Search Trees:
AVL-Trees, Multi-way Search Trees, B-Trees

AVL Trees
AVL = Adelson-Vel’skii and Landis, 1962
The idea: Keep a balance information at each node

The balance $h(v)$ of a node $v$:
- $LeftHeight(v)$:
  - 0 if $v$ has no left child
  - $1 + Height(LC(v))$ otherwise
- $RightHeight(v)$:
  - 0 if $v$ has no right child
  - $1 + Height(RC(v))$ otherwise
- $h(v) = LeftHeight(v) - RightHeight(v)$

A BST $T$ is an AVL-tree if $h(v) \in [-1, 0, 1]$ for every node $v$

Worst AVL Tree
A worst AVL-tree of height $h$:
- as few nodes as possible to achieve height $h$

Construction:
Node balance always $-1$, subtrees always worst AVL trees

<table>
<thead>
<tr>
<th>$h$</th>
<th>$n_0$</th>
<th>$n_1$</th>
<th>$n_2$</th>
<th>$n_3$</th>
<th>$n_4$</th>
<th>$n_5$</th>
<th>$n_6$</th>
<th>$n_7$</th>
<th>$n_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>23</td>
<td>47</td>
<td>95</td>
<td>190</td>
<td>381</td>
</tr>
</tbody>
</table>

Fibonacci numbers $f_i = f_{i-1} + f_{i-2}$: 0, 1, 2, 3, 5, 8, 13, ...
We can prove: $n_h = f_{h+2} - 1$ (by induction)

AVL Tree Height Theorem: $h < 1.44 \log_{2} n$

Hence: Lookup in $\Omega(\log n)$ steps
(Example: insert 44, 17, 78, 32, 50, 88, 62)
AVL-Tree Insertion – Rebalance...

Insertion may destroy the balance.

Insertion with rebalancing:
1. search downwards the tree, mark the critical node: the lowest ancestor of the insertion node with balance ≠ 1 
   (before insertion)
2. insert
3. rotate at critical node if necessary

AVL-Tree Insertion... (Goodrich/Tamassia)

How to insert an item with key 50?
- Perform a LookUp(50)
- While searching for 50, keep track of last passed node with balance ≠ 0 - critical node
- If not found, insert at the leaf where search ended
- Recompute balance on the way back
- Check critical node!
  If balance ∉ [-1..1], rebalance!

AVL-Tree Insertion & Re-balance...

Now try an item with key 15...
- Perform a LookUp(15)
- While searching for 15, keep track of last passed node with balance ≠ 0 - critical node
- If not found, insert at the leaf where search ended
- Recompute balance on the way back
- Check critical node!
  If balance ∉ [-1..1], rebalance!

Time to re-balance...

- Label the critical node and its 2 descendants on the path to "15" as a, b, c, such that a < b < c, in an in-order traversal
- Re-structure the nodes a, b and c so that b has a and c as children!
Time to re-balance...

- Label the critical node and its 2 descendants on the path to "15" as \( a, b, c \), such that \( a < b < c \), in an in-order traversal.
- Re-structure the nodes \( a, b \) and \( c \) so that \( b \) has \( a \) and \( c \) as children.
- Update balance!

Removal of a node...

Same thing, but in reverse!

1. Perform a LookUp and Remove as in an ordinary binary tree.
2. Update the balance on the way back to the root.
3. If too unbalanced: Re-structure! ...but:
   - Label the critical node, the child on the deepest side, and its descendants on the deepest side as \( a, b, c \), such that \( a < b < c \), in an in-order traversal.
   - Re-structure as previous.
   - Go to #2 and continue update and check towards the root (we may have to re-balance more than once!)

Tri-node restructuring = rotations....

Other authors use left and right rotations:

- Single left rotation:
  - left part of the subtree (\( a \) and \( j \)) is lowered.
  - We have "rotated (up) \( b \) over \( a \)"...

Double rotations...

Two rotations are needed when the nodes to re-balance are placed in a zig-zag pattern...

1. Rotate up \( b \) over \( a \)
2. Rotate up \( b \) over \( c \)

Note: Labeling of \( a, b \) and \( c \) same as before!
New approach: relax some condition...

- AVL-Tree: strict binary tree, accepts a small unbalance...
- Recall:
  - Full binary tree: nonempty; degree is either 0 or 2 for each node
  - Perfect binary tree: full, all leaves have the same depth

- Can we build and maintain a perfect tree (if we skip "binary")??
  - we would always know the worst search time exactly!

(2,3) trees

Previously:
- A single "pivot element"
- If larger we search to the right
- If smaller we search to the left

Now:
- Multiple pivot elements
- No. of children = no. of pivot elements + 1

(a,b) trees

- Each node is either a leaf, or has \( c \) children where \( a \leq c \leq b \)
- Lookup works approximately as before
- Insert must check that a node does not overflow (then we split the node)
- Delete must check that a node does not become empty (then we transfer or merge nodes)

Insert in (a,b)-tree where \( a=2 \) and \( b=3 \)

- As long as there is room in the child we find, add element to that child...
- If full, split and push the selected pivot element upwards...
  ...may happen repeatedly
Delete in a (2,3)-tree

Three cases:
1. No constraints are violated by removal
2. A leaf is removed (becomes empty) → transfer some other key to that leaf, ...ok if we have a sibling with 2+ elements
3. An internal node becomes empty
   Root: replace with in-order pred. or succ.
   → repair inconsistencies with suitable merge and transfer operations...

Properties of a (2,3) tree

- Always a perfect tree
- A minimal tree of height \( h \) will have \( n = 2^{h+1} - 1 \) nodes (it’s a full binary tree with full set of nodes at all levels)
- A maximal (2,3)-tree will have a branching factor of 3, thus
  \[
  n = \sum_{i=0}^{h} 3^i = (3^{h+1} - 1) / 2
  \]
  ...2 keys in every node → \( k = 3^{h+1} - 1 \)
- Thus the height \( h = \lceil \log_3 k \rceil \)
B-Tree

- Used to keep an index over external data (e.g. content of a disc)
- It's only an (a,b)-tree where $a = \left\lfloor \frac{b}{2} \right\rfloor$
- We may now choose $b$ so that $b-1$ references to children (other disc blocks) fit into a single disc block
- By defining $a = \left\lfloor \frac{b}{2} \right\rfloor$ we will always fill up a disc block when two blocks are merged!