CS-300 DSA

Laura McAroy

Module 3 Milestone

February 13, 2022

**Vector Pseudocode**

// Opening and reading file

Open file

If file is open

While not end of file

Read line

If line does not contain course and course number

Print Error

Else

If line contains prerequisite

Find course number for course in file

If course number not found

Print Error

Read next line

//Store objects in vector

For each line in file

If course object not already in Vector

Add course to vector

// Print course Info

For all courses

If course matches courseNumber

Print out course info

For each prerequisite

Print the prerequisite course information

**Hashtable Pseudo:**

// Opening and reading file

Open file

If file is open

While not end of file

Read line

If line does not contain course and course number

Print Error

Else

If line contains prerequisite

Find course number for course in file

If course number not found

Print Error

Read next line

//Storing Items in Hashtable

int numPrerequisiteCourses(Hashtable<Course> courses) {

if course not found in Hashtable<Course>

insert course in next empty index

}

//Print functions

void printSampleSchedule(Hashtable<Course> courses) {

for all key, value pair in courses

print key course name

if course has prerequisites

for each prerequisite

print prerequisite

}

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

for all courses

if course is equal to courseNumber

print course information

for each prerequisite of the course

print prerequisite course information

**Binary Tree Pseudo:**

// Opening and reading file

Open file

If file is open

While not end of file

Read line

If line does not contain course and course number

Print Error

Else

If line contains prerequisite

Find course number for course in file

If course number not found

Print Error

Read next line

//Storing items in Tree

int numPrerequisiteCourses(Tree<Course> courses) {

for each course

if course has a left or right node

add nodes to totalPrerequisites

print totalPrerequisites

}

//Print functions

void printSampleSchedule(Tree<Course> courses) {

for all courses

print course

if course has a left node

print left node as prerequisite

if course has a right node

print right node as prerequisite

}

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

while node is not empty

if course is equal to courseNumber

print course

if course has a left node

print node as prerequisite

if course has a right node

print node as prerequisite

else if course is less than courseNumber

traverse left

else

traverse right

}

**Menu Pseudocode**

Display Menu Options

1. Load Data Structure
2. Print Course List
3. Print Course
4. Exit

Obtain user input

If user input is invalid choice

Display error message

Obtain new input

While user input is valid

If user input = 1

Open/ load file containing data structure

If file is not found

Print error message

Else

Print “Courses loaded”

If user input = 2

If file containing data is not open

Print error message

Else

printSampleSchedule()

If user input = 3

If file containing data is not open

Print error message

Else

Prompt user for desired course

Obtain user input

If user input not found

Print error message

Obtain new input

Else

printCourseInformation(user input)

If user input = 4

Exit program

**Sorting Courses in Alphabetical Order**

//Print Course Information

For each course

For each course name/number

If course number > next course number

Swap courses

For each course

Print course information

**Runtime Analysis**

**Vector**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times executed** | **Total Cost** |
| For each line in file | 1 | N | N |
| If item is not found in vector | 1 | N | N |
| Add item to vector | 1 | N | N |
|  |  |  |  |
|  |  | **Total Cost** | **3n** |
|  |  | **Runtime** | **O(n)** |

**HashTable**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| For each line in file | 1 | N | N |
| If course not found in Hashtable | 1 | N | N |
| Insert course in next empty index | 1 | N | N |
|  |  |  |  |
|  |  | **Total Cost** | **3n** |
|  |  | **Runtime** | **O(n)** |

**Binary Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| For each course | 1 | N | N |
| If course has a left or right node | 1 | N | N |
| Add nodes to totalprerequisites | 1 | N | N |
| Print totalprerequisiites | 1 | 1 | 1 |
|  |  |  |  |
|  |  | **Total Cost** | **3n+1** |
|  |  | **Runtime** | **O(n)** |

While each of these charts exhibit a worst case runtime scenario of O(n), the average runtime of a Binary Search Tree is O(logN) and a hashtable has an average runtime of O(1), making them both more efficient than a vector, whose average runtime is O(n).

Binary search trees are preferred to other data structures because items are stored in a way that all of the nodes to the left of an element are less than the element’s value and all of the elements to the right have a greater value, making it easy to locate specific items in the tree and contributing to its quick runtime. Hashtables store items in key: value pairs, which uses a larger amount of memory than a BST, but allows for a constant runtime of O(1) in most cases. A downside to using Hashtables however, is the probability of collisions when attempting to insert a value in a key that is already occupied. Although there are methods to avoid collisions, collision handling measures often results in increased runtime. Vectors tend to occupy more memory space, since they are able to increase size to accommodate added elements, and their linear runtime is slower than both BST and Hashtables, as each element in the list needs to be adjusted based on where the insertion and deletion index is. However, Vectors are much easier to initiate and utilize than the other two data structures.

Based on these findings, I believe it would be in the company’s best interest to use a binary search tree as the data structure to hold the information for the computer science courses moving forward. Although it has a slightly slower runtime than the hash table, binary search trees use less memory, and the data is sorted in a more ordered fashion than the hash table. Binary trees also do not require implementation of hash functions to determine keys or prevent collisions. Vectors, although the easiest of the three to implement, also require a lot of memory and have the slowest runtime, making them the least efficient.