**Data Structures Workspace Testing Manual**

* Individuals interested in the type of JUnit tests performed for each data structure, as well as diagrams of the data sets tested for each data structure should use this document. Each data structure which has an implemented test class is contained in this document. Sections are dedicated to the tested data structures and within these sections are the methods that are tested, as well as the type of tests performed for that method. If there are special notes on the tests performed this will be added at the start of the section. Starting on page 5/6 we lay out diagrams of how the data sets tested for each data structure should look to give future developers an easier visual of the various conditions being tested for.

**Table of Contents**

Contents

[List of Tests Performed 2](#_Toc98698431)

[Single Linked List (int) 2](#_Toc98698432)

[Double Linked List (int) 3](#_Toc98698433)

[Single Linked List (Generic) 4](#_Toc98698434)

[Double Linked List (Generic) 4](#_Toc98698435)

[Stack (Int) 4](#_Toc98698436)

[Queue (Int) 4](#_Toc98698437)

[Binary Search Tree (int) 4](#_Toc98698438)

[Binary Search Tree (generic) 8](#_Toc98698439)

[General Tree (int) 12](#_Toc98698440)

[Data Sets Tested 16](#_Toc98698441)

[Single Linked List Test (int) 16](#_Toc98698442)

[Empty List 16](#_Toc98698443)

[Populated List 17](#_Toc98698444)

[Single Node List 17](#_Toc98698445)

[Double Linked List (int) 17](#_Toc98698446)

[Empty List 17](#_Toc98698447)

[Populated List 17](#_Toc98698448)

[Single Node List 18](#_Toc98698449)

[Single Linked List (Generic) 18](#_Toc98698450)

[String List 18](#_Toc98698451)

[Double List 18](#_Toc98698452)

[Double Linked List (Generic) 18](#_Toc98698453)

[String List 18](#_Toc98698454)

[Double List 18](#_Toc98698455)

[Stack (Int) 19](#_Toc98698456)

[Empty Stack 19](#_Toc98698457)

[Single Value Stack 19](#_Toc98698458)

[Populated Stack 19](#_Toc98698459)

[Queue (Int) 20](#_Toc98698460)

[Empty Queue 20](#_Toc98698461)

[Single Value Queue 20](#_Toc98698462)

[Populated Queue 21](#_Toc98698463)

[Binary Trees Tested (int) 21](#_Toc98698464)

[One Node tree 21](#_Toc98698465)

[Complete Tree 22](#_Toc98698466)

[Full Tree 23](#_Toc98698467)

[Right Leaning Tree 24](#_Toc98698468)

[Left Leaning Tree 25](#_Toc98698469)

[Binary Trees Tested (generic) 25](#_Toc98698470)

[One Node tree 25](#_Toc98698471)

[Initialized Tree 26](#_Toc98698472)

[Complete Tree 26](#_Toc98698473)

[Full Tree 27](#_Toc98698474)

[Right Leaning Tree 28](#_Toc98698475)

[Left Leaning Tree 29](#_Toc98698476)

[General Trees tested(int) 29](#_Toc98698477)

[One Node Tree 29](#_Toc98698478)

[Initialized Tree 30](#_Toc98698479)

[All One Child Tree 31](#_Toc98698480)

[Heap Trees Tested (int) 31](#_Toc98698481)

[One Node Tree 32](#_Toc98698482)

[Initialized Tree 32](#_Toc98698483)

[Complete Tree 33](#_Toc98698484)

[Full Tree 34](#_Toc98698485)

[Heap Trees Tested (generic) 34](#_Toc98698486)

[Initialized Tree 35](#_Toc98698487)

[Complete Tree 36](#_Toc98698488)

[Full Tree 37](#_Toc98698489)

**Table of figures**

[Figure 1: Diagram of how tree should look after testing BST insert method 1](#_Toc98145526)

[Figure 2 Diagram of Initialized Tree to showcase how BST should look after testing add method 1](#_Toc98145527)

[Figure 3 Diagram of how tree should look after testing generic BST insert method 1](#_Toc98145528)

[Figure 4 Diagram of how tree should look after testing generic BST add method 1](#_Toc98145529)

[Figure 5 Diagram of how tree should look after testing general tree add method 1](#_Toc98145530)

[Figure 6 Diagram of how tree should look after testing general tree insert method 1](#_Toc98145531)

[Figure 7 Diagram of empty single linked list 1](#_Toc98145532)

[Figure 8 Diagram of initialized single linked list 1](#_Toc98145533)

[Figure 9 Diagram of single linked list with only one node 1](#_Toc98145534)

[Figure 10 Diagram of empty doubly linked list 1](#_Toc98145535)

[Figure 11 Diagram of initialized doubly linked list 1](#_Toc98145536)

[Figure 12 Diagram of doubly linked list with only one node 1](#_Toc98145537)

[Figure 13 Diagram of initialized string single linked list 1](#_Toc98145538)

[Figure 14 Diagram of initialized double single linked list 1](#_Toc98145539)

[Figure 15 Diagram of initialized string doubly linked list 1](#_Toc98145540)

[Figure 16 Diagram of initialized double doubly linked list 1](#_Toc98145541)

[Figure 17 Diagram of empty stack 1](#_Toc98145542)

[Figure 18 Diagram of stack with only one element 1](#_Toc98145543)

[Figure 19 Diagram of initialized stack 1](#_Toc98145544)

[Figure 20 Diagram of empty queue 1](#_Toc98145545)

[Figure 21 Diagram of queue with only one element 1](#_Toc98145546)

[Figure 22 Diagram of initialized queue 1](#_Toc98145547)

[Figure 23 Diagram of binary tree with only one node 1](#_Toc98145548)

[Figure 24 Diagram of binary tree initialized with a few nodes 1](#_Toc98145549)

[Figure 25 Diagram of complete binary tree 1](#_Toc98145550)

[Figure 26 Diagram of full binary tree 1](#_Toc98145551)

[Figure 27 Diagram of a right leaning binary tree 1](#_Toc98145552)

[Figure 28 Diagram of a left leaning binary tree 1](#_Toc98145553)

[Figure 29 Diagram of a one node generic binary tree 1](#_Toc98145554)

[Figure 30 Diagram of an initialized generic binary tree 1](#_Toc98145555)

[Figure 31 Diagram of a complete generic binary tree 1](#_Toc98145556)

[Figure 32 Diagram of a full generic binary tree 1](#_Toc98145557)

[Figure 33 Diagram of a right leaning generic binary tree 1](#_Toc98145558)

[Figure 34 Diagram of a generic left leaning tree 1](#_Toc98145559)

[Figure 35 Diagram of a one node general tree 1](#_Toc98145560)

[Figure 36 Diagram of an initialized general tree 1](#_Toc98145561)

[Figure 37 Diagram of a general tree where each node has only one child 1](#_Toc98145562)

[Figure 38 Diagram of a one node heap tree 1](#_Toc98145563)

[Figure 39 Diagram of an initialized heap tree 1](#_Toc98145564)

[Figure 40 Diagram of a complete heap tree 1](#_Toc98145565)

[Figure 41 Diagram of a full heap tree 1](#_Toc98145566)

[Figure 42 Diagram of a one node generic heap tree 1](#_Toc98145567)

[Figure 43 Diagram of an initialized generic heap tree 1](#_Toc98145568)

[Figure 44 Diagram of a complete generic heap tree 1](#_Toc98145569)

[Figure 45 Diagram of a full generic heap tree 1](#_Toc98145570)

# List of Tests Performed

## Single Linked List (int)

The Single Linked List tests are implemented in order to test each method of the SingleLinkedList.java class. The types of lists created to test each case include an empty list, a populated list, and a list with a single node. The empty list tests adding and removal of nodes when there is no existing data. The populated list interacts with all methods to ensure they are working properly in a traditional scenario. The single node list ensures that removal from the structure resets the structure back to an empty list.

Methods being tested

size(): Initialized and empty linked list with assertEquals

isEmpty(): Initialized and empty linked list are checked with assertTrue and assertFalse

clear(): Initialized list tested after running clear() method with assertTrue

contains(): Initialized list, we assertTrue the contains() method with a value to confirm that the list contains an element.

indexOf(): Initialized list assertEquals of three values and predetermined indexes

addFirst(): Initialized list with a value at the start. We assertEquals that that value’s index is one, then addFirst’s an element and checks with assertEquals that the index of that is one and the other element went up by one.

removeFirst(): initialized list with a value at the start. We assertEquals that the first element of the list is the first one by default. Then we assertEquals that the removeFirst pops that same value. Then, we assertEquals that the element that should take its place has the index 1.

addLast(): Initialized list with a set amount of values. We assertEquals that the last element of the initialized list is the last value we put in it. We then assertEquals that removeLast pops this same value. We then assertFalse that the Initialized list contains this value after removal.

getFirst(): Initialized with six values. We then assertEquals that the getFirst returns the first value we put into the linked list. We then add first another value, and assertEquals that it returns that value. Then, we assertEquals that removeFirst gets rid of that value. We then run that same first assertEquals command after the removeFirst took place, and get the same original value.

getLast(): Initialized with six given. We then assertEquals that the last value in the list is the last one from the setup. We addLast an element and check that the last value is now that element. We removeLast, assertEquals that the element popped, and assertEquals that the getLast value after removeLast is the original one.

deletePrev(): Initialized with six values. We assertEquals that the index of an element in the third position. We then delete the element in the fourth index, and assertEquals that the element in the place that is being removed is the element from the original assertEquals. After running that, we assertFalse that the list contains that element. Lastly, we check that the element at that third index is the one after the one we removed.

getAtIndex(): Initialized with six values. We first assertEquals that the element at index 1 is the first one from setup. Then, we addFirst an element to the linked list. We then assertEquals that the element at index 1 is that new index.

setAtIndex(): Initialized with six values. We assertEquals that the third element in the linked list is the third value from setup. We then setAtIndex(3) an element, and assertEquals that the element after this command at index 3 is that element we inserted.

removeAtIndex(): Initialized with six values. We assertEquals that the third element in the linked list is the third value from setup. We then remove the third element from the linked list and assertEquals that it is the same value that we found earlier. We then assertEquals that the element after the removal is now the element originally in the fourth place. We then assertEquals that the element in the fourth index is the element from setup that follows the successor of the one removed.

addAtIndex(): Initialized with six values. We assertEquals that the third element in the linked list is the third value from setup. We then addAtIndex(3) a new element and assertEquals that it returns that value. We then assertEquals that getAtIndex(3) returns that new element, and getAtIndex(4) returns the element that was originally in that place.

## Double Linked List (int)

The Double Linked List tests are implemented in order to test each method of the DoubleLinkedList.java class. The types of lists created to test each case include an empty list, a populated list, and a list with a single node. These tests are essentially the same as the Single Linked List tests, just more specific to Double Linked List methods and being able to retrieve previous node in the list.

## Single Linked List (Generic)

The Single Linked List tests are implemented in order to test each method of the SingleLinkedListGeneric.java class. The types of lists created include the original integer lists, along with lists to test String data nodes and double data nodes. These added data type lists are tested using the same methods as the integer list.

## Double Linked List (Generic)

The Double Linked List tests are implemented in order to test each method of the DoubleLinkedListGeneric.java class. The types of lists created include the original integer lists, along with lists to test String data nodes and double data nodes. These added data type lists are tested using the same methods as the integer list.

## Stack (Int)

The Stack tests are implemented in order to test each method in the Stack.java class. The types of stacks created in order to test these methods thoroughly include an empty stack, a stack with one node, and a populated stack. The empty stack is used to test adding, removing, and getting nodes or data when there is none. The stack with one node ensures that the stack is reset when removing the single node. The populated stack is tested with all methods and cases for each method.

## Queue (Int)

The Queue tests are implemented in order to test each method in the QueueArray.java class. The types of queues created in order to test these methods thoroughly include an empty queue, a queue with one node, and a populated queue. The empty queue is used to test adding, removing, and getting nodes or data when there is none. The queue with one node ensures that the queue is reset when removing the single node. The populated queue is tested with all methods and cases for each method.

## Binary Search Tree (int)

The Binary Search Tree tests are laid out to test for all the methods in BinarySearchTree.java. The data sets used for our tests include an empty tree, a one node tree, an initialized tree with five nodes, a complete tree, a full tree, an empty tree used to test the add/insert methods, and a right/left leaning tree. Diagrams of how all of these trees look are included in this section. Since an empty tree simply does not have any initialized nodes, we omit the visualization of this tree.

Methods being tested:

* Height()
* Test for all trees to ensure correct value is set for height. Height should be 0 for empty tree
* nodeCount()
* Test for all trees to ensure correct return value of calling nodeCount() method. Node count should be 0 for empty tree.
* degree()
* Test for calling degree method with empty tree (throws exception)
* Otherwise, test for all trees to ensure correct return value of calling degree() method. All cases (node has 0 , 1, and 2 children) are tested.
* isFull()
* Test for calling method with empty tree (throws exception)
* Otherwise, test for all trees to ensure correct return value of calling isFull() method. The initializedTree is a full tree, as well as the full and one node trees, whereas all other trees tested are not full (complete tree, right/left leaning trees)
* isComplete()
* Test for calling method with empty tree (throws exception)
* Otherwise, test for all trees to ensure correct return value of calling isComplete() method. The initialized, complete, full, and one node trees are all complete, but the left and right leaning trees are not.
* isIncomplete()
* Test for calling method with empty tree (throws exception)
* Otherwise, test for all the trees to ensure correct return value of calling isIncomplete() method. The left and right leaning trees are incomplete, whereas all the other trees tested are not incomplete.
* isRoot()
* Test for calling method when tree is uninitialized (throws exception)
* Test for passing the root node to the method (returns true)
* Test for passing any other node that isn’t the root node (returns false)
* isParent()
* Test for calling method with empty tree (throws exception)
* Test for passing node which has children (returns true)
* Test for passing any other node which doesn’t have children, such as the leaf node (returns false)
* isChild()
* Test for calling method with empty tree (throws exception)
* Test for passing node which has a parent (returns true)
* Test for passing any other node (root) which isn’t a child (returns false)
* isLeaf()
* Test for calling method with empty tree (throws exception)
* Test for passing node which has no children (returns true)
* Test for passing any other node (root, parent, child w/ children) which has children (returns false)
* insert(int value)
* First we add a simply initialized BST (exactly similar to initialized tree tested) and ensure that the insert method creates the desired BST. See below for what the tree looks like:

**Diagram

Description automatically generated**

Figure 1: Diagram of how tree should look after testing BST insert method

* Height of tree should be 3
* Node count should be 5
* In order traversal should be 75, 80, 90, 100, 120
* Test for adding a node that already exists in the bst (returns false)
* search(int value)
* Ensure method returns -1 when we search for a value that doesn’t exist or is being searched in an uninitialized tree
* Otherwise, ensure that we return the value of the node that has been successfully searched for.
* Test for a node that exists in the bst, and make sure that the return value is that node
* locateNode(BinaryTreeNode root, int value)
* Similar to search tests:
* Ensure method returns -1 when we search for a value that doesn’t exist or is being searched in an uninitialized tree
* Otherwise, ensure that we return the value of the node that has been successfully searched for.
* Test for a node that exists in the bst, and make sure that the return value is that node
* predecessor(BinaryTreeNode treeNode)
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, ensure that method returns the node which is the predecessor for its target.
* successor(BinaryTreeNode treeNode)
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, ensure that method returns the node which is the successor for its target.
* getLargest(BinaryTreeNode root)
* Test for calling method when BST is uninitialized (throw exception)
* Test for calling method in any initialized tree and ensure that the correct value is returned.
* getSmallest(BinaryTreeNode root)
* Test for calling method when BST is uninitialized (throw exception)
* Test for calling method in any initialized tree and ensure that the correct value is returned.
* getSecondLargest(BinaryTreeNode root)
* Test for calling method when BST is uninitialized (throw exception)
* Test for calling method when there is only one node in the tree (throw exception)
* Test for calling method in any initialized tree and ensure that the correct value is returned.
* getSecondSmallest(BinaryTreeNode root)
* Test for calling method when BST is uninitialized (throw exception)
* Test for calling method when there is only one node in the tree (throw exception)
* Test for calling method in any initialized tree and ensure that the correct value is returned.
* remove(int value)
* Test for calling remove method when BST is uninitialized (throw exception)
* Test for calling remove method for node that doesn’t exist (return -1)
* Test removing any node from an initialized tree and test for the successful removal of that node by traversing through the tree.
* RotateLeft(BinaryTreeNode pivot)
* Test for calling the method when BST is uninitialized (throw exception)
* Test for calling method with initialized BST that is unbalanced. (after calling the method traverse through the tree and ensure successful output)
* RotateRight(BinaryTreeNode pivot)
* Test for calling the method when BST is uninitialized (throw exception)
* Test for calling method with initialized BST that is unbalanced. (after calling the method traverse through the tree and ensure successful output)
* inOrder(BinaryTreeNode root)
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, perform inorder traversal for each of the examples tested and compare the output to the expected result.
* preOrder(BinaryTreeNode root)
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, perform prorder traversal for each of the examples tested and compare the output to the expected result.
* postOrder(BinaryTreeNode root)
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, perform postorder traversal for each of the examples tested and compare the output to the expected result.
* levelOrder(BinaryTreeNode root)
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, perform levelorder traversal for each of the examples tested and compare the output to the expected result.
* add(int value)
* First we add a simply initialized BST (exactly similar to initialized tree tested) and ensure that the insert method creates the desired BST. See below for what the tree looks like:

**Diagram

Description automatically generated**

Figure 2 Diagram of Initialized Tree to showcase how BST should look after testing add method

* Height of tree should be 3
* Node count should be 5
* In order traversal should be 75, 80, 90, 100, 120
* Test for adding a node that already exists in the bst (returns false)
* getRoot()
* Test for calling the method for an uninitialized tree (returns -1)
* Otherwise, ensure that the return value is equal to the expected result of calling getRoot()
* isEmpty()
* Test for calling the method for an uninitialized tree (returns true)
* Otherwise, test for any initialized tree (returns false)
* clear()
* Test for calling clear method for any initialized tree, after calling the method the nodecount, height, and any search for a node should all be comparable to calling these methods for an empty tree.

## Binary Search Tree (generic)

The Binary Search Tree tests are laid out to test for all the methods in BinarySearchTreeGeneric.java. Data sets visualized in the below section are similar in fashion to the ones described in **Binary Search Tree (int)** except now our complete tree is initialized with all nodes being floats, our full tree is initialized as double, and our right leaning tree is initialized as string.

* Height()
* Test for all trees to ensure correct value is set for height. Height should be 0 for empty tree
* nodeCount()
* Test for all trees to ensure correct return value of calling nodeCount() method. Node count should be 0 for empty tree.
* degree()
* Test for calling degree method with empty tree (throws exception)
* Otherwise, test for all trees to ensure correct return value of calling degree() method. All cases (node has 0 , 1, and 2 children) are tested.
* isFull()
* Test for calling method with empty tree (throws exception)
* Otherwise, test for all trees to ensure correct return value of calling isFull() method. The initializedTree is a full tree, as well as the full and one node trees, whereas all other trees tested are not full (complete tree, right/left leaning trees)
* isComplete()
* Test for calling method with empty tree (throws exception)
* Otherwise, test for all trees to ensure correct return value of calling isComplete() method. The initialized, complete, full, and one node trees are all complete, but the left and right leaning trees are not.
* isIncomplete()
* Test for calling method with empty tree (throws exception)
* Otherwise, test for all the trees to ensure correct return value of calling isIncomplete() method. The left and right leaning trees are incomplete, whereas all the other trees tested are not incomplete.
* isRoot()
* Test for calling method when tree is uninitialized (throws exception)
* Test for passing the root node to the method (returns true)
* Test for passing any other node that isn’t the root node (returns false)
* isParent()
* Test for calling method with empty tree (throws exception)
* Test for passing node which has children (returns true)
* Test for passing any other node which doesn’t have children, such as the leaf node (returns false)
* isChild()
* Test for calling method with empty tree (throws exception)
* Test for passing node which has a parent (returns true)
* Test for passing any other node (root) which isn’t a child (returns false)
* isLeaf()
* Test for calling method with empty tree (throws exception)
* Test for passing node which has no children (returns true)
* Test for passing any other node (root, parent, child w/ children) which has children (returns false)
* insert(int value)
* First we add a simply initialized BST (exactly similar to initialized tree tested) and ensure that the insert method creates the desired BST. See below for what the tree looks like:

**Diagram

Description automatically generated**

Figure 3 Diagram of how tree should look after testing generic BST insert method

* Height of tree should be 3
* Node count should be 5
* In order traversal should be 75, 80, 90, 100, 120
* Test for adding a node that already exists in the bst (returns false)
* search(int value)
* Ensure method returns -1 when we search for a value that doesn’t exist or is being searched in an uninitialized tree
* Otherwise, ensure that we return the value of the node that has been successfully searched for.
* Test for a node that exists in the bst, and make sure that the return value is that node
* locateNode()
* Similar to search tests:
* Ensure method returns -1 when we search for a value that doesn’t exist or is being searched in an uninitialized tree
* Otherwise, ensure that we return the value of the node that has been successfully searched for.
* Test for a node that exists in the bst, and make sure that the return value is that node
* predecessor()
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, ensure that method returns the node which is the predecessor for its target.
* successor()
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, ensure that method returns the node which is the successor for its target.
* getLargest()
* Test for calling method when BST is uninitialized (throw exception)
* Test for calling method in any initialized tree and ensure that the correct value is returned.
* getSmallest()
* Test for calling method when BST is uninitialized (throw exception)
* Test for calling method in any initialized tree and ensure that the correct value is returned.
* getSecondLargest()
* Test for calling method when BST is uninitialized (throw exception)
* Test for calling method when there is only one node in the tree (throw exception)
* Test for calling method in any initialized tree and ensure that the correct value is returned.
* getSecondSmallest()
* Test for calling method when BST is uninitialized (throw exception)
* Test for calling method when there is only one node in the tree (throw exception)
* Test for calling method in any initialized tree and ensure that the correct value is returned.
* remove()
* Test for calling remove method when BST is uninitialized (throw exception)
* Test for calling remove method for node that doesn’t exist (return -1)
* Test removing any node from an initialized tree and test for the successful removal of that node by traversing through the tree.
* RotateLeft)
* Test for calling the method when BST is uninitialized (throw exception)
* Test for calling method with initialized BST that is unbalanced. (after calling the method traverse through the tree and ensure successful output)
* RotateRight()
* Test for calling the method when BST is uninitialized (throw exception)
* Test for calling method with initialized BST that is unbalanced. (after calling the method traverse through the tree and ensure successful output)
* inOrder()
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, perform inorder traversal for each of the examples tested and compare the output to the expected result.
* preOrder()
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, perform prorder traversal for each of the examples tested and compare the output to the expected result.
* postOrder()
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, perform postorder traversal for each of the examples tested and compare the output to the expected result.
* levelOrder()
* Test for calling method when BST is uninitialized (throw exception)
* Otherwise, perform levelorder traversal for each of the examples tested and compare the output to the expected result.
* add()
* First we add a simply initialized BST (exactly similar to initialized tree tested) and ensure that the insert method creates the desired BST. See below for what the tree looks like:

**Diagram

Description automatically generated**

Figure 4 Diagram of how tree should look after testing generic BST add method

* Height of tree should be 3
* Node count should be 5
* In order traversal should be 75, 80, 90, 100, 120
* Test for adding a node that already exists in the bst (returns false)
* getRoot()
* Test for calling the method for an uninitialized tree (returns -1)
* Otherwise, ensure that the return value is equal to the expected result of calling getRoot()
* isEmpty()
* Test for calling the method for an uninitialized tree (returns true)
* Otherwise, test for any initialized tree (returns false)
* clear()
* Test for calling clear method for any initialized tree, after calling the method the nodecount, height, and any search for a node should all be comparable to calling these methods for an empty tree.

## General Tree (int)

The general tree tests were conducted for all methods in the GeneralTree.java class found in edu.sru.thangiah.datastructure.tree.generaltree. Example trees tested for were for a one node tree, an initialized tree with multiple nodes and also multiple children for a given parent, and a all one child tree which has 0 or 1 children for any given level in the tree.

Methods being tested:

* Height()
* Test for all trees to ensure correct value is set for height. Height should be 0 for empty tree
* nodeCount()
* Test for all trees to ensure correct return value of calling nodeCount() method. Node count should be 0 for empty tree.
* isEmpty()
* Test for all trees to ensure correct return value of calling isEmpty() method. For empty tree we should expect true, while all other trees will have a return value of false
* degree()
* Test for calling the method when a given tree is uninitialized (throw exception)
* Otherwise, test for all different conditions (node has 0, 1, 2, 3, n children)
* isRoot()
* Test for calling method when tree is uninitialized (throws exception)
* Test for passing the root node to the method (returns true)
* Test for passing any other node that isn’t the root node (returns false)
* isParent()
* Test for calling method with empty tree (throws exception)
* Test for passing node which has children (returns true)
* Test for passing any other node which doesn’t have children, such as the leaf node (returns false)
* isChild()
* Test for calling method with empty tree (throws exception)
* Test for passing node which has a parent (returns true)
* Test for passing any other node (root) which isn’t a child (returns false)
* isLeaf()
* Test for calling method with empty tree (throws exception)
* Test for passing node which has no children (returns true)
* Test for passing any other node (root, parent, child w/ children) which has children (returns false)
* getLargest()
* Test for calling method when tree is uninitialized (Throws exception)
* Test for calling the method for all the example trees tested and compare the return value with the expected result.
* getSmallest()
* Test for calling method when tree is uninitialized (Throws exception)
* Test for calling the method for all the example trees tested and compare the return value with the expected result.
* getSecondLargest()
* Test for calling method when tree is empty and also when tree only has one node (throws exception)
* Test for calling the method for all the example trees tested and compare the return value with the expected result.
* getSecondSmallest()
* Test for calling method when tree is empty and also when tree only has one node (throws exception)
* Test for calling the method for all the example trees tested and compare the return value with the expected result.
* add()
* Test with adding nodes to a tree that already exist, the target value which doesn’t exist, or adding an additional root node
* Test the add method with an uninitialized tree and build a tree similar to the initialized tree tested:

**Diagram

Description automatically generated**

Figure 5 Diagram of how tree should look after testing general tree add method

* Ensure the successful completion of this tree by comparing the result of inOrder traversal with the expected result
* insert()
* Test with inserting nodes to a tree that already exist, the target value which doesn’t exist, or adding an additional root node
* Test the insert method with an uninitialized tree and build a tree similar to the initialized tree tested:

**Diagram

Description automatically generated**

Figure 6 Diagram of how tree should look after testing general tree insert method

* Ensure the successful completion of this tree by comparing the result of inOrder traversal with the expected result
* remove()
* Test for removing a node from an uninitialized tree
* Test for removing the root node from any initialized tree and then ensure the node was removed by comparing the result of inorder traversal with the expected result.
* Test for removing a parent node from any initialized tree and then ensure the node was removed by comparing the result of inorder traversal with the expected result.
* Test for removing a leaf node from any initialized tree and then ensure the node was removed by comparing the result of inorder traversal with the expected result.
* Test for removing the last node from a tree and ensure that the tree is now uninitialized
* search()
* Test for searching for a value in an empty tree (throws exception)
* Test for searching for a value that doesn’t exist in an initialized tree (returns null)
* Otherwise, test all the initialized trees and compare the result of searching for a node (root, parent, child, leaf) with the expected result (should return the located node)
* locateNode()
* Test for searching for a value in an empty tree (throws exception)
* Test for searching for a value that doesn’t exist in an initialized tree (returns null)
* Otherwise, test all the initialized trees and compare the result of searching for a node (root, parent, child, leaf) with the expected result (should return the located node)
* inOrder()
* Test for calling method when general tree is uninitialized (throw exception)
* Otherwise, perform inorder traversal for each of the examples tested and compare the output to the expected result.
* preOrder()
* Test for calling method when general tree is uninitialized (throw exception)
* Otherwise, perform prorder traversal for each of the examples tested and compare the output to the expected result.
* postOrder()
* Test for calling method when general tree is uninitialized (throw exception)
* Otherwise, perform postorder traversal for each of the examples tested and compare the output to the expected result.
* levelOrder()
* Test for calling method when general tree is uninitialized (throw exception)
* Otherwise, perform levelorder traversal for each of the examples tested and compare the output to the expected result.
* getRoot()
* Test for calling the method for an uninitialized tree (returns -1)
* Otherwise, ensure that the return value is equal to the expected result of calling getRoot()

# Data Sets Tested

The data sets and tree examples used in our JUnit tests are standardized across all data structure Junit tests to ensure uniformity among all the tests. Subsections for each data structure describes the kinds of initialized data structures tested for as well as diagrams depicting what these structures should look like.

## Single Linked List Test (int)

### Empty List

**Shape

Description automatically generated with medium confidence**

Figure 7 Diagram of empty single linked list

### Populated List

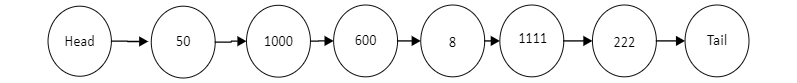
****

Figure 8 Diagram of initialized single linked list

### Single Node List

**Shape

Description automatically generated with medium confidence**

Figure 9 Diagram of single linked list with only one node

## Double Linked List (int)

### Empty List

**Shape

Description automatically generated with medium confidence**

Figure 10 Diagram of empty doubly linked list

### Populated List

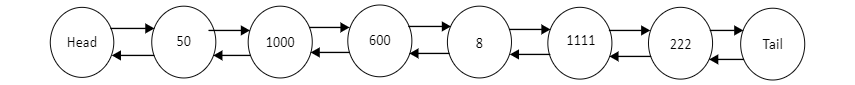
****

Figure 11 Diagram of initialized doubly linked list

### Single Node List

**Shape

Description automatically generated with medium confidence**

Figure 12 Diagram of doubly linked list with only one node

## Single Linked List (Generic)

### String List

**Shape

Description automatically generated with medium confidence**

Figure 13 Diagram of initialized string single linked list

### Double List

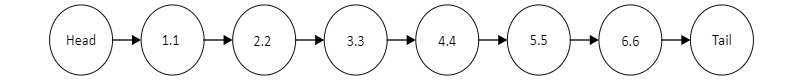
****

Figure 14 Diagram of initialized double single linked list

## Double Linked List (Generic)

### String List

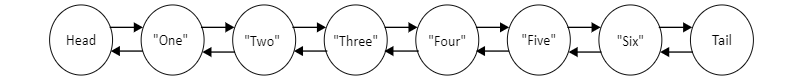
****

Figure 15 Diagram of initialized string doubly linked list

### Double List

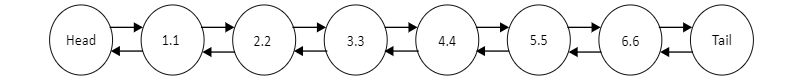
****

Figure 16 Diagram of initialized double doubly linked list

## Stack (Int)

### Empty Stack

**Shape

Description automatically generated with medium confidence**

Figure 17 Diagram of empty stack

### Single Value Stack

**Shape

Description automatically generated with medium confidence**

Figure 18 Diagram of stack with only one element

### Populated Stack

**Shape

Description automatically generated with medium confidence**

Figure 19 Diagram of initialized stack

## Queue (Int)

### Empty Queue

**Shape

Description automatically generated with low confidence**

Figure 20 Diagram of empty queue

### Single Value Queue

**Shape

Description automatically generated with low confidence**

Figure 21 Diagram of queue with only one element

### Populated Queue

Shape

Description automatically generated with medium confidence

Figure 22 Diagram of initialized queue

## Binary Trees Tested (int)

### One Node tree

**Diagram

Description automatically generated**

Figure 23 Diagram of binary tree with only one node

#### Initialized tree

**Diagram

Description automatically generated**

Figure 24 Diagram of binary tree initialized with a few nodes

### Complete Tree

**Diagram

Description automatically generated**

Figure 25 Diagram of complete binary tree

### Full Tree

**Diagram

Description automatically generated**

Figure 26 Diagram of full binary tree

### Right Leaning Tree

**Diagram

Description automatically generated**

Figure 27 Diagram of a right leaning binary tree

### Left Leaning Tree

**Diagram

Description automatically generated**

Figure 28 Diagram of a left leaning binary tree

## Binary Trees Tested (generic)

### One Node tree

**Diagram

Description automatically generated**

Figure 29 Diagram of a one node generic binary tree

### Initialized Tree

**Diagram

Description automatically generated**

Figure 30 Diagram of an initialized generic binary tree

### Complete Tree

**Diagram

Description automatically generated**

Figure 31 Diagram of a complete generic binary tree

### Full Tree

**Diagram

Description automatically generated**

Figure 32 Diagram of a full generic binary tree

### Right Leaning Tree

**Diagram

Description automatically generated**

Figure 33 Diagram of a right leaning generic binary tree

### Left Leaning Tree

**Diagram

Description automatically generated**

Figure 34 Diagram of a generic left leaning tree

## General Trees tested(int)

### One Node Tree

**Graphical user interface, diagram

Description automatically generated**

Figure 35 Diagram of a one node general tree

### Initialized Tree

**Diagram

Description automatically generated**

Figure 36 Diagram of an initialized general tree

### All One Child Tree

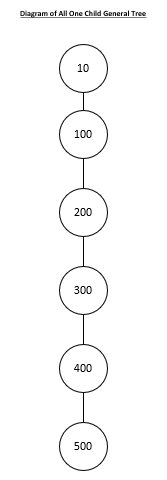
****

Figure 37 Diagram of a general tree where each node has only one child

## Heap Trees Tested (int)

* Since a heap tree is a balanced binary tree our examples do not include right/left leaning trees.

### One Node Tree

**Graphical user interface, application

Description automatically generated**

Figure 38 Diagram of a one node heap tree

### Initialized Tree

**Diagram

Description automatically generated**

Figure 39 Diagram of an initialized heap tree

### Complete Tree

**Diagram

Description automatically generated**

Figure 40 Diagram of a complete heap tree

### Full Tree

**Diagram

Description automatically generated**

Figure 41 Diagram of a full heap tree

## Heap Trees Tested (generic)

* Since a heap tree is a balanced binary tree our examples do not include right/left leaning trees.

**One Node Tree**

Graphical user interface, application

Description automatically generated

Figure 42 Diagram of a one node generic heap tree

### Initialized Tree

**Diagram

Description automatically generated**

Figure 43 Diagram of an initialized generic heap tree

### Complete Tree

**Diagram

Description automatically generated**

Figure 44 Diagram of a complete generic heap tree

### Full Tree

**Diagram

Description automatically generated**

Figure 45 Diagram of a full generic heap tree