PRATHYUSHA ENGINEERING COLLEGE

(An ISO 9001:2000 Certified Institution)
Aranvoyalkuppam

(Affiliated to Anna University, Chennai)

LAB MANUAL

FOR

CS 6311/ PROGRAMMING AND DATASTRUCTURES – II LABORATORY

REGULATION 2013

B.E COMPUTER SCIENCE AND ENGINEERING

YEAR/SEM: II / III

ACADEMIC YEAR: 2016-2017

Prepared by

Ms. S. Famitha / Asst. Prof – II / CSE

UNIT I

OBJECT ORIENTED PROGRAMMING FUNDAMENTALS

C++ Programming features - Data Abstraction - Encapsulation - class - object - constructors – static members – constant members – member functions – pointers – references - Role of this pointer – Storage classes – function as arguments.

INTRODUCTION

C++ is an object oriented programming language. It was developed by **Bjarne Strousturp at AT&T Bell Laboratories** in Murray Hill, New Jersey, USA, in the early 1980's. C++ is a superset of C.

C++ is a general purpose programming language with a bias towards systems programming that

– is a better than C,

– supports data abstraction, data hiding

– supports object oriented programming, and

– supports generic programming.

PROGRAMMING METHODOLOGIES *

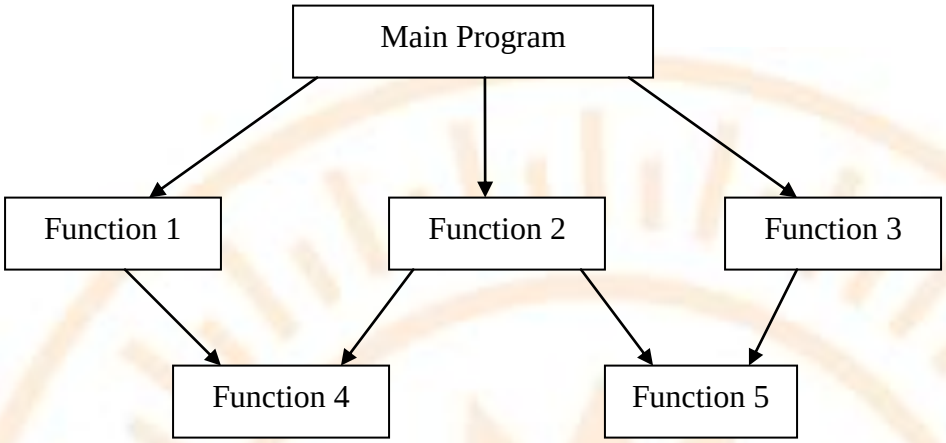
The earliest computers were programmed in binary. Mechanical switches were used to load programs. With the advent of mass storage devices and larger and cheaper computer memories, the first high-level computer programming languages came into existence. They suffered from following limitations:

- No facilities to reuse existing program code.
- Control was transferred via the dangerous goto statement.
- All variables in the program were global. Tracking down spurious changes in global data was a very tedious job.
- Writing, understanding and maintaining long programs are very difficult.

Structured Programming:

- A structured program is built by breaking down the program's primary purpose into smaller pieces that then become functions within the program.
- Each function can have its own data and logic.
- Information is passed between functions using parameters and functions can have local data that cannot be accessed outside the function scope. It helps you to write cleaner code and maintain control over each function.

- By isolating the processes within the functions, a structured program minimizes the chance that one procedure will affect another.



Limitations:

- Data is given second class status in the procedural paradigm even though data is the reason for program's existence.
- Data types are processed in many functions, and when changes occur in data types, modifications must be made at every location that acts on those data types within the program.
- Structured programming components – functions and data structures doesn't model the real world very well.
- It emphasizes on fitting a problem to the procedural approach of a language.

Object oriented programming (OOP)

Definition of OOP: It is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand.

OOP allows decomposition of a problem into a number of entities called **objects** and then builds data and functions around these objects.

The organization of data and functions in object-oriented programs is shown in fig.1.1. The data of an object can be accessed only by the functions associated with that object. However functions of one object can access the functions of other objects.

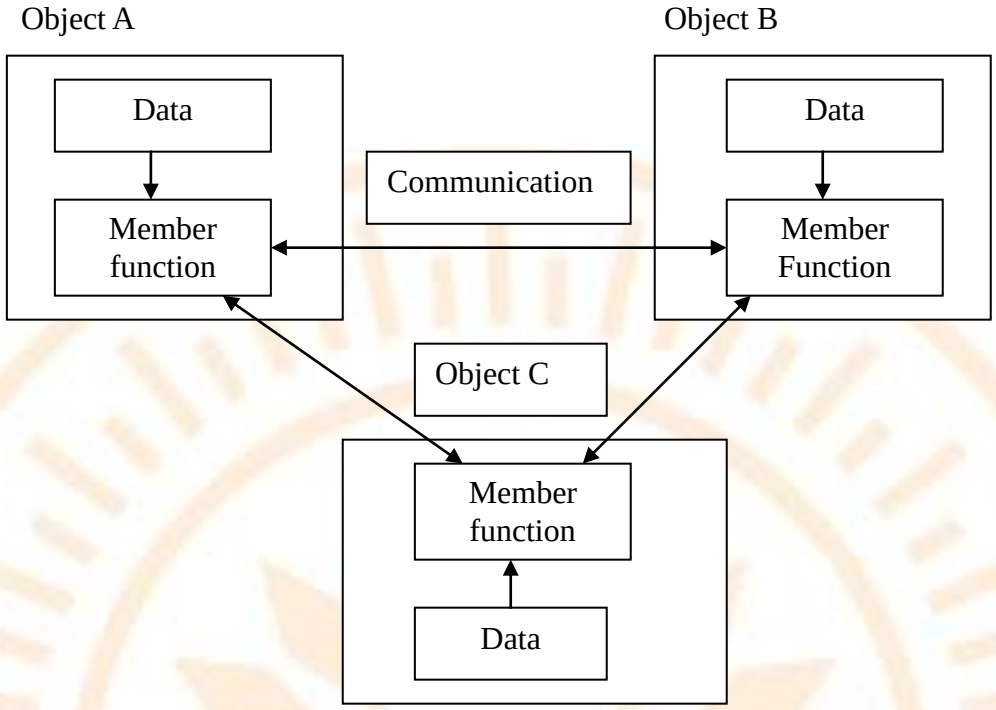


Fig. 1.1 Organization of data and functions in OOP

Procedural Programming

The original programming paradigm is:

Decide which procedures you want; use the best algorithms you can find.

The focus is on the processing – the algorithm needed to perform the desired computation. Languages support this paradigm by providing facilities for passing arguments to functions and returning values from functions.

Comparison between Structured and Object Oriented Programming:

Structured Programming	**Object Oriented programming**
Emphasis is on doing things	Emphasis is on data
Large programs are divided into smaller programs known as functions	Programs are divided into what are known as objects.
Most of the functions share global data	Functions that operate on data of the object are tied together in the data structures
Data move openly around the system from function to function.	Data is hidden and cannot be accessed by external function.

Its components doesn't model the real world objects	Its components model the real world objects
Employs top-down approach in program design.	Follows bottom-up approach in program design.

Modular Programming

A set of related procedures with the data they manipulate is often called a module.

Decide which modules you want; partition the program so that data is hidden within modules.

This paradigm is also known as the **datahiding** Principle. There is no grouping of procedures with related data. The techniques for designing ''good procedures'' are now applied for each procedure in a module.

1.1 C++ PROGRAMMING FEATURES

C++ is the multi paradigm, compile, free form, general purpose, statistically typed programming language. This is known as middle level language as it comprises of low level and high level language features.

The main features of the C++ are

1. **Classes**
2. **Objects**
3. **Methods**
4. **Message Passing**
5. **Data Abstraction**
6. **Encapsulation**
7. **Inheritance**
8. **Polymorphism**
9. **Dynamic Binding**

1) Classes

A class is a user defined type. **A class is a way to bind the data and its associated functions together.** It allows the data to be hidden if necessary from external use. While defining a class, we are creating a new **abstract data type** that can be treated as a built-in data type. A class specification has two parts:

1. Class declaration – describes the type & scope of its members
2. Class function definitions – describe how the class functions are implemented.

Syntax for class

```
class class_name
{
        private:
                variable declarations;
                function declarations:
        public:
                variable declarations;
                function declarations;
```

Fig. 1.1. General form of class declaration

- The keyword **class** specifies that what follows is an abstract data of type **class_name.**
- The body of the class is enclosed within braces and terminated by a semicolon.
- The class body contains the declaration of variables and functions. These functions and variables are collectively called **class members.**
- The keywords **private and public** are known as visibility labels or access specifiers and it specify which members are private which of them are public. These should followed by a colon.

Private members can be accessed only within the class whereas public members can be accessed from outside the class. **By default, the members of a class are private**. If both the labels are missing, the members are private to the class.

2) Objects

Objects are the instances of the classes. Objects are the basic run-time entities/ real time entities in object oriented programming. They may represent a person, a place, a bank account or any item that the program has to handle.

Example: Fruit is a Class where apple, mango, orange are objects for that class Fruit.

Each object contains data and code to manipulate the data. Objects can interact without having to know the details of each other data or code. Fig. 1.2 shows the representation of an object

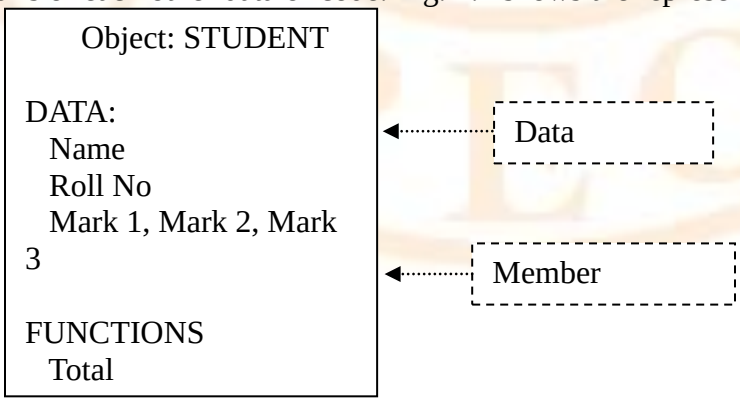


Fig. 1.2. Notation of an object

3) Methods

The functions we use in C++ is called as **methods.** There are two types of functions. **Member functions and non- member functions.**

One belongs to the class which is used to access the data of that particular class. This type of functions is called as **member functions**. The member functions may be defined inside the class or outside the class. If the function is defined outside the class, scope resolution operator **(::)** is used.

There are cases we can have functions which are not belong to the classes. They are called as **non-member functions**.

Functions in C++:

Syntax :

```
Return type  function name( );              //Empty function
Return type  function name(arg);    //  Function with argument
```

Example :

```
void show();                        //Function declaration
void main( )
{       …..
        show();                     //Function call
        …….
}
void show( )                        //Function definition
{       ……
        ……                          //Function body          }
```

4) Messages Passing

The process of programming in OOP involves the following basic steps:

- Creating classes that define objects and their behavior
- Creating objects from class definitions
- Establishing communication among objects

A message for an object is a request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.

E.g.: employee.salary(name);

Object: employee

Message: salary

Information: name

5) Data Abstraction

Abstraction represents the act of representing the essential features without including the background details or explanations. The attributes are called data members and functions are called member functions or methods. They should be facilitated only with the operations and not with the implementation.

For example, Applying break is an operation. It is enough for the person who drives the car to know how much pressure he has to apply on the break pad rather than how the break system functions. The car mechanic will take care of the breaking system.

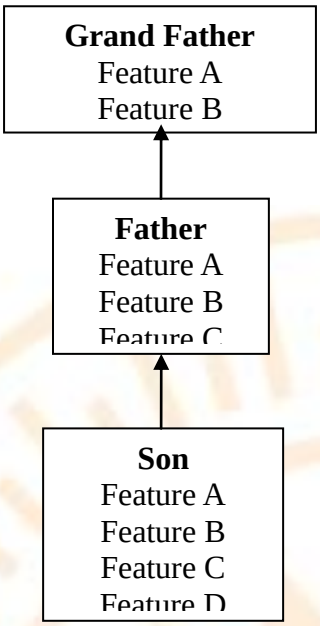
6) Encapsulation

The wrapping up of data and functions into a single unit is known as encapsulation. It is the most striking feature of the class. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. This insulation of the data from direct access by the program is called **data hiding or information hiding.**

7) Inheritance

It is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example, the person 'son' is a part of the class 'father' which is again a part of the class 'grand father'.

This concept provides the idea of **reusability**. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from an existing one. The new class will have the combined features of both the base class and the derived class.



8) Abstract Classes

The classes without any objects are known as abstract classes. These classes are usually outcome of generalization process which finds out common elements of the classes of interest and stores them in a base class. The abstract classes are very important from the designer's point of view. Whenever a designer has to model an abstract concept which cannot have objects, then abstract classes are handy.

For example consider the class **shape**. It can be inherited to rectangle, ellipse, which in turn can be inherited to parallelograms into squares and circles. Except shape, all other can have objects of them.

9) Polymorphism

It means the ability to take more than one form. An operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example the operation addition will generate sum if the operands are numbers whereas if the operands are strings then the operation would produce a third string by concatenation.

The process of making an operator to exhibit different behaviors in different instances is known as **operator overloading.**

A single function name can be used to handle different types of tasks based on the number and types of arguments. This is known as **function overloading.**

10 Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as Late binding) means that the code associated with a given procedure call is not known until the time of at run-time. It is associated with polymorphism and inheritance.

Benefits of OOP:

- Through inheritance, we can eliminate redundant code and extend the use of existing classes
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
- It is easy to partition the work in a project based on the objects.
- Object oriented systems can be easily upgraded from small to large systems.
- Software complexity can be easily managed.
- Object oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects make the interface description with external systems much simpler.

Applications of OOP:

- Development of Operating Systems
- Creation of DLL (Dynamic Linking Library)
- Computer Graphics
- Network Programming (Routers, firewalls)
- Voice Communication (Voice over IP)
- Web-servers (search engines)

Namespaces

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can **define the context in which names are defined**. In essence, a namespace defines a scope.

Example :

#include<iostream.h>

using namespace std;

***Limitations of C structures:**

The standard C doesn't allow the struct data type to be created like built-in types. They do not permit data hiding. Structure members can be directly accessed by the structure variables by any function anywhere in their scope. In simple words, the structure members are public members.

***Extensions to structures:**

C++ supports all the features of structures as defined in C.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as **'private'** so that they cannot be accessed directly by the external functions. In C++, the **keyword struct can be omitted** in the **declaration of structure variable**.

In short, **a class serves as a blueprint or a plan or a template. It specifies what data and what functions will be included in objects of that class.**

Let us study an example of an object of student class. Consider the problem of printing student information.

Example1:

// Program to demonstrate on objects:

```
#include<iostream.h>
#include<string.h>
using namespace std; // optional
class student
{
    public:
        int Rollno;
        char Name[20];
        char Address[20];
    public:
        void GetDetails()
        {
            cout<<" Enter the roll number";
            cin>> Rollno;
            cout<<" Enter the Name;
            cin>>Name;
            cout<<"Enter the Address";
            cin>>Address;        }
        void PrintDetails()
        {
```

```
                cout<<"Roll Number is " << Rollno<<"\n";
                cout<<"Name is " << Name<<"\n";
                cout<<"Address is " << Address<<"\n";
        }
    };
void main( )
{
    student S;
    S.GetDetails();
    S.PrintDetails( );      }
```

Example 2:

// Program to demonstrate on objects:

```
#include<iostream.h>
#include<string.h>
using namespace std;
class student
{
    public :
        int RNo;
        string Name;
        string Address;
        void PrintDetails()
        {
            cout<<"Roll Number is " << RNo<<"\n";
            cout<<"Name is " << Name<<"\n";
            cout<<"Address is " << Address<<"\n";
        }
};
void main( )
{
    student S;
```

```
        S..RNo = 1;
        S.Name = "Vinston Raja";
        S.Address = "Anna Nagar";
        S.PrintDetails( );
}
```

The only difference between structure and class in C++ is that, by default, the members of a class are private, while, by default, the members of a structure are public.

1.2 DATA ABSTRACTION

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

In television, which can be turned on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT we do not know its internal details, that is, how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

If we talk in terms of C++ Programming, C++ classes provides great level of **data abstraction**.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this:

```
#include <iostream.h>
int main( )
{
cout << "Hello C++" <<endl;
return 0;
}
```

In C++, we use access labels to define the abstract interface to the class.

- Members defined with a public label are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.
- Members defined with a private label are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.
- Members defined with a protected label are accessible to code that uses the class and immediate derived class.

- Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Data abstraction provides two important advantages/ benefits:

- Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.
- The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.
- By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data are public, then any function that directly accesses the data members of the old representation might be broken.

Data Abstraction Example:

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example:

```
#include <iostream.h>
class Add
{
        public:
                int total = 0;
        public:
        void addNum(int number)
            {
            total += number;
                 }
        int getTotal()
                {
                        return total;
                    }
};

int main( )
{
 Add a;
```

```
a.addNum(10);
a.addNum(20);
 a.addNum(30);
cout << "Total " << a.getTotal() <<endl;
 return 0;
}
```

1.3 ENCAPSULATION

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**.

All C++ programs are composed of the following two fundamental elements:

- **Program statements (code):** This is the part of a program that performs actions and they are called functions.
- **Program data:** The data is the information of the program which affected by the program functions.

Example for Encapsulation

```
#include <iostream.h>
class Square
{
        private:
                int area; // hidden data from outside world
        public:
        void calcArea(int x)
        {
                area = a*a;          }
        int getArea()
        {
                return(area);        }
 };
```

```
int main( )
{
  Square s;
  s.calcArea(10);
  cout << "Area  : " << s.getArea() <<endl;
  return 0;
}
```

1.4 CLASS

In OOP, class is a user defined data type that can hold data and their associated functions into single unit.

The class members are divided into two types:

1. Data Member
2. Function Member

In class there are certain privilege for accessing both data and function members. They are said to be access specifiers. It is divided into three categories.

They are 1.Public 2.private 3.Protected.

- By default the data members are private. So the private members of a class are not accessible from outside the class. If the data members are specified as public then the members of a class can be accessible from inside and outside of the class. Members defined with a protected label are accessible to code that uses the class and immediate derived class.

Syntax of a class

```
class class_name
{
Access Specifier:
Data Members
Access Specifier:
Function
Members
};
```

Example:

```
class student
{
public:
 int rollno,age;
char *name;
void getdetail( )
{
        cin>>rollno>>age;
        cin>>name; }
void printdetail( )
{
        cout<<rollno<<age;
        cout<<name;  }        };
```

Function Definition outside the Class The function members which are declared inside the class can be defined outside of the class using **Scope Resolution Operator ::**

Syntax for Defining the Function outside the Class:

Return Type class_name::function_name( )

{

Function Body

}

Return type specifies the type of a function.

Class name specifies the name of a class in which the function belongs.

The operator:: denotes scope resolution operator.

Function name specifies the name of a function.

Example:

```
class student
{
```

```
public:
int rollno,age;
char *name;
void getdetail();
 void printdetail();
 };
void student::getdetail( )
{
cin>>rollno>>age;
 cin>>name;
}
void student::printdetail( )
{
cout<<rollno<<age;
cout<<name;
}
```

1.5 OBJECTS

Objects are instances of the class. It is a basic runtime entity in OOPs concept.

Objects are the variables of user defined data type called class. Once a Class has been created we can declare any number of variables belongs to that class.

In the above example class student we can declare any number of variables of class student in the main function. Using objects we can access any public members of a class using **Dot Operator**.

For accessing Data Members and assigning value:

object_name.data_member=value;

For accessing Function Members:

Object_name.function_name();

Syntax: Class_Name object1, object2, object3…….object n;

```
void main ( )
    {
    student s1, s2,s3; // Class name is student; s1,s2,s3 are objects(variables) of class Student;
```

s1.rollno=28; // Object s1 assigns the value for the data member rollno=28

s1.getdetail (); // Object s1 access (call) the function member getdetail () of student class

s1.printdetail (); Object s1 access (call) the function member printdetail () of student class

}

Memory Requirement for Objects and Class:

Once the object is created memory is allocated for the data members of a class and not for its function Members .So each object holds the total memory size of the data members.

For example in the class student the object s1 holds the memory size of 5 bytes. Char 1 byte, int 2 bytes.

Once the class is created only single copy of function member is maintained which is common for all the objects.

1.6 CONSTRUCTOR

The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

A constructor is a special member function of a class that is executed whenever an object of that class is created.

- A constructor will have exact same name as the class.
- Must be declared as a public member of a class.
- It does not have any return type at all, not even void.
- Constructors are useful for initializing the objects by setting initial values for certain member variables.
- Constructor gets automatically called when an object of class is created.

For example:

```
class Date {
/ / ...
Date (int , int , int ); // constructor     };
```

Types of constructors:

1. Default constructor
2. Parameterized Constructor
3. Copy Constructor
4. Dynamic constructor

1. **Default Constructor:-**

Default Constructor is also called as Empty Constructor which has no arguments and it is automatically called when we create an object of the class.

```
#include <iostream>
 using namespace std;
 class Line
{
  private:
     double length;
   public:
     void setLength( double len );
     double getLength( void );
     Line( );  // This is the constructor
};
 // Member functions definitions including constructor
Line::Line()                    // default constructor
{
   Length=0;  // default value
}
 void Line::setLength( double len )
{
   length = len;
}
 double Line::getLength( void )
{
   return length;
}
// Main function for the program
int main( )
{
   Line line;                   // no arguments are passed
```

```
    line.setLength(6.0);            // set line length
    cout << "Length of line : " << line.getLength() <<endl;
     return 0;
}
```

2. **Parameterized Constructor: -** This type of constructor has some arguments and function name as class name. While creating objects of class, arguments need to be passed along with the name of class. Values of these arguments are used to initialize the data members of the class.

```
#include <iostream>
using namespace std;
class Line
{
private:
double length;
public:
void setLength( double len );
double getLength( void );
Line(double len); // This is the constructor
};
// Member functions definitions including constructor
//parameterized constructor
Line::Line( double len)
{
cout << "Object is being created, length = " << len << endl;
length = len;
}

void Line::setLength( double len )
{
length = len;
}
double Line::getLength( void )
```

```
{
return length;
}
// Main function for the program
int main( )
{
Line line(10.0);  //parameters are passed during object creation
cout << "Length of line : " << line.getLength() <<endl;              // get initially set length.
line.setLength(6.0);                                                 // set line length again
cout << "Length of line : " << line.getLength() <<endl;
return 0;
}
```

3. **Copy Constructor**

The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (const classname &obj) {
// body of constructor
}
```

Here, **obj** is a reference to an object that is being used to initialize another object.

Copy constructor is required in the following cases (but not in assignment).

- A variable is declared which is initialized from another object, eg,

```
Person q("Mickey"); // constructor is used to build q.
Person r(p);        // copy constructor is used to build r.
```

```
Person p = q;       // copy constructor is used to initialize in declaration.
    p = q;          // Assignment operator, no constructor or copy constructor.
```

The copy constructor takes a reference to a const parameter. It is **const** to guarantee that the copy constructor doesn't change it.

Example : Object created from existing object using copy constructor

```
#include <iostream>
using namespace std;
 class Point {
    private:
    int x, y;
    public:
    Point(const Point& p);   // copy constructor
    void setPoint(int a, int b)
     {
                x=a; y=b;}

    void showPoint()
     {cout<<x<<","<<y<<endl;}
};

Point::Point(const Point& p)  // copy constructor
{
    x = p.x;
    y = p.y;              }
void main()
{
Point p;          // calls default constructor
p.setPoint(10,10);
Point s = p;       // calls copy constructor is invoked
p.showPoint();
s. showPoint();          }
```

Example : Copy constructor for passing object as argument

```
#include <iostream>
using namespace std;
class Line
{
private:
int *ptr;
public:
int getLength( void );
Line( int len ); // simple constructor
Line(const  Line &obj); // copy constructor
~Line(); //
};

// Member functions definitions including constructor
Line::Line(int len)
{
ptr = new int;            // allocate memory for the pointer;
*ptr = len;
}
Line::Line(const Line &obj)
{
cout << "Copy constructor allocating ptr." << endl;
ptr = new int;
*ptr = *obj.ptr; // copy the value
}
Line::~Line(void)
{
cout << "Freeing memory!" << endl;
delete ptr;
}
```

```
int Line::getLength( void )
{
return *ptr;
}
//non-member function taking an object as argument
void display(Line obj)
{
cout << "Length of line : " << obj.getLength() <<endl;
}
// Main function for the program
int main( )
{
Line line(10);
Line line2 = line; // This also calls copy constructor
display(line); // copy constructor is invoked to copy the content of object line to object obj.
display(line2);
return 0;
}
```

4. Dynamic constructor

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator.

Example:

```
class dynamiccons
{
      int  *ptr;
      public:
         dynamiccons
                  {
                           ptr = new int;
                           *ptr = 100;
                           }
```

```
        dynamiccons(int v)
           {
                    ptr = new int;
                    *ptr = 100;
                    }
int display( )
{
       return   (*ptr);
       }
};
void main( )
{
     dynamic cons obj1,obj2(90);
    cout<<" The Value of obj1 ptr is";
    cout<<obj1.display( );
    cout<<" The Value of obj2 ptr is";
    cout<<obj2.display( );
     }
```

OUTPUT:
The value of obj1 ptr is 100
The value of obj2 ptr is 90;

1.7 DESTRUCTOR

A **destructor,** as the name implies, is a special member function of a class is used to destroy the objects that have been created by a constructors. Like constructor, the destructor is a member function whose name is same as the class name but it is preceded by a tilde symbol (~).

Syntax: ~classname( );

Example: ~ Line ( );

Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically called when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

Some important points about destructors:

- Destructors take the same name as the class name.
- Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
- The Destructor does not take any argument which means that destructors cannot be overloaded.
- No return type is specified for destructors.

Following example explains the concept of destructor:

```
#include <iostream.h>
class Line
{
        public:
                Line(); // This is the constructor declaration
                ~Line(); // This is the destructor: declaration
        };
// Member functions definitions including constructor
Line::Line(void)
{
        cout << "Object is being created" << endl; }
Line::~Line(void)
{
         cout << "Object is being deleted" << endl;}
// Main function for the program
int main( )
{
Line L1,L2;
return 0; }
```

Output

```
Object is being created
Object is being created
Object is being deleted
Object is being deleted
```

1.8 **STATIC MEMBERS**

A variable that is part of a class, yet is not part of an object of that class, is called a static member. **There is exactly one copy of a static member instead of one copy per object**, as for ordinary non static members.

Static members include both

a. **Static data members**
b. **Static member functions**

Syntax

```
class stat
{       static int count; //static Data Member Declaration
        } ;
        int stat::count; //static Data member Defintion
```

a) Static data members

Static variables are normally used to maintain values common to the entire class. Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many

objects are created.

Some special characteristics of static variable:

- It is initialized to zero when the first object of its class is created. No other initialization is allowed.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class, but its lifetime is entire program
- Static variables are normally used to maintain values common to the entire class.

Example(Static data member)

```
#include<iostream.h>
 #include<conio.h>
 class item
{
        static int count;
        int number;
        public:
        void getdata ( int a)
        {
                number = a;
                count++;
        }

        Void getcount (vod)
        {
                cout<<"Count = " << count;
                }
};
int item::count;
void main()
{
Item a,b,c;
a.getcount( );
b.getcount( );
c.getcount( );
a.getdata(100);
b.getdata(200);
c.getdata(300);
a.getcount( );
b.getcount( );
c.getcount( );
}
```

Output:

Count = 0

Count = 0
Count = 0
Count = 3
Count = 3
Count = 3

In this example class count has a static data member count. This program is used for counting the number of objects which is declared in the class.

NOTE: Once the static data member is defined it is automatically initialized to zero.

b) Static member Function

A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **"this"** pointer of the class.

Syntax for declaring static member function:

```
classname :: function name ( );
```

Example: (Static member Function)

```
#include <iostream.h>
using namespace std;
class test
{
        int code;
        static int count; // static data member
        public:
                void setcode (void)
                {       code = ++ count;
                }
                void  showcode (void)
                {
                cout<<"Object Number :" << code;
                }
        static void showcount(void)
                {
                        cout<<"Count"<<count;
                        }
                        };
```

```
int test::count = 0;
int main(void)
{
        test T1,T2;
        T1.setcode ( );
         T2.setcode ( );
        test :: showcount ( ); // static member functions
        test T3;
        T3.setcode ( );
        test :: showcount( );
        T1.showcode( );
        T2.showcode( );
        T3.showcode( );
        return  0;
        }
```

Output:

```
Count = 2
Count = 3
Object number 1
Object number 2
Object number 3
```

1.9 CONSTANT MEMBERS

A constant member function is a member function that guarantees it will not change any class variable or call any non const member functions. The keyword **const** is used for specifying constant members.

Constant objects and constant functions

- Once the object is declared as constant, it cannot be modified by any function.
- Initially the value for the data members is set by constructor during the object creation.
- Constant objects can access only by constant member functions. The constant function is read only the function. It cannot alter the value of object data member.
- Const member functions declared outside the class definition must specify the **const** keyword on both function prototype in the class definition and on the function definition
- Any const member that attempts to change a member variable or call a non –member function will causes an error.
- Constructor should not mark as **const**.

Type Qualifiers in C++

The type qualifiers provide additional information about the variables they precede

Qualifier	Meaning
const	Objects of type **const** cannot be changed by your program during execution
volatile	The modifier **volatile** tells the compiler that a variable's value may be changed in ways not explicitly specified by the program.
restrict	A pointer qualified by **restrict** is initially the only means by which the object it points to can be accessed. Only C99 adds a new type qualifier called restrict.

Const member variable

These are data members that are declared as **const.** These const variable are not initialized during declaration but the initialization occur in the constructor.

```
Class myclass
{
const int i;      // const member declared
public:
myclass(int j)
{ i=j;} // const member initialized
void show()
{cout<<"i="<<i<<endl;}
int main()
{
myclass(5);
myclass(25);
}
```

In this program, 'i' is a const member of the class myclass. In every object its independent copy is present, hence it is initialized with each object using constructor. Once initialized, it's value cannot be changed.

Syntax:

```
class classname
        {
                classname( parameter)                  //      consttructor
                {    …...                          }
                returntype  func_name( ) const
                { // read only function                        // constant member function
                        }
                };
```

```
void main ( )
{
        const classname object(parameter);          //      constant object
        object. function( );            // constant objects access constant member function
}
```

Example

```
#include < iostream.h>
#include< conio.h>
class time
{
        public :
                int hr, min, sec;
        time ( int thr, int tmin, int tsec)
        {
                hr = thr;
                min = tmin;
                sec = tsec;     }
        void disp( ) const              // constant member function
        {
                cout<<" Time is : "<< hr<<" : "<<min <<":"<<sec<<"\n";
                }
                ~time( )
                {
                cout<< " Object is deleted"<<endl;  }
                void gettime ( ) const
                {
                cin>>hr>>min>>sec; }
                };
                void main( )
                {
                const time T(8,55,50);          // constant object
                T.disp( );
                T.gettime( );
                T.disp( );
                }
```

Output

```
Time is: 8:55:50

4

12

56

Time is: 4:12:56
```

1.10 MEMBER FUNCTIONS

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of the class.

Member functions can be defined within the class definition or separately using **scope resolution operator, ::**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier.

Example : member function defined inside a class definition

```
class Box
{
public:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box
Box(int x, int y, int z) // constructor
{length=x; breadth=y; height=z;}

double getVolume(void)        // member function defined inside the class definition
{                             // by default it is inline
return length * breadth * height;
}
};

void main()
{
Box b(10,20,10);
cout<<"volume = "<<b.getVolume();
}
```

Example : member function defined outside a class definition

```
class Box
{
public:
double length; // Length of a box
double breadth; // Breadth of a box
double height; // Height of a box

Box(int x, int y, int z) // constructor
{length=x; breadth=y; height=z;}
double getVolume(void);        // member function declared in a class
};

double Box::getVolume(void) // member function defined outside the class definition
{
return length * breadth * height;
}

void main()
{
Box b(10,20,10);
cout<<"volume = "<<b.getVolume();
}
```

Friend Functions :

In OOP environment, the private members cannot be accessed from outside the class. That is, a non-member function cannot have an access to the private data of a class. To allow a function (non-member) to have access to the private data of a class, the function has to be made friendly to that class. To make an outside function "friendly" to a class, we have to simply declare this function as a **friend** of the

class as shown below :

```
class ABC
{
                ……..                    // data members
                ……..
        public :
                ……..                    // member functions
                ……..
                friend void xyz(void);  // declaration
};
```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or the scope resolution operator **: :**.

The functions that are declared with the keyword friend are known as friend functions. A friend function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

Characteristics (certain special characteristics) of Friend functions:

- It is not in the scope of the class to which it has been declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member access.(e.g. A . x).
- It can be declared either in the public or private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

// program to demonstrates the use of a friend function

```
#include<iostream>
using namespace std;
class sample
{               int a; int b;
```

```
        public:
                void setvalue( )
                {
                        a = 25;  b = 40;
                }
                friend float mean (sample s);                   // declaration
};
float mean (sample s)
{
        return float(s.a + s.b) / 2.0 ; }
int main( )
{       sample X;                       // object X
        X.setvalue( );
        cout << "Mean Value = " << mean(X) << "\n";      return 0; }
```

The output of the above program would be :

Mean Value = 32.5

The friend function accesses the class variables a and b by using the dot operator and the object passed to it. The function calls mean(**X)** passes the object X by value to the friend function.

Member functions of one class can be **friend** functions of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
class X
{
        …….
        int fun1( );                    // member function of X
        …….
};
class Y
{
        …….
        friend int X : : fun1( );
```

```
        …….
};
```

The function **fun1 ( )** is a member of **class X** and a **friend** of class **Y**.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class Z
{
        …….
        friend class X;          // all member functions of X are friends to Z
};
```

//program with a function friendly to two classes

```
#include<iostream>
using namespace std;
class ABC ;                                  // forward declaration
class XYZ
{               int x;
        public :
                void setvalue(int i) { x = i ;}
                friend void max(XYZ, ABC);
};
class ABC
{               int a;
        public :
                void setvalue(int i) { a = i ;}
                friend void max(XYZ, ABC);          };
// definition of friend function
void max(XYZ m, ABC n)
{
```

```
        if(m.x >=  n.a)
                cout << m.x;
        else
                cout << n.a;
}
int main( )
{
        ABC abc;
        abc.setvalue(10);
        XYZ xyz;
        xyz.setvalue(20);
        max(xyz, abc);
        return 0;
}
```

The output of the above program would be :

20

Note : The function max( ) has arguments from both XYZ and ABC. When the functions max( ) is declared as a friend in XYZ for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as

```
class ABC;
```

This is known as 'forward' declaration.

The friend function is a '**non member function'** of a class. It can access non public members of the class. A friend function is external to the class definition.

A friend function has the following advantages:

- Provides additional functionality which is kept outside the class.
- Provides functions that need data which is not normally used by the class.
- Allows sharing private class information by a non member function.

1.11 POINTERS

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, we must declare a pointer before using it. The general form of a pointer variable declaration is:

type *var-name;

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

There are few important operations.

(a) define a pointer variables

(b) assign the address of a variable to a pointer

(c) access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand.

```
#include <iostream.h>
using namespace std;
int main ()
{
int var = 20;    // actual variable declaration.
int *ip;                    // pointer variable
ip = &var;                  // store address of var in pointer variable
cout << "Value of var variable: ";
cout << var << endl;
                            // print the address stored in ip pointer variable
cout << "Address stored in ip variable: ";
cout << ip << endl;
                            // access the value at the address available in pointer
cout << "Value of *ip variable: ";
cout << *ip << endl;
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

Value of var variable: 20

Address stored in ip variable: 0xbfc601ac

Value of *ip variable: 20

Pointer Arithmetic

The NULL pointer is a constant with a value of zero defined in several standard libraries, including iostream.

```
int *ptr = NULL;
```

There are four arithmetic operators that can be used on pointers: ++, --, +, -

Incrementing a Pointer:

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

```
#include <iostream.h>
using namespace std;
const int MAX = 3;
int main ()
{
int var[MAX] = {10, 100, 200};
int *ptr;
// let us have array address in pointer.
ptr = var;
for (int i = 0; i < MAX; i++)
{
cout << "Address of var[" << i << "] = ";
cout << ptr << endl;
cout << "Value of var[" << i << "] = ";
cout << *ptr << endl; // point to the next location
ptr++;
}
return 0;         }
```

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main ()
{
int var[MAX] = {10, 100, 200};
int *ptr;
// let us have address of the last element in pointer.
ptr = &var[MAX-1];
for (int i = MAX; i > 0; i--)
{
cout << "Address of var[" << i << "] = ";
cout << ptr << endl;
cout << "Value of var[" << i << "] = ";
cout << *ptr << endl; // point to the previous location
ptr--;
}
return 0;
}
```

Pointer to a pointer

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

Declaration

int **var;

Passing Pointers to Functions

C++ allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Example:

```
#include <iostream>
#include <ctime>
using namespace std;
void getSeconds(unsigned long *par);

int main ()
{
unsigned long sec;
getSeconds( &sec );
// print the actual value
cout << "Number of seconds :" << sec << endl;
return 0;
}
void getSeconds(unsigned long *par)
{
// get the current number of seconds
*par = time( NULL );
return;
}
```

Return Pointer from Functions

We have to declare a function returning a pointer as in the following example:

```
int * myFunction()
{
. . .
}
```

Example

```
#include <iostream.h>
#include <ctime.h>
using namespace std;
// function to generate and return random numbers.

int * getRandom( )
{
static int r[10];
// set the seed
srand( (unsigned)time( NULL ) );
for (int i = 0; i < 10; ++i)
{
r[i] = rand();
cout << r[i] << endl;
}
return r;
}

// main function to call above defined function.
int main ()
{
int *p;  // a pointer to an int.
p = getRandom();
for ( int i = 0; i < 10; i++ )
{
cout << "*(p + " << i << ") : ";  cout << *(p + i) << endl;  }
return 0; }
```

1.12 REFERENCES

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the

variable. A variable can be declared as reference by putting '&' in the declaration.

C++ References vs Pointers:

References are often confused with pointers but three major differences between references and pointers are:

- We cannot have NULL references. We must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

```
int i = 17;
int& r = i;
```

Read the & in these declarations as **reference**.

Ex1:

```
#include<iostream>
using namespace std;
void swap (int& first, int& second)
{
   int temp = first;
   first = second;
   second = temp;
}

int main()
{
   int a = 2, b = 3;
   swap( a, b );
   cout << a << " " << b;
   return 0;
}
```

Output: 3 2

Ex2:

```
#include<iostream>
using namespace std;
int main()
{
  int x = 10;            // ref is a reference to x.
  int& ref = x;          // Value of x is now changed to 20
  ref = 20;
  cout << "x = " << x << endl ;            // Value of x is now changed to 30
  x = 30;
  cout << "ref = " << ref << endl ;
  return 0;
}
```

Output:

```
x = 20
ref = 30
```

Returning reference from a function

A function can also be defined to return a reference to a primitive type. When declaring such a function, you must indicate that it is returning a reference by preceding it with the & operator.

The most important rule to apply is that the function must return not only a reference but a reference to the appropriate type. When returning the value, don't precede the name of the variable with &. Here is an example:

When calling a function that returns a reference, you can proceed as if it returns a regular value but of the appropriate type. Here is an example:

```
//-------------------------------------------------------------------------
#include <iostream>
using namespace std;
double & GetWeeklyHours()
{
   double h = 46.50;
```

```
    double &hours = h;
    return hours;
}
//------------------------------------------------------------------------
int main()
{
    double hours = GetWeeklyHours();
    cout << "Weekly Hours: " << hours << endl;
    return 0;
}
//------------------------------------------------------------------------
```

This would produce:

```
Weekly Hours: 46.5
```

1.13 ROLE OF 'THIS' POINTER

Every object in C++ has access to its own address through an important pointer called **this** pointer. The '**this'** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a '**this'** pointer, because friends are not members of a class. Only member functions have a '**this'** pointer.

The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is a constant pointer that holds the memory address of the current object. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

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

/* local variable is same as a member's name */
```

```
class Test
{
private:
  int x;
public:
  void setX (int x)
  {
     // The 'this' pointer is used to retrieve the object's x
     // hidden by the local variable 'x'
     this->x = x;
  }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
  return 0;
}
```

Output:

x = 20

1.14 STORAGE CLASSES

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

- auto
- register

- static
- extern
- mutable

- **The auto Storage Class**

The **auto** storage class is the default storage class for all local variables.

```
{
int mount;
auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

- **The register Storage Class**

The **register** storage class is used to define local variables that should be stored in a register instead of RAM memory.

```
{
register int miles;
}
```

The register should only be used **for variables that require quick access** such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it might be stored in a register depending on hardware and implementation restrictions.

- **The static Storage Class**

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include<iostream>
// Function declaration
void func(void);
static int count =10;/* Global variable */
main()
{
        while(count--)
        {
                func();
        }
return0;
}
// Function definition
void func(void)
{
staticint i =5;// local static variable
i++;
std::cout <<"i is "<< i ;
std::cout <<" and count is "<< count << std::endl;
}
```

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

```
i is6and count is9
i is7and count is8
i is8and count is7
i is9and count is6
i is10and count is5
i is11and count is4
i is12and count is3
i is13and count is2
i is14and count is1
```

i is15and count is0

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.cpp

```
#include<iostream>
int count ;
externvoid write_extern();
main()
{
count =5;
write_extern();
}
```

Second File: write.cpp

```
#include<iostream>
externint count;
void write_extern(void)
{
std::cout <<"Count is "<< count << std::endl;
}
```

Here, extern keyword is being used to declare count in another file. Now, compile these two files as follows:

```
$g++ main.cpp write.cpp -o write
```

This will produce **write** executable program, try to execute **write** and check the result as follows:

```
$./write                    5
```

The mutable Storage Class

It allows a member of an object to override constness. That is, a mutable member can be modified by a const member function.

1.15 FUNCTION AS ARGUMENTS

A function pointer points to executable code within memory and can be used to invoke the function it points to and pass it arguments just like a normal function call. Such an invocation is also known as an "indirect" call, because the function is being invoked indirectly through a variable instead of directly through a fixed name or address. Function pointers can be used to simplify code by providing a simple way to select a function to execute based on run-time values.

A C++ typical use of "pointers to functions" is for passing a function as an argument to another function.

```
// Pointer to functions
 #include <iostream.h>
using namespace std;
int add(int first, int second)
{
   return first + second;         }
int subtract(int first, int second)
{
   return first - second;         }
int operation(int first, int second, int (*functocall)(int, int))
{
   return (*functocall)(first, second);                              }
int main()
{
   int  a, b;
   int  (*plus)(int, int) = add;
   int (*minus)(int, int) = subtract;
    a = operation(7, 5, plus);
   b = operation(20, a, minus);
   cout << "a = " << a << " and b = " << b << endl;
   return 0;             }
```

UNIT-II

OBJECT ORIENTED PROGRAMMING CONCEPTS

String Handling – Copy Constructor - Polymorphism – compile time and run time polymorphisms – function overloading – operators overloading – dynamic memory allocation - Nested classes - Inheritance – virtual functions.

2.1 String Handling

A string is a sequence of characters. The standard library string provides string manipulation operations such as subscripting, assignment, comparison , appending, concatenation and searching for substrings. From C, C++ inherited the notion of strings as zero terminated arrays of char and a set of functions for manipulating such C style strings.

C++ provides following two types of string representations:

- The C-style character string.
- The string class type introduced with Standard C++.
- The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a nullcharacter '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.
- The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

  ```
  char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
  ```

- If you follow the rule of array initialization, then you can write the above statement as follows:

  ```
  char greeting[] = "Hello";
  ```

- Following is the memory presentation of above defined string in C/C++:

C++ supports a wide range of functions that manipulate null-terminated strings:

S.No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

A s t r i n g can be initialized by a C style string, by another s t r i n g , by part of a C style string, by part of a

s t r i n g , or from a sequence of characters. However, a s t r i n g cannot be initialized by a character or an integer:

```
v o i d f (c h a r * p ,v e c t o r <c h a r >&v )
{
s t r i n g  s 0 ; // the empty string
s t r i n g s 0 0 = " "; // also the empty string
s t r i n g s 1 = ´a ´; // error: no conversion from char to string
s t r i n g s 2 = 7 ; // error: no conversion from int to string
s t r i n g s 3 (7 ); // error: no constructor taking one int argument
s t r i n g s 4 (7 ,´a ´); // 7 copies of 'a'; that is "aaaaaaa"
s t r i n g s 5 = "F r o d o "; // copy of "Frodo"
s t r i n g s 6 = s 5 ; // copy of s5
s t r i n g s 7 (s 5 ,3 ,2 ); // s5[3] and s5[4]; that is "do"
s t r i n g s 8 (p +7 ,3 ); // p[7], p[8], and p[9]
s t r i n g s 9 (p ,7 ,3 ); // string(string(p),7,3), possibly expensive
s t r i n g s 1 0 (v .b e g i n (), v .e n d ()); // copy all characters from v   }
```

The l e n g t h () of a string is simply a synonym for its s i z e (); both functions return the number of characters in the string.

String Errors:

Most string operations take a character position plus a number of characters. A position larger than the size of the string throws an out of range exception.

```
void f()
{
String s="Snobol4";
String s2(s, 100, 2);// character position beyond end of string: throw out□of□range()
String s3(s,2,100);// character count too large: equivalent to s3(s,2,s.size()– 2)
String s4(s,2,string::npos);// the characters starting from s[2]
}
```

Example for string manipulation

```
#include <iostream>
#include <cstring>
using namespace std;
int main ()
{
char str1[10] = "Hello";
char str2[10] = "World";
char str3[10];
int len ;
// copy str1 into str3
strcpy( str3, str1);
cout << "strcpy( str3, str1) : " << str3 << endl;
// concatenates str1 and str2
strcat( str1, str2);
cout << "strcat( str1, str2): " << str1 << endl;
// total lenghth of str1 after concatenation
len = strlen(str1);
cout << "strlen(str1) : " << len << endl;
```

```
// compare string
int status = strcmp(str1,str2);
if (status==0)
cout<< “strings are equal”<<endl;
elseif(status<0)
cout<<”str1 is lesser than str2;
else
cout<<”str1 is greater than str2;
// string reverse
Cout<<strrev(str1);
//string conversion
Cout<<strlwr(str1);
Cout<<strupr(str2);
return 0;
}
```

OUTPUT

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
strcmp(str1,str2): str1 is lesser than str2
strrev(str1): olleH
strlwr(str1): hello
strupr(str2): WORLD
```

The String Class in C++:

The standard C++ library provides a **string** class type that supports all the string operations.

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
string str1 = "Hello";
```

```
string str2 = "World";
string str3;
int len ;
// copy str1 into str3
str3 = str1;
cout << "str3 : " << str3 << endl;
// concatenates str1 and str2
str3 = str1 + str2;
cout << "str1 + str2 : " << str3 << endl;
// total lenghth of str3 after concatenation
len = str3.size();
cout << "str3.size() : " << len << endl;
return 0;
}
```

OUTPUT

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() : 10
```

2.1.1 String Assignment:

When one string is assigned to another, the assigned string is copied and two separate strings with the same value exist after the assignment.

For example:

```
v o i d g ()
{
s t r i n g s 1 = "K n o l d ";
s t r i n g s 2 = "T o t ";
s 1 = s 2 ; // two copies of "Tot"
s 2 [1 ] = ´u ´; // s2 is "Tut", s1 is still "Tot"
}
```

Assignment with a single character to a string is supported even though initialization by a single
character isn't:

```
v o i d f ()
{
s t r i n g s = ´a ´; // error: initialization by char
s = ´a ´; // ok: assignment
s = "a ";
s = s ;
}
```

2.1.2 Comparison

Strings can be compared to strings of their own type and to arrays of characters with the same character type

Example:

```
i n t c m p _ n o c a s e (c o n s t s t r i n g & s , c o n s t s t r i n g & s 2 )
{
    s t r i n g :: c o n s t _ i t e r a t o r p = s .b e g i n ();
    s t r i n g :: c o n s t _ i t e r a t o r p 2 = s 2 .b e g i n ();
    w h i l e (p !=s .e n d () && p 2 !=s 2 .e n d ())
    { i f (t o u p p e r (*p )!=t o u p p e r (*p 2 ))
    r e t u r n (t o u p p e r (*p )<t o u p p e r (*p 2 )) ? 1: 1 ;
    ++p;
    ++p 2 ;
    }
     r e t u r n (s 2 .s i z e ()==s .s i z e ()) ? 0 : (s .s i z e ()<s 2 .s i z e ()) ? 1
    : 1 ; // size is unsigned }
```

2.1.3 Insert

Appending and for inserting characters before some character position.

The += operator is provided as the conventional notation for the most common forms of append.

For example:

```
string complete_name (const string&first_name,const string&family_name)
{
strings=first_name;
s+=´´;
s+=family_name;
return s;
}
```

Appending to the end can be noticeably more efficient than inserting into other positions. For example:

```
string complete_name2(const string&first_name,const string&family
_name)
{
strings=family_name;
s.insert(s.begin(),´´);
return s.insert(0,first_name);
}
```

2.1.4 Concatenation

Concatenation – constructing a string out of two strings by placing one after the other – is provided by the + operator.

For example:

```
stringconcate(conststring&first_name,conststring&family_name)
{
return(first_name+´´+family_name);
}
```

2.1.5 Find

Finding substrings in the existing string.

```
voidf()
{
```

```
strings="accdcde";
string::size_typei1=s.find("cd"); //i1=2 s[2]==’c’ && s[3]==’d’
string::size_typei2=s.rfind("cd");          //i2=4 s[4]==’c’ && s[5]==’d’
string::size_typei3=s.find_first_of("cd");       //i3=1 s[1]==’c’
string::size_typei4=s.find_last_of("cd");        //i4=5 s[5]==’d’
string::size_typei5=s.find_first_not_of("cd");                  //i5=0 s[0]!=’c’ && s[0]!=’d’
string::size_typei6=s.find_last_not_of("cd");                   //i6=6 s[6]!=’c’ && s[6]!=’d’
        }
```

2.1.6 Replace

Once a position in a string is identified, the value of individual character positions can be changed using subscripting or whole substrings can be replaced with new characters using r e p l a c e ():

```
voidf()
{
strings="butIhaveheardit worksevenifyoudon´tbelieve in it";
s.erase(0,4);// erase initial "but"
s.replace(s.find("even"),4,"only");
s.replace(s.find("don´t"),5," ");// erase by replacing with ""
        }
```

Resultant sting S is “I h a v e h e a r d i t w o r k s e v e n i f y o u b e l i e v e i n i t”

2.2 Copy Constructor

2.2.1 Copy constructor

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object by passing it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

Example : Write a C++ program to initialize one object from another of the same type

```
#include<iostream.h>
class rect
{
   int length,breadth;
  public:
     rect(int l,int b)                    //  Constructor
       {
          length =l;
          breadth=b;
       }
     void showvalue()
        {
           cout<< "length is:"<< length
           cout<< "breadth is:"<< breadth;
       }
};
int main( )
{
   rect r1(10,20);
   rect r2=r1;
   r1.showvalue ( );
   r2.showvalue ( );
}
```

Example Write a C++ program to Copy an object by passing it as an argument to a function.

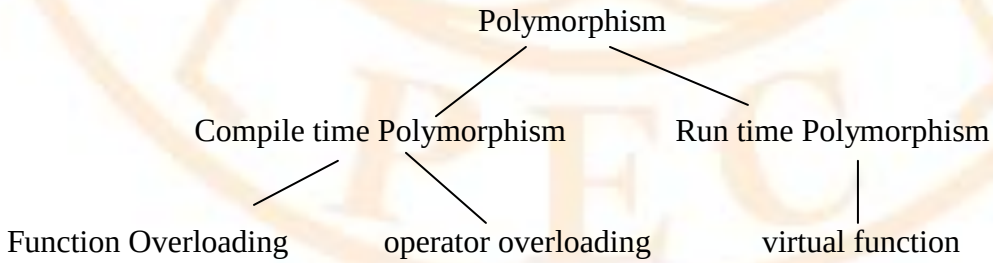
```
#include<iostream.h>
class rect
{
   int length,breadth;
  public:
     rect(int &p)                    //  Constructor
```

```
    {
       length =p.l;
       breadth=p.b;
    }
   void showvalue()
    {
       cout<< "length is:"<< length
       cout<< "breadth is:"<< breadth;
    }
};
int main( )
{
   rect r1(10,20);
   rect r2(r1);
   r1.showvalue ( );
   r2.showvalue ( );
}
```

2.3 Polymorphism

"Poly" means "many" and "morph" means "form". Polymorphism is the ability to take more than one form. Polymorphism is implemented using overloaded functions and operators.

Example: function overloading, function overriding, virtual functions.



2.4 Compile time and Runtime polymorphisms

2.4.1 Compile time Polymorphism / Static Polymorphism

The overloaded member function are selected for invoking by matching arguments, both type and member. This information is known to the complier at the compile time and therefore complier is

able to select the appropriate function for a particular call at the compile time itself. This is called early binding or static linking or static binding also called as **Compile time Polymorphism.**

Consider the following function declaration:

```
void add(int , int);
void add(float, float);
```

When the add() function is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

2.4.2 Runtime polymorphisms / Dynamic polymorphism

If the appropriate member function could be selected while the program is running, this is called run time polymorphism. C++ supports a mechanism known as virtual function to achieve run time polymorphism.

2.5 Function overloading

Function overloading is a feature of C++ that allows us to create multiple functions with the same name, but with different parameters types. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.

Assigning one or more function body to the same name is known as **function overloading or function name overloading.**

Example 1:

```
#include< iostream.h>
#include< conio.h>
int area(int);
int area(int,int);
float area(float);
void main()
{
clrscr();
        cout< < " Area Of Square: "< < area(4);
        cout< < " Area Of Rectangle: "< < area(4,4);
```

```
        cout< < " Area Of Circle: "< < area(3.2);
        getch();
}
int area(int a)
{
        return (a*a);   }
int area(int a,int b)
{
        return(a*b);    }
float area(float r)
{
        return(3.14 * r * r);    }
```

Example 2 : Function overloading in a class

```
#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 3:

// Multiple swap function , function overloading.

```
#include<iostream .h>
// swap for character.
Void swap( char & x, char & y)
{
chat t;
t= x;
x=y;
y=t;
}
// swap for integer.
Void swap(int & x, int & y)
{
int t;
t= x;
x=y;
y=t;     }
// swap for float
Void swap(float & x, float & y)
{
float t;
t= x;
x=y;
```

```
y=t;     }
Void main( )
{
char  ch1,ch2;
cout<< “Enter two characters”;
cin>>ch1>>ch2;
swap(ch1,ch2);
cout<<” After swapping”<<ch1<<”  “<<ch2;
int  ch1,ch2;
cout<< “Enter two interger values”;
cin>>ch1>>ch2;
swap(ch1,ch2);
cout<<” After swapping”<<ch1<<”  “<<ch2;
float  ch1,ch2;
cout<< “Enter two floats”;
cin>>ch1>>ch2;
swap(ch1,ch2);
cout<<” After swapping”<<ch1<<”  “<<ch2;
}
```

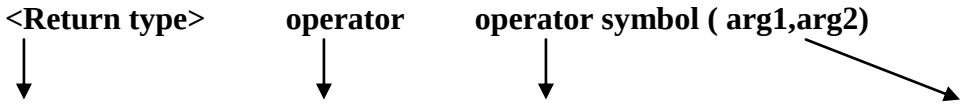
Note:

C++ does not permit overloading of function differing only in their return values.

2.6 Operator overloading

A feature in C++ programming which allows programmer to redefine the meaning of operator when they operate on class objects is known as operator overloading. Overloaded operators are functions with special names. The keyword ‘operator’ followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list. To overload an operator, an operator function inside the class is defined as follows:

Defining operator overloading




```
class classname
{
….. ….. …..
public:
  return_type operator sign(arguments) // operator overloading
  {
    ……. …….
  }
…… ……..
};
```

The return_type comes first which is followed by keyword **operator**, followed by operator sign and finally the arguments is passed. Then, inside the body the task to be performed can be specified. This operator function is called when the operator (sign) operates on the object of that class classname.

Following is the list of operators which can be overloaded

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	Delete	delete []

Operators that cannot be overloaded:

.	(Member Access or Dot operator)
?:	(Ternary or Conditional Operator)
::	(Scope Resolution Operator)
.*	(Pointer-to-member Operator)
sizeof	(Object size Operator)
typeid	(Object type Operator)

Example 1

Write a c++ program to illustrate the concept of Unary operator overloading in C++ Programming

```
#include <iostream.h>
using namespace std;
class temp
{
  private:
    int count;
  public:
    temp()
    {
      count =5;                }
    void operator ++()  // operator overloading
    {
     count=count+1;
    }
    void Display() { cout<<"Count: "<<count; }
};
int main()
{
   temp t;
```

```
    ++t;       /* operator function void operator ++() is called */
    t.Display();
    return 0;
}
```

Example 2

Write a c++ program to illustrate the concept of Unary operator overloading in C++ Programming

```
#include <iostream>
using namespace std;
class Distance
{           private:
    int feet; // 0 to infinite
    int inches; // 0 to 12
    public:
    // required constructors
    Distance(int f, int i){
    feet = f; inches = i;
    }
    // method to display distance
    void displayDistance()
    {
    cout << "F: " << feet << " I:" << inches <<endl;
    }
    // overloaded minus (-) operator
    Distance operator-()
    {
    feet = -feet;
    inches = -inches;
    return Distance(feet, inches);
    }
```

```
};
int main()
{               Distance D1(11, 10), D2(-5, 11);
        -D1; // apply negation
        D1.displayDistance(); // display D1
        -D2; // apply negation
        D2.displayDistance(); // display D2
        return 0;
}
```

Example 3:

Write a c++ program to illustrate the concept of Binary operator overloading in C++ Programming

```
#include<iostream.h>
#include<conio.h>
class complex
{
        int real,imag;
   public:
        void getvalue()
        {
          cout<<"Enter the value of Complex Numbers real & imag:";
          cin>>real>>imag;
              cout<< “The Complex number C1 is” <<real<<”i”<<imag;
        }
        complex operator+(complex C2)    // ‘+’ operator overloaded for complex number addition
        {
                complex temp;
                temp.real=real+C2.real;
                temp.imag=imag+C2.imag;
                return(temp);
```

```
        }
        complex operator-(complex C2)     // '-' operator overloaded for complex number addition
        {
                complex temp;
                temp.real=real-C2.real;
                temp.imag=imag-C2.imag;
                return(temp);
        }
        }
        void display()
        {
                cout<<real<<"+<<"i"<<imag<<"\n";
        }};

void main()
{       clrscr();
        complex C1,C2,C3;
        count<<"enter Complex number C1";
        C1.getvalue();
        count<<"enter Complex number C2";
        C2.getvalue();
        C3=C1+C2      //'+' operator overloaded
        C3.display();
        C3=C1-C2      //'-' operator overloaded
        C3.display();
        getch();
}
```

Rules for overloading operators

1] Only existing operators can be overloaded. New operators cannot created.

2] The overloaded operator must have at least one operand that is of user-defined type.

3] We cannot change the basic meaning of an operator. That is, we cannot redefine the plus (+) operator to subtract one value from the other.

4] Overloaded operators follow the syntax rules of the original operators.

5] There are some operators that cannot be overloaded.

6] We cannot use friend functions to overload certain operator.

Unary operators, overloaded by means of a member function, take no explicit arguments and return no

explicit values. But, those overloaded by means of a friend return (din);

7] function take one reference argument.

8] Binary operators overloaded through a member function take one explicit argument and those which one overloaded through a friend function take two

9] When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.

10] Binary arithmetic operators such as +, - *, and / must explicitly return a value. They must not attempt to change their own arguments.

2.7 Dynamic memory Allocation

The technique of allocating memory during runtime on demand is called dynamic memory allocation. Dynamically allocated space is usually placed in a program segment known as the heap or the free store.

C++ provides two special operators for performing memory management dynamically. They are

(i) new operator for memory allocation

(ii) delete operator for memory deallocation

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below:

```
double* pvalue = NULL;
if( !(pvalue = new double ))
{
cout << "Error: out of memory." <<endl;
exit(1); }
```

The malloc() function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

2.7.1 new operator

The new operator is used to dynamically allocate memory on the heap.

The general syntax of new operator is:

p_var = new typename;

Where

new – It is a keyword

p_var – It is a previously declared pointer of any basic type , typename - It can be any basic data type.

For example

int *p; p=new int;

It allocates memory space for an integer variable.

2.7.2 delete operator

The delete operator is used to deallocate memory on the heap. The general synatax for releasing the memory

is shown below

delete p_var;

where

delete – It is a keyword p_var – pointer variable of any basic type.

write a simple c++ program to illustrate the concept of new and delete operators.

```
#include <iostream>
using namespace std;
int main ()
{
  double *pvalue;
  pvalue  = new double;
```

```
  *pvalue = 29494.99;
  cout << "Value of pvalue : " << *pvalue << endl;
  delete pvalue;
  return 0;
}
```

2.7.3 Dynamic creation of arrays

The new operator can also be used to create an array dynamically. The general syntax is:

p_var = new typename [size];

Where

new – It is a keyword and p_var – It is a previously declared pointer of any basic type

typename - It can be any basic data type.

size-It specifies the length of one-dimensional array to create.

The delete operator is also used to release the memory for arrays. The general syntax is:

delete [ ]p_var; where

delete – It is a keyword

p_var – pointer variable of any basic type.

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

```
char* pvalue = NULL; // Pointer initialized with null
pvalue = new char[20]; // Request memory for the variable
```

To remove the array that we have just created the statement would look like this:

```
delete [] pvalue; // Delete array pointed to by pvalue
```

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows:

```
double** pvalue = NULL; // Pointer initialized with null
pvalue = new double [3][4]; // Allocate memory for a 3x4 array
```

However, the syntax to release the memory for multi-dimensional array will still remain same as above:

delete [] pvalue; // Delete array pointed to by pvalue

2.7.4 Dynamic Memory Allocation for Objects:

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept:

write a c++ program to illustrate the concept of Dynamic memory allocation to objects

```
#include <iostream>
using namespace std;
class Box
{
  public:
    Box() {
      cout << "Constructor called!" <<endl;              }
    ~Box() {
      cout << "Destructor called!" <<endl;       }
};
int main( )
{
  Box* myBoxArray = new Box[4];
  delete [] myBoxArray; // Delete array
  return 0;}
```

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

Output:

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

Constructor called!

Destructor called!

Destructor called!

Destructor called!

Destructor called!

2.8 Nested Classes

Classes can be defined inside other classes. Classes that are defined inside other classes are called nested classes. (Or) **Nested class** is a class defined inside a class that can be used within the scope of the class in which it is defined.

A nested class is declared within the scope of another class. The name of a nested class is local to its enclosing class. Unless you use explicit pointers, references, or object names, declarations in a nested class can only use visible constructs, including type names, static members, and enumerators from the enclosing class and global variables.

Member functions of a nested class follow regular access rules and have no special access privileges to members of their enclosing classes. Member functions of the enclosing class have no special access to members of a nested class. The following example demonstrates this:

Example 1:

```
#include <iostream.h>
class Nest
{
public:
class Display
{
private:
int s;
public:
void sum( int a, int b)
{ s =a+b; }
void show( )
{ cout << "\nSum of a and b is:: " << s;}          }; };
```

```
void main()
{
Nest::Display x;
x.sum(12, 10);
x.show();
}
```

Output:

Sum of a and b is::22

In the above example, the nested class "Display" is given as "public" member of the class "Nest".

Example 2:

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

class student
{
    string name;
    public:
    class Test
          {
        int mark1,mark2;
        public:
          class result
            {
             int total;
             public:
              void display();
             };
            void display();
         };
         void display();
        };

void Student::Test::Result::display()
{
  cout<<"result display called... "<<endl
     <<"Total: "<<total<<endl;
}
```

```
void Student::Test::display()
{
   cout<<"Test display called... "<<endl
      <<"Marks : "<<mark1<<mark2<<endl;
      }

void Student::display()
{
  cout<<"Student display called... "<<endl
     <<"Student's name is: "<<name<<endl;
     }

int main()
{
   Student h;
   Student::Test t;
   Student::Test::Result r;
   h.display();
  t.display();
  r.display();}
```

Example3:

A class can be nested in every part of the surrounding class: in the public, protected or private section. If a class is nested in the public section of a class, it is visible outside the surrounding class. If it is nested in the protected section it is visible in subclasses, derived from the surrounding class, if it is nested in the private section, it is only visible for the members of the surrounding class.

```
class Surround
 {
    public:
       class FirstWithin
       {
          int d_variable;

          public:
             FirstWithin();
             };
    private:
       class SecondWithin
       {
          int d_variable;

          public:
             SecondWithin();
                   };
 };
```

```
inline int Surround::FirstWithin()
{
   return d_variable;
}
inline int Surround::SecondWithin ()
{
   return d_variable;
}
```

Example 4: (Static members)

```
class outside
{
public:
     class nested
     {
     public:
          static int x;
          static int y;
          int f();
          int g();
     };
};
int outside::nested::x = 5;
int outside::nested::f() { return 0; };
```

2.9 Inheritance:

The mechanism that allows us to extend the definition of a class without making any physical changes to the existing class is inheritance. It allows us to create new classes from existing class. Any new class that we create from an existing class is called **derived class**; existing class is called **base class**. Sometimes the base classes are called **super class** and derived classes are called **sub class**. The general syntax for deriving one class from other is shown below:

```
class derivedclass: memberAccessSpecifier baseclass
{
          ...
};
```

Where

derivedclass – It is the name of the derived class

baseclass- It is the name of the class on which it is based.

Member Access Specifier – It may be public, protected or private. This access specifier describes the access level for the members that are inherited from the base class.

The various types of inheritance are:

(i) Single Inheritance
(ii) Multiple inheritance
(iii) Multilevel inheritance
(iv) Hybrid inheritance
(v) Hierarchical inheritance

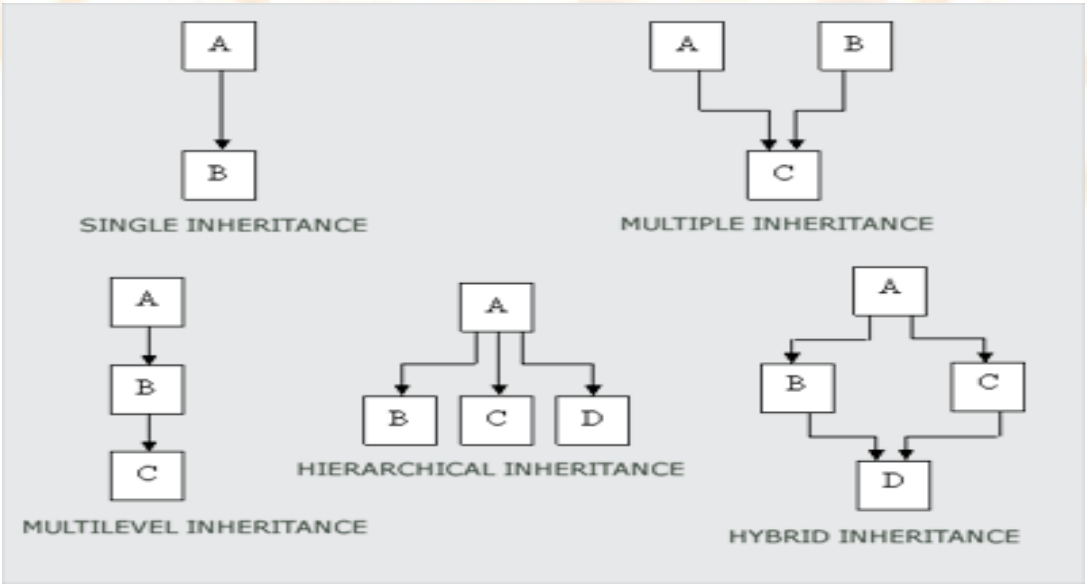
Single Inheritance: It is the inheritance hierarchy wherein one derived class inherits from one base class.

Multiple Inheritances: It is the inheritance hierarchy wherein one derived class inherits from multiple base classes.

Hierarchical Inheritance: It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.

Multilevel Inheritance: It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

Hybrid Inheritance: The inheritance hierarchy that reflects any legal combination of other four types of inheritance.



Derived And Base Class

1. **Base class: -** A base class can be defined as a normal C++ class, which may consist of some data and functions.

Ex .

```
Class Base
{
: : : : :
}
```

2. **Derived Class: -**

i). A class which acquires properties from base class is called derived class.

ii). A derived class can be defined by class in addition to its own detail.

The general form is .

```
Class derived-class-name: visibility-mode base-class-name
{
: : : : :
}
```

The colon indicates that the derived class name is derived from the base-classname.

The visibility-mode is optional and if present it is either private or public.

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es).

A class derivation list names one or more base classes and has the form:

class derived-class: access-specifier base-class Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Access Specifier:

(i) Public Inheritance: When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never

accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

(ii) Protected Inheritance: When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.

(iii) Private Inheritance: When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Examples:

1. Single Inheritance

```
#include<iostream.h>
#include<conio.h>
class emp
{
public:
int eno;
char name[20],des[20];
void get()
{
cout<<"Enter the employee number:";
cin>>eno;
cout<<"Enter the employee name:";
cin>>name;
cout<<"Enter the designation:";
cin>>des;
}
};


class salary:public emp
{
float bp,hra,da,pf,np;
public:
void get1()
{
cout<<"Enter the basic pay:";
cin>>bp;
cout<<"Enter the Humen Resource Allowance:";
cin>>hra;
cout<<"Enter the Dearness Allowance :";
cin>>da;
cout<<"Enter the Profitablity Fund:";
cin>>pf; }
```

```
void calculate()
{
np=bp+hra+da-pf;
}
void display()
{
cout<<eno<<"\t"<<name<<"\t"<<des<<"\t"<<bp<<"\t"<<hra<<"\t"<<da<<"\t"<<pf<<"\t"<<np<<"
\n;
}
};
void main()
{
int i,n;
char ch;
salary s[10];
clrscr();
cout<<"Enter the number of employee:";
cin>>n;
for(i=0;i<n;i++)
{
s[i].get();
s[i].get1();
s[i].calculate();
}
cout<<"\ne_no \t e_name\t des \t bp \t hra \t da \t pf \t np \n";
for(i=0;i<n;i++)
{
s[i].display();
}
getch();
}
```

Output:

Enter the Number of employee:1

Enter the employee No: 150

Enter the employee Name: ram

Enter the designation: Manager

Enter the basic pay: 5000

Enter the HR allowance: 1000

Enter the Dearness allowance: 500

Enter the profitability Fund: 300

E.No E.name des BP HRA DA PF NP

150 ram Manager 5000 1000 500 300 6200

2.Multiple Inheritance

```
#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();
}
```

Output:

```
Enter the Roll no: 100
Enter two marks
90
80
Enter the Sports Mark: 90
Roll No: 100
Total : 260
Average: 86.66
```

3 Hierarchical Inheritance

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

```
{
protected:
int width, height;
public:
void input(int x, int y)
{
width = x;
height = y;
}
};
class rectangle : public polygon
{
public:
int areaR ()
{
return (width * height);
}
};
class triangle : public polygon
{
public:
int areaT ()
{
return (width * height / 2);
}
};
void main ()
{
clrscr();
rectangle rect;
triangle tri;
```

```
rect.input(6,8);
tri.input(6,10);
cout <<"Area of Rectangle: "<<rect.areaR()<< endl;
cout <<"Area of Triangle: "<<tri.areaT()<< endl;
getch();
}
```

Output:

Area of Rectangle: 48

Area of triangle: 30

4. Multilevel Inheritance

```
#include<iostream.h>
#include<conio.h>
class student
{
protected:
int roll;
public:
void get_number(int a)
{
roll = a;
}
void put_number()
{
cout<<"Roll Number: "<<roll<<"\n";
}
};
class test : public student
{
protected:
float sub1;
```

```
float sub2;
public:
void get_marks(float x,float y)
{
sub1 = x;
sub2 = y;
}
void put_marks()
{
cout<<"Marks in Subject 1 = "<<sub1<<"\n";
cout<<"Marks in Subject 2 = "<<sub2<<"\n";
}
};
class result : public test
{
private:
float total;
public:
void display()
{
total = sub1 + sub2;
put_number();
put_marks();
cout<<"Total = "<<total<<"\n";
}
};
void main()
{
clrscr();
result student;
student.get_number(83);
```

```
student.get_marks(99.0,98.5);
student.display();
getch();
}
```

Output:

Roll No:83

Marks in sub 1:99

Marks in sub 2:98.5

Total:197.5

5. Hybrid Inheritance

```
 #include<iostream.h>
#include<conio.h>
class stu
{
protected:
int rno;
public:
void get_no(int a)
{
rno=a;
}
void put_no(void)
{
out<<"Roll no"<<rno<<"\n";
}
};
class test:public stu
{
protected:
float part1,part2;
```

```
public:
void get_mark(float x,float y)
{
part1=x;
part2=y;
}

void put_marks()
{
cout<<"Marks obtained:"<<"part1="<<part1<<"\n"<<"part2="<<part2<<"\n";
}
};
class sports
{
protected
float score;
public:
void getscore(float s)
{
score=s;
}
void putscore(void)
{
cout<<"sports:"<<score<<"\n";
}
};

class result: public test, public sports
{
float total;
public:
```

```
void display(void);
};
void result::display(void)
{
total=part1+part2+score;
put_no();
put_marks();
putscore();
cout<<"Total Score="<<total<<"\n";
}

int main()
{
clrscr();
result stu;
stu.get_no(123);
stu.get_mark(27.5,33.0);
stu.getscore(6.0);
stu.display();
return 0;
}
```

2.10 Virtual function

Virtual function is the member function of a class that can be overriden in its derived class. It is declared with virtual keyword. Virtual function is a member function that is declared within a base class and can be redefined by a derived class. If a function with same name exists in base as well as derived class, then the pointer to the base class would call the functions associated only with the base class. However, if the function is made virtual and the base pointer is initialized with the address of the derived class, then the function in the derived class would be called. Virtual function call is resolved at run-time. So it is also called dynamic binding . Runtime polymorphism is achieved.

Consider following program code:

Example1:

```
Class A
{
        int a;
        public:
        A()
        {               a = 1;
        }
        virtual void show()
        {
                cout <<a;               }
};

Class B: public A
{
         int b;
         public:
         B()
         {              b = 2;
         }

         virtual void show()
         {                  cout <<b;
         }
};


int main()
{
          A *pA;
          B oB;
          pA = &oB;
          pA->show();
          return 0;
}
```

Output is 2 since pA points to object of B and show() is virtual in base class A.

Example 2:

```
#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

Pure Virtual Functions:

Pure virtual functions are a virtual function with no function definition. They are start with **virtual** keyword and end with =0.

It's possible that you'd want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following:

Example1:

```
class Shape
{
protected:
int width, height;
public:
Shape( int a=0, int b=0)
{
width = a;
height = b;
}
// pure virtual function
virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

Example2:

```
class Base
{
public:
virtual void show( ) =0;        //        pure virtual function;
};

class Derived : public base
{
public:

void show( )
{
Cout << "Implementation of virtual function in derived class ";
}
};
```

```
Void main( )
{
Base obj;        // compiler error;
Base *b;
Derived d;
b = &d;
b->show( );    }
```

Output: Implementation of virtual function in derived 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.

```
#include<iostream.h>
#include<conio.h>
 class student
{
   int rno;
  public:
   void getnumber()
   {
            cout<<"Enter Roll No:";
            cin>>rno;       }
void putnumber()
   {
            cout<<"\n\n\tRoll No:"<<rno<<"\n";          }
};

class test:virtual public student
{
  public:
   int part1,part2;
   void getmarks()
   {
            cout<<"Enter Marks\n";
            cout<<"Part1:";
            cin>>part1;
            cout<<"Part2:";
            cin>>part2;   }
   void putmarks()
   {
            cout<<"\tMarks Obtained\n";
            cout<<"\n\tPart1:"<<part1;
```

```
            cout<<"\n\tPart2:"<<part2;
   }      };
class sports:public virtual student
{

  public:
    int score;
    void getscore()
    {
            cout<<"Enter Sports Score:";                    cin>>score;
    }
    void putscore()
    {
            cout<<"\n\tSports Score is:"<<score;
    }               };

class result:public test,public sports
{
    int total;
  public:
    void display()
    {
      total=part1+part2+score;
      putnumber();              putmarks();               putscore();
      cout<<"\n\tTotal Score:"<<total;
    }      };
void main()
{
   result obj;
   obj.getnumber();
   obj.getmarks();
   obj.getscore();
   obj.display();
   getch();                 }
```

Output: Enter Roll No: 200

```
            Enter Marks
            Part1: 90
            Part2: 80
            Enter Sports Score: 80
           Roll No: 200
            Marks Obtained
            Part1: 90
            Part2: 80
            Sports Score is: 80
            Total Score is: 250
```

UNIT III

C++ PROGRAMMING ADVANCED FEATURES

Abstract class – Exception handling - Standard libraries - Generic Programming - templates – class template - function template – STL – containers – iterators – function adaptors – allocators - Parameterizing the class - File handling concepts.

3.1 Abstract Class

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function; otherwise they will also become abstract class.

Characteristics of Abstract Class

- ✓ Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- ✓ Abstract class can have normal functions and variables along with a pure virtual function.
- ✓ Abstract classes are mainly used for **Upcasting**, so that its derived classes can use its interface.
- ✓ Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with "= 0". Here is the syntax for a pure virtual function,

virtual void f() = 0;

Examples of Abstract Class

Example 1:

```
class Base
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};
```

```
class Derived:public Base
{
 public:
 void show()
 { cout<< "Implementation of Virtual Function in Derived class"; }
};
int main()
{          Base obj;      //Compile Time Error. Since no object can be created for Abstract class.
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Output:

Implementation of Virtual Function in Derived class

Example 2:

```
class Base
{
 public:
 virtual void show() = 0;           //Pure Virtual Function
};
void base:: show( )
{
cout<<" Pure Virtual Function Definition";
class Derived: public Base
{
 public:
 void show()
 {
          cout<< "Implementation of Virtual Function in Derived class"; }
          };
```

```
int main()
{
 Base obj;      //Compile Time Error. Since no object can be created for Abstract class.
 Base *b;
 Derived d;
 b = &d;
 b->show();     }
```

Output:

Pure Virtual Function Definition

Implementation of Virtual Function in Derived class

Refer Mastering c++ text book:

Program in Page No: 586-587 (Note: Abstract class can also be defined as classes with atleast one virtual function)

3.2 Exception Handling

The unusual condition could be faults, an error which in turn causes the program to fail is called an exception. Or in other words an exception is a problem that arises during the execution of a program.

Types of an exception:

1.Synchromous exception:

The exception which occur during the program execution, due to some fault in the input-data or techniques not suitable to handle the current class of data, within the program, are known as synchronous exception. Example: divide by zero, out of range, overflow, underflow and so on…

2. Asynchromous exception:

The exception caused by events or faults unrelated to the program and beyond the control of the program is called asynchronous exception. Example unplug power cable, keyboard interrupt, hardware failure, dish failur and so on..

A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

The error handling mechanism of C++ is called as exception handling.

Basics of exception handling:

The error handling mechanism that perform the following task:

1. **Hit exception**: Find the problem
2. **Throw exception:** Inform that an error has occurred
3. **Catch Exception:** Receive the error information.
4. **Handle the exception:** Take corrective actions.

Exception Handling Model:

Exceptions provide a way to transfer control from one part of a program to another.

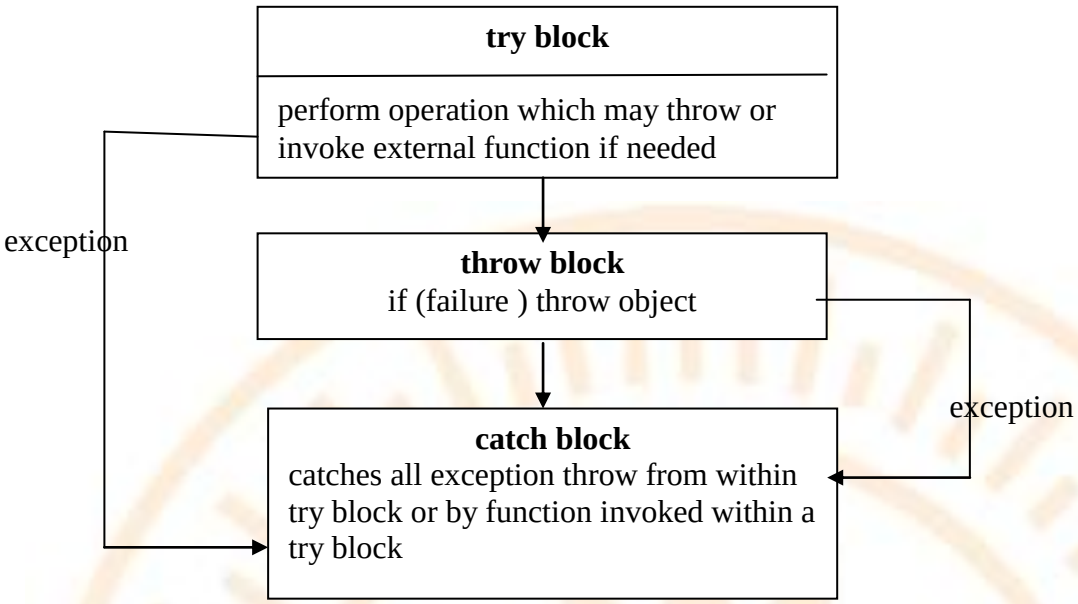
C++ provides three language constructs or blocks to implement exception handling:

- ✓ Try blocks
- ✓ Catch blocks
- ✓ Throw expressions

throw: A program throws an exception when a problem shows up. This is done using a throw keyword.

catch: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.



Syntax: (throw in try block)

…….

……..

try

{

……..

…..

throw exception

……

…….

}

Catch(type arg)

{

……

……

}

……

Syntax: (throw in function)

```
Type function name(arg list)
{
……..
Throw(object);
……
}
…….
……..
try
{
……..
…..
//invoke function
……
…….
}
Catch(type arg)
{
……
……
}
```

Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using throw statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Syntax: throw T; where T is a named object, nameless object, or by default nothing.

```
throw exception;
throw (exception);
throw;                    // rethrow
```

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b)
{
  if( b == 0 )
  {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

Catching Exceptions:

A catch block follows immediately after a try statement or immediately after another catch block that catches the exception. The exception handler is declared with the catch keyword immediately after the closing brace of the try block. The syntax for catch is similar to a regular function with one parameter. we can specify what type of exception we want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
Catch (T)
{
//        Actions for handling exception;
}
```

Where T is a object name or nameless object (same as throw argument)

The following is an example, which throws a division by zero exception and we catch it in catch block.

Example1:

```
// throw in try block
#include <iostream>
using namespace std;
int main ()
{
  int x = 50;
```

```
int y = 0;
double z = 0;
 try
        {
         if(y==0)
        {
                Throw(y)
        }
        else
        {
        z=x/y;
        }
        }
        catch(int i)
        {
        cout<<" Divide by zero error "<<endl";
        }
        Cout<<"END";
        }

        Output:
        Divide by zero error
        END
```

Example2:

```
//throw in function
#include <iostream>
using namespace std;

double division(int a, int b)
{
```

```
  if( b == 0 )
  {
    throw (b);
  }
  return (a/b);
}

int main ()
{
  int x = 50;
  int y = 0;
  double z = 0;

  try {
   z = division(x, y);
   cout << z << endl;
  }
  catch (int i) {
   cout << "Divide by zero error" << endl;
  }

  return 0;
}
```

Multiple Exception

Multiple catch statements can be defined to catch different type of exceptions in case a try block raises more than one exception in different situations.

```
try
{
  // try block
}
```

```
catch( type1 arg )
{
  // catch block1
}
catch(type2 arg)
{
  // catch block2
}
……
…….
catch(typeN arg)
 {
  // catch blockN
}
```

When an exception is thrown, the exception are searched in order for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last catchblock for that try. When no match is found, the program is terminated.

Example

```
#include<iostream>
void test (int x)
{
        try
        {
        if(x==1) throw x;       // int
        else if(x==0) throw 'x';        // char
        elseif (x== -1) throw 1.0; // double
        cout<< "End of try block";
        }
```

```
catch( int m)
{
cout<<"Caught an integer";
}
catch( char c)
{
cout<<"Caught a character";          }
catch( double d)
{
cout<<"Caught a double";
}
cout<<" End of try- catch block";
}
void main()
{
test(1);
test(0);
test(-1);
test(2);
}
```

Output:

```
Caught an integer
End of try- catch block
Caught a character
End of try- catch block
Caught a double
End of try- catch block
End of try block
End of try- catch block
```

Catch all exception

Catch construct catches all the exception instead of a certain type alone. This could be achieved by defining the catch statement using ellipse as follows.

```
catch(…)
{
//Catching all exception
}
```

Example:

```
#include<iostream>
void test (int x)
{
        try
        {
        if(x==1) throw x;       // int
        if(x==0) throw 'x';     // char
        if (x== -1) throw 1.0; // double
        }
catch(…)
{
cout<< "exception  is caught";
}

void main()
{
test(1);
test(0);
test(-1);
}
```

Output:

exception is caught

exception is caught

exception is caught

Rethrowing an Exception

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression, throw with no argument, causes the originally thrown object to be rethrown.

When an exception is rethrown, it will not be caught by the same catch statement or any other catch in that group. Rather it will be caught by the appropriat catch in the outer try/ catch sequence only.

A catch handler itself may detect and throw an exception. It will be passed on to the next outer try/ catch sequence processing.

```
#include<iostream>

double division(int a, int b)
{
try
{
  if( b == 0 )
  {
    throw (b);
  }
  return (a/b);
}
catch(int)
{
cout<<"caught exception inside the function";
throw;            //rethrow
}
cout<<"End of function";      }
```

```
int main ()
{
try
{
    division(10,2);
    division(20,0);      }
catch (int i) {
    cout << "Divide by zero error" << endl;
  }

  return 0;
}
```

Output:

5

End of function

Divide by zero error

Refer Mastering c++ text book for other programs:

Exception handling using abstract class: Page No: 708-709

Exception due to out of array boundaries: 710- 711.

3.3 Standard libraries

C++ Standard Library

The C++ Standard Library can be categorized into two parts:

The Standard Function Library: This library consists of general-purpose,stand-alone functions that are not part of any class. The function library is inherited from C.

The Object Oriented Class Library: This is a collection of classes and associated functions.

The Standard Function Library:

The standard function library is divided into the following categories:

1. I/O
2. String and character handling
3. Mathematical
4. Time, date, and localization
5. Dynamic allocation
6. Miscellaneous
7. Wide-character functions

The Object Oriented Class Library:

Standard C++ Object Oriented Library defines an extensive set of classes that provide support for a number of common activities, including I/O, strings, and numeric processing. This library includes the following:

1. The Standard C++ I/O Classes
2. The String Class
3. The Numeric Classes
4. The STL Container Classes
5. The STL Algorithms
6. The STL Function Objects
7. The STL Iterators
8. The STL Allocators
9. The Localization library
10. Exception Handling Classes
11. Miscellaneous Support Library

Utilities library

<cstdlib>	General purpose utilities: program control, dynamic memory allocation, random numbers, sort and search
<ctime>	C-style time/date utilites

Dynamic memory management

<new>	Low-level memory management utilities
<memory>	Higher level memory management utilities

Error handling

<exception>	Exception handling utilities
<stdexcept>	Standard exception objects

Strings library

<cctype>	functions to determine the type contained in character data
<cstring>	various narrow character string handling functions
<string>	std::basic_string class template

Containers library

<array> (since C++11)	std::array container
<vector>	std::vector container
<deque>	std::deque container
<list>	std::list container
<forward_list>	std::forward_list container
<set>	std::set and std::multiset associative containers
<map>	std::map and std::multimap associative containers
<stack>	std::stack container adaptor
<queue>	std::queue and std::priority_queue container adaptors

Algorithms library

<algorithm>	Algorithms that operate on containers

Iterators library

<iterator>	Container iterators

Numerics library

<cmath>	Common mathematics functions
<complex>	Complex number type

Input/output library

<ios>	std::ios_base class, std::basic_ios class template and several typedefs
<istream>	std::basic_istream class template and several typedefs
<ostream>	std::basic_ostream, std::basic_iostream class templates and several typedefs
<iostream>	several standard stream objects
<fstream>	std::basic_fstream, std::basic_ifstream, std::basic_ofstream class templates and several typedefs
<cstdio>	C-style input-output functions

<cstring> (string.h)

strncpy Copy characters from string

strcat Concatenate strings

strncat Append characters from string

strcmp Compare two strings

strncmp Compare characters of two strings

strstr Locate substring

strlen Get string length

<ctime> (time.h)

C Time Library : This header file contains definitions of functions to get and manipulate date and time

information.

time	Get current time
ctime	Convert time_t value to string
strftime	Format time as string

<cmath> (math.h)

C numerics library: Header <cmath> declares a set of functions to compute common mathematical operations and transformations:

cos	Compute cosine
sin	Compute sine
tan	Compute tangent
exp	Compute exponential function
log	Compute natural logarithm
log10	Compute common logarithm
pow	Raise to power
sqrt	Compute square root
ceil	Round up value
floor	Round down value

3.4 Generic Programming

Generic Programming is a programming paradigm for developing **efficient, eliminating redundant code**, **reusable** software libraries. Important features of C++ called templates strengthen this benefits of OOPs and provide great flexibility. Templates support generic programming In C++, **class and function templates** are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency. Generic algorithms in C++ are written using **C++ templates**.

Templates are the foundation of generic programming, which involves writing code in a way that is

independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function.

Types of template:

1. **Function template**
2. **Class template**

Function template:

Template declared for function are called function template. Function template operates on any type of the argument passed to the function defined. It is also called **generic function.** A function can specifies how an individual function can be constructed.

The general form of a template function definition is shown here:

```
template <class type T>
ret-type func-name(parameter list)
{
  // body of function
}
```

Example 1:

The following is the example of a function template that swap two values:

```
#include <iostream>
Template <class T>
void swap( T &x, T &y)
{
        T t;
        t=x;
        x=y;
        y=t;
}
void main( )
{
```

```
char c1,c2;
cout<<" Enter two characters"
cin>>c1>>c2;
swap(c1,c2);
cout<<" the swapped values are"<<c1<<c2;
int c1,c2;
cout<<" Enter two integer values"
cin>>c1>>c2;
swap(c1,c2);
cout<<" the swapped values are"<<c1<<c2;
float  c1,c2;
cout<<" Enter two float values"
cin>>c1>>c2;
swap(c1,c2);
cout<<" the swapped values are"<<c1<<c2;
}
```

Output:

```
Enter two characters   C  Z
the swapped values are Z  C
Enter two integer values  15  32
the swapped values are  32  15
Enter two float values  10.2   2.4
the swapped values are  2.4   10.2
```

Example 2:

The following is the example of a function template that returns the maximum of two values:

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

```
template <typename T>
inline T Max (T a, T b)
{
   return a < b ? b:a;
}
int main ()
{
    int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;
   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;
   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;
  return 0;
}
```

Example3:

```
//function template to sort different types of arrays
#include <iostream>
using namespace std;
//function template for printing array elements
        template <class T>
        void printArray(T a[], int n)
        {
           for(int i = 1; i <= n; i++)
              cout << a[i] << " ";
           cout << endl;
        }
```

```
//function template for sorting array elements
        template <class T>
        void Sort(T a[], int n);
        {
           int i,j;
           T temp;
        for(i=0;i<n;i++)
        {
                for(j=i+1;j<n;j++)
                {
                if(a[i]>a[j])
                {
                        temp=a[i];
                        a[i]=a[j];
                        a[j]=temp;
        }               }
        }       }

        int main(void)
        {
        int a[]={7,3,5,1,8};
        float f[]={19.3, 20.8, 10.3, 35.7, 2.5, 7.3};

        cout << "This program sorts different variable types" << endl;
        Sort(a,5); // sorting an integer array
        Sort(f,6); // sorting an array of floating point values
        printArray(a,5); // printing an array of integer values
        printArray(f,6); //// printing an array of floating point values
        }
```

Overloaded function template:

The function template can also be overloaded with multiple declarations. It may be overloaded either by (other) function of its name or by (other) template function of the same name. The overloaded function must differ either in terms of number of parameters at their type.

```
#include<iostream>

template< class T>
Cout<< data<<endl;   }
void print void print( T data)

{

 ( T data, int n)

{

for (i=0; i<n;i++)
Cout<< data<<endl;   }
void main()
{
print(1);
print(1.5);
print(100,2);
print("OOPs is best",3);
}
```

Output:

1

1.5

100

100

OOPs is best

OOPs is best

OOPs is best

Multiple arguments function template:

Declaration of a function template for function having multiple parameters of different types requires multiple generic arguments.

```
struct sample

{
int x;
double y;
};
template <class T,class U>
void assign( T a, U b,sample & S1)
{
S1.x=a;
S1.y=b;
}
void main()
{
sample  S1;
assign(3,45.1,S1);
}
```

In the above program,

void assign(T a, T b, sample &S1) will show an error. Since a and b are different datatype.

Nesting and recursive of Function calls:

Call a function byitself is called recursive function. Function template can also support for recursive and also nested function.

Class Template:

Classes are declared to operate on different data types. Such classes are called class templates.

we can also define class templates.

The general form of a generic class declaration is shown here:

```
template <class  T>
class class-name
{
.
. // class members; data member and member function;
.
}
```

You can define more than one generic data type by using a comma-separated list.

```
template <class T1, class T2, …..>
class classname
{
// data items of template type T1, T2,…
        T1 a;
        T2 b;
        …
// functions of tempalte argumets T1, T2, …..
void  func1 (T1 a, T2 b)
…
T2 func2 (T2 a, T2 b)…
};
```

Syntax for class template installation or creating objects for the class

Class name < datatype> object name;

Example: stack <int> stack_int; where stack is class and stack_int ia an object accepts only intger value.

Synatx for declaring member function outside of the class

```
template < class T>
return-type class name <T>:: func name(arg list)
{
        Func body;      }
```

example:

```
template < class T>
        T sample <T>:: add(T a, T b)
        {
        return(a+b);      }
```

Example 1: // add and multiply of different data type

```
template <class T>
class calc
{
  public:
    T multiply(T x, T y);
    T add(T x, T y);
};
template <class T>
T calc<T>::multiply(T x,T y)
{
  return x*y;
}
template <class T>
T calc<T>::add(T x, T y)
{
  return x+y;
}
void main()
{
        calc<int> c1; // object of calc class with integer member variable
```

```
        calc<float> f1; // object of calc class with float member variable
                c1.add(2,3);
        c1.multiply(2,3);
        f1.add(2.5, 3.5);
        f1.multiply(2.5,3.5);
}
```

Template arguments:

A template can have character strigs, function names, and constant expressions in addition to template type arguments. Consider the following class templete to explain, how the complier handles the creation of objects using class templates:

```
template < class T, int size>
class myclass
{
T arr[size];          };
```

When the object of the class is created using a statement,

```
        myclass <float, 15> obj1;
```

Then the complier creates the following class.

```
class myclass
{
 float arr[15];
};
```

Example 2: vector class which has a data member which is a pointer to an array type T. The type T can be integer, float, etc depenting on the type of the object created.

Refer program in Mastering C++ page No: 613- 614

Class template with opeartor overloading

The class template can also be declared for a class having operator overloaded member functions. The synatx for daclaring operator oveloaded function is same as class template members and overloaded functions.

STL (Standard Template Library)

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.

STL can be categorized into the following groupings:

Container classes:

Sequences:

- **vector**: Dynamic array of variables, struct or objects. Insert data at the end.
- **deque**: Array which supports insertion/removal of elements at beginning or end of array
- **list**: Linked list of variables, struct or objects. Insert/remove anywhere.

Associative Containers:

- **set** (duplicate data not allowed in set),
- **multiset** (duplication allowed): Collection of ordered data in a balanced binary tree structure. Fast search.
- **map** (unique keys),
- **multimap** (duplicate keys allowed): Associative key-value pair held in balanced binary tree structure.

Container adapters:

- **stack** LIFO
- **queue** FIFO
- **priority_queue** returns element with highest priority.
- **string**: Character strings and manipulation
- **rope**: String storage and manipulation
- **bitset**: Contains a more intuitive method of storing and manipulating bits.
- Operations/Utilities:

iterator: STL class to represent position in an STL container. An iterator is declared to be associated with a single container class type.

algorithm: Routines to find, count, sort, search, ... elements in container classes

auto_ptr: Class to manage memory pointers and avoid memory leaks.

These components have a rich set of pre-defined functions which help us in doing complicated tasks in very easy fashion.

Containers

A container is an object that represents a group of elements of a certain type, stored in a way that depends on the type of container (i.e., array, linked list, etc.).

The interesting detail about the STL containers, is that the STL introduces the idea of a generic type of container, for which the operations and element manipulations are identical regardless the type of container that you are using.

Containers are implemented via template class definitions. This allows us to define a group of elements of the required type. For example, if we need an array in which the elements are simply integers, we would declare it as:

vector<int> values;

Most frequently used containers:

vector	Array
list	Doubly-linked list
slist	Singly-linked list
queue	FIFO (first-in, first-out) structure
deque	Array-like structure, with efficient insertion and removal at both ends
set	Set of unique elements
stack	LIFO (last in, first out) structure

The following program demonstrates **the vector container (a C++ Standard Template)** which is similar to an array with an exception that it automatically handles its own storage requirements in case it grows:

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

```
int main()
{
  // create a vector to store int
  vector<int> vec;
  int i;
  // display the original size of vec
  cout << "vector size = " << vec.size() << endl;
  // push 5 values into the vector
  for(i = 0; i < 5; i++){
    vec.push_back(i);
  }
  // display extended size of vec
  cout << "extended vector size = " << vec.size() << endl;
  // access 5 values from the vector
  for(i = 0; i < 5; i++){
    cout << "value of vec [" << i << "] = " << vec[i] << endl;
  }
  // use iterator to access the values
  vector<int>::iterator v = vec.begin();
  while( v != vec.end()) {
    cout << "value of v = " << *v << endl;
    v++;
  }
  return 0;                }
```

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

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
```

value of vec [3] = 3

value of vec [4] = 4

value of v = 0

value of v = 1

value of v = 2

value of v = 3

value of v = 4

Here are following points to be noted related to various functions we used in the above example:

The push_back( ) member function inserts value at the end of the vector, expanding its size as needed.

The size( ) function displays the size of the vector.

The function begin( ) returns an iterator to the start of the vector.

The function end( ) returns an iterator to the end of the vector.

Most containers provide member-functions to insert or remove elements from the ends. The "front" operations refer to operations performed at the beginning (first element), and the "back" operations at the end (last element). The following member-functions are provided for most containers:

push_front	Inserts element before the first (not available for vector)
pop_front	Removes the first element (not available for vector)
push_back	Inserts element after the last
pop_back	Removes the last element

Also, most containers provide the following member-functions:

empty	Boolean indicating if the container is empty
size	Returns the number of elements
insert	Inserts an element at a particular position
erase	Removes an element at a particular position
clear	Removes all the elements
resize	Resizes the container
front	Returns a reference to the first element
back	Returns a reference to the last element

<array>

Array header

Header that defines the fixed-size array container class:

Array class

Arrays are fixed-size sequence containers: they hold a specific number of elements ordered in a strict linear sequence.

Container properties

- Sequence : Elements in sequence containers are ordered in a strict linear sequence.
- Individual elements are accessed by their position in this sequence.
- Contiguous storage : The elements are stored in contiguous memory locations, allowing constant time random access to elements. Pointers to an element can be offset to access other elements.
- Fixed-size aggregate
- The container uses implicit constructors and destructors to allocate the required space statically. Its size is compile-time constant. No memory or time overhead.

Member functions

Capacity

- **size** : Return size (public member function)
- **max_size**: Return maximum size (public member function)
- **empty**: Test whether array is empty (public member function)

Element access

operator[] : Access element (public member function)

at : Access element (public member function)

front : Access first element (public member function)

back : Access last element (public member function)

data : Get pointer to data (public member function)

<list>

List header : Header that defines the list container class

list

- List (class template)
- Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.

  List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

  They are very similar to forward_list: The main difference being that forward_list objects are single-linked lists, and thus they can only be iterated forwards, in exchange for being somewhat smaller and more efficient. The main drawback of lists and forward_lists compared to these other sequence containers is that they lack direct access to the elements by their position;

Container properties

- ✓ Sequence
  - Elements in sequence containers are ordered in *a* strict linear sequence. Individual elements are accessed by their position in this sequence.
- ✓ Doubly-linked list
  - Each element keeps information on how to locate the next and the previous elements, allowing constant time insert and erase operations before or after a specific element (even of entire ranges), but no direct random access.
- ✓ Allocator-aware

  The container uses an allocator object to dynamically handle its storage needs.

Member functions

(constructor)	Construct list (public member function)
(destructor)	List destructor (public member function)
operator=	Assign content (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
max_size	Return maximum size (public member function)
front	Access first element (public member function)
back	Access last element (public member function)
sort	Sort elements in container (public member function)

reverse Reverse the order of elements (public member function)

<queue>

Queue header

Header that defines the queue and priority_queue container adaptor classes:

std::queue

FIFO queue : **queue**s are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other. **queue**s are implemented as *containers adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed* into the *"back"* of the specific container and *popped* from its *"front"*. The underlying container may be one of the standard container class template or some other specifically designed container class. This underlying container shall support at least the following operations:

empty

size

front

back

push_back

pop_front

The standard container classes deque and list fulfill these requirements. By default, if no container class is specified for a particular queue class instantiation, the standard container deque is used.

Example to show the use of the list containers. This example indirectly illustrate the generic use of the **STL containers.**

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

```
class Student
{
public:
  // ... various functions to perform the required operations

private:
  string name, ID;
  int mark;
};

int main()
{
   list<Student> students;

   // Read from data base

   while (more_students())
   {
     Student temp;
     temp.read();
     students.push_back (temp);
   }

   // Now print the students that failed (mark < 60%) - of
   // course, the particular Student object should provide a
   // member-function (say, passed()) that will determine that

  list<Student>::iterator i;
  for (i = students.begin(); i != students.end(); ++i)
  {
    if (! i->passed())    // iterators also provide operator ->
    {
      cout << "The student " << *i << " failed." << endl;
```

```
        // provided that class Student provides the overloaded
        // stream insertion operator <<
      }
    }

    // Now remove the failed students (of course, this could have been done in the previous loop)

    i = students.begin()
    while (i != students.end())
    {
      if (! i->passed())
      {
        i = students.erase (i);
      }

      else
      {
        ++i;
      }
    }

    // ...

    return 0;
}
```

Iterator:

An iterator is a pointer-like object (that is, an object that supports pointer operations) that is able to "point" to a specific element in the container.

The iterator is container-specific. That is, the iterator's operations depend on what type of container we are using. For that reason, the iterator class definition is inside the container class; this has a good side effect, which is that the syntax to declare an iterator suggests that the definition type belongs in the context of the container definition. Thus, we would declare an iterator for the values object as follows:

```
vector<int>::iterator current;
```

Algorithms:

The Standard Template Library provides a number of useful, generic algorithms to perform the most commonly used operations on groups/sequences of elements. These operations include traversals, searching, sorting and insertion/removal of elements.

The way that these algorithms are implemented is closely related to the idea of the containers and iterators, although, as we will see, they can be used with standard arrays and pointers. (in fact, we could use most of the STL algorithms in a program that uses exclusively C idioms)

Suppose that we want to find a particular value in a linked list (or in a vector, or any other container - it doesn't make any difference). We could do the following: (the example assumes that the values are integers)

```
list<int> values;
int search_value;

// ... code to fill values

list<int>::iterator i;
for (i = values.begin(); i != values.end(); ++i)
{
  if (*i == search_value)
  {
    break;
  }      }

if (i != values.end())
{
```

```
    // Found! Use now *i if necessary
}
else
{
   // Not found! Do whatever is required
}
```

Find function can be defined as follows:

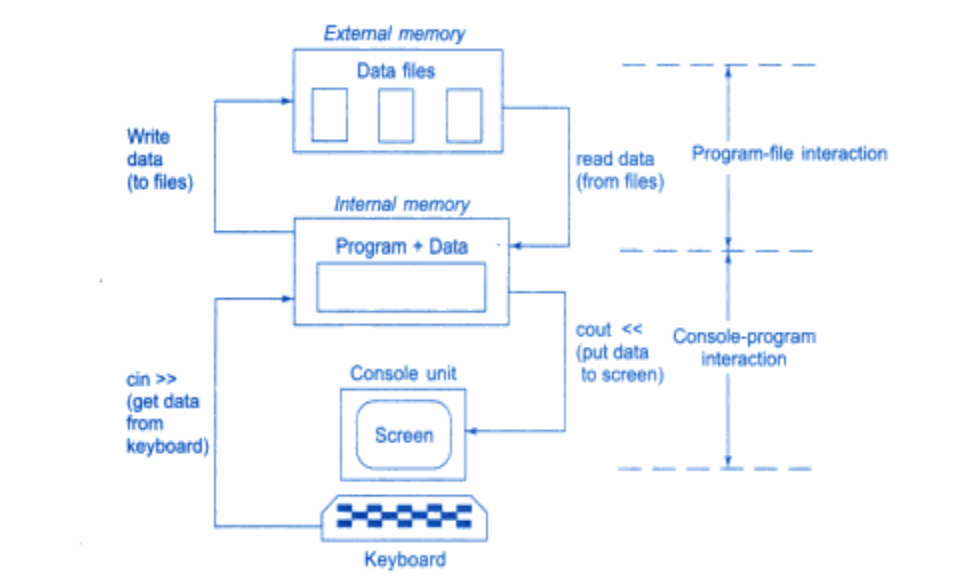
```
 list<int>::iterator find (const list<int> & lst, int search_value)
{
   list<int>::iterator i;
   for (i = lst.begin(); i!= lst.end(); ++i)
   {
      if (*i == search_value)
      {
         break;
      }
   }

   return i; // will return lst.end() if the value was not found
}
```

File Handling concepts:

A file is a collection of related data stored in a particular area on the disk. Program can be designed to perform the read and write operations on these files.

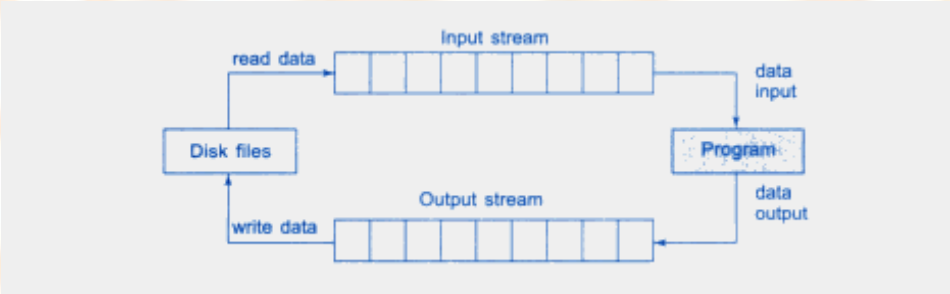
1. Data tranfer between the console unit and the program.
2. Data tranfer between the program and a disk file.

Consol – Program – File interaction



The I/O system of C++ handles file operations which are very similar to the console input and output operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as input stream and the one that receives data from the program is known as output stream.

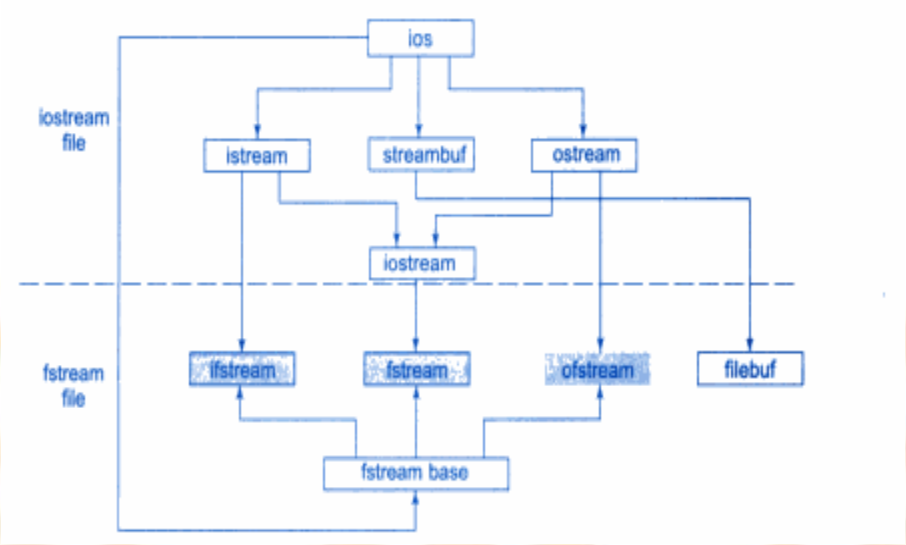
File input and output streams



Classes for file stream operations:

The I/O system of C++ contains a set of classes that define the file handling methods. These include ifstream, ofstream and fstream.

Stream classes for file operations



Opening and closing a file

The following things are needed for the files

Suitable name for the file

Data type and structure

Purpose

Opening method

Details of file stream classes

Class	Contents
`filebuf`	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()** and **open()** as members.
`fstreambase`	Provides operations common to the file streams. Serves as a base for **fstream, ifstream** and **ofstream** class. Contains **open()** and **close()** functions.
`ifstream`	Provides input operations. Contains **open()** with default input mode. Inherits the functions **get(), getline(), read(), seekg()** and **tellg()** functions from **istream**.
`Ofstream`	Provides output operations. Contains **open()** with default output mode. Inherits **put(), seekp(), tellp(),** and **write(),** functions from **ostream**.
`fstream`	Provides support for simultaneous input and output operations. Contains **open()** with default input mode. Inherits all the functions from **istream** and **ostream** classes through **iostream**.

Opening files using open( )

The function open( ) can be used to open multiple files that use the same stream object.

```
file-stream-class stream-object;
stream-object.open ("filename");
```

Example:

```
ofstream outfile;           // Create stream (for output)
outfile.open("DATA1");      // Connect stream to DATA1
.....
.....
outfile.close();            // Disconnect stream from DATA1
outfile.open("DATA2");      // Connect stream to DATA2
.....
.....
outfile.close();            // Disconnect stream from DATA2
.....
.....
```

Detecting End of File

Detection of the end of file condition is necessary for preventing any further attempt to read data from the file.

Example: while(fin)

An ifstream object, such as fin, returns a value 0 if any error occurs in the file operations including the end-of-file condition. The while loop terminates when fin return a value of zero on reaching the end –of-file condition.

There is another approaches to detect the end of file condition.

If (fin1.eof() !=0)

{

exit (1); }

eof is a member function of ios class. It returns a non zero value if the end of the file (EOF) condition is encountered.

ios::in for ifstream functions meaning open for reading only

ios::out for ofstream functions meaning open for writing only

File mode parameters

Parameter	Meaning
ios :: app	Append to end-of-file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: in	Open file for reading only
ios :: nocreate	Open fails if the file does not exist
ios :: noreplace	Open fails if the file already exists
ios :: out	Open file for writing only
ios :: trunc	Delete the contents of the file if it exists

File Pointers and manipulations

Seek call	Action
fout.seekg(o, ios::beg);	Go to start
fout.seekg(o, ios::cur);	Stay at the current position
fout.seekg(o, ios::end);	Go to the end of file
Fout.seekg(m,ios::beg);	Move to (m + 1)th byte in the file
fout.seekg(m,ios::cur);	Go forward by m byte form the current position
fout.seekg(-m,ios::cur);	Go backward by m bytes from the current position
fout.seekg(-m,ios::end);	Go backward by m bytes form the end

Sequential input and output operations

The file stream classes support a number of member functions for performing the input and output operations on files. One pair of functions, put( ), get( ), are designed for handling a single character at a time. Another pair of function, write () and read ( ), are designed to write and read blocks of binary data.

put() and get () functions

The function get( ) reads a single character from the associated stream. On receiving the string, the program writes it, character by character, to the file using the put( ) function in a for Loop

```
#include<iostream>
#include<fstream>
        void main( )
        {
        int len = strlen(string);
        fstream file;
        file.open("Text", ios:: in / ios:: out);
        for( int i=0;i<len ; i++)
        file. put(string[i]);
        file.seekg(0);
        char ch;
        while(file)
        {
        File.get(ch);
        Cout<<ch;
        }
        Return 0;
        }
        Output:
        Enter a string
        Object Oriented Programming
        Object Oriented Programming
```

File Updating :: Random Access

The updating would include one or more of the following tasks:

- Displaying the contents of a file
- Modifying an existing item
- Adding a new item
- Deleting an existing item

```
#include<iostream>
#include<fstream>
#include<iomanip>
```

```
class inventory
{
char name[10];
int code;
float cost;
public:
void getdata(void)
{
cout<<"Enter Name";   cin>> name;
cout<<"enter code"; cin>>code;
cout<<"enter cost"; cin>>cost; }
void putdata(void)
{
cout<<setw(10)<<name<<setw(10)<<code<<setprecision(2)<<setw(10)<<cost<<endl;
}
};
int  main()
{
inventory item;
fstream inoutfile;
inoutfile.open("stock.dat",ios::ate | ios :: in | ios::out | ios:: binary);
inoutfile.seekg(0,ios::beg);
cout<<"Current contents of stock";
while(inoutfile.read((char *) & item, sizeof(item))
{
item.putdata();
}
inoutfile.clear();
}
cout<<" ADD AN Item";
item.getdata();
```

```
char ch;
cin.get(ch);
inoutfile.write((char * ) & item, sizeof(item))
inoutfile.seekg(0);
item.putdata();
}
inoutfile.close();
return 0;
}
```

Error Handling during file operations:

The following things may happen while opening and using the file for reading and writing will be consider as error.

A file which we are attempting to open for reading does not exist.

The file name used for a new file may already exist

We may attempt an invalid operation such as reading past the end – of –file

There may not be any space in the disk for storing more data.

We may use an invalid file name.

We may attempt to perform an operation when the file is not opened for that purpose.

Error handling function

Function	*Return value and meaning*
eof()	Returns *true* (non-zero value) if end-of-file is encountered while reading; Otherwise returns *false*(zero)
fail()	Returns *true* when an input or output operation has failed
bad()	Returns *true* if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is *false*, it may be possible to recover from any other error reported, and continue operation.
good()	Returns true if no error has occurred. This means, all the above functions are false. For instance, if **file.good**() is *true*, all is well with the stream **file** and we can proceed to perform I/O operations. When it returns *false*, no further operations can be carried out.

EXAMPLE FOR STL ALGORITHMS

1. Using the STL generic reverse algorithm with an array

```
#include <iostream>
# rse(&array1[0], &array1[N1]);
  for(iinclude <string>
#include <algorithm> // For reverse algorithm
using namespace std;
int main()
{
  char array1[] = "abc";
  int N1 = strlen(array1);
  revent i=0; i<N1;i++)
          cout<<array1[i];
  return 0;
}
```

2. Use min for char and integer

```
#include <iostream>
using std::cout;
using std::endl;
#include <algorithm>
void main()
{
   cout << "\nThe minimum of 'G' and 'Z' is: " << std::min( 'G', 'Z' );
   cout << "\nThe minimum of 12 and 7 is: " << std::min( 12, 7 );
   cout << endl;
}
```

Output:

```
The minimum of 'G' and 'Z' is: G
The minimum of 12 and 7 is: 7
```

3. Sort a vector and print out the sorted elements

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

int main()
{
   vector<int> v1;
   vector<int>::iterator pos;
   // insert elements from 1 to 6 in arbitrary order
   v1.push_back(2);
   v1.push_back(5);
   v1.push_back(4);
   v1.push_back(1);
   v1.push_back(6);
   v1.push_back(3);
   // sort all elements
   sort (v1.begin(), v1.end());
   // print all elements
   for (pos=v1.begin(); pos!=v1.end(); ++pos) {
      cout << *pos << ' ';
   }
}
```

Output:

1 2 3 4 5 6

4. set_union

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

int main()
{
  bool result;
  string s("abcde");
  string s2("aeiou");
  vector<char> v1(s.begin(), s.end());
  vector<char> v2(s2.begin(), s2.end());
  vector<char> resultset;
  set_union(v1.begin(), v1.end(),v2.begin(), v2.end(),back_inserter(resultset));
  for(int i=0;i<resultset.size();i++)
  {
    cout << resultset[i];          }          return 0;                }
```

Output: abcdeiou

5. Determine elements in one set but not in another set

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{
   const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
   int a1[ SIZE1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int a2[ SIZE2 ] = { 4, 5, 6, 7, 8 };
   int a3[ SIZE2 ] = { 4, 7, 6, 11, 15 };
```

```
  ostream_iterator< int > output( cout, " " );
  cout << "a1 contains: ";
  copy( a1, a1 + SIZE1, output );
  cout << "\na2 contains: ";
  copy( a2, a2 + SIZE2, output );
  cout << "\na3 contains: ";
  copy( a3, a3 + SIZE2, output );
  int set4[ SIZE1 ];
  int *ptr =  set_difference( a1, a1 + SIZE1, a2, a2 + SIZE2, set4 );
  cout << "\n\nset_difference of a1 and a2 is: ";
   copy( set4, ptr, output );
  return 0;
}
```

Output:

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 7 6 11 15
set_difference of a1 and a2 is: 1 2 3 9 10
```

FUNCTION ADAPTORS

- STL adaptors implement the Adapter design pattern
  - i.e., they convert one interface into another interface clients expect
- STL has predefined functor adaptors that **will change their functors** so that they can:
  - Perform function composition & binding
  - Allow fewer created functors
- These functors allow one to combine, transform or manipulate functors with each other, certain values or with special functions
- STL function adapters include
  - Binders (bind1st() & bind2nd()) bind one of their arguments
  - Negators (not1 & not2) adapt functors by negating arguments
  - Member functions (ptr fun & mem fun) allow functors to be class members

STL Binder Function Adaptor

• A binder can be used to **transform a binary functor into an unary one by acting as a converter between the functor & an algorithm**

• Binders always store both the binary functor & the argument internally (the argument is passed as one of the arguments of the functor every time it is called)

– bind1st(Op, Arg) calls 'Op' with 'Arg' as its first parameter

– bind2nd(Op, Arg) calls 'Op' with 'Arg' as its second parameter

Example 1:

```
// bind 1st & bind2nd example
#include <iostream>
#include <functional>
#include <algorithm>
using namespace std;
int main () {
  int numbers[] = {10,-20,-30,40,-50};
  int cx;
  cx = count_if ( numbers, numbers+5, bind2nd(less<int>(),0) );
  cout << "There are " << cx << " negative elements.\n";
  cx = count_if ( numbers, numbers+5, bind1st(less<int>(),0) );
  cout << "There are " << cx << " Positive elements.\n";                return 0;              }
```

Output:

```
There are 3 negative elements.
There are 2 Positive elements.
```

Example 2:

```
// ptr_fun example
#include <iostream>
#include <functional>
#include <algorithm>
#include <cstdlib>
```

```
#include <numeric>
using namespace std;
int main () {
  char* foo[] = {"10","20","30","40","50"};
  int bar[5];
  int sum;
  transform (foo, foo+5, bar, ptr_fun(atoi) );
  sum = accumulate (bar, bar+5, 0);
  cout << "sum = " << sum << endl;
  return 0;
}
```

output:

sum = 150

Example 3:

This example removes spaces in a string that uses the equal to and bind2nd functors to perform remove if when equal to finds a blank char in the string

```
#include <iostream>
#include <string>
using namespace std;
int main() {
string s = "spaces in text";
cout << s << endl;
string::iterator new_end = remove_if (s.begin (), s.end (), bind2nd (equal_to<char>(), ' '));
// remove_if() just moves unwanted elements to the end and returns an iterator
// to the first unwanted element since it is a generic algorithm
s.erase (new_end, s.end ());
cout << s << endl;
return 0;
}
```

Generically, function objects are **instances of a class with member function operator()** defined.

This member function allows the object to be used with the same syntax as a regular function call, and therefore its type can be used as template parameter when a generic function type is expected.

In the case of unary function objects, this operator() member function takes a single parameter. unary_function is just a base class, from which specific unary function objects are derived. It has no operator() member defined (which derived classes are expected to define) - it simply has two public data members that are typedefs of the template parameters.

It is defined as:

```
template <class Arg, class
Result>
 struct unary_function
{
   typedef Arg argument_type;
   typedef Result result_type  };
```

ponter_to_unary_function"

```
template <class Arg, class Result>
 pointer_to_unary_function<Arg,Result> ptr_fun (Result (*f)(Arg));
```

pointer_to_binary_function

```
template <class Arg1, class Arg2, class Result>
 pointer_to_binary_function<Arg1,Arg2, Result> ptr_fun (Result (*f)(Arg1,Arg2)); // Convert function
```

pointer to function object. Returns a function object that encapsulates function f.

Member types

member type	definition	notes
argument_type	The first template parameter (Arg)	Type of the argument in member operator()
result_type	The second template parameter (Result)	Type returned by member operator()

Example4

```
// unary_function example
#include <iostream>     // cout, cin
#include <functional>   // unary_function
 using namespace std;
class IsOdd : public unary_function<int,bool> {
  bool operator() (int number) {return (number%2!=0);}
};

int main () {
  IsOdd IsOdd_object;
  int input;
  bool result;
  cout << "Please enter a number: ";
  cin >> input;
  result = IsOdd_object (input); //using object name as function call
  cout << "Number " << input << " is " << (result?"odd":"even") <<"\n";
  return 0;
}
```

Possible output:

```
Please enter a number: 2
Number 2 is even.
```

Example5

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<functional>
void main()
{
vector<int> v;        // fill v with 4 6 10 3 13 2
int bound = 5;
```

```
v.push_back(4);  v.push_back(6);  v.push_back(10);  v.push_back(3);  v.push_back(13);
v.push_back(2);
replace_if (v.begin(), v.end(), bind2nd (less<int>(), bound), bound);
```

Output: 5 6 10 5 13 5

Example 6:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <cstring>
using namespace std;
int main()
{
  vector<char *> vectorObject;
  vector<char *>::iterator p;
  int i;
  vectorObject.push_back("One");
  vectorObject.push_back("Two");
  vectorObject.push_back("Three");
  vectorObject.push_back("Four");
  vectorObject.push_back("Five");
  cout << "Sequence contains:";
  for(i = 0; i <vectorObject.size(); i++)
    cout << vectorObject[ i ] << " ";
  cout << endl;
  cout << "Searching sequence for Three.\n";
  // use a pointer-to-function adaptor
  p = find_if(vectorObject.begin(), vectorObject.end(), not1(bind2nd(ptr_fun(strcmp), "Three")));
```

```
 if(p != vectorObject.end()) {
   cout << "Found.";
   cout << "Sequence from that point is:";
   do {
     cout << *p++ << " ";    } while (p != vectorObject.end());          }          return 0;        }
```

OUTPUT:

Found

Sequence from that point is: Four Five

Function adaptor : not1

```
template <class Predicate> unary_negate<Predicate> not1(constPredicate& pred)
{
 return unary_negate<Predicate>(pred);
}
```

Example:

```
// binary_negate example
#include <iostream>
#include <functional>   //not1
#include <algorithm>    // count_if
using namespaces std;

struct IsOdd {
 bool operator() (const int& x) const {return x%2==1;}
 typedef int argument_type;
};

int main () {
 int values[] = {1,2,3,4,5};
 int cx = count_if (values, values+5, std::not1(IsOdd()));
```

```
  cout << "There are " << cx << " elements with even values.\n";
  return 0;
}
```

Output:

```
There are 2 elements with even values.
```

Function adaptor : not2:

```
template <class Predicate> binary_negate<Predicate> not2 (const Predicate& pred)
{
  return binary_negate<Predicate>(pred);
}
```

```
// not2 example
#include <iostream>     // std::cout
#include <functional>   // std::not2, std::equal_to
#include <algorithm>    // std::mismatch
#include <utility>      // std::pair



int main () {
  int foo[] = {10,20,30,40,50};
  int bar[] = {0,15,30,45,60};
  std::pair<int*,int*> firstmatch,firstmismatch;
  firstmismatch = std::mismatch (foo, foo+5, bar, std::equal_to<int>());
  firstmatch = std::mismatch (foo, foo+5, bar, std::not2(std::equal_to<int>()));
  std::cout << "First mismatch in bar is " << *firstmismatch.second << '\n';
  std::cout << "First match in bar is " << *firstmatch.second << '\n';
  return 0;
}
```

```
First mismatch in bar is 0
First match in bar is 30
```

Allocators

The std::allocator class template is the default Allocator used by all standard library containers if no user-specified allocator is provided. Allocators are classes that define memory models to be used by some parts of the Standard Library, and most specifically, by STL containers.

Encapsulates a memory allocation and deallocation strategy.

Every standard library component that may need to allocate or release storage, from std::string, std::vector, and every container except std::array, to std::shared_ptr and std::function, does so through an Allocator. Some requirements are optional: the template std::allocator_traits supplies the default implementations for all optional requirements, and all standard library containers and other allocator-aware classes access the allocator through std::allocator_traits, not directly.

member	definition in allocator	represents
value_type	T	Element type
pointer	T*	Pointer to element
reference	T&	Reference to element
const_pointer	const T*	Pointer to constant element
const_reference	const T&	Reference to constant element
size_type	size_t	Quantities of elements
difference_type	ptrdiff_t	Difference between two pointers
rebind<Type>	member class	Its member type other is the equivalent allocator type to allocate elements of type Type

Member functions

(constructor)	Construct allocator object
(destructor)	Allocator destructor
address	Return address
allocate	Allocate block of storage
deallocate	Release block of storage
max_size	Maximum size possible to allocate
construct	Construct an object
destroy	Destroy an object

```
template < >
class allocator<void>
{       public:
  typedef void* pointer;
  typedef const void* const_pointer;
  typedef void value_type;
  template <class U> struct rebind { typedef allocator<U> other; };
```

Example :

```
#include <memory>
#include <iostream>
#include <string>
 int main()
{
    allocator<int> a1; // default allocator for int
    int* a = a1.allocate(10); // space for 10 int
     a[9] = 7;
     cout << a[9] << '\n';
     a1.deallocate(a, 10);
    // default allocator for strings          allocator<string> a2;
     // same, but obtained by rebinding from the type of a1, decltype(a1)::rebind<string>::other a2;
     // same, but obtained by rebinding from the type of a1 via alolcator_traits
    allocator_traits<decltype(a1)>::rebind_alloc<string> a2;
     string* s = a2.allocate(2); // space for 2 string
     a2.construct(s, "foo");
    a2.construct(s + 1, "bar");
     cout << s[0] << ' ' << s[1] << '\n';
     a2.destroy(s);
    a2.destroy(s + 1);
    a2.deallocate(s, 2);                              }
```

Output:

```
7
foo bar
```

UNIT IV
ADVANCED NON-LINEAR DATA STRUCTURES

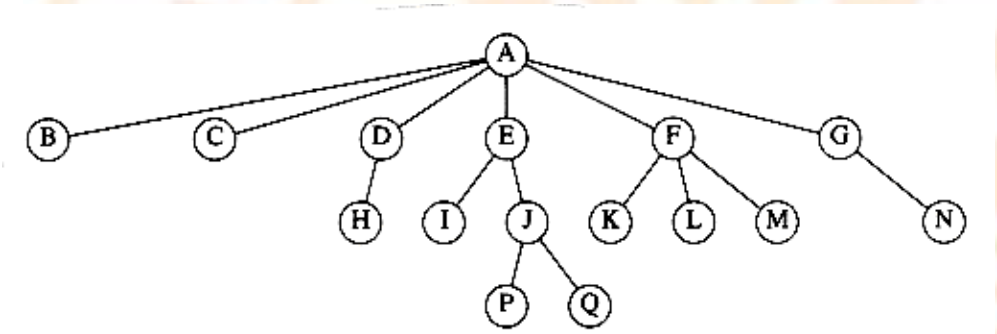
AVL trees – B-Trees – Red-Black trees – Splay trees - Binomial Heaps – Fibonacci Heaps – Disjoint Sets – Amortized Analysis – accounting method – potential method – aggregate analysis.

TREES

Tree is a Non- Linear datastructure in which data are stored in a hierarchal manner. It is also defined as a collection of nodes. The collection can be empty. Otherwise, a tree consists of a distinguished node r, called the root, and zero or more (sub) trees T1, T2, . . . , Tk, each of whose roots are connected by a directed edge to r.

The root of each subtree is said to be a child of r, and r is the parent of each subtree root. A tree is a collection of n nodes, one of which is the root, and n - 1 edges. That there are n - 1 edges follows from the fact that each edge connects some node to its parent and every node except the root has one parent

A tree



Terms in Tree

In the tree above figure, the **root** is A.

- ✓ Node F has A as a **parent** and K, L, and M as children.
- ✓ Each node may have an arbitrary number of children, possibly zero.
- ✓ Nodes with no children are known as **leaves**;
- ✓ The **leaves** in the tree above are B, C, H, I, P, Q, K, L, M, and N.
- ✓ Nodes with the same parent are **siblings**; thus K, L, and M are all siblings. **Grandparent** and **grandchild** relations can be defined in a similar manner.
- ✓ A **path** from node n1 to nk is defined as a sequence of nodes n1, n2, . . . , nk such that ni is the parent of ni+1 for 1 i < k.
- ✓ The **length of this path** is the number of edges on the path, namely k -1.
- ✓ There is a path of length zero from every node to itself.

- ✓ For any node ni, the **depth of ni** is the length of the unique path from the root to ni. Thus, the root is at depth 0.
- ✓ The **height of ni** is the longest path from ni to a leaf. Thus all leaves are at height 0.
- ✓ The height of a tree is equal to the height of the root.

Example: For the above tree, E is at depth 1 and height 2;

F is at depth 1 and height 1; the height of the tree is 3. T

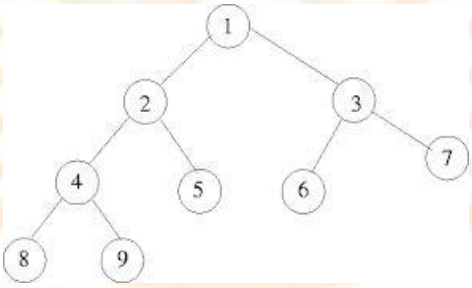
Note:

- ✓ **The depth of a tree is equal to the depth of the deepest leaf; this is always equal to the height of the tree.**
- ✓ **If there is a path from n1 to n2, then n1 is an ancestor of n2 and n2 is a descendant of n1. If n1 n2, then n1 is a proper ancestor of n2 and n2 is a proper descendant of n1.**
- ✓ **A tree there is exactly one path from the root to each node.**

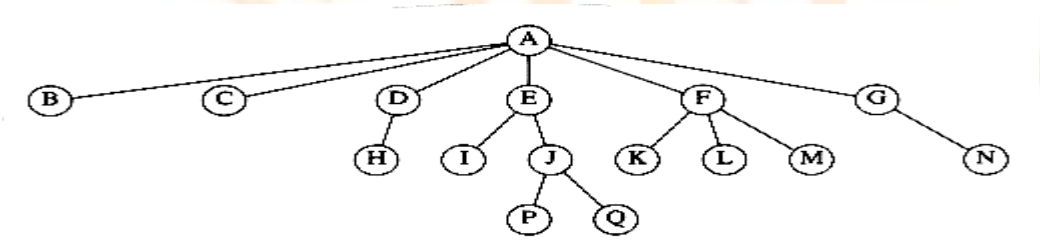
Types of the Tree: Based on the no. of children for each node in the tree, it is classified into two to types.

1. Binary tree
2. General tree

Binary tree : In a tree, each and every node has a maximum of two children. It can be empty, one or two. Then it is called as Binary tree. Eg:



General Tree : In a tree, node can have any no of children. Then it is called as general Tree. Eg:



Tree Traversals

Visiting of each and every node in a tree exactly only once is called as **Tree traversals.** Here Left

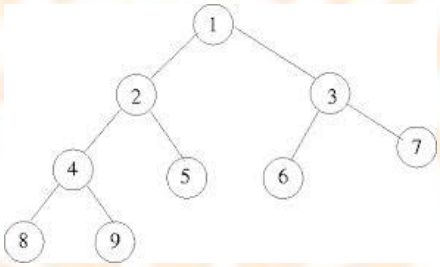
subtree and right subtree are traversed recursively.

Types of Tree Traversal:

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

Inorder traversal: **Rules:**

- Traverse Left subtree recursively
- Process the node
- Traverse Right subtree recursively



Inorder traversal: 8 4 9 2 5 1 6 3 7

Preorder traversal: **Rules:**

- Process the node
- Traverse Left subtree recursively
- Traverse Right subtree recursively

Preorder traversal: 1 2 4 8 9 5 3 6 7

Postorder traversal: **Rules:**

- Traverse Left subtree recursively
- Traverse Right subtree recursively
- Process the node

Postorder traversal: 8 9 4 5 2 6 7 3 1

Routines:

```
//structure for the tree
struct  Tree
{
elementtype element;
tree  * left;
tree  * right;
};
```

```
//routine for Inorder
void inorder( tree *t)
{
If (T!=NULL)
{
inorder(t->left);
print(t->element);
inorder(t->right);
}
}

//routine for preorder
void preorder( tree *t)
{
If (T!=NULL)
{
print(t->element);
preorder(t->left);
preorder(t->right);
}
}
//routine for postorder
void postorder( tree *t)
{
If (T!=NULL)
{
postorder(t->left);
postorder(t->right);
print(t->element);
}
}
```

Binary Trees

A binary tree is a tree in which no node can have more than two children.

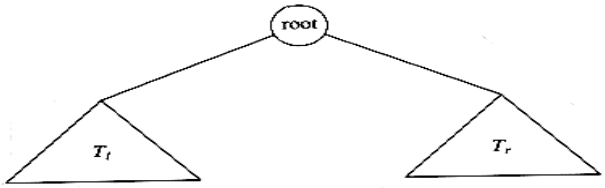
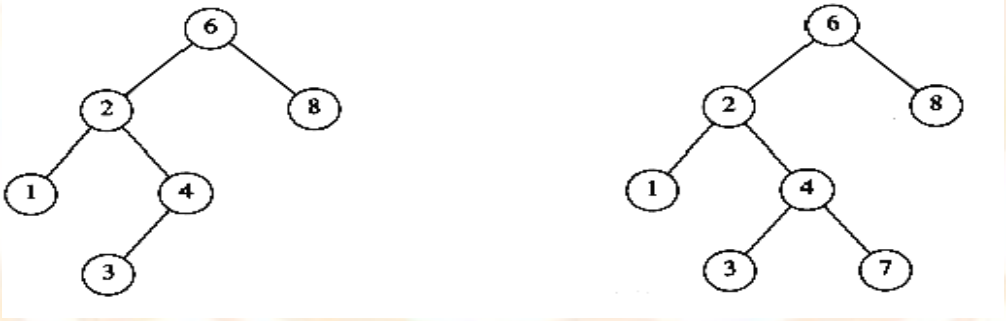


Figure shows that a binary tree consists of a root and two subtrees, Tl and Tr, both of which could possibly be empty.

The Search Tree ADT-Binary Search Tree

The property that makes a binary tree into a binary search tree is that for every node, X, in the tree, the values of all the keys in the left subtree are smaller than the key value in X, and the values of all the keys in the right subtree are larger than the key value in X.

Notice that this implies that all the elements in the tree can be ordered in some consistent manner.



In the above figure, the tree on the left is a binary search tree, but the tree on the right is not. The tree on the right has a node with key 7 in the left subtree of a node with key 6.

To perform a findmin, start at the root and **go left as long as there is a left child**. The stopping point is the smallest element. (deepest left child node)

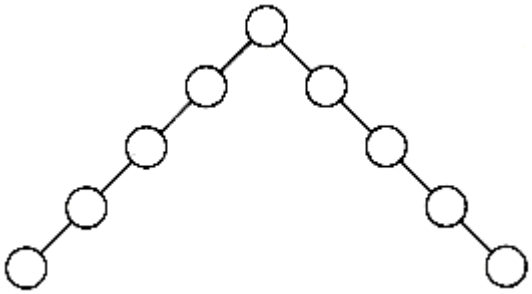
The findmax routine is the same, except that **branching is to the right child. (deepest right child node)**

AVL Trees (Height Balanced BST)

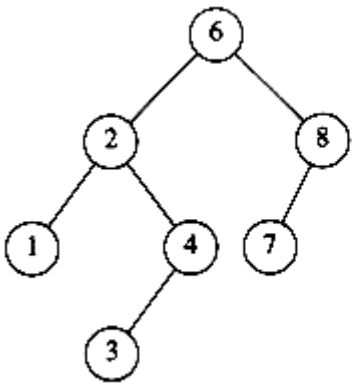
The balance condition and allow the tree to be arbitrarily deep, but after every operation, a restructuring rule is applied that tends to make future operations efficient. These types of data structures are generally classified as **self-adjusting.**

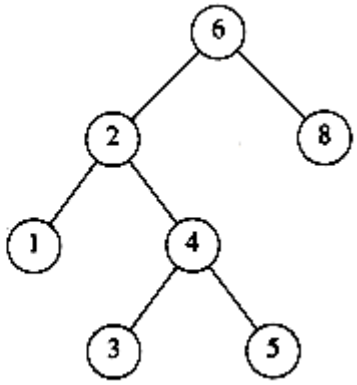
An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be -1.)

An AVL (**Adelson-Velskii and Landis**) tree is a binary search tree with a balance condition. The simplest idea is to require that the left and right subtrees have the same height. The balance condition must be easy to maintain, and it ensures that the depth of the tree is O(log n).



The above figure shows, a bad binary tree. Requiring balance at the root is not enough.





In Figure, the tree on the left is an AVL tree, but the tree on the right is not.

Thus, all the tree operations can be performed in O(log n) time, except possibly insertion.

When we do an insertion, we need to update all the balancing information for the nodes on the path back to the root, but the reason that insertion is difficult is that inserting a node could violate the AVL tree property.

Inserting a node into the AVL tree would destroy the balance condition.

Let us call the unbalanced node α. Violation due to insertion might occur in four cases:

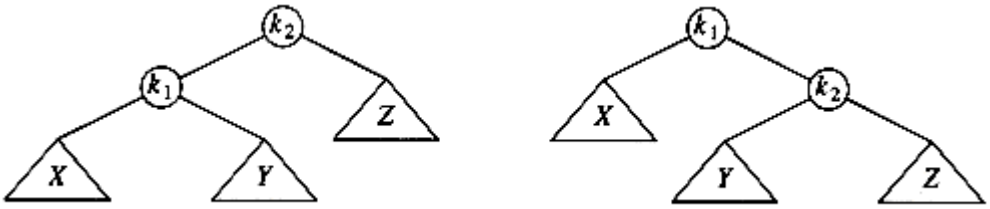
1. **An insertion into the left subtree of the left child of α**
2. **An insertion into the right subtree of the left child of α**
3. **An insertion into the left subtree of the right child of α**
4. **An insertion into the right subtree of the right child of α**

Violation of AVL property due to insertion can be avoided by doing some modification on the node α. This modification process is called as **Rotation**.

Types of rotation

1. Single Rotation
2. Double Rotation

Single Rotation (case 1) – Single rotate with Left



The two trees in the above Figure contain the same elements and are both binary search trees.

First of all, in both trees k1 < k2. Second, all elements in the subtree X are smaller than k1 in both trees. Third, all elements in subtree Z are larger than k2. Finally, all elements in subtree Y are in between k1 and k2. The conversion of one of the above trees to the other is known as **a rotation.**

Node declaration for AVL trees:

```
typedef struct avlnode *position;
typedef struct avlnode *avltree;
struct avlnode
{
Comparable element;
avlnode *left;
avlnode *right;
int height;
```

```
avlnode( comparable & theelement, avlnode *lt, avlnode *rt, int h=0; )
 : element(theelement), left(lt),right(rt),height(h)
};
```

Routine for finding height of an AVL node

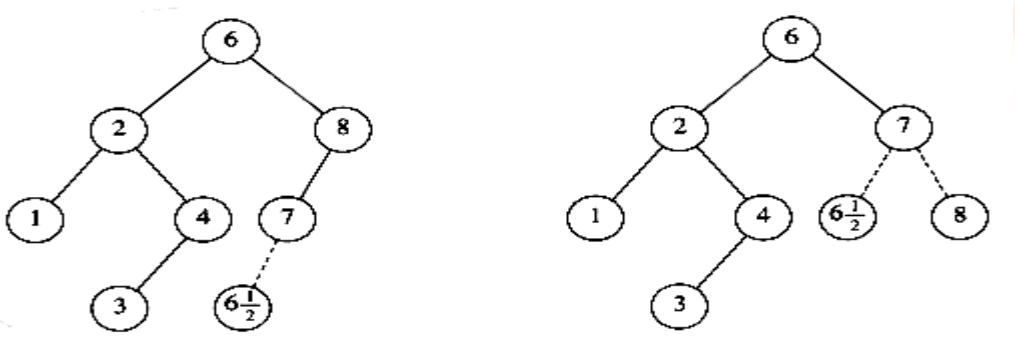
```
int height (avlnode *t)
{
Return t== NULL ? -1:t->height;
}
```

Routine :

```
Static position Singlerotatewithleft( Position K2)
        {
        Position k1;
        K1=k2->left;
        K2->left=k1->right;
        K1->right=k2;
        K2->height=max(height(k2->left),height(k2->right));
        K1->height=max(height(k1->left),k2->height);
        Return k1;
        }
```

In an AVL tree, if an insertion causes some node in an AVL tree to lose the balance property: Do a rotation at that node.

The basic algorithm is to start at the node inserted and travel up the tree, updating the balance information at every node on the path.



In the above figure, after the insertion of the in the original AVL tree on the left, node 8 becomes

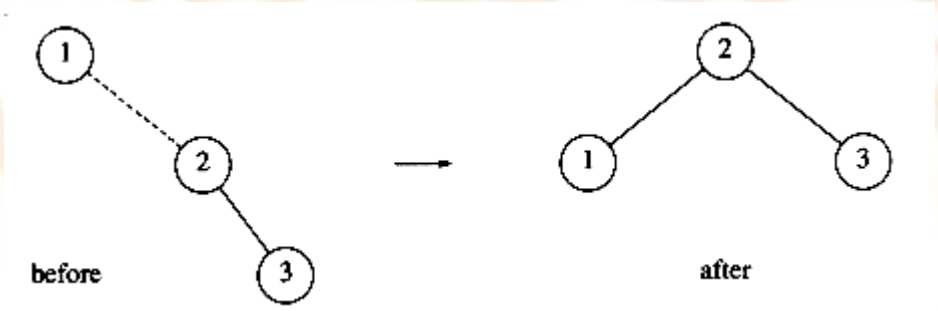
unbalanced. Thus, we do a single rotation between 7 and 8, obtaining the tree on the right.

Single Rotation (case 4) – Single rotate with Right

Routine :

```
Static position Singlerotatewithright( Position K1)
        {
        Position k2;
        K2=k1->right;
        K1->right=k2->left;
        K2->left=k1;
        K1->height=max(height(k1->left),height(k1->right));
        K2->height=max(height(k2->left),k1->height);
        Return k2;
        }
```

Suppose we start with an initially empty AVL tree and insert the keys 1 through 7 in sequential order. The first problem occurs when it is time to insert key 3, because the AVL property is violated at the root. We perform a single rotation between the root and its right child to fix the problem. The tree is shown in the following figure, before and after the rotation.

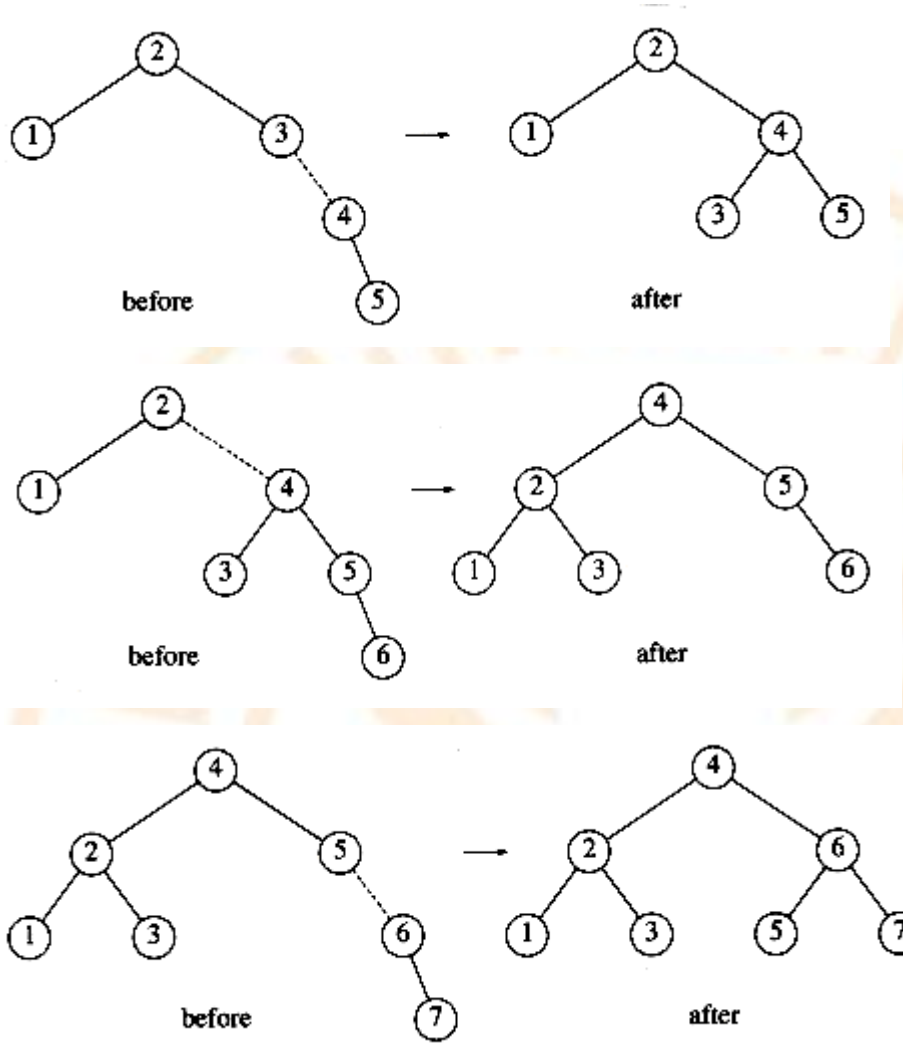


A dashed line indicates the two nodes that are the subject of the rotation. Next, we insert the key 4, which causes no problems, but the insertion of 5 creates a violation at node 3, which is fixed by a single rotation.

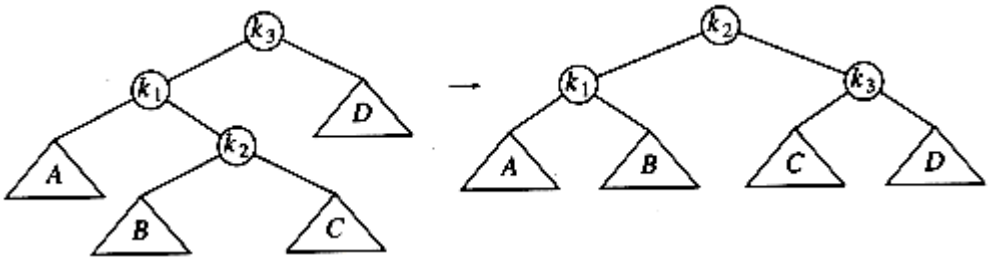
Next, we insert 6. This causes a balance problem for the root, since its left subtree is of height 0, and its right subtree would be height 2. Therefore, we perform a single rotation at the root between 2 and 4.

The rotation is performed by making 2 a child of 4 and making 4's original left subtree the new right subtree of 2. Every key in this subtree must lie between 2 and 4, so this transformation makes sense.

The next key we insert is 7, which causes another rotation.



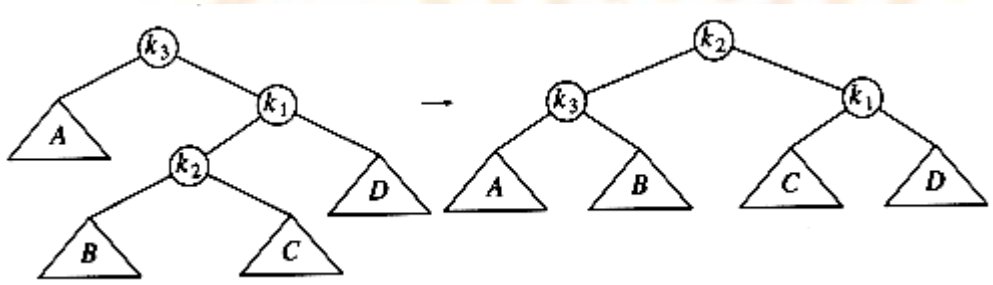
Case ii) (Left-right) double rotation



Routine for double Rotation with left (Case 2)

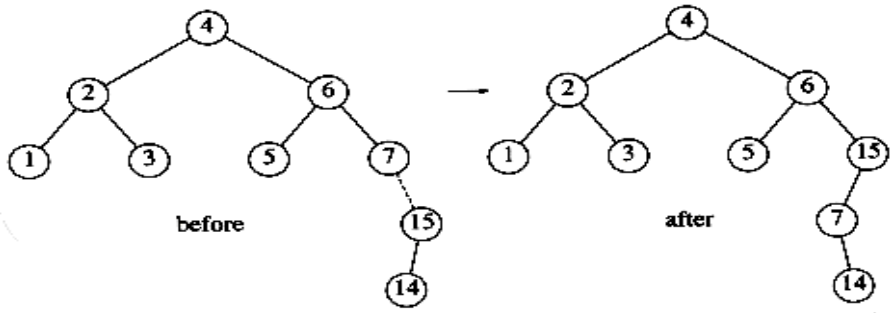
```
Static position doublerotatewithleft(position k3)
        {
        K3->left=singlerotatewithright(k3->left);
        Return singlerotatewithleft(k3);
        }
```

Double Rotation (Right-left) double rotation



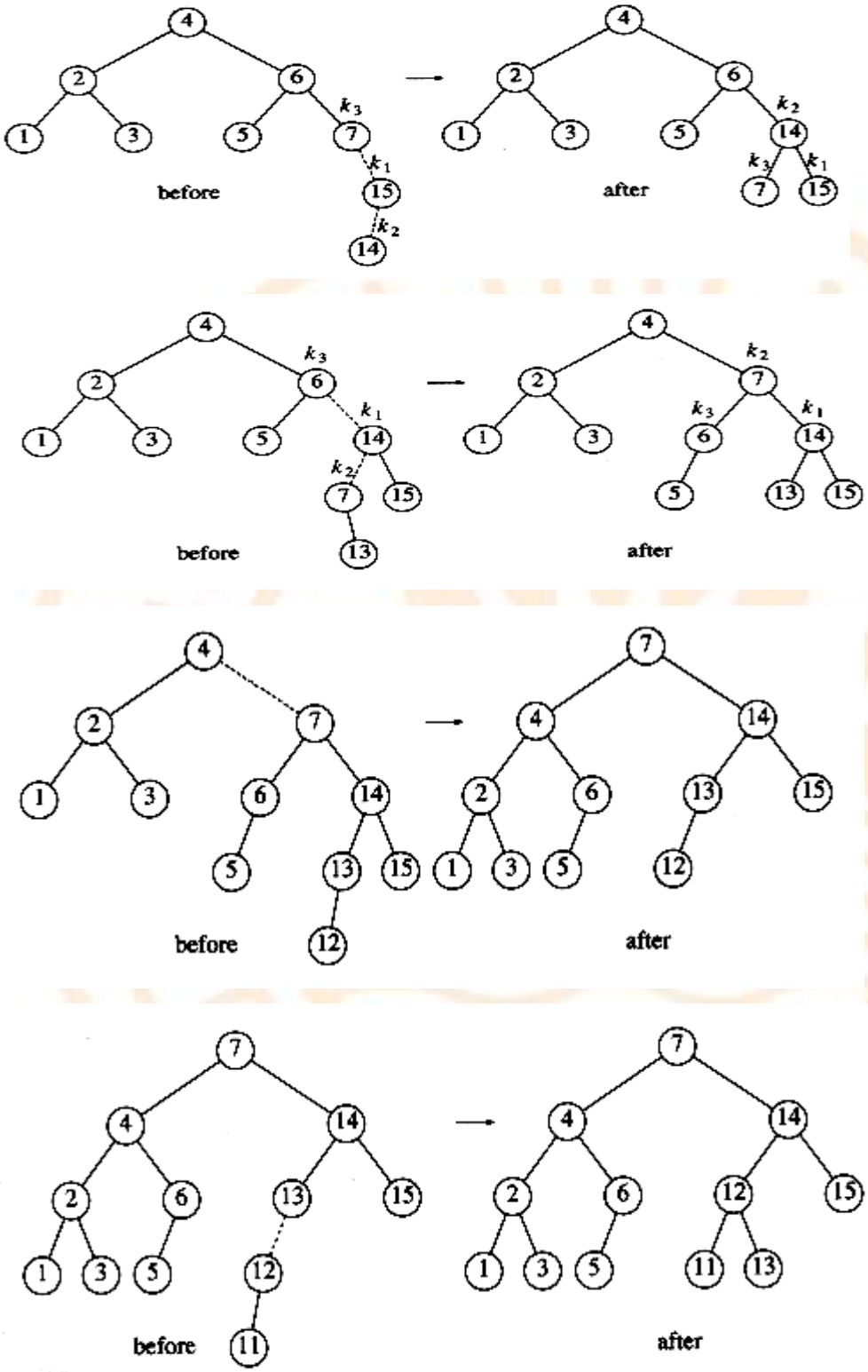
Routine for double Rotation with right (Case 3)

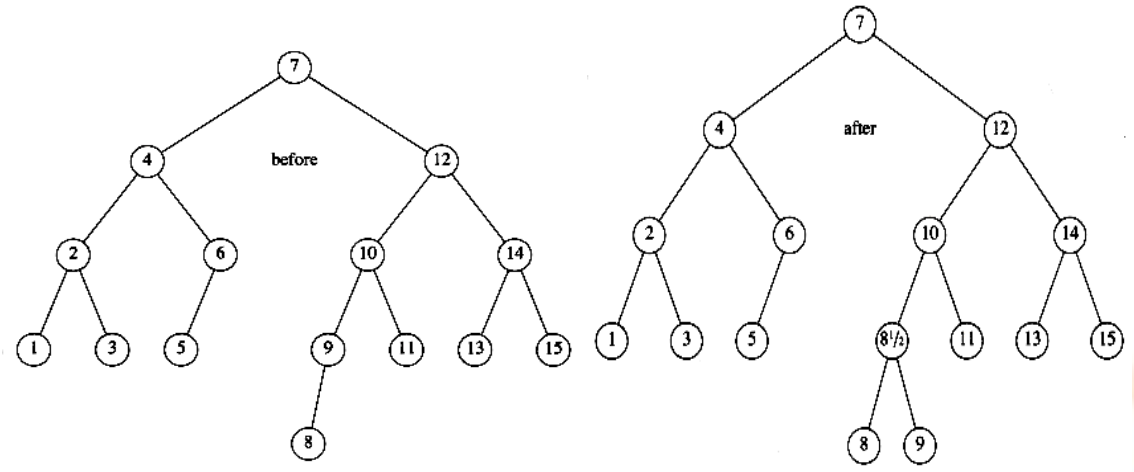
```
Static position doublerotatewithlright(position k1)
        {
        K1->right=singlerotatewithleft(k1->right);
        Return singlerotatewithright(k1);
        }
```



In the above diagram, suppose we insert keys 8 through 15 in reverse order. Inserting 15 is easy, since it does not destroy the balance property, but inserting 14 causes a height imbalance at node 7.

As the diagram shows, the single rotation has not fixed the height imbalance. The problem is that the height imbalance was caused by a node inserted into the tree containing the middle elements (tree Y in Fig. (Right-left) double rotation) at the same time as the other trees had identical heights. This process is called as **double rotation**, which is similar to a single rotation but involves four subtrees instead of three.


before
after
before
after
before
after
before
after



In our example, the double rotation is a right-left double rotation and involves 7, 15, and 14. Here, k3 is the node with key 7, k1 is the node with key 15, and k2 is the node with key 14.

Next we insert 13, which require a double rotation. Here the double rotation is again a right-left double rotation that will involve 6, 14, and 7 and will restore the tree. In this case, k3 is the node with key 6, k1 is the node with key 14, and k2 is the node with key 7. Subtree A is the tree rooted at the node with key 5, subtree B is the empty subtree that was originally the left child of the node with key 7, subtree C is the tree rooted at the node with key 13, and finally, subtree D is the tree rooted at the node with key 15.

If 12 is now inserted, there is an imbalance at the root. Since 12 is not between 4 and 7, we know that the single rotation will work. Insertion of 11 will require a single rotation:

To insert 10, a single rotation needs to be performed, and the same is true for the subsequent insertion of 9. We insert 8 without a rotation, creating the almost perfectly balanced tree.

B-Trees

AVL tree and Splay tree are binary; there is a popular search tree that is not binary. This tree is known as a B-tree.

A B-tree of order m is a tree with the following structural properties:

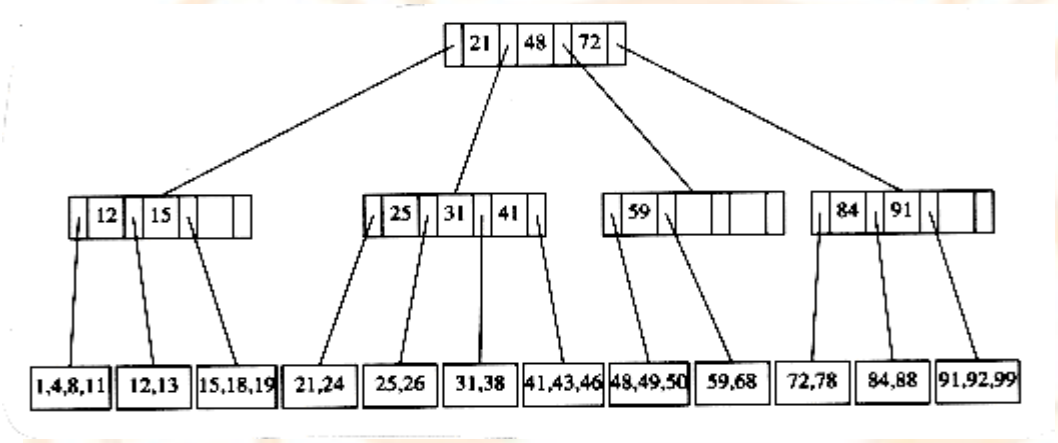
a. The data items are stored at leaves.
b. The non leaf node store up to M-1 keys to guide the searching; key I represents the smallest key in subtree i+!.
c. The root is either a leaf or has between 2 and m children.
d. All nonleaf nodes (except the root) have between m/2 and m children.
e. All leaves are at the same depth and have between L/2 and L data items,

All data is stored at the leaves. Contained in each interior node are pointers p1, p2, . . . , pm to the children, and values k1, k2, . . . , km - 1, representing the smallest key found in the subtrees p2, p3, . . . , pm respectively. Some of these pointers might be NULL, and the corresponding ki would then be undefined.

For every node, all the keys in subtree p1 are smaller than the keys in subtree p2, and so on. The leaves contain all the actual data, which is either the keys themselves or pointers to records containing the keys.

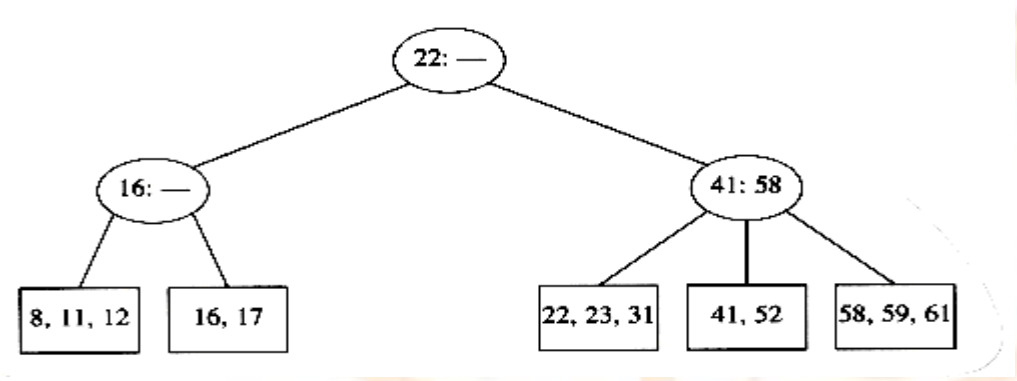
The number of keys in a leaf is also between *m*/2 and m.

An example of a B-tree of order 3 and L = 3



A B-tree of order 4 is more popularly known as a 2-3-4 tree, and a B-tree of order 3 is known as a 2-3 tree

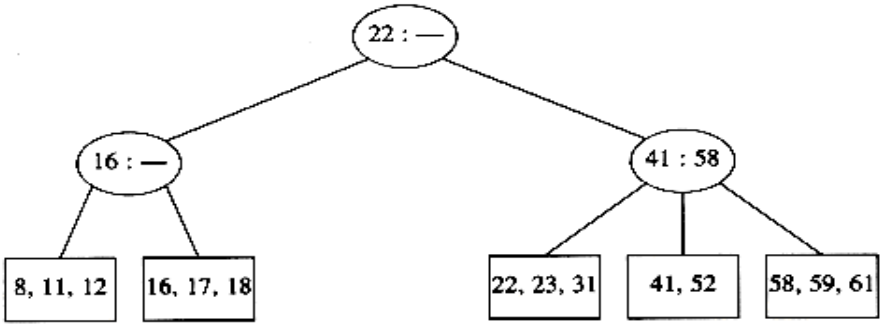
Our starting point is the 2-3 tree that follows.



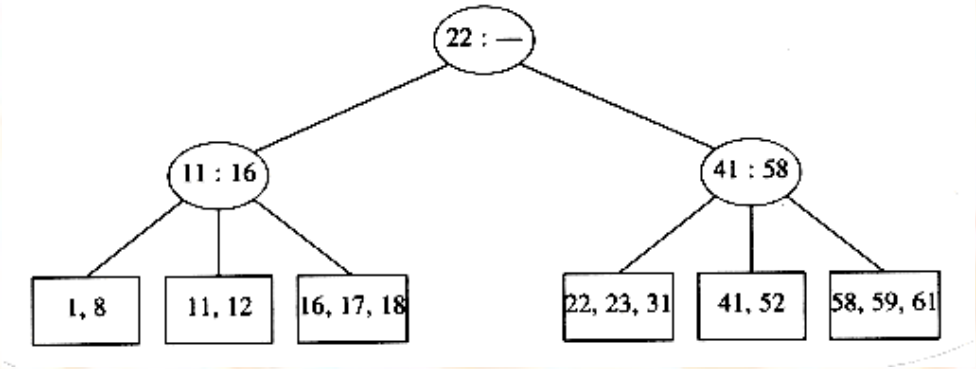
We have drawn interior nodes (nonleaves) in ellipses, which contain the two pieces of data for each node. A dash line as a second piece of information in an interior node indicates that the node has only two children. Leaves are drawn in boxes, which contain the keys. The keys in the leaves are ordered.

To perform a find, we start at the root and branch in one of (at most) three directions, depending on the relation of the key we are looking for to the two values stored at the node.

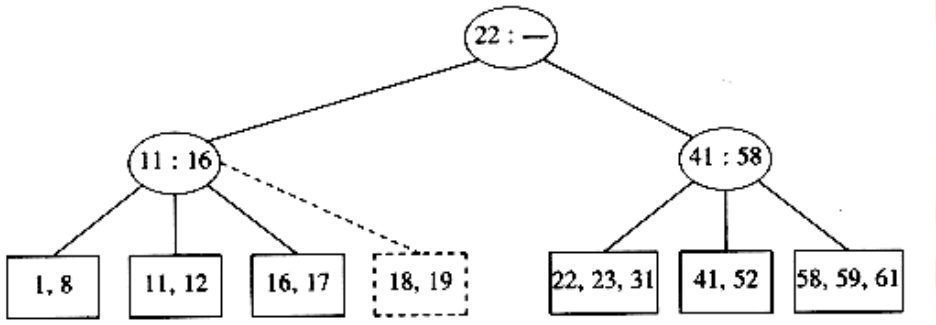
When we get to a leaf node, we have found the correct place to put x. Thus, to insert a node with key 18, we can just add it to a leaf without causing any violations of the 2-3 tree properties. The result is shown in the following figure.



If we now try to insert 1 into the tree, we find that the node where it belongs is already full. Placing our new key into this node would give it a fourth element which is not allowed. This can be solved by making two nodes of two keys each and adjusting the information in the parent.



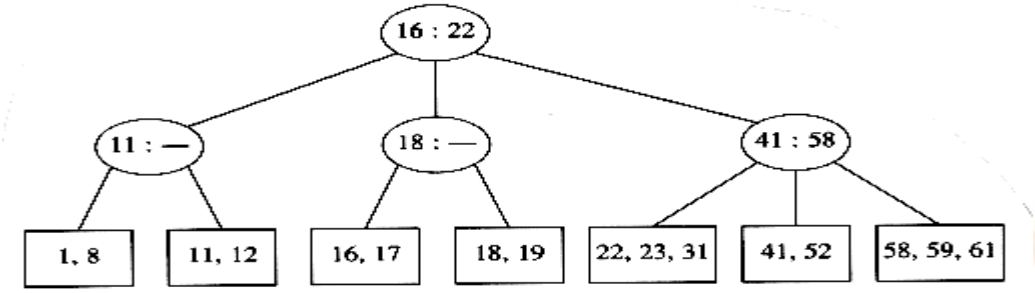
To insert 19 into the current tree, two nodes of two keys each, we obtain the following tree.



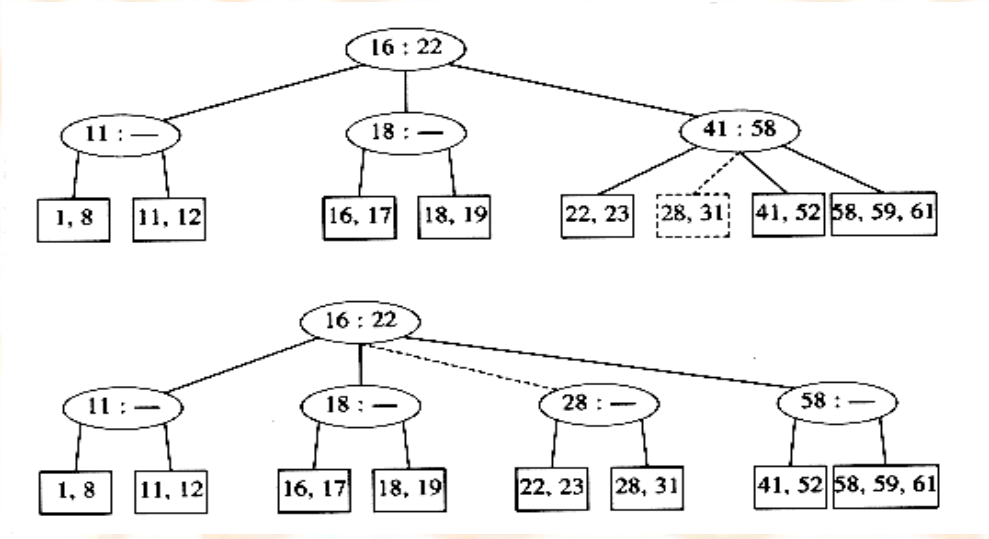
This tree has an internal node with four children, but we only allow three per node.

Again split this node into two nodes with two children. Now this node might be one of three children itself, and thus splitting it would create a problem for its parent but we can keep on splitting nodes on the

way up to the root until we either get to the root or find a node with only two children.



If we now insert an element with key 28, we create a leaf with four children, which is split into two leaves of two children.



This creates an internal node with four children, which is then split into two children. Like to insert 70 into the tree above, we could move 58 to the leaf containing 41 and 52, place 70 with 59 and 61, and adjust the entries in the internal nodes.

Deletion in B-Tree (Refer the attached Xerox copy : Magnus publication for example)

- If this key was one of only two keys in a node, then its removal leaves only one key. We can fix this by combining this node with a sibling. If the sibling has three keys, we can steal one and have both nodes with two keys.
- If the sibling has only two keys, we combine the two nodes into a single node with three keys. The parent of this node now loses a child, so we might have to percolate this strategy all the way to the top.
- If the root loses its second child, then the root is also deleted and the tree becomes one level shallower.

We repeat this until we find a parent with less than m children. If we split the root, we create a new root with two children.

The depth of a B-tree is at most log m/2 n.

The worst-case running time for each of the insert and delete operations is thus **O(m logm n) = O((m / log m) log n), but a find takes only O(log n).**

Splay Trees

It is also like binary search tree (BST). In splay tree, the frequently accessing node is placed as the root node. So that it can be accessed easily again and again inorder to reduce the accessing time of the particular node.

BST can be converted into splay tree by splaying procedure. It uses the idea of AVL tree rotations. But uses selectively user rotation.

All BST operations are combined into one operation called **splaying.**

Splaying:

a. If the accessed node is already in the root, do nothing.

b. If the accessed node X has root as its parent, then rotate X with its parent. Hers splaying is done using **Zig-Rotation.**

c. If the accessed node X has both parent and grand parent, there are four possible cases using **Zig-Zig and Zig- Zag Rotation**

Zig-Zig Rotation:

Zig – Zig rotation is used, if the accessed node X is

- ✓ Left – Left child of the grand parent
- ✓ Right - Right child of the grand parent

Zig- Zag Rotation

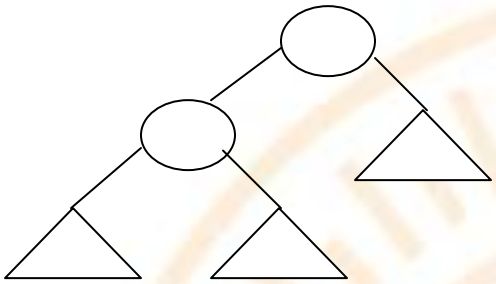
Zig – Zig rotation is used, if the accessed node X is

- ✓ Left – Right Left child of the grand parent
- ✓ Right - Left child of the grand parent

Zig-Rotation

The accessed node X is the child of the parent then rotate node X with the parent either left or right using Zig rotation.

Case i: (Right rotation on P)



Procedures for Zig Rotation

Procedure Zig (P)

```
Begin
        S←Rlink(X);
        X←Llink(P);
        Llink(P)←S;
        Rlink(X)←P;
        Root←X
end
```

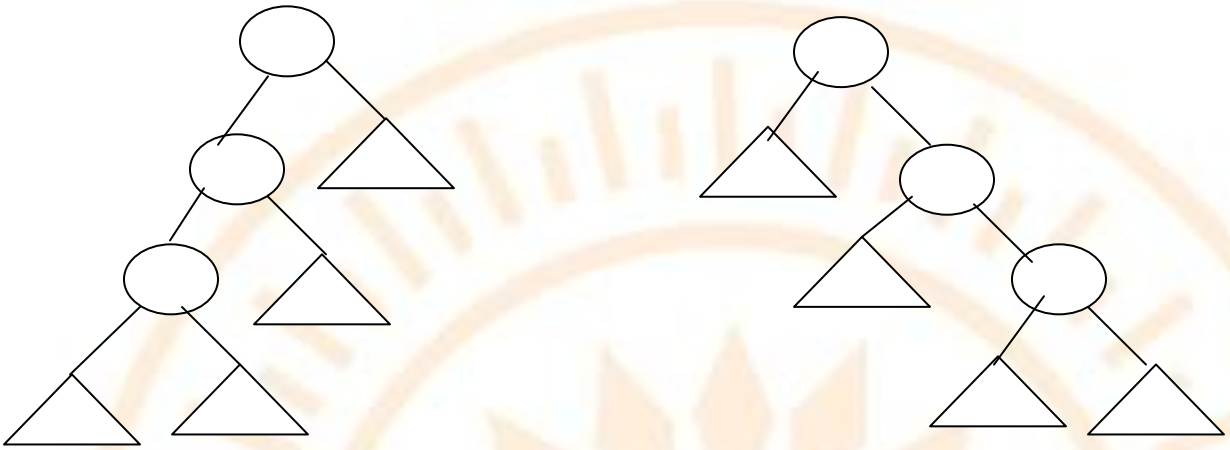
Case ii: (Left rotation on P)

Procedure Zig (P)

```
Begin
        S←Llink(X);
        X←Rlink(P);
        Rlink(P)←S;
        Llink(X)←P;
        Root←X
End
```

Zig-Zig Rotation:

Zig – Zig rotation is used, if the frequently accessed node X is Left – Left child of the grand parent or Right - Right child of the grand parent.



Procedure Zig- Zig (G) Begin

S←Rlink(X);

R←Rlink(P);

Rlink(P) ← G;

Llink(G) ←R;

Llink(P)←S;

Rlink(X)← P;

Root←X

End

Zag – Zag Rotation

Procedure Zag- Zag(G)

Begin

S←Llink(X);

R←Llink(P);

Llink(P) ← G;

Rlink(G) ←R;

Rlink(P)←S;

Llink(X)← P;

Root←X

End

Zig-Zag Rotation:

Zig – Zag rotation is used, if the frequently accessed node X is Left – Right child of the grand parent or Right - Left child of the grand parent.

Procedure Zig- Zag (G)

Begin

S←Llink(X);

R←Rlink(X);

Llink(X) ← P;

Rlink(X) ←G;

Rlink(P)←S;

Llink(G)← R;

Root←X

End

Zag – Zig Rotation

Procedure Zag- Zig(G)

Begin

S←Llink(X);

R←Rlink(X);

Rlink(G) ← S;

Llink(P) ←R;

Llink(X)←G;

Rlink(X)← P;

Root←X

End

Uses:

1. It is used to perform non – uniform sequential operations.
2. It is used to give better performance than other trees.
3. It doesn't require any book keeping information such as balance factor for an AVL tree. So it reduces memory usage.
4. Mainly used in cache memory and garbage collection

Advantages:

Simpler

More space efficient

More flexible

Faster

Disadvantages:

Worst case issues of splay tree algorithm is, all the elements are sequentially accessed in sorted order.

It leads splay tree into unbalanced tree.

Disjoint Set:

Definition:

Set is a collection of a finite data items. A set is an Abstract Data Type(ADT) on which such mathematical functions such as " X is an element of ", "set complement ", " set interaction", etc can be performed.

An ordered set is therefore a collection of items with two defining properties.

1. **Uniqueness**
2. **Ordering**

Relation in a set

A relation R is defined on a set S if for every pair of elements (a, b), a, b S, a R b is either true or false. If a R b is true, then we say that a is related to b.

An **equivalence relation** is a relation R that satisfies three properties:

1. (Reflexive) a R a, for all a€ S.
2. (Symmetric) a R b if and only if b R a.
3. (Transitive) a R b and b R c implies that a R c.

Examples:

Electrical connectivity, where all connections are by metal wires, is an equivalence relation. The relation is clearly reflexive, as any component is connected to itself. If a is electrically connected to b, then b must be electrically connected to a, so the relation is symmetric. Finally, if a is connected to b and b is connected to c, then a is connected to c. Thus electrical connectivity is an equivalence relation.

The ≤ relationship is not an equivalence relationship. Although it is reflexive, since aRa, and transitive, since aR b and bR c implies a Rc, it is not symmetric, since aR b does not imply bR a.

The Dynamic Equivalence Problem

Given an equivalence relation ~, for any a and b, if a ~ b. As an example, suppose the equivalence relation is defined over the five-element set {a1, a2, a3, a4, a5}. Then there are 25 pairs of elements, each of which is either related or not. However, the information a1 ~ a2, a3 ~ a4, a5 ~ a1, a4 ~ a2 implies that all pairs are related.

The **equivalence class** of an element a € S is the subset of S that contains all the elements that are related to a. Notice that the equivalence classes form a partition of S: Every member of S appears in exactly one equivalence class. To decide if a ~ b, we need only to check whether a and b are in the same equivalence class. This provides our strategy to solve the equivalence problem.

The input is initially a collection of n sets, each with one element. This initial representation is that all relations (except reflexive relations) are false. Each set has a different element, so that Si Sj = ; this makes the sets disjoint.

There are two permissible operations to identify the equivalence relation between a and b. they are

Find : which returns the name of the set containing a given element.

Add : add the relation a ~ b, finds on both a and b and checking whether they are in the same equivalence class. If they are not, then we apply union. This operation merges the two equivalence classes containing a and b into a new equivalence class. To create a new set Sk = Si U Sj.

There are two strategies to solve this problem.

- ✓ **Find (a) :** return the name of the set containing the element a;
- ✓ **Union (a,b) :** the element a and b belongs to the same equivalence class.

For that Find(a) = Find(b). If not merge two sets containing a & b to keep both a and b in the same equivalence class.

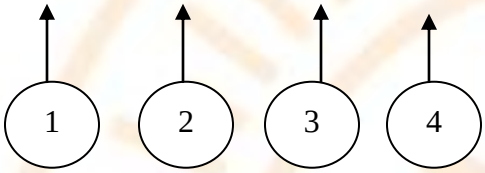
One idea is to keep all the elements that are in the same equivalence class in a linked list. If we also keep

track of the size of each equivalence class, and when performing unions, we change the name of the smaller equivalence class to the larger, then the total time spent for n - 1 merges is O (n log n).

The reason for this is that each element can have its equivalence class changed at most log n times, since every time its class is changed; its new equivalence class is at least twice as large as its old. Using this strategy, any sequence of m finds and up to n - 1 unions takes at most O(m + n log n) time.

Basic Data Structure:

Tree data structure is used to implement the set. The name of a set is given by the node at the root. the array implementation of tree represents that p[i] is the parent of i^{th} element. If i is the root, then p[i] = -1.



-1	-1	-1	-1

Binomial Queues / Binomial Heap

Binomial Queues / Binomial Heap

Binomial queues support all three operations merging, insertion, and delete_min in O(log n) worst-case time per operation, but insertions take constant time on average.

Binomial Queue Structure

Binomial queues differ from all the priority queue implementations that a binomial queue is not a heap-ordered tree but rather a collection of heap-ordered trees, known as a forest. Each of the heap-ordered trees are of a constrained form known as a binomial tree There is at most one binomial tree of every height. A binomial tree of height 0 is a one-node tree; a binomial tree, Bk, of height k is formed by attaching a binomial tree, Bk-1, to the root of another binomial tree, Bk-1.

From the above diagram, a binomial tree, Bk consists of a root with children B0, B1, . . ., Bk-1. Binomial trees of height k have **exactly 2^k** nodes, and the number of nodes at depth d is the binomial coefficient $\binom{k}{d}$. If we impose heap order on the binomial trees and allow at most one binomial tree of any height, we can uniquely represent a priority queue of any size by a collection of binomial trees.

A priority queue of size 13 could be represented by the forest B3, B2, B0. We might write this representation as 1101, which not only represents 13 in binary but also represents the fact that B3, B2 and B0 are present in the representation and B1 is not.

As an example, a priority queue of six elements could be represented as in Figure.

Binomial queue H1 with six elements

Binomial Queue Operations

The minimum element can then be found by scanning the roots of all the trees. Since there are at most log n different trees, the minimum can be found in O(log n) time. Alternatively, we can maintain knowledge of the minimum and perform the operation in O(1) time, if we remember to update the minimum when it changes during other operations.

Merging two binomial queues is a conceptually easy operation, which we will describe by example.

Consider the two binomial queues, H1 and H2 with six and seven elements, respectively, shown in Figure.

Let H3 be the new binomial queue. Since H1 has no binomial tree of height 0 and H2 does, we can just use the binomial tree of height 0 in H2 as part of H3.

Next, we add binomial trees of height 1. Since both H1 and H2 have binomial trees of height 1, we merge them by making the larger root a subtree of the smaller, creating a binomial tree of height 2.

H3 will not have a binomial tree of height 1. There are now three binomial trees of height 2, namely, the original trees of H1 and H2 plus the tree formed by the previous step.

We keep one binomial tree of height 2 in H3 and merge the other two, creating a binomial tree of height 3. Since H1 and H2 have no trees of height 3, this tree becomes part of H3 and we are finished. The merge is performed by essentially adding the two queues together.

Since merging two binomial trees takes constant time with almost any reasonable implementation, and there are O(log n) binomial trees, the merge takes O(log n) time in the worst case.

As an example, we show in Figures, the binomial queues that are formed by inserting 1 through 7 in order.

A delete_min can be performed by first finding the binomial tree with the smallest root. Let this tree be Bk, and let the original priority queue be H. We remove the binomial tree Bk from the forest of trees in H, forming the new binomial queue H'. We also remove the
root of Bk, creating binomial trees B0, B1, . . . , Bk - l, which collectively form priority queue H". We finish the operation by merging H' and H".

As an example, suppose we perform a delete_min on H3, which is shown again in Figure. The minimum root is 12, so we obtain the two priority queues H' and H". The binomial queue that results from merging H' and H" is the final answer and is shown in Figure .
The delete_min operation breaks the original binomial queue into two. It takes O (log n) time to find the tree containing the minimum element and to create the queues H' and H". Merging these two queues takes O (log n) time, so the entire delete_min operation takes O (log n) time.

Implementation of Binomial Queues

The delete_min operation requires the ability to find all the subtrees of the root quickly, so the standard representation of general trees is required: The children of each node are kept in a linked list, and each node has a pointer to its first child (if any).
Two binomial trees can be merged only if they have the same size. The size of the tree must be stored in the root. Also, when two trees are merged, one of the trees is added as a child to the other. Since this new tree will be the last child (as it will be the largest subtree), we must be able to keep track of the last child of each node efficiently. Only then will we be able to merge two binomial trees..
One way to do this is to use a circular doubly linked list. In this list, the left sibling of the first child will be the last child. The right sibling of the last child could be defined as the first child. This makes it easy to test whether the child we are pointing to is the last.
To summarize, then, each node in a binomial tree will contain the data, first child, left and right sibling, and the number of children (which we will call the rank). Since a binomial queue is just a list of trees, we can use a pointer to the smallest tree as the reference to the data structure.
The binomial queue representation **(Left child- Right sibling representation)**

The type declarations for a node in the binomial tree

```
typedef struct tree_node *tree_ptr;
struct tree_node
{
element_type element;
tree_ptr firstchild;
tree_ptr nextsibling;
unsigned int rank;
};
typedef tree_ptr PRIORITY_QUEUE;
```

A routine to merge two binomial trees of the same size

First, the root of the new tree gains a child, so we must update its rank. We then need to change several pointers in order to splice one tree into the list of children of the root of the other tree.

```
/* Merge two equal-sized binomial trees */
BinomialNode * combinetrees(BinomialNode * T1, BinomialNode *T2)
        {
                if( T2->element <T1->element )
                return combinetrees ( T2, T1 );
                T2->nextsibling = T1->leftchild;
                T1->leftchild = T2;
                return T1;      }
```

Routine to merge two priority queues

```
PRIORITY_QUEUE merge( PRIORITY_QUEUE H1, PRIORITY_QUEUE H2 )
{
PRIORITY_QUEUE H3;      tree_ptr T1, T2, T3;
if( H1 == NULL )      return H2;
if( H2 == NULL )      return H1;
if( H1->rank < H2->rank )
{
        T1 = extract( H1 ); /* extract is a macro */
```

```
H3 = merge( H1, H2 );
T1->l_sib = H3->l_sib;
H3->l_sib->r_sib = NULL;
T1->r_sib = H3; H3->l_sib = T1;
return T1;
{
        if( H2->rank < H1->rank )
        return merge( H2, H1 ); /* Otherwise, first two trees have same rank */
        T1 = extract( H1 ); T2 = extract( H2 );
        H3 = merge( H1, H2 );
        T3 = merge_tree( T1, T2 );
        return merge( T3, H3 );
         }
```

Fibonacci heap

The Fibonacci heap is a data structure that supports all the basic heap operations in O(1) amortized time, with the exception of delete_min and delete, which take O (log n) amortized time. It immediately follows that the heap operations in Dijkstra's algorithm will require a total of O(|E| + |V| log |V|) time.

Fibonacci heaps generalize binomial queues by adding two new concepts:

- A **decrease_key:** The method we have seen before is to percolate the element up toward the root. It does not seem reasonable to expect an O(1) amortized bound for this strategy, so a new method is needed.
- **Lazy merging**: Two heaps are merged only when it is required to do so. This is similar to lazy deletion. For lazy merging, merges are cheap, but because lazy merging does not actually combine trees, the delete_min operation could encounter lots of trees, making that operation expensive.

Cutting Nodes in Leftist Heaps

In binary heaps, the decrease_key operation is implemented by lowering the value at a node and then percolating it up toward the root until heap order is established. In the worst case, this can take O(log n) time, which is the length of the longest path toward the root in a balanced tree.

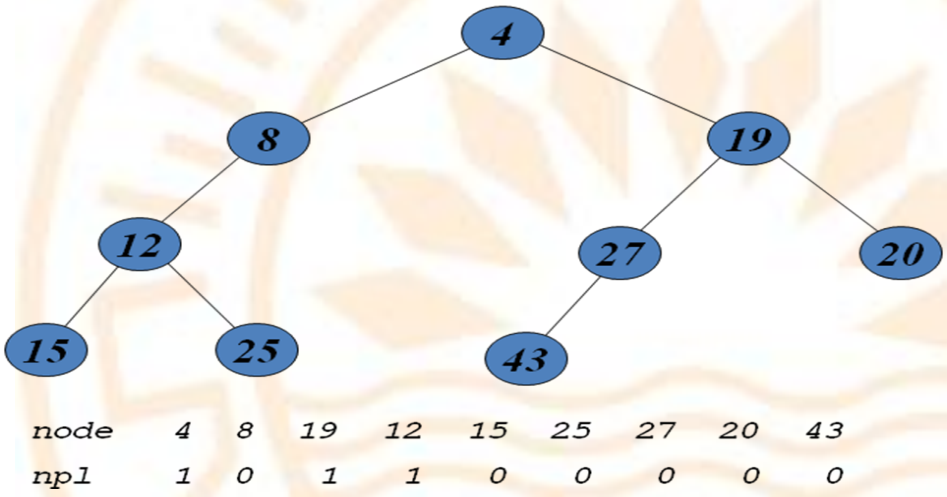
As an example, if this strategy is applied to leftist heaps, then the decrease_key operation could take (n) time, as the example in Figure.

A Leftist (min)Heap is a binary tree that satisfies the follow-ing conditions. If X is a node and L and R are its left and right children, then:

1. X.value ≤ L.value
2. X.value ≤ R.value
3. null path length of L ≥ null path length of R

where the null path length of a node is the shortest distance between from that node to a descendant with 0 or 1 child. If a node is null, its null path length is −1.

Example for Leftist Heap



npl – Null Path length

We see that for leftist heaps, another strategy is needed for the decrease_key operation. An example of leftist heap in below figure

Decrease Key

Suppose we want to decrease the key with value 9 down to 0. If we make the change, we find that we have created a violation of heap order. We do not want to percolate the 0 to the root. The solution is to cut the heap along the dashed line, thus creating two trees, and then merge the two trees back into one.

- Only nodes on the path from p to the root of T2 can be in violation of the leftist heap property; these can be fixed by swapping children.
- Since the maximum right path length has at most log(n + 1) nodes, we only need to check the first log(n + 1) nodes on the path from p to the root of T2.

Lazy Merging

The second idea that is used by Fibonacci heaps is lazy merging. We will apply this idea to binomial queues and show that the amortized time to perform a merge operation (as well as insertion, which is a special case) is O(1). The amortized time for delete_min will still be O(log n).

The idea is as follows:

To merge two binomial queues, merely concatenate the two lists of binomial trees, creating a new binomial queue. This new queue may have several trees of the same size, so it violates the binomial queue property. We will call this a lazy binomial queue in order to maintain consistency. This is a fast operation, which always takes constant (worst-case) time. As before, an

insertion is done by creating a one-node binomial queue and merging. The difference is that the merge is lazy. The delete_min operation is much more painful, because it is where we finally convert the lazy binomial queue back into a standard binomial queue, but, as we will show, it is still O (log n) amortized time-but not O(log n) worst-case time, as before.

To perform a delete_min, we find (and eventually return) the minimum element. As before, we delete it from the queue, making each of its children new trees. We then merge all the trees into a binomial queue by merging two equal-sized trees until it is no longer possible.

UNIT V

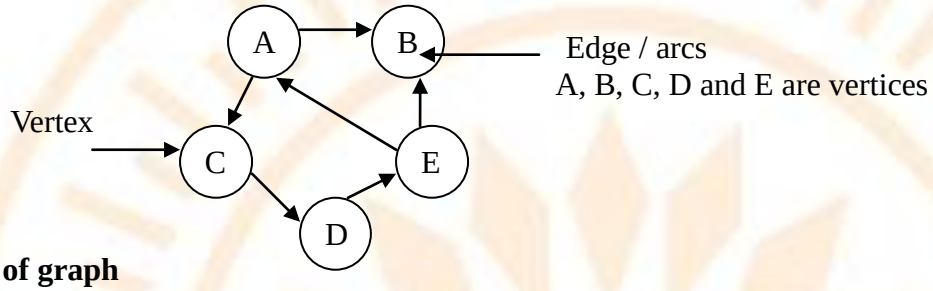
GRAPHS

Representation of Graphs – Breadth-first search – Depth-first search – Topological sort – Minimum Spanning Trees – Kruskal and Prim algorithm – Shortest path algorithm – Dijkstra's algorithm – Bellman-Ford algorithm – Floyd - Warshall algorithm.

Definitions

Graph

A graph G = (V, E) consists of a set of vertices, V, and a set of edges, E. Each edge is a pair (v,w), where v,w € V. Edges are sometimes referred to as **arcs**.

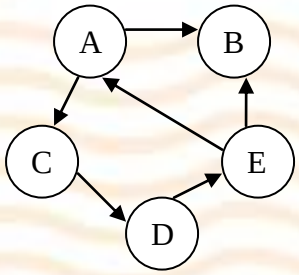


Types of graph

1. Directed Graph

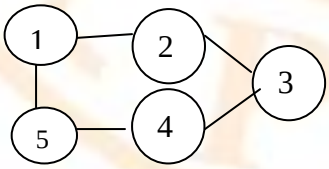
If the pair is ordered, then the graph is directed. In a graph, if all the edges are directionally oriented, then the graph is called as **directed Graph.** Directed graphs are sometimes referred to as **digraphs**.

Vertex w is adjacent to v if and only if (v, w) has an edge E.



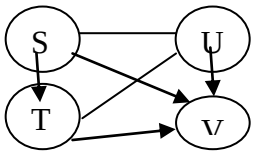
2. Undirected Graph

In a graph, if all the edges are not directionally oriented, then the graph is called as **undirected Graph.** In an undirected graph with edge (v,w), and hence (w,v), w is adjacent to v and v is adjacent to w.



3. Mixed Graph

In a graph if the edges are either directionally or not directionally oriented, then it is called as mixed graph.



Path

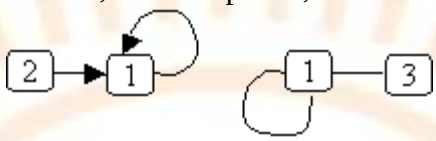
A path in a graph is a sequence of verices w1, w2, w3, . . . , wn such that (wi, wi+i) € E for 1< i < n.

Path length

The length of a path is the number of edges on the path, which is equal to n – 1 where n is the no of vertices.

Loop

A path from a vertex to itself; if this path contains no edges, then the path length is 0. If the graph contains an edge (v,v) from a vertex to itself, then the path v, v is sometimes referred to as a **loop**.



Simple Path

A simple path is a path such that all vertices are distinct, except that the first and last could be the same.

Cycle

In a graph, if the path starts and ends to the same vertex then it is known as **Cycle.**

Cyclic Graph

A directed graph is said to be cyclic graph, if it has cyclic path.

Acyclic Graph

A directed graph is acyclic if it has no cycles. A directed acyclic graph is also referred as **DAG.**

Connected Graph

An undirected graph is connected if there is a path from every vertex to every other vertex.

Strongly connected Graph

A directed graph is called strongly connected if there is a path from every vertex to every other vertex.

Weakly connected Graph

If a directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be weakly connected.

Complete graph

A complete graph is a graph in which there is an edge between every pair of vertices.

Weighted Graph

In a directed graph, if some positive non zero integer values are assigned to each and every edges, then it is known as **weighted graph**. Also called as **Network**

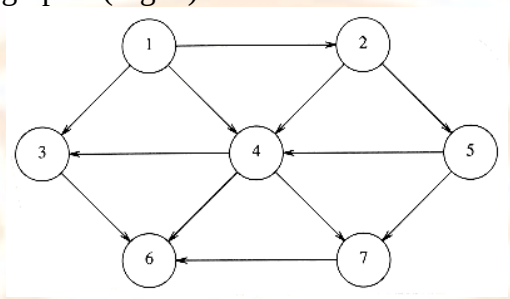
An example of a real-life situation that can be modeled by a graph is the airport system. Each airport is a vertex, and two vertices are connected by an edge if there is a nonstop flight from the airports that are represented by the vertices. The edge could have a weight, representing the time, distance, or cost of the flight.

Representation of Graphs

1. Adjacency matrix / Incidence Matrix
2. Adjacency Linked List/ Incidence Linked List

Adjacency matrix

We will consider directed graphs. (Fig. 1)



Now we can number the vertices, starting at 1. The graph shown in above figure represents 7 vertices and 12 edges.

One simple way to represent a graph is to use a two-dimensional array. This is known as an adjacency matrix representation.

For each edge (u, v), we set a[u][v]= 1; otherwise the entry in the array is 0. If the edge has a weight associated with it, then we can set a[u][v] equal to the weight and use either a very large or a very small weight as a sentinel to indicate nonexistent edges.

Advantage is, it is extremely simple, and the space requirement is ($|V|^2$).

For directed graph

A[u][v]= { 1, if there is edge from u to v
0 otherwise }

For undirected graph

A[u][v]= { 1, if there is edge between u and v
0 otherwise }

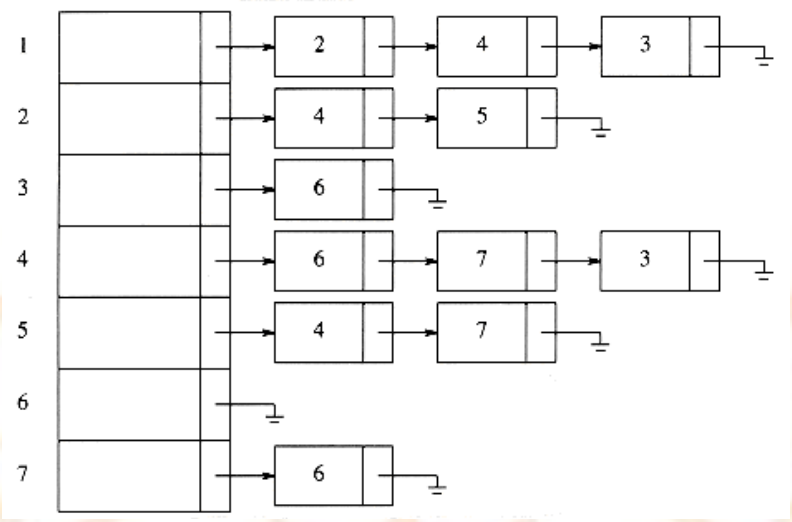
For weighted graph

A[u][v]= { value , if there is edge from u to v
∞, if no edge between u and v }

Adjacency lists

Adjacency lists are the standard way to represent graphs. Undirected graphs can be similarly represented; each edge (u, v) appears in two lists, so the space usage essentially doubles. A common requirement in graph algorithms is to find all vertices adjacent to some given vertex v, and this can be done, in time proportional to the number of such vertices found, by a simple scan down the appropriate adjacency list.

An adjacency list representation of a graph (See above fig 1)

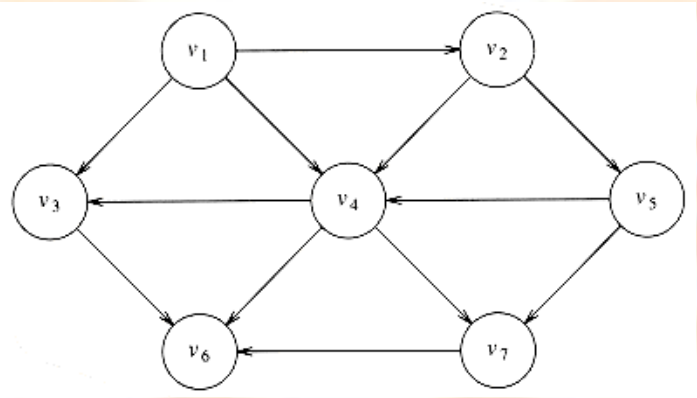


Topological Sort

A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from vi to vj, then vj appears after vi in the ordering.

It is clear that a topological ordering is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v.

Directed acyclic graph



In the above graph v1, v2, v5, v4, v3, v7, v6 and v1, v2, v5, v4, v7, v3, v6 are both topological orderings.

A simple algorithm to find a topological ordering

First, find any vertex with no incoming edges (Source vertex). We can then print this vertex, and remove it, along with its edges, from the graph.

To formalize this, we define the indegree of a vertex v as the number of edges (u,v). We compute the indegrees of all vertices in the graph. Assuming that the indegree array is initialized and that the graph is read into an adjacency list,

	Indegree Before Dequeue #						
Vertex	1	2	3	4	5	6	7
v1	0	0	0	0	0	0	0
v2	1	0	0	0	0	0	0
v3	2	1	1	1	0	0	0
v4	3	2	1	0	0	0	0
v5	1	1	0	0	0	0	0
v6	3	3	3	3	2	1	0
v7	2	2	2	1	0	0	0
Enqueue	v1	v2	v5	v4	v3	v7	v6
Dequeue	v1	v2	v5	v4	v3	v7	v6

Simple Topological Ordering Routine

```
Void   topsort( graph G )
{
unsigned int counter;
vertex v, w;
for( counter = 0; counter < NUM_VERTEX; counter++ )
{
v = find_new_vertex_of_indegree_zero( );
if( v = NOT_A_VERTEX )
{
error("Graph has a cycle");
break;
}
top_num[v] = counter;
for each w adjacent to v
indegree[w]--;
}
}
```

Explanation

The function find_new_vertex_of_indegree_zero scans the indegree array looking for a vertex with indegree 0 that has not already been assigned a topological number. It returns NOT_A_VERTEX if no such vertex exists; this indicates that the graph has a cycle.

Routine to perform Topological Sort

```
Void topsort( graph G )
{
QUEUE Q;
unsigned int counter;
vertex v, w;
Q = create_queue( NUM_VERTEX );
makeempty( Q );
counter = 0;
```

```
        for each vertex v
        if( indegree[v] = 0 )
        enqueue( v, Q );
        while( !isempty( Q ) )
        {
        v = dequeue( Q );
        top_num[v] = ++counter; /* assign next number */
        for each w adjacent to v
        if( --indegree[w] = 0 )
        enqueue( w, Q );
        }
        if( counter != NUMVERTEX )
        error("Graph has a cycle");
        dispose_queue( Q ); /* free the memory */
}
```

Shortest-Path Algorithms

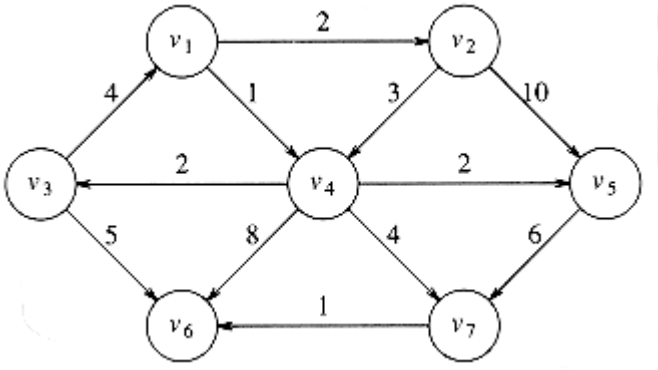
Problem statement:

The input is a weighted graph: associated with each edge (vi, vj) is a cost ci,j to traverse the arc. The cost of a path v1v2 ... vn is $\sum_{i=1}^{N-1} C_{i,i+1}$. This is referred to as the weighted path length. The unweighted path length is merely the number of edges on the path, namely, n - 1.

SINGLE-SOURCE SHORTEST-PATH PROBLEM:

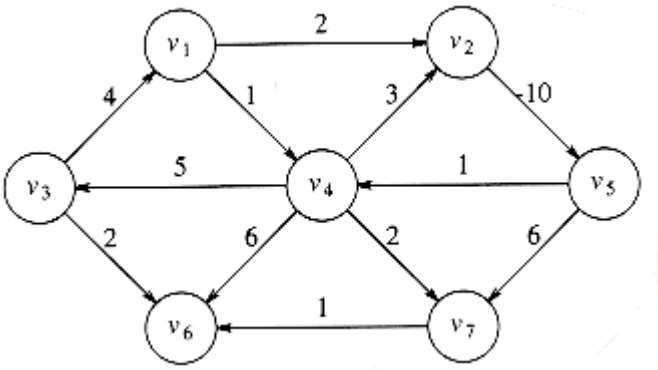
Given as input a weighted graph, G = (V, E), and a distinguished vertex, s , find the shortest weighted path from s to every other vertex in G.

For example, in the graph shown below



The shortest weighted path from v1 to v6 has a cost of 6 and goes from v1 to v4 to v7 to v6. The shortest unweighted path between these vertices is 2. Notice also that in this graph there is no path from v6 tov1. The graph in the preceding example has no edges of negative cost.

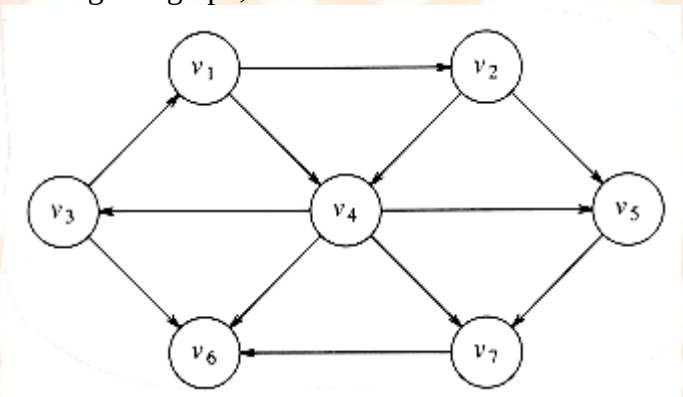
Negative Cost Cycle



The path from v5 to v4 has cost 1, but a shorter path exists by following the loop v5, v4, v2, v5, v4, which has cost -5. This loop is known as a **negative-cost cycle**.

Unweighted Shortest Paths

The below figure shows an unweighted graph, G



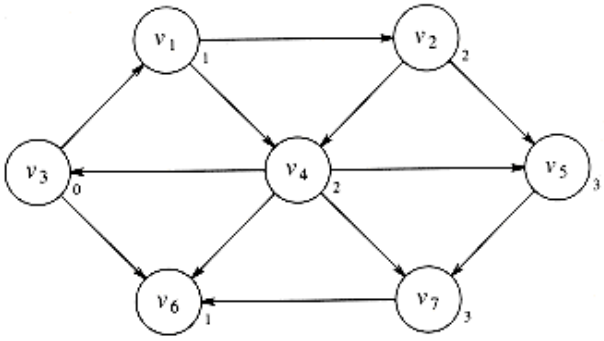
Using some vertex, s, we would like to find the shortest path from s to all other vertices. We are only interested in the number of edges contained on the path, so there are no weights on the edges. We could assign all edges a weight of 1.

Suppose we choose s to be v3. Immediately, we can tell that the shortest path from s to v3 is then a path of length 0.

Now we can start looking for all vertices that are a distance 1 away from s. These can be found by looking at the vertices that are adjacent to s. If we do this, we see that v1 and v6 are one edge from s.

We can now find vertices whose shortest path from s is exactly 2, by finding all the vertices adjacent to v1 and v6 (the vertices at distance 1), whose shortest paths are not already known. This search tells us that the shortest path to v2 and v4 is 2.

Finally we can find, by examining vertices adjacent to the recently evaluated v2 and v4, that v5 and v7 have a shortest path of three edges. All vertices have now been calculated, and so Figure shows the final result of the algorithm.



For each vertex, we will keep track of three pieces of information. First, we will keep its distance from s in the entry dv. Initially all vertices are unreachable except for s, whose path length is 0. The entry in pv is the actual paths. The entry known is set to 1 after a vertex is processed.

Initially, all entries are unknown, including the start vertex.

When a vertex is known, processing for that vertex is essentially complete.

Pseudocode for unweighted shortest-path algorithm

```
void  Graph::unweighted( Vertex s ) /* assume T is initialized */
{
        for each vertex v
        {
         v.dist = INFINITY;
        v.known = false;
        }
        s.dist =0;
        for(int currdist=0; currdist < numvertex; currdist++)

        for each vertex v
        if( ! v. known && v.dist == currdist)
        {
        v.known = true;
         for each vertex v
         if( !v.known && v.dist == currdist)
         {
                v.known = true;
                for each vertex w adjacent to v
                if( w. dist == INFINITY )
                {
                        w.dist = currdist + 1;
                        w. path = v;
                }
           }
        }
```

At any point in time, there are only two types of unknown vertices that have dv . Some have dv = currdist, and the rest have dv = cur_dist + 1.

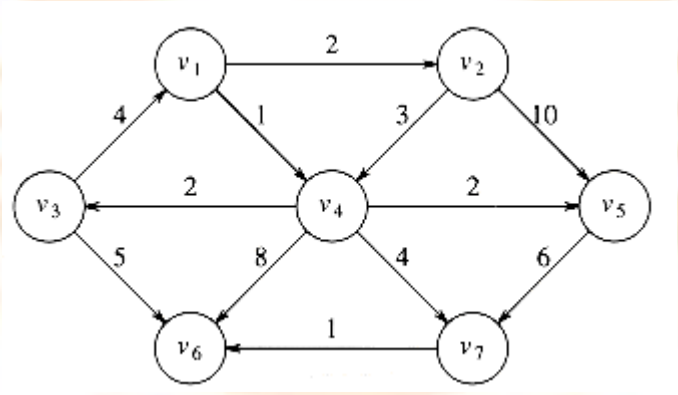
In the unweighted case, we set dw = dv + 1 if dw = INFINITY. Lowered the value of dw if vertex v offered a shorter path.

Pseudocode for unweighted shortest-path algorithm using Queue

```
void Graph:: unweighted( Vertex S )
{
Queue < Vertex> q;
for each vertex v
        v.dist = INFINITY;
S.dist = 0;
q.enqueue (S);
while( !q. is empty( ) )
{
        vertex v = q.dequeue( );
        for each w adjacent to v
        if( w.dist = INFINITY )
        {
wdist = v.dist + 1;
wpath = v;
q.enqueue( w );         }       }       }
```

Dijkstra's Algorithm

If the graph is weighted, the problem becomes harder, but we can still use the ideas from the unweighted case.



Each vertex is marked as either known or unknown. A tentative distance dv is kept for each vertex. The shortest path length from s to v using only known vertices as intermediates.
The general method to solve the single-source shortest-path problem is known as Dijkstra's algorithm.

Dijkstra's algorithm proceeds in stages, just like the unweighted shortest-path algorithm. At each stage, Dijkstra's algorithm selects a vertex v, which has the smallest dv among all the unknown vertices, and declares that the shortest path from s to v is known.

In the above graph, assuming that the start node, s, is v1. The first vertex selected is v1, with path length 0. This vertex is marked known. Now that v1 is known.

Initial configuration table

v	Known	dv	pv
v1	0	0	0
v2	0	∞	0
v3	0	∞	0
v4	0	∞	0
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

The vertices adjacent to v1 are v2 and v4. Both these vertices get their entries adjusted, as indicated below

After v1 is declared known

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	∞	0
v4	0	1	v1
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

Next, v4 is selected and marked known. Vertices v3, v5, v6, and v7 are adjacent.

After v4 is declared known

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	3	v4
v4	1	1	v1
v5	0	3	v4
v6	0	9	v4
v7	0	5	v4

Next, v2 is selected. v4 is adjacent but already known, so no work is performed on it. v5 is adjacent but not adjusted, because the cost of going through v2 is 2 + 10 = 12 and a path of length 3 is already known.

After v2 is declared known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	0	3	v4
v4	1	1	v1
v5	0	3	v4
v6	0	9	v4
v7	0	5	v4

The next vertex selected is v5 at cost 3. v7 is the only adjacent vertex, but it is not adjusted, because 3 + 6 > 5. Then v3 is selected, and the distance for v6 is adjusted down to 3 + 5 = 8.

After v5 and v3 are declared known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	0	8	v3
v7	0	5	v4

Next v7 is selected; v6 gets updated down to 5 + 1 = 6. The resulting table is

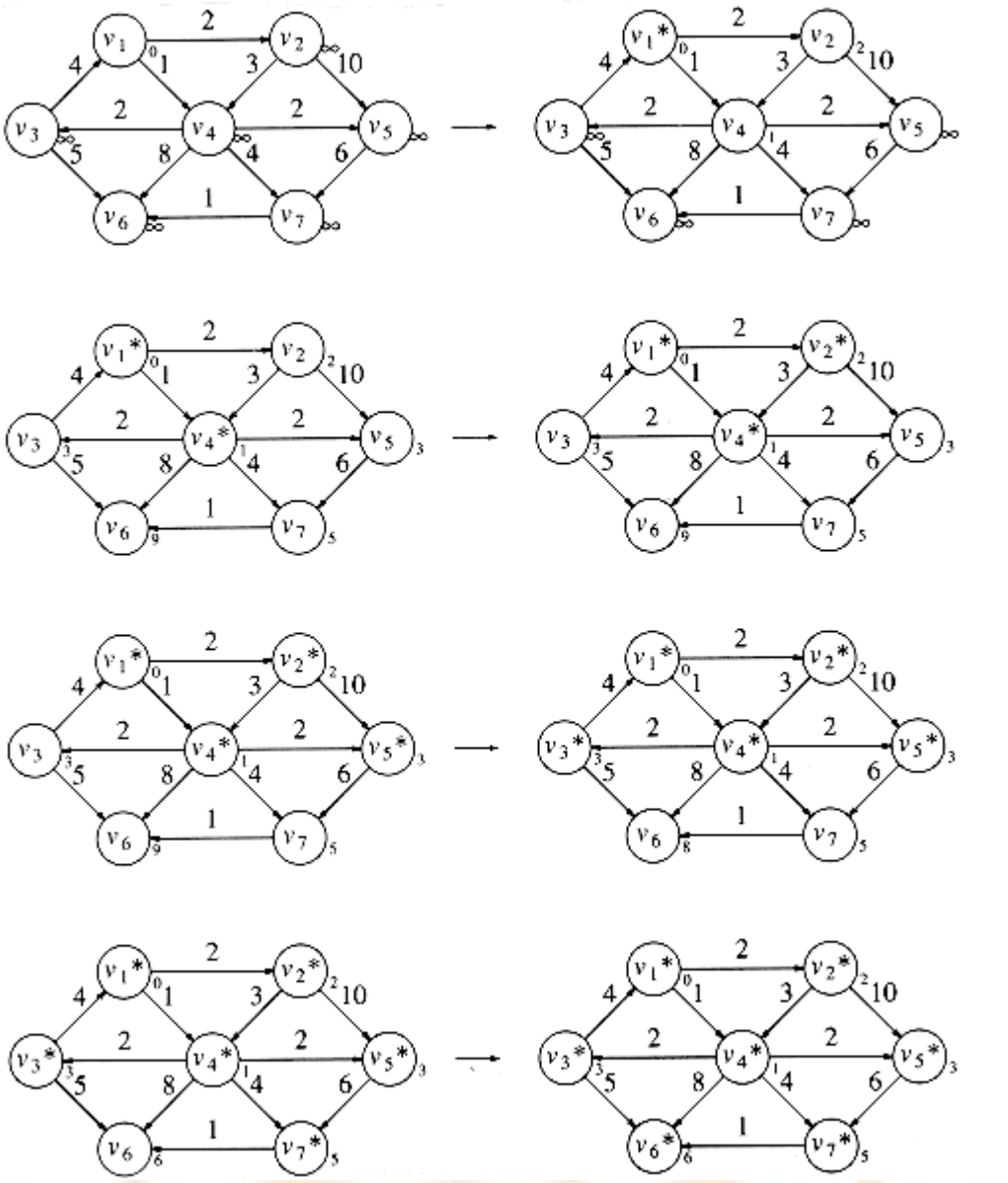
After v7 is declared known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	0	6	v7
v7	1	5	v4

Finally, v6 is selected. The final table is shown

After v6 is declared known and algorithm terminates

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	1	6	v7
v7	1	5	v4



Vertex class for Dijikstra's algorithm

```
struct Vertex
{
   List adj;
  Bool  known;
  disttype dist;
  Vertex path;
 };

#define NOTAVERTEX  0
```

Routine for Dijkstra's algorithm

```
void graph :: dijkstra( Vertex S )
{
        for each vertex v
        {
         v.dist = INFINITY;
        v.known = false;
        }
        s.dist =0;

for( ; ; )
{
        v = smallest unknown distance vertex;
if(v== NotAVertex)
        break;
v. known = TRUE;

for each w adjacent to v
if( !w. known )
        if( v.dist + Cv,w < w.dist )
        {
                decrease( w.dist to v.dist + Cv,w );
                w.path = v;     }         }         }
```

Routine to print the actual shortest path

```
void Graph:: printpath( Vertex v)
{
        if( v.path != NOTAVERTEX )
        {
                printpath(v.path);
                cout<< " to " ;
                }
                cout<< v ;
                }
```

Minimum Spanning Tree

Definition:

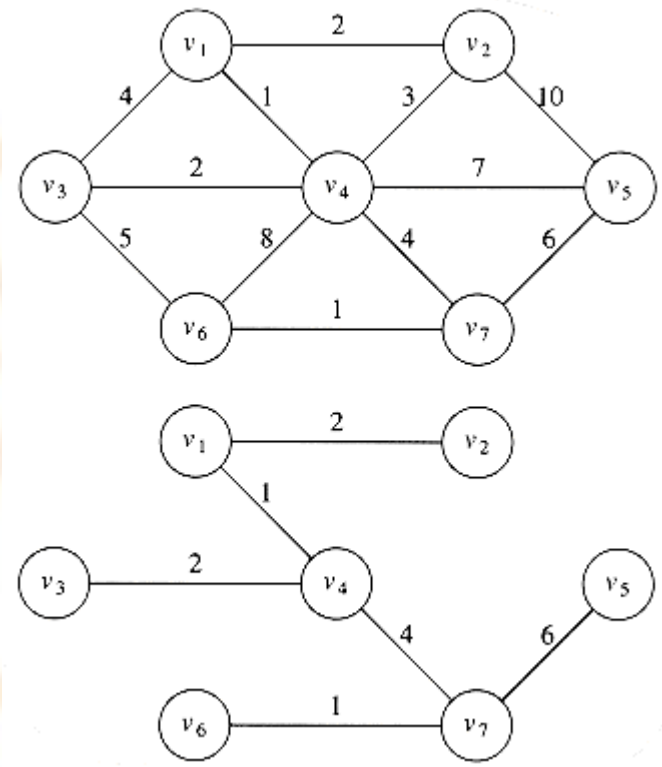
A minimum spanning tree exists if and only if G is connected. A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at lowest total cost.

The number of edges in the minimum spanning tree is |V| - 1. The minimum spanning tree is a tree because it is acyclic, it is spanning because it covers every edge.

Application:

- House wiring with a minimum length of cable, reduces cost of the wiring.

A graph G and its minimum spanning tree



There are two algorithms to find the minimum spanning tree

1. Prim's Algorithm
2. Kruskal's Algorithm

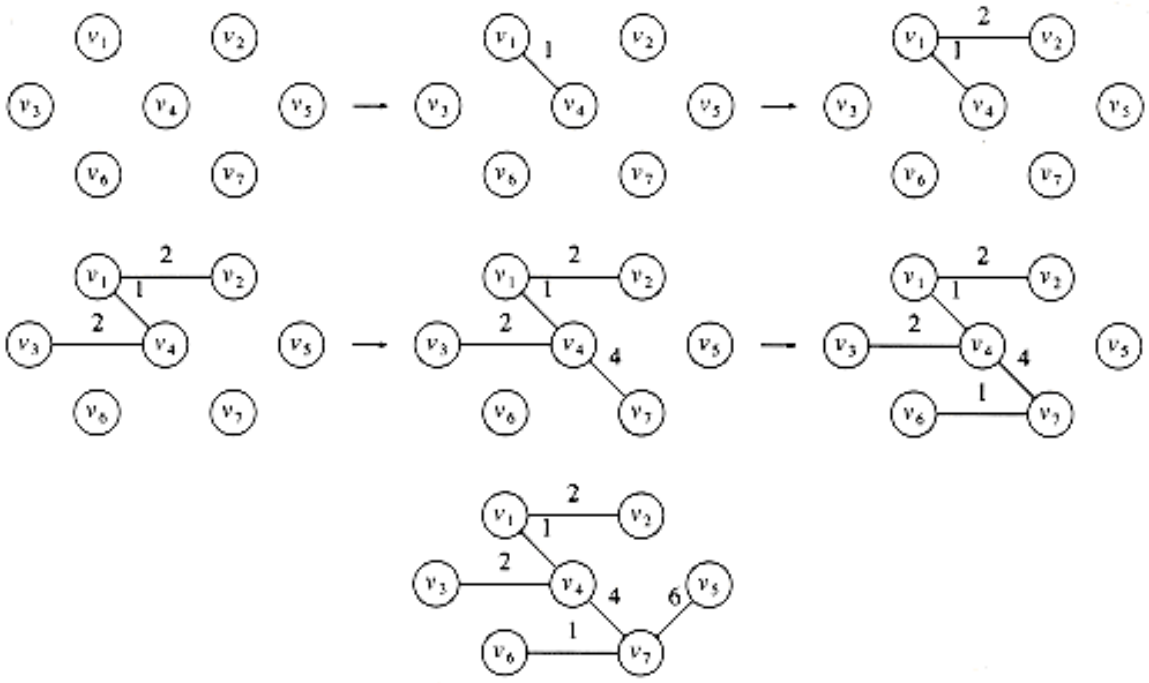
Prim's Algorithm

One way to compute a minimum spanning tree is to grow the tree in successive stages. In each stage, one node is picked as the root, and we add an edge, and thus an associated vertex, to the tree.

At any point in the algorithm, we can see that we have a set of vertices that have already been included in the tree; the rest of the vertices have not.

The algorithm then finds, at each stage, a new vertex to add to the tree by choosing the edge (u, v) such that the cost of (u, v) is the smallest among all edges where u is in the tree and v is not.

Figure shows how Prim's algorithm after each stage



This algorithm would build the minimum spanning tree. We can see that Prim's algorithm is essentially identical to Dijkstra's algorithm for shortest paths.

Steps:

a. Starting from v1, initially, v1 is in the tree as a root with no edges. Each step adds one edge and one vertex to the tree.
b. For each vertex we keep values dv and pv and an indication of whether it is known or unknown. dv is the weight of the shortest arc connecting v to a known vertex, and pv, as before, is the last vertex to cause a change in dv.
c. The rest of the algorithm is exactly the Structures, same, with the exception that since the definition of dv is different, so is the update rule.
d. After a vertex v is selected, for each unknown w adjacent to v, dv = min(dw, Cw,v).

The initial configuration of the table is shown below.

Initial configuration table

v	Known	dv	pv
v1	0	0	0
v2	0	∞	0
v3	0	∞	0
v4	0	∞	0
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

The vertices adjacent to v1 are v2, V3and v4. These vertices get their entries adjusted, as indicated below

After v1 is declared known

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	4	v1
v4	0	1	v1
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

Next, v4 is selected and marked known. Vertices v3, v5, v6, and v7 are adjacent.

After v4 is declared known

v	Known	dv	pv
v1	1	0	0
v2	0	2	v1
v3	0	2	v4
v4	1	1	v1
v5	0	7	v4
v6	0	8	v4
v7	0	4	v4

Next, v2 is selected. The cost of v5 from v2 is 10. But v5 already has minimum cost. So no change after v2 is visited. The v3 is declared to visit.

After v2 and then v3 are declared known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	2	v4
v4	1	1	v1
v5	0	7	v4
v6	0	5	v3
v7	0	4	v4

The next vertex selected v7.

After v7 is declared known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	2	v4
v4	1	1	v1
v5	0	6	v7
v6	0	1	v7
v7	1	4	v4

Next v6 is selected; no changes for the cost in the table. And v5 is declared as known.

After v6 and v5 are declared known

v	Known	dv	pv
v1	1	0	0
v2	1	2	v1
v3	1	2	v4
v4	1	1	v1
v5	1	6	v7
v6	1	1	v7
v7	1	4	v4

Now algorithm will be terminated. The edges in the spanning tree can be read from the table: (v2, v1), (v3, v4), (v4, v1), (v5, v7), (v6, v7), (v7, v4). The total cost is 16.

Routine for Prim's Algorithm

```
void graph :: dijkstra( Vertex S )
{
        for each vertex v
        {
         v.dist = INFINITY;
        v.known = false;
        }
        s.dist =0;

for( ; ; )
{
        v = smallest unknown distance vertex;
if(v== NotAVertex)
        break;
v. known = TRUE;

for each w adjacent to v
if( !w. known )
        if( v.dist + Cv,w < w.dist )
        {
                decrease( w.dist to v.dist + Cv,w );
                w.path = v;
                }
        }
}
```

Kruskal's Algorithm

A second greedy strategy is continually to select the edges in order of smallest weight and accept an edge if it does not cause a cycle.

Formally, **Kruskal's algorithm maintains a forest**. **Forest is a collection of trees.**

Procedure

- Initially, there are |V| single-node trees.
- Adding an edge merges two trees into one.
- When the algorithm terminates, there is only one tree, and this is the minimum spanning tree.
- The algorithm terminates when enough edges are accepted.

It is simple to decide whether edge (u,v) should be accepted or rejected. It is decided using union/find algorithm in set.

At any point in the process, two vertices belong to the same set if and only if they are connected in the current spanning forest. Thus, each vertex is initially in its own set.

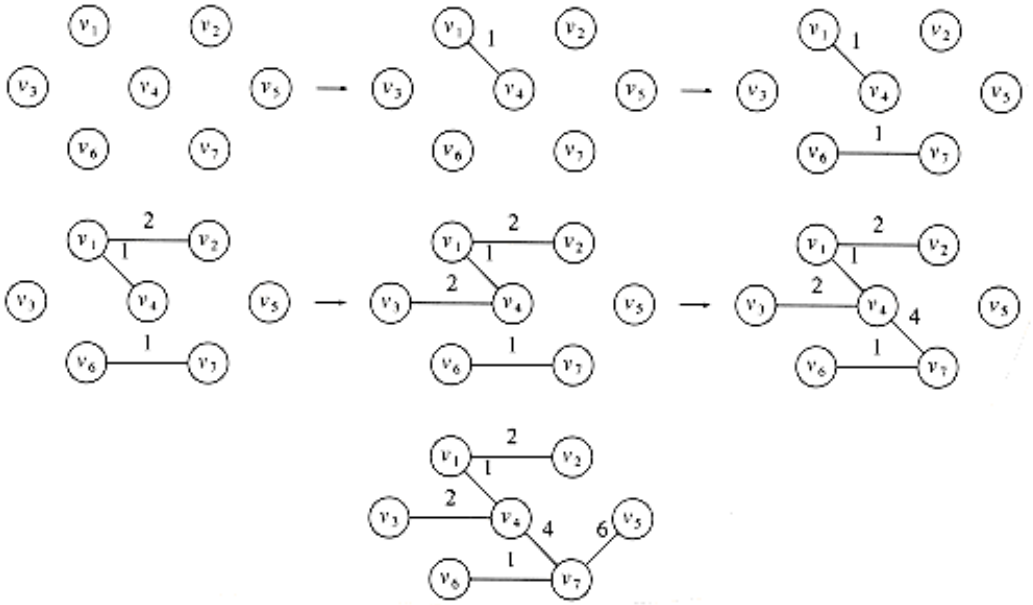
- If u and v are in the same set, **the edge is rejected**, because since they are already connected, adding (u, v) would form a cycle.
- Otherwise, **the edge is accepted**, and a union is performed on the two sets containing u and v.

It is easy to see that this maintains the set invariant, because once the edge (u, v) is added to the spanning forest, if w was connected to u and x was connected to v, then x and w must now be connected, and thus belong in the same set.

Action of Kruskal's algorithm on G

Edge	Weight	Action
(v1,v4)	1	Accepted
(v6,v7)	1	Accepted
(v1,v2)	2	Accepted
(v3,v4)	2	Accepted
(v2,v4)	3	Rejected
(v1,v3)	4	Rejected
(v4,v7)	4	Accepted
(v3,v6)	5	Rejected
(v5,v7)	6	Accepted

Kruskal's algorithm after each stage



Routine for Kruskal's algorithm

```
void Graph:: kruskal( )
{
        int edgesaccepted = 0;                  DISJSET ds ( Numvertex);
        PRIORIT_QUEUE < edge> pg( getedges ( ));
        Edge e;                                 Vertex U, V;
                while( edgesaccepted < NUMVERTEX-1 )
                {
                        Pq. deletemin( e );     // e = (u, v)
                        Settype Uset =ds. find( U, S );
                        Settype Vset = ds.find( V, S );
                        if( Uset != Vset )
                                {// accept the edge
                                edgesaccepted++;
                                ds.setunion( S, Uset, Vset );
                                }
                }
}
```

Graph Traversal:

Visiting of each and every vertex in the graph only once is called as **Graph traversal.**

There are two types of Graph traversal.

1. Depth First Traversal/ Search (DFS)
2. Breadth First Traversal/ Search (BFS)

Depth First Traversal/ Search (DFS)

Depth-first search is a generalization of preorder traversal. Starting at some vertex, v, we process v and then recursively traverse all vertices adjacent to v. If this process is performed on a tree, then all tree vertices are systematically visited in a total of O(|E|) time, since |E| = (|V|).

We need to be careful to avoid cycles. To do this, when we visit a vertex v, we mark it visited, since now we have been there, and recursively call depth-first search on all adjacent vertices that are not already marked.

The two important key points of depth first search

1. If path exists from one node to another node walk across the edge – **exploring the edge**
2. If path does not exist from one specific node to any other nodes, return to the previous node where we have been before – **backtracking**

Procedure for DFS

Starting at some vertex V, we process V and then recursively traverse all the vertices adjacent to V. This process continues until all the vertices are processed. If some vertex is not processed recursively, then it will be processed by using backtracking. If vertex W is visited from V, then the vertices are connected by means of tree edges. If the edges not included in tree, then they are represented by back edges. At the end of this process, it will construct a tree called as DFS tree.

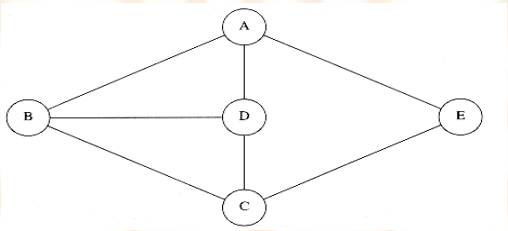
Routine to perform a depth-first search void

```
void dfs( vertex v )
{
visited[v] = TRUE;
for each w adjacent to v
if( !visited[w] )
dfs( w );
}
```

The (global) boolean array visited[ ] is initialized to FALSE. By recursively calling the procedures only on nodes that have not been visited, we guarantee that we do not loop indefinitely.

* An efficient way of implementing this is to begin the depth-first search at v1. If we need to restart the depth-first search, we examine the sequence vk, vk + 1, . . . for an unmarked vertex,where vk - 1 is the vertex where the last depth-first search was started.

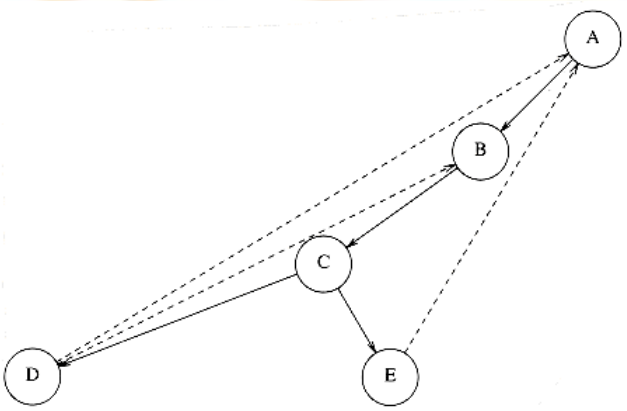
An undirected graph



Steps to construct depth-first spanning tree

a. We start at vertex A. Then we mark A as visited and call dfs(B) recursively. dfs(B) marks B as visited and calls dfs(C) recursively.
b. dfs(C) marks C as visited and calls dfs(D) recursively.
c. dfs(D) sees both A and B, but both these are marked, so no recursive calls are made. dfs(D) also sees that C is adjacent but marked, so no recursive call is made there, and dfs(D) returns back to dfs(C).
d. dfs(C) sees B adjacent, ignores it, finds a previously unseen vertex E adjacent, and thus calls dfs(E).
e. dfs(E) marks E, ignores A and C, and returns to dfs(C).
f. dfs(C) returns to dfs(B). dfs(B) ignores both A and D and returns.
g. dfs(A) ignores both D and E and returns.

Depth-first search of the graph



---------> Back edge ⟶ Tree edge

The root of the tree is A, the first vertex visited. Each edge (v, w) in the graph is present in the tree. If, when we process (v, w), we find that w is unmarked, or if, when we process (w, v), we find that v is unmarked, we indicate this with a **tree edge.**

If when we process (v, w), we find that w is already marked, and when processing (w, v), we find that v is already marked, we draw a dashed line, which we will call a **back edge**, to indicate that this "edge" is not really part of the tree.

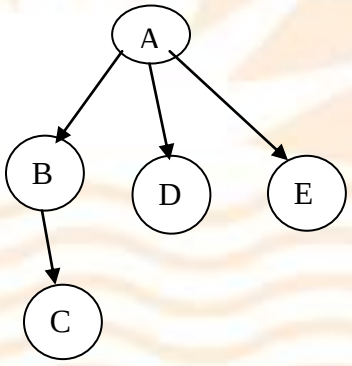
Breadth First Traversal (BFS)

Here starting from some vertex v, and its adjacency vertices are processed. After all the adjacency vertices are processed, then selecting any one the adjacency vertex and process will continue. If the vertex is not visited, then backtracking is applied to visit the unvisited vertex.

Routine:

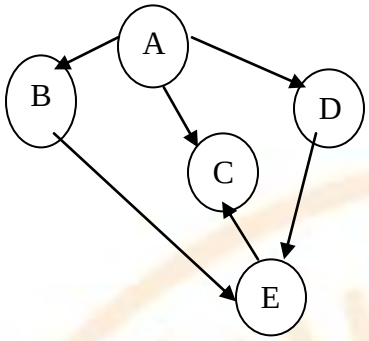
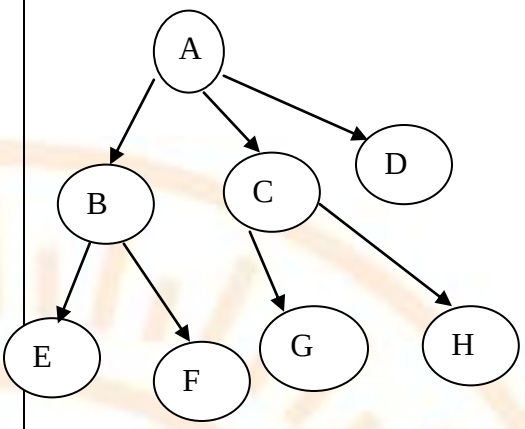
```
void  BFS (vertex v)
{
visited[v]= true;
For each w adjacent to v
If (!visited[w])
visited[w] = true;
}
```

Example: BFS of the above graph



Difference between DFS & BFS

S. No	DFS	BFS
1	Back tracking is possible from a dead end.	Back tracking is not possible.
2	Vertices from which exploration is incomplete are processed in a LIFO order.	The vertices to be explored are organized as a FIFO queue.
3	Search is done in one particular direction at the time.	The vertices in the same level are maintained parallel. (Left to right alphabetical ordering)

4	 Order of traversal: A→B→C→D→E	 Order of traversal: A→B→C→D→E→F→G→H