**Project 1:**

**ABC University Pseudocode and Data Structure Evaluation**

Michael Duteau

Southern New Hampshire University

CS-300: DSA: Analysis and Design

Dr. Michael Susalla

August 6, 2023

**Pseudocode for opening and reading a file (all data structures)**

Using #include <fstream> allows for file reading and writing

ReadFile {

call to open file;

if file not found {

print “File not found.” and break method;

}

else {

while not EOF {

for each line of the file until EOF {

if currentLine parameters < 2 {

return error “Incorrect file formatting”;

}

else {

read parameters;

}

if currentLine parameters > 2 {

if any parameters succeeding the first 2 are not present as the first parameter elsewhere in the file {

return error “Nonexistent prerequisite.”;

}

}

}

}

}

call to close file;

}

**Vector Pseudocodes**

Creating *Course* objects:

class Course {

string *courseNumber*;

string *courseName*;

vector<string> **PreReqs**;

}

vector<Course> LoadCourses(*file path*) {

vector<Course> **courses**;

call to open *file path* file;

if *file path* null {

return error “File not found.”;

}

for every row in file from *file path* {

if current line parameters < 2 {

return error “Incorrect file formatting.”;

}

*courseNumber* = parameter 1;

*courseName* = parameter 2;

if current line parameters > 2 {

for each extra parameter {

push\_back parameter to **PreReqs**;

}

}

push\_back current Course to **courses**;

}

call to close *file path* file;

}

*Course* searching and printing:

void PrintCourseInformation(vector<Course> **courses**, *courseNumber*) {

for each course in **courses** {

if **courses**.courseNumber == *courseNumber* {

print *courseNumber* and *courseName*;

for each *preReq* in **PreReqs** {

print *preReq*;

}

return;

}

}

// Case where *courseNumber* is not found in **courses**

print “Course not found.”

return;

}

**Hash Table Pseudocodes**

Creating *Course* objects:

struct Course {

string *courseNumber*;

string *courseName*;

vector<string> **PreReqs**;

}

class HashTable {

struct Node {

Course *course*;

unsigned int *key*;

Node \*next;

}

vector<Node> *nodes*;

}

void LoadCourses(string *file path*, HashTable) {

vector<Node> **nodes**;

call to open *file path* file;

if *file path* null {

return error “File not found.”;

}

for every row in file from *file path* {

if current line parameters < 2 {

return error “Incorrect file formatting.”;

}

*courseNumber* = parameter 1;

*courseName* = parameter 2;

if current line parameters > 2 {

for each extra parameter {

push\_back parameter to **PreReqs**;

}

}

append current Course to **nodes**;

move to next node;

}

call to close *file path* file;

}

*Course* searching and printing:

void PrintCourseInformation(vector<Node> **nodes**, *courseNumber*) {

unsigned int *key* = hash(*courseNumber*);

for each node in **nodes** {

if **nodes**.at(*key*)->courseNumber == *courseNumber* {

print **nodes**.at(*key*)->*courseNumber* and ->*courseName*;

for each *preReq* in **nodes**.at(*key*)->**PreReqs** {

print *preReq*;

}

return;

}

}

// Case where *courseNumber* is not found in **courses**

print “Course not found.”

return;

}

**Binary Search Tree Pseudocodes**

Creating *Course* objects:

struct Course {

string *courseNumber*;

string *courseName*;

vector<string> **PreReqs**;

}

struct Node {

Course *course*;

Node\* left;

Node\* right;

}

class BinarySearchTree {

Node\* root;

BinarySearchTree;

addNode(Node\* *node*, Course *course*);

inOrder(Node\* *node*);

}

void LoadCourses(string *file path*, BinarySearchTree) {

call to open *file path* file;

if *file path* null {

return error “File not found.”;

}

for every row in file from *file path* {

if current line parameters < 2 {

return error “Incorrect file formatting.”;

}

*courseNumber* = parameter 1;

*courseName* = parameter 2;

if current line parameters > 2 {

for each extra parameter {

push\_back parameter to **PreReqs**;

}

}

addNode current Course to BinarySearchTree;

move to next node;

}

call to close *file path* file;

}

addNode(Node\* *node*, Course *course*) { *node* is the root in this case

*currentCourse* = *course*’s courseNumber;

if (*node* is null) {

*currentCourse* becomes the root;

}

else if (*course* courseNumber < *node* courseNumber) {

if (*node*->left is null) { node left of *node* is empty and becomes *course*

*node*->left = new Node(*course*);

}

else { node left of *node* is not empty, recurse left

this->addNode(*node*->left, *course*);

}

}

else { instance where *course* courseNumber is greater than *node* courseNumber

if (*node*->right is null) { node right of *node* is empty and becomes *course*

*node*->right = new Node(*course*);

}

else { node right of *node* is not empty, recurse right

this->addNode(*node*->right, *course*);

}

}

}

*Course* printing:

void PrintCoursesInOrder (Node\* *node*) {

PrintCoursesInOrder(*node*->left);

output *node*’s *courseNumber*, *courseName*, and **PreReqs**;

PrintCoursesInOrder(*node*->right);

}

void PrintCoursesPreOrder (Node\* *node*) {

output *node*’s *courseNumber*, *courseName*, and **PreReqs**;

PrintCoursesPreOrder(*node*->left);

PrintCoursesPreOrder(*node*->right);

}

void PrintCoursesPostOrder (Node\* *node*) {

PrintCoursesPostOrder(*node*->left);

PrintCoursesPostOrder(*node*->right);

output *node*’s *courseNumber*, *courseName*, and **PreReqs**;

}

**Pseudocode for Menu:**

int main() {

int *choice* = 0;

While *choice* is not 9 {

cout << "Menu:" << endl;

cout << " 1. Load Courses" << endl;

cout << " 2. Display All Courses" << endl;

cout << " 3. Print Course Information" << endl;

cout << " 9. Exit" << endl;

cout << "Enter choice: ";

cin >> choice;

if (*choice* is 1) {

load course data;

}

if (*choice* is 2) {

print course list;

}

if (*choice* is 3) {

take *input* for *courseNumber* to be searched;

if (*input* is found in the list) {

print course *courseNumber*, course *courseName*, and course **preReqs**;

}

else {

output “Course not found.”;

}

}

else {

output “Invalid input.”

}

}

output “Good bye.”;

return 0;

}

**Pseudocode for printing in alphanumeric order (Vector):**

The quickSort() method will sort the vector, which can then be parsed to the print method.

partition(<vectorName>, begin, end) {

*low* = begin;

*high* = end;

*mid* = *low* + (*high* – *low*) / 2; this finds the middle index

*pivot* = vectorName[mid]; holds the middle element of the vector as the pivot variable

*done* = false;

while (not *done*) {

while (vectorName[*low*] is less than *pivot*) {

*low* += 1; *low* increments until vectorName[*low*] is greater or equal to *pivot*

}

While (*pivot* is less than vectorName[*high*]) {

*High* -= 1; *high* decrements until vectorName[*high*] is less or equal to *pivot*

}

if (*low* greater or equal to *high*) {

*done* = true; when low is great or equal to high, all elements are partitioned and method completes

}

else {

swap vectorName[*low*] and vectorName[*high*];

*low* +=1;

*high* -= 1;

low and high *courseNumber*s are swapped and values are incremented/decremented to bring the values closer together

}

}

return *high*;

}

quickSort(<vectorName>, begin, end) {

*mid* = 0;

if (*begin* is greater or equal to *end*) {

return; base case where 1 or 0 elements to sort, meaning partition is already sorted

}

*mid* = partition(<vectorName>, begin, end); partitions into *low* and *high* such that *mid* becomes the last element in *low*

quickSort(<vectorName>, begin, *mid*); recursively sorts lower partition

quickSort(<vectorName>, *mid*+1, end); recursively sorts higher partition

}

**Data Structure Evaluations and Recommendation**

Due to the nature of Binary Search Trees (BSTs), alphanumeric order of printing can be achieved using the InOrder() printing method, as the tree’s construction can automatically sort the data alphanumerically based on the *courseNumber* variable. On the contrary, hash tables are not typically ordered or able to be sorted by typical means, especially when considering element placements revolve around the element’s *key* rather than the *courseNumber* variable we wish to sort by. Because of this, printing a hash table’s contents in alphanumeric order may require either transposing the hash table’s elements into a sorted array or vector, or using a comparison algorithm in the printing method to compare every element in the table to determine the order in which they are printed.

Unlike BSTs and hash tables, vectors are not always able to be automatically sorted upon construction but *can* be sorted using algorithms; in this case, alphanumeric order can be achieved by implementing string comparison using the *less than* and/or *greater than* operands on the *courseNumber* variables within a sorting algorithm. The *.compare()* method can also achieve this kind of comparison. When considering the speed of operation, it’s worth noting that a quicksort would be much faster than using a selection sort; so, using a quicksort would be the wiser option in printing the course list in alphanumeric order where the operation speed is a key factor. After performing the sort method as outlined on the previous page, the sorted vector can then be printed in alphanumeric order by iterating through the list from the lowest index to the highest index.

When examining the runtime complexity of the data structures, different operations will yield different complexities based on which structure is being used. Vectors in general have a run time of O(n) for n number of elements in the vector, but accessing an element by index or using the push\_back() method both yield a complexity of O(1) provided that the vector does not need resizing when implementing the push\_back() method. Hash tables tend to have an average complexity of O(1) for the majority of operations; however, this can degrade to O(n) depending on how well collisions are handled. Binary search trees average O(log(n)) for the majority of operations while also having the possibility of degrading to O(n) based on the number of elements within the tree.

Due to the need for printing in alphanumeric order, I would say that the hash table structure is not a good fit; while creating code to compare elements after accessing them by the *key* is a possibility, this could end up being a needlessly complex endeavor compared to how the other data structures can simplify this need. While a vector would be a good fit for this scenario, I believe a binary search tree would provide a faster running program based on its average runtime complexities and the fact that the data set will be automatically sorted by the BST’s initial construction. Based on this information, I would recommend the binary search tree being the data structure used for the needs of ABC University.