

Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von
Algorithmen (IEMS)
WS 2014 / 2015

Vorlesung 10, Donnerstag 8. Januar 2015
(Verkettete Listen, Binäre Suchbäume)

Junior-Prof. Dr. Olaf Ronneberger
Image Analysis Lab
Institut für Informatik
Universität Freiburg

Blick über die Vorlesung heute

- Verkettete Listen
- Binäre Suchbäume (binary search trees)
 - **Übungsblatt 10:** Einfache Implementierung eines Binären Suchbaums (nur **insert** und **lookup**, kein **remove**) + ein paar einfache Laufzeittests

Rückmeldungen zu Ü9 (Mastertheorem)

- War einfach und schnell erledigt. (Zeitaufwand bei fast allen deutlich unter 4 Stunden)
- Wie ausführlich sollen die Lösungen sein? → es reichen ein paar Zeilen (siehe Musterlösung)

Motivation: Sortierte Folgen

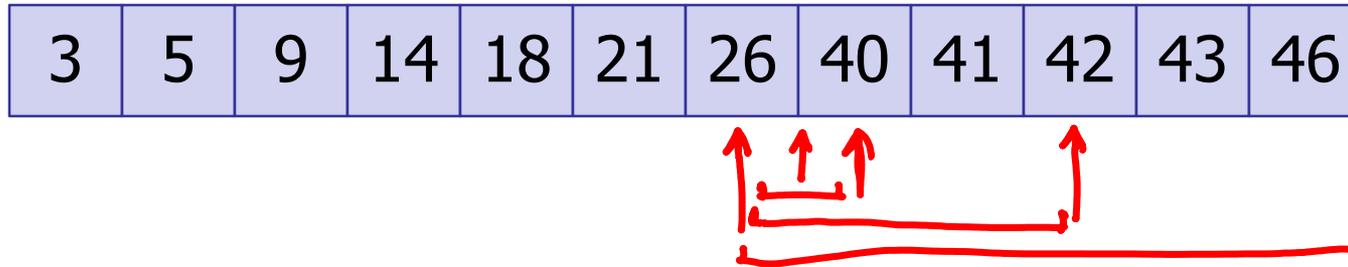
■ Problem

- Wir wollen wieder $(key, value)$ Paare / Elemente verwalten
- Wir haben wieder eine Ordnung $<$ auf den Keys
- Diesmal wollen wir folgende Operationen unterstützen
 - $insert(key, value)$: füge das gegebene Paar ein
 - $remove(key)$: entferne das Paar mit dem gegebenen Key
 - $lookup(key)$: finde das Element mit dem gegebenen Key; falls es das nicht gibt, finde das Element mit dem kleinsten Key der $> key$ ist
 - $next / previous$: für ein gegebenes Element, finde das mit dem nächstgrößeren / nächstkleineren Schlüssel; damit lässt sich insbesondere über alle Elemente iterieren

Wo braucht man das?

- Typisches Anwendungsbeispiel: Datenbanken
 - Eine große Menge von Records
 - Zum Beispiele Bücher, Produkte, Wohnungen, ...
 - Typische Suchanfrage: alle Wohnungen zwischen 400 und 600 Euro Monatsmiete
 - Ein sogenannter **range query**
 - Das bekommt man mit **lookup** und **next**
 - Man beachte: es ist dafür nicht wichtig, dass es eine Wohnung gibt, die **genau 400 Euro** kostet
 - Wenn man ein paar records hinzufügt oder alte löscht, will man nicht jedes Mal erst alles wieder neu sortieren

Lösung 1 (nicht gut): Einfache Arrays



- Mit einem einfachen sortierten Array bekommen wir
 - `lookup` in Zeit $O(\log n)$
 - das geht mit **binärer Suche**,
 - z.B. `lookup(41)`
 - `next` und `previous` in Zeit $O(1)$
 - klar, sie stehen ja direkt nebeneinander
 - `insert` und `remove` in Zeit bis zu $\Theta(n)$
 - bis zu $\Theta(n)$ Elemente müssen umkopiert werden

Lösung 2 (schlecht): Hashtabellen

- Mit einer Hashtabelle bekommt man
 - **insert** und **remove** in erwarteter Zeit $O(1)$
 - bei genügend großer Hashtabelle und guter Hashfunktion
 - **lookup** in erwarteter Zeit $O(1)$
 - aber nur wenn es ein Element mit **exakt** dem gegebenen Key gibt, sonst bekommt man gar nichts
 - **next** und **previous** in Zeit bis zu $\Theta(n)$
 - die Reihenfolge, in der die Elemente in einer Hashtabelle stehen hat nichts mit der Reihenfolge der Keys zu tun!

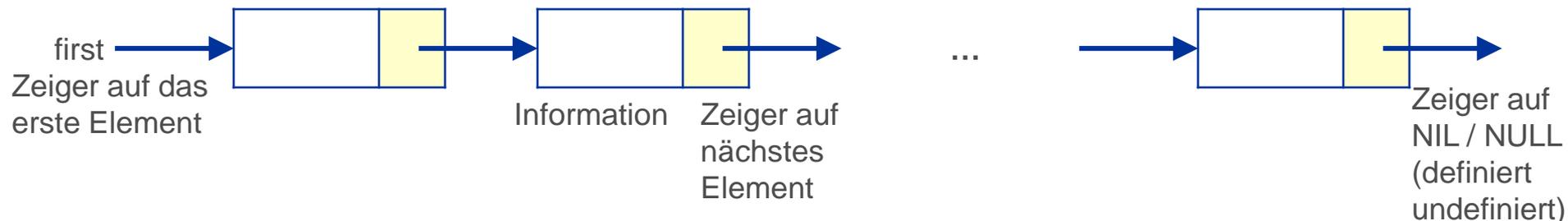
Lösung 3 (gut?) Verkettete Liste

- Mit einer doppelt verketteten Liste bekommt man
 - `next` und `previous` in Zeit $O(1)$
 - `insert` und `remove` in Zeit $O(1)$
 - `lookup` in Zeit bis zu $\Theta(n)$
- Auch noch nicht das, was wir wollten, aber von der Struktur her ähnlich, wie die binären Suchbäume
- Die schauen wir und jetzt mal genauer an

Verkettete Liste



- dynamische Datenstruktur
- Zahl der Elemente während der Laufzeit frei wählbar
- Elemente bestehen aus einfachen oder zusammengesetzten Datentypen
- Elemente sind durch Zeiger / Referenzen auf ein folgendes Element verbunden
- einfache oder doppelte Verkettung



Eigenschaften im Vergleich zum Feld



- geringer Mehrbedarf an Speicher
- Einfügen und Löschen von Elementen erfolgt ohne Umkopieren anderer Elemente
- Zahl der Elemente kann beliebig verändert werden
- kein direkter Zugriff auf Elemente
(Liste muss durchlaufen werden)

Varianten

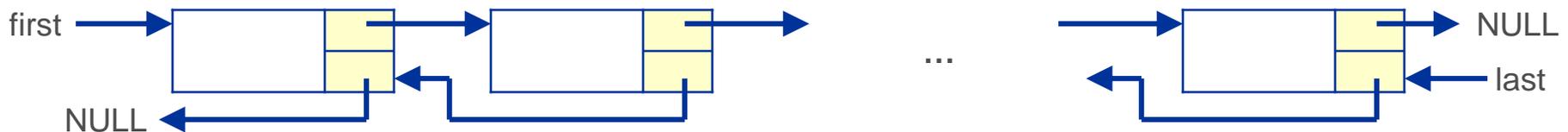


■ Liste mit Listenkopf und Ende-Zeiger



- enthält Zeiger auf erstes Element
- kann weitere Informationen enthalten (z. B. Zahl der Elemente)

■ doppelt verkettete Liste (ein Zeiger zum folgenden Element, ein zweiter Zeiger zum vorherigen Element)



Element / Knoten

Implementierung in Java



basiert auf Mary K. Vernon:
"Introduction to Data Structures"
<http://pages.cs.wisc.edu/~vernon/cs367/cs367.html>

```
public class Listnode
{
    private int data;
    private Listnode next;

    public Listnode(int d)
    { data = d; next = null; }

    public Listnode(int d, Listnode n)
    { data = d; next = n; }

    public int getData() { return data; }
    public void setData(int d) { data = d; }

    public Listnode getNext() { return next; }
    public void setNext(Listnode n) { next = n; }
}
```

2 Felder: Daten (hier lediglich ein Integer) und ein Zeiger / eine Referenz auf Listnode
private: nur innerhalb der Klasse zugreifbar

2 Konstruktoren: Initialisierung von Instanzen der Klasse

Funktionen zum Lesen und Schreiben der private-Felder.
Kapselung: Erlaubt die Einhaltung von Zusicherungen an den Inhalt der Felder.

Die Daten eines Knoten sind hier durch "int" vereinfacht repräsentiert. Üblicherweise verwendet man hier selbst definierte Referenzdatentypen, z. B. Object data; . In dem Fall reservieren die Konstruktoren lediglich Speicher für Zeiger / Referenzen auf den Datentyp Object.

Element / Knoten

Implementierung in C++



```
class Listnode
{
private:
    int data;
    Listnode* next;           Pointer statt Referenzen

public:
    Listnode(int d)
    { data = d; next = NULL; }

    Listnode(int d, Listnode* n)
    { data = d; next = n; }

    int getData() { return data; }
    void setData(int d) { data = d; }

    Listnode* getNext() { return next; }
    void setNext(Listnode* n) { next = n; }
}
```

Beispiele Java



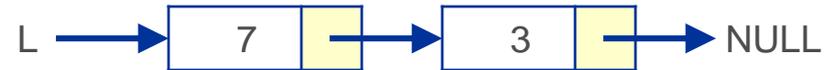
- `Listnode L;`



- `L = new Listnode(7);`



- `L.setNext(new Listnode(3));`



- `L.getNext().setData(4);`



Beispiele C++



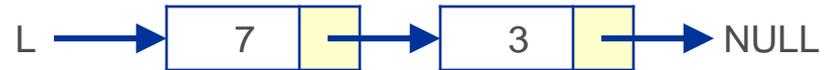
- `Listnode* L;`



- `L = new Listnode(7);`



- `L->setNext(new Listnode(3));`



- `L->getNext()->setData(4);`

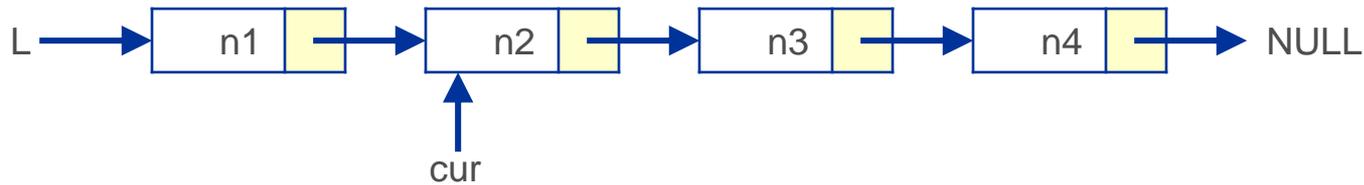


Einfügen eines Knotens

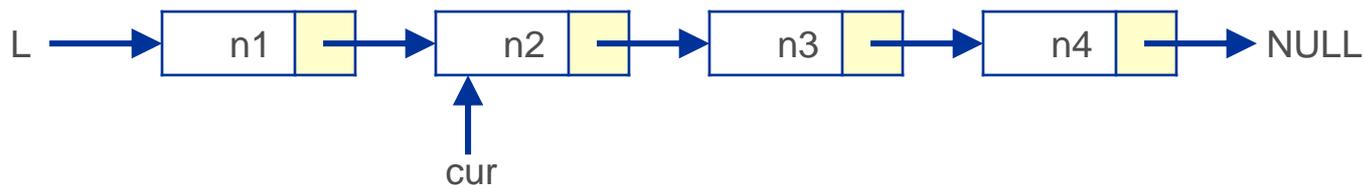
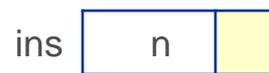
(ab hier nur noch in Java)



- Einfügen eines Knotens ins hinter einem Knoten cur



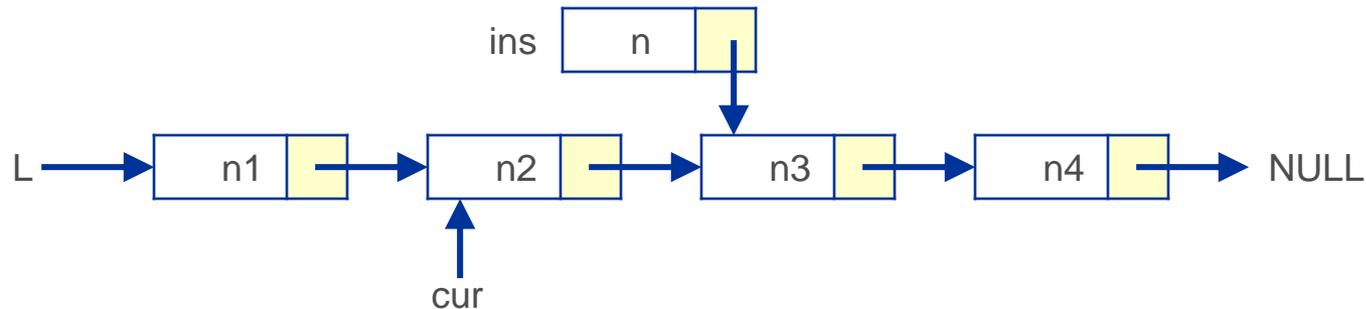
```
Listnode ins = new Listnode(n);
```



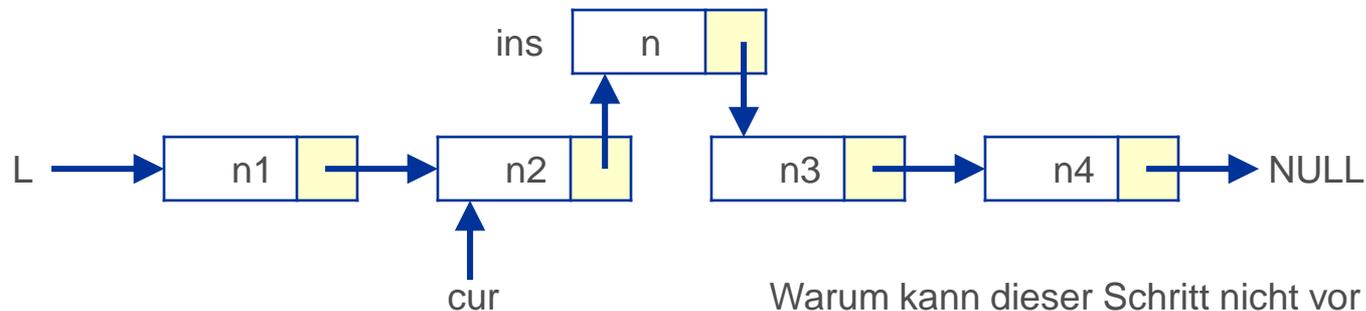
Einfügen eines Knotens



- `ins.setNext(cur.getNext());`



- `cur.setNext(ins);`



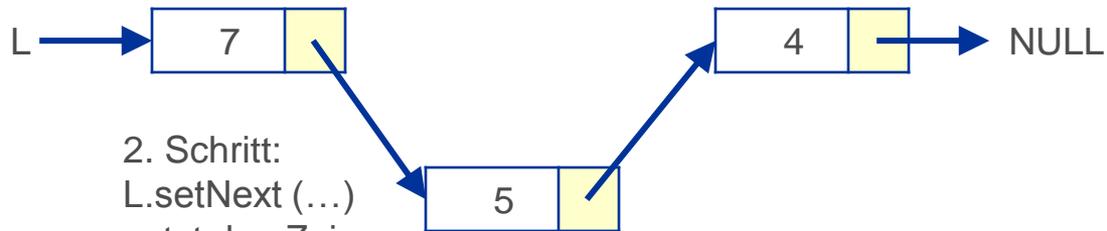
Warum kann dieser Schritt nicht vor `insert.setNext` durchgeführt werden?

Beispiele (in einer Zeile Code)



- Einfügen zwischen dem ersten und dem zweiten Element

```
L.setNext( new Listnode(5, L.getNext()) );
```



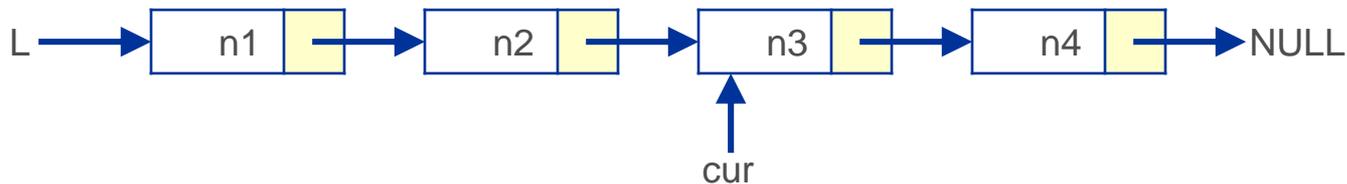
2. Schritt:
L.setNext (...)
setzt den Zeiger
auf das neue
Element

1. Schritt:
new Listnode (5, L.getNext ()) erzeugt
den Knoten und setzt den Zeiger auf das
nachfolgende Element von 7, also 4.

Löschen eines Knotens



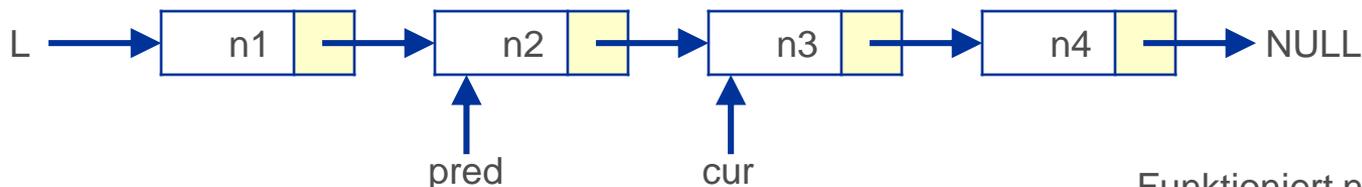
- Löschen eines Knotens cur



- Finde den Vorgängerknoten pred

```
Listnode pred = L;  
while (pred.getNext() != cur)  
    pred = pred.getNext();
```

Laufe elementweise durch die
Liste, bis pred.next auf cur zeigt.
O (n)



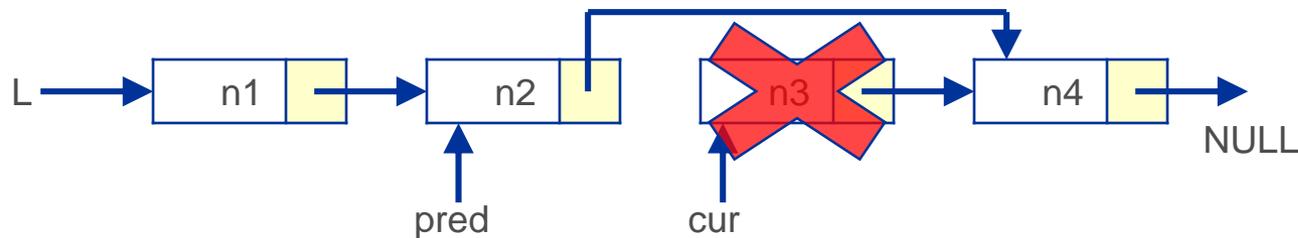
Funktioniert nicht für
den ersten Knoten!

Löschen eines Knotens



- setze den next-Zeiger des Vorgängers von cur auf den Nachfolger von cur

```
pred.setNext( cur.getNext() );
```

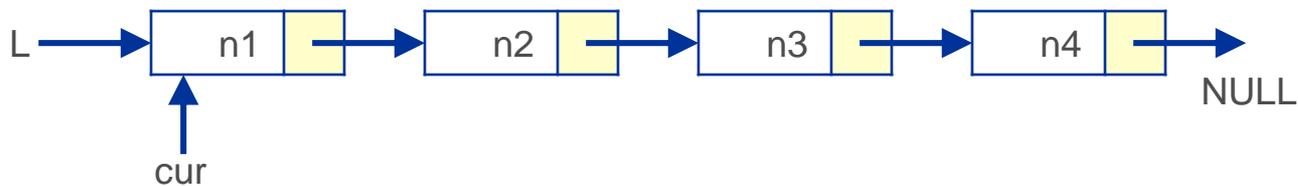


- Wenn keine weitere Referenz auf die durch cur angegebene Instanz existiert und cur auf eine andere Instanz gesetzt wird, ist das gelöschte Element nicht mehr zugreifbar. In C++ muss der belegte Speicher durch delete als Gegenstück zu new freigegeben werden.

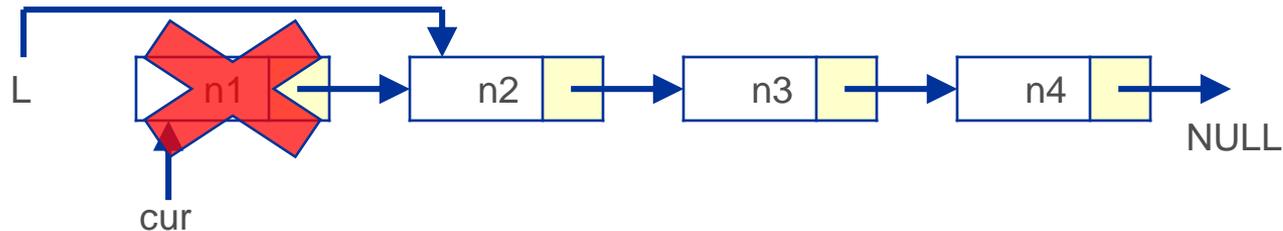
Sonderfall



- Löschen des ersten Knotens



- `if (cur == L) { L = cur.getNext(); }`



Löschen eines Knotens



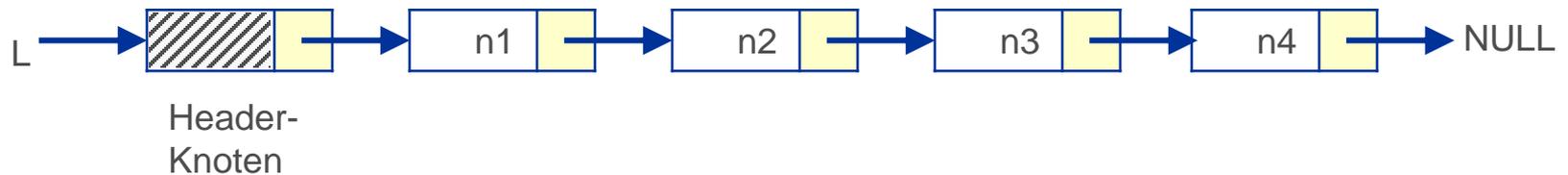
- generelles Löschen

```
if (cur == L)
{
    L = cur.getNext();
}
else
{
    Listnode pred = L;
    while (pred.getNext() != cur)
        pred = pred.getNext();
    pred.setNext(cur.getNext());
}
```

Vermeidung des Sonderfalls



■ Header-Knoten



■ Vorteil

- Löschen des ersten Knotens ist kein Sonderfall

■ Nachteil

- weitere Funktionen müssen berücksichtigen, dass der erste Knoten nicht zur Liste gehört, z. B. Ausgabe aller Elemente, Zählen der Elemente

Verkettete Liste (mit Header-Knoten)



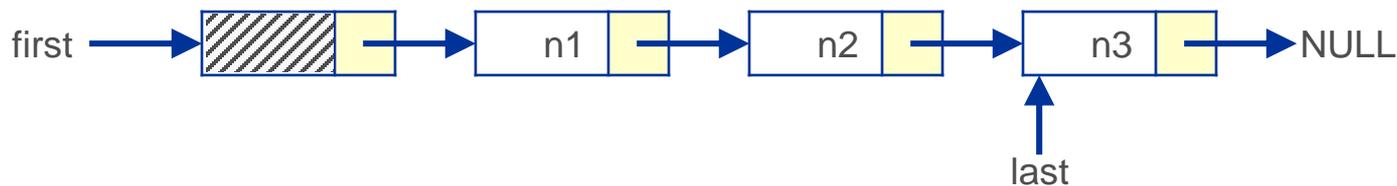
```
public class List
{
    private Listnode first, last;
    private int numItems;

    public List() { ... }

    public int size() { return numItems; }
    public boolean isEmpty() { return first==last; }
    isEmpty ist nur korrekt, wenn ein
    Header-Knoten vorhanden ist.

    public void add (int data) { ... }
    public void insertAfter (Listnode cur, int data) { ... }
    public void remove (Listnode cur) { ... }
    public Listnode get (int pos) { ... }
    public boolean contains (int data) { ... }
}
```

first, last

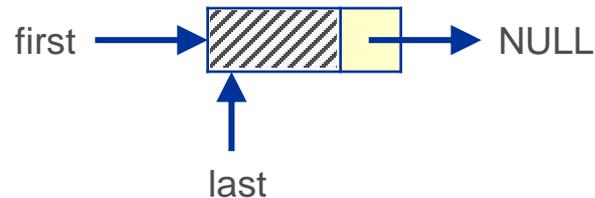


- First zeigt auf das erste Element, last auf das letzte.
- Last erlaubt Einfügen von Elementen am Ende in $O(1)$, Liste muss nicht durchlaufen werden.
- Last muss durch alle Operationen aktuell gehalten werden.

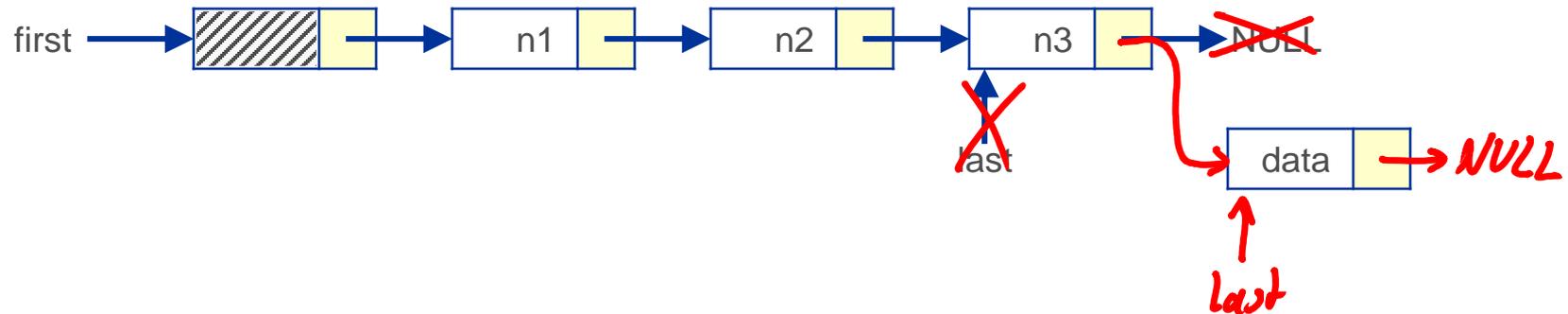
Konstruktor (mit Header-Knoten)



```
■ public List ()  
  {  
    first = last = new Listnode (null);  
    numItems = 0;  
  }
```



Einfügen am Ende der Liste

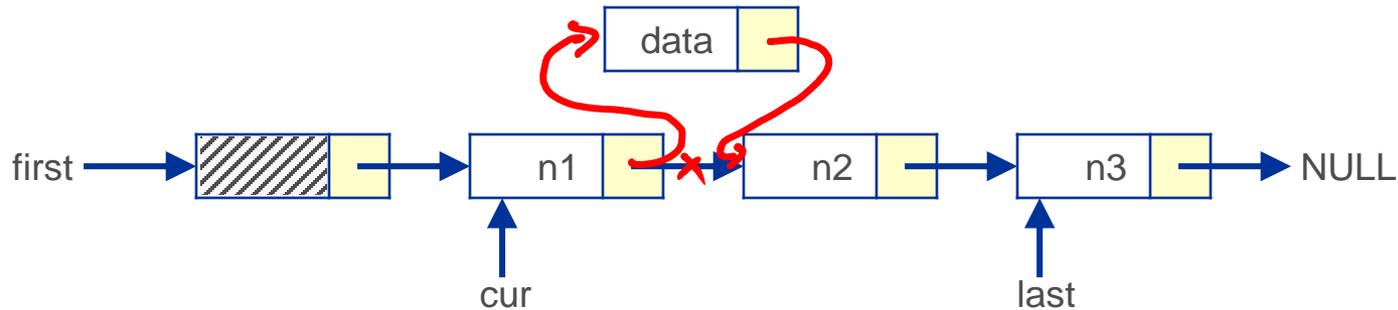


- ```
public void add (int data)
{
 last.setNext(new Listnode(data));
 last = last.getNext();
 numItems++;
}
```

Durchlaufen der Liste wird durch last-Zeiger vermieden.

last-Zeiger und Zahl der Elemente numItems müssen aktualisiert werden.

# Einfügen hinter cur

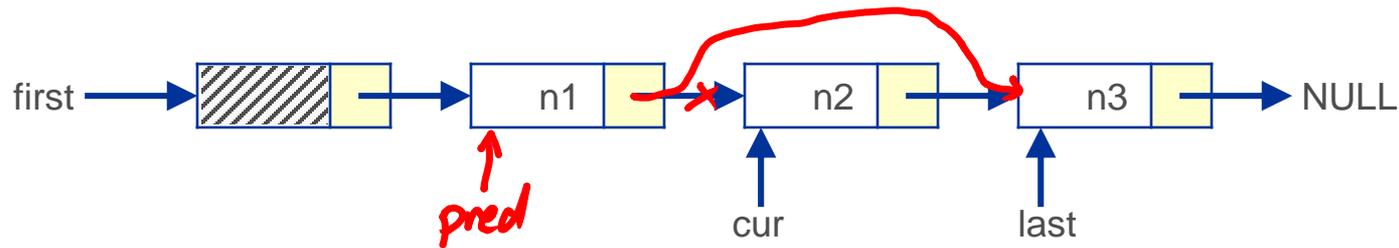


- ```
public void insertAfter(Listnode cur, int data)
{
    if (cur == last) add(data);           Füge am Ende ein.
    else
    {
        cur.setNext(new Listnode(data, cur.getNext()));
        numItems++;
    }
}
```

first wird nicht verändert,
da first auf den Header-
Knoten zeigt.

last wird nicht verändert,
da nicht am Ende
eingefügt wird.

Löschen von Element cur



```
■ public void remove (Listnode cur)
```

```
{
```

```
    Listnode pred = first;
```

```
    while (pred.getNext() != cur)
```

```
        pred = pred.getNext();
```

Finde den Vorgänger des Elementes cur: $O(n)$

```
    pred.setNext(cur.getNext());
```

```
    numItems--;
```

```
    if (pred.getNext() == null) last = pred;
```

```
    (in C++ hier löschen des Elements cur)
```

```
}
```

Referenz auf Element an Position pos



```
■ public Listnode get(int pos)
{
    if (pos < 1 || pos > numItems)
        return null;

    Listnode cur = first;
    for (int k=0; k<pos; k++)
        cur = cur.getNext();

    return cur;
}
```

Laufe durch die Liste
bis Position pos.

Suchen eines Elements



```
■ public boolean contains(int data)
{
    Listnode cur = first;

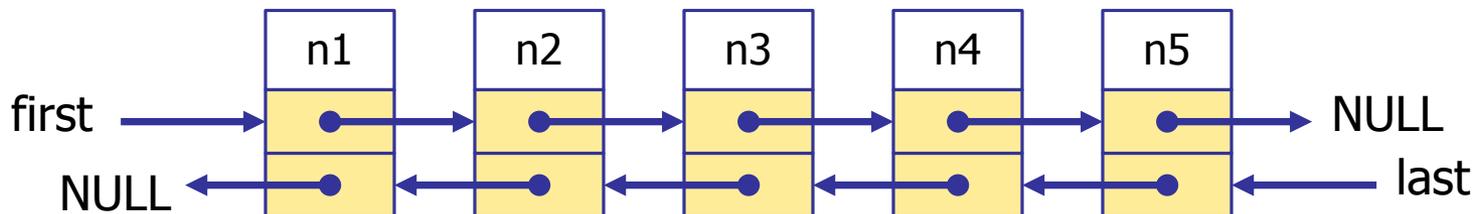
    for (int k=0; k<numItems; k++)
    {
        cur = cur.getNext();
        if (cur.getData() == data) return true;
    }

    return false;
}
```

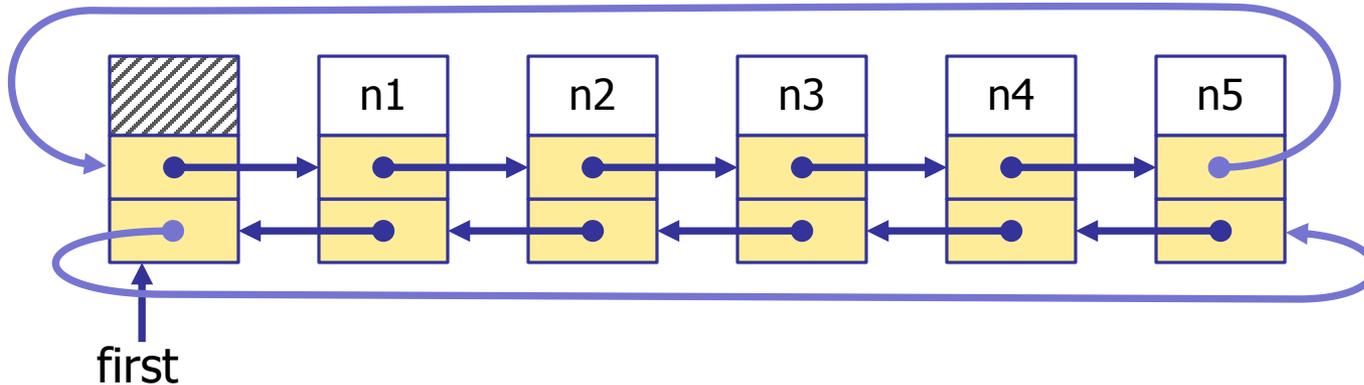
Erstes Element ist
der Header, kein
Element unserer
Liste.

Laufzeiten einfach verkettete Liste

- next in Zeit $O(1)$
- previous in Zeit bis zu $\Theta(n)$
- insert in Zeit $O(1)$
- remove in Zeit bis zu $\Theta(n)$
- lookup in Zeit bis zu $\Theta(n)$
- Das geht besser mit doppelt verketteten Listen
 - Jedes Element hat einen Zeiger auf den Nachfolger **und Vorgänger**



Doppelt verkettete Liste mit Header-Element



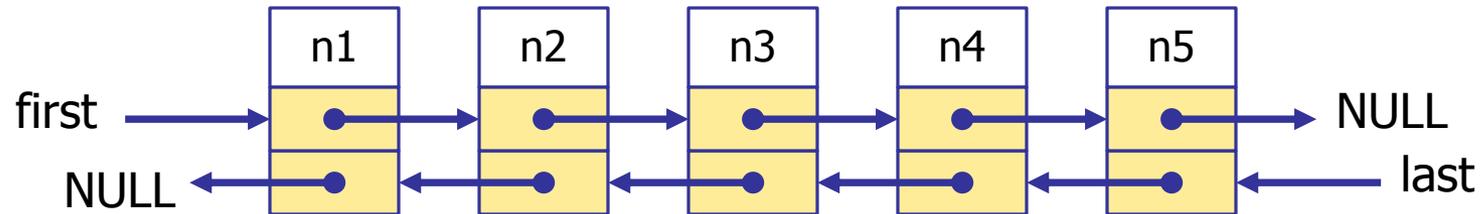
- Auch hier ist es geschickt, ein Header-Element anzulegen, um Sonderfälle zu vermeiden
- Durch zyklische Verbindung reicht ein Header-Element aus

Laufzeit doppelt verkettete Liste

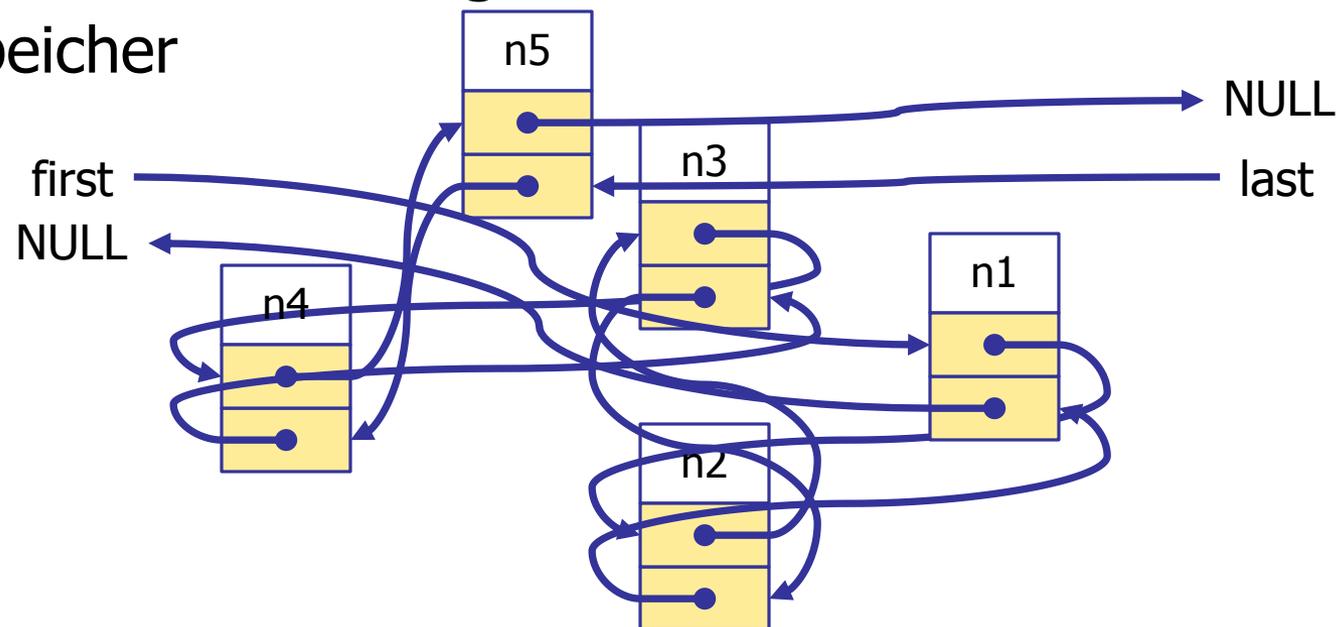
- Mit einer doppelt verketteten Liste bekommt man
 - `next` und `previous` in Zeit $O(1)$
 - jedes Element hat einen Zeiger zum Vorgänger / Nachfolger
 - `insert` und `remove` in Zeit $O(1)$
 - es müssen nur konstant viele Zeiger umgesetzt werden
 - `lookup` in Zeit bis zu $\Theta(n)$
 - Selbst wenn die Elemente in der Liste sortiert wären, könnte man sie nur in Zeit $\Theta(n)$ finden – **warum?**

Listen im realen Programm

Liste im Lehrbuch:



Liste im realen Programm: Die Elemente stehen irgendwo im Speicher

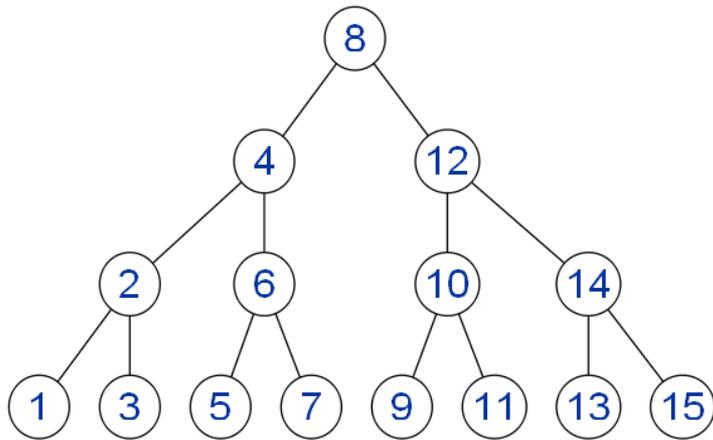


Lösung 4 (gut): Suchbäume

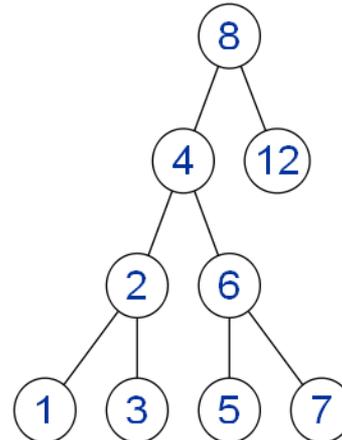
- Mit einem geeigneten Suchbaum bekommt man
 - `next` und `previous` in Zeit $O(1)$
 - entsprechende Zeiger wie bei der verketteten Liste
 - `insert` und `remove` in Zeit $O(\log n)$
 - Sehen wir gleich, warum
 - `lookup` in Zeit $O(\log n)$
 - eine Baumstruktur hilft jetzt beim effizienten Suchen

Binäre Suchbäume — Idee

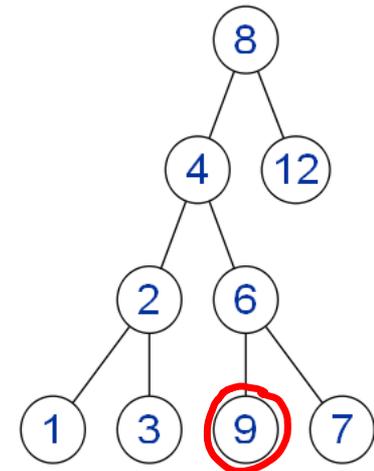
- Anordnung komplett sortiert: Für jeden Knoten gilt:
 - alle Elemente im linken Unterbaum haben einen kleineren Key
 - alle Elemente im rechten Unterbaum haben einen größeren Key



binärer Suchbaum



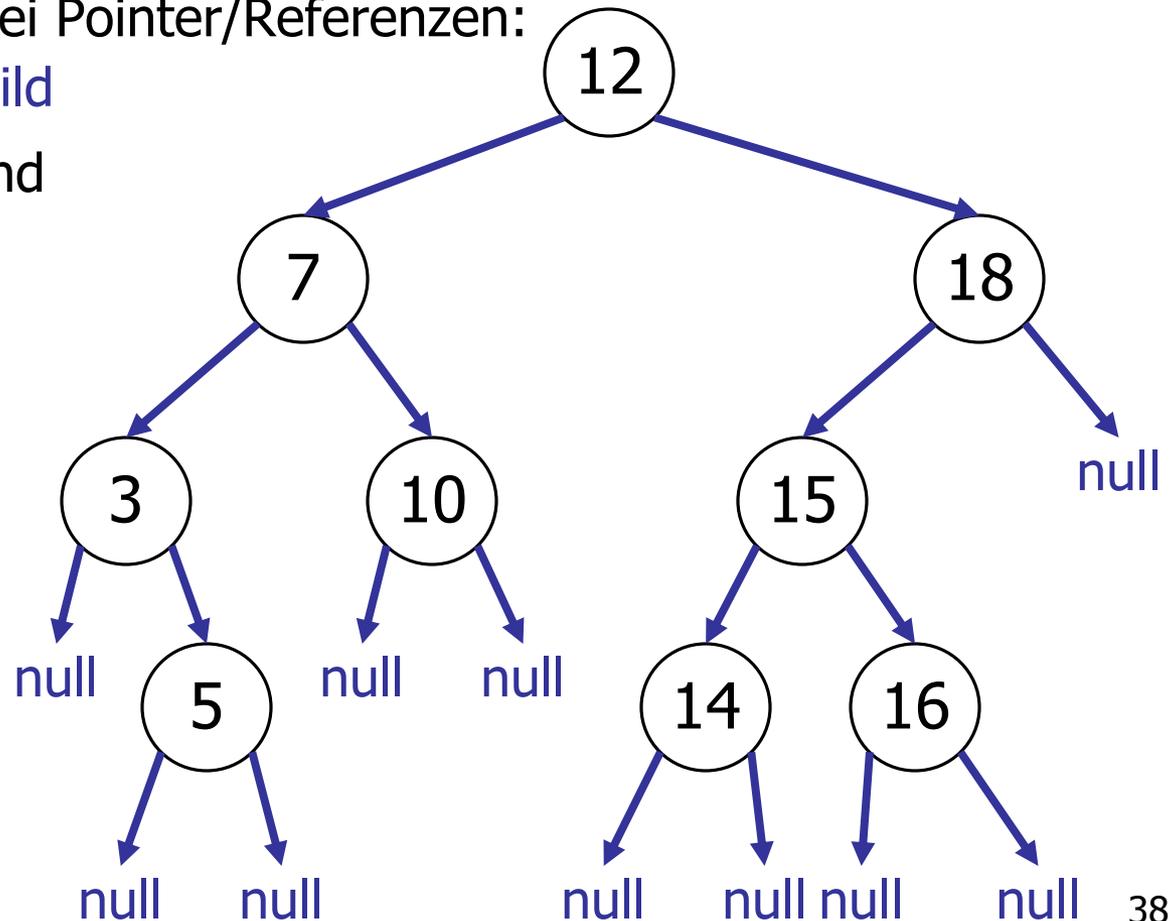
binärer Suchbaum



kein binärer
Suchbaum

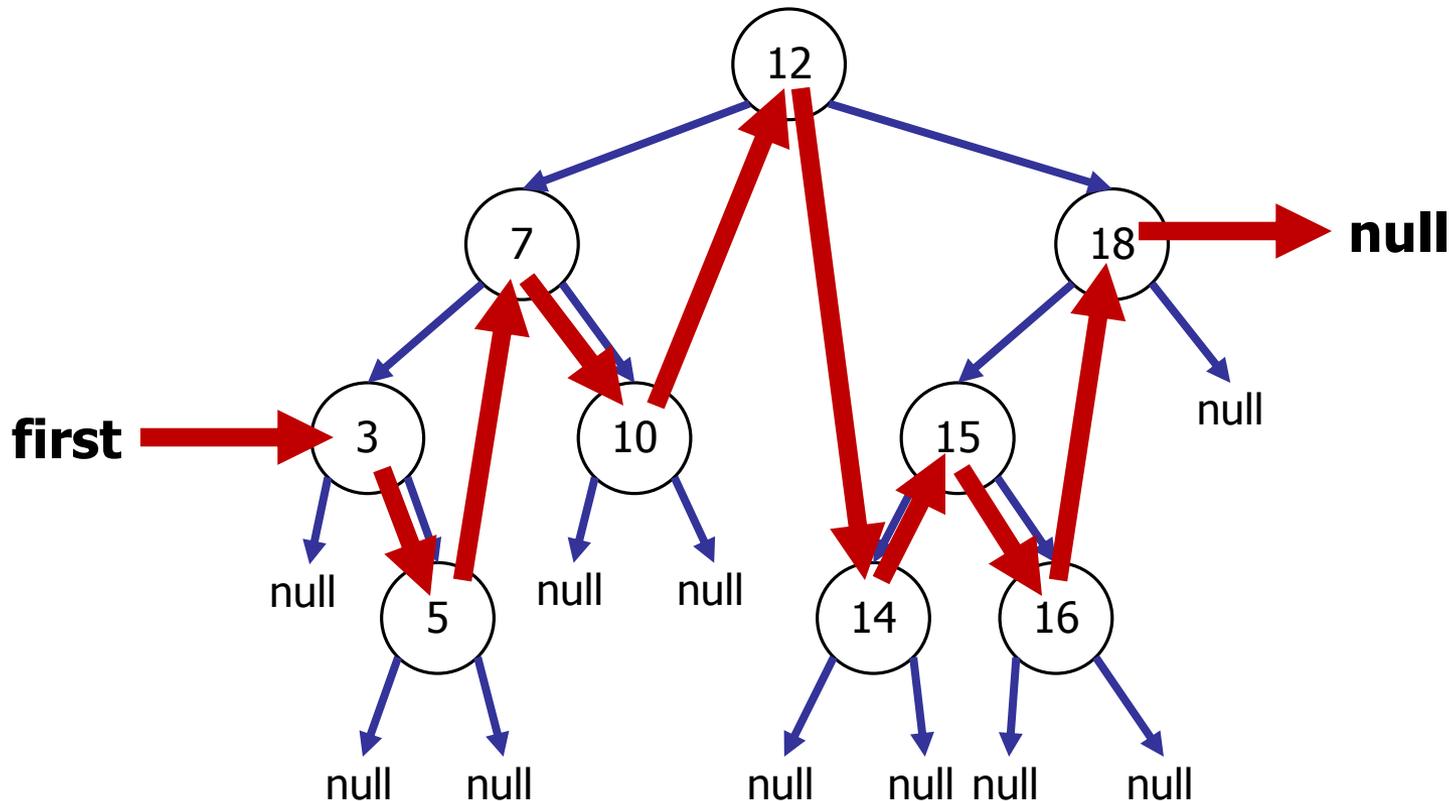
Implementierung

- Für den Heap hatten wir alle Knoten in einem Array gespeichert.
- Hier jetzt mit Pointern/Referenzen wie bei den verketteten Listen:
- Jeder Knoten hat zwei Pointer/Referenzen:
leftChild und rightChild
- null für fehlendes Kind



Binäre Suchbäume — Idee

- Für effiziente Implementierung von `next` / `previous`
 - **gleichzeitig** eine doppelt verkettete Liste der Elemente (hier sind nur die **Vorwärtspointer** dargestellt)
 - die können Sie für's Übungsblatt aber weglassen!



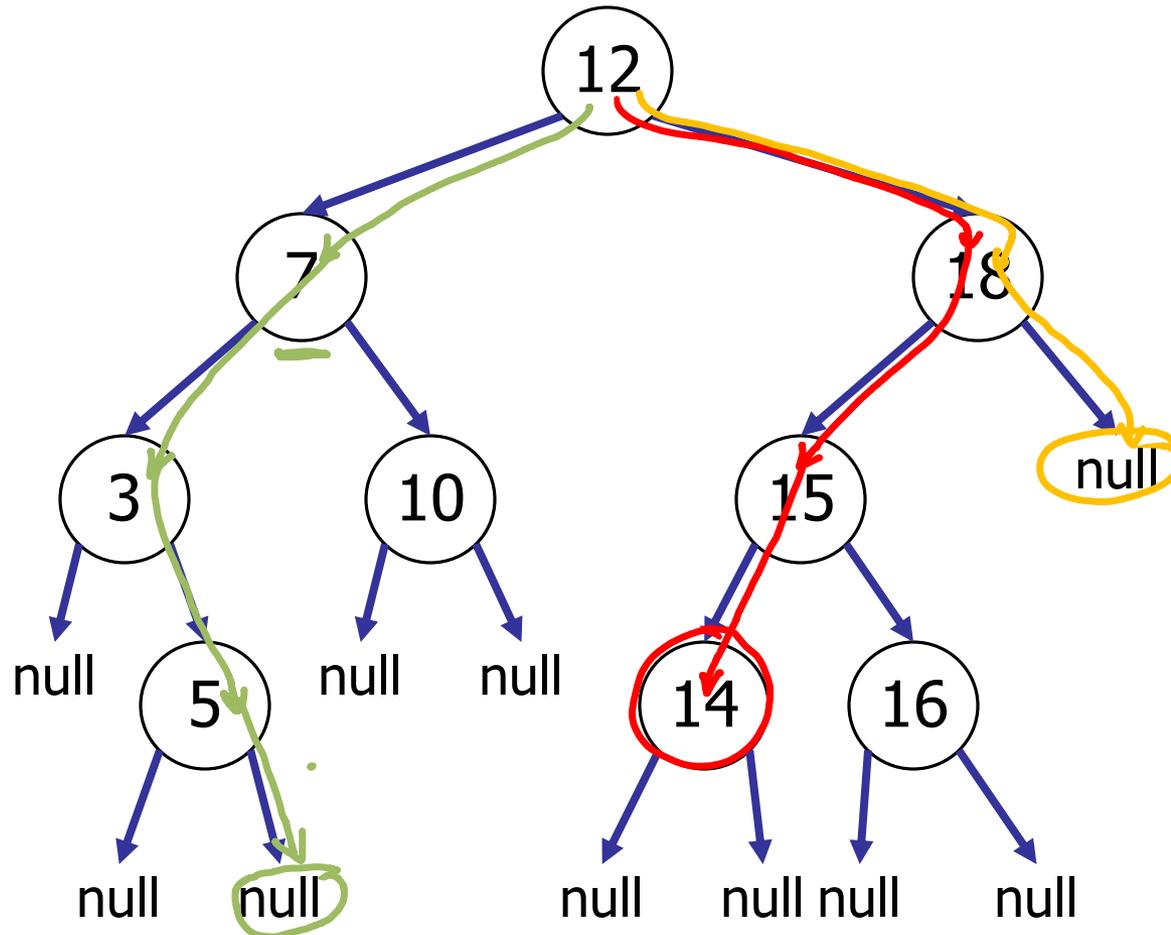
Binäre Suchbäume — Lookup

- Definition `lookup(key)` war:
 - finde das Element mit dem gegebenen Key; falls es das nicht gibt, finde das Element mit dem kleinsten Key der $> key$ ist
- Wir suchen einfach von der Wurzel abwärts
 - und gehen je nach Key links oder rechts
 - und merken uns dabei immer den **letzten** Knoten, an dem wir nach **links** gegangen sind ... warum?

lookup

An jedem Knoten gilt:

Alle Schlüssel im linken Teilbaum $<$ Knoten.Schlüssel $<$ alle Schlüssel im rechten Teilbaum



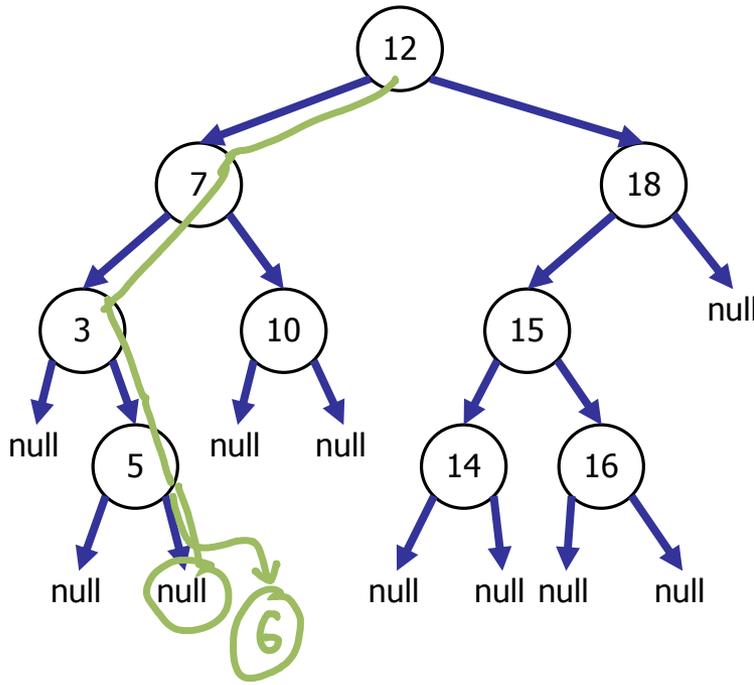
Beispiele

- lookup(14)
- lookup(6)
- lookup(19)

Binäre Suchbäume — Einfügen

■ Wir suchen erstmal den gegebenen Key

- Wenn wir ihn gefunden haben, überschreiben wir einfach den Value an dem Knoten
- Sonst fügen wir an geeigneter Stelle einen neuen Knoten ein

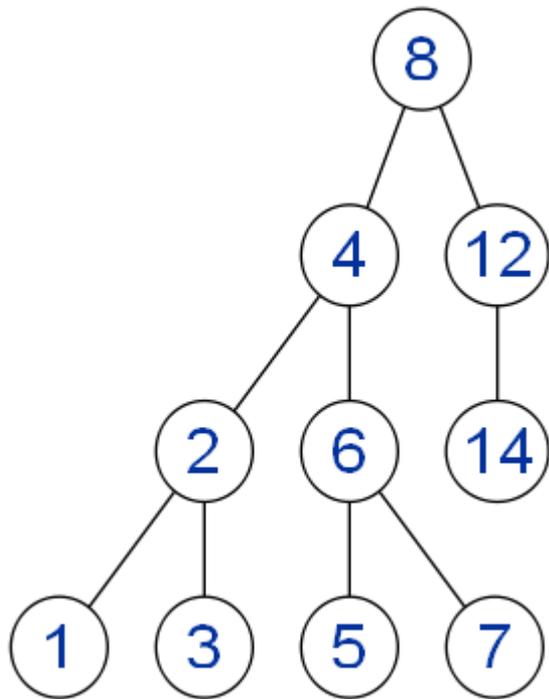


- Wenn es den Key im Baum noch nicht gab, kommen wir an einen null-pointer
- Genau dort müssen wir den neuen Knoten einfügen
- Z.B. `insert(11)`

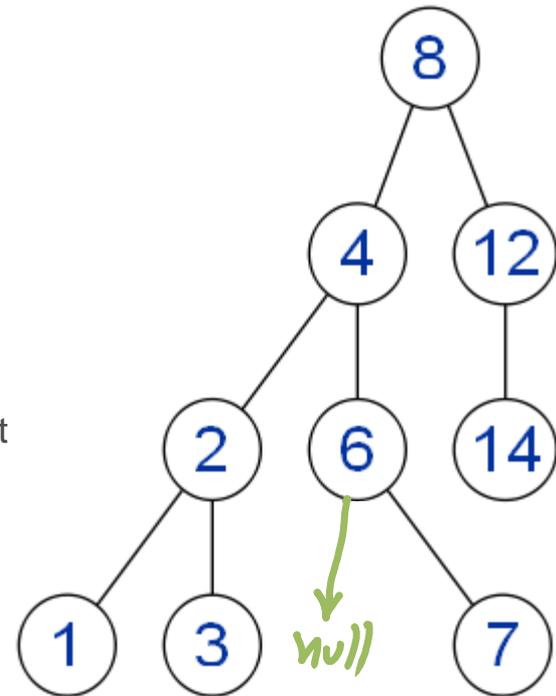
Löschen



- Fall 1: zu löschender Knoten hat keine Kinder



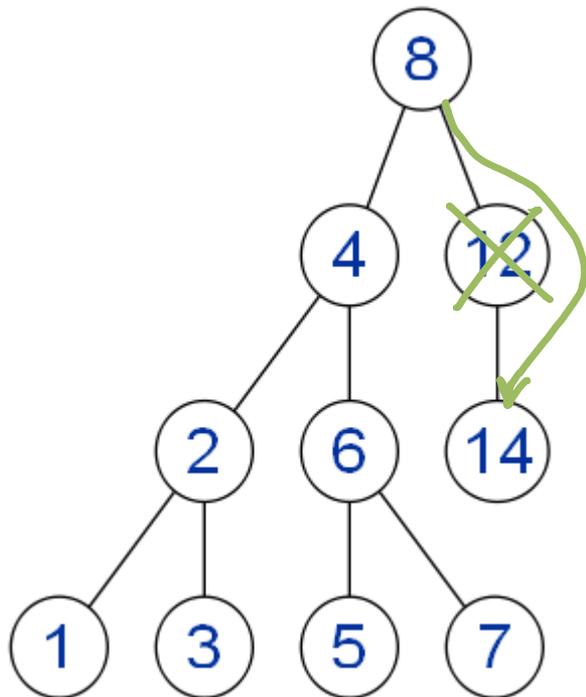
Lösche 5:
ermittle Vater (6)
ermittle, ob 5 linker
oder rechter Sohn ist
setze 6.left = null;



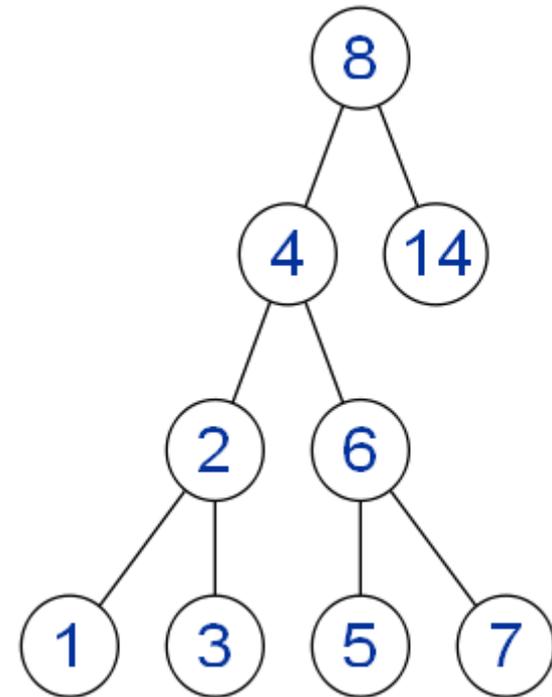
Löschen



- Fall 2: zu löschender Knoten hat ein Kind



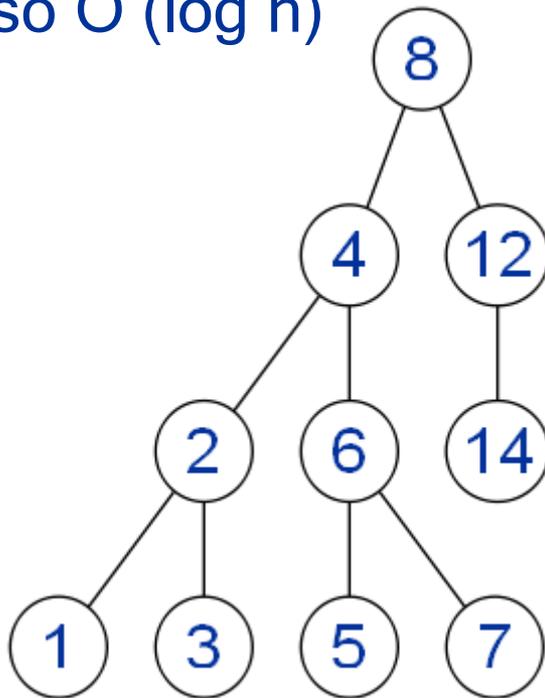
Lösche 12:
ermittle Sohn (14)
ermittle Vater (8)
ermittle, ob 12 linker
oder rechter Sohn ist
setze `8.right = 14`;



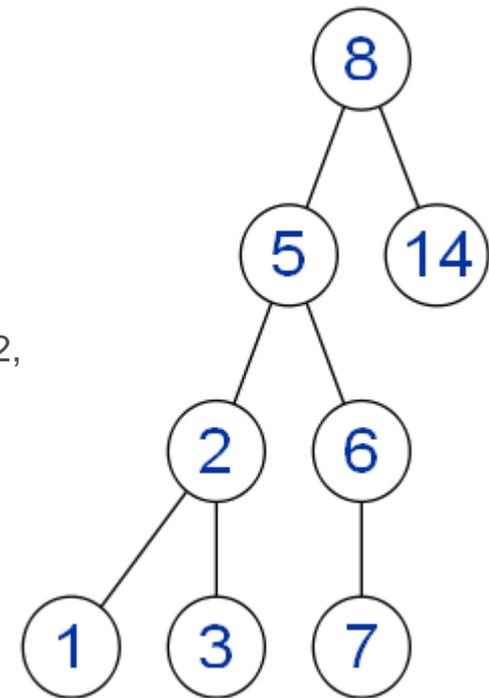
Löschen



- Fall 3: zu löschender Knoten hat zwei Kinder
- Laufzeit ergibt sich aus Laufzeit für `successor`, also $O(\log n)$



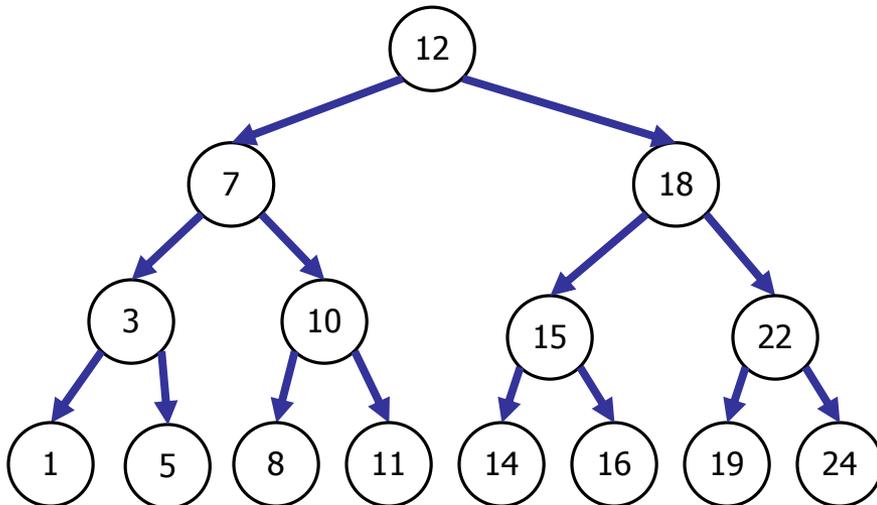
Lösche 4:
ermittle `successor(4) = 5`
ersetze 4 durch 5
lösche 5 (Fall 1 oder Fall 2,
da 5 als Nachfolger von 4
keinen linken Sohn haben
kann).



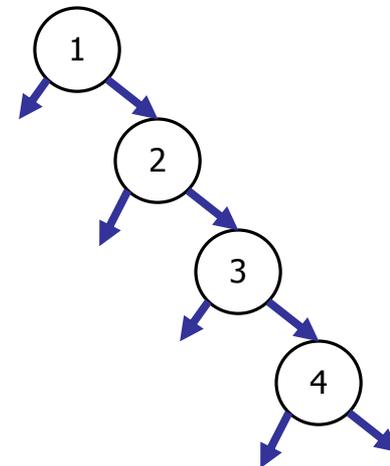
Binäre Suchbäume — Komplexität

■ Wie lange dauern **insert** und **lookup** ?

- Bis zu Zeit $\Theta(d)$, wobei d die Tiefe des Baumes ist
= die größte Tiefe eines Blattes
- Im besten Fall ist das $\Theta(\log n)$, im schlechtesten Fall aber $\Theta(n)$, wobei n die Anzahl der Knoten im Baum ist
- Wenn man **immer** $\Theta(\log n)$ will, muss man den Baum gelegentlich **rebalancieren** → nächste Vorlesung



Vollst. Baum: $d = O(\log n)$



Entart. Baum: $d = n$

- 1) Implementierung eines binären Suchbaums mit
 - insert()
 - lookup()
- 2) Ausprobieren wie schnell die Suche ist, bei verschiedener Einfügereihenfolge der Elemente

Literatur / Links

■ Suchbäume

- In Mehlhorn/Sanders:

7 Sorted Sequences

- In Cormen/Leiserson/Rivest

13 Binary Search Trees

- In Wikipedia

http://de.wikipedia.org/wiki/Binärer_Suchbaum

http://en.wikipedia.org/wiki/Binary_search_tree

- In Java / C++

- die `java.util.TreeMap` und die `std::map` sind typischerweise mit (balancierten) Suchbäumen implementiert