18. Natürliche Suchbäume

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

Bäume

Bäume sind

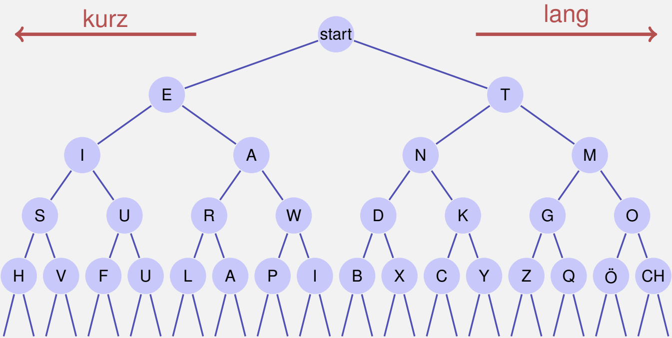
- Verallgemeinerte Listen: Knoten können mehrere Nachfolger haben
- Spezielle Graphen: Graphen bestehen aus Knoten und Kanten. Ein Baum ist ein zusammenhängender, gerichteter, azyklischer Graph.

Bäume

Verwendung

- Entscheidungsbäume: Hierarchische Darstellung von Entscheidungsregeln
- Syntaxbäume: Parsen und Traversieren von Ausdrücken, z.B. in einem Compiler
- Codebäume: Darstellung eines Codes, z.B. Morsealphabet, Huffmann Code
- Suchbäume: ermöglichen effizientes Suchen eines Elementes

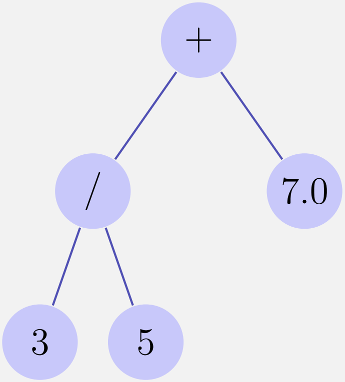
Beispiele



Morsealphabet

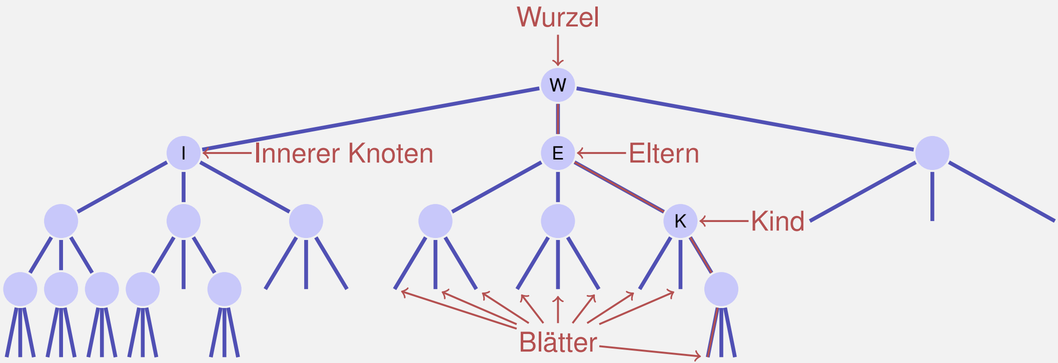
Beispiele

$$3/5 + 7.0$$



Ausdrucksbaum

Nomenklatur



- Ordnung des Baumes: Maximale Anzahl Kindknoten, hier: 3
- Höhe des Baumes: maximale Pfadlänge Wurzel – Blatt (hier: 4)

Binäre Bäume

Ein binärer Baum ist

- entweder ein Blatt, d.h. ein leerer Baum,
- oder ein innerer Knoten mit zwei Bäumen T_l (linker Teilbaum) und T_r (rechter Teilbaum) als linken und rechten Nachfolger.

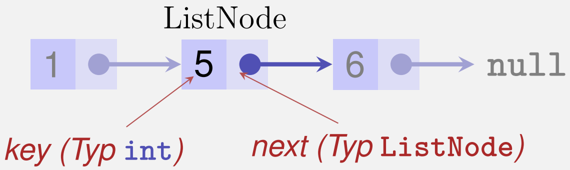
In jedem Knoten v wird gespeichert

key	
left	right

- ein Schlüssel $v.\mathrm{key}$ und
- zwei Zeiger $v.\mathrm{left}$ und $v.\mathrm{right}$ auf die Wurzeln der linken und rechten Teilbäume.

Ein Blatt wird durch den **null**-Zeiger repräsentiert

Zur Erinnerung: Listknoten in Java



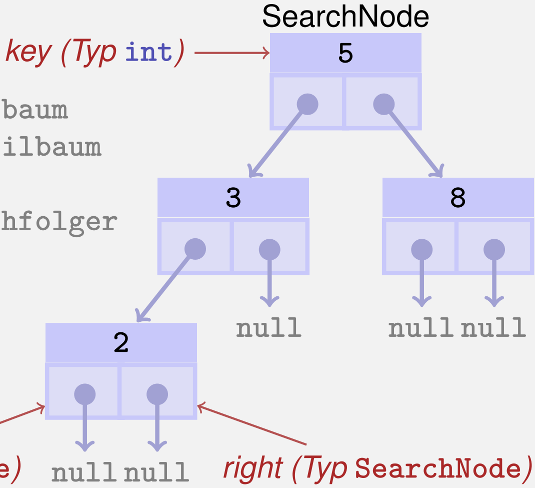

```java
class ListNode {
  int key;
  ListNode next;

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

Jetzt: 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;
  }
}
```

Jetzt: Baumknoten in Java

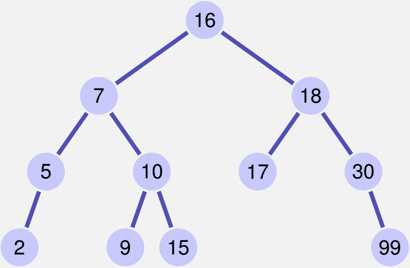
```
public class SearchNode {
  int key;          // Schluessel
  SearchNode left; // linker Teilbaum
  SearchNode right; // rechter Teilbaum

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



Binärer Suchbaum

Ein binärer Suchbaum ist ein binärer Baum, der die *Suchbaumeigenschaft* erfüllt:

- Jeder Knoten `v` speichert einen Schlüssel
- Schlüssel im linken Teilbaum `v.left` kleiner als `v.key`
- Schlüssel im rechten Teilbaum `v.right` grösser als `v.key`



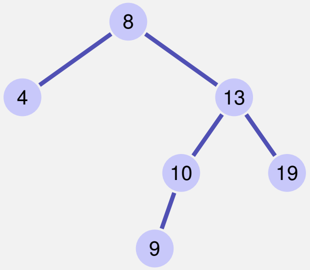
Suchen

Input: Binärer Suchbaum mit Wurzel r,
Schlüssel k

Output: Knoten v mit $v.\text{key} = k$ oder **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

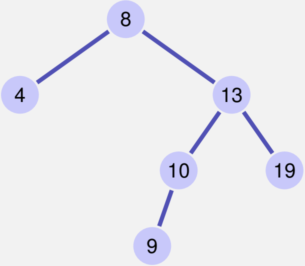


Suchen

Input: Binärer Suchbaum mit Wurzel r,
Schlüssel k
Output: Knoten v mit $v.\mathrm{key} = k$ oder **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)

Suchen

Input: Binärer Suchbaum mit Wurzel r,
Schlüssel k

Output: Knoten v mit $v.\mathrm{key} = k$ oder **null**

$v \leftarrow r$

while $v \neq$ **null do**

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

        return v

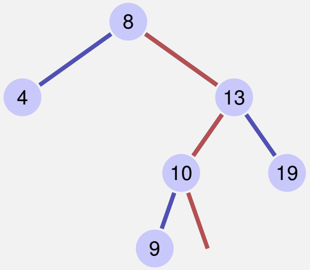
    else if $k < v.\mathrm{key}$ **then**

        $v \leftarrow v.\mathrm{left}$

    else

        $v \leftarrow v.\mathrm{right}$

return null



Search (12)

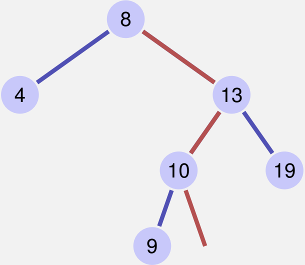
Suchen

Input: Binärer Suchbaum mit Wurzel r,
Schlüssel k

Output: Knoten v mit $v.\mathrm{key} = k$ oder **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**

Suchbaum und Suchen in Java

```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
}
```

Höhe eines Baumes

Die Höhe $h(T)$ eines Baumes T mit Wurzel r ist gegeben als

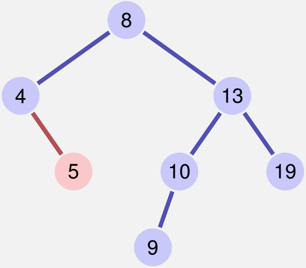
$$h(r) = \begin{cases} 0 & \text{falls } r = \textbf{null} \\ 1 + \max\{h(r.\mathrm{left}), h(r.\mathrm{right})\} & \text{sonst.} \end{cases}$$

Die Laufzeit der Suche ist (im schlechtesten Fall) bestimmt durch die Höhe des Baumes

Einfügen eines Schlüssels

Einfügen des Schlüssels k

- Suche nach k.
- Wenn erfolgreich: Fehlerausgabe
- Wenn erfolglos: Einfügen des Schlüssels am erreichten Blatt.
- Implementation: der Teufel steckt im Detail



Insert (5)

Knoten Einfügen in Java

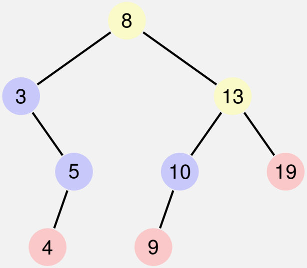
```
public boolean add (int k) {
  if (root == null) {root = new SearchNode(k); return true;}
  SearchNode t=root;
  while (k != t.key) {
    if (k < t.key) {
      if (t.left == null){ t.left = new SearchNode(k); return true;}
      else { t = t.left; }
    }
    else { // k > t.key
      if (t.right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

Knoten entfernen

Drei Fälle möglich

- Knoten hat keine Kinder
- Knoten hat ein Kind
- Knoten hat zwei Kinder

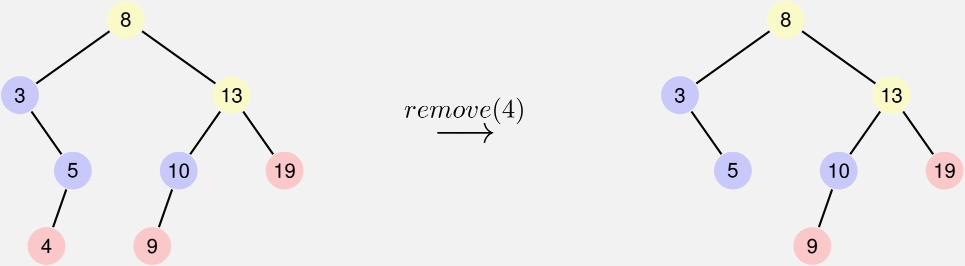
[Blätter zählen hier nicht]



Knoten entfernen

Knoten hat keine Kinder

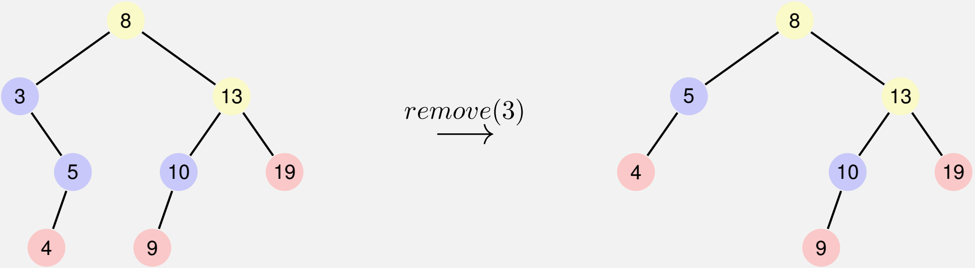
Einfacher Fall: Knoten durch Blatt ersetzen.



Knoten entfernen

Knoten hat ein Kind

Auch einfach: Knoten durch das einzige Kind ersetzen.



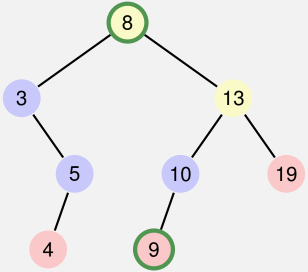
Knoten entfernen

Knoten `v` hat zwei Kinder

Beobachtung: Der kleinste Schlüssel im rechten Teilbaum `v.right` (der *symmetrische Nachfolger* von `v`)

- ist kleiner als alle Schlüssel in `v.right`
- ist grösser als alle Schlüssel in `v.left`
- und hat kein linkes Kind.

Lösung: ersetze `v` durch seinen symmetrischen Nachfolger

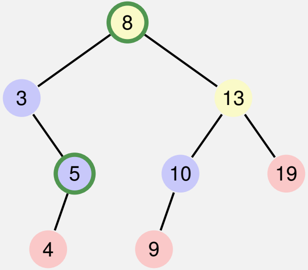


Aus Symmetriegründen...

Knoten v hat zwei Kinder

Auch möglich: ersetze v durch seinen symmetrischen Vorgänger

Implementation: der Teufel steckt im Detail!



Algorithmus SymmetricSuccessor(v)

Input: Knoten v eines binären Suchbaumes
Output: Symmetrischer Nachfolger von 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

SymmetricDesc in Java

```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;
}
```

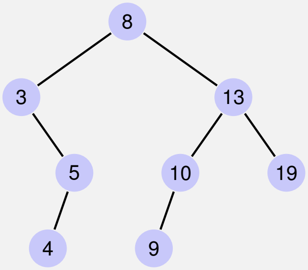
Dieser Algorithmus gibt den symmetrischen Nachfolger zurück. Aber tut noch mehr: er behandelt auch die Fälle mit einem oder keinem Nachfolger. Ausserdem ersetzt er den Symmetrischen Nachfolger durch dessen Nachfolgeknoten.

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;
}
```

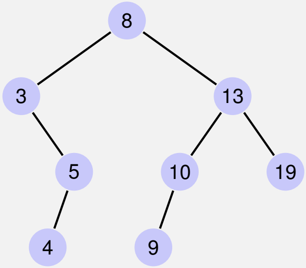
Traversierungsarten

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



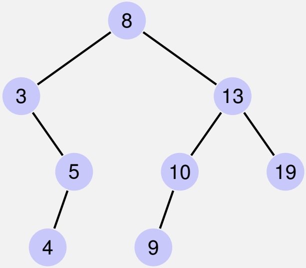
Traversierungsarten

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



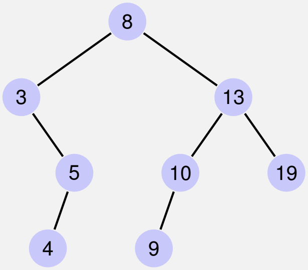
Traversierungsarten

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



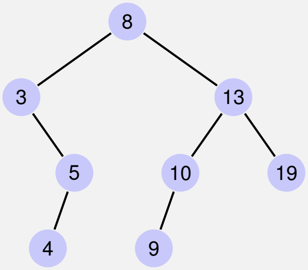
Traversierungsarten

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



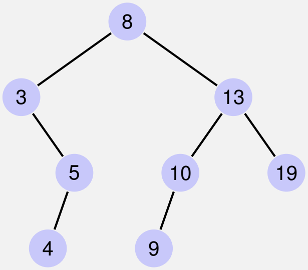
Traversierungsarten

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

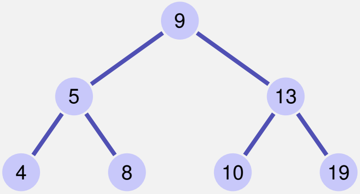


Traversierungsarten

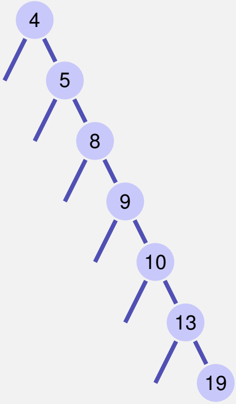
- Hauptreihenfolge (preorder): v, dann $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$.
  8, 3, 5, 4, 13, 10, 9, 19
- Nebenreihenfolge (postorder): $T_{\text{left}}(v)$, dann $T_{\text{right}}(v)$, dann v.
  4, 5, 3, 9, 10, 19, 13, 8
- Symmetrische Reihenfolge (inorder): $T_{\text{left}}(v)$, dann v,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



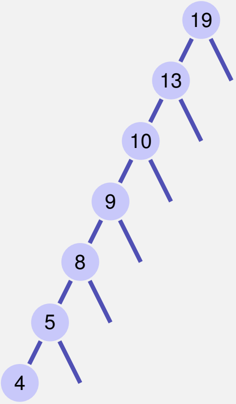
Degenerierte Suchbäume



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



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



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

Effizienzbetrachtungen

Offensichtlich hängt die Laufzeit der Algorithmen Suchen, Einfügen und Löschen im schlechtesten Falle von der Höhe des Baumes ab.

Entartete Bäume sind im schlechtesten Fall also nicht besser als eine verkettete Liste

Balancierte Bäume stellen beim Einfügen und Entfernen (z.B. durch *Rotationen*) sicher, dass der Baum balanciert bleibt und liefern gewisse Laufzeitgarantien für die Algorithmen

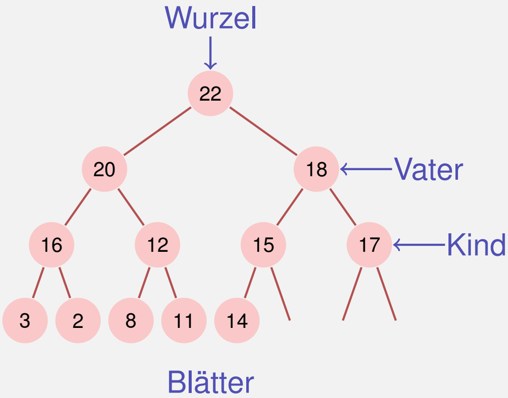
Die `TreeSet` und `TreeMap` Klassen in Java sind mit balancierten Bäumen (sog. Rot-Schwarz-Bäumen) implementiert.

19. Heaps

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

[Max-]Heap[9]

Binärer Baum mit folgenden Eigenschaften

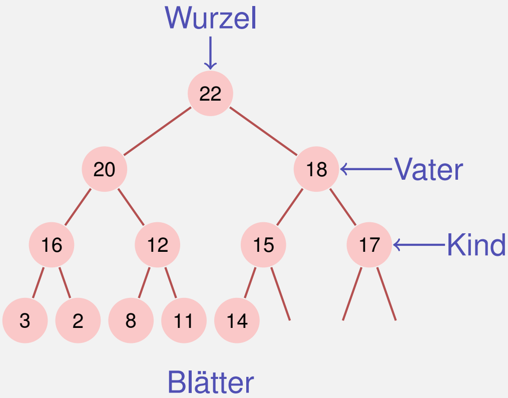


[9] Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

[Max-]Heap[9]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene

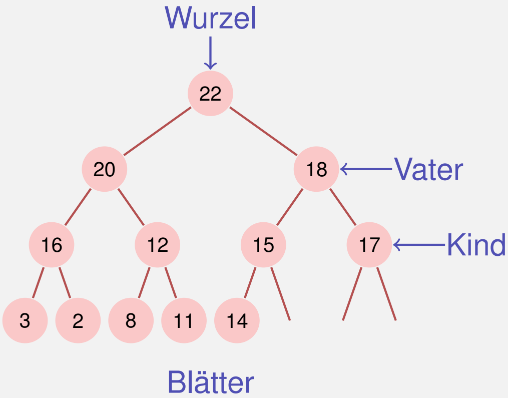


[9] Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

[Max-]Heap[9]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.

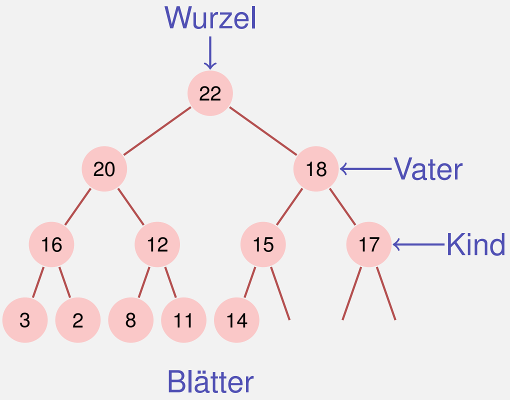


[9]Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

[Max-]Heap[9]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.
3. *Heap-Bedingung:*
   Max-(Min-)Heap: Schlüssel eines Kindes kleiner (grösser) als der des Vaters

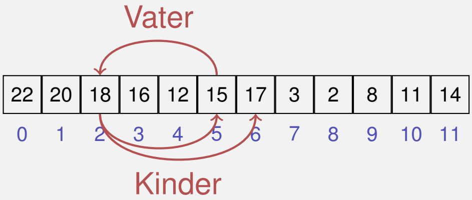


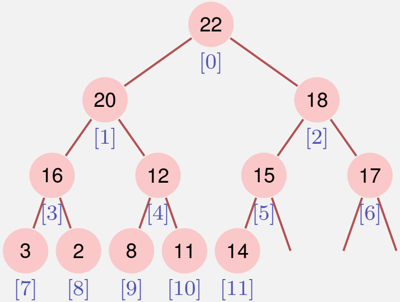
[9]Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

Heap als Array

Baum $\rightarrow$ Array:

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





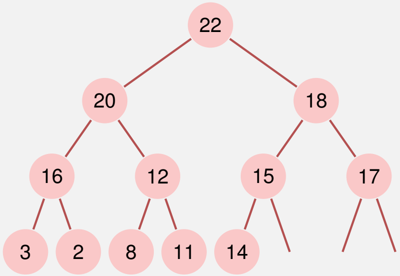
Abhängig von Startindex![10]

[10] Für Arrays, die bei 1 beginnen: $\{2i+1, 2i+2\} \rightarrow \{2i, 2i+1\}$, $\lfloor (i-1)/2 \rfloor \rightarrow \lfloor i/2 \rfloor$

Heap in Java

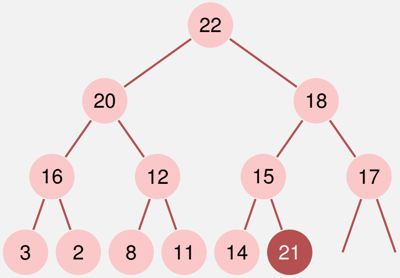
```
public class Heap {
  double[] A;// will need to grow
  int sz;
  // Heap initialized with 16 elements
  Heap () {
    A = new double[16]; sz = 0;
  }
  // binary growth of the array
  void grow(){ ... }
  // insert element in the heap
  public void add(double value){...}
  // extract and return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

Einfügen



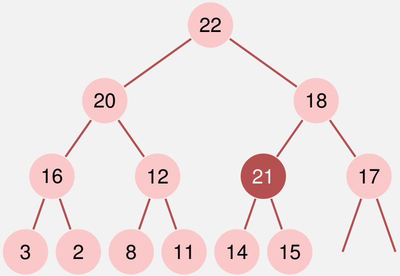
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.



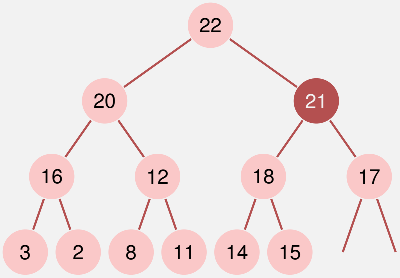
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.



Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.



Algorithmus Aufsteigen(A, m)

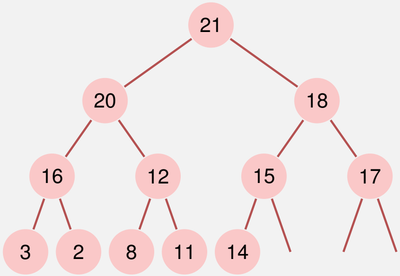
Input: Array A mit mindestens $m+1$ Elementen und Max-Heap-Struktur auf $A[0,\dots,m-1]$
Output: Array A mit Max-Heap-Struktur auf $A[0,\dots,m]$.

$v \leftarrow A[m]$ // Wert
$c \leftarrow m$ // derzeitiger Knoten
$p \leftarrow \lfloor (c-1)/2 \rfloor$ // Elternknoten
while $c > 0$ and $v > A[p]$ **do**
    $A[c] \leftarrow A[p]$ // Wert Elternknoten $\rightarrow$ derzeitiger Knoten
    $c \leftarrow p$ // Elternknoten $\rightarrow$ derzeitiger Knoten
    $p \leftarrow \lfloor (c-1)/2 \rfloor$
$A[c] \leftarrow v$ // Wert $\rightarrow$ derzeitiger Knoten

add

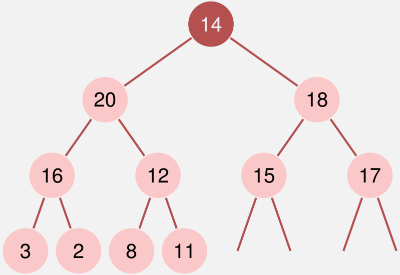
```
// insert element to the heap
public void add(double value){
  if (sz == A.length){ grow(); }
  int current = sz;
  int parent = (current-1)/2;
  // sift value up
  while (current > 0 && value > A[parent]) {
    A[current] = A[parent];
    current = parent;
    parent = (current-1)/2;
  }
  A[current]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

Maximum entfernen



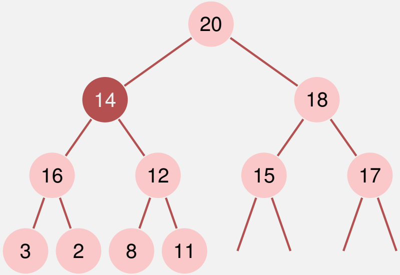
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.



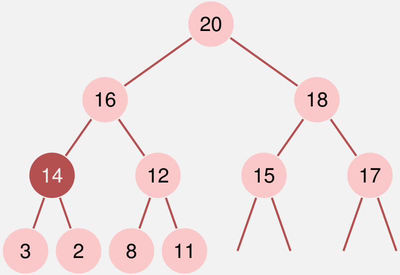
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



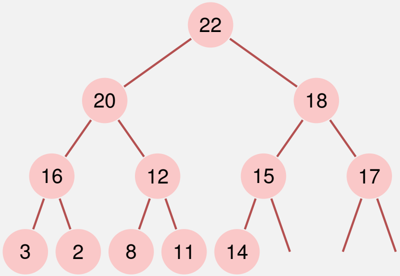
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



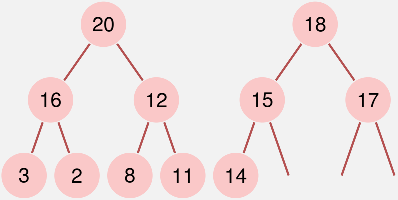
Warum das korrekt ist: Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:



Warum das korrekt ist: Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:



Algorithmus Versickern(A, i, m)

Input: Array A mit Max-Heap-Struktur für die Kinder von i. Letztes Element m.
Output: Array A mit Max-Heap-Struktur für i mit letztem Element m.

while $2i + 1 \le m$ **do**
  $j \leftarrow 2i + 1$; // j linkes Kind
  if $j < m$ and $A[j] < A[j+1]$ **then**
    $j \leftarrow j + 1$; // j rechtes Kind mit grösserem Schlüssel
  if $A[i] < A[j]$ **then**
    swap($A[i], A[j]$)
    $i \leftarrow j$; // weiter versickern
  else
    $i \leftarrow m$; // versickern beendet

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;
}
```

Heap Sortieren

7	6	4	5	1	2

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

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

Heap Sortieren

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

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

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

Heap Sortieren

	7	6	4	5	1	2
Tauschen $\Rightarrow$	2	6	4	5	1	7
Versickern $\Rightarrow$	6	5	4	2	1	7

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

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

Heap Sortieren

$A[1,...,n]$ ist Heap.
Solange $n > 1$

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

		7	6	4	5	1	2
Tauschen	$\Rightarrow$	2	6	4	5	1	7
Versickern	$\Rightarrow$	6	5	4	2	1	7
Tauschen	$\Rightarrow$	1	5	4	2	6	7

Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

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

		7	6	4	5	1	2
Tauschen	$\Rightarrow$	2	6	4	5	1	7
Versickern	$\Rightarrow$	6	5	4	2	1	7
Tauschen	$\Rightarrow$	1	5	4	2	6	7
Versickern	$\Rightarrow$	5	4	2	1	6	7
Tauschen	$\Rightarrow$	1	4	2	5	6	7
Versickern	$\Rightarrow$	4	1	2	5	6	7
Tauschen	$\Rightarrow$	2	1	4	5	6	7
Versickern	$\Rightarrow$	2	1	4	5	6	7
Tauschen	$\Rightarrow$	1	2	4	5	6	7

Höhe eines Heaps

Welche Höhe $H(n)$ hat ein Heap mit n Knoten? Auf der i-ten Ebene eines Binären Baumes befinden sich höchstens 2^i Knoten. Bis auf die letzte Ebene sind alle Ebenen eines Heaps aufgefüllt.

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

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

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

also

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

Laufzeit der Heap-Algorithmen

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

Die Algorithmen Einfügen und Extrahieren machen jeweils etwa $\log_2(n+1)$ „Schritte”.[11]

Da der Logarithmus sehr langsam wächst, ist der Heap damit eine sehr schnelle Datenstruktur. Er wird zum Sortieren und zur Implementation von Priority-Queues eingesetzt.

[11] Wird in Informatik II präzisiert.