Shebin Sajan

CS-300

16 October 2024

**1. Pseudocode for Each Data Structure**

**Vector-Based Pseudocode**

This is a simple array-like data structure that dynamically grows in size.

// Function to open file

function openFile(filename):

    open file with name 'filename'

    if file not found:

        print "File not found"

        exit program

    return file pointer

// Function to read file and parse data

function readFile(filePointer):

    initialize empty vector 'courseList'

    for each line in filePointer:

        split line by ',' into 'courseData'

        if length of courseData < 2:

            print "Invalid format"

            continue

        courseNumber = courseData[0]

        courseTitle = courseData[1]

        prerequisites = courseData[2..n] // All remaining elements are prerequisites

        create course object using courseNumber, courseTitle, and prerequisites

        append course object to 'courseList'

    return courseList

Course Object Creation:

// Define structure for course object

struct Course:

    courseNumber

    courseTitle

    prerequisites (vector)

// Function to create a course object

function createCourse(courseNumber, courseTitle, prerequisites):

    initialize course object

    set course.courseNumber = courseNumber

    set course.courseTitle = courseTitle

    set course.prerequisites = prerequisites

    return course

Menu and Course Operations:

// Function to print course information

function searchCourse(courseList, courseNumber):

    for each course in courseList:

        if course.courseNumber == courseNumber:

            print "Course Number: " + course.courseNumber

            print "Course Title: " + course.courseTitle

            if course.prerequisites is empty:

                print "No prerequisites"

            else:

                print "Prerequisites: " + join(course.prerequisites, ", ")

            return

    print "Course not found"

// Main program

function main():

    filePointer = openFile("course\_data.txt")

    courseList = readFile(filePointer)

    userInput = get user input "Enter course number:"

    searchCourse(courseList, userInput)

**Hash Table-Based Pseudocode**

Hash tables allow for fast lookups and are ideal when order is not needed.

File Handling and Parsing:

// Function to open file and parse data

function processFile(filename):

    open file with name 'filename'

    initialize empty hash table 'courseTable'

    for each line in file:

        courseData = parseLine(line)

        if courseData is invalid:

            print "Invalid line format"

            continue

        addCourseToHashTable(courseData)

    return courseTable

// Function to parse line

function parseLine(line):

    split line by ',' into tokens

    if tokens.length < 2:

        return invalid

    courseNumber = tokens[0]

    courseTitle = tokens[1]

    prerequisites = tokens[2..n] // All remaining elements are prerequisites

    return {courseNumber, courseTitle, prerequisites}

Course Object Creation:

// Function to add course to hash table

function addCourseToHashTable(courseData):

    course = createCourse(courseData)

    hashKey = hash(course.courseNumber)

    add course to courseTable[hashKey]

// Define structure for course object

struct Course:

    courseNumber

    courseTitle

    prerequisites (list)

Menu and Course Operations:

// Function to search and print course information

function searchCourse(courseTable, courseNumber):

    hashKey = hash(courseNumber)

    course = courseTable[hashKey]

    if course is not found:

        print "Course not found"

        return

    print "Course Number: " + course.courseNumber

    print "Course Title: " + course.courseTitle

    if course.prerequisites is empty:

        print "No prerequisites"

    else:

        print "Prerequisites: " + join(course.prerequisites, ", ")

**Binary Search Tree (BST)-Based Pseudocode**

BSTs maintain a sorted structure, ideal for ordered operations like printing courses in order.

File Handling and Parsing:

// Open file and read course data into BST

function processFile(filename):

    open file with name 'filename'

    initialize empty binary search tree 'bst'

    for each line in file:

        courseData = parseLine(line)

        if courseData is invalid:

            print "Invalid line format"

            continue

        course = createCourse(courseData)

        bst.insert(course)

    return bst

Course Object Creation:

// Define structure for course object

struct Course:

    courseNumber

    courseTitle

    prerequisites (list)

// Function to insert course into BST

function bst.insert(course):

    if bst is empty:

        set root to course

    else:

        insert recursively in sorted order

Menu and Course Operations:

// Function to search and print course information

function searchCourse(bst, courseNumber):

    course = bst.search(courseNumber)

    if course is not found:

        print "Course not found"

        return

    print "Course Number: " + course.courseNumber

    print "Course Title: " + course.courseTitle

    if course.prerequisites is empty:

        print "No prerequisites"

    else:

        print "Prerequisites: " + join(course.prerequisites, ", ")

// Function to print courses in order

function bst.inOrderTraversal():

    recursively visit left subtree, print course, visit right subtree

**2. Runtime Analysis**

Runtime Complexity Chart:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Vector | Hash Table | Binary Search Tree |
| Loading Data | O(n) | O(n) | O(n log n) |
| Searching for a Course | O(n) | O(1) | O(log n) |
| Printing Courses in Order | O(n log n) (sorting) | O(n log n) (sorting) | O(n) (in-order traversal) |

* Vector: Simple to implement but slow for searching and printing courses in order (due to sorting).
* Hash Table: Fast lookup (O(1)) but requires sorting for ordered printing.
* Binary Search Tree: Naturally maintains order, making it optimal for searching and printing in order.

**3. Advantages and Disadvantages**

Vector:

* Advantages: Simple implementation, dynamically grows.
* Disadvantages: Slow searches (O(n)), needs sorting for ordered printing (O(n log n)).

Hash Table:

* Advantages: Fast lookup (O(1)), efficient for searching.
* Disadvantages: Does not maintain order, needs sorting for ordered printing (O(n log n)).

Binary Search Tree:

* Advantages: Efficient for both searching (O(log n)) and printing in order (O(n)).
* Disadvantages: More complex to implement, insertion and balancing required.

4. Final Recommendation

After analyzing all three data structures, the binary search tree is the best choice because it provides an efficient way to maintain ordered data (for alphanumeric sorting) and allows for fast searches (O(log n)). Although slightly more complex to implement, the BST's advantages in both ordered printing and search efficiency make it optimal for this program.