COS 226	Algorithms and Data Structures	Fall 2020

Midterm Solutions

1. **Initialization.** Don't forget to do this.

2. **Memory.**

   A `LinearProbingHashTable` object uses $12m + 88 \sim 12m$ bytes of memory, where m denotes the lengths of the `keys[]` and `vals[]` arrays. Here's a complete accounting (even though, for the purposes of this question, it would suffice to focus only on the two arrays since we will be discarding lower-order terms).

   - 16 bytes of object overhead
   - 4 bytes for the integer n
   - 4 bytes of padding
   - 8 bytes for the reference to the `keys[]` array
   - 8 bytes for the reference to the `vals[]` array
   - $24 + 4m$ bytes for `keys[]` array (plus 4 bytes of padding if m is odd)
   - $24 + 8m$ bytes for `vals[]` array

   (a) $\sim 12n$
   If the hash table is full, then $m = n$.

   (b) $\sim 96n$
   In the worst case, the hash table is 1/8 full, so $m = 8n$.

3. **Five sorting algorithms.**

   (3.1) *mergesort just before the last call to* `merge()`

   (3.2) *quicksort after first partitioning step*

   (3.3) *insertion sort after 16 (or 17) iterations*

   (3.4) *selection sort after 12 iterations*

   (3.5) *heapsort immediately after the heap construction phase*

4. **Analysis of algorithms.**

   (4.1) $\sim \frac{1}{2}n^2$
   Selection sort makes $\sim \frac{1}{2}n^2$ *compares to sort any array of length* n.

   (4.2) $\sim \frac{1}{8}n^2$
   In each of the first $n/2$ *iterations, there are* i *exchanges in iteration* i. *In each of the next* $n/2$ *iterations, there are* 0 *exchanges. So, the total number of exchanges is* $0 + 1 + 2 + \ldots + (n/2 - 1) \sim \frac{1}{8}n^2$. *The number of compares in insertion sort is always within* n *of the number of exchanges.*

(4.3) $\sim \frac{1}{2} n \log_2 n$

Mergesort makes $\sim \frac{1}{2} n \log_2 n$ *compares to sort a sorted (or reverse sorted) array of length* n*. Thus, mergesort makes* $\sim \frac{1}{4} n \log_2 n$ *compares to sort the left subarray (of length* $n/2$*) and* $\sim \frac{1}{4} n \log_2 n$ *compares to sort the right subarray (of length* $n/2$*). Finally, it makes* $n/2$ *compares to merge the two subarrays together.*

(4.4) $O(n^2)$, $O(n^2 \log n)$, $O(n^3)$, $\Theta(n^2)$

$f(n) = 1 + 2 + 4 + 8 + \ldots + n^2 = 2n^2 - 1$ (*geometric sum from array resizing*)

Big O and big Theta notations discard both lower-order terms and the leading coefficient. The main difference is that big O notation includes functions that grow more slowly. So, $O(n^2 \log n)$ *includes not only functions like* $2n^2 \log n$ *and* $\frac{1}{2} n^2 \log n$*, but also* $2n^2 - 1$ *and* $\frac{1}{8} n$*.*

5. **1D range search.**

(5.1) A C H F J D I

```
private void range(Node x, int lo, int hi) {
    if (x == null) return;
    if (x.key > lo) range(x.left, lo, hi);
    if (x.key >= lo && x.key <= hi) StdOut.println(x.key);
    if (x.key < hi) range(x.right, lo, hi);
}
```

(5.2) $\Theta(m + \log n)$

6. **Representation.**

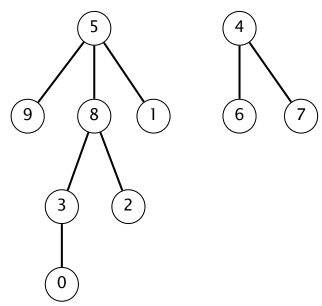
(6.1) return a maximum key
delete a maximum key
return a second largest key
delete a random key

(6.2) insert a key–value pair (*put*)
return the value associated with a given key (*get*)
return the largest key that is less than or equal to a given key (*floor*)

7. **Data structures.**

(7.1) could not arise

The height of the 7-node tree containing is 3. However, the height of any weighted quick-union tree on k elements is at most $\log_2 k$. Note that $\log_2 7 < \log_2 8 = 3$, so the height must be strictly less than 3.



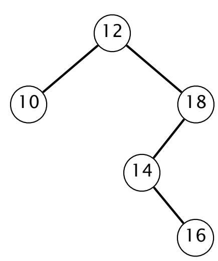
An alternative argument is to consider the point in time when 8 was linked to 5. At this moment, the subtree rooted at 8 contained 4 elements (8, 3, 2, and 0) and the subtree rooted at 5 contained at most 3 elements (5, 9, and 1). Weighted quick union would not have merged the larger tree (rooted at 8) into the smaller tree (rooted at 5).

(7.2) could not arise

It is not a complete binary tree: 30 has no children but 20 has two children.

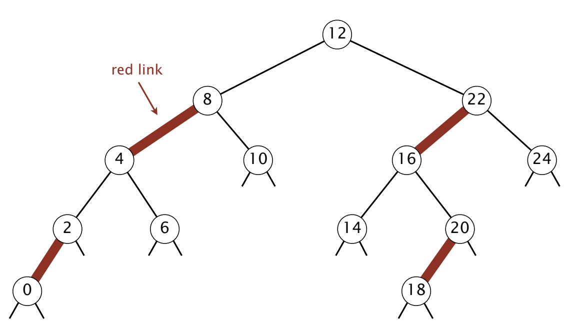
(7.3) could not arise

The start of the BST would must look like the following. But, then there's no way to include 34 so that it ends up in the proper position.



(7.4) could arise

Here is the (unique) coloring of the links that achieves perfect black balance.
(It could have arisen by inserting the keys in level order: 12, 8, 22, 4, 10, ...)



(7.5) could not arise

It's invalid because the left child of $(2, 6)$ *is* $(3, 5)$*. But, the left subtree must contain only points with smaller* x*-coordinates.*

(7.5) could not arise

It's invalid because $hash(F) = 2$*;* F *is stored at index 4; and there is no key at index 3. So, you would not find* F *in the hash table if you searched for it.*

8. **Problem identification.**

8.1 Implement `open()` in $O(\log n)$ time.

Note that `percolates()` *takes* $\Theta(1)$ *time with* quick-find*.*

8.2 Delete at both front and back in constant time.

A doubly linked list uses more memory but enables deletion of not only the first node in the linked list but also the last one.

8.3 Implement `numberOfMatches()` with $O(\log n)$ compares.

Note that `allMatches()` *makes* $\Theta(n \log n)$ *compares in the worst case (when there are* $m = n$ *matches) because it has to sort the* m *matches in descending order by weight.*

8.4 None of the above.

The main reason to use k-d trees is for logarithmic performance in practice (on typical inputs), not worst-case performance. In fact, nearest neighbor search can take $\Theta(n)$ *time in the worst case, even if the k-d tree is balanced.*

9. **Find the missing integer.**

The main idea is to use *binary search*, maintaining a subarray `a[lo..hi]` with the invariant that `a[lo] = lo` and `a[hi] > hi`.

- Initialize $lo \leftarrow 0$ and $hi \leftarrow n-1$
  Since the missing integer is neither 0 nor n, we have $a[0] = 0$ and $a[n-1] = n > n-1$ and our two invariants are satisfied.
- Terminate the loop when $hi = lo + 1$, in which case hi is the missing integer.
  This follows because lo and hi are adjacent indices with $a[lo] = lo$ and $a[hi] > hi$.
- Otherwise,
  - Set $mid = (lo + hi)/2$.
  - If $a[mid] > mid$, then update $hi \leftarrow mid$.
    This ensures $a[hi] > hi$ and maintains $a[lo] = lo$.
  - If $a[mid] = mid$, then update $lo \leftarrow mid$.
    This ensures $a[lo] = lo$ and maintains $a[hi] > hi$.

Here's the corresponding Java code.

```
public static int find1(int[] a) {
    int n = a.length;
    int lo = 0, hi = n - 1;
    while (hi > lo + 1) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] > mid) hi = mid;
        else              lo = mid;
    }
    return hi;
}
```

Here's an alternative solution that maintains the smallest index `champ` *encountered so far with the property that* `a[champ] > champ`.

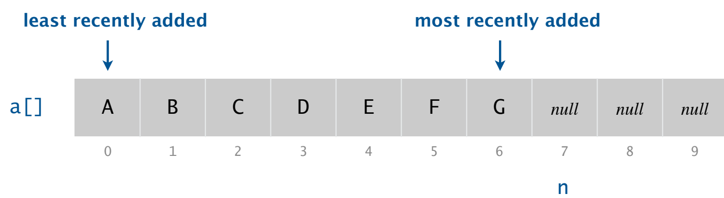
```
public static int find2(int[] a) {
    int n = a.length;
    int lo = 0, hi = n - 1;
    int champ = -1;
    while (hi >= lo) {
        int mid = lo + (hi - lo) / 2;
        if (a[mid] > mid) { champ = mid; hi = mid - 1; }
        else              { lo = mid + 1; }
    }
    return champ;
}
```

Yet another approach is to think of the key of element i as $a[i] - i$ (so the keys are a bunch of 0s followed by a bunch of 1s) and binary search for the first occurrence of 1 via `BinarySearchDeluxe.firstIndexOf()`.

10. **Random-access stack.**

Half-credit solution. We can do this in constant amortized time by copying our resizing array implementation of a stack and implementing the `get()` operation in the natural way.

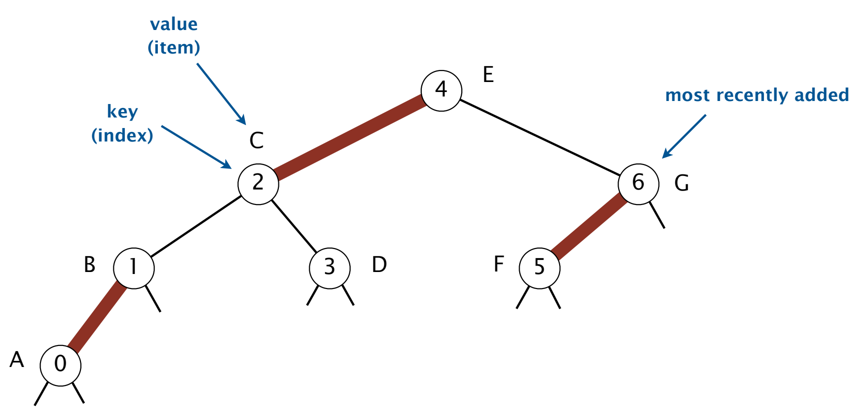
- Store the strings in a resizing array `a[]` with the least recently added string at `a[0]` and the most recently added string at `a[n-1]`.
- `push()`: add the string at index `n` and increment `n`.
- `pop()`: remove the string at index `n-1` and decrement `n`.
- `get()`: return the string at index `n-k-1`.



To ensure that the array has sufficient capacity, double the length of the array when it becomes full and halve the length when it becomes 1/4 full. These array resizing operations take $\Theta(n)$ time, so we don't achieve the $O(\log n)$ worst-case performance requirement.

Full-credit solution. The main idea to achieve a logarithmic performance guarantee is to use a *red–black BST* to represent the array in the half-credit solution. Specifically, we use a symbol table whose keys are the array indices and whose values are the string items, associating the least recently added string with index 0 and the most recently added string with index $n - 1$.

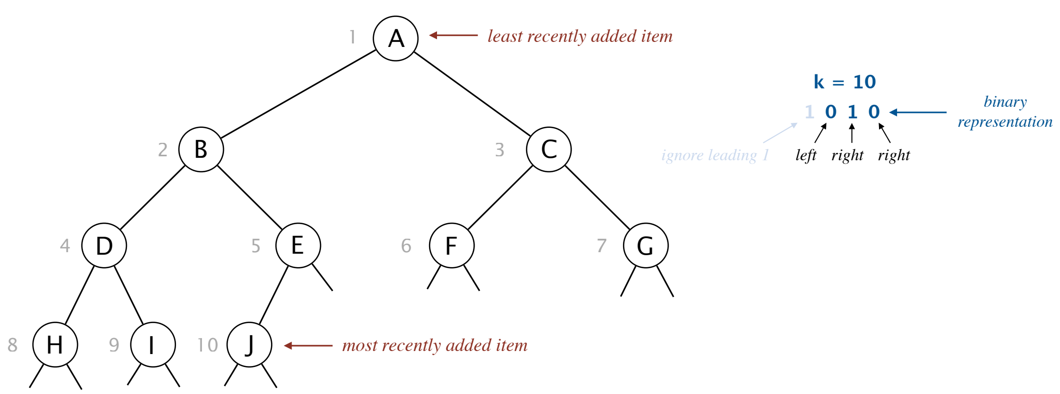
For example, this is a red–black BST corresponding to a stack with the 7 strings A, B, C, D, E, F, and G, in that order and with G at the top.



Here's the Java code. All instance methods take $\Theta(\log n)$ time in the worst case.

```
public class RandomAccessStack {
    private int n;                          // number of elements in stack
    private TreeMap<Integer, String> st;    // st[i] = ith least recently added item

    // create an empty random-access stack
    public RandomAccessStack() {
        n = 0;
        st = new TreeMap<>();
    }

    public void push(String item) {
        st.put(n++, item);
    }

    public String pop() {
        return st.remove(--n);
    }

    public String get(int k) {
        return st.get(n-k-1);
    }
}
```

Alternative full-credit solution. We can use a complete binary tree (instead of a red–black BST) to store the string items, adding the nodes to the tree in *level order*. This reduces memory and avoids the need for the BST rebalancing operations.



You can locate the node at level-order index k in a complete binary tree by examining the binary representation of k and using the bits ($0 =$ left, $1 =$ right) to guide the search. Since the height of a complete binary tree is $\Theta(\log n)$, you can access any node in $O(\log n)$ time.

Challenge-for-the-bored. $\Theta(1)$ time for both *push* and *pop* and $O(\log n)$ time for *get*.