# Project 1

# Shannon Musgrave

# CS Department of Southern New Hampshire University

# CS-300 Analysis and Design

# Professor Chao Ling

# 02/18/2024

**Section 1 Pseudocode Prerequisites**

# // Common to all, Course Object for pseudocode

# Structure Course{

# courseId: string

# courseName: string

# prereqs: vector<string>

# }

// Common to all, printing course

void PrintCourse(Course course){

**OUTPUT** courses courseId

**OUTPUT** courses courseName

**FOR** all prereqs in course prereq list

**OUTPUT** prereq

**END FOR**

}

# // Common to all, read file and store in two-dimensional array pseudocode

# String [][] ParseCourseFile(String filePath) {

# INITIALIZE two dimensional array dualArray

**INITIALIZE** stringArray

# TRY to open file at filePath

**FOR** all lines in file

**ADD** line to stringArray

**END FOR**

# IF lines of stringArray is less than one

# THROW exception

# END IF

# FOR every line in stringArray

# IF commas in this line of stringArray is less than one

# THROW exception

# END IF

# FOR every section of string in this line of stringArray delimited by commas

**ADD** this section of string to the dualArray at the row and column

# END FOR

**END FOR**

// Check prereqs

**FOR** all rows of dualArray

**FOR** all columns after index one in this row of dualArray

**SET** bool to false

**FOR** all rows of dualArray

**IF** this column zero string is equal to this row/column string

**SET** bool to true

**END IF**

**END FOR**

**IF** bool is false

**THROW** Exception

**END IF**

**END FOR**

**END FOR**

# RETURN dualArray

# }

# // Vector Create Object from two dimensional array

# vector<course> CreateCourseStructure(string[][] dualArray) {

# INITIALIZE a vector of courses

# FOR all rows of the dualArray

# INITIALIZE a new course

# SET course courseId to column zero of dualArray at this row

# SET course courseName to to column one dual array at this row

# FOR the rest of the columns in dualArray after column one

# IF dualArray at this row and this column is not an empty string

# ADD this string to the courses prereq list

# END IF

# END FOR

# ADD course to courses

# END FOR

# RETURN courses

# }

# // Vector Print Course Pseudocode

# void PrintCourseInformation(Vector<Course> courses, String courseNumber) {

# FOR all courses

# IF the courseId is the same as courseNumber

# CALL PrintCourse with this course

# END IF

# END FOR

# }

# // Hashmap Create Object from two dimensional array

# unordered\_map<string, Course> CreateCourseStructure(string[][] dualArray) {

# INITIALIZE hashmap<course> courses

# FOR every row in dualArray

# INITIALIZE new course

# SET course courseId to dualArray at index zero of this row

# SET course courseName to dual array at index one of this row

# FOR every column after index one

# IF dualArray at this row and column is not an empty string

# ADD dualArray index at this row and column to course prereqs list

# END IF

# END FOR

# ADD course to courses with courseId as key and course object as value

# END FOR

# RETURN courses

# }

# // Hashmap Print Course Pseudocode

# void PrintCourseInformation(unordered\_map<string, Course> courses, String courseNumber) {

# CALL PrintCourse with the courses course at the courseNumber key

# }

# // Binary Search Tree Create Object from two dimensional array

# BinarySearchTree CreateCourseStructure(string[][] dualArray) {

# INITIALIZE BinarySearchTree courses

# FOR every row in dualArray

# INITIALIZE new course

# SET course courseId to dualArray at index zero of this row

# SET course courseName to dual array at index one of this row

# FOR every column after index one

# IF dualArray at this row and column is not an empty string

# ADD dualArray index at this row and column to course prereqs list

# END IF

# END FOR

# CALL Insert on courses with course structure

# END FOR

# RETURN courses

# }

// Binary Search Tree instance method

void BinarySearchTree::Insert(Course course){

**IF** root is null

**SET** root equal to a new node with the course

**ELSE**

**CALL** addNode with root node and course structure

**END IF**

}

// Binary Search Tree instance method

void BinarySearchTree::addNode(Node\* node, Course course){

**IF** nodes courseid is greater than the courses course id

**IF** the nodes left node is null

**SET** nodes left node equal to a new node with the course

**ELSE**

**CALL** addNode with the nodes left node and the course

**END IF**

**ELSE**

**IF** the nodes right node is null

**SET** nodes right node equal to a new node with the course

**ELSE**

**CALL** addNode with the nodes right node and the course

**END IF**

**END IF**

}

# // Binary Search Tree Print Course Pseudocode

# void PrintCourseInformation(BinarySearchTree courses, String courseNumber) {

# INITIALIZE new treenode

**SET** new treenode equal to courses root

**WHILE** courseId on the treenode is not equal to courseNumber

**IF** the treenodes courseId is less than courseNumber

**SET** the treenode equal to the right node of the treenode

**ELSE**

**SET** the treenode equal to the left node of the treenode

**END IF**

**END WHILE**

**IF** the treenode is null

**OUTPUT** course not found

**RETURN**

**END IF**

# CALL PrintCourse with this nodes course

# }

**Section 2 Pseudocode Main Menu**

// One and only main

int main(){

// Overloaded functions will take dataStructure initialized as hashmap, vector, or BST.

**CREATE** a dataStructure of Vector, Hashmap, or BST

**CREATE** a dualArray

**CREATE** a filePath

**WHILE** user input is not nine

**OUTPUT** a list of commands to the user

**INPUT** user choice

**SWITCH** on user choice

**CASE** of user selection one

**CALL** ParseCourseFile with filePath argument returning a dual array

**SET** dualArray equal to returned array

**CALL** CreateCourseStructure with dualArray returning structure

**SET** dataStructure equal to returned structure

**CASE** of user selection two

**CALL** PrintOrderedList with dataStructure

**CASE** of user selection three

**INITIALIZE** courseId string

**CALL** GetCourseId returning string

**SET** courseId equal to returned string

**CALL** PrintCourseInformation with dataStructure and courseId

**END SWITCH**

**END WHILE**

**OUTPUT** farewell to user

**RETURN**

}

// Get course id string from user helper function

String GetCourseId(){

**INITIALIZE** string courseId

**OUTPUT** request for course id

**INPUT** users request

**SET** courseId equal to users request

**RETURN** courseId

}

**Section 3 Pseudocode for course list**

// Vector Sorting and printing, menu call to overloaded function with vector data type

void PrintOrderedList(Vector<Course> list){

**CALL** QuickSort with arguments list, zero, and list size minus one

**FOR** all courses in list

**CALL** PrintCourse with argument course

**END FOR**

}

// Vector Quicksort function overloaded for a vector of courses

# void QuickSort(Vector<Course> list, int begin, int end){

**INITIALIZE** integer mid

**IF** begin is greater or equal to end

**RETURN**

**END IF**

**CALL** Partition with arguments list, begin, and end returning mid

**CALL** QuickSort with arguments list, begin, and mid

**CALL** QuickSort with arguments list, mid plus one, and end

}

// Vector Quicksort partition function overloaded for a vector of courses

int Partition(Vector<Course> list, int begin, int end){

**SET** low equal to begin

**SET** high equal to end

**SET** middle equal to value half way between low and high

**SET** pivot equal to course at list index at middle

**SET** boolean done equal to false

**WHILE** done is false

**WHILE** courseId of the course at index low is less than pivots courseId

**INCREMENT** low

**END WHILE**

**WHILE** courseId of the course at index high is less than pivots courseId

**DECREMENT** high

**END WHILE**

**IF** low is greater than or equal to high

**SET** done equal to true

**ELSE**

**SWAP** course at list index of low with course at list index of high

**INCREMENT** low

**DECREMENT** high

**END IF**

**END WHILE**

**RETURN** high

}

// Hashmap Sorting and printing, menu call to overloaded function with Hashmap data type

# void PrintOrderedList(unordered\_list<String, Course> hash){

**INITIALIZE** array of strings named keys

**FOR** all keys in hash

**ADD** key to keys

**END FOR**

**CALL** QuickSort with argument keys, zero, and count of keys minus one

// Keys now ordered use them to print ordered list

**FOR** all strings in keys array

**CALL** PrintCourse with the course in hash at the index of the string

**END FOR**

}

// Overloaded Quicksort function for an array of strings

# void QuickSort(String[] keys, int begin, int end){

**INITIALIZE** integer mid

**IF** begin is greater or equal to end

**RETURN**

**END IF**

**CALL** Partition with arguments keys, begin, and end returning mid

**CALL** QuickSort with arguments keys, begin, and mid

**CALL** QuickSort with arguments keys, mid plus one, and end

}

// Overloaded Quicksort partition function for use with an array of strings

int Partition(String[] keys, int begin, int end){

**SET** low equal to begin

**SET** high equal to end

**SET** middle equal to value half way between low and high

**SET** pivot equal to string at keys index at middle

**SET** boolean done equal to false

**WHILE** done is false

**WHILE** keys item at index low is less than pivot

**INCREMENT** low

**END WHILE**

**WHILE** keys item at index high is less than pivot

**DECREMENT** high

**END WHILE**

**IF** low is greater than or equal to high

**SET** done equal to true

**ELSE**

**SWAP** string at index of low with string at list index of high

**INCREMENT** low

**DECREMENT** high

**END IF**

**END WHILE**

**RETURN** high

}

// BST Sorting and printing, menu call to overloaded function with BST data type

# void PrintOrderedList(BinarySearchTree tree){

**CALL** inOrder with tree root

}

// BST instance method

void BinarySearchTree::inOrder(Node\* node){

**IF** node is not null

**CALL** inOrder with nodes left node

**CALL** PrintCourse with argument of nodes course

**CALL** inOrder with nodes right node

**END IF**

}

**Section 4 Run-Time Evaluation**

Analysis: ParseCourseFile

| Code | Line Cost | Times executed # | Total Cost |
| --- | --- | --- | --- |
| INITIALIZE two dimensional array dualArray | 1 | 1 | 1 |
| INITIALIZE stringArray | 1 | 1 | 1 |
| FOR all lines in file | 1 | n | n |
| ADD line to stringArray | 1 | n | n |
| IF lines of stringArray is less than one | 1 | n | n |
| THROW exception | 1 | 0 | 0 |
| FOR every line in stringArray | 1 | n | n |
| IF commas in this line of stringArray is less than one | 1 | n | n |
| THROW exception | 1 | 0 | 0 |
| FOR every section of string in this line of stringArray delimited by commas | 1 | n | n |
| ADD this section of string to the dualArray at the row and column | 1 | n | n |
| FOR all rows of dualArray | 1 | n | n |
| FOR all columns after index one in this row of dualArray | 1 | n | n |
| SET bool to false | 1 | n | n |
| FOR all rows of dualArray | 1 | n(n-1)/2 | n^2 |
| IF this column zero string is equal to this row/column string | 1 | n(n-1)/2 | n^2 |
| SET bool to true | 1 | n(n-1)/2 | n^2 |
| IF bool is false | 1 | n(n-1)/2 | n^2 |
| THROW Exception | 1 | 0 | 0 |
| RETURN dualArray | 1 | 1 | 1 |

| Total Cost | 4n^2 + 10n + 3 |
| --- | --- |
| Runtime | O(n^2) |

Analysis: CreateCourseStructure: Vector

| Code | Line Cost | Times executed # | Total Cost |
| --- | --- | --- | --- |
| INITIALIZE a vector of courses | 1 | 1 | 1 |
| FOR all rows of the dualArray | 1 | n | n |
| INITIALIZE a new course | 1 | n | n |
| SET course courseId to column zero of dualArray at this row | 1 | n | n |
| SET course courseName to to column one dual array at this row | 1 | n | n |
| FOR the rest of the columns in dualArray after column one | 1 | n | n |
| IF dualArray at this row and this column is not an empty string | 1 | n | n |
| ADD this string to the courses prereq list | 1 | n | n |
| ADD course to courses | 1 | n | n |
| RETURN courses | 1 | 1 | 1 |

| Total Cost | 8n + 2 |
| --- | --- |
| Runtime | O(n) |

Analysis: CreateCourseStructure: Hashmap

| Code | Line Cost | Times executed # | Total Cost |
| --- | --- | --- | --- |
| INITIALIZE hashmap<course> courses | 1 | 1 | 1 |
| FOR every row in dualArray | 1 | n | n |
| INITIALIZE new course | 1 | n | n |
| SET course courseId to dualArray at index zero of this row | 1 | n | n |
| SET course courseName to dual array at index one of this row | 1 | n | n |
| FOR every column after index one | 1 | n | n |
| IF dualArray at this row and column is not an empty string | 1 | n | n |
| ADD dualArray index at this row and column to course prereqs list | 1 | n | n |
| ADD course to courses with courseId as key and course object as value | 1 | n | n |
| RETURN courses | 1 | 1 | 1 |

| Total Cost | 8n + 2 |
| --- | --- |
| Runtime | O(n) |

Analysis: CreateCourseStructure: Binary Search Tree

| Code | Line Cost | Times executed # | Total Cost |
| --- | --- | --- | --- |
| INITIALIZE BinarySearchTree courses | 1 | 1 | 1 |
| FOR every row in dualArray | 1 | n | n |
| INITIALIZE new course | 1 | n | n |
| SET course courseId to dualArray at index zero of this row | 1 | n | n |
| SET course courseName to dual array at index one of this row | 1 | n | n |
| FOR every column after index one | 1 | n | n |
| IF dualArray at this row and column is not an empty string | 1 | n | n |
| ADD dualArray index at this row and column to course prereqs list | 1 | n | n |
| CALL Insert on courses with course structure | 1 | n | n |
| RETURN courses | 1 | 1 | 1 |

| Total Cost | 8n + 2 |
| --- | --- |
| Runtime | O(n) |

**Section 5 Advantages and Disadvantages**

The vector data structure is very useful for inserting and removing items at the end of the list, accessing items by index, and traversing through the entire structure using a for loop. Since a positional index is used, accessing items by index is O(1). However, this data structure is not ideal when items need to be removed or inserted from random index locations in the list, this is because when an item is inserted or removed, all items after it must be moved forward or backward by one location. In a very large list, this can be a very timely requirement, making other structures more important in these situations.

The hashmap solves the problem of insertions and deletions by placing an item inside the structure by hashing a key. This allows the hashmap to find an index that is unique to this item. If two or more items would hash to the same index, the internal implementation of the hash map will use a link list or some other method to handle the collision. Due to the hashmaps implementation, insertions and removals will have O(1) speed making them much better for these operations than the vector. The hashmap has a disadvantage of not having a positional index. Items will need to be retrieved with their key. If a search operation is used because the key is not known, the advantage of the hashmap is lost and it will have an O(n) search speed.

Binary Search Trees handle insertions and removals by storing nodes in a hierarchy, each node having a left and right node. The left node must be smaller, while the right node should be larger. If the tree is balanced, insertion and removal should work at a speed of Θ(log n). This is because when the tree is balanced it should have a maximum amount of comparisons of its height plus one. The tree also has an advantage of being ordered and by using an in-order algorithm, can traverse recursively through the tree visiting nodes in order. The main disadvantage to this structure is the need to balance the tree. If items were inserted into this tree in order, the tree would resemble a linked list, and lose its benefits.

**Section 6 Recommendation**

The algorithm requirements for this project are to print the computer science courses in alphanumerical order and quickly search the structure to print a course that the user requests. Due to these requirements, the data structure should be a vector. The reason the vector is chosen is due to the fact that once the list is read into memory, no further insertions are made. This means that once the list is sorted, it will not need to be sorted again. The quicksort algorithm provides a very efficient solution that will only need to be utilized once after the initial creation of the list. Once the list is made and sorted, calls to print the list to the console will require only an iteration through the list. Next, a binary search algorithm can be used to find specific records. With the vector already sorted, the time to find a record can be greatly reduced. So while the worst case time for searching a vector linearly is O(n), the use of the binary search algorithm can reduce the speed to O(log n). The vector, in conjunction with quicksort and binary search, will accomplish the tasks required by the program while dodging the complexities associated with other structures, such as tree balancing and hashmap sorting.