Exam Time: 75 minutes. Each problem is worth 10 points. 60 pts total. Choose any 6 problems. Specify which 6 problems you want graded.

1. **Sorting.** When Randomized-Quicksort runs, how many calls are made to the random-number generator Random

   1. in the worst case?
   2. In the best case?

   Explain (just giving an answer will not do) and use Θ-notation. Recall:

   ```
   RANDOMIZED-PARTITION(A, p, r)
   1   i = RANDOM(p, r)
   2   exchange A[r] with A[i]
   3   return PARTITION(A, p, r)
   ```

   ```
   RANDOMIZED-QUICKSORT(A, p, r)
   1   if p < r
   2       q = RANDOMIZED-PARTITION(A, p, r)
   3       RANDOMIZED-QUICKSORT(A, p, q − 1)
   4       RANDOMIZED-QUICKSORT(A, q + 1, r)
   ```

   **Solution:** 1. Every call to Random removes one element from any subsequent call. The call to Partition calls Random only if the array to be partitioned has length 2 or greater. The maximum number of calls is the n − 1. 2. The configuration that skips the most calls to Random is the following: in an array A[1..n], A[2], A[5], A[8], . . . , A[3k + 2], . . . are chosen as pivots (the final configuration is after the partitionings have been carried out), starting from the midpoint of the array, then the midpoints of the left and right subarrays, etc., so a best case has no more than ⌈n/3⌉ calls to Random. Both cases are thus Θ(n).
2. Sorting.
   1. Describe bucket sort.
   2. Explain why the worst-case time for bucket sort is $O(n^2)$.
   3. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

   **Solution:** 1. pp. 200-201 of the text. 2. If every item ends in the same bucket, the worst case of InsertionSort is $\Omega(n^2)$. 3. Replace InsertionSort with an $O(n \lg n)$ sort.


   **Solution:** 1. p. 216. 2. $\Theta(n^2)$. See bottom of p. 216. 3. $\Theta(n)$: you have to run Randomized-Partition at least once. In the best case, the item chosen as the pivot during the first call will end up in position $i$.

4. Hashing. Prof. Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor’s modification affect the running time for successful searches, unsuccessful searches, insertions and deletions?

   **Solution:** The order of insertion, used for the regular case, no longer matters. The expected position of something found is in the middle of the chain attached to the hash position. Since any item has equal probability as any other, you can use the fact that all chains are sorted to conclude that an unsuccessful search, with the use of order, will terminate, on average, in the middle of the chain. Insertion will no longer be $O(1)$, since it involves inserting the new item in the correct position (which will be, on average, in the middle of the chain). Deletion: if you have a pointer to the item to be deleted (and you have a doubly linked list) the cost is $O(1)$. If you have to search, then it must include the cost of the search and the cost of the actual deletion. Restate everything in terms of the load factor $\alpha$, which will give you the expected length of each chain.

5. Stacks and Queues. Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

   **Solution:** Let $S_1$ and $S_2$ be stacks, with the operations Push, Pop and Stack-empty. We can implement the operation Enqueue and Dequeue as follows:

   - Enqueue($x$):
     ```
     if not Stack-Empty(S2) Move(S2, S1))
     Push(S1, x)
     ```
   - Dequeue
     ```
     if not Stack-Empty(S1) Move(S1, S2)
     if not Stack-Empty(S2) Pop(S2)
     ```
else nil
Move(A, B)
while not Stack-Empty(A)
    Push(B, Pop(A))

For the running time, the best-case cost for an Enqueue or a Dequeue is $\Theta(1)$: the correct stack is empty and so the operation reduces to a Push or a Pop. The worst case requires moving the contents of one stack into the other before the Push or Pop: $\Theta(n)$, for a queue of $n$ elements.

6. Red-Black Trees. You are given the rotations for half the cases of the insertion algorithm for Red-Black Trees. The other half are symmetric. Show the Red-Black trees that result after successively inserting 3, 7, 13, 9, 12, 8. In each case (i.e., after each individual insertion), if you are using a rotation, tell me which one you are using (use Case i symmetric to denote a rotation symmetric to one of the pictures above).

**Case 1: y is red**

**Case 2: y is black, z is a right child**

**Case 2**

**Case 3**

7. Red-Black Trees. Prof. Teach is concerned that RB-Insert-Fixup might set $T.nil.color$ to Red, in which case the test in line 1 would not cause the loop to terminate when $z$ is the root. Show that the professor’s concern is unfounded by arguing that RB-Insert-Fixup
never sets $T.nil.color$ to Red.

Recall: what is the initial color of $z$?

**Solution:** First of all, $z$ is RED. At the beginning of RB-Insert-Fixup, the only violation of the Red-Black conditions could be that we have two RED nodes in a parent-child relation. If $z.p.color = BLACK$, we are done (the loop is skipped) and nobody is set to RED. If $z.p.color = RED$, then we know $z.p.p = BLACK$ and $z.p.p$ cannot be $T.nil$, since the root ($root.color == BLACK$) must be an ancestor of $z.p$, and the nearest ancestor of $z.p$ is $z.p.p$. The first if simply checks whether $z$’s parent is a left or right child of $z$’s grandparent ($z.p.p$), and sets to color. The first branch of the inner if sets the color of the uncle and parent of $z$ to BLACK, moving the RED up to the grandparent $z.p.p$. Since none of these nodes was $T.nil$, we are safe. The second branch of the inner if involves some rotations, and you must make sure that, at the end of the rotations, $z.p.p \neq T.nil$ so it can be safely set to RED.