Patrick Chu

CS 300

Professor David Ostrowski

April 10, 2022

Project 1

//Define Course Structure

Class Course {

String courseNum;

String courseName;

Vector<string> preReqs;

Void PrintInfo(){

Output courseNum, courseName, all preReqs

}

**Vector**

//Print course info

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) {

}

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**

}

//Search for a course

Course Search(vector<Course> courses, string courseNum){

Create course object

For each course in courses:

If current course matches courseNum, return current course

Return empty course object

}

//open, read, store course info

Void loadCourses(string filename, Vector<string>\* schedule){

Create new vector for strings named classes

Initialize stream to read contents of file

Open filename

For each line in file

Push line into vector

Close filename

For each string in classes:

Create new course

Create list from string delimiting by “,”

If list size > 2

For each element > 2:

Push element to course PreReq list

Set courseNum to list index 0

Set courseName to list index 1

Else:

Set courseNum to list index 0

Set courseName to list index 1

Push course into schedule

}

**Binary Search Tree**

//Define BinarySearchTree

Class BinarySearchTree{

Struct Node {

Course course;

Node\* left;

Node\* right;

Node(){

Define left and right nodes

}

//Define default node

}

}

Void BinarySearchTree::inOrder(){

if(node is not null){

recursively travel to left sub-tree

call node’s course print

recursively travel to node’s right sub-tree

}

}

Void BinarySearchTree::insert(Course acourse){

If(root is null){

Set root to new course node

}

Else addnode with new acourse

}

Void BinarySearchTree::addNode(Node\* node, Course acourse){

If acourse < current node course number:

If left node is empty, add new course node in left node

Else recursively call addNode on left subtree

Else if node’s right child is empty, add new course node in right node

Else recursively call addNode on right subtree

}

Void BinarySearchTree::remove(string courseNum){

removeNode(root, courseNum)

}

Void BinarySearchTree::removeNode(Node\* node, string courseNum){

if node is null, return empty node

if courseNum < node course

recursively call removeNode on left child

else if courseNum > node course

recursively call removeNode on right child

else {

if node has node children, delete node

else if node has left child, delete node, set current ptr to left child

else if node has right child, delete node, set current ptr to right child

else left and right children exist, find middle most value of children (leftmost child of right child (temp), and set current node to that value, then delete temp node

}

}

Course BinarySearchTree::Search(String courseNum){

Set current node = root

While current node is not null:

If current node course matches courseNum

Return current courseNode

If current node course < courseNum

Set current node to left child

Else set current node to right child

}

//open, read, store course info

Void LoadCourses(string filename, BinarySearchTree \*tree){

Initialize filestream and string variables to read file and hold content of file

Open filename

Initialize count variable

For each line in file:

Create a new course object

Read line into new list, delimited by “,”

if listsize > 2:

set node courseNum to list index 0

set node courseName to list index 1

for all index >= 2, push into pre-req list

else:

set node courseNum to list index 0

set node courseName to list index 1

add course node to tree

}

**Hash Table**

//Define hashtable Structure

Const unsigned int DEFAULT\_SIZE = 8;

class Hashtable {

struct Node {

Course\* course;

Node\* next;

unsigned int key;

Node() {

key = UINT\_MAX;

next = nullptr;

}

Node(Course\* acourse): Node() {

course = acourse;

}

Node(Course\* acourse, unsigned int akey) : Node(acourse) {

key = akey;

}

};

unsigned int size = DEFAULT\_SIZE;

vector<Node> table;

};

//hash function

Unsigned int Hashtable::Hash(string courseNum){

return atoi(courseNum.c\_str() % size);

}

//Add course

Void Hashtable::Add(Course\* acourse){

Create key from acourse hashed courseNum

Create node pointer using key

If( current node is null){

Create new course node with passed argument and key

Insert node into hashtable

}

Else{

If(node key = UINT\_MAX){

Set node’s key to key

Set node’s course to acourse

Set node’s next to null

}else{

While(next node is not null){

Create new course node with passed argument and key

Set node-> next to new node

}

}

}

}

//Print course info

Void Hashtable::Print(){

For each node in table

If ( node key does not equal UINT\_MAX){

Print node key and call that node’s printinfo method

}

Create Node Pointer for next node

While (next node is not null){

Print node key and call that node’s printinfo method

Increment to the next node

}

}

//Search for a course

Course\* Hashtable:: Search(string courseNum){

Create course object

Create key for courseNum

For each node in hashtable:

If (node key matches search key):

Return matching node course

If no matching course found, return empty course object

}

//open, read, store course info

Void LoadCourses(string filename, Hashtable &table){

Initialize filestream and string variables to read file and hold content of file

Open filename

Initialize count variable

For each line in file:

Create a new course object

Read line into new list, delimited by “,”

if listsize > 2:

set node courseNum to list index 0

set node courseName to list index 1

for all index >= 2, push into pre-req list

else:

set node courseNum to list index 0

set node courseName to list index 1

add course node to table

}

**Print computer science courses in alphanumeric order**

Void PrintAlphaNumeric(){

Create list for strings

Iterate across all data points of datatype(vector, Binary Tree, Hashtable), push course name into list

Sort list

Print list from beginning to end

}

**Menu**

Create Schedule object(vector/hashtable/binarytree)

Declare and initialize string for courseNum

Declare and initialize Course aCourse

Declare and initialize choice variable to 0

While (choice does not = 4):

Output: menu

Output: 1. Load Courses

Output: 2. Display Courses

Output: 3. Print course information

Output: 4. Exit

Switch(choice):

Case 1:

LoadCourses(filename, schedule object)

Break;

Case 2:

Void PrintAlphaNumeric()

Break;

Case 3:

Ask for courseNum

PrintCourseInformation(courseNum)

Break;

**Analysis**

**Vector**

|  |  |  |  |
| --- | --- | --- | --- |
| **Reading File and loading courses** | **Line cost** | **#Times executed** | **Total Cost** |
| Create new vector for strings named classes | **1** | **1** | **1** |
| Initialize stream to read contents of file | **1** | **1** | **1** |
| open file | **1** | **1** | **1** |
| For each line in file | **1** | **1** | **1** |
| Push line into vector | **1** | **n** | **n** |
| Close filename | **1** | **1** | **1** |
| For each string in classes: | **1** | **1** | **1** |
| Create new course | **1** | **n** | **n** |
| Create list from string delimiting by “,” | **1** | **2n** | **2n** |
| If list size > 2: | **1** | **n** | **n** |
| For each element > 2: | **1** | **n** | **n** |
| Push element to course PreReq list | **1** | **2n** | **2n** |
| Set courseNum to list index 0 | **1** | **n** | **n** |
| Set courseName to list index 1 | **1** | **n** | **n** |
| Else: | **1** | **n** | **n** |
| Set courseNum to list index 0 | **1** | **n** | **n** |
| Set courseName to list index 1 | **1** | **n** | **n** |
| Push course into schedule | **1** | **n** | **n** |
|  | | | **14n+6** |
|  | | | **O(n)** |

**Hashtable**

|  |  |  |  |
| --- | --- | --- | --- |
| **Reading File and loading courses** | **Line cost** | **#Times executed** | **Total Cost** |
| Initialize filestream and string variables to read file and hold content of file | **1** | **1** | **1** |
| open file | **1** | **1** | **1** |
| For each line in file | **1** | **1** | **1** |
| Create new course | **1** | **n** | **n** |
| Create list from line delimiting by “,” | **1** | **2n** | **2n** |
| If list size > 2: | **1** | **n** | **n** |
| For each element > 2: | **1** | **n** | **n** |
| Push element to course PreReq list | **1** | **2n** | **2n** |
| Set courseNum to list index 0 | **1** | **n** | **n** |
| Set courseName to list index 1 | **1** | **n** | **n** |
| Else: | **1** | **n** | **n** |
| Set courseNum to list index 0 | **1** | **n** | **n** |
| Set courseName to list index 1 | **1** | **n** | **n** |
| add course node to table | **1** | **n** | **n** |
| Close filename | **1** | **1** | **1** |
|  | | | **9n+6** |
|  | | | **O(n)** |

**Binary Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| **Reading File and loading courses** | **Line cost** | **#Times executed** | **Total Cost** |
| Initialize filestream and string variables to read file and hold content of file | **1** | **1** | **1** |
| open file | **1** | **1** | **1** |
| For each line in file | **1** | **1** | **1** |
| Create new course | **1** | **n** | **n** |
| Create list from line delimiting by “,” | **1** | **2n** | **2n** |
| If list size > 2: | **1** | **n** | **n** |
| For each element > 2: | **1** | **n** | **n** |
| Push element to course PreReq list | **1** | **2n** | **2n** |
| Set courseNum to list index 0 | **1** | **n** | **n** |
| Set courseName to list index 1 | **1** | **n** | **n** |
| Else: | **1** | **n** | **n** |
| Set courseNum to list index 0 | **1** | **n** | **n** |
| Set courseName to list index 1 | **1** | **n** | **n** |
| Add course node to tree | **1** | **n** | **n** |
| Close filename | **1** | **1** | **1** |
|  | | | **9n+6** |
|  | | | **O(n)** |

**Advantages and Disadvantages**

Vectors are easy to implement. They have a constant time insertion if inserted in the back, and can be quickly searched in O(logn) time. However, they must be sorted in order to search quickly, and removing items from a vector causes shifting and is more resource intensive than other structures.

Hashtables are incredibly quick and efficient when utilized properly, and will insert or delete in constant time. However, they are resource intensive and do not store order of elements.

Binary search trees are very quick to access and will easily retrieve items in order. They can also insert and delete in O(logn­) time.

For this particular project, I would recommend using vectors. While I believe that Hashtables or binary search trees would be more efficient were the data set larger than it currently is, vectors are more easily implemented from a coding perspective, and the difference in run time for data sets as small as this project requires would make a negligible real-world difference. Therefore, the time saved on using a less complicated data structure would result in a more efficient project implementation.