**CS-300**

# **CS 300 Project One: Pseudocode and Runtime Analysis**

**Hamad Alghaithi**

**Introduction**

This document presents pseudocode implementations for ABCU's course advising program using three different data structures: Vector, Hash Table, and Binary Search Tree. Each implementation includes file loading, course object creation, menu functionality, and course information display capabilities. A comprehensive runtime analysis follows to determine the optimal data structure for this application.

## Vector Data Structure Implementation

### File Input and Validation Pseudocode

### Course Object Structure

Function LoadCoursesVector(fileName):

Initialize courseVector as Vector<Course>

Initialize allCourseNumbers as empty Set<String>

// First pass: collect all course numbers for validation

OPEN file with fileName

IF file cannot open THEN

PRINT "Error: Cannot open file"

RETURN empty vector

END IF

WHILE not end of file:

READ line from file

SPLIT line by commas into fields

IF number of fields < 2:

CONTINUE to next line

END IF

SET courseNumber = TRIM(fields[0])

ADD courseNumber to allCourseNumbers

END WHILE

CLOSE file

// Second pass: parse and validate course data

REOPEN file with fileName

WHILE not end of file:

READ line from file

SPLIT line by commas into fields

IF number of fields < 2:

PRINT "Invalid line format: " + line

CONTINUE to next line

END IF

SET courseNumber = TRIM(fields[0])

SET courseTitle = TRIM(fields[1])

Initialize prerequisites as empty Vector<String>

FOR index from 2 to end of fields:

SET prereq = TRIM(fields[index])

IF prereq NOT IN allCourseNumbers:

PRINT "Error: Prerequisite " + prereq + " not found"

CONTINUE

END IF

ADD prereq to prerequisites

END FOR

CREATE Course object with courseNumber, courseTitle, prerequisites

ADD Course object to courseVector

END WHILE

CLOSE file

RETURN courseVector

Structure Course

String courseNumber

String title

Vector<String> prerequisites

Constructor Course(courseNumber, title, prerequisites):

SET this.courseNumber = courseNumber

SET this.title = title

SET this.prerequisites = prerequisites

### Menu System Pseudocode

Function DisplayMenu(courseVector):

WHILE true:

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "9. Exit"

PRINT "What would you like to do? "

INPUT choice

SWITCH choice:

CASE "1":

courseVector = LoadCoursesVector("courses.txt")

PRINT "Courses loaded successfully"

BREAK

CASE "2":

PrintSortedCourseList(courseVector)

BREAK

CASE "3":

PRINT "What course do you want to know about? "

INPUT courseNumber

SearchAndPrintCourse(courseVector, courseNumber)

BREAK

CASE "9":

PRINT "Thank you for using the course planner!"

EXIT

BREAK

DEFAULT:

PRINT "Invalid option. Please try again."

### Print Sorted Course List Pseudocode

Function PrintSortedCourseList(courseVector):

IF courseVector is empty:

PRINT "No courses loaded. Please load data first."

RETURN

END IF

// Sort courses alphanumerically by course number

FOR i = 0 to courseVector.size() - 2:

FOR j = i + 1 to courseVector.size() - 1:

IF courseVector[i].courseNumber > courseVector[j].courseNumber:

SWAP courseVector[i] and courseVector[j]

END IF

END FOR

END FOR

PRINT "Here is a sample schedule:"

FOR each course in courseVector:

PRINT course.courseNumber + ", " + course.title

END FOR

### Search and Print Course Information

Function SearchAndPrintCourse(courseVector, searchCourseNumber):

FOR each course in courseVector:

IF course.courseNumber == searchCourseNumber:

PRINT course.courseNumber + ", " + course.title

IF course.prerequisites is not empty:

PRINT "Prerequisites: "

FOR each prereq in course.prerequisites:

PRINT prereq

END FOR

ELSE:

PRINT "No prerequisites"

END IF

RETURN

END IF

END FOR

PRINT "Course not found."

## Hash Table Data Structure Implementation

### File Input and Validation Pseudocode

Function LoadCoursesHashTable(fileName):

Initialize courseTable as HashTable<String, Course>

Initialize allCourseNumbers as empty Set<String>

// First pass: collect all course numbers

OPEN file with fileName

IF file cannot open THEN

PRINT "Error: Cannot open file"

RETURN empty hash table

END IF

WHILE not end of file:

READ line from file

SPLIT line by commas into fields

IF number of fields < 2:

CONTINUE to next line

END IF

SET courseNumber = TRIM(fields[0])

ADD courseNumber to allCourseNumbers

END WHILE

CLOSE file

// Second pass: validate and store courses

REOPEN file with fileName

WHILE not end of file:

READ line from file

SPLIT line by commas into fields

IF number of fields < 2:

PRINT "Invalid line format: " + line

CONTINUE to next line

END IF

SET courseNumber = TRIM(fields[0])

SET courseTitle = TRIM(fields[1])

Initialize prerequisites as empty Vector<String>

FOR index from 2 to end of fields:

SET prereq = TRIM(fields[index])

IF prereq NOT IN allCourseNumbers:

PRINT "Error: Prerequisite " + prereq + " not found"

CONTINUE

END IF

ADD prereq to prerequisites

END FOR

CREATE Course object with courseNumber, courseTitle, prerequisites

INSERT Course into courseTable with courseNumber as key

END WHILE

CLOSE file

RETURN courseTable

### Hash Table Menu System

Function DisplayMenuHashTable(courseTable):

WHILE true:

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course"

PRINT "9. Exit"

PRINT "What would you like to do? "

INPUT choice

SWITCH choice:

CASE "1":

courseTable = LoadCoursesHashTable("courses.txt")

PRINT "Courses loaded successfully"

BREAK

CASE "2":

PrintSortedCourseListHashTable(courseTable)

BREAK

CASE "3":

PRINT "What course do you want to know about? "

INPUT courseNumber

SearchAndPrintCourseHashTable(courseTable, courseNumber)

BREAK

CASE "9":

PRINT "Thank you for using the course planner!"

EXIT

BREAK

DEFAULT:

PRINT "Invalid option. Please try again."

### Hash Table Print Sorted List

Function PrintSortedCourseListHashTable(courseTable):

IF courseTable is empty:

PRINT "No courses loaded. Please load data first."

RETURN

END IF

Initialize courseList as Vector<Course>

FOR each key in courseTable:

ADD courseTable[key] to courseList

END FOR

// Sort courses alphanumerically

FOR i = 0 to courseList.size() - 2:

FOR j = i + 1 to courseList.size() - 1:

IF courseList[i].courseNumber > courseList[j].courseNumber:

SWAP courseList[i] and courseList[j]

END IF

END FOR

END FOR

PRINT "Here is a sample schedule:"

FOR each course in courseList:

PRINT course.courseNumber + ", " + course.title

END FOR

### Hash Table Search Function

Function SearchAndPrintCourseHashTable(courseTable, searchCourseNumber):

IF searchCourseNumber EXISTS in courseTable:

SET course = courseTable[searchCourseNumber]

PRINT course.courseNumber + ", " + course.title

IF course.prerequisites is not empty:

PRINT "Prerequisites: "

FOR each prereq in course.prerequisites:

PRINT prereq

END FOR

ELSE:

PRINT "No prerequisites"

END IF

ELSE:

PRINT "Course not found."

END IF

## Binary Search Tree Data Structure Implementation

### BST Node Structure

Structure TreeNode:

Course course

TreeNode\* left

TreeNode\* right

Structure BinarySearchTree:

TreeNode\* root

### File Input and BST Creation

Function LoadCoursesBST(fileName):

Initialize courseBST as BinarySearchTree

Initialize allCourseNumbers as empty Set<String>

// First pass: collect all course numbers

OPEN file with fileName

IF file cannot open THEN

PRINT "Error: Cannot open file"

RETURN empty BST

END IF

WHILE not end of file:

READ line from file

SPLIT line by commas into fields

IF number of fields < 2:

CONTINUE to next line

END IF

SET courseNumber = TRIM(fields[0])

ADD courseNumber to allCourseNumbers

END WHILE

CLOSE file

// Second pass: validate and insert into BST

REOPEN file with fileName

WHILE not end of file:

READ line from file

SPLIT line by commas into fields

IF number of fields < 2:

CONTINUE to next line

END IF

SET courseNumber = TRIM(fields[0])

SET courseTitle = TRIM(fields[1])

Initialize prerequisites as empty Vector<String>

FOR index from 2 to end of fields:

SET prereq = TRIM(fields[index])

IF prereq NOT IN allCourseNumbers:

PRINT "Error: Prerequisite " + prereq + " not found"

CONTINUE

END IF

ADD prereq to prerequisites

END FOR

CREATE Course object with courseNumber, courseTitle, prerequisites

INSERT Course into courseBST

END WHILE

CLOSE file

RETURN courseBST

### BST Insert Function

Function InsertNode(root, course):

IF root is NULL:

CREATE new TreeNode with course

RETURN new node

END IF

IF course.courseNumber < root.course.courseNumber:

root.left = InsertNode(root.left, course)

ELSE:

root.right = InsertNode(root.right, course)

END IF

RETURN root

### BST In-Order Traversal for Sorted Printing

Function PrintSortedCourseListBST(courseBST):

IF courseBST.root is NULL:

PRINT "No courses loaded. Please load data first."

RETURN

END IF

PRINT "Here is a sample schedule:"

InOrderTraversal(courseBST.root)

Function InOrderTraversal(node):

IF node is not NULL:

InOrderTraversal(node.left)

PRINT node.course.courseNumber + ", " + node.course.title

InOrderTraversal(node.right)

END IF

### BST Search Function

Function SearchAndPrintCourseBST(courseBST, searchCourseNumber):

SET foundCourse = SearchBST(courseBST.root, searchCourseNumber)

IF foundCourse is not NULL:

PRINT foundCourse.courseNumber + ", " + foundCourse.title

IF foundCourse.prerequisites is not empty:

PRINT "Prerequisites: "

FOR each prereq in foundCourse.prerequisites:

PRINT prereq

END FOR

ELSE:

PRINT "No prerequisites"

END IF

ELSE:

PRINT "Course not found."

END IF

Function SearchBST(node, courseNumber):

IF node is NULL OR node.course.courseNumber == courseNumber:

RETURN node

END IF

IF courseNumber < node.course.courseNumber:

RETURN SearchBST(node.left, courseNumber)

ELSE:

RETURN SearchBST(node.right, courseNumber)

END IF

## Runtime Analysis

### Vector Data Structure Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Code | Line Cost | # Times Executes | Total Cost |
| Load file (read all lines) | 1 | n | n |
| Parse each line | 1 | n | n |
| Validate prerequisites | 1 | n × k | n × k |
| Create course objects | 1 | N | n |
| **Loading Total** |  |  | **O(n × k)** |
| Search for course (linear) | 1 | n | n |
| Print course info | 1 | 1 | 1 |
| **Search Total** |  |  | **O(n)** |
| Sort courses (bubble sort) | 1 | n² | n² |
| Print sorted list | 1 | n | n |
| **Sort/Print Total** |  |  | **O(n²)** |

**Overall Vector Runtime: O(n²)** due to sorting

### Hash Table Data Structure Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Load file (read all lines) | 1 | n | n |
| Parse each line | 1 | n | n |
| Hash table insertion | 1 | n | n |
| **Loading Total** |  |  | **O(n)** |
| Hash table lookup | 1 | 1 (average) | 1 |
| Print course info | 1 | 1 | 1 |
| **Search Total** |  |  | **O(1) average** |
| Extract all courses to vector | 1 | n | n |
| Sort extracted courses | 1 | n² | n² |
| **Sort/Print Total** |  |  | **O(n²)** |

**Overall Hash Table Runtime: O(n²)** due to sorting for display

### Binary Search Tree Data Structure Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| Load file (read all lines) | 1 | n | n |
| Parse each line | 1 | n | n |
| BST insertion | 1 | n × log n (average) | n × log n |
| **Loading Total** |  |  | **O(n log n)** |
| BST search | 1 | log n (average) | log n |
| Print course info | 1 | 1 | 1 |
| **Search Total** |  |  | **O(log n)** |
| In-order traversal | 1 | n | n |
| **Sort/Print Total** |  |  | **O(n)** |

**Overall BST Runtime: O(n log n)** for loading, O(log n) for search

## Data Structure Comparison

### Vector Advantages:

* Simple implementation and easy to understand
* Good memory locality for sequential access
* No additional memory overhead for pointers
* Straightforward to implement and debug

### Vector Disadvantages:

* O(n) search time for finding specific courses
* O(n²) sorting time using simple algorithms
* Inefficient for frequent lookups
* Poor scalability as course catalog grows

### Hash Table Advantages:

* O(1) average-case search time for course lookup
* Excellent performance for frequent searches
* Direct access to courses by course number
* Scales well with large datasets

### Hash Table Disadvantages:

* O(n²) time complexity for sorted output
* Potential for hash collisions in worst case
* Additional memory overhead for hash structure
* More complex implementation than vector

### Binary Search Tree Advantages:

* O(log n) search time (balanced tree)
* Natural ordering provides O(n) sorted traversal
* Good balance between search and sort performance
* Efficient memory usage compared to hash table

### Binary Search Tree Disadvantages:

* Can degrade to O(n) in worst case (unbalanced)
* More complex implementation than vector
* Requires careful balancing for optimal performance
* Recursive operations may cause stack overflow with very large datasets

## Recommendation

Based on the runtime analysis and requirements for ABCU's advising program, I recommend using the **Hash Table** data structure for the following reasons:

1. **Primary Use Case Optimization**: The most frequent operation will be looking up individual courses (Option 3), where hash tables provide O(1) average-case performance compared to O(n) for vectors and O(log n) for BSTs.
2. **Acceptable Trade-offs**: While sorted printing (Option 2) requires O(n²) time due to extraction and sorting, this operation will be used less frequently than individual course lookups.
3. **Scalability**: As ABCU's course catalog grows, the hash table will maintain constant-time lookups, while vector performance degrades linearly and BST performance increases logarithmically.
4. **Real-world Performance**: Academic advisors will primarily search for specific courses during student meetings, making the O(1) lookup time crucial for user experience.
5. **Implementation Simplicity**: Hash tables provide the best performance for the primary use case while remaining relatively straightforward to implement and maintain.

The hash table strikes the optimal balance between the most important operation (course lookup) and acceptable performance for less frequent operations (sorted display), making it the best choice for this academic advising application.