CS 300 – Project One: Pseudocode and Runtime Analysis

Part 1: Vector Data Structure Pseudocode

BEGIN PROGRAM

DEFINE STRUCT Course

courseNumber : STRING

courseTitle : STRING

prerequisites : LIST OF STRING

END STRUCT

DEFINE VECTOR courseList

FUNCTION loadCoursesFromFile(fileName)

OPEN fileName FOR reading

IF file cannot be opened THEN

PRINT "Error: File not found"

EXIT FUNCTION

END IF

WHILE NOT end of file

READ line

SPLIT line by comma INTO tokens

IF number of tokens < 2 THEN

PRINT "Error: invalid format"

CONTINUE

END IF

CREATE newCourse

newCourse.courseNumber = tokens[0]

newCourse.courseTitle = tokens[1]

FOR i FROM 2 TO tokens.size - 1

ADD tokens[i] TO newCourse.prerequisites

END FOR

APPEND newCourse TO courseList

END WHILE

CLOSE file

END FUNCTION

FUNCTION printAllCourses()

SORT courseList BY courseNumber ascending

FOR EACH course IN courseList

PRINT course.courseNumber + ", " + course.courseTitle

END FOR

END FUNCTION

FUNCTION printCourseDetails(courseNumber)

FOR EACH course IN courseList

IF course.courseNumber == courseNumber THEN

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

IF course.prerequisites IS EMPTY THEN

PRINT "Prerequisites: None"

ELSE

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

END IF

RETURN

END IF

END FOR

PRINT "Course not found."

END FUNCTION

FUNCTION showMenu()

REPEAT

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course Information"

PRINT "9. Exit"

READ userChoice

IF userChoice == 1 THEN

CALL loadCoursesFromFile("course\_data.txt")

ELSE IF userChoice == 2 THEN

CALL printAllCourses()

ELSE IF userChoice == 3 THEN

PROMPT "Enter course number:"

READ courseNumber

CALL printCourseDetails(courseNumber)

END IF

UNTIL userChoice == 9

END FUNCTION

CALL showMenu()

END PROGRAM

Part 2: Hash Table Data Structure Pseudocode

BEGIN PROGRAM

DEFINE STRUCT Course

courseNumber : STRING

courseTitle : STRING

prerequisites : LIST OF STRING

END STRUCT

DEFINE CLASS HashTable

DEFINE ARRAY table[DEFAULT\_SIZE] OF LISTS // each index can hold multiple courses for collision handling

FUNCTION hash(courseNumber)

hashValue = 0

FOR EACH character IN courseNumber

hashValue = (hashValue + ASCII(character)) MOD DEFAULT\_SIZE

END FOR

RETURN hashValue

END FUNCTION

FUNCTION insert(course)

index = hash(course.courseNumber)

APPEND course TO table[index]

END FUNCTION

FUNCTION search(courseNumber)

index = hash(courseNumber)

FOR EACH course IN table[index]

IF course.courseNumber == courseNumber THEN

RETURN course

END IF

END FOR

RETURN null

END FUNCTION

FUNCTION printAllCourses()

FOR EACH bucket IN table

FOR EACH course IN bucket

PRINT course.courseNumber + ", " + course.courseTitle

END FOR

END FOR

END FUNCTION

END CLASS

FUNCTION loadCoursesFromFile(fileName)

OPEN fileName FOR reading

IF file cannot be opened THEN

PRINT "Error: File not found"

EXIT FUNCTION

END IF

WHILE NOT end of file

READ line

SPLIT line by comma INTO tokens

IF number of tokens < 2 THEN

PRINT "Error: invalid format"

CONTINUE

END IF

CREATE newCourse

newCourse.courseNumber = tokens[0]

newCourse.courseTitle = tokens[1]

FOR i FROM 2 TO tokens.size - 1

ADD tokens[i] TO newCourse.prerequisites

END FOR

CALL hashTable.insert(newCourse)

END WHILE

CLOSE file

END FUNCTION

FUNCTION printCourseDetails(courseNumber)

course = hashTable.search(courseNumber)

IF course IS NOT null THEN

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

IF course.prerequisites IS EMPTY THEN

PRINT "Prerequisites: None"

ELSE

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

END IF

ELSE

PRINT "Course not found."

END IF

END FUNCTION

FUNCTION showMenu()

REPEAT

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course Information"

PRINT "9. Exit"

READ userChoice

IF userChoice == 1 THEN

CALL loadCoursesFromFile("course\_data.txt")

ELSE IF userChoice == 2 THEN

CALL hashTable.printAllCourses()

ELSE IF userChoice == 3 THEN

PROMPT "Enter course number:"

READ courseNumber

CALL printCourseDetails(courseNumber)

END IF

UNTIL userChoice == 9

END FUNCTION

// MAIN PROGRAM

CREATE hashTable

CALL showMenu()

END PROGRAM

Part 3: Binary Search Tree Data Structure Pseudocode

BEGIN PROGRAM

DEFINE STRUCT Course

courseNumber : STRING

courseTitle : STRING

prerequisites : LIST OF STRING

END STRUCT

DEFINE STRUCT Node

Course : Course

Left : POINTER TO Node

Right : POINTER TO Node

END STRUCT

DEFINE CLASS BinarySearchTree

Root : POINTER TO Node

FUNCTION insert(course)

root = addNode(root, course)

END FUNCTION

FUNCTION addNode(node, course)

IF node IS NULL THEN

CREATE newNode WITH course

RETURN newNode

END IF

IF course.courseNumber < node.course.courseNumber THEN

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

ELSE

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

END IF

RETURN node

END FUNCTION

FUNCTION search(courseNumber)

current = root

WHILE current IS NOT NULL

IF current.course.courseNumber == courseNumber THEN

RETURN current.course

ELSE IF courseNumber < current.course.courseNumber THEN

current = current.left

ELSE

current = current.right

END IF

END WHILE

RETURN null

END FUNCTION

FUNCTION printInOrder(node)

IF node IS NOT NULL THEN

printInOrder(node.left)

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

printInOrder(node.right)

END IF

END FUNCTION

END CLASS

FUNCTION loadCoursesFromFile(fileName)

OPEN fileName FOR reading

IF file cannot be opened THEN

PRINT "Error: File not found"

EXIT FUNCTION

END IF

WHILE NOT end of file

READ line

SPLIT line by comma INTO tokens

IF number of tokens < 2 THEN

PRINT "Error: invalid format"

CONTINUE

END IF

CREATE newCourse

newCourse.courseNumber = tokens[0]

newCourse.courseTitle = tokens[1]

FOR i FROM 2 TO tokens.size - 1

ADD tokens[i] TO newCourse.prerequisites

END FOR

CALL bst.insert(newCourse)

END WHILE

CLOSE file

END FUNCTION

FUNCTION printCourseDetails(courseNumber)

course = bst.search(courseNumber)

IF course IS NOT null THEN

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

IF course.prerequisites IS EMPTY THEN

PRINT "Prerequisites: None"

ELSE

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

END IF

ELSE

PRINT "Course not found."

END IF

END FUNCTION

FUNCTION showMenu()

REPEAT

PRINT "1. Load Data Structure"

PRINT "2. Print Course List"

PRINT "3. Print Course Information"

PRINT "9. Exit"

READ userChoice

IF userChoice == 1 THEN

CALL loadCoursesFromFile("course\_data.txt")

ELSE IF userChoice == 2 THEN

PRINT "Course List (In Alphanumeric Order):"

CALL bst.printInOrder(bst.root)

ELSE IF userChoice == 3 THEN

PROMPT "Enter course number:"

READ courseNumber

CALL printCourseDetails(courseNumber)

END IF

UNTIL userChoice == 9

END FUNCTION

// MAIN PROGRAM

CREATE bst AS BinarySearchTree

CALL showMenu()

END PROGRAM

Part 4: Runtime Analysis

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation/Structure** | **Vector** | **Hash Table** | **Binary Search Tree** |
| **File Read & Parse** | O(n) – each line read once | O(n) – each line read once | O(n) – each line read once |
| **Insert Course** | O(1) – append  O(n) – if inserting sorted | O(1) – average  O(n) – worst | O(log n) – average  O(n) – worst |
| **Search for Course** | O(n) – linear search | O(1) – average  O(n) – worst | O(log n) – average  O(n) – worst |
| **Print Sorted List** | O(n log n) – sort | O(n + k log k) | O(n) |
| **Memory Usage** | Compact and grows linearly | Moderate due to buckets | Moderate due to node pointers |
| **Advantages** | Simple, easy to implement | Very fast lookups, ideal for direct access | Keeps data sorted, efficient for searches and inserts |
| **Disadvantages** | Slow searching/sorting for large data sets | Loses order, possible collisions | Can become unbalanced, uses more memory |

Part 5: Evaluation and Recommendation

After analyzing all three data structures, the vector, hash table, and binary search tree each have strengths depending on the program’s needs. The vector is simple to build and great for small data sets, but as the number of courses grows, searching and sorting become inefficient because both require linear or log-linear time. The hash table is the fastest for lookups and insertions on average, but it does not preserve sorted order and can experience collisions that slow down performance.

The binary search tree offers the best balance of efficiency and functionality. It allows for quick searching and insertion while automatically maintaining courses in alphanumeric order through in-order traversal. Even though it can become unbalanced in the worst case, it performs consistently well for most data sets and meets the advisor’s needs to both print sorted course lists and search by course. Therefore, I recommend implementing the **binary search tree** as the primary data structure for ABCU’s course advising program because it provides efficient runtime performance and a naturally sorted output.