Project One

CSC 300

Jonathan Kleven

Table of Contents

[VECTOR 3](#_Toc110796797)

[Vector Runtime Analysis 5](#_Toc110796798)

[HASH TABLE 6](#_Toc110796799)

[Hash Table Runtime Analysis 8](#_Toc110796800)

[BINARY SEARCH TREE 9](#_Toc110796801)

[Binary Search Tree Runtime Analysis 11](#_Toc110796802)

[Menu 12](#_Toc110796803)

[Advantages and Disadvantages 13](#_Toc110796804)

[Vector 13](#_Toc110796805)

[Hash Table 13](#_Toc110796806)

[Binary Search Tree 13](#_Toc110796807)

[Recommendation 14](#_Toc110796808)

# VECTOR

**DEFINE Vector<string> Courses**

**DEFINE Vector<String> \_originalFile**

//structure to hold course information

**DEFINE Course Structure**

String courseId

String courseTitle

Vector<String> CoursePreReq

//Open and read all lines

OPEN filename

IF filename is OPEN

WHILE filnename IS good

Getline(filename, line)

IF line DOES NOT EQUAL “"

APPEND Vector \_originalFile with line

CLOSE filename

IF Vector \_originalFile size EQUAL ZERO

THROW Error

parseContent()

ELSE

DISPLAY "File failed to open"

CLOSE filename

//This function is similar to the CSVParser file in the previous modules

**DEFINE parseContent()**

FOR element (i) in beginning of \_originalFile TO end of \_originalFile

//If there are two parameters listed

IF element (i) AT column 0 DOES NOT EQUAL “" AND element (i) AT column 1 DOES NOT EQUAL “"

FOR item (j) in element

//While the data is not blank

WHILE element (i) of item (j) IS NOT “"

//I have seen examples of the use of sstream from internet and BST CSVparser.cpp

IF element (i) of item (j) IS string

CREATE new Course

IF item (j) EQUALS 0

Course CourseId EQUALS element (i) of item (j)

IF item (j) EQUALS 1

Course CourseTitle EQUALS element (i) of item (j)

IF item (j) IS GREATER THAN 1

IF SearchVector( element (i) of item (j)) EQUALS TRUE

APPEND Course Vector CoursePreReqs WITH element (i) of item (j)

APPEND Vector **Courses** WITH Course

ELSE

DISPLAY "Data is not formatted correctly"

ELSE

DISPLAY "Data is Corrupted"

**DEFINE SearchVector(Data)**

FOR element (i) in \_originalFile

IF element(i) AT [0] EQUALS DATA

RETURN TRUE

IF element(i) AT [0] DOES NOT EQUALS DATA

INCREMENT I

ELSE

RETURN FALSE

//Find course and relevant prerequisite

**DEFINE CourseInfo(course)**

FOR element (i) at courseID in **Courses**

IF element(i) AT [0] EQUALS DATA

DISPLAY element (i) at courseId

DISPLAY element (i) at courseTitle

IF element (i) at coursePreReqs EQUALS “”

CONTINUE

ELSE

DISPLAY element (i) at coursePreReqs

IF element(i) AT [0] DOES NOT EQUALS DATA

INCREMENT (i)

ELSE

DISPLAY "No search results for [course]"

//Display all courses in order

**DEFINE DisplayALL()**

SelectionSort(courses)

FOR entry in <Course Vector>

DISPLAY <Course Vector> courseId “ :” <Course Vector> courseTitle

IF entry CONTAINS coursePreReq

DISPLAY "Prerequisite course as follows:"

FOR entryPreReq in entry

DISPLAY newline“🡺🡺” entryPreReq

ELSE

DISPLAY "No Prerequisites"

//Sort Vector from smallest to Largest

**DEFINE SelectionSort(Courses)**

**DEFINE int min**

**DEFINE int i**

**DEFINE size EQUALS Courses size**

FOR element in courses size MINUS 1

Min EQUALS element

FOR element PLUS 1 in Courses

IF element PLUS 1 is LESS THAN element

Min EQUALS i

IF min DOES NOT EQUAL element

SWAP(courses at element with courses at min)

## Vector Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Initialize vector Courses | 1 | 1 | 1 |
| Initialize readLines | 1 | 1 | 1 |
| Open File | 1 | 1 | 1 |
| If file is Open | 1 | 1 | 1 |
| While file is Good | 1 | n | n |
| Get the line of data | 1 | n | n |
| If line of data Does Not Equal nothing | 1 | n | n |
| Add to Vector Courses | 1 | n | n |
| If Vector Course Equal nothing | 1 | n | n |
| Throw Error | 1 | n | n |
| ParseContent() | n | n | n |
| ELSE | 1 | n | n |
| Display "file Failed to Open | 1 | n | n |
| Close File | 1 | 1 | 1 |
|  |  | **Total Cost** | **9n+5** |
|  |  | **Runtime** | **O(n)** |
|  |  |  |  |
| Initialize ParseContent | 1 | 1 | 1 |
| For each element in Courses | 1 | 1 | 1 |
| If Element Column 1 and 2 are not nothing | 1 | n | n |
| For each item in element | 1 | 1 | 1 |
| While Element and item are not nothing | 1 | n | n |
| IF Element and Item are String | 1 | n | n |
| Create Course | 1 | n | n |
| IF Item Equals 0 | 1 | n | n |
| Course CourseId Equals Element and Item | 1 | n | n |
| IF Item Equals 1 | 1 | n | n |
| Course courseTitle Equals Element and Item | 1 | n | n |
| IF Item is Greater than 1 | 1 | n | n |
| IF SearchVector(element and item) Equals True | n | n | n^2 |
| Append coursePreReqs Equals Element and Item | 1 | n | n |
| ELSE Display "Data is not Formatted" | 1 | 1 | 1 |
| ELSE DISPLAY "Data is Corrupted" | 1 | 1 | 1 |
|  |  | **Total Cost** | **n^2+10n+3** |
|  |  | **Runtime** | **O(n^2)** |
|  |  |  |  |
| Initialize SearchVector | 1 | 1 | 1 |
| For element in Courses | 1 | 1 | 1 |
| IF element [0] Equals Data | 1 | n | n |
| Return True | 1 | n | n |
| IF element [0] Does Not Equals Data | 1 | n | n |
| Increment (i) | 1 | n | n |
| ELSE Return False | 1 | n | n |
|  |  | **Total Cost** | **5n+2** |
|  |  | **Runtime** | **O(n)** |
|  |  |  |  |
| Initialize SelectionSort() | 1 | 1 | 1 |
| Initialize min | 1 | 1 | 1 |
| Initialize i | 1 | 1 | 1 |
| Initialize size of Courses | 1 | 1 | 1 |
| For element in courses size minus 1 | 1 | n | n |
| min Equals element | 1 | n | n |
| For element Plus 1 in Courses | 1 | n | n |
| IF element Plus 1 Is Less than element | 1 | n | n |
| min Equals i | 1 | n | n |
| IF min Does Not Equal element | 1 | n | n |
| SWAP(couses at element with courses at min) | n | n | n^2 |
|  |  | **Total Cost** | **n^2+6n+4** |
|  |  | **Runtime** | **O(n^2)** |

# HASH TABLE

**DEFINE Structure of Course**

String courseId

String courseTitle

Vector<String> CoursePreReq

**DEFINE Structure of course Node**

DEFINE Course node

DEFINE key

DEFINE next pointer

**DEFINE Vector<String> \_originalFile**

**DEFINE Vector <Node> nodes**

**DEFINE HashTableSize**

**DEFINE Hash (key)**

//Open and read all lines

OPEN filename

IF filename is OPEN

WHILE filnename IS good

Getline(filename, line)

IF line DOES NOT EQUAL “"

APPEND Vector \_originalFile with line

CLOSE filename

IF Vector \_originalFile size EQUAL ZERO

THROW Error

parseContent()

ELSE

DISPLAY "File failed to open"

CLOSE filename

//This function is similar to the CSVParser file in the previous modules

**DEFINE parseContent()**

FOR element (i) in beginning of \_originalFile TO end of \_originalFile

//If there are two parameters listed

IF element (i) AT column 0 DOES NOT EQUAL “" AND element (i) AT column 1 DOES NOT EQUAL “"

FOR item (j) in element

//While the data is not blank

WHILE element (i) of item (j) IS NOT “"

//I have seen examples of the use of sstream from internet and BST CSVparser.cpp

IF element (i) of item (j) IS string

CREATE new Course

IF item (j) EQUALS 0

Course CourseId EQUALS element (i) of item (j)

IF item (j) EQUALS 1

Course CourseTitle EQUALS element (i) of item (j)

IF item (j) IS GREATER THAN 1

IF SearchVector( element (i) of item (j)) EQUALS TRUE

APPEND Course Vector CoursePreReqs WITH element (i) of item (j)

Insert(new Course)

ELSE

DISPLAY "Data is not formatted correctly"

ELSE

DISPLAY "Data is Corrupted"

**DEFINE SearchVector(Data)**

FOR element (i) in \_originalFile

IF element(i) AT [0] EQUALS DATA

RETURN TRUE

IF element(i) AT [0] DOES NOT EQUALS DATA

INCREMENT I

ELSE

RETURN FALSE

**DEFINE PrintAll()**

FOR node in nodes

IF node key NOT NULL

DISPLAY node courseId AND node courseTitle

If node course CoursePreReq DOES NOT EQUAL 0 OR NULL

FOR CoursePreReq in node

DISPLAY CoursePreReq

**DEFINE PrintCourse(courseId)**

CREATE the key for courseId

CREATE new node to node at key

IF node EQUALS NULL OR is int MAX

RETURN courseId

IF node DOES NOT EQUALS NULL OR is NOT int MAX AND node courseId EQUALS courseId

RETURN node courseID

WHILE node DOES NOT EQUAL NULL

IF node key is NOT int MAX AND node courseId EQUALS courseId

RETURN node courseID

Node EQUALS next node

**DEFINE hash value(courseId)**

RETURN key modulation of table size

**DEFINE insertNode(node)**

CREATE key FROM courseId

CREATE reference node

IF reference node EQUALS NULL

CREATE NEW node WITH course AND key

INSERT NEW node AT beginning PLUS key

ELSE

IF reference node key EQUALS UINT\_MAX //no nodes have been assigned

Reference node key EQUALS key

Reference node course EQUALS course

Reference node next EQUALS next

ELSE

While reference node next DOES NOT EQUAL NULL

Reference node EQUALS reference node next

Reference node next EQUALS NEW node Course AND key

## Hash Table Runtime Analysis

| **Code** | **Line Cost** | **# Times Executed** | **Total Cost** |
| --- | --- | --- | --- |
| Initialize vector Courses | 1 | 1 | 1 |
| Initialize readLines | 1 | 1 | 1 |
| Open File | 1 | 1 | 1 |
| If file is Open | 1 | 1 | 1 |
| While file is Good | 1 | n | n |
| Get the line of data | 1 | n | n |
| If line of data Does Not Equal nothing | 1 | n | n |
| Add to Vector Courses | 1 | n | n |
| If Vector Course Equal nothing | 1 | n | n |
| Throw Error | 1 | n | n |
| ParseContent() | n | n | n |
| ELSE | 1 | n | n |
| Display "file Failed to Open | 1 | n | n |
| Close File | 1 | 1 | 1 |
|  |  | **Total Cost** | **9n+5** |
|  |  | **Runtime** | **O(n)** |
|  |  |  |  |
| Initialize ParseContent | 1 | 1 | 1 |
| For each element in Courses | 1 | 1 | 1 |
| If Element Column 1 and 2 are not nothing | 1 | n | n |
| For each item in element | 1 | 1 | 1 |
| While Element and item are not nothing | 1 | n | n |
| IF Element and Item are String | 1 | n | n |
| Create Course | 1 | n | n |
| IF Item Equals 0 | 1 | n | n |
| Course CourseId Equals Element and Item | 1 | n | n |
| IF Item Equals 1 | 1 | n | n |
| Course courseTitle Equals Element and Item | 1 | n | n |
| IF Item is Greater than 1 | 1 | n | n |
| IF SearchVector(element and item) Equals True | n | n | n^2 |
| Append coursePreReqs Equals Element and Item | 1 | n | n |
| ELSE Display "Data is not Formatted" | 1 | 1 | 1 |
| ELSE DISPLAY "Data is Corrupted" | 1 | 1 | 1 |
|  |  | **Total Cost** | **n^2+10n+3** |
|  |  | **Runtime** | **O(n^2)** |
|  |  |  |  |
| Initialize insertNode() | 1 | 1 | 1 |
| CREATE key FROM courseId | 1 | 1 | 1 |
| CREATE reference node | 1 | 1 | 1 |
| IF reference node EQUALS NULL | 1 | 1 | 1 |
| CREATE NEW node WITH course AND key | 1 | 1 | 1 |
| INSERT NEW node AT beginning PLUS key | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| IF reference node key EQUALS UINT\_MAX //no nodes have been assigned | 1 | 1 | 1 |
| Reference node key EQUALS key | 1 | 1 | 1 |
| Reference node course EQUALS course | 1 | 1 | 1 |
| Reference node next EQUALS next | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| While reference node next DOES NOT EQUAL NULL | 1 | n | n |
| Reference node EQUALS reference node next | 1 | 1 | 1 |
| Reference node next EQUALS NEW node Course AND key | 1 | 1 | 1 |
|  |  | **Total Cost** | **n+14** |
|  |  | **Runtime** | **O(n)** |

# BINARY SEARCH TREE

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include <sstream>

**DEFINE Vector<String> \_originalFile**

**DEFINE root**

**DEFINE new Node**

//structure to hold course information

**DEFINE Course Structure**

String courseId

String courseTitle

Vector<String> CoursePreReq

//Structure of a tree node

**DEFINE Node Structure**

Course course

Node point left

Node point right

Node()

Left EQUALS NULL

Right EQUALS NULL

Node(course) OF Node()

course EQUALS course

//Open and read all lines

OPEN filename

IF filename is OPEN

WHILE filnename IS good

Getline(filename, line)

IF line DOES NOT EQUAL “"

APPEND Vector \_originalFile with line

CLOSE filename

IF Vector \_originalFile size EQUAL ZERO

THROW Error

parseContent()

ELSE

DISPLAY "File failed to open"

CLOSE filename

//This function is similar to the CSVParser file in the previous modules

**DEFINE parseContent()**

FOR element (i) in beginning of \_originalFile TO end of \_originalFile

//If there are two parameters listed

IF element (i) AT column 0 DOES NOT EQUAL “" AND element (i) AT column 1 DOES NOT EQUAL “"

FOR item (j) in element

//While the data is not blank

WHILE element (i) of item (j) IS NOT “"

//I have seen examples of the use of sstream from internet and BST CSVparser.cpp

IF element (i) of item (j) IS string

CREATE new Course

IF item (j) EQUALS 0

Course CourseId EQUALS element (i) of item (j)

IF item (j) EQUALS 1

Course CourseTitle EQUALS element (i) of item (j)

IF item (j) IS GREATER THAN 1

IF SearchVector( element (i) of item (j)) EQUALS TRUE

APPEND Course Vector CoursePreReqs WITH element (i) of item (j)

Insert(new Course)

ELSE

DISPLAY "Data is not formatted correctly"

ELSE

DISPLAY "Data is Corrupted"

**DEFINE SearchVector(Data)**

FOR element (i) in \_originalFile

IF element(i) AT [0] EQUALS DATA

RETURN TRUE

IF element(i) AT [0] DOES NOT EQUALS DATA

INCREMENT I

ELSE

RETURN FALSE

**DEFINE Insert(course)**

IF root EQUALS NULL

Root = new Node(course)

ELSE

InsertNode(root, course)

**DEFINE InsertNode(node, course)**

IF root EQUALS NULL

root EQUALS new node value

IF root value GREATER THAN value

IF root left node EQUALS NULL

root left EQUALS new node value

RETURN

insertNode(value, root left node)

IF root value LESS THAN value

IF root right node EQUALS NULL

root right EQUALS new Node(value)

RETURN

insertNode(value, root right node)

**DEFINE DisplayCourseInfo(course)**

DEFINE currentNode EQUALS root

WHILE currentNode NOT EQUAL TO NULL

IF currentNode Course is FOUND

DISPLAY currentNode courseId, currentNode courseTitle

DISPLAY “🡺” currentNode CoursePreReq

IF course LESS THAN currentNode course

currentNode EQUALS left currentNode

ELSE

currentNode EQUALS right currentNode

DEFINE Course course

RETURN course

## Binary Search Tree Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executed | Total Cost |
| Initialize vector Courses | 1 | 1 | 1 |
| Initialize readLines | 1 | 1 | 1 |
| Open File | 1 | 1 | 1 |
| If file is Open | 1 | 1 | 1 |
| While file is Good | 1 | n | n |
| Get the line of data | 1 | n | n |
| If line of data Does Not Equal nothing | 1 | n | n |
| Add to Vector Courses | 1 | n | n |
| If Vector Course Equal nothing | 1 | n | n |
| Throw Error | 1 | n | n |
| ParseContent() | n | n | n |
| ELSE | 1 | n | n |
| Display "file Failed to Open | 1 | n | n |
| Close File | 1 | 1 | 1 |
|  |  | **Total Cost** | **9n+5** |
|  |  | **Runtime** | **O(n)** |
|  |  |  |  |
| Initialize ParseContent | 1 | 1 | 1 |
| For each element in Courses | 1 | 1 | 1 |
| If Element Column 1 and 2 are not nothing | 1 | n | n |
| For each item in element | 1 | 1 | 1 |
| While Element and item are not nothing | 1 | n | n |
| IF Element and Item are String | 1 | n | n |
| Create Course | 1 | n | n |
| IF Item Equals 0 | 1 | n | n |
| Course CourseId Equals Element and Item | 1 | n | n |
| IF Item Equals 1 | 1 | n | n |
| Course courseTitle Equals Element and Item | 1 | n | n |
| IF Item is Greater than 1 | 1 | n | n |
| IF SearchVector(element and item) Equals True | n | n | n^2 |
| Append coursePreReqs Equals Element and Item | 1 | n | n |
| ELSE Display "Data is not Formatted" | 1 | 1 | 1 |
| ELSE DISPLAY "Data is Corrupted" | 1 | 1 | 1 |
|  |  | **Total Cost** | **n^2+10n+3** |
|  |  | **Runtime** | **O(n^2)** |
|  |  |  |  |
| Initialize Insert(Course) | 1 | 1 | 1 |
| IF root Equals Null | 1 | 1 | 1 |
| root = new node (course) | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| InsertNode(root, Course) | 1 | n | n |
|  |  | **Total Cost** | **n+4** |
|  |  | **Runtime** | **O(n)** |
|  |  |  |  |
| Initialize insertNode(node, course) | 1 | 1 | 1 |
| IF root EQUALS NULL | 1 | 1 | 1 |
| root EQUALS new node value | 1 | 1 | 1 |
| IF root value GREATER THAN value | 1 | 1 | 1 |
| IF root left node EQUALS NULL | 1 | 1 | 1 |
| root left EQUALS new node value | 1 | 1 | 1 |
| RETURN | 1 | 1 | 1 |
| insertNode(value, root left node) | 1 | n | n |
| IF root value LESS THAN value | 1 | 1 | 1 |
| IF root right node EQUALS NULL | 1 | 1 | 1 |
| root right EQUALS new Node(value) | 1 | 1 | 1 |
| RETURN | 1 | 1 | 1 |
| insertNode(value, root right node) | 1 | n | n |
|  |  | **Total Cost** | **2n+11** |
|  |  | **RunTime** | **O(n)** |

# Menu

DEFINE choice EQUAL 0

WHILE choice DOES NOT EQUAL 9

DISPLAY "MENU:"

DISPLAY "1. Load Course Data"

DISPLAY "2. Print All Courses (Sorted)"

DISPLAY "3. Print Individual Course"

DISPLAY "9. Exit"

DISPLAY "Enter a Number: "

GET user INPUT

Switch (choice)

Case 1:

Load course data from file to data structure

Break

Case 2:

FOR element in courses

DISPLAY courseId, courseTitle

IF course has coursePreReqs

DISPLAY coursePreReqs

Break

Case 3:

DISPLAY "Enter a Course number: "

courseSearch EQUALS GET user INPUT

IF courseSearch IS IN Course data structure

DISPLAY courseId, courseTitle

IF course has coursePreReqs

DISPLAY coursePreReqs

ELSE

DISPLAY "Course not found"

Break

//If 9 is entered the while loop exits

DISPLAY "Good bye."

Return 0

# Advantages and Disadvantages

## Vector

Vectors are simple data structures that allow a program to add data to. They are, in a simplistic sense, a list of data. Within this list are easy methods to search for information and add additional data to the beginning or the end with very little programming. However, parsing or searching for specific data can have a long runtime as the data in the vector increases. Additionally, anytime more data is added, the list will need to be sorted again to get the most benefits of the vector. Similarly, inserting and deleting items within a sorted vector can be inefficient. Furthermore, the time to search, sort, insert and delete depend on the vector's size as each item in the vector will need to be visited. The worst case scenario is that the all items will be visited in the vector before an action can take place.

## Hash Table

A hash table is a data structure that can store unordered data with a key; in other words, a hash table can map its data to a location. It ensures that data within the hash table is associated with a specific and unique key or hash. If data security is a priority, the key can help identify data without providing access to the data directly. Inserting and deleting data from a hash table is more straightforward than a vector as there is only a need to reassign the data pointers before the data to the next data in the table in the case of deletion or reassigning the data pointers of prior data to the new data and to the following data in the table to the already existing data pointers. However, depending on the initial size of the hash table, there is the possibility of data collisions with data that may already exist in the same hash or bucket. Collisions can be avoided by adding more buckets, though collisions are often inevitable and can be planned for within the program. Searching for items in a hash table is merely comparing the hash or key of the search value with the hash or key of the hash table. Any collisions that are mapped to that hash value would require additional comparisons to that bucket, slowing the runtime if there is a lot of data in that bucket, but not nearly as much since most of the hash table is ignored in the search.

## Binary Search Tree

A binary search tree is a data structure that creates a data hierarchy. Each node in a binary search tree has one root node and up to 2 child nodes. It provides a means to add nodes with a small comparison of less than or greater than the root node. This enables fast deletion, insertion, searching, and display of data in order, despite the size and tree shape. It should be noted that a balanced binary search tree improves the runtime of searching, insertion, and deletion of data. Additionally, creating a binary search tree with sorted data will likely incur a stack overflow error as sorted data will create an unbalanced tree, a tree with only left or right child nodes.

# Recommendation

Based on the results of the runtime analysis and practice with the data structures, I recommend using binary search trees as the data structure for storing, organizing, and searching the course information. For me, it was one of the most straightforward implementations of data structures with the additional benefits of speed and inherent order. More importantly, courses are already structured hierarchically, and mirroring the data structure to the data makes the most sense from an object-orientated perspective.