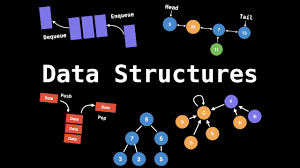
# Data Structures Workspace

## Final Report



April 26, 2022

## *Slippery Rock University*

Adam Schultz [als1054@sru.edu](mailto:als1054@sru.edu)

David Lane [dwl1003@sru.edu](mailto:dwl1003@sru.edu)

Luke Brison [ldb1009@sru.edu](mailto:ldb1009@sru.edu)

Group Number: 4

Class: CPSC 488 Section 01

Problem: Upgrade and Test Data Structures

Table of Contents

[Data Structures Workspace 1](#_Toc101782142)

[Final Report 1](#_Toc101782143)

[*Slippery Rock University* 1](#_Toc101782144)

[Table of Figures 9](#_Toc101782145)

[References 12](#_Toc101782146)

[Completion 12](#_Toc101782147)

[Complete Modules 12](#_Toc101782148)

[Incomplete Modules 13](#_Toc101782149)

[Contribution 13](#_Toc101782150)

[Glossary 13](#_Toc101782151)

[Problem Explanation 15](#_Toc101782152)

[Overview of the System 16](#_Toc101782153)

[Class and Package Descriptions 17](#_Toc101782154)

[edu.sru.thangiah 17](#_Toc101782155)

[edu.sru.thangiah.abstraction 17](#_Toc101782156)

[edu.sru.thangiah.arrays 18](#_Toc101782157)

[edu.sru.thangiah.datastructures 18](#_Toc101782158)

[edu.sru.thangiah.datastructures.generic 19](#_Toc101782159)

[edu.sru.thangiah.datastructures.generic.linkedlist 19](#_Toc101782160)

[edu.sru.thangiah.datastructures.generic.matrix 20](#_Toc101782161)

[edu.sru.thangiah.datastructures.generic.queue 21](#_Toc101782162)

[edu.sru.thangiah.datastructures.generic.stack 21](#_Toc101782163)

[edu.sru.thangiah.datastructures.generic.tree.avltree 22](#_Toc101782164)

[edu.sru.thangiah.datastructures.generic.tree.binarytree 22](#_Toc101782165)

[edu.sru.thangiah.datastructures.generic.tree.heaptree 23](#_Toc101782166)

[edu.sru.thangiah.datastructures.generic.tree.redblacktree 23](#_Toc101782167)

[edu.sru.thangiah.datastructure.graph 24](#_Toc101782168)

[edu.sru.thangiah.datastructures.hashtable 24](#_Toc101782169)

[edu.sru.thangiah.datastructures.linkedlist 25](#_Toc101782170)

[edu.sru.thangiah.datastructures.matrix 26](#_Toc101782171)

[edu.sru.thangiah.datastructures.queue 26](#_Toc101782172)

[edu.sru.thangiah.datastructures.search.minimax 27](#_Toc101782173)

[edu.sru.thangiah.datastructures.stack 27](#_Toc101782174)

[edu.sru.thangiah.datastructures.tree.avltree 28](#_Toc101782175)

[edu.sru.thangiah.datastructures.tree.binarytree 28](#_Toc101782176)

[edu.sru.thangiah.datastructures.tree.generaltree 29](#_Toc101782177)

[edu.sru.thangiah.datastructures.tree.heaptree 29](#_Toc101782178)

[edu.sru.thangiah.datastructures.tree.redblacktree 30](#_Toc101782179)

[edu.sru.thangiah.datastructures.tree.twothreefourtree 30](#_Toc101782180)

[edu.sru.thangiah.datastructures.vector 31](#_Toc101782181)

[edu.sru.thangiah.inheritance 31](#_Toc101782182)

[edu.sru.thangiah.interfaceex 32](#_Toc101782183)

[edu.sru.thangiah.polymorphism 32](#_Toc101782184)

[edu.sru.thangiah.recursion 32](#_Toc101782185)

[edu.sru.thangiah.sorting 33](#_Toc101782186)

[edu.sru.thangiah.vector 34](#_Toc101782187)

[Overview of the Solution 34](#_Toc101782188)

[Path to Solution 35](#_Toc101782189)

[System Requirements 35](#_Toc101782190)

[Software Requirements 35](#_Toc101782191)

[Minimum Hardware Requirements 35](#_Toc101782192)

[Recommended Hardware Requirements 35](#_Toc101782193)

[UML Diagrams 36](#_Toc101782194)

[Use Case Diagrams 36](#_Toc101782195)

[Single Linked List 36](#_Toc101782196)

[Doubly Linked List 37](#_Toc101782197)

[Binary Search Tree 38](#_Toc101782198)

[General Tree 39](#_Toc101782199)

[Heap Tree 40](#_Toc101782200)

[Diagram

Description automatically generated 40](#_Toc101782201)

[Class Diagrams: 41](#_Toc101782202)

[Edu.sru.thangiah 41](#_Toc101782203)

[Graphical user interface, text

Description automatically generated 41](#_Toc101782204)

[Figure 36 Class diagram of edu.sru.thangiah package 41](#_Toc101782205)

[Edu.sru.thangiah.arrays 42](#_Toc101782206)

[Edu.sru.thangiah.datastructures 43](#_Toc101782207)

[Edu.sru.thangiah.datastructures.generic 44](#_Toc101782208)

[Edu.sru.thangiah.datastructures.generic.matrix 45](#_Toc101782209)

[Edu.sru.thangiah.datastructures.generic.queue 46](#_Toc101782210)

[Edu.sru.thangiah.datastructures.generic.stack 47](#_Toc101782211)

[Edu.sru.thangiah.datastructure.generic.linkedlist 48](#_Toc101782212)

[Edu.sru.thangiah.datastructures.generic.linkedlist 50](#_Toc101782213)

[Edu.sru.thangiah.datastructures.tree.binarytree 52](#_Toc101782214)

[edu.sru.thangiah.datastructures.generic.tree.binarytree 53](#_Toc101782215)

[Edu.sru.thangiah.datastructures.tree.generaltree 54](#_Toc101782216)

[edu.sru.thangiah.datastructures.tree.heaptree 57](#_Toc101782217)

[edu.sru.thangiah.datastructures.generic.tree.heaptree 58](#_Toc101782218)

[edu.sru.thangiah.datastructures.generic.tree.avltree 59](#_Toc101782219)

[edu.sru.thangiah.datastructures.tree.avltree 60](#_Toc101782220)

[edu.sru.thangiah.datastructures.tree.redblacktree 61](#_Toc101782221)

[edu.sru.thangiah.datastructures.generic.tree.redblacktree 62](#_Toc101782222)

[edu.sru.thangiah.datastructures.stack 62](#_Toc101782223)

[sru.edu.thangiah.datastructures.queue 63](#_Toc101782224)

[edu.sru.thangiah.datastructurs.matrix 64](#_Toc101782225)

[edu.sru.thangiah.datastructures.search.minimax 65](#_Toc101782226)

[edu.sru.thangiah.datastructures.hashtable 66](#_Toc101782227)

[edu.sru.thangiah.datastructures.vector 67](#_Toc101782228)

[edu.sru.thangiah.datastructures.inheritance 68](#_Toc101782229)

[edu.sru.thangiah.datastructures.interfaceex 69](#_Toc101782230)

[edu.sru.thangiah.datastructures.polymorphism 70](#_Toc101782231)

[edu.sru.thangiah.datastructures.recursion 70](#_Toc101782232)

[edu.sru.thangiah.datastructures.sorting 71](#_Toc101782233)

[Sequence Diagrams: 72](#_Toc101782234)

[In-Order Traversal for any Tree Data Structure 72](#_Toc101782235)

[Pre-Order Traversal for any Tree Data Structure 74](#_Toc101782236)

[Post-Order Traversal for any Tree Data Structure 75](#_Toc101782237)

[Level-Order Traversal for any Tree Data Structure 76](#_Toc101782238)

[A picture containing diagram

Description automatically generated 76](#_Toc101782239)

[Hash Table Add Method (Array) 77](#_Toc101782240)

[Hash Table Add Method (ArrayList) 78](#_Toc101782241)

[Hash Table add method (Vector) 79](#_Toc101782242)

[Minimax Tic Tac Toe 80](#_Toc101782243)

[UML Statechart Diagrams 82](#_Toc101782244)

[In-order Traversal for any Tree Data Structure 82](#_Toc101782245)

[Pre-Order Traversal for any Tree Data Structure 83](#_Toc101782246)

[Post-Order Traversal for any Tree Data Structure 84](#_Toc101782247)

[Level Order Traversal for any Tree Data Structure 85](#_Toc101782248)

[Hash Table Add Method for any Container (Array, ArrayList, Vector) 86](#_Toc101782249)

[Minimax Tic Tac Toe 87](#_Toc101782250)

[UML Activity Diagrams 89](#_Toc101782251)

[In-Order Traversal for any Tree Data Structure 89](#_Toc101782252)

[Pre-Order Traversal for any Tree Data Structure 90](#_Toc101782253)

[Post-Order Traversal for any Tree Data Structure 91](#_Toc101782254)

[Level-Order Traversal for any Tree Data Structure 92](#_Toc101782255)

[Hash Table Add Method for any Container (Array, ArrayList, Vector) 93](#_Toc101782256)

[Minimax Tic Tac Toe Game 94](#_Toc101782257)

[Caveats/Minefields 97](#_Toc101782258)

[Introduction 97](#_Toc101782259)

[Core Data Structures and Arrays 97](#_Toc101782260)

[Keeping Track of Visited Nodes for Graph Algorithms 97](#_Toc101782261)

[Ensuring the Functionality of Core Data Structures First 97](#_Toc101782262)

[Documentation 98](#_Toc101782263)

[Documentation Files 98](#_Toc101782264)

[File Path Names 98](#_Toc101782265)

[JavaDoc 98](#_Toc101782266)

[User Manual 98](#_Toc101782267)

[Config & Install Manual 98](#_Toc101782268)

[Security Manual 98](#_Toc101782269)

[Technical Manual 99](#_Toc101782270)

[Testing Manual 99](#_Toc101782271)

[Code Reusability 99](#_Toc101782272)

[Reusable Classes/Packages/Methods 99](#_Toc101782273)

[edu.sru.thangiah.datastructures 99](#_Toc101782274)

[edu.sru.thangiah.datastructures.generic 99](#_Toc101782275)

[edu.sru.thangiah.datastructures.linkedlist (int/generic) 99](#_Toc101782276)

[edu.sru.thangiah.datastructures.matrix (int/generic) 99](#_Toc101782277)

[edu.sru.thangiah.datastructures.queue (int/generic) 100](#_Toc101782278)

[edu.sru.thangiah.datastructures.stack (int/generic) 100](#_Toc101782279)

[edu.sru.thangiah.datastructures.tree.avltree (int/generic) 100](#_Toc101782280)

[edu.sru.thangiah.datastructures.tree.binarytree (int/generic) 100](#_Toc101782281)

[edu.sru.thangiah.datastructures.tree.heaptree (int/generic) 100](#_Toc101782282)

[edu.sru.thangiah.datastructures.tree.redblacktree (int/generic) 100](#_Toc101782283)

[edu.sru.thangiah.datastructures.twothreefourtree (int) 100](#_Toc101782284)

[edu.sru.thangiah.datastructures.graph (int) 101](#_Toc101782285)

[edu.sru.thangiah.datastructures.hashtable (int) 101](#_Toc101782286)

[bestMove() in Board.java class located in edu.sru.thangiah.datastructures.search.minimax 101](#_Toc101782287)

[Testing 101](#_Toc101782288)

[Human-Computer Testing 101](#_Toc101782289)

[Blackbox Testing 101](#_Toc101782290)

[Whitebox Testing 101](#_Toc101782291)

[Unit Testing 102](#_Toc101782292)

[Integration Testing 102](#_Toc101782293)

[Regression Testing 102](#_Toc101782294)

[Boundary Analysis 102](#_Toc101782295)

[Volume Testing 102](#_Toc101782296)

[Stress Testing 102](#_Toc101782297)

[Missing Resources 102](#_Toc101782298)

[Recovering from Crashes 103](#_Toc101782299)

[Deployment/Maintenance 103](#_Toc101782300)

[Post-Mortem Analysis 103](#_Toc101782301)

# Table of Figures

[Figure 1 Diagram of Beginning OOP Examples to Run 1](#_Toc101782051)

[Figure 2 Diagram of Beginning Abstraction Example to Run 1](#_Toc101782052)

[Figure 3 Diagram of Array Examples to Run 1](#_Toc101782053)

[Figure 4 Diagram of Linked List Examples to Run 1](#_Toc101782054)

[Figure 5 Diagram of Matrix Examples to Run 1](#_Toc101782055)

[Figure 6 Diagram of Queue Examples to Run 1](#_Toc101782056)

[Figure 7 Diagram of Stack Examples to Run 1](#_Toc101782057)

[Figure 8 Diagram of AVL Tree Classes to Run 1](#_Toc101782058)

[Figure 9 Diagram of Binary Tree Classes to Run 1](#_Toc101782059)

[Figure 10 Diagram of Heap Tree Classes to Run 1](#_Toc101782060)

[Figure 11 Diagram of Red Black Tree Classes to Run 1](#_Toc101782061)

[Figure 12 Diagram of Classes to Run for Graph Data Structure 1](#_Toc101782062)

[Figure 13 Diagram of Hash Table (int) Classes to Run 1](#_Toc101782063)

[Figure 14 Diagram of Single and Double Linked List (int) Classes to Run 1](#_Toc101782064)

[Figure 15 Diagram of Matrix (int) Classes to Run 1](#_Toc101782065)

[Figure 16 Diagram of Queue (int) Classes to Run 1](#_Toc101782066)

[Figure 17 Diagram of Minimax (int) Classes to Run 1](#_Toc101782067)

[Figure 18 Diagram of Stack (int) Classes to be Run 1](#_Toc101782068)

[Figure 19 Diagram of AVL Tree (int) Classes to Run 1](#_Toc101782069)

[Figure 20 Diagram of Binary Tree (int) Classes to Run 1](#_Toc101782070)

[Figure 21 Diagram of General Tree (int) Classes to Run 1](#_Toc101782071)

[Figure 22 Diagram of Heap Tree (int) Classes to Run 1](#_Toc101782072)

[Figure 23 Diagram of Red Black Tree (int) Classes to Run 1](#_Toc101782073)

[Figure 24 Diagram for 2-3-4 Tree (int) Classes to Run 1](#_Toc101782074)

[Figure 25 Diagram of Vector (int) Classes to Run 1](#_Toc101782075)

[Figure 26 Diagram of Inheritance Classes to Run 1](#_Toc101782076)

[Figure 27 Diagram of Interface Classes to Run 1](#_Toc101782077)

[Figure 28 Diagram of Polymorphism Classes to Run 1](#_Toc101782078)

[Figure 29 Diagram of Recursion Classes to Run 1](#_Toc101782079)

[Figure 30 Diagram of Sorting Classes to Run 1](#_Toc101782080)

[Figure 31: Use Case Diagram of Single Linked List 1](#_Toc101782081)

[Figure 32: Use Case Diagram of Doubly Linked List 1](#_Toc101782082)

[Figure 33: Use Case Diagram of Binary Search Tree 1](#_Toc101782083)

[Figure 34: Use Case Diagram of General Tree Implementation 1](#_Toc101782084)

[Figure 35 Heap Tree Use Case Diagram 1](#_Toc101782085)

[Figure 36 Class diagram of edu.sru.thangiah package 1](#_Toc101782086)

[Figure 37 Class diagram of edu.sru.thangiah.arrays 1](#_Toc101782087)

[Figure 38 Class diagram of edu.sru.thangiah.datastructures 1](#_Toc101782088)

[Figure 39 Class diagram of edu.sru.thangiah.datastructures.generic 1](#_Toc101782089)

[Figure 40 Class diagram of edu.sru.thangiah.datastructures.generic.matrix 1](#_Toc101782090)

[Figure 41 Class Diagram of edu.sru.thangiah.datastructures.generic.queue 1](#_Toc101782091)

[Figure 42 Class diagram of edu.sru.thangiah.datastructures.generic.stack 1](#_Toc101782092)

[Figure 43: Class Diagram of Int Single Linked List Implementation 1](#_Toc101782093)

[Figure 44: Class Diagram of Doubly Linked List Implementation 1](#_Toc101782094)

[Figure 45: Class Diagram of Generic Single Linked List Implementation 1](#_Toc101782095)

[Figure 46: Class Diagram of Generic Doubly Linked List Implementation 1](#_Toc101782096)

[Figure 47: Class Diagram of Int Binary Tree Implementation 1](#_Toc101782097)

[Figure 48: Class Diagram of Generic Binary Tree Implementation 1](#_Toc101782098)

[Figure 49: Class Diagram of Int General Tree Implementation 1](#_Toc101782099)

[Figure 50: Class Diagram of Heap Tree Implementation 1](#_Toc101782100)

[Figure 51: Class Diagram of Generic Heap Tree Implementation 1](#_Toc101782101)

[Figure 52: Class Diagram of Generic AVL Tree 1](#_Toc101782102)

[Figure 53:Class Diagram of Int AVL Tree 1](#_Toc101782103)

[Figure 54: Class Diagram of Int Red Black Tree Implementation 1](#_Toc101782104)

[Figure 55 Class Diagram of Generic Red Black Tree Implementation 1](#_Toc101782105)

[Figure 56 : Class diagram of int stack implementation 1](#_Toc101782106)

[Figure 57 : Class diagram of int queue implementation 1](#_Toc101782107)

[Figure 58 : Class diagram of int matrix implementation 1](#_Toc101782108)

[Figure 59 : Class diagram of Tic Tac Toe minimax implementation 1](#_Toc101782109)

[Figure 60 : Class diagram of the Vector Data Structure 1](#_Toc101782110)

[Figure 61 : Class diagram of the different examples of inheritance in the inheritance package 1](#_Toc101782111)

[Figure 62 : Class diagram of the interface example 1](#_Toc101782112)

[Figure 63 : Class diagram of the polymorphism example 1](#_Toc101782113)

[Figure 64 : Class diagram of the recursion example 1](#_Toc101782114)

[Figure 65 : Class diagram of the sorting package classes 1](#_Toc101782115)

[Figure 66 : Class diagram of int hash table implementation 1](#_Toc101782116)

[Figure 67 Sequence Diagram of In-Order Traversal for any Tree Data Structure 1](#_Toc101782117)

[Figure 68 Sequence Diagram of Pre-Order Traversal for any Tree Data Structure 1](#_Toc101782118)

[Figure 69 Sequence Diagram of Post Order Traversal for any Tree Data Structure 1](#_Toc101782119)

[Figure 70 Sequence Diagram of Level Order Traversal for any Tree Data Structure 1](#_Toc101782120)

[Figure 71: Sequence Diagram of Add Method for Array Hash Table Implementation 1](#_Toc101782121)

[Figure 72 Sequence Diagram of Hash Table Add Method for ArrayList Hash Table Implementation 1](#_Toc101782122)

[Figure 73 Sequence Diagram of Add Method for Vector Hash Table Implementation 1](#_Toc101782123)

[Figure 74 MiniMax Tic Tac Toe Sequence Diagram 1](#_Toc101782124)

[Figure 75 Sequence Diagram of AVL Tree Addition 1](#_Toc101782125)

[Figure 76 Statechart diagram of In-Order traversal for any Tree Data Structure Implementation 1](#_Toc101782126)

[Figure 77 State Chart Diagram of Pre-Order Traversal for any Tree Data Structure Implementation 1](#_Toc101782127)

[Figure 78 Statechart diagram of Post-Order Traversal for any Tree Data Structure Implementation 1](#_Toc101782128)

[Figure 79 Statechart diagram of Level Order Traversal for any Tree Data Structure Implementation 1](#_Toc101782129)

[Figure 80 Statechart Diagram of Add Method for any Hash Table Implementation (Array, ArrayList, Vector) 1](#_Toc101782130)

[Figure 81 State Chart Diagram of Minimax Tic Tac Toe Game 1](#_Toc101782131)

[Figure 82 Statechart Diagram of AVL Tree Addition 1](#_Toc101782132)

[Figure 83 Activity Diagram of In-Order Tree Traversal for any Tree Data Structure 1](#_Toc101782133)

[Figure 84 Activity Diagram of Pre-Order Traversal for any Tree Data Structure Implementation 1](#_Toc101782134)

[Figure 85 Activity Diagram of Post-Order Traversal for any Tree Data Structure Implementation 1](#_Toc101782135)

[Figure 86 Activity Diagram of Level-Order Traversal for any Tree Data Structure 1](#_Toc101782136)

[Figure 87 Activity Diagram of Hash Table Add Method for any Container (Array, ArrayList, Vector) 1](#_Toc101782137)

[Figure 88 Activity Diagram of Minimax Tic Tac Toe Game 1](#_Toc101782138)

[Figure 89 Activity Diagram of Addition in AVL Trees 1](#_Toc101782139)

[Figure 90 Activity Diagram of Minimax Tic Tac Toe Game 1](#_Toc101782140)

[Figure 91 Visual Indication of Documents Folder in Data Structures Workspace 1](#_Toc101782141)

# References

Bruegge, B., Dutoit, A. (2009.), Object-Oriented Software Engineering: Using UML, Patterns, and Java*.* 3rd

edition. New York: Pearson.

Freeman, E., Robson, E., (2004.), Head First Design Patterns: A Brain-Friendly Guide., 1st Edition. New

York: O’Reilly Media.

Sedgewick, R., Wayne, K. (2011). Algorithms, 4th Edition., New York: Addison-Wesley.

# Completion

Completion Status: *Incomplete*

### Complete Modules

The following modules have been completed and tested for the current version of the data structures workspace.

* edu.sru.thangiah
* edu.sru.thangiah.abstraction
* edu.sru.thangiah.arrays
* edu.sru.thangiah.datastructures
* edu.sru.thangiah.datastructure.generic.linkedlist
* edu.sru.thangiah.datastructure.generic.matrix
* edu.sru.thangiah.datastructures.generic.queue
* edu.sru.thangiah.datastructures.generic.stack
* edu.sru.thangiah.datasturctures.generic.tree.avltree
* edu.sru.thangiah.datastructures.generic.tree.binarytree
* edu.sru.thangiah.datastructures.generic.tree.heaptree
* edu.sru.thangiah.datastructures.generic.tree.redblacktree
* edu.sru.thangiah.datastructures.hashtable
* edu.sru.thangiah.datastructures.linkedlist
* edu.sru.thangiah.datastructures.matrix
* edu.sru.thangiah.datastructures.queue
* edu.sru.thangiah.datastructures.minimax
* edu.sru.thangiah.datastructures.stack
* edu.sru.thangiah.datastructures.tree.avltree
* edu.sru.thangiah.datastructures.tree.binarytree
* edu.sru.thangiah.datastructures.tree.generaltree
* edu.sru.thangiah.datastructures.tree.heaptree
* edu.sru.thangiah.datastructures.tree.redblacktree
* edu.sru.thangiah.datastructures.tree.twothreefourtree
* edu.sru.thangiah.datastructures.vector
* edu.sru.thangiah.inheritance
* edu.sru.thangiah.interfaceex
* edu.sru.thangiah.polymorphism
* edu.sru.thangiah.recursion
* edu.sru.thangiah.sorting
* edu.sru.thangiah.vector
* edu.sru.thangiah.datastructures.graph

### Incomplete Modules

#### Not Tested

* edu.sru.thangiah.datastructures.graph.algorithms.beamSearch
* edu.sru.thangiah.datastructures.graph.algorithms.dijkstra
* edu.sru.thangiah.datastructures.graph.algorithms.prims
* edu.sru.thangiah.datastructures.graph.algorithms.FloydWarshall
* edu.sru.thangiah.datastructures.graph.algorithms.kruskal

#### Not Coded

* Full implementation of Steiner Trees is not complete

# Contribution

The workspace contains 15% of methodologies/code which is retrieved either from Algorithms, 4th Edition or Object-Oriented Software Engineering: Using UML, Patterns, and Java*.* 3rd Edition. Examples methodologies used are separate chaining for hash tables, a recursive method for retrieving the height of any tree, and the reference of pseudocode for specific sorting/graph algorithms. The other 85% is original content.

# Glossary

**Reverse Engineering** – The deductive process of engineering a software system. In other words, the software engineer starts by developing code for the system and then builds the tests and models to mimic the behavior of the code.

**Forward Engineering –** The methodological process of engineering a software system such that you begin with the high-level model or tests for the system and then work on implementing the code based on that model.

**Data structures -** Methodologies and structures for organizing and storing data in order to increase efficiency access and modification.

**Generic Types –** A parameterized type to allow for any particular type to be a parameter to methods, classes, and interfaces. A common use for generics is to allow for the instantiation of instances with different data types.

**Linked Lists** – A linear container of data whose order isn’t bound by their physical placement in memory. The container instead consists of a next or previous pointer which points to the next or previous item in the container.

**Matrix –** A dimensional container which stores a series of rows and columns for data to be partitioned into.

**Array** – A linear container of data identified by a key or index which is used to access a specific data item. Memory allocation is static in Java allowing for lengths of array indices to be next to each other in memory.

**Vector –** A growable container which acts similarly to an array in the sense that data items can be accessed using an index. In this case vectors pose an advantage over arrays given the dynamic structure of the container.

**Queue –** A linear container which maintains a First in First Out (FIFO) structure. In a typical removal an item recently added is removed from the queue. This data structure is dynamic like Vectors but boasts a specialized process for maintaining the container which may be preferred in specific cases.

**Stack –** A linear container which maintains a Last in First Out (LIFO) structure. In a typical removal the first item added to the stack is the first to be removed. This data structure is dynamic like Vectors but boasts a specialized process for maintaining the container which may be preferred in specific cases.

**AVL Tree -** An AVL tree is a self-balancing tree which checks the height of subtrees to ensure that the left and right subtrees don't have a height difference greater than one.

**Red Black Tree –** A Red Black tree is a self-balancing tree similar to AVL that uses colors of nodes and uncle nodes to perform rotations to keep the balance of a binary tree.

**Binary Tree -** A binary search tree is a tree containing any number of nodes and edges. A binary tree node can have at most two children nodes, and the topmost node is denoted as the root node. Any node which contains children is referred to as a parent node. Within a binary tree we can traverse through the nodes, search for a particular value, and perform insertion/deletions as necessary.

**Heap Tree** - A heap tree is a type of tree in which all the nodes are sorted in either ascending or descending order. After an insertion or deletion, the adjacent nodes are compared until the all the nodes are in ascending or descending order.

**2-3-4 Tree -** A 2-3-4 tree is another self-balancing tree in which a node can store at most three values sorted from smallest to largest. Each node can be a 2-node, 3-node, or 4-node node each of which stores either 1,2, or 3 data elements.

**Graph -** A graph contains a collection of vertices (nodes) and a series/list of edges which connect the

vertices together. Our implementation represents a directional graph data structure in which edges

have a destination vertex that the source vertex points to. This makes traversals and search strategies

more interesting since now algorithms have to consider the direction of the graph vertices in order to

produce the desired output. The graph package contains a collection of traversal (breadth/depth first

traversal) and search (Dijkstra, Kruskal, Steiner, Floyd-Warshall, Beam/Linear Discriminate) algorithms.

**Minimax -** Minimax is an artificial intelligence decision making algorithm popular in game theory. The objective is to find the optimal move in a given state. Minimax contains a maximizer and a minimizer. The maximizer bases moves on a greedy approach which achieves the highest score, while the minimizer attempts to get the lowest score possible. Backtracking is used in a tree like data structure to navigate through all potential action states and determine which optimizes the given goal.

**Dijkstra’s Algorithm -** Shortest path algorithm that starts and a beginning vertex, considering all the adjacent vertices, and chooses the path that contains the smallest weight. This process is then continued until the destination vertex is reached, at which point the output should be the shortest path from source to destination. Our implementation outputs the shortest path from the start to all other nodes in the graph.

**Floyd-Warshall Algorithm -** Shortest path algorithm which finds the shortest path for every pair of vertices in a given edge weighted directed Graph.

**Kruskal’s Algorithm -** Another shortest path algorithm that constructs a minimum spanning forest and finds the minimum spanning tree.

**Prim’s Algorithm -** Greedy algorithm for finding the minimum spanning tree for a given starting vertex. The algorithm maintains a list of vertices included in the tree, and those that are not Prim's algorithm then considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing nodes included in the minimum spanning tree

**Steiner Trees -** Constructs a Steiner tree used for calculating the shortest path between two vertices. Starting with a subtree consisting of one terminal vertex, the algorithm loops through the subtree checking if the vertex does not span all the terminal vertices. If it doesn't then we select a terminal vertex not in the subtree which is closest to the terminal vertex and add it to the shortest path subtree.

**Breadth First Search -** Search algorithm which visits every vertex and edge once at each level in the tree to search for a specific node

**Breadth First Traversal -** Traversal algorithm which visits every vertex and edge once at each level in the tree to search for a specific node

**Depth First Search -** Search algorithm which starts at the root node and visits as far left/right in the tree before backtracking

**Depth First Traversal -** Traversal algorithm which starts at the root node and visits as far left/right in the tree before backtracking

# Problem Explanation

Currently, the workspace used to educate students on the fundamentals of data structures lacks a lot of the raw code, content, and information needed to properly demonstrate each area of the Algorithms and Data Structures class. The general structure is messy and could use some refining by categorizing files and class content. The workspace, in most cases, only supports specific data types such as a standard integer. There are some cases where generics are supported, but the goal is to have generics and integers supported in every example given. ListOpsInt and ListOpsIntGeneric should have a greater pool of methods such as removeLast(), indexOf(), setAtIndex(), etc. to allow for smoother implementations of other structures using this interface class. Additionally, we propose reorganizing the datastructures package, which includes moving standardizing all interface classes in their own separate package. An example of this is the tree.binarytree package, which houses the TreeOpsInt class. This interfance class should be moved to edu.sru.thangiah.datastructures to maintain the structure of the workspace. There are also instances in which classes are in the wrong package. Zoo.java features a beginner example of inheritance, but it is not in the edu.sru.thangiah.inheritance package. Having all the regular int interface classes in the datastructures package will also ensure the importance of the package since it is integral to most of the data structures. Reorganizing these discrepancies will improve the useability of our program and will enable a more objective structure of the overall workspace. Another area where changes will be made is in the use of abstract classes. Abstract classes will be used to inherit from the int/generic interface class but will also enable special customization of methods specific to the particular datastructure implementation. An example of this is the doubly/singly linked list datastructures which both contain separate methods, but still should inherit from the base list methods. In this case we will create AbstractSingleLinkedList and AbstractDoublyLinkedList classes which inhert from ListOpsInt and allow for the necessary customization described above. In addition to the restructuring of the workspace, we propose additional documentation of each class and data structure implementation to aid as a supplement to the student/user’s understanding of the topics. Some documentation can be improved through block comments for convenience as well as a separate written manual for readability. The data structures that should be explored are stacks, queues, linked lists, doubly linked lists, and variations of binary trees. The binary tree variations will include red-black trees, 2-3-4 trees, and AVL trees. All these structures will have generic versions included with their standard integer implementations. Another category that needs improvement is with respect to sorting and hashing functions. Sorting should be implemented with generic and integer sorting similarly to the other data structures. Methods such as bubble sort, insertion sort, shell sort, merge sort, quicksort, and others will be included. Design patterns and other core Java concepts which can improve the quality of the data structures workspace will be implemented as needed. An example of this is the use of iterators to be able to quickly retrieve and manipulate elements in a container (list, queue, etc.) one by one. With iterators, we can remove indices while still being able to iterate over the contents of the container. The normal method of removing an element involving a loop causes exceptions to be raised since the contents of the container are changing during each iteration. Software engineering concepts should be included when applicable and when learned throughout the class. The testing process will be documented and each new implementation, whether it be a small function or entire structure, should be thoroughly tested to prove its correctness and capability at solving the problem given to it.

## Overview of the System

* The complexity of our system to support this software development is relatively moderate, and minimum/required hardware resources are minimal enough we can support a wider audience of users without having to worry much about resources impacting the quality of our delivered solution.
* The main component of this project consists of the Eclipse IDE for Enterprise Java and Web Development 2021-12. All data structure implementations only rely natively on this IDE and do not include any third-party libraries/databases to aid in the implementation. This further increases the useability of this solution since the likelihood for users running into issues while running this solution is limited to only a handful of causes.

## Class and Package Descriptions

### edu.sru.thangiah

Standard package with examples of Java fundamentals

* BankAccount.java
  + Shows basic Object-Oriented Programming
* GoodbyeWorld.java
  + Shows constructors and instances of an object
* HelloWorld.java
  + Demonstrates overloaded constructors
* LawyerType.java
  + Full example program with classes, instances, and usage of methods
* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Beginning OOP Examples to Run

* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.abstraction

* BoatTypes.java
  + Shows an example of abstraction in Java
* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)



Figure Diagram of Beginning Abstraction Example to Run

* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.arrays

Various array examples in Java

* ArrayAlias.java
  + Shows aliasing of arrays in java. A1 and a2 are array reference variables so any changes made to a2 are also made to a1.
* ArrayHalfFlip.java
  + Example of flipping the ending half of an array and demonstrates how arrays can be iterated through
* ArrayInteger.java
  + Example of instantiating array with type Integer (object)
* Arrays.java
  + Example of standard instantiating procedure for arrays and printing out the contents of the container.
* ShowMonth.java
  + Example of storing the days of each month using arrays.
* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Array Examples to Run

* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures

* This package acts as the beginning framework for all subsequent data structure implementations. For the basic user looking to simply learn certain data structures this package is not necessary to fully understand, just keep in mind that this is the first starting point for developing any data structure in our workspace.

Collection of Ops interface classes that each data structure related will build off of by extension

* AbstractTree.java
* ArrayListStructure.java
* ArrayStructure.java
* BaseOpsInt.java
* ListOpsInt.java
* TreeOpsInt.java
* TreeSearchOps.java
* TreeTraversalOps.java
* How to run:
* Because these are interface classes, these when run will not provide any output. For this reason we don’t provide any suggestions/instructions for running
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.generic

* This package acts as the beginning framework for all subsequent data structure implementations. For the basic user looking to simply learn certain data structures this package is not necessary to fully understand, just keep in mind that this is the first starting point for developing any data structure in our workspace.

Collection of generic Ops Interface classes that each generic data structure will extend off

* AbstractTreeGeneric.java
* BaseOpsGeneric.java
* ListOpsGeneric.java
* TreeOpsGeneric.java
* TreeSearchOpsGeneric.java
* TreeTraversalOpsGeneric.java
* How to run:
* Because these are interface classes, these when run will not provide any output. For this reason we don’t provide any suggestions/instructions for running
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.generic.linkedlist

Single and Double Generic Linked List package

Single linked lists are like arrays that are dynamically allocated in the memory, with each node having its own data and known next node. Double linked lists are the same, except they have the previous node stored as well. These nodes store the generic data type.

* AbstractDoubleLinkedListGeneric.java
* AbstractSingleLinkedListGeneric.java
* DoubleLinkedListGeneric.java
* DoubleLinkedListGenericTest.java
* NodeOneLinkGeneric.java
* NodeTwoLinksGeneric.java
* SingleLinkedListGeneric.java
* SingleLinkedListGenericTest.java
* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated with medium confidence

Figure Diagram of Linked List Examples to Run

* DoubleLinkedListGeneric and SingleLinkedListGeneric both contain implementations of the data structures, whereas DoubleLinkedListGenericTest/SingleLinkedListGenericTest are the JUnit test classes. NodeOneLinks and NodeTwoLinksGeneric should also be looked at prior to running to understand how linked lists function as a series of nodes which contain a next (or previous) pointers.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.generic.matrix

Generic Matrix class for matrices storing generic values

* AbstractMatrixGeneric.java
* MatrixClassGeneric.java
* MatrixNodeGeneric.java
* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text, website

Description automatically generated

Figure Diagram of Matrix Examples to Run

* MatrixClassGeneric contains the implementation of the matrix data structure. It is important to note that the user should also look at MatrixNodeGeneric to better understand how each element in the matrix is a node which consists of a set of cardinal direction pointers (up, down, left, right)
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.generic.queue

Generic Queue package

Queues are a First In First Out (FIFO) data structure where in an array, there is a designated front and back, and when data is enqueued, it is added to the front. When data is dequeued, it is removed from the back. This queue uses generic data types.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Graphical user interface, text, website

Description automatically generated

Figure Diagram of Queue Examples to Run

* QueueArrayGeneric contains a queue implementation using arrays. The abstract class merely acts as another foundation on top of the interface classes, so running this class doesn’t provide any additional information.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.generic.stack

**Generic Stack Package**

Stacks are a Last In First Out (LIFO) data structure in which data items are only placed at the top and

removed from the top. The push method is used to add an element to the stack, while the pop method

removes an element from the stack. This implementation uses an array to create the stack abstraction.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Stack Examples to Run

* StackGeneric uses a fixed size array to implement the stack, and stack specific methods such as push, pop, and top are implemented to further demonstrate the data structure. The abstract class merely acts as another foundation on top of the interface classes, so running this class doesn’t provide any additional information.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.generic.tree.avltree

**Generic AVL Tree Package**

An AVL tree is a self-balancing binary search tree that checks to ensure that the tree is balanced after each insertion/deletion. A series of left/right rotations are performed to balance the tree.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of AVL Tree Classes to Run

* AvlTreeGeneric acts as the main implementation of the datastructure, while AvlTreeGenericTest is the JUnit test class for ensuring all the methods are functioning as intended. Users interested in learning more about this data structure should check out the AvlTreeNodeGeneric class to see how the individual nodes are structured and how links between the nodes are formed.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.generic.tree.binarytree

**Generic Binary Tree Package**

A binary search tree is a tree containing any number of nodes and edges. A binary tree node

can have at most two children nodes, and the topmost node is denoted as the root node. Any

node which contains children is referred to as a parent node. Within a binary tree we can

traverse through the nodes, search for a particular value, and perform insertion/deletions as

necessary.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Binary Tree Classes to Run

* BinarySearchTreeGeneric acts as the implementation of the data structure, while BinarySearchTreeGenericTest is the JUnit tests for all the binary tree methods. For those interested in learning more about this data structure users can view the BinaryTreeNodeGeneric class to understand how binary tree nodes are instantiated and how the node relationships are established and managed.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.generic.tree.heaptree

**Generic Heap Tree Package**

A heap tree is a type of tree in which all the nodes are sorted in either ascending or descending

Order. After an insertion or deletion the adjacent nodes are compared until the all the nodes

are in ascending or descending order.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Heap Tree Classes to Run

* MaxHeapTreeGeneric demonstrates a generic implementation of a max heap tree. MaxHeapTreeGenericTest is the JUnit test class to ensure all the methods in the max heap tree are working as intended. For those interested in learning more about the heap tree data structure users can view the MaxHeapTreeNodeGeneric class to learn more about how heap tree nodes are instantiated and how they create/maintain their relationships.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.generic.tree.redblacktree

**Generic Red Black Tree Package**

A red black tree is another kind of self balancing tree in which each node retains a color

Property (red or black) which is checked to ensure proper balance of the tree.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Graphical user interface, text

Description automatically generated with medium confidence

Figure Diagram of Red Black Tree Classes to Run

* RedBlackTreeGeneric contains the implementation of the tree data structure. It is important also to check out the RedBlackTreeNodeGeneric class to learn how the individual nodes are instantiated and how the color is determined, but running this class won’t provide any additional information.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructure.graph

**Graph Package**

A graph contains a collection of vertices (nodes) and a series/list of edges which connect the vertices

together. Our implementation represents a directional graph data structure in which edges have a

destination vertex that the source vertex points to. This makes traversals and search strategies more

interesting since now algorithms have to consider the direction of the graph vertices in order to produce

the desired output. The graph package contains a collection of traversal (breadth/depth first traversal)

and search (Dijkstra, Kruskal, Steiner, Floyd-Warshall, Beam/Linear Discriminate) algorithms.

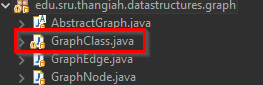
* **How to run:**
* See below for a diagram of the example which can be run (highlighted in red)

Figure Diagram of Classes to Run for Graph Data Structure

* Note that all of the traversal/search algorithms are contained within the GraphClass.java class. GraphEdge.java and GraphNode.java should be looked at prior to running any graph examples in order to better understand how the data structure is implemented.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.hashtable

**Hash Table Package**

A hash table acts as a container which stores key/value pairs aiding towards faster lookup

times. Keys are hashed, meaning that their values are passed to a hash function, which assigns

a unique value. From this hash value we can then create an index for the key/value pair to be

placed into. In some instances keys may overlap, and as a result we run into collisions. For our

implementation we use separate chaining, meaning that each element stores a linked list of

key/value pairs, such that if there are collisions, both pairs will still be stored in the same index

but a next pointer will be used to refer to the next key/value pair in the index.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Hash Table (int) Classes to Run

* Note that there are array, arraylist, and vector implementations for the hash table package. Each performs the same functions, but are merely implemented using different containers to demonstrate the versatility of hash tables. HashTableTest tests all these classes. HashTableNode should be looked at prior to running the implementations to gain a better understanding of how hash table nodes are instantiated/maintained.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.linkedlist

Single and Double Linked List package

Single linked lists are like arrays that are dynamically allocated in the memory, with each node having its own data and known next node. Double linked lists are the same, except they have the previous node stored as well.

* AbstractDoubleLinkedList.java
* AbstractSingleLinkedList.java
* DoubleLinkedList.java
* DoubleLinkedListTest.java
* NodeOneLink.java
* NodeTwoLinks.java
* SingleLinkedList.java
* SingleLinkedListTest.java
* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Single and Double Linked List (int) Classes to Run

* DoubleLinkedList and SingleLinkedList both contain implementations of the data structures, whereas DoubleLinkedListTest/SingleLinkedListTest are the JUnit test classes. NodeOneLinks and NodeTwoLinks should also be looked at prior to running to understand how linked lists function as a series of nodes which contain a next (or previous) pointers.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.matrix

Generic Matrix class for matrices storing generic values

* AbstractMatrix.java
* MatrixClass.java
* MatrixNode.java
* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Graphical user interface

Description automatically generated

Figure Diagram of Matrix (int) Classes to Run

* MatrixClass contains the implementation of the matrix data structure. It is important to note that the user should also look at MatrixNode to better understand how each element in the matrix is a node which consists of a set of cardinal direction pointers (up, down, left, right)
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.queue

Generic Queue package

Queues are a First In First Out (FIFO) data structure where in an array, there is a designated front and back, and when data is enqueued, it is added to the front. When data is dequeued, it is removed from the back. This queue uses generic data types.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Queue (int) Classes to Run

* QueueArray contains a queue implementation using arrays. CircularQueue is a queue implementation where the last item in the queue points to the first item. QueueLinkedList is an implementation using linked lists, and QueueVector is an implementation using Vectors. The abstract class merely acts as another foundation on top of the interface classes, so running this class doesn’t provide any additional information.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.search.minimax

**Minimax package**

Minimax is a decision algorithm which minimizes the possible loss for a worst case scenario.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Graphical user interface, application

Description automatically generated

Figure Diagram of Minimax (int) Classes to Run

* TicTacToe represents an example implementation of minimax in which a player can play against a bot whose decisions are based on minimax. It is recommended that users first look at the Board and Player classes to understand how the environment is created and how the player is maintained throughout the game.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.stack

**Stack Package**

Stacks are a Last In First Out (LIFO) data structure in which data items are only placed at the top and

removed from the top. The push method is used to add an element to the stack, while the pop method

removes an element from the stack. This implementation uses an array to create the stack abstraction.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Graphical user interface, text

Description automatically generated

Figure Diagram of Stack (int) Classes to be Run

* Stack uses a fixed size array to implement the stack, and stack specific methods such as push, pop, and top are implemented to further demonstrate the data structure. We create a similar implementation in StackLinkedList, except now we use a linked list instead of an array. The abstract class merely acts as another foundation on top of the interface classes, so running this class doesn’t provide any additional information.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.tree.avltree

**AVL Tree Package**

An AVL tree is a self-balancing binary search tree that checks to ensure that the tree is balanced after each insertion/deletion. A series of left/right rotations are performed to balance the tree.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated with low confidence

Figure Diagram of AVL Tree (int) Classes to Run

* AvlTree acts as the main implementation of the data structure, while AvlTreeTest is the JUnit test class for ensuring all the methods are functioning as intended. Users interested in learning more about this data structure should check out the AvlTreeNode class to see how the individual nodes are structured and how links between the nodes are formed.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.tree.binarytree

**Binary Tree Package**

A binary search tree is a tree containing any number of nodes and edges. A binary tree node

can have at most two children nodes, and the topmost node is denoted as the root node. Any

node which contains children is referred to as a parent node. Within a binary tree we can

traverse through the nodes, search for a particular value, and perform insertion/deletions as

necessary.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Binary Tree (int) Classes to Run

* BinarySearchTree acts as the implementation of the data structure, while BinarySearchTreeTest is the JUnit tests for all the binary tree methods. For those interested in learning more about this data structure users can view the BinaryTreeNode class to understand how binary tree nodes are instantiated and how the node relationships are established and managed.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.tree.generaltree

**General Tree Package**

A general tree is a binary tree in which each node can have n-number children. Instead of being limited

to at most 2 children per parent we can have a parent with 3 children each of which have 4 children of

their own.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Graphical user interface

Description automatically generated

Figure Diagram of General Tree (int) Classes to Run

* GeneralTree is the implementation class, whereas GeneralTreeTest is the JUnit test class for ensuring all the methods in GeneralTree.java function as intended. It is recommended that users first check out GeneralTreeNode to gain an understanding of how we instantiate a tree with any number of children for a given node.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.tree.heaptree

**Heap Tree Package**

A heap tree is a type of tree in which all the nodes are sorted in either ascending or descending

Order. After an insertion or deletion the adjacent nodes are compared until the all the nodes

are in ascending or descending order.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Heap Tree (int) Classes to Run

* MaxHeapTree demonstrates a generic implementation of a max heap tree. MaxHeapTreeTest is the JUnit test class to ensure all the methods in the max heap tree are working as intended. For those interested in learning more about the heap tree data structure users can view the MaxHeapTreeNode class to learn more about how heap tree nodes are instantiated and how they create/maintain their relationships.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.tree.redblacktree

**Red Black Tree Package**

A red black tree is another kind of self balancing tree in which each node retains a color

Property (red or black) which is checked to ensure proper balance of the tree.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated with low confidence

Figure Diagram of Red Black Tree (int) Classes to Run

* RedBlackTree contains the implementation of the tree data structure. RedBlackTreeTest contains the JUnit tests for the red black tree implementation. It is important also to check out the RedBlackTreeNode class to learn how the individual nodes are instantiated and how the color is determined, but running this class won’t provide any additional information.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.tree.twothreefourtree

**Two Three Four Package**

A 2-3-4 tree is another self-balancing tree in which a node can store at most three values sorted

from smallest to largest. Each node can be a 2-node, 3-node, or 4-node node each of which

stores either 1,2, or 3 data elements.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Graphical user interface, text, website

Description automatically generated

Figure Diagram for 2-3-4 Tree (int) Classes to Run

* TwoThreeFourTreeInt contains the implementation of the data structure. It is recommended to first check out the TwoThreeFourTreeNode class to gain a further understanding of how each node can maintain 2, 3, or 4 children.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.datastructures.vector

**Vector Package**

A vector is a dynamic container which is able to store data elements using specific add/remove

methods. Vectors also allow users to get the data element for a given index similarly to arrays.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Vector (int) Classes to Run

* ReverseVectorStructure acts like a normal vector except elements are stored in ascending order from front to back. VectorDemo demonstrates the use of the java.util.Vector package to instantiate a simple vector. VectorStructure demonstrates a vector implementation without the use of any existing libraries
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.inheritance

**Inheritance Package**

Inheritance is an OOP concept in which the attributes and methods of one class is acquired by

another class. This is important when considering that a template class for an animal can be

made which is inherited by other specific animals (elephant, squirrel, bird, etc.). Each specific

animal shares general characteristics/properties as an animal and as such the need for

inheritance is brought about. There are two kinds of inheritance, the first being aggregation in

which the child class can exist independently of the parent class. Composition is the other type,

in which the child class cannot exist independently of the parent class.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Inheritance Classes to Run

* Car is a composition example. Cartoon demonstrates an association inheritance as well as Department. Zoo also shows an example of association.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.interfaceex

**Interfaceex Package**

Interface Classes are important for laying the foundation or template for classes which share

similar properties/attributes. This concept is based on inheritance and the programmer

specifies a template(interface class) in which all the child classes will inherit its’ public

attributes/methods.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)



Figure Diagram of Interface Classes to Run

* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.polymorphism

**Polymorphism Package**

Polymorphism is another pillar of OOP in which something can be displayed in multiple

different forms. A common example of this is function/operator overloading in which you can

have multiple occurrences of the same function/operator but each performs different tasks.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)



Figure Diagram of Polymorphism Classes to Run

* Shapes demonstrates how a template shape class can be extended and used in multiple different forms (circle, square, triangle, etc.)
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.recursion

**Recursion Package**

Recursion is a technique that reduces the size/complexity of a function by instead making the

function call itself.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Graphical user interface

Description automatically generated

Figure Diagram of Recursion Classes to Run

* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.sorting

**Sorting Package**

Sorting is an important concept in which we can define methodologies or algorithms that can take an unsorted list of elements and return the sorted version as output. Sorting helps demonstrate the common programming pitfalls which can increase the time complexity for your implementation.

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)

Text

Description automatically generated

Figure Diagram of Sorting Classes to Run

* BubbleSort works by repeatedly swapping the adjacent elements if they are in the wrong order.
* HeapSort visualizes the input unsorted list as a heap tree and makes comparisons similar to a heap tree until the data is sorted.
* InsertionSort builds the sorted output result by scanning and sorting the list one element at a time. Starting at the first element t works its way to the end by comparing each element against the previous already sorted elements.
* MergeSort breaks the list roughly in half until it cannot be split further and then compares these halves.
* QuickSort picks a pivot element in the unsorted list and then partitions all the other elements around the pivot. A new pivot is selected until the list is sorted.
* RadixSort operates multiple iterations of count sort for each digit place (ones, tens, hundreds, etc.) and then sorts the elements according to their digit value during each iteration.
* SelectionSort the goal is to identify the index of the largest element of the array. Assume the first element is the largest, and then form a competition among all the remaining values. As we come across larger values, we update the index of the current maximum value. In the end, the index must point to the largest value.
* ShellSort works in similar fashion to insertion sort in which elements far apart from each other are sorted first. Afterwards the interval between elements is reduced until the list is sorted.
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

### edu.sru.thangiah.vector

* **How to run:**
* See below for a diagram of the examples which can be run (highlighted in red)
* These examples can simply be run by right-clicking on the .java file > Run as > Java Application

## Overview of the Solution

* A potential data structures workspace which is to be utilized primarily by educators, students, and individuals interested in improving their understanding of data structures will need to be generic enough in order to allow for any potential implementation of a given data structure. In order to perform this task our solution must introduce the integration of generic types, to allow for any data type instance of a data structure. Since generics can often produce unforeseen challenges and issues within the system it is our responsibility to also introduce a heavy set of tests with the goal of ensuring the functionality of all components/modules given the possibility for an instance of any data type. We reference a slew of testing strategies in the Testing section of this document. Further tests can also be found in our Testing Manual found within the documents folder of the data structures workspace.
* The included implementations contained in this project are as follows (A \* indicates that a generic and int implementation has been developed for the given implementation):
* Beginning Java Fundamentals (constructors, instantiation, classes)
* Java OOP examples
* Inheritance
* Interface classes
* Abstraction
* Arrays
* Vectors
* Single/Doubly Linked List\*
* Matrices\*
* Queues (Array implementation & Linked List implementation)\*
* Stacks(Array implementation & Linked List implementation)\*
* Binary Search Trees\*
* Heap Trees\*
* 2-3-4 Trees\*
* AVL Trees\*
* General trees\*
* Red-Black Trees\*
* Sorting\*
* Graphs and their associated algorithms (Dijkstra, Floyd-warshall, steiner, Kruskal, etc.)
* Hashing algorithms

## Path to Solution

* The process and development flow of our project centered on a reverse engineering approach. Since being given an initial workspace containing existing implementation our approach involved a reverse engineering methodology by which data structures were implemented or improved first, and then tests were conducted afterwards. Since the structure of the Software Engineering course involved learning fundamentals first and then focusing on topics such as UML and testing our decision for reverse engineering was majorly driven by our current understanding of certain software engineering principles. If given an opportunity to start a similar project we would choose a forward engineering approach which prioritizes the team’s understanding of the intended implementation for a module/component.

# System Requirements

## Software Requirements

* Java SE 17 ([Download](https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html))
* Git command line tool (https://docs.github.com/en/get-started/quickstart/set-up-git) or [eGit](https://www.eclipse.org/egit/#:~:text=EGit%20is%20an%20Eclipse%20Team,history%20very%20fast%20and%20versatile.) for version control natively in Eclipse
  + additionally, an active github account is needed
* Eclipse 2021 IDE for Enterprise Java and Web Developers – 2021-12 ([Download](https://www.eclipse.org/downloads/packages/release/2021-12))

## Minimum Hardware Requirements

* Required Java version:
  + Java SE 17 ([Download](https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html))
* Memory:
  + 512 MB
* Free Disk Space
  + 500 MB
* Processor Speed
  + 1 GHZ

## Recommended Hardware Requirements

* Required Java version:
  + Java SE 17 ([Download](https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html))
* Memory:
  + 1GB or more
* Free Disk Space:
  + 1GB or more
* Processor Speed:
  + 1.5 Ghz or more

# UML Diagrams

## Use Case Diagrams

### Single Linked List

**Diagram

Description automatically generated**

Figure : Use Case Diagram of Single Linked List

### Doubly Linked List

**Chart, diagram

Description automatically generated**

Figure : Use Case Diagram of Doubly Linked List

### Binary Search Tree

**Diagram

Description automatically generated**

Figure : Use Case Diagram of Binary Search Tree

### General Tree

**Diagram

Description automatically generated**

Figure : Use Case Diagram of General Tree Implementation

### Heap Tree

## Diagram Description automatically generated

Figure Heap Tree Use Case Diagram

## Class Diagrams:

### Edu.sru.thangiah

### Graphical user interface, text Description automatically generated

### Figure Class diagram of edu.sru.thangiah package

### Edu.sru.thangiah.arrays

Graphical user interface, text, application

Description automatically generated

Figure Class diagram of edu.sru.thangiah.arrays

### Edu.sru.thangiah.datastructures

Timeline

Description automatically generated

Figure Class diagram of edu.sru.thangiah.datastructures

### Edu.sru.thangiah.datastructures.generic

Timeline

Description automatically generated

Figure Class diagram of edu.sru.thangiah.datastructures.generic

### Edu.sru.thangiah.datastructures.generic.matrix

Diagram, timeline

Description automatically generated

Figure Class diagram of edu.sru.thangiah.datastructures.generic.matrix

### Edu.sru.thangiah.datastructures.generic.queue

Timeline

Description automatically generated with low confidence

Figure Class Diagram of edu.sru.thangiah.datastructures.generic.queue

### Edu.sru.thangiah.datastructures.generic.stack

Timeline

Description automatically generated with low confidence

Figure Class diagram of edu.sru.thangiah.datastructures.generic.stack

### Edu.sru.thangiah.datastructure.generic.linkedlist

**Timeline

Description automatically generated**

Figure : Class Diagram of Int Single Linked List Implementation

**Timeline

Description automatically generated with medium confidence**

Figure : Class Diagram of Doubly Linked List Implementation

### Edu.sru.thangiah.datastructures.generic.linkedlist

**Timeline

Description automatically generated**

Figure : Class Diagram of Generic Single Linked List Implementation

**Timeline

Description automatically generated**

Figure : Class Diagram of Generic Doubly Linked List Implementation

### Edu.sru.thangiah.datastructures.tree.binarytree

**Timeline

Description automatically generated**

Figure : Class Diagram of Int Binary Tree Implementation

### edu.sru.thangiah.datastructures.generic.tree.binarytree

**Timeline

Description automatically generated**

Figure : Class Diagram of Generic Binary Tree Implementation

### Edu.sru.thangiah.datastructures.tree.generaltree

**Text, Word

Description automatically generated**

Figure : Class Diagram of Int General Tree Implementation

### edu.sru.thangiah.datastructures.tree.heaptree

**Timeline

Description automatically generated**

Figure : Class Diagram of Heap Tree Implementation

### edu.sru.thangiah.datastructures.generic.tree.heaptree

**Timeline

Description automatically generated**

Figure : Class Diagram of Generic Heap Tree Implementation



Figure : Class Diagram of Generic AVL Tree

### edu.sru.thangiah.datastructures.generic.tree.avltree



Figure :Class Diagram of Int AVL Tree

### edu.sru.thangiah.datastructures.tree.avltree

### edu.sru.thangiah.datastructures.tree.redblacktree



Figure : Class Diagram of Int Red Black Tree Implementation

### edu.sru.thangiah.datastructures.generic.tree.redblacktree



Figure Class Diagram of Generic Red Black Tree Implementation

### edu.sru.thangiah.datastructures.stack

Timeline

Description automatically generated with medium confidence

Figure : Class diagram of int stack implementation

### sru.edu.thangiah.datastructures.queue

Graphical user interface, application

Description automatically generated

Figure : Class diagram of int queue implementation

### edu.sru.thangiah.datastructurs.matrix

Graphical user interface, application

Description automatically generated with medium confidence

Figure : Class diagram of int matrix implementation

### edu.sru.thangiah.datastructures.search.minimax

Graphical user interface, application

Description automatically generated

Figure : Class diagram of Tic Tac Toe minimax implementation

### edu.sru.thangiah.datastructures.hashtable

### edu.sru.thangiah.datastructures.vector



Figure : Class diagram of the Vector Data Structure

### edu.sru.thangiah.datastructures.inheritance



Figure : Class diagram of the different examples of inheritance in the inheritance package

### edu.sru.thangiah.datastructures.interfaceex



Figure : Class diagram of the interface example

### edu.sru.thangiah.datastructures.polymorphism



Figure : Class diagram of the polymorphism example

### edu.sru.thangiah.datastructures.recursion



Figure : Class diagram of the recursion example

### edu.sru.thangiah.datastructures.sorting



Figure : Class diagram of the sorting package classes

Graphical user interface

Description automatically generated with medium confidence

Figure : Class diagram of int hash table implementation

## Sequence Diagrams:

### In-Order Traversal for any Tree Data Structure

Diagram

Description automatically generated

Figure Sequence Diagram of In-Order Traversal for any Tree Data Structure

### Pre-Order Traversal for any Tree Data Structure

Diagram

Description automatically generated

Figure Sequence Diagram of Pre-Order Traversal for any Tree Data Structure

### Post-Order Traversal for any Tree Data Structure

Diagram

Description automatically generated

Figure Sequence Diagram of Post Order Traversal for any Tree Data Structure

### Level-Order Traversal for any Tree Data Structure

### A picture containing diagram Description automatically generated

Figure Sequence Diagram of Level Order Traversal for any Tree Data Structure

### Hash Table Add Method (Array)

Graphical user interface

Description automatically generated with medium confidence

Figure : Sequence Diagram of Add Method for Array Hash Table Implementation

### Hash Table Add Method (ArrayList)

Graphical user interface

Description automatically generated

Figure Sequence Diagram of Hash Table Add Method for ArrayList Hash Table Implementation

### Hash Table add method (Vector)

Graphical user interface

Description automatically generated

Figure Sequence Diagram of Add Method for Vector Hash Table Implementation

### Minimax Tic Tac Toe

Graphical user interface

Description automatically generated with medium confidence

Figure MiniMax Tic Tac Toe Sequence Diagram



Figure Sequence Diagram of AVL Tree Addition

## UML Statechart Diagrams

### In-order Traversal for any Tree Data Structure

Diagram

Description automatically generated

Figure Statechart diagram of In-Order traversal for any Tree Data Structure Implementation

### Pre-Order Traversal for any Tree Data Structure

Diagram, timeline

Description automatically generated

Figure State Chart Diagram of Pre-Order Traversal for any Tree Data Structure Implementation

### Post-Order Traversal for any Tree Data Structure

Diagram

Description automatically generated

Figure Statechart diagram of Post-Order Traversal for any Tree Data Structure Implementation

### Level Order Traversal for any Tree Data Structure

Timeline

Description automatically generated

Figure Statechart diagram of Level Order Traversal for any Tree Data Structure Implementation

### Hash Table Add Method for any Container (Array, ArrayList, Vector)

Diagram

Description automatically generated

Figure Statechart Diagram of Add Method for any Hash Table Implementation (Array, ArrayList, Vector)

### Minimax Tic Tac Toe

Diagram

Description automatically generated

Figure State Chart Diagram of Minimax Tic Tac Toe Game



Figure Statechart Diagram of AVL Tree Addition

## UML Activity Diagrams

### In-Order Traversal for any Tree Data Structure

Diagram

Description automatically generated

Figure Activity Diagram of In-Order Tree Traversal for any Tree Data Structure

### Pre-Order Traversal for any Tree Data Structure

Diagram

Description automatically generated

Figure Activity Diagram of Pre-Order Traversal for any Tree Data Structure Implementation

### Post-Order Traversal for any Tree Data Structure

Diagram

Description automatically generated

Figure Activity Diagram of Post-Order Traversal for any Tree Data Structure Implementation

### Level-Order Traversal for any Tree Data Structure

Diagram

Description automatically generated

Figure Activity Diagram of Level-Order Traversal for any Tree Data Structure

### Hash Table Add Method for any Container (Array, ArrayList, Vector)

Diagram

Description automatically generated

Figure Activity Diagram of Hash Table Add Method for any Container (Array, ArrayList, Vector)

### Minimax Tic Tac Toe Game

Diagram

Description automatically generated

Figure Activity Diagram of Minimax Tic Tac Toe Game



Figure Activity Diagram of Addition in AVL Trees

Diagram

Description automatically generated

Figure Activity Diagram of Minimax Tic Tac Toe Game

# Caveats/Minefields

## Introduction

* Developers wishing to continue progress on the data structures workspace should be aware of certain caveats/minefields which can stall or prevent further development if left unforeseen. This section talks about each specific component/module which needs to be looked at before continuing development, as well as potential solutions to these issues.

## Core Data Structures and Arrays

* Although this issue is not present in the data structures workspace, the issue of using arrays to implement data structures is one that should be mentioned before continuing any further progress. Although arrays are useful in demonstrating the concept of a data structure, when we branch out to larger use cases which require a more robust container arrays can create pitfalls. Arrays are inefficient in insertion/deletion times, and waste memory since the container is not dynamic. One can note the use of linked lists as the core container for many data structures. The reasoning behind this, and the importance of continuing to use linked lists for future development is that there is a more efficient use of memory, and insertion/deletions are accomplished at a faster pace. Although search time for linked lists is limited to a worst case of n(O(n)), the risk of wasted memory and a static size outweighs the disadvantages of linked lists.

## Keeping Track of Visited Nodes for Graph Algorithms

* For implementations of graph algorithms there always must be a secondary concern of efficiency while ensuring accuracy of the result. For many of our algorithms we noticed that vertices would often be visited multiple times even after adjacency matrices or shortest paths have been calculated for the given vertex. This proved to be an area desperately needing optimization as runtime would often be run longer by iterating through these already visited nodes. Instead of wasting extra runtime we decided to create an additional linked lists for nodes in the graph which have been visited. Through adding this container we ensured that the shortest paths to the visited node have already been calculated, and then we add this to the linked list to save extra runtime. The results improved the efficiency of our algorithms in reaching their solutions, and also cut down on the time it took to reach that solution. Future implementation of graph algorithms and even tree algorithms should consider this in order to maintain the requirement of delivering a solution which is accessible to as many individuals as possible.

## Ensuring the Functionality of Core Data Structures First

* Often when continuing the development of the data structures workspace we would focus too much on jumping to new tasks rather than ensuring that the core data structures which depend on these new tasks work and function properly. This inefficiency caused too much jumping around in the code, and a better methodology of ensuring the core components/modules are working first is critical in ensuring that future development is as successful as possible. We achieved this as much as possible as demonstrated in the Testing section, but future development must build off of this existing principle.

# Documentation

## Documentation Files

* All current documentation including the Config & Install, Security, Technical, Testing, and User Manual can be found within the project workspace in a separate documents folder. See below for where the documents folder is located in the project hierarchy.

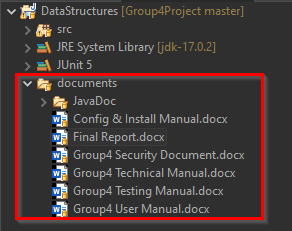


Figure Visual Indication of Documents Folder in Data Structures Workspace

* Further documentation pertaining to the JavaDoc can be found within the documents folder inside a separate JavaDoc folder. Documentation is specified in the JavaDoc for each class within each package and can be accessed either in a web format or by navigating to each class individuall.

# File Path Names

## JavaDoc

* JavaDoc is contained within the Data Structures workspace in documents>JavaDoc. Accessing an individual classes JavaDoc requires navigating through edu>sru>thangiah and following the similar structure as how the class is placed in the workspace. (ex: JavaDoc for HeapTreeNodeGeneric is under documents>JavaDoc>edu>sru>thangiah>datastructures>generic>heaptree>eapTreeNodeGeneric)

## User Manual

* The user manual is currently housed in the documents directory of the Data Structures workspace

## Config & Install Manual

* The config & install manual is currently housed in the documents directory of the Data Structures workspace. Additionally users can find the config & install manual in the master branch of our github repository

## Security Manual

* The security manual is currently housed in the documents directory of the Data Structures workspace

## Technical Manual

* The technical manual is currently housed in the documents directory of the Data Structures workspace

## Testing Manual

* The testing manual is currently housed in the documents directory of the Data Structures workspace

# Code Reusability

* One of the main intentions behind developing this software system was to allow for data structures to be generic enough to ensure reusability for any domain. Through the use of generic types, and a slew of containers/data structures that can be instantiated by the user in whatever application we have strived to ensure that all of our classes are reusable in at least some sense, but in most cases we note that our implementations are fully reusable. Integer implementations are only partially reusable due to their limitation of being instantiated with only one type, but these examples are still useful and are included in the list of reusable classes/packages. At the foundation of understanding data structures lies the principle of ensuring reusability. Since data structures strive to improve the efficiency of storing and modifying data they must accommodate to the various forms in which this data can be represented. We ensure this principle and build our implementations in a manner such that users can also understand the usefulness of this principle.

## Reusable Classes/Packages/Methods

* The following packages and classes are reusable by the programmer for future implementations or domains which require the functionality of specific data structures:

### edu.sru.thangiah.datastructures

* Contains interface/abstract classes which house reusable methods which should be implemented for any new data structure not implemented in this workspace.

### edu.sru.thangiah.datastructures.generic

* Contains interface/abstract classes which house reusable methods which should be implemented for any new data structure not implemented in this workspace.

### edu.sru.thangiah.datastructures.linkedlist (int/generic)

* Not only provides interface/abstract classes for templates of singly/doubly linked lists but also provides implementations for singly/doubly linked lists that house key methods for manipulating/viewing the container. Each linked list implementation is able to easily be instantiated and the classes/methods for the implementations are designed such that if modifications/additions need to be made to the core functionality of the data structure programmers are able to perform these tasks.

### edu.sru.thangiah.datastructures.matrix (int/generic)

* Includes abstract/interface classes which can be applied to any new matrix type by providing the core intended functionality of the container. In addition to these classes users are able to reuse the MatrixClass for a variety of uses. One example which is used in our project is the use of a matrix to store the weights of edges as well as the shortest path between adjacent nodes for Floyd-Warshall’s algorithm.

### edu.sru.thangiah.datastructures.queue (int/generic)

* Includes abstract/interface classes which can be applied to any new queue variety by providing the core intended functionality of how the container should operate. In addition to these classes, users are able to reuse the QueueArray and QueueLinkedList implementations for a variety of uses. One example considered in this project is the use of queues for tree/graph traversal and for calculating the shortest path between two nodes in a graph.

### edu.sru.thangiah.datastructures.stack (int/generic)

* Includes abstract/interface classes which can be applied to any new stack variety by providing the core intended functionality of how the container should operate. In addition to these classes, users are able to reuse the StackArray and StackLinkedList implementation for a variety of uses. One example considered in this project is the use of queues for in/pre-order traversal.

### edu.sru.thangiah.datastructures.tree.avltree (int/generic)

* Includes abstract/interface classes which can be applied to any modified AVLtree through providing the core intended functionality of how the tree should operate. In addition to these classes, users are able to reuse the AvlTree class implementation for any particular use case (in-memory sorts of sets, dictionaries, database applications, etc.)

### edu.sru.thangiah.datastructures.tree.binarytree (int/generic)

* Includes abstract/interface classes which can be applied to any modified binary tree through providing the core intended functionality of how the tree should operate. In addition to these classes, users are able to reuse the BinaryTree class implementation for any particular use case (searching/sorting algorithms, decision trees, Huffman Coding, etc.)

### edu.sru.thangiah.datastructures.tree.heaptree (int/generic)

* Includes abstract/interface classes which can be applied to any modified heap tree through providing the core intended functionality of how the tree should operate. In addition to these classes, users are able to reuse the MaxHeapTreeArray and MaxHeapTreeLinkedList classes for any particular use case (Dijkstra’s algorithm, priority queues, schedulers, heapsort, etc.)

### edu.sru.thangiah.datastructures.tree.redblacktree (int/generic)

* Includes abstract/interface classes which can be applied to any modified red black tree, through providing the core intended functionality of how the tree should operate. In addition to these classes, users are able to reuse the RedBlackTree class for any particular use case (TreeSet, TreeMap, Hashmap, applications which require optimal computational times for insertion/deletion/search operations, etc.)

### edu.sru.thangiah.datastructures.twothreefourtree (int)

* Includes abstract/interface classes which can be applied to any modified 2-3-4 tree, through providing the core intended functionality of how the tree should operate. In addition to these classes, users are able to reuse the TwoThreeFourTree class for any particular use case (Completely Fair Scheduler, Linux Kernel, file systems, etc.)

### edu.sru.thangiah.datastructures.graph (int)

* Includes abstract/interface classes which can be applied to any modified graph example, through providing the core intended functionality of how the graph should operate. In addition to these classes, users are able to reuse the graph class for any particular use case (Web searching, GPS, social media connections between friends, Google Maps routing, knowledge graphs, etc.). Although the current implementation is limited to only one data type the reusability of this data structure is still possible, and more importantly, valuable.

### edu.sru.thangiah.datastructures.hashtable (int)

* Includes abstract/interface classes which can be applied to any modified graph example, through providing the core intended functionality of how the table should operate. In addition to these classes, users are able to reuse the HashTable class for any particular use case (databases, searching through large data sets, etc.)

### bestMove() in Board.java class located in edu.sru.thangiah.datastructures.search.minimax

* Although the application domain for our minimax implementation is specific only to a tictactoe environment, the basic principle and pseudocode behind the minimax algorithm can be reused to apply to any applicable domain (other turn-based games).

# Testing

## Human-Computer Testing

* The goals of our project were to implement data structures in a way to maximize their learnability, reusability to create other structures, and usability in new projects. In order to ensure that the human interaction with our project was exceptional, we carefully documented our code and thinking behind it. With that documentation, we read through code from the perspective of a new user and tried to ensure that it could be understood from the written code and documentation.

## Blackbox Testing

* Testers initiated in black box testing our system by interacting with the implemented structures through their mains. The main allowed testers to submit inputs into a structure, and the outputs would be printed in the console, allowing the tester to confirm whether the output matches their expectation based on the input provided.

## Whitebox Testing

* Extensive white box testing was done on each structure, both during implementation of the structure and after to confirm its functionality. Structures had their methods swept through to determine weak points, and those points were documented to ensure they were tested in our other methods, as well as tested through simple input output in the main.

## Unit Testing

* Unit testing each structure was crucial, and the primary way each structure was tested. The majority of packages had their own jUnit 5 test in order to automate the unit testing. Each test had specific data sets that were made to test all variations of inputs and automate the output verification.

## Integration Testing

* Integration testing was implemented using jUnit 5 as well by using the suite capabilities. Suites tied standard jUnit tests together in order to run them all at the same time. This allowed testers to ensure that when combining packages, or including new structures, the tests of previous working code would still succeed.

## Regression Testing

* Regression testing was covered in the same way that integration testing was. The suite allowed us to rerun all tests to make sure that they are working properly, even after code was removed, added, or modified. With how our structures relied on one another, the suite made regression testing simple and essentially automated most of our testing.

## Boundary Analysis

* Boundary analysis was implemented in combination with our white box testing. When a boundary was identified, it was documented and added to our testing data sets for each structure. The boundaries were then included in each jUnit test, allowing us to test inside and outside both the lower and upper bounds of our structures. The bounds were retested each time code changed.

## Volume Testing

* Volume testing was mostly done in the main of each structure in order to keep jUnit tests running at a higher efficiency. The main structures tested with massive amounts of volume were list structures, as a lot of other structures handled volume by implementing a lower-level list such as a linked list. These lists had many nodes added, and methods ran on them to see if large amounts of data would slow the list methods, in turn slowing other structures that relied on those lists.

## Stress Testing

* Stress testing was implemented the same way volume testing was implemented. Our structures, being such low-level structures, did not have as many stress points as a larger project with more moving parts. The main stress point for each structure, such as a tree with a traversal method, would be the amount of data put into the structure.

## Missing Resources

* Resources were kept in check through our integration testing using the suite tests. These suites would run when new resources were added, resources were removed, and resources changed. There were not as many instances of missing resources since the majority of our structures only relied on structure we had already tested and created within the project.

## Recovering from Crashes

* Crash recovery was extensively handled by the IDE, Eclipse, since it is where our project was created and ran from. Eclipse handled logging and reporting of crashes if they were to occur but were not much of a concern in this project due to a lot of our structures being separate from each other.

# Deployment/Maintenance

* Deployment of our solution depends primarily on the quality of the GitHub repository which stores the project. Ensuring that no errors are present in the current copy pushed to the master branch is critical in ensuring the future maintenance of the solution. Since the project is housed entirely as an eclipse workspace the possibility for failure is limited only to a single failpoint. Through consistent checking and testing of new changes developers can ensure that maintenance is as efficient as possible.

# Post-Mortem Analysis

* Possibly the simplest modification to the development lifecycle of this project would’ve been to approach implementations using a forward engineering methodology. Often inconsistencies between approaches on how best to implement a module/component would merge between development among team members. Having a standardized methodology for having all group members agree on the structure/design of the module/component before coding it would further boost the potential for work to completed in an efficient manner.
* Often when implementing a new data structure that depends on another more core data structure (queues, arrays, stacks, linkedlists, trees, etc.) we ran into issues with method’s not returning the intended output or just failing in general. Having the capability to solidify the core data structures first and ensure that they function properly, without any errors, and work on any/all examples should’ve been our first priority instead of prioritizing the quantity of our work/progress. Often when situations such as this arose we would spend more time debugging the issues rather than focusing on the actual implementation. If given another opportunity to work on this project we would ensure the core functionality of basic data structures first before moving onto more advanced ones which require the core data structures in some capacity.
* Enabling larger datasets for our tests is also another area of improvement which would be worked on if given another opportunity to start this project again. Specifically for trees, our examples would only have a max height of at most seven levels, which may suffice for basic examples, but really should be tested with tree heights more than seven levels. Most examples will only include a few nodes due to the main user demographic being students starting their data structures adventure for the first time, but if maintaining the core requirement for reusability and generalization of this workspace then real-life examples will consist of many levels and will be more extreme than traditional or previously tested examples.
* Understanding the power (and potential for major mistakes) with the use of git version control proved to be a challenge at the beginning of our development lifecycle. Since we had experience with git in different domain applications we felt our experience was sufficient for this project. The major mistake in our thought process was to assume that we would be able to always handle and understand git. With any new environment comes the need for learning the new components before using them. Now knowing the ins-and-outs of git and GitHub we would approach the project with constant fear/weariness of GitHub and the potential for overwriting code which shouldn’t be overwritten. In short, praise be GitHub, but with great power comes great responsibility.