COS 226 Algorithms and Data Structures Fall 2017

Midterm Solutions

1. **Memory and data structures.**

   $\sim 40n$ bytes

   Each node object requires 32 bytes: 16 (object overhead) + 4 (int) + 8 (reference) + 4 (padding). So, the node objects themselves consume $32n$ bytes. The array of references to the n objects consumes an additional $\sim 8n$ bytes.

2. **Five sorting algorithms.**

   0 *original input*

   5 *quicksort (after first partition)*

   2 *insertion sort (after 16 iterations)*

   1 *selection sort (after 12 iterations)*

   3 *mergesort (just after first subarray in second half is sorted)*

   4 *heapsort (after heap construction phase and putting 12 largest keys into place)*

   6 *sorted*

3. **Analysis of algorithms.**

   (a) $\sim 2n^2$

   Selection sort always makes $\sim \frac{1}{2}m^2$ compares to sort an array of length m. Here $m = 2n$.

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

   Each integer i in the right half is inverted with with $n - i$ elements in the left half. So, the number of inversions is $(n-1) + (n-2) + \ldots + 1 + 0 \sim \frac{1}{2}n^2$. The number of compares and inversions differ by at most n.

   (c) $\sim n \log_2 n$

   Consider the top level of the recursive algorithm. Sorting the left and right subarrays take $\frac{1}{2} n \log_2 n$ compares each. (Recall, a sorted array is the best case for mergesort.) Merging the two subarrays (each of length n) takes an additional $2n - 1$ compares.

4. **Red–black BSTs.**

   (a) 28 31

   (b) left rotate 18, left rotate 30, right rotate 32

   The sequence of elementary operations is: left rotate 30, right rotate 32, color flip 31, color flip 28, left rotate 18.

5. **Hash tables.**

   - B N
   - O
   - B
   - O R

6. **Programming assignments.**

   (a) n^3
   Opening a site takes a constant number of union–find operations. There are $m = n^2$ sites, so each union–find operation takes time proportional to $\sqrt{m} = n$ time. Performing a percolation experiment involves opening $0.593n^2$ sites, on average.

   (b) *All operations except* `removeLast()` *can be implemented in constant time, by maintaining references to both the first and last nodes in the singly linked list.*

   (c) *The pruning rule guarantees to return a nearest neighbor and achieves $\log n$ time per operation on inputs likely to arise in practice. (Recall that if the points in the 2d tree are all on the circumference of a circle and the query point is in the center, then the nearest-neighbor search will examine every node in the 2d tree.)*

7. **Data structure and algorithm properties.**

   E G A E C I

   The last two are the most difficult.

   - The black links in a red–black BST correspond to the links in a 2–3 tree. The minimum height of a 2–3 tree with n nodes is $\sim \log_3 n$ (all 3-nodes).
   - Here's an example sequence of $10n + 9$ push and pop operations that trigger $\sim 2n$ resizing array operations. Push 9 items (array expands to length 16). Then, pop 5 items (array shrinks to length 8) and push 5 items (array expands to length 16); repeat n times.

8. **Largest bandwidth.**

   The key insight is that the the bandwidth demand at time t is the sum of the bandwidths of the intervals that start before time t *minus* the sum of the bandwidths of the intervals that end before time t. This insight suggests the following *sweep line* algorithm:

   - Create one array with the intervals *sorted by left endpoint* and another array with the intervals *sorted by right endpoint.*
   - Initialize `sum = 0`.
   - Scan the two arrays, selecting the next largest left or right endpoint.
     – If it is a left endpoint, increase `sum` by the bandwidth of the interval.
     If this is the largest `sum` seen so far, save the champion sum and time.
     – If it is a right endpoint, decrease `sum` by the bandwidth of the interval.

   The bottleneck is sorting. By using mergesort (or heapsort), we guarantee $n \log n$ time.

9. **Data structure design.**

The core idea is to maintain the strings in *two* separate data structures: a *queue* (to keep the items in FIFO order) and a *set* or *symbol table* (to identify duplicates).

```
import java.util.HashSet;
import edu.princeton.cs.algs4.Queue;

public class UniQueue {
   private Queue<String> queue = new Queue<String>();     // singly linked list
   private HashSet<String> set = new HashSet<String>();   // linear probing

   // add s to the uni-queue (assuming it is not already in the uni-queue)
   public void enqueue(String s) {
       if (!set.contains(s)) {
          queue.enqueue(s);
          set.add(s);
       }
   }

   // remove and return the item least recently added to the uni-queue
   public String dequeue() {
      String result = queue.dequeue();
      set.remove(result);
      return result;
   }
}
```

The running time depends upon the choice of queue and set.

- If we use a *singly linked list* for the queue and a *linear-probing hash table* for the set, then the average time per operation is constant. This analysis relies on the uniform hash assumption (for hashing). It is an amortized bound because the hash table uses a resizing array. (It's also fine to use a *resizing array* for the queue and/or a *separate-chaining hash table* for the set.)
- If we use a *singly linked list* for the queue and a *red–black BST* for the set, then the time per operation is $\log n$ in the worst case.

Using a symbol table instead of a set is also acceptable (but there is no need to waste memory for the values).