6-2: Project One

Sulochana Pradhan

ID: 2117223

DSA: Analysis and Design

Date: 10/07/2022

**CS 300 Pseudocode Document**

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

**ReadFile**

Reading from the file is same for all three data structures.

The The ReadFile function has a file path parameter. It returns true if all data are valid. Otherwise, it returns false.

**ReadFile(filePath)**

vector<string> prerequisiteCourseIds

vector<string> courseIds

file = open(filePath)

if file == null then:

return false

end if

while not end of file:

line = readline(file)

dataArray = split(line, ‘,’)

course = null

if length of dataArray < 2 then:

return false

else:

prerequisiteCourseIds

for i = 2 to length of dataArray:

append dataArray[i] to prerequisiteCourseIds

end for

end if

end while

for courseId in prerequisiteCourseIds:

if courseId is not in courseIds:

return false

end if

end for

return true

**1.1 Vector Data Structure**

**Assumption**

There is a data type Course.

**class Course**

courseId: string

description: string

prerequisites: vector<string>

**Pseudocode**

The CreateObject function uses a file object and creates a list of courses. It returns all the courses in the vector list.

**CreateObject(file)**

vector<Course> courses

while not end of file

line = readline(file)

dataArray = split(line, ‘,’)

if length of dataArray < 2 then

return false

else

vector<string> prerequisiteCourseIds

for i = 2 to length of dataArray

append dataArray[i] to prerequisiteCourseIds

end for

course = Course(dataArray[0], dataArray[1], prerequisiteCourseIds)

append course to courses

end if

end while

return courses

The PrintCourse function has courses and courseId parameters. It prints the course information and its prerequisites

**PrintCourse(Vector<Course> courses, string courseId)**

for course in courses

if course.courseId == courseId then

print courseId and description of course

for prerequisite in course.prerequisites

print prerequisite

end for

return

end if

end for

print “course not found”

The PrintCourseList function has courses parameters. It prints all course information.

**PrintCourseList(Vector<Course> courses)**

for course in courses

print courseId and description of course

for prerequisite in course.prerequisites

print prerequisite

end for

end for

**1.2 Hash Table Data Structure**

**Assumption**

Data types:

**class Course**

courseId: string

description: string

prerequisites: vector<string>

**class Node**

data: Course

key: int

next: Node

**class Hashtable**

courses: vector<Node>

size: int

hash(courseId: string): int

Insert(course, key)

The CreateObject function uses a file object and creates a list of courses. It returns all the courses in the hashtable.

**CreateObject(file)**

Hashtable courses

while not end of file

line = readline(file)

dataArray = split(line, ‘,’)

if length of dataArray < 2 then

return false

else

vector<string> prerequisiteCourseIds

for i = 2 to length of dataArray

append dataArray[i] to prerequisiteCourseIds

end for

course = Course(dataArray[0], dataArray[1], prerequisiteCourseIds)

hashKey = hash(dataArray[0])

courses[hashKey] = Insert(hashKey, course)

end if

end while

return courses

Insert function inserts course to hash table: courses using given hashKey

**Insert(hashKey, course)**

existingNode = courses[hashKey]

if existingNode == null then:

node = Node(course, hashKey)

nodes[hashKey] = node

else if existingNode->key is unused:

existingNode ->key = hashKey

existingNode->course = course

existingNode->next = null

else:

while existingNode->next != null:

existingNode = existingNode->next

end while

existingNode->next = Node(course, hashKey)

end if

The PrintCourse function prints the course information of given courseId

**PrintCourse (Hashtable courses, string courseId)**

hashKey = hash(courseId)

courseNode = courses[hashKey]

if existingNode == null then:

print “No course found”

return

else:

while existingNode != null:

if existingNode->course->courseId == courseId then:

course = node->next->data

print node->course information

print course->prerequisites

return

end if

existingNode = existingNode->next

end while

print “Course not found”

end if

The PrintCourseList function displays all course information.

**PrintCourseList (Hashtable courses)**

for i from 0 to size(courses) -1:

Node node = courses[i]

if node->key is not unused:

while node != null:

course = node->data

print course information

print course->prerequisites

node = node->next

end while

end if

end for

**1.3 Tree Data Structure**

**Assumption**

The following data types are assumed to be available. To avoid page break in pseudocode, I have added some gaps between pages.

Data types:

**class Course**

courseId: string

description: string

prerequisites: vector<string>

**class Node**

course: Course

left: Node

right: Node

**class CourseTree**

root: Node

**Pseudocode**

The CreateObject function uses a file object and creates a list of courses. It returns all the courses in the binary tree structure. It uses addNode method.

**CreateObject(file)**

CourseTree tree

while not end of file

line = readline(file)

dataArray = split(line, ‘,’)

if length of dataArray < 2 then

return false

else

vector<string> prerequisiteCourseIds

for i = 2 to length of dataArray

append dataArray[i] to prerequisiteCourseIds

end for

course = Course(dataArray[0], dataArray[1], prerequisiteCourseIds)

If tree == null:

tree = new Node(course)

else:

addNode(tree->root, course)

end if

end if

end while

return courses

**addNode(node, course)**

if node-> course ->courseId > course->courseId then:

if node->left == null then:

node->left = Node(course)

else:

addNode(node->left, course)

end if

else:

if node->right == null then:

node->right = Node(course)

else:

addNode(node->right, course)

end if:

end if

The PrintCourse function prints the course information of given courseId

**PrintCourse (CourseTree courses, string courseId)**

**recursiveFindAndPrint**(courses->node, courseId)

**recursiveFindAndPrint(Node node, string courseId)**

if node->course->courseId == courseId then:

print node->course information

print node->course->prerequisite courses information

return

else if courseId < node->course->courseId:

recursiveFind(node->left, courseId)

else:

recursiveFind(node->right, courseId)

end if:

print “Course not found”

The PrintCourseList function displays all course information. It uses inorderTraverse method.

**PrintCourseList(CourseTree courses)**

inorderTraverse(courses->node)

**inorderTraverse(node)**

if node != null then:

inorderTraverse(node->left)

print node->course information

print node->course->prerequisite courses information

inorderTraverse(node->right)

end if

# 2. Create pseudocode for a menu.

DisplayMenu()

filePath = “data.csv”

userChoice = 0

courses

while userChoice != 4:

print “Menu”

print “1. Load Data Structure”

print “2. Print Course List”

print “3. Print Course”

print “4. Exit”

userChoice = INPUT (“Menu choice”)

switch(userChoice):

case 1:

file = ReadFile(filePath)

courses = CreateObject(file)

break

case 2:

PrintCourseList(courses)

break

case 3:

courseId = INPUT(“Course Id”)

PrintCourse(courses, courseId)

break

end switch

end while

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

**3.1 Vector data structure**

I will use quicksort algorithm to sort the vector data

**QuickSort(vector<Course> courses, low, high)**

if low >= high then:

return

pivotEnd = Partition(courses, low, high)

QuickSort(courses, low, pivotEnd)

QuickSort(courses, pivotEnd + 1, high)

**Partition(vector<Course> courses, low, high)**

mid = low + (high – low) / 2

pivot = courses->at(mid)

while high > low:

while courses->at(low)->courseId < pivot->courseId:

low = low + 1

end while

while courses->at(high)->courseId > pivot->courseId:

high = high – 1

end while

if low >= high then:

break

end if

tempCourse = courses->at(low)

courses->at(low) = courses->at(high)

courses->at(high) = tempCourse

low = low + 1

high = high - 1

end while

**PrintSortedList(vector<Course> courses)**

QuickSort(courses, 0, courses->size() – 1)

PrintCourseList(courses)

**3.2 Hash Table Data Structure**

There is no point in sorting hash table as we can access an item in O(1) time complexity. If we want to sort our hastable, then we can covert it to vector data structure and just use same Quick sort algorithm as in above example. Other way is to create ordered hash table data structure.

**3.3 Tree Data Structure**

We can use inorder traversal which just prints the course information in the sorted order.

**PrintSortedList(CourseTree courses)**

inorderTraverse(courses->node)

**inorderTraverse(node)**

if node != null then:

inorderTraverse(node->left)

print node->course information

print node->course->prerequisite courses information

inorderTraverse(node->right)

end if

**4. Evaluation**

Reading each line from the files takes O(n2) time complexity. Here n is the number of lines. The square is because we have another loop to get all the prerequisites for each course. The space complexity is O(n).

**4.1 Loading data time complexity**

Loading to the data structure is different for different data structures as it depends on the insert implementation. For vector data structure, the time complexity is O(n2) as we are simply appending each course item into a vector. For the hash table, if we use the perfect hash function it takes O(1) to insert but we need to consider the worst case it will be O(n). So, the total time complexity will be O(n3). For tree data structure, the insertion takes O(n) for the worst case. Usually, the time complexity will be O(logn). So, our total time complexity becomes O(n3). The space complexity for all data structures is O(n2) as we are saving prerequisites with each course.

**4.2 Explain the advantages and disadvantages of each structure in your evaluation.**

The vector data structure is easy implementation. We are just appending each course in vectors. So, each operation is simple and easy to understand. It takes O(1) to insert a new course. To remove and search it takes O(n) time complexity which is not ideal when we have a large number of data. As data in vectors are not sorted, we need to use some sorting algorithm to sort the data. If we use quick to sort it takes O(n2) time complexity for the worst case. However, we can use merge sort to reduce the time complexity to O(nlogn). The merge sort implementation increases the space complexity.

The hash table data structure provides faster data retrieval. If the hash function is good, it can insert, retrieve and remove an item in O(1) time complexity. The key idea of the hash table is to provide faster access to the data. There is a hidden cost behind which comes when we need to resize our hash table. Moreover, we need to allocate certain spaces beforehand which may not even be used. There is no point in sorting data in the hash table and we may need to convert it to a vector data structure and sort it.

The binary tree structure provides the sorted data structure. Ideally, all the operation takes O(logn) time complexity, however, in the worst-case time complexity becomes O(n). There is no need to sort the binary tree as it is already sorted. As most of the time we can assume the data is random, so most of the time it takes O(logn) time complexity.

**4.3 Recommendation**

The project is for the computer science department. We are not talking about a large data set. I would go with a binary search tree data structure. Though in the worst-case time complexity of most of the operation is O(n), the chances of that are slim. If the computer science department has 200 courses, ideally it takes less than 8 operations to perform the action. Another advantage is as the data structure is already sorted; we do not need to perform another sorting implementation.