DDS-Balanced trees
Search balanced trees

- In search trees, giving find, insert, and delete methods, complexity is bounded by $O(h)$
- Key to usefulness of search trees is to keep balanced: height bounded by $O(\log n)$ instead of $O(n)$.
- The fundamental insight, first achieved height-balanced tree, called AVL tree.
- This height-balanced tree achieves a height bound:
  \[ h \leq 1.44 \log n + O(1). \]
- Because any tree with $n$ leaves has height at least $\log n$, this is already quite good.
- There are many other methods to consider here.
A tree is height-balanced if, in each interior node, height of right subtree and height of left subtree differ by at most 1.

A height-balanced tree has necessarily small height.

Theorem. A height-balanced tree of height $h$ has at least

$$\left(\frac{3+\sqrt{5}}{2\sqrt{5}}\right)^h \left(\frac{1+\sqrt{5}}{2}\right)^h - \left(\frac{3-\sqrt{5}}{2\sqrt{5}}\right)^h \left(\frac{1-\sqrt{5}}{2}\right)^h$$

leaves.

A height-balanced tree with $n$ leaves has height at most

$$\left\lceil \log_{1+\sqrt{5}/2 \cdot \sqrt{5}} n \right\rceil = \left\lceil c_{Fib} \log_2 n \right\rceil \approx 1.44 \log_2 n,$$

where $c_{Fib} = (\log_2(\frac{1+\sqrt{5}}{2}))^{-1}$. 


Height-balanced trees

Fibonacci Trees of Height 0 to 5

Height-Balanced Tree with Node Heights
**Height-balanced trees**

**Rebalancing a Node in a Height-Balanced Tree: Case 2.1**

Since we do only $O(1)$ work on each node of the path, at most two rotations and at most three recomputations of the height, and the path has length $O(\log n)$, these rebalancing operations take only $O(\log n)$ time. But we still have to show that they do restore the height balancedness.

We have to show this only for one step and then the claim follows for the entire tree by induction. Let $n_{old}$ denote a node before the rebalancing step, whose left and right subtrees are already height balanced but their height differs by 2, and let $n_{new}$ be the same node after the rebalancing step. By symmetry we can assume that $n_{old} \rightarrow \text{left} \rightarrow \text{height } h+2 = n_{old} \rightarrow \text{right} \rightarrow \text{height } h+2 + 2$. 

Rebalancing a Node in a Height-Balanced Tree: Case 2.1
Height-balanced trees

Rebalancing a Node in a Height-Balanced Tree: Case 2.2

Let $h = n_{\text{old}} - \text{right}$.

Because $n_{\text{old}} - \text{left} = h + 2$, we have $\max(n_{\text{old}} - \text{left} - \text{left}, n_{\text{old}} - \text{right}) = h + 1$, and because $n_{\text{old}} - \text{left}$ is height balanced, there are the following cases:

(a) $n_{\text{old}} - \text{left} - \text{left} = h + 1$ and $n_{\text{old}} - \text{right} \in \{h, h+1\}$.

By rule 2.1 we perform a right rotation around $n_{\text{old}}$.

By this $n_{\text{old}} - \text{left} - \text{left}$ becomes $n_{\text{new}} - \text{left}$, $n_{\text{old}} - \text{right}$ becomes $n_{\text{new}} - \text{right} - \text{left}$, and $n_{\text{old}} - \text{right}$ becomes $n_{\text{new}} - \text{right} - \text{right}$.

So $n_{\text{new}} - \text{left} = h + 1$, $n_{\text{new}} - \text{right} - \text{left} \in \{h, h+1\}$, $n_{\text{new}} - \text{right} - \text{right} = h$.

Thus, the node $n_{\text{new}} - \text{right}$ is height-balanced, with $n_{\text{new}} - \text{right} \in \{h+1, h+2\}$.

Therefore, the node $n_{\text{new}}$ is height-balanced.

(b) $n_{\text{old}} - \text{left} - \text{left} = h$ and $n_{\text{old}} - \text{left} - \text{left} = h + 1$. 
Theorem. The height-balanced tree structure supports find, insert, and delete in $O(\log n)$ time.

Further analysis of the rebalancing transformation shows that the rotations can occur during an insert only on at most one level. Delete might occur on every level if, for example, a leaf of minimum depth in a Fibonacci tree is deleted.

Even if there is only one level in rebalancing during an insert, there are many levels in which the nodes change: the height information must be updated.

The average depth of leaves in a Fibonacci tree with $n$ leaves is even better than $1.44 \log n$. 
A different method is to allow tree nodes of higher degree. This idea was introduced as B-trees by Bayer and McCreight (1972) and turned out to be very fruitful.

It was originally intended as external memory data structure, but it has interesting uses also as normal main memory data structure.

The characteristic of external memory is that access to it is very slow, compared to main memory, and is done in blocks, units much larger than single main memory locations.
In the 1970s, computers were still very memory limited but usually already had a large external memory, so that it was a necessary consideration how a structure operates when a large part of it is not in main memory, but on external memory. This situation is now less important, but it is still relevant for database applications, where B-tree variants are still much used as index structures.
The degree interval $a$ to $2a - 1$ is the smallest interval for which the rebalancing algorithm for B-trees works. Because each block has room for at most $2a - 1$ elements and is at least half full this way, it sounded like a good choice to optimize the space utilization.

But then it was discovered by Huddleston and Mehlhorn (1982) and independently by Maier and Salveter (1981) that choosing the interval a bit larger makes an important difference for the rebalancing algorithm.
if one allows node degrees from $a$ to $b$ for $b \geq 2a$, then rebalancing changes only amortized $O(1)$ blocks, whereas for $b = 2a - 1$, the original choice, $\Theta(\log n)$ block changes can be necessary.

For a main memory data structure, the number of changes in rebalancing makes little difference, for an external memory structure it is essential because all changed blocks have to be written again to the external memory device.

So these trees, known as $(a, b)$-trees, are the method of choice.
(a, b)- and B-Trees

A (4, 8)-Tree
Now we describe the insert and delete operations and the rebalancing that keeps the structure of the \((a, b)\)-tree.

Insert and delete begin straightforward as in the binary search-tree case: first one goes down in the tree to find the place where a new leaf should be inserted or an old one should be deleted. This is in a node of height 0.

If there is still room in the node for the new leaf or after the deletion the leaf still contains at least \(a\) objects, there is no problem.
But the node could overflow during an insertion or become underfull during a deletion. In these cases we have to change something in the structure of the tree and possibly propagate the structure upward. The restructuring rules for these situations are as follows:

A: If the current node is the root, create two new nodes, copy into each half the root entries, and put into the root just pointers to these two new nodes together with the key that separates them. Increase the height of the root by 1.
(a, b)- and B-Trees

- **B**: Else create a new node and move half the entries from the overflowing node to the new node. Then insert the pointer to the new node into the upper neighbor.

  The case b) is known as “splitting”.

- For a deletion: if the current node becomes underfull

- **A**: If the current node is the root, it is underfull if it has only one remaining pointer. Copy the content of the node to which the pointer points into the root node and return the node to the system.
B: Else find the block of the same height that immediately precedes or follows it in the key order and has the same upper neighbor. If that block is not already almost underfull, move a key and its associated pointer from that block and correct the key value separating these two blocks in the upper neighbor.

C: Else copy entries of the current node into that almost underfull neighboring node of the same height, return the current node to the system, and delete the reference to it from the upper neighbor.
The cases b) and c) are known as “sharing” and “joining,” respectively.

**Theorem.** The (a, b)-tree structure supports find, insert, and delete with $O(\log_a n)$ block read or write operations and needs only an amortized $O(1)$ block writes per insert or delete.

The (a, b)-tree structure allows for $b \geq 2a$, also a top-down rebalancing method, where all the rebalancing is done on the way from the root to the leaf and no pass back from the leaf to the root is necessary.
The idea is simple: for insertion, we split any node of degree $b$ we encounter along the path down. This splitting does not propagate up because the node above was already split before, so it still has room for an additional entry.

At the bottom level, we arrive with a node that still has room for the new leaf that we insert.

For deletion, we perform joining or sharing for each node on the path down that has degree $a$; again this does not propagate up because the node above already has degree at least $a + 1$. 
On the bottom level we arrive with a node that can spare the entry that we delete.

Thus we perform a preemptive splitting or joining; we still change only $O(\log_a n)$ nodes, but the amortized $O(1)$ bound no longer holds.

Also, we require $b \geq 2a$, so this method does not apply to classical B-trees (with $b = 2a - 1$). A potential useful aspect of the top-down method is that it requires only a lock on the current node and its neighbors, instead of the entire path to the root.
A different balance criterion for the same type of block nodes as (a, b) - trees was given:

their r-dense m-ary multiway trees also have all leaves at the same depth, but balancing is achieved by the property that any nonroot node that is not of maximum degree (which is m) has at least r nodes with the same upper neighbor that are of maximum degree.

This criterion is similar to the brother trees and inherits from there an inefficiency in the deletion algorithm (O ((log n)^m−1 ) for m-ary trees instead of O(log n)).
Red-Black Trees and Trees of Almost Optimal Height

- (2, 4)-trees called “symmetric binary B-trees” (SBB-trees). In these binary search trees, the edges are labeled as “downward” or “horizontal” with the restrictions:
  - the paths from the root to any leaf have the same number of downward edges, and
  - there are no two consecutive horizontal edges.
- This structure directly corresponds to (2, 4)-trees.
- Further reformulation label nodes instead of edges, making the top node of each small binary tree replacing a (2, 4)-node black and the other nodes red.
Red-Black Trees and Trees of Almost Optimal Height

- This is the red-black tree now used in many textbooks - a binary search tree with nodes colored red and black such that:
  - the paths from the root to any leaf have the same number of black nodes,
  - there are no two consecutive red nodes,
  - the root is black.
Red-Black Trees and Trees of Almost Optimal Height

We also assign colors to the leaves; this breaks the complete analogy to (2, 4)-trees but is convenient for the rebalancing algorithm.

Replacement of (2, 4)-Nodes to Red-Black-Labeled Binary Trees

We can collapse any red node in the black node above it and obtain a (2, 4)-tree apart from the nodes on leaf level. So a red-black tree has height at most \(2 \log n + 1\). And we have from the underlying (2, 4)-tree structure a rebalancing algorithm with \(O(\log n)\) worst-case complexity and that changes amortized only \(O(1)\) nodes. The only disadvantage with regard to our previous framework is that this rebalancing algorithm uses instead of rotations the more complex operations of split, share, and join. But there is also a rotation-based algorithm with the same properties that we will describe later.

Red-Black Tree with Node Colors

Other equivalent versions of the same structure are the half-balanced trees by Olivié (1982), characterized by the property that for each internal node, the longest path to a leaf is at most twice as long as the shortest path, whose equivalence to the red-black trees was noticed by Tarjan (1983a) and the standard son-trees by Ottmann and Six (1976) and Olivié (1980), which are trees with unary and binary nodes, whose all leaves are at the same depth, and there are no unary nodes on the even levels. Several alternative rebalancing algorithms for these structures have been proposed in Tarjan (1983a), Zivani, Olivié, and Gonnet (1985), Andersson (1993), Chen and Schott (1996).

Guibas and Sedgewick (1978) also observed that several other rebalancing schemes could be expressed as color labels on the vertices associated with certain rebalancing actions. For the height-balanced trees, it was already long known that one need not store the height in each node but just the information whether the two subtrees have equal height, or the left or the right height is...
Red-Black Trees and Trees of Almost Optimal Height

As in the case of height-balanced trees, not only this worst-case height bound is tight, but it is possible that almost all leaves are at that depth; such a red-black tree was constructed in Cameron and Wood (1992).

We will describe now the red-black tree with its standard bottom-up rebalancing method because it is classical textbook material, and in Section 3.5 an alternative top-down rebalancing method. Both work on exactly the same structure. The node of a red-black tree contains as rebalancing information just that color entry.

We have to maintain the following balancedness properties:

(1) each path from the root to a leaf contains the same number of black nodes,
(2) if a red node has lower neighbors, they are black.

It is also convenient to add the condition

- the root is black.
Theorem:
- A red-black tree of height \( h \) has at least \( 2^{(h/2)+1} - 1 \) leaves for \( h \) even and at least \( (3/2)2^{(h-1)/2} - 1 \) leaves for \( h \) odd.
- The maximum height of a red-black tree with \( n \) leaves is \( 2 \log n - O(1) \).

Rebalancing: rotations.
- This is description of red-black tree with its standard bottom-up rebalancing method because it is classical textbook material. The node of red-black tree contains as rebalancing information just that color entry.
Red-black trees allow a top-down rebalancing, as did weight-balanced trees and (a, b)-trees, which performs the rebalancing on the way down to the leaf, without the need to return to the root.

For insertion, go down from root to leaf and ensure by some transformations that the current black node has at most one red lower neighbor.

So each time we meet a black node with two red lower neighbors, we have to apply some rebalancing transformation; this corresponds to splitting of (2, 4)-nodes of degree 4.
Top-Down Rebalancing for Red-Black Trees: Inserting Items

Insert 36  insert in leaf  divide leaf and move middle value up to parent

(a) 30 39
    /        \
   /          /
10 20 36 37 38 40

(b) 50
    /  \  ...
30 37 39
      /  \
10 20
36 38 40

overcrowded node
Top-Down Rebalancing for Red-Black Trees: Inserting Items

... still inserting 36

divide overcrowded node,
move middle value up to parent,
attach children to smallest and largest

result

(c) 37 50
   30 39
  10 20 36 38 40

(d) 37 50
   30 39
  10 20 36 38 40 60 80 100
Top-Down Rebalancing for Red-Black Trees: Inserting Items

After Insertion of 35, 34, 33
Top-Down Rebalancing for Red-Black Trees

- Thus, at leaf level we always arrive at a black leaf, so we can insert a new leaf below that black node without any further rebalancing.
- For deletion, go down from root to leaf and ensure by some transformations that current black node has at least one red lower neighbor.
- So each time we meet a black node with two black lower neighbors, we have to apply some rebalancing transformation: this corresponds to joining or sharing of (2, 4)-nodes of degree 2.
- Thus we arrive at leaf level in a black node that has at least one red lower neighbor, so we can delete a leaf below that black node without any further rebalancing.
Top-Down Rebalancing for Red-Black Trees

2.1

2.2

2.3

Cases 2.1 to 2.3 of Top-Down Insertion:
upper and current Are Marked with current Moving Down

Next we give an implementation of insert in red-black trees with top-down rebalancing.

```c
int insert(tree_t* tree, int key, obj_t* obj)
{
    if (tree->left == NULL)
    {
        tree->left = (tree_node_t*)new obj;
        tree->key = new_key;
        tree->color = black;
        return;
    }
    else
    {
        tree_node_t* current, *next, *upper;
        current = tree;
        upper = NULL;
        while (current->right != NULL)
        {
            if (new_key < current->key)
                next_node = current->left;
            else
                next_node = current->right;
            if (current->color == black ||
                current->right->color == black)
```