Austin Frey

Professor Ling

CS-300

December 4, 2022

Project One – Pseudocode, Evaluation, & Recommendations

# **Pseudocode**

## *Vector*

1. **File opening, reading, parsing, and formatting**

Function Parser

Pass in: file name (file), the data type (type), and the separator character (sep)

Initialize line variable for reading in data and Vector list

IF type is an eFILE

Create input file stream for file

IF file is open

WHILE input file stream does not have errors

Read line

IF line is not empty

Push line to list

END IF

END WHILE

Close file

IF file is empty

Throw error

END IF

Call: parseHeader() & parseContent()

ELSE Throw error

END IF ELSE

ELSE

Create input stream for user-entered data

WHILE there are lines to process

IF line is not empty

Push line to list

END IF

END WHILE

IF input size is 0

Throw error

END IF

Call: parseContent()

END IF ELSE

Pass out: Parser object

End function

Function parseHeader // Potentially redundant function

Pass in: None

Create stream for each line to be parsed

Create item variable for storing header names

Create course object called header to store header lines

WHILE Not at the end of the line

Add item to header course

END WHILE

Pass out: None

End function

Function parseContent

Pass in: None

Initialize vector iterator index to beginning of the file + 1 to skip the header

FOR iterator doesn’t equal list end, increment iterator

Initialize courseID,

FOR length of iterator’s line

IF less than two elements in line

Throw error

ELSE

Add each element to array for creating the course object

END IF ELSE

END FOR

Create course object and add it to courses

END FOR

Pass out: None

End function

Function validateCourseInfo

Pass in: Array of courses, prerequisite course ID being tested

FOR each element in array

IF a course ID matches prerequisite

Return true

END IF

END FOR

Return false

Pass out: Boolean representing whether or not the course ID was found

End function

1. **Course objects and data structures**

Struct Course

Pass in: Course ID, course title, and prerequisites

Initialize class variables (ID, title, prerequisites array)

Pass out: Course object

End struct

Function addCourse

Pass in: courseID, courseTitle, prerequisites

Create new course object

Append course object to vector list of courses

Pass out: None

End function

*Note: Adding courses to a vector from a file occurs when parsing the content. Thus, there is no additional method for iterating over the courses and adding them to a vector.*

1. **Search and Print**

Function search

Pass in: courseTitle or courseID, courseList

FOR every course in courseList

IF course matches the courseTitle or courseID searched

Return course

END IF

END FOR

Return empty course

Pass out: Course object

End function

Function printCourse

Pass in: course

Print course ID

Print course title

IF length of course prerequisites is less than or equal to zero

Print that there are no prerequisites

ELSE

FOR each prerequisite in course

Print prerequisites courseID

END FOR

END IF ELSE

Pass out: None

End function

## *Hash table*

1. **File opening, reading, parsing, and formatting**

Function Parser

Pass in: file name (file), the data type (type), and the separator character (sep)

Initialize line variable for reading in data and Vector list

IF type is an eFILE

Create input file stream for file

IF file is open

WHILE input file stream does not have errors

Read line

IF line is not empty

Push line to list

END IF

END WHILE

Close file

IF file is empty

Throw error

END IF

Call: parseHeader() & parseContent()

ELSE Throw error

END IF ELSE

ELSE

Create input stream for user-entered data

WHILE there are lines to process

IF line is not empty

Push line to list

END IF

END WHILE

IF input size is 0

Throw error

END IF

Call: parseContent()

END IF ELSE

Pass out: Parser object

End function

Function parseHeader // Potentially redundant function

Pass in: None

Create stream for each line to be parsed

Create item variable for storing header names

Create course object called header to store header lines

WHILE Not at the end of the line

Add item to header course

END WHILE

Pass out: None

End function

Function parseContent

Pass in: None

Initialize vector iterator index to beginning of the file + 1 to skip the header

FOR iterator doesn’t equal file end, increment iterator

Initialize courseID,

FOR length of iterator’s line

IF less than two elements in line

Throw error

ELSE

Add each element to array for creating the course object

END IF ELSE

END FOR

Create course object and add it to courses

END FOR

Pass out: None

End function

Function validateCourseInfo

Pass in: Array of courses, prerequisite course ID being tested

FOR each element in array

IF a course ID matches prerequisite

Return true

END IF

END FOR

Return false

Pass out: Boolean representing whether or not the course ID was found

End function

1. **Course objects and data structures**

Struct Course

Pass in: Course ID, course title, and prerequisites

Initialize class variables (ID, title, prerequisites array, key,)

Pass out: Course object

End struct

Struct Node

Pass in: None OR course OR course and key

Initialize node variables (course, key, next)

Create vector of nodes

Set tableSize to a constant number

Declare hash function

Pass out: Node object

End struct

Function hash

Pass in: key

RETURN key modulo tableSize

Pass out: hash value

End function

Function addCourse

Pass in: course

Create key from course’s courseID using hash function

Create new course object

Call: insertCourse(course)

Pass out: None

End function

Function insertCourse

Pass in: course

Create hash key using hash(courseID)

Retrieve node at respective hash key index

IF node is null

Create new course using courseID and hash key

ELSE IF node’s key is the preset value

Assign course to node

SET node’s next pointer to null

ELSE

WHILE next node is not null

SET node to node’s next node

END WHILE

SET node’s next node to new node using course and hash key

END IF ELSE

Pass out: None

End function

Function printAll

Pass in: None

FOR every node in hash table

IF node’s key is not default value

Print information

CREATE nextNode from node’s next pointer

WHILE nextNode is not null

Print information

SET nextNode to nextNode’s next pointer

END WHILE

END IF

END FOR

Pass out: None

End function

## *Tree*

1. **File opening, reading, parsing, and formatting**

Function Parser

Pass in: file name (file), the data type (type), and the separator character (sep)

Initialize line variable for reading in data and Vector list

IF type is an eFILE

Create input file stream for file

IF file is open

WHILE input file stream does not have errors

Read line

IF line is not empty

Push line to list

END IF

END WHILE

Close file

IF file is empty

Throw error

END IF

Call: parseHeader() & parseContent()

ELSE Throw error

END IF ELSE

ELSE

Create input stream for user-entered data

WHILE there are lines to process

IF line is not empty

Push line to list

END IF

END WHILE

IF input size is 0

Throw error

END IF

Call: parseContent()

END IF ELSE

Pass out: Parser object

End function

Function parseHeader // Potentially redundant function

Pass in: None

Create stream for each line to be parsed

Create item variable for storing header names

Create course object called header to store header lines

WHILE Not at the end of the line

Add item to header course

END WHILE

Pass out: None

End function

Function parseContent

Pass in: None

Initialize vector iterator index to beginning of the file + 1 to skip the header

FOR iterator doesn’t equal file end, increment iterator

Initialize courseID,

FOR length of iterator’s line

IF less than two elements in line

Throw error

ELSE

Add each element to array for creating the course object

END IF ELSE

END FOR

Create course object and add it to courses

END FOR

Pass out: None

End function

Function validateCourseInfo

Pass in: Array of courses, prerequisite course ID being tested

FOR each element in array

IF a course ID matches prerequisite

Return true

END IF

END FOR

Return false

Pass out: Boolean representing whether or not the course ID was found

End function

1. **Course objects and data structures**

Struct BinarySearchTree

Pass in: None

Initialize Node root and set root to null

Initialize int size and set to 0

Pass out: BinarySearchTree object

End struct

Struct Course

Pass in: Course ID, course title, and prerequisites

Initialize class variables (ID, title, prerequisites array, key,)

Pass out: Course object

End struct

Struct Node

Pass in: None OR course OR course, parent

Declare Course, Left Node, Right Node, Parent Node

IF Node(None) // Default constructor

SET Course to empty Course object

SET Left Node, Right Node, Parent Node to null

ELSE IF Node(Course) //Initialize with Course

SET Course to passed in Course

SET parent, left, & right nodes to null

ELSE IF Node(Course, Node) //Initialize with Course and parent

SET Course to passed in Course, parent to passed in parent

SET left & right to null

END IF, ELSE IF, ELSE IF

Pass out: Node object

End struct

*Note: If, else if, else if are actually constructors, but for simplicity’s sake they have been written as conditional statements.*

Function addCourse

Pass in: Current node, Course object

IF BinarySearchTree root is null

SET root to new node(Course)

INCREMENT size

ELSE

IF Course id is less than or equal to current node course id

IF current Node’s left pointer is null

SET current Node’s left pointer to new node(Course, Current node)

INCREMENT size

ELSE

Call: addCourse(current node’s left, course)

END IF ELSE

ELSE //Course id is greater than current node course id

IF current Node’s right pointer is null

SET current Node’s right pointer to new node(Course, current Node)

INCREMENT size

ELSE

Call: addCourse(current node’s right, course)

END IF ELSE

END IF ELSE

END IF ELSE

Pass out: None

End function

1. **Print course information & prerequisites**

Function inOrderPrintAll

Pass in: Node to start at

IF Node is null

RETURN

END IF

IF Node’s left child is not null

Call: inOrderPrintAll(Node’s left child)

END IF

Print course information

Call: inOrderPrintAll(Node’s right child)

End function

## *Menu*

Function parseContent

Pass in: data structure

Initialize vector iterator index to beginning of the file + 1 to skip the header

FOR iterator doesn’t equal file end, increment iterator

Initialize courseID,

FOR length of iterator’s line

IF less than two elements in line

Throw error

ELSE

Add each element to array for creating the course object

END IF ELSE

END FOR

IF data structure’s 1st element is null

Create empty Node object node

ELSE

SET node to data structure’s 1st element

END IF ELSE

Call: data structure’s addCourse(node, course)

END FOR

Pass out: None

End function

*Note: this parseContent function would only apply if the user was choosing the data structure. Otherwise, this function is unnecessary. However, for modularity purposes, this parseContent function is optimal.*

Function menu

Pass in: None

CREATE exit Boolean and set to false

WHILE exit is false

Print menu options

INPUT menu choice

SWITCH menu choice

IF menu choice is load data structure

INPUT file path

INPUT data type

CREATE data structure

CREATE Parser(file path, data type, data separator character)

Call: parseContent(empty data structure)

BREAK

ELSE IF menu choice is print course list

Call: data structure’s print method

BREAK

ELSE IF menu choice is print specific course

INPUT desired course

Call: data structure’s modified print method with desired course id

BREAK

ELSE IF menu choice is exit

SET exit to true

BREAK

ELSE

Print error message

BREAK

END IF, ELSE IF, ELSE IF, ELSE IF, ELSE

END SWITCH

END WHILE

Print exit message

Close program

Pass out: None

End function

## *Ordered Print*

Function sortedPrint

Pass in: data structure

CREATE Vector equal to data structure’s size

ITERATE through data structure and add elements to Vector

Call: quicksort(Vector reference, beginning index (default: 0), end index (default: data structure’s size -1))

ITERATE through Vector and print each element

Pass out: None

End function

Function quicksort

Pass in: Vector reference, beginning index, end index

SET int mid to zero

IF beginning index is greater than or equal to end index

RETURN

END IF

Call: partition(Vector reference, beginning index, end index) and assign returned value to mid

Call: quicksort(Vector reference, beginning index, mid)

Call: quicksort(Vector reference, mid + 1, end index)

Pass out: None

End function

Function partition

Pass in: Vector reference, beginning index, end index

SET low to beginning index and high to end index – 1

SET mid to low + (high-low)/2

SET pivot to Vector’s mid index

SET Boolean done to false

WHILE done is false

WHILE Vector’s low index element is less than the pivot element

INCREMENT low

END WHILE

WHILE Vector’s high index element is greater than the pivot element

DECREMENT high

END WHILE

IF low is greater than or equal to high

SET done to true

ELSE

Call: swap(vector’s element at low, vector’s element at high)

INCREMENT low index

DECREMENT high index

END IF ELSE

END WHILE

RETURN high index

Pass out: The midpoint element between the low and high partitions

End function

# **Evaluation**

## *Run-time Complexity*

**Note**: The program is designed to utilize the same parsing methods for each data structure, so a line-by-line analysis for each data structure is unnecessary. Each data structure also uses the same code for creating course objects as well. The only difference between the data structure’s object creation is within the nodes. Each course is created with a for loop that iterates through the elements separated by commas in the CSV. To avoid confusion between lines in a file and attributes of a course, the tables below represent the number of attributes of a course with *m*.

Since constants are ignored in Big O analyses, we can focus on the for loops within the functions. Reading files in only requires one for loop to iterate through each line, so there will be *n* iterations of the for loop. Creating course objects only requires one for loop to assign attributes to the course, so there will be *m* iterations to create each course object.

### *Vector*

|  |  |
| --- | --- |
| Action | Worst case |
| Reading file | O(n) |
| Creating course objects | O(m) |

### *Hash Table*

|  |  |
| --- | --- |
| Action | Worst case |
| Reading file | O(n) |
| Creating course objects | O(m) |

### *Binary Search Tree*

|  |  |
| --- | --- |
| Action | Worst case |
| Reading file | O(n) |
| Creating course objects | O(m) |

## *Space Complexity*

### *Vector*

|  |  |
| --- | --- |
| Action | Worst case |
| Reading file | O(n) |
| Creating course objects | O(n) |

### *Hash Table*

|  |  |
| --- | --- |
| Action | Worst case |
| Reading file | O(n) |
| Creating course objects | O(n) |

### *Binary Search Tree*

|  |  |
| --- | --- |
| Action | Worst case |
| Reading file | O(n) |
| Creating course objects | O(n) |

## *Structures Advantages/Disadvantages*

### *Vector*

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| * Easy to implement * Easy to visualize * Simple to debug * Access run-time complexity of O(1) for known indexes | * Search, Insert, and Delete run-time complexity of O(n) * May require resizing which can waste memory |

### *Hash Table*

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| * Easy to implement * Easy to visualize * Average run-time complexity for search, insert, and delete is O(1) * More secure than most data structures thanks to hashing functions | * Worst case run-time complexity for search, insert, and delete is O(n) * May require resizing which can waste memory * Not using lists at keys can result in unexpected element placements |

### *Binary Search Tree*

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| * Worst run-time complexity for access, search, insert, and delete is O(log n) * Easily sorted during insertion | * Challenging to visualize and implement * Difficult to debug * Minimum complexity for all functions is O(log n) compared to O(1) for some functions in other structures |

## *Recommendation*

With the information that is currently available to me, I will utilize a binary search tree for the program. Vectors are the least effective choices because they have an average and worst-case run-time complexity of O(n) for all functions except accessing elements. Hash tables provide the best average run-time complexity, O(1), for search, insertion, and deletion. However, the risk of elements being out of place is not worth the benefit of an average run-time complexity of O(1), especially because the worst case run-time complexity is still O(n) for search, insert, and delete functions. Binary search trees have an average run-time complexity of O(log n) for all functions. The worst-case run-time complexity for all functions in a binary search tree is also O(n). However, binary search trees have a unique advantage over hash tables: there’s no need to sort the data structure.

Elements in a hash table or a vector may need additional operations performed, such as sorting, for printing courses in alphanumeric order. This could increase the execution time for the program significantly, depending on the number of courses within a file. Due to the structure of binary search trees, there are no additional operations required. An in-order traversal is all that is needed to print the courses in order. Thus, the program will most likely operate the fastest utilizing binary search trees because they will not introduce additional operations to the program, especially if we judge the optimal data structure by its worst-case run-time complexity. Since the worst-case runtime complexity for all three data structures is O(n), adding sorting operations to print a sorted list of courses is not ideal. Therefore, I will utilize binary search trees as the data structure for the program.