COS 226	Algorithms and Data Structures	Spring 2025

Midterm Solutions

1. **Initialization.**

   You will lose points if you neglect to write your name; select the wrong precept; go to the wrong room; or fail to write and sign the honor code.

2. **Memory.**

   ~ $72n$ bytes

   There are n `Node` objects. Each `Node` object is 72 bytes:

   - object overhead (16 bytes)
   - inner class object overhead (8 bytes)
   - 5 object references (5×8 bytes)
   - 1 int (4 bytes)
   - padding (4 bytes)

3. **Data structures.**

   (a) 0 4 8

   (b) 50–30 50–55 50–75 20–30 10–55 70–75

   (c) 20 22, red

4. **Five sorting algorithms.**

   F D B E C

   F. *heapsort after heap construction phase and putting 12 keys into place*

   D. *mergesort just before the last call to* `merge()`

   B. *selection sort after 12 iterations*

   E. *quicksort after first partitioning step*

   C. *insertion sort after 16 iterations*

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

   (a) $\sim 2n^2$

   (b) $\sim \frac{1}{2} n^2$

   (c) $\sim \frac{3}{2} n \log_2 n$

6. **Algorithms.**

   - T *Similar to merging two sorted subarrays in mergesort. But, in a linked list, you don't need $\Theta(n)$ extra space (because you can just relink the nodes).*
   - F *The worst-case memory usage is ~ $32n$; it occurs immediately after the length of the array is quadrupled (not after it is halved).*
   - T *The* `enqueue()` *and* `dequeue()` *methods in* `RandomizedQueue` *take $O(1)$ amortized time. So, starting from an empty data structure, any sequence of $2n$* `enqueue()` *and* `dequeue()` *operations takes $O(n)$ time in the worst case.*
   - T *You can do it with zero compares. An inorder traversal of a BST yields the keys in ascending order. An array in ascending (descending) order is a min-oriented (max-oriented) binary heap.*
   - T *The only time you perform a right rotation during an insertion is when you have two left-leaning red links in a row. The reason for performing the right rotation is to setup a color flip.*

7. **Linked structures.**

   E H C J B

8. **Algorithm design.**

   Full credit solution: The key idea is to do a version of *binary search*, maintaining the invariant that `a[lo]` is orange and `a[hi]` is black.

   - Initialize $lo = 0$ and $hi = n - 1$.
   - Repeat until $hi = lo + 1$:
     - set $mid = \frac{lo+hi}{2}$
     - if `a[mid]` is orange, update $lo = mid$
     - otherwise, `a[mid]` is black, so update $hi = mid$
   - Return lo

   Partial credit solution: Treat orange as less than black in a sorted ordering. The partial credit assumption means that `a[]` is sorted with respect to this ordering. Call `lastIndexOf()` from Assignment 3 to find the index i of the rightmost orange entry in the array.

9. **Data structure design.**

The key idea is to modify the *weighted quick union* data structure to replace the `parent[]` array with explicit nodes and parent links. We also maintain a symbol table `nodes` to map from elements to nodes, so that `nodes.get(p)` is the node corresponding to element p.

To avoid adding all n elements to the symbol table (which would take $\Omega(n)$ time), we add an element only *when* it first becomes an argument to either `find()` or `union()`.

For reference, we include not only the instance variables, but the full Java code:

```
public class UF {
    // mapping from elements to nodes
    private RedBlackBST<Integer, Node> nodes = new RedBlackBST<>();

    private static class Node {
        private int element;   // the element
        private Node parent;   // the parent of this node in tree; null if this node is a root
        private int size;      // number of elements in subtree rooted at this node

        public Node(int p) {
            element = p;
            parent = null;
            size = 1;
        }
    }

    // returns leader of set containing p (the element in root of tree containing p)
    public int find(int p) {
        Node x = rootOf(p);
        return x.element;
    }

    // returns root node of tree containing p
    // (adds p to symbol table if not already present)
    private Node rootOf(int p) {
        if (!nodes.contains(p)) nodes.put(p, new Node(p));   // add p to symbol table
        Node x = nodes.get(p);
        while (x.parent != null) {
            x = x.parent;
        }
        return x;
    }

    // merge sets containing p and q (by linking root of smaller tree to root of larger tree)
    public void union(int p, int q) {
        Node root1 = rootOf(p);
        Node root2 = rootOf(q);
        if (root1 == root2) return;    // elements are already in the same set

        if (root1.size < root2.size) {
            root1.parent = root2;
            root2.size = root1.size + root2.size;
        }
        else {
            root2.parent = root1;
            root1.size = root1.size + root2.size;
        }
    }
```

If we use a red–black BST for the symbol table, the `union()` and `find()` operations each take $O(\log n)$ time, with the bottleneck being `rootOf()`:

- $O(\log n)$ for the symbol-table operations in a symbol table with $\leq n$ keys.
- $O(\log n)$ to follow parent pointers up the tree, which has height $\leq \log_2 n$.

Almost full credit solution. Instead of implementing the weighted quick union data type using explicit nodes and links, we could implement it using two symbol tables:

- One symbol table with key = element and value = parent of element in quick union tree.
- A second symbol table with key = element and value = size of subtree rooted at element.

This almost achieves the performance requirements, except that `union()` and `find()` each take $\Theta(\log^2 n)$ time in the worst case instead of $\Theta(\log n)$ time. The reason for this is that

- We might have to follow $\log_2 n$ parent pointers in the weighted quick union tree.
- Following a parent pointer might take $\Theta(\log n)$ time if we use a red–black tree for the symbol table.