Data Structures 1 – Spring 2025 - Assignment A2

version 1.0

Rules:

- **Submission Deadline: May 9 at 13:00**. Late submissions will incur the following penalties:
  - **May 9 (after 13:00) – May 10:** 10-point deduction
  - **May 11:** 20-point deduction
  - **May 12:** 30-point deduction

  Submissions will not be accepted after May 12.
- 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. **This includes submitting solutions found online or generated by AI tools.** All group members will be held accountable for any submission that violates this policy.

Exercise 1: Min-Heap [20 points]

In this exercise, we focus on the **min-heap** variant of the heap-tree data structure discussed in the lecture. We will adopt the same representation structure used for the `max-heap`.

Tasks

1. **Representation Invariant and Abstraction Function**

   Provide a modified version of the *representation invariant* and *abstraction function* that were introduced for the max-heap structure in the lecture. Your version should reflect the semantics of a `min-heap`.

2. **Checking the Representation Invariant**

   Implement a Java function:

   ```
   public static <T extends Comparable<T>>
   boolean minHeapRepOK(T[] elems, int start, int end)
   ```

   This function takes an array of elements and checks whether the subarray `elems[start..end]` satisfies the min-heap representation invariant. The function should return `true` if the invariant holds and `false` otherwise.

   Additionally, provide relevant test cases to check your implementation.

3. **MIN-HEAPIFY**

   Like the `MAX-HEAPIFY` procedure discussed in the lecture, a min-heap relies on a `MIN-HEAPIFY` algorithm to restore the heap property after insertions. Although we covered a recursive version in class, `MIN-HEAPIFY` can also be implemented iteratively using loops.

   Write a **pseudo-code definition** of an **iterative** version of `MIN-HEAPIFY`. Your implementation should have worst-case time complexity `O(log n)`, where `n` is the number of nodes in the heap.

4. **MIN-HEAPIFY Complexity Analysis**

   Prove that your iterative implementation of `MIN-HEAPIFY` has worst-case time complexity `O(log n)`, where `n` is the number of nodes in the heap.

5. **Correctness Proof**

   Show that your `MIN-HEAPIFY` implementation is correct. Begin by clearly stating the algorithm's **contract**, i.e., its preconditions and postconditions. Your correctness argument should include loop invariants and use the **loop invariant theorem** to demonstrate that the algorithm is correct.

Exercise 2: Min-heap over dynamic structures[20 points]

`Heap` data structures are commonly implemented over arrays. We will now consider the challenges of implementing them on a dynamic data structure. We will consider an implementation consisting of a binary tree seen in lecture, in which each node has an element, and the references to its left and right children and its parent.

Tasks: 1. **Representation invariant**: Define the representation invariant for a `min-heap` data structure over a dynamic structure.

Implement a Java function:

```java
public static <T extends Comparable<T>>
boolean minHeapRepOK(Node<T> root)
```

The function checks whether the subtree rooted at `root` is a `min-heap`.

Your solution should provide relevants test for checking the given implementation.

2. **Insertion** Define an algorithm for inserting an element in a `min-heap`. You may consider defining the following auxiliary operations:
   - `parentOfNew(Node<T> node)` that returns the node that would be the parent of the node to be added next. This operation may return `null` if root is an empty tree.
   - `MIN-HEAPIFY(Node<T> node)` that restablises the `min-heap` property starting from the `node` by assumming that the only node that may violate the `min-heap` property is `node`.
3. **Complexity of Insertion** Show the complexity of the algorithm proposed above.
4. **Java implementation** Give a Java implementation of the proposed algorithm for insertion. Your solution should provide meaningful tests for checking your implementation.

Exercise 3: Sorted Arrays, Balanced and Binary Search Trees [30 points]

An **almost-complete binary search tree** can be represented by a sorted array. Example: `[1, 2, 3, 4, 5, 6]` corresponds to

```
      4
    /   \
   2     6
  / \   /
 1   3 5
```

Tasks 1. **Representation Invariant** Define the representation for **almost-complete binary search trees** over an array and give the representation invariant.

2. **Almost-Complete BST as AVL** Explain why an almost-complete binary search tree is automatically an AVL tree.

3. **Naive AVL Construction Algorithm** Given the following algorithm to build an AVL from an ordered array:

```
AVLNode <T> build(<T> A)
  root = null
  for i = 1 to |A|
    root = insertAVL(root, A[i]) // Standard AVL insertion
  return root;
```

3. a) For the sorted sequence `[1, 2, ..., 11]`, compute the total number of rotations performed by `build()`.

Hint: To avoid manual computation, automate rotation counting by instrumenting your AVL implementation to increment a counter whenever a rotation occurs during insertion.

3. b) Is the statement **"For any sorted array `A`, `build(A)` returns an almost-complete AVL"** true? Justify.

4. **Rotation-Free AVL Construction** Propose an algorithm to build an AVL from a sorted array **without** rotations. *Hint*: Use a divide-and-conquer approach.

5. **Complexity** Analyse the worst-case runtime complexity of the algorithm proposed in the previous question.

6. **Implementation and Testing** Implement your algorithm in Java and provide tests verifying:

- The output is an AVL tree.
- Zero rotations occur during construction.

7. **Almost-Completeness** Does your algorithm always generate an almost-complete AVL? Justify. If not, modify it to ensure this property.

Exercise 4: Galactic Siege: The Last Line of Defense [30 points]

Problem Description

We are require to evolve the game developed in the Assignment 01 do deal with the following new requirements.

a. Warrior select a suitable shield

Warriors' shields are no longer organized in a layered structure. Instead, each warrior now possesses a set of shields. When repelling an attack, the warrior selects the shield with the minimum power sufficient to block the attack. If no shield can repel the attack, the warrior defaults to using the shield with the highest available power.

Example 1: Repelling Attacks **Available Shields:** `{75, 50, 100}`

Case 1: Attack Power = 50

- **Shield Used:** 50 (exact match)
- **Outcome:**
  - Shield is destroyed (complete defense)
  - Remaining shields: `{75, 100}`

Case 2: Attack Power = 70

- **Shield Used:** 75 (smallest sufficient shield)
- **Outcome:**
  - Power of the shield reduced by 70 (75 - 70 = 5)
  - Remaining shields: `{5, 50, 100}`

Case 3: Attack Power = 170

- **Defense Sequence:**
  1. Uses shield 100 (destroyed)
  2. Uses shield 75 (reduced by remaining 70 attack power)
- **Outcome:**
  - One shield removed, other's shield power updated: 5 (75 - 70)
  - Remaining shields: `{5, 50}`

b. Army Squadron

Warriors are now organized into **squadrons**. Each squadron contains up to 100 warriors and warriors in a squadron alternate defending against attack strategies

Defense Priority Rules

1. The **healthiest warrior** defends next attack
2. Health is measured by **total remaining shield power**
3. After defending, warrior's power is updated
4. Selection re-evaluated for each new attack

Example Scenario

Initial Warrior States

Warrior	Shields	Remaining Power
W1	{90, 80, 30}	200
W2	{75, 50, 100}	225
W3	{15, 50}	65

Attack Sequence

1. **First Attack**: [50]
   - Defender: W2 (highest power: 225)
   - Defense:
     - Uses 50-power shield (exact match)
     - Shield destroyed
   - **Updated State**:

Warrior	Shields	Remaining Power
W1	{90, 80, 30}	200
W2	{75, 100}	175
W3	{15, 50}	65

- Next defender selection: **New defender**: W1

Objective

Improve, adapt and extend the solution of Assignment 1 to the new requirements.

Your solution should be **modular**. Specifically, you must identify appropriate **data structures** (such as lists, queues, or stacks) and implement the necessary abstractions. For example, if the warrior's shields are stored in a list, then **list**

operations should not be part of the warrior's logic—instead, the warrior should interact with the list through the corresponding interface.

The following is the **minimal set of required operations** for each abstraction. However, you may need to implement **additional operations with appropriate complexity** to support these functionalities. In all cases, complexity refers to worst-time runtime complexity.

Attack Strategy (`AttackStrategy`)

- `add(int) (O(1))` Adds a new attack with the given power to the strategy.
  - The new attack is applied **after** all existing attacks in the strategy.
  - The operation fails by throwing a suitable exception if the given power is **not positive**.

Warrior (`Warrior`)

In the following, `S` represents the number of shields a warrior possesses, and `A` represents the number of attacks in the applied attack strategy.

- `boolean alive() (O(1))` Returns whether the warrior is alive, i.e., whether they still have shields.

- `void addShield(int) (O(log S))` Adds an **shield** with the given power to the warrior.

  - The implementation should ensure that power is **positive**. If it is not, the operation should fail by throwing a suitable exception.

- `boolean repel(int) (O(S * log S))` Applies an attack of the given power to the warrior.

  - Shields are used by selecting the smallest one capable of repelling the attack. If no single shield can repel the attack on its own, they are used successively in order of decreasing strength until the attack is neutralized or no shields remain.
  - Returns `true` if the attack is repelled, `false` if the warrior is **defeated**.
  - This method **modifies the warrior**, meaning that after the attack, only **remaining shields** are kept.

- `boolean repel(AttackStrategy)` Applies all attacks in the given strategy, one by one.

  - The process stops if an attack **cannot be repelled**.
  - Returns `true` if the **entire** strategy is repelled, `false` otherwise.
  - This method **modifies the warrior**, removing destroyed shields.

- `int shields() (O(1))` Returns the **number of shields** the warrior has left.

- `int remainingPower() (O(1))` Returns the **sum of the power of all the shields** the warrior has.

Army Squadron (`ArmySquadron`)

- `Warrior next() O(1)` This operation returns the healthiest warrior in the squadron. If multiple warriors have the maximum remaining power, it may return any one of them. The operation raises an exception if called on an empty squadron. This operation does not modify the squadron.

In the following we use:

- `W` = Number of warriors in the squadron
- `n` = Length of the input `AttackStrategy[]` array
- `l` = Length of the longest attack strategy
- `S` = Maximum number of shields per warrior
- `int repel(AttackStrategy[])` `O(n * (log W + l * S * log S))` This operation:

1. Iteratively selects the healthiest warrior (highest remaining shield power)
2. Applies the next unexecuted `AttackStrategy` from the sequence
3. Finally, it returns the count of defeated warriors
4. **Important**:
    - Each `AttackStrategy` in the sequence is applied to at most one warrior
    - If a warrior is defeated mid-strategy, all remaining attacks in that strategy are discarded. However, subsequent strategies in the sequence continue to be applied until either the sequence completes or no warriors remain in the squadron.

Tasks

1. Adapt the definition of the classes `AttackStrategy` and `Warrior` to the new functionalies and the required runtime complexity. Provide a complexity analysis of the implemented methods to demonstrate that they satisfy the required complexity. When choosing the appropriate data structures note that the number of shields a warrior can have and the number of attacks in a strategy are unbounded.

2. Show the complexity of the operation `repel(AttackStrategy)` of the type `Warrior`.

3. Give the implemetation of `ArmySquadron`:

- Define its implementation and gives the representation invariant.

- Provide the pseudocode for the required operations and also for the operations nedeed for the manipulation of the selected structure.
- show that the operation `repel(AttackStrategy[])` and `next()` have the required complexity.

4. Implement your solution in Java by modifying assignment 01. You should adapt `AttackStrategy` and `Warrior` and implement `ArmySquadron`, along with any supporting data structures, observing the following additional requirements:
    - If your implementation requires collections such as lists, stacks, or queues, you must define and implement them yourself. Your solution cannot use any collection provided by Java (e.g., the Collections library) or any other external implementation. The use of basic arrays (e.g., `int[]`, `double[]`, etc.) is allowed.
    - Your source code should be modular, meaning you should provide separate source files for `AttackStrategy`, `Warrior`, `ArmySquadron` and any collections you define. Additionally, the interface for each auxiliary data structure should be provided in a separate file.
5. Provide tests for both the collections you have implemented and the ADTs `AttackStrategy`, `Warrior` and `ArmySquadron`. The quality of the tests you provide will be evaluated.

Submission Requirement Notice Please address all issues identified in the grader's feedback when revising your code. Note that:

- Resubmissions containing the same unresolved issues will result in **double the previous deduction**
- You must demonstrate meaningful improvements based on the feedback