Giovanni Inglese

CS-300

Dr Farley

8/4/25

Project 1 Pseudocode

**Vector Pseudocode**

Struct Course

**String courseNumber**

**String title**

**Vector<string> prerequisites**

End struct

Declare vector < Course> courseVector;

Function loadCourseFile(string: filename) vector<course>

**Create empty vector vector<course> courses**

**Open input file with the name stored in file name**

**If file cannot be opened**

**Print “Error: Cannot open file”**

**Exit program**

**End if**

**lineNumber = 0**

**while there is another line in the file**

**read the line**

**lineNumber = lineNumber+1**

**If line is empty**

**Continue to next line**

**Endif**

**Split line into tokens by coma (“,”)**

**If tokens.size < 2**

**Print “Format error on line” + lineNumber**

**Continue to next line**

**End if**

**Create new course object**

**Set courseNumber = tokens[0]**

**Set title = tokens[1]**

**For I from 2 to tokens.size() -1**

**Add tokens[i] to course.prerequisites**

**End for**

**Add course to course vector**

**End while**

**Close file**

**For each course in courses**

**For each prereq in course.prerequisites**

**If not courseExists(courses, prereq)**

**Print “Error: prerequisites” + prereq + “ does not exist” + course.courseNumber**

**End if**

**End for**

**End for**

**Return courses**

**End function**

void searchCourse(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**

Function PrintCourse\_Vector(courseId)

For each course in courseVector

If course.courseNumber == courseId Then

Print course.courseNumber + ": " + course.title

If course.prerequisites is empty

Print "Prerequisites: None"

Else

Print "Prerequisites: " + Join(course.prerequisites)

Return

Print "Course not found"

EndFunction

Function PrintSortedCourses\_Vector()

Sort courseVector by course.courseNumber

For each course in courseVector

Print course.courseNumber + ": " + course.title

END function

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **Read Each Line** | 1 | n | n |
| **Split each line** | 1 | n | n |
| **Validate Each line prereqs** | n | n | N^2 |
| **Create Course Object** | 1 | n | n |
| **Insert into vector** | 1 | n | n |
| **Total Cost** | | | N^2 + 4n + 1 |
| **Runtime** | | | O(n^2) |

**Hash Table Pseudocode**

STRUCT course{

String CourseNumber;

String title;

Vector<string> prerequisites;

};

Function LoadCourse(String filename):

DECLARE hashTable as an UNORDERED\_MAP < STRING, Course>

DECLARE allLines as vector <string>

// Opens the file and reads lines into the vector

OPEN filename for reading file

If file NOT open:

Print: “Error: Could not open file”

Return hashTable

While getLine(file,line):

If line is NOT empty:

Append line to allLines

CLOSE file

// Collect all course numbers

DECLARE courseNumbers as UNORDERED\_SET<string>

FOR each line IN allLines:

SPLITline by coma(,) into TOKENS

Trim tokens

ADD tokens[0] to courseNumbers

//Parse each line and validate

For line in allLines:

SPLIT by coma(,) into TOKENS

Trim tokens

If Tokens.size() < 2:

PRINT “ Errpr: INVALID line format”

Continue

CourseNumber = tokens[0]

Title = tokens[1]

DECLARE prereqs as vector<string>

//Validate prereqs

FOR I at 2 to tokensize() -1:

Prereq = tokens[i]

IF prereq NOT in corseNumbers:

PRINT “Error: prerequisites “ + prereq + “ For “ + courseNumber + “ Does Not Exist “

Else:

APPEND prereq to prereqs

// Declare a course object

DECLARE course AS Course

course.courseNumber = courseNumber

course.title = title

course.prerequisites = prereqs

// Insert into hash table

hashTable[courseNumber] = course

Return HashTable

End Function

Function PrintAllCourses(UNORDERED\_MAP<String, Course> hashTable):

FOR each pair in hashTable:

Course = pair.second

PRINT “Course: “ + courseNumber + “ – “ + course.title

If course.prerequisites is not empty:

PRINT “Prerequisites: “

For each prereq in course.prerequisites:

PRINT prereq

Else:

PRINT “Prerequisites: None”

END function

Function PrintSortedCourses\_HashTable()

Create tempList

For each key in hashTable

Add hashTable[key] to tempList

Sort tempList by course.courseNumber

For each course in tempList

Print course.courseNumber + ": " + course.title

END function

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **Read Each Line** | 1 | n | n |
| **Split each line** | 1 | n | n |
| **Validate Each line prereqs** | 1 | n | n |
| **Create Course Object** | 1 | n | n |
| **Insert into vector** | 1 | n | n |
| **Total Cost** | | | 5n+1 |
| **Runtime** | | | O(n) |

**Binary Search Tree**

STRUCT Course:

courseNumber : String

courseTitle : String

Prerequisites : STRING list

END STRUCT

Create binarySearchTree called courseTree

FUNCTION loadCourses(fileName):

Open file with fileName

If file cannot be opened

PRINT “ Error: Couldn’t open file”

Return

CREATE empty LIST allCourses

FOR each line in file

SPLIT line by comma or space into tokens

If number of tokens < 2

Print “ Error: Line does not have course Number or title”

CONTINUE to next line

CREATE new course called courseObj

courseObj.courseNumber = tokens[0]

courseObj.courseTitle = tokens[1]

FOR I from 2 TO end of tokens

ADD tokens[i] to courseObj.prerequisites

ADD courseObj to allCourses

CLOSE file

[\\VALIDATE](file:///\\VALIDATE) PREREQS

FOR each course in allCourses

FOR each prereq in course.prerequisites

IF prereq does not exist as courseNumber in allCourses

PRINT “Error: prerequisite “ + prereq + “ not found in file”

[\\INSERT](file:///\\INSERT) COURSES INTO BST

FOR each course in allCourses

courseTree.insert(course)

END FUNCTION

FUNCTION Insert(course):

IF root is NULL

root = new Node(course)

ELSE

CALL addNode(root, course)

END FUNCTION

FUNCTION addNode(node, course):

IF course.courseNumber < node.course.courseNumber

IF node.left is NULL

node.left = new Node(course)

ELSE

addNode(node.left, course)

ELSE

IF node.right is NULL

node.right = new Node(course)

ELSE

addNode(node.right, course)

FUNCTION PrintCourse(courseNumber):

course = courseTree.Search(courseNumber)

IF course is NULL

PRINT "Course not found"

RETURN

PRINT course.courseNumber + ": " + course.courseTitle

IF course.prerequisites is empty

PRINT "No prerequisites"

ELSE

PRINT "Prerequisites: "

FOR each prereq in course.prerequisites

PRINT prereq + " "

FUNCTION PrintAllCourses():

CALL InOrderTraversal(courseTree.root)

END function

FUNCTION InOrderTraversal(node):

IF node is NULL

RETURN

InOrderTraversal(node.left)

PRINT node.course.courseNumber + ": " + node.course.courseTitle

InOrderTraversal(node.right)

END function

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open file** | 1 | 1 | 1 |
| **Read Each Line** | 1 | n | n |
| **Split each line** | 1 | n | n |
| **Validate Each line prereqs** | Log(n) | n | Nlog(n) |
| **Create Course Object** | 1 | n | n |
| **Insert into vector** | Log(n) | n | Nlog(n) |
| **Total Cost** | | | 2n log n + 3n +1 |
| **Runtime** | | | O(n log n) |

Function Menu()

Declare choice

Declare dataLoaded = False

Do

Print "1. Load Data"

Print "2. Print Course List (Alphanumeric)"

Print "3. Print Course Details"

Print "9. Exit"

Input choice

If choice == 1 Then

LoadCourses\_[Structure]() // Vector / HashTable / BST

dataLoaded = True

Else If choice == 2 Then

If dataLoaded == False

Print "Load data first"

Else

PrintSortedCourses\_[Structure]()

Else If choice == 3 Then

If dataLoaded == False

Print "Load data first"

Else

Input courseId

PrintCourse\_[Structure](courseId)

Else If choice == 9 Then

Print "Exiting..."

Else

Print "Invalid choice"

While choice != 9

EndFunction

**Advantages and disadvantages of each data structure**

Vectors- Vectors offer simple implementation and have built in sort functions that can allow for seamless data printing and manipulation. They offer fast insertion and are easy to iterate . However, when it comes to searching and validating course prerequisites, vectors must use a linear search and can lead to long runtimes when the course you are searching for is not present in the list. The validation is expensive because it must search the whole list. Working with large data can be very slow and performance will decrease with size.

Hash Tables- Hash tables offer fast insertion and fast search times with 0(1) average search complexity. Hash tables make it easy for validation because it utilizes keys to check for prerequisites so the lookup will be fast. Hash tables are scalable and can handle large datasets. However, hash tables need extra implementation to allow ordering, which can take more time. They can become complex when collisions occur within the table, this gives you more scenarios to have to handle. Buckets of the hash tables may need resizing depending on the data, some buckets may be filled which can cause high memory usage. When collisions occur, this can have a worst-case scenario of O(n).

Binary Search tree – Binary search trees sort data naturally using the right and left branches, lesser values on the left and greater values on the right. This allows for fast and efficient searching and insertion because you can eliminate half of the data set based on what you are looking for. The average search case is O(log n) which is also a faster validation than the vector which is only O(n). The trees, however, must be balanced to receive better performance. The trees are more complex to implement because they require pointer nodes as well as being able to differentiate smaller and larger numbers throughout each branch. Trees are also naturally ordered through traversals down the left and right branches which can make sorting easy.

In conclusion, for the requirements of this project which involve fast insertion and fast lookup times for searching courses and inserting them into your data structure, the best structure to utilize would be the Binary Search Tree. The binary search tree naturally orders itself through traversals having larger values on the right branch of a parent and smaller values on the left branch. Search times are efficient because half of the dataset can be eliminated based on the key value and insertion can be done the same just by updating pointers to the parent and child branches. Validation is also seamless, just not as fast as the hash table because hash tables utilize a key value pair which can validate your course quickly