**Dat Nguyen**

**CS-300**

**Module 6**

**June 16, 2024**

**David Ostrowski**

**Pseudocode**

**Vector**

**1. Vector Pseudocode to Open the File, Read Data, and Check for Format Errors:**

|  |
| --- |
| // Function to load courses from a file  Function LoadCoursesFromFile(filename):  Open file with the given filename for reading  Initialize an empty vector called courses  Initialize an empty set called courseNumbers    For each line in the file:  Split the line by commas into tokens  If the number of tokens is less than 2:  Print "Error: Invalid line format"  Continue to the next line    Extract course number and course title from the tokens  Add course number to the courseNumbers set  Initialize an empty vector called prerequisites  For each token from the third token onward:  Add the token to the prerequisites vector    Create a course object with the course number, title, and prerequisites  Add the course object to the courses vector    For each course in courses:  For each prerequisite in the course's prerequisites:  If the prerequisite is not in courseNumbers:  Print "Error: Missing prerequisite " + prerequisite    Close the file  Return courses |

**2. Vector Pseudocode to Create and Store Course Objects:**

|  |
| --- |
| // Course structure  Struct Course:  courseNumber  courseTitle  prerequisites (a vector of strings)    // Function to create course objects and store in a vector  Function CreateCourseObjectsAndStore(filename):  courses = LoadCoursesFromFile(filename)  Initialize an empty vector called courseVector    For each course in courses:  Add the course to courseVector    Return courseVector |

**3. Vector Pseudocode to Search for a Course and Print Course Information and Prerequisites:**

|  |
| --- |
| // Function to search for a course and print its information  Function SearchCourse(courseVector, courseNumber):  For each course in courseVector:  If the course's course number is equal to the given courseNumber:  Print "Course Number:", course.courseNumber  Print "Course Title:", course.courseTitle  Print "Prerequisites:"  For each prerequisite in course.prerequisites:  Print prerequisite  Return  Print "Course not found" |

**4. Vector Pseudocode for a Menu System:**

|  |
| --- |
| // Function to display the menu and perform actions based on user input  Function DisplayMenu():  While true:  Print "1. Load Courses"  Print "2. Print All Courses"  Print "3. Print Course Information"  Print "9. Exit"  choice = Get user input  If choice is 1:  courses = CreateCourseObjectsAndStore("courses.txt")  Print "Courses loaded."  Else If choice is 2:  PrintAllCourses(courses)  Else If choice is 3:  courseNumber = Get user input for course number  SearchCourse(courses, courseNumber)  Else If choice is 9:  Print "Exiting."  Exit |

**5. Vector Pseudocode to print all courses in alphanumeric order**

|  |
| --- |
| // Function to print all courses in alphanumeric order  Function PrintAllCourses(courseVector):  Sort courseVector by courseNumber  For each course in courseVector:  Print "Course Number:", course.courseNumber, "Course Title:", course.courseTitle |

**Hash Table**

**1. Hash Table Pseudocode to Open the File, Read Data, and Check for Format Errors:**

|  |
| --- |
| // Function to open the file, read the data, and validate the file format  Function loadCourseData(fileName):  Open file fileName  If file is not opened successfully:  Print "Error: File could not be opened."  Return  Initialize an empty list courseList  While not end of file:  Read line from file  Split line by comma into tokens  If number of tokens < 2:  Print "Error: Invalid format - less than 2 parameters on a line."  Return  courseNumber = tokens[0]  courseName = tokens[1]  prerequisites = tokens[2 to end]  For prerequisite in prerequisites:  If not courseExists(prerequisite, courseList):  Print "Error: Prerequisite", prerequisite, "does not exist in the course list."  Return  Create a new Course object with courseNumber, courseName, and prerequisites  Add Course object to courseList  Close file  Return courseList    // Helper function to check if a course exists in the course list  Function courseExists(courseNumber, courseList):  For course in courseList:  If course.courseNumber == courseNumber:  Return true  Return false |

**2. Hash Table Pseudocode for Creating and Storing Course Objects:**

|  |
| --- |
| // Course structure  Struct Course:  courseNumber  courseName  prerequisites  // Function to create a hash table and store course objects in it  Function createHashTable(courseList):  Initialize an empty hash table courseTable  For course in courseList:  courseTable[course.courseNumber] = course  Return courseTable |

**3. Hash Table Pseudocode for a Menu System:**

|  |
| --- |
| // Function to display the menu and perform actions based on user input  Function DisplayMenu():  While true:  Print "1. Load Courses"  Print "2. Print All Courses"  Print "3. Print Course Information"  Print "9. Exit"  choice = Get user input  If choice is 1:  courseList = loadCourseData("courses.txt")  courseTable = createHashTable(courseList)  Print "Courses loaded."  Else If choice is 2:  PrintAllCourses(courseTable)  Else If choice is 3:  courseNumber = Get user input for course number  SearchCourse(courseTable, courseNumber)  Else If choice is 9:  Print "Exiting."  Exit |

**4. Hash Table Pseudocode to print all courses in alphanumeric order**

|  |
| --- |
| // Function to print all courses in alphanumeric order  Function PrintAllCourses(courseTable):  sortedKeys = sort(keys(courseTable))  For each courseNumber in sortedKeys:  course = courseTable[courseNumber]  Print "Course Number:", course.courseNumber, "Course Title:", course.courseName, "Prerequisites:", join(course.prerequisites, ", ") |

**5. Hash Table Pseudocode to search for a course and print its information**

|  |
| --- |
| // Function to search for a course and print its information  Function SearchCourse(courseTable, courseNumber):  If courseNumber in courseTable:  course = courseTable[courseNumber]  Print "Course Number:", course.courseNumber  Print "Course Title:", course.courseName  Print "Prerequisites:"  For each prerequisite in course.prerequisites:  Print prerequisite  Else:  Print "Course not found." |

**Binary Search Tree (BST)**

**1. BST Pseudocode to Open the File, Read Data, and Check for Format Errors:**

|  |
| --- |
| // Function to open the file, read data, and check for format errors  Function loadCourseData(fileName):  Open file fileName  If file is not opened successfully:  Print "Error: File could not be opened."  Return  Initialize an empty list courseList  While not end of file:  Read line from file  courseData = parseLine(line)  If courseData is not valid:  Print "Error: Invalid file format."  Return  Add courseData to courseList  Return courseList    Function parseLine(line):  Split line by commas into courseInfo  If length of courseInfo < 2:  Return "Error: Each line must contain at least a course number and course name."  courseNumber = courseInfo[0]  courseName = courseInfo[1]  prerequisites = courseInfo[2:]  Return (courseNumber, courseName, prerequisites)    Function validatePrerequisites(courseList):  Initialize empty set of courseNumbers  For course in courseList:  Add course number to courseNumbers set  For course in courseList:  For prerequisite in course.prerequisites:  If prerequisite not in courseNumbers:  Print "Error: Prerequisite " + prerequisite + " does not exist."  Return false  Return true |

**2. BST Pseudocode for Creating and Storing Course Objects:**

|  |
| --- |
| // Course structure  Class Course:  attribute courseNumber  attribute courseName  attribute prerequisites    Function createCourse(courseNumber, courseName, prerequisites):  course = new Course()  course.courseNumber = courseNumber  course.courseName = courseName  course.prerequisites = prerequisites  Return course    // TreeNode structure  Class TreeNode:  attribute course  attribute leftChild  attribute rightChild    Function insertCourse(root, course):  If root is null:  Return new TreeNode(course)  If course.courseNumber < root.course.courseNumber:  root.leftChild = insertCourse(root.leftChild, course)  Else if course.courseNumber > root.course.courseNumber:  root.rightChild = insertCourse(root.rightChild, course)  Return root    Function buildCourseTree(courseList):  Initialize root as null  For courseData in courseList:  course = createCourse(courseData.courseNumber, courseData.courseName, courseData.prerequisites)  root = insertCourse(root, course)  Return root |

**3. BST Pseudocode for Printing Course Information and Prerequisites:**

|  |
| --- |
| // Function to print course information  Function printCourseInfo(course):  Print "Course Number: " + course.courseNumber  Print "Course Name: " + course.courseName  If course.prerequisites is empty:  Print "No prerequisites."  Else:  Print "Prerequisites: "  For prerequisite in course.prerequisites:  Print prerequisite |

**4. BST Pseudocode for a Menu System:**

|  |
| --- |
| // Function to display the menu and perform actions based on user input  Function DisplayMenu():  While true:  Print "1. Load Courses"  Print "2. Print All Courses"  Print "3. Print Course Information"  Print "9. Exit"  choice = Get user input  If choice is 1:  courseList = loadCourseData("courses.txt")  courseTree = buildCourseTree(courseList)  Print "Courses loaded."  Else If choice is 2:  inorderTraversal(courseTree)  Else If choice is 3:  courseNumber = Get user input for course number  searchCourse(courseTree, courseNumber)  Else If choice is 9:  Print "Exiting."  Exit |

**5. BST Pseudocode to print all courses in alphanumeric order:**

|  |
| --- |
| Function inorderTraversal(node):  If node is not null:  inorderTraversal(node.leftChild)  printCourseInfo(node.course)  inorderTraversal(node.rightChild) |

**6. BST Pseudocode to search for a course and print its information:**

|  |
| --- |
| Function searchCourse(treeRoot, courseNumber):  If treeRoot is null:  Print "Course not found."  Return  If courseNumber < treeRoot.course.courseNumber:  searchCourse(treeRoot.leftChild, courseNumber)  Else if courseNumber > treeRoot.course.courseNumber:  searchCourse(treeRoot.rightChild, courseNumber)  Else:  printCourseInfo(treeRoot.course) |

**Analysis**

**Big O Analysis for Vector:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Function** | **Operation** | **Line Cost** | **# Times Executes** | **Total Cost** | **Big O Notation** |
| LoadCoursesFromFile | Opening file | 1 | 1 | 1 | O(1) |
| Initializing vector and set | 1 | 2 | 2 | O(1) |
| Iterating through lines | 1 | n | n | O(n) |
| Splitting line by commas | 1 | n \* m | nm | O(nm) |
| Checking and adding details | 1 | n \* m | nm | O(nm) |
| Creating course object | 1 | n | n | O(n) |
| Nested loop (prerequisites) | 1 | n^2 | n^2 | O(n^2) |
| CreateCourseObjectsAndStore | Calling LoadCoursesFromFile | O(n^2) | 1 | O(n^2) | O(n^2) |
| Iterating through courses | 1 | n | n | O(n) |
| Adding to vector | 1 | n | n | O(n) |
| SearchCourse | Iterating through vector | 1 | n | n | O(n) |
| Printing course details | 1 | n | n | O(n) |
| **Total** | | | | 4n^2 + 2nm + 2n + 3 | O(n^2) |

**Big O Analysis for Hash Table:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Function** | **Operation** | **Line Cost** | **# Times Executes** | **Total Cost** | **Big O Notation** |
| LoadCoursesFromFile | (Same as Vector) | | | O(n^2) | O(n^2) |
| CreateHashTable | Creating hash table | 1 | 1 | 1 | O(1) |
| Iterating through courses | 1 | n | n | O(n) |
| Adding to hash table | 1 | n | n | O(n) |
| SearchCourse | Checking course in hash table | 1 | 1 | 1 | O(1) |
| Printing course details | 1 | 1 | 1 | O(1) |
| **Total** | | | | 2n + 2 | O(n) |

**Big O Analysis for Binary Search Tree (BST):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Function** | **Operation** | **Line Cost** | **# Times Executes** | **Total Cost** | **Big O Notation** |
| LoadCoursesFromFile | (Same as Vector) | | | O(n^2) | O(n^2) |
| BuildCourseTree | Initializing root | 1 | 1 | 1 | O(1) |
| Iterating through courses | 1 | n | n | O(n) |
| Inserting into BST | log n | n | n log n | O(n log n) |
| SearchCourse | Searching in BST | log n | log n | log n | O(log n) |
| Printing course details | 1 | log n | log n | O(log n) |
| **Total** | | | | n log n | O(n log n) |

**Advantages and Disadvantages**

|  |  |  |
| --- | --- | --- |
| **Data Structure** | **Advantages** | **Disadvantages** |
| **Vector** | * Simple Implementation: Vectors are straightforward to implement and use. They provide dynamic array functionality that can grow as needed. * Efficient Indexing: Accessing elements by index is fast (O(1)), which is beneficial when traversing the list of courses. | * Inefficient Search: Searching for a specific course is less efficient (O(n)), especially as the number of courses increases. * Memory Usage: Vectors can be inefficient in terms of memory usage when resizing the array, leading to memory reallocation, and copying of elements. * Insertion and Deletion: These operations can be slow (O(n)) since they may require shifting elements. |
| **Hash Table** | * Fast Lookup, Insertion, and Deletion: Hash tables provide average-case O(1) time complexity for these operations, making them very efficient for search operations. * Efficient Memory Usage: They can be memory efficient if a good hash function is used and collisions are minimized. | * Hash Collisions: Managing collisions can complicate implementation and reduce performance if not handled correctly. * Memory Overhead: Hash tables may require extra memory for storing hash keys, and poorly designed hash functions can lead to high memory usage. * Unordered: Hash tables do not maintain any order of the elements, which can be a drawback if ordered traversal is needed. |
| **BST** | * Efficient Search, Insertion, and Deletion: These operations have a time complexity of O(log n) in a balanced BST, which is efficient for large datasets. * Ordered Structure: BSTs maintain elements in sorted order, which is beneficial for tasks that require ordered traversal. | * Complex Implementation: Implementing and maintaining a balanced BST (such as AVL or Red-Black Tree) can be complex. * Worst-Case Performance: In the worst case, if the tree becomes unbalanced (e.g., resembling a linked list), the time complexity for operations can degrade to O(n). |

**Recommendation**

The hash table data structure is recommended for this project based on a detailed analysis of different data structures. This recommendation stems from the need to efficiently manage and retrieve course information, as outlined by the project's requirements. The project requires a system that can handle a large dataset of courses, provide fast search and retrieval operations, and maintain efficient performance as the system scales. With their unique properties, Hash tables are the best fit for these requirements.

First, the Hash Table data structure offers a unique advantage in search operations efficiency. Specifically, when printing course information for a specific course, the average-case O(1) lookup time of hash tables is a game changer. This means that retrieving course details remains quick and efficient even as the dataset grows. The ability to maintain constant-time complexity for search operations sets hash tables apart from other data structures in this context.

Secondly, despite hash tables potentially having higher memory overhead due to hash keys, this can be effectively managed with a well-designed hash function. The benefit of having fast operations outweighs the possible memory overhead, making hash tables a practical choice. Efficient memory usage is vital, but the primary goal of achieving rapid data retrieval justifies this trade-off.

Moreover, the Hash Table data structure is a winner when handling large datasets. As the number of courses increases, the efficiency of search, insertion, and deletion operations in hash tables remains consistent. This is in stark contrast to vectors and more balanced than binary search trees (BSTs) in practical scenarios. This scalability is critical to ensuring the system's performance as it evolves and expands.

Another point to consider is that ordered traversal, which refers to visiting each element in a specific order, is optional for this project. While BSTs provide ordered traversal, which can be helpful scenarios, the focus of this project is the efficient search and retrieval of specific course information. Hash tables excel in these operations without maintaining order, simplifying implementation, and enhancing performance.

In contrast, although vectors are simple to implement and efficient for indexing, their search inefficiency (O(n)) makes them less suitable for the frequent search operations required in this project. For instance, if we have a large dataset of courses, the search operation in vectors would take a significant amount of time. On the other hand, BSTs, while providing efficient operations (O(log n)) and maintaining sorted order, have complexities in implementation and the potential for degraded performance if the tree becomes unbalanced. These factors make BSTs less favorable than hash tables' consistent average-case performance.

In summary, the hash table's advantages in fast lookup and overall efficiency align well with the project's requirements, making it the optimal choice for implementing the course management system. This data structure not only meets the project's functional needs but also ensures robust performance as the system scales, providing a reliable and effective solution.