**CS 300 Project One Pseudocode (with abbreviated vector, hash table, BST implementation)**

**Huan Ai**

**4/13/2025**

struct Course {

std::string courseCode; // Key field (e.g., "CSCI100")

std::string title;

std::vector<std::string> prerequisites;

};

**// ===== Vector implementation =====**

**// 1. Load Data**

vector<Course> loadCourseData(string filePath) {

vector<Course> courseList;

Open file at filePath

If file fails to open:

Output "Error: File not found"

Return empty courseList

While getline(file, currentLine):

stringstream ss(currentLine)

vector<string> tokens

string token

// Tokenize line

While getline(ss, token, ','):

token = trimWhitespace(token)

tokens.push\_back(token)

// Validate

If tokens.size() < 2:

Output "Invalid line: " + currentLine

Continue

// Create course

Course newCourse

newCourse.courseCode = tokens[0]

newCourse.title = tokens[1]

// Add prerequisites

For i from 2 to tokens.size()-1:

newCourse.prerequisites.push\_back(tokens[i])

courseList.push\_back(newCourse)

Return courseList

}

**// 2. Search**

Course\* findCourse(vector<Course>& courses, string targetCode) {

For each course in courses:

If course.courseCode == targetCode:

Return &course

Return nullptr

}

**// 3. Sort and Print**

void printSortedVector(vector<Course>& courses) {

// Modified merge sort

sort(courses.begin(), courses.end(),

[](const Course& a, const Course& b) {

return a.courseCode < b.courseCode; // Alphanumeric comparison

})

For each course in sorted courses:

Output course.courseCode + ": " + course.title

}

**// ===== Hash table implementation =====**

**// 1. Hash Function**

size\_t hashFunction(string courseCode) {

size\_t hash = 0

For each character in courseCode:

hash = (hash \* 31) + ASCII value of character

Return hash % TABLE\_SIZE

}

**// 2. Load Data**

unordered\_map<string, Course> loadHashTable(string filePath) {

unordered\_map<string, Course> courseTable

// ... (same file loading logic as vector)

// Insert using: courseTable[newCourse.courseCode] = newCourse

Return courseTable

}

**// 3. Sort and Print**

void printSortedHashTable(unordered\_map<string, Course>& courses) {

vector<Course> tempList

For each key-value pair in courses:

tempList.push\_back(value)

sort(tempList.begin(), tempList.end(),

[](const Course& a, const Course& b) {

return a.courseCode < b.courseCode

})

For each course in tempList:

Output course.courseCode + ": " + course.title

}

**// ===== BST implementation =====**

struct Node {

Course course;

Node\* left;

Node\* right;

Node(Course c) : course(c), left(nullptr), right(nullptr) {}

};

**// 1. Insert**

void insertNode(Node\*& root, Course newCourse) {

If root is null:

root = new Node(newCourse)

Else if newCourse.courseCode < root->course.courseCode:

insertNode(root->left, newCourse)

Else if newCourse.courseCode > root->course.courseCode:

insertNode(root->right, newCourse)

Else:

Output "Duplicate course found: " + newCourse.courseCode

}

**// 2. In-Order Traversal (auto-sorted)**

void printInOrder(Node\* root) {

If root is not null:

printInOrder(root->left)

Output root->course.courseCode + ": " + root->course.title

printInOrder(root->right)

}

**// ===== Menu System =====**

main() {

// Initialize all data structures

vector<Course> coursesVector

unordered\_map<string, Course> coursesHashTable

Node\* bstRoot = nullptr

While True:

Display "ABCU Course Advisor"

Display "1. Load Data"

Display "2. Print Course List (Sorted)"

Display "3. Find Course"

Display "9. Exit"

choice = GetInput("Select option: ")

Switch(choice):

Case 1: // Load Data

filePath = GetInput("Enter file path [default: courses.csv]: ")

If filePath empty: filePath = "courses.csv"

// Load into all structures

coursesVector = loadCourseData(filePath)

coursesHashTable = loadHashTable(filePath)

bstRoot = loadCoursesIntoBST(filePath)

Case 2: // Print Sorted List

Display "Select data structure:"

Display "a. Vector"

Display "b. Hash Table"

Display "c. Binary Search Tree"

structChoice = GetInput("Choice: ")

If structChoice == 'a':

printSortedVector(coursesVector)

Else if structChoice == 'b':

printSortedHashTable(coursesHashTable)

Else if structChoice == 'c':

printInOrder(bstRoot)

Case 3: // Find Course

courseCode = GetInput("Enter course code: ")

// Search all structures

Display "\nVector Results:"

If findCourse(coursesVector, courseCode):

Print course details

Else: "Not found"

Display "\nHash Table Results:"

If coursesHashTable.find(courseCode) exists:

Print course details

Else: "Not found"

Display "\nBST Results:"

If searchBST(bstRoot, courseCode):

Print course details

Else: "Not found"

Case 9:

Exit program

Default:

Display "Invalid choice"

}

**Runtime Analysis**

**Vector:**

loadCourseData()

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Open file | 1 | 1 | 1 |
| While getline(file, currentLine) | 1 | n (courses) | n |
| Tokenize line | 1 | n | n |
| Validate tokens | 1 | n | n |
| Create course | 1 | n | n |
| Add prerequisites | 1 | p (avg prerequisites) | p |
| Total Cost |  |  | O(n + p) |
| Runtime |  |  | O(n) |

findCourse()

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| For each course in courses | 1 | n | n |
| If course.courseCode == targetCode | 1 | n (worst case) | n |
| Total Cost |  |  | 2n |
| Runtime |  |  | O(n) |

printSortedVector()

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| sort() (merge sort) | 1 | n log n | n log n |
| For each course | 1 | n | n |
| Total Cost |  |  | n log n + n |
| Runtime |  |  | O(n log n) |

**Hash table:**

hashFunction()

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| For each character in courseCode | 1 | k (length of code) | k |
| Total Cost |  |  | O(k) |
| Runtime |  |  | O(1) |

loadHashTable()

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Insert into hash table | 1 | n | n (avg case) |
| Total Cost |  |  | O(n + p) |
| Runtime |  |  | O(n) |

printSortedHashTable()

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Copy to temp vector | 1 | n | n |
| sort() | 1 | n log n | n log n |
| For each course | 1 | n | n |
| Total Cost |  |  | 2n + n log n |
| Runtime |  |  | O(n log n) |

**BST:**

insertNode()

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| Recursive insert | 1 | height of tree | h |
| Worst case (unbalanced) |  | n | n |
| Average case (balanced) |  | log n | log n |
| Runtime (worst case) |  |  | O(n) |
| Runtime (avg case) |  |  | O(log n) |

printInOrder()

| Code | Line Cost | # Times Executes | Total Cost |
| --- | --- | --- | --- |
| In-order traversal | 1 | n | n |
| Runtime |  |  | O(n) |

Summary of Time Complexities:

| Operation | Vector | Hash Table | BST (avg) | BST (worst) |
| --- | --- | --- | --- | --- |
| Load Data | O(n) | O(n) | O(n log n) | O(n²) |
| Find Course | O(n) | O(1) | O(log n) | O(n) |
| Print Sorted List | O(n log n) | O(n log n) | O(n) | O(n) |

**Advantages and Disadvantages**

Advantages of vector include simple implementation, and fast data loading. However using vectors also results in relatively slow sorting, and slower search unless searching on the index.

Hash tables are very fast at both loading and retrieving data. But it’s slow to sort the data since the data can’t be sorted in place, making any sortation expansive.

BST naturally maintains sorted order, and tends to operate most consistently at or near O(log N). It’s also efficient at printing sorter lists. However it’s more complex to implement, and it can become unbalanced (O(n) worst case) depending on the data.

**Recommendation**

My recommended data structure to use is the binary search tree. The main reason is its comprehensiveness. With O(log n) average search time, it provides better performance than the vector (O(n)) while being more predictable than the hash table's worst-case O(n) scenario (collisions and resizing). The BST also naturally maintains courses in sorted order, making the alphanumeric listing requirement easy to implement (O(n) in-order traversal vs O(n log n) sorting for other structures).