Final

Note

This is a closed-book exam. Each problem accounts for 10 points, unless otherwise marked.

Problems

1. Below is a partition procedure that we discussed in class as part of the quick sort algorithm.

```
partition(theArray: ItemArray, first: integer, last: integer): integer
  pivotIndex = last
  pivot = theArray[pivotIndex]

  indexFromLeft = first
  indexFromRight = last - 1
  done = false
  while (not done) {
    while (theArray[indexFromLeft] < pivot)
      indexFromLeft = indexFromLeft + 1

    while (theArray[indexFromRight] > pivot)
      indexFromRight = indexFromRight - 1

    if (indexFromLeft < indexFromFight) {
      Interchange theArray[indexFromLeft] and
                  theArray[indexFromFight]
      indexFromLeft = indexFromLeft + 1
      indexFromRight = indexFromRight - 1
    }
    else
      done = true
  }

  Interchange theArray[pivotIndex] and theArray[indexFromLeft]
  pivotIndex = indexFromLeft

  return pivotIndex
```

   Apply the partition procedure to the following array. Show the contents of the array after each Interchange operation.

1	2	3	4	5	6	7	8	9	10	11	12
10	9	4	6	11	7	5	1	2	12	3	8

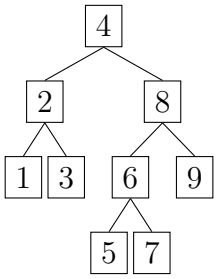
2. Consider the partition procedure again in the preceding problem. Suppose we now want to use as the pivot the first, instead of the last, entry of the array. More specifically, let us change the line "`pivotIndex = last`" into "`pivotIndex = first`". (The option of interchanging the first and the last entries and then following the same procedure is thus excluded.) What other changes should be made to the rest of the procedure?

3. Design an algorithm that, given an array of positive and negative numbers, rearranges the array entries so that all negative numbers appear before the positive numbers. Please present your algorithm in suitable pseudocode. Give a performance analysis of your algorithm. The more efficient your algorithm is the more points you will be credited for this problem.

4. Rearrange the entries in the following array so that it becomes a (max) heap. You may proceed in either a "top-down" or a "bottom-up" fashion. Show the contents of the intermediate array after an entry is added to the processed region.

1	2	3	4	5	6	7	8	9	10	11	12
8	2	4	6	11	7	5	1	9	12	3	10

5. Below is a binary tree with 9 nodes, each storing an integer key value.



   For each of the following orders of traversal, write the sequence of nodes (represented by their key values) visited in that order.

   (a) preorder

   (b) postorder

6. This problem concerns the notion of "inorder successor" in a binary search tree.

   (a) What is the inorder successor of a node? Why is this concept needed?

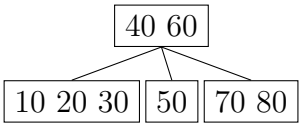
   (b) Give an algorithm (in suitable pseudocode) that finds the inorder successor of a given node.

7. Given a binary search tree, you may save the key values in a file using the preorder traversal and later restore the original tree by inserting the key values in the order saved into an empty tree.

(a) Prove by induction (or give convincing arguments) that the statement above indeed is true.

(b) Given a sequence of key values, if we insert the key values into an empty tree and then traverse the tree in preorder, will we get the original sequence of key values? Please justify your answer.

8. Below is a 2-3-4 tree with 4 nodes (including the root and its three children which are also leaves) and 8 key values.



Suppose we now want to insert 45, 25, 75, and 90 (in this order) into the tree. Please show the resulting tree after the completion of each insertion. (Note: Recall that the insertion algorithm for a 2-3-4 tree splits every 4-node encountered on the way down the tree from the root to the leaf where the input data item is to be placed.)

9. Consider the 2-3-4 tree again in the preceding problem. Give two distinct sequences of key values such that insertions of the key values in each sequence result in the tree as shown above.

10. Below is the algorithm we discussed in class for the BFS traversal of a graph.

```
bfs(v: Vertex)
   q = a new empty queue
   q.enqueue(v)
   Mark v as visited

   while (!q.isEmpty()) {
     w = q.peekFront()
     q.dequeue()

     for (each unvisited vertex u adjacent to w) {
       Mark u as visited
       q.enqueue(u)
     }
   }
```

Modify the code to use a stack so that it becomes an algorithm for the DFS traversal of a graph. Be careful about when the marking of an unvisited node should be done.