# **Project 1**

## **Pseudocode:**

### **Vector Data Structure**

Function loadCoursesFromFile(fileName):

Open file fileName

courses = Vector<Course>()

While line = file.readLine():

tokens = line.split(delimiter)

If tokens.size() < 2:

Print "Error: Invalid line format"

Continue

course = createCourse(tokens)

For i from 2 to tokens.size() - 1:

prerequisite = tokens[i]

If findCourse(courses, prerequisite) == NULL:

Print "Error: " + prerequisite + " does not exist."

courses.add(course)

Close file

Return courses

Struct Course:

String courseNumber

String courseTitle

Vector<String> prerequisites

Function createCourse(tokens):

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = Vector<String>()

For i from 2 to tokens.size() - 1:

prerequisites.add(tokens[i])

Return Course(courseNumber, courseTitle, prerequisites)

Function printCourse(course):

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.courseTitle

If course.prerequisites is not empty:

Print "Prerequisites: " + course.prerequisites

Else:

Print "No prerequisites"

Function findCourse(courses, courseNumber):

For each course in courses:

If course.courseNumber == courseNumber:

Return course

Return NULL

### **Hash Table Data Structure:**

Function loadCourseData(filePath):

Open file filePath

hashTable = HashTable<Course>()

While line = file.readLine():

tokens = line.split(delimiter)

If tokens.size() < 2:

Print "Error: Invalid line format"

Continue

course = createCourse(tokens)

For i from 2 to tokens.size() - 1:

prerequisite = tokens[i]

If prerequisite not in hashTable:

Print "Error: " + prerequisite + " does not exist."

insertCourseIntoHashTable(hashTable, course)

Close file

Return hashTable

Function createCourse(tokens):

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = Vector<String>()

For i from 2 to tokens.size() - 1:

prerequisites.add(tokens[i])

Return Course(courseNumber, courseTitle, prerequisites)

Function printCourse(course):

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.courseTitle

If course.prerequisites is not empty:

Print "Prerequisites: " + course.prerequisites

Else:

Print "No prerequisites"

Function insertCourseIntoHashTable(hashTable, course):

key = hash(course.courseNumber)

If hashTable[key] is empty:

hashTable[key] = course

Else:

HandleCollision(hashTable, key, course)

Function searchCourse(hashTable, courseNumber):

key = hash(courseNumber)

If course exists at hashTable[key]:

Print course.courseNumber + ": " + course.title

Else:

Print "Course not found"

### **Binary Search Tree Data Structure**

Struct Course:

String courseNumber

String courseTitle

Vector<String> prerequisites

Struct Node:

Course course

Node\* left

Node\* right

Class BinarySearchTree:

Node\* root

Function Insert(course):

If root is NULL:

root = new Node(course)

Else:

Call addNode(root, course)

Function addNode(node, course):

If course.courseNumber < node.course.courseNumber:

If node.left is NULL:

node.left = new Node(course)

Else:

Call addNode(node.left, course)

Else:

If node.right is NULL:

node.right = new Node(course)

Else:

Call addNode(node.right, course)

Function Search(courseNumber):

Return searchNode(root, courseNumber)

Function searchNode(node, courseNumber):

If node is NULL:

Return NULL

If courseNumber == node.course.courseNumber:

Return node.course

Else If courseNumber < node.course.courseNumber:

Return searchNode(node.left, courseNumber)

Else:

Return searchNode(node.right)

Function InOrder():

Call inOrderTraversal(root)

Function inOrderTraversal(node):

If node is not NULL:

Call inOrderTraversal(node.left)

Print node.course.courseNumber + ": " + node.course.courseTitle

If node.course.prerequisites is not empty:

Print "Prerequisites: " + node.course.prerequisites

Call inOrderTraversal(node.right)

Function loadCoursesFromFile(fileName):

Open file fileName

While line = readLine(file):

Split line by delimiter into tokens

If tokens.size() < 2:

Print "Error: Invalid line"

Continue

course = Course(tokens[0], tokens[1], Vector<String>())

For i = 2 to tokens.size() - 1:

prerequisite = tokens[i]

If Search(prerequisite) == NULL:

Print "Error: " + prerequisite + " does not exist."

Else:

course.prerequisites.add(prerequisite)

Call Insert(course)

Close file

Function searchCourse(tree, courseNumber):

course = tree.Search(courseNumber)

If course == NULL:

Print "Course not found"

Else:

Print course.courseNumber + ": " + course.courseTitle

For each prerequisite in course.prerequisites:

prereqCourse = tree.Search(prerequisite)

Print prereqCourse.courseNumber + ": " + prereqCourse.courseTitle

## **Menu Pseudocode:**

Function displayMenu():

Print "1: Load courses from file"

Print "2: Print all courses in alphanumeric order"

Print "3: Search and display course information"

Print "9: Exit"

choice = Get user input

Return choice

Function menu():

tree = BinarySearchTree()

While True:

choice = displayMenu()

If choice == 1:

loadCoursesFromFile("courses.txt")

Else If choice == 2:

tree.InOrder()

Else If choice == 3:

courseNumber = Get user input

searchCourse(tree, courseNumber)

Else If choice == 9:

Exit program

## **Print Courses in Alphanumeric Order**

### **Vector:**

Function printAllCourses(courses):

Sort courses by courseNumber

For each course in courses:

Print course.courseNumber + ": " + course.courseTitle

If course.prerequisites is not empty:

Print "Prerequisites: " + course.prerequisites

### **Hash Table:**

Function printAllCourses(hashTable):

courses = Vector<Course>()

For each course in hashTable:

If course is not NULL:

courses.add(course)

Sort courses by courseNumber

For each course in courses:

Print course.courseNumber + ": " + course.courseTitle

If course.prerequisites is not empty:

Print "Prerequisites: " + course.prerequisites

### **Binary Search Tree:**

Function printAllCourses(tree):

tree.InOrder()

## **Runtime Analysis:**

### **Vector Data Structure Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Open file fileName | 1 | 1 | 1 |
| courses = Vector<Course>() | 1 | 1 | 1 |
| While line = file.readLine(): | 1 | n | n |
| tokens = line.split(delimiter) | 1 | n | n |
| If tokens.size() < 2: | 1 | n | n |
| Print "Error: Invalid line" | 1 | m | m |
| Continue | 1 | m | m |
| course = createCourse(tokens) | 1 | n - m | n - m |
| For i from 2 to tokens.size() - 1: | 1 | n \* k | n \* k |
| prerequisite = tokens[i] | 1 | n \* k | n \* k |
| If findCourse(courses, prerequisite) == NULL: | 1 | n \* k | n \* k^2 |
| Print "Error: prerequisite" | 1 | c | c |
| courses.add(course) | 1 | n | n |
| Close file | 1 | 1 | 1 |
| Return courses | 1 | 1 | 1 |

Total Cost: 5n + n \* k^2 + m + c

Runtime Complexity: O(n \* k^2)

### **Hash Table Data Structure Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Open file filePath | 1 | 1 | 1 |
| hashTable = HashTable<Course>() | 1 | 1 | 1 |
| While line = file.readLine(): | 1 | n | n |
| tokens = line.split(delimiter) | 1 | n | n |
| If tokens.size() < 2: | 1 | n | n |
| Print "Error: Invalid line" | 1 | m | m |
| Continue | 1 | m | m |
| course = createCourse(tokens) | 1 | n - m | n - m |
| For i from 2 to tokens.size() - 1: | 1 | n \* k | n \* k |
| prerequisite = tokens[i] | 1 | n \* k | n \* k |
| If prerequisite not in hashTable: | 1 | n \* k | n \* k |
| Print "Error: prerequisite" | 1 | c | c |
| insertCourseIntoHashTable(hashTable, course) | 1 | n | n |
| Close file | 1 | 1 | 1 |
| Return hashTable | 1 | 1 | 1 |

Total Cost: 4n + n \* k + m + c

Runtime Complexity: O(n \* k)

### **Binary Search Tree (BST) Data Structure Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Open file fileName | 1 | 1 | 1 |
| While line = file.readLine(): | 1 | n | n |
| tokens = line.split(delimiter) | 1 | n | n |
| If tokens.size() < 2: | 1 | n | n |
| Print "Error: Invalid line" | 1 | m | m |
| Continue | 1 | m | m |
| course = createCourse(tokens) | 1 | n - m | n - m |
| For i from 2 to tokens.size() - 1: | 1 | n \* k | n \* k |
| prerequisite = tokens[i] | 1 | n \* k | n \* k |
| If Search(prerequisite) == NULL: | 1 | n \* k | n \* k \* log(n) |
| Print "Error: prerequisite" | 1 | c | c |
| Insert(course) | 1 | n | n \* log(n) |
| Close file | 1 | 1 | 1 |
| Return tree | 1 | 1 | 1 |

Total Cost: 4n + n \* k \* log(n) + m + c

Runtime Complexity: O(n \* k \* log(n))

## **Analyze the Vector, Hash Table, and Binary Search Tree Data Structures**

#### **Vector Data Structure**

**Advantages**:

* + Simple to implement and use.
  + Efficient when working with small datasets.
  + Insertion at the end of the vector is fast (O(1) for appending).

**Disadvantages**:

* + Searching for an element requires a linear scan, which has O(n) complexity.
  + Finding a prerequisite results in quadratic complexity (O(n \* k²)), which makes this data structure less efficient as the number of courses (n) and prerequisites (k) increases.
  + Not ideal for large datasets due to the linear search complexity and performance degradation when dealing with large numbers of courses and prerequisites.

#### **Hash Table Data Structure**

**Advantages**:

* + Provides constant-time O(1) average lookup and insertion operations, making it very efficient for searching and inserting courses.
  + Handles large datasets effectively, as the insertion and search for prerequisites are much faster compared to a vector.
  + No need to manually sort data since lookups are based on hashed keys.

**Disadvantages**:

* + Collision handling mechanisms (like chaining or open addressing) can lead to performance degradation in the worst case.
  + Inserting and searching a prerequisite depends on the hashing function, and a poor hashing function can affect performance.
  + Consumes more memory due to the additional overhead required for maintaining hash tables.

**Use Case**: Best suited when fast lookups and insertions are required, especially when the dataset is large.

#### **Binary Search Tree (BST) Data Structure**

**Advantages**:

* + Automatically sorts the data as it is inserted, making it easier to retrieve data in sorted order.
  + Searching and inserting are logarithmic operations in a balanced tree (O(log n)), which offers good performance as long as the tree remains balanced.
  + Efficient for both searching and ordered data traversal.

**Disadvantages**:

* + Worst-case time complexity for search, insert, and delete operations is O(n) if the tree becomes unbalanced (e.g., if data is inserted in sorted order).
  + Requires additional maintenance (such as self-balancing) to ensure it remains efficient.
  + Slightly more complex to implement compared to a hash table or vector.

## **Recommendation Based on the Analysis**

#### **Recommendation: Hash Table Data Structure**

**Justification**:

1. **Efficiency**:
   1. The hash table provides the best average-case time complexity for searching and inserting courses, both of which are O(1) on average. This makes it ideal for handling large datasets where courses and prerequisites need to be looked up quickly.
   2. In contrast, the vector requires O(n \* k²) time complexity, which grows rapidly as the number of courses and prerequisites increases. The binary search tree, while better than a vector, has logarithmic complexity (O(n \* k \* log(n)), which is slower than a hash table for large datasets.
2. **Scalability**:
   1. As the number of courses and prerequisites grows, the hash table remains the most scalable option due to its constant-time lookups. In contrast, a vector's performance degrades significantly with larger datasets due to its O(n) linear search.
   2. The binary search tree provides good scalability as well (O(log n)), but the risk of the tree becoming unbalanced and having O(n) worst-case complexity makes it less reliable without additional balancing mechanisms.
3. **Simplicity in Handling Prerequisites**:
   1. The hash table excels at quickly checking if a prerequisite exists, which is crucial for this application. The constant-time search for prerequisites gives it a significant edge over the binary search tree and vector.
4. **Drawbacks Considered**:
   1. Although hash tables require more memory overhead and could suffer from collisions, these are manageable issues. Proper hashing techniques and collision resolution strategies can mitigate these drawbacks.
   2. The vector and binary search tree are easier to understand and implement, but they cannot match the efficiency of the hash table for this specific use case.

In conclusion, **the hash table** is the best data structure for this application due to its fast average-case performance for both insertion and lookup operations, which are critical for managing course data with prerequisites efficiently.