James Jones

David Ostrowski

CS-300

12/3/22

Project 1

Milestone One Pseudocode

**//Original Pseudocode for each data type**

//Pseudocode for reading/parsing file (loadFile function)

Using fstream to open file

Call file to open

If (file returns -1)

Print file not found.

Else

While( not at end of file)

Read line

If(tokens less than 2)

Print “error”

Else

Read line

Close file

//Vector PseudoCode

//Creating course objects

Create course vector courseInfo()

For each line

courseId equals value before ,

courseName equals value before , or endline

For (each line after row 2)

if (at end of line)

next line

else (not at end of line)

this prereq equals value before , or endline

//verify prereq courses

For each course prereq in courseInfo vector

For each course courseId

If course prereq equals courseId(i)

Return

Else

Print invalid course prereq

//Search Vector for course

For(index 0; while index is less than vector size; add 1 after each iteration)

If(courseId equals key)

Print vector courseId, courseName, prereq’s at index

Else return -1

//HashTable PseudoCode

Create Course Vector <node> nodes

Create HashTable

Insert method to insert into Hashtable

While(not end of file)

For(index = 0; index < nodes; ++index)

Create temp item for first 2 values

If(3rd value does not equal null)

Add new value (prerequisite course)

Call insert() for each value

Search/print from HashTable

Get user input

Set user input equal to key

If (key equals value)

Print course object

For(index = 0; index < size; ++i)

Print prerequisites for course object

//Binary Search Tree

Create Binary tree class

Create root pointing to null

Create Insert method to insert into tree

If root equals null

Root equals current course

Else if course number less than root

If left equals null

Add course number

Else

If course number less than leaf

Add left

Else

Add right

Else course number greater than root

If right equals null

Add course number

Else

If course number less than leaf

Add left

Else

Add right

Print course information

Get user input

Set user input equal to course number

Search tree

If root equals null

Print “No current classes”

Else if course number less than root

If left equals null

Print this course number

Else

If course number less than leaf

Print this course number

Else

Print this course number

Else course number greater than root

If right equals null

Print this course number

Else

If course number less than leaf

Print this course number

Else

Print this course number

**//New Pseudocode for each data type**

//Print Ordered list

//vector

Create quicksort function to sort courses into alphanumerical order

Set low to starting index, and high to ending index

quicksort(vector[], low, high)

If(low less than high)

Call partitionI(vector[], low, high)

Call quicksort(vector[], low, partition – 1) // sets before partition

Call quicksort(vector[], partition + 1, high) //sets after partition

//partition for use in quick sort algortithm

Create partition function arguments (vector[], low, high)

Set pivot equal to high

Set I equal to low – 1

For(j = low; j <= high – 1; ++j)

If(vector[j] < pivot)

I++

Swap vector[i] and vector[j]

Swap vector[i+1] and vector[high]

Return(I +1)

//print

Create printList function

For(i=0; I = vector.size; ++i)

Print vector[i]’s course Id

If(vector[i] has prerequisites)

Print vector[i] prerequisites

//Binary Search Tree

Create inOrder function

If(node = null)

Return

inOrder(node -> left)

print course id, course name, course prerequisites

inOrder(node -> right)

print course id, course name, course prerequisites

**//Menu Pseudocode**

While(userChoice does not equal 4)

Display menu options

1: load Course File

2: Print Course List

3: Print Course

4: exit

Create switch statement with userChoice

Case 1: load course files

Case 2: print sorted course list

Case 3: print course information (course ID)

Case 4: exit

//Run-Time evaluation

//Vector

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Create Vector | 1 | 1 | 1 |
| For each line | 1 | n | n |
| Create Course object | 1 | n | n |
| For each line after row 2 | 1 | n | n |
| If(end of line) | 1 | n | n |
| Next line | 1 | n | N |
| Else(prerequisites exist) | 1 | n | n |
| Append prerequisites | 1 | n | n |
| Total Cost | | | 7n + 1 |
| Runtime | | | O(n) |

//HashTable

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Create node vector | 1 | 1 | 1 |
| Create hash table | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| While(not EOF) | 1 | n | n |
| For(i=0; i<node;++i) | 1 | n | n |
| Create temp1 | 1 | n | n |
| Create temp 2 | 1 | n | n |
| If(3rd value exist) | 1 | n | n |
| Add prereq | 1 | n | n |
| Call insert | 1 | n | n |
| Total Cost | | | 7n + 2 |
| Runtime | | | O(n) |

//Binary Search Tree

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Create Binary Tree | 1 | 1 | 1 |
| Create root = null | 1 | 1 | 1 |
| Insert method | 0 | 0 | 0 |
| If(root = null) | 1 | n | n |
| Else(root less than root) | 1 | n | n |
| If (left = null) | 1 | n | n |
| Add course | 1 | n | n |
| else | 1 | n | n |
| If course less than leaf | 1 | n | n |
| Add left | 1 | n | n |
| else | 1 | n | n |
| Add right | 1 | n | n |
| Else (course num less than root) | 1 | n | n |
| If (right equals null) | 1 | n | n |
| Add right | 1 | n | n |
| else | 1 | n | n |
| If (course less than leaf) | 1 | n | n |
| Add left | 1 | n | n |
| Else (course greater than leaf) | 1 | n | n |
| Add right | 1 | n | n |
| Total Cost | | | 16n+3 |
| Runtime | | | O(n) |

**//Advantages, disadvantages, and recommendation**

**Vectors have the easiest insertion from a file over the other data structures. However in order to add courses at a later time the while vector has to be reworked and shifted in order to be properly sorted. Searching through a vector can also be more difficult in a vector than the other data structures. However you can easily edit specific object in a vector using the .at() function, this can be a big pro as the courses change over time.**

**Hash tables are capable of quickly searching a list through a key. However it is a bit slower to pull from a file and insert into the hash table due to requiring a key and then hashing said object into an available slot in the hash table. There is also no sort function for a hash table since access is instead through a key. This makes a hash table ill suited for this particular system since one of the requirements is printin an alphanumerical list.**

**Binary trees can easily sort objects due to their structure. Even as new courses are added later on the binary tree can easily adjust and keep the list sorted. Searches can also be performed rather quickly. One big drawback is not having direct access to an object instead having to search through the tree.**

**For a recommendation this is quite difficult the hash table can easily be removed bringing us to vector and search tree data structures. According to our runtime evaluation a vector is going to be the ideal data structure. With that being said future scalability isn’t taken into account. As the program grows adding additional courses is likely. Under the current requirements a vector is the better structure to use however the binary tree has scalability advantages. I recommend the binary tree for this reason while it currently has a slower run time, as the system grows sorting and searching with the binary tree will be more efficient than with a vector.**