Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

Data structures and algorithms

Assignment 1 – Search in dynamic sets

Petra Miková

Petra Miková
ID: 120852

Contents

Petra Miková
ID: 120852

Aims and objectives

The aim of this assignment is to implement and then compare 4 implementations of data structures in terms of the effectiveness of insert, delete and search operations in different situations:

- implementation of a binary search tree (BVS) with any balancing algorithm, e.g. AVL, Red-Black Trees, (2,3) Trees, (2,3,4) Trees, Splay Trees, ...
- second implementation of BVS with a different balancing algorithm than in the previous point
- implementation of a hash table with collision resolution of your choice (the hash table size adjustment must also be implemented)
- second implementation of the hash table with collision resolution in a different way than in the previous point

In my case, I decided to implement an AVL and Red-Black tree, where the integer element represents both the key and the value, and then a hash table with separate chaining and quadratic probing, which have a string key and integer value. For all the implementations I have used Java programming language and the IntelliJ IDE. I will be comparing the trees and hash tables separately, as they have different data types of keys.

Regarding the testing, the expected average time complexity for both the balanced binary trees is O(logn) and for both the hash tables, the expected time complexity is O(1). So the testing should provide us with a logarithmic curve in trees testing and a linear curve in hash table testing.

Petra Miková
ID: 120852

Implementation of AVL binary search tree

Description

In this implementation of BST, I decided to use the AVL (Adelson-Velsky and Landis) balancing algorithm. The node of an AVL tree has all the properties of a standard binary search tree node, but it also has one additional attribute, which is the height of the node. That is because the AVL balancing algorithm is based on the fact that the tree is only balanced if balance factor of each node is in the interval from -1 to 1 (integers only). If balance factor of any node is not from the interval after insertion or deletion, the tree becomes unbalanced and specific rotations must be performed. The balance factor of a node in this case means the difference of height of its left child and height of its right child. In all three functionalities (insert, delete, search), I used recursive approach.

The class for constructing a node in AVL tree:

```
static class AVLTreeNode{
    int element;
    int height;
    AVLTreeNode left;
    AVLTreeNode right;

    public AVLTreeNode(int element)
    {
        this.left = null;
        this.right = null;
        this.height = 1;
        this.element = element;
    }
}
```

The class has 4 attributes – the element (also being used as a key), its height, and pointers for both left and right children. The constructor for a new node sets its children pointers to null, its height to 1 by default, and sets the element to the one which is being inserted.

Insert

To insert a node into an AVL tree, I first implemented a public method insert that takes an integer element as input and calls the private method insert with the root node of the AVL tree and the element to be inserted as arguments. The reason I did this is to make inserting in the main function easier, as with the implemented public method we can just call AVLTree.insert(element) without having to care about any other parameters. The private insert method then recursively inserts the new element into the AVL tree, and then rebalances the tree using the BalanceAVLTree method (is explained later in this documentation).

```
public void insert(int element) {
    root = insert(root, element);
}

private AVLTreeNode insert(AVLTreeNode root, int element) {
    if (root == null) {
        return new AVLTreeNode(element);
    }
    if (element < root.element) {
        root.left = insert(root.left, element);
    } else if (element > root.element) {
        root.right = insert(root.right, element);
    } else {
        return root;
    }

    root.height = 1 + Math.max(getHeight(root.left),getHeight(root.right));
    int balance = getBalance(root);
    return BalanceAVLTree(root, balance);
}
```

If the tree is empty (root is null), then the function returns a new root. Otherwise, it recursively traverses the tree and inputs it purely based on the standard BST insert rules. After that, I update the height of the root of the subtree where new node has been inserted as a left or right child. The height of a node in an AVL tree is defined as the maximum number of edges between that node and its deepest leaf node. To get the correct height of the root, we also must add 1 to this maximum, as the root node itself is one edge above its deepest leaf node. After that, I get the balance of the root to check whether imbalance has happened after inserting a new node. Eventually, the BalanceAVLTree function takes care of all the balancing in the AVL tree.

Delete

Same as in insert method, firstly I implemented a public method delete that takes an integer element as input and calls the private method delete with the root node of the AVL tree and the element to be inserted as arguments for the exact same reason. The private delete method recursively traverses the tree and deletes the node based on its children. Then there is the same balancing scenario as in insert method.

```
public void delete(int element) {
    root = delete(root, element);
}

private AVLTreeNode delete(AVLTreeNode root, int element) {
    if (root == null) {
        return root;
    }
    if (element < root.element) {
        root.left = delete(root.left, element);
    } else if (element > root.element) {
        root.right = delete(root.right, element);
    } else if ((root.left == null) && (root.right != null)) {
        root = root.right;
    } else if ((root.right == null) && (root.left != null)) {
        root = root.left;
    } else if ((root.left == null) && (root.right == null)) {
        root = null;
    } else {
        AVLTreeNode node = Min(root.right);
        root.element = node.element;
        root.right = delete(root.right, node.element);
    }
    if (root == null) {
        return root;
    }

    root.height = 1 + Math.max(getHeight(root.left),getHeight(root.right));
    int balance = getBalance(root);
    return BalanceAVLTree(root, balance);
}
```

In this method, if root is null, meaning that tree is empty, it immediately returns null and that is it. If tree is not empty, then there are three cases possible: the node to be deleted has one child, no children, or two children. Firstly, the function traverses the tree recursively to get to the node that is about to be deleted. After the node is found, it is crucial to check which case is happening. If node only has right child, we simply bypass the node which we want to delete and set it directly to this right child. The same scenario goes for case where node only has left child. If node to be deleted has no children, we simply set it to null. The case of node having both children is a bit trickier. At first, we need to find the smallest inorder successor in the right subtree of the node to be deleted. Then, we set the element of this inorder successor to be the element of the node to bypass the deleted node. After that, we delete the successor. These steps could unbalance our tree, so we update the node height, check its balance, and balance the tree so the AVL tree rules are maintained.

Helper functions

1. BalanceAVLTree

This method is called in both insert and delete functions, and it takes the node and its balance as parameters and performs the needed rotations in the tree in order to keep it balanced. There are four cases of rotations: right rotation, left rotation, right-left rotation, and left-right rotation. It also updates the height of the node as it changes throughout the process of balancing.

```
private AVLTreeNode BalanceAVLTree(AVLTreeNode node, int balance) {
    //1. Left Left case
    if (balance > 1 && (getHeight(node.left.left) >=
getHeight(node.left.right))) {
        node = AVLTreeRightRotation(node);
    }
    //2. Right Right case
    else if (balance < -1 && (getHeight(node.right.right) >=
getHeight(node.right.left))) {
        node = AVLTreeLeftRotation(node);
    }
    //3. Left Right case
    else if (balance > 1 && (getHeight(node.left.left) <
getHeight(node.left.right))) {
        node.left = AVLTreeLeftRotation(node.left);
        node = AVLTreeRightRotation(node);
    }
    //4. Right Left case
    else if (balance < -1 && (getHeight(node.right.right) <
getHeight(node.right.left))) {
        node.right = AVLTreeRightRotation(node.right);
        node = AVLTreeLeftRotation(node);
    }
    return node;
}
```

In the first case, the left subtree is heavier, and the height of the left subtree's left child is greater than or equal to the height of its right child. According to the rules of AVL tree, we have to perform a right rotation on the node. In second case, the right subtree is heavier, and the height of the right subtree's right child is greater than or equal to the height of its left child. In this case, we have to perform a left rotation on the node. In the third case, the left subtree is again heavier, but the height of the left subtree's left child is less than the height of its right child. This means we must first perform a left rotation on the left child of the node, and then a right rotation on the node itself. In the last case, the right subtree is heavier, but the height of the right subtree's right child is less than the height of its left child, meaning that we perform a right rotation on the right child of the node, and then a left rotation on the node itself.

2. getHeight

Aim of this helper function is to avoid getting the NullPointerException when trying to access the height of a null node in other functions. If a node is null, it returns its height as 0, else it just returns the height of non-null node.

3. getBalance

Aim of this helper function is to again avoid getting errors if we tried to get balance of a null node by accessing its height. That is why if node is null, this function simply returns 0, and in other case it returns the balance of the node, which is the difference between height of its left child and height of its right child.

4. AVLTreeRightRotation

This function performs needed steps to right rotate with a given node in order to keep tree balanced.

```
private AVLTreeNode AVLTreeRightRotation(AVLTreeNode node) {
    AVLTreeNode leftChild = node.left;
    AVLTreeNode rightGrandchild = leftChild.right;
    leftChild.right = node;
    node.left = rightGrandchild;
    node.height = 1 + Math.max(getHeight(node.left),getHeight(node.right));
    leftChild.height = 1 +
Math.max(getHeight(leftChild.left),getHeight(leftChild.right));
    return leftChild;
}
```

To perform a right rotation in AVL tree, firstly the left child of the node is stored into its own variable leftChild. Then we need to get the right grandchild of the given node. After this, a rotation is performed, by setting node to be the right child of left child of the node and updating the left child of the node to right grandchild. The rotations made heights of the nodes change, so we recalculate these and return the new root of the subtree, which is the leftChild node.

5. AVLTreeLeftRotation

This function performs needed steps to left rotate with a given node in order to keep tree balanced.

```
private AVLTreeNode AVLTreeLeftRotation(AVLTreeNode node) {
    AVLTreeNode rightChild = node.right;
    AVLTreeNode leftGrandchild = rightChild.left;
    rightChild.left = node;
    node.right = leftGrandchild;
    node.height = 1 + Math.max(getHeight(node.left),getHeight(node.right));
    rightChild.height = 1 + Math.max(getHeight(rightChild.left),
getHeight(rightChild.right));
    return rightChild;
}
```

To perform a left rotation in AVL tree, we need to store the right child of the node into a variable rightChild, and then store this right child's left child into leftGrandchild variable, which represents the relationship between this node and the node which was passed through the function parameter. Then the rotation itself is performed, by setting the left child of this grandchild to be the original node, and then the original node's right child to be the left grandchild. Followed by essential recalculation of the nodes' heights, finally a new root of the subtree, which is the rightChild node, is returned.

6. Min

This helper function is used in the delete function, where we have to find the smallest inorder successor of a node which is about to be deleted. It simply lets the node enter a while loop that continues until node's left child is not null, so until it reaches the leftmost node in the subtree. Inside the loop it updates node to be its own left child, and after exiting the while loop, it returns the new node, which is now the node with the minimum value in the subtree.

Search

To make search in the main class easier, I implemented a public method search, which has a node called found, to which we store returned node from the private method search. That way, we can simply call AVLTree.search(element) in main. The search algorithm in AVL tree is the same as you would use for standard BST, as there is no balancing required.

```
public void search(int element){
    AVLTreeNode found = search(root, element);
}

private AVLTreeNode search(AVLTreeNode root, int element){
    if (root == null) {
        return null;
    }
    if (element < root.element) {
        return search(root.left, element);
    } else if (element > root.element) {
        return search(root.right, element);
    } else {
        return root;
    }
}
```

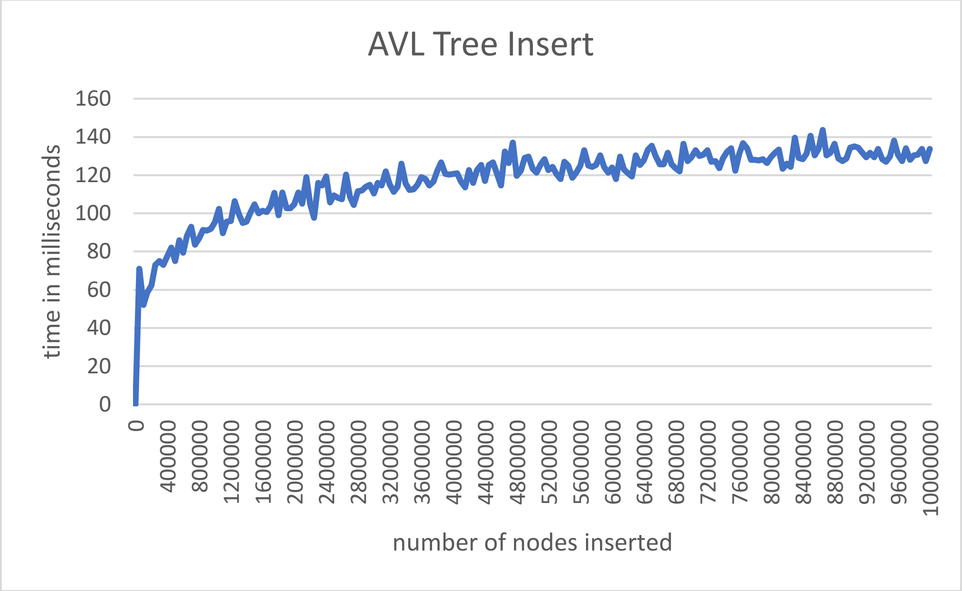
The private method returns null if the tree is empty, otherwise it recursively traverses the AVL tree until the node we are looking for is found. If the node is found, it returns it.

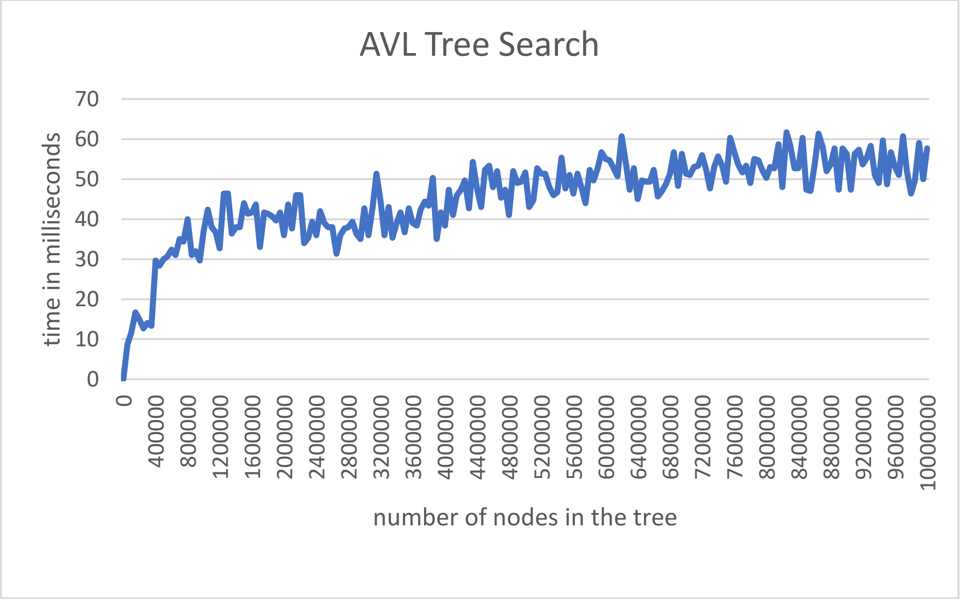
Testing

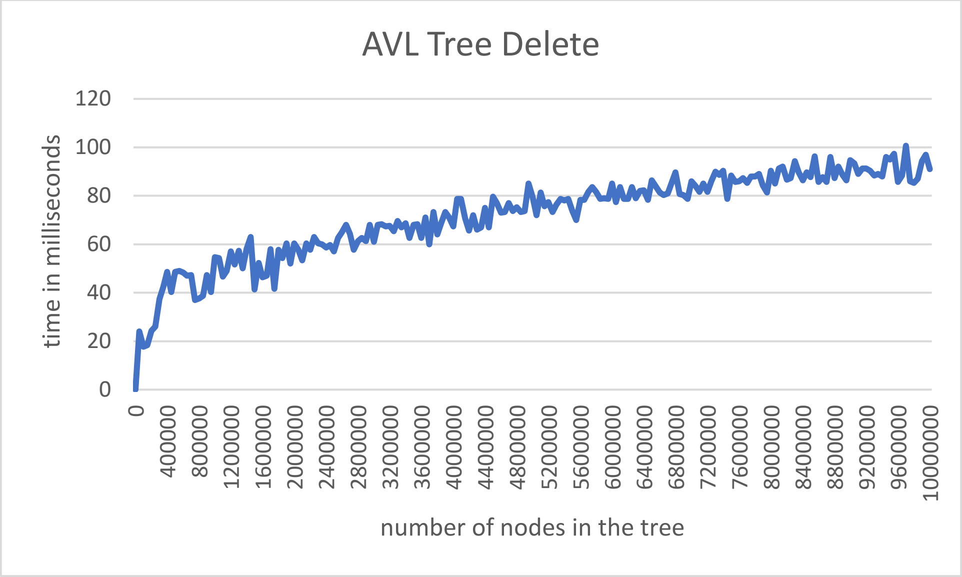
To test the time complexity of my AVL Tree implementation, I have done mulitple tests and represented the results in a graph form. For insert, I sequentially inserted 50K nodes up to the amount of 10 millions. For both search and delete, I then searched for or removed 50K nodes in every interval to get the most unbiased results when comparing all the three functions. For every function, I ran the test three times and then averaged the results and represented them in a graph.

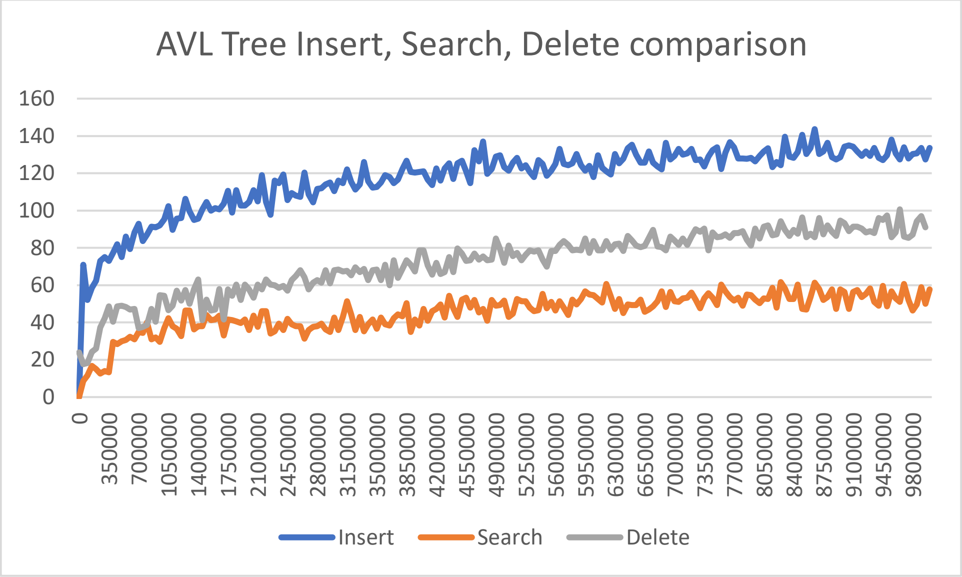
Results

All of the functionalities – insert, search, and delete have the time complexity that results in a logarithmic graph.









We can see that the search is the fastest, which is probably result of the fact that searching in AVL tree does not require any balancing. The delete is a bit faster than insert, and as we can see in the graph, insert takes the most time.

Implementation of Red-Black binary search tree

Description

In the second BST implementation, I decided to use the Red-Black algorithm, which “colors” the node either red or black. The node of the Red-Black tree has a few more attributes than the standard BST, and these are the parent pointer and boolean variable Red, which stores the color of the node. Besides from all the standard BST features, the Red-Black tree must follow these rules:

- the root must be black
- children of red node must be black
- every leaf (null child) must be black
- all the leaves have the same black depth

The tree is self-balancing, and the balancing is performed after every insertion or deletion, which ensures that the tree is always perfectly balanced. To implement insert and delete, I decided to use iterative approach in this case. The search remains the same as for AVL tree, as there is no balancing required when just looking for a specific node.

```
static class RedBlackTreeNode{
    int element;
    boolean Red;
    RedBlackTreeNode left;
    RedBlackTreeNode right;
    RedBlackTreeNode parent;

    public RedBlackTreeNode(int element){
        this.element = element;
        this.Red = true;
        this.left = this.right = this.parent = null;
    }
```

The class RedBlackTreeNode has 5 attributes – the element (an integer also used as a key), Boolean variable Red to set color of the node, and three pointers to node's children and parent. The constructor set every new node's element to inserted element, its color to red as every new node is colored red by default, and all the pointers to null.

Insert

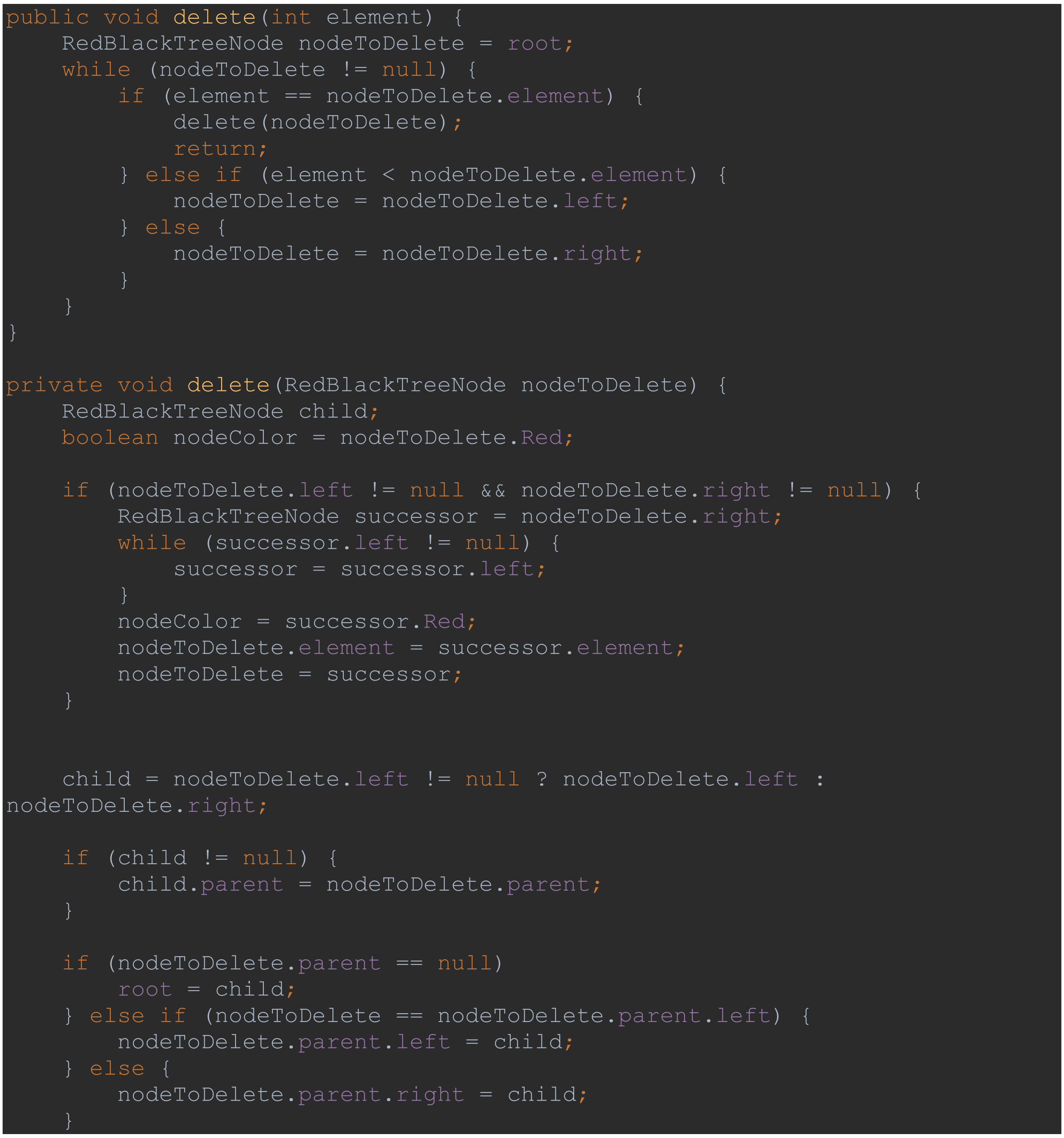
The insert method firstly proceeds to iteratively insert a node as it would be inserted into a standard BST, but after that, the parent pointer of the node has to be updated as well. Lastly, an InsertBalanceRedBlackTree method is called (which is explained later in this documentation), to maintain the perfect balance in Red-Black tree according to the rules.

```
public void insert(int element) {
    RedBlackTreeNode newRedBlackTreeNode = new RedBlackTreeNode(element);
    RedBlackTreeNode parent = null;
    RedBlackTreeNode current = root;
    while (current != null) {
        parent = current;
        if (element < current.element) {
            current = current.left;
        } else {
            current = current.right;
        }
    }

    newRedBlackTreeNode.parent = parent;

    if (parent == null) {
        root = newRedBlackTreeNode;
    } else if (element < parent.element) {
        parent.left = newRedBlackTreeNode;
    } else {
        parent.right = newRedBlackTreeNode;
    }

    InsertBalanceRedBlackTree(newRedBlackTreeNode);
}
```

The insert method first creates a red colored new node. Then we declare two new variables parent and current, and set current to root in order to traverse the tree in the while loop to find the leaf node where the new node should be inserted. In the while loop, we also update the parent to be the current node in every iteration. After that, we set this parent to be the new node's parent. Now we do the standard BST insertion, where we also check for the case where tree does not exist. Lastly the function to balance the tree is called after every insertion.

Delete

The implementation in Red-Black tree is definitely a lot more complex than in AVL tree. Firstly, I created a public method delete, which simply traverses a tree in a while loop to get to the node we want to delete, and if the node has been found, it calls the private delete function, which contains the deletion algorithm. At the end of the private delete method, a function DeleteBalanceRedBlackTree is called, which performs the necessary rotations to keep the tree balanced.



```
public void delete(int element) {
    RedBlackTreeNode nodeToDelete = root;
    while (nodeToDelete != null) {
        if (element == nodeToDelete.element) {
            delete(nodeToDelete);
            return;
        } else if (element < nodeToDelete.element) {
            nodeToDelete = nodeToDelete.left;
        } else {
            nodeToDelete = nodeToDelete.right;
        }
    }
}

private void delete(RedBlackTreeNode nodeToDelete) {
    RedBlackTreeNode child;
    boolean nodeColor = nodeToDelete.Red;

    if (nodeToDelete.left != null && nodeToDelete.right != null) {
        RedBlackTreeNode successor = nodeToDelete.right;
        while (successor.left != null) {
            successor = successor.left;
        }
        nodeColor = successor.Red;
        nodeToDelete.element = successor.element;
        nodeToDelete = successor;
    }


    child = nodeToDelete.left != null ? nodeToDelete.left :
nodeToDelete.right;

    if (child != null) {
        child.parent = nodeToDelete.parent;
    }

    if (nodeToDelete.parent == null)
        root = child;
    } else if (nodeToDelete == nodeToDelete.parent.left) {
        nodeToDelete.parent.left = child;
    } else {
        nodeToDelete.parent.right = child;
    }
    if (nodeColor == false) {
        if (child != null && child.Red) {
            child.Red = false;
        } else {
            DeleteBalanceRedBlackTree(child, nodeToDelete.parent);
        }
    }
}
```

As I mentioned before, the public method delete just traverses the tree in order to find the node that we want to delete and calls the private delete function with this node as a parameter.

In the private delete method, we deal with all the possible cases that could happen: node with two, one, or no children, or if the node is the root of the tree. We store the color of the node we want to delete into a variable nodeColor and create node child. Firstly, we handle the case where node has two children. Here we need to find the node's successor, which is the node with next highest value ant replace the element of the node we want to delete with the successor's element. Then we remove this successor node. Next, we handle the two cases of node having one or no child. To do this, we use the child node created at the beginning of the method, and store either the left or right child of the node to be deleted. If a child is present in the node, we set the this node's parent to the child's parent. We handle a case of node to be deleted being the root, and we simply let the child be the root. Otherwise, we update the appropriate child pointer of the node's parent to point to the child. This also ensures that if no children were present in the node, the children pointer of parent will be set to null. Eventually, we balance the tree, but only if the node deleted was black, as from the Red-Black tree rules we know that if it was red, no balancing is needed. If the child node is red, we simply recolor it black, and in other case, we call DeleteBalanceRedBlackTree method, which handles all the rotations essential for keeping the tree balanced.

Helper functions

1. getUncle

A helper function to get an uncle of a node which is used in the balancing method after node insertion in Red-Black tree. An uncle of a node is the sibling of the parent of the node. We declare the node's grandparent as well here, because if it is null, then there is no possibility that uncle exists, so we simply return null. If grandparent exists, then we set its child which is not the node's parent to be the node's uncle and return this uncle.

2. RedBlackTreeRightRotation

This function performs needed steps to right rotate a given node in order to keep tree balanced.

```
private void RedBlackTreeRightRotation(RedBlackTreeNode node) {
    RedBlackTreeNode leftChild = node.left;
    node.left = leftChild.right;
    if (leftChild.right != null) {
        leftChild.right.parent = node;
    }
    leftChild.parent = node.parent;
    if (node.parent == null) {
        root = leftChild;
    } else if (node == node.parent.left) {
        node.parent.left = leftChild;
    } else {
        node.parent.right = leftChild;
    }
    leftChild.right = node;
    node.parent = leftChild;
}
```

Firstly, the left child of the node is stored into a variable leftChild, and then the current left child of the node is replaced with this left child's right child. If this right child exists, then we set the original node to be its parent. Then, we set the parent of the left child to the parent of the original node. If the node was the root of the tree, we simply let its left child be the root. Otherwise, if the node was the left child of its parent, we let the node's left child be this child. If it was a right child, we do the same thing, but set the left child of node to be the right child of node' s parent. Eventually we set the original node to be its left child's right child, and its parent to be this left child. After that, the rotation is done according to the Red-Black tree properties.

3. RedBlackTreeLeftRotation

This function performs needed steps to right rotate a given node in order to keep tree balanced.

```
private void RedBlackTreeLeftRotation(RedBlackTreeNode node) {
    RedBlackTreeNode rightChild = node.right;
    node.right = rightChild.left;
    if (rightChild.left != null) {
        rightChild.left.parent = node;
    }
    rightChild.parent = node.parent;
    if (node.parent == null) {
        root = rightChild;
    } else if (node == node.parent.left) {
        node.parent.left = rightChild;
    } else {
        node.parent.right = rightChild;
    }
    rightChild.left = node;
    node.parent = rightChild;
}
```

The approach is the same as in the right rotation. We store the right child of the original node into a rightChild variable. Then we set the left child of rightChild to be original node's right child, which moves the left subtree of rightChild to the right subtree of original node. If the left child of this right child of the original node exists, we set the original node to be its new parent. Then, we set the parent of the right child to the parent of the original node. If the node was the root of the tree, we simply let its right child be the root. Otherwise, if the node was the left child of its parent, we let the node's left child be this child. If it was a right child, we do the same thing, but set the right child of node to be the right child of node' s parent. Eventually we set the original node to be its right child's left child, and its parent to be this right child. After that, the rotation is done according to the Red-Black tree properties.

4. InsertBalanceRedBlackTree

This method is called after the insertion of a node into the Red-Black tree to maintain its balancing properties. I implemented it in a way that there are 6 cases which can happen and which require specific steps to regain balance.

Case 1: node is the root of the tree
Solution: simply recolor it black

Case 2: parent of the node is black
Solution: no balancing needed

Case 3: parent and the uncle of the node are both red
Solution: color them black and move the tree up

```
RedBlackTreeNode parent = node.parent;
RedBlackTreeNode uncle = getUncle(node);

if (uncle != null && uncle.Red) {
    parent.Red = false;
    uncle.Red = false;
    parent.parent.Red = true;
    InsertBalanceRedBlackTree(parent.parent);
    return;
}
```

As mentioned, we recolor the parent and uncle nodes to black and also we set the grandparent node to red. Then, we recursively call the InsertBalanceRedBlackTree method on the grandparent node to handle the possibility of the grandparent node violating the Red-Black tree's properties. This keeps iterating until the uncle node is red, if it is finally not, we return and end the balancing.

Case 4: the node is a right child and its parent is a left child
Solution: left rotation and then update node to be its own left child

Case 5: the node is a left child and its parent is a right child
Solution: right rotation and then update node to be its own right child

Case 6: the node's parent is red, but the uncle is black
Solution: recolor and rotate

```
parent = node.parent;
RedBlackTreeNode grandparent = parent.parent;
parent.Red = false;
grandparent.Red = true;

if (node == parent.left && parent == grandparent.left) {
    RedBlackTreeRightRotation(grandparent);
} else {
    RedBlackTreeLeftRotation(grandparent);
}
```

As mentioned, we recolor parent to black, and grandparent to red. Then we check whether this node is a left child or a right child of its parent and also if the parent is the left child or right child of its parent. According to these conditions, suitable rotations are performed and thus the tree is again balanced.

5. DeleteBalanceRedBlackTree

This method is called after the deletion of a node into the Red-Black tree to maintain its balancing properties. The reason why we have two separate balancing functions is because in the contrast to AVL tree, an red-black tree requires different balancing steps after insertion and after deletion. Again, I implemented this method in a way that there are mulitple cases which have to be handled in order to keep the tree balanced. All the cases are handled inside a while loop, which starts at the deleted node, and loops until the node is the root or a red node. After this loop, we color the deleted node black, according to the Red-Black tree properties.

I divided the code inside the while loop into two main cases, and these are if a deleted node is a right child or a left child. These contain the same algorithm and cases, but personalized based on the node position.

The node is a left child:
Case 1: the sibling of the deleted node is red
Solution: recolor parent and sibling, and rotate left the parent

Case 2: both children of the sibling of the deleted node are black
Solution: recolor sibling and move up the tree

Case 3: the sibling of the deleted node has a black right child and red left child
Solution: recolor sibling and its left child, rotate right and update sibling

Case 4: sibling is black, and its right child is red
Solution: recolor parent and sibling, and rotate left

The node is a right child:
Case 1: the sibling of the deleted node is red
Solution: recolor parent and sibling, and rotate right the parent

Case 2: both children of the sibling of the deleted node are black
Solution: recolor sibling and move up the tree

Case 3: the sibling of the deleted node has a black left child and red right child
Solution: recolor sibling and its right child, rotate left and update sibling

Case 4: sibling is black, and its left child is red
Solution: recolor parent and sibling, and rotate right

Search

The search implementation is made of a public and private method, where public method with only element parameter calls the private one. The search algorithm in Red-Black tree is the same as you would use for standard BST, as there is no balancing required.

```
public void search(int element){
    RedBlackTreeNode found = search(root, element);
}
```

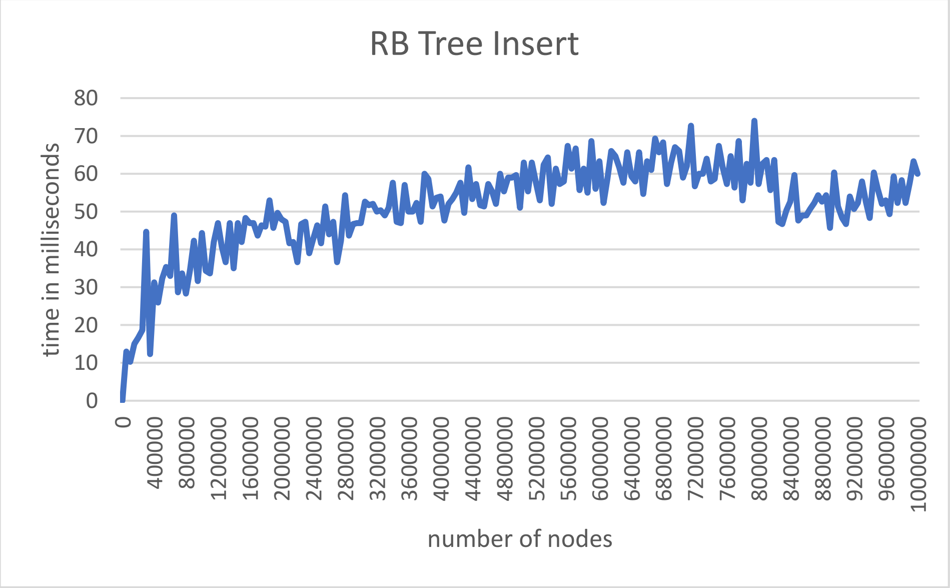
```
private RedBlackTreeNode search(RedBlackTreeNode root, int element){
    if (root == null) {
        return null;
    }
    if (element < root.element) {
        return search(root.left, element);
    } else if (element > root.element) {
        return search(root.right, element);
    } else {
        return root;
    }
}
```

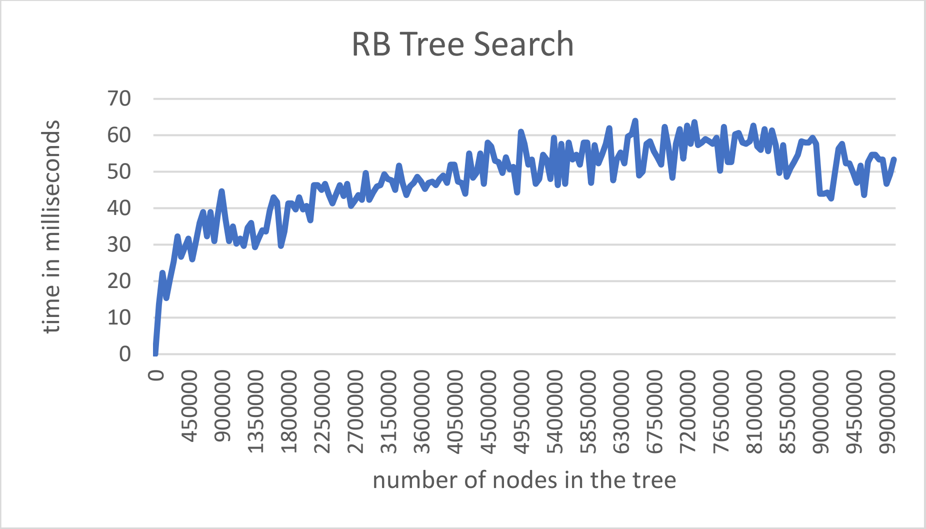
The private method returns null if the tree is empty, otherwise it recursively traverses the Red-Black tree until the node we are looking for is found. If the node is found, it returns it.

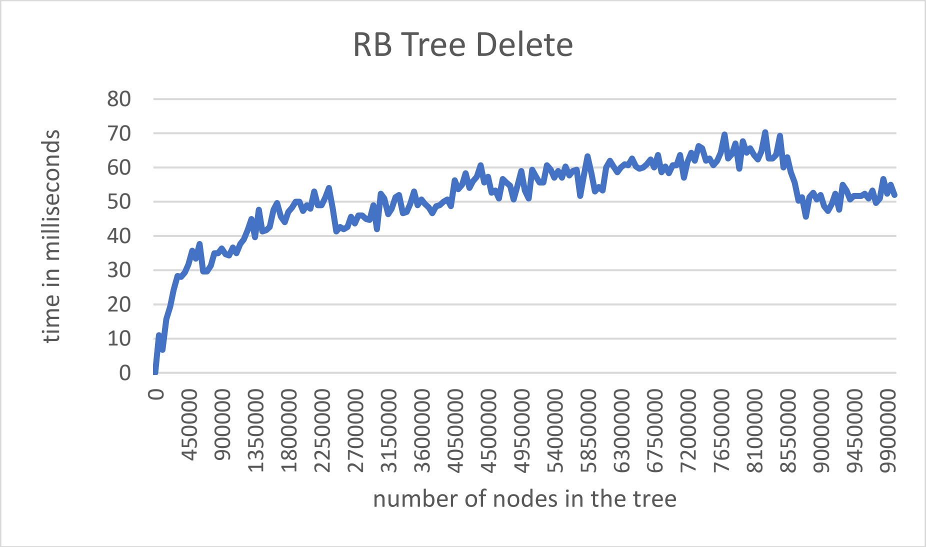
Testing

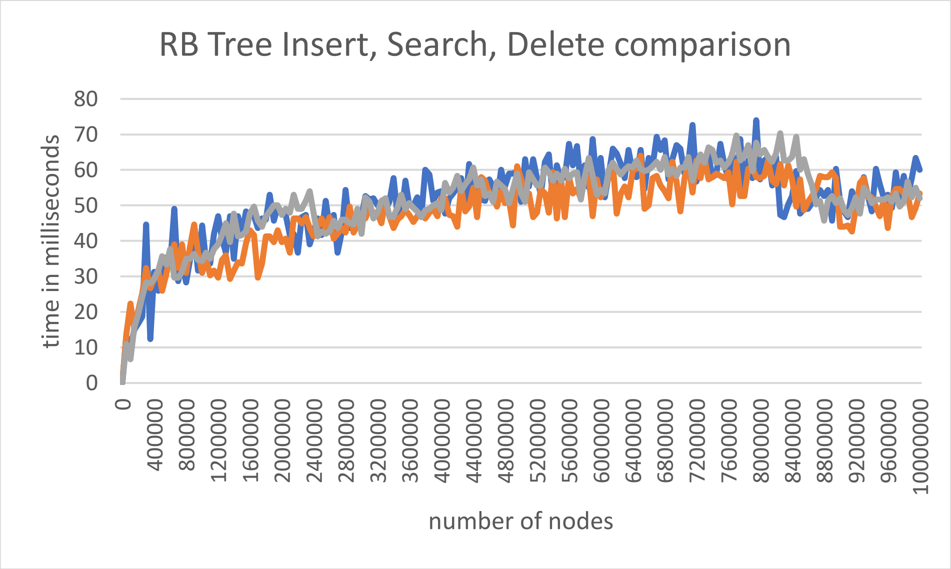
For testing my Red Black tree implementation, I used the exact same approach as with the AVL tree. To test insert, I have been inserting nodes in 50K intervals from 0 to 10 millions. For search and delete, I searched for or deleted 50K nodes from each interval to test how much time it takes when the number of nodes in the tree is growing.

Results









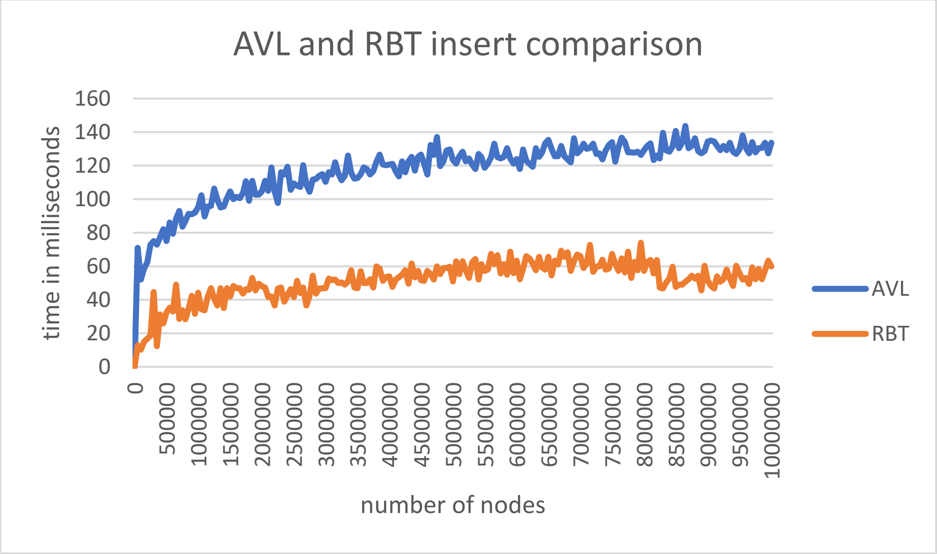
From the last graph we can see that in a Red Black tree, all the three functions took almost the same time, with the search function being slightly faster than others.

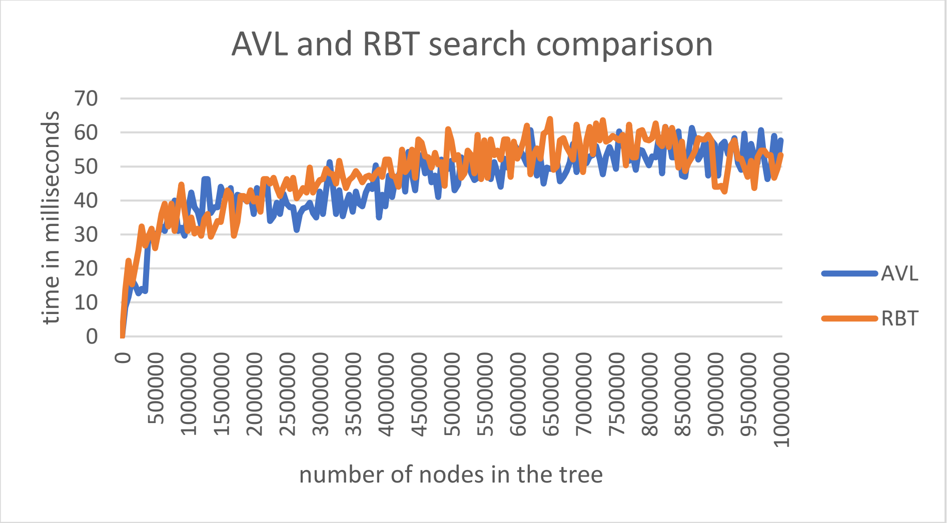
Petra Miková
ID: 120852

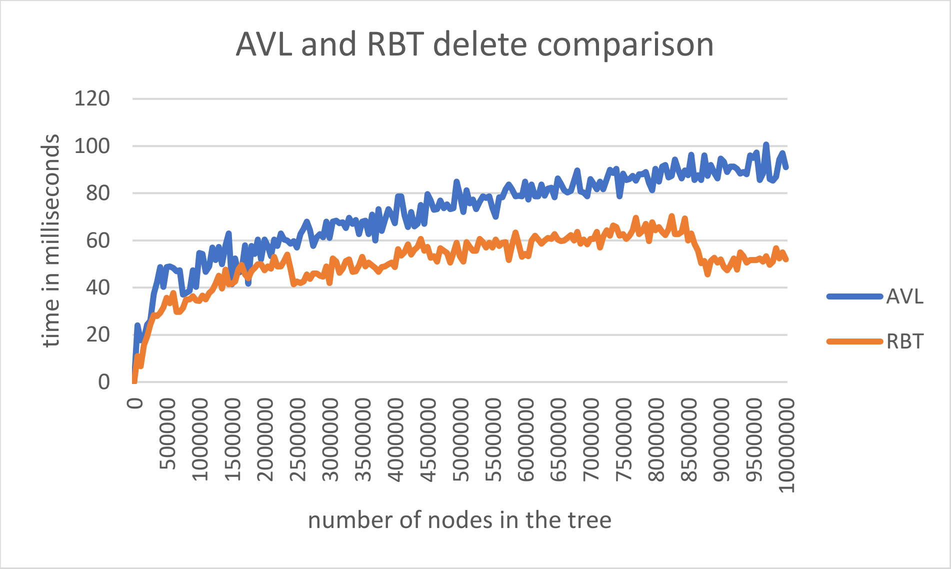
Comparison of AVL and Red-Black Tree

In this part, I compared both of my tree implementations in terms of insert, search, and delete. I used the previous tests from testing both of my implementations separately and put them in a graph next to each other for comparison.

Results







From the graphs, we can see that Red Black tree is faster when inserting and deleting nodes, which may be the result of it being less stricly balanced than AVL tree. On the other hand, the search is a bit faster in AVL thanks to this strict balance.

Implementation of hash table with separate chaining

Description

A hash table is a data structure which uses a hash function to get an index for each key in order for it to get stored into an array of slots. It maps keys to values. However, no hash function is 100 percent perfect, and especially with millions of keys, it is normal that some get the same index – this we call a collision. To resolve collisions, there are mulitple approaches, and in this implementation, I decided to implement separate chaining.

Separate chaining combines using a hash table with a linked list. All the keys that got the same index are stored in one slot, but in a linked list.

The class for constructing a hash table with separate chaining node:

```
static class ChainingHashTableNode<Key, Value> {
    Key key;
    Value value;
    ChainingHashTableNode<Key, Value> next;


     public ChainingHashTableNode(Key key, Value value) {
        this.key = key;
        this.value = value;
        this.next = null;
    }
}
```

It has three attributes – key and value, which can be later in main when creating a hash table set to any data type, and a pointer to next node, as we are implementing chaining collision resolution which uses singly linked list.

The hash table class itself has multiple attributes – default capacity and load factors set to widely accepted values so that the table is efficient; size, capacity and also threshold – capacity multipled by load factor.

Hashing

For hashing, I use a hash method which calls the hashCode method.

```
private int hash(String key, int capacity) {
    int hashCode = hashFunction(key);
    int index = hashCode % capacity;
    return index < 0 ? index + capacity : index;
}
```

Here we suppose that the key is a string, as it was one of the requirements for this to be a string data type. To get the index, we do modulo of hash code and the capacity of the table and then return it, however, if we get a negative index, we add the capacity to it in order to prevent getting an error.

The hashing function:

```
private int hashFunction(String key) {
    int hash = 0x811c9dc5;
    for (int i = 0; i < key.length(); i++) {
        hash ^= key.charAt(i);
        hash *= 0x01000193;
    }
    return hash;
}
```

As we are later testing this implementation on lots of keys, it is important to have a good hash function to keep the efficiency of the hash table. I decided to implement a Fowler–Noll–Vo (FNV1) hash function where huge prime numbers are used in their hexadecimal representation. It successively hashes every character of the key.

Put

To put a key-value pair into a hash table, we need to compute an index and then check whether the „chain“ already exists and inserts it after the last node in it, whether we need to update the value, or just simply add a new node to a nonexistent chain. Eventually we also check whether we need to uspize the hash table.

```
public void put(Key key, Value value) {
    int index = getIndex(key, capacity);
    ChainingHashTableNode<Key, Value> node = table[index];
    while (node != null && !node.key.equals(key)) {
        node = node.next;
    }
    if (node != null) {
        node.value = value;
    } else {
        ChainingHashTableNode<Key, Value> newNode = new
ChainingHashTableNode<>(key, value);
        newNode.next = table[index];
        table[index] = newNode;
        size++;
        if (size >= threshold) {
            Upsize();
        }
    }
}
```

As I mentioned, firstly we hash the key and get index, and then create a node with this index. If it already exists, we do chaining and insert our new node as a next key in a chain. If we find the same key but different value, we update the value. Else we insert this node as a head of chain if the chain did not exist, increase the size, and check whether upsizing is needed.

Remove

To remove a key from this hash table, we again need to compute an index to see where it is stored and then check whether it is somewhere in chain or the head pointer of the linked list. After that, we also check whether it is necessary to downsize the hash table.

```
public void remove(Key key) {
    int index = getIndex(key,capacity);
    ChainingHashTableNode<Key,Value> node = table[index];
    ChainingHashTableNode<Key, Value> prevNode = null;
    while (node != null) {
        if (node.key.equals(key)) {
            if (prevNode == null) table[index] = node.next;
            else {
                prevNode.next = node.next;
            }
            size--;
            if (size < min_lf * capacity) {
                Downsize();
            }
        }
        prevNode = node;
        node = node.next;
    }
}
```

If the key to be deleted is in an existing chain, we iterate through it in a while loop and delete it as we would in a linked list. We also check whether downsizing is required after every iteration. If the key to be deleted is a head of the chain, the head pointer is set to the next node, so if the next node does not exist, the head pointer will also be null and the key won't be in the table anymore.

Get

The get method in a hash table is pretty straightforward.

```
public Value get(Key key) {
    int index = getIndex(key,capacity);
    ChainingHashTableNode<Key, Value> node = table[index];
    while (node != null) {
        if (node.key.equals(key)) {
            System.out.println(node.value);
            return node.value;
        }
        node = node.next;
    }
    return null;
```

Basically, it it retrieves index of this key and looks for it in the table, iterating through it in a while loop. It returns the key if is has been found, otherwise it returns null.

Helper functions

1. getIndex

   This function just calls the hash function and returns the computed index.

2. Upsize

   This method is called when the hash table exceeds the threshold (capacity * load factor). When upsizing is needed, we rehash all the existing elements and increase the capacity. Firstly I double the old capacity, create a new table with this doubled capacity, rehash all the elements in a loop into it and lastly update the previous table and capacity with the new ones.

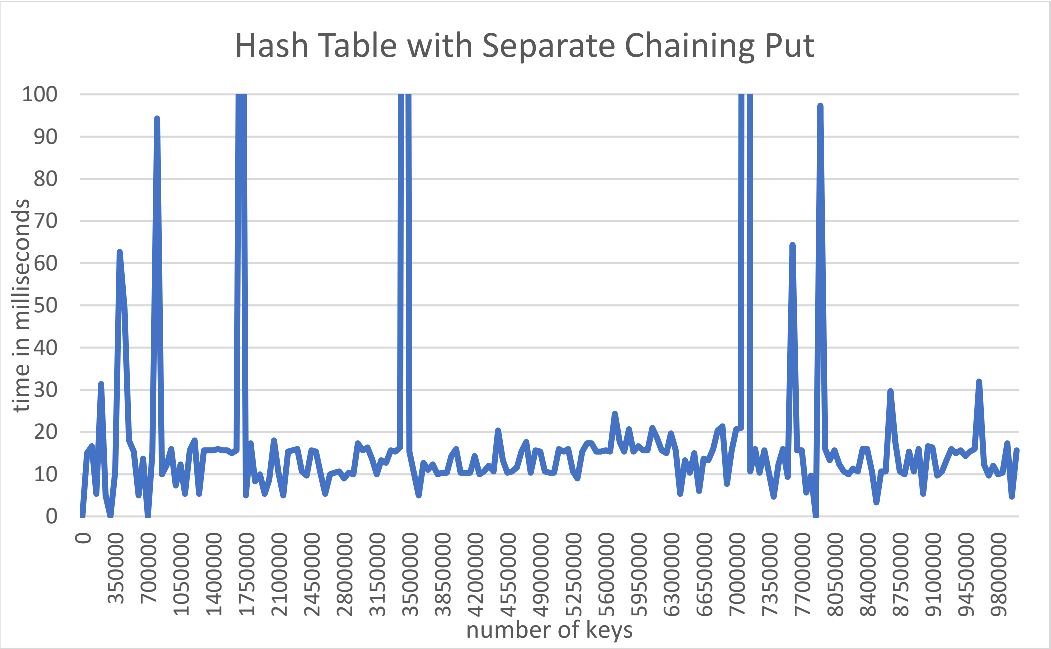
3. Downsize

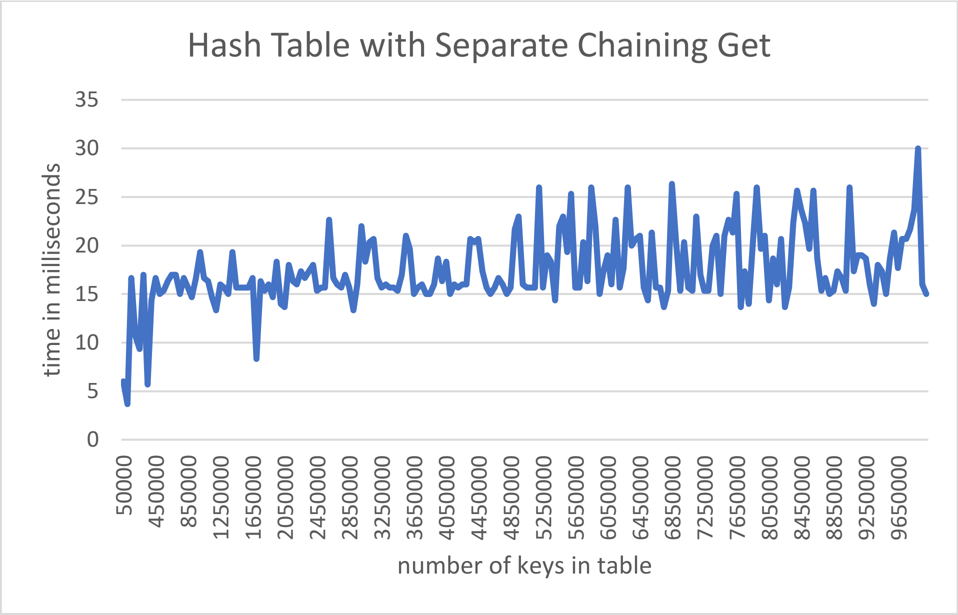
   This method does the same exact thing as the Upsize, but the capacity is not doubled, but halved, and then all the elements are rehashed into a new smaller table and previous variables are updated.

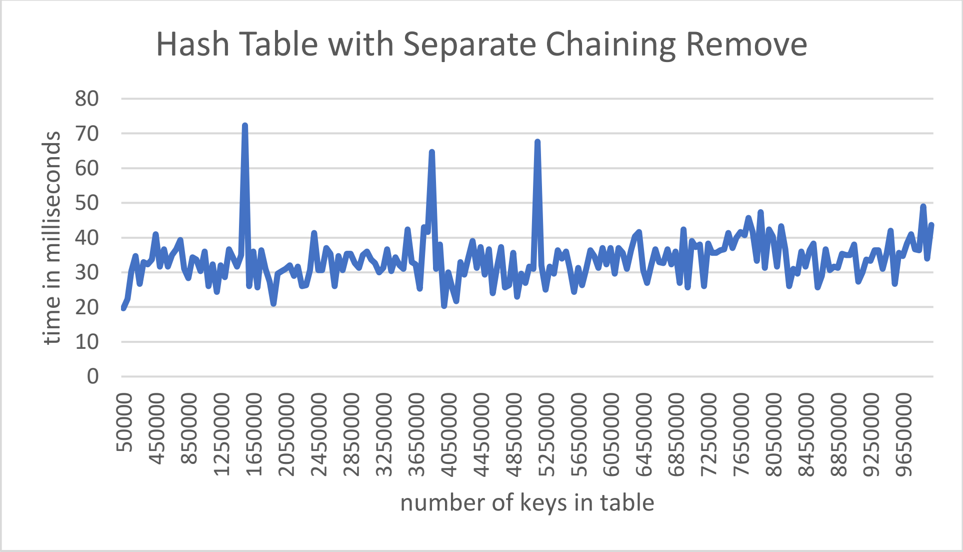
Testing

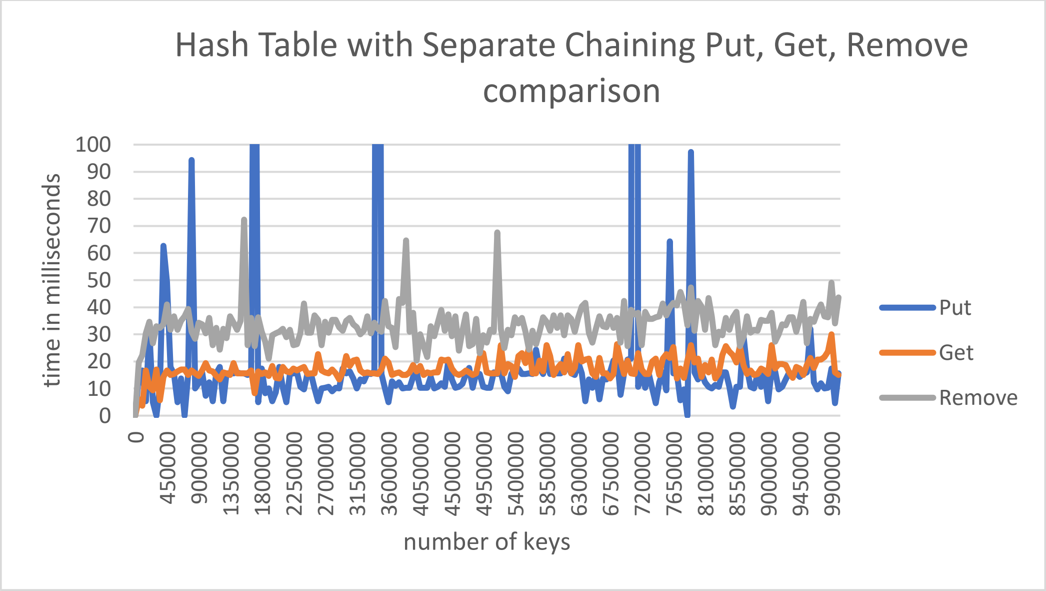
To test the hash table with separate chaining collision resolution, I generated two datasets – one with strings and one with integers. To test the put method, I inserted 10M keys sequentially in intervals by 50K keys. For the testing of get and remove, I then removed or searched for 50K keys in every interval.

Results




Hash Table with Separate Chaining Get


Hash Table with Separate Chaining Remove


Hash Table with Separate Chaining Put, Get, Remove comparison

We can see the graphs for put and remove methods tests are spikey, which can be result of the table being resized at that time. Other than that, all the graphs fluctuate around linear curve. In the comparison of all functions, we see that besides the spikes where resizing happens, the put implementation is the fastest. Bit slower is the get (search), and the slowest is the remove method.

Implementation of hash table with open adressing (quadratic probing)

Description

It already has been mentioned what a hash table is in the previous part of this documentation, but the difference in this implementation is a different approach to collision resolution. Here, I implemented a hash table with quadratic probing (open addressing scheme), where if the index is already occupied by a key, it stores the new key to a new index, which is the computed index added to i (probe attempts made so far) squared and taking this whole result modulo capacity.

The class for constructing a hash table with quadratic probing node:

```
static class QuadraticProbingHashTableNode<Key, Value> {
    Key key;
    Value value;
    QuadraticProbingHashTableNode<Key, Value> next;


    public QuadraticProbingHashTableNode(Key key, Value value) {
        this.key = key;
        this.value = value;
        this.next = null;
    }
}
```

It has the same attributes as in the previous hash table implementation, and the hash table itself also. The differences are in the put, remove and get methods.

Hashing

For hashing, I used the exact same approach as in the previous hash table implementation.

Put

To put an element into a hash table using quadratic probing, we firstly need to get an index for this element and then check whether the slot with this index is empty or not and proceed to do essential steps for inserting this element.

```
public void put(Key key, Value value) {
    int index = getIndex(key, capacity);
    int i = 0;

    while (table[index] != null && !table[index].key.equals(key)) {
        i++;
        index = (index + i * i) % capacity;
    }

    if (table[index] != null && table[index].key.equals(key)) {
        table[index].value = value;
    } else {
        table[index] = new QuadraticProbingHashTableNode<>(key, value);
        size++;

        if (size >= threshold) {
            Upsize();
        }
    }
}
```

If a slot is not empty and the key differs from the one in the slot, then we proceed to do quadratic probing, where we calculate the new index as the old index and the increment squared added. Then we take this result modulo the table capacity in order to not exceed the number of slots. If slot is empty but the keys are equal, we just update the value. Eventually the element is inserted at the computed index (if the slot is empty it simply puts it there), and check whether upsizing is required based on the size of table.

Remove

The index of an element to be deleted is computed in the same way as in put, the element is then deleted and then all the others are re inserted to ensure that quadratic probing sequence is still maintained.

```
public void remove(Key key) {
    int index = getIndex(key, capacity);
    int i = 0;

    while (table[index] != null && !table[index].key.equals(key)) {
        i++;
        index = (index + i * i) % capacity;
    }

    if (table[index] != null && table[index].key.equals(key)) {
        table[index] = null;
        size--;

        int nextIndex = (index + 1) % capacity;
        while (table[nextIndex] != null) {
            QuadraticProbingHashTableNode<Key, Value> node =
table[nextIndex];
            table[nextIndex] = null;
            size--;
            put(node.key, node.value);
            nextIndex = (nextIndex + 1) % capacity;
            if (size < min_lf * capacity) {
                Downsize();
            }
        }
    }
}
```

After computing the index, we iterate in a while loop to find an empty slot or a slot with the same key and set it to null. If the removed node was part of a cluster of elements that were put using quadratic probing, we need to reinsert them in a way that we get the index of a position right after the deleted node and then in a while loop we successively delete each element and put it to different index in a hash table. After each iteration we also check the size of the table whether downsizing is needed.

Get

This method computes the key index (using quadratic probing) and then returns the node if it was found or null if such key is not in the hash table.

```
public Value get(Key key) {
    int index = getIndex(key, capacity);
    int i = 0;

    while (table[index] != null && !table[index].key.equals(key)) {
        i++;
        index = (index + i * i) % capacity;
    }

    if (table[index] != null && table[index].key.equals(key)) {
        return table[index].value;
    } else {
        return null;
    }
}
```

Helper functions

1. getIndex

Same as in previous implementation of hash table

2. Upsize

To upsize a table, the previous capacity is doubled and then new table with this capacity is created. Consequently, all the elements are rehashed from the new table into a new one and we update the variables with new values.

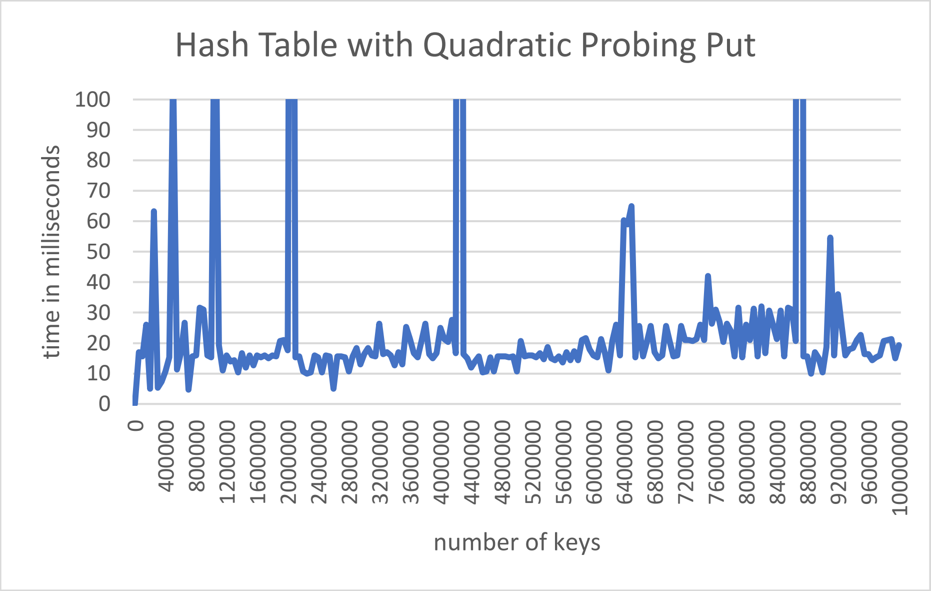
3. Downsize

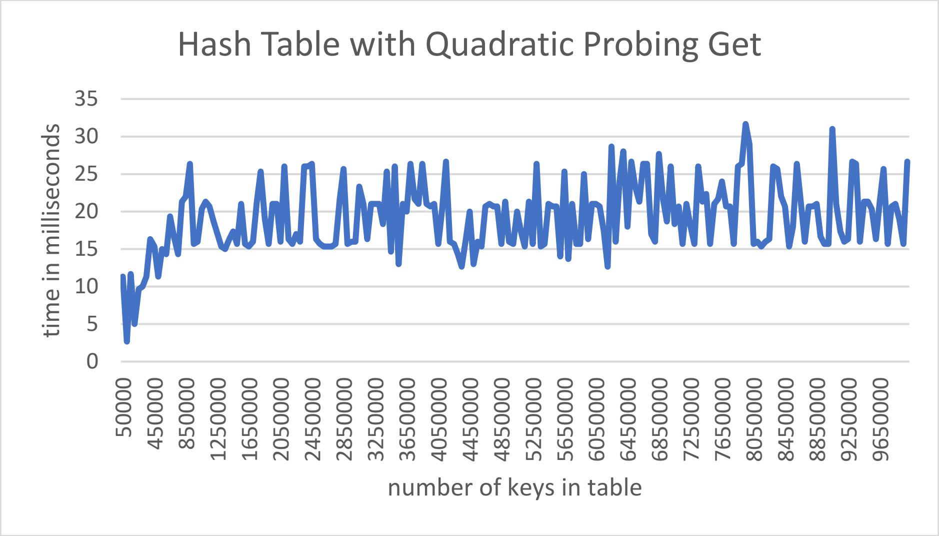
Similar as the Upsize method, but the capacity is halved. The process after making the capacity less than it was is the same as in upsizing.

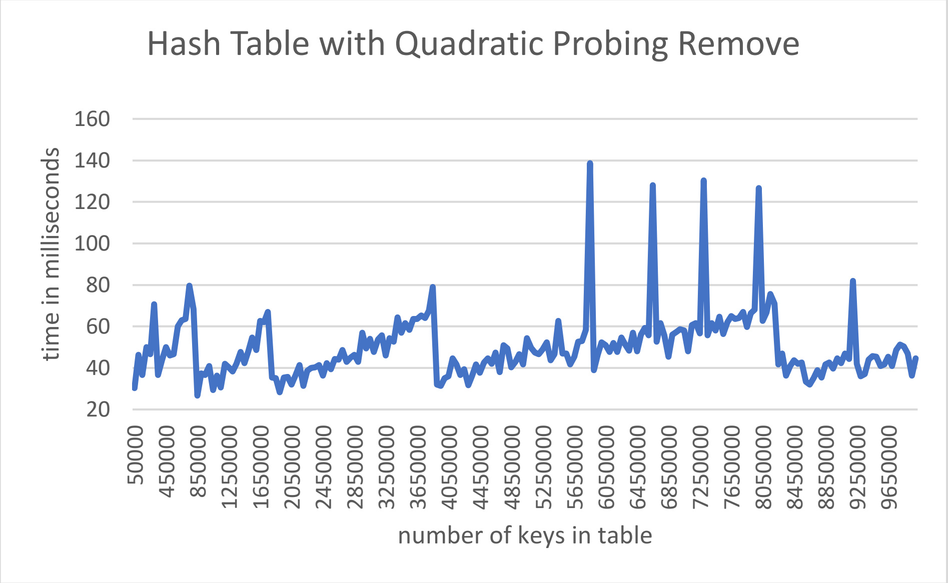
Testing

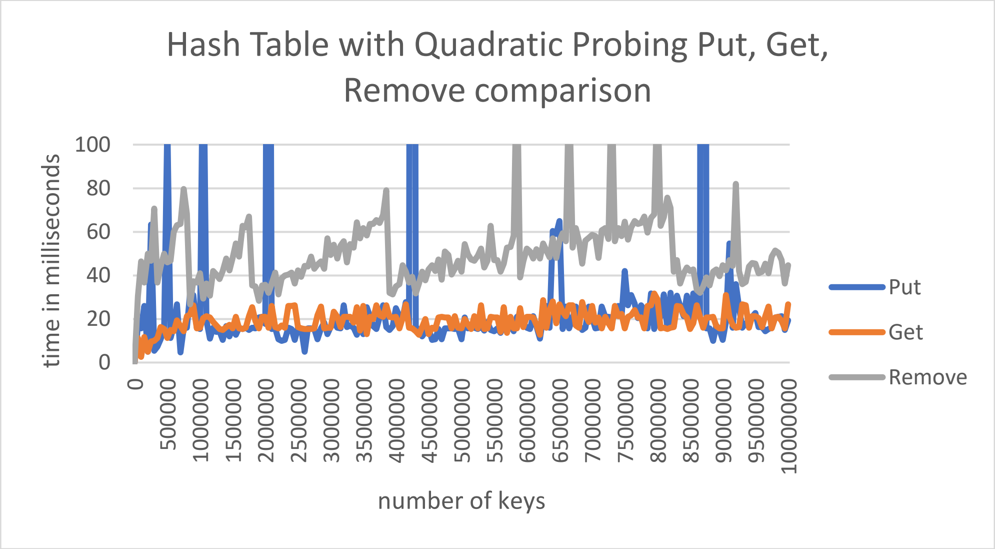
To test the last hash table implementation, this time with quadratic probing collision resolution, I used the exact same test scenario as with the separate chaining.

Results





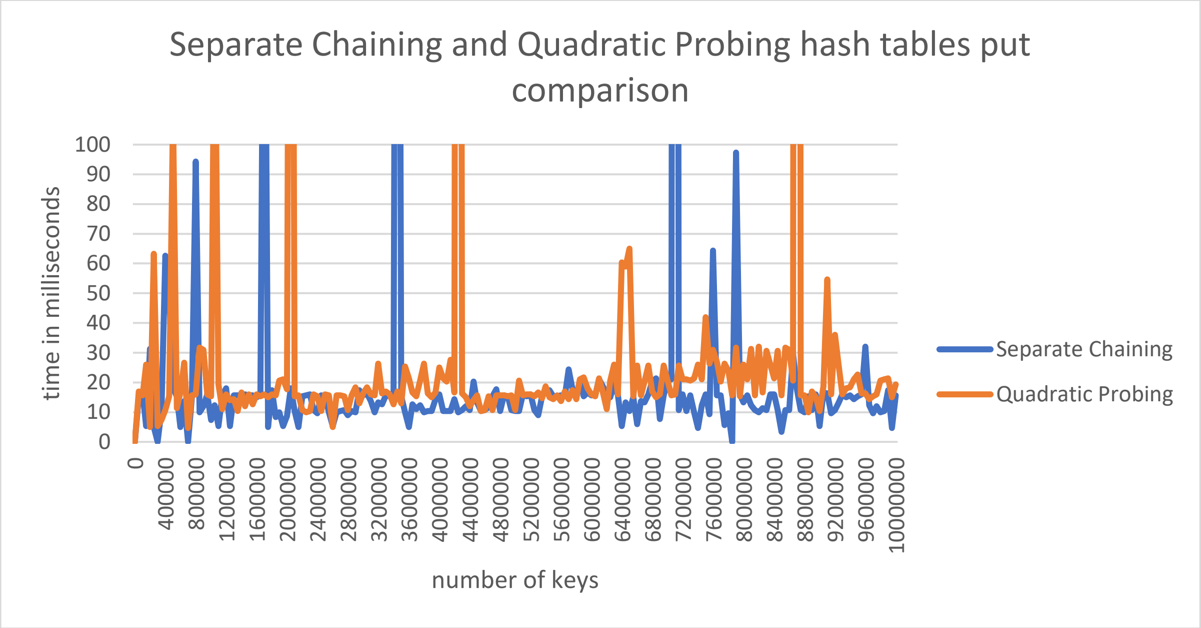


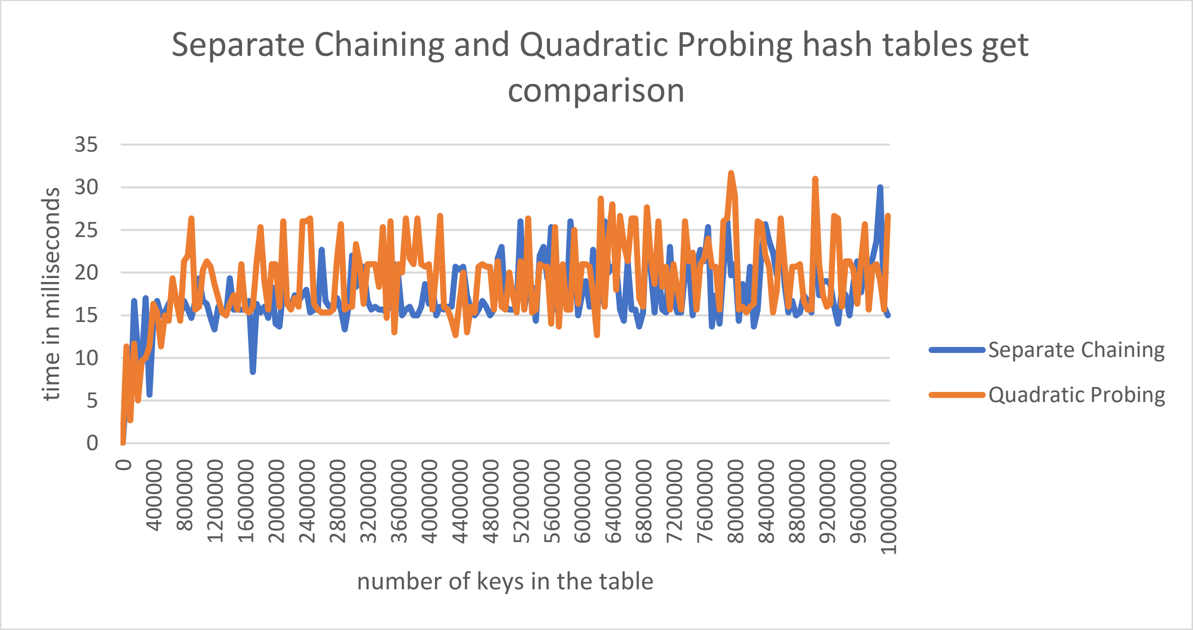


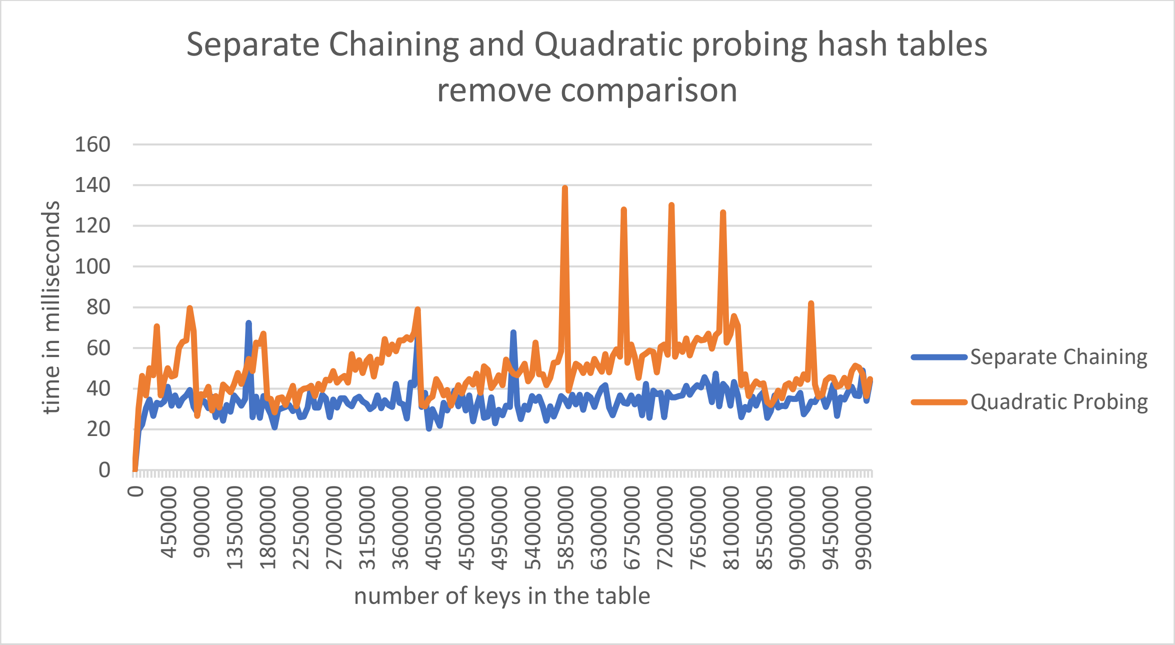
We can see that the put and remove method are again spikey thanks to the resizing which takes up way more time than basic insertion. From the comparison, the put and get method are close to each other regarding time complexity, and remove takes the biggest amount of time.

Comparison of separate chaining hash table and quadratic probing hash table

In this part, I will compare both of my hash table implementations in terms of put, get, and remove. I used the previous tests from testing both of my implementations separately and put them in a graph next to each other for comparison.







From the graphs it is visible that the separate chaining gave better results than the quadratic probing, besides the get method, where these two are almost the same.

Conclusion

My task was to implement and compare four data structures – two binary search trees with balancing algorithm, and two hash tables with collision handling and resizing.

My implementation of binary search trees ensures that they are balanced using the AVL and Red Black algorithms. These algorithms ensure that the height of the tree is logarithmic in terms of the number of elements, which improves the efficiency of search, insert, and delete operations. I observed that AVL trees have a smaller height and faster search times than Red Black trees, but they need more time when inserting and deleting. On the other hand, Red Black trees are not as strictly balanced, so insertion and deletion takes less time - in my testing, inserting into a Red Black tree was twice as fast as into AVL tree.

For hash tables, I used separate chaining and quadratic probing to handle collisions. I observed that looking up a key in both hash tables has almost equal time complexity, but in other operations, quadratic probing table suffers from clustering and requires more memory. On the other hand, separate chaining is more flexible, however, it may have a slightly longer search time due to the need to iterate over the linked lists, but if a good hash function is used, this should not be a problem.

Resources

Besides my knowledge from this year’s lectures in the subject DSA, which are all available in the document server on AIS, I also used some external materials and information from the internet:

1. Paraschiv, E. (2022) *Guide to AVL trees in Java*, *Baeldung*. Available at: https://www.baeldung.com/java-avl-trees (Accessed: March 10, 2023).

2. Woltmann, S. (2022) *Red-black tree (fully explained, with java code)*, *HappyCoders.eu*. HappyCoders.eu. Available at:

https://www.happycoders.eu/algorithms/red-black-tree-java/ (Accessed: March 15, 2023).

3. *Hash table (data structures) - javatpoint* (date unavailable) *www.javatpoint.com*. Available at: https://www.javatpoint.com/hash-table (Accessed: March 18, 2023).

4. *Separate chaining collision handling technique in hashing* (2022) *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/separate-chaining-collision-handling-technique-in-hashing/ (Accessed: March 21, 2023).

5. *Open addressing collision handling technique in hashing* (2023) *GeeksforGeeks*. GeeksforGeeks. Available at: https://www.geeksforgeeks.org/open-addressing-collision-handling-technique-in-hashing/ (Accessed: March 20, 2023).

6. Paraschiv, E. (2023) *Hash table vs. balanced binary tree, Baeldung on Computer Science*. Available at: https://www.baeldung.com/cs/hash-table-vs-balanced-binary-tree (Accessed: March 25, 2023).

7. *Red Black Trees in JAVA | Explained, Examples and Coded* (2021) *YouTube*. Available at: https://www.youtube.com/watch?v=bWNyAU4BpSw&t=2874s&ab_channel=TylerProgramming (Accessed: March 11, 2023).