Caio Mauro

CS300

Project 1

**Reading File:**

Use fstream to open the file

Create method void loadCourses(string csvPath, dataStructure)

Make a call to open the file; if the return value is "-1", the file is not found, otherwise, the file is found

While it is not the End Of File (EOF), read each line

Close the file

**Hold Course Information:**

IF there are fewer than two values in a line, return ERROR

ELSE read parameters

IF there is a third or more parameter:

IF the third or more parameters are found elsewhere in the first parameter, continue

ELSE return Error

Create struct Course{}

Create Identifiers: Course ID, Course Name, Prerequisite

*//Vector*

vector<Course> loadCourses(string csvPath)

for (int i = 0; i < file.rowCount(); i++) {

Create a data structure and add to the collection of courses

Course course;

course.courseId = file[i][1];

course.name = file[i][0];

while not end of line course.prerequisite. = file[i][8];

courses.push\_back(course);

}

*//HashTable*

Create Hashtable

Create Node struct

Course course

Unsigned int key

Vector<Node> nodes

Define tableSize

Unsigned int has(int key)

Create insert method void HashTable::Insert(Course course)

create the key for the given course, search for a node with the key value

if no entry found for the key, assign this node to the key position

else if the node is used, assign the old node key to MAX\_UNIT, set to key, set the old node to course, and the old node next to a null pointer

else find the next open node

add the new newNode to end

void loadCourses(string csvPath, HashTable\* hashTable)

loop to read rows of a CSV file

for (unsigned int i = 0; i < file.rowCount(); i++) {

Create a data structure and add to the collection of courses

Course course;

course.courseId = file[i][1];

course.name = file[i][0];

while not end of line course.prerequisite. = file[i][8];

hashTable->Insert(course);

}

*//Tree*

Define a binary search tree to hold all courses

BinarySearchTree\* bst;

bst = new BinarySearchTree();

Course course;

Create add node method void BinarySearchTree::addNode(Node\* node, Course course)

If the root is null, add root

if the node is less than the root then add to the left

if no left node, this node becomes left

if the node is greater than the root add right

if no right node, this node becomes right

void loadCourses(string csvPath, BinarySearchTree\* bst)

loop to read rows of a CSV file

for (unsigned int i = 0; i < file.rowCount(); i++) {

Create a data structure and add to the collection of courses

Course course;

course.courseId = file[i][1];

course.name = file[i][0];

while not end of line course.prerequisite. = file[i][8];

bst->Insert(course);

}

**Print Course Information and Prerequisiteuisites:**

*//Vector*

Create method void printCourseInformation(Vector<Course> courses, String courseId)

Get input for courseId

While the vector is not empty

if the input is the same as courseId

output course.courseId << output course.name

while (prerequisite = true) output course.prerequisite

*//HashTable*

Create method void printCourseInformation(Hashtable<Course> courses, String courseId)

Get input for courseId

Assign key = courseId

Assign node to the node.at(key) if the current node matches the key

Return course, displayCourse(nodes[key].course)

If the node points to null, return null

Else while the node is not Null, check against the key

If the key matches the courseId, Return course, displayCourse(nodes[key].course)

Point to the next node

*//Tree*

Create method void printCourseInformation(Tree<Course> courses, String courseId)

Get input for courseId

Assign the current node to root

While the current is not NULL

If course.courseId matches the current

Return current, output course.courseId << output course.name

while (prerequisite = true) output course.prerequisite

If courseId is less than root

Set current to left

Else set current to right

**Menu:**

Set choice to 0;

Create a while loop for the menu. While choice is not equal to 4

Output menu choices (1. Load Course File, 2. Print Course List 3. Print Individual Course 4.Exit)

Create a switch(choice)

Case 1: loadCourses(courseFile, dataStructure) FIXME: use the structure of the data structure chosen

Case 2: printSorted(courses) call function to print a sorted class list

Case 3: printCourseInformation(courseId)

Case 4: Terminate Program

**Print Sorted List:**

*//Vector*

Create a sorted print method printSorted(courses)

Create a partition method int partition(vector<Course>& courses, int begin, int end)

Set lowIndex to the first element, set highIndex to the last element

Set midpoint to lowIndex + (highIndex - lowIndex) / 2

Set pivot to midpoint

Decrement highIndex while the pivot is less than highIndex

Swap lower values to the left of the pivot, higher values to the right of the pivot

Set temp value to low index

Set low index to high index

Set high index to temp

Create a quicksort method void quickSort(vector<Course>& courses, int begin, int end)

Set mid to 0, lowIndex to begin, highIndex to end

If begin >= end, return

Set lowEndIndex to partition(courses, lowIndex, highIndex)

Make a recursive call to quicksort

quickSort(courses, lowIndex, lowEndIndex);

quickSort(courses, lowEndIndex + 1, highIndex)

Create a display course method void displayCourse(Course course) {

cout << course.courseId << ": " << course.name << " | " << course.prerequisite << endl;

}

Loop through the vector to display courses

for (int i = 0; i < courses.size(); ++i)

displayCourse(courses[i])

*//Tree*

Create an inOrder method void BinarySearchTree::inOrder(Node\* node)

If (node != Null)

Check the most left side first

inOrder(node->left)

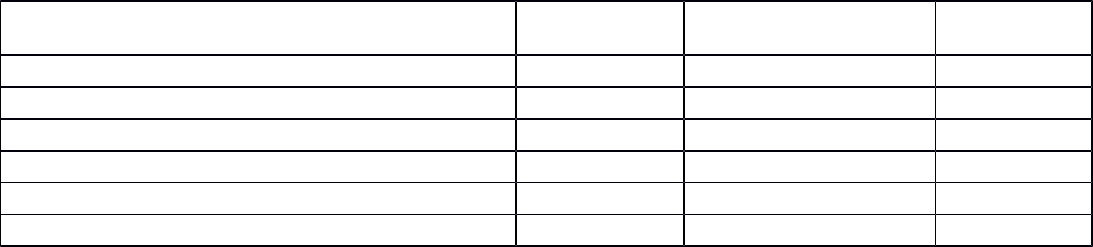
cout << course.courseId << ": " << course.name << " | " << course.prerequisite << endl;

check the next right leaf

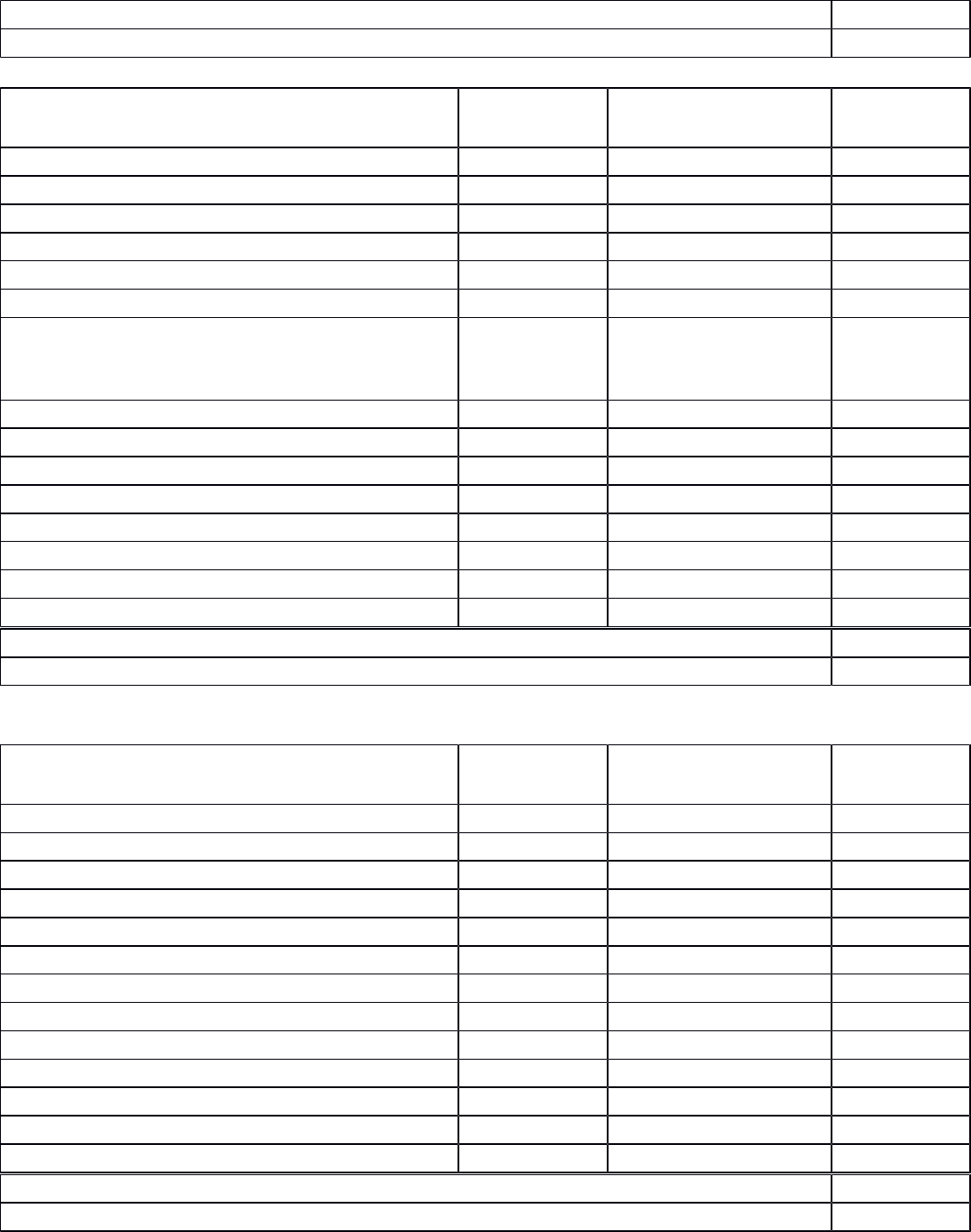
inOrder(node->right)

cout << course.courseId << ": " << course.name << " | " << course.prerequisite << endl;

**Runtime Analysis:**



|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | **Line Cost** | **Times Executes** | **Total Cost** |
| **Create Vector** | **1** | **1** | **1** |
| **For each line in file** | **1** | **n** | **n** |
| **Create vector course object** | **1** | **n** | **N** |
| **While prerequisiteuisite exists** | **1** | **n** | **n** |
| **Append prerequisiteuisite** | **1** | **n** | **n** |
| **Pushback course object** | **1** | **N** | **N** |
|  |  | **Total Cost** | **5n+1** |
|  |  | **Runtime** | **O(n)** |
| **Hash Table** | **Line Cost** | **#Times Executes** | **Total Cost** |
| **Create hash table** | **1** | **1** | **1** |
| **Insert method** | **0** | **0** | **0** |
| **Create key for course** | **1** | **n** | **n** |
| **If no entry found for key** | **1** | **n** | **n** |
| **Assign node to key** | **1** | **n** | **n** |
| **Else** | **1** | **n** | **n** |
| **Assign old node key to MAX\_UNIT,** | **4** | **n** | **4n** |
| **set to key, set old node to course and old** |  |  |  |
| **node next to null pointer** |  |  |  |
| **Else** | **1** | **n** | **n** |
| **Find the next open node** | **1** | **n** | **n** |
| **Add new newNode to end** | **1** | **n** | **n** |
| **For each new line in file** | **1** | **n** | **n** |
| **Create vector course object** | **1** | **n** | **n** |
| **While prerequisite exists** | **1** | **n** | **n** |
| **Append prerequisite** | **1** | **n** | **n** |
| **Insert course object** | **1** | **n** | **n** |
|  |  | **Total Cost** | **16n+1** |
|  |  | **Runtime** | **O(n)** |
| **Tree** | **Line Cost** | **#Times Executes** | **Total Cost** |
| **Add node method** | **0** | **0** | **0** |
| **If root is null, add root** | **1** | **1** | **1** |
| **If node is less than root then add to left** | **1** | **n** | **n** |
| **If no left node** | **1** | **n** | **n** |
| **This node becomes left** | **1** | **n** | **n** |
| **If node is greater than root add to right** | **1** | **n** | **n** |
| **If no right node** | **1** | **n** | **n** |
| **This node becomes right** | **1** | **n** | **n** |
| **For each line in file** | **1** | **n** | **n** |
| **Create vector course object** | **1** | **n** | **n** |
| **While prerequisite exists** | **1** | **n** | **n** |
| **Append prerequisite** | **1** | **n** | **n** |
| **Insert course object** | **1** | **n** | **n** |
|  |  | **Total Cost** | **11n+2** |
|  |  | **Runtime** | **O(n)** |



Each data structure has distinctive advantages and drawbacks based on what the course program requires. The usen of a limit us to a slower linear search through the list for a specific course, requiring the program to look at each object until a match is identified. Still, the vector method is very quick when it comes to its speed when reading the file and adding course objects. It brings a straightforward and efficient approach. Out of the three data structures looked at, the vector method has the shortest total runtime at 5n+1.

In contrast, hash tables offer the advantage of rapid list searches by employing keys, enabling quick retrieval and display the data needed. However, the initial creation of the list and initialization for the insertion points for each course brings a slower creation time for the data structure. Hash tables don't naturally organize themselves. To make a list of courses in alphabetical order, we need to take out the information, arrange it, and then display it again. These qualities make hash tables not the best choice for the current course program.

Creating a tree structure, specifically binary trees, provides a notable advantage when compared to using vectors. The strength lies in their ability to sort data quickly. While the process isn't as simple as hash tables, binary trees excel in speedy search operations, boasting a time complexity of O(h), where 'h' is the height of the tree. This characteristic makes binary trees a favorable choice when efficiency in sorting and searching is crucial for a program like ours.

In consideration of the analysis of each data structure, I would recommend for the implementation of a vector sort in this project. The ability to employ quicksort for efficiently printing the entire catalogue is seen as more valuable, despite the minor increase in search time. In my assessment, the vector stands out as the optimal choice for this program.

lOMoARcPSD|23665018