# CS 300 Pseudocode Document

## Function Signatures

Below are the function signatures that you can fill in to address each of the three program requirements using each of the data structures. The pseudocode for printing course information, if a vector is the data structure, is also given to you below (depicted in bold).

// Vector pseudocode

int numPrerequisiteCourses(Vector<Course> courses, Course c) {

totalPrerequisites = prerequisites of course c

for each prerequisite p in totalPrerequisites

add prerequisites of p to totalPrerequisites

print number of totalPrerequisites

}

void printCourseInformation(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**

}

**//Vector Sorting Pseudocode**

// Reading file:

Function takes in CSV path and hash table

Try to parse file

Try to complete call to open file

While max rowcount is not reached read each line

While column at row i is not null

Add values into course structure using indexing (file[i][0])

If prerequisite not null

Try to find prerequisite in courses vector

Load into course structure

Catch error of prerequisite does not exist

Push course to back using Insert function

Catch error from CSV class

Close file

//Define Print Function

Create method for print function

Take input from user for requested course

If input does not match syntax of courses

Return error

Else for loop until i reaches vector size

If courseNumber at i equals input

If prerequisite is null

Print courseNumber, name and professor

Else print courseNumber, name, professor and prerequisite

Return course not found

//**Hash Table Pseudocode**

// Reading file:

Function takes in CSV path and hash table

Try to parse file

Try to complete call to open file

While max rowcount is not reached read each line

While column at row i is not null

Add values into course structure using indexing (file[i][0])

If prerequisite not null

Try to find prerequisite in courses vector

Load into course structure

Catch error of prerequisite does not exist

Push course to back using Insert function

Catch error from CSV class

Close file

//Create HashTable

Create HashTable class with members and methods

Create Courses Vector

Set default size for the tableSize

//Define Insert Function

Create insert function to load items

Define hash function to index each course

If there is collision

Create a LinkedList for chaining

//Define PrintAll Function

Create print all function to display all loaded courses

Until node end is reached, loop through the nodes

Print courseName, name, professor and prerequisites

If there is collision

Go through the chaining until nullptr

Print courseName, name, professor and prerequisites

//**BinarySearchTree Pseudocode**

// Reading file:

Function takes in CSV path and hash table

Try to complete call to open file

If fails, return error file is not found

Else file is open

Try

For all rows in file

Try If courseName and name exist

If there are prerequisites

Try insert courseName, name, prerequisites

Catch prerequisite error

Else Insert courseName and name

Catch exception and return error of less than 2 parameters

Catch CSV error

Close file

//Create BinarySearchTree

Create BinarySearchTree class with members and methods

Create Courses Vector

Initialize root

//Define Insert Function

If root is null

Set root to new Node

Else call helper function to add Node

//Define addNode Helper Function

If node is not null and node is larger

If node left is not null

Add node to left

Else recurse down left node

Else If node is not null and node is smaller

If node right is not null

Add node to right

Else recurse down right side

//Define Print Course Information Function

Call inOrder helper function with root node

//Define Print Course Information Helper Function

If node not null

return

Recursive call into left subtree

Print courseNumber, name, prerequisites

Recursive call down right subtree

Print courseNumber, name, prerequisites

//Create menu

Create cases for each command line argument

Load in arguments for each choice

While choice does not equal 9

Print out each choice

Take in choice as user input

Case 1:

Initialize timer variable

Call loadCourses function

Print out timer details

Break

Case 2:

Try to sort courses vector

Throw error if courses vector is null

For courseNumber in course list

Print courseNumber

Break

Case 3:

Try to check if data structure is empty

Throw error if null

Call Search function

Return found course

Print title and prerequisites

Break

Case 4:

Exit

//Create function to print computer science course list in alphanumeric order

If first four characters of courseNumber string equals CSCI

Slice last three characters of the courseNumber

For loop i equals 0 and until i is larger than course list

Initialize j int equal to i

While j is greater than 0 and courseNumber[j] > courseNumber[j-1]

Swap courseNumber[j] and courseNumber[j-1]

Decrease j

Print courseNumber list

## Runtime Analysis

Vector

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Read CSV, parse, check for errors | 1 | n | n |
| Create object and insert data | 1 | n | n |
| **Total Cost** | | | 4n |
| **Runtime** | | | O(n) |

Worst runtime would be O(n) for insert

The main advantage of the vector data structure is adding items either to the start or end of the list. Additionally, the vector size is adjustable and does not have to be defined at the beginning. The disadvantage would be the time of the search or insert function for anything that is in the middle of the list.

Hash Table

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Read CSV, parse, check for errors | 1 | n | n |
| Create object and insert data | 1 | n | n |
| **Total Cost** | | | 4n |
| **Runtime** | | | O(n) |

Worst runtime would be O(n) for inserting if each item shares the same key

Advantage of the hash table is the time to look up an item. If the key is created efficiently, there will be no collisions and each key will only store one item. The disadvantage is predicting the size of the data structure resulting in the table to likely fill up. Additionally, hash tables can be complicated to develop and retrieving a sorted list is difficult.

Binary Search Tree

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Read CSV, parse, check for errors | 1 | n | n |
| Create object and insert data | O log(n) | n | n |
| **Total Cost** | | | 4n |
| **Runtime** | | | O(n) |

Worst runtime for insert would be the height of the tree

Advantage of a binary search tree is fast for searching, insertion and deletion with a best runtime of O(log n). The downside of a BST is maintaining a balance so one side is not overloaded compared to the other.

**Recommendation**

Overall, I would recommend the hash table over the vector or binary search tree because of the performance for our main three functions: search, insertion, and delete. The average time for each operation is O(1) compared to O(log (n)) for BSTs or O(n) for vectors. There are two concerns with using hash tables with the first being a hash table runs the risk of filling up depending on how many courses continue to be added. Also, coming up with a key that results in zero collisions is difficult especially as the number of elements added grows.