CS 1332 Exam 2 - Practice
Fall Semester 2025

Name: _____

GT Username: _____ **GT ID:** _____

By signing here, I understand and accept my responsibility to uphold the Georgia Tech Honor Code. I affirm that I have neither provided nor received any unauthorized aid during this exam. I will not discuss the exam contents with any other students, including those who have already taken it, until the exams are returned.

Signature: _____

- ☕ You must scan your Buzzcard before leaving the exam room or you may get a zero. Please talk to a TA if your Buzzcard is not available.
- ☕ You are not allowed to leave the exam room and return except for emergencies. If you leave the room for any other reason, then you must turn in your exam as complete.
- ☕ Raise your hand if you need clarification on a question or need to use the restroom.
- ☕ Notes, calculators, phones, laptops, smart watches, headphones, etc. are not allowed. Extra paper is not allowed. If you have exhausted all space, talk with a TA. There are blank pages in the exam for extra space. Do not share pencils or other items.
- ☕ If you plan on using ear plugs (foam or silicone, NOT earbuds) during the exam, you must show them to a TA for approval.
- 🦆 If you have a rubber duck, 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.
- ☕ All code must be in Java. Pseudocode is not sufficient for credit.
- ☕ Efficiency matters. For example, if you code something that uses O(n) time when there is a way to do it in O(1) time, your solution may lose credit. If your code traverses data 5 times when once would be sufficient, this also is considered poor efficiency even though both are O(n).
- ☕ Writing after time is called will result in a point deduction.

Table of Contents

1) Efficiency - Multiple Choice [12 points]

Goal: Determine the time complexity of each scenario below:

Requirements:

- Give the **worst-case** time complexity unless specified.
- Assume an optimal implementation, as taught in the course.
- Do not assume internal access to the data structures unless specified.
- **Do not** use amortized analysis unless specified.
- Choose the tightest Big-O bound possible for the scenario.

A.) Adding to an external chaining Hashmap where the chains are 2-4 Trees.

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

B.) Inserting *n* elements into a Skip List.

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

C.) Calculating the height of a 2-4 Tree.

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

D.) **Space complexity** of a SkipList where the coin NEVER lands on heads (head = promote).

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

E.) Removing from an AVL, where a double rotation occurs that cascades to the root.

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

F.) Sorting an array of random integers by using BuildHeap and popping each element one by one.

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

G.) Removing an element from a HashMap that uses quadratic probing to handle collisions.

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

2) Scenarios - Multiple Choice [8 points]

Goal: Match each of the following scenarios with the best possible data structure based on the problem description.

A.) Kyle dislikes cleaning, and wants to sell his old vacuum cleaner through an auction. He needs a data structure to help him record all bids and easily get the highest bid from the group. Which data structure is best suited for this?

◯ SkipList ◯ MinHeap ◯ Binary Tree ◯ MaxHeap

B.) Akhil is developing a real-time leaderboard for an online game. Players are constantly gaining and losing points, and the leaderboard must maintain a sorted order of scores while allowing fast updates, insertions, deletions, and searches of arbitrary scores. Which data structure would best suit this use case?

◯ Stack ◯ Heap ◯ Linked List ◯ SkipList

C.) Shriyan is creating an online rubber duck store. Each product has a unique product ID, and he wants to be able to quickly look up, update, and manage product information using that ID. Which data structure would best support fast lookups based on this relationship?

◯ ArrayList ◯ HashMap ◯ Linked List ◯ Binary Tree

D.) Meghna wants to create a system that tracks employee information where each employee has a unique ID and needs to be quickly located, added, or removed. The company frequently hires and fires employees, so we should ensure optimal performance for all operations. Which of the following data structures would best suit this implementation?

◯ BST ◯ ArrayList ◯ AVL ◯ Deque

3) Data Structure Properties - Multiple Select [12 points]

Goal: Match a property given to that of all valid data structures which contain this property.

Question	**Options** (**select all applicable**)
Data structure that utilizes open-addressing to handle collisions.	☐ AVL ☐ External Chaining Hashmap ☐ Quadratic Probing Hashmap ☐ Linear Probing Hashmap ☐ None of the above
Data structure which allows for insertion at the front in constant time complexity (worst case).	☐ Arraylist ☐ Singly Linked List (without tail) ☐ Linked List-backed Deque ☐ Array-backed Deque ☐ None of the above
Data structure which allows for constant time access to the largest value (worst case).	☐ AVL ☐ MaxHeap ☐ 2-4 Tree ☐ MinHeap ☐ None of the above
Data structure which maintains elements in a sorted order at all times.	☐ 2-4 Tree ☐ Array-backed Queue ☐ AVL ☐ Binary Search Tree ☐ None of the above

4) 2-4 Tree - Multiple Choice [2 points]

Roderic wants to remove data from his 2-4 Tree but needs to maintain a consistent tree height.

Goal: Given the following **initial state** of a 2-4 Tree (shown on left), **in what order are the below elements removed** from the tree to result in the **final state** of the 2-4 Tree (shown on right)?

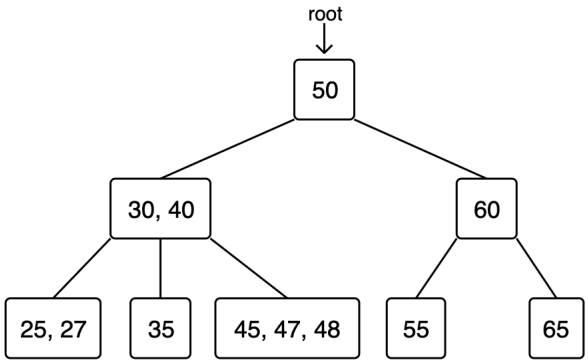
Example:

- "`X, Y, Z`" means that we call `remove(X)`, **THEN** `remove(Y)`, **THEN** `remove(Z)`.

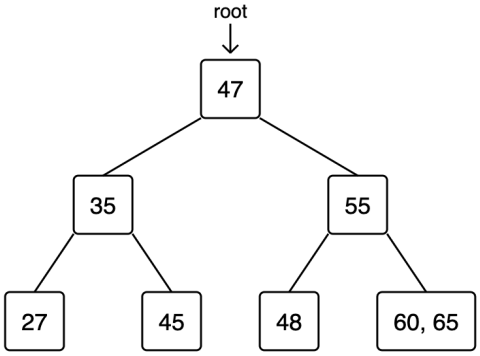
Requirements:

- If you need to promote an element from a node, use the **second** element.
- If needed for any operation, use the **successor**.
- When checking if a transfer is possible, check the **left** sibling before the right sibling.
- If fusion is necessary and the node has multiple valid parent data, choose the left parent.

Initial State:



Final State:



Correct Removal Order:

◯ `30, 25, 50, 40`

◯ `30, 25, 40, 50`

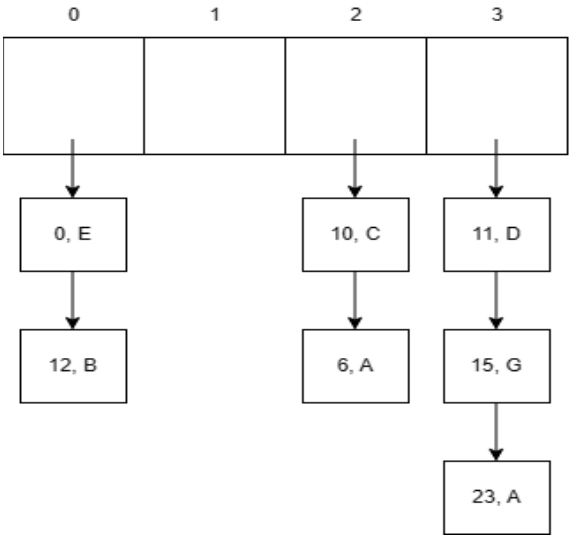
◯ `25, 30, 50, 40`

◯ `25, 30, 40, 50`

5) HashMap – Short Answer [8 points]

Goal: Given the following External Chaining HashMap, answer the following short answer questions. The hash value of the integers keys are the keys themselves. The map will not resize.

Initial Backing Array:



A) Which **index** will the key 22 be added to? [ ]

B) How many **keys will be compared** to put <15, D> into the HashMap? Ignore the put from the previous question. [ ]

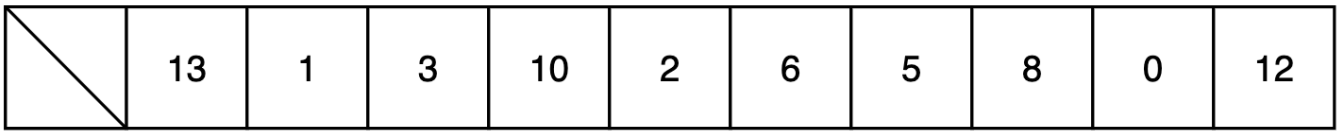
C) In several sentences, explain how to determine if the External Chaining HashMap needs to resize, and then describe each step of the resize process.

6) Heaps - Diagramming [8 points]

Goal: Given the array below, perform BuildHeap for MinHeap. Show your final MinHeap in the box below.

Requirements:

- Write the final MinHeap in **Tree** form.



BuildHeap(Minimum Heap)

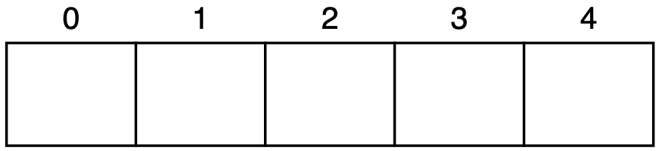
Final Minimum Heap:

7) Quadratic Probing HashMaps - Diagramming [9 points]

Goal: Trace through the following operations using the given QuadraticProbingHashMap and draw your final result in the box below.

Requirements:

- The load factor is 0.5.
- If a resize is necessary, resize to a new length of 2n + 1, where n is your previous length.
- If you show intermediate steps for your operations, make sure you **circle your final answer**.




```
put(12, A);
put(17, B);
put(23, X);
remove(12);
put(17, T);
remove(23);
put(23, Y);
```

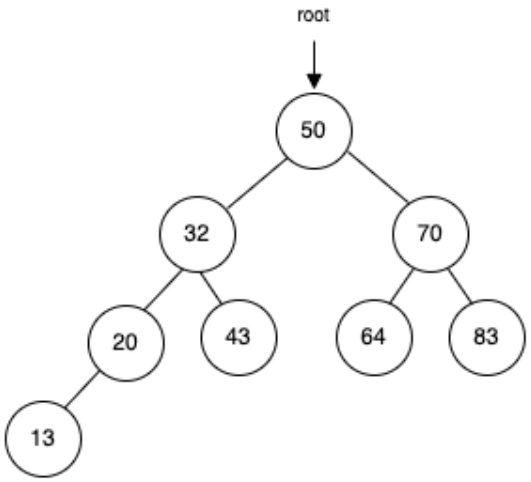
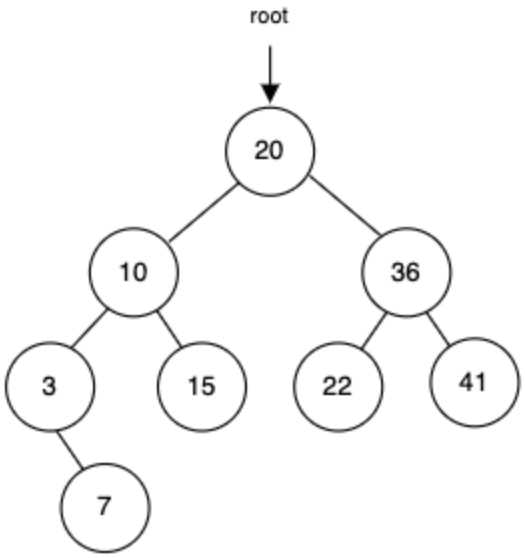
Final Result:

8) AVL - Diagramming [12 points]

Goal: Perform the following operations on the AVL trees given and draw the resulting tree following the operation.

Requirements:

- When applicable, **use the predecessor node** for your operations.
- Perform any required rotations to maintain the properties that all AVLs must preserve prior to your final answer.
- **Circle your final tree.**

	`add(11)`
	`remove(20)`

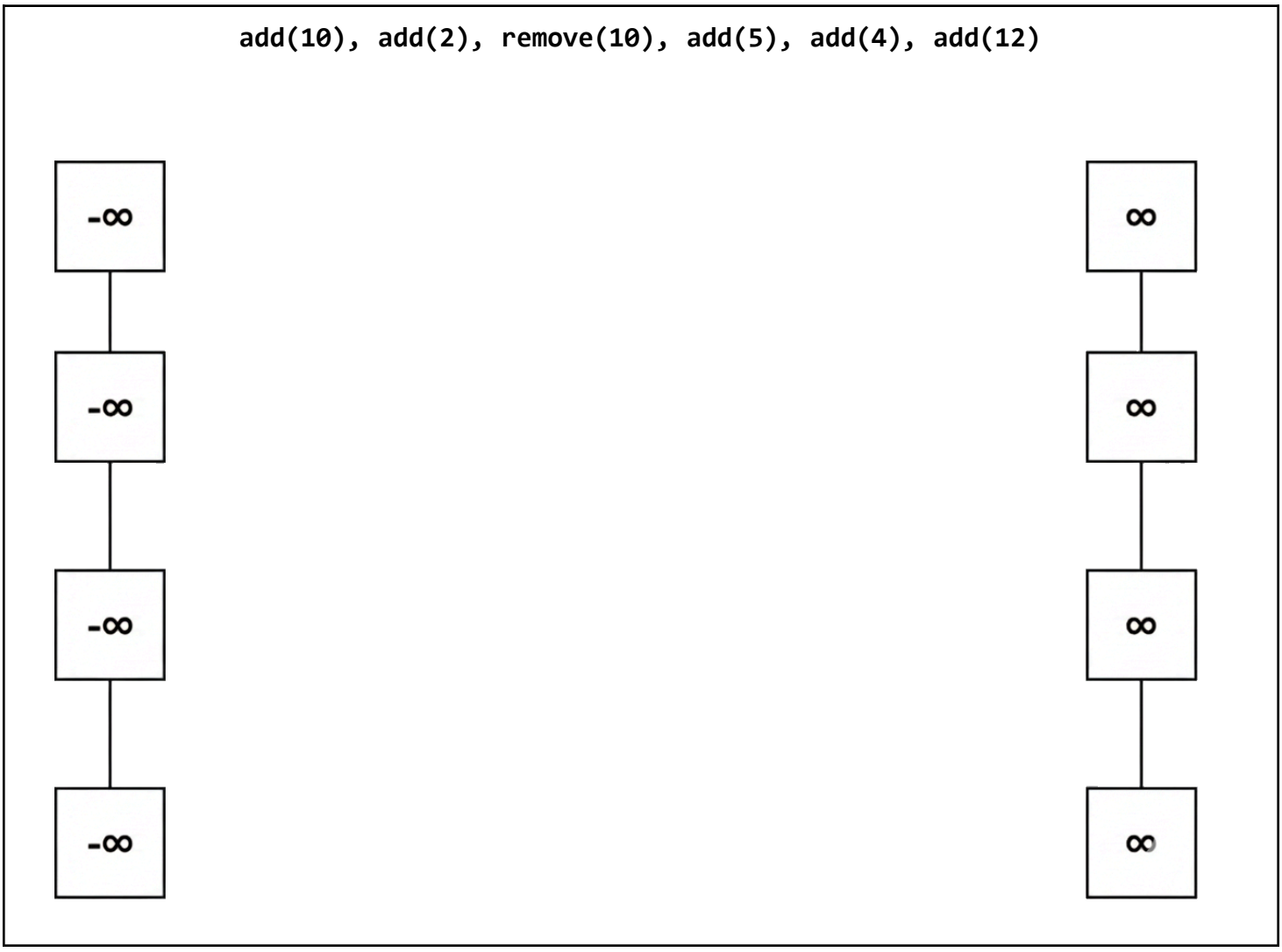
9) SkipList - Diagramming [8 points]

Goal: Given the operations below, fill out the empty SkipList correctly. Your answer should depict the SkipList in its **final state** after **all the operations are completed**.

Requirements:

- The coin flips for necessary operations are as follows: **HTTHHTHHHTHT**
- Connect the lines between nodes used in the SkipList. Any empty nodes will be disregarded as not part of the completed SkipList.
- The operations for adding and removing from SkipLists should follow methods taught in the course.

`add(10), add(2), remove(10), add(5), add(4), add(12)`

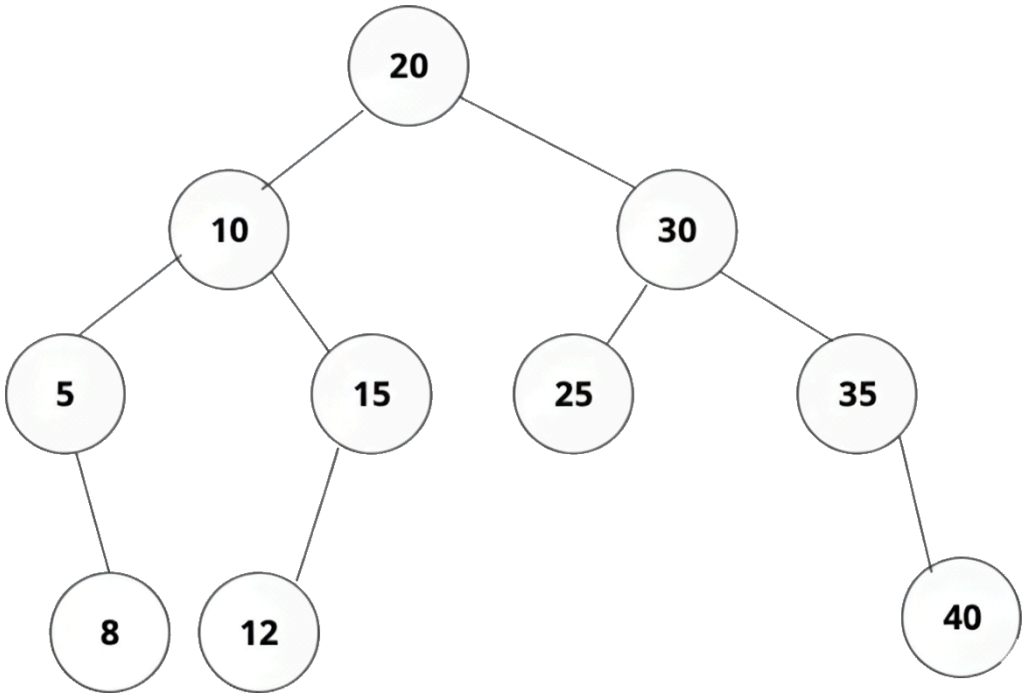


10) Binary Search Tree - Coding [15 points]

Goal: Given a BST with integers, write the `sumOutsideRange` method. The method should return an integer representing the sum outside of the range of the 2 numbers passed in. The nodes in the tree each contain a variable `subtreeSum` that contains the sum of all subtrees below and including the node.

Requirements:

- Your implementation **MUST make use of a recursive helper method.**
- You may **NOT** modify the tree or the method header provided
- You may **NOT** assume any other BST methods have been implemented
- **Your code should be as efficient as possible**



Example: `sumOutsideRange(12, 30) -> 98 (5 + 8 + 10 + 35 + 40)`

```
public class BST {

    protected class Node {

        protected int data;
        protected Node left;
        protected Node right;
        protected int subtreeSum;

    }
```

```
    protected Node root;
    protected int size;

    /**
     * Computes the sum of all node values in the BST that are outside of the
     * range a and b. In order words, sum all values
     * lower than a and all values greater than b
     * @param a the lower bound of the range
     * @param b the upper bound of the range
     * @return the sum of all nodes outside of the range
     * @throws IllegalArgumentException if a > b.
     */
     public int sumOutsideRange(int a, int b) {




     } // END OF METHOD - WRITE ANY HELPER METHODS BELOW THIS LINE




} // END OF CLASS
```

Blank page.