Andrew Robinson
Module Six Project One
CS 300
08/04/22

Project One: ABC University Pseudocode/Evaluation

Trying to figure out which algorithm that is best for ABC University is going to require pseudocode from many different types of data structures which will hold all of this data. After this there will be a need to have a runtime analysis to see which option is more beneficial to their overall needs. The data structures that will be focused on are vectors, hash tables, and binary search trees. Each of these data structures will be evaluated and will show the runtime and memory use. Each pseudocode will load a file, parse the file, insert course data into the data structure and print the course data.

Vector Pseudocode

//Dependencies
Import parsing libraries and headers

//Classes
class Course
    private string **courseID**
    private string **courseTitle**
    private int **prereqCount**
    private <vector> **prereqVector**
    private <vector> **courseVector**

//Mutators
setCourseID(string id)
    courseID = id

setCourseTitle(string title)
    courseTitle = title

setPrereqCount(string count)
    prereqCount = count

setPrereqVector(vector reqlist)
    prereqVector = reqlist

setCourseVector(object courseinfo)
    courseVector = courseinfo

//Accessors
getCourseID()
    return courseID

getCourseTitle()
    return courseTitle

getPrereqCount()
    return prereqCount

getPrereqVector()
    return prereqVector

//Validate prerequisites
validatePrereq(string prereq)
    Bool is false
    For each element is courseVector
        If elements courseID is equal to prereq
            Bool equals true
            Continue
        Else
            Bool equals false
            Print "No prerequisite found"
            Prompt user to press key to continue

//Display menu
displayMenu()
    Print "[1] Load courses"
    Print "[2] Search for course"
    Print "[3] Print course schedule"
    Print "[4] Create new course schedule"
    Print "[9] Exit program"

```
//Find csv file
getCsvPath()
    Request user input for string of file path
    filePath = userInput
    If no data is input
        Default path is used

//Iterate through csv file
lineParser(string filePath)
    Object curCourse
    <Vector> String curVector
    String firstItem
    String secondItem
    String thirdItem

    Open file at String filePath
    Loop iteration comma separated values
        If line exists
            Initialize firstItem with first value of line
            Initialize secondItem with second value of line
            Call curCourse object and setCourseID(firstLine)
            Call curCourse object and setCourseTitle(secondLine)
            If next line is not blank
                Initialize thirdItem with third value of line
                Call curCourse setPrereqCount(getPrereqCount() + 1)
                Add thirdItem to curVector
            Call curCourse object setPrereqVector(curVector)
            Clear curVector
            Call courseVector(curCourse)
        Else
            Move to next line
            If no line exists

//Search list
searchList(string id)
    For each element in courseVector
        If elements courseID is equal to id
            Call printCourse(element)
        Else
            Print “No such course found”
```

```
//Print Course
printCourse(Course element)
        curObject = element
        Print “Course ID: “ << getCourseID(curObject) << end line
        Print “Course Title: “ << getCourseTitle(curObject) << end line
        Print “Number of pre-requisites: “ << getPrereqCount(curObject) << end line
        Print “Pre-requisites: “ << getPrereqVector(curObject) << end line

//Print all courses
printAllCourses()
        For each element is courseVector
                Print “Course ID: << getCourseID() << end line
                Print “Course Title: “ << getCourseTitle() << end line
                Print “Number of pre-requisites: “ << getPrereqCount() << end line
                Print “ Prerequisites: “ << getPrereqVector() << end line

//Main
Main()
        Initialize <vector> courseVector
        Initialize int userChoice = 0;
        Initialize string userInput = “ ”
        Call DisplayMenu()
        Assign userChoice with input
        SWITCH userChoice
                CASE  = 1
                        Call lineParser(getCsvPath())
                CASE  = 2
                        Print “Enter Course ID: “
                        Assign userInput with input
                        Call searchList(userInput)
                CASE  = 3;
                        Call printAllCourses()
                CASE  =  9
                        Print “Good Bye”
                        EXIT PROGRAM
```

Vector Evaluation:

The worst runtime would mainly run around insertion, searching, deletion and printing of the data in the vector data structure. This is proportionately linked to the number of elements in the set, therefore we can surmise that with this logic the runtime complexity will be O(n). In order to insert a course we must parse the csv file line by line while searching each element. To accomplish this a nested while loop in the lineParser function will handle this. This means that for this to take place the runtime is O(n^2) because for N lines times N elements on each line being parsed is going to equal this runtime complexity. The searching operations are the worst case runtime of O(n) due to it being linear as once it matches the value the operation is done. Printing from the data structure will require at its worst case of O(n^2) as the operation will require it to search and print and prerequisites it may have. The space complexity of these functions are O(1) as the functions do not need to store any additional memory. Each of these functions utilize allocated memory from the vector and this vector data structure has a space complexity itself of O(n) as this is mainly dependent on the input size of the data.

Code	Line Cost	Iterations	Total Cost
All Courses	1	n	n
Course Info	1	n	n
Prerequisite	1	n	n*n
Print prereq info	1	n	n*2
Total Cost	4		n*n
Runtime			4n*n

Hash Table Pseudocode

```
//Classes
Class HashTable
Private
    Course course
    Node* nextNode
    Node()
        key = UINT_MAX
        nextNode = null
    Node(course, key)
        this course = course
        this key = key
    vector<node>table
Public
    HashTable()
    HashTable(int size)
    delHashtable //Destructor
    insert(Course)
    printAll()
        Return void

//Display menu
displayMenu()
    Print "[1] Load courses"
    Print "[2] Print Course List"
    Print "[3] Print Course"
    Print "[9] Exit program"

//Delete Hashtable
delHashtable()
    ERASE nodes vector from beginning

//Insert Course
insert(Course)
    string hashNum
    int temp = 0
    key = null
    Node node
    FOR (int i = 0; i < Course size; ++i)
        IF i < 4
```

```
                temp = Course courseNum at i
                APPEND temp to hashNum
        ELSE
                APPEND Course courseNum at i
    key = ATOI hashNum by c_str()
    node = key
    IF node != null
        Node newNode = Node(course, key)
    ELSE
        IF node = UINT_MAX
            Node key = key
            node course = course
            node* nextNode = null
        ELSE
            WHILE node* nextNode != null
                node = nextNode
            node* nextNode = node(course, key)
```

//How the program opens the file, reads the data, parses each line and checks for errors

readFile(fileName, HashTable hashTable)

```
    Fstream file = fileName
    String line
    String courseData

    OPEN file
    IF file is open
        Continue
    ELSE
        Print "Error message"

    WHILE not at eof
        CREATE Course course
        ARRAY string type textLine
        WHILE textLine is good
             GETLINE from file by parsing
            APPEND to end of textLine
        FOR textLine
            IF string at index 0
                Course courseNum = string
            ELSE IF string at index 1
```

```
                course courseName to string
            ELSE
                IF string at index 3 = null
                    Print “Error reading data”
                    Return courses
                course coursePrereq to string
            insert(course) to hashTable
    CLOSE file

//How to print out course information and prerequisites
printCourseInfo(HashTable)
    FOR (i = 0; i < nodes length; ++i)
        int prereqVector = course prereq size
        IF i != UINT_MAX
            Print courseNumber, courseName
            IF prereqVector > 0
                FOR (int prereq = 0; prereq < prereqVector; ++prereqVector)
                    Print course prereq
            ELSE
                Print “No prerequisites”
        Node node = node* nextNode
//Main
main()
    String fileName
    Int userChoice = 0
    HashTable courseTable
    Course course

    WHILE userChoice != 9
        Print displayMenu()
        userChoice = user input
        SWITCH userChoice
            CASE 1:
                CALL readFile(fileName, courseTable)
            CASE 2:
                CALL printCourseInfo(courseTable)
            CASE 3:
                CALL printCourseInfo(course)
            CASE 9:
                Print “Good Bye”
```

Hash Table Evaluation:

The hash table is great for insertion as well as lookup times provided by the hash function. The hash table data structure utilizes a similar parsing method as the vector data structure; however, the insertion function makes this faster with a time complexity of O(1) on average. This is because “On average, a good hash function will achieve O(1) inserts, searches, and removes, but in the worst-case may require O(N).” (Vahid). The worst runtime complexity for a hash table is O(n) which may occur if many elements of the input are hashed to the same bucket. A good hash function will avoid these issues and thus in turn the runtime complexity will be on average O(1). Searching a hash table also has on average O(1) and O(n) worst case. To print a single entry the runtime complexity would be O(1) as the hash function computes the value and corresponds that to the key. However, to print the entire hash table the runtime complexity would be O(n^2) as the for loop and while loops will ensure each node is reached and each next node is pointed to afterwards. The space complexity for a hash table data structure does not require additional space since all the allocated memory to compute the key and return the value are already established.

Code	Line Cost	Iterations	Total Cost
All Courses	1	n	n
If course not UINT MAX	1	n	n
Print course info	1	n	n*n
For each prerequisite	1	n*n	n*n
Print prereq info	1	n*n	n*n
Create node set to next pointer	1	n	n
While node not nullptr	1	n*n	n*n
Print out the course table	1	n*n	n*n
Total Cost	8	n*n	n*n
Runtime			8n*n
			O(n^2)

Binary Tree Pseudocode

```
Struct Course
	String courseName
	String courseNumber
	Vector<String> Prereq

Struct Node
	Course course
	Create key

	IF(entry matches key)
		Return node (course)
	IF(no entry matches)
		Return course

Class Tree
	Private:
		Node* root
		Void addNode(Node* node, Course course)

	Public:
		BinarySearchTree()
		Void inOrder()
		Void insert(Course course)
		courseSearch(string couseNum)
```

BinarySearchTree::BinarySearchTree

```
	Root = nullptr
Void BinarySearchTree::Insert(Course course)
	IF(root = nullptr)
		Root = newNode(course)
	ELSE
		this->addNode(root, course)
```

void BinarySearchTree::addNode(Node* node, Course course)

```
IF(node -> bidID > 0)
    IF(node != nullptr)
        Node -> right = new Node(course)
    ELSE
        This -> addNode(node -> right, course)
```

Course BinarySearchTree::Search(string courseNum)

```
Node* current = root
WHILE(current != nullptr)
    IF (courseNum matches)
        Return current -> course
    IF (current node compare to courseNum > 0)
        Current = current -> left node
    ELSE
        Current = current -> right node
Return course
```

readFile(fileName, new BinarySearchTree)

```
Fstream file fileName
String line
Open file
IF file is OPEN
    WHILE (getline(file))
        Course course
        ARRAY string textFromLine
        WHILE textFromLine is good
            APPEND line to textFromLine
        IF textFromLine length < 3
            course.courseNum = textFromLine[0]
            course.courseName = textFromLine[1]
            course.coursePrereq = null
        ELSE
            course.courseNum = textFromLine[0]
            course.courseName = textFromLine[1]
            course.coursePrereq = textFromLine[2]
        INSERT course into BinarySearchTree

        CLOSE file
```

```
printInOrder(tree)
    IF node != null
    CALL printInOrder(left node)
    Print node courseNum, courseName
    FOR (prereq in course.prereq)
        Print prereq
    CALL printInOrder(right node)

printCourseInfo(String courseNum)
    Course course.search(courseNum)
    Print coursenum, courseName
    FOR (each prereq in course.prereq)
        Print prereq

//Main
main()
    STRING fileName = “file path”
    INT userChoice = 0
    BinarySearchTree tree
    COURSE course
    STRING courseNum
    Tree = new BinarySearchTree

    WHILE userChoice != null
        displayMenu()
        userChoice = user input
        SWITCH userChoice
            CASE 1:
                CALL readFile(fileName, tree)
            CASE 2:
                CALL printInOrder(tree)
            CASE 3:
                CIN courseNum
                CALL printCourse(courseNum)
            CASE 9:
                Print “Good Bye”
                EXIT PROGRAM
```

Binary Search Tree Evaluation:

The binary search tree is fast for insertion, searching and printing. The time complexity for the binary search tree is on average O(logN). The binary search tree benefits from only making as many comparisons as the number of nodes that correlates with the largest depth of any node. The worst case time complexity is O(n) as all the nodes are on the same branch as the height of the tree relates to the amount of elements from the input. Searching for a result requires that the algorithm make only as many comparisons and the height of the tree. This will allow the tree to be balanced and result in faster results. When inserting, the algorithm searches for an empty position on average it will have runtime complexity of O(logN). Printing each node could have a worst case runtime complexity of O(n) since the node that is to be printed could have a prerequisite which would mean a loop will run to print all prerequisite courses. When printing the entire tree in alphanumeric order, the runtime is O(n^2) for each node which is recursively called and then printed with prerequisites.

Code	Line Cost	Iterations	Total Cost
IF node not nullptr	1	1	1
Recursive call to left pointer	1	n	n
Print course information	1	n	n
For each prerequisite	1	n	n
Print the prerequisite info	1	n*n	n*n
Recursive call to right pointer	1	n	n
Total Cost	6	n*n	n*n
Runtime			1 + 5n * n
			O(n^2)

Advantages and Disadvantages:

Each of these data structures have their own advantages and disadvantages. A vector data structure is quite fast however a hash table or binary search tree are must more efficient. The other hand though, vector data structure is easy to implement with the code so this could be more appealing to some developers. The hash table is the fastest out of the remaining two data structures. The reason for this is because to insert, search or print a single node has an average runtime complexity of O(1). The downside of a hash table is that to avoid collisions it will be difficult with the code and this could cause runtime complexity to increase. The way to solve this is to establish the key ahead of time. The last option being the binary search tree shows some promise as it is also fast for inserting, searching and printing data from the nodes.

Recommendation:

With speed and efficiency in mind the hash table data structure would suit best. With the expected key being known has the average runtime complexity of O(1). This will be the right choice for ABC University to improve their data processing strategy. A proper hash table and hash function will be absolutely effective. Parsing the file is not that different when comparing each data structure, however when it involves inserting data into the structure this is much more important. Thus, the logical choice for what data structure to use must be the hash table data structure.

Resources

Vahid, F., Lysecky, S., Wheatland, N., Siu, R., Lysecky, R., Edgcomb, A., & Yuen, J. (2019, February). CS 300: Data Structures and Algorithms. Zybooks.