# **6-2 Project One**

Brian Roche-Campos

Southern New Hampshire University

CS-300 DSA: Analysis and Design

Professor Jeff H. Sanford

June 2nd, 2023

**Project Explanation:**

As part of my Computer Science program at ABCU, I have been given an assignment to meet the requirements set by my academic advisor. After receiving approval for my pseudocode in a previous submission, I am now tasked with expanding it to fulfill two new objectives: printing a list of CS courses in alphanumeric order and displaying the title and prerequisites for a given course. To accomplish this, I will be creating pseudocode for the data structures I have previously explored: vector, hash table, and tree. This involves opening the input file, reading its contents, parsing each line, checking for formatting errors, and creating course objects to store the data. I will also write pseudocode for displaying course information and prerequisites and designing a menu with options to load the data structure, print the course list, print an individual course, and exit the program.

In addition to creating the program, I must also conduct a Big O analysis to determine which data structure is the most suitable for the task at hand. The analysis will evaluate the worst-case running time of each structure for reading the input file and creating course objects. After conducting the analysis, I will explain the advantages and disadvantages of each data structure and recommend the most efficient one to use in the program. Ultimately, my goal is to create a program that fulfills my academic advisor's requirements and utilizes the most appropriate data structure for optimal performance.

**(Please refer to the next page for the Pseudocode.)**

Here is the pseudocode for each data structure, starting with the Vector data structure.

**Vector:**

vector<Course> courseList

function readCourseData(file):

open file

for each line in file:

courseData = split(line, ",")

course = new Course(courseData[0], courseData[1:])

courseList.push\_back(course)

function printCourseList():

sort(courseList)

for course in courseList:

print(course.number)

function printCourse(courseNum):

for course in courseList:

if course.number == courseNum:

print(course.title)

print(course.prereqs)

**Explanation for Vector:**

1. **vector<Course> courseList:** This line declares a vector, which is an array-like data structure, of objects type **Course**. The purpose of this vector is to store all the courses managed by the program.
2. The function **readCourseData(file)** takes a filename as input and reads the required data from the file. Each line of the file is separated into a course number and its corresponding prerequisites, which are separated by commas, and a new **Course** object is then created using this information. The new course object is then added to the **courseList** vector.
3. The function **printCourseList()** arranges the **courseList** vector by sorting the courses according to their respective course numbers. The function prints the course numbers for every course present in the list.
4. The function **printCourse(courseNum)** works by taking in a course number as input and then searches the **courseList** vector for a **Course** object with a matching course number. In the event that a match is found, this function will print the course's title alongside its prerequisites.

**(Please refer to the next page for the HashTable.)**

**HashTable:**

hashTable<Course> courseTable

function readCourseData(file):

open file

for each line in file:

courseData = split(line, ",")

course = new Course(courseData[0], courseData[1:])

courseTable.insert(course.number, course)

function printCourseList():

courseList = courseTable.getKeys()

sort(courseList)

for courseNum in courseList:

print(courseNum)

function printCourse(courseNum):

course = courseTable.get(courseNum)

if course != null:

print(course.title)

print(course.prereqs)

The following pseudocode describes a **Hash Table** named "**courseTable**", where each course is stored with its number attribute as the key.

**Explanation for HashTable:**

1. The **readCourseData()** function reads course data from a file, creates a new **Course** object for each line, and inserts the course into **courseTable** using its number as the key.
2. The **printCourseList()** function retrieves all the keys from the **courseTable**, sorts them in ascending order, and prints each course number to the console.
3. The **printCourse()** function takes a **courseNum** parameter, retrieves the course object associated with that number from **courseTable**, and if the course exists, prints its **title** and **prerequisites** attributes to the console.

Additionally, this pseudocode uses a hash table to efficiently store and retrieve courses by their course number and provides two functions to print either a list of all course numbers or the details of a specific course given its number.

**(Please refer to the next page for the Tree.)**

**Tree is given below :**

class TreeNode:

Course course

TreeNode leftChild

TreeNode rightChild

class CourseTree:

TreeNode root

function insertCourse(node, course):

if node is null:

node = new TreeNode(course)

else if course.number < node.course.number:

node.leftChild = insertCourse(node.leftChild, course)

else:

node.rightChild = insertCourse(node.rightChild, course)

return node

function readCourseData(file):

open file

for each line in file:

courseData = split(line, ",")

course = new Course(courseData[0], courseData[1:])

root = insertCourse(root, course)

function printCourseList():

inOrderTraversal(root)

function inOrderTraversal(node):

if node is not null:

inOrderTraversal(node.leftChild)

print(node.course.number)

inOrderTraversal(node.rightChild)

function printCourse(courseNum):

node = findNode(root, courseNum)

if node is not null:

print(node.course.title)

print(node.course.prereqs)

function findNode(node, courseNum):

if node is null or node.course.number == courseNum:

return node

else if courseNum < node.course.number:

return findNode(node.leftChild, courseNum)

else:

return findNode(node.rightChild, courseNum)

**(Please refer to the next page.)**

**Explanation for Tree:**

1. The **insertCourse()** function is used to insert new Course objects into the binary search tree, following the binary search tree property where all nodes in the left subtree have keys less than the node's key and all nodes in the right subtree have keys greater than the node's key.
2. The **readCourseData()** function reads course data from a file and inserts each course into the binary search tree using the **insertCourse()** function.
3. The **printCourseList()** function performs an in-order traversal of the binary search tree to print out the course numbers in sorted order.
4. The **inOrderTraversal()** function is used to traverse the binary search tree in order.
5. The **printCourse()** function searches for a Course object with a given course number and prints its title and prerequisites. It uses the **findNode()** function to locate the TreeNode instance that contains the Course object.
6. The **findNode()** function searches the binary search tree for a node with a given course number. It follows the binary search tree property to traverse the left or right subtree as appropriate until the desired node is found or the search reaches a null node.

This implementation offers an efficient way to search and retrieve courses from a large collection of courses, with most operations having a time complexity of O(log n).

The pseudocode for opening the file, reading the data, and creating course objects would be the same for each data structure, so the Big O analysis would be the same for each.

**Pseudocode:**

function readCourseData(file):

open file

for each line in file:

courseData = split(line, ",")

course = new Course(courseData[0], courseData[1:])

// add course to data structure

The **readCourseData(file)** function is a generic pseudocode function that reads course data from a file and adds each course to a data structure.

**Evaluation:**

To analyze the worst-case running time of the pseudocode for reading the file and creating course objects, I need to examine the cost per line of code and the number of times each line will execute.

Reading the file:

a. Open file: cost = 1, number of times executed = 1

b. Loop over lines in file: cost = 1, number of times executed = number of lines in file

c. Close file: cost = 1, number of times executed = 1

The total cost for reading the file is 2 + number of lines in file.

Creating course objects:

a. Parse line: cost = 1, number of times executed = number of lines in file

b. Create course object: cost = 1, number of times executed = number of lines in file

c. Add course object to data structure: depends on the data structure used.

1 .**Vector:**

Adding an element to the end of a vector has a worst-case running time of O(1). However, if the vector needs to resize itself to accommodate the new element, the worst-case running time can be O(n). Thus, the worst-case running time for adding n elements to a vector is O(n^2).

Advantages:

Elements are stored contiguously in memory, making access fast.

Adding elements to the end of a vector is efficient.

Disadvantages:

Inserting or deleting elements in the middle of a vector can be slow.

Resizing a vector can be expensive.

2. **Hash table:**

Inserting an element into a hash table has a worst-case running time of O(1), assuming the hash function distributes the keys evenly. However, in the worst case, all the keys could hash to the same value, resulting in a worst-case running time of O(n).

Advantages:

Inserting and retrieving elements is fast, on average.

The size of the hash table can be adjusted dynamically.

Disadvantages:

Hashing collisions can slow down operations.

The order of elements is not preserved.

3. **Tree:**

Inserting an element into a balanced binary search tree has a worst-case running time of O(log n). However, if the tree becomes unbalanced, the worst-case running time can be O(n).

Advantages:

Inserting, retrieving, and deleting elements is fast, on average.

The order of elements is preserved.

Disadvantages:

The worst-case running time can be O(n) if the tree becomes unbalanced.

Trees can require more memory than other data structures.

**Based on the analysis, the recommended data structure is a hash table**.

Inserting and retrieving elements is fast, on average, and the size of the hash table can be adjusted dynamically. Although there is a risk of hashing collisions, this can be mitigated by using a good hash function and implementing collision resolution strategies, such as chaining or open addressing. The worst-case running time of inserting n elements into a hash table is O(n), which is better than the worst-case running time of O(n^2) for a vector and the worst-case running time of O(n log n) for a tree.