COS 226	Algorithms and Data Structures	Spring 2020

Midterm Solutions

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

2. **Memory.**

   (a) `isEmpty()`, `addFront()`, `removeFront()`, `addBack()`
   To implement `size()`, `removeBack()`, *and* `sample()`, *you would have to traverse the singly linked list, from* `first` *to* `last`. *The challenge with implementing* `removeBack()` *efficiently is updating the* `last` *pointer.*

   (b) $\sim 32n$
   Each `Node` object uses 32 bytes of memory and there are n nodes.

   - 16 bytes of object overhead
   - 8 bytes for `Node` reference
   - 8 bytes for `double` item

3. **Five sorting algorithms.**

   (3.1) *insertion sort after 16 iterations*

   (3.2) *heapsort after heap construction phase and putting 6 keys into place*

   (3.3) *selection sort after 12 iterations*

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

   (3.5) *quicksort after first partitioning step*

4. **Analysis of algorithms.**

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

   (4.2) $\sim n^2$
   In each of the first n *iterations, there are 0 exchanges. In each of the next* $n-1$ *iterations, there are* n *exchanges. So, the total number of exchanges is* $n(n-1)$. *The number of compares in insertion sort is always within an additive factor of* n *of the number of exchanges.*

   (4.3) $\sim n \log_2 n$
   Mergesort requires $\frac{1}{2}n \log_2 n$ *compares to sort a sorted array of length* n. *Thus, mergesort makes* $\frac{1}{2}n \log_2 n$ *compares to sort the left subarray (of length* n*) and* $\frac{1}{2}n \log_2 n$ *compares to sort the right subarray (of length* n*). Finally, it makes* $2n-1$ *compares to merge the two subarrays together, which is a lower-order term.*

(4.4) Timsort

Timsort is optimized for situations when an array has a small number of non-decreasing (or strictly decreasing) runs. In this case, there are only two runs (the first n elements containing the value n, and the last n elements containing the integers 1 to n). So, Timsort will run in linear time on staircase arrays.

(4.5) $O(n^3), O(n^4), \Theta(n^3)$

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, $O(n^4)$ includes not only functions like $2n^4$ and $\frac{1}{2}n^4$, but also $3n^3$ and $5n^2$.

5. **Level-order traversal.**

B F H J L M A

```
public Iterable<Key> levelOrder() {
     Queue<Key> keys = new Queue<Key>();
     Queue<Node> queue = new Queue<Node>();
     queue.enqueue(root);
     while (!queue.isEmpty()) {
         Node x = queue.dequeue();
         if (x != null) {
             keys.enqueue(x.key);
             queue.enqueue(x.left);
             queue.enqueue(x.right);
         }
     }
     return keys;
}
```

6. **Hash tables.**

(6.1) B D F

The first key inserted always ends up at its desired index.

(6.2) B E

We can deduce that F, D, G, and A are inserted before C because C's desired index is 4 but it ends up at 0 (with F, D, G, and A at indices 4, 5, 6, and 7). Similarly, we can deduce that C is inserted before E.

(6.3) D G

We can deduce that D and G are inserted before A because A's desired index is 5 and it ends up at index 7.

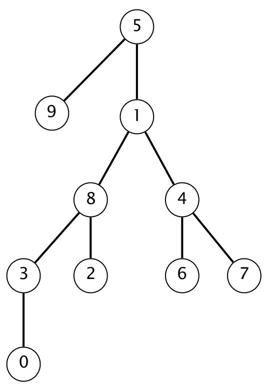
(6.4) C E

We can deduce that C and E are inserted after A because C's desired index is 4 and E's desired index is 6 and both end up after A (at index 7).

7. **Data structures.**

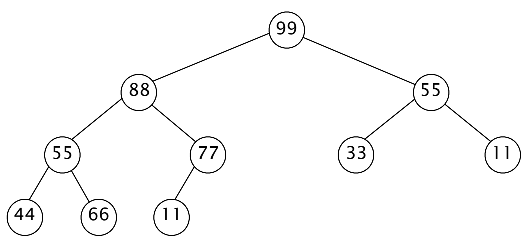
(7.1) could not arise

The height of the tree is 4. However, the height of any weighted quick-union tree on n elements is at most $\log_2 n$. Note that $\log_2 10 < \log_2 16 = 4$, so the height must be strictly less than 4.



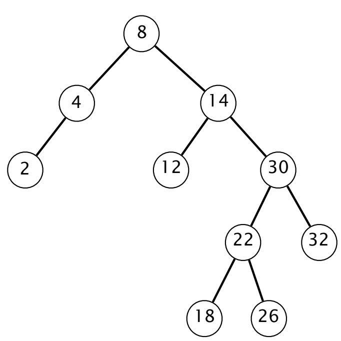
(7.2) could not arise

The corresponding binary tree is not heap-ordered because 55 is less than 66.



(7.3) could arise

Here is the BST with the given level-order traversal.



(7.4) could not arise

Perfect black balance is not satisfied. The path from the root to the right null link of 8 has only 2 black links (including the null link) but all other paths from the root to null links have 3 black links.

(7.5) could arise

It's a valid kd-tree. It could have arisen by inserting the points in a variety of orders, including level order: $(6,7),(1,4),(8,5),(4,2),(2,8),(0,9),(3,6)$.

8. **Problem identification.**

8.1 Possible

This can be done with mergesort, as discussed in lecture.

8.2 Possible

This can be done with 3-way quicksort. The number of 3-way partitioning steps equals the number of distinct keys. Each partitioning step makes at most n *compares.*

8.3 Impossible

This would violate the sorting lower bound. We could insert the n *keys; then delete-max the* n *keys to get them in sorted order. This would give us a compare-based sorting algorithm that makes* $\Theta(n \log \log n)$ *compares in the worst case.*

8.4 Possible

You could use binary search directly. Or you could compose an algorithm by combining operations that we've seen in the course. For example, if k *is not in the array, then the predecessor is the floor (which we saw how to compute using binary search). If* k *is in the array, then you could search for the first occurrence of* k *and return the previous key (which you did on the Autocomplete assignment using binary search).*

8.5 Impossible

This would violate the sorting lower bound. We could insert the n *keys into a BST; then we could perform an inorder traversal to get them in sorted order. Since performing an inorder traversal doesn't require any key compares, this would give us a compare-based sorting algorithm that makes* $\Theta(n)$ *compares in the worst case.*

8.6 Impossible

There may be $\Theta(n^2)$ *pairs that intersect, so it will take* $\Theta(n^2)$ *time to collect them in a list.*

9. **Design question.**

   9.1 true

   9.2 true

   9.3 The main idea is to use *binary search* to find the adjacent inversion, maintaining a subarray $a[lo..hi]$ for which (lo, hi) is an inversion: $lo < hi$ and $a[lo] > a[hi]$.

   - Initialize $lo \leftarrow p$ and $hi \leftarrow q$
   - Terminate the loop when $hi = lo + 1$, in which case (lo, hi) is an adjacent inversion.
   - Otherwise,
     - Set $mid = (lo + hi)/2$.
     - If $a[mid] > a[hi]$, then update $lo \leftarrow mid$.
       This guarantees $a[lo] > a[hi]$.
     - If $a[mid] \leq a[hi]$, then update $hi \leftarrow mid$.
       This guarantees $a[lo] > a[hi]$ because $a[lo]$ stays the same and $a[hi]$ does not increase.

   Here's the corresponding Java code.

```
int lo = p, hi = q;
while (hi > lo + 1) {
    int mid = lo + (hi - lo) / 2;
    if (a[mid] > a[hi]) lo = mid;
    else hi = mid;
}
```

   Here's a symmetric version that compares `a[mid]` *to* `a[lo]`*.*

```
int lo = p, hi = q;
while (hi > lo + 1) {
    int mid = lo + (hi - lo) / 2;
    if (a[lo] > a[mid]) hi = mid;
    else lo = mid;
}
```

   Here's another version that does two compares per iteration of the `while` *loop. The second compare is unnecessary because, if the first compare fails, then it must be the case that* `a[mid] >= a[lo] > a[hi]`*.*

```
int lo = p, hi = q;
while (hi > lo + 1) {
    int mid = lo + (hi - lo) / 2;
    if (a[lo] > a[mid]) hi = mid;
    else if (a[mid] > a[hi]) lo = mid;
}
```