CSE 373: 23 Spring PRACTICE Midterm

Name:	UW Email:	@uw.edu

Instructions

- The allotted time is **50** minutes. Please do not turn the page until staff says to do so
- This is an open-book exam. You are NOT permitted to access electronic devices including calculators.
- Read the directions carefully, especially for problems that require you to show work or provide an explanation.
- We can only give partial credit for work that you've written down.
- Unless otherwise noted, every time we ask for an O, Ω, or Θ bound, it must be simplified and tight.
- If you run out of room on a page, indicate where the answer continues. Try to avoid writing on the very edges of the pages: we scan your exams and edges often get cropped off.

Advice

- If you feel like you're stuck on a problem, you may want to skip it and come back at the end if you have time.
- Relax and take a few deep breaths. You've got this :-)

Question	**Max Points**
1. ADT Design	50
2. Code Analysis	50
Total	**100**

Resubmission Details

- This exam will be graded out of 100 points. If you are not satisfied with your grade, you will be given the opportunity to resubmit it online and earn up to 50% of the missed points back.
- For example, a student scoring 80/100 points may receive up to 90/100 points on the resubmission.

Question 1:

Yafqa is tasked with designing a program that represents a music playlist, such as for Spotify. The playlist should have the following functionality:

- pushSong(trackName):
  - Add a new song to the beginning of the playlist.
- appendSong(trackName):
  - Add a new song to the end of the playlist
- popSong(trackName):
  - Remove a song from the playlist
- removeSong(n):
  - Remove the n^{th} song from the playlist.
- shuffle():
  - Play a random song from the playlist
- play(n):
  - Play the n^{th} song in the playlist

Yafqa has two potential data structures to build this program with:
- A non-circular arraylist
- A singly-linked list with a front pointer.

Help Yafqa decide on which solution is best for a given scenario! For each data structure, explain the differences between both implementations and the associated tradeoffs. Be sure to include in-practice runtime analysis for each method. Limit your answers to 3-5 sentences per data structure.

Yafqa's kind TA friend Sravani recommended that he can optimize his playlist by using a circular array. How might the in-practice runtimes for each method change with this new implementation?

Question 2:

Suppose Simon is implementing a phone book using a trie data structure. Each node in the trie represents a prefix of a phone number, and the leaves of the trie store the full phone numbers and associated names. Each phone number is a string of digits (0-9), and each name is a string of characters. Suppose each phone number is a maximum of 100 digits long (of varying lengths), and suppose there are ***n*** phone numbers overall. Here is his code so far:

```
class TrieNode {
    Map<Character, TrieNode> children;
    boolean isEnd;
    String phoneNumber;
    String name;

    public TrieNode() {
        this.children = new HashMap<>();
        this.isEnd = false;
        this.phoneNumber = null;
        this.name = null;
    }
}

public class PhoneBook {
    private TrieNode root;

    public PhoneBook() {
        this.root = new TrieNode();
    }

    public void insert(String phoneNumber, String name) {
        TrieNode node = root;
        // converts a given string to an array of chars
        for (char c : phoneNumber.toCharArray()) {
            if (!node.children.containsKey(c)) {
                node.children.put(c, new TrieNode());
            }
            node = node.children.get(c);
        }
        node.isEnd = true;
        node.phoneNumber = phoneNumber;
        node.name = name;
    }
```

```
   public String search(String phoneNumber) {
          TrieNode node = searchNode(phoneNumber);
          if (node != null && node.isEnd) {
              return node.name;
          } else {
              return null;
          }
    }

    private TrieNode searchNode(String phoneNumber) {
        TrieNode node = root;
        for (char c : phoneNumber.toCharArray()) {
            // we can assume that get runs in constant time
            if (node.children.get(c) == null) {
                return null;
            }
            node = node.children.get(c);
        }
        return node;
    }

    public List<String> findMatchingPrefix(String prefix) {
        List<String> result = new ArrayList<>();
        TrieNode node = searchNode(prefix);
        if (node != null) {
            findMatchingPrefixHelper(node, result);
        }
        return result;
    }

    private void findMatchingPrefixHelper(TrieNode node, List<String> result) {
        if (node.isEnd) {
            result.add(node.phoneNumber);
        }
        for (char c : node.children.keySet()) {
            findMatchingPrefixHelper(node.children.get(c), result);
        }
    }
}
```

Answer the following questions in terms of *n*:

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

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

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

Suppose Simon has a full phonebook storing various numbers of his many many friends, but suddenly had a falling out with all of his existing contacts, and wants to remove most of the numbers from his phonebook. Additionally, he has too many games on his phone and is running out of storage!

He is considering two implementation styles to “delete” numbers from his phonebook.

Solution 1:

```
public boolean delete(String phoneNumber) {
    TrieNode node = searchNode(phoneNumber);
    if (node != null && node.isEnd) {
        node.isEnd = false; // mark node as deleted
        return true;
    } else {
        return false;
    }
}
```

Solution 2:

```
public boolean delete(String phoneNumber) {
    TrieNode node = searchNode(phoneNumber);
    if (node != null && node.isEnd) {
        deleteHelper(root, phoneNumber, 0);
        return true;
    } else {
        return false;
    }
}

private boolean deleteHelper(TrieNode node, String phoneNumber, int depth) {
    if (depth == phoneNumber.length()) {
        node.isEnd = false;
        return node.children.isEmpty(); // return true if node has no children
    }
    char c = phoneNumber.charAt(depth);
    TrieNode child = node.children.get(c);
    if (child == null) {
        return false; // node not found
    }
    boolean shouldDelete = deleteHelper(child, phoneNumber, depth + 1);
    if (shouldDelete) {
        node.children.remove(c);
        return node.children.isEmpty();
    }
    return false;
}
```

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, as well as analyzing runtime complexity!