Chris Trimmer

CS-300-T1159 DSA: Analysis and Design

Project 1: Pseudocode and Runtime Analysis

22EW1 – 10/04/2022

Table of Contents

[I. Introduction 3](#_Toc115435608)

[II. Pseudocode 4](#_Toc115435609)

[1. Pseudocode for each data structure. 4](#_Toc115435610)

[Vector 4](#_Toc115435611)

[Hash Table 5](#_Toc115435612)

[Binary Search Tree 7](#_Toc115435613)

[2. Main Menu Pseudocode. 9](#_Toc115435614)

[3. Print courses alphanumerically. 11](#_Toc115435615)

[Vector 11](#_Toc115435616)

[Hash Table 12](#_Toc115435617)

[Binary Search Tree 13](#_Toc115435618)

[III. Evaluation 14](#_Toc115435619)

[4. Runtime and memory analysis. 14](#_Toc115435620)

[Vector 14](#_Toc115435621)

[Hash Table 15](#_Toc115435622)

[Binary Search Tree 17](#_Toc115435623)

[5. Data structure advantages and disadvantages. 19](#_Toc115435624)

[Vector 19](#_Toc115435625)

[Hash Table 19](#_Toc115435626)

[Binary Search Tree (BST) 20](#_Toc115435627)

[6. Final Recommendation. 21](#_Toc115435628)

# I. Introduction

The purpose of this document is to provide a pseudocode and algorithm analysis of code for a course planner that we are going to design for ABC University (ABCU). The pseudocode will consist of functions that pertain to file parsing, creating objects and storing them in a data structure, searching the data structure, and printing the data. I will also provide pseudocode for the main menu of the application. The main menu will include options for loading data in the data structure, printing the entire course list, printing information on individual courses, and exiting the application.

The main object used in the assignment is a Course class. Each course will contain a string value for its id, a string value for its title, and a vector to store prerequisite courses for the given Course object. In this scenario, the CourseId will be used as the key for the object. I use a vector to store prerequisites because a course can have more than one prerequisite and a vector is suitable for dynamic insertion. The tree data structure is built using objects called Nodes, which store the course object and pointers to children nodes.

We are comparing three data structures in this project: vector, hash table, and binary search tree. First, I will provide pseudocode for each data structure. Next, I will provide an evaluation of each data structure in the form of an algorithm analysis this show the runtime in Big-O notation. After that, I will explain advantages and disadvantages of each structure. Finally, I will provide a final recommendation of the data structure I will use in the final coding project.

# II. Pseudocode

## 1. Pseudocode for each data structure.

### Vector

*File opening, reading, parsing, and formatting*

Vector<Course> LoadSchedule(string filepath) {

Create infile ifstream object

Use infile object to open the filepath

If infile is not null

Create string object to hold a line read from the file

Create a char object and set to a comma, as the delimiter

Create a string object to hold a word in the stringstream object, parsed using the delimiter

Create a vector of strings that will hold words of each line

Loop through infile object and store each line in the line object

Pass each line as an object to stringstream object and parse using delim

Loop through stringstream and push back each word in the line to temp vector

// check file format

If the line contains less than two words, then this is an incomplete record

Print output to user regarding invalid file format

Return to caller

// verify that prereqs are valid

For each word after the first two

If the word is in not the course list

Skip the word

// Creating and storing the object is covered in AddCourse function

Call AddCourse and pass the vector of valid words

Clear the courseLine vector before starting the next loop

} // end LoadSchedule

*Course object creation*

Void AddCourse(vector<string>& line, vector<Course>& courses) {

Instantiate Course object

For each word in line

Set course id to line[0]

Set course title to line[1]

For each additional word in line

Push back word into the prereq vector stored in the Course object

// the course now has all its data

Push back the course object into the vector of courses

}

*Print course and its prerequisites*

Void DisplayIndividualCourse() {

For each Course in Schedule vector

Print course information (id and title)

For each Course p in c.prereq vector

Print prereq information (id only)

}

### Hash Table

*File opening, reading, parsing, and formatting*

Void LoadCourses(string filepath, HashTable\* hashTable) {

Create infile ifstream object

Use infile object to open the filepath

If the infile object returns null

return to caller immediately

Create string object, line, to hold a line read from the file

Create a char object, delim, which is a “,” (comma), to use as delimiter when reading each line

Create a string object, word, to hold each word in the line

Create a vector of strings, courseLine, that will hold each word

Loop through infile object, and store each line in the line object

Pass each line as an object to stringstream object, fullLine and parse using delim

Loop through stringstream and push back each word in the line to temp vector

// check file format

If the line contains less than two words, then this is an incomplete record

Print output to user regarding invalid file format

Return to caller

// verify that prereqs are valid

For each word after the first two

If the word is in not the course list

Skip the word

// Creating and storing the object is covered in AddCourse function

Call AddCourse and pass vector of valid words

Clear the courseLine vector before starting the next loop

} // end LoadCourses

*Course object creation*

Void AddCourse(vector<string>& line, vector<Course>& courses) {

Instantiate Course object

For each word in line

Set course id to line[0]

Set course title to line[1]

For each additional word in line

Push back word into the prereq list stored in the Course object

// the course now has all its data

Call insert function of hash table class passing the Course object

}

Void HashTable::Insert(Course course) {

Assign local courseKey variable by calling hash function

Create pointer (curr) to the node at the index of the hashed courseKey

If curr is equal to nullptr

Assign curr to the node at this index

Else

If the old key at this node is (UINT\_MAX)

Set curr key to currKey

Set curr course to course

Set curr next to nullptr

Else

Loop through the list until we get to end

Set next node of curr to be the new node

}

*Print course and its prerequisites*

void PrintIndividualCourse() {

if list is empty

return to caller

Set a pointer to the head of the list

Loop through list until the current node is not nullptr

If the current node courseId is equal to the course the user is searching for

Print the course information and its prerequisites

Set current node pointer to the next node

}

### Binary Search Tree

*File opening, reading, parsing, and formatting*

Void LoadCourses(string filepath, BinarySearchTree\* bst) {

Create vector to store the master course list

Call to create the master course list using filepath

Create infile ifstream object

Use infile object to open the filepath

If the infile object returns null

return to caller immediately

Create string object, line, to hold a line read from the file

Create a char object, delim, which is a “,” (comma), to use as delimiter when reading each line

Create a string object, word, to hold each word in the line

Create a vector of strings, courseLine, that will hold each word

Loop through infile object, and store each line in the line object

Pass each line as an object to stringstream object, fullLine and parse using delim

Loop through fullLine, and push back each word in the line to courseLine vector

// check file format

If the courseLine vector contains less than two objects, then this is an incomplete record

Print output to user regarding invalid file format

Return to caller

// verify if prerequisite classes are valid

For each word in courseLine vector

Remove whitespace from each word

If the word is not a valid course

Then skip the word

// now that we have a valid courseLine and valid prerequisites, perform object creation

Call AddCourse

(… outer loop continues after completion of creating the course object, and inserting into the hash table)

Clear the courseLine vector before starting the next loop

} // end LoadCourses

*Course object creation*

Void AddCourse(String courseLine) {

Instantiate Course object

For each word in line

Set course id to line[0]

Set course title to line[1]

For each additional word in line

Push back word into the prereq list stored in the Course object

// the course now has all its data

Call insert function of binary search tree class passing the Course object

}

Void InsertRecursive(Node\* node, Course course) {

If course.id is less than the node.id

If node->left is null

Create new node with course object

Assign new node to node->left

Else

Recursively call InsertRecursive with node->left and the course

Else

If node->right is null

Create new node with course object

Assign new node to node->right

Else

Recursively call InsertRecursive with node->right and the course

}

*Print course and its prerequisites*

Void PrintIndividualCourse(Course course) {

Get user input for the courseId they are searching for

Search the BST for the courseId

If the course exists

Print the course information and its prerequisites

Else

Inform user the string they are searching for does not exist

}

## 2. Main Menu Pseudocode.

The menu pseudocode will consist of three functions: a function to print the menu options, a function to get user input, and a function to process the input in the main loop.

Void PrintMenu() {

Print “Main Menu”

Print “1. Load Courses”

Print “2. Display Courses”

Print “3. Display Individual Course Information”

Print "4. Exit Application”

}

Int GetChoice() {

Set variable for the choice to default value of 0

Print “Enter your selection”

Get user input and set it to the choice variable

Clear the input stream

Return the choice to calling function

}

Int main() {

Create an int variable for the choice

Create a string variable to hold user input for courseId

While the choice is not 4 {

Call the print menu function

Call the GetChoice function and set it to the choice variable

Use switch logic statement based on the choice:

// load courses

Case 1:

call LoadCourses function

Break from case

// print the courses alphanumerically

Case 2:

Call the BST InOrderTravesal function

Break from case

// print an individual course and its associated prerequisites

Case 3:

Get user input for the courseId they are searching for and set it to the search string

Search for the course using the BST search function

If the course exists

Call the print course info function

Else

Inform user that the string they entered cannot be found

Break from the case

// exit the application

Case 4:

Print “Thank you for using the ABCU Course Planner application”

Break from the case

// Default case statement

Default:

Print “invalid input, please try again”

Break from the case

}

}

## 3. Print courses alphanumerically.

### Vector

The vector data structure is usually built upon a dynamically resizing array. When inserting into a vector, nodes can be inserted by index, or most optimally at the back. It is more efficient to insert them at the back, which means the nodes may not be inserted in correct order. Thus, to print the courses alphanumerically we must first sort them. For this will use quicksort as it has an average O(NLogN) runtime. However, in the worst case, quicksort will perform in O(N^2). This can happen based on the pivot that is chosen, and the order of the elements. Selection sort will perform at O(N^2) on average and in worst-case. The quicksort function works by creating a pivot in the vector and dividing elements into low and high partitions. Care must be taken to choose a pivot point that will not result in worst case performance. The process effectively divides the vector at the pivot, and then continues to divide the partitions. Once the partitioning is completed, the process will sort the nodes recursively. Finally, the vector will be sorted very efficiently. This is competed using two functions: the main quicksort function and a pivot function. After performing the quicksort, we loop through each node in the vector and print the course information. Printing each node in a vector is a linear operation O(N) runtime because it is dependent on the number of nodes in the vector.

void PrintVectorCourses() {

if vector is empty

return to caller

Call quicksort function

For each node in the vector

Call the print course function

}

### Hash Table

A hash table can be built upon a foundational structure like a vector or array. Each index in the vector contains a list of nodes, which are programmed as linked lists. This is a highly efficient way to store large amounts of data for insertion, retrieval, and deletion. For printing the entire list of courses, we simply just create a temp pointer to the head node where the bucket is located, and then walk the list using the pointer by setting the temp pointer to the next node in the list. A hash table is built based on hashing the key of each node and mapping it to the location within the table. Therefore, nodes are not inserted in order, so we must sort the nodes. We can use a process like what we did for the vector and sort the nodes using quicksort. First, we check to make sure the list is not empty. After that, we walk the list of nodes and push each node to a temporary vector. Then, we can sort the vector. Finally, we loop through the vector and print each node in alphanumeric order. Even though quicksort is O(NLogN) on average, the overall worst case runtime analysis of this entire printing process is O(N^2) in the case of a poor pivot. This turns out to be one of the disadvantages of a hash table because it is not suitable for sorting.

void PrintHashTable() {

if list is empty

return to caller

Create temporary vector to hold the nodes so that can be sorted

Set a pointer to the head of the list

Loop through list until the current node is not nullptr

Push each node to the vector

Set current node pointer to the next node

Call the quick sort function

Loop through the vector

Print the node information

}

### Binary Search Tree

A Binary Search Tree (BST) is a hierarchical data structure. Each node contains pointers to left and right children nodes. As nodes are inserted into the tree, they are stored to the left or right of a parent node. This process of dividing the data into a tree like structure is highly efficient at most operations, operating at O(NlogN) at average runtime. In the worst case, it will operate at O(N) time for printing our data. This will occur is data is already in order when it is inserted. The InOrder traversal can be used to print the courses alphanumerically. It first tests to make sure the tree is not empty. Then, it recursively calls itself passing the left side during each iteration. Next, we call the print statement. Then, we recursively call the right side for each iteration.

As the call stack unwinds, the list is printed alphanumerically. From the perspective of space, there is slightly more memory required in a BST, because each node must hold two pointers at minimum. The runtime is completed in O(NLogN) on average, but as mentioned earlier, it will run at O(N) if the data is already sorted.

Void InOrderTraversal (Node\* node) {

if node is not null

recursively call inOrder with left side

print the node info

recursively call inOrder with right side

}

# III. Evaluation

## 4. Runtime and memory analysis.

This section will detail algorithm analysis of the file reading, parsing, and creation of course objects. This analysis will include the Big-O value based on cost per line and the number of times the line will execute. I will provide this analysis for each data structure. Note that the hash table and BST data structures use an insertion helper function to insert into the respective data structure. For the vector insertion, we just use the built-in push\_back function, so I didn’t need to create a separate insertion function.

### Vector

*File Opening and Parsing*

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | Execution Times | Total Cost |
| Create ifstream object | 1 | 1 | 1 |
| Open filepath | 1 | 1 | 1 |
| if infile is not null | 2 | 1 | 2 |
| Create 5 local variables | 5 | 1 | 5 |
| for each line | 1 | n | n |
| create sstream object | 1 | 1 | 1 |
| for each sstream object | 1 | n | n |
| if courseLine less than 2 | 2 | 1 | 2 |
| return to caller | 1 | 1 | 1 |
| for each word after 2 | 1 | n | n |
| if the word not in course list | 1 | 1 | 1 |
| skip the word (continue) | 1 | 1 | 1 |
| add word to vector | 1 | 1 | 1 |
| call AddCourse | 2n + 5 | 1 | 2n + 5 |
| Total Cost: | | | 5n + 21 |
| Runtime | | | O(n) |

*Creating and Storing Course Objects*

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | Execution Times | Total Cost |
| Create course object | 1 | 1 | 1 |
| for each word in lines | 1 | n | n |
| set course id to line[0] | 1 | 1 | 1 |
| set course title to line[1] | 1 | 1 | 1 |
| for each additional word in lines | 1 | n | n |
| push back word to prereq vector | 1 | 1 | 1 |
| push back completed Course | 1 | 1 | 1 |
| Total Cost: | | | 2n + 5 |
| Runtime | | | O(n) |

### Hash Table

*File Opening and Parsing*

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | Execution Times | Total Cost |
| Create ifstream object | 1 | 1 | 1 |
| Open filepath | 1 | 1 | 1 |
| if infile is not null | 2 | 1 | 2 |
| Create 5 local variables | 5 | 1 | 5 |
| for each line | 1 | n | n |
| create sstream object | 1 | 1 | 1 |
| for each sstream object | 1 | n | n |
| if courseLine less than 2 | 2 | 1 | 2 |
| return to caller | 1 | 1 | 1 |
| for each word after 2 | 1 | n | n |
| if the word not in course list | 1 | 1 | 1 |
| skip the word (continue) | 1 | 1 | 1 |
| add word to vector | 1 | 1 | 1 |
| call AddCourse | 4n + 32 | 1 | 4n + 32 |
| Total Cost: | | | 7n + 48 |
| Runtime | | | O(n) |

*Creating and Storing Course Objects*

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | Execution Times | Total Cost |
| Instantiate blank course object | 1 | 1 | 1 |
| for each word in lines | 1 | n | n |
| set course id to line[0] | 1 | 1 | 1 |
| set course title to line[1] | 1 | 1 | 1 |
| for each additional word in lines | 1 | n | n |
| push back word to prereq vector | 1 | 1 | 1 |
| call insert function to pass course object | n + 14 | 1 | n + 14 |
| Total Cost: | | | 3n + 18 |
| Runtime | | | O(n) |

*Insert Function*

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | Execution Times | Total Cost |
| set local var to result of hash | 2 | 1 | 2 |
| create pointer to index node | 1 | 1 | 1 |
| if curr is equal to nullptr | 2 | 1 | 2 |
| assign curr to the node index | 1 | 1 | 1 |
| else if old key is UINT\_MAX | 2 | 1 | 2 |
| set curr key to courseId | 1 | 1 | 1 |
| set curr course to course | 1 | 1 | 1 |
| set curr next to nullptr | 1 | 1 | 1 |
| else loop through list till end | 1 | n | n |
| set next of curr to new node | 1 | 1 | 1 |
| return key mod tableSize | 2 | 1 | 2 |
| Total Cost: | | | n + 14 |
| Runtime | | | O(n) |

### Binary Search Tree

*File Opening and Parsing*

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | Execution Times | Total Cost |
| Create master course vector | 1 | n | n |
| Create ifstream object | 1 | 1 | 1 |
| Open filepath | 1 | 1 | 1 |
| if infile is null, return | 2 | 1 | 2 |
| Create 4 local variables | 4 | 1 | 4 |
| while file has lines to read | 1 | n | n |
| create sstream object | 1 | 1 | 1 |
| while there are words in the stream object | 1 | n | n |
| trim whitespace from each word | 4 | n | 4n |
| for each sstream object | 1 | n | n |
| push each word to vector | 1 | n | n |
| if courseLine less than 2 | 2 | 1 | 2 |
| return to caller | 1 | 1 | 1 |
| Create course object | 1 | 1 | 1 |
| if size of courseLine > 2 | 2 | 1 | 2 |
| for each prereq | 1 | n | n |
| compare prereq to master course list | 1 | n | n |
| if the prereq is found | 2 | 1 | 2 |
| insert prereq to course | 1 | 1 | 1 |
| Search tree to verify if course has already been added | 1 | logN+1 | logN+1 |
| if the course doesn't exist | 2 | 1 | 2 |
| then call Add Course | 1 | 2logN + 11 | 2logN + 11 |
| clear the courseLine vector | 1 | 1 | 1 |
| close the file | 1 | 1 | 1 |
| Total Cost: | | | 3logN + 11n + 34 |
| Runtime | | | O(NlogN) |

*Creating and Storing Course Objects*

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | Execution Times | Total Cost |
| Instantiate blank course object | 1 | 1 | 1 |
| for each word in lines | 1 | n | n |
| set course id to line[0] | 1 | 1 | 1 |
| set course title to line[1] | 1 | 1 | 1 |
| for each additional word in lines | 1 | n | n |
| push back word to prereq vector | 1 | 1 | 1 |
| call insert function to pass course object | 1 | 1 | 2logN + 10 |
| Total Cost: | | | 2logN + 2n + 14 |
| Runtime | | | O(NlogN) |

*Insert Function*

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | Execution Times | Total Cost |
| if id is less than node Id | 2 | 1 | 2 |
| if node->left is null | 2 | 1 | 2 |
| create new node with object | 1 | 1 | 1 |
| assign new node to node->left | 1 | 1 | 1 |
| else, recursively call insert w/left | 1 | logN+1 | logN+1 |
| else if node->right is null | 1 | 1 | 1 |
| create new node with object | 1 | 1 | 1 |
| assign new node to node->right | 1 | 1 | 1 |
| else, recursively call insert w/right | 1 | logN+1 | logN+1 |
| Total Cost: | | | 2logN + 9 |
| Runtime: | | | O(NlogN) |

## 5. Data structure advantages and disadvantages.

### Vector

A vector is one of the fundamental data structures used in C++. It is a standard container in C++, and therefore has many built-in operations that make it easy to use and easy to integrate with in many aspects of the C++ language. The underlying structure of a vector is built on a dynamically resizing array, so operations such as searching or inserting into the middle or beginning have a worst-case runtime of O(N). Vectors have a multitude of advantages. Firstly, vectors can be resized dynamically, as opposed to arrays which require re-allocating memory. Second, vectors have built-in safety to deal with unintentionally trying to access indexes that are out of bounds. Another benefit is that we can access indexes in constant O(1) time.

### Hash Table

Hash tables are fast for insertion, retrieval, and deletion, especially when dealing with very large amounts of data. Hash tables work by mapping a key to a hashed value. This location, or bucket, can be used to store a list of nodes that map to the same hash. Consider that we may need to store millions of records. It would not be efficient to store them consecutively in an array or vector of that size. With a hash table, we can effectively create a small number of buckets (by hashing) relative to the size of the whole list, and each bucket would contain several elements that map to that location. Therefore, when searching, we first hash the search key to find the bucket, and then search for the item within the bucket which contains a much smaller number of nodes.

A primary advantage of the hash table is the constant O(1) insertion and retrieval of a node. However, the worst case is O(N) - but on average it operates at O(1). Another advantage is that we get this performance regardless of the size of the data structure if our hash and table implementation is designed well. The design of the hashing algorithm and underlying table can be considered an area of concern, and one of the disadvantages of a hash table – the implementation itself can be complex. As mentioned previously, hash tables are not a good choice for sorting data because data is stored based on the hashing value of the key. However, for large volumes data, it cannot be overstated how powerful and advantageous it is to use hash table.

### Binary Search Tree (BST)

A BST is considered a hierarchical data structure such that each node in the tree can have none, one, or two child nodes. The previous data structures we have discussed are linear structures. The BST is implemented with left and right pointers in each node. The topmost node is the root. Any node that does not have children is a leaf node. The power of a BST is that it solves problems effectively via divide-and-conquer, performing at O(LogN) time for the insertion, search, and deletion operations. It works this way because nodes are inserted to the left or right of a parent node based on a key value that is either less than or greater than the parent key.

However, there is a case when a BST will have a worst-case runtime of O(N). This occurs when the tree is not balanced. An example of when this can happen is when nodes are inserted into the tree in order. This can be avoided if the BST is completed with a balancing implementation such as AVL, or red-black tree, for example. This can be one of the disadvantages of a BST, as implementing a balanced tree is somewhat complex. Another disadvantage can occur if the tree is implemented recursively, because with many nodes, the recursive call stack can impede performance. This can be avoided by implementing the tree with an iterative solution; however, this can lead to complex code. So as part of the design decision, we must consider if the system merits complex code for the benefit of performance.

Having said that, a BST has the powerful advantage of performing at O(LogN) time in most cases. As the data set grows larger, a balanced BST will outperform most other data structures.

## 6. Final Recommendation.

The data structure I plan to use for my project is the BST. Personally, this was my favorite data structure to implement because it builds upon the foundational linked list implementation, and it works by divide-and-conquer. In our ZyBook, they give the non-recursive and recursive implementations for the BST. I found the recursive implementation to be extremely fun and just a great learning experience.

From a technical perspective, I would also choose to implement the project as a BST. The main reason is because of the O(LogN) performance. For the project, we only must deal with a small number of courses. However, we must consider that this project would inevitably include many courses offered at a college. So, even though the sample size is low in the project, we must consider that this will be part of a larger system.

As we see in the earlier analysis of the functions (reading and parsing the file, creating, and storing objects), we can perform these operations in O(LogN) time on the average case. This clearly beats the performance of the vector, which will perform O(N) best case and a worst case of O(N^2) if we must sort using selection sort. The hash table has a very attractive best-case runtime of O(1), which makes it a suitable structure to use for this project as well. However, in this project, it will run at O(N^2) in the worst case because we must sort the data and iterate over a supplementary structure to print the nodes in order. In the BST, we can print the nodes in order using the InOrder traversal, which operates at O(LogN) time on average, and O(N) in worst case. Therefore, from a technical perspective and my own personal experience, I will implement the project using the BST.