18. Binary Search Trees

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

Trees

Trees are

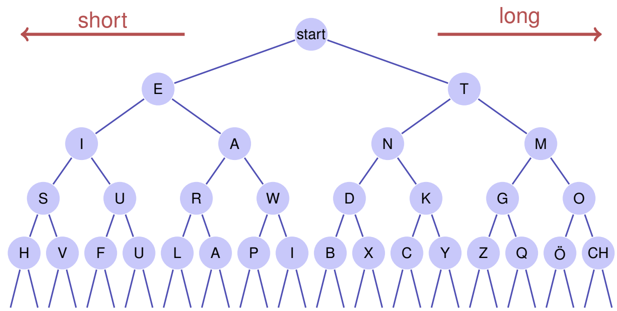
- Generalized lists: nodes can have more than one successor
- Special graphs: graphs consist of nodes and edges. A tree is a fully connected, directed, acyclic graph.

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

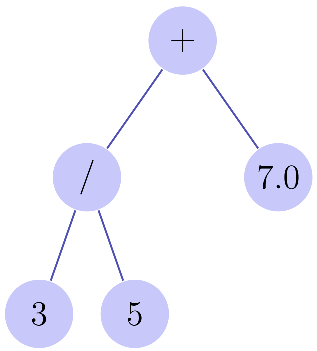
Examples



Morsealphabet

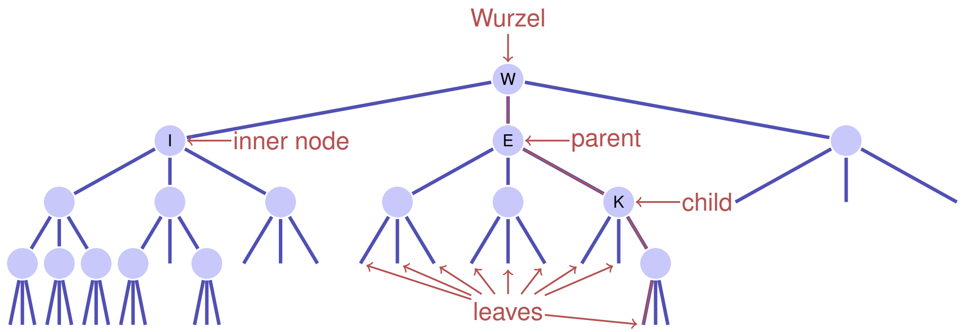
Examples

$3/5 + 7.0$



Expression tree

Nomenclature



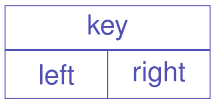
- Order of the tree: maximum number of child nodes, here: 3
- Height of the tree: maximum path length root – leaf (here: 4)

Binary Trees

A binary tree is either

- 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.

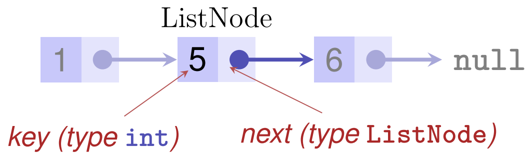
In each node v we store



- 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

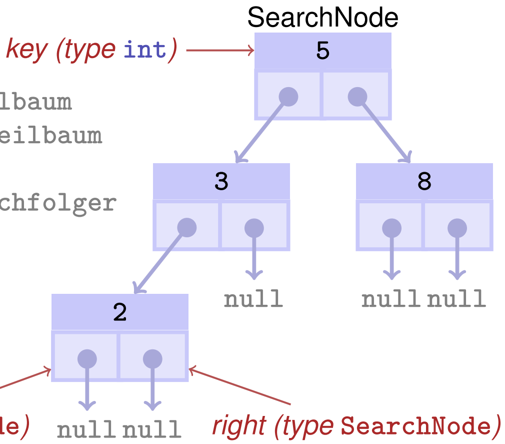
Recall: Linked List Node in Java




```
class ListNode {
  int key;
  ListNode next;

  ListNode (int key, ListNode next){
    this.key = key;
    this.next = next;
  }
}
```

Now: tree nodes in Java

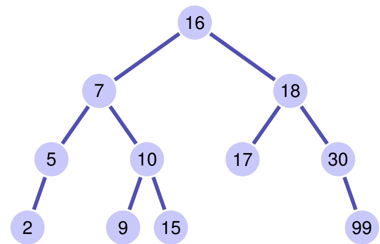
```
public class SearchNode {
  int key;          // Schluessel
  SearchNode left; // linker Teilbaum
  SearchNode right; // rechter Teilbaum

  // Konstruktor: Knoten ohne Nachfolger
  SearchNode(int k){
    key = k;
    left = right = null;
  }
}
```



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`

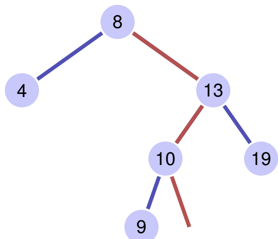


Searching

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) → **null**

Search Tree and Searching in Java

```
public class SearchTree {
  SearchNode root = null; // Wurzelknoten

  // Gibt zurueck, ob Knoten mit Schluessel k existiert
  public boolean contains (int k){
    SearchNode n = root;
    while (n != null && n.key != k){
      if (k < n.key) n = n.left;
      else n = n.right;
    }
    return n != null;
  }
  ... // Einfuegen, Loeschen
}
```

Height of a tree

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

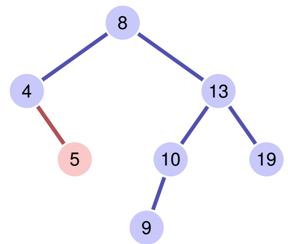
$$h(r) = \begin{cases} 0 & \text{if } r = \textbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{otherwise.} \end{cases}$$

The worst case search time is determined by the height of the tree.

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
- Implementation: devil is in the detail



Insert (5)

Knoten Einfügen in Java

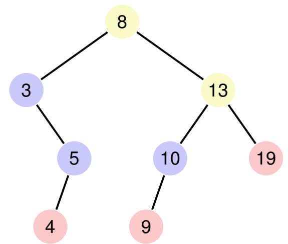
```
public boolean add (int k) {
  if (root == null) {root = new SearchNode(k); return true;}
  SearchNode t=root;
  while (k != t.key) {
    if (k < t.key) {
      if (t.left == null){ t.left = new SearchNode(k); return true;}
      else { t = t.left; }
    }
    else { // k > t.key
      if (t.right == null){ t.right = new SearchNode(k); return true;}
      else { t = t.right; }
    }
  }
  return false;
}
```


Remove node

Three cases possible:

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

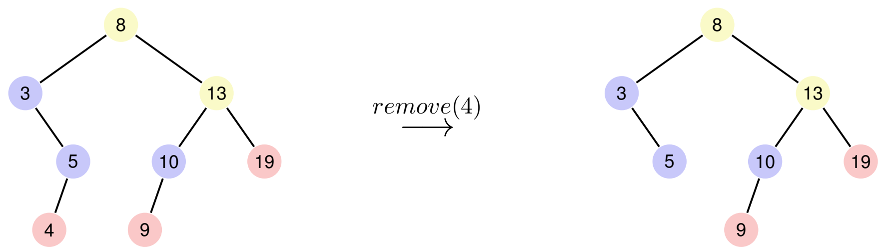
[Leaves do not count here]



Remove node

Node has no children

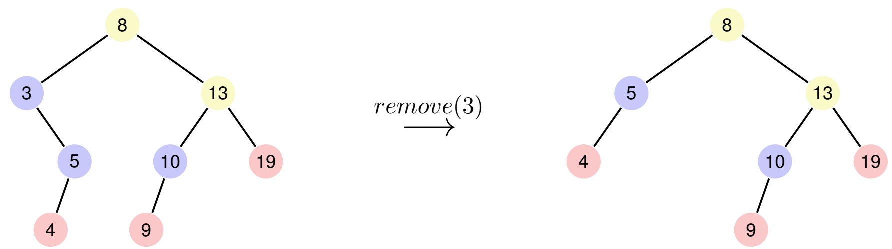
Simple case: replace node by leaf.



Remove node

Node has one child

Also simple: replace node by single child.



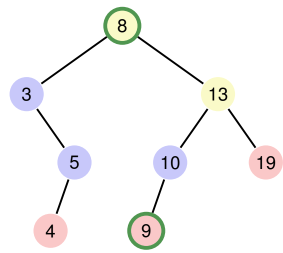
Remove node

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.

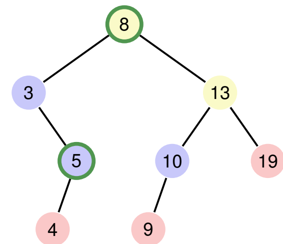


By symmetry...

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

SymmetricDesc in Java

```
public SearchNode symmetricDesc(SearchNode node) {
  if (node.left == null) { return node.right; }
  if (node.right == null) { return node.left; }
  SearchNode n = node.right; // cannot be null
  SearchNode parent = null;
  while (n.left != null) { parent = n; n = n.left; }
  if (parent != null){
    parent.left = n.right;
    n.right = node.right;
  } // else n == node.right
  n.left = node.left;
  return n;
}
```

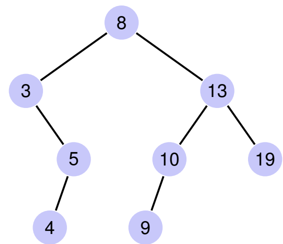
This algorithm returns the symmetric descendent. But it does even more: it handles also all cases with one or no descendent. And it replaces the symmetric descendent by its successor.

Knoten Löschen in Java

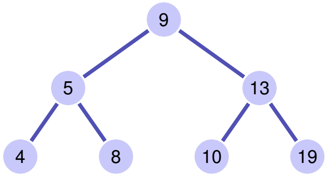
```
public boolean remove (int k) {
  SearchNode n = root;
  if (n != null && n.key == k) {
    root = SymmetricDesc(root); return true;
  }
  while (n != null) {
    if (n.left != null && k == n.left.key) {
      n.left = SymmetricDesc(n.left); return true;
    } else if (n.right != null && k == n.right.key) {
      n.right = SymmetricDesc(n.right); return true;
    } else if (k < n.key) { n = n.left;
    } else { n = n.right; }
  }
  return false;
}
```


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

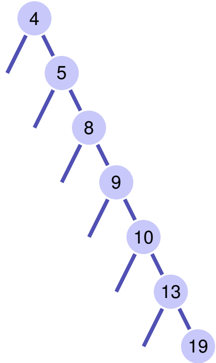


Degenerated search trees



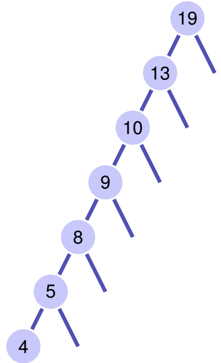
Insert 9,5,13,4,8,10,19

ideally balanced



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

linear list



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

linear list

Efficiency Considerations

Obviously the runtime of the algorithms search, insert and delete depend in the worst case on the heigh of the tree.

Degenerated trees are in the worst case thus not better than a linked list

Balanced trees make sure (e.g. with *rotations*) during insertion or deletion that the tree stays balanced and provide certain guarantees for the algorithms.

The data structures `TreeSet` and `TreeMap` in Java are implemented with balanced trees (so called Red-Black-Trees).

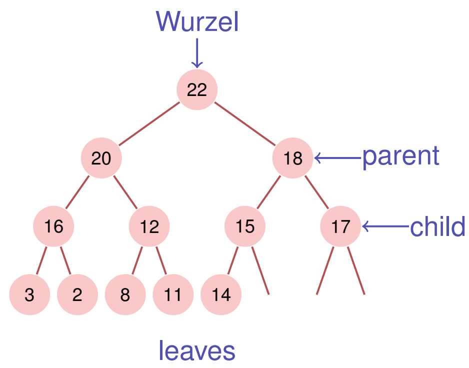
19. Heaps

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

[Max-]Heap[9]

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) thant that of the parent node

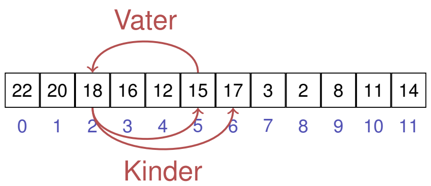


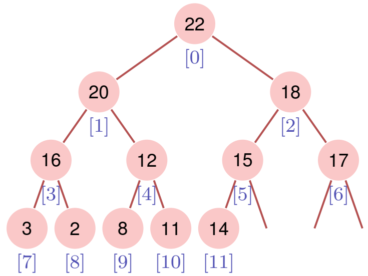
[9] Heap(data structure), not: as in "heap and stack" (memory allocation)

Heap as Array

Tree $\to$ Array:

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





Depends on the starting index[10]

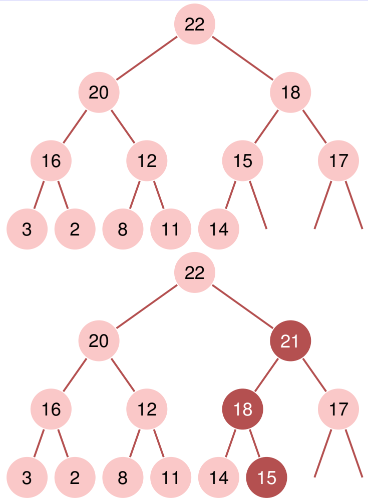
[10] For array that start at 1: $\{2i+1, 2i+2\} \to \{2i, 2i+1\}$, $\lfloor (i-1)/2 \rfloor \to \lfloor i/2 \rfloor$

Heap in Java

```
public class Heap {
  double[] A;// will need to grow
  int sz;
  // Heap initialized with 16 elements
  Heap () {
    A = new double[16]; sz = 0;
  }
  // binary growth of the array
  void grow(){ ... }
  // insert element in the heap
  public void add(double value){...}
  // extract and return first (maximal) element
  public double remove() {...}
}
```


Insert

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



Algorithm Sift-Up(A, m)

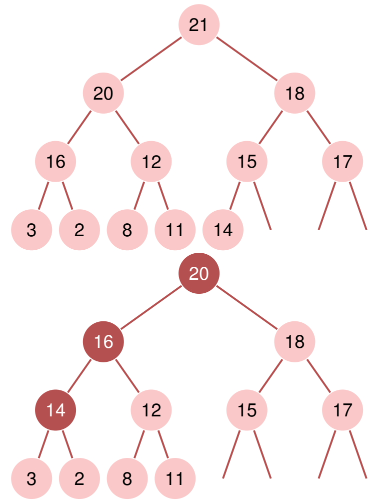
Input: Array A with at least $m+1$ and Max-Heap-Structure on $A[0, \ldots, m-1]$
Output: Array A with Max-Heap-Structure on $A[0, \ldots, m]$.
$v \leftarrow A[m]$ // value
$c \leftarrow m$ // current position
$p \leftarrow \lfloor (c-1)/2 \rfloor$ // parent node
while $c > 0$ and $v > A[p]$ **do**
  $A[c] \leftarrow A[p]$ // Value parent node $\to$ current node
  $c \leftarrow p$ // parent node $\to$ current node
  $p \leftarrow \lfloor (c-1)/2 \rfloor$
$A[c] \leftarrow v$ // value $\to$ current node

add

```
// insert element to the heap
public void add(double value){
  if (sz == A.length){ grow(); }
  int current = sz;
  int parent = (current−1)/2;
  // sift value up
  while (current > 0 && value > A[parent]) {
    A[current] = A[parent];
    current = parent;
    parent = (current−1)/2;
  }
  A[current] = value;
  sz++;
}
```

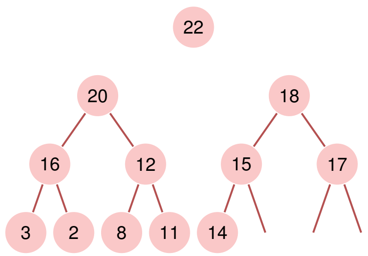

Remove the maximum

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



Why this is correct: Recursive heap structure

A heap consists of two heaps:



Algorithm Sift-down(A, i, m)

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

while $2i + 1 \leq m$ **do**
- $j \leftarrow 2i + 1$; // 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
- **else**
  - $i \leftarrow m$; // sinking finished

remove

```
public double remove() {
  double max = A[0];
  double value = A[sz--];
  int i = 0; int j = 0;
  // sift value down
  while (2*i+1 < sz){
    j = 2*i+1; // left child
    if (j < sz-1 && A[j] < A[j+1]){ ++j;} // right key greater
    if (value < A[j]){ // heap condition still violated
      A[i] = A[j]; i = j; // sift down
    } else { i = sz; } // finished
  }
  A[j] = value;
  return max;
}
```


Sort heap

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

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

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

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

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

Runtime of the Heap-Algorithms

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

The algorithms insert and extrat therefore make about $\log_2(n+1)$ "Steps".[11]

That makes the heap a very fast data structure because the logarithm grows only very slowy. It is used for sorting data and to implement priority Queues.

[11] will be made more precise in Computer Science II.