# Project One – Adrienne Sturgeon

## Pseudocode

//Loading Courses Pseudocode

void openFile(String csvLocation) {

Create an ifsteam object to open the file called fileInput(“filename.csv”)

If fileInput.is\_open() is not true

Output error message to user

Return 1

End if

}

void parseFile(String csvLocation) {

Create file variable with file location set to it.

Set vector object which will contain data content

Call openFile method

Call parseHeader method

Call parseContent method

}

Void parseContent inheriting from parser (void) {

While there are lines in the file use getLine(file, line) to search each line

If there are not two parameters per line

Throw error inadequate parameters

Return 1

For loop to parse over each line

Push back each row content

If value is missing (row size is not equal to header size)

Throw error message corrupted

Push back row contents

End While

}

Void parseHeader inheriting from parser (void) {

While file contains valid header entries

Push back header titles

End while

}

//Vector - Milestone 1

Method for creating course objects and store them in the appropriate data structure:

Main method will create vector list object courseList and Course object Course, and strings courseNum, courseName, and coursePrereq for course object contents.

Void courseLoader(string csvLocation, vector courseList) {

Call parseFile for csv file set to the string csvLocation

Try {

For loop iterating over rows of csv file

Initialize course object course

course.courseNum equals file row i content 1 (0 ignored as header)

course.courseName equals file row i content 2

course.coursePrereq equals file row remaining contents (0, 1 or more)

append course object to vector list courseList

} catch (csv error if file format is found incorrect or parameters missing via parseFile method throws)

Output error message to user

Search Method:

Void courseSearch (vector courseList, string courseNum) {

For loop iterating over all courses in courseList

If the course.courseNum object content is equal to the string courseNum

Print out course information

For loop iterating over prerequisites of course

If no prerequisites exist return

Else print prerequisite course information

End if

}

//Hash Table - Milestone 2

Main method will create Hash Table object courseList and Course object Course, and strings courseNum, courseName, and coursePrereq for course object contents.

Void courseLoader(string csvLocation, hash table courseList) {

Call parseFile for csv file set to the string csvLocation

Create vector named header to store headers from csv file

Try {

For loop iterating over rows of csv file

Initialize course object course

course.courseNum equals file row i content 1 (0 ignored as header)

course.courseName equals file row i content 2

course.coursePrereq equals file row remaining contents (0, 1 or more)

call hash table -> insert function to insert course object into hash table list

} catch (csv error if file format is found incorrect or parameters missing via parseFile method throws)

Output error message to user

}

Print Method:

Void printCourseSchedule(Hashtable<Course> courseList) {

For loop that iterates over all hash table buckets

Output course key coursed and courseName

If course contains prerequisite object fields

Output course prerequisites

}

Void outputcourseInfo(Hashtable<Course> courseList, String courseID) {

Create course object

Create course key

Create new node to contain course key values

While loop iterating over courseList until current bucket is equal to null

If current bucket key is equal to coursed (key)

Output information about course

End While

}

//Binary Search Tree – Milestone 3

Main method will create Binary Tree object courseList and Course object Course, and strings courseNum, courseName, and coursePrereq for course object contents.

Void courseLoader(string csvLocation, Tree courseList) {

Call parseFile for csv file set to the string csvLocation

Create vector named header to store headers from csv file

Try {

For loop iterating over rows of csv file

Initialize course object course

course.courseNum equals file row i content 1 (0 ignored as header)

course.courseName equals file row i content 2

course.coursePrereq equals file row remaining contents (0, 1 or more)

call Binary Tree -> insert function to insert course object into Binary Tree

} catch (csv error if file format is found incorrect or parameters missing via parseFile method throws)

Output error message to user

}

Print Method:

Void courseBinarySearchTree::inOrder(Node\* node) {

If the node is not a null pointer

Recursively traverse the tree with left preference until branch is completed

Output course key coursed and courseName

If course contains prerequisite object fields

Output course prerequisites

Traverse right to begin recursion down next branch (will end traversal if node is null)

}

Void BinarySearchTree::outputCourseInfo(string courseId){

Create course object

Set node pointer to

While loop iterating over Binary Tree until the current node is null

If current node bidId is equal to courseId

Output information about course

Else if the current node is larger than the bidId I am searching for

Traverse Tree left

Else

Traverse Tree right

End While

}

//Menu Pseudocode

Main {

While (choice is not exit) {

Display Menu with list of choices:

Choice 1 “Load Courses”

Choice 2 “Print Schedule”

Choice 3 “Print Course Information”

Choice 4 “Exit Program”

Switch Case for Choices {

Case 1: Call courseLoader method

Case 2: Call coursePrint/inOrder method

Case 3: Call courseSearch/outputCourseInfo method

Case 4: Break/exit program

}

}

}

//AlphaNumeric Order Pseudocode

//Vector

void alphaNumPrint(vector courseList) {

create list vector sortedCourseList

for loop iterating over courses list vector {

while loop iterating over courses list until current course is alphanumerically larger or until the end of the list is reached {

for loop iterating over list sortedCourseList until insert course is smaller than current course or end of list is reached {

insert course into sortedCourseList

}

}

}

Call print method to print vector sortedCourseList

}

//Hash Table

Void alphaNumPrint(Hashtable<Course> courseList) {

create list vector sortedCourseList

While loop iterating over hash table nodes until node is equal to null {

If current node is greater in alphanumeric order than node insert at front of vector sortedCourseList

Else insert at back of vector sortedCourseList

Set node to next node using lambda expression

}

Output vector sortedCourseList

}

//Binary Search Tree

The tree prints in alpha numeric order by default when using the inOrder method due to the nature of the tree structure.

Void courseBinarySearchTree::inOrder(Node\* node) {

If the node is not a null pointer

Recursively traverse the tree with left preference until branch is completed

Output course key coursed and courseName

If course contains prerequisite object fields

Output course prerequisites

Traverse right to begin recursion down next branch (will end traversal if node is null)

}

## Runtime Analysis

| **Vector** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **void alphaNumPrint(vector courseList) {** | 1 | 1 | 1 |
| **create list vector sortedCourseList** | 1 | 1 | 1 |
| **for loop iterating over courses list vector {** | 1 | n | n |
| **while loop iterating over courses list until current course is alphanumerically larger or until the end of the list is reached {** | 1 | n | n |
| **for loop iterating over list sortedCourseList until insert course is smaller than current course or end of list is reached {** | 1 | n | n |
| **insert course into sortedCourseList** | 1 | n | n |
| **}** | 1 | 1 | 1 |
| **}** | 1 | 1 | 1 |
| **}** | 1 | 1 | 1 |
| **Call print method to print vector sortedCourseList** | 6 | 1 | 6 |
| **Total Cost** | | | 3n + 10 |
| **Runtime** | | | O(n) |

| **Hash Table** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Void alphaNumPrint(Hashtable<Course> courseList) {** | 1 | 1 | 1 |
| **create list vector sortedCourseList** | 1 | 1 | 1 |
| **While loop iterating over hash table nodes until node is equal to null {** | 1 | n | n |
| **If current node is greater in alphanumeric order than node insert at front of vector sortedCourseList** | 1 | n | n |
| **Else insert at back of vector sortedCourseList** | 1 | n | n |
| **Set node to next node using lambda expression** | 1 | n | n |
| **}** | 1 | 1 | 1 |
| **Call print method to output vector sortedCourseList** | 6 | 1 | 6 |
| **}** |  |  |  |
| **Total Cost** | | | 4n + 9 |
| **Runtime** | | | O(n) |

| **Tree** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Void courseBinarySearchTree::inOrder(Node\* node) {** | 1 | 1 | 1 |
| **If the node is not a null pointer** | 1 | n | n |
| **Recursively traverse the tree with left preference until branch is completed** | 1 | Log(n) | Log(n) |
| **Output course key coursed and courseName** | 1 | n | n |
| **If course contains prerequisite object fields** | 1 | n | n |
| **If course has prerequisites output prerequisites** | 1 | n | n |
| **Output course prerequisites** | 1 | n | n |
| **Traverse right to begin recursion down next branch (will end traversal if node is null)** | 1 | Log(n) | Log(n) |
| **}** | 1 | n | n |
| **Total Cost** | | | 6n+ 2Log(n) + 1 |
| **Runtime** | | | O(log(n)) |

**Advantages/Disadvantages**

After completing a runtime analysis and experimenting with the three main data storage structures in this course, it is clear that consideration of the advantages and disadvantages of each must be completed to ensure an efficient application. Therefore, a simple explanation of each structure is provided: A Binary Search Tree is great for showing relationships that exist within data with an existing structure. Theoretically, the tree structure could benefit the nature of courses and their subsequent prerequisite courses, but adding, deleting, and searching for courses are also affected by this because the time it takes for them to complete grows logarithmically with the number of courses. So, if there are many courses in a course list, the time it takes for operations to complete could be much longer than the other data structure options available, especially if change operations are done often. Binary Search Trees also aren’t flexible with their organization, so the left-right node inheritance could be a problem.

Next, I study linked lists. The linked list vector has the simplest inherent structure of the available options with efficient operation complexity when completing addition or deletion of course objects in the list when compared to the Binary Search Tree. The main disadvantage, however, is that linked lists must be accessed in sequential order, which limits efficiency of operations to O(n) at best when that operation requires searching through the list. This leaves room for a more efficient structure, although if simplicity were the primary goal, this data structure would be the best choice.

Last, I review the Hash Table. A Hash tables main advantage is efficiency. Because of its key-value relationship between nodes and buckets, adding, deleting, and searching of courses has the potential to have an efficiency of O(1) if the key is used in correspondence with the desired coursed. This means that when using large lists of courses, the Hash Table would quickly become the most efficient method for data storage, and many schools have hundreds or more courses and prerequisites. The primary disadvantage of Hash Tables is their complexity. More advanced code is necessary to take advantage of a Hash Table successfully, and more design time must be spent when implementing this data structure. Hash Tables also have the potential to use their built-in key security via Hash Mapping, but that would not be necessary for the scope of this course.

With this information, I have chosen to use a Hash Table for my data structure when completing the course list storage system. I decided that the higher upfront design cost is worth the extreme efficiency benefit with the main operations that I will be using with the course list. The operating efficiency of O(1) is unbeatable when compared to my other options, and I believe I will be able to implement this data structure effectively. During the course loading process, I will assign keys that are based on the course ID in order to maximize efficiency.

**Resources**

Bisht, H. (2021, January 22). *Intro into 8 basic data structures*. DEV Community. https://dev.to/aws-builders/intro-into-8-basic-data-structures-3e3p