**Nicholas Rusinski**

**Saba Jamaliannasrabadi**

**CS-300-R3285**

**Feb-16-2024**

**Project 1**

**Vector Example.**

Structure Course

String classId

String className

Vector<String> prerequisites <0>

Course()

Vector<Course> LoadCourses(path)

Vector<Course> courses

Open file as path

If file not open

Output "File not open"

Return courses

While getline(file) is true

Put line to stream

Create new Course structure as course

Parse stream until "," Set as course classId

Parse stream until "," Set as course className

While getline(iss) is not the end of the line

Add getline output from stream to prerequisites vector

Add course structure to the end of courses vector

Close file

Return courses

PrintAll(vector<Course>)

For (0 to vector size - 1)

Output "Course ID:", course ID

Output "Course name:", course name

If (prerequisites size is greater than 0)

For (0 to prerequisites size - 1)

Output "Prerequisite:", prerequisite

Else

Output "No prerequisites"

Partition(vector<Course> courses, int low, int high):

pivot = courses[high].className

i = low - 1

for j = low to high - 1:

if courses[j].className <= pivot:

i = i + 1

swap courses[i] with courses[j]

swap courses[i + 1] with courses[high]

return i + 1

QuickSortAlpha(vector<Course> courses, int low, int high):

if low < high:

pivot\_index = Partition(courses, low, high)

QuickSortAlpha(courses, low, pivot\_index - 1)

QuickSortAlpha(courses, pivot\_index + 1, high)

PrintCourseList(vector<Course>)

Partition(vector<Course>)

QuickSortAlpha(vector<Course>)

PrintAll(vector<Course>)

FindCourse(vector<Course>, searchName)

found = False

For each course in courses:

If course.className equals searchName:

Output "Course Name:", course.className

If course.prerequisites is not empty:

Output "Prerequisites:"

For each prerequisite in course.prerequisites:

Output prerequisite

Else:

Output "No prerequisites"

found = True

Break // Break out of loop after finding the first match

If not found:

Output "Course not found"

**Hash Table Example.**

Structure Course:

String classId

String className

Vector<String> prerequisites

Structure Node:

Key

Course

Next

Structure Hash:

Vector<Node> Nodes

Int size

Hash(key):

Return key % size

LoadCourses(path):

Open file as path

If file not open:

Output "File not open"

Return

While getline(file) is true:

Put line to stream

Create new Course structure as course

Parse stream until “,” Set as course classId

Parse stream until “,” Set as course className

While getline(iss) is not the end of the line:

Add getline output from stream to prerequisites vector

Key = hash(course.classId)

If Nodes[key] is null:

Create new node

Set node key to key

Store current course object in node

Set node next to null

Set Nodes[key] to node

Else:

Create new node

Set new node key to key

Store current course object in new node

Set new node next to Nodes[key]

Set Nodes[key] to new node

Close file

PrintAll():

For i = 0 to Nodes size - 1:

Node = Nodes[i]

While Node is not null:

Output Node key

Output Node Course classId

Output Node Course className

If (Node Course prerequisites size is greater than 0):

For j = 0 to Node Course prerequisites size - 1:

Output Node Course prerequisites[j]

Else:

Output "No prerequisites"

Node = Node next

AddToVectorInOrder(node, vector):

If node is not null:

AddToVectorInOrder(node left, vector)

Add node's data to vector

AddToVectorInOrder(node right, vector)

Partition(vector<Course> courses, int low, int high):

pivot = courses[high].className

i = low - 1

for j = low to high - 1:

if courses[j].className <= pivot:

i = i + 1

swap courses[i] with courses[j]

swap courses[i + 1] with courses[high]

return i + 1

QuickSortAlpha(vector<Course> courses, int low, int high):

if low < high:

pivot\_index = Partition(courses, low, high)

QuickSortAlpha(courses, low, pivot\_index - 1)

QuickSortAlpha(courses, pivot\_index + 1, high)

PrintCourseList(tree):

If tree root is null:

Output "No courses to display"

Return

vector<Course> sortedCourses

AddToVectorInOrder(tree root, sortedCourses)

Partition(sortedCourses, 0, sortedCourses.size() - 1)

QuickSortAlpha(sortedCourses, 0, sortedCourses.size() - 1)

For each course in sortedCourses:

Output course ID

Output course name

If (course prerequisites size is greater than 0):

For each prerequisite in course prerequisites:

Output prerequisite

Else:

Output "No prerequisites"

Delete temp vector

FindCourse(tree, searchName):

If tree root is null:

Output "No courses available"

Return

currentNode = tree Root

While currentNode is not null:

If searchName equals currentNode Course className:

Output currentNode Course className

If currentNode Course prerequisites size is greater than 0:

For each prerequisite in currentNode Course prerequisites:

Output prerequisite

Else:

Output "No prerequisites"

Return

If searchName is less than currentNode Course className:

currentNode = currentNode Left

Else:

currentNode = currentNode Right

Output "Course not found"

**Binary Search Tree Example**

Structure Course

classId

className

Vector<> prerequisites

Structure Node

Key

Course

Left

Right

Structure Tree

Root

LoadCourses(path)

Open file as path

If file not open

Output “File not open”

Return

While getline(file) is true

Create new Course structure as course

Parse line from file into course data

If course data is not valid

Output “Invalid data format”

Continue to next line

If course has prerequisites

For each prerequisite in course

If prerequisite not found in courses vector

Output “Prerequisite not found”

Continue to next line

Add course to tree

Close file

AddCourse(tree, course)

If tree root is null

Create new node with course as data, set as root

Else

AddNode(tree root, course)

AddNode(node, course)

If course classId < node Key

If node left is null

Create new node with course as data, set as node left

Else

AddNode(node left, course)

Else

If node right is null

Create new node with course as data, set as node right

Else

AddNode(node right, course)

PrintAll(tree)

If tree root is null

Output “No courses to display”

Return

TraverseTree(tree root)

TraverseTree(node)

If node is not null

TraverseTree(node left)

Output node course information

If node course has prerequisites

Output prerequisites

TraverseTree(node right)

AddToVectorInOrder(node, vector)

If node is not null:

AddToVectorInOrder(node left, vector)

Add node's data to vector

AddToVectorInOrder(node right, vector)

Partition(vector<Course> courses, int low, int high):

pivot = courses[high].className

i = low - 1

for j = low to high - 1:

if courses[j].className <= pivot:

i = i + 1

swap courses[i] with courses[j]

swap courses[i + 1] with courses[high]

return i + 1

QuickSortAlpha(vector<Course> courses, int low, int high):

if low < high:

pivot\_index = Partition(courses, low, high)

QuickSortAlpha(courses, low, pivot\_index - 1)

QuickSortAlpha(courses, pivot\_index + 1, high)

PrintCourseList(tree)

If tree root is null

Output "No courses to display"

Return

vector<Course> sortedCourses

AddToVectorInOrder(tree root, sortedCourses)

Partition(sortedCourses, 0, sortedCourses.size() - 1)

QuickSortAlpha(sortedCourses, 0, sortedCourses.size() - 1)

For each course in sortedCourses:

Output course ID

Output course name

If (course prerequisites size is greater than 0)

For each prerequisite in course prerequisites:

Output prerequisite

Else

Output "No prerequisites"

Delete temp vector

FindCourse(tree, searchName)

If tree root is null

Output "No courses available"

Return

currentNode = tree Root

While currentNode is not null:

If searchName equals currentNode Course className:

Output currentNode Course className

If currentNode Course prerequisites size is greater than 0:

For each prerequisite in currentNode Course prerequisites:

Output prerequisite

Else:

Output "No prerequisites"

Return

If searchName is less than currentNode Course className:

currentNode = currentNode Left

Else:

currentNode = currentNode Right

Output "Course not found"

**Menu Example**

Exit\_Program():

Output "Exiting program..."

Exit

While True:

Output "Menu:"

Output "1. Load Data Structure"

Output "2. Print Course List"

Output "3. Print Course"

Output "4. Exit"

choice = Input "Enter your choice:"

Switch choice:

Case 1:

LoadCourses()

Break

Case 2:

PrintCourseList()

Break

Case 3:

searchName = Input "Enter the course name to search:"

FindCourse(searchName)

Break

Case 4:

Exit\_Program()

Break

Default:

**Run Time Analysis (Vector)**

| Loading course | Line Cost | # Times | Total Cost |
| --- | --- | --- | --- |
| Reading line | 1 | n | n |
| Creating course object | 1 | n | n |
| Adding course Object to vector | 1 | n | n |
| Total Cost | | | 3n |
| Runtime | | | O(n) |
| Printing All Courses | Line Cost | # Times | Total Cost |
| Iterating through the vector | 1 | n | n |
| Printing each course | 3 | n | n |
| Total Cost | | | 4n |
| Runtime | | | O(n) |
| Finding course | Line Cost | # Times | Total Cost |
| Reading line | 1 | n | n |
| Creating course object | 1 | n | n |
| Total Cost | | | 2n |
| Runtime | | | O(n) |

**Total Cost: 3n + 4n + 2n = 9n**

**Total Runtime: O(n) + O(n) + O(n) = O(n)**

**Advantages:**

Simple implementation.

Direct access to elements by index.

**Disadvantages:**

Insertion and deletion operations are expensive (O(n)) if done frequently.

Searching can be slow for large datasets (O(n)).

**Run Time Analysis (Hash)**

| Loading course | Line Cost | # Times | Total Cost |
| --- | --- | --- | --- |
| Hashing a course ID | 1 | n | n |
| Creating a node | 1 | n | n |
| Adding a node to the hash table | 1 | n | n |
| Total Cost | | | 3n |
| Runtime | | | O(n) |
| Printing All Courses | Line Cost | # Times | Total Cost |
| Iterating through the hash table | 1 | n | n |
| Printing each course in a bucket | 1 | m | m |
| Total Cost | | | n+m |
| Runtime | | | O(n+m) |
| Finding a Course | Line Cost | # Times | Total Cost |
| Hashing the search key | 1 | n | n |
| Accessing the appropriate bucket | 1 | n | n |
| Searching within the bucket | 1 | n | n |
| Total Cost | | | 3n |
| Runtime | | | O(n) |

**Total Cost = 3n + (n + m) + 3n + n + n = 7n + m**

**Runtime is O(n + m).**

**Advantages:**

Fast average-case lookup time (O(1)).

Efficient insertion and deletion (O(1) on average).

**Disadvantages:**

Worst-case lookup time can degrade to O(n) if many collisions occur.

Requires a good hash function to distribute keys evenly.

**Run Time Analysis (Binary Search Tree)**

| **Loading Courses** | Line Cost | # Times | Total Cost |
| --- | --- | --- | --- |
| Creating a Course object | 1 | n | 2 |
| Adding a node to the tree | 1 | n | 2 |
| Total Cost | | | 2n |
| Runtime | | | O(n) |
| **Printing All Courses** | Line Cost | # Times | Total Cost |
| Traversing the tree | 1 | n | n |
| printing each course | 1 | n | n |
| Total Cost | | | 2n |
| Runtime | | | O(n) |
| **Finding a Course** | Line Cost | # Times | Total Cost |
| Traversing the tree | Log n | n | Log n |
| Total Cost | | | Log n |
| Runtime | | | O(Log n) |

**Total Cost: 2n + 2n + log n**

**Runtime: O(n) + O(n) + O(log n)**

**Advantages:**

Provides sorted access to elements.

Average-case performance is good for searching (O(log n)).

**Disadvantages:**

Worst-case performance for insertion, deletion, and searching is O(n) if the tree becomes unbalanced.

**Recommendation:**

Based on the runtime analysis and the characteristics of the data structures, a Binary Search Tree (BST) emerges as the most suitable choice for storing and accessing course information in this project. The runtime complexity of a BST for common operations like insertion, deletion, and searching is O(log n) on average, which is more efficient compared to the Vector and Hash Table implementations. While the worst-case scenario for a BST can degrade to O(n) if the tree becomes unbalanced, it still offers competitive performance for typical use cases.

Furthermore, a BST provides sorted access to elements, allowing for efficient traversal and retrieval of course information in sorted order. This aligns well with the requirement to print course lists in sorted order based on course names. Additionally, the BST offers flexibility in sorting data upon entry, which can be advantageous for different sorting criteria, such as sorting alphabetically by course name.

In conclusion, considering the balance between runtime efficiency, sorted access, and flexibility in sorting, the Binary Search Tree is recommended as the optimal data structure for this project**.**