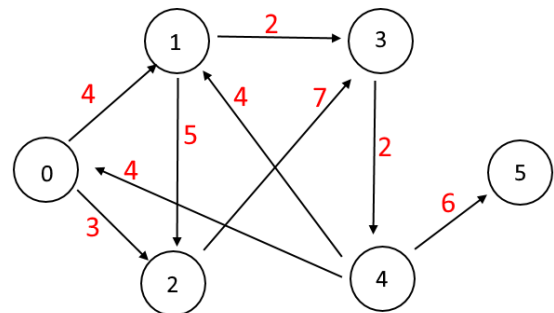
```

## C++ Weighted Graph Implementation Using Adjacency List

```

#include<iostream>
#include<list>
using namespace std;
class node {
public:
    int startV; int endV; int cost;
};
class Graph {
public:
    list<node>* adjList;
    int n;// no of nodes
    Graph(int v) {
        adjList = new list<node>[v];
        n = v;
    }
    void addEdge(int startV, int endV, int cost) {
        node* t = new node();
        t->startV = startV; t->endV = endV; t->cost = cost;
        adjList[startV].push_back(*t);
    }
    void print() {
        for (int i = 0; i < n; i++)
        {
            for (auto it = adjList[i].begin(); it != adjList[i].end(); it++)
            {
                cout << "Adjacent of: " << it->startV << " -> " << it->endV << " (w:" << it->cost << ")" << endl;
            }
            cout << endl;
        }
    }
};

```



Weighted Graph

```

    }
}
void listAdjacentNodes(int n) {
    for (auto it = adjList[n].begin(); it != adjList[n].end(); it++)
        cout << "Adjacent of: " << it->startV
            << " -> " << it->endV << " (w:" << it->cost << ")" <<
endl;
}
};
int main() {
    Graph g(5);
    g.addEdge(0, 1, 3);
    g.addEdge(0, 2, 3);
    g.addEdge(1, 3, 2);
    g.addEdge(1, 2, 5);
    g.addEdge(2, 3, 7);
    g.addEdge(3, 4, 2);
    g.addEdge(4, 5, 6);
    g.addEdge(4, 1, 4);
    g.addEdge(4, 0, 4);
    g.print();
    int n;
    cout << "Enter node number
to see adjacent nodes: ";
    cin >> n; g.listAdjacentNodes(n);}

```

```

D:\Anjum FAST Lab\Data Structure -Lab\lab 14\Graph3.exe
Adjacent of: 0 -> 1 (w:3)
Adjacent of: 0 -> 2 (w:3)

Adjacent of: 1 -> 3 (w:2)
Adjacent of: 1 -> 2 (w:5)

Adjacent of: 2 -> 3 (w:7)

Adjacent of: 3 -> 4 (w:2)

Adjacent of: 4 -> 5 (w:6)
Adjacent of: 4 -> 1 (w:4)
Adjacent of: 4 -> 0 (w:4)

Enter node number to see adjacent nodes:2
Adjacent of: 2 -> 3 (w:7)

```

## C++ Graph Implementation Using Adjacency List Example 2

Now we present a C++ implementation to demonstrate a simple graph using the adjacency list.

Here we are going to display the adjacency list for a weighted directed graph. We have used two structures to hold the adjacency list and edges of the graph. The adjacency list is displayed as (start\_vertex, end\_vertex, weight).

The C++ program is as follows:

```

#include <iostream>
using namespace std;
// stores adjacency list items
struct adjNode
{
    int val, cost;
    adjNode* next;
};
// structure to store edges
struct graphEdge
{
    int start_ver, end_ver, weight;
};
class DiaGraph
{
    // insert new nodes into adjacency list from given graph
    adjNode* getAdjListNode(int value, int weight, adjNode* head)
    {
        adjNode* newNode = new adjNode;
    }
}

```

```
newNode->val = value;
newNode->cost = weight;

newNode->next = head;    // point new node to current head
return newNode;
}
int N; // number of nodes in the graph
public:
adjNode** head;          //adjacency list as array of pointers
// Constructor
DiaGraph(graphEdge edges[], int n, int N)
{
    // allocate new node
    head = new adjNode*[N]();
    this->N = N;
    // initialize head pointer for all vertices
    for (int i = 0; i < N; ++i)
        head[i] = nullptr;
    // construct directed graph by adding edges to it
    for (unsigned i = 0; i < n; i++)
    {
        int start_ver = edges[i].start_ver;
        int end_ver = edges[i].end_ver;
        int weight = edges[i].weight;
        // insert in the beginning
        adjNode* newNode = getAdjListNode(end_ver, weight, head[start_ver]);

        // point head pointer to new node
        head[start_ver] = newNode;
    }
}
// Destructor
~DiaGraph()
{
    for (int i = 0; i < N; i++)
        delete[] head[i];
    delete[] head;
}
};
// print all adjacent vertices of given vertex
void display_AdjList(adjNode* ptr, int i)
{
    while (ptr != nullptr)
    {
        cout << "(" << i << ", " << ptr->val
              << ", " << ptr->cost << ") ";
        ptr = ptr->next;
    }
    cout << endl;
}
// graph implementation
int main()
{
    // graph edges array.
    graphEdge edges[] = {
        // (x, y, w) -> edge from x to y with weight w
        {0,1,2},{0,2,4},{1,4,3},{2,3,2},{3,1,4},{4,3,3}
    }
}
```

```
};  
int N = 6;          // Number of vertices in the graph  
// calculate number of edges  
int n = sizeof(edges) / sizeof(edges[0]);  
// construct graph  
DiaGraph diagraph(edges, n, N);  
// print adjacency list representation of graph  
cout << "Graph adjacency list " << endl << "(start_vertex, end_vertex, weight):" << endl;  
for (int i = 0; i < N; i++)  
{  
    // display adjacent vertices of vertex i  
    display_AdjList(diagraph.head[i], i);  
}  
return 0;  
}
```

Output:

Output:

Graph adjacency list

(start\_vertex, end\_vertex, weight):

(0, 2, 4)

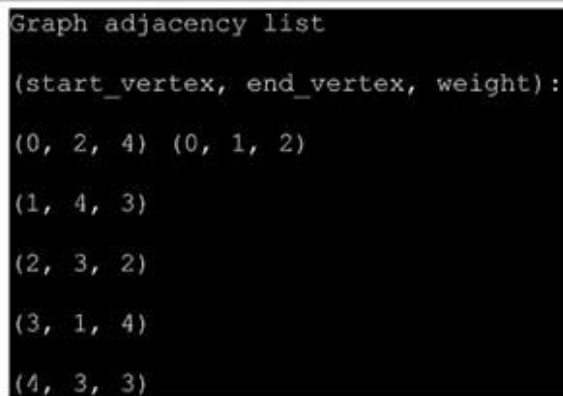
(0, 1, 2)

(1, 4, 3)

(2, 3, 2)

(3, 1, 4)

(4, 3, 3)



```
Graph adjacency list  
(start_vertex, end_vertex, weight):  
(0, 2, 4) (0, 1, 2)  
(1, 4, 3)  
(2, 3, 2)  
(3, 1, 4)  
(4, 3, 3)
```

## Applications Of Graphs

Let us discuss some of the applications of graphs.

- Graphs are used extensively in computer science to depict network graphs, or semantic graphs or even to depict the flow of computation.
- Graphs are widely used in Compilers to depict allocation of resources to processes or to indicate data flow analysis, etc.
- Graphs are also used for query optimization in database languages in some specialized compilers.
- In social networking sites, graphs are main the structures to depict the network of people.
- Graphs are extensively used to build the transportation system especially the road network. A popular example is Google maps that extensively uses graphs to indicate directions all over the world.

### Problem 1 | Implement Adjacency Matrix

Given a undirected Graph of N vertices 1 to N and M edges in form of 2D array. Array [][] whose every row consists of two numbers X and Y which denotes that there is an edge between X and Y, the task is to write C program to create Adjacency Matrix of the given Graph.

Input: N = 8, M = 7, array [][] = {{1, 2}, {2, 3}, {4, 5}, {1, 5}, {6, 1}, {7, 4}, {3, 8}}

Your output will be

0	1	0	0	1	1	0	0
1	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	1	0	1	0
1	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0

### Problem 2 | Implement Adjacency Matrix

The following list of edges are given to you by using this edge list show the adjacency matrix and count that how many edges each node has.

[0,1], [0,6], [0,8], [1,4], [1,6], [1,9], [2,4], [2,6], [3,4], [3,5], [3,8], [4,5], [4,9], [7,8], [7,9]



## Problem 3 | Implement Adjacency List

Implement Adjacency list for undirected Graph of  $N$  vertices 1 to  $N$  and  $M$  edges. Test your code on at least 5 vertices.

Best of luck



## LAB-15

### OBJECTIVE

- Breath First Search
- Depth First Search

### PRE-LAB READING ASSIGNMENT

None

### Tools/ Apparatus:

C++

### LAB RELATED CONTENT

## Depth First Search (DFS) In C++

Unlike BFS in which we explore the nodes breadthwise, in DFS we explore the nodes depth-wise. In DFS we use a stack data structure for storing the nodes being explored. The edges that lead us to unexplored nodes are called 'discovery edges' while the edges leading to already visited nodes are called 'block edges'.

Next, we will see the algorithm and pseudo-code for the DFS technique.

### DFS Algorithm

- **Step 1:** Insert the root node or starting node of a tree or a graph in the stack.
- **Step 2:** Pop the top item from the stack and add it to the visited list.
- **Step 3:** Find all the adjacent nodes of the node marked visited and add the ones that are not yet visited, to the stack.
- **Step 4:** Repeat steps 2 and 3 until the stack is empty.

### Pseudocode

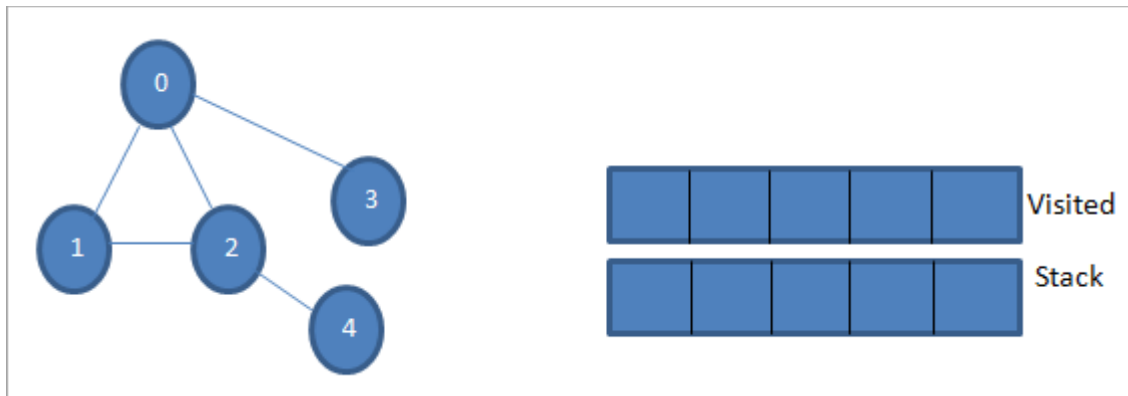
The pseudo-code for DFS is given below.

```
Procedure DFS(G, x)
    G->graph to be traversed
    x->start node
begin
    x.visited = true
    for each v ∈ G.Adj[x]
        if v.visited == false
            DFS(G,v)
    end procedure
init() {
    For each x ∈ G
        x.visited = false
    For each x ∈ G
        DFS(G, x)
}
```

From the above pseudo-code, we notice that the DFS algorithm is called recursively on each vertex to ensure that all the vertices are visited.

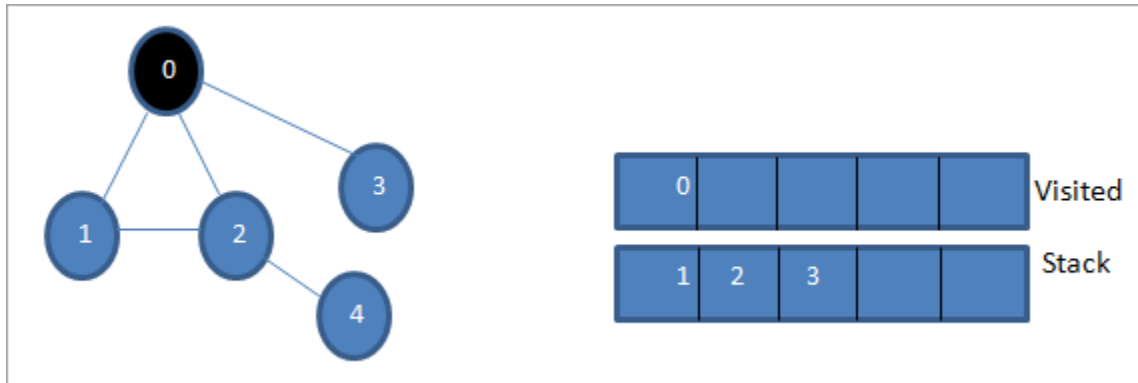
## Traversals With Illustrations

Let us now illustrate the DFS traversal of a graph. For clarity purposes, we will use the same graph that we used in the BFS illustration.

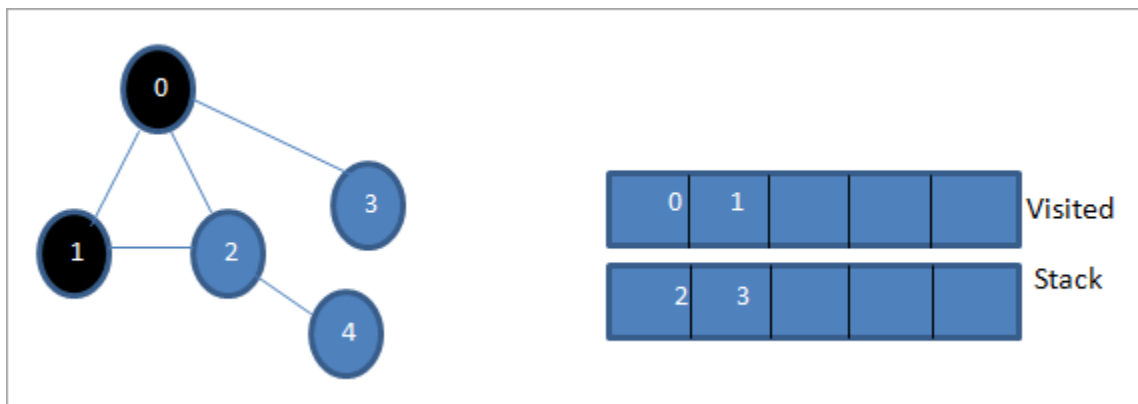


Let 0 be the starting node or source node. First, we mark it as visited and add it to the visited list. Then we push all its adjacent nodes in the stack.

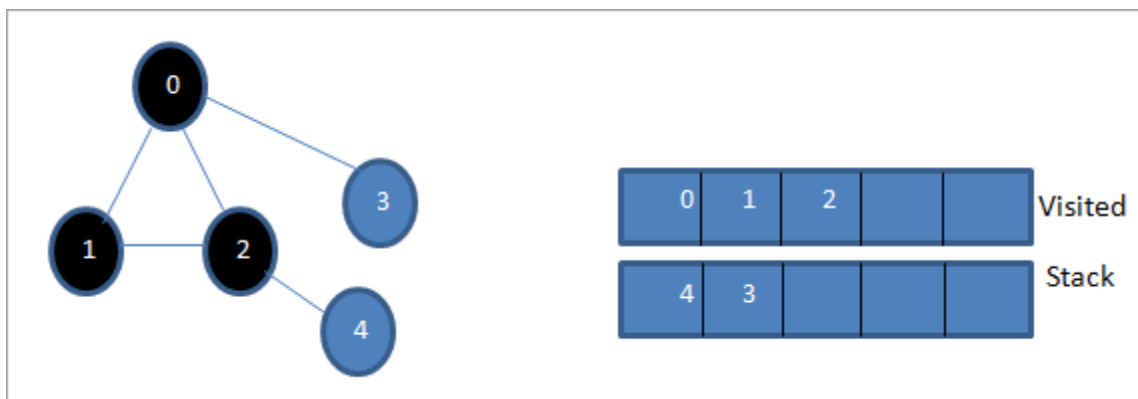




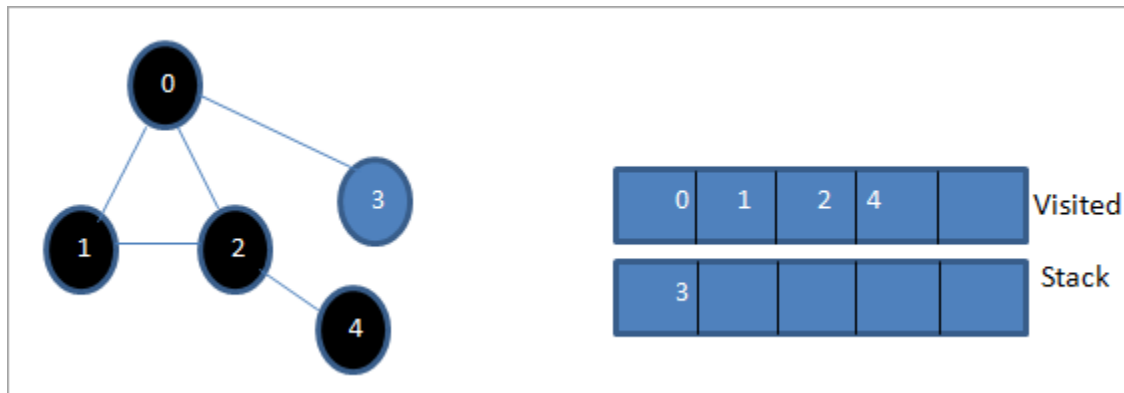
Next, we take one of the adjacent nodes to process i.e. the top of the stack which is 1. We mark it as visited by adding it to the visited list. Now look for the adjacent nodes of 1. As 0 is already in the visited list, we ignore it and we visit 2 which is the top of the stack.



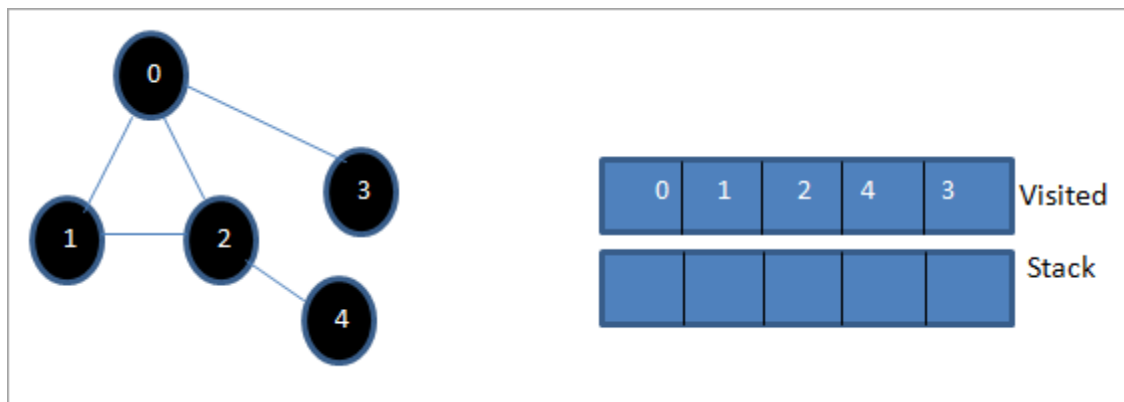
Next, we mark node 2 as visited. Its adjacent node 4 is added to the stack.



Next, we mark 4 which is the top of the stack as visited. Node 4 has only node 2 as its adjacent which is already visited, hence we ignore it.



At this stage, only node 3 is present in the stack. Its adjacent node 0 is already visited, hence we ignore it. Now we mark 3 as visited.



Now the stack is empty and the visited list shows the sequence of the depth-first traversal of the given graph.

If we observe the given graph and the traversal sequence, we notice that for the DFS algorithm, we indeed traverse the graph depth-wise and then backtrack it again to explore new nodes.

## Depth-First Search Implementation

Let's implement the DFS traversal technique using C++.

```
#include <iostream>
#include <list>
using namespace std;
//graph class for DFS traversal
class DFSGraph
{
    int V; // No. of vertices
    list<int>* adjList; // adjacency list
    void DFS_util(int v, bool visited[]); // A function used by DFS
public:
    // class Constructor
    DFSGraph(int V)
    {
        this->V = V;
```

```
    adjList = new list<int>[V];
}
// function to add an edge to graph
void addEdge(int v, int w)
{
    adjList[v].push_back(w); // Add w to v's list.
}

void DFS(); // DFS traversal function
};
void DFSGraph::DFS_util(int v, bool visited[])
{
    // current node v is visited
    visited[v] = true;
    cout << v << " ";

    // recursively process all the adjacent vertices of the node
    list<int>::iterator i;
    for (i = adjList[v].begin(); i != adjList[v].end(); ++i)
        if (!visited[*i])
            DFS_util(*i, visited);
}

// DFS traversal
void DFSGraph::DFS()
{
    // initially none of the vertices are visited
    bool* visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // explore the vertices one by one by recursively calling DFS_util
    for (int i = 0; i < V; i++)
        if (visited[i] == false)
            DFS_util(i, visited);
}

int main()
{
    // Create a graph
    DFSGraph gdfs(5);
    gdfs.addEdge(0, 1);
    gdfs.addEdge(0, 2);
    gdfs.addEdge(0, 3);
    gdfs.addEdge(1, 2);
    gdfs.addEdge(2, 4);
    gdfs.addEdge(3, 3);
    gdfs.addEdge(4, 4);

    cout << "Depth-first traversal for the given graph:" << endl;
    gdfs.DFS();

    return 0;
}
```

**Output:**

Depth-first traversal for the given graph:

0 1 2 4 3

We have once again used the graph in the program that we used for illustration purposes. We see that the DFS algorithm (separated into two functions) is called recursively on each vertex in the graph in order to ensure that all the vertices are visited.

## Runtime Analysis

The time complexity of DFS is the same as BFS i.e.  $O(|V|+|E|)$  where  $V$  is the number of vertices and  $E$  is the number of edges in a given graph.

Similar to BFS, depending on whether the graph is scarcely populated or densely populated, the dominant factor will be vertices or edges respectively in the calculation of time complexity.

## Iterative DFS

The implementation shown above for the DFS technique is recursive in nature and it uses a function call stack. We have another variation for implementing DFS i.e. “**Iterative depth-first search**”. In this, we use the explicit stack to hold the visited vertices.

We have shown the implementation for iterative DFS below. Note that the implementation is the same as BFS except the factor that we use the stack data structure instead of a queue.

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

// graph class
class Graph
{
    int V;    // No. of vertices
    list<int>* adjList;    // adjacency lists
public:
    Graph(int V) //graph Constructor
    {
        this->V = V;
        adjList = new list<int>[V];
    }
    void addEdge(int v, int w) // add an edge to graph
    {
        adjList[v].push_back(w); // Add w to v's list.
    }
    void DFS();    // DFS traversal

    // utility function called by DFS
    void DFSUtil(int s, vector<bool> &visited);
};

//traverses all not visited vertices reachable from start node s
void Graph::DFSUtil(int s, vector<bool> &visited)
{
    // stack for DFS
    stack<int> dfsstack;
    // current source node inside stack
    dfsstack.push(s);
```

```
while (!dfsstack.empty())
{
    // Pop a vertex
    s = dfsstack.top();
    dfsstack.pop();

    // display the item or node only if its not visited
    if (!visited[s])
    {
        cout << s << " ";
        visited[s] = true;
    }

    // explore all adjacent vertices of popped vertex.
    //Push the vertex to the stack if still not visited
    for (auto i = adjList[s].begin(); i != adjList[s].end(); ++i)
        if (!visited[*i])
            dfsstack.push(*i);
    }
}

// DFS
void Graph::DFS()
{
    // initially all vertices are not visited
    vector<bool> visited(V, false);

    for (int i = 0; i < V; i++)
        if (!visited[i])
            DFSUtil(i, visited);
}

//main program
int main()
{
    Graph gidfs(5); //create graph
    gidfs.addEdge(0, 1);
    gidfs.addEdge(0, 2);
    gidfs.addEdge(0, 3);
    gidfs.addEdge(1, 2);
    gidfs.addEdge(2, 4);
    gidfs.addEdge(3, 3);
    gidfs.addEdge(4, 4);

    cout << "Output of Iterative Depth-first traversal:\n";
    gidfs.DFS();

    return 0;
}
```

**Output:**

Output of Iterative Depth-first traversal:

0 3 2 4 1

We use the same graph that we used in our recursive implementation. The difference in output is because we use the stack in the iterative implementation. As the stacks follow LIFO order, we get a

different sequence of DFS. To get the same sequence, we might want to insert the vertices in the reverse order.

## BFS Vs DFS

So far we have discussed both the traversal techniques for graphs i.e. BFS and DFS.

Now let us look into the differences between the two.

BFS	DFS
Stands for “Breadth-first search”	Stands for “Depth-first search”
The nodes are explored breadth wise level by level.	The nodes are explored depth-wise until there are only leaf nodes and then backtracked to explore other unvisited nodes.
BFS is performed with the help of queue data structure.	DFS is performed with the help of stack data structure.
Slower in performance.	Faster than BFS.
Useful in finding the shortest path between two nodes.	Used mostly to detect cycles in graphs.

## Applications Of DFS

- **Detecting Cycles In The Graph:** If we find a back edge while performing DFS in a graph then we can conclude that the graph has a cycle. Hence DFS is used to detect the cycles in a graph.
- **Pathfinding:** Given two vertices x and y, we can find the path between x and y using DFS. We start with vertex x and then push all the vertices on the way to the stack till we encounter y. The contents of the stack give the path between x and y.
- **Minimum Spanning Tree And Shortest Path:** DFS traversal of the un-weighted graph gives us a minimum spanning tree and shortest path between nodes.
- **Topological Sorting:** We use topological sorting when we need to schedule the jobs from the given dependencies among jobs. In the computer science field, we use it mostly for resolving symbol dependencies in linkers, data serialization, instruction scheduling, etc. DFS is widely used in Topological sorting.

## Conclusion

In the last couple of tutorials, we explored more about the two traversal techniques for graphs i.e. BFS and DFS. We have seen the differences as well as the applications of both the techniques. BFS and DFS basically achieve the same outcome of visiting all nodes of a graph but they differ in the order of the output and the way in which it is done.

We have also seen the implementation of both techniques. While BFS uses a queue, DFS makes use of stacks to implement the technique. With this, we conclude the tutorial on traversal techniques for graphs. We can also use BFS and DFS on trees.

## Breadth First Search (BFS) C++ Program To Traverse A Graph Or Tree

**This Tutorial Covers Breadth First Search in C++ in Which The Graph or Tree is Traversed Breadthwise. You will Also Learn BFS Algorithm & Implementation:**

This explicit C++ tutorial will give you a detailed explanation of traversal techniques that can be performed on a tree or graph.

Traversal is the technique using which we visit each and every node of the graph or a tree. **There are two standard methods of traversals.**

- Breadth-first search(BFS)
- Depth-first search(DFS)

### Breadth First Search (BFS) Technique In C++

In this tutorial, we will discuss in detail the breadth-first search technique.

In the breadth-first traversal technique, the graph or tree is traversed breadth-wise. This technique uses the queue data structure to store the vertices or nodes and also to determine which vertex/node should be taken up next.

Breadth-first algorithm starts with the root node and then traverses all the adjacent nodes. Then, it selects the nearest node and explores all the other unvisited nodes. This process is repeated until all the nodes in the graph are explored.

### Breadth-First Search Algorithm

Given below is the algorithm for BFS technique.

Consider G as a graph which we are going to traverse using the BFS algorithm.

Let S be the root/starting node of the graph.

- **Step 1:** Start with node S and enqueue it to the queue.
- **Step 2:** Repeat the following steps for all the nodes in the graph.
- **Step 3:** Dequeue S and process it.
- **Step 4:** Enqueue all the adjacent nodes of S and process them.
- [END OF LOOP]
- **Step 6:** EXIT

### Pseudocode

**The pseudo-code for the BFS technique is given below.**

Procedure BFS (G, s)

### Procedure BFS (G, s)

G is the graph and s is the source node

begin

let q be queue to store nodes

q.enqueue(s) //insert source node in the queue

mark s as visited.

while (q is not empty)

//remove the element from the queue whose adjacent nodes are to be processed

n = q.dequeue( )

//processing all the adjacent nodes of n

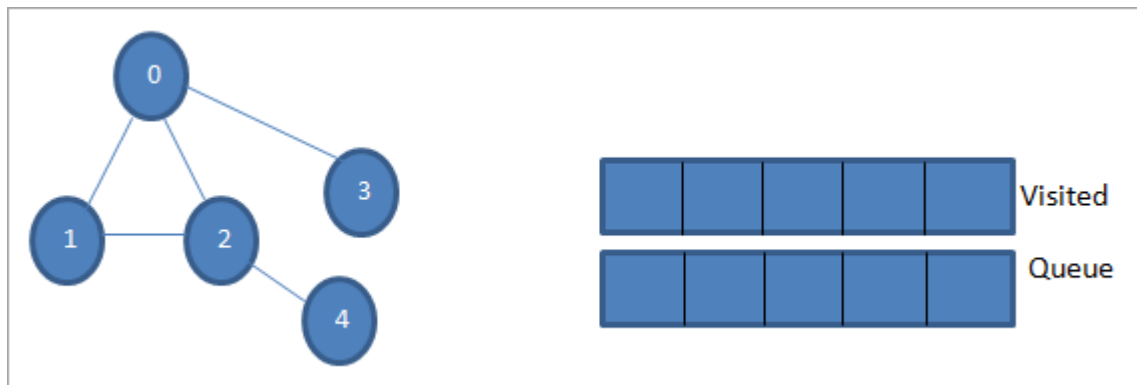
for all neighbors m of n in Graph G if w is not visited

q.enqueue (m) //Stores m in Q to in turn visit its adjacent nodes

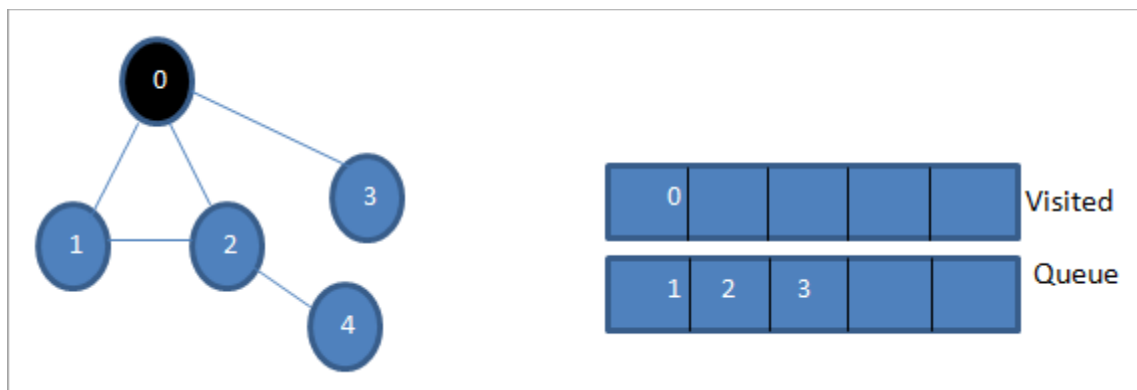
mark m as visited.

end

## Traversals With Illustrations

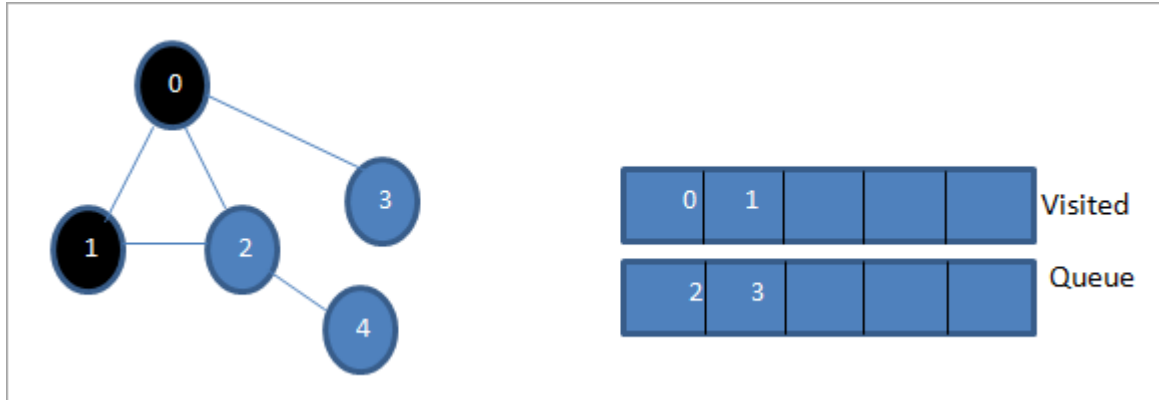


Let 0 be the starting node or source node. First, we enqueue it in the visited queue and all its adjacent nodes in the queue.

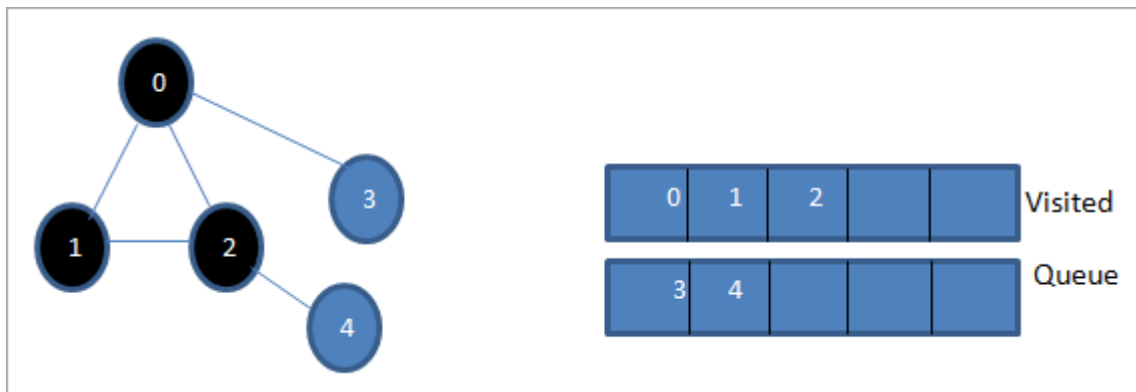




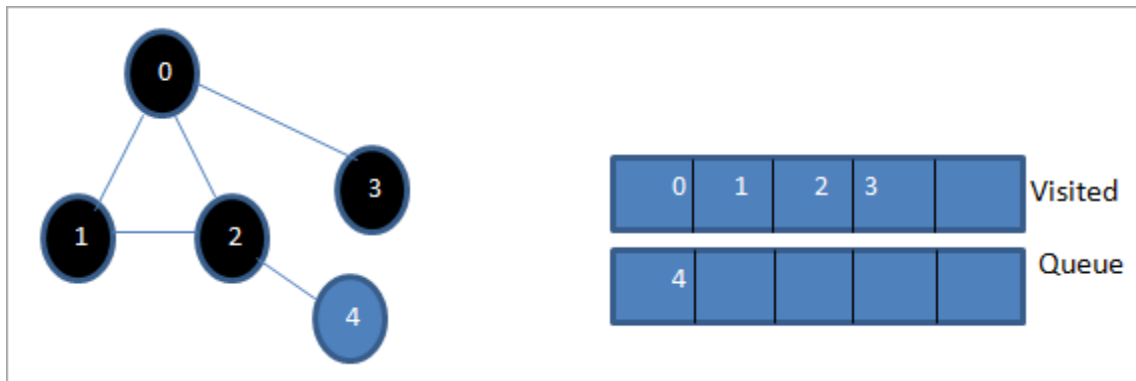
Next, we take one of the adjacent nodes to process i.e. 1. We mark it as visited by removing it from the queue and put its adjacent nodes in the queue (2 and 3 already in queue). As 0 is already visited, we ignore it.



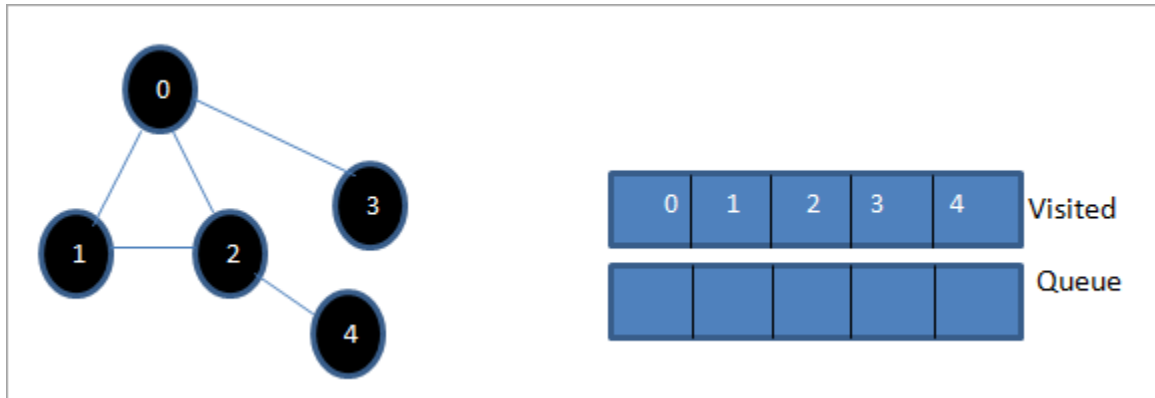
Next, we dequeue node 2 and mark it as visited. Then, its adjacent node 4 is added to the queue.



Next, we dequeue 3 from the queue and mark it as visited. Node 3 has only one adjacent node i.e. 0 which is already visited. Hence, we ignore it.



At this stage, only node 4 is present in the queue. Its adjacent node 2 is already visited, hence we ignore it. Now we mark 4 as visited.



Next, the sequence present in the visited list is the breadth-first traversal of the given graph.

If we observe the given graph and the traversal sequence, we can notice that for the BFS algorithm, we indeed traverse the graph breadth-wise and then go to the next level.

## BFS Implementation

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

// a directed graph class
class DiGraph
{
    int V;    // No. of vertices

    // Pointer to an array containing adjacency lists
    list<int>* adjList;
public:
    DiGraph(int V); // Constructor

    // add an edge from vertex v to w
    void addEdge(int v, int w);

    // BFS traversal sequence starting with s -> starting node
    void BFS(int s);
};

DiGraph::DiGraph(int V)
{
    this->V = V;
    adjList = new list<int>[V];
}

void DiGraph::addEdge(int v, int w)
{
    adjList[v].push_back(w); // Add w to v's list.
}

void DiGraph::BFS(int s)
{
    // initially none of the vertices is visited
    bool* visited = new bool[V];
```

```
for (int i = 0; i < V; i++)
    visited[i] = false;

// queue to hold BFS traversal sequence
list<int> queue;

// Mark the current node as visited and enqueue it
visited[s] = true;
queue.push_back(s);

// iterator 'i' to get all adjacent vertices
list<int>::iterator i;

while (!queue.empty())
{
    // dequeue the vertex
    s = queue.front();
    cout << s << " ";
    queue.pop_front();

    // get all adjacent vertices of popped vertex and process each if not already visited
    for (i = adjList[s].begin(); i != adjList[s].end(); ++i)
    {
        if (!visited[*i])
        {
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
}

// main program
int main()
{
    // create a graph
    DiGraph dg(5);
    dg.addEdge(0, 1);
    dg.addEdge(0, 2);
    dg.addEdge(0, 3);
    dg.addEdge(1, 2);
    dg.addEdge(2, 4);
    dg.addEdge(3, 3);
    dg.addEdge(4, 4);

    cout << "Breadth First Traversal for given graph (with 0 as starting node): " << endl;
    dg.BFS(0);

    return 0;
}
```

### Output:

Breadth-First Traversal for the given graph (with 0 as starting node):

0 1 2 3 4

We have implemented the BFS in the above program. Note that the graph is in the form of an adjacency list and then we use an iterator to iterate through the list and perform BFS.

We have used the same graph that we used for illustration purposes as an input to the program to compare the traversal sequence.

## Runtime Analysis

If  $V$  is the number of vertices and  $E$  is the number of edges of a graph, then the time complexity for BFS can be expressed as  $O(|V|+|E|)$ . Having said this, it also depends on the data structure that we use to represent the graph.

If we use the adjacency list (like in our implementation), then the time complexity is  $O(|V|+|E|)$ .

If we use the adjacency matrix, then the time complexity is  $O(V^2)$ .

Apart from the data structures used, there is also a factor of whether the graph is densely populated or sparsely populated.

When the number of vertices exceeds the number of edges, then the graph is said to be sparsely connected as there will be many disconnected vertices. In this case, the time complexity of the graph will be  $O(V)$ .

On the other hand, sometimes the graph may have a higher number of edges than the number of vertices. In such a case, the graph is said to be densely populated. The time complexity of such a graph is  $O(E)$ .

To conclude, what the expression  $O(|V|+|E|)$  means is depending on whether the graph is densely or sparsely populated, the dominating factor i.e. edges or vertices will determine the time complexity of the graph accordingly.

## Applications Of BFS Traversal

- **Garbage Collection:** The garbage collection technique, “Cheney’s algorithm” uses breadth-first traversal for copying garbage collection.
- **Broadcasting In Networks:** A packet travels from one node to another using the BFS technique in the broadcasting network to reach all nodes.
- **GPS Navigation:** We can use BFS in GPS navigation to find all the adjacent or neighboring location nodes.
- **Social Networking Websites:** Given a person ‘P’, we can find all the people within a distance, ‘d’ from p using BFS till the d levels.
- **Peer To Peer Networks:** Again BFS can be used in peer to peer networks to find all the adjacent nodes.
- **Shortest Path And Minimum Spanning Tree In The Un-weighted Graph:** BFS technique is used to find the shortest path i.e. the path with the least number of edges in the un-weighted graph. Similarly, we can also find a minimum spanning tree using BFS in the un-weighted graph.



## Conclusion

The breadth-first search technique is a method that is used to traverse all the nodes of a graph or a tree in a breadth-wise manner.

This technique is mostly used to find the shortest path between the nodes of a graph or in applications that require us to visit every adjacent node like in networks

### **Problem 1 | Dijkstra Algorithm**

Explain Dijkstra Algorithm in detail, why we use it.

### **Problem 2 | Bellman Ford Algorithm**

Explain **Bellman Ford** Algorithm in detail, why we use it.