

# Informatik II - SS 2014

## (Algorithmen & Datenstrukturen)

Vorlesung 6 (14.5.2014)

Abstrakte Datentypen,  
Einfache Datenstrukturen

Fabian Kuhn

Algorithmen und Komplexität



**UNI  
FREIBURG**

**Dictionary:** (auch: Maps, assoziative Arrays)

- Verwaltet eine Kollektion von Elementen, wo bei jedes Element durch einen eindeutigen Schlüssel (key) repräsentiert wird

## Operationen:

- *create* : erzeugt einen leeren Dictionary
- *D.insert(key, value)* : fügt neues (*key,value*)-Paar hinzu
  - falls schon ein Eintrag für *key* besteht, wird er ersetzt
- *D.find(key)* : gibt Eintrag zu Schlüssel *key* zurück
  - falls ein Eintrag vorhanden (gibt sonst einen Default-Wert zurück)
- *D.delete(key)* : löscht Eintrag zu Schlüssel *key*

## Dictionary:

### Weitere mögliche Operationen:

- *D.minimum()* : gibt kleinsten *key* in der Datenstruktur zurück
- *D.maximum()* : gibt grössten *key* in der Datenstruktur zurück
- *D.successor(key)* : gibt nächstgrösseren *key* zurück
- *D.predecessor(key)* : gibt nächstkleineren *key* zurück
- *D.getRange(k1, k2)* : gibt alle Einträge mit Schlüsseln im Intervall  $[k1, k2]$  zurück

## Queue (Warteschlange):

- Verwaltet eine Menge (“Sequenz”) von Werten

### Operationen:

- *create* : erzeugt eine leere Queue
- *Q.enqueue(x)* : hängt Element *x* hinten an
- *Q.dequeue()* : gibt vorderstes Element zurück und löscht es
- *Q.isEmpty()* : Ist die Queue leer?

Heisst auch FIFO Queue (FIFO = first in first out)

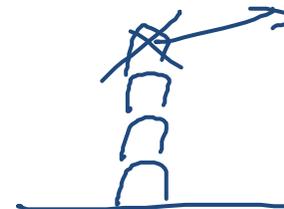
## Stack (Stapel):

- Verwaltet eine Menge (“Sequenz”) von Werten

### Operationen:

- *create* : erzeugt einen leeren Stack
- *S.push(x)* : legt Element *x* auf den Stack
- *S.pop()* : gibt oberstes Element zurück und löscht es
- *S.isEmpty()* : Ist der Stack leer?

Heisst auch LIFO Queue (LIFO = last in first out)



## Heap / Priority Queue (Prioritätswarteschlange):

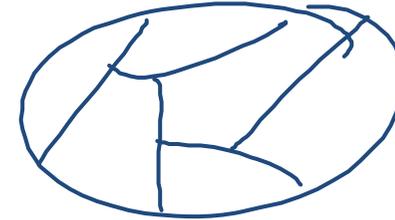
- Verwaltet eine Menge von  $(key, value)$ -Paaren

### Operationen:

- *create* : erzeugt einen leeren Heap
- *H.insert(x, key)* : fügt Element  $x$  mit Schlüssel  $key$  ein
- *H.getMin()* : gibt Element mit kleinstem Schlüssel zurück
- *H.deleteMin()* : löscht Element mit kleinstem Schlüssel
- *H.decreaseKey(x, newkey)* : Falls  $newkey$  kleiner als der aktuelle Schlüssel von  $x$  ist, wird der Schlüssel von  $x$  auf  $newkey$  gesetzt

# Abstrakte Datentypen : Beispiele

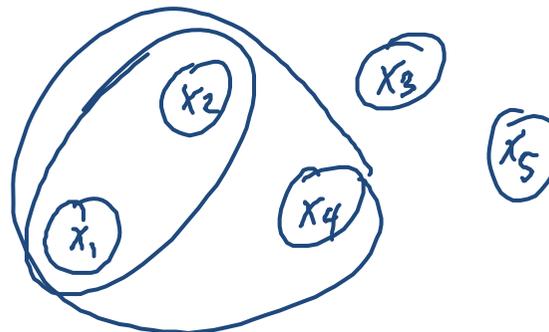
## Union-Find / Disjoint Sets:



- Verwaltet eine Partition von Elementen

### Operationen:

- *create* : erzeugt eine leere Union-Find-DS
- *U.makeSet(x)* : fügt Menge  $\{x\}$  zur Partition hinzu
- *U.find(x)* : gibt Menge mit Element  $x$  zurück
- *U.union(S1, S2)* : vereinigt die Mengen  $S1$  und  $S2$



# Array-Implementierung Stack

Versuchen wir den Stack-Datentyp zu implementieren

- **Operationen:** create, push, pop, isEmpty
- **Annahme:** Stack muss nur für NMAX Elemente Platz bieten

Variablen, um den Zustand des Stack zu speichern:

- stack : Array der Länge NMAX class Stack }
- size : Aktuelle Anzahl Elemente im Stack }

create:

```
stack = new array of length NMAX  
size = 0
```

# Array-Implementierung Stack

isEmpty:

```
return (size == 0)
```

S.push(x):

```
if (size < NMAX)
```

```
    stack[size] = x
```

```
    size += 1
```

S.pop():

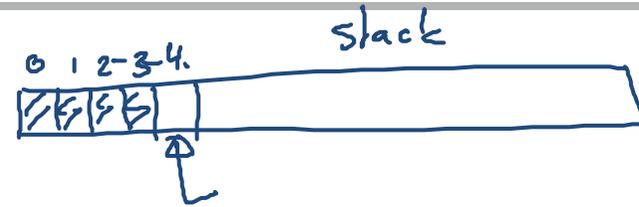
```
if (size == 0)
```

```
    report error (or return default value)
```

```
else
```

```
    size -= 1
```

```
    return stack[size]
```



# Analyse Stack Array Implementierung

**Laufzeit (Zeitkomplexität) der Operationen:** (abhängig von  $n$ )

- create:  $O(1)$ 
  - falls man davon ausgeht, dass Speicher in  $O(1)$  Zeit alloziert werden kann
- push:  $O(1)$
- pop:  $O(1)$  *sehr schnell*
- isEmpty:  $O(1)$

## Nachteile der Implementierung:

- Speicherverbrauch (space complexity) :  $O(NMAX)$ 
  - man braucht immer gleich viel Speicher, egal wie viele Elemente im Stack gespeichert sind!
- Der Stack kann nur  $NMAX$  Elemente aufnehmen...
- Wir werden sehen, wie man beides beheben kann...

# Stack : Anwendungen

- Umdrehen einer Sequenz:

`push(1); push(2); push(3); pop(); pop(); pop()`  
↓        ↓        ↓  
3        2        1

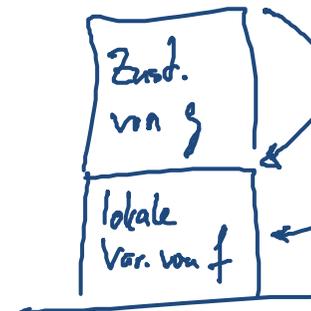
- Undo-Funktion bei Editoren

- lege Beschreibung von (umkehrbaren) Operationen auf Stack ab

- Programmstack für Funktionen/Methoden-Aufrufe

- Bemerkung: Mit einem Stack kann man Rekursion explizit aufschreiben

```
f() {  
  g()  
}
```



# Rekursion explizit mit Stack

```
MergeSort(A):  
  n = A.length
```

```
  MergeSortRec(A, 0, n)
```

```
  MergeSortRec(A, l, r):  
    // Sortiere A[l..r-1]
```

```
  if (r - l > 1) then  
    middle = (l + r) / 2  
    MergeSortRec(A, l, middle)  
    MergeSortRec(A, middle, r)  
    Merge(A, l, middle, r)
```

push((0, n))  
:  
push((0, m))  
push((m, n))

# Rekursion explizit mit Stack

```
MergeSort(A):
```

```
  n = A.length
```

```
  MergeSortRec(A, 0, n)
```

```
MergeSortRec(A,l,r):
```

```
    // Sortiere A[l..r-1]
```

```
  if (r - l > 1) then
```

```
    middle = (l + r) / 2
```

```
    MergeSortRec(A, l, middle)
```

```
    MergeSortRec(A, middle, r)
```

```
    Merge(A, l, middle, r)
```

```
MergeSort(A):
```

```
  n = A.length
```

```
  stack = createEmptyStack()
```

```
  stack.push([false,0,n])
```

```
  while not stack.isEmpty() do
```

```
    [sorted,l,r] = stack.pop()
```

```
    middle = (l + r) / 2
```

```
    if (!sorted) then
```

```
      if (r - l > 1) then
```

```
        stack.push([true,l,r])
```

```
        stack.push([false,l, middle])
```

```
        stack.push([false,middle, r])
```

```
    else
```

```
      Merge(A, l, middle, r)
```

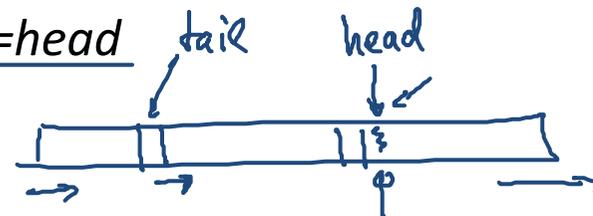
# Array-Implementierung Queue

Versuchen wir den Queue-Datentyp zu implementieren

- **Operationen:** *create*, *enqueue*, *dequeue*, *isEmpty*
- **Annahme:** Queue muss nur für  $NMAX-1$  Elemente Platz bieten

Variablen, um den Zustand des Stack zu speichern:

- *queue* : Array der Länge  $NMAX$
- *head* : Position des vordersten Elements + 1 (zyklisch)
  - Position des nächsten vordersten Elements
- *tail* : Position des hintersten Elements
  - falls die Queue nicht leer ist, sonst ist  $tail=head$



create:

```
queue = new array of length NMAX
```

```
head = 0
```

```
tail = 0
```

# Array-Implementierung Queue

---

# Array-Implementierung Queue

```
S.size():  
    return (head - tail) % mod NMAX
```

```
S.enqueue(x):  
    if (S.size() < NMAX - 1)  
        queue[head] = x  
        head = (head + 1) mod NMAX
```

```
S.dequeue():  
    if (S.size() == 0)  
        report error (or return default value)  
    else  
        x = queue[tail]  
        tail = (tail + 1) mod NMAX  
        return x
```

## Laufzeit (Zeitkomplexität) der Operationen:

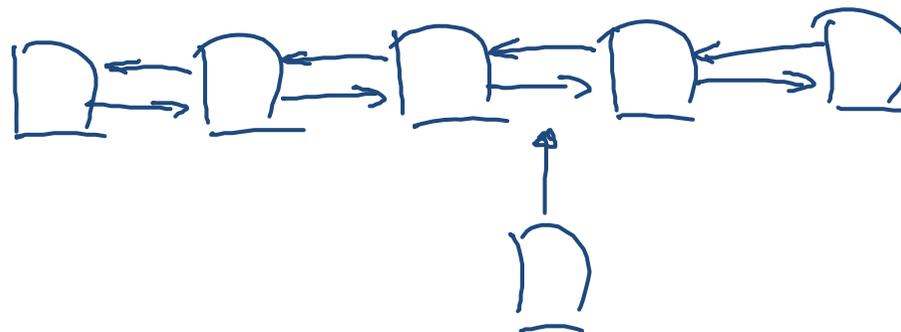
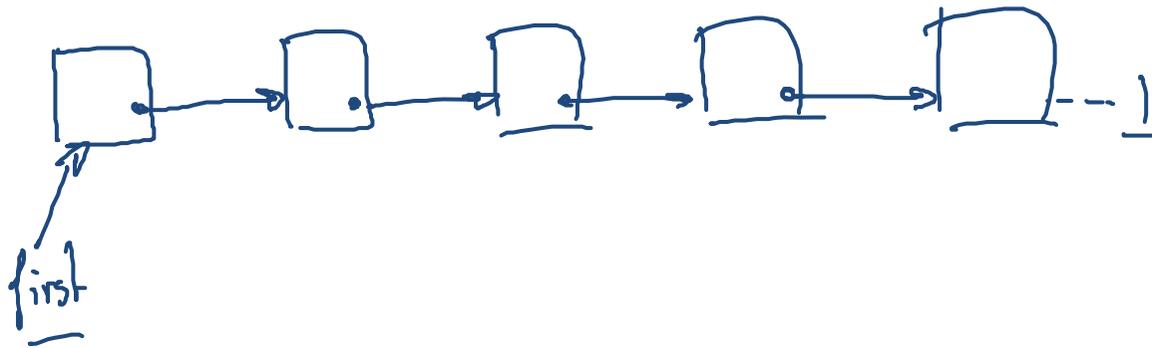
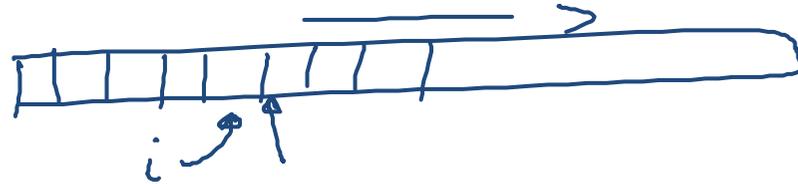
- create:  $O(1)$ 
  - falls man davon ausgeht, dass Speicher in  $O(1)$  Zeit alloziert werden kann
- enqueue :  $O(1)$
- dequeue :  $O(1)$
- isEmpty :  $O(1)$

## Nachteile der Implementierung:

- Speicherverbrauch (space complexity) :  $O(NMAX)$ 
  - man braucht immer gleich viel Speicher, egal wie viele Elemente in der Queue gespeichert sind!
- Die Queue kann nur  $NMAX-1$  Elemente aufnehmen...
- Wir werden gleich sehen, wie man beides beheben kann...

# Verkettete Listen (Linked Lists)

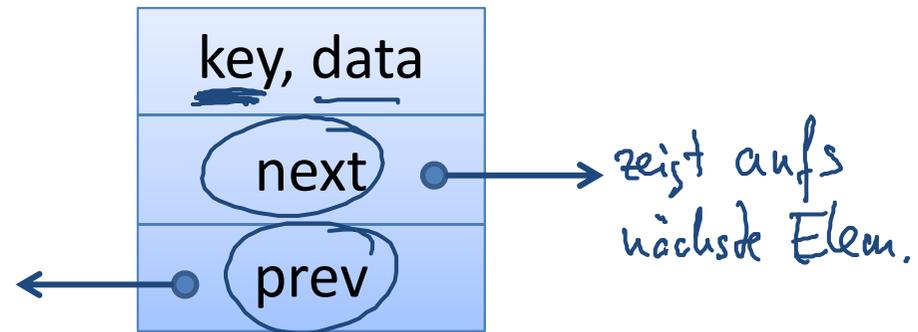
- Datenstruktur, um eine Liste (Sequenz) von Werten zu verwalten



# Listenelemente

- Klasse, um Listenelemente zu beschreiben

## ListElement



### Java:

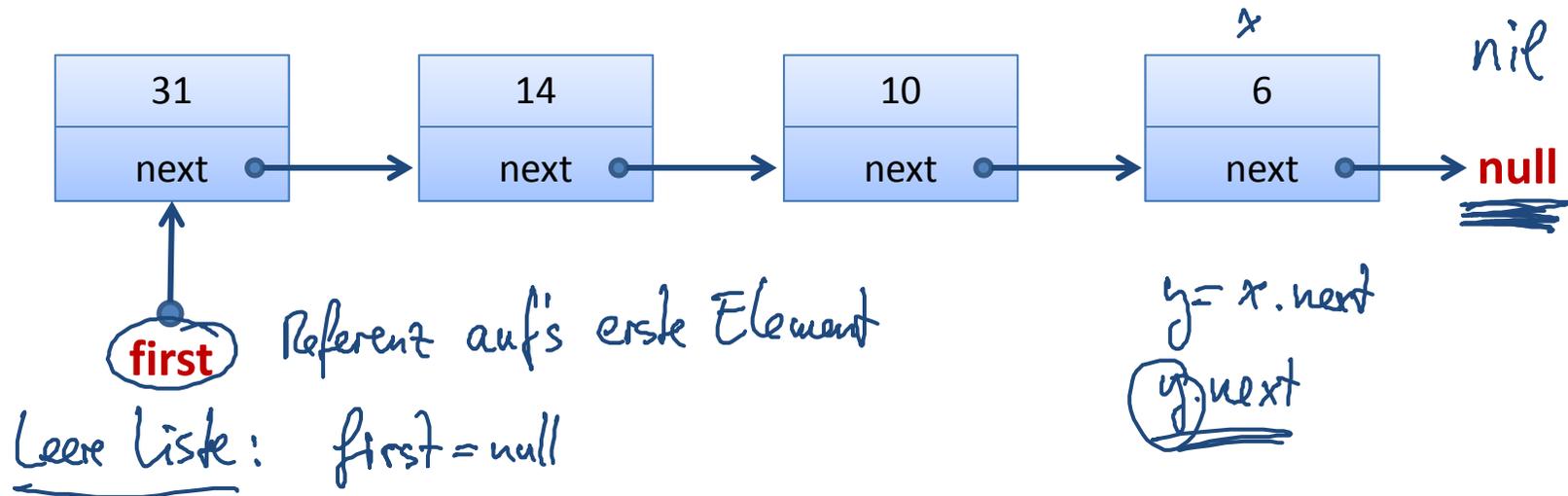
```
public class ListElement {  
    int/String/... key;  
    Object/... data;  
    ListElement next;  
    ListElement prev;  
}
```

### C++:

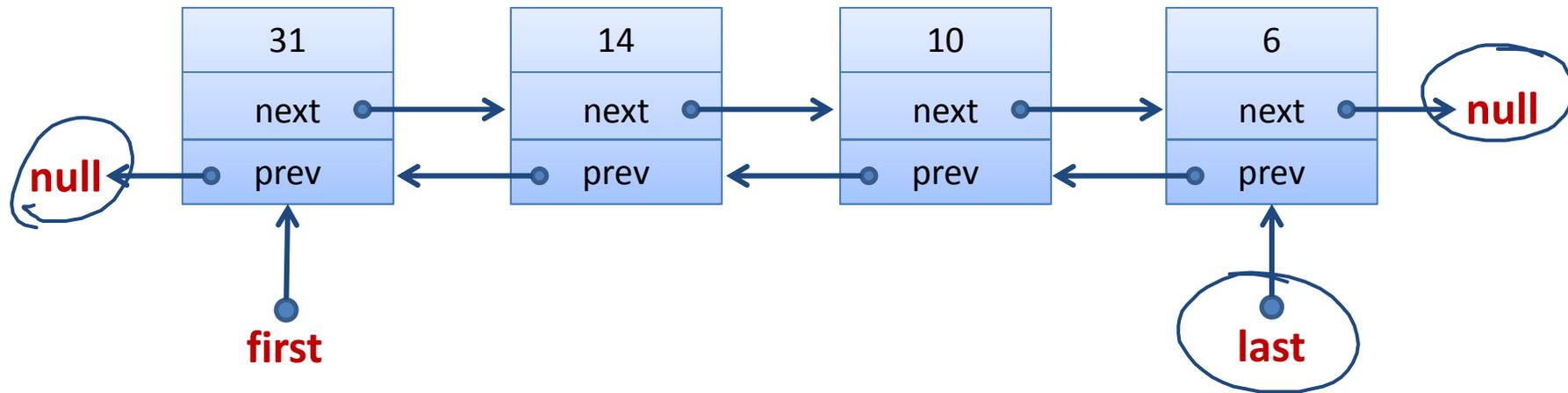
```
class ListElement {  
public/private:  
    int/... key;  
    void*/... data;  
    ListElement* next;  
    ListElement* prev;  
    pointer  
}
```

# Verkettete Listen: Struktur

## Einfach verkettete Liste (Singly Linked List):

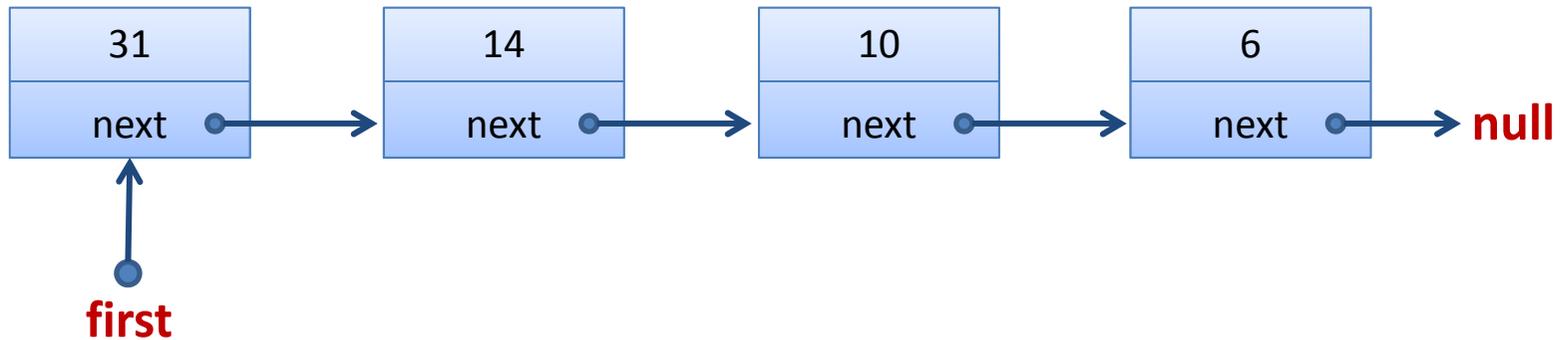


## Doppelt verkettete Liste (Doubly Linked List):



# Suchen in verketteten Listen

## Einfach verkettete Liste (Singly Linked List):



Finde Element mit  $key=x$

ListElement  $current = first$

while ( $current \neq null$  and  $current.key \neq x$ ) {  
     $current = current.next;$

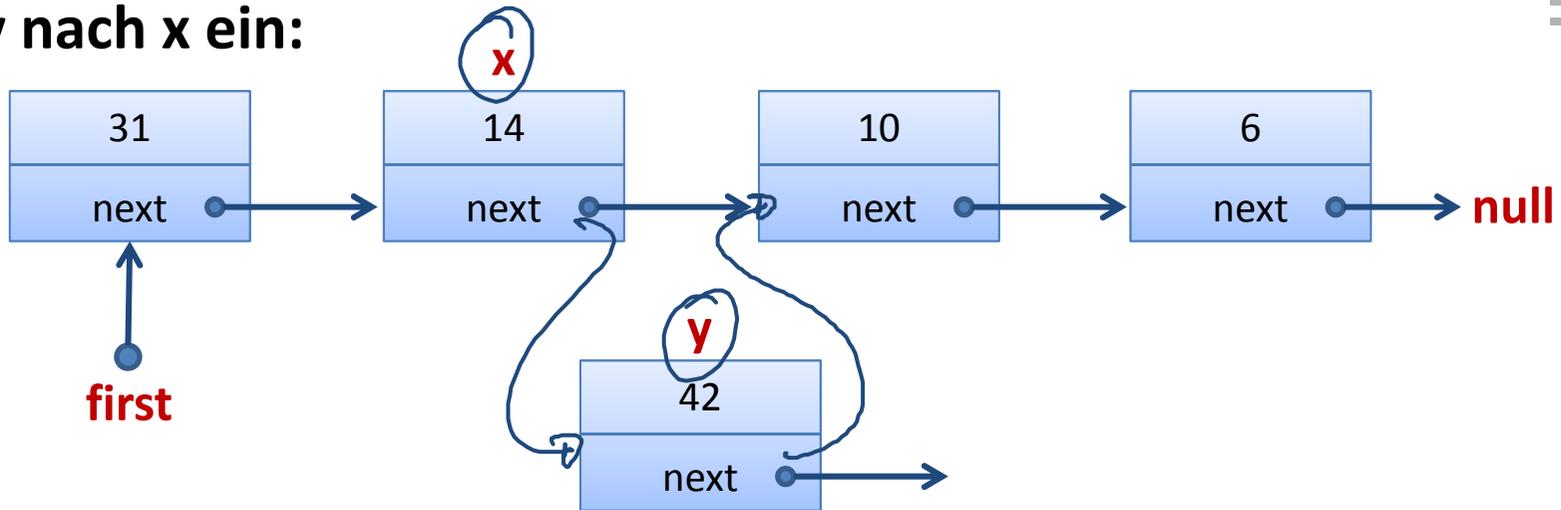
}

Laufzeit:

$O(n)$

# Einfügen in einfach verketteten Listen

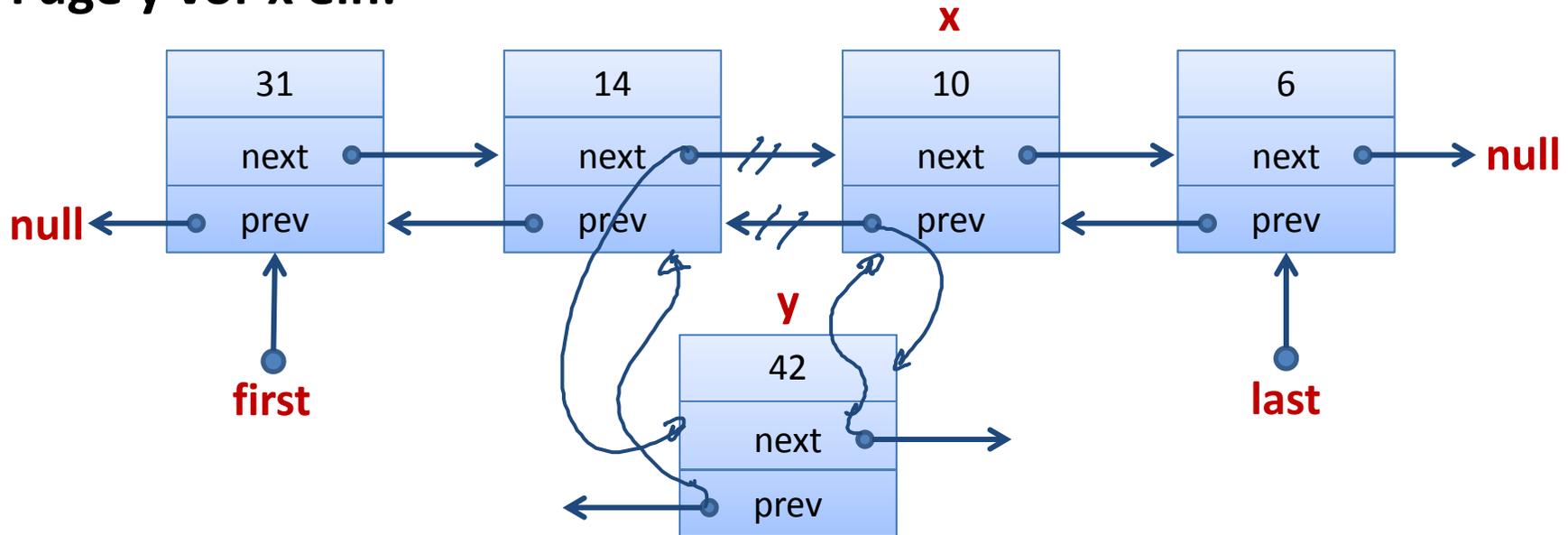
Füge y nach x ein:



**Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!**

# Einfügen in doppelt verketteten Listen

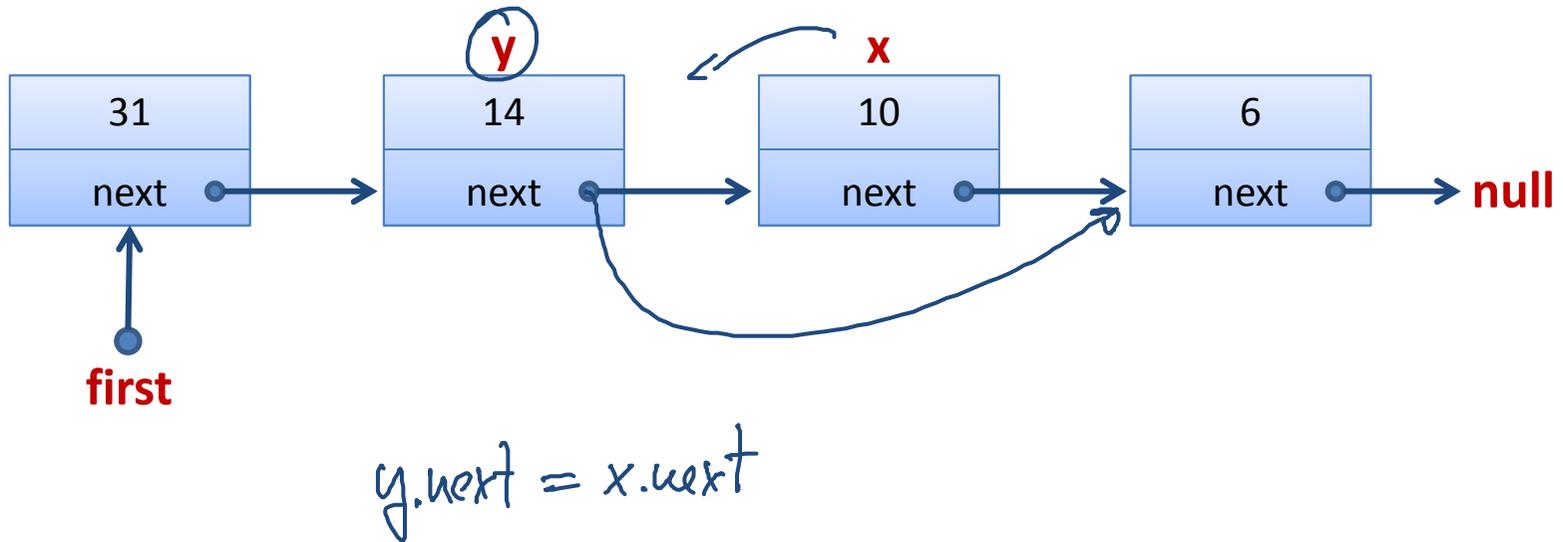
Füge y vor x ein:



**Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!**

# Löschen in einfach verketteten Listen

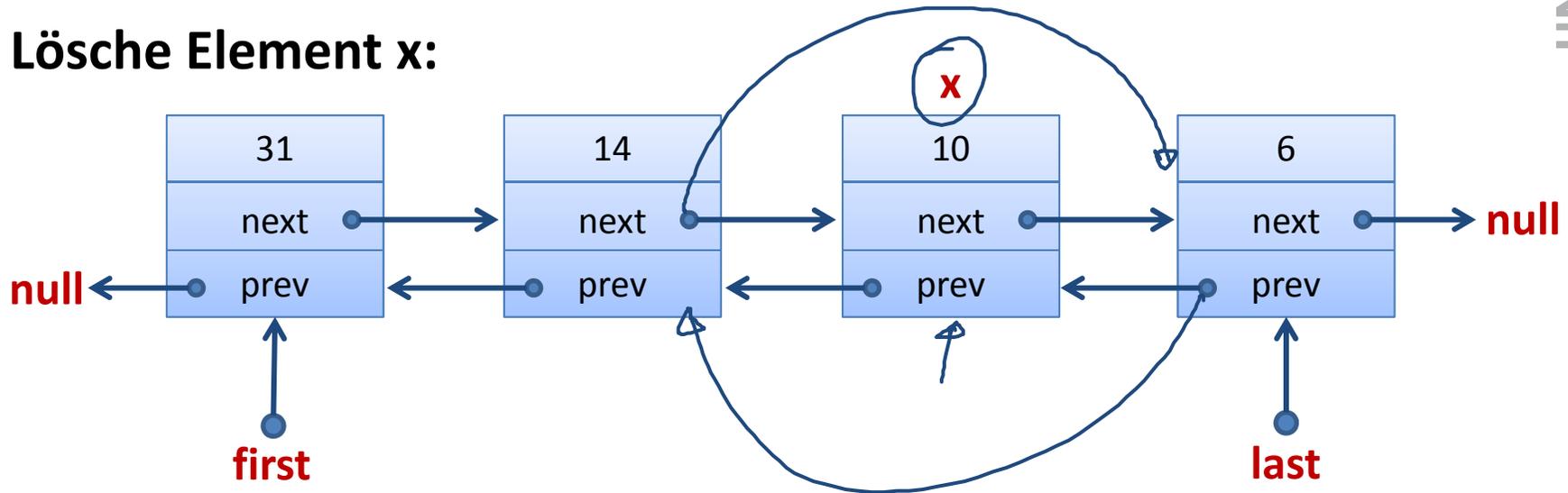
Lösche Element x:



**Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!**

# Löschen in doppelt verketteten Listen

Lösche Element x:



**Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!**

# Laufzeit Listenoperationen

Annahme: Liste hat Länge  $n$

Suche nach Element mit Schlüssel  $x$ :  $O(n)$

Einfügen eines Elements:  $O(1)$   
nur falls eine Referenz auf Element vorher/nachher habe

Löschen eines Elements:  $O(1)$   
"

Aneinanderhängen (concatenate) von zwei Listen:  $O(1)$   
mit last-Pointer

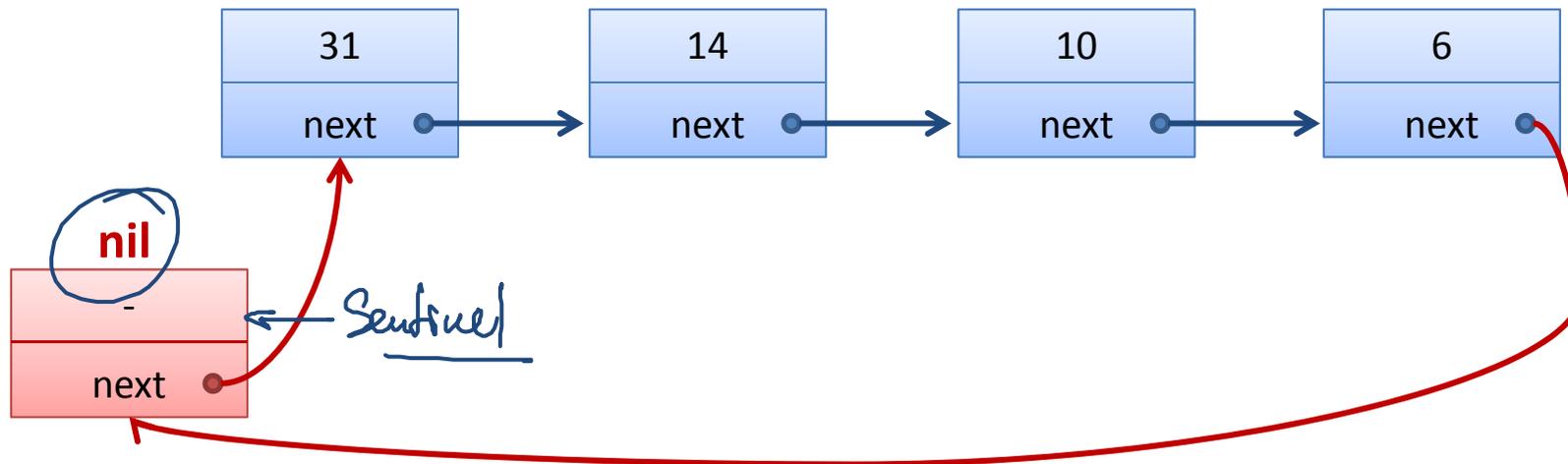
Stack und Queue mit verketteten Listen:

- Alle Operationen in  $O(1)$  Zeit
- Grösse nicht beschränkt, Speicherverbrauch  $O(n)$

# Listen mit Sentinel

## Sentinel:

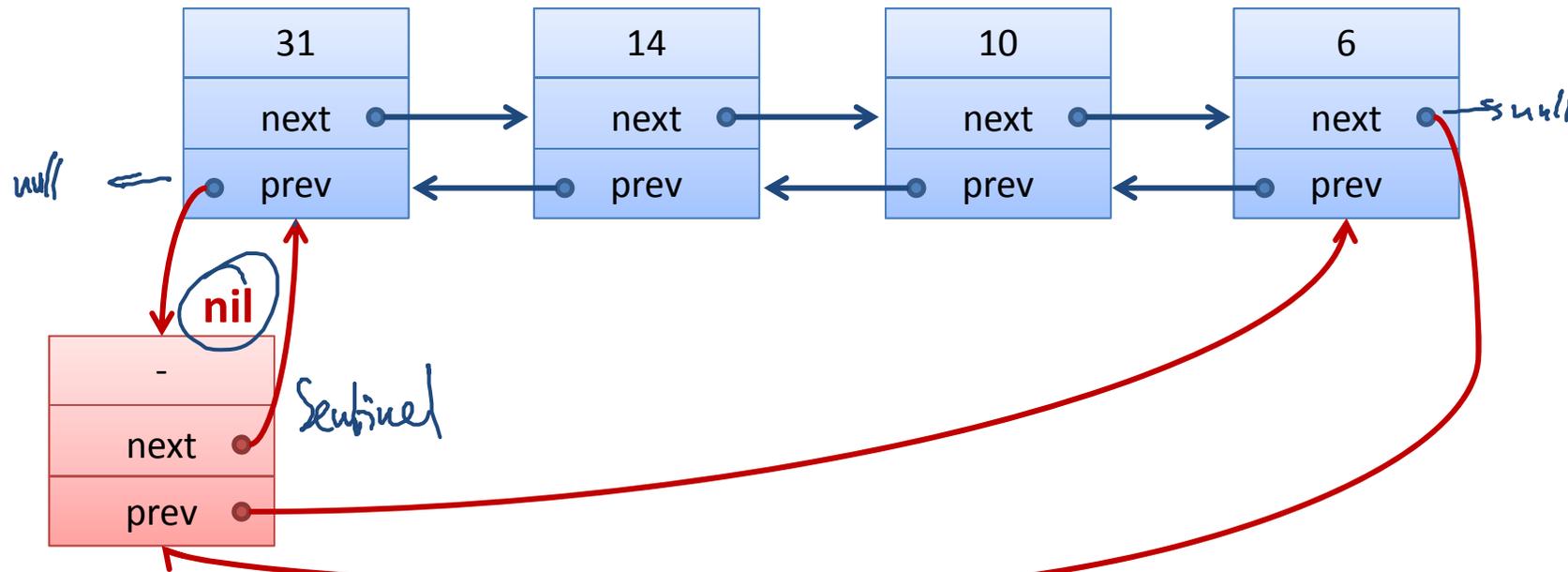
- Ein Dummy-Element, welches Anfang/Ende der Liste bildet



- Anstatt auf first, greift man über nil.next auf die Liste zu
- ersetzt null-Pointer am Schluss der Liste
- Leere Liste: Sentinel zeigt auf sich selbst (nil.next = nil)
- Sentinel ist einfach Teil der Implementierung der Liste und sollte **nicht** nach aussen sichtbar sein.

# Listen mit Sentinel

## Sentinel bei doppelt verketteten Listen:



- Zugriff auf *first*, *last*, greift man auf *nil.next*, *nil.prev* zu
- Ersetzt die beiden null-Pointers am Anfang und Schluss
- Ergibt eine zyklisch verkettete doppelt verlinkte Liste
- Leere Liste:  $nil.next = nil$  ,  $nil.prev = nil$

## **Vorteile:**

- Spezialfälle bei Einfügen/Löschen am Anfang/Ende fallen weg
- Code wird einfacher und allenfalls etwas schneller
- Man vermeidet Null Pointer Exceptions ...
  - Nicht klar, wieviel man bezügl. Robustheit wirklich gewinnt...

## **Nachteile:**

- Bei vielen, kleinen Listen kann der Zusatzplatzverbrauch ins Gewicht fallen (allerdings nie asymptotisch)
- Sentinels machen wohl vor allem da Sinn, wo man den Code wirklich vereinfacht

# Bemerkungen zur Übung

---

- Eine Aufgabe des aktuellen Übungsblattes ist es, eine doppelt verkettete Liste zu programmieren
- Klasse für die Listenelemente und Grundgerüst der DoublyLinkedList-Klasse stellen wir zur Verfügung
  - in Java und C++
- Ob Sie die Liste mit oder ohne Sentinel programmieren, ist Ihnen überlassen
- Für Übungsblatt 3 haben Sie 2 Wochen Zeit
- Tipp: Schauen Sie die Programmieraufgaben jetzt schon an und kommen Sie nächsten Dienstag in die Frage-/Übungsstunde, um konkrete Probleme zu klären!
- Fragestunde am Di, 20.5. wird von 16:15-18:00 im Kinohörsaal stattfinden (82-00-006)

## Java:

- Objekte sind automatisch Referenzen (Pointers)
- Man muss sich nicht ums Speichermanagement kümmern
  - nicht mehr benutzte Objekte werden vom Garbage Collector entfernt
- Neues Objekt vom Typ ListElement generieren:

```
ListElement le = new ListElement(...); ←
```

- next, prev – Pointers sind einfach vom Typ ListElement
  - Bei der vorgegebenen Implementierung über getNext, setNext, getPrevious, setPrevious zugreifbar

# Bemerkungen zu Pointers in Java / C++

## C++:

- Variablen explizit als Pointers auf Objekte definieren
- Man muss sich ~~nicht~~ explizit ums Speichermanagement kümmern
  - nicht mehr benutzte Objekte müssen entfernt werden
- Neues Objekt vom Typ ListElement generieren:

```
ListElement* le = new ListElement(...);
```

- Objekt löschen (le ist vom Typ ListElement\*)

```
delete le;
```

- next, prev – Pointers sind vom Typ ListElement\*
  - Bei der vorgegebenen Implementierung über getNext, setNext, getPrevious, setPrevious zugreifbar
  - Da le ein Pointer ist, le->getNext() statt le.getNext()  
(\*le).