CSE 373: 24 Wi Makeup Midterm

Name:	UW Email:	@uw.edu

Instructions

- The allotted time is **50** minutes. Please do not turn the page until the staff says to do so.
- This is an open-book and open-notes 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, when we ask for algorithm runtime, it must be **simplified and tight**.
- **You may assume that all hash functions and operations (find, add, remove) are O(1).**
- 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, skip it and come back at the end if you have time.
- Relax and take a few deep breaths. You got this :-)

Questions
1. ADT Design
2. Code Analysis

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.

Problem 1 (ADT Design):

Yafqa cooks a lot, but is always running out of spices! To keep better track of how much of each spice he still has left, he wants to design a program to represent his spice cabinet.

Each spice will be identified by its name (a String), and will be associated with a percentage, which is a double between 0.0 and 100.0 representing the amount of the spice still remaining.

The program should have the following functionality:

- `void stockSpice(String spiceName)`
    - adds a new spice to the cabinet, or restocks an existing spice, setting the percentage to 100.0
- `int getStock(String spiceName)`
    - returns the percentage of the spice left
    - throws `NoSuchElementException` if the spice is not contained in the cabinet
- `void useSpice(String spiceName, double percentage)`
    - consumes the spice specified, reducing the supply by the given percentage
    - throws `NoSuchElementException` if the spice is not contained in the cabinet
    - throws `IllegalArgumentException` if the given `percentage` exceeds the current amount of the spice left
- `List<String> needRestock()`
    - returns a list of all the spices that have less than 10.0 percent supply remaining

Jaylyn, Parker, and Joshua have each come up with a solution to this problem. For each of their solutions, you will be asked to provide the **tight Big-Oh** runtime of the specified methods. Use **n** to denote the number of spices in the spice cabinet.

Assume Yafqa is always careful and never has more than 5 spices that need to be restocked. Pay CAREFUL attention to what the KEYS are in each solution.

Jaylyn's Solution:

We can store `(spiceName, percentage)` pairs in a hash table, where the keys are the **spice names**.

1) What is the **in-practice**, tight Big-Oh runtime of `stockSpice`?

2) What is the **in-practice**, tight Big-Oh runtime of `needRestock`?

Parker's Solution:

We can store `(percentage, spiceName)` pairs in an ArrayList that is sorted by the **percentage**.

3) What is the **worst case**, tight Big-Oh runtime of `useSpice`?

4) What is the **worst case**, tight Big-Oh runtime of `needRestock`?

Joshua's Solution: We can store the data in an AVL Tree, where the keys are **percentages** and the values are `ArrayList`s of spice name(s) that are currently stocked at that percentage.

5) What is the **worst case**, tight Big-Oh runtime of `getStock`?

6) What is the **worst case**, tight Big-Oh runtime of `needRestock`?

Yafqa buys his spices in bulk, and doesn't need to restock very often. His top priority is to quickly use his multitude of spices while cooking. In other words, he wants his `useSpice` operation to be as fast as possible.

7) Which of Jaylyn's, Parker's, or Joshua's solutions would be most optimized for this scenario, and why? Explain your answer in at most 2 sentences.

Yafqa is very satisfied with his spice cabinet, until one day, he notices that Simon also has a spice cabinet, but Simon's spice cabinet can perform all of its operations in O(1) time! (Note: Simon is also careful and never has more than 5 spices low on stock.)

Yafqa thinks this can be achieved by using a combination of multiple data structures. Below, describe how you might achieve an **in-practice O(1) runtime** on **all operations** (`stockSpice`, `getStock`, `useSpice`, and `needRestock`).

8) What data structures would you use, and what would they represent? Explain in at most 3 sentences.

9) Describe your implementation of `stockSpice`, `getStock`, and `useSpice` (either plain English or pseudocode). If the three methods have similar implementations, you can describe them together. If not, describe the implementations individually. Explain in at most 6 sentences.

10) Describe your implementation of `needRestock` (either plain English or pseudocode). Explain in at most 4 sentences.

Problem 2 (Code Analysis):

Rachel has too many shoes, and can't fit them all by her front door. Rachel wants a shoe rack by her front door to store the shoes she wears most.

Rachel wants to design an object called `ShoeRack`. A `ShoeRack` has the following functionalities:

- `ShoeRack(int n)`
  - constructs an empty `ShoeRack` object that can fit at most `n` shoes
- `boolean wearShoe(String shoe)`
  - if `shoe` is on the `ShoeRack`, mark this interaction and returns `true`
    - `shoe` remains on the `ShoeRack` after it has been "worn"
  - if `shoe` is not on the `ShoeRack`, returns `false`
- `String addShoe(String shoe)`
  - adds the specified `shoe` to the `ShoeRack`
  - if `n` shoes are already on the rack, the new shoe will replace the shoe with the longest time since interaction. Interaction is triggered either by calling `wearShoe` or `addShoe`.
  - returns the shoe that was replaced, or `null` if no shoe was replaced

Here's an example of how Rachel would use the ShoeRack object:

```
// shoe rack has a capacity of 3 shoes
ShoeRack rack = new ShoeRack(3);
rack.addShoe("Doc Martens");                           // returns null
rack.addShoe("Flip Flops");                            // returns null
rack.addShoe("Air Force 1");                           // returns null

// the rack now is completely full,
// with 3 shoes on it
rack.wearShoe("Doc Martens");                          // returns true

// the shoe that was interacted with
// longest ago is "Flip Flops"
rack.addShoe("Hiking Boots");                          // returns "Flip Flops"

// "Hiking Boots" have now replaced "Flip Flops"
// rack now contains "Hiking Boots",
// "Doc Martens", and "Air Force 1"
rack.wearShoe("Flip Flops");                          // returns false
```

Rachel thinks of two implementations of the `ShoeRack` object.
Below is the first solution. Pay attention to comments.

```
public class ShoeRack {
    private int n;
    private int timestamp;          // incremented each time a shoe is added or worn
    private Map<String, Integer> shoeToTimestamp;
    private TreeMap<Integer, String> timestampToShoe;

    public ShoeRack(int n) {
        if (n < 1) {
            throw new IllegalArgumentException("n must be at least 1");
        }
        this.n = n;
        timestamp = 1;
        shoeToTimestamp = new HashMap<>();  // constructs a HashMap
        timestampToShoe = new TreeMap<>();  // this is a self-balancing BST
    }

    public boolean wearShoe(String shoe) {
        if (!shoeToTimestamp.containsKey(shoe)) {
            return false;
        }

        int oldTimestamp = shoeToTimestamp.get(shoe);
        timestampToShoe.remove(oldTimestamp);
        timestampToShoe.put(timestamp, shoe);
        shoeToTimestamp.put(shoe, timestamp);

        timestamp++;
        return true;
    }

    public String addShoe(String shoe) {
        String replacedShoe = null;
        if (shoeToTimestamp.size() == n) {
            // gets the entry with the lowest timestamp
            // firstEntry() finds the smallest key in the BST
            Map.Entry<Integer, String> replacedEntry = timestampToShoe.firstEntry();
            int oldTimestamp = replacedEntry.getKey();
            replacedShoe = replacedEntry.getValue();

            timestampToShoe.remove(oldTimestamp);
            shoeToTimestamp.remove(replacedShoe);
        }

        timestampToShoe.put(timestamp, shoe);
        shoeToTimestamp.put(shoe, timestamp);

        timestamp++;
        return replacedShoe;
    }
}
```

Analyze the asymptotic runtime, as a function of `n`, of the methods in **first** implementation. If there is no distinction between best and worst cases, provide the same answer to both cases.

Recall that `n` is the **capacity** of the `ShoeRack`, and not the number of shoes contained in it.

1) What is the **best-case**, simplified, tight big-oh runtime of `wearShoe`?

2) What is the **worst-case**, simplified, tight big-oh runtime of `wearShoe`?

3) What is the **best-case**, simplified, tight big-oh runtime of `addShoe`?

4) What is the **worst-case**, simplified, tight big-oh runtime of `addShoe`?

Below is Rachel's second solution. Pay attention to comments.

```java
public class ShoeRack {
    private int n;

    // front is most recent shoe, back is least recent shoe
    private Deque<String> shoeDeque;
    private Set<String> shoes;

    public ShoeRack(int n) {
        if (n < 1) {
            throw new IllegalArgumentException("n must be at least 1");
        }
        this.n = n;

        // this is a doubly-linked list implementation from project 1
        shoeDeque = new LinkedDeque<>();
        shoes = new HashSet<>();
    }

    public boolean wearShoe(String shoe) {
        if (!shoes.contains(shoe)) {
            return false;
        }

        //shoe guaranteed to be in deque
        Deque<String> temp = new LinkedDeque<>();
        String currShoe = shoeDeque.removeFirst();
        while (!currShoe.equals(shoe)) {
            temp.addFirst(currShoe);
            currShoe = shoeDeque.removeFirst();
        }
        while (!temp.isEmpty()) {
            shoeDeque.addFirst(temp.removeFirst());
        }
        shoeDeque.addFirst(currShoe);
        return true;
    }


    public String addShoe(String shoe) {
        String replacedShoe = null;
        if (shoes.size() == n) {
            replacedShoe = shoeDeque.removeLast();
        }
        shoeDeque.addFirst(shoe);
        shoes.remove(replacedShoe);
        shoes.add(shoe);
        return replacedShoe;
    }
}
```

Analyze the asymptotic runtime, as a function of `n`, of the methods in **second** implementation.

Recall that `n` is the **capacity** of the `ShoeRack`, and not the number of shoes contained in it.

5) What is the **worst-case**, simplified, tight big-oh runtime of `wearShoe`?

6) What is the **worst-case**, simplified, tight big-oh runtime of `addShoe`?

Rachel realizes she might be able to adjust her **second** `ShoeRack` implementation by directly accessing the list nodes in `shoeDeque`. She modifies her implementation by changing the field `shoes` from a `HashSet<String>` of shoes to a `HashMap<String, Node<String>>` of shoes mapped to their respective nodes in `shoeDeque`. This way, whenever she has to update the "recency of interaction" of a particular shoe, she can locate the node corresponding to the shoe through her hashmap, and update node links accordingly.

7) Overall, is the asymptotic runtime of Rachel's newest implementation better, worse, or neither, compared to her **second** implementation? Assume that Rachel has public access to the fields in the `LinkedDeque`. Also assume Rachel uses her modified data structures in the most efficient manner possible. Justify your answer in at most 2 sentences.