Joseph Roberts

CS -300: Analysis and Design

6-2: Project

2-10-2025

**MENU PSEUDOCODE**

**VECTOR PSEUDOCODE**

**HASH TABLE PSEUDOCODE**

**BINARY SEARCH TREE PSEUDOCODE**

**TIME COMPLEXITY ANALYSIS**

**RECOMMENDATIONS**

**Menu Pseudocode**

// Start program

START program

// Display menu and get choice

FUNCTION displayMenu():

PRINT "1. Load data"

PRINT "2. Print all courses"

PRINT "3. Print course info"

PRINT "9. Exit"

RETURN input choice

// Load data

FUNCTION loadCourseData():

OPEN file "course\_data.txt"

INITIALIZE 'courseTable' and 'courseNumbers'

WHILE NOT end of file:

READ line, split by ","

IF length < 2:

PRINT "Error"

ELSE:

courseNumber, courseTitle = components[0], components[1]

prerequisites = components[2 to end]

ADD courseNumber to 'courseNumbers'

IF all(prerequisite in 'courseNumbers' for prerequisite in prerequisites):

CREATE course, add to 'courseTable'

close file

PRINT "Data loaded"

// Print all courses

FUNCTION printAllCourses():

IF 'courseTable' is empty:

PRINT "Load data first"

ELSE:

FOR courseNumber in sort(keys of 'courseTable'):

print courseNumber, courseTable[courseNumber].courseTitle

// Print course info

FUNCTION printCourseInfo():

IF 'courseTable' is empty:

PRINT "Load data first"

ELSE:

INPUT courseNumber

IF courseNumber in courseTable:

PRINT courseTable[courseNumber]

ELSE:

PRINT "Course not found"

// Main loop

INITIALIZE 'courseTable'

WHILE True:

choice = displayMenu()

IF choice == 1:

loadCourseData()

ELIF choice == 2:

printAllCourses()

ELIF choice == 3:

printCourseInfo()

ELIF choice == 9:

PRINT "Exiting"

BREAK

ELSE:

PRINT "Invalid choice"

// End program

END program

**Vector Pseudocode:**

// Start program

START program

//Define Course Class

CLASS Course:

courseNumber

courseTitle

prerequisites

FUNCTION \_\_init\_\_(self, courseNumber, courseTitle, prerequisites):

self.courseNumber = courseNumber

self.courseTitle = courseTitle

self.prerequisites = prerequisites

// Open file and initialize

OPEN file

INITIALIZE vector 'courseList' and set 'courseNumbers'

// Process each line in the file

WHILE not end of file:

read line, split by ","

IF length < 2:

PRINT "Error"

ELSE:

courseNumber, courseTitle = components[0], components[1]

prerequisites = components[2 to end]

ADD courseNumber to 'courseNumbers'

valid = all(prerequisite in 'courseNumbers' for prerequisite in prerequisites)

IF valid:

CREATE course with courseNumber, courseTitle, prerequisites

ADD course to 'courseList'

CLOSE file

//Validate prerequisites

FOR each course IN courseList:

valid = ALL(prerequisite IN 'courseNumbers' FOR prerequisite IN course.prerequisites)

IF NOT valid:

PRINT "Error: Invalid prerequisites for course ", course.courseNumber

// Print all courses in alphanumeric order

FUNCTION printAllCoursesVector():

SORT courseList by courseNumber

FOR course in courseList:

PRINT course.courseNumber, course.courseTitle

// Search and print course

FUNCTION searchCourses(courses, courseNumber) {

FOR each course in courses:

IF course.courseNumber == courseNumber:

PRINT course.courseNumber, course.courseTitle, join(course.prerequisites, ", ")

FOR prerequisite in course.prerequisites:

PRINT prerequisite

RETURN

PRINT "Course not found"

// End program

END program

**Hash Table Pseudocode:**

// Start program

start program

// Define Course structure

class Course:

def \_\_init\_\_(self, courseNumber, courseTitle, prerequisites):

self.courseNumber = courseNumber

self.courseTitle = courseTitle

self.prerequisites = prerequisites

// Open file and initialize

open file

INITIALIZE hash table 'courseTable' and set 'courseNumbers'

// Process each line in the file

WHILE not end of file:

read line, split by ","

IF length < 2:

PRINT "Error"

ELSE:

courseNumber, courseTitle = components[0], components[1]

prerequisites = components[2 to end]

ADD courseNumber to 'courseNumbers'

create course with courseNumber, courseTitle, prerequisites

ADD course to 'courseTable' with key courseNumber

close file

// Second pass Validate prerequisites

FOR course in courseTable.values():

valid = all(prerequisite in 'courseNumbers' for prerequisite in course.prerequisites)

IF not valid:

PRINT "Error: prerequisites not found for course " + course.courseNumber

REMOVE course from 'courseTable'

// Print all courses in alphanumeric order

FUNCTION printAllCoursesHashTable():

courseNumbers = keys of courseTable

SORT courseNumbers

FOR courseNumber in courseNumbers:

PRINT courseNumber, courseTable[courseNumber].courseTitle

// Search and print course

FUNCTION searchCourse(HashTable<Course> courses, String courseNumber) {

IF courseNumber in courses:

course = courses[courseNumber]

PRINT course.courseNumber, course.courseTitle, join(course.prerequisites, ", ")

FOR prerequisite in course.prerequisites:

PRINT prerequisite

ELSE:

PRINT "Course not found"

// End program

end program

**Binary Search Tree Pseudocode:**

// Start program

start program

//Define Course Class

CLASS Course:

FUNCTION \_\_init\_\_(self, courseNumber, courseTitle, prerequisites):

self.courseNumber = courseNumber

self.courseTitle = courseTitle

self.prerequisites = prerequisites

// Open file and initialize

open file

INITIALIZE binary search tree 'courseTree'

INITIALIZE list 'courseList'

INITIALIZE set 'courseNumbers'

// Process each line in the file

WHILE not end of file:

read line, split by ","

IF length < 2:

PRINT "Error"

ELSE:

courseNumber = line[0]

courseTitle=line[1]

prerequisites = line[2 to end]

ADD (courseNumber, courseTitle, prerequisites) to 'courseList'

ADD courseNumber to 'courseNumbers'

Close file

// Process and validate prerequisites

FOR each (courseNumber, courseTitle, prerequisites) in 'courseList':

valid = True

FOR prerequisite in prerequisites:

IF prerequisite not in 'courseNumbers':

valid = False

BREAK

IF valid:

newCourse = Course(courseNumber, courseTitle, prerequisites)

ADD newCourse to 'courseTree'

// Print all courses in alphanumeric order

FUNCTION printAllCoursesTree():

FUNCTION inOrderTraversal(node):

IF node is not null:

inOrderTraversal(node.left)

PRINT node.courseNumber, node.courseTitle

inOrderTraversal(node.right)

inOrderTraversal(courseTree.root)

// Search and print course

FUNCTION searchCourse(Tree<Course> courses, String courseNumber) {

current = courses.root

WHILE currentis not NULL:

IF current.courseNumber == courseNumber:

PRINT course.courseNumber, course.courseTitle, join(course.prerequisites, ", ")

FOR prerequisite in current.prerequisites:

PRINT prerequisite

RETURN

current = current.left if courseNumber < current.courseNumber else current.right

PRINT "Course not found"

// End program

end program

**Runtime Analysis:**

| **Data Structure** | **Initialization & File Reading** | **Insertion** | **Sorting** | **Search** |
| --- | --- | --- | --- | --- |
| **Vector** | O (n) | O (1) | O (n log n) | O(n) |
| **Hash Table** | O (n) | O (1) | O (n log n) | O (1) (avg) |
| **BST** | O (n) | O (log n) | O (n) (in-Order Traversal) | O (log n) (avg) |

**Advantages and Disadvantages:**

In considering data structures for managing course data, vectors, hash tables, and binary search trees (BSTs) each offer unique benefits and drawbacks. Vectors are advantageous due to their dynamic sizing and rapid random access, making them suitable for frequently changing datasets. However, they struggle with efficient insertion and deletion, particularly in the middle of the structure, and resizing operations can be costly in terms of performance.

Hash tables provide an average-case time complexity of O(1) for insertion, deletion, and search operations, which is beneficial for handling large datasets and frequent updates. Despite their higher memory requirements due to hashing and potential performance issues from collisions, these drawbacks can be minimized with a well-designed hash function, making hash tables highly efficient for quick lookups and dynamic data management.

Binary search trees, especially when balanced, offer efficient O(log n) time complexity for search, insertion, and deletion operations. They maintain elements in a sorted order, which is helpful for range queries. However, unbalanced trees can degrade performance to O(n), and the complexity of maintaining balance adds to their implementation and upkeep challenges.

**Recommendation:**

Given the need for efficient performance and scalability, the hash table stands out as the most suitable data structure for this scenario. It offers a balance of speed, efficiency, and scalability, handling large datasets effectively while supporting frequent updates and quick lookups.