**6-2 Project One**

**Cory Remick**

**CS-300 Data Structure Analysis and Design**

**David Ostrowski**

**12/03/2021**

**Introduction**

The purpose of this document is to provide a pseudocode implementation of a College Computer Science advising system that lists Computer Science courses and their prerequisites. The implementation will explore several abstract data types and provide runtime analysis in Big-O notation. The different implementations that will be analyzed will use a vector abstract data type (CS 300: Data Structures and Algorithms, p. Section 1.4), a hash table abstract data type (CS 300: Data Structures and Algorithms, p. Section 5.1), and binary search tree abstract data type (CS 300: Data Structures and Algorithms, 2019, p. Section 6.1).

Each implementation, the vector, the hash table, and the tree are expected to provide a standard programming interface to promote quick replacement. Methods needed by the program should use "Insert" for adding elements, "Search" for finding elements, "PrintCourse" for printing one course, and "PrintAll" for printing all courses in order of the course id. Properties and methods internal to the abstract data type may use similar naming conventions but are not required.

The pseudocode will use code re-use but will provide overload implementations for each of the different types where necessary. The Big-O analysis will follow and then recommend which implementation to use. The final decision for the implementation choice is left to the discretion of ABCU and its agents.

**Pseudocode**

Data Structure Section

**HashNode Class**

[**Course Class Hash Table**] Course

HashNode\* next;

**HashNode() {**

Set next = nullptr

}

**HashNode(course) {**

Call HashNode()

Set Course = course

}

**TreeNode Class**

[**Course Class Tree]** Course

TreeNode\* left;

TreeNode\* right

**TreeNode() {**

left = nullptr;

right = nullptr;

**}**

**TreeNode(course) {**

Call TreeNode()

Set Course = course

**}**

**Course Class Vector**

String Id

String Title

Vector<Course&> Prerequisites

**Course Class Hash Table**

String Id

String Title

Hashtable<Course&> Prerequisites

**Course Class Tree**

String Id

String Title

BinarySearchTree<Course&> Prerequisites

**Vector Class**

DeclareInteger Size

Declare Boolean Sorted

Allocate [**Course Class Vector**][] courses = New [**Course Class Vector**] [100]

**Insert(course) {**

If Size equals courses length

Call ExpandArray()

Set courses[size] = course

Set Size = Size + 1

If Sorted

Set Sorted = False

}

**ExpandArray() {**

Allocate tempArray with size twice courses array

For each index in courses

Set tempArray[index] = courses[index]

Free courses

Set courses = tempArray

}

**Sort() {**

If Not Sorted

Call MergeSort(courses, 0, Size – 1) // courses by reference

}

**MergeSort(courses[] by ref, low, high) {**

Declare Integer mid

If (low < high) {

Set mid = (low + high) / 2 // find mid index point

MergeSort(courses, low, mid) // recursive sort left half

MergeSort(courses, mid + 1, high) // recursive sort right half

Merge(courses, low, mid, high)

}

}

**Merge(courses[] by ref, low, mid, high) {**

Declare Integer LeftPosition

Declare Integer RightPosition

Declare Integer MergePosition

Declare Integer MergeSize

Set MergeSize = high – low + 1

Set LeftPosition = low

Set RightPosition = mid + 1

Set MergePosition = 0

Declare [**Course Class Vector]** mergedCourses[MergeSize]

// add smallest course id to mergedCourses array from either left or right partition

While LeftPosition <= mid And RightPosition <= high

If courses[LeftPosition].Id <= courses[RightPosition].Id

Set mergedCourses[MergePosition] = courses[LeftPosition]

Set LeftPosition = LeftPosition + 1

Else

Set mergedCourses[MergePosition] = courses[RightPosition]

Set RightPosition = RightPosition + 1

Set MergePosition = MergePosition + 1

// add remaining left partition entries, if any

While LeftPosition <= mid

Set mergedCourses[MergePosition] = courses[LeftPosition]

Set LeftPosition = LeftPosition + 1

Set MergePosition = MergePosition + 1

// add remaining right partition entries, if any

While RightPosition <= high

Set mergedCourses[MergePosition] = courses[RightPosition]

Set RightPosition = RightPosition + 1

Set MergePosition = MergePosition + 1

// copy merged courses back to original array

For each index in mergedCourses

Set courses [low + index] = mergedCourses[index]

}

**Course Search(id) {**

Call Sort()

// use a binary search

Declare Integer Mid

Declare Integer Low

Declare Integer High

While High >= Low

Mid = (High + Low) / 2 // find mid index

If sortedVector [mid].Id < id

Set Low = Mid + 1 // raise low index up

Else If sortedVector [mid].Id > key

Set High = Mid – 1 // lower high index down

Else

Return sortedVector[mid] // found

Return Default [**Course Class Vector]** () // not found

}

**GetSize() {**

Return Size

}

**PrintAll() {**

Call Sort()

For each Course in courses

Output course.Id

Output course.Title

}

**PrintCourse(id) {**

Set Course = Search(id)

If Course not equal Default Course

Output course.Id

Output course.Title

For each prerequisite in course.Prerequisites

Output prerequisite.Id

Output prerequisite.Title

Else

Output “Course not found: $id”

}

**Hashtable Class**

DeclareInteger Size

Declare Boolean Sorted

Allocate HashNodes[] nodes = New HashNodes[100]

**Integer HashId(id)**

Return id modulus nodes.size

**Insert(course) {**

Declare Integer Hash

Set Hash = Call HashId (course.Id)

Set currentNode = nodes[Hash]

If currentNode.Course.Id not equal default course().Id

Set nodes[Hash].Course = course

Else Inserting into node’s linked list

Allocate a new Node as newNode

Set newNode.Course = course

Set newNode.next = nullptr

If first node in linked list

Set currentNode.next = newNode

Set nodes[Hash] = currentNode // must update copy in array

else

// inserting at beginning of linked list

Set newNode.next = currentNode.next

Set currentNode.next = newNode

Set Size = Size + 1

If Sorted

Set Sorted = False

}

[**Course Class Hash Table**] [] **Sort() {**

If Not Sorted

Declare Course courses[Size]

Declare Integer index;

Declare HashNode\* chainNode

Set Index = 0

For each node in nodes

Set courses[index] = node.Course

Set Index = Index + 1

Set chainNode = node.next

While chainNode not equal nullptr

Set courses[index] = chainNode.Course

Set Index = Index + 1

Set chainNode = chainNode.next

Call MergeSort(courses, 0, Size – 1) // courses by reference

Return courses

}

**MergeSort(courses[] by ref, low, high) {**

Declare Integer mid

If (low < high) {

Set mid = (low + high) / 2 // find mid index point

MergeSort(courses, low, mid) // recursive sort left half

MergeSort(courses, mid + 1, high) // recursive sort right half

Merge(courses, low, mid, high)

}

}

**Merge(courses[] by ref, low, mid, high) {**

Declare Integer LeftPosition

Declare Integer RightPosition

Declare Integer MergePosition

Declare Integer MergeSize

Set MergeSize = high – low + 1

Set LeftPosition = low

Set RightPosition = mid + 1

Set MergePosition = 0

Declare Course mergedCourses[MergeSize]

// add smallest course id to mergedCourses array from either left or right partition

While LeftPosition <= mid And RightPosition <= high

If courses[LeftPosition].Id <= courses[RightPosition].Id

Set mergedCourses[MergePosition] = courses[LeftPosition]

Set LeftPosition = LeftPosition + 1

Else

Set mergedCourses[MergePosition] = courses[RightPosition]

Set RightPosition = RightPosition + 1

Set MergePosition = MergePosition + 1

// add remaining left partition entries, if any

While LeftPosition <= mid

Set mergedCourses[MergePosition] = courses[LeftPosition]

Set LeftPosition = LeftPosition + 1

Set MergePosition = MergePosition + 1

// add remaining right partition entries, if any

While RightPosition <= high

Set mergedCourses[MergePosition] = courses[RightPosition]

Set RightPosition = RightPosition + 1

Set MergePosition = MergePosition + 1

// copy merged courses back to original array

For each index in mergedCourses

Set courses [low + index] = mergedCourses[index]

}

**Course Search(id) {**

Declare [**Course Class Hash Table**] [] sortedCourses

Set sortedCourses = Call Sort()

// use a binary search

Declare Integer Mid

Declare Integer Low

Declare Integer High

While High >= Low

Mid = (High + Low) / 2 // find mid index

If sortedVector [mid].Id < id

Set Low = Mid + 1 // raise low index up

Else If sortedVector [mid].Id > key

Set High = Mid – 1 // lower high index down

Else

Return sortedVector[mid] // found

Return Default Course() // not found

}

**GetSize() {**

Return Size

}

**PrintAll() {**

Declare [**Course Class Hash Table**] [] sortedCourses

Set sortedCourses = Call Sort()

For each Course in courses

Output course.Id

Output course.Title

}

**PrintCourse(id) {**

Set Course = Search(id)

If Course not equal Default Course

Output course.Id

Output course.Title

For each prerequisite in course.Prerequisites

Output prerequisite.Id

Output prerequisite.Title

Else

Output “Course not found: $id”

}

**BinarySearchTree Class**

Integer Size

TreeNode\* root;

**BinarySearchTree**() {

Set root = nullptr;

}

**Insert(course) { N**

If very first node inserted

Set root node = allocate new TreeNode(course)

Set Size = Size + 1

Else

Call addNode(root, course) // recursive

**}**

**addNode (node, course) {**

If course.Id < node.Course.Id

If node.left Pointer Is Unused

Set node.left = New TreeNode (course)

Set Size = Size + 1

Else

Call addNode(node.left, course) // recursive

Else course.Id >= node.Course.Id

If node.right Pointer Is Unused

Set node.right = New TreeNode (course)

Set Size = Size + 1

Else

Call addNode (node.right, course) // recursive

**}**

**PrintAll() {**

Call InOrder(root)

**}**

**InOrder(currentNode) {**

If currentNode not equal nullptr

Call InOrder(currentNode.left) // recursive

Output currentNode.Course.Id

Output currentNode.Course.Title

Call InOrder(currentNode.right) // recursive

**}**

**[Course Class Tree] Search (id) {**

Set currentNode = root

While currentNode not equal nullptr

If currentNode.Course.Id is equal to id

Return currentNode.Course

Else If id < currentNode.Course.Id

Set currentNode = currentNode.left

Else

Set currentNode = currentNode.right

Return **[Course Class Tree]()** with default values // not found

**}**

**PrintCourse(id) {**

Set Course = Search(id)

If Course not equal Default Course

Output course.Id

Output course.Title

For each prerequisite in course.Prerequisites

Output prerequisite.Id

Output prerequisite.Title

Else

Output “Course not found: $id”

}

Code Section

**Program Start Vector**

Declare Vector<Course> courses

Call MenuLoop(courses) // by reference

**Program Start Hash Table**

Declare Hashtable<Course> courses

Call MenuLoop (courses) // by reference

**Program Start Tree**

Declare BinarySearchTree<Course> courses

Call MenuLoop(courses) // by reference

**MenuLoop (courses by ref)**

Declare command

Set command = 0

While command Not Equal to 9

Call Print Menu

Read Input into command

Call HandleCommand with command by value and courses by reference

**Print Menu**

Output “1. Load Data Structure”

Output “2. Print Course List”

Output “3. Print Course”

Output “9. Exit Program”

Output “Please enter number of command to perform:”

**HandleCommand (command, courses by ref)**

If command equals 1

Call LoadDataFile(courses) // by reference

Else If command equals 2

Call courses.PrintAll()

Else If command equals 3

Output “Enter course id to print”

Get courseId from input

Call courses.PrintCourse(courseId)

Else If command equals 9

Output “Exiting program.”

**LoadDataFile (courses by ref)**

Open File with File Path

Set lineNumber = 0

While Not End of File

Create Course Object // can vary with vector, hash table, or binary search tree

Read Line from File

Set lineNumber = lineNumber + 1

Split line by comma into array of tokens

If tokens.size < 2

Output “Error: Line format error, expected 2 or more tokens, got $size tokens, line number: $lineNumber”

Continue While

Set course.Id = Token[0]

Set lookupCourse = Call courses.Search(course.Id)

If lookupCourse is not null

Print “Warning: Duplicate course detected, line number: $lineNumber”

Continue While

Else

Set course.Title = token[1]

For each token after index 1

Declare prerequisite

Set prerequisite = token

Set Course = Call courses.Search(prerequisite.Id)

If Course Is Null

Output “Error: Prerequisite not found, course: $course.Id, prerequisite: $prerequisite.Id”

Continue While

Else

Call courses.Prerequisites.Insert(prerequisite)

Call courses.Insert(course)

# **Big-O Analysis**

The analysis focuses on the worst-case scenario. The requirements are to treat most lines as a cost of 1 unless calling a function. When calling a function, its total cost is added up and reported in the calling line. There is a complication in handling the different data types. Each data type can have a different cost. The methodology used is to take a subtotal without the Insert and Search functions. Then total each data type's totals into separate subtotals, and then report the combined subtotals for each datatype. When an algorithm's cost complexity was challenging to figure out, the Big-O from the textbook was substituted instead. The following algorithms from the textbook are used, Binary Search and Merge Sort. The algorithms are selected for their efficiency out of the choices available. A Binary Search has a runtime Big-O complexity of O(Log N) (CS 300: Data Structures and Algorithms, p. Section 3.8), and a Merge Sort has a complexity of O(N Log N) (CS 300: Data Structures and Algorithms, p. Section 3.19). The substitution is expected to lower the accuracy of the subtotals. However, it will not affect the overall accuracy of the Big-O analysis since it is common practice to ignore slight differences and instead report the most complex runtime factors.

# **LoadDataFile**

| **Code** | | **Line Cost** | | **Vector** | **Hash table** | **Binary Search Tree** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Open File with File Path | | 1 | |  |  |  | 1 | 1 |
| Set lineNumber = 0 | | 1 | |  |  |  | 1 | 1 |
| While Not End of File | | 1 | |  |  |  | n | n |
| Create Course Object | | 1 | |  |  |  | n | n |
| Read Line from File | | 1 | |  |  |  | n | n |
| Set lineNumber = lineNumber + 1 | | 1 | |  |  |  | n | n |
| Split line by comma into array of tokens | | 1 | |  |  |  | n | n |
| If tokens.size < 2 | | 1 | |  |  |  | n | n |
| Output "Error: Line format error… | | 1 | |  |  |  | n | n |
| Continue While | | 1 | |  |  |  | n | n |
| Set course.Id = Token[0] | | 1 | |  |  |  | n | n |
| Set lookupCourse = Call courses.Search(course.Id) | |  | | 2+(N+1) Log N | 3+(n+2) Log n | n | n |  |
| If lookupCourse is not null | | 1 | |  |  |  | n | n |
| Print "Warning:… | | 1 | |  |  |  | n | n |
| Continue While | | 1 | |  |  |  | n | n |
| Else | | 1 | |  |  |  | n | n |
| Set course.Title = token[1] | | 1 | |  |  |  | n | n |
| For each token after index 1 | | 1 | |  |  |  | n\*m | n\*m |
| Declare prerequisite | | 1 | |  |  |  | n\*m | n\*m |
| Set prerequisite = token | | 1 | |  |  |  | n\*m | n\*m |
| Set Course = Call courses.Search(prerequisite.Id) | |  | | 2+Log N | 3+(n+2) Log n | n | n\*m |  |
| If Course Is Null | | 1 | |  |  |  | n\*m | n\*m |
| Output “Error: … | | 1 | |  |  |  | n\*m | n\*m |
| Continue While | | 1 | |  |  |  | n\*m | n\*m |
| Else | | 1 | |  |  |  | n\*m | n\*m |
| Call courses.P...Insert | |  | | 8 + 2N | 15 | n | n\*m |  |
| Call courses.Insert | |  | | 8 + 2N | 15 | n | n |  |
|  | | **Datatype Sub**  **total** | | 10n+8mn+n Log n+mn Log n+2n^2 +n^2 Log n | 18n+2n Log n+18mn+mn Log n+n^2 Log n | 2n^2+2nm |  |  |
|  |  |  | **Subtotal Cost** | | | | | 2+21n+7nm |
|  |  |  | **Vector Total Cost** | | | | | 2+31n+15mn+n Log n+mn Log n+2n^2+n^2 Log n |
|  |  |  | **Hash Table Total Cost** | | | | | 2+39n+25mn+2n Log n+mn Log n+n^2 Log n |
|  |  |  | **Binary Search Tree Total Cost** | | | | | 2+21n+9nm+2n^2 |
|  |  |  | **Vector Runtime** | | | | | O(n+mn+n Log n+mn Log n+n^2 +n^2 Log n) |
|  |  |  | **Hash Table Runtime** | | | | | O(n+mn+n\*Log n+mn\*Log n+n^2\*Log n+mn^2\*Log n) |
|  |  |  | **Binary Search Tree Runtime** | | | | | O(n+nm+n^2) |

The "m" in the notation above represents the number of prerequisites per course. However, it is not anticipated that "m" will be a significantly large number and could be considered a constant for runtime analysis. For example, if the average number of prerequisites in a file were 3, taking 3 \* N would still result in an O (N) runtime evaluation. Since there is no data file to evaluate, the "m" variable is not optimized in order to maintain a worst-case analysis.

**Recommendation**

The requirement was to analyze the load data function's complexity and recommend a data type based on its complexity. However, the load function is not the totality of the program and is not central to its intended use. Its intended use allows the user to look up and print out courses and their prerequisites, which are expected to be performed many times. Loading data will happen one time. Therefore, sort, lookup, and retrieval are the essential functions of the user's needs. There is no need to recommend one data type over another for the load scenario.

The simplicity of the Binary Search Tree when it comes to lookup and retrieval of data would be ideal in this scenario. However, examining the sample data in the table below shows that the course identifiers are nearly in order. An ordered file will create an unbalanced tree and turn most searches into an O (n) operation. If the order in the data file could be randomized, this would lead to a balanced tree that would search in an O (Log N) operation. However, that would present a problem for checking whether the prerequisite exists during loading, which is a requirement. Therefore, a Binary Search Tree data type is not recommended.

Sample Record Format

CSCI100,Introduction to Computer Science   
CSCI101,Introduction to Programming in C++,CSCI100   
CSCI200,Data Structures,CSCI101   
MATH201,Discrete Mathematics   
CSCI300,Introduction to Algorithms,CSCI200,MATH201   
CSCI301,Advanced Programming in C++,CSCI101   
CSCI350,Operating Systems,CSCI300   
CSCI400,Large Software Development,CSCI301,CSCI350

# By its design, the hash table data type cannot be sorted in place. After analyzing the hash table implementation, the design is not optimal. The implementation opted to copy the table to an array, Merge Sort the array, and Binary Search the array. After consideration, it would be more efficient to turn it into a straight linear search of each vector node and linked chain node. This would turn it into an O(N) operation on par with the analysis of the Binary Search Tree above. However, a complete traversal of the Hash table would still require the copy to an array, a merge sort of the array, and then a traversal for printing out the entire table or O(2N + N Log N). Which is still higher than an O(N) operation from a Binary Search Tree. A hash table is not recommended, as it is not better than a Binary Search Tree.

# The vector data type's main advantage is that it will be in a sorted state after loading is finished, ready for user interactions. A Binary Search on a vector will take O(Log N) operations, and a complete traversal for printing out the entire vector will be O(N). This gives it an advantage for lookups and puts it on par with a Binary Search Tree data type for complete traversals. It may take longer to load the file than a Binary Search Tree due to the design to expand the vector should it run out of room. Also, the need to Merge Sort the vector after every record insert will slow down the loading of the data file. However, given that loading only needs to occur once and lookups and traversals will be performed more often, the vector datatype is recommended.

# **References**

*CS 300: Data Structures and Algorithms*. (2019, Feb). (Zyante Inc.) Retrieved from zybooks.com: https://learn.zybooks.com/zybook/SNHUCS300v1

*vector*. (n.d.). Retrieved from cplusplus.com: https://www.cplusplus.com/reference/vector/vector/