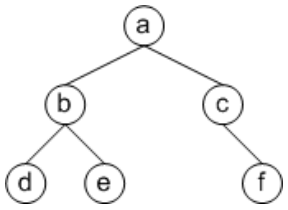
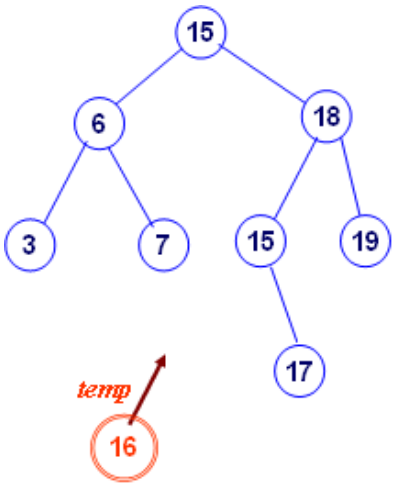
Tutorial 6

1. **Traverse the following binary tree: (a) in preorder; (b) in inorder; (c) in postorder. Show the contents of the traversal as the algorithm progresses.**



2. **(Understanding the execution of a recursive algorithm.) For the binary search tree below, trace step by step the execution of the algorithm *BSTinsert_recurs( ).***

```
BSTinsert_recurs (root, temp) {
    if (temp.data ≤ root.data) {
       if (root.left = = null)
          root.left = temp
       else
          BSTinsert_recurs (root.left,
temp);
    }
    else {  // goes to right
       if (root.right = = null)
          root.right = temp
       else
          BSTinsert_recurs (root.right,
temp);
    }
}
```



3. **The following algorithm seeks to compute the number of leaves in a binary tree. Is this algorithm correct? If it is, prove it; if it is not, make an appropriate correction.**

```
Input: T (i.e. a binary tree)
Output: the number of leaves in T

Algorithm LeafCounter (T);
        // Computes recursively the number of leaves in a binary tree
        if T == NULL then
                return 0;
        else
                return LeafCounter(T.left ) + LeafCounter(T.right );
```

4. **(Understanding the design of a recursive algorithm.) The binary tree is recursive in nature. The tree traversal algorithms that we discussed in the lecture exemplify the basic fact that we are led to consider recursive algorithms for binary tree. That is, we process a tree by processing the root node and (recursively) its subtrees. Based on this understanding and following the definition of the height of a given binary tree, devise an algorithm for calculating the height of a tree.**

Tutorial 7

1. **Assume that the following algorithms are available for binary trees, write an algorithm *depth* that computes the depth of a node *v* of the tree.**

   // Returns whether the tree T is empty.
   boolean isEmpty(T);

   // Returns the parent of a given node.
   parent(node v);

   /** Returns whether a given node is the root of the tree.
   boolean isRoot(node v);

2. **Write a linear-time algorithm that determines whether a binary tree is a binary search tree.**

3. **Write a *siftup* algorithm for a max-heap. The input to *siftup* is an index i and a max-heap structure in which the value of each node is greater than or equal to the values of its children (if any), except for the node at index i which has a value which is greater than its parent. siftup restores the max-heap.**

4. **For a heap of size n, show that the time complexity of applying heapify to it is $O(n)$.**

Tutorial 8

1. **Let A be the following array.**

16	23	31	20	4

(i) Show the array A after calling heapify on it to produce a max-heap.

(ii) Starting from the array in (i), trace the steps of the heapsort algorithm on A.

2. **Trace the execution of the partition algorithm to show how it partitions the array: 'N', 'A', 'N', 'Y', 'A', 'N', 'G', 'U', 'N', 'I' in the alphabetical order.**

3. **Show the steps of quicksort on the array:**

12, 30, 21, 8, 6, 9, 1, 7

4. **Consider performing counting sort on the following array:**

5	7	5	1	3	7	6	3	1	6	6	5	3

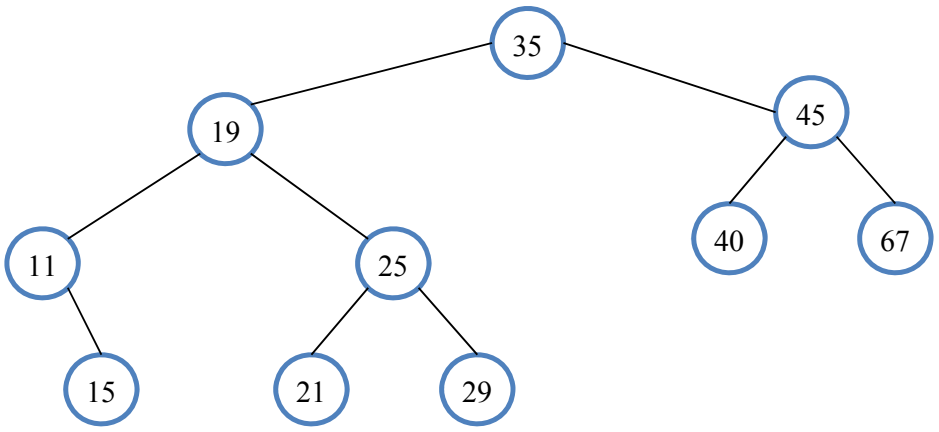
(i) What does the final count array look like?
(ii) Use the count array to determine the sorted array.

Tutorial 9

1. **Write an algorithm that, given an array *A* consisting of a sorted subarray of size *m* followed by a sorted subarray of size *n*, merges them into *A* using an extra array *B* of size min [*m*, *n*]. What's the worst-case time complexity of the algorithm?**

2. **Write a divide-and-conquer algorithm that counts the total numbers of nodes in a binary tree.**

3. **Consider the AVL tree shown below. Show the steps taken to balance the tree when**
    1. **14 is inserted**
    2. **23 is inserted**
    3. **70 is inserted**



Tutorial 10

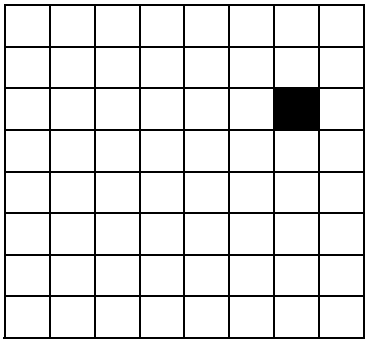
1. **Trace the steps of radix sort on the following sequence of numbers, each with 4 digits.**

   5890, 6204, 8267, 6850, 4668

2. **An *n*-element array *A* contains only the numbers 0, 1, 2. Write an *O*(*n*) algorithm to sort the numbers. Legal operations on the data are swapping two elements in the array.**

3. **A tromino is an L-shaped tile formed by** 1**-by-**1 **adjacent squares. The problem is to cover any** *n***-by-***n* **chessboard with one missing square (anywhere on the board) with trominos, where** *n* **is a power of** 2**. Trominos should cover all the squares except the missing one with no overlaps. Design a divide-and-conquer algorithm for this problem.**

A tromino



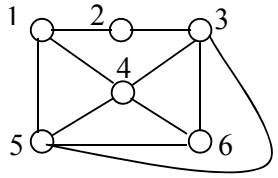
Tutorial 11

1. **Find in the following array the index of the element of 22 by applying the binary search algorithm. How many times is the binary search algorithm faster than the linear search in this case?**

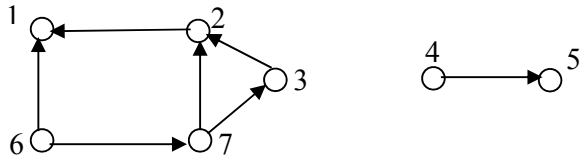
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

2. **Write an algorithm for the recursive implementation of binary search.**

3. **Draw adjacency lists for the graph**



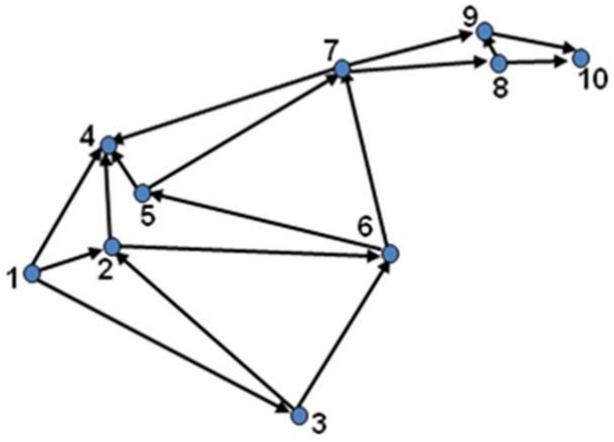
4. **Draw adjacency lists for the digraph**



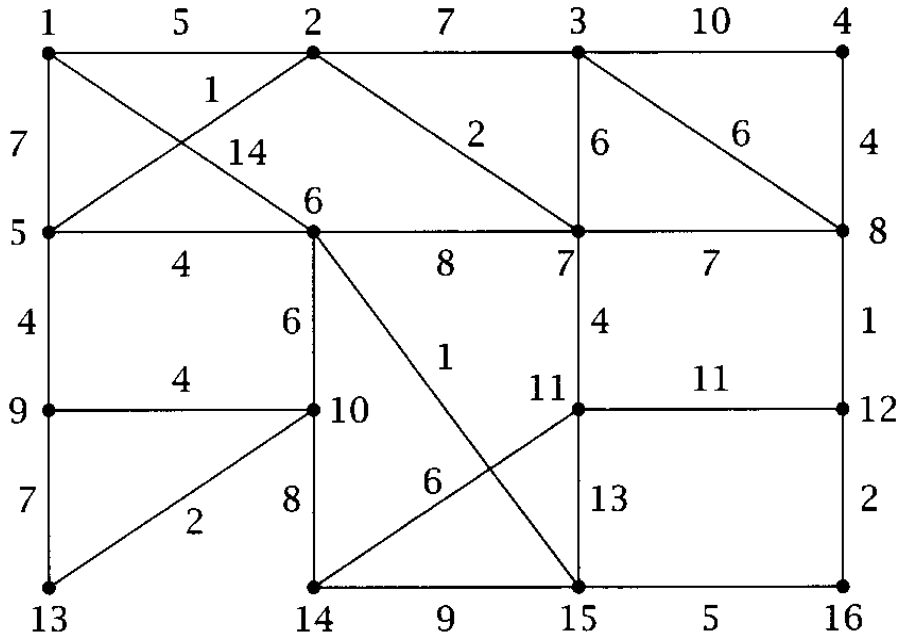
5. **Write an algorithm that prints the indegree and outdegree of every vertex in a digraph, where the digraph is represented using adjacency lists.**

Tutorial 12

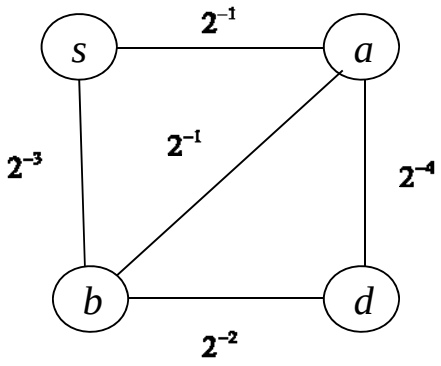
1. **Show one topological sort of the following DAG.**



2. **Find the minimum spanning tree of the following graph by using *Kruskal*'s algorithm.**



3. **Consider a weighted graph G. If (u,v) is an edge in the graph, let $w(u,v)$ denote its weight. Suppose that all edge weights are between 0 and 1. Given a starting vertex s and a destination vertex d, we wish to find a path $(s,u_1,u_2,\ldots,u_m,d)$ with *maximum edge weight product* $w(s,u_1)\times w(u_1,u_2)\times\cdots\times w(u_m,d)$. Suggest an efficient algorithm to do this and trace its steps for the following graph.**



4. **Let G be a connected weighted graph and v be a vertex in G. Suppose that the weights of the edges incident on v are distinct. Let e be the edge of minimum weight incident on v. Show that e is contained in every minimal spanning tree.**

5. **Suppose that $G=(V,E)$ is a tree represented by an adjacency list. Write in pseudo-code an algorithm that constructs the adjacency list for a new graph $G'=(V,E')$ with the same set of vertices V as G, and with edges between any two vertices if and only if they are 2 hops away in G, i.e., G' contains the edge (u,v) if and only there is a path of length 2 in G connecting u and v.**