**Vector Pseudocode:**

vector<Courses> LoadCourse(string fileName) {

vector<Courses> courses

openFile(fileName)

first loop through the file to find all the courseNumbers

store the courseNumber in a vector of strings

vector<String> courseNumber

loop while (file is not end)

string line = file.NextLine

vector<String> words = split(line at ',')

if(words.size < 2) Then

return error to the user

Else

courseNumber.add(words[0])

closeFile(fileName)

openFile(fileName)

loop while (file is not end)

string line = file.NextLine

vector<String> words = split(line at ',')

if(words.size < 2) Then

return error to the user

Course course

course.CourseNumber = words[0]

course.CourseName = words[1]

if(words.size > 2) Then

loop for(int i = 2; i < size of words; i++)

check if (the prequesit is in the courseNumber vector) Then

course.prerequisites.add(words[i])

courses.add(course)

closeFile(fileName)

return courses

}

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printCourseInformation(Vector<Course> courses, String courseNumber) {

loop for(int i = 0; i < courses.size(); i++)

Course course = courses at i;

if the course.courseNumber is the same as courseNumber Then

print( CourseNumber + course.CourseName)

loop for each prerequisite of the course.prerequisites

printCourseInformation(courses, prerequisite);

}

**Hash Table Pseudocode:**

class HashTable {

struct Node {

Course course

integer key

Node next

}

Node() {

key = maximum value of integer

next = null

}

Node(Course pCourse) {

key = maximum value of integer

course = pCourse

next = null

}

Node(Course pCourse, integer pKey) {

course = pCourse

key = pKey

next = null

}

HashTable() {

node.size = tableSize

}

void insertCourse()

integer tableSize = 1000

node vector of Node

}

void HashTable::InsertCourse(Course course) {

create a hash key for the course

Get the Node in the vector node

if node is unused add it to the vector node Then

Create a newNode(course, key)

insert the newNode into the vector node

else

find the next open node

loop while( currentNode.next does not equal null)

currentNode = currentNode.next

Create a newNode(course, key)

Add it to the next node

currentNode.next = newNode

}

HashTable LoadCourse(String FileName) {

HashTable hashTable

openFile(fileName)

first loop through the file to find all the courseNumbers

store the courseNumber in a vector of strings

vector<String> courseNumber

loop while (file is not end)

string line = file.NextLine

vector<String> words = split(line at ',')

if(words.size < 2) Then

return error to the user

courseNumber.add(words[0])

closeFile(fileName)

openFile(fileName)

loop while (file is not end)

string line = file.NextLine

vector<String> words = split(line at ',')

if(words.size < 2) Then

return error to the user

Course course

course.CourseNumber = words[0]

course.CourseName = words[1]

if(words.size > 2) Then

loop for(int i = 2; i < size of words; i++)

check if (the prequesit is in the courseNumber vector) Then

course.prerequisites.add(words[i])

hashTable.InsertNode(course)

closeFile(fileName)

return hashTable

}

int numPrerequisiteCourses(Course course) {

totalPrerequisites = prerequisites of course c

loop for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printCourseInformation(Hashtable courses, String courseNumber) {

Generate the key for the courseNumber

create a new node that will hold the value found at the hashtable

newNode = courses.node[key]

if(node is empty) Then

then return null

Else

loop while (the newNode is not empty)

if(the newNode.course.courseNumber is the same as courseNumber) Then

return the newNode.course

Else

newNode equals the next node

}

**Binary Search Tree:**

class BinaryTree {

Node rootNode

}

BinaryTree LoadCourse(String FileName) {

BinaryTree binaryTree

openFile(fileName)

first loop through the file to find all the courseNumbers

store the courseNumber in a vector of strings

vector<String> courseNumber

loop while (file is not end) {

string line = file.NextLine

vector<String> words = split(line at ',')

if(words.size < 2) Then

return error to the user

courseNumber.add(words[0])

closeFile(fileName)

openFile(fileName)

loop while (file is not end)

string line = file.NextLine

vector<String> words = split(line at ',')

if(words.size < 2) Then

return error to the user

Course course

course.CourseNumber = words[0]

course.CourseName = words[1]

if(words.size > 2) Then

loop for(int i = 2; i < size of words; i++)

check if (the prequesit is in the courseNumber vector)

course.prerequisites.add(words[i])

binaryTree.Insert(course)

closeFile(fileName)

return binaryTree

}

void BinaryTree::Insert(Course course) {

if (the rootNode is empty) Then

rootNode = new Node(course);

Else

addNode(rootNode, bid);

}

void addNode(Node node, Course course) {

if (node.CourseNumber is less than course.CourseNumber) Then

if (node.leftPath is empty)

node.leftPath equals new Node(course)

else

addNode(node.leftPath, course)

else

if (node.rightPath is empty) Then

node.rightPath equals new Node(course)

else

addNode(node.rightPath, course)

}

void printCourseInformation(binaryTree tree, String courseNumber) {

Node currentNode = tree.rootNode

loop while (currentNode not empty)

if (currentNode.course.CourseNumber is equal to courseNumber) Then

print(currentNode.course.CourseNumber)

print(currentNode.course.CourseName)

if(there are prerequisites in the current course) then

for each prerequisite in currentNode.course.prerequisites)

printCourseInformation(tree, prerequisite)

if (currentNode.bidId is greater than BidId)

currentNode = currentNode.left

else

currentNode = currentNode.right

}

**Run Time Analysis**

**Vector:**

**Read and parse file**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Create vector course object | 1 | 1 | 1 |
| OpenFile(fileName) | 1 | 2 | 2 |
| Create vector courseNumber object | 1 | 1 | 1 |
| While( file is not end) | 1 | n | n |
| Create string line = nextLine | 1 | n | n |
| Vector<string> words = split(line at ‘,’) | 1 | n | n |
| If(words.size < 2) | 1 | n | n |
| Cource cource | 1 | n | n |
| course.courseNumer = words[0] | 1 | n | n |
| course.CourseName = words[1] | 1 | n | n |
| If(words.size > 2) | 1 | n | n |
| for(int i = 2; i < size of words; i++) | 1 | n | n |
| if (prequesit is in the courseNumber vector) | 1 | n | n |
| courses.add(course) | 1 | n | n |
| Close File | | 1 | 2 |
| Return course | | 1 | 1 |

**Hash Table**

**Read and parse file**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Create HashTable hashTable object | 1 | 1 | 1 |
| OpenFile(fileName) | 1 | 2 | 2 |
| Create vector courseNumber object | 1 | 1 | 1 |
| While( file is not end) | 1 | n | n |
| Create string line = nextLine | 1 | n | n |
| Vector<string> words = split(line at ‘,’) | 1 | n | n |
| If(words.size < 2) | 1 | n | n |
| Cource cource | 1 | n | n |
| course.courseNumer = words[0] | 1 | n | n |
| course.CourseName = words[1] | 1 | n | n |
| If(words.size > 2) | 1 | n | n |
| for(int i = 2; i < size of words; i++) | 1 | n | n |
| if (prequesit is in the courseNumber vector) | 1 | n | n |
| hashTable.InsertNode(course) | 2n+8 | 1 | 2n+8 |
| Close File | 1 | 2 | 2 |
| Return hashTable | 1 | 1 | 1 |
|  | | Total Cost | 12n+15 |
|  | | Runtime | O(n) |

**Insert Node:**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | Times Executed | Total Cost |
| Create a hash key for the course | 1 | 1 | 1 |
| Create a Node object pointing to the vector of nodes | 1 | 1 | 1 |
| if node is unused add it to the vector node | 1 | 1 | 1 |
| Create a NewNode(course, key) | 1 | 1 | 1 |
| Insert new node to the vector of nodes | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| while( currentNode.next does not equal null) | 1 | n | n |
| currentNode = currentNode.next | 1 | n | n |
| Create a NewNode(course, key) | 1 | 1 | 1 |
| currentNode.next = newNode | 1 | 1 | 1 |
|  | | Total Cost | 2n+8 |
|  | | Run time | O(n) |

**Binary Tree**

**Read and parse file:**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Create BinaryTree binaryTree object | 1 | 1 | 1 |
| OpenFile(fileName) | 1 | 2 | 2 |
| Create vector courseNumber object | 1 | 1 | 1 |
| While( file is not end) | 1 | n | n |
| Create string line = nextLine | 1 | n | n |
| Vector<string> words = split(line at ‘,’) | 1 | n | n |
| If(words.size < 2) | 1 | n | n |
| Cource cource | 1 | n | n |
| course.courseNumer = words[0] | 1 | n | n |
| course.CourseName = words[1] | 1 | n | n |
| If(words.size > 2) | 1 | n | n |
| for(int i = 2; i < size of words; i++) | 1 | n | n |
| if (prequesit is in the courseNumber vector) | 1 | n | n |
| binaryTree.Insert(course) | 2Logn+9 | 1 | 2Logn+9 |
| Close File | 1 | 2 | 2 |
| Return binaryTree | 1 | 1 | 1 |
|  | | Total Cost | 2Logn+10n+15 |
|  | | Runtime | O(nLogn) |

**Insert Course:**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| if (the rootNode is empty) | 1 | 1 | 1 |
| rootNode = new Node(course) | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| addNode(rootNode, bid) | 2Logn+6 | 1 | 2Logn+6 |
|  | | Total Cost | 2Logn+9 |
|  | | Runtime | O(nLogn) |

**Add Node:**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| if (node.CourseNumber is less than course.CourseNumber) | 1 | 1 | 1 |
| if (node.leftPath is empty) | 1 | 1 | 1 |
| node.leftPath equals new Node(course) | 1 | 1 | 1 |
| addNode(node.leftPath, course) | 1 | Logn | Logn |
| Else | 1 | 1 | 1 |
| if (node.rightPath is empty) | 1 | 1 | 1 |
| node.rightPath equals new Node(course) | 1 | 1 | 1 |
| Else | 1 | 1 | 1 |
| addNode(node.rightPath, course) | 1 | Logn | Logn |
|  | | Total Cost | 2Logn+6 |
|  | | Runtime | O(nLogn) |

**Main Menu Pseudocode:**

Int main()

{

integer choice = 0

while(choice is not equal to 4)

print("Load Data structure = 1")

print("Print Course List = 2")

print("print Couse = 3")

print("Exit = 4")

swtich(choice)

case 1:

call loadCourses method

break

case 2:

Call the vector quicksort method

Call the HashTable alphanumeric method

Call the binaryTree inOrder method

break

case 3:

input from the user taking in the courseNumber stored in

String courseNumber

call the search method to find the course that will also print if found in list

break

case 4:

print("Good bye")

break

default:

if the user enters a number not specified then

print("Unknown command")

}

**Print Alphanumeric:**

**Vector:**

integer partition(vector<Course> course, int begin, int end) {

integer low = begin

integer high = end

integer pivot = (end + begin) / 2

bool done = false

while (done is not true)

while (course.at(low).CourseNumber.compare(course.at(pivot).CourseNumber) < 0)

low++;

while (course.at(pivot).CourseNumber.compare(course.at(high).CourseNumber) < 0)

high--;

if (low >= high)

done = true

else

swap (the course.at(low) with course.at(high))

++low

--high

return high

}

void quickSort(vector<Course> course, int begin, int end) {

integer mid = 0

integer listSize = course.size()

if (listSize <= 1)

return

mid = partition(course, begin, end);

quickSort(course, begin, mid);

quickSort(course, mid + 1, end);

}

**Hash Table:**

void quickSort(vector<Course> course, int begin, int end)

{

integer mid = 0

integer listSize = course.size()

if (listSize <= 1)

return

mid = partition(course, begin, end);

quickSort(course, begin, mid);

quickSort(course, mid + 1, end);

}

print alphanumeric() {

vector<Course> course

for (int i = 0; i < nodes.size(); i++)

Node currentNode = &nodes[i]

course.add(currentNode)

if(node is not empty)

Node nextNode = currentNode.next

while(nextNode is not empty)

course.add(nextNode)

nextNode = nextNode.next

quickSort(course, 0 , size of course - 1)

for each course in course

print course information

}

**Binary Tree:**

void inOrder(Node node)

{

if(node is not empty) Then

Call inOrder(node.leftside)

Print(node.courseInformation)

Call inOrder(node.right)

}

**Evaluation:**

**Advantages & Disadvantages**

**Vectors:**

Advantages:

Using a vector as a data structure has many advantages to the developer and the user One important advantage is that the data is in a dynamic sized array, meaning it can change the size of the list if additional data is required to be stored. Also, vectors can check whether the program or user is attempting to look in the list at a position that is out of bounds if so, the program is able to return an error to the user. Finally, vectors handle all memory allocations and deallocations compared to arrays this saves the developer to have to worry about any memory leaks.

Disadvantages: As with its advantages, there are some disadvantages when it comes to using vectors. One of which is inserting and deleting data from the middle or beginning of the row of data every element after the insertion will need to be shifted, which can cost in performance.

**Hash Table:**

Advantages: One of the many advantages of using a hash table is that it is able to insert, delete, and search for nodes at an O(1) rate, no matter the size of the hash table. In addition, a hash table has direct access to the data that we would like to retrieve, as the data is stored using keys to identify where the data is stored.

Disadvantages: However, there are some disadvantages using a hash table one of which is if we are dealing with a large amount of data this can introduce a situation where values may be given the same key, and we would have to keep looking for the next available node in this index in order to store the value which can result to O(n) time costing performance.

**Binary Tree:**

Advantages: one advantage of using a binary tree is that it can perform insertions and deleting of nodes at a rate of O(Logn) when the binary tree is balanced which is more efficient than the other linear data structures. Moreover, another advantage is its ability to change size which is useful when new data needs to be added.

Disadvantages: The disadvantage of using a binary tree is if it is not designed well it can result in an unbalanced tree which will result in a more linear structure like that of a linked list which can result in operations such as insertion and deleting to take O(n) time. Another disadvantage is that there is no direct access to the values meaning there is no O(1) access as there is only O(logn) at best for balanced trees and O(n) at worst when the tree is unbalanced.

**Recommendation:**

For my recommendation, I recommend using Binary tree data structure for this project as it would suit the needs for the project. This includes some of the advantages of being able to resize the structure if more courses are added. Unlike linear data structures, like vectors and hash tables, we are given the ability to insert and delete nodes at a rate of O(logn) when the tree is balanced. Lastly, using this structure does allow data to be more organized when it is first created based on the value compared to the parent for example if the value is greater than the parent add node to the left otherwise add to the right.