9. Binary Search Trees

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

Trees

Trees are

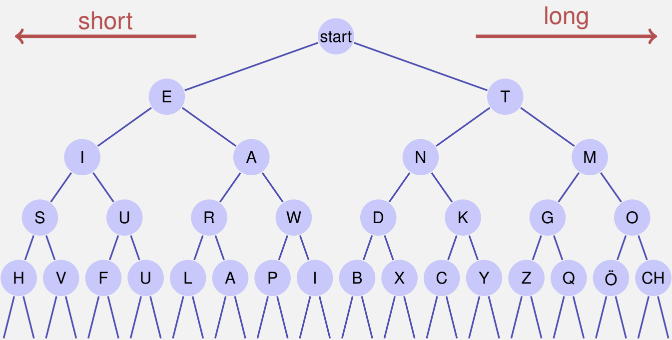
- Generalized lists: nodes can have more than one successor
- Special graphs: graphs consist of nodes and edges. A tree is a fully connected, directed, acyclic graph.

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

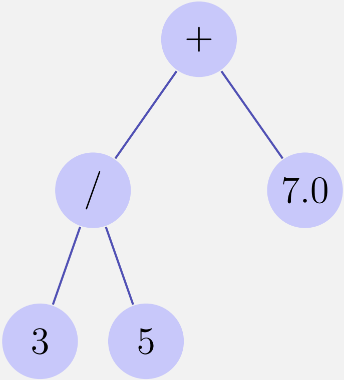
Examples



Morsealphabet

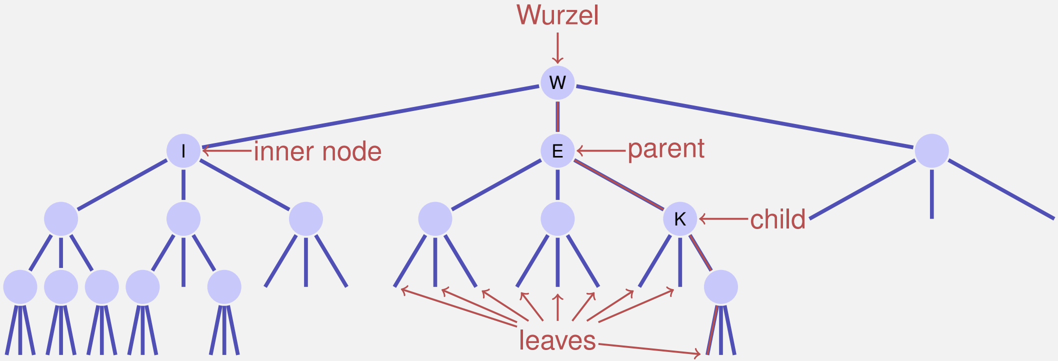
Examples

$$3/5 + 7.0$$



Expression tree

Nomenclature



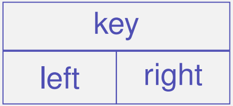
- Order of the tree: maximum number of child nodes, here: 3
- Height of the tree: maximum path length root – leaf (here: 4)

Binary Trees

A binary tree is either

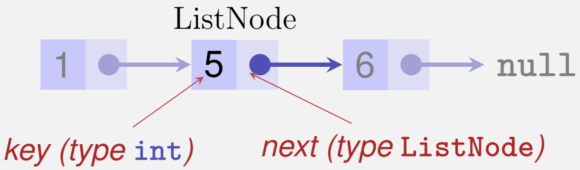
- a leaf, i.e. an empty tree, or
- an inner leaf with two trees T_l (left subtree) and T_r (right subtree) as left and right successor.

In each 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




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

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

Baumknoten in Java

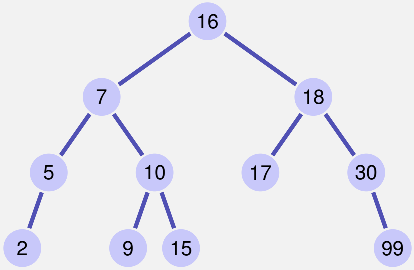
```
public class SearchNode {
  int key;      // Schluessel
  SearchNode left;    // linker Teilbaum
  SearchNode right;   // rechter Teilbaum

  // 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 the left subtree $v.\mathrm{left}$ of v are smaller than $v.\mathrm{key}$
- Key in the right subtree $v.\mathrm{right}$ of v are larger than $v.\mathrm{key}$

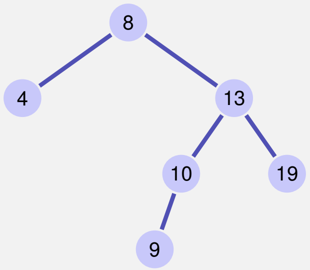


Searching

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

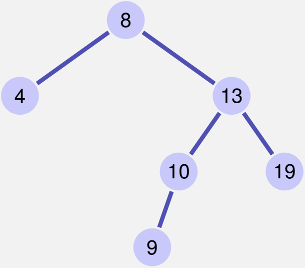


Searching

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

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

return null



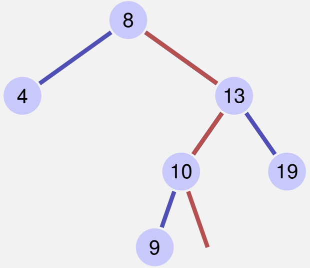
Search (12)

Searching

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

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

return null

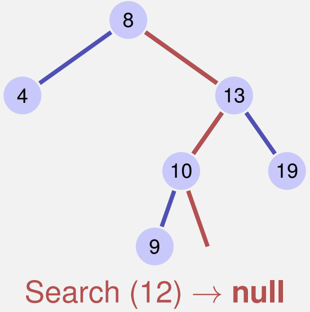


Search (12)

Searching

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

```
v ← r
while v ≠ null do
    if k = v.key then
        return v
    else if k < v.key then
        v ← v.left
    else
        v ← v.right
return null
```



Suchbaum und Suchen in Java

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

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

Height of a tree

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

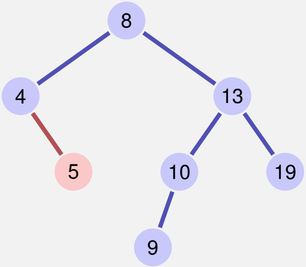
$$h(r) = \begin{cases} 0 & \text{if } r = \textbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{otherwise.} \end{cases}$$

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

Insertion of a key

Insertion of the key k

- Search for k
- If successful search: output error
- Of no success: insert the key at the leaf reached
- Implementation: devil is in the detail



Insert (5)

Knoten Einfügen in Java

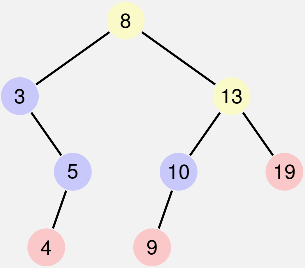
```
public SearchNode Insert (int k) {
  if (root == null) { return root = new SearchNode(k); }
  SearchNode t=root;
  while (true) {
    if (k == t.key) { return null; }
    if (k < t.key) {
      if (t.left == null) { return t.left = new SearchNode(k); }
      else { t = t.left; }
    }
    else {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

Remove node

Three cases possible:

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

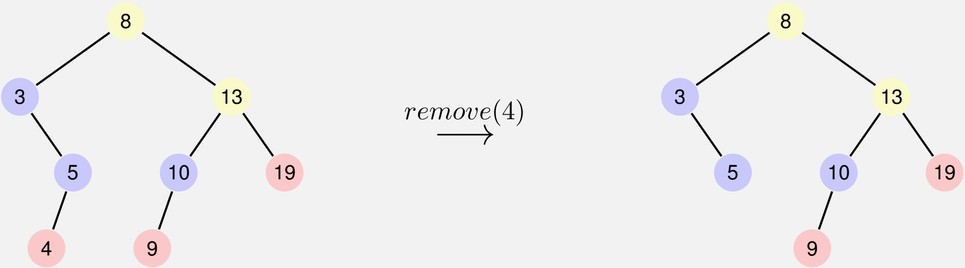
[Leaves do not count here]



Remove node

Node has no children

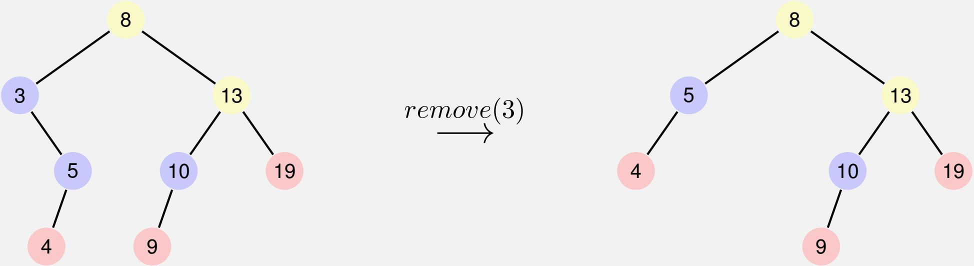
Simple case: replace node by leaf.



Remove node

Node has one child

Also simple: replace node by single child.



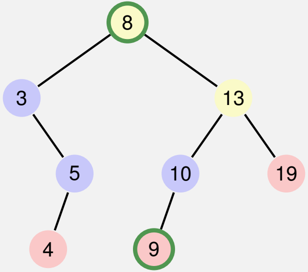
Remove node

Node has two children

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

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

Solution: replace v by its symmetric successor.

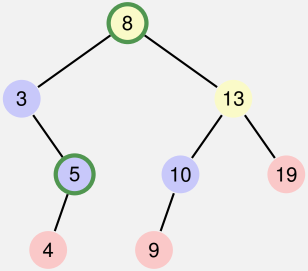


By symmetry...

Node has two children

Also possible: replace v by its symmetric predecessor.

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;
  SearchNode parent = null;
  n = n.right;
  while (n.left != null) { parent = n; n = n.left; }
  if (parent != null) {
    parent.left = n.right;
    n.left = node.left;
    n.right = node.right;
  } else { n.left = node.left; }
  return n;
}
```

Dieser Algorithmus gibt den symmetrischen Nachfolger zurück. Aber tut noch mehr: er behandelt auch die Fälle mit einem oder keinem Nachfolger. Ausserdem entfernt er den Symmetrischen Nachfolger und setzt dessen Nachfolgeknoten.

Knoten Löschen in Java

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

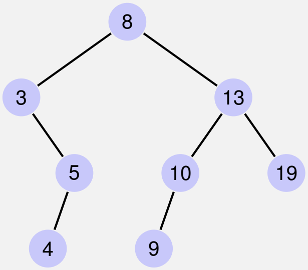
Analysis

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

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

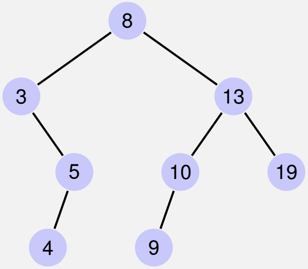
Traversal possibilities

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



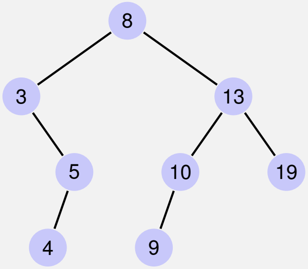
Traversal possibilities

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



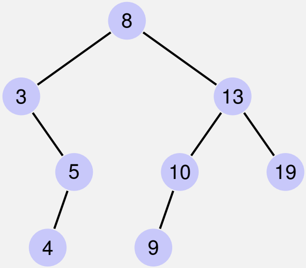
Traversal possibilities

- 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.



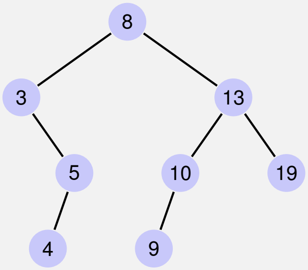
Traversal possibilities

- 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



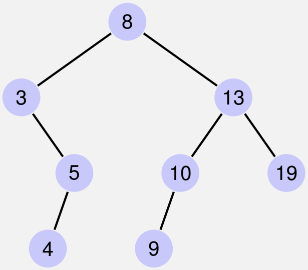
Traversal possibilities

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

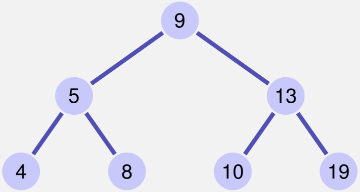


Traversal possibilities

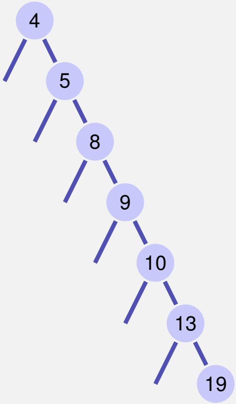
- preorder: v, then $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- postorder: $T_{\text{left}}(v)$, then $T_{\text{right}}(v)$, then v.
  4, 5, 3, 9, 10, 19, 13, 8
- inorder: $T_{\text{left}}(v)$, then v, then $T_{\text{right}}(v)$.
  3, 4, 5, 8, 9, 10, 13, 19



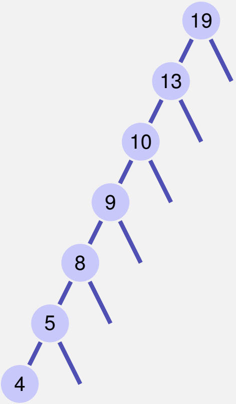
Degenerated search trees



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

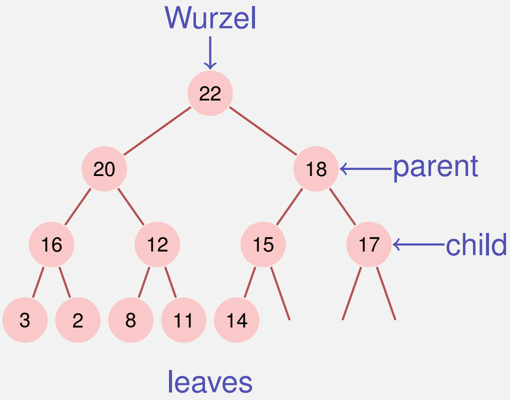
Balanced trees make sure (e.g. with *rotations*) during insertion or deletion that the tree stays balanced and provide a $\mathcal{O}(\log n)$ Worst-case guarantee.

10. Heaps

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

[Max-]Heap[5]

Binary tree with the following properties

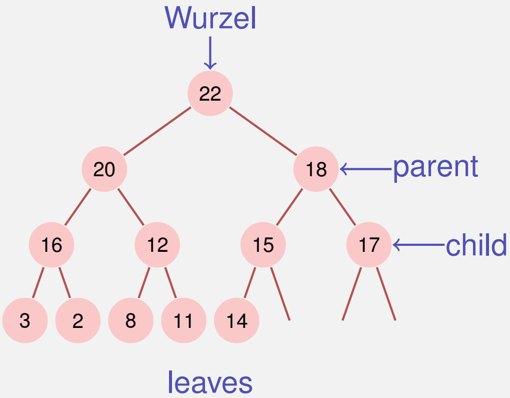


[5]Heap(data structure), not: as in “heap and stack” (memory allocation)

[Max-]Heap[5]

Binary tree with the following properties

1. complete up to the lowest level

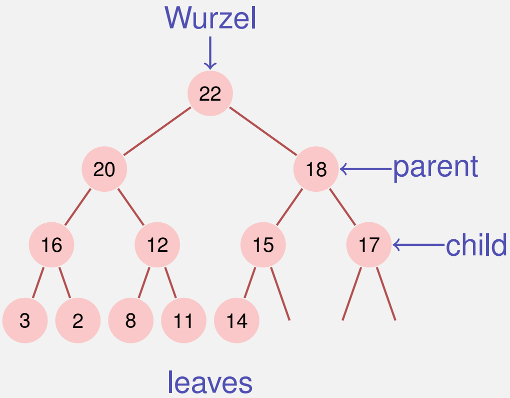


[5] Heap(data structure), not: as in "heap and stack" (memory allocation)

[Max-]Heap[5]

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

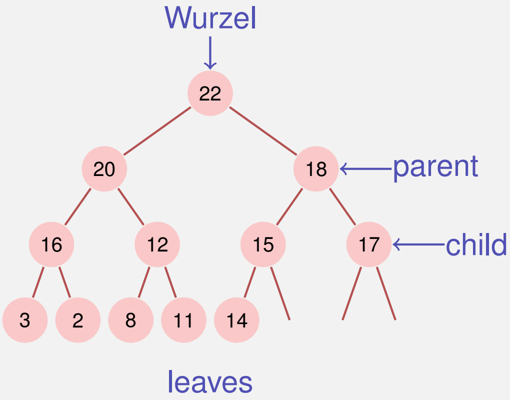


[5] Heap(data structure), not: as in “heap and stack” (memory allocation)

[Max-]Heap[5]

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

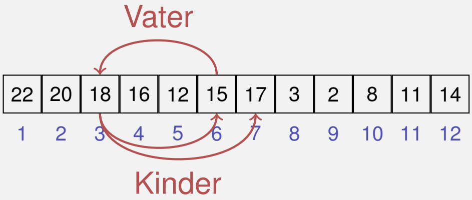


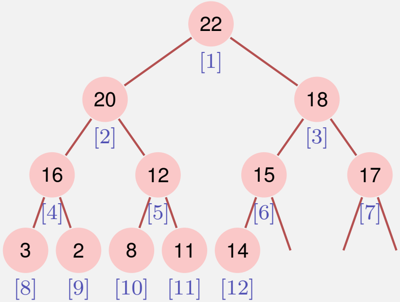
[5]Heap(data structure), not: as in “heap and stack” (memory allocation)

Heap and Array

Tree → Array:

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



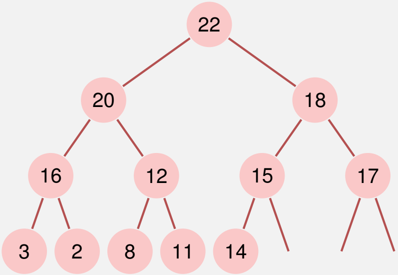


Depends on the starting index[6]

[6] For array that start at 0: $\{2i, 2i+1\} \to \{2i+1, 2i+2\}$, $\lfloor i/2 \rfloor \to \lfloor (i-1)/2 \rfloor$

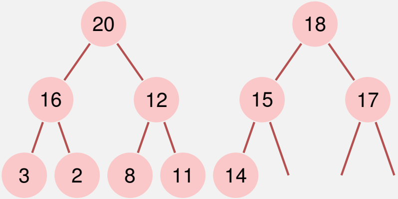
Recursive heap structure

A heap consists of two heaps:

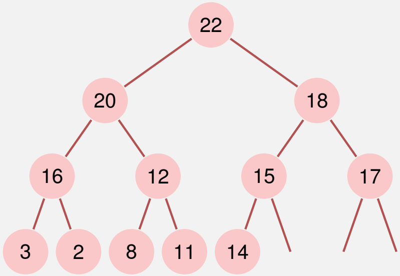


Recursive heap structure

A heap consists of two heaps:

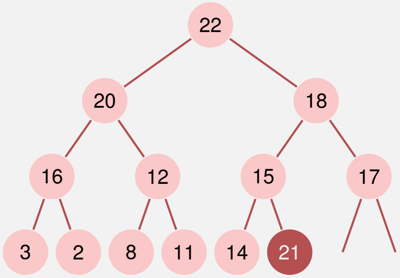


Insert



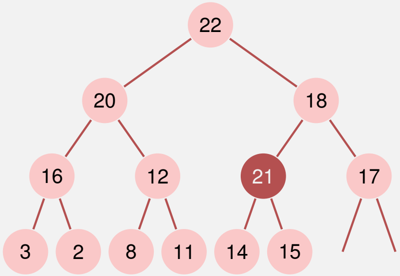
Insert

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



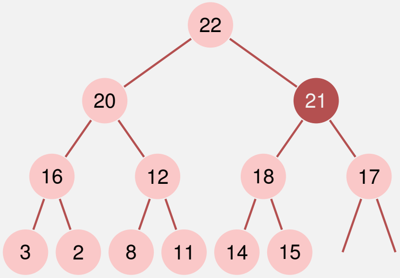
Insert

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



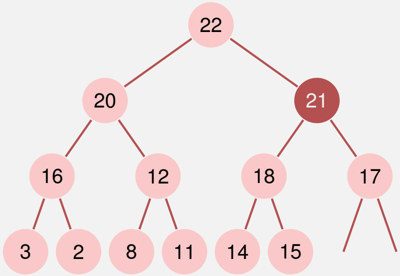
Insert

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

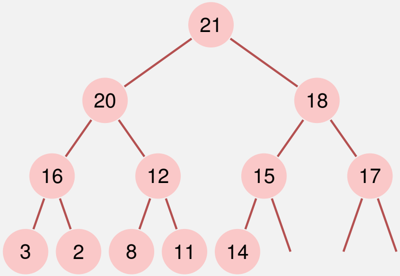


Insert

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

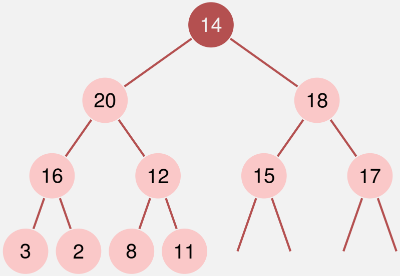


Remove the maximum



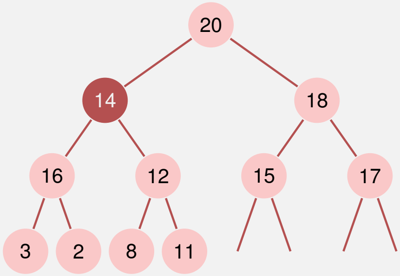
Remove the maximum

- Replace the maximum by the lower right element



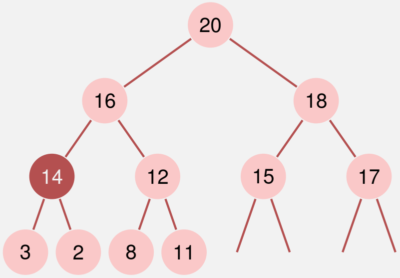
Remove the maximum

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



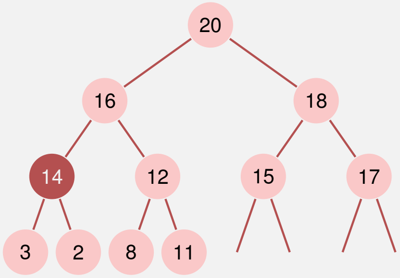
Remove the maximum

- 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: $\mathcal{O}(\log n)$

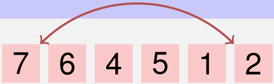


Algorithm Sink(A, i, m)

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 ≤ m do
    j ← 2i; // j left child
    if j < m and A[j] < A[j + 1] then
        j ← j + 1; // j right child with greater key
    if A[i] < A[j] then
        swap(A[i], A[j])
        i ← j; // keep sinking
    else
        i ← m; // sinking finished
```

Sort heap



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

While $n > 1$

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

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

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

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

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

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
    Sink(A, i, n);
// Now A is a heap.
for i ← n downto 2 do
    swap(A[1], A[i])
    Sink(A, 1, i − 1)
// Now A is sorted.
```

Analysis: sorting a heap

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