16. Binary Search Trees

[Ottman/Widmayer, Kap. 5.1, Cormen et al, Kap. 12.1 - 12.3]

Dictionary implementation

Hashing: implementation of dictionaries with expected very fast access times.

Disadvantages of hashing:

Dictionary implementation

Hashing: implementation of dictionaries with expected very fast access times.

Disadvantages of hashing: linear access time in worst case. Some operations not supported at all:

Dictionary implementation

Hashing: implementation of dictionaries with expected very fast access times.

Disadvantages of hashing: linear access time in worst case. Some operations not supported at all:

- enumerate keys in increasing order

Dictionary implementation

Hashing: implementation of dictionaries with expected very fast access times.

Disadvantages of hashing: linear access time in worst case. Some operations not supported at all:

- enumerate keys in increasing order
- next smallest key to given key

Trees

Trees are

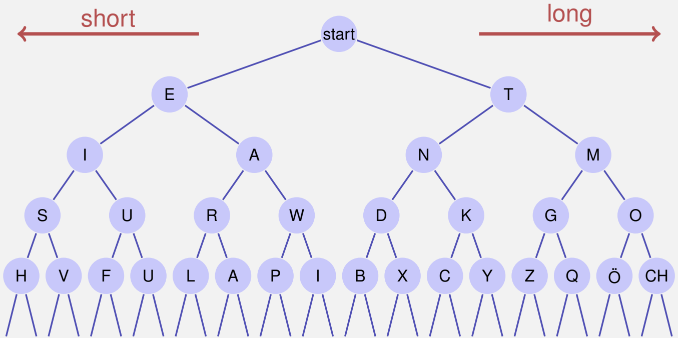
- Generalized lists: nodes can have more than one successor
- Special graphs: graphs consist of nodes and edges. A tree is a fully connected, directed, acyclic graph.

Trees

Use

- Decision trees: hierarchic representation of decision rules
- syntax trees: parsing and traversing of expressions, e.g. in a compiler
- Code tress: representation of a code, e.g. morse alphabet, huffman code
- Search trees: allow efficient searching for an element by value

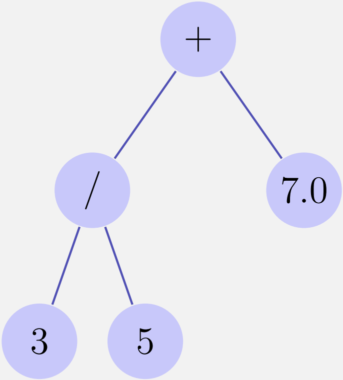
Examples



Morsealphabet

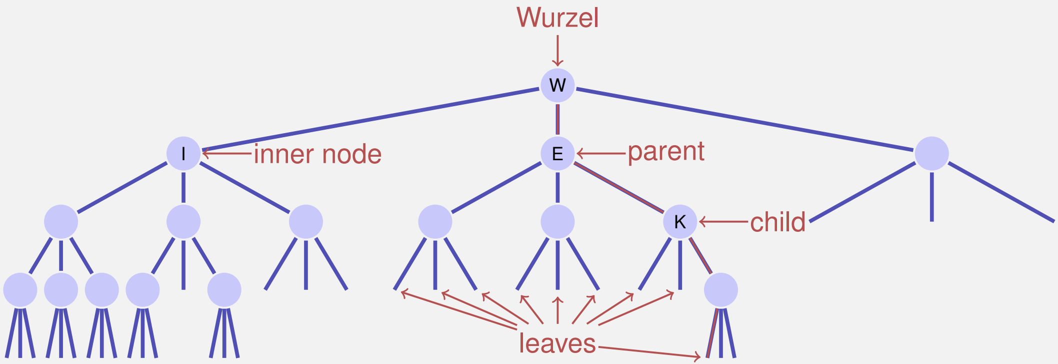
Examples

$$3/5 + 7.0$$



Expression tree

Nomenclature



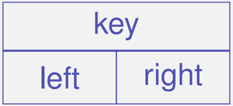
- Order of the tree: maximum number of child nodes, here: 3
- Height of the tree: maximum path length root – leaf (here: 4)

Binary Trees

A binary tree is either

- a leaf, i.e. an empty tree, or
- an inner leaf with two trees T_l (left subtree) and T_r (right subtree) as left and right successor.

In each node v we store

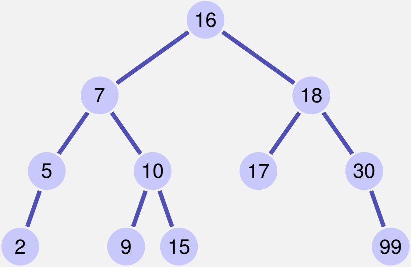


- a key $v.\mathrm{key}$ and
- two nodes $v.\mathrm{left}$ and $v.\mathrm{right}$ to the roots of the left and right subtree.
- a leaf is represented by the **null**-pointer

Binary search tree

A binary search tree is a binary tree that fulfils the search tree property:

- Every node v stores a key
- Keys in the left subtree $v.\mathrm{left}$ of v are smaller than $v.\mathrm{key}$
- Key in the right subtree $v.\mathrm{right}$ of v are larger than $v.\mathrm{key}$

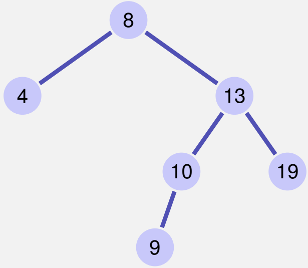


Searching

Input : Binary search tree with root r, key k
Output : Node v with $v.\mathrm{key} = k$ or **null**

$v \leftarrow r$
while $v \neq$ **null do**
  if $k = v.\mathrm{key}$ **then**
    return v
  else if $k < v.\mathrm{key}$ **then**
    $v \leftarrow v.\mathrm{left}$
  else
    $v \leftarrow v.\mathrm{right}$

return null

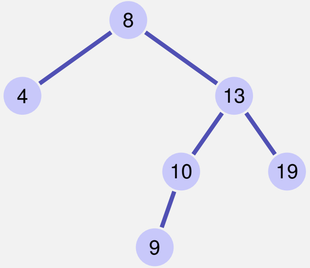


Searching

Input : Binary search tree with root r, key k
Output : Node v with $v.\text{key} = k$ or **null**

$v \leftarrow r$
while $v \neq$ **null do**
    if $k = v.\text{key}$ **then**
        return v
    else if $k < v.\text{key}$ **then**
        $v \leftarrow v.\text{left}$
    else
        $v \leftarrow v.\text{right}$

return null



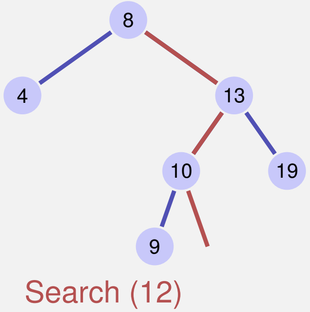
Search (12)

Searching

Input : Binary search tree with root r, key k
Output : Node v with $v.\mathrm{key} = k$ or **null**

$v \leftarrow r$
while $v \neq$ **null do**
    if $k = v.\mathrm{key}$ **then**
        return v
    else if $k < v.\mathrm{key}$ **then**
        $v \leftarrow v.\mathrm{left}$
    else
        $v \leftarrow v.\mathrm{right}$

return null

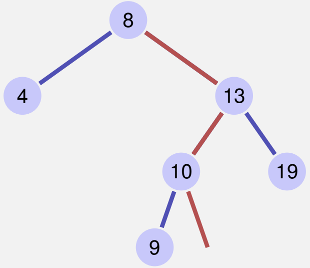


Searching

Input : Binary search tree with root r, key k
Output : Node v with $v.\mathrm{key} = k$ or **null**

$v \leftarrow r$
while $v \neq$ **null do**
    if $k = v.\mathrm{key}$ **then**
        return v
    else if $k < v.\mathrm{key}$ **then**
        $v \leftarrow v.\mathrm{left}$
    else
        $v \leftarrow v.\mathrm{right}$

return null



Search (12) $\to$ **null**

Height of a tree

The height $h(T)$ of a tree T with root r is given by

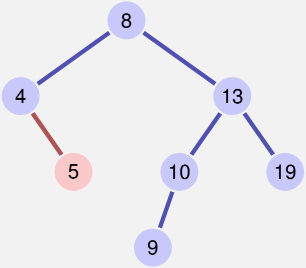
$$h(r) = \begin{cases} 0 & \text{if } r = \textbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{otherwise.} \end{cases}$$

The worst case run time of the search is thus $\mathcal{O}(h(T))$

Insertion of a key

Insertion of the key k

- Search for k
- If successful search: output error
- Of no success: insert the key at the leaf reached



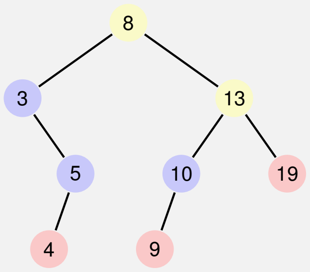
Insert (5)

Remove node

Three cases possible:

- Node has no children
- Node has one child
- Node has two children

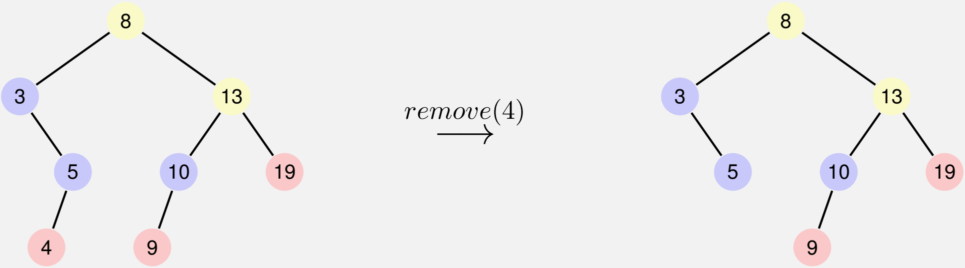
[Leaves do not count here]



Remove node

Node has no children

Simple case: replace node by leaf.

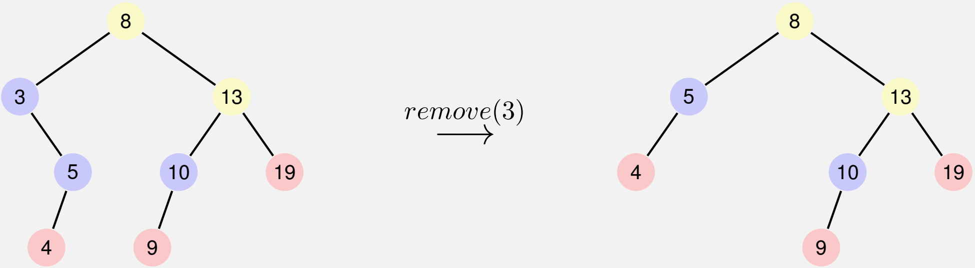


$\xrightarrow{remove(4)}$

Remove node

Node has one child

Also simple: replace node by single child.



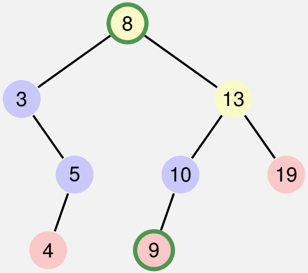
Remove node

Node has two children

The following observation helps: the smallest key in the right subtree $v.\mathrm{right}$ (the *symmetric successor* of v)

- is smaller than all keys in $v.\mathrm{right}$
- is greater than all keys in $v.\mathrm{left}$
- and cannot have a left child.

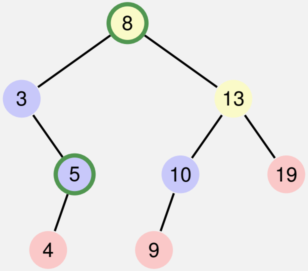
Solution: replace v by its symmetric successor.



By symmetry...

Node has two children

Also possible: replace v by its symmetric predecessor.



Algorithm SymmetricSuccessor(v)

Input : Node v of a binary search tree.
Output : Symmetric successor of v

$w \leftarrow v.\mathrm{right}$
$x \leftarrow w.\mathrm{left}$
while $x \neq$ **null do**
    $w \leftarrow x$
    $x \leftarrow x.\mathrm{left}$

return w

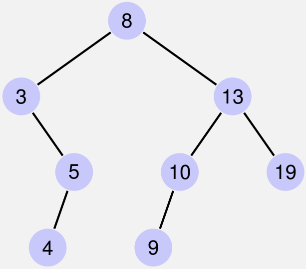
Analysis

Deletion of an element v from a tree T requires $\mathcal{O}(h(T))$ fundamental steps:

- Finding v has costs $\mathcal{O}(h(T))$
- If v has maximal one child unequal to **null**then removal takes $\mathcal{O}(1)$ steps
- Finding the symmetric successor n of v takes $\mathcal{O}(h(T))$ steps. Removal and insertion of n takes $\mathcal{O}(1)$ steps.

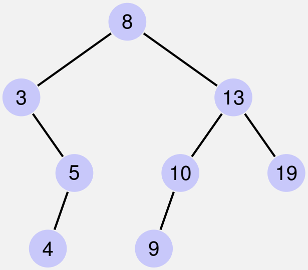
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.



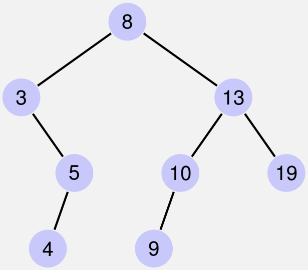
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19



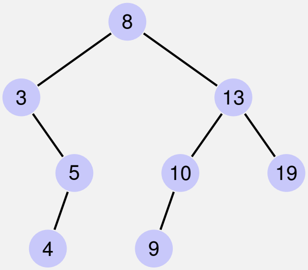
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.



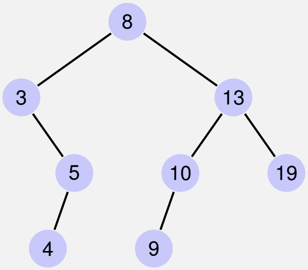
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8



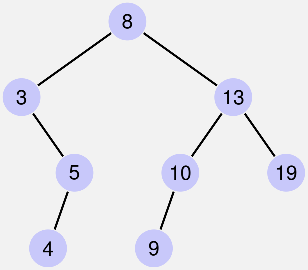
Traversal possibilities

- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- inorder: $T_{\text{left}}(v)$, then v, then $T_{\text{right}}(v)$.



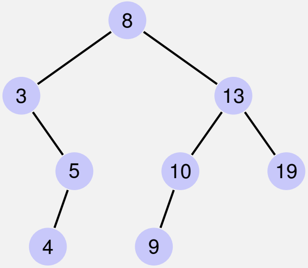
Traversal possibilities

- preorder: v, then $T_{\mathrm{left}}(v)$, then $T_{\mathrm{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\mathrm{left}}(v)$, then $T_{\mathrm{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- inorder: $T_{\mathrm{left}}(v)$, then v, then $T_{\mathrm{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19

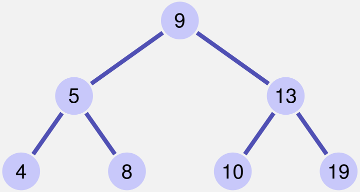


Further supported operations

- Min(T): Read-out minimal value in $\mathcal{O}(h)$
- ExtractMin(T): Read-out and remove minimal value in $\mathcal{O}(h)$
- List(T): Output the sorted list of elements
- Join(T_1, T_2): Merge two trees with $\max(T_1) < \min(T_2)$ in $\mathcal{O}(n)$.

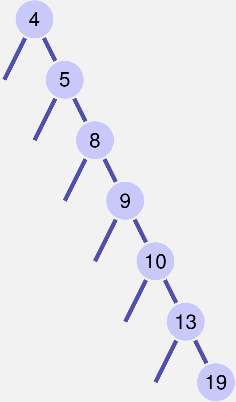


Degenerated search trees



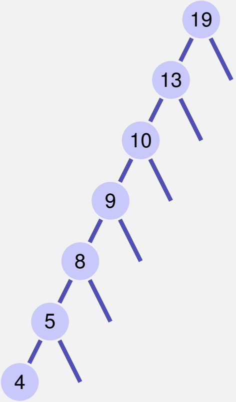
Insert 9,5,13,4,8,10,19

ideally balanced



Insert 4,5,8,9,10,13,19

linear list



Insert 19,13,10,9,8,5,4

linear list

Probabilistically

A search tree constructed from a random sequence of numbers provides an an expected path length of $\mathcal{O}(\log n)$.

Attention: this only holds for insertions. If the tree is constructed by random insertions and deletions, the expected path length is $\mathcal{O}(\sqrt{n})$.

Balanced trees make sure (e.g. with *rotations*) during insertion or deletion that the tree stays balanced and provide a $\mathcal{O}(\log n)$ Worst-case guarantee.

17. AVL Trees

Balanced Trees [Ottman/Widmayer, Kap. 5.2-5.2.1, Cormen et al, Kap. Problem 13-3]

Objective

Searching, insertion and removal of a key in a tree generated from n keys inserted in random order takes expected number of steps $\mathcal{O}(\log_2 n)$.

But worst case $\Theta(n)$ (degenerated tree).

Goal: avoidance of degeneration. Artificial balancing of the tree for each update-operation of a tree.

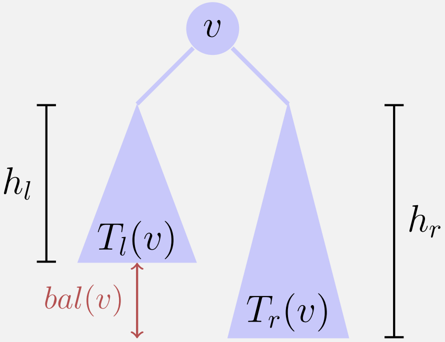
Balancing: guarantee that a tree with n nodes always has a height of $\mathcal{O}(\log n)$.

Adelson-Venskii and Landis (1962): AVL-Trees

Balance of a node

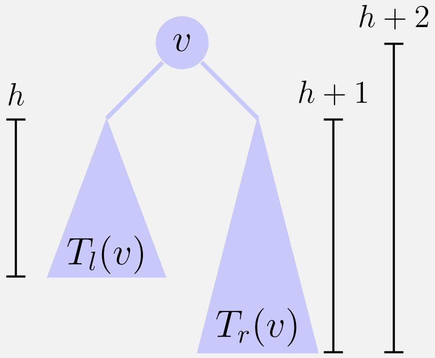
The height *balance* of a node v is defined as the height difference of its sub-trees $T_l(v)$ and $T_r(v)$

$$\mathrm{bal}(v) := h(T_r(v)) - h(T_l(v))$$

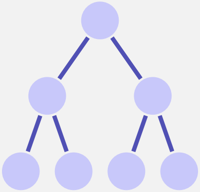


AVL Condition

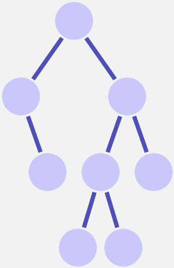
AVL Condition: for eacn node v of a tree $\mathrm{bal}(v) \in \{-1, 0, 1\}$



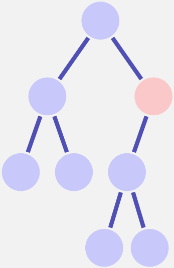
(Counter-)Examples



AVL tree with height 2



AVL tree with height 3

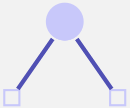


No AVL tree

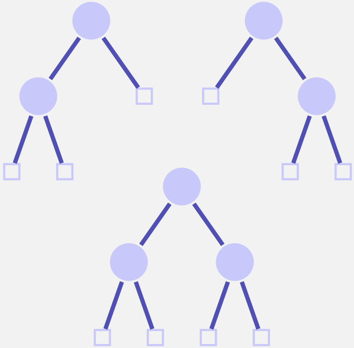
Number of Leaves

- 1. observation: a binary search tree with n keys provides exactly $n + 1$ leaves. Simple induction argument.
- 2. observation: a lower bound of the number of leaves in a search tree with given height implies an upper bound of the height of a search tree with given number of keys.

Lower bound of the leaves



AVL tree with height 1 has
$M(1) := 2$ leaves.



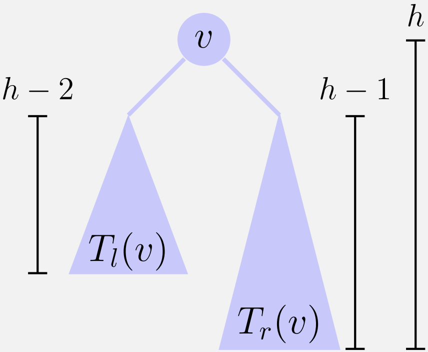
AVL tree with height 2 has
at least $M(2) := 3$ leaves.

Lower bound of the leaves for $h > 2$

- Height of one subtree $\geq h-1$.
- Height of the other subtree $\geq h-2$.

Minimal number of leaves $M(h)$ is

$$M(h) = M(h-1) + M(h-2)$$



Overal we have $M(h) = F_{h+2}$ with *Fibonacci-numbers* $F_0 := 0$, $F_1 := 1$, $F_n := F_{n-1} + F_{n-2}$ for $n > 1$.

[Fibonacci Numbers: closed form]

Closed form of the Fibonacci numbers: computation via generation functions:

1. Power series approach

$$f(x) := \sum_{i=0}^{\infty} F_i \cdot x^i$$

[Fibonacci Numbers: closed form]

2 For Fibonacci Numbers it holds that $F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2}\ \forall i > 1$. Therefore:

$$\begin{aligned} f(x) = x + \sum_{i=2}^{\infty} F_i \cdot x^i &= x + \sum_{i=2}^{\infty} F_{i-1} \cdot x^i + \sum_{i=2}^{\infty} F_{i-2} \cdot x^i \\ &= x + x \sum_{i=2}^{\infty} F_{i-1} \cdot x^{i-1} + x^2 \sum_{i=2}^{\infty} F_{i-2} \cdot x^{i-2} \\ &= x + x \sum_{i=0}^{\infty} F_i \cdot x^i + x^2 \sum_{i=0}^{\infty} F_i \cdot x^i \\ &= x + x \cdot f(x) + x^2 \cdot f(x). \end{aligned}$$

[Fibonacci Numbers: closed form]

3 Thus:

$$
\begin{aligned}
& f(x) \cdot (1 - x - x^2) = x. \\
\Leftrightarrow \quad & f(x) = \frac{x}{1 - x - x^2} = -\frac{x}{x^2 + x - 1}
\end{aligned}
$$

with the roots $-\phi$ and $-\hat{\phi}$ of $x^2 + x - 1$,

$$
\phi = \frac{1 + \sqrt{5}}{2} \approx 1.6, \qquad \hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.6.
$$

it holds that $\phi \cdot \hat{\phi} = -1$ and thus

$$
f(x) = -\frac{x}{(x + \phi) \cdot (x + \hat{\phi})} = \frac{x}{(1 - \phi x) \cdot (1 - \hat{\phi} x)}
$$

[Fibonacci Numbers: closed form]

4 It holds that:

$$(1-\hat{\phi}x)-(1-\phi x)=\sqrt{5}\cdot x.$$

Damit:

$$\begin{aligned} f(x) &= \frac{1}{\sqrt{5}}\frac{(1-\hat{\phi}x)-(1-\phi x)}{(1-\phi x)\cdot(1-\hat{\phi}x)} \\ &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi x}-\frac{1}{1-\hat{\phi}x}\right) \end{aligned}$$

[Fibonacci Numbers: closed form]

5 Power series of $g_a(x) = \frac{1}{1-a\cdot x}$ ($a \in \mathbb{R}$):

$$\frac{1}{1-a\cdot x} = \sum_{i=0}^{\infty} a^i \cdot x^i.$$

E.g. Taylor series of $g_a(x)$ at $x = 0$ or like this: Let $\sum_{i=0}^{\infty} G_i \cdot x^i$ a power series of g. By the identity $g_a(x)(1 - a\cdot x) = 1$ it holds that for all x (within the radius of convergence)

$$1 = \sum_{i=0}^{\infty} G_i \cdot x^i - a \cdot \sum_{i=0}^{\infty} G_i \cdot x^{i+1} = G_0 + \sum_{i=1}^{\infty} (G_i - a \cdot G_{i-1}) \cdot x^i$$

For $x = 0$ it follows $G_0 = 1$ and for $x \neq 0$ it follows then that $G_i = a \cdot G_{i-1} \Rightarrow G_i = a^i$.

[Fibonacci Numbers: closed form]

6 Fill in the power series:

$$f(x) = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi x} - \frac{1}{1-\hat{\phi} x}\right) = \frac{1}{\sqrt{5}}\left(\sum_{i=0}^{\infty} \phi^i x^i - \sum_{i=0}^{\infty} \hat{\phi}^i x^i\right)$$

$$= \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) x^i$$

Comparison of the coefficients with $f(x) = \sum_{i=0}^{\infty} F_i \cdot x^i$ yields

$$F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

Fibonacci Numbers, Inductive Proof

It holds that $F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$ with roots ϕ, $\hat{\phi}$ of the equation $x^2 = x + 1$ (golden ratio), thus $\phi = \frac{1+\sqrt{5}}{2}$, $\hat{\phi} = \frac{1-\sqrt{5}}{2}$.

Proof (induction). Immediate for $i = 0, i = 1$. Let $i > 2$:

$$
\begin{aligned}
F_i = F_{i-1} + F_{i-2} &= \frac{1}{\sqrt{5}}(\phi^{i-1} - \hat{\phi}^{i-1}) + \frac{1}{\sqrt{5}}(\phi^{i-2} - \hat{\phi}^{i-2}) \\
&= \frac{1}{\sqrt{5}}(\phi^{i-1} + \phi^{i-2}) - \frac{1}{\sqrt{5}}(\hat{\phi}^{i-1} + \hat{\phi}^{i-2}) = \frac{1}{\sqrt{5}}\phi^{i-2}(\phi + 1) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi} + 1) \\
&= \frac{1}{\sqrt{5}}\phi^{i-2}(\phi^2) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi}^2) = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).
\end{aligned}
$$

Tree Height

Because $\hat{\phi} < 1$, overal we have

$$M(h) \in \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right) \subseteq \Omega(1.618^h)$$

and thus

$$h \leq 1.44 \log_2 n + c.$$

AVL tree is asymptotically not more than 44% higher than a perfectly balanced tree.

Insertion

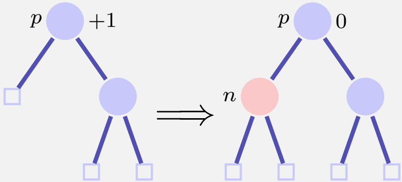
Balance

- Keep the balance stored in each node
- Re-balance the tree in each update-operation

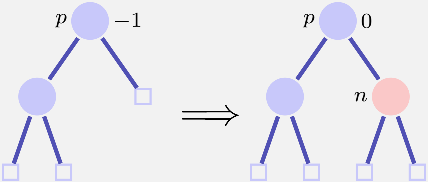
New node n is inserted:

- Insert the node as for a search tree.
- Check the balance condition increasing from n to the root.

Balance at Insertion Point



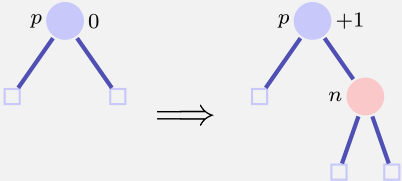
case 1: $\mathrm{bal}(p) = +1$



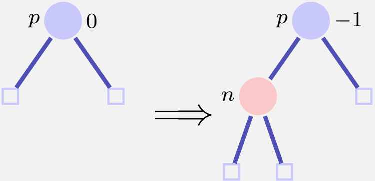
case 2: $\mathrm{bal}(p) = -1$

Finished in both cases because the subtree height did not change

Balance at Insertion Point



case 3.1: $\mathrm{bal}(p) = 0$ right



case 3.2: $\mathrm{bal}(p) = 0$, left

Not finished in both case. Call of `upin(p)`

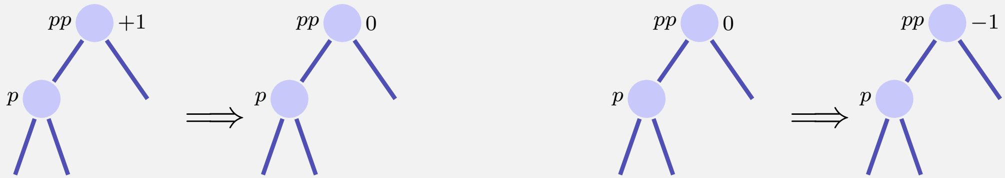
upin(p) - invariant

When `upin(p)` is called it holds that

- the subtree from p is grown and
- $\mathrm{bal}(p) \in \{-1, +1\}$

upin(p)

Assumption: p is left son of pp[20]



case 1: $\mathrm{bal}(pp) = +1$, done.

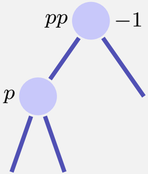
case 2: $\mathrm{bal}(pp) = 0$, `upin(pp)`

In both cases the AVL-Condition holds for the subtree from pp

[20] If p is a right son: symmetric cases with exchange of $+1$ and -1

upin(p)

Assumption: p is left son of pp



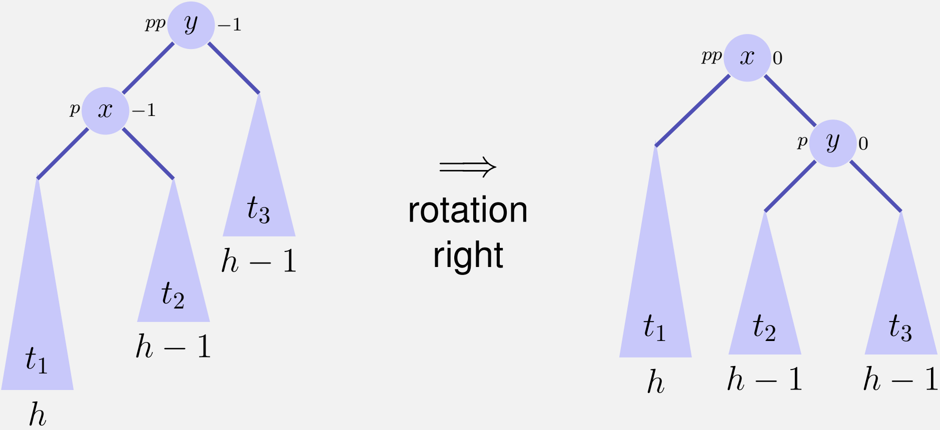
case 3: $\mathrm{bal}(pp) = -1$,

This case is problematic: adding n to the subtree from pp has violated the AVL-condition. Re-balance!

Two cases $\mathrm{bal}(p) = -1$, $\mathrm{bal}(p) = +1$

Rotationen

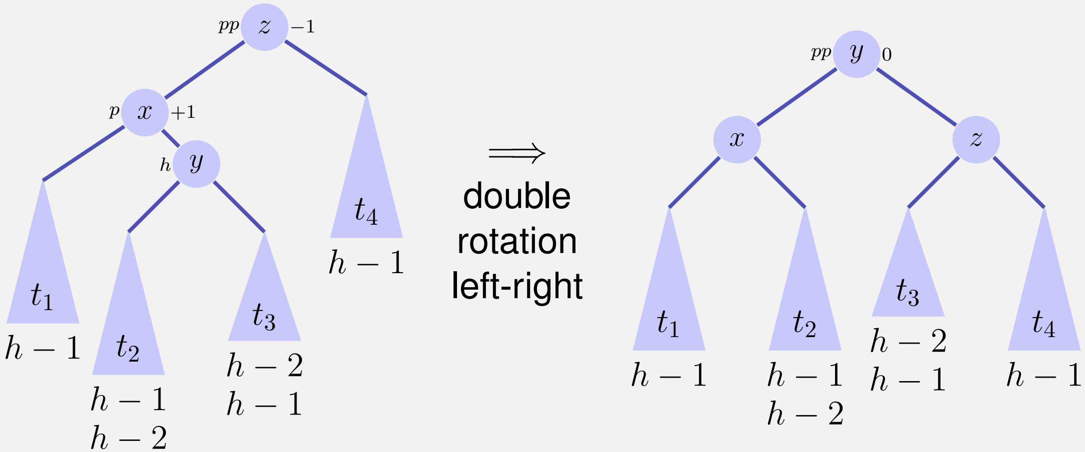
case 1.1 $\mathrm{bal}(p) = -1$. [21]



[21] p right son: $\mathrm{bal}(pp) = \mathrm{bal}(p) = +1$, left rotation

Rotationen

case 1.1 $\mathrm{bal}(p) = -1$. [22]



[22] p right son: $\mathrm{bal}(pp) = +1$, $\mathrm{bal}(p) = -1$, double rotation right left

Analysis

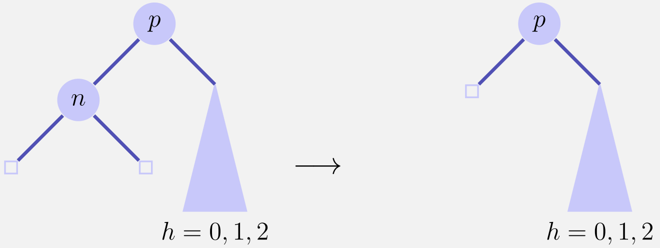
- Tree height: $\mathcal{O}(\log n)$.
- Insertion like in binary search tree.
- Balancing via recursion from node to the root. Maximal path lenght $\mathcal{O}(\log n)$.

Insertion in an AVL-tree provides run time costs of $\mathcal{O}(\log n)$.

Deletion

Case 1: Children of node n are both leaves Let p be parent node of n. $\Rightarrow$ Other subtree has height $h' = 0$, 1 or 2.

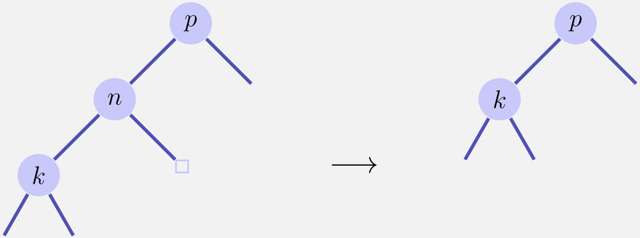
- $h' = 1$: Adapt $\mathrm{bal}(p)$.
- $h' = 0$: Adapt $\mathrm{bal}(p)$. Call `upout(p)`.
- $h' = 2$: Rebalanciere des Teilbaumes. Call `upout(p)`.



Deletion

Case 2: one child k of node n is an inner node

- Replace n by k. `upout(k)`



Deletion

Case 3: both children of node n are inner nodes

- Replace n by symmetric successor. `upout(k)`
- Deletion of the symmetric successor is as in case 1 or 2.

`upout(p)`

Let pp be the parent node of p.

(a) p left child of pp

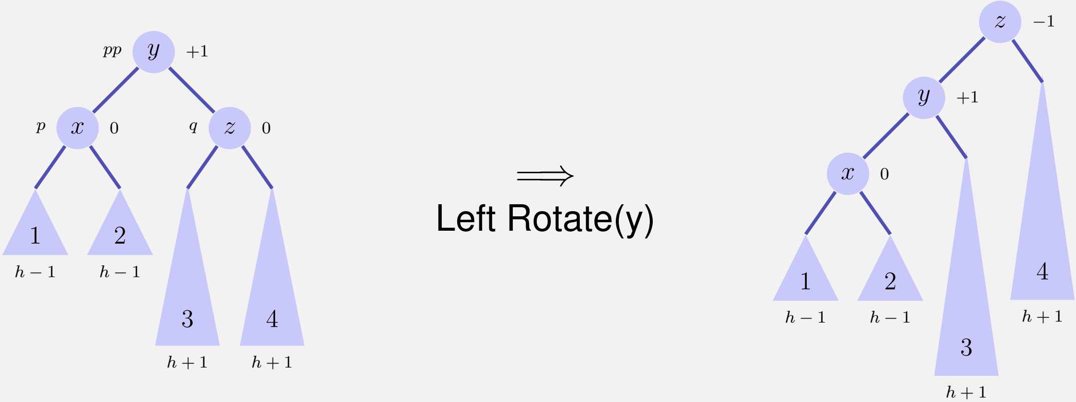
1. $\mathrm{bal}(pp) = -1 \;\Rightarrow\; \mathrm{bal}(pp) \leftarrow 0$. `upout(pp)`
2. $\mathrm{bal}(pp) = 0 \;\Rightarrow\; \mathrm{bal}(pp) \leftarrow +1$.
3. $\mathrm{bal}(pp) = +1 \Rightarrow$ next slides.

(b) p right child of pp: Symmetric cases exchanging $+1$ and -1.

`upout(p)`

Case (a).3: $\mathrm{bal}(pp) = +1$. Let q be brother of p

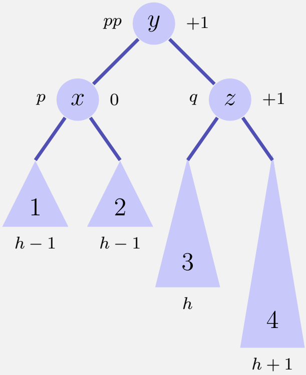
(a).3.1: $\mathrm{bal}(q) = 0$.[23]



[23] (b).3.1: $\mathrm{bal}(pp) = -1$, $\mathrm{bal}(q) = -1$, Right rotation

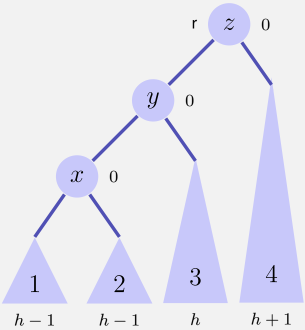
`upout(p)`

Case (a).3: $\mathrm{bal}(pp) = +1$. (a).3.2: $\mathrm{bal}(q) = +1$.[24]



$\Longrightarrow$

Left Rotate(y)

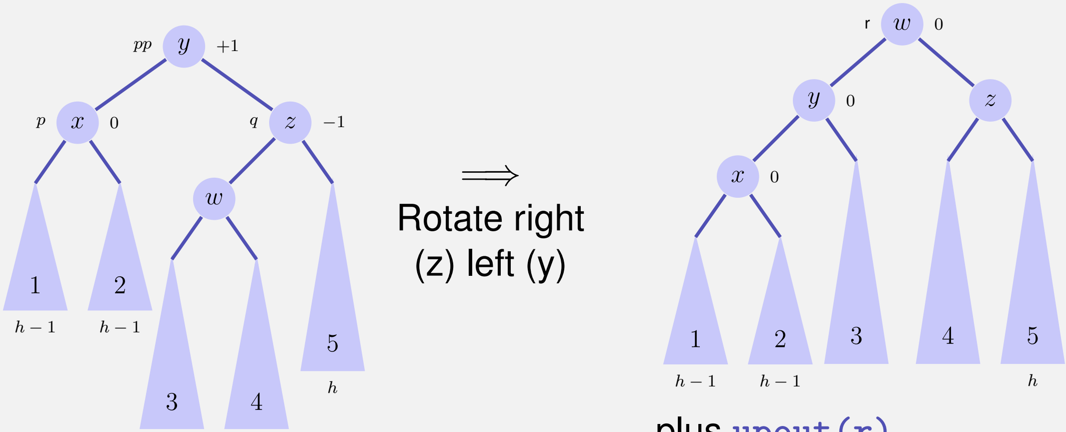


plus `upout(r)`.

[24](b).3.2: $\mathrm{bal}(pp) = -1$, $\mathrm{bal}(q) = +1$, Right rotation+upout

upout(p)

Case (a).3: $\mathrm{bal}(pp) = +1$. (a).3.3: $\mathrm{bal}(q) = -1$.[25]



plus upout(r).

[25] (b).3.3: $\mathrm{bal}(pp) = -1$, $\mathrm{bal}(q) = -1$, left-right rotation + upout

Conclusion

- AVL trees have worst-case asymptotic runtimes of $\mathcal{O}(\log n)$ for searching, insertion and deletion of keys.
- Insertion and deletion is relatively involved and an overkill for really small problems.