

9. 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.

185

186

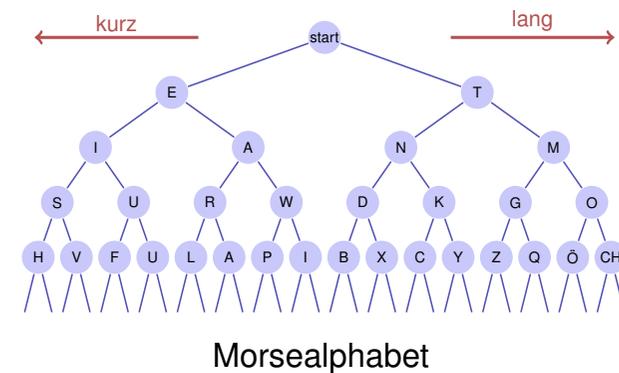
Bäume

Verwendung

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



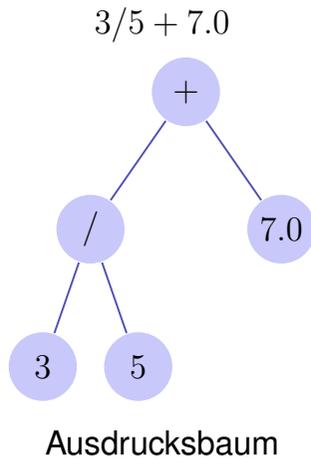
Beispiele



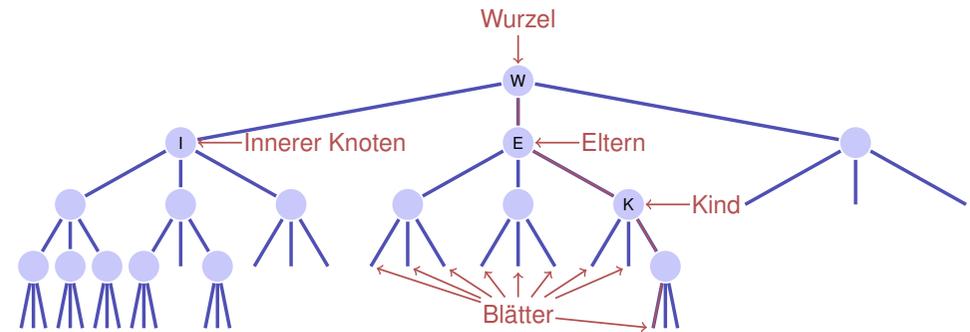
187

188

Beispiele



Nomenklatur



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

189

190

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.

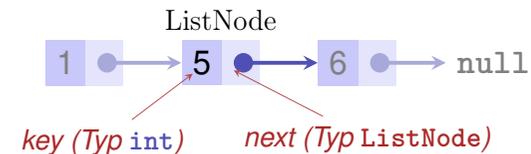
In jedem Knoten v speichern wir

key	
left	right

- einen Schlüssel $v.key$ und
- zwei Zeiger $v.left$ und $v.right$ auf die Wurzeln der linken und rechten Teilbäume.
- Ein Blatt wird durch den **null**-Zeiger repräsentiert

191

Zur Erinnerung: Listknoten in Java



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

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

192

Baumknoten in Java

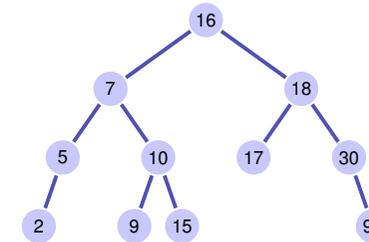
```
public class SearchNode {
    int key; // Schluessel
    SearchNode left; // linker Teilbaum
    SearchNode right; // rechter Teilbaum

    // 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 $v.left$ von v sind kleiner als $v.key$
- Schlüssel im rechten Teilbaum $v.right$ von v sind grösser als $v.key$



193

194

Suchen

Input : Binärer Suchbaum mit Wurzel r ,
Schlüssel k

Output : Knoten v mit $v.key = k$ oder **null**

$v \leftarrow r$

while $v \neq \text{null}$ **do**

if $k = v.key$ **then**

return v

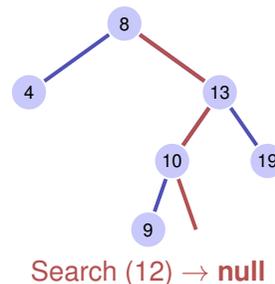
else if $k < v.key$ **then**

$v \leftarrow v.left$

else

$v \leftarrow v.right$

return null



195

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
}
```

196

Höhe eines Baumes

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

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

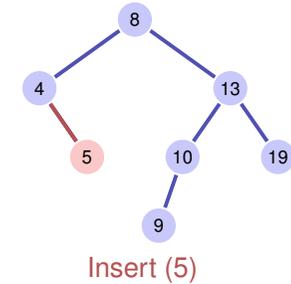
Die Laufzeit der Suche ist somit im schlechtesten Fall $\mathcal{O}(h(T))$

197

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



198

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; }
        }
    }
}
```

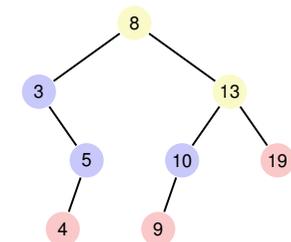
199

Knoten entfernen

Drei Fälle möglich

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

[Blätter zählen hier nicht]

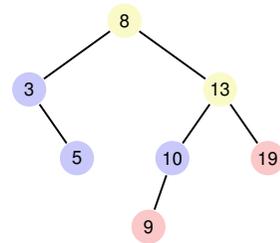
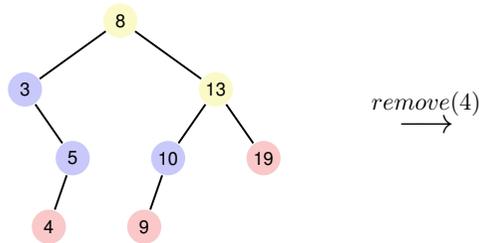


200

Knoten entfernen

Knoten hat keine Kinder

Einfacher Fall: Knoten durch Blatt ersetzen.

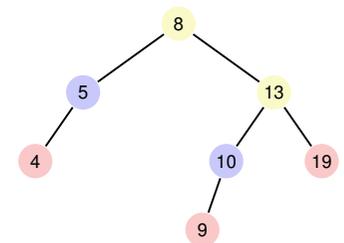
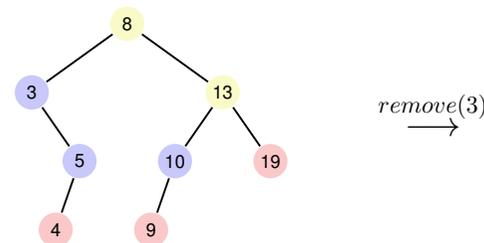


201

Knoten entfernen

Knoten hat ein Kind

Auch einfach: Knoten durch das einzige Kind ersetzen.



202

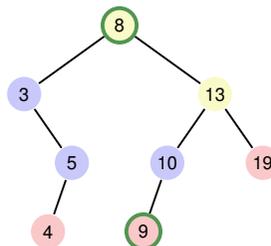
Knoten entfernen

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



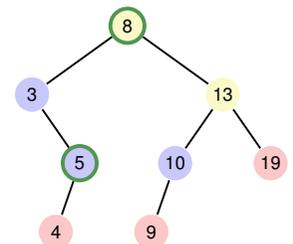
203

Aus Symmetriegründen...

Knoten v hat zwei Kinder

Auch möglich: ersetze v durch seinen symmetrischen Vorgänger

Implementation: der Teufel steckt im Detail!



204

Algorithmus SymmetricSuccessor(v)

Input : Knoten v eines binären Suchbaumes

Output : Symmetrischer Nachfolger von v

```
 $w \leftarrow v.\text{right}$   
 $x \leftarrow w.\text{left}$   
while  $x \neq \text{null}$  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.

205

206

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; }  
        }  
    }  
}
```

207

Analyse

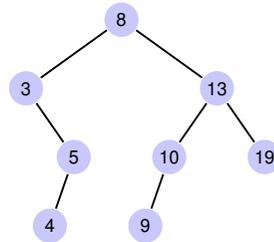
Löschen eines Elementes v aus einem Baum T benötigt $\mathcal{O}(h(T))$
Elementarschritte:

- Suchen von v hat Kosten $\mathcal{O}(h(T))$
- Hat v maximal ein Kind ungleich **null**, dann benötigt das Entfernen $\mathcal{O}(1)$
- Das Suchen des symmetrischen Nachfolgers n benötigt $\mathcal{O}(h(T))$ Schritte. Entfernen und Einfügen von n hat Kosten $\mathcal{O}(1)$

208

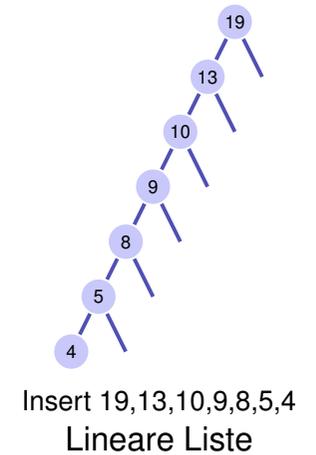
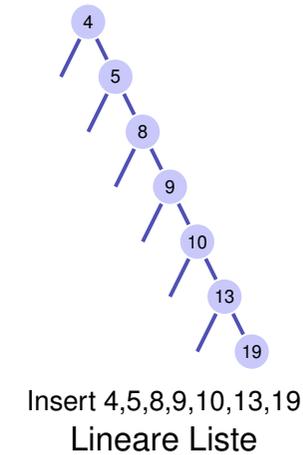
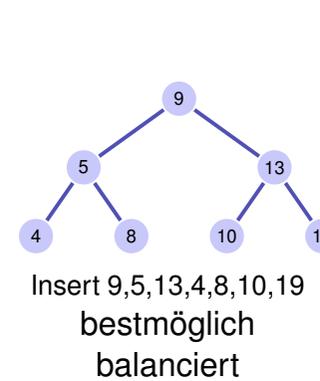
Traversierungsarten

- 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



209

Degenerierte Suchbäume



210

Probabilistisch

Ein Suchbaum, welcher aus einer zufälligen Sequenz von Zahlen erstellt wird hat erwartete Pfadlänge von $\mathcal{O}(\log n)$.

Achtung: das gilt nur für Einfügeoperation. Wird der Baum zufällig durch Einfügen und Entfernen gebildet, ist die erwartete Pfadlänge $\mathcal{O}(\sqrt{n})$.

Balancierte Bäume stellen beim Einfügen und Entfernen (z.B. durch *Rotationen*) sicher, dass der Baum balanciert bleibt und liefern eine $\mathcal{O}(\log n)$ Worst-Case-Garantie.

211

10. Heaps

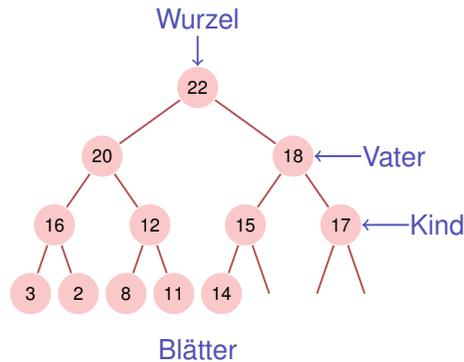
Datenstruktur optimiert zum schnellen Extrahieren von Minimum oder Maximum und Sortieren. [Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

212

[Max-]Heap⁶

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

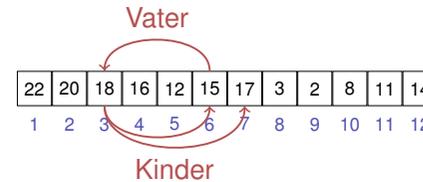


⁶Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

Heap und Array

Baum → Array:

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

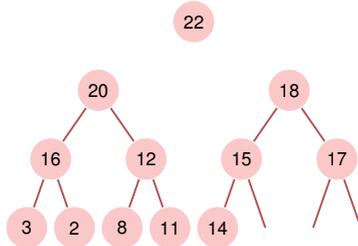


Abhängig von Startindex!⁷

⁷Für Arrays, die bei 0 beginnen: $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$

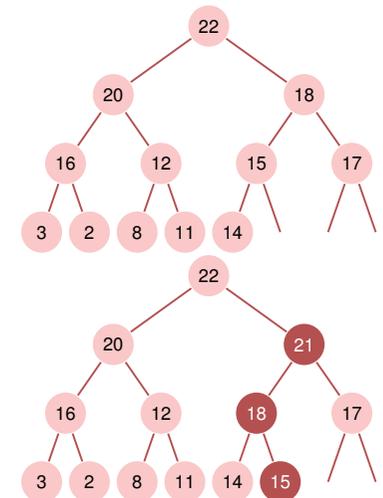
Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:



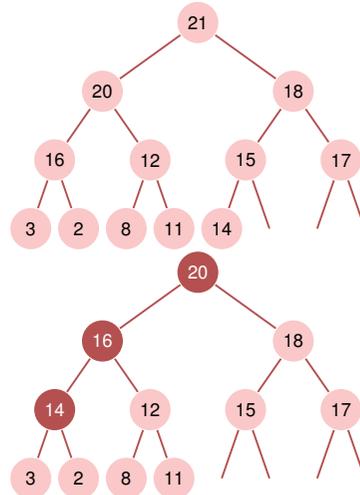
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



217

Algorithmus Versickern(A, i, m)

Input : Array A mit Heapstruktur für die Kinder von i . Letztes Element m .

Output : Array A mit Heapstruktur für i mit letztem Element m .

```

while  $2i \leq m$  do
   $j \leftarrow 2i$ ; //  $j$  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
    
```

218

Heap Sortieren

$A[1, \dots, n]$ 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	⇒	2	6	4	5	1	7
Versickern	⇒	6	5	4	2	1	7
Tauschen	⇒	1	5	4	2	6	7
Versickern	⇒	5	4	2	1	6	7
Tauschen	⇒	1	4	2	5	6	7
Versickern	⇒	4	1	2	5	6	7
Tauschen	⇒	2	1	4	5	6	7
Versickern	⇒	2	1	4	5	6	7
Tauschen	⇒	1	2	4	5	6	7

219

Heap erstellen

Beobachtung: Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

Folgerung: Induktion von unten!

220

Algorithmus HeapSort(A, n)

Input : Array A der Länge n .

Output : A sortiert.

// Heap Bauen.

for $i \leftarrow n/2$ **downto** 1 **do**

└ Versickere(A, i, n);

// Nun ist A ein Heap.

for $i \leftarrow n$ **downto** 2 **do**

└ swap($A[1], A[i]$)

└ Versickere($A, 1, i - 1$)

// Nun ist A sortiert.

Analyse: Sortieren eines Heaps

Versickere durchläuft maximal $\log n$ Knoten. An jedem Knoten 2 Schlüsselvergleiche. \Rightarrow Heap Sortieren kostet im schlechtesten Fall $2n \log n$ Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch $\mathcal{O}(n \log n)$.