# Gage Sovey

CS300: Project 1

Prof. Lusby

10/6/2024

# Project 1: Pseudocode

1. **Resubmit pseudocode from previous pseudocode assignments and update as necessary.**

**Vector Pseudocode**

Begin Program

Define Structure Course:

String courseNumber

String title

Vector prerequisites

Define Vector courses // To store all Course objects

// Open and validate the file

Function LoadCoursesFromFile(filePath):

Open file at filePath

If file cannot be opened:

Print "Error: File cannot be opened."

Exit function

While not end of file:

Read a line from the file

Split line by commas into courseData array

// Validate the format

If length of courseData is less than 2:

Print "Error: Invalid format, less than 2 parameters."

Continue to the next line

// Extract course number and title

String courseNumber = courseData[0]

String courseTitle = courseData[1]

// Create a new Course object

Define newCourse as Course

Set newCourse.courseNumber = courseNumber

Set newCourse.title = courseTitle

// Parse and validate prerequisites

For i from 2 to length of courseData:

String prereq = courseData[i]

If prereq is not found in existing course numbers:

Print "Error: Prerequisite " + prereq + " not found in courses."

Continue to the next line

Add prereq to newCourse.prerequisites

// Add newCourse to courses vector

Add newCourse to courses vector

Close the file

// Search for a course by course number and print its information

Function PrintCourseInfo(courseNumber):

For each course in courses:

If course.courseNumber equals courseNumber:

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.title

If course.prerequisites is not empty:

Print "Prerequisites: "

For each prereq in course.prerequisites:

Print prereq

Else:

Print "No prerequisites."

Exit function

Print "Course not found."

// Main function to control program flow

Function Main():

// Load courses from the file

Call LoadCoursesFromFile("course\_data.txt")

// Prompt user to search for a course

Print "Enter a course number to search: "

Input userCourseNumber

Call PrintCourseInfo(userCourseNumber)

Call Main()

End Program

**Hash Table Pseudocode**

Begin Program

Define Structure Course:

String courseNumber

String title

Vector prerequisites

Define HashTable courseTable // To store Course objects with courseNumber as the key

// Open and validate the file

Function LoadCoursesFromFile(filePath):

Open file at filePath

If file cannot be opened:

Print "Error: File cannot be opened."

Exit function

While not end of file:

Read a line from the file

Split line by commas into courseData array

// Validate the format

If length of courseData is less than 2:

Print "Error: Invalid format, less than 2 parameters."

Continue to the next line

// Extract course number and title

String courseNumber = courseData[0]

String courseTitle = courseData[1]

// Create a new Course object

Define newCourse as Course

Set newCourse.courseNumber = courseNumber

Set newCourse.title = courseTitle

// Parse and validate prerequisites

For i from 2 to length of courseData:

String prereq = courseData[i]

If prereq is not found in courseTable:

Print "Error: Prerequisite " + prereq + " not found in courses."

Continue to the next line

Add prereq to newCourse.prerequisites

// Add newCourse to courseTable with courseNumber as the key

courseTable[courseNumber] = newCourse

Close the file

// Search for a course in the hash table and print its information

Function PrintCourseInfo(courseNumber):

If courseNumber exists in courseTable:

Set course = courseTable[courseNumber]

Print "Course Number: " + course.courseNumber

Print "Course Title: " + course.title

If course.prerequisites is not empty:

Print "Prerequisites: "

For each prereq in course.prerequisites:

Print prereq

Else:

Print "No prerequisites."

Else:

Print "Course not found."

// Main function to control program flow

Function Main():

// Load courses from the file

Call LoadCoursesFromFile("course\_data.txt")

// Prompt user to search for a course

Print "Enter a course number to search: "

Input userCourseNumber

Call PrintCourseInfo(userCourseNumber)

Call Main()

End Program

**BinarySearchTree Pseudocode**

START

OPEN file "courseData.txt" FOR reading

IF file cannot be opened THEN

PRINT "Error: Unable to open file"

EXIT program

END IF

INITIALIZE binary search tree (BST) with root = NULL

WHILE there are lines to read in the file DO

READ currentLine

SPLIT currentLine into tokens (courseNumber, courseTitle, prerequisites)

IF number of tokens is less than 2 THEN

PRINT "Error: Invalid file format. Each line must contain at least a course number and title."

CONTINUE to next line

END IF

FOR each prerequisite in the list of prerequisites DO

IF prerequisite does not exist in any previous course lines THEN

PRINT "Error: Prerequisite " + prerequisite + " does not exist as a course."

CONTINUE to next line

END IF

END FOR

CREATE new Course object

SET Course.courseNumber = token[0]

SET Course.courseTitle = token[1]

IF prerequisites exist THEN

ADD prerequisites to the Course object

END IF

IF root is NULL THEN

SET root = new Node(Course)

ELSE

SET currentNode = root

WHILE currentNode is not NULL DO

IF Course.courseNumber < currentNode.course.courseNumber THEN

IF currentNode.left is NULL THEN

SET currentNode.left = new Node(Course)

BREAK

ELSE

SET currentNode = currentNode.left

END IF

ELSE

IF currentNode.right is NULL THEN

SET currentNode.right = new Node(Course)

BREAK

ELSE

SET currentNode = currentNode.right

END IF

END IF

END WHILE

END IF

END WHILE

CLOSE file

PRINT "Courses Loaded into the Binary Search Tree"

DEFINE FUNCTION inOrder(node)

IF node is not NULL THEN

CALL inOrder(node.left)

PRINT "Course Number: " + node.course.courseNumber

PRINT "Course Title: " + node.course.courseTitle

IF node.course.prerequisites is not empty THEN

PRINT "Prerequisites: " + node.course.prerequisites

ELSE

PRINT "No Prerequisites"

END IF

CALL inOrder(node.right)

END IF

END FUNCTION

CALL inOrder(root)

END

1. **Create pseudocode for a menu**

BEGIN Program

// Define variables

Define dataStructure // The chosen data structure (e.g., hash table)

Define userOption // To store user input for menu option

Define isDataLoaded as FALSE // To check if data has been loaded

// Function to Load Data

Function LoadData():

IF isDataLoaded is TRUE THEN

PRINT "Data has already been loaded."

RETURN

END IF

// Load the data from file into dataStructure

CALL LoadCoursesFromFile("courseData.txt", dataStructure)

isDataLoaded = TRUE

PRINT "Data loaded successfully."

END Function

// Function to Print Alphanumerically Ordered List of Courses

Function PrintAllCourses():

IF isDataLoaded is FALSE THEN

PRINT "Please load the data first."

RETURN

END IF

// Print courses in alphanumerical order

CALL PrintCoursesInOrder(dataStructure)

END Function

// Function to Print Course Information

Function PrintCourseInfo():

IF isDataLoaded is FALSE THEN

PRINT "Please load the data first."

RETURN

END IF

PRINT "Enter course number: "

INPUT userCourseNumber

// Search and print course information

CALL PrintCourseDetails(userCourseNumber, dataStructure)

END Function

// Function to Display Menu

Function DisplayMenu():

PRINT "Menu:"

PRINT "1. Load Course Data"

PRINT "2. Print All Courses in Alphanumeric Order"

PRINT "3. Print Course Information"

PRINT "9. Exit"

END Function

// Main Program Loop

WHILE TRUE DO

CALL DisplayMenu()

PRINT "Enter your option: "

INPUT userOption

IF userOption == 1 THEN

CALL LoadData()

ELSE IF userOption == 2 THEN

CALL PrintAllCourses()

ELSE IF userOption == 3 THEN

CALL PrintCourseInfo()

ELSE IF userOption == 9 THEN

PRINT "Exiting program."

BREAK

ELSE

PRINT "Invalid option, please try again."

END IF

END WHILE

END Program

1. **Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order**
2. **Vector Data Structure**

Function PrintCoursesInOrder\_Vector(courses):

// Sort the vector based on course numbers (alphanumeric order)

CALL Sort(courses by course.courseNumber)

// Iterate over the sorted vector and print each course

FOR each course in courses DO

PRINT "Course Number: " + course.courseNumber + ", Course Title: " + course.title

END FOR

END Function

1. **Hash Table Data Structure**

Function PrintCoursesInOrder\_HashTable(courseTable):

// Create an empty list to hold courses

Define courseList as empty list

// Extract all courses from the hash table into the courseList

FOR each key in courseTable DO

ADD courseTable[key] to courseList

END FOR

// Sort the list of courses based on course numbers (alphanumeric order)

CALL Sort(courseList by course.courseNumber)

// Iterate over the sorted courseList and print each course

FOR each course in courseList DO

PRINT "Course Number: " + course.courseNumber + ", Course Title: " + course.title

END FOR

END Function

1. **Binary Search Tree Data Structure**

Function PrintCoursesInOrder\_BST(node):

// Perform an in-order traversal to print courses in sorted order

IF node is not NULL THEN

CALL PrintCoursesInOrder\_BST(node.left)

PRINT "Course Number: " + node.course.courseNumber + ", Course Title: " + node.course.courseTitle

CALL PrintCoursesInOrder\_BST(node.right)

END IF

END Function

# Project 1: Evaluation

**4. Evaluate the run time and memory of data structures that could be used to address the requirements.**

**1. Vector - Revised Runtime Analysis (Best Case & Worst Case)**

| **Code** | **Line Cost** | **#Times Executes (Best Case)** | **Total Cost (Best Case)** | **#Times Executes (Worst Case)** | **Total Cost (Worst Case)** |
| --- | --- | --- | --- | --- | --- |
| For All Courses | 1 | n | n | n | n |
| If the course is the same as courseNumber | 1 | n | n | n | n |
| Print out the course information | 1 | 1 | 1 | 1 | 1 |
| For each prerequisite of the course | 1 | 0 (best) | 0 | p (worst) | p |
| Print the prerequisite course info | 1 | 0 (best) | 0 | p (worst) | p |
|  |  |  |  |  |  |
| **Total Cost** |  |  | **2n + 1** |  | **2n + 2p** |
| **Runtime** |  |  | **O(n)** |  | **O(n + p)** |

**2. Hash Table - Revised Runtime Analysis (Best Case & Worst Case)**

| **Code** | **Line Cost** | **#Times Executes (Best Case)** | **Total Cost (Best Case)** | **#Times Executes (Worst Case)** | **Total Cost (Worst Case)** |
| --- | --- | --- | --- | --- | --- |
| For All Courses | 1 | n | n | n | n |
| If the course is the same as courseNumber | 1 | n | n | n | n |
| Print out the course information | 1 | 1 | 1 | 1 | 1 |
| For each prerequisite of the course | 1 | 0 (best) | 0 | p (worst) | p |
| Print the prerequisite course info | 1 | 0 (best) | 0 | p (worst) | p |
|  |  |  |  |  |  |
| **Total Cost** |  |  | **2n + 1** |  | **2n + 2p** |
| **Runtime** |  |  | **O(n)** |  | **O(n + p)** |

**3. Binary Search Tree (BST) - Revised Runtime Analysis (Best Case & Worst Case)**

| **Code** | **Line Cost** | **#Times Executes (Best Case)** | **Total Cost (Best Case)** | **#Times Executes (Worst Case)** | **Total Cost (Worst Case)** |
| --- | --- | --- | --- | --- | --- |
| For All Courses | 1 | n | n | n | n |
| If the course is the same as courseNumber | log(n) | log(n) \* n | n log(n) | n | n |
| Print out the course information | 1 | 1 | 1 | 1 | 1 |
| For each prerequisite of the course | 1 | 0 (best) | 0 | p (worst) | p |
| Print the prerequisite course info | 1 | 0 (best) | 0 | p (worst) | p |
|  |  |  |  |  |  |
| **Total Cost** |  |  | **n log(n) + 1** |  | **n + 2p** |
| **Runtime** |  |  | **O(n log n)** |  | **O(n + p)** |

**4. Breakdown by Course (Explaining p)**

| **Course** | **Number of Prerequisites (p)** |
| --- | --- |
| **CSCI100** | 0 |
| **CSCI101** | 1 |
| **CSCI200** | 1 |
| **MATH201** | 0 |
| **CSCI300** | 2 |
| **CSCI301** | 1 |
| **CSCI350** | 1 |
| **CSCI400** | 2 |

**Explanation of Best Case and Worst Case:**

* **Best Case**:
  + For the **vector** and **hash table**, the best case occurs when each course has no prerequisites, meaning there’s no additional overhead for prerequisite processing (p = 0).
  + For the **binary search tree**, the best case is when the tree is balanced, leading to logarithmic search time (log(n)).
* **Worst Case**:
  + For the **vector** and **hash table**, the worst case occurs when each course has multiple prerequisites, increasing the processing time for validating and printing prerequisites (p > 0).
  + For the **binary search tree**, the worst case happens when the tree is unbalanced, causing search times to degrade to linear time (O(n)).

**Summary of Runtime Complexities (Best Case vs Worst Case):**

1. **Vector**:
   * Best Case: **O(n)**
   * Worst Case: **O(n + p)**
2. **Hash Table**:
   * Best Case: **O(n)**
   * Worst Case: **O(n + p)**
3. **Binary Search Tree**:
   * Best Case: **O(n log n)**
   * Worst Case: **O(n + p)**

**5. Explain the advantages and disadvantages of each structure in your evaluation.**

**1. Vector Data Structure**

**Advantages:**

- Simple and Easy to Use: Vectors provide a straightforward way to store and retrieve courses sequentially. This simplicity makes them easy to implement.

- Efficient for Iteration: Vectors are efficient for iterating over a list of courses in insertion order, especially when the dataset is relatively small.

- Low Memory Overhead: Vectors use contiguous memory blocks, making them more memory-efficient compared to more complex structures like hash tables and trees.

**Disadvantages:**

- Linear Search Time: Searching for a course requires checking each element one by one, leading to linear time complexity (O(n)) in the worst case.

- Not Suitable for Large Datasets: As the dataset grows, searching and updating data becomes increasingly inefficient due to the sequential access pattern.

- Inefficient Insertion in Order: If order matters, inserting a course at a specific point requires a full scan or reordering, increasing the cost.

**2. Hash Table Data Structure**

**Advantages:**

- Fast Lookups: Hash tables offer constant-time lookups on average, allowing courses to be retrieved quickly based on their course numbers (O(1) in the best case).

- Efficient for Insertion: Inserting courses into a hash table takes constant time, provided there are no hash collisions.

- Scalability: Hash tables perform well even with large datasets, making them ideal for scaling.

**Disadvantages:**

- Potential for Collisions: Hash collisions can degrade the performance, leading to O(n) in the worst case. While rehashing or chaining can mitigate collisions, they add complexity and overhead.

- No Ordering: Hash tables do not maintain the order of insertion, which can be a disadvantage if the advisor expects courses to be listed in a specific sequence.

- Memory Overhead: Hash tables require extra memory for handling hash codes and collisions, leading to a larger memory footprint compared to simpler structures like vectors.

**3. Binary Search Tree (BST) Data Structure**

**Advantages:**

- Efficient Search in Balanced Trees: In a balanced binary search tree, searching, inserting, and deleting courses can be done in O(log n) time, making it more efficient than vectors for larger datasets.

- Maintains Sorted Order: Binary search trees naturally maintain the order of the courses based on their course numbers, which could be useful if order is important.

- Scalable: A balanced BST is scalable and can handle moderate to large datasets efficiently.

**Disadvantages:**

- Risk of Becoming Unbalanced: In the worst case, the binary search tree can become unbalanced, degrading the performance to O(n), similar to a vector. Maintaining balance requires more complex self-balancing techniques (e.g., AVL or Red-Black trees).

- More Complex Implementation: Compared to vectors and hash tables, binary search trees are more complex to implement, particularly if self-balancing techniques are used.

- Memory Overhead: Each node in the binary search tree requires extra memory for pointers to its left and right children, which increases memory usage compared to a vector.

**6. make a recommendation for which data structure you plan to use in your code**

Based on the analysis of the three data structures (vector, hash table, and binary search tree), I recommend using the hash table data structure for the following reasons:

1. **Superior Search and Insertion Performance:**

Hash tables offer constant-time (O(1)) average-case lookup and insertion performance, making them the most efficient choice for retrieving courses based on their course numbers. In this project, fast lookups are essential for checking prerequisites and finding course information. With the hash table, the time complexity remains constant even as the dataset grows, which ensures scalability and efficiency.

2. **Handling Prerequisites Efficiently:**

With prerequisites being a key component of the course structure, the ability to quickly validate and retrieve courses by their course number is crucial. The hash table's constant-time search provides a clear advantage over vectors and binary search trees, both of which require more time as the dataset grows.

3. **Scalability:**

As the number of courses increases, the hash table continues to provide fast access to data without the performance degradation seen in vectors (O(n)) or unbalanced binary search trees (which degrade to O(n) in the worst case). This makes the hash table suitable for larger datasets.

4. **Simple and Practical for Unordered Data:**

Since the advisor's requirements seem to focus on searching for courses by course number, maintaining an ordered list is not a priority. A hash table is ideal in this scenario because it doesn't store data in a sorted order, which simplifies its implementation without sacrificing performance.

5. **Drawbacks Mitigated by Good Hashing:**

While hash tables can suffer from collisions (which degrade performance to O(n)), this issue can be mitigated by using a good hash function and implementing collision handling techniques like chaining or open addressing. These methods help ensure that the hash table retains its constant-time performance even with larger datasets.

**Why Not Vector or Binary Search Tree?**

- Vector: Although vectors are simple to implement, their linear search time (O(n)) makes them inefficient for large datasets, particularly when fast lookups are required. This makes vectors less suitable for scenarios where quick retrieval and prerequisite checking are important.

- Binary Search Tree: While a balanced binary search tree can provide logarithmic time complexity (O(log n)), the risk of the tree becoming unbalanced introduces a significant downside, where performance degrades to O(n). Additionally, binary search trees are more complex to implement and maintain compared to hash tables, especially if self-balancing techniques are required.

**Conclusion:**

The hash table is the best choice for this project due to its fast average-case lookup and insertion times, scalability, and ability to efficiently handle prerequisites. Its performance advantages outweigh the complexity of managing collisions, making it the most suitable data structure for the advisor's requirements.