COS 226 Algorithms and Data Structures Spring 2015

Midterm Solutions

1. **Memory and data structures.**

   40 bytes.
   16 bytes object overhead; 4 bytes for each of two ints; 8 bytes for each of two references.

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

   0 6 5 4 7 8 2 3 1 9

3. **Analysis of algorithms and sorting.**

   This sorting algorithm is commonly known as *bubblesort* after the way the smaller elements "bubble" to the front of the array.

   (a) i and ii only.

   (b) The algorithm is both stable and in place.
   Stability follows because an entry can be exchanged only with its immediate neighbor, and equal keys are never exchanged.

   (c) $\sim \frac{1}{2}N^2$
   As implemented, bubblesort performs the same number of compares on any array of length N.

   (d) Insertion sort makes exactly the same number of exchanges for all arrays.
   To see why, note that every exchange in bubblesort reduces the number of inversions by exactly one. Thus, the number of exchanges in bubble sort is equal to the number of inversions, exactly as in insertion sort.

4. **Red-black BSTs.**

   (a) Color flip 24

   (b) Rotate 18 left; rotate 22 left; rotate 26 right; rotate 28 right.

   The full sequence of elementary operations is:

   - *Rotate 26 right*
   - *Color flip 24*
   - *Rotate 22 left*
   - *Rotate 28 right*
   - *Color flip 24*
   - *Rotate 18 left*

5. **Hash tables.**

   (a) (iii)

   A *collision* occurs when two key-value pairs have different keys but hash to the same index.

   (b) (iii) and (v) only

   (c) It can result in a separate-chaining hash table having some very long chains and a linear-probing hash table having some very large clusters. Either of these situations can lead to poor (e.g., linear-time) performance for insert, search hit, and search miss.

   A linear-probing hash table can never become 100% full because it would be resized before this happened.

6. **Properties of data structures.**

   (a) **Binary heaps.**

      i. False.
      Key $N-2$ can be also be a grandchild of the root node: `a[4]`, `a[5]`, `a[6]`, *or* `a[7]`.

      ii. True.
      Since the keys are inserted in descending order, each insertion (other than the first) takes only 1 compare and 0 exchanges to swim it up from the bottom.

      iii. True.
      The height of a complete ternary tree on N keys is $\sim \log_3 N$. Inserting a key into a 3-way heap involves at most one compare and exchange per node on the path from a leaf to the root.

   (b) **Binary search trees.**

      i. False.
      If you could insert any N comparable keys into a BST using $2N$ compares, then you could sort those N keys using $2N$ compares by performing an inorder traversal of the BST. This would violate the sorting lower bound.

      ii. True.
      This is a basic propery of red-black BSTs.

      iii. True.

7. **Algorithm design.**

Solution 1. The key idea is to use a version of binary search to find the index r of the smallest key and observe that it divides the array into two sorted subarrays. Then, once we know r, we can binary search for the search key x in either the sorted subarray $a[0..r)$ or the sorted subarray $a[r..n)$.

If either $n \leq 1$ or $b[0] < b[n-1]$, then the array is sorted and $r = 0$. Otherwise, finding the index r is similar to problem 6 on the Fall 2014 midterm because r is the unique index for which $b[r-1] > b[r]$. We maintain the invariant that $b[lo] > b[hi]$ for two indices $lo < hi$. Initially, we set $lo = 0$ and $hi = n - 1$. Repeat the following:

- If $hi = lo + 1$, return hi.
- Set $mid = (lo + hi)/2$ and compare $b[mid]$ to $b[hi]$.
- If $b[mid] < b[hi]$, then set $hi = mid$.
- Else if $b[mid] > b[hi]$, then set $lo = mid$.

The number of compares is $\sim 2\log_2 n$ in the worst case, $\sim \log_2 n$ to find r and $\sim \log_2 n$ to search for x.

Solution 2. This solutions searches for the search key x without necessarily finding the crossover index r. We maintain the invariant that if x is in the array b, then it is in $b[lo..hi]$. Initially, we set $lo = 0$ and $hi = n - 1$. Repeat the following until true or false is returned:

- If $hi < lo$, return false.
- Set $mid = (lo + hi)/2$ and compare x to $b[mid]$.
- If $x = b[mid]$, return true.
- Else if $x < b[mid]$,
  - If $b[lo] \leq x$, do a regular binary search for x in $b[lo..mid-1]$.
  - Else set $lo = mid + 1$.
- Else if $x > b[mid]$
  - If $b[hi] \geq x$, do a regular binary search for x in $b[mid+1..hi]$.
  - Else set $hi = mid - 1$.

The number of 3-way compares of x to array entries is at most $\sim 2\log_2 n$ in the worst case since each iteration performs at most 2 compares and halves the length of the subarray to search.

We also remark that there is no way to solve the problem in logarithmic time if duplicate keys are permitted. To see why, consider an array b containing all 0s, but with a single 1 somewhere in the interior. Note that b is a circular shift of a sorted array. Now, there is no efficient way to search for the key 1.

8. **Data structure design.**

   There are many different viable approaches. The simplest solution is to maintain a single symbol table that maps each key to the previously inserted key.

```
import java.util.HashMap;

public class ThreadedSet {
    private String last = null;          // last key inserted
    private HashMap<String, String> st;  // st.get(s) = key inserted
                                         // immediately before s
    public ThreadedSet() {
        st = new HashMap<String, String>();
    }

    public void add(String s) {
        if (st.containsKey(s)) return;  // ignore if already in set
        st.put(s, last);
        last = s;
    }

    public boolean contains(String s) {
        return st.containsKey(s);
    }

    public String previousKey(String s) {
        if (!st.containsKey(s)) throw new RuntimeException();
        return st.get(s);
    }
```

   If we use a hash table for the underlying symbol table, then we expect each operation to take constant amortized time (subject to the uniform hashing assumption); if we use a red-black BST, then each operation takes logarithmic time in the worst case.