Spring 2013 Midterm Solutions (Beta Edition)

1. **Analysis of Algorithms.**
   (a) N^2. Reading in the genomes is linear time. The for loops are just insertion sort, which is N^2 in the worst case (where the break statement never occurs).
   (b) 0.05 N (ok if left in terms of a fraction)
   (c) Multiple acceptable answers
      a. The input may not be a worst case input (e.g. already sorted)
      b. The time to read in a 64 megabytes of genomes is much larger than the time to perform 64^2 string length compares and 64^2 swaps of 8 byte references.
      c. Partial credit: N is too small / time is too short

2. **Red-Black BSTs.**

   (a) A B C E F H K L N P T V X Y Z

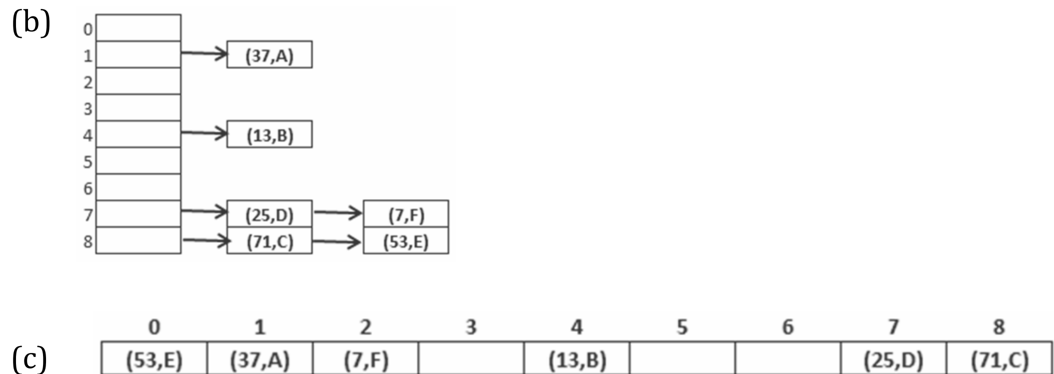
   (b) S M W G Q U Z D J O R I
   ```
     ^                     ^
   ```

   (c) Returns true if the Node is the root of a properly constructed BST. Best approach is probably just to draw some arbitrary BST with integer keys and trace the algorithm, relying on human inductive skill to arrive at the answer.

3. **Symbol Tables.**

   (a) D (trivial to calculate index using angle, half credit for hash table)
   C (no natural order exists, better than unordered array)
   B (for any data set, basic operations guaranteed to be logarithmic)

   (b)



   (c)

0	1	2	3	4	5	6	7	8
(53,E)	(37,A)	(7,F)		(13,B)			(25,D)	(71,C)

   (d) N^2. The entire linked list must still be searched to avoid duplicates.

4. **Quicksort.**
   (a) I M P E A E F R I S S S T W

(b) Must partition only 7 times total. Each partition takes linear time.

5. **Heaps and Priority Queues.**
   a. – V J R E J G F F B F E A E B D
   b. Heapification is linear time. Best algorithm is to use 'bottom up heapification', where each node is successively sunk starting with the rightmost element of the array and working backwards to the left.
   c. Many possible answers:
      i. Yes, bottom up heapification is linear time, whereas constructing an LLRB takes linearithmic time. Also, finding the max with a heap is constant time, but in an LLRB it is logarithmic time (though this can be trivially fixed by adding an extra instance variable to an LLRB based PQ).
      ii. Yes, a max heap uses less memory since we don't need to store lots of node objects (which include node links and colors).
      iii. Partial credit: Yes, because the number of compares / height of an LLRB is larger for deletion. However, this is not an apples-to-apples comparison, as the deletion process for both is fundamentally different (max heap involves much more data movement).

6. **Sorting**.
   a. 5 7 6 4 3 9 2 8

7. **You can't do that on television.**
   I. Directly violates our sorting bound N lg N.
   P. By performing quicksort but only recursing on the side with that contains index n/2, you can find the median on average in linear time (this is also known as quick select).
   P. This is the Tarjan algorithm that was described in the Quick Select lecture. This algorithm completes in linear time (though it is impractical).
   I. LLRB is guaranteed to be of height 2 lg N or less.
   P. One way is to simply sort the array, then repeatedly insert the median of the remaining data points.
   P. One can simply use an ordered array (or equivalently, use a trivial hash function which simply returns the key as its own hash). Since there are only 65,536 different 16 bit integers, this array based approach is practical.
   I. Each LLRB insertion requires worst case time equal to the height of the tree. Since the tree cannot have height more than 2 lg N, there is no way to reach N^2 time.
   P. This is bottom up heapification (sink each item starting with the rightmost one and work your way to the left). We showed in class that it is linear time (see Heaps and Priority Queues lecture).
   I. If only using swim operations, the worst case is that every item will swim all the way to the very top (e.g. the items are swum in ascending order). Without additional operations (e.g. conditionals) there is no way to avoid this possibility. We can lower bound the overall runtime by considering the last N/2 items, which will take lg N time in the worst case.

I. The height of this weighted quick union tree is too large, i.e. 4 > lg(10).

8. **Addendum.**
   a. Since M is much less than N, the absolute worst case is that each of the M items is moved all the way to the front of the original array. In this case, the run time is MN.
   b. Using an array is inherently a bad idea if our goal is to achieve time optimality, since no matter what we try, we're still going to have to move all of our old N items. Instead, we want to use some sort of data structure which maintains order AND does not involve movement of our old data.

      This leaves us with either a linked list or a tree of some kind. A linked list avoids the perils of moving all N of our items, but unfortunately traversing the tree to find the right position can take up to linear time (leaving us right back at MN). We could try to do something like a skip list, but we didn't talk about those in class.

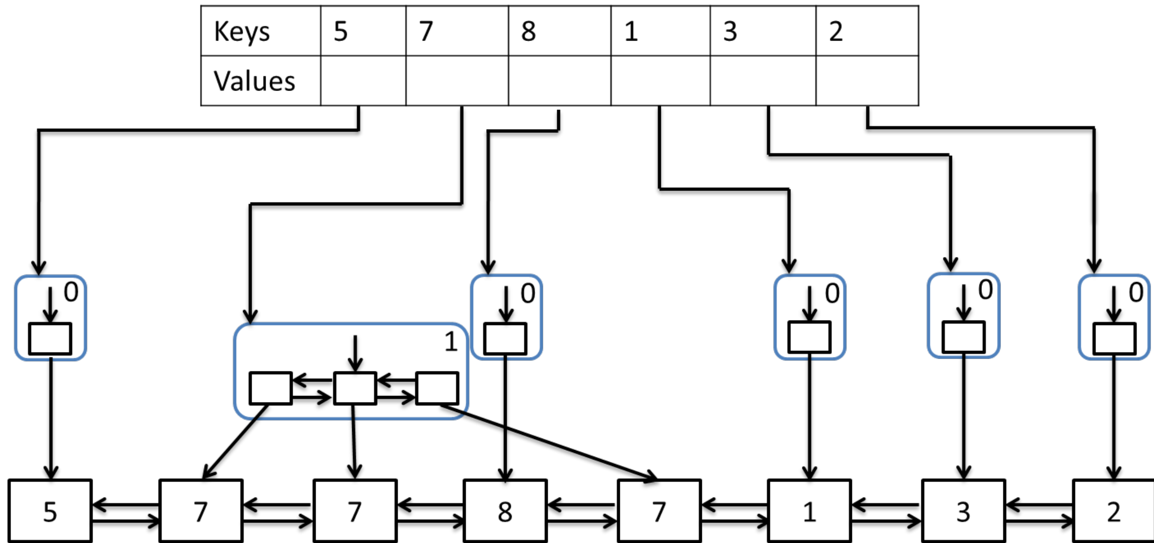
      This leaves us with a tree. If we use a standard BST, we could be in terrible shape, with a tree of up to height N, leaving us right back in MN time. If we use an LLRB, our max tree height is 2 lg N, meaning that insertion of our M new items will have a runtime of order M lg N.

      The amount of memory used is only M + N, which is optimal to within a constant factor (since we have to store all the data).

9. **Excavating Deque.**
   (a) We have a few fundamental needs that our data structures must meet.
      i. We must keep track of the order of the inserted items (e.g. if there are no duplicates, the ExcavatingDeque should behave exactly like a Deque). This means that a **Deque of integers** (or equivalently a doubly linked list) is a great starting point. For clarity, let's call this our **main Deque**.
      ii. For each duplicate, we must also be able to identify the most deeply nested copy. There are many solutions, but the easiest to draw is a **Deque with a middle pointer**. To ensure that we can keep the middle pointer in the right spot, we also store the current position of the middle pointer as an integer. For clarity, let's call this second Deque a **mostNestedTracker**. The nodes in this Deque store references to nodes in the main Deque (to allow deletion).
      iii. Finally, given a front or back key, we need to be able to find the appropriate mostNestedTracker, so we have a **hash table that maps integers to mostNestedTrackers**.

   An overly verbose diagram is shown below. In your answer, you could have drawn only the mostNestedTracker for 7 (since it's the only non-

trivial one). For answers with the correct answers but no diagram, we awarded full points.



(b) addFirst(int x): First, we simply add the key to the front of the main Deque (the bottom one in the diagram). We then look up this key in the hash table. If it is present, then we add the key to the front of the mostNestedTracker for that item; we also move the arrow towards the front (and decrement the position integer) if the position integer is greater than n/2. If it is not present, then we create a new mostNestedTracker. Finally we point the first node of the firstMiddleTracker to the front item in the main Deque.

(c) deleteFirst(): We get the front item from the main Deque and look up the appropriate firstMiddleTracker. We get the middle item using the middle pointer, and delete it from the Deque and the firstMiddleTracker. Finally, we move the middle pointer to the right (and increment the position integer) if the position integer is less than n/2.