# Binary Search Trees

## Introduction

Binary search trees are data structures that store nested nodes. Each node stores data and can have up to two nested nodes: a left child node and a right child node. Additionally, binary search trees are sorted, meaning that the left child node is always less than the right child node. Binary search trees have the following operations: insert, delete, maximum, and traverse. First, the insertion operation creates a new node using specific data. Second, the deletion operation deletes a specific node, replacing it if necessary. Third, the maximum operation finds the maximum value in the tree. Finally, the traverse operation helps you traverse the tree's data in a specific order. I implemented a binary search tree with these operations.

## Structure

The BinarySearchTree's structure consists of nested nodes. BinarySearchTree has methods for interacting with the nodes and contains the head node, which is the main node. The properties and methods are described below:

## BinarySearchTree

The Binary Search Tree Class

**Kind**: global class

* [BinarySearchTree](#binarysearchtree)
  + [new BinarySearchTree(head)](#new_BinarySearchTree_new)
  + [.minimum](#BinarySearchTree+minimum) ⇒ DataNode.<Data> | null
  + [.maximum](#BinarySearchTree+maximum) ⇒ DataNode.<Data> | null
  + [.head](#binarysearchtreehead)
  + [.head](#binarysearchtreehead)
  + [.getIfShouldInsertOnSide(node, data, side)](#BinarySearchTree+getIfShouldInsertOnSide) ⇒ boolean
  + [.tryInsertingNodeOnSide(node, side, data)](#BinarySearchTree+tryInsertingNodeOnSide)
  + [.insert(data, startNode)](#BinarySearchTree+insert)
  + [.replaceWithNode(nodeToReplace, node)](#BinarySearchTree+replaceWithNode)
  + [.getSideTraverser(side)](#BinarySearchTree+getSideTraverser) ⇒ function
  + [.getMaximumValue()](#BinarySearchTree+getMaximumValue) ⇒ Data | null
  + [.replaceNodeWithMinimumOfRightTree(nodeToDelete)](#X476707ff03de4c389331407a701c90e2a6789eb)
  + [.replaceNodeWithChild(nodeToDelete)](#BinarySearchTree+replaceNodeWithChild)
  + [.findParentNode(data, node)](#BinarySearchTree+findParentNode)
  + [.removeNodeFromParent(nodeToDelete)](#BinarySearchTree+removeNodeFromParent)
  + [.deleteNode(nodeToDelete)](#BinarySearchTree+deleteNode)
  + [.traverse(startNode)](#BinarySearchTree+traverse) ⇒ IterableIterator.<Data>

### new BinarySearchTree(head)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| head | Data | Generic type describing the type of DataNode |

**Example**

new BinarySearchTree<number>()

### binarySearchTree.minimum ⇒ DataNode.<Data> | null

Gets the minimum node of a tree.

**Kind**: instance property of [BinarySearchTree](#binarysearchtree)  
**Returns**: DataNode.<Data> | null - : Returns the minimum node or null

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
|  | node | The node to start traversing from |

**Example**

tree.minimum()

### binarySearchTree.maximum ⇒ DataNode.<Data> | null

Gets the maximum node of a tree.

**Kind**: instance property of [BinarySearchTree](#binarysearchtree)  
**Returns**: DataNode.<Data> | null - : Returns the maximum node or null

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
|  | node | The node to start traversing from |

**Example**

tree.maximum()

### binarySearchTree.head

**Kind**: instance property of [BinarySearchTree](#binarysearchtree)  
**Example**

tree.head

### binarySearchTree.head

**Kind**: instance property of [BinarySearchTree](#binarysearchtree)  
**Example**

tree.head = null

### binarySearchTree.getIfShouldInsertOnSide(node, data, side) ⇒ boolean

Returns whether it should insert on a side depending on the node's value

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| node | node | The node to insert on |
| data | data | The data to insert |
| side | side | "left" or "right" |

**Example**

this.getIfShouldInsertOnSide(node, data, side)

### binarySearchTree.tryInsertingNodeOnSide(node, side, data)

Gets whether a node should be inserted on a side, then tries to insert it.

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| node | node | The node to insert on |
| side | data | The data to insert |
| data | side | "left" or "right" |

**Example**

this.tryInsertingNodeOnSide(startNode, 'left', data)

### binarySearchTree.insert(data, startNode)

Creates a new node with data and inserts it into the tree.

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| data | data | The data to insert |
| startNode | startNode | The node to start the insertion process on |

**Example**

tree.insert(100)

### binarySearchTree.replaceWithNode(nodeToReplace, node)

Replaces a node

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| nodeToReplace | nodeToReplace | The node to replace |
| node | node | The node to replace with |

**Example**

tree.insert(100)  
tree.insert(99)  
tree.replaceWithNode(tree.head, tree.head.left)

### binarySearchTree.getSideTraverser(side) ⇒ function

Gets a function that traverses the tree in a given direction.

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)  
**Returns**: function - - a traverser function

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| side | side | Either "left" or "right" |

**Example**

tree.getSideTraverser('left')

### binarySearchTree.getMaximumValue() ⇒ Data | null

Gets the maximum value of a tree.

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)  
**Returns**: Data | null - : Returns the maximum value or null  
**Example**

tree.getMaximumValue()

### binarySearchTree.replaceNodeWithMinimumOfRightTree(nodeToDelete)

Replaces the node that will be deleted with the minimum of the right side.

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| nodeToDelete | nodeToDelete | The node to delete |

**Example**

tree.replaceNodeWithMinimumOfRightTree(tree.head)

### binarySearchTree.replaceNodeWithChild(nodeToDelete)

Replaces a node with either its left or right child, depending on which one exists.

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| nodeToDelete | nodeToDelete | The node to delete |

**Example**

tree.replaceNodeWithChild(tree.head)

### binarySearchTree.findParentNode(data, node)

Finds the parent node of a node with specific data

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| data | data | The data to find |
| node | node | The node to begin traversing from |

**Example**

tree.insert(90)  
tree.insert(100)  
// Returns the head node (the one with 90)  
tree.findParentNode(100)

### binarySearchTree.removeNodeFromParent(nodeToDelete)

Finds the parent node and removes its reference to a specific child

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| nodeToDelete | nodeToDelete | The node to delete |

**Example**

tree.insert(100)  
tree.insert(90)  
tree.removeNodeFromParent(tree.head.left)

### binarySearchTree.deleteNode(nodeToDelete)

Deletes a node, replacing it if necessary.

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| nodeToDelete | nodeToDelete | The node to delete |

**Example**

tree.insert(100)  
tree.deleteNode(tree.head)

### binarySearchTree.traverse(startNode) ⇒ IterableIterator.<Data>

A generator method to traverse the tree. Yields the data, then left, then right.

**Kind**: instance method of [BinarySearchTree](#binarysearchtree)  
**Returns**: IterableIterator.<Data> - : An iterator with the node data

|  |  |  |
| --- | --- | --- |
| Param | Type | Description |
| startNode | startNode | The node to start traversing from |

**Example**

const generator = tree.traverse()  
const { value } = generator.next()

## Analysis

### Insertion

The time complexity for insertions is O(n) in the worst case. The insert method only uses recursion when inserting data on a side and the node already has children. Since it always calls itself with a nested node when using recursion, the maximum number of times it can call itself is n. Therefore, the time complexity is O(n).

The insertion operation has an average time complexity of O(log n). When using recursion, the operation will compare the data to the current node's data. Then, it will choose to call itself with either the left or right side (but not both). If the tree is balanced, then the operation will disregard half of the nodes on each recursive iteration. Therefore, the time complexity is O(log n).

The best-case time complexity is constant because the operation will never call itself if the given node is null.

### Deletion

The deletion operation has a worst-case time complexity of O(n). The deleteNode function must be called with a valid node. When it has two children, the function will replace the node with the minimum value on the right side. Therefore, it must find the minimum value by calling minimum, which calls itself to traverse the left subtree until it reaches a leaf node. Since the tree is not guaranteed to be balanced, minimum has a time complexity of about O(n). When deleteNode is called with a node that has either a left or right child (but not both), it will replace the node with one of its children, which takes constant time. Finally, when deleteNode is called with a node with no children, it will find the parent node by calling findParentNode. This method can call itself as long as it still has children, which means it has a time complexity of about O(n). Therefore, the worst-case time complexity is O(n).

This operation has an average time complexity of O(log n) because both findParentNode and minimum disregard half of the nodes on each recursive call, given a balanced tree.

The best-case time complexity is constant because the operation will not call itself if the number of children is 0 or 1.

### Maximum

The getMaximumValue method uses the same principles as the minimum function. Since the tree is not guaranteed to be balanced, it has a worst-case time complexity of O(n).

This operation has an average time complexity of O(log n) because it disregards half of the nodes (one side) for each recursive call, given a balanced tree.

The best-case time complexity is constant because the operation will never call itself if the given node is null.

### Traverse

The traverse generator is designed to traverse through each node in the tree. Since it calls itself with a nested node until it reaches a leaf, traverse has a time complexity of O(n).

Since it is designed to iterate over each node, this operation has an average time complexity of O(n).

The best-case time complexity is constant because the operation will never call itself if the given node is null.

## Commands

Install packages

npm install

Run the tests

npm run test