COS 226	Algorithms and Data Structures	Fall 2023

Midterm Solutions

1. **Initialization.**

   Don't forget to do this.

2. **Resizable arrays.**

   (a) $\sim 12n$

   An integer array of length m *uses* $\sim 4m$ *bytes. In the worst case, just after a* `push()` *operation that triples the length of the underlying array, with* $m = 3(n-1)$.

   (b) F T T

3. **Data structures.**

   (a) 0–3, 0–4, 0–5, 0–6

   The link 3–7 must have been created during the last call to `union()`.

   - *The link 3–7 can't occur until the tree containing 3, 4, 5, and 6 is fully formed because, after 3 points to 7, new links always point to tree roots.*
   - *The link 3–7 can't occur until the tree containing 0, 1, 2, and 7 has size 4 because the tree rooted at 3 will be larger than the other tree until this occurs.*

   (b) 30 40 50

   The last key inserted must be on the path from the last node in the binary heap (30) to the root node. However, 60 is not a possibility because that would mean that 30, 40, and 50 were all one level higher prior to the insertion, including 50 as the root with 55 as its left child.

4. **Five sorting algorithms.**

   D C B F E

   D. *mergesort just before the left half of the array is sorted*

   C. *insertion sort after 16 iterations*

   B. *selection sort after 12 iterations*

   F. *heapsort after heap construction phase and putting 6 keys into place*

   E. *quicksort after first partitioning step*

5. **Analysis of algorithms and sorting.**

   (a) $\sim 8n^2$

   Selection sort always makes $\sim \frac{1}{2}m^2$ *compares, where* m *is the length of the array. Here,* $m = 4n$.

   (b) $\sim \frac{7}{2} n \log_2 n$

   In general, mergesort will be merging two arrays of the AABBCCDD and AABBCCDD, i.e., k *As, followed by* k *Bs, followed by* k *Cs, followed by* k *Ds. If the merged array is of length* m*, then this will take* $\frac{7}{8}m$ *compares because, when the left array is exhausted, there will be* k *(of the original* $8k$*) elements remaining in the right subarray. Thus, the number of compares satisfies the divide-and-conquer recurrence* $T(m) = 2T(m/2) + \frac{7}{8}m$*. So,* $T(m) = \frac{7}{8} m \log_2 m$*. The length of the array* $m = 4n$.

   (c) $\Theta(n)$

   There are only 4 distinct keys. The number of 3-way partitioning steps is at most the number of distinct keys. Each partitioning step takes $\Theta(n)$ *time.*

6. **Advanced Java.**

   G B G G

7. **Properties of BSTs.**

   T F T T

8. **Rank in a BST.**

   J A E D F C

9. **Algorithm design.**

   (a) Use `BinarySearchDeluxe.firstIndexOf()` and `BinarySearchDeluxe.lastIndexOf()` to determine how many times x appears in the array. Then compare that value with $n/4$.

   (b) The key idea is that if an integer appears more than $n/4$ times in a sorted array `a[]`, it must appear at either `a[n/4]`, `a[n/2]`, or `a[3*n/4]`. So, it suffices to run (a) on each of these three candidate keys to identify all keys that appear strictly more than $n/4$ times.

10. **Data structure design.**

The key idea is to insert the integers into a *balanced search tree* and use the *floor* and *ceiling* methods to process nearest neighbor queries (because the nearest neighbor of x must be either the *floor* or *ceiling* of x).

For reference, here is a full Java implementation. By using a red–black BST, we ensure that `insert()` and `nearest()` take $O(\log n)$ time in the worst case. The value component in the symbol table is irrelevant, so we use -1.

```
public class NearestNeighbor {
    private RedBlackBST<Integer, Integer> bst = new RedBlackBST<>();

    private void insert(int x) {
        bst.put(x, -1);
    }

    private int nearest(int x) {
        if (bst.size() == 0) return -1;
        if (x <= bst.min()) return bst.min();   // x is smaller than smallest key
        if (x >= bst.max()) return bst.max();   // x is largest than largest key
        int floor = bst.floor(x);
        int ceiling = bst.ceiling(x);
        if (Math.abs(x - floor) < Math.abs(x - ceiling)) return floor;
        else                                             return ceiling;
    }
}
```