SECTION 0 – Universal Course Definition

STRUCT Course

STRING number // “CSCI200”

STRING title // “Data Structures”

LIST prereqs // may be empty

END STRUCT

ENUM DataStructureChoice = { VECTOR , HASH , BST }

GLOBAL DataStructureChoice DS // chosen once in main( )

// Singleton containers – only ONE is populated at a time

GLOBAL VECTOR gVector

GLOBAL HASH\_TABLE<STRING,Course> gHash

GLOBAL TreeNode\* gRoot ← NULL // BST root

SET idSet // used while loading for duplicate / prereq checks

LIST errorLog

SECTION 1 – Common Loader (works with ANY container)

PROCEDURE loadCourses( fileName )

// 1. Reset everything

idSet.CLEAR() ; errorLog.CLEAR()

SWITCH DS

CASE VECTOR gVector.CLEAR()

CASE HASH gHash.CLEAR()

CASE BST gRoot ← NULL

END SWITCH

Collapse

// 2. Open file

OPEN fileName FOR READ AS inFile

IF inFile NOT\_OPENED

PRINT "ERROR: cannot open " + fileName

RETURN

ENDIF

// 3. FIRST PASS – read, parse, RULE-A (≥2 tokens) + duplicates

line ← 0

WHILE NOT EOF( inFile )

line ← line + 1

READ raw FROM inFile

raw ← TRIM( raw )

IF raw = "" CONTINUE

tok ← SPLIT( raw , ',' )

FOR i ← 0 TO tok.SIZE-1 tok[i] ← TRIM( tok[i] )

IF tok.SIZE < 2

errorLog.ADD( "Line " + line + ": fewer than 2 fields" )

CONTINUE

ENDIF

id ← tok[0] ; title ← tok[1]

IF idSet.CONTAINS( id )

errorLog.ADD( "Line " + line + ": duplicate " + id )

CONTINUE

ENDIF

idSet.ADD( id )

prereq ← EMPTY\_LIST

FOR j ← 2 TO tok.SIZE-1

IF tok[j] ≠ "" prereq.ADD( tok[j] )

ENDFOR

c ← NEW Course

c.number ← id ; c.title ← title ; c.prereqs ← prereq

// store in the selected container

SWITCH DS

CASE VECTOR gVector.ADD( c )

CASE HASH gHash.PUT( id , c )

CASE BST gRoot ← bstInsert( gRoot , c )

END SWITCH

ENDWHILE

CLOSE inFile

// 4. SECOND PASS – RULE-B every prereq must exist

PROCEDURE checkP( c )

FOR EACH p IN c.prereqs

IF NOT idSet.CONTAINS( p )

errorLog.ADD( "Course " + c.number +

" lists unknown prereq " + p )

ENDIF

ENDFOR

END PROCEDURE

SWITCH DS

CASE VECTOR FOR EACH c IN gVector checkP( c )

CASE HASH FOR EACH c IN gHash.VALUES() checkP( c )

CASE BST bstInOrder( gRoot , checkP ) // see §3

END SWITCH

// 5. Report errors (if any) and abort the load

IF NOT errorLog.IS\_EMPTY()

PRINT "FILE FORMAT ERRORS:"

FOR EACH e IN errorLog PRINT " " + e

// wipe container so program remains in a safe state

SWITCH DS

CASE VECTOR gVector.CLEAR()

CASE HASH gHash.CLEAR()

CASE BST gRoot ← NULL

END SWITCH

RETURN

ENDIF

PRINT "Loaded " + idSet.SIZE() + " courses successfully."

END PROCEDURE

SECTION 2 – Vector Helper Routines

PROCEDURE vecPrintOne( id )

FOR EACH c IN gVector

IF c.number = id

printBlock( c ) ; RETURN

ENDFOR

PRINT "Course " + id + " not found."

END PROCEDURE

PROCEDURE vecPrintAll()

SORT gVector BY Course.number ASCENDING // O(n log n)

FOR EACH c IN gVector printLine( c )

END PROCEDURE

SECTION 3 – BST Helper Routines

STRUCT TreeNode

Course data

TreeNode\* left

TreeNode\* right

END STRUCT

FUNCTION bstInsert( node , c ) RETURNS TreeNode\*

IF node = NULL

node ← NEW TreeNode

node.data ← c ; node.left ← node.right ← NULL

RETURN node

ENDIF

IF c.number < node.data.number

node.left ← bstInsert( node.left , c )

ELSE

node.right ← bstInsert( node.right , c )

ENDIF

RETURN node

END FUNCTION

FUNCTION bstSearch( node , id ) RETURNS Course | NULL

IF node = NULL RETURN NULL

IF id = node.data.number RETURN node.data

IF id < node.data.number RETURN bstSearch( node.left , id )

ELSE RETURN bstSearch( node.right, id )

END FUNCTION

PROCEDURE bstInOrder( node , VISIT )

IF node = NULL RETURN

bstInOrder( node.left , VISIT )

VISIT( node.data )

bstInOrder( node.right, VISIT )

END PROCEDURE

PROCEDURE bstPrintOne( id )

result ← bstSearch( gRoot , id )

IF result = NULL PRINT "Course " + id + " not found."

ELSE printBlock( result )

END PROCEDURE

PROCEDURE bstPrintAll()

bstInOrder( gRoot , printLine ) // already sorted O(n)

END PROCEDURE

SECTION 4 – Hash-Table Helper Routines

PROCEDURE hashPrintOne( id )

IF gHash.CONTAINS\_KEY( id )

printBlock( gHash.GET( id ) )

ELSE

PRINT "Course " + id + " not found."

ENDIF

END PROCEDURE

PROCEDURE hashPrintAll()

tmp ← gHash.VALUES() // O(n)

SORT tmp BY Course.number ASCENDING // O(n log n)

FOR EACH c IN tmp printLine( c )

END PROCEDURE

SECTION 5 – Universal Print Utilities

PROCEDURE printBlock( c ) // full multi-line version

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

IF c.prereqs.IS\_EMPTY()

PRINT " Prerequisites: none"

ELSE

PRINT " Prerequisites:"

FOR EACH p IN c.prereqs PRINT " " + p

ENDIF

END PROCEDURE

PROCEDURE printLine( c ) // single line (for sorted list)

PRINT c.number + " | " + c.title

END PROCEDURE

SECTION 6 – Menu (works for ANY structure)

PROCEDURE main()

PRINT "Select container 1-Vector 2-Hash 3-BST : "

READ pick

IF pick = 1 DS ← VECTOR

IF pick = 2 DS ← HASH

IF pick = 3 DS ← BST

Collapse

REPEAT

PRINT "\nMenu"

PRINT " 1 – Load data file"

PRINT " 2 – Print entire course list (A→Z)"

PRINT " 3 – Print ONE course & prerequisites"

PRINT " 9 – Exit"

READ choice

SWITCH choice

CASE 1

PRINT "Enter file name: "

READ f

loadCourses( f )

CASE 2

SWITCH DS

CASE VECTOR vecPrintAll()

CASE HASH hashPrintAll()

CASE BST bstPrintAll()

END SWITCH

CASE 3

PRINT "Enter course number: "

READ cid

SWITCH DS

CASE VECTOR vecPrintOne( cid )

CASE HASH hashPrintOne( cid )

CASE BST bstPrintOne( cid )

END SWITCH

CASE 9 BREAK

DEFAULT PRINT "Invalid menu choice."

END SWITCH

UNTIL FALSE

PRINT "Good-bye!"

END PROCEDURE

SECTION 7 – Big-O & Memory Comparison

Let n = #courses, m = total prerequisites.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Load File | Print ONE | Print ALL | Extra Space |
| Vector | O(n + m) | O(n) | O(n log n) | O(n + m) |
| Hash Table | O(n + m) | O(1) avg | O(n log n) | O(n + m) |
| BST (avg) | O(n + m) | O(log n) | O(n) | O(n + m) |
| BST (worst unbal.) | O(n + m) | O(n) | O(n) | O(n + m) |

//Notes

• Hash and vector both need an O(n log n) sort before printing the list; BST’s in-order traversal is already sorted.

• All three hold every Course object exactly once → Θ(n + m) extra memory.

• Hash has fastest single-lookup; BST has fastest full-list print; vector is simplest.

**C. Advantages / Disadvantages**

Vector

* Simplest to code and debug.
* Data kept in contiguous memory → cache-friendly.  
  – printOne is linear; slows down as n grows.  
  – Needs Θ(n log n) sort every time “printAll” is invoked.

Hash Table

* Average Θ(1) lookup — best for many individual course queries.
* Insertion during file load also Θ(1).  
  – Unordered by nature; must copy & sort to print alphabetically (Θ(n log n)).  
  – Slightly higher constant memory overhead for buckets.

Binary-Search Tree

* In-order traversal already sorted → printAll in Θ(n) with no extra memory.
* Balanced tree gives Θ(log n) searches (still fast).  
  – If not self-balancing, the worst case degenerates to linear.  
  – Pointer-heavy; more bookkeeping code than vector/hash.

SECTION 8 – Recommendation

• If the advising tool will be used mostly to list the entire curriculum (Option 2) then

use the Binary Search Tree – O(n) list printing with no extra sort.

• If the tool will be used mostly for many random look-ups of individual courses (Option 3)

choose the Hash Table – O(1) average search.

• The vector version is easy to implement but scales worst for frequent look-ups (O(n) each)

and still needs a full O(n log n) sort to list the courses.

For the combined workload described in the scenario (one full list + many individual queries)

the Hash Table offers the lowest total run time:

– Same O(n log n) cost to create the sorted list (one extra sorting pass).

– Significantly faster O(1) individual queries thereafter.

Therefore the recommended data structure for the final coded program is: HASH TABLE.