**Introduction**: The document comprizes of a overall software modeling design, which serves to specify the overall functionality of the trip planning application “Visit Faroe Islands”  
**Methods**: The document comprizes of a collection UML models, such as Use cases, sequence diagrams, state diagrams and more.  
**Results**: Now i know how to apply software modeling techniques, and soon, i will rule the world.  
**Conclusion**: Give me more grant money.

# 1) Introduction

Data structures and algorithms play a fundamental role in computer science and are essential for solving complex problems efficiently. In this assignment, we will be exploring the topic of search in dynamic sets, which involves various algorithms designed to search for elements in dynamic sets.

The main objective of this assignment is to implement and compare four different data structures in terms of their effectiveness in insert, delete, and search operations in different situations. The four data structures include two implementations of binary search trees (BVS) with different balancing algorithms, and two implementations of hash tables with different collision resolution methods.

The assignment requires the implementation of these data structures as separate source files, and the documentation of the implementations, testing scenarios, and the achieved results in a technical documentation file. It is also mandatory to submit a test program that measures the effectiveness of the implemented data structures. **The quality of testing and the processing of results in the documentation will mainly be evaluated**.

The technical documentation should contain detailed testing scenarios that identify the most suitable situations for each of the implementations. It should also include tables and graphs that compare the performance (speed) of individual solutions. As the results depend on the implementation of the solution and the test scenarios, it is important to document various different test scenarios to ensure accurate results.

In summary, this assignment offers an opportunity to explore and implement various data structures and algorithms in the context of searching in dynamic sets. Through this assignment, students will gain valuable experience in implementing data structures and analyzing their performance.

based on provided literature and information from the professor, we have managed to see basic AVL tree implementations in 3 languages (C++, Python and Java)(AlgotithmTutor [2019](#ref-AlgotithmTutor-2019)). by reading the literature and discussions about binary search trees, and self balancing search trees, we can say that there are 7 types of binary search trees. 1. AA Tree 2. AVL Tree 3. 2-3 Tree 4. Red-black tree 5. Scapegoat tree 6. Splay tree 7. Treap

we will in this assignment have a closer look at the two **self-balancing binary search trees** and compare the *AVL tree* with the *Red-black tree*.

# 1) Self balancing binary search trees comparison

## 1.1) The AVL tree

The binary search tree, known as the AVL tree is a **self-balancing binary search tree**, which uses the tree *height* as its balancing factor. This means that the height of the AVL tree is in the order of .

In Java, we represent the tree node as the following code block.

// data structure that represents a node in the tree  
class Node {  
 int data; // holds the key  
 Node parent; // pointer to the parent  
 Node left; // pointer to left child  
 Node right; // pointer to right child  
 int bf; // balance factor of the node  
  
 public Node(int data) {  
 this.data = data;  
 this.parent = null;  
 this.left = null;  
 this.right = null;  
 this.bf = 0;  
 }  
}

As indicatedby this code block, the node comprizes of five members and one method. when building the tree structure, we are utilizing this node in a recursive maner, which results in each node object are identical. And each node can therefore been seen as a small tree, the root, left and right child, and each child node can therefore be refered to as a sub tree.

*The height of a (sub) tree indicates how far the root is from the lowest node. Therefore, a (sub) tree that consists of only a root node has a height of 0.*

The AVL tree ensures height balance through a specific property, which dictates that the difference between the heights of the left and right subtrees of each node should not exceed one. This balance property is checked and maintained after every insert or delete operation through AVL rotate, which restores balance if needed.

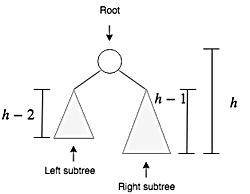
*The heights of the left and right subtrees differ by at most 1. If hl be the height of the left subtree and hr be the height of the right subtree, then,*

This implies that for each time we insert or delete a node in the tree, we need to update the balancing factor, and we do that as indicated in the following code block.

// update the balance factor the node  
 private void updateBalance(Node node) {  
 if (node.bf < -1 || node.bf > 1) {  
 rebalance(node);  
 return;  
 }  
  
 if (node.parent != null) {  
 if (node == node.parent.left) {  
 node.parent.bf -= 1;  
 }   
  
 if (node == node.parent.right) {  
 node.parent.bf += 1;  
 }  
  
 if (node.parent.bf != 0) {  
 updateBalance(node.parent);  
 }  
 }  
 }

For a tree to qualify as an AVL tree, each of its nodes must adhere to the specified balance property. If this condition is not met by any of the nodes, the tree must be restructured to ensure that the property holds. The following proof aims to demonstrate that the AVL property guarantees that the height of the tree will be proportional to the logarithm of the number of nodes in the tree .

To illustrate this, let us consider an AVL tree as shown in Figure 1. Let h represent the height of the tree, and Nh denote the number of nodes present in the tree at height h.



An AVL tree of height h

The sum of the number of nodes in the left and right subtrees, along with the root node, equals the total number of nodes in the tree.

The recurrence relation is homogeneous and similar to the one for the Fibonacci numbers. The solution to this recurrence relation is…

$$N\_h = {\varphi^h \over \\sqrt5}$$

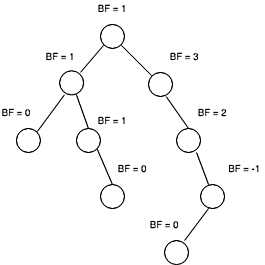
Where is the golden ratio. If we take the on both sides, we get

This proves that the height of the AVL tree is in the order of .

## 1.1.1) The Balancing Factor

The balancing factor of a node, denotes tha there is a difference of heights between the left and right subtree of the node.

in an AVL tree, the balancing factor must be , or . if the balancing factor of a node is greater than (right heavy) or less than -1 (left heavy), the node needs to be rebalanced. Figure 2 shows a tree with a balance factor



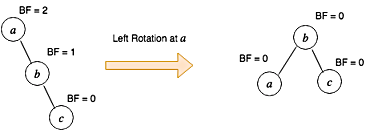
A binary tree with balance factor

## 1.2) AVL tree rotations

If a node’s balance factor exceeds the values of -1 or 1, we execute tree rotations on that particular node to achieve balance within the tree. Such rotations modify the tree’s arrangement and ensure its equilibrium. The AVL tree employs four different types of rotations, which are explained below.

### 1.2.1) Left Rotation

Figure 3 illustrates the left rotation on AVL tree.



Left Rotation

The tree at the left side of Figure 3 is right heavy. To fix this, we must perform a left rotation on node a . This is done in the following steps.

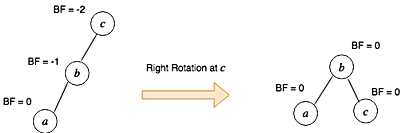
* becomes the new root
* takes ownership of ’s child as its right child, or in this case, null
* takes ownership of as its left child.

In AVL tree, we perform the left rotation on node x when

* Node x is right heavy

### 1.2.1) Left Rotation

Figure 4 illustrates the right rotation.



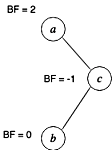
Right Rotation (RR)

We fix the tree on the left side of Figure 4 using following steps.

* becomes the new root.
* takes ownership of ’s right child, as its left child. In this case, that value is null.
* takes ownership of , as its right child. In AVL tree, we perform the right rotation on node when
* Node is left heavy
* Node ’s left subtree is not right heavy

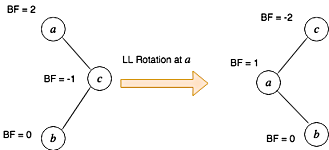
### 1.2.1) Left Rotation

Sometimes a single rotation is not sufficient to balance an unbalanced tree. Consider a tree given in Figure 5.



An unbalanced tree. Node a is right heavy with BF 2

Since node is right heavy, first thing that comes in mind is to perform the left rotation . Let’s see what happens if we do the left rotation on node a.

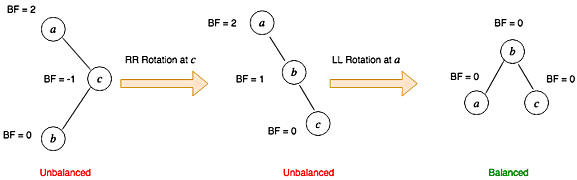


Performing left rotation on the root node

As we can see in the figure, the tree after the left rotation is still *un-balanced*. Therefore the single left operation is not effective in this case. To fix this we do the following using two step rotation.

* Perform the *right rotation* on the right subtree. In the above figure, perform the right rotation on node (NOT ).
* Perform the left rotation on the root node.

This is illustrated in Figure 7 below.



Illustrating the left-right rotation

We perform the left right rotation on node when

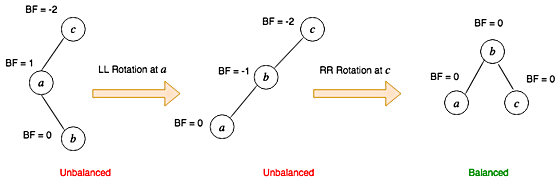
* Node is right heavy
* Node ’s right subtree is left heavy

### 1.2.2) Left Rotation

rotation is symmetric to rotation. In rotation, we do the following

* Perform the *left* rotation on the left subtree.
* Perform the *right* rotation on the root node.

This is illustrated in Figure 8.



Illustrating the right-left rotation

We perform the right left rotation on node when

* Node is left heavy
* Node ’s left subtree is right heavy

## 1.3) Operations on AVL tree

The source code used in this implementation of the AVL tree originates from the solution develloped by (Bibeknam [2019](#ref-Bibeknam-2019)). The source code is however slightly modified to fit our needs.

## 1.2) Red-Black tree

# References:

AlgotithmTutor. 2019. “AVL Trees with Implementation in C++, Java, and Python.” <https://algorithmtutor.com/Data-Structures/Tree/AVL-Trees/>.

Bibeknam. 2019. “AVLTree Source Code.” <https://github.com/Bibeknam/algorithmtutorprograms/blob/master/data-structures/avl-trees/AVLTree.java>.