

16. Natürliche Suchbäume

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

Wörterbuchimplementationen

Hashing: Implementierung von Wörterbüchern mit erwartet sehr schnellen Zugriffszeiten.

Nachteile von Hashing:

Wörterbuchimplementationen

Hashing: Implementierung von Wörterbüchern mit erwartet sehr schnellen Zugriffszeiten.

Nachteile von Hashing: im schlechtesten Fall lineare Zugriffszeit. **Manche Operationen gar nicht unterstützt:**

- Aufzählen von Schlüssel in aufsteigender Anordnung

Wörterbuchimplementationen

Hashing: Implementierung von Wörterbüchern mit erwartet sehr schnellen Zugriffszeiten.

Nachteile von Hashing: im schlechtesten Fall lineare Zugriffszeit. **Manche Operationen gar nicht unterstützt:**

- Aufzählen von Schlüssel in aufsteigender Anordnung
- Nächst kleinerer Schlüssel zu gegebenem Schlüssel

Wörterbuchimplementationen

Hashing: Implementierung von Wörterbüchern mit erwartet sehr schnellen Zugriffszeiten.

Nachteile von Hashing: im schlechtesten Fall lineare Zugriffszeit. **Manche Operationen gar nicht unterstützt:**

- Aufzählen von Schlüssel in aufsteigender Anordnung
- Nächst kleinerer Schlüssel zu gegebenem Schlüssel
- Schlüssel k in vorgegebenem Intervall $k \in [l, r]$

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.

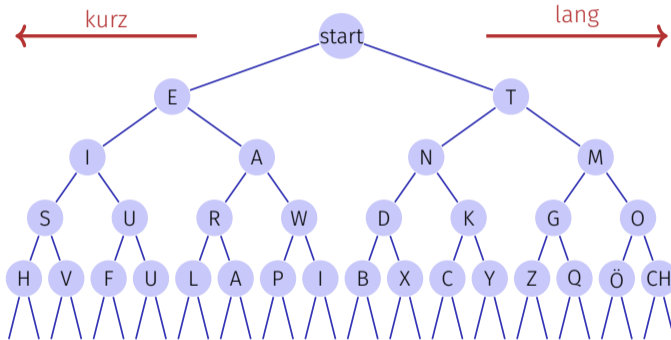
Bäume

Verwendung

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



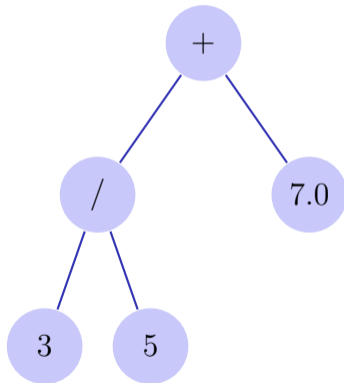
Beispiele



Morsealphabet

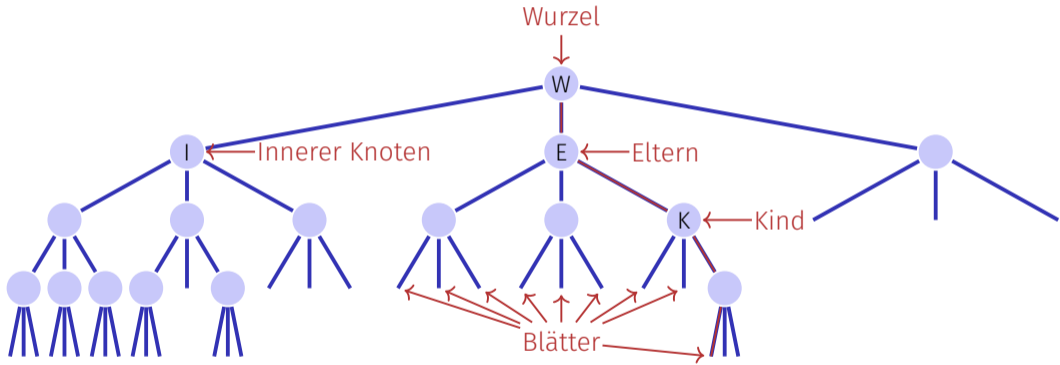
Beispiele

$3/5 + 7.0$



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 inneren Knoten v wird gespeichert

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

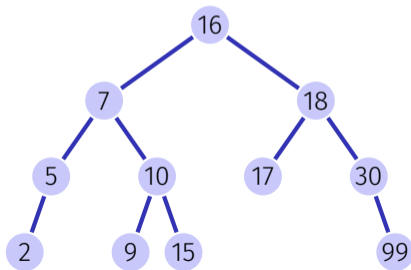


Ein Blatt wird durch den **null**-Zeiger repräsentiert

Binärer Suchbaum

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

- Jeder Knoten v speichert einen Schlüssel
- Schlüssel im linken Teilbaum $v.\text{left}$ kleiner als $v.\text{key}$
- Schlüssel im rechten Teilbaum $v.\text{right}$ grösser als $v.\text{key}$



Suchen

Input: Binärer Suchbaum mit Wurzel r ,
Schlüssel k

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

$v \leftarrow r$

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

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

 | **return** v

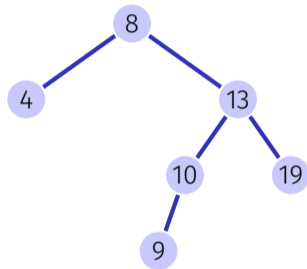
else if $k < v.\text{key}$ **then**

 | $v \leftarrow v.\text{left}$

else

 | $v \leftarrow v.\text{right}$

return null



Suchen

Input: Binärer Suchbaum mit Wurzel r ,
Schlüssel k

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

$v \leftarrow r$

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

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

 | **return** v

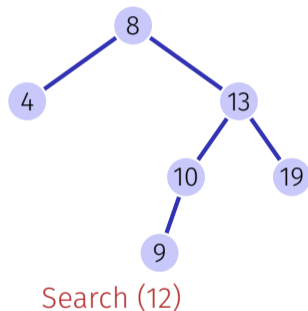
else if $k < v.key$ **then**

 | $v \leftarrow v.left$

else

 | $v \leftarrow v.right$

return null



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

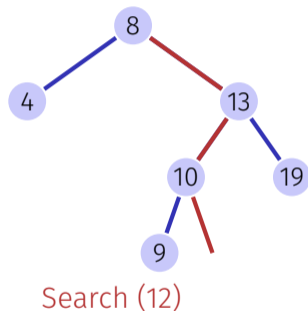
else if $k < v.key$ **then**

 | $v \leftarrow v.left$

else

 | $v \leftarrow v.right$

return null



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

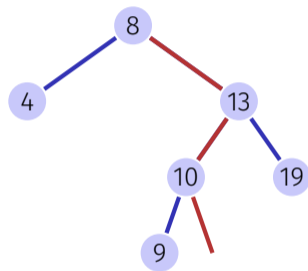
else if $k < v.key$ **then**

 | $v \leftarrow v.left$

else

 | $v \leftarrow v.right$

return null



Search (12) \rightarrow **null**

Höhe eines Baumes

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

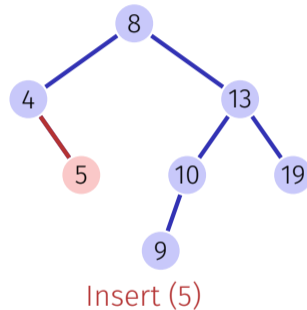
$$h(r) = \begin{cases} 0 & \text{falls } r = \mathbf{null} \\ 1 + \max\{h(r.\text{left}), h(r.\text{right})\} & \text{sonst.} \end{cases}$$

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

Einfügen eines Schlüssels

Einfügen des Schlüssels k

- Suche nach k .
- Wenn erfolgreich: z.B. Fehlerausgabe
- Wenn erfolglos: Einfügen des Schlüssels am erreichten Blatt.

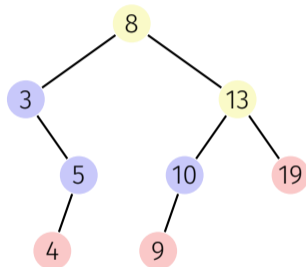


Knoten entfernen

Drei Fälle möglich

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

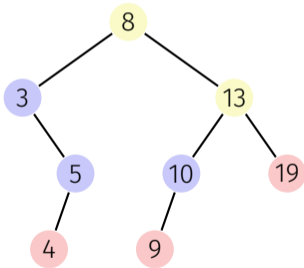
[Blätter zählen hier nicht]



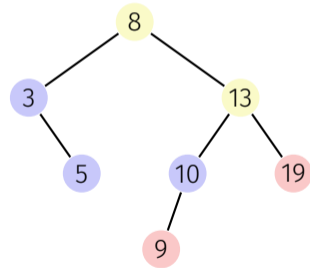
Knoten entfernen

Knoten hat keine Kinder

Einfacher Fall: Knoten durch Blatt ersetzen.



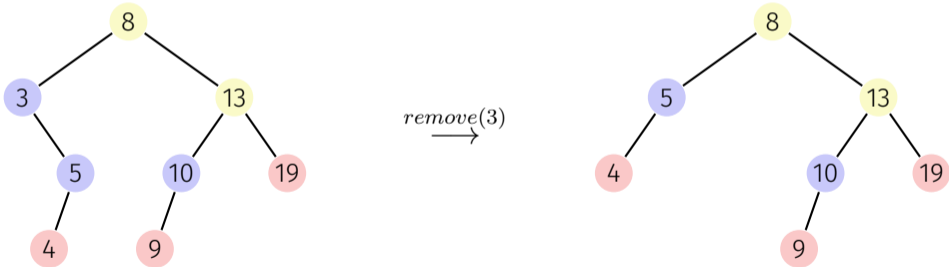
remove(4)
→



Knoten entfernen

Knoten hat ein Kind

Auch einfach: Knoten durch das einzige Kind ersetzen.



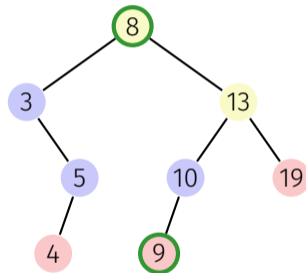
Knoten entfernen

Knoten v hat zwei Kinder

Beobachtung: Der kleinste Schlüssel im rechten Teilbaum $v.\text{right}$ (der **symmetrische Nachfolger** von v)

- ist kleiner als alle Schlüssel in $v.\text{right}$
- ist grösser als alle Schlüssel in $v.\text{left}$
- und hat kein linkes Kind.

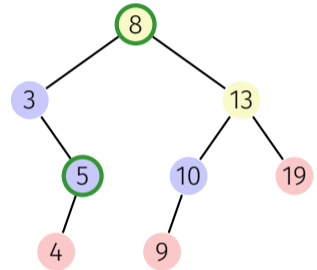
Lösung: ersetze v durch seinen symmetrischen Nachfolger



Aus Symmetriegründen...

Knoten v hat zwei Kinder

Auch möglich: ersetze v durch seinen symmetrischen Vorgänger



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**

$w \leftarrow x$
 $x \leftarrow x.\text{left}$

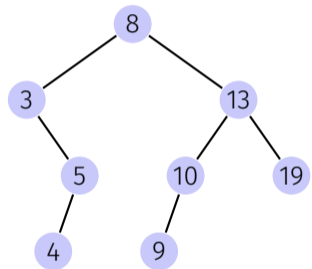
return w

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 $\mathcal{O}(1)$

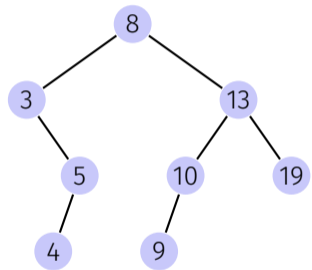
Traversierungsarten

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



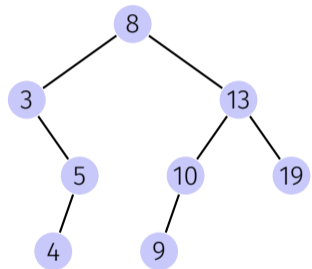
Traversierungsarten

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



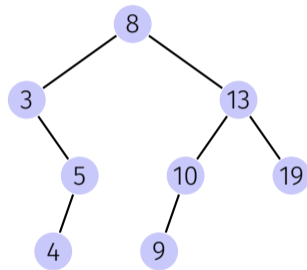
Traversierungsarten

- 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 .



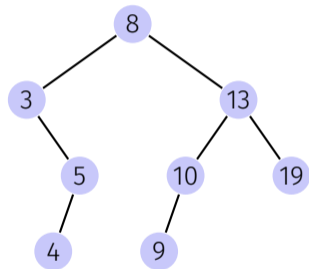
Traversierungsarten

- 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



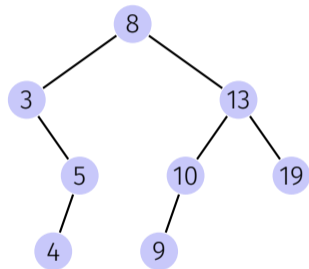
Traversierungsarten

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



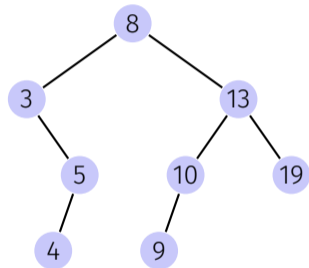
Traversierungsarten

- 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

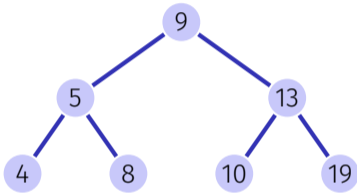


Weitere unterstützte Operationen

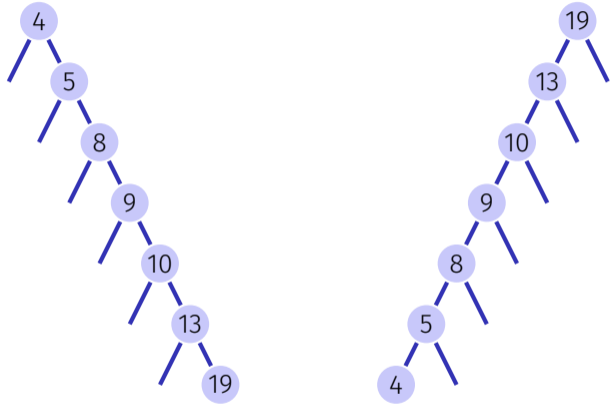
- $\text{Min}(T)$: Auslesen des Minimums in $\mathcal{O}(h)$
- $\text{ExtractMin}(T)$: Auslesen und Entfernen des Minimums in $\mathcal{O}(h)$
- $\text{List}(T)$: Ausgeben einer sortierten Liste der Elemente von T
- $\text{Join}(T_1, T_2)$: Zusammenfügen zweier Bäume mit $\max(T_1) < \min(T_2)$ in $\mathcal{O}(n)$.



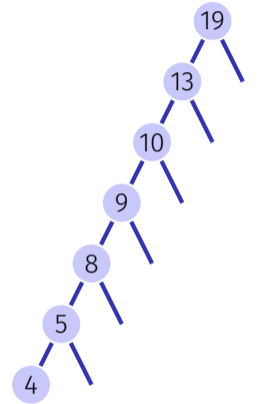
Degenerierte Suchbäume



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

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

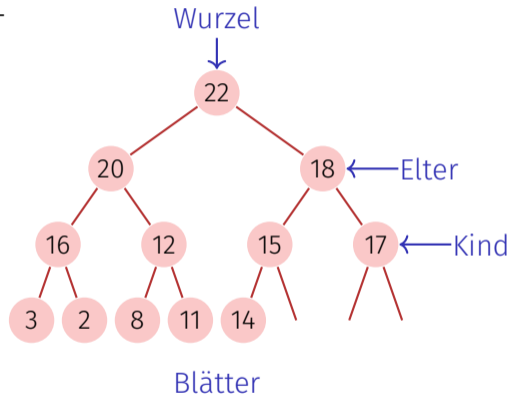
Balancierte Bäume stellen beim Einfügen und Entfernen (z.B. durch *Rotationen*) sicher, dass der Baum balanciert bleibt und liefern eine $\mathcal{O}(\log n)$ Worst-Case-Garantie.

17. Heaps

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

[Max-]Heap*

Binärer Baum mit folgenden Eigenschaften

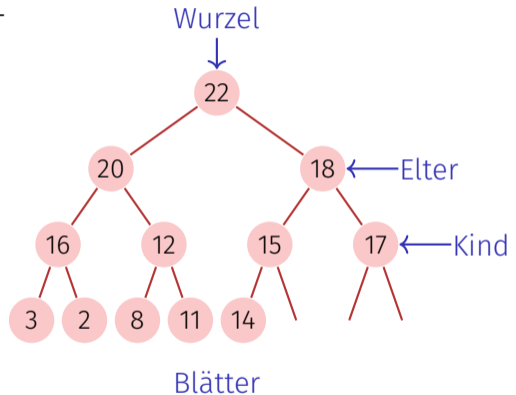


*Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

[Max-]Heap*

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene

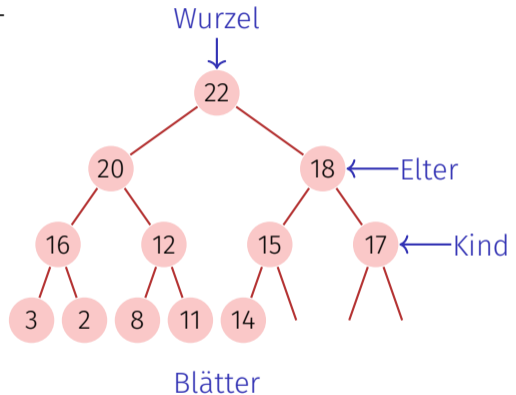


*Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

[Max-]Heap*

Binärer Baum mit folgenden Eigenschaften

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



*Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

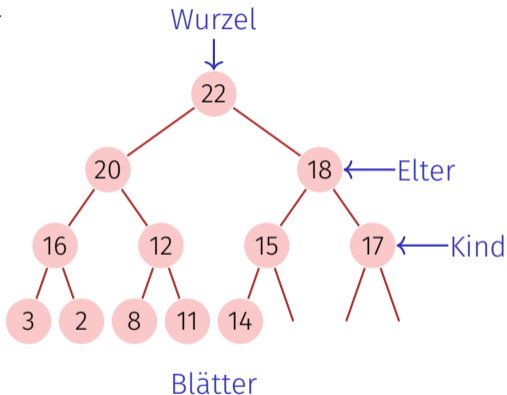
[Max-]Heap*

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 Elternknotens

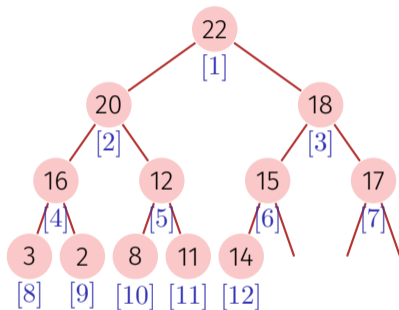
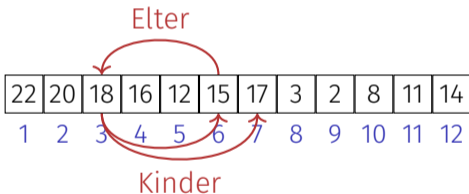


*Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

Heap als Array

Baum \rightarrow Array:

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



Abhängig von Startindex!²⁴

²⁴Für Arrays, die bei 0 beginnen: $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$

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.

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

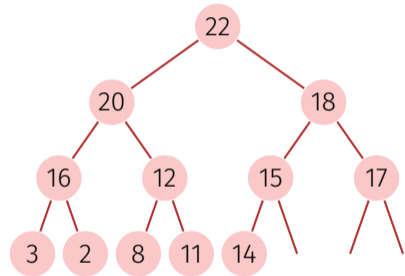
Mit $\sum_{i=0}^{h-1} 2^i = 2^h - 1$:

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

also

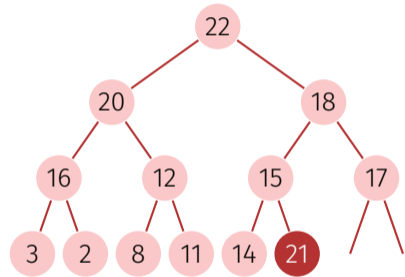
$$H(n) = \lceil \log_2(n + 1) \rceil.$$

Einfügen



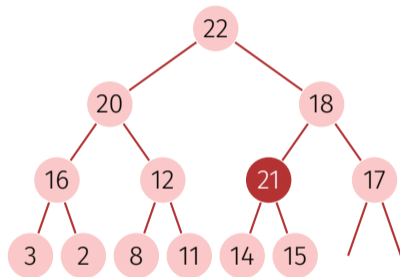
Einfügen

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



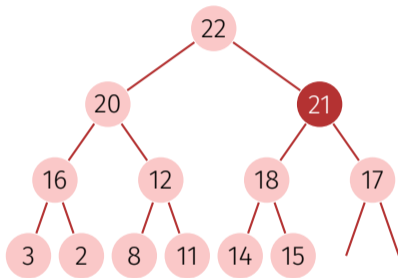
Einfügen

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



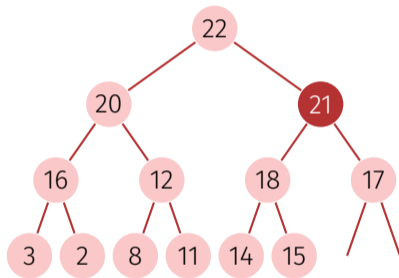
Einfügen

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



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



Algorithmus Aufsteigen(A, m)

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

Output: Array A mit Max-Heap-Struktur auf $A[1, \dots, m]$.

$v \leftarrow A[m]$ // Wert

$c \leftarrow m$ // derzeitiger Knoten (child)

$p \leftarrow \lfloor c/2 \rfloor$ // Elternknoten (parent)

while $c > 1$ and $v > A[p]$ **do**

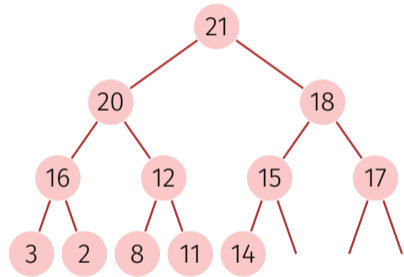
$A[c] \leftarrow A[p]$ // Wert Elternknoten \rightarrow derzeitiger Knoten

$c \leftarrow p$ // Elternknoten \rightarrow derzeitiger Knoten

$p \leftarrow \lfloor c/2 \rfloor$

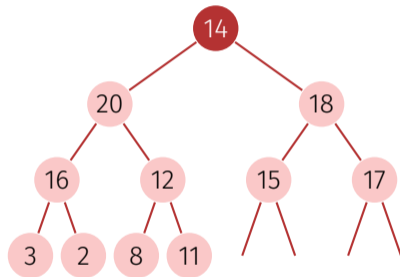
$A[c] \leftarrow v$ // Wert \rightarrow Wurzel des (Teil-)Baumes

Maximum entfernen



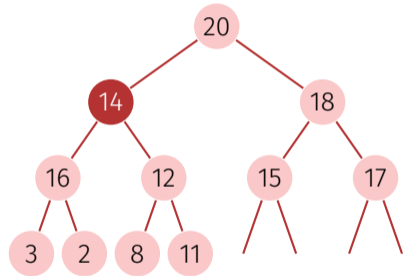
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.



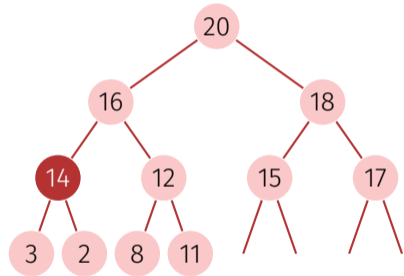
Maximum entfernen

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



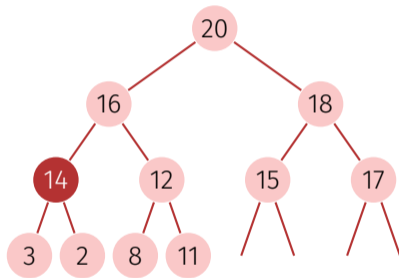
Maximum entfernen

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



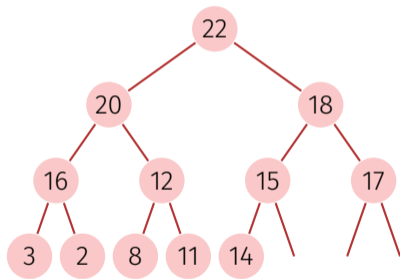
Maximum entfernen

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



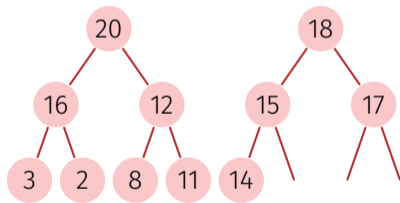
Warum das korrekt ist: Rekursive Heap-Struktur

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 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

Heap Sortieren



$A[1, \dots, n]$ ist Heap.

Solange $n > 1$

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

Heap Sortieren

Tauschen \Rightarrow

7	6	4	5	1	2
2	6	4	5	1	7

$A[1, \dots, n]$ ist Heap.

Solange $n > 1$

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

Heap Sortieren

$A[1, \dots, n]$ ist Heap.

Solange $n > 1$

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

Tauschen \Rightarrow

7	6	4	5	1	2
---	---	---	---	---	---

2	6	4	5	1	7
---	---	---	---	---	---

Versickern \Rightarrow

6	5	4	2	1	7
---	---	---	---	---	---

Heap Sortieren

$A[1, \dots, n]$ ist Heap.

Solange $n > 1$

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

Tauschen \Rightarrow

7	6	4	5	1	2
---	---	---	---	---	---

2	6	4	5	1	7
---	---	---	---	---	---

Versickern \Rightarrow

6	5	4	2	1	7
---	---	---	---	---	---

Tauschen \Rightarrow

1	5	4	2	6	7
---	---	---	---	---	---

Heap Sortieren

$A[1, \dots, n]$ ist Heap.

Solange $n > 1$

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

		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

Heap erstellen

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

Folgerung:

Heap erstellen

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

Folgerung: Induktion von unten!

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

Analyse: Heap bauen

Aufrufe an Versickern: $n/2$.

Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

Versickerpfade sind aber im Mittel viel kürzer:

Wir verwenden, dass $h(n) = \lceil \log_2 n + 1 \rceil = \lfloor \log_2 n \rfloor + 1$ für $n > 0$

$$\begin{aligned} v(n) &= \sum_{l=0}^{\lfloor \log_2 n \rfloor} \underbrace{2^l}_{\text{Anzahl Heaps auf Level } l} \cdot \underbrace{(\lfloor \log_2 n \rfloor + 1 - l - 1)}_{\text{Höhe Heaps auf Level } l} = \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^{\lfloor \log_2 n \rfloor - k} \cdot k \\ &= 2^{\lfloor \log_2 n \rfloor} \cdot \sum_{k=0}^{\lfloor \log_2 n \rfloor} \frac{k}{2^k} \leq n \cdot \sum_{k=0}^{\infty} \frac{k}{2^k} \leq n \cdot 2 \in \mathcal{O}(n) \end{aligned}$$

mit $s(x) := \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ ($0 < x < 1$) und $s(\frac{1}{2}) = 2$

Nachteile

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

Nachteile von Heapsort?

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

Nachteile von Heapsort?

- ⚠️ Wenig Lokalität: per Definition springt Heapsort im sortierten Array umher (Negativer Cache Effekt).

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

Nachteile von Heapsort?

- ⚠️ Wenig Lokalität: per Definition springt Heapsort im sortierten Array umher (Negativer Cache Effekt).
- ⚠️ Zwei Vergleiche vor jeder benötigten Bewegung.

18. AVL Bäume

Balancierte Bäume [Ottman/Widmayer, Kap. 5.2-5.2.1, Cormen et al, Kap. Problem 13-3]

Ziel

Suchen, Einfügen und Entfernen eines Schlüssels in Baum mit n Schlüssel, welche in zufälliger Reihenfolge eingefügt wurden im Mittel in $\mathcal{O}(\log_2 n)$ Schritten.

Schlechtester Fall jedoch: $\Theta(n)$ (degenerierter Baum).

Ziel: Verhinderung der Degenerierung. Künstliches, bei jeder Update-Operation erfolgtes Balancieren eines Baumes

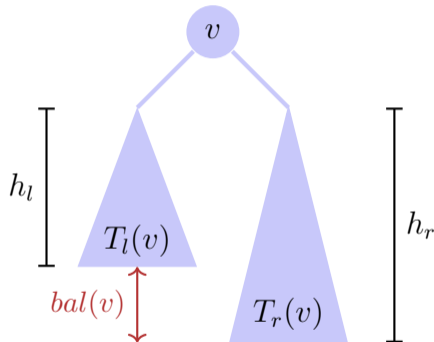
Balancierung: garantiere, dass ein Baum mit n Knoten stets eine Höhe von $\mathcal{O}(\log n)$ hat.

Adelson-Venskii und Landis (1962): AVL-Bäume

Balance eines Knotens

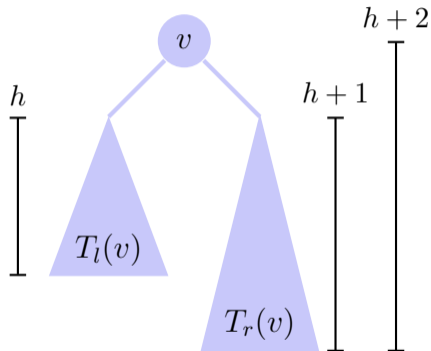
Die **Balance** eines Knotens v ist definiert als die Höhendifferenz seiner beiden Teilbäume $T_l(v)$ und $T_r(v)$

$$\text{bal}(v) := h(T_r(v)) - h(T_l(v))$$

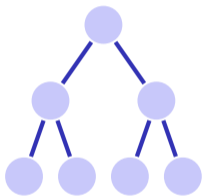


AVL Bedingung

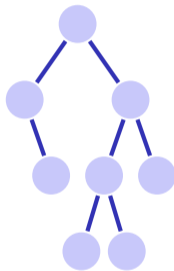
AVL Bedingung: für jeden Knoten v eines Baumes gilt $\text{bal}(v) \in \{-1, 0, 1\}$



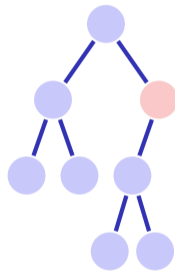
(Gegen-)Beispiele



AVL Baum der Höhe 2



AVL Baum der Höhe 3

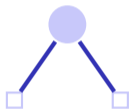


Kein AVL Baum

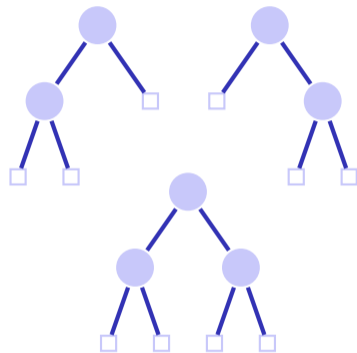
Anzahl Blätter

- 1. Beobachtung: Ein Suchbaum mit n Schlüsseln hat genau $n + 1$ Blätter. Einfaches Induktionsargument.
 - Der Suchbaum mit $n = 0$ Schlüsseln hat $m = 1$ Blätter
 - Wird ein Schlüssel (Knoten) hinzugefügt ($n \rightarrow n + 1$), so ersetzt er ein Blatt und fügt zwei Blätter hinzu ($m \rightarrow m - 1 + 2 = m + 1$).
- 2. Beobachtung: untere Grenze für Anzahl Blätter eines Suchbaums zu gegebener Höhe erlaubt Abschätzung der maximalen Höhe eines Suchbaums zu gegebener Anzahl Schlüssel.

Untere Grenze Blätter



AVL Baum der Höhe 1 hat
 $N(1) := 2$ Blätter



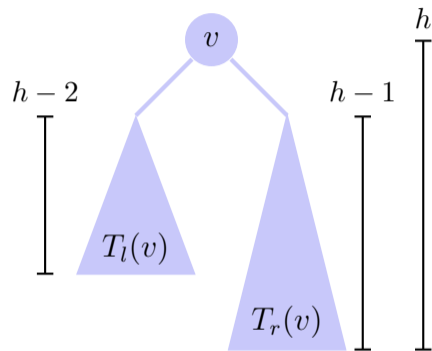
AVL Baum der Höhe 2 hat
mindestens $N(2) := 3$ Blätter

Untere Grenze Blätter für $h > 2$

- Höhe eines Teilbaums $\geq h - 1$.
- Höhe des anderen Teilbaums $\geq h - 2$.

Minimale Anzahl Blätter $N(h)$ ist

$$N(h) = N(h - 1) + N(h - 2)$$



Insgesamt gilt $N(h) = F_{h+2}$ mit **Fibonacci-Zahlen** $F_0 := 0$, $F_1 := 1$,
 $F_n := F_{n-1} + F_{n-2}$ für $n > 1$.

Fibonacci Zahlen, geschlossene Form

Es gilt²⁵

$$F_i = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)$$

mit den Wurzeln $\phi, \hat{\phi}$ der Gleichung vom goldenen Schnitt $x^2 - x - 1 = 0$:

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.618$$

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.618$$

²⁵Herleitung mit Erzeugendenfunktionen (Potenzreihen) im Anhang

Fibonacci Zahlen, Induktiver Beweis

$$F_i \stackrel{!}{=} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i) \quad [*] \quad \left(\phi = \frac{1+\sqrt{5}}{2}, \hat{\phi} = \frac{1-\sqrt{5}}{2}\right).$$

1. Klar für $i = 0, i = 1$.
2. Sei $i > 2$ und Behauptung $[*]$ wahr für alle $F_j, j < i$.

$$\begin{aligned} F_i &\stackrel{\text{def}}{=} F_{i-1} + F_{i-2} \stackrel{[*]}{=} \frac{1}{\sqrt{5}}(\phi^{i-1} - \hat{\phi}^{i-1}) + \frac{1}{\sqrt{5}}(\phi^{i-2} - \hat{\phi}^{i-2}) \\ &= \frac{1}{\sqrt{5}}(\phi^{i-1} + \phi^{i-2}) - \frac{1}{\sqrt{5}}(\hat{\phi}^{i-1} + \hat{\phi}^{i-2}) = \frac{1}{\sqrt{5}}\phi^{i-2}(\phi + 1) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi} + 1) \end{aligned}$$

$(\phi, \hat{\phi}$ erfüllen $x + 1 = x^2$)

$$= \frac{1}{\sqrt{5}}\phi^{i-2}(\phi^2) - \frac{1}{\sqrt{5}}\hat{\phi}^{i-2}(\hat{\phi}^2) = \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i).$$

Baumhöhe

Da $|\hat{\phi}| < 1$, gilt insgesamt

$$N(h) \in \Theta\left(\left(\frac{1 + \sqrt{5}}{2}\right)^h\right) \subseteq \Omega(1.618^h)$$

und somit

$$\begin{aligned} N(h) &\geq c \cdot 1.618^h \\ \Rightarrow h &\leq 1.44 \log_2 n + c'. \end{aligned}$$

Ein AVL Baum ist asymptotisch nicht mehr als 44% höher als ein perfekt balancierter Baum.²⁶

²⁶Ein perfekt balancierter Baum hat Höhe $\lceil \log_2 n + 1 \rceil$

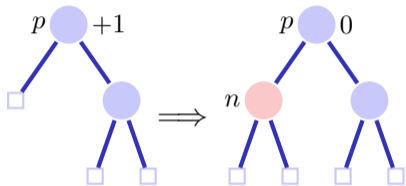
Balancieren

- Speichern der Balance für jeden Knoten
- Baum rebalancieren bei jeder Update-Operation

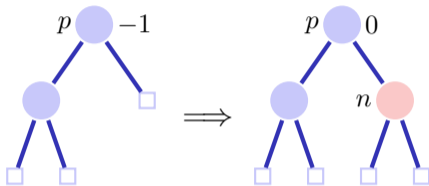
Neuer Knoten n wird eingefügt:

- Zuerst einfügen wie bei Suchbaum.
- Prüfe die Balance-Bedingung für alle Knoten aufsteigend von n zur Wurzel.

Balance am Einfügeort



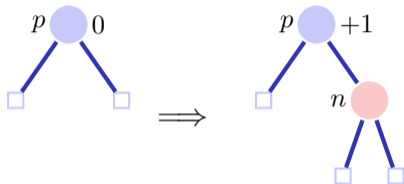
Fall 1: $\text{bal}(p) = +1$



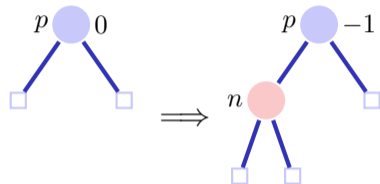
Fall 2: $\text{bal}(p) = -1$

Fertig in beiden Fällen, denn der Teilbaum ist nicht gewachsen.

Balance am Einfügeort



Fall 3.1: $\text{bal}(p) = 0$ rechts



Fall 3.2: $\text{bal}(p) = 0$, links

In beiden Fällen noch nicht fertig. Aufruf von **upin(p)**.

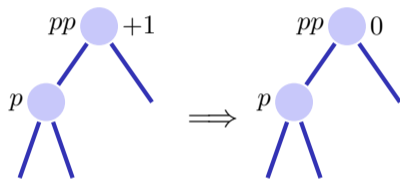
upin(p) - Invariante

Beim Aufruf von **upin(p)** gilt, dass

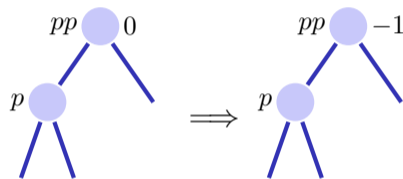
- der Teilbaum ab p gewachsen ist und
- $\text{bal}(p) \in \{-1, +1\}$

upin(p)

Annahme: p ist linker Sohn von pp^{27}



Fall 1: $\text{bal}(pp) = +1$, fertig.



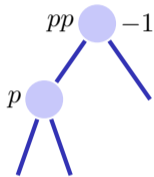
Fall 2: $\text{bal}(pp) = 0$, **upin(pp)**

In beiden Fällen gilt nach der Operation die AVL-Bedingung für den Teilbaum ab pp

²⁷Ist p rechter Sohn: symmetrische Fälle unter Vertauschung von +1 und -1

upin(p)

Annahme: p ist linker Sohn von pp



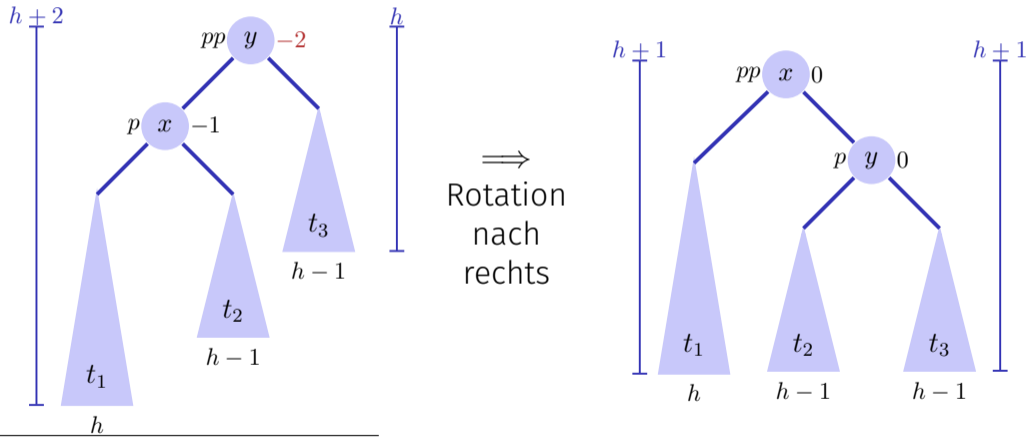
Fall 3: $\text{bal}(pp) = -1,$

Dieser Fall ist problematisch: das Hinzufügen von n im Teilbaum ab pp hat die AVL-Bedingung verletzt. Rebalancieren!

Zwei Fälle $\text{bal}(p) = -1, \text{bal}(p) = +1$

Rotationen

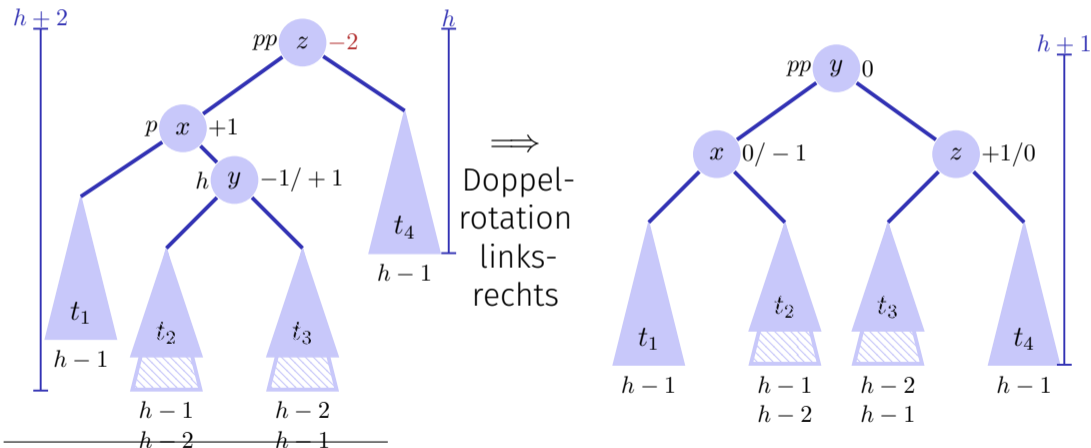
Fall 1.1 $\text{bal}(p) = -1$.²⁸



²⁸ p rechter Sohn $\Rightarrow \text{bal}(pp) = \text{bal}(p) = +1$, Linksrotation

Rotationen

Fall 1.2 $\text{bal}(p) = +1$.²⁹



²⁹ p rechter Sohn $\Rightarrow \text{bal}(pp) = +1, \text{bal}(p) = -1$, Doppelrotation rechts links

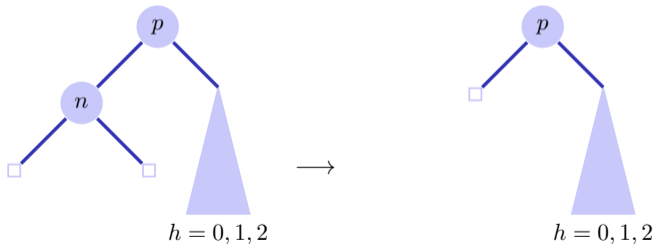
- Höhe des Baumes: $\mathcal{O}(\log n)$.
- Einfügen wie beim binären Suchbaum.
- Balancieren durch Rekursion vom Knoten zur Wurzel. Maximale Pfadlänge $\mathcal{O}(\log n)$.

Das Einfügen im AVL-Baum hat Laufzeitkosten von $\mathcal{O}(\log n)$.

Löschen

Fall 1: Knoten n hat zwei Blätter als Kinder Sei p Elternknoten von n . \Rightarrow Anderer Teilbaum hat Höhe $h' = 0, 1$ oder 2

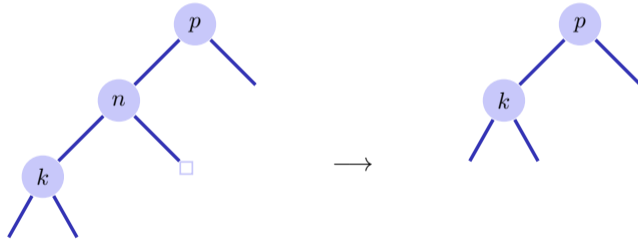
- $h' = 1$: $\text{bal}(p)$ anpassen.
- $h' = 0$: $\text{bal}(p)$ anpassen. Aufruf **upout**(p).
- $h' = 2$: Rebalancieren des Teilbaumes. Aufruf **upout**(p).



Löschen

Fall 2: Knoten n hat einen inneren Knoten k als Kind

- Ersetze n durch k . **upout(k)**



Fall 3: Knoten n hat zwei inneren Knoten als Kinder

- Ersetze n durch symmetrischen Nachfolger. **upout(k)**
- Löschen des symmetrischen Nachfolgers wie in Fall 1 oder 2.

upout (p)

Sei pp der Elternknoten von p

(a) p linkes Kind von pp

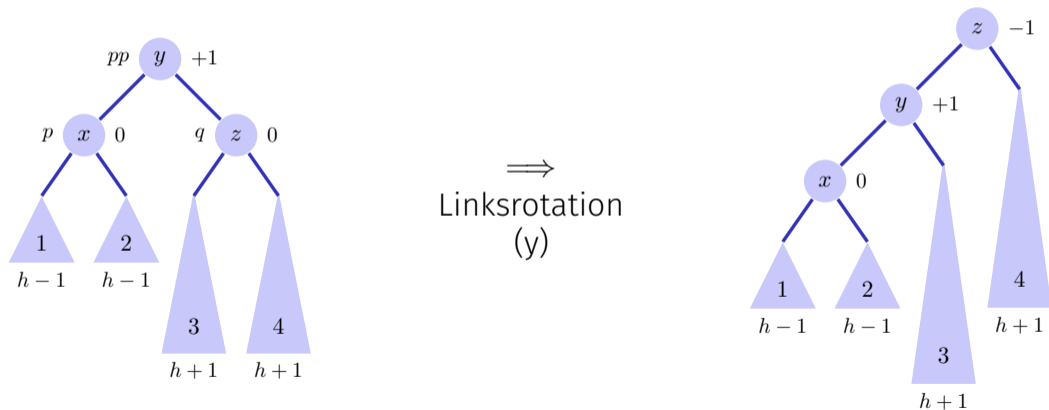
1. $\text{bal}(pp) = -1 \Rightarrow \text{bal}(pp) \leftarrow 0$. **upout (pp)**
2. $\text{bal}(pp) = 0 \Rightarrow \text{bal}(pp) \leftarrow +1$.
3. $\text{bal}(pp) = +1 \Rightarrow$ nächste Folien.

(b) p rechtes Kind von pp : Symmetrische Fälle unter Vertauschung von $+1$ und -1 .

upout (p)

Fall (a).3: $\text{bal}(pp) = +1$. Sei q Bruder von p

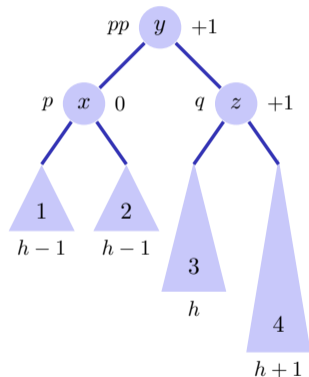
(a).3.1: $\text{bal}(q) = 0$.³⁰



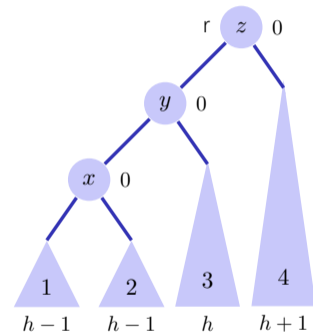
³⁰(b).3.1: $\text{bal}(pp) = -1$, $\text{bal}(q) = -1$, Rechtsrotation.

upout (p)

Fall (a).3: $\text{bal}(pp) = +1$. (a).3.2: $\text{bal}(q) = +1$.³¹



\Rightarrow
Linksrotation
(y)

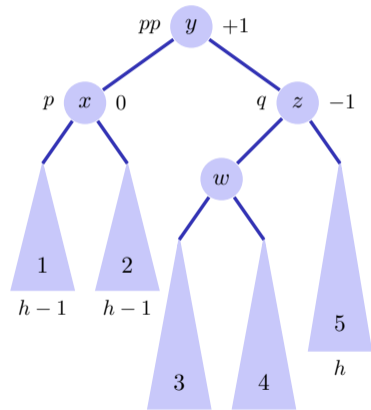


plus **upout (r)**.

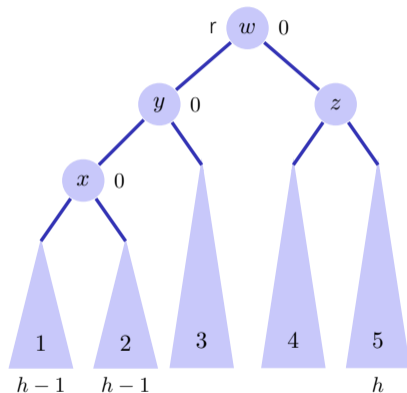
³¹(b).3.2: $\text{bal}(pp) = -1$, $\text{bal}(q) = +1$, Rechtsrotation+upout

upout (p)

Fall (a).3: $\text{bal}(pp) = +1$. (a).3.3: $\text{bal}(q) = -1$.³²



\Rightarrow
Doppelrotation
rechts (z) links
(y)



plus **upout (r)**.

³²(b).3.3: $\text{bal}(pp) = -1$, $\text{bal}(q) = -1$, Links-Rechts-Rotation + upout

- AVL-Bäume haben asymptotische Laufzeit von $\mathcal{O}(\log n)$ (schlechtester Fall) für das Suchen, Einfügen und Löschen von Schlüsseln
- Einfügen und Löschen ist verhältnismässig aufwändig und für kleine Probleme relativ langsam.

18.5 Anhang

Herleitung einiger mathematischen Formeln

[Fibonacci Zahlen: geschlossene Form]

Geschlossene Form der Fibonacci Zahlen: Berechnung über erzeugende Funktionen:

1. Potenzreihenansatz

$$f(x) := \sum_{i=0}^{\infty} F_i \cdot x^i$$

[Fibonacci Zahlen: geschlossene Form]

2. Für Fibonacci Zahlen gilt $F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2} \forall i > 1$. Daher:

$$\begin{aligned}f(x) &= x + \sum_{i=2}^{\infty} F_i \cdot x^i = x + \sum_{i=2}^{\infty} F_{i-1} \cdot x^i + \sum_{i=2}^{\infty} F_{i-2} \cdot x^i \\&= x + x \sum_{i=2}^{\infty} F_{i-1} \cdot x^{i-1} + x^2 \sum_{i=2}^{\infty} F_{i-2} \cdot x^{i-2} \\&= x + x \sum_{i=0}^{\infty} F_i \cdot x^i + x^2 \sum_{i=0}^{\infty} F_i \cdot x^i \\&= x + x \cdot f(x) + x^2 \cdot f(x).\end{aligned}$$

[Fibonacci Zahlen: geschlossene Form]

3. Damit:

$$f(x) \cdot (1 - x - x^2) = x.$$
$$\Leftrightarrow f(x) = \frac{x}{1 - x - x^2} = -\frac{x}{x^2 + x - 1}$$

Mit den Wurzeln $-\phi$ und $-\hat{\phi}$ von $x^2 + x - 1$,

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.6, \quad \hat{\phi} = \frac{1 - \sqrt{5}}{2} \approx -0.6.$$

gilt $\phi \cdot \hat{\phi} = -1$ und somit

$$f(x) = -\frac{x}{(x + \phi) \cdot (x + \hat{\phi})} = \frac{x}{(1 - \phi x) \cdot (1 - \hat{\phi} x)}$$

[Fibonacci Zahlen: geschlossene Form]

4. Es gilt:

$$(1 - \hat{\phi}x) - (1 - \phi x) = \sqrt{5} \cdot x.$$

Damit:

$$\begin{aligned} f(x) &= \frac{1}{\sqrt{5}} \frac{(1 - \hat{\phi}x) - (1 - \phi x)}{(1 - \phi x) \cdot (1 - \hat{\phi}x)} \\ &= \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi x} - \frac{1}{1 - \hat{\phi}x} \right) \end{aligned}$$

[Fibonacci Zahlen: geschlossene Form]

5. Potenzreihenentwicklung von $g_a(x) = \frac{1}{1-a \cdot x}$ ($a \in \mathbb{R}$):

$$\frac{1}{1-a \cdot x} = \sum_{i=0}^{\infty} a^i \cdot x^i.$$

Sieht man mit Taylor-Entwicklung von $g_a(x)$ um $x = 0$ oder so: Sei $\sum_{i=0}^{\infty} G_i \cdot x^i$ eine Potenzreihenentwicklung von g . Mit der Identität $g_a(x)(1 - a \cdot x) = 1$ gilt für alle x (im Konvergenzradius)

$$1 = \sum_{i=0}^{\infty} G_i \cdot x^i - a \cdot \sum_{i=0}^{\infty} G_i \cdot x^{i+1} = G_0 + \sum_{i=1}^{\infty} (G_i - a \cdot G_{i-1}) \cdot x^i$$

Für $x = 0$ folgt $G_0 = 1$ und für $x \neq 0$ folgt dann $G_i = a \cdot G_{i-1} \Rightarrow G_i = a^i$.

[Fibonacci Zahlen: geschlossene Form]

6. Einsetzen der Potenzreihenentwicklung:

$$\begin{aligned} f(x) &= \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi x} - \frac{1}{1 - \hat{\phi} x} \right) = \frac{1}{\sqrt{5}} \left(\sum_{i=0}^{\infty} \phi^i x^i - \sum_{i=0}^{\infty} \hat{\phi}^i x^i \right) \\ &= \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^i - \hat{\phi}^i) x^i \end{aligned}$$

Koeffizientenvergleich mit $f(x) = \sum_{i=0}^{\infty} F_i \cdot x^i$ liefert

$$F_i = \frac{1}{\sqrt{5}} (\phi^i - \hat{\phi}^i).$$