# Ve281 Data Structures and Algorithms
# Written Assignment Four

**This assignment is announced on Nov. 12th, 2016. It is due by 3:40 pm on Nov. 21st, 2016. The assignment consists of six problems.**

1. Min Heap

   (a) Suppose that we are inserting the keys 3, 1, 4, 1, 5, 9, 2, 6, 5, 4 **one by one** into an initially empty min heap. Show the resulting min heap in the form of a tree.

   (b) For the min heap you obtained for Problem (1a), show the resulting heap after calling two `dequeueMin` operations.

2. Min heap initialization

   Given a sequence of keys 3, 9, 7, 2, 5, 2, 8, 6, 1, 4, show the resulting min heap if we initialize it with the efficient algorithm we talked in lecture that takes $O(n)$ time complexity on an array of $n$ elements. Show the intermediate steps in the form of a tree. Do not just write the final result.

3. Binary search tree

   (a) Suppose that we insert a sequence of keys 4, 9, 2, 5, 1, 7, 6, 3, 8 into an initially empty binary search tree. Draw the resulting tree.

   (b) Suppose that we further delete the key 5 from the tree you get in Problem (3a). Draw the resulting tree.

   (c) Suppose that we further delete the root from the tree you get in Problem (3b). Draw the resulting tree.

4. Given a binary tree, in which each node is associated with an integer key, it may not possess the binary search tree property. Describe a **most runtime-efficient** algorithm that determines whether such a tree is indeed a binary search tree. Also, tell us what the runtime of your algorithm is. Show us why your algorithm is a most runtime-efficient one.

   Note: You can describe your algorithm in English. However, if you find that writing pseudo-code/code is helpful for your illustration, feel free to do so.

5. Suppose the node of a binary search tree is defined as follows.

```
struct node {
    Key   key;   // key
    node* left;  // left child
    node* right; // right child
};
```

Implement the following function which gets the predecessor of a given key in the tree:

```
node* getPred(node* root, Key key);
// REQUIRES: The tree rooted at "root" is non-empty.
//           "key" is in the tree rooted at "root".
// EFFECTS:  Return the predecessor of "key" in the tree rooted at "root".
//           Return NULL if there is no predecessor
```

You can assume the following function is availabe:

```
node* findMax(node* root);
// REQUIRES: The tree rooted at "root" is non-empty.
// EFFECTS:  Return the node with the maximal key in the tree rooted
//           at "root".
```

6. Suppose nodes $A, B, ..., J$ are located on a 2-D plane shown in Figure 1. We insert these nodes in the order $A, B, ..., J$ into a $k$-d tree. Show the final tree. Assume the comparison dimension of the root is the $x$ dimension.

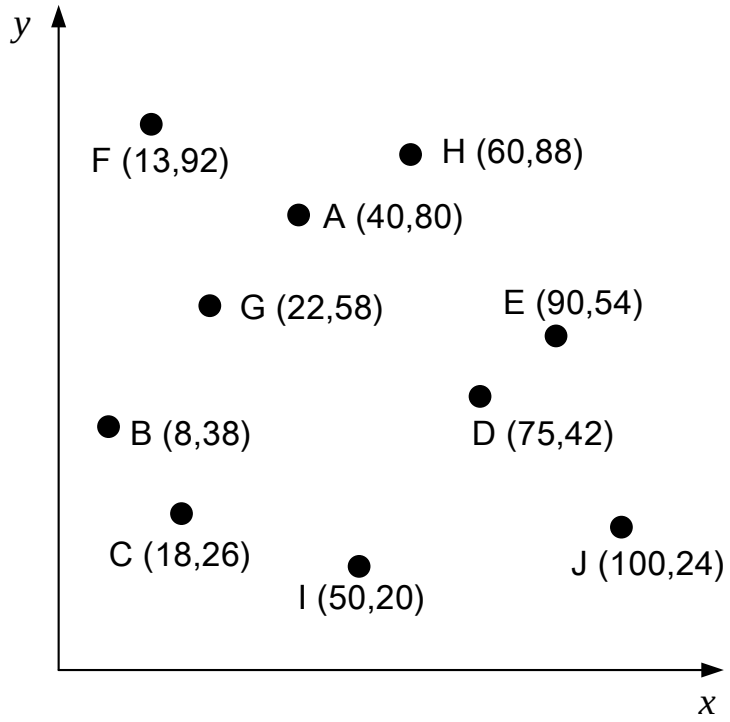


Figure 1: The locations of a number of nodes in a 2-D plane.