COS 226 Algorithms and Data Structures Fall 2024

Midterm Solutions

1. **Initialization.**

   Don't forget to do this.

2. **Asymptotics.**

   (a) $\sim 50n$

   The body of the `i` loop executes $n + (n-1) + (n-2) + \cdots + 0$ times. By the triangle sum, this is $\sim \frac{1}{2}n^2$. The `k` loop executes $\frac{n}{n/100} = 100$ times.

   (b) $O(n^3)$, $O(n\sqrt{n})$, $\Omega(\log n)$

   The order of growth of the function is $\Theta(n\sqrt{n})$, or $\sim 3n\sqrt{n}$ in tilde notation. n^3 is larger and $n\sqrt{n}$ is the same order of growth of the function. (And $\sim n\sqrt{n}$ has a smaller constant.)
   $\log n$ is a smaller order growth than the function.

3. Five sorting algorithms.

   D C E B F

   D. Mergesort just before the left half of the array is sorted.

   C. Insertion sort after 16 iterations.

   E. Quicksort (standard, no shuffle) after first partitioning step.

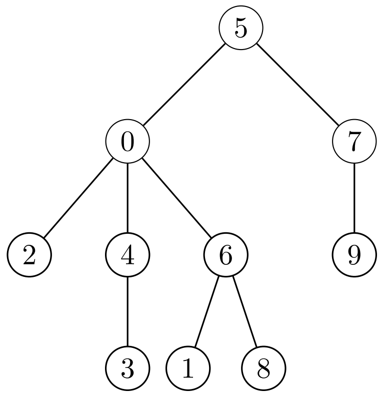
   B. Selection sort after 12 iterations.

   F. Heapsort after heap construction phase and placing the 12 largest keys.

4. **Data structure invariants.**

   (a) Impossible

   The forest defined by the array has a single tree, shown below. Consider the point in time when 0 was connected to its parent 5. At that moment, the subtree rooted at 0 contained 7 elements $(1, 2, 3, 4, 6, 8$ and $0)$ and the subtree rooted at 5 contained at most 3 elements (either 5 alone or $5, 7,$ and 9). (In fact, the latter is not possible: 7 and 9 would both become children of 5.) The link-by-size rule would not have merged the larger tree (rooted at 0) into the smaller tree (rooted at 5).



(b) Impossible

A binary heap is a *complete* binary tree: all levels are fully filled except possibly the last, which is populated from left to right. In this case, however, the node with value 65 has no children, while the node with value 35 does.

(c) Impossible

A binary search tree should maintain symmetric order, meaning that keys in nodes in the left subtree must be less than the root, and keys in nodes in the right subtree must be greater. In this case, however, 28 is in the left subtree of the root 25, despite being greater than 25. Additionally, 20 appears in the right subtree of 25, even though it is smaller.

5. **Balanced search trees.**

(a) *B*

Tree B is the only fully balanced tree, meaning that every path from the root to a null link is of equal length. Recall, also, that red links glue nodes in red-black trees, which form a single 3-node in the equivalent 2-3 tree.

(b) color flip 17, rotate left 13, rotate right 20, color flip 17

6. **Queues.**

(a) 1 2 3 3 4 5 4 5 6 7

(b) $\Theta(n)$

`enqueue()` may result in the entire queue being printed.

(c) $\Theta(n)$

Consider a sequence of n `enqueue()` operations. The third insert operation results in 3 elements being printed, the sixth insert operation results in 6 elements being printed, etc. Prints alone make the total running time $\Omega(3+6+9+12+\cdots+n) = \Omega(n^2)$ (factor 3 out and apply the triangle sum), so the amortized time is $\Omega(n)$. Each operation also takes $O(n)$ time (for inserting/removing one and printing at most n elements); therefore, the amortized running time is $\Theta(n)$ (i.e., both $\Omega(n)$ and $O(n)$).

7. **Analysis of algorithms and sorting.**

   (a) $\sim 8n^2$

   Selection sort always makes $\sim \frac{1}{2}m^2$ compares on an array of length m. Here, $m = 4n$, so $\frac{1}{2}m^2 = \frac{1}{2}(4n)^2 = 8n^2$.

   (b) $\sim 4n^2$

   We count the number of exchanges, as for insertion sort the number of compares is at most the number of exchanges plus the size of the array. Each element in the second half of the array is exchanged with all the elements in the first half of the array. Thus, we get $\sim (2n \cdot 2n) \sim 4n^2$ exchanges.

   (c) $\sim 2n \log_2 n$

   The topmost `merge()` merges two subarrays of length $2n$ and therefore makes $O(n)$ compares. The subarrays are both of length $2n$ and are already sorted, therefore each level of the recursion tree (below the root) makes $\sim \frac{1}{2}\big(2n \log(2n)\big) \sim n \log n$ compares. Therefore, the total number of compares is $2n \log n$ ($O(n)$ is a low order term).

   (d) $\Theta(n)$

   The top-most 3-way partition uses $2n$ as a pivot on an array of length $4n$, and thus makes $\sim 4n$ compares. After partitioning, the array is as follows:

$$n \; n \; \ldots \; n \; \ldots \; 8\;8\;8\;8\;8\;8\;8\;8\;4\;4\;4\;4\;2\;2\;1\;0\;2n\;2n\;\ldots\;2n.$$

   Only the left half remains for further recursive sorting, as the pivots are already in their final sorted positions.

   The next execution of 3-way partition uses n as a pivot on an array of length $2n$, and thus makes $\sim 2n$ compares. After partitioning, the (remaining) array is as follows:

$$\frac{n}{2}\;\frac{n}{2}\;\ldots\;\frac{n}{2}\;\ldots\;8\;8\;8\;8\;8\;8\;8\;8\;4\;4\;4\;4\;2\;2\;1\;0\;n\;n\;\ldots\;n.$$

   Only the left half remains for further recursive sorting, as the pivots are already in their final sorted positions.

   This reasoning extends for all recursive calls, so the number of compares is

$$\sim \left(4n + 2n + n + \frac{1}{2}n + \frac{1}{4}n + \cdots + 1\right) \sim 8n = \Theta(n).$$

   (Notice that the number of keys being $\Theta(\log n)$ implies the runtime is $O(n \log n)$: the number of calls is $\Theta(\log n)$ and each partition makes $O(n)$ compares. But this is only an upper bound.)

8. **Algorithm design.**

   (a) Algorithm:

      i. Reverse the second half of the array so its order becomes ascending. (This can be done in-place or with an auxiliary, array since there are no memory constraints.)

      ii. With both halves now sorted in ascending order, apply the `merge()` method on the two halves to get a sorted array.

An alternative algorithm is obtained by modifying the `merge()` method to scan the first half of the array left-to-right (starting at $i = 0$ and incrementing `i`) and the second half right-to-left (starting at $j = n - 1$ and decrementing `j`).

(b) No.

Assuming an $O(n)$ time implementation of `makeMountainLike()`, we could create a compare-based sorting algorithm that runs in $O(n)$ time, contradicting the $\Omega(n \log n)$ *sorting lower bound*. To do this, first transform the input array into a mountain-like form with `makeMountainLike()`, which we assume runs in $O(n)$ time, and thus makes $O(n)$ compares. Additionally, since the input of `makeMountainLike()` is a `Comparable[]` array, it is itself compare-based (i.e., only uses the `compareTo()` method to compare elements). Next, sort this mountain-like array using the algorithm in part (a), which also makes only $O(n)$ compares. This yields a total of $O(n)$ compares for sorting an arbitrary array.

9. **Data structure design.**

We maintain a red-black tree called `players` where keys are player names, and values are their scores; to support `getLead()`, we can maintain a `String` and an `int` variable that store the maximum score with the respective player (and return the string). Alternatively, we can use another data structure `scores`, where keys are scores and values are player names. `players` can be implemented as a red-black tree and `scores` can be a red-black tree, or a max-heap.

To implement `insert(String name, int score)`, we add `(name, score)` to `players` and update the score and name if the new player has larger score than the maximum. (If using a second data structure `scores`, and `(score, name)` to it.)

For `getScore(String name)`, we search for the key `name` in `players`. For `getLead()`, we retrieve the maximum from `scores`, which takes $\Theta(1)$ time if it is a max-heap or $O(\log n)$ time if it is a red-black tree.

Observe that since we did not require an implementation for the `delete` method, the two data structures do become out of sync, so there is no need to connect them with pointers.