Thoura Chris Ouk

Southern New Hampshire University

CS-300 Data Analysis and Design

Professor Charles Griffith

10/07/25

Project One

**Milestone One Pseudocode Vector Data Structure:**

// Data structures to store courses and their titles  
// courseDependencies: Dictionary  
// allCourses: List of all courses  
  
// Data structures  
courses = new List() // All courses  
dependencies = new Dictionary() // Course -> Prerequisites  
  
// Add courses and prerequisites  
courses.add("CSCI100")  
dependencies.put("CSCI100", new List())  
  
courses.add("CSCI101")  
dependencies.put("CSCI101", new List("CSCI100"))  
  
courses.add("CSCI200")  
dependencies.put("CSCI200", new List("CSCI101"))  
  
courses.add("MATH201")  
dependencies.put("MATH201", new List())  
  
courses.add("CSCI300")  
dependencies.put("CSCI300", new List("CSCI200", "MATH201"))  
  
courses.add("CSCI301")  
dependencies.put("CSCI301", new List("CSCI101"))  
  
courses.add("CSCI350")  
dependencies.put("CSCI350", new List("CSCI300"))  
  
courses.add("CSCI400")  
dependencies.put("CSCI400", new List("CSCI301", "CSCI350"))  
  
// Function to check if a course can be taken  
function canTake(course, taken):  
prereqs = dependencies.get(course)  
for p in prereqs:  
if p not in taken:  
return false  
return true  
  
// Example usage  
takenCourses = ["CSCI100", "CSCI101", "MATH201"] // List of already taken courses  
courseToCheck = "CSCI300"  
  
if canTake(courseToCheck, takenCourses):  
print(courseToCheck + " can be taken")  
else:  
print(courseToCheck + " cannot be taken yet.")

**Milestone Two Pseudocode Hash Table:**

CLASS Bid  
STRING id  
STRING title  
STRING fund  
DOUBLE amount  
  
FUNCTION CreateBid(bidId, title, fund, amount)  
SET [this.id](http://this.id/) = bidId  
SET this.title = title  
SET this.fund = fund  
SET this.amount = amount  
RETURN this  
END FUNCTION  
END CLASS  
  
CLASS HashTable  
LIST nodes  
UINT tableSize  
  
FUNCTION Initialize(size)  
SET tableSize = size  
CREATE list of size tableSize  
END FUNCTION  
  
FUNCTION Hash(key)  
RETURN key MOD tableSize  
END FUNCTION  
  
FUNCTION Insert(bid)  
UINT key = Hash(atoi([bid.id](http://bid.id/)))  
IF nodes[key] is empty THEN  
SET nodes[key] = NEW LIST()  
END IF  
ADD bid to nodes[key]  
END FUNCTION  
  
FUNCTION PrintAll()  
FOR each node in nodes  
IF node is not empty THEN  
PRINT node's bid details  
END IF  
END FOR  
END FUNCTION  
  
FUNCTION RemoveBid(bidId)  
UINT key = Hash(atoi(bidId))  
IF nodes[key] is not empty THEN  
REMOVE bid from nodes[key]  
END IF  
END FUNCTION  
  
FUNCTION Search(bidId)  
UINT key = Hash(atoi(bidId))  
RETURN nodes[key] // Simplified return  
END FUNCTION  
END CLASS  
  
FUNCTION LoadBids(csvPath)  
OPEN CSV file at csvPath  
FOR each row IN CSV  
bid = CreateBid([row.id](http://row.id/), row.title, row.fund, row.amount)  
hashTable.Insert(bid)  
END FOR  
END FUNCTION  
  
FUNCTION Main()  
SET csvPath = "file.csv"  
HashTable bidTable  
bidTable.Initialize(100)  
  
LoadBids(csvPath)  
bidTable.PrintAll()  
END FUNCTION

**Milestone Three Pseudocode Binary Search Trees:**

Class Node:  
Attribute value  
Attribute leftChild  
Attribute rightChild  
  
Constructor(value):  
Set this.value = value  
Set this.leftChild = None  
Set this.rightChild = None  
  
Class BinarySearchTree:  
Attribute root  
  
Constructor():  
Set root = None  
  
Method insert(value):  
If root is None:  
root = Node(value)  
Else:  
insertRec(root, value)  
  
Method insertRec(currentNode, value):  
If value < currentNode.value:  
If currentNode.leftChild is None:  
currentNode.leftChild = Node(value)  
Else:  
insertRec(currentNode.leftChild, value)  
Else:  
If currentNode.rightChild is None:  
currentNode.rightChild = Node(value)  
Else:  
insertRec(currentNode.rightChild, value)  
  
Method search(value):  
return searchRec(root, value)  
  
Method searchRec(currentNode, value):  
If currentNode is None:  
return False  
If value == currentNode.value:  
return True  
If value < currentNode.value:  
return searchRec(currentNode.leftChild, value)  
Else:  
return searchRec(currentNode.rightChild, value)  
  
Method inOrderTraversal():  
inOrderRec(root)  
  
Method inOrderRec(currentNode):  
If currentNode is not None:  
inOrderRec(currentNode.leftChild)  
Print currentNode.value  
inOrderRec(currentNode.rightChild)

END CODE.

I just resubmitted the three original pseudocodes that I submitted before in this course with changes to make them better and clearer to understand. They will all have the same purpose and outcome when making code based from them. Each data structure has its own strengths and weaknesses. The structure that will be chosen will depend on the specific needs of the application. Things such as expected size of the dataset, performance requirements and whether the data is static or dynamic. For example, a bid management necessity would require a Hash Table. A Binary Search Tree would be beneficial when trying to maintain sorted order.

**Pseudocode for printing out courses in Alphanumeric order:**

FUNCTION PrintCoursesInorder(courses):

// Sort the list of courses in alphanumeric order

sortedCourses = Sort(courses):

// Print each course in the sorted list

FOR each course IN sortedCourses:

PRINT Course

END FUNCTION

FUNCTION Sort(courses):

// Implementing a simple sorting algorithm

// Use built in sort for simplicity

RETURN courses.sort() // This will sort the courses in alphanumeric order

END FUNCTION

// Example usage

Courses = [“CSCI100”, “CSCI200”, “CSCI101”, “CSCI300”, “MATH201”, “CSCI350”, “CSCI400”, “CSCI1301”]  
PrintCoursesInOrder(courses)

END

Pseudocode made based off of the requirement of the assignment.

**Pseudocode for a Menu:**

// Data structures

Courses = new List()

Prerequisties = new Dictionary()

FUNCTION Main():

Choice = 0

WHILE choice != 9:

PRINT “Load Data”

PRINT “Print Course”

PRINT “Exit”

PRINT “Enter Choices”

INPUT choice

IF choice == 1 THEN

PRINT “Enter file name:”

INPUT filename

FOR each line in file:

data = split line by comma

courseId = data

courses.add(coursed)

prerequisties.put(coursed, data[first to end])

END FOR

PRINT “Data loaded:”

ELSE IF choice == 2 THEN

PRINT “Enter course ID:”

INPUT course

PRINT course

PRINT “Prerequisites:”

Prereqs = prerequisities.get(courseId)

FOR each prereq in preqs:

PRINT prereq

END FOR

ELSE IF choice == 9 THEN

PRINT “Goodbye”

END IF

END WHILE

END FUNCTION

Pseudocode is made based on the requirement of the assignment.

**Runtime Analysis table for Vector, Hash Tables and Binary Search Trees:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Loading Data** | **Search** | **Sort/Print** |
| Vector | O(n) | O(n) | O(n log n) |
| Hash Table | O(n) | O(1) average | O(n) Traversal |
| Binary Search Tree | O(n) | O(h) (O(n) worst case | O(n) Traversal |
| Binary Search Tree (Balance) | O(n) | O(log n) | O(n) Traversal |
|  |  |  |  |

**Complete Runtime Comparison Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Load Data** | **Search Course** | **Print Alphanumeric** |
| Vector | O(n) | O(n) | O(n log n) |
| Hash Table | O(n) | O(1) average | O(n log n) |
| Binary Search Tree | O(n log n) | O(log n) | O(n) |

**Chart Table with Code, Line Cost, Number of Execution and Total Cost:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Cost** | **Times Run** | **Total** |
| Load File | 5 | 1 | 5 |
| Add Course | 1 | 8 | 8 |
| Search | 2 | 10 | 20 |
| Print | 3 | 1 | 3 |
|  |  |  |  |

Advantages and disadvantages of each structure. One advantage of the Vector structure is that it has straightforward insertion and access patterns. It also has good memory efficiency with good storage and no overhead for pointers. However, one disadvantage of this structure is the fact that it has a slow search operation, requiring an amount of time to find specific data. Hash Tables are good with direct access to course information using ID as the key. One advantage of the Hash Table structure is that it has no inherent ordering, meaning this will make printing in alphanumeric order more difficult. Binary Search Trees on the other hand has efficient search time and it is also good with organizing storage and reasonable search performance. Disadvantages of Binary Search Trees are the fact that they can become unbalanced; this would be degrading search performance if the courses are inserted in sorted order.

My recommendation for which structure to use would be based on what a person wants to achieve and what outcome they hope to get. However, I think the Hash Table would be best for this course management system. Hash table will provide the most balance based on performance, loading efficiency, acceptable print performance, scalability and implementation. Performance is important because we would need to search for specific courses to check prerequisites and print the course information. The hash table’s O(1) average search time would outperform both the Vector’s O(n) and the BST’s O(log n) for this operation. The loading efficiency is better than BST’s. So, the initial data load will be on the faster side. Print performance would mean that printing would be in alphanumeric order, but this is a less frequent operation compared to searching. The person would search many times but will only on occasion need the whole sorted list. The Hash table structure would do this better than the Vector and BST. Scalability will take care of the growing number of courses. Hash table will maintain consistent search performance. Hash table will also support standard libraries, making implementation straightforward and reliable without requiring custom balancing logic. For those reasons, I believe the hash table structure would make the most sense to use for this course project.