**Pseudo Code Week 6**

By: Ray Clifford

Date: 02/18/24

CS-300-R3284

DSA Analysis and Design

Professor: Jamalian, Saba

Pseudocode

//Reading file

Use fstream to open file;

//Create a method

void loadCourses(string csvPath, dataStructure) {

//Make a call to open the file

if(returned value <= -1) {

file is not found;

}

else {

file found;

}

}

//While loop to check if it is the end of the file

while(not end of file) {

//Read each line

if(values < 2) {

return error;

}

else {

read parameters;

if(there are 3 or more parameters) {

if (the third or more parameter is in the first parameter elsewhere) {

continue;

}

else {

return error;

}

close file;

}

//Course information holder

//create structure

struct Course{}

//Creating identifiers

Course id;

Course name;

Prerequisite;

//Creating a Vector

vector<Course>loadCourses(string csvPath)

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

//Creating a Data structure and adding it to the collection of courses

Course course;

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

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

//While loop to check if it is the end of the line

while(not end of line) {

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

courses.push\_back(course);

}

//Creating a Hash Table

//Create Hash Table

Create Node struct

Course course

unsigned int key;

Vector<Node> nodes

define tableSize;

unsigned int has (int key)

//Create an Insert Method

void HashTable::Insert(Course course)

//Create the key for the course given, search for Node with key value

if(no entry is found for the key) {

assign this Node to the key

else if(Node is being used)

assign oldNode key to UNIT\_MAX, set to key, set oldNode to course, and oldNode next to null pointer;

}

else {

find the next open Node;

add new newNode to end;

}

}

void loadCourses(string csvPath, HashTable\* hashTable)

//Create a loop to read the rows of the csv file

for(unsigned int i; i < file.rowCount(); i++)

//Creating a Data structure and adding it to the 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 a BST to hold all courses

BinarySearchTree\* bst;

bst = new BinarySearchTree();

Course course;

//Create an addNode method

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

if root is null, then add a root

if node is less than root , add to the left

if no left node, this node becomes left

if node is greater than root, add to the right

if no right node, this node becomes right

void loadCourses(string csvPath, BinarySearchTree\* bst)

//Create a loop to read rows from a csv file

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

//Creating a Data structure and adding it to the 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 Information and Prerequisites

//Vector

//Creating a method for printing

void printCourseInformation(vector<Course> courses, String courseId)

get input for courseId ;

//Create a loop to check if the vector is empty

While(vector not empty) {

if(input == coursed) {

output course.courseId << output course.name;

while(prereq = true) {

output course.prereq;

//Hash Table

//Creating method

void printCourseInformation(HashTable<Course> courses, String courseId)

//Get input for courseId

Assign key = courseId

Assign node to the node.at(key)

if(current node matches key) {

return course, displayCourse(nodes[key].course);

if(node points to null) {

return null;

}

else {

while(node is not equal to null) {

check against the key

}

if(key matches coursed) {

return Course, displayCourse(nodes[key].course);

point to next node;

//Tree

//Creating method

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

get input for courseId

Assign current node to root

//Create a loop to check if the current node is equivalent to null

while(current is not equal to null) {

if(course.courseId matches current) {

return current, output course.courseId << output course.name;

}

}

//Create another while loop to check the prerequisite

while(prereq is equal to true) {

output course.prereq

if(courseId < root) {

set current to left;

}

else {

set current to right;

}

//Menu

Set choice to 0;

//Create a while loop for the menu

while(choice is not equal to 4) {

output Menu choices(1. Load Course File., 2. Print Course List., 3.Print Individual Course., 4. End Program.)

//Create switch(choice)

Case 1: Load Courses(courseFile, dataStructure) //you can use tree, binary or hash here

Case 2: printSorted(courses) //here call the function to print the sorted class list

Case 3: printCourseInformation(courseId)

Case 4: End program

//Printing the sorted list

//Vector

//Creating a print sorted method

printSorted(courses)

//Creating a partitioning 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 the left of the pivot, higher values to the right

set temp value to lowIndex

set lowIndex to highIndex

set highIndex to temp

//Create a quickSort method

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

set mid to 0, lowIndex to begin, highIndex to end

if(begin >= end) {

return;

set lowEndIndex to partition(courses, lowIndex, highIndex)

//Make a recursive call to quickSort

quickSort(courses, lowIndex, lowEndIndex)

quickSort(courses, lowEndIndex + 1, highIndex)

//Create displayCourse method

void displayCourse(Course course) {

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

//Create a loop to go through the vector to display courses

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

displayCourse(courses[i]) //Tree

//Create an inOrder method

void BinarySearchTree::inOrder(Node\* node)

//Check the very left side first

if(node != null) {

inOrder(node->left);

//Check the next right leaf

inOrder(node->right)

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

Run time analysis for reading file and creating course objects:

|  |  |  |  |
| --- | --- | --- | --- |
| Vector | Line cost | # of times it runs | Total cost |
| Create a vector | 1 | 1 | 1 |
| For each of the lines  in the file | 1 | n | n |
| Create vector course  item | 1 | n | n |

|  |  |  |  |
| --- | --- | --- | --- |
| Vector | Line cost | # of times it runs | Total cost |
| Create vector | 1 | 1 | 1 |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| Pushback course item | 1 | n | n |
|  |  | Total cost | 5n + 1 |
|  |  | Runtime | 0(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| Hash Table | Line cost | # of times it runs | Total cost |
| Create Hash Table | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| Create key for course | 1 | n | n |
| if no entry found for key | 1 | n | n |
| Assign node to key | 1 | n | n |
| else | 1 | n | n |
| find the 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 cost | 16n + 1 |
|  |  | Runtime | 0(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| Tree | Line cost | # of times it runs | Total cost |
| Create Tree | 1 | 1 | 1 |
| Add node method | 0 | 0 | 0 |
| if root is null, add root | 1 | 1 | 1 |
| if node is less than root then add to left | 1 | n | n |
| if no left node | 1 | n | n |
| this node becomes left | 1 | n | n |
| if node is greater than root, add to the right | 1 | n | n |
| If no right node | 1 | n | n |
| this 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 cost | 11n + 2 |
|  |  | Runtime | 0(n) |

As I learned while taking this course, each structure has advantages and disadvantages. According to the tables above the Vector has the fastest runtime at 5n +1. While this may be great for runtime it can have it’s disadvantages too. For instance, if you need to search for an item, it will have to parse through the entire list.

Hash tables on the other hand, have the ability to search quickly. By creating a key, the locations of a given course will be known and easily printed. The downside of this is that for each item a key has to be created and a spot found to insert each course. Hash tables are also not good for sorting. In order to print an alpha numeric of all the courses, it has to extract, sort and print each one. With a program such as this one, this would be worst case scenario.

Binary trees can search faster than a vector, knowing the course ahead of time makes it easier to search down a tree until a match is found. Worst case scenario in a tree with only right branches is the runtime would then by h(n) where h would be the height of the tree.

I think after reviewing these options I would have to go with a vector. Being able to quickSort and print the entirety of the file is a must have for a program such as this. It has the downside of having to search the entire list, but it has the fastest runtime. Overall I would recommend this data structure for the project.