Dan Peterson

13FEB2024

Project One

# Psuedocode

**READING FILE**

Use FSTREAM to enable opening file

METHOD VOID load Courses(STRING csvPath, dataStructure)

Make CALL to open file

IF the return value is “-1” \\ file not found

Set END OF FILE to TRUE

ELSE \\File is found

SET EOF to FALSE;

WHILE EOF is FALSE

Read each line

IF there are less than two values in a line RETURN ERROR;

ELSE read parameters

IF third or more is in first parameter or elsewhere CONTINUE

ELSE return ERROR

CLOSE FILE

**Hold Course Information**

Create struct Course {}

Create Identifiers: courseID, courseName, Prerequisite

**VECTOR**

Vector<Coursse> loadCourses(STRING, csvPath)

FOR (int I = 0; I < file.rowCount(); i++)

Create data structure in order to add to 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);

**HASH TABLE**

**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 node with the key value**

**IF no entry found for the key**

**assign this node to the key position**

**ELSE IF node is used**

**assign old node key to UNIT\_MAX,**

**set to key,**

**set old node to course and old node NEXT to NULL POINTER**

**ELSE find the next open node**

**add new newNode to end**

**void loadCourses(string csvPath, HashTable\* hashTable)**

**loop to read rows of file**

**FOR (NSIGNED 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.prereq. = file[i][8];**

**hashTable->Insert(course);**

**TREE**

DEFINE binary search tree to hold all courses

BinarySearchTree\* bst;

Bst = new BinarySearchTree()

Course course;

add node METHOD VOID BinarySearchTree::addNode(Node\* node, Course course)

IF root is null add root

IF node is less than root then add to left

IF no left node

Current node becomes left

IF node is greater than root then add to right

IF no right node

Current node becomes right

**VOID loadCourses(STRING csvPath, BinarySearchTree\* bst)**

Loop to read rows of file

FOR (UNSIGNED INT 1 = 0; I < file.rowCount(); i++)

Create data structure and add 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**

**VECTOR**

METHOD VOID printCourseInformation(Vector<Course> courses, STRING courseID)

ASK for USER INPUT

WHILE VECTOR is not empty

IF USER INPUT is equal to courseID

OUTPUT course.courseID, OUTPUT cpurse.name

WHILE course has prerequisites

OUTPUT course.prerequisite

**HASH TABLE**

METHOD VOID printCourseInformation(Hashtable<Course> courses, STRING courseId)

ASK for USER INPUT (courseId)

Assign Key = courseId

Assign NODE to node.at(key)

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

IF NODE points to NULL RETURN NULL

IF CURRENT NODE equals key

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

IF NODE points to NULL RETURN NULL

ELSE WHILE NODE is not NULL compare to key

IF key matches courseID RETURN course, displayCourse(nodes[key].course)

POINT to NEXT NODE

**TREE**

METHOD VOID printCourseInformation(Tree<Course> courses, STRING courseId)

ASK for USER INPUT (courseId)

Assign CURRENT NODE to ROOT

WHILE CURRENT is not NULL

IF course.courseId equals CURRENT

RETURN current OUTPUT course.courseId OUTPUT course.name

WHILE (course has prerequisite)

OUTPUT course.prerequisite

IF courseId is less than root

Set CURRENT to LEFT

ELSE set CURRENT to RIGHT

**MENU**

Set CHOICE to 0

WHILE choice is not 4

OUTPUT menu choices (1 Load Course File 2 Print Course List 3 Print Individual Course 4 Exit)

Switch (choice)

Case 1: loadCourses(courseFile, dataStructure)

Case 2: printSorted(courses) CALL FUNCTION PRINT SORTED LIST

Case 3: printCourseInformation(courseId)

Case 4: Exit Program

**Print Sorted List**

**VECTOR**

METHOD printSorted(courses)

METHOD INT partition(vector<Course>& courses, INT begin, INT end)

Set lowIndex to first ELEMENT, sethighIndex to last ELEMENT

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

Set pivot to midpoint

DECREMENT highIndex while pivot is less than highIndex

SWAP lower values to left of pivot, higher values to right

Set temp value to lowIndex

Set lowIndex to highIndex

Set highIndex to temp

QUICKSORT METHOD VOID quicksort(vector<Course>& courses, INT begin, INT end)

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

IF begin is greater than or equal to end RETURN

Set LowEndIndex to partition(courses, lowIndex,highIndex)

RECURSIVE CALL QUICKSORT

quicksort(courses, lowIndex, lowEndIndex)

quicksort(courses, lowEndIndex +1, highIndex)

DISPLAY COURSE METHOD VOID displayCourse course)

OUTPUT course.courseId : course.name | course.prerequisite

LOOP through VECTOR to OUTPUT all courses

FOR (INT I = 0; I < course.size(); ++i)

displayCourse(courses[i]

**TREE**

inOrder METHOD coid BinarySearchTree::inOrder(Node\* node)

IF node is not NULL

Check LEFT

inOrder(node->left)

OUTPUT course.courseId : course.name | course.prerequisite

Check RIGHT

inOrder(node->right)

OUTPUT course.courseId : course.name | course.prerequisite

# Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| VECTOR | LINE COST | #TIMES EXECUTED | TOTAL COST |
| Create Vector | 1 | 1 | 1 |
| For each line | 1 | n | n |
| Create course item | 1 | n | n |
| With prerequisites | 1 | n | n |
| APPEND prerequisites | 1 | n | n |
| Pushback course item | 1 | n | N |
| TOTAL COST | | | 5n+1 |
| RUNTIME | | | O{n) |

|  |  |  |  |
| --- | --- | --- | --- |
| HASH TABLE | LINE COST | #TIMES EXECUTED | 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 UNIT\_MAX,  set to key, set old node to course and old  node next to null pointer  Assign old node key to UNIT\_MAX,  set to key, set old node to course and old  node next to null pointer  Assign old node key to UNIT\_MAX, set to key, set old node to course and old node next to null pointer | 4 | n | 4n |
| 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 item | 1 | n | n |
| While prereq exists | 1 | n | n |
| Append prereq | 1 | n | n |
| n  Insert course item  n  Insert course item  Insert course item | 1 | n | n |
| TOTAL COST | | | 16n+1 |
| RUN TIME | | | O(n) |

|  |  |  |  |
| --- | --- | --- | --- |
| TREE | LINE COST | #TIMES EXECUTED | TOTAL COST |
| Add node method | 0 | 0 | 0 |
| If root is null, add root | 1 | 1 | 1 |
| If node is less than root add left | 1 | n | n |
| If no left node | 1 | n | n |
| node becomes left | 1 | n | n |
| If node is greater than root add right | 1 | n | n |
| If no right node | 1 | n | n |
| node becomes right | 1 | n | n |
| For each line in file | 1 | n | n |
| Create vector course item | 1 | n | n |
| With prerequisites | 1 | n | n |
| Append prerequisites | 1 | n | n |
| Insert course item | 1 | n | n |
| TOTAL COST | | | 11n+2 |
| RUNTIME | | | O(n) |

Hash tables have the advantage of being able to search through data quickly. Hash tables also have the disadvantage of being slower at creating the initial list of courses. Also, hash tables can’t be sorted once created. To output a list that is sorted alphabetically the program would need to extract the data and then sort it. Every time a course is added or removed this lengthy process will have to be completed again. This makes the entire process much slower over time and the fast searching through the table would not be worth it. If the nature of the program was more static, I would recommend Hash Tables.

Binary Search Trees are fast at sorting data. The time required to sort through data is also predictable because the algorithm is determined by the height of the tree. BTS is optimal when sorting through large amounts of data. Perhaps far larger than a course catalog.

Vector is slower at searching the lists but faster at reading the file and compiling the list. As shown in my tables Vector is slightly over twice as fast as Binary Search Trees and over three times as fast as Hash Tables. This becomes more important as the size of the data that is being compiled grows larger. Using a quick sort method, Vector also allows sorting the lists alphabetically without any extra methods such as recompiling data, like Hash Tables. Additionally, the nature of a course catalog is that the courses provided are always changing. This means that over time a Vector method would be exponentially faster than a hash table or binary search tree. Thus, I recommend that the course catalog be developed with a Vector Data Structure.

It is probably outside the scope of this project, but I think it’s worth considering using both Vector and Hash Tables. I honestly don’t know if it’s even possible, but I’m imagining it as follows. First, the program uses Vector methods to read the file and sort the data alphabetically. Then the program uses Hash Tables methods to create a Hash Table from the alphabetically sorted data. This would enable the program to easily print an alphabetically sorted list of courses while also allowing a fast search function when users are searching for specific criteria. This would likely only be worth the extra time of compiling two data structures if dealing with a large amount of data. Our assignments throughout the course have shown that regardless of the data structure the time difference is very narrow unless very large quantities of data are being handled.