COS 226 Algorithms and Data Structures Spring 2014

Midterm Solutions

1. **Memory and data structures.**

   $\sim 56N$ bytes.

   Each `Node` object uses 56 bytes: 16 (object overhead) + 8 (inner class) + 8 (reference to `Key`) + 24 (references to three `Node` objects).

2. **Eight sorting algorithms and a shuffling algorithm.**

   0 4 5 7 2 3 9 6 8 10 1

   4. Knuth shuffle after 12 iterations.
   5. Shellsort (with 3x + 1 increment sequence) after 4-sorting phase.
   7. Mergesort (bottom-up) after merging sorted subarrays of size 4.
   2. Selection sort after 12 iterations.
   3. Insertion sort after 16 iterations.
   9. Quicksort (3-way, no shuffle) after first partitioning step.
   6. Mergesort (top-down) just before second-to-last call to merge
   8. Quicksort (standard, no shuffle) after first partitioning step
   10. Heapsort after heap construction phase

3. **Analysis of algorithms.**

   (a) $\sim \frac{1}{2}N^2$.
   Selection makes $\sim \frac{1}{2}N^2$ compares on any array of N keys.

   (b) $\sim \frac{1}{8}N^2$.
   The total number of inversions is $1 + 2 + 3 + \ldots + \frac{1}{2}N \sim \frac{1}{8}N^2$ because the ith A is inverted with i Bs. The number of compares in insertion sort is never larger than the number of inversions + N.

   (c) $\sim \frac{3}{4}N \log_2 N$.
   In general, each merge involves two subarrays of the form AAAABBBB, i.e., $\frac{1}{2}N$ As followed by $\frac{1}{2}N$ Bs. This takes $\frac{3}{4}N$ compares because the left subarray is exhausted with $\frac{1}{4}N$ Bs remaining in the right subarray. Thus, the total number of compares satisfies the recurrence $T(N) = 2T(N/2) + \frac{3}{4}N$

4. **Binary heaps.**

   (a) 19 26 32 35

   To insert a node in a binary heap, we place it in the next available leaf node and swim it up. Thus, 19, 26, 32, 35, and 38 are the only keys that might move. But, the last inserted key could not have been 38, because, then, 35 would have been the old root (which would violate heap order because the the left child of the root is 37).

   (b) 14 17 19 34 35 36 37

   The compares are 35-37, 19-37 34-36 19-36 14-17 19-17.

5. **Red-black BSTs.**

   24 21 20 14

6. **Problem identification.**

   P P I I P P P

   P. The weighted quick-union data type achieves logarithmic time per union,find, and connected operation.

   P. In Java, `hashCode()` returns a 32-bit integer, so there are only 2^{32} possible values. Since there are infinitely many different strings, an infinite number of strings share the same hash code. Moreover, sice the formula for computing a string hash code is fixed, it's easy to find pairs of strings that have the same hash code, such as `"Aa"` and `"BB"`.

   I. If you could do this, then the number of compares in mergesort would satisfy the recurrence $C(N) = 2C(N/2) + \frac{1}{2}N$, whose solution is $C(N) \sim \frac{1}{2}N \log_2 N$. This compare-based sorting algorithm would violate the $\sim N \log_2 N$ sorting lower bound.

   I. Recall that we can build a heap on any N keys with at most $2N$ compares. If we could then convert this heap into a BST with an additional $\sim 17N$ compares, then we could sort the keys by doing an inorder traversal of the BST. That is, we could sort using only $\sim 19N$ compares, which would violate the $\sim N \log_2 N$ sorting lower bound.

   P. The reverse of an inorder traversal yields the keys in reverse order, which is heap ordered. That is, we can do it with 0 compares.

   P. Mergesort is such an algorithm.

   P. This can be accomplished by two binary searches: one to find the first key in the array equal to the query key and one to find the last key in the array equal to the query key (as you did on the autocomplete assignment).

7. **Leaky stack.**

   The core idea is to use an ordered symbol table to store the strings on the stack, where the i^{th} string inserted is the *value* associated with the *integer key* i. We can use the *delete-max* operation (or, alternatively, *select* and *delete*) to delete the string that was most recently inserted; we can use the *select* operation to identify a random string and the *delete* operation to delete it.

```
public class LeakyStack {
    private int counter = 0;
    private RedBlackBST<Integer, String> st = new RedBlackBST<Integer, String>();

    public void push(String item) {
        st.put(counter++, item);
    }

    public String pop() {
        String item = st.get(st.max());
        st.deleteMax();
        return item;
    }

    public void leak() {
        int r = StdRandom.uniform(st.size());
        st.delete(st.select(r));
    }
}
```

   By using a red-black BST for the symbol table, all operations take logarithmic time in the worst case.

8. **Largest common item.**

   (a)
   1. Sort each row using heapsort.
   2. For each number in row 0, from largest to smallest, use binary search to check if it appears in the other $N-1$ rows.
   3. Return the first number that appears in all N rows.

   The order of growth of the running time is $N^2 \log N$, with the bottleneck being steps 1 and 2. Correctness follows because the largest common number must appear in row 0. Scanning the numbers in row 0 from largest to smallest ensures that we find the *largest* common number.

   (b) $N^2 \log N$