METHODIST

Estd:2008

COLLEGE OF ENGINEERING AND TECHNOLOGY

(Affiliated to Osmania University & Approved by AICTE, New Delhi)

LABORATORY MANUAL

DATA STRUCTURES AND ALGORITHM LAB

BE III Semester (AICTE Model Curriculum): 2020-21

NAME: ______________________________	
ROLL NO: ______________________________	
BRANCH: ______________	**SEM:** ______________________

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERNG

Empower youth- Architects of Future World

Estd:2008

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

VISION

To produce ethical, socially conscious and innovative professionals who would contribute to sustainable technological development of the society.

MISSION

To impart quality engineering education with latest technological developments and interdisciplinary skills to make students succeed in professional practice.

To encourage research culture among faculty and students by establishing state of art laboratories and exposing them to modern industrial and organizational practices.

To inculcate humane qualities like environmental consciousness, leadership, social values, professional ethics and engage in independent and lifelong learning for sustainable contribution to the society.

METHODIST

Estd:2008 COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT
OF
COMPUTER SCIENCE AND ENGINEERING

LABORATORY MANUAL

DATA STRUCTURES AND ALGORITHM LAB

Prepared
By

Mrs. E. Shailaja,

Associate Professor.

Estd:2008

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION & MISSION

VISION

To become a leader in providing Computer Science & Engineering education with emphasis on knowledge and innovation.

MISSION

- To offer flexible programs of study with collaborations to suit industry needs.
- To provide quality education and training through novel pedagogical practices.
- To expedite high performance of excellence in teaching, research and innovations.
- To impart moral, ethical values and education with social responsibility.

Estd:2008

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM EDUCATIONAL OBJECTIVES

After 3-5 years of graduation, the graduates will be able to

PEO1: Apply technical concepts, Analyze, Synthesize data to Design and create novel products and solutions for the real life problems.

PEO2: Apply the knowledge of Computer Science Engineering to pursue higher education with due consideration to environment and society.

PEO3: Promote collaborative learning and spirit of team work through multidisciplinary projects

PEO4: Engage in life-long learning and develop entrepreneurial skills.

Estd:2008

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PROGRAM OUTCOMES

Engineering graduates will be able to:

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the Engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES

At the end of 4 years, Computer Science and Engineering graduates at MCET will be able to:

PSO1: Apply the knowledge of Computer Science and Engineering in various domains like networking and data mining to manage projects in multidisciplinary environments.

PSO2: Develop software applications with open-ended programming environments**.**

PSO3: Design and develop solutions by following standard software engineering principles and implement by using suitable programming languages and platforms

Course Code	Course Title						Core/Elective
PC252CS	**Data Structures and Algorithm Lab**						**Core**
Prerequisite	Contact Hours per Week				CIE	SEE	Credits
	L	T	D	P			
-	**-**	**-**	**-**	**2**	**25**	**50**	**1**

Course Objectives

- Design and construct simple programs by using the concepts of structures as abstract datatype.
- To have a broad idea about how to use pointers in the implement of datastructures.
- To enhance programming skills while improving their practical knowledge in datastructures.
- To strengthen the practical ability to apply suitable data structure for real timeapplications.

Course Outcomes

After completing this course, the student will be able to:

1. Implement the abstract data type and reusability of a particular datastructure.
2. Implement linear data structures such as stacks, queues using array and linkedlist.
3. Understand and implements non-linear data structures such as trees,graphs.
4. Implement various kinds of searching, sorting and traversal techniques and know when to choose whichtechnique.
5. Understanding and implementing hashingtechniques.
6. Decide a suitable data structure and algorithm to solve a real worldproblem.

Programming Exercise using C++:

1. C++ Programs to implement: Classes, Constructors, Inheritance, Polymorphism, Dynamic Memory Allocation, Class Templates, ExceptionHandling.
2. Implementation of Stacks, Queues (using both arrays and linkedlists).
3. Implementation of Singly Linked List, Doubly Linked List and CircularList.
4. Implementation of Infix to Postfix conversion and evaluation of postfixexpression.
5. Implementation of Polynomial arithmetic using linkedlist.
6. Implementation of Linear search and BinarySearch
7. Implementation of HashingTechnique
8. Implementation of Binary Tree and Binary tree traversal techniques (inorder, preorder, postorder, level-order)
9. Implementation of Binary search tree and itsoperations
10. Implementation of Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Quick Sort, HeapSort
11. Implementation of operations on AVLtrees.
12. Implementation of Graph SearchMethods.

Note: It is recommended to use a debugging tool to debug the programs.

METHODIST

Estd:2008

COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Outcomes (CO's):

SUBJECT NAME: DATA STRUCTURES AND ALGORITHM LAB **CODE : PC252CS**

SEMESTER : III

CO No.	Course Outcomes	Taxonomy Level
PC252CS.1	Understand and Implement the abstract data type and reusability of a particular data structure.	APPLYING-3
PC252CS.2	Implement linear data structures such as stacks, queues using array and linked list.	APPLYING-3
PC252CS.3	Understand and implements non-linear data structures such as trees, graphs.	APPLYING-3
PC252CS.4	Implement various kinds of searching, sorting and traversal techniques and know when to choose which technique.	APPLYING-3
PC252CS.5	Understanding and implementing hashing techniques.	APPLYING-3
PC252CS.6	Decide a suitable data structure and algorithm to solve a real world problem.	EVALUATING-5

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

Estd:2008

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to starting time), those who come after 5 minutes will not be allowed into the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab with the program / experiment details.
3. Student should enter into the laboratory with:
   a. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.
   b. Laboratory Record updated up to the last session experiments.
   c. Formal dress code and Identity card.
4. Sign in the laboratory login register, write the TIME-IN, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.
8. Students / Faculty must keep their mobile phones in SWITCHED OFF mode during the lab sessions. Misuse of the equipment, misbehaviours with the staff and systems etc., will attract severe punishment.
9. Students must take the permission of the faculty in case of any urgency to go out. If anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.
10. Students should SHUT DOWN the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Head of the Department **Principal**

METHODIST

Estd:2008 COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CODE OF CONDUCT FOR THE LABORATORY

- All students must observe the dress code while in the laboratory
- Footwear is NOT allowed
- Foods, drinks and smoking are NOT allowed
- All bags must be left at the indicated place
- The lab timetable must be strictly followed
- Be PUNCTUAL for your laboratory session
- All programs must be completed within the given time
- Noise must be kept to a minimum
- Workspace must be kept clean and tidy at all time
- All students are liable for any damage to system due to their own negligence
- Students are strictly PROHIBITED from taking out any items from the laboratory
- Report immediately to the lab programmer if any damages to equipment

BEFORE LEAVING LAB:

- Arrange all the equipment and chairs properly.
- Turn off / shut down the systems before leaving.
- Please check the laboratory notice board regularly for updates.

Lab In – charge

METHODIST

COLLEGE OF ENGINEERING AND TECHNOLOGY

Estd:2008

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LIST OF EXPERIMENTS

ADDITIONAL EXPERIMENTS

SI.No.	Name of the Experiment	Date of Experiment	Date of Submission	Page No	Faculty Signature
13	Write a C++ Program to implement balanced parenthesis using stacks.			123	
14	Write a C++ Program to sort a singly linked list.			125	

Program 1:

To implement Classes, Constructors, Inheritance, Polymorphism, Dynamic Memory Allocation, Class Templates, Exception Handling.

Aim : **C++ Programs to implement: Classes, Constructors, Inheritance, Polymorphism, Dynamic Memory Allocation, Class Templates, Exception Handling.**

Description :

1.A)Classes: A class in C++ is the building block, that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

Syntax to Define Class in C++

```
class className {
// some data
// some functions
};
```

C++ Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.To use the data and access functions defined in the class, we need to create objects.

Syntax to Define Object in C++

```
classNameobjectVariableName;
```

Aim :Write a C++ program to add two times using classes

```
#include<iostream.h>
class Time
{
int hours;
int minutes;
int seconds;
public:
void getTime();
void displayTime();
void addTime(Time t1, Time t2);
}
void Time::getTime()
{
```

```
cout<<"Enter hours"<<endl;
cin>>hours;
cout<<"Enter minutes"<<endl;
cin>>minutes;
cout<<"Enter seconds";
cin>>seconds;
}
void Time::displayTime()
{
cout<<"HH:MM:SS"<<hours<<":"<<minutes<<":"<<seconds;
}
void Time::addTime(Time t1,Time t2)
{

this-> seconds=t1.seconds+t2.seconds;
this->minutes=t1.minutes+t2.minutes+this->seconds/60;
this->hours=t1.hours+t2.hours+this->minutes/60;
this->minutes=this->minutes%60;
this->seconds=this -> seconds%60;
}
int main()
{
Time t1,t2,t3;
t1.getTime();
t2.getTime();
t3.addTime(t1,t2);
t3.displayTime();
return 0;
}
```

Expected Output:

```
Enter hours
3
Enter minutes
45
Enter seconds34
Enter hours
4
Enter minutes
56
Enter seconds46
HH:MM:SS8:42:20
```

1.B)Constructors: A constructor is a special member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) is created.

```
class MyClass{
  public:
  MyClass() {
```

```
  cout<< "Hello World!";
   }
};

int main() {
MyClassmyObj;    // Create an object of MyClass (this will call the constructor)
 return 0;
}
```

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a defaultconstructor for us (expects no parameters and has an empty body).

Types of Constructors

1. Default Constructors: Default constructor is the constructor whichdoesn't take any argument. It has no parameters.

2. Parameterized Constructors: It is possible to pass arguments toconstructors. Typically, these arguments help initialize an object when it iscreated. To create a parameterized constructor, simply add parameters to
it the way you would to any other function. When you define theconstructor's body, use the parameters to initialize the object.

3. Copy Constructor: A copy constructor is a member function whichinitializes an object using another object of the same class.

Aim:Write a Program to create a default constructor, Parameterized constructor, copy constructor

```
#include<iostream>
using namespace std;
class rectangle
{
int l;
int b;
public:
rectangle() // Default Constructor
{
l=5;
b=6;
}
rectangle(int l1,int b1) //Parameterized Constructor
{
l=l1;
b=b1;
}
rectangle(rectangle &r1) //copy constructor
{
```

```
l=r1.l;
b=r1.b;

}
int area()
{
return(l*b);
}
};
int main()
{
rectangle r;          //default constructor is called
rectangle r1(6,3); // parameterized constructor is called
rectangle r2(r1); //copy constructor is called
cout<<"Area of rectangle is"<<r.area();
cout<<"\n Area of rectangle is"<<r1.area();
cout<<"\n Area of rectangle is"<<r2.area();
return 0;
}
```

Expected Output:

```
Area of rectangle is30
Area of rectangle is18
Area of rectangle is18
```

1.C) Inheritance: The capability of a class to derive properties and characteristics from anotherclass is called Inheritance. Inheritance is one of the most important feature ofObject Oriented Programming.
Sub Class: The class that inherits properties from another class is called Sub classor Derived Class.
Super Class:The class whose properties are inherited by sub class is called BaseClass or Super class.
Modes of Inheritance:

- Public mode: If we derive a sub class from a public base class. Thenthe public member of the base class will become public in thederived class and protected members of the base class will becomeprotected in derived class.
- Protected mode: If we derive a sub class from a Protected baseclass. Then both public member and protected members of thebase class will become protected in derived class.
- Private mode: If we derive a sub class from a Private base class.Then both public member and protected members of the base classwill become Private in derived class.

Note : The private members in the base class cannot be directly accessed in thederived class, while protected members can be directly accessed.
Syntax:

```
class subclass_name :access_modebase_class_name
{
//body of subclass
};
```

- subclass_name is the name of the sub class,
- access_mode is the mode in which we want to inherit this sub class for example:public, private etc.
- base_class_name is the name of the base class from which we want to inheritthe sub class.

Types of Inheritance

C++ supports five types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance
- Hybrid Inheritance

Single inheritance is defined as the inheritance in which a derived class isinherited from the only one base class.

Syntax:

```
class base_class
{
//…
};
class base_class :access_modederived_class
{

//…
};
```

Multilevel Inheritance

If a class is derived from another derived class then it is called multilevel inheritance.

Syntax:

```
class A
{
//…
};
class B: access_specifier  class A
{
//…
};
lass C: access_specifier class B
{
//…
};
```

Multiple Inheritance If a class is derived from two or more base classes then it is called multiple inheritance.

Syntax

```
class A
{
//…
};
```

```
class B
{
//…
};
class C :access_specifier A, access_specifier B
{
//…
};
```

Hierarchical Inheritance When several classes are derived from common base class it is called hierarchical inheritance.

Syntax:

```
class A
{
//….
};
class B :access_specifier A
{
//…
};
class C :access_specifier A
{
//…
};
class D :access_specifier A
{
//…
};
```

Hybrid Inheritance The inheritance in which the derivation of a class involves more than one form of any inheritance is called hybrid inheritance. Basically C++ hybrid inheritance is combination of two or more types of inheritance.

Syntax:

```
class A
{
//….
};
class B :access_specifier A
{
//…
};
class C
{
//…
};
class D :access_specifierB, access_specifier C
{
//…
};
```

Aim:Write a Program in c++to create a base class shape and a derived class rectangle to calculate the area of rectangle.

```
#include <iostream>

using namespace std;

// Base class
class Shape {
public:
void setWidth(int w) {
width = w;
}
void setHeight(int h) {
height = h;
}

protected:
int width;
int height;
};

// Derived class
class Rectangle: public Shape {
public:
int getArea() {
return (width * height);
}
};

int main(void) {
Rectangle Rect;

Rect.setWidth(5);
Rect.setHeight(7);
// Print the area of the object.
cout<< "Total area: " <<Rect.getArea() << endl;
return 0;
}
```

Expected Output:

```
Total area: 35
```

1.D)Polymorphism: Polymorphism is an important concept of object-oriented programming. It simply means more than one form. That is, the same entity (function or operator) behaves differently in different scenarios.
For example, The + operator in C++ is used to perform two specific functions. When it is used with numbers (integers and floating-point numbers), it performs addition.

```
int a = 5;
int b = 6;
int sum = a + b;   // sum = 11
```

And when we use the + operator with strings, it performs string concatenation.

For example,

```
stringfirstName = "hello ";
stringlastName = "world";

// name = "hello world"
string name = firstName + lastName;
```

We can implement polymorphism in C++ using the following ways:

- Function overloading
- Operator overloading
- Function overriding
- Virtual functions

1.E) C++ Function Overloading

In C++, we can use two functions having the same name if they have different parameters (either types or number of arguments).
And, depending upon the number/type of arguments, different functions are called. It's a compile-time polymorphism because the compiler knows which function to execute before the program is compiled.

Aim:Program to implement concept of function overloading

```
#include<iostream>
usingnamespacestd;
intsum(int num1, int num2){
return num1 + num2;
}
doublesum(double num1, double num2){
return num1 + num2;
}
intsum(int num1, int num2, int num3){
return num1 + num2 + num3;
}
intmain(){
// Call function with 2 int parameters
cout<<"Sum 1 = "<<sum(5, 6) <<endl;
// Call function with 2 double parameters
cout<<"Sum 2 = "<<sum(5.5, 6.6) <<endl;
// Call function with 3 int parameters
cout<<"Sum 3 = "<<sum(5, 6, 7) <<endl;
return0;
}
```

Expected Output:

```
Sum 1 = 11
Sum 2 = 12.1
Sum 3 = 18
```

1.F) C++ Operator Overloading

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as operator overloading.
For example,
Suppose we have created three objects *c1*, *c2* and *result* from a class named Complex that represents complex numbers.
Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers of *c1* and *c2* by writing the following code:

Result = c1+c2;

Instead of:

Result = c1.addnumbers(c2);

Note: We cannot use operator overloading for fundamental data types like int, float, char and so on.
Syntax for C++ Operator Overloading

To overload an operator, we use a special operator function.

```
class className {
... .. ...
public
returnType operator symbol (arguments) {
... .. ...
}
... .. ...
};
```

Where
- returnType is the return type of the function.
- operator is a keyword.
- symbol is the operator we want to overload. Like: +, <, -, ++, etc.
- arguments is the arguments passed to the function.

Operator Overloading in Unary Operators

Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

Aim:Program to Overload ++ when used as prefix and postfix

```
#include<iostream>
usingnamespacestd;

classCount {
private:
int value;
public:
// Constructor to initialize count to 5
Count() : value(5) {}
// Overload ++ when used as prefix
voidoperator ++ () {
++value;
}
// Overload ++ when used as postfix
voidoperator ++ (int) {    // Here (int) inside the parenthesis is used as
++value;                   //dummy to differentiate between postfix and prefix
}
voiddisplay(){
cout<<"Count: "<< value <<endl;
}
};
intmain(){
Count count1;
// Call the "void operator ++ (int)" function
count1++;
count1.display();
// Call the "void operator ++ ()" function
++ count1;
count1.display();
return0;
}
```

Expected Output:

```
Count: 6
Count: 7
```

Aim:Program to implement binary operator overloading to add two complex numbers

```
#include<iostream>
usingnamespacestd;

classComplex {
private:
float real;
floatimag;
public:
// Constructor to initialize real and imag to 0
Complex() : real(0), imag(0) {}
voidinput(){
cout<<"Enter real and imaginary parts respectively: ";
cin>> real;
cin>>imag;
}
// Overload the + operator
Complex operator + (const Complex&obj) {
Complex temp;
temp.real = real + obj.real;
temp.imag = imag + obj.imag;
return temp;
}
voidoutput(){
if (imag<0)
cout<<"Output Complex number: "<< real <<imag<<"i";
else
cout<<"Output Complex number: "<< real <<"+"<<imag<<"i";
}
};
intmain(){
Complex complex1, complex2, result;
cout<<"Enter first complex number:\n";
complex1.input();
cout<<"Enter second complex number:\n";
complex2.input();
result = complex1 + complex2; // complex1 calls the operator function
// complex2 is passed as an argument to the function

result.output();
return0;
}
```

Expected Output:

```
Enter first complex number:
Enter real and imaginary parts respectively: 9 5
Enter second complex number:
Enter real and imaginary parts respectively: 7 6
Output Complex number: 16+11i
```

Things to Remember in C++ Operator Overloading

1. Two operators = and & are already overloaded by default in C++. For example, to copy objects of the same class, we can directly use the = operator. We do not need to create an operator function.
2. Operator overloading cannot change the precedence and associativity of operators. However, if we want to change the order of evaluation, parentheses should be used.
3. There are 4 operators that cannot be overloaded in C++. They are:
1. :: (scope resolution)
2. . (member selection)
3. .* (member selection through pointer to function)
4. ?: (ternary operator)

1.G)Templates: Templates are powerful features of C++ which allows us to write generic programs. In simple terms, we can create a single function or a class to work with different data types using templates.Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.
The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

- A function template works in a similar to a normal function, with one key difference.

- A single function template can work with different data types at once but, a single normal function can only work with one set of data types.
- Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.
- However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

How to declare a function template?

A function template starts with the keyword **template** followed by template parameter/s inside **<>** which is followed by function declaration.

```
template<class T>
T someFunction(T arg)
{
... .. ...
}
```

Aim:Program to display largest among two numbers using function templates.

```
#include<iostream>
usingnamespacestd;

// template function
template<classT>
TLarge(Tn1, Tn2)
{
if (n1 > n2)
return(n1);

else
return(n2);
}

intmain()
{
int i1, i2;
float f1, f2;
char c1, c2;

cout<<"Enter two integers:\n";
cin>> i1 >> i2;
cout<<Large(i1, i2) <<" is larger."<<endl;
cout<<"\nEnter two floating-point numbers:\n";
cin>> f1 >> f2;
cout<<Large(f1, f2) <<" is larger."<<endl;

cout<<"\nEnter two characters:\n";
cin>> c1 >> c2;
cout<<Large(c1, c2) <<" has larger ASCII value.";

return0;
}
```

Expected Output:

```
Enter two integers:
5
10
10 is larger.

Enter two floating-point numbers:
12.4
10.2
12.4 is larger.

Enter two characters:
z
Z
z has larger ASCII value.
```

Class Templates

- Like function templates, we can also create class templates for generic class operations.Sometimes, we need a class implementation that is same for all classes, only the data types used are different.
- Normally, we would need to create a different class for each data type OR create different member variables and functions within a single class.
- This will unnecessarily bloat our code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.
- However, class templates make it easy to reuse the same code for all data types.

How to declare a class template?

```
template<classT>
class className
{
... .. ...
public:
T var;
T someOperation(T arg);
... .. ...
};
```

In the above declaration, T is the template argument which is a placeholder for the data type used. Inside the class body, a member variable *var* and a member
function someOperation() are both of type T.

How to create a class template object?

To create a class template object, we need to define the data type inside a <> when creation.

```
className<dataType>classObject;
```

For example:

```
className<int>classObject;
className<float>classObject;
className<string>classObject;
```

Aim:Program to add, subtract, multiply and divide two numbers using class template

```
#include<iostream>
Usingnamespacestd;
```

```
template<classT>
classCalculator
{
private:
T num1, num2;

public:
Calculator(T n1, T n2)
{
num1 = n1;
num2 = n2;
}

voiddisplayResult()
{
cout<<"Numbers are: "<< num1 <<" and "<< num2 <<"."<<endl;
cout<<"Addition is: "<<add() <<endl;
cout<<"Subtraction is: "<<subtract() <<endl;
cout<<"Product is: "<<multiply() <<endl;
cout<<"Division is: "<<divide() <<endl;
}
T add(){ return num1 + num2; }
T subtract(){ return num1 - num2; }
T multiply(){ return num1 * num2; }
T divide(){ return num1 / num2; }
};

intmain()
{
Calculator<int>intCalc(2, 1);
Calculator<float>floatCalc(2.4, 1.2);
cout<<"Int results:"<<endl;
intCalc.displayResult();
cout<<endl<<"Float results:"<<endl;
floatCalc.displayResult();
return0;
}
```

Expected Output:

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

1.H)Dynamic memory Allocation

Explicitly done by the programmer.

• Programmers explicitly requests the system to allocate the memory and return starting address of the memory allocated. This address is used by the programmer to access the allocated memory.

• When usage of memory is done , it has to be freed explicitly.

Explicitly Allocating memory in C++:
The 'new' operator

• Used to allocate memory dynamically.

• Can be used to allocate single variable / object or an array of variables / objects.

• The new operator returns pointer to the type allocated.

Examples:-
char *ptr = new char;
int *myint = new int[20];
rectangle *r = new rectangle(4,5);
Before the assignment, the pointer may or may not point to a legitimate memory.
After the assignment, the pointer points to a legitimate memory.

Explicitly freeing memory in C++:
The 'delete' operator

- Is used to free the memory allocated with the new operator.
- The delete operator must be called on a pointer to dynamically allocated memory when it is no longer needed.

1. Can delete a single variable /object or an array
2. delete pointer_name;
3. delete [] Arrayname;

After delete is called on a memory region, that region should no longer be accessed by the program.

Aim :Write a program to create an array dynamically , read the elements in an array and print an array.

```
include <iostream>
using namespace std;
int main ()
{
int i,n;
int * p;
cout<< "How many numbers would you like to type? ";
cin>>i;
p= new (nothrow) int[i];
if (p == 0)
cout<< "Error: memory could not be allocated";
else
{
for (n=0; n<i; n++)
{
cout<< "Enter number: ";
cin>> p[n];
}
cout<< "You have entered: ";
for (n=0; n<i; n++)
cout<< p[n] << ", ";
delete[] p;
}
return 0;
}
```

Expected Output:

```
How many numbers would you like to type? 5
Enter number: 1
Enter number: 2
Enter number: 3
Enter number: 4
Enter number: 5
You have entered: 1, 2, 3, 4, 5,
```

1.I)Exception Handling

- Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing.
- Exception handling mechanism provides a means to detect and report an exception circumstances.
- Find the problem (Hit the exception)
- Inform that an error has occurred(Throw the exception)
- Receive the error information(Catch the exception)
- Take corrective actions(Handle the exception)

The exception handling mechanism is built upon three keywords:

1. Try

Is used to preface a block of statements which may generate exceptions.

2. Throw

When an exception is detected, it is thrown using a throw statement in the try block

3. Catch

A catch block defined by the keyword catch catches the exception thrown by the throw statement in the try block and handles it appropriately.

Aim:Write a Program to raise divide by zero exception

```
#include<iostream>
using namespace std;
int main()
{
int a=10,b=0;
try
{
if (b==0)
{
throw "division by zero not possible";
}
}
catch (const char *ex)
{
cout<<ex;
}
return 0;
}
```

Expected Output:

```
division by zero not possible
```

Program 2:Implementation of Stacks, Queues (using both arrays and linked lists).

Aim : **C++ Programs to implement Implementation of Stacks, Queues (using both arrays and linked lists).**

Description :

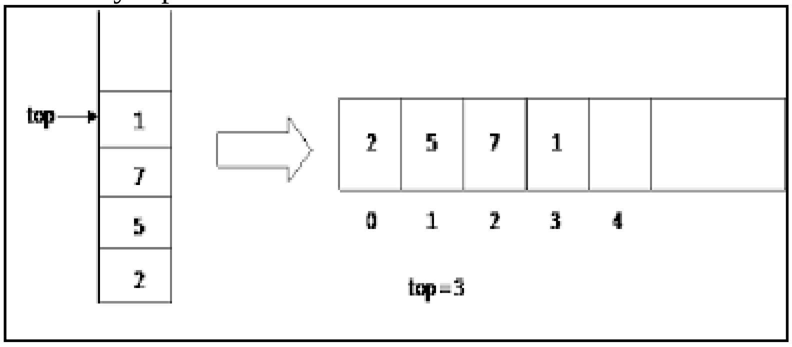
2.A) Stack ADT

Stacks: A stack is a linear data structure in which elements are added to or deleted from a single end called as Top of the stack. The elements last inserted is the first to be removed, therefore stack is said to follow the Last In First Out principle, LIFO. In this context, the insert operation is more commonly known as Push and deletes operation as Pop. Other operations on a stack are: to find size of stack, return top most element, search for an element in the stack etc.
Stacks are used in: Function calls, Recursion, Evaluation of expressions by compilers,undo mechanism in an editor, etc

The stack data structures can be represented using Arrays or Linked Lists. When represented using an array, the initial value of Top, the index for top most element, when stack is empty, is -1, a nonexistent index.
The ADT for an array based stack is :

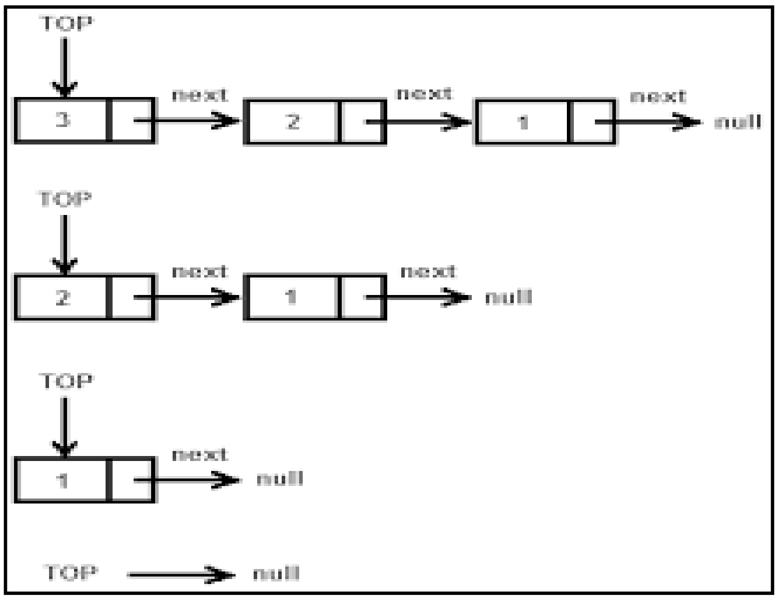
```
ADT Stack
{
Private:
An array;
Top index;
Public:
Stack();
boolean isEmpty()
boolean isFull();
void push( item e);
int pop();
int peek();
}
```

When elements are pushed into stack, the Top gradually increases towards right. Each pop operation causes the Top index to decrease by one.
Figure illustrates the array representation.



Stacks can also be represented using linked representation, wherein the elements of the stack are not stored contiguously using an array, rather the elements are stored using a node structure that can be stored in the memory non-contiguously. Each node contains information about the data element and the address of where the next element is stored in the memory.

Figure for linked stack



The ADT for a
linked stack is: ADT

```
Struct Node
{
Private:
T data; //template type data
Node<T> *link;
};
ADT LinkedStack
{
Private:
Node<T> *Top;
Int size;
Public:

Push(T item);
Pop();
boolisEmpty();
}
```

Aim:A program that implements a stack operations using array.

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

using namespace std;

#define MAX 1000

class Stack {
int top;
int a[MAX]; // Maximum size of Stack
public:
Stack() { top = -1; }
void push(int x);
int pop();
int peek();
bool isEmpty();
bool isFull();
void display();
};

void Stack::push(int x)
{
if (isFull()) {
cout << "Stack Overflow";
}
else {
a[++top] = x;
cout << x << " pushed into stack\n";
}
}
int Stack::pop()
{
if (isEmpty()) {
cout << "Stack Underflow";
return 0;

}

else {

int x = a[top--];
return x;
}
}
int Stack::peek()
{
if (isEmpty()) {
cout << "Stack is Empty";
return 0;
}
else {
int x = a[top];
return x;
}
}
bool Stack::isEmpty()
{
return (top < 0);
}
bool Stack::isFull()
{
return(top>=MAX-1);
}
void Stack::display()
{
cout<<"Stack elements are\n";
for(int i=top;i>=0;i--)
cout<<a[i]<<" ";
}
// Driver program to test above functions
int main()
{
class Stack s;
s.push(10);
s.push(20);
s.push(30);
cout << s.pop() << " Popped from stack\n";
s.display();
return 0;
}
```

Expected Output:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack

30 Popped from stack
Stack elements are
20 10
```

Aim :Program to implement Linked Stack

```
#include<iostream>
using namespace std;
struct Node
{
int data;
Node *next;
}*top=NULL,*p;

Node* newnode(int x)
{
p=new Node;
p->data=x;
p->next=NULL;
return(p);
}

void push(Node *q)
{
if(top==NULL)
top=q;
else
{
q->next=top;
top=q;
}
}
void pop(){
if(top==NULL){
cout<<"Stack is empty!!";
}
else{
cout<<"Deleted element is "<<top->data;
p=top;
top=top->next;
delete(p);
}
}
void showstack()
{
Node *q;
q=top;

if(top==NULL){
cout<<"Stack is empty!!";
}
else{
while(q!=NULL)
{
cout<<q->data<<" ";
q=q->next;
}
}
}
int main()
{
int ch,x;
Node *nptr;

while(1)
{
cout<<"\n\n1.Push\n2.Pop\n3.Display\n4.Exit";
cout<<"\nEnter your choice(1-4):";
cin>>ch;

switch(ch){
case 1: cout<<"\nEnter data:";
cin>>x;
nptr=newnode(x);
push(nptr);
break;

case 2: pop();
break;

case 3: showstack();
break;

case 4: exit(0);

default: cout<<"\nWrong choice!!";
}
}

return 0;
}
```

Expected Output:

```
1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):1
Enter data:10
1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):1
Enter data:20
1.Push
2.Pop
3.Display
```

```
4.Exit
Enter your choice(1-4):1
Enter data:30
1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):2
Deleted element is 30
1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):3
20 10
1.Push
2.Pop
3.Display
4.Exit
Enter your choice(1-4):4
```

2.B) Queue ADT

A queue is a linear data structure in which elements can be inserted and deleted from two different ends. The end at which elements are inserted into queue is referred to as Rear and the end from which elements are deleted is known as Front. The element first inserted will be the first to delete, therefore queue is said to follow, FirstIn First Out, FIFO principle. Queues are used: in processor scheduling, request processing systems, etc. A queue can be implemented using arrays or linked representation.

The ADT for array based Queue is

```
ADT Queue
{
public:
Queue();
void enqueue(const T& newItem);
int dequeue();
void display();
private:
int front;
int rear;
int q[50];
}
```

Aim:Program to implement Queue operations using arrays

```
#include<iostream>
using namespace std;
#define SIZE 10
class Queue
```

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

public:
Queue()
{
rear = front = -1;
}
//declaring enqueue, dequeue and display functions
void enqueue(int x);
int dequeue();
void display();
};

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

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

// function to display the queue elements
void Queue :: display()
{
int i;
for( i = front; i <= rear; i++)
{
cout << a[i] << endl;
}
}
// the main function
int main()
{
Queue q;
q.enqueue(10);
q.enqueue(100);
q.enqueue(1000);
q.enqueue(1001);
q.enqueue(1002);
cout<<"Queue elements Are \n";
q.display();
q.dequeue();
q.enqueue(1003);
q.dequeue();
q.dequeue();
q.enqueue(1004);
cout<<"Queue elements are \n";
q.display();
return 0;
}
```

Expected Output:

```
Queue elements Are
10
100
1000
1001
1002
Queue elements are
1001
1002
1003
1004
```

The ADT for linked Queue is:

```
struct Node{
int data;
Node *next;
};
class Queue{
Node *front,*rear;
Public:
Queue(){front=rear=NULL;}

void insert(int n);
void deleteitem();
void display();
~Queue();
};
```

2.C) Aim: C++ program to Implement a QUEUE using singly linked list

```
#include<iostream>
using namespace std;
struct Node{
int data;
Node *next;
};
class Queue{
public:
Node *front,*rear;
Queue(){front=rear=NULL;}
void insert(int n);
void deleteitem();
void display();
~Queue();
};
void Queue::insert(int n){
Node *temp=new Node;
if(temp==NULL){
cout<<"Overflow"<<endl;
return;
}
temp->data=n;
temp->next=NULL;
//for first node
if(front==NULL){
front=rear=temp;
}
else{
rear->next=temp;
rear=temp;
}
cout<<n<<" has been inserted successfully."<<endl;
}
void Queue::display(){
if(front==NULL){
cout<<"Underflow."<<endl;
return;
}
Node *temp=front;
//will check until NULL is not found
while(temp){
cout<<temp->data<<" ";
temp=temp->next;
}
cout<<endl;
}
void Queue :: deleteitem()
{
if (front==NULL){
cout<<"underflow"<<endl;
return;
}
cout<<front->data<<" is being deleted "<<endl;
if(front==rear)//if only one node is there
front=rear=NULL;
else
front=front->next;
}
Queue ::~Queue()
{
while(front!=NULL)
{
Node *temp=front;
front=front->next;
delete temp;
}
rear=NULL;
}
int main(){
Queue Q;
Q.display();
Q.insert(10);
Q.insert(24);
Q.insert(28);
Q.insert(32);
Q.insert(30);
Q.display();
Q.deleteitem();
Q.deleteitem();
Q.deleteitem();
Q.deleteitem();
Q.deleteitem();
return 0;
}
```

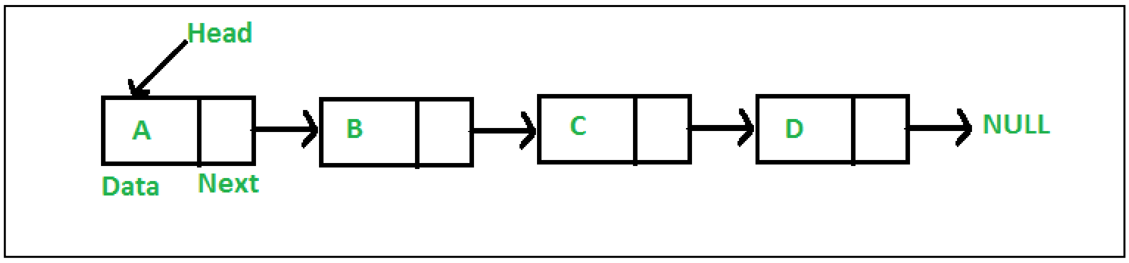
Expected Output:

```
Underflow.
10 has been inserted successfully.
24 has been inserted successfully.
28 has been inserted successfully.
32 has been inserted successfully.
30 has been inserted successfully.
10 24 28 32 30
10 is being deleted
24 is being deleted
28 is being deleted
32 is being deleted
30 is being deleted
```

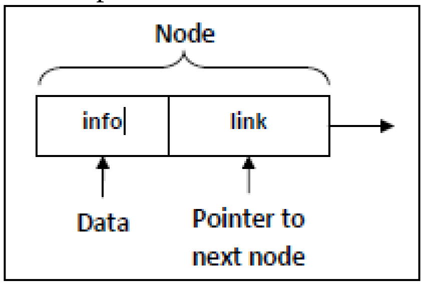
Program 3:Implementation of Singly Linked List, Double Linked List, Circular List.
Aim : **C++ Programs to Implementation of Singly Linked List,Double Linked List, Circular List.**

Description :

3.A)Linked Lists:
A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown below:



The elements of a linked list are called the nodes. A node has two fields i.e. data and next. The data field contains the data being stored in that specific node. It cannot just be a single variable. There may be many variables presenting the data section of a node. The next field contains the address of the next node. So this is the place where the link between nodes is established.



Operations on Linked Lists:

Insert: Insert at first position, insert at last position, insert into ordered list

Delete: Delete an element from first, last or any intermediate position

Traverse List: print the list

Copy the linked List, Reverse the linked list, search for an element in the list, etc

Types of Linked Lists:

Singly Linked List:

- It is a basic type of linkedlist.
- Each node contains data and pointer to next node and last node’s pointer isNULL.
- Limitation of SLL is that we can traverse the list in only one direction, forwarddirection.

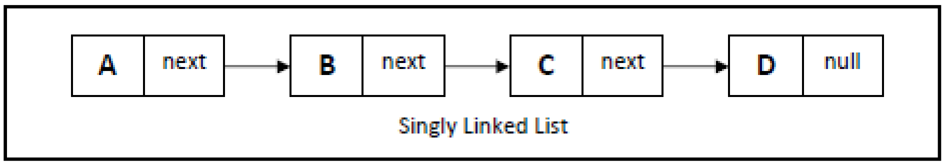


Figure : Single Linked List

Circular Linked List:

- CLL is a SLL where last node points to first node in thelist
- It does not contain null pointers likeSLL
- We can traverse the list in only onedirection
- Its advantage is that when we want to go from last node to first node, it directly

points to first node

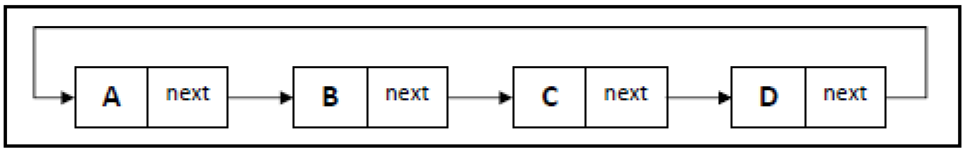
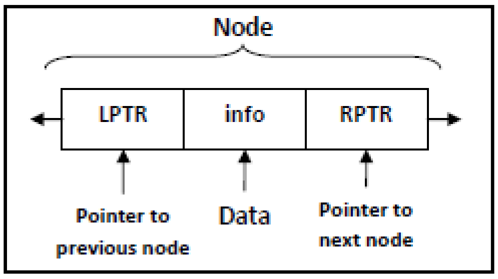


Figure: CLL

Doubly Linked List:

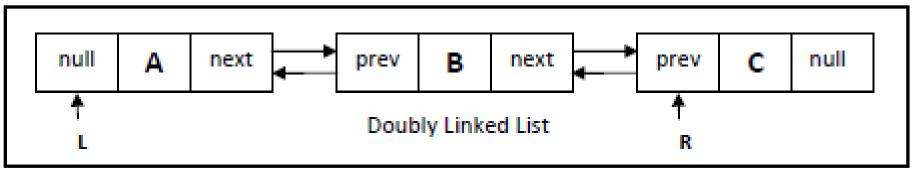
- Each node of doubly linked list contains data and two pointer fields, pointer to

previous and next node.

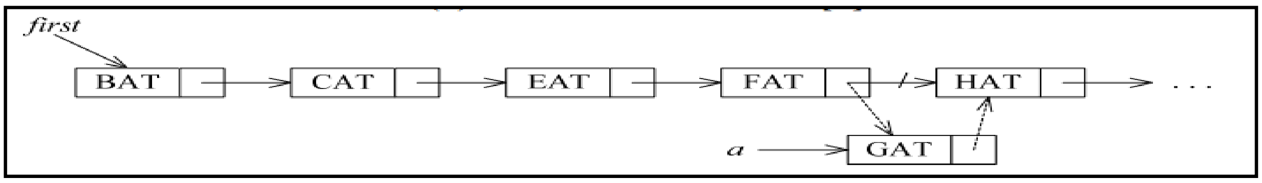


- Advantage of DLL is that we can traverse the list any direction, forward orreverse.
- Other advantages of DLL are that we can delete a node with little trouble, since we

have pointers to the previous and next nodes. A node on a SLL cannot be removed unless we have pointer to itspredecessor.



Inserting into a SLL:



3.A)Aim :Program to Create a singly linked list , Insert an element , delete an element,searchan element and display linked list

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;
class Node
{
public:
int info;
Node* next;
};
class List:public Node
{

Node *first,*last;
public:
List()
{
first=NULL;
last=NULL;
}
void create();
void insert();
void delet();
void display();
void search();
};
void List::create()
{
Node *temp;
temp=new Node;
int n;
cout<<"\nEnter an Element:";
cin>>n;
temp->info=n;
temp->next=NULL;
if(first==NULL)
{
first=temp;
last=first;
}
```

```
else
{
last->next=temp;
last=temp;
}
}
void List::insert()
{
Node *prev,*cur;
prev=NULL;
cur=first;
int count=1,pos,ch,n;
Node *temp=new Node;
cout<<"\nEnter an Element:";
cin>>n;
temp->info=n;
temp->next=NULL;
cout<<"\nINSERT AS\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES";
cout<<"\nEnter Your Choice:";
cin>>ch;
switch(ch)
{
case 1:
temp->next=first;
first=temp;
break;
case 2:
last->next=temp;
last=temp;
break;
case 3:
cout<<"\nEnter the Position to Insert:";
cin>>pos;
while(count!=pos)
{
prev=cur;
cur=cur->next;
count++;
}
if(count==pos)
{
prev->next=temp;
temp->next=cur;
}
else
cout<<"\nNot Able to Insert";
break;
}
}
void List::delet()
{
```

```
Node *prev=NULL,*cur=first;
int count=1,pos,ch;
cout<<"\nDELETE\n1:FIRSTNODE\n2:LASTNODE\n3:IN BETWEEN FIRST&LAST
NODES";
cout<<"\nEnter Your Choice:";
cin>>ch;
switch(ch)
{
case 1:
if(first!=NULL)
{
cout<<"\nDeleted Element is "<<first->info;
first=first->next;
}
else
cout<<"\nNot Able to Delete";
break;
case 2:
while(cur!=last)
{
prev=cur;
cur=cur->next;
}
if(cur==last)
{
cout<<"\nDeleted Element is: "<<cur->info;
prev->next=NULL;
last=prev;
}
else
cout<<"\nNot Able to Delete";
break;
case 3:
cout<<"\nEnter the Position of Deletion:";
cin>>pos;
while(count!=pos)
{
prev=cur;
cur=cur->next;
count++;
}
if(count==pos)
{
cout<<"\nDeleted Element is: "<<cur->info;
prev->next=cur->next;
}
else
cout<<"\nNot Able to Delete";
break;
}
}
void List::display()
```

```
{
Node *temp=first;
if(temp==NULL)
{
cout<<"\nList is Empty";
}
while(temp!=NULL)
{
cout<<temp->info;
cout<<"-->";
temp=temp->next;
}
cout<<"NULL";
}
void List::search()
{
int value,pos=0;
bool flag=false;
if(first==NULL)
{
cout<<"List is Empty";
return;
}
cout<<"Enter the Value to be Searched:";
cin>>value;
Node *temp;
temp=first;
while(temp!=NULL)
{
pos++;
if(temp->info==value)
{
flag=true;
cout<<"Element"<<value<<"is Found at "<<pos<<" Position";
return;
}
temp=temp->next;
}
if(!flag)
{
cout<<"Element "<<value<<" not Found in the List";
}
}
int main()
{
List l;
int ch;
while(1)
{
cout<<"\n**** MENU ****";
cout<<"\n1:CREATE\n2:INSERT\n3:DELETE\n4:SEARCH\n5:DISPLAY\n6:EXIT\n";
cout<<"\nEnter Your Choice:";
cin>>ch;
switch(ch)
{
case 1:
l.create();
break;
case 2:
l.insert();
break;
case 3:
l.delet();
break;
case 4:
l.search();
break;
case 5:
l.display();
break;
case 6:
return 0;
}
}
return 0;
}
```

EXPECTED OUTPUT:

```
**** MENU ****
1:CREATE
2:INSERT
3:DELETE
4:SEARCH
5:DISPLAY
6:EXIT

Enter Your Choice:1

Enter an Element:10

**** MENU ****
1:CREATE
2:INSERT
3:DELETE
4:SEARCH
5:DISPLAY
6:EXIT

Enter Your Choice:2

Enter an Element:20

INSERT AS
1:FIRSTNODE
```

```
2:LASTNODE
3:IN BETWEEN FIRST&LAST NODES
Enter Your Choice:2

**** MENU ****
1:CREATE
2:INSERT
3:DELETE
4:SEARCH
5:DISPLAY
6:EXIT

Enter Your Choice:2

Enter an Element:30

INSERT AS
1:FIRSTNODE
2:LASTNODE
3:IN BETWEEN FIRST&LAST NODES
Enter Your Choice:1

**** MENU ****
1:CREATE
2:INSERT
3:DELETE
4:SEARCH
5:DISPLAY
6:EXIT

Enter Your Choice:5
30-->10-->20-->NULL
**** MENU ****
1:CREATE
2:INSERT
3:DELETE
4:SEARCH
5:DISPLAY
6:EXIT

Enter Your Choice:4
Enter the Value to be Searched:20
Element20is Found at 3 Position
**** MENU ****
1:CREATE
2:INSERT
3:DELETE
4:SEARCH
5:DISPLAY
6:EXIT

Enter Your Choice:4
Enter the Value to be Searched:50
Element 50 not Found in the List
**** MENU ****
1:CREATE
2:INSERT
3:DELETE
4:SEARCH
5:DISPLAY
6:EXIT

Enter Your Choice:3

DELETE
1:FIRSTNODE
2:LASTNODE
3:IN BETWEEN FIRST&LAST NODES
Enter Your Choice:3

Enter the Position of Deletion:2

Deleted Element is: 10
**** MENU ****
1:CREATE
2:INSERT
3:DELETE
4:SEARCH
5:DISPLAY
6:EXIT

Enter Your Choice:5
30-->20-->NULL
**** MENU ****
1:CREATE
2:INSERT
3:DELETE
4:SEARCH
5:DISPLAY
6:EXIT

Enter Your Choice:6
```

3.B) AIM :C++ Program to Implement Doubly Linked List

```
#include<iostream>
#include<cstdio>
#include<cstdlib>

* Node Declaration

using namespace std;
struct node
{
```

```
int info;
struct node *next;
struct node *prev;
}*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)
{
struct node *s, *temp;
temp = new(struct 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;
}
struct node *temp;
temp = new(struct 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;
}
struct 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(struct 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)
{
struct 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()
{
struct 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()
{
struct 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()
{
struct 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;
}
```

EXPECTED OUTPUT:

```
$ g++ doubly_llist.cpp
$ a.out

Operations on Doubly linked list
1.Create Node
2.Add at begining
3.Add after
4.Delete
5.Display
6.Count
7.Reverse
8.Quit
Enter your choice : 2
Enter the element: 100
First Create the list.

Operations on Doubly linked list
Enter your choice : 3
Enter the element: 200
Insert Element after postion: 1
First Create the list.
```

```
Operations on Doubly linked list
Enter your choice : 4
List empty,nothing to delete

Operations on Doubly linked list
Enter your choice : 5
List empty,nothing to display

Operations on Doubly linked list

Enter your choice : 6
Number of elements are: 0

Operations on Doubly linked list
Enter your choice : 7
List empty,nothing to reverse

Operations on Doubly linked list

Enter your choice : 1
Enter the element: 100

Operations on Doubly linked list
Enter your choice : 5
The Doubly Link List is :
100 <->NULL

Operations on Doubly linked list
Enter your choice : 2
Enter the element: 200
Element Inserted

Operations on Doubly linked list

Enter your choice : 5
The Doubly Link List is :
200 <-> 100 <->NULL

Operations on Doubly linked list
Enter your choice : 3
Enter the element: 50
Insert Element after postion: 2
Element Inserted

Operations on Doubly linked list
Enter your choice : 5
The Doubly Link List is :
200 <-> 100 <-> 50 <->NULL

Operations on Doubly linked list
Enter your choice : 3
Enter the element: 150
```

```
Insert Element after postion: 3
Element Inserted

Operations on Doubly linked list
Enter your choice : 5
The Doubly Link List is :
200 <-> 100 <-> 50 <-> 150 <->NULL

Operations on Doubly linked list
Enter your choice : 6
Number of elements are: 4

Operations on Doubly linked list
Enter your choice : 4
Enter the element for deletion: 50
Element Deleted

Operations on Doubly linked list
Enter your choice : 5
The Doubly Link List is :
200 <-> 100 <-> 150 <->NULL

Operations on Doubly linked list
Enter your choice : 6
Number of elements are: 3

Operations on Doubly linked list
Enter your choice : 7
List Reversed

Operations on Doubly linked list
Enter your choice : 5
The Doubly Link List is :
150 <-> 100 <-> 200 <->NULL

Operations on Doubly linked list
Enter your choice : 3
Enter the element: 200
Insert Element after postion: 100
There are less than 100 elements.

Operations on Doubly linked list
Enter your choice : 4
Enter the element for deletion: 150
Element Deleted

Operations on Doubly linked list
Enter your choice : 5
The Doubly Link List is :
100 <-> 200 <->NULL

Operations on Doubly linked list
```

```
Enter your choice : 8
```

3.C)Aim:C++ Program to Implement Circular Linked List

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
using namespace std;
/*
* Node Declaration
*/
struct node
{
int info;
struct node *next;
}*last;

/*
* Class Declaration
*/
class circular_llist
{
public:
void create_node(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_list();
void update();
void sort();
circular_llist()
{
last = NULL;
}
};

/*
* Main :contains menu
*/
int main()
{
int choice, element, position;
circular_llist cl;
while (1)
{
cout<<endl<<"---------------------------"<<endl;
cout<<endl<<"Circular singly linked list"<<endl;
cout<<endl<<"---------------------------"<<endl;
cout<<"1.Create Node"<<endl;
cout<<"2.Add at beginning"<<endl;
cout<<"3.Add after"<<endl;
```

```
cout<<"4.Delete"<<endl;
cout<<"5.Search"<<endl;
cout<<"6.Display"<<endl;
cout<<"7.Update"<<endl;
cout<<"8.Sort"<<endl;
cout<<"9.Quit"<<endl;
cout<<"Enter your choice : ";
cin>>choice;
switch(choice)
{
case 1:
cout<<"Enter the element: ";
cin>>element;
cl.create_node(element);
cout<<endl;
break;
case 2:
cout<<"Enter the element: ";
cin>>element;
cl.add_begin(element);
cout<<endl;
break;
case 3:
cout<<"Enter the element: ";
cin>>element;
cout<<"Insert element after position: ";
cin>>position;
cl.add_after(element, position);
cout<<endl;
break;
case 4:
if (last == NULL)
{
cout<<"List is empty, nothing to delete"<<endl;
break;
}
cout<<"Enter the element for deletion: ";
cin>>element;
cl.delete_element(element);
cout<<endl;
break;
case 5:
if (last == NULL)
{
cout<<"List Empty!! Can't search"<<endl;
break;
}
cout<<"Enter the element to be searched: ";
cin>>element;
cl.search_element(element);
cout<<endl;
break;
case 6:
cl.display_list();
break;
case 7:
cl.update();
break;
case 8:
cl.sort();
break;
case 9:
exit(1);
break;
default:
cout<<"Wrong choice"<<endl;
}
}
return 0;
}

/*
* Create Circular Link List
*/
void circular_llist::create_node(int value)
{
struct node *temp;
temp = new(struct node);
temp->info = value;
if (last == NULL)
{
last = temp;
temp->next = last;
}
else
{
temp->next = last->next;
last->next = temp;
last = temp;
}
}

/*
* Insertion of element at beginning
*/
void circular_llist::add_begin(int value)
{
if (last == NULL)
{
cout<<"First Create the list."<<endl;
return;
}
struct node *temp;
temp = new(struct node);
```

```
temp->info = value;
temp->next = last->next;
last->next = temp;
}

/*
* Insertion of element at a particular place
*/
void circular_llist::add_after(int value, int pos)
{
if (last == NULL)
{
cout<<"First Create the list."<<endl;
return;
}
struct node *temp, *s;
s = last->next;
for (int i = 0;i< pos-1;i++)
{
s = s->next;
if (s == last->next)
{
cout<<"There are less than ";
cout<<pos<<" in the list"<<endl;
return;
}
}
temp = new(struct node);
temp->next = s->next;
temp->info = value;
s->next = temp;
/*Element inserted at the end*/
if (s == last)
{
last=temp;
}
}

/*
* Deletion of element from the list
*/
void circular_llist::delete_element(int value)
{
struct node *temp, *s;
s = last->next;
/* If List has only one element*/
if (last->next == last && last->info == value)
{
temp = last;
last = NULL;
free(temp);
return;
```

```
}
if (s->info == value)  /*First Element Deletion*/
{
temp = s;
last->next = s->next;
free(temp);
return;
}
while (s->next != last)
{
/*Deletion of Element in between*/
if (s->next->info == value)
{
temp = s->next;
s->next = temp->next;
free(temp);
cout<<"Element "<<value;
cout<<" deleted from the list"<<endl;
return;
}
s = s->next;
}
/*Deletion of last element*/
if (s->next->info == value)
{
temp = s->next;
s->next = last->next;
free(temp);
last = s;
return;
}
cout<<"Element "<<value<<" not found in the list"<<endl;
}

/*
* Search element in the list
*/
void circular_llist::search_element(int value)
{
struct node *s;
int counter = 0;
s = last->next;
while (s != last)
{
counter++;
if (s->info == value)
{
cout<<"Element "<<value;
cout<<" found at position "<<counter<<endl;
return;
}
s = s->next;
```

```
}
if (s->info == value)
{
counter++;
cout<<"Element "<<value;
cout<<" found at position "<<counter<<endl;
return;
}
cout<<"Element "<<value<<" not found in the list"<<endl;
}

/*
* Display Circular Link List
*/
void circular_llist::display_list()
{
struct node *s;
if (last == NULL)
{
cout<<"List is empty, nothing to display"<<endl;
return;
}
s = last->next;
cout<<"Circular Link List: "<<endl;
while (s != last)
{
cout<<s->info<<"->";
s = s->next;
}
cout<<s->info<<endl;
}

/*
* Update Circular Link List
*/
void circular_llist::update()
{
int value, pos, i;
if (last == NULL)
{
cout<<"List is empty, nothing to update"<<endl;
return;
}
cout<<"Enter the node position to be updated: ";
cin>>pos;
cout<<"Enter the new value: ";
cin>>value;
struct node *s;
s = last->next;
for (i = 0;i<pos - 1;i++)
{
if (s == last)
```

```
{
cout<<"There are less than "<<pos<<" elements.";
cout<<endl;
return;
}
s = s->next;
}
s->info = value;
cout<<"Node Updated"<<endl;
}

/*
* Sort Circular Link List
*/
void circular_llist::sort()
{
struct node *s, *ptr;
int temp;
if (last == NULL)
{
cout<<"List is empty, nothing to sort"<<endl;
return;
}
s = last->next;
while (s != last)
{
ptr = s->next;
while (ptr != last->next)
{
if (ptr != last->next)
{
if (s->info >ptr->info)
{
temp = s->info;
s->info = ptr->info;
ptr->info = temp;
}
}
else
{
break;
}
ptr = ptr->next;
}
s = s->next;
}
}
```

EXPECTED OUTPUT:

```
Operations on Circular singly linked list
1.Create Node
2.Add at beginning
```

```
3.Add after
4.Delete
5.Search
6.Display
7.Update
8.Sort
9.Quit
Enter your choice : 4
List is empty, nothing to delete

Operations on Circular singly linked list
Enter your choice : 5
List is empty, nothing to search

Operations on Circular singly linked list
Enter your choice : 6
List is empty, nothing to display

Operations on Circular singly linked list
Enter your choice : 7
List is empty, nothing to update

Operations on Circular singly linked list
Enter your choice : 8
List is empty, nothing to sort

Operations on Circular singly linked list
Enter your choice : 1
Enter the element: 100

Operations on Circular singly linked list

Enter your choice : 2
Enter the element: 200

Operations on Circular singly linked list
Enter your choice : 6
Circular Link List:
200->100

Operations on Circular singly linked list
Enter your choice : 3
Enter the element: 50
Insert element after position: 2

Operations on Circular singly linked list
Enter your choice : 6
Circular Link List:
200->100->50

Operations on Circular singly linked list
Enter your choice : 3
Enter the element: 150
Insert element after position: 3

Operations on Circular singly linked list
Enter your choice : 6
Circular Link List:
200->100->50->150

Operations on Circular singly linked list
Enter your choice : 3
Enter the element: 1000
Insert element after position: 50
There are less than 50 in the list

Operations on Circular singly linked list
Enter your choice : 4
Enter the element for deletion: 150
Operations on Circular singly linked list
Enter your choice : 6
Circular Link List:
200->100->50
Operations on Circular singly linked list
Enter your choice : 5
Enter the element to be searched: 100
Element 100 found at position 2

Operations on Circular singly linked list
Enter your choice : 7
Enter the node position to be updated: 1
Enter the new value: 1010
Node Updated

Operations on Circular singly linked list
Enter your choice : 6
Circular Link List:
1010->100->50
Operations on Circular singly linked list
Enter your choice : 8
Operations on Circular singly linked list
Enter your choice : 6
Circular Link List:
50->100->1010
Operations on Circular singly linked list
            Enter your choice : 9
```

Program 4:Implementation of Infix to Postfix conversion and evaluation of postfix expression:

Aim : C++ Programs to Implementation of Infix to Postfix conversion and evaluation of postfix expression

Description :

Conventional notation is called infix notation. The arithmetic operators appears between two operands. Parentheses are required to specify the order of the operations. For example: a + (b * c).

Post fix notation (also, known as reverse Polish notation) eliminates the need for parentheses. There are no precedence rules to learn, and parenthese are never needed. Because of this simplicity, some popular hand-held calculators use postfix notation to avoid the complications of multiple sets of parentheses. The operator is placed directly after the two operands it needs to apply. For example: a b c * +

4.A) Aim :Program to convert an Infix expression to Postfix expression

```
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
#include<ctype.h>
class expression
{
private:
char infix[100]; char stack[200]; int top;
int r;
char postfix[100]; public:
void convert(); int input_p(char); int stack_p(char); int rank(char);
};
int expression::input_p(char c)
{
if(c==’+’ || c==’-') return 1;
else if(c==’*’ || c==’/') return 3;
else if(c==’^') return 6;
else if(isalpha(c)!=0) return 7;
else if(c==’(‘) return 9;
else if(c==’)') return 0;
else
{
cout<<”Invalid expression ::input error\n”; exit(0);
}
}
int expression::stack_p(char c)
{
if(c==’+’ || c==’-') return 2;
else if(c==’*’ || c==’/')

<<”\n**************************************************\n”;
cout<<”Enter an infix expression ::\n”; cin>>infix;
int l=strlen(infix); infix[l]=’)'; infix[l+1]=”;
```

```
//Convertion starts top=1; stack[top]='(';r=0;
int x=-1; int i=0;
char next=infix[i]; while(next!=")
{
//Pop all the elements to outputin stack which have higher precedence while( input_p(next)
<stack_p(stack[top]) )
{
if(top<1)
{
cout<<"invalid expression ::stack error\n"; exit(0);
}
postfix[++x]=stack[top]; top–; r=r+rank(postfix[x]);

if(r<1)
{
cout<<"Invalid expression ::r<1\n"; exit(0);
}
}
if(input_p( next ) != stack_p( stack[top])) stack[++top]=next;
else top–; i++;
next=infix[i];
}
postfix[++x]="; if(r!=1 || top!=0)
{
cout<<"Invalid expression ::error in rank or stack\n"; exit(0);
}
cout<<"\n\nThe corresponding postfix expression is ::\n"; cout<<postfix<<endl;
}
int main()
{
expression obj; obj.convert(); return 0;
}
```

EXPECTED OUTPUT::

```
************************************************ This program converts the given
infix expression
in to postfix form
************************************************ Enter an infix expression ::
(a+b^c^d)*(c+d)
The corresponding postfix expression is :: abcd^^+cd+*
Press any key tocontinue
```

4.B)Aim:Program to evaluatea Postfix expression.

```
#include  <iostream.h>
#include <math.h>
#include <stdlib.h>
#include <ctype.h>
const int MAX = 50;
class postfix
{
private :
```

```
int stack[MAX] ;
int top, nn ;
char *s ; public :
postfix( ) ;
void setexpr( char *str ) ;
void push ( int item ) ;
int pop( ) ;
void calculate( ) ;
void show( ) ;
} ;
postfix :: postfix( )
{
top = -1 ;
}
void postfix :: setexpr ( char *str )
{
s = str ;
}
void postfix :: push ( int item )
{
if ( top == MAX - 1 )
cout<< endl << "Stack is full" ;
else
{
top++ ;
stack[top] = item ;
}
int postfix :: pop( )
{
if ( top == -1 )
{
cout<< endl << "Stack is empty" ; return NULL ;
}
int data = stack[top] ; top-- ;
return data ;
}
void postfix :: calculate( )
{
int n1, n2, n3 ; while ( *s )
{
if ( *s == ' ' || *s == '\t' )
{
s++ ;
continue ;
}
if ( isdigit ( *s ) )
{
}
else
{
nn = *s - '0' ;
push ( nn ) ;
n1 = pop( ) ;
n2 = pop( ) ;
switch ( *s )
{
case '+' :
n3 = n2 + n1 ; break ;
case '-' :
n3 = n2 - n1 ; break ;
case '/' :
n3 = n2 / n1 ; break ;
case '*' :
n3 = n2 * n1 ; break ;
case '%' :
n3 = n2 % n1 ; break ;
case '$' :
n3 = pow ( n2 , n1 ) ;
break ;
default :
cout<<"Unknown operator";
exit(1);
} s++ ;
}
}
}
push ( n3 ) ;
cout<< "Unknown operator" ; exit ( 1 ) ;
void postfix :: show( )
{
nn = pop ( ) ;
cout<< "Result is: " <<nn ;
}
void main( )
{
char expr[MAX] ;
cout<< "\nEnter postfix expression to be evaluated : " ; cin.getline ( expr, MAX ) ;
postfix q ;q.setexpr ( expr ) ; q.calculate( ) ; q.show( ) ;
}
```

Expected Output:

```
Enter postfix expression to be evaluated : 123*4- Result is:2
```

Program 5:Implementation of Polynomial arithmetic using linked list

Aim : **C++ Programs to Implementation of Polynomial arithmetic using linked list**

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

// Creating a NODE Structure struct node
{
int coe,exp;          // data
struct node *next; // link to next node and previous node
};

// Creating a class Polynomial class polynomial
{
struct node *start,*ptrn,*ptrp; public:
void get_poly(); // to get a polynomial void show();        // show
void add(polynomial p1,polynomial p2); // Add two polynomials void subtract(polynomial
p1,polynomial p2); //Subtract2 polynomials
};

void polynomial::get_poly()             // Get Polynomial
{
char c='y'; ptrn=ptrp=start=NULL; while(c=='y' || c=='Y')
{
ptrn=new node; ptrp->next=ptrn; if(start==NULL) start=ptrn;
ptrp=ptrn;
cout<<"\nEnter the coefficient: "; cin>>ptrn->coe;
cout<<"\nEnter the exponent: "; cin>>ptrn->exp;
ptrn->next=NULL;
cout<<"\nEnter y to add more nodes: "; cin>>c;
}
return;
}


void polynomial::show() // Show Polynomial
{
struct node *ptr; ptr=start; while(ptr!=NULL)
{
cout<<ptr->coe<<"X^"<<ptr->exp<<" + "; ptr=ptr->next;
}
cout<<" ";
}

void polynomial::add(polynomial p1,polynomial p2) // Add Polynomials
{
struct node *p1ptr,*p2ptr; int coe,exp; ptrn=ptrp=start=NULL; p1ptr=p1.start; p2ptr=p2.start;
while(p1ptr!=NULL && p2ptr!=NULL)
{
if(p1ptr->exp==p2ptr->exp) // If coefficients are equal
```

```
{
coe=p1ptr->coe+p2ptr->coe; exp=p1ptr->exp; p1ptr=p1ptr->next; p2ptr=p2ptr->next;
}
else if(p1ptr->exp>p2ptr->exp)
{
coe=p1ptr->coe; exp=p1ptr->exp; p1ptr=p1ptr->next;
}
else if(p1ptr->exp<p2ptr->exp)
{
coe=p2ptr->coe; exp=p2ptr->exp; p2ptr=p2ptr->next;
}
ptrn=new node; if(start==NULL)
start=ptrn; ptrn->coe=coe; ptrn->exp=exp; ptrn->next=NULL; ptrp->next=ptrn; ptrp=ptrn;
} // End of While if(p1ptr==NULL)
{
while(p2ptr!=NULL)
{
coe=p2ptr->coe; exp=p2ptr->exp; p2ptr=p2ptr->next; ptrn=new node; if(start==NULL)
start=ptrn; ptrn->coe=coe; ptrn->exp=exp;
ptrn->next=NULL; ptrp->next=ptrn; ptrp=ptrn;
}
}
else if(p2ptr==NULL)
{
while(p1ptr!=NULL)
{
coe=p1ptr->coe; exp=p1ptr->exp; p1ptr=p1ptr->next; ptrn=new node; if(start==NULL)
start=ptrn; ptrn->coe=coe; ptrn->exp=exp;
ptrn->next=NULL; ptrp->next=ptrn; ptrp=ptrn;
}
}
} // End of addition

// Subtract two polynomials
void polynomial::subtract(polynomial p1,polynomial p2) // Subtract
{
struct node *p1ptr,*p2ptr; int coe,exp; ptrn=ptrp=start=NULL; p1ptr=p1.start; p2ptr=p2.start;
while(p1ptr!=NULL && p2ptr!=NULL)
{
if(p1ptr->exp==p2ptr->exp) // If coefficients are equal
{
coe=p1ptr->coe-p2ptr->coe; exp=p1ptr->exp; p1ptr=p1ptr->next; p2ptr=p2ptr->next;
}
else if(p1ptr->exp>p2ptr->exp)
{
coe=p1ptr->coe; exp=p1ptr->exp; p1ptr=p1ptr->next;
}
else if(p1ptr->exp<p2ptr->exp)
{
coe=0-p2ptr->coe; exp=p2ptr->exp; p2ptr=p2ptr->next;
}
ptrn=new node; if(start==NULL)
```

```
start=ptrn; ptrn->coe=coe; ptrn->exp=exp; ptrn->next=NULL; ptrp->next=ptrn; ptrp=ptrn;
} // End of While if(p1ptr==NULL)
{
while(p2ptr!=NULL)
{
coe=0-p2ptr->coe; exp=p2ptr->exp; p2ptr=p2ptr->next; ptrn=new node; if(start==NULL)
start=ptrn; ptrn->coe=coe; ptrn->exp=exp;
ptrn->next=NULL; ptrp->next=ptrn; ptrp=ptrn;
}
}
else if(p2ptr==NULL)
{
while(p1ptr!=NULL)
{
coe=p1ptr->coe; exp=p1ptr->exp; p1ptr=p1ptr->next; ptrn=new node; if(start==NULL)
start=ptrn; ptrn->coe=coe; ptrn->exp=exp;
ptrn->next=NULL; ptrp->next=ptrn; ptrp=ptrn;
}
}
} // End of subtraction


int main()
{
clrscr();
polynomial p1,p2,sum,diff; cout<<"\nFirst Polynomial"; p1.get_poly(); cout<<"\nSecond
polynomial."; p2.get_poly();
cout<<"\nThe First polynomial is: "; p1.show();
cout<<"\nThe second polynomial is: "; p2.show();
cout<<"\nThe sum of two polynomials is: "; sum.add(p1,p2);
sum.show();
cout<<"\nThe difference of two polynomials is: "; diff.subtract(p1,p2);
diff.show();
getch();
return 0;
}
```

Expected Output:

```
First Polynomial
Enter the coefficient: 2 Enter the exponent: 2
Enter y to add more nodes: 3

Second polynomial. Enter the coefficient: 3

Enter the exponent: 3

Enter y to add more nodes: 4 The First polynomial is: 2X^2 +
The second polynomial is: 3X^3 +
The sum of two polynomials is: 3X^3 + 2X^2 +
            The difference of two polynomials is: -3X^3 + 2X^2 +
```

Program 6:Implementation of Linear search and Binary search

Aim: Given an array arr[] of n elements, write a function to search a given element x in arr[] using linear search.

Description :

A simple approach is to do a **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

6.A) Linear Search

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

// linear search template function
// arr: array
// n: size of array
// x: value to search
// the function returns the index of x if it is present in arr
// otherwise it returns -1
template <class T>
int LinearSearch(T arr[], int n, T x) {

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

if (arr[i] == x)
return i;

}

return -1;

}

int main() {

int arr[] = { 6 , 43 ,23 ,6, 12 ,43, 3, 4, 2, 6 };
int n, index, x;
n = sizeof(arr) / sizeof(int); // size of arr

cout << "Integer Array: ";
for (int i = 0; i < n; ++i)      cout << arr[i] << ' ';
cout << endl;

cout << "Enter Value you want to search: ";
cin >> x;

index = LinearSearch(arr, n, x);
if (index != -1)
cout << x << " is present in the array at position " << index << endl;
else
cout << x << " is not present in the array \n" << endl;

/*---------------------------------------------------------------------------*/

char charArr[] = { 'A', 'v', 'D', 'R', 'T','u', 'j', 'o' };
char c;
n = sizeof(charArr) / sizeof(char);

cout << "Char Array: ";
for (int i = 0; i < n; ++i)      cout << charArr[i] << ' ';
cout << endl;

cout << "Enter character you want to search: ";
cin >> c;

index = LinearSearch(charArr, n, c);

if (index != -1)
cout << c << " is present in the array at position " << index << endl;
else
cout << c << " is not present in the array " << endl;

}
```

Expected Output:

```
Integer Array: 6 43 23 6 12 43 3 4 2 6
Enter Value you want to search: 43
43 is present in the array at position 1
Char Array: A v D R T u j o
Enter character you want to search: t
t is not present in the array
```

6.B)Aim :Program to search for an element in an array using Binary search. The elements in the array has to be considered in sorted order.

```
#include<iostream>
using namespace std;
// binary search function using template
// n: size of arr
// x: value to search
// the fucntion returns -1 if x is not found in arr
// otherwise it returns index of x
template<typename T>
int binary_search(T arr[],int n,T x)
{
int start = 0;
int end = n-1;
while(start<=end)
{
int mid = (start+end)/2;
if(arr[mid]==x)
return mid;
else if(arr[mid]<x)
start = mid + 1;
else
end = mid - 1;
}
return -1;
}

// Template function to print array
// n: size of arr
template<typename T>
void PrintArray(T arr[], int n)
{
for (int i = 0; i < n; ++i)
cout << arr[i] << " ";
cout << "\n\n";
}
int main()
{
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ,12 };
int n = sizeof(arr) / sizeof(int);
cout << "Array : " << endl;
PrintArray(arr, n);
int x, index;
cout<<"Enter value you want to Search: ";
cin>>x;

index = binary_search(arr, n, x);

if(index==-1)
cout<<x<<" is not present in the array"<<endl;
else
cout<<x<<" is present in the array at position "<<index<<endl;
}
```

Expected Output:

```
Array :
1 2 3 4 5 6 7 8 9 10 11 12
Enter value you want to Search: 2
2 is present in the array at position 1

Array :
1 2 3 4 5 6 7 8 9 10 11 12

Enter value you want to Search: 20
20 is not present in the array
```

Program 7:Implementation of Hashing Technique

Aim: Implementation of Hashing Technique

Description :

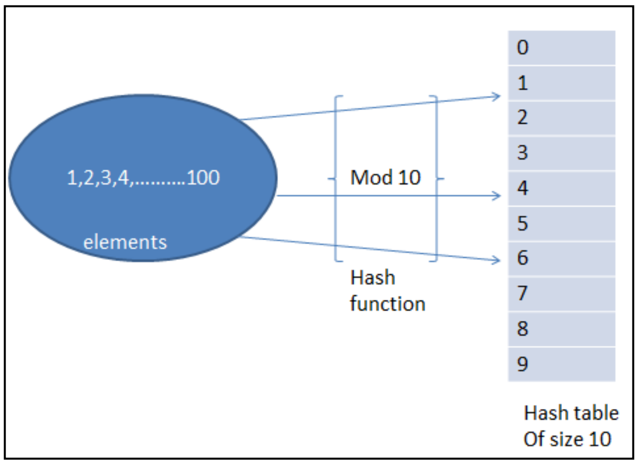
Hashing: Hashing is an important concept in Computer Science. A *Hash Table* is a data structure that allows you to store and retrieve data very quickly. There are three components that are involved with performing storage and retrieval with Hash Tables:

A hash table. This is a fixed size table that stores data of a giventype.
A hash function: This is a function that converts a piece of data into an integer. Sometimes we call this integer a hash value. The integer should be at least as big as the hash table. When we store a value in a hash table, we compute its hash value with the hash function, take that value modulo the hash table size, and that's where we store/retrieve thedata.
A collision resolution strategy: There are times when two pieces of data have hash values that, when taken modulo the hash table size, yield the same value. That is called a collision. You need to handle collisions. We will detail four collision resolution strategies: Separate chaining, linear probing, quadratic probing, and doublehashing.

Example

We have numbers from 1 to 100 and hash table of size 10. Hash function is mod 10. That means number 23 will be mapped to (23 mod 10 = 3) 3rd index of hash table.

•



But problem is if elements (for example) 2, 12, 22, 32, elements need to be inserted then they try to insert at index 2 only. This problem is called Collision. To solve this collision problem we use different types of hash function techniques. Those are given below.

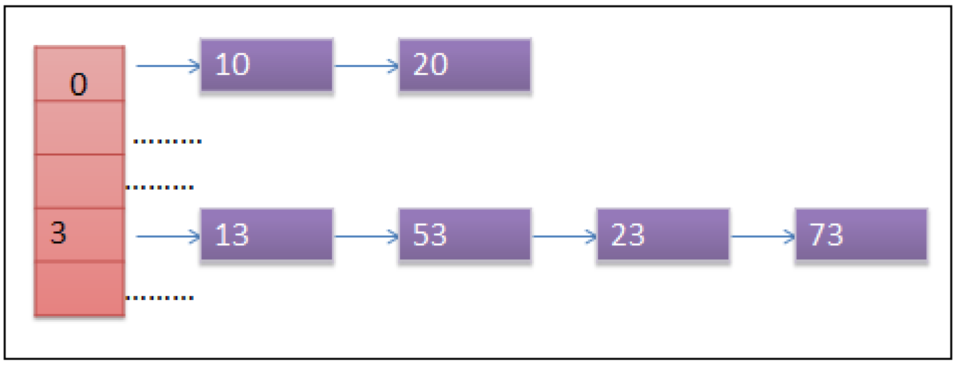
•Chaining

- Open addressing
- Linear probing

a.Quadratic probing
b.Double hashing

These also called collision resolution techniques.

Chaining
In hash table instead of putting one element in index we maintain a linked list. When collision happened we place that element in corresponding linked list. Here some space is wasted because of pointers.



Aim:Program to implement hashing with chaining

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

class Hash
{
int BUCKET; // No. of buckets

// Pointer to an array containing buckets
list<int> *table;
public:
Hash(int V); // Constructor

// inserts a key into hash table
void insertItem(int x);

// deletes a key from hash table
void deleteItem(int key);

// hash function to map values to key
int hashFunction(int x) {
return (x % BUCKET);
}

void displayHash();
};
```

```
Hash::Hash(int b)
{
this->BUCKET = b;
table = new list<int>[BUCKET];
}

void Hash::insertItem(int key)
{
int index = hashFunction(key);
table[index].push_back(key);
}

void Hash::deleteItem(int key)
{
// get the hash index of key
int index = hashFunction(key);

// find the key in (inex)th list
list <int> :: iterator i;
for (i = table[index].begin();
i != table[index].end(); i++) {
if (*i == key)
break;
}

// if key is found in hash table, remove it
if (i != table[index].end())
table[index].erase(i);
}

// function to display hash table
void Hash::displayHash() {
for (int i = 0; i< BUCKET; i++) {
cout<<i;
for (auto x : table[i])
cout<< " --> " << x;
cout<< endl;
}
}

// Driver program
int main()
{
// array that contains keys to be mapped
int a[] = {15, 11, 27, 8, 12};
int n = sizeof(a)/sizeof(a[0]);

// insert the keys into the hash table
Hash h(7); // 7 is count of buckets in
// hash table
for (int i = 0; i< n; i++)
h.insertItem(a[i]);
// delete 12 from hash table
h.deleteItem(12);
// display the Hash table
h.displayHash();
return 0;
}
```

Expected Output:

```
0
1 --> 15 --> 8
2
3
4 --> 11
5
6 --> 27
```

Program 8:Implementation of BinaryTree and Binary Tree Traversal techniques (in order, pre order, post order, level-order)

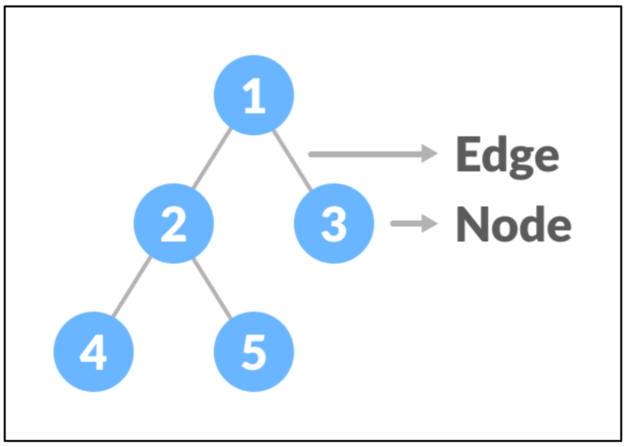
Aim: Implementation of BinaryTree and Binary Tree Traversal techniques (in order, pre order, post order, level-order)

Description :

Trees: A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges**.**

Node : A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called leaf nodes or external nodes that do not contain a link/pointer to child nodes.The node having at least a child node is called an internal node.
Edge : It is the link between any two nodes.



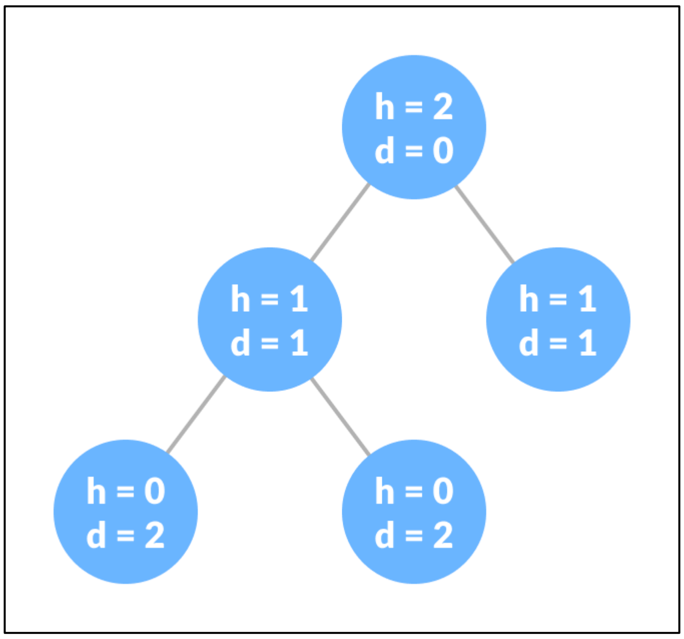
Nodes and edges of a tree

Root: It is the topmost node of a tree**.**

Height of a Node : The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

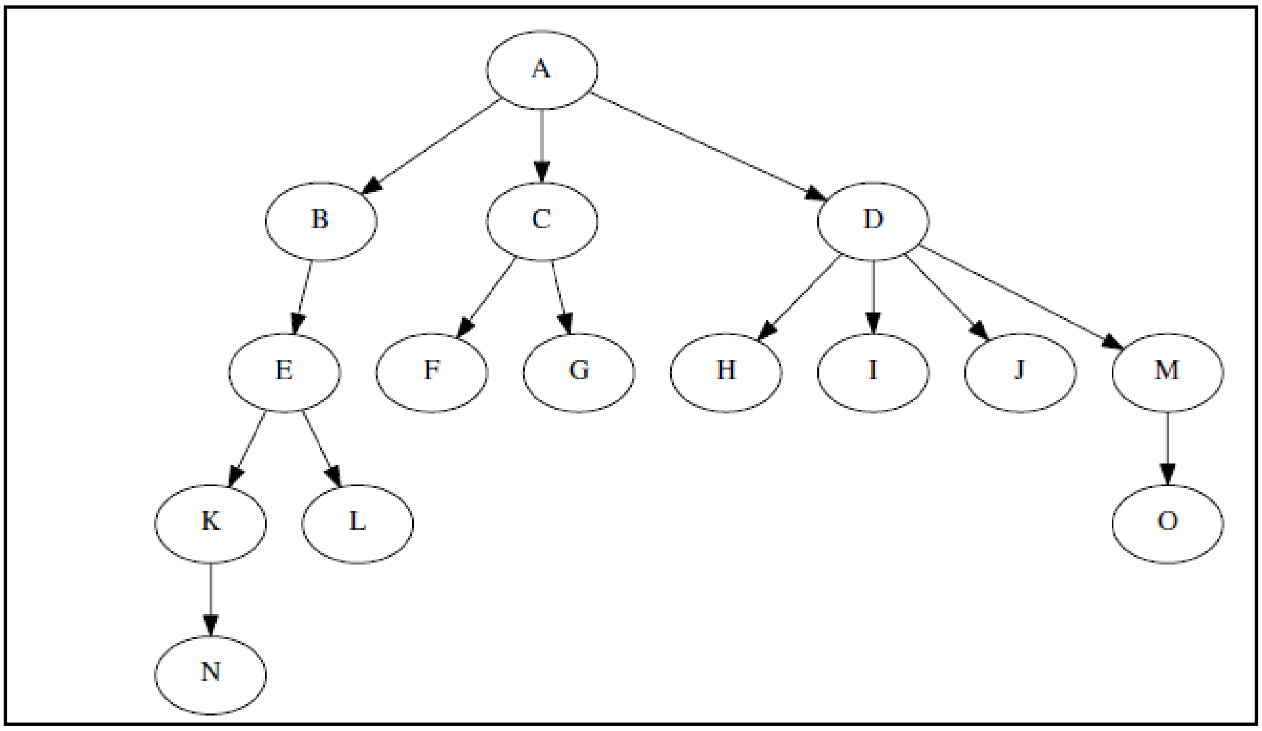
Depth of a Node : The depth of a node is the number of edges from the root to the node.

Height of a Tree: The height of a Tree is the height of the root node or the depth of the deepest node**.**

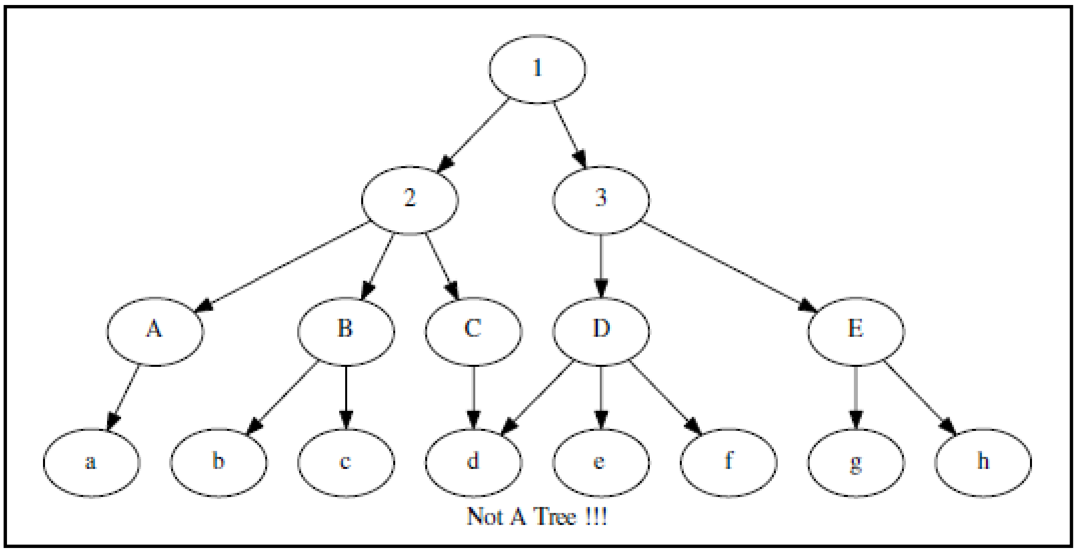


Height and depth of each node in a tree

Degree of a Node: The degree of a node is the total number of branches of that node**.**



An example of a Tree with 15 nodes

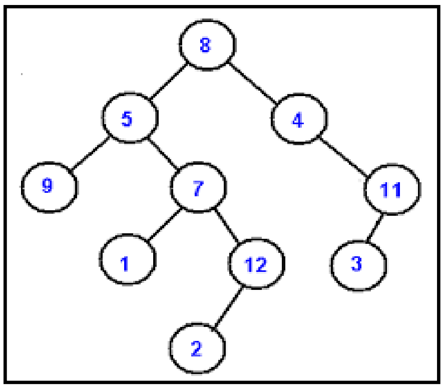


This is not a Tree, as it has multiple paths between a pair of nodes.

Binary Trees: A binary tree is a tree data structure in which each parent node can have at most two children.A binary tree is a special kind of tree in which each node can have at most two children: they are distinguished as a left child and a right child. The subtree rooted at the left child of a node is called its left subtree and the subtree rooted at the right child of a node is called its rightsubtree.

Level of a node refers to the distance of a node from the root. The level of the root is 1. Children of the root are at level 2, grandchildren or children of the children of the root are at level 3, etc.

Maximum number of nodes on a level *i* of a binary tree is 2^{i-1}. Also the maximum number of nodes in a binary tree of depth k is $2^k – 1$, k>0.



A) Binary Tree

Aim :Program to create a Binary tree and perform inorder,preorder,postorder traversal on the tree.

```
#include<stdio.h>
#include<stdlib.h>
typedef char EleType;
typedef struct BiTNode{
EleType data;
struct BiTNode *lchild,*rchild;
}BiTNode,*BiTree;

//Create a binary tree recursively according to the pre-order traversal method
bool createBitTree(BiTree *T){
EleType data;
scanf("%c",&data);
if('#' == data)
*T = nullptr;
else{
(*T) = (BiTree)malloc(sizeof(BiTNode));
if(!(*T)){
exit(-1);
}
(*T)->data = data;
createBitTree(&(*T)->lchild);//Create the left subtree
createBitTree(&(*T)->rchild);//Create the right subtree
}
return true;
}

void visite(EleType data){
printf("%c",data);
}

//Recursively realize the first order traversal of the binary tree
bool proOrderTraverse(BiTree &T){
if(T == nullptr)
return false;
visite(T->data);
proOrderTraverse(T->lchild);
proOrderTraverse(T->rchild);

return true;
}
//Recursively implement in-order traversal of the binary tree
bool inOrderTraverse(BiTree &T){
if(T == nullptr)
return false;

inOrderTraverse(T->lchild);
visite(T->data);
inOrderTraverse(T->rchild);

return true;
}
//Recursively implement post-order traversal of the binary tree
bool postOrderTraverse(BiTree &T){
if(T == nullptr)
return false;

postOrderTraverse(T->lchild);
postOrderTraverse(T->rchild            );
visite(T->data);

return true;
}
int main(){
BiTree T;
createBitTree(&T);
proOrderTraverse(T);
printf("\n");
inOrderTraverse(T);
printf("\n");
postOrderTraverse(T);
getchar();
return 0;
}
```

Expected Output:

```
ABC##DE#G##F###
ABCDEGF
CBEGDFA
CGEFDBA
```

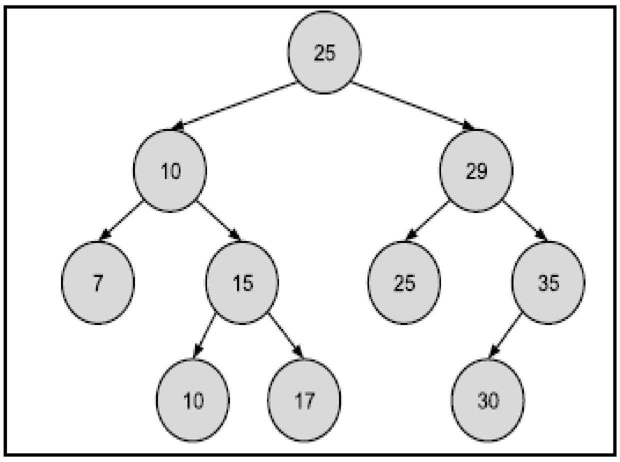
Program 9:Implementation of Binary Search Tree and its operations.

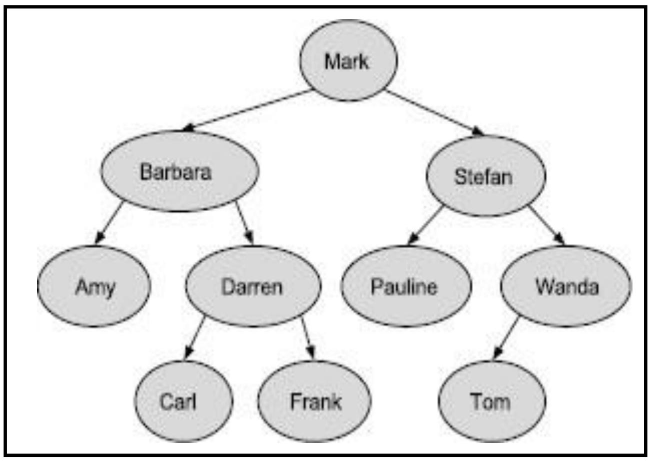
Aim: Implementation of Binary Search Tree and its operations.

Description

Binary Search Tree: A binary search tree is a binary tree which may be empty. If not empty, it satisfies the following properties:

1. Every element has a key and no two elements have the same key.
2. The keys (if any) in the left subtree are smaller than the key in theroot
3. The keys (if any) in the right subtree are greater than the key in theroot
4. The left and right subtrees are binary searchtrees.



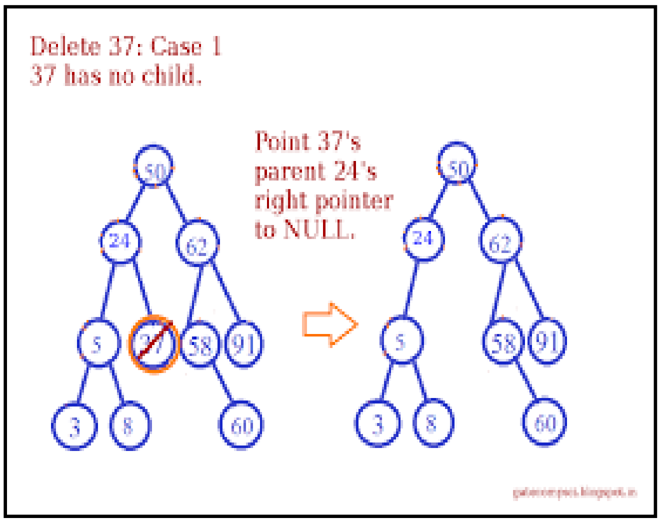


Binary Search Treeon Number Binary Search Tree onStrings

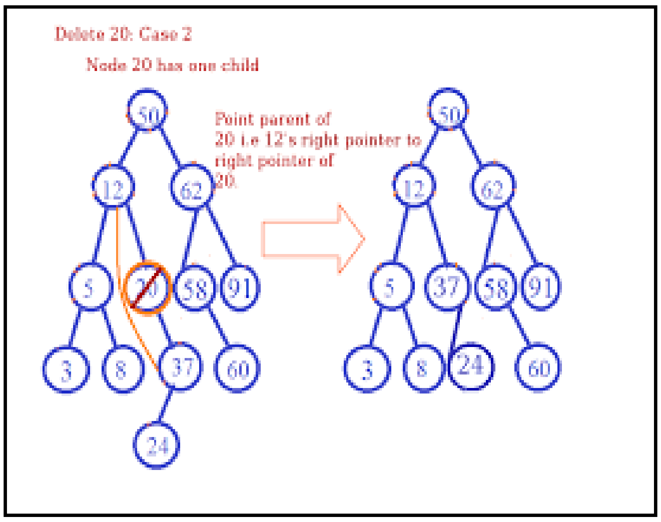
Searching a Binary Search Tree: Suppose we wish to search for an element with key x. We being at root. If the root is 0, then the search tree contains no elements and the search terminates unsuccessfully. Otherwise, we compare x with key in root. If x equals key in root, then search terminates successfully. If x is less than key in root, then no element in right sub tree can have key value x, and only left subtree is to be searched. If x is larger than key in root, the no element in left subtree can have the key x, and only right subtree is to be searched. The subtrees can be searched recursively.

Insertion into a Binary Search Tree: To insert an element x, we must first verify that its key is different from those of existing elements. To do this, a search is carried out. If search is unsuccessful, then element is inserted at point where the search terminated.

Deleting from a Binary Search Tree: Deletion from a leaf element is achieved by simply removing the leaf node and making its parent’s child field to be null. Other cases are deleting a node with one subtree and two subtrees.



Deleting aleaf node


Delete 20: Case 2
Node 20 has one child
Point parent of 20 i.e 12's right pointer to right pointer of 20.
50
12
62
5
20
58
91
3
8
37
60
24

Deleting a non leafnode.

Aim :Program in C++ to create a Binary search tree, insert a node a binary tree, delete a node , perform traversal on BST.

```
#include<iostream>

#include<stdlib.h>

using namespace std;

struct treeNode

{

int data;

treeNode *left;

treeNode *right;

};

treeNode* FindMin(treeNode *node)

{

if(node==NULL)

{

/* There is no element in the tree */
```

```
return NULL;
}
if(node->left) /* Go to the left sub tree to find the min element */
return FindMin(node->left);
else
return node;
}
treeNode* FindMax(treeNode *node)
{
if(node==NULL)
{
/* There is no element in the tree */
return NULL;
}
if(node->right) /* Go to the left sub tree to find the min element */
return(FindMax(node->right));
else
return node;
}
treeNode *Insert(treeNode *node,int data)
{
if(node==NULL)
{
treeNode *temp;
temp=new treeNode;
//temp = (treeNode *)malloc(sizeof(treeNode));
temp -> data = data;
temp -> left = temp -> right = NULL;
```

```
return temp;
}
if(data >(node->data))
{
node->right = Insert(node->right,data);
}
else if(data < (node->data))
{
node->left = Insert(node->left,data);
}
/* Else there is nothing to do as the data is already in the tree. */
return node;
}
treeNode * Delet(treeNode *node, int data)
{
treeNode *temp;
if(node==NULL)
{
cout<<"Element Not Found";
}
else if(data < node->data)
{
node->left = Delet(node->left, data);
}
else if(data > node->data)
{
node->right = Delet(node->right, data);
}
```

```
else
{
/* Now We can delete this node and replace with either minimum element
in the right sub tree or maximum element in the left subtree */
if(node->right && node->left)
{
/* Here we will replace with minimum element in the right sub tree */
temp = FindMin(node->right);
node -> data = temp->data;
/* As we replaced it with some other node, we have to delete that node */
node -> right = Delet(node->right,temp->data);
}
else
{
/* If there is only one or zero children then we can directly
remove it from the tree and connect its parent to its child */
temp = node;
if(node->left == NULL)
node = node->right;
else if(node->right == NULL)
node = node->left;
free(temp); /* temp is longer required */
}
}
return node;
}
treeNode * Find(treeNode *node, int data)
{
```

```
if(node==NULL)
{
/* Element is not found */
return NULL;
}
if(data > node->data)
{
/* Search in the right sub tree. */
return Find(node->right,data);
}
else if(data < node->data)
{
/* Search in the left sub tree. */
return Find(node->left,data);
}
else
{
/* Element Found */
return node;
}
}
void Inorder(treeNode *node)
{
if(node==NULL)
{
return;
}
Inorder(node->left);
```

```
cout<<node->data<<" ";
Inorder(node->right);
}
void Preorder(treeNode *node)
{
if(node==NULL)
{
return;
}
cout<<node->data<<" ";
Preorder(node->left);
Preorder(node->right);
}
void Postorder(treeNode *node)
{
if(node==NULL)
{
return;
}
Postorder(node->left);
Postorder(node->right);
cout<<node->data<<" ";
}
int main()
{
treeNode *root = NULL,*temp;
int ch;
//clrscr();
```

```
while(1)
{
cout<<"\n1.Insert\n2.Delete\n3.Inorder\n4.Preorder\n5.Postorder\n6.FindMin\n7.FindMax\n8.
Search\n9.Exit\n";
cout<<"Enter ur choice:";
cin>>ch;
switch(ch)
{
case 1:
cout<<"\nEnter element to be insert:";
cin>>ch;
root = Insert(root, ch);
cout<<"\nElements in BST are:";
Inorder(root);
break;
case 2:
cout<<"\nEnter element to be deleted:";
cin>>ch;
root = Delet(root,ch);
cout<<"\nAfter deletion elements in BST are:";
Inorder(root);
break;
case 3:
cout<<"\nInorder Travesals is:";
Inorder(root);
break;
case 4:
cout<<"\nPreorder Traversals is:";
```

```
Preorder(root);
break;
case 5:
cout<<"\nPostorder Traversals is:";
Postorder(root);
break;
case 6:
temp = FindMin(root);
cout<<"\nMinimum element is :"<<temp->data;
break;
case 7:
temp = FindMax(root);
cout<<"\nMaximum element is :"<<temp->data;
break;
case 8:
cout<<"\nEnter element to be searched:";
cin>>ch;
temp = Find(root,ch);
if(temp==NULL)
{
cout<<"Element is not foundn";
}
else
{
cout<<"Element "<<temp->data<<" is Found\n";
}
break;
case 9:
```

```
exit(0);
break;
default:
cout<<"\nEnter correct choice:";
break;
}
}
return 0;
}
```

Expected Output:

```
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
6.FindMin
7.FindMax
8.Search
9.Exit
Enter ur choice:1

Enter element to be insert:10
Elements in BST are:10
1.Insert
2.Delete
3.Inorder
4.Preorder
5.Postorder
```

```
6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:1

Enter element to be insert:13

Elements in BST are:10 13

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:1

Enter element to be insert:12

Elements in BST are:10 12 13

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit
```

```
Enter ur choice:1

Enter element to be insert:14

Elements in BST are:10 12 13 14

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:1

Enter element to be insert:9

Elements in BST are:9 10 12 13 14

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:1

Enter element to be insert:7

Elements in BST are:7 9 10 12 13 14

1.Insert
```

```
2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:3

Inorder Travesals is:7 9 10 12 13 14

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:4

Preorder Traversals is:10 9 7 13 12 14

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax
```

```
8.Search

9.Exit

Enter ur choice:5

Postorder Traversals is:7 9 12 14 13 10

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:6

Minimum element is :7

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:7

Maximum element is :14

1.Insert

2.Delete
```

```
3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:8

Enter element to be searched:12

Element 12 is Found

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin

7.FindMax

8.Search

9.Exit

Enter ur choice:2

Enter element to be deleted:12

After deletion elements in BST are:7 9 10 13 14

1.Insert

2.Delete

3.Inorder

4.Preorder

5.Postorder

6.FindMin
```

```
7.FindMax

8.Search

9.Exit

Enter ur choice:9
```

Program 10. Implementation of Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Quick Sort, Heap Sort

Aim: Implementation of Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Quick Sort, Heap Sort

Description

Sorting is nothing but arranging the data in ascending or descending order.**Sorting** arranges data in a sequence which makes searching easier.

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques:

- Insertion Sort
- Selection Sort
- Bubble Sort
- Quick Sort
- Merge Sort
- Heap Sort

10.A)Aim: Program to sort elements of an array using Insertion sort.

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

int main ()
{

int i,j, k,temp;
int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};

cout<<"\nprinting sorted elements...\n";
for(k=1; k<10; k++)

{
temp = a[k];

j= k-1;
while(j>=0 && temp <= a[j])

{
a[j+1] = a[j];

j = j-1;
}

a[j+1] = temp;
}

for(i=0;i<10;i++)
{
cout <<a[i]<<"\n";
}

}
```

Expected Output:

```
printing sorted elements...

7

9

10

12

23

23

34

44

78

101
```

10.B)Aim: Program to sort elements of an array using Selection Sort

```
#include<iostream>
using namespace std;
int smallest(int[],int,int);
int main ()
{
int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
int i,j,k,pos,temp;
for(i=0;i<10;i++)
{
pos = smallest(a,10,i);
temp = a[i];
a[i]=a[pos];
a[pos] = temp;
}
cout<<"\n printing sorted elements...\n";
for(i=0;i<10;i++)
{
```

```
cout<<a[i]<<"\n";
}
return 0;
}
int smallest(int a[], int n, int i)
{
int small,pos,j;
small = a[i];
pos = i;
for(j=i+1;j<10;j++)
{
if(a[j]<small)
{
small = a[j];
pos=j;
}
}
return pos;
}
```

Expected Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

10.C) Aim: Write a Program to sort the elements of an array using Bubble Sort

```
#include<iostream>
usingnamespacestd;
intmain()
{
inta[50],n,i,j,temp;
cout<<"Enter the size of array: ";
cin>>n;
cout<<"Enter the array elements: ";
for(i=0;i<n;++i)
cin>>a[i];
for(i=0;i<n;++i)
{
for(j=0;j<(n-i-1);++j)
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
cout<<"Array after bubble sort:";
for(i=0;i<n;++i)
cout<<" "<<a[i];
return0;
}
```

Expected Output:

```
Enter the size of array: 10
Enter the array elements: 2
1
4
0
-1
3
5
7
6
10
Array after bubble sort: -1 0 1 2 3 4 5 6 7 10
```

10.D)Aim:Program to sort the elements of an array using Quick sort

```
#include<iostream>
#include<vector>
using namespace std;
// template function to find the position of pivot element
// last element is taken as pivot
template <typename T>
int Partition(T arr[], int start, int end){
int pivot = end;
int j = start;
for(int i=start;i<end;++i){
```

```
if(arr[i]<arr[pivot]){
swap(arr[i],arr[j]);
++j;
}
}
swap(arr[j],arr[pivot]);
return j;
}
// template function to perform quick sort on array arr
template <typename T>
void Quicksort(T arr[], int start, int end ){
if(start<end){
int p = Partition(arr,start,end);
Quicksort(arr,start,p-1);
Quicksort(arr,p+1,end);
}
}
// Template function to print array
// n: size of arr[]
template <typename T>
void PrintArray(T arr[], int n){
for(int i=0;i<n;++i)
cout<<arr[i]<<" ";
cout<<"\n\n";
}
int main() {
int arr[] = { 1 , 10 , 11 , 9 , 14 , 3 , 2 , 20 , 19, 43, 57, 3, 2 };
int n = sizeof(arr)/sizeof(int);
cout<<"Array Before Sorting: "<<endl;
PrintArray(arr, n);
Quicksort(arr,0,n-1);
cout<<"Array After Sorting: "<<endl;
PrintArray(arr, n);
}
```

Expected Output:

```
Array Before Sorting:
1 10 11 9 14 3 2 20 19 43 57 3 2
Array After Sorting:
1 2 2 3 3 9 10 11 14 19 20 43 57
```

10.E) Aim:Program to sort elements of array using merge sort

```
#include <iostream>
using namespace std;
// A function to merge the two half into a sorted data.
void Merge(int *a, int low, int high, int mid)
{
// We have low to mid and mid+1 to high already sorted.
int i, j, k, temp[high-low+1];
i = low;
k = 0;
j = mid + 1;
// Merge the two parts into temp[].
while (i <= mid && j <= high)
{
if (a[i] < a[j])
{
temp[k] = a[i];
k++;
i++;
}
else
{
temp[k] = a[j];
k++;
j++;
}
}
// Insert all the remaining values from i to mid into temp[].
while (i <= mid)
{
temp[k] = a[i];
k++;
i++;
}
// Insert all the remaining values from j to high into temp[].
while (j <= high)
{
temp[k] = a[j];
k++;
j++;
}
// Assign sorted data stored in temp[] to a[].
for (i = low; i <= high; i++)
{
a[i] = temp[i-low];
}
}
// A function to split array into two parts.
void MergeSort(int *a, int low, int high)
{
int mid;
if (low < high)
{
mid=(low+high)/2;
// Split the data into two half.
MergeSort(a, low, mid);
MergeSort(a, mid+1, high);

// Merge them to get sorted output.
Merge(a, low, high, mid);
}
}
int main()
{
int n, i;
cout<<"\nEnter the number of data element to be sorted: ";
cin>>n;
int arr[n];
for(i = 0; i < n; i++)
{
cout<<"Enter element "<<i+1<<": ";
cin>>arr[i];
}
MergeSort(arr, 0, n-1);
// Printing the sorted data.
cout<<"\nSorted Data ";
for (i = 0; i < n; i++)
cout<<"->"<<arr[i];
return 0;
```

```
}
```

Expected Output:

```
Enter the number of data element to be sorted: 5
Enter element 1: 6
Enter element 2: 4
Enter element 3: 9
Enter element 4: 0
Enter element 5: 1
Sorted Data ->0->1->4->6->9
```

10.F) Aim:Program to sort elements of array using Heap sort

```
#include <iostream>
using namespace std;
// A function to heapify the array.
void MaxHeapify(int a[], int i, int n)
{
int j, temp;
temp = a[i];
j = 2*i;
while (j <= n)
{
if (j < n && a[j+1] > a[j])
j = j+1;
// Break if parent value is already greater than child value.
if (temp > a[j])
break;
// Switching value with the parent node if temp < a[j].
else if (temp <= a[j])
{
a[j/2] = a[j];
j = 2*j;
}
}
a[j/2] = temp;
return;
}
void HeapSort(int a[], int n)
{
int i, temp;
for (i = n; i >= 2; i--)
{
// Storing maximum value at the end.
temp = a[i];
a[i] = a[1];
a[1] = temp;
// Building max heap of remaining element.
MaxHeapify(a, 1, i - 1);
}
}
void Build_MaxHeap(int a[], int n)
{
int i;
for(i = n/2; i >= 1; i--)
MaxHeapify(a, i, n);
}
int main()
{
int n, i;
cout<<"\nEnter the number of data element to be sorted: ";
cin>>n;
n++;
int arr[n];
for(i = 1; i < n; i++)
{
cout<<"Enter element "<<i<<": ";
cin>>arr[i];
}
// Building max heap.
Build_MaxHeap(arr, n-1);
HeapSort(arr, n-1);

// Printing the sorted data.
cout<<"\nSorted Data ";
for (i = 1; i < n; i++)
cout<<"->"<<arr[i];
return 0;
}
```

Expected Output:

```
Enter the number of data element to be sorted: 5
Enter element 1: 4
Enter element 2: 2
Enter element 3: 1
Enter element 4: 5
Enter element 5: 0

Sorted Data ->0->1->2->4->5
```

Program 11. Implementation of operations on AVL Trees.

Aim: Implementation of operations on AVL Trees

Discription:

AVL Tree

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

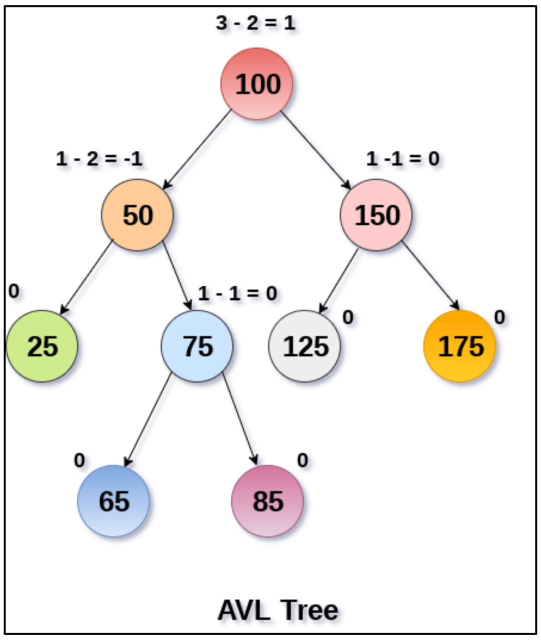
Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

Insertion: Insertion in AVL tree is performed in the same way as it is performed in a binary search tree.However , it may lead to violation in the AVL tree property and therefore the tree may need balancing, The tree can be balanced by applying rotations.
Deletion : Deletion can be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

Why AVL Tree?

AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

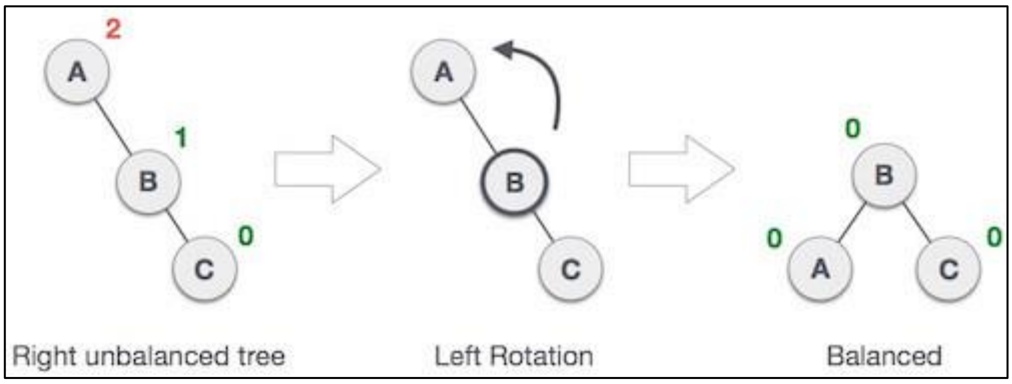
1. L L rotation: Inserted node is in the left subtree of left subtree of A
2. R R rotation : Inserted node is in the right subtree of right subtree of A
3. L R rotation : Inserted node is in the right subtree of left subtree of A
4. R L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2.

1. RR Rotation

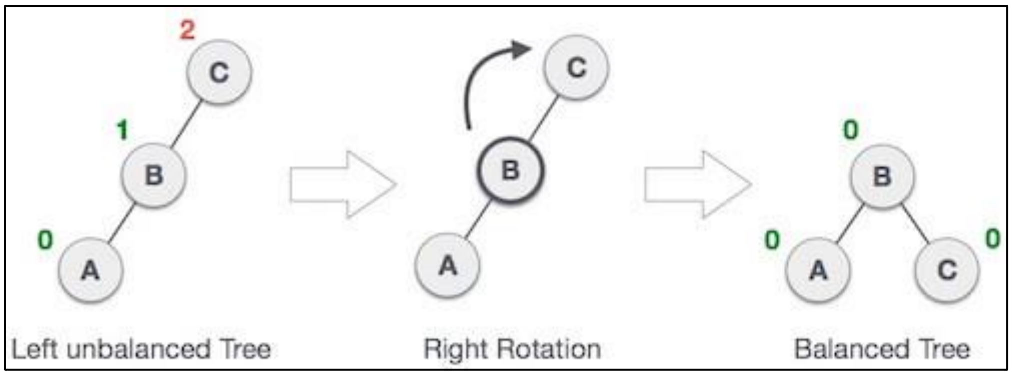
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

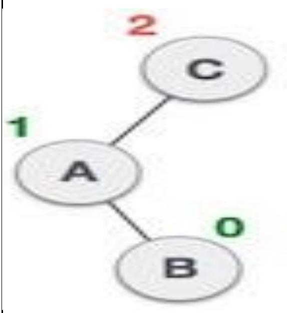
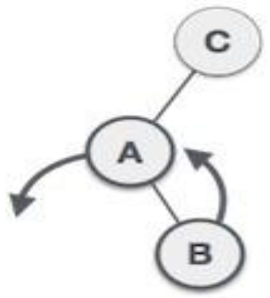
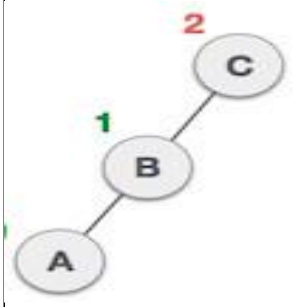
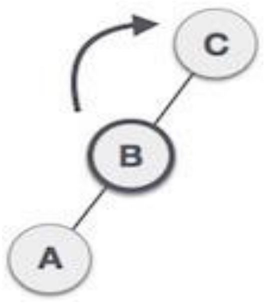
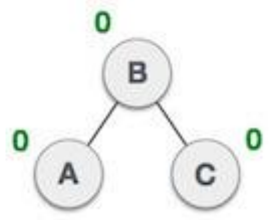


In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

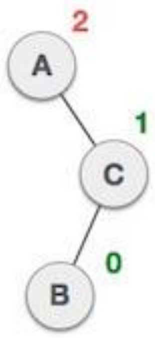
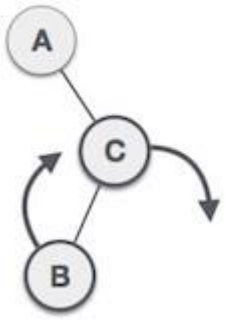
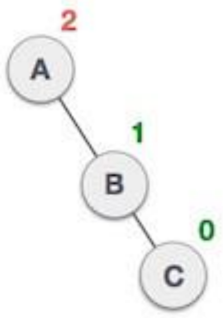
Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

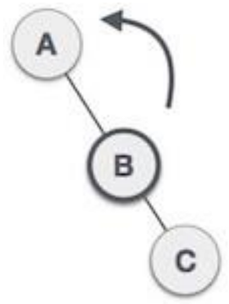
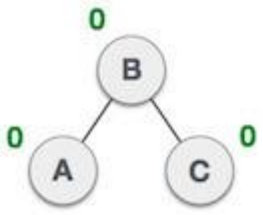
Let us understand each and every step very clearly:

	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C**
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

	A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**.
	After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.

	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

AIM:Write a program to create an AVL tree , perform insertion , deletion operations and wherever required perform rotations to make the tree balanced.

```
#include<iostream>
#include<cstdio>
#include<sstream>
#include<algorithm>
#define pow2(n) (1 << (n))
using namespace std;
/*
* Node Declaration
*/
struct avl_node
{
int data;
struct avl_node *left;
struct avl_node *right;
}*root;

/*
* Class Declaration
*/
class avlTree
{
public:
```

```
int height(avl_node *);
int diff(avl_node *);
avl_node *rr_rotation(avl_node *);
avl_node *ll_rotation(avl_node *);
avl_node *lr_rotation(avl_node *);
avl_node *rl_rotation(avl_node *);
avl_node* balance(avl_node *);
avl_node* insert(avl_node *, int );
void display(avl_node *, int);
void inorder(avl_node *);
void preorder(avl_node *);
void postorder(avl_node *);
avlTree()
{
root = NULL;
}
};

/*
* Main Contains Menu
*/
int main()
{
int choice, item;
avlTree avl;
while (1)
{
cout<<"\n---------------------"<<endl;
cout<<"AVL Tree Implementation"<<endl;
cout<<"\n---------------------"<<endl;
cout<<"1.Insert Element into the tree"<<endl;
cout<<"2.Display Balanced AVL Tree"<<endl;
cout<<"3.InOrder traversal"<<endl;
cout<<"4.PreOrder traversal"<<endl;
cout<<"5.PostOrder traversal"<<endl;
cout<<"6.Exit"<<endl;
cout<<"Enter your Choice: ";
cin>>choice;
switch(choice)
{
case 1:
cout<<"Enter value to be inserted: ";
cin>>item;
root = avl.insert(root, item);
break;
case 2:
if (root == NULL)
{
cout<<"Tree is Empty"<<endl;
continue;
}
cout<<"Balanced AVL Tree:"<<endl;
avl.display(root, 1);
break;
case 3:
cout<<"Inorder Traversal:"<<endl;
avl.inorder(root);
cout<<endl;
break;
case 4:
cout<<"Preorder Traversal:"<<endl;
avl.preorder(root);
cout<<endl;
break;
case 5:
cout<<"Postorder Traversal:"<<endl;
avl.postorder(root);
cout<<endl;
break;
case 6:
exit(1);
break;
default:
cout<<"Wrong Choice"<<endl;
}
}
return 0;
}
```

```
/*
* Height of AVL Tree
*/
int avlTree::height(avl_node *temp)
{
int h = 0;
if (temp != NULL)
{
int l_height = height (temp->left);
int r_height = height (temp->right);
int max_height = max (l_height, r_height);
h = max_height + 1;
}
return h;
}

/*
* Height Difference
*/
int avlTree::diff(avl_node *temp)
{
int l_height = height (temp->left);
int r_height = height (temp->right);
int b_factor= l_height - r_height;
return b_factor;
}

/*
* Right- Right Rotation
*/
avl_node *avlTree::rr_rotation(avl_node *parent)
{
avl_node *temp;
temp = parent->right;
parent->right = temp->left;
temp->left = parent;
return temp;
}
```

```
/*
* Left- Left Rotation
*/
avl_node *avlTree::ll_rotation(avl_node *parent)
{
avl_node *temp;
temp = parent->left;
parent->left = temp->right;
temp->right = parent;
return temp;
}

/*
* Left - Right Rotation
*/
avl_node *avlTree::lr_rotation(avl_node *parent)
{
avl_node *temp;
temp = parent->left;
parent->left = rr_rotation (temp);
return ll_rotation (parent);
}
/*
* Right- Left Rotation
*/
avl_node *avlTree::rl_rotation(avl_node *parent)
{
avl_node *temp;
temp = parent->right;
parent->right = ll_rotation (temp);
return rr_rotation (parent);
}
/*
* Balancing AVL Tree
*/
avl_node *avlTree::balance(avl_node *temp)
{
int bal_factor = diff (temp);
if (bal_factor > 1)
```

```
{
if (diff (temp->left) > 0)
temp = ll_rotation (temp);
else
temp = lr_rotation (temp);
}
else if (bal_factor < -1)
{
if (diff (temp->right) > 0)
temp = rl_rotation (temp);
else
temp = rr_rotation (temp);
}
return temp;
}
/*
* Insert Element into the tree
*/
avl_node *avlTree::insert(avl_node *root, int value)
{
if (root == NULL)
{
root = new avl_node;
root->data = value;
root->left = NULL;
root->right = NULL;
return root;
}
else if (value < root->data)
{
root->left = insert(root->left, value);
root = balance (root);
}
else if (value >= root->data)
{
root->right = insert(root->right, value);
root = balance (root);
}
return root;
```

```
}
/*
* Display AVL Tree
*/
void avlTree::display(avl_node *ptr, int level)
{
int i;
if (ptr!=NULL)
{
display(ptr->right, level + 1);
printf("\n");
if (ptr == root)
cout<<"Root -> ";
for (i = 0; i < level && ptr != root; i++)
cout<<"        ";
cout<<ptr->data;
display(ptr->left, level + 1);
}
}
/*
* Inorder Traversal of AVL Tree
*/
void avlTree::inorder(avl_node *tree)
{
if (tree == NULL)
return;
inorder (tree->left);
cout<<tree->data<<"  ";
inorder (tree->right);
}
/*
* Preorder Traversal of AVL Tree
*/
void avlTree::preorder(avl_node *tree)
{
if (tree == NULL)
return;
cout<<tree->data<<"  ";
preorder (tree->left);
preorder (tree->right);

}
/*
* Postorder Traversal of AVL Tree
*/
void avlTree::postorder(avl_node *tree)
{
if (tree == NULL)
return;
postorder ( tree ->left );
postorder ( tree ->right );
cout<<tree->data<<"  ";
}
```

Expected Output:

```
---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 10
---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 7
---------------------
AVL Tree Implementation
```

```
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 8
---------------------
AVL Tree Implementation
--------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 12
---------------------
AVL Tree Implementation
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 1
Enter value to be inserted: 13
---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
```

```
Enter your Choice: 1
Enter value to be inserted: 14
---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice: 2
Balanced AVL Tree:
14
13
Root -> 12
10
8
7
---------------------
AVL Tree Implementation
---------------------
1.Insert Element into the tree
2.Display Balanced AVL Tree
3.InOrder traversal
4.PreOrder traversal
5.PostOrder traversal
6.Exit
Enter your Choice:6
```

Program 12. Implementation of Graph Search Methods.

Aim: Implementation of Graph Search Methods

Discription

Graph: Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

Graph is a collection of vertices and arcs in which vertices are connected with arcs. Graph is a collection of nodes and edges in which nodes are connected with edges.

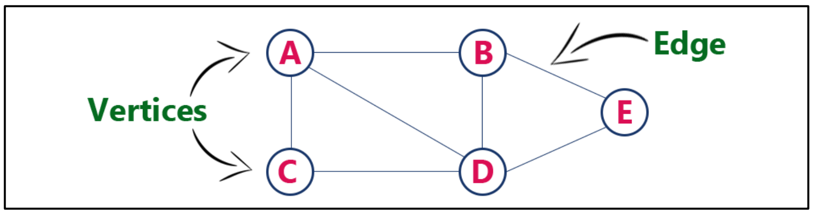
Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.

Example:

The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Traversal :

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path. There are two graph traversal techniques and they are as follows...

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

DFS (Depth First Search)

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

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

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

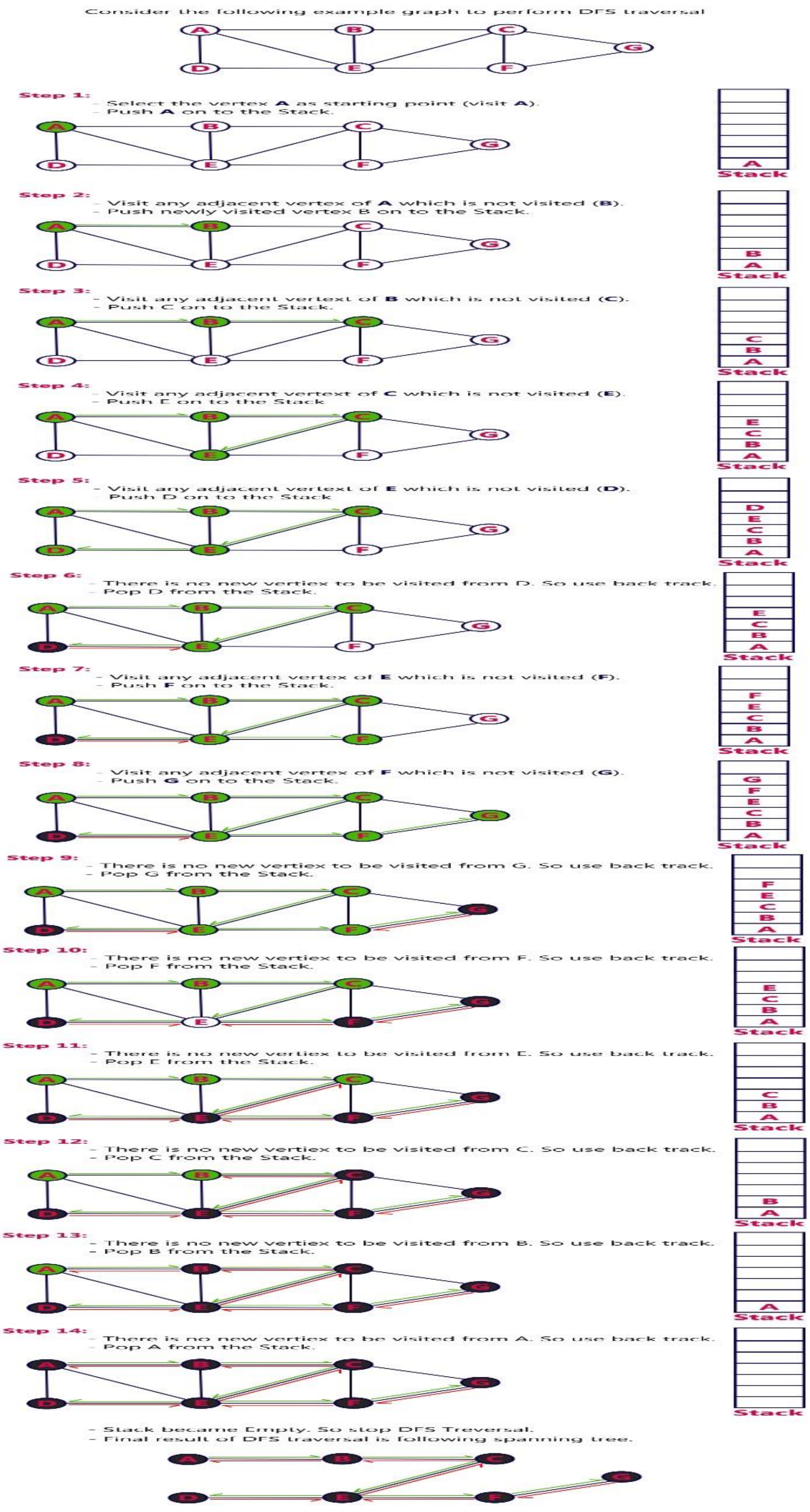
Step 3 - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph



Consider the following example graph to perform DFS traversal

Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

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

12.A) Aim: Write a C++ Program to implement Depth first Search

```
#include<iostream>
using namespace std;
class graph
{
int a[10][10],n,start;
public:
void getdata();
void dfs_traverse();
};
void graph::getdata()
{
cout<<"Enter the number of vertices in the graph ";
cin>>n;
cout<<"Enter the adjacency matrix of graph "<<endl;
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
cin>>a[i][j];
cout<<"Enter the vertex from which you want to traverse ";
cin>>start;
}
void graph::dfs_traverse()
{
int *visited= new int[n];
int stack[10],top=-1,i;
for(int j=0;j<n;j++)
visited[j]=0;
cout<<"The Depth First Search Traversal : "<<endl;
i=stack[++top]=start;
visited[start]=1;
while(top>=0)
{
i=stack[top];
cout<<stack[top--]<<endl;
for(int j=n-1;j>=0;j--)
if(a[i][j]!=0&&visited[j]!=1)
{
stack[++top]=j;
visited[j]=1;
}
}
}
int main()
{
graph dfs;
dfs.getdata();
dfs.dfs_traverse();
return 0;
}
```

Expected Output:

```
Enter the number of vertices in the graph 5

Enter the adjacency matrix of graph

0 1 1 0 1

1 0 0 1 0

1 0 0 1 1

0 1 1 0 0

0 0 1 1 0

Enter the vertex from which you want to traverse 1

The Depth First Search Traversal :

1

0

2

4

3
```

BFS (Breadth First Search)
BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.
We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

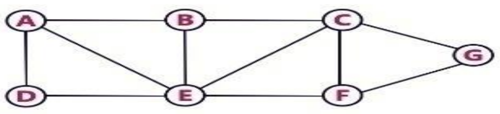
Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

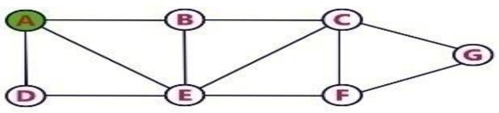
Step 5 - Repeat steps 3 and 4 until queue becomes empty.

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

Consider the following example graph to perform BFS traversal



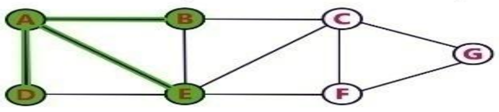
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



Queue

A						

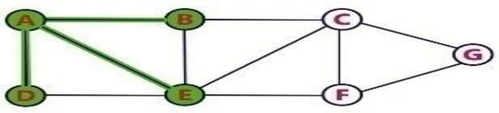
Step 2:
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



Queue

	D	E	B			

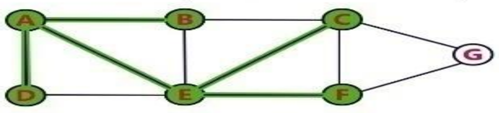
Step 3:
- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



Queue

		E	B			

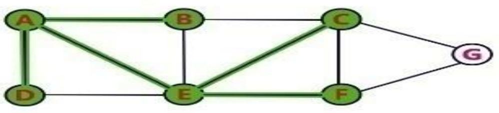
Step 4:
- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



Queue

			B	C	F	

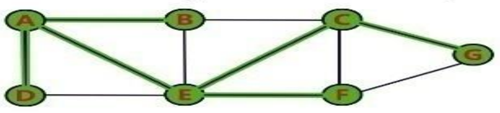
Step 5:
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



Queue

				C	F	

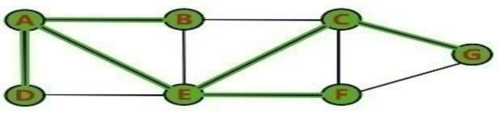
Step 6:
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



Queue

					F	G

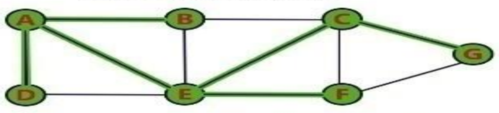
Step 7:
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



Queue

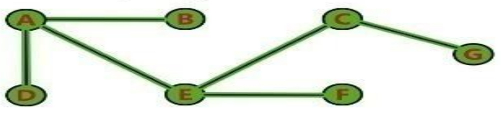
						G

Step 8:
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

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



12.B) Aim: Write a C++ program to implement Breadth first search traversal

```
#include<iostream>
using namespace std;
class graph
{
int a[10][10],n,start;
public:
void getdata();
void bfs_traverse();
};
void graph::getdata()
{
cout<<"Enter the number of vertices in the graph ";
cin>>n;
cout<<"Enter the adjacency matrix of graph "<<endl;
for(int i=0;i<n;i++)
for(int j=0;j<n;j++)
cin>>a[i][j];
cout<<"Enter the vertex from which you want to traverse ";
cin>>start;
}
void graph::bfs_traverse()
{
int *visited= new int[n];
int queue[10],front=-1,rear=0,i;
for(int j=0;j<n;j++)
visited[j]=0;
cout<<"Traversing the graph using breadth first search algorithm : "<<endl;
queue[rear]=start;
visited[start]=1;
while(front!=rear)
{
cout<<queue[++front]<<endl;
i=queue[front];
for(int j=0;j<n;j++)
if(a[i][j]!=0&&visited[j]!=1)
{
queue[++rear]=j;
visited[j]=1;
}
}
}
int main()
{
graph bfs;
bfs.getdata();
bfs.bfs_traverse();
return 0;
}
```

Expected Output:

```
Enter the number of vertices in the graph 5
Enter the adjacency matrix of graph
0 1 1 0 1
1 0 0 1 0
1 0 0 1 1
0 1 1 0 0
0 0 1 1 0
Enter the vertex from which you want to traverse 1
Traversing the graph using breadth first search algorithm :
1
0
3
2
4
```

Additional Programs

Program 13: Write a C++ Program to implement balanced parenthesis using stack

Aim: Write a C++ Program to implement balanced parenthesis using stack

Discription:

This C++ program, uses a stack data strucure, computes whether the given parantheses expression is valid or not by checking whether each parentheses is closed and nested in the input expression.

```
#include<stack>//Include STL stack
#include<iostream>
using namespace std;
class BalancedParenthesisChecker{
public:
bool isParenthesisBalanced(char s[]){

//Char STL stack
stack<char> Stack;
int i=0;

/* Traverse the given string expression
to check matching brackets */

while(s[i])
{
/*If the s[i] is a opening bracket then
push it to Stack*/

if( s[i]=='(' || s[i]=='{' || s[i]=='[' )
{
Stack.push(s[i]);
}
/* If s[i] is a opening bracket then
*get top character from stack and match it
*to the current character if match
*found then pop it from Stack else
*return false*/

if( s[i]==')' || s[i]=='}' || s[i]==']' )
{
if( Stack.empty() || !isEqual(Stack.top(),s[i]) )
{
return false;
}
else
{
//Corresponding brackets matched
//pop it from stack
Stack.pop();
}
}
i++;
}

/*If Stack is empty then parenthesis
are balanced or else not balanced
*/

return Stack.empty();
}

private:
bool isEqual(char c1, char c2){

if( c1=='(' && c2==')' )
return true;
else if(c1=='{' && c2=='}')
return true;
else if(c1=='[' && c2==']')
return true;
else
return false;
}
};

int main()
{
char str[50];
BalancedParenthesisCheckercheker;

cout<<"Enter parenthesis expression:"<<endl;
cin.getline(str,50);

bool isBalanced = cheker.isParenthesisBalanced(str);

if(isBalanced){
cout<<"Balanced"<<endl;
}else{
cout<<"Not Balanced"<<endl;
}
return 0;
}
```

Expected Output:

```
Enter parenthesis expression:
"({})"
Balanced

Enter parenthesis expression:
"(((("
Not Balanced
```

Program 14: T**o sort a singly linked list**

Aim: Write a program to sort singly linked list

```
#include<iostream>
usingnamespace std;
classnode{
public:
int data;
node* next;
};
voidinsertionathead(node *&head,int data)
{
node *temp=newnode();
temp->data=data;
node *q=head;
head=temp;
temp->next=q;
return;
}
voiddisplay(node *head)
{
node *temp=head;
while(temp!=NULL)
{
cout<<temp->data<<"-->";
temp=temp->next;
}
cout<<endl;
}
```

```
voidmysort(node *&head,node *p1,node *p2)
{
if(p2==NULL)
{
return;
}
node *p3=p1->next;
while(p3!=NULL)
{
if(p1->data>p3->data)
{
swap(p1->data,p3->data);
}
p3=p3->next;
}
mysort(head,p2,p2->next);
}
intmain()
{
node *head1=NULL;
int n;
cin>>n;
int data;
for(inti=0;i<n;i++)
{
cin>>data;
insertionathead(head1,data);
}
cout<<"before sorting:"<<endl;
display(head1);
mysort(head1,head1,head1->next);
cout<<"after sorting:"<<endl;
display(head1);
}
```

Expected Output:

```
5
2 1 3 5 4
before sorting:
4-->5-->3-->1-->2-->
after sorting:
1-->2-->3-->4-->5-->
```

Viva – Questions:

1) Define OOPS?

OOPS stands for Object Oriented Programming System. It is also a program design technique, which is used to resolve problems of structured programming by binding data and function in a single unit called class. Here, data can be accessed only by the associated functions.

2) Define class?

Class is a logical encapsulation of data members and member functions in a single unit. It is a template of object. Class does not occupy any space in memory but when object creates, it occupies space in the memory according to data member. An empty class takes 1 byte space in memory.

For example, a HUMAN is a class and person "RAM" and "SHYAM" are the objects.

3) What is an Encapsulation?

It is a one of the basic feature of OOPS. Encapsulation means Binding data members and member functions in a single unit. Encapsulation can be useful to keep data safe from outside interfaces.

4) What is an Inheritance?

Inheritance is a mechanism (also an important component/feature of object oriented programming system) to inherit features from one to another class.

If we want to use/access existing features of any class, we can access them by using Inheritance. There will two classes Base class and Derived class.

If there is an existing class named "class_one" and new class named "class_two" that will access the features of class_one. In this case "class_one" will be considered as Base class and "class_two" as Derived class.

5) What is a polymorphism?

Polymorphism is the most important concept of OOPS. Polymorphism provides ability to use same name for different purposes.

In C++, functions and operators can be used to perform several (different) tasks by having same names. Two types of polymorphism are used:

Static or Compile time polymorphism
Function Overloading
Operator Overloading
Dynamic or Runtime polymorphism
Virtual function or dynamic binding

6) What is an Abstraction?

Abstraction means hiding the implementation detail for simplicity. It is a good programming practice to keep implementation and interface independent. Abstraction allows doing the same.

We do not need to change interface, if we are going to change the implementation

7) What is friend function?

A friend function is a function which is use to access the private data member of different class.

8) What is function overloading?

Function polymorphism for function overloading is a concept that allows multiple functions to share the same name with different arguments type assigning one or more function body to the same name is known as function overloading.

9) What are access specifiers in C++?

Access specifiers are the keywords that determines the availability of the data members/member functions of a class outside. The three access specifiers available in C++ are:

- Public: Data members and Member functions declared under public specification can be accessed from outside the class.
- Private: Data members and Member functions declared under private specification can be accessed only within the class and cannot be accessed from outside the class even by derived class objects.
- Protected: Data members and Member functions declared under protected specification can be accessed by its own class and the derived classes.

10) What is constructor?

A constructor is a special member function of a class that initializes the class object. The compiler provides a default constructor if the user has not provided a constructor. The constructor function name should be declared as the same name of Class. For example:

```
Class Bird
{
char name [50];
public:
Bird ()
{
printf("\nThis is Default Constructor of Class Bird\n")
}
};
```

There are basically three types of constructor available namely:

- Default Constructor
- Parameterized Constructor
- Copy Constructor

11) What is destructor?

A destructor is a special member function of a class that destroys the resources that have been allocated to the class object. The destructor class name is same as that of the class but prefixed with a ~ symbol.

12) Explain Virtual Destructor

A virtual destructor does the same function as that of a normal destructor but along with the destruction of derived class objects too. The virtual keyword must be employed before the function name as shown:
virtual ~Bird()

13) What is copy constructor?

A copy constructor is one of the types of constructor that is used for initializing a class object with the help of another object. It takes the same name as that of the class with one const class reference object as argument For example:

14) Can a constructor be overloaded?

Yes, but only through number of input arguments. Example – Default and parameterized constructors.

15) Can a destructor be overloaded?

No. A destructor must simply destroy all the resources allocated for the object.

16) Explain Function Overriding

Function Overriding is a run time polymorphism which allows derived class to provide its own implementation of the base class member functions. The newly implemented function in derived class is called Overridden Function.
Example:

```
class Base
{
public:
void display ()
{ cout<< "Base Class Display" <<endl; }
};
class Derived: public Base
{
public:
void display ()
{ cout<< "Derived Class Display" <<endl; }
};
```

17) What are Virtual Functions?

Virtual functions are base class member functions which helps to resolve function calls when there is redefinition for the same function provided by the derived class. Some of the important points to be remembered:

- ➢ Virtual functions of the base class can be redefined in the derived class.
- ➢ Provided virtual function in base class, when there is a base class pointer pointing to derived class object, the invoke of the function will call the derived class implementation.
- ➢ Virtual is just a keyword used to help C++ deiced at run-time which method has to be called depending on the type of the object pointed by base class pointer.

18) What is pure virtual function?

Pure virtual function is a type of virtual function which has only declaration and not definition. A pure virtual function declaration is assigned with zero. All deriving class must implement the pure virtual function.

virtual Display() = 0;

19) What is an Abstract Class?

A class is said to be abstract when it contains at least one pure virtual function. Instantiation is not allowed for an abstract class. And the deriving class must implement or provide definition for the pure virtual functions.

Example:

class Base { public: virtual void display () = 0; };

20) What are new and delete operator?

The new operator is used to dynamically allocate memory and delete operator is used to destroy the memory allocated by the new operator.

21) What is a Friend class?

Similar to friend function one class can be made as a friend to another class. Let’s say X and Y are separate classes. When X is made friend to class Y, the member functions of Class X becomes friend functions to Class Y. Member functions of Class X can access the private and protected data of Class Y but Class Y cannot access the same of ClassX.

```
class X;
class Y
{
// class X is a friend class of class Y
friend class X;
... .. ...
}
class X
{
... .. ...
}
```

22) Explain Call by Value?

Call by Value method passes only the values of the actual parameters to the formal parameters. So, two different copies are made. Any change made to the formal parameters will not affect the actual parameters.

Example:

```
#include <iostream>
using namespace std;
// function declaration
void Exchange(int A, int B) {
```

```
int temp;
temp = A;
A = B;
B = temp;
}
int main () {
int X = 100, Y = 200;
cout<< "Before Exchange, value of X :" << X <<
endl;
cout<< "Before Exchange, value of Y :" << Y <<
endl;
Exchange (X, Y);
cout<< "After Exchange, value of X :" << X <<
endl;
cout<< "After Exchange, value of Y :" << Y <<
endl;
return 0;
}
Expected Output:
Before Exchange, value of X :100
Before Exchange, value of Y :200
After Exchange, value of X :100
After Exchange, value of Y :200
```

23) Explain Call by Address?

In Call by Address, the address of the actual parameters are passed to the formal parameters which are pointer variables. Hence, any change made to the formal parameters are reflected in the actual parameters too.

Example:

```
 <iostream>
mespace std;
n declaration
hange(int *A, int *B) {

A;
;
p;

() {
```

```
00, Y = 200;
Before Exchange, value of X :" << X << endl;
Before Exchange, value of Y :" << Y << endl;
e (&X, &Y);
After Exchange, value of X :" << X << endl;
After Exchange, value of Y :" << Y << endl;



l Output:
xchange, value of X :100
xchange, value of Y :200
change, value of X :200
change, value of Y :100
```

24) Explain Call by Reference?

In Call by reference the formal parameters are reference variables and so both the actual and formal parameters point to the same memory locations. Hence, any changes made to the formal parameters will reflect in the actual parameters too.

Example:

```
 <iostream>
mespace std;
n declaration
hange(int &A, int &B) {

A;
;
p;

() {
00, Y = 200;
Before Exchange, value of X :" << X << endl;
Before Exchange, value of Y :" << Y << endl;
e (X, Y);
After Exchange, value of X :" << X << endl;
After Exchange, value of Y :" << Y << endl;
```

```
l Output:
xchange, value of X :100
xchange, value of Y :200
change, value of X :200
change, value of Y :100
```

25)

Expand STL in C++?

Standard template library.

26) What is algorithm?

Algorithm is a step by step procedure, which defines a set of instructions to be executed in certain order to get the desired output.

27) Why we need to do algorithm analysis?

A problem can be solved in more than one ways. So, many solution algorithms can be derived for a given problem. We analyze available algorithms to find and implement the best suitable algorithm.

28) What is asymptotic analysis of an algorithm?

Asymptotic analysis of an algorithm, refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.

29) What are asymptotic notations?

Asymptotic analysis can provide three levels of mathematical binding of execution time of an algorithm −

- Best case is represented by Ω(n) notation.
- Worst case is represented by O(n) notation.
- Average case is represented by Θ(n) notation.

30) What is a data structure?

Data structure is the way data is organized (stored) and manipulated for retrieval and access. It also defines the way different sets of data relate to one another, establishing relationships and forming algorithms.

31) What is a linear data structure? Name a few examples.

A data structure is linear if all its elements or data items are arranged in a sequence or a linear order. The elements are stored in a non-hierarchical way so that each item has successors and predecessors except the first and last element in the list.

Examples of linear data structures are Arrays, Stack, Strings, Queue, and Linked List.

32) What is a stack?

A stack is an abstract data type that specifies a linear data structure, as in a real physical stack or piles where you can only take the top item off the stack in order to remove things. Thus, insertion (push) and deletion (pop) of items take place only at one end called top of the stack, with a particular order: LIFO (Last In First Out) or FILO (First In Last Out).

33) Where are stacks used?

- Expression, evaluation, or conversion of evaluating prefix, postfix, and infix expressions

- ➢ Syntax parsing
- ➢ String reversal
- ➢ Parenthesis checking
- ➢ Backtracking

34) What is a queue data structure?

A queue is an abstract data type that specifies a linear data structure or an ordered list, using the First In First Out (FIFO) operation to access elements. Insert operations can be performed only at one end called REAR and delete operations can be performed only at the other end called FRONT.

35) List some applications of queue data structure.

To prioritize jobs as in the following scenarios:

- ➢ As waiting lists for a single shared resource in a printer, CPU, call center systems, or image uploads; where the first one entered is the first to be processed
- ➢ In the asynchronous transfer of data; or example pipes, file IO, and sockets
- ➢ As buffers in applications like MP3 media players and CD players
- ➢ To maintain the playlist in media players (to add or remove the songs)

36) What operations can be performed on queues?

- ➢ enqueue() adds an element to the end of the queue
- ➢ dequeue() removes an element from the front of the queue
- ➢ init() is used for initializing the queue
- ➢ isEmpty tests for whether or not the queue is empty
- ➢ Front is used to get the value of the first data item but does not remove it
- ➢ Rear is used to get the last item from a queue

37) What are the advantages of the heap over a stack?

Generally, both heap and stack are part of memory and used in Java for different needs:

- ➢ Heap is more flexible than the stack because memory space can be dynamically allocated and de-allocated as needed
- ➢ Heap memory is used to store objects in Java, whereas stack memory is used to store local variables and function call
- ➢ Objects created in the heap are visible to all threads, whereas variables stored in stacks are only visible to the owner as private memory
- ➢ When using recursion, the size of heap memory is more whereas it quickly fill-ups stack memory.

38) Define the graph data structure?

It is a type of non-linear data structure that consists of vertices or nodes connected by edges or arcs to enable storage or retrieval of data. Edges may be directed or undirected.

39) What are the applications of graph data structure?

- ➢ Transport grids where stations are represented as vertices and routes as the edges of the graph
- ➢ Utility graphs of power or water, where vertices are connection points and edges the wires or pipes connecting them
- ➢ Social network graphs to determine the flow of information and hotspots (edges and vertices)

➢ Neural networks where vertices represent neurons and edges the synapses between them

40) What is an AVL tree?

An AVL (Adelson, Velskii, and Landi) tree is a height balancing binary search tree in which the difference of heights of the left and right subtrees of any node is less than or equal to one. This controls the height of the binary search tree by not letting it get skewed. This is used when working with a large data set, with continual pruning through insertion and deletion of data.

41) Explain the max heap data structure.

It is a type of heap data structure where the value of the root node is greater than or equal to either of its child nodes.

42) How do you find the height of a node in a tree?

The height of the node equals the number of edges in the longest path to the leaf from the node, where the depth of a leaf node is 0.

43) Explain the Types of Data Structures?

There are mainly two types:

➢ **Linear Data Structure**: When all of its components are organized in a proper sequence, a data structure is called linear. The components are stored in linear data structures in a non-hierarchical manner where each item has the successors and predecessors except for the first and final element.

➢ **Non-linear data structure**: The Non-linear data structure does not form a series, i.e. each object or entity is linked to two or more than two objects in a non-linear manner. The elements of the data are not organized within the sequential model.

44) Discuss the Different Operations that can be Carried out on Data Structures?

Following are the different operation that are generally carried out in Data Structures:

Insert– Add a new data item in the already existing set of data items.

Delete– Remove an existing data item from the given data item set.

Traverse– Access each piece of data precisely once so that it can be processed.

Search– Figure out where the data item resides in the specified data item set.

Sort– Arrange the data objects in certain order i.e. in ascending or descending order for numerical data and in dictionary order for alphanumeric data.

45) Convert the expression ((A + B) * C - (D - E) ^ (F + G)) to equivalent Prefix and Postfix notations.

a. Prefix Notation: - * +ABC ^ - DE + FG

b. Postfix Notation: AB + C * DE - FG + ^ -

46) List out few of the Application of tree data-structure?

➢ The manipulation of Arithmetic expression,

➢ Symbol Table construction,

➢ Syntax analysi

47) Classify the Hashing Functions based on the various methods by which the key value is found.

➢ Direct method,

➢ Subtraction method,

➢ Modulo-Division method,

- Digit-Extraction method,
- Mid-Square method,
- Folding method,
- Pseudo-random method

48) What are the types of Collision Resolution Techniques and the methods used in each of the type?

- Open addressing (closed hashing), The methods used include: Overflow block.
- Closed addressing (open hashing), The methods used include: Linked list, Binary tree.

49) What is hashing?

Hashing is the process of converting a given key into another smaller value for O(1) retrieval time.

- This is done by taking the help of some function or algorithm which is called as hash function to map data to some encrypted or simplified representative value which is termed as "hash code" or "hash". This hash is then used as an index to narrow down search criteria to get data quickly.

50) **Time Complexity of Kruskal's**

In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogV), which is the overall Time Complexity of the algorithm.

51) **Time Complexity of Prim's Algorithm**

The time complexity of the Prim's Algorithm is O((V+E)logV) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.