ABCU Advising Program

Author: Christopher Phillips

Date: 04/13/2025

Introduction

This document outlines the design for a program that manages Computer Science courses at ABCU. The program will:

1. Load course data from a file.
2. Provide a menu to:
   * Print all course information in alphanumeric order.
   * Print details (title and prerequisites) of a given course.
3. Support three possible data structures for storing courses:
   * Vector
   * Hash Table
   * Binary Search Tree (BST)

I will present pseudocode for each approach, evaluate the runtime performance, discuss the advantages and disadvantages of each structure, and offer a final recommendation based on the advising staff’s needs.

**File and Loading and Validation**

FUNCTION loadCoursesFromFile(filename):

OPEN file with name = filename

IF file could NOT be opened THEN

PRINT "Error: Unable to open file."

RETURN an empty list (or null)

courseLines = empty list

// Read each line

WHILE not end-of-file in file DO

currentLine = READ line from file

IF currentLine is NOT empty THEN

APPEND currentLine to courseLines

ENDIF

ENDWHILE

CLOSE file

// Validate each line has at least a course number and a course title

validCourseNumbers = CREATE an empty set

FOR EACH line IN courseLines DO

tokens = SPLIT(line, ",")

IF length(tokens) < 2 THEN

PRINT "Error: Invalid line format. Skipping line: " + line

// Optionally remove or mark the line invalid

ELSE

// Record this course number in our set for later prerequisite checks

validCourseNumbers.add(tokens[0])

ENDIF

ENDFOR

// Optional: Validate that each prerequisite exists in the set of known course numbers

// (We will handle this again during the build phase, but you could do it here as well.)

RETURN courseLines

END FUNCTION

**Building and Printing Data for Each Structure**

**Build Vector**

FUNCTION buildVectorFromLines(courseLines):

vector courses = empty vector

validCoursesSet = CREATE an empty set // For prerequisite validation

// First pass: gather all course numbers

FOR EACH line IN courseLines DO

tokens = SPLIT(line, ",")

IF length(tokens) >= 2 THEN

validCoursesSet.add(tokens[0])

ENDIF

ENDFOR

// Second pass: build Course objects

FOR EACH line IN courseLines DO

tokens = SPLIT(line, ",")

IF length(tokens) < 2 THEN

// We skip invalid lines

CONTINUE

ENDIF

courseNumber = tokens[0]

courseTitle = tokens[1]

// Gather prerequisites, if any

prerequisitesList = empty list

IF length(tokens) > 2 THEN

FOR i FROM 2 TO (length(tokens)-1) DO

prereq = tokens[i]

IF validCoursesSet.contains(prereq) THEN

APPEND prereq TO prerequisitesList

ELSE

PRINT "Warning: Prerequisite " + prereq +

" not found for course " + courseNumber

ENDIF

ENDFOR

ENDIF

CREATE newCourse of type Course

newCourse.courseNumber = courseNumber

newCourse.courseTitle = courseTitle

newCourse.prerequisites = prerequisitesList

// Insert into vector

APPEND newCourse TO courses

ENDFOR

RETURN courses

END FUNCTION

**Print All Courses in Alphanumeric Oder**

FUNCTION printAllCoursesVector(courses):

// We assume the vector may be unsorted. We'll call mergesort for clarity.

CALL mergeSort(courses, compareByCourseNumber)

FOR EACH course IN courses DO

PRINT course.courseNumber + ": " + course.courseTitle

ENDFOR

END FUNCTION

FUNCTION compareByCourseNumber(courseA, courseB):

// Compare alphanumeric strings, e.g., "CSCI100" < "CSCI200"

RETURN (courseA.courseNumber < courseB.courseNumber)

END FUNCTION

**Print One Course (Vector)**

FUNCTION printCourseVector(courses, targetCourseNumber):

// Linear search

FOR EACH course IN courses DO

IF course.courseNumber == targetCourseNumber THEN

PRINT "Course Number: " + course.courseNumber

PRINT "Title: " + course.courseTitle

IF course.prerequisites is not empty THEN

PRINT "Prerequisites:"

FOR EACH prereq IN course.prerequisites DO

PRINT " " + prereq

ENDFOR

ELSE

PRINT "Prerequisites: None"

ENDIF

RETURN // done

ENDIF

ENDFOR

PRINT "Course " + targetCourseNumber + " not found."

END FUNCTION

**Build Hash Table**

FUNCTION buildHashTableFromLines(courseLines):

CREATE hashTable // key = courseNumber, value = Course object

validCoursesSet = CREATE an empty set

// First pass: gather all course numbers

FOR EACH line IN courseLines DO

tokens = SPLIT(line, ",")

IF length(tokens) >= 2 THEN

validCoursesSet.add(tokens[0])

ENDIF

ENDFOR

// Second pass: build Course objects and insert

FOR EACH line IN courseLines DO

tokens = SPLIT(line, ",")

IF length(tokens) < 2 THEN

CONTINUE

ENDIF

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisitesList = empty list

IF length(tokens) > 2 THEN

FOR i FROM 2 TO (length(tokens)-1) DO

prereq = tokens[i]

IF validCoursesSet.contains(prereq) THEN

APPEND prereq TO prerequisitesList

ELSE

PRINT "Warning: Prerequisite " + prereq +

" not found for course " + courseNumber

ENDIF

ENDFOR

ENDIF

CREATE newCourse of type Course

newCourse.courseNumber = courseNumber

newCourse.courseTitle = courseTitle

newCourse.prerequisites = prerequisitesList

hashTable.insert(courseNumber, newCourse)

ENDFOR

RETURN hashTable

END FUNCTION

**Print All Courses in Alphanumeric Order ( Hash Table)**

FUNCTION printAllCoursesHash(hashTable):

keysList = hashTable.getAllKeys()

// Sort keys using mergesort or quicksort

CALL mergeSort(keysList, compareKeys)

FOR EACH key IN keysList DO

course = hashTable.search(key)

PRINT course.courseNumber + ": " + course.courseTitle

ENDFOR

END FUNCTION

FUNCTION compareKeys(keyA, keyB):

RETURN (keyA < keyB) // Alphanumeric comparison

END FUNCTION

**Print One Course (Hash Table)**

FUNCTION printCourseHash(hashTable, targetCourseNumber):

course = hashTable.search(targetCourseNumber)

IF course == null THEN

PRINT "Course " + targetCourseNumber + " not found."

RETURN

ENDIF

PRINT "Course Number: " + course.courseNumber

PRINT "Title: " + course.courseTitle

IF course.prerequisites is not empty THEN

PRINT "Prerequisites:"

FOR EACH prereq IN course.prerequisites DO

PRINT " " + prereq

ENDFOR

ELSE

PRINT "Prerequisites: None"

ENDIF

END FUNCTION

**Binary Search Tree (BST) Implementation**

**Build BST**

STRUCT BSTNode:

course // the Course object

left // pointer to BSTNode

right // pointer to BSTNode

END STRUCT

FUNCTION buildBSTFromLines(courseLines):

CREATE BST with BST.root = null

validCoursesSet = CREATE an empty set

// First pass: gather course numbers

FOR EACH line IN courseLines DO

tokens = SPLIT(line, ",")

IF length(tokens) >= 2 THEN

validCoursesSet.add(tokens[0])

ENDIF

ENDFOR

// Second pass: build Course objects and insert

FOR EACH line IN courseLines DO

tokens = SPLIT(line, ",")

IF length(tokens) < 2 THEN

CONTINUE

ENDIF

courseNumber = tokens[0]

courseTitle = tokens[1]

prerequisitesList = empty list

IF length(tokens) > 2 THEN

FOR i FROM 2 TO (length(tokens)-1) DO

prereq = tokens[i]

IF validCoursesSet.contains(prereq) THEN

APPEND prereq TO prerequisitesList

ELSE

PRINT "Warning: Prerequisite " + prereq +

" not found for course " + courseNumber

ENDIF

ENDFOR

ENDIF

CREATE newCourse of type Course

newCourse.courseNumber = courseNumber

newCourse.courseTitle = courseTitle

newCourse.prerequisites = prerequisitesList

BSTInsert(BST, newCourse)

ENDFOR

RETURN BST

END FUNCTION

FUNCTION BSTInsert(BST, course):

node = CREATE new BSTNode

node.course = course

node.left = null

node.right = null

IF BST.root == null THEN

BST.root = node

RETURN

ENDIF

CURRENT = BST.root

WHILE true DO

IF course.courseNumber < CURRENT.course.courseNumber THEN

IF CURRENT.left == null THEN

CURRENT.left = node

RETURN

ELSE

CURRENT = CURRENT.left

ENDIF

ELSE

IF CURRENT.right == null THEN

CURRENT.right = node

RETURN

ELSE

CURRENT = CURRENT.right

ENDIF

ENDIF

ENDWHILE

END FUNCTION

**Print All Courses in Alphanumeric Order (BST)**

FUNCTION printAllCoursesBST(BST):

inOrderPrint(BST.root)

END FUNCTION

FUNCTION inOrderPrint(node):

IF node == null THEN

RETURN

ENDIF

inOrderPrint(node.left)

PRINT node.course.courseNumber + ": " + node.course.courseTitle

inOrderPrint(node.right)

END FUNCTION

**Print One Course(BST)**

FUNCTION printCourseBST(BST, targetCourseNumber):

course = BSTSearch(BST.root, targetCourseNumber)

IF course == null THEN

PRINT "Course " + targetCourseNumber + " not found."

RETURN

ENDIF

PRINT "Course Number: " + course.courseNumber

PRINT "Title: " + course.courseTitle

IF course.prerequisites is not empty THEN

PRINT "Prerequisites:"

FOR EACH prereq IN course.prerequisites DO

PRINT " " + prereq

ENDFOR

ELSE

PRINT "Prerequisites: None"

ENDIF

END FUNCTION

FUNCTION BSTSearch(node, targetCourseNumber):

IF node == null THEN

RETURN null

ENDIF

IF targetCourseNumber == node.course.courseNumber THEN

RETURN node.course

ELSE IF targetCourseNumber < node.course.courseNumber THEN

RETURN BSTSearch(node.left, targetCourseNumber)

ELSE

RETURN BSTSearch(node.right, targetCourseNumber)

ENDIF

END FUNCTION

**Menu**

***\**** *menu function that validates user inputs, allows them to pick a data structure (Vector, Hash Table, BST), load data, print the sorted list, print details of one course, or exit.*

FUNCTION mainMenu():

// Prompt user for data structure choice, with basic validation:

dataStructureChoice = 0

WHILE dataStructureChoice NOT IN [1, 2, 3] DO

PRINT "Select Data Structure:"

PRINT " (1) Vector"

PRINT " (2) Hash Table"

PRINT " (3) Binary Search Tree"

dataStructureChoice = GET USER INPUT as integer

IF dataStructureChoice NOT IN [1, 2, 3] THEN

PRINT "Invalid selection. Please try again."

ENDIF

ENDWHILE

coursesLoaded = false

vectorCourses = empty vector

hashTable = create empty hash table

BST = create empty BST

WHILE true:

PRINT ""

PRINT "Menu Options:"

PRINT " 1. Load Data File"

PRINT " 2. Print Course List (Sorted)"

PRINT " 3. Print Course Information"

PRINT " 9. Exit"

userChoice = GET USER INPUT as integer

SWITCH (userChoice):

CASE 1:

filename = ASK USER: "Enter the file name:"

courseLines = loadCoursesFromFile(filename)

IF dataStructureChoice == 1:

vectorCourses = buildVectorFromLines(courseLines)

ELSE IF dataStructureChoice == 2:

hashTable = buildHashTableFromLines(courseLines)

ELSE IF dataStructureChoice == 3:

BST = buildBSTFromLines(courseLines)

ENDIF

coursesLoaded = true

PRINT "Data loaded successfully."

CASE 2:

IF coursesLoaded == false:

PRINT "No courses loaded. Please load the file first."

CONTINUE

// Print all courses in sorted order

IF dataStructureChoice == 1:

printAllCoursesVector(vectorCourses)

ELSE IF dataStructureChoice == 2:

printAllCoursesHash(hashTable)

ELSE IF dataStructureChoice == 3:

printAllCoursesBST(BST)

ENDIF

CASE 3:

IF coursesLoaded == false:

PRINT "No courses loaded. Please load the file first."

CONTINUE

targetCourse = ASK USER: "Enter course number to search for (e.g., CSCI100):"

IF dataStructureChoice == 1:

printCourseVector(vectorCourses, targetCourse)

ELSE IF dataStructureChoice == 2:

printCourseHash(hashTable, targetCourse)

ELSE IF dataStructureChoice == 3:

printCourseBST(BST, targetCourse)

ENDIF

CASE 9:

PRINT "Exiting program."

BREAK

DEFAULT:

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

END SWITCH

ENDWHILE

END FUNCTION

**Runtime Analysis**

When loading data and creating course objects, we assume there are n lines of valid course data in the file. Each line typically has a small, fixed number of tokens (the course number, title, and prerequisites). As a result, splitting each line into tokens can be considered O(1) for practical purposes, although in theory it could be O(m) if a line had many tokens. Reading n lines from a file amounts to O(n), and creating a Course object for each line is another O(n) total. The more significant differences come from inserting each course into a particular data structure.

For a Vector, insertion involves appending a new Course at the end of the array. This operation is O(1) amortized, so across all n courses, the total insertion time is O(n). In a Hash Table, insertion is O(1) on average for each course, which likewise produces O(n) total. However, if there are many collisions—perhaps due to a poor hashing function—worst-case performance can degrade to O(n^2). In a Binary Search Tree (BST), insertion time depends on the height of the tree. If the data happens to arrive in ascending or descending order, the tree will be extremely unbalanced, and each insertion can be O(n), leading to a total of O(n^2). If the input is random enough, or the BST is self-balancing, the average insertion cost is O(log n), resulting in O(n log n) over all inserts.

Searching and printing courses also differs among data structures. A Vector requires O(n) to find a single course by linear search. Printing a sorted list of all courses means sorting the entire vector first, which is O(n log n), then printing in O(n). In a Hash Table, searching for a specific course is O(1) on average, but printing a sorted list requires extracting and sorting the keys (O(n) + O(n log n) to sort, plus O(n) to print), making it O(n log n) overall. By contrast, a BST supports an in-order traversal in O(n) to naturally produce a sorted list, and searching is O(log n) on average. Still, if the BST is unbalanced, both insertion and searching degrade to O(n).

**Advantages and Disadvantages of Each Structure**

Each data structure comes with its own strengths and weaknesses. A Vector is appealing for its simplicity. It is straightforward to implement, storing all data contiguously, and appending new courses is an amortized O(1). However, any search for a specific course number requires scanning the entire array (O(n)), and printing a sorted list involves O(n log n) every time the user needs a complete, ordered listing.

A Hash Table is attractive because it offers O(1) average-case insertion and lookup, making it extremely fast when advisors need to locate a specific course by its course number. On the downside, generating a sorted list of all courses requires collecting and sorting keys, adding an O(n log n) operation. Collisions in a hash table can also degrade performance to O(n), and implementing a hash function and collision resolution strategy can be more intricate compared to a simple array-based approach.

Lastly, a BST (especially if it is self-balancing) can provide a nice balance. An in-order traversal of a BST visits all nodes in ascending order, so printing a sorted list takes O(n). Searching is typically O(log n), as long as the tree remains balanced. The main drawback is that a naive BST can become skewed if the data is sorted, resulting in O(n) insertion and searching. Self-balancing BSTs (like AVL or Red-Black trees) solve that problem but at the cost of more complex implementation.

**Recommendation**

Considering the advising office’s key requirements finding specific courses quickly and occasionally printing a complete sorted list either a Hash Table or a BST can be effective. A Hash Table will provide the fastest lookups on average (O(1)), but sorting all course numbers when needed is O(n log n). This trade-off is often acceptable if the advisors spend most of their time looking up individual courses and only occasionally require a full sorted listing. On the other hand, a BST easily produces a sorted list via an in-order traversal, which is O(n), and searching is O(log n) when the tree is balanced. However, there is a risk of poor performance if the data arrives in a way that creates a highly unbalanced tree, unless a self-balancing approach is used.

For most typical scenarios, a Hash Table is the simpler go-to solution for rapid lookups, and paying the O(n log n) cost to sort courses on demand is reasonable. If the data set is large and you want consistently fast sorting without re-sorting, a self-balancing BST might be the more optimal choice, as it ensures searching remains O(log n) and provides O(n) sorted printing. Ultimately, the decision should reflect how often advisors expect to retrieve single courses versus how often they need to view all courses in order.