

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:**

- 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

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
- Key k in given interval $k \in [l, r]$

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.

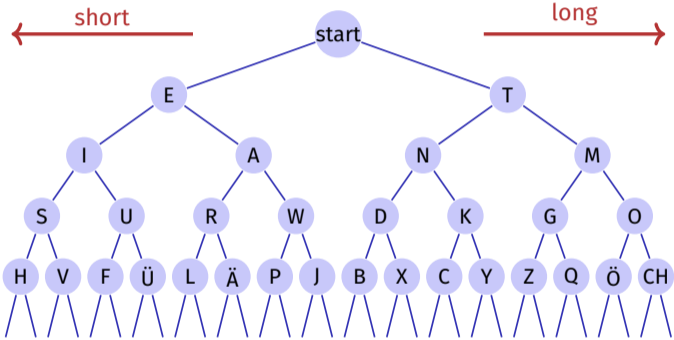
Trees

Use

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

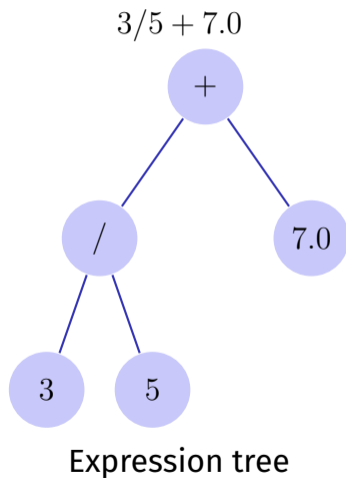


Examples

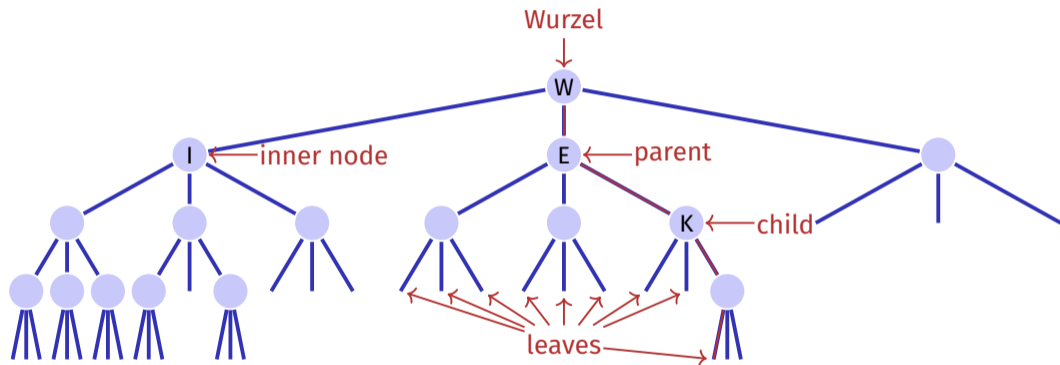


Morsealphabet

Examples



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 inner node v we store

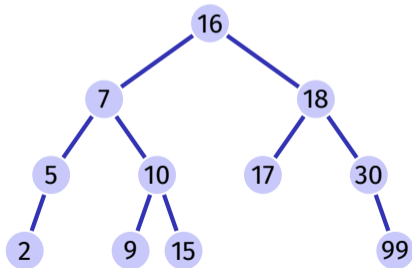
- a key $v.key$ and
 - two nodes $v.left$ and $v.right$ to the roots of the left and right subtree.
- a leaf is represented by the **null**-pointer



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 left subtree $v.\text{left}$ are smaller than $v.\text{key}$
- Keys in right subtree $v.\text{right}$ are greater than $v.\text{key}$



Searching

Input: Binary search tree with root r , key k

Output: Node v with $v.\text{key} = k$ or **null**

$v \leftarrow r$

while $v \neq \text{null}$ **do**

if $k = v.\text{key}$ **then**

 | **return** v

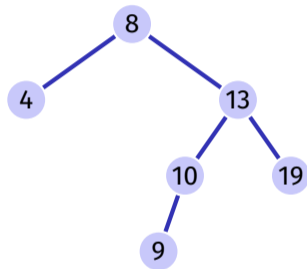
else if $k < v.\text{key}$ **then**

 | $v \leftarrow v.\text{left}$

else

 | $v \leftarrow v.\text{right}$

return null



Searching

Input: Binary search tree with root r , key k

Output: Node v with $v.\text{key} = k$ or **null**

$v \leftarrow r$

while $v \neq \text{null}$ **do**

if $k = v.\text{key}$ **then**

 | **return** v

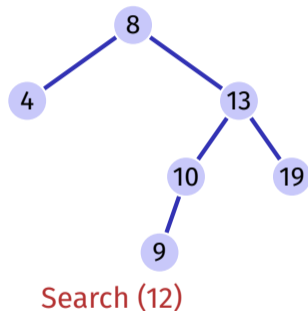
else if $k < v.\text{key}$ **then**

 | $v \leftarrow v.\text{left}$

else

 | $v \leftarrow v.\text{right}$

return null



Searching

Input: Binary search tree with root r , key k

Output: Node v with $v.\text{key} = k$ or **null**

$v \leftarrow r$

while $v \neq \text{null}$ **do**

if $k = v.\text{key}$ **then**

 | **return** v

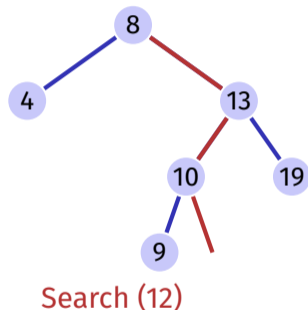
else if $k < v.\text{key}$ **then**

 | $v \leftarrow v.\text{left}$

else

 | $v \leftarrow v.\text{right}$

return null



Searching

Input: Binary search tree with root r , key k

Output: Node v with $v.\text{key} = k$ or **null**

$v \leftarrow r$

while $v \neq \text{null}$ **do**

if $k = v.\text{key}$ **then**

 | **return** v

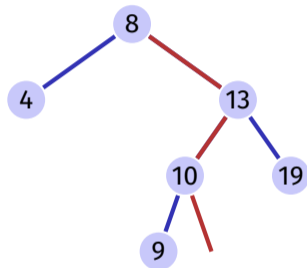
else if $k < v.\text{key}$ **then**

 | $v \leftarrow v.\text{left}$

else

 | $v \leftarrow v.\text{right}$

return null



Search (12) \rightarrow **null**

Height of a tree

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

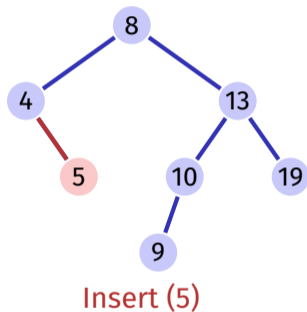
$$h(r) = \begin{cases} 0 & \text{if } r = \mathbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{otherwise.} \end{cases}$$

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: e.g. output error
- Of no success: insert the key at the leaf reached

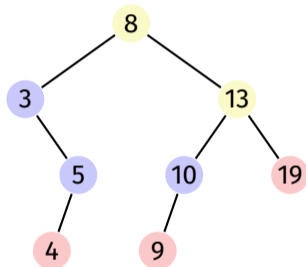


Remove node

Three cases possible:

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

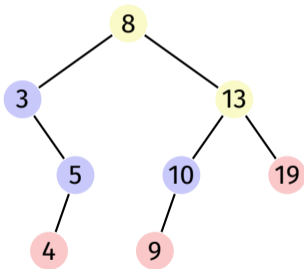
[Leaves do not count here]



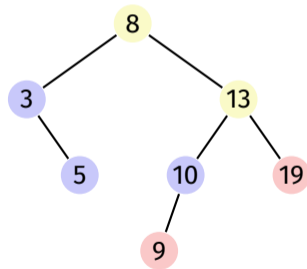
Remove node

Node has no children

Simple case: replace node by leaf.



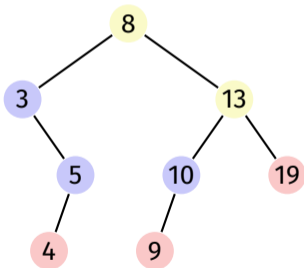
remove(4)
→



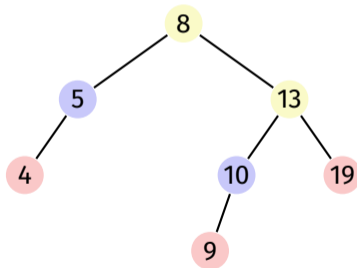
Remove node

Node has one child

Also simple: replace node by single child.



remove(3)
→



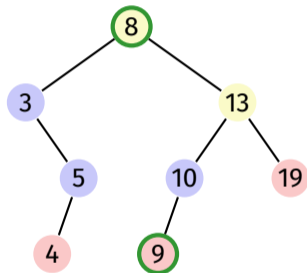
Remove node

Node v has two children

The following observation helps: the smallest key in the right subtree $v.right$ (the symmetric successor of v)

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

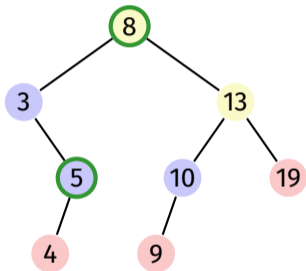
Solution: replace v by its symmetric successor.



By symmetry...

Node v 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

$w \leftarrow v.\text{right}$

$x \leftarrow w.\text{left}$

while $x \neq \text{null}$ **do**

$w \leftarrow x$

$x \leftarrow x.\text{left}$

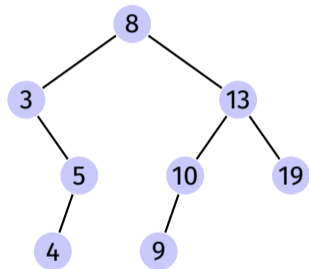
return w

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.

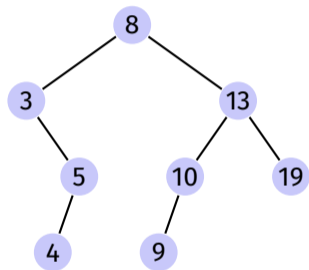
Traversal possibilities

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



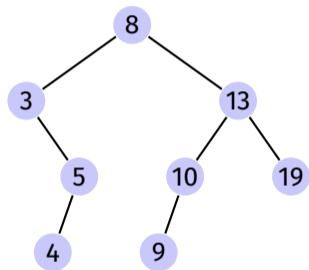
Traversal possibilities

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



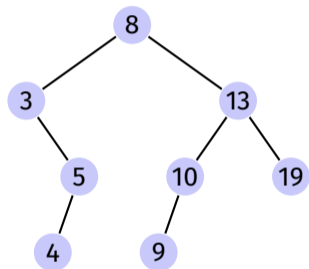
Traversal possibilities

- 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 .



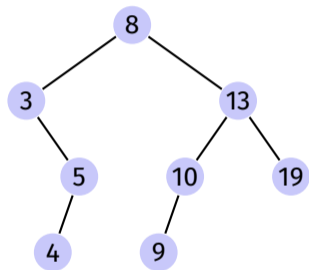
Traversal possibilities

- 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



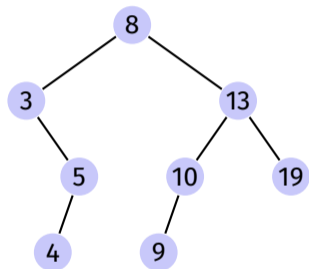
Traversal possibilities

- 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 $T_{\text{right}}(v)$.



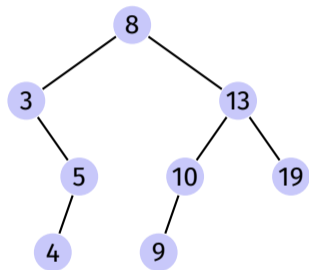
Traversal possibilities

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

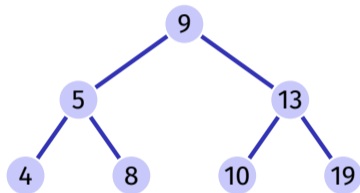


Further supported operations

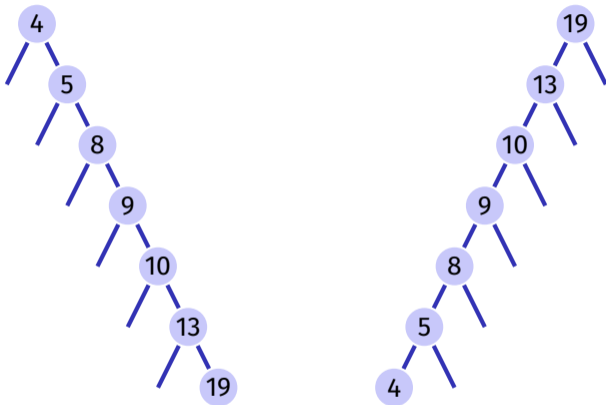
- $\text{Min}(T)$: Read-out minimal value in $\mathcal{O}(h)$
- $\text{ExtractMin}(T)$: Read-out and remove minimal value in $\mathcal{O}(h)$
- $\text{List}(T)$: Output the sorted list of elements
- $\text{Join}(T_1, T_2)$: Merge two trees with $\max(T_1) < \min(T_2)$ in $\mathcal{O}(n)$.



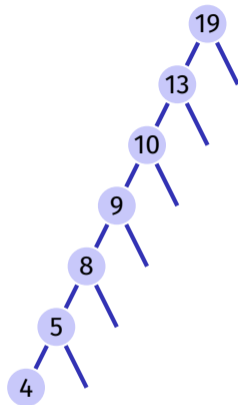
Degenerated search trees



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 $\mathcal{O}(\sqrt{n})$.

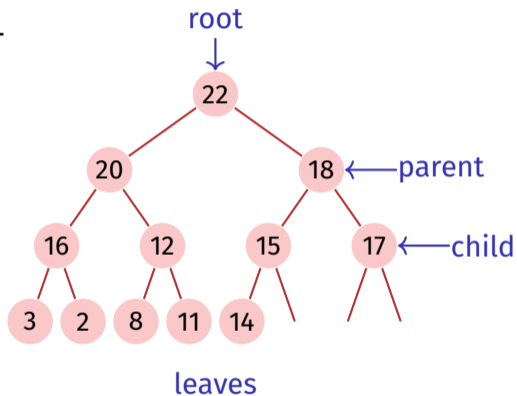
Balanced trees make sure (e.g. with *rotations*) during insertion or deletion that the tree stays balanced and provide a $\mathcal{O}(\log n)$ Worst-case guarantee.

17. Heaps

Data structure optimized for fast extraction of minimum or maximum and for sorting. [Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

[Max-]Heap*

Binary tree with the following properties

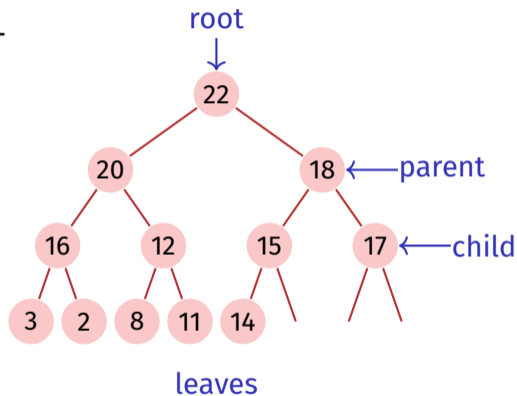


*Heap(data structure), not: as in “heap and stack” (memory allocation)

[Max-]Heap*

Binary tree with the following properties

1. complete up to the lowest level

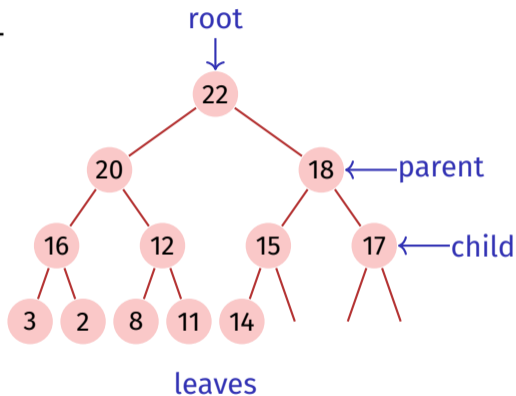


*Heap(data structure), not: as in “heap and stack” (memory allocation)

[Max-]Heap*

Binary tree with the following properties

1. complete up to the lowest level
2. Gaps (if any) of the tree in the last level to the right

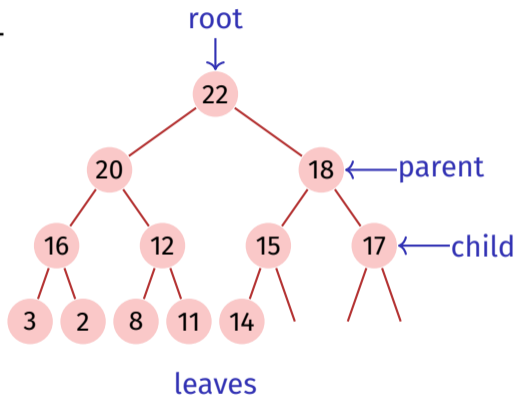


*Heap(data structure), not: as in “heap and stack” (memory allocation)

[Max-]Heap*

Binary tree with the following properties

1. complete up to the lowest level
2. Gaps (if any) of the tree in the last level to the right
3. Heap-Condition:
Max-(Min-)Heap: key of a child smaller (greater) than that of the parent node

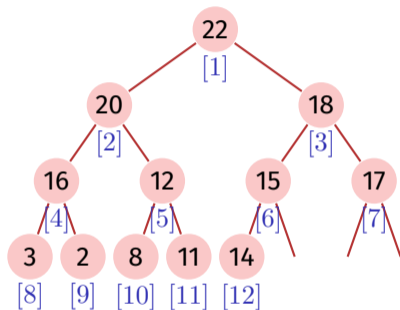
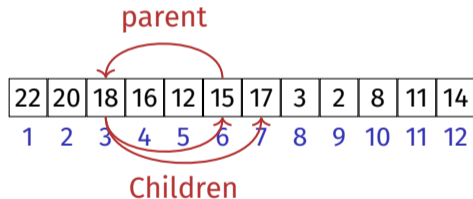


*Heap(data structure), not: as in “heap and stack” (memory allocation)

Heap as Array

Tree \rightarrow Array:

- $\text{children}(i) = \{2i, 2i + 1\}$
- $\text{parent}(i) = \lfloor i/2 \rfloor$



Depends on the starting index¹⁹

¹⁹For array that start at 0: $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$

Height of a Heap

What is the height $H(n)$ of Heap with n nodes? On the i -th level of a binary tree there are at most 2^i nodes. Up to the last level of a heap all levels are filled with values.

$$H(n) = \min\{h \in \mathbb{N} : \sum_{i=0}^{h-1} 2^i \geq n\}$$

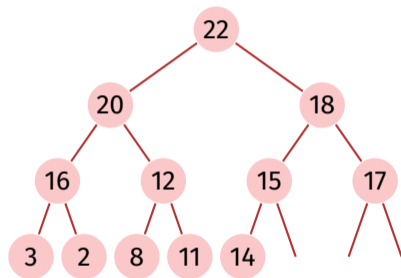
with $\sum_{i=0}^{h-1} 2^i = 2^h - 1$:

$$H(n) = \min\{h \in \mathbb{N} : 2^h \geq n + 1\},$$

thus

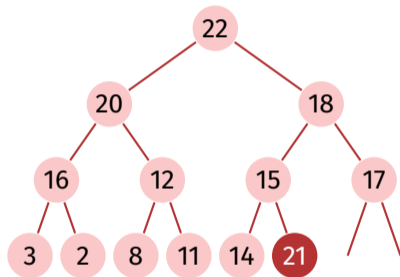
$$H(n) = \lceil \log_2(n + 1) \rceil.$$

Insert



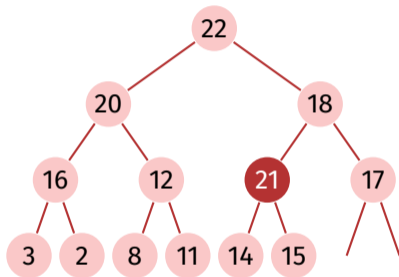
Insert

- Insert new element at the first free position. Potentially violates the heap property.



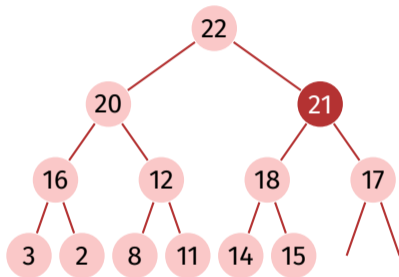
Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively



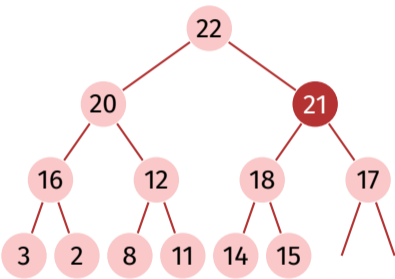
Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively



Insert

- Insert new element at the first free position. Potentially violates the heap property.
- Reestablish heap property: climb successively
- Worst case number of operations: $\mathcal{O}(\log n)$



Algorithm Sift-Up(A, m)

Input: Array A with at least m elements and Max-Heap-Structure on $A[1, \dots, m-1]$

Output: Array A with Max-Heap-Structure on $A[1, \dots, m]$.

$v \leftarrow A[m]$ // value

$c \leftarrow m$ // current position (child)

$p \leftarrow \lfloor c/2 \rfloor$ // parent node

while $c > 1$ and $v > A[p]$ **do**

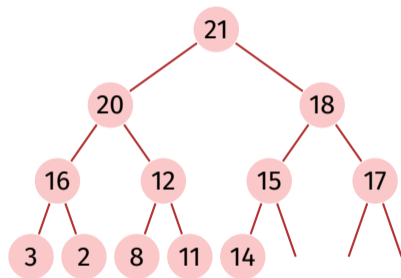
$A[c] \leftarrow A[p]$ // Value parent node \rightarrow current node

$c \leftarrow p$ // parent node \rightarrow current node

$p \leftarrow \lfloor c/2 \rfloor$

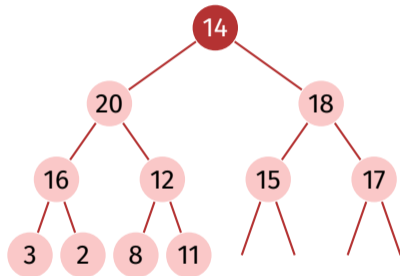
$A[c] \leftarrow v$ // value \rightarrow root of the (sub)tree

Remove the maximum



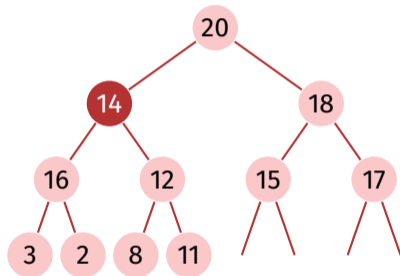
Remove the maximum

- Replace the maximum by the lower right element



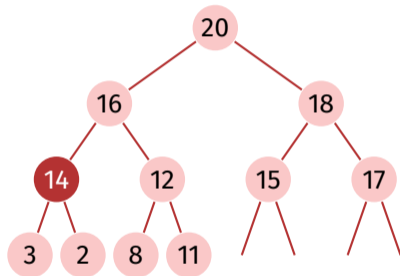
Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)



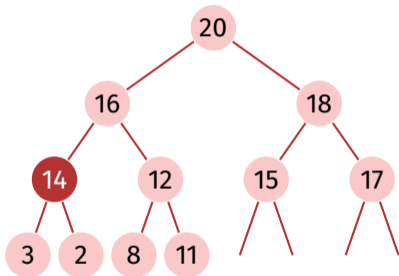
Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)



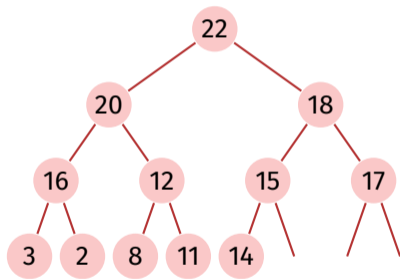
Remove the maximum

- Replace the maximum by the lower right element
- Reestablish heap property: sink successively (in the direction of the greater child)
- Worst case number of operations: $\mathcal{O}(\log n)$



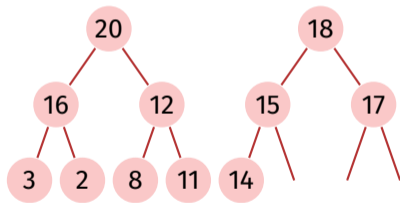
Why this is correct: Recursive heap structure

A heap consists of two heaps:



Why this is correct: Recursive heap structure

A heap consists of two heaps:



Algorithm SiftDown(A, i, m)

Input: Array A with heap structure for the children of i . Last element m .

Output: Array A with heap structure for i with last element m .

while $2i \leq m$ **do**

$j \leftarrow 2i$; // j left child

if $j < m$ and $A[j] < A[j + 1]$ **then**

$j \leftarrow j + 1$; // j right child with greater key

if $A[i] < A[j]$ **then**

 swap($A[i], A[j]$)

$i \leftarrow j$; // keep sinking down

else

$i \leftarrow m$; // sift down finished

Sort heap



$A[1, \dots, n]$ is a Heap.

While $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{SiftDown}(A, 1, n - 1);$
- $n \leftarrow n - 1$

Sort heap

swap \Rightarrow

7	6	4	5	1	2
2	6	4	5	1	7

$A[1, \dots, n]$ is a Heap.

While $n > 1$

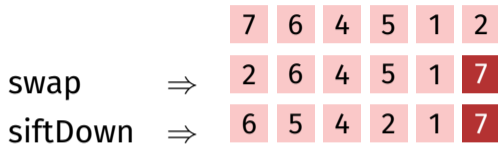
- $\text{swap}(A[1], A[n])$
- $\text{SiftDown}(A, 1, n - 1);$
- $n \leftarrow n - 1$

Sort heap

$A[1, \dots, n]$ is a Heap.

While $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{SiftDown}(A, 1, n - 1);$
- $n \leftarrow n - 1$



Sort heap

$A[1, \dots, n]$ is a Heap.

While $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{SiftDown}(A, 1, n - 1);$
- $n \leftarrow n - 1$

swap

\Rightarrow

7	6	4	5	1	2
---	---	---	---	---	---

siftDown

\Rightarrow

2	6	4	5	1	7
---	---	---	---	---	---

swap

\Rightarrow

6	5	4	2	1	7
---	---	---	---	---	---

1	5	4	2	6	7
---	---	---	---	---	---

Sort heap

$A[1, \dots, n]$ is a Heap.

While $n > 1$

- $\text{swap}(A[1], A[n])$
- $\text{SiftDown}(A, 1, n - 1);$
- $n \leftarrow n - 1$



Heap creation

Observation: Every leaf of a heap is trivially a correct heap.

Consequence:

Heap creation

Observation: Every leaf of a heap is trivially a correct heap.

Consequence: Induction from below!

Algorithm HeapSort(A, n)

Input: Array A with length n .

Output: A sorted.

// Build the heap.

for $i \leftarrow n/2$ **downto** 1 **do**

└ SiftDown(A, i, n);

// Now A is a heap.

for $i \leftarrow n$ **downto** 2 **do**

└ swap($A[1], A[i]$)

└ SiftDown($A, 1, i - 1$)

// Now A is sorted.

Analysis: sorting a heap

SiftDown traverses at most $\log n$ nodes. For each node 2 key comparisons.

\Rightarrow sorting a heap costs in the worst case $2 \log n$ comparisons.

Number of memory movements of sorting a heap also $\mathcal{O}(n \log n)$.

Analysis: creating a heap

Calls to siftDown: $n/2$.

Thus number of comparisons and movements: $v(n) \in \mathcal{O}(n \log n)$.

But mean length of the sift-down paths is much smaller:

We use that $h(n) = \lceil \log_2 n + 1 \rceil = \lfloor \log_2 n \rfloor + 1$ für $n > 0$

$$\begin{aligned} v(n) &= \sum_{l=0}^{\lfloor \log_2 n \rfloor} \underbrace{2^l}_{\text{number heaps on level } l} \cdot \underbrace{(\lfloor \log_2 n \rfloor + 1 - l - 1)}_{\text{height heaps on level } l} = \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^{\lfloor \log_2 n \rfloor - k} \cdot k \\ &= 2^{\lfloor \log_2 n \rfloor} \cdot \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k} \leq n \cdot \sum_{k=0}^{\infty} \frac{k}{2^k} \leq n \cdot 2 \in \mathcal{O}(n) \end{aligned}$$

with $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ ($0 < x < 1$) and $s(\frac{1}{2}) = 2$

Disadvantages

Heapsort: $\mathcal{O}(n \log n)$ Comparisons and movements.

Disadvantages of heapsort?

Disadvantages

Heapsort: $\mathcal{O}(n \log n)$ Comparisons and movements.

Disadvantages of heapsort?

- ⚠ Missing locality: heapsort jumps around in the sorted array (negative cache effect).

Disadvantages

Heapsort: $\mathcal{O}(n \log n)$ Comparisons and movements.

Disadvantages of heapsort?

- ❗ Missing locality: heapsort jumps around in the sorted array (negative cache effect).
- ❗ Two comparisons required before each necessary memory movement.

18. 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.

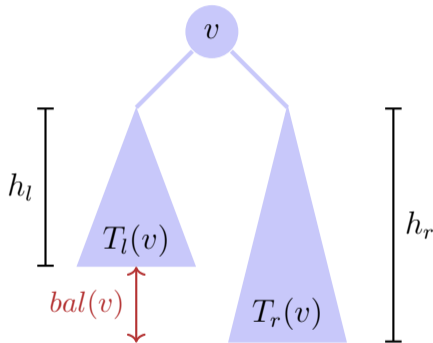
Balancing: guarantee that a tree with n nodes always has a height of $\mathcal{O}(\log n)$.

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

Balance of a node

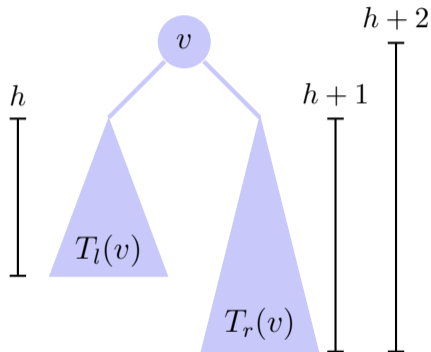
The height balance of a node v is defined as the height difference of its sub-trees $T_l(v)$ and $T_r(v)$

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

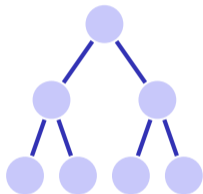


AVL Condition

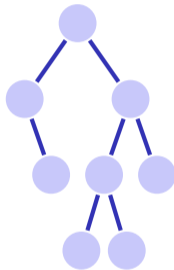
AVL Condition: for each node v of a tree
 $\text{bal}(v) \in \{-1, 0, 1\}$



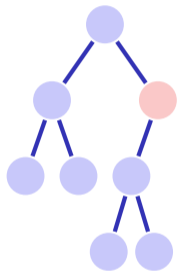
(Counter-)Examples



AVL tree with height 2



AVL tree with height 3

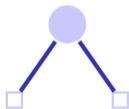


No AVL tree

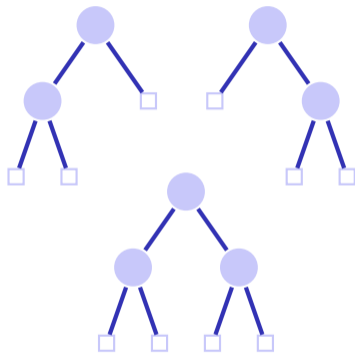
Number of Leaves

- 1. observation: a binary search tree with n keys provides exactly $n + 1$ leaves. Simple induction argument.
 - The binary search tree with $n = 0$ keys has $m = 1$ leaves
 - When a key is added ($n \rightarrow n + 1$), then it replaces a leaf and adds two new leaves ($m \rightarrow m - 1 + 2 = m + 1$).
- 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
 $N(1) := 2$ leaves.



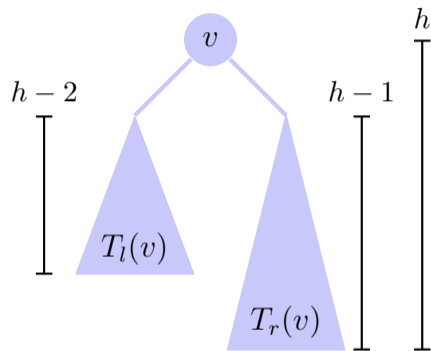
AVL tree with height 2 has at
least $N(2) := 3$ leaves.

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 $N(h)$ is

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



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

Fibonacci Numbers, closed Form

It holds that²⁰

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

with the roots $\phi, \hat{\phi}$ of the golden ratio equation $x^2 - x - 1 = 0$:

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618$$

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.618$$

²⁰Derivation using generating functions (power series) in the appendix.

Fibonacci Numbers, Inductive Proof

$$F_i \stackrel{!}{=} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) \quad [*] \quad \left(\phi = \frac{1+\sqrt{5}}{2}, \hat{\phi} = \frac{1-\sqrt{5}}{2}\right).$$

1. Immediate for $i = 0, i = 1$.
2. Let $i > 2$ and claim $[*]$ true for all $F_j, j < i$.

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

$(\phi, \hat{\phi} \text{ fulfil } x + 1 = x^2)$

$$= \frac{1}{\sqrt{5}}\phi^{i-2}(\phi^2) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi}^2) = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

Tree Height

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

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

and thus

$$\begin{aligned} N(h) &\geq c \cdot 1.618^h \\ \Rightarrow h &\leq 1.44 \log_2 n + c' \end{aligned}$$

An AVL tree is asymptotically not more than 44% higher than a perfectly balanced tree.²¹

²¹The perfectly balanced tree has a height of $\lceil \log_2 n + 1 \rceil$

Insertion

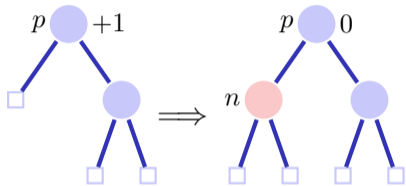
Balance

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

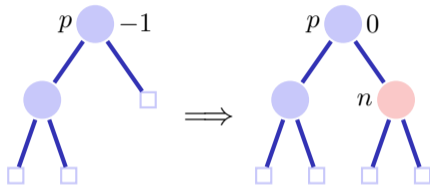
New node n is inserted:

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

Balance at Insertion Point



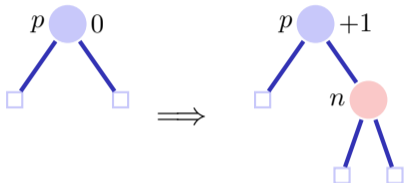
case 1: $\text{bal}(p) = +1$



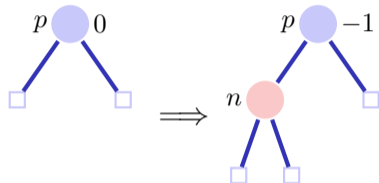
case 2: $\text{bal}(p) = -1$

Finished in both cases because the subtree height did not change

Balance at Insertion Point



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



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

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

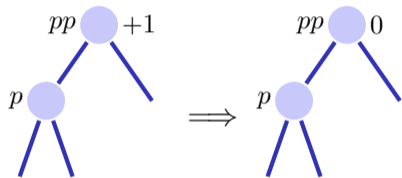
upin(p) - invariant

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

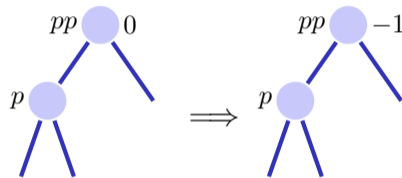
- the subtree from p is grown and
- $\text{bal}(p) \in \{-1, +1\}$

upin(p)

Assumption: p is left son of pp ²²



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



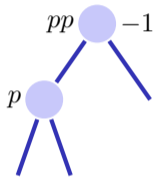
case 2: $\text{bal}(pp) = 0$, $\text{upin}(pp)$

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

²²If p is a right son: symmetric cases with exchange of $+1$ and -1

upin(p)

Assumption: p is left son of pp



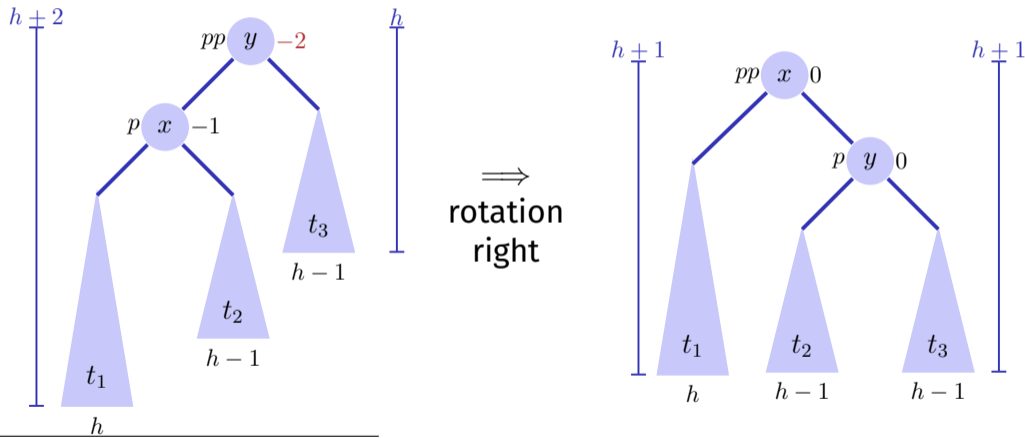
case 3: $\text{bal}(pp) = -1,$

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

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

Rotations

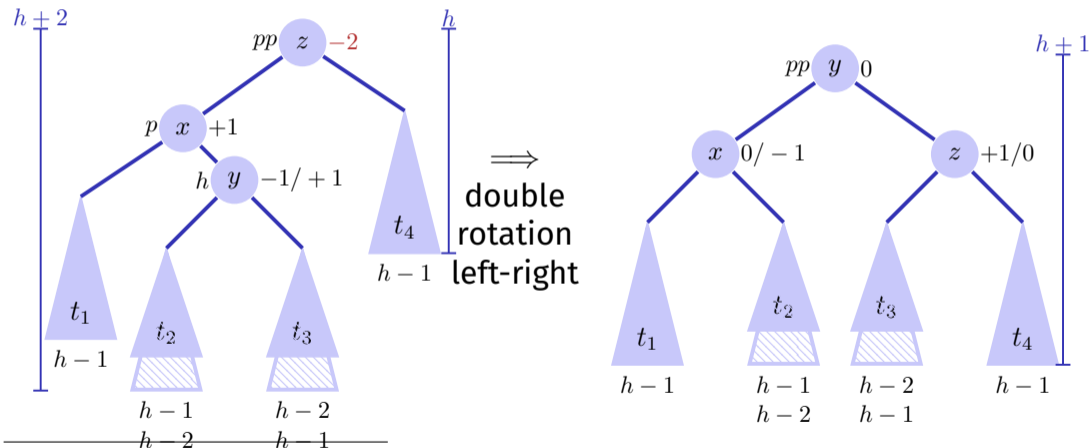
case 1.1 $\text{bal}(p) = -1$.²³



²³ p right son: $\Rightarrow \text{bal}(pp) = \text{bal}(p) = +1$, left rotation

Rotations

case 1.2 $\text{bal}(p) = +1$.²⁴



²⁴ p right son $\Rightarrow \text{bal}(pp) = +1, \text{bal}(p) = -1$, double rotation right left

Analysis

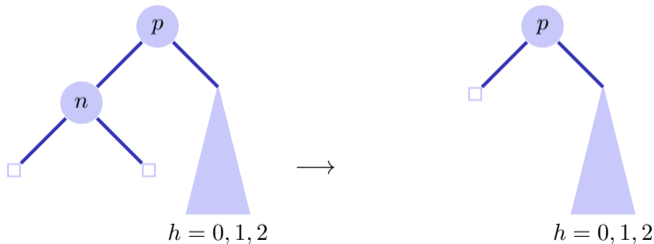
- Tree height: $\mathcal{O}(\log n)$.
- Insertion like in binary search tree.
- Balancing via recursion from node to the root. Maximal path length $\mathcal{O}(\log n)$.

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 .

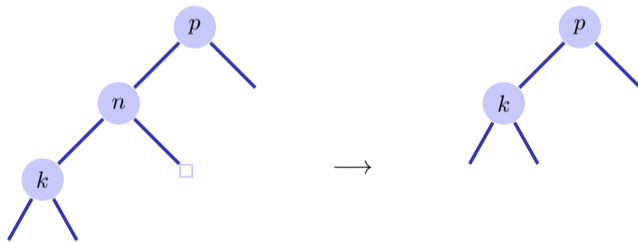
- $h' = 1$: Adapt $\text{bal}(p)$.
- $h' = 0$: Adapt $\text{bal}(p)$. Call $\text{upout}(p)$.
- $h' = 2$: Rebalanciere des Teilbaumes. Call $\text{upout}(p)$.



Deletion

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

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



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

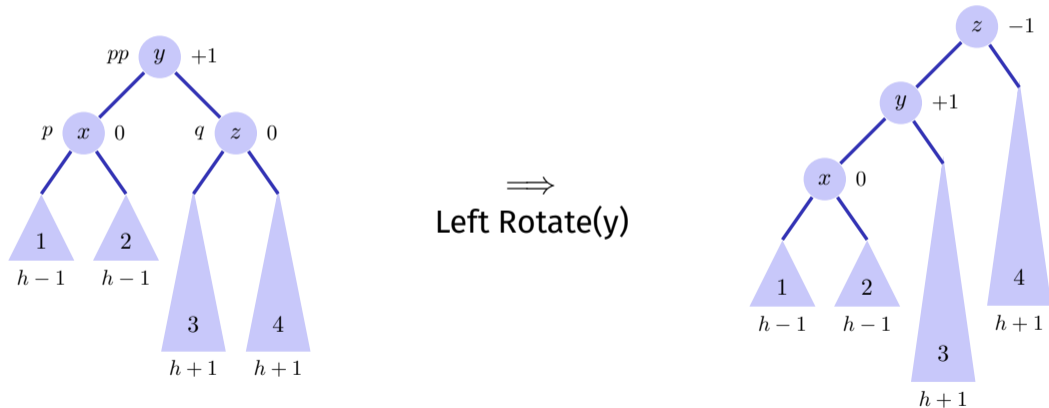
1. $\text{bal}(pp) = -1 \Rightarrow \text{bal}(pp) \leftarrow 0$. **upout (pp)**
2. $\text{bal}(pp) = 0 \Rightarrow \text{bal}(pp) \leftarrow +1$.
3. $\text{bal}(pp) = +1 \Rightarrow$ **next slides**.

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

upout (p)

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

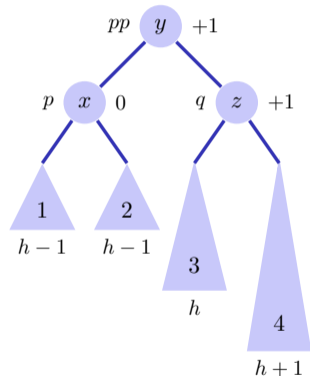
(a).3.1: $\text{bal}(q) = 0$.²⁵



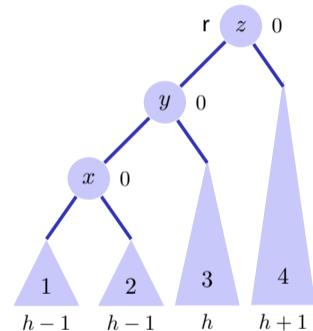
²⁵(b).3.1: $\text{bal}(pp) = -1$, $\text{bal}(q) = -1$, Right rotation

upout (p)

Case (a).3: $\text{bal}(pp) = +1$. (a).3.2: $\text{bal}(q) = +1$.²⁶



\implies
Left Rotate(y)

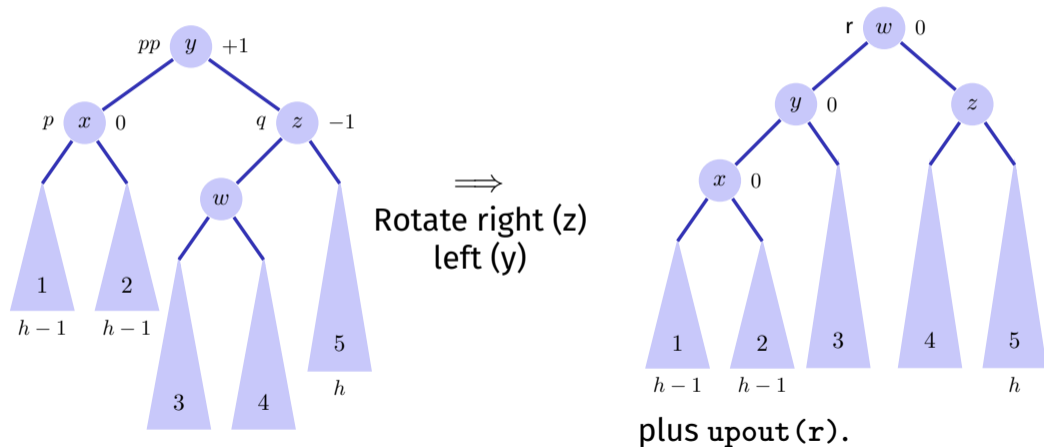


plus upout (r).

²⁶(b).3.2: $\text{bal}(pp) = -1$, $\text{bal}(q) = +1$, Right rotation+upout

upout (p)

Case (a).3: $\text{bal}(pp) = +1$. (a).3.3: $\text{bal}(q) = -1$.²⁷



²⁷(b).3.3: $\text{bal}(pp) = -1$, $\text{bal}(q) = -1$, 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.

18.5 Appendix

Derivation of some mathematical formulas

[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:

$$f(x) \cdot (1 - x - x^2) = x.$$
$$\Leftrightarrow f(x) = \frac{x}{1 - x - x^2} = -\frac{x}{x^2 + x - 1}$$

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

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

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:

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

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

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