COS 226 **Algorithms and Data Structures** **Fall 2025**

Midterm Solutions

1. **Initialization.**

   You will lose points if you neglect to write your name; select the wrong precept; go to the wrong room; or fail to write and sign the honor code.

2. **Performance.**

   (a) $\sim 96n$ bytes

   The amount of memory is dominated by the two arrays of length m: $4m$ bytes for the `int[]` *and $8m$ bytes for the* `String[]` *array, for a total of $\sim 12m$ bytes. The worst-case memory usage arises just before a shrinking operation, at which point the array is $\frac{1}{8}$ full and $m = 8n$.*

   Very similar to the question asked in `readme.txt` *on Assignment 2.*

   (b) $\Theta(n \log n)$

   Each `insert()` *and* `delMin()` *operation takes $O(\log n)$ amortized time, so any sequence of $2n$ such operations takes $O(n \log n)$ time in the worst case. Since the code fragment is a compare-based sorting algorithm, the sorting lower bound implies that it makes $\Omega(n \log n)$ time in the worst case.*

   This was the last iClicker question in the Priority Queues lecture.

   (c) $\sim n^4 \log_2 n$

   For each i and j, the k loop iterates $\log_2(n^2) \sim 2 \log_2 n$ times. The outer two loops iterate $\sim \frac{1}{2}n^4$ times. Multiplying together yields $\sim n^4 \log_2 n$.

   (d) $O(n^2)$, $O(n^3)$, $\Omega(\log n)$, $\Omega(n)$, $\Omega(n^2)$

   Big O notation provides an upper bound; Big Omega notation provides a lower bound.

3. **Data structures.**

   (a) $(4, 0)$, $(5, 0)$, $(8, 9)$

   The link 0→4 must be the last link added to the big tree (rooted at 4). If the last linked added were 5→4 or (6→4), then, just prior to adding this link, the tree rooted at 4 would have height 3 but only 6 (or 7) nodes. In weighted quick union, any tree of height h must have at least 2^h nodes.

   (b) $(10, 77)$, $(75, 77)$, $(77, 80)$

   (c) 1. color flip 60
   2. right rotate 90
   3. color flip 75
   4. left rotate 45

4. **Five sorting algorithms.**

C D F E B

C. *Insertion sort after 16 iterations*

D. *Mergesort just before the second-to-last last call to* `merge()`

F. *Heapsort after heap construction phase*

E. *Quicksort after first partitioning step*

B. *Selection sort after 12 iterations*

5. **Analysis of a sorting algorithm.**

(a) ☐ ■ ☐ ■

At the beginning of each iteration i, the last i array elements contain the i largest element in the array, and they are in sorted order. This is a version of selection sort (known as bubblesort*), where the largest elements are put in place first.*

(b) yes, in-place

Only $\Theta(1)$ extra memory is used.

(c) yes, stable

Two elements are exchanged if and only if they are adjacent and the element to the right is strictly smaller than the element to the left. So, the relative order of equal keys remains the same.

(d) C C H

- In each iteration of the i loop (except the last), there is exactly 1 exchange—exchanging the element 1 with the element immediately to its left. So, the total number of exchanges is $n-1$. (The total number of compares is $\sim \frac{1}{2}n^2$, but this question is about exchanges, not compares.)
- A best-case input is an array of n keys in ascending order. When $i = 0$, there will $n-1$ compares and zero exchanges. Since there are no exchanges when $i = 0$, the algorithm will terminate.
- A worst-case input is an array of n keys in strictly descending order. Iteration i makes $n-i-1$ compares and $n-i-1$ exchanges. So, the total number of compares is $(n-1)+(n-2)+\ldots+1+0 \sim \frac{1}{2}n^2$.

6. **Properties of algorithms and data structures.**

   F T T F T F

   - In the best case (e.g., all equal keys), mergesort makes only $\sim \frac{1}{2} m \log_2 m$ compares to sort an array of length m. Here, $m = 2n$.
   - From precept, we know that any compare-based algorithm for *searching for a key in a sorted array* of length m must make $\Omega(\log m)$ compares in the worst case. Here, $m = 3n$.
   - The number of partitioning steps in 3-way quicksort is less than (ore equal to) the number of distinct keys k. In this problem, $k = 4$. Each partitioning step makes $\leq n$ compares. So, there are $\leq 4n$ compares in total.
   - Consider the binary heap from Q3 (b), with $x = 10$ at depth 2 and $y = 60$ at depth 3.
   - Take a level-order traversal of the two BSTs. Then, merge the two sorted sequences, each of length n, together using the merging algorithm from mergesort. Finally, form the new BST (e.g., by building a completely unbalanced BST with each successive key being the right child of the previous one). No compares are needed for either the level-order traversals or building the BST; the merging step makes $\leq 2n$ compares.
   - During an LLRB insertion, there is never more than one right-leaning red link at a time. A precondition for performing a left rotation is the existence of a right-leaning red link. The left rotation eliminates the right-leaning red link. So, before the next left rotation, there must be some intervening operation.

7. **Binary search.**

   E I A I I L

   This is a special case of `firstIndexOf()` *from Assignment 3. It maintains the two loop invariants (*`a[lo] = 0` *and* `a[hi] = 1`*) from the comments. So, when the loop terminates, we know that the first 1 is at index* `hi`*.*

   We note that D I A J I F and D I A J I L also solve the problem and were awarded full credit (despite not maintaining the invariant `a[lo] = 0`).

8. **Data structure design.**

Full-credit solution.

The main idea is to combine two data structures:

- A *doubly linked list* that maintains the uni-stack items in order (from most recently added to least recently added), with no duplicates. It must be *doubly linked* to support efficient deletion of an arbitrary node.
- A *red–black BST* that maps from strings on the uni-stack to the corresponding doubly linked list nodes. This symbol table has two roles: (1) to determine efficiently whether a string is already on the uni-stack and (2) if so, to get a reference to the corresponding linked list node (so that the node can be efficiently deleted).

(a)
```
public class UniStack {
    private RedBlackBST<String, Node> st;
    private Node first;        // first node in the linked list

    // node in a doubly linked list
    private static Node {
        private String item;   // the string
        private Node next;     // next node in the linked list
        private Node prev;     // previous node in the linked list
    }
```

(b)
- If `item` is in the symbol table, get the linked list node containing `item` and delete that node from the linked list.
- Create a new node `x` containing `item`; add `x` to the front of the linked list; and update `first`
- Update the symbol table by mapping `item` to the newly created linked list node `x`.

(c)
- Delete the first node from the linked list, updating `first`; let `s` be the string in that node.
- Delete `s` from the symbol table.
- Return `s`.

All symbol table operations take $O(\log n)$ time in the worst case. All linked list operations take $O(1)$ time in the worst case. The amount of memory for the red–black BST and the doubly linked list are each $\Theta(n)$.

With this implementation, it's also easy to implement `size()` in $O(1)$ time since the number of strings on the uni-stack is always equal to the number of string keys in the symbol table.

Alternative full-credit solution. Maintain an integer counter (that counts the number of uni-stack push operations) and two symbol tables:

- Symbol table 1: map the string s to the integer k if s is currently on the uni-stack and s was the k^{th} string pushed onto the uni-stack.
- Symbol table 2: map the integer k to the string s if s is currently on the uni-stack and s was the k^{th} string pushed onto the uni-stack.

The string associated with the largest integer in the second symbol table is the one that was added most recently to the uni-stack.

(a)
```
public class UniStack {
    private int counter = 0;  // increments after each call to push()
    private RedBlackBST<String, Integer> st1;
    private RedBlackBST<Integer, String> st2;
```

(b)
- If `item` is in `st1`, get the corresponding integer `k` and delete `k` from `st2`.
- Map `item` to `counter` in `st1`; map `counter` to `item` in `st2`.
- Increment `counter`.

(c)
- Get the largest integer `k` in `st2`.
- The string `s` to pop is `st2.get(k)`.
- Delete `s` from `st1`; delete `k` from `st2`.
- Return `s`.

Implement both symbol tables using red–black BSTs so that all operations take $O(\log n)$ time in the worst case.

Partial credit solution. Maintain a *stack* along with a *symbol table* to keep track of the strings on the stack.

- *push*: If `item` is already in the symbol table, do nothing. Otherwise, push `item` onto the stack and add `item` to the symbol table.
- *pop*: pop from the stack (and return the popped string); delete the popped string from the symbol table.

Implement the stack with a singly linked list and the symbol table with a red–black BST so that all operations take $O(\log n)$ time in the worst case.