CSE 373 18wi: Practice Midterm Solutions

Name:

UW email address:

Instructions

- **Do not start the exam until told to do so**.
- You have **80** minutes to complete the exam.
- This exam is closed book and closed notes.
- You may not use a cell phone, a calculator, or any other electronic devices.
- Write your answers neatly in the provided space. Be sure to leave some room in the margins: we will be scanning your answers.
- If you need extra space, use the back of the page.
- If you have a question, raise your hand to ask the course staff for clarification.

Question	Max points	Earned
Question 1	??	
Question 2	??	
Question 3	??	
Question 4	??	
Question 5	??	
Question 6	??	
Question 7	??	
Question 8	??	
Question 9	??	
Total	**??**	

Additional notes about this practice exam

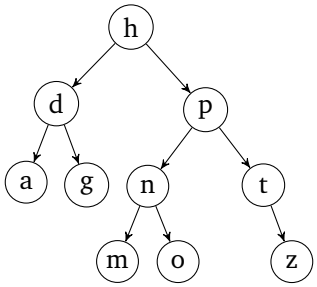
- This practice midterm is structured in roughly the same way the actual midterm is.
- These questions are either roughly about the same difficulty or are slightly harder then the questions you will be asked on your midterm.
- Since this is a new practice exam, our solutions may contain a few mistakes or typos. Please ask on Piazza if you see something that does not make sense.

1. AVL rotations

Insert the following sequence of values into an AVL tree *in the given order*.

a, d, h, m, p, a, g, n, t, m, o, z

Solution:



2. Hash tables

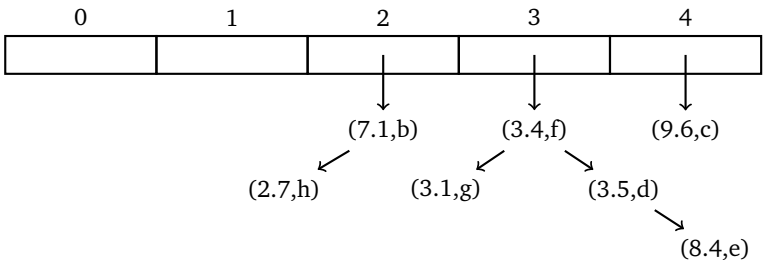
Consider the following sequence of key-value pairs:

(3.4, a), (7.1, b), (9.6, c), (3.5, d), (8.4, e), (3.4, f), (3.1, g), (2.7, h)

Suppose that these floats are implemented such that they use the hash function "$h(k) = \text{roundDown}(k)$". For example, the key 3.4 would have a hash code of 3; the key 9.9 would have a hash code of 9.

(a) Insert the above sequence of key-value pairs *in the given order* into a hash table with an internal array of size 5 using *separate chaining*. Assume each bucket is implemented using an *binary search tree*. Do not worry about resizing the internal array.

Solution:



(b) Insert the same pairs into an a hash table with capacity 10 that uses *quadratic probing*. Again, do not worry about resizing the internal array.

Solution:

0	1	2	3	4	5	6	7	8	9
		(3.1,g)	(3.4,f)	(3.5,d)		(2.7,h)	(7.1,b)	(8.4,e)	(9.6,c)

3. Asymptotic analysis

(a) Consider the following two functions:

$$f(n) = \frac{1}{10}n^2 \qquad g(n) = \begin{cases} n^5 & \text{if } n \leq 5 \\ 99n + \log(n) & \text{otherwise} \end{cases}$$

Show that $f(n) \in \Omega(g(n))$ is true by finding a c and n_0 that satisfies the definition of "dominates" and big-Ω. Please show your work.

Solution:

First, note that $g(n) = 99n + \log(n)$ when $n > 5$. So, to simplify our analysis, we will show that $f(n) \in \Omega(g(n))$ specifically for $n \geq 6$.

Next, to show that the definition of big-Ω holds, we must find some c and n_0 such that $\frac{1}{10}n^2 \geq c(99n + \log(n))$ is true for all $n \geq n_0$.

In order to do so, we first observe that the following chain of inequalities are true:

$$\begin{aligned} c(99n + \log(n)) &\leq c(99n + n) && \text{for } n \geq 1 \\ c(99n + n) &\leq c100n \\ c100n &\leq c100n^2 && \text{for } n \geq 1 \end{aligned}$$

Therefore, we conclude that $c100n^2 \geq c(99n + \log(n))$ holds for all $n \geq 1$.

Next, observe that $\frac{1}{10}n^2 \geq c100n^2$ is true when $c = \frac{1}{1000}$ and for all values of n.

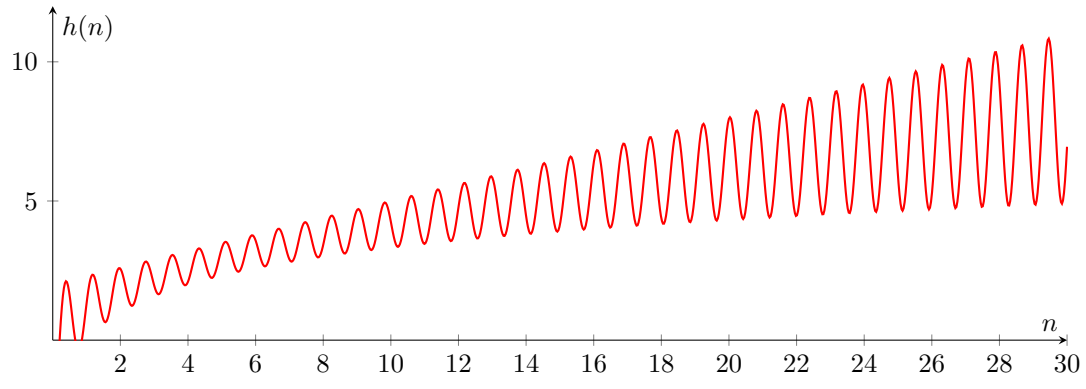
Therefore, if we apply the inequality we discovered previously, it must be the case that $\frac{1}{10}n^2 \geq c(99n + \log(n))$ is true for the same value of c.

We previously declared $g(n) = 99n + \log(n)$ only when $n \geq 6$. Therefore, we conclude $f(n) \geq cg(n)$ for $c = \frac{1}{1000}$ and $n_0 = 6$.

(b) Suppose we discovered a function $h(n)$ and discovered that the tightest possible upper bound is $h(n) \in \mathcal{O}(n)$ and the tightest possible lower bound is $h(n) \in \Omega(\log(n))$. Draw a plot of what this function might look like.

Solution:

There are several solutions; one possible one is:



4. Eyeballing Big-Θ bounds

For each of the following snippets of code, please give a big-Θ bound of the worst-case runtime with respect to n. You do not need to justify your answer.

(a)

```
public void partA(int n) {
    for (int i = 0; i < n * n; i++) {
        if (i % 2 == 0) {
            for (int j = 0; j < i; j++) {
                System.out.println("?");
            }
        }
    }
}
```

Solution:

$\Theta\left(n^4\right)$

(b)

```
public void partB(int n) {
    // An AvlDictionary is a dictionary internally implemented
    // using an AVL tree.
    IDictionary<Integer, Integer> dict = new AvlDictionary<>();
    for (int i = 0; i < n; i++) {
        if (i < 100000) {
            for (int j = 0; j < i; j++) {
                dict.put(j * i, i);
            }
        } else {
            dict.put(i, i);
        }
    }
}
```

Solution:

$\Theta\left(n \log(n)\right)$

(c)

```
public void partC(int n) {
    IDictionary<Integer, Integer> dict = new AvlDictionary<>();
    for (int i = 0; i < n; i++) {
        dict.put(4, i);
        for (int j = 0; j < dict.size(); j++) {
            System.out.println(dict.containsKey(j));
        }
    }
}
```

Solution:

$\Theta\left(n\right)$

(d)

```
public void partD(int n) {
    if (n == 0) {
        System.out.println("...");
    } else {
        System.out.println("...");
        partD(n - 1)
    }
}
```

Solution:

$\Theta\left(n\right)$

5. Modeling code

Consider the following Java program. Let n represent the value of the input parameter `n` and let m represent the value of the parameter `m`.

```
public static int mystery(int n, int m) {
    if (n >= 40) {
        for (int i = 0; i < n * m; i++) {
            if (i % m == 0) {
                System.out.println("...");
            }
        }
        return -2 * mystery(n - 3, m / 3) + 3 * mystery(n - 5, m + 3);
    } else {
        return m * 2;
    }
}
```

In the following questions, you will be asked to construct several mathematical functions modeling different aspects of the `mystery` method. Your answers to all three questions should be a recurrence. Your recurrence may include summations, if you want. You do *NOT* need to find a closed form to your models.

(a) Construct a mathematical function $T(n, m)$ that represents the *approximate worst-case runtime* of `mystery`.

Solution:

$$T(n,m) = \begin{cases} 1 & \text{If } n < 40 \\ nm + T\left(n-3, \frac{m}{3}\right) + T\left(n-5, m+3\right) & \text{Otherwise} \end{cases}$$

(b) Construct a mathematical function $P(n, m)$ that represents the *total number of lines printed out* by `mystery`.

Solution:

$$P(n,m) = \begin{cases} 0 & \text{If } n < 40 \\ n + P\left(n-3, \frac{m}{3}\right) + P\left(n-5, m+3\right) & \text{Otherwise} \end{cases}$$

(c) Construct a mathematical function $F(n, m)$ that represents the *exact integer output* of `mystery`. That is, it should be the case that $F(n, m)$ `== mystery(n, m)`.

Solution:

$$F(n,m) = \begin{cases} 2m & \text{If } n < 40 \\ -2F\left(n-3, \frac{m}{3}\right) + 3F\left(n-5, m+3\right) & \text{Otherwise} \end{cases}$$

6. Systems and B-Trees

(a) Consider the following code:

```
public static int sum(IList<Integer> list) {
    int output = 0;
    for (int i = 0; i < 128; i++) {
        // Reminder: foreach loops in Java use the iterator behind-the-scenes
        for (int item : list) {
            output += item;
        }
    }
    return output;
}
```

You try running this method twice: the first time, you pass in an array list, and the second time you pass in a linked list. Both lists are of the same length and contain the exact same values.

You discover that calling `sum` on the array list is consistently 4 to 5 times faster then calling it on the linked list. Why do you suppose that is?

Solution:

This is most likely due to spatial locality. When we iterate through a linked list, accessing the value at one particular index will load the next few elements into the cache, speeding up the overall time needed to access each element.

In contrast, each node in the linked list is likely loaded in a random part of memory – this means we likely must load each node into the cache, which slows down the overall runtime by some constant factor.

(b) Suppose you are trying to use a B-Tree somebody else wrote for your system. You know the following facts:

- $M = 10$ and $L = 12$
- The size of each pointer is 16 bytes
- The size of each key is 14 bytes
- The size of each value is 11 bytes

Assuming M and L were chosen wisely, what is most likely the page size on this system?

Solution:

If L is 12, and each key-value pair occupies $14 + 11 = 25$ bytes, we know each leaf node occupies at least $12 \times 25 = 300$ bytes.

If M is 10, we know each each branch node occupies at least $M \times 16 + (M - 1) \times 14 = 160 + 126 = 286$ bytes.

This leads us to conclude that the page size is most likely 300 bytes on this system.

7. Short answer

This section has questions that require very short answers. For full credit, write at most two to three sentences per each question.

(a) Suppose you were trying to implement edit/undo functionality in an image editing program. We want to implement this by keeping track of each operation the user makes. Which ADT would be the most appropriate way of storing these operations: a stack, a queue, or a list? Pick one, and briefly justify.

Solution:

Most likely a stack – we want the ability to undo/redo operations, which corresponds to the pop/push operations on a stack.

(b) What is the worst-case runtime to append a value to the end of a singly-linked list?

Solution:

$\Theta(n)$

(c) True or false: If $f(n) \in \mathcal{O}(g(n))$ is true, then $g(n) \in \mathcal{O}(f(n))$ is also always true. Briefly justify.

Solution:

False: consider $f(n) = n$ and $g(n) = n^2$.

(d) True or false: "$f(n) \in \mathcal{O}\left(n^3\right)$" means the exact same thing as "$f(n)$ has a worst-case runtime of n^3". Briefly justify.

Solution:

False: we don't know what the function $f(n)$ is supposed to represent. If $f(n)$ represents the weight of some iron sphere of radius n, for example, saying that $f(n)$ has a "worst-case runtime" would make no sense.

(e) Suppose you need a dictionary where you can traverse over the keys in sorted order. Which data structure should you use?

Solution:

Either a BST or an AVL tree.

(f) True or false: an AVL tree is always more asymptotically efficient then a BST. Briefly justify.

Solution:

False: if the keys are random, both the AVL tree and the BST are likely to be balanced. In that case, both trees are likely to have a runtime of $\Theta(\log(n))$ for their operations.

(g) Suppose you want to implement an efficient dictionary where the iterator always returns key-value pairs in the order the client added them to the dictionary. You know the client will never remove any key-value pairs or

update any previously-added pairs. Briefly describe how you would implement this dictionary by combining two data structures we studied in class. The `put(...)` method should still have an average runtime of $\Theta(1)$.

Solution:

Create hash table with an arraylist field and append to that list whenever the client adds a new key-value pair. The iterator simply iterates through the array list field.

(h) True or false: A hash table's `get(...)` method will always have a runtime of $\Theta(1)$. Briefly justify.

Solution:

False: if the client gives us keys with a bad `hashCode()` function and everything collides on the same cell, we could have a worst-case runtime of $\Theta(n)$.

(i) Which is faster: printing a list of numbers stored in an in-memory array or stored in a text file? Briefly justify.

Solution:

Printing the numbers in the in-memory array would likely be faster: file IO is tends to be an order of magnitude slower then in-memory operations.

8. Debugging

In this problem, we will consider an algorithm named `isBalanced(String str)` that returns "true" if the input string has a "balanced" number of parenthesis and false otherwise. We say a string has "balanced" parenthesis if each opening paren is paired with a matching closing one.

For example, this string is balanced: "((a)b)". This string is also balanced: "(x)(y)(z)".

However, the following two strings are **not** balanced: "((((" and ")))z(".

(a) List at least four distinct kinds of inputs you would try passing into the `isBalanced` algorithm to test it. For each input, also list the expected outcome (assuming the algorithm was implemented correctly). Be sure to think about different edge cases.

Solution:

Some test cases:

(a) Strings with balanced parens.
Examples: "(())", "()()()"
Expected outcome: true

(b) Strings with both letters and parens.
Examples: "((a))", "(a)(b)(c)"
Expected outcome: true

(c) Strings with no parens.
Examples: "foo", "bar"
Expected outcome: true

(d) The empty string.
Example: ""
Expected outcome: true

(e) Strings where there are extra opening parens.
Examples: "(()", "()()("
Expected outcome: false

(f) Strings where there are extra closing parens.
Examples: "())", "()())"
Expected outcome: false

(g) Strings where there are an equal number of opening and closing parens, but they don't match.
Examples: ")))(((", ")()()("
Expected outcome: false

(b) Here is one (buggy) implementation of this algorithm in Java. List every bug you can find.

```
boolean isBalanced(String str) {
    if (str == null || str.size() == 0) {
        return false;
    }
    int numUnmatchedOpenParens = 0;
    for (char c : str) {
        if (c == '(') {
            // Handle opening parens
            numUnmatchedOpenParens += 1;
        } else {
            // Handle closing parens
            numUnmatchedOpenParens -= 1;
        }
    }
    return numUnmatchedOpenParens == 0;
}
```

Solution:

This solution does not correctly handle...

- Strings containing letters that are not opening or closing parenthesis
- Strings with an equal number of opening and closing parens, but are not matched – for example, "))((".
- The empty string.

9. Design

In this problem, you will implement an algorithm named `containsString(String searchTerm, String document)` that returns 'true' if the document contains the search term, and false otherwise.

A naive way of implementing this is to use two nested loops that check if `searchTerm` is equal to a substring of `document`. However, this is inefficient if `searchTerm` is large: comparing two strings of length n takes $\mathcal{O}(n)$ time since we need to check each char one by one.

Your goal is design a faster algorithm by using a kind of hashing algorithm known as a "rolling hash". A rolling hash implements the following methods:

```
public class RollingHash {
    // Instructs the object to keep track of the last 'k' characters eaten.
    public RollingHash(int k) { ... }

    // The number of characters remembered. Returns a number between 0 to k.
    public int size() { ... }

    // Adds the given char to the internal state. If size() > k, forgets the oldest char eaten.
    public void eat(char c) { ... }

    // Returns the hash of the last k characters eaten
    public int getHash() { ... }
}
```

Amazingly, every method in this class has a worst-case runtime of $\mathcal{O}(1)$!

(a) List at least four distinct kinds of inputs you would try passing into your `containsString` algorithm to test it. For each input, also list the expected outcome (assuming the algorithm was implemented correctly). Be sure to think about different edge cases.

Solution:

Some possible test cases:

- If the document does contain the search string, we should return true.
- If the document does not contain the search string, we should return false.
- If the document is shorter then the search string, we should return false.
- If the document only contains substrings that happen to hash to the same value as the search string, but are not identical to the search string, we should return false.

(b) Write an **English description** or **high-level pseudocode** describing an algorithm that implements `containsString`. Note: do **NOT** write Java code.

You may assume `RollingHash` is already implemented for you.

Solution:

One possible solution:

Let s be the searchTerm's size; let d be the document size.

If $d < s$, return false.

Create a new rolling hash and feed it each char in searchTerm. Afterwards, call the `getHash()` method and store that int in a variable named `targetHash`.

Create a second rolling hash, and start feeding it each character in the document one-by-one.

If the rolling hash has a size of s, call its `getHash()` method and see if the result is equal to `targetHash`. If they're not equal, move on to the next character.

If they're equal, take the substring of the last s characters in the document and check to see if they're equal to the search term. If they are, return true. Otherwise, keep going.

If we've finished eating every char in the document and still haven't terminated, return false.

(c) Provide a tight big-Θ bound of the worst-case runtime of your algorithm. Write your answer in terms of s and d, where s is the length of the `searchTerm` string and d is the length of the `document` string.

Briefly justify your answer

Solution:

In the worst possible case, we will have a collision on every single step and will be forced to compare the search term against every single substring in the document.

In that case, our algorithm actually performs no better then the naive one. We initially need to perform s iterations to create `targetHash`. We then perform about d iterations; on each iterations, we do s work to do the substring equality check.

So, the worst-case runtime would be $\Theta(s + ds)$, which simplifies into just $\Theta(ds)$.

(However, if collisions are rare, we would need to perform the substring equality check at most once or twice. In that case, our runtime would be $\Theta(s + d)$ which is much better.)