# **Data structure and algorithms**
documentation for assignment 1

Teacher : Ing. Mohammad Yusuf Momand , MSc.
Email: xmomand@stuba.sk

Student: Alexandra Vetrov
Email: xvetrov@stuba.sk

## Contents

Introduction ........ 3
AVL tree ........ 4
Red-Black tree ........ 7
Hash Tables ........ 10
    Quadratic probing ........ 10
    Separate Chaining ........ 13
Testing ........ 16
    Tree testing ........ 16
    Tables testing ........ 18
        Scenario (Insert->Search->Delete) ........ 20
Conclusion ........ 21

Alexandra Vetrov ID:123102

## Introduction

Theme for assignment one: Search in dynamic sets.

Task: implement and then compare 4 implementations of data structures in terms of the effectiveness of insert, delete and search operations in different situations.

Implementations that I have done in my task are : Splay tree,  Red-Black tree, Hash tables with collision resolution using quadratic probing and chaining.

Programming language : Java.

Code editor : IntelliJ IDEA.

Processor:	11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz   2.80 GHz

RAM:	16.0

System type:	64-bit operating system, x64-based processor

OS:	Windows 11

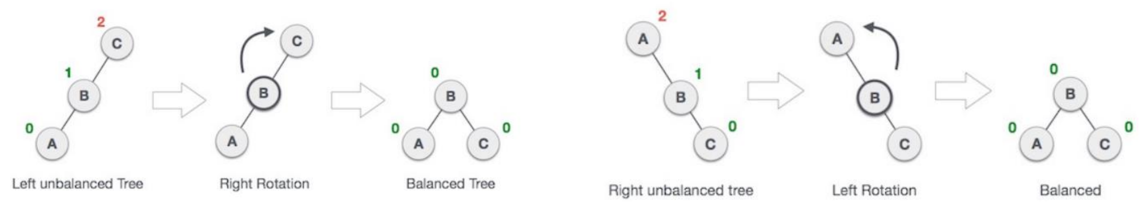
Alexandra Vetrov ID:123102

## AVL tree

An AVL tree is a self-balancing binary search tree. It's similar to a regular binary search tree, but it maintains a balance property to ensure that the height of the tree remains logarithmic.

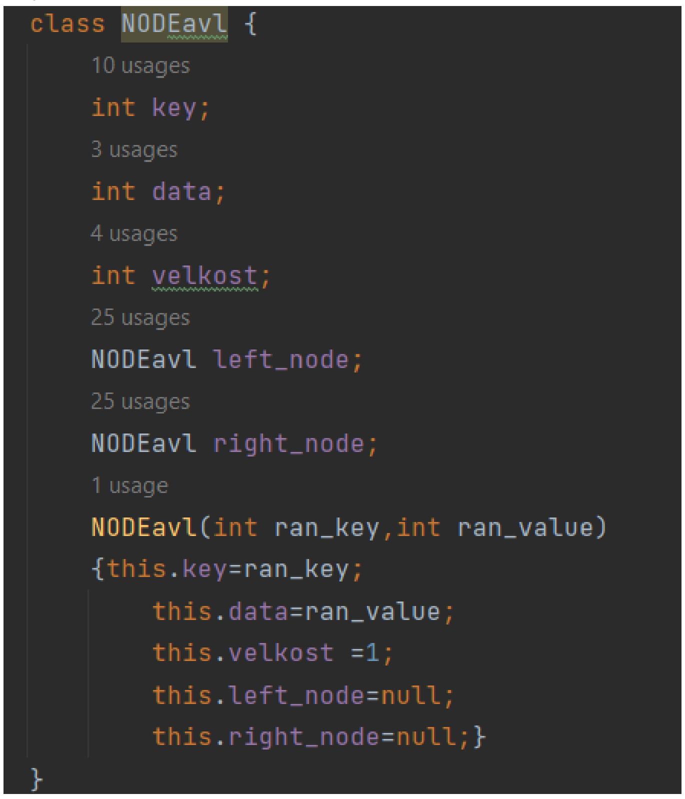
AVL tree, every node has a balance factor, which is the difference between the heights of its left and right subtrees. If the balance factor of a node is greater than 1 or less than -1, then the tree is unbalanced, and it needs to be rebalanced.

There are four types of rotations that can be performed to balance the tree: left rotation, right rotation, left-right rotation, and right-left rotation. These rotations adjust the positions of the nodes to ensure that the balance factor is maintained.



In terms of functions, in my AVL tree I have this functions:

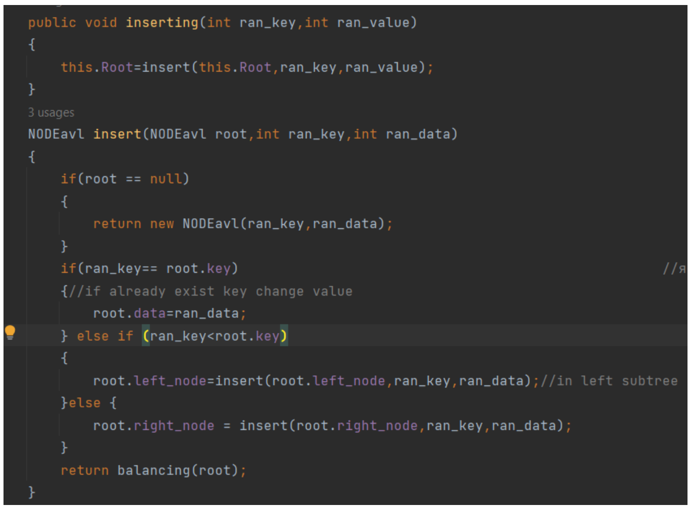
- The NODEavl class represents a node in the AVL tree, which contains a key, a data value, a size value, and references to its left and right child nodes. The avltree class represents the AVL tree itself and contains a reference to the root node.



```
class NODEavl {
        10 usages
        int key;
        3 usages
        int data;
        4 usages
        int velkost;
        25 usages
        NODEavl left_node;
        25 usages
        NODEavl right_node;
        1 usage
        NODEavl(int ran_key,int ran_value)
        {this.key=ran_key;
            this.data=ran_value;
            this.velkost =1;
            this.left_node=null;
            this.right_node=null;}
}
```

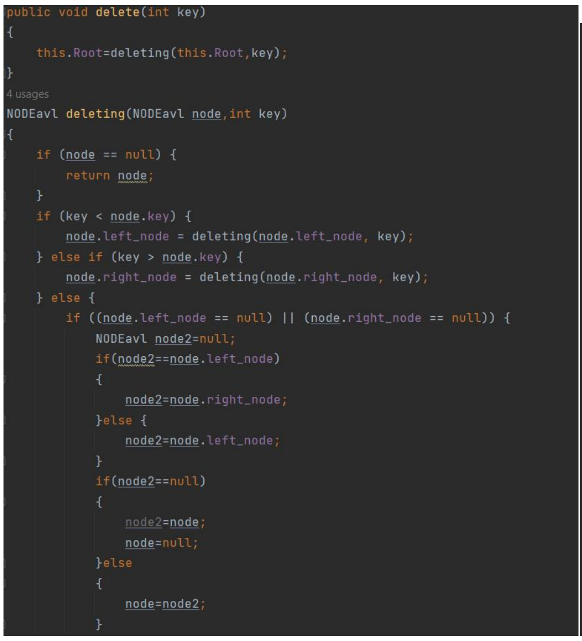
- insert(node): This function is used to insert a new node into the AVL tree. It takes in a node as an argument and inserts it into the tree while maintaining the AVL balance property.

Alexandra Vetrov ID:123102

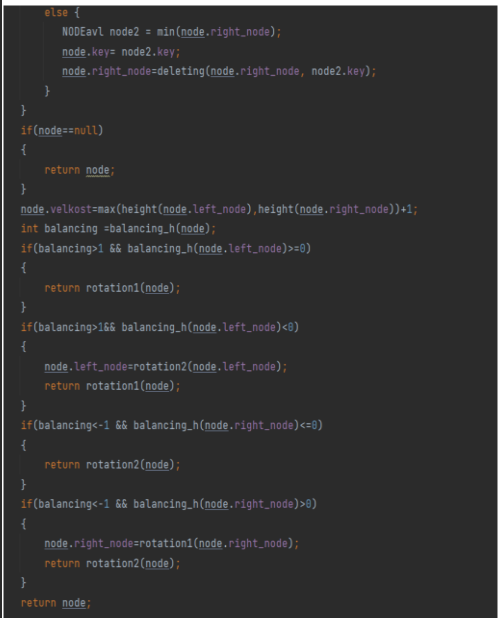


```java
public void inserting(int ran_key,int ran_value)
{
    this.Root=insert(this.Root,ran_key,ran_value);
}
3 usages
NODEavl insert(NODEavl root,int ran_key,int ran_data)
{
    if(root == null)
    {
        return new NODEavl(ran_key,ran_data);
    }
    if(ran_key== root.key)                                       //я
    {//if already exist key change value
        root.data=ran_data;
    } else if (ran_key<root.key)
    {
        root.left_node=insert(root.left_node,ran_key,ran_data);//in left subtree
    }else {
        root.right_node = insert(root.right_node,ran_key,ran_data);
    }
    return balancing(root);
}
```

- delete(node): This function is used to delete a node from the AVL tree. It takes in a node as an argument and removes it from the tree while maintaining the AVL balance property.

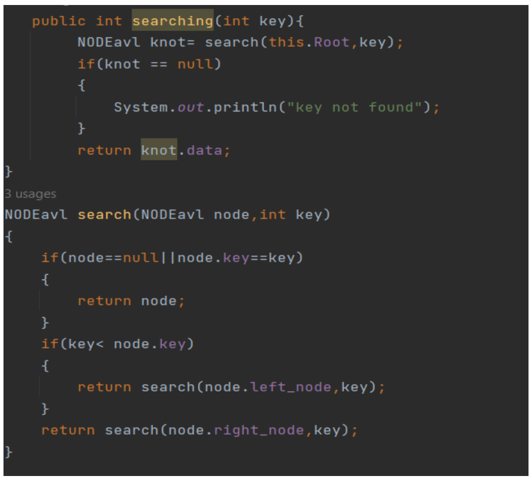


```java
public void delete(int key)
{
    this.Root=deleting(this.Root,key);
}
4 usages
NODEavl deleting(NODEavl node,int key)
{
    if (node == null) {
        return node;
    }
    if (key < node.key) {
        node.left_node = deleting(node.left_node, key);
    } else if (key > node.key) {
        node.right_node = deleting(node.right_node, key);
    } else {
        if ((node.left_node == null) || (node.right_node == null)) {
            NODEavl node2=null;
            if(node2==node.left_node)
            {
                node2=node.right_node;
            }else {
                node2=node.left_node;
            }
            if(node2==null)
            {
                node2=node;
                node=null;
            }else
            {
                node=node2;
            }
```



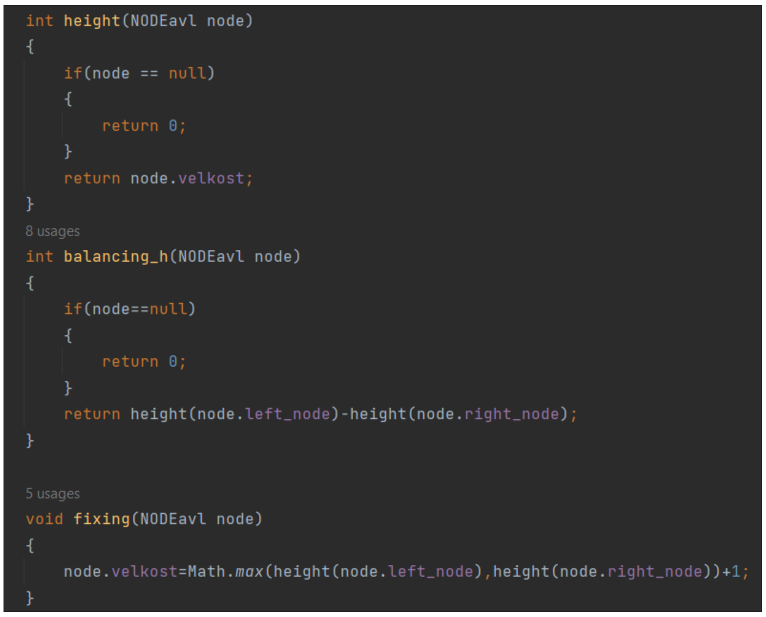
```java
        else {
            NODEavl node2 = min(node.right_node);
            node.key= node2.key;
            node.right_node=deleting(node.right_node, node2.key);
        }
    }
    if(node==null)
    {
        return node;
    }
    node.velkost=max(height(node.left_node),height(node.right_node))+1;
    int balancing =balancing_h(node);
    if(balancing>1 && balancing_h(node.left_node)>=0)
    {
        return rotation1(node);
    }
    if(balancing>1&& balancing_h(node.left_node)<0)
    {
        node.left_node=rotation2(node.left_node);
        return rotation1(node);
    }
    if(balancing<-1 && balancing_h(node.right_node)<=0)
    {
        return rotation2(node);
    }
    if(balancing<-1 && balancing_h(node.right_node)>0)
    {
        node.right_node=rotation1(node.right_node);
        return rotation2(node);
    }
    return node;
```

- search(key): This function is used to search for a node with a specific key value in the AVL tree. It takes in a key as an argument and returns the node if it exists in the tree, otherwise returns null.



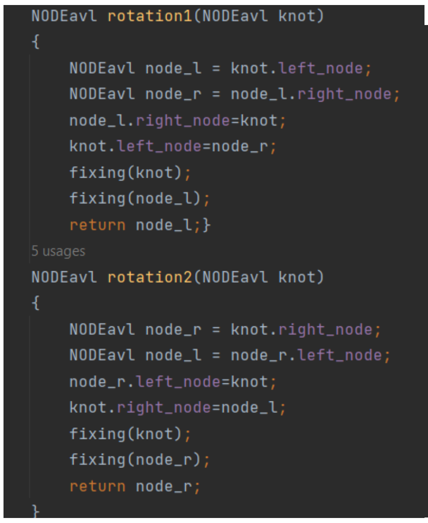
```java
    public int searching(int key){
        NODEavl knot= search(this.Root,key);
        if(knot == null)
        {
            System.out.println("key not found");
        }
        return knot.data;
}
3 usages
NODEavl search(NODEavl node,int key)
{
    if(node==null||node.key==key)
    {
        return node;
    }
    if(key< node.key)
    {
        return search(node.left_node,key);
    }
    return search(node.right_node,key);
}
```

- The height method returns the height of a node, which is defined as the maximum depth of its subtrees. The balancing_h method returns the balance factor of a node, which is defined as the difference between the heights of its left and right subtrees.
- The fixing method updates the size value of a node based on the sizes of its left and right subtrees.

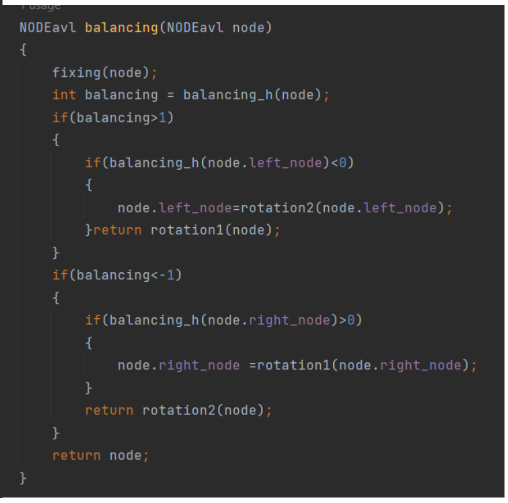


```java
int height(NODEavl node)
{
    if(node == null)
    {
        return 0;
    }
    return node.velkost;
}
8 usages
int balancing_h(NODEavl node)
{
    if(node==null)
    {
        return 0;
    }
    return height(node.left_node)-height(node.right_node);
}

5 usages
void fixing(NODEavl node)
{
    node.velkost=Math.max(height(node.left_node),height(node.right_node))+1;
}
```

- The rotation1 and rotation2 methods perform left and right rotations, respectively, on a given node to balance the tree. A left rotation on a node knot involves rotating its left child node_l up and promoting knot to the right child of node_l. A right rotation on a node knot involves rotating its right child node_r up and promoting knot to the left child of node_r.



```java
NODEavl rotation1(NODEavl knot)
{
    NODEavl node_l = knot.left_node;
    NODEavl node_r = node_l.right_node;
    node_l.right_node=knot;
    knot.left_node=node_r;
    fixing(knot);
    fixing(node_l);
    return node_l;}
5 usages
NODEavl rotation2(NODEavl knot)
{
    NODEavl node_r = knot.right_node;
    NODEavl node_l = node_r.left_node;
    node_r.left_node=knot;
    knot.right_node=node_l;
    fixing(knot);
    fixing(node_r);
    return node_r;
}
```



```java
1 usage
NODEavl balancing(NODEavl node)
{
    fixing(node);
    int balancing = balancing_h(node);
    if(balancing>1)
    {
        if(balancing_h(node.left_node)<0)
        {
            node.left_node=rotation2(node.left_node);
        }return rotation1(node);
    }
    if(balancing<-1)
    {
        if(balancing_h(node.right_node)>0)
        {
            node.right_node =rotation1(node.right_node);
        }
        return rotation2(node);
    }
    return node;
}
```

- The balancing method balances the tree by performing the appropriate rotation(s) on a given node, based on its balance factor. If the balance factor is greater than 1, a left rotation is performed if the balance factor of the left child is less than 0, followed by a right rotation on the given node. If the balance factor is less than -1, a right rotation is performed if the balance factor of the right child is greater than 0, followed by a left rotation on the given node.

.

## Red-Black tree

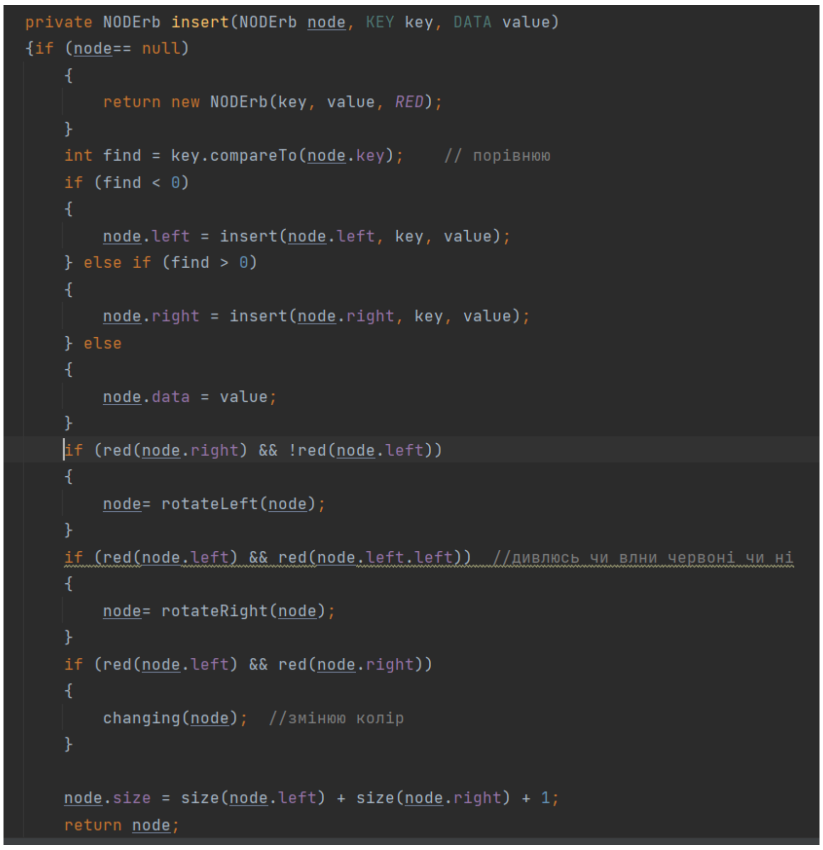
A red-black tree is a type of binary search tree that is designed to be self-balancing, which means that it automatically adjusts the structure of the tree to ensure that its height stays within a certain range. This makes red-black trees very efficient for certain types of operations, such as searching, inserting, and deleting nodes.

In a red-black tree, each node is either red or black. The tree is constructed so that it satisfies the following properties:

- Every node is either red or black.
- The root node is always black.
- Every leaf (i.e., a node with no children) is black.
- If a node is red, then both of its children must be black.
- Every path from a given node to any of its descendant leaves contains the same number of black nodes.

The red-black tree has several functions that are used to manipulate the tree structure, including:

Insertion: This function is used to add a new node to the tree. The algorithm works by first adding the node as a leaf, then adjusting the colors and positions of nodes as needed to ensure that the tree remains balanced

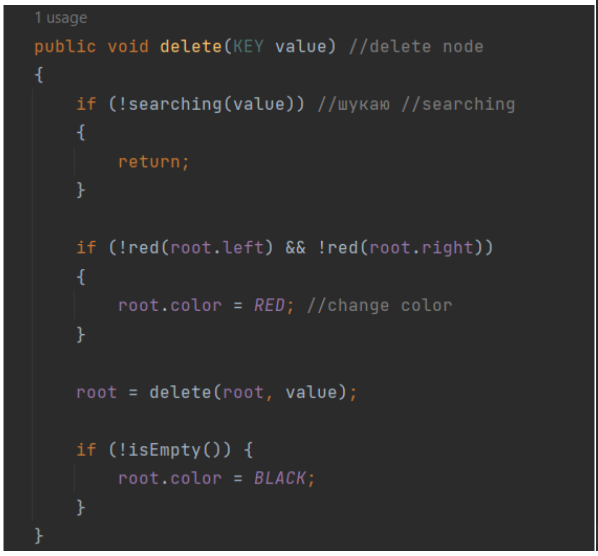


```
private NODErb insert(NODErb node, KEY key, DATA value)
{if (node== null)
    {
        return new NODErb(key, value, RED);
    }
    int find = key.compareTo(node.key);    // порівнюю
    if (find < 0)
    {
        node.left = insert(node.left, key, value);
    } else if (find > 0)
    {
        node.right = insert(node.right, key, value);
    } else
    {
        node.data = value;
    }
    if (red(node.right) && !red(node.left))
    {
        node= rotateLeft(node);
    }
    if (red(node.left) && red(node.left.left))  //дивлюсь чи влни червоні чи ні
    {
        node= rotateRight(node);
    }
    if (red(node.left) && red(node.right))
    {
        changing(node);  //змінюю колір
    }

    node.size = size(node.left) + size(node.right) + 1;
    return node;
```

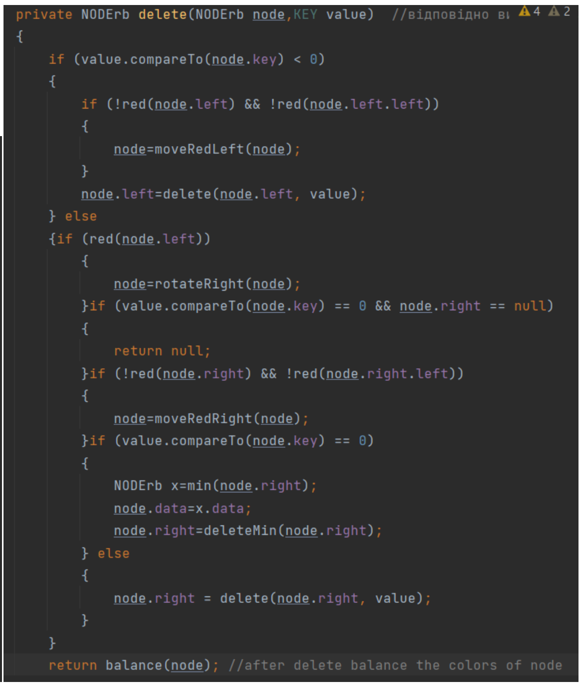
.

Alexandra Vetrov ID:123102

Deletion: This function is used to remove a node from the tree. The algorithm works by first identifying the node to be deleted, then replacing it with either its successor (i.e., the next highest node in the tree) or its predecessor (i.e., the next lowest node in the tree), and then adjusting the colors and positions of nodes as needed to ensure that the tree remains balanced.

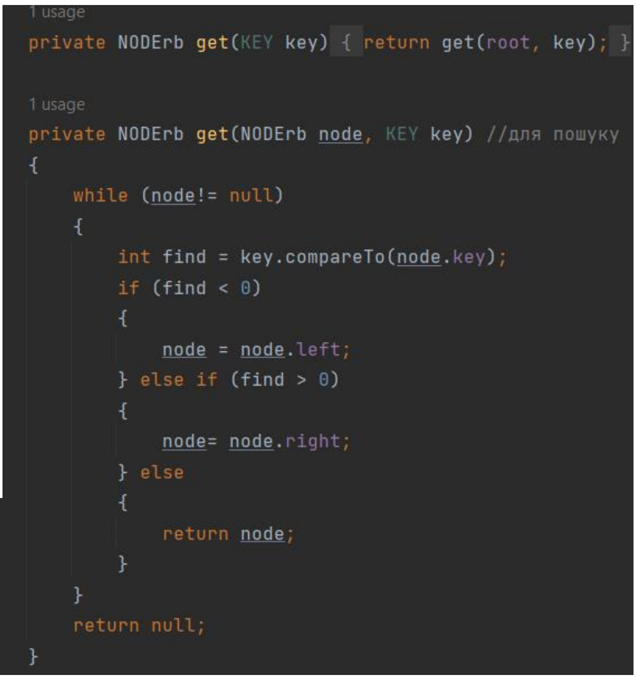


```
1 usage
public void delete(KEY value) //delete node
{
    if (!searching(value)) //шукаю //searching
    {
        return;
    }

    if (!red(root.left) && !red(root.right))
    {
        root.color = RED; //change color
    }

    root = delete(root, value);

    if (!isEmpty()) {
        root.color = BLACK;
    }
}
```

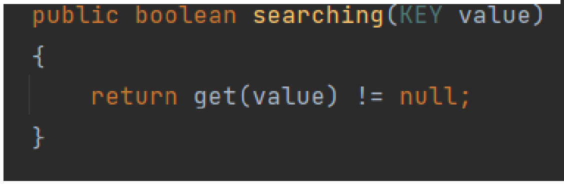


```
private NODErb delete(NODErb node,KEY value)  //відповідно ві  ⚠4 ⚠2
{
    if (value.compareTo(node.key) < 0)
    {
        if (!red(node.left) && !red(node.left.left))
        {
            node=moveRedLeft(node);
        }
        node.left=delete(node.left, value);
    } else
    {if (red(node.left))
        {
            node=rotateRight(node);
        }if (value.compareTo(node.key) == 0 && node.right == null)
        {
            return null;
        }if (!red(node.right) && !red(node.right.left))
        {
            node=moveRedRight(node);
        }if (value.compareTo(node.key) == 0)
        {
            NODErb x=min(node.right);
            node.data=x.data;
            node.right=deleteMin(node.right);
        } else
        {
            node.right = delete(node.right, value);
        }
    }
    return balance(node); //after delete balance the colors of node
```

Search: This function is used to locate a specific node in the tree. The algorithm works by starting at the root node and recursively traversing the tree until the desired node is found.

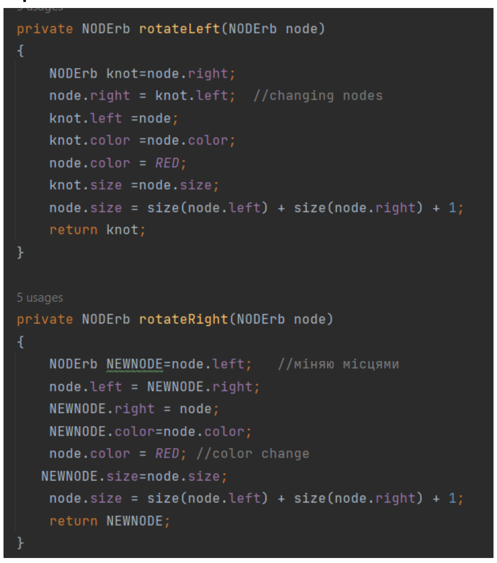


```
1 usage
private NODErb get(KEY key) { return get(root, key); }

1 usage
private NODErb get(NODErb node, KEY key) //для пошуку
{
    while (node!= null)
    {
        int find = key.compareTo(node.key);
        if (find < 0)
        {
            node = node.left;
        } else if (find > 0)
        {
            node= node.right;
        } else
        {
            return node;
        }
    }
    return null;
}
```



```
public boolean searching(KEY value)
{
    return get(value) != null;
}
```

Rotations: Rotations are used to adjust the positions of nodes within the tree. There are two types of rotations: left rotations and right rotations. A left rotation moves the node to the left and its right child up, while a right rotation moves the node to the right and its left child up.



```
private NODErb rotateLeft(NODErb node)
{
    NODErb knot=node.right;
    node.right = knot.left;  //changing nodes
    knot.left =node;
    knot.color =node.color;
    node.color = RED;
    knot.size =node.size;
    node.size = size(node.left) + size(node.right) + 1;
    return knot;
}

5 usages
private NODErb rotateRight(NODErb node)
{
    NODErb NEWNODE=node.left;   //міняю місцями
    node.left = NEWNODE.right;
    NEWNODE.right = node;
    NEWNODE.color=node.color;
    node.color = RED; //color change
   NEWNODE.size=node.size;
    node.size = size(node.left) + size(node.right) + 1;
    return NEWNODE;
}
```

Overall, the red-black tree is a very useful data structure that is widely used in computer science for its efficient and self-balancing properties. Its functions, including insertion, deletion, search, and rotations, make it a powerful tool for managing large sets of data in an efficient and scalable way.

Alexandra Vetrov ID:123102

# Hash Tables

Hash tables are a common data structure used in computer science to quickly store and retrieve data. They work by using a hash function to map a unique key to a specific index in an array. The value associated with the key is then stored in that array index.

However, sometimes two or more keys can map to the same index in the array, which is called a collision. There are various techniques to handle collisions, one of which is called quadratic probing.

## Quadratic probing

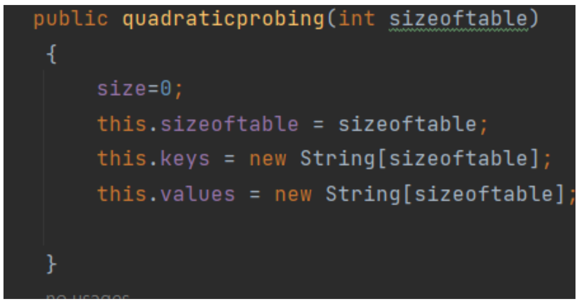
Quadratic probing is a type of open addressing collision resolution strategy that tries to find the next available slot in the array by incrementing the index in a quadratic manner instead of a linear manner.

For example, if a collision occurs at index 5, the algorithm will check the next available slot at (5 + 1^2) = 6. If that slot is also occupied, the algorithm will try the next slot at (5 + 2^2) = 9, and so on, until an empty slot is found.

While quadratic probing can help to evenly distribute items in the hash table and reduce collisions, it can become less effective as the table becomes more densely populated. Therefore, other collision resolution methods such as chaining may be more suitable in certain situations.

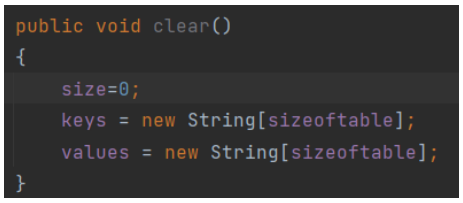
In my  implementation of quadratic probing I have:

- quadraticprobing(int sizeoftable): This is the constructor function of the class. It takes an integer sizeoftable as an argument and initializes the size variable to 0. It also initializes the keys and values arrays with the size sizeoftable.



```
public quadraticprobing(int sizeoftable)
 {
     size=0;
     this.sizeoftable = sizeoftable;
     this.keys = new String[sizeoftable];
     this.values = new String[sizeoftable];

 }
```

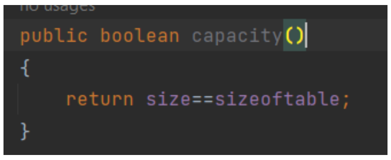
- clear(): This function clears the hash table by setting the size variable to 0 and creating new empty arrays for keys and values.



```
public void clear()
{
    size=0;
    keys = new String[sizeoftable];
    values = new String[sizeoftable];
}
```

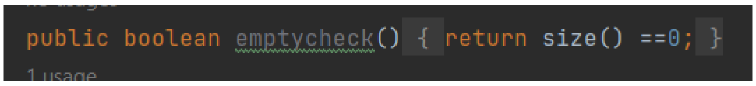
- capacity(): This function checks if the hash table is at full capacity. It returns true if the size variable is equal to the sizeoftable variable, indicating that the hash table is full.

Alexandra Vetrov ID:123102



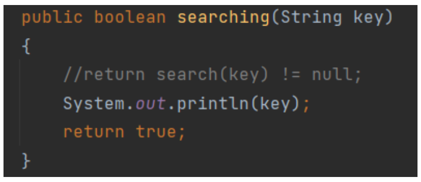
```java
public boolean capacity()
{
    return size==sizeoftable;
}
```

- emptycheck(): This function checks if the hash table is empty. It returns true if the size() function returns 0.



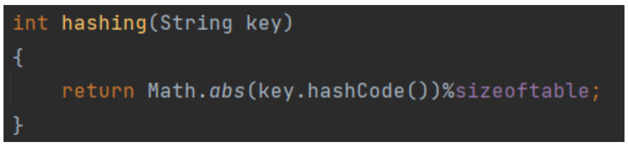
```java
public boolean emptycheck() { return size() ==0; }
1 usage
```

- searching(String key): This function takes a string key as an argument and returns true if the key is found in the hash table. It does this by calling the search(String key) function and checking if the return value is null.



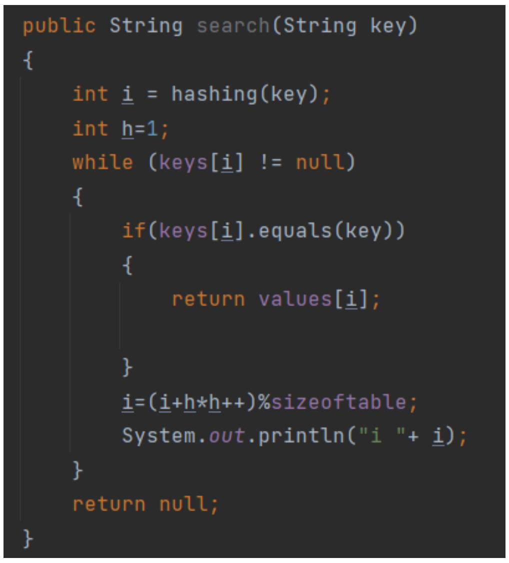
```java
public boolean searching(String key)
{
    //return search(key) != null;
    System.out.println(key);
    return true;
}
```

- hashing(String key): This function takes a string key as an argument and returns the index in the keys and values arrays where the key-value pair should be stored based on the hash function. The hash function used in this code is Math.abs(key.hashCode())%sizeoftable, which takes the absolute value of the hash code of the string key and performs modulo operation with the sizeoftable variable to get the index.



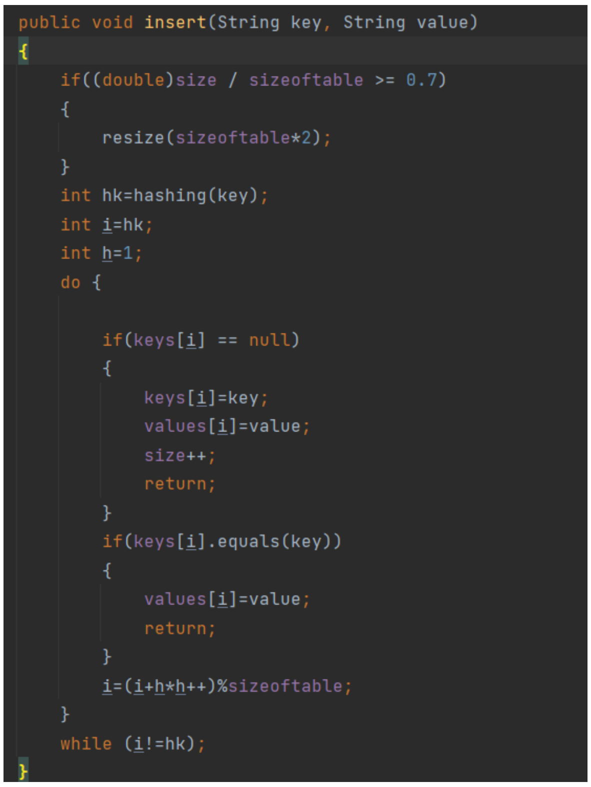
```java
int hashing(String key)
{
    return Math.abs(key.hashCode())%sizeoftable;
}
```

- search(String key): This function takes a string key as an argument and returns the value associated with the key in the hash table. It does this by using the hash function to get the index where the key-value pair should be stored, and then uses quadratic probing to find the actual index where the key-value pair is stored. Quadratic probing is a technique where a quadratic function is used to determine the next index to check if the current index is already occupied. The function returns null if the key is not found in the hash table.



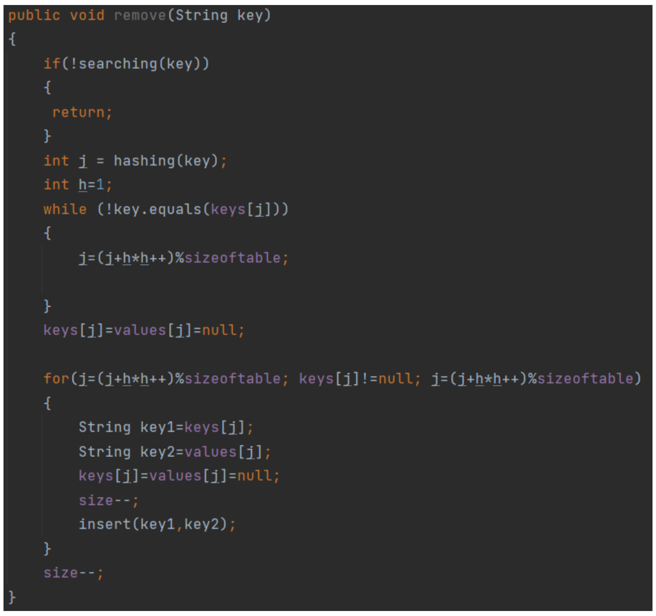
```
public String search(String key)
{
    int i = hashing(key);
    int h=1;
    while (keys[i] != null)
    {
        if(keys[i].equals(key))
        {
            return values[i];

        }
        i=(i+h*h++)%sizeoftable;
        System.out.println("i "+ i);
    }
    return null;
}
```

- insert(String key, String value): This function takes two string arguments key and value and inserts the key-value pair into the hash table. If the hash table is at 70% capacity, it calls the resize(int size) function to double the size of the hash table. It then uses the hash function to get the index where the key-value pair should be stored, and uses quadratic probing to find the actual index where the key-value pair can be stored. If the key already exists in the hash table, the function updates the value associated with the key.



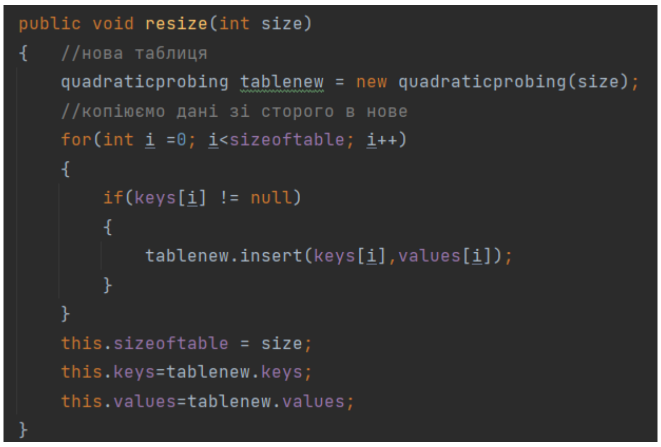
```
public void insert(String key, String value)
{
    if((double)size / sizeoftable >= 0.7)
    {
        resize(sizeoftable*2);
    }
    int hk=hashing(key);
    int i=hk;
    int h=1;
    do {

        if(keys[i] == null)
        {
            keys[i]=key;
            values[i]=value;
            size++;
            return;
        }
        if(keys[i].equals(key))
        {
            values[i]=value;
            return;
        }
        i=(i+h*h++)%sizeoftable;
    }
    while (i!=hk);
}
```

- remove(String key): This function takes a string key as an argument and removes the key-value pair from the hash table. It first calls the searching(String key) function to check if the key exists in the hash table. If it does, it uses quadratic probing to find the index where the key-value pair is stored, sets the key-value pair to null, and then rehashes and inserts all the remaining key-value pairs to maintain the integrity of the hash table.



```java
public void remove(String key)
{
    if(!searching(key))
    {
     return;
    }
    int j = hashing(key);
    int h=1;
    while (!key.equals(keys[j]))
    {
        j=(j+h*h++)%sizeoftable;

    }
    keys[j]=values[j]=null;

    for(j=(j+h*h++)%sizeoftable; keys[j]!=null; j=(j+h*h++)%sizeoftable)
    {
        String key1=keys[j];
        String key2=values[j];
        keys[j]=values[j]=null;
        size--;
        insert(key1,key2);
    }
    size--;
}
```

- resize(int size): This function takes an integer size as an argument and creates a new hash table with the new size. It then iterates through the original hash table and inserts all the key-value pairs into the new hash



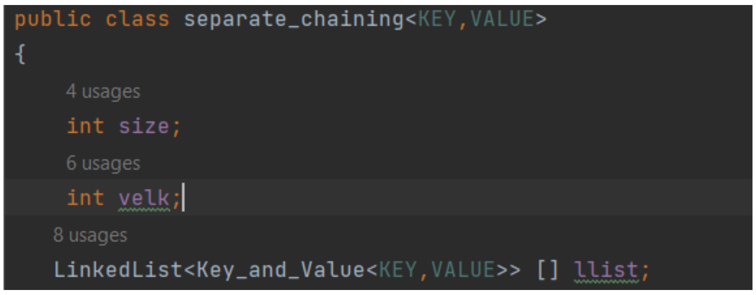
```java
public void resize(int size)
{   //нова таблиця
    quadraticprobing tablenew = new quadraticprobing(size);
    //копіюємо дані зі сторого в нове
    for(int i =0; i<sizeoftable; i++)
    {
        if(keys[i] != null)
        {
            tablenew.insert(keys[i],values[i]);
        }
    }
    this.sizeoftable = size;
    this.keys=tablenew.keys;
    this.values=tablenew.values;
}
```

Another method for collision resolution is separate chaining.

## Separate Chaining

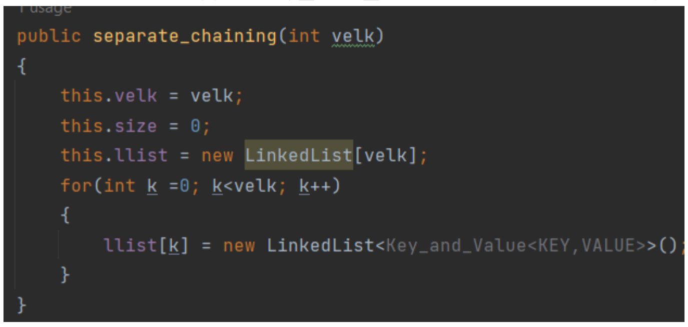
- The code defines a Java class called separate_chaining that implements a hash table using separate chaining as the collision resolution method.The class takes two type parameters: KEY and VALUE, which are used to represent the keys and values that will be stored in the hash table.The class has a size and velk variable, which are used to keep track of the number of elements in the hash table and the size of the underlying array used to store the key-value pairs, respectively

Alexandra Vetrov ID:123102

.


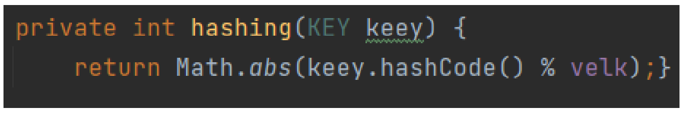
```
public class separate_chaining<KEY,VALUE>
{
    4 usages
    int size;
    6 usages
    int velk;
  8 usages
  LinkedList<Key_and_Value<KEY,VALUE>> [] llist;
```

- Constructor that takes an integer value representing the size of the hash table to be created. This constructor initializes the size and velk variables, and creates an array of LinkedLists of type Key_and_Value to store the key-value pairs.



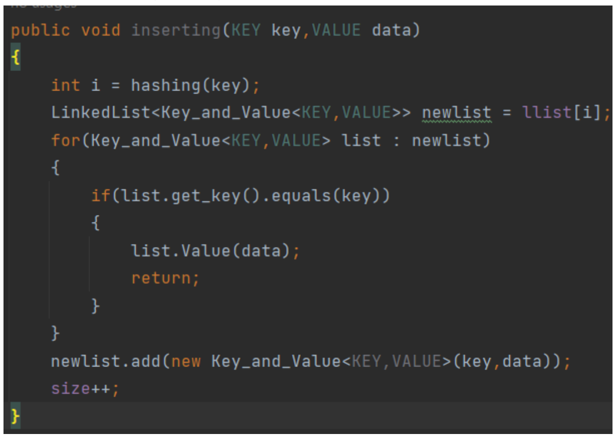
```
public separate_chaining(int velk)
{
    this.velk = velk;
    this.size = 0;
    this.llist = new LinkedList[velk];
    for(int k =0; k<velk; k++)
    {
        llist[k] = new LinkedList<Key_and_Value<KEY,VALUE>>();
    }
}
```

- The class defines a private hashing function that takes a key as input and returns an integer value representing the index of the LinkedList in the array where the key-value pair will be stored.



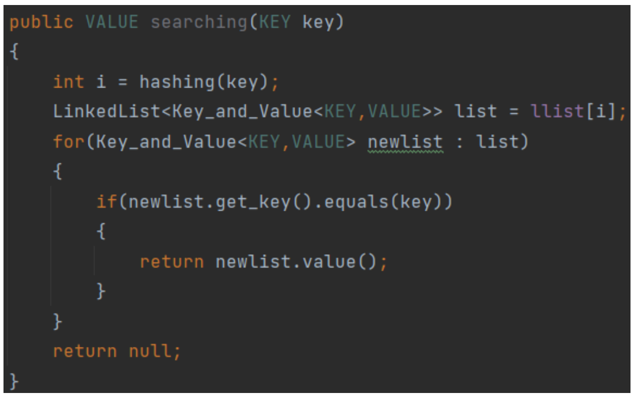
```
private int hashing(KEY keey) {
    return Math.abs(keey.hashCode() % velk);}
```

- Inserting function that takes a key and a value as input and inserts the key-value pair into the hash table. If the key already exists in the hash table, the value associated with the key is updated with the new value.



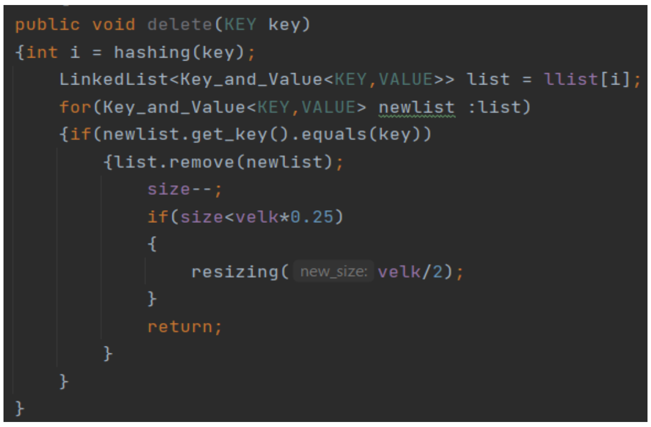
```
public void inserting(KEY key,VALUE data)
{
    int i = hashing(key);
    LinkedList<Key_and_Value<KEY,VALUE>> newlist = llist[i];
    for(Key_and_Value<KEY,VALUE> list : newlist)
    {
        if(list.get_key().equals(key))
        {
            list.Value(data);
            return;
        }
    }
    newlist.add(new Key_and_Value<KEY,VALUE>(key,data));
    size++;
}
```

- Searching function that takes a key as input and returns the value associated with the key in the hash table, or null if the key is not found.



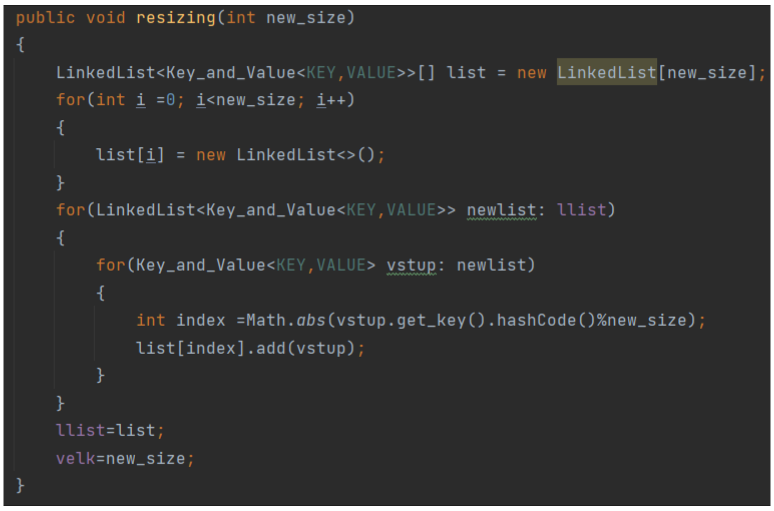
```
public VALUE searching(KEY key)
{
    int i = hashing(key);
    LinkedList<Key_and_Value<KEY,VALUE>> list = llist[i];
    for(Key_and_Value<KEY,VALUE> newlist : list)
    {
        if(newlist.get_key().equals(key))
        {
            return newlist.value();
        }
    }
    return null;
}
```

- Delete function that takes a key as input and removes the key-value pair from the hash table. If the size of the hash table falls below 25% of the array size, the hash table is resized to half its current size.



```
public void delete(KEY key)
{int i = hashing(key);
    LinkedList<Key_and_Value<KEY,VALUE>> list = llist[i];
    for(Key_and_Value<KEY,VALUE> newlist :list)
    {if(newlist.get_key().equals(key))
        {list.remove(newlist);
            size--;
            if(size<velk*0.25)
            {
                resizing( new_size: velk/2);
            }
            return;
        }
    }
}
```

- Resizing function that takes a new size as input and resizes the hash table to the new size.



```
public void resizing(int new_size)
{
    LinkedList<Key_and_Value<KEY,VALUE>>[] list = new LinkedList[new_size];
    for(int i =0; i<new_size; i++)
    {
        list[i] = new LinkedList<>();
    }
    for(LinkedList<Key_and_Value<KEY,VALUE>> newlist: llist)
    {
        for(Key_and_Value<KEY,VALUE> vstup: newlist)
        {
            int index =Math.abs(vstup.get_key().hashCode()%new_size);
            list[index].add(vstup);
        }
    }
    llist=list;
    velk=new_size;
}
```

## Testing

In my task, I do time testing of trees and hash tables, and then compare the trees with each other and the tables, respectively, in the same way.

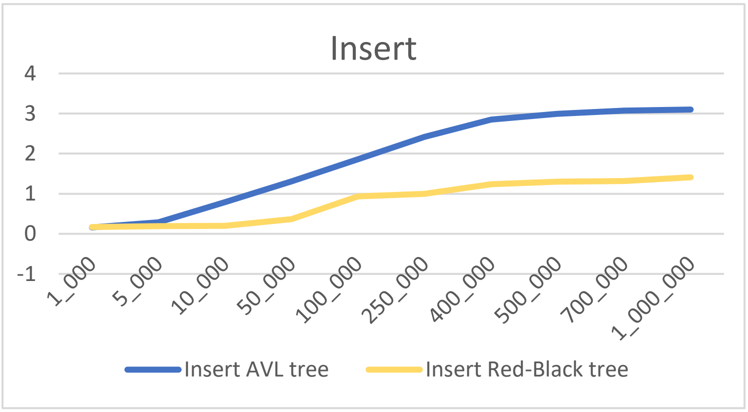
Each function(insert,delete,search) for each numbers of nodes I test 10 times to find average of time.

Testing works like this: I fill the tree ot tables with values, for example 1000, and then insert a single value into this tree ot table and measure the time milliseconds for a single node.

### Tree testing

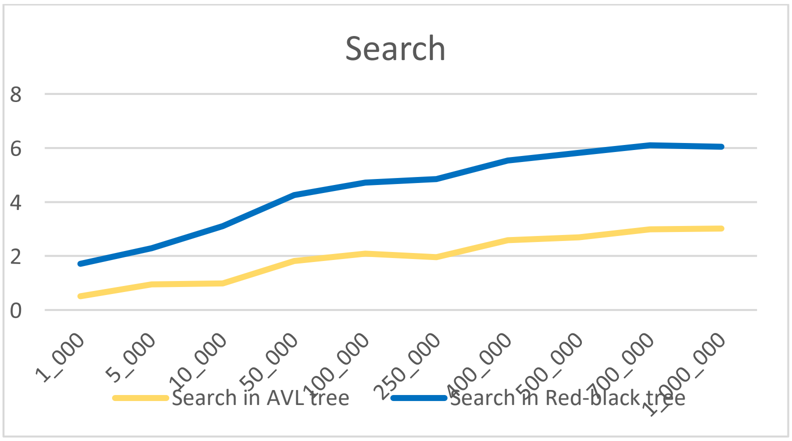
Time Complexity

| Tree | Insert | Search | Delete |
|---|---|---|---|
| AVL | O(log n) | O(log n) | O(log n) |
| Red-Black | O(log n) | O(log n) | O(log n) |

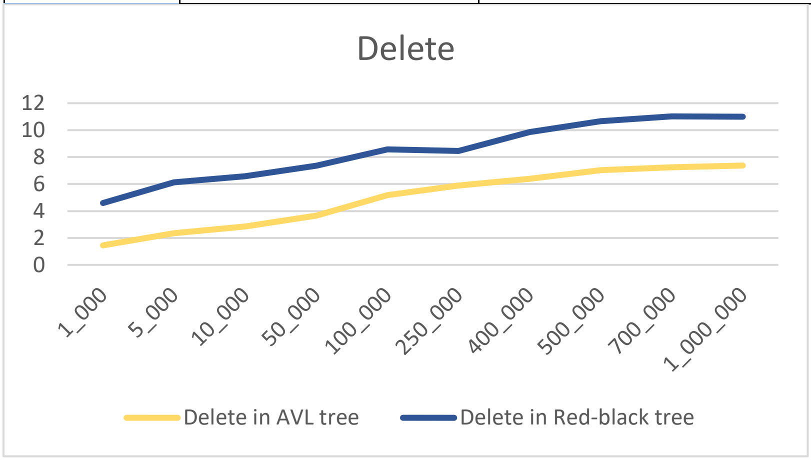


| Nodes | Insert AVL tree | Insert Red-Black tree |
|---|---|---|
| 1_000 | 0.16 | 0.17 |
| 5_000 | 0.29 | 0.19 |
| 10_000 | 0.79 | 0.2 |
| 50_000 | 1.31 | 0.37 |
| 100_000 | 1.86 | 0.93 |
| 250_000 | 2.42 | 1 |
| 400_000 | 2.85 | 1.24 |
| 500_000 | 2.99 | 1.3 |
| 700_000 | 3.07 | 1.32 |
| 1_000_000 | 3.1 | 1.41 |

Alexandra Vetrov ID:123102



| Nodes | Search in AVL tree | Search in Red-black tree |
|---|---|---|
| 1_000 | 0.51 | 1.2 |
| 5_000 | 0.95 | 1.34 |
| 10_000 | 0.99 | 2.12 |
| 50_000 | 1.81 | 2.45 |
| 100_000 | 2.09 | 2.632 |
| 250_000 | 1.958 | 2.89 |
| 400_000 | 2.59 | 2.95 |
| 500_000 | 2.689 | 3.13 |
| 700_000 | 2.9875 | 3.111 |
| 1_000_000 | 3.0154 | 3.024 |



| Nodes | Delete in AVL tree | Delete in Red-black tree |
|---|---|---|
| 1_000 | 1.45 | 4.59 |
| 5_000 | 2.34 | 6.12 |
| 10_000 | 2.84 | 6.58 |
| 50_000 | 3.65 | 7.365 |
| 100_000 | 5.18 | 8.56 |
| 250_000 | 5.9 | 8.459 |
| 400_000 | 6.398 | 9.85 |
| 500_000 | 7.036 | 10.67 |
| 700_000 | 7.236 | 11.02 |
| 1_000_000 | 7.37 | 10.987 |

We see that both implementations work correctly, because the graphs are logarithmic.

- Balance Factor: AVL tree maintains the balance factor for each node, which is the difference between the heights of its left and right subtrees. In contrast, Red-Black tree uses a color scheme to balance the tree, where each node is either red or black.
- Insertion and Deletion: AVL tree requires more rotations during insertion and deletion operations to maintain balance, whereas Red-Black tree must in addition to the rotations check the color and then with help of rotations return the correct one .
- Searching: Both AVL tree and Red-Black tree have the same searching complexity of O(log n).

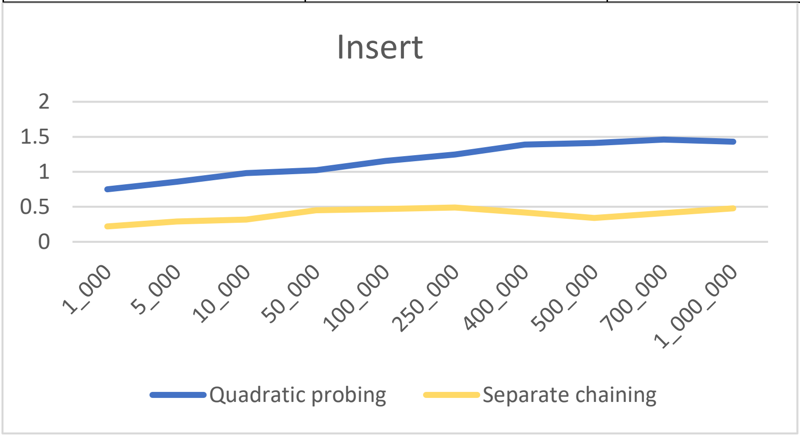
In summary, AVL tree is more suited for scenarios where searching operations are more frequent and more important, while Red-Black tree is more suited for scenarios where insertion and deletion operations are more frequent and more important.

## Tables testing

For testing the tables, I use the following data set,  the key represents the generated id to which the value corresponds, i.e the person`s name .

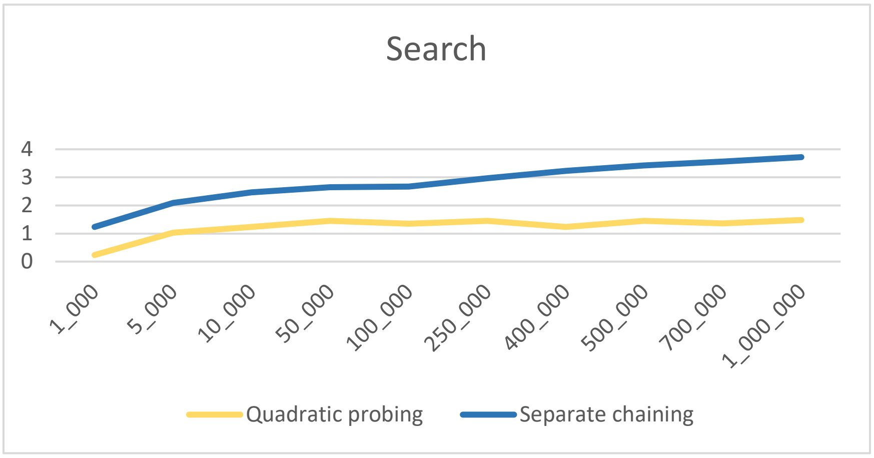
Time complexity

| Methods | Insert | Search | Delete |
|---|---|---|---|
| Quadratic probing | O(1) | O(1) | O(1) |
| Separate chaining | O(1) | O(1) | O(1) |

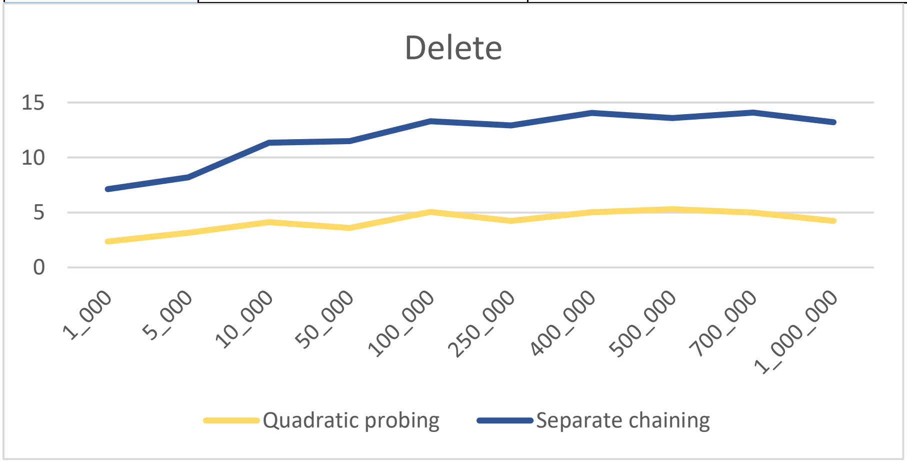


| Nodes | Quadratic probing | Separate chaining |
|---|---|---|
| 1_000 | 0.75 | 0.22 |
| 5_000 | 0.86 | 0.29 |
| 10_000 | 0.98 | 0.32 |
| 50_000 | 1.024 | 0.45 |
| 100_000 | 1.155 | 0.47 |
| 250_000 | 1.248 | 0.49 |
| 400_000 | 1.39 | 0.42 |
| 500_000 | 1.41 | 0.34 |
| 700_000 | 1.46 | 0.41 |
| 1_000_000 | 1.43 | 0.48 |

Alexandra Vetrov ID:123102



| Nodes | Quadratic probing | Separate chaining |
|---|---|---|
| 1_000 | 0.234 | 1 |
| 5_000 | 1.023 | 1.07 |
| 10_000 | 1.234 | 1.23 |
| 50_000 | 1.45 | 1.2 |
| 100_000 | 1.3459 | 1.324 |
| 250_000 | 1.45 | 1.52 |
| 400_000 | 1.234 | 2 |
| 500_000 | 1.45 | 1.98 |
| 700_000 | 1.36 | 2.2 |
| 1_000_000 | 1.48 | 2.24 |



| Nodes del | Quadratic probing | Separate chaining |
|---|---|---|
| 1_000 | 2.36 | 4.76 |
| 5_000 | 3.17 | 5.023 |
| 10_000 | 4.11 | 7.23 |
| 50_000 | 3.59 | 7.89 |
| 100_000 | 5.04 | 8.26 |
| 250_000 | 4.234 | 8.69 |
| 400_000 | 5.023 | 9.023 |

| | | |
|---|---|---|
| 500_000 | 5.3 | 8.2978 |
| 700_000 | 4.98 | 9.098 |
| 1_000_000 | 4.23 | 8.9856 |

We see that both implementations work correctly, because the graphs are O(1).

Collision Resolution: Quadratic Probing is a method of open addressing, where collisions are resolved by searching for the next available slot in the table using a quadratic function. Separate Chaining is a method of closed addressing, where collisions are resolved by creating a linked list of all the keys that hash to the same value.

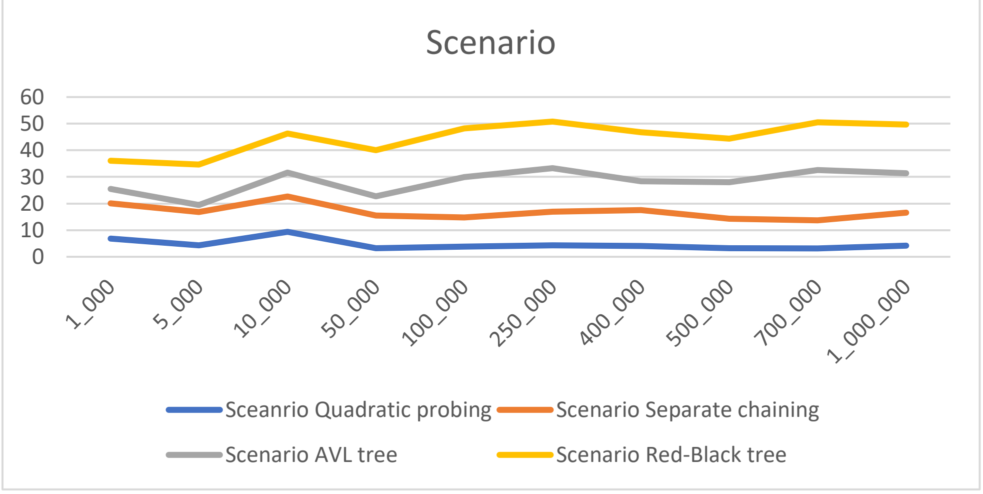
Space Efficiency: Separate Chaining can be more space-efficient than Quadratic Probing since it does not require any extra space for probing. The size of the hash table can be smaller than in Quadratic Probing. On the other hand, Quadratic Probing can be more space-efficient if the load factor is high and there are many collisions in Separate Chaining, as it does not require additional pointers to store the linked lists.

Time Complexity: In the average case, both Quadratic Probing and Separate Chaining have a time complexity of O(1) for insert, delete, and search operations.

In summary, Quadratic Probing is best suited for small hash tables with a low load factor, while Separate Chaining is more suitable for larger hash tables with a high load factor or where collisions are likely to occur.

## Scenario (Insert->Search->Delete)

| Nodes | Sceanrio Quadratic probing | Scenario Separate chaining | Scenario AVL tree | Scenario Red-Black tree |
|---|---|---|---|---|
| 1_000 | 6.83 | 13.21 | 5.43 | 10.6 |
| 5_000 | 4.36 | 12.5 | 2.59 | 15.2 |
| 10_000 | 9.38 | 13.27 | 8.99 | 14.65 |
| 50_000 | 3.27 | 12.2 | 7.2 | 17.37 |
| 100_000 | 3.83 | 10.92 | 15.21 | 18.23 |
| 250_000 | 4.36 | 12.64 | 16.29 | 17.5 |
| 400_000 | 4.1 | 13.5 | 10.79 | 18.42 |
| 500_000 | 3.18 | 11.1 | 13.75 | 16.35 |
| 700_000 | 3.14 | 10.55 | 18.91 | 17.9 |
| 1_000_000 | 4.236 | 12.3 | 14.81 | 18.36 |



## Conclusion

In summary, AVL tree and Red-Black tree are self-balancing binary search trees that maintain balance by performing rotations. AVL trees are more suited for scenarios where searching operations are more important, while Red-Black trees are more suited for scenarios where insertion and deletion operations are more frequent.  Hash Tables with Separate Chaining are more suited for scenarios where the load factor is high or when collisions are likely to occur, while Hash Tables with Quadratic Probing are more suited for scenarios where the load factor is low or when the number of elements to be stored is small.

In conclusion, the choice of data structure to use depends on the specific requirements of the problem you are solving. AVL trees and Red-Black trees are useful for maintaining a sorted set of data, while Hash Tables are useful for fast searching, insertion, and deletion operations. Hash Tables with Separate Chaining are more suited for scenarios where collisions are likely to occur, while Hash Tables with Quadratic Probing are more suited for scenarios where the load factor is low or when the number of elements to be stored is small.

Sources which I use for writing theory about these implementations:
https://www.geeksforgeeks.org/quadratic-probing-in-hashing/
https://www.topcoder.com/thrive/articles/hash-table-probing-technique-quadratic-probing#:~:text=Time%20Complexity%3A%20O(N%20*,size%20of%20the%20hash%20table.
https://www.geeksforgeeks.org/introduction-to-red-black-tree/
https://www.scaler.com/topics/data-structures/separate-chaining/
https://www.geeksforgeeks.org/what-is-avl-tree-avl-tree-meaning/?ref=rp