ESO207A: Data Structures and Algorithms	*July-Nov 2017*
Mid-Semester Exam	*20th September, 2017*

Instructions. For questions in which algorithms are asked for, your answer should provide, (a) a clear description of the algorithm in English and/or pseudo-code, (b) A correctness proof, and (c) an analysis of the running time of the algorithm. Marks will be deducted for imprecise and unclear descriptions or arguments.
You may reference and use algorithms and their properties as described in the class or homework solutions and programming assignments

---

Problem 1. For problems requiring algorithms, give clear pseudo-code and explain the correctness and complexity arguments briefly but clearly. (6+6+6+7 = 25)

1 (a) Give reasons whether any array of 7 numbers can in general be sorted by a *comparison-sort* algorithm using 12 comparisons. ($2^8 = 256, 2^9 = 512, 2^{10} = 1024, 2^{11} = 2048, 2^{12} = 4096, 2^{13} = 8192, 2^{14} = 16384$).

Solution. As shown by the lower bound, one needs $\lceil \log_2 8! \rceil = \lceil \log_2 5040 \rceil = 13$. Hence 13 comparisons are necessary in general.

1 (b) You are given a segment of a red-black tree shown in Figure 1 that has resulted from the deletion of some node and is in violation of red-black color properties. Show how to fix this violation using at most one rotation and by possibly changing colors of some nodes. Darkly shaded nodes are black. Node A is doubly black and node D is colored red. The node B has color c and node E has color c', where, $c, c' \in \{\text{RED}, \text{BLACK}\}$.

Solution.

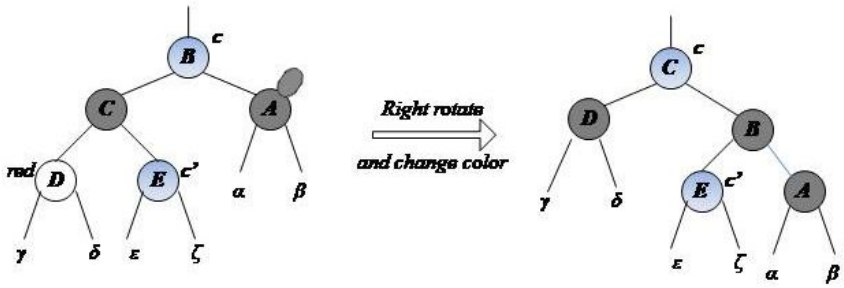

Figure 1: Fixing RB-tree fragment in violation of RB-tree coloring properties.

1 (c) Suppose you are given a randomized procedure PARTITION(A, p, r) that takes an array segment $A[p, \ldots, r]$ of numbers and returns two indices q and s such that $p \le q \le s \le r$ such that $A[p, \ldots, q-1]$ are all strictly less than $A[q]$; $A[q], \ldots, A[s]$ are all equal; and $A[s+1, \ldots, r]$ are all strictly greater than $A[q]$. Write a recursive definition of the function SELECT(A, p, r, i) that returns the ith smallest element in $A[p, \ldots, r]$, for any given i, where, $1 \le i \le r - p + 1$. ***Solution.***

```
SELECT(A, p, r, i)          // assumes 1 ≤ i ≤ r − p + 1
1.   (q, s) = PARTITION(A, p, r)
2.   if q − p + 1 ≤ i and i ≤ s − p + 1
3.         return A[p + i − 1]
4.   elseif   i ≤ q − p
5.         return SELECT(A, p, q − 1, i)
6.   else   return SELECT(A, s + 1, r, i − (s − p + 1))
```

1 (d) Given a binary tree on n nodes, outline an $O(n)$ time procedure that prints a leaf node that is furthest away from the root node (in terms of number of number of edges). Briefly argue correctness of the algorithm.

Solution. Use the breadth-first search procedure described in the solution to the maze problem in HW2. Start the breadth-first-search from the root of the tree. Enter the root node T into an empty queue and set $T.d = 0$. At each iteration until the queue is empty, dequeue a node u from the front of the queue. Enqueue all non-NIL children v of u and set $v.d = u.d + 1$. Return the last node whose dequeue makes the queue empty.

Time complexity is $O(n)$, as all edges and vertices are visited once. Note that the d attributes are unnecessary, since, vertices are inserted as per the $.d$ values and therefore are dequeued as per the d values. Thus, the vertex dequeued at the end is one of the furthest vertices. The proof of correctness of the algorithm follows from the lemmas proved in the solution to the maze problem in HW2.

```
FURTHEST_LEAF(T)
1.   Q = φ
2.   T.d = 0
3.   Q.ENQUEUE(T)
4.   while not ISEMPTY(Q)
5.         u = Q.DEQUEUE
6.         if   u.left ≠ NIL
7.               Q.ENQUEUE(u.left)
8.               u.left.d = u.d + 1
9.         if   u.right ≠ NIL
10.              Q.ENQUEUE(u.right)
11.              u.right.d = u.d + 1
12.  return   u
```

Problem 2. Non-dominated points (25)

A two-dimensional point (x, y) is said to dominate another two-dimensional point (u, v) if $x \geq u$ and $y \geq v$. Given a set P of points, a point $p = (x, y)$ is said to be a non-dominated point of P (also called a maximal point) if no other point q in P dominates p. See Figure 2 for examples. Given a set of n points P placed in arbitrary order in an array, give a time efficient algorithm FIND_NON_DOM(P, n) to find the set of all non-dominated points in P. *Notes*:

1. For simplicity, assume that no two points have the same x-coordinate or the same y-coordinate.
2. A point p is represented as a structure with two attributes $p.x$ and $p.y$. The set of points P, is represented as an array $P[1, \ldots, n]$ of points in arbitrary order.
3. You can find the index of the median of the points in $P[k, \ldots, l]$ by the x coordinate by using an informal statement like "let $i =$ MEDIAN(P, k, l) by x-coordinate" Similarly, a statement like "let $i =$MEDIAN(P, k, l) by y coordinate" can be used to find the index of the median of the points in $P[k, \ldots, l]$ by the y coordinates. These functions run in time $O(k - l + 1)$. The median of n points is returned as the $\lfloor (n + 1)/2 \rfloor$ th ranked item.
4. Full marks will be provided for a correct solution that takes $O(n \log n)$ time.
5. A correct solution of $O(n^2)$ time will earn only 10 points in total.

Figure 2: Dominated and non-dominated points in a point-set

Solution. The idea behind a divide and conquer solution is illustrated in Figure 3.

The top-level call is FIND_NON_DOM$(P, 1, n)$. The call to FIND_NON_DOM(P, k, l) returns a pair (N, r), where, N is an array of r points which are all (and only) the non-dominating points in the set $P[k, \ldots, l]$ *and are sorted* in increasing order of x-coordinate (and therefore, sorted in decreasing order of y-coordinate).

```
FIND_NON_DOM(P, k, l)
1.  if  l==k
2.        Create a new array N of size 1
3.        N[1] = P[k]
4.        return (N, 1)
1.  let m =MEDIAN(P, k, l) by x coordinate
2.  exchange P[l] with P[m]
3.  let s =PARTITION(P, k, l) by x coordinate
4.  (N_1, r) = FIND_NON_DOM(P, k, s)
5.  (N_2, u) = FIND_NON_DOM (P, s + 1, l)
6.  return  FIND_NON_DOM_MERGE(N_1, r, N_2, u)
```

The algorithm uses the procedure FIND_NON_DOM_MERGE(N_1, r, N_2, s) for merging two sets of non-dominating points, N_1 with r points and N_2 with s points, such that $N_2.x \geq N_1.x$ that is, for every $p \in N_1$ and every $q \in N_2$, $p.x \geq q.x$. For proof, refer to Figure 3. All points in N_1 that have y value lower than or equal to the highest y-point in N_2 (which is the first point in the sorted order by x) are dominated by the highest y point of N_2. Conversely, each point in N_1 with y value larger than the highest

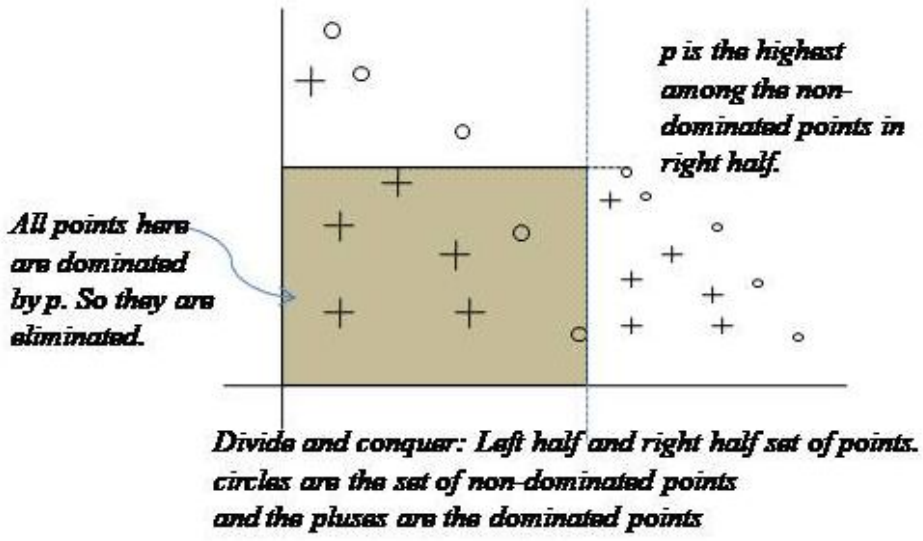

Figure 3: The set of points is divided into two almost equal parts by the median according to x coordinate. The set of non-dominated points in each set are computed. All the circled points, i.e., non-dominated points of the left half, lying in the shaded region on or below $p.y$ line, are eliminated. The non-dominating set consists of all points from the left-half non-dominating set above the shaded region and all the points in the non-dominating set of the second half.

y value point of N_2 is not dominated by it, and therefore by any other point of N_2, since they all have smaller y values. On the other hand, no point in N_2 can be dominated by a point in N_1 since the x values are larger.

FIND_NON_DOM_MERGE(N_1, r, N_2, s)
1. // first find the points p in N_1 with higher y-coordinate than the first point of N_2.
2. $t = 0$
3. **while** $t < r$ and $N_1[t+1].y > N_2[1].y$
4. $t = t + 1$
5. Create a new array $N[1, \ldots, t+s]$.
6. copy first t elements of N_1 into the respective first t positions of N.
7. Subsequently, copy all s elements of N_2 into positions $t+1, \ldots t+s$ of N.
8. **return** $(N, s+t)$

The time complexity of FIND_NON_DOM_MERGE (N_1, r, N_2, s) is $O(r+s)$. Going back to the function FIND_NON_DOM(P, k, l), in line 4, $r \leq s - k + 1$ and in line 5, $u \leq l - s$. Therefore, the call to FIND_NON_DOM_MERGE (N_1, r, N_2, u) takes time $O(l - k + 1)$. The time taken by MEDIAN and partition is again $O(l - k + 1)$ each. By taking the median, and assuming no repeated values, the two partitions are almost equal, the left one has size $\lceil n/2 \rceil$ and the right partition is $\lfloor n/2 \rfloor$. If $l - k + 1 = n$, we get the recurrence relation,

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$$

whose solution is $T(n) = O(n \log n)$.

Alternative Solution. This solution does not use the divide and conquer method. Let $P[1, \ldots, n]$ be sorted in increasing order by x coordinates, with ties broken by y coordinate in increasing order. Now scan P backwards from the point with highest x. Let $y_m = P[n].x$. Maintain the loop invariant that any point p in the backwards scan of P must have $p.y > y_m$ to be included in the non-dominating set. The initialization is obviously true.

To see the maintenance, suppose the invariant is true for n downwards to $k \geq 2$. Now consider $k-1$. Then, $y_m = \max_{t=k}^{n} P[t].y$. If $P[k-1].y \leq y_m$, then, $P[k-1]$ is dominated by at least one point in $P[k \ldots n]$ and we discard $P[k-1]$. Otherwise, $P[k-1]$ is included into N, which is filled in the backwards order (increasing order of y, and decreasing order of x).

```
FIND_NON_DOM(P, n)      //assumes n ≥ 1
1.  sort P[1, ..., n] in increasing order of x, ties broken in increasing order of y.
2.  Create a new array N[1, ..., n]
3.  y_m = −∞
4.  t = 0
5.  for  k = n downto 1
6.       if  P[k].y > y_m
7.           t = t + 1
8.           N[t] = P[k]
9.           y_m = P[k].y
```

The time required is dominated by the sorting in line 1, which requires $\Theta(n \log n)$ time.

Problem 3. Stack *Algorithm and Correctness: 20 Complexity 5*

You are given a sequence S consisting of the characters left parenthesis '(' and right parenthesis ')' only and of ***even*** length. Find the smallest number of parenthesis reversals that are needed to make the sequence a balanced parenthesis expression.

Suppose we start by designing the simplest program. A stack is used that is initially empty.

1. Scan the next input character.
2. If it is `'('` then push the input index (position in input string) on stack, and if it is `')'` then pop the stack (if it is non-empty), pair the character `'('` on top of the stack with the current input character `')'` and discard the pair.

Let us incrementally fix this program. There is one problem, namely, what if we see a `')'` and the stack is empty. At this point, we have no choice but to reverse it to a left parenthesis and push it back onto stack. Increment a variable *reversal_count* by 1. Initially, set *reversal_count* $= 0$.

1. Scan the next input character. If it is `'('` then push its index on stack.
2. If it is `')'` and the stack is non-empty, then pop the stack, pair them and discard the pair.
3. If it is `')'` and the stack is empty, reverse it to `'('`, push current index onto stack, and increment *reversal_count* by 1.

Return *reversal_count* + (stack height)/2.

After steps 1,2 and 3 repeated, we are at the end of the input. By the argument above, the reversal_count is so far minimum possible, since we reversed a right parenthesis to a left parenthesis only when necessary to avoid impossible expressions that cannot be balanced otherwise. Suppose the stack contains some k number of input positions of left parenthesis (the stack only contains left parentheses, and k may be 0), which are unbalanced. The minimum number of reversals needed to balance an expression with only k left parenthesis is $k/2$; just reverse the top $k/2$ left parentheses (other possibilities also exist). Note that k must be even, since, our input string is of even length, and each time, we pair a left parenthesis with a right one and then discard.

Notes: Let S be any sequence of left and right parentheses pairs. Suppose we consider the following pairing operation.

Identify a *matching* parentheses pair in S and remove it from S.

Repeating this operation in any order until it cannot be applied to S any more leaves S in the form $)^k(^l$, where, $)^k$ is the sequence with k consecutive occurrences of) and $(^l$ is analogous. Here, k or l or both may be 0. In order to balance S, change the first $\lceil k/2 \rceil$ of the) characters to). Now by repeating the pairing operation, we get the sequence $(^{k \mod 2}(^l$. This can be balanced iff $k + l$ is even and in that case, we change $\lceil l/2 \rceil$ rightmost left parenthesis to). Thus converting S of the form $)^k(^l$ into a balanced parenthesis expression is possible iff $k + l$ is even and the number of reversals required is $\lceil k/2 \rceil + \lceil l/2 \rceil$.

If one only needs the count, but not the indices of the reversals, then, keeping a single count would suffice. Let *lcount* be a variable that counts the number of unpaired left parenthesis so far. It is initialized to 0.

1. Scan the next input character. If it is `'('` then increment *lcount* by 1.

2. If it is `')'` and *lcount* > 0, then decrement *lcount* by 1.

3. If it is `')'` and *lcount* $==$ 0, then, (a) increment *reversal_count* by 1, and, (b) increment *lcount* by 1.

When the input is completely read, then, a solution exists iff *lcount* is even and in this case return *reversal_count* + *lcount*/2.

Examples

$S_1 =$ `)(`	Output = 2
$S_2 =$ `)()(`	Output = 2
$S_3 =$ `)(((`	Output = 3

Explanation:

1. Write $S_1 =)_1(_2$ where the subscripts are used to label the parenthesis characters. No balanced parenthesis can start with) so this must be reversed to $(_1$. This gives $(_1(_2$ which requires a reversal of $(_2$ to $)_2$ to obtain $(_1)_2$ which is balanced.

2. Let $S_2 =)_1(_2)_3(_4$. The sub-expression $(_2)_3$ is balanced and can be removed giving $)_1(_4$ which is the same as S_1 and requires two reversals.

3. Write $S_3 =)_1(_2(_3(_4$. First reverse $)_1$ to $(_1$ as above. This gives the sequence $(_1(_2(_3(_4$. Then, $(_3$ and $(_4$ each can be reversed to give the sequence $(_1(_2)_3)_4$ which is balanced. (Also $(_2$ and $(_4$ can be reversed to give the sequence $(_1)_2(_3)_4$. Either way requires 3 reversals).

(*Notes*: The subscript numbering is shown only for convenience. It may not be needed in the algorithm.)

Problem 4. Range queries over Red-Black Trees. A *range-query* $\text{RQ}(T, a, b)$ takes a red-black tree T (with root node T) and keys a and b such that $a \leq b$, and returns *all* the nodes x of T such that $a \leq x.key \leq b$. The output order of the nodes is according to ordering in the inorder traversal of T. Assume T has n nodes in total.

1. Write an algorithm for $\text{CLOSEST\_GE}(T, a)$ stands for that runs in time $O(\log n)$ and returns the leftmost node x in the inorder traversal of T such that $a \leq x.key$. (*Algorithm + Correctness*: 12 *Complexity*: 3)

   Solution. A pseudo-code for a recursive algorithm is shown below. The top-level call is $\text{CLOSEST\_GE\_REC}(T, a)$.

```
CLOSEST_GE_REC(x, a)
1.  if  x == NIL
2.      return NIL
3.  if  a ≤ x.key
4.      w = CLOSEST_GE_REC(x.left, a)
5.      if  w ≠ NIL
6.          return w
7.      else return x
8.  else
9.      w = CLOSEST_GE_REC(x.right, a)
10.     return w
```

   Let us analyze the recursive code first. If x is at a height of h, then, after the test in line 3, the method makes a recursive call to either $x.left$ or to $x.right$. Excepting for the recursive calls, the time taken is $O(1)$. Hence, we get the recurrence relation for an upper bound of $T(h)$ to be $T(h) = T(h-1) + O(1)$, giving $T(h) = O(h)$. For the root of the tree, $h = O(\log n)$ (for red-black trees) and hence, $T(n) = O(\log n)$. The correctness argument is shown in Figure 4.

   Figure 4: Illustrating the proof of correctness of the recursive procedure CLOSEST_GE_REC

   There are two cases, either, (1) $a \geq x.key$, or, (2) $a < x.key$. Consider Case 1. No node in the right subtree of x qualifies to be the $\text{CLOSEST\_GE}(x, a)$, since they all follow x in the inordering. Now let w be the node returned by the recursive call $\text{CLOSEST\_GE\_REC}(x.left, a)$. If w is not NIL, then, w must be the answer, since, w precedes x in the inordering and is $\geq a$. However, if w is NIL, then there is all nodes in the subtree rooted at $x.left$ have keys less than a. Hence the answer must be x.

   Case 2 is argued similarly. if $a < x.key$, then, all nodes in the left subtree of x including x are less than a and therefore cannot qualify to be $\text{CLOSEST\_GE}(x, a)$. Let w be the result of the recursive call $\text{CLOSEST\_GE\_REC}(x.right, a)$. If w is not NIL then we return w, since the rest of the nodes in the subtree rooted at x do not qualify. If w is NIL, then we have to return NIL.

   An iterative code is shown below.

```
CLOSEST_GE(T, a)
1.  x = T
2.  w = NIL
3.  while  x ≠ NIL
4.      if  x.key ≥ a
5.          w = x
6.          x = x.left
7.      else
8.          x = x.right
9.  return  w
```

The iterative algorithm takes $O(h) = O(\log n)$ time (for red-black trees) since it follows a simple downward path from the root to some NIL node. As in the recursive algorithm, if $x.key \geq a$, then, x is a possible closest_ge node for a, and so none of the nodes in the right subtree of x qualify as they occur after x in the inordering. Thus, the algorithm sets x to its left child node. If otherwise, $x.key < a$, then, none of the nodes in the left subtree of x, including x qualify as they all have smaller keys than a. Hence the algorithm moves to the right child of x.

The algorithm keeps the current value of x in a variable w while moving left. This is currently the left most node in the inordering of T that is at least a.

Consider the following algorithm for RANGE_QUERY(T, a, b) that uses CLOSEST_GE(T, a) as a subroutine and the SUCCESSOR(T, x) function on binary search trees. Show that this simple algorithm runs in time $O(m + \log n)$, where, m is the number of nodes output. (10)

```
RANGE_QUERY(T, a, b)
1.  x =CLOSEST_GE(T, a)
2.  while  x ≠ NIL and x.key ≤ b
3.      print x
4.      x = SUCCESSOR(x)
```

Solution. An edge between a node u and $u.left$ is classified as a *left edge* and the edge between a node u and $u.right$ is classified as a *right edge*. Let $x = \text{CLOSEST\_GE}(T, a)$. Consider the sequence of edges S_1 traversed in a simple downward path from the root to x. Let S_2 be the sequence of edges traversed in lines 2-4 of the procedure RANGE_QUERY. Along with each edge in S_1 and S_2, we also note the direction of its traversal, whether, downwards or upwards. Let $S = S_1\ S_2$. Since, the size of S_1 is $O(\log n)$ edges, it suffices to bound the size of S_2.

In the traversal sequence S_2, an edge $u-v$, where, $u = v.p$ is said to be *paired* if $u-v$ is traversed in the downward direction sometime and *and* in the upwards direction at a later time. (By inordering, the converse is not possible). The pair of sibling edges $\{(u, u.left), (u, u.right)\}$ is said to be *teamed* if both are traversed (in some direction). Since a sequence of SUCC calls gives the same sequence as inordering, a team occurs in S_2 iff $u - u.left$ is paired and either $u - u.right$ is only traversed in the downwards direction or it is also paired. The node u is printed iff the edge $u - u.left$ occurs paired in $S = S_1S_2$, u is printed.

Remove from S_1S_2 all occurrence of (a) teamed sibling edges, and (b) paired edges that are not teamed. A team may have between 3 and 4 edge occurrences, and must print a vertex. A paired edge, which is not part of a team has 2 edge occurrences and prints a node. Since there are m nodes printed, the number of edges so removed lies between $2m$ and $4m$. After this removal, let S' be the sequence of edges remaining. These edges are all unpaired in S. What types are these edges? (1) It could be a left edge traversed downwards, or (2) a right edge traversed downwards. The total number of such edges cannot exceed the height of the tree, which is $O(\log n)$ for red-black trees.

Thus the total size of S is $O(m + \log n)$.

For reference, the SUCCESSOR function is given below.

```
SUCCESSOR(x)
1.  if  x.right ≠ NIL
2.      return  MINIMUM(x.right)
3.  else
4.      while  x.p ≠ NIL and x == x.p.right
5.          x = x.p
6.      return  x.p
```