Question Paper Code : 27158

B.E./B.Tech. DEGREE EXAMINATION, NOVEMBER/DECEMBER 2015.

Third Semester

Computer Science and Engineering

CS 6301 — PROGRAMMING AND DATA STRUCTURES — II

(Common to Information Technology)

(Regulations 2013)

Time : Three hours Maximum : 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. Give the significance of declaring a member of a class static.
2. What is the use of "this" pointer?
3. How the C string differs from a C++ type string?
4. What is dynamic initialization of objects?
5. Compare overloaded functions versus function templates.
6. When do we use multiple catch handlers?
7. What are the various operations that can be performed on B-trees?
8. What are Splay trees?
9. What is the minimum number of spanning trees possible for a complete graph with n vertices?
10. What is topological sorting?

PART B — (5 × 16 = 80 marks)

11. (a) (i) How can you specify a class? (6)

(ii) Describe the different mechanisms of accessing data members and member functions in a class with a suitable example. (10)

Or

(b) (i) Explain the different types of constructors with suitable examples. (10)

(ii) Describe the types of storage classes. (6)

12. (a) (i) Write a C++ program to overload the increment operator with prefix and postfix forms. (12)

(ii) Distinguish the term overloading and overriding. (4)

Or

(b) (i) Write a C++ program to explain how the run time polymorphism is achieved. (8)

(ii) Illustrate any four types of inheritance supported in C++ with suitable examples. (8)

13. (a) (i) Write a function template for finding the maximum value in an array. (8)

(ii) Write a C++ program to handle a divide by zero exception. (8)

Or

(b) (i) Describe the components of STL. (8)

(ii) Write a class template to represent a stack of any possible data type. (8)

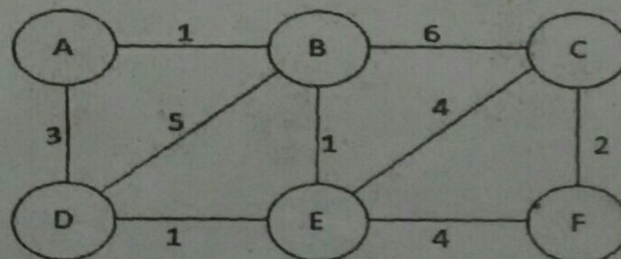
14. (a) (i) Define AVL tree and starting with an empty AVL search tree, insert the following elements in the given order: 35, 45, 65, 55, 75, 15, 25 (8)

(ii) Explain the AVL rotations with a suitable example. (8)

Or

(b) Illustrate the construction of Binomial Heaps and its operations with a suitable example. (16)

15. (a) (i) Compute the minimum spanning tree for the following graph. (8)



(ii) Discuss any two applications of depth-first search. (8)

Or

(b) Explain the Dijkstra's algorithm for finding the shortest path with a sample graph.

CS 6301 –Programming and Data structures –II
Nov-Dec 2015
Solved University Question Paper

PartA-(10X 2=20 Marks)

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

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

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

4..**What is dynamic initialization of objects?**
Dynamic initialization is that in which initialization value isn't known at compile-time. It's computed at runtime to initialize the variable.

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

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

7.What are the various operations that can be performed on B Trees
Adding an element to a B-tree.

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

8.What are splay trees
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.

9.What is the minimum number of spanning tree possible for a complete graph with n vertices? The number of spanning trees of G, denoted by t(G), is the total number of distinct spanning sub graphs of G that are trees. We consider the problem of determining some special classes of graphs having the maximum number of spanning trees (or the maximum spanning tree graph problem).

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

11.i)How can you specify a class? (4)

Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.
An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.
Classes are defined using either keyword class or keyword struct, with the following syntax:

```
class class_name {
  access_specifier_1:
    member1;
```

```
 access_specifier_2:
   member2;
 ...
} object_names;
```

ii) Describe different mechanisms of accessing data members and member functions in a class with a suitable example.

Accessing Data Members of Class

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

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

Accessing Public Data Members

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

```
class Student
{
 public:
 int rollno;
 string name;
};
int main()
{
 Student A;
 Student B;
 A.rollno=1;
 A.name="Adam";
 B.rollno=2;
 B.name="Bella";
 cout <<"Name and Roll no of A is :"<< A.name << A.rollno;
 cout <<"Name and Roll no of B is :"<< B.name << B.rollno;
}
```

Accessing Private Data Members

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

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

Example :

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

 public:     // public accessor and mutator functions
 int getRollno()
 {
  return rollno;
 }
 void setRollno(int i)
 {
  rollno=i;
 }
};
int main()
{
 Student A;
 A.rollono=1;  //Compile time error
 cout<< A.rollno; //Compile time error

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

11.b i)Explain the different types of constructor with suitable examples

Constructors

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

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

While defining a contructor you must remeber that the name of constructor will be same as the name of the class, and contructors never have return type.

Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A
```

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

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

Types of Constructors

Constructors are of three types :

1. Default Constructor
2. Parametrized Constructor
3. Copy Constructor

Default Constructor

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

Syntax :

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

Example :

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

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

Output : 10

In this case, as soon as the object is created the constructor is called which initializes its data members.

A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

```
class Cube
{
 int side;
};

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

Output : 0

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

Parameterized Constructor

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

Example :

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

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

OUTPUT : 10 20 30

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

b. ii)Describe the different types of 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.

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 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_ex
     tern();
}
```

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.

12.a.i) Write a C++ program to overload the increment operator in prefix and postfix forms (12) using namespace std;

```
class Check
{
  private:
    int i;
  public:
```

```
    Check(): i(0) {   }
    Check operator ++() /* Notice, return type Check*/
    {
       Check temp;  /* Temporary object check created */
       ++i;         /* i increased by 1. */
       temp.i=i;   /* i of object temp is given same value as i */
       return temp; /* Returning object temp */
    }
   void Display()
      { cout<<"i="<<i<<endl; }
};
int main()
{
    Check obj, obj1;
    obj.Display();
    obj1.Display();
    obj1=++obj;
    obj.Display();
    obj1.Display();
    return 0;
}
```

Output

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

Operator Overloading of Postfix Operator

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

/* Notice int inside barcket which indicates postfix increment. */
    Check operator ++ (int)
    {
```

```
        Check temp;
        temp.i=i++;
        return temp;
    }
   void Display()
      { cout<<"i="<<i<<endl; }
};
int main()
{
    Check obj, obj1;
    obj.Display();
    obj1.Display();
    obj1=++obj;     /* Operator function is called then only value of obj is assigned to obj1. */
    obj.Display();
    obj1.Display();
    obj1=obj++;     /* Assigns value of obj to obj1++ then only operator function is called. */
    obj.Display();
    obj1.Display();
    return 0;
}
Output
i=0
i=0
i=1
i=1
i=2
i=1
```

12.a)ii) Distinguish the term overloading and overriding (4)

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

12.b.i)Write a C++ program to explain how the runtime polymorphism is achieved (8)

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

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

class Derived: public Base
```

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

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

ii) Illustrate any four types of inheritance supported in C++ with suitable example (8)

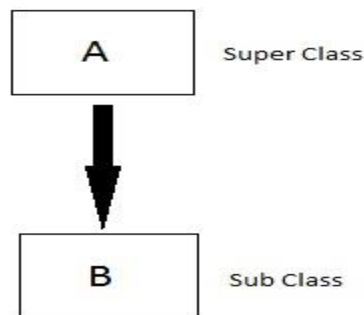
Types of Inheritance

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

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

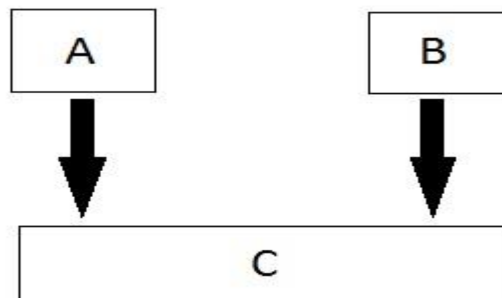
Single Inheritance

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



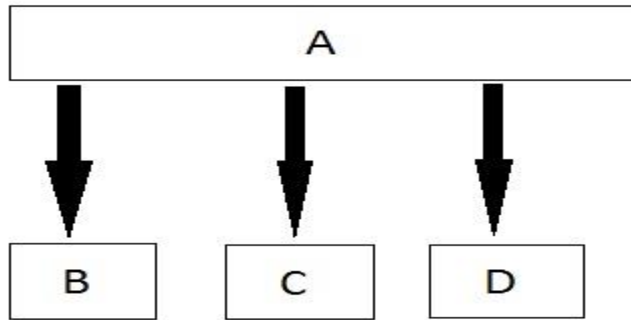
Multiple Inheritance

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



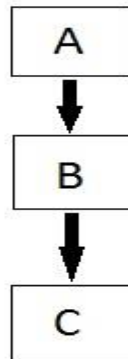
Hierarchical Inheritance

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



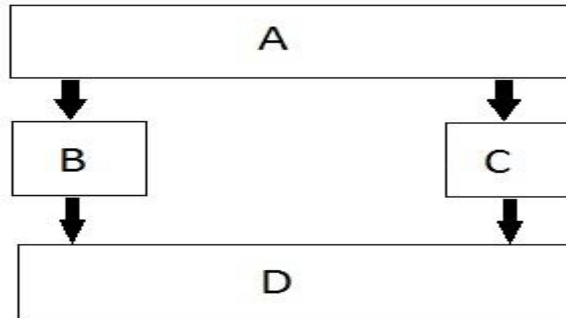
Multilevel Inheritance

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



Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



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

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

Two container :

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

- Portability and reusability

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

ii)Write a class template to represent a stack of any possible data type

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>
using namespace std;
template <class T>
class Stack {
  private:
    vector<T> elems;     // elements

  public:
    void push(T const&);  // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const{       // return true if empty.
        return elems.empty();
    }
};
template <class T>
void Stack<T>::push (T const& elem)
{
    // append copy of passed element
    elems.push_back(elem);
}

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

```
}

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

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

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

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

13.b.i)Write a C++ program to handle a divide by zero exception

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

   try
```

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

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

  getch();
}
```

Output:

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

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

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

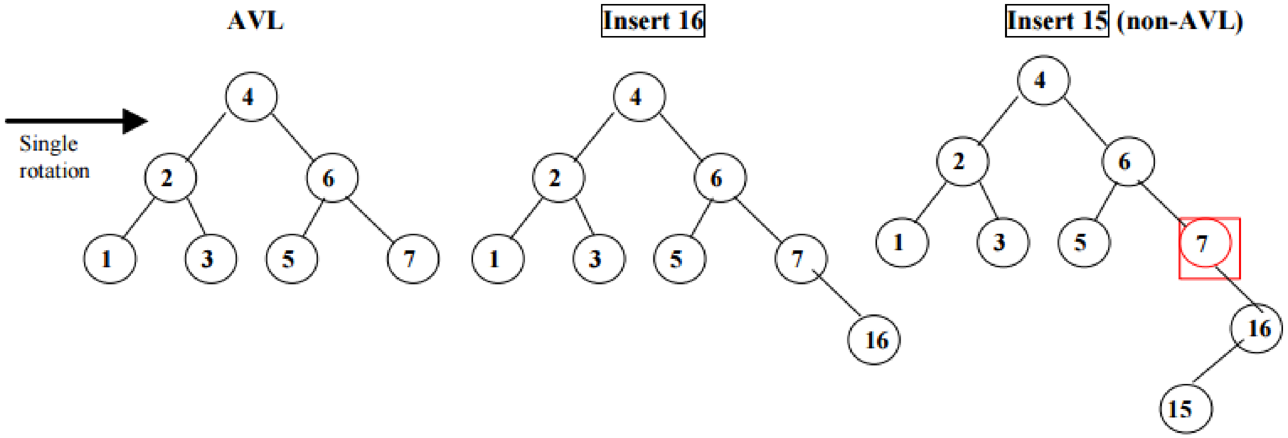
```
#include <iostream>
using std::cout;
using std::endl;
template<class T> T max(const T* data, int size) {
    T result = data[0];
    for(int i = 1 ; i < size ; i++)
      if(result < data[i])
        result = data[i];
    return result;
  }
template<class T> T min(const T* data, int size) {
    T result = data[0];
    for(int i = 1 ; i < size ; i++)
      if(result > data[i])
        result = data[i];
    return result;
  }
int main() {
  double data[] = {1.5, 4.6, 3.1, 1.1, 3.8, 2.1};
  int numbers[] = {2, 22, 4, 6, 122, 12, 1, 45};

  const int dataSize = sizeof data/sizeof data[0];
```

```
  cout << "Minimum double is " << min(data, dataSize) << endl;
  cout << "Maximum double is " << max(data, dataSize) << endl;

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

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

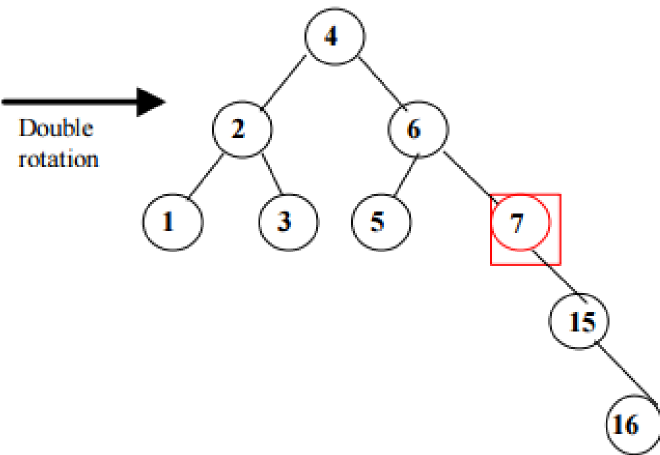
14.a)i)Define AVL Tree and starting with an empty AVL search Tree insert the following elements in the given order: 35,45,65,55,75,15,25(8) (NOV/DEC 2015)

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

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

Insert 75          Insert 15          Insert 25

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

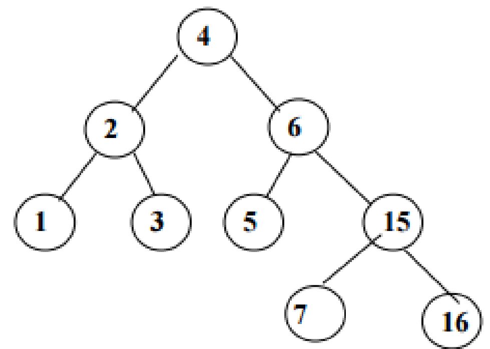
ii)Explain the AVL rotations with suitable example.(8)(NOV/DEC 2015)



Step 1: Rotate child and grandchild



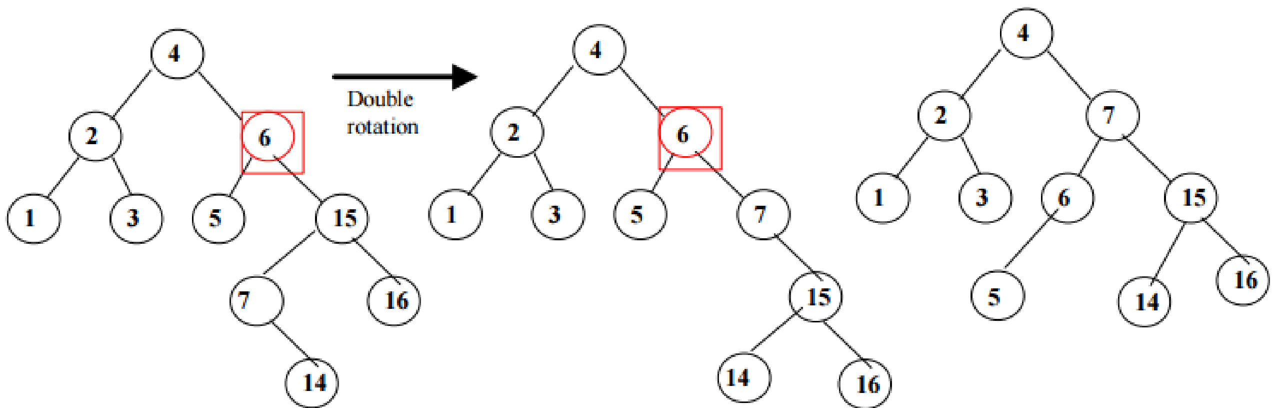
Step 2: Rotate node and new child (AVL)



Insert 14 (non-AVL)

Step 1: Rotate child and grandchild

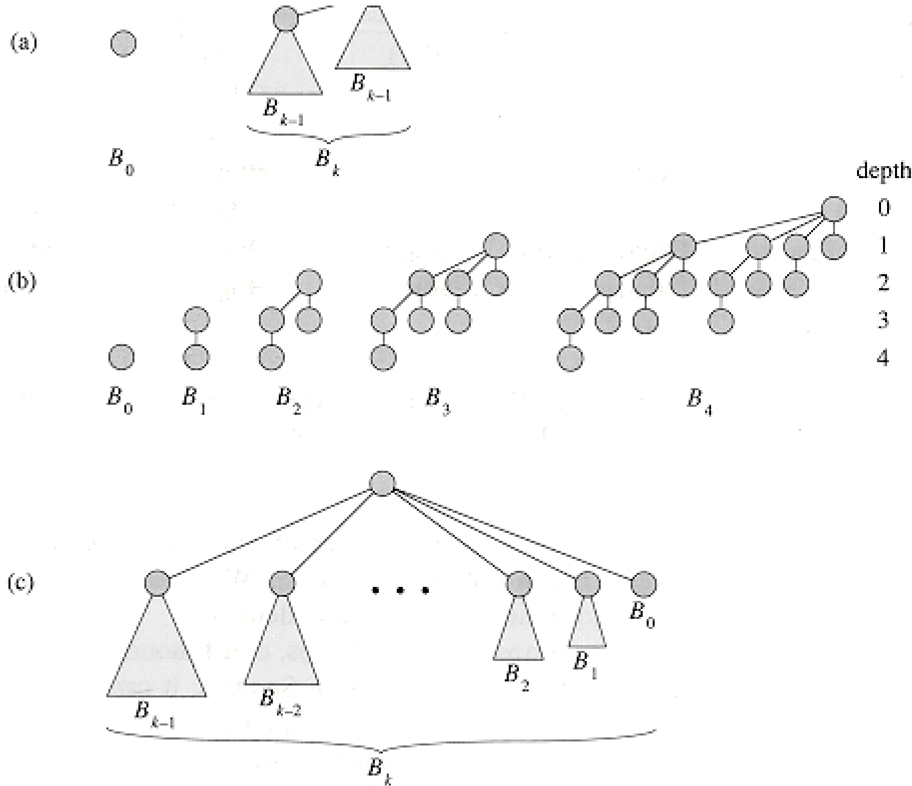
Step 2: Rotate node and new child (AVL)



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

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

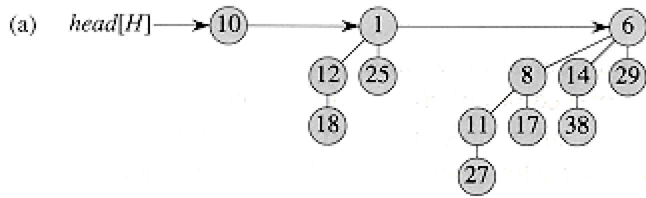
Some properties of binomial trees are given by the following lemma.

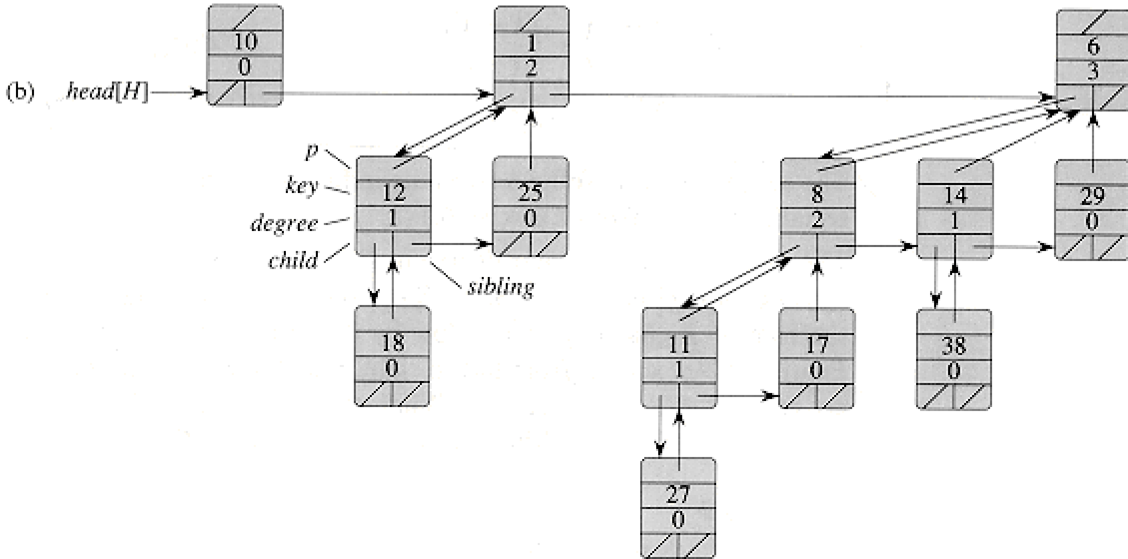


A ***binomial heap H*** is a set of binomial trees that satisfies the following ***binomial-heap properties.***

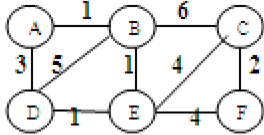
1. Each binomial tree in *H* is ***heap-ordered:*** the key of a node is greater than or equal to the key of its parent.

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





15.i)Compute the minimum spanning tree for the following graph (8)(NOV/DEC 2015)



F

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

F

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

F

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

F

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

F

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

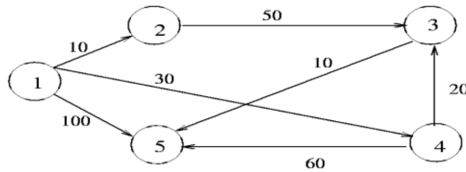
F

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

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

1. GPS Navigation systems: Navigation systems such as the Google Maps, which can give directions to reach from one place to another use BFS. They take your location to be the source node and your destination as the destination node on the graph. (A city can be represented as a graph by taking landmarks as nodes and the roads as the edges that connect the nodes in the graph.) BFS is applied and the shortest route is generated which is used to give directions or real time navigation.
2. Computer Networks: Peer to peer (P2P) applications such as the torrent clients need to locate a file that the client (one who wants to download the file) is requesting.This is achieved by applying BFS on the hosts (one who supplies the file) on a network. Your computer is the host and it keeps traversing through the network to find a host for the required file (maybe your favourite movie).

15.b)Illustrate the Dijkstra's algorithm algorithm for finding the shortest path with the following a sample graph(16)



Initially:

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

Iteration 1

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

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

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

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

Iteration 2

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

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

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

Iteration 3

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

$D[5] = \min(90, D[3] + C[3, 5]) = 60$

teration 4

Select w = 5, so that S = {1, 2, 4, 3, 5}

$D[2] = 10$

$D[3] = 50$

$D[4] = 30$

$D[5] = 60$

ii) Illustrate the comparison of Floyds algorithm with dijkstra's algorithm(4)

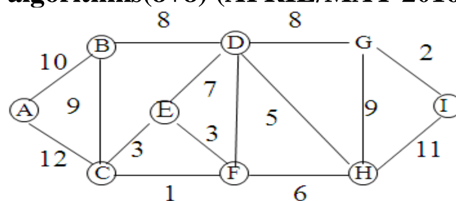
Dijkstra's Algorithm:

- Finds shortest path from a given startNode to all other nodes reachable from it in a digraph.
- Assumes that each link cost $c(x, y) \geq 0$.
- Complexity: $O(N^2)$, N = #(nodes in the digraph)

Floyd's Algorithm:

- Finds a shortest-path for all node-pairs (x, y).
- We can have one or more links of negative cost, c(x, y)
- Complexity: $O(N^3)$, where N = #(nodes in digraph).

12.Find the minimum spanning tree for the given graph using both Prims and kruskal's algorithm and write that algorithms(8+8) (APRIL/MAY 2016)



- MST is a SPANNING Tree
  - Nodes of MST = Nodes of G
  - MST contains a path between any two nodes
- MST is a MINIMUM Spanning Tree

- Sum of edges is a minimum
- MST may not be unique
- Initialize
  - M = { }
  - vertex sets: {A},{B},{C},{D},{E},{F},{G},{H},{I}
  - edge list, sorted by weight: 1 - CF, 2- GI, 3- EF, 3 -CE, 5- DH, 6- FH, 7- DE, 8-BD, 9-BC, 9-GH, 10-AB, 11-HI, 12-AC
- (u, v) = (C, F)
  - C and F are in different vertex sets (ie {C} and {F}
  - Add CF to M: M = {CF} (omit the comma from the edge)
  - Merge {C} and {F}: {A},{B},{C,F},{D},{E},{G},{H},{I}
- (u, v) = (G, I)
  - G and I are in different vertex sets (ie {G} and {I}
  - Add GI to M: M = {CF, GI}
  - Merge {G} and {I}: {A},{B},{C,F},{D},{E},{G,I},{H}
- (u, v) = (E, F)
  - E and F are in different vertex sets (ie {E} and {C,F}
  - Add EF to M: M = {CF, EF, GI}
  - Merge {E} and {F}: {A},{B},{C,E,F},{D},{G,I},{H}
- (u, v) = (C, E)
  - C and E are both in {C,E,F}: don't add CE since it would create a cycle

A Minimum Spanning Tree (MST) is a subgraph of an undirected graph such that the subgraph spans (includes) all nodes, is connected, is acyclic, and has minimum total edge weight

```
struct Edge
{
   int src, dest, weight;
};

struct Graph
{
       int V, E;

   struct Edge* edge;
};

struct Graph* createGraph(int V, int E)
{
   struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
   graph->V = V;
   graph->E = E;

   graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

   return graph;
}

struct subset
{
   int parent;
```

```
    int rank;
};

int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);


    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

        else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V];  // Tnis will store the resultant MST
    int e = 0;  // An index variable, used for result[]
    int i = 0;  // An index variable, used for sorted edges

    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

       struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

        for (int v = 0; v < V; ++v)
    {
```

```
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

        while (e < V - 1)
    {

        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);

        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }

    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
                                                   result[i].weight);
    return;
}

int main()
{

    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

     graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
```

```
    graph->edge[4].weight = 4;

    KruskalMST(graph);
     return 0;
}
```