Joseph Grzywinski

CS-300

04/14/24

**Menu:**

Main(){

Print(“Select an option:”

“1. Load course information”

“2. Print Course List”

“3. Find a Course”

“4. Exit”)

While(user input does not equal 4) {

If(input equals 1){

Print(enter a file name)

Call LoadCourses(file name)

}

If(input equals 2){

Call PrintAll()

}

If(input equals 3){

Print(enter a course number)

Call FindCourse(course number)

}

If(unput equals 4){

Exit the application

}

}

}

**Read file functions:**

Parser {

If a valid file {

Open file

While file is open {

Get line of file

If there are 2 parameters or 3 parameters

For length of line {

If character equals “,”

Add string to a new column

}

}

Close file

}

}

void LoadCourses(){

Parser()

For column in row {

Create new Course object

Course number is column 1

Course name is column 2

If number of columns is greater than 2 {

For columns greater than 2

{

Add object to prerequisite array within object

}

}

Insert(Course)

}

**Vector functions:**

Void FindCourse(courseNumber) {

For course in vector<Course>courses {

If courseNumber equals course.courseNumber{

Print “Course Number: ” + courseNumber

Print “Course Name: ” + courseName

Print “Prerequisites: ”

For prereq in prerequisites {

Print prereq

}

}

}

}

**Hash table functions:**

void Insert(Course) {

Set default size for hash table

Key equals course number % table size

Set prevNode to the node at Key

If prevNode equals nullptr {

newNode = Course, key

insert(key, newNode)

}

Else {

If prevNodes key equals UINT\_MAX {

prevNodes key equals key

prevNodes course equals course

next node after prevNode equals nullptr

}

Else {

While next node after prevNode does not equal nullptr{

prevNode equals next node

}

next node after prevNode equals new node(course, key)

}

}

}

Void PrintAll(){

For all nodes beginning to end {

If the nodes key does not equal nullptr{

Print course details from that node

nextNode equals the next node in the hash table

While nextNode does not equal nullptr {

Print course details from current node

Set nextNode equal to the following node in the hash table

}

}

}

}

**Binary tree functions:**

void Insert(Node, Course) {

If root of binary tree is null {

Root node is equal to Course

}

Else {

If left node equals nullptr and its less than the parent node{

Set left node to the new Course object

}

Else If the right node equals nullptr and its greater than the parent node{

Set right node to the new Course object

}

Else If the Course number of the new course is less than the current node{

Call insert method again using the left child of current node

}

Else If the Course number of the new course is greater than the current node{

Call insert method again using the right child of current node

}

}

}

Void displayCoursesInOrder(Node)

If node does not equal nullptr{

Call displayCourses(left node)

Print course information

Call displayCourses(right node)

}

**Parser:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Breakdown** | **Cost** | **Iterations** | **Final Cost** |
| If a valid file | 1 | 1 | 1 |
| Open file | 1 | 1 | 1 |
| While file is open | 1 | n | n |
| Get line of file | 1 | n | n |
| For length of line | 1 | p | p |
|  |  |  |  |
| **Worst Runtime** | 1 | n\*p | O(n\*p) |

**LoadCourses:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Breakdown** | **Cost** | **Iterations** | **Final Cost** |
| For column in row | 1 | n | n |
| If number of columns is greater than 2 | 1 | 1 | 1 |
| For columns greater than 2 | 1 | p | p |
|  |  |  |  |
| **Worst Runtime** | 1 | n\*p | O(n\*p) |

**Vector FindCourse:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Breakdown** | **Cost** | **Iterations** | **Final Cost** |
| For course in vector<Course>courses | 1 | n | n |
| If courseNumber equals course.number | 1 | 1 | 1 |
| For prereq in prerequisites | 1 | p | p |
|  |  |  |  |
| **Worst Runtime** | 1 | n\*p | O(n\*p) |

**Hash table Insert:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Breakdown** | **Cost** | **Iterations** | **Final Cost** |
| Set default size for hash table | 1 | 1 | 1 |
| Key equals course number % table size | 1 | 1 | 1 |
| Set prevNode to the node at Key | 1 | 1 | 1 |
| If prevNode equals nullptr | 1 | 1 | 1 |
| Else If prevNodes key equals UINT\_MAX | 1 | 1 | 1 |
| While next node after prevNode does not equal nullptr | 1 | n | n |
|  |  |  |  |
| **Worst Runtime** | 1 | n | O(n) |

**Hash table Print:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Breakdown** | **Cost** | **Iterations** | **Final Cost** |
| For all nodes beginning to end | 1 | n | n |
| If the nodes key does not equal nullptr | 1 | 1 | 1 |
| While nextNode does not equal nullptr | 1 | n | n |
|  |  |  |  |
| **Worst Runtime** | 1 | n | O(n) |

**Binary Tree Insert:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Breakdown** | **Cost** | **Iterations** | **Final Cost** |
| If root of binary tree is null | 1 | 1 | 1 |
| Else | 1 | Log n | Log n  \*if tree is balanced\* |
| If left node equals nullptr and its less than the parent node | 1 | 1 | 1 |
| Else If the right node equals nullptr and its greater than the parent node | 1 | 1 | 1 |
| Else If the Course number of the new course is less than the current node | 1 | 1 | 1 |
| Else If the Course number of the new course is greater than the current node | 1 | 1 | 1 |
|  |  |  |  |
| **Worst Runtime** | 1 | n | O(n)  \*if tree is unbalanced\* |

**Binary Tree Print:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Breakdown** | **Cost** | **Iterations** | **Final Cost** |
| If node does not equal nullptr | 1 | 1 | 1 |
| Call displayCourses(left node) | 1 | Log n  \*if tree is balanced\* | Log n  \*if tree is balanced\* |
| Print course information | 1 | 1 | **1** |
| Call displayCourses(right node) | 1 | Log n  \*if tree is balanced\* | Log n  \*if tree is balanced\* |
|  |  |  |  |
| **Worst Runtime** | 1 | n | O(n)  \*if tree is unbalanced\* |

**Vector Advantages:**

Vectors are quite easy to use especially when it comes to small datasets and the number of elements that are being added to them are known. Implementation is very straightforward when it comes to using a vector as well since it uses linear traversal it is easy to understand and create.

**Vector Disadvantages:**

If the data within the vector is changing regularly via adding or deleting elements, vectors are very low performing since remaining elements will have to be shifted within the vector whenever there is a change. Searching can also have an extremely high runtime especially if you do not know the index of the element you are looking for. With linear traversal, there is a possibility the program will have to search every single element within the vector to return a value.

**Hash Table Advantages:**

One of the biggest advantages hash tables has over the other two data structures is that it has a consistently faster runtime. This is because hash tables use a key value pair when assigning objects into the dataset. The key is derived from using a hash function which will assign its position within the table. So, when trying to locate a specific object the hash table can utilize this hash function to find the correct object without having to check each key within the table. Another key advantage is that has tables will resize themselves based on how many positions are needed.

**Hash Table Disadvantages:**

One disadvantage of using a hash table is having to deal with collisions within specific keys. When creating a program with a hash table, you will need to develop a method for solving these collisions. Another disadvantage is ordering the values of a hash table can be difficult since items will most likely not keep their order from when they are inserted into the table.

**Binary Tree Advantages:**

In our scenario the biggest advantage of a binary tree is that it keeps a sorted order, so we will not have to implement a separate function to sort the order when we print the courses. In a balanced binary tree, insertion has an average runtime of log n which for bigger datasets is a good runtime. This type of runtime is great for inserting and deleting objects which in our case we will need to insert courses from our text file to run the application.

**Binary Tree Disadvantages:**

The biggest downside to using a binary tree is when the tree becomes unbalanced. This causes the worst runtime to drop to O(n) and can affect operations like inserting and deleting elements. This is something that if you are changing information a ton could become a problem if you need objects in a specific order.

**My Dataset Suggestion:**

For this program I would recommend going with a binary search tree to hold the data. Since this is a set course structure based on a pre-existing document, the binary tree should stay balanced since there are no functions to delete courses from the program or to insert a new singular course into the tree. With a balanced tree we should be able to achieve an average runtime speed of O(log n) for all functions of the program. Binary trees will be great for structuring our data in an alphanumerical order as well since they are ordered datasets. This will make printing all courses in order much more efficient while also achieving good runtime speeds as well. The final function for my binary tree pseudocode called “displayCoursesInOrder” will be able to print the tree in the correct order based on their course number.