Vincent Noce

CS-300

11/29/2023  
  
 6-2 Project

**Pseudocode and Runtime Analysis**

A pseudocode and runtime analysis will play a crucial role in the development of ABCU's Computer Science academic advising program. The purpose of this project is to print an alphabetical listing of Computer Science courses, as well as to display the course title and prerequisites. In order to determine the most efficient data structure, it is necessary to design pseudocode systematically and then runtime analyze the result.

A straightforward approach is outlined for the vector data structure. Course objects are stored in the coursesVector, and functions are provided for storing data, printing course lists, and managing individual courses. Based on runtime analysis, it may be a reasonable choice for simplicity but may have limitations when dealing with larger datasets due to its linear time complexity (O(n)).

For efficient data retrieval, the coursesHashTable makes use of key-value pairs. Data can be read and loaded, course lists printed, and individual courses printed using the pseudocode provided. Based on the runtime analysis, key-based operations have constant time complexity (O(1)), which makes them highly efficient.

Course objects are arranged hierarchically in the tree data structure. This paper describes how to construct a CoursesTree, how to read and load data, and how to print course lists and individual courses. Nonetheless, due to the inherent sorting during insertion, the runtime analysis reveals an O(n log n) time complexity when reading and loading data.

**Analysis of Data Structures**

There are advantages and disadvantages to each data structure. It is easy to comprehend and implement the vector because of its simplicity. While it performs key operations in linear time, greater datasets may pose challenges. This type of operation is highly efficient due to its constant time complexity, but it must be carefully considered due to the possibility of collisions. Searching and ordering is made easier with the tree, but there is a compromise in terms of complexity and memory usage.

As a result of these considerations, the ABCU advising program recommends a hash table. In line with academic advisor's recommendations, it exhibits a constant time complexity that makes efficient retrieval and key-based operations possible. Using proper handling mechanisms can mitigate the potential drawbacks of collisions. In a real-world scenario with varying dataset sizes, efficiency is paramount, and this decision is substantiated by the understanding that efficiency is paramount.

Consequently, ABCU advising program coding will be enhanced by the pseudocode and runtime analysis. An informed decision aligned with the program's needs and efficiency goals is made by carefully examining data structures and their implications.

**Pseudocode for Vector Data Structure**

# Vector Data Structure Pseudocode

# Define a simple Course structure

struct Course:

course\_number

course\_title

prerequisites

# Define a vector data structure to store course objects

vector<Course> course\_vector

# Function to open a file

function open\_file(file\_path):

# Function to close a file

function close\_file(file):

# Function to read a line from a file

function read\_line(file):

# Function to parse a line and extract course data

function parse\_line(line):

# Split the line into tokens

tokens = split(line, ',')

# Check for at least two parameters on each line

if length(tokens) < 2:

print("Error: Insufficient parameters on line", line)

return null

# Extract course number, title, and prerequisites

course\_number = tokens[0].strip()

course\_title = tokens[1].strip()

prerequisites = [p.strip() for p in tokens[2:]]

return Course(course\_number, course\_title, prerequisites)

# Function to validate and process the file

Procedure LoadDataIntoVector(file\_path):

Try:

# Open the file

file = open\_file(file\_path)

# Loop through each line in the file

while not EOF(file):

# Read each line

line = read\_line(file)

# Parse the line and check for file format errors

course = parse\_line(line)

if course is not null:

# Check if prerequisites exist in the vector

for prerequisite in course.prerequisites:

if not exists\_in\_vector(course\_vector, prerequisite):

print("Error: Prerequisite not found for course", course.course\_number)

return # Exit if a prerequisite is not found

# Add the course object to the vector

course\_vector.push\_back(course)

# Close the file

close\_file(file)

Catch Exception as e:

print("Error: Unable to open or read file")

Exit the program

End Try

# Function to print course information

function print\_course\_information\_vector(course\_vector):

for each course in course\_vector:

print("Course Number:", course.course\_number)

print("Course Title:", course.course\_title)

print("Prerequisites:", course.prerequisites)

print("\n")

**Pseudocode for Hash Table Data Structure**

# Hash Table Data Structure Pseudocode (Updated)

class HashTable:

# Node structure

Node:

Course course

Node\* next

vector<Node> nodes

unsigned int table\_size

# Other private members and methods

# Constructor for initializing the hash table

HashTable(size):

table\_size = size

nodes.resize(table\_size)

# Method to calculate the hash value

unsigned int hash(course\_number):

return hash\_function(course\_number) % table\_size

# Method to insert a course into the hash table

void Insert(course):

key = hash(course.course\_number)

Node& node = nodes[key]

# If the slot is empty, add the course

if (node.course.course\_number == NULL):

node = Node(course)

else:

# Handle collisions by adding to the linked list

while (node.next != nullptr):

node = \*node.next

node.next = new Node(course, NULL)

# Method to print all courses

void PrintAll():

for each node in nodes:

while (node.course.course\_number != NULL):

# Output course number, title, and prerequisites

# Move to the next node in case of collisions

if (node.next != nullptr):

node = \*node.next

else:

break

# Method to remove a course by number

void Remove(course\_number):

key = hash(course\_number)

Node& node = nodes[key]

# If the course exists, remove it

if (node.course.course\_number != NULL):

nodes[key] = Node()

# Method to search for a course by number

Course Search(course\_number):

key = hash(course\_number)

Node& node = nodes[key]

# Search for the course in the linked list

while (node.course.course\_number != NULL):

if (node.course.course\_number == course\_number):

return node.course

if (node.next != nullptr):

node = \*node.next

else:

break

return NULL # Return NULL if not found

**Pseudocode for Tree Data Structure:**

# Tree Data Structure Pseudocode (Updated)

# Define a simple Course structure

struct Course:

course\_number

course\_title

prerequisites

# Define a tree data structure to store course objects

struct TreeNode:

Course course

TreeNode\* left

TreeNode\* right

TreeNode\* root = NULL

# Function to open a file

function open\_file(file\_path):

# Function to close a file

function close\_file(file):

# Implementation to close the file

# Example in Python: file.close()

# Function to read a line from a file

function read\_line(file):

# Function to parse a line and extract course data

function parse\_line(line):

# Split the line into tokens

tokens = split(line, ',')

# Check for at least two parameters on each line

if length(tokens) < 2:

print("Error: Insufficient parameters on line", line)

return null

# Extract course number, title, and prerequisites

course\_number = tokens[0].strip()

course\_title = tokens[1].strip()

prerequisites = [p.strip() for p in tokens[2:]]

return Course(course\_number, course\_title, prerequisites)

# Function to validate and process the file

Procedure LoadDataIntoTree(file\_path):

Try:

# Open the file

file = open\_file(file\_path)

# Loop through each line in the file

while not EOF(file):

# Read each line

line = read\_line(file)

# Parse the line and check for file format errors

course = parse\_line(line)

if course is not null:

# Check if prerequisites exist in the tree

for prerequisite in course.prerequisites:

if not exists\_in\_tree(root, prerequisite):

print("Error: Prerequisite not found for course", course.course\_number)

return # Exit if a prerequisite is not found

# Add the course object to the tree

root = insert\_into\_tree(root, course)

# Close the file

close\_file(file)

Catch Exception as e:

print("Error: Unable to open or read file")

Exit the program

End Try

# Function to print course information for tree data structures

function print\_course\_information\_tree(root):

in\_order\_traversal(root)

# Function for in-order traversal

function in\_order\_traversal(root):

if root is not null:

in\_order\_traversal(root.left)

print("Course:", root.data.course\_number, "-", root.data.course\_title)

if root.data.prerequisites:

print("Prerequisites:", ', '.join(root.data.prerequisites))

else:

print("Prerequisites: None")

print("\n")

in\_order\_traversal(root.right)

# Function to check if a course exists in the tree

function exists\_in\_tree(root, course\_number):

if root is null:

return false

if root.data.course\_number == course\_number:

return true

if course\_number < root.data.course\_number:

return exists\_in\_tree(root.left, course\_number)

else:

return exists\_in\_tree(root.right, course\_number)

# Function to insert a course into the tree

function insert\_into\_tree(root, course):

if root is null:

return new TreeNode(course)

if course.course\_number < root.data.course\_number:

root.left = insert\_into\_tree(root.left, course)

else:

root.right = insert\_into\_tree(root.right, course)

return root

**Runtime Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Vector** | **Hash Table** | **Tree** |
| Data reading and loading from files | O(n) | O(n) | O(n log n) |
| Print Course List | O(n log n) | O(n log n) | O(n) |
| Print Course | O(n) | O(1) | O(log n) |

**Analysis of Data Structures**

**Vector**

-Implementation is simple and easy to understand.

-As a result of linear time complexity, searching and insertion are slower.

**Hash Table**

-With key-based operations, you can access and retrieve data faster, with a constant time complexity.

-Collisions are possible, which require handling.

**Tree**

-Efficacious for searching and ordering data.

-Complexity and memory consumption are disadvantages of this structure.

**Recommendation:**

As a result of the requirements and Big O analysis, it is recommended to use a Hash Table. Aside from displaying and searching course information efficiently, the program is aligned with the program's objectives. This scenario is a good fit for it because of its constant time complexity.