SINGAPORE UNIVERSITY
OF SOCIAL SCIENCES

ICT235
Data Structures and Algorithms

Tutor-Marked Assignment

January 2025 Presentation

TUTOR-MARKED ASSIGNMENT (TMA)

This assignment is worth 18% of the final mark for ICT235 – Data Structures and Algorithms

The cut-off date for this assignment is **Sunday, 16 March 2025, 2355 hours.**

Important Note to Students:

You are to include the following particulars in your submission: Course Code, Title of the TMA, SUSS PI No., Your Name, and Submission Date.

---

1. Do NOT import any Python modules otherwise specified. Failure to comply will result in ZERO marks for the question.
2. You must carefully read each question and STRICTLY follow the instructions.
3. You are implementation is evaluated based on various aspects, including but not limited to:
    a. Correctness and efficiency of the implementation
    b. Handling of edge cases and error conditions
    c. Code readability and organization
4. Ensure your class and function names align with specified requirements. This is a must since a set of unit tests will be utilized to evaluate your work.
5. You may identify and implement unspecified properties and functions (e.g., private properties, functions, and any others) required to actualize the requested operations. However, do note that the unit tests will only exploit the classes and functions defined/provided officially.

Answer all questions. (Total 100 marks)

Question 1 (30 marks)

Playlist Manager using Positional List

Background:
Playlists are a fundamental feature of music streaming services, allowing users to manage songs dynamically. In this problem, you will implement a playlist manager using a positional list (a doubly linked list with position abstractions) and support key operations such as adding, moving, removing, shuffling, and undoing shuffles. This problem will test your understanding of linked lists, stacks, and randomization.

To help you get started, we have provided a framework that includes class structures and function definitions. Your task is to complete the missing parts of the implementation.

Constraints and Expectations:

- Efficiency:
  - Each operation should be efficient. For example:
  - Adding, removing, or moving songs should take $O(1)$ or $O(n)$, depending on the operation.
  - Shuffling and undoing should take $O(n)$, where n is the size of the playlist.
- Correctness:
  - Ensure the playlist behaves as expected in all cases.
  - Handle invalid positions gracefully (e.g., raise an error for out-of-bound indices).

Question 1a (15 marks)

Complete the PositionalList class.

- This class is the foundation of the playlist manager.
- Implement a doubly linked list with positional abstractions:
  - Adding elements at the beginning, end, or before/after a specific position.
  - Removing elements at a specific position.
  - Traversing the list efficiently.

Question 1b (15 marks)

- A high-level manager that uses PositionalList to organize and manipulate a playlist of songs.
- Supports key playlist operations such as adding, moving, removing, shuffling, and undoing shuffles.
- Using PositionalList, implement the following in the PlaylistManager class:
  - Add a Song:
    - Add a song at a specific position in the playlist.
    - Ensure the position is valid (e.g., no out-of-bound indices).
  - Move a Song:
    - Move a song from one position to another.
    - Handle edge cases like moving a song to the beginning or end of the playlist.
  - Remove a Song:
    - Remove a song at a given position.
    - Handle invalid positions gracefully.
  - Shuffle:
    - Randomly shuffle the playlist using a reproducible random seed.
    - Save the current playlist order in a stack before shuffling.
    - The shuffle method must support reproducible shuffling using a random seed.
    - **Use Python's random module for shuffling**.

- Undo Shuffle:
  - Restore the playlist to the state before the most recent shuffle using the stack.
  - Ensure multiple undo operations are supported.

Question 2 (20 marks)

Serialize and Deserialize a Binary Tree

Background:
Serialization refers to converting a data structure, such as a binary tree, into a format that can be stored or transmitted easily. Deserialization is the reverse process—reconstructing the data structure from its serialized representation. These processes are essential for applications like saving trees to files, transferring them over networks, or caching their states.

In this problem, you will implement functions to serialize a binary tree into a string and deserialize the string back into the original binary tree structure. This problem will test your understanding of tree traversal algorithms, recursion, and handling edge cases in tree structures. To help you get started, we have provided a framework that includes class definitions and function signatures. Your task is to complete the missing parts of the implementation.
Please note that the TreeNode class has already been completed.

Do not modify the TreeNode class.

Constraints and Expectations:

- Efficiency:
  - Serialization and deserialization should both run in O(n), where n is the number of nodes in the tree.
  - Minimize extra space usage beyond what is necessary for recursion.
- Correctness:
  - The implementation should correctly serialize and deserialize any arbitrary binary tree.
- Edge Case Handling:
  - Handle edge cases, such as:
    - An empty tree.
    - A tree with only one child or deeply unbalanced trees.

Question 2a. (10 marks)

Complete the serialize() function in the Codec class.
- Convert a binary tree into a single string representation.
- Use a preorder traversal to serialize the tree into a string.
- Represent None (null) nodes with the string "null".
- The output string should be compact and suitable for reconstructing the tree.
- Ensure the serialization handles:
  - Fully balanced trees.
  - Unbalanced trees.

- Trees with only left or only right children.

Question 2b (10 marks)

Complete the deserialize() function in the Codec class.

- Convert the serialized string back into the original binary tree.
- Reconstruct the structure of the tree based on the traversal used during serialization.
- Handle all edge cases, including empty trees and unbalanced trees.
- Ensure that deserialize(serialize(tree)) reconstructs the original tree.

Question 3 (50 marks)

Performance of Binary Heap Operations

Background:

A binary heap is a complete binary tree that satisfies one of the following properties:

- Max-Heap: Every parent node is greater than or equal to its children.
- Min-Heap: Every parent node is less than or equal to its children.

Binary heaps are often used to implement priority queues, where elements are inserted with priorities and removed in order of priority.

In this problem, you will:

1. Implement a binary heap that supports both **min-heap** and **max-heap**.
2. Compare the performance of building a heap using two approaches:
   - Repeatedly inserting elements into the heap.
   - Building the heap from an array in one step using a bottom-up approach.

Constraints and Expectations:

- Heap Operations:
  - Heapify:
    - The heapify operation should build the heap using a bottom-up approach.
    - Avoid using repeated push calls for heapify.
- Correctness:
  - Ensure the heap behaves as expected in all cases.
  - Handle invalid operations gracefully (e.g., raising an error when popping from an empty heap).
- Edge Case Handling:
  - The heap should at least handle the following edge cases:
    - Empty arrays: heapify([]) should produce an empty heap.
    - Single-element arrays: Ensure push, pop, and heapify behave correctly.
    - Arrays with duplicate values: Maintain correct behavior with identical priorities.
    - Arrays with sorted values (both ascending and descending).
    - and more.

Question 3a (10 marks)

Implement the BinaryHeap class.

- Use a Python list to represent the heap.
- Write a class BinaryHeap that supports:
  - push(value): Insert a value into the heap.
  - pop(): Remove and return the top element (max or min) of the heap.
  - heapify(array): Build a heap from an unordered array.
- Support both min-heap and max-heap functionality using an optional parameter is_min_heap during initialization.

Question 3b (15 marks)

Conduct Performance Comparisons by implementing measure_performance().

- Generate random arrays of integers for testing.
  - **Use Python's random module to generate random arrays**.
- Compare the time taken to build a heap for both min-heap and max-heap using:
  - Repeatedly calling push() for each element in the array.
  - Using the heapify() method.
- Run your comparisons for arrays of increasing sizes (e.g., 10^4, 10^5, 10^6, or as much as your system can support).
- Measure and compare the time taken to build the heap using both methods for different heap types.
  - **Use Python's time module to measure the elapsed time**.

Question Q3c (10 marks)

Visualize Results by implementing plot_results().

- Plot the time taken for both approaches against the array size.
- Compare results for min-heap and max-heap.
- Plot results using matplotlib.
  - **Use matplotlib module for the plots**.
- You must generate two separate plots
  - First plot:
    - plot title: "Performance Comparison: Min-Heap"
    - y-axis title: Time
    - x-axis title: Array Size
    - Should include two functions (one for repeated push and another for heapify)
  - Second plot:
    - plot title: "Performance Comparison: Max-Heap"
    - y-axis title: Time
    - x-axis title: Array Size

- Should include two functions (one for repeated push and another for heapify)

Question 3d (10 marks)

Analyze Results.

- Analyze the plots you generated by verifying their alignment with theoretical time complexities.
- Answer the below questions using the markdown cell below the framework (labeled as "Q3d Answers").

Question 3d(i)

What are the expected time complexities of the two approaches (repeated push and heapify)? Explain why such performances are expected.

(5 marks)

Question 3d(ii)

Does our expectation (found in Q3d(i)) match with the plots generated (in Q3c)? If not, explain why the actual performance and our expectations do not align well.

(10 marks)

---- END OF ASSIGNMENT ----