# **Resubmit pseudocode from previous pseudocode assignments and update as necessary**. In the previous assignments, you created pseudocode for each of the three data structures (vector, hash table, and tree). Be sure to resubmit the following pseudocode for each data structure.

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

METHOD ValidateFile (string filename)

BEGIN

CREATE stringstream currentLine

CREATE stringstream searchLine

CREATE filestream file

CREATE currentFilePosition

CREATE validLine

CREATE lineCount

CREATE compareString

file <- open filename

lineCount <- 0

validLine <- true

WHILE not end of file AND validLine is true

INCREMENT lineCount

currentLine <- READ line from file

currentFilePosition <- current file position

// check for course number presence

IF currentLine status is good

courseNumber <- currentLine // get sub-string delimited by comma

ELSE

validLine <- false

ENDIF

// check for course description presence

IF currentLine status is good AND validLine is true

courseDescription <- currentLine // get sub-string delimited by comma

ELSE

validLine <- false

ENDIF

// check for prerequisites

WHILE currentLine status is good AND validLine is true

coursePrerequisite <- currentLine // get sub-string delimited by comma

validLine <- false

file <- beginning position

WHILE validLine AND NOT eof

searchLine <- READ line from file

IF searchLine status is good

compareString <- searchLine // get sub-string delimited by comma

IF courseNumber equals compareString

validLine <- true

ENDIF

ENDIF

IF searchLine status is good

compareString <- searchLine // get rest of line

ENDWHILE

ENDWHILE

file position <- currentFilePosition

ENDWHILE

CLOSE filename

IF NOT validLine

PRINT “STOPPED: Invalid Line in file at Line: ” + lineCount

ENDIF

END

## **Your pseudocode should show how to create course objects, so that one course object holds data from a single line from the input file.**

### **VECTOR METHOD:**

METHOD LoadFile (parameters: vector<course> courses, string filename)

BEGIN

CREATE filestream file

CREATE stringstream currentLine

file <- open filename

WHILE not end of file

CREATE course

CREATE prerequisiteCourseNumber

currentLine <- READ line from file

course.number <- currentLine a // get sub-string delimited by comma

courseDescription <- currentLine // get sub-string delimited by comma

WHILE currentLine is good

prerequisiteCourseNumber <- currentLine // get sub-string delimited by comma

ADD prerequisiteCourseNumber TO course.prerequisites

ENDWHILE

ADD course TO courses

ENDWHILE

close filename

END

### **HASH TABLE METHOD:**

METHOD LoadFile (parameters: vector<Node> nodes, string filename)

BEGIN

CREATE filestream file

CREATE stringstream currentLine

file <- open filename

WHILE not end of file

CREATE course

CREATE prerequisiteCourseNumber

currentLine <- READ line from file

course.number <- currentLine a // get sub-string delimited by comma

courseDescription <- currentLine // get sub-string delimited by comma

WHILE currentLine is good

prerequisiteCourseNumber <- currentLine // get sub-string delimited by comma

ADD prerequisiteCourseNumber TO course.prerequisites

ENDWHILE

CALCULATE hashkey

ACCESS node AT hashkey IN nodes

IF element IS available

node.key <- hashkey

node.course <- course

ELSE

currentNode <- node AT hashkey IN nodes

CREATE newNode

newNode.key <- hashkey

newNode.course <- course

WHILE currentNode.next IS NOT nullptr

currentNode <- currentNode.next

ENDWHILE

currentNode.next <- newNode

ENDIF

ENDWHILE

close filename

END

### **TREE METHOD**

METHOD LoadFile (parameters: Node node, string filename)

BEGIN

CREATE filestream file

CREATE stringstream currentLine

file <- open filename

WHILE not end of file

CREATE course

CREATE prerequisiteCourseNumber

currentLine <- READ line from file

course.number <- currentLine a // get sub-string delimited by comma

courseDescription <- currentLine // get sub-string delimited by comma

WHILE currentLine is good

prerequisiteCourseNumber <- currentLine // get sub-string delimited by comma

ADD prerequisiteCourseNumber TO course.prerequisites

ENDWHILE

CALCULATE node

node.course <- course

CALL add(BinarySearchTree, node)

ENDWHILE

close filename

END

METHOD add(startNode, newNode)

IF startNode is NULL THEN

startNode <- newNode

RETURN

ENDIF

IF newNode < startNode THEN

add(startNode’s left child, newNode)

ELSE

add(startNode’s right child, newNode)

ENDIF

END

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

### **VECTOR METHOD:**

METHOD PrintCourse(Vector<Course> courses, string courseNum)

BEGIN

CREATE index

CREATE found

index <- 0

found <- false

WHILE index < size of courses AND found is false

IF course.number at index of courses is equal to courseNum

found <- true

ELSE

INCREMENT index

ENDIF

ENDWHILE

IF found is true

PRINT course.number at index of courses

PEINT course.description at index of courses

FOR EACH prerequisiteCourseNumber in course.prerequisite at index of courses

CALL PrintCourse(course, prerequisiteCourseNumber)

END FOR EACH

ELSE

PRINT “COURSE “ + courseNum + “ not found.”

ENDIF

END

### **HASH TABLE METHOD:**

METHOD PrintCourse(Vector<Node> nodes, string courseNum)

BEGIN

CALCULATE hashkey for courseNum

ACCESS node AT hashkey IN nodes

WHILE NOT (node.course.courseNum IS courseNum OR

node.next IS nullptr)

node <- node.next

ENDWHILE

IF courseNum FOUND THEN

PRINT course.number at index of courses

PRINT course.description at index of courses

FOR EACH prerequisiteCourseNumber in course.prerequisite at index of courses

CALL PrintCourse(course, prerequisiteCourseNumber)

END FOR EACH

ELSE

PRINT “COURSE “ + courseNum + “ not found.”

ENDIF

END

### **TREE METHOD:**

METHOD PrintCourse(Node node, string courseNum)

BEGIN

WHILE node.course.courseNum IS NOT courseNum AND node IS NOT null pointer THEN

IF course < node.course THEN

node <- FindCourse(node.left, course)

ELSE IF course > node.course THEN

node <- FindeCourse(node.right, course)

ENDIF

ENDWHILE

IF node IS NOT NULL THEN

PRINT course.number at index of courses

PRINT course.description at index of courses

FOR EACH prerequisiteCourseNumber in course.prerequisite at index of courses

CALL PrintCourse(course, prerequisiteCourseNumber)

END FOR EACH

ELSE

PRINT “COURSE “ + courseNum + “ not found.”

ENDIF

END

# **Create pseudocode for a menu**. The menu will need to perform the following:

* 1. Load Data Structure: 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.
  2. Print Course List: This will print an alphanumerically ordered list of all the courses in the Computer Science department.
  3. Print Course: This will print the course title and the prerequisites for any individual course.
  4. Exit: This will exit you out of the program.

METHOD Menu()

BEGIN

IF vector data structure THEN

CREATE dataStructure <- vector<Node> // vector method node

ELSE IF hash tree data structure THEN

CREATE dataStructure <- vector<Node> // hash table method node

ELSE

CREATE dataStructure <- Node // tree method node

ENDIF

CREATE filename <- name and path of file name to load

CREATE MenuChoice

WHILE TRUE

PRINT “MAIN MENU”

PRINT “1) Load Data Structure”

PRINT “2) Print Course List”

PRINT “3) Print Course”

PRINT “9) Exit”

PRINT “Please enter menu option: “

MenuChoide <- User Input

IF MenuChoice IS 1

CALL LoadFile (dataStructure, filename)

ELSE IF MenuChoice IS 2

CALL PrintCourseListInOrder(dataStructure)

ELSE IF MenuChoice IS 3

CREATE course

PRINT “Enter Course Alphanumeric Number: “

course <- User Input

CALL FindCourse(dataSTructure, course)

ELSE

EXIT PROGRAM

ENDIF

ENDWHILE

END

# **Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.** Continue working with the Pseudocode Document linked in the Supporting Materials section. Note that you will be designing for the same three data structures that you have been using in your previous pseudocode milestones (vector, hash table, and tree). This time you will create the final pieces of pseudocode that you will need for ABCU’s advising program. To complete this part of the process, do the following:

1. Sort the course information by alphanumeric course number from lowest to highest.
2. Print the sorted list to a display.

### **VECTOR METHOD:**

METHOD SortCourses (vector<Node> nodes)

BEGIN

CREATE sortMin Node

CREATE sortMax Node

CREATE nextSortMax Node

CREATE currentSortNode Node

CREATE swapNode Node

IF nodes.head IS NOT nodes.tail // only one node or no nodes, nothing to sort

sortMin <- nodes.head

sortMax <- nodes.tail

WHILE NOT (sortMin IS sortMax OR sortMin.next IS sortMax)

currentSortNode <- sortMin.next

IF sortMin.courseNum > sortMax.courseNum THEN

swapNode <- sortMin

sortMin <- sortMax

sortMax <- swapNode

END IF

WHILE currentSortNode IS NOT sortMax

nextSortMax <- currentSortNode

IF sortMin.courseNum > currentSortNode.courseNum THEN

swapNode <- sortMin

sortMin <- currentSortNode

currentSortNode <- sortMin

ELSE IF sortMax.courseNum < currentSortNode.courseNum THEN

swapNode <- sortMax

sortMax <- currentSortNode

currentSortNode <- swapNode

END IF

currentSortNode <- currentSortNode.next

END WHILE

sortMax <- nextSortMax

sortMin <- sortMin.next

END WHILE

IF (sortMin IS NOT sortMax) THEN

IF sortMin.courseNum > sortMax.courseNum THEN

swapNode <- sortMin

sortMin <- sortMax

sortMax <- swapNode

END IF

END IF

END IF

END

METHOD PrintCourses(vector<Node> nodes)

BEGIN

CREATE currentNode Node

WHILE currentNode.next IS NOT NULL

PRINT currentNode.course information

currentNode <- currentNode.next

END WHILE

END

METHOD PrintCoursesInOrder(vector<Node> nodes)

BEGIN

CALL SortCourses(nodes)

CALL PrintCoursesInOrder(nodes)

END

### **HASH TABLE METHOD:**

METHOD SortCourses (vector<Node> nodes)

BEGIN

// hash tables do not lend themselves to sorting since they

// are assigned position by a hashkey. This only puts courses

// in order within the key

CREATE minNode

CREATE currentNode

CREATE swapNode

FOR EACH KEY

minNode <- key.node

currentNode <- minNode.next

WHILE minNode IS NOT NULL

IF minNode.courseNum > currentNode.courseNum THEN

swapNode <- minNode.course

minNode.course <- currentNode.course

currentNode.course <- swapNode.course

ENDIF

currentNode <- currentNode.next

END WHILE

END FOR

END

METHOD PrintCourses(vector<Node> nodes)

BEGIN

// PRINT will be in hashkey order, not alphanumeric order

CREATE currentNode

FOR EACH key

currentNode <- key.node

WHILE currentNode IS NOT NULL

PRINT currentNode.course information

currentNode <- currentNode.next

END WHILE

END FOR

END

METHOD PrintCoursesInOrder(vector<Node> nodes)

BEGIN

CALL SortCourses(nodes)

CALL PrintCourses(nodes)

END

### **TREE METHOD:**

METHOD PrintCourses(Node node)

BEGIN

IF node IS NOT NULL THEN

PrintCourses(node.left)

PRINT node.course information

PrintCourses(node.right)

END IF

END

METHOD PrintCoursesInOrder(Node node)

BEGIN

// The tree is already in order. Although not balanced.

CALL PrintCourses(tree)

END

**Evaluation**

1. **Evaluate the run-time and memory of data structures that could be used to address the requirements**. In a previous assignment, you created pseudocode to do the following:
   1. Define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors.
   2. Show how to create course objects, so that one course object holds data from a single line from the input file.

Using this pseudocode written for the previous assignments, analyze the worst-case running time of each, reading the file and creating course objects, which will be the Big O value. This should not include the pseudocode written for the menu or the sample schedule above. To do this, do the following:

* 1. Specify the cost per line of code and the number of times the line will execute. Assume there are n courses stored in the data structure.
  2. Assume the cost for a line to execute is 1 unless it is calling a function, in which case the cost will be the running time of that function.

METHOD LoadFile

|  |  |  |  |
| --- | --- | --- | --- |
| **CODE (VECTOR)** | Line Cost | Times Executes | Total Cost |
| CREATE filestream file | 1 | 1 | 1 |
| CREATE stringstream currentLine | 1 | 1 | 1 |
| file <- open filename | 1 | 1 | 1 |
| WHILE not end of file | 1 | n | n |
| CREATE course | 1 | n | n |
| CREATE prerequisiteCourseNumber | 1 | n | n |
| currentLine <- READ line from file | 1 | n | n |
| course.number <- currentLine | 1 | n | n |
| courseDescription <- currentLine | 1 | n | n |
| WHILE currentLine is good | 1 | k | kn |
| prerequisiteCourseNumber <- currentLine | 1 | k | kn |
| ADD prerequisiteCourseNumber TO course.prerequisites | 1 | k | kn |
| ADD course TO courses | 1 | n | n |
| close filename | 1 | 1 | 1 |
| **Total Cost** | | | (3kn)\*(7n)+5 |
| **Runtime** | | | O(kN) |

|  |  |  |  |
| --- | --- | --- | --- |
| **CODE (Tree)** | Line Cost | Times Executes | Total Cost |
| CREATE filestream file | 1 | 1 | 1 |
| CREATE stringstream currentLine | 1 | 1 | 1 |
| file <- open filename | 1 | 1 | 1 |
| WHILE not end of file | 1 | n | n |
| CREATE course | 1 | n | n |
| CREATE prerequisiteCourseNumber | 1 | n | n |
| currentLine <- READ line from file | 1 | n | n |
| course.number <- currentLine | 1 | n | n |
| courseDescription <- currentLine | 1 | n | n |
| WHILE currentLine is good | 1 | k | kn |
| prerequisiteCourseNumber <- currentLine | 1 | k | kn |
| ADD prerequisiteCourseNumber TO course.prerequisites | 1 | k | kn |
| ENDWHILE | 0 | 0 | 0 |
| CALCULATE node | 1 | n | n |
| node.course <- course | 1 | n | n |
| CALL add(BinarySearchTree, node) | 4 | n | 4n |
| ENDWHILE | 0 | 0 | 0 |
| close filename | 1 | 1 | 1 |
| END | 0 | 0 | 0 |
| **Total Cost** | | | (3kn)+(12n)+ |
| **Runtime** | | | O(kn) |
|  |  |  |  |
| **METHOD add(startNode, newNode)** | | | |
| IF startNode is NULL THEN | 1 | 1 | 1 |
| startNode <- newNode | 0 | 0 | 0 |
| RETURN | 0 | 0 | 0 |
| ENDIF | 0 | 0 | 0 |
| IF newNode < startNode THEN | 1 | 1 | 1 |
| add(startNode’s left child, newNode) | 1 | 1 | 1 |
| ELSE | 0 | 0 | 0 |
| add(startNode’s right child, newNode) | 1 | 1 | 1 |
| ENDIF | 0 | 0 | 0 |
| END | 0 | 0 | 0 |
| **Total Cost** | | | 4 |
| **Runtime** | | | O(4) |

|  |  |  |  |
| --- | --- | --- | --- |
| **CODE (Hash Table)** | Line Cost | Times Executes | Total Cost |
| CREATE filestream file | 1 | 1 | 1 |
| CREATE stringstream currentLine | 1 | 1 | 1 |
| file <- open filename | 1 | 1 | 1 |
| WHILE not end of file | 1 | n | n |
| CREATE course | 1 | n | n |
| CREATE prerequisiteCourseNumber | 1 | n | n |
| currentLine <- READ line from file | 1 | n | n |
| course.number <- currentLine | 1 | n | n |
| courseDescription <- currentLine | 1 | n | n |
| WHILE currentLine is good | 1 | k | kn |
| prerequisiteCourseNumber <- currentLine | 1 | k | kn |
| ADD prerequisiteCourseNumber TO course.prerequisites | 1 | k | kn |
| ENDWHILE | 0 | 0 | 0 |
| CALCULATE hashkey | 1 | n | n |
| ACCESS node AT hashkey IN nodes | 1 | n | n |
| IF element IS available | 1 | n | n |
| node.key <- hashkey | 1 | n | n |
| node.course <- course | 1 | n | n |
| ELSE | 0 | 0 | 0 |
| currentNode <- node AT hashkey IN nodes | 1 | n | n |
| CREATE newNode | 1 | n | n |
| newNode.key <- hashkey | 1 | n | n |
| newNode.course <- course | 1 | n | n |
| WHILE currentNode.next IS NOT nullptr | 1 | j | jn |
| currentNode <- currentNode.next | 1 | j | jn |
| ENDWHILE | 0 | 0 | 0 |
| currentNode.next <- newNode | 1 | n | n |
| ENDIF | 0 | 0 | 0 |
| ENDWHILE | 0 | 0 | 0 |
| close filename | 1 | 1 | 1 |
| **Total Cost** | | | (2jn)+(3kn)+(16n)+4 |
| **Runtime** | | | O (kN) |

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

The insertion cost appears to have the same worst cost Big O value of N. With this said, I would still choose the tree method, preferably with balancing and or completion management. This would provide the overall best worst access to search for specific courses. Furthermore, adding a line of data to the tree results in a sorted data structure where the vector is not sorted and the hash table cannot be truly sorted. For actual loading speed, the vector data structure requires less decision processing since it simply appends a new class to the vector for each line if the data file, additionally there is no extra memory reserved for blank or null keys as in the hash table. The hash table is simpler in storing the data since a key is calculated from the data line, this is used to predetermine a position in the hash table where the new data can be appended. As the tree structure is loaded with random data such as the line data, it results in the data being sorted by the insertion process. Since the data is sorted, the average search time is reduced. The worst search time Big O could be reduced to O(logN) if the tree were balanced and completed.

1. Now that you have analyzed all three data structures, **make a recommendation for which data structure you will plan to use in your code**. Provide justification for your recommendation, based on the Big O analysis results and your analysis of the three data structures.

I recommend that the tree data structure be implemented. The structure is sorted as a result of the insertion process and providing a reasonably balanced tree structure should consistently provide near O(logN) performance on searches.