COS 226 **Algorithms and Data Structures** **Fall 2026**

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 pledge.

2. **Memory.**

   (a) 72 bytes

   A `Node` *object has 16 bytes of object overhead, five 8-byte references (two* `String` *references and three* `Node` *references), and two 8-byte* `double` *values. Thus, the total is* $16 + 5(8) + 2(8) = 72$ *bytes. Since* `Node` *is static, there is no reference to an enclosing* `TwoThreeTree` *object.*

   (b) $\min \sim \frac{1}{2}n, \qquad \max \sim n$

   Each node contains either one or two key–value pairs. The minimum number of nodes occurs when each node contains two key–value pairs; the maximum occurs when each node contains one key–value pair.

3. **Data structures.**

(a) i. (p,q), (q,p), (x,y), (y,x)

The tree rooted at y has size 10. As shown below, the tree rooted at x has size at most 8, so weighted quick union links x to y regardless of which of the two components supplies the first argument. The pair (x,x) performs no link, and (r,y) links r to y.

ii. 1, 2, 3, 4

Besides subtree A, the tree rooted at x contains four nodes. When the root of A was linked directly to x, the component rooted at x could contain at most those four other nodes. By the link-by-size rule, A therefore has size at most 4. Each size from 1 through 4 is possible.

iii. 0, 1, 2

In weighted quick union, a tree of height h has at least 2^h nodes. Since subtree A has at most 4 nodes, its height is at most 2. Each height 0, 1, and 2 is possible.

(b) 71, 77

The new key is inserted as the left child of 40. To make exactly two exchanges, it must swim past 40 and 70 but not past 80. Thus, $70 < x < 80$. The third compare is with 80.

(c) 1. color flip 8
2. rotate left 4
3. rotate right 10
4. color flip 8

Inserting 9 creates two red children at 8, triggering a color flip. On the way back up the search path, the standard LLRB fix-up rules then trigger a left rotation at 4, a right rotation at 10, and a color flip at 8.

4. **Five sorting algorithms.**

C D B E F

C. *Insertion sort after 8 iterations*

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

B. *Selection sort after 12 iterations*

E. *Quicksort after the first partitioning step*

F. *Heapsort after heap construction and putting 12 keys into place*

5. **Analysis of algorithms and sorting.**

   (a) $\sim 8n^2$

   Selection sort makes one compare for every pair of array positions. Since the array has length $4n$, the number of compares is $\sim \frac{1}{2}(4n)^2 \sim 8n^2$.

   (b) $\sim \frac{3}{2}n^2$

   The leading term is the number of inversions. Each 6 precedes three 2s in every later block, so the number of inversions is

   $$3[(n-1)+(n-2)+\cdots+1] \sim \frac{3}{2}n^2.$$

   The additional unsuccessful compares contribute only $O(n)$.

   (c) $\sim 5n$

   The first 3-way partition uses pivot 2 and makes $4n$ compares. All $3n$ copies of 2 are then in place, leaving a subarray containing n copies of 6. Partitioning that subarray makes another n compares. Thus, the total is $5n$.

6. **Percolation.**

   (a) `(i * n * n) + (j * n) + k`

   This is the usual row-major mapping from three coordinates to one index. Any equivalent one-to-one mapping to $0, \ldots, n^3 - 1$ is also correct.

   (b) 6

   An interior site has six neighboring sites. A site on the top or bottom face has at most five grid neighbors plus one virtual-site union, so six is still the maximum.

   (c) 2

   Check whether the virtual top and virtual bottom sites are in the same set, using one `find()` *call for each:* `find(n*n*n) == find(n*n*n + 1)`.

   (d) $\Theta(n^6)$

   The union–find data structure has $n^3 + 2$ elements, so each union operation takes $\Theta(n^3)$ time. An experiment opens $\Theta(n^3)$ sites and performs $\Theta(1)$ union operations per opened site. Thus, the overall running time is $\Theta(n^6)$.

7. **Linked lists.**

A B B D D

A. *Maintain a pointer to the first node. Both* `addFirst()` *and* `removeFirst()` *take constant time.*

B. *Maintain pointers to the first and last nodes in a singly linked list. Then* `addFirst()`*,* `addLast()`*, and* `removeFirst()` *take constant time.*

B. *Store the items in reverse order, with the logical last item at the beginning of the linked list. With pointers to both ends,* `addLast()` *and* `removeLast()` *operate at the beginning of the list, while* `addFirst()` *operates at the end.*

D. *Both ends must be removable in constant time. A singly linked list cannot remove its last node in constant worst-case time, and a doubly linked list with only one node pointer cannot access both ends in constant time. Two pointers in a doubly linked list suffice.*

D. *The standard doubly linked list with pointers to the first and last nodes supports all four deque operations in constant worst-case time.*

8. **Binary search.**

B H G I E F

Since `a[]` *is strictly increasing and* `b[]` *is strictly decreasing, the quantity* `a[i] - b[i]` *is strictly increasing. Thus, ordinary binary search applies. If* `a[mid] < b[mid]`*, search to the right; if* `a[mid] > b[mid]`*, search to the left; otherwise, return* `mid`*.*

The completed code is:

```
public static int findCrossingIndex(int[] a, int[] b) {
    int lo = 0;
    int hi = a.length - 1;
    while (hi >= lo) {
        int mid = lo + (hi - lo) / 2;
        if      (a[mid] < b[mid]) lo = mid + 1;
        else if (a[mid] > b[mid]) hi = mid - 1;
        else return mid;
    }
    return -1;
}
```

Note any solution that starts with C (instead of B) can fail to find a matching index, e.g., if both arrays are of length 1 and contain the same integer. Solutions like C H F I F F may also go into an infinite loop when there is no matching index.

9. **Sum of the k smallest.**

Full-credit solution (two red-black BSTs).

Maintain two red–black BSTs:

- `small` contains the $\min(k, n)$ smallest integers in the set.
- `large` contains all remaining integers.

Also maintain the sum of the integers in `small`. The invariant is that every key in `small` is smaller than every key in `large`.

(a)
```
public class KSmallestSum {
    private final int k;
    private int sum;
    private RedBlackBST<Integer, Boolean> small;
    private RedBlackBST<Integer, Boolean> large;
}
```
Initialize `sum` *to 0 and both symbol tables to empty red–black BSTs in the constructor.*

(b)
```
return sum;
```
By the invariant, `small` *contains exactly the integers that must be summed. Thus,* `sum()` *takes* $\Theta(1)$ *time.*

(c) For `add(x)`:

- If x is already in either BST, do nothing.
- If `small.size() < k`, insert x into `small` and add x to `sum`.
- Otherwise, if $x <$ `small.max()`, move `small.max()` from `small` to `large`, updating `sum`, and insert x into `small`, again updating `sum`.
- Otherwise, insert x into `large`.

Only $\Theta(1)$ *red–black BST operations are performed, so* `add()` *takes* $O(\log n)$ *time.*

(d) For `remove(x)`:

- If x is in `small`, delete it and subtract x from `sum`. If `large` is nonempty, move `large.min()` from `large` to `small`, adding it to `sum`.
- Otherwise, if x is in `large`, delete it.
- If x is absent from both BSTs, do nothing.

Only $\Theta(1)$ *red–black BST operations are performed, so* `remove()` *takes* $O(\log n)$ *time.*

Alternative full-credit solution (one red–black BST).

Maintain all integers in one red–black BST `st`, together with the sum of the $\min(k, n)$ smallest integers. Use `rank()` to determine whether an integer belongs among the k smallest and `select()` to identify the integer that enters or leaves that group after an update.

(a)
```
public class KSmallestSum {
    private final int k;
    private int sum;
    private RedBlackBST<Integer, Boolean> st;
}
```
Initialize `sum` *to 0 and* `st` *to an empty red–black BST in the constructor.*

(b)
```
return sum;
```
Since `sum` *is maintained after every update,* `sum()` *takes* $\Theta(1)$ *time.*

(c) For `add(x)`:

- If `st.contains(x)`, do nothing.
- If `st.size() < k`, add x to `sum`.
- Otherwise, if `st.rank(x) < k`, add x – `st.select(k - 1)` to `sum`.
- Insert x into `st`.

If x has rank less than k, it enters the k smallest integers and displaces the old integer of rank $k-1$; otherwise, the k smallest integers do not change. Only $\Theta(1)$ red–black BST operations are performed, so `add()` *takes $O(\log n)$ time.*

(d) For `remove(x)`:

- If `!st.contains(x)`, do nothing.
- If `st.size() <= k`, subtract x from `sum`.
- Otherwise, if `st.rank(x) < k`, add `st.select(k)` to `sum` and subtract x.
- Delete x from `st`.

If x has rank less than k, it leaves the k smallest integers and the old integer of rank k takes its place; otherwise, the k smallest integers do not change. Only $\Theta(1)$ red–black BST operations are performed, so `remove()` *takes $O(\log n)$ time.*

Half-credit solutions.

- For $O(n)$ updates and $\Theta(1)$ `sum()`, maintain all integers in any convenient data structure (e.g., a linked list, resizable array, BST, or binary heap) together with the desired sum; after each successful `add()` or `remove()`, recompute the sum of the k smallest integers in $O(n)$ time.

- For $O(\log n)$ updates and $O(k \log n)$ `sum()`, maintain all integers in one red–black BST. Then `add()` and `remove()` are standard symbol-table operations; compute `sum()` by obtaining the keys of ranks $0, 1, \ldots, k-1$ (or all keys if $n < k$) and adding them.