Project 1 Evaluation

Jose Medina  
08/10/2024

# Pseudocode and Runtime Analysis

In this evaluation, I will analyze the worst-case running time and memory usage of three different data structures: vector, hash table, and binary search tree (BST). The focus is on the efficiency of reading a file, creating course objects, and storing these objects in the respective data structures. The analysis will also include the Big O notation for each operation.

## Vector Data Structure

Pseudocode Summary:

The vector implementation involves reading each line from a file, splitting it into course data, creating course objects, and storing them in a vector.

Cost per Line of Code:

1. File Reading Loop (`for each line in file`): Cost = 1, executed `n` times.

2. Splitting Line (split line by ","): Cost = O(m), where m is the length of the line, executed n times.

3. Creating Course Object (createCourse(data)): Cost = O(p), where p is the number of prerequisites, executed n times.

4. Appending to Vector (courses.push\_back(course)): Cost = O(1) on average, but O(n) in the worst case due to vector resizing, executed n times.

Total Complexity:

- File Reading: O(n)

- Splitting Lines: O(n \* m)

- Creating Courses: O(n \* p)

- Appending to Vector: O(n) on average, but worst case could be O(n^2) due to resizing.

### Memory Usage:

- Memory usage is O(n), growing linearly with the number of courses and their associated data. Each course object requires storage for its data, and the vector may need additional memory for resizing operations.

## Hash Table Data Structure

Pseudocode Summary:

The hash table pseudocode reads a file, validates the course data, and inserts courses into a hash table, including checks for duplicates and prerequisite validity.

Cost per Line of Code:

1. File Reading Loop (for each line in file): Cost = 1, executed n times.

2. Splitting Line (split(line, ",")): Cost = O(m), executed n times.

3. Validating Data and Creating Course: Cost = O(p) + O(n) for prerequisite checks and insertion, executed n times.

4. Inserting into Hash Table: Average case = O(1), worst case = O(n) due to collisions, executed n times.

Total Complexity:

- File Reading: O(n)

- Splitting Lines: O(n \* m)

- Validating and Creating Courses: O(n^2) in the worst case due to hash collisions and checking for duplicates.

### Memory Usage:

- Memory usage is O(n), accounting for the course objects and the underlying structure of the hash table. The memory required for the hash table also depends on the load factor and resizing operations.

## Binary Search Tree Data Structure

Pseudocode Summary:

The BST pseudocode reads a file, creates course objects, and inserts them into a binary search tree.

Cost per Line of Code:

1. File Reading Loop (for each line in file): Cost = 1, executed n times.

2. Splitting Line (split(line, ",")): Cost = O(m), executed n times.

3. Creating Course Object (createCourse(data)): Cost = O(p), executed n times.

4. Inserting into Tree: Worst case = O(n) if the tree becomes unbalanced, executed n times.

Total Complexity:

- File Reading: O(n)

- Splitting Lines: O(n \* m)

- Creating Courses: O(n \* p)

- Inserting into Tree: O(n^2) in the worst case due to unbalanced tree insertions.

### Memory Usage:

- Memory usage is O(n), considering the course objects and the BST nodes, which might require more memory due to pointers in the tree nodes. The BST may also require additional memory for balancing operations.

## Runtime Analysis Chart

Below is the runtime analysis of different operations for Vector, Hash Table, and Binary Search Tree data structures:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Vector (O) | Hash Table (O) | Binary Search Tree (O) |
| File Reading | O(n) | O(n) | O(n) |
| Splitting Lines | O(n \* m) | O(n \* m) | O(n \* m) |
| Creating Courses | O(n \* p) | O(n \* p) | O(n \* p) |
| Appending/Inserting into Structure | O(n) | O(n^2) | O(n^2) |

## Advantages and Disadvantages

- Vector:

- Advantages: Simple to implement, efficient for append operations when resizing is infrequent.

- Disadvantages: Poor performance for frequent insertions/deletions at arbitrary positions and not ideal for lookups.

- Hash Table:

- Advantages: Efficient average-case performance for lookups and insertions (O(1)), especially suitable for scenarios with frequent lookups.

- Disadvantages: Performance can degrade significantly in the worst case due to collisions, and resizing the hash table can be costly.

- Binary Search Tree:

- Advantages: Maintains sorted data, making it useful for range queries and in-order traversals.

- Disadvantages: Insertions and lookups can degrade to O(n) if the tree becomes unbalanced, requiring additional balancing mechanisms such as AVL or Red-Black trees for efficiency.

## Recommendation

Based on the analysis of the runtime complexity and memory usage, I recommend using the hash table for this project. Hash tables provide efficient average-case performance for lookups and insertions, which aligns well with the expected use cases. While the worst-case scenario is O(n^2) due to potential hash collisions, this is relatively rare with a well-designed hash function. If maintaining sorted data is essential, a balanced binary search tree could be considered, though it adds complexity to the implementation.

This recommendation balances performance considerations with the complexity of implementation, ensuring that the project can handle the expected workload effectively.