Kileigh Adams

SNHU

CS 300

Project One

## Vector Implementation

//VECTOR IMPLEMENTATION  
// Structure for course information  
Define a struct called Course:  
 courseNumber: string  
 courseTitle: string  
 prerequisites: list of strings  
  
// Function to read course data from a file and store in a vector  
Function loadCourses(fileName):  
 Open the file with the given fileName  
 Create an empty vector called courses  
 Create an empty set called courseIDs  
  
 For each line in the file:  
 If the line is empty, skip it  
 Split the line by commas into parts  
 If the number of parts < 2:  
 Print "Error: Each line must include at least a course number and title"  
 Exit the program  
 Create a new Course object c  
 Set c.courseNumber = parts[0]  
 Set c.courseTitle = parts[1]  
 For i from 2 to length(parts)-1:  
 Append parts[i] to c.prerequisites  
 Add c.courseNumber to courseIDs  
 Append c to courses  
 Close the file  
  
 // Validate prerequisites  
 For each course in courses:  
 For each prereq in course.prerequisites:  
 If prereq NOT IN courseIDs:  
 Print "Error: Prerequisite " + prereq + " not found."  
 Exit the program  
 Return courses  
  
// Function to search for a course  
Function searchCourse(courses, courseNumber):  
 For each course in courses:  
 If course.courseNumber == courseNumber:  
 Print course details  
 If prerequisites is empty: Print "None"  
 Else: Print each prerequisite  
 Return  
 Print "Course not found."  
  
// Function to print all courses in sorted order  
Function printAllCourses(courses):  
 Sort courses by courseNumber ascending  
 For each course in courses:  
 Print course.courseNumber + ": " + course.courseTitle  
  
// Menu function  
Function menuVector():  
 courses <- empty  
 loaded <- false  
 LOOP:  
 Print menu options  
 Read choice  
 If choice == 1:  
 courses <- loadCourses("courses.txt")  
 loaded <- true  
 Else if choice == 2:  
 If NOT loaded: Print "Load data first" ; CONTINUE  
 printAllCourses(courses)  
 Else if choice == 3:  
 If NOT loaded: Print "Load data first" ; CONTINUE  
 key <- prompt for course number  
 searchCourse(courses, key)  
 Else if choice == 9:  
 BREAK  
 Else:  
 Print "Invalid option"

## Hash Table Implementation

// HASH TABLE IMPLEMENTATION  
// Structure for course information  
Define a struct called Course:  
 courseNumber: string  
 courseTitle: string  
 prerequisites: list of strings  
  
// Function to read course data and store in a hash table  
Function loadCourses(fileName):  
 Create empty hash table courseTable  
 Create empty set courseIDs  
 Create empty list rawCourseLines  
 Open the file  
 For each line:  
 If empty, skip  
 Append to rawCourseLines  
 Split into parts  
 If parts < 2: error and exit  
 Add parts[0] to courseIDs  
 // Second pass: validate and store  
 For each line in rawCourseLines:  
 Split into parts  
 Create new Course object c  
 Set c.courseNumber = parts[0]  
 Set c.courseTitle = parts[1]  
 For each remaining part:  
 If part NOT IN courseIDs: error and exit  
 Append to c.prerequisites  
 Insert into courseTable with key = c.courseNumber  
 Close file  
 Return courseTable  
  
// Search function  
Function searchCourse(courseTable, courseNumber):  
 Retrieve course from courseTable  
 If null: Print "Not found" ; return  
 Print details and prerequisites  
  
// Print all sorted  
Function printAllCourses(courseTable):  
 keys <- list of all keys from courseTable  
 Sort keys ascending  
 For each key:  
 Print courseTable[key].courseNumber + ": " + courseTable[key].courseTitle  
  
// Menu function  
Function menuHash():  
 loaded <- false  
 LOOP:  
 Print menu  
 If choice == 1:  
 courseTable <- loadCourses("courses.txt")  
 loaded <- true  
 Else if choice == 2:  
 If NOT loaded: message ; CONTINUE  
 printAllCourses(courseTable)  
 Else if choice == 3:  
 If NOT loaded: message ; CONTINUE  
 key <- prompt  
 searchCourse(courseTable, key)  
 Else if choice == 9: BREAK

## Binary Search Tree Implementation

// BINARY SEARCH TREE IMPLEMENTATION  
// Structure for course  
Define struct Course:  
 courseNumber: string  
 courseTitle: string  
 prerequisites: list of strings  
  
// BST Node  
Define struct Node:  
 data: Course  
 left: Node  
 right: Node  
  
// BST Class  
Class BinarySearchTree:  
 root: Node

Method insert(course):  
 If root null: root = new Node(course)  
 Else: call insertNode(root, course)  
  
 Method insertNode(current, course):  
 If course.courseNumber < current.data.courseNumber:  
 If current.left null: current.left = new Node(course)  
 Else: insertNode(current.left, course)  
 Else if course.courseNumber > current.data.courseNumber:  
 If current.right null: current.right = new Node(course)  
 Else: insertNode(current.right, course)  
  
 Method search(courseNumber):  
 current = root  
 While current not null:  
 If equal: return current.data  
 Else if less: current = current.left  
 Else: current = current.right  
 Return null  
  
 Method inOrder(node):  
 If node null: return  
 inOrder(node.left)  
 Print node.data.courseNumber + ": " + node.data.courseTitle  
 inOrder(node.right)  
  
// Load courses into BST  
Function loadCourses(fileName):  
 Create new BinarySearchTree courseTree  
 Create set courseIDs  
 Create list rawCourseLines  
 Open file  
 First pass: collect IDs, store lines  
 Second pass: validate prerequisites, insert into BST  
 Return courseTree  
  
// Print sorted  
Function printAllCourses(courseTree):  
 courseTree.inOrder(courseTree.root)  
  
// Search and print  
Function searchCourse(courseTree, courseNumber):  
 course = courseTree.search(courseNumber)  
 If null: message ; return  
 Print details and prerequisites  
  
// Menu function  
Function menuBST():  
 loaded <- false  
 LOOP:  
 Print menu  
 If choice == 1:  
 courseTree <- loadCourses("courses.txt")  
 loaded <- true  
 Else if choice == 2:  
 If NOT loaded: message ; CONTINUE  
 printAllCourses(courseTree)  
 Else if choice == 3:  
 If NOT loaded: message ; CONTINUE  
 key <- prompt  
 searchCourse(courseTree, key)  
 Else if choice == 9: BREAK

# Runtime Analysis

## Application functions independent of data structure

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Average Case Time | Notes | Space |
| DisplayMenu() + GetUserChoice() | O(1) | Print menu and read one value | O(1) |
| ParseFileLines() | O(n + p) | Read n course lines; validate p prerequisites | O(n) temporary set/list |
| DisplayCourse(course) | O(k) | Print one course and its k prerequisites after found | O(1) |
| DisplayCatalog(courses) | O(n) | Print n courses in current order | O(1) |
| Exit() | O(1) | End program | O(1) |

n = number of courses; p = total prerequisites; k = number of prerequisites for one course.  
Overall independent time: O(n + p). Overall independent space: O(n) due to the temporary course ID set used during validation.

## Application functions dependent on data structure

## Time Complexity

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Structure | Load() | Search(courseNumber) | Ordered Output for Catalog | Total for Typical Run |
| Vector | O(n + p) parse and push\_back | O(n) scan or O(log n) if sorted + binary search | O(n log n) to sort once; O(n) to print | If sorted once: O(n log n + p); then lookups O(log n) |
| Hash Table | O(n + p) average | O(1) average, O(n) worst | Extract keys O(n) + sort O(n log n) + print O(n) | O(n log n + p) average |
| BST (plain) | O(n log n + p) avg; O(n^2 + p) worst if unbalanced | O(log n) avg; O(n) worst | In-order traversal O(n) | O(n log n + p) avg; O(n^2 + p) worst |

## Space Complexity

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Structure | Load() | Search() | Sort() | Total |
| Vector | O(1) | O(1) | O(log n) | O(log n) |
| Hash Table | O(n) | O(1) | O(n + log n) | O(n) |
| Binary Search Tree | O(n) | O(1) | 1 | O(n) |

## Advantages and disadvantages

* Vectors are simple to work with and store data in a contiguous block of memory, which makes iteration quick and memory use efficient. They handle insertions well and work best when data does not need to be constantly sorted. However, sorting can be slow depending on the algorithm, and searching is also slower unless the vector is sorted and binary search is used. Vectors are a good choice when the main task is adding lots of courses and only occasionally searching or printing the catalog in order.
* Hash tables store data in buckets using a hashed key, making searches and insertions very fast in most cases. They are a good fit for situations where quick lookups by course number are common. The trade-off is that they do not store items in order, so producing a sorted list requires extra work and time. They also use more memory for their bucket structure and can slow down if hashing is poor or collisions increase.
* Binary search trees keep their data sorted as items are inserted, which makes ordered output easy and fast. They also provide good average search performance if the tree is balanced. The main drawback is that if data is inserted in sorted order without balancing, performance drops sharply as the tree becomes skewed. They also require more careful pointer management than vectors or hash tables. BSTs work best when ordered output is needed frequently and the insertion order can be varied to keep the tree balanced.