CS 300 Project One

**Pseudocode**

1. **Resubmit pseudocode from previous pseudocode assignments and update as necessary**.

(Attached)

1. **Create pseudocode for a menu**.

CREATE function named displayMenu()

PRINT "1. Load file data into the data structure"

PRINT "2. Print an alphanumerically ordered list of all courses"

PRINT "3. Print the course title and prerequisites for any individual course"

PRINT "9. Exit the program"

PRINT "Enter your choice:"

CREATE function named MenuOption(choice)

IF choice EQUALS 1

CALL readAndParseFile()

ELSE IF choice EQUALS 2

CALL printCourses()

ELSE IF choice EQUALS 3

PRINT "Enter course number:"

DECLARE courseNumber

GET INPUT courseNumber

CALL printCourseInfo(courseNumber)

ELSE IF choice EQUALS 9

PRINT "Goodbye"

// in main

WHILE choice IS NOT 9

CALL displayMenu()

GET INPUT choice

CALL MenuOption(choice)

1. **Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.**

**For vector:**

CREATE function named sortCourses(vector<Course> &courses)

SORT courses BY courseNumber

CREATE function named printCourses(vector<Course> &courses)

CALL sortCourses(courses)

FOR EACH course IN courses

PRINT course.courseNumber + " - " + course.title

**For hash table:**

CREATE function named printCourses(hashTable<string, Course> &courses)

DECLARE vector<string> courseNumbers

FOR EACH courseNumber IN courses

ADD courseNumber TO courseNumbers

SORT courseNumbers

FOR EACH courseNumber IN courseNumbers

PRINT courses[courseNumber].courseNumber + " - " + courses[courseNumber].title

**For binary search tree:**

CREATE function named sortCourses(vector<Course> &courses)

SORT courses BY courseNumber

CREATE function named traverseInOrder(node, vector<Course> &courses)

IF node IS NOT null

CALL traverseInOrder(node.left, courses)

ADD node.course TO courses

CALL traverseInOrder(node.right, courses)

CREATE function named printCourses()

DECLARE vector<Course> courses

CALL traverseInOrder(root, courses)

CALL sortCourses(courses)

FOR EACH course IN courses

PRINT course.courseNumber + " - " + course.title

**Evaluation**

1. **Evaluate the run time and memory of data structures that could be used to address the requirements**. In previous assignments, you created pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors and to show how to create course objects so that one course object holds data from a single line from the input file.
   1. Using the pseudocode you wrote for the previous assignments, analyze the worst-case running time of each, reading the file and creating course objects, which will be the Big O value. This analysis should not include the pseudocode written for the menu or the search/print functions Print Course List (Option 2) above. To complete this part of the project, do the following actions:  
      1. Specify the cost per line of code and the number of times the line will execute. Assume there are n courses stored in the data structure.
      2. Assume the cost for a line to execute is 1 unless it is calling a function, in which case the cost will be the running time of that function.

Vector:

| CREATE function named addCourse(courseNumber, title, vector<prerequisites>) |  |
| --- | --- |
| SET course TO new Course | 1 |
| course.courseNumber = courseNumber | 1 |
| course.title = title | 1 |
| course.prerequisites = prerequisites | 1 |
| PUSH BACK course in Course | 1 |
|  | **= 1** |
| CREATE function named isValidFormat |  |
| DECLARE courseNumber, title, prereq | 1 |
| GET LINE until , AS courseNumber | 1 |
| GET LINE until , AS title | 1 |
| IF !(courseNumber) OR !(title) | 1 |
| RETURN False | 1 |
| WHILE GET LINE until , AS prereq | 1 |
| IF prereq NOT IN Courses | n |
| RETURN false | 1 |
| RETURN true | 1 |
|  | **=n** |
| CREATE function named readAndParseFile() |  |
| DECLARE courseNumber, title, prereq | 3 |
| GET LINE | 1 |
| WHILE GET LINE from file IS line | n |
| IF NOT isValidFormat | n |
| continue | 1 |
| GET LINE until , AS courseNumber | n |
| GET LINE until , AS title | n |
| WHILE GET LINE until , AS prereq | n |
| PUSH BACK prereq in <vector>prerequisite | n |
|  | **= n^2** |
| TOTAL | **n^2** |

Hash Table:

| CREATE function named insertCourse(courseNumber, title, vector<prerequisites>) |  |
| --- | --- |
| SET key TO hash(courseNumber) | 1 |
| SET current TO coursesTable[key] | 1 |
| IF current.key EQUALS UINT\_MAX | 1 |
| SET current.courseNumber TO courseNumber | 1 |
| SET current.title TO title | 1 |
| SET current.prerequisites TO vector<prerequisites> | 1 |
| SET current.next TO null | 1 |
| ELSE SET previous TO current | 1 |
| WHILE current IS NOT null | n |
| SET previous TO current | 1 |
| SET current TO current.next | 1 |
| SET previous.next TO new | 1 |
|  | **= n** |
| CREATE function named isValidFormat |  |
| DECLARE courseNumber, title, prereq | 3 |
| GET LINE until , AS courseNumber | 1 |
| GET LINE until , AS title | 1 |
| IF !(courseNumber) OR !(title) | 1 |
| RETURN False | 1 |
| WHILE GET LINE until , AS prereq | 1 |
| IF prereq NOT IN coursesTable | 1 |
| RETURN false | 1 |
| RETURN true | 1 |
|  | **= 1** |
| CREATE function named readAndParseFile() |  |
| DECLARE courseNumber, title, prereq | 3 |
| GET LINE | 1 |
| WHILE GET LINE from file IS line | n |
| IF NOT isValidFormat | n |
| CONTINUE | 1 |
| GET LINE until , AS courseNumber | n |
| GET LINE until , AS title | n |
| CREATE vector<string> prerequisites | 1 |
| WHILE GET LINE until , AS prereq | n |
| PUSH BACK prereq IN prerequisites | n |
| CALL addCourse(courseNumber, title, prerequisites) | n |
|  | **= n** |
| TOTAL | **= n** |

Binary Search Tree:

| CREATE function named readAndParseFile(fileName) |  |
| --- | --- |
| OPEN fileName | 1 |
| DECLARE line | 1 |
| WHILE GET LINE from file IS line | n |
| IF NOT isValidFormat(line) | n \* (**log n)** |
| CONTINUE | 1 |
| DECLARE courseNumber, title, vector<string> prerequisites | 3n |
| SPLIT line BY ',' INTO courseNumber, title, prerequisites | n |
| CALL addCourse(courseNumber, title, prerequisites) | n \* **log n** |
| CLOSE file | 1 |
|  | **=n\*log n** |
| CREATE function named isValidFormat(line) |  |
| DECLARE courseNumber, title, prereq | 3 |
| SPLIT line BY ',' INTO courseNumber, title, prereq | 1 |
| IF !(courseNumber) OR !(title) | 1 |
| RETURN False | 1 |
| FOR EACH prereq IN prereqs | 1 |
| IF NOT searchCourse(root, prereq) | log n |
| RETURN False | 1 |
| RETURN True | 1 |
|  | =**log n** |
| CREATE function named addCourse(courseNumber, title, vector<prerequisites>) |  |
| SET newCourse TO new Course | 1 |
| newCourse.courseNumber = courseNumber | 1 |
| newCourse.title = title | 1 |
| newCourse.prerequisites = prerequisites | 1 |
| CALL insertNode(root, newCourse) | log n |
|  | =**log n** |
| CREATE function named insertNode(node, course) |  |
| IF node IS null | 1 |
| SET node TO new Node | 1 |
| node.course = course | 1 |
| node.left = null | 1 |
| node.right = null | 1 |
| ELSE IF course.courseNumber < node.course.courseNumber | log n |
| CALL insertNode(node.left, course) | log n |
| ELSE CALL insertNode(node.right, course) | log n |
|  | =**log n** |
| CREATE function named searchCourse(node, courseNumber) |  |
| IF node IS null | 1 |
| RETURN null | 1 |
| ELSE IF courseNumber EQUALS node.course.courseNumber | log n |
| RETURN node.course | 1 |
| ELSE IF courseNumber < node.course.courseNumber | log n |
| RETURN searchCourse(node.left, courseNumber) | log n |
| ELSE | log n |
| RETURN searchCourse(node.right, courseNumber) | log n |
|  | =**log n** |
| TOTAL | **n\*log n** |

1. Based on the advisor’s requirements, analyze each of the vector, hash table, and tree data structures. **Explain the advantages and disadvantages of each structure in your evaluation.**

These three data structures each present trade-offs. The vector, while simplest to implement and read, suffers from O(n²) runtime complexity due to nested loops in the readAndParse function, making it the least efficient. However, its contiguous memory storage and dynamic resizing offer better memory management than hash tables or binary search trees. Vectors also perform poorly with frequent middle insertions/deletions (O(n)). The hash table offers the best runtime complexity (O(n)) and efficient insertion/deletion (O(1)), but requires careful management to minimize collisions, increasing both complexity and memory overhead. Finally, the binary search tree, with O(n log n) runtime complexity, enables faster searching and traversal due to its organized structure. However, the added pointers increase memory usage compared to vectors. Both hash tables and binary search trees are more complex to implement than vectors.

1. Now that you have analyzed all three data structures, **make a recommendation for which data structure you plan to use in your code**. Provide justification for your recommendation based on the Big O analysis results and your analysis of the three data structures.

For scalability, the balanced binary search tree is the most suitable data structure for this course dataset. While vectors offer better memory efficiency and hash tables provide faster lookups, the binary search tree offers a crucial balance. Its logarithmic time complexity for search operations is significantly faster than the linear search required with vectors, a difference that will become increasingly pronounced as the dataset grows. Although vectors are more memory-efficient and hash tables offer faster average lookup times, the binary search tree provides a strong middle ground, optimizing for search speed without excessive memory overhead. Given the anticipated frequency of search operations, the balanced binary search tree's performance advantage makes it the ideal choice for future scalability.