CS 300 Project One

Nick Williams

Southern New Hampshire University

Jayantha Muthukudage

10/07/25

**Pseudocode**:

PROCEDURE MainMenu()

REPEAT

PRINT "1. Load Data"

PRINT "2. Print Course List (sorted)"

PRINT "3. Print Course Information"

PRINT "9. Exit"

INPUT choice

IF choice = 1 THEN

INPUT path

LoadCourses(path) // DS-specific implementation

ELSE IF choice = 2 THEN

PrintAllCoursesSorted() // DS-specific

ELSE IF choice = 3 THEN

INPUT target

PrintSingleCourse(target) // DS-specific

ELSE IF choice = 9 THEN

PRINT "Goodbye"

EXIT LOOP

ELSE

PRINT "Invalid option"

END IF

UNTIL FALSE

END PROCEDURE

DATA STRUCTURES

Course:

number : STRING

title : STRING

prereqs : LIST of STRING

courses : LIST of Course // vector of Course

courseIds : LIST of STRING // vector of known IDs for validation

FUNCTION LoadCourses(filePath : STRING) RETURNS BOOL

IF file cannot be opened THEN

PRINT "Error: file not found"

RETURN FALSE

END IF

CLEAR courses

CLEAR courseIds

// Pass 1: read lines → build Course objects

FOR EACH line IN file

toks ← SplitByComma(line)

IF Count(toks) < 2 THEN

PRINT "Warning: missing number/title → skipped"

CONTINUE

END IF

c ← NEW Course

c.number ← Trim(toks[0])

c.title ← Trim(toks[1])

c.prereqs ← EMPTY LIST

FOR i FROM 2 TO Count(toks) - 1

t ← Trim(toks[i])

IF t ≠ "" THEN APPEND t TO c.prereqs

END FOR

APPEND c TO courses

APPEND c.number TO courseIds

END FOR

// Pass 2: validate prereqs against courseIds (linear contains)

FOR EACH c IN courses

i ← 0

WHILE i < Count(c.prereqs)

p ← c.prereqs[i]

IF NOT Contains(courseIds, p) THEN

PRINT "Warning: prereq '" + p + "' not found for " + c.number + " → removed"

REMOVE c.prereqs[i]

ELSE

i ← i + 1

END IF

END WHILE

END FOR

RETURN TRUE

END FUNCTION

FUNCTION Contains(list : LIST of STRING, key : STRING) RETURNS BOOL

FOR EACH x IN list

IF x = key THEN RETURN TRUE

END FOR

RETURN FALSE

END FUNCTION

PROCEDURE PrintAllCoursesSorted()

temp ← COPY(courses)

SORT temp BY course.number ASCENDING // alphanumeric

FOR EACH c IN temp

PRINT c.number + ": " + c.title

END FOR

END PROCEDURE

PROCEDURE PrintSingleCourse(target : STRING)

FOR EACH c IN courses

IF c.number = target THEN

PRINT c.number + ": " + c.title

IF Count(c.prereqs) = 0 THEN

PRINT "Prerequisites: none"

ELSE

PRINT "Prerequisites: " + Join(c.prereqs, ", ")

END IF

RETURN

END IF

END FOR

PRINT "Course " + target + " not found"

END PROCEDURE

DATA STRUCTURES

Course:

number : STRING

title : STRING

prereqs : LIST of STRING

courses : HASH\_TABLE<STRING, Course> // key = course number

FUNCTION LoadCourses(filePath : STRING) RETURNS BOOL

IF file cannot be opened THEN

PRINT "Error: cannot open file"

RETURN FALSE

END IF

CLEAR(courses)

// Pass 1: collect all IDs

allIds : SET<STRING> ← EMPTY

FOR EACH line IN file

toks ← SplitByComma(line)

IF Count(toks) ≥ 2 THEN ADD Trim(toks[0]) TO allIds

END FOR

REWIND(file)

// Pass 2: build objects, validate prereqs, insert

FOR EACH line IN file

toks ← SplitByComma(line)

IF Count(toks) < 2 THEN CONTINUE

c : Course

c.number ← Trim(toks[0])

c.title ← Trim(toks[1])

c.prereqs ← EMPTY LIST

FOR i FROM 2 TO Count(toks) - 1

p ← Trim(toks[i])

IF p ≠ "" AND p IN allIds THEN

APPEND p TO c.prereqs

ELSE IF p ≠ "" THEN

PRINT "Warning: prereq '" + p + "' not found for " + c.number + " → skipped"

END IF

END FOR

courses.Insert(c.number, c)

END FOR

RETURN TRUE

END FUNCTION

PROCEDURE PrintAllCoursesSorted()

keys : LIST<STRING> ← courses.Keys()

SORT keys ASCENDING (alphanumeric)

FOR EACH k IN keys

c ← courses.Get(k)

PRINT c.number + ": " + c.title

END FOR

END PROCEDURE

PROCEDURE PrintSingleCourse(target : STRING)

IF NOT courses.Contains(target) THEN

PRINT "Course " + target + " not found"

RETURN

END IF

c ← courses.Get(target)

PRINT c.number + ": " + c.title

IF Count(c.prereqs) = 0 THEN

PRINT "Prerequisites: none"

ELSE

PRINT "Prerequisites: " + Join(c.prereqs, ", ")

END IF

END PROCEDURE

DATA STRUCTURES

Course:

number : STRING

title : STRING

prereqs : LIST of STRING

Node:

course : Course

left : Node

right : Node

CLASS BinarySearchTree

PRIVATE root : Node

PROCEDURE Insert(course)

IF root = null THEN

root ← NEW Node(course, null, null)

ELSE

AddNode(root, course)

END IF

END PROCEDURE

PROCEDURE AddNode(node, course)

IF course.number < node.course.number THEN

IF node.left = null THEN node.left ← NEW Node(course, null, null)

ELSE AddNode(node.left, course)

ELSE IF course.number > node.course.number THEN

IF node.right = null THEN node.right ← NEW Node(course, null, null)

ELSE AddNode(node.right, course)

ELSE

// duplicate → ignore or replace (ignore here)

END IF

END PROCEDURE

FUNCTION Search(number : STRING) RETURNS Node

cur ← root

WHILE cur ≠ null DO

IF number = cur.course.number THEN RETURN cur

IF number < cur.course.number THEN cur ← cur.left ELSE cur ← cur.right

END WHILE

RETURN null

END FUNCTION

PROCEDURE InOrder(visit)

InOrderRec(root, visit)

END PROCEDURE

PROCEDURE InOrderRec(node, visit)

IF node = null THEN RETURN

InOrderRec(node.left, visit)

visit(node.course)

InOrderRec(node.right, visit)

END PROCEDURE

END CLASS

// Catalog + Load (two-pass with validation)

coursesMap : MAP<STRING, Course> // courseNum → Course

tree : BinarySearchTree

FUNCTION LoadCourses(filePath : STRING) RETURNS BOOL

IF file cannot be opened THEN

PRINT "Error: cannot open file"

RETURN FALSE

END IF

CLEAR(coursesMap)

tree ← NEW BinarySearchTree

// Pass 1: create Course records and collect raw prereqs

rawPrereqs : MAP<STRING, LIST<STRING>> ← EMPTY

FOR EACH line IN file

toks ← SplitByComma(line)

IF Count(toks) < 2 THEN

PRINT "Format error: need number and title"

CONTINUE

END IF

num ← Trim(toks[0]); ttl ← Trim(toks[1])

IF num IN coursesMap THEN

PRINT "Duplicate course ignored: " + num

CONTINUE

END IF

c : Course

c.number ← num

c.title ← ttl

c.prereqs ← EMPTY LIST

coursesMap[num] ← c

rawPrereqs[num] ← EMPTY LIST

FOR i FROM 2 TO Count(toks) - 1

p ← Trim(toks[i])

IF p ≠ "" THEN APPEND p TO rawPrereqs[num]

END FOR

END FOR

// Pass 2: validate prerequisites and populate Course.prereqs

FOR EACH (num, list) IN rawPrereqs

FOR EACH pid IN list

IF pid IN coursesMap THEN

APPEND coursesMap[num].prereqs, pid

ELSE

PRINT "Missing prereq '" + pid + "' for " + num + " → skipped"

END IF

END FOR

END FOR

// Build BST from catalog

FOR EACH c IN Values(coursesMap) DO

tree.Insert(c)

END FOR

RETURN TRUE

END FUNCTION

PROCEDURE PrintAllCoursesSorted()

visit(course):

PRINT course.number + ": " + course.title

END visit

tree.InOrder(visit)

END PROCEDURE

PROCEDURE PrintSingleCourse(target : STRING)

node ← tree.Search(target)

IF node = null THEN

PRINT "Course " + target + " not found"

RETURN

END IF

c ← node.course

PRINT c.number + ": " + c.title

IF Count(c.prereqs) = 0 THEN

PRINT "Prerequisites: none"

ELSE

PRINT "Prerequisites: " + Join(c.prereqs, ", ")

END IF

END PROCEDURE

**Runtime Analysis Chart:**

|  |  |  |  |
| --- | --- | --- | --- |
| ***Operation*** | ***Vector*** | ***Hash Table*** | ***Binary Search Tree*** |
| Load file and create objects | Reads *n* lines and validates each prereq linearly through a list -> **O(n^2) worst case.** | Reads *n* lines and uses **O(1)** hash insertion and **O(1)** set validation🡪 O**(N).** | Reads *n* lines, inserts into BST **O(log n)** each 🡪 **O(n log n)**. |
| Search single course | Linear vector scan. 🡪 **O(n)**. | Direct key lookup 🡪 **O(1)** average, **O(n)** worst. | Tree search depth Log n 🡪 **O(log n)**. |
| Print Sorted courses | Manuel vector sort 🡪 **O(n log n)** + **O(n)** print. | Must sort and extract keys 🡪 **O(n log n)** + **O(n)** print. | In order sorted traversal 🡪 **O(n)**. |
| Memory usage | Continuous array storage. Possible duplicates during sort 🡪 **O(n)** space. | Hash table buckets + key overhead 🡪 **O(n)** expected, slightly higher constant factor. | Each node has pointers, left and right, 🡪 **O(n)** but heavier per node overhead. |

**Data structures comparison chart:**

|  |  |  |
| --- | --- | --- |
| ***Data structure*** | ***Advantages*** | ***Disadvantages*** |
| Vector | Simple to implement and traverse. Cache memory efficient. Good for ordered processing. | Slow linear search **O(n)**. Needs resorted in output. Slow insert and delete for large *n*. |
| Hash table | Fast insert/ search average **O (1)**. Ideal for lookup via course ID. | Unordered by default thus requires an extra search for sorting. Extra collisions downgrade to **O(n)**. Potential for higher memory usage. |
| Binary search tree | Maintains sorted order by default. Search and insert average **O(log n)**. Easy in order printing. | Unbalanced tree can downgrade to **O(n)**. Complex implementation. Use of pointers increases memory usage. |

**Data structure recommendation:**

After carefully reviewing each data structure, I would recommend a hash table. Hash tables allow for simple data loading and sorted output, which is ideal for ordered course printing. It also provides constant time lookup for course information, which directly benefits the need to find individual courses and prerequisites quickly. Although printing requires an extra O(n log n) key sort, that only occurs when the user selects option 2. During runtime, the memory performance (O(n) load, O(1) lookup) outperforms the vector’s O(n^2) and more predictable than the BST’s (log n) only when the tree is balanced. Therefore, I can confidently recommend a hash table implementation as the core data structure for ABCU’s program.