ESO207: Data Structures and Algorithms	*Time: 90 min.*
Quiz 1	*Max. Marks: 100*

Instructions.

a. Please answer all parts of a question together. You may leave blank spaces if you wish to return later to complete parts of the question.

b. The exam is closed-book and closed-notes. Collaboration of any kind is ***not*** permitted. You may not have your cell phone on your person. Please use the mounted clock in the rear of the examination hall for time reference.

c. You may cite and use results done in the class.

d. Grading will be based not only on the correctness of the answer but also on the justification given and on the clarity with which you express it. Please be neat. Argue correctness of your algorithms.

Problem 1. *Order-Statistic Tree.* Consider an order-statistic tree which is a red-black tree with a *key* field and an augmented *size* field. For any tree node x, $x.size$ is the number of nodes in the sub-tree rooted at x. Assume that all keys in the tree are distinct.

1. Give an algorithm that takes an input parameter i and a node x and returns the ith successor of x in time $O(\log n)$ (independent of i). (25)
2. Explain the correctness and analyze the time complexity of the algorithm. (25)

Hint: Name the procedure Successor(x, i). If $x.right.size \geq i$, then the ith successor lies in the right sub-tree, and recurse appropriately. Otherwise, let y be the lowest ancestor of x whose left child is also an ancestor of x. If y is NIL, then the ith successor of x does not exist.
Now call Successor$(y, i - x.right.size - 1)$.

Soln. (Let us assume that $i \geq 1$.) We need the ith successor of node x. If x has a right child and $x.right.size \geq i$, then there are at least i nodes in the right subtree of x that come after x in the inorder traversal of the tree T. In this case, we need to search for the ith largest node in $x.right$. This is the same as OS-SELECT$(x.right, i)$, which can be used directly. Otherwise, there are less than i nodes in the right subtree of x. In that case, we find the lowest ancestor y of x whose left child is also an ancestor of x. This node y is the $x.right.size+1$th successor of x. Hence the problem reduces to finding the $i-$ $x.right.size$ -1th successor of y. This can be written as follows.

```
OS-SELECT(y, i) // Returns ith smallest node in subtree rooted at x.
1.  found = FALSE
2.  while not found and y ≠ T.nil
3.      count = y.left.size+1
4.      if  count == i
5.          found = TRUE
6.      elseif count ≤ i
7.      y = y.left
8.      else
9.          y = y.right
10.         i = i − count
11. return y
```

```
SUCC-ANCESTOR(x) // Finds the lowest ancestor that comes after x in inorder sequence.
1.  y = x
2.  while  y.p ≠ NIL and y = y.p.right
3.      y = y.p
4.  return y.p
```

```
SUCCESSOR(x, i)
1.  if  i == 0 return x
2.  if x.right.size ≥ i
3.      OS-SELECT(x.right, i)
4.  else
5.      y = SUCC-ANCESTOR(x)
6.      if  y == NIL
7.          return NIL
8.      else
9.          return SUCCESSOR(y, i − x.right.size − 1)
```

The algorithm makes at most one pass upwards from x till some node (which could be the root) and then makes one pass downwards from the right child of that node. This is because the algorithm moves up the tree during a call to SUCC-ANCESTOR(x) and down $x.right$ during the call to OS-SELECT($x.right, i$). Once OS-SELECT is called, the algorithm will terminate and find the desired successor. Hence, there may be multiple calls to SUCC-ANCESTOR followed by one (or zero) calls to OS-SELECT. Thus, the algorithm makes at most one upward pass from x to the root and one downward pass, for a total of at most $2h = O(\log n)$ nodes visited. The work done per node is $O(1)$. Therefore, total time taken is $O(\log n)$.

Alternative Solution. The problem can be reduced to the OS-SELECT and OS-RANK done in the class. Suppose we have to find the ith successor of x. Then, we can find the rank of x using the OS-RANK routine, which is say r. Then, we select the element with rank $r + i$ using OS-SELECT.

```
SUCCESSOR(x, i)
1.  return OS-SELECT(OS-RANK(T, x) + i)
```

Each of these routines takes time $O(\log n)$, and there are two calls, hence total time is $O(\log n)$.

Problem 2. Suppose there is a monetary system with m different coins of denominations $c_1, c_2, \ldots, c_m$. You have to devise an algorithm that can make change for a given amount n using minimum number of coins.

1. Let $S[i]$ be the smallest number of coins that can be used to make change for amount i. Design a recurrence relation for $S[i]$. Assume $S[0] = 0$. (*Hint*: The first coin could be of value $c_1, c_2, \ldots, c_n$.) (15)

$$S[i] = \begin{cases} \min_{1 \le j \le m : c_j \le i}(S[i - c_j] + 1) & \text{if } i > 0 \\ 0 & \text{if } i = 0 \end{cases}$$

2. Write pseudo-code for procedure that computes $S[n]$, given n. (10)

```
COIN-CHANGE(c[1...m], n) // find best change of n using coins in c[1...m].
1.  S[1...n] is an array of size n
2.  S[0] = 0
3.  for i = 1 to n {
4.      S[i] = ∞
5.      for j = 1 to m {
6.          if c[j] ≤ i {
7.              S[i] = min(S[i], S[i − c[j]] + 1) }
8.      }
9.  }
```

3. What is the additional information that you can store to output the required number $S[n]$ of coins of different denominations to obtain the change for n. Explain and write the pseudo-code that prints this output. (15)

We can modify COIN-CHANGE to keep track of the coin for which the minimum is attained in line 7 of the code. Keep an array $D[1 \ldots n]$ such that $D[i]$ stores the first coin that minimizes the change for i.

```
COIN-CHANGE(c[1...m], n) // find best change of n using coins in c[1...m].
1.  S[1...n], D[1..., n] are arrays of size n
2.  S[0] = 0
3.  for i = 1 to n {
4.      S[i] = ∞
5.      for j = 1 to m {
6.          if c[j] ≤ i {
7.              if S[i] > S[i − c[j]] + 1 {
8.                  D[i] = j      // one can also store c[j]
9.                  S[i] = S[i − c[j]] + 1) }
10.     }
11. }
```

```
PRINT-COIN-CHANGE(c[1...m], D[1...n], n)
1.  i = n
2.  if S[i] == ∞
3.      print Change for n is not possible
4.  else {
5.      print "Smallest change for" n " is coins of values respectively "
6.      while (i > 0) {
7.          print "c[D[i]] "
8.          i = i − c[D[i]]
9.      }
10. }
```

4. Analyze the complexity of your algorithm. (10)

   The complexity of the dynamic programming algorithm is $O(mn)$. The complexity of the printing procedure is $O(S[n])$.