CS 1332 Exam 2
Spring Semester 2021

Name (including first and last name): _____

Signature: _____

GT account username (gburdell3, etc): _____

GT account number (903XXXXXX, etc): _____

- ☕ You must have your BuzzCard or other form of identification on the table in front of you during the exam. It is your responsibility to have your ID prior to beginning the exam.
- ☕ You are not allowed to leave the exam room and return. If you leave the room for any reason, then you must submit your exam as complete. (If you need to use the restroom for an emergency, then a TA will escort you.)
- ☕ Signing and/or taking this exam signifies you are aware of and in accordance with the Academic Honor Code of Georgia Tech and the Georgia Tech Code of Conduct.
- ☕ Notes, books, calculators, phones, laptops, smart watches, headphones, earbuds, etc. are not allowed. Extra paper is not allowed. If you have exhausted all space on this exam, talk with your instructor. There are extra blank pages in the exam for extra space.
- ☕ If you plan on using ear plugs (foam or silicone, NOT AirPods) during the exam, you must show them to the instructor for approval.
- ☕ Pens/pencils and erasers are allowed. Do not share.
- 🦆 If you brought a duck with you to the exam, you may silently consult with it at any time.
- ☕ All work entered on this exam, whether code, diagrams, or multiple choice, must be implemented as was presented in lecture / module videos and recitation.
- ☕ All code must be in Java.
- ☕ Efficiency matters. For example, if you code something that uses O(n) time or worse when there is an obvious way to do it in O(1) time, your solution may lose credit. If your code traverses the data 5 times when once would be sufficient, then this also is considered poor efficiency even though both are O(n).
- ☕ Comments are not required unless a question explicitly asks for them.

---

1) Efficiency - Multiple Choice [21 points]

For each of the operations listed below, determine the time complexity of the operation as it pertains to the data structure. Select the bubble corresponding to your choice in the space provided, and completely fill in the bubble. Unless otherwise stated, assume the ***worst-case*** time complexity. However, make sure you choose the tightest Big-O upper bound possible for the operation. Do **not** use an amortized analysis for these operations ***unless*** otherwise specified.

A) What is the average case cost of finding an arbitrary key-value pair in an array-backed HashMap that uses linear probing and `H(key) = 0` for any key.

◯ O(1) ◯ O(log n) ◯ O(n) ◯ O(n log n) ◯ $O(n^2)$

B) What is the average-case cost of finding the median (i. e. Middle-most) element in a SkipList where the coin being flipped always comes up Tails?

◯ O(1) ◯ O(log n) ◯ O(n) ◯ O(n log n) ◯ $O(n^2)$

C) What is the worst-case cost of adding data to a 2-4 Tree when the addition of the data causes overflow in multiple nodes?

◯ O(1) ◯ O(log n) ◯ O(n) ◯ O(n log n) ◯ $O(n^2)$

D) What is the cost of calculating the height of a balanced binary tree?

◯ O(1) ◯ O(log n) ◯ O(n) ◯ O(n log n) ◯ $O(n^2)$

E) What is the cost of determining the second smallest value in a Min-heap without modifying the heap? You may assume that the heap has at least three more values and that you have access to the backing array.

◯ O(1) ◯ O(log n) ◯ O(n) ◯ O(n log n) ◯ $O(n^2)$

F) What is the cost of finding an arbitrary key-value pair in an array-backed HashMap that uses external chaining, uses AVL trees to handle collisions instead of Singly-Linked Lists, and `H(key) = 0` for any key.

◯ O(1) ◯ O(log n) ◯ O(n) ◯ O(n log n) ◯ $O(n^2)$

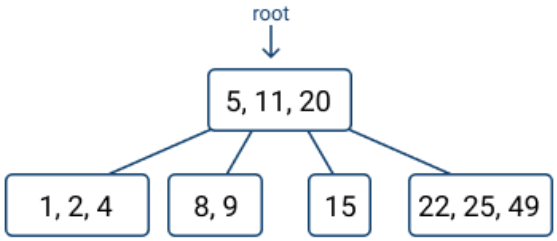
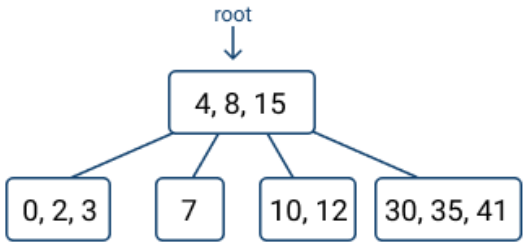
G) What is the cost of finding the value of the node that is the “deepest” (i. e. Furthest node from the root) in an AVL Tree? You may assume that the tree will only have one such deepest node, you begin search at the root.

◯ O(1) ◯ O(log n) ◯ O(n) ◯ O(n log n) ◯ $O(n^2)$

2) 2-4 Trees - Diagramming [6 points]

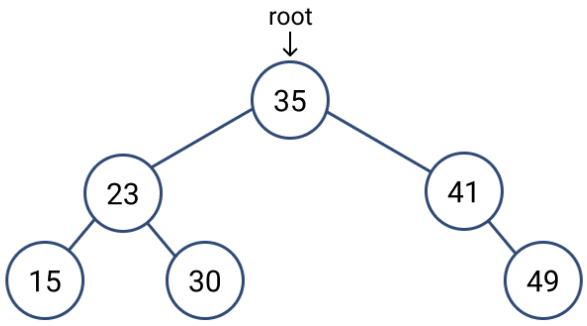
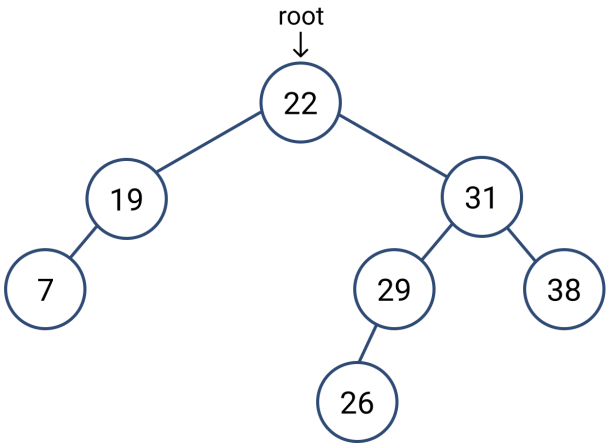
Given the following initial 2-4 trees in the left column below. Perform the stated operation, add or remove, for each tree. Draw the resulting 2-4 tree in the right column. If you want, you can draw multiple steps (**circle the final step if you do so**). Follow the implementation taught in the **1332 module videos and live lectures**.

Implementation Details:
If you need to promote an element from a node, use the **second** element. When removing from an internal node, use the **predecessor**. When checking if a transfer is possible, check the **left** sibling before the right sibling. If a fusion is necessary and the node has more than one parent data, choose the **left** parent data.

add(3) 	
remove(7) 	

3) AVL Operations - Diagramming [12 points]

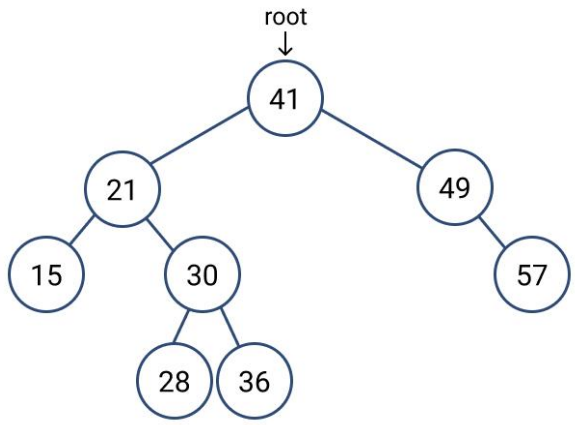
Given the following initial AVLs in the left column below. Perform the stated operation, add or remove, for each tree. Draw the resulting AVL in the right column. If you want, you can draw multiple steps (**circle the final step if you do so**). If necessary for any operation, use the **predecessor node**.

add(57) 	
***remove*(22)** 	

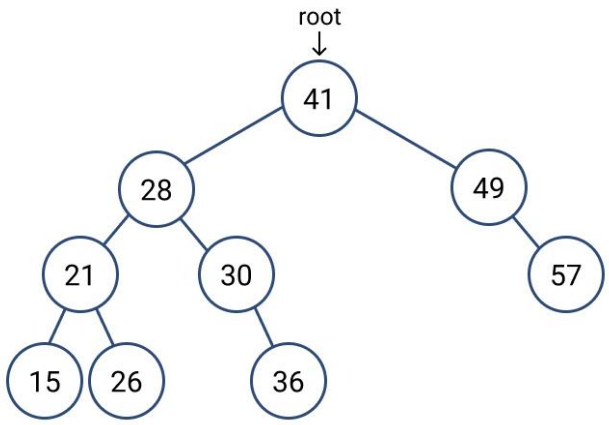
4) AVL Operations - MC [6 points]

Given the following initial AVL tree directly below. A node was **added** to the tree with the data **26**. The resulting tree after the add operation is at the bottom. Choose the correct rotation(s) that produced the resulting tree. If necessary for any operation, use the **successor node**.

Initial AVL Tree



Resulting AVL Tree



Choose the correct rotation(s) that produced the resulting tree.:

◯ Right rotation on 30

◯ Left rotation on 21

◯ Left rotation on 21 followed by the right rotation on 30

◯ Left rotation on 30 followed by the right rotation on 21

◯ Right rotation on 30 followed by the left rotation on 21

5) Heaps - Diagramming [15 points]

Given the following initial states of the array-backed MaxHeap, perform the following operations. Show all intermediate steps/swaps on a new line of the table. Indicate the swapped elements by appending an “x” to the number: for example, if 32 was swapped, write “32x.”

add(17)

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
null	18	16	5	13	12	3	2	6	9	10	

remove()

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
null	18	16	5	13	12	3	2	6	9	11	10

6) HashMap – Diagramming [10 points]

Goal: The HashMap below is backed by an array of capacity 5. **Add (-25, c)** to the HashMap. The maximum load factor for this HashMap is **0.5**.

Requirements: If you need a collision resolution strategy, use **linear probing**. Deleted entries are indicated by **DEL** markers. If you need to resize the HashMap, resize it to a capacity of **(2 × current capacity) + 1**. The hashcode of a particular number is the absolute value of the number itself. The compression function is to mod by the table length.

Initial Backing Array:

<10, d> DEL	<15, e> DEL	<12, a>		<14, c>

Final Backing Array:

7) Heaps - Coding [15 points]

Goal: Given the following **MinHeap** class, implement the **remove()** method.

Requirements: **Your code should be as efficient as possible.** You may **not** assume any other method in the MinHeap class is implemented. You may assume that **everything from java.util is imported**.

```
public class MinHeap<T> {

      private T[] backingArray;
      int size;

    /**
     * Removes and returns the smallest element in the MinHeap.
     *
     * The order property of the heap must be maintained after removing.
     *
     * @return the smallest data in the heap
     * @throws java.util.NoSuchElementException if the heap is empty
     */
     public T remove() {
         // YOUR CODE HERE, USE THE NEXT PAGE IF NEEDED
```

```
    } // END OF METHOD

} // END OF CLASS
```

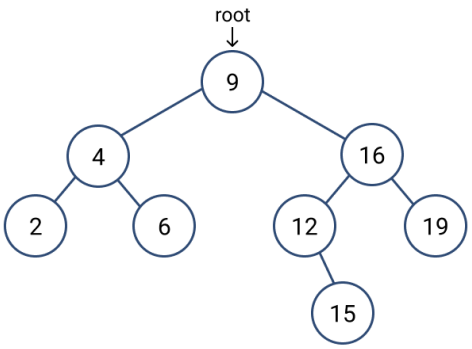
8) Binary Search Tree - Coding [15 points]

Goal: Write the countNodesLessThan() method. This method will take in a data input of type int, and you are to count the number of nodes that hold data **strictly less than (and NOT equal to)** the data input. You may assume that a valid BST has been constructed and is **not empty**, which you have access to via the instance variable root. You may also assume that every Node's data is an **int**, and that there are **no nodes that contain duplicate data or null data**.

Requirements: Your code should be as efficient as possible. Do not attempt to use any methods other than those you write yourself, as doing so may result in large deductions. However, you are allowed to write and use your own helper methods. Additionally, note that BST does **not** have a size variable, and attempting to access a size variable without declaration may result in deductions. Since Node is a private inner class of BST, you should access its fields directly (e.g. use node.data instead of getters/setters such as node.getData()).

Example:

- If given the BST shown below and the inputted data 7, you would return 3
- If given the BST shown below and the inputted data 1, you would return 0
- If given the BST shown below and the inputted data 15, you would return 5




```
public class BST {

      private class Node {
            public int data;
            public Node left;
            public Node right;
            public Node(int data, Node left, Node right) { ... }
      }

      private Node root;

     /**
      * Count the number of nodes that hold data less than the inputted data.
      *
      * @param data The data to find nodes less than
```

```
* @return An integer representing the number of nodes that hold data less
*         than the inputted data. If there are no smaller nodes, return 0.
*/
public int countNodesLessThan(int data) {
    // YOUR CODE HERE, USE THE NEXT PAGE IF NEEDED
```

```
    } // END OF METHOD

} // END OF CLASS
```

This page is blank beyond your Wildest Dreams.

(If you use this page for your work, please reference this page number on the question you are answering so we can find your response)