Coty Mugford

2/15/2024

CS300 – Project 1

**Vector Data Structure**

**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.**

Open the file using ifstream

If the file does not open

Output error message and exit the program

Create a vector named courses

Create a string variable named line

While not at the end of the file

Use getline(filename, line) function to read each line of the file separately

If line does not equal “” (empty)

Use courses.push\_back(line) to store the data from the file to the vector

Close the file

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

Create a struct named Course

Create a string variable named courseNumber.

Create a string variable named courseName.

Create a string variable named prerequisite.

Open the file again to read.

While the file is open

For each line in file

parseLine(line)

If parseLine is not null

Set courseNumber equal to parseLine[0]

Set courseName equal to parseLine[1]

While parseLine is greater than parseLine[1] and does not equal “”

Create variable i and set it equal to 0.

Set prerequisite[i] equal to parseLine[i]

Increment i by 1

Save each struct object to vector Courses

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

Use find function to search for specific courses

Put courses.begin(), courses.end(), and the course name we are trying to find into the find function (find(courses.begin(), courses.end(), courseName)).

If courseName is found within the vector

Output the courses name alongside the courseNumber and its prerequisites

Output any prerequisites course information as well

Else output an error message that states the element was not found

**Hash Table Data Structure**

**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.**

Open the file using ifstream

If the file does not open

Output error message and exit the program

Create a vector named courses

Create a string variable named line

While not at the end of the file

Use getline(filename, line) function to read each line of the file separately

If line does not equal “” (empty)

Use courses.push\_back(line) to store the data from the file to the vector

Close the file

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

Create a struct named Course.

Create a string variable named courseNumber.

Create a string variable named courseName.

Create a string variable named prerequisite.

Use an ifstream to open the file for reading.

While the file is open

For each line in file

Create a parseLine variable that reads each line of the file.

If parseLine is not null.

Set courseNumber equal to parseLine[0].

Set courseName equal to parseLine[1].

While parseLine is greater than parseLine[1] and does not equal “”.

Create variable i and set it equal to 0.

Set prerequisite[i] equal to parseLine[i].

Increment i by 1.

Close the file.

Create a vector called nodes of type node.

Create a variable named tableSize and set it equal to the size of the nodes vector

Create a hashtable of size tableSize.

Create a method called hash with a parameter called key of type int.

Return key modulo tableSize

Insert the key of each node into the hashTable

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

Create a method named PrintAll

Create a node of type Node

Create a course of type Course

Create an unsigned variable named key and set it equal to the hash of the courseNumber (courseNumber modulo tableSize)

For i equals 0, i less than size of the nodes vector and increment i by 1.

Set node equal to the node that is at i in the nodes vector

If key at the node does not equal UINT\_MAX

Output to the user that nodes key, courseNumber, courseName, and all prerequisites

While the next node does not equal null

Set course equal to the next nodes courseNumber

Output to the user that nodes key, courseNumber, courseName, and all prerequisites

Set node equal to the next node

Create a method named Search with parameter courseNumber

Create a Course object course

Create an unsigned variable named key and set it equal to the hash of the courseNumber (courseNumber modulo tableSize)

Create a Node object node and set it equal to the address of the current nodes key

If node is equal to null or that nodes key is equal to UINT\_MAX

Return Course

If node does not equal null and that nodes key does not equal UINT\_MAX and that nodes courseNumber is equal to the searched for courseNumber

Return the course of the node

While node does not equal null

If the nodes key does not equal UINT\_MAX and the nodes courseNumber is equal to the searched for courseNumber

Return that nodes course information including the prerequisite and their information

Set node equal to the next node

Return course

**Binary Tree Data Structure**

**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.**

Open the file using ifstream

If the file does not open

Output error message and exit the program

Create a vector named courses

Create a string variable named line

While not at the end of the file

Use getline(filename, line) function to read each line of the file separately

If line does not equal “” (empty)

Use courses.push\_back(line) to store the data from the file to the vector

Close the file

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

Create a struct named Course.

Create a string variable named courseNumber.

Create a string variable named courseName.

Create a string variable named prerequisite.

Use an ifstream to open the file for reading.

While the file is open

For each line in file

Create a parseLine variable that reads each line of the file.

If parseLine is not null.

Set courseNumber equal to parseLine[0].

Set courseName equal to parseLine[1].

While parseLine is greater than parseLine[1] and does not equal “”.

Create variable i and set it equal to 0.

Set prerequisite[i] equal to parseLine[i].

Increment i by 1.

Close the file.

**Create a binary tree class.**

Create a root node that points to null.

Create an insert method.

If the root is null

set the root to be equal to the current course.

If the root is greater than the course number.

If the left node is null.

The new node becomes the left node.

Else recurse through the left node passing in the left node and the course number.

Else

If the right node is null.

The new node becomes the right node.

Else recurse through the right node passing in the left node and the course number.

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

Create a method called inOrder with a node parameter.

If node is not null

Recursively call inOrder passing in the node->left

Output to the display the current nodes course number, course name, and any prerequisites associated with the course number

Recursively call inorder passing in the node->right

Create a method called search with a course number parameter.

Create a new node called current and set it equal to root.

While current does not equal null.

If the current course number is equal to the passed in course number.

Return the course information.

If the passed in course number is less than the current course number.

Set current equal to current->left.

Else.

Set current equal to current->right.

Return course number.

**Menu Pseudocode**

Create an int variable named choice

While choice does not equal 9

Output 1.Load File, 2. Display Course List, 3. Search For Specific Course, 9. Exit

Create a switch with choice as a parameter

Case 1 (Load Courses)

Call loadCourses method

Case 2 (Display Course List)

Call the PrintAll method on the vector of courses (Call the inOrder method if using Binary Tree)

Case 3

Ask user to enter a courseNumber

Get input from user about courseNumber

Call the search function with parameter courseNumber

If the courseNumber is found

Display the course information including all prerequisites

Else tell the user that the course was not found

If 9 is entered, output “Good Bye” to the user

**Quicksort**

Create int function called partition with parameters Vector<Course>& courses, int begin, int end

Create an int variable named low and set low to begin

Create an int variable named high and set high to end

Create an int variable named midpoint and set it equal to begin + (end – begin) / 2

Create a string variable pivot and set it equal to courses.at(midpoint).courseName

Create a bool variable named done and set it equal to false

While not done

While low index is less than the pivot

Increment low

While pivot is less than the high index

Decrement high

If low is greater than or equal to high

Set done equal to true

Else

Swap(courses.at(low) and courses.at(high))

Increment low

Decrement high

Return high

Create a quicksort function with parameters (Vector<course>& courses, int begin and int end)

Create an int variable named mid and set it equal to 0

If begin is greater than or equal to end

Return

Set mid equal to partition with parameters courses, begin and end

Recursively call quicksort with parameters courses, begin and mid

Recursively call quicksort with parameters courses, mid + 1 and end

**Selection Sort**

Create a selectionSort function with parameter Vector<course>& courses

Create an int variable named I and set it to 0

Create an int variable named J and set it to 0

Create an int variable named min and set it equal to 0

For i = 0, i < courses.size() – 1, i++

Set min equal to i

For (j equal to i + 1, j < courses.size(), j++

If element at j in courses is less than min

Set min equal to j

If min does not equal i

Swap(courses at i and courses at min)

**Display Courses**

For int i = 0, i less than courses.size(), i++

Output the course.courseNumber, course.courseName, course.prerequisites

Output endline

Break

**Runtime Evaluation - Reading the File and Creating Course Objects**

**Vector**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Parse Each line in the file | 1 | n | n |
| Set courseNumber equal to parseLine[0] | 1 | n | n |
| Set courseName equal to parseLine[1] | 1 | n | n |
| While parseLine does not equal null and is greater than parseLine[1] | 1 | n | n |
| After a full line is read, save each course to Courses vector | 1 | n | n |
| Total Cost | | | 5n |
| Runtime | | | O(n) |

**Hash Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Parse Each line in the file | 1 | n | n |
| Set courseNumber equal to parseLine[0] | 1 | n | n |
| Set courseName equal to parseLine[1] | 1 | n | n |
| While parseLine is greater than parseLine[1] and is not null | 1 | n | n |
| Create a key for each course by taking each courseNumber modulo the size of the hashtable | 1 | n | n |
| Insert the key of each node into the hashtable | 1 | 1 | 1 |
| Total Cost | | | 5n + 1 |
| Runtime | | | O(n) |

**Binary Tree**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| For each line in the file | 1 | n | n |
| Set courseNumber equal to parseLine[0] | 1 | n | n |
| Set courseName equal to parseLine[1] | 1 | n | n |
| While parseLine is greater than parseLine[1] and is not null | 1 | n | n |
| Set prerequisite equal to parseLine[i] | 1 | n | n |
| If the root is null, set the root to be equal to the current course | 1 | 1 | 1 |
| If the root is greater than the course number | 1 | 1 | 1 |
| If the left node is null | 1 | 1 | 1 |
| Set the left node equal to the new node | 1 | 1 | 1 |
| Else recurse through the left node passing in the left node and course number | 1 | n | n |
| Else if the right node is null, the new node becomes the right node | 1 | n | n |
| Else recurse through the right node passing in the right node and the course number | 1 | n | n |
| Total Cost | | | 8n + 4 |
| Runtime | | | O(n) |

**Advantages and Disadvantages of Each Structure**

Vectors tend to be a basic data structure and are best used when data size is not fixed, when insertion and deletion does not need to happen from the middle of the vector, when you have smaller amounts of data, and when you need to randomly get information about an item from anywhere in the vector.

Hash Tables are the bread and butter for efficient programs in terms of time complexity. Due to their bucket system, it is much easier to search for data, delete specific data and insert new data. On top of that, they also tend to be much more secure data structures and are the basis for data encryption. One major con of using hash tables is when you have many collisions occur. As the number of collisions increases, the time complexity becomes more linear. To avoid collisions, you can increase the size of the hash table so that each bucket only has 1 item, but if you have to increase the size of an already full hash table, it will need to recalculate each and every data item in the hash table and readjust their bucket. This can become very inefficient in terms of space complexity.

Binary Trees tend to be very easy to use when searching for data, quickly inserting new data, time efficient when sorting due to the binary tree being inherently sorted and supports multiple ways to traverse the tree. On top of this, binary trees tend to be space efficient because they only use up as much space as they need to, unlike hash tables that often are set to a higher table size than the number of data items. Some of the disadvantages are that it tends to have slower search speeds than that of a hash table.

**Data Structure Recommendation**

The data structure that I would like to use is the binary tree. The reason I chose the binary tree is that we want to have the ability to print out the list of courses alphanumerically. Since the binary tree is inherently sorted, this is going to be a non-issue for this. Alongside that, while it may not be as efficient as a hash table when it comes to search, add and remove functions, it is still in the ballpark. Hash tables are difficult to sort due to the keys being hashed. Vectors are easy to sort but are very difficult to add and delete items from the middle of the vector. On top of that, vectors are best when used with small data sizes. Therefore I think that it would be best to use a balanced binary tree for this project.