4 Dynamische Mengen

4 Dynamische Mengen

Datenstruktur: Speichere Daten so, dass bestimmte Operationen effizient möglich sind.

Datenstruktur: Speichere Daten so, dass bestimmte Operationen effizient möglich sind.

Dynamische Menge

Eine dynamische Menge speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

Datenstruktur: Speichere Daten so, dass bestimmte Operationen effizient möglich sind.

Dynamische Menge

Eine dynamische Menge speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- SEARCH(k)**:** Testet, ob ein Element mit dem Schlüssel k in der Menge vorhanden ist und gibt dieses gegebenenfalls zurück.

Datenstruktur: Speichere Daten so, dass bestimmte Operationen effizient möglich sind.

Dynamische Menge

Eine dynamische Menge speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- SEARCH(k)**:** Testet, ob ein Element mit dem Schlüssel k in der Menge vorhanden ist und gibt dieses gegebenenfalls zurück.
- INSERT(x)**:** Fügt das Element x unter dem Schlüssel x.key in die Menge ein.

Datenstruktur: Speichere Daten so, dass bestimmte Operationen effizient möglich sind.

Dynamische Menge

Eine dynamische Menge speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- SEARCH(k)**:** Testet, ob ein Element mit dem Schlüssel k in der Menge vorhanden ist und gibt dieses gegebenenfalls zurück.
- INSERT(x)**:** Fügt das Element x unter dem Schlüssel x.key in die Menge ein.
- DELETE(k)**:** Löscht das Element mit dem Schlüssel k aus der Menge.

Implementierung als verkettete Liste:

Speichere Daten (unsortiert) in einer verketteten Liste.

Implementierung als verkettete Liste:

Speichere Daten (unsortiert) in einer verketteten Liste.

Sei n die Anzahl an Daten.

- SEARCH(k)**:** Laufzeit $O(n)$

Implementierung als verkettete Liste:

Speichere Daten (unsortiert) in einer verketteten Liste.

Sei n die Anzahl an Daten.

- SEARCH(k)**:** Laufzeit $O(n)$
- INSERT(x)**:** Laufzeit $O(1)$ (falls nicht auf Duplikat getestet wird)

Implementierung als verkettete Liste:

Speichere Daten (unsortiert) in einer verketteten Liste.

Sei n die Anzahl an Daten.

- SEARCH(k)**:** Laufzeit $O(n)$
- INSERT(x)**:** Laufzeit $O(1)$ (falls nicht auf Duplikat getestet wird)
- DELETE(k)**:** Laufzeit $O(n)$

Implementierung mithilfe eines Feldes:

Speichere Daten in einem Feld.

Implementierung mithilfe eines Feldes:

Speichere Daten in einem Feld.

Problem: Anzahl von Daten im Vorhinein nicht immer bekannt.

Schätze deshalb die Größe (size) und vergrößere das Feld wenn nötig.

Implementierung mithilfe eines Feldes:

Speichere Daten in einem Feld.

Problem: Anzahl von Daten im Vorhinein nicht immer bekannt.

Schätze deshalb die Größe (size) und vergrößere das Feld wenn nötig.

```
INIT(int size)
1    n = 0;
2    a = new Element[size];
```

Laufzeit: $O(1)$

Implementierung mithilfe eines Feldes:

Speichere Daten in einem Feld.

Problem: Anzahl von Daten im Vorhinein nicht immer bekannt.

Schätze deshalb die Größe (size) und vergrößere das Feld wenn nötig.

```
INIT(int size)
1    n = 0;
2    a = new Element[size];
```

Laufzeit: $O(1)$

```
SEARCH(int k)
1    for (int i = 0; i < n; i++)
2        if (a[i].key == k) return a[i];
3    return null;
```

Laufzeit: $O(n)$

```
INSERT(Element x)
 1  if (n == a.size) {
 2     Element[] b = new Element[2*a.size];
 3     b[0 ... a.size − 1] = a[0 ... a.size − 1];
 4     a = b;
 5  }
 6  a[n] = x;
 7  n++;
```

Laufzeit: $O(1)$ oder $O(n)$

```
INSERT(Element x)
 1 if (n == a.size) {
 2     Element[] b = new Element[2*a.size];
 3     b[0 ... a.size − 1] = a[0 ... a.size − 1];
 4     a = b;
 5 }
 6 a[n] = x;
 7 n++;
```

Laufzeit: $O(1)$ oder $O(n)$

```
DELETE(int k)
 1 for (int i = 0; i < n; i++)
 2     if (a[i].key == k) {
 3         for (int j = i; j < n − 1; j++)
 4             a[j] = a[j + 1];
 5         a[n − 1] =null;
 6         n−−;
 7         return;
 8     }
```

Laufzeit: $O(n)$

Lemma 4.1

Die amortisierte Laufzeit von INSERT beträgt in einem dynamischen Feld $O(1)$.

Lemma 4.1

Die amortisierte Laufzeit von INSERT beträgt in einem dynamischen Feld $O(1)$.

Beweis:

Annahme 1: Es werden keine Elemente gelöscht.

Annahme 2: size $= 1$

Lemma 4.1

Die amortisierte Laufzeit von INSERT beträgt in einem dynamischen Feld $O(1)$.

Beweis:

Annahme 1: Es werden keine Elemente gelöscht.

Annahme 2: size $= 1$

$\forall k \in \mathbb{N}_0$: beim Einfügen des $(2^k + 1)$-ten Elementes erfolgt Vergrößerung von 2^k auf 2^{k+1}.

Lemma 4.1

Die amortisierte Laufzeit von INSERT beträgt in einem dynamischen Feld $O(1)$.

Beweis:

Annahme 1: Es werden keine Elemente gelöscht.

Annahme 2: size $= 1$

$\forall k \in \mathbb{N}_0$: beim Einfügen des $(2^k + 1)$-ten Elementes erfolgt Vergrößerung von 2^k auf 2^{k+1}.

$\Rightarrow$ Die Größe nach dem Einfügen des n-ten Elementes beträgt 2^ℓ für $\ell = \lceil \log_2 n \rceil$.

Lemma 4.1

Die amortisierte Laufzeit von INSERT beträgt in einem dynamischen Feld $O(1)$.

Beweis:

Annahme 1: Es werden keine Elemente gelöscht.

Annahme 2: size $= 1$

$\forall k \in \mathbb{N}_0$: beim Einfügen des $(2^k + 1)$-ten Elementes erfolgt Vergrößerung von 2^k auf 2^{k+1}.

$\Rightarrow$ Die Größe nach dem Einfügen des n-ten Elementes beträgt 2^ℓ für $\ell = \lceil \log_2 n \rceil$.

Gesamtlaufzeit:

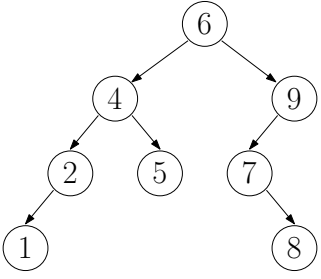
$$O\left(\sum_{i=1}^{\ell} 2^i\right) + O(n) = O(2^{\ell+1} - 2) + O(n) = O(2^{\log_2(n)}) + O(n) = O(n). \quad \square$$

	SEARCH	INSERT	DELETE
verkettete Liste	$O(n)$	$O(1)$	$O(n)$
dynamisches Feld	$O(n)$	$O(n)$, amortisiert $O(1)$	$O(n)$
sortiertes dynamisches Feld	$O(\log n)$	$O(n)$	$O(n)$

Binärer Suchbaum

Binärer Baum, in dem in jedem Knoten ein Datum gespeichert ist.

Für jeden Knoten v mit Inhalt x gilt, dass alle im **linken (rechten) Teilbaum** von v gespeicherten Daten Schlüssel besitzen, die **kleiner (größer)** als der Schlüssel von x sind.



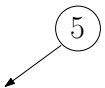
```
SEARCH(Node v, int k)
1    if ((v == null) || (v.key == k)) return v;
2    if (v.key > k) return SEARCH(v.left,k);
3    return SEARCH(v.right,k);
```

```
SEARCH(Node v, int k)
1   if ((v == null) || (v.key == k)) return v;
2   if (v.key > k) return SEARCH(v.left,k);
3   return SEARCH(v.right,k);
```

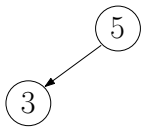
```
INSERT(Element x)
1   Führe eine (erfolglose) Suche nach x durch.
2   Füge x unter den erreichten Nullzeiger ein.
```

INSERT(5)

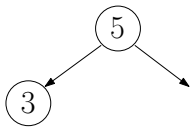
INSERT(3)



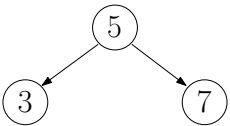
INSERT(3)



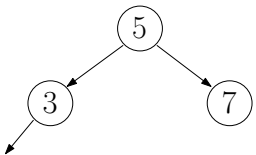
INSERT(7)



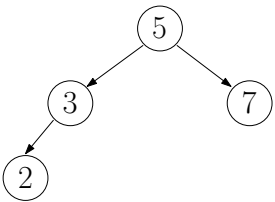
INSERT(7)



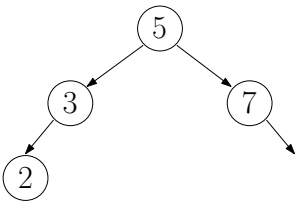
INSERT(2)



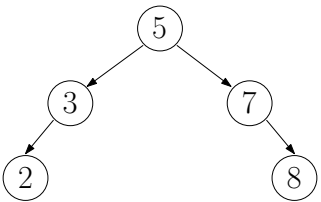
INSERT(2)



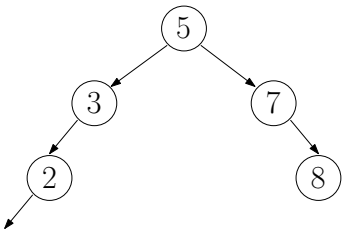
INSERT(8)



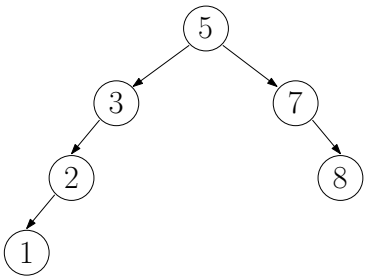
INSERT(8)



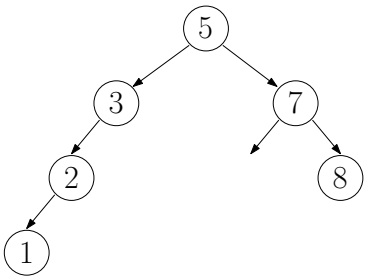
INSERT(1)



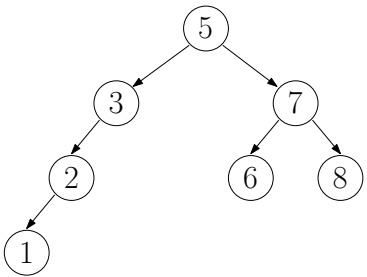
INSERT(1)



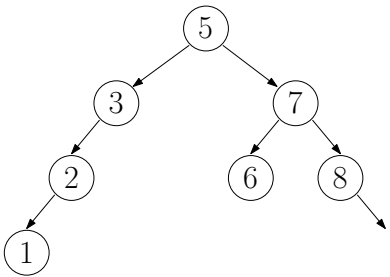
INSERT(6)



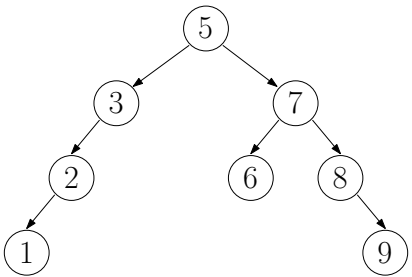
INSERT(6)



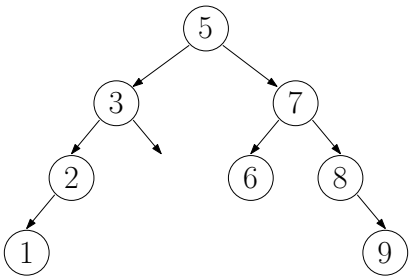
INSERT(9)



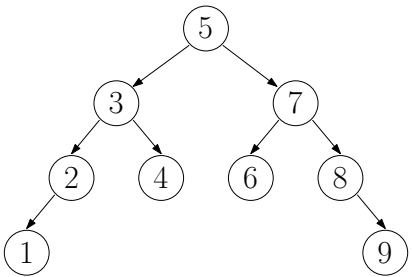
INSERT(9)



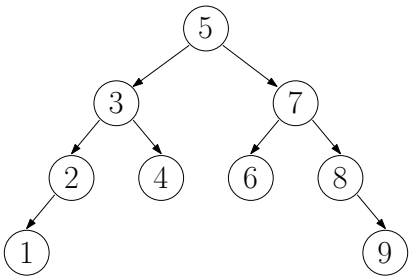
INSERT(4)



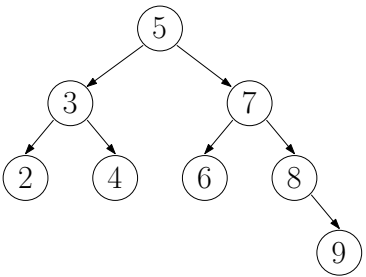
INSERT(4)



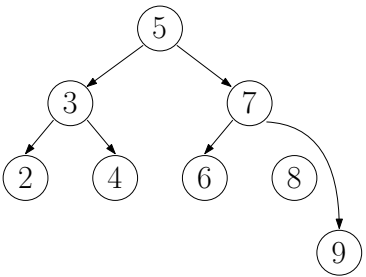
DELETE(1)



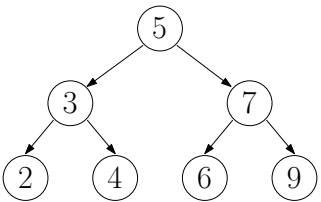
DELETE(8)



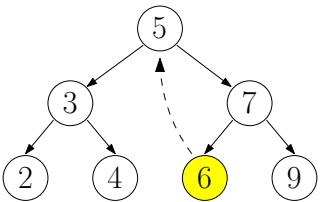
DELETE(8)



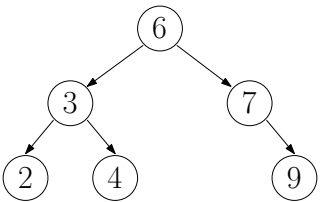
DELETE(5)



DELETE(5)



DELETE(5)



```
DELETE(k)
1    v = SEARCH(k);
2    if (v Blatt)
3       Entferne v.
4    if (v besitzt ein Kind w)
5       Lösche v durch Umsetzen des Zeigers, der auf v zeigt, auf w.
6    if (v besitzt zwei Kinder w_1 und w_2)
7       Suche im rechten Teilbaum von v den Knoten u mit dem kleinsten Schlüssel.
8       Tausche die Daten in v und u und lösche u.
9       // Da u kein linkes Kind besitzt, führt uns dies in einen der ersten beiden Fälle.
```

```
DELETE(k)
1    v = SEARCH(k);
2    if (v Blatt)
3       Entferne v.
4    if (v besitzt ein Kind w)
5       Lösche v durch Umsetzen des Zeigers, der auf v zeigt, auf w.
6    if (v besitzt zwei Kinder w_1 und w_2)
7       Suche im rechten Teilbaum von v den Knoten u mit dem kleinsten Schlüssel.
8       Tausche die Daten in v und u und lösche u.
9       // Da u kein linkes Kind besitzt, führt uns dies in einen der ersten beiden Fälle.
```

Lemma 4.2

Wird aus einem binären Suchbaum ein Element mit der obigen Methode DELETE gelöscht, so ist der resultierende Baum wieder ein binärer Suchbaum.

Theorem 4.3

Die Laufzeit der Operationen SEARCH, INSERT und DELETE in einem binären Suchbaum der Höhe h beträgt $O(h+1)$, wobei die Höhe die Anzahl der Kanten auf dem längsten Weg von der Wurzel zu einem der Blätter bezeichnet.

Theorem 4.3

Die Laufzeit der Operationen SEARCH, INSERT und DELETE in einem binären Suchbaum der Höhe h beträgt $O(h+1)$, wobei die Höhe die Anzahl der Kanten auf dem längsten Weg von der Wurzel zu einem der Blätter bezeichnet.

Theorem 4.4

Es sei T ein binärer Suchbaum mit n Knoten und Höhe h. Dann gilt $h \geq \lceil \log_2(n+1) - 1 \rceil$.

Theorem 4.3

Die Laufzeit der Operationen SEARCH, INSERT und DELETE in einem binären Suchbaum der Höhe h beträgt $O(h+1)$, wobei die Höhe die Anzahl der Kanten auf dem längsten Weg von der Wurzel zu einem der Blätter bezeichnet.

Theorem 4.4

Es sei T ein binärer Suchbaum mit n Knoten und Höhe h. Dann gilt $h \geq \lceil \log_2(n+1) - 1 \rceil$.

Beweis: Auf Ebene $i \in \{0, 1, \ldots, h\}$ gibt es höchstens 2^i Knoten.

Theorem 4.3

Die Laufzeit der Operationen SEARCH, INSERT und DELETE in einem binären Suchbaum der Höhe h beträgt $O(h+1)$, wobei die Höhe die Anzahl der Kanten auf dem längsten Weg von der Wurzel zu einem der Blätter bezeichnet.

Theorem 4.4

Es sei T ein binärer Suchbaum mit n Knoten und Höhe h. Dann gilt $h \geq \lceil \log_2(n+1) - 1 \rceil$.

Beweis: Auf Ebene $i \in \{0, 1, \ldots, h\}$ gibt es höchstens 2^i Knoten.
Knotenzahl eines Suchbaumes der Höhe h beträgt höchstens

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

Theorem 4.3

Die Laufzeit der Operationen SEARCH, INSERT und DELETE in einem binären Suchbaum der Höhe h beträgt $O(h+1)$, wobei die Höhe die Anzahl der Kanten auf dem längsten Weg von der Wurzel zu einem der Blätter bezeichnet.

Theorem 4.4

Es sei T ein binärer Suchbaum mit n Knoten und Höhe h. Dann gilt $h \geq \lceil \log_2(n+1) - 1 \rceil$.

Beweis: Auf Ebene $i \in \{0, 1, \ldots, h\}$ gibt es höchstens 2^i Knoten.
Knotenzahl eines Suchbaumes der Höhe h beträgt höchstens

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

$\Rightarrow n \leq 2^{h+1} - 1$

Theorem 4.3

Die Laufzeit der Operationen SEARCH, INSERT und DELETE in einem binären Suchbaum der Höhe h beträgt $O(h+1)$, wobei die Höhe die Anzahl der Kanten auf dem längsten Weg von der Wurzel zu einem der Blätter bezeichnet.

Theorem 4.4

Es sei T ein binärer Suchbaum mit n Knoten und Höhe h. Dann gilt $h \geq \lceil \log_2(n+1) - 1 \rceil$.

Beweis: Auf Ebene $i \in \{0, 1, \ldots, h\}$ gibt es höchstens 2^i Knoten.
Knotenzahl eines Suchbaumes der Höhe h beträgt höchstens

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

$\Rightarrow n \leq 2^{h+1} - 1 \qquad \Rightarrow h \geq \log_2(n+1) - 1.$ □

4.2.1 AVL-Bäume

Definition 4.5

Ein binärer Suchbaum heißt **AVL-Baum**, wenn für jeden Knoten v des Baumes gilt, dass sich die Höhen des linken und des rechten Teilbaumes um maximal 1 unterscheiden.

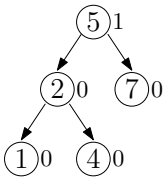
Seien T_L und T_R die Teilbäume von v und seien $h(T_L)$ und $h(T_R)$ ihre Höhen. Dann bezeichne $\mathrm{bal}(v) = h(T_L) - h(T_R)$ die **Balance** von Knoten v. In AVL-Bäumen gilt $\mathrm{bal}(v) \in \{-1, 0, 1\}$ für jeden Knoten v.

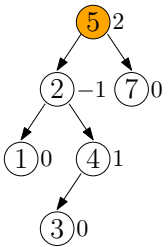
4.2.1 AVL-Bäume

Definition 4.5

Ein binärer Suchbaum heißt **AVL-Baum**, wenn für jeden Knoten v des Baumes gilt, dass sich die Höhen des linken und des rechten Teilbaumes um maximal 1 unterscheiden.

Seien T_L und T_R die Teilbäume von v und seien $h(T_L)$ und $h(T_R)$ ihre Höhen. Dann bezeichne $\mathrm{bal}(v) = h(T_L) - h(T_R)$ die **Balance** von Knoten v. In AVL-Bäumen gilt $\mathrm{bal}(v) \in \{-1, 0, 1\}$ für jeden Knoten v.





4.2.1 AVL-Bäume

Theorem 4.6

Ein AVL-Baum mit $n \in \mathbb{N}$ Knoten besitzt eine Höhe von mindestens $\lceil \log_2(n+1) - 1 \rceil$ und höchstens $1{,}4405 \log_2(n+1)$.

Theorem 4.6

Ein AVL-Baum mit $n \in \mathbb{N}$ Knoten besitzt eine Höhe von mindestens $\lceil \log_2(n+1) - 1 \rceil$ und höchstens $1{,}4405 \log_2(n+1)$.

Beweis:

$A(h) =$ geringste Anzahl an Knoten, die ein AVL-Baum der Höhe h haben kann

Theorem 4.6

Ein AVL-Baum mit $n \in \mathbb{N}$ Knoten besitzt eine Höhe von mindestens $\lceil \log_2(n+1) - 1 \rceil$ und höchstens $1{,}4405 \log_2(n+1)$.

Beweis:

$A(h) =$ geringste Anzahl an Knoten, die ein AVL-Baum der Höhe h haben kann
Es gilt $A(0) = 1$ und $A(1) = 2$.

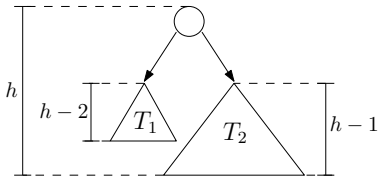
Theorem 4.6

Ein AVL-Baum mit $n \in \mathbb{N}$ Knoten besitzt eine Höhe von mindestens $\lceil \log_2(n+1) - 1 \rceil$ und höchstens $1{,}4405 \log_2(n+1)$.

Beweis:

$A(h) =$ geringste Anzahl an Knoten, die ein AVL-Baum der Höhe h haben kann
Es gilt $A(0) = 1$ und $A(1) = 2$.

Für $h \geq 2$ gilt $A(h) = 1 + A(h-1) + A(h-2)$.



Es gilt $A(0) = 1$ und $A(1) = 2$.

Für $h \geq 2$ gilt $A(h) = 1 + A(h-1) + A(h-2)$.

Es ergibt sich die Folge **1, 2, 4, 7, 12, 20, 33, ...**.

Es gilt $A(0) = 1$ und $A(1) = 2$.

Für $h \geq 2$ gilt $A(h) = 1 + A(h-1) + A(h-2)$.

Es ergibt sich die Folge **1, 2, 4, 7, 12, 20, 33, ...**.

Zum Vergleich die Fibonacci-Folge: 0, 1, 1, **2, 3, 5, 8, 13, 21, 34, ...**.

Vermutung: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$

Es gilt $A(0) = 1$ und $A(1) = 2$.

Für $h \geq 2$ gilt $A(h) = 1 + A(h-1) + A(h-2)$.

Es ergibt sich die Folge **1, 2, 4, 7, 12, 20, 33, ...**.

Zum Vergleich die Fibonacci-Folge: 0, 1, 1, **2, 3, 5, 8, 13, 21, 34, ...**.

Vermutung: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$

Beweis per Induktion:

Induktionsanfang: $A(0) = 1 = 2 - 1 = f_3 - 1 \quad A(1) = 2 = 3 - 1 = f_4 - 1$

Es gilt $A(0) = 1$ und $A(1) = 2$.

Für $h \geq 2$ gilt $A(h) = 1 + A(h-1) + A(h-2)$.

Es ergibt sich die Folge **1, 2, 4, 7, 12, 20, 33, ...**.

Zum Vergleich die Fibonacci-Folge: 0, 1, 1, **2, 3, 5, 8, 13, 21, 34, ...**.

Vermutung: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$

Beweis per Induktion:

Induktionsanfang: $A(0) = 1 = 2 - 1 = f_3 - 1 \quad A(1) = 2 = 3 - 1 = f_4 - 1$

Induktionsschritt: Für $h \geq 2$ gilt

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

Es gilt $A(0) = 1$ und $A(1) = 2$.

Für $h \geq 2$ gilt $A(h) = 1 + A(h-1) + A(h-2)$.

Es ergibt sich die Folge **1, 2, 4, 7, 12, 20, 33, ...**.

Zum Vergleich die Fibonacci-Folge: 0, 1, 1, **2, 3, 5, 8, 13, 21, 34, ...**.

Vermutung: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$

Beweis per Induktion:

Induktionsanfang: $A(0) = 1 = 2 - 1 = f_3 - 1 \quad A(1) = 2 = 3 - 1 = f_4 - 1$

Induktionsschritt: Für $h \geq 2$ gilt

$$A(h) = 1 + A(h-1) + A(h-2) = 1 + (f_{h+2} - 1) + (f_{h+1} - 1)$$

Es gilt $A(0) = 1$ und $A(1) = 2$.
Für $h \geq 2$ gilt $A(h) = 1 + A(h-1) + A(h-2)$.
Es ergibt sich die Folge **1, 2, 4, 7, 12, 20, 33, ...**.

Zum Vergleich die Fibonacci-Folge: 0, 1, 1, **2, 3, 5, 8, 13, 21, 34, ...**.
Vermutung: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$

Beweis per Induktion:
Induktionsanfang: $A(0) = 1 = 2 - 1 = f_3 - 1 \quad A(1) = 2 = 3 - 1 = f_4 - 1$

Induktionsschritt: Für $h \geq 2$ gilt

$$\begin{aligned} A(h) &= 1 + A(h-1) + A(h-2) = 1 + (f_{h+2} - 1) + (f_{h+1} - 1) \\ &= f_{h+2} + f_{h+1} - 1 \end{aligned}$$

Es gilt $A(0) = 1$ und $A(1) = 2$.
Für $h \geq 2$ gilt $A(h) = 1 + A(h-1) + A(h-2)$.
Es ergibt sich die Folge **1, 2, 4, 7, 12, 20, 33, ...**.

Zum Vergleich die Fibonacci-Folge: 0, 1, 1, **2, 3, 5, 8, 13, 21, 34, ...**.
Vermutung: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$

Beweis per Induktion:
Induktionsanfang: $A(0) = 1 = 2 - 1 = f_3 - 1 \quad A(1) = 2 = 3 - 1 = f_4 - 1$

Induktionsschritt: Für $h \geq 2$ gilt

$$\begin{aligned} A(h) &= 1 + A(h-1) + A(h-2) = 1 + (f_{h+2} - 1) + (f_{h+1} - 1) \\ &= f_{h+2} + f_{h+1} - 1 = f_{h+3} - 1. \end{aligned}$$

Also: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$.

Also: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$.

Für die Fibonacci-Zahlen gilt (Übung):

$$f_h = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right].$$

Also: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$.

Für die Fibonacci-Zahlen gilt (Übung):

$$f_h = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right].$$

Wegen $n \geq A(h)$ bedeutet das

$$n + 1 \geq A(h) + 1 = f_{h+3} = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right].$$

Also: $A(h) = f_{h+3} - 1$ für jedes $h \in \mathbb{N}_0$.

Für die Fibonacci-Zahlen gilt (Übung):

$$f_h = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right].$$

Wegen $n \geq A(h)$ bedeutet das

$$n+1 \geq A(h)+1 = f_{h+3} = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - \left(\frac{1-\sqrt{5}}{2} \right)^{h+3} \right].$$

Aus $\left| \frac{1-\sqrt{5}}{2} \right| < 1$ folgt

$$n+1 \geq \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^{h+3} - 1 \right].$$

Durch Umformen ergibt sich hieraus

$$
\begin{aligned}
h + 3 &\leq \frac{\log_2(\sqrt{5}(n+1)+1)}{\log_2\left(\frac{1+\sqrt{5}}{2}\right)} \\
&< 1{,}4405 \cdot \log_2(\sqrt{5}(n+1)+1) \\
&\leq 1{,}4405 \cdot \log_2((\sqrt{5}+1)(n+1)) \\
&= 1{,}4405 \cdot \log_2(n+1) + 1{,}4405 \cdot \log_2(\sqrt{5}+1) \\
&< 1{,}4405 \cdot \log_2(n+1) + 2{,}5.
\end{aligned}
$$

Durch Umformen ergibt sich hieraus

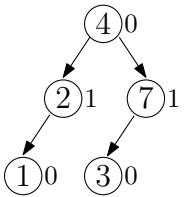
$$
\begin{aligned}
h + 3 &\leq \frac{\log_2(\sqrt{5}(n+1)+1)}{\log_2\left(\frac{1+\sqrt{5}}{2}\right)} \\
&< 1{,}4405 \cdot \log_2(\sqrt{5}(n+1)+1) \\
&\leq 1{,}4405 \cdot \log_2((\sqrt{5}+1)(n+1)) \\
&= 1{,}4405 \cdot \log_2(n+1) + 1{,}4405 \cdot \log_2(\sqrt{5}+1) \\
&< 1{,}4405 \cdot \log_2(n+1) + 2{,}5.
\end{aligned}
$$

Daraus folgt

$$h < 1{,}4405 \cdot \log_2(n+1). \quad \square$$

INSERT(Element x)

Füge x zunächst wie in einen normalen binären Suchbaum ein.

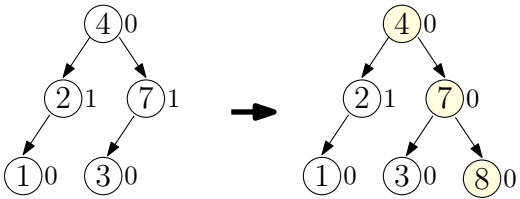


INSERT(Element x)

Füge x zunächst wie in einen normalen binären Suchbaum ein.

Betrachte **Pfad P von neuem Blatt zur Wurzel**.

Nur auf diesem Pfad ändern sich die Balancen.

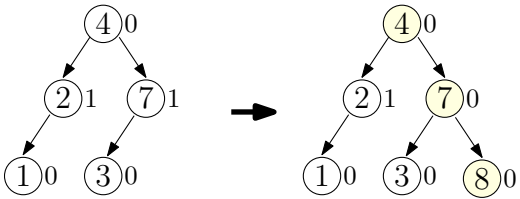


INSERT(Element x)

Füge x zunächst wie in einen normalen binären Suchbaum ein.

Betrachte **Pfad P von neuem Blatt zur Wurzel**.

Nur auf diesem Pfad ändern sich die Balancen.



Folge P vom neuen Blatt zur Wurzel zurück und **korrigiere die Balancen**.

INSERT(Element x)

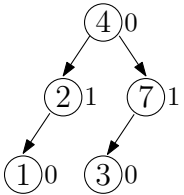
Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem rechten Kind.

INSERT(Element x)

Sei v auf dem Pfad P.

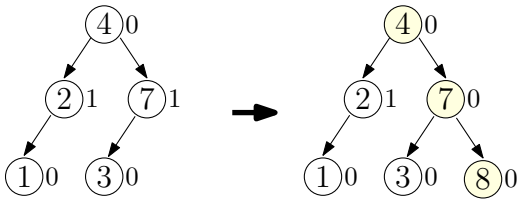
Annahme: Wir erreichen v von seinem rechten Kind.



INSERT(Element x)

Sei v auf dem Pfad P.

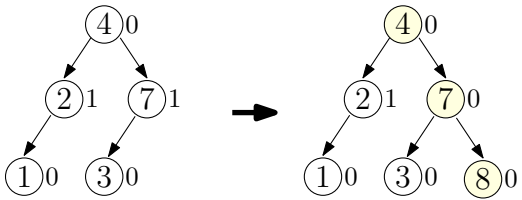
Annahme: Wir erreichen v von seinem rechten Kind.



INSERT(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem rechten Kind.



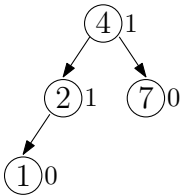
1. Fall: $\mathrm{bal}(v) = 1$: Setze $\mathrm{bal}(v) = 0$.

Die Rebalancierung ist abgeschlossen, da sich die Höhe des Teilbaumes mit Wurzel v durch das Einfügen nicht geändert hat.

2. Fall: $\mathrm{bal}(v) = 0$: Setze $\mathrm{bal}(v) = -1$.
Ist v die Wurzel des Baumes, dann ist die Rebalancierung abgeschlossen. Ansonsten gehen wir zum Elternknoten von v und setzen die Rebalancierung fort.

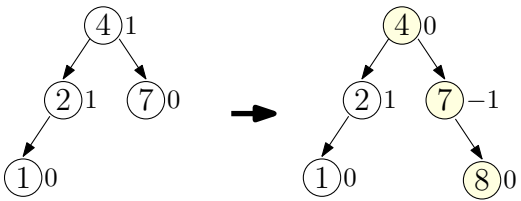
2. Fall: $\mathrm{bal}(v) = 0$: Setze $\mathrm{bal}(v) = -1$.

Ist v die Wurzel des Baumes, dann ist die Rebalancierung abgeschlossen. Ansonsten gehen wir zum Elternknoten von v und setzen die Rebalancierung fort.



2. Fall: $\mathrm{bal}(v) = 0$: Setze $\mathrm{bal}(v) = -1$.

Ist v die Wurzel des Baumes, dann ist die Rebalancierung abgeschlossen. Ansonsten gehen wir zum Elternknoten von v und setzen die Rebalancierung fort.

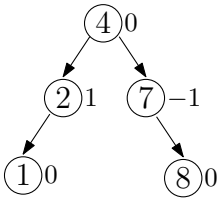


3. Fall: $\mathrm{bal}(v) = -1$: Setzen von $\mathrm{bal}(v) = -2$ nicht erlaubt.

AVL-Eigenschaft ist nun verletzt. Der Baum muss *rebalanciert* werden.

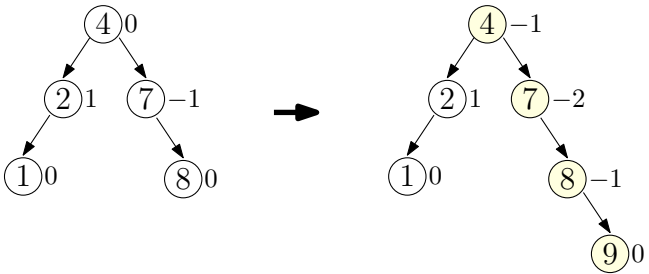
3. Fall: $\mathrm{bal}(v) = -1$: Setzen von $\mathrm{bal}(v) = -2$ nicht erlaubt.

AVL-Eigenschaft ist nun verletzt. Der Baum muss *rebalanciert* werden.



3. Fall: $\mathrm{bal}(v) = -1$: Setzen von $\mathrm{bal}(v) = -2$ nicht erlaubt.

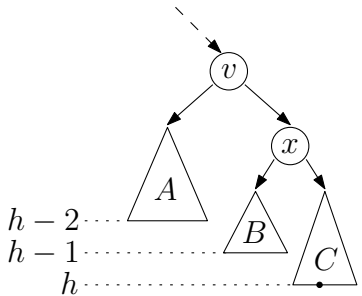
AVL-Eigenschaft ist nun verletzt. Der Baum muss *rebalanciert* werden.



3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v.

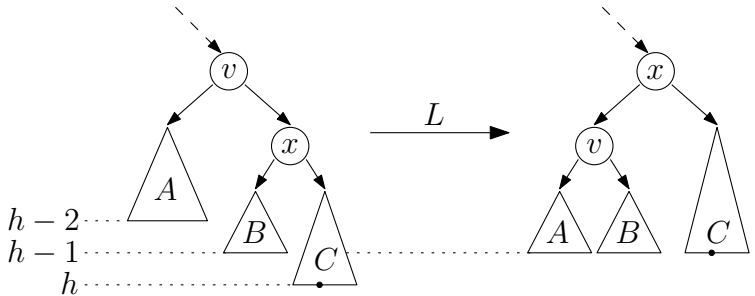
- 1. Unterfall: Die Höhe des **rechten** Teilbaumes von x ist um eins gewachsen.



3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v.

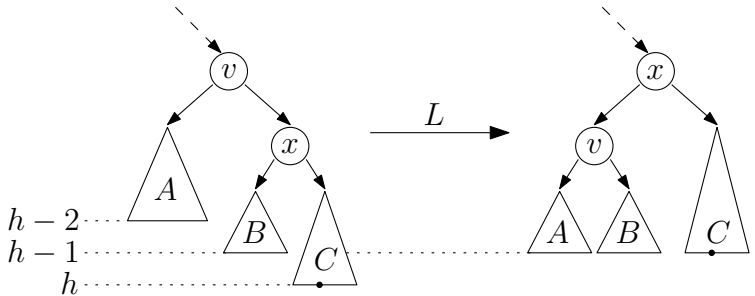
- 1. Unterfall: Die Höhe des **rechten** Teilbaumes von x ist um eins gewachsen.
  Es genügt eine **Linksrotation**:



3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v.

- 1. Unterfall: Die Höhe des **rechten** Teilbaumes von x ist um eins gewachsen.
  Es genügt eine **Linksrotation**:

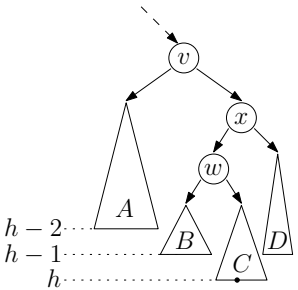


Die Rebalancierung ist damit abgeschlossen

3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v und w das linke Kind von x.

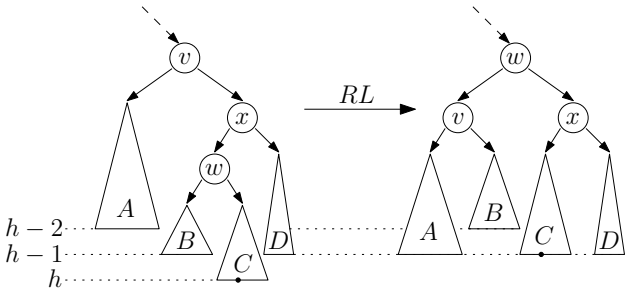
- 2. Unterfall: Die Höhen des **linken** Teilbaumes von x sowie des **rechten** Teilbaumes von w sind um eins gewachsen.



3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v und w das linke Kind von x.

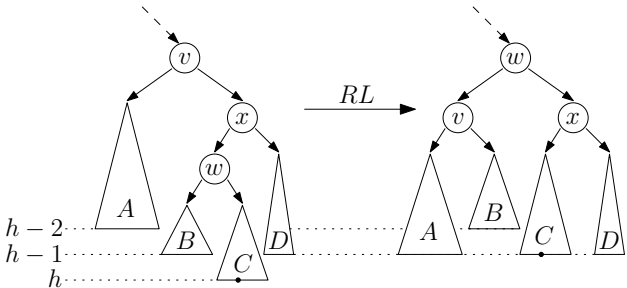
- 2. Unterfall: Die Höhen des **linken** Teilbaumes von x sowie des **rechten** Teilbaumes von w sind um eins gewachsen.
  Es genügt eine **RL-Rotation**:



3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v und w das linke Kind von x.

- 2. Unterfall: Die Höhen des **linken** Teilbaumes von x sowie des **rechten** Teilbaumes von w sind um eins gewachsen.
  Es genügt eine **RL-Rotation**:

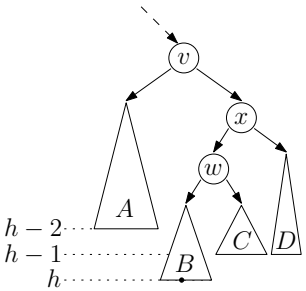


Die Rebalancierung ist damit abgeschlossen

3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v und w das linke Kind von x.

- 3. Unterfall: Die Höhen des **linken** Teilbaumes von x sowie des **linken** Teilbaumes von w sind um eins gewachsen.

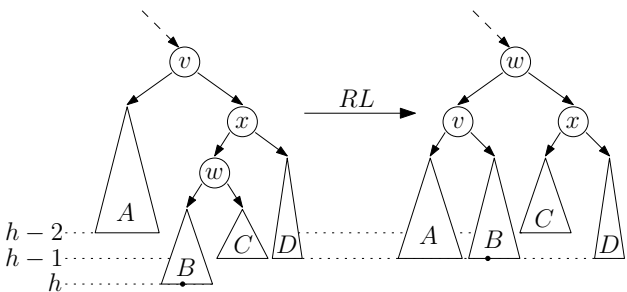


3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v und w das linke Kind von x.

- 3. Unterfall: Die Höhen des **linken** Teilbaumes von x sowie des **linken** Teilbaumes von w sind um eins gewachsen.

  Es genügt eine **RL-Rotation**:

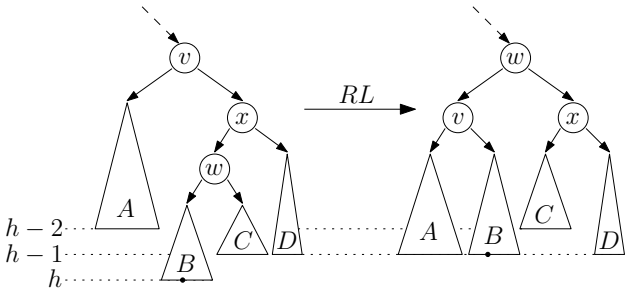


3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechtes Kind von v und w das linke Kind von x.

- 3. Unterfall: Die Höhen des **linken** Teilbaumes von x sowie des **linken** Teilbaumes von w sind um eins gewachsen.

  Es genügt eine **RL-Rotation**:

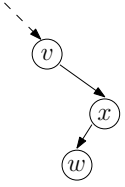


Die Rebalancierung ist damit abgeschlossen

3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v und w das linke Kind von x.

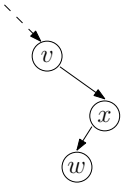
- 4. Unterfall: Die Höhen des **linken** Teilbaumes von x um eins gewachsen und w besitzt **kein Kind**.



3. Fall: Nach Einfügen gilt bal(v) $= -2$.

Sei x das rechte Kind von v und w das linke Kind von x.

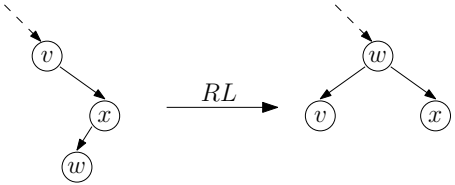
- 4. Unterfall: Die Höhen des **linken** Teilbaumes von x um eins gewachsen und w besitzt **kein Kind**.
  $\Rightarrow$ w ist neu eingefügter Knoten und Teilbäume A, B, C, D leer.



3. Fall: Nach Einfügen gilt bal(v) $= -2$.

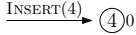
Sei x das rechte Kind von v und w das linke Kind von x.

- 4. Unterfall: Die Höhen des **linken** Teilbaumes von x um eins gewachsen und w besitzt **kein Kind**.
  $\Rightarrow$ w ist neu eingefügter Knoten und Teilbäume A, B, C, D leer.

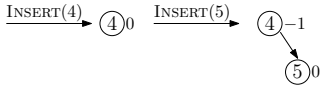


Die Rebalancierung ist damit abgeschlossen

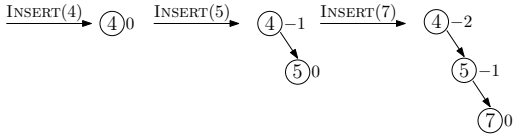
4.2.1 AVL-Bäume



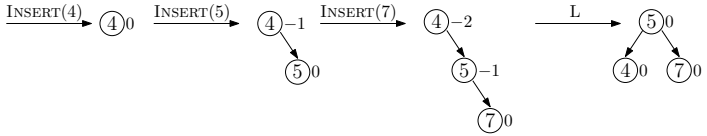
4.2.1 AVL-Bäume



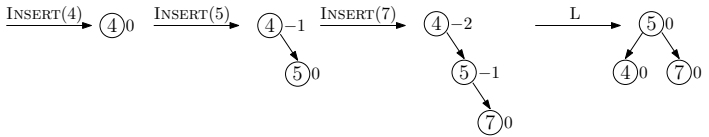
4.2.1 AVL-Bäume

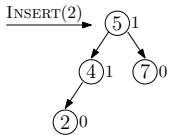


4.2.1 AVL-Bäume

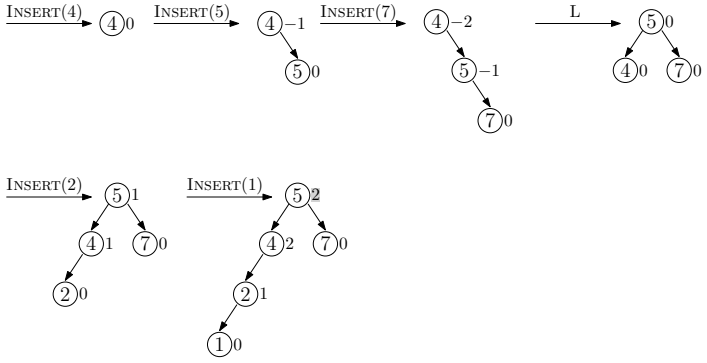


4.2.1 AVL-Bäume

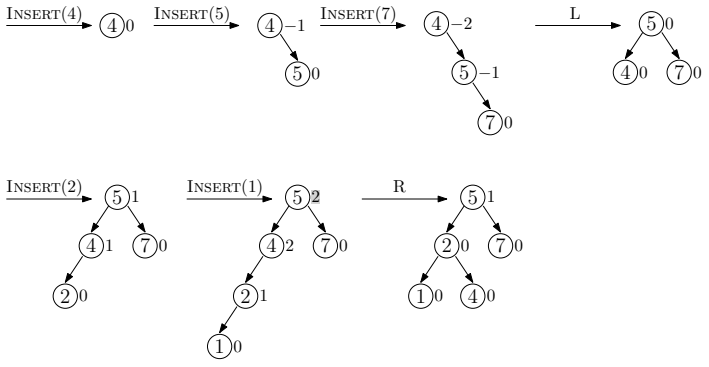




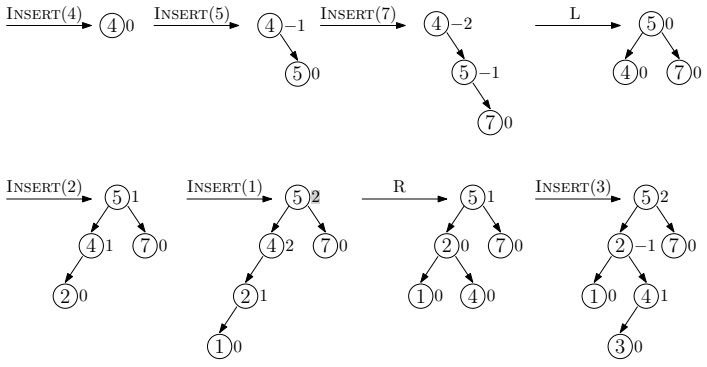
4.2.1 AVL-Bäume



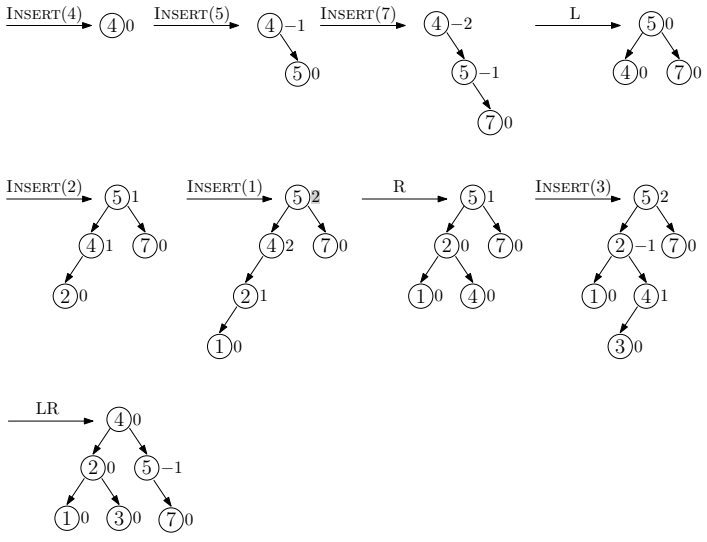
4.2.1 AVL-Bäume



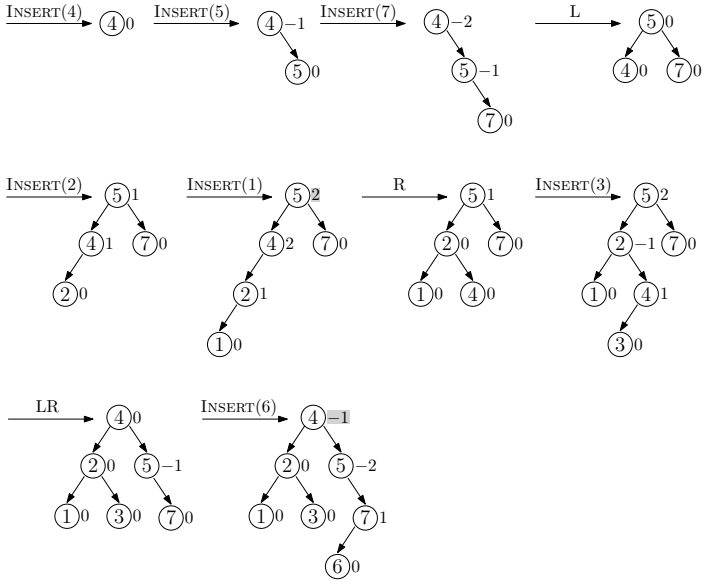
4.2.1 AVL-Bäume



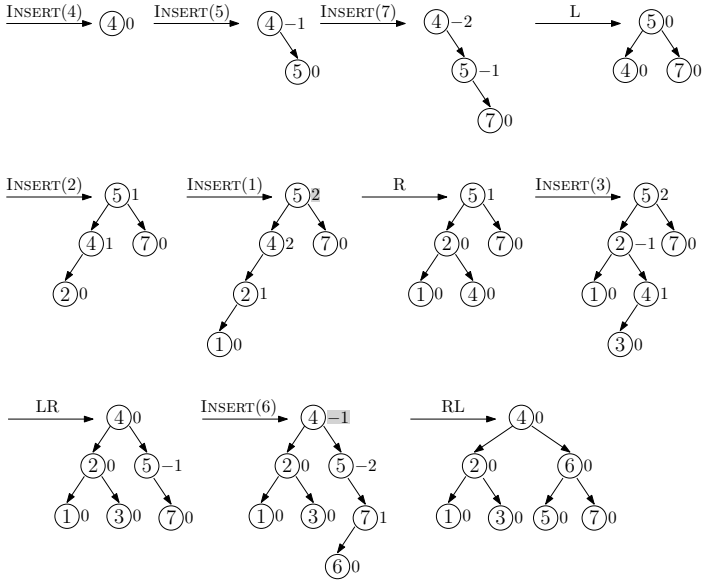
4.2.1 AVL-Bäume



4.2.1 AVL-Bäume

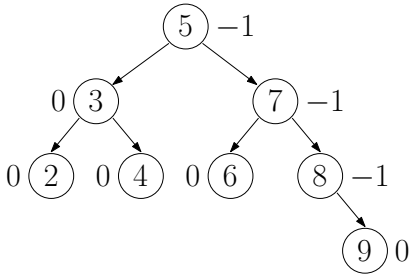


4.2.1 AVL-Bäume



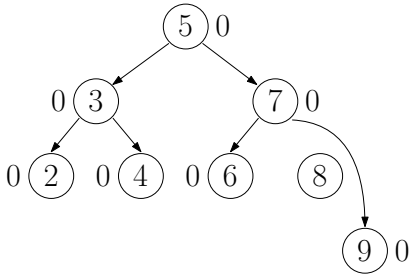
Delete(Element x)

Lösche x zunächst wie in einen normalen binären Suchbaum.



Delete(Element x)

Lösche *x* zunächst wie in einen normalen binären Suchbaum.

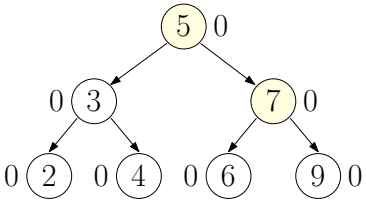


Delete(Element x)

Lösche *x* zunächst wie in einen normalen binären Suchbaum.

Betrachte **Pfad P von gelöschtem Knoten zur Wurzel**.

Nur auf diesem Pfad ändern sich die Balancen.

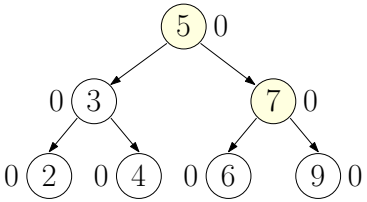


Delete(Element x)

Lösche x zunächst wie in einen normalen binären Suchbaum.

Betrachte **Pfad P von gelöschtem Knoten zur Wurzel**.

Nur auf diesem Pfad ändern sich die Balancen.



Folge P vom gelöschten Knoten zur Wurzel zurück und **korrigiere die Balancen**.

Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

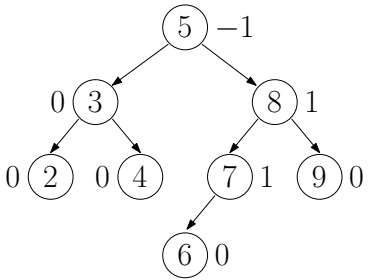
1. Fall: bal$(v) = 1$:

Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

1. Fall: bal(v) = 1:

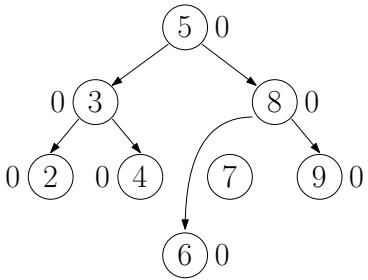


Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

1. Fall: bal(v) = 1:

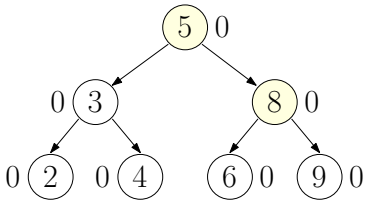


Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

1. Fall: bal(v) = 1: Setze $\mathrm{bal}(v) = 0$.

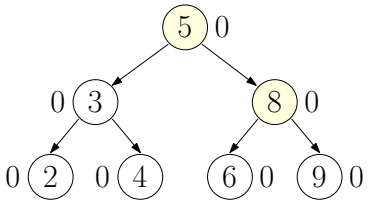


Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

1. Fall: bal(v) = 1: Setze $\mathrm{bal}(v) = 0$.



Die Rebalancierung wird am Elternknoten von v fortgesetzt.

Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

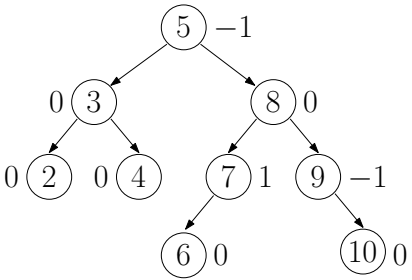
2. Fall: bal$(v) = 0$:

Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

2. Fall: bal$(v) = 0$:

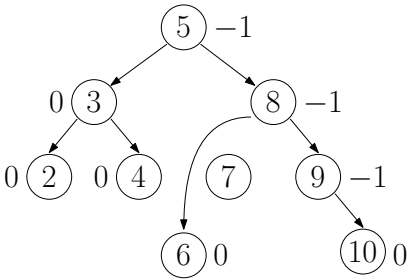


Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

2. Fall: bal$(v) = 0$:

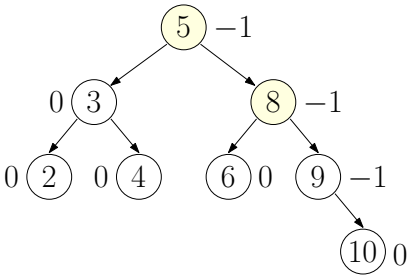


Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

2. Fall: bal(v) = 0: Setze $\mathrm{bal}(v) = -1$.

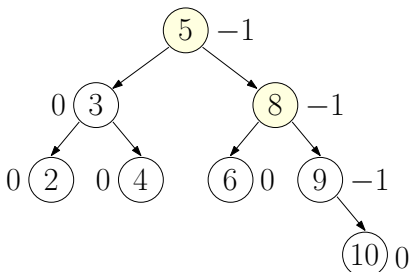


Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

2. Fall: bal(v) = 0: Setze $\mathrm{bal}(v) = -1$.



Die Rebalancierung ist abgeschlossen, da sich die Höhe des Teilbaumes mit Wurzel v durch das Einfügen nicht geändert hat.

Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

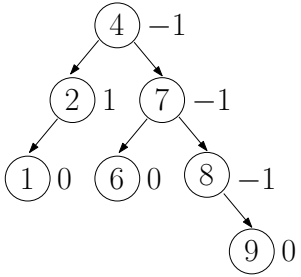
3. Fall: bal$(v) = -1$:

Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

3. Fall: bal(v) $= -1$:

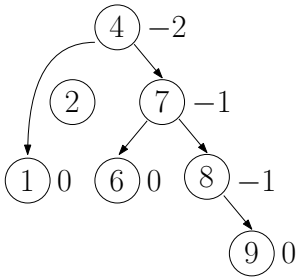


Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

3. Fall: bal(v) $= -1$:

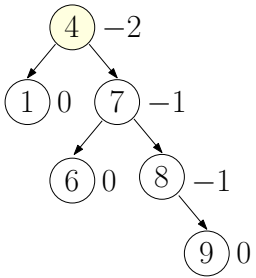


Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

3. Fall: bal(v) $= -1$: Setzen von $\mathrm{bal}(v) = -2$ nicht erlaubt.

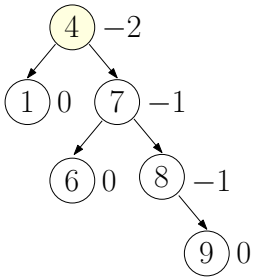


Delete(Element x)

Sei v auf dem Pfad P.

Annahme: Wir erreichen v von seinem linken Kind.

3. Fall: bal(v) $= -1$: Setzen von $\mathrm{bal}(v) = -2$ nicht erlaubt.

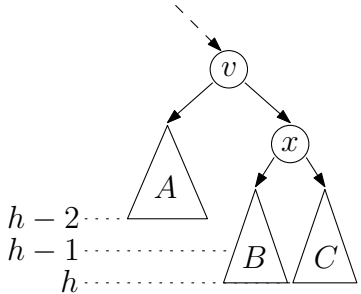


AVL-Eigenschaft ist nun verletzt. Der Baum muss *rebalanciert* werden.

3. Fall: Nach Löschen gilt $\mathbf{bal(v) = -2}$.

Sei x das rechte Kind von v.

- 1. Unterfall: $\mathbf{bal(x) = 0}$

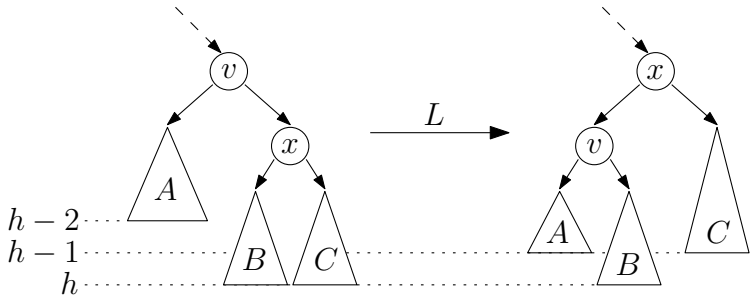


3. Fall: Nach Löschen gilt $\mathbf{bal(v) = -2}$.

Sei x das rechte Kind von v.

- 1. Unterfall: $\mathbf{bal(x) = 0}$

  Es genügt eine **Linksrotation**:

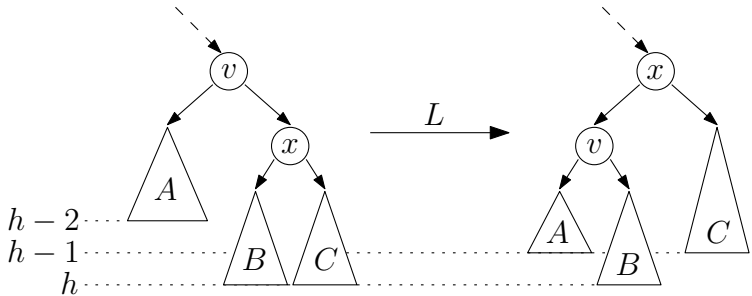


3. Fall: Nach Löschen gilt $\mathbf{bal(v) = -2}$.

Sei x das rechte Kind von v.

- 1. Unterfall: $\mathbf{bal(x) = 0}$

  Es genügt eine **Linksrotation**:

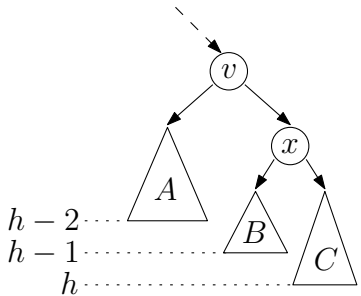


Die Rebalancierung ist damit abgeschlossen.

3. Fall: Nach Löschen gilt $\mathbf{bal(v) = -2}$.

Sei x das rechte Kind von v.

- 2. Unterfall: $\mathbf{bal(x) = -1}$

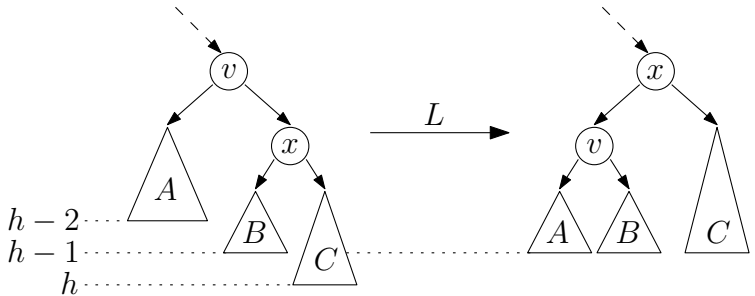


3. Fall: Nach Löschen gilt $\mathbf{bal(v) = -2}$.

Sei x das rechte Kind von v.

- 2. Unterfall: $\mathbf{bal(x) = -1}$

  Es genügt eine **Linksrotation**:

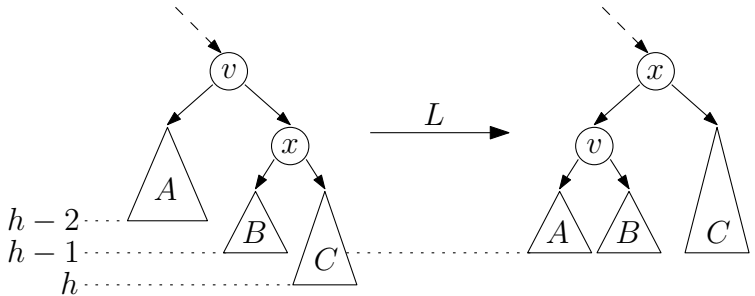


3. Fall: Nach Löschen gilt $\mathbf{bal(v) = -2}$.

Sei x das rechte Kind von v.

- 2. Unterfall: $\mathbf{bal(x) = -1}$

  Es genügt eine **Linksrotation**:

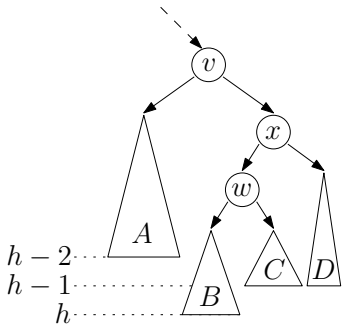


Die Rebalancierung muss am Elternknoten von v fortgesetzt werden.

3. Fall: Nach Löschen gilt $\mathbf{bal(v) = -2}$.

Sei x das rechte Kind von v.

- 3. Unterfall: $\mathbf{bal(x) = 1}$

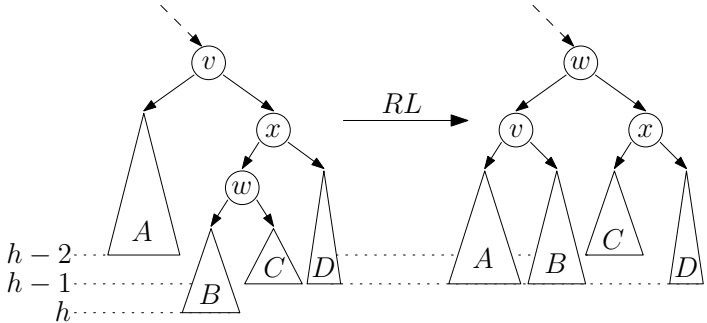


3. Fall: Nach Löschen gilt bal(v) = −2.

Sei x das rechte Kind von v.

- 3. Unterfall: **bal(x) = 1**

  Es genügt eine **RL-Rotation**:

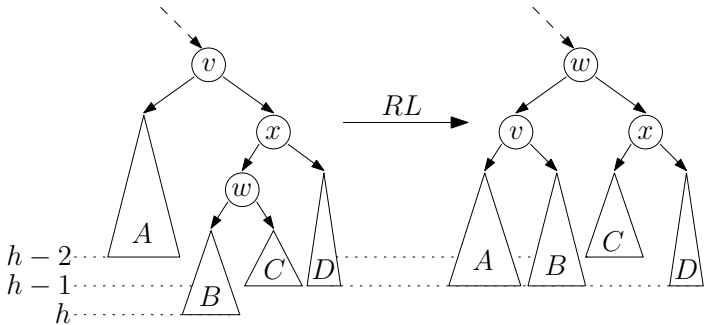


3. Fall: Nach Löschen gilt $\mathbf{bal(v) = -2}$.

Sei x das rechte Kind von v.

- 3. Unterfall: $\mathbf{bal(x) = 1}$

  Es genügt eine **RL-Rotation**:



Die Rebalancierung muss am Elternknoten von v (bzw. w) fortgesetzt werden.

Theorem 4.7

Die Operationen SEARCH, INSERT und DELETE benötigen in AVL-Bäumen mit n Daten eine Laufzeit von $O(\log(n))$.

4.2.2 B-Bäume

B-Bäume: Weitere Datenstruktur für dynamische Mengen

- Dieselben **asymptotischen Laufzeiten wie AVL-Bäume**.

4.2.2 B-Bäume

B-Bäume: Weitere Datenstruktur für dynamische Mengen

- Dieselben **asymptotischen Laufzeiten wie AVL-Bäume**.
- Berücksichtigt **unterschiedliche Zugriffsgeschwindigkeiten** auf RAM und HD/SSD sowie die Organisation des Speichers in Seiten.

4.2.2 B-Bäume

B-Bäume: Weitere Datenstruktur für dynamische Mengen

- Dieselben **asymptotischen Laufzeiten wie AVL-Bäume**.
- Berücksichtigt **unterschiedliche Zugriffsgeschwindigkeiten** auf RAM und HD/SSD sowie die Organisation des Speichers in Seiten.
- Wird häufig in **Datenbanken** oder **Dateisystemen** eingesetzt.

4.2.2 B-Bäume

Verallgemeinerter Suchbaum

Jeder Knoten enthält **einen oder mehrere Schlüssel**.

4.2.2 B-Bäume

Verallgemeinerter Suchbaum

Jeder Knoten enthält **einen oder mehrere Schlüssel**.

Enthält ein innerer Knoten ℓ **Schlüssel**, so besitzt er $\ell + 1$ **Kinder**.

4.2.2 B-Bäume

Verallgemeinerter Suchbaum

Jeder Knoten enthält **einen oder mehrere Schlüssel**.

Enthält ein innerer Knoten ℓ **Schlüssel**, so besitzt er $\ell + 1$ **Kinder**.

Seien $v.\text{key}(1) \leq \ldots \leq v.\text{key}(\ell)$ die Schlüssel von v und seien $u_1, \ldots, u_{\ell+1}$ die Kinder.

Sei k_i ein beliebiger Schlüssel in dem Unterbaum mit Wurzel u_i.

4.2.2 B-Bäume

Verallgemeinerter Suchbaum

Jeder Knoten enthält **einen oder mehrere Schlüssel**.

Enthält ein innerer Knoten ℓ **Schlüssel**, so besitzt er $\ell + 1$ **Kinder**.

Seien $v.\text{key}(1) \leq \ldots \leq v.\text{key}(\ell)$ die Schlüssel von v und seien $u_1, \ldots, u_{\ell+1}$ die Kinder.

Sei k_i ein beliebiger Schlüssel in dem Unterbaum mit Wurzel u_i.

Dann gilt

$$k_1 \leq v.\text{key}(1) \leq k_2 \leq v.\text{key}(2) \leq \ldots \leq v.\text{key}(\ell) \leq k_{\ell+1}.$$

4.2.2 B-Bäume

Verallgemeinerter Suchbaum

Jeder Knoten enthält **einen oder mehrere Schlüssel**.

Enthält ein innerer Knoten ℓ **Schlüssel**, so besitzt er $\ell + 1$ **Kinder**.

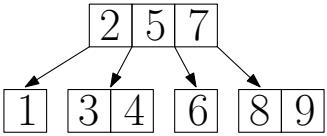
Seien $v.\text{key}(1) \leq \ldots \leq v.\text{key}(\ell)$ die Schlüssel von v und seien $u_1, \ldots, u_{\ell+1}$ die Kinder.

Sei k_i ein beliebiger Schlüssel in dem Unterbaum mit Wurzel u_i.

Dann gilt

$$k_1 \leq v.\text{key}(1) \leq k_2 \leq v.\text{key}(2) \leq \ldots \leq v.\text{key}(\ell) \leq k_{\ell+1}.$$

Beispiel:



4.2.2 B-Bäume

Definition 4.8

Ein verallgemeinerter Suchbaum T heißt **B-Baum** der Ordnung $t \in \mathbb{N}$ mit $t \geq 2$, falls:

1. Alle **Blätter** haben **dieselbe Tiefe**.
2. Jeder Knoten mit Ausnahme der Wurzel enthält **mindestens** $t - 1$ **Daten** und jeder Knoten einschließlich der Wurzel enthält **höchstens** $2t - 1$ **Daten**.
3. Ist der Baum nicht leer, so enthält die Wurzel mindestens ein Datum.

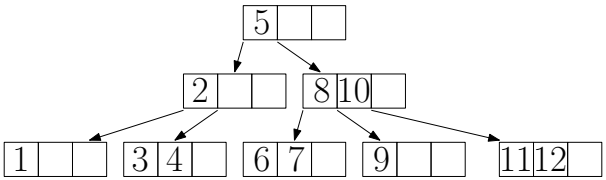
4.2.2 B-Bäume

Definition 4.8

Ein verallgemeinerter Suchbaum T heißt **B-Baum** der Ordnung $t \in \mathbb{N}$ mit $t \geq 2$, falls:

1. Alle **Blätter** haben **dieselbe Tiefe**.
2. Jeder Knoten mit Ausnahme der Wurzel enthält **mindestens** $t - 1$ **Daten** und jeder Knoten einschließlich der Wurzel enthält **höchstens** $2t - 1$ **Daten**.
3. Ist der Baum nicht leer, so enthält die Wurzel mindestens ein Datum.

Beispiel: $t = 2$



Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

4.2.2 B-Bäume

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**

4.2.2 B-Bäume

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten Tiefe 1: ≥ 2 Knoten

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten Tiefe 1: ≥ 2 Knoten Tiefe 2: $\geq 2t$ Knoten

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten Tiefe 1: ≥ 2 Knoten Tiefe 2: $\geq 2t$ Knoten
  Tiefe 3: $\geq 2t^2$ Knoten

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten Tiefe 1: ≥ 2 Knoten Tiefe 2: $\geq 2t$ Knoten
  Tiefe 3: $\geq 2t^2$ Knoten Tiefe $i \leq h$: $\geq 2t^{i-1}$ Knoten

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten Tiefe 1: ≥ 2 Knoten Tiefe 2: $\geq 2t$ Knoten
  Tiefe 3: $\geq 2t^2$ Knoten Tiefe $i \leq h$: $\geq 2t^{i-1}$ Knoten

$$\Rightarrow n \geq 1 + (t-1) \sum_{i=1}^{h} (2t^{i-1})$$

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten Tiefe 1: ≥ 2 Knoten Tiefe 2: $\geq 2t$ Knoten
  Tiefe 3: $\geq 2t^2$ Knoten Tiefe $i \leq h$: $\geq 2t^{i-1}$ Knoten

$$\Rightarrow n \geq 1 + (t-1) \sum_{i=1}^{h} (2t^{i-1}) = 1 + 2(t-1) \sum_{i=0}^{h-1} t^i$$

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten Tiefe 1: ≥ 2 Knoten Tiefe 2: $\geq 2t$ Knoten
  Tiefe 3: $\geq 2t^2$ Knoten Tiefe $i \leq h$: $\geq 2t^{i-1}$ Knoten

$$\Rightarrow n \geq 1 + (t-1) \sum_{i=1}^{h} (2t^{i-1}) = 1 + 2(t-1) \sum_{i=0}^{h-1} t^i = 1 + 2(t-1) \frac{t^h - 1}{t-1}$$

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten Tiefe 1: ≥ 2 Knoten Tiefe 2: $\geq 2t$ Knoten
  Tiefe 3: $\geq 2t^2$ Knoten Tiefe $i \leq h$: $\geq 2t^{i-1}$ Knoten

$$\Rightarrow n \geq 1 + (t-1) \sum_{i=1}^{h} (2t^{i-1}) = 1 + 2(t-1) \sum_{i=0}^{h-1} t^i = 1 + 2(t-1) \frac{t^h - 1}{t-1} = 2t^h - 1.$$

Theorem 4.9

Für die **Höhe h** eines jeden B-Baumes der Ordnung $t \geq 2$ mit $n \geq 1$ Daten gilt

$$h \leq \log_t \left(\frac{n+1}{2} \right).$$

Beweis:

- Wurzel **mindestens ein Datum**, jeder andere Knoten **mindestens t − 1 Daten**
- Wurzel **mindestens zwei Kinder** (sofern kein Blatt),
  andere innere Knoten **mindestens t Kinder**
- Tiefe 0: ein Knoten Tiefe 1: ≥ 2 Knoten Tiefe 2: $\geq 2t$ Knoten
  Tiefe 3: $\geq 2t^2$ Knoten Tiefe $i \leq h$: $\geq 2t^{i-1}$ Knoten

$$\Rightarrow n \geq 1 + (t-1) \sum_{i=1}^{h} (2t^{i-1}) = 1 + 2(t-1) \sum_{i=0}^{h-1} t^i = 1 + 2(t-1) \frac{t^h - 1}{t-1} = 2t^h - 1.$$

Das Theorem folgt durch Umstellen dieser Ungleichung nach t. □

Vergleich zu AVL-Bäumen:

$$\log_t\left(\frac{n+1}{2}\right) = \frac{\log_2((n+1)/2)}{\log_2 t} = \frac{\log_2(n+1)-1}{\log_2 t} \approx \frac{\log_2(n+1)}{\log_2 t}$$

Vergleich zu AVL-Bäumen:

$$\log_t\left(\frac{n+1}{2}\right) = \frac{\log_2((n+1)/2)}{\log_2 t} = \frac{\log_2(n+1)-1}{\log_2 t} \approx \frac{\log_2(n+1)}{\log_2 t}$$

Zahl der Speicherzugriffe sinkt etwa um Faktor $\log_2 t$.

4.2.2 B-Bäume

$v.\ell$: **Anzahl an Daten** in v

$v.\text{key}(1), \ldots, v.\text{key}(v.\ell)$: **Schlüssel** in v

$v.\text{child}(1), \ldots, v.\text{child}(v.\ell + 1)$: **Kinder** von v

```
B-TREE-SEARCH(Node v, int k)
1   if (v == null) return null;
2   int i = 1;
3   while ((i <= v.ℓ) && (k > v.key(i))) i++;
4   if ((i <= v.ℓ) && (k == v.key(i))) return (v, i);
5   return B-TREE-SEARCH(v.child(i),k);
```

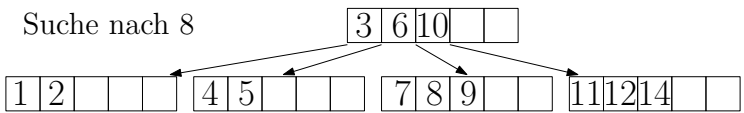
4.2.2 B-Bäume

$v.\ell$: **Anzahl an Daten** in v

$v.\text{key}(1), \ldots, v.\text{key}(v.\ell)$: **Schlüssel** in v

$v.\text{child}(1), \ldots, v.\text{child}(v.\ell + 1)$: **Kinder** von v

```
B-TREE-SEARCH(Node v, int k)
1   if (v == null) return null;
2   int i = 1;
3   while ((i <= v.ℓ) && (k > v.key(i))) i++;
4   if ((i <= v.ℓ) && (k == v.key(i))) return (v, i);
5   return B-TREE-SEARCH(v.child(i),k);
```



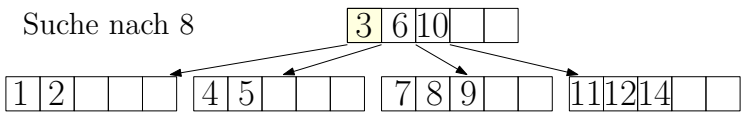
4.2.2 B-Bäume

$v.\ell$: **Anzahl an Daten** in v

$v.\text{key}(1), \ldots, v.\text{key}(v.\ell)$: **Schlüssel** in v

$v.\text{child}(1), \ldots, v.\text{child}(v.\ell + 1)$: **Kinder** von v

```
B-TREE-SEARCH(Node v, int k)
1   if (v == null) return null;
2   int i = 1;
3   while ((i <= v.ℓ) && (k > v.key(i))) i++;
4   if ((i <= v.ℓ) && (k == v.key(i))) return (v, i);
5   return B-TREE-SEARCH(v.child(i),k);
```



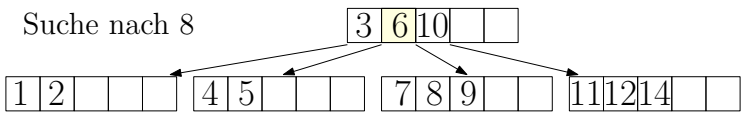
4.2.2 B-Bäume

$v.\ell$: **Anzahl an Daten** in v

$v.\text{key}(1), \ldots, v.\text{key}(v.\ell)$: **Schlüssel** in v

$v.\text{child}(1), \ldots, v.\text{child}(v.\ell + 1)$: **Kinder** von v

```
B-TREE-SEARCH(Node v, int k)
1   if (v == null) return null;
2   int i = 1;
3   while ((i <= v.ℓ) && (k > v.key(i))) i++;
4   if ((i <= v.ℓ) && (k == v.key(i))) return (v, i);
5   return B-TREE-SEARCH(v.child(i),k);
```



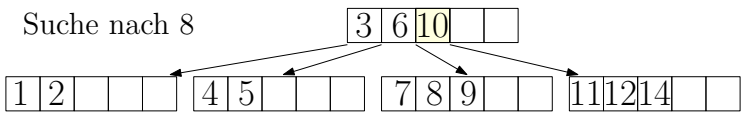
4.2.2 B-Bäume

$v.\ell$: **Anzahl an Daten** in v

$v.\text{key}(1), \ldots, v.\text{key}(v.\ell)$: **Schlüssel** in v

$v.\text{child}(1), \ldots, v.\text{child}(v.\ell + 1)$: **Kinder** von v

```
B-TREE-SEARCH(Node v, int k)
1   if (v == null) return null;
2   int i = 1;
3   while ((i <= v.ℓ) && (k > v.key(i))) i++;
4   if ((i <= v.ℓ) && (k == v.key(i))) return (v, i);
5   return B-TREE-SEARCH(v.child(i),k);
```



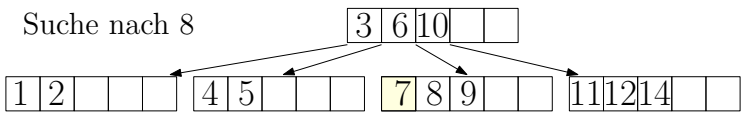
4.2.2 B-Bäume

$v.\ell$: **Anzahl an Daten** in v

$v.\text{key}(1), \ldots, v.\text{key}(v.\ell)$: **Schlüssel** in v

$v.\text{child}(1), \ldots, v.\text{child}(v.\ell + 1)$: **Kinder** von v

```
B-TREE-SEARCH(Node v, int k)
1   if (v == null) return null;
2   int i = 1;
3   while ((i <= v.ℓ) && (k > v.key(i))) i++;
4   if ((i <= v.ℓ) && (k == v.key(i))) return (v, i);
5   return B-TREE-SEARCH(v.child(i),k);
```



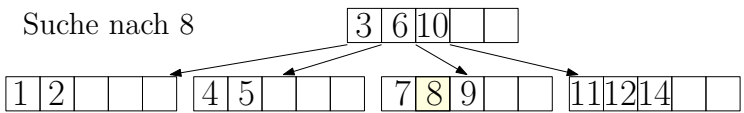
4.2.2 B-Bäume

$v.\ell$: **Anzahl an Daten** in v

$v.\text{key}(1), \ldots, v.\text{key}(v.\ell)$: **Schlüssel** in v

$v.\text{child}(1), \ldots, v.\text{child}(v.\ell + 1)$: **Kinder** von v

```
B-TREE-SEARCH(Node v, int k)
1   if (v == null) return null;
2   int i = 1;
3   while ((i <= v.ℓ) && (k > v.key(i))) i++;
4   if ((i <= v.ℓ) && (k == v.key(i))) return (v, i);
5   return B-TREE-SEARCH(v.child(i),k);
```



4.2.2 B-Bäume

$v.\ell$: **Anzahl an Daten** in v

$v.\text{key}(1), \ldots, v.\text{key}(v.\ell)$: **Schlüssel** in v

$v.\text{child}(1), \ldots, v.\text{child}(v.\ell + 1)$: **Kinder** von v

```
B-TREE-SEARCH(Node v, int k)
1   if (v == null) return null;
2   int i = 1;
3   while ((i <= v.ℓ) && (k > v.key(i))) i++;
4   if ((i <= v.ℓ) && (k == v.key(i))) return (v, i);
5   return B-TREE-SEARCH(v.child(i),k);
```

Laufzeit: $O(th) = O(t \log_t(n))$

Seitenzugriffe: $O(\log_t(n))$

4.2.2 B-Bäume

Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

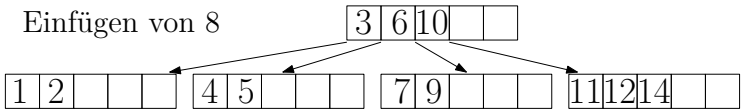
   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

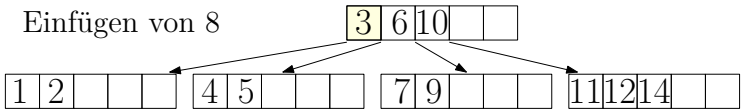


Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

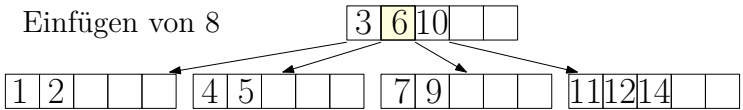


Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

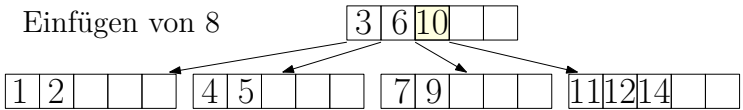


Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

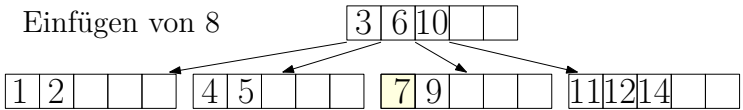


Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

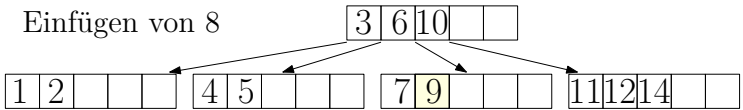


Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

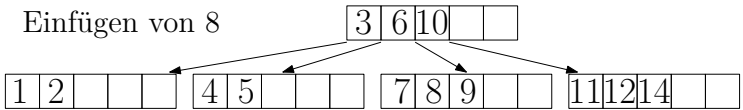


Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

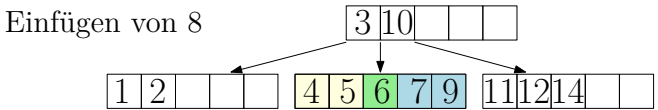


Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

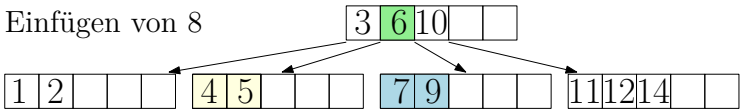


Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

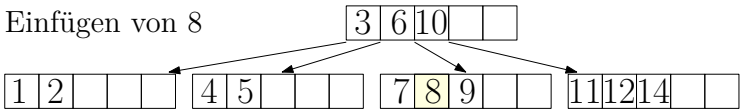


Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k

   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.



Einfügen von Datum x mit Schlüssel $k = x$.key

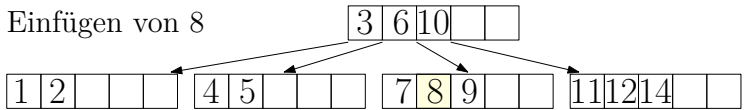
INSERT(Element x)

1 Suche nach k

Annahme: **Suche endet erfolglos in Blatt v**.

2 Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.

3 Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

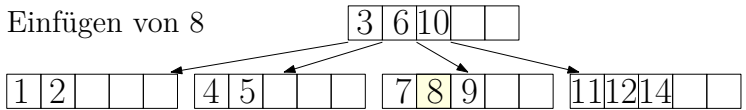


Zeile 3: v enthält $2t - 1$ Daten. Eins wird nach oben geschoben, bleiben $2(t - 1)$.

Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k
   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.



Zeile 3: v enthält $2t - 1$ Daten. Eins wird nach oben geschoben, bleiben $2(t - 1)$.

Achtung: Wenn Elternknoten von v bereits $2t - 1$ Daten enthält, wird dort analog verfahren.

4.2.2 B-Bäume

Einfügen von Datum x mit Schlüssel $k = x$.key

INSERT(Element x)

1. Suche nach k
   Annahme: **Suche endet erfolglos in Blatt v**.
2. Falls v weniger als $2t - 1$ Daten: **Füge x in v ein**.
3. Sonst: **Teile v in zwei Knoten** und füge x anschließend ein.

Laufzeit: $O(th) = O(t \log_t(n))$
Seitenzugriffe: $O(\log_t(n))$

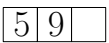
Zeile 3: v enthält $2t - 1$ Daten. Eins wird nach oben geschoben, bleiben $2(t - 1)$.
Achtung: Wenn Elternknoten von v bereits $2t - 1$ Daten enthält, wird dort analog verfahren.

Beispiel:

INSERT(9)

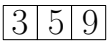
Beispiel:

INSERT(3)

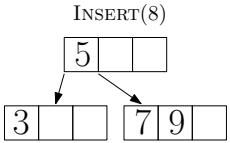


Beispiel:

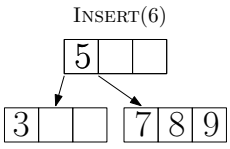
INSERT(7)



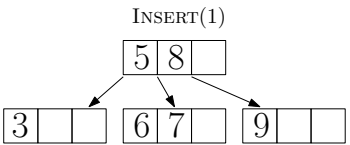
Beispiel:



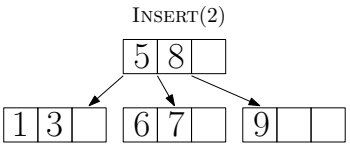
Beispiel:



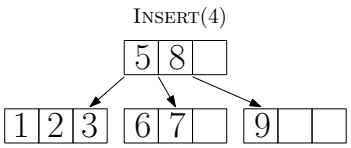
Beispiel:



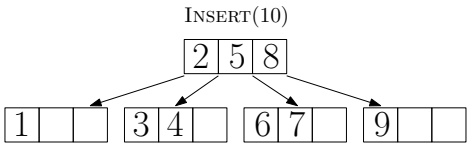
Beispiel:



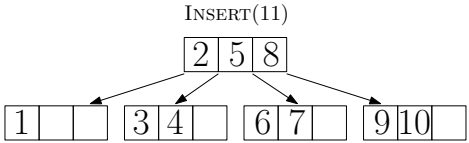
Beispiel:



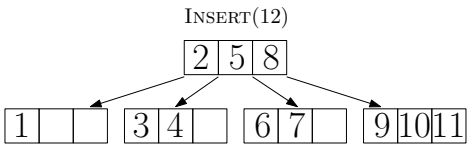
Beispiel:



Beispiel:

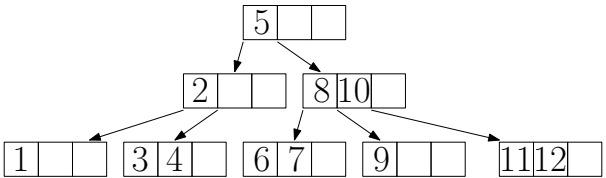


Beispiel:



4.2.2 B-Bäume

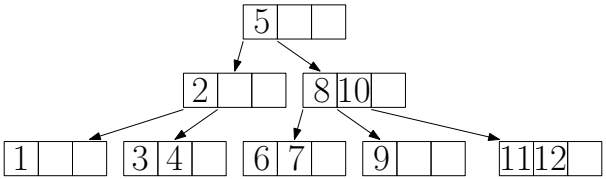
Beispiel:



Löschen von Schlüssel k

DELETE(Schlüssel k)

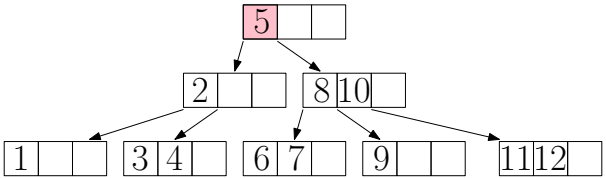
1. Suche nach k
   Annahme: **Suche endet erfolgreich in Knoten v**.
2. Falls v kein Blatt: **Suche größten Schlüssel a $<$ k**. Dieser liegt in einem Blatt u.
   Tausche a und k.
3. **Lösche k** aus Blatt u.
4. Falls u danach $t - 2$ Schlüssel enthält, müssen **Daten rotiert werden**.



Löschen von Schlüssel k

DELETE(Schlüssel k)

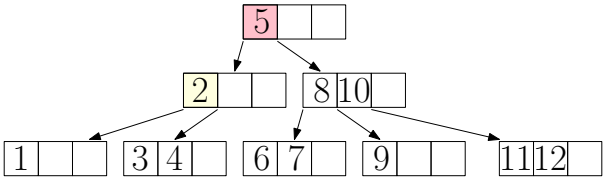
1. Suche nach k
   Annahme: **Suche endet erfolgreich in Knoten v**.
2. Falls v kein Blatt: **Suche größten Schlüssel a $<$ k**. Dieser liegt in einem Blatt u.
   Tausche a und k.
3. **Lösche k** aus Blatt u.
4. Falls u danach $t - 2$ Schlüssel enthält, müssen **Daten rotiert werden**.



Löschen von Schlüssel k

DELETE(Schlüssel k)

1. Suche nach k
   Annahme: **Suche endet erfolgreich in Knoten v**.
2. Falls v kein Blatt: **Suche größten Schlüssel a $<$ k**. Dieser liegt in einem Blatt u.
   Tausche a und k.
3. **Lösche k** aus Blatt u.
4. Falls u danach $t - 2$ Schlüssel enthält, müssen **Daten rotiert werden**.



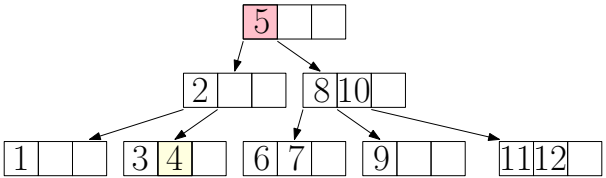
Löschen von Schlüssel k

DELETE(Schlüssel k)

1. Suche nach k

   Annahme: **Suche endet erfolgreich in Knoten v**.
2. Falls v kein Blatt: **Suche größten Schlüssel a $<$ k**. Dieser liegt in einem Blatt u.

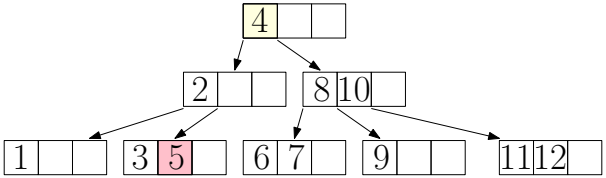
   Tausche a und k.
3. **Lösche k** aus Blatt u.
4. Falls u danach $t - 2$ Schlüssel enthält, müssen **Daten rotiert werden**.



Löschen von Schlüssel k

DELETE(Schlüssel k)

1. Suche nach k
   Annahme: **Suche endet erfolgreich in Knoten v**.
2. Falls v kein Blatt: **Suche größten Schlüssel a $<$ k**. Dieser liegt in einem Blatt u.
   Tausche a und k.
3. **Lösche k** aus Blatt u.
4. Falls u danach $t - 2$ Schlüssel enthält, müssen **Daten rotiert werden**.



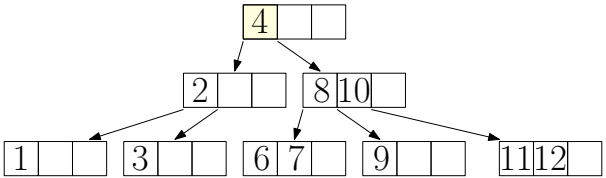
Löschen von Schlüssel k

DELETE(Schlüssel k)

1. Suche nach k

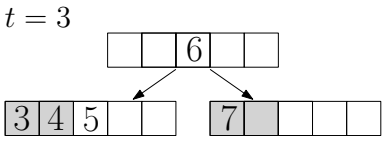
   Annahme: **Suche endet erfolgreich in Knoten v**.
2. Falls v kein Blatt: **Suche größten Schlüssel a $<$ k**. Dieser liegt in einem Blatt u.

   Tausche a und k.
3. **Lösche k** aus Blatt u.
4. Falls u danach $t - 2$ Schlüssel enthält, müssen **Daten rotiert werden**.



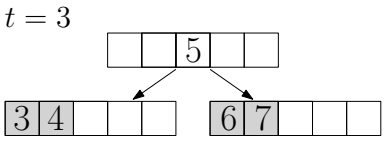
Rotation: u besitzt nur noch $t - 2$ Daten.

1. Fall: Ein Nachbar von u hat mindestens t Daten.



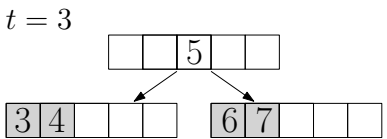
Rotation: u besitzt nur noch $t - 2$ Daten.

1. Fall: Ein Nachbar von u hat mindestens t Daten.

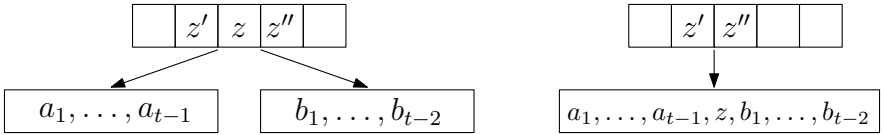


Rotation: u besitzt nur noch $t-2$ Daten.

1. Fall: Ein Nachbar von u hat mindestens t Daten.

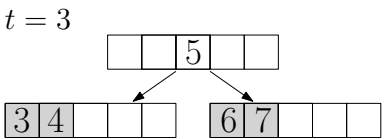


2. Fall: Alle Nachbarn von u haben nur $t-1$ Daten.

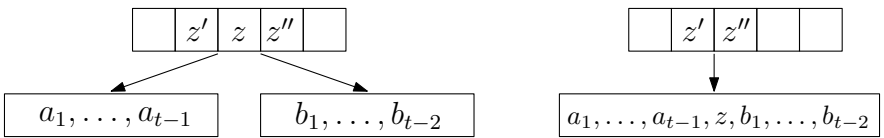


Rotation: u besitzt nur noch $t-2$ Daten.

1. Fall: Ein Nachbar von u hat mindestens t Daten.

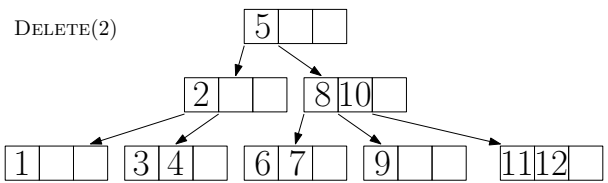


2. Fall: Alle Nachbarn von u haben nur $t-1$ Daten.

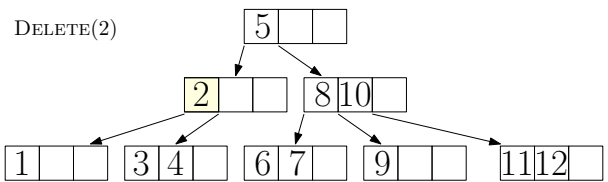


Achtung: Wenn Elternknoten von u nur $t-1$ Daten enthält, wird dort analog verfahren.

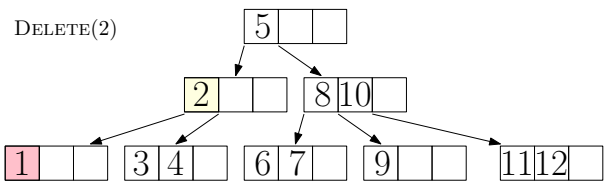
Beispiel:



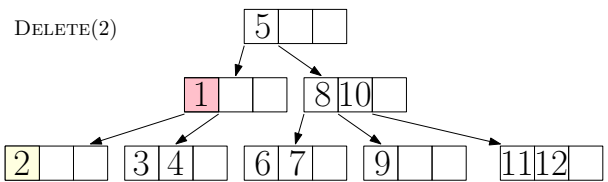
Beispiel:



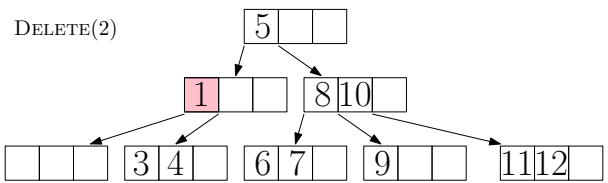
Beispiel:



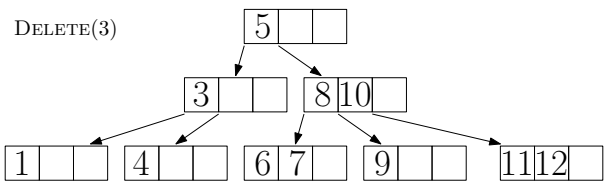
Beispiel:



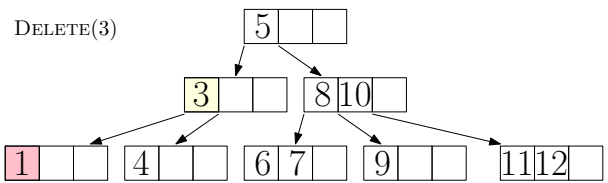
Beispiel:



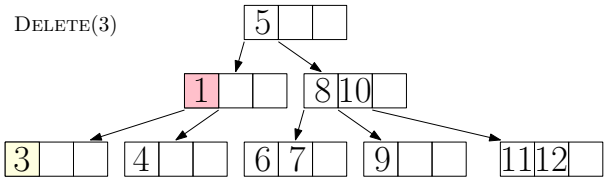
Beispiel:



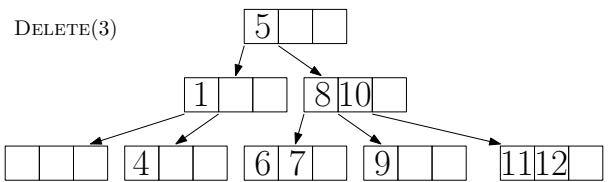
Beispiel:



Beispiel:

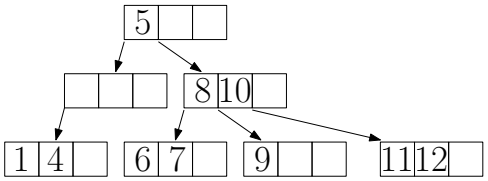


Beispiel:



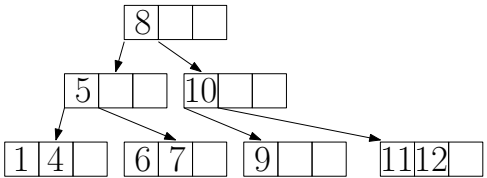
Beispiel:

DELETE(3)



Beispiel:

DELETE(3)



Theorem 4.10

Die Operationen SEARCH, INSERT und DELETE benötigen in einem B-Baum der Ordnung t mit n Daten eine Laufzeit von $O(th) = O(t \log_t(n))$ und sie benötigen $O(h) = O(\log_t(n))$ Seitenzugriffe, wenn alle Daten eines Knotens in derselben Seite gespeichert sind.

Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

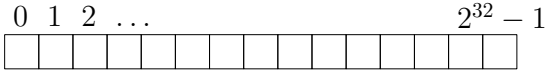
Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Array der Länge $|U|$.

Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

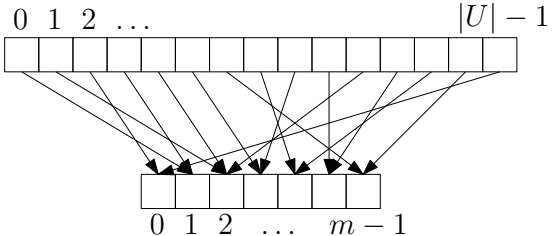
Speichere Daten in Array der Länge $|U|$. **Nicht praktikabel!**



Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Array der Länge $|U|$. **Nicht praktikabel!**



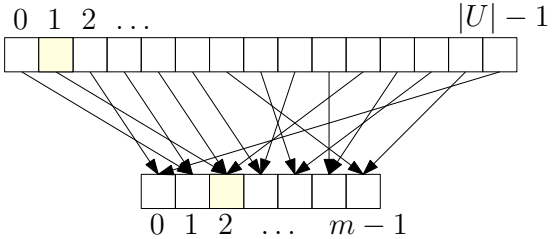
Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$ für $m \ll |U|$

Nutze Feld der Länge m. Speichere Element mit Schlüssel k an der Position $h(k)$.

Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Array der Länge $|U|$. **Nicht praktikabel!**



Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$ für $m \ll |U|$

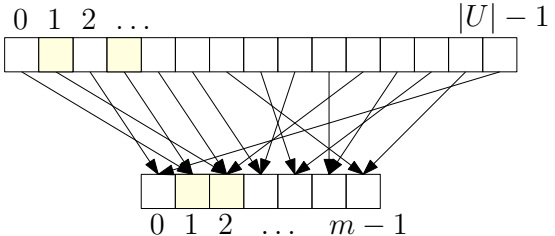
Nutze Feld der Länge m. Speichere Element mit Schlüssel k an der Position $h(k)$.

4.3 Hashing

Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Array der Länge $|U|$. **Nicht praktikabel!**



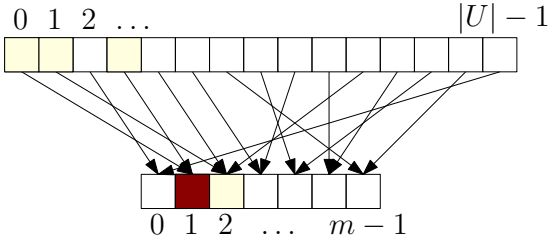
Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$ für $m \ll |U|$

Nutze Feld der Länge m. Speichere Element mit Schlüssel k an der Position $h(k)$.

Hashtabelle: Und noch eine Datenstruktur für dynamische Mengen ...

Sei U das Universum, aus dem die Schlüssel kommen.

Speichere Daten in Array der Länge $|U|$. **Nicht praktikabel!**



Hashfunktion $h\colon U \to \{0, 1, \ldots, m-1\}$ für $m \ll |U|$

Nutze Feld der Länge m. Speichere Element mit Schlüssel k an der Position $h(k)$.

Problem: Kollisionen können auftreten. Wir benötigen Strategien, um damit umzugehen.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

- **Divisionsmethode**: $h(k) = k \bmod m$
  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

- **Divisionsmethode**: $h(k) = k \bmod m$
  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.
- **Multiplikationsmethode**: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ für Konstante $A \in (0, 1)$.
  Die Wahl $A \approx \frac{\sqrt{5}-1}{2} = 0{,}61803\ldots$ hat sich als gut herausgestellt.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

- **Divisionsmethode**: $h(k) = k \bmod m$
  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.
- **Multiplikationsmethode**: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ für Konstante $A \in (0, 1)$.
  Die Wahl $A \approx \frac{\sqrt{5}-1}{2} = 0{,}61803\ldots$ hat sich als gut herausgestellt.

In Java hat jede Klasse Methode „**int** hashCode()“.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

- **Divisionsmethode**: $h(k) = k \bmod m$
  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.
- **Multiplikationsmethode**: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ für Konstante $A \in (0, 1)$.
  Die Wahl $A \approx \frac{\sqrt{5}-1}{2} = 0{,}61803\ldots$ hat sich als gut herausgestellt.

In Java hat jede Klasse Methode „**int** hashCode()“.
Bei String berechnet diese $\sum_{i=0}^{\ell-1} s_i \cdot 31^i$ für Zeichenkette $s_{\ell-1} s_{\ell-2} \ldots s_1 s_0$.

4.3.1 Hashfunktionen

Bei **guter Hashfunktion** ist $h^{-1}(i) = \{k \in U \mid h(k) = i\}$ für jedes i in etwa gleich groß.

Theoretisch sind alle solche Hashfunktionen gleich gut, **praktisch nicht** (z. B. erster Buchstabe einer Zeichenkette als Hashfunktion).

Methoden für $U = \mathbb{N}$:

- **Divisionsmethode**: $h(k) = k \bmod m$
  m sollte keine Zweierpotenz sein. m typischerweise Primzahl.
- **Multiplikationsmethode**: $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ für Konstante $A \in (0, 1)$.
  Die Wahl $A \approx \frac{\sqrt{5}-1}{2} = 0{,}61803\ldots$ hat sich als gut herausgestellt.

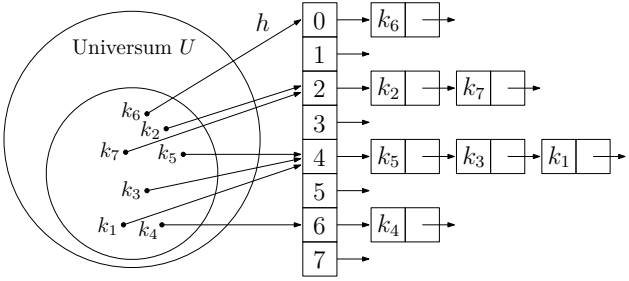
In Java hat jede Klasse Methode „**int** hashCode()“.
Bei String berechnet diese $\sum_{i=0}^{\ell-1} s_i \cdot 31^i$ für Zeichenkette $s_{\ell-1} s_{\ell-2} \ldots s_1 s_0$.

Annahme: $h(k)$ kann in konstanter Zeit berechnet werden.

4.3.2 Hashing mit verketteten Listen

Hashing mit verketteten Listen:

Lege Feld $T[0 \ldots m-1]$ an. Dabei sei jedes $T[i]$ **Zeiger auf verkettete Liste**.



4.3.2 Hashing mit verketteten Listen

Hashing mit verketteten Listen:

Lege Feld $T[0 \ldots m-1]$ an. Dabei sei jedes $T[i]$ **Zeiger auf verkettete Liste**.

- SEARCH(k):
  Gib das Element mit dem Schlüssel k in der Liste $T[h(k)]$ zurück, sofern es existiert.
- INSERT(x):
  Füge das Element x an den Anfang der Liste $T[h(x.\text{key})]$ ein.
- DELETE(k):
  Lösche das Element mit dem Schlüssel k aus der Liste $T[h(k)]$, sofern es existiert.

4.3.2 Hashing mit verketteten Listen

Hashing mit verketteten Listen:

Lege Feld $T[0 \ldots m-1]$ an. Dabei sei jedes $T[i]$ **Zeiger auf verkettete Liste**.

- SEARCH(k):
  Gib das Element mit dem Schlüssel k in der Liste $T[h(k)]$ zurück, sofern es existiert.
- INSERT(x):
  Füge das Element x an den Anfang der Liste $T[h(x.\text{key})]$ ein.
- DELETE(k):
  Lösche das Element mit dem Schlüssel k aus der Liste $T[h(k)]$, sofern es existiert.

Laufzeit: im Worst Case $O(n)$

4.3.2 Hashing mit verketteten Listen

Annahme: h bildet jeden Schlüssel **uniform zufällig** und **unabhängig** auf ein Element aus $\{0, 1, \ldots, m-1\}$ ab (**uniformes Hashing**).

4.3.2 Hashing mit verketteten Listen

Annahme: h bildet jeden Schlüssel **uniform zufällig** und **unabhängig** auf ein Element aus $\{0, 1, \ldots, m-1\}$ ab (**uniformes Hashing**).

Betrachte Hashtabelle, in die die Schlüssel $k_1, \ldots, k_n$ in dieser Reihenfolge eingefügt wurden. Sei $\alpha = n/m$ der **Auslastungsfaktor**.

4.3.2 Hashing mit verketteten Listen

Annahme: h bildet jeden Schlüssel **uniform zufällig** und **unabhängig** auf ein Element aus $\{0, 1, \ldots, m-1\}$ ab (**uniformes Hashing**).

Betrachte Hashtabelle, in die die Schlüssel $k_1, \ldots, k_n$ in dieser Reihenfolge eingefügt wurden. Sei $\alpha = n/m$ der **Auslastungsfaktor**.

Theorem 4.11

Unter der Annahme des uniformen Hashings benötigt eine erfolglose Suche nach einem Schlüssel, der sich nicht in der Hashtabelle befindet, im Erwartungswert eine Laufzeit von $\Theta(1 + \alpha)$.

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

4.3.2 Hashing mit verketteten Listen

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(x_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(x_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|]$$

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(x_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right]$$

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(x_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} \mathbf{E}[X_j]$$

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(x_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} \mathbf{E}[X_j] = \frac{n}{m} = \alpha$$

Beweis: Suche nach k mit $i = h(k)$. Uns interessiert $E[|T[i]|]$.

Für jedes $j \in \{1, \ldots, n\}$ definiere Zufallsvariable X_j:

$$X_j = \begin{cases} 1 & \text{falls } h(k_j) = i, \\ 0 & \text{falls } h(k_j) \neq i. \end{cases}$$

Es gilt $\mathbf{Pr}[X_j = 1] = \mathbf{Pr}[h(x_j) = i] = 1/m$ und demnach auch $\mathbf{E}[X_j] = 1/m$.

$$\Rightarrow \mathbf{E}[|T[i]|] = \mathbf{E}\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} \mathbf{E}[X_j] = \frac{n}{m} = \alpha$$

Somit folgen wir bei der Suche nach dem Schlüssel k im Erwartungswert $1 + \alpha$ vielen Zeigern, wobei $+1$ für den Null-Zeiger des letzten Eintrages der Liste $T[i]$ steht. □

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1 + \alpha)$.

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1 + \alpha)$.

Beweis: Schlüssel $k_1, \ldots, k_n$ wurden in dieser Reihenfolge eingefügt.

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1 + \alpha)$.

Beweis: Schlüssel $k_1, \ldots, k_n$ wurden in dieser Reihenfolge eingefügt.

Wie vielen Zeigern folgen wir bei Suche nach zufällig ausgewähltem Schlüssel k_i?

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1 + \alpha)$.

Beweis: Schlüssel $k_1, \ldots, k_n$ wurden in dieser Reihenfolge eingefügt.

Wie vielen Zeigern folgen wir bei Suche nach zufällig ausgewähltem Schlüssel k_i?
Da neue Elemente an den Anfang der Liste eingefügt werden, ist dazu nur von Interesse, wie viele Schlüssel **nach k_i** in die Liste $T[h(k_i)]$ eingefügt werden.

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1+\alpha)$.

Beweis: Schlüssel $k_1, \ldots, k_n$ wurden in dieser Reihenfolge eingefügt.

Wie vielen Zeigern folgen wir bei Suche nach zufällig ausgewähltem Schlüssel k_i?
Da neue Elemente an den Anfang der Liste eingefügt werden, ist dazu nur von Interesse, wie viele Schlüssel **nach k_i** in die Liste $T[h(k_i)]$ eingefügt werden.

Für jedes $i \in \{1, \ldots, n\}$ und jedes $j \in \{i+1, \ldots, n\}$ definieren wir eine Zufallsvariable

$$X_{ij} = \begin{cases} 1 & \text{falls } h(k_j) = h(k_i), \\ 0 & \text{falls } h(k_j) \neq h(k_i). \end{cases}$$

4.3.2 Hashing mit verketteten Listen

Theorem 4.12

Unter der Annahme des uniformen Hashings benötigt eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert eine Laufzeit von $\Theta(1+\alpha)$.

Beweis: Schlüssel $k_1, \ldots, k_n$ wurden in dieser Reihenfolge eingefügt.

Wie vielen Zeigern folgen wir bei Suche nach zufällig ausgewähltem Schlüssel k_i?
Da neue Elemente an den Anfang der Liste eingefügt werden, ist dazu nur von Interesse, wie viele Schlüssel **nach k_i** in die Liste $T[h(k_i)]$ eingefügt werden.

Für jedes $i \in \{1, \ldots, n\}$ und jedes $j \in \{i+1, \ldots, n\}$ definieren wir eine Zufallsvariable

$$X_{ij} = \begin{cases} 1 & \text{falls } h(k_j) = h(k_i), \\ 0 & \text{falls } h(k_j) \neq h(k_i). \end{cases}$$

Es gilt $\mathbf{Pr}[X_{ij} = 1] = \mathbf{Pr}[h(k_i) = h(k_j)] = 1/m$ und demnach auch $\mathbf{E}[X_{ij}] = 1/m$.

4.3.2 Hashing mit verketteten Listen

Für festes $i \in \{1, \ldots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right]$$

Für festes $i \in \{1, \ldots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right] = 1 + \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}]$$

4.3.2 Hashing mit verketteten Listen

Für festes $i \in \{1, \ldots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right] = 1 + \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] = 1 + \frac{n-i}{m}.$$

4.3.2 Hashing mit verketteten Listen

Für festes $i \in \{1, \ldots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right] = 1 + \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] = 1 + \frac{n-i}{m}.$$

Durchschnitt über alle i:

$$\frac{1}{n} \sum_{i=1}^{n} \left(1 + \frac{n-i}{m}\right)$$

Für festes $i \in \{1, \ldots, n\}$ gilt

$$\mathbf{E}\left[1 + \sum_{j=i+1}^{n} X_{ij}\right] = 1 + \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}] = 1 + \frac{n-i}{m}.$$

Durchschnitt über alle i:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1 + \frac{n-i}{m}\right) = \frac{1}{n}\left(n + \sum_{i=1}^{n}\frac{n-i}{m}\right) = 1 + \frac{1}{n}\sum_{i=1}^{n}\frac{n-i}{m} = 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$= 1 + \frac{1}{nm}\sum_{i=1}^{n-1} i = 1 + \frac{1}{nm}\cdot\frac{n(n-1)}{2} = 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{1}{2m} = \Theta(1+\alpha).$$

Damit ist das Theorem bewiesen. □

4.3.3 Geschlossenes Hashing

geschlossenes Hashing oder **Hashing mit offener Adressierung**:
Speichere alle Daten in Feld T.

4.3.3 Geschlossenes Hashing

geschlossenes Hashing oder **Hashing mit offener Adressierung**:
Speichere alle Daten in Feld T.

Betrachte dazu Hashfunktionen der Form

$$h\colon U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

4.3.3 Geschlossenes Hashing

geschlossenes Hashing oder **Hashing mit offener Adressierung**:
Speichere alle Daten in Feld T.

Betrachte dazu Hashfunktionen der Form

$$h\colon U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Idee: Teste beim Einfügen die Positionen $h(k, 0)$, $h(k, 1)$, $h(k, 2)$ usw., bis freie Position gefunden.

4.3.3 Geschlossenes Hashing

geschlossenes Hashing oder **Hashing mit offener Adressierung**:
Speichere alle Daten in Feld T.

Betrachte dazu Hashfunktionen der Form

$$h\colon U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}.$$

Idee: Teste beim Einfügen die Positionen $h(k, 0)$, $h(k, 1)$, $h(k, 2)$ usw., bis freie Position gefunden.

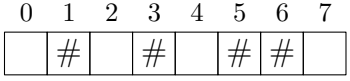
Annahme: $h(k, 0), h(k, 1), \ldots, h(k, m-1)$ ist Permutation von $\{0, 1, \ldots, m-1\}$.

4.3.3 Geschlossenes Hashing

```
INSERT(x)
1    for (int i = 0; i < m; i++) {
2        j = h(x.key, i);
3        if (T[j] == null) {
4            T[j] = x;
5            return j;
6        }
7    }
8    return „Überlauf“;
```

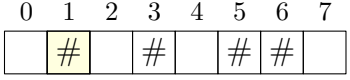
4.3.3 Geschlossenes Hashing

```
INSERT(x)
1    for (int i = 0; i < m; i++) {
2        j = h(x.key, i);
3        if (T[j] == null) {
4            T[j] = x;
5            return j;
6        }
7    }
8    return „Überlauf“;
```



4.3.3 Geschlossenes Hashing

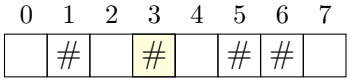
```
INSERT(x)
1    for (int i = 0; i < m; i++) {
2        j = h(x.key, i);
3        if (T[j] == null) {
4            T[j] = x;
5            return j;
6        }
7    }
8    return „Überlauf“;
```



$h(k, 0) = 1$

4.3.3 Geschlossenes Hashing

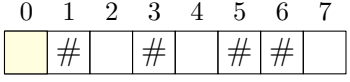
```
INSERT(x)
1   for (int i = 0; i < m; i++) {
2       j = h(x.key, i);
3       if (T[j] == null) {
4           T[j] = x;
5           return j;
6       }
7   }
8   return „Überlauf“;
```



$h(k, 1) = 3$

4.3.3 Geschlossenes Hashing

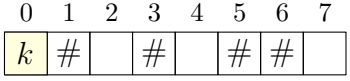
```
INSERT(x)
1    for (int i = 0; i < m; i++) {
2        j = h(x.key, i);
3        if (T[j] == null) {
4            T[j] = x;
5            return j;
6        }
7    }
8    return „Überlauf“;
```



$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
INSERT(x)
1    for (int i = 0; i < m; i++) {
2        j = h(x.key, i);
3        if (T[j] == null) {
4            T[j] = x;
5            return j;
6        }
7    }
8    return „Überlauf“;
```



$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
INSERT(x)
1   for (int i = 0; i < m; i++) {
2       j = h(x.key, i);
3       if (T[j] == null) {
4           T[j] = x;
5           return j;
6       }
7   }
8   return „Überlauf“;
```

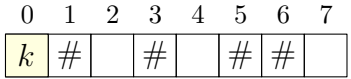
```
SEARCH(k)
1   for (int i = 0; i < m; i++) {
2       j = h(k, i);
3       if ((T[j] ≠ null) && (T[j].key == k)) {
4           return j;
5       } else if (T[j] == null) {
6           return −1; //„nicht vorhanden“;
7       }
8   }
9   return −1; // „nicht vorhanden“;
```

0	1	2	3	4	5	6	7
k	#		#		#	#	

$h(k, 2) = 0$

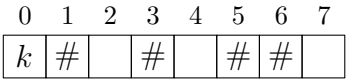
4.3.3 Geschlossenes Hashing

```
INSERT(x)
1   for (int i = 0; i < m; i++) {
2       j = h(x.key, i);
3       if (T[j] == null) {
4           T[j] = x;
5           return j;
6       }
7   }
8   return „Überlauf“;
```



$h(k, 2) = 0$

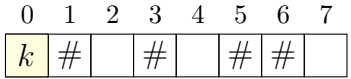
```
SEARCH(k)
1   for (int i = 0; i < m; i++) {
2       j = h(k, i);
3       if ((T[j] ≠ null) && (T[j].key == k)) {
4           return j;
5       } else if (T[j] == null) {
6           return −1; //„nicht vorhanden“;
7       }
8   }
9   return −1; // „nicht vorhanden“;
```



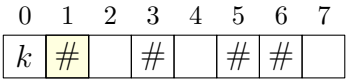
4.3.3 Geschlossenes Hashing

```
INSERT(x)
1   for (int i = 0; i < m; i++) {
2       j = h(x.key, i);
3       if (T[j] == null) {
4           T[j] = x;
5           return j;
6       }
7   }
8   return „Überlauf“;
```

```
SEARCH(k)
1   for (int i = 0; i < m; i++) {
2       j = h(k, i);
3       if ((T[j] ≠ null) && (T[j].key == k)) {
4           return j;
5       } else if (T[j] == null) {
6           return −1; //„nicht vorhanden“;
7       }
8   }
9   return −1; // „nicht vorhanden“;
```



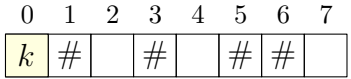
$h(k, 2) = 0$



$h(k, 0) = 1$

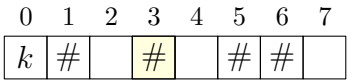
4.3.3 Geschlossenes Hashing

```
INSERT(x)
1   for (int i = 0; i < m; i++) {
2       j = h(x.key, i);
3       if (T[j] == null) {
4           T[j] = x;
5           return j;
6       }
7   }
8   return „Überlauf“;
```



$h(k, 2) = 0$

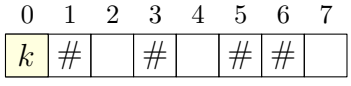
```
SEARCH(k)
1   for (int i = 0; i < m; i++) {
2       j = h(k, i);
3       if ((T[j] ≠ null) && (T[j].key == k)) {
4           return j;
5       } else if (T[j] == null) {
6           return −1; //„nicht vorhanden“;
7       }
8   }
9   return −1; // „nicht vorhanden“;
```



$h(k, 1) = 3$

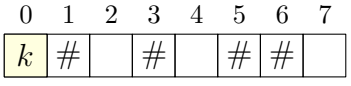
4.3.3 Geschlossenes Hashing

```
INSERT(x)
1   for (int i = 0; i < m; i++) {
2       j = h(x.key, i);
3       if (T[j] == null) {
4           T[j] = x;
5           return j;
6       }
7   }
8   return „Überlauf“;
```



$h(k, 2) = 0$

```
SEARCH(k)
1   for (int i = 0; i < m; i++) {
2       j = h(k, i);
3       if ((T[j] ≠ null) && (T[j].key == k)) {
4           return j;
5       } else if (T[j] == null) {
6           return −1; //„nicht vorhanden“;
7       }
8   }
9   return −1; // „nicht vorhanden“;
```



$h(k, 2) = 0$

```
DELETE(k)
1    j = SEARCH(k);
2    if (j ≠ −1) {

4        T[j] = null;
5    }
```

```
DELETE(k)
1    j = SEARCH(k);
2    if (j ≠ −1) {

4        T[j] = null;
5    }
```

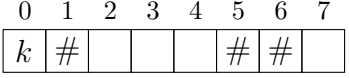
0	1	2	3	4	5	6	7
k	#		k'		#	#	

```
DELETE(k)
1    j = SEARCH(k);
2    if (j ≠ −1) {

4        T[j] = null;
5    }
```

0	1	2	3	4	5	6	7
k	#		k'		#	#	

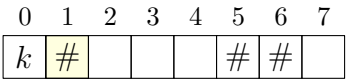
DELETE(k')

```
DELETE(k)
1    j = SEARCH(k);
2    if (j ≠ −1) {

4        T[j] = null;
5    }
```



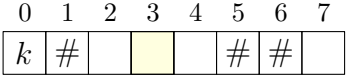
DELETE(k')

```
DELETE(k)
1    j = SEARCH(k);
2    if (j ≠ −1) {

4        T[j] = null;
5    }
```



$h(k, 0) = 1$

```
DELETE(k)
1    j = SEARCH(k);
2    if (j ≠ −1) {

4        T[j] = null;
5    }
```



$h(k, 1) = 3$

```
DELETE(k)
1    j = SEARCH(k);
2    if (j ≠ −1) {

4        T[j] = null;
5    }
```

0	1	2	3	4	5	6	7
k	#				#	#	

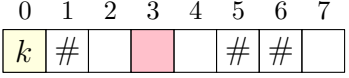
$h(k, 2) = 0$

```
DELETE(k)
1    j = SEARCH(k);
2    if (j ≠ −1) {
3         Del[j] = true;
4         T[j] = null;
5    }
```

0	1	2	3	4	5	6	7
k	#				#	#	

$h(k, 2) = 0$

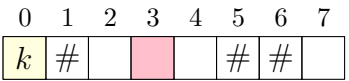
```
DELETE(k)
1    j = SEARCH(k);
2    if (j ≠ −1) {
3         Del[j] = true;
4         T[j] = null;
5    }
```



$h(k, 2) = 0$

4.3.3 Geschlossenes Hashing

```
DELETE(k)
1   j = SEARCH(k);
2   if (j ≠ −1) {
3       Del[j] = true;
4       T[j] = null;
5   }
```

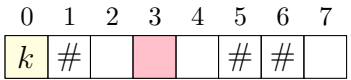


$h(k, 2) = 0$

```
SEARCH(k)
1   for (int i = 0; i < m; i++) {
2       j = h(k, i);
3       if ((T[j] ≠ null) && (T[j].key == k)) {
4           return j;
5       } else if ((T[j] == null)                    ) {
6           return −1; //„nicht vorhanden“;
7       }
8   }
9   return −1; // „nicht vorhanden“;
```

4.3.3 Geschlossenes Hashing

```
DELETE(k)
1   j = SEARCH(k);
2   if (j ≠ −1) {
3       Del[j] = true;
4       T[j] = null;
5   }
```



$h(k, 2) = 0$

```
SEARCH(k)
1   for (int i = 0; i < m; i++) {
2       j = h(k, i);
3       if ((T[j] ≠ null) && (T[j].key == k)) {
4           return j;
5       } else if ((T[j] == null) && (Del[j] == false) ) {
6           return −1; //„nicht vorhanden“;
7       }
8   }
9   return −1; // „nicht vorhanden“;
```

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

$$h(k, i) = (h'(k) + i) \bmod m.$$

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

$$h(k, i) = (h'(k) + i) \bmod m.$$

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

$$h(k, i) = (h'(k) + i) \bmod m.$$

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' : U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

$$h(k, i) = (h'(k) + i) \bmod m.$$

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:

Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' \colon U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

$$h(k,i) = (h'(k) + i) \bmod m.$$

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:

Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

- $\mathbf{Pr}[Z = 1] = \mathbf{Pr}[h'(k) \in \{0,1\}] = 2/8 = 1/4$

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' \colon U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

$$h(k,i) = (h'(k) + i) \bmod m.$$

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:

Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

- $\mathbf{Pr}[Z = 1] = \mathbf{Pr}[h'(k) \in \{0, 1\}] = 2/8 = 1/4$
- $\mathbf{Pr}[Z = 2] = \mathbf{Pr}[h'(k) = 2] = 1/8$

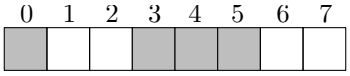
4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' \colon U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

$$h(k,i) = (h'(k) + i) \bmod m.$$

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:



Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

- $\mathbf{Pr}[Z = 1] = \mathbf{Pr}[h'(k) \in \{0,1\}] = 2/8 = 1/4$
- $\mathbf{Pr}[Z = 2] = \mathbf{Pr}[h'(k) = 2] = 1/8$
- $\mathbf{Pr}[Z = 6] = \mathbf{Pr}[h'(k) \in \{3,4,5,6\}] = 4/8 = 1/2$

4.3.3 Geschlossenes Hashing

Sondierungsreihenfolgen: gegeben sei „normale“ Hashfunktion $h' \colon U \to \{0, \ldots, m-1\}$

lineares Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k)+1, h'(k)+2, \ldots$, d. h.

$$h(k,i) = (h'(k) + i) \bmod m.$$

Nachteil: Tendenz, längere zusammenhängende Blöcke zu bilden:

Einfügen eines neues Datums mit Schlüssel k an Position Z, wobei $h'(k)$ uniform zufällig

- $\mathbf{Pr}[Z = 1] = \mathbf{Pr}[h'(k) \in \{0,1\}] = 2/8 = 1/4$
- $\mathbf{Pr}[Z = 2] = \mathbf{Pr}[h'(k) = 2] = 1/8$
- $\mathbf{Pr}[Z = 6] = \mathbf{Pr}[h'(k) \in \{3,4,5,6\}] = 4/8 = 1/2$
- $\mathbf{Pr}[Z = 7] = \mathbf{Pr}[h'(k) = 7] = 1/8$

4.3.3 Geschlossenes Hashing

quadratisches Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k) + 1^2, h'(k) + 2^2, h'(k) + 3^2, \ldots$ (jeweils modulo m).

quadratisches Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k) + 1^2, h'(k) + 2^2, h'(k) + 3^2, \ldots$ (jeweils modulo m).

Gilt $h(k_1, 0) = h(k_2, 1)$, so werden (anders als beim linearen Sondieren) verschiedene Sequenzen durchlaufen. Es gilt insbesondere im Allgemeinen nicht $h(k_1, 1) = h(k_2, 2)$.

4.3.3 Geschlossenes Hashing

quadratisches Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k) + 1^2, h'(k) + 2^2, h'(k) + 3^2, \ldots$ (jeweils modulo m).

Gilt $h(k_1, 0) = h(k_2, 1)$, so werden (anders als beim linearen Sondieren) verschiedene Sequenzen durchlaufen. Es gilt insbesondere im Allgemeinen nicht $h(k_1, 1) = h(k_2, 2)$.

doppeltes Hashing: gegeben seien zwei „normale“ Hashfunktionen $h' \colon U \to \{0, \ldots, m-1\}$ und $h'' \colon U \to \{0, \ldots, m-1\}$

4.3.3 Geschlossenes Hashing

quadratisches Sondieren: Für Schlüssel $k \in U$ testen wir die Positionen $h'(k), h'(k) + 1^2, h'(k) + 2^2, h'(k) + 3^2, \ldots$ (jeweils modulo m).

Gilt $h(k_1, 0) = h(k_2, 1)$, so werden (anders als beim linearen Sondieren) verschiedene Sequenzen durchlaufen. Es gilt insbesondere im Allgemeinen nicht $h(k_1, 1) = h(k_2, 2)$.

doppeltes Hashing: gegeben seien zwei „normale“ Hashfunktionen $h' \colon U \to \{0, \ldots, m-1\}$ und $h'' \colon U \to \{0, \ldots, m-1\}$

Die Hashfunktion h ist dann definiert als

$$h(k, i) = h'(k) + i \cdot h''(k) \bmod m.$$

4.3.3 Geschlossenes Hashing

Analyse von geschlossenem Hashing: (ohne Löschen)

Analyse von geschlossenem Hashing: (ohne Löschen)

Uniformes Hashing: Beim Sondieren werden die Positionen in einer uniform zufälligen Reihenfolge getestet.

4.3.3 Geschlossenes Hashing

Analyse von geschlossenem Hashing: (ohne Löschen)

Uniformes Hashing: Beim Sondieren werden die Positionen in einer uniform zufälligen Reihenfolge getestet.

Sei $\alpha = n/m$ wieder der Auslastungsfaktor. Es gilt $\alpha \leq 1$.

Theorem 4.13

Unter der Annahme des uniformen Hashings untersucht eine erfolglose Suche nach einem Schlüssel, der sich nicht in der Hashtabelle befindet, beim geschlossenen Hashing im Erwartungswert höchstens $1/(1-\alpha)$ Positionen.

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.
- Analog kann man für jedes $i \in \mathbb{N}$ argumentieren:

$$\mathbf{Pr}[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \alpha^{i-1}.$$

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.
- Analog kann man für jedes $i \in \mathbb{N}$ argumentieren:

$$\mathbf{Pr}[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \alpha^{i-1}.$$

Aus der obigen Überlegung folgt

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i]$$

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.
- Analog kann man für jedes $i \in \mathbb{N}$ argumentieren:

$$\mathbf{Pr}[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \alpha^{i-1}.$$

Aus der obigen Überlegung folgt

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1}$$

Beweis:

Es bezeichne X die Zufallsvariable, die die Anzahl untersuchter Positionen angibt.

- Es gilt $\mathbf{Pr}[X \geq 1] = 1 = \alpha^0$, da in jedem Fall eine Position betrachtet wird.
- Es gilt $\mathbf{Pr}[X \geq 2] = \frac{n}{m} = \alpha^1$.
- Es gilt $\mathbf{Pr}[X \geq 3] = \frac{n}{m} \cdot \frac{n-1}{m-1} \leq \alpha^2$, wobei $\frac{n-1}{m-1} \leq \frac{n}{m}$ aus $m \geq n$ folgt.
- Analog kann man für jedes $i \in \mathbb{N}$ argumentieren:

$$\mathbf{Pr}[X \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \alpha^{i-1}.$$

Aus der obigen Überlegung folgt

$$\mathbf{E}[X] = \sum_{i=1}^{\infty} \mathbf{Pr}[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha},$$

womit das Theorem bewiesen ist. □

Theorem 4.14

Unter der Annahme des uniformen Hashings untersucht eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert höchstens $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$ Positionen.

Theorem 4.14

Unter der Annahme des uniformen Hashings untersucht eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert höchstens $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$ Positionen.

Beweis:

Suche nach Schlüssel x erzeugt die gleiche Sequenz wie das Einfügen des Schlüssels.

Theorem 4.14

Unter der Annahme des uniformen Hashings untersucht eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert höchstens $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$ Positionen.

Beweis:

Suche nach Schlüssel x erzeugt die gleiche Sequenz wie das Einfügen des Schlüssels.

Beim Einfügen des i-ten Schlüssels beträgt der Auslastungsfaktor $\alpha_i = (i-1)/m$

Theorem 4.14

Unter der Annahme des uniformen Hashings untersucht eine erfolgreiche Suche nach einem uniform zufällig gewählten Schlüssel in der Hashtabelle im Erwartungswert höchstens $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$ Positionen.

Beweis:

Suche nach Schlüssel x erzeugt die gleiche Sequenz wie das Einfügen des Schlüssels.

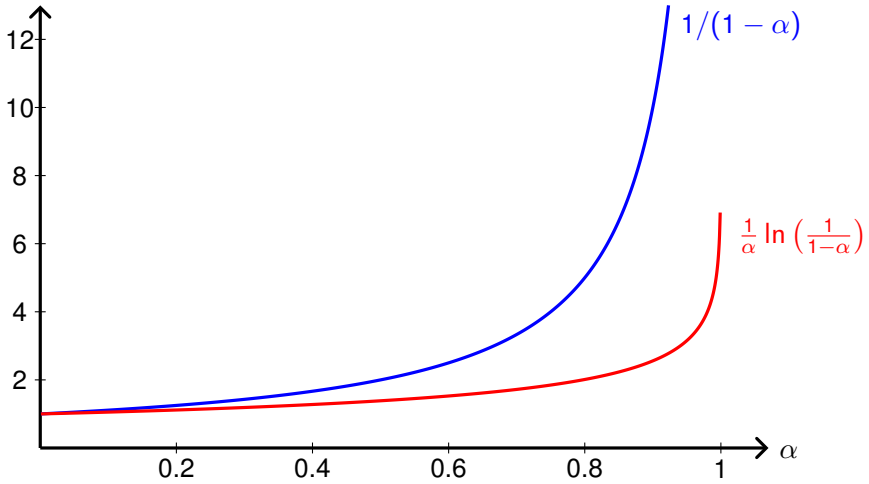
Beim Einfügen des i-ten Schlüssels beträgt der Auslastungsfaktor $\alpha_i = (i-1)/m$ und demnach beträgt gemäß Theorem 4.13 die Anzahl der Positionen, die betrachtet werden, im Erwartungswert höchstens $1/(1-\alpha_i) = 1/(1-(i-1)/m)$.

Die Bildung des Durchschnitts über alle Schlüssel ergibt nun

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\frac{1}{1-(i-1)/m} &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{1-i/m} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\leq \frac{1}{\alpha}\int_{k=m-n}^{m}\frac{1}{x}\,dx \\
&= \frac{1}{\alpha}\ln\left(\frac{m}{m-n}\right) \\
&= \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right),
\end{aligned}$$

womit das Theorem bewiesen ist. □

4.3.3 Geschlossenes Hashing



4.4 Heaps

Max-Heap

Binärer Baum, in dem in jedem Knoten ein Datum gespeichert ist.

Für jeden Knoten v mit Inhalt x gilt, dass alle Daten im **linken oder rechten Teilbaum** von v Schlüssel besitzen, die **höchstens so groß** wie der Schlüssel von x sind.

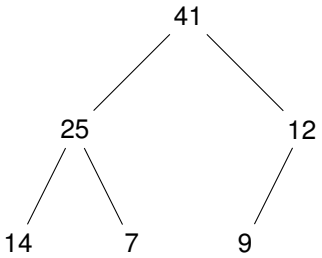
Darüber hinaus ist der Binärbaum bis auf möglicherweise die letzte Ebene **vollständig**. Die letzte Ebene wird **von links nach rechts** gefüllt.

Max-Heap

Binärer Baum, in dem in jedem Knoten ein Datum gespeichert ist.

Für jeden Knoten v mit Inhalt x gilt, dass alle Daten im **linken oder rechten Teilbaum** von v Schlüssel besitzen, die **höchstens so groß** wie der Schlüssel von x sind.

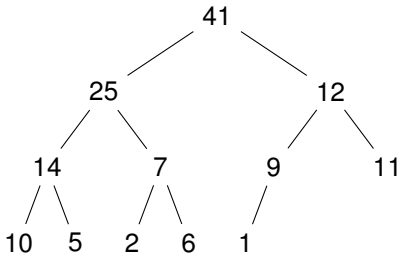
Darüber hinaus ist der Binärbaum bis auf möglicherweise die letzte Ebene **vollständig**. Die letzte Ebene wird **von links nach rechts** gefüllt.



Speichern eines Max-Heaps in einem Feld

Betrachte Max-Heap mit heap-size vielen Elementen.

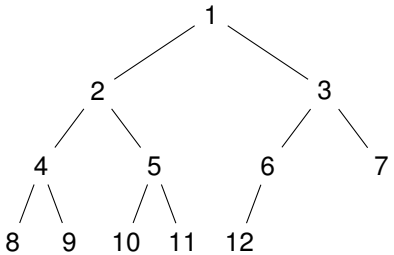
Speichere diesen in Feld a mit Positionen $1, \ldots,$ heap-size.



Speichern eines Max-Heaps in einem Feld

Betrachte Max-Heap mit heap-size vielen Elementen.

Speichere diesen in Feld a mit Positionen $1, \ldots,$ heap-size.

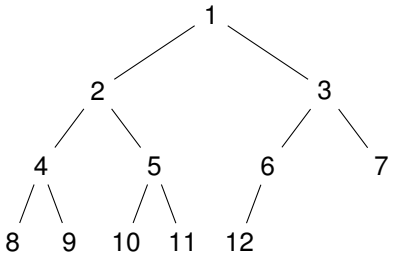


$$a = [41, 25, 12, 14, 7, 9, 11, 10, 5, 2, 6, 1]$$

Speichern eines Max-Heaps in einem Feld

Betrachte Max-Heap mit heap-size vielen Elementen.

Speichere diesen in Feld a mit Positionen $1, \ldots,$ heap-size.



$$a = [41, 25, 12, 14, 7, 9, 11, 10, 5, 2, 6, 1]$$

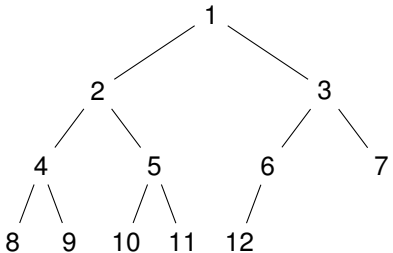
Kinder des Knotens an Position i an Positionen $2i$ und $2i + 1$.

Elternknoten zu dem an Position $i > 1$ gespeicherten Knoten an Stelle $\lfloor i/2 \rfloor$.

Speichern eines Max-Heaps in einem Feld

Betrachte Max-Heap mit heap-size vielen Elementen.

Speichere diesen in Feld a mit Positionen $1, \ldots,$ heap-size.



$$a = [41, 25, 12, 14, 7, 9, 11, 10, 5, 2, 6, 1]$$

Kinder des Knotens an Position i an Positionen $2i$ und $2i + 1$.

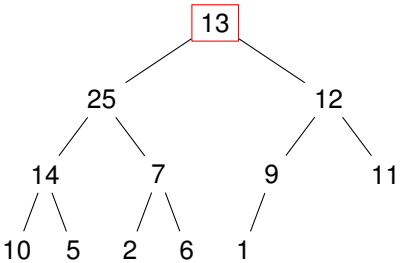
Elternknoten zu dem an Position $i > 1$ gespeicherten Knoten an Stelle $\lfloor i/2 \rfloor$.

Heap-Bedingung: Für alle i $a[i]$.key $\geq a[2i]$.key und $a[i]$.key $\geq a[2i + 1]$.key.

Erzeugen eines Max-Heaps

Annahme: Heap-Eigenschaft überall erfüllt bis auf evtl. die Wurzel.

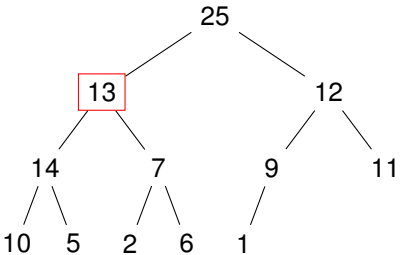
```
MAX-HEAPIFY(int i)
1   ℓ = 2i; r = 2i + 1; largest = i;
2   if (ℓ ≤ heap-size && a[ℓ].key > a[largest].key)
3       largest = ℓ;
4   if (r ≤ heap-size && a[r].key > a[largest].key)
5       largest = r;
6   if (largest ≠ i) {
7       vertausche a[i] und a[largest];
8       MAX-HEAPIFY(largest);
9   }
```



Erzeugen eines Max-Heaps

Annahme: Heap-Eigenschaft überall erfüllt bis auf evtl. die Wurzel.

```
MAX-HEAPIFY(int i)
1   ℓ = 2i; r = 2i + 1; largest = i;
2   if (ℓ ≤ heap-size && a[ℓ].key > a[largest].key)
3       largest = ℓ;
4   if (r ≤ heap-size && a[r].key > a[largest].key)
5       largest = r;
6   if (largest ≠ i) {
7       vertausche a[i] und a[largest];
8       MAX-HEAPIFY(largest);
9   }
```

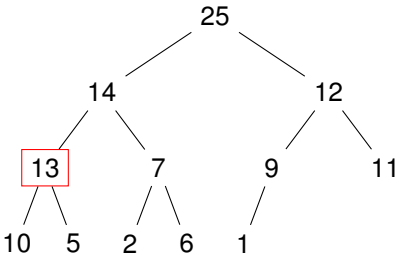


4.4.1 Heap-Eigenschaft herstellen

Erzeugen eines Max-Heaps

Annahme: Heap-Eigenschaft überall erfüllt bis auf evtl. die Wurzel.

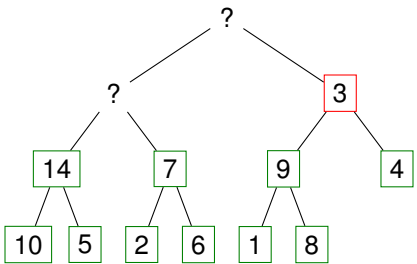
```
MAX-HEAPIFY(int i)
1   ℓ = 2i; r = 2i + 1; largest = i;
2   if (ℓ ≤ heap-size && a[ℓ].key > a[largest].key)
3       largest = ℓ;
4   if (r ≤ heap-size && a[r].key > a[largest].key)
5       largest = r;
6   if (largest ≠ i) {
7       vertausche a[i] und a[largest];
8       MAX-HEAPIFY(largest);
9   }
```



4.4.1 Heap-Eigenschaft herstellen

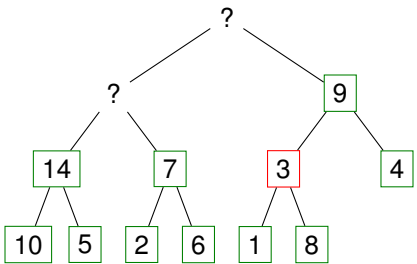
Lemma 4.15

Gilt vor dem Aufruf von MAX-HEAPIFY(i) die Heap-Bedingung für alle $j > i$, so gilt sie anschließend für alle $j \geq i$.



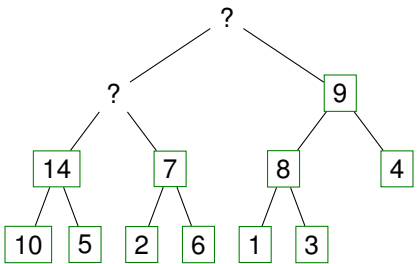
Lemma 4.15

Gilt vor dem Aufruf von MAX-HEAPIFY(i) die Heap-Bedingung für alle $j > i$, so gilt sie anschließend für alle $j \geq i$.



Lemma 4.15

Gilt vor dem Aufruf von MAX-HEAPIFY(i) die Heap-Bedingung für alle $j > i$, so gilt sie anschließend für alle $j \geq i$.



4.4.1 Heap-Eigenschaft herstellen

Erzeugen eines Max-Heaps:

```
BUILD-HEAP()
1    for (i = ⌊a.length/2⌋; i ≥ 1; i--) {
2        MAX-HEAPIFY(i);
3    }
```

4.4.1 Heap-Eigenschaft herstellen

Erzeugen eines Max-Heaps:

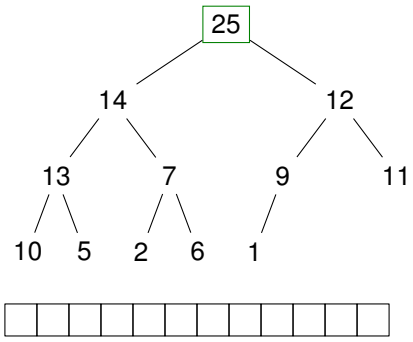
```
BUILD-HEAP()
1   for (i = ⌊a.length/2⌋; i ≥ 1; i--) {
2       MAX-HEAPIFY(i);
3   }
```

Lemma 4.16

Nach der Ausführung von BUILD-HEAP gilt die Heap-Eigenschaft für alle i.

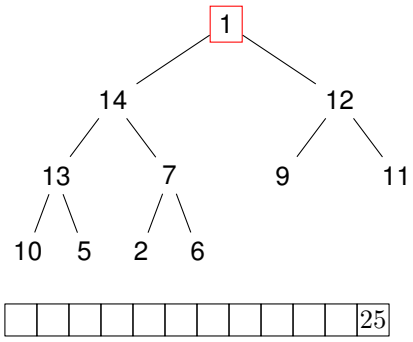
Heapsort

```
HEAPSORT(int[ ] a)
      // Das Feld a besitzt die Positionen 1, . . . , n.
1     BUILD-HEAP();
2     heap-size = n;
3     while (heap-size > 0) {
4           vertausche a[1] und a[heap-size];
5           heap-size−−;
6           MAX-HEAPIFY(1);
7     }
```



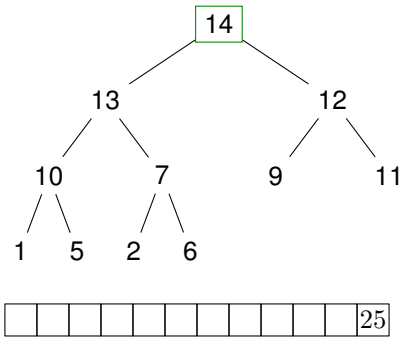
Heapsort

```
HEAPSORT(int[ ] a)
      // Das Feld a besitzt die Positionen 1, . . . , n.
1     BUILD-HEAP();
2     heap-size = n;
3     while (heap-size > 0) {
4           vertausche a[1] und a[heap-size];
5           heap-size−−;
6           MAX-HEAPIFY(1);
7     }
```



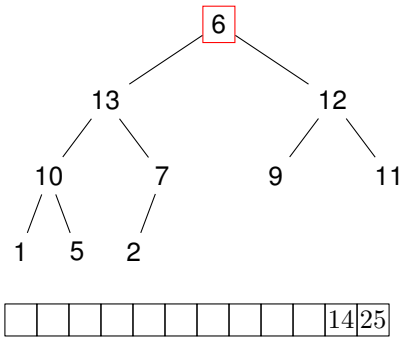
Heapsort

```
HEAPSORT(int[ ] a)
      // Das Feld a besitzt die Positionen 1, . . . , n.
1     BUILD-HEAP();
2     heap-size = n;
3     while (heap-size > 0) {
4           vertausche a[1] und a[heap-size];
5           heap-size−−;
6           MAX-HEAPIFY(1);
7     }
```



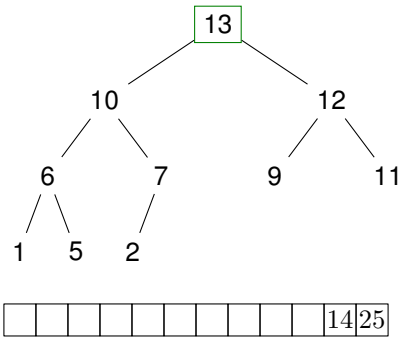
Heapsort

```
HEAPSORT(int[ ] a)
      // Das Feld a besitzt die Positionen 1, . . . , n.
1     BUILD-HEAP();
2     heap-size = n;
3     while (heap-size > 0) {
4           vertausche a[1] und a[heap-size];
5           heap-size−−;
6           MAX-HEAPIFY(1);
7     }
```



Heapsort

```
HEAPSORT(int[ ] a)
      // Das Feld a besitzt die Positionen 1, . . . , n.
1     BUILD-HEAP();
2     heap-size = n;
3     while (heap-size > 0) {
4           vertausche a[1] und a[heap-size];
5           heap-size−−;
6           MAX-HEAPIFY(1);
7     }
```



Heapsort

```
HEAPSORT(int[ ] a)
      // Das Feld a besitzt die Positionen 1, . . . , n.
1     BUILD-HEAP();
2     heap-size = n;
3     while (heap-size > 0) {
4           vertausche a[1] und a[heap-size];
5           heap-size−−;
6           MAX-HEAPIFY(1);
7     }
```

1	2	5	6	7	9	10	11	12	13	14	25

Heapsort

```
HEAPSORT(int[ ] a)
      // Das Feld a besitzt die Positionen 1, . . . , n.
1     BUILD-HEAP();
2     heap-size = n;
3     while (heap-size > 0) {
4           vertausche a[1] und a[heap-size];
5           heap-size−−;
6           MAX-HEAPIFY(1);
7     }
```

1	2	5	6	7	9	10	11	12	13	14	25

Theorem 4.17

HEAPSORT sortiert jedes Feld mit n Einträgen in Zeit $O(n \log n)$.

4.4.3 Prioritätswarteschlangen

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

4.4.3 Prioritätswarteschlangen

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- INSERT(x, k): Füge ein neues Element x mit Schlüssel k ein.

4.4.3 Prioritätswarteschlangen

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- INSERT(x, k): Füge ein neues Element x mit Schlüssel k ein.
- FIND-MAX$()$: Gib ein Element mit dem größtem Schlüssel zurück.

4.4.3 Prioritätswarteschlangen

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- INSERT(x, k): Füge ein neues Element x mit Schlüssel k ein.
- FIND-MAX$()$: Gib ein Element mit dem größtem Schlüssel zurück.
- EXTRACT-MAX$()$: Entferne ein Element mit dem größtem Schlüssel und gib dieses Element zurück.

4.4.3 Prioritätswarteschlangen

Prioritätswarteschlange

Eine Prioritätswarteschlange speichert **Element/Schlüssel-Paare** und unterstützt die folgenden Operationen.

- INSERT(x, k): Füge ein neues Element x mit Schlüssel k ein.
- FIND-MAX$()$: Gib ein Element mit dem größtem Schlüssel zurück.
- EXTRACT-MAX$()$: Entferne ein Element mit dem größtem Schlüssel und gib dieses Element zurück.
- INCREASE-KEY(i, k): Erhöhe den Schlüssel des an Stelle i gespeicherten Objektes auf k.

```
EXTRACT-MAX()
1    if (heap-size > 0) {
2        max = a[1];
3        a[1] = a[heap-size];
4        heap-size−−;
5        MAX-HEAPIFY(1);
6        return max;
7    }
```

```
EXTRACT-MAX()
1    if (heap-size > 0) {
2        max = a[1];
3        a[1] = a[heap-size];
4        heap-size−−;
5        MAX-HEAPIFY(1);
6        return max;
7    }
```

```
INCREASE-KEY(i,k)
1    a[i].key = k;
2    while (i > 1 && a[i].key > a[⌊i/2⌋].key) {
3        vertausche a[i] und a[⌊i/2⌋];
4        i = ⌊i/2⌋;
5    }
```

```
EXTRACT-MAX()
1   if (heap-size > 0) {
2       max = a[1];
3       a[1] = a[heap-size];
4       heap-size−−;
5       MAX-HEAPIFY(1);
6       return max;
7   }
```

```
INCREASE-KEY(i,k)
1   a[i].key = k;
2   while (i > 1 && a[i].key > a[⌊i/2⌋].key) {
3       vertausche a[i] und a[⌊i/2⌋];
4       i = ⌊i/2⌋;
5   }
```

```
INSERT(x,k)
1   heap-size++;
2   a[heap-size] = x;
3   a[heap-size].key = −∞;
4   INCREASE-KEY(heap-size, k);
```

Theorem 4.18

Die Laufzeit der Operationen EXTRACT-MAX, INCREASE-KEY und INSERT beträgt in einem Heap mit n gespeicherten Daten $O(\log n)$. Die Operation FIND-MAX benötigt nur konstante Laufzeit.