CS-300 Project One

CS-300: 23EW1

Graham Hedrick

**Vector Pseudocode**

Class: Course

Declare:

courseNum;

courseName;

prereqs;

Constructor (num, name, prereqs):

this.num = num;

this.name = name;

this.prereqs = prereqs;

Function: loadCourses(filename):

courses = empty vector for Course objects;

file = open(filename, "r");

For: each line in file:

courseData = split line by comma;

courseNum = courseData[0].trim();

courseName = courseData[1].trim();

prereqs = empty list;

If: length of courseData > 2:

prereqsData = split courseData[2] by comma;

For: each prereq in prereqsData:

prereqs.append(prereq.trim());

course = new Course(courseNum, courseName, prereqs):

courses.append(course);

course = new Course(courseNum, courseName, prereqs):

courses.append(course);

Close file;

IF: FileNotFound:

Output "ERROR: File not found or cannot be read!”;

Return courses;

Function: PrintCourseInfo (courses, courseNum):

For: each course in courses:

If course.num equals courseNum:

Output "Course Number:", course.num;

Output "Course Title:", course.name;

Output "Prerequisites:";

For each prereq in course.prereqs:

Output " ", prereq;

Return;

Print "Course not found!";

**Hash Table Pseudocode**

Function: validInput(line):

tokens = split(line, “,”);

If length(tokens) < 2:

Return: False;

For: ‘i’ from 2 to total length(tokens) -1:

If: courseInvalid(tokens[i]):

Return: False;

Return True;

Function: courseValid(courseNum):

Return: hashTable.node(courseNum);

Function: createCourse(line);

Tokens = split(line, “,”);

courseNum = tokens[0];

courseName = tokens[1];

prereqs = [];

For: ‘i’ from 2 to total length(tokens) -1:

prereqs.append(tokens[i]);

Return: Course(courseNum, courseName, prereqs);

Function: printCourseInfo(course):

Output: “Course Number: ”, course.courseNum;

Output: “Course Name: ”, course.courseName;

If: length(course.prereqs) > 0:

Output: “Prerequisites: ”;

For: ‘i’ from 0 to total length(course.prereqs) – 1:

Output: course.prereqs[i];

Output: prereqs;

Function: loadCourse(filename):

File = open(filename, “read”);

For: each line in file:

If inputValid(line):

course = createCourse (line);

hashTable.insert (course.courseNum, course);

Function: main():

hashTable = newHashTable();

loadCourse(“file.csv”);

For: each key value in hashTable:

Output: printCourseInfo (key.node);

**Binary Search Tree Pseudocode**

Declare LoadCourseInfo:

Declare course as data structure;

Declare ‘courseFile’ as std::ifstream;

Declare ‘line’ as String;

Declare courseMap as <String, Course>;

Declare CreateCourse (courseNum, courseName, prereqs):

Declare newCourse as Course;

Set newCourse.num = courseNum;

Set newCourse.name = courseName;

Set newCourse.prereqs = prereqs;

Return newCourse;

Open courseFile using a path to the file:

If not courseFile.isOpen()

Then output "Error opening file.";

While not courseFile.end():

Read ‘line’ from courseFile;

Declare token = Split(line, ",");

If token.size() < 2:

Then output "Invalid line format.”;

Declare courseNum = token[0];

Declare courseName = token[1];

Declare prereqs = [];

For ‘i’ from 2 to token.size() – 1:

prereqs.push(tokens[i]);

Declare newCourse = CreateCourse(courseNum, courseName, prereqs);

course.push(newCourse);

courseMap[courseNum] = newCourse;

For each course in ‘course’:

For each prerequisite in course.prereqs:

If not courseMap.contains(prereq):

Output "Invalid prerequisite:", prereq, "for course:", course.num;

For each course in structure ‘course’:

Output "Course:", course.num, " | ", course.name;

If course.prereqs.size() > 0:

output "Prerequisites:";

For each prereq in course.prereqs:

output prereq;

**Menu Pseudocode**

Function: MainMenu():

While: True:

Output "--------- MENU ---------"

Output "1. Load Data Structure";

Output "2. Print Course List";

Output "3. Print Course";

Output "4. Exit";

choice = user input "Enter your choice: ";

If: choice == 1:

filename = input "Enter filename: "

LoadDataStructure(filename)

ElseIf: choice == 2:

PrintCourseList();

ElseIf: choice == 3:

courseNum = user input "Enter course number: "

PrintCourse(courseNum);

ElseIf: choice == 4:

Output "Exiting program...";

Exit program;

Else: Output "Invalid choice. Please try again.";

MainMenu();

**Alphanumeric Sort Pseudocode**

**(Vector)**

Function: DisplayCourses (Courses):

sortedCourses = SortAlphaNum (Courses);

For: each course in sortedCourses:

Output course.number, course.title;

**(Hash Table)**

Function: DisplayCourses(Courses):

listCourses = empty list;

For: each key in Courses:

Append Courses[key] to listCourses;

sortedCourses = SortAlphaNum(listCourses);

For: each course in sortedCourses:

Output course.number, course.title;

**(Binary Search Tree)**

Function: DisplayCourses(node):

If: node is not null:

DisplayCourses(node.left);

Output node.course.number, node.course.title;

DisplayCourses(node.right);

Function: SortedOutput(root):

DisplayCourses(root);

**Evaluation**

Big O analysis of the segments of above pseudocode:

1. O(1): Opening a file is a constant-time operation.
2. O(n): For reading the file, ‘n’ represents the number of lines present.
3. O(1): Each line of data is parsed and is of equal value.
4. O(1): When checking each line for incorrectly formatted data.
5. O(1): For creating an object of ‘Course’.

The total time-complexity for reading the file and creating course objects correlates to O(n).

**Comparison of Each Data Structure**

(‘+’ = advantages, ‘-’= disadvantages)

**(Vector)**

+Elements with an index value of O(1) can be accessed directly and efficiently.

+Vectors can be easily resized dynamically.

+Efficiency is increased because memory being contiguous.

-Insertion and deletion within the vector requires O(n), which creates a larger strain on processing requirements.

**(Hash Table)**

+Offers an average constant-time complexity of O(1) for access, insertion, and deletion.

+More efficient for data that is assigned unique key values.

-Performance can vary dramatically based on the consistency of the hashed data.

-Time complexity can degrade to O(n) is hashed data is poorly organized.

-No intrinsic order of data values is maintained.

-Memory overhead costs are significantly higher.

**(Binary Search Tree)**

+The logarithmic time-complexity for access, insertion, and deletion of entries within a balanced tree is O(log n).

+Order is maintained which makes traversals throughout much more efficient.

-Processing requirements can increase to O(n) if the tree is unbalanced, virtually eliminating the advantage over vectors pending specific circumstances.

-More memory will be required upfront to store all the branches of the tree.

**Recommendation**

Given the above analysis, the best options in terms of computation efficiency would be the Vector or Binary search tree. Furthermore, given the consistency of the data that is being processed (each entry having a name, number, designation), the Binary Search Tree would be the clear choice due to the increased performance offered to structured data. It is worth considering that this advantage could be negated if the tree becomes unbalanced. Given the requirements for the definition of a ‘Course’ that is unlikely to present an issue. Overall, the Binary Search Tree is the recommended option due to its general performance demands relative to its organizational advantages such as proper maintenance of ordered data entries as well as efficient access and modification operations within the data structure.