CSE 373 23sp Practice Midterm Solutions

Question 1:

Here are the runtime comparisons for each method:

- `pushSong`
  - An array implementation would take O(n) time to shift values, but a linked list would take O(1) time. We also don't have to resize a linked list.
- `appendSong`
  - An array would take O(1) amortized* time but a linked list would take O(n) time because we don't have a back pointer.
- `popSong`
  - Linked list would take O(1) time but an array would take O(n) time for the same reasons as `pushSong`.
- `removeSong`
  - Both linked list and array would take O(n) time.
- `shuffle`
  - Array would take O(1) time, but linked list would take O(n) time because we have to iterate through the songs.
- `Play`
  - Similar comparison as shuffle.

Overall, we see that a linked list can efficiently push and pop songs, but for all other operations an array would be at least as fast. If we want a data structure where we can play particular songs or pick random ones frequently, an array would be the best choice.

*amortized means that, when we factor in resizing, the runtime of appending a song would be constant if we averaged it across many appends

Now let’s consider the same operations for a circular array:

- `pushSong`
  - A circular array would take O(1) time because we can just decrement the front pointer. We still have to worry about resizing, but the in-practice/amortized runtime is still O(1).
- `appendSong`
  - A circular array would take O(1) time in practice for reasons similar to `pushSong.`
- `popSong`
  - A circular array would take O(1) time in-practice for similar reasons as above.
- `removeSong`
  - Circular array would take O(n) time because we would have to shift values.
- `shuffle`
  - Circular array would take O(1) time - just pick a random index from front to back. You may have to do some extra math if the front pointer was behind the back pointer, but it would still be constant.
- `Play`
  - Circular array would take O(1) time.

Question 2:

What is the worst-case time complexity of `insert()`?

> The runtime of insert is O(1). This is because we traverse down the trie one character at a time, and each step takes a constant time operation for creating a new node or retrieving an existing one from the children map, with a maximum possibility of 100 steps (digits). Since 100 is a constant factor in relation to n, we can say the runtime analysis is O(1).

What is the worst-case time complexity of `search()`?

> The runtime of search is O(1). This is because we traverse down the trie one character at a time, and each step takes a constant time operation for checking if the current character exists in the children map, with a maximum possibility of 100 steps (digits). Since 100 is a constant factor in relation to n, we can say the runtime analysis is O(1).

What is the worst-case time complexity of `findMatchingPrefix()`?

> The runtime of findMatchingPrefix is O(n). This is because in our worst case scenario (if the prefix was an empty string), we would hypothetically have to add all of the distinct phone numbers in our trie to our list of elements, requiring some amount of constant work at each element and for us to iterate through the entire data structure. In our worst case scenario, all of the strings in our trie are at maximum length and share no nodes, meaning we have to travel down 100 nodes *n* times, giving us a runtime of O(100 * n) which can simplify down to O(n).

Which of the two solutions would be the best implementation of delete for Simon’s current situation? Be sure to describe the tradeoffs between the two implementations!

> Both implementations have a worst-case runtime of O(1), since we would need to find the node to delete either way, requiring us to traverse down up to 100 nodes in the trie. Once we get to that node, the first solution simply marks it as “false,” meaning that though the node is functionally “irrelevant” data, it and all preceding non-storage nodes “end” nodes would still persist and take up space. Method 2, on the other hand, would “clean up” all non-useful nodes and actually remove them from memory, which might potentially require us to delete the entire chain of nodes we traversed down if no nodes in our path are shared with a different phone number. In this case, we would be traveling through 100 * 2 nodes (since go to the node and back to the root), which is still a constant factor in relation to n. Therefore, for Simon’s scenario, option 2 would be the best choice because we are able to optimize the limited amount of space usage on Simon’s phone.