

Content

1. Introduction
2. Binary search tree
 - i. Implementation
 - a. `getMinimum()` / `getMaximum()`
 - b. `getSuccessor()`
 - c. Main methods
 - a. `insert()`
 - b. `search()`
 - c. `delete()`
 - ii. Self-balancing binary search tree
 - a. Implementation
 - a. `rotateLeft()`
 - b. `rotateRight()`
 - b. AVL tree
 - a. `rebalance()`
 - b. Tests
 - a. For `insert()`
 - b. For `search()`
 - c. For `delete()`
 - c. Splay tree
 - a. `splay()`
 - b. Tests
 - a. For `insert()`
 - b. For `search()`
 - c. For `delete()`
 - d. Comparing summary
3. Hash table
 - i. Hashing
 - ii. Open addressing
 - a. Implementation
 - a. `class Node`
 - b. `getIndex()`
 - c. `get()`
 - d. `put()`
 - e. `remove()`
 - f. Tests
 - iii. Closed addressing
 - a. Implementation
 - a. `class Node`
 - b. `getIndex()`
 - c. `get()`


- d. put()
 - e. remove()
 - f. Test
- iv. Comparing summary

Introduction

In this assignment we needed to implement two binary search trees and two hash table variations.

I chose to implement the AVL and Splay trees because they are the most common and popular. For hash tables, there are only two: with open addressing and with closed addressing.

For the best testing I decided to move away from the idea of testing with `Date.now()` , because it's silly and can lead to incorrect results. Therefore, I decided to use `jmh-core` , since I chose to use Java.

 In my implementation I tested up to one million, because, with numbers higher than that - I got an error `java.lang.OutOfMemoryError: native memory extradition` . In addition, I think tests above this number are useless, because it's better to specify performance on a set of tests, not on how big those tests are.

Binary search trees

A binary search tree (BST) is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. In other words, it is a tree data structure where each node has at most two children, which are referred to as the left child and the right child. The left subtree of a node contains only nodes with keys lesser than the node's key while the right subtree of a node contains only nodes with keys greater than the node's key.

Binary search trees are used to quickly find a value in a large set of values. They are particularly useful for searching through sorted data.

Implementation

| | |
|--------------|--|
| getMinimum() | <pre>protected Node getMinimum(Node node) { while (node.left != null) { node = node.left; } return node; }</pre> |
| | |

| | |
|----------------|---|
| getMaximum() | <pre>protected Node getMaximum(Node node) { while (node.right != null) { node = node.right; } return node; }</pre> |
| getSuccessor() | <pre>protected Node getSuccessor(Node node) { // if there is right branch, then successo // subtree if (node.right != null) { return getMinimum(node.right); } else { // otherwise it is the lowest anc // ancestor of node Node currentNode = node; Node parentNode = node.parent; while (parentNode != null && currentNo // go up until we find parent that // subtree. currentNode = parentNode; parentNode = parentNode.parent; } return parentNode; } }</pre> |