UNIT 1
PART-A

1.* Distinguish between methods and messages *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.

- 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

Static variable are normally used to maintain values common to the entire class.
Feature:
• It is initialized to zero when the first object is created. No other initialization is permitted
• only one copy of that member is created for the entire 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?

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.

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?

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 ints, 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 advantageous of default argument?

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 are 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?

- 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 like built-in types of the programming language.

13. What are objects?

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 datatype, 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 statesmen.

Eg:

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

20. What do you mean by dynamic initialization of variables?

C++ permits initialization of the variables at run-time. This is referred to as dynamic initialization of variables.

In C++ ,a variable can be initialized at run-time using expressions at the place of declaration as,

……..

…......

int n =strlen(string);
……..
float area=3.14*rad*rad;
Thus declaration and initialization is done simultaneously at the place where the variable is used for the first time.

21. **What are reference variable?**
A reference variable provides an alias(alternative name) for a previously defined variable.
sum total For example , if make the variable a reference to the variable , then sum and total can be used interchancheably to represent that variable.
Syntax :
Data-type &reference-name = variable-name
Eg:
float total = 100;
float sum = total;

22. **What is member-dereferencing operator?**
C++ permits to access the class members through pointers. It provides three pointer-to-member operators for this purpose,
: :* To declare a pointer to a member of a class.
* To access a member using object name and a pointer to the member
->* To access a member using a pointer to the object and a pointer to that member.

23. **What is an inline function ?**
An inline function is a function that is expanded in line when it is invoked. That is compiler replaces the function call with the corresponding function code.
The inline functions are defined as Inline function-header
{
function body
}

24. **Define const member**
If a member function does not alter any data in the class, then we may declare it as const member function as
Void mul(int ,int)const;

PART-B

1. **i) Highlight the features of object oriented programming language. (8)**

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 in 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 feature
- Inheritance and class hierarchy – Reusability and extension of existing classes
- Polymorphism – Multiple definitions for a single name - 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 classes
- Message passing – Objects communicates through invoking 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)

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;
cout<<"\n b = ";
cin>>b;
swap(a,b);
cout<<"\n Enter two Characters : \n";
cout<<" s1 = ";
cin>>s1;
cout<<"\n s2 = ";
cin>>s2;
swap(s1,s2);
cout<<"\n After Swapping \n";
cout<<" \n Integers i = "<<i<<"\t j =  "<<j;
cout<<" \n Float Numbers a= "<<a<<"\t b =  "<<b;
cout<<" \n Characters s1 = "<<s1<<"\t s2 =  "<<s2;
getch();
}
```

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.**

```
#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.**

- 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)**

```
#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)

```
#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 parameters 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 what are 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 is created. No other initialization is permitted
• only one copy of that member is created for the entire 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.**

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. **Explain about various access specifiers with examples.**

- Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type.
- The access restriction to the class members is specified by the labelled **public, private,** and **protected** sections within the class body.
- The keywords public, private, and protected are called access specifiers.
- A class can have multiple public, protected, or private labeled sections.
- Each section remains in effect until either another section label or the closing right brace of the class body is seen.
- The default access for members and classes is private.

```
class Base {
   public:
  // public members go here
   protected:
 // protected members go here
   private:
  // private members go here
 };
```

The public members:
A public member is accessible from anywhere outside the class but within a program.

```
#include <iostream>
class Line
{
  public:
      double length;
      void setLength( double len );
      double getLength( void );
};
 double Line::getLength(void)
{
   return length ;
}
void Line::setLength( double len )
{
   length = len;
}
int main( )
{
  Line line;
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;
  line.length = 10.0; // OK: because length is public
  cout << "Length of line : " << line.length <<endl;
  return 0;
}
```

Output

```
Length of line: 6
Length of line: 10
```

The private members:

- A **private** member variable or function cannot be accessed, or even viewed from outside the class.
- Only the class and friend functions can access private members.
- By default all the members of a class would be private

```
#include <iostream>

class Box
{
  double width;
  public:
     double length;
     void setWidth( double wid );
     double getWidth( void );
};
double Box::getWidth(void)
{
   return width ;
}
 void Box::setWidth( double wid )
{
   width = wid;
}
int main( )
{
  Box box;
  box.length = 10.0; // OK: because length is public
  cout << "Length of box : " << box.length <<endl;
 // box.width = 10.0; // Error: because width is private
```

```
  box.setWidth(10.0);
  cout << "Width of box : " << box.getWidth() <<endl;
   return 0;
}
```

```
Output:
Length of box: 10
Width of box: 10
```

The protected members:
A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

```
#include <iostream>
class Box
{
  protected:
     double width;
};
class SmallBox:Box
{
  public:
     void setSmallWidth( double wid );
     double getSmallWidth( void );
};
double SmallBox::getSmallWidth(void)
{
   return width ;
}
void SmallBox::setSmallWidth( double wid )
{
   width = wid;
}
int main( )
{
  SmallBox box;
  box.setSmallWidth(5.0);
  cout << "Width of box : "<< box.getSmallWidth() << endl;
   return 0;
}
```

```
Output:
Width of box: 5
```

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

- auto
- register
- static
- extern
- mutable

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

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

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

The register Storage Class

- The **register** storage class is used to define local variables that should be stored in a register instead of RAM.
- 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 2
PART-A

1. What is a constructor? How do we invoke a constructor function?

- 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?

- 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

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?**
   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. **Write the prototype for a typical pure virtual function?**
   Pure Virtual Function:
7. 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 : same example can re-written
   ```
   class base
   {
   public:
   virtual void show()=0; //pure virtual function
   };
   ```
8. **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.
9. **What is the need for initialization of objects using constructors?**
   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.
10. **What is destructor?**
    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
11. **What is inheritance? What are the types of inheritance?**
    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

12. **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'.
13. **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
14. **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);
15. **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
16. **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);
}
```

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

- T hey 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. 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.

20. Write at least four rules for Operator overloading.

- Only the existing operators can be overloaded.
- The overloaded operator must have at least one operand that is of user defined data type.
- The basic meaning of the operator should not be changed.

• Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.

21. List out the operators that cannot be overloaded using Friend function.

• Assignment operator =
• Function call operator ( )
• Subscripting operator [ ]
• Class member access operator

22. Explain basic to class type conversion with an example.

Conversion from basic data type to class type can be done in destination class.
Using constructors does it. Constructor takes a single argument whose type is to be converted.
Eg: Converting int type to class type

```
class time
{
int hrs,mins;
public:
…………
Time ( int t) //constructor
{
hours= t/60 ; //t in minutes
mins =t % 60;
} };
```

Constructor will be called automatically while creating objects so that this conversion is done automatically.

23. Explain class to basic type conversion with an example.

Using Type Casting operator, conversion from class to basic type conversion can be done. It is done in the source class itself.

```
Eg: vector : : operator double( )
{
double sum=0;
for(int I=0;I<size;I++)
sum=sum+v[ i ] *u[ i ] ;
return sqrt ( sum ) ;  }
```

This function converts a vector to the corresponding scalar magnitude.

24. Explain one class to another class conversion with an example.

Conversion from one class type to another is the combination of class to basic and basic to class type conversion. Here constructor is used in destination class and casting operator function is used in source class.
Eg: objX = objY
objX is the object of class X and objY is an object of class Y. The class Y type data is converted into class X type data and the converted value is assigned to the obj X. Here class Y is the source class and class X is the destination class.

PART-B

1. **Write brief notes on Friend function and show how Modifying a Class's private Data with a Friend** Function. **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.Write a program in C++ that uses functions to read two strings ,overload the operators + to perform concatenation of two strings and overload the operator = = to check for the equality of the strings. #include<conio.h>

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

class string {
public:
char *s;
int size;
void getstring(char *str)
{
size = strlen(str);
s = new char[size];
strcpy(s,str);
}

void operator+(string);
};
void friend operator==(String, String);

void operator==(String ob1, String  ob2)
{
if(strcmp(ob1.s,ob2.s)==0)
cout<<"\nStrings are Equal";
else
cout<<"\nStrings are not Equal";
}

void string::operator+(string ob)
{
size = size+ob.size;
s = new char[size];
strcat(s,ob.s);
cout<<"\nConcatnated String is: "<<s;
}
void main()
{
string ob1, ob2;
char *string1, *string2;
clrscr();

cout<<"\nEnter First String:";
cin>>string1;

ob1.getstring(string1);

cout<<"\nEnter Second String:";
cin>>string2;

ob2.getstring(string2);
//Calling + operator to Join/Concatenate strings
ob1+ob2;
//Call Equality Operator
ob1==ob2
getch();
}Enter the First String C++
Enter the First String C++
Concatnated String is:C++C++
Strings are Equal
```

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

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)

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

< Classname > = (value, value,...........);

· By calling the constructor implicitly

(value, value, ...);

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 base class? When do we make a class virtual

Virtual Base Class:

When a class is declared as virtual c++ takes care to see that only copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.Consider following example:

Class make a virtual:

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as *virtual* to ensure that Band C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects of class L, one through class B1and another through class B2. You can use the keyword virtual in front of the base class specifiers in the *base lists* of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

For example:

```
class A
{
   public:
       int i;
};
class B : virtual public A
{
     public:
            int j;
};
class C: virtual public A
{
      public:
          int k;
};
class D: public B, public C
{
public:
     int sum;
};
int main()
{
        D ob;
        ob.i = 10; //unambiguous since only one copy of i is inherited.
        ob.j = 20;
        ob.k = 30;
        ob.sum = ob.i + ob.j + ob.k;
        cout << "Value of i is : "<< ob.i<<"\n";
        cout << "Value of j is : "<< ob.j<<"\n"; cout << "Value of k is :"<< ob.k<<"\n";
        cout << "Sum is : "<< ob.sum <<"\n";
return 0;
}.
```

ii) Explain the concept behind dynamic casting and cross casting

Dynamic-cast Typecast

Dynamic casts are only available in C++ and only make sense when applied to members of a class hierarchy ("polymorphic types"). Dynamic casts can be used to safely cast a superclass pointer (or reference) into a pointer (or reference) to a subclass in a class hierarchy. If the cast is invalid because the the real type of the object pointed to is not the type of the desired subclass, the dynamic will fail gracefully.

Pointer dynamic cast

When casting a pointer, if the cast fails, the cast returns NULL. This provides a quick method of determining if a given object of a particular dynamic type.
The syntax for a pointer dynamic cast is
<type> *p_subclass = dynamic_cast<<*type*> *>(p_obj);

Reference dynamic cast

When casting a reference, it is not possible to return a NULL pointer to indicate failure; a dynamic cast of a reference variable will throw the exception std::bad_cast (from the <typeinfo> header).
<type> subclass = dynamic_cast<<*type*> &>(ref_obj);

```
class A
{public:
int i;
virtual void show(){}
};


class B:public A
{
public:
int j;
virtual void show()  { cout<<"B";  }
};

int main()
{
A* ptr=new B; // UPCASTING
B* ptrb;
ptrb=dynamic_cast(ptr); //DOWNCASTING
ptrb->show(); // upcasting helped in implementing interface
}
```

Cross casting refers to casting from derived class to proper base class when there are multiple base classes in the case of multiple-inheritance. When a multiple derived class object is pointed to by one of its base class pointers, casting from one base class pointer into another base class pointer is known as cross casting.

5 i) What is virtual function? When do we make a virtual function "pure"?

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 : same example can re-written

```
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.

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;
}
```

6 i) What are the rules to be followed in function overloading? (4)

Rules in function overloading

- The overloaded function must differ either by the arity or data types.
- The same function name is used for various instances of function call.

It is a classification of static polymorphism in which a function call is resolved using the 'best match technique', i.e., the function is resolved depending upon the argument list. Method overloading is usually associated with statically-typed programming languages which enforce type checking in function calls. When overloading a method, you are really just making a number of different methods that happen to have the same name. It is resolved at compile time which of these methods are used.
Method overloading should not be confused with forms of polymorphism where the correct method is chosen at runtime, e.g. through virtual functions, instead of statically.

**(ii) Write a C++ program that can take either two integers or two floating point numbers and
outputs the smallest number using function overloading. (12)**

```
#include <iostream>

using namespace std;
```

```
/* Function arguments are of different data type */

long add(long, long);
float add(float, float);

int main()
{
long a, b, x;
float c, d, y;

cout << "Enter two integers\n";
cin >> a >> b;

x = add(a, b);

cout << "Sum of integers: " << x << endl;

cout << "Enter two floating point numbers\n";
cin >> c >> d;

y = add(c, d);

cout << "Sum of floats: " << y << endl;

return 0;
}

long add(long x, long y)
{
long sum;

sum = x + y;

return sum;
}

float add(float x, float y)
{
float sum;

sum = x + y;

return
```

7 (i) List the special characteristics of friend function. (6)

A friend function is not in the scope of the class n which it has been declared as friend.
It cannot be called using the object of that class.
It can be invoked like a normal function without any object.
Unlike member functions, it cannot use the member names directly.
It can be declared in public or private part without affecting its meaning.
Usually, it has objects as arguments.

(ii) Write a C++ program to find the area of the square, rectangle, circle using function overloading. (10)

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

```
const float pi=3.14;
float area(float n,float b,float h)
{
float ar;
ar=n*b*h;
return ar;
}
float area(float r)
{
float ar;
ar=pi*r*r;
return ar;
}
float area(float l,float b)
{
float ar;
ar=l*b;
return ar;
}
void main()
{
float b,h,r,l;
float result;
clrscr();
cout<<"\nEnter the Base & Hieght of Triangle: \n";
cin>>b>>h;
result=area(0.5,b,h);
cout<<"\nArea of Triangle: "<<result<<endl;
cout<<"\nEnter the Radius of Circle: \n";
cin>>r;
result=area(r);
cout<<"\nArea of Circle: "<<result<<endl;
cout<<"\nEnter the Length & Bredth of Rectangle: \n";
cin>>l>>b;
result=area(l,b);
cout<<"\nArea of Rectangle: "<<result<<endl;
getch();
```

8. Explain the different types of constructors with suitable examples.

Parameterized constructors

Constructors that can take arguments are termed as parameterized constructors. The number of arguments can be greater or equal to one(1). For example:

```
>
class Example
{
int x, y;
public:
Example();
Example(int a, int b);                    //parameterized constructor
};
Example :: Example()
{
}
Example :: Example(int a, int b)
{
```

```
x = a;
y = b;
}
```

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.

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

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).[1]

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
}
```

9.(i) Write the rules for overloading the operators. (6)

1) Only built-in operators can be overloaded. New operators cannot be created.
2) Arity of the operators cannot be changed.
3) Precedence and associativity of the operators cannot be changed.

4) Overloaded operators cannot have default arguments except the function call operator () which can have default arguments.
5) Operators cannot be overloaded for built in types only. At least one operand must be used defined type.
6) Assignment (=), subscript ([]), function call ("()"), and member selection (->) operators must be defined as member functions
7) Except the operators specified in point 6, all other operators can be either member functions or a non member functions.
8) Some operators like (assignment)=, (address)& and comma (,) are by default overloaded

(ii) Write a C++ program to add two complex numbers using operator overloading.(10)

10.Explain copy constructor with suitable C++ coding.

```
#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();
            }
```

```
            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
```

11. Write short notes on:

i)Type conversions

Implicit conversion

Implicit conversions are automatically performed when a value is copied to a compatible type. For example:

short a=2000;
int b;
b=a;

Keyword explicit

On a function call, C++ allows one implicit conversion to happen for each argument. This may be somev problematic for classes, because it is not always what is intended. For example, if we add the following function to the last example:

void fn (B arg) {}

This function takes an argument of type B, but it could as well be called with an object of type A as argument:

fn (foo);

ii)Explicit constructor

In explicit copy constructor is one that is declared explicit by using the **explicit** keyword.
For example:

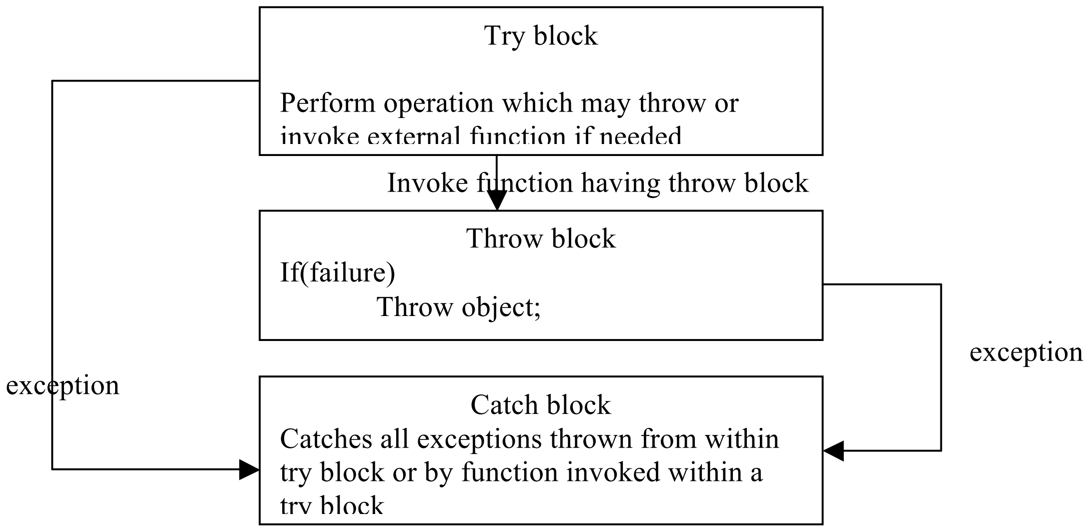
```
explicit X(const X& copy_from_me);

#include <iostream>

class Person {
public:
int age;

explicit Person(int a)
: age(a)
{
}
};

int main()
{
Person timmy(10);
Person sally(15);

Person timmy_clone = timmy;
std::cout << timmy.age << " " << sally.age << " " << timmy_clone.age << std::endl;
timmy.age = 23;
std::cout << timmy.age << " " << sally.age << " " << timmy_clone.age << std::endl;
}
```

UNIT 3
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?**
   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. **Illustrate the exception handling mechanism**


Try block
Perform operation which may throw or invoke external function if needed
Invoke function having throw block
Throw block
If(failure)
Throw object;
exception
exception
Catch block
Catches all exceptions thrown from within try block or by function invoked within a try block

5. **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

6. **What is throw() ? What is its use ?**
   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

7. **What is meant by abstract class ?**

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.

8. **What is namespace ?**

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?**

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**

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?**

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)throw 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

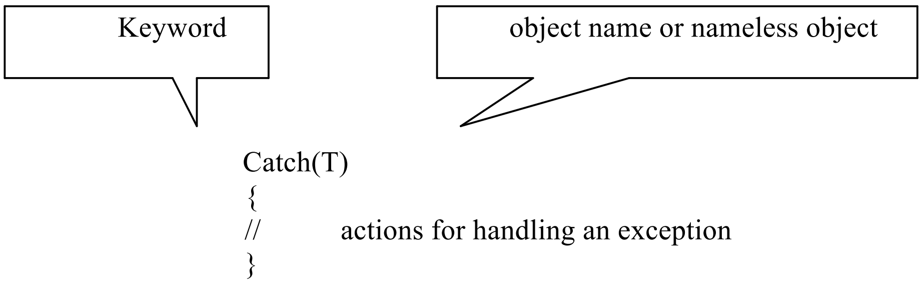
Keyword

```
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

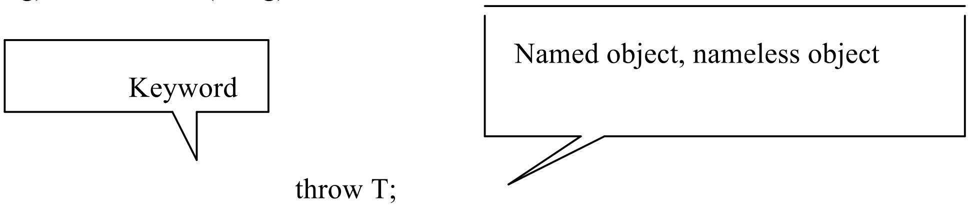
The exception handler is indicated by the catch keyword.It must be used immediately after the statements marked by the try keyword.The catch handler can also occur immediately after another catch.Each handler will only evaluate an exception that matches,or can be covered to the type specified in its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



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

16. 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 argyment arg)used in throw(T arg)



```
throw T;
```

17. Write the syntax of function template

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

18. Write the syntax of class template

```
template <class T1,class T2,…>
    class classname
      {
    T1 data1;
    ….
    //functions of template arguments T1,T2,….
    void func1(T1 a,T2 &b);
    …
    T func2(T2 *x,T2 *y);
    };
```

19. What is STL?

The **Standard Template Library** (**STL**) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called *algorithms*, *containers*, *functional*, and *iterators*. The STL provides a ready-made set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.The STL achieves its results through the use of templates.

20. Define Allocators

In C++ computer programming, **allocators** are an important component of the C++ Standard Library. The standard library provides several data structures, such as list and set, commonly referred to as containers. A common trait among these containers is their ability to change size during the execution of the program. To achieve this, some form of dynamic memory allocation is usually required. Allocators handle all the requests for allocation and deallocation of memory for a given container. The C++ Standard Library provides general-purpose allocators that are used by default, however, custom allocators may also be supplied by the programmer.

21. What is iterator?

An iterator is any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators The most obvious form of iterator is a pointer: A pointer can point to elements in an array, and can iterate through them using the increment operator (++). But other kinds of iterators are possible.

22. What is the use of function adaptors?

A function adaptor is an instance of a class that adapts a global or member function so that the function can be used as a function object. A function adaptor may also be used to alter the behavior of a function or function object. Each function adaptor provides a constructor that takes a global or member function. The adaptor also provides a parenthesis operator that forwards its call to that associated global or member function.

PART-B

1. (i) write the syntax for member function template.

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>
            classclassname
            {
                    private:
                            __________
                            __________
                    public:
                            __________
                            __________
            };
```

Program :

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

```
template<class T>
class sample
{
private:
T value,value1,value2;
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 rethrowing of an exception is done. (4)**

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 MyHandler\n”;
        throw; //rethrow char* out of function
    }
}
int main()
{
      cout<< “Main start”;
      try
```

```
        {
             MyHandler();
        }
        catch(const char*)
        {
            cout<<"Caught exception inside Main\n";
        }
            cout<< "Main end";
        return 0;
}
```

O/p:

```
Main 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<conio.h>
void test(int x)
{
  try
  {
        if(x>0)
          throw 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 for finding the minimum value contained in an array. (8)

```
#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) List the advantages of exception handling mechanisms. (4)

separating Error Handling Code from ``regular'' one

provides a way to separate the details of what to do when something out-of-the-ordinary happens from the normal logical flow of the program code;

propagating Errors Up the Call Stack

lets the corrective action to be taken at a higher level. This allows the corrective action to be taken in the method that calling that one where an error occurs;

grouping Error Types and Error Differentiation

allows to create similar hierarchical structure for exception handling so groups they in logical way.

(ii) Write a c++ program for the following :

1) **A function to read two double type numbers from keyboard.**
2) **A function to calculate the division of these two numbers.**
3) **A try block to throw an exception when a wrong type of data is keyed in.**
4) **A try block to detect and throw an exception if the condition "divide – by – zero " occurs**
5) **Appropriate catch block to handle the exceptions thrown. (12)**

```
      Class divbyzero
    {
            Private :
            Double a,b;
            Public :
            Void getdata(double e,double f)
            {
                    Cout<<"pls enter the two value : "<<endl;
                    Cin>>e>>f;
            }
```

```
                    Void divv()
                    {
                    Try{
                              If(f==0)
                                        Throw(f);
                              Else
                                        Cout<<"The Quotient:"<<e/f<<endl;
                    }catch(double e)
                    {
                              Cout<<"The Divisible by Zero error will occur"<<endl;
                    }
               }
 Void main()
{
        Divbyzeros,p;
        s.getdata(2.3,4.5);
        s.divv();
        p.getdata(4.5,0.0);
        p.divv();
}
```

5. Discuss the need for exception with try, catch and throw keywords.

Exceptions are of 2 kinds

Synchronous Exception:

Out of rage

Over flow

Asynchronous Exception: Error that are caused bycauses beyond the control of the program

Keyboard interrupts

In C++ only synchronous exception can behandled.

Exception handling mechanism

Find the problem (Hit the exception)

Inform that an error has occurred (Throw the exception)

Receive the error information (Catch the exception)

Take corrective action (handle the exception)

The keyword **try** is used to preface a block ofstatements which may generate exceptions.When an exception is detected, it is thrown using a**throw** statement in the try block. A **catch** block defined by the keyword 'catch'catches the exception and handles it appropriately.The catch block that catches an exception mustimmediately follow the try block that throws theexception.

6. Explain in detail about abstract class with suitable example.

An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

abstract void moveTo(double deltaX, double deltaY);

If a class includes abstract methods, then the class itself *must* be declared abstract, as in:

```
public abstract class GraphicObject {
  // declare fields
  // declare nonabstract methods
abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

7. (i) Explain any two sequence containers supported by Standard Template Library. (4)

Sequences

Common properties of all sequence containers:
constructors
Fill constructor Container(n,val) fills container with n copies of val.
Default fill constructor Container(n) fills container with n default constructed values.
Range constructor Container(i,j) fills container with the contents of the iterator range [i,j).
assign
fill assignment assign(n,val)
range assignment assign(i,j)
old elements are assigned to or destroyed
insert
insert(p,val) inserts val just before the position pointed by iterator p.
insert(p,n,val) inserts n copies.
insert(p,i,j) inserts the contents of range [i,j).
erase
erase(p) erases the element pointed by iterator p.
erase(p,q) erases the range [p,q)
returns iterator to the position immediately following the erased element(s)
clear() erases all

vector

vector should be used by default as the (sequence) container:
It is more (space and time) efficient than other STL containers.
It is more convenient and safer than primitive array.
automatic memory management
rich interface
Properties of vector in addition to sequences:v[i], at(i)
v.at(i) checks that 0<=i<v.size()
front(), back()
return reference to the first and last element (not beyond last)
push_back(val)
insertsval to the end
pop_back() removes
removes the last element and returns it
resize
change the number of elements in the vector
resize(n) makes n the size; fills with default values if necessary
resize(n,val) fills with val if necessary
capacity(), reserve(n) (see below)

Memory management

The elements are stored into a contiguous memory area on the heap.
capacity() is the number of elements that fit into the area.
size() is the actual number of elements. The remainder of the area is unused (raw memory).
reserve(n) increases the capacity to n without changing the size.
The capacity is increased automatically if needed due to insertions.
Capacity increase may cause copying of all elements.
A larger memory area is obtained and elements are copied there.
Capacity increase by an insertion doubles the capacity to achieve *amortized constant time.*
Capacity never decreases.
Memory is not released.
But the following gets rid of all extra capacity/memory:

```
vector<int> v;
...
vector<int>(v).swap(v);  // copy and swap
```

Use &v[0] to obtain a pointer to the memory area.

May be needed as an argument to non-STL functions.

```
vector<char> v(12);
strcpy(&v[0], "hello world");
```

Limitations of vector

Insertions and deletions in the beginning or in the middle are slow.
Requires moving other elements.
Prefer push_back() and pop_back().
Insert or erase many elements at a time by using the range forms of insert and erase.
Insertions and deletions *invalidate* all iterators, and pointers and references to the elements.

```
vector<int> v;
...
vector<int> b = v.begin();
v. push_back(x);
find(b, v.end());  // error: b is invalid
```

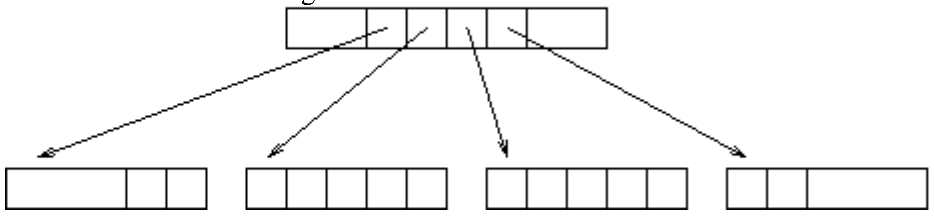
deque

deque stands for double-ended queue. It is much like vector.
Differences to vector
Insertion and deletion in the beginning in (amortized) constant time.
push_front, pop_front
Slower element access and iterators.
No capacity() or reserve(n) but also less need for them.
Insertions and deletions to the beginning and end do not invalidate pointers and references to other elements.
But iterators may be invalidated.

```
deque<int> d(5,1);
deque<int>::iterator i = d.begin() + 2;
int* p = &*i;
d.push_front(2);
int x = *p;  // OK
int y = *i;  // error: i may be invalid
```

Memory management

deque stores the elements something like this:



Element access and iterators are more complicated.
Fast insertions and deletions to the beginning.
Handles size changes gracefully: - Capacity increases or decreases one block at a time.
Memory area is not contiguous.

(ii) Write a C++ Program using lists from STL to input 10 numbers and store them in a list. From this list, create wo more lists, one containing the even numbers, and the other containing the odd numbers. Output all the three lists(12).

```
int main()
{
        List <int>li,oli,eli;
        IntI,x,temp;
          list<int>::iterator Iter;

        For(i=0;i<10;i++)
        {
                Cin>>temp;
```

```
                    li.push_back(temp);
          }
            cout<<" list of data: ";
  for(Iter = li.begin(); Iter != li.end(); Iter++)
  cout<<" "<<*Iter;
  cout<<endl;

          For(i=0;i<10;i++)
          {
                    X=li.pop_back();
                    If(x%2==0)
                              Eli.push_back(x);
                    Else
                              Oli.push_back(x);
          }
          cout<<"odd list data: ";
  for(Iter = oli.begin(); Iter != oli.end(); Iter++)
  cout<<" "<<*Iter;
  cout<<endl;
  cout<<"even list data: ";
  for(Iter = eli.begin(); Iter != eli.end(); Iter++)
  cout<<" "<<*Iter;
  cout<<endl;

  }
```

8. Explain the overloading of iterators with suitable example.

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

int main(void)
{
int i;
// vector container
vector<int>vec;
// vector iterator
vector<int>::iterator veciter;

// push/insert data
for(i = 10; i<=15; ++i)
vec.push_back(i);

// print the data
cout<<"The vec vector data: ";
for(veciter = vec.begin(); veciter != vec.end(); veciter++)
cout<<*veciter<<" ";
cout<<endl;

vector<int>::reverse_iterator rveciter1 = vec.rbegin();
cout<<"\nThe iterator rveciter1 initially points to the first element in the reversed sequence:
"<<*rveciter1<<endl;
cout<<"\nOperation: rveciter1 = diff + rveciter1"<<endl;
vector<int>::difference_type diff = 4;
rveciter1 = diff + rveciter1;
cout<<"The iterator rveciter1 now points to the fifth element in the reversed sequence:
"<<*rveciter1<<endl;
return 0;
}
```

Output examples:

```
The vec vector data: 10 11 12 13 14 15
The iterator rveciter1 initially points to the first element in the reversed sequence: 15
Operation: rveciter1 = diff + rveciter1
The iterator rveciter1 now points to the fifth element in the reversed sequence: 11
Press any key to continue . . .
```

9. Write short notes for the following:

(i)Container

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

Two container :

i) Sequence container – Vector, List, Deque
ii) 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 :

i) Small in number
ii) Generality
iii) Efficient, Tested, Debugged and standardized
iv) Portability and reusability

(ii) Function adaptor

In the STL, adapters are used to build function objects out of ordinary functions or class methods. In fact, most of the predefined STL algorithms don't care about their argument being a function or a function object. However some standard algorithms, like for example bind() which we will discuss in more detail in the next section, rely on the fact that their argument is a function object. In this case a ``pointer to function adapter'' can be used to transform a function pointer into an function object.
Since algorithms invoke their function argument by simply calling the arguments function operator, it would not be possible for algorithms to call a member function for all their operands. In this case, a member function adapter can be used which transforms a member function into a general function object. Let's have a look at an example adapted from [Strou97](18.4.4.2):

```
voiddraw_all(list<Shape>&ls) {
for_each(ls.begin(), ls.end(), mem_fun_ref(&Shape::draw));
}
```

Provided that Shape is a class which provides a draw method, the function draw_all will call draw for every Shape object inside its list argument. The adapter mem_fun_ref which is defined in <functional> as follows [ANSI-CPP](20.3.8#7):

```
template<class R, class T>
mem_fun_ref_t<R, T>mem_fun_ref(R (T::*pm)());
```

transforms its argument, a pointer to a member function without arguments, into a function object of type mem_fun_ref_t. Note however that mem_fun_ref_t is a unary function and its function operator takes one argument, namely the object for which the method will be called. The definition of mem_fun_ref_t is [ANSI-CPP](20.3.8#5):

```
template<class R, class T>
structmem_fun_ref_t : publicunary_function<T, R> {
explicitmem_fun_ref_t(R (T::*pm)());
```

```
  R operator() (T &x) const;
};
```

Adaptable function objects require in addition to regular function objects some local types that describe the result type and the argument types. A function pointer can be a valid model for a function object, but it cannot be a valid model of an adaptable function object.

Concept	**Refinement of**	**Syntactic requirements, model T**
Adaptable Generator	Generator	T::result_type
Adaptable Unary Function	Unary Function	T::result_type, T::argument_type
Adaptable Binary Function	Binary Function	T::result_type, T::first_argument_type, T::second_argument_type
Adaptable Predicate	Predicate, Adaptable Unary Function	
Adaptable Binary Predicate	Binary Predicate, Adaptable Binary Function	

Small helper classes help to define adaptable function objects easily. For example, our function object equals from above could be derived from std::binary_function to declare the appropriate types.

```
#include <functional>

template<class T>
struct equals : public std::binary_function<T,T,bool> {
bool operator()( const T& a, const T& b) { return a == b; }
};
```

The definition of binary_function in the STL is as follows:

```
template<class Arg1, class Arg2, class Result>
structbinary_function {
typedef Arg1   first_argument_type;
typedef Arg2   second_argument_type;
typedef Result result_type;
};
```

Adaptable function objects can be used with adaptors to compose function objects. The adaptors need the annotated type information to declare proper function signatures etc. An examples is the negaterunary_negate that takes an unary predicate and is itself a model for an unary predicate, but with negated boolean values.

```
template<class Predicate>
classunary_negate
   : publicunary_function<typename Predicate::argument_type, bool> {
protected:
   Predicate pred;
public:
explicitunary_negate( const Predicate& x) : pred(x) {}
bool operator()(consttypename Predicate::argument_type& x) const {
return !pred(x);
   }
};
```

The function adaptors are paired with function templates for easy creation. The idea is that the function template derives the type for the template argument automatically (because of the matching types).

```
template<class Predicate>
inlineunary_negate< Predicate>
not1(const Predicate&pred) {
returnunary_negate< Predicate>( pred);
}
```

(iii) Allocators

Default allocator

Allocators are classes that define memory models to be used by some parts of the Standard Library, and most specifically, by STL containers.

This section describes the *default allocator template*allocator (lowercase). This is the allocator that all standard containers will use if their last (and optional) template parameter is not specified, and is the only predefined allocator in the standard library.

Other allocators may be defined. Any class having the same members as this *default allocator* and following its minimum requirements can be used as an allocator with the standard containers.

Technically, a memory model described by allocators might be specialized for each type of object to be allocated and even may store local data for each container they work with. Although this does not happen with the default allocator.

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

Member functions

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

Template specializations

Header <memory> provides a specialization of allocator for the void type, defined as:

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

};

UNIT 4
PART-A

1. Define AVL Tree.

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?

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.

Eg: AVL trees, Splay trees.

3. What are the various rotations in AVL trees?

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 subtree of the left child of A.

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

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

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

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

The height of left subtree minus height of right subtree 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. Define splay tree.

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.

6. 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.

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. 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.

9. Define B-tree of order M.

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 2 and M children.

• All non-leaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.

• All leaves are at the same depth.

10. 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.

11. What do you mean by 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.
• All leaves are at the same depth.

12. What are the applications of B-tree?

• Database implementation
• Indexing on non primary key fields

13. 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.

14. 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.

15. 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

16. 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∩Sj=Φ. Each set is identified by a unique element called its representative.

17. 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.

18. 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

19. What do you mean by union-by-weight?

Keep track of the weight ie)size of each tree and always append the smaller tree to the larger one when performing UNION.

20. 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.

21. Define Red-Black tree

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

22. Write the properties of Red-Black tree

- 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

23. Define black height of a tree

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

24. 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

25. Compare 2-3 tree with 2-3-4 tree

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.

Part – B

1. **Discuss, Compare and Contrast Binomial heap and Fibonacci heap in terms of insertion, deletion and applications.**

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. **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?**

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

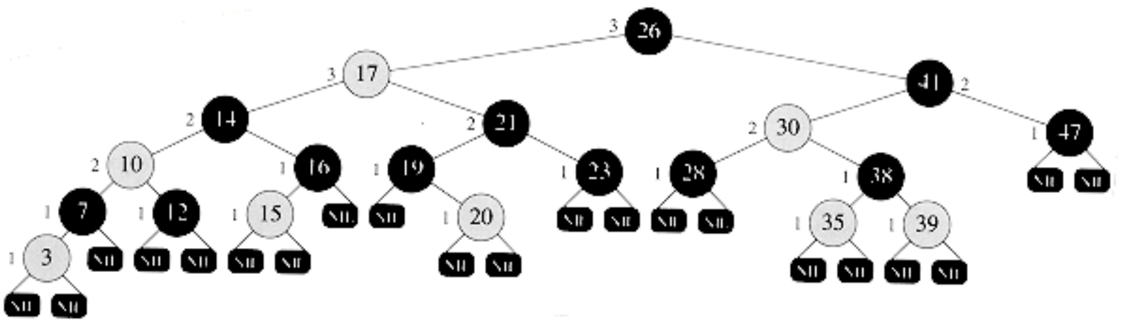
A ***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, 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 black-height 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 black-height; 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 subtree. (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 subtree (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 colours.

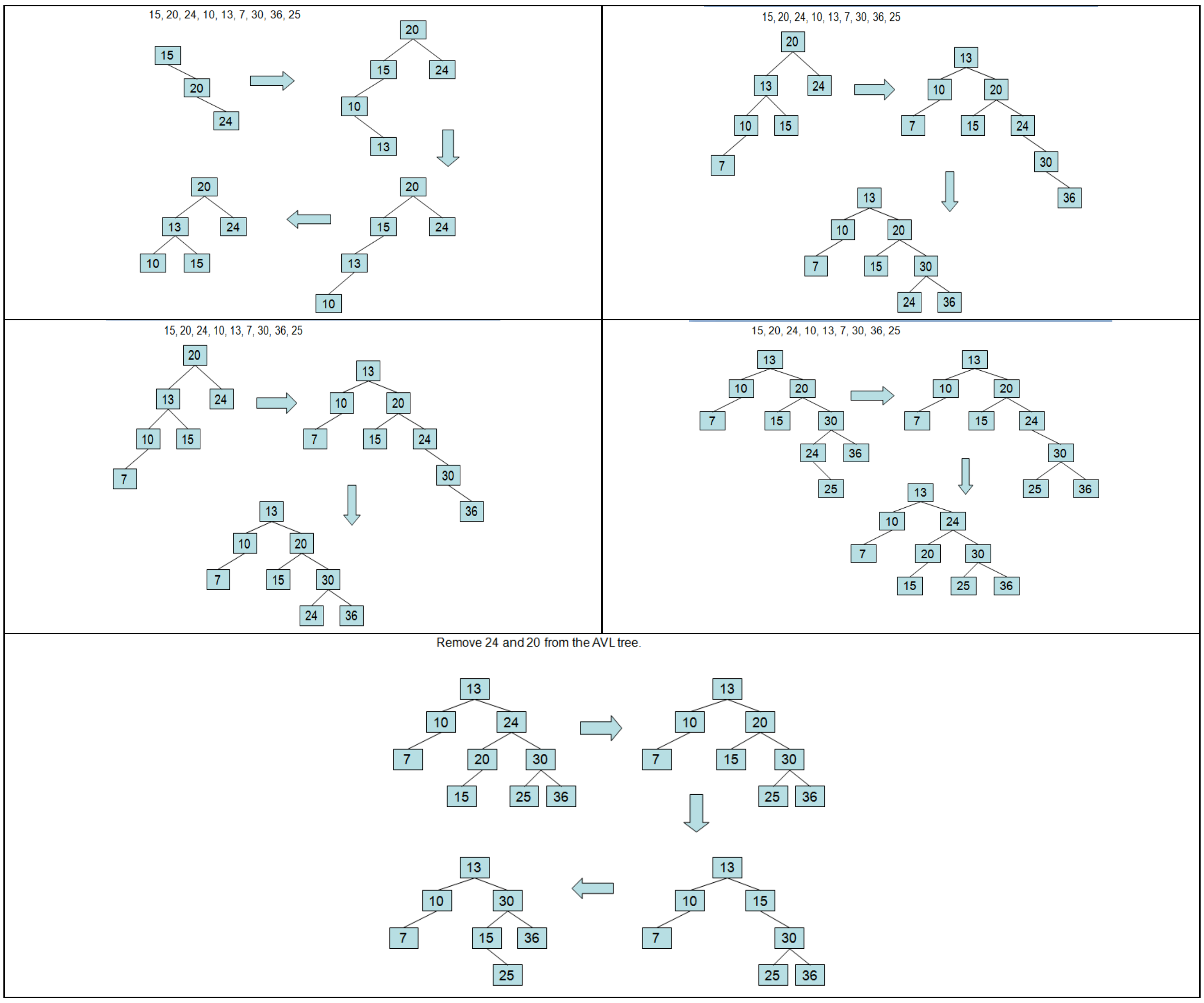
- The item to be deleted is now in a node having only a left child or only a right child. Replace this node with its sole child. This may violate red constraint or black constraint. Violation of red constraint can be easily fixed.
- If the deleted node is block, the black constraint is violated. The 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 colour it black to make up for the loss of one black node.

2. The replacing node is black.

3. Illustrate with an algorithm, to show the insertion of data into an AVL Tree

An AVL tree with the following values: 15, 20, 24, 10, 13, 7, 30, 36, 25



```
TreeNode *InsertAVL(TreeNode *root, TreeNode *newnode, Boolean *taller)
{
if (!root) {
root = newnode;
root->left = root->right = NULL;
root->bf = EH;
*taller = TRUE;
} else if (EQ(newnode->entry.key, root->entry.key)) {
Error("Duplicate key is not allowed in AVL tree.");
} else if (LT(newnode->entry.key, root->entry.key)) {
root->left = InsertAVL(root->left, newnode, taller);
if (*taller) /* Left subtree is taller. */
switch(root->bf) {
case LH: /* Node was left high. */
root = LeftBalance(root, taller); break;
case EH:
root->bf = LH; break; /* Node is now left high. */
case RH:
root->bf = EH; /* Node now has balanced height.*/
*taller = FALSE; break;
}
}
else
{
TreeNode *InsertAVL(TreeNode *root, TreeNode *newnode, Boolean *taller)
{
(continued..)
} else {
root->right = InsertAVL(root->right, newnode, taller);
if (*taller) /* Right subtree is taller. */
switch(root->bf) {
case LH:
root->bf = EH; /* Node now has balanced height.*/
*taller = FALSE; break;
case EH:
root->bf = RH; break; /* Node is right high. */
case RH: /* Node was right high. */
root = RightBalance(root, taller); break;
}
}
return root;
}
```

4. **Explain Amortized analysis with an example**

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(n^2) 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 $=\sum_{i=0}^{k-1} A[i].2^i$, 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:

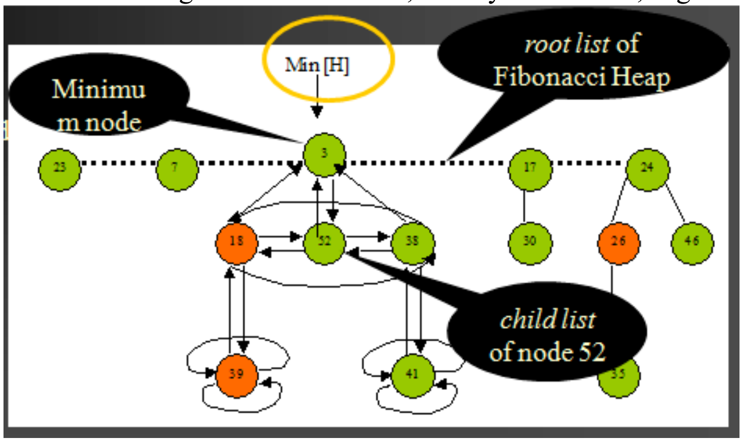
```
Increment()
i=0
while(i<k and A[i]=1)
```

```
Reset(A[i])
i++
if(i<k)
Set(A[i])
```

5. **Explain Fibonacci heap Deletion and Decrease Key operation using cascading cut procedure with example**

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]

Summary

1. 2^k nodes
2. k = height of tree
3. $\binom{k}{i}$ nodes at depth i
4. Unordered binomial tree U_k has root with degree k greater than any other node. Children are trees U_0, U_1, .., U_{k-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

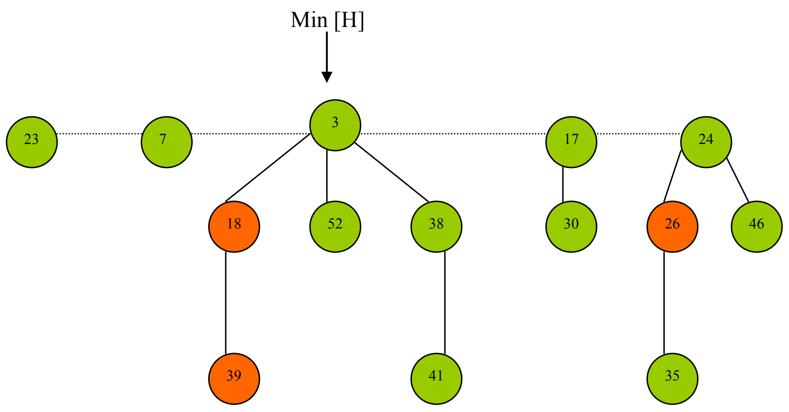
Amortized cost is equal to actual cost of O(1)

Insert – amortized cost is O(1)

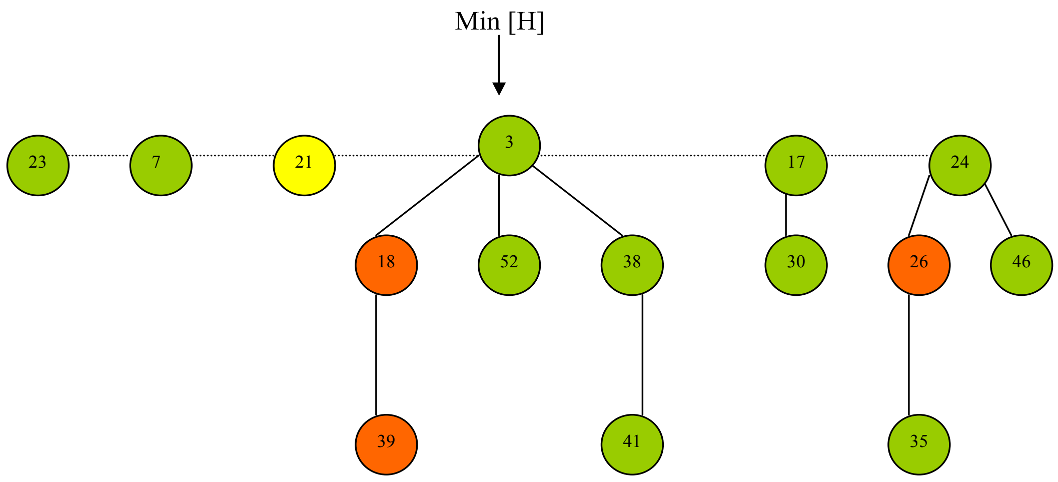
- Initialize the structural fields of node x, add it to the root list of H

- 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 Hea
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.

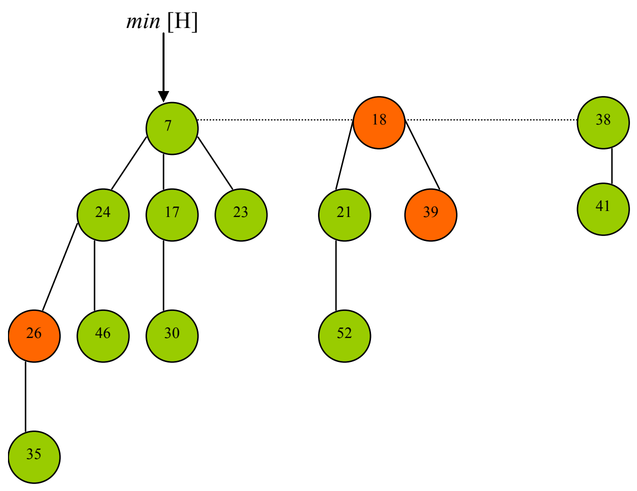
We mark the fields to obtain the desired time bounds. This relates to the "potential function" used to analyze the time complexity of Fibonacci Heaps.

Decrease-Key - amortized cost is O(1)

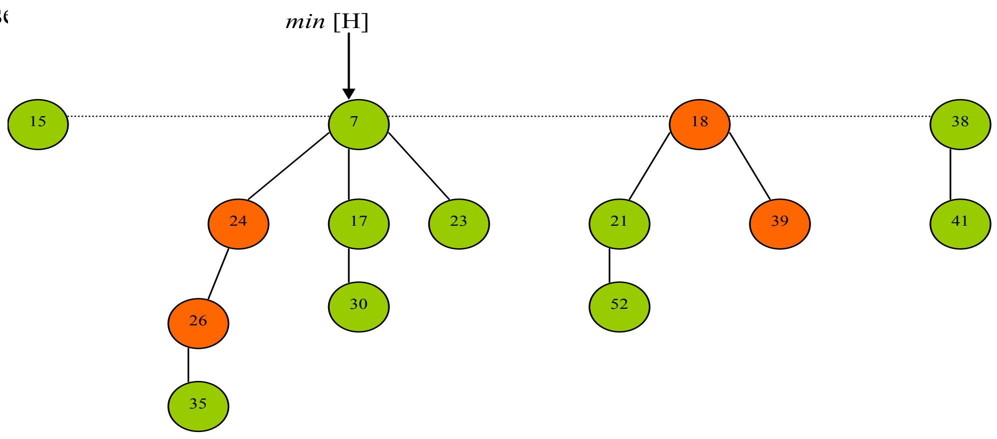
- Check that new key is not greater than current key, then assign new key to x
  - If x is a root or if heap property is maintained, no structural changes
  - Else
    - ***Cut* x:** make x a root, remove link from parent y, clear marked field of x
    - Perform a ***Cascading cut*** on x's parent y(relevant if parent is marked):
      - if unmarked: mark y, return
      - if marked: *Cut* y, recurse on node y's parent
    - if y is a root, return

The Cascading cut procedure recurses its way up the tree until a root, or an unmarked node is found
- Update min[H] if necessary

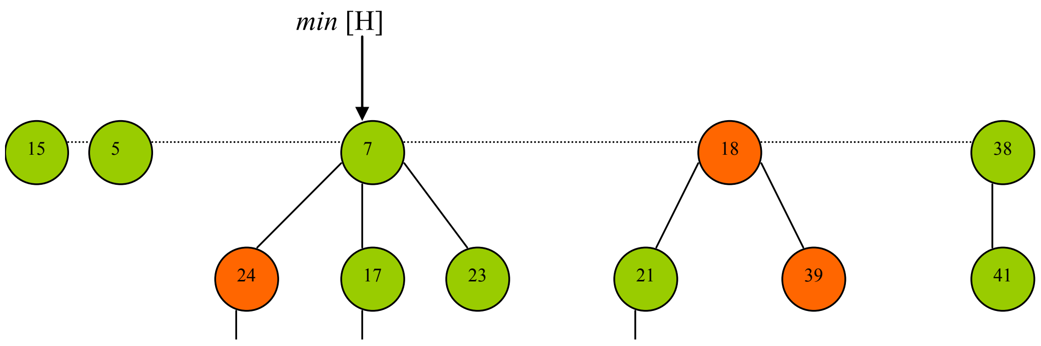
Decrease-Key - amortized cost is O(1)
- We start with a heap
- Suppose we want to decrease the keys of node 46 to 15



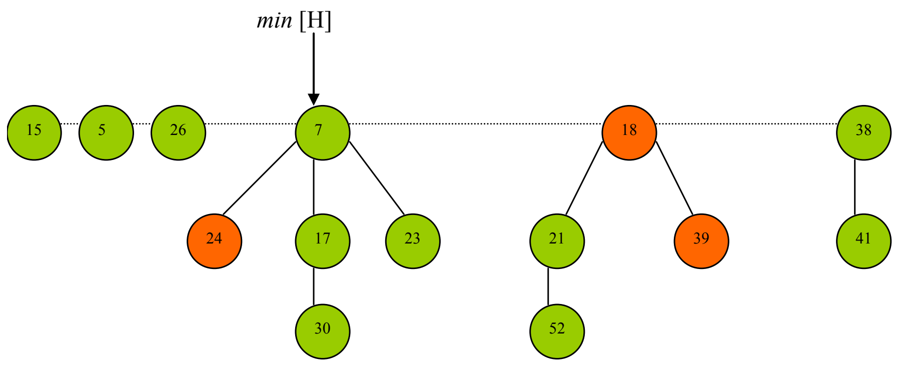
Decrease-Key (example continued)
- Since the new key of node 46 is 15 it violates the min-heap property, so it is cut and put on the root
- Suppos



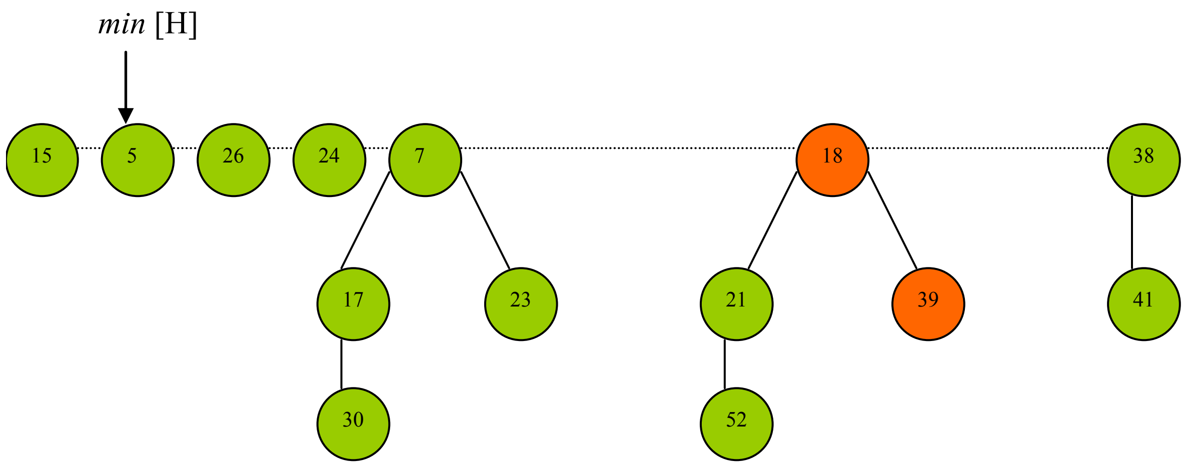
Decrease-Key (example continued)
- So node 5 is cut and place on the root list because again the heap property is violated
- But the parent of node 35, which is node 26, is ***marked***, so we have to perform a ***cascading cut***



Decrease-Key (example continued)
- The cascading cut involves cutting the marked node, 26, unmarking it, and putting it on the root list
- We now repeat this procedure (recurse) on the marked nodes parent until we hit a node on the root list



Decrease-Key (example continued)
- Since the next node encountered in the cascading cut procedure is a root node – node 7 – the procedure terminates
- Now the min heap pointer is updated



6. **Explain Insertion Procedure in Red Black tree and insert the following sequence {20,10,5,30,40,57,3,2,4,35,25,18,22,21}**

- Insert the new node the way it is done in binary search trees
- Color the node red

- If a discrepancy arises for the red-black tree, fix the tree according to the type of discrepancy.
  A discrepancy can result from a parent and a child both having a red color. The type of discrepancy is determined by the location of the node with respect to its grand parent, and the color of the sibling of the parent.
  Discrepancies in which the sibling is red, are fixed by changes in color. Discrepancies in which the siblings are black, are fixed through AVL-like rotations.

  Changes in color may propagate the problem up toward the root. On the other hand, at most one rotation is sufficient for fixing a discrepancy.

7. **a) Show the result of inserting 10,17,2,4,9, 6,8 into an AVL tree**
   Let the newly inserted node be w
   1) Perform standard BST insert for w.
   2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
   3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
   a) y is left child of z and x is left child of y (Left Left Case)
   b) y is left child of z and x is right child of y (Left Right Case)
   c) y is right child of z and x is right child of y (Right Right Case)
   d) y is right child of z and x is left child of y (Right Left Case)

   b) Write the procedure to implement single rotation and double rotation while inserting nodes in an AVL tree(8)
   Single Rotations.

```
 // rotate n2 with (left child) n1, i.e., clockwise
Node rotateRight (Node n2) {
 Node k = n2.left;
 n2.left = k.right;
 k.right = n2;
  // update heights here if needed
 // ...
 return k;
}
  // rotate n2 with (right child) n3, i.e., counter clockwise
Node rotateLeft (Node n2) {
 Node k = n2.right;
 n2.right = k.left;
 k.left = n2;
 // heights ...
 return k;
}
```

   Double Rotations

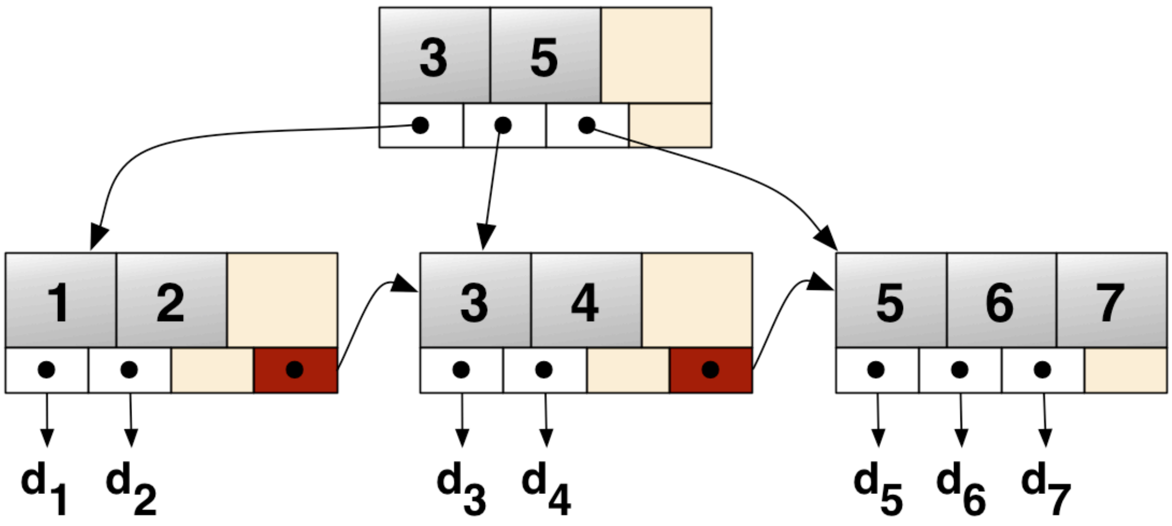
```
 /* rotate n3's left child (n1) with n1's right child (n2)
  * i.e., -> rotate left
  * followed by rotate n3 with n3's (new) left child (n2)
  * i.e., -> rotate right
  * see picture below
  */
 Node doubleRotateLeftRight (Node n3) {
```

```
 n3.left = rotateLeft(n3.left);
 Node k = rotateRight (n3);
 return k;
}
 /* rotate n1's right child(n3) with n3's left child (n2)
 * -> rotate right,
 * followed by rotate n1 with n1's (new) right child (n2)
 * -> rotate left
 * see picture below
 */
 Node doubleRotateRightLeft (Node n1) {
N1.right = rotateRight(n1.right);
Node k = rotateLeft(n1);
Return k;
}
```

8. Discuss in detail the B-tree. What are its advantages?

A **B+ tree** is an n-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.
The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout (number of pointers to child nodes in a node, typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.



A ***B-tree*** *T* is a rooted tree (with root *root*[*T*]) having the following properties.
1. Every node *x* has the following fields:
   a. *n*[*x*], the number of keys currently stored in node *x*,
   b. the *n*[*x*] keys themselves, stored in nondecreasing order: $key_1[x] \leq key_2[x] \leq \cdots \leq key_n[x][x]$, and
   c. *leaf* [*x*], a boolean value that is TRUE if *x* is a leaf and FALSE if *x* is an internal node.
2. If *x* is an internal node, it also contains *n*[*x*] + 1 pointers $c_1[x], c_2[x], \ldots, c_n[x]+1[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then $k_1 \le key_1[x] \le k_2 \le key_2[x] \le \cdots \le key_{n[x]}[x] \le k_n[x]+1$.
4. Every leaf has the same depth, which is the tree's height *h*.
5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer $t \ge 2$ called the ***minimum degree*** of the B-tree:
   a. Every node other than the root must have at least t - 1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.
   b. Every node can contain at most $2t$ - 1 keys. Therefore, an internal node can have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t$ - 1 keys.

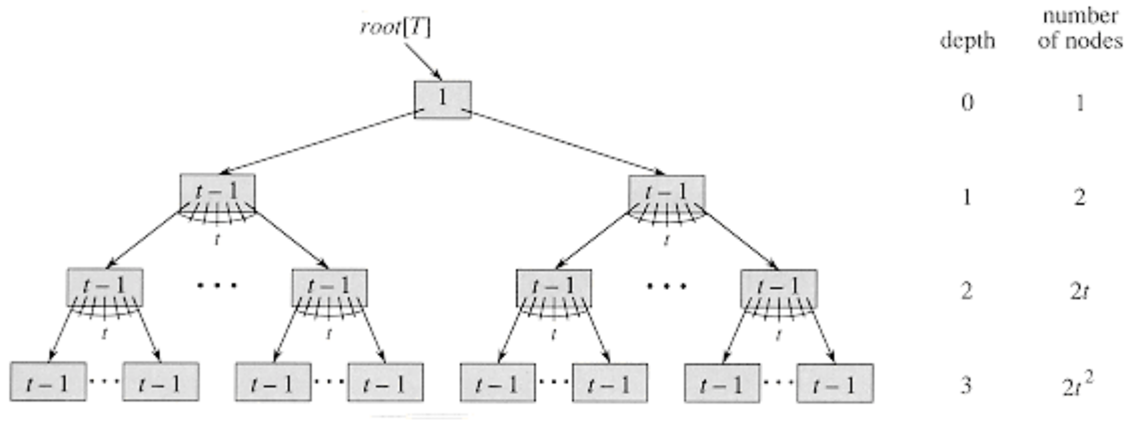
The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a ***2-3-4 tree.*** In practice, however, much larger values of t are typically used.

The height of a B-tree

The number of disk accesses required for most operations on a B-tree is proportional to the height of the B-tree. We now analyze the worst-case height of a B-tree.

If $n \ge 1$, then for any n-key B-tree T of height h and minimum degree $t \ge 2$,

$$h \le \log_t \frac{n+1}{2} .$$



A B-tree of height 3 containing a minimum possible number of keys. Shown inside each node x is n[x]

B-Trees take advantage of this by maintaining a balanced binary tree structure through the use of two files:

- index file: contains all the keys and the tree's topology is represented by the organization of data in this file.
- data file: a file that contains all the objects and information stored by the “tree”. Objects contained here are referenced by block pointer references stored in the index file.

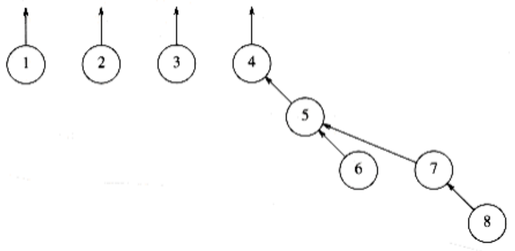
9. a) Explain the operation of Union by height with an algorithm

The unions in the basic tree data structure representation were performed arbitrarily, by making the second tree a subtree of the first. A basic improvement is to make the smaller tree a subtree of the larger. We call this approach union-by-size

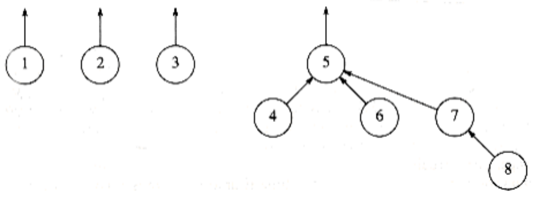
If unions are done by size, the depth of any node is never more than log n. Note that a node is initially at depth 0. When its depth increases as a result of a union, it is placed in a tree that is at least twice as large as before. Thus, its depth can be increased at most log n times. This implies that the running time for a find operation is O(log n), and a sequence of m operations takes O(m log n).

If unions are done by size, the depth of any node is never more than log n. Note that a node is initially at depth 0. When its depth increases as a result of a union, it is placed in a tree that is at least twice as large as before. Thus, its depth can be increased at most log n times. This implies that the running time for a find operation is O(log n), and a sequence of m operations takes O(m log n).

An alternative implementation, which also guarantees that all the trees will have depth at most O(log n), is union-by-rank. We keep track of the height, instead of the size, of each tree and perform unions by making the shallow tree a subtree of the deeper tree. This is an easy algorithm, since the height of a tree increases only when two equally deep trees are joined (and then the height goes up by one). Thus, union-by-height is a trivial modification of union-by-size.



Result of arbitrary union



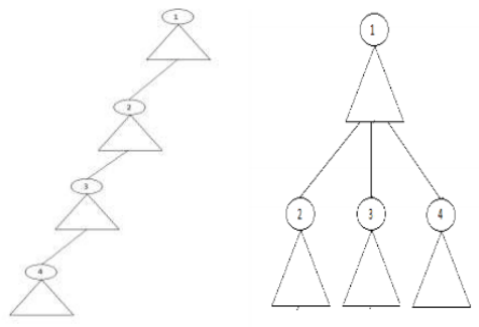
Result of union-by-size/ union-by-rank

A small snippet for the union-by-rank algorithm is shown below: Assume x and y are two nodes

```
Union(x, y)
 Link(Find-set(x), Find-set(y))
 Link(x, y)
 if rank[x] > rank[y]
 then p[y] = x
 else p[x] = y
 if rank[x] == rank[y]
 then rank[y] = rank[y] + 1
```

b) Discuss the disjoint set find with path compression using suitable algorithm

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.



(a) **(b)**

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.

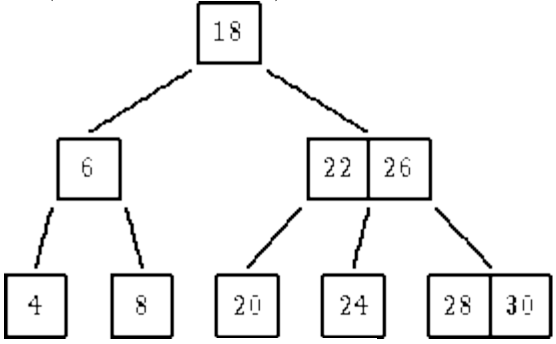
We do not prove it, but, if there are n Make-set operations and at most n - 1 Union operations and f Find-set operations, the path-compression heuristic alone gives a worst-case running time of Θ (n + f · (1 + log2+ f / n n)).

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, $\alpha(m,n) \leq 4$. Thus, for practical purposes the running time can be viewed as linear in m (no. of operations).

10. a) 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

A m-way tree in which

- The root has at least one key
- Non-root nodes have at least $\lceil m/2 \rceil$ subtrees (i.e., at least $\lfloor (m - 1)/2 \rfloor$ keys)
- All the empty subtrees (i.e., external nodes) are at the same level



B-tree of order 3

Insertions

- Insert the key to a leaf
- Overfilled nodes should send the middle key to their parent, and split into two at the location of the submitted key.

b) Discuss how to insert an element in a AVL tree. explain with example

Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

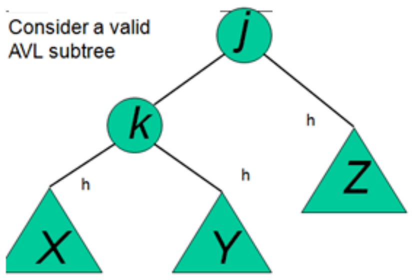
1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

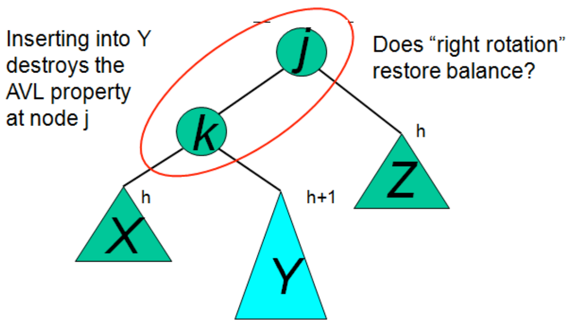
Inside Cases (require double rotation) :

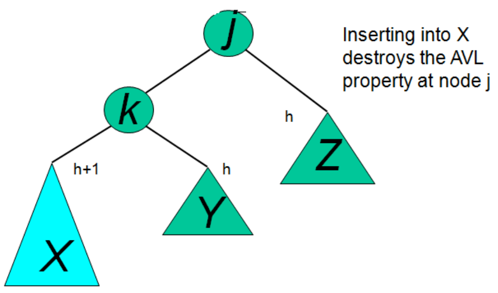
3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

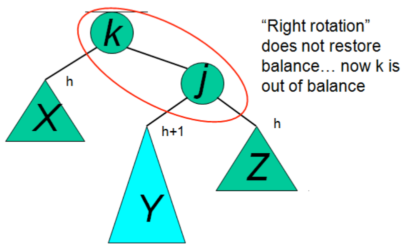
The rebalancing is performed through four separate rotation algorithms.

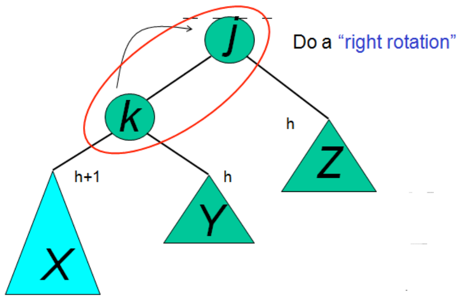
AVL Insertion: Outside Case

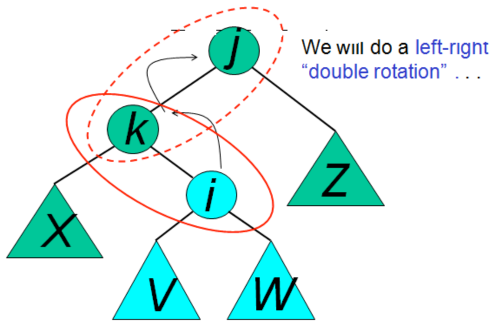



Inserting into Y destroys the AVL property at node j
Does "right rotation" restore balance?
j
k
h
h+1
h
X
Y
Z

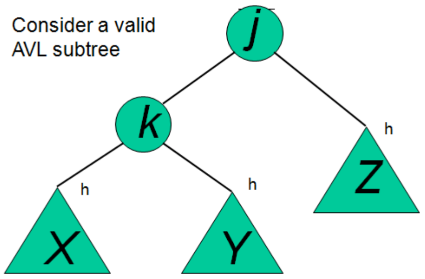


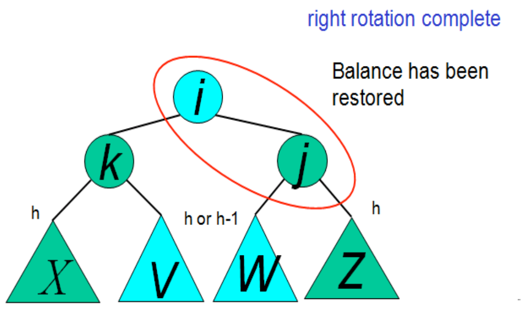

"Right rotation" does not restore balance… now k is out of balance
k
j
h
h+1
h
X
Y
Z





AVL Insertion: Inside Case





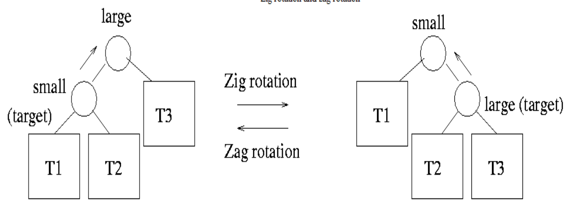
11. What is splay tree? Discuss briefly the various rotations in a splay tree with an example

- A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.
- Amortized complexity
  - Splay trees can become highly unbalanced so that a single access to a node of the tree can be quite expensive.
  - However, over a long sequence of accesses, the few expensive cases are averaged in with many inexpensive cases to obtain good performance.
- Does not need heights or balance factors as in AVL trees and colours as in Red-Black trees.
- The surgery on the tree is done using **rotations**, also called as splaying steps. There are six different splaying steps.

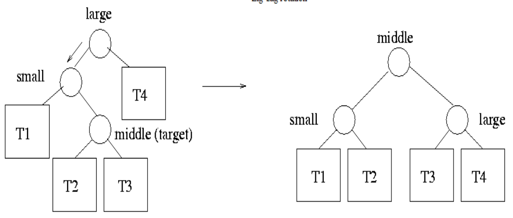
1.Zig Rotation (Right Rotation)
2.Zag Rotation (Left Rotation)
3.Zig-Zag (Zig followed by Zag)
4.Zag-Zig (Zag followed by Zig)
5.Zig-Zig
6.Zag-Zag

- Consider the path going from the root down to the accessed node.
  - Each time we move left going down this path, we say we ``zig'' and each time we move right, we say we ``zag.''

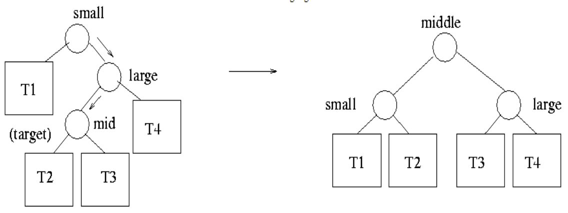
Zig rotation and zag rotation



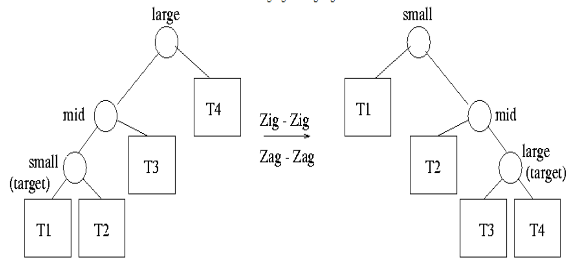
Zig-zag rotation



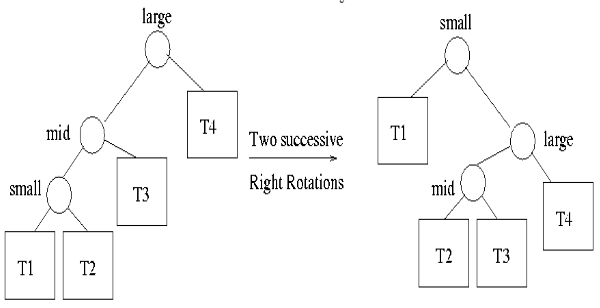
Zag-zig rotation



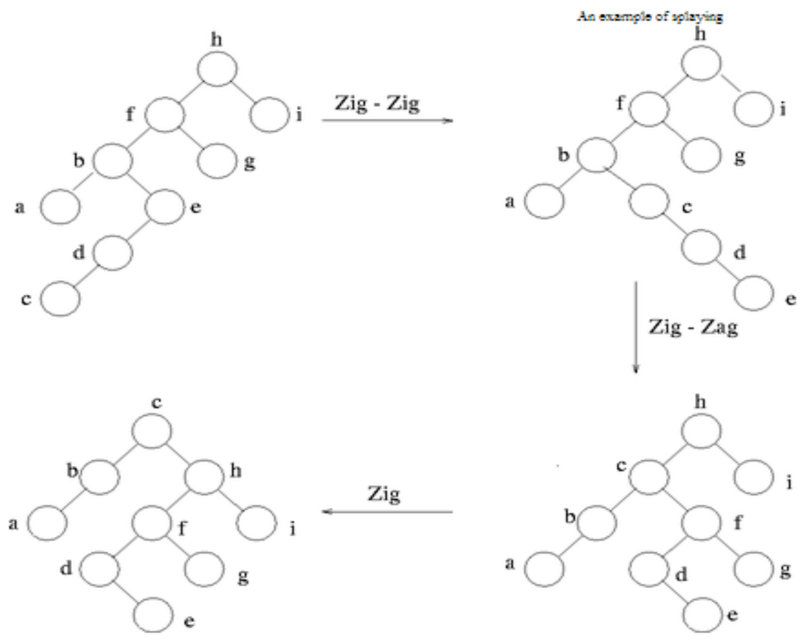
Zig-zig and zag-zag rotations



Two successive right rotations



Example



Unit 5
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 graph in which every edge is directed is called a directed graph.

4. What is an undirected graph?

A graph in which every edge is undirected is called a directed graph.

5. What is a loop?

An edge of a graph which connects to itself is called a loop or sling.

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 graph in which weights are assigned to every edge is called a weighted graph.

8. Define outdegree 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 indegree 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 indegree 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.

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.

12. What is a cycle or a circuit?

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

13. What is an acyclic graph?

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

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.Name the different ways of representing a graph?

a.Adjacencymatrix
b. Adjacency list

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?

a.Breadthfirstsearch
b. Depth first search

19. What is a minimum spanning tree?

A minimum spanning tree of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at the lowest total cost.

A spanning tree of a graph in an undirected tree consists of only those edges necessary to connect all those nodes in the original graph. A spanning tree has the property that for any pair of node, there exists only one path between them. Since it is a tree, it does not contain cycle.

20. Name two algorithms two find minimum spanning tree

Kruskal'salgorithm
Prim's algorithm

21. Define graph traversals.

Traversing a graph is an efficient way to visit each vertex and edge exactly once.

22. 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.

23. What do you mean by breadth first search (BFS)?

The breadth first traversal is a graph search algorithm that begins at root node and explores all the beginning nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes and so on, until it finds the goal.

24.Differentiate BFSandDFS.

No.	DFS	BFS
1.	Backtracking is possible from a dead end	Backtracking is not possible
2.	Vertices from which exploration is incomplete are processed in a	The vertices to be explored are organized as a
3.	Search is done in one particular Direction	The vertices in the same level are maintained

25. What is topological sort?

A topological sort is a ordering of vertices in a directed acyclic graph such that if there is a path from v_i to v_j then v_j appears after v_i

26. What is the purpose of Dijkstra's algorithm?

Dijkstra's Algorithm is to solve the single source shortest path problem in a weighted graph. It is a greedy approach. At each stage, it selects a vertex v, which has the smallest distance among all the unknown vertices, and declares that as the shortest path from S to V and make it to be known.

27. Mention the algorithm to find the single source shortest path

- Dijkstra's algorithm Bellman Ford algorithm Floyd Warshall algorithm

Part – B

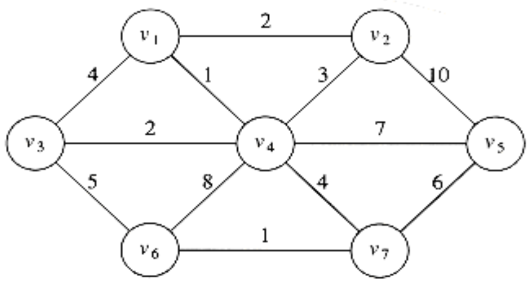
1. Write and explain the prim's algorithm with an example

Prim's Algorithm to Construct a Minimal Spanning Tree

Input: A weighted, connected and undirected graph G = (V,E).

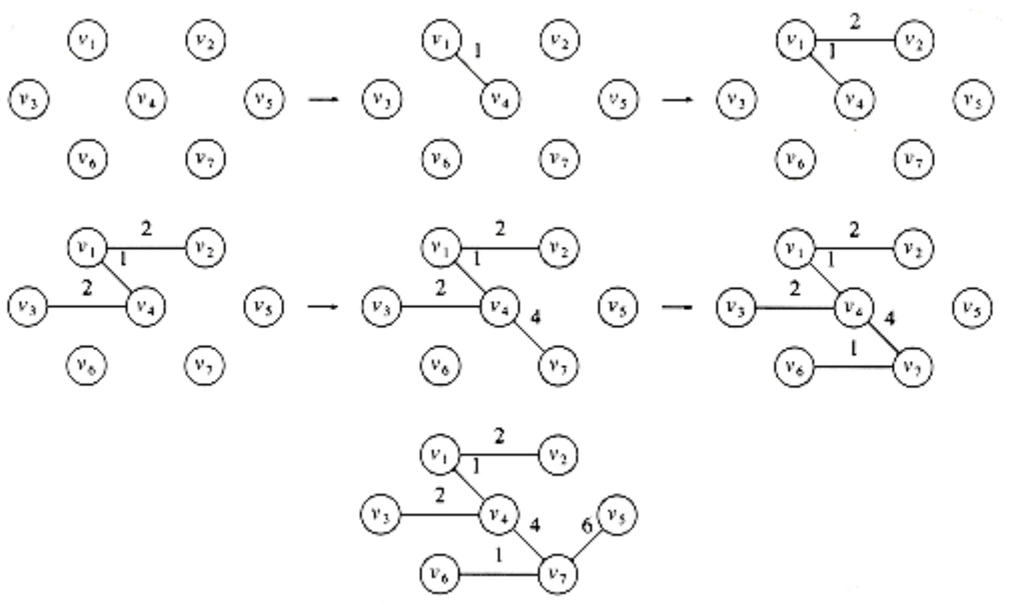
Output: A minimal spanning tree of G.

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



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

Initial configuration of table used in Prim's algorithm

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

The table after v6 and v5 are selected (Prim's algorithm terminates)

2. Explain topological sort with suitable algorithm and example

Definition: A **topological sort** is a linear ordering of vertices in a directed acyclic graph such that if there is a path from V_i to V_j, then V_j appears after V_i 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 */
}
```

Compare BFS and DFS

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.

4. Describe the various representation of graphs
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
$A_{ij} = 1$, if there is an edge V_i to V_j
$A_{ij} = 0$, if there is no edge.

Advantage
* Simple to implement.

Disadvantage
* Takes $O(n^2)$ space to represents the graph
* It takes $O(n^2)$ 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.

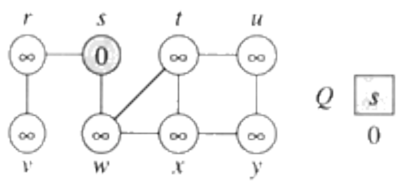
5. Explain the breadth first search algorithm

Algorithm:

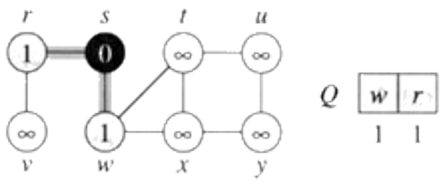
```
bfs(g)
  {
  list l = empty
  tree t = empty
  choose a starting vertex x
  search(x)
  while(l nonempty)
     remove edge (v,w) from start of l
     if w not yet visited
     {
     add (v,w) to t
     search(w)
     }
  }
```

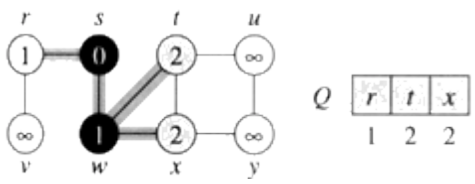
Example:

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*.

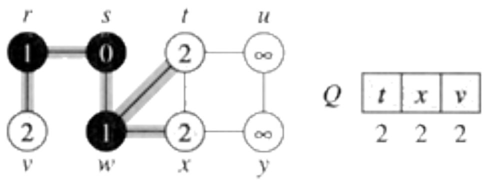


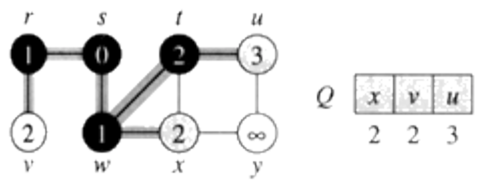
The algorithm discovers all vertices 1 edge from *s* i.e., discovered all vertices (*w* and *r*) at level 1.

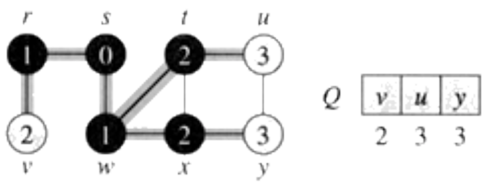




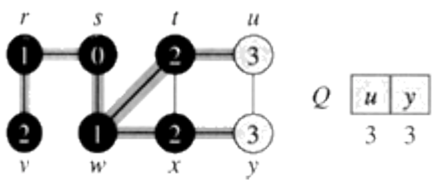
The algorithm discovers all vertices 2 edges from *s* i.e., discovered all vertices (*t*, *x*, and *v*) at level 2.

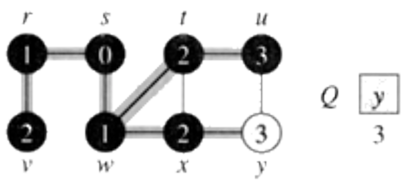




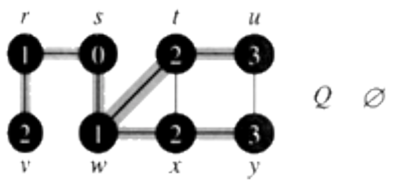


The algorithm discovers all vertices 3 edges from *s* i.e., discovered all vertices (*u* and *y*) at level 3.





The algorithm terminates when every vertex has been fully explored.



6. Consider the following graph shown in figure 1 Obtain the minimum spanning tree using Kruskal's algorithm

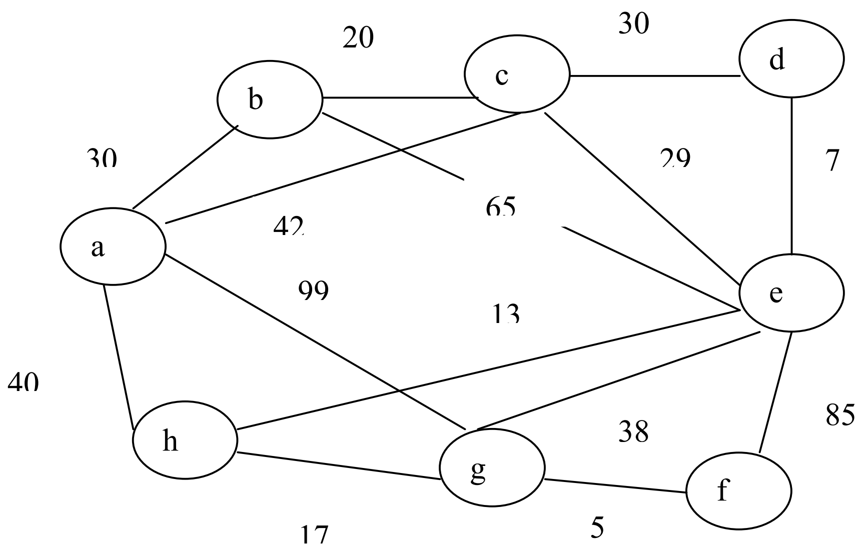
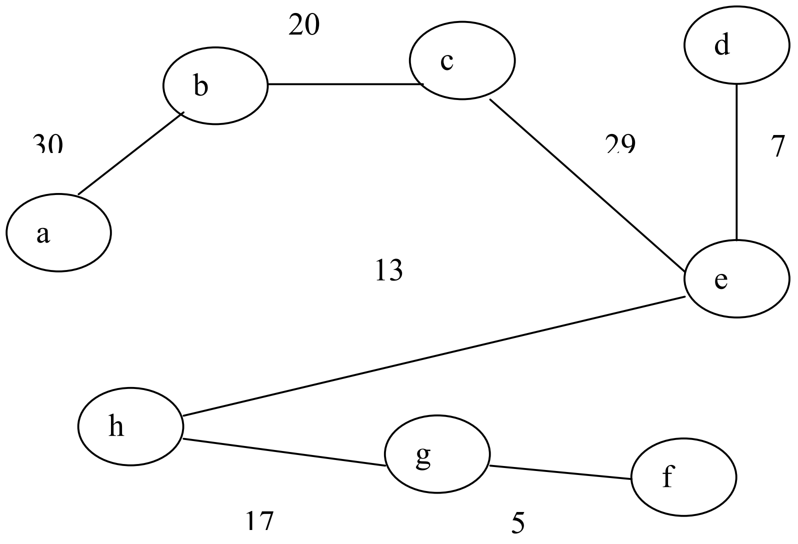


Figure : 1

- 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

7. Find the shortest path from a to d using Dijkstra's algorithm in the graph shown above in Figure : 1

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

Kruskal's Algorithm to Construct a Minimal Spanning Tree

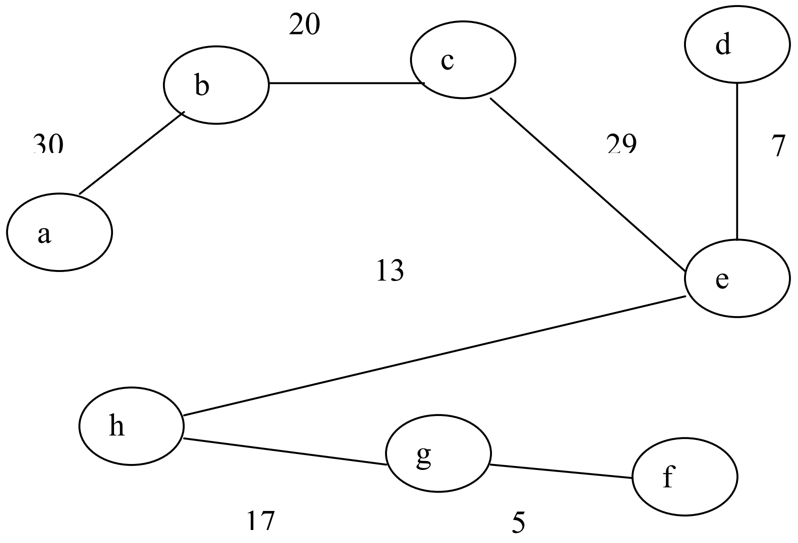
Input: A weighted, connected and undirected graph *G* = (*V*,*E*).

Output: A minimal spanning tree of *G*.

```
Step 1: $T = \phi$.
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:



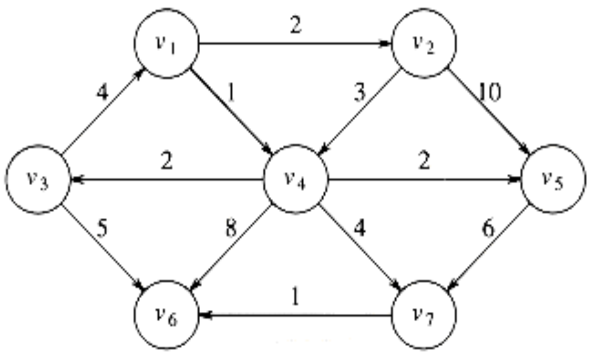
9. What is single source shortest path problem? Discuss Dijkstra's single source shortest path algorithm with an example [Nov/Dec 2007]

Single source shortest path: Given a graph $G = (V, E)$, we want to find a shortest path from a given ***source*** vertex $s \in V$ to every vertex $v \in V$.

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.

v Known ***dv pv***
----

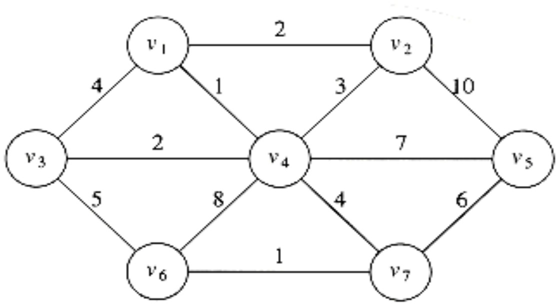
v1	0	0	0
v2	0	∞	0
v3	0	∞	0
v4	0	∞	0
v5	0	∞	0
v6	0	∞	0
v7	0	∞	0

Final table:

v Known ***dv pv***
----

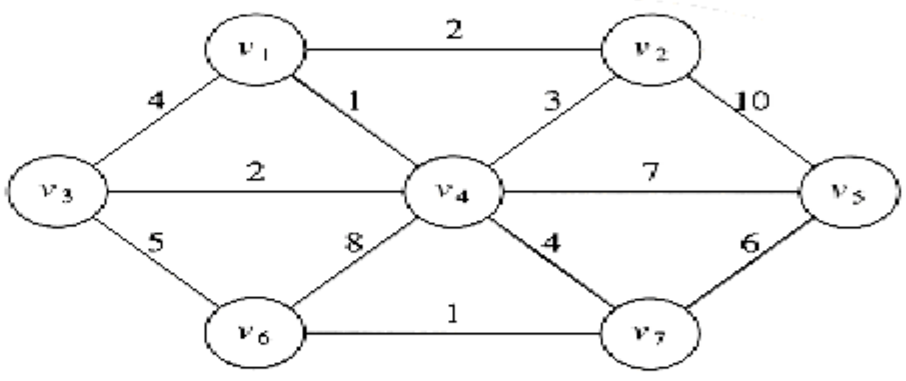
v1	1	0	0
v2	1	2	v1
v3	1	3	v4
v4	1	1	v1
v5	1	3	v4
v6	1	6	v7
v7	1	5	v4

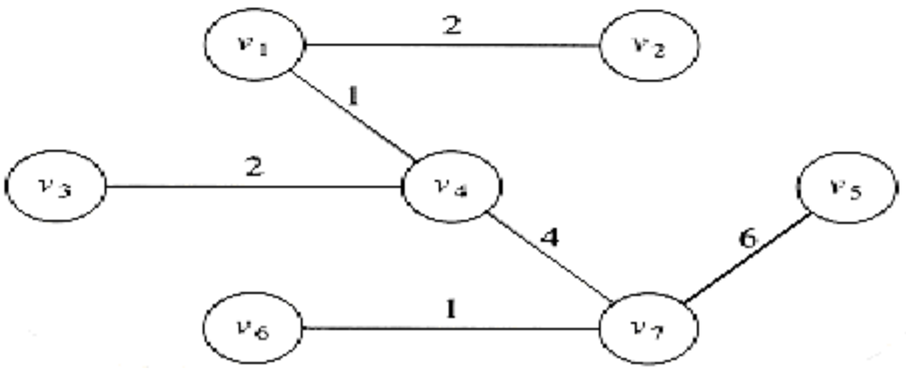
10. a) Write an algorithm to find the minimum spanning tree of an undirected weighted graph (8)



Kruskal's algorithm

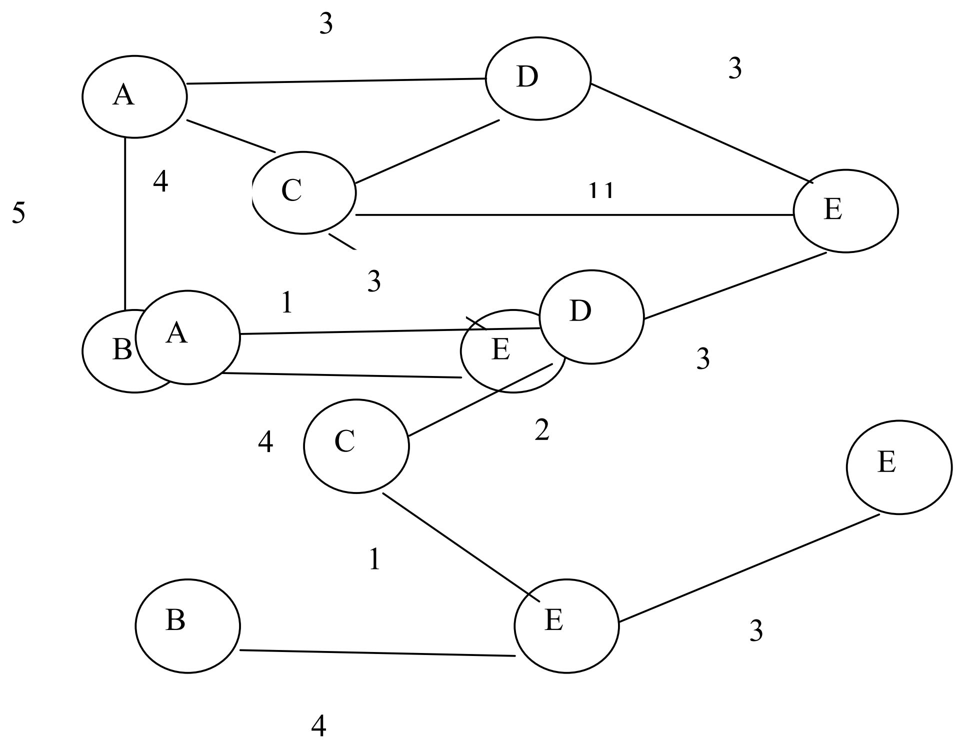
Edge	Weight	Action
(*v*1,*v*4)	1	Accepted
(*v*6,*v*7)	1	Accepted
(*v*1,*v*2)	2	Accepted
(*v*3,*v*4)	2	Accepted
(*v*2,*v*4)	3	Rejected
(*v*1,*v*3)	4	Rejected
(*v*4,*v*7)	4	Accepted
(*v*3,*v*6)	5	Rejected
(*v*5,*v*7)	6	Accepted





b) Find MST for the following graph (8)

Solution:



11. Explain Bellman Ford Algorithm with an example

- Single source shortest path algorithm
- Algorithm:

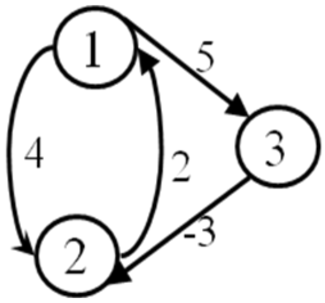
```
INITIALIZE-SINGLE-SOURCE (G, s)
for each vertex i = 1 to V[G] - 1 do
    for each edge (u, v) in E[G] do
        RELAX (u, v, w)
For each edge (u, v) in E[G] do
    if d[u] + w(u, v) < d[v] then
        return FALSE
return TRUE
```

12. Explain Floyd Warshall algorithm with an example

- 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
        // Computing D' from D
    do for i ← 1 to n
        do for j ← 1 to n
            if (D[ i, j ] > D[ i, k ] + D[ k, j ] )
            then  $D'[i,j] \leftarrow D[i,k] + D[k,j]$
                  $P[i,j] \leftarrow k$;
              else $D'[i,j] \leftarrow D[i,j]$
    Move D' to D.
```

Example:



Answer:

D^0

0	4	5
2	0	∞
∞	-3	0

D^3

0	2	5
2	0	7
-1	-3	0