4 Balanced trees, AVL trees
Balanced trees

- A class of binary search trees is balanced if each of the three dictionary operations
  - search
  - insert
  - delete
  of keys for a tree with $n$ keys be carried out in $O(\log n)$ steps

- Possible balancing conditions:
  - height condition $\Rightarrow$ AVL-Trees
  - weight condition $\Rightarrow$ BB[$\alpha$] -Trees
  - structural conditions $\Rightarrow$ B-Tree
AVL trees

Developed by Adelson-Velskii and Landis (1962)

- **Idea of AVL trees**: modified procedures for insertion and deletion, which prevents the tree from degenerating

- **Goal of AVL trees**: height is in $O(\log n)$ and search, insertion and deletion can be carried out in logarithmic time
Definition: A binary search tree is called AVL tree or height-balanced tree, if for each node $n$ the height of the right subtree $h(T_r)$ of $n$ and the height of the left subtree $h(T_l)$ of $n$ differ by at most 1.

Balance factor:

$$\text{bal}(n) = h(T_r) - h(T_l) \in \{-1, 0, +1\}$$
Examples

AVL tree

not an AVL tree

AVL tree
Properties of AVL trees

- AVL trees cannot degenerate into linear lists
- AVL trees with $n$ nodes have a height in $O(\log n)$

Apparently:
- an AVL tree of height 0 has 1 leaf
- an AVL tree of height 1 has 2 leaves
- an AVL tree of height 2 with a minimal number of leaves has 3 leaves
- ...
- what is the minimal number of leaves in a AVL tree of height $h$?
Hence: an AVL tree of height $h$ has at least $F_{h+2}$ leaves, where

\begin{align*}
F_0 &= 0 \\
F_1 &= 1 \\
F_{i+2} &= F_{i+1} + F_i
\end{align*}

$F_i$ is the $i$-th Fibonacci number.
Minimal AVL tree of height 10
Theorem: The height $h$ of an AVL tree with $n$ leaves is at most $c \cdot \log_2 n$, i.e.

$$h \in O(\log n)$$

Proof: First show by induction that for $h \geq 6$, $F_h \geq 2^\frac{h}{2}$. Then, use the fact that $n \geq F_{h+2}$ to show $h \in O(\log n)$.

Base cases:

$$F_6 = 8 \geq 2^2$$
$$F_7 = 13 \geq 2^2 \approx 11.314$$

Inductive step:

$$F_i = F_{i-1} + F_{i-2}$$

$$F_{i-1} + F_{i-2} \geq 2 \frac{i-1}{2} + 2 \frac{i-2}{2}$$
$$F_{i-1} + F_{i-2} \geq 2^2 \cdot 2 \frac{i-2}{2} + 2 \frac{i-2}{2}$$

$$F_{i-1} + F_{i-2} \geq 2 \frac{i-2}{2} \cdot (2^2 + 1)$$
$$2 \frac{i-2}{2} \cdot 2.4142 \geq 2 \frac{i}{2} \cdot 2 = 2^2$$
Height of an AVL tree (2)
Height of an AVL tree (3)

Theorem: The height $h$ of an AVL tree with $n$ leaves is at most $c \cdot \log_2 n$, i.e.

$$h \in O(\log n)$$

Proof:

For an AVL tree with $n$ leaves we have $n \geq F_{h+2} \geq 2^{\frac{h+2}{2}}$.

$$n \geq 2 \cdot 2^{\frac{h}{2}}$$

$$\log_2 n \geq \log_2 \left(2 \cdot 2^{\frac{h}{2}}\right)$$

$$\log_2 n \geq \log_2 2 + \log_2 2^{\frac{h}{2}}$$

$$\log_2 n - 1 \geq \frac{h}{2}$$

$$h \leq 2(\log_2 n - 1)$$

$$h \in O(\log n)$$

∎
For each modification of the tree we have to guarantee that the AVL property is maintained.

Insertion in an AVL tree

Original situation: After inserting key 5:

Problem: How can we modify the new tree such that it will be an AVL tree?
Storing the balance factors in the node

- According to the definition
- In order to restore the AVL property it is sufficient to store, in each node, the balance factor.

\[ \text{bal}(p) = h(p.\text{right}) - h(p.\text{left}) \in \{-1, 0, +1\} \]

- Example:
Different situation of insertions in an AVL tree

1. The tree is empty: create a single node with two leaves, store $x$ in it. Done!

   ![Tree Diagram]

2. The tree is not empty and the search ends in a leaf. Let node $p$ be the parent of the leaf where the search ended.

   Since $\text{bal}(p) \in \{-1,0,1\}$, we know that either
   
   - the left child of $p$ is a leaf, but not the right one (case 1) or
   - the right child of $p$ is a leaf, but not the left one (case 2) or
   - both children of $p$ are leaves (case 3).
Example of an AVL tree
Overall height unchanged (1)

- **Case 1:** $[bal(p) = +1]$ and $x < p.key$, since the search ends at a leaf with parent $p$. 

![Diagram showing tree with balance factor +1 being adjusted to balance the tree.](image-url)
Overall height unchanged (2)

- **Case 2:** $\text{bal}(p) = -1$ and $x > p.key$, since the search ends at a leaf with parent $p$.

Both cases are uncritical:

The height of the subtree containing $p$ does not change.
The critical case

Case 3: \([bal(p) = 0]\) Then both children of \(p\) are leaves. The height increases!

We distinguish the cases whether the new key \(x\) must be inserted as the right or left child of \(p\):

- \([bal(p) = 0 \text{ and } x > p.key]\)
- \([bal(p) = 0 \text{ and } x < p.key]\)

In both cases we need a procedure \(upin(p)\) which traces back the search path, checks the balance factors and carries out restructuring operations (so-called rotations or double rotations).
The procedure \texttt{upin}(p)

- When \texttt{upin}(p) is called, we always have \( bal(p) \in \{-1, +1\} \) and the height of the subtree rooted in \( p \) has increased by 1.

- \texttt{upin}(p) starts at \( p \) and goes upwards stepwise (until the root if necessary).

- In each step it tries to restore the AVL property.

- In the following we concentrate on the situation where \( p \) is the left child of its parent \( \varphi p \).

- The situation where \( p \) is the right child of its parent \( \varphi p \) is handled similarly.
Case 1: \( bal(\varphi p) = 1 \)

1. The parent \( \varphi p \) has balance factor +1. Since the height of the subtree rooted in \( p \) (the left child of \( \varphi p \)) has increased by 1, it is sufficient to set the balance factor of \( \varphi p \) to 0:

\[
\begin{align*}
\text{Case 1: } bal(\varphi p) &= 1 \\
1. \text{ The parent } \varphi p \text{ has balance factor +1. Since the height of the subtree rooted in } p \text{ (the left child of } \varphi p) \text{ has increased by 1, it is sufficient to set the balance factor of } \varphi p \text{ to 0:}
\end{align*}
\]
Case 2: $bal(\varphi p) = 0$

2. The parent $\varphi p$ has balance factor 0. Since the height of the subtree rooted in $p$ (the left child of $\varphi p$) has increased by 1, the balance factor of $\varphi p$ changes to -1. Since the height of the subtree rooted in $\varphi p$ has also changed, we must call $upin$ recursively with $\varphi p$ as the argument.
The critical case 3: \( \text{bal}(\varphi p) = -1 \)

- If \( \text{bal}(\varphi p) = -1 \) and the height of the left subtree of \( \varphi p \) (rooted in \( p \)) has increased by 1, the AVL property is now violated in \( \varphi p \).

- In this case we have to **restructure the tree**.

- Again we distinguish two cases: \( \text{bal}(p) = -1 \) (case 3.1) and \( \text{bal}(p) = +1 \) (case 3.2).

- The invariant for the call of \( \text{upin}(p) \) is \( \text{bal}(p) \neq 0 \). The case \( \text{bal}(p) = 0 \) can therefore not occur!
Case 3.1: $\text{bal}(\varphi p) = -1, \text{bal}(p) = -1$

right rotation

\[ \varphi p \quad \text{y} \quad -1 \]
\[ x \quad -1 \]
\[ 3 \quad h - 1 \]
\[ 2 \quad h - 1 \]
\[ 1 \quad h \]

\[ \varphi p \quad x \quad 0 \]
\[ y \quad 0 \]
\[ 3 \quad h - 1 \]
\[ 2 \quad h - 1 \]
\[ 1 \quad h \]

right rotation done!
Is the resulting tree still a search tree?

We must guarantee that the resulting tree fulfils the

1. search tree condition and the

2. AVL property.

Search tree condition: Since the original tree was a search tree, we know that

   all keys in tree 1 are smaller than \( x \).

   all keys in tree 2 are greater than \( x \) and smaller then \( y \).

   all keys in tree 3 are greater than \( y \) (and \( x \)).

Hence, the resulting tree also fulfils the search tree condition.
Is the resulting tree balanced?

**AVL property:** Since the original tree was an AVL tree, we know:

- since \( \text{bal}(\varphi p) = -1 \), tree 2 and tree 3 have the same height \( h-1 \).
- since \( \text{bal}(p) = -1 \) after the insertion, tree 1 has height \( h \), while tree 2 has height \( h-1 \).

Hence, after the rotation:

- the node containing \( y \) has balance factor 0.
- node \( \varphi p \) has balance factor 0.

Thus, the AVL property has been restored.
Case 3.2: \( bal(\varphi p) = -1, \ bal(p) = +1 \)
Case 3.2: $bal(\varphi p) = -1, bal(p) = +1$

double-rotation left-right
Properties of the subtrees

1. The new key must have been inserted into the right subtree of \( p \).

2. Trees 2 and 3 must have different height, since otherwise the method \( upin \) would not have been called.

3. The only possible combination of heights in trees 2 and 3 is therefore \((h-1, h-2)\) and \((h-2, h-1)\), unless they are empty.

4. Since \( bal(p) = 1 \), tree 1 must have height \( h-1 \)

5. Finally, tree 4 also must have height \( h-1 \) (because \( bal(\phi p) = -1 \)).

Hence, the resulting tree also fulfils the AVL property
Search tree condition

We have:

1. All keys in tree 1 are smaller than $x$.

2. All keys in tree 2 are smaller than $y$ but greater than $x$.

3. All keys in tree 3 are greater than $y$ and $x$ but smaller than $z$.

4. All keys in tree 4 are greater than $x$, $y$ and $z$.

Hence, the tree resulting from the double rotation is also a search tree.
Remarks

- We have only considered the case where \( p \) is the left child of its parent \( \phi p \).

- The case where \( p \) is the right child of its parent \( \phi p \) is handled similarly.

- For an efficient implementation of the method \( upin(p) \), we have to create a list of all visited nodes during the search for the insert position.

- Then we can use this list during the recursive calls to proceed to the parent and carry out the necessary rotations or double rotations.
Insertion in a non-empty AVL tree

Search for \( x \) ends in a leaf with parent \( p \)

1. Right child of \( p \) not a leaf, \( x < p.key \) \( \rightarrow \) Append as left child of \( p \), done.

2. Left child of \( p \) not a leaf, \( x > p.key \) \( \rightarrow \) append as right child of \( p \), done.

3. Both children of \( p \) are leaves: append \( x \) as child of \( p \) and call \( upin(p) \).

The method \( upin(p) \):

1. \( p \) is left child of \( \varphi p \)
   (a) \( bal(\varphi p) = +1 \) \( \rightarrow \) \( bal(\varphi p) = 0 \), done.
   (b) \( bal(\varphi p) = 0 \) \( \rightarrow \) \( bal(\varphi p) = -1 \), \( upin(\varphi p) \)
   (c) i. \( bal(\varphi p) = -1 \) und \( bal(p) = -1 \) right rotation, done.
      ii. \( bal(\varphi p) = -1 \) und \( bal(p) = +1 \) double rotation left-right, done.

2. \( p \) is righter child of \( \varphi p \).

...
An example (1)

Original situation:
An example (2)

Insert key 9:

AVL property is violated!
An example (3)

Left rotation at \( p \) yields:

![Tree Diagram](image)
An example (4)

Insertion of 8 followed by double rotation (left-right):

\[
\begin{array}{c}
\text{10} \\
\phi p \\
\text{7} \\
\text{3} \\
\text{8} \\
\end{array}
\quad \rightarrow 
\begin{array}{c}
\text{9} \\
\text{0} \\
\text{3} \\
\text{8} \\
\text{15} \\
\end{array}
\]

left-right