Problem Set 3

Name: Subhankari Mishra
Assigned: June 10
Due: June 17

PROBLEM 1.

Suppose that you are given the problem of returning in sorted order the K smallest elements in an array of size N, where K is much smaller than log(N), but much larger than 1.

Describe how quicksort can be adapted to this problem.

Ans:

1. Once the partition point is found, do the quick sort for the first section of the array.
2. Check if the number of elements from beginning of the array to the partition point is less than K.
   If yes, change K to K – total number of elements already sorted by the first sort and then do the quick sort on the second section of the array.
3. Finally return the first K elements of the partially sorted array.

```
Get_K_smallest(A, K){
        QuickSort(A, L, U, K);
        Return first K elements of the partially sorted list A
}

QuickSort(A, L, U , K){
        If(U – L < 7) insertion sort (A, L, U)

        Else{
                M = Partition(A, L, U)
                        QuickSort(A, L, M-1, K)
                If(M – L + 1 < K)
                        K = K – (M – L + 1)
                        QuickSort(A, M + 1, U , K)
                EndIf

        EndIf

        }
}

Partition(A, L, U){
        IPIVOT = some value between L and U
        PIVOT = A[IPIVOT]
        SWAP(A, L, IPIVOT)
        J = L + 1
        K = U
        WHILE(J < K){
                WHILE((A[J] < PIVOT) & J < K) J++;
                WHILE((A[K] > PIVOT) & J < K) K--;
                IF (J >= K) Exit Loop
                SWAP(A,J,K)
```

```
                J++;
                K--;
            }

    If (A[J] > PIVOT) J--;
            SWAP(A, L, J)
            RETURN J;
    }

}
```

PROBLEM 2.

Suppose that we modify the standard definition of a binary search tree to add a field N.size at each node, which records the size of the subtree under N (including N itself).

A. Explain how to modify the procedure for adding an element X to a tree. Be sure to consider both the case where X is not yet in the tree and is added, and the case where X is already in the tree, and the tree remains unchanged.

   You only need to describe the changes that are made to the standard algorithm; you do not have to repeat the standard algorithm.

   Ans:
   1. If the element is found then no change is needed.
   2. If the element is not found then add the element as per the standard algorithm and assign one to the size field of the node and increment the size field by 1 of the corresponding parent node at each level moving up the tree till the root including the root.

B. Explain how to modify the procedure for deleting an element. As in (A), consider both cases.
   Ans:
   1. If the element is not found then no change is needed.
   2. If the element is found then:
      i. If the element is a leaf node delete the node and decrement the size field by 1 of the corresponding parent nodes at each level moving up the tree till the root including the root node.
      ii. If the element has one child then follow the steps as per the standard algorithm decrement the size field by 1 of the corresponding parent nodes at each level moving up the tree till the root including the root node.
      iii. If the element has two children, then follow the steps as per the standard algorithm to delete the element.
      Copy the size field of the node being deleted to the size field of the replacing node then decrement the size field by 1 then decrement the size field by 1 of the corresponding parent nodes at each level moving up the tree till the root including the root node.

C. Describe a procedure for finding the K^{th} largest element in the tree.
   Ans:
   Below is the recursive procedure for finding the K^{th} largest element in the tree.
   The procedure takes a tree and K as input and maintains a global variable S which is added to the value of the node's size field while comparing it with K. S is initialized to zero. If there is not right child for a node we consider the size of the right subtree as zero.
   The logic uses the size of the nodes and the position of the node whether right or left or the parent to calculate the node's rank.

Below is the description of the algorithm:

1. If K is equal to the sum of Node.size and S then return the value of the leftmost leaf of the leftmost subtree.
   If there is no left subtree return the value of the node.

2. If K is less than the sum of Node.size and S, then check:
   i. If K is less than the sum of size of right subtree and S, call the same procedure with tree as the right subtree.
   ii. If K is equal to the sum of size of right subtree, S and 1, then return the value of the node.

   iii. If K is greater than the sum of size of right subtree, S and 1, increment the value of S by one plus size of right subtree (right subtree + 1 + S) and call the same procedure with tree as the left subtree.

3. If K is greater than the sum of Node.size and S, return error.

Below is the algorithm for the above description:

```
Let S be a global variable initialized to zero.
Validate K should be less than or equal to the size of the tree.
Find_K^th_largest(Tree, K){
    If(K = (root.size + S))
        If(Tree.left == Null)
            Return root.value
        Else
            Return the value of the leftmost leaf of the leftmost
            subtree
        Endif
    Else If(K < (root.size + S))
        If(Tree.right != Null)
            If( K < (Tree.right.root.size + S))
                Find_K^th_Largest(Tree.right, K)
            Else If( K = (Tree.right.root.size + S + 1))
                Return root.value
            Else If( K > (Tree.right.root.size + S + 1))
                S = S + Tree.right.root.size + 1
                Find_K^th_Largest(Tree.left, K)
            Endif
        Else
            If(K = (S + 1))
                Return root.value
            Else If ( K > (S+ 1))
                S = S + 1
                Find_K^th_Largest(Tree.left, K)
            Endif.

        Endif
    Else If(K > (root.size +S))
        Return Error.
    Endif
}
```

D. Describe a procedure for finding the number of elements in the tree less than X .
   Ans: Below is the recursive procedure for finding the number of elements in the tree less than

X.
The procedure takes a tree and X(a value) as input.
The logic uses the size field at each node the value of the node to calculate the no. of elements less than or greater than the given value.
Below is the description for the algorithm:

Let S be a global variable initialized to the value of the size field of the root node.

1. Compare the given value with the root.value.
    a. If the given value is greater than root.value, call the same procedure on the right subtree. If there is no right subtree, return the S.
    b. If the given value is equal to the root.value, then return S – size of right subtree – 1.
       If there are no right subtree, then return S – 1.
    c. If the given value is less than root.value, then set S = S – size of right subtree – 1 if right subtree is not NULL, else set S = S – 1.
       And if left subtree no left subtree return S else call the same procedure on the left subtree.

Let S be a global variable initialized to the value of the size field of the root node.

```
Find_Less_Than(Tree, X){

    If(X < (root.value))

        If (Tree.right != NULL)
            S = S – Tree.right.root.size – 1
        Else
            S = S – 1
        Endif

        If(Tree.left == NULL)
            Return S
        Else
            Find_Less_Than(Tree.left, X)
        Endif

    Else If (X > (root.value))
        If(Tree.right == NULL)
            Return S
        Else
            Find_Less_Than(Tree.right, X)
        EndIf.

    Else If (X == (root.value))
        If(Tree.right == NULL)
            Return S – 1
        Else
            Return S – Tree.right.root.size – 1
        Endif
    EndIf
```

All of these procedures should run in time proportional to the height of the tree.

PROBLEM 3

Suppose you have two binary search trees P and Q. Let $|P|$ and $|Q|$ be the number of elements in P and Q, and let H_P and H_Q be the heights of P and Q. Assume that that is, $H_P \ll H_Q \ll |P| \ll |Q|$ and

A. Give a destructive algorithm for creating a binary search tree containing the union $P \cup Q$ that runs in time $O(|P|^2)$ in the worst case.

Ans:

```
Merge(P, Q){
        While (P.root != Null){          ----------|P|
            L = Delete_smallest(P)  ---------- HP
            Add(L,Q)                ----------  HQ + |P|
        End While
    }

Delete_smallest(K){     ---------- HP
        Let M be the leftmost node of the leftmost subtree in K
        If( M has a right child)
            N = M
            Delete M from the tree and replace it with the root node of the right subtree.
            Return N
        Else
            N = M
            Delete M
            Return N
        Endif
}
Add(X, S){

        Find where X should go as in S
        IF(Not There)
            Stick X There
        EndIf
}
```

The worst case running time for the above algorithm would be $O(|P| * (H_P + H_Q + |P|))$ which is equal to $O(|P| * |P|) = O(|P|^2)$ as $H_P << H_Q << |P|$.

B. Assume now that it is known that the largest element of P is less than the smallest element of Q. Give a destructive algorithm for creating a binary search tree containing the union $P \cup Q$ that runs in time H_P.

```
Merge(P, Q){
        Find the largest node in P, which is the rightmost node of the rightmost subtree.
        Add Q as the right child of the largest node in P.
}
```

And as we already know that the largest element of P is less than the smallest element in Q there cannot be any duplicates.

The above algorithm runs in time O(HP).

C. (1 point extra credit). Find a solution to part (A) that runs in time $O(|P| \cdot H_Q)$ in the worst case, and additionally guarantees that the height of the output tree is no greater than $H_P + H_Q$.

Ans:

```
Merge(P,Q){
        While( P != Null){ ------------------------------  |P|
                L = Delete_root(P)  ---------------------- HP
                Add (L,Q)  ------------------------- (HQ + HP) = HQ
        }
}

Delete_root(P){ ------------------------------------------------- HP
        M = root
        If(root has one child)
                Delete and Replace root with child
        Else If(root has two children)
                 Delete and Replace root with the root of the right subtree,
                 continue the shifting on the right subtree till the node which do
                 not have a right child is reached, then replace the node with root
                 of its left subtree if any.
        End If
}

Add(X,S){
        Find where X should go in S
        If (Not There)
                Stick X There
        EndIf.
}
```

The running time for this algorithm would be $O(|P|.(HP + HQ + HP)) = O(|P|.HQ)$ as $HP << HQ$.

The height of the output tree is guaranteed to be HQ + HP in worst case as the delete_root method returns the parent node of the right subtree i.e. it supplies the values in an order which will form the same tree structure as P, if added to an empty tree, so in worst case the algorithm will form a similar tree as P under one of the leaf nodes of Q resulting into a height of HQ + HP of final tree.