CS 300 Project One: Final Submission

Andrew Torrez

Aug 15th, 2024

# Introduction

This document contains the final pseudocode submission for CS 300 Project One. The project involves the creation of pseudocode for managing course information using three different data structures: vector, hash table, and binary search tree. Additionally, a menu system is included for interaction, and a Big O runtime analysis is performed to evaluate the performance of each data structure. The final recommendation for the best data structure to use is provided at the end of this document.

# 1. Vector Pseudocode

FUNCTION openAndReadFile(filename):  
 OPEN file with name filename  
 FOR each line in file:  
 SPLIT line by commas into elements  
 IF length of elements < 2:  
 PRINT 'Error: Each line must have at least a course number and a course title'  
 CONTINUE to next line  
 SET courseNumber to elements[0]  
 SET courseTitle to elements[1]  
 SET prerequisites to elements from index 2 to end  
 FOR each prereq in prerequisites:  
 IF NOT courseExists(prereq):  
 PRINT 'Error: Prerequisite', prereq, 'does not exist as a course'  
 CALL createAndStoreCourse(courseNumber, courseTitle, prerequisites)  
 CLOSE file  
  
FUNCTION createAndStoreCourse(courseNumber, courseTitle, prerequisites):  
 CREATE newCourse of type Course  
 SET newCourse.courseNumber to courseNumber  
 SET newCourse.courseTitle to courseTitle  
 SET newCourse.prerequisites to prerequisites  
 ADD newCourse to vector courses  
  
FUNCTION searchCourse(courseNumber):  
 FOR each course in courses:  
 IF course.courseNumber EQUALS courseNumber:  
 PRINT course details  
 RETURN  
 PRINT 'Course not found'

# 2. Hash Table Pseudocode

STRUCT Course:  
 courseNumber  
 courseTitle  
 prerequisites  
  
SET courses to an empty hash table  
  
FUNCTION createAndStoreCourse(courseNumber, courseTitle, prerequisites):  
 CREATE newCourse of type Course  
 SET newCourse.courseNumber to courseNumber  
 SET newCourse.courseTitle to courseTitle  
 SET newCourse.prerequisites to prerequisites  
 ADD newCourse to courses with key courseNumber  
  
FUNCTION searchCourse(courseNumber):  
 IF courseNumber EXISTS in courses:  
 PRINT course details  
 ELSE:  
 PRINT 'Course not found'

# 3. Binary Search Tree Pseudocode

STRUCT Course:  
 courseNumber  
 courseTitle  
 prerequisites  
 leftChild  
 rightChild  
  
SET root to NULL  
  
FUNCTION createAndStoreCourse(courseNumber, courseTitle, prerequisites):  
 CREATE newCourse of type Course  
 SET newCourse.courseNumber to courseNumber  
 SET newCourse.courseTitle to courseTitle  
 SET newCourse.prerequisites to prerequisites  
 CALL insertIntoTree(root, newCourse)  
  
FUNCTION insertIntoTree(node, newCourse):  
 IF node IS NULL:  
 RETURN newCourse  
 IF newCourse.courseNumber < node.courseNumber:  
 node.leftChild = CALL insertIntoTree(node.leftChild, newCourse)  
 ELSE:  
 node.rightChild = CALL insertIntoTree(node.rightChild, newCourse)  
 RETURN node  
  
FUNCTION searchCourse(node, courseNumber):  
 IF node IS NULL:  
 PRINT 'Course not found'  
 RETURN  
 IF courseNumber < node.courseNumber:  
 CALL searchCourse(node.leftChild, courseNumber)  
 ELSE IF courseNumber > node.courseNumber:  
 CALL searchCourse(node.rightChild, courseNumber)  
 ELSE:  
 PRINT course details

# 4. Menu Pseudocode

FUNCTION displayMenu():  
 PRINT '1. Load data into structure'  
 PRINT '2. Print an alphanumeric ordered list of all courses'  
 PRINT '3. Print course title and prerequisites for a given course'  
 PRINT '9. Exit'  
 GET userInput  
 SWITCH userInput:  
 CASE 1:  
 CALL loadData()  
 CASE 2:  
 CALL printCourseList()  
 CASE 3:  
 GET courseNumber  
 CALL searchCourse(courseNumber)  
 CASE 9:  
 EXIT program

# 5. Sorting Pseudocode

FUNCTION printCourseList():  
 SORT courses by courseNumber in alphanumeric order  
 FOR each course in sorted courses:  
 PRINT course details

# 6. Runtime and Memory Analysis

This section provides an analysis of the runtime and memory usage for each data structure:  
  
1. Vector:  
 - Worst-case runtime for file reading and inserting: O(n)  
 - Searching: O(n)  
 - Memory usage: O(n)  
 - Pros: Simple implementation, good for small datasets.  
 - Cons: Inefficient for large datasets, linear search is slow.  
  
2. Hash Table:  
 - Worst-case runtime for file reading and inserting: O(n)  
 - Searching: O(1) on average, O(n) in worst case  
 - Memory usage: O(n)  
 - Pros: Fast average-case search times.  
 - Cons: Collisions can degrade performance, extra memory for hash table overhead.  
  
3. Binary Search Tree:  
 - Worst-case runtime for file reading and inserting: O(n)  
 - Searching: O(log n) on average, O(n) in worst case  
 - Memory usage: O(n)  
 - Pros: Efficient search in balanced trees, ordered structure.  
 - Cons: Performance can degrade with unbalanced trees.

# 7. Recommendation

Based on the analysis, the hash table offers the best overall performance for this task. It provides the fastest average-case search times, which is crucial for efficiently accessing course information. While the binary search tree is more efficient in maintaining order and offers good performance in balanced trees, the risk of unbalanced trees makes the hash table a safer choice for consistent performance. Therefore, the hash table is recommended as the best data structure for the program.

- Searching: O(n)  
 - Memory usage: O(n)  
 - Pros: Simple implementation, good for small datasets.  
 - Cons: Inefficient for large datasets, linear search is slow, and insertion operations take time proportional to the size of the dataset.  
  
2. Hash Table:  
 - Worst-case runtime for file reading and inserting: O(n)  
 - Searching: O(1) on average, O(n) in the worst case  
 - Memory usage: O(n)  
 - Pros: Fast average-case search times due to O(1) lookup time for hashed keys.  
 - Cons: Collisions can degrade performance, requiring strategies like chaining or open addressing, which consume additional memory.  
  
3. Binary Search Tree:  
 - Worst-case runtime for file reading and inserting: O(n)  
 - Searching: O(log n) on average, O(n) in the worst case when the tree is unbalanced  
 - Memory usage: O(n)  
 - Pros: Efficient search in balanced trees, which provide O(log n) time complexity for most operations. Maintains order, making it easy to retrieve sorted data.  
 - Cons: Performance can degrade with unbalanced trees, making operations as slow as O(n). Implementing self-balancing mechanisms like AVL or Red-Black trees adds complexity.

# 7. Enhanced Recommendation

After careful consideration of the runtime and memory trade-offs, the hash table is recommended as the optimal data structure for this project. Its O(1) average-case time complexity for lookups and insertions provides superior performance for managing course data, especially when dealing with a large dataset. While binary search trees offer a more structured and ordered approach, the risk of unbalanced trees makes them less reliable without added complexity. Vectors, while simple to implement, suffer from poor performance in searching and insertion tasks for larger datasets.   
For an advising system where quick lookups are critical and data is frequently accessed, the hash table stands out as the best choice. To mitigate the potential drawbacks of collisions, using a larger table size and a good hash function will ensure consistent performance.