6-2 Submit Project One

Duane Wegner

Southern New Hampshire University

CS-300 DSA: Analysis and Design

Prof. Jack Lusby (MS CS)

October 7, 2024

**Pseudocode**

**Milestone One: Vector:**

**Open File, Read Data, Parse Lines, and Check Formatting Errors**

FUNCTION loadCoursesFromFile(filePath)

OPEN file at filePath

IF file is not open THEN

DISPLAY "Error opening file."

RETURN

END IF

WHILE NOT end of file DO

READ line from file

IF line is empty THEN

CONTINUE

END IF

SPLIT line into components based on comma

IF components.length NOT EQUALS expected number THEN

DISPLAY "Formatting error in line: " + line

CONTINUE

END IF

CREATE Course object using components

ADD Course object to courses vector

END WHILE

CLOSE file

END FUNCTION

**Create Course Objects**

FUNCTION createCourse(components)

courseId = components[0]

courseTitle = components[1]

prerequisites = components[2] // Optional; handle as needed

RETURN new Course(courseId, courseTitle, prerequisites)

END FUNCTION

**Print Course Information and Prerequisites**

FUNCTION printCourseInfo(course)

DISPLAY "Course ID: " + course.courseId

DISPLAY "Course Title: " + course.courseTitle

IF course.prerequisites IS NOT NULL THEN

DISPLAY "Prerequisites: " + course.prerequisites

ELSE

DISPLAY "No prerequisites."

END IF

END FUNCTION

**Menu Options**

FUNCTION displayMenu()

DISPLAY "1. Load courses from file"

DISPLAY "2. Print alphanumerically ordered list of courses"

DISPLAY "3. Print course title and prerequisites"

DISPLAY "9. Exit"

END FUNCTION

**Milestone Two: Hash Table:**

**Open File, Read Data, Parse Lines, and Check Formatting Errors**

FUNCTION loadCoursesFromFile(filePath)

OPEN file at filePath

IF file is not open THEN

DISPLAY "Error opening file."

RETURN

END IF

WHILE NOT end of file DO

READ line from file

IF line is empty THEN

CONTINUE

END IF

SPLIT line into components based on comma

IF components.length NOT EQUALS expected number THEN

DISPLAY "Formatting error in line: " + line

CONTINUE

END IF

course = createCourse(components)

INSERT course into hashTable with course.courseId as key

END WHILE

CLOSE file

END FUNCTION

**Create Course Objects**

FUNCTION createCourse(components)

courseId = components[0]

courseTitle = components[1]

prerequisites = components[2] // Optional; handle as needed

RETURN new Course(courseId, courseTitle, prerequisites)

END FUNCTION

**Print Course Information and Prerequisites**

FUNCTION printCourseInfo(course)

DISPLAY "Course ID: " + course.courseId

DISPLAY "Course Title: " + course.courseTitle

IF course.prerequisites IS NOT NULL THEN

DISPLAY "Prerequisites: " + course.prerequisites

ELSE

DISPLAY "No prerequisites."

END IF

END FUNCTION

**Menu Options**

FUNCTION displayMenu()

DISPLAY "1. Load courses from file"

DISPLAY "2. Print alphanumerically ordered list of courses"

DISPLAY "3. Print course title and prerequisites"

DISPLAY "9. Exit"

END FUNCTION

**Milestone Three: Binary Search Tree:**

**Open File, Read Data, Parse Lines, and Check Formatting Errors**

FUNCTION loadCoursesFromFile(filePath)

OPEN file at filePath

IF file is not open THEN

DISPLAY "Error opening file."

RETURN

END IF

WHILE NOT end of file DO

READ line from file

IF line is empty THEN

CONTINUE

END IF

SPLIT line into components based on comma

IF components.length NOT EQUALS expected number THEN

DISPLAY "Formatting error in line: " + line

CONTINUE

END IF

course = createCourse(components)

INSERT course into binarySearchTree with course.courseId

END WHILE

CLOSE file

END FUNCTION

**Create Course Objects**

FUNCTION createCourse(components)

courseId = components[0]

courseTitle = components[1]

prerequisites = components[2] // Optional; handle as needed

RETURN new Course(courseId, courseTitle, prerequisites)

END FUNCTION

**Print Course Information and Prerequisites**

FUNCTION printCourseInfo(course)

DISPLAY "Course ID: " + course.courseId

DISPLAY "Course Title: " + course.courseTitle

IF course.prerequisites IS NOT NULL THEN

DISPLAY "Prerequisites: " + course.prerequisites

ELSE

DISPLAY "No prerequisites."

END IF

END FUNCTION

**Menu Options**

FUNCTION displayMenu()

DISPLAY "1. Load courses from file"

DISPLAY "2. Print alphanumerically ordered list of courses"

DISPLAY "3. Print course title and prerequisites"

DISPLAY "9. Exit"

END FUNCTION

Additional Pseudocode for Sorting and Displaying Course Information

**Sort and Print Courses**

FUNCTION printSortedCourses(courses)

SORT courses by alphanumeric courseId

FOR EACH course IN courses DO

printCourseInfo(course)

END FOR

END FUNCTION

**Evaluations**

To evaluate the run time and memory of different data structures that can be used for the course management program, we will focus on three data structures: vectors, hash tables, and binary search trees (BSTs). This evaluation will consider the pseudocode from previous assignments regarding file operations, parsing lines, and creating course objects.

**Vector**

**Worst-Case Time Complexity:**

**Opening the File**: O(1) (constant time, since this operation does not depend on the number of courses).

**Reading the File**: O(n), where n is the number of courses. Each line in the file must be read individually.

**Parsing Each Line**: O(n), as each line needs to be parsed to create course objects. The parsing operation involves reading and splitting strings, which can be done in linear time.

**Creating Course Objects:** O(n) because a new object is created for each course, and this requires iterating through all n lines in the file.

**Total Time Complexity for Vector**:

O(n) for reading and parsing lines + O(n) for creating course objects = O(n).

**Memory Usage:**

Memory is allocated for storing n course objects, each consuming a fixed amount of space based on its attributes. Hence, the memory complexity is O(n).

**Hash Table**

**Worst-Case Time Complexity:**

Opening the File: O(1).

Reading the File: O(n).

Parsing Each Line: O(n).

**Creating Course Objects and Inserting into Hash Table:**

Inserting into a hash table is O(1) on average, but in the worst case, it can degrade to O(n).

However, we will assume a good hash function and uniform distribution, leading to an average-case of O(1) for each insertion, so for n courses, this becomes O(n).

Total Time Complexity for Hash Table:

O(n) for reading and parsing lines + O(n) for creating course objects + O(n) for inserting into the hash table = O(n).

**Memory Usage:**

Memory is required for the hash table itself, which will require storage for n course objects plus some overhead for the hash table’s structure. The memory complexity remains O(n).

**Binary Search Tree (BST)**

**Worst-Case Time Complexity:**

**Opening the File:** O(1).

**Reading the File:** O(n).

**Parsing Each Line:** O(n).

**Creating Course Objects and Inserting into BST:**

Inserting into a BST takes O(h) time, where h is the height of the tree. In the worst case (when the tree is unbalanced), h can be O(n), leading to a total of O(n) for inserting all n course objects.

If balanced, the height h would be O(log n), but for worst-case analysis, we will consider O(n).

**Total Time Complexity for Binary Search Tree:**

O(n) for reading and parsing lines + O(n) for creating course objects + O(n) for inserting into the BST = O(n).

**Memory Usage:**

Like the other data structures, the BST will also require O(n) memory for storing n course objects along with pointers for each node.

Summary of Evaluation

**Vector:**

**Time Complexity:** O(n)

**Memory Complexity:** O(n)

**Hash Table:**

**Time Complexity**: O(n)

**Memory Complexity:** O(n)

**Binary Search Tree (BST):**

**Time Complexity:** O(n)

**Memory Complexity:** O(n)

**Advantages and Disadvantages**

When selecting a data structure for a course management program, it's essential to consider the unique characteristics of vectors, hash tables, and binary search trees (BSTs). Each structure has its advantages and disadvantages that can impact performance, memory usage, and ease of implementation based on the advisor's requirements.

**Vector**

**Advantages:**

* **Simplicity**: Vectors are straightforward to implement and use. They provide dynamic resizing, which means they can grow and shrink in size as needed.
* **Contiguous Memory**: Vectors store data in contiguous memory locations, leading to better cache performance and faster access times, especially for sequential access.
* **Random Access**: Accessing elements by index is O(1), allowing for quick retrieval of course information if the index is known.

**Disadvantages:**

* **Insertion/Deletion Performance**: Inserting or deleting elements (especially in the middle of the vector) requires shifting elements, resulting in O(n) time complexity for these operations.
* **Memory Overhead**: While vectors dynamically resize, they may occasionally allocate more memory than necessary, which can lead to wasted space.

**Hash Table**

**Advantages:**

* **Fast Lookups**: Hash tables offer average-case O(1) time complexity for lookups, making it very efficient to access course information if you know the course identifier.
* **Dynamic Size**: Hash tables can grow dynamically, allowing them to manage varying amounts of data without a fixed size constraint.
* **Manages Sparse Data**: Ideal for scenarios with a wide variety of course identifiers, where not every identifier is used.

**Disadvantages:**

* **Collisions**: In the worst-case scenario, hash collisions can occur, leading to degraded performance (up to O(n) in the worst case). A poorly chosen hash function can exacerbate this issue.
* **Memory Overhead**: Hash tables often require extra memory for the underlying array and handling collisions (linked lists or open addressing).
* **Complexity**: The implementation of a hash table can be more complex than a simple vector or BST, particularly regarding the design of an effective hash function.

**Binary Search Tree (BST)**

**Advantages:**

* **Ordered Data**: BSTs maintain a sorted order, making it easy to retrieve all courses in an alphanumeric order. This feature is advantageous when frequent sorted access is needed.
* **Dynamic Size**: Like hash tables, BSTs can grow and shrink dynamically as elements are added or removed.
* **Efficient Range Queries**: BSTs can efficiently support range queries (e.g., finding all courses with identifiers between two values).

**Disadvantages:**

* **Height Imbalance**: In the worst-case scenario (e.g., inserting sorted data into an unbalanced BST), the height can become O(n), leading to O(n) time complexity for operations like search, insert, and delete.
* **Memory Overhead**: Each node in a BST requires additional memory for pointers to child nodes, which can increase memory usage compared to a vector.

**Summary**

In summary, the choice of data structure depends on the specific requirements of the advising program:

* **Vectors** are suitable for scenarios where simple, sequential access is needed, but their performance can degrade for insertion and deletion.
* **Hash Tables** excel in quick access and are beneficial for large datasets with unique identifiers but require careful management of collisions and may have memory overhead.
* **Binary Search Trees** provide ordered access and an efficient range of queries but can suffer from performance issues if not balanced.

**Recommendation Based on Analysis**

Based on the analysis of vectors, hash tables, and binary search trees (BSTs), I recommend using a **hash table** for the course management program. This recommendation is supported by several key factors related to performance, memory efficiency, and the specific requirements outlined in the advisor's guidelines. Hash tables provide average-case O(1) time complexity for lookups, making it incredibly efficient to retrieve course information using unique identifiers, such as course codes. This speed is crucial for a program that needs to access and display course data quickly, especially as the dataset grows. Additionally, hash tables dynamically resize as needed, allowing for flexibility in handling varying amounts of data without significant performance penalties, unlike vectors, which can suffer from performance degradation during insertions and deletions. Moreover, given that course identifiers can vary widely and that not every identifier will be used, hash tables efficiently manage sparse datasets, ensuring that memory is utilized effectively without allocating space for unused identifiers, which is an advantage over both vectors and BSTs.

As the number of courses increases, the hash table can continue to perform efficiently. In scenarios where new courses are frequently added or existing courses are updated, the hash table's performance remains consistent. While hash tables may have some disadvantages, such as potential collisions that can lead to O(n) performance in the worst case, implementing a well-designed hash function can significantly mitigate these issues. Additionally, handling collisions with methods like chaining or open addressing can keep operations efficient. In contrast, vectors are not ideal due to their O(n) time complexity for insertion and deletion, which could become a bottleneck when managing a dynamic dataset like courses. Their reliance on contiguous memory can also lead to inefficiencies in certain scenarios. Binary search trees, while beneficial for maintaining sorted order, can degrade to O(n) for various operations if the tree becomes unbalanced, especially when inserting sorted data, which is not acceptable for the performance requirements of the course management system.

In conclusion, the hash table strikes a balance between performance and flexibility, making it the most suitable choice for the course management program. Its efficiency in handling large datasets, coupled with fast access times and dynamic resizing capabilities, aligns well with the project's needs. If a robust hash function is implemented to manage collisions effectively, a hash table will provide the best performance for the operations required in this program.