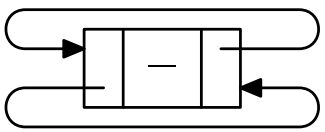
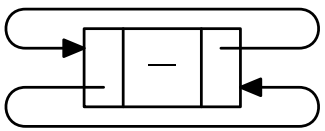
Der Konstruktor muß den Listenkopf head erzeugen.
Der Vorgänger und Nachfolger von head ist head selbst.



Der Konstruktor muß den Listenkopf head erzeugen.
Der Vorgänger und Nachfolger von head ist head selbst.



Java

```
public class ADList⟨K, D⟩ extends AbstractMap⟨K, D⟩ {
    Listnode⟨K, D⟩ head;
    public ADList() {
        head = new Listnode⟨K, D⟩(null, null);
        head.pred = head.succ = head;
    }
```

Java

```
class Listnode⟨K, D⟩ {
    K key;
    D data;
    Listnode⟨K, D⟩ pred, succ;
    Listnode(K k, D d) {
        key = k; data = d; pred = null; succ = null; }
    void delete() {
        pred.succ = succ; succ.pred = pred; }
    void copy(Listnode⟨K, D⟩ n) {
        key = n.key; data = n.data; }
    void append(Listnode⟨K, D⟩ newnode) {
        newnode.succ = succ; newnode.pred = this;
        succ.pred = newnode; succ = newnode; }
}
```

Java

```
public void append(K k, D d) {
    head.pred.append(new Listnode⟨K, D⟩(k, d));
}
```

Java

```
public void prepend(K k, D d) {
    head.append(new Listnode⟨K, D⟩(k, d));
}
```

Java

```
public void delete(K k) {
    Listnode⟨K, D⟩ n = findnode(k);
    if(n ≠ null) n.delete();
}
```

Java

```
public D find(K k) {
    Listnode⟨K, D⟩ n = findnode(k);
    if(n == null) return null;
    return n.data;
}
```

Java

```
protected Listnode⟨K, D⟩ findnode(K k) {
    Listnode⟨K, D⟩ n;
    head.key = k;
    for(n = head.succ;  !n.key.equals(k);  n = n.succ);
    head.key = null;
    if(n == head) return null;
    return n;
}
```

Einfach verkettete Listen

- Kein Zeiger auf Vorgänger
- Einfachere Datenstruktur
- Operationen können komplizierter sein

Frage: Wie kann ein Element *vor* einen gegebenen Knoten eingefügt werden?

Ersetzen und alten Knoten anfügen!

Vorsicht: Eine gefährliche Technik.

Einfach verkettete Listen

- Kein Zeiger auf Vorgänger
- Einfachere Datenstruktur
- Operationen können komplizierter sein

Frage: Wie kann ein Element *vor* einen gegebenen Knoten eingefügt werden?

Ersetzen und alten Knoten anfügen!

Vorsicht: Eine gefährliche Technik.

Listen – Laufzeit

Manche Operation sind schnell, manche langsam...

Operation	Laufzeit
append()	$\Theta(1)$
delete() *	$\Theta(1)$
delete()	$\Theta(n)$
find()	$\Theta(n)$
insert()	$\Theta(n)$

* direktes Löschen eines Knotens, nicht über Schlüssel

Java

```
public class Stack⟨D⟩ {
  private ADList⟨Object, D⟩ stack;
  private int size;
  public Stack() { stack = new ADList⟨Object, D⟩(); size = 0; }
  public boolean isempty() { return size == 0; }
  public D pop() {
    D x = stack.firstnode().data;
    stack.firstnode().delete();
    size−−;
    return x;
  }
  public void push(D x) { stack.prepend(null, x); size++; }
  public int size() { return size; }
}
```

Java

```
public class Queue⟨D⟩ {
  private ADList⟨Object, D⟩ queue;
  private int size;
  public Queue() { queue = new ADList⟨Object, D⟩(); size = 0; }
  public D dequeue() {
    D x = queue.lastnode().data;
    queue.lastnode().delete();
    size−−;
    return x;
  }
  public void enqueue(D x) { queue.prepend(null, x); size++; }
  public int size() { return size; }
}
```

Übersicht

Übersicht

Lineare Suche

Wir suchen x im Array $a[0, \ldots, n-1]$

Algorithmus

```
function find1(int x) boolean :
for i = 0 to n − 1 do
  if x = a[i] then return true fi
od;
return false
```

Die innere Schleife ist langsam.

Lineare Suche – verbessert

Wir verwenden ein *Sentinel*-Element am Ende.

Das Element $a[n]$ muß existieren und unbenutzt sein.

Algorithmus

```
function find2(int x) boolean :
i := 0;
a[n] := x;
while a[i] ≠ x do i := i + 1 od;
return i < n
```

Die innere Schleife ist sehr schnell.

Lineare Suche – ohne zusätzlichen Platz

Algorithmus

```
function find3(int x) boolean :
if x = a[n − 1] then return true fi;
temp := a[n − 1];
a[n − 1] := x;
i := 0;
while a[i] ≠ x do i := i + 1 od;
a[n − 1] := temp;
return i < n − 1
```

Erheblicher Zusatzaufwand.

Innere Schleife immer noch sehr schnell.

Lineare Suche – Analyse

Wieviele Vergleiche benötigt eine erfolgreiche Suche nach x im Mittel?

Wir nehmen an, daß jedes der n Elemente mit gleicher Wahrscheinlichkeit gesucht wird.

Es gilt $\Pr[x = a[i]] = 1/n$ für $i \in \{0, \ldots, n-1\}$.

Sei C die Anzahl der Vergleiche:
$C = i + 1$ gdw. $x = a[i]$

Lineare Suche – Analyse

Wieviele Vergleiche benötigt eine erfolgreiche Suche nach x im Mittel?

Wir nehmen an, daß jedes der n Elemente mit gleicher Wahrscheinlichkeit gesucht wird.

Es gilt $\Pr[x = a[i]] = 1/n$ für $i \in \{0, \ldots, n-1\}$.

Sei C die Anzahl der Vergleiche:
$C = i + 1$ gdw. $x = a[i]$

Lineare Suche – Analyse

Wir suchen den Erwartungswert von C.

$$\Pr[C = i] = 1/n \text{ für } i = 1, \ldots, n$$

$$E(C) = \sum_{k=1}^{n} k \Pr[C = k] = \sum_{k=1}^{n} \frac{k}{n} = \frac{n+1}{2}$$

Es sind im Mittel $(n+1)/2$ Vergleiche.

Lineare Suche – Analyse

Wir suchen den Erwartungswert von C.

$$\Pr[C = i] = 1/n \text{ für } i = 1, \ldots, n$$

$$E(C) = \sum_{k=1}^{n} k \Pr[C = k] = \sum_{k=1}^{n} \frac{k}{n} = \frac{n+1}{2}$$

Es sind im Mittel $(n+1)/2$ Vergleiche.

Lineare Suche – Analyse

Unser Ergebnis:

Es sind im Mittel $(n+1)/2$ Vergleiche.

Ist das Ergebnis richtig?

Wir überprüfen das Ergebnis für kleine n.

- $n = 1$?
- $n = 2$?

Lineare Suche – Analyse

Unser Ergebnis:

Es sind im Mittel $(n+1)/2$ Vergleiche.

Ist das Ergebnis richtig?

Wir überprüfen das Ergebnis für kleine n.

- $n = 1$?
- $n = 2$?

Binäre Suche

Wir suchen wieder x in $a[0, \ldots, n-1]$.

Algorithmus

function binsearch(**int** x) **boolean** :
$l := 0;\ r := n - 1;$
while $l \leq r$ **do**
  $m := \lfloor (l + r)/2 \rfloor;$
  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

Wir halbieren den Suchraum in jedem Durchlauf.

Binäre Suche

Java

```
public boolean binsearch(D d) {
  int l = 0, r = size − 1, m, c;
  while(l ≤ r) {
    m = (l + r)/2;
    c = d.compareTo(get(m));
    if(c == 0) return true;
    if(c < 0) r = m − 1;
    else l = m + 1;
  }
  return false;
}
```

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:

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

$n > 1$:

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

Noch zu zeigen:

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

$\rightarrow$ Übungsaufgabe.

Binäre Suche – Analyse

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

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

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

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

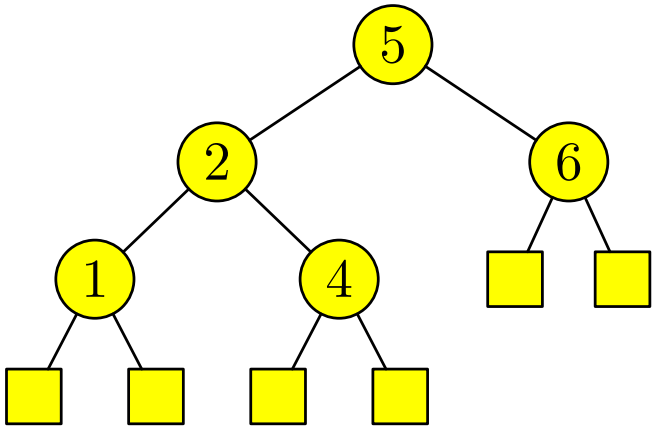
$$\lfloor \log n \rfloor + 1 = \log(n) + O(1)$$

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

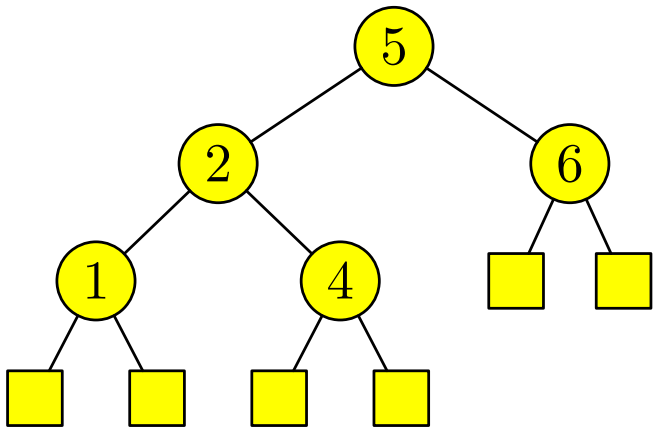
Java

```
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

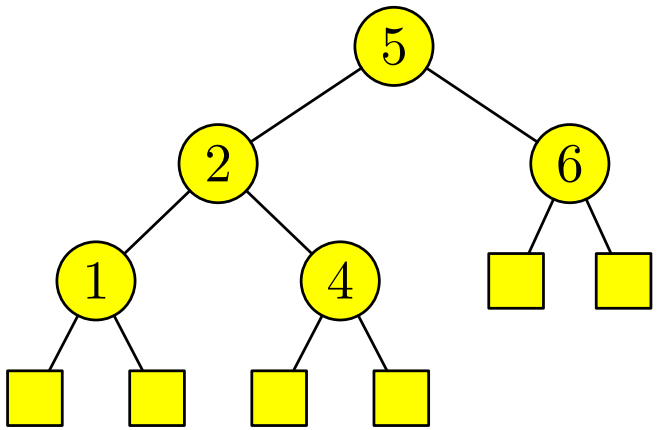


Wo fügen wir 3 ein?

Wie fügen wir es ein?

In den richtigen externen Knoten!

Binäre Suchbäume – Einfügen

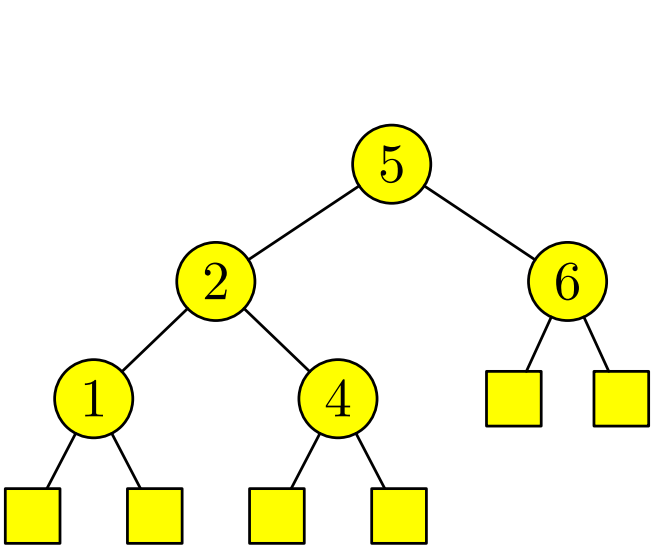


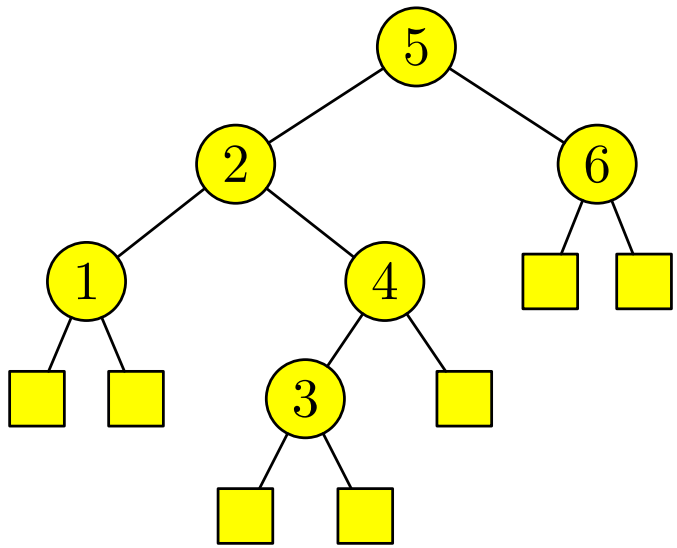
Wo fügen wir 3 ein?

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

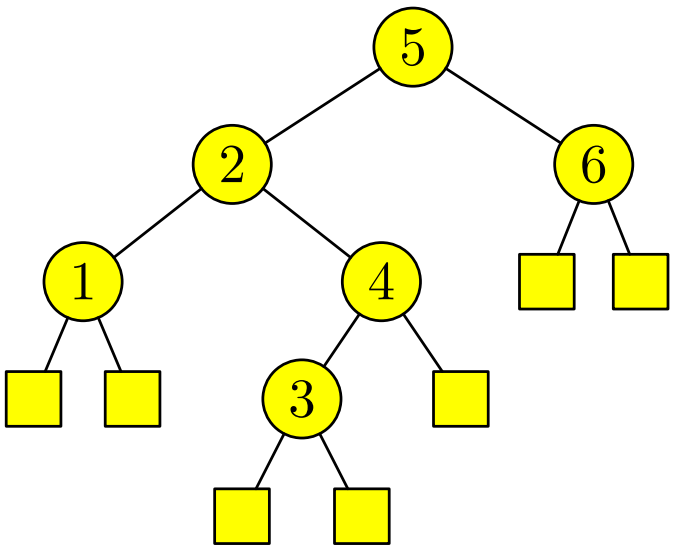
Beim Löschen unterscheiden wir drei Fälle:

- 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

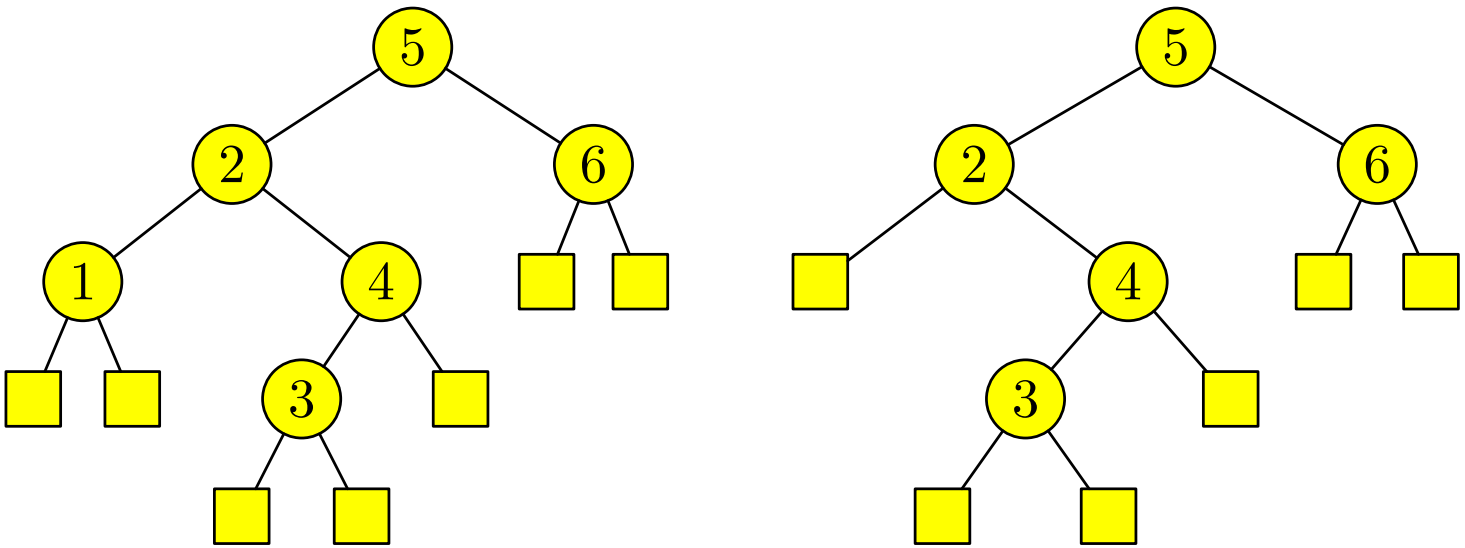
Löschen eines Blatts:



Ein Blatt kann durch Zeigerverbiegen gelöscht werden.

Binäre Suchbäume – Löschen

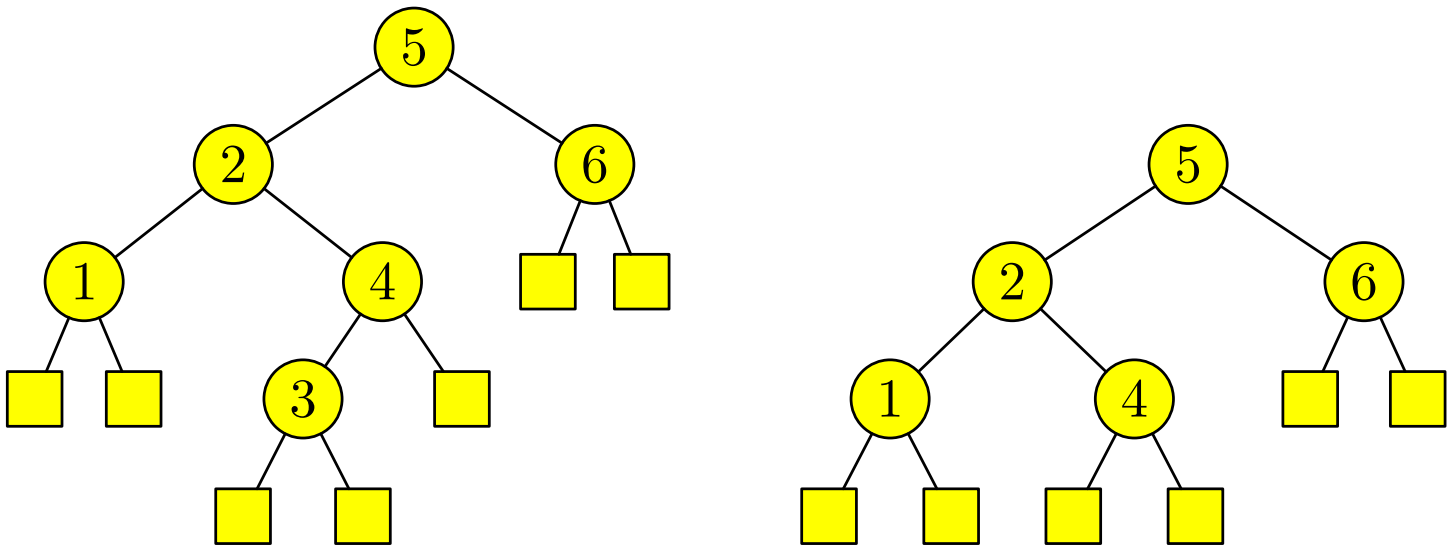
Löschen eines Blatts:



Ein Blatt kann durch Zeigerverbiegen gelöscht werden.

Binäre Suchbäume – Löschen

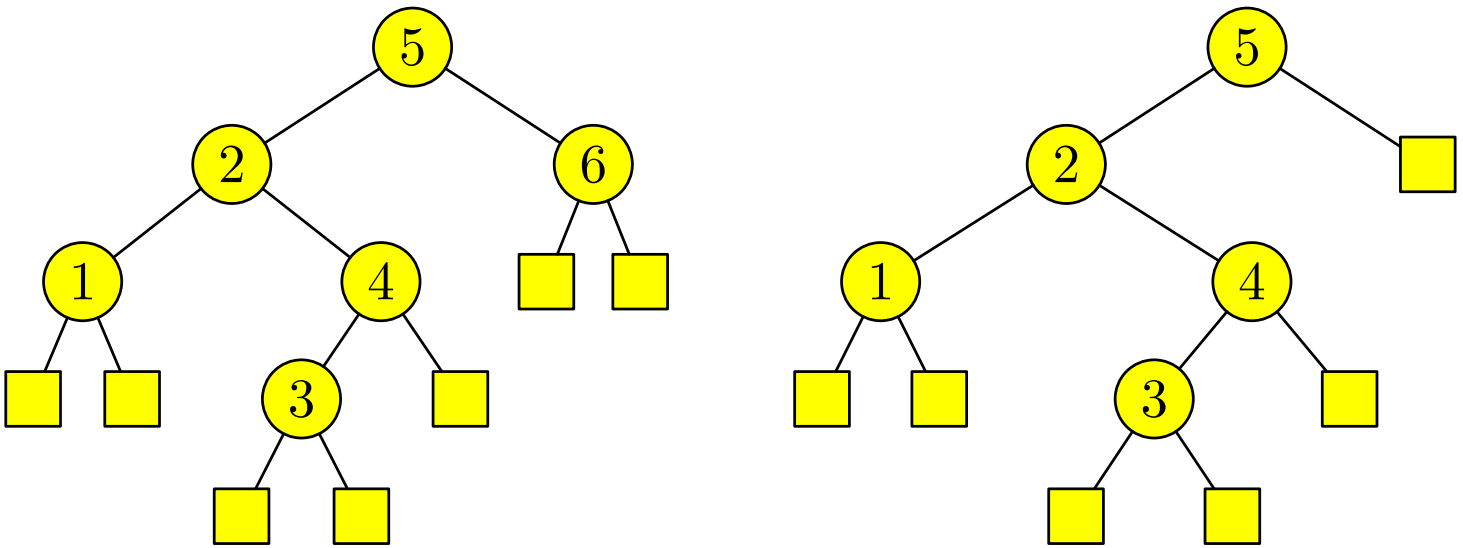
Löschen eines Blatts:



Ein Blatt kann durch Zeigerverbiegen gelöscht werden.

Binäre Suchbäume – Löschen

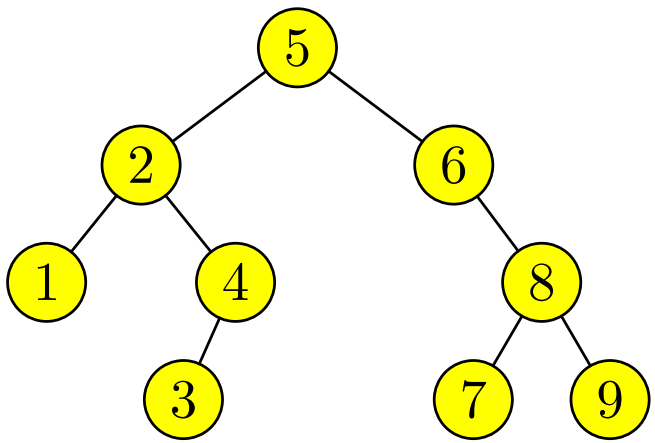
Löschen eines Blatts:



Ein Blatt kann durch Zeigerverbiegen gelöscht werden.

Binäre Suchbäume – Löschen

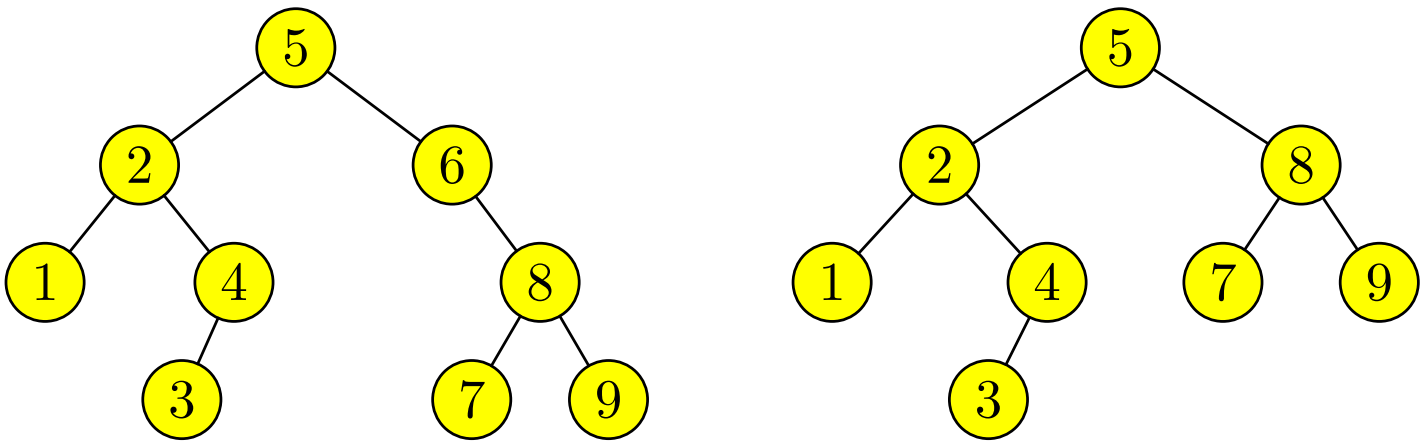
Löschen eines Knotens ohne linkes Kind:



Wir können kopieren oder Zeiger verbiegen.

Binäre Suchbäume – Löschen

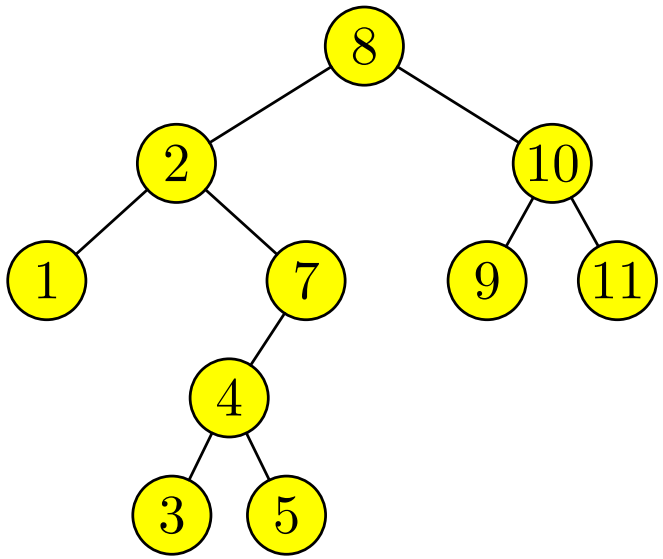
Löschen eines Knotens ohne linkes Kind:



Wir können kopieren oder Zeiger verbiegen.

Binäre Suchbäume – Löschen

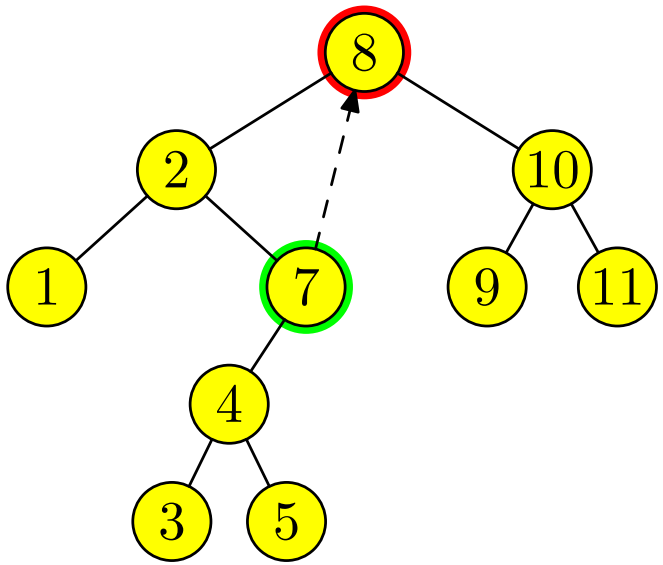
Löschen eines Knotens mit linkem Kind:



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

Binäre Suchbäume – Löschen

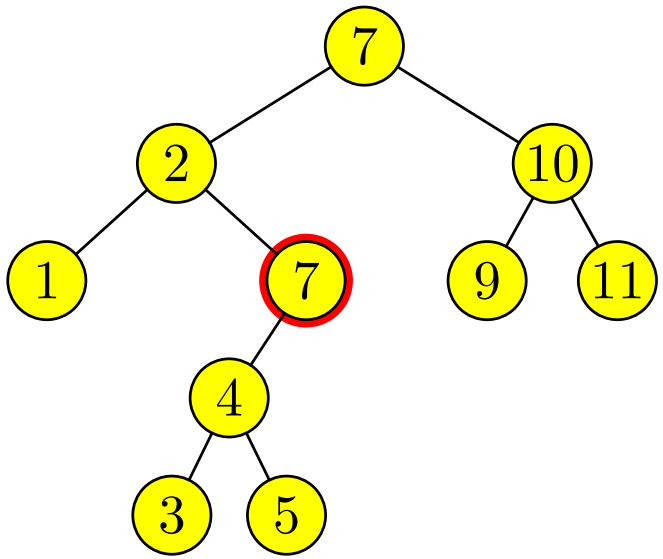
Löschen eines Knotens mit linkem Kind:



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

Binäre Suchbäume – Löschen

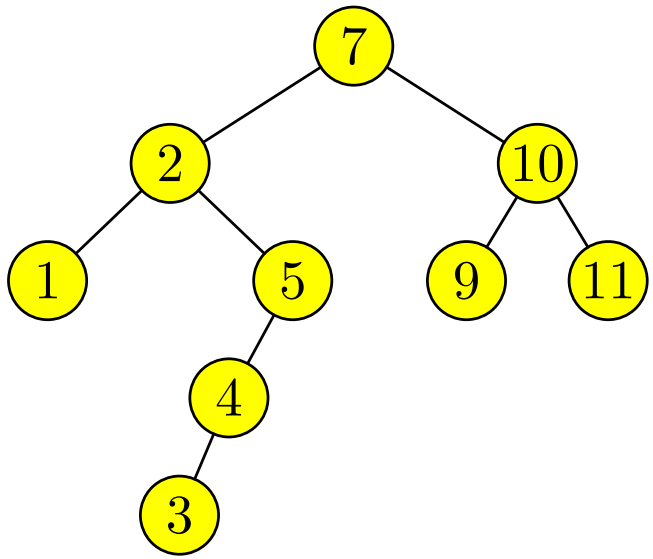
Löschen eines Knotens mit linkem Kind:



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