8. Binary Search Trees

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

Trees

Trees are

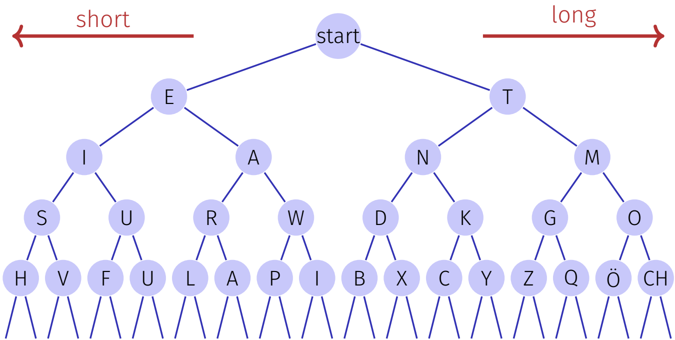
- Generalized lists: nodes can have more than one successor
- Special graphs: graphs consist of nodes and edges. A tree is a fully connected, directed, acyclic graph.

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

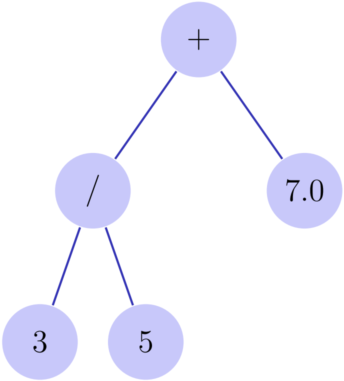
Examples



Morsealphabet

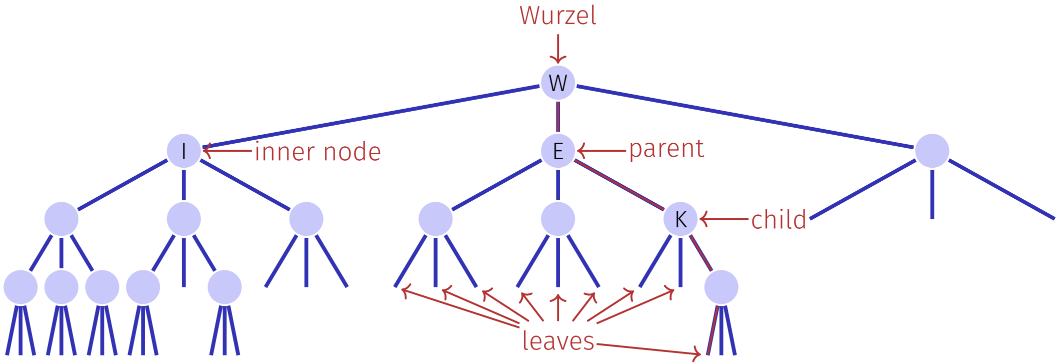
Examples

$$3/5 + 7.0$$



Expression tree

Nomenclature



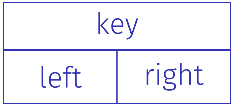
- Order of the tree: maximum number of child nodes, here: 3
- Height of the tree: maximum path length root – leaf (here: 4)

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.

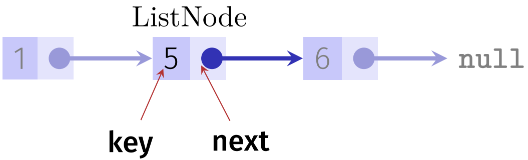
In each inner node **`v`** we store



- 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

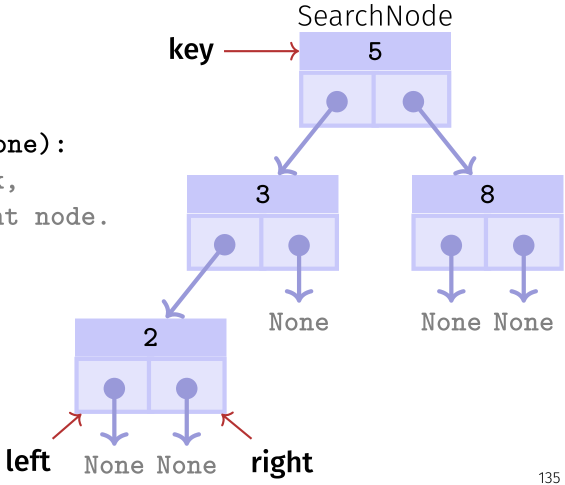
Linked List Node in Python




```
class ListNode:
   # entries key, next implicit via constructor

  def __init__(self, key , next = None):
    """Constructor that takes a key and, optionally, next."""
    self.key = key
    self.next = next
}
```

Now: tree nodes in Python

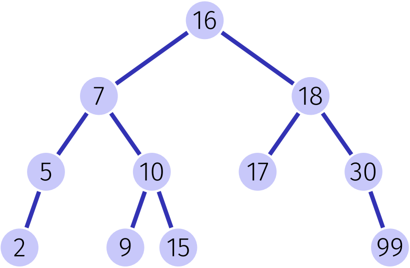
```
class SearchNode:
# implicit entries key, left, right

  def __init__(self, k, l=None, r=None):
    # Constructor that takes a key k,
    # and optionally a left and right node.
    self.key = k
    self.left, self.right = l, r
```



Binary search tree

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

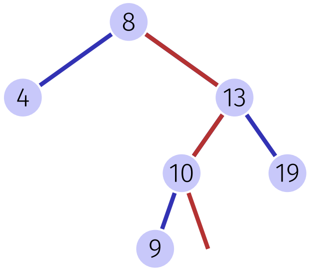
- Every node `v` stores a key
- Keys in left subtree `v.left` are smaller than `v.key`
- Keys in right subtree `v.right` are greater than `v.key`



Searching

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

Searching in Python

```python
def findNode(root, key):
    n = root
    while n != None and n.key != key:
        if key < n.key:
            n = n.left
        else:
            n = n.right
    return n
```

Height of a tree

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

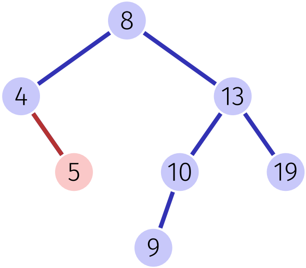
$$h(r) = \begin{cases} 0 & \text{if } r = \textbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{otherwise.} \end{cases}$$

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



Insert (5)

Insert Nodes in Python

```python
def addNode(root, key):
  n = root
  if n == None:
    root = Node(key)
  while n.key != key:
    if key < n.key:
      if n.left == None:
        n.left = Node(key)
      n = n.left
    else:
      if n.right == None:
        n.right = Node(key)
      n = n.right
  return root
```

Tree in Python

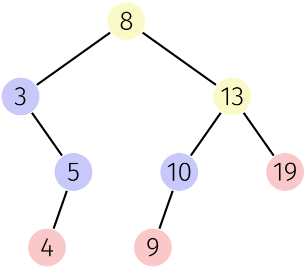
```python
class Tree:
    def __init__(self):
        self.root = None

    def find(self,key):
        return findNode(self.root, key)

    def has(self,key):
        return self.find(key) != None

    def add(self,key):
        self.root = addNode(self.root, key)

    # ....
```

Remove node

Three cases possible:

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

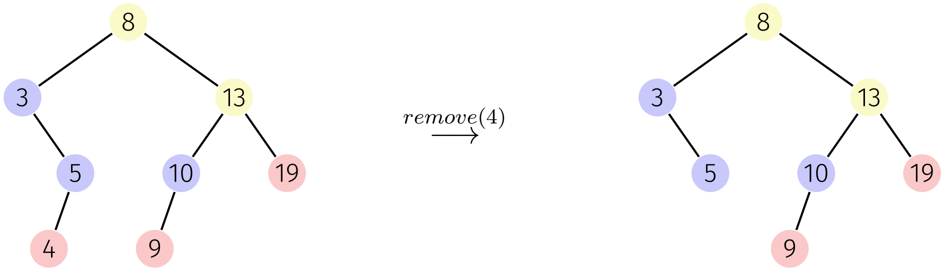
[Leaves do not count here]



Remove node

Node has no children

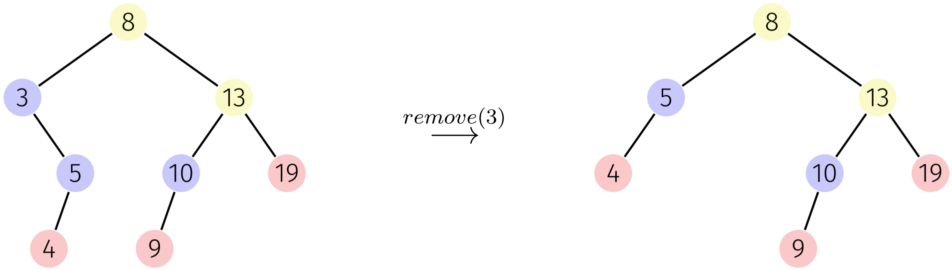
Simple case: replace node by leaf.



Remove node

Node has one child

Also simple: replace node by single child.



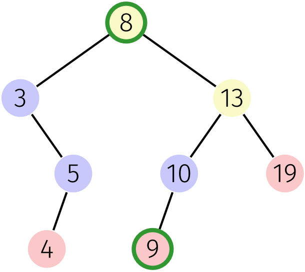
Remove node

Node `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.

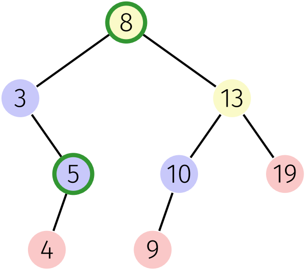


By symmetry...

Node `v` has two children

Also possible: replace `v` by its symmetric predecessor.

Implementation: devil is in the detail!



Algorithm SymmetricSuccessor(v)

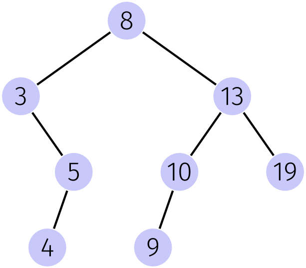
Input: Node v of a binary search tree.
Output: Symmetric successor of v

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

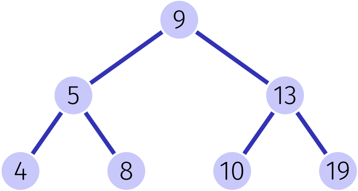
return w

Traversal possibilities

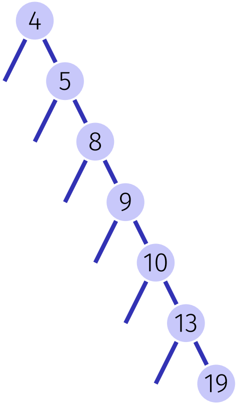
- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- inorder: $T_{\text{left}}(v)$, then v, then $T_{\text{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19



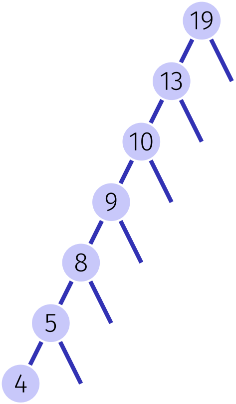
Degenerated search trees



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.

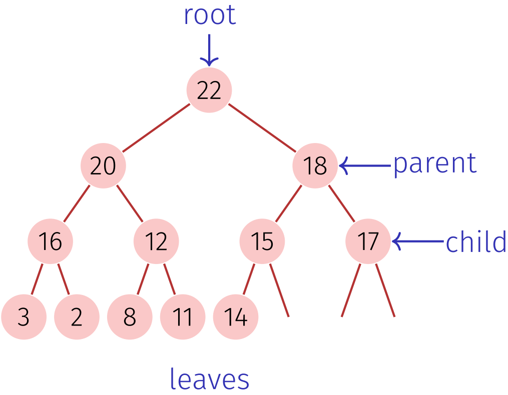
9. Heaps

Datenstruktur optimiert zum schnellen Extrahieren von Minimum oder Maximum und Sortieren. [Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

[Max-]Heap*

Binary tree with the following proper-ties

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



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

Heap as Array

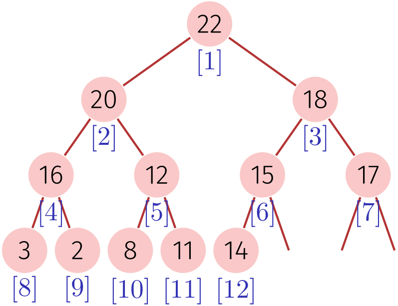
Tree $\rightarrow$ Array:

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

parent

22	20	18	16	12	15	17	3	2	8	11	14
1	2	3	4	5	6	7	8	9	10	11	12

Children



Depends on the starting index[4]

[4] 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

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

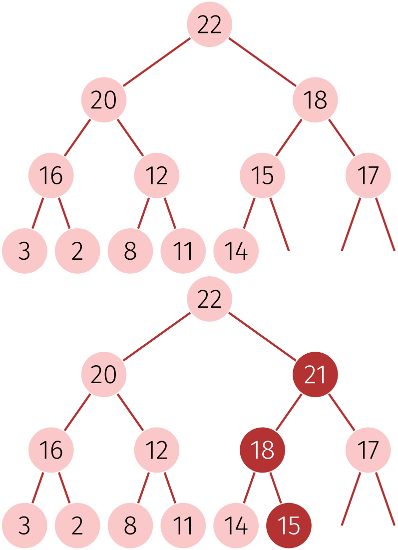
$$H(n) = \min\{h \in \mathbb{N} : 2^h \geq n + 1\},$$

thus

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

Insert

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



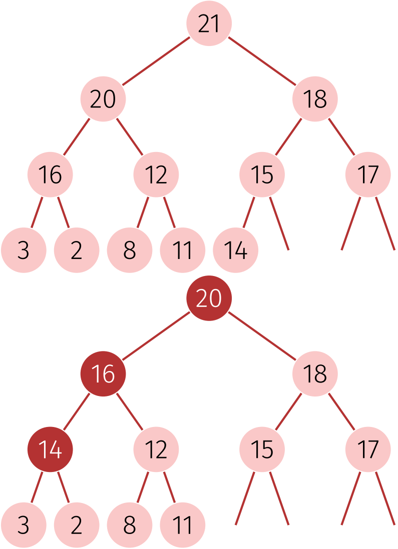
Algorithm Sift-Up(A, m)

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

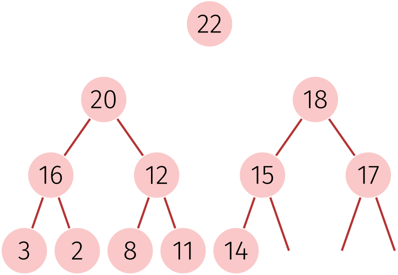
Remove the maximum

- 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:



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

$A[1, ..., n]$ 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: 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: $v(n) \in \mathcal{O}(n \log n)$.

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 $n > 0$

$$v(n) = \sum_{l=0}^{\lfloor \log_2 n \rfloor} \underbrace{2^l}_{\text{number heaps on level l}} \cdot (\underbrace{\lfloor \log_2 n \rfloor + 1 - l}_{\text{height heaps on level l}} - 1) = \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^{\lfloor \log_2 n \rfloor - k} \cdot k$$

$$= 2^{\lfloor \log_2 n \rfloor} \cdot \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k} \leq n \cdot \sum_{k=0}^{\infty} \frac{k}{2^k} \leq n \cdot 2 \in \mathcal{O}(n)$$

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