Connor Camire

Professor Othman

CS 300

6/3/2025

Struct Course {

courseNumber as string

courseTitle as string

prerequisites as list of strings

}

Function DisplayMenu() {

set userChoice to 0

set fileLoaded to false

while userChoice is not 9 {

print "1. Load Data Structure"

print "2. Print Course List"

print "3. Print Course"

print "9. Exit"

get userChoice from input

switch userChoice {

case 1:

set courseLines to LoadFileData("CourseData.csv")

// Uncomment one depending on implementation

// set dataStructure to CreateVector(courseLines)

// set dataStructure to CreateHashTable(courseLines)

// set dataStructure to CreateCourseTree(courseLines)

set fileLoaded to true

break

case 2:

if fileLoaded is false {

print "Please load the file first"

break

}

PrintCourses(dataStructure)

break

case 3:

if fileLoaded is false {

print "Please load the file first"

break

}

print "Enter course number to search:"

get searchValue from input

SearchCourse(dataStructure, searchValue)

break

case 9:

print "Exiting program..."

break

default:

print "Invalid option"

}

}

}

Function PrintCourses(courseVector) {

sort courseVector by courseNumber ascending

for each course in courseVector {

print course.courseNumber + ", " + course.courseTitle

if course.prerequisites is not empty {

print "Prerequisites: " + course.prerequisites

} else {

print "No prerequisites"

}

}

}

Function SearchCourse(courseVector, searchValue) {

set found to false

for each course in courseVector {

if course.courseNumber equals searchValue {

print course.courseNumber + ", " + course.courseTitle

if course.prerequisites is not empty {

print "Prerequisites: " + course.prerequisites

} else {

print "No prerequisites"

}

set found to true

break

}

}

if found is false {

print "Course not found"

}

}

Function PrintCourses(hashTable) {

set courseList to all values in hashTable

sort courseList by courseNumber ascending

for each course in courseList {

print course.courseNumber + ", " + course.courseTitle

if course.prerequisites is not empty {

print "Prerequisites: " + course.prerequisites

} else {

print "No prerequisites"

}

}

}

Function SearchCourse(hashTable, searchValue) {

set course to find course in hashTable using searchValue

if course is null {

print "Course not found"

} else {

print course.courseNumber + ", " + course.courseTitle

if course.prerequisites is not empty {

print "Prerequisites: " + course.prerequisites

} else {

print "No prerequisites"

}

}

}

Function PrintBinaryTree(node) {

if node is not null {

call PrintBinaryTree(node.left)

print node.courseNumber + ", " + node.courseTitle

if node.prerequisites is not empty {

print "Prerequisites: " + node.prerequisites

} else {

print "No prerequisites"

}

call PrintBinaryTree(node.right)

}

}

Function PrintCourses(courseTree) {

print "All courses:"

call PrintBinaryTree(courseTree.root)

}

Function SearchCourse(courseTree, searchValue) {

set course to courseTree.search(searchValue)

if course is null {

print "Course not found"

} else {

print course.courseNumber + ", " + course.courseTitle

if course.prerequisites is not empty {

print "Prerequisites: " + course.prerequisites

} else {

print "No prerequisites"

}

}

}

Different run time complexities from past assignments

| Line Description | Line Cost | Times Executed | Total Cost |
| --- | --- | --- | --- |
| Create courseVector | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Trim whitespace, split line, add to list | 3 | n | 3n |
| Validate prerequisites in nested loop | n | n | n^2 |
| For each amount in courseLines | 1 | n | n |
| Create course object | 1 | n | n |
| For each prereq in amount | 1 | n | n |
| Append prereq to list | 1 | n | n |
| Push back course to vector | 1 | n | n |
| Total Cost | 6n+n^2+1 | Runtime | O(n^2) |

| Line Description | Line Cost | Times Executed | Total Cost |
| --- | --- | --- | --- |
| Create empty hash table | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Trim, split line, add to list | 3 | n | 3n |
| Validate prerequisites | 1 | n\*m | nm |
| For each tokens in allCourseLines | 1 | n | n |
| Create course object | 1 | n | n |
| For each prereq in tokens | 1 | n | n |
| Add prereq to list | 1 | n | n |
| Insert course into hashTable | 1 | n | n |
| Total Cost | 8n + nm +1 | Runtime | O(n × m) avg  O(n^2) worst |

| Line Description | Line Cost | Times Executed | Total Cost |
| --- | --- | --- | --- |
| Create BinarySearchTree | 1 | 1 | 1 |
| For each line in file | 1 | n | n |
| Trim, split, add to list | 3 | n | 3n |
| Validate prerequisites | 1 | n\*m | nm |
| For each tokens in courseLines | 1 | n | n |
| Create course object | 1 | n | n |
| Add prereq to list | 1 | n | n |
| Insert course into BST | Log n | n | n log n |
| Total Cost | 7n + n log n + nm + 1 | Runtime | O(n log n + nm) avg  O(n^2) worst |

Each of the three utilized search algorithms have their own strengths and weaknesses. A vector is the most simple and straightforward to implement. It's best for displaying the information in the order that it was in from file that we gave it information from. A consequence, though, is to search for information in a vector, it is linear, so it must iterate through each item in the list. leaving it with a large worst-case scenario, runtime complexity.

A hash table uses a key (course number) to allow for quick access of information. As long as you minimize the amount of collisions that exist within the hash table, it can scale significantly without badly damaging its performance. Too many collisions can cause a significantly longer runtime. This is because the more collisions there are, the larger a particular bucket can be, and the larger that bucket gets, the larger it would take to search for something within that bucket, since it needs to be done linearly.

Lastly, a binary search tree. Binary search trees keep things ordered, in this case, by course number, and when searching for information, will beat out the vector in terms of worst case scenario. The downside of a binary search tree is its average runtime can be lower than the optimal runtime of the other search methods, and the implementation can be more difficult. And in the case of a poor implementation, it could create an unbalanced search tree which will increase the worst-case scenario significantly.

With all this in mind, which method would I choose? Well, I suppose that would depend on exactly what you were looking for. For vectors, you would only want to use one when the search speed is not necessary and you value the simple implementation. For hash tables, you have the fastest searching possible, with the downside being that you do not have access to ordering. Lastly, the binary search tree is generally faster than a vector, slower than a hash table, but can be organized. It's structured. And if correctly implemented, one of its major downsides can be dealt with. Assuming that we do not care for ordering, I would ultimately go with a hash table as it has the fastest search time complexity and is less likely to have implementation issues, assuming you can avoid collision.