B.E-CSE **2017-2018**

17BECS5E02 **Advanced Data Structures**

Instruction Hours/week: L: 3 T:0 P:0 **Marks:** Internal**:40** External**:60** Total**:100**

End Semester Exam:3 Hours

COURSE OBJECTIVES:

- To impart the basic concepts of data structures and algorithms.
- To understand concepts about searching and sorting techniques
- To understand basic concepts about stacks, queues, lists, trees and graphs.
- To enable them to write algorithms for solving problems with the help of fundamental data structures

COURSE OUTCOMES:

- For a given algorithm student will able to analyze the algorithms to determine the time and computation complexity and justify the correctness.
- For a given Search problem (Linear Search and Binary Search) student will able to implement it.
- For a given problem of Stacks, Queues and linked list student will able to implement it and analyze the same to determine the time and computation complexity.Student will able to write an algorithm Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort and compare their performance in term of Space and Time complexity.
- Student will able to implement Graph search and traversal algorithms and determine the time and computation complexity.

UNIT 1:

Introduction: Basic Terminologies: Elementary Data Organizations, Data Structure Operations: insertion, deletion, traversal etc.; Analysis of an Algorithm, Asymptotic Notations, Time-Space trade off. **Searching:** Linear Search and Binary Search Techniques and their complexity analysis.

UNIT 2:

Stacks and Queues: ADT Stack and its operations: Algorithms and their complexity analysis, Applications of Stacks: Expression Conversion and evaluation – corresponding algorithms and complexity analysis. ADT queue, Types of Queue: Simple Queue, Circular Queue, Priority Queue; Operations on each types of Queues: Algorithms and their analysis.

UNIT 3:

Linked Lists: Singly linked lists: Representation in memory, Algorithms of several operations: Traversing, Searching, Insertion into, Deletion from linked list; Linked representation of Stack and Queue, Header nodes, Doubly linked list:

operations on it and algorithmic analysis; Circular Linked Lists: all operations their algorithms and the complexity analysis.

Trees: Basic Tree Terminologies, Different types of Trees: Binary Tree, Threaded Binary Tree, Binary Search Tree, AVL Tree; Tree operations on each of the trees and their algorithms with complexity analysis. Applications of Binary Trees. B Tree, B+ Tree: definitions, algorithms and analysis.

UNIT 4:

Sorting and Hashing: Objective and properties of different sorting algorithms:

Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort; Performance and Comparison among all the methods, Hashing.

UNIT 5:

Graph: Basic Terminologies and Representations, Graph search and traversal algorithms and complexity analysis.

TEXT BOOKS:

1. “Fundamentals of Data Structures”, Illustrated Edition by Ellis Horowitz, Sartaj Sahni, Computer Science Press.

REFERENCES:

1. Algorithms, Data Structures, and Problem Solving with C++”, Illustrated Edition by Mark Allen Weiss, Addison-Wesley Publishing Company

2. “Howto Solve it by Computer”, 2nd Impression by R. G. Dromey, Pearson Education.

KARPAGAM ACADEMY OF HIGHER EDUCATION

Faculty of Engineering

Department of Computer Science and Engineering

Lecture Plan

Subject Name: **ADVANCED DATA STRUCTURES**

Subject Code: **17BECS5E02**

S.No	Lecturer (Hr)	Topics to be covered	
UNIT-I (Introduction)			

1	1	Fundamentals	
2	1	Overview of C++, Structures	T1 (33), T1(37)
3	1	Class Scope and Accessing Class Members, Reference Variables Initialization	T1(261,484),T1(424)
4	1	Constructors, Destructors	T1(88),T1(496)
5	1	Member Functions and Classes	T1(71-73)
6	1	Friend Function	T1(536)
7	1	Dynamic Memory Allocation	T1(545)
8	1	Static Class Members	T1(547)
9	1	Container Classes and Integrators, Proxy Classes	T1(556), T1(557)
10	1	Overloading: Function overloading and Operator Overloading.	R1(119),T1(568)
11	1	University question paper discussion	
UNIT-2 (Stacks and Queues)			
12	1	Fundamentals	
13	1	Base Classes and Derived Classes, Protected Members	T1(630), T1(633)
14	1	Casting Class pointers and Member Functions, Overriding	R4(408-414)
15	1	Public, Protected and Private Inheritance, Constructors and Destructors in derived Classes	T1(673, 665)
16	1	Implicit Derived, Class Object To Base	R4(415)
17	1	Class Object Conversion, Composition Vs. Inheritance	T1(702)
18	1	Virtual functions	T1(696)
19	1	This Pointer, Abstract Base Classes and Concrete Classes	R1(429),T1(703)
20	1	Virtual Destructors	T1(730)
21	1	Dynamic Binding.	T1(723)
22	1	University question paper discussion	
UNIT-3 (Linked Lists)			
23	1	Fundamentals	
24	1	Abstract Data Types (ADTs)	T2(57&58)
25	1	List ADT	T2(58)

26	1	Array-based implementation	T2(59)
27	1	linked list implementation	T2(59 - 67)
28	1	singly linked lists	T2(60)
29	1	singly linked lists	T2(60-68)
30	1	Polynomial Manipulation	T2(68 - 70)
31	1	Stack ADT	T2(78 -86)
32	1	Queue ADT	T2(95 -100)
33	1	Evaluating arithmetic expressions	T2(87-93)
34	1	University question paper discussion-2 marks	
35	1	University question paper discussion- 16 marks	
UNIT-4 (Sorting and Hashing)			
36	1	Fundamentals	
37	1	Sorting algorithms: Insertion sort	T2(105 to 111)
38	1	Quick sort-ANALYSIS	T2(111 to 113)
39	1	Quick sort- ALGORITHM	T2(114 to 116)
40	1	Merge sort –ANALYSIS	T2(107)
41	1	Merge sort- ALGORITHM	T2(299)
42	1	Searching: Linear search	T2(300)
43	1	Binary Search	T2(307-311)
44	1	Binary Search	T2(335-346)
UNIT-5 (Graph)			
45	1	Fundamentals	
46	1	Trees	T2(236 -237)
47	1	Binary Trees	T2(251 - 261)
48	1	Binary tree representation and traversals	T2(251 - 262)
49	1	Application of trees: Set representation and Union-Find operations	T2(246 - 250)
50	1	Graph and its representations ,Graph Traversals	T2(246 - 250)

51	1	Representation of Graphs	T2(59 -64)
52	1	Breadth-first search	T2(117 -119)
53	1	Depth-first search	T2(117 -119)
54	1	Connected components	T2(119 -121)
55	1	University question paper discussion	
		TOTAL	55

TEXT BOOKS:

1. “Fundamentals of Data Structures”, Illustrated Edition by Ellis Horowitz, Sartaj
Sahni, Computer Science Press.

REFERENCES:

1. Algorithms, Data Structures, and Problem Solving with C++”, Illustrated Edition by Mark Allen Weiss, Addison-Wesley Publishing Company

2. “Howto Solve it by Computer”, 2nd Impression by R. G. Dromey, Pearson Education.

LECTURE NOTES

1.1 OVERVIEW OF C++

C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.

C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.

C++ was developed by **Bjarne Stroustrup starting in 1979** at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.

C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

STRUCTURE VS CLASS IN C++

In C++, a structure is same as class except the following differences:

1) Members of a class are private by default and members of struct are public by default. For example program 1 fails in compilation and program 2 works fine.

```
// Program 1
#include <stdio.h>
 class Test
  {
    int x;             // x is private
    };
int main()
{
  Test t;
  t . x = 20; // compiler error because x is private
  getchar();
  return 0;
}
// Program 2
#include <stdio.h>
 struct Test {
    int x; // x is public
};
int main()
{
  Test t;
  t  . x = 20; // works fine because x is public
  getchar();
  return 0;
}
```

OOPs – Object Oriented Programming System

Object-oriented programming (OOP) is a programming paradigm that uses "Objects "and their interactions to design applications and computer programs.

Features of OOPS

1. **Object**
2. **Class**
3. **Data Abstraction & Encapsulation**
4. **Inheritance**
5. **Polymorphism**
6. **Dynamic Binding**
7. **Message Passing**

1) **Object :**

- **Object** is the basic unit of object-oriented programming.
- Objects are identified by its unique name.
- An object represents a particular instance of a class.
- There can be more than one instance of an object.
- An Object is a collection of data members and associated member functions also known as methods.

  For example :

The class of Dog defines all possible dogs by listing the characteristics and behaviors they can have; the object Lassie is one particular dog, with particular versions of the characteristics. A Dog has fur; Lassie has brown-and-white fur.

2) Class :

Classes are data types based on which objects are created.

- Objects with similar properties and methods are grouped together to form a Class.
- Characteristics of an object are represented in a class as Properties.
- The actions that can be performed by objects become functions of the class and is referred to as Methods.

  For example:

- consider we have a Class of Cars under which Santro Xing, Alto and WaganR represents individual Objects. In this context each Car Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc

- Properties of the Car class and the associated actions i.e., object functions like Start, Move, Stop form the Methods of Car Class.
- Memory is allocated only when an object is created, i.e., when an instance of a class is created.

3) Data abstraction & Encapsulation : The wrapping up of data and its functions into a single unit is called Encapsulation.

Data Abstraction refers to the act of representing essential features without including the background details or explanation between them.

For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

4) Inheritance :

Inheritance is the process of forming a new class from an existing class or base class.

The base class is also known as parent class or super class.

Derived class is also known as a child class or sub class. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

It is classified into different types, they are

Single level inheritance

Multi-level inheritance

Hybrid inheritance

Hierarchical inheritance

5) Polymorphism :

Poly a Greek term ability to take more than one form. Overloading is one type of

Polymorphism. It allows an object to have different meanings, depending on its context. When an exiting operator or function begins to operate on new data type, or class, it is understood to be overloaded.

6) Dynamic binding :

It refers to linking a procedure call to the code that will be executed only at run time. The code associated with the procedure in not known until the program is executed, which is also known as late binding.

7) Message Passing : It refers to that establishing communication between one place to another.

I/O Library Header Files:

There are following header files important to C++ programs:

Header File	Function and Description
<iostream>	This file defines the **cin, cout, cerr** and **clog** objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, espectively.
<iomanip>	This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as **setw** and**setprecision**.
<fstream>	This file declares services for user-controlled file processing. We will discuss about it in detail in File and Stream related chapter.

The standard output stream (cout):

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream>
 using namespace std;
 int main( )
{   char str[] = "Hello C++";
    cout << "Value of str is : " << str << endl;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of str is : Hello C++
```

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

The standard input stream (cin):

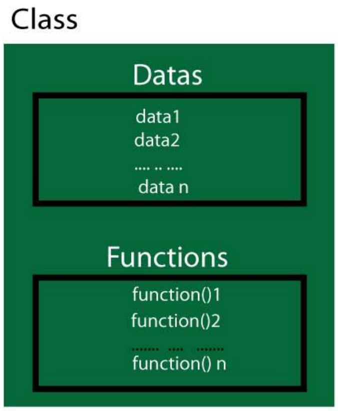
The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>
 using namespace std;
 int main( )
{   char name[50];
    cout << "Please enter your name: ";
   cin >> name;
   cout << "Your name is: " << name << endl;
}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the result something as follows: Please enter your name: cplusplus Your name is: cplusplus

1.2 C++ Class

A class is the collection of related data and function under a single name. A C++ program can have any number of classes. When related data and functions are kept under a class, it helps to visualize the complex problem efficiently and effectively.



A Class is a blueprint for objects

When a class is defined, no memory is allocated. You can imagine like a datatype.

```
int var;
```

The above code specifies *var* is a variable of type integer; int is used for specifying variable *var* is of integer type. Similarly, class are also just the specification for objects and object bears the property of that class.

Defining the Class in C++

Class is defined in C++ programming using keyword class followed by identifier(name of class). Body of class is defined inside curly brackets an terminated by semicolon at the end in similar way as structure.

```
class class_name

  {   // some data

  // some functions   };
```

Example of Class in C++

```
class temp

  {

     private:

       int data1;

       float data2;
```

```
    public:

      void func1()

        {   data1=2;  }

      float func2(){

          data2=3.5;

          retrun data;

        }

  };
```

Access specifiers defines the access rights for the statements or functions that follows it until another access specifier or till the end of a class. The three types of access specifiers are "private", "public", "protected".

private:

The members declared as "private" can be accessed only within the same class and not from outside the class.

public:

The members declared as "public" are accessible within the class as well as from outside the class.

protected:

The members declared as "protected" cannot be accessed from outside the class, but can be accessed from a derived class. This is used when inheritaance is applied to the members of a class.
The members declared as "public" are accessible within the class as well as from outside the class.

Object Declaration

Once a class is defined, you can declare objects of that type. The syntax for declaring a object is the same as that for declaring any other variable. The following statements declare two objects of type circle:

Circle c1, c2;

Accessing Class Members

Once an object of a class is declared, it can access the public members of the class.

```
c1.setRadius(2.5);
```

```
// Header Files
#include <iostream>
#include<conio.h>
using namespace std;

// Class Declaration
class person
{
//Access - Specifier
public:

//Varibale Declaration
  string name;
  int number;
};

//Main Function
int main()
{
    // Object Creation For Class
    person obj;

    //Get Input Values For Object Varibales
    cout<<"Enter the Name :";
    cin>>obj.name;

    cout<<"Enter the Number :";
    cin>>obj.number;

    //Show the Output
    cout << obj.name << ": " << obj.number << endl;

    getch();
     return 0;
}
```

SCOPE RESOLUTION OPERATOR

:: is the scope resolution operator. When local variable and global variable are having same name, local variable gets the priority. C++ allows flexibility of accessing both the variables through a scope resolution operator.

To access the global version of the variable, C++ provides scope resolution operator. It is used when a member function is defined outside the class

e.g.

```
Class MyClass
{
int n1, n2;
public:
{
void func1();                ---------Function Declaration
}
};

public void MyClass::func1() ---Use of Scope Resolution Operator to write
                              function definition outside class definition
{
        // Function Code
}
```

1.3 CONSTRUCTOR AND DESTRUCTOR

Constructor

It is a member function having same name as it's class and which is used to initialize the objects of that class type with a legal initial value. Constructor is automatically called when object is created.

Types of Constructor

Default Constructor-: A constructor that accepts no parameters is known as default constructor.

If no constructor is defined then the compiler supplies a default constructor.

```
Circle :: Circle()
{
  radius = 0;
}
```

Parameterized Constructor -: A constructor that receives arguments/parameters, is called parameterized constructor.

```
Circle :: Circle(double r)
```

```
{
   radius = r;
}
```

Copy Constructor-: A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

```
Circle :: Circle(Circle &t)
{
   radius = t.radius;
}
```

Destructor

A destructor is a member function having sane name as that of its class preceded by ~(tilde) sign and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object's scope is over.

```
~Circle() {}
```

Example 1 : In the following program constructors, destructor and other member functions are defined inside class definitions. Since we are using multiple constructor in class so this example also illustrates the concept of constructor overloading

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

class Circle //specify a class
{
   private :
      double radius; //class data members
   public:
      Circle() //default constructor
      {
         radius = 0;
      }
      Circle(double r) //parameterized constructor
      {
         radius = r;
      }
      Circle(Circle &t) //copy constructor
      {
         radius = t.radius;
      }
      void setRadius(double r) //function to set data
      {
         radius = r;
      }
      double getArea()
```

```
        {
           return 3.14 * radius * radius;
        }
        ~Circle() //destructor
        {}
};

int main()
{
   Circle c1; //defalut constructor invoked
   Circle c2(2.5); //parmeterized constructor invoked
   Circle c3(c2); //copy constructor invoked
   cout << c1.getArea()<<endl;
   cout << c2.getArea()<<endl;
   cout << c3.getArea()<<endl;
   return 0;
}
```

Example 2:

```
#include <iostream>
using namespace std;
class Distance                              // Be careful using Distance D is capital .
{
private:
            int feet;
            float inches;
public:
            Distance (){ feet=0,inches=0.0;} // no argument constructor
            {}
            Distance (int f,float i){feet=f,inches=i;}    //two argument constructor
            {}
            void get()
            {
                        cout<<"enter feet";
                        cin>>feet;
                        cout<<"enter inches";
                        cin>>inches;          }
            void print()
            {
                        cout<<"feet is"<<feet;
                        cout<<"inches is "<<inches;
            }
            void add_dist(Distance,Distance);
};
void Distance ::add_dist(Distance d2,Distance d3)
{
            inches = d2.inches + d3.inches;
```

```
        feet=0;
        if(inches>=12)
        {
                inches -=12;
                feet++;
        }
        feet += d2.feet + d3.feet;
}
void main ()
{
        Distance d1,d3;
        Distance d2(12,6.43);
        d1.get();
        d3.add_dist(d1 , d2);   // d3=d1+d2
        d1.print();
        d2.print();
        d3.print();
}
```

1.4 MEMBER FUNCTIONS IN CLASSES

Member functions are the functions, which have their declaration inside the class definition and works on the data members of the class. The definition of member functions can be inside or outside the definition of class.

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution :: operator along with class name along with function name.

Example :

```
class Cube

{

 public:

 int side;

 int getVolume();     // Declaring function getVolume with no argument and return type int.

};
```

If we define the function inside class then we don't not need to declare it first, we can directly define the function.

```
class Cube
```

```
{
 public:
 int side;
 int getVolume()
 {
  return side*side*side;       //returns volume of cube
 }
};
```

But if we plan to define the member function outside the class definition then we must declare the function inside class definition and then define it outside.

```
class Cube
{
 public:
 int side;
 int getVolume();
}

int Cube :: getVolume()     // defined outside class definition
{
 return side*side*side;
}
```

1.5 FUNCTIONS

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is knows as with various names like a method or a **sub-routine** or a **procedure** etc.

Defining a Function:

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
  body of the function }
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body:** The function body contains a collection of statements that define what the function does.

Example:

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers
 int max(int num1, int num2)
{
  // local variable declaration
  int result;
   if (num1 > num2)
```

```
     result = num1;
   else
     result = um2;
    return result;
}
```

Function Declarations:

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function:

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <iostream>
using namespace std;
 // function declaration
int max(int num1, int num2);
 int main ()
{   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;
    // calling a function to get max value.
   ret = max(a, b);
    cout << "Max value is : " << ret << endl;
    return 0;
}
// function returning the max between two numbers
int max(int num1, int num2)
{
   // local variable declaration
   int result;
    if (num1 > num2)
      result = num1;
   else
      result = num2;    return result; }
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
Call by reference	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters:

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example:

```
#include <iostream>
```

```
using namespace std;
 int sum(int a, int b=20)
{
  int result;
  result = a + b;
   return (result);
}
int main ()
{
  // local variable declaration:
  int a = 100;
  int b = 200;
  int result;
   // calling a function to add the values.
  result = sum(a, b);
  cout << "Total value is :" << result << endl;
  // calling a function again as follows.
  result = sum(a);
  cout << "Total value is :" << result << endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total value is :300
Total value is :120
```

INLINE FUNCTIONS

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example, which makes use of inline function to return max of two numbers:

```
#include <iostream>
 using namespace std;
inline int Max(int x, int y)
{
  return (x > y)? x : y;
}
// Main function for the program
int main( )
{
  cout << "Max (20,10): " << Max(20,10) << endl;
  cout << "Max (0,200): " << Max(0,200) << endl;
  cout << "Max (100,1010): " << Max(100,1010) << endl;
  return 0;
}
```

FRIEND FUNCTION IN C++

If a function is defined as a friend function then, the private and protected data of class can be accessed from that function. The complier knows a given function is a friend function by its keyword **friend**. The declaration of friend function should be made inside the body of class (can be anywhere inside class either in private or public section) starting with keyword friend

```
#include <iostream>
using namespace std;
class Distance
{
    private:
        int meter;
    public:
        Distance(){ meter=0;}
        friend int func(Distance);  //friend function
};
int func(Distance d)            //function definition
{
    d.meter=5;         //accessing private data from non-member function
    return d.meter;
}
int main()
{
    Distance D;
    cout<<"Distace: "<<func(D);
    return 0;
}
```

Example to operate on Objects of two Different class using friend Function

```

#include <iostream>
using namespace std;
class B;    // forward declaration
class A {
    private:
      int data;
    public:
      A(): data(12){ }
      friend int func(A , B);   //friend function Declaration
};
class B {
    private:
       int data;
    public:
       B(): data(1){ }
       friend int func(A , B);  //friend function Declaration
};
int func(A d1,B d2)
```

```
/*Function func() is the friend function of both classes A and B. So, the private data
of both class can be accessed from this function.*/
{
  return (d1.data+d2.data);
}
int main()
   {
      A a;
      B b;
      cout<<"Data: "<<func(a,b);
      return 0;
   }
```

Friend Class:

```
#include<iostream.h>
 #include<conio.h>
 class readint
 {
 float a,b;
 public:
 void read()
 {
 cout<<"\n\nEnter the First Number : ";
 cin>>a;
 cout<<"\n\nEnter the Second Number : ";
 cin>>b;
 }
 friend class sum;
 };
 class sum
 {
 public:
 float c;
 void add(readint rd)
 {
 c=rd.a+rd.b;
 cout<<"\n\nSum="<<c;
 }
 };
 void main()
 {
 int cont;
 readint rd;
 sum s;
 clrscr();
 do
 {
 clrscr();
 rd.read();
 s.add(rd);
 cout<<"\n\nDo you want to continue?(1-YES,0-NO)";
 cin>>cont;
 }while(cont==1);
 getch();
 }
```

1.7 Static Keyword

Static is a keyword in C++ used to give special characteristics to an element. Static elements are allocated storage only once in a program lifetime in static storage area. And they have a scope till the program lifetime. Static Keyword can be used with following,

1. Static variable in functions
2. Static Class Objects
3. Static member Variable in class
4. Static Methods in class

Static variables inside Functions

Static variables when used inside function are initialized only once, and then they hold there value even through function calls.

These static variables are stored on static storage area , not in stack.

```
void counter()
{
 static int count=0;
 cout << count++;
}
int main()
```

```
{
 for(int i=0;i<5;i++)
 {
   counter();
 }
}
```

Output : 0 1 2 3 4

Let's se the same program's output **without using static** variable.

```
void counter()
{
 int count=0;
 cout << count++;
}
int main(0
{
 for(int i=0;i<5;i++)
 {
   counter();
 }
}
```

Output : 0 0 0 0 0

If we do not use static keyword, the variable count, is reinitialized everytime when counter() function is called, and gets destroyed each time when counter() functions ends. But, if we make it static, once initialized count will have a scope till the end of main() function and it will carry its value through function calls too.

If you don't initialize a static variable, they are by default initialized to zero.

Static class Objects

Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.

Static objects are also initialized usig constructors like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

```
class Abc
{
 int i;
 public:
 Abc()
 {
  i=0;
  cout << "constructor";
 }
 ~Abc()
 {
   cout << "destructor";
 }
};
void f()
{
 static Abc obj;
}
int main()
{
 int x=0;
 if(x==0)
 {
  f();
 }
 cout << "END";
}
```

```
Output : constructor END destructor
```

You must be thinking, why was destructor not called upon the end of the scope of if condition. This is because object was static, which has scope till the program lifetime, hence destructor for this object was called when main() exits.

Static data member in class

Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members.

Static member variables (data members) are not initialied using constructor, because these are not dependent on object initialization.

Also, it must be initialized explicitly, always outside the class. If not initialized, Linker will give error.

```
class X
{
 static int i;
 public:
 X(){};
};
int X::i=1;
int main()
{
 X obj;
 cout << obj.i;   // prints value of i
}
```

Once the definition for static data member is made, user cannot redefine it. Though, arithmetic operations can be performed on it.

Static Member Functions

These functions work for the class as whole rather than for a particular object of a class.

It can be called using an object and the direct member access `.` operator. But, its more typical to call a static member function by itself, using class name and scope resolution `::` operator.

Example :

```
class X
{
 public:
 static void f(){};
};
int main()
{
 X::f();   // calling member function directly with class name
}
```

These functions cannot access ordinary data members and member functions, but only static data members and static member functions.

1.8 STL Containers

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provides general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

At the core of the C++ Standard Template Library are following three well-structured components:

Component	**Description**
Containers	Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
Algorithms	Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.

Iterators	Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

Sequence containers

Sequence containers maintain the ordering of inserted elements that you specify.

- A **vector** container behaves like an array, but can automatically grow as required. It is random access and contiguously stored, and length is highly flexible. For these reasons and more, **vector** is the preferred sequence container for most applications. For more information, see vector Class.
- An **array** container has some of the strengths of **vector**, but the length is not as flexible. For more information, see array Class (STL).
- A **deque** (double-ended queue) container allows for fast insertions and deletions at the beginning and end of the container. It shares the random-access and flexible-length advantages of **vector**, but is not contiguous. For more information, see deque Class.
- A **list** container is a doubly linked list that enables bidirectional access, fast insertions, and fast deletions anywhere in the container, but you cannot randomly access an element in the container.

Associative Containers

In associative containers, elements are inserted in a pre-defined order—for example, as sorted ascending. The associative containers can be grouped into two subsets: maps and sets.

- A **map**, sometimes referred to as a dictionary, consists of a key/value pair. The key is used to order the sequence, and the value is associated with that key. For example, a **map**might contain keys that represent every unique word in a text and corresponding values that represent the number of times that each word appears in the text. The unordered version of **map** is **unordered_map**..
- A **set** is just an ascending container of unique elements—the value is also the key. The unordered version of **set** is **unordered_set**. For more information, see set Class andunordered_set Class.

Both **map** and **set** only allow one instance of a key or element to be inserted into the container. If multiple instances of elements are required, use **multimap** or **multiset**. The unordered versions are **unordered_multimap** and **unordered_multiset.**

Container Adapters

A container adapter is a variation of a sequence or associative container that restricts the interface for simplicity and clarity. Container adapters do not support iterators.

- A **queue** container follows FIFO (first in, first out) semantics. The first element *pushed*—that is, inserted into the queue—is the first to be *popped*—that is, removed from the queue..
- A **priority_queue** container is organized such that the element that has the highest value is always first in the queue. A **stack** container follows LIFO (last in, first out) semantics. The last element pushed on the stack is the first element popped

Accessing Container Elements: Iterators

The elements of containers are accessed by using iterators.

The STL facilities make widespread use of iterators to mediate among the various algorithms and the sequences upon which they act. The name of an iterator type (or its prefix) indicates the category of iterators required for that type. In order of increasing power, the categories are summarized here as:

- **Output.** An output iterator *X* can only have a value *V* stored indirect on it, after which it *must* be incremented before the next store, as in (**X*++ = *V*), (**X* = *V*, ++*X*), or (**X* =*V*, *X*++).
- **Inpu**t. An input iterator *X* can represent a singular value that indicates end of sequence. If an input iterator does not compare equal to its end-of-sequence value, it can have a value *V* accessed indirect on it any number of times, as in (*V* = **X*). To progress to the next value or end of sequence, you increment it, as in ++*X*, *X*++, or (*V* = **X*++). Once you increment any copy of an input iterator, none of the other copies can safely be compared, dereferenced, or incremented thereafter.
- **Forward.** A forward iterator *X* can take the place of an output iterator for writing or an input iterator for reading. You can, however, read (through *V* = **X*) what you just wrote (through **X* = *V*) through a forward iterator. You can also make multiple copies of a forward iterator, each of which can be dereferenced and incremented independently.
- **Bidirectional.** A bidirectional iterator *X* can take the place of a forward iterator. You can, however, also decrement a bidirectional iterator, as in --*X*, *X*--, or (*V* = **X*--).

- **Random access.** A random-access iterator *X* can take the place of a bidirectional iterator. You can also perform much the same integer arithmetic on a random-access iterator that you can on an object pointer. For *N*, an integer object, you can write *x*[*N*], *x* + *N*, *x* - *N*, and *N* + *X*.

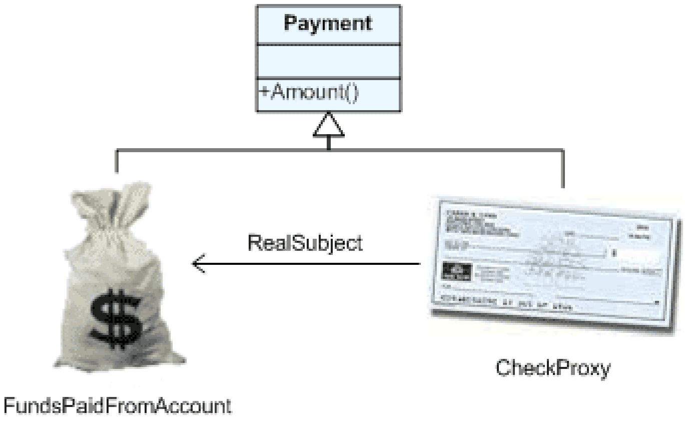
1.9 C++ example of Proxy Design Pattern

The Proxy pattern is used when you need to represent a complex object by a simpler one. If creating an object is expensive in time or computer resources, Proxy allows you to postpone this creation until you need the actual object. A Proxy usually has the same methods as the object it represents, and once the object is loaded, it passes on the method calls from the Proxy to the actual object.

There are several cases where a Proxy can be useful:

1. If an object, such as a large image, takes a long time to load.
2. If the object is on a remote machine and loading it over the network may be slow, especially during peak network load periods.
3. If the object has limited access rights, the proxy can validate the access permissions for that user.

Proxies can also be used to distinguish between requesting an instance of an object and the actual need to access it. For example, program initialization may set up a number of objects which may not all be used right away. In that case, the proxy can load the real object only when it is needed.



```
#include<iostream>
#include<string>
using namespace std;
// The 'Subject interface
class IMath
{
public:
  virtual double Add(double x, double y) = 0;
  virtual double Sub(double x, double y) = 0;
  virtual double Mul(double x, double y) = 0;
  virtual double Div(double x, double y) = 0;
};
// The 'RealSubject' class
class Math : public IMath
{
public:
  double Add(double x, double y)
  {
    return x + y;
  }
  double Sub(double x, double y)
  {
    return x - y;
  }
  double Mul(double x, double y)
  {
    return x * y;
  }
  double Div(double x, double y)
  {
    return x / y;
  }
};
// The 'Proxy Object' class
class MathProxy : public IMath
{
public:
  MathProxy()
  {
    math_ = NULL;
  }
  virtual ~MathProxy()
  {
    if(math_)
      delete math_;
  }
  double Add(double x, double y)
  {
    return getMathInstance()->Add(x, y);
  }
  double Sub(double x, double y)
  {
    return getMathInstance()->Sub(x, y);
  }
  double Mul(double x, double y)
  {
   return getMathInstance()->Mul(x, y);
  }
  double Div(double x, double y)
  {
   return getMathInstance()->Div(x, y);
  }
private:
  Math* math_;
  Math* getMathInstance(void)
  {
   if(!math_)
     math_ = new Math();
   return math_;
  }
};

//The Main method
int main()
{
  // Create math proxy
  MathProxy proxy;

  //Do the math
  cout<<"4 + 2 = "<<proxy.Add(4, 2)<<endl;
  cout<<"4 - 2 = "<<proxy.Sub(4, 2)<<endl;
  cout<<"4 * 2 = "<<proxy.Mul(4, 2)<<endl;
  cout<<"4 / 2 = "<<proxy.Div(4, 2)<<endl;

  return 0;
}
```

1.10 FUNCTION OVERLOADING IN C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types:

```
#include <iostream>

using namespace std;
```

```
 class printData
{
  public:
     void print(int i) {
       cout << "Printing int: " << i << endl;
     }
     void print(double  f) {
       cout << "Printing float: " << f << endl;
     }
     void print(char* c) {
       cout << "Printing character: " << c << endl;
     }
};
int main(void)
{
  printData pd;
   // Call print to print integer
  pd.print(5);
  // Call print to print float
  pd.print(500.263);
  // Call print to print character
  pd.print("Hello C++");
   return 0;
}
```

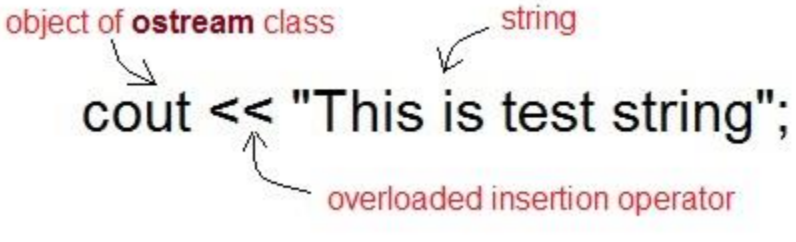
Example 2:

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

void area(int r)
{
 cout<<endl<<"Area of circle="<<3.142*r*r;
}
```

```
void area(int b,int h)
{
 cout<<endl<<"Area of triangle="<<0.5*b*h;
}

void area(float l,float b)
{
 cout<<endl<<"Area of rectangle="<<l*b;
}

int main()
{
 int r,b,h;
 float l,br;
 clrscr();
 cout<<endl<<"Enter radius of circle: ";
 cin>>r;
 area(r);
 cout<<endl<<"Enter base and height of triangle: ";
 cin>>b>>h;
 area(b,h);
 cout<<endl<<"Enter length and breadth of rectangle: ";
 cin>>l>>br;
 area(l,br);
```

1.11 OPERATOR OVERLOADING

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.

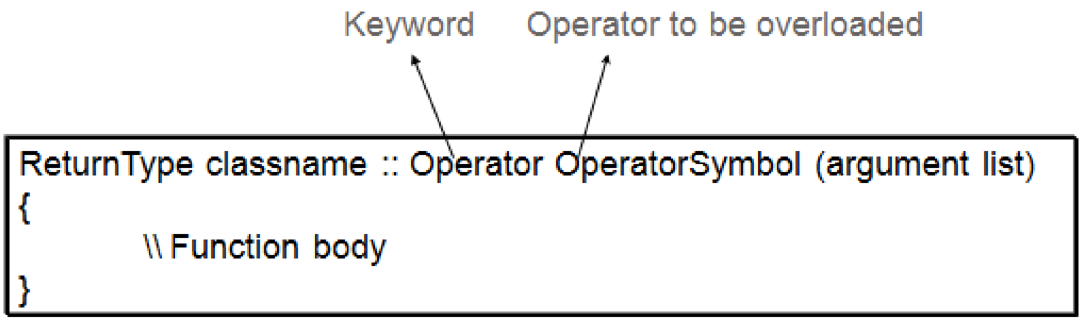


```
cout << "This is test string";
```

Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded.**Operator that are not overloaded** are follows

- scope operator - ::
- sizeof
- member selector - .
- member pointer selector - *
- ternary operator - ?:

Operator Overloading Syntax



```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
        \\ Function body
}
```

Implementing Operator Overloading

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.

Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Restrictions on Operator Overloading

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.

3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Overloading Arithmetic Operator

Arithmetic operator are most commonly used operator in C++. Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type. In the below example we have overridden the **+** operator, to add to Time(hh:mm:ss) objects.

Example: overloading '+' Operator to add two time object

```
#include< iostream.h>

#include< conio.h>

class time

{

 int h,m,s;

 public:

  time()

  {

   h=0, m=0; s=0;

  }

  void getTime();

  void show()

  {

   cout<< h<< ":"<< m<< ":"<< s;

  }

 time operator+(time);   //overloading '+' operator

};

time time::operator+(time t1)            //operator function

{

 time t;

 int a,b;

 a=s+t1.s;
 t.s=a%60;
 b=(a/60)+m+t1.m;
 t.m=b%60;
 t.h=(b/60)+h+t1.h;
 t.h=t.h%12;
 return t;
}
void time::getTime()
{
  cout<<"\n Enter the hour(0-11) ";
  cin>>h;
  cout<<"\n Enter the minute(0-59) ";
  cin>>m;
  cout<<"\n Enter the second(0-59) ";
  cin>>s;
}
void main()
{
 clrscr();
 time t1,t2,t3;
 cout<<"\n Enter the first time ";
 t1.getTime();
 cout<<"\n Enter the second time ";
 t2.getTime();
 t3=t1+t2;              //adding of two time object using '+' operator
 cout<<"\n First time ";
 t1.show();
 cout<<"\n Second time ";
 t2.show();
 cout<<"\n Sum of times ";
 t3.show();
 getch();
}
```

Overloading I/O operator

- Overloaded to perform input/output for user defined datatypes.
- Left Operand will be of types `ostream&` and `istream&`
- Function overloading this operator must be a Non-Member function because left operand is not an Object of the class.
- It must be a friend function to access private data members.

You have seen above that **<<** operator is overloaded with **ostream** class object `cout` to print primitive type value output to the screen. Similarly you can overload **<<** operator in your class to print user-defined type to screen. For example we will overload **<<** in **time** class to display time object using `cout`.

```
time t1(3,15,48);
cout << t1;
```

NOTE: When the operator does not modify its operands, the best way to overload the operator is via friend function.

Example: overloading '<<' Operator to print time object

```
#include< iostream.h>
#include< conio.h>
class time
{
 int hr,min,sec;
 public:
  time()
 {
  hr=0, min=0; sec=0;
 }
   time(int h,int m, int s)
 {
  hr=h, min=m; sec=s;
 }
 friend ostream& operator << (ostream &out, time &tm);  //overloading '<<' operator
};

ostream& operator<< (ostream &out, time &tm)              //operator function
{
 out << "Time is " << tm.hr << "hour : " << tm.min << "min : " << tm.sec << "sec";
 return out;
}

void main()
{
 time tm(3,15,45);
 cout << tm;
}
```

Output

```
Time is 3 hour : 15 min : 45 sec
```

Overloading Relational operator

You can also overload Relational operator like `==` , `!=` , `>=` , `<=` etc. to compare two user-defined object.

Example

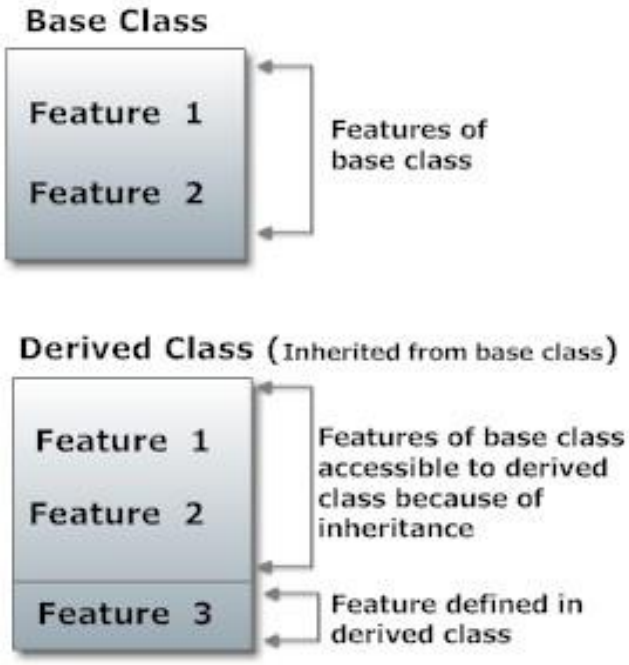
```
class time
```

```
{
int hr,min,sec;
 public:
  time()
  {
   hr=0, min=0; sec=0;
  }

  time(int h,int m, int s)
  {
   hr=h, min=m; sec=s;
  }
 friend bool operator==(time &t1, time &t2);   //overloading '==' operator
};
bool operator== (time &t1, time &t2)                    //operator function
{
return ( t1.hr == t2.hr &&
                 t1.min == t2.min &&
         t1.sec == t2.sec );
}
```

UNIT-2

2.1 INHERITANCE

Inheritance is one of the key feature of object-oriented programming including C++ which allows user to create a new class(derived class) from a existing class(base class). The derived class inherits all feature from a base class and it can have additional features of its own.


Base Class
Feature 1
Feature 2
Features of base class
Derived Class (Inherited from base class)
Feature 1
Feature 2
Features of base class accessible to derived class because of inheritance
Feature 3
Feature defined in derived class

Concept of Inheritance in OOP

We want to calculate either area, perimeter or diagonal length of a rectangle by taking data(length and breadth) from user. You can create three different objects(*Area*, *Perimeter* and*Diagonal*) and asks user to enter length and breadth in each object and calculate corresponding data. But, the better approach would be to create a additional object *Rectangle* to store value of length and breadth from user and derive objects *Area*, *Perimeter* and *Diagonal* from *Rectangle* base class. It is because, all three objects *Area*, *Perimeter* and *diagonal* are related to object *Rectangle*and you don't need to ask user the input data from these three derived objects as this feature is included in base class.

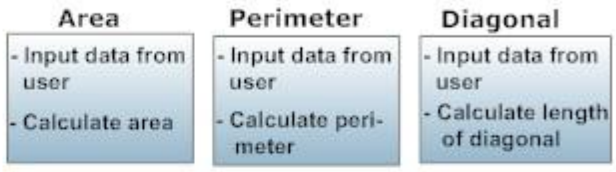


Figure: Visualization of problem without using Inheritance

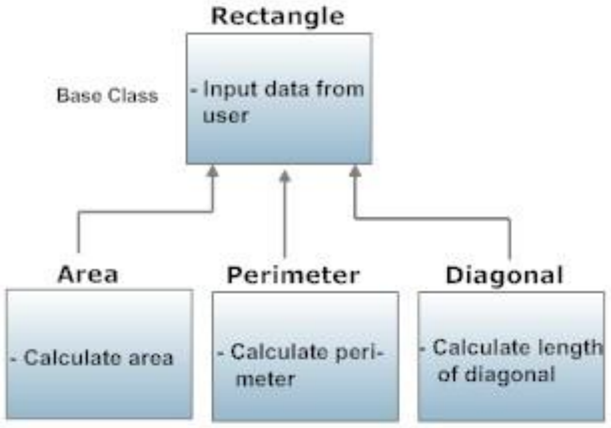


Figure: Visualization of problem using Inheritance

Note: Arrow is pointing from derived from to base class

Implementation of Inheritance in C++ Programming

```
class Rectangle {
  ... .. ...
};

class Area : public Rectangle
{
  ... .. ...
};

class Perimeter : public Rectangle
{
  .... .. ...
};
```

In the above example, class *Rectangle* is a base class and classes *Area* and *Perimeter* are the derived from *Rectangle*. The derived class appears with the declaration of class followed by a colon, the keyword *public* and the name of base class from which it is derived.

Since, *Area* and *Perimeter* are derived from *Rectangle, all data member and member function of base class Rectangle can be accessible from derived class.*

Source Code to Implement Inheritance in C++ Programming

This example calculates the area and perimeter a rectangle using the concept of inheritance.

```
/* C++ Program to calculate the area and perimeter of rectangles using concept of inheritance.
*/
#include <iostream>
using namespace std;
class Rectangle
{
   protected:
      float length, breadth;
   public:
       Rectangle(): length(0.0), breadth(0.0)
       {
          cout<<"Enter length: ";
          cin>>length;
          cout<<"Enter breadth: ";
          cin>>breadth;
       }
};
/* Area class is derived from base class Rectangle. */
class Area : public Rectangle
{
   public:
      float calc()
        {
           return length*breadth;
        }

};
/* Perimeter class is derived from base class Rectangle. */
class Perimeter : public Rectangle
{
   public:
      float calc()
        { return 2*(length+breadth);}
};
int main()
{
    cout<<"Enter data for first rectangle to find area.\n";
    Area a;
    cout<<"Area = "<<a.calc()<<" square meter\n\n";

    cout<<"Enter data for second rectangle to find perimeter.\n";
    Perimeter p;
    cout<<"\nPerimeter = "<<p.calc()<<" meter";
    return 0;
}
```

Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely,

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)

Single Inheritance

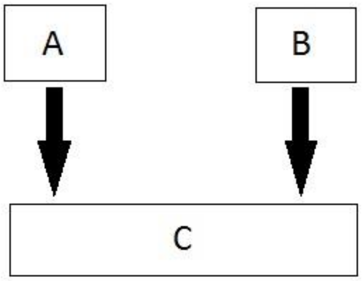
In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.

```
#include<iostream.h>
#include<conio.h>
class B
{
int a;
public:
int b;
void get_ab();
int get_a();
void show_a();
};
class D: private B
{
int c;
public:
void mul();
void display();
};
void B::get_ab()
{
cout<<"Enter Values for a and b";
cin>>a>>b;
}
int B::get_a()
```

```
{
return a;
}
void B::show_a(){
                              cout<<"a= "<<a<<"\n";
                    }
void D::mul()
{
                              get_ab();
                              c=b*get_a();
          }
void D:: display()
{
show_a();
cout<<"b= "<<b<<"\n";
cout<<"c= "<<c<<"\n\n";
}
void main()
{
                              clrscr();
                              D d;
                              d.mul();
                              d.display();
                              d.mul();
                              d.display();
                              getch();
                    }
```

Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



```
#include<iostream.h>

#include<conio.h>

 class student
```

```
{
   protected:
     int rno,m1,m2;
   public:
            void get()
          {
                    cout<<"Enter the Roll no :";
                    cin>>rno;
                    cout<<"Enter the two marks   :";
                    cin>>m1>>m2;
          }
};
class sports
{
   protected:
     int sm;                 // sm = Sports mark
   public:
            void getsm()
          {
            cout<<"\nEnter the sports mark :";
            cin>>sm;

          }
};
class statement:public student,public sports
{
   int tot,avg;
   public:
   void display()
          {
            tot=(m1+m2+sm);
```

```
              avg=tot/3;
              cout<<"\n\n\tRoll No    : "<<rno<<"\n\tTotal      : "<<tot;
             cout<<"\n\tAverage    : "<<avg;
            }
};
void main()
{
  clrscr();
  statement obj;
  obj.get();
  obj.getsm();
  obj.display();
  getch();}
```

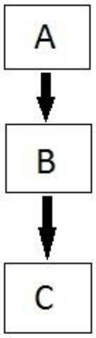
Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.

```
include<iostream.h>
#include<conio.h>
class A  //Base Class
{
   public:
   int a,b;
   void getnumber()
   {
   cout<<"\n\nEnter Number :::\t";
   cin>>a;
   }
 };
class B : public A  //Derived Class 1
{
   public:
   void square()
   {
   getnumber();  //Call Base class property
   cout<<"\n\n\tSquare of the number :::\t"<<(a*a);
   cout<<"\n\n\t--------------------------------------------------";
   }
};
 class C :public A //Derived Class 2
{
   public:
   void cube()
   {
   getnumber(); //Call Base class property
   cout<<"\n\n\tCube of the number :::\t"<<(a*a*a);
   cout<<"\n\n\t--------------------------------------------------";
   }
};

int main()
{
clrscr();
 B b1;        //b1 is object of Derived class 1
b1.square();  //call member function of class B
C c1;        //c1 is object of Derived class 2
c1.cube();   //call member function of class C
 getch();}
```

Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



When a class is derived from another derived class is called multilevel inheritance. It is implemented by defining at least three classes. In multilevel inheritance, there is one base class and the remaining two is derived class.

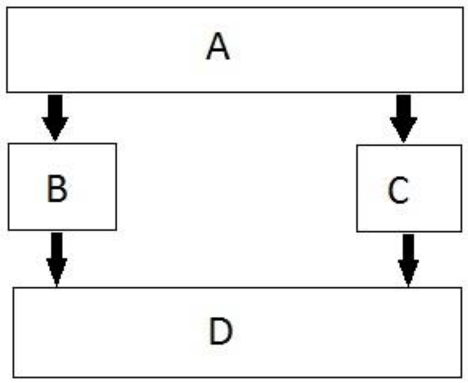
In the below program class bottom inherits property(member function) of class middle which is square() and class middle inherits property of class
top which is getdata().

```
#include<iostream.h>
#include<conio.h>
class top                    //base class
{
public :
int a;
void getdata()
{
cout<<"\n\nEnter first Number :::\t";
cin>>a;
}
void putdata()
{
cout<<"\nFirst Number Is :::\t"<<a;
}
};

//First level inheritance
```

```
class middle :public top      // class middle is derived_1
{
public:
int b;
void square()
{
getdata();
b=a*a;
cout<<"\n\nSquare Is :::"<<b;
}
};
 //Second level inheritance
class bottom :public middle    // class bottom is derived_2
{
public:
int c;
void cube()
{
square();
c=b*a;
cout<<"\n\nCube :::\t"<<c;
}
};

int main()
{
clrscr();
bottom b1;
b1.cube();
getch();
}
```

Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



2.2 OVERRIDING

What is overriding?

Defining a function in the derived class with same name as in the parent class is called overriding. In C++, the base class member can be overridden by the derived class function with the same signature as the base class function. Method overriding is used to provide different implementations of a function so that a more specific behaviour can be realized.

Consider following sample code:

```
class A
{
    int a;
    public:
     A()
     {
         a = 10;      }
     void show()
     {
        cout << a;
```

```
         }
};

int main()class B: public A
{
        int b;
        public:
        B()
        {
              b = 20;
        }
        void show()
        {
                 cout << b;
        }
};

int main()
{
        A ob1;
        B ob2;
        ob2.show(); // calls derived class show() function. o/p is 20
        return 0;
}
```

As seen above, the derived class functions override base class functions. The problem with this is, the derived class objects can not access base class member functions which are overridden in derived class.

Base class pointer can point to derived class objects; but it has access only to base members of that derived class.

Therefore, pA = &ob2; is allowed. But pa->show() will call Base class show() function and o/p would be 10.

2.3 PUBLIC, PROTECTED AND PRIVATE INHERITANCE

private inheritance: all the public and protected members in base become private.
protected inheritance: all the public and protected members in base class become protected.
public inheritance: in case of public inheritance, public remains public and protected remains protected..

```
class A
{
public:
   int x;
protected:
   int y;
private:
   int z;
};

class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};

class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

2. 4 CONSTRUCTOR AND DESTRUCTOR IN INHERITANCE

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

Points to Remember

Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.

To call base class's parameterised constructor inside derived class's parameterised constructo, we must mention it explicitly while declaring derived class's parameterized constructor.

Base class Default Constructor in Derived class Constructors

```
class Base
{ int x;
  public:
  Base() { cout << "Base default constructor"; }
};
class Derived : public Base
{ int y;
  public:
  Derived() { cout << "Derived default constructor"; }
  Derived(int i) { cout << "Derived parameterized constructor"; }};
int main()
{
 Base b;
 Derived d1;
 Derived d2(10);
}
```

You will see in the above example that with both the object creation of the Derived class, Base class's default constructor is called.

Base class Parameterized Constructor in Derived class Constructor

We can explicitly mention to call the Base class's parameterized constructor when Derived class's parameterized constructor is called.

```
class Base
```

```
{ int x;
  public:
  Base(int i)
  { x = i;
    cout << "Base Parameterized Constructor";
  }
};
class Derived : public Base
{ int y;
  public:
  Derived(int j) : Base(j)
  { y = j;
    cout << "Derived Parameterized Constructor";
  } };
int main()
{
 Derived d(10) ;
 cout << d.x ;    // Output will be 10
 cout << d.y ;    // Output will be 10
}
```

Why is Base class Constructor called inside Derived class ?

Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

Constructor call in Multiple Inheritance

Its almost the same, all the Base class's constructors are called inside derived class's constructor, in the same order in which they are inherited.

class A : public B, public C ;

In this case, first class B constructor will be executed, then class C constructor and then class A constructor.

Destructors:

Destructors are always called in the reverse order.

```
class B
{
public:


  B()
  {
    cout<<"Construct B"<<endl;
  }

~B()
  {
    cout<<"Destruct B"<<endl;
  }
};
class D : public B
{
public:
  D()
  {
    cout<<"Construct D"<<endl;
  }

~D()
  {
    cout<<"Destruct D"<<endl;
  }
};
int main()
{
  D d;
  return 0;
}
```

Output of example:

```
Construct B

Construct D

Destruct D

Destruct B
```

2.5 INHERITANCE AND COMPOSITION

One of the fundamental activities of any software system design is establishing relationships between classes. Two fundamental ways to relate classes are *inheritance* and *composition*. Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition. This article will compare these two approaches to relating classes and will provide guidelines on their use.

First, some background on the meaning of inheritance and composition.

About inheritance

In this article, I'll be talking about single inheritance through class extension, as in:

```
class Fruit {


   //...
}
class Apple extends Fruit
{
   //... }
```

In this simple example, class Apple is related to class Fruit by inheritance, because Apple extends Fruit. In this example, Fruit is the *superclass* and Apple is the *subclass*.

I won't be talking about multiple inheritance of interfaces through interface extension. That topic I'll save for next month's **Design Techniques** article, which will be focused on designing with interfaces.

Here's a UML diagram showing the inheritance relationship between Apple and Fruit:

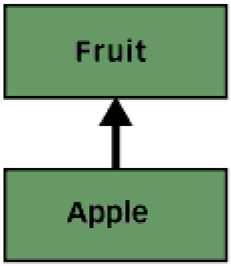


Figure 1. The inheritance relationship

About composition

By composition, I simply mean using instance variables that are references to other objects. For example:

```
class Fruit {
    //...
}class Apple {
    private Fruit fruit = new Fruit();
    //...
```

}In the example above, class Apple is related to class Fruit by composition, because Apple has an instance variable that holds a reference to a Fruit object. In this example, Apple is what I will call the *front-end class* and Fruit is what I will call the *back-end class*. In a composition relationship, the front-end class holds a reference in one of its instance variables to a back-end class.

The UML diagram showing the composition relationship has a darkened diamond, as in:

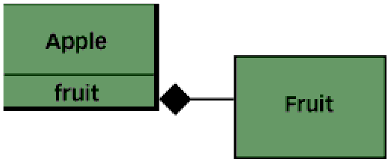


Figure 2. The composition relationship

The Address-of Operator &

The **&** operator can find address occupied by a variable. If var is a variable then, &var gives the address of that variable.

Example 1: Address-of Operator

```
#include <iostream>
using namespace std;
int main() {
   int var1 = 3;
   int var2 = 24;
   int var3 = 17;
   cout<<&var1<<endl;
   cout<<&var2<<endl;
   cout<<&var3<<endl;
}
```

Output

```
0x7fff5fbff8ac
0x7fff5fbff8a8
0x7fff5fbff8a4
```

The **0x** in the beginning represents the address is in hexadecimal form. (You may not get the same result on your system.). Notice that first address differs from second by 4-bytes and second address differs from third by 4-bytes. It is because the size of integer(variable of type int) is 4 bytes in 64-bit system.

2.6 VIRTUAL FUNCTIONS& BASE CLASS

VIRTUAL BASE CLASS:

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

```
class A
{
    public:
        int i;
};

class B : virtual public A
{
      public:
             int j;
};

class C: virtual public A
{
       public:
           int k;
};

class D: public B, public C
{
public:
     int sum;
};

int main()
{
        D ob;
        ob.i = 10; //unambiguous since only one copy of i is inherited.
        ob.j = 20;
        ob.k = 30;
        ob.sum = ob.i + ob.j + ob.k;
        cout << "Value of i is : "<< ob.i<<"\n";
        cout << "Value of j is : "<< ob.j<<"\n"; cout << "Value of k is :"<< ob.k<<"\n";
        cout << "Sum is : "<< ob.sum <<"\n";

return 0;
}.
```

What is the use virtual functions?

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raiseSalary()*, *transfer()*, *promote()*,.. etc. Different types of employees like *Manager*, *Engineer*, ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if *raiseSalary()* is present for a specific employee type, only that function would be called.

```
#include<iostream.h>
#include<conio.h>
class base
{
   public:
     virtual void show()
     {
          cout<<"\n  Base class show:";
     }
     void display()
     {
         cout<<"\n  Base class display:" ;
     }
};

class drive:public base
{
  public:
     void display()
     {
         cout<<"\n  Drive class display:";
     }
     void show()
     {
         cout<<"\n  Drive class show:";
     }
};

void main()
{
  clrscr();
  base obj1;
  base *p;
  cout<<"\n\t P points to base:\n"  ;
  p=&obj1;
  p->display();
  p->show();

  cout<<"\n\n\t P points to drive:\n";
  drive obj2;
  p=&obj2;
  p->display();
  p->show();
  getch();
}
```

Output:

```
P points to Base

Base class display
Base class show

P points to Drive

Base class Display
Drive class Show
```

2.7 POINTERS VARIABLES

Consider a normal variable as in above example, these variables holds data. But pointer variables or simply pointers are the special types of variable that holds memory address instead of data.

How to declare a pointer?

```
int *p;
```

The statement above defines a pointer variable p. The pointer p holds the memory address. The asterisk is a dereference operator which means pointer to. Here pointer p is a pointer to int, that is, it is pointing an integer. The content(value) of the memory address pointer p holds is given by expression *p.

C++ Program to demonstrate the working of pointer.

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

```
int main() {
   int *pc, c;
   c = 5;
   cout<< "Address of c (&c): " << &c << endl;
   cout<< "Value of c (c): " << c << endl << endl;
   pc = &c;    // Pointer pc holds the memory address of variable c
   cout<< "Address that pointer pc holds (pc): "<< pc << endl;
   cout<< "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;
    c = 11;    // The content inside memory address &c is changed from 5 to 11.
   cout << "Address pointer pc holds (pc): " << pc << endl;
   cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;
  *pc = 2;
   cout<< "Address of c (&c): "<< &c <<endl;
   cout<<"Value of c (c): "<< c<<endl<< endl;    return 0; }
```

Output

```
Address of c (&c): 0x7fff5fbff80c

Value of c (c): 5

Address that pointer pc holds (pc): 0x7fff5fbff80c

Content of the address pointer pc holds (*pc): 5

Address pointer pc holds (pc): 0x7fff5fbff80c

Content of the address pointer pc holds (*pc): 11

Address of c (&c): 0x7fff5fbff80c

Value of c (c): 2
```

Explanation of program

- When c = 5; the value 5 is stored in the address of variable *c*.
- When pc = &c; the pointer *pc* holds the address of *c* and the expression *pc contains the value of that address which is 5 in this case.
- When c = 11; the address that pointer *pc* holds is unchanged. But the expression *pc is changed because now the address &c (which is same as *pc*) contains 11.
- When *pc = 2; the content in the address *pc*(which is equal to &c) is changed from 11 to 2. Since the pointer *pc* and variable *c* has address, value of *c* is changed to 2.

Pointer Arithmetic:

There are only two arithmetic operations that can be performed on pointers such as addition and subtraction. The integer value can be added or subtracted from the pointer. The result of addition and subtraction is an address. The difference of the two memory addresses results an integer and not the memory address. When a pointer is incremented it points to the memory location of the next element of its base type and when it is decremented it points to the memory location of the previous element of the same base type. For example,

```
 int *x;
int *p;
p=x++;
```

there x and p are pointers of integer type. Pointer x is incremented by 1. Now variable p points to the memory location next to the memory location of the pointer x. Suppose memory address of x is 2000 and as a result p will contain memory address 2004 because integer type takes four bytes so the memory address is incremented by 4. Incrementing the pointer results in incrementing the memory address by the number of bytes occupied by the base type. For example,

```
 double *x;
double *p;
p=x++;
```

variables x and p are pointers of double type. Pointer x is incremented by 1. Now if the memory address of x was 2000 and after incrementing p will contain memory address 2008 as double takes 8 bytes. Decrementing the pointer results in decrementing the memory address by the number of bytes occupied by the base type. You cannot add two pointers. No multiplication and division can be performed on pointers. Here is a program which illustrates the working of pointer arithmetic.

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

int main ()
{
	int *x;
	int *p,*q;
	int c=100,a;
	x=&c;
	p=x+2;
	q=x-2;
	a=p-q;
	cout << "The address of x : " << x << endl;
	cout << "The address of p after incrementing x by 2 : " << p << endl;
	cout << "The address of q after derementing  x by 2 : " << q << endl;
	cout << " The no of elements between p and q :" << a << endl;
	return(0);
}
```

The result of the program is:-

Pointer to an array:

```
#include <iostream>
 using namespace std;
const int MAX = 3;
 int main ()
{
   int  var[MAX] = {10, 100, 200};
   int *ptr[MAX];

   for (int i = 0; i < MAX; i++)
   {
     ptr[i] = &var[i]; // assign the address of integer.
   }
   for (int i = 0; i < MAX; i++)
   {
     cout << "Value of var[" << i << "] = ";
     cout << *ptr[i] << endl;
   }
   return 0;
}
```

Pointer to an Object:

A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator **->** operator, just as you do with pointers to structures.

```
#include <iostream>
 using namespace std;
class Box
{
  public:
    // Constructor definition
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume()
    {
      return length * breadth * height;
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

int main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2
  Box *ptrBox;                // Declare pointer to a class.
  // Save the address of first object
  ptrBox = &Box1;

  // Now try to access a member using member access operator
  cout << "Volume of Box1: " << ptrBox->Volume() << endl;

  // Save the address of first object
  ptrBox = &Box2;

  // Now try to access a member using member access operator
  cout << "Volume of Box2: " << ptrBox->Volume() << endl;

  return 0;
}
```

this pointer

The this pointer stores the address of the class instance, to enable pointer access of the members to the member functions of the class. There is a secret not obvious to the new programmers - *this* ***pointer holds the key to the question:***
How does C++ know which object it was called on?
The answer is that C++ has a hidden pointer named *this****!***

Suppose that we create an object named **objA** of class A, and class A has a nonstatic member function void f(). If you call the function **objA.f()**, the keyword this in the body of f() stores the address of objA. The type of the this pointer for a member function of a class type A, is **A* const**.

So, when we call f(), we are actually calling **f(&objA)**. Since C++ converts the function call, it also needs to convert the function itself:

void A::f() { }

converted internally to: void A::f(A* const this);

C++ has added a new parameter to the function. The added parameter is a pointer to the class object the class function is working with. It is always named **this**. The **this** pointer is a hidden pointer inside every class member function that points to the class object. As a result, an object's this pointer is not part of the object itself; it is not reflected in the result of a sizeof statement on the object. Instead, when a nonstatic member function is called for an object, the address of the object is passed by the compiler as a hidden argument to the function.

```
#include <iostream>
class Cart {
```

```
private:
        int total;
public:
        Cart(int item){
                this->total = item;
        }
        Cart& addItem(int itemPrice) {
                total += itemPrice;
                return *this;
        }
        void checkOut() {
                std::cout << "total: " << total << " $\n";          } };
int main()
{
        Cart myCart(0);
        myCart.addItem(100);
        myCart.checkOut();
}
```

In the code, when we call

```
myCart.addItem(100);
```

we are actually calling a function converted by compiler:

```
myCart.addItem(&myCart, 100);
```

the function itself should be converted from

```
Cart& addItem(int itemPrice) {
        total += itemPrice;
        return *this;
}
```

to

```
Cart& addItem(myCart* const this, int itemPrice) {
        this->total += itemPrice;
        return *this;
}
```

Note that the following expressions are equivalent:

```
this->total;
(*this).total;
```

2.7 ABSTRACT CLASS AND CONCRETE CLASS

This C++ program differentiates between the concrete and abstract class. An abstract class is meant to be used as a base class where some or all functions are declared purely virtual and hence can not be instantiated. A concrete class is an ordinary class which has no purely virtual functions and hence can be instantiated.

Here is the source code of the C++ program which differentiates between the concrete and abstract class. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

```
/*
 * C++ Program to differentiate between concrete class and abstract class
 */
#include <iostream>
#include <string>
using namespace std;

class Abstract {
    private:
        string info;
    public:
        virtual void printContent() = 0;
};
 class Concrete {
    private:
        string info;
    public:
        Concrete(string s) : info(s) { }
        void printContent() {
            cout << "Concrete Object Information\n" << info << endl;
        }
};
 int main()
```

```
{
   /*
    * Abstract a;
    * Error : Abstract Instance Creation Failed
    */
   string s;
    s = "Object Creation Date : 23:26 PM 15 Dec 2013";
   Concrete c(s);
   c. printContent();
}
```

An abstract class is meant to be used as the base class from which other classes are derived. The derived class is expected to provide implementations for the member functions that are not implemented in the base class. A derived class that implements all the missing functionality is called a *concrete class* .

2.8 VIRTUAL DESTRUCTORS

Destructors in the Base class can be Virtual. Whenever Upcasting is done, Destructors of the Base class must be made virtual for proper destrucstion of the object when the program exits.

NOTE : Constructors are never Virtual, only Destructors can be Virtual.

Upcasting without Virtual Destructor

Lets first see what happens when we do not have a virtual Base class destructor.

```
class Base

{

 public:

 ~Base() {cout << "Base Destructor\t"; }

};

class Derived:public Base
```

```
{
 public:
 ~Derived() { cout<< "Derived Destructor"; }
};
int main()
{
 Base* b = new Derived;     //Upcasting
 delete b;
}
```

```
Output : Base Destructor
```

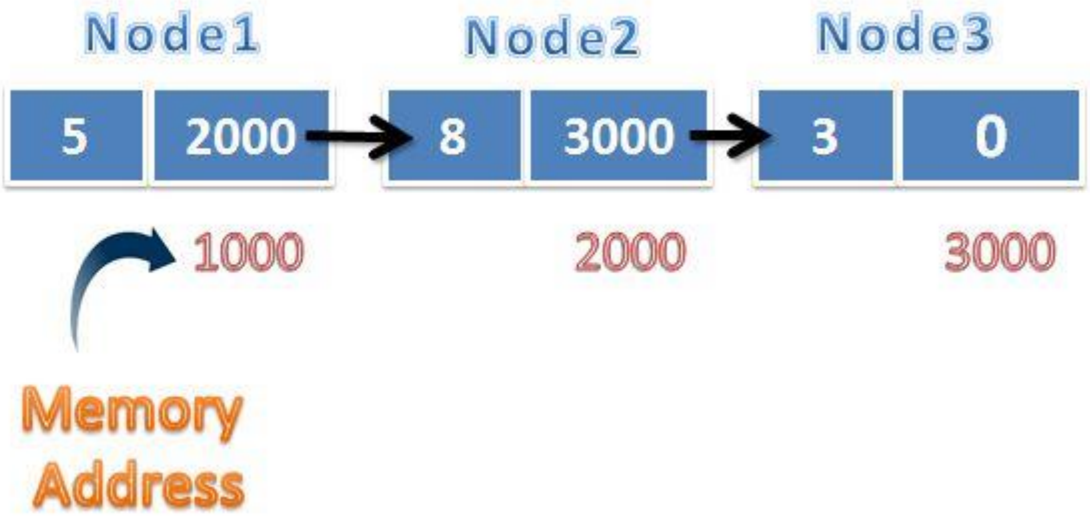
In the above example, **delete b** will only call the Base class destructor, which is undesirable because, then the object of Derived class remains undestructed, because its destructor is never called. Which results in memory leak.

2.8 DYNAMIC BINDING

UNIT-III

3.1 LINKED LIST

Linked list comprise of group or list of nodes in which each node have link to next node to form a chain.



Linked List definition

1. Linked List is **Series of Nodes**
2. Each node Consist of two Parts **Data Part & Pointer Part**
3. Pointer Part stores the **address of the next node**

Advantages of linked list:

1. Linked List is **Dynamic data Structure** .
2. Linked List **can grow and shrink during run time**.
3. **Insertion and Deletion** Operations are Easier
4. **Efficient Memory Utilization** ,i.e no need to pre-allocate memory
5. Faster Access time,can be expanded in **constant time without memory overhead**
6. Linear Data Structures such as Stack,Queue can be **easily implemeted** using Linked list

Dynamic memory allocation

The C programming language provides several functions for memory allocation and management. These functions can be found in the**<stdlib.h>** header file.

S.No	Function and Description
1	void *calloc(int num, int size); This function allocates an array of num elements each of which size in bytes will be size.

2	void free(void *address); This function release a block of memory block specified by address.
3	void *malloc(int num); This function allocates an array of num bytes and leave them initialized.
4	void *realloc(void *address, int newsize); This function re-allocates memory extending it upto newsize.

Difference Between Linked List & Linear Array

S. No.	Array	Linked List
1.	Insertions and deletions are difficult.	Insertions and deletions can be done easily.
2.	It needs movements of elements for insertion and deletion.	It does not need movement of nodes for insertion and deletion.
3.	In it space is wasted.	In it space is not wasted.
4.	It is more expensive.	It is less expensive.
5.	It requires less space as only information is stored.	It requires more space as pointers are also stored along with information.
6.	Its size is fixed.	Its size is not fixed.
7.	It can not be extended or reduced according to requirements.	It can be extended or reduced according to requirements.
8.	Same amount of time is required to access each element.	Different amount of time is required to access each element.
9.	Elements are stored in consecutive memory locations.	Elements may or may not be stored in consecutive memory locations.
10.	If have to go to a particular element then we can reach there directly.	If we have to go to a particular node then we have to go through all those nodes that come before that node.

TYPES OF LINKED LIST

- Singly linked list
- Singly circular linked list
- Doubly linked list

- Doubly circular linked list

SINGLY LINKED LIST
Single Linked Lists are uni-directional as they can only point to the next Node in the list.
A node can be represented as

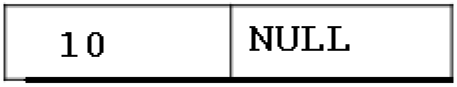
```
struct Node
{
int Data;
struct Node *Next;
};
```

Head is a pointer variable of type struct Node which acts as the Head to the list. Initially we set '**Head**' as **NULL** which means list is empty.

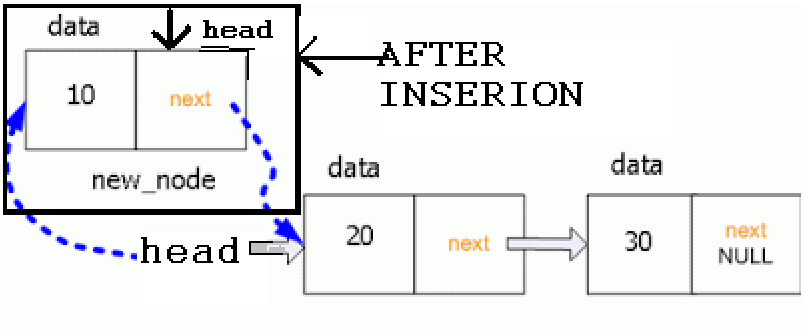
Basic Operations on a Singly Linked List

- Creation
- Traversing a List
- Inserting a Node in the List
- Deleting a Node from the List

1) Creating a node from empty list



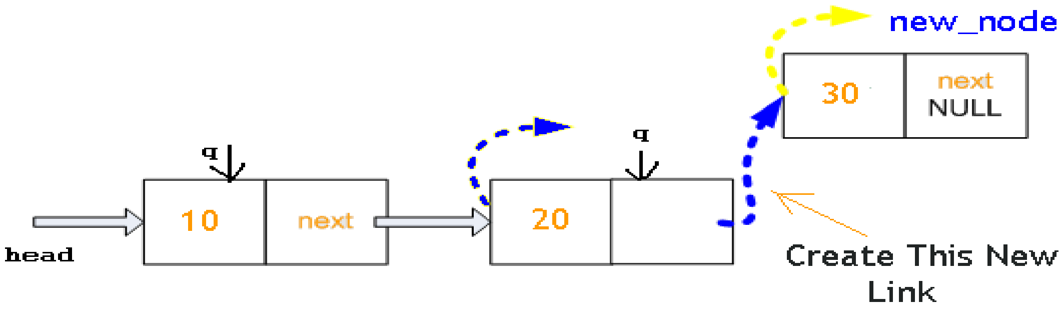
Insertion
Insertion at beginning



void single_llist::insert_begin()

```
{
   int value;
   cout<<"Enter the value to be inserted: ";
   cin>>value;
   struct node *temp, *p;
   temp = create_node(value);
   if (start == NULL)
   {
      start = temp;
      start->next = NULL;
   }
   else
   {
      p = start;
      start = temp;
      start->next = p;
   }
   cout<<"Element Inserted at beginning"<<endl;
}
```

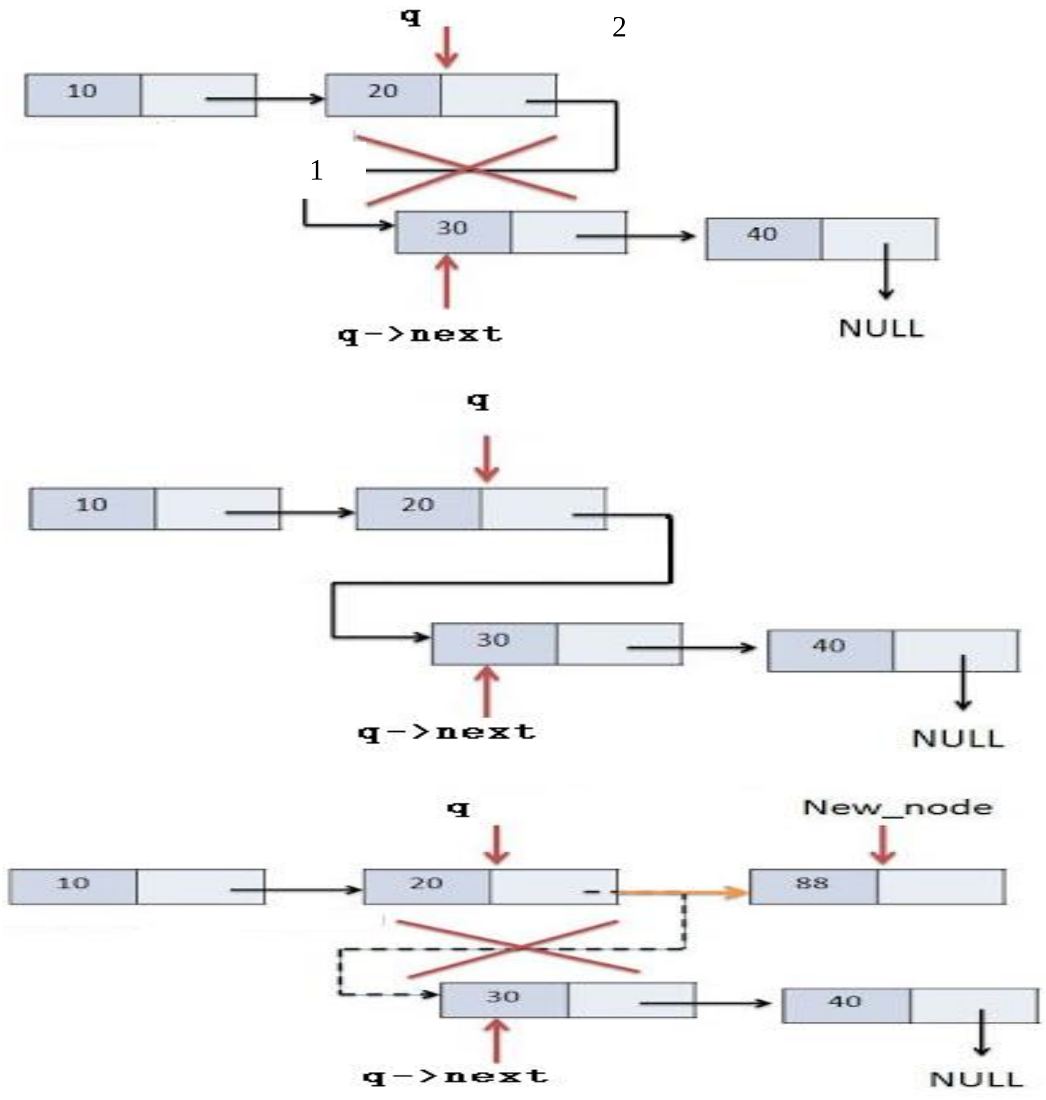
Insertion at end

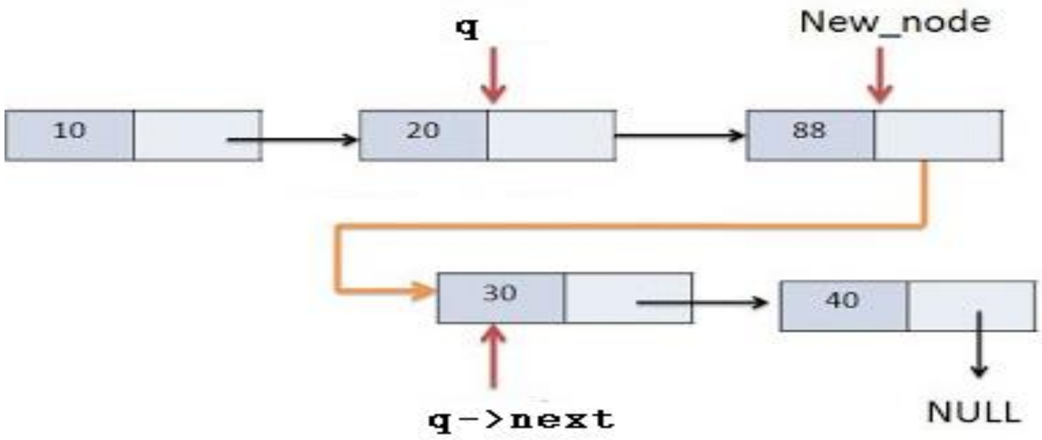


```
void single_llist::insert_last()
{
   int value;
   cout<<"Enter the value to be inserted: ";
   cin>>value;
   struct node *temp, *s;
   temp = create_node(value);
   s = start;
   while (s->next != NULL)
```

```
    {
        s = s->next;
    }
    temp->next = NULL;
    s->next = temp;
    cout<<"Element Inserted at last"<<endl;
}
```

Insertion at middle
Based on position

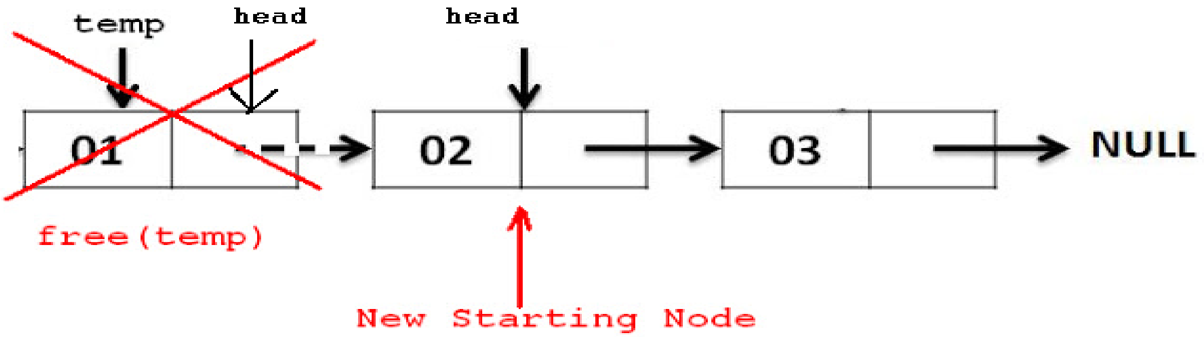




```
void single_llist::insert_pos()
{
    int value, pos, counter = 0;
    cout<<"Enter the value to be inserted: ";
    cin>>value;
    struct node *temp, *s, *ptr;
    temp = create_node(value);
    cout<<"Enter the postion at which node to be inserted: ";
    cin>>pos;
    int i;
    s = start;
    while (s != NULL)
    {
        s = s->next;
        counter++;
    }
    if (pos == 1)
    {
        if (start == NULL)
        {
            start = temp;
            start->next = NULL;
        }
        else
        {
            ptr = start;
            start = temp;
            start->next = ptr;
        }
    }
    else if (pos > 1  && pos <= counter)
    {
        s = start;
        for (i = 1; i < pos; i++)
        {
            ptr = s;
            s = s->next;
        }
        ptr->next = temp;
        temp->next = s;
    }
    else
    {
        cout<<"Positon out of range"<<endl;
    }
}
```

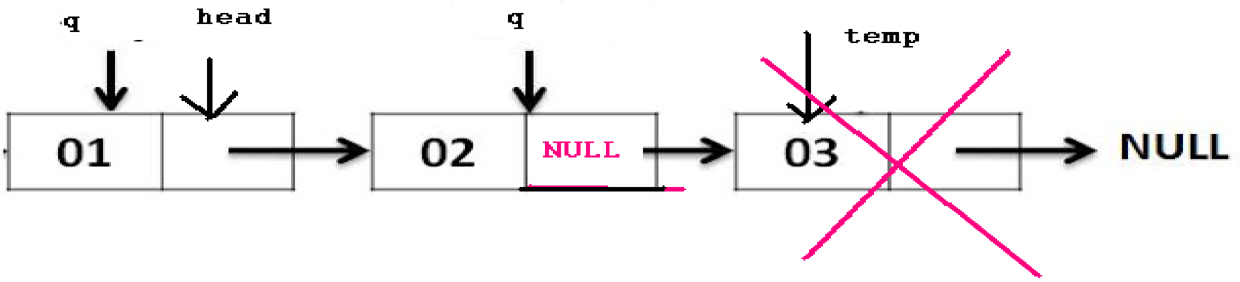
DELETION IN SINGLY LINKED LIST

AT FRONT



```
struct node *temp;
temp = head;
head= head->next;
free(temp);
printf("nThe Element deleted Successfully ");
```

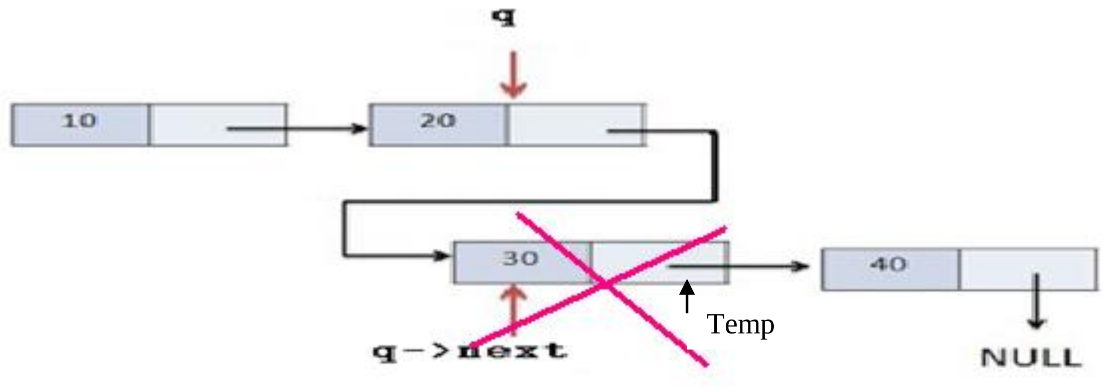
At last



```
Struct node *temp;
q=head;
while(q->next->next !=NULL)
{
```

```
                                        q=q->next;
                                        }
                                        temp=q->next;
                                        free(temp);
                                        q->next=NULL
```

Deletion at middle



```
void single_llist::delete_pos()
{
   int pos, i, counter = 0;
   if (start == NULL)
   {
      cout<<"List is empty"<<endl;
      return;
   }
   cout<<"Enter the position of value to be deleted: ";
   cin>>pos;
   struct node *s, *ptr;
   s = start;
   if (pos == 1)
   {
      start = s->next;
   }
   else
   {
      while (s != NULL)
      {
         s = s->next;
         counter++;
      }
      if (pos > 0 && pos <= counter)
      {
         s = start;
         for (i = 1;i < pos;i++)
         {
            ptr = s;
            s = s->next;
        }
        ptr->next = s->next;
    }
    else
    {
        cout<<"Position out of range"<<endl;
    }
    free(s);
    cout<<"Element Deleted"<<endl;
}
```

3.2 ARRAY IMPLEMENTATION OF LINKED LIST

```
/ A program to simulate a linear linked list using an array
/**********************************************************************************************
**********************/

#include <iostream>
#include <string>

using namespace std;

struct node {
  int data;
  int prev;
  int next;

  // Constructor:
  node() {
    data = -1;
    prev = -1;
    next = -1;
  }
};

node A[100];   // Array of nodes used to simulate the linked list
int slot = 1;  // Array index of the next free space (beginning from index 1 initially)
int head = -1; // Array index of the starting element in the list (-1 implies no entries yet)
int tail = -1; // Array index of the tail in the list
int elements = 0;  // Number of elements in the list

// Function prototypes:
void insert(int);
void insertAfter(int,int);
void deleteElement(int);
void reverse();
void print();
int findIndex(int);


int main() {
   int a, b;
   char c;
   cout << "Please input the instructions (enter 'E' to exit): \n";


   do {
      cin >> c;
      switch(c) {
            case 'A':   cin >> a;
                     insert(a);
                     break;
            case 'I':   cin >> a >> b;
                     insertAfter(a,b);
                     break;
            case 'D':   cin >> a;
                     deleteElement(a);
                     break;
            case 'R':   reverse();
                     cout << "\nLinked list successfully reversed\n\n";
                     break;
            case 'T':   print();
                     break;
            case 'E':   cout << "\nExiting program...\n\n";
                     break;
            default:    cout << "\nInvalid input entered\n\n";
      }

   } while (c != 'E');

   return 0;
}


void insert(int value) {
   if (slot == -1) {
      cout << "\nNo free space available.\n\n";
   } else {
      elements++;
      A[slot].data = value;
      A[slot].next = 0;
      if (head == -1) {
         // If the list is empty prior to the insertion
         A[slot].prev = 0;
         head = slot;
         tail = slot;
      } else {
         A[tail].next = slot;
         A[slot].prev = tail;
         tail = slot;
      }
      cout << endl << "Element \'" << value << "\'"<< " successfully inserted\n\n";


      // Finding the index of the next free location:
      do {
         do {
            slot = (slot + 1) % 100;
         } while(slot == 0);
         if (A[slot].next == -1) return;
      } while(slot != tail);
      slot = -1;
      cout << "\nNo more free space available. Please delete some elements before inserting new.\n\n";
   }
}


void insertAfter(int value1,int value2) {
   if (slot == -1) {
      cout << "\nNo free space available.\n\n";
   } else {
      int predecessor = findIndex(value1);

      if (predecessor == -1) {
         cout << "\nElement " << value1 << " not found\n\n";
         return;
      }

      if (A[predecessor].next == 0) {
         // If value1 is the last element in the list
         insert(value2);
         return;
      }

      elements++;
      A[slot].data = value2;
      A[slot].prev = predecessor;
      A[slot].next = A[predecessor].next;
      A[A[predecessor].next].prev = slot;
      A[predecessor].next = slot;
      cout << endl << "Element \'" << value2 << "\'"<< " successfully inserted\n\n";

      // Finding the index of the next free location:
      int temp = slot;
      do {
         do {
            slot = (slot + 1) % 100;
         } while(slot == 0);
         if (A[slot].next == -1) return;
      } while(slot != temp);
      slot = -1;
      cout << "\nNo more free space available. Please delete some elements before inserting
new.\n\n";
   }
}


void deleteElement(int value) {
   if (head == -1) {
      cout << "\nNo elements to delete.\n\n";
      return;
   }

   if (elements == 1) {
      // Deleting the only element in the list
      A[head].data = -1;
      A[head].prev = -1;
      A[head].next = -1;
      head = -1;
      elements--;
      return;
   }

   int index = findIndex(value);

   if (index == -1) {
      cout << "\nElement not found.\n\n";
      return;
   }

   if (index == head) {
      // Deleting the first element in the list
      int successor = findIndex(A[A[index].next].data);
      A[successor].prev = 0;
      head = successor;
      A[index].data = -1;
      A[index].prev = -1;
      A[index].next = -1;
      cout << endl << "Element \"" << value << "\""<< " successfully deleted\n\n";
      elements--;
      return;
   }

   if (index == slot) {
      // Deleting the last element in the list
      int predecessor = findIndex(A[A[index].prev].data);
      A[predecessor].next = 0;
      A[index].data = -1;
      A[index].prev = -1;
      A[index].next = -1;
      cout << endl << "Element \"" << value << "\""<< " successfully deleted\n\n";
      elements--;
      return;
   }

   int successor = findIndex(A[A[index].next].data);
   int predecessor = findIndex(A[A[index].prev].data);
   A[predecessor].next = successor;
   A[successor].prev = predecessor;
   A[index].data = -1;
   A[index].prev = -1;
   A[index].next = -1;
   cout << endl << "Element \"" << value << "\""<< " successfully deleted\n\n";
   elements--;
}

void reverse() {
   int i = head;
   tail = head;
   int temp;
   int hold;
   while (i != 0) {
      hold = i;
      i = A[i].next;

      //swap the previous and next data members to reverse the list:
      temp = A[hold].prev;
      A[hold].prev = A[hold].next;
      A[hold].next = temp;
   }
   head = hold;
}

void print() {
   cout << "\nThe current list is: ";
   int i = head;
```

```
    while (i != 0) {
        cout << A[i].data << " ";
        i = A[i].next;
    }
    cout << endl << endl;
}




int findIndex (int x) {
    // A helper function to return the index of the element x in A
    int i = head;
    while (i != 0) {
        if (A[i].data == x) {
            return i;
        }
        i = A[i].next;
    }
    return (-1);
}
```

3.3 LINKED LIST IMPLEMENTATION

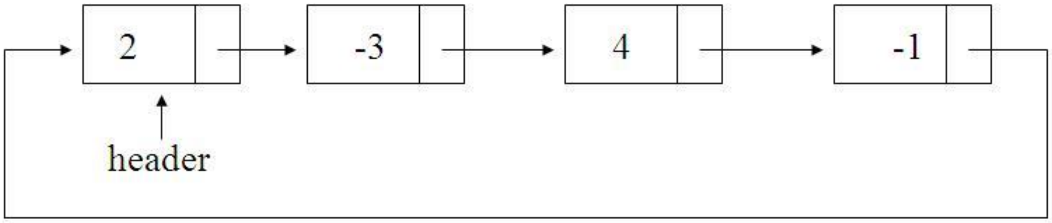
```
#include<iostream>
#include<cstdio>
#include<cstdlib>
using namespace std;
/*
 * Node Declaration
 */
struct node
{
    int info;
    struct node *next;
}*start;

/*
 * Class Declaration
 */
class single_llist
{
    public:
        node* create_node(int);
        void insert_begin();
        void insert_pos();
        void insert_last();
        void delete_pos();
        void sort();
        void search();
        void update();
        void reverse();
        void display();
        single_llist()
        {
            start = NULL;
        }
};

/*
 * Main :contains menu
 */
main()
{
    int choice, nodes, element, position, i;
    single_llist sl;
    start = NULL;
    while (1)
    {
        cout<<endl<<"---------------------------------"<<endl;
        cout<<endl<<"Operations on singly linked list"<<endl;
        cout<<endl<<"---------------------------------"<<endl;
        cout<<"1.Insert Node at beginning"<<endl;
        cout<<"2.Insert node at last"<<endl;
        cout<<"3.Insert node at position"<<endl;
        cout<<"4.Sort Link List"<<endl;
        cout<<"5.Delete a Particular Node"<<endl;
        cout<<"6.Update Node Value"<<endl;
        cout<<"7.Search Element"<<endl;
        cout<<"8.Display Linked List"<<endl;
        cout<<"9.Reverse Linked List "<<endl;
        cout<<"10.Exit "<<endl;
        cout<<"Enter your choice : ";
        cin>>choice;
        switch(choice)
        {
        case 1:
            cout<<"Inserting Node at Beginning: "<<endl;
```

```
      sl.insert_begin();
      cout<<endl;
      break;
    case 2:
      cout<<"Inserting Node at Last: "<<endl;
      sl.insert_last();
      cout<<endl;
      break;
    case 3:
      cout<<"Inserting Node at a given position:"<<endl;
      sl.insert_pos();
      cout<<endl;
      break;
    case 4:
      cout<<"Sort Link List: "<<endl;
      sl.sort();
      cout<<endl;
      break;
    case 5:
      cout<<"Delete a particular node: "<<endl;
      sl.delete_pos();
      break;
    case 6:
      cout<<"Update Node Value:"<<endl;
      sl.update();
      cout<<endl;
      break;
    case 7:
      cout<<"Search element in Link List: "<<endl;
      sl.search();
      cout<<endl;
      break;
    case 8:
      cout<<"Display elements of link list"<<endl;
      sl.display();
      cout<<endl;
      break;
    case 9:
      cout<<"Reverse elements of Link List"<<endl;
      sl.reverse();
      cout<<endl;
      break;
    case 10:
            cout<<"Exiting..."<<endl;
            exit(1);
            break;
        default:
            cout<<"Wrong choice"<<endl;
        }
    }
}

/*
 * Creating Node
 */
node *single_llist::create_node(int value)
{
    struct node *temp, *s;
    temp = new(struct node);
    if (temp == NULL)
    {
        cout<<"Memory not allocated "<<endl;
        return 0;
    }
    else
    {
        temp->info = value;
        temp->next = NULL;
        return temp;
    }
}

/*
 * Inserting element in beginning
 */
void single_llist::insert_begin()
{
    int value;
    cout<<"Enter the value to be inserted: ";
    cin>>value;
    struct node *temp, *p;
    temp = create_node(value);
    if (start == NULL)
    {
        start = temp;
        start->next = NULL;
    }
    else
    {
        p = start;
        start = temp;
        start->next = p;
    }
    cout<<"Element Inserted at beginning"<<endl;
}

/*
 * Inserting Node at last
 */
void single_llist::insert_last()
{
    int value;
    cout<<"Enter the value to be inserted: ";
    cin>>value;
    struct node *temp, *s;
    temp = create_node(value);
    s = start;
    while (s->next != NULL)
    {
        s = s->next;
    }
    temp->next = NULL;
    s->next = temp;
    cout<<"Element Inserted at last"<<endl;
}

/*
 * Insertion of node at a given position
 */
void single_llist::insert_pos()
{
    int value, pos, counter = 0;
    cout<<"Enter the value to be inserted: ";
    cin>>value;
    struct node *temp, *s, *ptr;
    temp = create_node(value);
    cout<<"Enter the postion at which node to be inserted: ";
    cin>>pos;
    int i;
    s = start;
    while (s != NULL)
    {
        s = s->next;
        counter++;
    }
    if (pos == 1)
    {
        if (start == NULL)
        {
            start = temp;
            start->next = NULL;
        }
        else
        {
            ptr = start;
            start = temp;
            start->next = ptr;
        }
    }
    else if (pos > 1  && pos <= counter)
    {
        s = start;
        for (i = 1; i < pos; i++)
        {
            ptr = s;
            s = s->next;
        }
        ptr->next = temp;
        temp->next = s;
    }
    else
    {
        cout<<"Positon out of range"<<endl;
    }
}

/*
 * Sorting Link List
 */
void single_llist::sort()
{
    struct node *ptr, *s;
    int value;
    if (start == NULL)
    {
        cout<<"The List is empty"<<endl;
        return;
    }
    ptr = start;
    while (ptr != NULL)
    {
        for (s = ptr->next;s !=NULL;s = s->next)
        {
            if (ptr->info > s->info)
            {
                value = ptr->info;
                ptr->info = s->info;
                s->info = value;
            }
        }
        ptr = ptr->next;
    }
}

/*
 * Delete element at a given position
 */
void single_llist::delete_pos()
{
    int pos, i, counter = 0;
    if (start == NULL)
    {
        cout<<"List is empty"<<endl;
        return;
    }
    cout<<"Enter the position of value to be deleted: ";
    cin>>pos;
    struct node *s, *ptr;
    s = start;
    if (pos == 1)
    {
        start = s->next;
    }
    else
    {
        while (s != NULL)
        {
            s = s->next;
            counter++;
        }
        if (pos > 0 && pos <= counter)
        {
            s = start;
            for (i = 1;i < pos;i++)
            {
                ptr = s;
                s = s->next;
            }
            ptr->next = s->next;
        }
        else
        {
            cout<<"Position out of range"<<endl;
        }
        free(s);
        cout<<"Element Deleted"<<endl;
    }
}

/*
 * Update a given Node
 */
void single_llist::update()
{
    int value, pos, i;
    if (start == NULL)
    {
        cout<<"List is empty"<<endl;
        return;
    }
    cout<<"Enter the node postion to be updated: ";
    cin>>pos;
    cout<<"Enter the new value: ";
    cin>>value;
    struct node *s, *ptr;
    s = start;
    if (pos == 1)
    {
        start->info = value;
    }
    else
    {
        for (i = 0;i < pos - 1;i++)
        {
            if (s == NULL)
            {
                cout<<"There are less than "<<pos<<" elements";
                return;
            }
            s = s->next;
        }
        s->info = value;
    }
    cout<<"Node Updated"<<endl;
}

/*
 * Searching an element
 */
void single_llist::search()
{
    int value, pos = 0;
    bool flag = false;
    if (start == NULL)
    {
        cout<<"List is empty"<<endl;
        return;
    }
    cout<<"Enter the value to be searched: ";
    cin>>value;
    struct node *s;
    s = start;
    while (s != NULL)
    {
        pos++;
        if (s->info == value)
        {
            flag = true;
            cout<<"Element "<<value<<" is found at position "<<pos<<endl;
        }
        s = s->next;
    }
    if (!flag)
        cout<<"Element "<<value<<" not found in the list"<<endl;
}

/*
 * Reverse Link List
 */
void single_llist::reverse()
{
    struct node *ptr1, *ptr2, *ptr3;
    if (start == NULL)
    {
        cout<<"List is empty"<<endl;
        return;
    }
    if (start->next == NULL)
    {
        return;
    }
    ptr1 = start;
    ptr2 = ptr1->next;
    ptr3 = ptr2->next;
    ptr1->next = NULL;
    ptr2->next = ptr1;
    while (ptr3 != NULL)
    {
        ptr1 = ptr2;
        ptr2 = ptr3;
        ptr3 = ptr3->next;
        ptr2->next = ptr1;
    }
    start = ptr2;
}

/*
 * Display Elements of a link list
 */
void single_llist::display()
{
    struct node *temp;
    if (start == NULL)
    {
```

```
    cout<<"The List is Empty"<<endl;
    return;
  }
  temp = start;
  cout<<"Elements of list are: "<<endl;
  while (temp != NULL)
  {
    cout<<temp->info<<"->";
    temp = temp->next;
  }
  cout<<"NULL"<<endl;
}
```

CIRCULAR LINKED LIST

Circular list is a list in which the link field of the last node is made to point to the start/first node of the list.



The last node link back to the first (or the header).

Points to Circular Linked List

1.A circular linked list is a linked list in which the head element's previous pointer points to the last element and the last element's next pointer points to the head element.
2.A circularly linked list node looks exactly the same as a linear singly linked list.
Insertion In Circular Linked List

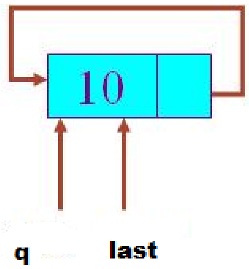
There are three situation for inserting element in Circular linked list.
1.Insertion at the front of Circular linked list.
2.Insertion in the middle of the Circular linked list.
3.Insertion at the end of the Circular linked list.

Rrepresentation of node:

```
Struct node
{
int data;
Struct node *next;
} *last;
```

Creation:

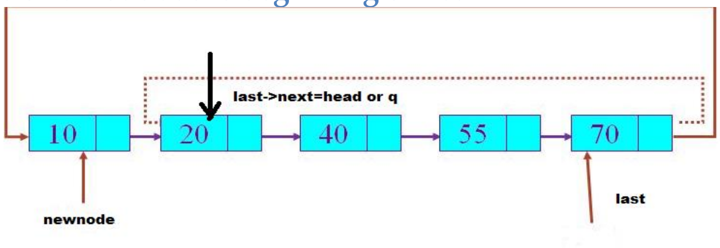
Insert into an empty Circular Linked list



```
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 at the front of Circular linked list

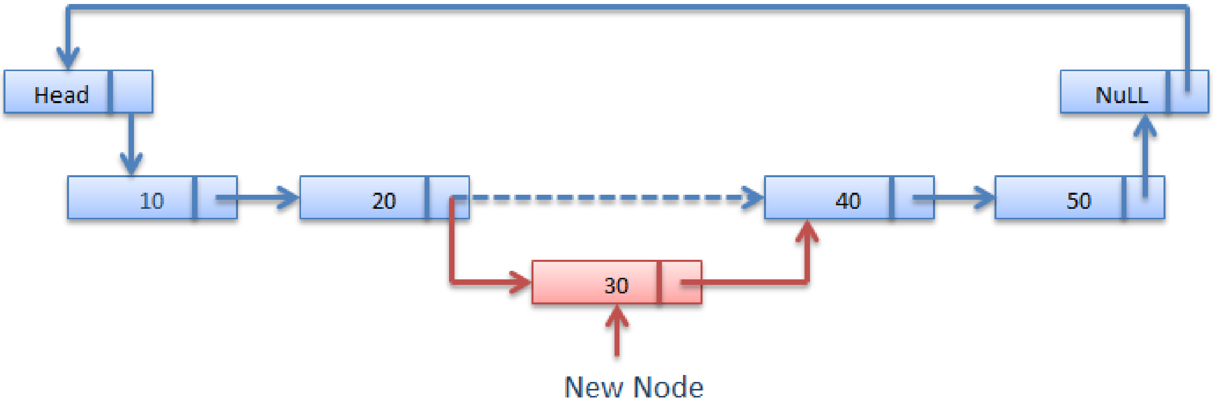
Procedure for insertion a node at the beginning of list



```
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 in the middle of the Circular linked list

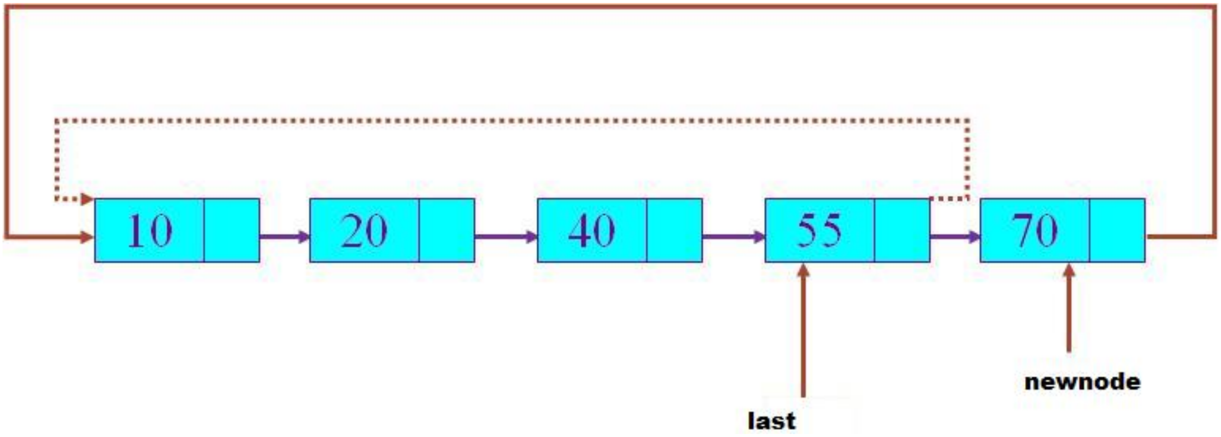
Insertion in the middle of the linked list is similar to the insertion in the middle of singly linked list



```
void circular_llist::add_after(int value, int pos)
{
   if (last == NULL)
   {
      cout<<"First Create the list."<<endl;
      return;
```

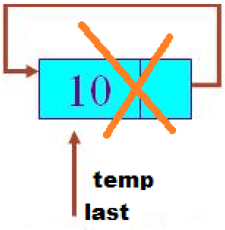
```
    }
    struct node *temp, *s;
    s = last->next;
    for (int i = 0;i < pos-1;i++)
    {
        s = s->next;
        if (s == last->next)
        {
            cout<<"There are less than ";
            cout<<pos<<" in the list"<<endl;
            return;
        }
    }
    temp = new(struct node);
    temp->next = s->next;
    temp->info = value;
    s->next = temp;
    /*Element inserted at the end*/
    if (s == last)
    {
        last=temp;
    }
}
```

Insertion at the end of Circular linked list



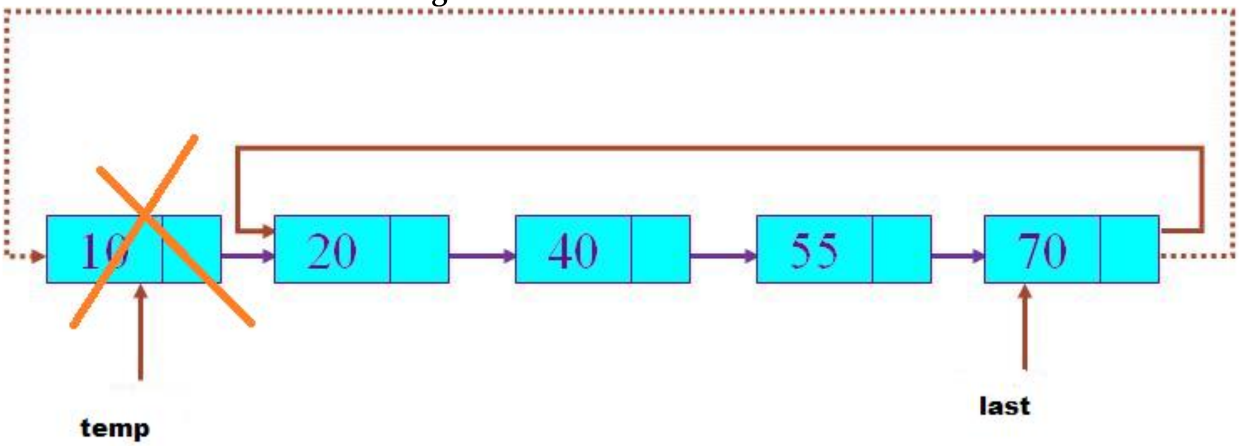
Procedure for insertion a node at the end of list

Delete a node from a single-node Circular Linked List



```
/*Deletion of last element*/
if (s->next->info == value)
{
    temp = s->next;
    s->next = last->next;
    free(temp);
    last = s;
    return;
}
```

Delete a node at front from a single-node Circular Linked List

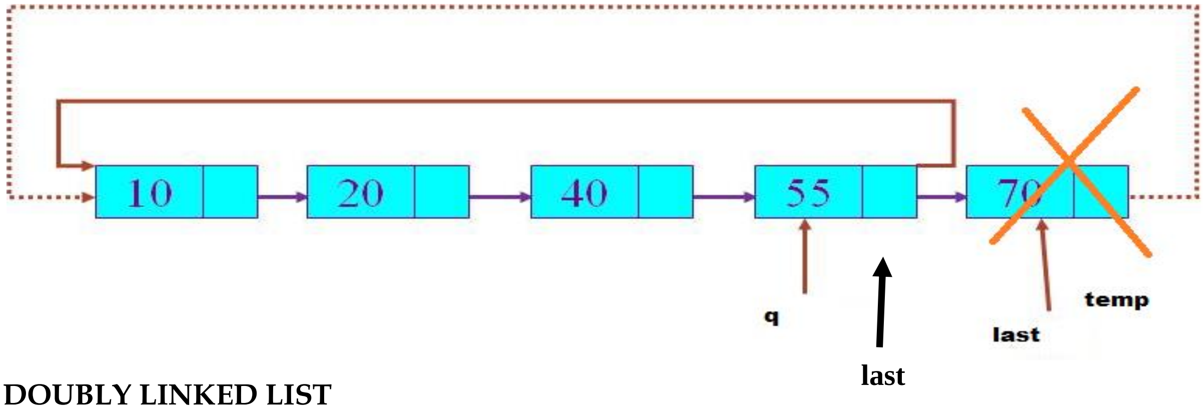


```
if (s->info == value)  /*First Element Deletion*/
{       temp = s;
 last->next = s->next;
free(temp);
return;
}
```

Deletion the middle of the Circular linked list

Deletion in the middle of the linked list is similar to the Deletion in the middle of singly linked list

```
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;
  }
```

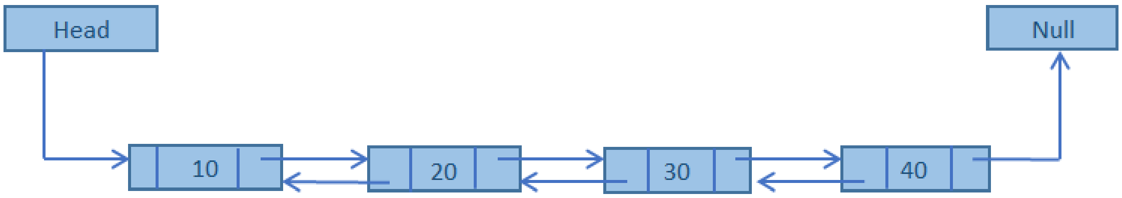


DOUBLY LINKED LIST

In this type of liked list each node holds two-pointer field. Pointers exist between adjacent nodes in both directions. The list can be traversed either forward or backward.

NEXT holds the memory location of the Previous Node in the List.
PREV holds the memory location of the Next Node in the List.

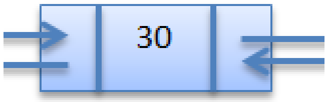
DATA holds Data



Creation

struct Node

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

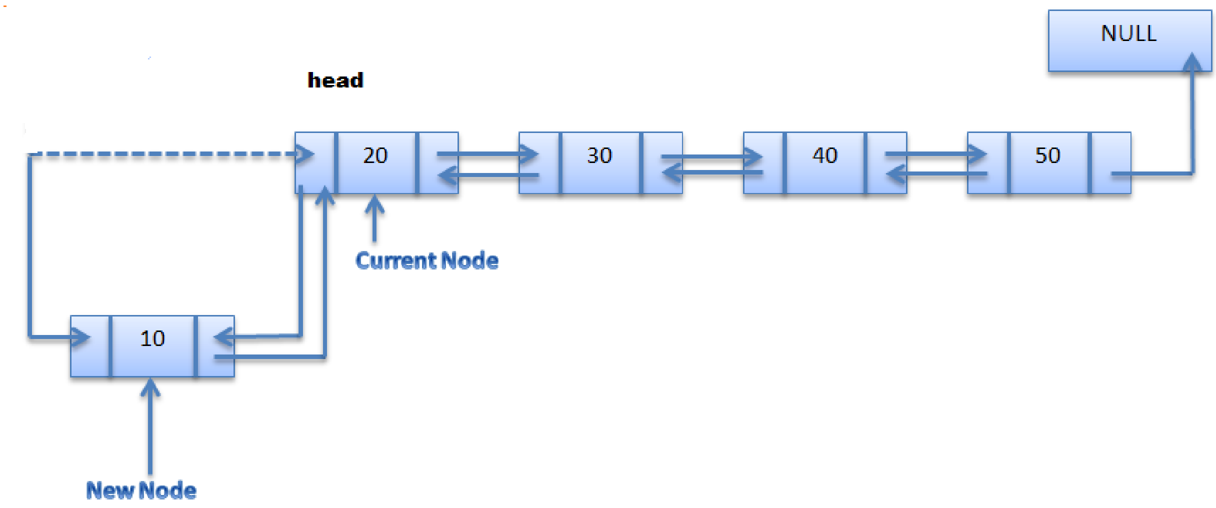


```
void double_llist::create_list(int value)
{
    struct node *s, *temp;
    temp = new(struct node);
    temp->info = value;
    temp->next = NULL;
    if (start == NULL)
    {
        temp->prev = NULL;
        start = temp;
    }
    else
    {
        s = start;
        while (s->next != NULL)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

Insertion in doubly linked list

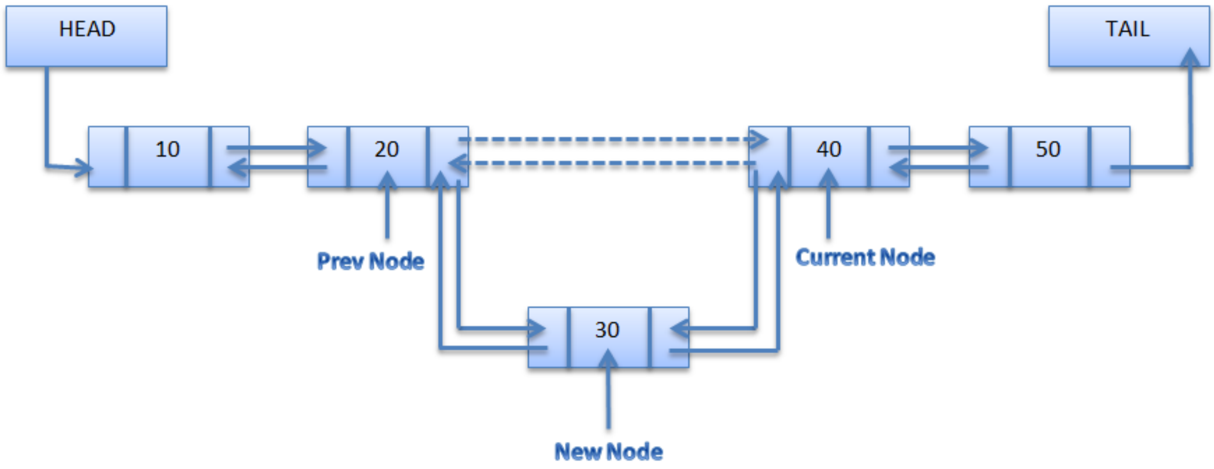
There are three situation for inserting element in list.
1.Insertion at the front of list.
2.Insertion in the middle of the list.
3.Insertion at the end of the list.

1.Insertion at the front of list.



```
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 = sta head
   start->prev = temp;
   start = temp;
   cout<<"Element Inserted"<<endl;
}
```

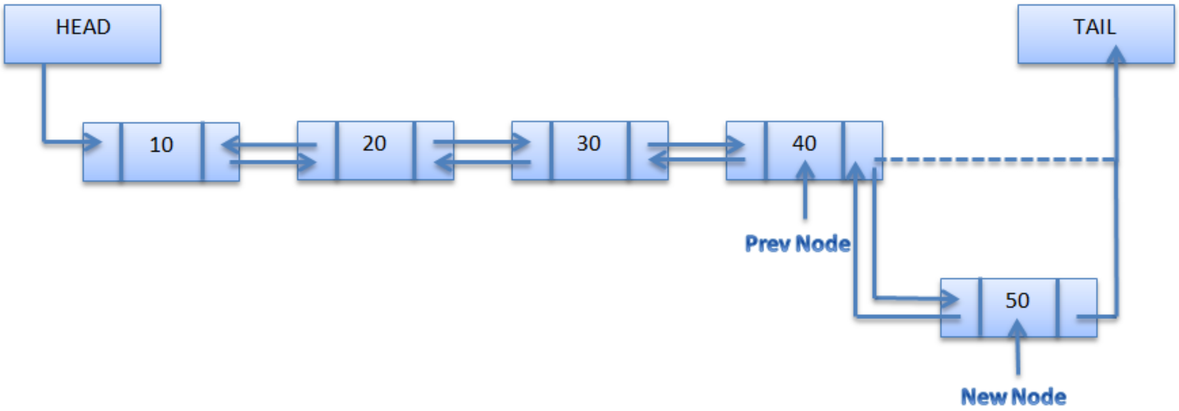
2.Insertion in the middle of the list.



```
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;
}
```

3. Insertion at the end of the list.



```
new_node=(struct node *)malloc(sizeof(struct node));
scanf("%d",&new_node->data);
q=head;
while(q->next!=NULL)
{
q=q->next;
}
q->next=newnode;
newnode->next=NULL;
newnode->prev=q;
```

DELETION:

```
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;}
```

APPLICATIONS OF LINKED LIST

- Polynomial Operations
- Radix Sort

3.5 POLYNOMIAL MANIPULATION

Representation of a Polynomial: A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

P(x) = $4x^3+6x^2+7x+9$

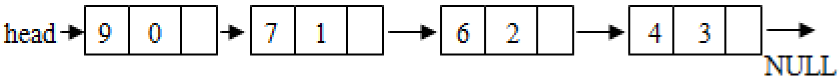
A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below:

arr	9	7	6	4	(coefficients)
	0	1	2	3	(exponents)

A polynomial may also be represented using a linked list. A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
 struct polynomial
{
int coefficient;
int exponent;
struct polynomial *next;
};
```

Thus the above polynomial may be represented using linked list as shown below:



Addition of two Polynomials:

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result. The complete program to add two polynomials is given in subsequent section.

Multiplication of two Polynomials:

Multiplication of two polynomials however requires manipulation of each node such that the exponents are added up and the coefficients are multiplied. After each term of first polynomial is operated upon with each term of the second polynomial, then the result has to be added up by comparing the exponents and adding the coefficients for similar exponents and including terms as such with dissimilar exponents in the result.

```
#include"stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class LinkList
{
protected:
struct node
{
int exp;
int info;
struct node* next;
}; typedef struct node * NODEPTR;
NODEPTR listPtr;
public:
LinkList(){
listPtr=0;
}
int IsEmpty();
void PushPoly(int cof,int ex);
//void Mult(list);
void Add(LinkList Poly1 , LinkList Poly2);
void Display(int num);
};
void LinkList::PushPoly(int cof , int ex){
NODEPTR p;
p = new node;
p->info=cof;
p->exp=ex;
p->next=listPtr;
listPtr=p;

}
void LinkList::Add(LinkList Poly1,LinkList Poly2){
int count1 , count2;
LinkList PolyR;
NODEPTR p,q,r;
for(p=Poly1.listPtr; p!=0 ; p=p->next) {
count1++;
}
for(q=Poly2.listPtr; q!=0 ; q=q->next)
count2++;

if(count1>count2)
{

for(int i=0;i<=count1;i++)
PolyR.PushPoly(0,0);
}
else if(count2>count1)
{
for(int i=1;i<=count2;i++)
PolyR.PushPoly(0,0);

}
else
{
for(int i=1;i<=count1;i++)
PolyR.PushPoly(0,0);

}

for(p=Poly1.listPtr, q=Poly2.listPtr,r=PolyR.listPtr; p!=0 && q!=0&&r!=0; p=p->next, q=q->next,
r=r->next)
{

if(p->exp==q->exp)
{
r->info = p->info + q->info;
r->exp=p->exp;
}
else
{
r->info = p->info;
r->exp=p->exp;
}
}
PolyR.Display(3);
cout<<"result "<<endl;

}
void LinkList::Display(int num){
NODEPTR p;
cout<<"Polynomial "<<num<<endl<<endl;
for(p=listPtr;p!=0;p=p->next){
cout <<p->info<<" x^"<<p->exp<<(p->next!=0?" + ":" ");
}cout<<endl;
}
int main(){
LinkList poly1,poly2;
poly1.PushPoly(4,3);
poly1.PushPoly(5,2);
poly1.PushPoly(6,1);
poly2.PushPoly(6,3);
```

```
poly2.PushPoly(5,2);
poly2.PushPoly(7,1);

poly1.Display(1);
cout<<endl;
poly2.Display(2);
cout<<endl;
poly1.Add(poly1,poly2);
//cout <<endl<<endl;
//poly1.multi(poly2);
//poly1.evaluate();

getch();
return 0;
}
```

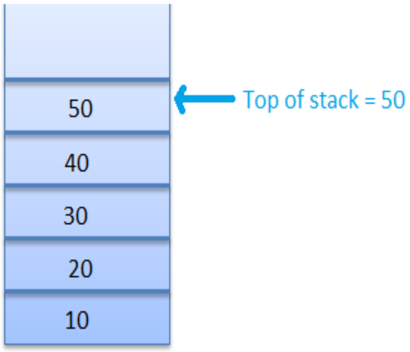
3.6 STACK

Stack is abstract data type and linear data structure.

Concept of Stack

A Stack is data structure in which addition of new element or deletion of existing element always takes place at a same end. **This end is known as the top of the stack**. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack.

One other way of describing the stack is as a last in, first out (LIFO) abstract data type and linear data structure.



Structure of Stack

Operations on Stack

The stack is basically performed two operations PUSH and POP.
Push and pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively.

- PUSH:- PUSH operation performed for the adding item to the stack.
- POP:- POP operation performed for removing an item from a stack.

Implementation of stack

- Array based implementation
- Linked list implementation

Array based implementation

PUSH FUNCTION

```
void push()
{
        int n;
                cin>>n
        if(top==size-1)
        {
                Cout<<"Stack is Full";
        }
        else
        {
                top=top+1;
                stack[top]=n;
        }
}
```

POP FUNCTION

```
void pop()
{
        int item;
        if(top==-1)
        {
                Cout<<"Stack is Empty";
        }
        else
        {
                item=stack[top];
                cout<<"item popped is="<< item;
                top--;
        }
}

/* static implementation of stack*/
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;



class stack
{
        int stk[5];
        int top;
    public:
        stack()
         {
           top=-1;
         }
        void push(int x)
         {
           if(top >  4)
                {
                  cout <<"stack over flow";
                  return;
                }
           stk[++top]=x;
           cout <<"inserted" <<x;
         }
        void pop()
         {
            if(top <0)
             {
                 cout <<"stack under flow";
                 return;
             }
             cout <<"deleted" <<stk[top--];
          }
        void display()
         {
             if(top<0)
             {
                   cout <<" stack empty";
                   return;
             }
             for(int i=top;i>=0;i--)
```

```
               cout <<stk[i] <<" ";
             }
};

main()
{
    int ch;
    stack st;
    while(1)
      {
          cout <<"\n1.push  2.pop  3.display  4.exit\nEnter ur choice";
          cin >> ch;
          switch(ch)
           {
            case 1:  cout <<"enter the element";
                   cin >> ch;
                   st.push(ch);
                   break;
            case 2:  st.pop();  break;
            case 3:  st.display();break;
            case 4:  exit(0);
            }
       }
return (0);
}
```

Linked list implementation

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

class stack : public node
{
         node *head;
         int tos;
     public:
          stack()
           {
```

```
    tos=-1;
 }
void push(int x)
 {
        if (tos < 0 )
   {
     head =new node;
     head->next=NULL;
     head->data=x;
     tos ++;
    }
else
   {
         node *temp,*temp1;
     temp=head;
     if(tos >= 4)
        {
                cout <<"stack over flow";
        return;
       }
     tos++;
     while(temp->next != NULL)
        temp=temp->next;
     temp1=new node;
     temp->next=temp1;
     temp1->next=NULL;
     temp1->data=x;
    }
  }
void display()
 {
   node *temp;
   temp=head;
   if (tos < 0)
    {
       cout <<" stack under flow";
       return;
     }
   while(temp != NULL)
     {
       cout <<temp->data<< " ";
       temp=temp->next;
     }
 }
```

```
        void pop()
        {
          node *temp;
          temp=head;
          if( tos < 0 )
           {
             cout <<"stack under flow";
             return;
           }
           tos--;
           while(temp->next->next!=NULL)
            {
               temp=temp->next;
            }
           temp->next=NULL;
         }
};
main()
{
       stack s1;
       int ch;
       while(1)
       {
       cout <<"\n1.PUSH\n2.POP\n3.DISPLAY\n4.EXIT\n enter ur choice:";
       cin >> ch;
       switch(ch)
          {
              case 1:   cout <<"\n enter a element";
                         cin >> ch;
                         s1.push(ch);
                         break;
              case 2:   s1.pop();break;



case 3:   s1.display();
                         break;
              case 4:   exit(0);
              }
   }
return (0);
}
```

3.8 APPLICATION OF STACK

1.Expression Evolution
2.Expression conversion

a.Infix to Postfix.
b.Infix to Prefix.
c.Postfix to Infix.
d.Prefix to Infix.

3.Parsing- balancing symbols
4.Simulation of recursion
5.Fuction call

Procedure for Postfix Conversion

1.	Scan the Infix string from left to right.
2.	Initialize an empty stack.
3.	If the scanned character is an operand, add it to the Postfix string.
4.	If the scanned character is an operator and if the stack is empty push the character to stack.
5.	If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack.
6.	If top Stack has higher precedence over the scanned character pop the stack else push the scanned character to stack. Repeat this step until the stack is not empty and top Stack has precedence over the character.
7.	Repeat 4 and 5 steps till all the characters are scanned.
8.	After all characters are scanned, we have to add any character that the stack may have to the Postfix string.
9.	If stack is not empty add top Stack to Postfix string and Pop the stack.
10.	Repeat this step as long as stack is not empty.

Algorithm for Postfix Conversion

```
S:stack
while(more tokens)
  x<=next token
    if(x == operand)
      print x
    else
      while(precedence(x)<=precedence(top(s)))
        print(pop(s))
    push(s,x)
while(! empty (s))  print(pop(s))
```

Conversion To Postfix
EXAMPLE:

A+(B*C-(D/E-F)*G)*H

Stack	Input	Output
Empty	A+(B*C-(D/E-F)*G)*H	-
Empty	+(B*C-(D/E-F)*G)*H	A
+	(B*C-(D/E-F)*G)*H	A
+(	B*C-(D/E-F)*G)*H	A
+(	*C-(D/E-F)*G)*H	AB
+(*	C-(D/E-F)*G)*H	AB
+(*	-(D/E-F)*G)*H	ABC
+(-	(D/E-F)*G)*H	ABC*
+(-(	D/E-F)*G)*H	ABC*
+(-(	/E-F)*G)*H	ABC*D
+(-(/	E-F)*G)*H	ABC*D
+(-(/	-F)*G)*H	ABC*DE
+(-(-	F)*G)*H	ABC*DE/
+(-(-	F)*G)*H	ABC*DE/
+(-(-	)*G)*H	ABC*DE/F
+(-	*G)*H	ABC*DE/F-
+(-*	G)*H	ABC*DE/F-
+(-*	)*H	ABC*DE/F-G
+	*H	ABC*DE/F-G*-
+*	H	ABC*DE/F-G*-
+*	End	ABC*DE/F-G*-H
Empty	End	ABC*DE/F-G*-H*+

Evaluating a postfix expression

—Use a stack to evaluate an expression in postfix notation
The postfix expression to be evaluated is scanned from left to right
Each operator in a postfix string refers to the previous two operands in the string

—Each time we read an operand we push it into a stack. When we reach an operator, its operands will then be top two elements on the stack. We can then pop these two elements, perform the indicated operation on them, and push the result on the stack
—So that it will be available for use as an operand of the next operator

Algorithm for evaluating a postfix expression

```
WHILE more input items exist
{
If symb is an operand
then push (operand_stk, symb)
else   //symbol is an operator
{
Opnd1=pop(operand_stk);
Opnd2=pop(operand_stk);
Value = opnd2 symb opnd1
Push(operand_stk, value);
}  //End of else
} // end while
```

Check for balanced parentheses in an expression

Given an expression string exp, write a program to examine whether the pairs and the orders of “{“,”}”,”(“,”)”,”[“,”]” are correct in exp. For example, the program should print true for exp = “[()]{}{[()()]()}” and false for exp = “[(])”

Algorithm:
1) Declare a character stack S.
2) Now traverse the expression string exp.
   a) If the current character is a starting bracket (‘(‘ or ‘{‘ or ‘[‘) then push it to stack.
   b) If the current character is a closing bracket (‘)’ or ‘}’ or ‘]’) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
3) After complete traversal, if there is some starting bracket left in stack then “not balanced”

```
  #include <iostream>
#include <conio.h>
using namespace std;
struct node
{
   char data;
   node *next;
}*p = NULL, *top = NULL, *save = NULL,*ptr;
void push(char x)
{
   p = new node;
   p->data = x;
   p->next = NULL;
   if (top == NULL)
   {
      top = p;
   }
   else
   {
      save = top;
      top = p;
      p->next = save;
   }
}
char pop()
{
   if (top == NULL)
   {
      cout<<"underflow!!";
   }
   else
   {
      ptr = top;
      top = top->next;
      return(ptr->data);
      delete ptr;
   }
}
int main()
{
   int i;
   char c[30], a, y, z;
   cout<<"enter the expression:\n";
   cin>>c;
   for (i = 0; i < strlen(c); i++)
   {
        if ((c[i] == '(') || (c[i] == '{') || (c[i] == '['))
        {
         push(c[i]);
        }
        else
        {
           switch(c[i])
           {
                    case ')':
                   a = pop();
                           if ((a == '{') || (a == '['))
                                  {
                                     cout<<"invalid expr!!";
                      getch();
                                  }
                                  break;
                    case '}':
                   y = pop();
                                  if ((y == '[') || (y == '('))
                                  {
                                     cout<<"invalid expr!!";
                      getch();
                                  }
                                  break;
                    case ']':
                   z = pop();
                                  if ((z == '{') || (z == '('))
                                  {
                                     cout<<"invalid expr!!";
                      getch();
                                  }
                                  break;
           }
        }
   }
   if (top == NULL)
   {
       cout<<"balanced expr!!";
   }
   else
   {
       cout<<"string is not valid.!!";
   }
   getch();
}
```

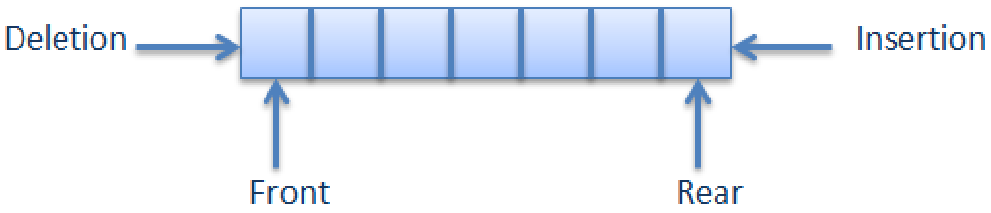
Output

```
enter the expression:
[{{{}}}{{()[[]]}}]    balanced expr!!
```

3.7 QUEUES

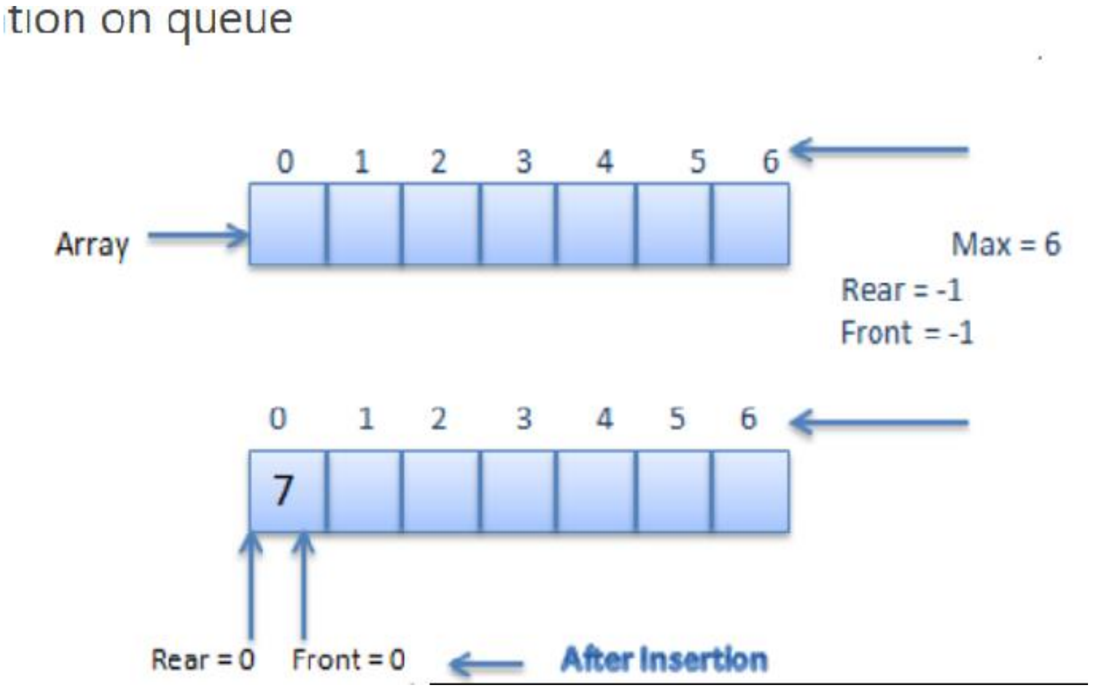
Queues is a kind of abstract data type where items are inserted one end (rear end) known as **enqueue** operation and deteted from the other end(front end) known as **dequeue** operation. This makes the queue a **First-In-First-Out (FIFO)** data structure.

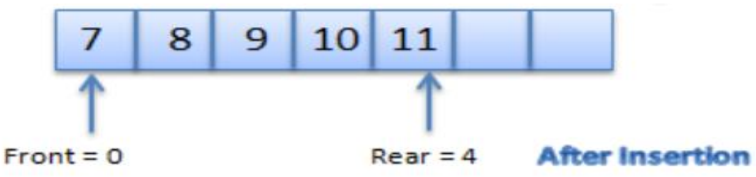
The queue performs the function of a buffer.

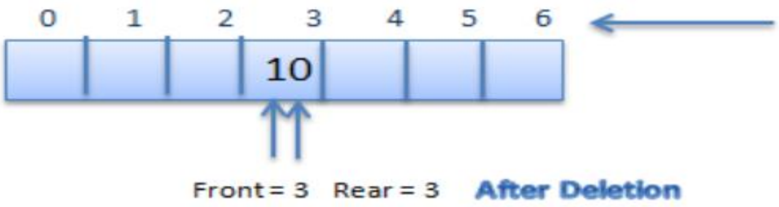


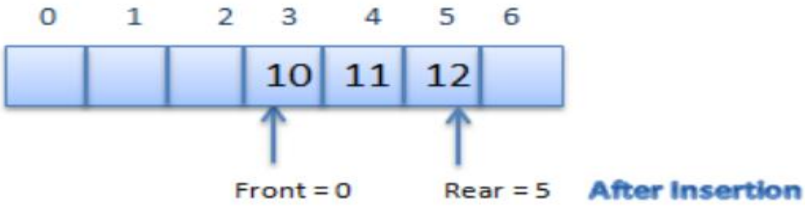
Operation on Queues

Operation	Description
initialize()	Initializes a queue by adding the value of rear and font to -1.
enqueue()	Insert an element at the rear end of the queue.
dequeue()	Deletes the front element and return the same.
empty()	It returns true(1) if the queue is empty and return false(0) if the queue is not empty.
full()	It returns true(1) if the queue is full and return false(0) if the queue is not full.









Implementation of Queues

1. Implementation using Array (Static Queue)
2. Implementation using Linked List (Dynamic Queue)
1. Implementation using Array (Static Queue)

```
#include<iostream>
#include<conio.h>
#include<stdlib.h>
using namespace std;

class queue
{
          int queue1[5];
          int rear,front;
    public:
          queue()
           {
               rear=-1;
               front=-1;
           }
          void insert(int x)
           {
              if(rear >  4)
          {
            cout <<"queue over flow";
            front=rear=-1;
            return;
          }
          queue1[++rear]=x;
          cout <<"inserted" <<x;
        }
       void delet()
        {
          if(front==rear)
           {
             cout <<"queue under flow";
             return;
           }
           cout <<"deleted" <<queue1[++front];
        }


       void display()
        {
          if(rear==front)
           {
             cout <<" queue empty";
             return;
           }
          for(int i=front+1;i<=rear;i++)
          cout <<queue1[i]<<" ";
        }
};

main()
{
   int ch;
   queue qu;
   while(1)
    {
       cout <<"\n1.insert  2.delet  3.display  4.exit\nEnter ur choice";
       cin >> ch;
       switch(ch)
        {
         case 1:    cout <<"enter the element";
                     cin >> ch;
                   qu.insert(ch);
                   break;
         case 2:  qu.delet();  break;
         case 3:  qu.display();break;
```

```
              case 4: exit(0);
              }
        }
return (0);
}
```

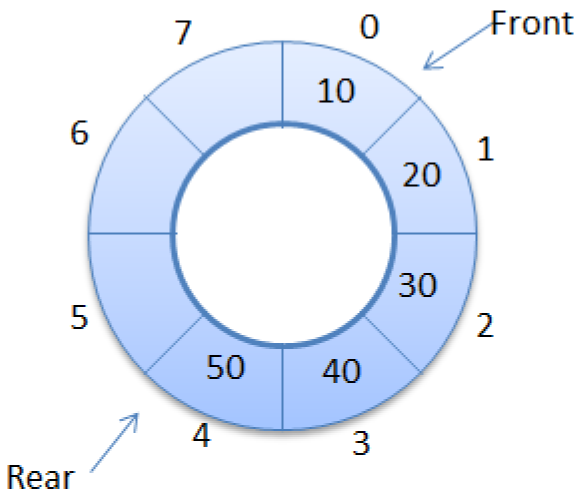
2. Implementation using Linked List (Dynamic Queue)

```
#include<iostream>
#include<stdio.h>
#include<conio.h>
using namespace std;
struct node
{
   int data;
   node *next;}*front = NULL,*rear = NULL,*p = NULL,*np = NULL;
void push(int x)
{
   np = new node;
   np->data = x;
   np->next = NULL;
   if(front == NULL)
   {
      front = rear = np;
      rear->next = NULL;
   }
   else
   {
      rear->next = np;
      rear = np;
      rear->next = NULL;
   }
}
int remove()
{
   int x;
   if(front == NULL)
   {
      cout<<"empty queue\n";
   }
   else
   {
      p = front;
      x = p->data;
      front = front->next;
      delete(p);
      return(x);
   }
}
int main()
{
   int n,c = 0,x;
   cout<<"Enter the number of values to be pushed into queue\n";
   cin>>n;
   while (c < n)
   {
          cout<<"Enter the value to be entered into queue\n";
          cin>>x;
      push(x);
      c++;    }
    cout<<"\n\nRemoved Values\n\n";
    while(true)
    {
      if (front != NULL)
         cout<<remove()<<endl;
      else
         break;
    }    getch();}
```

CIRCULAR QUEUE

A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue. Circular queues have a fixed size.

Circular queue follows FIFO principle. Queue items are added at the rear end and the items are deleted at front end of the circular queue.



Algorithm for Insertion in a circular queue.

```
Insert CircularQueue ( )
1. If (FRONT == 1 and REAR == N) or (FRONT == REAR + 1) Then
2.         Print: Overflow
3. Else
4.         If (REAR == 0) Then [Check if QUEUE is empty]
                   (a) Set FRONT = 1
                   (b) Set REAR = 1
5. Else If (REAR == N) Then [If REAR reaches end if QUEUE]
6.                 Set REAR = 1
7. Else
8.                 Set REAR = REAR + 1 [Increment REAR by 1]
  [End of Step 4 If]
9. Set QUEUE[REAR] = ITEM
10. Print: ITEM inserted
         [End of Step 1 If]
11. Exit
```

Implementation of insertion in circular queue

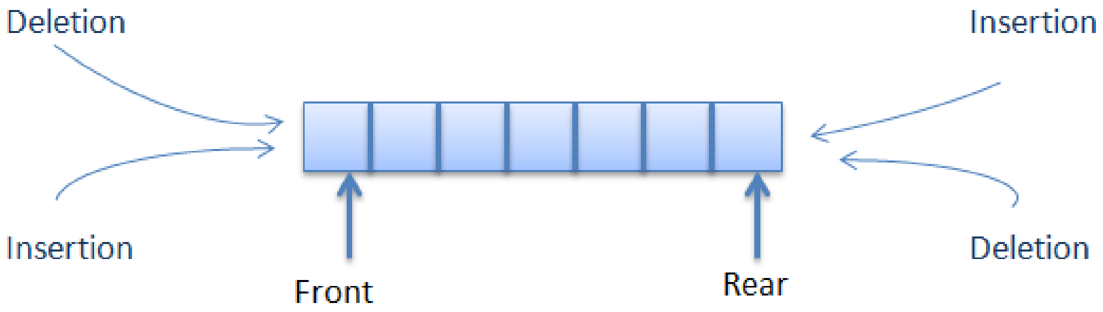
```
#include <iostream>
#define MAX 5
using namespace std;
/*
 * Class Circular Queue
 */
class Circular_Queue
{
   private:
      int *cqueue_arr;
      int front, rear;
   public:
      Circular_Queue()
      {
         cqueue_arr = new int [MAX];
         rear = front = -1;
      }
      /*
       * Insert into Circular Queue
       */
      void insert(int item)
```

```
{
    if ((front == 0 && rear == MAX-1) || (front == rear+1))
    {
        cout<<"Queue Overflow \n";
        return;
    }
    if (front == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        if (rear == MAX - 1)
            rear = 0;
        else
            rear = rear + 1;
    }
    cqueue_arr[rear] = item ;
}
/*
 * Delete from Circular Queue
 */
void del()
{
    if (front == -1)
    {
        cout<<"Queue Underflow\n";
        return ;
    }
    cout<<"Element deleted from queue is : "<<cqueue_arr[front]<<endl;
    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else
    {
        if (front == MAX - 1)
            front = 0;
        else
            front = front + 1;
    }
}
/*
 * Display Circular Queue
 */
```

```
        void display()
        {
            int front_pos = front, rear_pos = rear;
            if (front == -1)
            {
                cout<<"Queue is empty\n";
                return;
            }
            cout<<"Queue elements :\n";
            if (front_pos <= rear_pos)
            {
                while (front_pos <= rear_pos)
                {
                    cout<<cqueue_arr[front_pos]<<"  ";
                    front_pos++;
                }
            }
            else
            {
                while (front_pos <= MAX - 1)
                {
                    cout<<cqueue_arr[front_pos]<<"  ";
                    front_pos++;
                }
                front_pos = 0;
                while (front_pos <= rear_pos)
                {
                    cout<<cqueue_arr[front_pos]<<"  ";
                    front_pos++;
                }
            }
            cout<<endl;
        }
};
/*
 * Main
 */
int main()
{
    int choice, item;
    Circular_Queue cq;
    do
    {
        cout<<"1.Insert\n";
        cout<<"2.Delete\n";
        cout<<"3.Display\n";
        cout<<"4.Quit\n";
    cout<<"Enter your choice : ";
    cin>>choice;
    switch(choice)
    {
    case 1:
       cout<<"Input the element for insertion in queue : ";
       cin>>item;
       cq.insert(item);
           break;
        case 2:
       cq.del();
           break;
    case 3:
       cq.display();
           break;
        case 4:
           break;
        default:
           cout<<"Wrong choice\n";
        }/*End of switch*/
}
while(choice != 4);
return 0;}
```

Double-Ended Queue

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.



Algorithm for Insertion at rear end

```
Step -1: [Check for overflow]
         if(rear==MAX)
```

```
                Print("Queue is Overflow”);
      return;
Step-2: [Insert element]
        else
                rear=rear+1;
                q[rear]=no;
                [Set rear and front pointer]
                if rear=0
                        rear=1;
                if front=0
                        front=1;
Step-3: return
```

Algorithm for Insertion at font end

```
Step-1 : [Check for the front position]
        if(front<=1)
                Print (“Cannot add item at front end”);
        return;
Step-2  : [Insert at front]
        else
                front=front-1;
                q[front]=no;
Step-3  : Return
```

Algorithm for Deletion from front end

```
Step-1 [ Check for front pointer]
        if front=0
                print(" Queue is Underflow”);
        return;
Step-2 [Perform deletion]
        else
                no=q[front];
                print(“Deleted element is”,no);
                [Set front and rear pointer]
        if front=rear
                front=0;
                rear=0;
        else
                front=front+1;
step-3 : Return
```

Algorithm for Deletion from rear end

```
Step-1 : [Check for the rear pointer]
        if rear=0
                print(“Cannot delete value at rear end”);
        return;
Step-2: [ perform deletion]
```

```
        else
                no=q[rear];
                [Check for the front and rear pointer]
        if front= rear
                front=0;
                rear=0;
        else
                rear=rear-1;
                print("Deleted element is",no);
Step-3 : Return
```

PROGRAM:

```
#include <iostream>

#include <cstdlib>

using namespace std;

/*

* Node Declaration

*/

struct node

{

int info;

node *next;

node *prev;


}*head, *tail;


/*

* Class Declaration

*/

class dqueue

{

public:

int top1, top2;

void insert();

void del();
void display();
dqueue()
{
top1 = 0;
top2 = 0;
head = NULL;
tail = NULL;
}
};
□
/*
* Main: Contains Menu
*/
int main()
{
int choice;
dqueue dl;
while (1)
{
cout<<"\n--------------"<<endl;
cout<<"Operations on Deque"<<endl;
cout<<"\n--------------"<<endl;
cout<<"1.Insert Element into the Deque"<<endl;
cout<<"2.Delete Element from the Deque"<<endl;
cout<<"3.Traverse the Deque"<<endl;
cout<<"4.Quit"<<endl;
cout<<"Enter your Choice: ";
cin>>choice;
cout<<endl;
switch(choice)
{
```

```
case 1:
dl.insert();
break;
case 2:
dl.del();
break;
case 3:
dl.display();
break;
case 4:
exit(1);
break;
default:
cout<<"Wrong Choice"<<endl;
}
}
return 0;
}

/*
* Insert Element in Doubly Ended Queue
*/
void dqueue::insert()
{
struct node *temp;
int ch, value;
if (top1 + top2 >= 50)
{
cout<<"Dequeue Overflow"<<endl;
return;
}
```

```
if (top1 + top2 == 0)
{
cout<<"Enter the value to be inserted: ";
cin>>value;
head = new (struct node);
head->info = value;
head->next = NULL;
head->prev = NULL;
tail = head;
top1++;
cout<<"Element Inserted into empty deque"<<endl;
}
else
{
while (1)
{
cout<<endl;
cout<<"1.Insert Element at first"<<endl;
cout<<"2.Insert Element at last"<<endl;
cout<<"3.Exit"<<endl;
cout<<endl;
cout<<"Enter Your Choice: ";
cin>>ch;
cout<<endl;
switch(ch)
{
case 1:
cout<<"Enter the value to be inserted: ";
cin>>value;
temp = new (struct node);
temp->info = value;
temp->next = head;
temp->prev = NULL;
head->prev = temp;
head = temp;
top1++;
break;
case 2:
cout<<"Enter the value to be inserted: ";
cin>>value;
temp = new (struct node);
temp->info = value;
temp->next = NULL;
temp->prev = tail;
tail->next = temp;
tail = temp;
top2++;
break;
case 3:
return;
break;
default:
cout<<"Wrong Choice"<<endl;
}
}
}
}
□
/*
* Delete Element in Doubly Ended Queue
*/
void dqueue::del()
```

```
{
if (top1 + top2 <= 0)
{
cout<<"Deque Underflow"<<endl;
return;
}
int ch;
while (1)
{
cout<<endl;
cout<<"1.Delete Element at first"<<endl;
cout<<"2.Delete Element at last"<<endl;
cout<<"3.Exit"<<endl;
cout<<endl;
cout<<"Enter Your Choice: ";
cin>>ch;
cout<<endl;
switch(ch)
{
case 1:
head = head->next;
head->prev = NULL;
top1--;
break;
case 2:
tail = tail->prev;
tail->next = NULL;
top2--;
break;
case 3:
return;
```

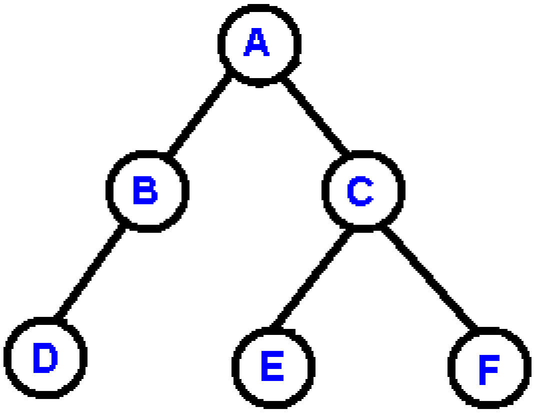
```
break;
default:
cout<<"Wrong Choice"<<endl;
}
}
}
□
/*
* Display Doubly Ended Queue
*/
void dqueue::display()
{
struct node *temp;
int ch;
if (top1 + top2 <= 0)
{
cout<<"Deque Underflow"<<endl;
return;
}
while (1)
{
cout<<endl;
cout<<"1.Display Deque from Beginning"<<endl;
cout<<"2.Display Deque from End"<<endl;
cout<<"3.Exit"<<endl;
cout<<endl;
cout<<"Enter Your Choice: ";
cin>>ch;
cout<<endl;
switch (ch)
{
```

```
case 1:
temp = head;
cout<<"Deque from Beginning:"<<endl;
while (temp != NULL)
{
cout<<temp->info<<" ";
temp = temp->next;
}
cout<<endl;
break;
case 2:
cout<<"Deque from End:"<<endl;
temp = tail;
while (temp != NULL)
{
cout<<temp->info<<" ";
temp = temp->prev;
}
temp = tail;
cout<<endl;
break;
case 3:
return;
break;
default:
cout<<"Wrong Choice"<<endl;
}
}
}
```

UNIT-4

4.1 TREES

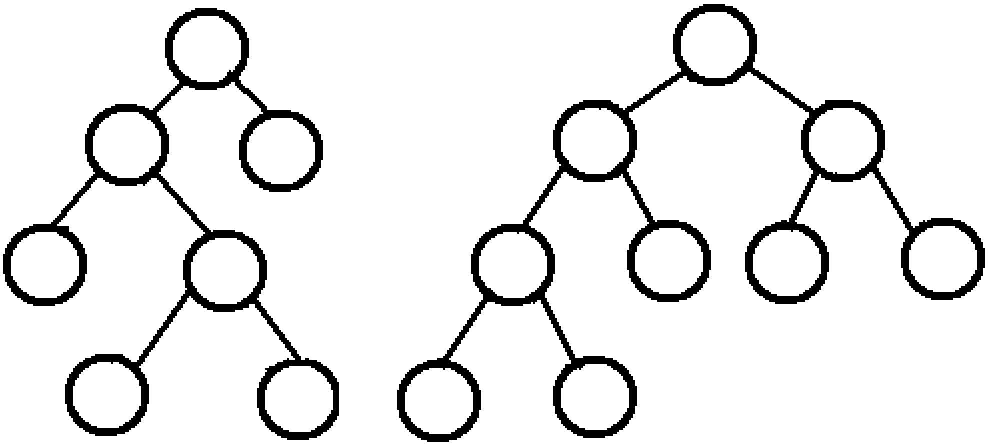
A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.



Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.

More tree terminology:

- The depth of a node is the number of edges from the root to the node.
- The height of a node is the number of edges from the node to the deepest leaf.
- The height of a tree is a height of the root.
- A full binary tree.is a binary tree in which each node has exactly zero or two children.
- A complete binary tree is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.



full tree complete tree

A complete binary tree is very special tree, it provides the best possible ratio between the number of nodes and the height. The height h of a complete binary tree with N nodes is at most O(log N). We can easily prove this by counting nodes on each level, starting with the root, assuming that each level has the maximum number of nodes:

$$n = 1 + 2 + 4 + \ldots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

Solving this with respect to h, we obtain

$$h = O(\log n)$$

where the big-O notation hides some superfluous details.

Advantages of trees

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

- Trees reflect structural relationships in the data

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

4.3 Traversals

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds

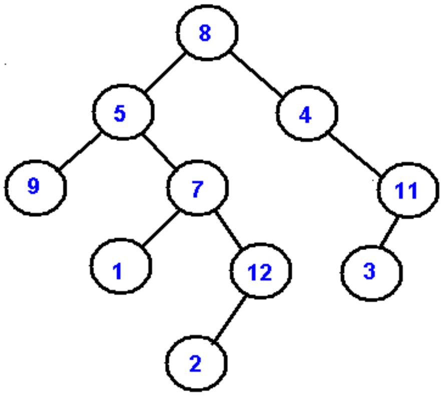
- depth-first traversal
- breadth-first traversal

There are three different types of depth-first traversals, :

- PreOrder traversal - visit the parent first and then left and right children;
- InOrder traversal - visit the left child, then the parent and the right child;
- PostOrder traversal - visit left child, then the right child and then the parent;

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.

As an example consider the following tree and its four traversals:



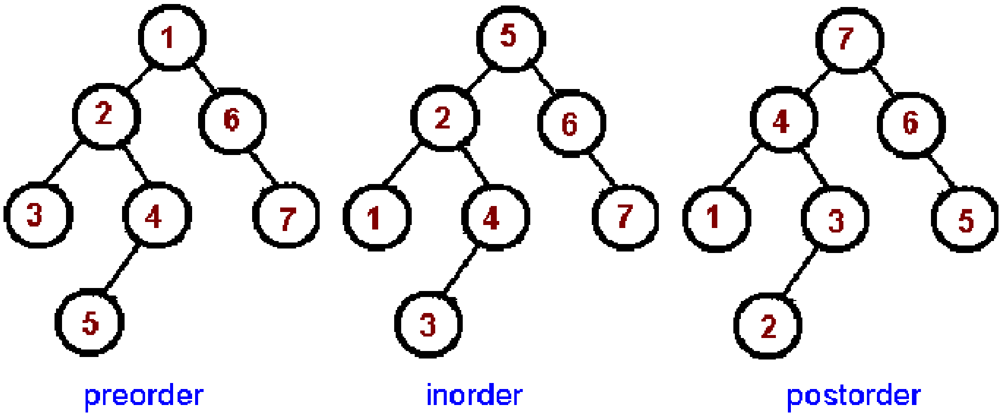
PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3

InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8

LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

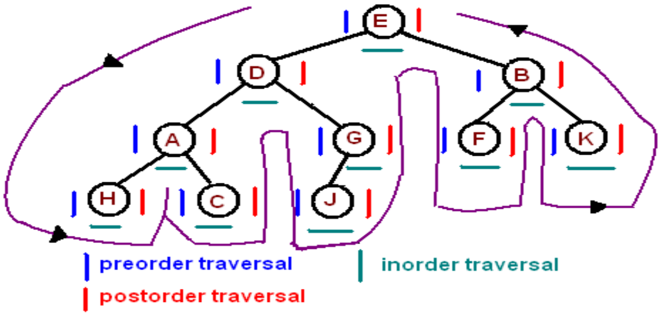
In the next picture we demonstarte the order of node visitation. Number 1 denote the first node in a particular traversal and 7 denote the last node.


preorder

inorder

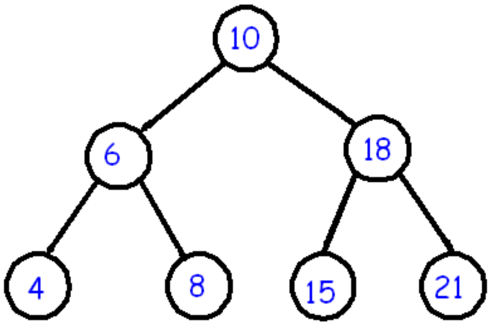
postorder

These common traversals can be represented as a single algorithm by assuming that we visit each node three times. An *Euler tour* is a walk around the binary tree where each edge is treated as a wall, which you cannot cross. In this walk each node will be visited either on the left, or under the below, or on the right. The Euler tour in which we visit nodes on the left produces a preorder traversal. When we visit nodes from the below, we get an inorder traversal. And when we visit nodes on the right, we get a postorder traversal.



4.2 Binary Search Trees

We consider a particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retrieving.



A BST is a binary tree where nodes are ordered in the following way:

- each node contains one key (also known as data)
- the keys in the left subtree are less then the key in its parent node, in short L < P;
- the keys in the right subtree are greater the key in its parent node, in short P < R;
- duplicate keys are not allowed.

In the following tree all nodes in the left subtree of 10 have keys < 10 while all nodes in the right subtree > 10. Because both the left and right subtrees of a BST are again search trees; the above definition is recursively applied to all internal nodes:

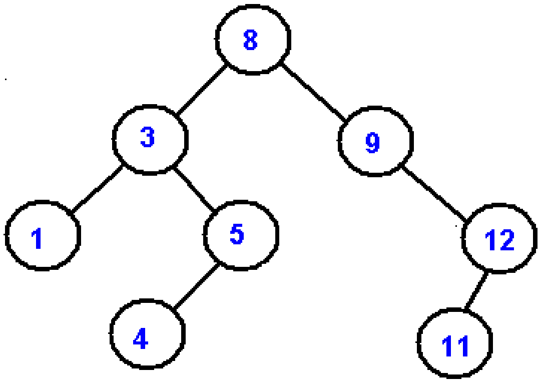
Implementation

We implement a binary search tree using a private inner class BSTNode. In order to support the *binary search tree property*, we require that data stored in each node is Comparable:

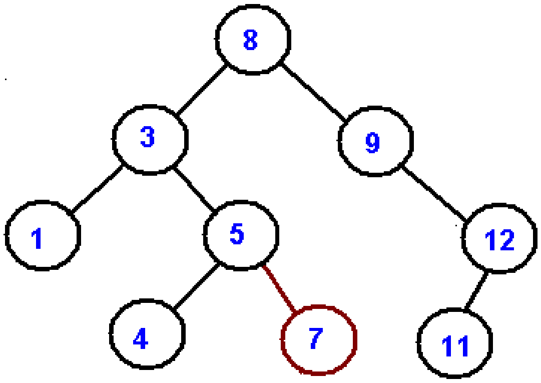
```
public class BST <AnyType extends Comparable<AnyType>>
{
  private Node<AnyType> root;


  private class Node<AnyType>
  {
    private AnyType data;
    private Node<AnyType> left, right;

    public Node(AnyType data)
    {
      left = right = null;
      this.data = data;
    }
  }
  ...

}
```

Insertion

The insertion procedure is quite similar to searching. We start at the root and recursively go down the tree searching for a location in a BST to insert a new node. If the element to be inserted is already in the tree, we are done (we do not insert duplicates). The new node will always replace a NULL reference.



before insertion

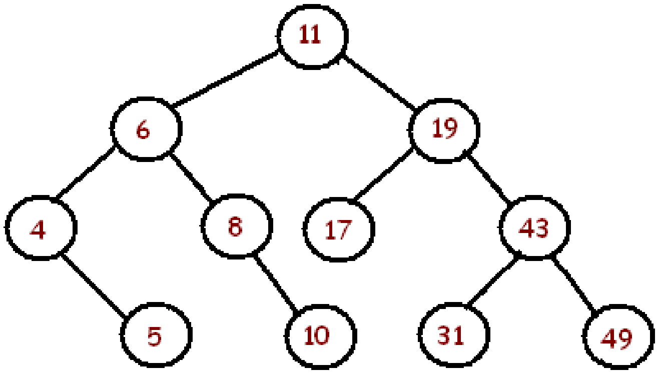


after insertion

Exercise. Given a sequence of numbers:

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

Draw a binary search tree by inserting the above numbers from left to right.



Searching

Searching in a BST always starts at the root. We compare a data stored at the root with the key we are searching for (let us call it as toSearch). If the node does not contain the key we proceed either to the left or right child depending upon comparison. If the result of comparison is negative we go to the left child, otherwise - to the right child. The recursive structure of a BST yields a recursive algorithm.

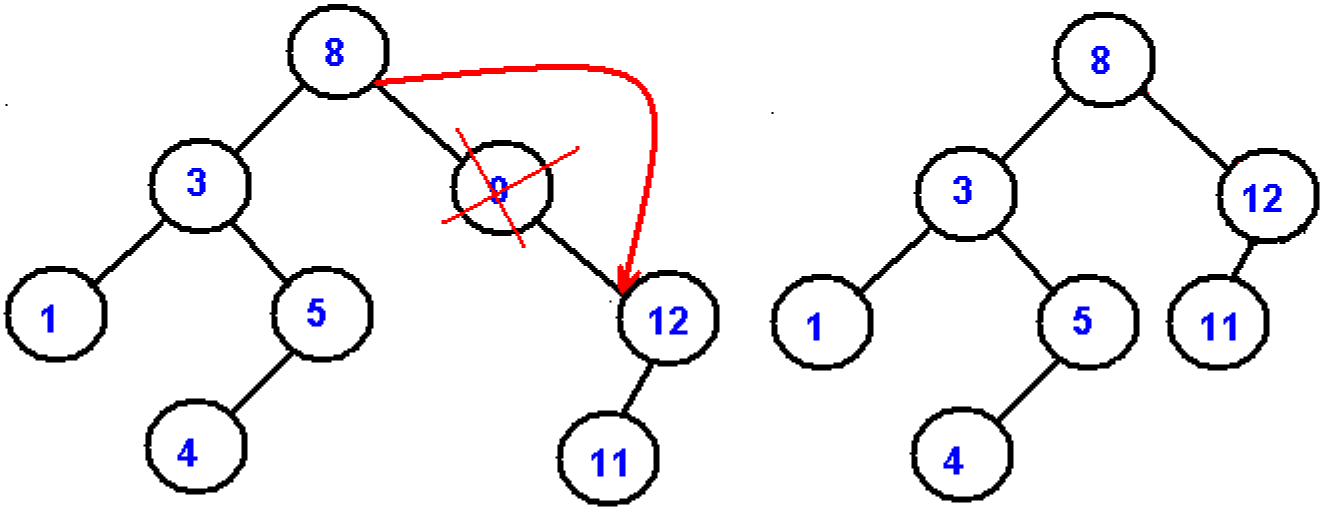
Searching in a BST has O(h) worst-case runtime complexity, where h is the height of the tree. Since s binary search tree with n nodes has a minimum of O(log n) levels, it takes at least O(log n) comparisons to find a particular node. Unfortunately, a binary serch tree can degenerate to a linked list, reducing the search time to O(n).

Deletion

Deletion is somewhat more tricky than insertion. There are several cases to consider. A node to be deleted (let us call it as toDelete)

- is not in a tree;
- is a leaf;
- has only one child;
- has two children.

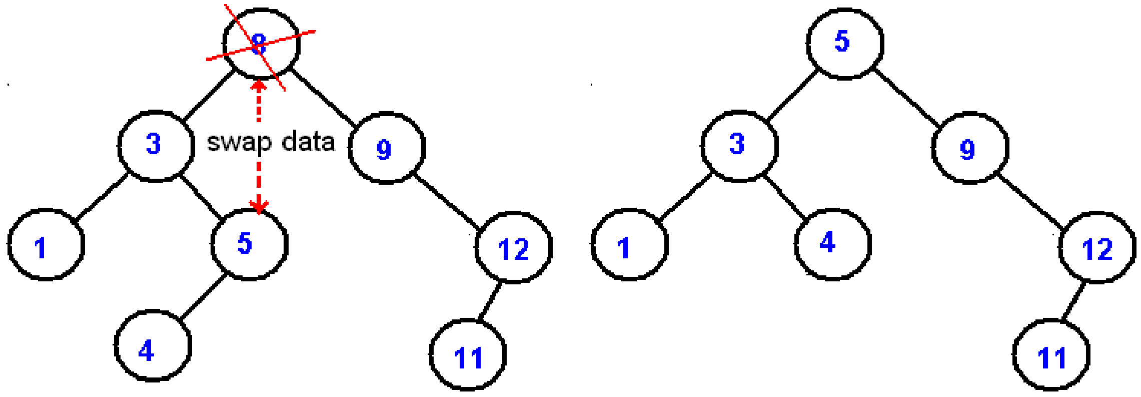
If toDelete is not in the tree, there is nothing to delete. If toDelete node has only one child the procedure of deletion is identical to deleting a node from a linked list - we just bypass that node being deleted



before deletion

after deletion

Deletion of an internal node with two children is less straightforward. If we delete such a node, we split a tree into two subtrees and therefore, some children of the internal node won't be accessible after deletion. In the picture below we delete 8:



before deletion

after deletion

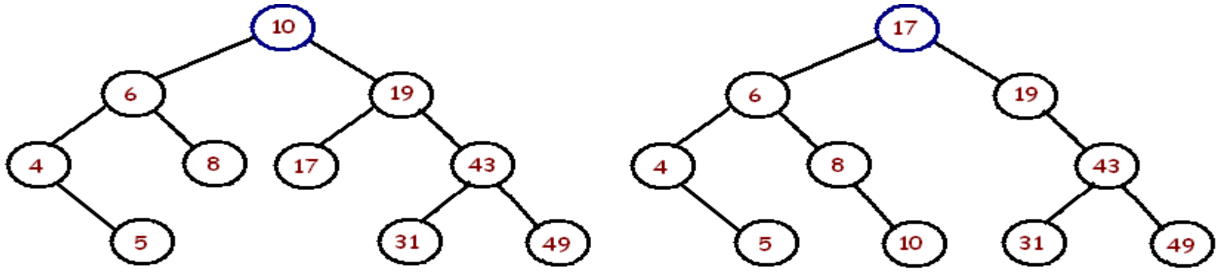
Deletion starategy is the following: replace the node being deleted with the largest node in the left subtree and then delete that largest node. By symmetry, the node being deleted can be swapped with the smallest node is the right subtree.

See BST.java for a complete implementation.

Exercise. Given a sequence of numbers:

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

Draw a binary search tree by inserting the above numbers from left to right and then show the two trees that can be the result after the removal of 11.



Non-Recursive Traversals

Depth-first traversals can be easily implemented recursively.A non-recursive implementation is a bit more difficult. In this section we implement a pre-order traversal as a tree iterator

```
public Iterator<AnyType> iterator()
{
  return new PreOrderIterator();
}
```

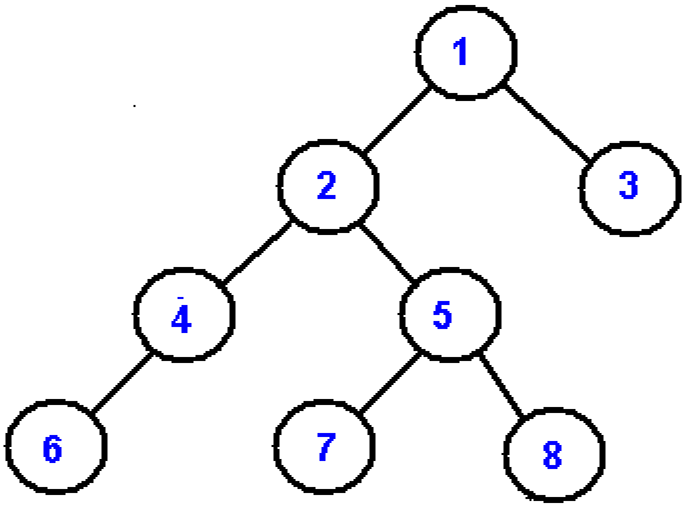
where the PreOrderIterator class is implemented as an inner private class of the BST class

```
private class PreOrderIterator implements Iterator<AnyType>
{
  ...
}
```

The main difficulty is with next() method, which requires the implicit recursive stack implemented explicitly. We will be using Java's Stack. The algorithm starts with the root and push it on a stack. When a user calls for the next() method, we check if the top element has a left child. If it has a left child, we push that child on a stack and return a parent node. If there is no a left child, we check for a right child. If it has a right child, we push that child on a stack and return a parent node. If there is no right child, we move back up the tree (by popping up elements from a stack) until we find a node with a right child. Here is thenext() implementation

```
public AnyType next()
{
  Node cur = stk.peek();
  if(cur.left != null)
  {
    stk.push(cur.left);
  }
  else
  {
    Node tmp = stk.pop();
    while(tmp.right == null)
    {
      if (stk.isEmpty()) return cur.data;
      tmp = stk.pop();
    }
    stk.push(tmp.right);
  }
  return cur.data;
}
```

The following example.shows the output and the state of the stack during each call to next(). Note, the algorithm works on any binary trees, not necessarily binary search trees..



Output		1	2	4	6	5	7	8	3
Stack	1	2 1	4 2 1	6 4 2 1	5 1	7 5 1	8 1	3	

A non-recursive preorder traversal can be eloquently implemented in just three lines of code. If you understand next()'s implementation above, it should be no problem to grasp this one:

```
public AnyType next()
{
   if (stk.isEmpty()) throw new java.util.NoSuchElementException();

   Node cur = stk.pop();
   if(cur.right != null) stk.push(cur.right);
   if(cur.left != null) stk.push(cur.left);
```

```
  return cur.data;
}
```

Note, we push the right child before the left child.

4.4 APPLICATION OF TREES

Union-Find Structure

- Used to store disjoint sets
- Can support two types of operations efficiently

– Find(x): returns the "representative" of the set that x belongs

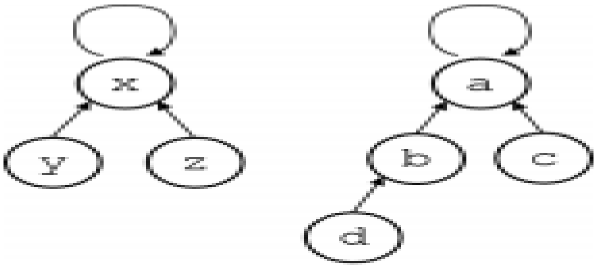
– Union(x, y): merges two sets that contain x and y

Main idea: represent each set by a rooted tree

– Every node maintains a link to its parent

– A root node is the "representative" of the corresponding set

– Example: two sets {x, y, z} and {a, b, c, d}



Find(x): follow the links from x until a node points itself

– This can take O(n) time but we will make it faster

Δ Union(x, y): run Find(x) and Find(y) to find

corresponding root nodes and direct one to the other

```
int Find(int x) {
while(x != L[x]) x = L[x];
return x;
}
void Union(int x, int y) {
```

```
L[Find(x)] = Find(y);
}
```

4.5 GRAPH

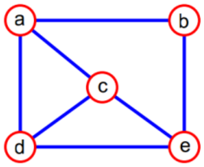
What is a Graph?

- A graph G = (V,E) is composed of:

  V: set of vertices

  E: set of edges connecting the vertices in V

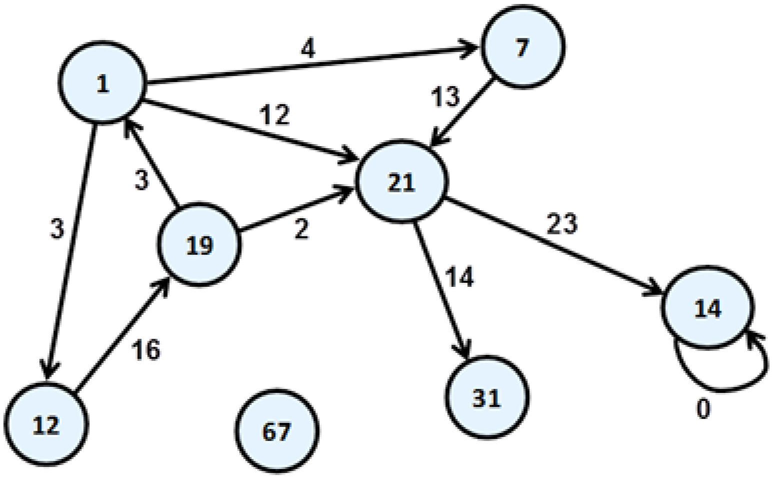
- An edge e = (u,v) is a pair of vertices



Example:

V= {a,b,c,d,e}

E={(a,b),(a,c),(a,d),(b,e),(c,d),(c,e),(d,e)}



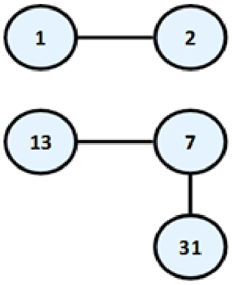
Path in a graph is a sequence of vertices v_1, v_2, …, v_n,, such as there is an edge from v_i to v_{i+1} for every i from 1 to n-1. In our example path is the sequence "1", "12", "19", "21". "7", "21" and "1" is not a path because there is no edge starting from "21" and ending in "1". **Length of path** is the number of edges connecting vertices in the sequence of the vertices in the path. This number is equal to the number of vertices in the path minus one. The length of our example for path "1", "12", "19", "21" is three.

Cost of path in a weighted graph, we call the sum of the weights (costs) of the edges involved in the path. In real life the road from Sofia to Madrid, for example, is equal to the length of the road from Sofia to Paris plus the length of the road from Madrid to Paris. In our example, the length of the path "1", "12", "19" and "21" is equal to 3 + 16 + 2 = 21.

Loop is a path in which the initial and the final vertex of the path match. Example of vertices forming loop are "1", "12" and "19". In the same time "1", "7" and "21" do not form a loop.

Looping edge we will call an edge, which starts and ends in the same vertex. In our example the vertex "14" is looped.

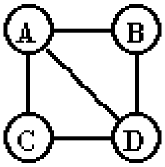
A **connected undirected graph** we call an undirected graph in which there is a path from each node to each other. For example, the following graph is**not connected** because there is no path from "1" to "7".



4.6 REPRESENTATIONS OF GRAPHS

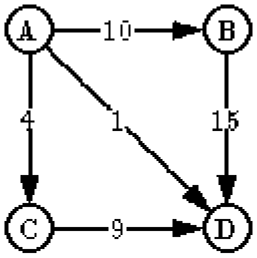
Adjacency Matrices

Graphs $G = (V, E)$ can be represented by **adjacency matrices** $G[v_1..v_{|V|}, v_1..v_{|V|}]$, where the rows and columns are indexed by the nodes, and the entries $G[v_i, v_j]$ represent the edges. In the case of unlabeled graphs, the entries are just boolean values.



$$\begin{array}{ccccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \mathbf{A} & 0 & 1 & 1 & 1 \\ \mathbf{B} & 1 & 0 & 0 & 1 \\ \mathbf{C} & 1 & 0 & 0 & 1 \\ \mathbf{D} & 1 & 1 & 1 & 0 \end{array}$$

In case of labeled graphs, the labels themselves may be introduced into the entries.

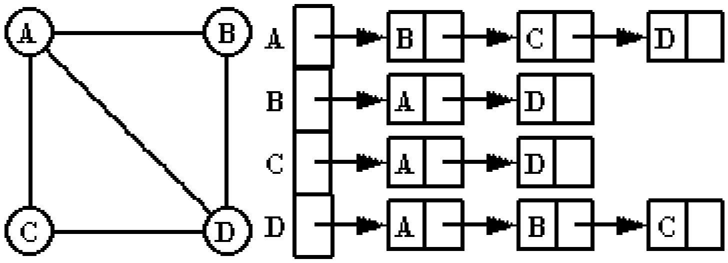


$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \infty & 10 & 4 & 1 \\ B & \infty & \infty & \infty & 15 \\ C & \infty & \infty & \infty & 9 \\ D & \infty & \infty & \infty & \infty \end{array}$$

Adjacency matrices require $O(|V|^2)$ space, and so they are space-efficient only when they are dense (that is, when the graphs have many edges). Time-wise, the adjacency matrices allow easy addition and deletion of edges.

Adjacency Lists

A representation of the graph consisting of a list of nodes, with each node containing a list of its neighboring nodes.



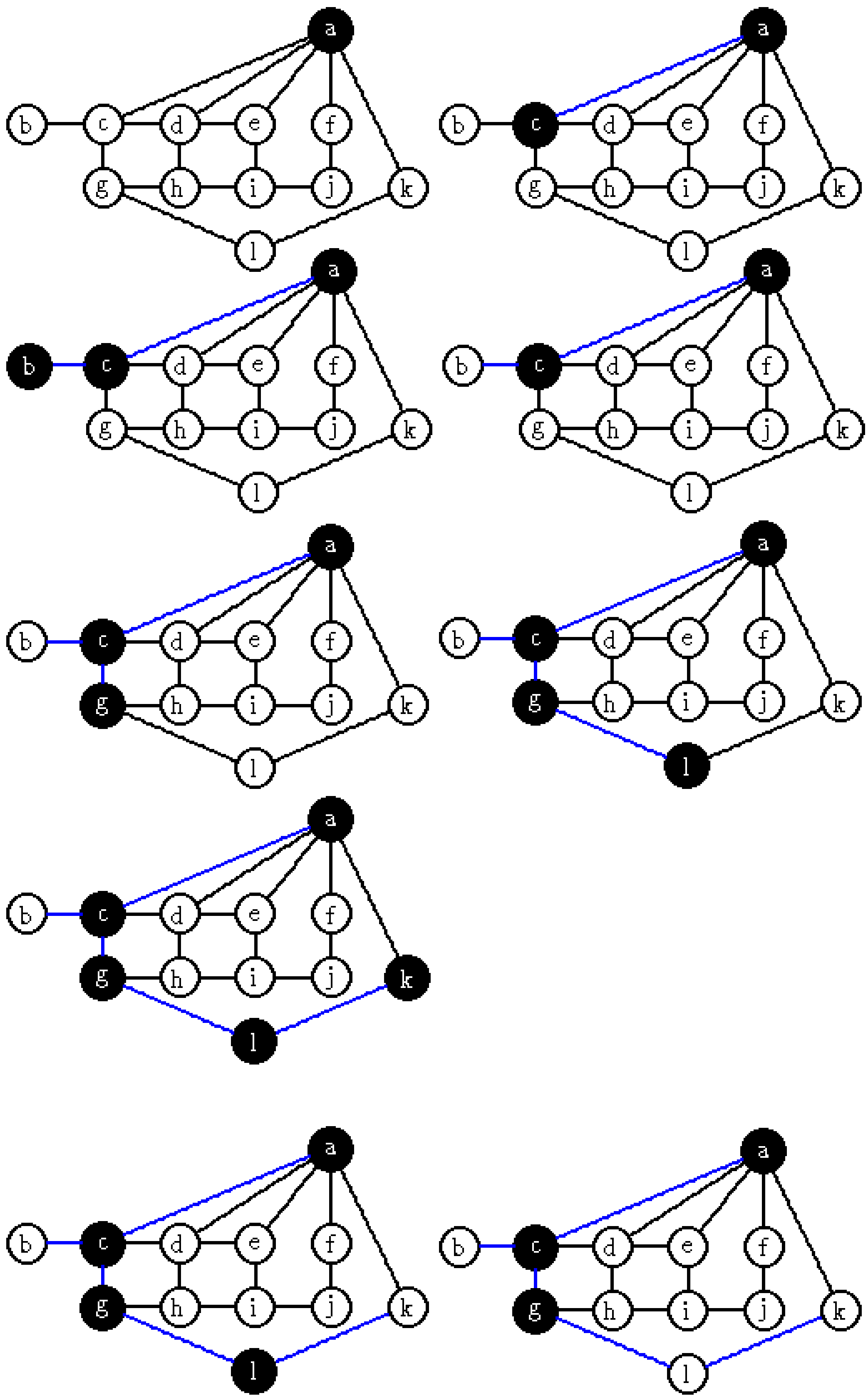
This representation takes $O(|V| + |E|)$ space.

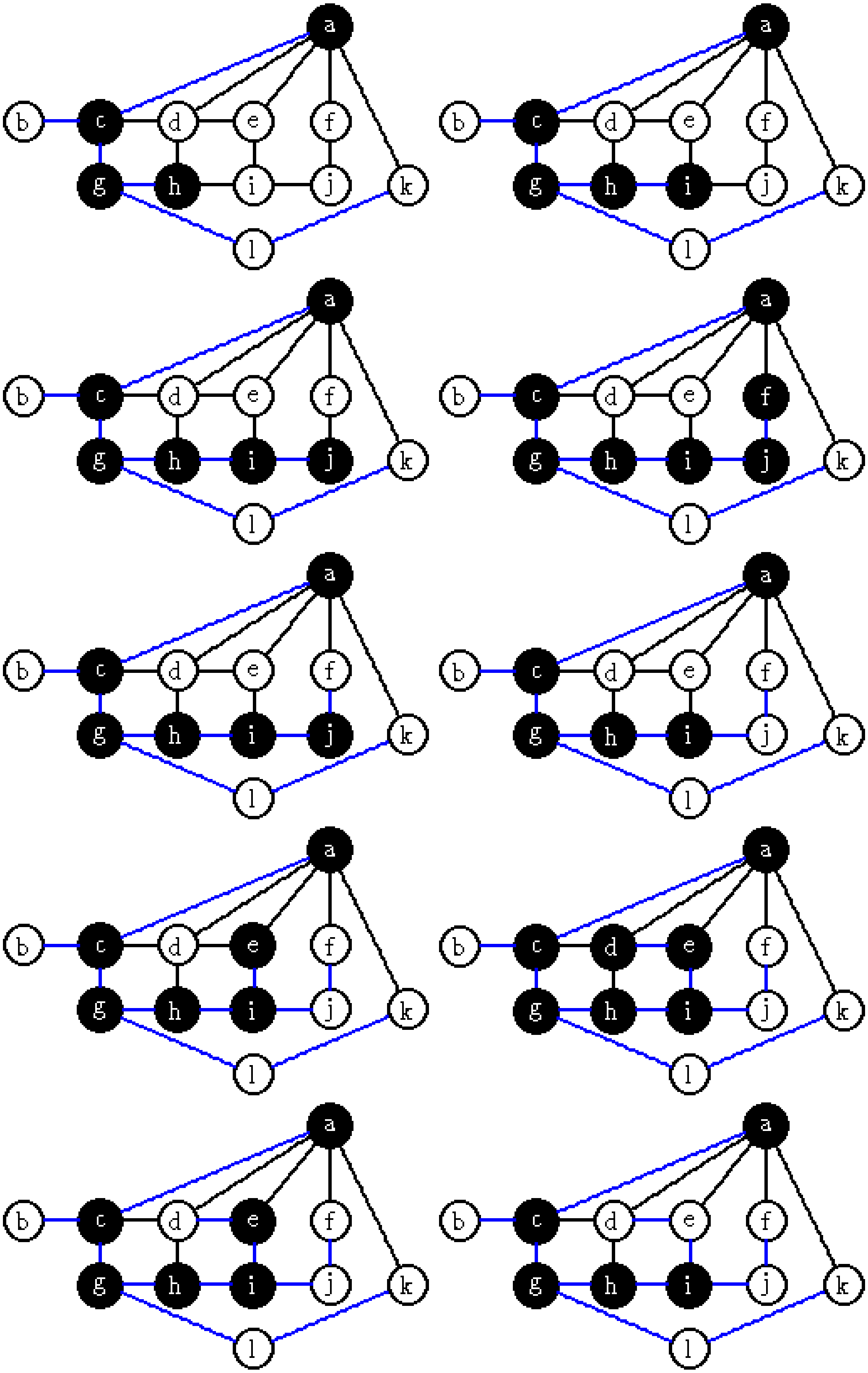
4.8 Depth-First Traversal

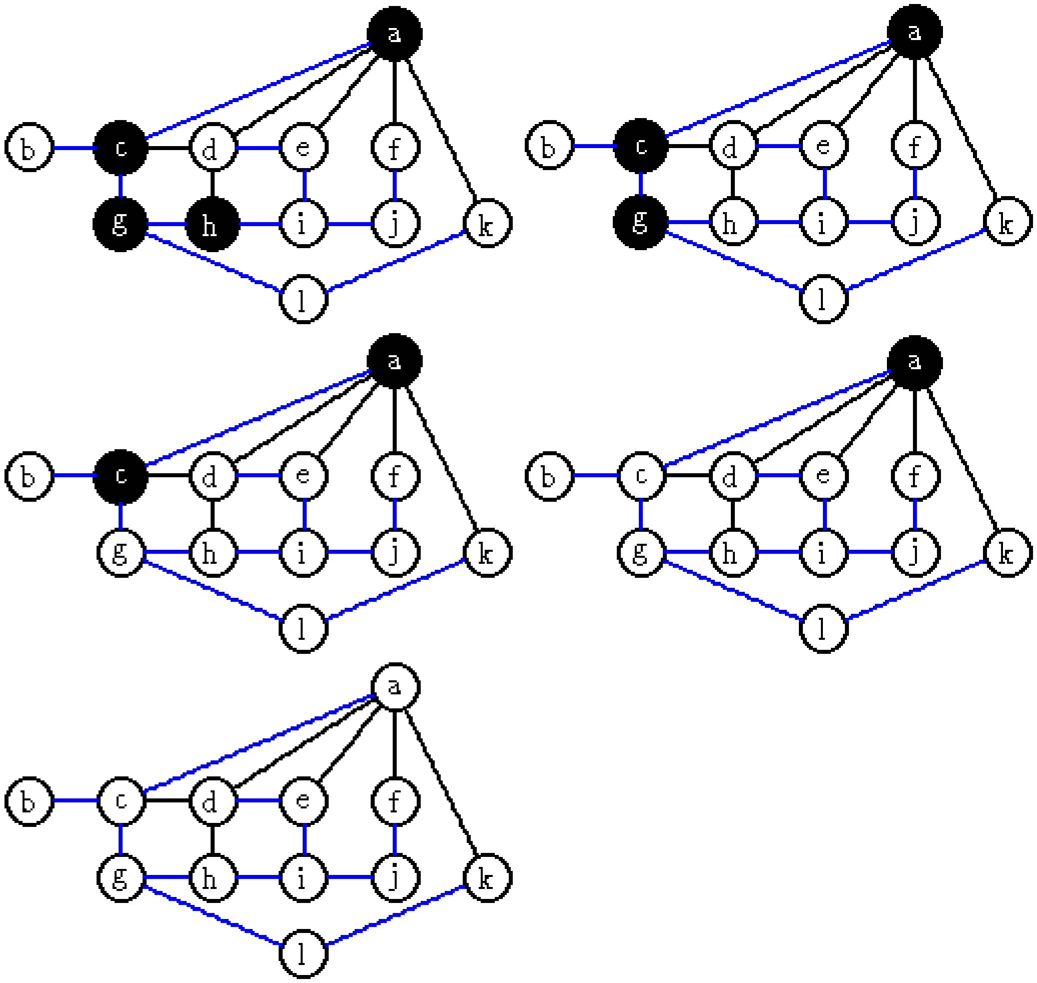
algorithm

```
dft(x)
  visit(x)
  FOR each y such that (x,y) is an edge DO
    IF y was not visited yet THEN
      dft(y)
```

A recursive algorithm implicitly recording a "backtracking" path from the root to the node currently under consideration

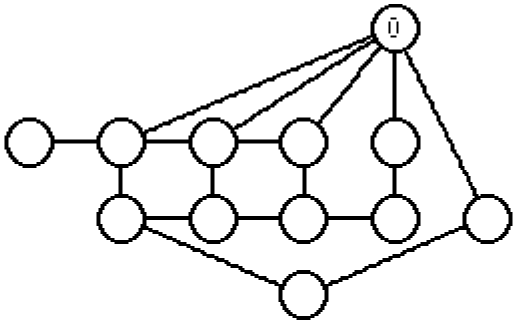

a
b
c
d
e
f
g
h
i
j
k
l

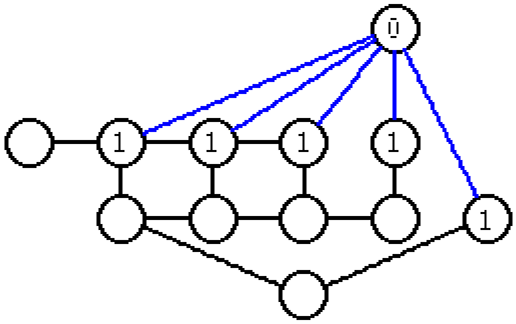

a
b
c
d
e
f
g
h
i
j
k
l

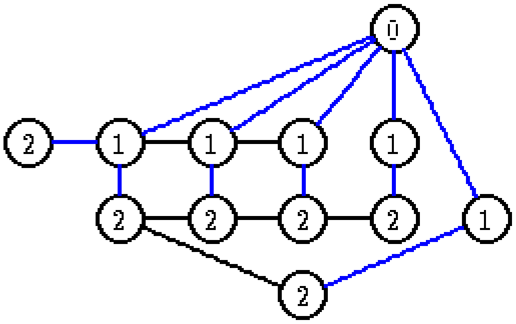


4.7 Breadth-First Traversal

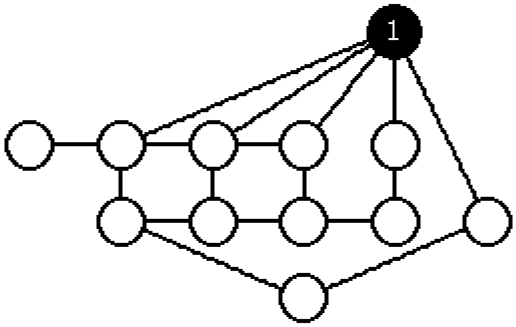
Visit the nodes at level i before the nodes of level i+1.

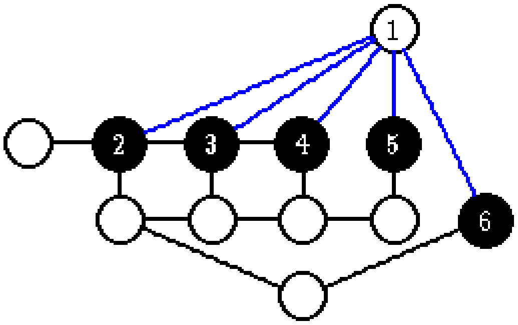


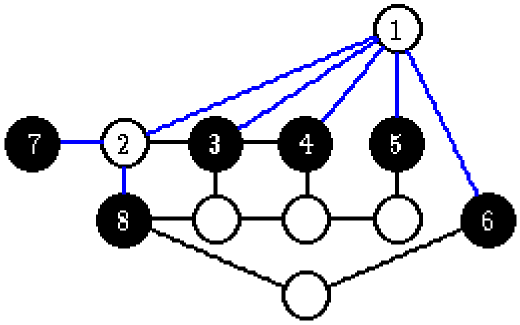


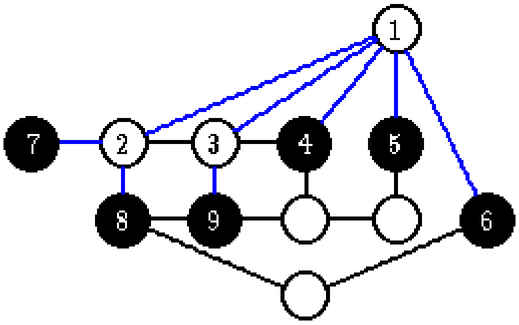


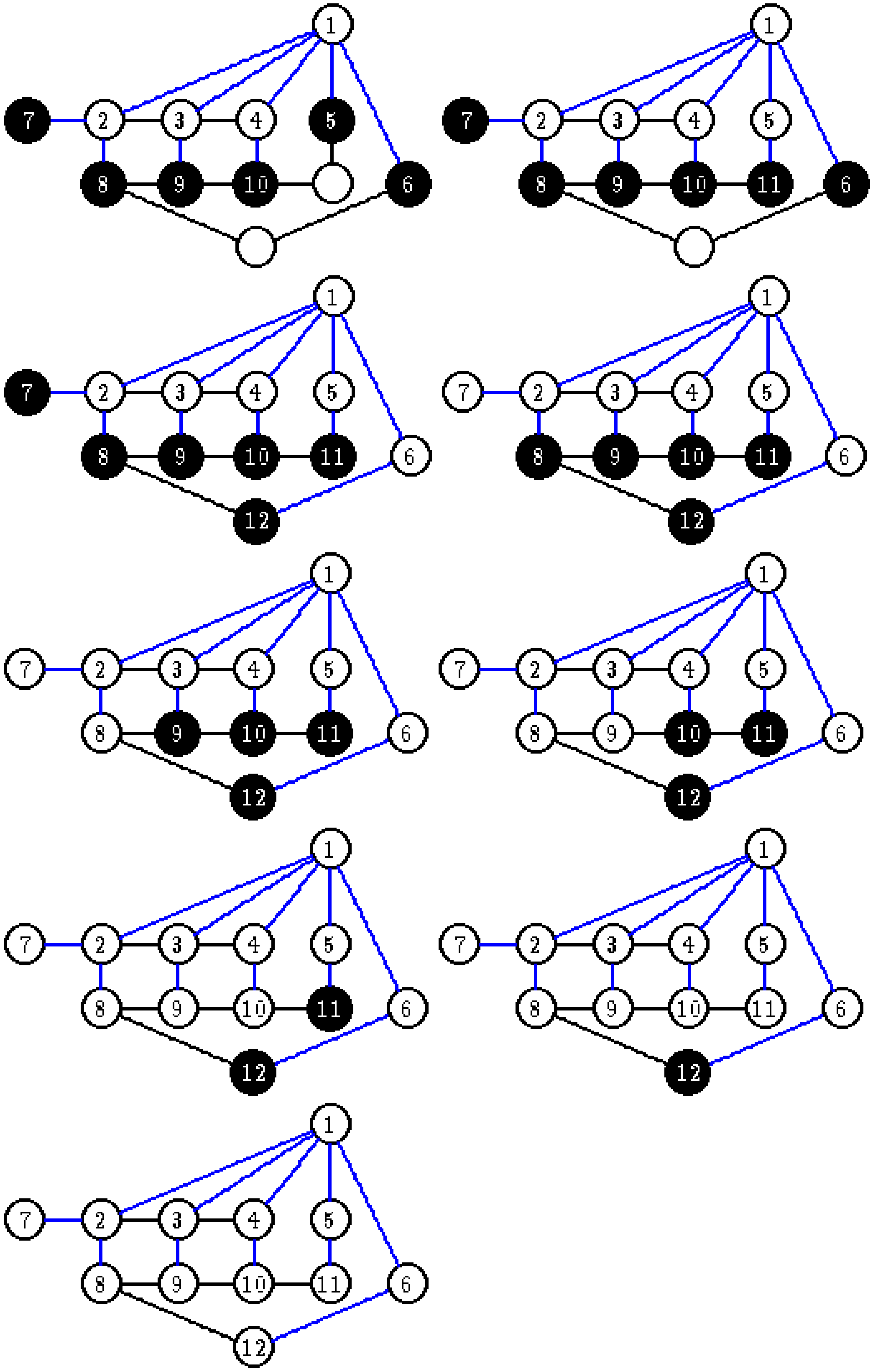
```
visit(start node)
queue <- start node
WHILE queue is nor empty DO
  x <- queue
  FOR each y such that (x,y) is an edge
                   and y has not been visited yet DO
    visit(y)
    queue <- y
  END
END
```









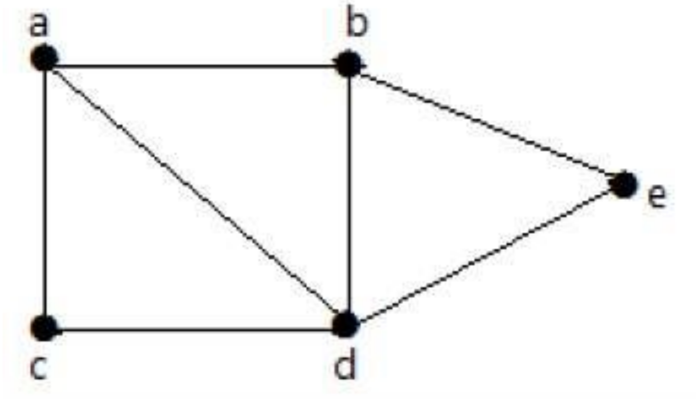


1.9 CONNECTIVITY

A graph is said to be **connected if there is a path between every pair of vertex**. From every vertex to any other vertex, there should be some path to traverse. That is called the connectivity of a graph. A graph with multiple disconnected vertices and edges is said to be disconnected.

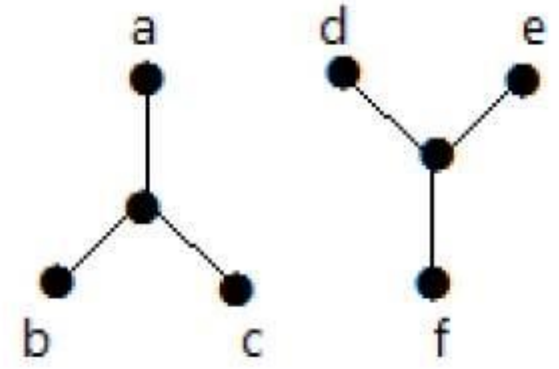
Example 1

In the following graph, it is possible to travel from one vertex to any other vertex. For example, one can traverse from vertex 'a' to vertex 'e' using the path 'a-b-e'.



Example 2

In the following example, traversing from vertex 'a' to vertex 'f' is not possible because there is no path between them directly or indirectly. Hence it is a disconnected graph.



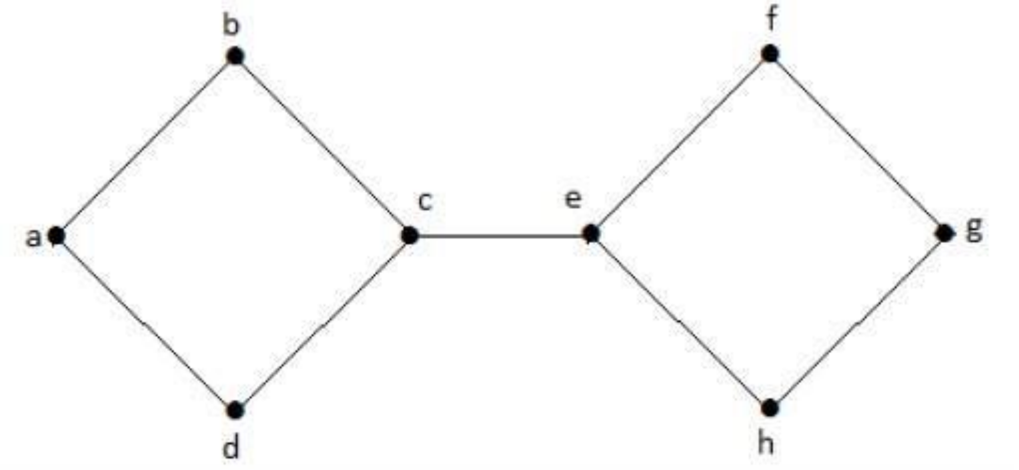
Let ‘G’ be a connected graph. A vertex V ∈ G is called a cut vertex of ‘G’, if ‘G-V’ (Delete ‘V’ from ‘G’) results in a disconnected graph. Removing a cut vertex from a graph breaks it in to two or more graphs.

Note − Removing a cut vertex may render a graph disconnected.

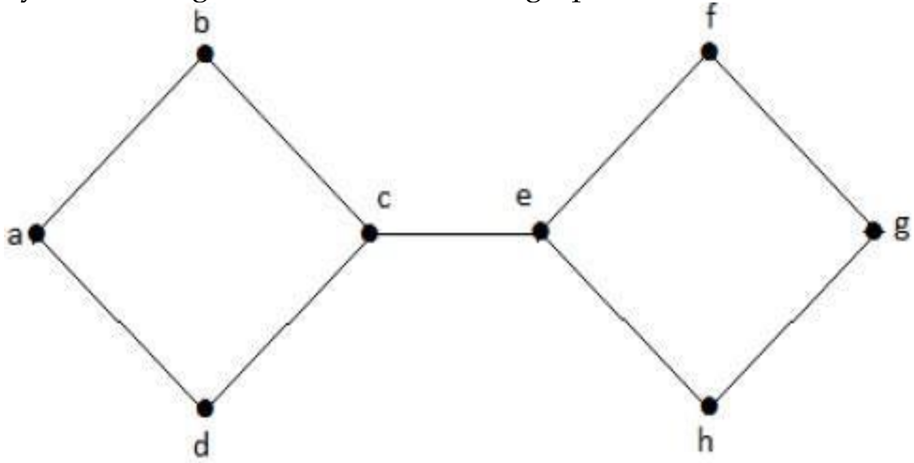
A connected graph ‘G’ may have at most (n–2) cut vertices.

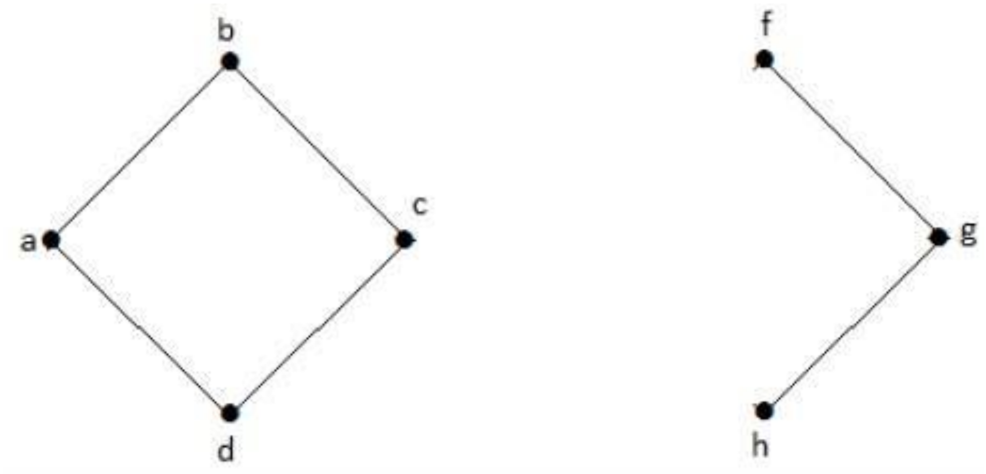
Example

In the following graph, vertices ‘e’ and ‘c’ are the cut vertices.



By removing ‘e’ or ‘c’, the graph will become a disconnected graph.





Without ‘g’, there is no path between vertex ‘c’ and vertex ‘h’ and many other. Hence it is a disconnected graph with cut vertex as ‘e’. Similarly, ‘c’ is also a cut vertex for the above graph.

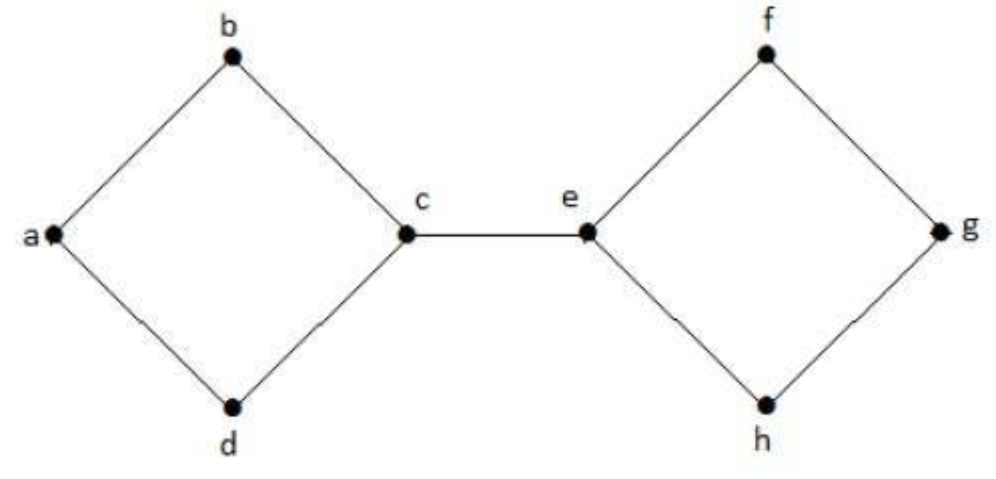
Cut Edge (Bridge)

Let ‘G’ be a connected graph. An edge ‘e’ ∈ G is called a cut edge if ‘G-e’ results in a disconnected graph.

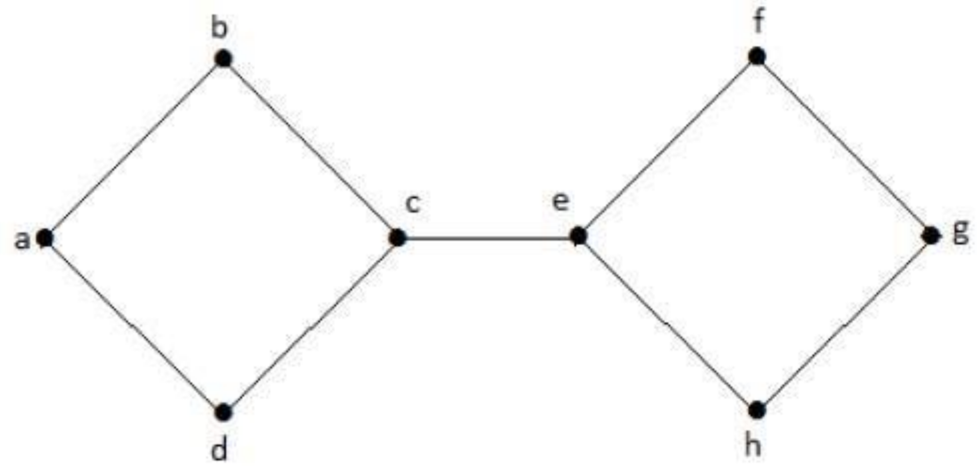
If removing an edge in a graph results in to two or more graphs, then that edge is called a Cut Edge.

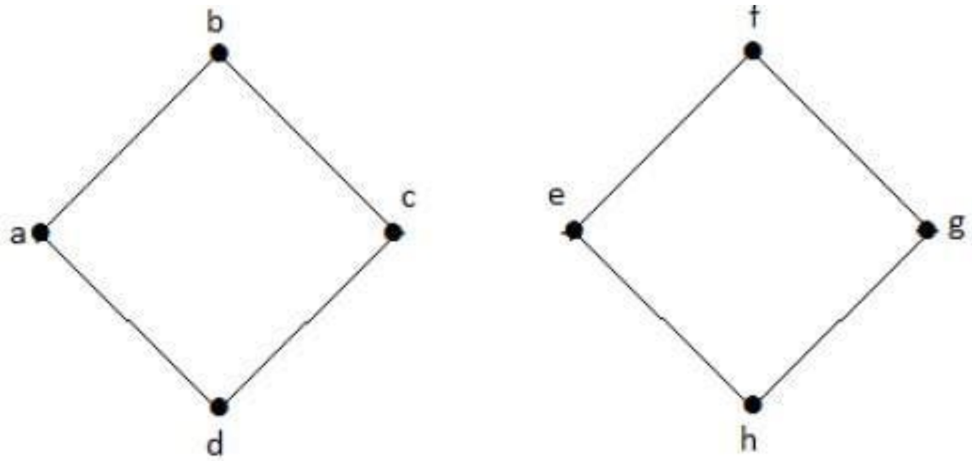
Example

In the following graph, the cut edge is [(c, e)]



By removing the edge (c, e) from the graph, it becomes a disconnected graph.





In the above graph, removing the edge (c, e) breaks the graph into two which is nothing but a disconnected graph. Hence, the edge (c, e) is a cut edge of the graph.

Note − Let 'G' be a connected graph with 'n' vertices, then

- a cut edge e ∈ G if and only if the edge 'e' is not a part of any cycle in G.
- the maximum number of cut edges possible is 'n-1'.
- whenever cut edges exist, cut vertices also exist because at least one vertex of a cut edge is a cut vertex.
- if a cut vertex exists, then a cut edge may or may not exist.

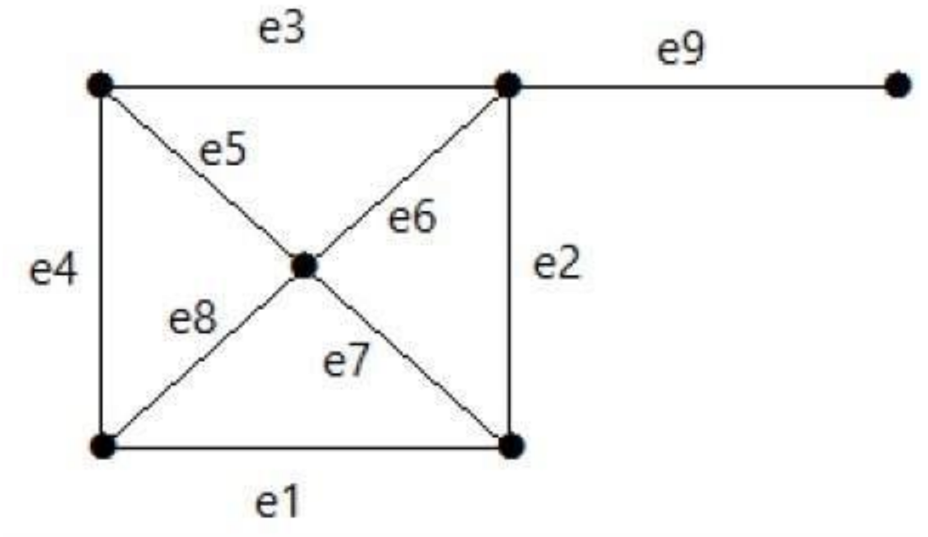
Cut Set of a Graph

Let 'G'= (V, E) be a connected graph. A subset E' of E is called a cut set of G if deletion of all the edges of E' from G makes G disconnect.

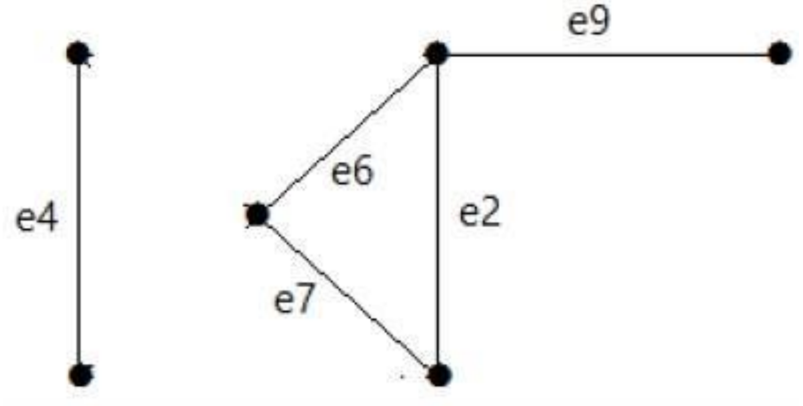
If deleting a certain number of edges from a graph makes it disconnected, then those deleted edges are called the cut set of the graph.

Example

Take a look at the following graph. Its cut set is E1 = {e1, e3, e5, e8}.



After removing the cut set E1 from the graph, it would appear as follows −



Similarly there are other cut sets that can disconnect the graph −

- E3 = {e9} – Smallest cut set of the graph.
- E4 = {e3, e4, e5}

Edge Connectivity

Let 'G' be a connected graph. The minimum number of edges whose removal makes 'G' disconnected is called edge connectivity of G.

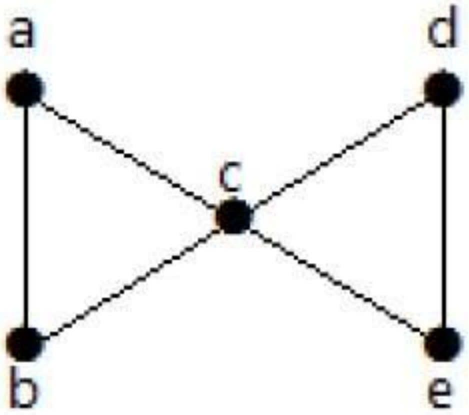
Notation − λ(G)

In other words, the **number of edges in a smallest cut set of G** is called the edge connectivity of G.

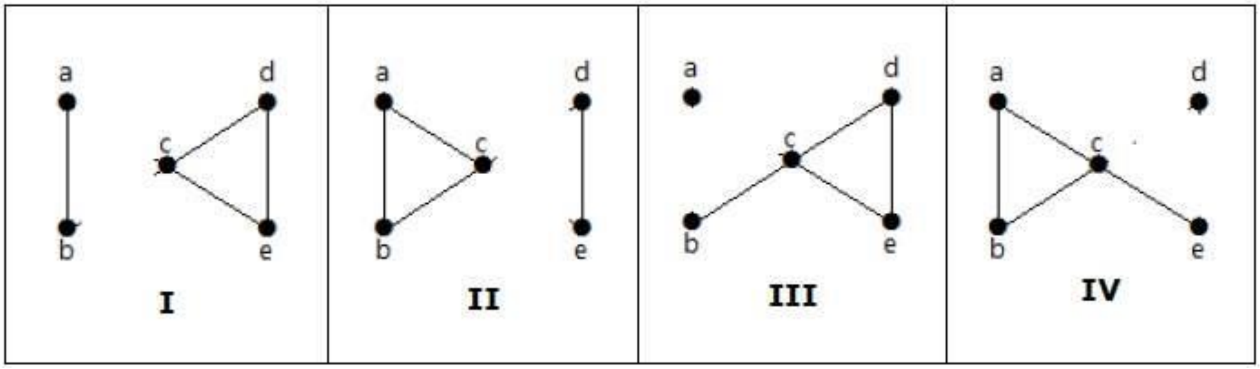
If 'G' has a cut edge, then *λ(G)* is 1. (edge connectivity of G.)

Example

Take a look at the following graph. By removing two minimum edges, the connected graph becomes disconnected. Hence, its edge connectivity (λ(G)) is 2.



Here are the four ways to disconnect the graph by removing two edges −



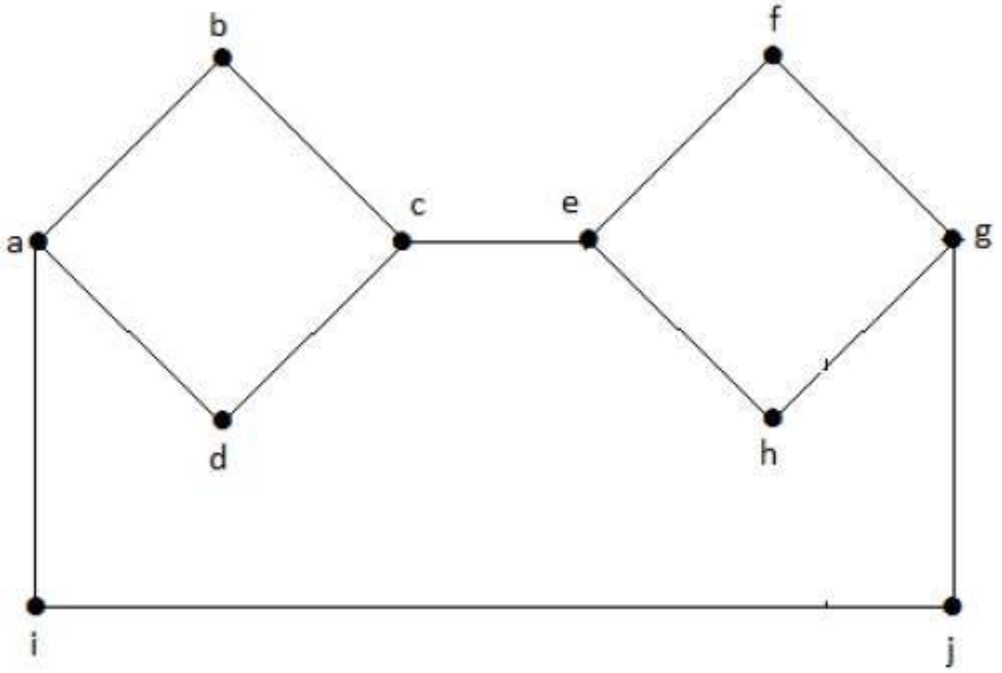
Vertex Connectivity

Let 'G' be a connected graph. The minimum number of vertices whose removal makes 'G' either disconnected or reduces 'G' in to a trivial graph is called its vertex connectivity.

Notation − K(G)

Example

In the above graph, removing the vertices ‘e’ and ‘i’ makes the graph disconnected.



If G has a cut vertex, then K(G) = 1.

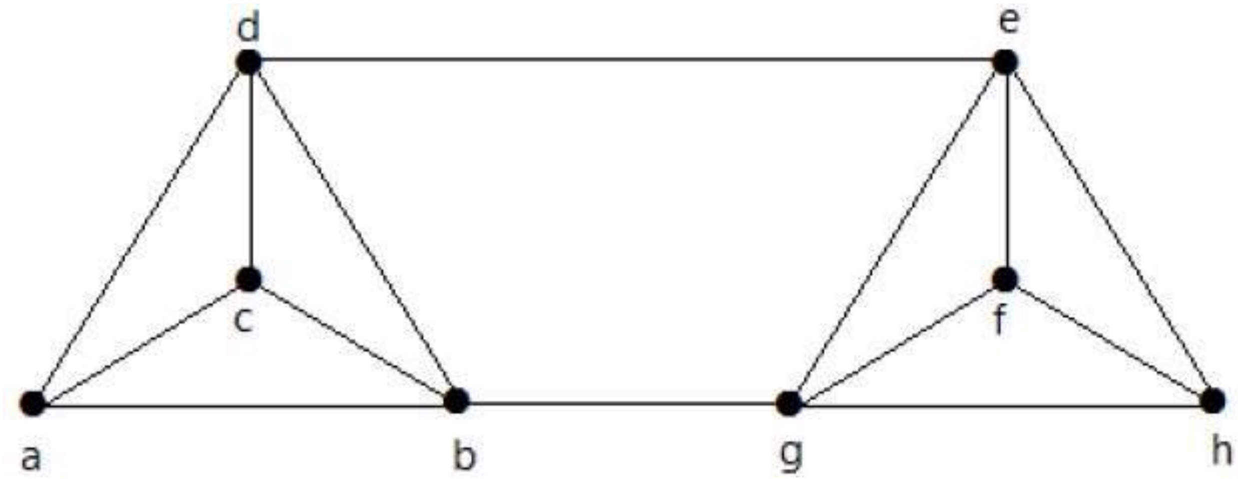
Notation − For any connected graph G,

K(G) ≤ λ(G) ≤ δ(G)

Vertex connectivity (K(G)), edge connectivity (λ(G)), minimum number of degrees of G(δ(G)).

Example

Calculate λ(G) and K(G) for the following graph −



Solution

From the graph,

δ(G) = 3

K(G) ≤ λ(G) ≤ δ(G) = 3 (1)

K(G) ≥ 2 (2)

Deleting the edges {d, e} and {b, h}, we can disconnect G.

Therefore,

λ(G) = 2

2 ≤ λ(G) ≤ δ(G) = 2 (3)

From (2) and (3), vertex connectivity K(G) = 2

UNIT V SORTING & SEARCHING

Sorting is a technique to rearrange the elements of a list in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. Sorting is a process through which the data is arranged in ascending or descending order.

Sorting can be classified in two types:

Internal Sorts:-This method uses only the primary memory during sorting process. All data items are heldin main memory and no secondary memory is required this sorting process. If all the data that is to be sorted can be accommodated at a time in memory is called internal sorting. There is a limitation for internal sorts; they can only process relatively small lists due to memory constraints.

(i) SELECTION SORT :-Ex:-Selection sort algorithm, Heap Sort algorithm

(ii) INSERTION SORT :-Ex:-Insertion sort algorithm, Shell Sort algorithm

(iii) EXCHANGE SORT :-Ex:-Bubble Sort Algorithm, Quick sort algorithm

External Sorts:-Sorting large amount of data requires external or secondary memory. This process uses external memory such as HDD, to store the data which is not fit into the main memory. So, primary memory holds the currently being sorted data only. All external sorts are based on process of merging. Different parts of data are sorted separately and merged together.

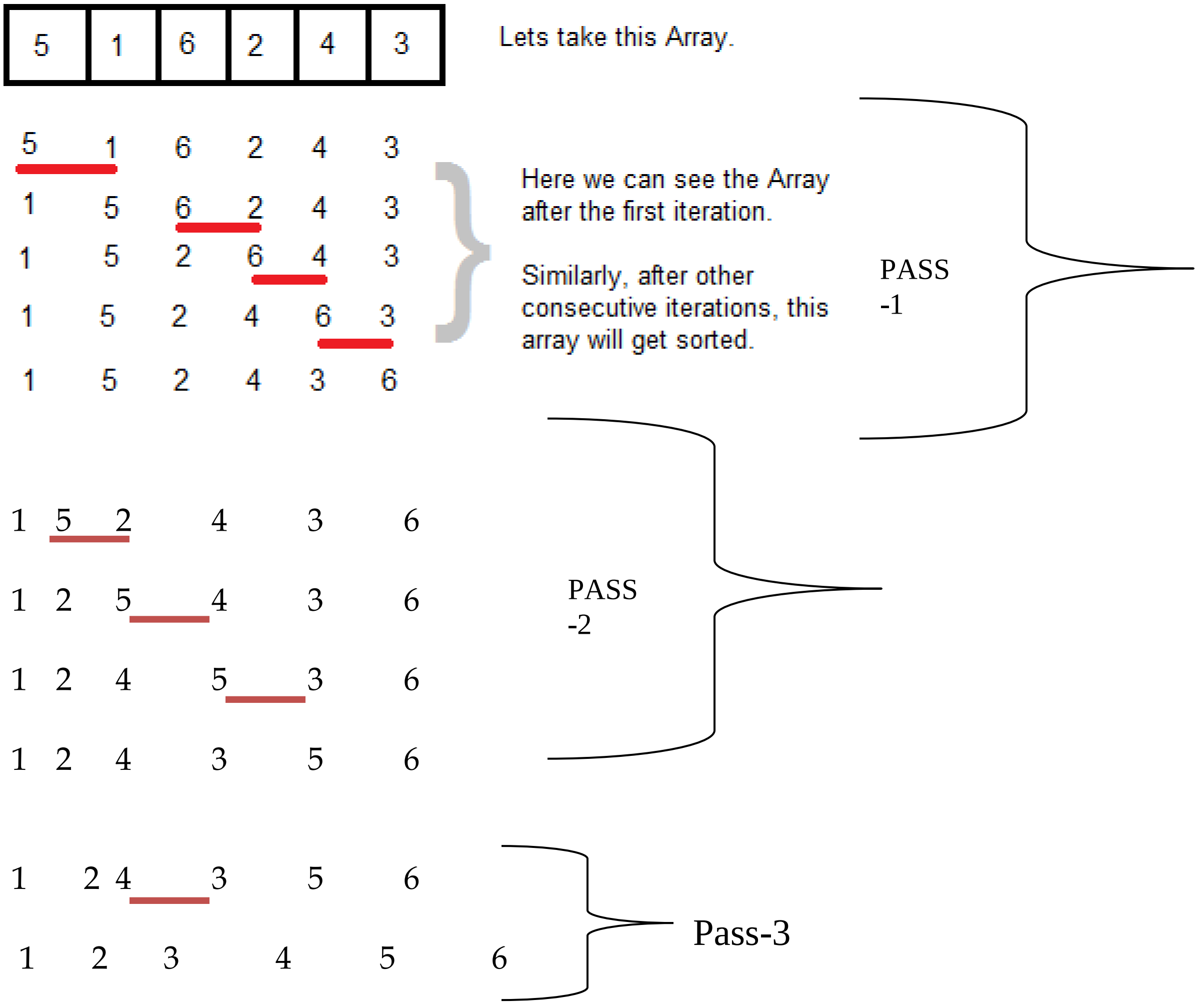
BUBBLE SORT

Bubble Sort is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.

Algorithm Of Bubble Sort

1. Each two adjacent elements are compared

2. Swap with lighter elements
3. Move forward and swap with each lighter item
4. If there is a heavier element, then this item begins to bubble to the surface.
5. Finally the Heaviest Element is on its place.



5 1 6 2 4 3

Lets take this Array.

5 1 6 2 4 3
1 5 6 2 4 3
1 5 2 6 4 3
1 5 2 4 6 3
1 5 2 4 3 6

Here we can see the Array after the first iteration.

Similarly, after other consecutive iterations, this array will get sorted.

1 5 2 4 3 6
1 2 5 4 3 6
1 2 4 5 3 6
1 2 4 3 5 6

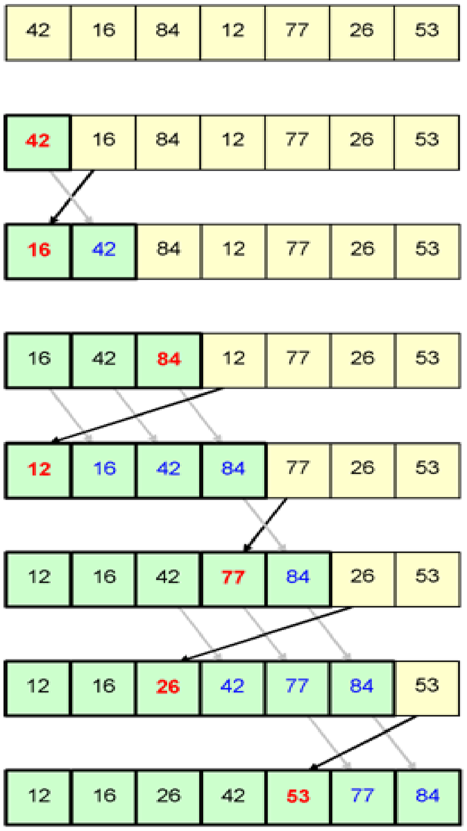
1 2 4 3 5 6
1 2 3 4 5 6

IMPLEMENTATION

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

//Bubble Sort
void bubble_sort (int arr[], int n)
 {
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n - i - 1; ++j)
      if (arr[j] > arr[j + 1])
     {
       int temp = arr[j];
    arr[j] = arr[j + 1];
    arr[j + 1] = temp;
   }
 }

//Driver Function
int main()
{
 int input_ar[] = {10, 50, 21, 2, 6, 66, 802, 75, 24, 170};
 int n = sizeof (input_ar) / sizeof (input_ar[0]);
 bubble_sort (input_ar, n);
 cout << "Sorted Array : " << endl;
 for (int i = 0; i < n; ++i)
  cout << input_ar[i] << " ";
 return 0;
}
```

Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.



The array, before the insertion sort operation begins.

The first element (**42**) is taken as the start of the sorted subsection.

16 is introduced to the subsection. 42 must move to make room to keep everything in sort order.

84 is introduced to the subsection. No shifting is necessary to preserve sort order.

12 is introduced to the subsection. 16, 42 and 84 must be shifted across to make room, to keep sort order.

77 is introduced to the subsection. 84 must be shifted across to make room, to keep sort order.

26 is introduced to the subsection. 42, 77 and 84 must be shifted across to make room, to keep sort order.

53 is introduced to the subsection. 77 and 84 must be shifted across to make room, to keep sort order.

IMPLEMENTATION

```
#include <iostream>
#include <conio.h>
using namespace std;
int main()
{
   int a[16], i, j, k, temp;
   cout<<"enter the elements\n";
   for (i = 0; i < 16; i++)
   {
      cin>>a[i];
   }
   for (i = 1; i < 16; i++)
   {
      for (j = i; j >= 1; j--)
      {
         if (a[j] < a[j-1])
         {
            temp = a[j];
            a[j] = a[j-1];
            a[j-1] = temp;
      }
      else
        break;
    }
  }
  cout<<"sorted array\n"<<endl;
  for (k = 0; k < 16; k++)
  {
           cout<<a[k]<<endl;
  }
  getch();
}
```

The Selection Sort

The selection sort algorithm works based on the idea of successive selection: it will select and then organise pieces of data in such a way that they are placed in order.

The selection sort cannot be implemented for a linked list very easily (it is, however, possible); the process will be explained using an array structure.

Basic Concept

The first step is for it to find the 'smallest' piece of data (in terms of the sort criteria, assuming we are sorting from smallest to largest), and swaps it with the first element of the array.

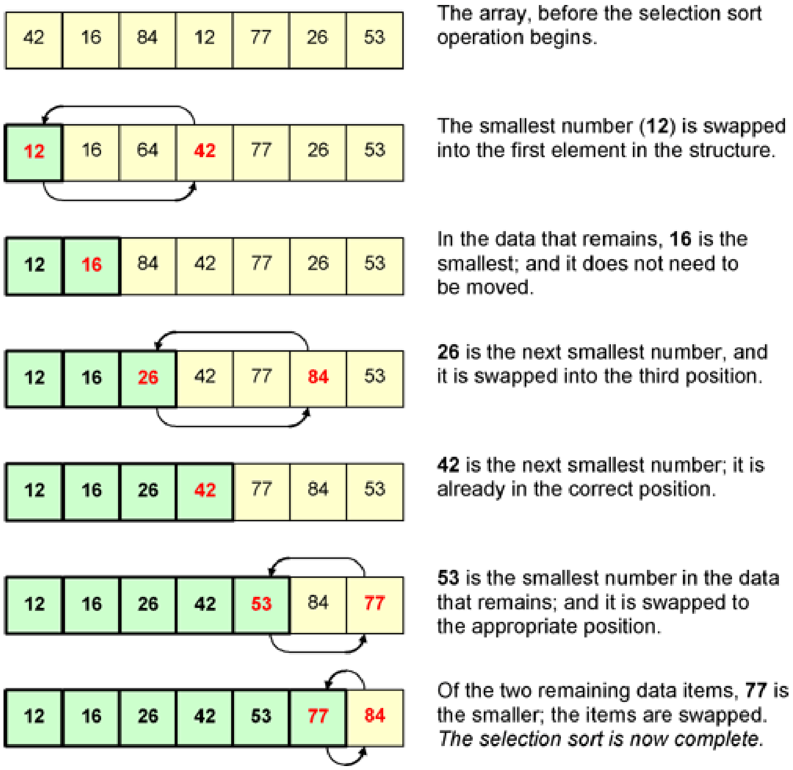
This process is then repeated; it finds the smallest piece of data in what's left over, and swaps that into the second element of the array. This process is repeated until the second-to-last data item has been selected and swapped

Algorithm

The algorithm for an array selection sort has two loops - one that controls which element is being swapped into (and runs from the first to second-last element), and another that finds the smallest data in the subsection.

- loop i from 0 to the second-last element
    - set smallest to i
    - loop j from i to the last element
        - if array[j] < array[smallest], set smallest to j
    - swap array[i] and array[smallest]

In each iteration of the outer loop, the smallest data is deduced from the inner loop, and the smallest data is swapped into the swapping element.



The array, before the selection sort operation begins.

The smallest number (**12**) is swapped into the first element in the structure.

In the data that remains, **16** is the smallest; and it does not need to be moved.

26 is the next smallest number, and it is swapped into the third position.

42 is the next smallest number; it is already in the correct position.

53 is the smallest number in the data that remains; and it is swapped to the appropriate position.

Of the two remaining data items, **77** is the smaller; the items are swapped. *The selection sort is now complete.*

IMPLEMENTATION

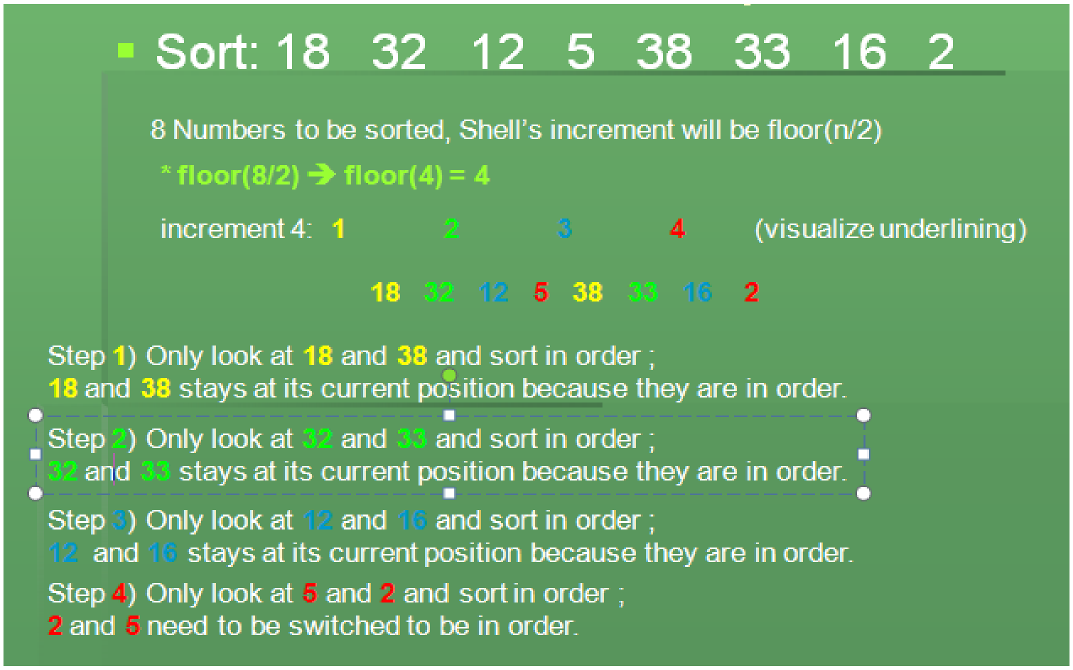
```
#include <iostream>
using namespace std;
void print (int [], int);
void selection_sort (int [], int);
 //Driver Function
int main ()
{
  int min_ele_loc;
  int ar[] = {10, 2, 45, 18, 16, 30, 29, 1, 1, 100};
  cout << "Array initially : ";
  print (ar, 10);
  selection_sort (ar, 10);
  cout << "Array after selection sort : ";
  print (ar, 10);
  return 0;
}
 // Selection Sort
void selection_sort (int ar[], int size)
{
  int min_ele_loc;
  for (int i = 0; i < 9; ++i)
  {
    //Find minimum element in the unsorted array
    //Assume it's the first element
    min_ele_loc = i;

    //Loop through the array to find it
    for (int j = i + 1; j < 10; ++j)
    {
      if (ar[j] < ar[min_ele_loc])
      {
        //Found new minimum position, if present
        min_ele_loc = j;
      }
    }

    //Swap the values
    swap (ar[i], ar[min_ele_loc]);
  }
}

//Print the array
void print (int temp_ar[], int size)
{
  for (int i = 0; i < size; ++i)
  {
    cout << temp_ar[i] << " ";
  }
  cout << endl;
}
```

SHELL SORT

The shell sort compares elements that are a certain distance away (d positions away) from each other and it compares these elements repeatedly (bubble sort only compares adjacent elements.) It uses the equation d = (n + 1) / 2.

- Shellsort uses a sequence h_1, h_2, …, h_t called the ***increment sequence***. Any increment sequence is fine as long as $h_1 = 1$ and some other choices are better than others.

- Shellsort makes multiple passes through a list and sorts a number of equally sized sets using the insertion sort.
- Shellsort is also known as ***diminishing increment sort***.



- Sort: 18 32 12 5 38 33 16 2

8 Numbers to be sorted, Shell's increment will be floor(n/2)

* floor(8/2) ➔ floor(4) = 4

increment 4: 1 2 3 4 (visualize underlining)

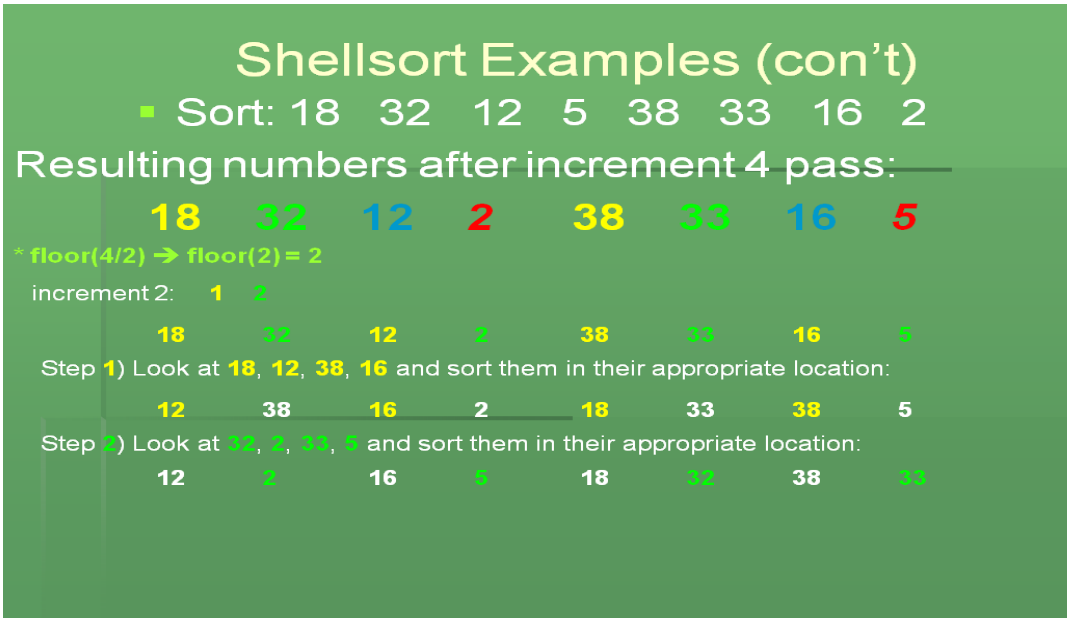
18 32 12 5 38 33 16 2

Step 1) Only look at 18 and 38 and sort in order ;
18 and 38 stays at its current position because they are in order.

Step 2) Only look at 32 and 33 and sort in order ;
32 and 33 stays at its current position because they are in order.

Step 3) Only look at 12 and 16 and sort in order ;
12 and 16 stays at its current position because they are in order.

Step 4) Only look at 5 and 2 and sort in order ;
2 and 5 need to be switched to be in order.



Shellsort Examples (con’t)

- Sort: 18 32 12 5 38 33 16 2

Resulting numbers after increment 4 pass:

18 32 12 2 38 33 16 5

* floor(4/2) → floor(2) = 2

increment 2: 1 2

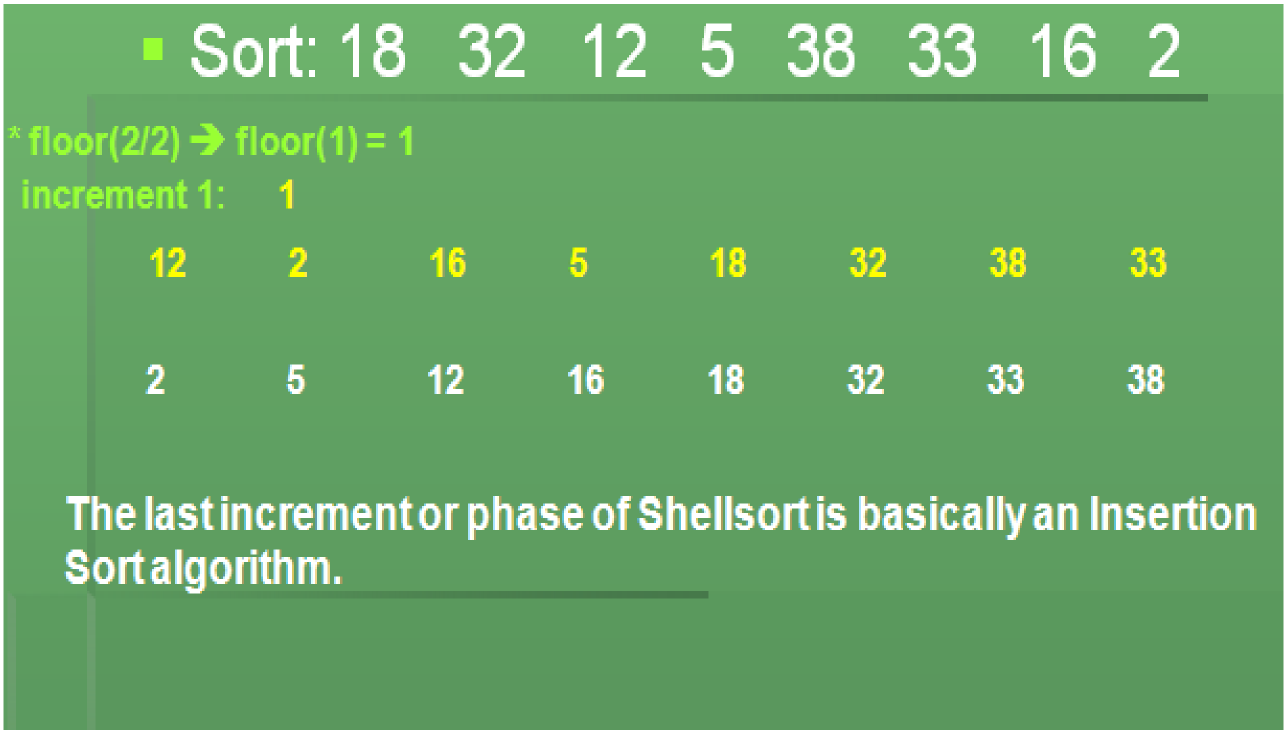
18 32 12 2 38 33 16 5

Step 1) Look at 18, 12, 38, 16 and sort them in their appropriate location:

12 38 16 2 18 33 38 5

Step 2) Look at 32, 2, 33, 5 and sort them in their appropriate location:

12 2 16 5 18 32 38 33



- Sort: 18 32 12 5 38 33 16 2

* floor(2/2) → floor(1) = 1

increment 1: 1

12 2 16 5 18 32 38 33

2 5 12 16 18 32 33 38

The last increment or phase of Shellsort is basically an Insertion Sort algorithm.

IMPLEMENTATION

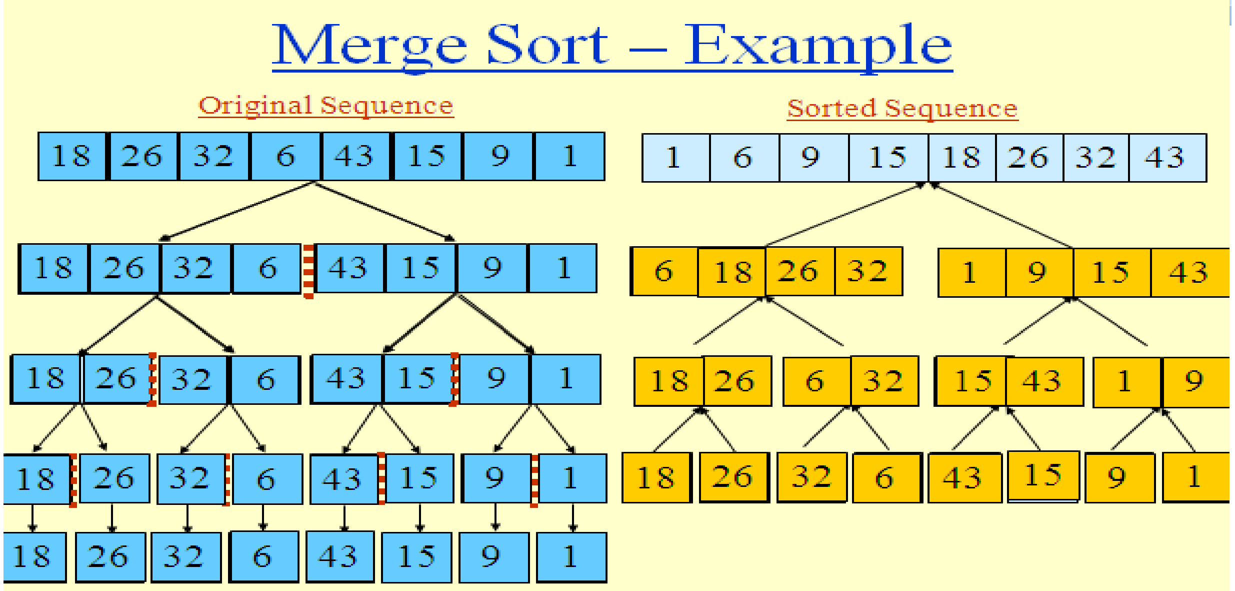
```
#include <iostream>
using namespace std;

//Print values
void print_ar (int ar[], int size)
{
  for (int i = 0; i < size; ++i)
```

```
 {
  cout << ar[i] << " ";
 }
 cout << endl;
}

//Shell sort function
void shell_sort (int ar[], int size)
{
 int j;

 //Narrow the array by 2 everytime
 for (int gap = size / 2; gap > 0; gap /= 2)
 {
  for (int i = gap; i < size; ++i)
  {
   int temp = ar[i];
   for (j = i; j >= gap && temp < ar[j - gap]; j -= gap)
   {
    ar[j] = ar[j - gap];
   }
   ar[j] = temp;
  }
 }
}

//Driver Functions
int main ()
{
 int ar[] = {1, 4, 16, 30, 29, 18, 100, 2, 43, 1};

 cout << "Intial Array : ";
 print_ar (ar, 10);

 shell_sort (ar, 10);

 cout << "Sorted Array : ";
 print_ar (ar, 10);

 return 0;
}
```

Merge Sort

Merge sort is based on Divide and conquer method.

- *Divide* the problem into sub-problems that are similar to the original but smaller in size
- *Conquer* the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
- *Combine* the solutions to create a solution to the original problem



IMPLEMETATION

```
#include <iostream>
using namespace std;
#include <conio.h>
void merge(int *,int, int , int );
void mergesort(int *a, int low, int high)
{
   int mid;
   if (low < high)
   {
      mid=(low+high)/2;
      mergesort(a,low,mid);
      mergesort(a,mid+1,high);
      merge(a,low,high,mid);
   }
   return;
}
void merge(int *a, int low, int high, int mid)
{
   int i, j, k, c[50];
   i = low;
```

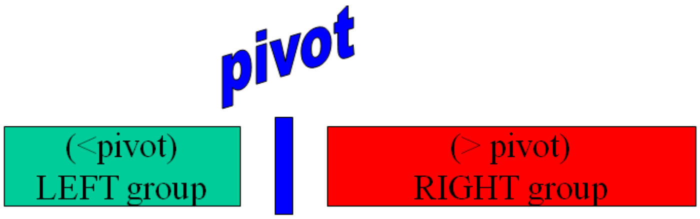
```
    k = low;
    j = mid + 1;
    while (i <= mid && j <= high)
    {
        if (a[i] < a[j])
        {
            c[k] = a[i];
            k++;
            i++;
        }
        else
        {
            c[k] = a[j];
            k++;
            j++;
        }
    }
    while (i <= mid)
    {
        c[k] = a[i];
        k++;
        i++;
    }
    while (j <= high)
    {
        c[k] = a[j];
        k++;
        j++;
    }
    for (i = low; i < k; i++)
    {
        a[i] = c[i];
    }
}
int main()
{
    int a[20], i, b[20];
    cout<<"enter  the elements\n";
    for (i = 0; i < 5; i++)
    {
        cin>>a[i];
    }
    mergesort(a, 0, 4);
```

```
  cout<<"sorted array\n";
  for (i = 0; i < 5; i++)
  {
     cout<<a[i];
  }
  cout<<"enter  the elements\n";
  for (i = 0; i < 5; i++)
  {
     cin>>b[i];
  }
  mergesort(b, 0, 4);
  cout<<"sorted array\n";
  for (i = 0; i < 5; i++)
  {
     cout<<b[i];
  }
  getch();
}
```

QUICK SORT

Quick sort is a divide and conquer algorithm. Its divided large list in mainly three parts:

1. Elements less than pivot element.
2. Pivot element.
3. Elements greater than pivot element.



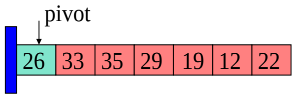
Steps to be followed

1. While **i** is less than **j** ,**i** less than **pivot**, we keep in incrementing **i**
2. Similarly, **i** is less than **j**,while **i** is greater than **pivot** keep decrementing **j**
3. If **i>j or j<pivot** swap the elements at the indexes of **i** and **j** respectively.
4. Once **i & j interchanges** fix the pivot and check whether the elements at left are less than pivot and elements at right are greater than pivot.

Repeat the steps 1,2,3,& 4 until elements to be sorted

Quicksort Step 1

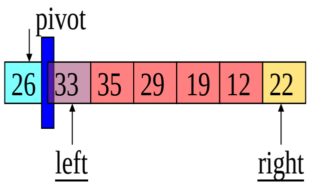
Step 1, select a pivot
(it is arbitrary)



We will select the first element, as presented in the original algorithm by C.A.R. Hoare in 1962.

Quicksort Step 2

Step 2, start process of dividing data into LEFT and RIGHT groups:



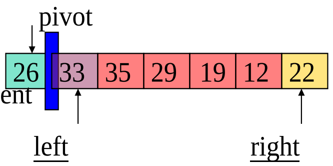
The LEFT group will
have elements less than
the pivot.

The RIGHT group will have
elements greater that the pivot.

Use markers left and right

Quicksort Step 3

Step 3,
If left element belongs
to LEFT group, then increment
left index.



If right index element belongs
to RIGHT, then decrement right.

Exchange when you find
elements that belong to the other
group.

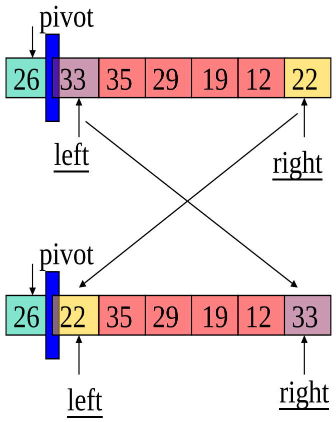
Quicksort Step 4

Step 4:

Element 33 belongs
to RIGHT group.

Element 22 belongs
to LEFT group.

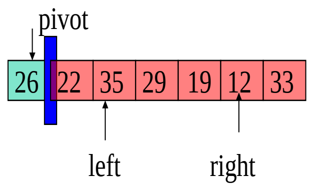
Exchange the two
elements.



Quicksort Step 5

Step 5:

After the exchange, increment left marker, decrement right marker.



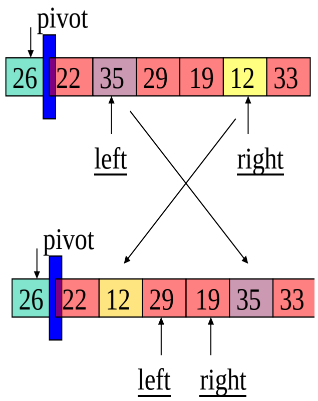
Quicksort Step 6

Step 6:

Element 35 belongs to RIGHT group.

Element 12 belongs to LEFT group.

Exchange, increment left, and decrement right.



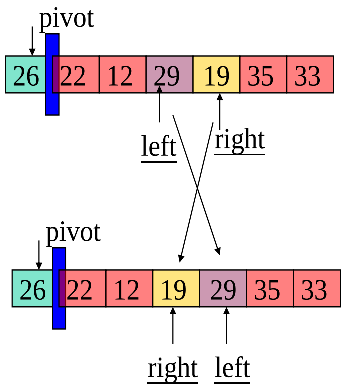
Quicksort Step 7

Step 7:

Element 29 belongs to RIGHT.

Element 19 belongs to LEFT.

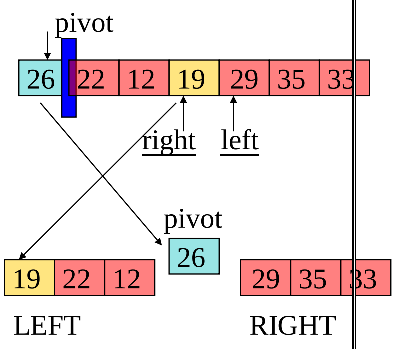
Exchange, increment left, decrement right.

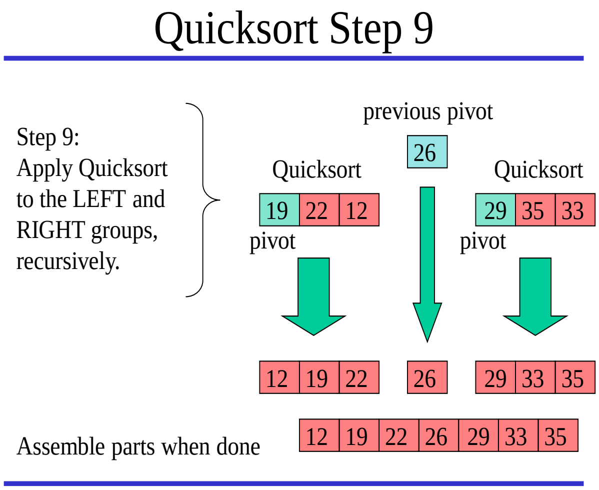


Quicksort Step 8

Step 8:
When the left and right markers pass each other, we are done with the partition task.

Swap the right with pivot.

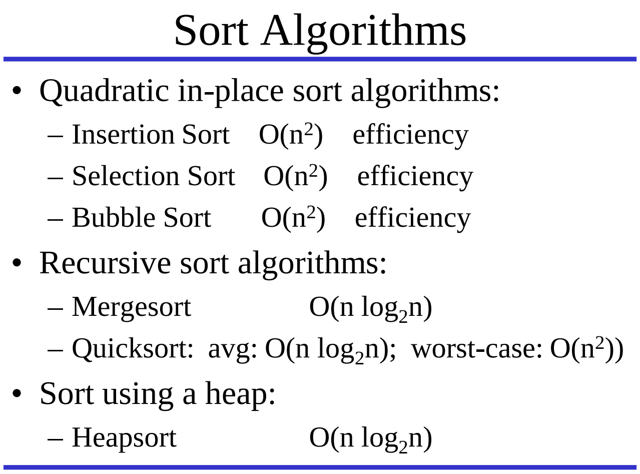




Quicksort Step 9

Step 9:
Apply Quicksort to the LEFT and RIGHT groups, recursively.

Assemble parts when done



Sort Algorithms

- Quadratic in-place sort algorithms:
  - Insertion Sort O(n^2) efficiency
  - Selection Sort O(n^2) efficiency
  - Bubble Sort O(n^2) efficiency
- Recursive sort algorithms:
  - Mergesort O(n $\log_2 n$)
  - Quicksort: avg: O(n $\log_2 n$); worst-case: O(n^2))
- Sort using a heap:
  - Heapsort O(n $\log_2 n$)

Efficiency of shell sort and radix sort –O(nlog n)

IMPLEMENTATION

```
#include<iostream>
#include<conio.h>
using namespace std;

//Function for partitioning array
int part(int low,int high,int *a)
{
    int i,h=high,l=low,p,t;  //p==pivot
    p=a[low];
    while(low<high)
    {
                while(a[l]<p)
                {
                            l++;
                }
                while(a[h]>p)
                {
                            h--;
                }
                if(l<h)
                {
                          t=a[l];
                          a[l]=a[h];
                          a[h]=t;
                }
                else
                {
                   t=p;
                   p=a[l];
                   a[l]=t;
                   break;
                }
    }
    return h;
```

```
}

void quick(int l,int h,int *a)
{
 int index,i;
 if(l<h)
 {
        index=part(l,h,a);
        quick(l,index-1,a);
        quick(index+1,h,a);
 }
}

int main()
{
    int a[100],n,l,h,i;
    cout<<"Enter number of elements:";
    cin>>n;
    cout<<"Enter the elements (Use Space As A Separator):";
    for(i=0;i<n;i++)
    cin>>a[i];
    cout<<"\nInitial Array:\n";
    for(i=0;i<n;i++)
    {
                cout<<a[i]<<"\t";
    }
    h=n-1;
    l=0;
    quick(l,h,a);
    cout<<"\nAfter Sorting:\n";
    for(i=0;i<n;i++)
    {
            cout<<a[i]<<"\t";
    }
    getch();
    return 0;
}
```

RADIX SORT

Sorting methods which utilize digital properties of the numbers (keys) in the sorting process are called *radix sorts*.

Example. Consider the list 459 254 472 534 649 239 432 654 477

Step 1: ordering ones	**Step 2: ordering tens**	**Step 3: ordering hundreds**
0	0	0
1	1	1
2 472 432	2	2 239 254
3	3 432 534 239	3
4 254 534 654	4 649	4 432 459 472 477
5	5 254 654 459	5 534
6	6	6 649 654
7 477	7 472 477	7
8	8	8
9 459 649 239	9	9
STEP 1	**STEP 2**	**STEP 3**

After step 1: 472 432 254 534 654 477 459 649 239
After step 2: 432 534 239 649 254 654 459 472 477
After step 3: 239 254 432 459 472 477 534 649 654

SEARCHING

Linear search or sequential search is one of the **searching algorithm** in which we have some data in a data structure like **array data structure** and we have to search a particular element in it which is know as key.

By traversing the whole **data structure elements from start to end** one by one to find key comparing with each data structure element to the key.

In case of an array we check that the given key or a number is present in array at any index or not by comparing each element of array

There can be two possible outcomes if we are assuming that **data structure like array contains unique values.**

- **Linear search successful Key Found** means in array at an index we found value which matches to our key
- **Linear search failed** to find the Key mean our key does not exist in data

Linear Search in C++ Program Example Code

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

int main() {
cout<<"Enter The Size Of Array:   ";
int size;
cin>>size;

int array[size], key,i;

// Taking Input In Array
 for(int j=0;j<size;j++){
 cout<<"Enter "<<j<<" Element: ";
 cin>>array[j];
 }

//Your Entered Array Is
 for(int a=0;a<size;a++){
    cout<<"array[ "<<a<<" ]  =  ";
    cout<<array[a]<<endl;
 }

 cout<<"Enter Key To Search  in Array";
 cin>>key;

   for(i=0;i<size;i++){
      if(key==array[i]){
  cout<<"Key Found At Index Number :  "<<i<<endl;
  break;
    }
 }

if(i != size){
cout<<"KEY FOUND at index :  "<<i;
}
else{
cout<<"KEY NOT FOUND in Array  ";
}
   return 0;
}
```

BINARY SEARCH

Algorithm:

It starts with the middle element of the list.

1. If the middle element of the list is equal to the 'input key' then we have found the position the specified value.

2. Else if the 'input key' is greater than the middle element then the 'input key' has to be present in the last half of the list.

3. Or if the 'input key' is lesser than the middle element then the 'input key' has to be present in the first half of the list.

Hence, the search list gets reduced by half after each iteration.

First, the list has to be sorted in non-decreasing order.

[low, high] denotes the range in which element has to be present and [mid] denotes the middle element. Initially low = 0, high = number_of_elements and mid = floor((low+high)/2). In every iteration we reduce the range by doing the following until low is less than or equal to high(meaning elements are left in the array) or the position of the ‘input key’ has been found.

(i) If the middle element (mid) is less than key then key has to present in range [mid+1 , high], so low=mid+1, high remains unchanged and mid is adjusted accordingly
(ii) If middle element is the key, then we are done.
(iii) If the middle element is greater than key then key has to be present in the range [low,mid-1], so high=mid-1, low remains unchanged and mid is adjusted accordingly.

Properties:

1. Best Case performance – The middle element is equal to the ‘input key’ O(1).

2. Worst Case performance - The ‘input key’ is not present in the list O(logn).

3. Average Case performance – The ‘input key’ is present, but it’s not the middle element
O(logn).

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

void main()
{
int search(int [],int,int);
clrscr();
int n,i,a[100],e=-3,res;
cout<<“How Many Elements:”;
cin>>n;
cout<<“nEnter Elements of Array in Accending ordern”;

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

```
cin>>a[i];
}
cout<<“nEnter element to search:”;
cin>>e;
res=search(a,n,e);
if(res!=0)
cout<<“nElement is Founded at “<<res+1<<“st position”;
else
cout<<“nElement is not found….!!!”;
getch();
}

int search(int a[],int n,int e)
{
int f,l,m;
f=0;
l=n-1;
while(f<=l)
{
m=(f+l)/2;
if(e==a[m])
return(m);
else
if(e>a[m])
f=m+1;
else l=m-1;}return 0;
}
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

COIMBATORE-21
Faculty of Engineering
Department of Computer Science and Engineering
UNIVERSITY EXAMINATION

Subject Code : 17BECS5E02
Name of the Course : III B.E CSE
Title of the paper : ADVANCED DATA STRUCTURES

Part-A

Answer All Questions (9*2=18)

1. What is byte code? Mention its advantage.
2. State the difference between Instance method and Class method.
3. What is the difference between Superclass and Subclass?
4. State the difference between Class and Interface.
5. What is an Exception? State its types.
6. What is Polymorphism?
7. State difference between inheritance and interface.
8. Give an example for non-parameterized Constructor.
9. What is a Pure Virtual function?

Part-B

Answer ALL Questions (3*14=42)

10. Explain the concepts of Object Oriented Programming.

OR

11. Explain Dynamic method dispatch with an example
12. Explain the types of inheritance in C++ with examples.

OR

13. What is sorting? Explain different types of sorting with examples.

14. State difference between BFS and DFS with example.

OR

15. What is binary tree. Explain with example.

ONLINE 1 MARK QUESTIONS

1. Which if the following is/are the levels of implementation of data structure
A) Abstract level B) Application level C) Implementation level D) All of the above

2. A binary search tree whose left subtree and right subtree differ in hight by at most 1 unit is called ……
A) AVL tree B) Red-black tree C) Lemma tree D) None of the above

3. ……………….. level is where the model becomes compatible executable code
A) Abstract level B) Application level C) Implementation level D) All of the above

4. Stack is also called as
A) Last in first out B) First in last out C) Last in last out D) First in first out

5. Which of the following is true about the characteristics of abstract data types? i) It exports a type. ii) It exports a set of operations
A) True, False B) False, True C) True, True D) False, False

6. …………… is not the component of data structure.
A) Operations B) Storage Structures C) Algorithms D) None of above

7. Which of the following is not the part of ADT description?
A) Data B) Operations C) Both of the above D) None of the above

8. Inserting an item into the stack when stack is not full is called …………. Operation and deletion of item form the stack, when stack is not empty is called ………..operation.
A) push, pop B) pop, push C) insert, delete D) delete, insert

9. ……………. Is a pile in which items are added at one end and removed from the other.
A) Stack B) Queue C) List D) None of the above

10. ………… is very useful in situation when data have to stored and then retrieved in reverse order.
A) Stack B) Queue C) List D) Link list

11. Which data structure allows deleting data elements from and inserting at rear?
A) Stacks B) Queues C) Dequeues D) Binary search tree

12. Which of the following data structure can't store the non-homogeneous data elements?
A) Arrays B) Records C) Pointers D) Stacks

13. A ……. is a data structure that organizes data similar to a line in the supermarket, where the first one in line is the first one out.
A) Queue linked list B) Stacks linked list C) Both of them D) Neither of them

14. Which of the following is non-liner data structure?
A) Stacks B) List C) Strings D) Trees

15. Herder node is used as sentinel in …..
A) Graphs B) Stacks C) Binary tree D) Queues

16. Which data structure is used in breadth first search of a graph to hold nodes?
A) Stack B) queue C) Tree D) Array

17. Identify the data structure which allows deletions at both ends of the list but insertion at only one end.
A) Input restricted dequeue B) Output restricted qequeue C) Priority queues D) Stack

18. Which of the following data structure is non linear type?
A) Strings B) Lists C) Stacks D) Graph

19. Which of the following data structure is linear type?
A) Graph B) Trees C) Binary tree D) Stack

20. To represent hierarchical relationship between elements, Which data structure is suitable?
A) Dequeue B) Priority C) Tree D) Graph

21. A directed graph is ………………. if there is a path from each vertex to every other vertex in the digraph.
A) Weakly connected B) Strongly Connected C) Tightly Connected D) Linearly Connected

22. In the …………….. traversal we process all of a vertex's descendants before we move to an adjacent vertex.
A) Depth First B) Breadth First C) With First D) Depth Limited