Project One

Jonathan Brugh

CS-300 DSA: Analysis and Design

Dr. Thoma

April 13, 2025

**Vector CSV parser pseudocode**

FUNCTION loadCSV(csvPath)

OPEN csvPath for READ

IF file doesn’t open

PRINT error

RETURN vector

WHILE not EOF

DECLARE 2D vector

GET line of input from file

PUSH input to vector row 1

IF input count > 2

PUSH input to vector line row 2

IF size of vector row 1 < 2

PRINT error

RETURN vector

RETURN vector

end

FUNCTION 2D vector<string> loadCourses(csvPath)

DECLARE 2D reference vector

CALL loadCSV(csvPath)

reference vector = returned vector

CALL loadCourses(reference vector)

End

**Pseudocode for creating course objects**

DECLARE Structure Course

Course Id

Course title

vector Course preReqs

end

FUNCTION vector<Courses> loadCourses(reference vector)

DECLARE 2D vector courses

FOR reference vector size

DECLARE Course course

SET Course Id

SET Course title

FOR prereqsSize

SET Course preReqs

APPEND Course to courses

RETURN courses

End

**Pseudocode for search**

FUNCTION displayCourse(Vector courses, String courseNumber)

FOR all courses

IF course == courseNumber

PRINT course information

FOR each prerequisite of course

PRINT the prerequisite course information

end

**Hash table CSV parser pseudocode**

FUNCTION loadCourses(csvPath, hashTable)

CALL Parser(csvPath):

Parser: OPEN csvPath for READ

DECLARE 2D vector line

Parser: IF file doesn’t open

PRINT error

RETURN vector line

Parser: WHILE not EOF

GET line of input from file

PUSH input to vector line row 1

IF input count > 2

PUSH input to vector line row 2

IF size of vector line row 1 < 2

PRINT error

RETURN vector line

DECLARE Course structure

Course Id

Course title

Course prerequisites

Parse vector line to Course

DECLARE vector hashTable

FOR vector line size

CALL Insert(Course)

Insert: CREATE hash key from Course Id

PUSH Course to hashTable at hash key

IF hashTable at index not empty

Prepend Course to index

FOR size of hashTable

FOR size of prerequisites at hashTable index

IF Course prerequisite not equal to any Course Title

PRINT error

End

**Pseudocode for creating course objects**

Structure Course

DECLARE course Id

DECLARE course title

DECLARE vector prerequisites

end

FUNCTION vector loadCourses(string csvPath)

DECLARE vector courses

DECLARE file ← Parser(csvPath)

WHILE file not EOF

DECLARE Course course

SET course course Id

SET course course Name

SET course prerequisites

APPEND course to courses

RETURN courses

end

**Pseudocode for hash table search**

FUNCTION displayCourse(HashTable courses, String courseNumber)

FOR courses size

IF course Id equals courseNumber

PRINT course information

FOR each prerequisite of course

PRINT the prerequisite course information

end

**BST CSV parser pseudocode**

FUNCTION loadCSV(csvPath)

OPEN csvPath for READ

IF file doesn’t open

PRINT error

RETURN vector

WHILE not EOF

DECLARE 2D vector

GET line of input from file

PUSH input to vector row 1

IF input count > 2

PUSH input to vector line row 2

IF size of vector row 1 < 2

PRINT error

RETURN vector

RETURN vector

end

FUNCTION addNode(current Node, Course)

IF Course number < current Node Course number

IF current Node left == null

CREATE new Node with Course

current Node left pointer = new Node

ELSE

current Node = current Node left pointer

CALL addNode(current Node, Course)

ELSE

IF current Node right == null

CREATE new Node with Course

current Node right pointer = new Node

ELSE

current Node = current Node right pointer

CALL addNode(current Node, Course)

end

FUNCTION Insert(Course)

IF root == null

CREATE new node with Course

root = new node

ELSE

CALL addNode(root, Course number)

end

FUNCTION loadCourses(csvPath, BST)

DECLARE Course structure

Course Id

Course title

vector Course prerequisites

DECLARE Node structure

DECLARE Course

DECLARE Node left pointer

DECLARE Node right pointer

Node()

INITIALIZE left pointer = null

INITIALIZE right pointer = null

Node(Course) extends Node()

DECLARE root node

INITIALIZE root = null

CALL loadCSV(csvPath)

INITIALIZE Course = returned vector

FOR vector row 2 > 0

Prerequisites = row 2 vector

FOR returned vector size

CALL Insert(Course)

end

**Pseudocode for creating course objects**

DECLARE Course structure

Course Id

Course title

vector Course prerequisites

DECLARE Node structure

DECLARE Course

DECLARE Node left pointer

DECLARE Node right pointer

Node()

INITIALIZE left pointer = null

INITIALIZE right pointer = null

Node(Course) extends Node()

INITIALIZE Course = parsed Course

**Pseudocode for BST search**

FUNCTION displayInOrder(current Node)

IF Node != null

displayInOrder (current Node left pointer)

PRINT Node data

displayInOrder (current Node right pointer)

end

**Pseudocode for user menu**

DECLARE data structure

INITIALIZE data structure

DECLARE Course

choice = 0

WHILE choice != 9

DISPLAY user menu

1. Load data into courses

2. Display all courses alphanumerically

3. Display course title and any prerequisite for any individual course

9. Exit

READ in user choice

choice 1:

LOAD courses into data structure

choice 2:

PRINT all courses alphanumerically

choice 3:

PRINT individual course and any prereqs associated with course

choice 9:

PRINT goodbye

exit

end

**Pseudocode for sorting data structures prior to print**

**Vector data structure**

FUNCTION Merge(dataStruct, leftPos, rightPos, structSize)

DECLARE tempDataStruct = dataStruct

DECLARE and INITIALIZE position variables

WHILE leftPos <= rightPos & rightPos <= vectorSize

IF leftPos < rightPos

tempDataStruct [mergePos] = dataStruct[leftPos]

++ leftPos

ELSE

tempDataStruct [mergePos] = dataStruct[rightPos]

++ rightPos

++ mergePos

WHILE leftPos <= rightPos

++ leftPos

++mergePos

WHILE rightPos <= structSize

++ rightPos

++mergePos

FOR tempDataStruct size

COPY tempDataStruct to dataStruct

FUNCTION MergeSort(dataStruct, structBegin, structSize)

FIND midpoint of parsed dataStruct

RECURSIVELY sort partitions to left and right of midpoint

MERGE partitions back into dataStruct

CALL displayCourses() for printing

**Hash table sort**

Hash tables are not sortable data structures by nature.

**BST sort**

Binary search trees are sorted as they are built. See “displayInOrder” function above for printing.

**Evaluations**

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **FUNCTION loadCSV(csvPath)** |  |  |  |
| OPEN csvPath for READ | 1 | 1 | 1 |
| IF file doesn’t open | 1 | 1 | 1 |
| PRINT error | 1 | 1 | 1 |
| RETURN vector | 1 | 1 | 1 |
| WHILE not EOF | 1 | n | n |
| DECLARE 2D vector | 1 | n | n |
| GET line of input from file | 1 | n | n |
| PUSH input to vector row 1 | 1 | n | n |
| IF input count > 2 | 1 | n | n |
| PUSH input to vector line row 2 | 1 | n | n |
| IF size of vector row 1 < 2 | 1 | n | n |
| PRINT error | 1 | n | n |
| RETURN vector | 1 | n | n |
| RETURN vector | 1 | 1 | 1 |
| **FUNCTION 2D vector<string> loadCourses(csvPath)** |  |  |  |
| DECLARE 2D reference vector | 1 | 1 | 1 |
| CALL loadCSV(csvPath) | 1 | 1 | 1 |
| reference vector = returned vector | 1 | 1 | 1 |
| CALL loadCourses(reference vector) | 1 | 1 | 1 |
| **FUNCTION vector<Courses> loadCourses(reference vector)** |  |  |  |
| DECLARE 2D vector courses | 1 | 1 | 1 |
| FOR reference vector size | 1 | n | n |
| DECLARE Course course | 1 | n | n |
| SET Course Id | 1 | n | n |
| SET Course title | 1 | n | n |
| FOR prereqsSize | 1 | n | n |
| IF Course prereqs not equal to any Course Title | 1 | n\*m | n\*m |
| PRINT error | 1 | n\*m | n\*m |
| SET Course preReqs | 1 | n | n |
| APPEND Course to courses | 1 | n | n |
| RETURN courses | 1 | 1 | 1 |
| **Total Cost** |  |  | 2nm + 16n + 11 |
| **Worst case Runtime** | | | O(n\*m+n) = O(nm) |

Function call costs were accounted for by expanding the functions.

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **FUNCTION loadCourses(csvPath, hashTable)** |  |  |  |
| CALL Parser(csvPath) | 1 | 1 | 1 |
| OPEN csvPath for READ | 1 | 1 | 1 |
| DECLARE 2D vector line | 1 | 1 | 1 |
| IF file doesn’t open | 1 | 1 | 1 |
| PRINT error | 1 | 1 | 1 |
| RETURN vector line | 1 | 1 | 1 |
| WHILE not EOF | 1 | n | n |
| GET line of input from file | 1 | n | n |
| PUSH input to vector line row 1 | 1 | n | n |
| IF input count > 2 | 1 | n | n |
| PUSH input to vector line row 2 | 1 | n | n |
| IF size of vector line row 1 < 2 | 1 | n | n |
| PRINT error | 1 | n | n |
| RETURN vector line | 1 | n | n |
| DECLARE Course structure | 1 | n | n |
| Course Id | 1 | n | n |
| Course title | 1 | n | n |
| Course prerequisites | 1 | n | n |
| Parse vector line to Course | 1 | n | n |
| DECLARE vector hashTable | 1 | 1 | 1 |
| FOR vector line size | 1 | n | n |
| CALL Insert(Course) | 1 | n | n |
| CREATE hash key from Course Id | 1 | n | n |
| PUSH Course to hashTable at hash key | 1 | n | n |
| IF hashTable at index not empty | 1 | n | n |
| Prepend Course to index | 1 | n | n |
| FOR size of hashTable | 1 | n | n |
| FOR size of prerequisites at hashTable index | 1 | n | n |
| IF Course prerequisite not equal to any Course Title | 1 | n\*m | n\*m |
| PRINT error | 1 | n\*m | n\*m |
| **FUNCTION vector loadCourses(string csvPath)** |  |  |  |
| DECLARE vector courses | 1 | 1 | 1 |
| DECLARE file ← Parser(csvPath) | 1 | 1 | 1 |
| WHILE file not EOF | 1 | n | n |
| DECLARE Course course | 1 | n | n |
| SET course course Id | 1 | n | n |
| SET course course Name | 1 | n | n |
| SET course prerequisites | 1 | n | n |
| APPEND course to courses | 1 | n | n |
| RETURN courses | 1 | 1 | 1 |
| **Total Cost** |  |  | 2nm + 27n + 10 |
| **Worst case Runtime** | | | O(n\*m+n) = O(nm) |

Function call costs were accounted for by expanding the functions.

|  |  |  |  |
| --- | --- | --- | --- |
| **BST Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| **FUNCTION loadCSV(csvPath)** |  |  |  |
| OPEN csvPath for READ | 1 | 1 | 1 |
| IF file doesn’t open | 1 | 1 | 1 |
| PRINT error | 1 | 1 | 1 |
| RETURN vector | 1 | 1 | 1 |
| WHILE not EOF | 1 | n | n |
| DECLARE 2D vector | 1 | n | n |
| GET line of input from file | 1 | n | n |
| PUSH input to vector row 1 | 1 | n | n |
| IF input count > 2 | 1 | n | n |
| PUSH input to vector line row 2 | 1 | n | n |
| IF size of vector row 1 < 2 | 1 | n | n |
| PRINT error | 1 | n | n |
| RETURN vector | 1 | n | n |
| RETURN vector | 1 | 1 | 1 |
| **FUNCTION addNode(current Node, Course)** |  |  |  |
| IF Course number < current Node Course number | 1 | n | n |
| IF current Node left == null | 1 | n | n |
| CREATE new Node with Course | 1 | n | n |
| current Node left pointer = new Node | 1 | n | n |
| ELSE | 1 | n | n |
| current Node = current Node left pointer | 1 | n | n |
| CALL addNode(current Node, Course) | 1 | n | n |
| ELSE | 1 | n | n |
| IF current Node right == null | 1 | n | n |
| CREATE new Node with Course | 1 | n | n |
| current Node right pointer = new Node | 1 | n | n |
| ELSE | 1 | n | n |
| current Node = current Node right pointer | 1 | n | n |
| CALL addNode(current Node, Course) | 1 | n | n |
| **FUNCTION Insert(Course)** |  |  |  |
| IF root == null | 1 | 1 | 1 |
| CREATE new node with Course | 1 | 1 | 1 |
| root = new node | 1 | 1 | 1 |
| ELSE | 1 | 1 | 1 |
| CALL addNode(root, Course number) | 1 | n | n |
| **FUNCTION loadCourses(csvPath, BST)** |  |  |  |
| DECLARE Course structure | 1 | n | n |
| Course Id | 1 | n | n |
| Course title | 1 | n | n |
| vector Course prerequisites | 1 | n | n |
| DECLARE Node structure | 1 | n | n |
| DECLARE Course | 1 | n | n |
| DECLARE Node left pointer | 1 | n | n |
| DECLARE Node right pointer | 1 | n | n |
| Node() | 1 | n | n |
| INITIALIZE left pointer = null | 1 | n | n |
| INITIALIZE right pointer = null | 1 | n | n |
| Node(Course) extends Node() | 1 | n | n |
| DECLARE root node | 1 | 1 | 1 |
| INITIALIZE root = null | 1 | 1 | 1 |
| CALL loadCSV(csvPath) | 1 | 1 | 1 |
| INITIALIZE Course = returned vector | 1 | n | n |
| FOR vector row 2 > 0 | 1 | n | n |
| Prerequisites = row 2 vector | 1 | n\*m | n\*m |
| IF Course prerequisite not equal to any Course Title | 1 | n\*m | n\*m |
| PRINT error | 1 | n\*m | n\*m |
| FOR returned vector size | 1 | n | n |
| CALL Insert(Course) | 1 | n | n |
| **Total Cost** |  |  | 3nm+40n+12 |
| **Worst case Runtime** | | | O(nm+n) = O(nm) |

Function call costs were accounted for by expanding the functions.

**Advantages and disadvantages of the selected data structures**

Vectors are dynamic arrays that manage memory for the implementer. Elements can be accessed, appended, and deleted from the end of the vector in linear time. However, all other functions are performed with a time complexity of O(n). For small vectors, this may still perform well, but after a certain size, the benefits diminish. Vectors require sorting to optimize certain functions such as searching and deleting but time complexity still remains O(n). Average time complexity does not reduce from O(n). In this scenario, the vector’s worst-case time complexity is O(n\*m) due to the assumption that all courses have prerequisites, while if no courses had prerequisites, the complexity reduces to O(n).

A hash-table is a data structure that has another data structure extending off of each index. Elements can be added and removed in worst-case O(n) time due to the hash value needing to be compared before the secondary data structure is acted upon. Searching a hash table requires O(n) time at worst-case due to each element or node requiring comparison but only the elements at that hash value. Hash tables are a good way to store and perform actions on a large data set when the average time complexity is expected. The average time complexity is O(log(n)) for a hash table due to data expecting to be sufficiently randomized. This average complexity and better can be achieved by using a very efficient hash function. This scenario has a worst-case time complexity of O(n\*m) using a hash table with chaining, but if using a highly efficient hash function that minimizes or eliminates collisions, time complexity can be reduced to O(log(n)) or even close to O(1), respectively.

A binary search tree is a tree that extends from a root node that has at most two children and continues the pattern as the tree is grown. The root’s left node contains values that are less than it, and the right node, values that are less. Since the tree is broken down into left or right at each tree level, the time to find, insert, delete, or access a particular node is fractionated causing the average time complexity being O(log(n)) for all these functions. However, if the data is skewed and since the BST is not a self-balancing tree, the time complexity can increase to a worst-case of O(n). In this scenario, the BST worst-case time complexity was O(n\*m) but only due to the assumption that every course had prerequisites and only during the insertion phase with the use of a vector to hold the prerequisites. If a vector was exchanged for another BST, time complexity could be reduced to O(n) at worst-case. The average time complexity for a BST is O(log(n)) if the data is sufficiently randomized.

I intend to use a BST in my code due to its average time complexity being O(log(n)). I would like to use a hash table since it has the capability of reducing the time complexity to essentially O(1), but I need more experience to achieve this. A BST is a good option as it is efficient at all necessary functions required to be performed on the data set given. A better option would be a self-balancing tree which would eliminate the chance of the incoming data skewing the tree. Compared to the hash table and the vector data structures, the BST increases the odds of the average time complexity being O(log(n)).