# Leo Prichard

CS-300

12/07/2024

# CS 300 Project 1 Document

**//Vector - Milestone 1**

**//Define Course Data Structure (Course Object)**

STRUCT Course

STRING courseNumber

STRING title

LIST prerequisites

END STRUCT

**//Open the File and Load Data**

FUNCTION loadDataFromFile(filename)

OPEN the file with the given filename

CourseList = []

WHILE there is a line to read in the file

line = READ next line

IF line is not empty

tokens = SPLIT line by commas

courseNumber = tokens[0]

title = tokens[1]

prerequisites = tokens[2 to end of tokens]

course = NEW Course(courseNumber, title, prerequisites)

CourseList.ADD(course)

END IF

END WHILE

RETURN CourseList

END FUNCTION

**// Validate the Data**

FUNCTION validateData(CourseList)

FOR each course in CourseList

IF course.courseNumber is empty OR course.title is empty

PRINT "Error: Course missing course number or title."

EXIT program

END IF

FOR each prerequisite in course.prerequisites

validPrerequisiteFound = FALSE

FOR each otherCourse in CourseList

IF prerequisite equals otherCourse.courseNumber

validPrerequisiteFound = TRUE

BREAK

END IF

END FOR

IF NOT validPrerequisiteFound

PRINT "Error: Invalid prerequisite for course " + course.courseNumber

EXIT program

END IF

END FOR

END FOR

END FUNCTION

**//Search for a Course and Print Information**

FUNCTION searchAndPrintCourse(CourseList, searchCourseNumber)

FOR each course in CourseList

IF course.courseNumber equals searchCourseNumber

PRINT "Course Number: " + course.courseNumber

PRINT "Course Title: " + course.title

IF course.prerequisites is not empty

PRINT "Prerequisites: " + course.prerequisites

ELSE

PRINT "No prerequisites."

END IF

RETURN

END IF

END FOR

PRINT "Course not found."

END FUNCTION

**//Main Program Flow**

FUNCTION main()

filename = "courses.txt"

CourseList = loadDataFromFile(filename)

validateData(CourseList)

PRINT "Enter a course number to search:"

searchCourseNumber = GET user input

searchAndPrintCourse(CourseList, searchCourseNumber)

END FUNCTION

**//Hash Table - Milestone 2**

**// Open the File**

OPEN the file "courses.txt" for reading

IF file does not exist

PRINT "Error: File not found"

EXIT program

ENDIF

**// Read and Parse Each Line**

FOR each line in the file

REMOVE whitespace from the beginning and end of the line

SPLIT the line by spaces or commas into a list of tokens

IF the number of tokens is less than 2

PRINT "Error: Missing parameters in the course line"

CONTINUE to next line

ENDIF

**// Check for File Format Errors**

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2:]

IF not isValidCourseNumber(courseNumber)

PRINT "Error: Invalid course number " + courseNumber

CONTINUE to next line

ENDIF

FOR each prerequisite in prerequisites

IF not existsInHashTable(prerequisite)

PRINT "Error: Prerequisite " + prerequisite + " not found for course " + courseNumber

CONTINUE to next line

ENDIF

ENDFOR

**// Create Course Object**

CREATE courseObject with:`

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisites = tokens[2:]

**// Store Course Object in Hash Table**

addCourseToHashTable(courseObject)

END FOR

**// Validate Prerequisites**

FUNCTION existsInHashTable(courseNumber)

IF HASH\_TABLE contains courseNumber

RETURN True

ELSE

RETURN False

ENDFunction

**// Print Course Information**

PRINT "Course Information:"

FOR each course in HASH\_TABLE

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

IF course has prerequisites

PRINT "Prerequisites: " + course.prerequisites

ELSE

PRINT "No prerequisites"

ENDIF

ENDFOR

**//Binary Search Tree – Milestone 3**

**// Opening and reading the file**

OPEN "course\_data.csv" AS file // Open the CSV file containing the course data

IF file DOES NOT EXIST

PRINT "File not found!"

EXIT PROGRAM

WHILE NOT EOF(file) // Read the file until the end

LINE = READ NEXT LINE FROM file // Read each line

PROCESS LINE

END WHILE

**// Validating the file format**

FUNCTION validateLine(LINE):

SPLIT LINE INTO tokens USING "," AS DELIMITER // Split the line by commas

IF NUMBER OF tokens < 2

PRINT "ERROR: Line does not contain enough parameters."

RETURN FALSE

course\_number = tokens[0] // First token is course number

course\_name = tokens[1] // Second token is course name

IF COURSE\_NUMBER IS NOT VALID(course\_number)

PRINT "ERROR: Invalid course number: " + course\_number

RETURN FALSE

prerequisites = tokens[2:] // All remaining tokens are considered prerequisites

FOR each prerequisite IN prerequisites:

IF prerequisite IS NOT A VALID COURSE(course\_number)

PRINT "ERROR: Prerequisite does not exist: " + prerequisite

RETURN FALSE

END FOR

RETURN TRUE

**// Creating course objects**

STRUCT Course:

STRING course\_number

STRING course\_name

LIST prerequisites

FUNCTION createCourseObject(LINE):

SPLIT LINE INTO tokens USING "," AS DELIMITER // Split the line by commas

course\_number = tokens[0] // First token is course number

course\_name = tokens[1] // Second token is course name

prerequisites = tokens[2:] // All remaining tokens are prerequisites

course = NEW Course(course\_number, course\_name, prerequisites) // Create a course object

RETURN course

**// Storing the course objects in a binary search tree (BST)**

STRUCT Node:

Course course

Node left\_child

Node right\_child

STRUCT BinarySearchTree:

Node root

FUNCTION insertCourse(tree, course):

IF tree.root IS NULL

tree.root = NEW Node(course)

ELSE

INSERT COURSE INTO BST(tree.root, course)

FUNCTION insertCourseInBST(node, course):

IF course.course\_number < node.course.course\_number

IF node.left\_child IS NULL

node.left\_child = NEW Node(course)

ELSE

insertCourseInBST(node.left\_child, course)

ELSE IF course.course\_number > node.course.course\_number

IF node.right\_child IS NULL

node.right\_child = NEW Node(course)

ELSE

insertCourseInBST(node.right\_child, course)

**// Printing course information and prerequisites**

FUNCTION printCourseInformation(node):

IF node IS NOT NULL

PRINT "Course Number: " + node.course.course\_number

PRINT "Course Name: " + node.course.course\_name

IF node.course.prerequisites IS NOT EMPTY

PRINT "Prerequisites: " + CONCATENATE node.course.prerequisites

ELSE

PRINT "No prerequisites."

**// Recursively print courses in the left and right subtrees**

printCourseInformation(node.left\_child)

printCourseInformation(node.right\_child)

FUNCTION printAllCourses(tree):

PRINT "Course Information:"

IF tree.root IS NULL

PRINT "No courses available."

ELSE

printCourseInformation(tree.root)

**//Menu – Project 1**

**// Menu System to interact with the program**

WHILE true

PRINT "Menu: "

PRINT "1: Load Course Data"

PRINT "2: Print All Courses"

PRINT "3: Print Course Details"

PRINT "9: Exit"

userChoice = GET user input

SWITCH userChoice

CASE 1:

CALL LoadCourseData()

BREAK

CASE 2:

CALL PrintSortedCourses()

BREAK

CASE 3:

courseNumber = GET user input

CALL searchCourse(courses, courseNumber) // Use the respective data structure (Vector, HashTable, Tree)

BREAK

CASE 9:

PRINT "Exiting the program"

EXIT

DEFAULT:

PRINT "Invalid choice"

END SWITCH

END WHILE

**//Course Sorting – Project 1**

**// Function to print sorted courses alphabetically (course number order)**

void PrintSortedCourses(Vector<Course> courses) {

SORT courses by courseNumber (alphanumeric order)

FOR each course IN courses

PRINT course.courseNumber + " - " + course.title

END FOR

}

void PrintSortedCourses(HashTable<Course> courses) {

CREATE an empty list 'coursesList'

FOR each course IN courses

ADD course to 'coursesList'

END FOR

SORT 'coursesList' by courseNumber (alphanumeric order)

FOR each course IN coursesList

PRINT course.courseNumber + " - " + course.title

END FOR

}

void PrintSortedCourses(Tree<Course> courses) {

IN-ORDER TRAVERSAL of the binary search tree

FOR each course IN in-order traversal

PRINT course.courseNumber + " - " + course.title

END FOR

}

**//Vector – Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For opening the file** | 1 | 1 | 1 |
| **For reading each line** | 1 | n | n |
| **For parsing and creating a course object** | 1 | n | n |
| **Total Cost** | | | 2n + 1 |
| **Runtime** | | | O(n) |

**//Hash Table – Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For opening the file** | 1 | 1 | 1 |
| **For reading each line** | 1 | n | n |
| **For parsing and creating a course object** | 1 | n | n |
| **For inserting into hash table** | 1 | n | n |
| **Total Cost** | | | 3n + 1 |
| **Runtime** | | | O(n) |

**//Binary Search Tree - Runtime Analysis**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **For opening the file** | 1 | n | 1 |
| **For reading each line** | 1 | n | n |
| **For parsing and creating a course object** | 1 | n | n |
| **For inserting into BST** | n | n | n^2 |
| **Total Cost** | | | n^2 + 2n + 1 |
| **Runtime** | | | O(n^2) |

**Advantages and Disadvantages**

**Vector**

Vectors are pretty simple to work with and don’t use up much memory, since they store elements in one continuous block. They're good when you just need to go through the list of courses in order. However, they aren’t great for searching. Like if you need to find a specific course, it takes O(n) time because you have to look at each course one by one. Checking if a course’s prerequisites are valid is also slow, with a time complexity of O(n^2), since you have to check every prerequisite against all courses. And if you want to print courses in order, you have to sort them, which takes O(n log n) time.

**Hash Table**

Hash tables are great when you need quick lookups. Searching for a course is super fast because it directly finds the course using its unique identifier. When it comes to validating prerequisites, hash tables are also efficient, because you can check if a prerequisite exists in constant time for each course. The downside is that hash tables don’t automatically keep things in order, so if you need to print courses in alphabetical order, you’d have to sort them afterward, which takes O(n log n) time. Also, hash tables use more memory, and if there are a lot of collisions (when two items hash to the same spot),

performance can suffer a bit.

**Binary Search Tree (BST)**

A binary search tree (BST) works well if you need to search for courses quickly and also want them sorted automatically. If the tree is balanced, you can search and validate prerequisites in O(log n) time, which is faster than a vector. The best part about BSTs is that they automatically keep elements in order, so printing courses in alphabetical order is easy, taking just O(n) time. The problem with BSTs is that they need to stay balanced. If they get unbalanced, search and insertion times could slow down to O(n). Plus, BSTs are a bit more complex to set up and use more memory because they need pointers to connect the nodes.

**Recommendation**

For this project, I’d recommend going with a hash table. It’s fast when it comes to searching for courses and validating prerequisites, both of which are key parts of the program. While sorting the courses for alphanumeric printing would take a little extra time (O(n log n)), the speed of lookups and validation makes it worth it. A binary search tree could be an option too, especially if you really want the courses in order all the time, but hash tables are easier to implement and perform better for searching and validating data. So, for this project, a hash table seems like the best fit.