**Data Structures Workspace User Manual**

* This manual is intended for individuals looking to interact with this project through interacting with the various implementations. We primarily cater this manual to first-time users and those looking to gain a better understanding of certain data structure concepts. A beginning section on getting started helps to better define who this workspace is for, and why it is being developed. Next, we move into a high-level view of the project listing the implementations which are completed and ready to be used, and finally in the last section we individually walk through each package and demonstrate how they are to be used. For a deeper understanding of the design processes behind this project see the Technical/Testing Manual for more information.

**Table of Contents**

Contents

[Getting Started 1](#_Toc98698158)

[Audience 1](#_Toc98698159)

[Purpose 1](#_Toc98698160)

[Use 1](#_Toc98698161)

[Project Outline 1](#_Toc98698162)

[Class and Package Descriptions 1](#_Toc98698163)

[edu.sru.thangiah 1](#_Toc98698164)

[edu.sru.thangiah.abstraction 1](#_Toc98698165)

[edu.sru.thangiah.arrays 1](#_Toc98698166)

[edu.sru.thangiah.datastructures 1](#_Toc98698167)

[edu.sru.thangiah.datastructures.generic 1](#_Toc98698168)

[edu.sru.thangiah.datastructures.generic.linkedlist 1](#_Toc98698169)

[**edu.sru.thangiah.datastructures.generic.matrix** 1](#_Toc98698170)

[edu.sru.thangiah.datastructures.generic.queue 1](#_Toc98698171)

[edu.sru.thangiah.datastructures.generic.stack 1](#_Toc98698172)

[edu.sru.thangiah.datastructures.generic.tree.avltree 1](#_Toc98698173)

[edu.sru.thangiah.datastructures.generic.tree.binarytree 1](#_Toc98698174)

[edu.sru.thangiah.datastructures.generic.tree.heaptree 1](#_Toc98698175)

[edu.sru.thangiah.datastructures.generic.tree.redblacktree 1](#_Toc98698176)

[edu.sru.thangiah.datastructures.hashtable 1](#_Toc98698177)

[edu.sru.thangiah.datastructures.linkedlist 1](#_Toc98698178)

[edu.sru.thangiah.datastructures.matrix 1](#_Toc98698179)

[edu.sru.thangiah.datastructures.queue 1](#_Toc98698180)

[edu.sru.thangiah.datastructures.search.minimax 1](#_Toc98698181)

[edu.sru.thangiah.datastructures.stack 1](#_Toc98698182)

[edu.sru.thangiah.datastructures.tree.avltree 1](#_Toc98698183)

[edu.sru.thangiah.datastructures.tree.binarytree 1](#_Toc98698184)

[edu.sru.thangiah.datastructures.tree.generaltree 1](#_Toc98698185)

[edu.sru.thangiah.datastructures.tree.heaptree 1](#_Toc98698186)

[edu.sru.thangiah.datastructures.tree.redblacktree 1](#_Toc98698187)

[edu.sru.thangiah.datastructures.tree.twothreefourtree 1](#_Toc98698188)

[edu.sru.thangiah.datastructures.vector 1](#_Toc98698189)

[edu.sru.thangiah.inheritance 1](#_Toc98698190)

[edu.sru.thangiah.interfaceex 1](#_Toc98698191)

[edu.sru.thangiah.polymorphism 1](#_Toc98698192)

[edu.sru.thangiah.recursion 1](#_Toc98698193)

[edu.sru.thangiah.sorting 1](#_Toc98698194)

[edu.sru.thangiah.vector 1](#_Toc98698195)

**Table of Figures**

[Figure 1 Diagram of Beginning OOP Examples to Run 1](#_Toc98697882)

[Figure 2 Diagram of Beginning Abstraction Example to Run 1](#_Toc98697883)

[Figure 3 Diagram of Array Examples to Run 1](#_Toc98697884)

[Figure 4 Diagram of Linked List Examples to Run 1](#_Toc98697885)

[Figure 5 Diagram of Matrix Examples to Run 1](#_Toc98697886)

[Figure 6 Diagram of Queue Examples to Run 1](#_Toc98697887)

[Figure 7 Diagram of Stack Examples to Run 1](#_Toc98697888)

[Figure 8 Diagram of AVL Tree Classes to Run 1](#_Toc98697889)

[Figure 9 Diagram of Binary Tree Classes to Run 1](#_Toc98697890)

[Figure 10 Diagram of Heap Tree Classes to Run 1](#_Toc98697891)

[Figure 11 Diagram of Red Black Tree Classes to Run 1](#_Toc98697892)

[Figure 12 Diagram of Hash Table (int) Classes to Run 1](#_Toc98697893)

[Figure 13 Diagram of Single and Double Linked List (int) Classes to Run 1](#_Toc98697894)

[Figure 14 Diagram of Matrix (int) Classes to Run 1](#_Toc98697895)

[Figure 15 Diagram of Queue (int) Classes to Run 1](#_Toc98697896)

[Figure 16 Diagram of Minimax (int) Classes to Run 1](#_Toc98697897)

[Figure 17 Diagram of Stack (int) Classes to be Run 1](#_Toc98697898)

[Figure 18 Diagram of AVL Tree (int) Classes to Run 1](#_Toc98697899)

[Figure 19 Diagram of Binary Tree (int) Classes to Run 1](#_Toc98697900)

[Figure 20 Diagram of General Tree (int) Classes to Run 1](#_Toc98697901)

[Figure 21 Diagram of Heap Tree (int) Classes to Run 1](#_Toc98697902)

[Figure 22 Diagram of Red Black Tree (int) Classes to Run 1](#_Toc98697903)

[Figure 23 Diagram for 2-3-4 Tree (int) Classes to Run 1](#_Toc98697904)

[Figure 24 Diagram of Vector (int) Classes to Run 1](#_Toc98697905)

[Figure 25 Diagram of Inheritance Classes to Run 1](#_Toc98697906)

[Figure 26 Diagram of Interface Classes to Run 1](#_Toc98697907)

[Figure 27 Diagram of Polymorphism Classes to Run 1](#_Toc98697908)

[Figure 28 Diagram of Recursion Classes to Run 1](#_Toc98697909)

[Figure 29 Diagram of Sorting Classes to Run 1](#_Toc98697910)

# Getting Started

## Audience

* This program is intended for students or those interested in learning about common Java data structures. Individuals are not required to have any previous data structures experience, but some programming knowledge is integral towards fully understanding the examples implemented.

## Purpose

* The Data Structures workspace contained in this repository provides a large number of Java implementations for commonly used and studied data structures. The intent is to help guide prospective users trying to learn or brush up on their knowledge of data structures by providing thoroughly documented code with default int and generic implementations. Users can expect to find generic and int implementations of arrays, stacks, matrices, vectors, sorting algorithms, linked lists, queues, binary search trees, heap trees, AVL trees, 2-3-4 trees, redblack trees, and hashing algorithms. Users can expect a wide variety of methods which demonstrate the useability of these data structures and also interface/abstract classes which can be inherited to easily build additional data structures examples.

## Use

* Users are able to run this project as an eclipse workspace, and ideally will use the methods contained in each data structure implementation to test out the functionality of the data structure and further solidify their understanding of how it works. Each data structure is housed in its own Eclipse package ([What is a package?](https://www.w3schools.com/java/java_packages.asp)) and can easily be imported to other implementations to help build an understanding of a particular data structure. There are also core datastructure packages which include the fundamental methods for categories of data structures (lists, trees, etc.). For the user just trying to understand a particular data structure this will not be of much importance but for those wishing to further solidify their understanding by creating their own data structures implementations these packages can be used to simplify the process.

## Project Outline

* 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

# 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)

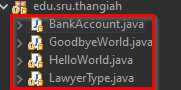


Figure 1 Diagram of Beginning OOP Examples to Run

* 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 2 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)

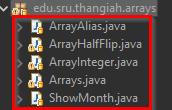


Figure 3 Diagram of Array Examples to Run

* 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)

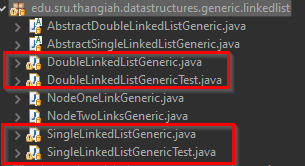


Figure 4 Diagram of Linked List Examples to Run

* 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)

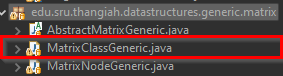


Figure 5 Diagram of Matrix Examples to Run

* 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)



Figure 6 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)

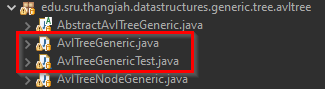


Figure 8 Diagram of AVL Tree Classes to Run

* 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)

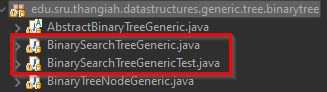


Figure 9 Diagram of Binary Tree Classes to Run

* 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)

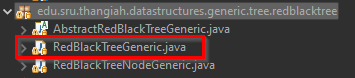


Figure 11 Diagram of Red Black Tree Classes to Run

* 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.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)

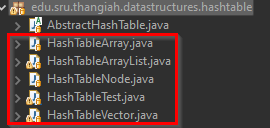


Figure 12 Diagram of Hash Table (int) Classes to Run

* 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)

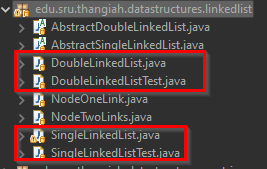


Figure 13 Diagram of Single and Double Linked List (int) Classes to Run

* 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)

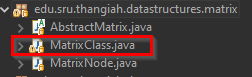


Figure 14 Diagram of Matrix (int) Classes to Run

* 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)

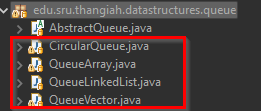


Figure 15 Diagram of Queue (int) Classes to Run

* 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)

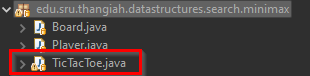


Figure 16 Diagram of Minimax (int) Classes to Run

* 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)

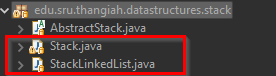


Figure 17 Diagram of Stack (int) Classes to be Run

* 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)

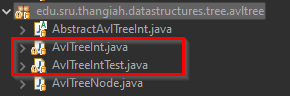


Figure 18 Diagram of AVL Tree (int) Classes to Run

* 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)

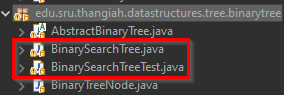


Figure 19 Diagram of Binary Tree (int) Classes to Run

* 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)

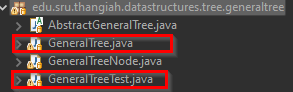


Figure 20 Diagram of General Tree (int) Classes to Run

* 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)

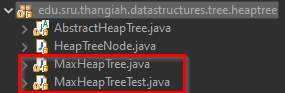


Figure 21 Diagram of Heap Tree (int) Classes to Run

* 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)

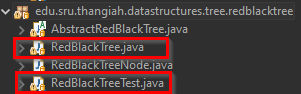


Figure 22 Diagram of Red Black Tree (int) Classes to Run

* 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)

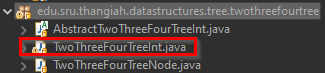


Figure 23 Diagram for 2-3-4 Tree (int) Classes to Run

* 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)

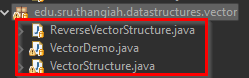


Figure 24 Diagram of Vector (int) Classes to Run

* 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)

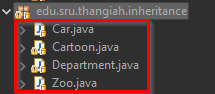


Figure 25 Diagram of Inheritance Classes to Run

* 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 26 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 27 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)



Figure 28 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)

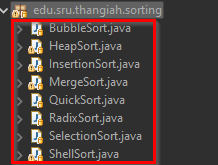


Figure 29 Diagram of Sorting Classes to Run

* 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