CS-300 6-2 Project One: Evaluation

Cade Bray

Cade.Bray@snhu.edu

Southern New Hampshire University

Dr. James O. Webb Jr.

JUNE 16th, 2024

# Table of Contents

*(Control Click the items below to navigate this document)*

[Table of Contents](#_Toc2015700539)

[About this document](#_Toc540832330)

[Pseudocode for Menu](#_Toc1209594951)

[Pseudocode for sorting data structure alphanumerically and printing](#_Toc293998146)

[Course Input Example](#_Toc723661896)

[Binary Tree Structure: Data Import Method](#_Toc1696198892)

[Binary Search Tree Class](#_Toc1618863072)

[Course Class Object](#_Toc1273052197)

[Hash Tables – Data Import Method](#_Toc1890541188)

[Hash Tables Class](#_Toc51798571)

[Vector Sorting](#_Toc1156857625)

[Advantages, Disadvantages, and Recommendation](#_Toc1077541205)

[References](#_Toc352316254)

*It may prove beneficial to navigate this document with the sections collapsed.*

# About this document

This document takes close notice of the requirements set out by ABCU and will implement various concepts to better analyze our project. These concepts primarily include pseudocode, step-count method for runtime analysis, and finally an evaluation of the various data structures. To help increase the readability of this document I will focus primarily on components of the pseudocode that contribute to the growth rate of the function. This means excluding lines that have a constant total cost. In addition to reducing the overall amount of pseudocode displayed I will be reducing the runtime analysis to exclude lines of a total cost that is a constant. It should be noted that all runtime complexities listed in this document are **worst-case** time complexities unless otherwise stated.

In this project you will find that many of the public functions are ‘wraps’ of private functions and only serve the purpose of preserving the integrity of the data structure. These functions evaluate the same big O notation as their function call, they are wrapping. For this purpose, they will not be included in this evaluation. Functions that evaluate to a constant number of operations such as a ‘setter’ or ‘getter’ function will not be displayed in this document either because we are focusing on the overall growth rate comparisons of the data structures.

This document will be inclusive of two additional requests for producing pseudocode for the menu and universal printing function which will enable use depending on which data structure it is given. Finally, I will compare the primary advantages and disadvantages of each data structure before providing my final recommendation of the data structure that best fits the requirements of this project.

# Pseudocode for Menu

Pseudo Code for the menu method:

void **displayMenu**() {

CREATE variable called choice set it equal to 0;

WHILE choice doesn’t equal 9

OUTPUT "Menu:"

// Load menu options

OUTPUT "1. Load Bids /n2. Display All Bids /n3. Find Bid /n 4. Remove Bid /n9. Exit"

OUTPUT "Enter choice: "

INPUT a number and save it as choice

SWITCH with choice variable

// Switch case allows for multiple outcomes on one variables critiera

CASE choice is equal to 1

// Initialize a timer variable before loading bids

CREATE variable called ticks and set ti equal to the return of Clock()

// Complete the method call to load the courses

CREATE a variable of type either BinarySearchTree, HashTable, or Vector called chosenDataStructure set equal to dataImport()

// Calculate elapsed time and display result

SET variable called ticks and set equal to clock() - ticks

// current clock ticks minus starting clock ticks

OUTPUT "time: " + ticks

OUTPUT "time: " + (ticks \* 1.0 / CLOCKS\_PER\_SEC) + "seconds"

CASE choice is equal to 2:

CALL chosenDataStructure’s alphanumeric ordered print method

CASE choice is equal to 3:

//Find and print single course object contents

CREATE variable called ticks equal to clock()

CREATE course equal to chosenDataStructure’s search method return value

SET ticks equal to (clock() - ticks)

// current clock ticks minus starting clock ticks

IF course is not an empty course object

OUTPUT course object contents

ELSE

OUTPUT "Couse " + CourseCode + " not found."

OUTPUT "time: " + ticks + " clock ticks"

OUTPUT "time: " + (ticks \* 1.0 / CLOCKS\_PER\_SEC) + " seconds"

# Pseudocode for sorting data structure alphanumerically and printing

void **print****AlphaNumeric**(auto data\_structure) {

IF data\_structure is of type BinarySearchTree

// The public wrap of InOrder() doesn’t require a starting point. It will start at root // and print items in ascending order or alphanumeric.

CALL InOrder()in

ELSE IF data\_structure is of type HashTable

// Hash tables don’t have a built-in method to sort the selection so we’re going to // iterate through the table building a sorted vector.

CREATE variable called sorted\_vector

FOR each node in data\_structure

IF sorted\_vector size is equal to zero

APPEND node to sorted\_vector

ELSE

FOR each n in sorted\_vector

// This IF/ELSE will sort to alphanumeric or ascending // order use references page to find more on strcmp()

IF the return of strcmp() passed n and node is >= 0

APPEND to sorted\_vector

ELSE

PREPPEND to sorted\_vector

FOR each node in sorted\_vector

OUTPUT the course information

ELSE IF data\_structure is of type vector

CREATE a vector called temp\_vector equal to data\_structure

// Use references page to learn more on sort(). It is called in place and sorts the // vector.

CALL sort() and pass it the beginning of temp\_vector, the end of temp\_vector combined with the size of temp\_vector

FOR each course in temp\_vector

OUTPUT the information of course

ELSE

THROW TypeError “Improper data structure provided. Please provide a Binary Search Tree, Hash table, or Vector.”

}

# Course Input Example

*Example of the course data found within a given file:*

CSCI100, Introduction to Computer Science

CSCI101, Introduction to Programming in C++, CSCI100

CSCI200, Data Structures, CSCI101

MATH201, Discrete Mathematics

CSCI300, Introduction to Algorithms, CSCI200, MATH201

CSCI301, Advanced Programming in C++, CSCI101

CSCI350, Operating Systems, CSCI300

CSCI400, Large Software Development, CSCI301, CSCI350

# Binary Tree Structure: Data Import Method

The data import method of this project is amongst the most difficult to implement and is specifically designed for the binary tree structure as per the return listed in the pseudocode. As for all the data import methods found within this document, I will be utilizing the course object. This method will also be utilizing class methods found within the Binary Search Tree class. In the runtime analysis below, we can see that there are many constants throughout this pseudocode. Some constants are under a for loop which makes their total cost equal to n, but some constants found in a nest for loop which produces their runtime complexity to equal n^2. The overall worst-case runtime for the dataImport method is O(n^2) which is a quadratic growth rate.

*Please note that all “//” characters groups are in reference to a comment that should not be perceived as part of the sequence of events.*

**Runtime Analysis of the dataImport method below:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| (Constant Operations evaluating to ‘1’ for each line. Reduced for readability) | 6 | 1 | 6 |
| Opening a file with ifStream and creating a string from the contents | n | 1 | n |
| FOR each character in raw\_file we will call char | 1 | n | n |
| (Various constant operations that evaluate the total cost of ‘n’. Reduced for readability.) | 9 | n | 9n |
| CREATE a course object called temp\_course and set it equal to the return of the method call Search a function of the variable course\_bt. | n | n | n^2 |
| FOR each course code in pre\_req called pre\_course\_code  (nested for loop) | n | n | n^2 |
| IF the pre\_course\_code is not equal to the return of method call Search a function of the variable course\_bt | n | n^2 | n^2 \* n |
| THROW exception with given message | 1 | n^2 | n^2 |
| CALL method Insert a function of the variable course\_bt with the parameter course. | n | n | n^2 |
| (Various constant operations that evaluate the total cost of ‘n’. Reduced for readability.) | 23 | n | 23n |
| RETURN course\_bt | 1 | 1 | 1 |
|  |  | **Total Cost** | 5n^2 + 35n + 7 |
|  |  | **Runtime** | O(n^2) |

***Pseudo Code for the data import method:***

BinarySearchTree **dataImport**(string file\_path) {

OPEN file at file\_path and save data as a string called raw\_file

CREATE an empty string called temp\_string

CREATE an empty vector of strings called pre\_req

CREATE an empty string called course\_descr

CREATE an empty string called course\_code

CREATE an integer variable called comma\_count

CREATE an empty BinaryTree variable called course\_bt

*// For more information on the BinaryTree object please refer to TOC*

FOR each character in raw\_file we will call char

*// For more information on the course object please refer to TOC*

CREATE an empty course object called course

IF char is equal to a new line character “\n”

*// This implies that a new line has occurred, and our collected data*  *// needs to be constructed into a course object and further sent to the*  *// Binary tree for construction.*

CALL method setCourseCode a function of the variable course with the parameter course\_code

CALL method setCourseDescription a function of the variable course with the parameter course\_descr

IF the size of pre\_req is greater than zero

/*/ Refer to the references page for more context on how the size function*  *// works for vectors.*

CALL method setCoursePreReqs a function of the variable course

with the parameter pre\_req

*// This is our data validation step.*

CREATE a course object called temp\_course and set it equal to the return of the method call Search a function of the variable course\_bt.

FOR each course code in pre\_req called pre\_course\_code

*// We are searching for each prerequisite in the vector to ensure it's*  *// already in the binary tree.*

IF the pre\_course\_code is not equal to the return of method call Search a function of the variable course\_bt

*// The Search function will search the binary tree and // return the course object unless it does not exist. If it // doesn’t exist it will return an empty course object.*

THROW exception with the message “Course prerequisite found that cannot be validated.”

*// All data has been validated per the requirements page and is ready to // insert into binary tree structure.*

CALL method Insert a function of the variable course\_bt with the parameter course.

SET comma\_count equal to 0

SET temp\_string equal to “” (Empty String)

SET course equal to an empty Course object

SET pre\_req equal to an empty vector of strings

SET course\_code equal to “” (Empty String)

SET course\_descr equal to “” (Empty String)

IF ELSE char is equal to a comma AND comma\_count is equal to zero

*//We have found the end of our course\_code because it’s the start of the line and // before the first comma.*

SET course\_code equal to temp\_string

INCREMENT comma\_count by one

SET temp\_string equal to “” (Empty String)

IF ELSE char is equal to a comma AND comma\_count is equal to 1

*//We have found the end of our course description because it’s the start of the line // first comma and before the second comma.*

SET course\_descr equal to temp\_string

SET temp\_string equal to “” (Empty String)

INCREMENT comma\_count by one

IF ELSE char is equal to a comma AND comma\_count is greater or equal to 1

*//We have found a prerequisite*

APPEND temp\_string as an item to vector pre\_req

SET temp\_string equal to “” (Empty String)

INCREMENT comma\_count by one

IF ELSE char is equal to “\0”

*// The character “\0” is the denotation of null or an empty item. Meaning we’re // at the end of the document.*

RETURN course\_bt

# Binary Search Tree Class

Binary search trees are efficient in their search operations. The data import method for this data structure and the de-constructor are the highest growth rate functions we will encounter for the implementation of this project at O(n^2). While all other functions of this class are either constant O(1 to 3) or linear O(n) growth functions.

**Runtime Analysis of ‘addNode’:**

Note that the equation is recursive and in the worst case will need to traverse each node individually. This implies that in the worst case the growth rate is linear or O(n). The line cost for these calls reflects the cost of the function call of n. This lands at the final total cost of

*T(n) = n + n + {1 + 1, 2 + 1, 3 + 1, ..., 8} = 2n + 8*

Dropping the constant term leaves us with *2n*. Since the constant 2 in 2n becomes less noticeable as the growth rate continues, we can drop this term as well leaving us with n or O(n). If the tree were balanced, we could see a runtime complexity of O(log n).

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| (Constant Operations evaluating to ‘1’ for each line. Reduced for readability) | 4 | 1 | 4 |
| CALL addNode with node’s left pointer as the node and course as the Course variable | n | 1 | n |
| ELSE | 1 | 1 | 1 |
| IF there is no right pointer to this node | 1 | 1 | 1 |
| SET node’s right pointer to this node | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| CALL addNode with the node’s right pointer as the node and course as the course variable | n | 1 | n |
|  |  | **Total Cost** | 2n + 8 |
|  |  | **Runtime** | O(n) |

**Pseudo Code for Binary Tree Class – Private functions:**

void BinarySearchTree::**addNode**( Node\* node, Course course) { ***// RT is O(n)***

IF node → course’s course\_code property is equal to course’s course\_code

*// We are adding/Traversing the left side*

IF there is no left pointer to this node

*// No left node let's put it there!*

SET node’s left pointer equal to course

ELSE

*// Traverse the left side until an open spot is found (Recursively)*

CALL addNode with node’s left pointer as the node and course as the Course variable

ELSE

// Traverse/Add to the right side

IF there is no right pointer to this node

*// No right node let's put it there!*

SET node’s right pointer to this node

ELSE

*//Traverse the right side until an open spot is found (Recursively)*

CALL addNode with the node’s right pointer as the node and course as the course variable

}

**InOrder, postOrder, preOrder Traversal:**

InOrder traversal is when you start in the left subtree, traverse to the root of the tree, and end at the right subtree. The postOrder traversal is when you start at the left subtree, traverse to the right subtree, and finish at the root of the tree. Finally, the preOrder traversal is when you start at the root of the tree, traverse to the left subtree, and finish at the right subtree. This is ultimately dictated by the order in which the recursive calls are made. Knowing this I will show an example of the postOrder method with an output call, but the complexity will remain the same for each form of traversal.

void BinarySearchTree:: **postOrder** (Node\* node) { ***// Runtime is O(n)***

IF node is not equal to a null pointer

CALL postOrder with node’s left pointer as the passed node

CALL postOrder with node’s right pointer as the passed node

OUTPUT node’s course code, course description, and prerequisites

}

**Runtime Analysis of postOrder method (Same complexity for inOrder and preOrder):**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| If node is not equal to a null pointer | 1 | 1 | 1 |
| CALL postOrder with node’s left pointer as the passed node | n | 1 | n |
| CALL postOrder with node’s right pointer as the passed node | n | 1 | n |
| OUTPUT node’s course code, course description, and prerequisites | 1 | 1 | 1 |
|  |  | **Total Cost** | 2n + 2 |
|  |  | **Runtime** | O(n) |

Node\* BinarySearchTree:: **removeNode**(Node\* node, string course\_code) { ***// Runtime is O(n)***

IF node is a null pointer

RETURN node

ELSE IF course\_code is less than node → course\_code property

*// This implies we need to traverse recursively down the left side*

SET node’s left pointer equal to the result of recursive call removeNode passed the node’s left pointer as node and course\_code as course\_code

ELSE IF course\_code is greater than node → course\_code property

*// This implies we need to traverse recursively down the right side*

SET node’s right pointer equal to the result of recursive call removeNode passed the node’s right pointer as node and course\_code as course\_code

ELSE IF node’s left pointer AND node’s right pointer are both null pointers

*// This implies we have nowhere left to traverse so delete node*

DELETE node

SET node equal to a null pointer

ELSE IF Node’s left pointer is the only pointer present

CREATE a Node\* called temp and set it equal to node

SET node equal to node’s left pointer

DELETE temp

ELSE IF Node’s right pointer is the only pointer present

CREATE a Node\* called temp and set it equal to node

SET node equal to node’s right pointer

DELETE temp

ELSE

*// Implies more than one child, find the min*

CREATE a Node\* called temp and set it equal to node’s right pointer

WHILE temp’s left pointer does not equal a null pointer

SET temp equal to temp’s left pointer

SET node → course equal to temp → course

SET node’s right pointer equal to the return of removeNode’s return pass node’s right pointer as node and temp → course\_code property as course

RETURN node

}

**Runtime Analysis of BinarySearchTree removeNode method:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| (Constant Operations evaluating to ‘1’ for each line. Reduced for readability) | 3 | 1 | 3 |
| SET node’s left pointer equal to the result of recursive call removeNode passed the node’s left pointer as node and course\_code as course\_code | n | 1 | n |
| ELSE IF course\_code is greater than node → course\_code property | 1 | 1 | 1 |
| SET node’s right pointer equal to the result of recursive call removeNode passed the node’s right pointer as node and course\_code as course\_code | n | 1 | n |
| (Constant Operations evaluating to ‘1’ for each line. Reduced for readability) | 12 | 1 | 12 |
| WHILE temp’s left pointer does not equal a null pointer | 1 | n | n |
| SET temp equal to temp’s left pointer | 1 | n | n |
| SET node → course equal to temp → course | 1 | 1 | 1 |
| SET node’s right pointer equal to the return of removeNode’s return pass node’s right pointer as node and temp → course\_code property as course | n | 1 | n |
| RETURN node | 1 | 1 | 1 |
|  |  | **Total Cost** | 5n + 18 |
|  |  | **Runtime** | O(n) |

**Pseudo Code for Binary Tree Class – Public functions:**

Functions excluded from this document:

|  |  |
| --- | --- |
| **Function** | **Runtime evaluation** |
| BinarySearchTree() constructor | Runtime is O(1) |
| InOrder(), PreOrder(), and PostOrder() (Wrapped methods) | Runtime is O(n) |
| Remove() (wrapped method) | Runtime is O(n) |

**Runtime Analysis of BinarySearchTree de-constructor:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| WHILE root does not equal null pointer | 1 | n | n |
| CALL Remove with pass root → course\_code property | n | n | n^2 |
|  |  | **Total Cost** | n^2 + n |
|  |  | **Runtime** | O(n^2) |

BinarySearchTree:: **~BinarySearchTree**() { ***// Runtime is O(n)***

*// De-constructor*

WHILE root does not equal null pointer

CALL Remove with pass root → course\_code property

}

**Runtime complexity for Insert method:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| IF root is equal to a null pointer | 1 | 1 | 1 |
| SET root equal to a new Node constructed from course | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| CALL addNode and pass it root as the node and course and the Course | n | 1 | n |
|  |  | **Total Cost** | n+3 |
|  |  | **Runtime** | O(n) |

void BinarySearchTree:: **Insert** (Course course) { **//Runtime is O(n)**

IF root is equal to a null pointer

SET root equal to a new Node constructed from course

ELSE

CALL addNode and pass it root as the node and course and the Course

}

**Runtime Analysis of Search method:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| CREATE a Node\* variable called curr\_node and set it equal to root | 1 | 1 | 1 |
| WHILE curr\_node doesn’t equal a null pointer | 1 | n | n |
| IF curr\_node → course\_code property is equal to course\_code | 1 | 1 | 1 |
| return curr\_node → course | 1 | 1 | 1 |
| ELSE IF curr\_node → course\_code is greater than course\_code | 1 | 1 | 1 |
| SET curr\_node equal to curr\_node’s left node | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| SET curr\_node equal to curr\_nodes’ right node | 1 | 1 | 1 |
| CREATE an empty course object called no\_item\_found | 1 | 1 | 1 |
| RETURN no\_item\_found | 1 | 1 | 1 |
|  |  | **Total Cost** | n+9 |
|  |  | **Runtime** | O(n) |

Course BinarySearchTree:: **Search** (string course\_code) { ***// Runtime is O(n)***

CREATE a Node\* variable called curr\_node and set it equal to root

WHILE curr\_node doesn’t equal a null pointer

IF curr\_node → course\_code property is equal to course\_code

return curr\_node → course

ELSE IF curr\_node → course\_code is greater than course\_code

SET curr\_node equal to curr\_node’s left node

ELSE

SET curr\_node equal to curr\_nodes’ right node

CREATE an empty course object called no\_item\_found

RETURN no\_item\_found

}

# Course Class Object

The course class object has the primary function of storing all parsed information in a uniform way. Once parsed in the data import method of various data structures we can see the methods also utilize construction of course objects. The structure has three private global variables with each having a public getter/setter method. The course code and description are both saved as strings while the prerequisites are saved in a vector of strings. This class has a constructor that initializes the three global variables as empty strings. Along with the constructor, I have made a deconstructor that releases those variables from memory. In all regards the course class has a constant runtime. Because of this constant runtime of all aspects, I will instead include a table of operations and their runtime in big O notation without the pseudocode or step-count method breakdown.

**Course Class Runtimes:**

|  |  |
| --- | --- |
| **Function** | **Runtime in Big O notation** |
| course() constructor | Runtime is O(3) |
| ~course() de-constructor | Runtime is O(3) |
| SetCourseCode() | Runtime is O(2) |
| GetCourseCode() | Runtime is O(1) |
| SetCourseDescription() | Runtime is O(2) |
| GetCourseDescription() | Runtime is O(1) |
| SetCoursePreReqs() | Runtime is O(2) |
| GetCoursePreReqs() | Runtime is O(1) |

# Hash Tables – Data Import Method

**Runtime Analysis of the dataImport method below:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| (Constant Operations evaluating to ‘1’ for each line. Reduced for readability) | 6 | 1 | 6 |
| Opening a file with ifStream and creating a string from the contents | n | 1 | n |
| FOR each character in raw\_file we will call char | 1 | n | n |
| (Constant Operations evaluating to ‘n’ for each line. Reduced for readability) | 2 | n | 2n |
| CALL method setCourseCode a function of the variable course with the parameter course\_code | 2 | n | 2n |
| CALL method setCourseDescription a function of the variable course with the parameter course\_descr | 2 | n | 2n |
| IF the size of pre\_req is greater than zero | 1 | n | n |
| CALL method setCoursePreReqs a function of the variable course with the parameter pre\_req | 2 | n | 2n |
| CREATE a course object called temp\_course and set it equal to the return of the method call Search a function of the variable course\_ht. | n | n | n^2 |
| FOR each course code in pre\_req called pre\_course\_code  (nested for loop) | n | n | n^2 |
| IF the pre\_course\_code is not equal to the return of method call Search a function of the variable course\_ht | n | n^2 | 2n^2 \* n |
| THROW exception with given message | 1 | n^2 | n^2 |
| CALL method Insert a function of the variable course\_ht with the parameter course. | n | n^2 | n^2\*n |
| (Constant Operations evaluating to ‘n’ for each line. Reduced for readability) | 23 | n | 23n |
| RETURN course\_ht | 1 | 1 | 1 |
|  |  | **Total Cost** | 6n^2 + 36n + 7 |
|  |  | **Runtime** | O(n^2) |

**Pseudo Code for the data import method for the HashTable:**

HashTable **dataImport**(string file\_path) { **//Runtime is O(n^2)**

*// @param: string file\_path is for the file\_path of the before mentioned file with the data.*

*// Structure of the file can be found in the table of contents.*

*// For opening the file, we can use the in-file stream method in c++ known as ifstream.*

*// For more information on ifstream please refer to the references page.*

OPEN file at file\_path and save data as a string called raw\_file

CREATE an empty string called temp\_string

CREATE an empty vector of strings called pre\_req

CREATE an empty string called course\_descr

CREATE an empty string called course\_code

CREATE an integer variable called comma\_count

CREATE an empty HashTable variable called course\_ht

FOR each character in raw\_file we will call char

CREATE an empty course object called course

IF char is equal to a new line character “\n”

*// This implies that a new line has occurred, and our collected data*  *// needs to be constructed into a course object and further sent to the*  *// Binary tree for construction.*

CALL method setCourseCode a function of the variable course with the parameter course\_code

CALL method setCourseDescription a function of the variable course with the parameter course\_descr

IF the size of pre\_req is greater than zero

CALL method setCoursePreReqs a function of the variable course

with the parameter pre\_req

CREATE a course object called temp\_course and set it equal to the return of the method call Search a function of the variable course\_ht.

FOR each course code in pre\_req called pre\_course\_code

*// We are searching for each prerequisite in the vector*

IF the pre\_course\_code is not equal to the return of method call Search a function of the variable course\_ht

*// The Search function will search the hash table and // return the course object unless it does not exist. If it // doesn’t exist it will return an empty course object.*

THROW exception with the message “Course prerequisite found that cannot be validated.”

*// All data has been validated per the requirements page and is ready to // insert into hash table.*

CALL method Insert a function of the variable course\_ht with the parameter course.

SET comma\_count equal to 0

SET temp\_string equal to “” (Empty String)

SET course equal to an empty Course object

SET pre\_req equal to an empty vector of strings

SET course\_code equal to “” (Empty String)

SET course\_descr equal to “” (Empty String)

IF ELSE char is equal to a comma AND comma\_count is equal to zero

*//We have found the end of our course\_code because it’s the start of the line and // before the first comma.*

SET course\_code equal to temp\_string

INCREMENT comma\_count by one

SET temp\_string equal to “” (Empty String)

IF ELSE char is equal to a comma AND comma\_count is equal to 1

*//We have found the end of our course description because it’s the start of the line // first comma and before the second comma.*

SET course\_descr equal to temp\_string

SET temp\_string equal to “” (Empty String)

INCREMENT comma\_count by one

IF ELSE char is equal to a comma AND comma\_count is greater or equal to 1

*//We have found a prerequisite*

APPEND temp\_string as an item to vector pre\_req

SET temp\_string equal to “” (Empty String)

INCREMENT comma\_count by one

IF ELSE char is equal to “\0”

*// The character “\0” is the denotation of null or an empty item. Meaning we’re // at the end of the document.*

RETURN course\_ht

# Hash Tables Class

Pseudo Code for Hash Table Class – Private functions:

**Runtime Analysis of HashTable class constructor:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| CREATE unsigned int called tableSize equal to global value DEFAULT\_SIZE | 1 | 1 | 1 |
| CREATE vector of type Node\* called nodes | 1 | 1 | 1 |
| CALL method for Globally defined vector ‘nodes’ of type ‘Node’ to use ‘resize’ passing the default class value of ‘tableSize’. | n | 1 | n |
|  |  | **Total Cost** | n+2 |
|  |  | **Runtime** | O(n) |

HashTable:: **HashTable**() { ***// Runtime is O(n)***

*// Constructor*

CREATE unsigned int called tableSize equal to global value DEFAULT\_SIZE

CREATE vector of type Node\* called nodes

*// Refer to references page for public member function std::vector::resize*  *// which defines the complexity as linear O(n).*

CALL method for Globally defined vector ‘nodes’ of type ‘Node’ to use ‘resize’ passing the default class value of ‘tableSize’.

}

**Runtime Analysis of HashTable class de-constructor:**

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # of Times Executed | Total Cost |
| FOR each node in global variable ‘nodes’ | 1 | n | n |
| WHILE the nodes pointer is NOT empty | 1 | n | n |
| CREATE variable ‘temp\_delete’ of type Node | 1 | n | n |
| SET ‘temp\_delete’ to equal the CURRENT node pointer value | 1 | n | n |
|  |  |  |  |
|  |  | Total Cost | 4n |
|  |  | Runtime | O(n) |

HashTable:: **~HashTable**() { ***// Runtime is O(n)***

*// De-constructor*

FOR each node in global variable ‘nodes’

WHILE the nodes pointer is NOT empty

CREATE variable ‘temp\_delete’ of type Node

SET ‘temp\_delete’ to equal the CURRENT node pointer value

SET the CURRENT node pointer to equal the next node point value

}

HashTable::**Hash**(int key) { **//Runtime is O(1)**

**//**Hash the key by finding the modulo of key by the tablesize

RETURN key % tableSize

}

**Runtime Analysis of Insert function:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| (Constant Operations evaluating to ‘1’ for each line. Reduced for readability) | 4 | 1 | 4 |
| WHILE the nodes pointer is NOT empty | 1 | n | n |
| (Constant Operations evaluating to ‘n’ for each line. Reduced for readability) | 10 | n | 10n |
|  |  |  |  |
|  |  | **Total Cost** | 11n + 4 |
|  |  | **Runtime** | O(n) |

HashTable::**insert**(Course course) { **//Runtime is O(n)**

CREATE variable ‘key’ and SET it equal to the return of method ‘hash’ passed the property of ‘course’ that is ‘courseCode’

CREATE variable ‘og\_key’ and SET it equal to ‘key’

CREATE variable ‘hashed\_node’ and SET it equal to the Node at ‘key’ in global ‘nodes’

CREATE variable ‘assigned’ and SET it to equal ‘false’

WHILE ‘assigned’ is equal to ‘false’

IF ‘hashed\_node’s key property is equal to constant ‘UINT\_MAX’

SET the node at ‘key’ equal to a new Node type variable with course and key as constructors

SET ‘assigned’ to equal ‘true’

ELSE IF ‘key’ is equal to ‘nodes’ size –1

SET ‘key’ equal to zero

ELSE IF ‘key’ is equal to ‘og\_key’ - 1

THROW and out of range exception because all buckets are full

ELSE

SET ‘key’ equal to ‘key’ + 1

SET ‘hashed\_node’ equal to global ‘nodes’ at the node position ‘key’

}

HashTable::**Remove**(Course course) { **//Runtime is O(n)**

CREATE variable ‘key’ and SET it equal to the return of method ‘hash’ passed ‘CourseCode’ a property of course

CALL method ‘erase’ for ‘nodes’ while passing position ‘nodes.begin() + key’

}

HashTable::**Search**(Course course) { **//Runtime is O(n)**

CREATE a variable of type Course called ‘course that is empty

CREATE a variable ‘key’ and SET it equal to the return of method ‘hash’ passed ‘CourseCode’ a property of course

IF node at ‘key’ position in ‘nodes’s courseCode property is equal course’s courseCode

RETURN node at ‘key’ position in ‘nodes’

ELSE

Return an empty course object

}

# Vector Sorting

**Runtime Analysis of data import method:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| OPEN file at file\_path and save data as a string called raw\_file | n | 1 | n |
| (Constant Operations evaluating to ‘1’ for each line. Reduced for readability) | 5 | 1 | 5 |
| FOR each character in raw\_file we will call char | 1 | n | n |
| (Constant Operations evaluating to ‘n’ for each line. Reduced for readability) | 25 | n | 25n |
| FOR each course in pre\_req we will call req | n | n | n^2 |
| CALL vector function find with parameters of the first and last elements of pre\_req and req as the key and save the return in found. | n | n^2 | n^2 \* n |
| (Constant Operations evaluating to ‘n^2’ for each line. Reduced for readability) | 10 | n^2 | 10n^2 |
|  |  | **Total Cost** | 12n^2 + 27n + 5 |
|  |  | **Runtime** | O(n^2) |

**Pseudo Code for the data import method:**

Vector<course> **dataImport**(string file\_path) { **//Runtime is O(n^2)**

*// @param: string file\_path is for the file\_path of the before mentioned file with the data*   *// structure.*

*// For opening the file, we can use the in-file stream method in c++ known as ifstream.*

OPEN file at file\_path and save data as a string called raw\_file

*// nl\_file is short for new line file and will be used to store the data before completing*  *validation.*

CREATE an empty course object called course

CREATE an empty string called temp\_string

CREATE an integer variable called comma\_count

CREATE an empty vector of type course called courses

CREATE an empty vector of type string called pre\_req

FOR each character in raw\_file we will call char

IF char is equal to a new line character “\n”

CALL method setCoursePreReqs a function of the variable course with the parameter pre\_req

APPEND course to the courses vector

SET comma\_count equal to 0

SET temp\_string equal to “” (Empty String)

IF ELSE char is equal to a comma AND comma\_count is equal to 0

CALL method setCourseCode a function of the variable course with the parameter temp\_string

SET temp\_string equal to “” (Empty String)

INCRIMENT comma\_count by one

IF ELSE char is equal to a comma AND comma\_count is equal to 1

CALL method setCourseDescription a function of the variable course with the parameter temp\_string

SET temp\_string equal to “” (Empty String)

INCRIMENT comma\_count by one

IF ELSE char is equal to a comma AND comma\_count is greater or equal to 1

APPEND temp\_string as an item to vector pre\_req

SET temp\_string equal to “” (Empty String)

INCRIMENT comma\_count by one

IF ELSE char is equal to “\0”

*// The character “\0” is the denotation of null or an empty item.*

CALL method setCoursePreReq a function of the variable course with the parameter pre\_req

*// This for loop is to check to ensure that each course code in pre\_req is a*  *// valid course code per the specifications in the rubric provided.*

CREATE boolean type variable called found equal to false

FOR each course in pre\_req we will call req

*//Using built-in vector function “find” with complexity of O(1).*  *//This takes in the first and last elements as a range to search and*  *//the item you’re looking for as a key. Refer to References page*  *//(std::find).*

CALL vector function find with parameters of the first and last elements of pre\_req and req as the key and save the return in found.

IF variable found is false

*//Prerequisite does not exist. We can change this to instead*  *delete the prerequisite instead of failing.*

THROW ERROR KeyError

APPEND course to the courses vector

*// Time to clean up variables used and free up memory*

DELETE temp\_string

DELETE comma\_count

DELETE pre\_req

DELETE course

RETURN the variable courses

ELSE

CONCATANATE char to temp\_string

**Runtime Analysis of vector print function:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# of Times Executed** | **Total Cost** |
| for all courses | 1 | n | n |
| if the course is the same as courseNumber | 1 | n | n |
| print out the course information | 1 | n | n |
| for each prerequisite of the course | 1 | n^2 | n^2 |
| print the prerequisite course information | 1 | n^2 | n^2 |
|  |  | **Total Cost** | 2n^2 + 3n |
|  |  | **Runtime** | O(n^2) |

**Pseudo Code for the print method:**

void print(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 the prerequisite course information

# Advantages, Disadvantages, and Recommendation

Each data structure has its own advantages and disadvantages, and each proves to be better in certain aspects of handling information. It ultimately boils down to the needs of the project to match the appropriate data structure. Vectors and Hash tables share many similarities, but the key differences shine when it comes to the complexity of their functions. Vectors use indices and have a search function of O(n) where ‘n’ is the size of the vector. This means that every item in the vector needs to be compared to the given key. Hash tables, however, use a hashing function to identify the ‘bucket’ the object is stored within. This means that Hash tables have an average time complexity of O(1) meaning it is near instantly located despite the input size. It should be noted that it can have a worst-case time complexity of O(n) like a vector. Hash tables come with a hefty price tag in terms of memory usage when compared to a binary search tree (BST). A BST typically searches for items in a time complexity of O(log n) but can face difficulties resulting in a O(n) worst-case time complexity. These ideas can be visualized in the tables below.

**Average Time complexities:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Search** | **Remove** | **Insert** |
| **Binary Search Tree** | O(log n) | O(log n) | O(log n) |
| **Hash Tables** | O(1) | O(1) | O(1) |
| **Vector** | O(n) | O(n) | O(n) |

**Worst Case Time complexities:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Search** | **Remove** | **Insert** |
| **Binary Search Tree** | O(n) | O(n) | O(n) |
| **Hash Tables** | O(n) | O(n) | O(n) |
| **Vector** | O(n) | O(n) | O(n) |

The table above shows three methods that are most used throughout the implementation. This implies that the optimal growth rate is a Hash Table because it has the best chance of staying at a constant complexity. We can also observe that all three data structures have the same worst-case time complexity of O(n). Regarding when to use a vector over a hash table, if you have a ‘key’ that can be hashed and is unique to each piece of data such as our course code field then it would be wise to choose a hash table over a vector. It should be noted that while hash tables time complexity is a constant a time complexity of O(log n) is not that far off in efficiency. The differences are most noticeable as the input size grows. Since we do not plan to have an extremely large input size because it's the number of courses in the computer science program these complexities should be near unnoticed in difference.

In a Binary Search Tree, you may be compelled to believe that it would prove useful when storing hierarchal data such as a class and what its prerequisites would be. You would eventually come across the scenario in which the data is sorted upon arrival which will prove your binary tree useless as it will always hit its worst-case time complexity because it is traversing every point of data collected up to that point. If the data structure needed to store duplicates, be sorted in the data structure, or even need range queries I would recommend a binary search tree but that doesn’t appear to be so. The needs of this project appear to instead be querying courses, printing all data, inserting data, and deleting data as its core use. I would recommend a hash table because of its constant complexity in these factors.

# References

Southern New Hampshire University (n.d.). CS 300 Course Information. SNHU. https://learn.snhu.edu/content/enforced/1609318-CS-300-11304.202451- 1/course\_documents/CS%20300%20Course%20Information.pdf? \_&d2lSessionVal=0Ng0J8VcRzLlAprpTzr7MZxGJ&ou=1332051&ou=1609318

Southern New Hampshire University. (2024). CS 300 Pseudocode Document. SNHU. https://learn.snhu.edu/content/enforced/1609318-CS-300-11304.202451- 1/course\_documents/CS%20300%20Pseudocode%20Document.docx? \_&d2lSessionVal=0Ng0J8VcRzLlAprpTzr7MZxGJ&ou=1332051&ou=1609318

CPlusPlus. (n.d.). Class <fstream> std::Ifstream. https://cplusplus.com/reference/fstream/ifstream/

CPlusPlus. (n.d.). Public Member Function std::Ifstream::open

https://cplusplus.com/reference/fstream/ifstream/open/

CplusPlus (n.d.). Public member function <vector> std::Vector::size

<https://cplusplus.com/reference/vector/vector/size/>

CplusPlus (n.d.). Public member function <vector> std::Vector::resize

<https://cplusplus.com/reference/vector/vector/resize/>

CPlusPlus (n.d.). Function <cstdlib> atoi

<https://cplusplus.com/reference/cstdlib/atoi/>

CPlusPlus (n.d.). Function template std::find

<https://cplusplus.com/reference/algorithm/find/>

CPlusPlus (n.d.). Function <cstring>strcmp

<https://cplusplus.com/reference/cstring/strcmp/>

CPlusPlus (n.d.). Function template std::Sort

https://cplusplus.com/reference/algorithm/sort/