CS 300

12/3/2023

Project One – Runtime Analysis

# Pseudocode

### Defining the Course class

// defining structure of the Course class

Class Course {

String courseNumber

String courseName

Vector<string>Prereqs

}

### Vector

#### File Parsing

// Opens the file

Vector<String> Courses(filename) {

// string to hold the contents of the file in a line

String line

// try to open file

Ifstream instream(filename)

If file open error:

Print “Error opening file”.

Return

Read instream and add to line until no more lines in the file

// Adds the line to the vector

While instream has new line:

Courses.Pushback(line)

Instream.close

Return

}

#### Loading the ADT (Vector)

// Create empty vector to store course objects

Vector<Course> loadCourses(Vector<String> lines) {

Vector<Course> courses;

For each line in lines:

Course course; // makes a new Course object for each line

Vector<String> courseCheck = split(line, ‘,’)

If courseCheck.size < 2

Print “Invalid course found.”

continue //Skips that line and goes to the next

Course.courseNumber = courseCheck[0]

Course.courseName = courseCheck[1]

// Add remaining items as prerequisites in Vector string Prereqs

For i=2 to courseCheck.count-1

Course.Prereqs.Add(courseCheck[i])

// Add course to vector

Courses.pushback(course)

Return

}

#### Searching for and printing a course

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

}

### Hash Table

#### Defining Hash Table

Class HashTable

Struct Node:

Course courses;

Node\* next;

Int key;

Node()

Node(course, key)

Vector <node> Nodes

Int hash(key)

#### Loading the ADT (Hash Table)

//creating an empty Hash Table for course objects

HashTable<String,Course> courseTable

For each line in lines:

Course course; // makes a new Course object for each line

//using a vector to check course info.

Vector<String> courseCheck = split(line, ‘,’)

If courseCheck.size < 2

Print “Invalid course found.”

continue //Skips that line and goes to the next

Course.courseNumber = courseCheck[0]

Course.courseName = courseCheck[1]

// Add remaining items as prerequisites in Vector string Prereqs

For i=2 to courseCheck.count-1

Course.Prereqs.Add(courseCheck[i])

// Add course to hash table – courseNumber is the key

courseTable.insert(course.courseNumber, course)

Return

}

#### Searching for and printing a course

void printCourseInformation(HashTable<String, Course> courses, String courseNumber) {

key = hash of courseNumber

node = key

while node != nullptr:

if the node.courseNumber = courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

else

node = next node

}

### Binary Tree

#### Defining nodes for BST

void printCourseInformation(HashTable<String, Course> courses, String courseNumber) {

key = hash of courseNumber

node = key

while node != nullptr:

if the node.courseNumber = courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

else

node = next node

}

#### File Parsing

void printCourseInformation(HashTable<String, Course> courses, String courseNumber) {

key = hash of courseNumber

node = key

while node != nullptr:

if the node.courseNumber = courseNumber

print out the course information

for each prerequisite of the course

print the prerequisite course information

else

node = next node

}

#### Loading the ADT(Binary Tree)

// inserting new course nodes into BST

Node insertNode(Node node, Course course) {

If node is null:

Node newNode = new Node()

newNode.data = course

return newNode

If course’s course number < node’s courseNumber:

node.leftChild = insert node(node.leftChild, course)

else if course’s course number > node’s course number:

node.rightChild = insertNode(node.rightChild, course)

return node

// Loading courses into the BST

Node loadCourses(Vector<String> lines) {

Root = null

For each line in lines:

Vector<String> courseCheck = split(line, ‘,’)

If courseCheck.size < 2:

Print “invalid course”

Continue //skips that course and goes to the next

Course course

Course.courseNumber = courseCheck[0]

Course.courseName = courseCheck[1]

// Add prereqs to the BST

BST<String> prereqsBST

For i=2 to courseCheck.count:

// Check if prerequisite exists

Boolean prereqExists = false

For each course in courses:

If course.courseNumber = courseCheck[i]

prereqExists = true

break

if prereqExists = true:

//add prereq to the BST

prereqsBST.insert(courseCheck[i])

else:

Print: “Invalid prerequisite:” + coursecheck[i]

Course.Prereqs = prereqsBST

Root = insertNode(root, course)

Return root

}

#### Searching for and printing a course

void printCourseInformation(HashTable<String, Course> courses, String courseNumber) {

key = hash of courseNumber

node = key

while node != nullptr:

if the node.courseNumber = courseNumber

print out the course information

for each prerequisite of the course:

print the prerequisite course information

else

node = next node

}

### Menu

The following pseudocode is assuming a Vector data structure is used.

#### Load Data Structure

Function loadData(fileName, Courses vector):

fileName = courselist.csv // Can change depending on file name

// Use the structure of the data structure chosen. Refer to “Loading the ADT (Vector)”

#### Print Course List

Function Partition(Courses Vector, int low, int high):

// This is a partition function as a precursor to the quick sort algorithm we need to implement to print the courses sorted alphanumerically.

Midpoint = low + (high-low)/2

Pivot = courses(midpoint)

While not done:

While courses(low) < pivot

Low += 1

While pivot < courses(high)

High -= 1

If low >= high

Done = true

Else

Swap courses(low) and courses(high)

Low += 1

High -= 1

Return high

Quicksort(Courses Vector, low, high)

// This is the actual recursive quick sort algorithm

If low >= high

Return

lowIndex = Partition(Courses, low, high)

Quicksort(Courses, low, lowIndex)

Quicksort(Courses, lowIndex + 1, high)

Function printCourseList:

For each course in Courses:

printCourse(course, course.courseId)

#### Print Course

Function printCourse(node, String id):

If node = null

Return

If Node’s courseId = id

Print Course ID: + node’s courseId

Print Course Name : + node’s courseTitle

For each prereq in prereq vector’s length:

Print node’s prerequisite information

#### Displaying the menu

Loop until userInput = 4:

Print “Menu:”

Print “1. Load Data”

Print “2. Print Course List”

Print “3. Print Course Information”

Print “4. Exit”

Get userInput

Switch:

Case 1:

loadData()

Case 2:

If data has been loaded:

printCourseList()

Else print “Please load data first.”

break

Case 3:

If data has been loaded:

printCourse()

Else print “Please load data first.”

break

Case 4:

Exit program

break

# Runtime Analysis

### Vector Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define Vector** | 1 | n | n |
| **For each line in file** | 1 | n | n |
| **Create a new Course object** | 1 | n | n |
| **Create courseCheck vector** | 1 | 1 | 1 |
| **Split read line by commas** | 1 | n | n |
| **If courseCheck size is less than 2, skip the invalid course** | 1 | n | n |
| **Assign courseNumber as string at courseCheck[0]** | 1 | n | n |
| **Assign courseName as string at courseCheck[0]** | 1 | n | n |
| **While prerequisite exists** | 1 | n | n |
| **Add remaining items as prerequisites** | 1 | n | n |
| **Add course to vector** | 1 | n | n |
| **Total Cost** | | | 10n + 1 |
| **Runtime** | | | O(n) |

### Hash Table Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Create HashTable** | 1 | n | n |
| **For each line in file** | 1 | n | n |
| **Create a new Course object** | 1 | n | n |
| **Create courseCheck vector** | 1 | 1 | 1 |
| **Split read line by commas** | 1 | n | n |
| **If courseCheck size is less than 2, skip the invalid course** | 1 | n | n |
| **Assign courseNumber as string at courseCheck[0]** | 1 | n | n |
| **Assign courseName as string at courseCheck[0]** | 1 | n | n |
| **While prerequisite exists** | 1 | n | n |
| **Add remaining items as prerequisites** | 1 | n | n |
| **Add course to Hash Table with courseNumber as key** | 1 | n | n |
| **Total Cost** | | | 10n + 1 |
| **Runtime** | | | O(n) |

### Binary Tree Runtime Analysis

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Define insertNode method** | 1 | 1 | 1 |
| **If node is null** | 1 | 1 | 1 |
| **Make this node the root node** | 1 | 1 | 1 |
| **If course < root, add to left branch** | 1 | n | n |
| **If no left node, this node becomes the left node** | 1 | n | n |
| **If course > root, add to right branch** | 1 | n | n |
| **If no right node, this node becomes the right node** |  |  |  |
| **Create a courseCheck Vector** | 1 | 1 | 1 |
| **Split read line by commas** | 1 | n | n |
| **If courseCheck size is less than 2, skip the invalid course** | 1 | n | n |
| **Assign courseNumber as string at courseCheck[0]** | 1 | n | n |
| **Assign courseName as string at courseCheck[0]** | 1 | n | n |
| **While prerequisite exists** | 1 | n | n |
| **Add remaining items as prerequisites** | 1 | n | n |
| **Add course to BST** | 1 | n | n |
| **Total Cost** | | | 10n + 4 |
| **Runtime** | | | O(n) |

# Recommendation

Every data structure has its advantages and disadvantages. Though all three data structures shared O(n) complexity through my runtime analysis, I found that the worst choice was clear, but the best choice is less so.

Vectors are the simplest to implement. Insertion/deletion may be slower than other data structures as each element must be accessed sequentially, but for a data set the size of a college’s course list, the extra time would be minimal. Additionally, to print a sorted list as the program requires, one would have to implement a sorting algorithm such as quick or bubble sort both of which can approach O(n^2) complexity.

Hash tables are able to be searched quickly and inserting/deleting new courses would be quick at O(1) complexity, but are unable to be sorted easily. Seeing as a key feature of this program is the ability to print all courses sorted alphanumerically, this would not be a good choice as the data structure for this program.

Binary Search Trees are efficient for searching, insertion, deletion with O(n) in the worst case, but tree imbalances are a concern. It is sorted inherently, but if we were to receive a course list that was sorted or nearly-sorted without any code to act as a balancing mechanism, the program may approach worst time efficiency or even crash.

For this program, I recommend the Vector data structure, as I think it’s worth the minor time increase to avoid the issues that may emerge with a Binary Search Tree.