COS 226	Algorithms and Data Structures	Fall 2014

Midterm Solutions

1. **Memory and data structures.**

   $\sim 64N$ bytes.

   Each of the `N` `Node` objects uses 64 bytes: 16 (object overhead) + 8 (inner class overhead) + 16 (reference to two `Node`) + 8 (reference to `Key`) + 8 (reference to `Value`) + 4 (integer) + 1 (boolean) + 3 (padding).

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

   0 6 7 5 4 2 8 3 1 9

   6. Quicksort (3-way, no shuffle) after first partitioning step
   7. Heapsort after heap construction phase and putting 12 keys into place
   5. Quicksort (standard, no shuffle) after first partitioning step
   4. Mergesort (bottom-up) after forming sorted subarrays of size 8
   2. Insertion sort after 16 iterations
   8. Knuth shuffle after 16 iterations
   3. Mergesort (top-down) just before the last call to merge()
   1. Selection sort after 16 iterations

3. **Analysis of algorithms.**

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

   (b) $\sim \frac{3}{32}N^2$
   The total number of inversions is $3+6+9+\ldots+\frac{3}{4}N = 3(1+2+3+\ldots+\frac{1}{4}N) \sim \frac{3}{32}N^2$ because the ith A is inverted with $3i$ Bs.

   (c) $\sim \frac{5}{8}N \log_2 N$
   In general, we will be merging two arrays of the form $AAAABBBBBBBBBBBB$ and $AAAABBBBBBBBBBBB$, k As followed by $3k$ Bs. If the merged array is of length N, then this will take $\frac{5}{8}N$ compares. Thus, the total number of compares satisfies the recurrence $T(N) = 2T(N/2) + \frac{5}{8}N$

4. **Balanced search trees.**

insertion key	*color flips*	*rotations*	*key in root*
17	2	0	18
1	2	2	18
31	2	3	28
19	0	0	18

5. **Hash tables.**

   A D X

   - A could be last (L S M N X D A)
   - D could be last (L S M N A X D)
   - L could not be last (otherwise S would end up in 6)
   - M could not be last (otherwise N or A would end up in 1)
   - N could not be last (otherwise A would not end up after 2)
   - S could not be last (otherwise M would end up in 0)
   - X could be last (L S M N A D X)

6. **Problem identification.**

   P Determine whether there are any intersections among a set of N axis-aligned rectangles in $N \log N$ time.

   This was done in the geometric applications of BST lecture.

   P Stably sort a singly *linked list* of N comparable keys using only a constant amount of extra memory and $\sim N \log_2 N$ compares.

   Merging two sorted linked lists of total length N can be done with at most N compares and constant extra space. Bottom-up mergesort avoids extra memory association with the function-call stack.

   I Given a binary heap of N distinct comparable keys, create a binary search tree on the same set of N keys, using at most $2N$ compares.

   This would violate the sorting lower bound. To see why, recall that you can construct a heap from N keys using at most $2N$ compares. Then, you could use the (hypothetical algorithm for creating a BST from the heap using $2N$ compares. Finally, an inorder traversal of the BST yields the keys in sorted order, using no extra compares. Thus, we could sort using at most $4N$ compares.

   P Uniformly shuffle an array in linear time using only constant memory (other than the input array), assuming access to a random number generator.

   The Knuth shuffle accomplishes this.

   P Find the kth smallest key in a left-leaning red-black BST in logarithmic time.

   This is the select *operation in an ordered symbol table.*

P Implement a FIFO queue using a resizing array, in constant amortized time per operation.

This was described in lecture and the textbook.

P Given an array $a[]$ of $N \geq 2$ distinct comparable keys (not necessarily in sorted order) with $a[0] < a[N-1]$, find an an index i such that $a[i] < a[i+1]$ in logarithmic time.

Use the following algorithm, similar to binary search. Maintain the invariant that $a[i] < a[j]$*; initially* $i = 0$ *and* $j = N - 1$*. Pick* $mid = (i + j)/2$*. There are three cases*

- *if* $j = i + 1$*, return* i
- *else if* $a[mid] < a[j]$*, then set* $i = mid$
- *else if* $a[mid] > a[j]$*, then set* $j = mid$

7. **Multiway merge.**

(a) This is similar to multiway merging algorithm on pp. 321–322 of the textbook.

- Initialize a min-oriented priority queue with k items, one corresponding to the first key in each of the k arrays. An *item* consists of a
  - a comparable key (from one of the k arrays)
  - an integer index (to indicate from which of the k arrays the key comes)
  - another integer index (to indicate its position in that array)

  and its natural order is based on the comparable key.
- Repeat until the priority queue is empty:
  - Delete the minimum item from the priority queue.
  - Prints its key.
  - If the key in the deleted item is not the last key in its array, insert into the priority queue a new item corresponding to the next key from its array.

For efficiency, implement the priority queue with a binary heap.

(b) $N \log k$

The binary heap never has more than k items at any one time. Thus, every operation takes time proportional to $\log k$. Each item is inserted and deleted exactly once, so the order of growth of the overall running time is $N \log k$.

(c) k

The algorithm uses extra memory both for the binary heap and for the items. The algorithm creates a total of N items but there are never more than k in existence at any one time.

It is possible to achieve the same performance guarantees using a red-black BST, but some care is needed to handle duplicate keys.

8. **Move-to-front.**

The core idea is to use an ordered symbol table to store the items in the sequence, where the i^{th} largest item in the symbol table is the i^{th} item in the sequence. The symbol table keys are integers (that we assign to achieve the desired ordering) and the symbol table values are the items. To assist with dispensing symbol table keys, we maintain an instance variable `min` that is the minimum key used so far.

- *item-at-index*: To get the i^{th} item in the sequence, we use the *select* operation.
- *add*: Whenever we add a new item, we want it to be smaller than any existing key, so that it becomes the first item in the sequence.
- *move-to-front*: Delete the i^{th} item in the sequence and then *add* it back to the front of the sequence.

For efficiency, implement the ordered symbol table with a red-black BST.

Note: this technique is similar to the solution for question 7 on the Spring 2014 midterm.

```
public class MoveToFront {
    private long min;
    private RedBlackBST<Integer, Item> st;

    public MoveToFront() {
        min = 0;
        st = new RedBlackBST<Integer, Item>();
    }

    public void add(Item item) {
        st.put(--min, item);
    }

    public Item itemAtIndex(int i) {
        return st.get(st.select(i));
    }

    public void mtf(int i) {
        int r = st.select(i);
        Item item = st.get(r);  // itemAtIndex(i)
        st.delete(r);
        add(item);
    }
}
```

The *add*, *item-at-index*, and *move-to-front* operations each take logarithmic time in the worst case (because *put*, *get*, *select*, and *delete* take logarithmic time in the worst case).