COS 226 **Algorithms and Data Structures** **Fall 2019**

Midterm Solutions

0. **Initialization.** Don't forget to do this.

1. **Memory.**

   (a) 104 bytes

   - 16 bytes of object overhead
   - 8 bytes of inner class overhead
   - 8 bytes for parent link
   - 8 bytes for reference to array of children
   - $56 = 24 + 8d$ bytes for `Node[]` array of length d
   - 4 bytes for integer instance variable d
   - 4 bytes of padding

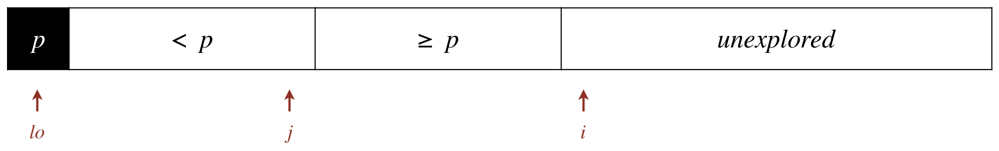
   (b) $\sim 104n$ bytes

2. **Five sorting algorithms.**

0	*original array*
1	*selection sort after 12 iterations*
4	*quicksort after first partitioning step*
3	*mergesort just before the last call to* `merge()`
2	*insertion sort after 16 iterations*
5	*heapsort after heap construction phase and putting 12 keys into place*
6	*sorted array*

3. **Quicksort and analysis.**

   This diagram shows the invariants maintained during the partitioning algorithm:



   (a) $\sim n$

   Calling `partition()` *on any subarray of length n involves exactly $n-1$ compares.*

   (b) $\sim n$

   The `i` *loop triggers one exchange for each element strictly less than the partitioning element. There is also one extra exchange after the loop terminates (to put the partitioning element in place). So, in the worst-case, there are exactly n exchanges (and this occurs when the partitioning element is the strictly largest element).*

   (c) in-place, not stable

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

   When all keys are equal, every partition is degenerate. In this case, `partition()` *is called on subarrays of length $n, n-1, n-2, \ldots, 1$. So, the total number of compares is $1 + 2 + 3 + \ldots + (n-1)$.*

   This scheme—known as *Lomuto partitioning*— is simpler than Hoare partitioning. However, it should not be used in practice because (1) it performs more exchanges on inputs with all distinct keys and (2) it takes quadratic time on inputs with a large number of duplicate keys.

4. **Red–black BSTs.**

   24 rotate right → 22 color flip → 20 color flip → 12 rotate left

5. **Collections.**

   E F A H C G B D

6. **Why did we do that?**

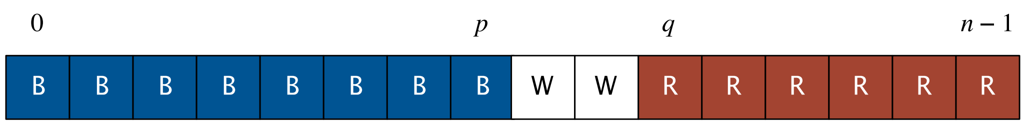
   B C B C D A A E

7. **Data structures.**

   A C C E B

8. **Red, white, and blue.**

The main idea is to use *binary search* to find the indices p and q of the color boundaries.



The easiest way to accomplish this is to reuse the code from Assignment 3 (that finds either the first or last occurrence of a given key in a *sorted* array). Specifically, we compute the index p of the last occurrence of the color of pebble `a[0]` and the index q of the first occurrence of the color of pebble `a[n-1]`. From the indices p and q, it's easy to compute the frequencies n_1, n_2, and n_3 of the three colors.

```
int p = BinarySearchDeluxe.lastIndexOf(a[0],    comparator);
int q = BinarySearchDeluxe.firstIndexOf(a[n-1], comparator);
int n1 = p + 1;         // frequency of color of pebble a[0]
int n3 = n - q;         // frequency of color of pebble a[n-1]
int n2 = q - p - 1;     // frequency or remaining color
```

Of course, `BinarySearchDeluxe` works only if the underlying array is *sorted*. This requires us to define a *comparator* that specifies a total order for pebbles based on color. We implement this comparator so that `a[0]` is treated as the "smallest" color in the total order; `a[n-1]` as the "largest" color; and a pebble of the remaining color as the "middle" color.

You weren't expected to write code for the comparator, but it would look something like this:

```
public class PebbleComparator implements Comparator<Pebble> {
    private final Pebble min;   // a pebble containing "smallest" color
    private final Pebble max;   // a pebble containing "largest" color

    public PebbleComparator(Pebble min, Pebble max) {
        this.min = min;
        this.max = max;
    }

    public int compare(Pebble x, Pebble y) {
        if (x.color() == y.color())   return  0;
        if (x.color() == min.color()) return -1;
        if (y.color() == min.color()) return +1;
        if (x.color() == max.color()) return +1;
        if (y.color() == max.color()) return -1;
        throw new IllegalStateException("more than 3 colors");
    }
}
```

9. **Data-type design.** (*red–black BST solution*)

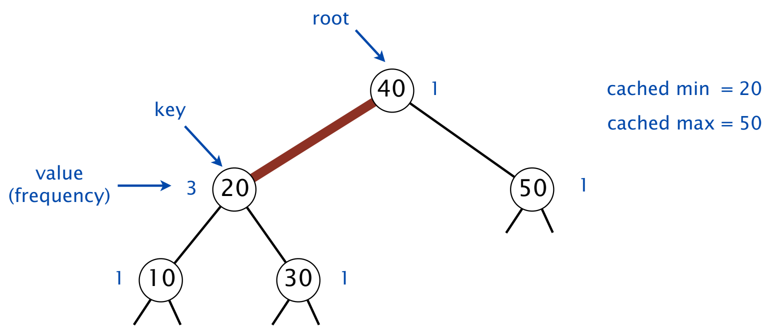
We maintain the keys in a *red–black BST*. Assuming no duplicate keys, we can implement *insert*, *delete-min*, and *delete-max* in logarithmic time using standard red–black BST operations. The main challenge with this approach is dealing with equal keys (because our symbol-table implementations do not allow duplicate keys).

(a) We use our `RedBlackBST` (or `ST`) implementation of a symbol table. To keep track of duplicate keys, we define the `Value` type so that it stores the the number of times the corresponding key appears in the priority queue.

In addition, we *cache* the minimum and maximum keys. This enables us to implement the `min()` and `max()` operations in constant time (instead of $\log n$ time).

```
public class MinMaxPQ<Key extends Comparable<Key>> {
    private ST<Key, Integer> st;  // keys and their frequencies
    private Key min;              // cache of minimum key
    private Key max;              // cache of maximum key
}
```

(b)



(c) Return the cached `min` or `max` instance variable.

(d)
- If the key is not in the symbol table, add it to the symbol table with a frequency of 1 (and update the cached min/max).
- Otherwise, increment the frequency of the key.

(e)
- If the min key in the symbol table has frequency > 1, decrement its frequency by 1 and return the cached min.
- If the min key in the symbol table has frequency = 1, delete the min key from the symbol table and return it. Just before returning it, update the cached min/max.

An alternative BST-based approach is to modify our red–black BST to directly support duplicate keys. Specifically, when inserting key x, if the key in the BST node equals x, simply go right (or left) and continue inserting (instead of replacing the old value with the new value).

9. **Data-type design.** (*two priority queue solution*)

We maintain each key both in a *min-oriented heap* and in a *max-oriented heap*. The main challenge is keeping the elements in the two heaps in sync (e.g., if a key is deleted from one heap via a *delete-min*, then we must be careful to delete it from the other heap).

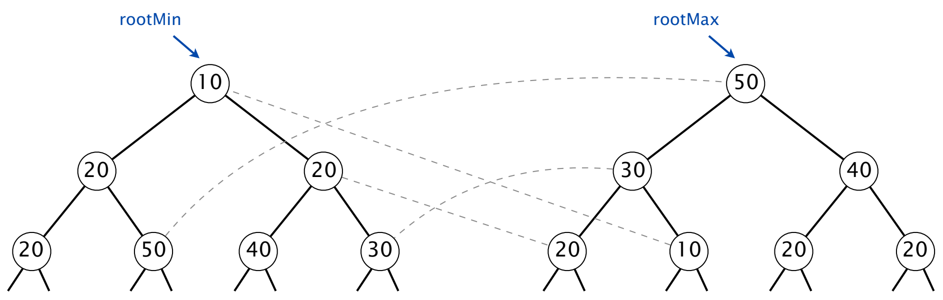
(a) We make corresponding nodes (that hold the same key) point to each other. This enables us to delete the corresponding key from one heap when a *delete-min* (or *delete-max*) operation is performed in the other heap.

We also represent the binary tree using explicit nodes and links instead of a resizing array. This is needed to achieve worst-case $\log n$ performance (instead of amortized).

```
public class MinMaxPQ<Key extends Comparable<Key>> {
    private int n;          // number of keys
    private Node rootMin;   // root of min-oriented heap
    private Node rootMax;   // root of max-oriented heap

    private class Node {
        private final Key key;       // the key
        private Node parent;         // parent link
        private Node left, right;    // children links
        private Node twin;           // corresponding node in the other heap
    }
}
```

(b) For clarity, this diagram shows only four of the seven pairs of twin links.



(c) Return the key in the root node of the min-oriented heap.

(d) Create two twin `Node` objects (containing given key) and insert one into min-heap (e.g., add to next available spot in the min-heap and perform a *swim* operation) and the twin into the max-heap.

(e)
- Perform a *delete-min* operation in the min-heap (i.e., exchange the root node with the last node in the heap and perform a *sink* operation at the root).
- Use the `twin` link to find the twin of deleted node in the max-heap.
- Delete the twin node from the max-heap (i.e., exchange the twin node with the last node in the heap and perform a *swim* operation).
- Return the key in (either) deleted node.