Nicholas Deniz

Southern New Hampshire University

CS-300 – Project One

Bryant Moscon

October 11, 2025

1. Vector:

//Vector Data Structure

BEGIN

//Containers

SET catalog to empty VECTOR of course //Data Structure

SET allCoureNumbers to empty SET of text //validate prereqs

SET parsedRecords to empty VECTOR of (num, name, and prereqs)

SET errors to empty VECTOR of text

//File input and per line checks

DISPLAY “Enter path to course CSV files:”

GET filePath

IF filePath does not exist THEN

DISPLAY “Error: file not found”

STOP

END IF

OPEN filePath for reading

WHILE file has more lines

GET line

SET line to TRIM

IF line = “” THEN

CONTINUE //ignores blank lines

END IF

SET tokens to SPLIT line by “,”

//trim around token

FOR EACH I from 0 to LENGTH(tokens) – 1

SET tokens[i] to TRIM(tokens[i])

END FOR

//2 items on each line, course number and name

IF LENGTH(tokens) < 2 THEN

ADD “Format error (needs 2 or more items): “ + line TO erros

CONTINUE

END IF

SET courseNum to tokens[0]

SET courseName to tokens[1]

IF courseNum = “” OT courseName = “” THEN

ADD “Format error (empty number or name): “ + line TO errors

CONTINUE

END IF

//get 0 or more prereqs

SET prereqList to empty VECTOR of text

IF LENGTH(tokens) > 2 THEN

FOR EACH j from 2 to LENGTH(tokens) – 1

IF tokens[j] not equal to “” THEN

ADD tokens[j] TO prereqList

END IF

END FOR

END IF

//remember course numbers to validate prereqs

ADD courseNum TO allCourseNumbers

//hold parsed info for after finish reading file

ADD(courseNum, courseName, prereqList) TO parsedRecords

END WHILE

CLOSE filePath

//if perline format error happens stop and display

IF LENGTH(errors) > 0 THEN

DISPLAY “File format errors:”

FOR EACH message IN errors

DISPLAY “ – “ + message

END FOR

STOP

END IF

//prereq validation

FOR EACH rec IN parsedRecords

FOR EACH p IN rec.prereqList

IF p NOT IN allCourseNumbers THEN

ADD “Missing prerequisite definition: “ + p + “ (required by “ + rec.courseNum + “)” TO errors

END IF

END FOR

END FOR

IF LENGTH(errors) > 0 THEN

DISPLAY “Prerequisite check errors:”

FOR EACH message in errors

DISPLAY “ – “ + message

END FOR

STOP

END IF

//Course objects in vector

FOR EACH rec IN parsedRecords

MAKE newCourse

SET newCourse.courseNumber to rec.courseNum

SET newCourse.name to rec.courseName

SET newCourse.prerequisites to rec.prereqList

ADD newCourse TO catalog

END FOR

//search and print course info

DISPLAY “Enter a course number to display or quit:”

GET query

WHILE UPPER(query) not equal to “quit”

SET found to FALSE

//find target course in vector

FOR EACH c IN catalog

IF c.courseNumber = query THEN

//print selected course

DISPLAY ‘Course; “ + c.courseNumber + “ – “ + c.name

IF LENGTH(c.prerequisites) = 0 THEN

DISPLAY “Prerequisites: none”

ELSE

DISPLAY “Prerequisites:”

FOR EACH p IN c.prerequisites

DISPLAY “ – “ + p

END FOR

END IF

//print info for each Prerequisite course

FOR EACH p IN c.prerequisite s

//look up prereq course in vector

FOR EACH d IN catalog

IF d.courseNumber = p THEN

DISPLAY “Prerequisite course details: “ + d.courseNumber + “ – “ + d.name

BREAK

END IF

END FOR

END FOR

SET found to TRUE

BREAK

END IF

END FOR

If found = FALSE THEN

DISPLAY “Course not found: “ + query

ENN IF

DISPLAY “Enter another course number or quit:”

GET query

END WHILE

END

Hash Table:

Define course object:

BEGIN Course

DECLARE courseNumber AS STRING

DECLARE name AS STRING

DECLARE prerequisites AS LIST OF STRING

METHOD setNumber(value)

SET courseNumber = value

END METHOD

METHOD getNumber()

RETURN courseNumber

END METHOD

METHOD setName(value)

SET name = value

END METHOD

METHOD getName()

RETURN name

END METHOD

METHOD addPrerequisite(pr)

APPEND pr TO prerequisites

END METHOD

METHOD getPrerequisites()

RETURN prerequisites

END METHOD

END

Define hash table that stores course by course numbers:

BEGIN HashTableOfCourse

DECLARE buckets AS ARRAY OF LIST Course

DECLARE bucketCount AS INT

METHOD makeWithSize(n)

SET bucketCount = n

CREATE buckets WITH n EMPTY LISTS

END METHOD

METHOD hash(key)

DECLARE sum AS INT = 0

FOR EACH ch IN key

SET sum = sum + ASCII\_VALUE\_OF(ch)

END FOR

RETURN sum MOD bucketCount

END METHOD

METHOD put(course)

SET key = course.getNumber()

SET index = hash(key)

FOR EACH item IN buckets[index]

IF item.getNumber() == key

REPLACE item WITH course

RETUN

END IF

END FOR

APPEND course TO buckets[index]

END METHOD

METHOD get(key)

SET index = hash(key)

FOR EACH item IN buckets[index]

IF item.getNumber() == key

RETURN item

END IF

END FOR

RETURN NULL

END METHOD

METHOD contains(key)

IF get(key) IS NULL

RETURN FALSE

ELSE

RETURN TRUE

END IF

END METHOD

METHOD forEachCourse(doSomething) //helps with loops over courses

FOR I FROM 0 TO bucketCount – 1

FOR EACH item IN buckets[i]

CALL doSomething(item)

END FOR

END FOR

END METHOD

END

Load data from file to create Course objects to store I hash:

FUNCTION loadCoursesIntoHashTable(filePath) RETURNS HashTableOfCourse

CREATE table AS HashTableOfCourse

CALL table.makeWithSize(101) //for any prime number

TRY TO OPEN filePath FOR READING AS file

IF file FAILS TO OPEN

DISPLAY “Error cannot open file”

RETURN table

END IF

FOR EACH line IN file

TRIM whitespace FROM line

IF line IS EMPTY

CONTINUE

END IF

SPLIT line BY “,” INTO parts

//format check

IF LENGTH(parts) < 2

Display “Error bad line, needs number and name: “ + line

CONTINUE

END IF

SET number = TRIM(parts[0])

SET name = TRIM(parts[1])

CREATE c AS Course

CALL c.setNumber(number)

CALL c.setName(nmame)

//remaining tokens are prerequisites

FOR I FROM 2 TO LENGTH(parts) – 1

SET prereq = TRIM(parts[i])

IF prereq IS NOT EMPTY

CALL c.addPrerequisite(prereq)

END IF

END FOR

CALL table.put(c)

END FOR

CLOSE FILE

//verify all list prerequisites as courses in file

CALL table.forEachCourse(CHECK\_PREREQS)

RETURN table

//helper for each course during validation

LABEL CHECK\_PREREQS(course)

FOR EACH pr IN course.getPrerequisites

IF NOT table.contains(pr)

DISPLAY “Error missing prerequisite “ + pr + “ for “ + course.getNumber()

END IF

END FOR

END LABEL

END FUNCTION

Print info for one course by course number:

FUNCTION searchCourse(table, inputCourseNumber)

SET course = table.get(inputCourseNumber)

IF course IS NULL

DISPLAY “Course “ + inputCourseNumber + “ not found”

RETURN

END IF

//print course

DISPLAY course.getNumber() + “, “ + course.getName()

//print prerequisites

SET prereqs = course.getPrerequisites()

IF prereqs IS EMPTY

DISPLAY “Prerequisites: none”

RETURN

END IF

DISPLAY “Prerequisites:”

FOR EACH prNum IN prereqs

SET prCourse = table.get(prNum)

IF prCourse IS NULL

DISPLAY prNum + “ (missing from file)”

ELSE

DISPLAY prCourse.getNumber() + “, “ + prCourse.getName()

END IF

END FOR

END FUNCTION

BST:

Course object:

STRUCT Course

STRING courseNumber

STRING courseName

LIST<STRING> prerequisites //0 or more

END STRUCT

BST node:

STRUCT Node

Course data

Node\* left

Node\* right

END STRUCT

BST Wrapper:

STRUCT CourseBST

Node\* root

END STRCUT

createNode:

FUNCTION createNode(course : Course) RETURNS Node\*

node <- NEW Node

node.data <- course

node.left <- null

node.right <- null

RETURN node

END FUNCTION

File Input, parsing, and validation:

PROCEDURE loadCoursesIntoBST(filePath : STRING, tree : CourseBST)

FILE f

LIST<RAWLINE> rawLines //store raw strings each line

LIST<Record> pending //temp records with parsing

SET<STRING> allCourseIds //for checks

STRUCT Record

STRING courseNumber

STRING courseName

LIST<STRING> prereqIds

END STRUCT

f <- OPEN filePath FOR READ

IF f fails to open THEN

DISPLAY “ERROR: Unable to open file.”

RETURN

ENDIF

WHILE f has another line DO //read and parse

line <- READLINE(f)

IF line is empty after trim THEN //skip whitespace/blank lines

CONTINUE

ENDIF

Tokens <- SPLIT line BY “,” //to keep raw order

TRIM whitespace on each token

IF LENGTH(tokens) < 2 THEN //check format

DISPLAY “ERROR: missing courseNumber or courseName -> “ + line

CONTINUE

ENDIF

rec <- NEW Record

rec.courseNumber <- tokens[0]

rec.courseName <- tokens[1]

//get remaining tokens as prereqs

rec.prereqIds <- EMPTY LIST

FOR I FROM 2 TO LENGTH(tokens)-1 DO

IF tokens[i] IS NOT empty THEN

APPEND tokens[i] TO rec.prereqIds

ENDIF

END FOR

APPEND rec TO pending

INSERT rec.courseNumber INTO allCourseIds

END WHILE

CLOSE f

//validate and insert in BST

FOR EACH rec IN pending DO

validPreres <- EMPTY LIST

FOR EACH pid IN rec.prereqIds DO

IF pid SET ALL IN allCourseIds THEN

APPEND pid TO validPrereqs

ELSE

DISPLAY “ERROR: Format issue with prerequisite “ + pid + “ is not listed as a course in the file for “ + rec.courseNumber

ENDIF //drop the messed up prerequisite

END FOR

//Build Course object with the prereqs that were validated

c <- Course

c.courseNumber <- req.courseNumber

c.courseName <- req.courseName

prerequisites <- validPrereqs

//Insert into BST by courseNumber

bstInsert(tree, c)

END FOR

END PROCEDURE

BST Operations:

Insert, by courseNumber:

PROCEDURE bstInsert(tree : CourseBST, c : Course)

IF tree.root IS null THEN

tree.root <- createNode(c)

RETURN

ENDIF

par <- null

cur <- tree.root

WHILE cur IS NOT null DO

par <- cur

IF c.courseNumber < cur.data.courseNumber THEN

cur <- cur.left

ELSE

cur <- cur.right

ENDIF

ENDWHILE

//new node below parent

newNode <- createNode(c)

IF c.courseNumber < par.data.courseNumber THEN

par.left <- newNode

ELSE

par.right <- newNode

ENDIF

END PROCEDURE

Search:

FUNCTION bstSearch(tree : CourseBST, targetNumber : STRING) RETURNS Node\*

cur <- tree.root

WHILE cur IS NOT null DO

IF targetNumber = cur.data.courseNumber THEN

RETURN cur

ELSE IF targetNumber < cur.data.courseNumber THEN

cur <- cur.left

ELSE

cur <- cur.right

ENDIF

END WHILE

RETURN NULL

END FUNCTION

In-order traversal in to print all courses sorted by the courseNumber:

PROCEDURE bstInOrderPrint(node : Node\*)

IF node IS null THEN RETURN

bstInOrderPrint(node.left)

DISPLAY node.data.courseNumber + “, “ + node.data.courseName

bstInOrderPrint(node.right)

END PROCEDURE

Print a single course and their prerequisites:

PROCEDURE printCourseInfo(tree : CourseBST. targetNumber : STRING)

found <- bstSearch(tree, targetNumber)

IF found IS null THEN

DISPLAY “Course “ + targetNumber + “ not found.”

RETURN

ENDIF

DISPLAY found.data.courseNumber + “, “ + found.data.courseName

IF LENGTH(found.data.prerequisites) = 0 THEN

DISPLAY “Prerequisites: none”

ELSE

line <- “Prerequisites: “

FOR I FROM 0 TO LENGTH(found.data.prerequisites)-1 DO

line <- line + found.data.prerequisites[i]

IF I < LENGTH(found.data.prerequisites)-1 THEN

Line <- line + “< “

ENDIF

END FOR

DISPLAY line

ENDIF

END PROCEDURE

Print all courses in alphanumeric order by the courseNumber:

PROCEDURE preintAllCourse(ree : CourseBST)

IF tree.root IS null THEN

DISPLAY “No course loaded”

RETURN

ENDIF

bestInOrderPrint(tree.root)

END PROCEDURE

1. Menu Pseudocode:

BEGIN Menu

DISPLAY “1. Load file 2. Print all courses 3. Print one course 9. Exit”

GET choice

WHILE choice != 9

IF choice = 1 THEN

DISPLAY “Enter path: “; GET path

bstRoot = LoadCoursesIntoBST(path)

// Get only one structure to actually use, recommending BST

DISPLAY “Completed Load”

ELSE IF choice = 2 THEN

IF bstRoot = NULL THE DISPLAY “No data loaded”

ELSE PrintAllCoursesBST(bstRoot)

// Call “print all sorted” for the specific structure, the helpers are set up for each structure

ELSE IF choice = 3 THEN

DISPLAY “Enter course number: “; GET goal

IF bstRoot = NULL THEN DISPLAY “No data loaded”

ELSE PrintOneCourseBST(bstRoot, goal)

// Call existing “print one with prerequisites” with the recommended structure (BST)

ELSE

DISPLAY “Invalid input”

END IF

DISPLAY “Please select 1. Load file, 2. Print all courses, 3. Print one course, or 9. Exit”

GET choice

END WHILE

END

1. Print all courses is alphanumeric order:

BST:

BEGIN PrintAllCourse\_BSTRoot

If root = NULL THEN

DISPLAY “No loaded data”

RETURN

END IF

//In order traversal would print courses in alphanumeric by courseNumber

CALL VistsInOrder(root)

END

BEGIN VistInOrder(node)

IF node = null THEN RETURN

CALL VisitInOrder(node.left)

DISPLAY node.data.courseNumber + “, “ + node.data.name

CALL VisitInOrder(node.right)

END

Vector:

BEGIN PrintAllCourses\_Vector(catalog)

SET temp = Copy of catalog

SORT temp BY courseNumber ASC

FOR EACH c IN temp

DISPLAY c.courseNumber + “, ” + c.name

END FOR

END

Hash Table:

BEGIN PrintAllCourse\_HashTable

CREATE list = empty list

CALL table.forEachCourse(add to list)

SORT list BY courseNumber ASC

FOR EACH c IN list

DISPLAY c.getNumber() + “, “

1. Runtime and Memory Evaluation (Worst Case)

Cost per line = 1

Function calls cost the time of function

n = number of course in the file

What is analyzed is the loading and building of the structures. For example, the file opening, parse, insert, and creating course objects. The menu and search/print won’t be counted in this analyzation.

Vector:

(Code – Line Cost – Number of Time – Total (Key))

Read/Trim/Split each of the input lines – 1 – n – n

Create Course and push back – 1 – n – n

Worst case total runtime is O(n) and memory is O(n)

Hash Table:

(Code – Line Cost – Number of Time – Total (Key))

Read/Trim/Split each of the input lines – 1 – n – n

put() (hash + scan + bucket + insert) – up to current size – each n insert – 1+2+3+…+n = n(n+1)/2

Worst case total runtime is O(n^2) and memory is O(n+b) which is items + bucket

BST:

(Code – Line Cost – Number of Time – Total (Key))

Read/Trim/Split each of the input lines – 1 – n – n

insert() (following path to leaf) – up to current height – each n insert – 1+2+3+…+n = n(n+1)/2

Worst case total runtime is O(n^2) and memory is O(n)

1. Advantages and Disadvantages of the three structures. Based upon the two advisor tasks, “Print a list of all the computer science courses in alphanumeric order.” and “For a given course, print out its title and prerequisites.”.

Vector:

Advantages:

- Vector is fast and straight to the point at a O(n), it also uses minimal memory

- It can produce a sorted list by calling sort once at O(n log n) and then would display it at O(n)

Disadvantages:

* Print one course is O(n) with a linear search unless you use a sorted order
* If a sorted list is needed a lot, then resort each time or keep a sorted copy

Hash Table:

Advantages:

* Print one course is O(1) on average. With a key lookup which is good for constant single course queries

Disadvantages:

* There isn’t a natural order. Printing all in alphanumeric order requires gathering, sorting, and then printing which is at O(n log n) + O(n)
* If there is poor hashing it would decrease to O(n)

BST:

Advantages:

* Print all sorted is practically built into it through in order traversal at O(n) so no extra sort is needed
* Print one course is O(log n) on average if the tree is balanced

Disadvantages:

* At the worst case, when inserting sorted keys, the height becomes n and it would decrease to O(n)
* It is more complicated than vector, especially with pointers

Recommendation:

I am recommending the structure Binary Search Tree (BST) to be used in Project Two. It fits best with the two advisors’ requirements. The sorted list, task 1, is basically already set up and doesn’t require much work to set up. The BST’s in order traversal prints courses in alphanumeric order at O(n). Vector and Hash would both need O(n log n) to sort each time an ordered list is needed. Single course lookup, task 2, scales well with BST. The BST lookup is O(log n) on average, this is efficient going by the advisor’s needs with their courses and prerequisites. Vector is O(n) which is not the best and Hash is O(1) but is more costly to implement to gather and sort. BST handles all the required menu options well. Load, print all, and print one are handled efficiently by BST. The worst case for BST insert would be O(n) but loading only needs to be completed one time. Vector should be used if the program only needs to print a list once or not constantly. While its simpleness and O(n) are efficient, O(n log n) is the time when sorting. Hash Table should be used if the program required many single course lookups and very rarely a print all sorted. You do get fast lookups, but it would be costly when a ordered list is needed. BST is the most balanced option out of these three structures.