IFT2015-A25: HW3

November 4, 2025

General Instructions

- **Due date: Monday, November 18th at 11:59 PM.**
- This assignment is to be completed individually or in pairs.
- No plagiarism will be accepted.
- It is possible that clarifications or modifications to the instructions will be sent by email and posted on Studium.
- Each day late in submitting your work will incur a penalty of 5 points.
- All code must be written in Java.
- You must submit the required Java files (`Q1.java`, `Q2.java`, `Q3.java`).
- **If you are working as a team:** one person submits the required files and the other submits only a `.txt` file with their partner's name.
- Code evaluation may be automated; make sure you have **the same function signatures**.
- You may use `java.util.*` for all problems.
- You are allowed to create as many helper functions as you wish.
- **Don't forget to add comments to your implementations. Undocumented code will be penalized.**
- For any questions regarding HW3, please post them on the "TP3 Q&A" forum on Studium.
- Good luck!

1 Problem 1 (5pts) - BST for Version Control System

Context

You are managing a source code version control system (like Git) where each commit is identified by a unique timestamp. You need to efficiently navigate the commit history using a Binary Search Tree.

Each tree node contains:

- `int timestamp`: The unique identifier (BST key)
- `String commitHash`: The commit hash (e.g., "a1b2c3d")
- `String author`: The author's name

Tasks to implement in `Q1.java`

1. `void insert(int timestamp, String commitHash, String author)`
   Insert a new commit into the binary search tree.
2. `String findCommit(int timestamp)`
   Find and return the hash of the commit with the given timestamp. Return `null` if not found.
3. `List<String> getCommitsBetween(int startTime, int endTime)`
   Return all commit hashes between `startTime` and `endTime` (inclusive) in chronological order.
4. `String findNearestCommit(int timestamp)`
   Find the commit closest to a given timestamp.

   Algorithm:

   - If number $\geq$ timestamp: search in left subtree
   - If number $<$ timestamp: search in right subtree
   - Keep track of the closest seen so far
5. `int countCommitsByAuthor(String author)`
   Count, the number of commits by a specific author.
6. `String getMostActiveAuthor()`
   Return the name of the author, with the most commits. In case of a tie, return the first one you find.
7. `void revertToCommit(int timestamp)`
   Delete all commits after (strictly greater than) the given timestamp.

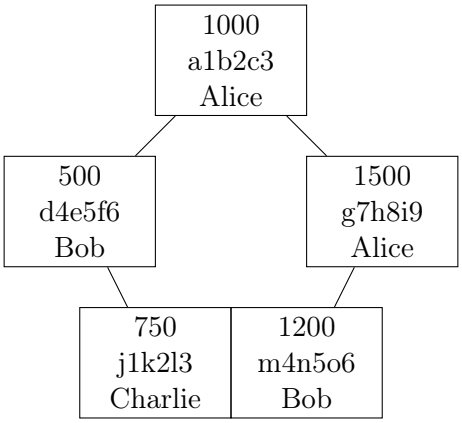
   Note: This operation modifies the tree.
8. `List<String> getTimeline()`
   Return all commits in chronological order with the format:
   `"[timestamp] hash - author"`

Example of execution

Initial commits:

```
insert(1000, "a1b2c3", "Alice")
insert(500, "d4e5f6", "Bob")
insert(1500, "g7h8i9", "Alice")
insert(750, "j1k2l3", "Charlie")
insert(1200, "m4n5o6", "Bob")
```

Resulting BST structure:



Some operations:

```
findCommit(750) -> "j1k2l3"

getCommitsBetween(600, 1300) -> ["j1k2l3", "a1b2c3", "m4n5o6"]

findNearestCommit(900) -> "a1b2c3"  (distance 100)

countCommitsByAuthor("Alice") -> 2

getMostActiveAuthor() -> "Alice" or "Bob" (tie at 2)

revertToCommit(1100)  // Deletes 1200 and 1500
getTimeline() -> ["[500] d4e5f6 - Bob",
"[750] j1k2l3 - Charlie",
"[1000] a1b2c3 - Alice"]
```

2 Problem 2 (5pts) - BST Structure Analysis in Binary Tree

Context

You are developing a code analysis tool that examines binary trees to identify subtrees that are valid Binary Search Trees (BST). The tool must calculate the maximum sum of keys among all valid BST subtrees.

A BST is defined by:

- The left subtree contains only nodes with keys strictly less than the node's key
- The right subtree contains only nodes with keys strictly greater than the node's key
- Both subtrees must be BSTs

Tasks to implement in `Q2.java`

1. `int maxSumBST(TreeNode root)`
   Return the maximum sum of keys of a valid BST subtree. If no valid BST subtree exists, return 0.

2. `boolean isValidBST(TreeNode root)`
   Verify if the entire tree is a valid BST.

3. `List<Integer> findAllBSTRoots()`
   Return the values of roots of all valid BST subtrees, inascending order.

4. `int countValidBSTs()`
   Count the total number of valid BST subtrees in the tree.

5. `int getMinBSTSum()`
   Return the minimum sum among all valid BST subtrees. Return `Integer.MAX_VALUE` if none exist.

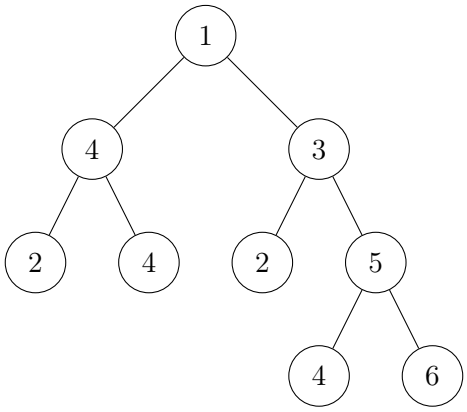
6. `Map<Integer, Integer> getBSTSizeDistribution()`
   Return a Map: subtree size → number of BST subtrees of that size.

7. `TreeNode findLargestBST()`
   Return the root of the largest BST subtree (by number of nodes). In case of tie, return the one with the sum.

8. `List<Integer> getInorderBST(TreeNode root)`
   If the subtree with `root` is a BST, return its inorder traversal. Otherwise, return an empty list.

Example execution

Input tree:



BST subtree analysis:

```
Subtree with root 2 (left): BST, sum = 2
Subtree with root 4 (right): BST, sum = 4
Subtree with root 4 (parent): Non-BST
Subtree with root 2 (under 3): BST, sum = 2
Subtree with root 5: BST, sum = 20 (4+5+6)
Subtree with root 3: BST, sum = 20
Subtree with root 1: Non-BST
```

Operation results:

```
maxSumBST(root) -> 20

isValidBST(root) -> false

findAllBSTRoots() -> [2, 2, 3, 4, 4, 5, 6]

countValidBSTs() -> 7

getMinBSTSum() -> 2

getBSTSizeDistribution() -> {1: 5, 3: 1, 5: 1}

findLargestBST() -> TreeNode(3)  (5 nodes)

getInorderBST(TreeNode(5)) -> [4, 5, 6]
```

3 Problem 3 (5pts) - Viral Growth Simulation

Context

You are working for a virology lab thatsimulates virus propagation. Each virus sample has an initial population and a multiplication rate. At each simulation cycle, the sample with the **smallest population** reproduces (its population is multiplied by its rate). If multiple samples have the same minimum population, the one with the smallest index is chosen.

To efficiently manage these simulations with potentially millions of cycles, you must use a **min heap**.

Important note: Populations can become very large. After all cycles, apply modulo $10^9 + 7$ to all values.

Tasks to implement in `Q3.java`

1. `void buildHeap(int[] populations, int[] multipliers)`
   Build the initial min heap with the samples. Each heap element must contain the population, the sample's index, and its multiplier.

   Comparison criteria: If two populations are equal, the one with the smaller index has priority.

2. `int[] simulate(int cycles)`
   Perform `cycles` multiplication operations. At each cycle: extract the min, multiply its population by its rate, reinsert it. Return the array of final populations after modulo $10^9 + 7$.

   Example : `simulate(5)` with `pops=[2,1,3,5,6], mults=[2,2,2,2,2]`
   → `[8, 4, 6, 5, 6]`

3. `int[] simulateOptimized(int cycles)`
   Optimized version for very large numbers of cycles (up to 10^9).

   Key observation: Once all populations become similar, you can mathematically calculate the result instead of simulating cycle by cycle.

   Suggested algorithm:

- While $\frac{\max}{\min} \geq$ multiplier, simulate normally
- Then, calculate how many times each sample will be multiplied
- Use fast exponentiation for large powers

Example : `pops=[100000,2000], mults=[1000000,1000000], cycles=2`
→ `[999999307, 999999993]`

4. `long[] getCurrentState()`
   Return the current state of populations (without modulo) in original index order.

5. `int findMinIndex()`
   Return the index of the sample that will be multiplied in the next cycle.

   Example : If heap = `[(1, idx=3), (2, idx=0), ...]` → returns `3`

6. `long predictPopulation(int sampleIndex, int futureCycles)`
   Predict the population of sample `sampleIndex` after `futureCycles` additional cycles.

   Note: Simulate mentally without modifying the heap.

Example of execution

Configuration:

```
populations = [5, 2, 8]
multipliers = [3, 2, 4]
cycles = 3
```

Simulation:

```
Initial state: [5, 2, 8]
Cycle 1: min=2 (idx=1, mult=2) -> [5, 4, 8]
Cycle 2: min=4 (idx=1, mult=2) -> [5, 8, 8]
Cycle 3: min=5 (idx=0, mult=3) -> [15, 8, 8]

Result: [15, 8, 8]
```

Academic Integrity Attestation

Mandatory Declaration

For each problem solved in this assignment, you must include in your code comments an attestation regarding the use of generative artificial intelligence tools.

Required Format

At the beginning of each file (`Q1.java`, `Q2.java`, `Q3.java`), add the following comment:

```
/*
* ACADEMIC INTEGRITY ATTESTATION
*
* [ ] I certify that I have not used any generative AI tool
*     to solve this problem.
*
* [ ] I have used one or more generative AI tools.
*     Details below:
*
*     Tool(s) used: ________________________________
*
*     Reason(s) for use:
*     ____________________________________________________
*     ____________________________________________________
*
*     Affected code sections:
*     ____________________________________________________
*     ____________________________________________________
*/
```

Instructions

- Check the appropriate box by replacing `[ ]` with `[X]`.
- If you used an AI tool, you must fill in all fields.
- AI tools include (but are not limited to): ChatGPT, Claude, GitHub Copilot, Bard, etc.
- **Failure to comply with this requirement or any false declaration will be considered a violation of the academic honor code.**
- AI use must be justified and generated code sections must be clearly identified.