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).

```
OpenFile()
    If return value is -1
        Print file not found
Else
    File is found
    While it is not EOF
        Read every line
            If line < 2 tokens
                Return ERROR
            Else read tokens
Close file

// 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 printSampleSchedule(Vector<Course> courses) {
    for all courses
        print course name
    if course has prerequisites
        for each prerequisite
            print prerequisite

}

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
}
```

```
void addPrerequisites(Vector<Course> courses, String courseNumber,
Vector<String> prerequisites) {
	for each course in courses:
		if course.courseNumber == courseNumber:
			course.prerequisites.addAll(prerequisites)
}


// Hashtable pseudocode
int numPrerequisiteCourses(Hashtable<Course> courses) {
	totalPrerequisites = Hashtable[c]
	for each prerequisite p in totalPrerequisites
		add prerequisites of Hashtable[p] to totalPrerequisites
	print number of totalPrerequisites

}

void printSampleSchedule(Hashtable<Course> courses) {
	for each (key, value) course in courses
	print key course name
	if value has prerequisites
		for each prerequisite in course
			print prerequisite

}

void printCourseInformation(Hashtable<Course> courses, String
courseNumber) {
   for all courses:
	if courseNumber == courses
		print out the course information of courseNumber
		for each prerequisite of Hashtable[course]
			print prerequisite course info

}

void addPrerequisites(Hashtable<Course> courses, String courseNumber,
Vector<String> prerequisites) {
	courses[courseNumber].prerequisites.addAll(prerequisites)
}


// Tree pseudocode
int numPrerequisiteCourses(Tree<Course> courses) {
	totalPrerequisites = prerequisites of course c
```

```
	recurse each prerequisite p in totalPrerequisites
		add prerequisites of p to totalPrerequisites
	return the size of totalPrerequisites }

}

void printSampleSchedule(Tree<Course> courses) {
	print course name
	if course has prerequisites
		recurse each prerequisite p of the course
			print p

}
void printCourseInformation(Tree<Course> courses, String courseNumber)
{
locate courseNumber
if courseNumber is found
	print out the course information of courseNumber
	recurse each prerequisite of courseNumber
		print the prerequisite course information
}

void addPrerequisites(Tree<Course> courses, String courseNumber,
Vector<String> prerequisites) {
	courses.find(courseNumber).prerequisites.addAll(prerequisites)
}
```

2.) Creating a menu

```
Void Menu(Vector<Course>vectorCourses, Hashtable<Course>
hashtableCourses, Tree<Course> treeCourses)


	userInput = 0

while userInput != 4:

	cout << ("-----Menu-----")

	cout << ("1. Load Data Struct")

	cout << ("2. Print Course List")
```

snhu

```
    cout << ("3. Print Course")

    cout << ("4. Exit")

    cin >> userInput

case 1:

    loadCourses(vectorCourses, hashtableCourses, treeCourses)

    break;

case 2:

    printAlphaNumericOrder(vectorCourses)

    break;

case 3:

    printCourse(vectorCourses, hashtableCourses, treeCourses)

    break;

case 4:

    cout << "Good-bye"

    END

    Break;

Default:

    Cout << ("Incorrect Input")

    Break;
```

3.) Alphanumeric order pseudocode for all 3 data structures

Vector:

```
Sort(vectorCourses.begin(), vectorCourses.end(),
compareByCourseNumber)
```

Hashtable:

```
Vector<String> courseIndex

For each key in hashtableCourses:

      courseIndex.push_back(key)

sort(courseIndex.begin(), courseIndex.end(), compareByCourseNumber)

for each key in courseIndex:

      print hashtableCourses[key]
```

Tree:

```
Vector<Course> inOrderCourses

inOrderTraverse(treeCourses.root, inOrderCourses)

sort(inOrderCourses.begin(), inOrderCourses.end(),
compareByCourseNumber)

for each course in inOrderCourses:

      print course
```

Runtime Evaluation Vector

Code	**Cost per line**	**Times executed**	**Big O value**
Reading a file	1	n	O(n)
Creating course objects	1	n	O(n)
Sorting courses	1	n	0(n log n)

Runtime Evaluation HashTable

Code	Cost per line	Times executed	Big O value
Reading a file	1	n	O(n)
Creating course objects	1	1	O(1)
Sorting keys	1	n	θ(n log n)

Runtime Evaluation Tree

Code	Cost per line	Times executed	Big O value
Reading a file	1	n	O(n)
Creating course objects	Log n	n	O(log n)
Sorting courses	1	n	θ(n)

Advantages and Disadvantages

Vectors are great for their simplicity and easy access of indexes however the searching can take quite a large amount of time if your vector has a lot of data.

- Simple implementation
- Efficient for small data sets

Hashtables are great for printing courses on their own so long as they don't have too much data and do not need sorting for such, however hashtables use a lot more memory.

- Quick retrieval
- Efficient for small data sets
- Additional memory

Binary Seach Trees are great for searching for courses fast and its traversal is good for printing in order n. However the sorting process can also be seen as a burden since it's not one step typically requiring ways to check if a number is less than or greater than so it can traverse.

- Efficient for search and sorted data
- Efficient for large data sets
- Additional Memory
- Slow insertions

Recommendation for project 2:

~~I think based on the big O analysis I would recommend NOT using a binary tree and possibly using a hashtable for its fast access to the courses if keeping a hashtable quick and concise we may not use that much memory.~~

After further investigation a BST could pose as a useful data structure that would benefit the advisors for sorting classes in alphanumeric order. The reason being that a Binary Seach Tree maintains a sorted order when performing inOrder traversal. This keeps data sorted such as courseNumbers. The runtime complexity of this would be equal to O(n).