VECTOR SORTING

|  |  |  |
| --- | --- | --- |
| Line | Line Cost | # Times Executes |
| ------------------------------------------ | --------- | ----------------- |
| OPEN Course information file | 1 | 1 |
| IF file open successful | 1 | 1 |
| CREATE empty VECTOR named ValidCourseIds | 1 | 1 |
| SET line to first line | 1 | 1 |
| WHILE current line is not null | 1 | n |
| READ line | 1 | n |
| WHILE current line contains 2 or more parameters | 1 | n |
| IF != all parameters in spot >= 3 match any line of ValidCourseIds | 1 | n |
| RETURN ERROR | 1 | m |
| Add courseNum (parameter 1) to ValidCourseIds | 1 | n |
| INCREMENT line | 1 | n |
| INITIATE new courseObject | 1 | 1 |
| SET courseName to string | 1 | 1 |
| SET courseNum to int | 1 | 1 |
| SET PreReqs to vector<string> | 1 | 1 |
| SET line to first line | 1 | 1 |
| WHILE current line is not null | 1 | n |
| FOR each parameter | 1 | n |
| Pushback to add value to the current vector until new line | 1 | n |
| Increment Line | 1 | n |
| IF line is null | 1 | 1 |
| BREAK | 1 | 1 |
| Total Cost: 16n + 7 | | |
| Runtime Complexity: O(n) | | |

HASH TABLE

|  |  |  |
| --- | --- | --- |
| Line | Line Cost | # Times Executes |
| --------------------------------------------- | --------- | ----------------- |
| INITIALIZE hashTable | 1 | 1 |
| OPEN Course information file | 1 | 1 |
| WHILE not end of file | 1 | n |
| IF countParameters(line) < 2 | 1 | n |
| PRINT("Error: Insufficient parameters") | 1 | n |
| courseNumber, name, prerequisites = readLine(line) | 1 | n |
| FOR each prerequisite in prerequisites | 1 | n |
| IF prerequisite not in hashTable | 1 | n |
| PRINT("Error: Prerequisite not found in the file") | 1 | n |
| Function countParameters(line) |  |  |
| tokens = Split(line) | 1 | 1 |
| Return Count(tokens) | 1 | 1 |
| Function readLine(line) |  |  |
| tokens = Split(line) | 1 | 1 |
| courseNumber = tokens[0] | 1 | 1 |
| name = tokens[1] | 1 | 1 |
| prerequisites = tokens[2:] | 1 | 1 |
| RETURN courseNumber, name, prerequisites | 1 | 1 |
| Function createCourseObject(courseNumber, name, prerequisites) |  |  |
| courseObject = new courseObject | 1 | 1 |
| courseObject.courseNumber = courseNumber | 1 | 1 |
| courseObject.name = name | 1 | 1 |
| courseObject.prerequisites = prerequisites | 1 | 1 |
| RETURN courseObject | 1 | 1 |
| FOR each courseNumber in hashTable | 1 | n |
| courseObject = hashTable.Get(courseNumber) | 1 | n |
| PRINT("Course Number:", courseObject.courseNumber) | 1 | n |
| PRINT("Title:", courseObject.name) | 1 | n |
| IF courseObject.prerequisites is not empty | 1 | n |
| PRINT("Prerequisites:", courseObject.prerequisites) | 1 | n |
| Total Cost: 10n+2 | | |
| Runtime Complexity: O(n) | | |

BINARY TREE

|  |  |  |
| --- | --- | --- |
| Line | Line Cost | # Times Executes |
| --------------------------------------------- | --------- | ----------------- |
| INITIALIZE myCourseTree | 1 | 1 |
| Open course information file | 1 | 1 |
| WHILE not end of file | 1 | n |
| courseNumber, title, prerequisites = ParseAndVerify(line) | 1 | n |
| IF InsufficientParameters(line) | 1 | n |
| PRINT("Error: Insufficient parameters in line") | 1 | n |
| courseObject = CreateCourseObject(courseNumber, courseTitle, coursePrereq) | 1 | n |
| IF new courseNumber < current courseNumber | 1 | n |
| IF left node is null | 1 | n |
| Create new node | 1 | n |
| Insert courseObject information into new node | 1 | n |
| ELSE | 1 | n |
| Iterate down the list | 1 | n |
| ELSE IF new courseNumber > current courseNumber | 1 | n |
| IF right node is null | 1 | n |
| Create new right node | 1 | n |
| Insert courseObject information into new right node | 1 | n |
| ELSE | 1 | n |
| Iterate down list | 1 | n |
| ELSE | 1 | n |
| PRINT("Course information already exists") | 1 | n |
| RETURN courseObject | 1 | n |
| FOR each courseObject in myCourseTree | 1 | n |
| PRINT("Course Number:", courseObject.courseNumber) | 1 | n |
| PRINT("Title:", courseObject.courseTitle) | 1 | n |
| IF courseObject.coursePrereq is not empty | 1 | n |
| PRINT("Prerequisites:", courseObject.coursePrereq) | 1 | n |
| Total Cost: 15n + 2 | | |
| Runtime complexity(AT BEST): O(log n) | | |
| Runtime complexity(AT WORST): O(n) | | |

**PART FIVE:**

**Vector Sorting:**

Advantages:

Simple Implementation: Using a vector for sorting is straightforward and easy to implement.

Efficient for Small Datasets: For small datasets or when the list of courses is not expected to change frequently, a vector can offer good performance.

Disadvantages:

Inefficient for Frequent Updates: When the list is frequently updated (courses added or removed), inserting or deleting elements in the middle of a vector can be inefficient.

Linear Search: Searching for a specific course involves a linear search, resulting in O(n) time complexity.

**Hash Table:**

Advantages:

Constant Time Lookup: Hash tables offer constant-time lookup on average, providing fast access to course information.

Efficient for Dynamic Datasets: Handling dynamic datasets, where courses are frequently added or removed, is efficient with hash tables.

Disadvantages:

Memory Overhead: Hash tables may have a memory overhead due to potential collisions and the need for a sufficiently large table.

No Natural Ordering: Courses are not stored in a natural order, making it challenging to retrieve a sorted list without additional data structures or operations.

**Binary Tree:**

Advantages:

Logarithmic Search Time (Balanced): In a balanced binary tree, search operations have a logarithmic time complexity (O(log n)), making them efficient.

Natural Ordering: Binary trees naturally maintain ordering, making it easier to retrieve a sorted list of courses.

Disadvantages:

Requires Balancing: If not balanced, a binary tree can degrade into a linked list, resulting in O(n) time complexity for search operations.

Complex Implementation: Implementing and maintaining a balanced binary tree can be more complex than other data structures.