CS 300 Project One

LINKS:

* [Part 1: Data Structure Pseudocode](#_Part_1:_Data)
* [Part 2: Menu Pseudocode](#_Part_2:_Menu)
* [Part 3: Pseudocode to Print Course Info Alphanumerically](#_Part_3:_Pseudocode)
* [Part 4: Runtime Analysis Charts](#_Part_4:_Runtime)
* [Part 5: Evaluation of Data Structures Pertaining to Use Case](#_Part_5:_Evaluation)
* [Part 6: Recommended Data Structure](#_Part_6:_Recommended)

# Part 1: Data Structure Pseudocode

Listed below are the pseudocode algorithms for opening /reading a file, parsing each file line into a course object, and printing out the relevant course information. Each algorithm uses a distinct data structure with its own inherent logic.

**//Vector Pseudocode**

/\*Pseudocode for Course class. Each object will have a course number, course name, and a vector of N prerequisites

Class Course{

String courseNumber

String name

Vector<String> prerequisites

}

/\*Pseudocode to import course objects into vector from file. Takes single parameter of an input file. Returns a vector of course objects.

Vector<Course> importCourses(file f){

for each line l in file f

append courseNumber to index i in vector

append name to index i in vector

for each prerequisite p in line l

append prerequisite to index p of prerequisites vector

}

/\*Pseudocode to open file, read/parse data, and check format errors.

The function has parameters of an input file and a vector containing a list of all available courses. Will return true if all conditions are met and false otherwise\*/

Bool importData(file f, Vector<Course> courses){

open file f

declare vector inputVector to store each line

while a new line remains in file f:

split the line into separate elements

if less than 2 elements

output error and return false

else append elements to inputVector

for each prerequisite in inputVector

if prerequisite not in courses vector

output error and return false

}

**// Tree pseudocode**

//opens a file, reads data from the file, parses each line, checks for errors, and creates course object for each line.

void loadFile(String filename){

if file is open and no errors exist

for each line in file

declare a course data object struct

parse course number, name, into struct

for each prerequisite in course

insert prerequisite into struct

INSERT course into BST

}

//Inserts courses objects into BST

void insertCourse(Node\* node Course c){

If root node is empty

Set course object as root node

Else

If course object id is less than current node

If current node’s left successor is null

Insert course object as left successor

Else

Recurse down the left subtree

Else

If current node’s right successor is null

Insert course object as right successor

Else

Recurse down the right subtree

}

//Prints the course objects for BST in order

Void printCourseInfo(Node\* node){

If root node is not null

Recurse down the left subtree

Output course ID, name

For each prerequisite p

Output p

Recurse down the right subtree

}

**// Hashtable pseudocode**

//opens a file, reads data from the file, parses each line, and checks for errors.

void loadFile(String filename){

if file is open and no errors exist

for each line in file

declare a course data struct

parse course number, name, into struct

for each prerequisite in course

insert prerequisite into struct

INSERT course into hash table

}

//Inserts course object into hash table data structure

void insertCourse(Course course){

hash course number to create key

retrieve node using key

if no entry found for key

assign node to the key position

else

while the next node is not null

set previous node equal to its next pointer

assign course data to new node

}

//prints contents of hash table to console

Void printCourseInfo(){

For each node in hash table

If node is not null

Output course number, course name, and prerequisites

}

# Part 2: Menu Pseudocode

*The use of the forementioned algorithms within a console application is simplified by creating a simple user interface to display from the console. The Pseudocode listed below provides logic for a main menu containing options to load a data structure from file, print a list of all courses from the data structure, print an individual course’s information, and exiting the program.*

DECLARE a variable to input user choice

DEFINE data structure to hold course information

INSTANTIATE course object

While the user hasn’t entered 4 to exit:

DISPLAY a menu with options 1-4

SWITCH case based on user choice

CASE user enters 1:

INSTANTIATE a data structure instance to parse data into

LOAD course information from file

FOR EACH line in file

CREATE a course object

INSERT object into data structure

CASE user enters 2:

IF data structure is null i.e. User hasn’t loaded a file:

OUTPUT an error and break

ELSE:

PRINT course list from data structure in alphanumeric order

CASE user enters 3:

IF data structure is null i.e. User hasn’t loaded a file:

OUTPUT an error and break

ELSE:

INPUT a course number from user

FOR EACH course number in data structure

IF the course number matches input

PRINT course title

FOR each prerequisite in course

PRINT prerequisite

IF the course number was not found

PRINT error message

CASE user enters 4:

BREAK the loop and exit program

# 

# Part 3: Pseudocode To Print Course Info Alphanumerically

*The menu pseudocode in part two provided an option for printing a list of courses in alphanumeric order. The pseudocode below elaborates on the existing pseudocode by defining the specific logic for alphanumeric printing of courses within each of our three data structures.*

//Vector Pseudocode

//Utilizes insertion sort to sort the array alphanumerically and

prints the result - O(n^2) time complexity

void printCourseInformation(Vector<Course> courses) {

**for all courses**

**initialize index variable j**

**while index j less than its previous index value**

**swap the elements**

**decrement j**

**for all courses**

**print course data**

}

//Tree Pseudocode

//Performs inorder traversal to print tree data - O(n) time complexity

void printCourseInformation(tree) {

**recurse down the left subtree**

**print course data**

**recurse down the right subtree**

}

//Hash Table Pseudocode

//hash tables are not sortable by nature and therefore we must first insert each node’s key into another data structure. Afterwards, we can sort the keys and retrieve the values from the hash table for printing. The below example utilizes a vector to store key data

void printCourseInformation(HashTable\* hashtable) {

**initialize vector to store keys**

**for each node in hastable**

**insert node key into vector**

**sort vector**

**for each key in vector**

**while node is not null**

**if node key matches vector key**

**print course info**

**set node equal to next node**

}

# Part 4: Runtime Analysis Charts

Listed below are the runtime analysis charts for each data structure

Vector Runtime Analysis

//Runtime complexity for course search function

| **Code** | **Line Cost** | **# Times Executes** | **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 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |
|  | | |  |

//Runtime complexity for import data function

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **While a new line remains in file** | 1 | n | n |
| **Split the line into separate elements** | 1 | 1 | 1 |
| **If less than 2 elements** | 1 | 1 | 1 |
| **Output error and return false** | 1 | 1 | 1 |
| **Else append elements to input vector** | 1 | 1 | 1 |
| **For each prerequisite in input vector** | 1 | n | n |
| **If prerequisite not in courses vector** | 1 | 1 | 1 |
| **Output error and return false** | 1 | 1 | 1 |
| **Total Cost** | | | 2N+6 |
| **Runtime** | | | O(n) |

//Runtime complexity for inserting course objects into vector

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For each line in file f** | 1 | n | n |
| **Append coursenumber to index i in vector** | 1 | 1 | 1 |
| **Append name to index i in vector** | 1 | 1 | 1 |
| **For each prerequisite p in line l** | 1 | n | n |
| **Append prerequisite to index p of prerequisite vector** | 1 | 1 | 1 |
| **Total Cost** | | | N+2\*n+1 |
| **Runtime** | | | O(n^2) |

Hash Table Runtime Analysis

//loadFile() runtime analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If file is open and no errors exist** | 1 | 1 | 1 |
| **For each line in file** | 1 | n | n |
| **Declare a course data struct** | 1 | n | n |
| **Parse course number, name, and struct** | 1 | n | n |
| **For each prerequisite in course** | 1 | n | n |
| **Insert prerequisite into struct** | 1 | n | n |
| **Insert course struct into hash table** | 1 | n | n |
| **Total Cost** | | | 6n + 1 |
| **Runtime** | | | O(n) |

//insertCourse() runtime analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Hash course number to create key** | 1 | 1 | 1 |
| **Retreive node using key** | 1 | 1 | 1 |
| **If no entry found for key** | 1 | 1 | 1 |
| **Assign node to the key position** | 1 | 1 | 1 |
| **else** | 1 | 1 | 1 |
| **While next node is not null** | 1 | n | n |
| **Set previous node equal to its next pointer** | 1 | 1 | 1 |
| **Assign course data to new node** | 1 | 1 | 1 |
| **Total Cost** | | | n + 7 |
| **Runtime** | | | O(n) |

Binary Search Tree Runtime Analysis

//loadFile() runtime analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If file is open and no errors exist** | 1 | 1 | 1 |
| **For each line in file** | 1 | n | n |
| **Declare a course data struct** | 1 | 1 | 1 |
| **Parse course number, name, and struct** | 1 | 1 | 1 |
| **For each prerequisite in course** | 1 | n | n |
| **Insert prerequisite into struct** | 1 | 1 | 1 |
| **Insert course struct into BST** | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 5 |
| **Runtime** | | | O(n) |

//insertCourse() runtime analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **IF root node is empty** | 1 | 1 | 1 |
| **Set course object as root node** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **IF course id is less than current node course id** | 1 | 1 | 1 |
| **IF current node’s left successor is null** | 1 | 1 | 1 |
| **Insert course object as left successor** | 1 | 1 | 1 |
| **Else** | 1 | 1 | 1 |
| **Recurse down the left subtree** |  | n | n |
| **Else** | 1 | 1 | 1 |
| **If current node’s right successor is null** | 1 | 1 | 1 |
| **Insert course object as right successor** | 1 | 1 | 1 |
| **Else** |  |  | 1 |
| **Recurse down the right subtree** | 1 | n | n |
| **Total Cost** | | | 2n + 11 |
| **Runtime** | | | O(n) |

//printCourseInfo() runtime analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **If root node is not null** | 1 | 1 | 1 |
| **Recurse down the left subtree** | 1 | n | n |
| **Output course ID and name** | 1 | 1 | 1 |
| **For each prerequisite p** | 1 | n | n |
| **Output p** | 1 | 1 | 1 |
| **Recurse down the right subtree** | 1 | n | n |
| **Total Cost** | | | 3n + 3 |
| **Runtime** | | | O(n) |

# 

# Part 5: Evaluation Of Data Structures Pertaining To Use Case

In our given scenario, the stakeholders have defined requirements of (1) Printing a list of all Computer Science courses in alphanumeric order and (2) For a given course, print out its title and prerequisites. For our purposes, we have chosen to analyze three separate data structures to handle the given use case: Binary Search Tree, Vector, and Hash Table.

**Hash Table Evaluation**

Starting first with the hash table, we can see that it has obvious strengths in terms of search, insertion, and deletion, performing all operations in constant time (O (1)). This would be particularly useful for our search functionality where we need to find a course based off user input and print its title and prerequisites. However, when we consider the fact that we require an alphanumerical sort in the program logic, the hash table has a distinct drawback. With a hash table, sorting logic requires the use of an additional data structure to facilitate the sort. This increases both the time and space complexity of the algorithm.

**Binary Search Tree Evaluation**

A Binary Search Tree is sorted by nature, with each left child being lesser in value than the parent, and each right child being greater. This would be of great use in printing our alphanumeric list of courses. Much of the logic is already handled and we would need only to execute an in order traversal that prints each node. Assuming our tree is well balanced, we would have relatively efficient runtimes of O (log N) for both of our required operations. On the downside, while the runtime complexity of the tree is relatively efficient, the hash table has a much better efficiency for search functions. Also, maintaining balance of the tree is of utmost importance, as an unbalance tree can decay runtime complexity to a worst case of O(n), corresponding to the tree’s height.

**Vector Evaluation**

The vector data structure has the strength of being able to access elements in constant time (O (1)). Appending to the end of the list and removing the last element also are performed in constant time. Direct access in constant time would be useful for our purposes, however this is only possible when the element’s index is already known and thus is not useful to us. In regard to our search functionality, assuming the list is already sorted, we could use a binary search to find courses in O (log(n)) time. An unsorted list would have a worst-case complexity of O(n) assuming we must search the entire list. On the downside, sorting logic is relatively inefficient compared to binary search tree with a worst-case complexity of O(n2).

# Part 6: Recommended Data Structure

Each data structure has its own distinct strengths and weaknesses. When choosing data structures for a particular use case, we must determine what tradeoffs are the most advantageous to overall performance of the system. In my opinion, the binary search tree is the best data structure to implement given our use case.

The sorted structure of the tree is a great advantage, allowing use to perform searches in logarithmic time O(log(n)). While hash tables do provide greater search efficiency, they have the drawback of being relatively inefficient to sort. Printing each element of the Binary Search Tree basically boils down to a simple in order traversal, printing each element along the way. This can be done in O(n) time as we must visit each node of the tree. Both vector and hash table data structures would need to first be sorted before printing the elements, meaning their runtime efficiencies would be no less than O(n). For these reasons I believe that a binary search tree provides the most efficient trade off in terms of performance and ease of implementation.