Brittney Miller

SNHU

CS 300

Psuedocode and Runtime Project 1

File reading:

Usefstream to open file

Create method void loadCourses(string csvPath, dataStructre)

Call to open file, if return value is “-1”, file is not found

Else file found

While not End of File

Read each line

If less than two values in line, return ERROR

Else read parameters

If there is a third or more parameter

If third or more parameter in first parameter elsewhere,

Continue

Else return Error

Close file

Hold Course Info:

Create struct Course{}

Create Identifiers: Course ID, Course Name, Prerequisite

//Vector

Vector<Course>loadCourses(string csvPath)

for(int i=0; I < file.rowCount(); i++) {

Create data structure

Add to collection of courses

Course course;

course.courseId = file[i][1];

course.name = file[i][0];

while not end of line

course.prereq. = file[i][8];

course.push\_back(course);

//Hashtable

Create Hashtable

Create Node Struct

Course course

Unsigned int key

Vector<Node> nodes

Define tableSize

Unsigned int has(int key)

Create insert method void HashTable::Insert(Course course)

Create key for given course

Search for node with key value

If no entry found for key

Assign current node to key position

Else if node is used

Assign old node key UNIT\_MAX, set to key, set old node to course and old node next to

Null pointer

Else find next open node

Add new newNode to end

Void loadCourses(string csvPath, HashTable\* hashTable)

Loop read rows of CSV file

For (unsigned int i=0; I <file.rowCount(); i++) {

Create data structure

Add to collection of courses

Course course;

course.courseId = file[i][1];

course.name = file[i][0];

while not end of line

course.prereq. =file[i][8];

hashTable->Insert(course);

//Tree

Define binary search tree to hold all courses

BinarySearchTree\*bst;

Bst = new BinarySearchTree();

Course course;

Create add node method

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

If root equals null

Add root

If node is less than root

Add to left

If no left node

Current node becomes left node

If node is greater than root

Add to right

If no right node

Current node becomes right node

Void loadCourses(string csvPath, BinarySearchTree\* bst)

Loop read rows of CSV file

For(unsigned int i=0; i< file.rowCount(); i++) {

Create data structure

Add to collection of courses

Course course;

course.courseId = file[i][1];

course.name = file[i][0];

while not end of line

course.prereq. = file[i][8];

bst->Insert(course);

Print Course Info and Prereq:

//Vector

Create method

Void printCourseInformation(Vector<Course> courses, String courseId)

Get user input courseId

While vector does not equal empty

If input is same as courseId

Output course.courseId << output course.name

While (prereq = true)

Output course.prereq

//HashTable

Create method

Void printCourseInformation(Hashtable<Course> courses, String courseId)

Get user input courseId

Assign key = courseId

Assign node to node.at(key)

If current node matches key

Return course, displayCourse(nodes[key].course)

If node equals null

Return null

Else while node does not equal Null, check against key

If key matches courseId

Return course, displayCourse(nodes[key].course)

Point to next node

//Tree

Create method

Void printCourseInformation(Tree<Course> courses, String courseId)

Get input for courseId

Assign current node to root

While current node does not equal NULL

If course.courseId matches current

Return current

Output course.courseId << output course.name

While (prereq = true)

Output course.prereq

If courseId is less than root

Set current node to left

Else set current node to right

Menu:

Set to 0;

Create while loop for menu

While choice does not equal 4

Ouput menu choices (1.Load Course file, 2.Print course list 3. Print Individual Course, 4.Exit)

Create switch(choice)

Case 1: loadCourses(courseFile, dataStructure)

Case 2: printSorted(courses)

Case 3: printCourseInformation(courseId)

Case 4: Exit Program

Print Sorted List:

//Vector

Create sorted print method printSorted(courses)

Create partition method int partition(vector<Course>& courses, int begin, int end)

Set lowIndex to first element

Set highIndex to last element

Set midpoint to lowIndex + (highIndex = lowIndex) /2

Set pivot to midpoint

Decrement highIndex while pivot is less than highIndex

Swap lower values to left of pivot, higher values to right of pivot

Set temp value to lowIndex

Set lowIndex to highIndex

Set highIndex to temp

Create quicksort method

Void quickSort(vector<Course>& courses, int begin, int end)

Set mid to 0, lowIndex to beginning, highIndex to end

If beginning >= end

Return

Set lowEndIndex to partition(courses, lowIndex, highIndex)

Make recursive call to quicksort

quickSort(courses, lowIndex, lowEndIndex);

quickSort(courses, lowEndIndex + 1, highIndex)

Create display course method

Void displayCourse(Course course) {

Cout << course.courseId << “: “ << course.name << “ | “ << course.prereq << endl;

Loop through vector display courses

For (int i=0; i<course.size(); ++i)

displayCourse(courses[i])//Tree

Create inOrder method

Void BinarySearchTree::inOrder(Node\* node)

If(node != Null)

Check most left side first

inOrder(node->left)

cout<<course.courseId << “: “ << course.name << “ | “ << course.prereq << endl;

check next right leaf

inOrder(node->right)

cout<<course.courseId<<”: “<<course.name << “ | “ <<course.prereq << endl;

|  |  |  |  |
| --- | --- | --- | --- |
| Vector | Line Cost | Times Executed | Total Cost |
| Create Vector | 1 | 1 | 1 |
| For each line in file | 1 | N | N |
| Create Vector course item | 1 | N | N |
| Create Vector | 1 | 1 | 1 |
| While prereq exists | 1 | N | N |
| Append preerq | 1 | N | N |
| Pushback course item | 1 | N | n |

Total: 5n + 1

Runtime: O(n)

|  |  |  |  |
| --- | --- | --- | --- |
| HashTable | Line cost | Times Executed | Total Cost |
| Create hash table | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| Create key for course | 1 | n | N |
| If no entry for key found | 1 | N | N |
| Assing node to key | 1 | N | N |
| Else | 1 | N | N |
| Assign UNIT\_MAX set key set old node course and old node to null pointer | 4 | N | 4n |
| Else | 1 | N | N |
| Find next open node | 1 | N | N |
| Add new newNode to end | 1 | N | N |
| For each line in file | 1 | N | N |
| Create vector course item | 1 | N | N |
| While prereq exists | 1 | N | N |
| Append prereq | 1 | N | N |
| Insert course item | 1 | N | n |

Total: 16n + 1

Runtime: O(n)

|  |  |  |  |
| --- | --- | --- | --- |
| Tree | Line cost | Times executed | Total cost |
| Create tree | 1 | 1 | 1 |
| Add node method | 0 | 0 | 0 |
| If root is null add root | 1 | 1 | 1 |
| If node less than root add left | 1 | N | N |
| If no left node | 1 | N | N |
| Current node becomes left | 1 | N | N |
| If node greater than root add right | 1 | N | N |
| If no right node | 1 | N | N |
| Current node becomes right | 1 | N | N |
| For each line in file | 1 | N | N |
| Create vector course item | 1 | N | N |
| While prereq exists | 1 | N | N |
| Append prereq | 1 | N | N |
| Insert course item | 1 | n | n |
|  |  |  |  |

Total: 11n + 2

Runtime: O(n)

There are both pros and cons for each data structure for the requirements of the program. When looking at the vector method it’s the fastest in reading files and adding course objects. The vector method is typically straightforward when appending to add items to the file. When looking at all the methods they all shared the O(n) notation for runtime but vector had the smallest total cost at 5n + 1.

When using Vector the only issue I can see is when searching lists for a single specific course. It takes time and the run time is increased since it has to search the list one by one until a match is found.

Hash tables on the other hand can search and filter through a list fairly quickly. Hash tables use a created key to pull locations of courses in a list and quickly search and print information.

The only disadvantage is the amount of time it takes to initially create the list since we create a key to search the list that means we have to assign each individual course a unique key. Hash tables also lack the ability to be sorted which means if you wanted to print in a certain order like alphabetical or numerical each key would be searched, extracted, sorted then printed which would take more time than necessary or wanted which means Hashtabels are most likely not the best data structure to use for printing school course information.

Binary trees are the fastest when it comes to search time, even faster than vectors. While searching a course is fairly easy and straightforward how the program runs down the tree until the match is found it may be quicker than a vector but is not as easy to follow as a hashtable. It is possible that thee tree would be stuck searching every single element if the tree only had one side leaves like left leaves. That means the annotation of runtime would switch from O(n) to O(h) or height of the tree which means it could be greatly increased.

Looking at all the pros and cons of these tree data structures it seems Vector will be the best option for this project. Mainly because while it’s not the fastest it is fast and there is no risk of an exponential growth of runtime based on how you input or search for course information. Where in binary search if you end with all left leaves this would be the case. Vector also allows you to sort quickly and print the entire course catalog which may be something a student would need especially when choosing which classes to pick.