Faculty of Informatics and Information Technologies STU, Bratislava Ilkovičova 2, 842 16 Bratislava 4

Assignment 1 – Search in dynamic sets

Data Structures and algorithms

Frederik Duvač
2nd semester
Date: 27.3. 2023

Table of contents

1. Theoretical base

1.1. Technical background

Hardware on which the source was compiled and even tested is Lenovo Legion S7, 12th Gen Intel(R) Core™ i5-12500H, 16GB RAM.

The whole project is implemented in Java 19 programming language. Two algorithms for binary trees which were implemented in this work are AVL trees and Splay trees. Hash tables were implemented with chaining and linear probing as a collision resolution method. In this implementation binary trees store only numeric values and hash tables store key-value pairs, which are fully represented by string.

1.2. Introduction

Binary trees and hash tables are two important data structures used in computer science for data storage and fast retrieval of the data. Binary trees are data structures that are used to store and retrieve values in sorted order. Hash tables are data structures that are used to efficiently insert, search and delete values using a hash function.

The advantages of binary trees lie in their ability to maintain ordered data and search it using efficient algorithms. Binary trees have different types and algorithms, such as binary balanced trees AVL tree or Red Black tree and binary nearest neighbour search trees, for example Splay trees. Each type of binary tree has its advantages and disadvantages, so it is important to choose the right type for a particular task.

Hash tables are data structures which use a hash function to index values into an array. This method allows fast insertion, search and deletion of values in constant O(1) time. There are two ways to implement hash tables: "chaining" and "linear probing". Chaining uses lists to store collisions, while linear probing tries to resolve collisions by placing values in the next free position in the table.

In this documentation, we will implement an AVL tree, a Splay tree and a hash table with both Chaining and Linear Probing collision resolution methods. Implementing these data structures will allow us to compare their performance and find out which data structure is best suited for different types of tasks.

1.3. Algorithm explanation

1.3.1. AVL binary tree

AVL trees are a type of self-balancing binary search tree. In a binary search tree, each node has at most two child nodes, and the values in the left subtree are less than the value in the current node, while the values in the right subtree are greater than the value in the current node. This allows efficient search, insertion and deletion of values.

However, in a regular binary search tree, if we insert the values in a certain order, the tree can become unbalanced, which can slow down the search time. This is because unbalanced trees can lead to long paths from the root to the leaves, which increases the number of comparisons needed to find a value.

The main idea behind an AVL tree is to maintain a balance factor for each node in the tree, which represents the difference in height between the right and left subtrees of that node. The balance factor can be one of {-1, 0, 1}, and the tree is considered balanced if the balance factor of each node is within this range. When a new value is inserted into the AVL

tree, the balance condition may be violated, so the tree is rebalanced by performing one or more rotations. Rotations are operations that change the structure of the tree while maintaining the binary search tree property.

There are four types of rotations that can be performed on an AVL tree:

- Left rotation: A root node of subtree is rotated to the left to balance the tree.
- Right rotation: A root node of subtree is rotated to the right to balance the tree.
- Left-Right rotation: A root node of subtree with a right-heavy child is first rotated left, then parent to the right.
- Right-Left Rotation: A root node of subtree with a left-heavy child is first rotated right, then parent to the left.

The goal of rebalancing is to minimise the number of rotations required to balance the tree, since each rotation is a costly operation. By maintaining a balance factor and performing appropriate rotations, AVL trees ensure that the height of the tree remains logarithmic, leading to efficient searches, insertions, and deletions of values.

Overall, AVL trees are a popular choice for applications where fast searching and updating of sorted data is required. However, they require more memory than other data structures and can be slower than other data structures for certain operations, such as inserting or deleting many values at once.

1.3.2. Splay binary tree

A Splay tree is a self-balancing binary search tree, which is used for efficient access of elements in the tree. The key feature of a Splay tree is that it is designed to store the most frequently used elements closer to the root of the tree and by these operations reduce the search time for these elements.

The way this is done is by performing a series of splay operations whenever an element is searched or inserted into the tree. A splay operation is essentially a type of rotation that restructures the tree in such a way that the accessed element moves to the root of the tree. There are 6 types of splay operations that can be performed, known as zig, zag, zig-zig, zag-zag, zig-zag, and zag-zig, which are nothing else than the known left (zag) and right (zig) rotations from AVL tree.

When an element is searched in a Splay tree, it is first located in the same way as in a regular binary search tree. However, once the element is found, a series of splay operations are performed to bring the element to the root of the tree. This has the effect of bringing the most frequently accessed elements closer to the root.

Splay trees have a number of advantages over other types of self-balancing binary search trees. One of the main advantages is that they are simple to implement, since the splay operations are relatively straightforward. Additionally, they can be used in a wide variety of applications, such as database indexing, web caching, and file systems, due to their efficient access times.

However, there are also some disadvantages to using Splay trees. One of the main disadvantages is that the cost of the splay operations can be relatively high in terms of both time and memory, particularly when a large number of elements are being accessed or inserted into the tree. Additionally, because of the way that splay operations work, there is no

guarantee that the tree will be perfectly balanced at all times, which can lead to increased access times for certain elements.

1.3.3. Hash table chaining

Chaining is one way to resolve collisions in a hash table, which occur when a hash function returns the same index for two or more keys. It works by creating an array of buckets, where each bucket in the array could possibly contain a link list or other data structure where the keys and values with the same index are stored. When a new value is added to the hash table, algorithms call for a hash function which returns the index from the key and then possibly determine where it should be inserted to the array of buckets.

Also the load factor is an important parameter in the design and implementation of hash tables, because if the table is too much filled, there is more risk of collision occurrence, which can have an impact on the performance of the hashing function.

One of the solutions for the prevention of a high load factor is the dynamic increase of the capacity of the hash table, when the capacity of the table increases when a certain load factor is reached. Otherwise,for too low load factor, we can consider reducing the capacity of the hash table in order to save memory.

When we are accessing a value in a hash table using chaining, the key is firstly hashed to determine the index in the array where the value might be located. Then, the appropriate data structure in the bucket at that index is searched for the value. If the value is found, it is returned true or the exact element. If the return value is false, it does not exist in the hash table.

When we are deleting a value from a hash table using chaining, the steps are similar to searching for an element, but If the value is found, then it is removed. If the value is not found, it does not exist in the hash table.

One of the advantages of chaining is that it can handle a large number of collisions without significantly impacting performance. However, it can become less efficient when the load factor of the hash table becomes high, as each data structure in the bucket will contain more values and searching through them will cost us more time. It can also be less memory efficient than other collision resolution techniques, as it requires additional memory to store for example the linked lists.

1.3.4. Hash table linear probing

Linear probing is another way of solving collisions when implementing hash tables. The hash table uses linear probing to resolve collisions, which means that if the hash code of an element's key maps to a table index that is already occupied, the algorithm will check the next table index until it finds an empty or deleted slot. The deleted slots are used to mark the places where elements have been deleted, so that they can be skipped during searches and not confuse the linear probing algorithm. If we reach the end of the table, we need to continue from the beginning.

The load factor has an impact on the performance of the hash table during linear probing. If the number of elements in the table increases above a certain limit, the probability of collisions increases and therefore, the number of necessary steps for inserting, searching and deleting elements. Therefore, when designing hash tables, the maximum value of the load factor is usually determined, at which the table will still be considered effective. When

this value is reached, the capacity of the table is usually increased and elements are rehashed to new indexes.

Increasing table capacity and hashing can be computationally and time-intensive, so it is important to choose the right initial capacity and load factor to determine when capacity should be increased.

2. Implementation

2.1. AVL binary tree

The AVL binary tree is represented as an **AVL** class that has an inner class called **Node**, which represents a node in the AVL tree. Each **Node** has a reference to its data, as well as references to its left and right children and height attribute representing the height of the node in the tree to handle the balance factor of the nodes. The AVL class has a root attribute which refers to the root node of the tree.

```
public class AVL {
    private static class Node {
        int data;
        Node left;
        Node right;
        int height;

        public Node(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
            this.height = 1;
        }
    }

    private Node root;

    public AVL() {
        this.root = null;
    }
```

The **insert** method takes a list of data as input and loops through all and inserts each into the AVL tree using **insertNode** method on the root node. The **insertNode** method is a recursive method that inserts the node into the subtree rooted at actualNode. The method first compares the value of the node to be inserted with the actual node to determine which subtree to recurse into. After insertion, the method updates the height of actualNode, calculates the balance factor of actualNode, and performs rotation operations if necessary to rebalance the AVL tree.

```
    public void insert(List<Integer> data) {
        for (int d : data) {
            root = insertNode(root, new Node(d));
        }
    }

    private Node insertNode(Node actualNode, Node insertNode) {
        if (actualNode == null)
            return insertNode;

        if (insertNode.data < actualNode.data) {
            actualNode.left = insertNode(actualNode.left,
insertNode);
        } else if (insertNode.data > actualNode.data){
            actualNode.right = insertNode(actualNode.right,
insertNode);
        } else {
            return actualNode;
        }

        actualNode.height = 1 +
Math.max(getAppropriateHeight(actualNode.left),
getAppropriateHeight(actualNode.right));

        int balanceFactor = getBalanceFactor(actualNode);

        if (balanceFactor > 1) {
            if (insertNode.data < actualNode.left.data) {
                return rightRotate(actualNode);
            } else if (insertNode.data > actualNode.left.data) {
                actualNode.left = leftRotate(actualNode.left);
                return rightRotate(actualNode);
            }
        }
        if (balanceFactor < -1) {
            if (insertNode.data > actualNode.right.data) {
                return leftRotate(actualNode);
            } else if (insertNode.data < actualNode.right.data) {
                actualNode.right = rightRotate(actualNode.right);
                return leftRotate(actualNode);
            }
        }

        return actualNode;
    }
```

The **rightRotate** and **leftRotate** methods perform right and left rotations on the subtree rooted at actualNode, respectively. These rotation operations are used to balance the AVL tree after insertions and deletions.

```
    private Node rightRotate(Node actualNode) {
        Node help1 = actualNode.left;
        Node help2 = help1.right;
        help1.right = actualNode;
        actualNode.left = help2;
                                        actualNode.height          =
Math.max(getAppropriateHeight(actualNode.left),
getAppropriateHeight(actualNode.right)) + 1;
         help1.height = Math.max(getAppropriateHeight(help1.left),
getAppropriateHeight(help1.right)) + 1;
        return help1;
    }

    private Node leftRotate(Node actualNode) {
        Node help1 = actualNode.right;
        Node help2 = help1.left;
        help1.left = actualNode;
        actualNode.right = help2;
                                        actualNode.height          =
Math.max(getAppropriateHeight(actualNode.left),
getAppropriateHeight(actualNode.right)) + 1;
         help1.height = Math.max(getAppropriateHeight(help1.left),
getAppropriateHeight(help1.right)) + 1;
        return help1;
    }
```

The **search** method takes a list of data to be searched as input and loops through all and searches for the result in the AVL tree by calling **searchNode**. The **searchNode** method is an iterative method, which loops from the root to leaves, that searches for the node containing searched data. The method returns true if the node's data are equal to searched data and otherwise returns false.

```
    public void search(List<Integer> data) {
        for (int d : data) {
            searchNode(d);
        }
    }

    private boolean searchNode(int data) {
        Node actualNode = getRoot();
```

```
        if (actualNode == null) {
            return false;
        }

        while (actualNode.data != data) {
            if (data < actualNode.data) {
                actualNode = actualNode.left;
            } else {
                actualNode = actualNode.right;
            }
            if (actualNode == null) {
                return false;
            }
        }
        return true;
    }
```

The **delete** method takes a list of numbers as input and deletes each Node with that data from the AVL tree by calling **deleteNode** on the root node. The **deleteNode** method is a recursive method that deletes the node from the subtree rooted at actualNode. The method first compares the value of data with actualNode to determine which subtree to recurse into. After deletion, the method updates the height of actualNode, calculates the balance factor of actualNode, and performs rotation operations if necessary to rebalance the AVL tree.

```
    public void delete(List<Integer> data) {
        for (int d : data) {
            root = deleteNode(getRoot(), d);
        }
    }

    private Node deleteNode(Node actualNode, int data) {
        if (actualNode == null)
            return null;

        if (actualNode.data > data)
            actualNode.left = deleteNode(actualNode.left, data);
        else if (actualNode.data < data)
            actualNode.right = deleteNode(actualNode.right, data);
        else {
            if ((actualNode.left == null) ||
    (actualNode.right == null)) {
                Node help;
                if (actualNode.left == null)
                    help = actualNode.right;
                else
```

```
                    help = actualNode.left;

                actualNode = help;
            } else {
                Node help = minimum(actualNode.right);
                actualNode.data = help.data;
                actualNode.right =
     deleteNode(actualNode.right, help.data);
            }
        }

        if (actualNode == null)
            return null;

        actualNode.height =
Math.max(getAppropriateHeight(actualNode.left),
getAppropriateHeight(actualNode.right)) + 1;

        int balanceFactor = getBalanceFactor(actualNode);

        if (balanceFactor < -1 &&
getBalanceFactor(actualNode.right) <= 0)
            return leftRotate(actualNode);

        if (balanceFactor > 1 &&
     getBalanceFactor(actualNode.left) >= 0)
            return rightRotate(actualNode);

        if (balanceFactor > 1 &&
     getBalanceFactor(actualNode.left) < 0) {
            actualNode.left = leftRotate(actualNode.left);
            return rightRotate(actualNode);
        }

        if (balanceFactor < -1 &&
getBalanceFactor(actualNode.right) > 0) {
            actualNode.right = rightRotate(actualNode.right);
            return leftRotate(actualNode);
        }

        return actualNode;
     }
```

The **getBalanceFactor** method returns the balance factor of a node, which is defined as the height of the left subtree minus the height of the right subtree.

```
    private int getBalanceFactor(Node node) {
        if (node == null)
            return 0;

        return getAppropriateHeight(node.left) -
getAppropriateHeight(node.right);
    }
```

The **getAppropriateHeight** method returns the height of a node, if the node is null then returns 0.

```
    private int getAppropriateHeight(Node node) {
        if (node == null)
            return 0;
        return node.height;
    }
```

The **minimum** method returns the minimum node in the subtree rooted at actualNode.

```
    private Node minimum(Node actualNode) {
        while (actualNode.left != null) {
            actualNode = actualNode.left;
        }
        return actualNode;
    }
```

2.2. Splay binary tree

In my case Splay tree is represented by its own class **Splay**. The **Splay** class has an inner class **Node** that represents a single tree element. **Node** contains references to its left and right child, parent, and data. The **Splay** class has a **root** attribute that represents the root of the binary tree.

```
    public class Splay {
        private static class Node {
            int data;
            Node left, right, parent;

            public Node(int data) {
```

```
            this.data = data;
            this.left = this.right = this.parent = null;
        }
    }

    private Node root;

    public Splay() {
        this.root = null;
    }
```

Implementation of Splay tree should consist of those similar following methods:

The **insert** method loops through all wanted elements to insert from the list **data** into the binary tree using the **insertNode** method. The **insertNode** method firstly creates **insertNode** to be inserted from the data then iteratively traverses the tree from the root to the leaves to find the exact position for the new element whether it should be left or right child of the leaf. Finally, the splay method is executed, which rearranges the tree so that the new element is at the root.

```
    public void insert(List<Integer> data) {
        for (int d : data) {
            insertNode(d);
        }
    }

    private void insertNode(int data) {
        Node insertNode = new Node(data);
        Node prev = null;
        Node actualNode = root;

        while (actualNode != null) {
            prev = actualNode;

            if (insertNode.data < actualNode.data) {
                actualNode = actualNode.left;
            } else {
                actualNode = actualNode.right;
            }
        }

        insertNode.parent = prev;

        if (prev == null) {
            setRoot(insertNode);
        } else if (insertNode.data < prev.data) {
            prev.left = insertNode;
```

```
        } else {
            prev.right = insertNode;
        }

        splay(insertNode);
    }
```

The **splay** method performs rotations on the parent of the inserted node until the element that was just inserted is not at the root position, while maintaining balance in the tree.

```
    private void splay(Node node) {
        while (node.parent != null) {
            if (node.parent.parent == null) {
                if (node == node.parent.left) {
                    // right rotation
                    doZigRotation(node.parent);
                } else {
                    // left rotation
                    doZagRotation(node.parent);
                }
            } else if (node == node.parent.left &&
    node.parent == node.parent.parent.left) {
                // right-right rotation
                doZigRotation(node.parent.parent);
                doZigRotation(node.parent);
            } else if (node == node.parent.right &&
    node.parent == node.parent.parent.right) {
                // left-left rotation
                doZagRotation(node.parent.parent);
                doZagRotation(node.parent);
            } else if (node == node.parent.right &&
    node.parent == node.parent.parent.left) {
                // left-right rotation
                doZagRotation(node.parent);
                doZigRotation(node.parent);
            } else {
                // right-left rotation
                doZigRotation(node.parent);
                doZagRotation(node.parent);
            }
        }
    }
```

The **zig rotation** is actually the right rotation and makes appropriate changes to references of the actual node to be rotated. Firstly it creates a helper which is the left child of the actual node, then the left child of the actual node will be the right child of the helper and if

it is not null then makes the actual node to be his parent. If the actual node is the root then we set the help to be the root node. Else if the actual node is the right child of its parent then help will be the right child of the parent . Else help will be the left child of the parent. Lastly, the actual node will be the right child of the help.

```
// Right rotation
private void doZigRotation(Node actualNode) {
    Node help = actualNode.left;
    actualNode.left = help.right;
    if (help.right != null) {
        help.right.parent = actualNode;
    }

    if (actualNode.parent == null) {
        setRoot(help);
    } else if (actualNode == actualNode.parent.right) {
        actualNode.parent.right = help;
        help.parent = actualNode.parent;
    } else {
        actualNode.parent.left = help;
        help.parent = actualNode.parent;
    }

    help.right = actualNode;
    actualNode.parent = help;
}
```

The **zag rotation** works the same way but mirrored. The helper will be the right child etc.

```
// Left rotation
private void doZagRotation(Node actualNode) {
    Node help = actualNode.right;
    actualNode.right = help.left;
    if (help.left != null) {
        help.left.parent = actualNode;
    }

    if (actualNode.parent == null) {
        setRoot(help);
    } else if (actualNode == actualNode.parent.left) {
        actualNode.parent.left = help;
        help.parent = actualNode.parent;
    } else {
        actualNode.parent.right = help;
```

```
            help.parent = actualNode.parent;
        }

        help.left = actualNode;
        actualNode.parent = help;
    }
```

The **search** method searches loops through all elements from the list and uses the **searchNode** method to find the element in the tree. If the element is in a tree, the **splay** method is executed to get the element to the root.

```
    public void search(List<Integer> data) {
        for (int d : data) {
            searchNode(d);
        }
    }

    private boolean searchNode(int data) {
        Node actualNode = root;

        if (actualNode == null) {
            return false;
        }

        while (actualNode.data != data) {
            if (data < actualNode.data) {
                actualNode = actualNode.left;
            } else {
                actualNode = actualNode.right;
            }
            if (actualNode == null) {
                return false;
            }
        }

        splay(actualNode);
        return true;
    }
```

The **delete** method removes elements from the list using the **deleteNode** method. Firstly we need to search whether the element is in the tree or not. Then this element is rearranged to the root using the **splay** method. If this element has left and right children, they are joined using the **join** method, which creates a new tree. Then the new tree is set as the root and the element is removed.

```
    public void delete(List<Integer> data) {
        for (int d : data) {
            deleteNode(d);
        }
    }

    private boolean deleteNode(int data) {
        Node actualNode = root;

        if (actualNode == null) {
            return false;
        }

        while (actualNode.data != data) {
            if (data < actualNode.data) {
                actualNode = actualNode.left;
            } else {
                actualNode = actualNode.right;
            }
            if (actualNode == null) {
                return false;
            }
        }

        Node deleteNode = actualNode;
        Node leftSubtree = null;
        Node rightSubtree = null;

        splay(deleteNode);

        if (deleteNode.right != null) {
            rightSubtree = deleteNode.right;
            rightSubtree.parent = null;
        }
        if (deleteNode.left != null) {
            leftSubtree = deleteNode.left;
            leftSubtree.parent = null;
        }

        setRoot(join(leftSubtree, rightSubtree));

        return true;
    }
```

The **join** method merges the left subtree and right subtree of the actual node on the root position which is deleted. If both subtrees are not null the new root will be found using

the **maximum** method which returns Node with max value from the left subtree. Otherwise it returns only the left or the right subtree.

```
private Node join(Node leftSubtree, Node rightSubtree) {
    if (leftSubtree == null) {
        return rightSubtree;
    }

    if (rightSubtree == null) {
        return leftSubtree;
    }

    Node maxFromLeftSubtree = maximum(leftSubtree);
    splay(maxFromLeftSubtree);
    maxFromLeftSubtree.right = rightSubtree;
    rightSubtree.parent = maxFromLeftSubtree;

    return maxFromLeftSubtree;
}

public Node maximum(Node actualNode) {
    while (actualNode.right != null) {
        actualNode = actualNode.right;
    }
    return actualNode;
}
```

2.3. Hash table chaining

The Hash table using chaining as a collision resolution was implemented as its own class **HashTableChaining** with inner class **Entry** with attributes key and value. The hash table is represented as a list of buckets, where each bucket is a list of key-value pairs (entries). The constructor of **HashTableChaining** takes two parameters: the initial capacity of the hash table and the load factor, which determines when the hash table should be resized.

```
public class HashTableChaining {
    public static class Entry {
        private final String key;
        private final String value;

        public Entry(String key, String value) {
            this.key = key;
            this.value = value;
```

```
        }

        public String getKey() {
            return key;
        }

        public String getValue() {
            return value;
        }
    }

    private List<List<Entry>> bucketList = new ArrayList<>();
    private int capacity;
    private final float loadFactor;
    private int size;

    public HashTableChaining(int capacity, float loadFactor) {
        this.capacity = capacity;
        this.loadFactor = loadFactor;
        size = 0;
        for (int i = 0; i < capacity; i++) {
            bucketList.add(new ArrayList<>());
        }
    }
```

The **insert**, **search**, and **delete** methods take a list of Entry objects as a parameter, and perform the corresponding operation on each entry in the list. The **insertValue**, **searchValue**, and **deleteValue** methods take a single **Entry** object as a parameter and call the corresponding method with parameter entry.

The **insertEntry** method inserts an entry into the hash table by first checking if the load factor has been exceeded and resizing the hash table if necessary using the **resize** method. It then calculates the hash code of the entry's key and retrieves the corresponding bucket. If the key is already in the bucket, the method returns without doing anything. Otherwise, the entry is added to the bucket and the size of the hash table is incremented.

```
    private void insertEntry(Entry e) {
        if ((float) size / capacity >= loadFactor) {
            resize();
        }

        int i = hash(e);
        List<Entry> bucket = bucketList.get(i);

        for (Entry entry : bucket) {
            if (entry.getKey().equals(e.getKey())) {
                return;
```

```
            }
        }

        bucket.add(e);
        size++;
    }
```

The **resize** method firstly doubles the capacity and creates a new table, then recalculates new hashes and remaps all the entries to new indexes from old table to new one and lastly changes the old table to new table.

```
    private void resize() {
        capacity *= 2;
        List<List<Entry>> newBucketList = new ArrayList<>();

        for (int i = 0; i < capacity; i++) {
            newBucketList.add(new ArrayList<>());
        }

        for (List<Entry> bucket : bucketList) {
            for (Entry entry : bucket) {
                int i = hash(entry);
                newBucketList.get(i).add(entry);
            }
        }

        bucketList = newBucketList;
    }
```

The **searchEntry** method searches for an entry in the hash table by calculating the hash code of the entry's key and retrieving the corresponding bucket. It then searches the bucket for an entry with the same key. If it finds one, it returns true. Otherwise, it returns false.

```
    private boolean searchEntry(Entry e) {
        int i = hash(e);
        List<Entry> entryList = bucketList.get(i);
        for (Entry entry : entryList) {
            if (entry.getKey().equals(e.getKey())) {
                return true;
            }
        }
        return false;
    }
```

The **deleteEntry** method deletes an entry from the hash table by calculating the hash code of the entry's key and retrieving the corresponding bucket. It then searches the bucket for an entry with the same key. If it finds one, it removes it from the bucket and decrements the size of the hash table. It then returns true. Otherwise, it returns false.

```
    private boolean deleteEntry(Entry e) {
        int i = hash(e);
        List<Entry> entryList = bucketList.get(i);

        for (int j = 0; j < entryList.size(); j++) {
            Entry entry = entryList.get(j);
            if (entry.getKey().equals(e.getKey())) {
                entryList.remove(entryList.get(j));
                size--;
                return true;
            }
        }
        return false;
    }
```

The **hash** method calculates the hash code of an entry's key using the Java default hashCode method of the key object and the capacity of the hash table.

```
    private int hash(Entry e) {
        return Math.abs(e.getKey().hashCode() % capacity);
    }
```

2.4. Hash table linear probing

My implementation of the hash table using linear probing as a collision resolution technique represents its own class **HashTableLinearProbing.** The hash table is represented as an array of entries, which are the type of **Entry** with attributes key and value same as in hash table chaining implementation. The constructor of **HashTableLinearProbing** also takes two parameters: the initial capacity of the hash table and the load factor.

```
    public class HashTableLinearProbing {
        private static final Entry deletedEntry = new
Entry("DELETED", "DELETED");
```

```
        private Entry[] table;
        private int capacity;
        private final float loadFactor;
        private int size;

        public HashTableLinearProbing(int capacity, float
loadFactor) {
            this.capacity = capacity;
            this.loadFactor = loadFactor;
            size = 0;
            table = new Entry[capacity];
        }
```

The **insert**, **search**, and **delete** methods take a list of Entry objects as a parameter, and perform the corresponding operation on each entry in the list. The **insertValue**, **searchValue**, and **deleteValue** methods take a single **Entry** object as a parameter and call the corresponding method with parameter entry.

The **insertEntry** method inserts a single Entry object into the hash table. This method first checks if the hash table needs to be resized if the current size is greater than or equal to the load factor times the capacity, and if so, it calls the **resize** method to double the capacity of the hash table. Then, it calculates the hash value of the key using the hash method and uses linear probing to find the next available slot in the hash table. If the slot is already occupied by another entry with a different key, it continues probing the next slot until it finds an empty or **deletedEntry** slot. Lastly, it stores the entry in the empty or **deletedEntry** slot, and increments the size.

```
    private void insertEntry(Entry e) {
        if (size >= loadFactor * capacity) {
            resize();
        }

        int i = hash(e);
        while (table[i] != null &&
    table[i] != deletedEntry &&
!table[i].getKey().equals(e.getKey())) {
            i = (i + 1) % capacity;
        }
        table[i] = e;
        size++;
    }
```

The **resize** method doubles the capacity of the hash table then creates a new table with the new capacity and rehashes all the entries from the old table to the new table using linear probing.

```
private void resize() {
    capacity *= 2;
    Entry[] newTable = new Entry[capacity];

    for (Entry entry : table) {
        if (entry != null) {
            int i = hash(entry);
            while (newTable[i] != null) {
                i = (i + 1) % capacity;
            }
            newTable[i] = entry;
        }
    }

    table = newTable;
}
```

The **searchEntry** method searches for a single **Entry** object in the hash table. This method calculates the hash value of the key using the hash method and uses linear probing to search for the entry with the equal key. If it searches for the entry, it returns true. Otherwise, it returns false.

```
private boolean searchEntry(Entry e) {
    int i = hash(e);
    while (table[i] != null) {
        if (table[i].getKey().equals(e.getKey())) {
            return true;
        }
        i = (i + 1) % capacity;
    }

    return false;
}
```

The **deleteEntry** method deletes a single **Entry** object from the hash table. This method calculates the hash value of the key using the **hash** method and uses linear probing to search for the entry with the matching key. If it finds the pair, it marks the slot as deleted by storing a special **deletedEntry** object in the slot, decrements the size variable, and returns true. Otherwise, it returns false.

```
private boolean deleteEntry(Entry e) {
    int i = hash(e);
```

```
        while (table[i] != null) {
            if (table[i].getKey().equals(e.getKey())) {
                table[i] = deletedEntry;
                size--;
                return true;
            }
            i = (i + 1) % capacity;
        }

        return false;
    }
```

The **hash** method calculates the hash code of an entry's key using the Java default **hashCode** method of the key object and the capacity of the hash table.

```
    private int hash(Entry e) {
        return Math.abs(e.getKey().hashCode() % capacity);
    }
```

3. Testing

3.1. Console interface

After run the measure class will print the following command “Binary tree => b | Hash table => h >>”. User is able to choose whether he wants to test binary trees or hash tables. Then after selection b or h there is an option to choose from full test, single test and single full test “Full test => f | Single test => s | Single full test => sf >>”.

Full test measures time complexity for insert, search and delete of all values from dataset to binary tree. It also provides an option to decide how many repeats it should make and then it returns minimal, average and maximal time duration in milliseconds for all cases. It is fully automated so the user needs to input a value for interval which makes from the one huge dataset different sub datasets for the testing cases.

Single test offers the user to decide how many datasets to create and what size they should be. Single test offers users to manually select between insert, search and delete to those datasets and then single test a value imputed by the user.

Lastly the main testing type is single full test. This requires the user to decide how many datasets at what size the program should create. Then the user is able to type the value to insert and the main point is to input the amount of repeats to fully test the value for all input, search and delete at once. The result is minimal, average and maximal duration in nanoseconds for all the different size datasets.

After all testing types, the results are written to csv files to better parse the data to tables and then subsequently visualised into graphs for further analysis.

Testing could look like in the following picture.

```
Binary tree => b | Hash table => h >> h
Full test => f | Single test => s | Single full test => sf >> sf
How many datasets to create from btDataset.txt >> 2
Amount of data in 1.dataset from btDataset.txt max size: 10000 >> 2500
Amount of data in 2.dataset from btDataset.txt max size: 10000 >> 7500
Data to insert [key value]: abcdefghij Riko
How many repeats to do >> 100
-> dataset 1(2500) HashTable Chaining total time: 157300nanos
    HashTable Chaining insert ==> min duration: 0nanos | avg duration: 662nanos | max duration: 36500nanos | total time for 100 repeats: 66200nanos
    HashTable Chaining search ==> min duration: 100nanos | avg duration: 313nanos | max duration: 5800nanos | total time for 100 repeats: 31300nanos
    HashTable Chaining delete ==> min duration: 300nanos | avg duration: 598nanos | max duration: 14800nanos | total time for 100 repeats: 59800nanos

-> dataset 1(2500) HashTable Linear Probing total time: 77800nanos
    HashTable Linear Probing insert ==> min duration: 100nanos | avg duration: 491nanos | max duration: 22700nanos | total time for 100 repeats: 49100nanos
    HashTable Linear Probing search ==> min duration: 100nanos | avg duration: 153nanos | max duration: 4100nanos | total time for 100 repeats: 15300nanos
    HashTable Linear Probing delete ==> min duration: 100nanos | avg duration: 134nanos | max duration: 1200nanos | total time for 100 repeats: 13400nanos

-> dataset 2(7500) HashTable Chaining total time: 56300nanos
    HashTable Chaining insert ==> min duration: 0nanos | avg duration: 100nanos | max duration: 3200nanos | total time for 100 repeats: 10000nanos
    HashTable Chaining search ==> min duration: 100nanos | avg duration: 180nanos | max duration: 1400nanos | total time for 100 repeats: 18000nanos
    HashTable Chaining delete ==> min duration: 200nanos | avg duration: 283nanos | max duration: 2100nanos | total time for 100 repeats: 28300nanos

-> dataset 2(7500) HashTable Linear Probing total time: 68000nanos
    HashTable Linear Probing insert ==> min duration: 0nanos | avg duration: 228nanos | max duration: 12400nanos | total time for 100 repeats: 22800nanos
    HashTable Linear Probing search ==> min duration: 200nanos | avg duration: 225nanos | max duration: 900nanos | total time for 100 repeats: 22500nanos
    HashTable Linear Probing delete ==> min duration: 200nanos | avg duration: 227nanos | max duration: 600nanos | total time for 100 repeats: 22700nanos

Successfully wrote to the file htcSingleMeasure.csv
Successfully wrote to the file htpSingleMeasure.csv
```

3.2. Testing implementation

Testing of data structures was implemented in the Measure class, which has a lot of attributes related to insert, search, delete, minimal, average, maximal and total time durations, which are used by the methods implemented in this class.

There are separate methods **insert…**, **search…** and **delete…** for all data structures, which measure insert, search and delete only for one data. Those methods are used in a single test.

```java
    public void insertAvl(int input, AVL avl) {
        long startAvl = System.nanoTime();
        avl.insertValue(input);
        long endAvl = System.nanoTime();
        insertDuration = endAvl - startAvl;
    }

    public void searchAvl(int input, AVL avl) {
        long startAvl = System.nanoTime();
        avl.searchValue(input);
        long endAvl = System.nanoTime();
        searchDuration = endAvl - startAvl;
    }

    public void deleteAvl(int input, AVL avl) {
        long startAvl = System.nanoTime();
        avl.deleteValue(input);
        long endAvl = System.nanoTime();
        deleteDuration = endAvl - startAvl;
    }
```

The **create…** methods are used at the beginning of single and single full type testing when it is necessary to create all data structures by inserting all data at once and then the appropriate single test methods could be performed.

```
    public void createAvl(List<Integer> data, AVL avl) {
        avl.insert(data);
    }

    public void createSplay(List<Integer> data, Splay splay) {
        splay.insert(data);
    }

    public void createHTChaining(List<Entry> data,
HashTableChaining tableChaining) {
        tableChaining.insert(data);
    }

    public void createHTLProbing(List<Entry> data,
HashTableLinearProbing tableLinearProbing) {
        tableLinearProbing.insert(data);
    }
```

The **startSingleMeasure…** methods are helper functions in a single full test. Firstly insert, search and delete of one value is performed once separately, it's because initialisation of minimal and maximal duration. Then the same operations are done in a loop until it reaches **measureCount** but now the durations are compared and summaries of durations are calculated to find the average time duration of each operation.

```
    public void startSingleMeasureAvl(int data,
    AVL avl, int dataSize, int datasetNumber) {
        List<Long> insertTime = new ArrayList<>();
        List<Long> searchTime = new ArrayList<>();
        List<Long> deleteTime = new ArrayList<>();
        long sumInsert = 0;
        long sumSearch = 0;
        long sumDelete = 0;

        insertAvl(data, avl);
        searchAvl(data, avl);
        deleteAvl(data, avl);

        insertTime.add(insertDuration);
        sumInsert += insertDuration;
        minInsertTime = insertDuration;
```

```
        maxInsertTime = insertDuration;

        searchTime.add(searchDuration);
        sumSearch += searchDuration;
        minSearchTime = searchDuration;
        maxSearchTime = searchDuration;

        deleteTime.add(deleteDuration);
        sumDelete += deleteDuration;
        minDeleteTime = deleteDuration;
        maxDeleteTime = deleteDuration;

        for (int i = 1; i < measureCount; i++) {
            insertAvl(data, avl);
            searchAvl(data, avl);
            deleteAvl(data, avl);

            insertTime.add(insertDuration);
            searchTime.add(searchDuration);
            deleteTime.add(deleteDuration);
            sumInsert += insertDuration;
            sumSearch += searchDuration;
            sumDelete += deleteDuration;

            if (insertDuration < minInsertTime) {
                minInsertTime = insertDuration;
            } else if (insertDuration > maxInsertTime) {
                maxInsertTime = insertDuration;
            }
            if (searchDuration < minSearchTime) {
                minSearchTime = searchDuration;
            } else if (searchDuration > maxSearchTime) {
                maxSearchTime = searchDuration;
            }
            if (deleteDuration < minDeleteTime) {
                minDeleteTime = deleteDuration;
            } else if (deleteDuration > maxDeleteTime) {
                maxDeleteTime = deleteDuration;
            }
        }

        totalInsertTime = sumInsert;
        totalSearchTime = sumSearch;
        totalDeleteTime = sumDelete;
        avgInsertTime = sumInsert / insertTime.size();
        avgSearchTime = sumSearch / searchTime.size();
        avgDeleteTime = sumDelete / deleteTime.size();
```

```
            printAvlMeasure(dataSize, datasetNumber, true);
        }
```

The **startFullMeasure…** methods are helper functions in full test. Firstly insert, search and delete of all values is performed because initialisation of minimal and maximal duration. Then the same operations are done in a loop until it reaches **measureCount** but now the durations are compared and summaries of durations are calculated to find the average time duration of each operation.

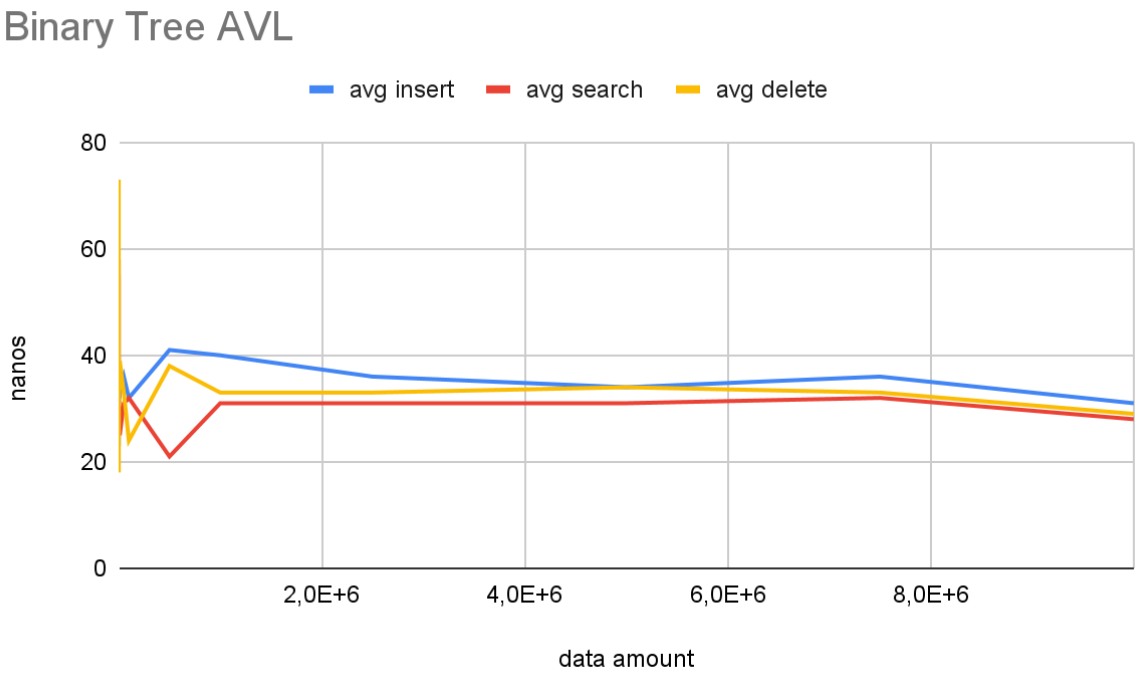
The **print…** methods are helper functions for print results to the console.

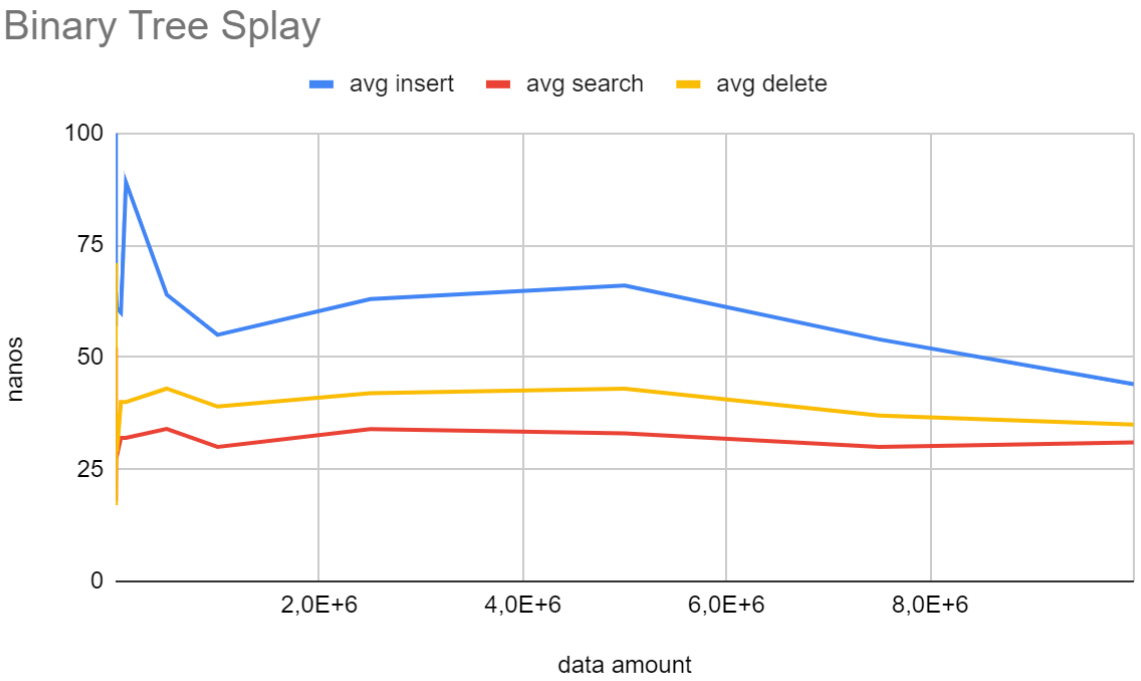
The **Measure** class also contains the **main** method. Firstly there is an initialisation of datasets for binary trees and for hash tables from external files which were generated by another DataGeneration class. Then the program asks for testing type and other options mentioned in the console interface section and after that creates required datasets and starts performing necessary testing of implemented data structures.

3.3. Time comparison

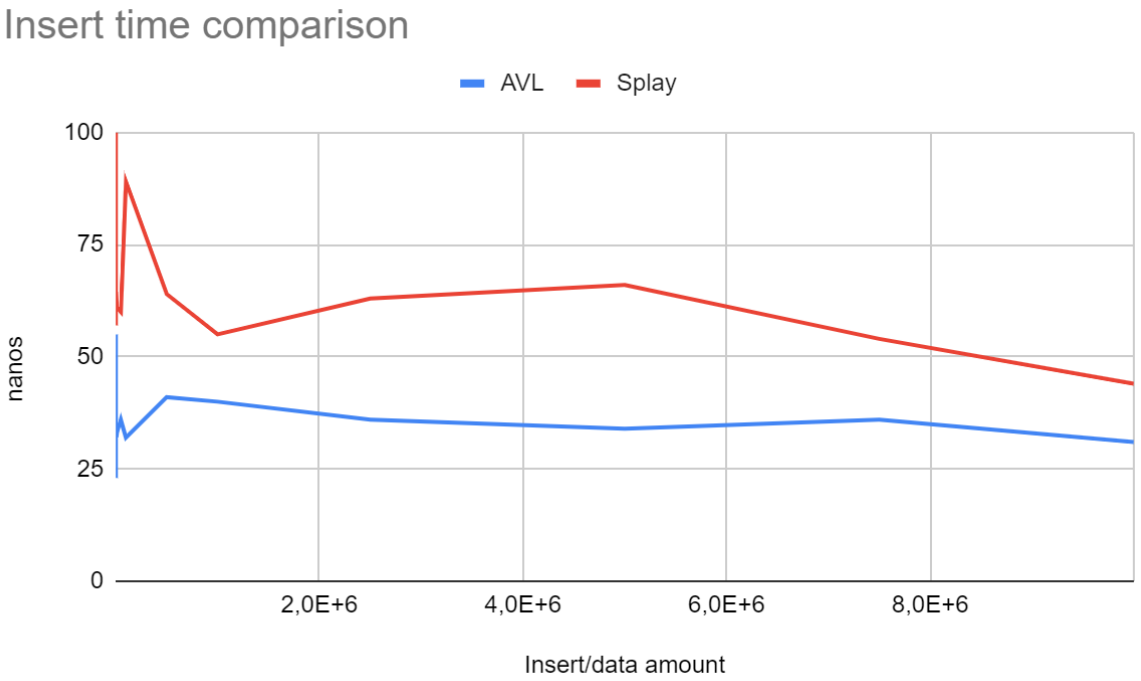
3.3.1. Binary tree time comparison

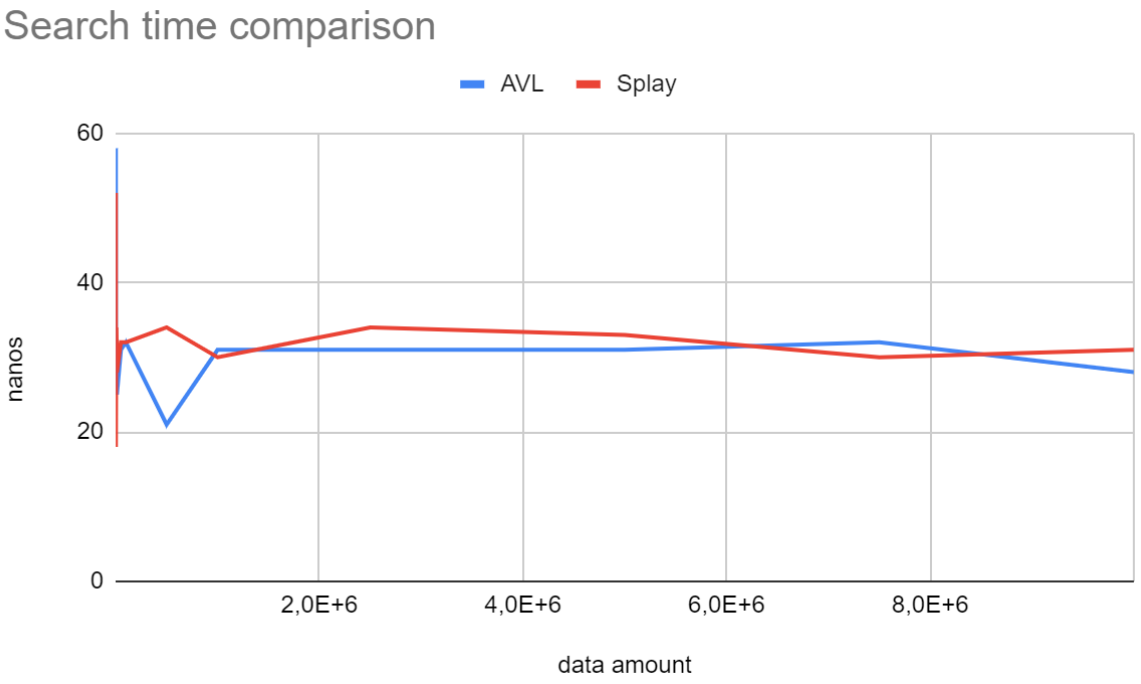
In this case value 12345678 was inserted into 15 datasets of this size 10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 2500000, 5000000, 7500000 and 10000000. And all operations were repeated 1000 times to determine the best average duration.

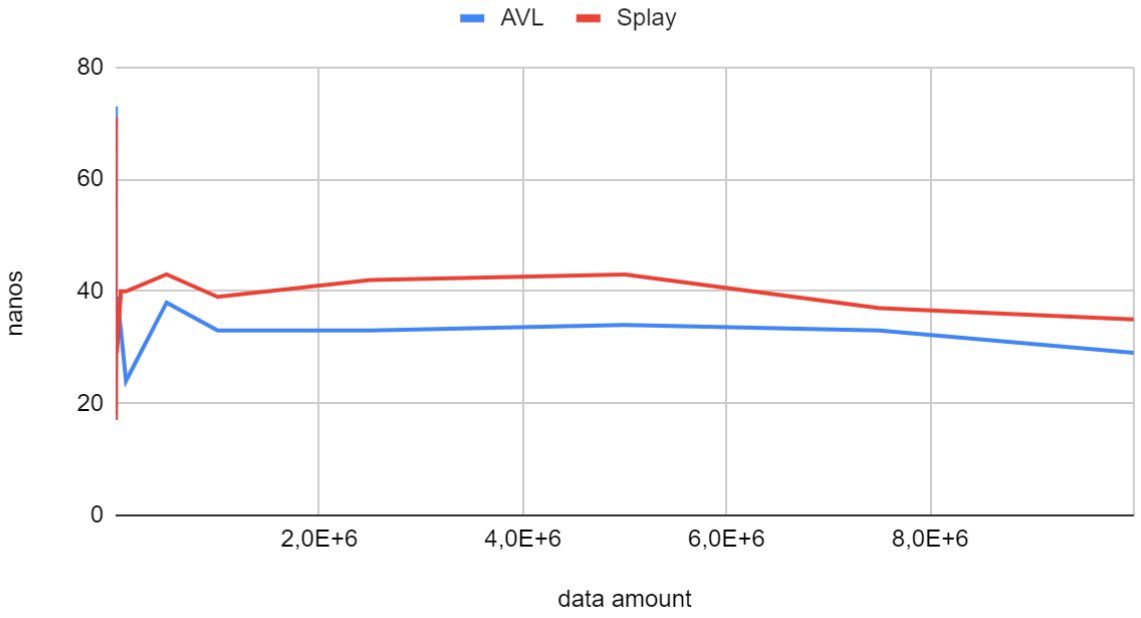




In comparison of AVL and Splay we can see that AVL is much better at inserting, because in the Splay tree, after insertion of the node a high amount of rotation needs to be done to splay the inserted node to the root. Searching and deleting is similar to both.

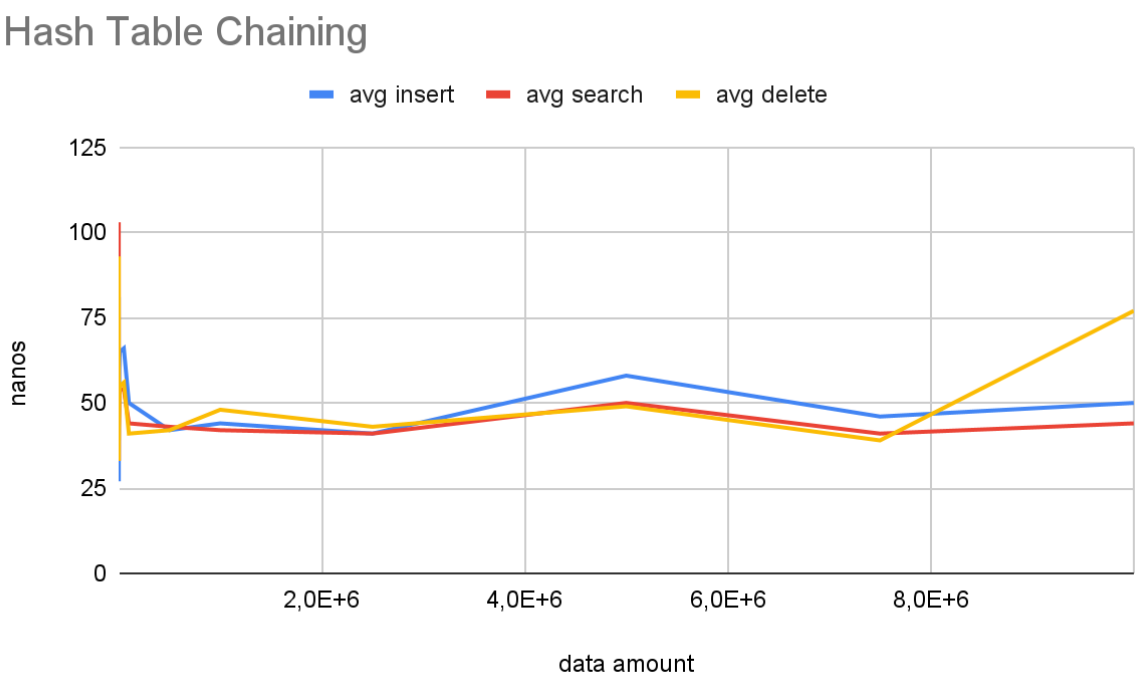


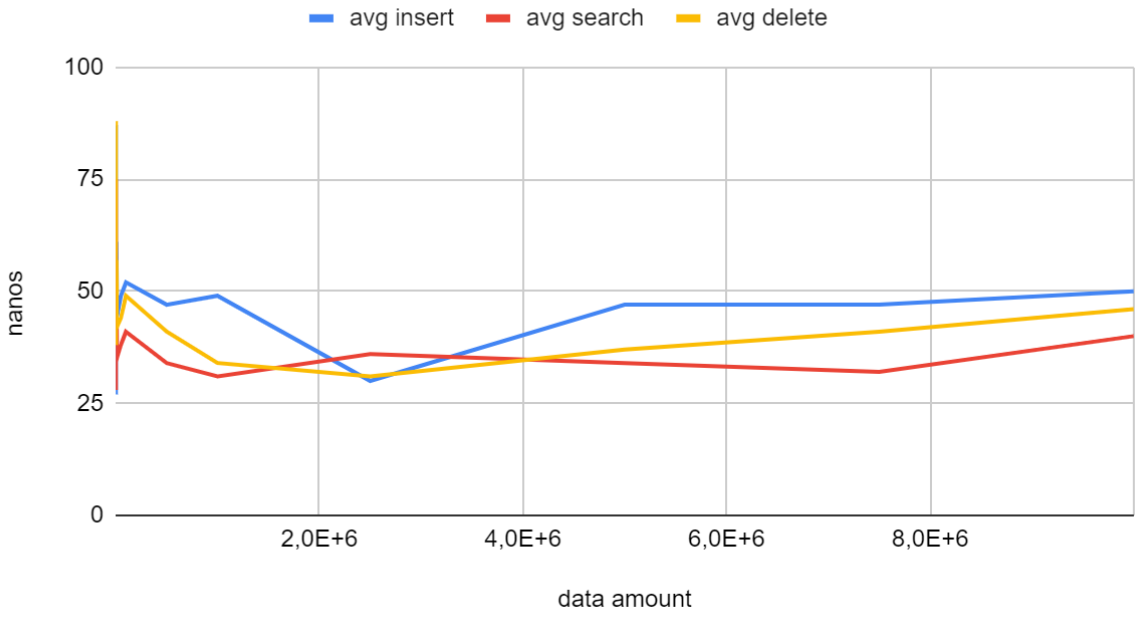




3.3.2. Hash table time comparison

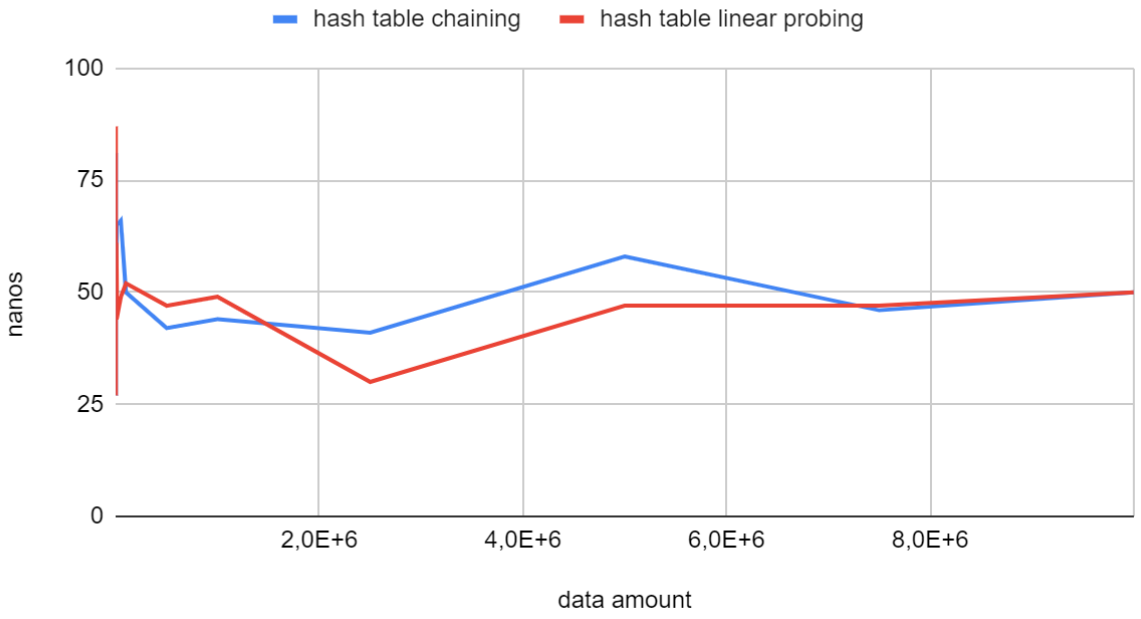
The initial table capacity was 10 and load factor for chaining was 1.5 and for linear probing 0.75. Testing of hash tables was done by the same datasets but the inserted entry was “key: abcdefghij; value: Riko”. And all operations were repeated again 1000 times to determine the best average duration.



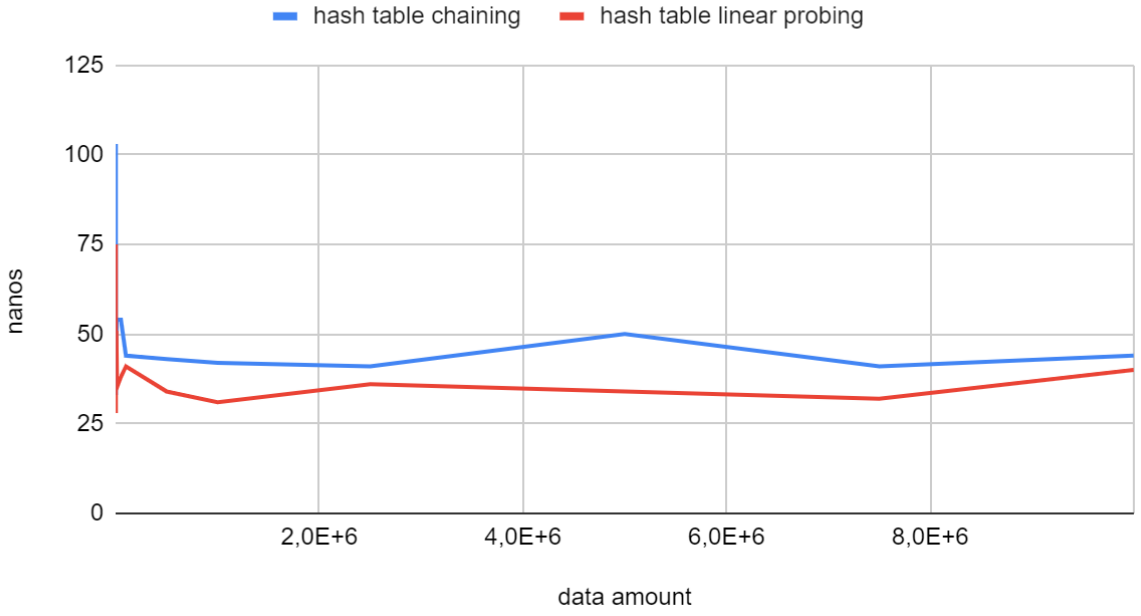


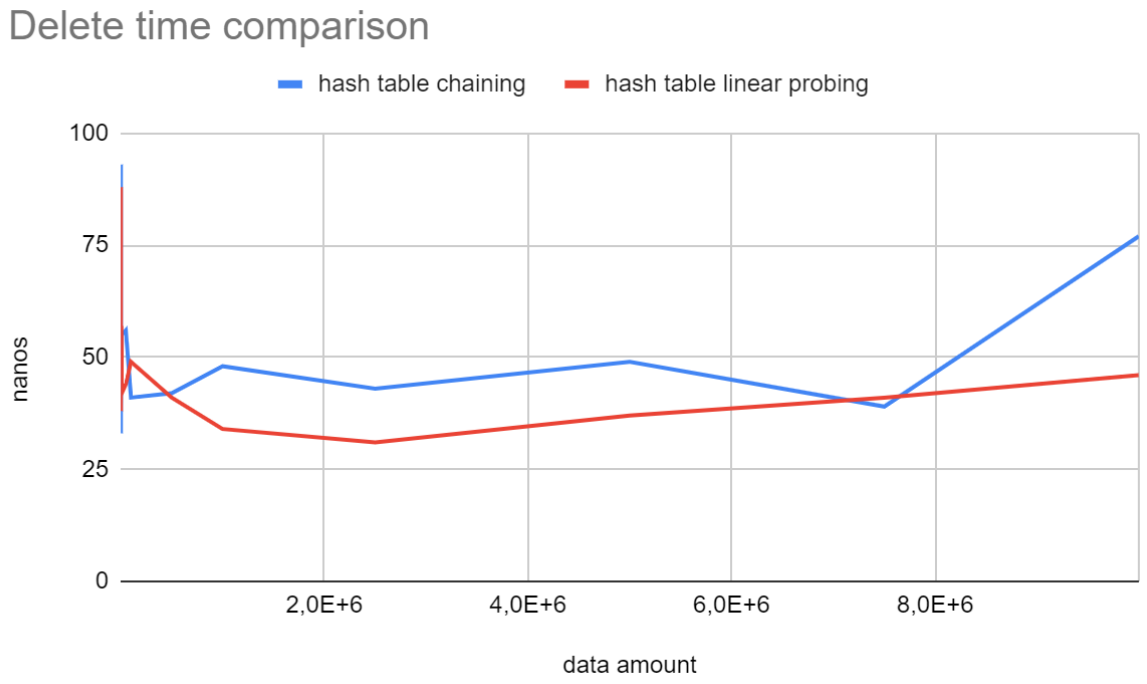
In comparison of chaining and linear probing we can see that table with chaining is slightly better at all insert, search and also delete.

Insert time comparison



Search time comparison





4. Conclusion

In conclusion, the results show that each data structure has its own advantages and disadvantages. So implementation of the right one depends on final specific requirements.

AVL trees maintain balance by performing rotations to ensure that the heights of the left and right subtrees differ by at most one. So the worst-case time complexity for search, insertion, and deletion is O(log n), where n is the number of nodes in the tree.

On the other hand, Splay trees bring recently accessed nodes to the root of the tree to improve access time. This means that frequently accessed nodes will be closer to the root, so it provides a shorter access time. The worst-case time complexity for search, insertion, and deletion in a Splay tree is also O(log n). So if we need to search random data, the AVL is highly effective for that, otherwise if we need to access the most used data then we surely need to implement a Splay tree.

From the graphs we demonstrate that the time complexity for inserting an element in a hash table with chaining is O(1) on average, assuming that the hash function distributes the keys evenly across the buckets.

The time complexity for inserting an element in a hash table with linear probing is also O(1) on average, assuming that the table is not too full. From the results we can say that the implementation and time measurements were successful.

5. References

https://www.programiz.com/dsa/avl-tree
https://www.programiz.com/dsa/hash-table
https://www.programiz.com/dsa/hashing
https://algorithmtutor.com/Data-Structures/Tree/AVL-Trees/
https://algorithmtutor.com/Data-Structures/Tree/Splay-Trees/
https://www.geeksforgeeks.org/deletion-in-an-avl-tree/?ref=lbp
https://www.geeksforgeeks.org/searching-in-splay-tree/?ref=lbp
https://www.youtube.com/watch?v=-9sHvAnLN_w
https://www.youtube.com/watch?v=qMmqOHr75b8