Zac McBride

CS-300

ABCU Courses Pseudocode

Course Class {

Private:  
 string courseNumber

String courseName

Int totalPrerequisites

Array prerequisiteList

Public:

getCourseNumber()

setCourseNumber()

getCourseName()

setCourseName()

getTotalPrerequisites()

setTotalPrerequisites()

appendPrerequisite()

printPrerequisites()

}

ReadCourseFile() {

Open courseFile

If courseFile does not open {

Print (courseFile “did not open”)

}

inputStream = courseFile line

while (inputStream is not end of file) {

new Courses inputCourse;

int numPrerequisites = 0

while (inputStream has ‘,’) {

dataString = parse line at ‘,’

inputCourse.setCourseNumber(dataString)

dataString = parse line at ‘,’;

inputCourse.setCourseName(dataString)

while (parse line) {

dataString = parse line at ‘,’

if (parse line) {

++numPrerequisites

}

inputCourse.appendPrerequisite(dataString)

}

(DataType)Insert(inputCourse, inputCourse.courseNumber)

}

}

PrintCourseInfo() {

Print “Enter course number”

inputCourse = input

course = SearchCourse(inputCourse)

Print (course.getCourseName() “, “ course.getCourseNumber() “\n”)

Print (course.getTotalPrerequisites() “ prerequisites: “)

course.printPrerequisites()

}

PrintMenu() {

Print “1 – Load Data Structure”

Print “2 – Print Course List”

Print “3 – Print Course”

Print “4 – Exit”

}

MenuLoop() {

PrintMenu()

inputValue = input

While (inputValue is not 4) {

if (inputValue == 1) {

ReadCourseFile()

}

if (inputValue == 2) {

(DataType)PrintCourses()

}

if (inputValue == 3) {

PrintCourseInfo()

}

PrintMenu()

inputValue = input

}

}

**LinkedList**

LinkedList Class {

Private:

Course course

Course <vector> csCourses

Public:

LinkedList()

Insert(Course course, course number)

SearchCourse(courseNumber)

PrintCourses()

SortCourses()

partition(lowIndex, highIndex)

}

Insert(Course course, courseNumber) {

csCourse.append(course)

}

SearchCourse(inputCourse) {

For (i = 0; I < csCourses.size(); ++i) {

If (csCourses[i] == inputCourse) {

Return csCourse[i]

}

}

}

SortCourse(lowIndex, highIndex) {

if (lowIndex >= highIndex) {

return

}

midIndex = partition(lowIndex, highIndex)

SortCourses(lowIndex, midIndex)

SortCourses(midIndex + 1, highIndex)

}

partition(lowIndex, highIndex) {

midIndex = lowIndex + (highIndex - lowIndex) / 2

pivot = csCourses[midIndex]

while (lowIndex >= highIndex) {

while (csCourses[lowIndex] < pivot) {

lowIndex += 1

}

while (pivot < csCourses[highIndex]) {

highIndex -= 1

}

if (lowIndex >= highIndex) {

return

}

else {

tempVal = csCourses[lowIndex]

csCourses[lowIndex] = csCourses[highIndex]

csCourses[highIndex] = tempVal

lowIndex += 1

highIndex -= 1

}

}

return highIndex

}

PrintCourses() {

SortCourses()

For (i = 0; I < csCourses.size(); ++i) {

csCourses[i].PrintCourseInfo()

}

}

**HashTable**

HashTable Class {

Private:  
 struct Node {

Course course

Int key

Node-> next

}

Node(Course newCourse, courseKey) {

Node course = newCourse

key = courseKey

}

Node <vector> courseNodes

Int tableSize = 8

Int hash(key)

Public:

HashTable()

Insert(Course course, course number)

SearchCourse(courseNumber)

PrintCourses()

}

HashTable() {

courseNodes.resize(tableSize)

}

hash(key) {

return key % tableSize

}

Insert(Course course, courseNumber) {

key = hash(course.courseNumber to int)

bucket = courseNodes.at(key)

BucketsProbed = 0

While (bucketsProbed < tableSize) {

If (courseNodes.at(bucket) == null) {

courseNodes.at(bucket) = course

Break

}

Bucket = (bucket + 1) % tableSize

++bucketsProbed

}

}

SearchCourse(courseNumber) {

searchKey = hash(courseNumber to int)

bucketsProbed = 0

While (courseNodes.at(bucket).courseNumber is not courseNumber and bucketsProbed < tableSize) {

If (courseNumber.at(bucket) is not null and courseNodes.at(bucket).courseNumber == courseNumber) {

Return course.at(bucket)

}

Bucket = (bucket + 1) % tableSize

++bucketsProbed

}

Return null

}

PrintCourses() {

While (nextCourse is not null) {

For (i = 0; i < tableSize; ++i) {

If (courseNode.at(i) is not null) {

nextCourse = courseNode.at(i)

break

}

}

For (i = 0; i < tableSize; ++i) {

If (nextCourse > courseNode.at(i) and courseNode.at(i) is not null) {

nextCourse = courseNode.at(i)

}

}

PrintCourseInfo(nextCourse)

previousCourse = nextCourse

For (i = 0; i < tableSize; ++i) {

For (j = 0; j < tableSize; ++j) {

If (courseNode.at(j) > previousCourse and courseNode.at(j) < nextCourse) {

If (courseNode.at(j) is not null) {

PrintCourseInfo(courseNode.at(j))

}

}

}

}

}

**BinarySearchTree**

BinarySearchTree Class {

Private:  
 struct Node {

Course course

Int key

Node-> left

Node->Right

Node() {

Left = null

Right = null

}

Node <vector> courseNodes

Public:

BinarySearchTree()

Insert(Course course, courseNumber)

SearchCourse(courseNumber)

PrintCourses()

BinarySeachPrintInOrder()

}

BinarySearchTree() {

Root = null

}

Insert(Course course, courseNumber) {

If (root == null) {

Root = new Node(course)

Return

root = node

while (node is not null) {

if (node->course.courseNumber < courseNumber) {

if (node->Left == null) {

node->Left = course

return

}

else {

node = node->Left

}

}

else if (node ->course.courseNumber > courseNumber) {

if (node->Right == null) {

node->right = course

return

}

else {

node = node->right

}

}

}

}

SearchCourse(courseNumber) {

currentNode = root

while (node is not null) {

if (currentNode->course.courseNumber == courseNumber) {

return currentNode

}

else if (currentNode->course.courseNumber > courseNumber) {

currentNode = currentNode->Left

}

else {

currentNode = currentNode->Right

}

}

Return currentNode

}

PrintCourses() {

BinarySeachPrintInOrder(root)

}

BinarySeachPrintInOrder(node) {

BinarySeachPrintInOrder(node->left)

PrintCousreInfo(node)

BinarySeachPrintInOrder(node->right)

}

**Evaluation**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Data Type | Read File | Print Courses | Search (Average) | Search (Worst) | Insert (Average) | Insert (Worst) | Memory |
| Linked-List | O() | O() | O() | O() | O() | O() | O() |
| Hash Table | O() | O() | O(1) | O() | O(1) | O() | O() |
| Binary Search Tree | O() | O() | O() | O() | O() | O() | O() |

This is a table of the Big O analysis for the pseudocode above. is the number of data points in the Course Class. There are a couple things to note with my analysis of the code. The ReadFile() function is the same for each data type calling a different Insert function for each data type. There may be a more efficient way to print the list of all courses in order for the HashTable data type but with this code the time complexity for Print Courses is O(). The memory used for each data type varies slightly form the Big O given. The Hash Table and Binary Search Tree will use slightly more space because they have other object items to get them to work, however for the sake of this code the differences are negligible.

The Linked-List is the most straight forward code which may be easier to write and maintain but aside from this it does not have many advantages over the other two. The HashTable has the biggest advantage in searching for a particular course given the best case for this is O(1). The HashTable falls short when it comes to printing all the courses in order. This is where the Binary Search Tree shines as it should allow for easy traversal through the data in order so no sorting or comparing is necessary. It may not have the best time complexity for each function, but it does have the best average time complexity over all the functions.

Looking at the advantages and disadvantages of all three data structures, I will write this program using the Binary Search Tree. For the requirements of the program, this data structure should preform the most efficiently for all the functions. After initially reading the file, the user will only use the program to print information about one course or the entire list of courses. The Binary Search Tree is fast in both circumstances which is the biggest reason I recommend using it.