**PROJECT ONE**

**// Read File**

void loadCourses(string csvPath, dataStructure){

use fstream variable to open file

make call to fstream to open,

if the return value is “-1”

file not found

else {

while (is not end of file)

For i=0, when i < the files row count, i++

CourseId = file[i][1]

CourseName = file[i][0]

If more info is available after number and name

prereqCourse = course number in that spot

If more info is still available

prereq2 = course number in that spot

// **Create course object**

struct Course {

string courseNumber;

string courseName;

vector<string> coursePrereqs;

};

**// Vector pseudocode**

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 printSampleSchedule(Vector<Course> courses) {

for all courses  
 print course name  
 if course has prerequisits   
 for each prereuisit  
 print prerequisit

}

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

for all courses

if the course is the same as courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

}

int partition(Vector<Course> courses, int begin, int end) {

low = begin

high = end

get mid

for all courses

while low is less than mid

++low

while mid is less than high

--high

if low >= high

return

else

swap low with high

++low

--high

}

void quickSort(Vector<Course> courses, int begin, int end) {

unsigned int mid = 0

if begin is greater than or equal to end

return

mid = partition courses from beginning to end

quickSort(courses, begin, mid)

quickSort(courses, mid+1, end)

}

**// Hashtable pseudocode**

int numPrerequisiteCourses(Hashtable<Course> courses) {

size = size of courses

totalPrerequisites = 0

key = courseNum

bucket = hash(course.key)

for all buckets in HashTable

if courses[bucket] is not empty and courses[bucket]->key==key

add prerequisites of p to totalPrerequisites

bucket = (bucket + 1) % size

return totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

key = courseNum

bucket = hash(course.key)

for all courses

if courses[bucket]->data is empty

bucketList = HashTable(hash item->key)

node = the new node

node->next = empty

node->data = courseNum

Append(bucketList, node) to list

else

return data at node

get user input course

for all courses

if the course at node->data has no collisions

print the courses[bucket]->data

else if the course at node->data has collisions

while (node->data is not null)

print node->data

node->next

else

go to next bucket

}

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

hashVal = courseNumber % key

for all courses

if bucket[courses] is the same as courseNumber

create new pointer node at head

print node->data

for each prerequisite of the course

print the prerequisite course information

}

int partition(Hashtable<Course> courses, int begin, int end) {

low = begin

high = end

get mid

node = new node at key

while node is not null

for all courses

while low is less than mid

++low

while mid is less than high

--high

if low >= high

return

else

swap low with high

++low

--high

}

void quickSort(Hashtable<Course> courses, int begin, int end) {

unsigned int mid = 0

if begin is greater than or equal to end

return

mid = partition courses from beginning to end

quickSort(courses, begin, mid)

quickSort(courses, mid+1, end)

}

**// Tree pseudocode**

int numPrerequisiteCourses(Tree<Course> courses) {

node = root of tree

totalPrerequisites = 0

while node is not null

for all prerequisites p in totalPrerequisites

if prerequisites in node->data == 0

return

else if prerequisites in node->data > 0

add p to totalPrerequisites

if node->right is not null

add p to totalPrerequisites

traverse right

else if node->left is not null

add p to totalPrerequisites

traverse left

return totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

node = root of tree

while node is not null

if key = node->key

return data at node + new line

else if key < node->key

traverse left

else

traverse right

}

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

curr = root of Tree

key = courseNumber

while curr is not null

if key = curr->key

if totalPrerequisites > 0

print course information for curr

print prerequisites

else

print course information for curr

else if key < curr->key

traverse left

else

traverse right

}

void printInOrder(Tree<Course> courses) {

new node at root

if node is null

return

printInOrder(node->left)

print node

printInOrder(node->right)

}

// Main method

int main() {

bool quit = false

print “1. Load Data Structure”

print “2. Print Course List”

print “3. Print Course”

print “9. Exit”

do {

switch

case 1:

initialize data structure (courses)

loadCourses(csvPath, courses)

break

case 2:

printSampleSchedule(courses)

break

case 3:

printCourseInformation(courses)

break

case 9:

quit = true

break

default:

print “Invalid Selection”

break

}while(quit = false)

}

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **totalPrerequisites = prerequisites of course c** | **1** | **1** | **1** |
| **for each prerequisite p in totalPrerequisites** | **1** | **n** | **n** |
| **add prerequisites of p to totalPrerequisites** | **1** | **1** | **1** |
| **print number of totalPrerequisites** | **1** | **1** | **1** |
| **for all courses** | **1** | **n** | **n** |
| **for all data in course information** | **1** | **n** | **n** |
| **print the course information** | **1** | **1** | **1** |
| **for all courses** | **1** | **n** | **n** |
| **if the course is the same as courseNumber** | **1** | **1** | **1** |
| **print out the course information** | **1** | **n** | **n** |
| **for each prerequisite of the course** | **1** | **n** | **n** |
| **print the prerequisite course information** | **1** | **1** | **1** |
| **Total Cost** | | | **6n + 1** |
| **Runtime** | | | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **HashTable** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Initialize key, bucket, totalPrerequisits** | **n** | **n** | **n** |
| **for all buckets in HashTable** | **1** | **1** | **1** |
| **if courses[bucket] is not empty** | **1** | **1** | **1** |
| **add prerequisites of p to totalPrerequisites** | **1** | **1** | **1** |
| **bucket = (bucket + 1) % size** | **1** | **1** | **1** |
| **for all courses** | **1** | **n** | **n** |
| **if courses[bucket]->data is empty** | **1** | **n** | **n** |
| **bucketList = HashTable(hash item->key)** | **1** | **n** | **n** |
| **node->next = empty** | **1** | **1** | **1** |
| **node->data = courseNum** | **1** | **1** | **1** |
| **Append(bucketList, node) to list** | **1** | **n** | **n** |
| **else** | **1** | **1** | **1** |
| **return data at node** | **1** | **1** | **1** |
| **for all courses** | **1** | **n** | **n** |
| **if the course at node->data has no collisions** | **1** | **1** | **1** |
| **print the courses[bucket]->data** | **1** | **n** | **n** |
| **else if the course at node->data has collisions** | **1** | **n** | **n** |
| **while (node->data is not null)** | **1** | **n** | **n** |
| **print node→data,node->next** | **1** | **1** | **1** |
| **else** | **1** | **1** | **1** |
| **go to next bucket** | **1** | **1** | **1** |
| **for all courses** | **1** | **n** | **n** |
| **if bucket[courses] is the same as courseNumber** | **1** | **1** | **1** |
| **create new pointer node at head** | **1** | **n** | **n** |
| **print node->data** | **1** | **1** | **1** |
| **for each prerequisite of the course** | **1** | **n** | **n** |
| **print the prerequisite course information** | **1** | **1** | **1** |
| **Total Cost** | | | **12n + 1** |
| **Runtime** | | | **O(n)** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **Node is equal to root of tree** | **1** | **1** | **1** |
| **totalPrerequisites = 0** | **1** | **1** | **1** |
| **while node is not null** | **1** | **n** | **n** |
| **for all prerequisites p in totalPrerequisites** | **1** | **n** | **n** |
| **if prerequisites in node->data == 0** | **1** | **1** | **1** |
| **else if prerequisites in node->data >** | **1** | **n** | **n** |
| **add p to totalPrerequisites** | **1** | **1** | **1** |
| **if node->right is not null** | **1** | **1** | **1** |
| **add p to totalPrerequisites** | **1** | **1** | **1** |
| **traverse right** | **1** | **1** | **1** |
| **else if node->left is not null** | **1** | **1** | **1** |
| **add p to totalPrerequisites** | **1** | **1** | **1** |
| **while node is not null** | **1** | **n** | **n** |
| **if key = node->key** | **1** | **1** | **1** |
| **return data at node + new line** | **1** | **1** | **1** |
| **else if key < node->key** | **1** | **1** | **1** |
| **traverse left** | **1** | **n** | **n** |
| **Else traverse right** | **1** | **1** | **1** |
| **while curr is not null** | **1** | **n** | **n** |
| **if key = curr->key** | **1** | **1** | **1** |
| **if totalPrerequisites > 0** | **1** | **1** | **1** |
| **print course information for curr** | **1** | **1** | **1** |
| **print prerequisites** | **1** | **1** | **1** |
| **else print course information for curr** | **1** | **1** | **1** |
| **else if key < curr->key** | **1** | **1** | **1** |
| **traverse left** | **1** | **n** | **n** |
| **else traverse right** | **1** | **n** | **n** |
| **new node at root** | **1** | **1** | **1** |
| **if node is null then return** | **1** | **1** | **1** |
| **printInOrder(node->left)** | **1** | **1** | **1** |
| **print node** | **1** | **1** | **1** |
| **printInOrder(node->right)** | **1** | **1** | **1** |
| **Total Cost** | | | **8n + 1** |
| **Runtime** | | | **O(n)** |

The three data structures have many similarities in terms of efficiency, but trees and hash tables generally perform better when handling larger datasets. They can quickly organize and search for data, making them suitable for managing large volumes of information. In this example, with only eight entries to consider, a vector data structure would be an excellent choice. However, when dealing with larger datasets, vectors are not as fast or efficient as trees and hash tables.

One downside of using a vector is that searching for a specific course requires checking each item until a match is found. On the other hand, the vector method is the quickest for reading the file and adding course objects, making it very user-friendly. The runtime for the vector method was the fastest among the three approaches, calculated as 6n+1.

Hash tables offer the advantage of rapid searches. You can quickly locate and print items using a key. However, creating the initial list and finding where to insert each course takes longer. Additionally, hash tables cannot be sorted directly; all values must be extracted, sorted, and printed separately to create an alphanumeric list of courses. For these reasons, hash tables may not be the best fit for this program.

Binary trees have the advantage of fast sorting speeds compared to vectors, making them quicker for sorting tasks. However, they are not as straightforward to implement as hash tables or vectors. Searching in a binary tree takes O(h) time, where h is the height of the tree.

For this specific project, I recommend using a vector data structure. Despite the time penalty for searching entries, the ability to quickly sort and print the entire catalog is more advantageous.