UNIT 1 OBJECT ORIENTED PROGRAMMING FUNDAMENTALS

PART-A

1. Distinguish between methods and messages (APRIL/MAY 2011)

Message:

Objects communicate by sending messages to each other. A message is sent to invoke a method.

Message is refer to instruction that is send to object which will invoke the related method.

Method:

Provides response to a message. It is an implementation of an operation. Method is a function or procedure that is defined for a class and typically can access the internal state of an object of that class to perform some operation

2. Define abstraction and encapsulation. (APRIL/MAY 2011) (NOV/DEC 2014) (NOV/DEC 2013)

Wrapping up of data and function within the structure is called as encapsulation

The insulation of data from direct access by the program is called as data hiding or information binding.

3. Justify the need for static members (APRIL/MAY 2011)

Static variable are normally used to maintain values common to the entire
class. Feature:

• It is initialized to zero when the first object i s created. No other initialization is permitted

• only one copy of that member is created for the e ntire class and is shared by all the objects

• It is only visible within the class, but its life time is the entire class type and scope of each static member variable must be defined outside the class

• It is stored separately rather than objects

Eg: static int count//count is initialized to zero when an object is
created. int classname::count;//definition of static data member

4. What is the difference between a local variable and a data member? (NOV/DEC 2011)

local variable

- A variable is something referred to by a reference, such as int age (an integer named age).
- A variable can be a primitive or an object reference

data member

Member data is/are variable(s) that belong to an object. A cat object for instance could have member data such as a string color and int age. Each cat object can then store, maintain and provide upon request its own information regarding its color and age.

5. Explain the purpose of a function parameter. (NOV/DEC 2011)

A function parameter, also called an argument, is a variable that you provide a function when you call it. For example, here is a function:

```
void printChar(char c)
{
cout << c << endl;
}
```

6. What is the difference between a parameter and an argument? (NOV/DEC 2011)

An argument is something passed into a function (value), whereas a parameter is the type of data plus the name. For example:

```
int main ()
{ int x = 5;
int y = 4;
return sum(x, y);
}
int sum(int one, int two)
{ return one + two;
}
```

In main() you are passing 5 and 4, those are arguments. sum() takes in two integers, those are the parameters (necessary conditions to be met so the function can be executed).

7. What is data hiding?

The insulation of data from direct access by the program is called as data hiding or information binding.

The data is not accessible to the outside world and only those functions, which are wrapped in the class, can access it.

8. What are the advantages of default arguments?

A default parameter is a function parameter that has a default value provided to it. If the user does not supply a value for this parameter, the default value will be used. If the user does supply a value for the default parameter, the user-supplied value is used.

Consider the following program:

```
void PrintValues(int nValue1, int nValue2=10)
{
   using namespace std;
   cout << "1st value: " << nValue1 << endl;
   cout << "2nd value: " << nValue2 << endl;
}
 int main()
{
   PrintValues(1); // nValue2 will use default parameter of 10
   PrintValues(3, 4); // override default value for nValue2
}
```

This program produces the following

output: 1st value: 1

2nd value: 10

1st value: 3

2nd value: 4

9. What is Procedure oriented language?

Conventional programming, using high-level language such as COBOL, FORTRAN and C are commonly known as Procedure oriented language (POP). In POP number of functions is written to accomplish the tasks such as reading, calculating and printing**.**

10. Write any four features of OOPS.

• Emphasis is on data rather than on procedure.

• Programs are divided into objects.

• Data is hidden and cannot be accessed by external functions.

• Follows bottom -up approach in program design.

11. What are the basic concepts of OOPS?

• Objects.

• Classes.

• Data abstraction and Encapsulation.

• Inheritance.

• Polymorphism.

• Dynamic binding.

• Message passing.

12. What is a class? (NOV/DEC 2013)

• The entire set of data and code of an object can be made a user-defined data type with the help of a class.

• Once a class has been defined, we can create any number of objects belonging to the classes.

• Classes are user-defined data types and behave li ke built-in types of the programming language.

13. What are objects? (NOV/DEC 2013)

Objects are basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. Each object has the data and code to manipulate the data and theses objects interact with each other.

14. What is dynamic binding or late binding?

Binding refers to the linking of a procedure to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at the run-time.

15. Give any four applications of OOPS

- Real-time systems.
- Simulation and modeling.
- Object-oriented databases.
- AI and expert systems.

16. What is a scope resolution operator?

Scope resolution operator is used to uncover the hidden variables. It also allows access to global version of variables. Eg:

```
#include<iostream. h>
int m=10; // global variable
m void main ( )
{
int m=20; // local variable m
cout<<"m="<<m<<"\n";
cout<<": : m="<<: : m<<"\n";
}
```

output:

20

10 (: : m access global m)

Scope resolution operator is used to define the function outside the class. Syntax:

Return type <class name> : : <function name> Eg:

Void x : : getdata()

17. What are free store operators (or) Memory management operators?

New and Delete operators are called as free store operators since they allocate the memory dynamically. New operator can be used to create objects of any data type.

Pointer-variable = new data type;

Initialization of the memory using new operator can be done. This can be done as,

Pointer-variable = new data-type(value)

Delete operator is used to release the memory space for reuse. The general form of its use is

Delete pointer-variable;

18. What are manipulators?

Setw, endl are known as manipulators. Manipulators are operators that are used to format the display. The endl manipulator when used in an output statement causes a linefeed to be inserted and its effect is similar to that of the newline character"\n".

Eg:Cout<<setw(5)<<sum<<endl;

19. What do you mean by enumerated datatype?

An enumerated datatype is another user-defined data type, which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The syntax of an enum statement is similar to that of the struct statement

Eg:

enum shape{ circle, square, triangle} ; enum color{ red, blue, green, yellow}

20.**Give the significance of declaring member of a class static (NOV/DEC 2015)**

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present.

21.**What is the use of "this" pointer? (NOV/DEC 2015)**

The 'this' pointer is passed as a hidden argument to all non static member function calls and is available as a local variable within the body of all non static functions. 'this' pointer is a constant pointer that holds the memory address of the current object.

22.With which concept, visibility and lifetime of the variables differ in c++. List its types(APRIL/MAY 2015)

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

PART-B

1. i) Highlight the features of object oriented programming language. (8) (NOV/DEC 2013)

The characteristics of OOP are:

- Class definitions – Basic building blocks OOP and a single entity which has data and operations on data together
- Objects – The instances of a class which are used i n real functionality – its variables and operations
  - Abstraction – Specifying what to do but not how to do ; a flexible feature for having a overall view of an object's functionality.
  - Encapsulation – Binding data and operations of data together in a single unit – A class adhere this fe ature
  - Inheritance and class hierarchy – Reusability and e xtension of existing classes
  - Polymorphism – Multiple definitions for a single na me - functions with same name with different functionality; saves time in investing many function names Operator and Function overloading
    Generic classes – Class definitions for unspecified data. They are known as container classes. They are flexible and reusable.
  - Class libraries – Built-in language specific classe s
  - Message passing – Objects communicates through invo king methods and sending data to them. This feature of sending and receiving information among objects through function parameters is known as Message Passing.

ii) Explain function overloading with an example. (8) (APRIL/MAY 2011) (NOV/DEC 2012)

A single function name can be used to perform different types of tasks. The same function name can be used to handle different number and different types of arguments. This is known as function overloading or function polymorphism.

```
#include<iostream.h>
#include<conio.h>
void swap(int &x,int &y)
{
int t;
t=x;
x=y;
y=t;
}
void swap(float &p,float &q)
{
float t;
t=p;
p=q;
q=t;
}
void swap(char &c1,char &c2)
{
char t;
t=c1;
c1=c2;
c2=t;
}
void main()
{
int i,j;
float a,b;
char s1,s2;
clrscr();
```

```
cout<<"\n Enter two integers : \n";
cout<<" i = ";
cin>>i;
cout<<"\n j = ";
cin>>j;
swap(i,j);
cout<<"\n Enter two float numbers : \n";
cout<<" a = ";
cin>>a;
```

2. Consider a Bank Account class with Acc No. and balance as data members. Write a C++ program to implement the member functions get_Account_Details ( ) and display_Account_Details ( ). Also write a suitable main function. (16) (APRIL/MAY 2011)

```
#include <iostream.h>
#include <conio.h>
class bank
{
int acno;
int bal;
public :
void get_Account_Details();
void display_Account_Details();
};
void bank :: get_Account_Details ()
{
cout<<"Enter A/c no. :-";
cin>>acno;
cout<<"Enter Balance:-";
cin>>bal;
}
void bank :: display_Account_Details ()
{
cout)<<"Account Details"<<endl;
cout)<<"A/c. No. "<<acno<<endl;
coutBalance "<<bal<<endl;
}
void main()
{
clrscr();
bank o1;
o1. get_Account_Details ();
o1. display_Account_Details ();
}
```

3. Write a C++ program to explain how the member functions can be accessed using pointers.(16) (APRIL/MAY 2011)

• C++ pointer can also have address for an member functions. They are known as pointer to member function.
• Pointer to function is known as call back functions.
• The size of the object is determined by the size of all non-static data members. Pointer to member use either
• star-dot combination
• arrow notation

Syntax: Class_name *Ptr_var;
Ptr_var=new student;

```
class student
  {
  public: int
  rollno;
```

```
string name;
void print()
{
cout<<”rollno”<<rollno;
cout<<”name”<<name;
}};
void main() //Star-dot notation
{
student *stu-ptr; stu-
ptr=new student; (*stu-
ptr).rollno=1; (*stu-
ptr).name=”xxx”; (*stu-
ptr).print();
}
```

4. i) Write a C++ program that calculates and prints the sum of the integers from 1 to 10 (8) (NOV/DEC 2011)

```
#include <iostream>
int main() {
        int num[10],sum=0; for
        (int i=0;i<10;i++)
          {
                cout<<"enter a number: ";
                cin>>num[i]; sum+=num[i];
                }
        cout<<"The sum of all these numbers is "<<sum<<"\n";
        return 0;

}
```

ii) Write a C++ program to calculate x raised to the power y. (8) (NOV/DEC 2011)

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
void main()
{
   int x,y;
   double power();
   clrscr();
   cout<<"Enter value of x : ";
   cin>>x;
   cout<<"Enter value of y : ";
   cin>>y;

   cout<<x<<” to power”<<y<<” is = “<<power(x,y);
 getch();
}
double power(x,y)
int x,y;
{
   double
   p; p=1.0;
   if(y>=0)
      while(y--
        ) p*=x;
   else
      while(y++
```

```
    )
        p/=x;
  return(p);
```

5.i) Explain default arguments with example (8)

Default arguments assign a default value to the parameter, which does not have matching argument in the function call.

Default values are specified when the function is declared.

The advantages of default arguments are,

• We can use default arguments to add new parameter s to the existing function.

• Default arguments can be used to combine similar functions into one.

```
#include<iostream.h>
int volume(int length, int width = 1, int height = 1);
int main()
{
int a,b,c;
a=solume(4, 6, 2);
b=volume(4, 6);
c=volume(4);
return 0;
}
int volume(int length, int width=1, int height=1)
{
int v;
v=length*width*height;
return v;
}
```

ii) What is a static member and list its common characteristics (8)

Static variable are normally used to maintain values common to the entire class.

Feature:

• It is initialized to zero when the first object i s created. No other initialization is permitted

• only one copy of that member is created for the e ntire class and is shared by all the objects

• It is only visible within the class, but its life time is the entire class type and scope of each static member variable must be defined outside the class

• It is stored separately rather than objects

Eg: static int count//count is initialized to zero when an object is created.

int classname::count;//definition of static data member

```
#include<iostream.h>
#include<conio.h>
 class stat
{
   int code;


   static int
   count; public:
   stat()
   {
    code=++count;
   }
   void showcode()
   {
    cout<<"\n\tObject number is :"<<code;
   }
   static void showcount()
   {
          cout<<"\n\tCount Objects :"<<count;
```

```
    }
};
int
stat::count;
void main()
{
  clrscr();
  stat obj1,obj2;

  obj1.showcount();
  obj1.showcode();
  obj2.showcount();
  obj2.showcode();
  getch();
}
```

Output:

```
Count Objects: 2
Object Number is: 1
Count Objects: 2
Object Number is: 2
```

6. Explain in detail about Class, Objects, Methods and Messages. (16) (MAY/JUNE 2012)

Class is a collection of objects of similar data types. Class is a user-defined data type. The entire set of data and code of an object can be made a user defined type through a class.

Objects are basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. Each object has the data and code to manipulate the data and theses objects interact with each other.

A method (or member function) is a subroutine (or procedure or function) associated with an object, and which has access to its data, its member variables.

Objects communicate between each other by sending and receiving information known as messages. A message to an object is a request for execution of a procedure. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

```
#include<iostream
 > class sample
{
  private:
    int
    var;
  public:
     void input_value()
    {
      cout << "Enter an integer\n";
      cin >> var;
    }
     void output_value()
    {
      cout << "Integer is ";
      cout << var << "\n";
    }
};
 void main()
{
  sample s;
  s.input_value();
  s.output_value();
}
```

7.i) List the different types of constructor. Write a program to illustrate the use of different types of constructor (10) (NOV/DEC 2014)

PARAMETERIZED CONSTRUCTORS

The constructor that can take arguments are called parameterized constructor. The arguments can be separated by commas and they can be specified within braces similar to the argument list in function.

When a constructor has been parameterized, the object declaration without parameter may not work. In case of parameterized constructor, we must provide the appropriate arguments to the constructor when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The implicit call method is sometimes known as shorthand method as it is shorter and is easy to implement.

Program : creating parameterized constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
private:

int roll,age, marks;
public:
student(int r, int m, int a);                    //parameterized constructor
void display( )
{
cout<<"\nRoll number :" <<roll <<endl;
cout<<"Total marks : <<marks<<endl;
cout<<"Age:"<<age<<endl;
}
};                                               //end of class declaration
student : : student(int r, int m, int a)         //constructor definition
{
roll = r;
marks =m;
age=a;
}
int main( )
{
Student manoj(5,430,16);                         //object creation
Cout<<"\n Data of student 1:"<<endl;
manoj.display();
Student ram(6,380,15);                           //object creation
Cout<<"\n Data of student 1:"<<endl;
ram.display();
getch();
return 0;
}
```

```
Data of student 1:
Roll number      :    5
Total Marks      :    430
Age              :    16
```

```
Data of student 2:
Roll number     :   6
Total Marks     :   380
Age             :   15
```

COPY CONSTRUCTOR

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.

Syntax:

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

Program : Copy Constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
private:
int roll, marks, age;
public:
student(int r,int m,int a)                    //parameterized constructor
{
rol l = r;
marks = m;
age = a;
}
student(student &s)                           //copy constructor
{
roll = s.roll;

marks = s.marks;
age=s.age;
}
void display( )
{
cout<<"Roll number :" <<roll <<endl;
cout<<"Total marks :"<<marks<<endl;
cout<<"Age:"<<age<<endl;
}
};
int main( )
{
clrscr();
student t(3,350,17);                          // or student k(t);
student k = t;                                // invokes copy constructor
cout<<"\nData of student t:"<<endl;
t.display();
cout<<"\n\nData of student k:"<<endl;
k.display();
getch();
```

```
return 0;
}
```

The outout of the above program will be like this:

```
Data of student t   :
Roll number         :   3
Total marks         :   350
Age                 :   17
Data of student k   :
Roll number         :   3
Total marks         :   350
Age                 :   17
```

DEFAULT CONSTRUCTOR

n C++, the standard describes the default constructor for a class as a constructor that can be called with no arguments (this includes a constructor whose parameters all have default arguments) For example:

```
class MyClass
{
public:
MyClass();  // constructor declared

private:
int
x;};

MyClass :: MyClass()  // constructor defined
{
x = 100;
}

int  main()
{
MyClass m;  // at runtime, object m is created, and the default constructor is called
}
```

DYNAMIC CONSTRUCTORS

The constructor 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. 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.

Program: illustrate the working of dynamic constructors:

```
#include<iostream.h>
 class Sample
    {
          char *name;
          int length;
      public:
           Sample( )
             {
                length = 0;
                name = new char[ length + 1];
             }
          Sample ( char  *s )
             {
```

```
            length = strlen(s);
            name = new char[ length + 1 ];
            strcpy( name , s );
        }
      void display( )
       {
            cout<<name<<endl;
       }
   };
int  main( )
  {
            char  *first = " C++ ";
            Sample S1(first), S2("ABC" ),
            S3("XYZ"); S1.display( );
            S2.display( );
            S3.display( );
            return 0;
  }
```

RESULT :

```
C++
ABC
XYZ
```

ii) Brief on the features of C++ programming features (6) (NOV/DEC 2014)

- Emphasis is on data rather than on procedure.
- Programs are divided into objects.
- Data is hidden and cannot be accessed by external functions.
- Follows bottom -up approach in program design.
- Functions that operate on the data of an object are tied together in the data structure.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.

8.i) Explain the ways in which member functions of a class can be defined and called using a suitable example (10) (NOV/DEC 2014)

Member Functions in Classes

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

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

Example :

```
class Cube
{
 public:
 int side;
 int getVolume(); // Declaring function getVolume with no argument and return type int. };
```

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

```
class Cube
{
 public:
 int side;
```

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

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

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

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

The main function for both the function definition will be same. Inside main() we will create object of class, and will call the member function using dot . operator.

```
int main()
{
 Cube C1;
 C1.side=4;  // setting side value
 cout<< "Volume of cube C1 ="<< C1.getVolume();
}
```

Similarly we can define the getter and setter functions to access private data members, inside or outside the class definition.

ii) Explain with an example the use of static members (6) (NOV/DEC 2014)

- We can define class members static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
- A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Static Data Member: It is generally used to store value common to the whole class. The static data member differsfrom an ordinary data member in the following ways :

(i) Only a single copy of the static data member is used by all the objects.

(ii) It can be used within the class but its lifetime is the whole program. For making a data member static, we require :

(a) Declare it within the class.

(b) Define it outside the class.

For example

```
Class student
{
Static int count;          //declaration within class
---------------
---------------
};
```

The static data member is defined outside the class as

```
 int student :: count;          //definition outside class
```

9. i) Explain constant data member and member function with example (8)

Constant Data Member

These are data variables in class which are made const. They are not initialized during declaration. Their initialization occur in the constructor.

Example:

```
class Test
{
 const int i; // Constant Data Member public:

 Test (int x)
 {
  i=x;
 }
};
int main()
{
 Test t(10);
 Test s(20);
}
```

In this program, **i** is a const data member, in every object its independent copy is present, hence it is initialized with each object using constructor. Once initialized, it cannot be changed.

Constant class Member function

A const member function never modifies data members in an object.

Syntax :

return_type function_name() const;

Example for const Object and const Member function

```
class X
{
 int i;
 public:
 X(int x)                    // Constructor
 {
 i=x;
 }
 int f() const               // Constant function
 {
 i++;
}
 int g()
{
 i++;
}
};
int main()
{
X obj1(10);                  // Non const Object
const X obj2(20);            // Const Object
obj1.f();                    // No error
obj2.f();                    // No error
cout << obj1.i << obj2.i ;
obj1.g();                    //No error
obj2.g();                    // Compile time error
}
```

Output : 10 20

Here, we can see, that const member function never changes data members of class, and it can be used with both const and non-const object. But a const object can't be used with a member function which tries to change its data members.

ii) Explain references with example (8)

```
#include<iostream>
using namespace std;
void swap(int &x,int
&y)
{
int
t;
t=x
;
x=
y;
y=t
;
}
int  main()
{
int a,b;
cout<<" enter two
integers<a,b>"; cin>>a>>b;
swap(a,b);
cout<<"value of a & b on swap(a,b) in main():"<<a<<" "<<b;
return 0;
}
```

OUTPUT

```
 enter two integers<a,b>25 9
value of a & b on swap(a,b) in main():9 25
```

10.i)What are the differences between pointer to constant and constant pointers? Give an example program and explain it.(10) (APRIL/MAY 2015)

A pointer is a special variable which holds the address of a variable of same type. For example : Lets say there is a variable 'int a=10' so,a pointer to variable 'a' can be defined as 'int* ptr = &a'.

'&' operator gives you the address of the variable on which this operator is applied

* operator is the value-of operator and fetches the value kept at address held by the variable it is applied on. For example, in the above example, *ptr will give the value of 'a' ie 10.
Constant pointers
Pointer to Constants

1) Constant Pointers : These type of pointers are the one which cannot change address they are pointing to. This means that suppose there is a pointer which points to a variable (or stores the address of that variable). Now if we try to point the pointer to some other variable (or try to make the pointer store address of some other variable), then constant pointers are incapable of this.
A constant pointer is declared as : 'int *const ptr' (the location of 'const' make the pointer 'ptr' as constant pointer)

2) Pointer to Constant : These type of pointers are the one which cannot change the value they are pointing to. This means they cannot change the value of the variable whose address they are holding.
A pointer to a constant is declared as : 'const int *ptr' (the location of 'const' makes the pointer 'ptr' as a pointer to constant.

1) Constant pointers :

```
#include<stdio.h>

int main(void)
{
   int a[] = {10,11};
   int* const ptr = a;

   *ptr = 11;

   printf("\n value at ptr is  : [%d]\n",*ptr);
   printf("\n Address pointed by ptr  : [%p]\n",(unsigned int*)ptr);

   ptr++;
   printf("\n Address pointed by ptr  : [%p]\n",(unsigned int*)ptr);

   return 0;
}
```

2) Pointer to Constants

```
#include<stdio.h>

int main(void)
{
   int a = 10;
   const int* ptr = &a;


   printf("\n value at ptr is  : [%d]\n",*ptr);
   printf("\n Address pointed by ptr  : [%p]\n",(unsigned int*)ptr);

   *ptr = 11;

   return 0;
}
```

ii)Explain the role of this pointer with a suitable program.(6)(APRIL/MAY 2015) (APRIL/MAY 2016)

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.

```
#include <iostream>

using namespace std;

class Box
{
  public:
    // Constructor definition
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
      cout <<"Constructor called." << endl;
```

```
      length = l;
      breadth = b;
      height = h;
    }
    double Volume()
    {
      return length * breadth * height;
    }
    int compare(Box box)
    {
      return this->Volume() > box.Volume();
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

int main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  if(Box1.compare(Box2))
  {
    cout << "Box2 is smaller than Box1" <<endl;
  }
  else
  {
    cout << "Box2 is equal to or larger than Box1" <<endl;
  }
  return 0;
}
```

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

```
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1
```

11.i)How can you specify a class? (4) (NOV/DEC 2015)

Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.

An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are defined using either keyword class or keyword struct, with the following syntax:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

ii) Describe different mechanisms of accessing data members and member functions in a class with a suitable example. (12) (NOV/DEC 2015)

Accessing Data Members of Class

Accessing a data member depends solely on the access control of that data member. If its public, then the data member can be easily accessed using the direct member access (.) operator with the object of that class.

If, the data member is defined as private or protected, then we cannot access the data variables directly. Then we will have to create special public member functions to access, use or initialize the private and protected data members. These member functions are also called Accessors and Mutator methods or getter andsetter functions.

Accessing Public Data Members

Following is an example to show you how to initialize and use the public data members using the dot (.) operator and the respective object of class.

```
class Student
{
 public:
 int rollno;
 string name;
};

int main()
{
 Student A;
 Student B;
 A.rollno=1;
 A.name="Adam";

 B.rollno=2;
 B.name="Bella";

 cout <<"Name and Roll no of A is :"<< A.name << A.rollno;
 cout <<"Name and Roll no of B is :"<< B.name << B.rollno;
}
```

Accessing Private Data Members

To access, use and initialize the private data member you need to create getter and setter functions, to get and set the value of the data member.

The setter function will set the value passed as argument to the private data member, and the getter function will return the value of the private data member to be used. Both getter and setter function must be defined public.

Example :

```
class Student
{
 private:    // private data member
 int rollno;
```

```
public:     // public accessor and mutator functions
int getRollno()
{
 return rollno;
}

void setRollno(int i)
{
 rollno=i;
}

};

int main()
{
 Student A;
 A.rollono=1;  //Compile time error
 cout<< A.rollno; //Compile time error

 A.setRollno(1);  //Rollno initialized to 1
 cout<< A.getRollno(); //Output will be 1
}
```

12. i)Explain the different types of constructor with suitable examples(10) (NOV/DEC 2015)

Constructors
Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created. Constructors iitialize values to object members after storage is allocated to the object.

```
class A
{
 int x;
 public:
 A();  //Constructor
};
```

While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.
Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A
{
 int i;
 public:
 A(); //Constructor declared
};

A::A()   // Constructor definition
{
 i=1;
}
```

Types of Constructors
Constructors are of three types :

1. Default Constructor

2. Parametrized Constructor
3. Copy COnstructor

Default Constructor

Default constructor is the constructor which doesn't take any argument. It has no parameter.
Syntax :

```
class_name ()
{ Constructor Definition }
```

Example :

```
class Cube
{
int side;
public:
Cube()
 {
  side=10;
 }
};

int main()
{
Cube c;
cout << c.side;
}
```

Output : 10
In this case, as soon as the object is created the constructor is called which initializes its data members.
A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
 int side;
};

int main()
{
 Cube c;
 cout << c.side;
}
```

Output : 0
In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 in this case.

Parameterized Constructor

These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.
Example :

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

int main()
```

```
{
 Cube c1(10);
 Cube c2(20);
 Cube c3(30);
 cout << c1.side;
 cout << c2.side;
 cout << c3.side;
}
```

OUTPUT : 10 20 30

By using parameterized construcor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

ii)Describe the different types of storage classes (6) (NOV/DEC 2015)

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.
- This means that the variable has a maximum size equal to the register size and can't have the unary '& operator applied to it (as it does not have a memory location).

```
{
  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.h>
void func(void);
static int count = 10; /* Global variable */
main()
{
   while(count--)
   {
     func();
   }
   return 0;
}
// Function definition
void func( void )
{
   static int i = 5; // local static variable
   i++;
   cout << "i is " << i ;
   cout << " and count is " << count <<endl;
}
```

Output:

```
i is 6 and count is 9 i
is 7 and count is 8 i
is 8 and count is 7 i
is 9 and count is 6 i
is 10 and count is 5 i
is 11 and count is 4 i
is 12 and count is 3 i
is 13 and count is 2 i
is 14 and count is 1 i
is 15 and count is 0
```

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 'extern' is used the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

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

The mutable Storage Class

- The **mutable** specifier applies only to class objects.

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

UNIT II OBJECT ORIENTED PROGRAMMING CONCEPTS
PART –A

1. What is a constructor? How do we invoke a constructor function? (NOV/DEC 2013)

- A constructor is a special method of a class or structure in object-oriented programming that initializes an object of that type.
- Constructor will be triggered automatically when an object is created.
- Purpose of the constructor is to initialize an object of a class.

2. What is the significance of overloading through friend functions? (MAY/JUNE 2011)

- Friend function offer better flexibility which is not provided by the member function of the class.
- The difference between member function and friend function is that the member function takes arguments explicitly.
- The friend function needs the parameters to be explicitly passed.
- The syntax of operator overloading with friend function is as follows:

```
friend return-type operator operator-symbol(variable1, variable2)
{
statement1;
statement2;
}
```

3. Define run time polymorphism (MAY/JUNE 2011)

Runtime polymorphism is a form of polymorphism at which function binding occurs at runtime. This means that the exact function that is bound to need not be known at compile time.

4. What is a copy constructor?

- A copy constructor is a special constructor in the C++ programming language for creating a new object as a copy of an existing object.
- The first argument of such a constructor is a reference to an object of the same type as is being constructed, which might be followed by parameters of any type.

5. What are the operators that cannot be overloaded? (NOV/DEC 2014)

In C++, following operators 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)

6. What are pure virtual function? Give example (NOV/DEC 2014)

A pure virtual function is a function that has notation "= 0" in the declaration of that function. Example:

```
#include <iostream>
using namespace std;
  class number
{ protected:
 int val;
```

```
public:
  void setval(int i) { val = i; }
     // show() is a pure virtual
  function virtual void show() = 0;
};
   class hextype : public number
{ public:
  void show() {
    cout << hex << val << "\n";
  }
};
class dectype : public number
{ public:
  void show() {
    cout << val <<
  "\n"; }};
class octtype : public number
{ public:
  void show() {
    cout << oct << val << "\n";
  }
};
   int main()
{
dectype d;
  hextype h;
  octtype o;
     d.setval(20);
  d.show(); // displays 20 -
     decimal h.setval(20);
  h.show(); // displays 14 –
     hexadecimal o.setval(20);
  o.show(); // displays 24 -
     octal return 0;
}
```

7. When do we make a class virtual?

In multiple inheritance situations, When class A is inherited by class B and class C, and class D inherits from both B and C, then it inherits two instances of A which introduces ambiguity. By declaring A to be virtual in both B and C, then D inherits directly from A, while B and C share the same instance if A.

8. What is the need for initialization of objects using constructors? (NOV/DEC 2012)

Constructors are the special type of member function that initializes the object automatically when it is created Compiler identifies that the given member function is a constructor by its name and return type. Constructor has same name as that of class and it does not have any return type.

9. What is destructor? (NOV/DEC 2012)

A destructor is a special method called automatically during the destruction of an object. Actions executed in the destructor include the following:

- Recovering the heap space allocated during the lifetime of an object
- Closing file or database connections
- Releasing network resources
- Releasing resource locks

- Other housekeeping tasks

10. What is inheritance? What are the types of inheritance? (NOV/DEC 2012)

Inheritance is the concept that when a class of objects is defined, any subclass that is defined can inherit the definitions of one or more general classes. This means for the programmer that an object in a subclass need not carry its own definition of data and methods that are generic to the class (or classes) of which it is a part. This not only speeds up program development; it also ensures an inherent validity to the defined subclass object (what works and is consistent about the class will also work for the subclass).

The Various Types of Inheritance those are provided by C++ are as followings: Single Inheritance
Multilevel Inheritance
Multiple Inheritance
Hierarchical
Inheritance Hybrid
Inheritance

11. What are Friend functions? Write the syntax

A function that has access to the private member of the class but is not itself a member of the class is called friend functions.

The general form is **friend data_type function_name( );**

Friend function is preceded by the keyword 'friend' .

12. Define default constructor

The constructor with no arguments is called default
constructor Eg:

```
Class integer
{
int
m,n;
Public:
Integer
( ); };
integer::integer( )//default constructor
{
m=0;n=0;
}
```

the statement **integer a;**
invokes the default
constructor

13. Define parameterized constructor

constructor with arguments is called parameterized
constructor Eg;

```
Class
integer {
int m,n;
public:
integer(int x,int y)
{
 m=x;n=y;
}
```

To invoke parameterized constructor we must pass the initial values as arguments to the constructor function when an object is declared. This is done in two ways

1.By calling the constructor
explicitly eg: integer
int1=integer(10,10);
2.By calling the constructor
implicitly eg: Integer int1(10,10);

14. Define default argument constructor

The constructor with default arguments are called default argument
constructor Eg:
Complex(float real,float imag=0);
The default value of the argument imag
is 0 The statement complex a(6.0)
assign real=6.0 and
imag=0 the statement
complex a(2.3,9.0)
assign real=2.3 and imag=9.0

15. Define dynamic constructor

Allocation of memory to objects at time of their construction is known as dynamic constructor. The memory is allocated with the help of the NEW operator
Eg:

```
Class string
{
char *name; int
length; public:
string( )
{
length=0; name=new char[ length +1];
}
void main( )
{
string name1("Louis"),name3(Lagrange);
}
```

16. What is operator overloading?

C++ has the ability to provide the operators with a special meaning for a data type. This mechanism of giving such special meanings to an operator is known as Operator overloading. It provides a flexible option for the creation of new definitions for C++ operators.

17. Define multiple constructors (constructor overloading).

The class that has different types of constructor is called multiple
constructors Eg:

```
#include<iostream.
h>
#include<conio.h>
class integer
{
int m,n;
public:
integer( ) //default constructor
{
m=0;
n=0;
}
```

```
integer(int a,int b) //parameterized constructor
{
m=a; n=b;
}
integer(&i) //copy constructor
{
m=i. m;
n=i.n;
}
void main()
{
integer i1; //invokes default constructor
integer i2(45,67);//invokes parameterized constructor
integer i3(i2); //invokes copy constructor
}
```

18. Write some special characteristics of constructor

- They should be declared in the public section
- They are invoked automatically when the objects are created
- They do not have return types, not even void and therefore, and they cannot return values
- They cannot be inherited, though a derived class can call the base class
- They can have default arguments
- Constructors cannot be virtual f unction

19.**How the C string differs from a C++ type string? (NOV/DEC 2015)(APRIL/MAY 2016)**

In C, strings are just char arrays which, by convention, end with a NULL byte. In C++, strings (std::string) are objects with all the associated automated memory management and control which makes them a lot safer and easier to use, especially for the novice.

20.**What is dynamic initialization of objects? (NOV/DEC 2015)**

Dynamic initialization is that in which initialization value isn't known at compile-time. It's computed at runtime to initialize the variable.

PART-B

1. **Write brief notes on Friend function and show how to modify a Class's private Data with a Friend Function. (16) (MAY/JUNE 2011) (NOV/DEC 2013)**

Friend Function:

- Friend function is a function that is not a member of a class but has access to the class's private and protected members.
- Friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

Example

Program:

```
class
Box
{
double width;
public:
double
length;
friend void printWidth( Box box );
void setWidth( double wid );
};
#include <iostream>
using namespace
std; class Box
{
double
width;
```

```
public:
friend void printWidth( Box box );
void setWidth( double wid
); };

// Member function definition
void Box::setWidth( double wid
)
{
width = wid;
}
// Note: printWidth() is not a member function of any class.
void printWidth( Box box )
{
/* Because setWidth() is a friend of Box, it can
directly access any member of this class */
cout << "Width of box : " << box.width <<endl;
}
// Main function for the
program int main( )
{
Box box;
// set box width without member
function box.setWidth(10.0);
// Use friend function to print the
wdith. printWidth( box );
return 0;
}
```

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

```
Width of box : 10
```

2. Explain the different types of constructors with suitable examples. (16) (NOV/DEC 2013)

PARAMETERIZED CONSTRUCTORS

The constructor that can take arguments are called parameterized constructor. The arguments can be separated by commas and they can be specified within braces similar to the argument list in function.

When a constructor has been parameterized, the object declaration without parameter may not work. In case of parameterized constructor, we must provide the appropriate arguments to the constructor when an object is declared. This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The implicit call method is sometimes known as shorthand method as it is shorter and is easy to implement.

Program : creating parameterized constructor

```
#include<iostream.h>
#include<conio.h
> class student
{
private:

int roll,age,
marks; public:
student(int r, int m, int a); //parameterized constructor void display( )

{
cout<<“\nRoll number :” <<roll <<endl;
cout<<“Total marks : <<marks<<endl;
cout<<“Age:”<<age<<endl;
```

```
}
};                                              //end of class declaration
student : : student(int r, int m, int a)        //constructor definition
{
roll = r;
marks =m;
age=a;
}
int main( )
{
Student manoj(5,430,16);                        //object creation
Cout<<"\n Data of student 1:"<<endl;
manoj.display();
Student ram(6,380,15);                          //object creation
Cout<<"\n Data of student 1:"<<endl;
ram.display();
getch();
return 0;
}
```

```
Data of student 1:
Roll number     :   5
Total Marks     :   430
Age             :   16
Data of student 2:
Roll number     :   6
Total Marks     :   380
Age             :   15
```

COPY CONSTRUCTOR

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.

Syntax:

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

Program : Copy Constructor

```
#include<iostream.h>
#include<conio.h>
class student
{
private:
int roll, marks, age;
public:
student(int r,int m,int a)                      //parameterized constructor
{
rol l = r;
marks = m;
age = a;
}
```

```
student(student &s)                    //copy constructor
{
roll = s.roll;

marks = s.marks;
age=s.age;
}
void display( )
{
cout<<"Roll number :" <<roll <<endl;
cout<<"Total marks :"<<marks<<endl;
cout<<"Age:"<<age<<endl;
}
};
int main( )
{
clrscr();
student t(3,350,17);                    // or student k(t);
student k = t;                          // invokes copy constructor
cout<<"\nData of student t:"<<endl;
t.display();
cout<<"\n\nData of student k:"<<endl;
k.display();
getch();
return 0;
}
```

The outout of the above program will be like this:

```
Data of student t   :
Roll number         :       3
Total marks         :       350
Age                 :       17
Data of student k   :
Roll number         :       3
Total marks         :       350
Age                 :       17
```

DEFAULT CONSTRUCTOR

n C++, the standard describes the default constructor for a class as a constructor that can be called with no arguments (this includes a constructor whose parameters all have default arguments). For example:

```
class MyClass
{
public:
MyClass();  // constructor declared

private:
int
x;};
MyClass :: MyClass()  // constructor defined
{
x = 100;
}
int  main()
{
MyClass m;  // at runtime, object m is created, and the default constructor is called
```

}

DYNAMIC CONSTRUCTORS

The constructor 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. 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.

Program: illustrate the working of dynamic constructors:

```
#include<iostream.h>
 class  Sample
    {
          char *name;
          int length;
      public:
           Sample( )
             {
                length = 0;
                name = new char[ length + 1];
             }
          Sample ( char  *s )
             {
                length = strlen(s);
                name = new char[ length + 1 ];
                strcpy( name , s );
             }
          void display( )
            {
                cout<<name<<endl;
             }
         };
        int  main( )
          {
                char  *first = " C++ ";
                Sample S1(first), S2("ABC" ),
                S3("XYZ"); S1.display( );
                S2.display( );
                S3.display( );
                return 0;
          }
```

RESULT :

```
C++
ABC
XYZ
```

3. i) Can we have more than one constructor in a class? Explain the need for it (8) (MAY/JUNE 2011)

Constructors Overloading are used to increase the flexibility of a class by having more number of constructor for a single class. By have more than one way of initializing objects can be done using overloading constructors

```
Overclass A;
Overclass A1(4);
Overclass A2(8, 12);
cout << "Overclass A's x,y value:: " <<
A.x << " , "<< A.y << "\n";
cout << "Overclass A1's x,y value:: "<<
A1.x << " ,"<< A1.y << "\n";
cout << "Overclass A2's x,y value:; "<<
```

```
A2.x << " , "<< A2.y << "\n";
return 0;
}
```

Result:

```
  Overclass A's x,y value:: 0 , 0
Overclass A1's x,y value:: 4 ,4
Overclass A2's x,y value:; 8 , 12
```

In the above example the constructor "Overclass" is overloaded thrice with different intialized values.

ii) What do you meant by parameterized constructor and explicit constructor? Explain with suitable example. (8) (MAY/JUNE 2011)

Parameterized constructor:-

A constructor can also take arguments. A constructor having some argument in it is called parameterized constructor. They allow us to initialize the various data element of different objects with different values.
We have to pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

- By calling the constructor explicitly
- By calling the constructor implicitly

Example

```
class integer
{
int m ,n;
public:
integer (int x , int y);
};
integer :: integer (int x , int y )
{
m=x; n = y;
}
integer int1 = integer( 0 , 100); // Explicit Call
integer int1(1,100); // implicit call
```

4. i) What is virtual function? When do we make a virtual function "pure"? (8) (MAY/JUNE 2011)

Virtual Function

• The compiler selects the appropriate function for a particular call at the run time only. It can be achieved using ***virtual functions***

Pure Virtual Function:

A virtual function body is known as Pure Virtual Function. In above example we can see that the function is base class never gets invoked. In such type of situations we can use pure virtual functions

Example

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


class derived1 : public base
{
public: void
show()
{
cout<<"\n Derived 1";
}
};
class derived2 : public base
```

```
{
public: void
show()
{
cout<<"\n Derived 2";
}
};
void main()
{
base *b;
derived1 d1;
derived2 d2;
b = &d1; b-
>show(); b
= &d2; b-
>show();
}
```

ii) Explain the concept of polymorphism with suitable example. (8)

Polymorphism

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

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

class Shape {
protected:
int width, height;
public:
Shape( int a=0, int b=0)
{
width = a;
height = b;
}
int area()
{
cout << "Parent class area :" <<endl;
return 0;
}
};
class Rectangle: public Shape{
public:
Rectangle( int a=0, int b=0)
{
Shape(a, b);
}
int area ()
{
cout << "Rectangle class area :" <<endl;
return (width * height);
}
};
```

```
class Triangle: public Shape{
public:
Triangle( int a=0, int b=0)
{
Shape(a, b);
}
int area ()
{
cout << "Rectangle class area :" <<endl;
return (width * height / 2);
}
};
// Main function for the program
int main( )
{
Shape *shape;
Rectangle rec(10,7);
Triangle tri(10,5);

// store the address of Rectangle
shape = &rec;
// call rectangle area.
shape->area();

// store the address of Triangle
shape = &tri;
// call triangle area.
shape->area();

return 0;
}
```

5.i) Write a program to perform string copy operation using dynamic constructor (6) (NOV/DEC 2014)

Dynamic Constructors

The constructor 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. 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.

Program

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class string
{
char *name;
int length;
public:
string()
{
length=0;
name=new char[length+1];
}
string(char*s)
{
length=strlen(s);
name=new char[length+1];
```

```
strcpy(name,s);
}
void display(void)
{
cout<<name<<"\n";
}
void join(string &a,string &b);
};
void string::join(string &a,string &b)
{
length=a.length+b.length;
delete name;
name=new char[length+1];
strcpy(name,a.name);
strcat(name,b.name);
};
void main()
{
clrscr();
string name1 ("Programming");
string name2 ("and");
string name3 ("Data Structures");
string s1,s2;
s1.join(name1,name2);
s2.join(s1,name3);
name1.display();
name2.display();
name3.display();
s1.display();
s2.display();
getch();
}
```

OUTPUT:

```
Programming
and
Data Structures
Programming and
Programming and Data Structures
```

ii) Consider the following arithmetic expressions: C = 2+B and K=S-T, where B, C, K, S, and T are the objects of a class called '1DArray'. Write a program to per form these operations by overloading the '+' and '-' operators respectively. (10) (NOV/DEC 2014)

```
#include<iostream.h>
#include<math.h>
#include<conio.h>
class 1Darray
{
int x;
public:
1Darray()
{
}
1Darray(int z)
{
x=z;
```

```
}
friend 1Darray operator+(int , 1Darray);
friend 1Darray operator-(1Darray , 1Darray);
friend void show(1Darray);
};
1Darray operator+ (int c1, 1Darray c2)
{
1Darray c3;
c3.x=c1+c2.x;
return c3;
}
1Darray operator- (1Darray c1, 1Darray c2)
{
1Darray c3;
c3.x=c1.x+c2.x;
return c3;
}
void show(1Darray c)
{
cout<<c.x<< c.y<<endl;
}
void main()
{
1Darray B;
B=1Darray(5);
1Darray C;
C=2+B;
cout<<"1Darray No:1 = ";
show(B);
cout<<"1Darray No:2 = ";
show(C);
1Darray S(7);
1Darray T(4);
1Darray K;
K=S-T;
cout<<"1Darray No:3 = ";
show(S);
cout<<"1Darray No:4 = ";
show(T);
cout<<"1Darray No:5 = ";
show(K);
getch();
}
```

6.i)Write a program to perform dynamic initialization of objects (6) (NOV/DEC 2014)

We can dynamically initialize objects through constructors. Object's data members can be initialized dynamically at run time even after their creation.

Program for dynamic initialization of object

```
#include<iostream.h>
#include<conio.h>
#include<math.h>
class number
{
public:
int num;
```

```
number(int n)
{
num=n;
}
int sum( )
{
num=num+5;
return(num);
}
};
int main( )
{
number obj1(1);                    // parameterized constructor invoked
number obj2(2);
clrscr();
cout<<"\nValue of object 1 and object 2 are : ";
cout<<obj1.num<<"\t"<<obj2.num;
number obj3(obj1.sum( ));          //dynamic initialization of object
cout<<"\nValue of object 1 after calling sum() : "<<obj1.num;
cout<<"\nValue of object 3 is :"<<obj3.num;
getch();
return 0;
}
```

Output:

```
Value of object 1 and object 2 are : 1 2
Value of object 1 after calling sum() : 6
Value of object 3 is : 6
```

ii)Write a program to illustrate the process of multi-level multiple inheritance concept of C++ language. (10) (NOV/DEC 2014)

"Hybrid Inheritance" is a method where one or more types of inheritance are combined together and used.

```
#include <iostream.h>
class mm
{
protected:
int rollno;
public:
void get_num(int a)
{ rollno = a; }
void put_num()
{ cout << "Roll Number Is:"<< rollno << "\n"; }
};
class marks : public mm
{
protected:
int sub1;
int sub2;
public:
void get_marks(int x,int y)
{
sub1 = x;
sub2 = y;
}
void put_marks(void)
```

```
{
cout << "Subject 1:" << sub1
<< "\n"; cout << "Subject 2:"
<< sub2 << "\n";
}
};
class extra
{
protected:
float e;
public:
void
get_extra(flo
at s) {e=s;}
void put_extra(void)
{ cout << "Extra Score::" << e
<< "\n";} };
class res : public marks,
public extra{ protected:
f
l
o
a
t

t
o
t
;

p
u
b
l
i
c
:
void disp(void)
{
tot =
sub1+sub
2+e;
put_num(
);
put_mark
s();
put_extra(
);
cout << "Total:"<< tot;
}
```

```
};
int main()
{
res std1;
std1.get_num
(10);
std1.get_mark
s(10,20);
std1.get_extra
(33.12);
std1.disp();
return 0;
}
```

Output:

```
Roll
Number
Is: 10
Subject 1:
10
Subject 2:
20 Extra
score:33.1
2 Total:
63.12
```

7.**Explain how memory is dynamically allocated and recovered in c++. Explain with example program.(8)(APRIL/MAY 2015)**

Dynamic memory

In the programs seen in previous chapters, all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators new and delete.

Operators new and new[]

Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its syntax is:

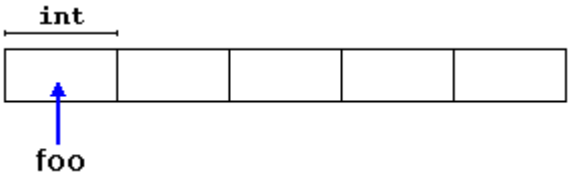
```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type type. The second one is used to allocate a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

```
1 int * foo;

2 foo = newint [5];
```

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to foo (a pointer). Therefore, foo now points to a valid

block of memory with space for five elements of type int.



Here, foo is a pointer, and thus, the first element pointed to by foo can be accessed either with the expression foo[0] or the expression *foo (both are equivalent). The second element can be accessed either with foo[1] or *(foo+1), and so on...

There is a substantial difference between declaring a normal array and allocating dynamic memory for a block of memory using new. The most important difference is that the size of a regular array needs to be a constant expression, and thus its size has to be determined at the moment of designing the program, before it is run, whereas the dynamic memory allocation performed by new allows to assign memory during runtime using any variable value as size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, there are no guarantees that all requests to allocate memory using operator new are going to be granted by the system.

C++ provides two standard mechanisms to check if the allocation was successful:

One is by handling exceptions. Using this method, an exception of type bad_alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now, you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the method used by default by new, and is the one used in a declaration like:

```
foo = newint [5];  // if allocation fails, an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution normally.

This method can be specified by using a special object called nothrow, declared in header <new>, as argument for new:

```
foo = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory fails, the failure can be detected by checking if foo is a null pointer:

```
1  int * foo;
2  foo = new (nothrow) int [5];
```

```
3  if (foo == nullptr) {
4  // error assigning memory. Take measures.
5  }
```

This nothrow method is likely to produce less efficient code than exceptions, since it implies explicitly checking the pointer value returned after each and every allocation. Therefore, the exception mechanism is generally preferred, at least for critical allocations. Still, most of the coming examples will use the nothrow mechanism due to its simplicity.

Operators delete and delete[]

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete, whose syntax is:

```
1  delete pointer;
2  delete[] pointer;
```

The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).

The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

```
1   // rememb-o-matic
2   #include <iostream>
3   #include <new>
4   usingnamespace std;
5
6   int main ()
7   {
8   int i,n;
9   int * p;
10   cout << "How many numbers would you like to type? ";
11   cin >> i;
12   p= new (nothrow) int[i];
13  if (p == nullptr)
14     cout << "Error: memory could not be allocated";
15  else
16    {
17  for (n=0; n<i; n++)
18     {
19       cout << "Enter number: ";
20       cin >> p[n];
21     }
22     cout << "You have entered: ";
23  for (n=0; n<i; n++)
24       cout << p[n] << ", ";
25  delete[] p;
```

```
How many numbers would you like to type
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

```
26    }
27   return 0;
28   }
```

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant expression:

```
p= new (nothrow) int[i];
```

There always exists the possibility that the user introduces a value for i so big that the system cannot allocate enough memory for it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program, and I got the text message we prepared for this case (Error: memory could not be allocated).

It is considered good practice for programs to always be able to handle failures to allocate memory, either by checking the pointer value (if nothrow) or by catching the proper exception.

ii)what are the operators that cannot be overloaded?write a program to overload any one of the binary operators? (8) (APRIL/MAY 2015)

In C++, following operators can not 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)

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

class complex
{
        int a,b;
   public:
        void getvalue()
        {
          cout<<"Enter the value of Complex Numbers a,b:";
          cin>>a>>b;
        }
        complex operator+(complex ob)
        {
                complex t;
                t.a=a+ob.a;
                t.b=b+ob.b;
                return(t);
        }
```

```
        complex operator-(complex ob)
        {
                complex t;
                t.a=a-ob.a;
                t.b=b-ob.b;
                return(t);
        }
        void display()
        {
                cout<<a<<"+"<<b<<"i"<<"\n";
        }
};

void main()
{
  clrscr();
  complex obj1,obj2,result,result1;

  obj1.getvalue();
  obj2.getvalue();

  result = obj1+obj2;
  result1=obj1-obj2;

  cout<<"Input Values:\n";
  obj1.display();
  obj2.display();

  cout<<"Result:";
  result.display();
 result1.display();

  getch();
}
```

Output:

```
Enter the value of Complex Numbers a, b
4               5
Enter the value of Complex Numbers a, b
2               2
Input Values
4 + 5i
2 + 2i
Result
6 +   7i
2 +   3i
```

8.i)Define a class Shape with constructor, destructor and pure virtual functions GetArea(),GetPerim() and Draw(). Derive classes Circle and Rectangle from Shape. Derive another class Square from Rectangle. Implement this hierarchy with essential functions and write a main. (10) (APRIL/MAY 2015)

```
//Implementing pure virtual functions
2:
3:    #include <iostream.h>
4:
5:    enum BOOL { FALSE, TRUE };
7:    class Shape
8:    {
9:    public:
10:      Shape(){}
11:      ~Shape(){}
12:      virtual long GetArea() = 0; // error
13:      virtual long GetPerim()= 0;
14:      virtual void Draw() = 0;
15:   private:
16:   };
17:
18:    void Shape::Draw()
19:   {
20:      cout << "Abstract drawing mechanism!\n";
21:   }
22:
23:   class Circle : public Shape
24:   {
25:   public:
26:         Circle(int radius):itsRadius(radius){}
27:         ~Circle(){}
28:         long GetArea() { return 3 * itsRadius * itsRadius; }
29:         long GetPerim() { return 9 * itsRadius; }
30:         void Draw();
31:   private:
32:      int itsRadius;
33:      int itsCircumference;
34:   };
35:
36:   void Circle::Draw()
37:   {
38:      cout << "Circle drawing routine here!\n";
39:      Shape::Draw();
40:   }
41:
42:
43:   class Rectangle : public Shape
44:   {
45:   public:
46:         Rectangle(int len, int width):
```

```
47:          itsLength(len), itsWidth(width){}
48:       ~Rectangle(){}
49:       long GetArea() { return itsLength * itsWidth; }
50:       long GetPerim() {return 2*itsLength + 2*itsWidth; }
51:       virtual int GetLength() { return itsLength; }
52:       virtual int GetWidth() { return itsWidth; }
53:       void Draw();
54:    private:
55:      int itsWidth;
56:      int itsLength;
57:    };
58:
59:    void Rectangle::Draw()
60:    {
61:      for (int i = 0; i<itsLength; i++)
62:      {
63:        for (int j = 0; j<itsWidth; j++)
64:          cout << "x ";
65:
66:        cout << "\n";
67:      }
68:      Shape::Draw();
69:    }
70:
71:
72:    class Square : public Rectangle
73:    {
74:    public:
75:       Square(int len);
76:       Square(int len, int width);
77:       ~Square(){}
78:       long GetPerim() {return 4 * GetLength();}
79:    };
80:
81:    Square::Square(int len):
82:      Rectangle(len,len)
83:    {}
84:
85:    Square::Square(int len, int width):
86:      Rectangle(len,width)
87:
88:    {
89:      if (GetLength() != GetWidth())
90:        cout << "Error, not a square... a Rectangle??\n";
91:    }
92:
93:    int main()
94:    {
95:      int choice;
96:      BOOL fQuit = FALSE;
97:      Shape * sp;
```

```
98:
99:       while (1)
100:      {
101:         cout << "(1)Circle (2)Rectangle (3)Square (0)Quit: ";
102:         cin >> choice;
103:
104:         switch (choice)
105:         {
106:            case 1: sp = new Circle(5);
107:            break;
108:            case 2: sp = new Rectangle(4,6);
109:            break;
110:            case 3: sp = new Square (5);
111:            break;
112:            default: fQuit = TRUE;
113:            break;
114:         }
115:         if (fQuit)
116:            break;
117:
118:         sp->Draw();
119:         cout << "\n";
120:      }
121:      return 0;
122: }
```

Output: (1)Circle (2)Rectangle (3)Square (0)Quit: 2
x x x x x x
x x x x x x
x x x x x x
x x x x x x
Abstract drawing mechanism!

(1)Circle (2)Rectangle (3)Square (0)Quit: 3
x x x x x
x x x x x
x x x x x
x x x x x
x x x x x
Abstract drawing mechanism!

(1)Circle (2)Rectangle (3)Square (0)Quit: 0

8.ii)Define a class Area to identify the area of square and rectangle using constructor and destructor. Use the parameters length(l) and breadth(b) for the constructor functions.(6) (APRIL/MAY 2015)

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

```
{
   private:
     int length;
     int breadth;

   public:
     Area(): length(5), breadth(2){ }   /* Constructor */
     void GetLength()
     {
        cout<<"Enter length and breadth respectively: ";
        cin>>length>>breadth;
     }
     int AreaCalculation() {  return (length*breadth);  }
     void DisplayArea(int temp)
     {
        cout<<"Area: "<<temp;
     }
};
int main()
{
   Area A1,A2;
   int temp;
   A1.GetLength();
   temp=A1.AreaCalculation();
   A1.DisplayArea(temp);
   cout<<endl<<"Default Area when value is not taken from user"<<endl;
   temp=A2.AreaCalculation();
   A2.DisplayArea(temp);
   return 0;}
```

9.i) Write a C++ program to overload the increment operator in prefix and postfix forms (12) (NOV/DEC 2015)

Operator overloading of prefix operator:

```
using namespace std;
class Check
{
  private:
    int i;
  public:
    Check(): i(0) {  }
    Check operator ++() /* Notice, return type Check*/
    {
      Check temp;  /* Temporary object check created */
      ++i;         /* i increased by 1. */
      temp.i=i;   /* i of object temp is given same value as i */
      return temp; /* Returning object temp */
```

```
    }
   void Display()
      { cout<<"i="<<i<<endl; }
};
int main()
{
    Check obj, obj1;
    obj.Display();
    obj1.Display();
    obj1=++obj;
    obj.Display();
    obj1.Display();
    return 0;
}
```

Output

```
i=0
i=0
i=1
i=1
```

Operator Overloading of Postfix Operator

```
#include <iostream>
using namespace std;
class Check
{
  private:
    int i;
  public:
    Check(): i(0) {  }
    Check operator ++ ()
    {
        Check temp;
        temp.i=++i;
        return temp;
    }

/* Notice int inside barcket which indicates postfix increment. */
    Check operator ++ (int)
    {
        Check temp;
        temp.i=i++;
        return temp;
    }
   void Display()
      { cout<<"i="<<i<<endl; }
};
```

```
int main()
{
   Check obj, obj1;
   obj.Display();
   obj1.Display();
   obj1=++obj;     /* Operator function is called then only value of obj is assigned to obj1. */
   obj.Display();
   obj1.Display();
   obj1=obj++;     /* Assigns value of obj to obj1++ then only operator function is called. */
   obj.Display();
   obj1.Display();
   return 0;
}
```

Output

```
i=0
i=0
i=1
i=1
i=2
i=1
```

ii) Distinguish the term overloading and overriding (4) (NOV/DEC 2015) (APRIL/MAY 2016)

Overloading vs. overriding.

- Overriding of functions occurs when one class is inherited from another class. Overloading can occur without inheritance.
- Overloaded functions must differ in function signature ie either number of parameters or type of parameters should differ. In overriding, function signatures must be same.
- Overridden functions are in different scopes; whereas overloaded functions are in same scope.
- Overriding is needed when derived class function has to do some added or different job than the base class function.
- Overloading is used to have same name functions which behave differently depending upon parameters passed to them.

10.i)Write a C++ program to explain how the runtime polymorphism is achieved (8) (NOV/DEC 2015) (APRIL/MAY 2016)

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

class Base
{
public:
   virtual void show() { cout<<" In Base \n"; }
};

class Derived: public Base
{
```

```
public:
   void show() { cout<<"In Derived \n"; }
};

int main(void)
{
   Base *bp = new Derived;
   bp->show();  // RUN-TIME POLYMORPHISM
   return 0;
}
```

ii) Illustrate any four types of inheritance supported in C++ with suitable example (8) (NOV/DEC 2015) (APRIL/MAY 2016)

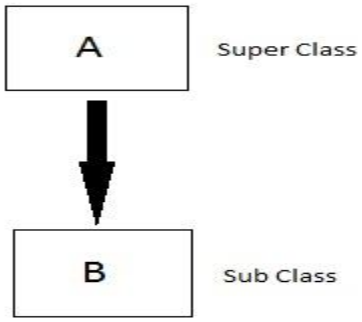
Types of Inheritance

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

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

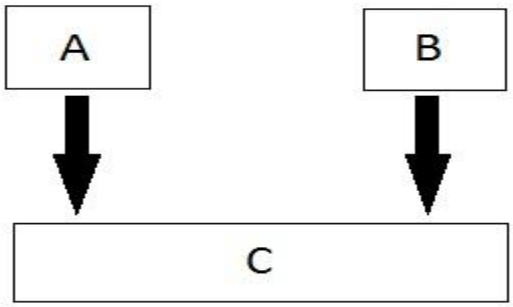
Single Inheritance

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



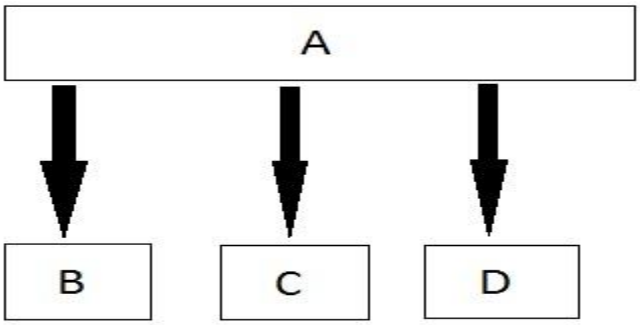
Multiple Inheritance

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



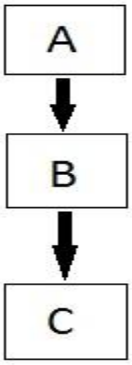
Hierarchical Inheritance

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



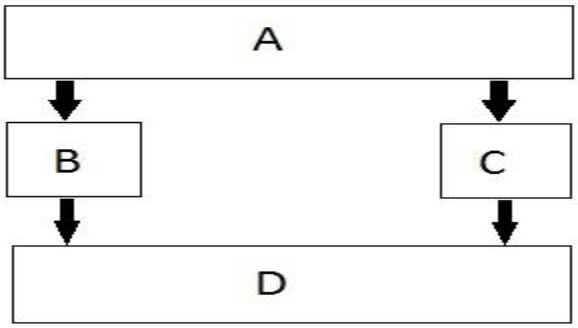
Multilevel Inheritance

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



Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



UNIT III C++ PROGRAMMING ADVANCED FEATURES

PART-A

1. What are the file stream classes used for creating input and output files?

- ofstream: Stream class to write on files
- ifstream: Stream class to read from files
- fstream: Stream class to both read and write from /to files.

2. List out any four containers supported by Standard Template Library.

- vector
- list
- set

- map

3. What are templates? (NOV/DEC 2012)

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one. This is effectively a Turing-complete language. Templates are of great utility to programmers in C++, especially when combined with multiple inheritance and operator overloading. The C++ Standard Library provides many useful functions within a framework of connected templates

4. What is file mode? List any four file modes.

A file mode describes how a file is to be used, to read, to write to append etc. When you associate a stream with a file, either by initializing a file stream object with a file name or by using open() method, you can provide a second argument specifying the file mode. e.g. *stream_object.open("filename",filemode);*

List of filemodes available in C++

- ios::in
- ios::out
- ios::binary
- ios::ate

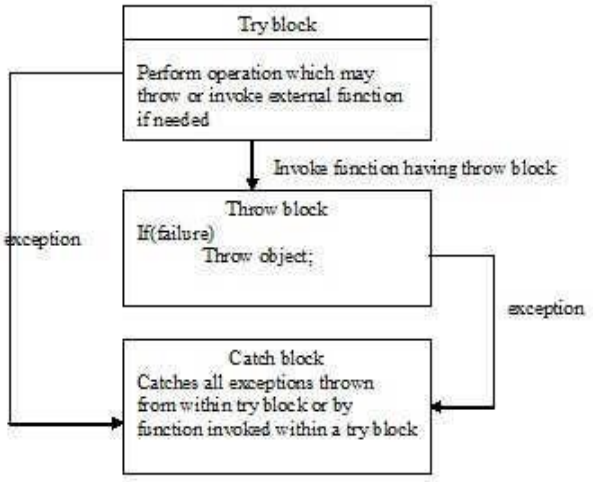
5. What is throw ()? What is its use? (NOV/DEC 2012)

In C++, a throw statement is used to signal that an exception or error case has occurred. Signaling that an exception has occurred is also commonly called raising an exception. To use a throw statement, simply use the throw keyword, followed by a value of any data type you wish to use to signal that an error has occurred. Typically, this value will be an error code, a description of the problem, or a custom exception class

6.What is meant by abstract class? (NOV/DEC 2012)

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class. The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

7. Illustrate the exception handling mechanism (NOV/DEC 2013)



8. What is namespace? (NOV/DEC 2012)

A **namespace** is a container for a set of identifiers (also known as symbols, names). Namespaces provide a level of indirection to specific identifiers, thus making it possible to distinguish between identifiers with the same exact name. For example, a surname could be thought of as a namespace that makes it possible to distinguish people who have the same first name. In computer programming, namespaces are typically employed for the purpose of grouping symbols and identifiers around a particular functionality.

9. What is stream? Why they are useful? (NOV/DEC 2012)

A stream means the flow of data. There are two types of flow namely Input flow and output flow. Two common kinds of input flows are from input devices (keyboard, mouse) or Files means reading or receiving of data from a source. First case source is an input device, in second case source is a File. Similarly for output stream uses output devices and Files.

10. Distinguish between class template and function template (MAY/JUNE 2011)

A function template specifies how an individual function can be constructed. The limitation of such functions is that they operate only on a particular data type. It can be overcome by defining that function as a function template or generic function.

Classes can also be declared to operate on different data types. Such classes are called class templates. A class template specifies how individual classes can be constructed similar to normal class specification

11. What is an exception? (MAY/JUNE 2011)

Exceptions which occur during the program execution, due to some fault in the input data.

12. What are the two types of exceptions?

Exceptions are classifieds into

a)*Synchronous exception*

The technique that is not suitable to handle the current class of data, within the program are known as synchronous exception

b) *Asynchronous exception*

The exceptions caused by events or faults unrelated to the program and beyond the control of program are called asynchronous exceptions

13. What are the blocks used in the Exception Handling?

The exception-handling mechanism uses three blocks 1)try block 2)thro w block 3)catch block

The **try-block** must be followed immediately by a handler, which is a **catch-block.** If an exception is thrown in the **try-block**

14. Write the syntax of try construct

The try keyword defines a boundary within which an exception can occur. A block of code in which an exception can occur must be prefixed by the keyword try. Following the try keyword is a block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions

```
try
{
//code raising exception or referring to a function raising exception
}
catch(type_id1)
{
//actions for handling an exception
}
catch(type_idn)
{
```

```
//actions for handling an exception
}
```

15. Write the syntax of catch construct

The exception handler is indicated by the catch keyword. It must be used immediately after the statements marked by the try keyword. The catch handler can also occur immediately after another catch. Each handler will only evaluate an exception that matches, or can be covered to the type specified in its argument list.

```
catch(T)
{
//        actions for handling an exception
}
```

16.What is the significance of Iterators? (NOV/DEC 2014)

- An Iterator is an object that enables to traverse across the container
- To expose elements in a range
- Saves time
- When using remove method inside for loop to traverse in an array, out of bound exception occurs since size of array changes when removing element. It can be prevented by using iterator.

17.Give an example of Function template (NOV/DEC 2014)

```
#include
<iostream>
#include
<string> using
namespace std;
template
<typename T>
inline T const& Max (T const& a, T const& b)
{
   return a < b ? b:a;
}
int main ()
{
   in
   t i
   =
   3
   9;
   in
   t j
   =
   2
   0;
   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;}
```

18.Write the syntax of throw construct

The keyword throw is used to raise an exception when an error is generated in the computation. The throw expression initializes a temporary object of the type T(to match the type of argument arg)used in throw(T arg)

throw T;

19. Write the syntax of function template

```
template<class T,…>
ReturnType FuncName(arguments)
{
        …..//body of the function template
}
```

20.Compare Overloaded functions versus function template. (NOV/DEC 2015)

In C++, two different functions can have the same name if their parameters are different; either because they have a different number of parameters, or because any of their parameters are of a different type. template <template-parameters> function-declaration. The template parameters are a series of parameters separated by commas.

21.When do we use multiple catch handlers? (NOV/DEC 2015)

Multiple handlers (i.e., catch expressions) can be chained; each one with a different parameter type. Only the handler whose argument type matches the type of the exception specified in the throw statement is executed.

22.**Distinguish template class and class template (APRIL/MAY 2016)**

Template class : A class that has generic definition or a class with parameters which is not instantiated until the information is provided by the client. It is referred to a jargon for plain templates.

Class template : The individual construction of a class is specified by a class template which is almost similar the way how individual objects are constructed by using a class. It is referred to a jargon for plain classes.

PART-B

1.i)Write the syntax for member function template. (4) (NOV/DEC 2013)

The syntax of member function templates is as follows:

```
template<class T>
returntypeclassname<T> || functionname(arglist)
{
        //……
        …….
        //Function
        body
        //……
        …….
}
```

ii) Write a C++ program using class template for finding the scalar product for *int* type vector and float type vector. (12)

A template which is used to create a family of classes with different data types known as class templates. Syntax :

```
template<class T>
        class classname
        {
                private:
                        _____
                        _____
                public:
```

};

Program :

```
#include<iostr
eam.h>
template<class
T> class
sample
{
private:
T
value,value1,va
lue2; Public:
Void
getdata(
);   Void
product(
); };

template<class T>
void sample<T>::getdata()
{
cin>>value1>>value2;
}
template<class T>
void sample<T>::product()
{
T value;
Value=value1*
value2;
Cout<<value;
}
void main()
{
sample<int>obj1;
sample<float>obj2;
obj1.getdata();
obj1.product();
obj2.getdata();
obj2.product();
}
```

2. i) Explain how re-throwing of an exception is done. (4) (NOV/DEC 2014)

Rethrowing an expression from within an exception handler can be done by calling throw, by itself, with no exception. This causes current exception to be passed on to an outer try/catch sequence. An exception can only be rethrown from within a catch block. When an exception is rethrown, it is propagated outward to the next catch block.

Consider following code:

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

```
void MyHandler()
{
    try
    {
        throw “hello”;
    }
    catch (const char*)
    {
        cout<<”Caught exception inside MyHan
        dler\n”; throw; //rethrow char* out of
        function
    }
}
int main()
{
    cout<< “Main
    start”; try
    {
        MyHandler();
    }
    catch(const char*)
    {
        cout<<”Caught exception inside Ma in\n”;
    }
        cout<<
    “Main end”;
    return 0;
}
```

O/p:
Mai
n
start
Caught exception inside
MyHandler Caught exception
inside Main Main end
Thus, exception rethrown by the catch block inside MyHandler() is caught inside main();

ii) Write a C++ program that illustrates multiple catch statements. (12)

A program will **throw** different types of errors. For each error, different **catch** blocks have to be defined. Syntax :

```
try {
        Code to Try
}
catch(Arg1)
{
        One Exception
}
catch(Arg2)
{
        Another Exception
}
#include<iostream.h>
```

```
#include<coni
o.h> void
test(int x)
{
  try
  {
        if(x>
          0)
          thro
          w
          x;
    else
          throw 'x';
  }
  catch(int x)
  {
        cout<<"Catch a integer and that integer is:"<<x;
  }
  catch(char x)
  {
        cout<<"Catch a character and that character is:"<<x;
  }
}
void main()
{
  clrscr();
  cout<<"Testing multiple
  catches\n:"; test(10);
  test(0);
  getch();
}
```

3. i) Explain the overloading of template function with suitable example. (8)

A template function overloads itself as needed. But we can explicitly overload it too. Overloading a function template means having different sets of function templates which differ in their parameter list.

Consider following example:

```
#include <iostream>
template<class X> void
func(X a)
{
    // Function code;
   cout<<"Inside f(X a)
   \n";
}
template<class X, class Y> void func(X a, Y b) //overloading function template func()
{
    // Function code;
   cout<<"Inside f(X a, Y b)
   \n";
}
int main()
{
```

```
    func(10); // calls func(X a)
    func(10, 20); // calls func(X a,
    Y b)
    return 0;
}
```

ii) Write a function template to find the minimum value contained in an array. (8) (NOV/DEC 2014)

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

template<class T>
T findMin(T arr[],int n)
{
int i;
 T min;
min=arr[0];
for(i=0;i<n;
i++)
   {
        if(min
        >arr[i])
        min=arr[i
        ];
   }
return(min);
}
void main()
{
clrscr();
intiarr[5];
charcarr[5];
doubledarr[5];
cout<<"Integer Values
\n"; for(int i=0; i < 5;
i++)
   {
cout<<"Enter integer value "<< i+1
<<" : "; cin>>iarr[i];
   }
cout<<"Character values
\n"; for(int j=0; j < 5;
j++)
   {
cout<<"Enter character value "<< j+1 <<"
: "; cin>>carr[j];
   }
cout<<"Decimal values
\n"; for(int k=0; k < 5;
k++)
```

```
    {
  cout<<"Enter decimal value "<< k+1 <<"
  : "; cin>>darr[k];
    }
  //calling Generic function...to find minimum value.
  cout<<"Generic Function to find Minimum from
  Array\n\n"; cout<<"Integer Minimum is :
  "<<findMin(iarr,5)<<"\n"; cout<<"Character Minimum
  is : "<<findMin(carr,5)<<"\n"; cout<<"Double Minimum
  is : "<<findMin(darr,5)<<"\n"; getch();
  }
```

4.i) Write a program to read a string to store it in a file and read the same string from the file to display it on the output device (8) (NOV/DEC 2014)

```
#include<st
dio>
int main()
{
int c;
printf("Enter the
value");
c=getchar();
printf("you have
entered");
putchar(c);
return 0;
}
```

ii) What is STL? Brief on its key components and their types. (8) (NOV/DEC 2014)

STL has generic software components called container. These are classes contain other object.

Two container :

- Sequence container – Vector, List, Deque
- Associative sorted Container – set, Multiset, map andmultimap.
    - Without writing own routine for programusing array,list,queue, etc.,stl provides tested and debugged components readily available. It provide reusability of code.
    - Eg: queue is used in os program to store program for execution.
    - Advantage of readymade components :
- Small in number
- Generality
- Efficient, Tested, Debugged and standardized
- Portability and reusability

5.i)Write a program to implement stack operations using class template.(8)(APRIL/MAY 2015)

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
template<class Type>
class Stack
```

```
{
Type s[10];
int top,n;
public:
Stack()
{
top=-1;
cout<<"\n\tEnter the Stack Size : ";
cin>>n;
}
void push(Type elt)
{
if(top<n-1)
s[++top]=elt;
else
cout<<"\n\tstack is full.Can't insert "<<elt<<endl;
}
void pop()
{
if(top<0)
cout<<"\n\tstack is empty.\n";
else
cout<<"\n\tPoped elt : "<<s[top--];
}
void stk_operation();
};
template<class Type>
void Stack<Type> :: stk_operation()
{
int choice=1,i;
Type elt;
while(choice>0 && choice<3)
{
cout<<"\n\n\t1.PUSH\t2.POP\tAny Key To Exit\n\tChoice : ";
cin>>choice;
switch(choice)
{
case 1 : //push
cout<<"\n\tEnter the Elt to push : ";
cin>>elt;
push(elt);
cout<<"\n\t\tstack content :\n\n\t";
for(i=0;i<=top;i++)
cout<<s[i]<<"\t";
break;
case 2 : //pop
pop();
```

```
cout<<"\n\t\tstack content :\n\n\t";
for(i=0;i<=top;i++)
cout<<s[i]<<"\t";
break;
}
}
}
void main()
{
clrscr();
cout<<"\n\t\tSTACK OPERATION USING TEMPLATE\n\n";
cout<<"\n\t INT\n";
Stack<int> stk1;
cout<<"\n\t FLOAT\n";
Stack<float> stk2;
int ch;
while(1)
{
cout<<"\n\t\t\tSTACK OPERATION \n\n";
cout<<"\t1.INT STACK\t2.FLOAT STK\tAny Key To Exit\n\tChoice : ";
cin>>ch;
switch(ch)
{
case 1 : //perform stk operation on int stk
stk1.stk_operation();
break;
case 2 : //float
stk2.stk_operation();
break;
default : exit(0); }}}
```

ii)Write a template function to sort the elements of an array(4)

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
template <class S>
void bubble (S a[],int n)
{
  int i,j;
  for(i=0;i<n-1;i++)
    for(j=0;j<n-i-1;j++)
    if(a[j]>a[j+1])
    {
   swap(a[j],a[j+1]);
    }
}
template<class T>
```

```
void swap(T &x,T &y)
{
  T temp;
  temp=x;
  x=y;
  y=temp;
}
void main()
{
   int i,j,a[5]={25,2,56,4,34};
   float p[5]={25.25,3.45,4.2,20.5,5.8};
   clrscr();
   cout.precision(2);
   cout<<"\nOriginal Integer Array a:";
   for(i=0;i<5;i++)
   cout<<a[i]<<' ';
   cout<<"\nOriginal float array b:";
   for(i=0;i<5;i++)
   cout<<p[i]<<' ';
   bubble(a,5);
   bubble(p,5);
   cout<<"\n\nSorted integer array a:";
   for(i=0;i<5;i++)
   cout<<a[i]<<' ';
   cout<<"\nSorted float array b:";
   for(i=0;i<5;i++);
   cout<<p[i]<<' ';
   getch();
}
```

iii)What is an exception? Explain how the control is transferred and handled in an C++ programs(4)

An exception is a problem that arises during the execution of a program. 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.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

- 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.

6.i)Write a template function to sort the elements of an array.(6) (APRIL/MAY 2015)

```
#include <iostream>
#include <time.h>
#include <string>
```

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

template <class Item>
void bubble_sort(Item a[], size_t size);

template <class Item>
void swap(Item& a, Item& b);

template <class Item>
void display_array(Item randNum, size_t size);

int main()
{

        int *randNum;
        int a,b;
        int size;



        swap(Item& a, Item& b);  //NOT SURE IF WORKS YET, UNTESTED

        srand((unsigned)time(NULL));
        cout << "How many integers?" << endl;
        cin >> size;
        size = size +1;



        randNum = new int [size];

         for (int i = 1; i < size; i++)
         {
                randNum[i] = 1+ rand() % 10;
         }

        display_array(randNum, size);



        cout << "\n \n sorted..." << endl;



        bubble_sort( a[], size); //ERROR

        //display_array(randNum, size);

        system("pause");
```

```
        return 0;

}


//============================
template <class Item>
void bubble_sort(Item a[], size_t size)
{
        size_t idx, pass;
        for (pass=1; pass<=size; ++pass)
        {
                for (idx=0; idx<=size-2; ++idx)
                {
                        if (a[idx] > a[idx+1])
                                swap(a[idx], a[idx+1]);
                }
        }

}

template <class Item>
void swap(Item& a, Item& b)
{
    Item temp;
    temp = a;
    a = b;
    b = temp;
}


template <class Item>
void display_array(Item randNum, size_t size)
{

         for ( int j = 1; j < size; j++ )
  {

         cout  << setw(16) << randNum[ j ];

  }

}
```

ii)Write a program to write the text in a file. Read the text from the file, from end of the file. Display contents of file in reverse order. Append the contents to existing file. (10) (APRIL/MAY 2015)

Classes for Reading/Writing Files

In C++, three major files are used for interacting with files.

1.fstream

ifstream (Input Stream) is a class that is used to obtain input from any file. This class extracts the contents from the file. In simplest words, we can say that in order to read data from a file we use ifstream class.

2.ofstream

ofstream (Output Stream) class performs functions opposite to that of ifstream class. It is used to create a file and insert content into the file. In simpler words, this class is used to write data to a file.

3.fstream

This is a generic base class and can be used for both reading and writing data to the file.

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

int main()
{
   ofstream OFileObject;  // Create Object of Ofstream
   OFileObject.open ("D:\\ExampleFile.txt"); // Opening a File or creating if not present
   OFileObject << "I am writing to a file opened from program.\n"; // Writing data to file
   cout<<"Data has been written to file";
   OFileObject.close(); // Closing the file
}
```

7.i)Describe the components of STL(8)(NOV/DEC 2015) (APRIL/MAY 2016)

STL has generic software components called container. These are classes contain other object.

Two container :

- Sequence container – Vector, List, Deque
- Associative sorted Container – set, Multiset, map andmultimap.
    - Without writing own routine for programusing array,list,queue, etc.,stl provides tested and debugged components readily available. It provide reusability of code.
    - Eg: queue is used in os program to store program for execution.
    - Advantage of readymade components :
- Small in number
- Generality
- Efficient, Tested, Debugged and standardized
- Portability and reusability

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.

ii)Write a class template to represent a stack of any possible data type(8)(NOV/DEC 2015) (APRIL/MAY 2016)

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template <class T>
class Stack {
  private:
    vector<T> elems;     // elements

  public:
    void push(T const&);  // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const{       // return true if empty.
        return elems.empty();
    }
};

template <class T>
void Stack<T>::push (T const& elem)
{
    // append copy of passed element
    elems.push_back(elem);
}

template <class T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::pop(): empty stack");
    }
                // remove last element
```

```
    elems.pop_back();
}

template <class T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw out_of_range("Stack<>::top(): empty stack");
    }
                // return copy of last element
    return elems.back();
}

int main()
{
    try {
        Stack<int>         intStack;  // stack of ints
        Stack<string> stringStack;    // stack of strings

        // manipulate int stack
intStack.push(7);
        cout << intStack.top() <<endl;

        // manipulate string stack
        stringStack.push("hello");
        cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (exception const& ex) {
        cerr << "Exception: " << ex.what() <<endl;
        return -1;
    }
}
```

8.i)Write a C++ program to handle a divide by zero exception (8) (NOV/DEC 2015)

```
#include<iostream.h>
#include<conio.h>
void main()
{
  int a,b,c;
  float  d;
  clrscr();
  cout<<"Enter the value of a:";
  cin>>a;
  cout<<"Enter the value of b:";
  cin>>b;
  cout<<"Enter the value of c:";
  cin>>c;
```

```
  try
  {
          if((a-b)!=0)
          {
            d=c/(a-b);
            cout<<"Result is:"<<d;
          }
          else
          {
            throw(a-b);
          }
  }

  catch(int i)
  {
          cout<<"Answer is infinite because a-b is:"<<i;
  }

  getch();
}
```

Output:

```
          Enter the value for a: 20
          Enter the value for b: 20
          Enter the value for c: 40

          Answer is infinite because a-b is: 0
```

ii)Write a function template to find the maximum value contained in an array. (8) (NOV/DEC 2015)

```
#include <iostream>
using std::cout;
using std::endl;


template<class T> T max(const T* data, int size) {
   T result = data[0];
   for(int i = 1 ; i < size ; i++)
     if(result < data[i])
       result = data[i];
   return result;
  }

template<class T> T min(const T* data, int size) {
   T result = data[0];
   for(int i = 1 ; i < size ; i++)
     if(result > data[i])
       result = data[i];
   return result;
  }
```

```
int main() {
  double data[] = {1.5, 4.6, 3.1, 1.1, 3.8, 2.1};
  int numbers[] = {2, 22, 4, 6, 122, 12, 1, 45};

  const int dataSize = sizeof data/sizeof data[0];
  cout << "Minimum double is " << min(data, dataSize) << endl;
  cout << "Maximum double is " << max(data, dataSize) << endl;

  const int numbersSize = sizeof numbers/sizeof numbers[0];
  cout << "Minimum integer is " << min(numbers, numbersSize) << endl;
  cout << "Maximum integer is " << max(numbers, numbersSize) << endl;

  return 0;
}
Minimum double is 1.1
Maximum double is 4.6
Minimum integer is 1
Maximum integer is 122
```

UNIT-IV / PART-A

1.Define AVL Tree. (NOV/DEC 2013)

An empty tree is height balanced. If T is a non-empty binary tree with TL and TR as its left and right subtrees, then T is height balanced if

i) TL and TR are height balanced and

ii) $|hL - hR| \leq 1$ Where hL and hR are the heights of TL and TR respectively.

2.What do you mean by balanced trees? (JAN 2013)

Balanced trees have the structure of binary trees and obey binary search tree properties. Apart from these properties, they have some special constraints, which differ from one data structure to another. However, these constraints are aimed only at reducing the height of the tree, because this factor determines the time complexity.Ex: AVL trees

3.What are the various rotations in AVL trees? (NOV/DEC 2011)

AVL tree has two rotations. They are single rotation and double rotation. Let A be the nearest ancestor of the newly inserted nod which has the balancing factor ±2. Then the rotations can be classified into the following four categories:

Left-Left: The newly inserted node is in the left sub tree of the left child of A.

Right-Right: The newly inserted node is in the right sub tree of the right child of A.

Left-Right: The newly inserted node is in the right sub tree of the left child of A.

Right-Left: The newly inserted node is in the left sub tree of the right child of A.

4.What do you mean by balance factor of a node in AVL tree?

The height of left sub tree minus height of right sub tree is called balance factor of a node in AVL tree. The balance factor may be either 0 or +1 or -1.The height of an empty tree is -1.

5.What is the idea behind splaying?

Splaying reduces the total accessing time if the most frequently accessed node is moved towards the root. It does not require to maintain any information regarding the height or balance factor and hence saves space and simplifies the code to some extent.

6.What is the minimum number of nodes in an AVL tree of height h?

The minimum number of nodes S(h), in an AVL tree of height h is given by S(h)=S(h-1)+S(h-2)+1. For h=0, S(h)=1.

7.List the types of rotations available in Splay tree.

Let us assume that the splay is performed at vertex v, whose parent and grandparent are p and g respectively. Then, the three rotations are named as:

Zig: If p is the root and v is the left child of p, then left-left rotation at p would suffice. This case always terminates the splay as v reaches the root after this rotation.

Zig-Zig: If p is not the root, p is the left child and v is also a left child, then a left- left rotation at g followed by a left-left rotation at p, brings v as an ancestor of g as well as p.

Zig-Zag: If p is not the root, p is the left child and v is a right child, perform a left-right rotation at g and bring v as an ancestor of p as well as g

8.Define B-tree of order M. What are the applications of B-Tree

A B-tree of order M is a tree that is not binary with the following structural properties:

• The root is either a leaf or has between M/2 and M children.

All non-leaf nodes (except the root) have between

and M children.

• All leaves are at the same depth.

Applications:

• Database implementation

• Indexing on non primary key fields

9.What do you mean by 2-3 tree?

A B-tree of order 3 is called 2-3 tree. A B-tree of order 3 is a tree that is not binary with the following structural properties:

• The root is either a leaf or has between 2 and 3 children.

• All non-leaf nodes (except the root) have between 2 and 3 children.

• All leaves are at the same depth.

10.Define a Relation.

A relation R is defined on a set S if for every pair of elements (a,b), a,b ε S, aRb is either true or false. If aRb is true, then we say that a is related to b.

11.Define an equivalence relation.

An equivalence relation is a relation R that satisfies three properties:

1. (Reflexive) aRa, for all a ε S.
2. (Symmetric) aRb if and only if bRa.
3. (Transitive) aRb and bRc implies that aRc.

12. List the applications of set ADT.

• Maintaining a set of connected components of a graph

• Maintain list of duplicate copies of web pages

•Constructing a minimum spanning tree for a graph

13.What do you mean by disjoint set ADT?

A collection of non-empty disjoint sets S=S1,S2,….,Sk i.e) each Si is a non- empty set that has no element in common with any other Sj. In mathematical notation this is: $Si \cap Sj = \Phi$. Each set is identified by a unique element called its representative.

14.Define a set.

A set S is an unordered collection of elements from a universe. An element cannot appear more than once in S. The cardinality of S is the number of elements in S. An empty set is a set whose cardinality is zero. A singleton set is a set

whose cardinality is one.

15.List the abstract operations in the set.

Let S and T be sets and e be an element.

• SINGLETON(e) returns {e}

• UNION(S,T) returns S Ụ T

• INTERSECTION(S,T) returns S ∩ T

FIND returns the name of the set containing a given element

16.What is the need for path compression?

Path compression is performed during a Find operation. Suppose if we want to perform Find(X), then the effect of path compression is that every node on the path from X to the root has its parent changed to the root.

17.Define Red-Black tree. Write the properties of Red-Black tree

A Red Black tree is a type of self balancing Binary Search Tree. Each node is either colored Red or Black

• No path from root to leaf has two consecutive red nodes (i.e. a parent and its child cannot both be red)

• Every path from leaf to root has the same number of black children

• The root is black

18.Define black height of a tree (JUNE/JULY 2013)

The number of black nodes from root to any leaf is called black height of a tree

Every path from a given node to any of its descendant NIL nodes contains the same number of black nodes. Some definitions: the number of black nodes from the root to a node is the node's black depth; the uniform number of black nodes in all paths from root to the leaves is called the black-height of the red–black tree.

19.Compare 2-3 tree with 2-3-4 tree (JAN 2013)

2-3 tree: often called as 2-3 B tree. Each internal node may have only 2 or 3 children. The internal nodes will store either

one key(with two child nodes) or two keys (with three child nodes)

2-3-4 tree: A B-tree of order 4 is called 2-3-4 tree. A B-tree of order 4 is a tree that is not binary with the following structural properties:

• The root is either a leaf or has between 2 and 4 children.

All non-leaf nodes (except the root) have between 2 and 4 children.

20.Define Fibonacci heaps (NOV/DEC 2014)

• A Fibonacci heap is a heap data structure consisting of a collection of trees. It has a better amortized running time than a binomial heap.

• Binomial heap: eagerly consolidate trees after each insert.

• Fibonacci heap: lazily defer consolidation until next delete-min.

• Set of Heap Ordered Trees

• Maintain pointer to minimum element

21.Mention any four applications of a set

• Used by Boost Graph library to implement its incremental connected components functionality

• Used for implementing Kruskal's algorithm to find the minimum spanning tree of a graph

• Computer networks and a list of bi-directional connections to transfer file from one system to another system

• Connected components of an un directed graph

22.What are the various operations that can be performed on B Trees (NOV/DEC 2015)(APRIL/MAY 2016)

- Adding an element to a B-tree.
- Removing an element from a B-tree.
- Searching for a specified element in a B-tree.

23.What are splay trees(NOV/DEC 2015)

A splay tree is a binary search tree in which restructuring is done using a scheme called splay. The splay is a heuristic method which moves a given vertex v to the root of the splay tree using a sequence of rotations.

24.What is amortized analysis? (APRIL/MAY 2016)(NOV/DEC 2014)

amortized analysis is a method for analyzing a given algorithm's time complexity, or how much of a resource,

especially time or memory in the context of computer programs, it takes to execute.

UNIT-IV / PART-B

1.Discuss, Compare and Contrast Binomial heap and Fibonacci heap in terms of insertion, Deletion and Applications(16)

Binomial heap

Binomial heap is a heap similar to a binary heap but also supports quick merging of two heaps. This is achieved by using a special tree structure. It is important as an implementation of the mergeable heap abstract data type (also called meldable heap), which is a priority queue supporting merge operation

Applications

• Discrete event simulation

• Priority queues

Fibonacci heap

Like a binomial heap, a fibonacci heap is a collection of tree But in fibonacci heaps, trees are not necessarily a binomial tree. Also they are rooted, but not ordered. If neither decrease-key not delete is ever invoked on a fibonacci heap each tree in the heap is like a binomial heap. Fibonacci heaps have more relaxed structure than binomial heaps.

Applications

Minimum spanning tree

Single source shortest path problem

2.i)Construct B tree to insert the following key elements (order of the tree is 3) 5,2,13,3,45,72,4,6,9,22 (8) (NOV/DEC 2011)

A B-tree of order m is an m-way tree (i.e., a tree where each node may have up to m children) in which:
the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree

- all leaves are on the same level
- all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
- the root is either a leaf node, or it has from two to m children
- a leaf node contains no more than m – 1 keys

Inserting into a B-Tree

Attempt to insert the new key into a leaf

If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent

If this would result in the parent becoming too big, split the parent into two, promoting the middle key

This strategy might have to be repeated all the way to the top

If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- 1 - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

ii) Discuss how to insert an element in AVL. Explain with example (8) (NOV/DEC 2011)

AVL trees are height-balanced binary search trees

Balance factor of a node

height(left subtree) - height(right subtree)

An AVL tree has balance factor calculated at every node

For every node, heights of left and right subtree can differ by no more than 1

Store current heights in each node

Height of an AVL Tree

N(h) = minimum number of nodes in an AVL tree of height h.

Basis
› N(0) = 1, N(1) = 2
Induction
› N(h) = N(h-1) + N(h-2) + 1
Solution (recall Fibonacci analysis)
N(h) > _h (_ _ 1.62)
Insert and Rotation in AVL Trees
Insert operation may cause balance factor to become 2 or –2 for some node
only nodes on the path from insertion point to root node have possibly changed in height
So after the Insert, go back up to the root node by node, updating heights
If a new balance factor (the difference hleft-hright) is 2 or –2, adjust tree by rotation around the node
Insert in AVL trees

```
Insert(T : reference tree pointer, x : element) : {
if T = null then
{T := new tree; T.data := x; height := 0; return;}
case
T.data = x : return ; //Duplicate do nothing
T.data > x : Insert(T.left, x);
if ((height(T.left)- height(T.right)) = 2){
if (T.left.data > x ) then //outside case
T = RotatefromLeft (T);
else //inside case
T = DoubleRotatefromLeft (T);}
T.data < x : Insert(T.right, x);
code similar to the left case
Endcase
T.height := max(height(T.left),height(T.right)) +1;
return;
```

3.Describe any scheme for implementing Red Black trees. Explain Insertion and Deletion algorithm in detail. How do these algorithms balance the height of the tree? Insert the following sequence {20,10,5,30,40,57,3,2,4,35,25,18,22,21} (16) (JUNE/JULY 2013)

A binary search tree of height h can implement any of the basic dynamic-set operations--such as SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT, and DELETE--in O(h) time. Thus, the set operations are fast if the height of the search tree is small; but if its height is large, their performance may be no better than with a linked list. Red-black trees are one of many search-tree schemes that are "balance" in order to guarantee that basic dynamic-set operations take O(lg n) time in the worst case.
Properties of red-black trees
A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the way nodes can be colored on any path from the root to a leaf, redblack trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.
Each node of the tree now contains the fields color, key, left, right, and p. If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NIL. We shall regard these NIL'Sas being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.
A binary search tree is a red-black tree if it satisfies the following red-black properties:
1. Every node is either red or black.
2. Every leaf (NIL) is black.
3. If a node is red, then both its children are black.

4. Every simple path from a node to a descendant leaf contains the same number of black nodes.
An example of a red-black tree is shown in Figure 14.1.
We call the number of black nodes on any path from, but not including, a node x to a leaf the black-height of the node, denoted bh(x). By property 4, the notion of black-height is well defined, since all descending paths from the node have the same number of black nodes. We define the black-height of a red-black tree to be the blackheight of its root.
A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, every leaf (NIL) is black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. Each non-NIL node is marked with its blackheight; NIL'S have black-height 0.
Insertion
RB-INSERT(T, x)
TREE-INSERT(T, x)
color[x] ← RED ⊳ only RB property 3 can be violated
while x ≠ root [T] and color [p[x]] = RED
do if p[x] = left[p[p[x]]
then y ← right[p[p[x]] ⊳ y = aunt/uncle of x
if color [y] = RED
then Case 1⟩
else if x = right[p[x]]
then Case 2 ⊳ Case 2 falls into Case 3
Case 3
else "then" clause with "left" and "right" swapped ⟩
color[root[T]] ← BLACK

Deletion

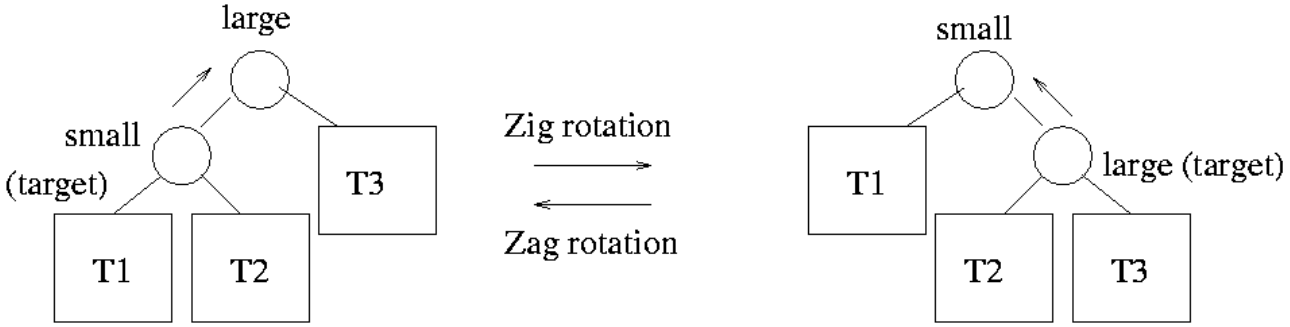
First, search for an element to be deleted.
• If the element to be deleted is in a node with only left child, swap this node with the one containing the largest element in the left sub tree. (This node has no right child).
• If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right sub tree (This node has no left child).
If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways.
While swapping, swap only the keys but not the colors.
• The item to be deleted is now in a node having only a left child or only a right child. Replace this node with its sole child. This may violate red constraint or black constraint. Violation of red constraint can be easily fixed.
• If the deleted node is block, the black constraint is violated. The removal of a black node y causes any path that contained y to have one fewer black node.
• Two cases arise:
1.The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
2. The replacing node is black.

4.What is Splay tree? Discuss briefly the various rotations in splay tree with example (16)

Splay Trees

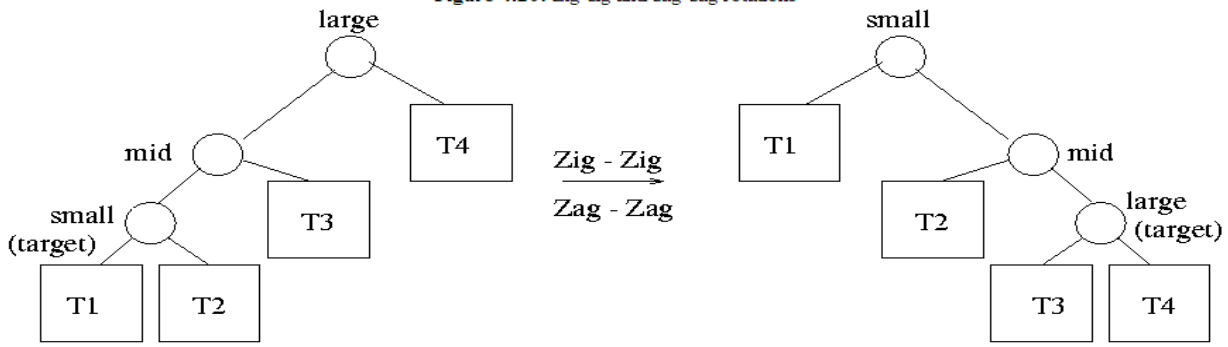
A splay tree is a binary search tree. It has one interesting difference, however: whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant. If the next lookup request is for the same element, it can be returned immediately. In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus found quickly.
Rotation 1: Simple rotation
The simple tree rotation used in AVL trees and treaps is also applied at the root of the splay tree, moving the

splayed node x up to become the new tree root. Here we have A < x < B < y < C, and the splayed node is either x or y depending on which direction the rotation is. It is highlighted in red.



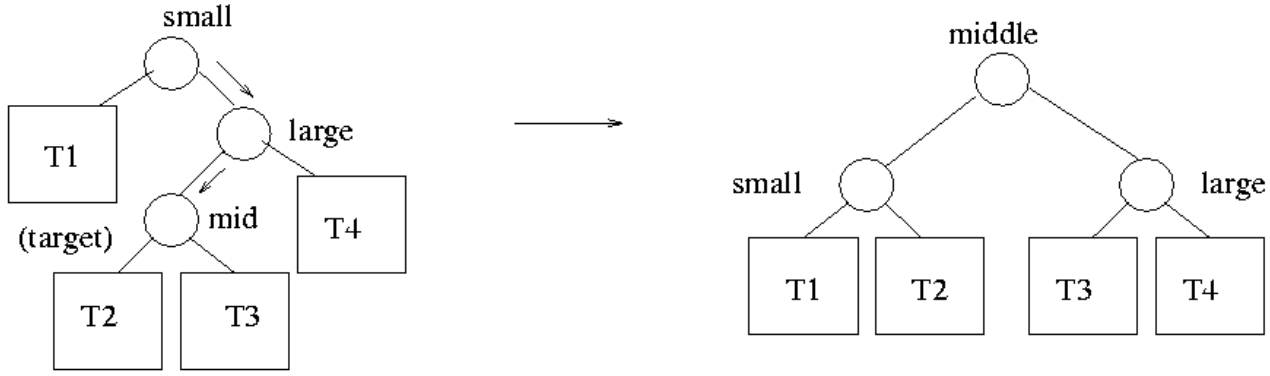
Rotation 2: Zig-Zig and Zag-Zag

Lower down in the tree rotations are performed in pairs so that nodes on the path from the splayed node to the root move closer to the root on average. In the "zig-zig" case, the splayed node is the left child of a left child or the right child of a right child ("zag-zag").



Rotation 3: Zig-Zag

In the "zig-zag" case, the splayed node is the left child of a right child or vice-versa. The rotations produce a subtree whose height is less than that of the original tree. Thus, this rotation improves the balance of the tree. In each of the two cases shown, y is the splayed node:



5.Discuss in detail about B-tree and clearly explain operations of B-tree with example (16)

B-tree is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children

```
B-Tree-Create (T)
x = allocate-node ();
leaf[x] = True
n[x] = 0
Disk-Write (x)
root[T] = x
```

Inserting a key into a B-tree

```
B-Tree-Insert (T, k)
r = root[T]
if n[r] = 2t - 1 then
s = allocate-node ()
root[T] = s
leaf[s] = False
n[s] = 0
c1[s] = r
B-Tree-Split-Child (s, 1, r)
B-Tree-Insert-Nonfull (s, k)
else
B-Tree-Insert-Nonfull (r, k)
Endif
B-Tree-Insert-Nonfull (x, k)
i = n[x]
if leaf[x] then
while i >= 1 and k < keyi[x] do
keyi+1[x] = keyi[x]
i--
end while
keyi+1[x] = k
n[x]++
else
while i >= 1 and k < keyi[x] do
i--
end while
i++
Disk-Read (ci[x])
if n[ci[x]] = 2t - 1 then
B-Tree-Split-Child (x, i, ci[x])
if k > keyi[x] then i++
end if
B-Tree-Insert-Nonfull (ci[x], k)
end if
```

To insert the key k into the node x, there are two cases:

- x is a leaf node. Then we find where k belongs in the array of keys, shift everything over to the left, and stick k inthere.
- x is not a leaf node. We can't just stick k in because it doesn't have any children; children are really only created
  when we split a node, so we don't get an unbalanced tree.

6.Explain Amortized analysis with an example (16) (JAN 2013)

Amortized Analysis (another way to provide a performance guarantee is to amortize the cost, by keeping track of the total cost of all operations, divided by the number of operations. In this setting, we can allow some expensive operations, while keeping the average cost of operations low. In other words, we spread the cost of the

few expensive operations, by assigning a portion of it to each of a large number of inexpensive operations)

Multipop Stack

This is a stack with the added operation MULTIPOP. Thus, the operations are:

• PUSH(x) pushes x onto the top of the stack

• POP()pops the top of the stack and returns the popped object

• MULTIPOP(x) pops the top xitems off the stack

If a MULTIPOP STACK has x items on it, PUSH and POP each require time O(1) time and multi-pop requires O(min(k,n))operations

We will analyze a sequence of x PUSH, POP, and MULTIPOP operations, assuming we are starting with an empty stack. The worst-case cost of a sequence of x operations is x times the worst-case cost of any of the operations. The worst-case cost of the most expensive operation, MULTIPOP(K), is O(n) , so the worst-case cost of a sequence of x operations is O(n2) We will see shortly that this bound is not tight

Binary Counter

A binary counter is x-bit counter that starts at 0, and supports the operation INCREMENT.

The counter value x is represented by a binary array A[0,...k-1] where x = , where To assign values to the bits, we use the operations SET and RESET, which set the value to 1 and 0,respectively. Each of these operations has a cost of 1. INCREMENT is implemented as follows:

Increment()
i=0
while(i<k and A[i]=1)
Reset(A[i])
i++
if(i<k)
Set(A[i])

7.Explain Fibonacci heap Deletion and Decrease Key operation using cascading cut procedure with example (16)(JUNE/JULY 2013)

Fibonacci Heap

- Heap ordered Trees
- Rooted, but unordered
- Children of a node are linked together in a Circular, doubly linked List, E.g node 52

Node Pointers

- left [x]
- right [x]
- degree [x] - number of children in the child list of x - - mark [x]
    - 2*k* nodes
    - k = height of tree
    - (k i) nodes at depth i
    - Unordered binomial tree *Uk* has root with degree k greater than any other node. Children are trees *U*0, *U*1, ..,
    - *Uk*-1 in some order.

• Collection of unordered Binomial Trees.

• Support Mergeable heap operations such as Insert, Minimum, Extract Min, and Union in constant time O(1) Desirable when the number of Extract Min and Delete operations are small relative to the number of other operations.

• Most asymptotically fastest algorithms for computing minimum spanning trees and finding single source shortest paths, make use of the Fibonacci heaps.

Fibonacci Heap Operations

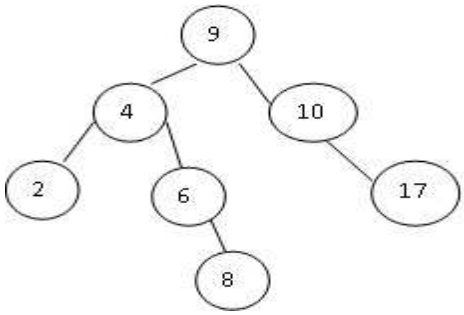
. Make Fibonacci Heap - Assign N[H] = 0, min[H] = nil
Amortized cost is O(1)

. Find Min - Access Min[H] in O(1) time

. Uniting 2 Fibonacci Heaps

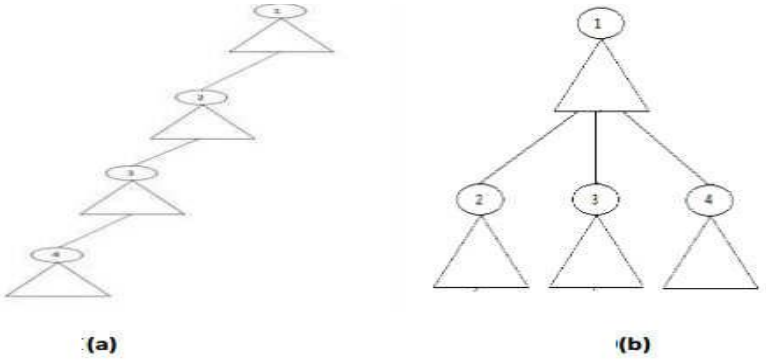
Concatenate the root lists of Heap1 and Heap2
- Set the minimum node of the new Heap
- Free the 2 heap objects
Amortized cost is equal to actual cost of O(1)
Insert – amortized cost is O(1)
- Initialize the structural fields of node x, add it to the root list of H
- Update the pointer to the minimum node of H, min[H]
- Increment the total number of nodes in the Heap, n[H]
Node 21 has been inserted
- Red Nodes are marked nodes – they will become relevant only in the delete operation
Fibonacci Heap Decrease Key Operation
A node is ***marked*** if:
• At some point it was a root then it was linked to anther node and 1 child of it has been cut
• Newly created nodes are always unmarked
A node becomes unmarked when ever it becomes the child of another node.
We mark the fields to obtain the desired time bounds. This relates to the "potential function" used to analyze the time complexity of Fibonacci Heaps.
Decrease-Key - amortized cost is O(1)
• Check that new key is not greater than current key, then assign new key to x
• If x is a root or if heap property is maintained, no structural changes
• Else
o ***Cut* x:** make x a root, remove link from parent y, clear marked field of x
-Perform a ***Cascading cut*** on x's parent y(relevant if parent is marked):
- if unmarked: mark y, return

8. i)Show the result of inserting 10,17,2,4,9, 6,8 into an AVL tree (8)



ii)Discuss the disjoint set find with path compression using suitable algorithm (8)

Path compression, is also quite simple and very effective. As shown in Figure we use it during Find-set operations to make each node on the find path point directly to the root. Path compression does not change any ranks.



Path compression during the operation Find-set. (a) A tree representing a set prior to executing Find-set(a). (b) The same set after executing Find-set(a). Each node on the find path now points directly to the root.

- The Find-set procedure is a two-pass method: it makes one pass up the find path to find the root, and it makes a second pass back down the find path to update each node so that it points directly to the root.
- Snippet for Find-set function using path compression:

```
Find-set(x)
if x ≠ p[x]
then p[x] = Find-set(p[x])
return p[x]
```

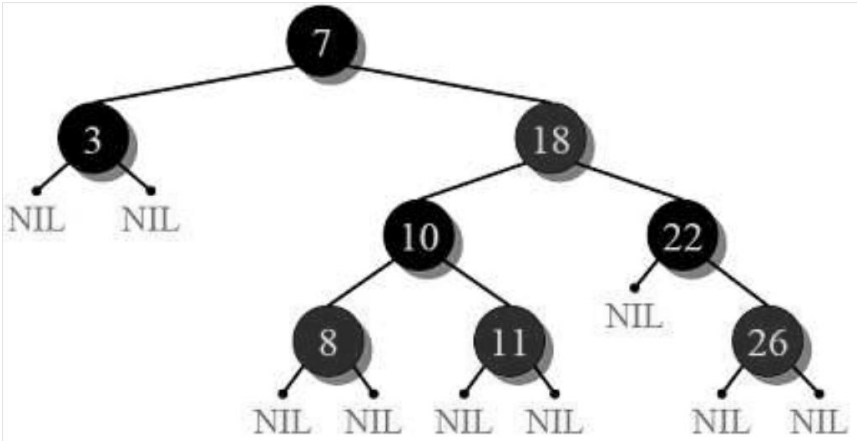
Union-by-rank or path-compression improves the running time of the operations on disjoint-set forests, and the improvement is even better when the two heuristics are used together.

We do not prove it, but, if there are n Make-set operations and at most n - 1 Union operations and f Find-set operations, the path-compression heuristic alone gives a worst-case running time of Θ (n + f · (1 + log2+ f / n n)).

When we use union by rank and path compression together, the worst-case running time is O(m α (m,n)), where α(m,n) is a very slowly growing function and the value of it is derived from the inverse of Ackermann function. In any application of a disjoint-set data structure, α(m,n) $\leq$ 4. Thus, for practical purposes the running time can be viewed as linear in m (no. of operations).

9.i)What is a Red Black Tree? Mention the properties that a R-B tree holds (6) (NOV/DEC 2014)

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.

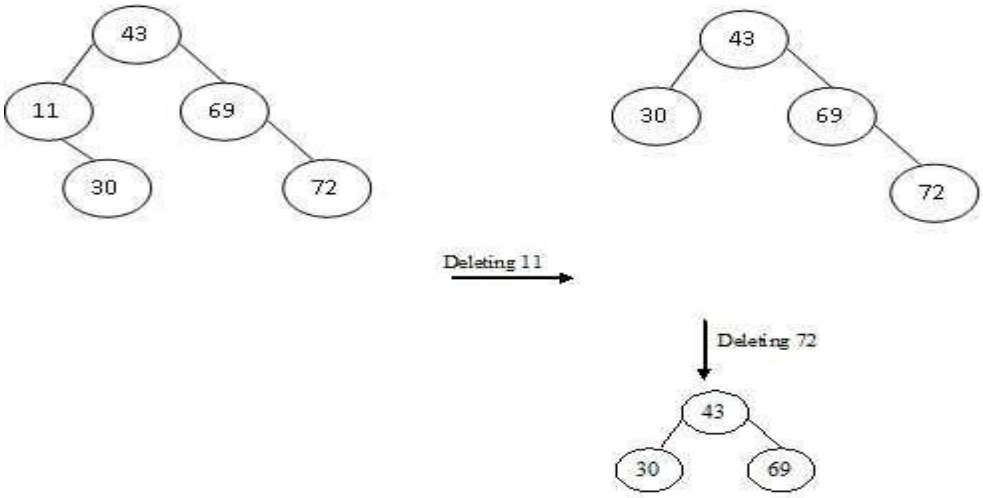


Every node has a color either red or black.
Root of tree is always black.
There are no two adjacent red nodes (A red node cannot have a red parent or red child).
Every path from root to a NULL node has same number of black nodes.

ii)Show the results of inserting 43,11,69,72 and 30 into an initially empty AVL tree.

Show the results of deleting the nodes 11 and 72 one after the other of the constructed tree. (10)

(NOV/DEC 2014)



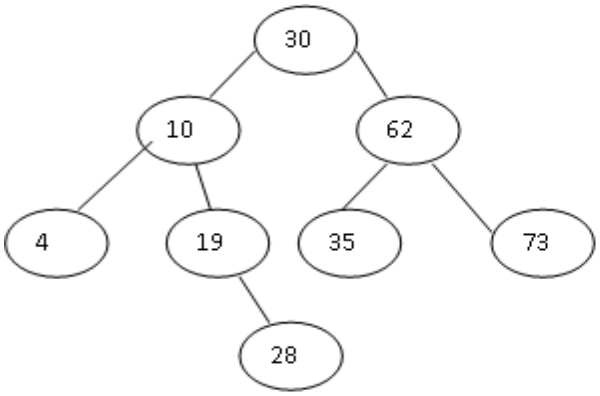
10.i) What is a B-Tree? Mention the properties that a B-Tree holds (6) (NOV/DEC 2014)

A **B-tree** is a **tree data structure** that keeps **data** sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.

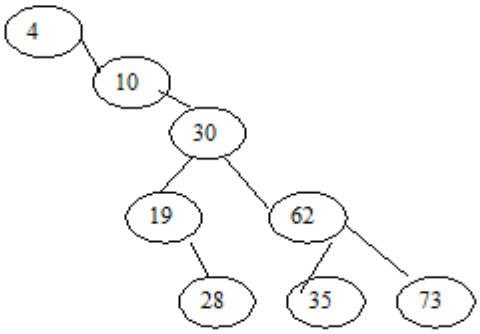
Properties of B-Tree

- All leaves are at same level.
- A B-Tree is defined by the term minimum degree 't'. The value of t depends upon disk block size.
- Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.
- All nodes (including root) may contain at most 2t – 1 keys.
- Number of children of a node is equal to the number of keys in it plus 1.
- All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in
- range from k1 and k2.
- B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward
- and also shrink from downward.
- Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

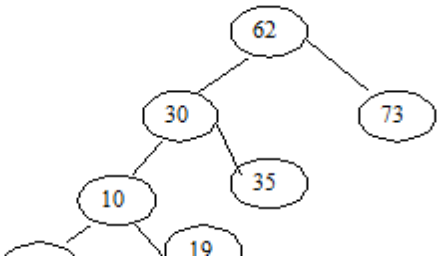
ii) Construct a Binary Search Tree by inserting 30, 10, 4, 19, 62, 35, 28, and 73 into an initially empty tree. Show the results of splaying the nodes 4 and 62 one after the other of the constructed tree (10) (NOV/DEC2014)



After Splaying 4



After Splaying 62

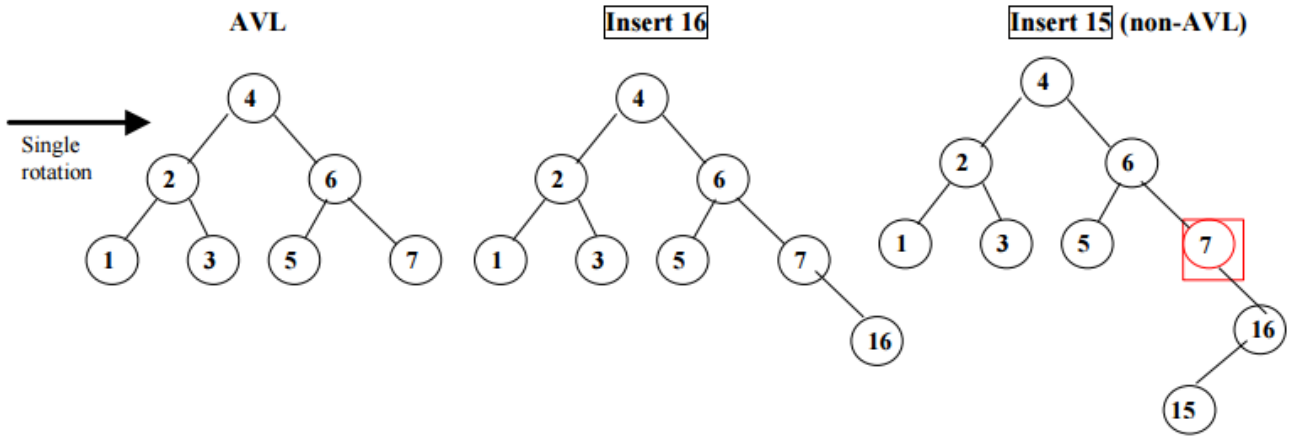


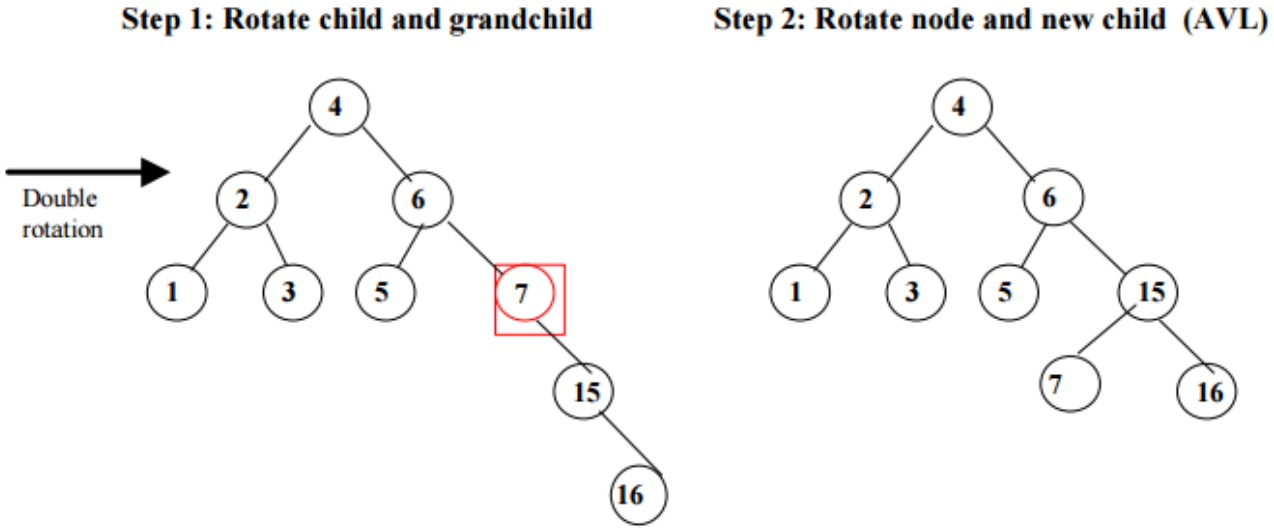
11.i)Define AVL Tree and starting with an empty AVL search Tree insert the following elements in the given order: 35,45,65,55,75,15,25(8) (NOV/DEC 2015)

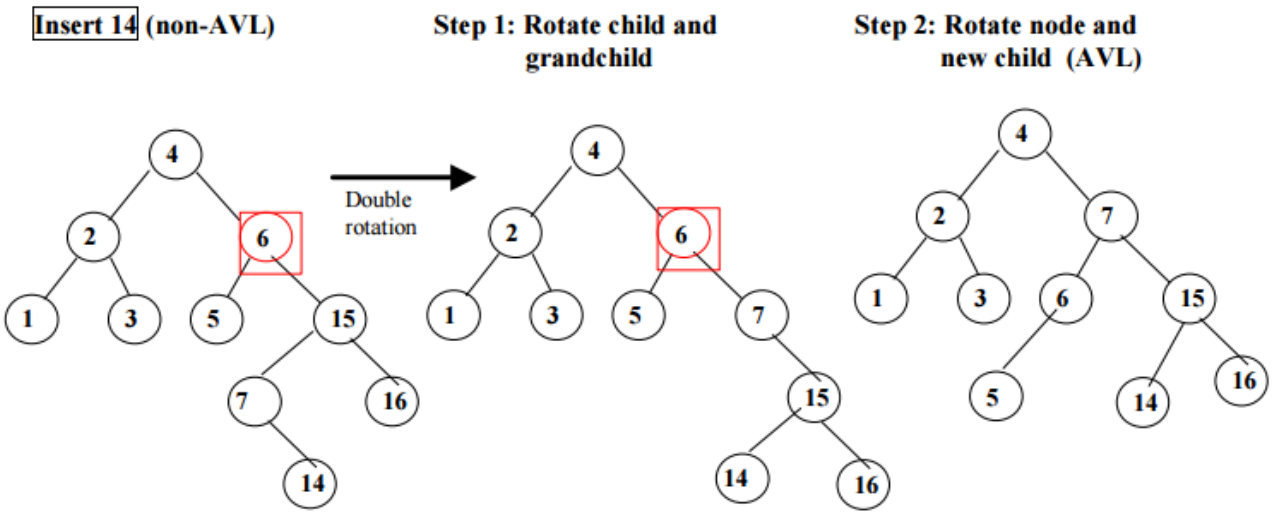
```
1.Insert 35          Insert 45          Insert 65          Insert 55

   (35)                (35)               (35)               (35)
                         \                  \                  \
                         (45)               (45)               (45)
                                              \                /   \
                                              (65)          (55)   (65)
```

```
Insert 75          Insert 15          Insert 25

   (35)               (35)               (35)
    \                 /   \              /   \
    (45)            (15)  (45)         (15)   (45)
     / \                  / \           / \    /  \
   (55)(65)             (55)(65)      (25)(55)(65)
          \                    \                  \
          (75)                 (75)               (75)
```

ii)Explain the AVL rotations with suitable example.(8)(NOV/DEC 2015)



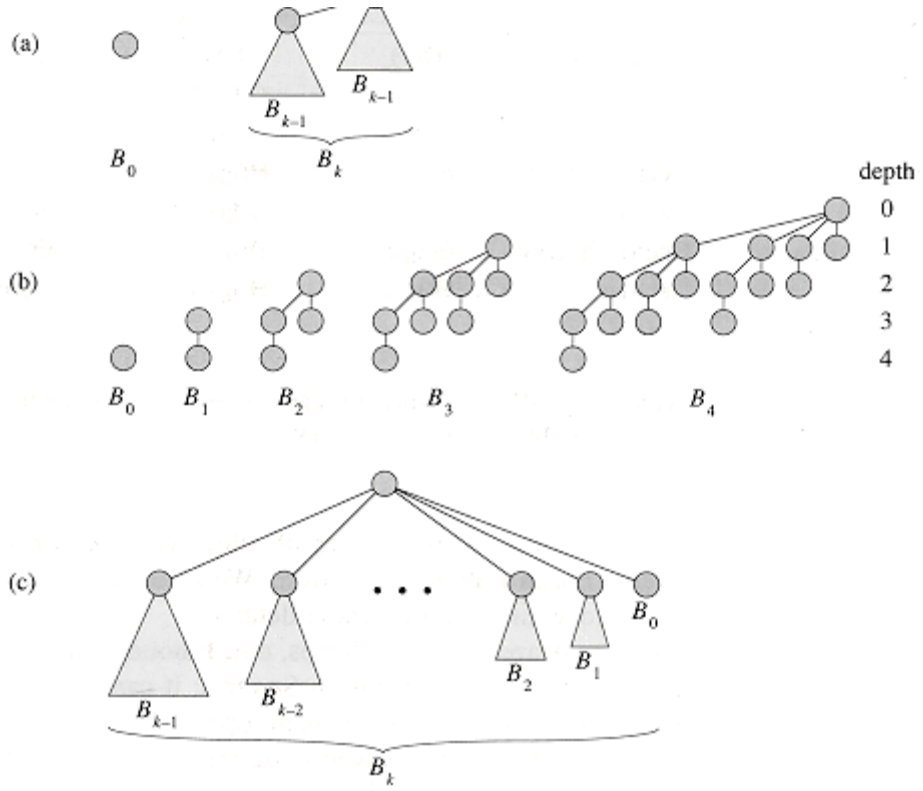




12.llustrate the construction of Binomial Heaps and its operations with a suitable example(16)(NOV/DEC 2015)

The ***binomial tree*** B_k is an ordered tree defined recursively, the binomial tree B_0 consists of a single node. The binomial tree B_k consists of two binomial trees B_k-1 that are ***linked*** together: the root of one is the leftmost child of the root of the other.

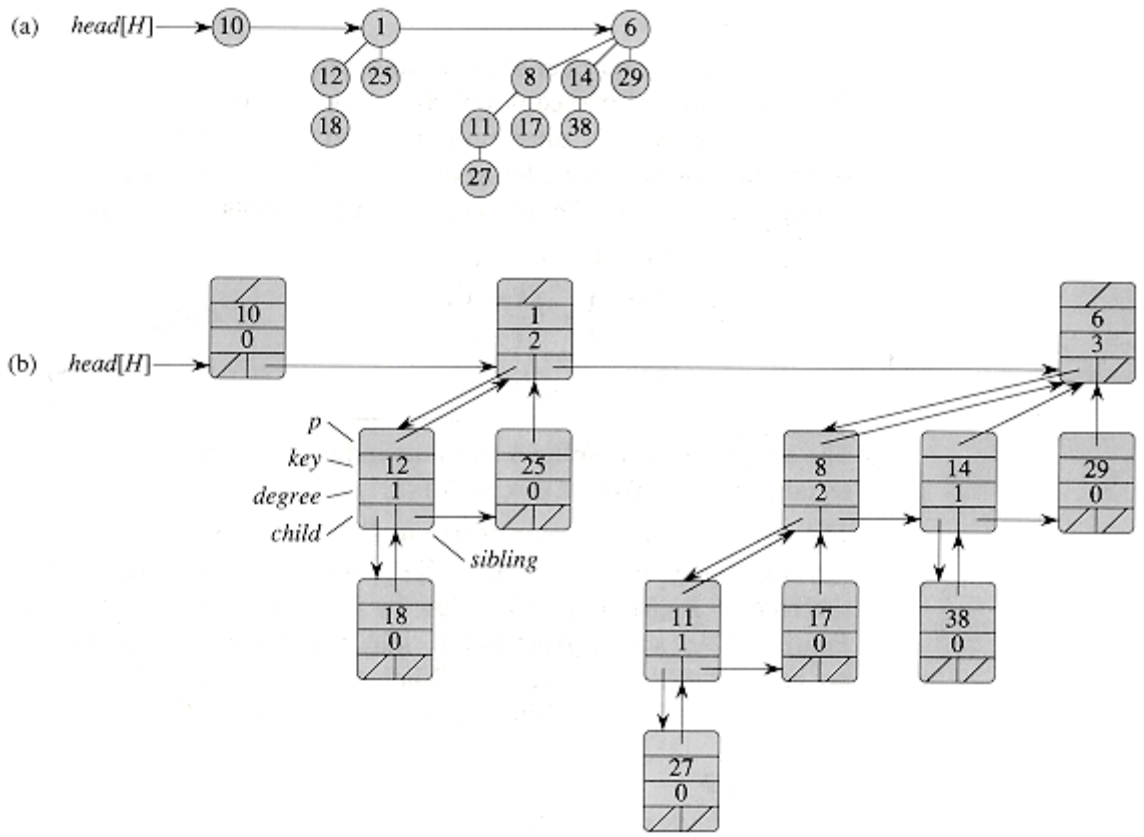
Some properties of binomial trees are given by the following lemma.



A ***binomial heap H*** is a set of binomial trees that satisfies the following ***binomial-heap properties.***

1. Each binomial tree in *H* is ***heap-ordered:*** the key of a node is greater than or equal to the key of its parent.

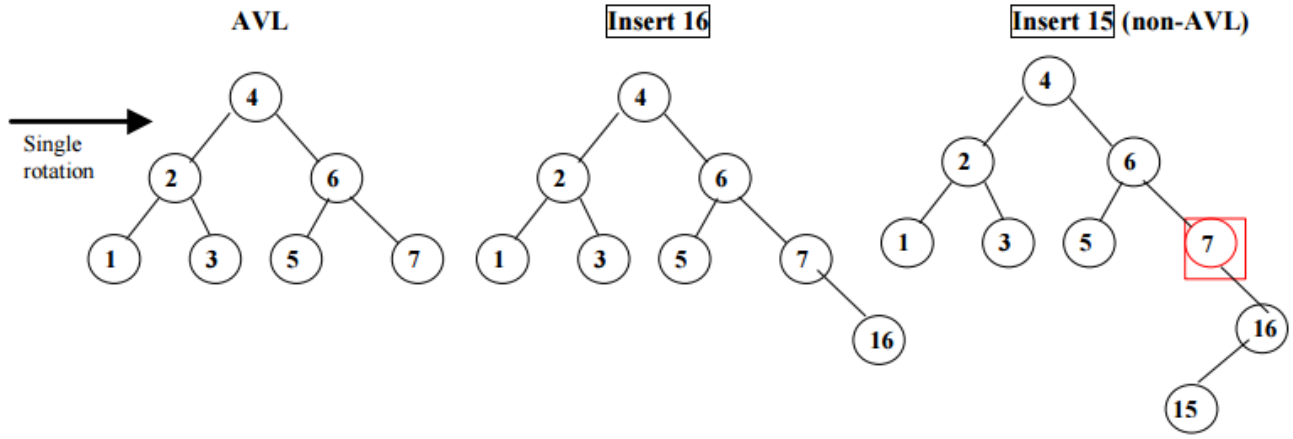
2. There is at most one binomial tree in *H* whose root has a given degree.

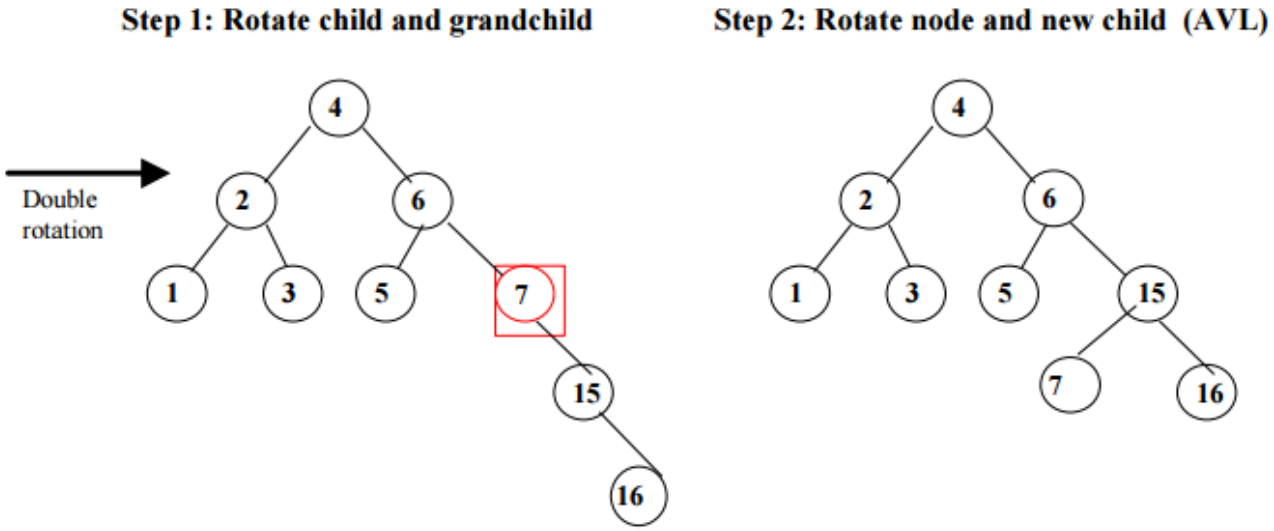


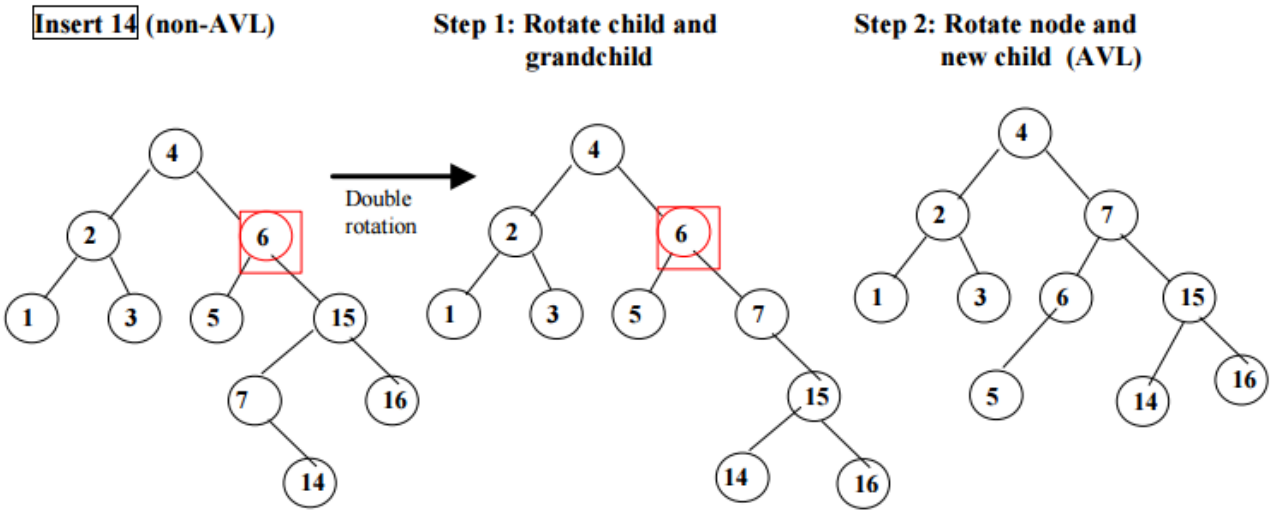
13.i)Define AVL Tree and starting with an empty AVL search Tree insert the following elements in the given order: 2,1,4,5,9,3,6,7(8) (APRIL/MAY 2016)

```
1.Insert 2          Insert 1          Insert 4          Insert 5

   (2)                (2)               (2)               (2)
                      /                 / \               / \
                    (1)               (1)  (4)          (1)  (4)
                                                              \
                                                              (5)

Insert 9          Insert 3          Insert 6          Insert 7

   (2)                (4)               (4)               (4)
   / \                / \               /  \              /  \
 (1)  (5)           (2)  (5)          (2)   (6)         (2)    (6)
     / \            / \    \          / \   / \         / \    / \
   (4)  (9)       (1)  (3)  (9)     (1)(3)(5)(9)      (1)(3)(5)  (9)
                                                                /
                                                              (7)
```

ii)Explain the AVL rotations with suitable example.(8)(APRIL/MAY 2016)







14.Implement the Fibonacci heaps and compare their performance with binary heaps when used in Dijkstras algorithm(16) (APRIL/MAY 2016)

We use Fibonacci heaps to implement the priority queue needed in Dijkstra's algorithm. The items and their keys are stored as nodes in a collection of heap-ordered trees. A heap-ordered tree is a rooted tree where the key of any node is no less than the key of its parent. The number of children of a node is called its rank. Nodes can be either marked or unmarked. Root nodes are never marked. A node containing the item I with key K is stored as an item(I,K,R,P,M) constraint, where R is its rank and P is the item of its parent (or 0 if it is a root). The last argument M is u if the node is unmarked and m if it is marked. The minimum-key pair is stored as a min/2 constraint:

keep_min @ min(_,A) \ min(_,B) <=> A =< B | true.

Insert: Inserting a new item I with key K is done by adding an unmarked isolated root node and updating the minimum: insert @ insert(I,K) <=> item(I,K,0,0,u), min(I,K).

Extract-min:Extracting the minimum node is done as follows.

- First we find and remove the min constraint (if there is none, the heap is empty and we fail) and the corresponding item.
- Then we convert the children of the minimum node to roots, which is done by the ch2rt constraint.
- Finally we find the new minimum (done by the findmin constraint) and return the (old) minimum item. extr @ extract_min(X,Y), min(I,K), item(I,_,_,_,_) <=> ch2rt(I), findmin, X=I, Y=K. extr_none @ extract_min(_,_) <=> fail. c2r @ ch2rt(I) \ item(C,K,R,I,_) <=> item(C,K,R,0,u). c2r_done @ ch2rt(I) <=> true. To find the new minimum, it suffices to search the root nodes: findmin @ findmin, item(I,K,_,0,_) ==> min(I,K). foundmin @ findmin <=> true.

Decrease-key: The decrease-key operation removes the original item constraint and calls decr/5 if the new key is smaller (and fails otherwise). decr @ decr(I,K), item(I,O,R,P,M) <=> K < O | decr(I,K,R,P,M). decr_nok @ decr(I,K)

<=> fail

When a key is decreased, we may have found a new minimum:

d_min @ decr(I,K,_,_,_) ==> min(I,K).

Decreasing the key of a root cannot cause a violation of the heap order:

d_root @ decr(I,K,R,0,_) <=> item(I,K,R,0,u).

If the new key is not smaller than the parent's key, there is also no problem:

d_ok @ item(P,PK,_,_,_) \ decr(I,K,R,P,M) <=> K>=PK | item(I,K,R,P,M).

UNIT-V / PART-A

1.Define Graph.
A graph G consist of a nonempty set V which is a set of nodes of the graph, a set E which is the set of edges of the graph, and a mapping from the set for edge E to a set of pairs of elements of V. It can also be represented as G=(V, E).

2.Define adjacent nodes.
Any two nodes which are connected by an edge in a graph are called adjacent nodes. For example, if an edge x ε E is associated with a pair of nodes (u,v) where u, v ε V, then we say that the edge x connects the nodes u and v.

3.What is a directed graph?
A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network.

4.What is an undirected graph?
An undirected graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are bidirectional. An undirected graph is sometimes called an undirected network.

5.What is a loop?
An edge of a graph which connects to itself is called a loop or sling. Where graphs are defined so as to allow loops and multiple edges, a graph without loops or multiple edges is often distinguished from other graphs by calling it a simple graph.

6.What is a simple graph?
A simple graph is a graph, which has not more than one edge between a pair of nodes than such a graph is called a simple graph.

7.What is a weighted graph?
A weighted graph is a graph in which each branch is given a numerical weight. A weighted graph is therefore a special type of labeled graph in which the labels are numbers (which are usually taken to be positive).

8.Define out degree of a graph?
In a directed graph, for any node v, the number of edges which have v as their initial node is called the out degree of the node v.

9.Define in degree of a graph?
In a directed graph, for any node v, the number of edges which have v as their terminal node is called the in degree of the node v.

10.Define path in a graph?
The path in a graph is the route taken to reach terminal node from a starting node. A path in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence.

11.What is a simple path?
A path in a diagram in which the edges are distinct is called a simple path. It is also called as edge simple.

A path in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence.

12.What is a cycle or a circuit?

A path which originates and ends in the same node is called a cycle or circuit.

A cycle is a path where the last vertex is adjacent to the first. A cycle in which no vertex repeats (such as 1-2-3-1 versus 1-2-3-2-1) is said to be simple.

13.What is an acyclic graph?

A simple diagram which does not have any cycles is called an acyclic graph.

An acyclic graph does not contain any cycles. Trees are connected acyclic undirected graphs. Directed acyclic graphs are called DAGs.

14.What is meant by strongly connected in a graph?

An undirected graph is connected, if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected.

15.When is a graph said to be weakly connected?

When a directed graph is not strongly connected but the underlying graph is connected, then the graph is said to be weakly connected.

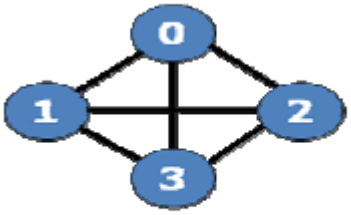
16.What are the different ways to represent graph (NOV/DEC 2014)(APRIL/MAY 2016)

Two ways of representing a graph are:

a. Adjacency matrix

b. Adjacency list

Consider the graph below:

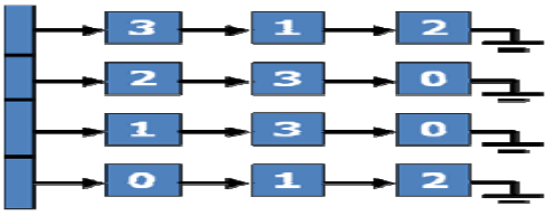


G1

Adjacency matrix representation is

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency list representation is



17.What is an undirected acyclic graph?

When every edge in an acyclic graph is undirected, it is called an undirected acyclic graph. It is also called as undirected forest.

18.What are the two traversal strategies used in traversing a graph?

• Breadth First Search

• Depth First Search

19.Name two algorithms two find minimum spanning tree

Kruskal'salgorithm

Prim's algorithm

20.List the two important key points of depth first search.

i) If path exists from one node to another node, walk across the edge – exploring the edge.

ii) If path does not exist from one specific node to any other node, return to the previous node where we have been before – backtracking.

21.What is the minimum number of spanning tree possible for a complete graph with n vertices? (NOV/DEC 2015)

The number of spanning trees of G, denoted by t(G), is the total number of distinct spanning sub graphs of G that are trees. We consider the problem of determining some special classes of graphs having the maximum number of spanning trees (or the maximum spanning tree graph problem).

22.What is topological sorting(NOV/DEC 2015)

Topological or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge u v from vertex u to vertex v, u comes before v in the ordering.

23.List out the applications of depth—first search(APRLL/MAY 2016)

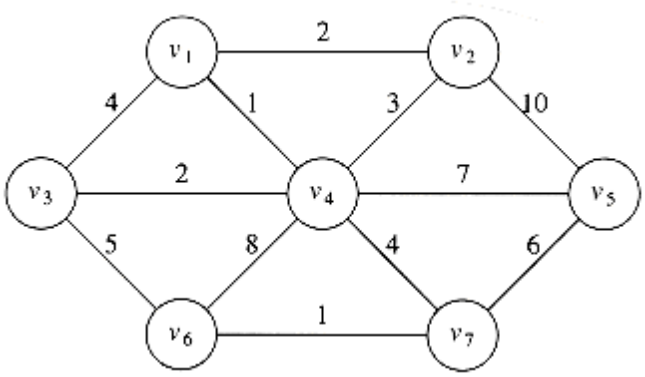
- Detecting cycle in a graph
- Path Finding
- Topological Sorting
- To test if a graph is bipartite

UNIT-V / PART-B

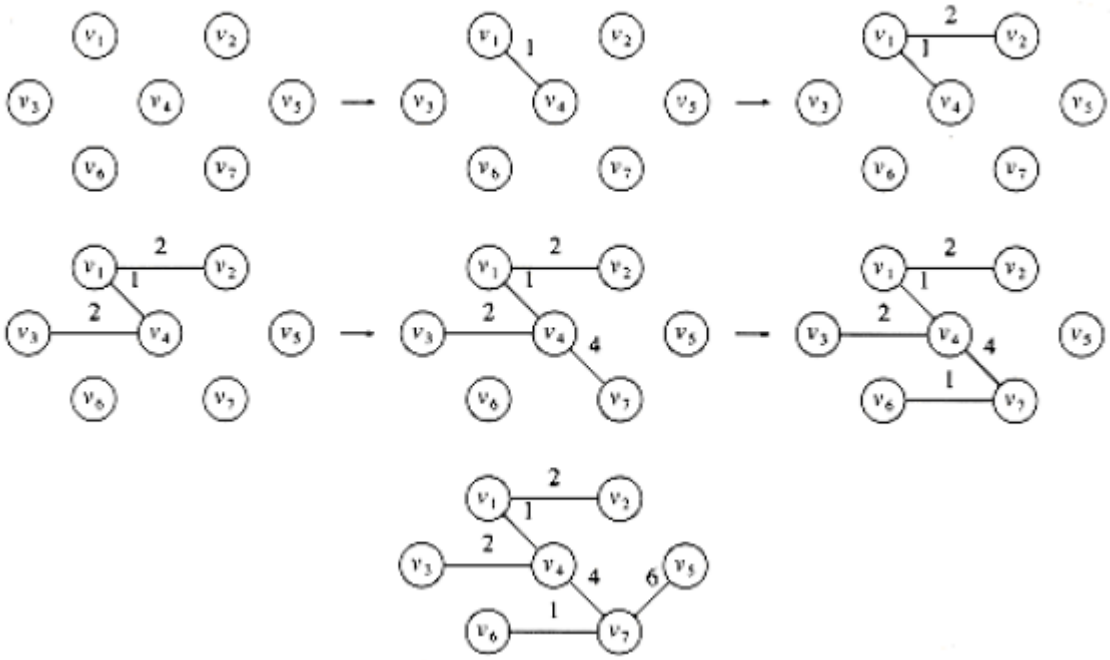
1.**Write and explain the prim's algorithm with an example (16) (MAY/JUNE 2012) (NOV/DEC 2011)**

Prim's Algorithm to Construct a Minimal Spanning Tree
Input: A weighted, connected and undirected graph G = (V,E).

Output: A minimal spanning tree of G.
Step 1: Let x be any vertex in V. Let X = and Y = V
Step 2: Select an edge (u,v) from E such that, and (u,v)has the smallest weight among edges between X and Y.
Step 3: Connect u to v.
Step 4: If Y is empty, terminate and the resulting tree is a minimal spanning tree. Otherwise, go to Step 2.



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. A minimum spanning tree exists if and only if *G* is connected
Prim's Algorithm



Initial configuration of table used in Prim's algorithm

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

Final answer:

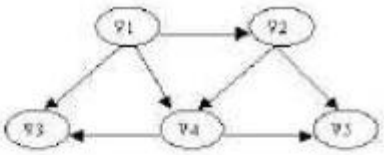
v	Known	*dv*	*pv*
v1	1	0	0
v2	1	2	v1
v3	1	2	v4
v4	1	1	v1
v5	1	6	v7
v6	1	1	v7
v7	1	4	v4

2.Explain topological sort with suitable algorithm and example (16) (MAY/JUNE 2012)

Definition: A **topological sort** is a linear 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 linear ordering.
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.
To implement the topological sort, perform the following steps.
Step 1 : - Find the indegree for every vertex.
Step 2 : - Place the vertices whose indegree is `0' on the empty queue.
Step 3 : - Dequeue the vertex V and decrement the indegree's of all its adjacent vertices.
Step 4 : - Enqueue the vertex on the queue, if its indegree falls to zero.
Step 5 : - Repeat from step 3 until the queue becomes empty.
Step 6 : - The topological ordering is the order in which the vertices dequeued.
Routine to perform Topological Sort

```
void topsort (graph g)
{
queue q ;
int counter = 0;
vertex v, w ;
q = createqueue (numvertex);
makeempty (q);
for each vertex v
if (indegree [v] = = 0)
enqueue (v, q);
while (! isempty (q))
{
v = dequeue (q);
topnum [v] = + + counter;
for each w adjacent to v
if (--indegree [w] = = 0)
enqueue (w, q);
}
if (counter ! = numvertex)
error (" graph has a cycle");
disposequeue (q); /* free the memory */
}
```

Example:



1. Compute the indegrees:

   V1: 0
   V2: 1
   V3: 2
   V4: 2
   V5: 2

2. Find a vertex with indegree 0: V1
3. Output V1 , remove V1 and update the indegrees:

   Sorted: V1
   Remove edges: (V1,V2) , (V1, V3) and (V1,V4)
   Updated indegrees:

   V2: 0
   V3: 1
   V4: 1
   V5: 2

The process is depicted in the following table:

	Indegree					
Sorted →		V1	V1,V2	V1,V2,V4	V1,V2,V4,V3	V1,V2,V4,V3,V5
V1	0					
V2	1	0				
V3	2	1	1	0		
V4	2	1	0			
V5	2	2	1	0	0	

One possible sorting: V1, V2, V4, V3, V5

3.Compare BFS and DFS (16) ((MAY/JUNE 2012)

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

• Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. Intuitively, one starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

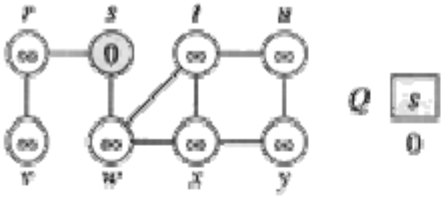
• The big advantage of DFS is that it has much lower memory requirements than BFS, because it's not necessary to store all of the child pointers at each level.

• **Algorithm: BFS**

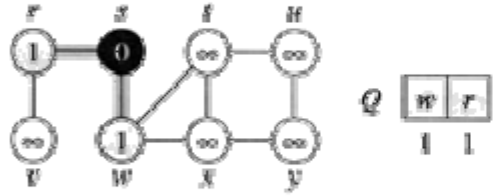
```
bfs(g)
{
list l = empty
tree t = empty
choose a starting vertex x
search(x)
while(l nonempty)
remove edge (v,w) from start of l
if w not yet visited
{
add (v,w) to t
```

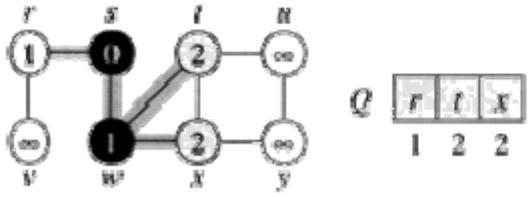
search(w)
}
}

Example: BFS
After initialization (paint every vertex white, set d[u] to infinity for each vertex u, and set the parent of every vertex to be NIL), the source vertex is discovered in line 5. Lines 8-9 initialize Q to contain just the source vertex s.

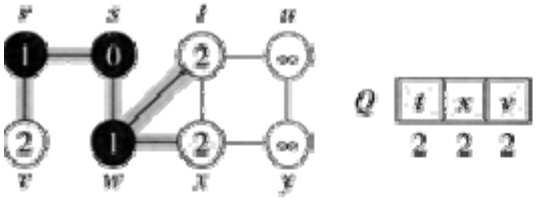


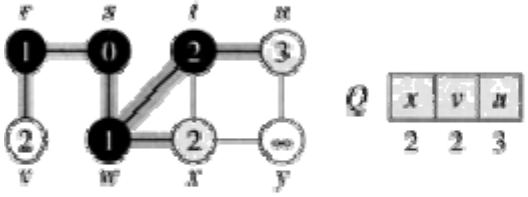
The algorithm discovers all vertices 1 edge from s i.e., discovered all vertices (w and r) at level 1.

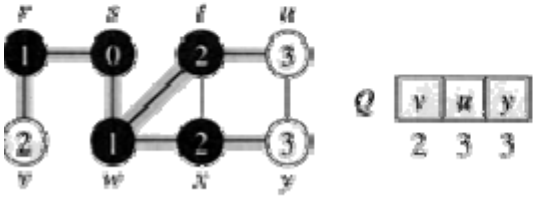




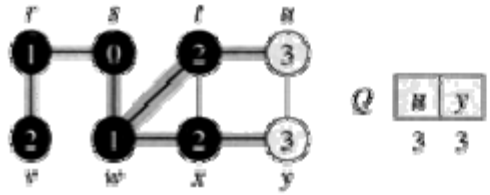
The algorithm discovers all vertices 2 edges from s i.e., discovered all vertices (t, x, and v) at level 2.

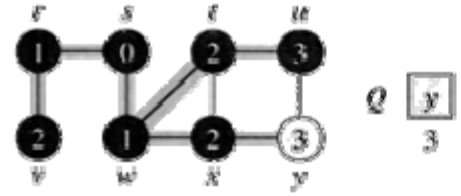




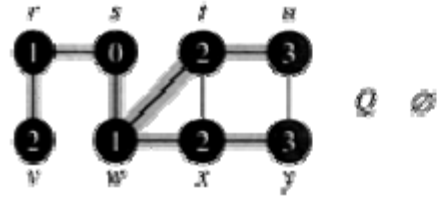


The algorithm discovers all vertices 3 edges from s i.e., discovered all vertices (u and y) at level 3.





The algorithm terminates when every vertex has been fully explored.



Algorithm: DFS
DFS (V, E)
1. **for** each vertex *u* in V[G]

2. **do** color[*u*] ← WHITE
3. π[*u*] ← NIL
4. time ← 0
5. **for** each vertex *u* in V[G]
6. **do if** color[u] ← WHITE
7. **then** DFS-Visit(*u*) ▷ build a new DFS-tree from *u*

DFS-Visit(u)

1. color[u] ← GRAY ▷ discover *u*
2. time ← time + 1
3. d[u] ← time
4. **for** each vertex *v* adjacent to *u* ▷ explore (*u*, *v*)
5. **do if** color[*v*] ← WHITE
6. **then** π[*v*] ← *u*
7. DFS-Visit(*v*)
8. color[*u*] ← BLACK
9. time ← time + 1
10. f[*u*] ← time ▷ we are done with *u*

4.i)Describe the various representations of graphs (6) (NOV/DEC 2011)

Graph can be represented by Adjacency Matrix and Adjacency list.

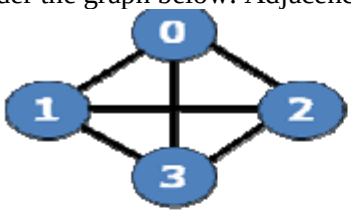
One simple way to represents a graph is Adjacency Matrix.

The adjacency Matrix A for a graph G = (V, E) with n vertices is an n x n matrix, such that

Aij = 1, if there is an edge Vi to Vj

Aij = 0, if there is no edge.

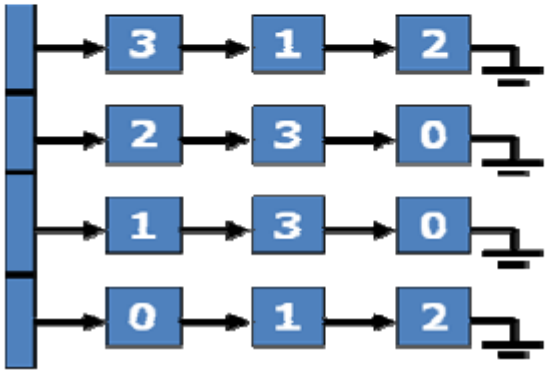
Consider the graph below: Adjacency matrix representation is



$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G1

Adjacency list representation is



Advantage

* Simple to implement.

Disadvantage

Takes O(n2) space to represents the graph

* It takes O(n2) time to solve the most of the problems.

Adjacency List Representation

In this representation, we store a graph as a linked structure. We store all vertices in a list and then for each vertex,

we have a linked list of its adjacency vertices

Disadvantage

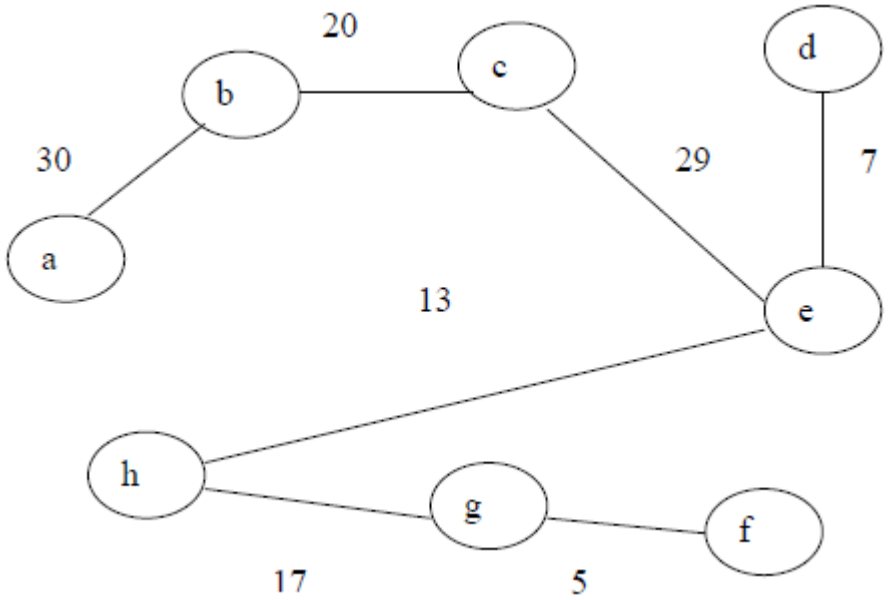
* It takes 0(n) time to determine whether there is an arc from vertex i to vertex j. Since there can be 0(n) vertices on the adjacency list for vertex i.

ii) Consider the following graph shown in figure: 1 Obtain the minimum spanning tree using Kruskal's algorithm (10) (NOV/DEC 2011)

Sort the edges in the increasing order of weight

• Include the edge to form minimum spanning tree, if it does not form a cycle.

Minimum Spanning tree



Total cost of a spanning tree = 30+20+29+7+13+5+17 = 121

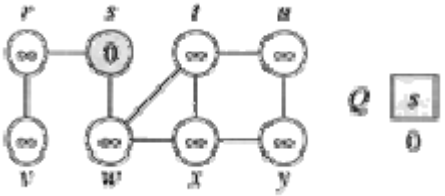
5.i)Write an algorithm for Breadth First Search on a graph and give the nodes of the graph 'G' given based on the algorithm (6) (NOV/DEC 2011) (NOV/DEC 2014)

Algorithm:

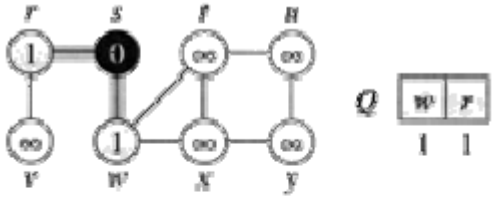
```
bfs(g)
{
list l = empty
tree t = empty
choose a starting vertex x
search(x)
while(l nonempty)
remove edge (v,w) from start of l
if w not yet visited
{
add (v,w) to t
search(w)
}
}
```

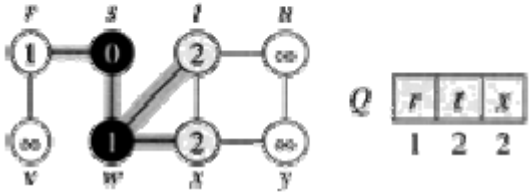
Example:

After initialization (paint every vertex white, set d[u] to infinity for each vertex u, and set the parent of every vertex to be NIL), the source vertex is discovered in line 5. Lines 8-9 initialize Q to contain just the source vertex s.

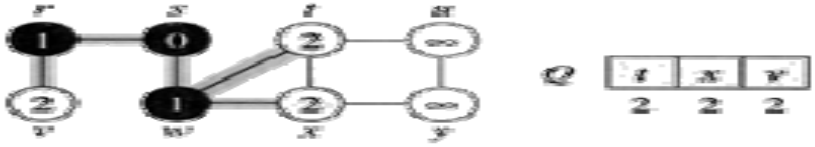


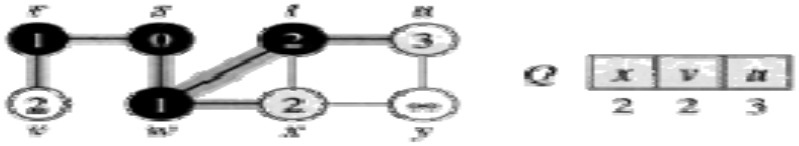
The algorithm discovers all vertices 1 edge from *s* i.e., discovered all vertices (*w* and *r*) at level 1.

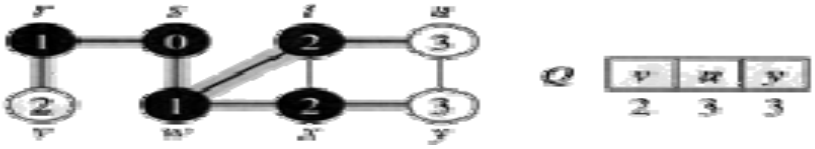




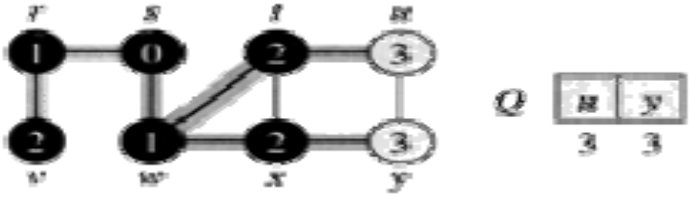
The algorithm discovers all vertices 2 edges from *s* i.e., discovered all vertices (*t*, *x*, and *v*) at level 2.

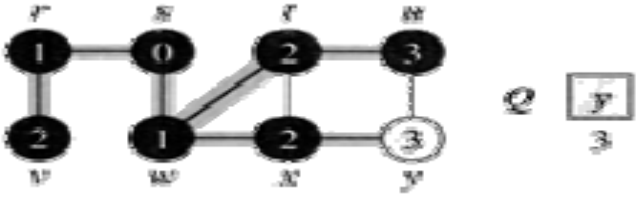




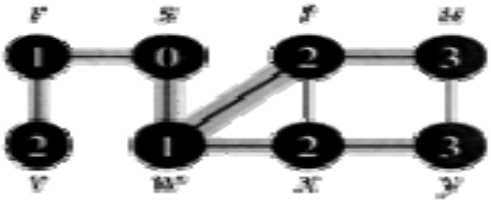


The algorithm discovers all vertices 3 edges from *s* i.e., discovered all vertices (*u* and *y*) at level 3.





The algorithm terminates when every vertex has been fully explored.



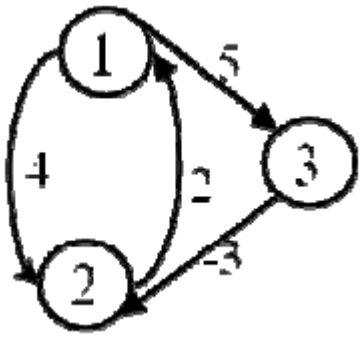
ii) Using Dijikstra's algorithm, find the shortest path from the source to all other nodes of the graph 'G' given in Figure: 3 (10) (NOV/DEC 2014)

Ans:

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	8	v3
v7	1	9	v5

6.i)Illustrate the working of Warshall' algorithm (6) (NOV/DEC 2014)

All pair shortest path
Floyd Warshall algorithm:
D *W* // initialize *D* array to *W* [ ]
P 0 // initialize P array to [0]
for *k* 1 to *n*
do for *i* 1 to *n*
do for *j* 1 to *n*
if (*D*[*i, j*] > *D*[*i, k*] + *D*[*k, j*])
then *D'*[*i, j*] *D*[*i, k*] + *D*[*k, j*]
P[*i, j*] *k*;
else *D'*[*i, j*] *D*[*i, j*]
Move D' to D.



Answer:

D^0

0	4	5
2	0	∞
∞	- 3	0

D^3

0	2	5
2	0	7
- 1	- 3	0

ii) Consider a directed acyclic graph 'D' given in Figure: 4. Sort the nodes of 'D' by applying topological sort on 'D' (10) (NOV/DEC 2014)

Vertex	1	2	3	4	5	6	7	8
A	0	0	0	0	0	0	0	0
B	1	0	0	0	0	0	0	0
C	1	1	1	0	0	0	0	0
D	2	2	1	0	0	0	0	0
E	3	3	3	2	1	0	0	0
F	1	1	1	1	1	1	0	0
G	0	0	0	0	0	0	0	0
Enque	A,G	G,B	B	C,D	D	E	F	
Deque	A	G		B	C	D	E	F

7.Find the shortest path from 'a' to 'd' using Dijkstra's algorithm in the graph (16)(NOV/DEC 2011)

Ans:

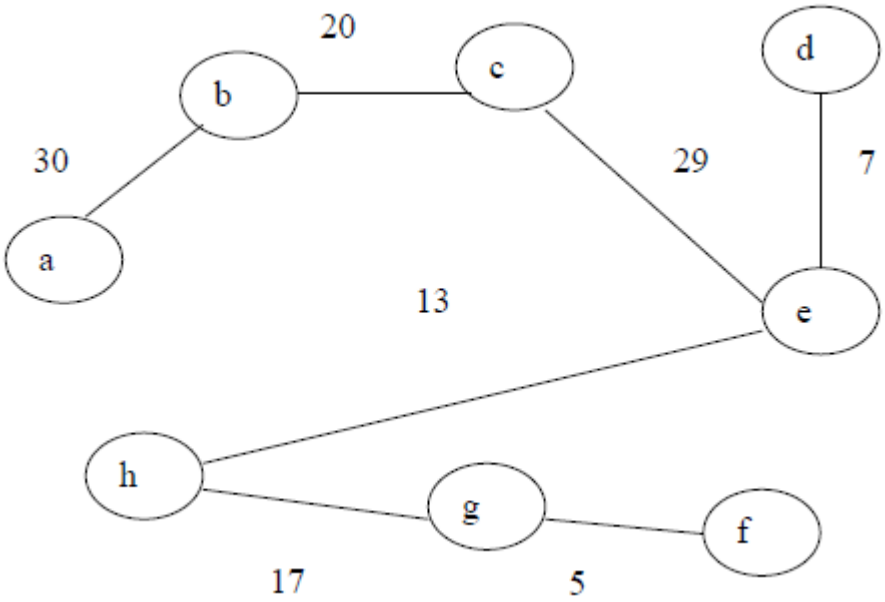
Vertex	Known	D_v	P_v
A	1	0	0
B	1	30	a
C	1	42	a
D	1	60	e
E	1	53	h
F	1	62	g
G	1	57	h
H	1	40	a

Path : a ---- h ----e------d
Cost: is computed as 40+13+7 = 60

8. Explain Kruskal's algorithm with an example (16)

Kruskal's Algorithm to Construct a Minimal Spanning Tree
Input: A weighted, connected and undirected graph $G = (V,E)$.

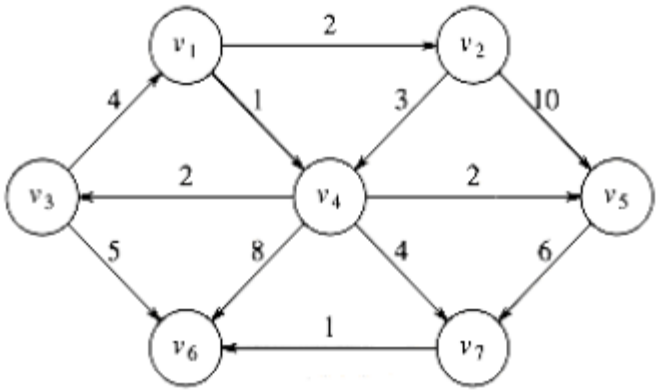
Output: A minimal spanning tree of *G*.
Step 1: *T* =_ .
Step 2: while *T* contains less than *n*-1edges **do**
Choose an edge (*v*,*w*) from *E* of the smallest weight.
Delete (*v*,*w*) from *E*.
If the adding of (*v*,*w*)does not create cycle in *T* **then**
Add (*v*,*w*) to *T*.
Else Discard (*v*,*w*).
end while
Let us consider the graph in Figure 1. The minimum spanning tree constructed is given as:



9.What is single source shortest path problem? Discuss Dijkstra's single source shortest path algorithm with an example (16)

Single source shortest path: Given a graph *G* = (*V*, *E*), we want to find a shortest path from a given ***source*** vertex *s V* to every vertex *v V*.
Algorithm:
dist[s] ←0 (distance to source vertex is zero)
for all v ∈ V–{s}
do dist[v] ←∞ (set all other distances to infinity)
S←∅ (S, the set of visited vertices is initially empty)
Q←V (Q, the queue initially contains all vertices)
while Q ≠∅ (while the queue is not empty)
do u ← mindistance(Q,dist) (select the element of Q with the min. distance)
S←S∪{u} (add u to list of visited vertices)
for all v ∈ neighbors[u]
do if dist[v] > dist[u] + w(u, v) (if new shortest path found)
then d[v] ←d[u] + w(u, v) (set new value of shortest path)
(if desired, add traceback code)
return dist

Example:



This algorithm is used to the single source shortest path from a single source to all the vertices in the graph.

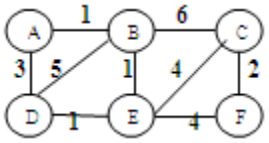
Vertex	Known	dv	pv
v1	0	0	0
v2	0	∞	0
v3	0	∞	0
v4	0	∞	0
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

Final table:

Vertex	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

10.i)Compute the minimum spanning tree for the following graph

(8)(NOV/DEC 2015)



F

Vertex X	C(v)	P(v) [parent of v]
6	-∞	-
5	2	6
4	4	6

F

Vertex X	C(v)	P(v) [parent of v]
~~6~~	-∞	-
5	2	6
4	4	6
1	1	5
2	6	5

F

	Vertex X	C(v)	P(v) [parent of v]
	~~6~~	-∞	-
	~~5~~	2	6
	4	4	6
	1	1	5
	2	6	5
remove->	5	3	4

F

Vertex X	C(v)	P(v) [parent of v]
~~6~~	-∞	-
~~5~~	2	6
~~4~~	4	6
1	1	5
2	6	5
2	5	1

replace->

F

Vertex X	C(v)	P(v) [parent of v]
~~6~~	-∞	-
~~5~~	2	6
~~4~~	4	6
~~1~~	1	5
~~2~~	5	1
3	7	2
4	8	3

remove->

F

Vertex X	C(v)	P(v) [parent of v]
~~6~~	-∞	-
~~5~~	2	6
~~4~~	4	6
~~1~~	1	5
~~2~~	5	1
~~3~~	7	2

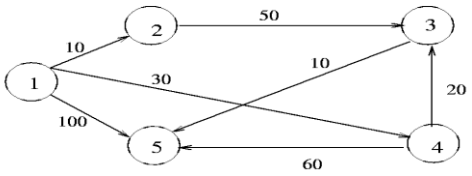
ii) Discuss any two applications of depth first search(8)(NOV/DEC 2015)

1. GPS Navigation systems: Navigation systems such as the Google Maps, which can give directions to reach from one place to another use BFS. They take your location to be the

source node and your destination as the destination node on the graph. (A city can be represented as a graph by taking landmarks as nodes and the roads as the edges that connect the nodes in the graph.) BFS is applied and the shortest route is generated which is used to give directions or real time navigation.

2. Computer Networks: Peer to peer (P2P) applications such as the torrent clients need to locate a file that the client (one who wants to download the file) is requesting.This is achieved by applying BFS on the hosts (one who supplies the file) on a network. Your computer is the host and it keeps traversing through the network to find a host for the required file (maybe your favourite movie).

11. i)Illustrate the Dijkstra's algorithm algorithm for finding the shortest path with the following graph(12)(APRIL/MAY 2016)



Initially:

$S = \{1\}$ $D[2] = 10$ $D[3] = \infty$ $D[4] = 30$ $D[5] = 100$

Iteration 1

Select w = 2, so that S = {1, 2}

$D[3] = \min(\infty, D[2] + C[2, 3]) = 60$

$D[4] = \min(30, D[2] + C[2, 4]) = 30$

$D[5] = \min(100, D[2] + C[2, 5]) = 100$

Iteration 2

Select w = 4, so that S = {1, 2, 4}

$D[3] = \min(60, D[4] + C[4, 3]) = 50$

$D[5] = \min(100, D[4] + C[4, 5]) = 90$

Iteration 3

Select w = 3, so that S = {1, 2, 4, 3}

$D[5] = \min(90, D[3] + C[3, 5]) = 60$

teration 4

Select w = 5, so that S = {1, 2, 4, 3, 5}

$D[2] = 10$

$D[3] = 50$

$D[4] = 30$

$D[5] = 60$

ii) Illustrate the comparison of Floyds algorithm with dijkstra's algorithm(4)

Dijkstra's Algorithm:

• Finds shortest path from a given startNode to all other nodes reachable from it in a digraph.

• Assumes that each link cost $c(x, y) \geq 0$.

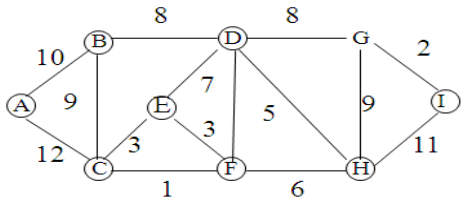
• Complexity: $O(N^2)$, N = #(nodes in the digraph)

Floyd's Algorithm:

• Finds a shortest-path for all node-pairs (x, y).
• We can have one or more links of negative cost, c(x, y)

- Complexity: O(N 3), where N = #(nodes in digraph).

12.Find the minimum spanning tree for the given graph using both Prims and kruskal's algorithm and write that algorithms(8+8) (APRIL/MAY 2016)



- MST is a SPANNING Tree
  - Nodes of MST = Nodes of G
  - MST contains a path between any two nodes
- MST is a MINIMUM Spanning Tree
  - Sum of edges is a minimum
- MST may not be unique
- Initialize
  - M = {}
  - vertex sets: {A},{B},{C},{D},{E},{F},{G},{H},{I}
  - edge list, sorted by weight: 1 - CF, 2- GI, 3- EF, 3 -CE, 5- DH, 6- FH, 7- DE, 8-BD, 9-BC, 9-GH, 10-AB, 11-HI, 12-AC
- (u, v) = (C, F)
  - C and F are in different vertex sets (ie {C} and {F}
  - Add CF to M: M = {CF} (omit the comma from the edge)
  - Merge {C} and {F}: {A},{B},{C,F},{D},{E},{G},{H},{I}
- (u, v) = (G, I)
  - G and I are in different vertex sets (ie {G} and {I}
  - Add GI to M: M = {CF, GI}
  - Merge {G} and {I}: {A},{B},{C,F},{D},{E},{G,I},{H}
- (u, v) = (E, F)
  - E and F are in different vertex sets (ie {E} and {C,F}
  - Add EF to M: M = {CF, EF, GI}
  - Merge {E} and {F}: {A},{B},{C,E,F},{D},{G,I},{H}
- (u, v) = (C, E)
  - C and E are both in {C,E,F}: don't add CE since it would create a cycle

A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight

```
struct Edge
{
   int src, dest, weight;
};

struct Graph
{
      int V, E;
```

```
    struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

struct subset
{
    int parent;
    int rank;
};

int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);


    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

        else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
```

```
    return a1->weight > b1->weight;
}

void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V];  // Tnis will store the resultant MST
    int e = 0;  // An index variable, used for result[]
    int i = 0;  // An index variable, used for sorted edges

    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

       struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

        for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

        while (e < V - 1)
    {

        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
                                          result[i].weight);
    return;
}

int main()
{

    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

     graph->edge[0].src = 0;
```

```
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);
     return 0;
}
```