**CS 300 Project One Milestone Submission: Course Management System**

**Student Name: Alexander Ouellet**  
**Date: 11 August 2024**  
**Course: CS 300**

### Introduction

This document outlines the design and pseudocode implementation of a course management system for the Computer Science department at ABC University (ABCU). The system is intended to help academic advisors access and manage course information, including printing a list of all courses in alphanumeric order and retrieving course information along with prerequisites. The pseudocode is provided for three different data structures: vector, hash table, and binary search tree. Each structure's pseudocode includes loading course data from a file, sorting and printing courses, and searching for specific course information. Finally, a runtime analysis is performed to evaluate the efficiency of each data structure, leading to a final recommendation.

### Pseudocode Implementation

#### 1. Vector Implementation

**Loading Data into Vector**

### // Function to load courses from a file into a vector

### function loadCoursesVector(fileName: String) -> Vector<Course>:

### // Open the file in read mode

### file = open(fileName, "r")

### // Initialize a vector to hold course objects

### courses = Vector<Course>()

### // Read each line from the file until the end

### while (line = readLine(file)) != null:

### // Split the line into tokens using comma as delimiter

### tokens = split(line, ",")

### 

### // Check for file format errors (at least course number and title should be present)

### if length(tokens) < 2:

### print "Error: Invalid line format"

### continue // Skip to the next line

### // Extract course number and title from the tokens

### courseNumber = tokens[0]

### courseTitle = tokens[1]

### 

### // Initialize a vector to hold prerequisites

### prerequisites = Vector<String>()

### 

### // Extract prerequisites from the remaining tokens, if any

### for i = 2 to length(tokens) - 1:

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

### 

### // Create a course object with the extracted data

### course = Course(courseNumber, courseTitle, prerequisites)

### 

### // Add the course object to the vector

### courses.push\_back(course)

### // Close the file after reading

### close(file)

### 

### // Return the vector containing all course objects

### return courses

**Sorting and Printing Courses in Vector**

// Function to sort and print courses stored in a vector

function sortAndPrintCoursesVector(courses: Vector<Course>):

// Use any efficient sorting algorithm (e.g., quicksort)

quickSort(courses, 0, courses.size() - 1)

// Print sorted list of courses

for course in courses:

print "Course Number: " + course.courseNumber

print "Course Title: " + course.courseTitle

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

**Searching and Printing a Specific Course in Vector**

// Function to search for and print a specific course in a vector

function searchCourseVector(courses: Vector<Course>, courseNumber: String):

// Iterate through the vector to find the course

for course in courses:

if course.courseNumber == courseNumber:

// Print course details

print "Course Number: " + course.courseNumber

print "Course Title: " + course.courseTitle

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

return

// If course is not found, print a message

print "Course not found."

#### 2. Hash Table Implementation

**Loading Data into Hash Table**

// Function to load courses from a file into a hash table

function loadCoursesHashTable(fileName: String) -> HashTable<Course>:

// Open the file in read mode

file = open(fileName, "r")

// Initialize a hash table to hold course objects

courses = HashTable<Course>()

// Read each line from the file until the end

while (line = readLine(file)) != null:

// Split the line into tokens using comma as delimiter

tokens = split(line, ",")

// Check for file format errors (at least course number and title should be present)

if length(tokens) < 2:

print "Error: Invalid line format"

continue // Skip to the next line

// Extract course number and title from the tokens

courseNumber = tokens[0]

courseTitle = tokens[1]

// Initialize a vector to hold prerequisites

prerequisites = Vector<String>()

// Extract prerequisites from the remaining tokens, if any

for i = 2 to length(tokens) - 1:

prerequisites.push\_back(tokens[i])

// Create a course object with the extracted data

course = Course(courseNumber, courseTitle, prerequisites)

// Insert the course object into the hash table using the course number as key

courses.insert(courseNumber, course)

// Close the file after reading

close(file)

// Return the hash table containing all course objects

return courses

**Sorting and Printing Courses in Hash Table**

// Function to sort and print courses stored in a hash table

function sortAndPrintCoursesHashTable(courses: HashTable<Course>):

// Extract all course objects from the hash table

courseList = courses.values()

// Use any efficient sorting algorithm (e.g., quicksort)

quickSort(courseList, 0, courseList.size() - 1)

// Print sorted list of courses

for course in courseList:

print "Course Number: " + course.courseNumber

print "Course Title: " + course.courseTitle

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

**Searching and Printing a Specific Course in Hash Table**

// Function to search for and print a specific course in a hash table

function searchCourseHashTable(courses: HashTable<Course>, courseNumber: String):

// Search for the course using the course number as the key

course = courses.search(courseNumber)

// Check if the course is found

if course != null:

// Print course details

print "Course Number: " + course.courseNumber

print "Course Title: " + course.courseTitle

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

else:

// If course is not found, print a message

print "Course not found."

#### 3. Binary Search Tree Implementation

**Loading Data into Binary Search Tree**

// Function to load courses from a file into a binary search tree

function loadCoursesBST(fileName: String) -> BinarySearchTree<Course>:

// Open the file in read mode

file = open(fileName, "r")

// Initialize a binary search tree to hold course objects

courses = BinarySearchTree<Course>()

// Read each line from the file until the end

while (line = readLine(file)) != null:

// Split the line into tokens using comma as delimiter

tokens = split(line, ",")

// Check for file format errors (at least course number and title should be present)

if length(tokens) < 2:

print "Error: Invalid line format"

continue // Skip to the next line

// Extract course number and title from the tokens

courseNumber = tokens[0]

courseTitle = tokens[1]

// Initialize a vector to hold prerequisites

prerequisites = Vector<String>()

// Extract prerequisites from the remaining tokens, if any

for i = 2 to length(tokens) - 1:

prerequisites.push\_back(tokens[i])

// Create a course object with the extracted data

course = Course(courseNumber, courseTitle, prerequisites)

// Insert the course object into the binary search tree

courses.insert(courseNumber, course)

// Close the file after reading

close(file)

// Return the binary search tree containing all course objects

return courses

**Sorting and Printing Courses in Binary Search Tree**

// Function to sort and print courses stored in a binary search tree

function sortAndPrintCoursesBST(courses: BinarySearchTree<Course>):

// Perform an in-order traversal of the binary search tree

courses.inOrderTraversal() // In-order traversal prints courses in alphanumeric order

**Searching and Printing a Specific Course in Binary Search Tree**

// Function to search for and print a specific course in a binary search tree

function searchCourseBST(courses: BinarySearchTree<Course>, courseNumber: String):

// Search for the course using the course number

course = courses.search(courseNumber)

// Check if the course is found

if course != null:

// Print course details

print "Course Number: " + course.courseNumber

print "Course Title: " + course.courseTitle

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

else:

// If course is not found, print a message

print "Course not found."

#### 4. Menu Options

**Menu Pseudocode**

// Function to display the menu and handle user input

function displayMenu():

while true:

// Print the menu options

print "Menu:"

print "1. Load Courses"

print "2. Print All Courses"

print "3. Print Course Information"

print "9. Exit"

// Get the user's choice

choice = input("Enter your choice: ")

// Handle the user's choice

if choice == "1":

// Load the courses from the file into the selected data structure

fileName = input("Enter the file name: ")

courses = loadCourses(fileName)

print "Courses loaded successfully."

else if choice == "2":

// Sort and print all courses in alphanumeric order

sortAndPrintCourses(courses)

else if choice == "3":

// Search for and print information about a specific course

courseNumber = input("Enter the Course Number: ")

searchCourse(courses, courseNumber)

else if choice == "9":

// Exit the program

print "Goodbye."

break

else:

// Handle invalid input

print "Invalid choice. Please enter 1, 2, 3, or 9."

### Runtime Analysis

#### Vector

* **Loading Data:** O(n)
  + The data is loaded into the vector sequentially, requiring one operation per course entry.
* **Search:** O(n)
  + Searching through the vector requires a linear scan, making the worst-case search time O(n).
* **Sort:** O(n log n)
  + Sorting the vector requires an efficient sorting algorithm like quicksort or mergesort, which operates in O(n log n) time.
* **Memory:** O(n)
  + The vector grows dynamically and uses O(n) memory for storing course objects.

#### Hash Table

* **Loading Data:** O(n)
  + Inserting each course into the hash table is a constant-time operation, so loading all courses is O(n).
* **Search:** O(1)
  + Searching for a course using the hash table key is a constant-time operation.
* **Sort:** O(n log n)
  + To sort the courses, they need to be extracted into a list and sorted, which takes O(n log n) time.
* **Memory:** O(n)
  + The hash table uses O(n) memory, though with some additional overhead for managing collisions.

#### Binary Search Tree

* **Loading Data:** O(n log n)
  + Inserting each course into the binary search tree requires O(log n) time, making the overall loading time O(n log n).
* **Search:** O(log n)
  + Searching for a course in a balanced binary search tree requires O(log n) time.
* **Sort:** O(n log n)
  + Sorting is handled by in-order traversal, which inherently provides the sorted order of elements.
* **Memory:** O(n)
  + The binary search tree uses O(n) memory, with each node pointing to its left and right children.

### Evaluation of Data Structures

**Vector:**

* **Advantages:**
  + Simple and intuitive to implement.
  + Efficient for small datasets.
* **Disadvantages:**
  + Linear search and sorting make it inefficient for large datasets.
  + Insertion and deletion operations can be slow due to the need to shift elements.

**Hash Table:**

* **Advantages:**
  + Provides constant-time complexity for search operations.
  + Ideal for fast lookups by key (course number).
* **Disadvantages:**
  + Inefficient for ordered operations like sorting.
  + Memory overhead due to potential collisions and hash table resizing.

**Binary Search Tree:**

* **Advantages:**
  + Balanced performance for both search and sorting operations.
  + Naturally ordered structure makes it easy to maintain sorted data.
* **Disadvantages:**
  + Performance can degrade to O(n) if the tree becomes unbalanced.
  + More complex to implement compared to vectors and hash tables.

### Recommendation

Based on the runtime analysis and the requirements of the ABCU course management system, **the Binary Search Tree (BST) is recommended** as the data structure of choice. The BST provides a good balance between search and sorting efficiency, making it suitable for handling the course data, especially with the need to print courses in alphanumeric order and retrieve course information efficiently. The balanced nature of BST ensures that operations remain efficient even as the dataset grows, making it the most robust solution for the task at hand.

