Binäre Suche – Analyse

Java

```
function binsearch(int x) boolean :
l := 0;  r := n − 1;
while l ≤ r do
  m := ⌊(l + r)/2⌋;
  if a[m] < x then l := m + 1 fi;
  if a[m] > x then r := m − 1 fi;
  if a[m] = x then return true fi
od;
return false
```

Es sei $n = r - l + 1$ die Größe des aktuellen Unterarrays.

Im nächsten Durchgang ist die Größe $m - l$ oder $r - m$.

Binäre Suche – Analyse

Lemma

Es sei $a \in \mathbf{R}$ und $n \in \mathbf{N}$. Dann gilt

1. $\lfloor a + n \rfloor = \lfloor a \rfloor + n$
2. $\lceil a + n \rceil = \lceil a \rceil + n$
3. $\lfloor -a \rfloor = -\lceil a \rceil$

Binäre Suche – Analyse

Im nächsten Durchlauf ist die Größe des Arrays $m - l$ oder $r - m$.

Hierbei ist $m = \lfloor (l + r)/2 \rfloor$.

Die neue Größe ist also

- $m - l = \lfloor (l + r)/2 \rfloor - l = \lfloor (r - l)/2 \rfloor = \lfloor (n - 1)/2 \rfloor$ oder
- $r - m = r - \lfloor (l + r)/2 \rfloor = \lceil (r - l)/2 \rceil = \lceil (n - 1)/2 \rceil$.

Im schlimmsten Fall ist die neue Größe des Arrays

$$\lceil (n - 1)/2 \rceil.$$

Rekursionsgleichung für binäre Suche

Sei S_n die Anzahl der Schleifendurchläufe im schlimmsten Fall bei einer erfolglosen Suche.

Wir erhalten die Rekursionsgleichung

$$S_n = \begin{cases} 0 & \text{falls } n < 1, \\ 1 + S_{\lceil (n-1)/2 \rceil} & \text{falls } n \geq 1. \end{cases}$$

Die ersten Werte sind:

n	0	1	2	3	4	5	6	7	8
S_n	0	1	2	2	3	3	3	3	4

Wir suchen eine geschlossene Formel für S_n.

Rekursionsgleichung für binäre Suche

Sei S_n die Anzahl der Schleifendurchläufe im schlimmsten Fall bei einer erfolglosen Suche.

Wir erhalten die Rekursionsgleichung

$$S_n = \begin{cases} 0 & \text{falls } n < 1, \\ 1 + S_{\lceil (n-1)/2 \rceil} & \text{falls } n \geq 1. \end{cases}$$

Die ersten Werte sind:

n	0	1	2	3	4	5	6	7	8
S_n	0	1	2	2	3	3	3	3	4

Wir suchen eine geschlossene Formel für S_n.

Lösen der Rekursionsgleichung

Wir betrachten den Spezialfall $n = 2^k - 1$:

$$\left\lceil \frac{(2^k - 1) - 1}{2} \right\rceil = \left\lceil \frac{2^k - 2}{2} \right\rceil = \lceil 2^{k-1} - 1 \rceil = 2^{k-1} - 1.$$

Daher: $S_{2^k-1} = 1 + S_{2^{k-1}-1}$ für $k \geq 1$

$$\Rightarrow S_{2^k-1} = k + S_{2^0-1} = k$$

Lösen der Rekursionsgleichung

Wir betrachten den Spezialfall $n = 2^k - 1$:

$$\left\lceil \frac{(2^k - 1) - 1}{2} \right\rceil = \left\lceil \frac{2^k - 2}{2} \right\rceil = \lceil 2^{k-1} - 1 \rceil = 2^{k-1} - 1.$$

Daher: $S_{2^k-1} = 1 + S_{2^{k-1}-1}$ für $k \geq 1$

$$\Rightarrow S_{2^k-1} = k + S_{2^0-1} = k$$

Lösen der Rekursionsgleichung

Wir betrachten den Spezialfall $n = 2^k - 1$:

$$\left\lceil \frac{(2^k - 1) - 1}{2} \right\rceil = \left\lceil \frac{2^k - 2}{2} \right\rceil = \lceil 2^{k-1} - 1 \rceil = 2^{k-1} - 1.$$

Daher: $S_{2^k-1} = 1 + S_{2^{k-1}-1}$ für $k \geq 1$

$$\Rightarrow S_{2^k-1} = k + S_{2^0-1} = k$$

Binäre Suche – Analyse

n	0	1	2	3	4	5	6	7	8
S_n	0	1	2	2	3	3	3	3	4

Vermutung: $S_{2^k} = S_{2^k-1} + 1$

S_n steigt monoton $\Rightarrow S_n = k$, falls $2^{k-1} \leq n < 2^k$.

Oder falls $k - 1 \leq \log n < k$.

Dann wäre $S_n = \lfloor \log n \rfloor + 1$.

Binäre Suche – Analyse

n	0	1	2	3	4	5	6	7	8
S_n	0	1	2	2	3	3	3	3	4

Vermutung: $S_{2^k} = S_{2^k-1} + 1$

S_n steigt monoton $\Rightarrow S_n = k$, falls $2^{k-1} \leq n < 2^k$.

Oder falls $k - 1 \leq \log n < k$.

Dann wäre $S_n = \lfloor \log n \rfloor + 1$.

Binäre Suche – Analyse

n	0	1	2	3	4	5	6	7	8
S_n	0	1	2	2	3	3	3	3	4

Vermutung: $S_{2^k} = S_{2^k-1} + 1$

S_n steigt monoton $\Rightarrow S_n = k$, falls $2^{k-1} \leq n < 2^k$.

Oder falls $k - 1 \leq \log n < k$.

Dann wäre $S_n = \lfloor \log n \rfloor + 1$.

Binäre Suche – Analyse

Wir vermuten $S_n = \lfloor \log n \rfloor + 1$ für $n \geq 1$.

Induktion über n:

$S_1 = 1 = \lfloor \log 1 \rfloor + 1$

$n > 1$:

$$S_n = 1 + S_{\lceil (n-1)/2 \rceil} \overset{\text{I.V.}}{=} 1 + \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1.$$

Noch zu zeigen:

$$\lfloor \log n \rfloor = \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1$$

$\rightarrow$ Übungsaufgabe.

Binäre Suche – Analyse

Wir vermuten $S_n = \lfloor \log n \rfloor + 1$ für $n \geq 1$.

Induktion über n:

$S_1 = 1 = \lfloor \log 1 \rfloor + 1$

$n > 1$:

$$S_n = 1 + S_{\lceil (n-1)/2 \rceil} \overset{\text{I.V.}}{=} 1 + \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1.$$

Noch zu zeigen:

$$\lfloor \log n \rfloor = \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1$$

$\rightarrow$ Übungsaufgabe.

Binäre Suche – Analyse

Wir vermuten $S_n = \lfloor \log n \rfloor + 1$ für $n \geq 1$.

Induktion über n:

$S_1 = 1 = \lfloor \log 1 \rfloor + 1$

$n > 1$:

$$S_n = 1 + S_{\lceil (n-1)/2 \rceil} \overset{\text{I.V.}}{=} 1 + \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1.$$

Noch zu zeigen:

$$\lfloor \log n \rfloor = \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1$$

$\rightarrow$ Übungsaufgabe.

Binäre Suche – Analyse

Wir vermuten $S_n = \lfloor \log n \rfloor + 1$ für $n \geq 1$.

Induktion über n:

$S_1 = 1 = \lfloor \log 1 \rfloor + 1$

$n > 1$:

$$S_n = 1 + S_{\lceil (n-1)/2 \rceil} \overset{\text{I.V.}}{=} 1 + \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1.$$

Noch zu zeigen:

$$\lfloor \log n \rfloor = \lfloor \log \lceil (n-1)/2 \rceil \rfloor + 1$$

$\rightarrow$ Übungsaufgabe.

Binäre Suche – Analyse

Wir vermuten $S_n = \lfloor \log n \rfloor + 1$ für $n \geq 1$.

Induktion über n:

$S_1 = 1 = \lfloor \log 1 \rfloor + 1$

$n > 1$:

$$S_n = 1 + S_{\lceil (n-1)/2 \rceil} \overset{\text{I.V.}}{=} 1 + \left\lfloor \log \lceil (n-1)/2 \rceil \right\rfloor + 1.$$

Noch zu zeigen:

$$\lfloor \log n \rfloor = \left\lfloor \log \lceil (n-1)/2 \rceil \right\rfloor + 1$$

$\rightarrow$ Übungsaufgabe.

Binäre Suche – Analyse

Theorem

Binäre Suche benötigt im schlimmsten Fall genau

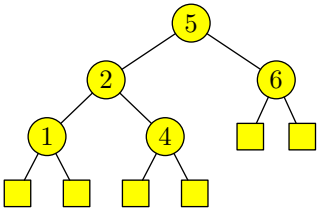
$$\lfloor \log n \rfloor + 1 = \log(n) + O(1)$$

viele Vergleiche.

Die Laufzeit ist $O(\log n)$.

Übersicht

Binäre Suchbäume



- Als assoziatives Array geeignet
- Schlüssel aus geordneter Menge
- Linke Kinder kleiner, rechte Kinder größer als Elternknoten
- Externe und interne Knoten
- Externe Knoten zu einem Sentinel zusammenfassen?

Binäre Suchbäume

Java

```
public class Searchtree<K extends Comparable<K>, D> extends AbstractMap<K, D> {
  protected Searchtreenode<K, D> root;
```

Java

```
class Searchtreenode<K extends Comparable<K>, D> {
  K key;
  D data;
  Searchtreenode<K, D> left, right, parent;
```

Binäre Suchbäume – Suchen

Java

```
public D find(K k) {
  if(root == null) return null;
  Searchtreenode⟨K, D⟩ n = root.findsubtree(k);
  return n == null ? null : n.data;
}
```

Im Gegensatz zu Liste:
Zusätzlicher Test auf **null**.

Binäre Suchbäume – Suchen

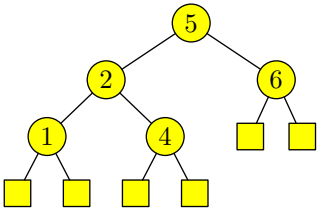
Java

```
Searchtreenode<K, D> findsubtree(K k) {
  int c = k.compareTo(key);
  if(c > 0) return right == null ? null : right.findsubtree(k);
  else if(c < 0) return left == null ? null : left.findsubtree(k);
  else return this;
}
```

Wieder zusätzlicher Test auf **null**.
Durch Sentinel verhindern!

Besser: Iterativ statt rekursiv.

Binäre Suchbäume – Einfügen

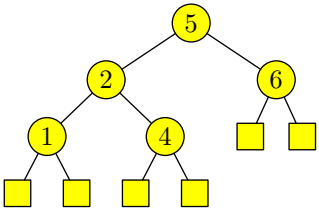


Wo fügen wir 3 ein?

Wie fügen wir es ein?

In den richtigen externen Knoten!

Binäre Suchbäume – Einfügen

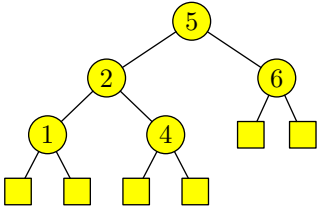


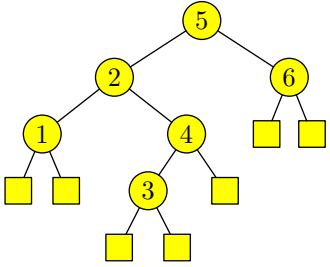
Wo fügen wir 3 ein?

Wie fügen wir es ein?

In den richtigen externen Knoten!

Binäre Suchbäume – Einfügen





Wo fügen wir 3 ein?

Wie fügen wir es ein?

In den richtigen externen Knoten!

Binäre Suchbäume – Einfügen

Java

```
public void insert(K k, D d) {
  if(root == null) root = newNode(k, d);
  else root.insert(newNode(k, d));
}
```

Binäre Suchbäume – Einfügen

Java

```
public void insert(Searchtreenode⟨K, D⟩ n) {
  int c = n.key.compareTo(key);
  if(c < 0) {
      if(left ≠ null) left.insert(n);
      else { left = n; left.parent = this; }
  }
  else if(c > 0) {
      if(right ≠ null) right.insert(n);
      else { right = n; right.parent = this; }
  }
  else copy(n);
}
```

Binäre Suchbäume – Löschen

Beim Löschen unterscheiden wir drei Fälle:

- Blatt (einfach)
- Der Knoten hat kein linkes Kind (einfach)
- Der Knoten hat ein linkes Kind (schwierig)

Damit sind alle Fälle abgedeckt!

Warum kein Fall: Kein rechtes Kind?

Binäre Suchbäume – Löschen

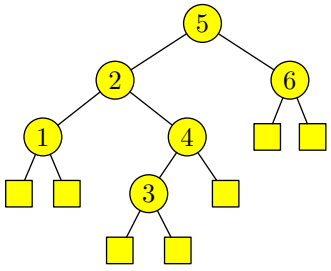
Beim Löschen unterscheiden wir drei Fälle:

- Blatt (einfach)
- Der Knoten hat kein linkes Kind (einfach)
- Der Knoten hat ein linkes Kind (schwierig)

Damit sind alle Fälle abgedeckt!
Warum kein Fall: Kein rechtes Kind?

Binäre Suchbäume – Löschen

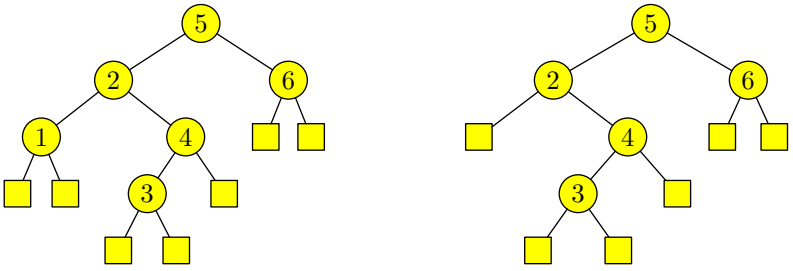
Löschen eines Blatts:



Ein Blatt kann durch Zeigerverbiegen gelöscht werden.

Binäre Suchbäume – Löschen

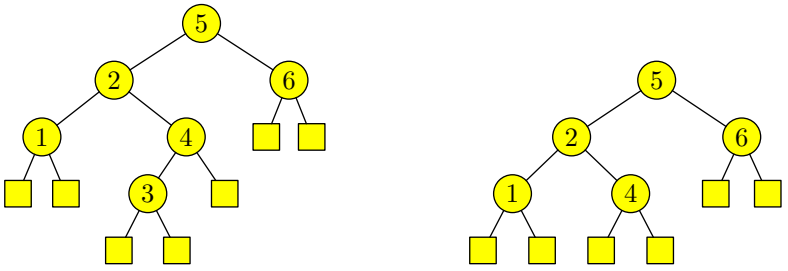
Löschen eines Blatts:



Ein Blatt kann durch Zeigerverbiegen gelöscht werden.

Binäre Suchbäume – Löschen

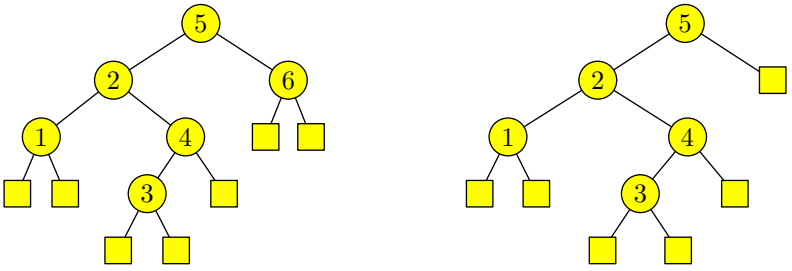
Löschen eines Blatts:



Ein Blatt kann durch Zeigerverbiegen gelöscht werden.

Binäre Suchbäume – Löschen

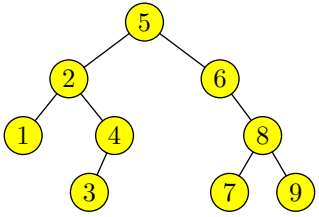
Löschen eines Blatts:



Ein Blatt kann durch Zeigerverbiegen gelöscht werden.

Binäre Suchbäume – Löschen

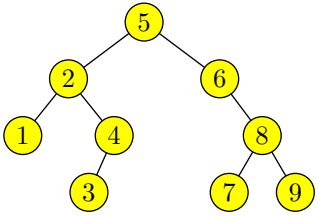
Löschen eines Knotens ohne linkes Kind:

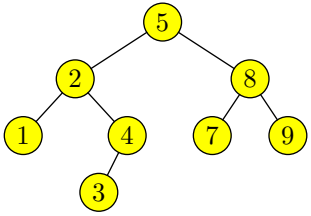


Wir können kopieren oder Zeiger verbiegen.

Binäre Suchbäume – Löschen

Löschen eines Knotens ohne linkes Kind:

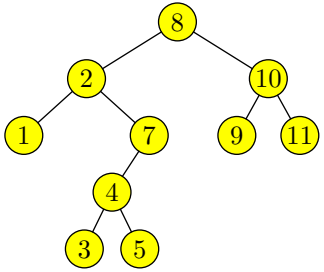




Wir können kopieren oder Zeiger verbiegen.

Binäre Suchbäume – Löschen

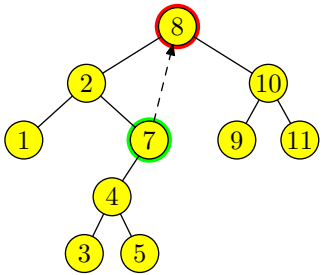
Löschen eines Knotens mit linkem Kind:



1. Finde den größten Knoten im linken Unterbaum
2. Kopiere seinen Inhalt
3. Lösche ihn

Binäre Suchbäume – Löschen

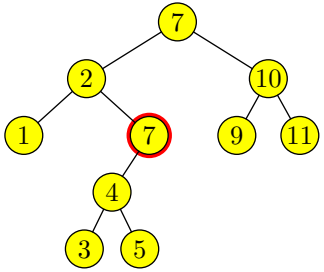
Löschen eines Knotens mit linkem Kind:



1. Finde den größten Knoten im linken Unterbaum
2. Kopiere seinen Inhalt
3. Lösche ihn

Binäre Suchbäume – Löschen

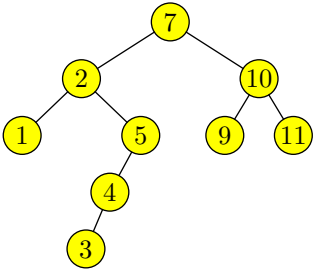
Löschen eines Knotens mit linkem Kind:



1. Finde den größten Knoten im linken Unterbaum
2. Kopiere seinen Inhalt
3. Lösche ihn

Binäre Suchbäume – Löschen

Löschen eines Knotens mit linkem Kind:



1. Finde den größten Knoten im linken Unterbaum
2. Kopiere seinen Inhalt
3. Lösche ihn

Binäre Suchbäume – Löschen

In der Klasse Searchtree⟨K, D⟩:

Java

```
public void delete(K k) {
  if(root == null) return;
  if(root.left == null && root.right == null && root.key == k)
    root = null;
  else {
    Searchtreenode⟨K, D⟩ n = root.findsubtree(k);
    if(n ≠ null) n.delete();
  }
}
```

Binäre Suchbäume – Löschen

Java

```
void delete() {
  if(left == null && right == null) {
    if(parent.left == this) parent.left = null;
    else parent.right = null; }
  else if(left == null) {
    if(parent.left == this) parent.left = right;
    else parent.right = right;
    right.parent = parent; }
  else {
    Searchtreenode<K, D> max = left;
    while(max.right ≠ null) max = max.right;
    copy(max); max.delete();
  }
}
```