CS 300 Pseudocode Document

Example Function Signatures
Below is an example of a function signature that you can use as a guide to help address the program requirements using each data structure for the milestones. The pseudocode for finding and printing course information is also given below and depicted in bold to help you get started. The provided pseudocode is for a vector data structure, so you may use this pseudocode in your first milestone as is. The hash table and tree structures are also shown below. But these structures are left for you to do in future milestones.

//Vector - Milestone 1

```
void searchCourse(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
}
```

1. Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for file format errors.
2. Design pseudocode to show how to create course objects and store them in the appropriate data structure.

```
Define a Course Class

String courseNumber

String title

    Vector<String> prerequisites


Void OpenFile&Store(file) {

    Open file

    Check if file was opened correctly

    Vector<Course> courses // a vector of course objects
    While there is a line in file
        Split the line by comas into tokens
            If num tokens < 2
                print error // data should have a course number and title

    Create a new course object
    Set course number = token[0]
    Set title = token[1]

    If num tokens > 2
    for each token from index 2 to end
    add token to prerequisites vector in course object
        Add course to courses vector
    End while loop

    Close file

//The code below checks if all prerequisites are in the data
Declare a string vector of course numbers
For each course in courses
    add course.coursenumber to coursenumbers

    //checks if prerequisites are in course numbers vector
    Make a list for incorrect prerequisites to be outputted later
    For each course in courses{
        if the course has prerequisites
            for each prerequisite course. prerequisites
            declare string to store current prerequisite
                for each courseNum in courseNumbers
                    If prerequisite == courseNum
                        continue
                    Else if prerequisite != courseNum
                        Add current prerequisite to list for output
                    Else
                        error message
        else the course has no prerequisites
    }
}
```

3. Design pseudocode that will search the data structure for a specific course and print out course information and prerequisites

```
void searchCourse(Vector of courses, String courseNumber) {
    for course in courses
        if the course.courseNumber == courseNumber parameter
            print out course information
            for each prerequisite in the course
                print the prerequisites
}
```

//Hash Table - Milestone 2

```
Define a Course Class {
    String courseNumber
    String title
    Vector<String> prerequisites
}

//Opens a file and stores it in a hash table. This code does not handle collisions
Void OpenFile&Store(file) {
    Open file
    Check if file was opened correctly

    Use an unOrdered map for hash table called courses
    //Will use courseNumber for key using string hashing
    Declare a string vector of course numbers // will be used to verify prerequisites


    While there is a line in the file {
        Split the line into tokens by coma
        If num tokens < 2 {
            output error // all data should have at least course id and name
        }
```

```
                Make a new course object with the data

                Add the course number or token[0] to courseNumbers vector

                If num tokens > 2 {

        Add prerequisites to vector

                }

                Add course to hash table using hash function on courseNumber

        }

        Close file

}


//The code below verifies if the prerequisites are on another line in the file.

// It could be implemented in the open file and store method


// all prerequisites should return a key value for another course

For each course. prerequisite in courses

        Using a hash table search function search for courseNumber not for another prerequisite

        If course ID is found

continue

        If it is not found

Output some error or add it to a list


// will print the entire hash table

Void printAll(HashTable) {

For loop to loop through hash table size

if bucket is in use
```

```
print out course information
//This code does not handle collisions
//However, if a chain is implemented, we can use a while loop for the linked list
while current != nullptr
output course information
set current to next

}

// Will print one courses information given a course ID
void searchCourse(HashTable<Course> courses, String courseNumber) {
        Use hash for key with courseNumber
        Use search function with key
        //Find node using key
        If empty
                Output course was not found
        If found
                Output course information
}
```

//Binary Search Tree – Milestone 3

```
Define a course class or struc {

    String courseNumber

    String title

    Vector<String> prerequisites

}


Define a Node struc {

    A left and right pointer initialized to nullptr
    A course initialized to input
    // So, a call Node(course) will make a new node with a course and two pointers

}


Void Open&ReadFile(file) {

    Open file
    Check if file was opened correctly

    Declare a string vector of course numbers // will be used to verify prerequisites


    While there is a line in the file {

        Split the line into tokens by coma

        If num tokens < 2 {

            output error // all data should have at least course id and name

        }

        Make a new course object with the data

        Add the course number or token[0] to courseNumbers vector

        If num tokens > 2 {
                Add prerequisites to vector

            }

            Add data to BST sorting by the courseNumbers lexicographically

            //Smaller course numbers to the left and larger to the right

        }

        Close file

}


// To ensure all the prerequisites are on another line in the file
// we can use the seachCourse function below
// The following code could be put into main or the Open&ReadFile method above
// all prerequisites when searched should be found elsewhere in the tree



//Loop the prerequisites in the vector we used to store course numbers

For each course.prerequisite in courses

        Use the search function for each course.prerequisite

            If course ID is found

                continue

            If it is not found

                Output some error or add it to a list




void searchCourse(Tree<Course> courses, String courseNumber) {
        Make a currentNode pointer and set it to the root

        Loop down the tree until you find a match

        While current pointer != nullptr {

            If match is found return the current course
        Else if the course is smaller than the current course node go left in the tree

        Else if course is larger than the current course node goes right in the tree

    }

    If nothing was found return an empty bid

}

//This method will print all data in the BST using Inorder traversal

// A print function could be made to search and print one course at a time using the search function above

//However, this function will recursively print the entire tree

Make a public InOrder function and pass root to InOrder

void inOrder(Node* node) {

If node == nullptr return

InOrder(node->left)

Output course information using the current node

Inorder(node->right)

}
```

```
}
```

Menu Pseudocode (Project One)

```
//The code for the menu could put into another method but putting it in the main method makes sense
int main(int argc, char* argv[]) {
Process command line arguments including the csv file path
Int choice = 0; // var to hold user menu choice
While (choice != 9) {
Output menu to user
Read in their input for choice

//Use a case switch for the menu
Case 1: // Load the file into the data structure
Load data into data structure
break
Case 2: //Print alphanumerically ordered list of all the courses
Print the data structure in alphanumeric order using the methods below
break
Case 3: //Print the course title and prerequisites for an individual course
Ask the user for the CourseNumber or input to find the course object
Output the course information using a search method
break
                    //Case 9: Exit the program

}
}
```

Print alphanumeric pseudocode (Project One)

The following three functions will print all of the courses ordered alphanumerically from low to high. These methods should be defined in the data structure class.

```
// This function will print a vector data structure alphanumerically
// This function requires the standard c++ library is imported using the c++ sort function and the < operator has been overloaded in the course object
Void PrintVectorAlphanumeric() {

// courses is a vector of course objects

Sort(courses.begin(), courses.end())



For all courses in courses vector

print courses sorted alphanumerically from low to high

}
```

```
//This function will print a hash table data structure alphanumerically

// This function requires the standard c++ library is imported using the c++ sort function and the < operator has been overloaded in the course object
Void PrintHashTableAlphanumeric() {

Declare a vector "sortedKeys" to store hash table keys to sort

Loop through the hash table adding every key to sortedKeys


//Sort the sortedKeys vector using the c++ sorting function

Sort(sortedKeys.begin(), sortedKeys.end())


//Loop through the sortedKeys vector using each key to access the hash table and print the course. This will result in courses being printed in alphanumeric order.

For each key in sortedKeys

print individual course information

}
```

```
//This function will print a BST data structure alphanumerically

// From the implementation above of the BST it is already ordered alphanumerically or lexicographically

// Because it is already sorted, we can use inOrder traversal and print the BST

Void PrintBSTAlphanumeric() {

InOrder(root) // assuming an inOrder function has already been created, printing course info
        between the left and right trees

}
```

Runtime Analysis (Project One)

Example Runtime Analysis
When you are ready to analyze the runtime for the Project One data structures for which you created the pseudocode, use the example chart below to support your work. This particular example is for printing course information when using the vector data structure. As a reminder, this is the same pairing that was bolded in the pseudocode from the first part of this document. The example only covers the search function for the vector structure. You do not have to complete your runtime analysis until Project One. However, working on your analysis now may help you understand the changes as you complete the milestones. Do not forget to include your charts in Project One. You will submit Project One in Module Six.

Code	Line Cost	# Times Executes	Total Cost
for all courses	1	n	n
if the course is the same as courseNumber	1	n	n
for each prerequisite of the course	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)

Let k = the number of tokens per line (will be a very small constant if we take the average a course will only have a few prerequisites)

n = number of courses

Vector::OpenFile&Store	Line Cost	# Times Executes	Total Cost
Define a course class	1	1	1
Open file	1	1	1
Check file	1	1	1
Declare courses vector	1	1	1
While there is line in file	1	n	n

Split the line into tokens by comas	1	k	k
If numTokens < 2	1	n	n
Create a new course object	1	n	n
If numTokens > 2	1	n	n
Add prerequisites to course object vector	1	k	k
Add course to main courses vector	1	n	n
Close file	1	1	1
		Total Cost	5n + 2k + 5
		Runtime	O(n)

HashTable::OpenFile&Store	**Line Cost**	**# Times Executes**	**Total Cost**
Define a course Class	1	1	1
Open file	1	1	1
Check file	1	1	1
Create un-Ordered map	1	1	1
Declare course numbers vector	1	1	1
While there is line in file	1	n	n
Split the line into tokens by comas	1	k	k
If numToekns < 2	1	n	n
Make new course object	1	n	n
Add course number to vector	1	n	n
If numTokens > 2	1	n	n
Add prerequisites to course object vector	1	k	k
Add course to hashtable	1	n	n
Close file	1	1	1
		Total Cost	6n + 2k + 6
		Runtime	O(n)

BinarySearchTree::Open&ReadFile	**Line Cost**	**# Times Executes**	**Total Cost**

Declare course object	1	1	1
Define a node struc with pointers	1	1	1
Open file	1	1	1
Check File	1	1	1
Declare a string vector	1	1	1
While there is line in file	1	n	n
Split lines into tokens	1	k	k
If numTokens < 2	1	n	n
Make a new course object	1	n	n
Add course number to vector	1	n	n
If numTokens > 2	1	n	n
Add prerequisites to course object vector	1	k	k
Add data to BST sorting lexicographically (using inOrder)	1	n	N log n
Close file	1	1	1
		Total Cost	(6n + 2k + 6) + n * log(n)
		Runtime	O(n log n)

Each insertion on average in a balanced BST is O(log n) and we are inserting n courses. Therefore, the overall runtime is O(n log n). Most of the time a BST will be balanced however if it becomes unbalanced each insertion will take O(n) which if we insert n times will become O(n^2).

Advantages and Disadvantages

These three data structures all have their pros and cons. None of them is the best or worst, they are just all best used in different situations. They all need to loop through the entire CSV file and create new course objects leading to O(n) time complexity for a vector and hash table while a BST is O (n log n) because it must traverse the tree and the CSV file. The main difference is the speed at which we can sort the data and the amount of memory each data structure takes up.

The vector is the clear winner in terms of space complexity. For hash tables and BSTs, we will need to allocate space for buckets and pointers. A vector also has an insertion time of O (1) when it is unordered. However, sorting the vector can take up more time. Vectors store data continuously in memory so adding more courses or deleting them could also become a problem or costly for memory and time. Vectors are great for simple and smaller datasets and for quick lookups.

A hash table is exceptionally good at operating on data or in this case courses. Insertion, deletion, and search can all be done in constant time or O(1). This is because we can use the course number directly as a key for the data structure. The main downside of a hash table is its significant memory usage. We will need to allocate memory for buckets and potentially linked lists to handle collisions. The hash table is also inherently un-ordered because of the hashing function. So, the keys would have to be sorted afterward and then used to access the hash table in the correct order. Doing this will increase the overall runtime and memory requirements of the program. Hash tables are excellent if order does not matter and you need to frequently insert, delete, or lookup data. In our situation order does matter and we will not be adding and removing courses frequently.

A binary search tree or BST has the advantage of being sorted upon insertion using in-order traversal. Because of this insertion, deletion, and lookup can be accomplished in O(log n) which is not constant time like a hash table but still fast. The main downside of a BST similar to a hash table is its memory overhead. Each node will need to hold a course object and two pointers for the left and right child nodes. The time complexity of adding courses to a BST is O (n log n) because we need to loop the CSV with n courses and then traverse the tree for each course. It is also important to unsort the data being inputted so the BST does not become unbalanced. If the BST becomes unbalanced then insertion, deletion, and lookup will become O(n). Binary search trees are great for keeping data sorted and performing quick queries on that data.

Recommendation

The data set we are working with is not exceptionally large. There will probably not be thousands or millions of courses at ABCU. The University of California, Los Angeles (UCLA) has 5,000 courses total and it is one of the largest universities in the United States (USnews, 2023). Considering we are only storing the ABCU computer science courses our data set may be in the hundreds in the worst case. The CSV file for project two and the given course dataset info are only about 8 lines long.

The vector data structure is likely the best for our needs here. Adding the data to the vector is O(n) runtime which is the same as a hash table and faster than a BST. However, the vector takes up considerably less space than both a BST and hash table. On top of that courses at ABCU are unlikely to be frequently removed or added. Even if they are, the size of the data set is small enough that it is not a significant problem. If courses need to be frequently removed and added, then a hash table would be better.

The main downside of the vector for our needs is printing the courses alphanumerically. The C++ standard libraries sort function has an O(n log n) runtime which is not great but considering the size of our dataset it is not a big problem. After the vector is sorted, we can use binary search to quickly search for individual courses that will have a runtime of O(log n). Or considering the size of our dataset a simple for loop would work fine and would have a runtime of O(n). If memory is not a consideration, then a hash table should be used. It would easily allow operations on courses with faster runtimes. And we can add/delete courses easier.

References

USnews. (2023). How does University of California--Los Angeles rank among America’s best colleges? University of California--Los Angeles. https://www.usnews.com/best-colleges/ucla-1315