Module Six: Project One Data Structures

Lyric Hart

Southern New Hampshire University

CS-300-10910-Analysis and Design

Mr. Michael Rissover

October 10, 2024

Abstract

This project explores the design and evaluation of three data structures-vector, hash table, and binary search tree (BST)-to manage and retrieve course information for academic advisors at ABC university (ABCU). The primary goals of the program are to provide an efficient way to store course data, print courses in alphanumeric order, and retrieve course details and their prerequisites for specific courses. Pseudocode was developed to handle the process of reading and parsing course data, create course objects, and implementing a user menu to to load, search, and display course information. In addition, the project analyzes the time and space complexities of each structure. This is done by looking at their runtime performance for common operations such as, insertion, sorting, and searching. From the analysis, a recommendation will then be made for the optimal data structure to be implemented to help balance efficiency and memory usage. The results of this study will help guide the coding phase of the project, so that the chosen data structure will meet the needs and requirements of the advisors.

Vector Data Structure Pseudocode

Course Structure:

- Define a course with:
    - Course number (alphanumeric).
    - Course name.
    - List of prerequisites.

Search for a Course by Number:

- Input: List of courses, course number.
- For each course in the list:
    - If course number matches the input, return the course.
- If no match is found, return "Course not found."

Open and Read File:

- Input: File name.
- Initialize an empty list to store file contents.
- Open the file:
    - If file cannot be opened, display an error message.
    - Otherwise, for each line in the file:
        - Add the line to the list of file contents.
- Return the list of file contents.

Create Course Schedule:

- Input: List of file contents.
- Initialize an empty list to store course objects.
- For each line in the file contents:
    - Split the line into tokens (e.g., course number, course name, prerequisites).
    - Create a new course object:
        - Set the course number and course name.
        - Add valid prerequisites (ensure they match existing courses).
    - If there are formatting errors (e.g., missing course number or name), display an error message.
    - Add the course to the list of courses.
- Return the list of courses.

Print Course Information and Prerequisites:

- Input: List of courses, course number.
- Search for the course using the course number.
    - If course is found:
        - Display the course number and course name.
        - If prerequisites exist, display them.
        - Otherwise, state that there are no prerequisites.
    - If course is not found, display an error message.

Hash Table Pseudocode

Open and Read File:

- Open the file containing course data.
- If the file cannot be opened:
    - Display an error message and exit the program.
- Otherwise, for each line in the file:
    - Split the line by commas to extract the course number, course title, and prerequisites.
    - If the line contains fewer than two parameters:
        - Display an error message and skip to the next line.
    - For each prerequisite in the line:
        - If the prerequisite does not exist in the file, display an error message.

Initialize Hash Table:

- Initialize an empty hash table to store courses.
- For each valid line from the file:
    - Create a course object containing the course number, course title, and prerequisites.
    - Use a hash function to generate an index based on the course number.
    - **Collision Handling**:
        - If the index is already occupied, use a collision resolution strategy (e.g., chaining or open addressing).
    - Insert the course object into the hash table using the computed index.

Print Course Information:

- For each course stored in the hash table:
    - Print the course number and course title.
    - If the course has prerequisites, print them as well.

End Program:

- Once all courses have been processed and printed, terminate the program.

Binary Search Tree Pseudocode

Define Classes:

- **BinarySearchTree Class**:
    - Define methods for:
        - insertNode(course) - to insert course objects into the tree.
        - inOrderTraversal() - to output the courses in sorted order (based on courseId).
        - removeNode(course) - to remove course objects.
        - **Destructor**: Use a recursive function to delete all nodes in the tree.
- **Course Class**:
    - Attributes: courseId, courseTitle, prerequisites (list of courseIDs).

Open and Validate File:

- **OpenFile(fileName)**:
    - Open the file with the specified filename.
    - For each line in the file:
        - Split the line by commas into courseId, courseTitle, and prerequisites.
        - Call ValidateLine to ensure the data is in the correct format.
    - Close the file.
- **ValidateLine(courseId, courseTitle, prerequisites)**:
    - If courseId or courseTitle is empty, display an error message.
    - For each prerequisite in the list:
        - If the prerequisite is not found in the course list, display an error message.

    - Return true if the line is valid, otherwise return false.

Create Course Object:

- **CreateCourseObject(courseId, courseTitle, prerequisites)**:
    - Create and return a new course object with courseId, courseTitle, and prerequisites.

Insert Course into Binary Search Tree:

- **InsertCourseIntoBST(course)**:
    - Call the insertNode method of the BinarySearchTree to add the course to the tree.
    - Ensure the insertion logic handles the comparison of courseId to maintain the correct BST structure.

Load Courses into Binary Search Tree:

- **LoadCourses(filename)**:
    - Open the file and read it line by line.
    - For each valid line:
        - Create a course object using CreateCourseObject.
        - Insert the course into the BST using InsertCourseIntoBST.

Print Course Information:

- **PrintCourseInformation(tree)**:
    - Call the inOrderTraversal method of the BinarySearchTree to output course data in sorted order.

    - For each course, print the courseId, courseTitle, and any prerequisites.

Main Program:

- Call LoadCourses('courseData.csv') to load course data into the BST.
- Call PrintCourseInformation(tree) to display the course information.

End Program.

Menu Pseudocode

Main Menu Function:

- **DisplayMenu()**:
    - Print the following options:
        1. Load Courses from File
        2. Search for a Course by ID
        3. Print All Courses
        4. Exit Program
    - Prompt the user to enter a choice (1-4).
    - Call the corresponding function based on the user's choice.

Load Courses Function:

- **LoadCourses(filename)**:
    - Prompt the user to enter the filename.
    - Call the respective data structure's load function (e.g., LoadCoursesIntoVector, LoadCoursesIntoHashTable, or LoadCoursesIntoBST).
    - Display a message confirming that the courses were successfully loaded.

Search for a Course Function:

- **SearchCourse()**:
    - Prompt the user to enter the courseId.
    - Call the respective data structure's search function (e.g., SearchInVector, SearchInHashTable, or SearchInBST).

    - If the course is found:
        - Display the courseId, courseTitle, and prerequisites.
    - If the course is not found:
        - Print an error message: "Course not found."

Print All Courses Function:

- **PrintAllCourses()**:
    - Call the respective data structure's print function (e.g., PrintCoursesFromVector, PrintCoursesFromHashTable, or PrintCoursesFromBST).
    - For each course, print the courseId, courseTitle, and any prerequisites.

Exit Program Function:

- **ExitProgram()**:
    - Print a message: "Exiting the program."
    - End the program.

Main Program:

- **Main()**:
    - While the user does not choose to exit:
        - Call DisplayMenu.
        - Handle invalid input (i.e., if the user enters an option outside 1-4, display an error message and prompt again).

Print Course Information and Prerequisites Pseudocode

1. **Print Course Information Function**:
    - **PrintCourseInformation(courseId)**:
        - Call the appropriate SearchCourse function (e.g., SearchInVector, SearchInHashTable, or SearchInBST) to retrieve the course by courseId.
        - If the course is found:
            - Print the courseId.
            - Print the courseTitle.
            - If the course has prerequisites:
                - Print "Prerequisites:".
                - For each prerequisite in course.PreReqs:
                    - Print the prerequisite.
            - If the course does not have prerequisites:
                - Print "No prerequisites for this course."
        - If the course is not found:
            - Print "Error: Course not found."
2. **Search for Course Function** (for data structures):
    - **SearchCourse(courseId)**:
        - **For Vector**:
            - Loop through each course in the vector.
            - If the courseId matches, return the course.
            - If not found, return a message "Course not found".

        - **For Hash Table**:
            - Use the hash function to find the course by courseId.
            - If found, return the course.
            - If not, return "Course not found."
        - **For Binary Search Tree**:
            - Traverse the BST to find the course by courseId.
            - If found, return the course.
            - If not, return "Course not found."

3. **Print All Courses Function** (Optional for listing multiple courses):
    - **PrintAllCourses()**:
        - Loop through the data structure (vector, hash table, or binary search tree).
        - For each course, print:
            - courseId
            - courseTitle
            - If the course has prerequisites, print them.
            - If no prerequisites, print "No prerequisites for this course."

Data Structure Runtime and Memory

Vector Data Structure: The vector data structure is an efficient choice for sequential operations. When it comes to the time complexity of reading, parsing, and storing data this would come up to O(n), where n is the number of courses. All of these operations are executed in linear time. Although, the issue lies in the performance when it's searching data. This comes at the cost of time, which is why it's O(n), because it requires an sequential operation. It is quick for appending and iterating through data cycles, however when it comes to searching, it's not an ideal option where quick loops are present. Lastly, when it comes to memory usage, the time complexity is O(n), and resizing it can impact performance, if the vector's capacity needs to be expanded.

Hash Table Structure: The hash table method excels in terms of search efficiency, offering an average time complexity of O(1) for inserting and searching courses. Reading the file, parsing lines, and creating course objects remain O(n), and inserting them hash table averages O(1) per course. This results in an overall time complexity of O(n). Albeit, if too many collisions occur in this stage, then it can affect the performance. But, having a well-developed hash function can reduce the probability of collision. The primary disadvantage of hash tables is that they do not preserve the order of elements, therefore making ordered traversals non-dependent. They also introduce memory overhead for maintaining the table, but this is known by their rapid search and insertion capabilities; especially in abundant datasets.

Binary Search Tree Structure: The BST method offers a mixed balanced between search efficiency and data ordering. This comes at an average time complexity of O(log n) for both insertion and searching operations. It makes the BST a likely candidate for ordering data. When it comes to file reading and parsing data, the time complexity remains O(n). But, in terms of insertion, it becomes a logarithmic function resulting in O(n log n). The only downside is its worst-case performance, which can become O(n), if the tree is unbalanced. The BST itself is more memory efficient than a hash table, but unbalanced trees can be an issue for search time operations.

Conclusion: To conclude, the best recommendation would be to utilize the hash table, because it has efficient searching and course retrieval. It provides the fastest search time of O(1), and with a robust hash function, it minimizes the risk of collisions. Despite it lacking the ability to maintain sorted data, it has superior performance ability for rapid search capabilities, and manageable memory overhead. This makes it the best option for the project. However, the vector and BST data structures are less ideal, regarding their search functions and potential complexities.