Informatik II

Vorlesung am D-BAUG der ETH Zürich

Vorlesung 9, 2.5.2016

[Nachtrag zu Vorlesung 8: Numerische Integration, Zusammenfassung Objektorientierte Programmierung]

Dynamische Datenstrukturen II: Bäume und Heaps

Bäume, Binäre Suchbäume, Heap, Effizienter Online-Median

DYNAMISCHE DATENSTRUKTUREN II

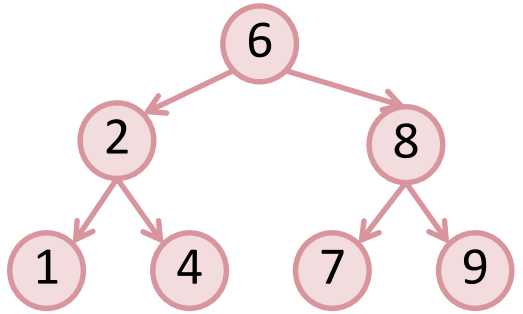
Bäume – Motivation

Erinnerung an die Fallstudie Online-Statistik

- Median konnte nicht effizient berechnet werden
- Verlinkte Listen schaffen Abhilfe bzgl. effizienterem sortierten Einfügen von Elementen. Laufzeit für die Suche eines Elementes wird jedoch nicht verringert

Suchbäume können verwendet werden, um indizierte Elemente effizient zu suchen

- Was nützt uns das für den Median?
- Antwort etwas später



Literatur: Ottmann, Widmayer, Algorithmen und Datenstrukturen, Kapitel 5

Bäume – Motivation

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.

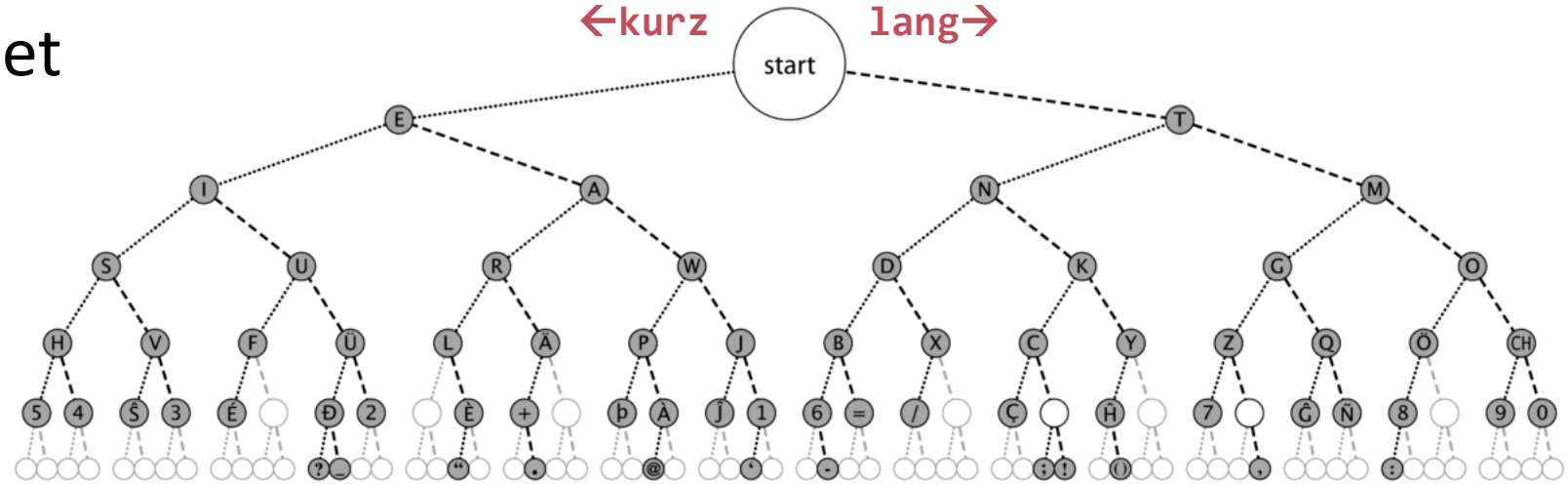
Verwendung

- 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

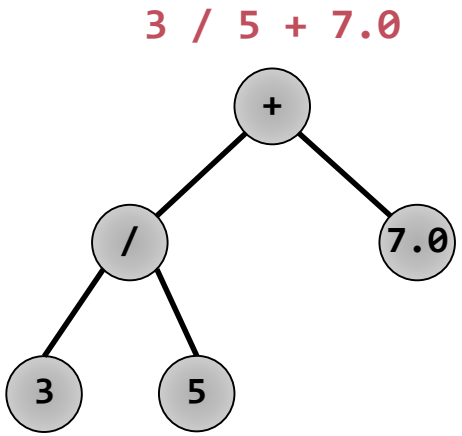
*manche Autoren betrachten auch ungerichtete Bäume (wir nicht)

Beispiele

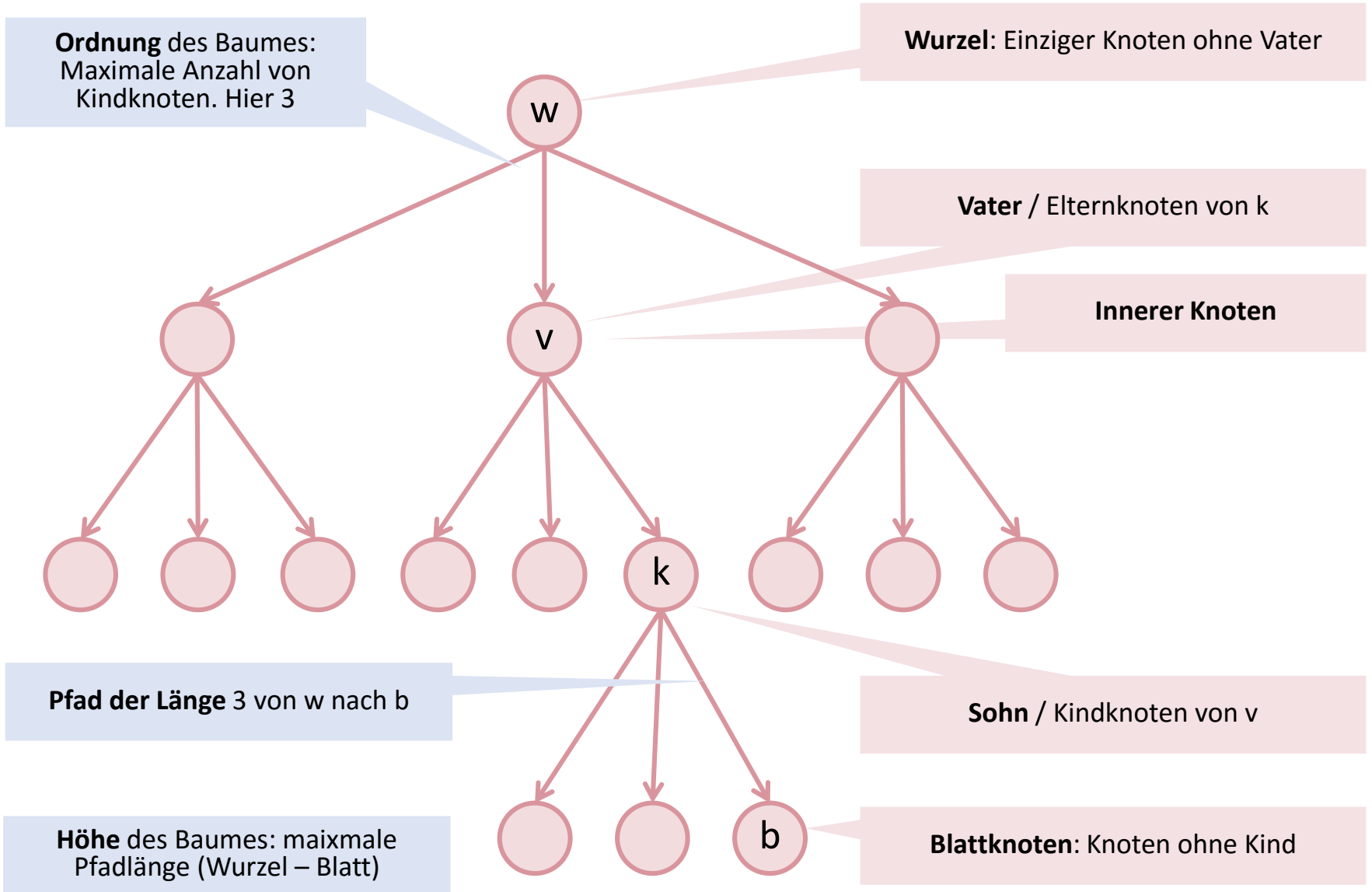
Morsealphabet



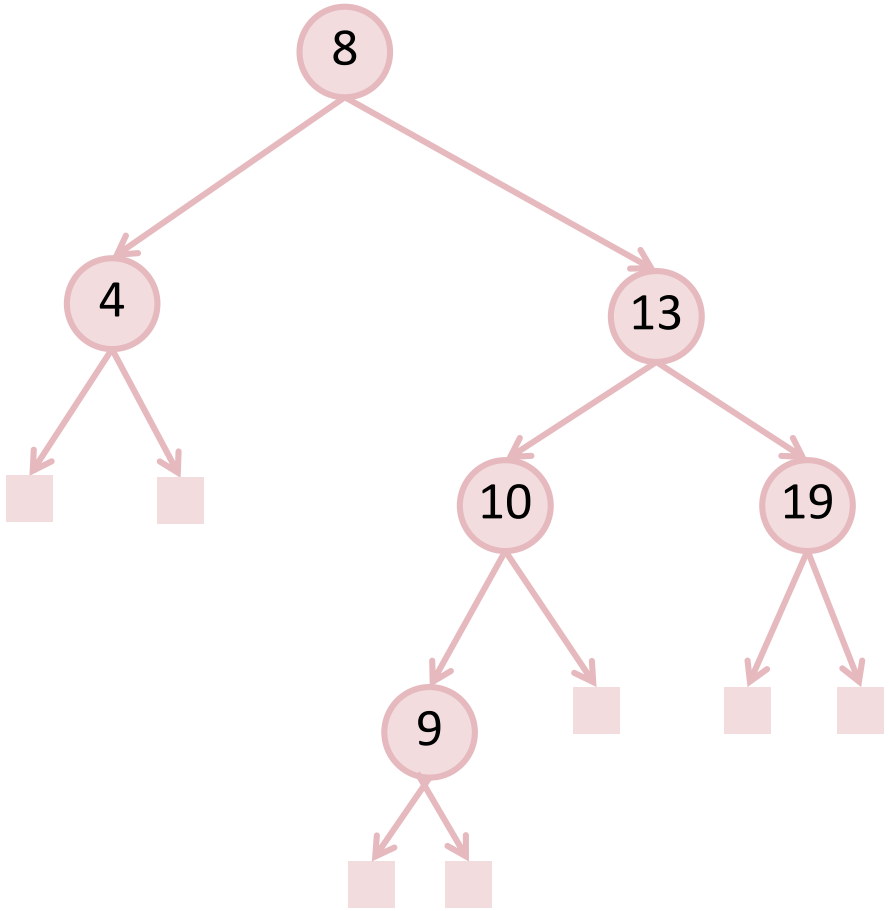
Ausdrucksbaum



Bäume: Nomenklatur



Binärer Suchbaum



$(-\infty, \mathbf{4}), (\mathbf{4}, \mathbf{8}), (\mathbf{8}, \mathbf{9}), (\mathbf{9}, \mathbf{10}), (\mathbf{10}, \mathbf{13},), (\mathbf{13}, \mathbf{19}), (\mathbf{19}, \infty)$

Baum der Ordnung 2

Knoten beherbergen paarweise verschiedene Schlüssel (und potentiell andere Nutzdaten)

Schlüssel im linken Teilbaum kleiner als am Knoten

Schlüssel im rechten Teilbaum grösser als am Knoten

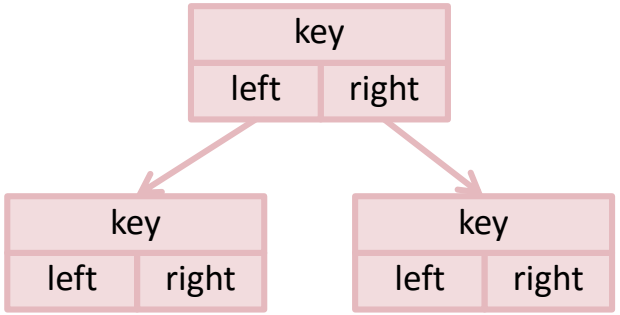
(Leere*) Blätter repräsentieren Schlüsselintervalle

*ausser bei Blattsuchbäumen, betrachten wir hier nicht

Datenstruktur Suchknoten

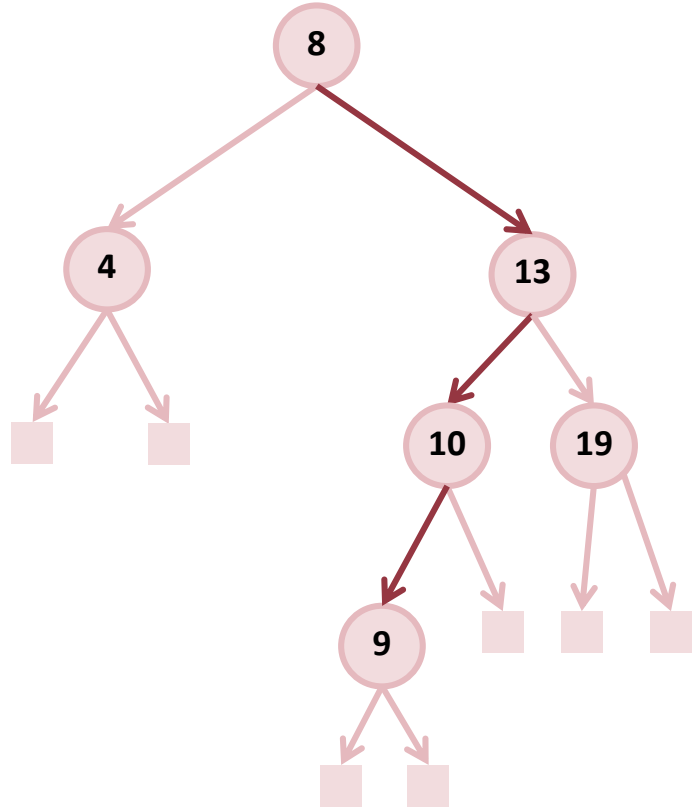
Ähnlich wie bei den Listen besteht ein Baum aus verketteten / verflochtenen Instanzen einer Datenstruktur Knoten (SearchNode)

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

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



Datenstruktur Suchbaum

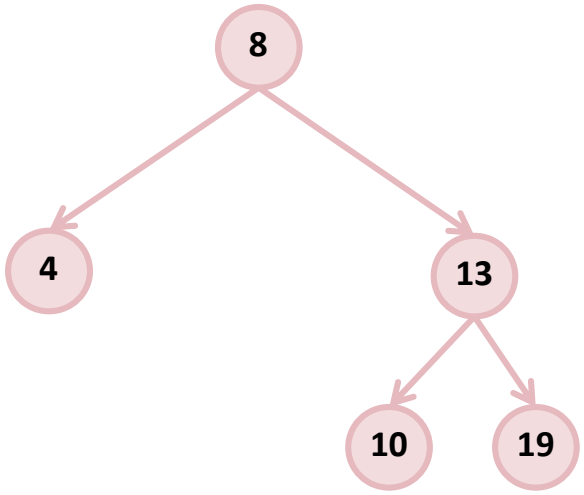
```
public class SearchTree {
   SearchNode root = null;// Wurzelknoten

   // Gibt Knoten mit Schlüssel k zurück.
   // 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;
   }

   ... // Einfügen, Löschen
}
```

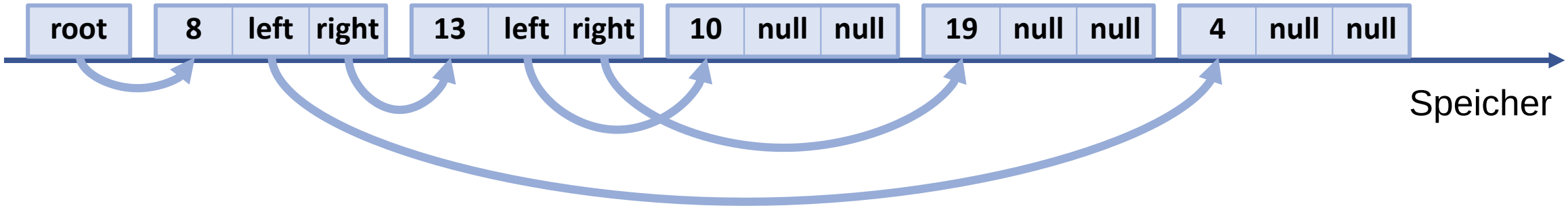


Beispiel: Search(9)

Sieht effizient aus.
Stimmt das denn (immer)?

Baum Im Speicher

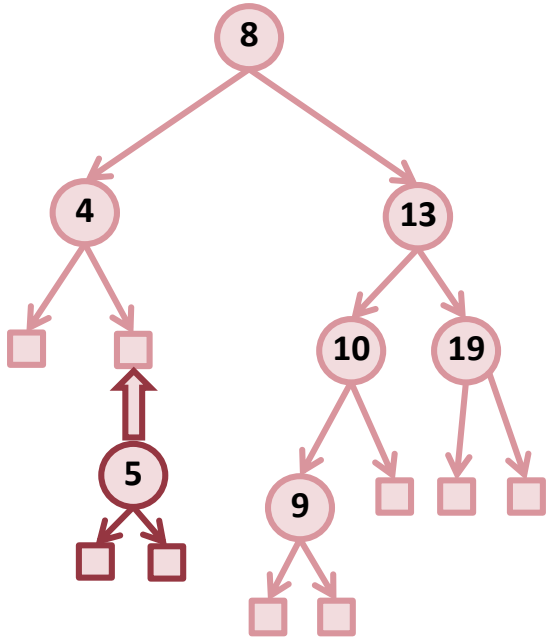




Knoten Einfügen

```
// Fügt Knoten mit Schlüssel k ein. Gibt erzeugten Knoten zurück.
// null, wenn Knoten mit Schlüssel k bereits existiert
public SearchNode Insert (int k) {
    if (root == null)
        return root = new SearchNode(k);
    SearchNode t = root;
    while (true){
        if (k == t.key)
            return null; // schon vorhanden
        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;
        }
    }
}
```

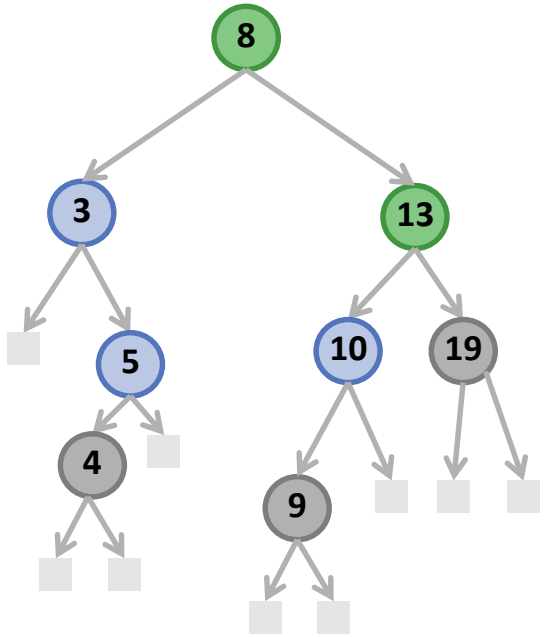
Traversiere Baum bis zum passenden Intervall-Blatt, ersetze Intervallblatt mit Knoten



Knoten Löschen

Drei Fälle

1. Knoten hat keine Kinder
   - Knoten entfernen
2. Knoten hat nur ein Kind
   - Knoten durch Kind ersetzen
3. Knoten hat zwei Kinder
   - Knoten durch einen *symmetrischen Nachfolger* ersetzen
   - Symmetrischer Nachfolger:
     Knoten im rechten (linken) Teilbaum, welcher am weitesten links (rechts) steht.
   - Korrespondiert mit dem kleinsten (grössten) Schlüssel, welcher gerade noch grösser (kleiner) als der Schlüssel des zu entfernenden Knotens ist
   - Symmetrischer Nachfolger hat maximal ein Kind



Symmetrischer Nachfolger

```
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;
}
```

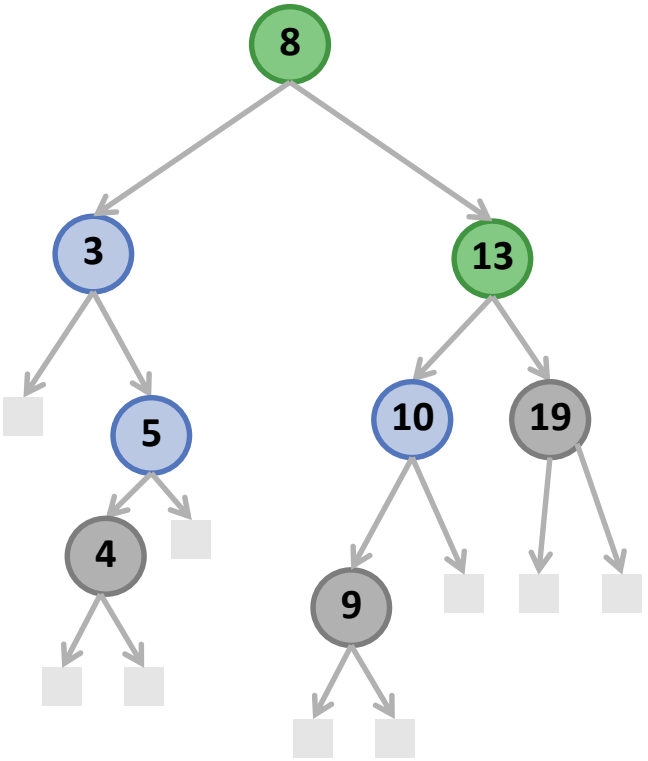
Wenn der Knoten nur ein Kind hat, gib dieses zurück.

Wenn der Knoten kein Kind hat, gib null zurück

Suche den symmetrischen Nachfolger von node: der Knoten im rechten Teilbaum, welcher am weitesten links steht

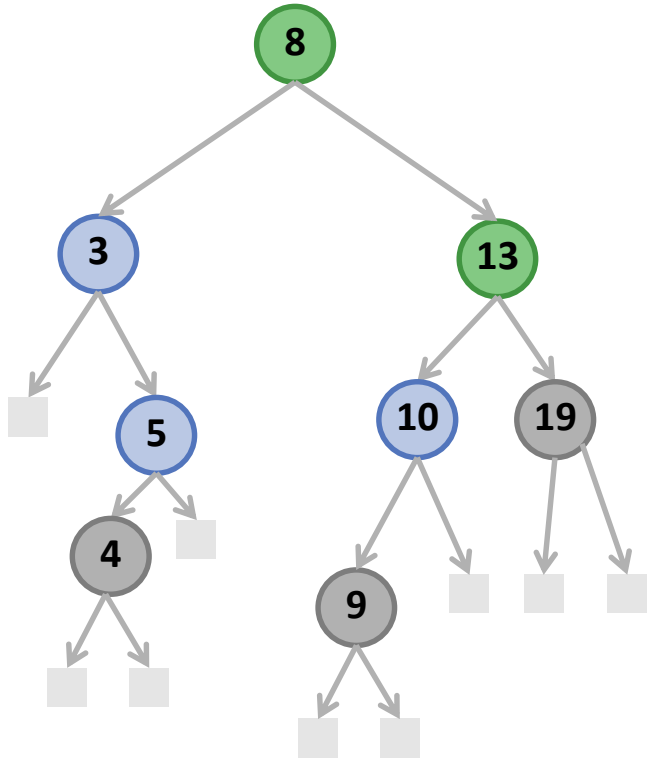
Entferne den symmetrischen Nachfolger: ersetze ihn durch sein rechtes Kind (er kann kein linkes Kind haben)!

Ersetze Kinder des gelöschten Knotens durch die Kinder von node.

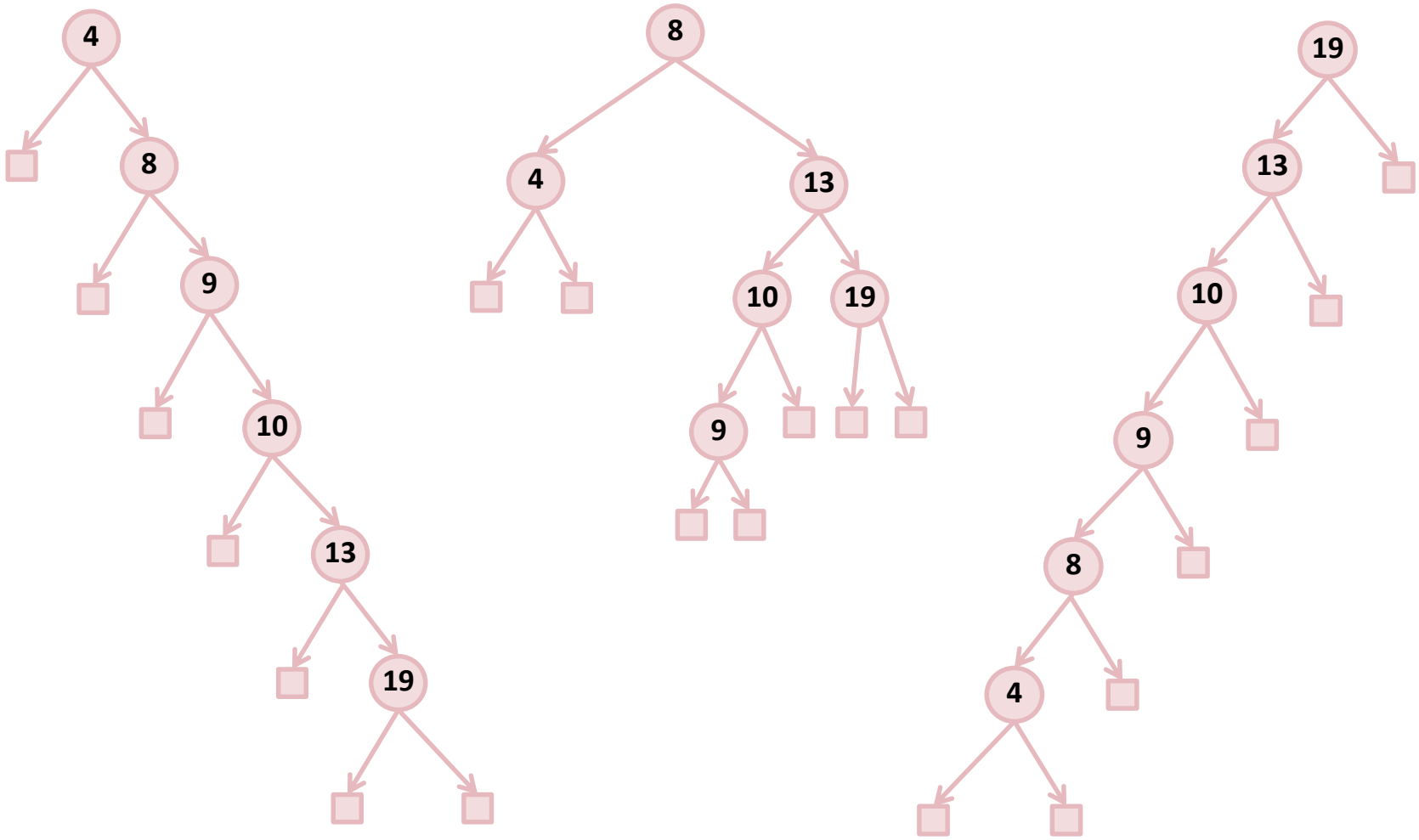


Knoten Löschen

```
public boolean Delete (int k) {
    SearchNode n = root;
    if (n != null && n.key == k) {
        root = SymmetricDesc(root);
        return true;
    }
    else { // suche und lösche mit Hilfe von SymmetricDesc()
        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;
    }
}
```



Degenerierte Bäume



Binäre Bäume können, je nach Updateoperationen / Einfügereihenfole, zu linearen Listen degenerieren! Folgerung?

Balancierte Bäume

Komplexität von Suchen, Einfügen und Löschen eines Knoten in binären Suchbäumen *im Mittel* $O(\log_2 n)$

Worst case: $O(n)$ bei degeneriertem Baum

Verhinderung der Degenerierung: künstliches, bei jeder Update-Operation erfolgtes Balancieren eines Baumes: worst case auch $O(\log_2 n)$

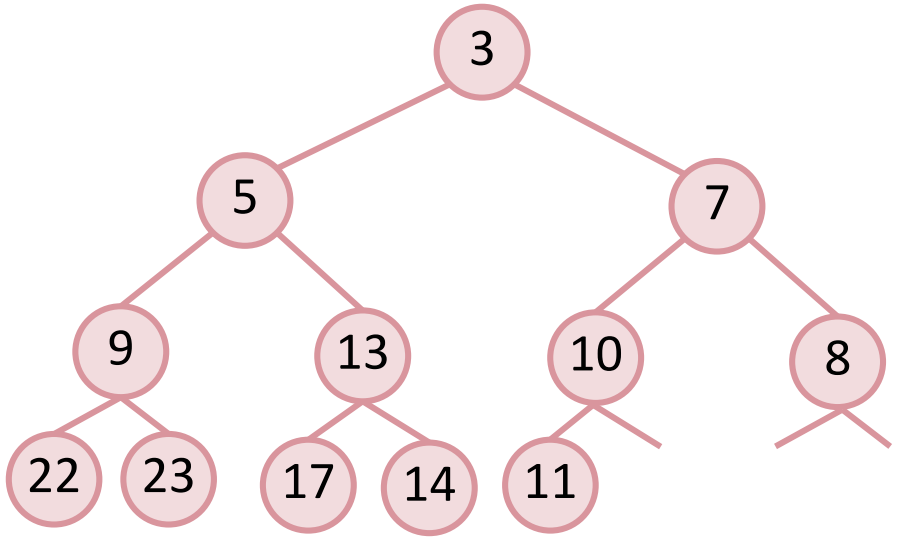
- Balancieren: garantiere, dass ein Baum mit n Knoten stets eine Höhe von $\mathrm{O}(\log n)$ hat.
- Z.B. AVL Bäume / Rot-Schwarz-Bäume etc. Wir behandeln das nicht.

Heaps

Heaps

Ein (Min-)Heap ist ein Binärbaum, welcher

- die (Min-)Heap-Eigenschaft hat: Schlüssel eines Kindes ist immer grösser als der des Vaters.
  [Max-Heap: Kinder-Schlüssel immer kleiner als Vater-Schlüssel]
- bis auf die letzte Ebene vollständig ist
- höchstens Lücken in der letzten Ebene hat, welche alle rechts liegen müssen

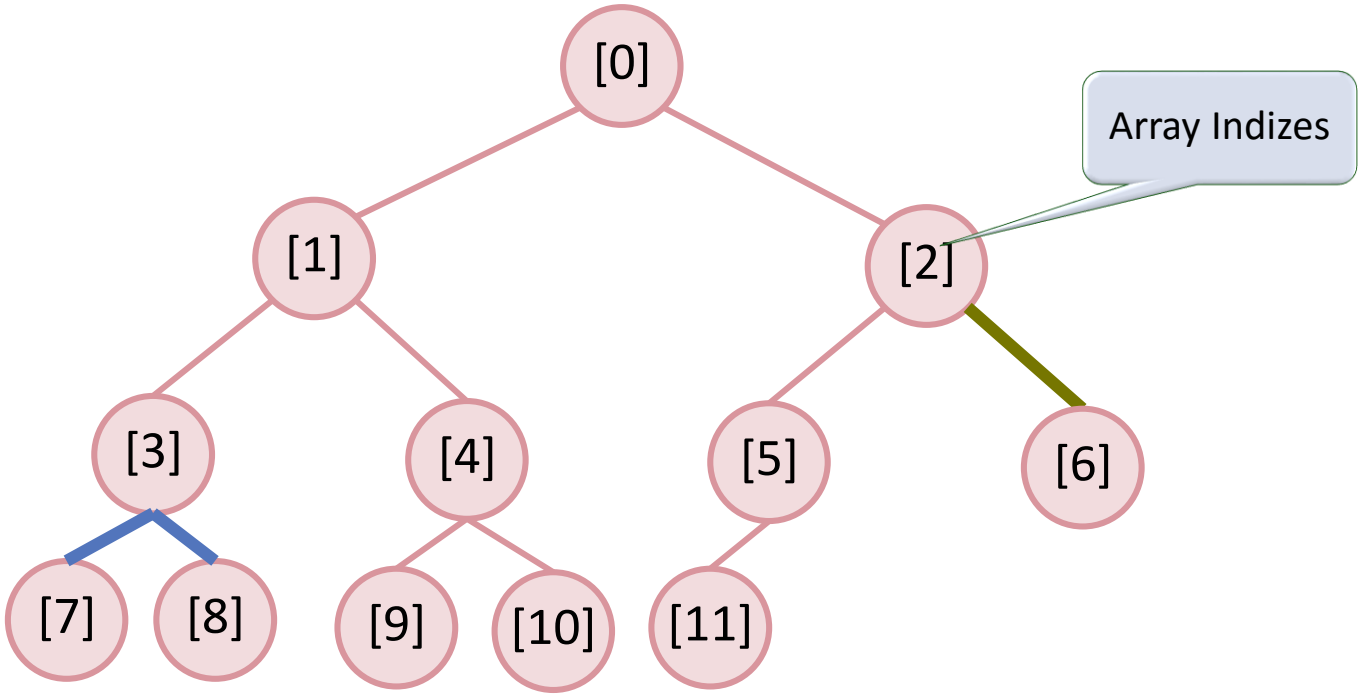


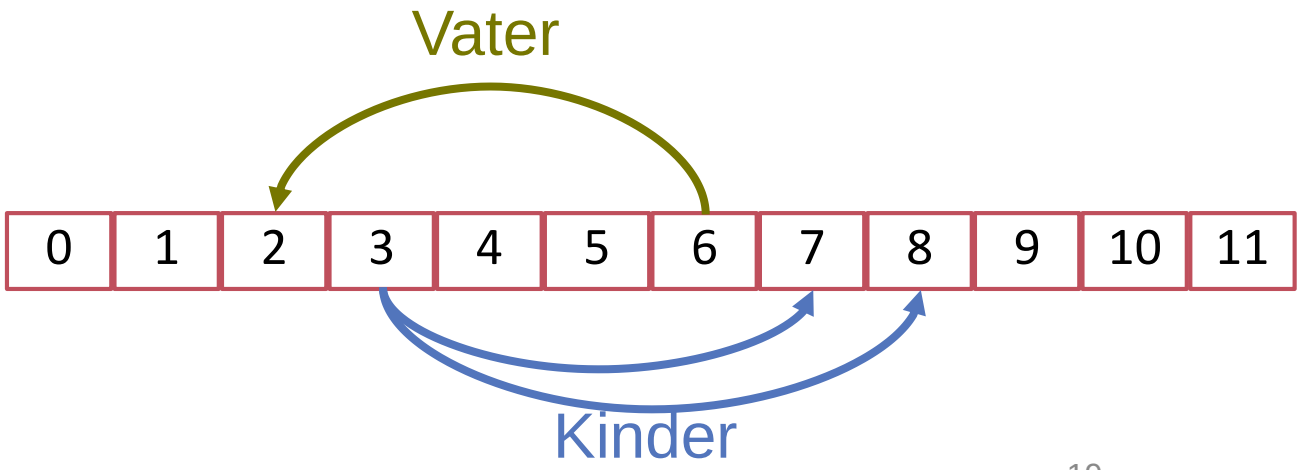
Heaps und Arrays

Ein Heap lässt sich sehr gut in einem Array speichern:

Es gilt

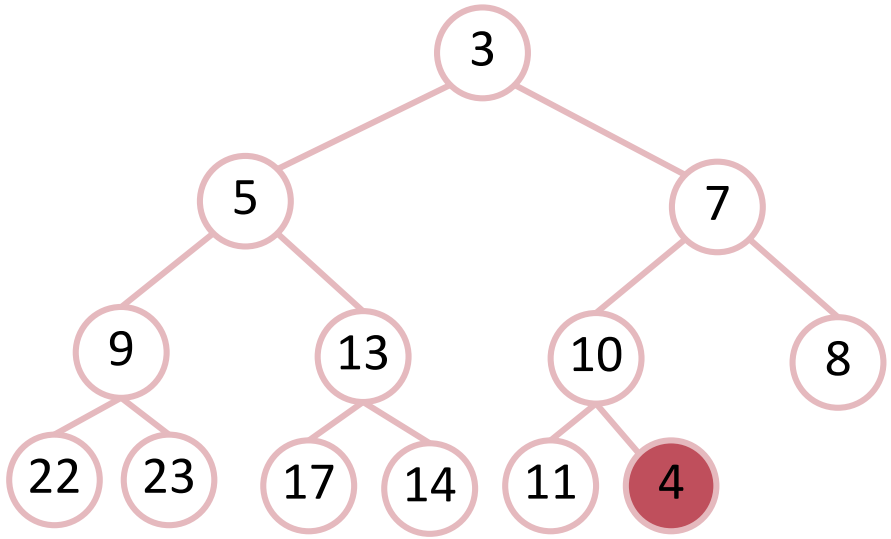
- $\text{Kinder}(i) = \{2i + 1, 2i + 2\}$
- $\text{Vater}(i) = \lfloor (i - 1)/2 \rfloor$





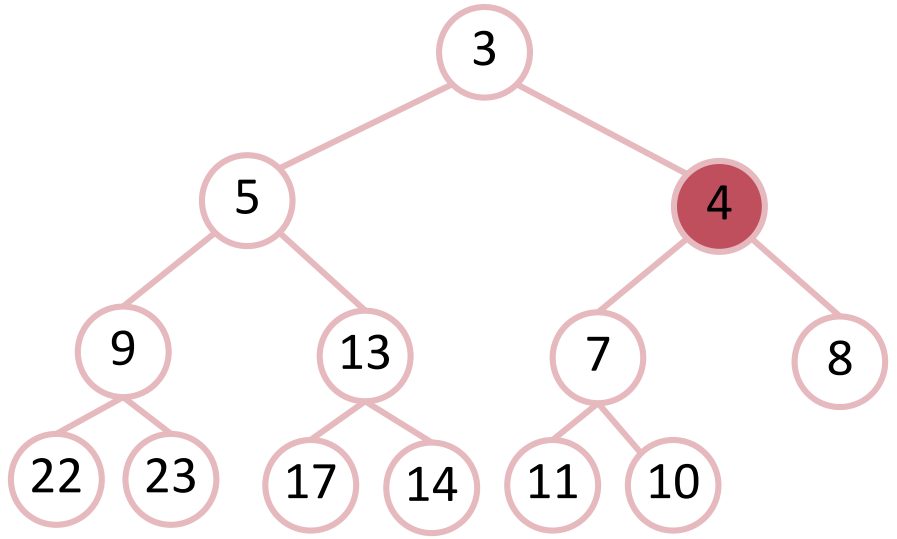
Einfügen

- Füge ein neues Element k an der ersten freien Stelle ein. Verletzt Heap-Eigenschaft potentiell.



- Stelle Heap-Eigenschaft wieder her durch sukzessives Aufsteigen von k.

- Worst-Case Komplexität $\boldsymbol{O(log\, n)}$

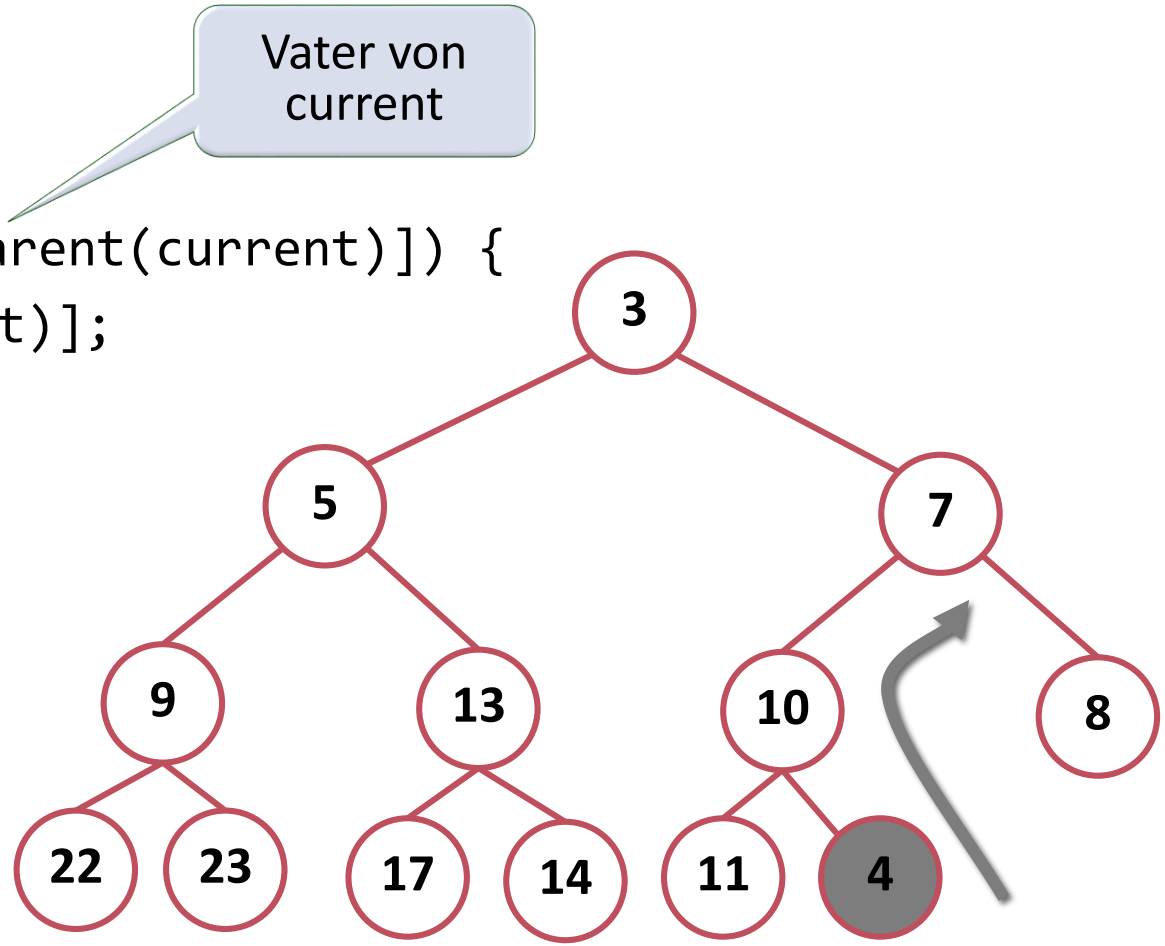


Datenstruktur ArrayHeap

```
public class ArrayHeap {

   float[] data;   // Array zum Speichern der Daten
   int used;       // Anzahl belegte Knoten

   ArrayHeap () {
      data = new float[16];
      used = 0;
   }

   int Parent(int of){
      return (of-1)/2:
   }

   void Grow(){ … } // Binäres Vergrössern von data, wenn nötig
   ...
}
```

Insert

```
public void Insert(double value){
   if (used == data.length)
      Grow();
   int current = used;
   while (current > 0 && value < data[Parent(current)]) {
      data[current] = data[Parent(current)];
      current = Parent(current);
   }
   data[current] = value;
   used++;
   Check(0); // Debugging check
}
```



GetMin

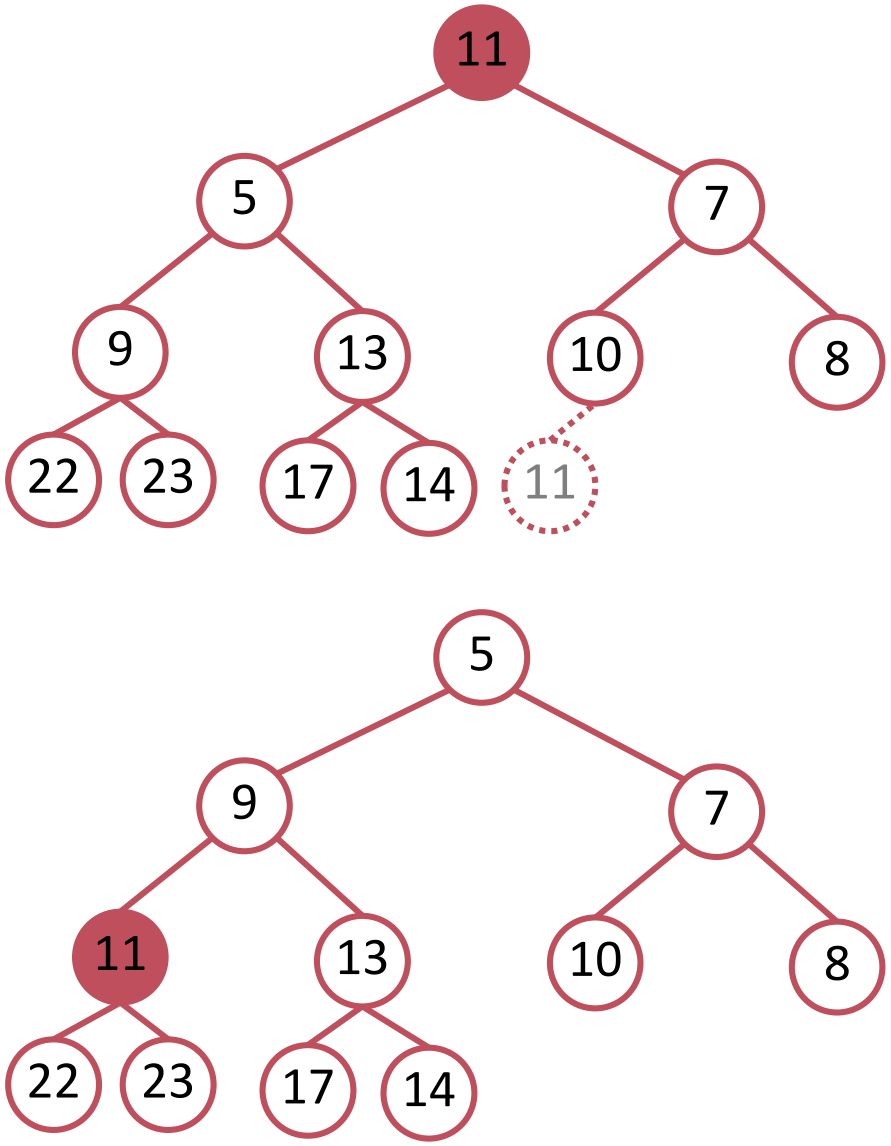
Das kleinste Element ist immer an der Wurzel im Baum. Somit kann es sehr schnell ausgelesen werden ($O(1)$).

Wie verhält es sich aber mit Auslesen ***und Entfernen****?*

Wiederholtes Entfernen der Wurzel ergibt Schlüssel in aufsteigender Reihenfolge: das kann z.B. auch zum *Sortieren* verwendet werden (Heap-Sort Algorithmus).

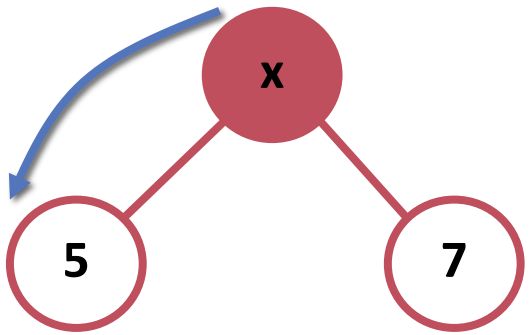
Minimum entfernen

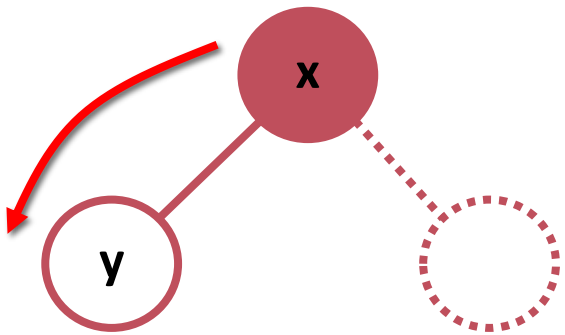
- Ersetze die Wurzel durch den letzten Knoten
- Lasse die Wurzel nach unten sinken, um die Invariante wiederherzustellen
- Worst-Case Komplexität $\boldsymbol{O(log\, n)}$

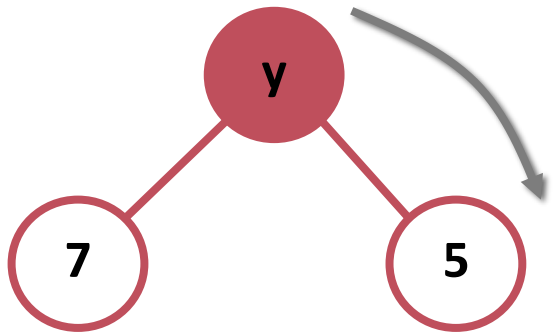


Absinken: Welche Richtung?

```
// return child with smaller value.
// If larger child index (2*current+2) is not in range, return smaller index
private int BestChild(int current) {
    if (2*current+2 >= used || data[2*current+1] < data[2*current+2])
        return 2*current+1;    // take left branch
    else
        return 2*current+2;    // right branch
}
```

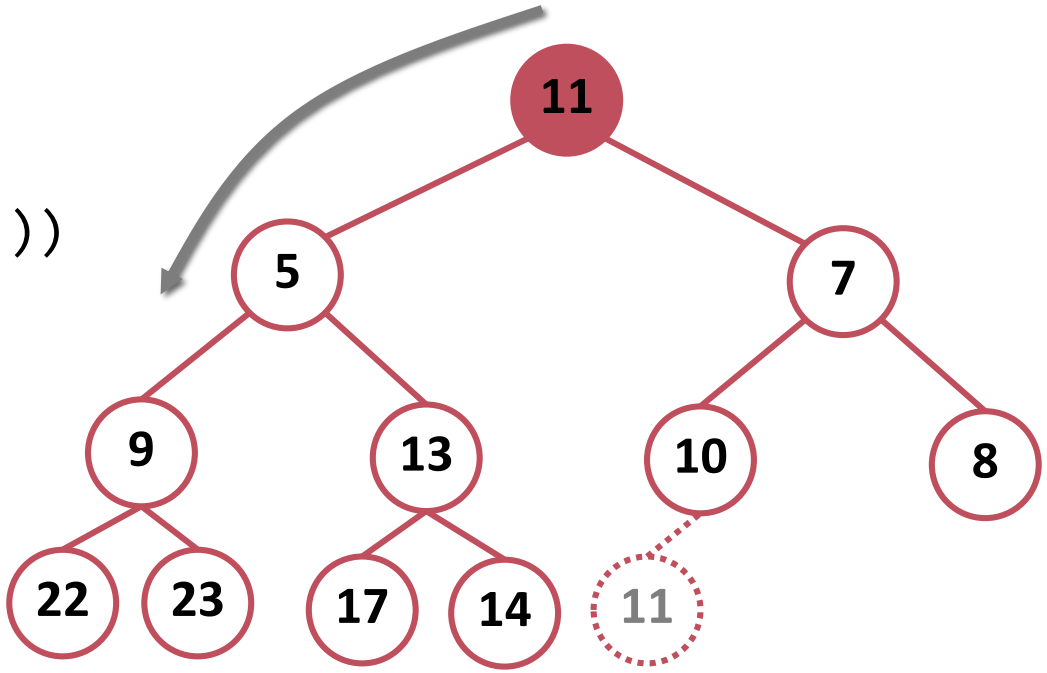






Wurzel Extrahieren

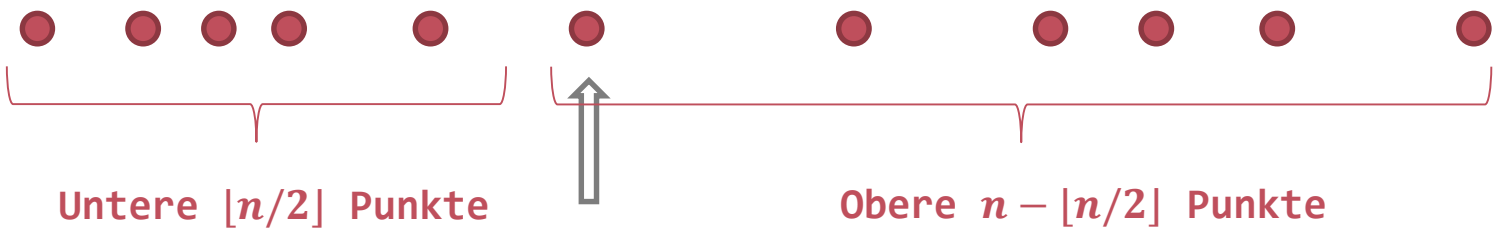
```
public double ExtractRoot() {
    double min = data[0];
    double value = data[used-1];
    int current = 0;
    int next = BestChild(current);
    while(next < used && !(value < data[next]))
    {
        data[current] = data[next];
        current=next;
        next = BestChild(current);
    }
    data[current] = value;
    used--;
    return min;
}
```



Verwendungsbeispiel Heap

Wir können das schnelle Auslesen, Extrahieren und Einfügen von Minimum (bzw. Maximum) im Heap für einen online-Algorithums des Median nutzen. Wie?

Beobachtung: Der Median bildet sich aus Minimum der oberen Hälfte der Daten und / oder Maximum der unteren Hälfte der Daten.



Online Median

Verwende Max-Heap H_{max} und Min-Heap H_{min}.
Bezeichne Anzahl Elemente jeweils mit $|H_{max}|$ und $|H_{min}|$

- Einfügen neuen Wertes v in
  - $H\_\max$, wenn $v \leq \max(H_{\max})$
  - $H\_\min$, sonst
- Rebalancieren der beiden Heaps
  - Falls $|H\_\max| > \lfloor n/2 \rfloor$, dann extrahiere Wurzel von $H\_\max$ und füge den Wert bei $H\_\min$ ein.
  - Falls $|H\_\max| < \lfloor n/2 \rfloor$, dann extrahiere Wurzel von H_min und füge den Wert bei H_max ein.

Gesamt worst-case Komplexität des Einfügens: $O(\log n)$

Berechnung Median

Berechnung Median

- Wenn n ungerade, dann

$$\text{median} = \min(H\_\text{min})$$

- Wenn n gerade, dann

$$\text{median} = \frac{\max(H_{\max}) + \min(H_{\min})}{2}$$

➔ worst-case Komplexität $O(1)$