Ryan Blackburn  
Southern New Hampshire University  
CS-300 Project 1  
Oct. 7th 2025

**Pseudocode**

1. **Resubmit pseudocode from previous pseudocode assignments.**
   1. Design pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.

//Vector - Milestone 1

FUNCTION LoadCourses\_Vector(filePath)

OPEN file at filePath

IF file cannot be opened THEN

DISPLAY "Error: file not found"

RETURN

ENDIF

CREATE empty LIST rawLines

CREATE empty LIST courseIds

WHILE there are lines to read

line = READ LINE

tokens = SPLIT line BY comma

TRIM whitespace from each token

IF LENGTH(tokens) < 2 THEN

DISPLAY "Error: line missing course number or title"

CONTINUE

ENDIF

ADD tokens TO rawLines

ADD tokens[0] TO courseIds

ENDWHILE

CLOSE file

FOR EACH tokens IN rawLines

FOR i FROM 2 TO LENGTH(tokens) - 1

prereq = tokens[i]

IF prereq NOT IN courseIds THEN

DISPLAY "Error: prerequisite " + prereq + " not found"

ENDIF

ENDFOR

ENDFOR

RETURN rawLines

END FUNCTION

//Hash- Milestone 1

FUNCTION LoadCourses\_Hash(filePath, table)

OPEN file at filePath

IF file cannot be opened THEN

DISPLAY "Error: file not found"

RETURN

ENDIF

CREATE empty LIST rawLines

CREATE empty SET courseIds

WHILE there are lines to read FROM file

line = READ LINE

tokens = SPLIT line BY comma

TRIM whitespace from each token

IF LENGTH(tokens) < 2 THEN

DISPLAY "Error: line missing course number or title"

CONTINUE

ENDIF

ADD tokens TO rawLines

ADD tokens[0] TO courseIds

ENDWHILE

CLOSE file

FOR EACH tokens IN rawLines

FOR i FROM 2 TO LENGTH(tokens) - 1

IF tokens[i] NOT IN courseIds THEN

DISPLAY "Error: prerequisite " + tokens[i] + " not found"

ENDIF

ENDFOR

ENDFOR

RETURN rawLines

END FUNCTION

//Binary Search Tree- Milestone 1

FUNCTION LoadCourses\_BST(filePath)

OPEN file at filePath

IF file cannot be opened THEN

DISPLAY "Error: file not found"

RETURN

ENDIF

CREATE empty LIST rawLines

CREATE empty LIST courseIds

WHILE there are lines to read

line = READ LINE

tokens = SPLIT line BY comma

TRIM whitespace from each token

IF LENGTH(tokens) < 2 THEN

DISPLAY "Error: line missing course number or title"

CONTINUE

ENDIF

ADD tokens TO rawLines

ADD tokens[0] TO courseIds

ENDWHILE

CLOSE file

FOR EACH tokens IN rawLines

FOR i FROM 2 TO LENGTH(tokens) - 1

prereq = tokens[i]

IF prereq NOT IN courseIds THEN

DISPLAY "Error: prerequisite " + prereq + " not found"

ENDIF

ENDFOR

ENDFOR

RETURN rawLines

END FUNCTION

* 1. Design pseudocode to show how to create course objects so that one course object holds data from a single line from the input file.

//Vector- Milestone 2

STRUCT Course

courseNum

title

prereqs

END STRUCT

FUNCTION BuildCourses\_Vector(rawLines)

CREATE empty VECTOR courses

FOR EACH tokens IN rawLines

CREATE new Course c

c.courseNum = tokens[0]

c.title = tokens[1]

c.prereqs = EMPTY LIST

i FROM 2 TO LENGTH(tokens) - 1

ADD tokens[i] TO c.prereqs

ENDFOR

ADD c TO courses

ENDFOR

RETURN courses

END FUNCTION

//Hash-- Milestone 2

STRUCT Course

courseNum: STRING

title: STRING

prereqs: LIST OF STRING

END STRUCT

FUNCTION BuildCourses\_Hash(rawLines)

CREATE empty HashTable table

FOR EACH tokens IN rawLines

CREATE new Course c

c.courseNum = tokens[0]

c.title = tokens[1]

c.prereqs = EMPTY LIST

FOR i FROM 2 TO LENGTH(tokens) - 1

ADD tokens[i] TO c.prereqs

ENDFOR

HashTable\_Insert(table, c.courseNum, c)

ENDFOR

RETURN table

END FUNCTION

//Binary Search- Milestone 2

STRUCT Course

courseNum

title

prereqs

END STRUCT

STRUCT Node

course

left

right

END STRUCT

CLASS BST

root

END CLASS

PROCEDURE AddNode(node, c)

IF c.courseNum < node.course.courseNum THEN

IF node.left IS NULL THEN

node.left = NEW Node(c)

ELSE

AddNode(node.left, c)

ENDIF

ELSE

IF node.right IS NULL THEN

node.right = NEW Node(c)

ELSE

AddNode(node.right, c)

ENDIF

ENDIF

END PROCEDURE

FUNCTION BuildCourses\_BST(rawLines)

CREATE new BST tree

FOR EACH tokens IN rawLines

CREATE new Course c

c.courseNum = tokens[0]

c.title = tokens[1]

c.prereqs = EMPTY LIST

FOR i FROM 2 TO LENGTH(tokens) - 1

ADD tokens[i] TO c.prereqs

ENDFOR

IF tree.root IS NULL THEN

tree.root = NEW Node(c)

ELSE

AddNode(tree.root, c)

ENDIF

ENDFOR

RETURN tree

END FUNCTION

* 1. Design pseudocode that will print out course information and prerequisites.

//Vector- Milestone 3

FUNCTION PrintCourse\_Vector(courses, courseNum)

FOR EACH course IN courses

IF course.courseNum == courseNum THEN

DISPLAY course.courseNum + ": " + course.title

IF course.prereqs IS EMPTY THEN

DISPLAY "Prerequisites: None"

ELSE

DISPLAY "Prerequisites: " + JOIN(course.prereqs, ", ")

ENDIF

RETURN

ENDIF

ENDFOR

DISPLAY "Course not found"

END FUNCTION

//Hash- Milestone 3

FUNCTION PrintCourse\_Hash(table, courseNum)

c = HashTable\_Search(table, courseNum)

IF c IS EMPTY THEN

DISPLAY "Course not found"

RETURN

ENDIF

DISPLAY c.courseNum + ": " + c.title

IF c.prereqs IS EMPTY THEN

DISPLAY "Prerequisites: None"

ELSE

DISPLAY "Prerequisites: " + JOIN(c.prereqs, ", ")

ENDIF

END FUNCTION

//Binary Search- Milestone 3

FUNCTION SearchBST(node, courseNum)

IF node IS NULL THEN

RETURN NULL

ELSE IF courseNum == node.course.courseNum THEN

RETURN node.course

ELSE IF courseNum < node.course.courseNum THEN

RETURN SearchBST(node.left, courseNum)

ELSE

RETURN SearchBST(node.right, courseNum)

ENDIF

END FUNCTION

PROCEDURE PrintCourse\_BST(tree, courseNum)

course = SearchBST(tree.root, courseNum)

IF course IS NULL THEN

DISPLAY "Course not found"

RETURN

ENDIF

DISPLAY course.courseNum + ": " + course.title

IF course.prereqs IS EMPTY THEN

DISPLAY "Prerequisites: None"

ELSE

DISPLAY "Prerequisites: " + JOIN(course.prereqs, ", ")

ENDIF

END PROCEDURE

1. **Create pseudocode for a menu**.
   1. Option 1: Load the file data into the data structure. Note that before you can print the course information or the sorted list of courses, you must load the data into the data structure.

PROCEDURE MainMenu()

dataLoaded = FALSE

dsType = 0

DECLARE coursesVec //Vectors

DECLARE table // Hash

DECLARE tree // Binary search

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

INPUT dsType

REPEAT

DISPLAY "1. Load course file"

DISPLAY "2. Print all courses (alphanumeric)"

DISPLAY "3. Print a specific course"

DISPLAY "9. Exit"

INPUT choice

SWITCH choice

CASE 1:

INPUT filePath

IF dsType == 1 THEN

raw = LoadCourses\_Vector(filePath)

IF raw IS EMPTY THEN

DISPLAY "No data loaded."

ELSE

coursesVec = BuildCourses\_Vector(raw)

dataLoaded = TRUE

DISPLAY "Loaded " + LENGTH(coursesVec) + " courses (Vector)."

ENDIF

ELSE IF dsType == 2 THEN

raw = LoadCourses\_Hash(filePath, /\*table param unused here\*/)

IF raw IS EMPTY THEN

DISPLAY "No data loaded."

ELSE

table = BuildCourses\_Hash(raw)

dataLoaded = TRUE

DISPLAY "Loaded courses into Hash Table."

ENDIF

ELSE IF dsType == 3 THEN

raw = LoadCourses\_BST(filePath)

IF raw IS EMPTY THEN

DISPLAY "No data loaded."

ELSE

tree = BuildCourses\_BST(raw)

dataLoaded = TRUE

DISPLAY "Loaded courses into BST."

ENDIF

ELSE

DISPLAY "Invalid DS selection; restart program."

ENDIF

* 1. Option 2: Print an alphanumerically ordered list of all the courses in the Computer Science department.

CASE 2:

IF NOT dataLoaded THEN

DISPLAY "Please load course data first (Option 1)."

BREAK

ENDIF

IF dsType == 1 THEN

SORT coursesVec BY courseNum ascending

DISPLAY "Vector"

FOR EACH c IN coursesVec

DISPLAY c.courseNum + " | " + c.title

ENDFOR

ELSE IF dsType == 2 THEN

allCourses = EMPTY LIST

FOR EACH bucket IN table

FOR EACH (key, c) IN bucket

APPEND c TO allCourses

ENDFOR

ENDFOR

SORT allCourses BY courseNum ascending

DISPLAY "Hash Table"

FOR EACH c IN allCourses

DISPLAY c.courseNum + " | " + c.title

ENDFOR

ELSE IF dsType == 3 THEN

DISPLAY "Computer Science Courses (BST)"

stack = EMPTY STACK

current = tree.root

WHILE (current != NULL) OR (NOT EMPTY(stack))

WHILE current != NULL

PUSH stack, current

current = current.left

ENDWHILE

current = POP stack

DISPLAY current.course.courseNum + " | " + current.course.title

current = current.right

ENDWHILE

ENDIF

END CASE

* 1. Option 3: Print the course title and the prerequisites for any individual course.

CASE 3:

IF NOT dataLoaded THEN

DISPLAY "Please load data first (Option 1)."

BREAK

ENDIF

INPUT courseNum

IF dsType == 1 THEN

PrintCourse\_Vector(coursesVec, courseNum)

ELSE IF dsType == 2 THEN

PrintCourse\_Hash(table, courseNum)

ELSE IF dsType == 3 THEN

PrintCourse\_BST(tree, courseNum)

ENDIF

* 1. Option 9: Exit the program.

CASE 9:

DISPLAY "Exiting program."

BREAK\_LOOP

DEFAULT:

DISPLAY "Invalid option. Please choose 1, 2, 3, or 9."

END SWITCH

UNTIL choice == 9

END PROCEDURE

1. **Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.**
   1. Sort the course information by alphanumeric course number from lowest to highest.

FUNCTION SortCourses(dsType, coursesVec, table, tree)

IF NOT dataLoaded THEN

DISPLAY "Please load course data first (Option 1)."

RETURN

ENDIF

IF dsType == 1 THEN // VECTOR

SORT coursesVec BY courseNum ascending

DISPLAY "Courses sorted according to Vector."

ELSE IF dsType == 2 THEN // HASH

allCourses = EMPTY LIST

FOR EACH bucket IN table

FOR EACH (key, c) IN bucket

APPEND c TO allCourses

ENDFOR

ENDFOR

SORT allCourses BY courseNum ascending

table = HashTable\_Rebuild(allCourses)

DISPLAY "Courses sorted according to Hash Table."

ELSE IF dsType == 3 THEN // BINARY SEARCH TREE

DISPLAY "Courses in BST are already stored in sorted order by key."

ELSE

DISPLAY "Invalid data structure selection."

ENDIF

END FUNCTION

* 1. Print the sorted list to a display.

FUNCTION PrintSortedCourses(dsType, coursesVec, table, tree)

IF NOT dataLoaded THEN

DISPLAY "Error, select 1 first.”

RETURN

ENDIF

DISPLAY "Courses Sorted.”

IF dsType == 1 THEN // VECTOR

FOR EACH c IN coursesVec

DISPLAY c.courseNum + " | " + c.title

ENDFOR

ELSE IF dsType == 2 THEN // HASH

FOR EACH bucket IN table

FOR EACH (key, c) IN bucket

DISPLAY c.courseNum + " | " + c.title

ENDFOR

ENDFOR

ELSE IF dsType == 3 THEN // BINARY

stack = EMPTY STACK

current = tree.root

WHILE (current != NULL) OR (NOT EMPTY(stack))

WHILE current != NULL

PUSH stack, current

current = current.left

ENDWHILE

current = POP stack

DISPLAY current.course.courseNum + " | " + current.course.title

current = current.right

ENDWHILE

ELSE

DISPLAY "Invalid data structure selection."

ENDIF

END FUNCTION

**Evaluation**

1. **Evaluate the run time and memory of data structures that could be used to address the requirements.**

Vector Data

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open and read file** | 1 | n | n |
| **Parse and validate** | 1 | n | n |
| **Create course object** | 1 | n | n |
| **Add course to Vector** | 1 | n | n |
|  |  |  |  |
| **Total Cost** | | | 4n |
| **Runtime** | | | O(n) |

Hash Table data

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open and read file** | 1 | n | n |
| **Parse and validate** | 1 | n | n |
| **Create course object** | 1 | n | n |
| **Add course to Vector** | 1 | n | n |
|  |  |  |  |
| **Total Cost** | | | 4n |
| **Runtime** | | | O(n) Average |

Binary Search Tree Data

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open and read file** | 1 | n | n |
| **Parse and validate** | 1 | n | n |
| **Create course object** | 1 | n | n |
| **Add course to Vector** | 1 | N^2 | N^2 |
|  |  |  |  |
| **Total Cost** | | | N^2+3n |
| **Runtime** | | | O(n) Worst |

1. **Based on the advisor’s requirements, analyze each of the vector, hash table, and tree data structures. Explain the advantages and disadvantages of each structure in your evaluation.**

Advantages of the vector approach are that it is the simplest to implement, provides fast sequential access and predictable memory use. It works well for small datasets and straightforward traversal but becomes inefficient as data grows. Each course has to be searched linearly with O(n) lookup time and printing the list in alphanumeric order requires an O(n log n) sort. It is clear and dependable, but it lacks scalability and it is not efficient for unordered lookups.

The hash table can be used for continuous O(1) time for inserting and retrieving courses by key, so it would be ideal for quick lookups of individual courses, but for it to print an alphanumeric list all values have to be gathered and sorted, adding O(n log n) time. They also use more memory to handle collisions and maintain bucket arrays. The hash table is good for a targeted lookup tool but not as much for ordered lists.

The binary search tree is a middle grounds search with a balance between order and efficiency. Insertion places each course in sorted position by key, and inorder traversal prints the full course list in O(n) time. Searching and inserting are typically O(log n) but can degrade to O(n²) if the tree becomes unbalanced. The binary search tree is ideally suited for storing ordered data because it meets advisor requirements and has minimal overhead.

1. **Make a recommendation for which data structure you plan to use in your code**.

Of the three structures, my preference would be to use the binary search tree in order to meet the advisors’ needs. This method keeps the data in alphanumeric order, which allows for O(n) printing without sorting, and supports efficient O(log n) average case lookups. This is chosen over Vector because, even though it is easy to implement, it is too slow for repeated searches. It is also chosen over the hash table because the extra steps needed to produce an ordered result create a net loss in efficiency even considering the faster single item lookups. Considering the scale of the project, it is better to evenly optimize both retrieval and ordered display, making the binary search tree fit the programs’ objective of clarity and efficiency.