**Project One**

**1. Common Definitions and Helpers**

STRUCT Course:

STRING courseNumber

STRING courseTitle

LIST of STRING prerequisites

END STRUCT

FUNCTION createCourseObject(line):

tokens = SPLIT(line, ",")

IF LENGTH(tokens) < 2 THEN

PRINT "Error: Invalid format -", line

RETURN NULL

END IF

course.courseNumber = TRIM(tokens[0])

course.courseTitle = TRIM(tokens[1])

course.prerequisites = EMPTY LIST

IF LENGTH(tokens) > 2 THEN

FOR i FROM 2 TO LENGTH(tokens)-1:

ADD TRIM(tokens[i]) TO course.prerequisites

END FOR

END IF

RETURN course

END FUNCTION

FUNCTION validatePrerequisites(courses):

FOR EACH course IN courses DO:

FOR EACH prereq IN course.prerequisites DO:

FOUND = FALSE

FOR EACH otherCourse IN courses DO:

IF otherCourse.courseNumber == prereq THEN

FOUND = TRUE

BREAK

END IF

END FOR

IF NOT FOUND THEN

PRINT "Error: Prerequisite", prereq, "not found for", course.courseNumber

END IF

END FOR

END FOR

END FUNCTION

**2. Data Structure Specific Implementations**

**Vector Implementation**

GLOBAL courseList = EMPTY VECTOR

FUNCTION loadCoursesVector(filename):

file = OPEN(filename)

IF file NOT OPENED THEN:

PRINT "Error: Unable to open", filename

RETURN

END IF

WHILE NOT END\_OF\_FILE(file):

line = READ\_LINE(file)

course = createCourseObject(line)

IF course IS NOT NULL THEN:

APPEND course TO courseList

END IF

END WHILE

CLOSE(file)

CALL validatePrerequisites(courseList)

END FUNCTION

FUNCTION printSortedCoursesVector():

SORT courseList BY course.courseNumber

FOR EACH course IN courseList DO:

PRINT course.courseNumber, ":", course.courseTitle

END FOR

END FUNCTION

FUNCTION printCourseInfoVector(searchCourseNumber):

FOUND = FALSE

FOR EACH course IN courseList DO:

IF course.courseNumber == searchCourseNumber THEN:

PRINT "Course Number: " + course.courseNumber

PRINT "Course Title: " + course.courseTitle

IF course.prerequisites IS EMPTY THEN:

PRINT "Prerequisites: None"

ELSE:

PRINT "Prerequisites: " + JOIN(course.prerequisites, ", ")

END IF

FOUND = TRUE

BREAK

END IF

END FOR

IF NOT FOUND THEN:

PRINT "Error: Course", searchCourseNumber, "not found."

END IF

END FUNCTION

**B. Hash Table Implementation**

GLOBAL courseTable = EMPTY HASH TABLE

FUNCTION loadCoursesHashTable(filename):

file = OPEN(filename)

IF file NOT OPENED THEN:

PRINT "Error: Unable to open", filename

RETURN

END IF

WHILE NOT END\_OF\_FILE(file):

line = READ\_LINE(file)

course = createCourseObject(line)

IF course IS NOT NULL THEN:

INSERT course INTO courseTable WITH KEY course.courseNumber

END IF

END WHILE

CLOSE(file)

courses = GET\_ALL\_VALUES(courseTable)

CALL validatePrerequisites(courses)

END FUNCTION

FUNCTION printSortedCoursesHashTable():

keys = GET\_ALL\_KEYS(courseTable)

SORT keys ALPHANUMERICALLY

FOR EACH key IN keys DO:

course = courseTable[key]

PRINT course.courseNumber, ":", course.courseTitle

END FOR

END FUNCTION

FUNCTION printCourseInfoHashTable(searchCourseNumber):

IF searchCourseNumber EXISTS IN courseTable THEN:

course = courseTable[searchCourseNumber]

PRINT "Course Number: " + course.courseNumber

PRINT "Course Title: " + course.courseTitle

IF course.prerequisites IS EMPTY THEN:

PRINT "Prerequisites: None"

ELSE:

PRINT "Prerequisites: " + JOIN(course.prerequisites, ", ")

END IF

ELSE:

PRINT "Error: Course", searchCourseNumber, "not found."

END IF

END FUNCTION

**C. Binary Search Tree (BST) Implementation**

STRUCT BSTNode:

Course course

BSTNode left

BSTNode right

END STRUCT

GLOBAL courseTree = { root: NULL }

FUNCTION insertIntoBST(node, course):

IF node IS NULL THEN:

node = CREATE NEW BSTNode WITH course

RETURN node

END IF

IF course.courseNumber < node.course.courseNumber THEN:

node.left = insertIntoBST(node.left, course)

ELSE:

node.right = insertIntoBST(node.right, course)

END IF

RETURN node

END FUNCTION

FUNCTION loadCoursesBST(filename):

file = OPEN(filename)

IF file NOT OPENED THEN:

PRINT "Error: Unable to open", filename

RETURN

END IF

WHILE NOT END\_OF\_FILE(file):

line = READ\_LINE(file)

course = createCourseObject(line)

IF course IS NOT NULL THEN:

courseTree.root = insertIntoBST(courseTree.root, course)

END IF

END WHILE

CLOSE(file)

courses = EMPTY VECTOR

CALL inOrderCollect(courseTree.root, courses)

CALL validatePrerequisites(courses)

END FUNCTION

FUNCTION inOrderCollect(node, list):

IF node IS NOT NULL THEN:

CALL inOrderCollect(node.left, list)

ADD node.course TO list

CALL inOrderCollect(node.right, list)

END IF

END FUNCTION

FUNCTION printSortedCoursesBST():

CALL inOrderTraversal(courseTree.root)

END FUNCTION

FUNCTION inOrderTraversal(node):

IF node IS NOT NULL THEN:

CALL inOrderTraversal(node.left)

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

CALL inOrderTraversal(node.right)

END IF

END FUNCTION

FUNCTION BSTSearch(node, searchCourseNumber):

IF node IS NULL THEN:

RETURN NULL

END IF

IF node.course.courseNumber == searchCourseNumber THEN:

RETURN node

ELSE IF searchCourseNumber < node.course.courseNumber THEN:

RETURN BSTSearch(node.left, searchCourseNumber)

ELSE:

RETURN BSTSearch(node.right, searchCourseNumber)

END IF

END FUNCTION

FUNCTION printCourseInfoBST(searchCourseNumber):

node = BSTSearch(courseTree.root, searchCourseNumber)

IF node IS NOT NULL THEN:

PRINT "Course Number: " + node.course.courseNumber

PRINT "Course Title: " + node.course.courseTitle

IF node.course.prerequisites IS EMPTY THEN:

PRINT "Prerequisites: None"

ELSE:

PRINT "Prerequisites: " + JOIN(node.course.prerequisites, ", ")

END IF

ELSE:

PRINT "Error: Course", searchCourseNumber, "not found."

END IF

END FUNCTION

**3. Main Program Flow**

FUNCTION main():

DISPLAY "ABCU CS Course Advisor"

DISPLAY "Select Implementation: 1 = Vector, 2 = Hash Table, 3 = BST"

dsChoice = GET\_USER\_INPUT("Enter choice (1-3): ")

REPEAT

DISPLAY "Menu Options: 1. Load Data, 2. Print Sorted List, 3. Print Course Info, 9. Exit"

option = GET\_USER\_INPUT("Enter option: ")

IF option == 1 THEN:

filename = "courses.txt"

IF dsChoice == 1 THEN:

CALL loadCoursesVector(filename)

ELSE IF dsChoice == 2 THEN:

CALL loadCoursesHashTable(filename)

ELSE IF dsChoice == 3 THEN:

CALL loadCoursesBST(filename)

END IF

ELSE IF option == 2 THEN:

IF dsChoice == 1 THEN:

CALL printSortedCoursesVector()

ELSE IF dsChoice == 2 THEN:

CALL printSortedCoursesHashTable()

ELSE IF dsChoice == 3 THEN:

CALL printSortedCoursesBST()

END IF

ELSE IF option == 3 THEN:

searchCourseNumber = GET\_USER\_INPUT("Enter course number: ")

IF dsChoice == 1 THEN:

CALL printCourseInfoVector(searchCourseNumber)

ELSE IF dsChoice == 2 THEN:

CALL printCourseInfoHashTable(searchCourseNumber)

ELSE IF dsChoice == 3 THEN:

CALL printCourseInfoBST(searchCourseNumber)

END IF

ELSE IF option == 9 THEN:

PRINT "Exiting..."

EXIT

ELSE:

PRINT "Invalid option"

END IF

UNTIL option == 9

END FUNCTION

|  |  |  |  |
| --- | --- | --- | --- |
| **Step** | **Cost Per Execution** | **Executions** | **Total Cost** |
| **Open File** | 1 | 1 | 1 |
| **For each line (n courses)** |  |  |  |
| -Read line | 1 | n | n |
| -Split line into tokens | 1 | n | n |
| -Check token count | 1 | n | n |
| -Extract course number and title | 2 | n | 2n |
| -Create empty prerequisites list | 1 | n | n |
| -Loop over prerequisites (constant c) | c (constant) | n | c\*n |
| -Create and return Course object | 1 | n | n |
| **Insertion into Data Structure:** |  |  |  |
| -Vector (append) | 1 | n | n (amortize) |
| -Hash table (insert) | 1 | n | n (average case) |
| -BST (balanced insertion) | O (log n) | n | O (n log n) |

**Analysis – Vector:**

Vectors are straightforward to implement and offer excellent memory locality because they use contiguous memory, leading to good cache performance. Inserting a course into a vector is efficient with an amortized O(1) time per insertion. However, since vectors do not inherently maintain sorted order, you must sort the vector each time you want to print the courses in alphanumeric order, which takes O(n log n) time. Additionally, searching for a course in an unsorted vector can require a linear scan (O(n) in the worst case) unless you keep the vector sorted at all times.

**Hash Table:**

Hash tables provide extremely fast average-case lookup times (O(1)), which makes retrieving a course by its course number very efficient. However, hash tables do not maintain any inherent order among the keys. To print a sorted list of courses, you need to extract all keys and sort them, resulting in an O(n log n) operation. Also, while average-case insertion is O(1), worst-case scenarios (due to collisions) could degrade performance if not handled properly. Memory usage can also be slightly higher due to additional overhead for handling collisions.

**BST:**

A balanced BST, such as an AVL tree or Red-Black tree, automatically maintains the data in sorted order. This means that an in-order traversal can print the courses in O(n) time without any extra sorting step. Lookups and insertions in a balanced BST are efficient, with O(log n) time per operation. The main drawbacks are that the insertion process is more complex compared to vectors or hash tables, and ensuring the tree remains balanced can add overhead. Additionally, the memory usage is slightly higher due to the pointers required for each node.

**Recommendation:**

Based on the advisor’s requirements—to efficiently print an alphanumerically sorted list of courses and quickly search for specific course details—the balanced BST is the recommended data structure. Although the BST has a slightly higher insertion cost (O(n log n) overall) compared to the vector or hash table, its ability to maintain sorted order inherently (allowing an O(n) in-order traversal for printing) and provide efficient O(log n) lookups for course details make it the best overall choice for this application.