Pseudocode Document

Fred Wahab

CS-300

//initialize global objects and variables

struct Course {

string courseNumber;

string courseName;

vector<string> coursePrereqs;

};

vector<Course> course;

**VECTOR**

**//LOAD VECTOR**

void LoadVector() {

//local variables

string fileName;

//get filename

output: “Please enter name of file: “;

input: fileName;

//open file

ifstream infile;

infile.open(fileName);

//insert data into courses

if infile is open {

string line;

//read file line by line

while there is a line infile {

vector <string> tokens;

string token;

stringstream ss(line);

//tokenize every string at the comma

while reading the string, seperate at ‘,’ {

push each token to tokens

}

//ensure there are at least two parameters on each line

if size of tokens is less than 2 {

break;

}

//create a new course and add data from tokens

Course newCourse equals Course();

newCourse.courseNumber equals tokens[0];

newCourse.courseName equals tokens[1];

//if prereqs exist in tokens, add to course

if size of tokens is greater than 2 {

for every token[i] after the first 2 {

//ensure any prerequisite that is provided on a line exists

//as a course in the file

for every course[j] in courses {

if courses[j].courseNumber equals token[i] {

push token to newCourse.coursePrereqs

}

}

}

}

push newCourse to courses

}

output: “Data Structure successfully loaded!”;

}

//if the fileName does not exist

else {

output: “Error: Filename not found.”;

}

infile.close();

}

**//PRINT COURSE LIST**

void PrintVector() {

//validate Load Data Structure was successful.

if size of courses is greater than 0 {

//sort courses alphanumerically

for every course[i] in courses {

for every course[j] before course[i] {

if courses[j] is greater than courses[i] {

swap course indexes

}

}

}

//loop courses to print courseNumber and courseName

for every course[i] in courses {

output: courses[i].courseNumber “,”;

output: courses[i].courseName;

}

}

//if data structure hasn’t been loaded

else {

output: “Error: Load Data Structure first.”;

}

}

**//PRINT COURSE**

void PrintVectorNode() {

//validate Load Data Structure was successful.

if courses size is greater than 0 {

string courseInput;

int courseFound equals 0;

output: “What course do you want to know about? “;

input: courseInput;

//loop through courses to find courseInput match

for every course[i] in courses {

if courseNumber[i] equals courseInput {

output: courses[i].courseNumber “, ”;

output: courses[i].courseName;

output: “Prerequisites: “;

if courses[i].coursePrereqs(except last) has data {

for every course[j] in courses[i].coursePrereqs {

output: courses[i].coursePrereqs[j] “,”;

}

//eliminate comma from last entry

output: courses[i].coursePrereqs[size of coursePrereqs - 1];

//flag for course was found

courseFound equals 1;

}

//print none if no prereqs

else {

output: “None”;

}

}

}

//course input validation

if courseFound equals 0 {

output: “Error: Invalid Course”;

}

}

//if data structure hasn’t been loaded

else {

output: “Error: Load Data Structure first.”;

}

}

**HASHTABLE**

const int DEF\_SIZE equals 10;

class HashTable {

public:

//initialize class objects and variables

struct Node {

Course course;

int key;

Node\* next;

}

int tableSize equals DEF\_SIZE;

int hash(int key);

vector<Node> nodes;

//initialize constructor

Node() {

key equals UINT\_MAX;

next equals nullptr;

}

//initialize Node with a Course

Node(Course newCourse) : Node() {

course equals newCourse;

}

//initialize Node with a Course and a Key

Node(Course newCourse, int newKey) : Node(newCourse) {

key equals newKey;

}

};

**//LOAD HASH TABLE**

void LoadHashTable() {

//local variables

string fileName;

//get filename

output: “Please enter name of file: “;

input: fileName;

//open file

ifstream infile;

infile.open(fileName);

//insert data into hash table

if infile is open {

string line;

//calculate table size

while there is a line infile {

incriment tableSize by 1;

}

resize hashTable to updated table size

//read file line by line

while there is a line infile {

vector <string> tokens;

string token;

stringstream ss(line);

//tokenize every string at the comma

while reading the string, seperate at ‘,’ {

push each token to tokens

}

//ensure there are at least two parameters on each line

if size of tokens is less than 2 {

break;

}

//create a newCourse and add data from tokens

course equals Course.newCourse;

HashTable:key equals tokens[0];

newCourse.courseName equals tokens[1];

//if prereqs exist in tokens, add to course

if size of tokens is greater than 2 {

for every token[i] after the first 2 {

//ensure any prerequisite that is provided on a line exists

//as a course in the file

for every course[j] in courses {

if HashTable.key equals token[i] {

add token to newCourse.coursePrereqs

}

}

}

}

add newCourse to HashTable

}

}

//if the fileName does not exist

else {

output: “Error: Filename not found.” << endl;

}

infile.close();

};

**//PRINT Hash Table**

void PrintHashTable() {

//validate Load Data Structure was successful.

if size of HashTable is greater than 0 {

//sort nodes alphanumerically

for every key[i] in HashTable {

for every key[j] before course[i] {

if key[j] is greater than key[i] {

swap nodes

}

}

}

//loop hash table to print courseNumber and courseName

for node in HashTable {

output: node.key “,”;

output: node.value[0];

}

}

//if data structure hasn’t been loaded

else {

output: “Error: Load Hash Table first.”;

}

}

**//PRINT Hash Table Node**

void PrintHTNode() {

//validate Load Data Structure was successful.

if courses size is greater than 0 {

string courseInput;

int courseFound equals 0;

output: “What course do you want to know about? “;

output: courseInput;

output: endl;

//loop through hash table to find courseInput match

for every node[i] in HashTable {

if node[i]->key equals courseInput {

output: node[i]->key “, ”;

output: node[i]->courseName;

output: “Prerequisites: “;

if node[i]->coursePrereqs(except last) has data {

for every node[j] in HashTable[i]->coursePrereqs {

output:node[i]->coursePrereqs[j] “,”;

}

//eliminate comma from last entry

output: last element of node;

//flag for course was found

courseFound equals 1;

}

//print none if no prereqs

else {

output: “None”;

}

}

}

//course input validation

if courseFound equals 0 {

output: “Error: Invalid Course”;

}

}

//if hash table hasn’t been loaded

else {

output: “Error: Load Hash Table first.”;

}

}

**TREE**

//initialize objects and variables

struct TreeNode {

Course course;

TreeNode\* left;

TreeNode\* right;

int data;

TreeNode(int data) {

this->data equals data;

left equals right equals nullptr;

}

//create key for course

int key;

TreeNode\* node;

//if key entry found

if node has data and key does not equal UINT\_MAX {

return node->course

//if key entry is not found

if node has no data or key equals UINT\_MAX {

return course

}

//loop for matching node

while node has data {

if node->key does not equal UINT\_MAX and key equals 0 {

return node->course;

}

node equals node->next;

}

return course;

}:

//preorder traversal

void preorderTrav(struct TreeNode\* node) {

if node is empty {

return;

}

preorderTrav(node->left);

preorderTrav(nod->right

}

//postorder traversal

void postorderTrav(struct TreeNode\* node) {

if node is empty {

return;

}

postorderTrav(node->left);

postorderTrav(node\_>right);

}

//inorder traversal

void inorderTrav(struct TreeNode\* node) {

if node is empty {

return;

}

inorderTrav(node->right);

**//LOAD Tree**

void LoadTree() {

//local variables

string fileName;

//get filename

output: “Please enter name of file: “;

input: fileName;

//csv Parse file

csv::Parser infile equals csv::Parser(fileName);

//create tree and add to collection

try {

for every row infile {

Course course;

course.courseNumber equals infile[i][0];

course.courseName equals infile[i][1];

if size of infile is greater than 2 {

for every index[j] infile after 1 {

course.courePrereqs equals infile[i]index[j];

}

}

} catch(csv::Error& e) {

e.what();

}

}

}

**//PRINT Tree**

void PrintTree() {

struct TreeNode\* root equals new Node();

root->left equals new TreeNode(node);

root->right equals new TreeNode(node);

root->left->left equals new TreeNode(node);

root->left->right equals new TreeNode(node);

//validate Load Data Structure was successful.

if size of Tree is greater than 0 {

//sort nodes alphanumerically

for every key[i] in Tree {

for every key[j] before course[i] {

if key[j] is greater than key[i] {

swap nodes

}

}

}

//loop tree to print courseNumber and courseName

for node in Tree {

output: node.key “,”;

output: node.value[0];

}

}

//if data structure hasn’t been loaded

else {

output: “Error: Load Tree first.”;

}

}

**//PRINT Tree Node**

void PrintTreeNode() {

//validate Load Data Structure was successful.

if courses size is greater than 0 {

string courseInput;

int courseFound equals 0;

output: “What course do you want to know about? “;

output: courseInput;

output: endl;

//loop through tree to find courseInput match

for every node[i] in Tree {

if node[i]->key equals courseInput {

output: node[i]->key “, ”;

output: node[i]->courseName;

output: “Prerequisites: “;

if node[i]->coursePrereqs(except last) has data {

for every node[j] in HashTable[i]->coursePrereqs {

output:node[i]->coursePrereqs[j] “,”;

}

//eliminate comma from last entry

output: last element of node;

//flag for course was found

courseFound equals 1;

}

//print none if no prereqs

else {

output: “None”;

}

}

}

//course input validation

if courseFound equals 0 {

output: “Error: Invalid Course”;

}

}

//if tree hasn’t been loaded

else {

output: “Error: Load Tree first.”;

}

}

**//MAIN**

int main() {

bool quit equals false;

string ds;

output: “Welcome to the course planner.”;

//data structure selection

output: “1. Vector”;

output: “2. Hash Table”;

output: “3. Tree”;

output: “Which Data Structure would you like to use? “;

input: ds;

//input validation

while ds does not equal 1,2, or 3 {

output: “Error: Invalid Input”;

input: ds;

}

do {

string input;

//main menu

output: “1. Load Data Structure”;

output: “2. Print Course List”;

output: “3. Print Course” <<;

output: “9. Exit”

output: “What would you like to do? “;

cin >> input;

//input validation

while input does not equal any of the listed options {

output: “Error: “ input “ is not a valid option.”;

output: “What would you like to do? “;

input: input;

}

//navigation

if input equals 1 {

if ds equals 1 { LoadVector }

if ds equals 2 { LoadHashTable }

if ds equals 3 { LoadTree }

};

if input equals 2 {

if ds equals 1 { PrintVector }

if ds equals 2 { PrintHashTable }

if ds equals 3 { PrintTree }

};

if input equals 3 {

if ds equals 1 { PrintVectorNode }

if ds equals 2 { PrintHTNode }

if ds equals 3 { PrintTreeNode }

};

if input equals 9 {

output: “Thank you for using the course planner!”;

exit(quit);

}

} while (quit is false);

return 0;

}

**Evalulation**

**Vector**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Load Vector** |  |  |  |
| **Initialize string fileName** | 1 | n | n |
| **Get fileName from input** | 1 | n | n |
| **Validate fileName** | 1 | 1 | 1 |
| **Open inFile with fileStream** | 1 | 1 | 1 |
| **Initialize vector tokens** | 1 | n | n |
| **Get inFile line** | 1 | n | n |
| **Tokenize line at comma** | 1 | 1 | 1 |
| **Add token to tokens vector** | 1 | 1 | 1 |
| **Initialize a new course** | 1 | n | n |
| **Set token[0] to coureNumber** | 1 | 1 | 1 |
| **Set token[1] to courseName** | 1 | 1 | 1 |
| **If prereqs exist add to prereqs vector** | 1 | n | n |
| **Add new course to courses** | 1 | n | n |
| **Close inFile** | 1 | 1 | 1 |
| **Print Course** |  |  |  |
| **Initialize string courseInput** | 1 | n | n |
| **Initialize int courseFound** | 1 | n | n |
| **Get courseInput from input** | 1 | 1 | 1 |
| **Loop through every course in courses** | 1 | 1 | 1 |
| **If courseInput matches a couresNumber** | 1 | n | n |
| **Print courseNumber, couresName, and prereqs** | 1 | n | n |
| **Else print course not found** | 1 | n | n |
| **Total Cost** | | | 9 + 11n |
| **Runtime** | | | 20(n) |

**Hash Table**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Load Hash Table** |  |  |  |
| **Initialize string fileName** | 1 | n | n |
| **Get fileName from input** | 1 | n | n |
| **Validate fileName** | 1 | 1 | 1 |
| **Open inFile with fileStream** | 1 | 1 | 1 |
| **Get inFile line** | 1 | n | n |
| **Tokenize line at comma** | 1 | 1 | 1 |
| **Add token to Node struct** | 1 | 1 | 1 |
| **Set token[0] to key** | 1 | 1 | 1 |
| **Set token[1] to courseName** | 1 | 1 | 1 |
| **If prereqs exist add to prereqs vector** | 1 | n | n |
| **Add node to hash table** | 1 | n | n |
| **Close inFile** | 1 | 1 | 1 |
| **Print HTNode** |  |  |  |
| **Initialize string courseInput** | 1 | n | n |
| **Initialize int courseFound** | 1 | n | n |
| **Get courseInput from input** | 1 | 1 | 1 |
| **Loop through every key in the hash table** | 1 | 1 | 1 |
| **If courseInput matches** | 1 | n | n |
| **Print key, courseName, and prereqs** | 1 | n | n |
| **Else print course not found** | 1 | n | n |
| **Total Cost** | | | 9 + 9n |
| **Runtime** | | | 18(n) |

**Tree**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Load Tree** |  |  |  |
| **Initialize tree structure** | 1 | 1 | 1 |
| **Get fileName from input** | 1 | n | n |
| **Validate fileName** | 1 | 1 | 1 |
| **Open inFile with csv::Parser** | 1 | 1 | 1 |
| **Get inFile line** | 1 | n | n |
| **Tokenize line at comma** | 1 | 1 | 1 |
| **Add token to Node struct** | 1 | 1 | 1 |
| **Set token[0] to key** | 1 | 1 | 1 |
| **Set token[1] to courseName** | 1 | 1 | 1 |
| **If prereqs exist add to prereqs vector** | 1 | n | n |
| **Add node to tree** | 1 | n | n |
| **Close inFile** | 1 | 1 | 1 |
| **Print Tree** |  |  |  |
| **Initialize string courseInput** | 1 | n | n |
| **Initialize int courseFound** | 1 | n | n |
| **Get courseInput from input** | 1 | 1 | 1 |
| **Search for courseInput in tree** | 1 | n | n |
| **If courseInput exists** | 1 | n | n |
| **Print key, courseName, and prereqs** | 1 | n | n |
| **Else print course not found** | 1 | n | n |
| **Total Cost** | | | 7 + 11n |
| **Runtime** | | | 18(n) |

**Advantages/Disadvantages**

Data structures help the storage of and provide convenience while retrieving the data from storage device. Each data structure used in this project has its unique advantages and disadvantages. The first of the three data structures that will be evaluated are vectors. Vectors are the simplest of the data structures in that they are basically dynamic arrays that can store various objects and data types. They are easy to implement, and they change size as elements are being added. One of the major disadvantages to vectors is that sorting and seaching requires looping through the entire vector. This can be exceedingly time consuming for larger scale vectors. The second data structure that will be evaluated are hash tables. Hash Tables are an ordered pair data structure with a key and values. This is the same structure as a JavaScript object. The keys help in identifying and element in the hash table and doesnt require looping through the entire data structure. The disadvantage for hash tables are that they are slow and are static. This means that if elements are added or removed, the hash table has to be constantly resized. Because they work off of keys, the indexes can change to different elements causeing chaining issues which have to be addressed in some manner. The final data structure to be evaluated is the tree. The nice thing about tree data structures is that they are fast. They also have internal sorting and managing tools that allow easy manipulation of the elements. The tree data structure is the most sensitive of the structures. This make erroneous data difficult to locate and remove. The data structure can cause stack overflow when using recursion and the shape of the tree is dependent on the first item that is inserted.

**// Recommendation**

After studying the different types of data structures and evaluating the project’s functional requirements, I think that a vector or hash table would work given the system’s current functionality. I would lean hash table to allow for the key to be the course number and the search and navigation bonuses it offers. If we wanted to expand on this system in the future, a binary tree would be the most complete solution for storing course objects. This is primarily due to the courses being sorted in alphabetical order the binary tree having the tree sort algorithm built into the data structure. While the other data structure has the ability to sort through data in alphabetic order, it is more cumbersome and can cause slower results. While the binary search tree may not have all of the features the other data structures have, it should be the most efficient tool for this purpose.

**Resources**

GeeksforGeeks. (2022, January). *Data Structures*. Retrieved February 15, 2022, from https://www.geeksforgeeks.org/data-structures/

Zybooks. (n.d.). *zyBooks*. Zybooks.Com. https://learn.zybooks.com/zybook/SNHUCS300v1/chapter/6/section/1