Project One Pseudocode and Runtime Analysis

Branden Langhals

CS-300 DSA: Analysis and Design

11/28/2022

**Vector Pseudocode and Runtime Analysis**

**//Read File(s) and load data into structure**

fstream to open

File.open(“doc.csv”)

IF file fails to open

return error

courseName= second string of each line

While file open = true

String line;

getLine(file,line)

Course(line)

**// Storing Course Object into Vector**

Vector<course>courses

Course(line)

String name

Course \*courseName = new Course

FOR each string

courseName-> number = first parsed string

courseName-> name = second parsed string

IF no number or name

Return Error

FOR each remaining string

IF prerequisites name found in courseNames

prerequisites = remaining parsed string

courseName -> prerequisites = prerequisites

prerequisites ++

Courses.push\_back(\*courseName)

**// Vector Pseudocode**

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) {

For all courses

print the course name

if the course has prerequisites

print the prerequisites

}

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

}

**//Print Course List Alphanumerically**

SelectionSort(courses

unsigned int min

For j=i+1<course.size() ++j

IF courses.at j less than courses.at I

min equals j

IF min not equal I

swap courses at i, and min

FOR I<courses.size()

print all courses(i)

WHILE i<courses.size()

print sort courses(i)

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| fstream to open  File.open(“doc.csv”) | 1 | 1 | 1 |
| IF file fails to open | 1 | n | n |
| return error | 1 | 1 | 1 |
| courseName= second string of each line | 1 | 1 | 1 |
| While file open = true | 1 | n | n |
| String line;  getLine(file,line) | 2 | n | n |
| Course(line) | 1 | n | n |
| **Total Cost** | | | 4n + 3 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| for each string | 1 | n | n |
| CourseName->num = first parsed string | 1 | 1 | 1 |
| CourseName->num = second parsed string | 1 | 1 | 1 |
| IF no number or name  Return Error | 1 | n | n |
| For each string remaining | 1 | n | n |
| If prerequisite name found in courseNames | 1 | n | n |
| Prerequisites = remaining string parsed | 1 | n | n |
| courseName-> prerequisite = prerequisite | 1 | n | n |
| Prerequisites ++ Courses.push\_back(\*courseName ) | 1 | n | n |
| **Total Cost** | | | 7n + 2 |
| **Runtime** | | | O(n) |

**// Hashtable**

Struct Course {

String courseName;

String courseNumber;

Vector<String> Prerequisite;

}

Class Hashtable {

Struct Node {

Course course;

Int key;

Node \* next;

Node()

Node(Course course, Int key)

}

Vector<Node> nodes;

Unsigned int hash(int key);

}

Hashtable();

Insert(Course course);

}

int numPrerequisiteCourses(Hashtable<Course> courses) {

totalPrequisite = Hashtable[]

For each prerequisite in totalPrerequisite

add prerequisite in Hashtable to the totalPrerequisite variable

Print totalPrerequisite

}

void printSampleSchedule(Hashtable<Course> courses) {

For all (key) value match in courses

print course name

if course has prerequisite

for each prerequisite

Print prerequisite

}

void printCourseInformation(Hashtable<Course> courses, String courseNumber) {

For all courses

If Course matches courseNumber

print course information

For each prerequisite of Hashtable{Course]

print prerequisite course information

}

Int main()

HashTable\*table = new HashTable()

Vector<string>temp

String line

Fstream infile(“file”)

WHILE(getLine(infile,line))

Stringstream ss(lin)

WHILE(ss.good())

String substr

getline(ss,substr)

temp.push\_back(substr)

Table.insert(parseLine(temp))

Temp clear()

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Create key by hashing courseNumber | 1 | n | n |
| while node != nullptr | 1 | n | n |
| CourseName->num = second parsed string | 1 | 1 | n |
| If node→course courseNumber != courseNumber | 1 | n | n |
| print course information | 1 | 1 | 1 |
| for each prerequisite of the course | 1 | n | n |
| print prerequisite information | 1 | n | n |
| **Total Cost** | | | 5n + 1 |
| **Runtime** | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| HashTable\* table = new HashTable() Vector temp | 2 | 1 | 1 |
| Ifstream infile(“filestream”) | 1 | 1 | 1 |
| while(getline(infile, line)) | 1 | n | n |
| Stringstream ss(line) | 1 | n | n |
| while(ss.good()) | 1 | n | n |
| String substr  getline(ss,substr)  temp.push\_back(substr) | 3 | n | n |
| table.insert(parseLine(temp)); | 1 | n | n |
| **Total Cost** | | | 5n + 2 |
| **Runtime** | | | O(n) |

**// Tree pseudocode**

Struct Course {

String courseName;

String courseNumber;

Vector<String> Prerequisite;

}

int numPrerequisiteCourses(Tree<Course> courses) {

totalPrerequisites = left and right of the child node

for each prerequisite in totalPrerequisites

add left and right nodes to total Prerequisites

Print totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

for all Nodes that are courses

print courseName

if course contains left node

print left node prerequisite

if course contains right node

print right node prerequisite

}

void printCourseInformation(Tree<Course> courses, String courseNumber) {

all Nodes

if course == courseNumber

print all Node data

if course has left node

print left node as prerequisite course data

if course has right node

print right node as prerequisite course data

else

if course has left node

print left node

If course has right node

print right node

}

**//Print Course List Alphanumerically**

InOrder(Node\* node)

IF node not equal nullptr

inOrder→node.left

print course information

InOrder→node.right

CourseInformation→inOrder()

print inOrder

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Create tree | 1 | 1 | 1 |
| 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 right | 1 | n | n |
| If no right node | 1 | n | n |
| This node becomes the right node | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| While prerequisite exists | 1 | n | n |
| Append prerequisite | 1 | n | n |
| Insert course item | 1 | n | n |
| **Total Cost** | | | 11n + 2 |
| **Runtime** | | | O(n) |

**//Menu Pseudocode**

Set choice to 0;

Create while loop for menu.

While choice is not equal to 4

Output menu choices:

1. Load Course File,

2. Print Course List

3. Print Chosen Course

4.Exit

Create switch(choice)

Case 1: loadCourses(courseFile, dataStructure)

Case 2: printSort(courses) call function to print the sorted list

Case 3: printCourseInformation(courseId)

Case 4: Terminate Program

**Data Structure Evaluation and Recommendation**

All the observed data structures have advantages and disadvantages when parsing and loading data for the program. Utilizing the Vector method was the fastest for reading the file and adding course data as objects. Vectors operate by adding each parsed item to the end of the vector and as noted in the runtime analysis it was the shortest. The main drawback of using the Vector method is that to search for a specific item, the vector must pass through each line of data in order until the item is located. Vectors are also not the best for sorting in alphanumeric order. For this reason, Vectors are not the best option for this project.

Hash tables are excellent for searching the list quickly. As keys are created the location of each course can be accessed promptly when searched for individually. The disadvantage to this method is that the initial list creation is time-consuming and for larger groups of data, a location for each key must also be created. When a user wants to sort the list alphanumerically the Hash Table will have to extract all keys before it can sort and print them again. Hash Tables can eat up a lot of space and are overall not best for large data sets.

Binary search trees are faster and can store custom classes using multiple fields. Binary trees traverse data in order, preorder, and postorder while using inserts and deletes as well. This data type is particularly useful in routing systems and is more scalable than the other choices. Searching for courses that are stored is faster and more efficient than the other options. The main disadvantage to using this method is that data must be sorted and stored in a way that adds to overall runtime complexity. Deleting nodes is not a short process, and ass the tree gets larger, they also get harder to search.

Based solely on the requirement of being searchable alphanumerically, Hash Tables and Binary Search Trees are the best options. However, I would recommend that this project uses Binary Search Trees because they require less sorting, and they work well with the alphanumeric ordering that is required. While the runtime analysis is not optimal, this method will work well with the user menu and should satisfy all program requirements.