10. Repetition Binary Search Trees and Heaps

[Ottman/Widmayer, Kap. 2.3, 5.1, Cormen et al, Kap. 6, 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

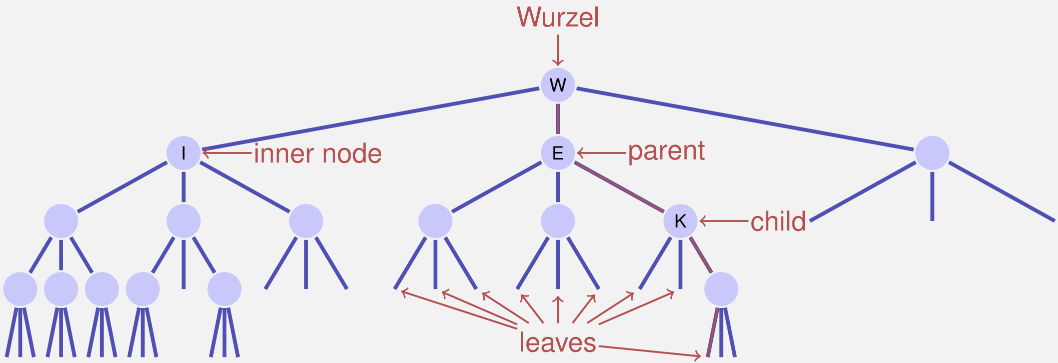
Dictionary implementation

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

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

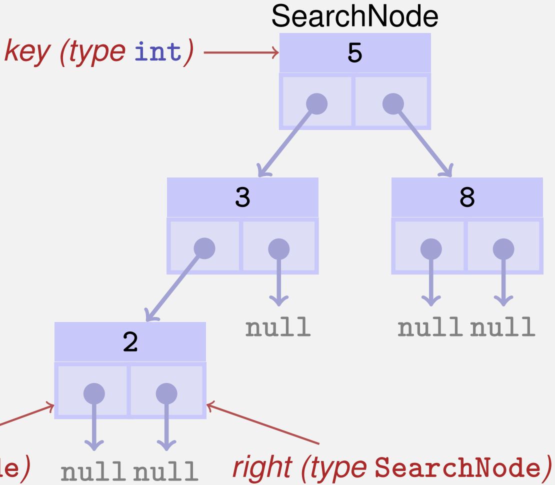
<table><tr><td colspan="2">key</td></tr><tr><td>left</td><td>right</td></tr></table>

- 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

Baumknoten in Java

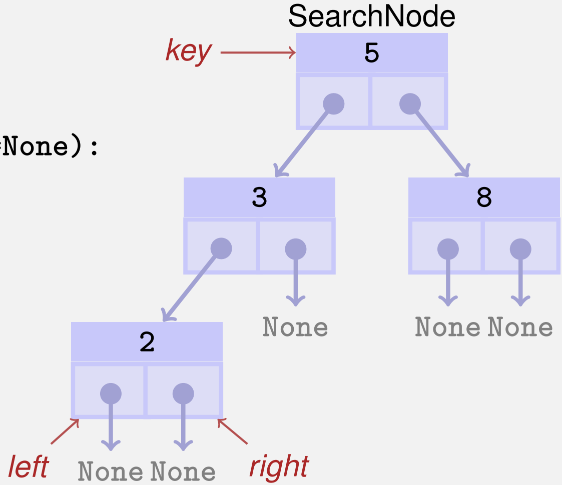
```
public class SearchNode {
  int key;
  SearchNode left;
  SearchNode right;

  SearchNode(int k){
    key = k;
    left = right = null;
  }
}
```



Baumknoten in Python

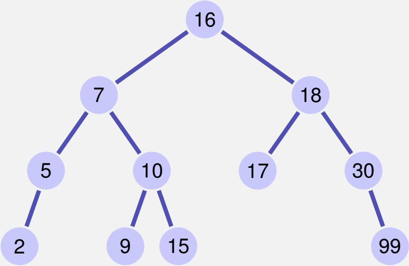
```python
class SearchNode:
  def __init__(self, k, l=None, r=None):
    self.key = k
    self.left, self.right = l, r
    self.flagged = False
```



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.left` are smaller than `v.key`
- Keys in right subtree `v.right` are greater than `v.key`

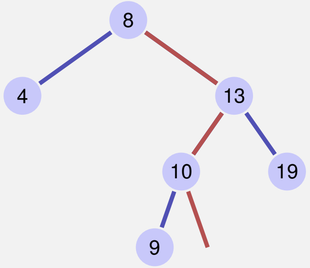


Searching

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

$v \leftarrow r$
while $v \neq$ **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

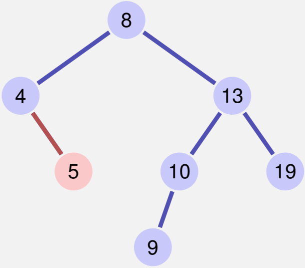


Search (12) $\rightarrow$ **null**

Insertion of a key

Insertion of the key k

- Search for k
- If successful search: output error
- No success: replace the reached leaf by a new node with key



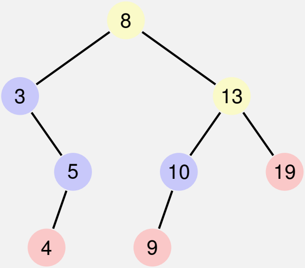
Insert (5)

Remove node

Three cases possible:

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

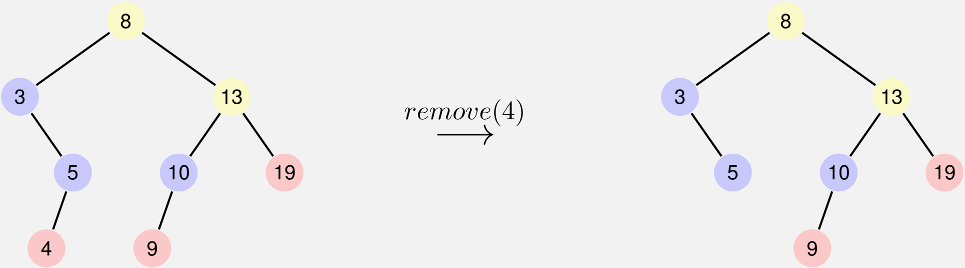
[Leaves do not count here]



Remove node

Node has no children

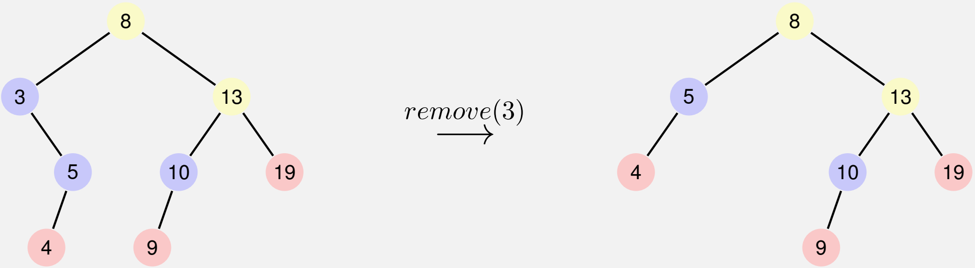
Simple case: replace node by leaf.



Remove node

Node has one child

Also simple: replace node by single child.



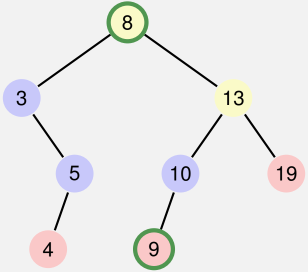
Remove node

Node has two children

The following observation helps: the smallest key in the right subtree `v.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.

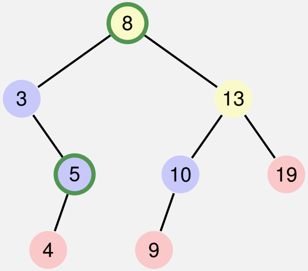
Solution: replace v by its symmetric successor.



By symmetry...

Node has two children

Also possible: replace v by its symmetric predecessor.



Algorithm SymmetricSuccessor(v)

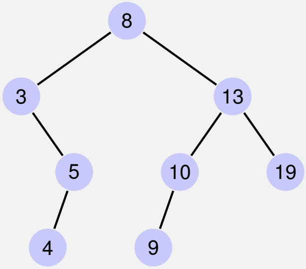
Input: Node v of a binary search tree.
Output: Symmetric successor of v

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

return w

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



Height of a tree

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

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

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

Analysis

Search, Insertion and Deletion of an element v from a tree T requires $\mathcal{O}(h(T))$ fundamental steps in the worst case.

Possible Heights

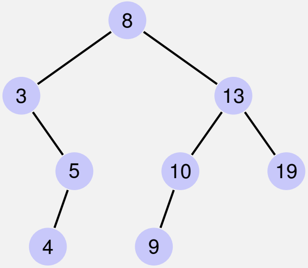
1. The maximal height h_n of a tree with n inner nodes is given with $h_1 = 1$ and $h_{n+1} \leq 1 + h_n$ by $h_n \geq n$.
2. The minimal height h_n of an (ideally balanced) tree with n inner nodes fulfils $n \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1$.

Thus

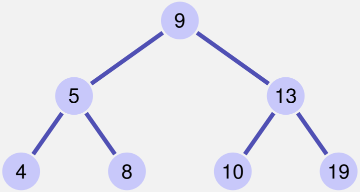
$$\lceil \log_2(n+1) \rceil \leq h \leq n$$

Further supported operations

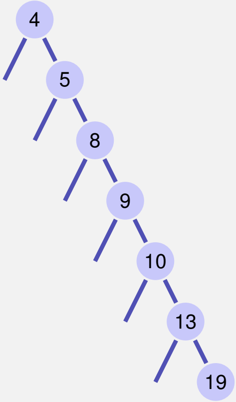
- Min(T): Read-out minimal value in $\mathcal{O}(h)$
- ExtractMin(T): Read-out and remove minimal value in $\mathcal{O}(h)$
- List(T): Output the sorted list of elements
- Join(T_1, T_2): Merge two trees with $\max(T_1) < \min(T_2)$ in $\mathcal{O}(n)$.



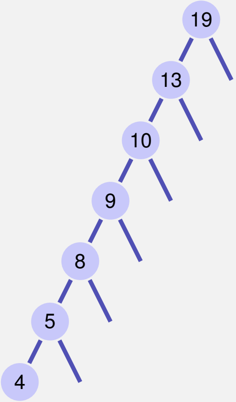
Degenerated search trees



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

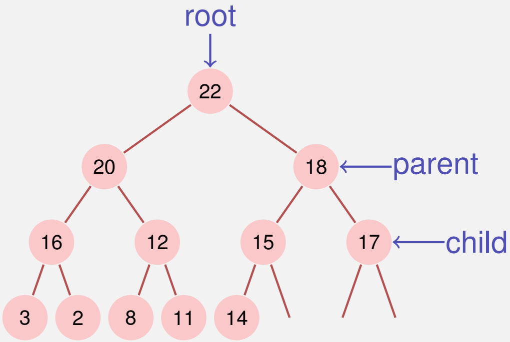
Attention: this only holds for insertions. If the tree is constructed by random insertions and deletions, the expected path length is $\mathcal{O}(\sqrt{n})$.

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.

[Max-]Heap[8]

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) that that of the parent node



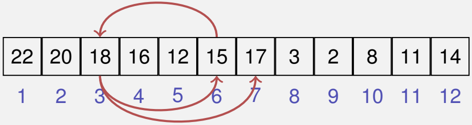
[8]Heap(data structure), not: as in “heap and stack” (memory allocation)

Heap and Array

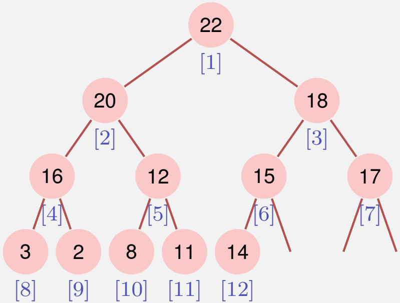
Tree $\rightarrow$ Array:

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

parent



Children



Depends on the starting index[9]

[9]For 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

A complete binary tree with height[10] h provides

$$1 + 2 + 4 + 8 + ... + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

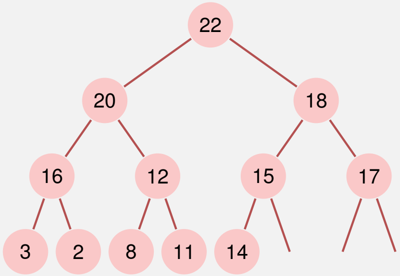
nodes. Thus for a heap with height h:

$$\begin{aligned} & 2^{h-1} - 1 < n \le 2^h - 1 \\ \Leftrightarrow \quad & 2^{h-1} < n + 1 \le 2^h \end{aligned}$$

Particularly $h(n) = \lceil \log_2(n+1) \rceil$ and $h(n) \in \Theta(\log n)$.

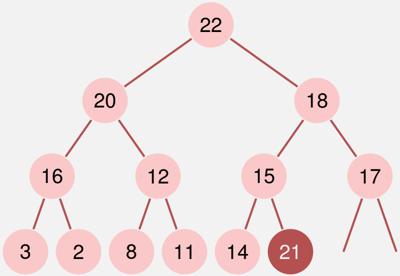
[10] here: number of edges from the root to a leaf

Insert



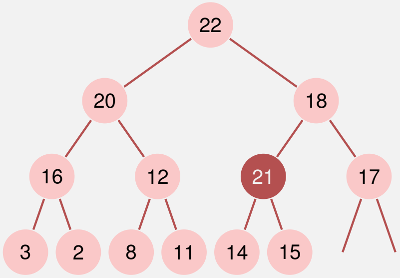
Insert

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



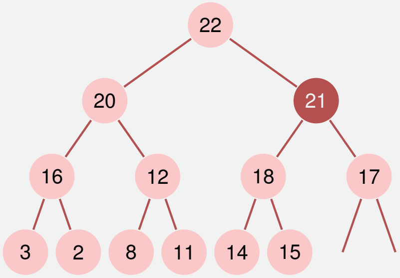
Insert

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



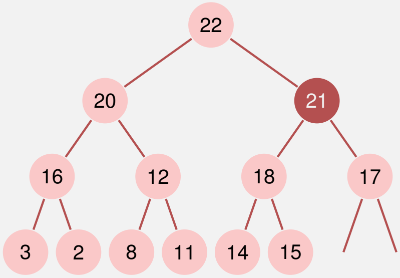
Insert

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

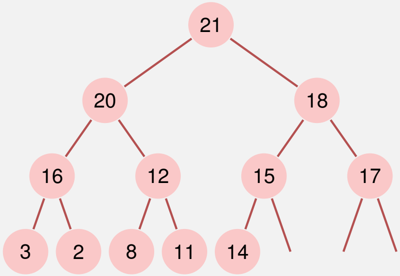


Insert

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

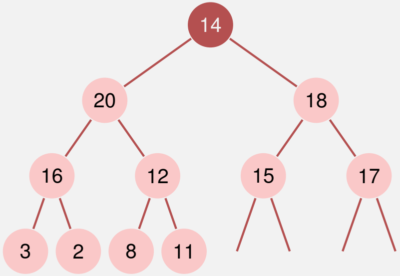


Remove the maximum



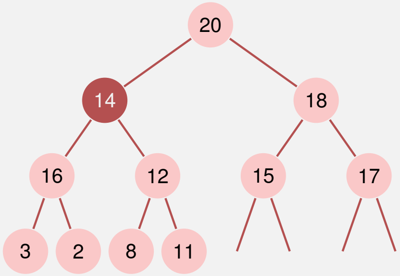
Remove the maximum

- Replace the maximum by the lower right element



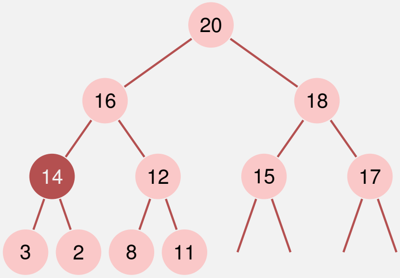
Remove the maximum

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



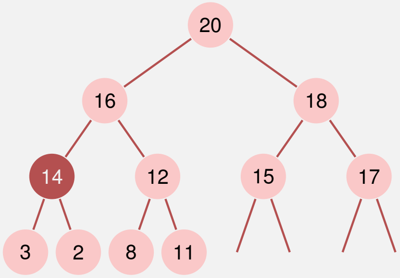
Remove the maximum

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



Remove the maximum

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



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

7	6	4	5	1	2

$A[1, ..., n]$ is a Heap.
While $n > 1$

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

Sort heap

	7	6	4	5	1	2
swap $\Rightarrow$	2	6	4	5	1	7

$A[1, ..., n]$ is a Heap.
While $n > 1$

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

Sort heap

		7	6	4	5	1	2
swap	$\Rightarrow$	2	6	4	5	1	7
siftDown	$\Rightarrow$	6	5	4	2	1	7

$A[1, ..., n]$ is a Heap.
While $n > 1$

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

Sort heap

$A[1, ..., n]$ is a Heap.
While $n > 1$

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

		7	6	4	5	1	2
swap	$\Rightarrow$	2	6	4	5	1	7
siftDown	$\Rightarrow$	6	5	4	2	1	7
swap	$\Rightarrow$	1	5	4	2	6	7

Sort heap

$A[1, ..., n]$ is a Heap.
While $n > 1$

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

		7	6	4	5	1	2
swap	$\Rightarrow$	2	6	4	5	1	7
siftDown	$\Rightarrow$	6	5	4	2	1	7
swap	$\Rightarrow$	1	5	4	2	6	7
siftDown	$\Rightarrow$	5	4	2	1	6	7
swap	$\Rightarrow$	1	4	2	5	6	7
siftDown	$\Rightarrow$	4	1	2	5	6	7
swap	$\Rightarrow$	2	1	4	5	6	7
siftDown	$\Rightarrow$	2	1	4	5	6	7
swap	$\Rightarrow$	1	2	4	5	6	7

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 ← n/2 downto 1 do
    SiftDown(A, i, n);
// Now A is a heap.
for i ← 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: $v(n) \in \mathcal{O}(n \log n)$.

---

[11] $f(x) = \frac{1}{1-x} = 1 + x + x^2 ... \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + ...$

[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:

$$v(n) = \sum_{l=0}^{\lfloor \log n \rfloor} \underbrace{2^l}_{\text{number heaps on level l}} \cdot \underbrace{(\lfloor \log n \rfloor - l)}_{\text{height heaps on level l}} = \sum_{k=0}^{\lfloor \log n \rfloor} 2^{\lfloor \log n \rfloor - k} \cdot k$$

$$\leq \sum_{k=0}^{\lfloor \log n \rfloor} \frac{n}{2^k} \cdot k = n \cdot \sum_{k=0}^{\lfloor \log n \rfloor} \frac{k}{2^k} \in \mathcal{O}(\mathbf{n})$$

with $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \quad (0 < x < 1)$ [11] and $s(\frac{1}{2}) = 2$

[11] $f(x) = \frac{1}{1-x} = 1 + x + x^2 \ldots \Rightarrow f'(x) = \frac{1}{(1-x)^2} = 1 + 2x + \ldots$

11. 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.

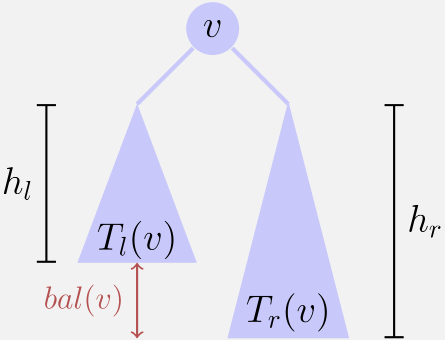
Balancing: guarantee that a tree with n nodes always has a height of $\mathcal{O}(\log n)$.

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

Balance of a node

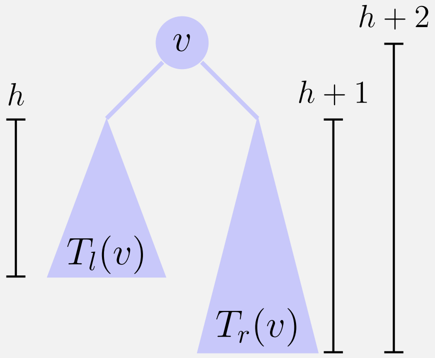
The height *balance* of a node v is defined as the height difference of its sub-trees $T_l(v)$ and $T_r(v)$

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

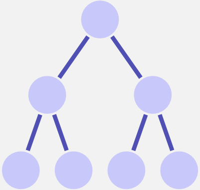


AVL Condition

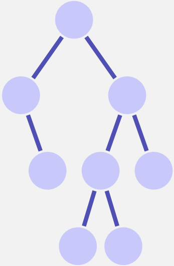
AVL Condition: for eacn node v *of a tree* $\mathrm{bal}(v) \in \{-1, 0, 1\}$



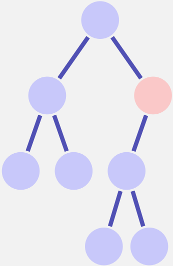
(Counter-)Examples



AVL tree with height 2



AVL tree with height 3

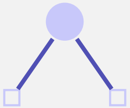


No AVL tree

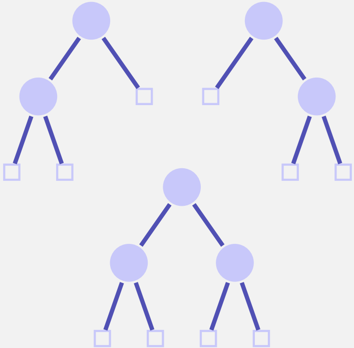
Number of Leaves

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



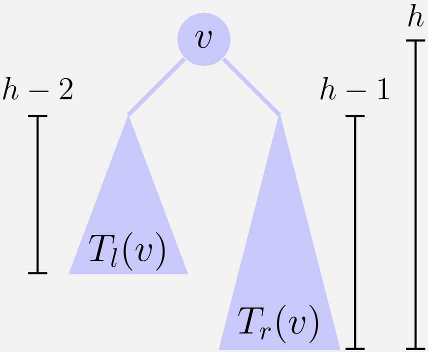
AVL tree with height 2 has
at least $N(2) := 3$ leaves.

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

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



Overal 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 that

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

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

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

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

[Fibonacci Numbers, Inductive Proof]

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

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

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

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

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

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

and thus

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

An AVL tree is asymptotically not more than 44% higher than a perfectly balanced tree.[12]

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

Insertion

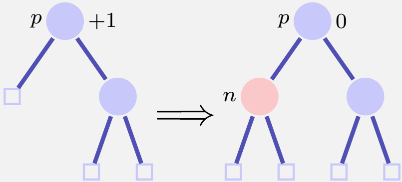
Balance

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

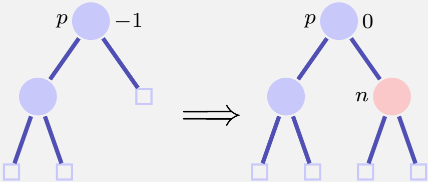
New node n is inserted:

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

Balance at Insertion Point



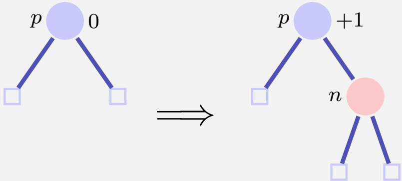
case 1: $\mathrm{bal}(p) = +1$



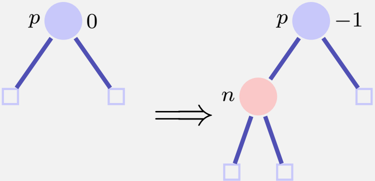
case 2: $\mathrm{bal}(p) = -1$

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)`

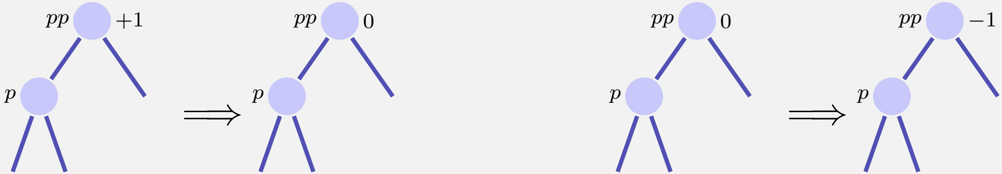
upin(p) - invariant

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[13]



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

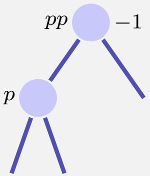
case 2: $\mathrm{bal}(pp) = 0$, `upin(pp)`

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

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

upin(p)

Assumption: p is left son of pp



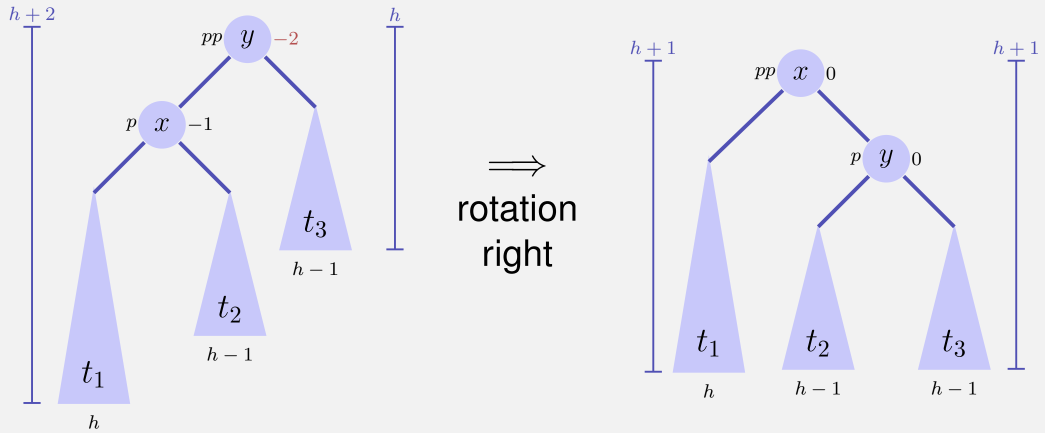
case 3: $\mathrm{bal}(pp) = -1$,

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

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

Rotations

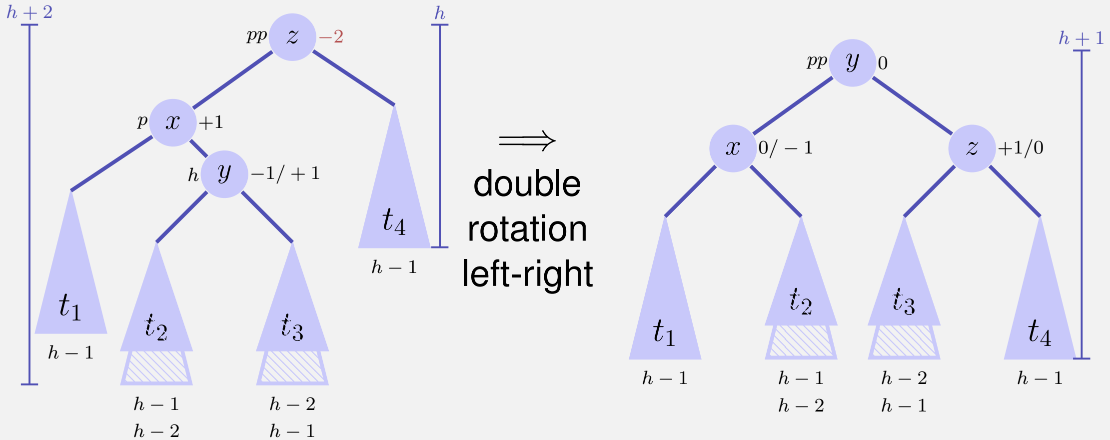
case 1.1 $\mathrm{bal}(p) = -1$. [14]



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

Rotations

case 1.1 $\text{bal}(p) = -1.$ [15]



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

Analysis

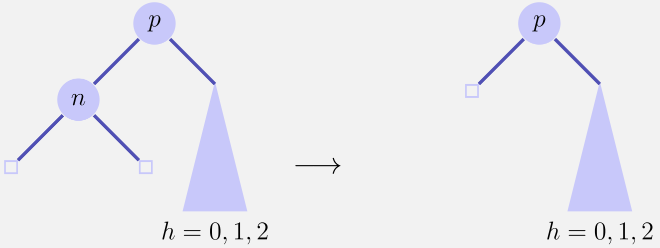
- Tree height: $\mathcal{O}(\log n)$.
- Insertion like in binary search tree.
- Balancing via recursion from node to the root. Maximal path lenght $\mathcal{O}(\log n)$.

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.

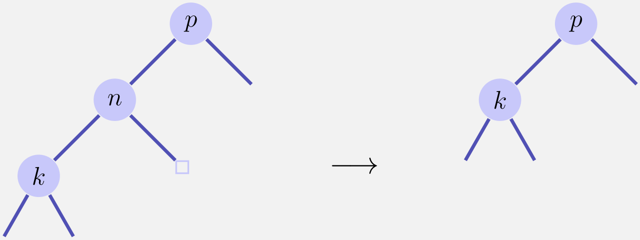
- $h' = 1$: Adapt $\text{bal}(p)$.
- $h' = 0$: Adapt $\text{bal}(p)$. Call `upout(p)`.
- $h' = 2$: Rebalanciere des Teilbaumes. Call `upout(p)`.



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

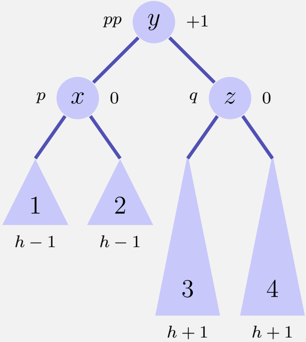
1. $\mathrm{bal}(pp) = -1 \;\Rightarrow\; \mathrm{bal}(pp) \leftarrow 0$. `upout(pp)`
2. $\mathrm{bal}(pp) = 0 \;\Rightarrow\; \mathrm{bal}(pp) \leftarrow +1$.
3. $\mathrm{bal}(pp) = +1 \Rightarrow$ next slides.

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

upout(p)

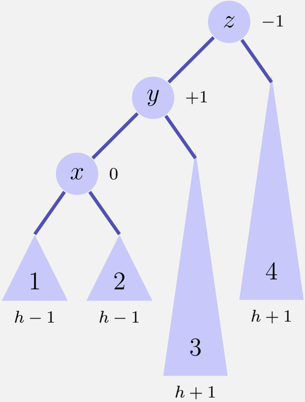
Case (a).3: $\mathrm{bal}(pp) = +1$. Let q be brother of p

(a).3.1: $\mathrm{bal}(q) = 0$.[16]



$\Longrightarrow$

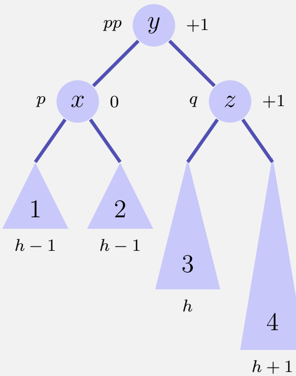
Left Rotate(y)



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

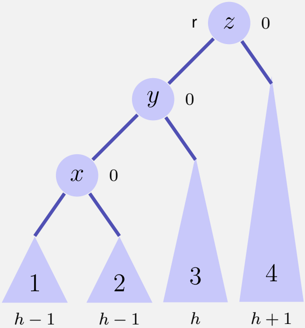
upout(p)

Case (a).3: $\mathrm{bal}(pp) = +1$. (a).3.2: $\mathrm{bal}(q) = +1$.[17]



$\Longrightarrow$

Left Rotate(y)



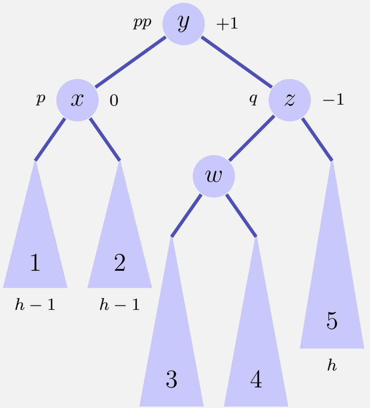
plus `upout(r)`.

---

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

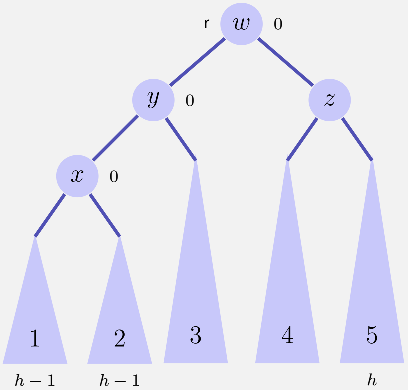
`upout(p)`

Case (a).3: $\mathrm{bal}(pp) = +1$. (a).3.3: $\mathrm{bal}(q) = -1$.[18]



⟹

Rotate right (z) left (y)



plus `upout(r)`.

[18] (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.