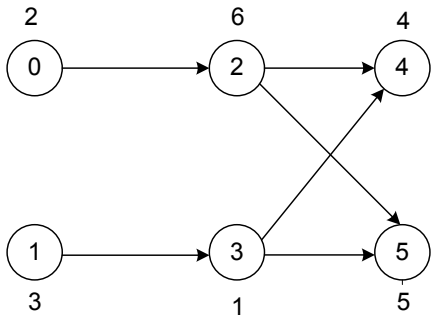
National University of Computer and Emerging Sciences, Lahore Campus

Course:	**Data Structure**	**Course Code:**	**CS201**
Program:	**BS(Computer Science)**	**Semester:**	**Fall 2016**
Duration:	**180 Minutes**	**Total Marks:**	**100**
Paper Date:	**16-Dec-16**	**Weight**	**40%**
Section:	**ALL**	**Page(s):**	**8**
		Roll No:	
Exam:	**Final (Solutions)**	**Section:**	

Instruction/Notes:
Answer in the designated space. You can get extra sheets for rough work but do not attach, they will not be marked.

QUESTION 1 [15 Marks]

You are given a directed graph ***G(V,E)*** in form an adjacency list and an array of integers, ***price*** of size |V| where ***price[i]*** is the price of vertex ***i*** (you can assume that vertices are labeled 0 .. |v|-1). You are required to write a function ***ComputeCost()*** that takes ***G*** and array ***price*** as parameters and computes the cost of each vertex. Cost of a vertex ***i*** is the minimum price of all the vertices reachable from ***i*** including ***i***. A vertex ***j*** is reachable from vertex ***i*** if there exists a path from vertex ***i*** to vertex ***j*** in ***G***. You can do this by computing an array of integers ***cost*** of size |V| where ***cost[i]*** would be the computed cost for vertex ***i*** . For instance in the graph below (with prices shown outside of each vertex), the cost values of vertex 0, 1, 2, 3, 4, and 5 are 2, 1, 4, 1, 4, and 5 respectively.



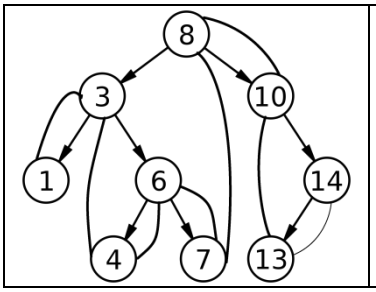

```
struct alnode{
   int v;
   alnode * next;
};

void getCostdfs(vector<alnode*> & G, vector<int> & price, int v, int & minp, vector<bool>&visited){
 //this is regular dfs with some added logic to compute min cost
   visited[v]=true;
   if(price[v]<minp)
      minp=price[v];
```

```
    //for all neighbors of v
    alnode * curr=G[v];
    while(curr!=NULL){
        int u=curr->v;
        if(!visited[u]){
          getCostdfs(G, price, u, minp,visited);
          curr=curr->next;
        }
    }
}


void ComputeCost(vector<alnode*> & G, vector<int> & price, vector<int> & cost){
//this method calls explore on each vertex

    int n=G.size();

    //perform a dfs starting at each node
    for(int i=0;i<n;i++){
        cost[i]=price[i];
        vector<bool> visited(n,false);
        visited[i]=true;
        getCostdfs(G,price,i,cost[i],visited);
    }

}
```

Note: the solution can be optimized by not computing the cost of any node more than once.

QUESTION 2 [15+10+5 = 30 Marks]

You are coding a binary search tree for an application which very often needs to go through the keys in sorted order. In order to avoid using the recursive traversal function, which goes through the nodes in order but uses the stack (due to recursion), you come up with an alternate strategy: you add an additional pointer ps to each node of the bst. This pointer points to the successor of the node in the tree, i.e. the next node in the tree in the sorted order of keys. In the following picture the ps pointers are indicated by curved lines: ps of 1 points to 3, ps of 3 points to 4, ps to 4 points to 6, ps of 6 points to 7, ps of 7 points to 8 and so on. The ps of 14 points to null.



New structure of the node is as follows:

```
struct node{
  int key;
  node* lchild, * rchild, * ps;
}
```

a. Write a recursive function ***connectSuccessors*** which goes through the entire tree once and connects all the ps pointers appropriately. Be very careful in picking the correct parameters for your function. Static or global variables are not allowed.

```
//recursive method (private in class bst)

void connectSuccessors(node * curr, node *& prev){
      if(curr!=NULL){
         connectSuccessors(curr->lc,prev);

         if(prev!=NULL)
            prev->ps=curr;

         prev=curr;

         connectSuccessors(curr->rc,prev);
      }
  }

//wrapper method  (public in class bst)
void connectSuccessors(){
      node * prev=NULL;
      this->connectSuccessors(root,prev);
      prev->ps=NULL;//set last ps to null
   }
```

b. Assuming all ps pointers have been connected, write an iterative function which prints all the nodes in sorted order, starting from the node with the smallest key and following the ps pointers.

```
//iterative printing method (public in class bst)
//if you've passed root as parameter, that's also fine.
void printInOrderItr(){
      node * curr=root;
      if(curr==NULL) return;
      while(curr->lc!=NULL){
         curr=curr->lc;
      }
      while(curr!=NULL){
         cout<<curr->key<<" ";
         curr=curr->ps;
      }
  }
```

c. Let's suppose we now want to insert a new key x into this tree of n keys and height h. Obviously the insert itself will take O(h) time. But we will also have to update the ps pointers. How much time will the insert function take now, using the most efficient possible method. Give a three line argument at most.

It takes O(h) to insert into tree
It takes further O(h) to find the successor, and the predecessor of a key in the tree. This can be done using two recursive methods which find the required nodes on the way up.

Overall time, O(h)

Note: O(n) solutions have been given partial credit based on the explanations.

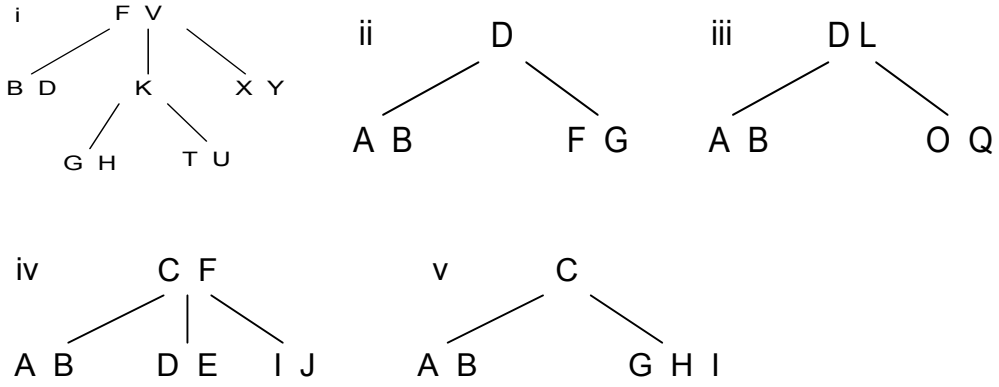
QUESTION 3 [15 Marks]

Add a function called ***updateKey*** to the class ***minHeap***. It accepts an index ***k*** and an integer value ***v***, and changes the key at index ***k*** to ***v***. Obviously, this can disturb the min-heap property, the function should resolve any violation and restore the min-heap property. Your function cannot call any other function and must work in ***O(lgn)*** where n is the number of keys in the heap.

```
//This is the skeleton of a solution, the parts not included here are obvious and were covered in class.
//function inside class minHeap
void updateKey(int i, int v){

   //h is the array of the heap
   //n is the size of the heap

   if(i<1 || i>n) return;

   //check the type of violation

   if(h[i]<h[i/2]){
      //violation with parent
      while(i>1){
         swap(h[i],h[i/2]);
         i=i/2;
      }
   }else if((i*2<=n && h[i]>h[i*2]) || ((i*2+1)<=n&&h[i]>h[i*2+1])){
      //violation with a child
      //reproduce code for minHeapify at i (either iterative or as separate recursive function)
   }
   }
```

QUESTION 4 [5*8 = 40 Marks]

a. which of the following are not valid 2-3 trees and why?. Each character is a key value or data item in the tree.



Following trees are invalid:
i) All leaves are not at the same level.
iii) D and L should have a middle child.
v) Right child of C has an extra key (at most two keys are allowed)

b. given a pointer ***p*** of a node in a singly link list which is not tail, how can you delete the node pointed by ***p*** in ***O(1)*** time

```
p->data=p->next->data;
p->next=p->next->next;
delete p->next;
//the only problem is when p is the tail node, in that case you will have to get the previous pointer
```

c. Given a max heap where will you find the minimum element of the data set?

The minimum element must be a leaf. In the array, it can be any of the elements from n/2+1 to n. (If it were not a leaf, then it would not be the min since its child must be smaller than it is).

d. Given two integer key values ***x, y*** and the root of an integer Binary Search tree, how will you determine the least common ancestor of the nodes containing ***x*** and ***y***. Least common ancestor of two nodes ***a*** and ***b*** is the node which is common ancestor of both ***a*** and ***b*** and has minimum height. Give 3-4 lines answer.

//Some of you have suggested storing the entire paths from root to each node and then comparing the //paths for the least common ancestor. I have considered that correct, and given credit.

- Perform recursion on the tree such that each node returns two flags fx and fy: fx is set if we have already seen x below this node, and similarly for fy.
- On the way back in recursion (post order), say in node z:
- Collect or-ing of (fx,fy) pairs returned by both children
- if z=x set fx
- if z=y set fy
- if both fx and fy are set note z as the ancestor, ignore if a z is already noted.

e. Suppose you have a 3-ary heap instead of binary heap. i.e. each node in the tree can have three children (left, middle and right). How would you find the indices of left, middle and right child of a node at placed position ***i*** assuming that heap is stored level wise in an array and first element of the heap is at position 1?

For a heap starting at index 1:
Left child of i: 3i-1
Middle child of i: 3i
Right child of i: 3i+1

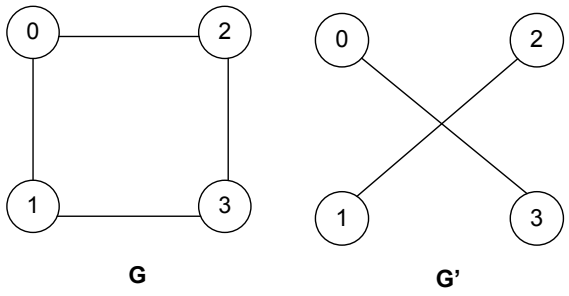
f. You want to convert three sorted arrays ***A, B*** and ***C*** into a single balanced BST. What is the fastest time in which you can do it, and how? No more than 3 lines of answer.

Merge three arrays into a single array in O(n)
Reclusively pick the middle elements as the root of the sub-tree with left and right sides being the left and right child trees. This will also take O(n)
So the entire process will take O(n)

g. We want to count the number of edges in a graph G=(V, E). How long will this take as a function of |V| and |E| in big Oh terms, if we used i) an adjacency list format ii) an adjacency matrix format.

Adjacency List: O(|V| + |E|), for each vertex count its neighbors
Adjacency Matric: O($|V|^2$), for each vertex look at all |V| columns and count the 1s

h. Suppose you are given an adjacency list of an undirected graph G(V,E) and you want to compute the complement G'(V,E') of G. The complement graph G' has same set of vertices as of G but the set of edges E' is all edges (i, j) that are not in E. Below is an example of G and G'. Assume that all the neighbors in the adjacency list of G are in sorted order, How will you efficiently compute the adjacency list of G' and what would be its time complexity in terms of big-oh? Give an algorithm in words using a bulleted list of actions.



//Let the original list be L1
//We create a second adjacency list L2, for the complement, and populate the edges in it.

(1) For each vertex x in V
(2) For each vertex y in V
(3) If {x,y} is not an edge in L1
(4) Add edge {x,y} in L2

This is a worst case $|V|^3$ algorithm.
Anything better also gets full credit. For example, using a hash-table as follows:

i) For all x in V
ii) Create hash table h
iii) Go through all neighbors of x in L1 and hash-them in h.
iv) For all y in V that are not in h
v) add {x,y} to L2

Using perfect hashing this will bring down the running time to O($|V|^2$)