CSE 123 Summer 2024 Practice Final 2 Answer Key

Name of Student: _____

Section (e.g., AA):_____ Student Number (7 digits):_____

Do not turn the page until you are instructed to do so.

Rules/Guidelines:

- You must not begin working before time begins, and you must stop working **promptly** when time is called. Any modifications to your exam (writing *or* erasing) before time begins or after time is called will result in a penalty.
- You are allowed one page of notes, no larger than 8.5 x 11 inches. You may not access any other resources or use any electronic devices (including calculators, phones, or smart watches, among others) during the exam. Using unauthorized resources or devices will result in a penalty.
- In general, you are limited to Java concepts or syntax covered in class. You may not use `break`, `continue`, a `return` from a `void` method, `try/catch`, or Java 8 stream/functional features.
- You are limited to the standard Java classes and methods listed on the provided reference sheet. You do not need to write import statements.
- If you abandon one answer and write another, ***clearly cross out*** the answer(s) you do not want graded and ***draw a circle or box*** around the answer you do want graded. When in doubt, we will grade the answer that appears in the space indicated, and the first such answer if there is more than one.
- If you require scratch paper, raise your hand and we will bring some to you.
- If you write an answer on scratch paper, please ***write your name and clearly label*** which question you are answering on the scratch paper, and ***clearly indicate*** on the question page that your answer is on scratch paper. Staple all scratch paper you want graded to the ***end*** of the exam before turning in.
- Answers must be written as proper Java code. Pseudocode or comments will not be graded.
- The exam is not graded on code quality. You are not required to include comments.
- You are also allowed to abbreviate `System.out.print` and `System.out.println` as `S.o.p` and `S.o.pln` respectively. You may **NOT** use any other abbreviations.

Grading:

- Each problem will receive a single E/S/N grade.
- Minor syntax errors will be ignored as long as it is unambiguous what was intended (e.g. forgetting a semicolon, misspelling a variable name where there is only one close option). Major syntax errors, or errors where it is unclear what was intended, may have an impact on your grade.

Advice:

- Read all questions carefully. Be sure you understand the question *before* you begin your answer.
- The questions are not necessarily in order of difficulty. Be sure you at least attempt every question.
- Write clearly and legibly. We cannot award credit for answers we cannot read.
- If you have questions, raise your hand to ask. The worst that can happen is we will say "I can't answer that."
- Ask questions as soon as you have them. Do not wait until you have several questions.

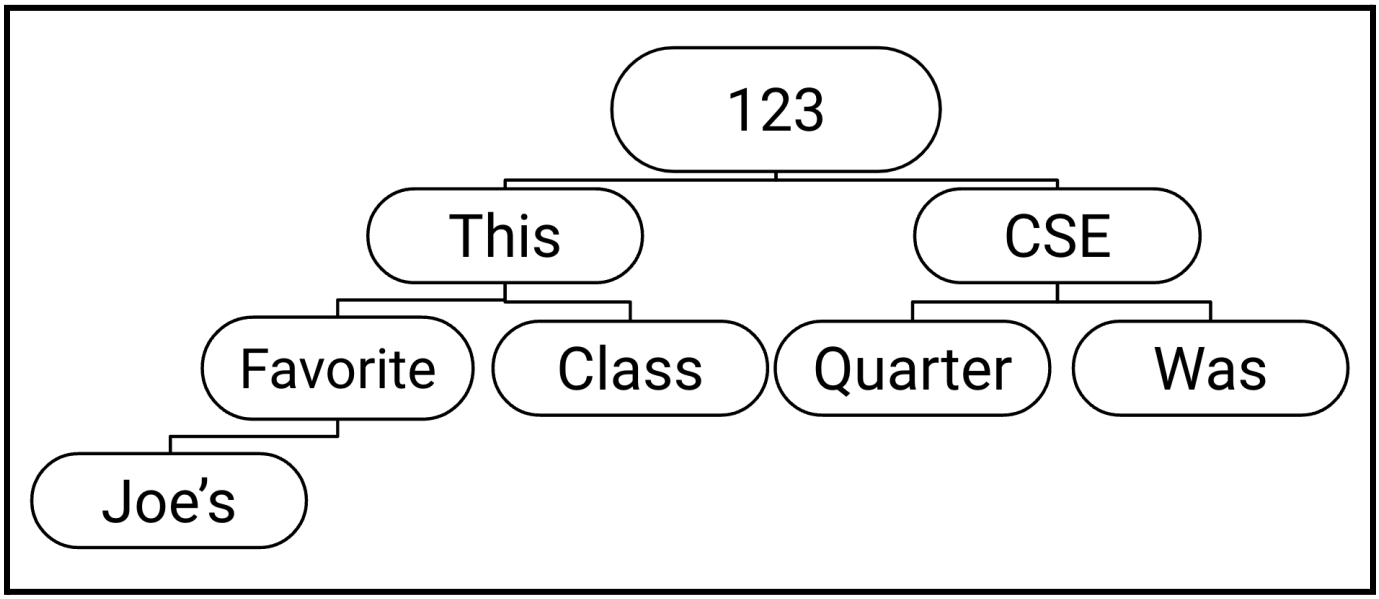
Initial here to indicate you have read and agreed to these rules: 

This page intentionally left blank

Nothing written on this page will be graded

Binary Trees (A)

Zachary has created this binary tree but forgot which traversal method he used to create it! Use this following binary tree to help Zach figure out which traversal method he originally used.



Based on this binary tree, please write the sentence that would be printed if we were to print each word (separated by a space) for each traversal order.

Traversal Order	Method
Example	Fat cats go down alleys eating bologna
Pre-order	123 This Favorite Joe's Class CSE Quarter Was
In-order	Joe's Favorite This Class 123 Quarter CSE Was
Post-order	Joe's Favorite Class This Quarter Was CSE 123

Given the three sentences, which of them makes the most sense? Write down the traversal method and a quick explanation on why the other two don't make sense. Answer in 1-3 sentences.

Post-order.

Pre-order doesn't make sense because:
- Grammatically doesn't make sense

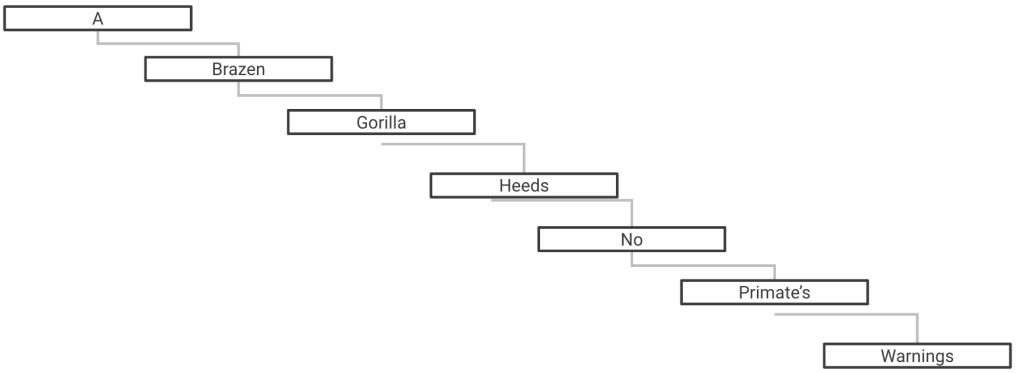
In-Order doesn't make sense because:
- Grammatically doesn't make sense

Binary Tree (continued) (A)

Now two TAs, Andras and Nicole, are discussing whether to use a Binary Search Tree or a Binary Tree for the given set of words:

A, Brazen, Gorilla, Heeds, No, Primate's, Warnings

Note that you do not need to note that this forms a sentence, that is irrelevant. Please draw a binary search tree created by these words starting on the left (with 'A') and then inserting each term one at a time until the last term ('Warnings').



Given this exact set of words, namely "A brazen gorilla heeds no primate's warnings", compare the runtime to a binary tree given the exact same set of words creating the tree in preorder (one node to left one node to right always). Place a check mark in the grid of the faster runtime, otherwise place a checkmark in the "same same" column.

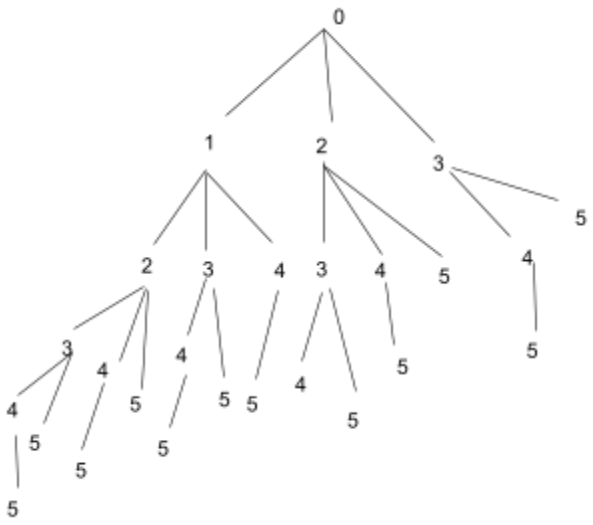
	Binary Search Tree	Eh. Same Same	Binary Tree
Access		x	
Search		x	
Insertion			x
Deletion		x	

Now real quickly before you run out of time, which data structure would you use for this exact set of words? A BST, a Binary Tree, or eh same, same? Write an answer in 2 or less sentences (a couple word answer is fine).

Binary Tree or Same

Exhaustive Search (A)

Due to overexhaustion and not sleeping enough, Trien is having trouble counting to five (5) because Trien can only increase his current count by **1, 2, or 3** each step before he falls asleep again. Draw a decision tree for all possible ways that Trien can count to five (5). A starting node and the levels have been given. Note you may not need all the levels.



Now, Sahej has a similar condition from sleeping at 5 a.m. everyday and wants to program her computer to be able to calculate the amount of possible ways she can count to five (5). **Strictly speaking, for what kind of problems do we need to use exhaustive search and why?** (2-3 sentences)

(Disregard methods such as Dynamic Programming, Memoization, etc. for the sake of the problem)

We use exhaustive search for problems that we must try all possible combinations.
Why?

- Computers have some problems they are bad at
- We must try all possible combinations in these problems (this is less adequate because computer is mentioned)

Sahej chose not to write the code (she already knows she can code it), so Daniel will now code it out. However Daniel is a little confused on where to start. What are two components of performing an exhaustive search and why do we need those two parts? **(This question has two right answers so if you can write both then it's very cool!) These two pairs have special names we are looking for.**

Choice 1: Public/Private Pair

Choice 2: Base/Recursive Case

Linked List (A)

What is the recursive definition of a Binary Tree and how does it differ from the recursive definition of a linked list? (3 sentences max)

Node with a left and right subtree or empty

Differs because for a linked list it is only a node with a linked list following, not a left and right subtree

Write a method called `sumPairs` which sums the pairs of every two numbers in the linkedIntList class. As an example, if we had the list:

(1 -> 2 -> 3 -> 4 -> 5 -> null)

It would sum each pair leading to the finished list:

(3 -> 7 -> 5 -> null)

Another example:

(7 -> 14 -> 14 -> 21 -> 21 -> 28 -> null)

Result of `sumPairs` on list:

(21 -> 35 -> 49 -> null)

Note that if there is an odd number of nodes, the last node will not be affected. If the head of the linkedIntList is null, then the method should throw an `IllegalStateException`. You may only create up to (`n / 2)` extra nodes at most. Your iterative program must run in O(N) time.

On the next page, write the iterative and recursive approach to this problem. There is a field `ListNode front` that stores the LinkedList. **The recursive solution must utilize x = change(x) and you may not modify node data fields.**

Write the method `sumPairs()` iteratively:

```
public void sumPairs() {
    if (front == null) {
        throw new IllegalStateException();
    }
    if (front != null && front.next != null) {
        front = new ListNode(front.data + front.next.data,
                             front.next.next);
    }
    if (front != null) {
        ListNode curr = front;
        while (curr.next != null && curr.next.next != null) {
            curr.next = new ListNode(curr.next.data + curr.next.next.data,
                                     curr.next.next.next);
            curr = curr.next;
        }
    }



}
```

Write the method `sumPairs()` recursively:

```
public void sumPairs() {
    if (front == null) {
        throw new IllegalStateException();
    }
    front = sumPair(front);
}

private ListNode sumPair(ListNode curr) {
    if (curr == null) {
      return null;
    }
    if (curr.next == null) {
      return curr;
    }
    ListNode ret = new ListNode(curr.data + curr.next.data);
    ret.next = sumPair(curr.next.next);
    return ret;
}
```

CSE 123 Quiz/Exam Reference Sheet

(DO NOT WRITE ANY WORK YOU WANTED GRADED ON THIS REFERENCE SHEET. IT WILL NOT BE GRADED)

Methods Found in ALL collections (`List`, `Set`, `Map`)

`clear()`	Removes all elements of the collection
`equals(`**`collection`**`)`	Returns `true` if the given other collection contains the same elements
`isEmpty()`	Returns `true` if the collection has no elements
`size()`	Returns the number of elements in a collection
`toString()`	Returns a string representation such as `"[10, -2, 43]"`

Methods Found in both `List` and `Set` (ArrayList, LinkedList, HashSet, TreeSet)

`add(`**`value`**`)`	Adds value to collection (appends at end of list)
`addAll(`**`collection`**`)`	Adds all the values in the given collection to this one
`contains(`**`value`**`)`	Returns `true` if the given value is found somewhere in this collection
`iterator()`	Returns an Iterator object to traverse the collection's elements
`remove(`**`value`**`)`	Finds and removes the given value from this collection
`removeAll(`**`collection`**`)`	Removes any elements found in the given collection from this one
`retainAll(`**`collection`**`)`	Removes any elements *not* found in the given collection from this one

`List<Type>` Methods

`add(`**`index, value`**`)`	Inserts given value at given index, shifting subsequent values right
`indexOf(`**`value`**`)`	Returns first index where given value is found in list (-1 if not found)
`get(`**`index`**`)`	Returns the value at given index
`lastIndexOf(`**`value`**`)`	Returns last index where given value is found in list (-1 if not found)
`remove(`**`index`**`)`	Removes/returns value at given index, shifting subsequent values left
`set(`**`index, value`**`)`	Replaces value at given index with given value

`Map<KeyType, ValueType>` Methods

`containsKey(`**`key`**`)`	`true` if the map contains a mapping for the given key
`get(`**`key`**`)`	The value mapped to the given key (`null` if none)
`keySet()`	Returns a `Set` of all keys in the map
`put(`**`key, value`**`)`	Adds a mapping from the given key to the given value
`putAll(`**`map`**`)`	Adds all key/value pairs from the given map to this map
`remove(`**`key`**`)`	Removes any existing mapping for the given key
`toString()`	Returns a string such as `"{a=90, d=60, c=70}"`
`values()`	Returns a `Collection` of all values in the map

`Math` Methods

`abs(`**`x`**`)`	Returns the absolute value of `x`
`max(`**`x, y`**`)`	Returns the larger of `x` and `y`
`min(`**`x, y`**`)`	Returns the smaller of `x` and `y`
`pow(`**`x, y`**`)`	Returns the value of `x` to the `y` power
`random()`	Returns a random number between `0.0` and `1.0`
`round(`**`x`**`)`	Returns `x` rounded to the nearest integer

`String` Methods

`charAt(`**i**`)`	Returns the character in this String at a given index
`contains(`**str**`)`	Returns `true` if this String contains the other's characters inside it
`endsWith(`**str**`)`	Returns `true` if this String ends with the other's characters
`equals(`**str**`)`	Returns `true` if this String is the same as *str*
`equalsIgnoreCase(`**str**`)`	Returns `true` if this String is the same as *str*, ignoring capitalization
`indexOf(`**str**`)`	Returns the first index in this String where *str* begins (-1 if not found)
`lastIndexOf(`**str**`)`	Returns the last index in this String where *str* begins (-1 if not found)
`length()`	Returns the number of characters in this String
`isEmpty()`	Returns `true` if this String is the empty string
`startsWith(`**str**`)`	Returns `true` if this String begins with the other's characters
`substring(`**i, j**`)`	Returns the characters in this String from index *i* (inclusive) to *j* (exclusive)
`substring(`**i**`)`	Returns the characters in this String from index *i* (inclusive) to the end
`toLowerCase()`	Returns a new String with all this String's letters changed to lowercase
`toUpperCase()`	Returns a new String with all this String's letters changed to uppercase

Inheritance Syntax

```
public class Example extends BaseClass {
    private type field;
    public Example() {
       field = something;
    }
    public void method() {
       // do something
    }
}

public interface InterfaceExample {
    public void method();
}
```

```
public abstract class AbstractExample {
    private type field;

    public void method() {
       // do something
    }

    public abstract void abstractMethod();
}
```

ArrayIntList

```
public class ArrayIntList {
    private int[] elementData;
    private int size;
}
```

LinkedIntList

```
public class LinkedIntList {
    private ListNode front;

    private static class ListNode {
        public int data;
        public ListNode next;

        public ListNode(int data) {
            this(data, null);
        }

        public ListNode(int data, ListNode next) {
            this.data = data;
            this.next = next;
        }
    }
}
```