Tristan Maloy

12/8/2024

CS 300

Project One

**Vector**

Void loadCourses(string cvsPath, Vector<course> courses)

Initialize CVS Parser

csv::Parser file = csv::Parser(csvPath);

Try loop to read rows of CSV file

For loop to read rows and columns in CSV file

For i = 0; i < file.rowCount(); ++i

For j = 0; j < file.columnCount(); ++j

Initialize courses using data from current row

Define data structure

course{String courseNumber, String courseTitle, String coursePrerequisites}

course.courseNumber = file[i][j]

course.courseTitle = file[i][j]

course.coursePrerequisites = file[i][j]

list->append(course)

Create Node\* newNode = new Node(course)

if head == nullptr

new node becomes head and tail

else

current tail points to newNode

tail becomes newNode

Increase vector size

catch file format errors

if course has less than 2 parameters

throw error

if course prerequisite does not exist as another course number

throw error

End Try

searchCourse(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 prerequisite course information

else if courseNumber not found

print courseNumber + “not found”

PrintList() //using mergesort for best run time O(nlogn)

Node \*MergeSort(Node \*head) //see below for mergesort

Create current node = head

While current node is not nullptr

Output course information

Current node points to current next node

Output new line

Node \*MergeSort(Node \*head)

If list empty or contains only one node list is sorted

If head == nullptr or head->next == nullptr

Return head

Split list into two halves

Node \*second = split(head)

Recursively sort each half

Head = Mergesort(head)

Second = MergeSort(second)

Merge two sorted halves

Return merge(head, second)

**Hash Table**

Void loadCourses(string cvsPath, HashTable\* hashTable)

Initialize CVS Parser

csv::Parser file = csv::Parser(csvPath);

Try loop to read rows of CSV file

For loop to read rows and columns in CSV file

For i = 0; i < file.rowCount(); ++i

For j = 0; j < file.columnCount(); ++j

Initialize courses using data from current row

Define data structure

course{String courseNumber, String courseTitle, String coursePrerequisites}

course.courseNumber = file[i][j]

course.courseTitle = file[i][j]

course.coursePrerequisites = file[i][j]

hashTable->Insert(course)

create key = hash(atoi(course.courseNumber.c\_str())))

create node\* currentNode = &(nodes.at(key))

if currentNode is nullptr

assign node to key position

else if node = UINT\_MAX

set node to key, currentNode->course = course, and currentNode->next = nullptr

else while currentNode->next is not nullptr

currentNode becomes next node

currentNode->next = new Node(course, key)

catch file format errors

if course has less than 2 parameters

throw error

if course prerequisite does not exist as another course number

throw error

End Try

void searchCourse(HashTable<Course> courses, String courseNumber)

create key = &(nodes.at(key))

if node != nullptr and nodes key != UINT\_MAX and nodes courseNumber matches

return node for course

if node is nullptr OR nodes key is UINT\_MAX

return course

while node != nullptr

if current node matches

return current node

node = node->next

return course

PrintTable()

For iterate from beginning to end of table

If key != UINT\_MAX //initial value

Output course information

Node becomes next iterater

While node != nullptr //every value after initial

Output course information

Node points to next node

**Tree**

Void loadCourses(string cvsPath, BinarySearchTree\* bst)

Initialize CVS Parser

csv::Parser file = csv::Parser(csvPath);

Try loop to read rows of CSV file

For loop to read rows and columns in CSV file

For i = 0; i < file.rowCount(); ++i

For j = 0; j < file.columnCount(); ++j

Initialize courses using data from current row

Define data structure

course{String courseNumber, String courseTitle, String coursePrerequisites}

course.courseNumber = file[i][j]

course.courseTitle = file[i][j]

course.coursePrerequisites = file[i][j]

bst->Insert(courses)

if root == nullptr

root is equal to new node(courses)

else

add node root and courses

catch file format errors

if course has less than 2 parameters

throw error

if course prerequisite does not exist as another course number

throw error

End Try

void searchCourse(Tree<Course> course, String courseNumber)

Create new node current = root

While current node is not nullptr

If course.courseNumber matches courseNumber

Output course number, course title, course prerequisites

If course.courseNumber is less than courseNumber

Traverse left root

Current = current->left

If course.courseNumber is greater than courseNumber

Traverse right root

Current = current->right

Course course

Return course

PrintTree(Node\* node) //inOrder pass root

If node != nullptr

Pass node->left to inOrder()

Output node to the left

Pass node->right to inOrder()

**Menu**

Initialize choice = 0

While choice != 9

Output “Menu:/n1. Load data\n2. Print all courses\n3. Print specific course\n9. Exit the program.”

Input choice

Create switch case statement

Switch (choice)

Case 1:

loadCourses(cvsPath, &courseList)

Output coursList.size() + “ courses have been loaded”

Break

Case 2:

courseList.PrintList() to print entire list or

courseList.PrintTable() to print table or

courseList.PrintTree() to print tree

break

Case 3:

Input courseNumber

searchCourse(courseNumber)

if (course.courseNumber.empty())

displayCourse(course)

else

Output “Course not found”

break

**Evaluation**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Vector | Hash Table | Binary Search Tree |
| Load Data | O(N^2) | O(N^2) | O(N^2) |
| Search | O(N) | O(N) | O(logN) |
| Print sorted | O(NlogN) | O(N) | O(N) |

The function to load data was O(N^2) as there were nested loops to traverse all rows and columns of the data file. The search function for the vector and hash table were both O(N) which is linear time complexity which means that the function iterates over an input size of N. Using the search function for the BST gives a runtime complexity of O(logN) because the program only runs for half the input size since the values are divided by the left and right nodes. To sort and print the vector linked list, we can use a merged sort which has an O(NlogN) runtime complexity which requires more memory and is not that fast at processing. The hash table and binary search tree should be implemented as sorted already or by using the inOrder() function so the runtime complexity for those are O(N).

Based on runtime complexity, I would use a binary search tree to implement in my code. With an O(N^2) for loading the data, O(logN) for searching for a course, and O(N) for sorting and printing all courses and their information, it has a quicker runtime than the other two. I also found that the BST was easier to understand how to code and traverse than the linked list and the hash table. A hash table would be second best as far as runtime complexity. A direct hash function could be used to prevent collisions but that would create a rather big index.