Southern New Hampshire University

CS 300

Ashli Campbell

Project One

1. **Pseudocode:**

**Vector Pseudocode:**

* Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors.

Parser::Parser

Open file

While not end of file

if file is open

getLine

if line != ""

push back

if file is closed

Generate error output “Failed to open”

getLine

Check to see if at least 1 comma is present

if no comma is present

Generate error message “Not enough information”

if more than 1 comma is present, then there are prereqs

Check to make sure the prereqs exist as a course in the file

Split string with delimiter “,”

Save each piece as the courseNumber, courseName, and following prereqs

Close file

* Design pseudocode to show how to create course objects and store them in the appropriate data structure.

vector<Course> loadCourses (string filePath)

vector<Course> courses;

For int i, i < file.rowCount, increment i

Course course;

course.courseNumber

course.courseName

course.prereq

courses.push\_back(course)

Return courses

* Design pseudocode that will search the data structure for a specific course and print out course information and prerequisites.

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) {

Print course name, course number

}

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

}

**Hash Table pseudocode:**

* Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors.

Parser::Parser

Open file

While not end of file

if file is open

getLine

if line != ""

push back

if file is closed

Generate error output “Failed to open”

getLine

Check to see if at least 1 comma is present

if no comma is present

Generate error message “Not enough information”

if more than 1 comma is present, then there are prereqs

Check to make sure the prereqs exist as a course in the file

Split string with delimiter “,”

Save each piece as the courseNumber, courseName, and following prereqs

Close file

* Design pseudocode to show how to create course objects and store them in the appropriate data structure.

void loadCourses (string filePath, HashTable\* hashTable)

Initialize the parser using the given path

For int i, i < file.rowCount, increment i

Course course;

course.courseNumber

course.courseName

course.prereq

hashTable->Insert(course)

* Design pseudocode that will print out course information and prerequisites.

int numPrerequisiteCourses(Hashtable<Course> courses) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {

Print course name, course number

}

void printCourseInformation(Hashtable<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

}

**Binary Tree pseudocode:**

* Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors.

Parser::Parser

Open file

While not end of file

if file is open

getLine

if line != ""

push back

if file is closed

Generate error output “Failed to open”

getLine

Check to see if at least 1 comma is present

if no comma is present

Generate error message “Not enough information”

if more than 1 comma is present, then there are prereqs

Check to make sure the prereqs exist as a course in the file

Split string with delimiter “,”

Save each piece as the courseNumber, courseName, and following prereqs

Close file

* Design pseudocode to show how to create course objects and store them in the appropriate data structure.

void loadCourses (string filePath, BinarySearchTree\* bst)

Print “Loading CSV file”

Initialize the parser using the given path

For int i, i < file.rowCount, increment i

Course course;

course.courseNumber

course.courseName

course.prereq

bst->Insert(course)

* Design pseudocode that will print out course information and prerequisites.

int numPrerequisiteCourses(Tree<Course> courses) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printSampleSchedule(Tree<Course> courses) {

Print course name, course number

}

void printCourseInformation(Tree<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

}

1. **Create pseudocode for a menu:**

Method main

Switch (process command line arguments)

While choice does not equal 9

Print “Menu:”

Print “1. Load Data Structure”

Print “2. Print Course List”

Print “3. Print Course”

Print “4. Exit”

User input menu choice

Case 1

Call loadCourses function

Case 2

For loop

Call printCourseInformation function

Case 3

For loop

Call printSampleSchedule function

Case 4

Exit

Print “Good bye.”

1. **Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.**

Function sortCourses (string a, string b)

Return a>b

//returns 1 if string a is alphabetically less than string b, else returns 0

End function

sort(courses.begin, courses.end, sortCourses)

//sorts the courses in alphanumeric order

Print courses

1. **Evaluate the run-time and memory of data structures that could be used to address the requirements**.

| **Vector Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |

The average time complexity for vectors is 0(n), and the worst case is also 0(n).

| **Binary Tree Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(log(n)) |

The average time complexity for binary search trees is 0(log(n)). The worst-case runtime is 0(n).

| **Hash Table Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **print out the course information** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(1) |

The average time complexity for hash tables is 0(1). The worst-case runtime is 0(n).

1. Based on the advisor’s requirements, analyze each data structure (vector, hash table, and tree). **Explain the advantages and disadvantages of each structure in your evaluation.**

The data structure chosen for an application has a big impact on the design and maintenance of the program. An advantage of a binary search tree is that you can have all of your keys sorted in order as a natural operation, whereas in other structures like a hash table it would require extra time and effort. The binary tree is simplistic and easy to understand, but there are some limitations due to certain functions being dependent on the height of the tree. The insertion, deletion and search are conducted in 0(log(n)) time.

Hash tables, on the other hand, are very efficient and can hold a large amount of data. One disadvantage is that hash functions can sometimes create duplicate keys, which causes “collisions” and is often difficult to avoid. The insertion, deletion and search are conducted in 0(1) time.

For vectors, the insertion, deletion and search are conducted in 0(n) time. Insertion at the tail of the vector is the fastest – meaning that if you need to insert elements anywhere but the end, then a different data structure should be used due to the vector’s inefficiency. This could cause the memory usage to spike to an amount that is higher than the memory required by the vector. However, vectors have low memory usage and allows random access, unlike binary search trees.

1. Now that you have analyzed all three data structures, **make a recommendation for which data structure you will plan to use in your code**. Provide justification for your recommendation, based on the Big O analysis results and your analysis of the three data structures.

The data structure that I would recommend is the hash table, because it is the most efficient tool to sort and search for courses within the data structure since the key is always known. The hash table pairs a key with a value, and objects can be accessed by the key. In this case, the “key” is the course code. This makes accessing random elements quick and easy, which is the priority for this data. No order is maintained in a hash table (like it would be in a binary tree), which is perfectly fine. This is great for any size database, so it works for the course list.