Data Structures 1 – Spring 2025 - Assignment A3

version 1.0

Rules:

- **Submission Deadline: June 1 at 23:59**. Late submissions will incur the following penalties:
  - **June 2:** 10-point deduction
  - **June 3:** 20-point deduction
  - **June 4:** 30-point deduction

  Submissions will not be accepted after June 4.
- The assignment should be completed in groups of two students (already assigned in Moodle).
- Students will be randomly selected to provide an oral explanation of their solution to the exercises.
- Questions regarding the exercise statements are welcome during office hours. However, teachers will not assess your solution.
- There is no guarantee that questions submitted via email will receive a response. Should you decide to submit questions by email anyway, you must address both professors: `herman.schinca@gtiit.edu.cn` and `hernan.melgratti@gtiit.edu.cn` . **Emails sent to only one professor will not be answered and no notification will be provided**.
- The code must be implemented in Java, compiled in VSCode, and pass the developed tests. This is the minimum requirement. Note that graders and professors may run additional tests. Furthermore, passing tests is not a sufficient condition for correctness. The quality of the submitted code will also be graded.
- Submit **one zip file containing the code for each exercise** (wet part) and **one PDF for all other exercises** (dry part).

THE DRY PART MUST BE PRODUCED USING A TEXT EDITOR (e.g., MS Word, LaTeX, or similar). Handwritten submissions are not permitted.

Plagiarism and Academic Misconduct:

In cases of copying or plagiarism, exams will be placed under review. If plagiarism is confirmed, the case will be reported to the GTIIT Discipline Committee. All group members will be held accountable for any submission that violates this policy. **THE USE OF AI TOOLS IS PERMITTED FOR THIS ASSIGNMENT**

Exercise 1: Quicksort [25 points]

In this exercise, we focus on the quicksort algorithm.

Tasks 1: Specification

Consider the following variation of the Lomuto's partitioning algorithm, designed to sort an array in **decreasing order**

```
def lomuto(a, l, r):
    i = r
    for j <- r to l step -1
        if a[j] < a[l]
            swap a[i] and a[j]
            i -= 1
    swap a[l] and a[i]
    return i
```

We aim to formally define the contract (preconditions and postconditions) of the modified Lomuto partitioning algorithm. An AI tool proposed the following specification:

Precondition (`Pre(a, l, r)`)

- `0 ≤ l ≤ r < length(a)`

Postcondition (`Post(a, l, r, i)`)

1. **Partitioning:**
   - For all `k` in `[l, i−1]` : `a[k] ≥ a[i]` ; and
   - For all `k` in `[i+1, r]` : `a[k] < a[i]` ; and
2. **Pivot Preservation:**
   - `a[i] == original(a[l])`

Determine whether the given specification (preconditions and postconditions) correctly describes the behavior of the modified Lomuto partitioning algorithm. If the specification is flawed:

- Identify the exact issues.
- Propose corrections to ensure the specification matches the expected algorithm's behavior.

Task 2: Correctness

Determine whether the algorithm is correct with respect to the correct specification of the problem (determined in the previous task).

- If the algorithm is correct, provide a formal correctness proof. In your proof, clearly state the loop invariant and use loop invariant theorem to justify the correctness of the algorithm.
- If the algorithm is not correct, first propose a corrected version of the algorithm. Your revised algorithm must still traverse the array from right to left (i.e., cannot scan left to right). Then, prove the correctness of your corrected algorithm by precisely stating the loop invariant and providing a detailed, rigorous correctness proof based on it.

NOTE: A vague or incomplete proof, or one that resembles a proof sketch generated by AI tools, will not be accepted and will result in zero marks for this question.

Exercise 2: Treaps [25 points]

Tasks 1: Possible outputs of Treap generation

We examine the construction of a treap using a fixed sequence of 3 elements. Our goal is to determine how many distinct tree structures (shapes) can result from inserting the same sequence `[1, 2, 3]` in that exact order.

When we repeatedly ask **DeepSeek-V3** the following question:

```
"How many different shapes of trees with 3 nodes can be obtained
if the Treap is generated by inserting the sequence of elements
[1,2,3] in that order?"
```

we receive varying answers—such as 2, 3, 4, or 5—after lengthy explanations. Similarly, ChatGPT tends to suggest 3, 4, or 5.

What is the correct answer to this problem? Justify your response by explicitly constructing all possible distinct treaps.

Note: Given that this problem involves querying AI tools to generate possible solutions, submitting an incorrect answer (i.e., listing fewer or more treaps than the correct number) will result in a grade of 0.

Task 2 Montecarlo approach for assessing the expected height of Treap

We recall the principles of the Monte Carlo approach, as introduced in Introduction to Computer Science M. The Monte Carlo method is a computational technique that relies on random sampling to solve problems that, although theoretically solvable analytically, may be difficult or complex to resolve directly. The core idea is to simulate many random outcomes of a process, observe the results, and use statistical analysis to estimate the quantity of interest.

In this assignment, you will use the Monte Carlo approach to assess whether the expected height of a Treap with `n` elements is `O(log n)`. The expected outcome is a discussion on whether your experimental results align with the theoretical expectation that the height of a Treap with `n` nodes is `O(log n)`. You may support your conclusion with graphical representations of your results.

Hint: Modify your Treap implementation as needed to collect relevant data. Write a Java program that uses the Monte Carlo method to estimate the expected height of randomly generated Treaps for various values of `n`. Your program does not need to generate plots; it should simply output the collected data. You can analyze this data graphically later using external tools or even manually. Be sure to include the data used to support your analysis.

Exercise 3: Tries [25 points]

We aim to design an algorithm that, given a word (which may or may not exist in a Trie), returns, if exists, the next word in **lexicographical order** stored within the Trie.

In this exercise, we will analyze and build upon a solution proposed by **DeepSeek-V3**, which is provided in the companion file `Trie.java`. For simplicity and testing purposes, we have modified the implementation by changing the visibility of the helper methods `findMinWord` and `findNextWordFromNode` to `public`.

Task 1: Verifying Correctness of the Provided Solution

Your first task is to evaluate the correctness of the provided solution. To do this, you should design a comprehensive test suite that systematically checks the behavior of **all public methods** in the `Trie` class, including:

- `insert(String word)`
- `nextWordInTrie(String word)`
- `findMinWord(TrieNode node)`
- `findNextWordFromNode(TrieNode node, String prefix)`

Provide your test suite as JUnit tests.

Note: The following two tasks are not required if you do not identify any bug.

Task 2: Bug Identification

If you identify any **bugs** or unintended behaviors during testing, you must clearly document each bug, including a description of the issue and the specific test case that reveals it.

Task 3: Bug Fixing

For each identified **bug**, update the implementation to fix the issue. Make sure that your revised implementation successfully passes **all** test cases from Task 1.

Excercise 4: Hash Tables [25 points]

Task 1: Open addressing with linear probing:

We would like to assess the quality of alternative solutions for the implementation of a hash table with open addressing with linear probing. If we query **DeepSeek** and **ChatGPT** with the following question:

```
I need to store approximately 100 integer keys in a hash table using open
addressing with linear probing. Please suggest an appropriate hash table
size and propose at least three different hash functions suitable for use
with linear probing.
```

we obtain the following answers:

- Suggested table size: 149, and the following hash functions for the first probe.

```java
public static int hash1(int key, int tableSize) {
    double A = 0.6180339887; // (√5 - 1) / 2, golden ratio fraction
    double frac = (key * A) % 1;
    return (int)(tableSize * frac);
}
```

```java
public static int hash2(int key, int tableSize) {
    int prime = 31; // A small prime multiplier
    return (key * prime) % tableSize;
}


public static int hash3(int key, int tableSize) {
    key ^= (key >>> 20) ^ (key >>> 12);
    key = key ^ (key >>> 7) ^ (key >>> 4);
    return key % tableSize;
}
```

- Suggested table size: 151, and the following hash functions for the first index.

```java
public int hashBest(int key,int tableSize) {
    key ^= (key >>> 16);  // Scramble lower and higher bits
    return Math.abs(key) %  tableSize;  // Prime table size
}


public int hashMul(int key, int tableSize) {
    double A = 0.61803398875;  // (√5 - 1)/2 (Knuth's golden ratio)
    double product = key * A;
    return (int) Math.abs(tableSize * (product - (int) product));
}


public int hashDiv(int key, tableSize) {
    return Math.abs(key) % tableSize;  // Must use a prime size!
}
```

Use the Monte Carlo approach to compare the different hash table strategies based on the following metrics:

- The expected number of probes required for a successful search
- The expected number of probes required for an unsuccessful search
- The expected size of the largest primary cluster (i.e., the longest consecutive sequence of occupied slots)

Hints: Compute the measures after having inserted exactly 100 keys (non-duplicated keys). For the generation of the keys use `Random.nextInt` in Java.

In addition to providing the code used for the Monte Carlo experiment, you should also draw conclusions about the performance of the different hash functions in the given context.

Task 2: Hashing with Overflow Area (Hashing with Overflow Buckets)

Consider an alternative approach to handling collisions in a hash table. In this method, the table is divided into two distinct regions:

- **Primary Area (Main Table):** Keys are initially placed here based on a hash function.
- **Overflow Area (Overflow Table or Bucket):** If a collision occurs (i.e., the hashed slot is already occupied), the key is inserted into a separate overflow area. This area is **not hashed**—entries are simply appended or stored

sequentially. If the overflow area is full, the key **cannot be inserted** into the table.

Implement a hash table that uses an overflow area. The sizes of the primary area and the overflow area should be specified at the time of creation. Provide appropriate test cases for your implementation.

Since you may use an AI tool to generate the implementation of the hash table, a significant portion of your grade will be based on the quality of the test suite you design to validate your implementation. Your implementation must pass all the provided tests.

Task 3: Performance Assessment

Revisit the Monte Carlo experiment from **Task 1**, this time using the new hash table design with an overflow area. In this case collect the following metrics:

- The expected number of probes required for a successful search
- The expected number of probes required for an unsuccessful search
- The expected number of insertions that fail due to the overflow area being full.

For the experiment consider the following table size:

- Primary: 101,
- Overflow: extra space for 20.

In addition to providing the code used for the Monte Carlo experiment, you should also draw conclusions about the performance of the different hash functions in the given context. Your discussion should also compare the hashing technique with Overflow Area against the results obtained in task 1.