**CS-300 Project One**

The primary objective of this project is to analyze and compare the performance of three different data structures; vector, hash table, and binary search tree (BST) for managing course data at ABC University (ABCU). The goal is to determine which data structure is most efficient and suitable for storing, searching, and retrieving course information, which is essential for academic advisors at ABCU when assisting students with their course selections. The data structures will be evaluated based on their performance in three key operations: loading data, searching for a course, and printing all courses.

**1 - Vector Data Structure**

* **File Input**

*Function loadCourses(fileName):*

*Try:*

*Open file fileName*

*Except IOError:*

*Print "Error: Cannot open file."*

*Return*

*Initialize vector courses*

*For each line in the file:*

*Split line by comma into courseData*

*If length of courseData is less than 2:*

*Print "Error: Line does not contain at least two parameters"*

*Continue to next line*

*Initialize course with courseNumber and name from courseData*

*For each prerequisite in courseData starting from index 2:*

*If prerequisite exists in courses:*

*Add prerequisite to course*

*Else:*

*Print "Error: Prerequisite course not found"*

*Add course to courses*

*Close file*

*Return courses*

* **Course Object**

*Struct Course:*

*String courseNumber*

*String name*

*Vector<String> prerequisites*

*Function createCourse(courseData):*

*Initialize course*

*course.courseNumber = courseData[0]*

*course.name = courseData[1]*

*For each prerequisite in courseData starting from index 2:*

*course.prerequisites.push\_back(prerequisite)*

*Return course*

* **Search and Print Course Information**

*Function searchCourse(vector courses, String courseNumber):*

*For each course in courses:*

*If course.courseNumber == courseNumber:*

*Print course.courseNumber, course.name*

*Print "Prerequisites:"*

*For each prerequisite in course.prerequisites:*

*Print prerequisite*

*Return*

*Print "Course not found"*

**2 - Hash Table Data Structure**

* **File Input**

*Function loadCourses(fileName):*

*Try:*

*Open file fileName*

*Except IOError:*

*Print "Error: Cannot open file."*

*Return*

*Initialize hashTable courses*

*For each line in the file:*

*Split line by comma into courseData*

*If length of courseData is less than 2:*

*Print "Error: Line does not contain at least two parameters"*

*Continue to next line*

*Initialize course with courseNumber and name from courseData*

*For each prerequisite in courseData starting from index 2:*

*If prerequisite exists in courses:*

*Add prerequisite to course*

*Else:*

*Print "Error: Prerequisite course not found"*

*Add course to hashTable courses with key course.courseNumber*

*Close file*

*Return courses*

* **Course Object**

*Struct Course:*

*String courseNumber*

*String name*

*Vector<String> prerequisites*

*Function createCourse(courseData):*

*Initialize course*

*course.courseNumber = courseData[0]*

*course.name = courseData[1]*

*For each prerequisite in courseData starting from index 2:*

*course.prerequisites.push\_back(prerequisite)*

*Return course*

* **Search and Print Course Information**

*Function searchCourse(hashTable courses, String courseNumber):*

*If courses contains courseNumber:*

*course = courses[courseNumber]*

*Print course.courseNumber, course.name*

*Print "Prerequisites:"*

*For each prerequisite in course.prerequisites:*

*Print prerequisite*

*Else:*

*Print "Course not found"*

**3 - Binary Search Tree (BST) Data Structure**

* **File Input**

*Function loadCourses(fileName):*

*Try:*

*Open fileName*

*Except IOError:*

*Print "Error: Cannot open file."*

*Return*

*Initialize binarySearchTree courses*

*For each line in the file:*

*Split line by comma into courseData*

*If length of courseData < 2:*

*Print "Error: Line does not contain at least two parameters."*

*Continue to next line*

*Initialize course with courseNumber and name from courseData*

*For each prerequisite in courseData starting from index 2:*

*If prerequisite does not exist in courses:*

*Print "Error: Prerequisite course not found."*

*Add course to binarySearchTree courses*

*Close file*

*Return courses*

* **Course Object**

Struct Course:

String courseNumber

String name

Vector<String> prerequisites

Function createCourse(courseData):

Initialize course

course.courseNumber = courseData[0]

course.name = courseData[1]

For each prerequisite in courseData starting from index 2:

course.prerequisites.push\_back(prerequisite)

Return course

Function insertCourse(binarySearchTree, course):

If tree is empty:

Set root to course

Else:

Call addNode(root, course)

Function addNode(node, course):

If course.courseNumber < node.courseNumber:

If node.left is null:

Set node.left to course

Else:

Call addNode(node.left, course)

Else:

If node.right is null:

Set node.right to course

Else:

Call addNode(node.right, course)

* **Search and Print Course Information**

*Function printCourseInformation(binarySearchTree, courseNumber):*

*Call searchCourse(binarySearchTree.root, courseNumber)*

*Function searchCourse(node, courseNumber):*

*If node is null:*

*Print "Course not found."*

*Return*

*If courseNumber == node.courseNumber:*

*Print node.courseNumber, node.name*

*Print "Prerequisites:"*

*For each prerequisite in node.prerequisites:*

*Print prerequisite*

*Else If courseNumber < node.courseNumber:*

*Call searchCourse(node.left, courseNumber)*

*Else:*

*Call searchCourse(node.right, courseNumber)*

**4 - Menu System Pseudocode**

The menu system will allow users to:

* Load the file data into the selected data structure (vector, hash table, or BST).
* Print an alphanumerically ordered list of all courses.
* Print course title and prerequisites for a specific course.
* Exit the program.

*Function displayMenu():*

*Print "1: Load Course Data"*

*Print "2: Print All Courses (Alphanumeric Order)"*

*Print "3: Print Course Information"*

*Print "9: Exit"*

*Print "Enter your choice: "*

*userInput = Get user input*

*Return userInput*

*Function main():*

*Initialize Boolean running = True*

*Initialize dataStructureType = None*

*While running:*

*choice = displayMenu()*

*Switch choice:*

*Case "1":*

*Print "Select data structure to use (1: Vector, 2: Hash Table, 3: BST):"*

*dataStructureType = Get user input*

*If dataStructureType == "1":*

*courses = loadCourses("Vector")*

*Else If dataStructureType == "2":*

*courses = loadCourses("Hash Table")*

*Else If dataStructureType == "3":*

*courses = loadCourses("BST")*

*Else:*

*Print "Invalid selection. Please choose a valid option."*

*Case "2":*

*If dataStructureType is None:*

*Print "Please load the course data first."*

*Else:*

*printAllCourses(courses, dataStructureType)*

*Case "3":*

*If dataStructureType is None:*

*Print "Please load the course data first."*

*Else:*

*Print "Enter the course number:"*

*courseNumber = Get user input*

*printCourseInformation(courses, courseNumber, dataStructureType)*

*Case "9":*

*Print "Exiting program."*

*running = False*

*Default:*

*Print "Invalid choice. Please try again."*

**5 - Alphanumeric Sorting and Printing**

This function will sort the courses by their course numbers in alphanumeric order and then print them.

*Function printAllCourses(courses, dataStructureType):*

*If dataStructureType == "Vector":*

*Sort courses by course.courseNumber alphanumerically*

*For each course in courses:*

*Print course.courseNumber, course.name*

*Else If dataStructureType == "Hash Table":*

*Convert hashTable to list*

*Sort list by course.courseNumber alphanumerically*

*For each course in sorted list:*

*Print course.courseNumber, course.name*

*Else If dataStructureType == "BST":*

*PrintInOrder(binarySearchTree.root)*

*Function PrintInOrder(node):*

*If node is not null:*

*PrintInOrder(node.left)*

*Print node.courseNumber, node.name*

*PrintInOrder(node.right)*

**Explanation of the Pseudocode:**

Menu System:

* The displayMenu() function displays the available menu options and gets the user's choice.
* The main() function drives the entire program, handling user input and delegating tasks based on the selected option.
* The main() function ensures the user first selects the data structure before attempting to perform operations like printing or searching.

Alphanumeric Sorting and Printing:

* Vector: Directly sorts the vector by course number and prints the sorted list.
* Hash Table: Converts the hash table into a list, sorts it, and then prints the sorted list.
* BST: Uses an in-order traversal to print the courses in alphanumeric order, which is naturally handled by the BST structure.

**Runtime Analysis for Each Data Structure**

**1. Vector Data Structure**

* **Loading Data (loadCourses)**:

**Operation**: Reading and inserting courses into a vector.

**Worst-case runtime**: O(n) where n is the number of courses. Each course is processed once.

* **Search for Course (searchCourse)**:

**Operation**: Linear search through the vector to find the course.

**Worst-case runtime**: O(n), as the search might need to traverse the entire vector.

* **Sorting Courses (printAllCourses)**:

**Operation**: Sorting the vector alphanumerically by course number.

**Worst-case runtime**: O(n log n), using a sorting algorithm like quicksort or mergesort.

* **Memory Usage**:

**Memory Complexity**: O(n), since a vector requires memory proportional to the number of courses.

**2. Hash Table Data Structure**

* **Loading Data (loadCourses)**:

**Operation**: Inserting courses into a hash table.

**Worst-case runtime**: O(n) for inserting n courses, assuming an average case of O(1) insertion.

* **Search for Course (searchCourse)**:

**Operation**: Searching for a course in the hash table.

**Worst-case runtime**: O(1) in the average case, but can degrade to O(n) in case of hash collisions.

* **Sorting Courses (printAllCourses)**:

**Operation**: Convert hash table to list and then sort the list.

**Worst-case runtime**: O(n log n), as sorting is required after converting to a list.

* **Memory Usage**:

**Memory Complexity**: O(n), with additional memory overhead for hash buckets.

**3. Binary Search Tree (BST) Data Structure**

* **Loading Data (loadCourses)**:

**Operation**: Inserting courses into a BST.

**Worst-case runtime**: O(n log n) if the tree is balanced, but O(n2) in the worst case (unbalanced).

* **Search for Course (searchCourse)**:

**Operation**: Searching for a course in the BST.

**Worst-case runtime**: O(log n) if the tree is balanced, but O(n) in the worst case (unbalanced).

* **Sorting Courses (printAllCourses)**:

**Operation**: In-order traversal of the BST.

**Worst-case runtime**: O(n), since in-order traversal naturally outputs elements in sorted order.

* **Memory Usage**:

**Memory Complexity**: O(n), with additional memory for tree nodes.

**Advantages and Disadvantages**

* **Vector**:

**Advantages**: Simple and efficient for small datasets. Easy to implement.

**Disadvantages**: Slow search and sort operations as the size of data increases.

* **Hash Table**:

**Advantages**: Fast average-case search and insert operations.

**Disadvantages**: Sorting requires conversion to a list, and performance can degrade with hash collisions.

* **Binary Search Tree (BST)**:

**Advantages**: Efficient search, insertion, and naturally sorted traversal if balanced.

**Disadvantages**: Can become unbalanced, leading to degraded performance.

**Final Recommendation**

Based on the runtime analysis:

* **For large datasets** where fast retrieval is crucial, a **Hash Table** is recommended due to its average-case O(1) search time.
* If the data needs to be frequently sorted or if you require naturally ordered data, a **BST** is preferred, especially if you can ensure the tree remains balanced.
* **Vector** might be suitable for smaller datasets or where simplicity is more important than performance.