**6-2: Project One**

Cristiano Miranda

Southern New Hampshire University

CS-300: Analysis and Design

Professor Muthukudage

12 October 2025

**6-2: Project One**

Global Pseudocode:

* STRUT Course
  + INIT number
  + INIT name
  + INIT Vector<Strings> preReqs
* ENDSTRUCT
* PUBLIC FUNCTION printCourse (course)
  + START
  + DISPLAY “Course: ” + course.number + “ | ” + course.name + “ | ”
  + INIT numPreReqs = GetLength (course.preReqs)
  + IF numPreReqs IS NOT EQUAL TO 0 THEN
    - FOR index FROM 0 UNTIL numPreReqs
      * DISPLAY course.preReqs[index] + “, ”
    - ENDFOR
  + ELSE
    - DISPLAY “No prerequisites for this course.”
  + ENDIF
  + END
* ENDMETHOD
* FUNCTION loadCourseData (String csvFilePath, courses) {
  + START
  + TRY
    - DISPLAY ”Loading File”
    - INIT file

//reads data from the csv file and will throw an error if the file path is incorrect or the file is unable to be opened

* + - CALL csv::Parser (csvFilePath) AND SET file = result
    - FOR index from 0 until file.rowCount()
      * INIT rowLength = GetLength(file[index])
      * IF rowLength IS GREATER THAN OR EQUAL TO 2 THEN
        + INIT course = new Course ()
        + SET course.number = file[index][0]
        + SET course.name = file[index][1]
        + IF rowLength IS GREATER THAN 2 THEN

FOR nestedIndex FROM 2 UNTIL rowLength

FOR fileIndex FROM 0 UNTIL file.rowCount()

// check that the prerequisite is associated with a course number in the file

IF file[index][nestedIndex] ID EQUAL TO file[fileIndex][0] THEN

CONTINUE

ELSE

INIT ERROR = “File row ” + index + “ has prerequisites for non-listed classes!”

THROW ERROR

ENDIF

ENDFOR

INSERT file[index][nestedIndex] INTO course.preReqs

ENDFOR

* + - * + ENDIF

// Call associate data structure’s insert method to insert new course object

* + - * + CALL courses->Insert (course)
      * ELSE
        + INIT ERROR = “File row ” + index + “ has insufficient parameters!”
        + THROW ERROR
      * ENDIF
    - ENDFOR
    - CLOSE file
    - RETURN courses
    - END
  + CATCH (error)
    - DISPLAY “Error loading data from the file!”
    - DISPLAY error message
    - RETURN EMPTY VECTOR
    - END
  + ENDTRY
* ENDFUNCTION
* FUNCTION main (argc, argv) // Will display menu upon launching program
  + START
  + INIT csvPath
  + INIT courseNumber
  + INIT courses // Either a Vector, Hash Table, or Binary Search Tree for its data structure
  + IF argc IS EQUAL TO 2 THEN
    - SET csvPath = element AT argv[1]
    - SET courseNumber = “CSCI300”
  + ELSE IF argc IS EQUAL TO 3 THEN
    - SET csvPath = element AT argv[1]
    - SET courseNumber = element AT argv[2]
  + ELSE
    - SET csvPath = “CS\_300\_Course\_Information.csv”
    - SET courseNumber = “CSCI300”
  + ENDIF
  + INIT choice
  + SET choice = 0
  + WHILE choice IS NOT EQUAL TO 9
    - DISPLAY “Menu:

1 – Load Course Data

2 – Display All Courses

3 – Display Course

9 – Exit

Enter Choice: “

* + - GET choice
    - IF choice IS EQUAL TO 1 THEN
      * CALL loadCourseData (csvPath, courses)
    - ELSE IF choice IS EQUAL TO 2 THEN
      * IF GetLength(courses) IS NOT EQUAL TO 0 THEN
        + CALL courses->DisplayAll ()
      * ELSE
        + DISPLAY “Load Courses First!”
      * ENDIF
    - ELSE IF choice IS EQUAL TO 3 THEN
      * IF GetLength(courses) IS NOT EQUAL TO 0 THEN
        + INIT course
        + CALL courses->Search (courseNumber) AND SET course = result
        + DISPLAY course
      * ELSE
        + DISPLAY “Load Courses First!”
      * ENDIF
    - ELSE IF choice IS EQUAL TO 9 THEN
      * BREAK
    - ELSE
      * DISPLAY “Invalid Input!”
      * SET choice = 0
      * CLEAR user input
    - ENDIF
  + ENDWHILE
  + DISPLAY “Closing Program…”
  + END
* ENDFUNCTION

Vector Data Structure Pseudocode:

* CLASS CoursesVector
  + PRIVATE VECTOR courses
  + PUBLIC CONSTRUCTOR ()
    - START
    - SET courses = EMPTY VECTOR
    - END
  + ENDCONSTRUCTOR
  + PUBLIC DECONSTRUCTOR ()
    - START
    - ERASE ALL elements FROM courses
    - END
  + ENDDECONSTRUCTOR
  + PUBLIC METHOD Insert (course)
    - START
    - INSERT course INTO courses
    - END
  + ENDMETHOD
  + PUBLIC METHOD Search (courseNumber)
    - START
    - FOR index FROM 0 UNTIL GetLength (courses)
      * IF courseNumber IS EQUAL TO element AT courses[index].number THEN
        + RETURN courses[index]
      * ENDIF
    - ENDFOR
    - END
    - ENDMETHOD
  + PRIVATE METHOD partitionVector (coursesVector, begin, end)
    - START
    - INIT lowIndex
    - INIT highIndex
    - INIT midpoint
    - INIT pivotValue
    - SET lowIndex = begin
    - SET highIndex = end
    - SET midpoint = lowIndex + (highIndex – lowIndex) / 2
    - // Set to object’s associated number value
    - SET pivotValue = element AT coursesVector [midpoint].number
    - INIT done
    - SET done = FALSE
    - WHILE done IS NOT TRUE
      * WHILE element AT coursesVector [lowIndex].number IS LESS THEN pivotValue
        + INCREMENT lowIndex
      * ENDWHILE
      * WHILE element AT coursesVector [lowIndex].number IS GREATER THEN pivotValue
        + DECREMENT highIndex
      * ENDWHILE
      * IF lowIndex IS GREATER THAN OR EQUAL TO highIndex THEN
        + SET done = TRUE
      * ELSE

// Swap element at the lowIndex and highIndex positions

* + - * + CALL std::swap (coursesVector [lowIndex], coursesVector [highIndex])
        + INCREMENT lowIndex
        + DECREMENT highIndex
      * ENDIF
    - ENDWHILE
    - RETURN highIndex
    - END
  + ENDMETHOD
  + PRIVATE METHOD quickSort (coursesVector, begin, end)
    - START
    - INIT midIndex
    - SET midIndex = 0
    - IF begin IS GREATER THAN OR EQUAL TO end THEN
      * RETURN
      * END
    - ENDIF
    - CALL partitionVector (coursesVector, begin, end) AND SET midIndex = result
    - CALL quickSort (coursesVector, begin, midIndex)
    - CALL quickSort (coursesVector, midIndex + 1, end)
    - END
  + ENDMETHOD
  + PUBLIC METHOD DisplayAll ()
    - START
    - INIT sortedCourses
    - FOR index FROM 0 UNTIL GetLength (courses)
      * INSERT courses[index] INTO sortedCourses
    - ENDFOR
    - CALL quickSort (sortedCourses, 0, GetLength (sortedCourses) - 1)
    - FOR index FROM 0 UNTIL GetLength (sortedCourses)
      * CALL printCourse (sortedCourses[index])
    - ENDFOR
    - END
  + ENDMETHOD
* ENDCLASS

HashTable Data Structure Pseudocode:

* CLASS HashTable
  + DEFINE PRIVATE STRUCT Node
    - INIT key // should never be a negative value
    - INIT course
    - INIT POINTER next
    - CONSTRUCTOR ()
      * START
      * SET key = MAXIMUM INTEGER VALUE
      * POINT next TO null
      * END
    - ENDCONSTRUCTOR
    - CONSTRUCTOR (newCourse)
      * START
      * SET course = newCourse
      * CALL CONSTRUCTOR ()
      * END
    - ENDCONSTRUCTOR
    - CONSTRUCTOR (newCourse, nodeKey)
      * START
      * SET key = nodeKey
      * CALL CONSTRUCTOR (newCourse)
      * END
    - ENDCONSTRUCTOR
  + ENDSTRUCT
  + INIT PRIVATE VECTOR courses
  + INIT PRIVATE tableSize = DEFAULT\_SIZE // set value to a predetermined value on program start
  + PUBLIC CONSTRUCTOR ()
    - START
    - RESIZE nodes VECTOR TO tableSize
    - END
  + ENDCONSTRUCTOR
  + PUBLIC CONSTRUCTOR (newSize)
    - START
    - SET tableSize = newSize
    - RESIZE nodes VECTOR TO size
    - END
  + ENDCONSTRUCTOR
  + PUBLIC DECONSTRUCTOR
    - START
    - ERASE all elements FROM nodes
    - END
  + ENDDECONSTRUCTOR
  + PUBLIC METHOD hashStringToKey (courseNumber)
    - START
    - INIT stringTotal
    - FOR EACH character IN courseNumber
      * ADD integer value for character TO stringTotal
    - CALCULATE stringTotal MOD tableSize
    - RETURN calculation result
    - END
  + ENDMETHOD
  + PUBLIC METHOD Insert (course)
    - START
      * INIT nodeKey
      * CALL hashStringToKey (course.number) AND SET nodeKey = result
      * INIT POINTER currNode AND POINT TO element AT courses[nodeKey]
      * IF currNode IS POINTING TO null THEN
        + INIT new Node (course, nodeKey)
        + INSERT new Node AT courses[nodeKey]
      * ELSE IF currNode->key IS EQUAL TO MAXIMUM INTEGER VALUE THEN
        + // element at vector was removed and can be repopulated
        + SET currNode->course = course
        + SET currNode->key = nodeKey
      * ELSE
        + WHILE currNode->next IS NOT POINTING TO null

POINT currNode TO currNode->next

* + - * + ENDWHILE
        + INIT new Node (course, nodeKey)
        + POINT currNode->next TO new Node
      * ENDIF
    - END
  + ENDMETHOD
  + PUBLIC METHOD Search (courseNumber)
    - START
    - INIT nodeKey
    - CALL hashStringToKey (courseNumber) AND SET nodeKey = result
    - INIT POINTER currNode AND POINT TO element AT courses[nodeKey]
    - IF currNode IS NOT POINTING TO null AND currNode->key IS NOT EQUAL TO MAXIMUM INTEGER VALUE THEN
      * WHILE currNode IS NOT POINTING TO null
        + IF currNode->course.number IS EQUAL TO courseNumber THEN

RETURN currNode->course

END

* + - * + ENDIF
        + POINT currNode TO currNode->next
      * ENDWHILE
    - ENDIF
    - DISPLAY “No associated course found!”
    - RETURN new Course Object
    - END
  + ENDMETHOD
  + PRIVATE METHOD partitionVector (coursesVector, begin, end)
    - START
    - INIT lowIndex
    - INIT highIndex
    - INIT midpoint
    - INIT pivotValue
    - SET lowIndex = begin
    - SET highIndex = end
    - SET midpoint = lowIndex + (highIndex – lowIndex) / 2
      * // Set to object’s associated number value
    - SET pivotValue = element AT coursesVector [midpoint].number
    - INIT done
    - SET done = FALSE
    - WHILE done IS NOT TRUE
      * WHILE element AT coursesVector [lowIndex].number IS LESS THEN pivotValue
        + INCREMENT lowIndex
      * ENDWHILE
      * WHILE element AT coursesVector [lowIndex].number IS GREATER THEN pivotValue
        + DECREMENT highIndex
      * ENDWHILE
      * IF lowIndex IS GREATER THAN OR EQUAL TO highIndex THEN
        + SET done = TRUE
      * ELSE
        + // Swap element at the lowIndex and highIndex positions
        + CALL std::swap (coursesVector [lowIndex], coursesVector [highIndex])
        + INCREMENT lowIndex
        + DECREMENT highIndex
      * ENDIF
    - ENDWHILE
    - RETURN highIndex
    - END
  + ENDMETHOD
  + PRIVATE METHOD quickSort (coursesVector, begin, end)
    - START
    - INIT midIndex
    - SET midIndex = 0
    - IF begin IS GREATER THAN OR EQUAL TO end THEN
      * RETURN
      * END
    - ENDIF
    - CALL partitionVector (coursesVector, begin, end) AND SET midIndex = result
    - CALL quickSort (coursesVector, begin, midIndex)
    - CALL quickSort (coursesVector, midIndex + 1, end)
    - END
  + ENDMETHOD
  + PRIVATE METHOD getAllCourses ()
    - START
    - INIT coursesVector
    - FOR index FROM 0 UNTIL GetLength(courses)
      * INIT POINTER currNode
      * POINT currNode TO element AT courses[index]
      * IF currNode IS NOT POINTING TO NULL AND currNode->key IS NOT EQUAL TO MAXIMUM INTEGER VALUE THEN
        + WHILE currNode IS NOT POINTING TO NULL

INSERT currNode->course INTO coursesVector

POINT currNode TO currNode->next

* + - * + ENDWHILE
      * ENDIF
    - ENDFOR
    - CALL quickSort (coursesVector, 0, GetLength (coursesVector) - 1)
    - RETURN coursesVector
    - END
  + ENDMETHOD
  + PUBLIC METHOD DisplayAll ()
    - START

// Hash Tables cannot be sorted.

// Each course must be added into a vector to be sorted and displayed in order

* + - INIT sortedCourses
    - CALL getAllCourses () AND SET sortedCourses = result
    - FOR EACH index FROM 0 UNTIL GetLength (sortedCourses)
      * CALL printCourse (sortedCourse[index])
    - ENDFOR
    - END
  + ENDMETHOD
* ENDCLASS

Binary Search Tree Data Structure Pseudocode:

* CLASS BinarySearchTree
  + DEFINE PRIVATE STRUCT Node
    - INIT course
    - INIT POINTER right
    - INIT POINTER left
    - CONSTRUCTOR ()
      * START
      * POINT right TO null
      * POINT left TO null
      * END
    - ENDCONSTRUCTOR
    - CONSTRUCTOR (newCourse)
      * START
      * SET course = newCourse
      * CALL CONSTRUCTOR ()
      * END
    - ENDCONSTRUCTOR
  + ENDSTRUCT
  + INIT PRIVATE POINTER root
  + PUBLIC CONSTRUCTOR ()
    - START
    - POINT root TO null
    - END
  + ENDCONSTRUCTOR
  + PUBLIC DECONSTRUCTOR
    - START
    - INIT POINTER currNode
    - POINT currNode TO root
    - WHILE currNode IS NOT POINTING TO null
      * CALL removeNode (currNode, currNode->course.number) AND POINT currNode TO result
    - ENDWHILE
    - END
  + ENDDECONSTRUCTOR
  + PRIVATE METHOD removeNode (node, courseNumber)
    - START
    - IF node IS POINTING TO null THEN
      * RETURN node
    - ELSE IF node->course.number IS GREATER THAN courseNumber THEN
      * CALL removeNode (node->left, courseNumber) AND POINT node->left TO result
    - ELSE IF node->course.number IS LESS THAN courseNumber THEN
      * CALL removeNode (node->right, courseNumber) AND POINT node->right TO result
    - ELSE
      * IF node->left IS POINTING TO null AND node->right IS POINTING TO null THEN
        + DELETE node
        + POINT node TO null
      * ELSE IF node->left IS NOT POINTING TO null AND node->right IS POINTING TO null THEN
        + INIT POINTER removedNode
        + POINT removedNode TO node
        + POINT node TO node->left
        + DELETE removedNode
        + POINT removedNode TO null
      * ELSE IF node->left IS POINTING TO null AND node->right IS NOT POINTING TO null THEN
        + INIT POINTER removedNode
        + POINT removedNode TO node
        + POINT node TO node->right
        + DELETE removedNode
        + POINT removedNode TO null
      * ELSE
        + INIT POINTER succNode
        + POINT succNode TO node->right
        + WHILE succNode->left IS NOT POINTING TO null

POINT succNode TO succNode->left

* + - * + ENDWHILE
        + INIT succNodeCourseData
        + SET succNodeCourseData = succNode->course
        + SET node->course = succNodeCourseData
        + CALL removeNode (node->right, succNodeCourseData.number)
      * ENDIF
    - ENDIF
    - RETURN node
    - END
  + ENDMETHOD
  + PUBLIC METHOD Insert (course)
    - START
    - IF root IS POINTING TO null THEN
      * INIT new Node (course)
      * POINT root TO new Node
    - ELSE
      * CALL addNode (root, course)
    - ENDIF
    - END
  + ENDPUBLIC METHOD
  + PRIVATE METHOD addNode (node, course)
    - START
    - IF node->course.number IS GREATER THAN course.number THEN
      * IF node->left IS POINTING TO null THEN
        + INIT new Node (course)
        + POINT node->left TO new Node
      * ELSE
        + CALL addNode (node->left, course)
      * ENDIF
    - ELSE
      * IF node->right IS POINTING TO null THEN
        + INIT new Node (course)
        + POINT node->right TO new Node
      * ELSE
        + CALL addNode (node->right, course)
      * ENDIF
    - ENDIF
    - END
  + ENDMETHOD
  + PUBLIC METHOD Search (courseNumber)
    - START
    - INIT POINTER currNode
    - POINT currNode TO root
    - WHILE currNode IS NOT POINTING TO null
      * IF currNode->course.number IS EQUAL TO courseNumber THEN
        + CALL printCourse (currNode->course)
        + RETURN
      * ELSE IF currNode->course.number IS GREATER THAN courseNumber THEN
        + POINT currNode TO currNode->left
      * ELSE
        + POINT currNode TO currNode->right
      * ENDIF
    - ENDWHILE
    - DISPLAY “No associated course found!”
    - RETURN
    - END
  + ENDMETHOD
  + PRIVATE METHOD inOrder (node)
    - START
      * IF node IS POINTING TO null THEN
        + RETURN
      * ENDIF
      * CALL inOrder (node->left)
      * CALL printCourse (node->course)
      * CALL inOrder (node->right)
    - END
  + ENDMETHOD
  + PUBLIC METHOD DisplayAll ()
    - START
      * CALL inOrder (root)
    - END
  + ENDMETHOD
* ENDCLASS

Run Time Evaluation:

* Vector Data Structure

|  |  |  |  |
| --- | --- | --- | --- |
| Insert Method | | | |
| Code | Line Cost | # Times  Executes | Total Cost |
| INSERT course INTO courses | i | 1 | i |
| Total Cost | | | i |
| Runtime | | | i |

|  |  |  |  |
| --- | --- | --- | --- |
| Loading File & Inserting New Courses | | | |
| Code | Line Cost | # Times  Executes | Total Cost |
| DISPLAY “Loading File” | 1 | 1 | 1 |
| INIT file | 1 | 1 | 1 |
| CALL csv::Parser AND SET file = result | 1 | 1 | 1 |
| FOR index FROM 0 UNTIL file.rowCount() | 1 | n | n |
| INIT rowLength | 1 | n | n |
| SET rowLength = GetLength(file[index]) | 1 | n | n |
| IF rowLength IS GREATER THAN OR EQUAL TO 2 THEN | 1 | n | n |
| INIT course | 1 | n | n |
| SET course = new Course () | 1 | n | n |
| SET course.number = file[index][0] | 1 | n | n |
| SET course.name = file[index][1] | 1 | n | n |
| IF rowLength IS GREATER THAN 2 THEN | 1 | n | n |
| FOR nestedIndex FROM 2 UNTIL rowLength | 1 | m | m |
| FOR fileIndex FROM 0 UNTIL file.rowCount() | 1 | n | n |
| IF file[index][nestedIndex] IS EQUAL TO file[fileIndex][0] THEN | 1 | n | n |
| CONTINUE | 1 | n | n |
| ELSE | 1 | 1 | 1 |
| INIT error | 1 | 1 | 1 |
| SET error = “File row ” + index + “ has prerequisites for non-listed classes!” | 1 | 1 | 1 |
| THROW error | 1 | 1 | 1 |
| INSERT file[index][nestedIndex] INTO course.preReqs | 1 | m | m |
| ELSE | 1 | 1 | 1 |
| INIT error | 1 | 1 | 1 |
| SET error = “File row ” + index + “ has insufficient parameters!” | 1 | 1 | 1 |
| THROW error | 1 | 1 | 1 |
| CALL courses->Insert (course) | i | n | n \* i |
| CLOSE file | 1 | 1 | 1 |
| RETURN courses | 1 | 1 | 1 |
| Total Cost | | | ((9n + (n \* i) + 4) \* (2m \* (3n + 4))) + 5 |
| Runtime | | | O (n2 \* i \* m) |

* Hash Table Data Structure

|  |  |  |  |
| --- | --- | --- | --- |
| Hash String To Key Method | | | |
| Code | Line Cost | # Times  Executes | Total Cost |
| INIT stringTotal | 1 | 1 | 1 |
| FOR EACH character IN courseNumber | 1 | j | j |
| ADD integer value for character TO stringTotal | 1 | j | j |
| CALCULATE stringTotal MOD tableSize | 1 | 1 | 1 |
| RETURN calculation result | 1 | 1 | 1 |
| Total Cost | | | 2j + 3 |
| Runtime | | | O(j) |

|  |  |  |  |
| --- | --- | --- | --- |
| Insert Method | | | |
| Code | Line Cost | # Times  Executes | Total Cost |
| INIT nodeKey | 1 | 1 | 1 |
| CALL hashStringToKey (course.number) AND SET nodeKey = result | j | 1 | j |
| INIT POINTER currNode AND POINT TO element AT courses [nodeKey] | 1 | 1 | 1 |
| IF currNode IS POINTING TO null THEN | 1 | 1 | 1 |
| INIT new Node (course, nodeKey) | 1 | 1 | 1 |
| INSERT new Node AT courses[nodeKey] | 1 | 1 | 1 |
| ELSE IF currNode->key IS EQUAL TO MAXIMUM INTEGER VALUE THEN | 1 | 1 | 1 |
| SET currNode->course = course | 1 | 1 | 1 |
| SET currNode->key = nodeKey | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| WHILE currNode->next IS NOT POINTING TO null | 1 | k | k |
| POINT currNode TO currNode->next | 1 | k | k |
| INIT new Node (course, nodeKey) | 1 | 1 | 1 |
| POINT currNode->next TO new Node | 1 | 1 | 1 |
| Total Cost | | | 2k + j + 11 |
| Runtime | | | O (k + j) |

|  |  |  |  |
| --- | --- | --- | --- |
| Loading File & Inserting New Courses | | | |
| Code | Line Cost | # Times  Executes | Total Cost |
| DISPLAY “Loading File” | 1 | 1 | 1 |
| INIT file | 1 | 1 | 1 |
| CALL csv::Parser AND SET file = result | 1 | 1 | 1 |
| FOR index FROM 0 UNTIL file.rowCount() | 1 | n | n |
| INIT rowLength | 1 | n | n |
| SET rowLength = GetLength(file[index]) | 1 | n | n |
| IF rowLength IS GREATER THAN OR EQUAL TO 2 THEN | 1 | n | n |
| INIT course | 1 | n | n |
| SET course = new Course () | 1 | n | n |
| SET course.number = file[index][0] | 1 | n | n |
| SET course.name = file[index][1] | 1 | n | n |
| IF rowLength IS GREATER THAN 2 THEN | 1 | n | n |
| FOR nestedIndex FROM 2 UNTIL rowLength | 1 | m | m |
| FOR fileIndex FROM 0 UNTIL file. rowCount() | 1 | n | n |
| IF file[index][nestedIndex] IS EQUAL TO file[fileIndex][0] THEN | 1 | n | n |
| CONTINUE | 1 | n | n |
| ELSE | 1 | 1 | 1 |
| INIT error | 1 | 1 | 1 |
| SET error = “File row ” + index + “ has prerequisites for non-listed classes!” | 1 | 1 | 1 |
| THROW error | 1 | 1 | 1 |
| INSERT file[index][nestedIndex] INTO course.preReqs | 1 | m | m |
| ELSE | 1 | 1 | 1 |
| INIT error | 1 | 1 | 1 |
| SET error = “File row ” + index + “ has insufficient parameters!” | 1 | 1 | 1 |
| THROW error | 1 | 1 | 1 |
| CALL courses->Insert (course) | k + j | n | n \* (k + j) |
| CLOSE file | 1 | 1 | 1 |
| RETURN courses | 1 | 1 | 1 |
| Total Cost | | | ((9n + n \* (k + j) + 4) \* (2m \* (3n + 4))) + 5 |
| Runtime | | | O (n2 \* (k + j) \* m) |

* Binary Search Tree Data Structure

|  |  |  |  |
| --- | --- | --- | --- |
| Add Node Method | | | |
| Code | Line Cost | # Times  Executes | Total Cost |
| IF node->course.number IS GREATER THAN course.number THEN | 1 | 1 | 1 |
| IF node->left IS POINTING TO NULL THEN | 1 | 1 | 1 |
| INIT new Node (course) | 1 | 1 | 1 |
| POINT node->left TO new Node | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| CALL addNode (node->left, course) | p | 1 | p |
| ELSE | 1 | 1 | 1 |
| IF node->right IS POINTING TO null THEN | 1 | 1 | 1 |
| INIT new Node (course) | 1 | 1 | 1 |
| POINT node->right TO new Node | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| CALL addNode (node->right, course) | p | 1 | p |
| Total Cost | | | 2p + 10 |
| Runtime | | | O (p) |

|  |  |  |  |
| --- | --- | --- | --- |
| Insert Method | | | |
| Code | Line Cost | # Times  Executes | Total Cost |
| IF root IS POINTING TO null THEN | 1 | 1 | 1 |
| INIT new Node (course) | 1 | 1 | 1 |
| POINT root TO new Node | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| CALL addNode (root, course) | p | 1 | p |
| Total Cost | | | p + 4 |
| Runtime | | | O (p) |

|  |  |  |  |
| --- | --- | --- | --- |
| Loading File & Inserting New Courses | | | |
| Code | Line Cost | # Times  Executes | Total Cost |
| DISPLAY “Loading File” | 1 | 1 | 1 |
| INIT file | 1 | 1 | 1 |
| CALL csv::Parser AND SET file = result | 1 | 1 | 1 |
| FOR index FROM 0 UNTIL file.rowCount() | 1 | n | n |
| INIT rowLength | 1 | n | n |
| SET rowLength = GetLength(file[index]) | 1 | n | n |
| IF rowLength IS GREATER THAN OR EQUAL TO 2 THEN | 1 | n | n |
| INIT course | 1 | n | n |
| SET course = new Course () | 1 | n | n |
| SET course.number = file[index][0] | 1 | n | n |
| SET course.name = file[index][1] | 1 | n | n |
| IF rowLength IS GREATER THAN 2 THEN | 1 | n | n |
| FOR nestedIndex FROM 2 UNTIL rowLength | 1 | m | m |
| FOR fileIndex FROM 0 UNTIL file.rowCount () | 1 | n | n |
| IF file[index][nestedIndex] IS EQUAL TO file[fileIndex][0] THEN | 1 | n | n |
| CONTINUE | 1 | n | n |
| ELSE | 1 | 1 | 1 |
| INIT error | 1 | 1 | 1 |
| SET error = “File row ” + index + “ has prerequisites for non-listed classes!” | 1 | 1 | 1 |
| THROW error | 1 | 1 | 1 |
| INSERT file[index][nestedIndex] INTO course.preReqs | 1 | m | m |
| ELSE | 1 | 1 | 1 |
| INIT error | 1 | 1 | 1 |
| SET error = “File row ” + index + “ has insufficient parameters!” | 1 | 1 | 1 |
| THROW error | 1 | 1 | 1 |
| CALL courses->Insert (course) | p | n | n \* p |
| CLOSE file | 1 | 1 | 1 |
| RETURN courses | 1 | 1 | 1 |
| Total Cost | | | ((9n + (n \* p) + 4) \* (2m \* (3n + 4))) + 5 |
| Runtime | | | O(n2 \* m \* p) |

Advantages and Disadvantages:

Upon completing the pseudocode and runtime analysis for the three data structures, the different implementations would all satisfy the requirements, but each has distinct advantages over the others. One of the most significant advantages of the Binary Search Tree is that it stores the courses in a sorted order, with the items smaller than the current node on the left and greater than or equal to the current node on the right. Having the data stored in an organized manner by default eliminates the need for sorting elements before displaying them to the user. In contrast, the Vector data structure, unless the data being inserted into it is already presorted, requires the use of a sorting algorithm to correctly display the courses in alphanumeric order, which increases the runtime of the display all courses method. In terms of the Hash Table, it encounters a larger issue, considering that data in a hash table cannot be sorted. The Hash Table utilizes hash keys, which do not store data in an organized manner. Thus, to sort and display the data within a Hash Table requires pulling all data from the Hash Table and inserting it into another data structure, which again introduces more complexity and memory requirements into the project. Still, the Hash Table does have the advantage of having the quickest average runtime when it comes to searching data. While the Hash Table's worst runtime is O(n), which occurs when all elements in the Hash Table share the same hash key and are stored in the same singly linked list, most Hash Tables can be easily resized, and their hashing function can be adapted to have each data node's key be unique for each element. The latter results in an average runtime of O (m) for this program when searching for data, considering that the length of the course number string, m, must be iterated over when calculating the hash key, meaning that the user would have a good experience when searching for data since rather than the runtime increasing based on the amount of data in the Hash Table, it is instead dependent on the length of the current course number string. Having runtime independent of the amount of data within the Hash Table means that even as the user adds more data nodes to the Hash Table, the program performs at a consistent speed, whether the Hash Table has 1000 elements or 1000000 elements. In contrast, the Vector data structure has an average runtime of O (n), and the Binary Search Tree has an average runtime of O (log(n)), with n equating to the amount of elements within the Vector or Binary Search Tree, which means that as the amount of data increases, the program will begin to slow down as the user searches or inserts more data. The Vector data structure has the benefit of being consistent, regardless of its average or worst-case runtime. For the Vector, it has the advantage of being simple to implement and include within the program. While this will not affect the end user, it does reduce the potential for errors during development, as the amount of boilerplate code needed is significantly reduced with the inclusion of vectors in C++. Thus, the need to create specific actions, such as insertion, is not necessary since these methods are already included.

However, there are also disadvantages to these data structures, especially in this situation. For the Binary Search Tree, one of the most significant issues is the potential for the tree to become unbalanced, which would result in the worst-case scenario runtimes occurring. This problem occurs when the data is inserted in ascending or descending order, as all the data becomes skewed to the left or right of the root, since all preceding elements will be greater than or less than the previous node. The resulting tree would be unbalanced and at its near-maximum height, resulting in a singly linked list rather than a Binary Search Tree. The Hash Table's most significant issue would be its inability to display its data in a sorted order without utilizing a different data structure, as mentioned above, there is no way to organize the data in a Hash Table, and since displaying all courses in the ascending alphanumeric order is required, this issue becomes problematic and requires the use of an additional data structure to first store the data, sort the data within it, and then finally display it to the user which takes more runtime and memory since the data has to be processed twice, once for pulling all data from the hashtable and inserting into the other data structure and again for sorting, and stored in twice in memory due to the need for an additional data structure. The Vector also has its issues, considering that as the data grows, the runtime greatly increases. Although it is the simplest to implement in the program, this comes at the cost of significantly increasing the runtime as the amount of data imported by the user increases. As the user inputs more data, rather than the runtime growing slowly, such as logarithmically or even remaining constant, it grows linearly. Thus, the program will run 10 times slower when the user loads 1000 courses as opposed to when they load 100 courses. While the initial test data consists of fewer than 10 elements, the amount of data used in other scenarios will likely be greater; thus, the program should prioritize reducing runtime for the end user rather than reducing program time for the developer.

Recommendation:

Upon completing the calculations for the worst-case scenario runtime of all data structures for this program, the Vector has a runtime of O (n2 \* i \* m), the Hash Table has a runtime of O (n2 \* (k + j) \* m), and the Binary Search Tree has a runtime of O(n2 \* m \* p), with n equating to the amount of rows in the csvFile, m equating to the length of a row from the csvFile, i equating to the length of the Vector, k equating to the length of the singly linked list within a Hash Table index, j equating to the length of the course number string, and p equating to the height of the Binary Search Tree. Based on these calculations and the advantages and disadvantages of these data structures, I would utilize the Binary Search Tree for this program, as it offers multiple benefits over the other data structures that are particularly advantageous for this application. For instance, unlike the Vector and Hash Table, which both store their data in an unorganized manner, the Binary Search Tree organizes its nodes by first comparing the new node's value to the current node and placing it to the left if it is smaller or to the right if it is greater than or equal. Thus, upon inserting new nodes into the tree, they are inserted into a sorted position, which removes the need to sort the tree later when displaying all courses. Instead, a simple recursive algorithm in which the tree is traversed in order, meaning that the left-most node is accessed, then its parent, and the parent's right node, which will display the nodes in order based on their value. In this case, calling the nodes in the displayed order will print all courses in their alphanumeric order, without having to sort through the data beforehand. Additionally, even in the worst-case scenario, in which the Binary Search Tree reaches its nearly maximum height and behaves like a linked list, it still would have a faster or similar runtime to the other data structures in their worst-case scenarios. Additionally, this issue can be addressed by shuffling the course objects before inserting them into the Binary Search Tree, which alleviates these issues by reordering the course elements before insertion. While this will require additional runtime, it will only be necessary when importing the data for the first time, which is typically done when the program is started. Even though additional runtime would be added to the program, it would significantly reduce the likelihood of the worst-case scenario runtime occurring for the user, which can substantially decrease the runtime of other, more critical aspects of the program, such as searching and displaying all courses.