# Informatik II - SS 2016

## (Algorithmen & Datenstrukturen)

Vorlesung 6 (6.5.2016)

## Abstrakte Datentypen, Einfache Datenstrukturen

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

# Abstrakte Datentypen : Beispiele

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*

# Abstrakte Datentypen : Beispiele

UNI FREIBURG

## 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

# Abstrakte Datentypen : Beispiele

## Queue (Warteschlange):

- Verwaltet eine Menge (“Sequenz”) von Werten

**Operationen:**

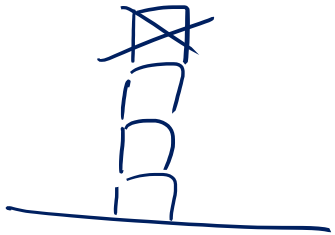
- *create* : erzeugt eine leere Queue
- *Q.enqueue(x)* : hängt Element *x* hinten an
- *Q.dequeue()* : gibt vorderstes Element zurück und löscht es
- *Q.isEmpty()* : Ist die Queue leer?

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

# Abstrakte Datentypen : Beispiele

## 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)

# Abstrakte Datentypen : Beispiele

UNI FREIBURG

## 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

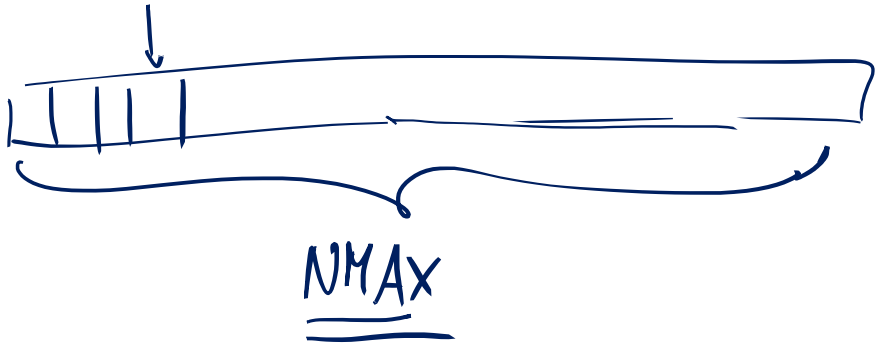
# Array-Implementierung Stack

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*
- $size$ : Aktuelle Anzahl Elemente im Stack



# Analyse Stack Array Implementierung

UNI FREIBURG

**Laufzeit (Zeitkomplexität) der Operationen:**

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

**Nachteile der Implementierung:**

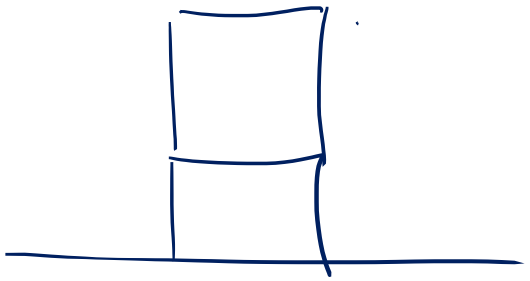
- Speicherverbrauch (space complexity) : $O(NMAX)$
  - man braucht immer gleich viel Speicher, egal wie viele Elemente im Stack gespeichert sind!
- Der Stack kann nur *NMAX* Elemente aufnehmen…
- Wir werden sehen, wie man beides beheben kann…

# Stack : Anwendungen

- Umdrehen einer Sequenz:

- 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



# 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)

# 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)
```

[false, l, 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)
```

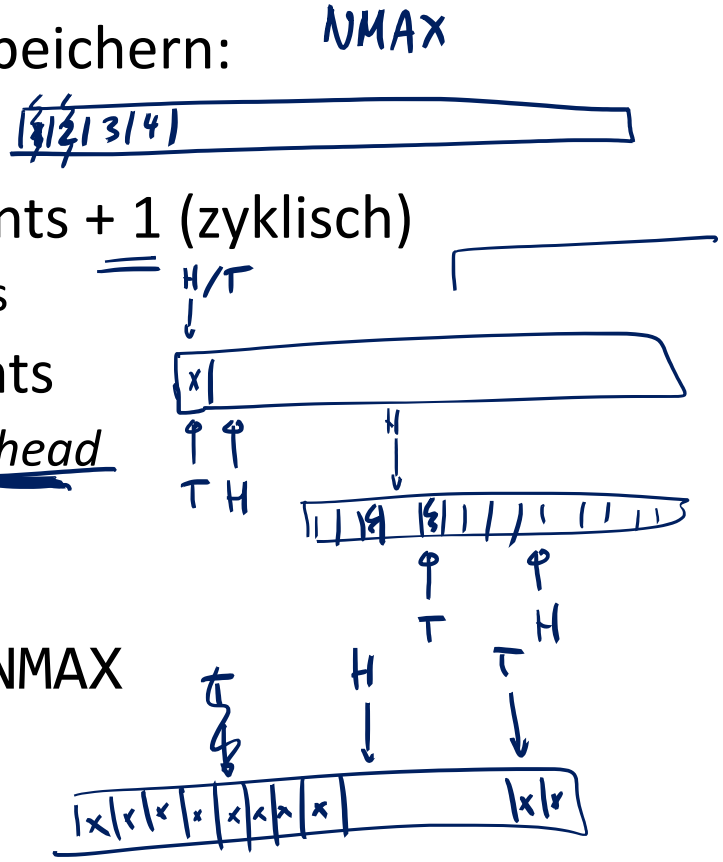
# Array-Implementierung Queue

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

```
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
```

0, ..., NMAX - 1

# Analyse Queue Array Implementierung

UNI FREIBURG

**Laufzeit (Zeitkomplexität) der Operationen:**

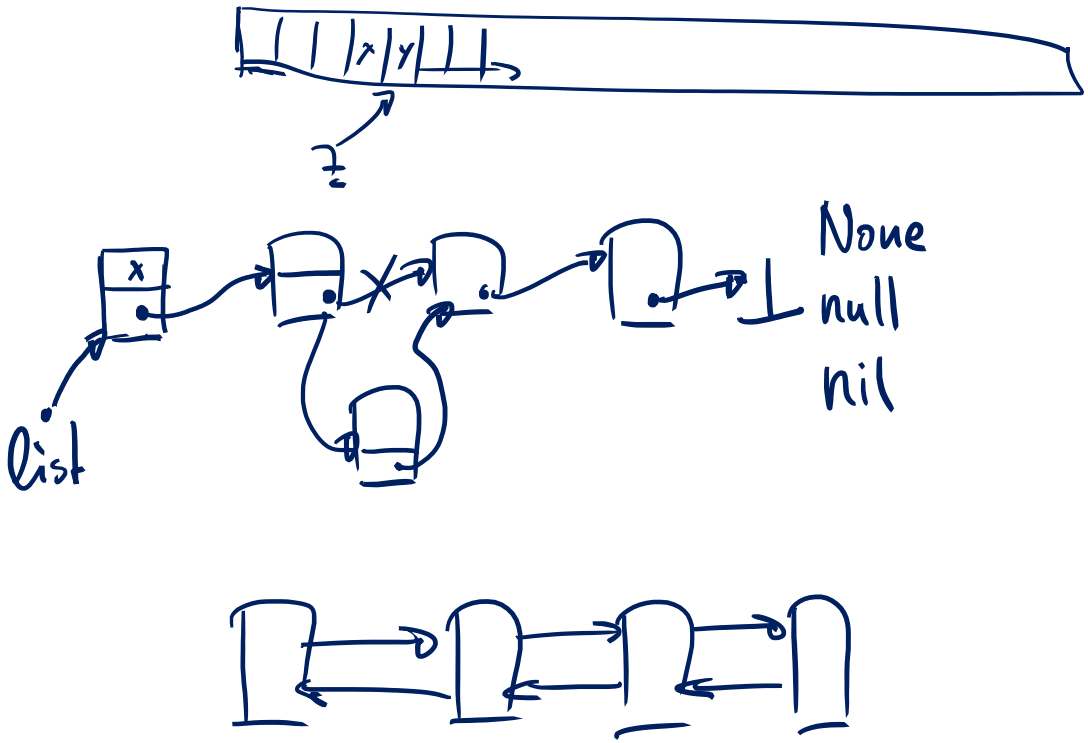
- create: $O(1)$
  - falls man davon ausgeht, dass Speicher in $O(1)$ Zeit alloziert werden kann
- enqueue : $O(1)$
- dequeue : $O(1)$
- isEmpty : $O(1)$

**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…

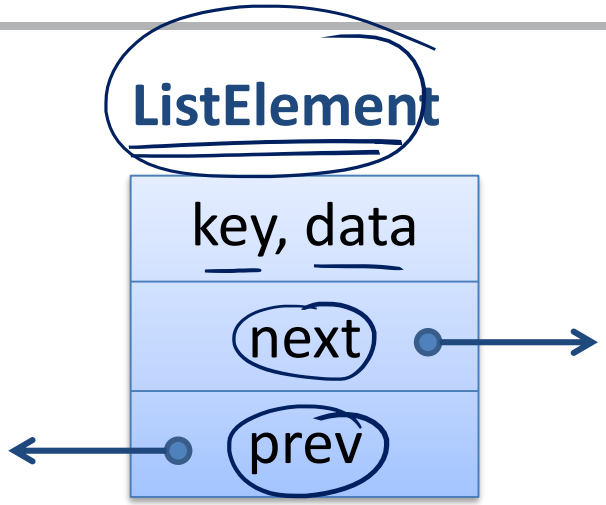
# Verkettete Listen (Linked Lists)

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



# Listenelemente

- Klasse, um Listenelemente zu beschreiben

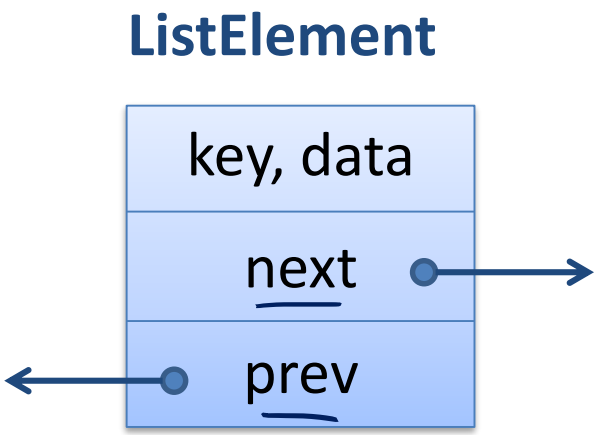



**Python:**

```
class ListElement:

    def __init__(self, key=0, data=None, next=None, prev=None):
        self.key = key
        self.data = data
        self.next = next
        self.prev = prev
```

# Listenelemente

- Klasse, um Listenelemente zu beschreiben




**Java:**

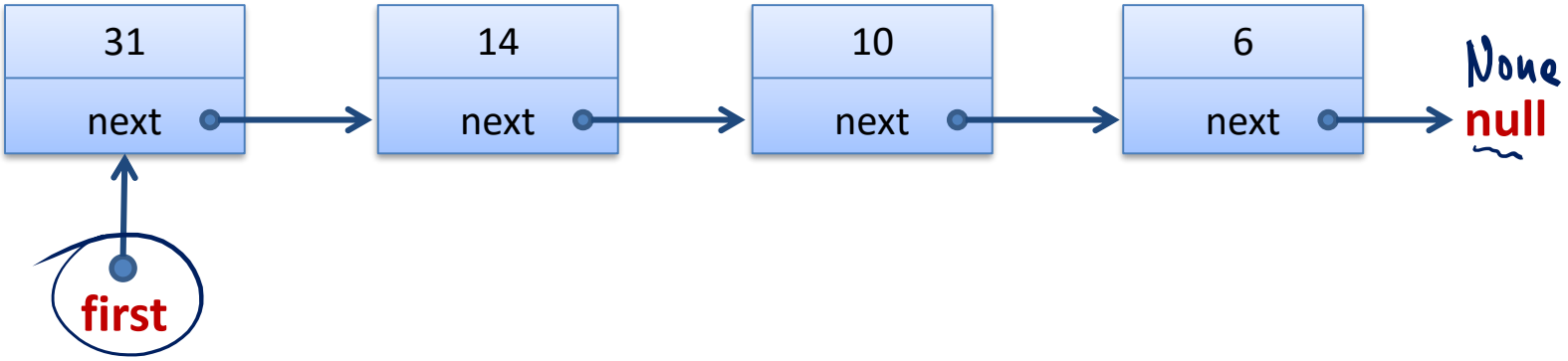
```
public class ListElement {

  int/String/… key;
  Object/… data;

  ListElement next;
  ListElement prev;

}
```

**C++:**

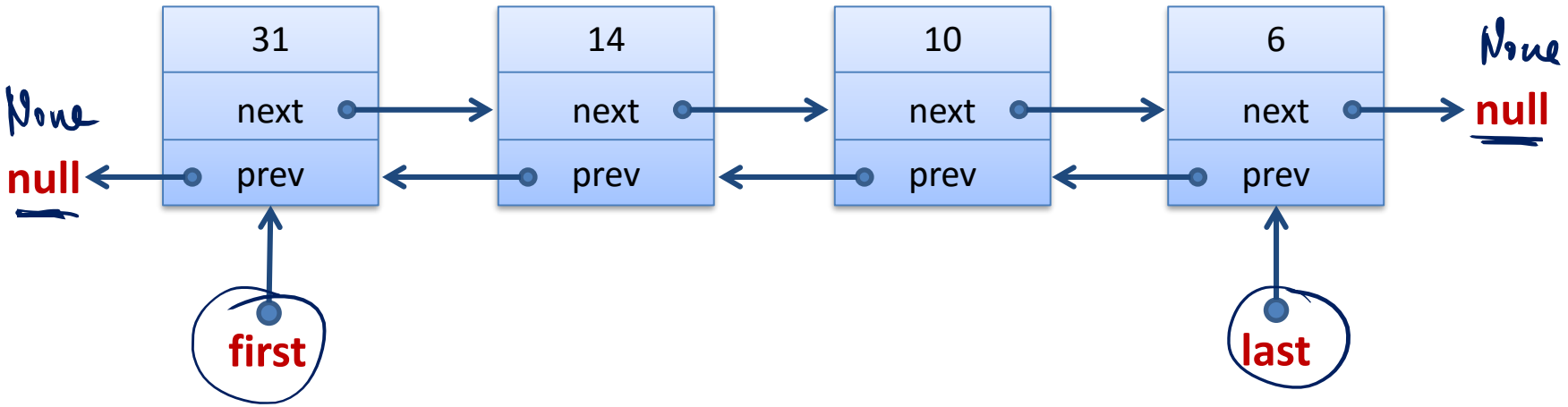
```
class ListElement {
public/private:

  int/… key;
  void*/… data;

  ListElement* next;
  ListElement* prev;

}
```

# Verkettete Listen: Struktur

## Einfach verkettete Liste (Singly Linked List):

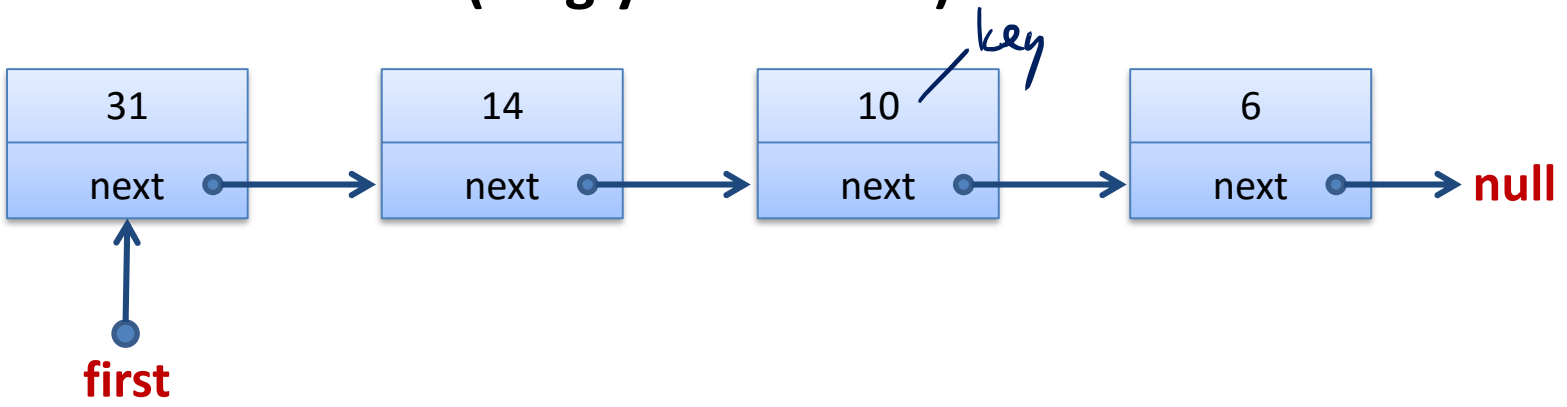



## Doppelt verkettete Liste (Doubly Linked List):



# Suchen in verketteten Listen

**Einfach verkettete Liste (Singly Linked List):**



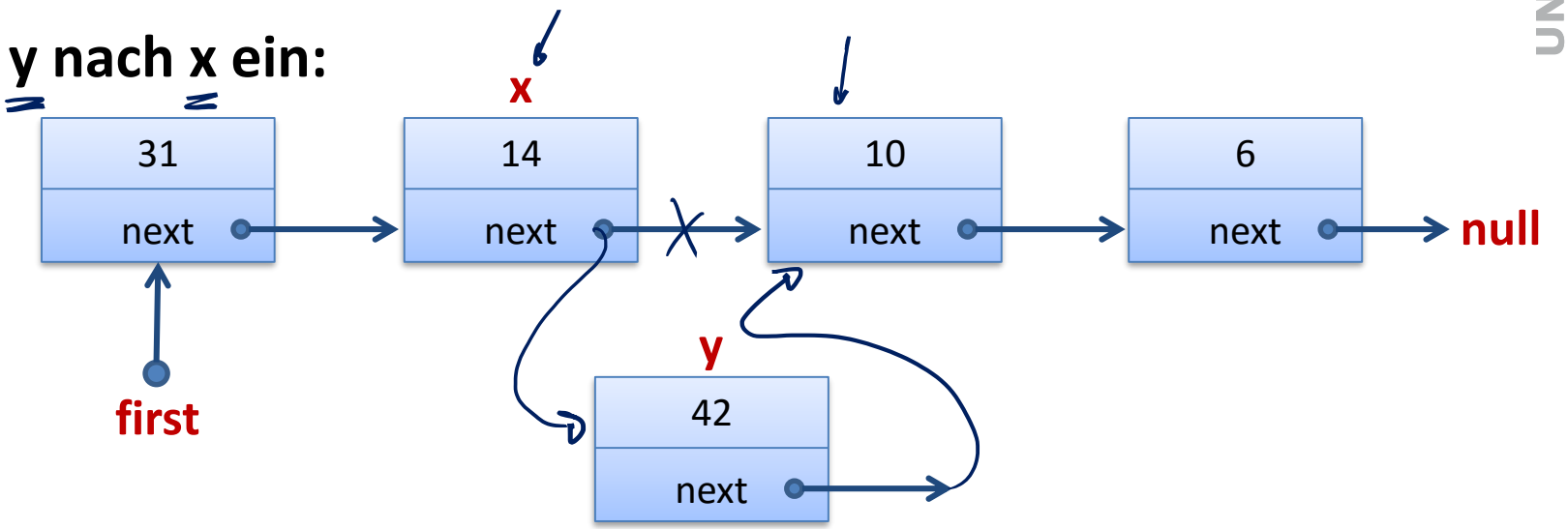

Finde Element mit Schlüssel x:

```
current = first
while current != None and current.key != x:
    current = current.next
```

Laufzeit: $O(n)$

# Einfügen in einfach verketteten Listen

**Füge y nach x ein:**



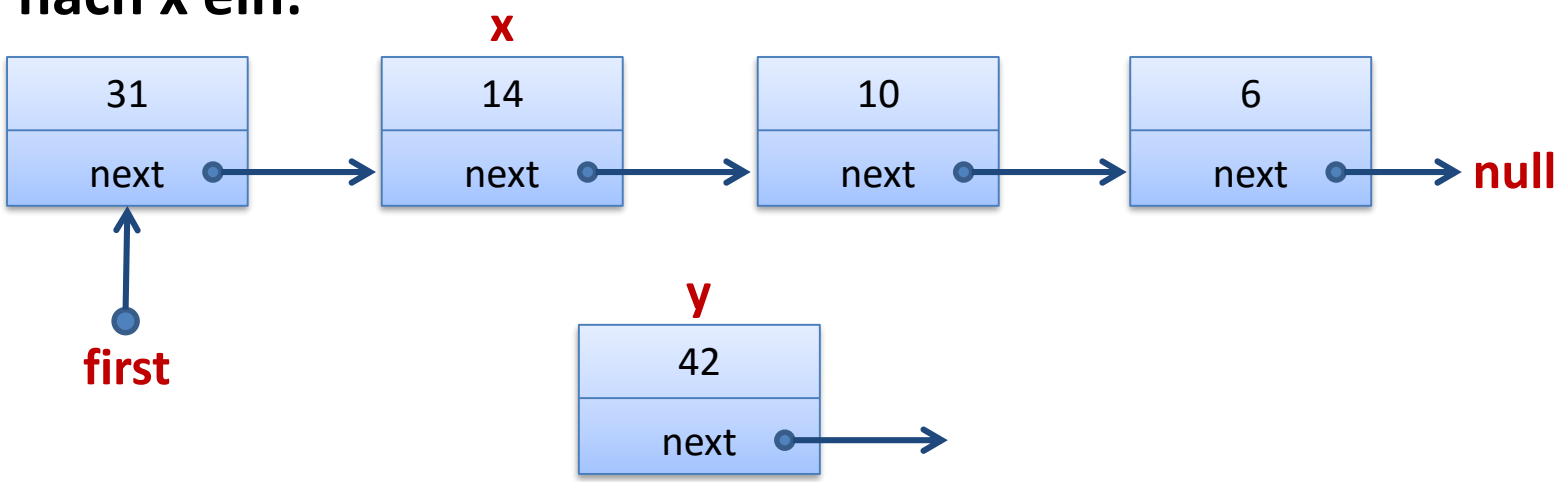

$$\begin{cases} y.next = x.next \\ x.next = y \end{cases}$$

$O(1)$ Zeit

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

# Einfügen in einfach verketteten Listen

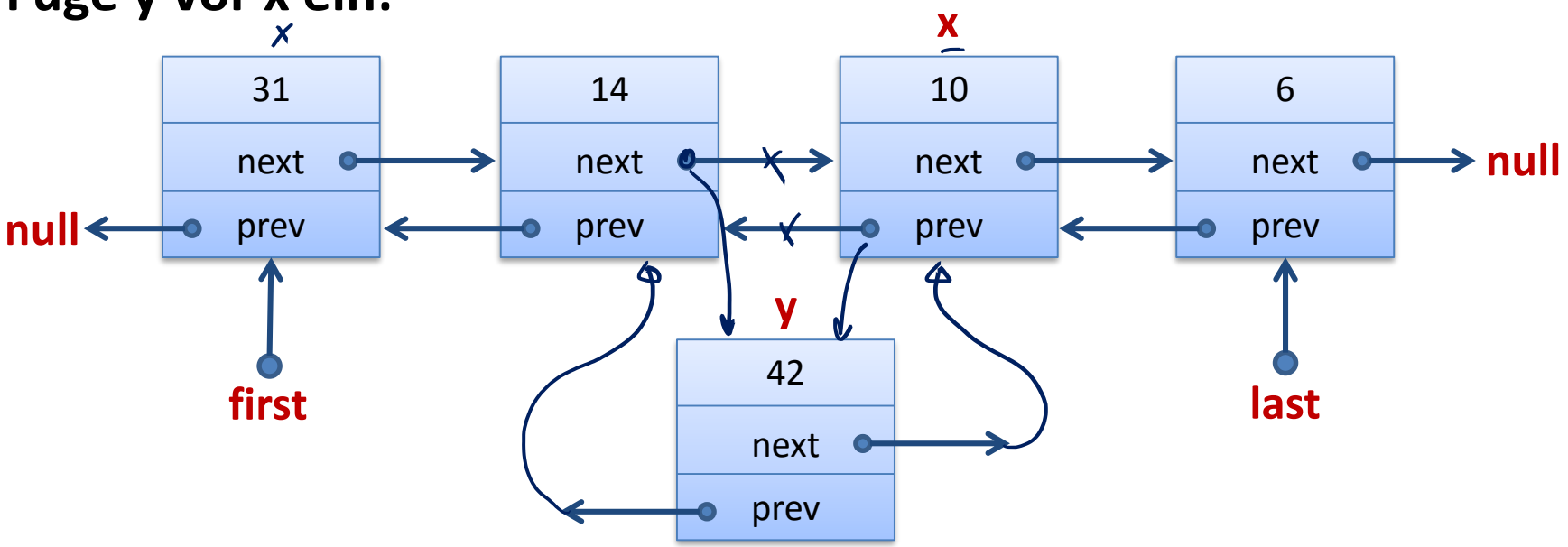
**Füge y nach x ein:**




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

# Einfügen in doppelt verketteten Listen

**Füge y vor x ein:**

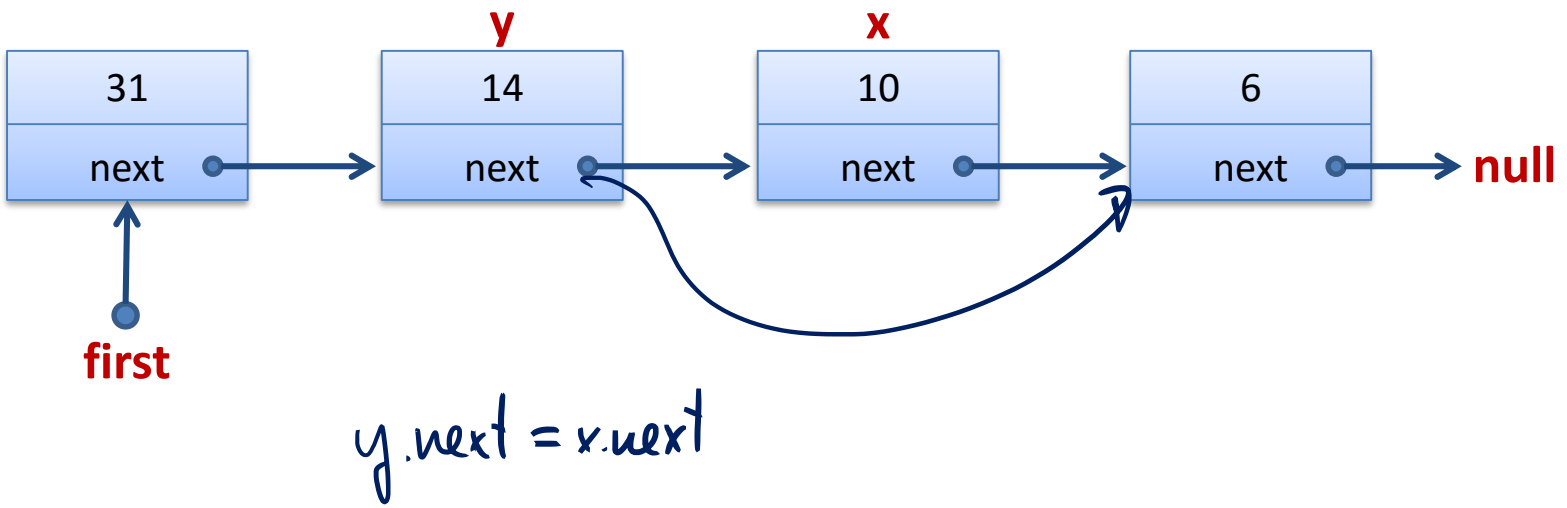



```
y.next = x
y.prev = x.prev
y.prev.next = y
X.prex = y
```

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

# Löschen in einfach verketteten Listen

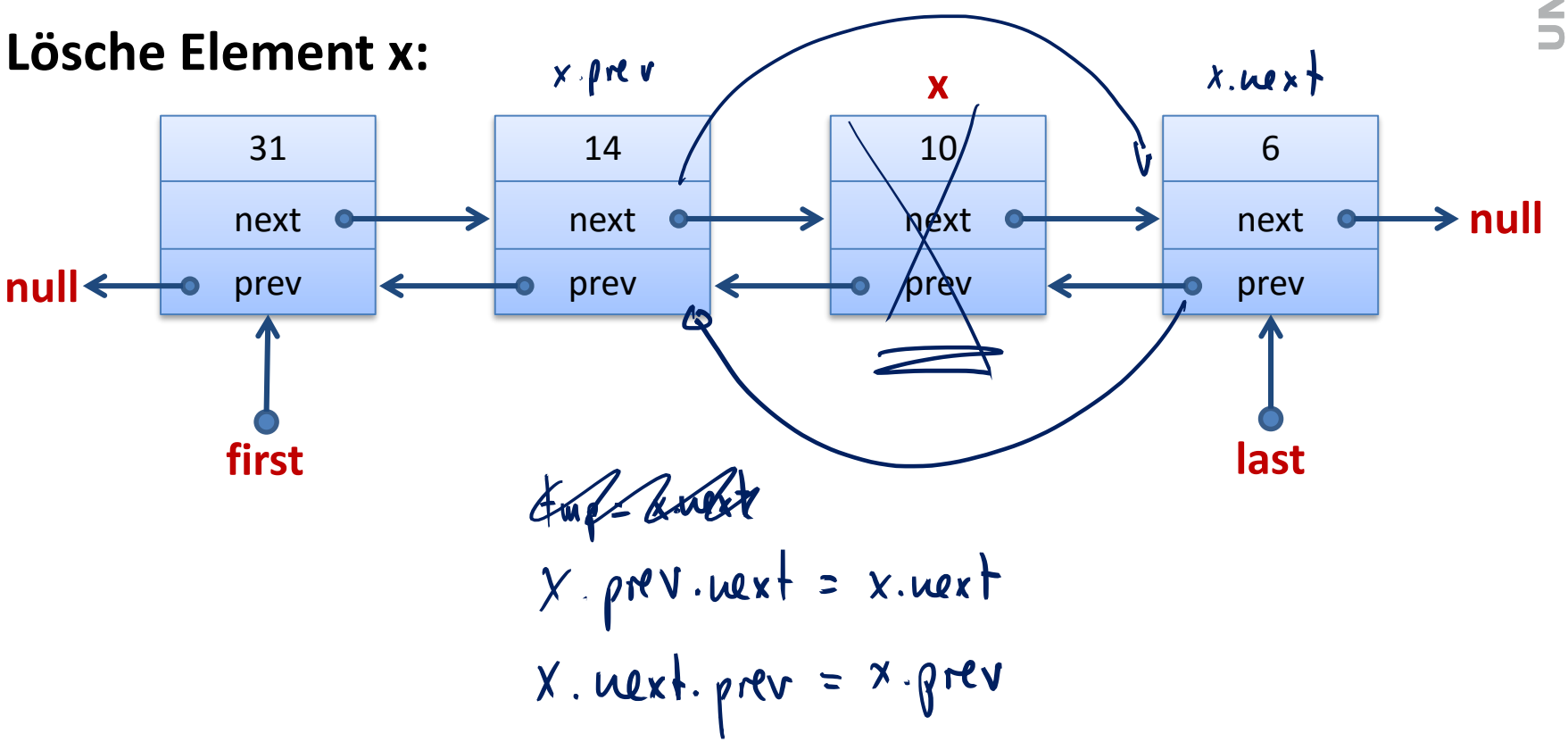
**Lösche Element x:**




**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

UNI FREIBURG

**Annahme:** Liste hat Länge $n$

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

**Einfügen** eines Elements:

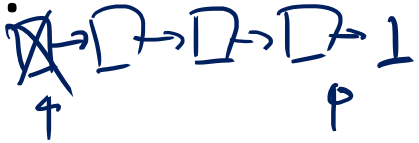
falls Referent auf Vorgänger : $O(1)$ , sonst $O(n)$

**Löschen** eines Elements: $O(1)$

**Aneinanderhängen (concatenate)** von zwei Listen: $O(1)$

mit last-Referent

**Stack und Queue mit verketteten Listen:**

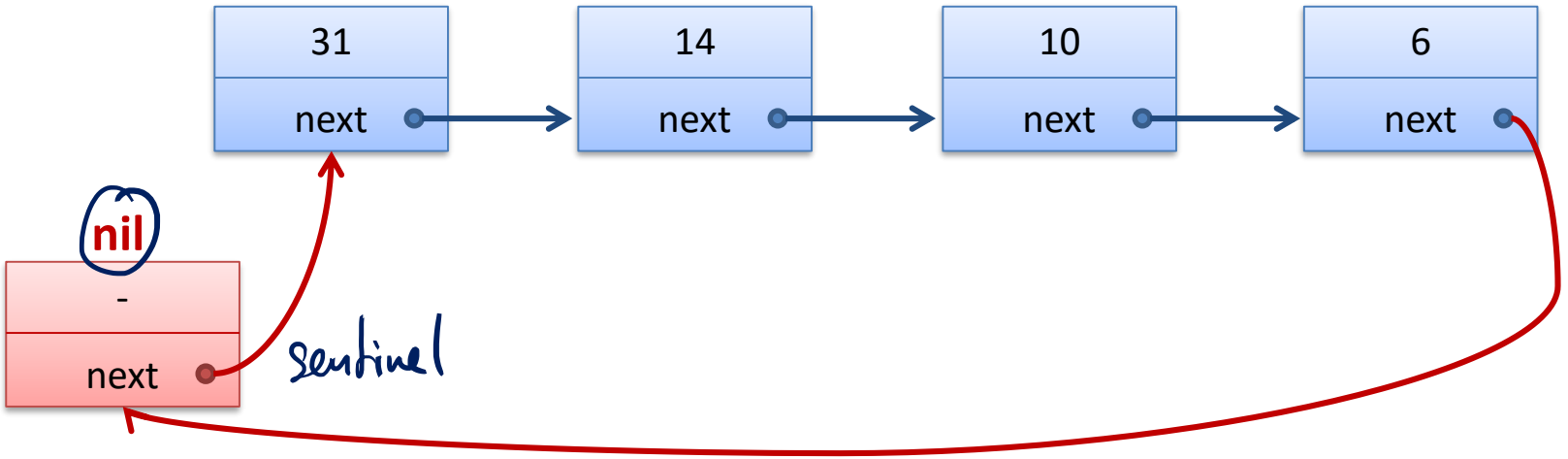



- Alle Operationen in $O(1)$ Zeit
- Grösse nicht beschränkt, Speicherverbrauch $O(n)$

# Listen mit Sentinel

**Sentinel:**

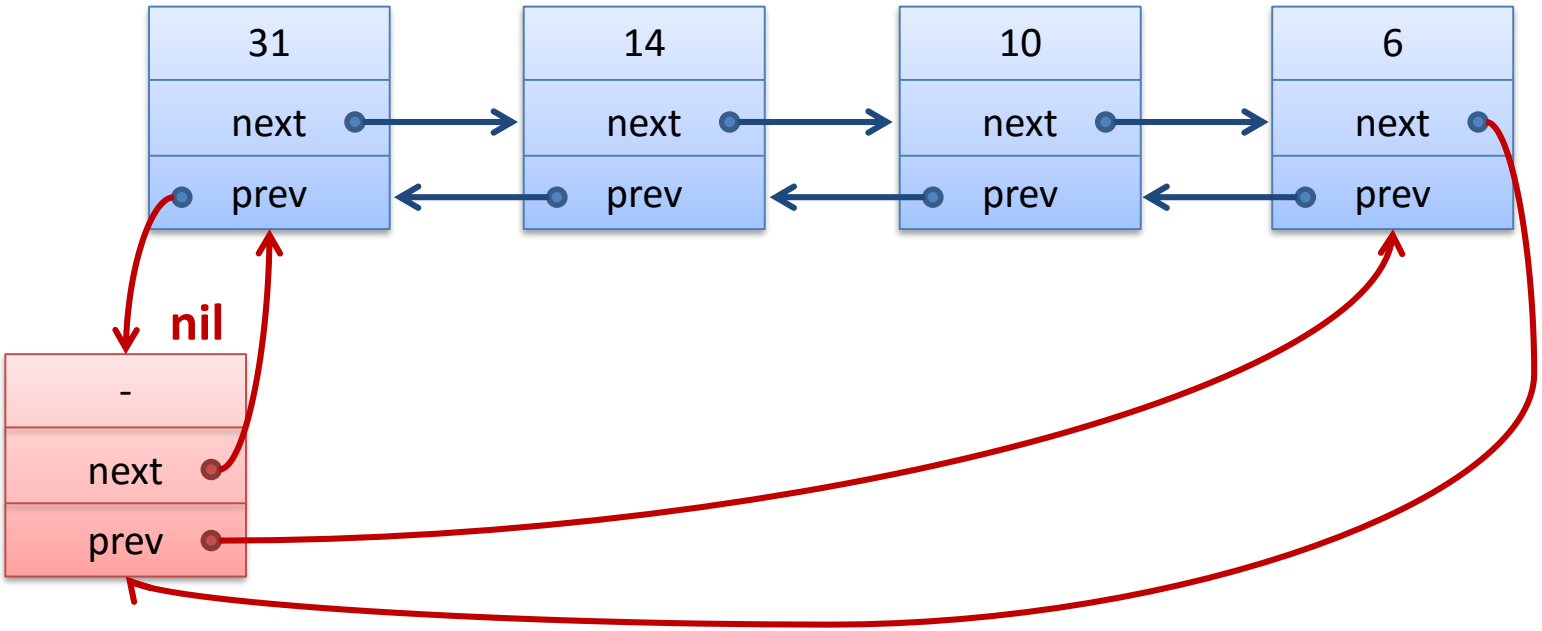
- Ein Dummy-Element, welches Anfang/Ende der Liste bildet




- 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*

# Sentinel : Bemerkungen

UNI FREIBURG

**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++~~ Python
- Ob Sie die Liste mit oder ohne Sentinel programmieren, ist Ihnen überlassen
- Fragestunde am Mi, 11.5. von 14:15-16:00 im 101-00-036

# Bemerkungen zu Pointers in Python / Java / C++

**Java / Python:**

- 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:

**Python:** `le = ListElement(…);`

**Java:** `ListElement le = new ListElement(…);`

- next, prev – Pointers sind einfach vom Typ ListElement
  - Bei der vorgegebenen Python-Struktur über get_next, set_next, get_previous, set_previous 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*
  - Normalerweise über Methoden zugreifbar
    (z.B. getNext, setNext, getPrevious, setPrevious)
  - Da le ein Pointer ist, `le->getNext()` statt `le.getNext()`

# Abstrakte Datentypen : Dictionary

UNI FREIBURG

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

- 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

UNI FREIBURG

- Wir kümmern uns in einer ersten Phase nur um die Basisoperationen *insert, find, delete* (und *create*)

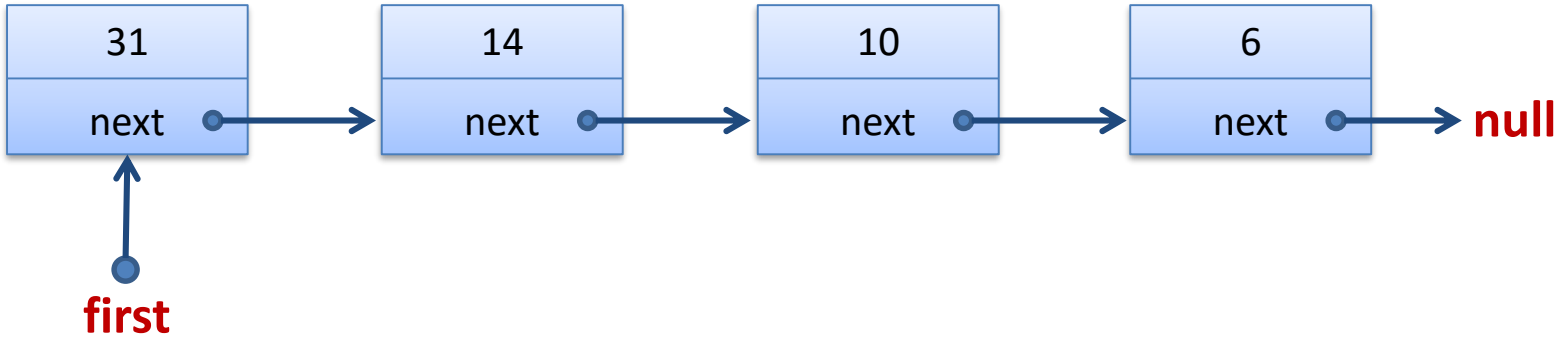
Dictionary Beispiele:

- Wörterbuch (key: Wort, value: Definition / Übersetzung)
- Telefonbuch (key: Name, value: Telefonnummer)
- DNS Server (key: URL, value: IP-Adresse)
- Python Interpreter (key: Variablenname, value: Wert der Variable)
  Java/C++ Compiler (key: Variablenname, value: Typinformation)

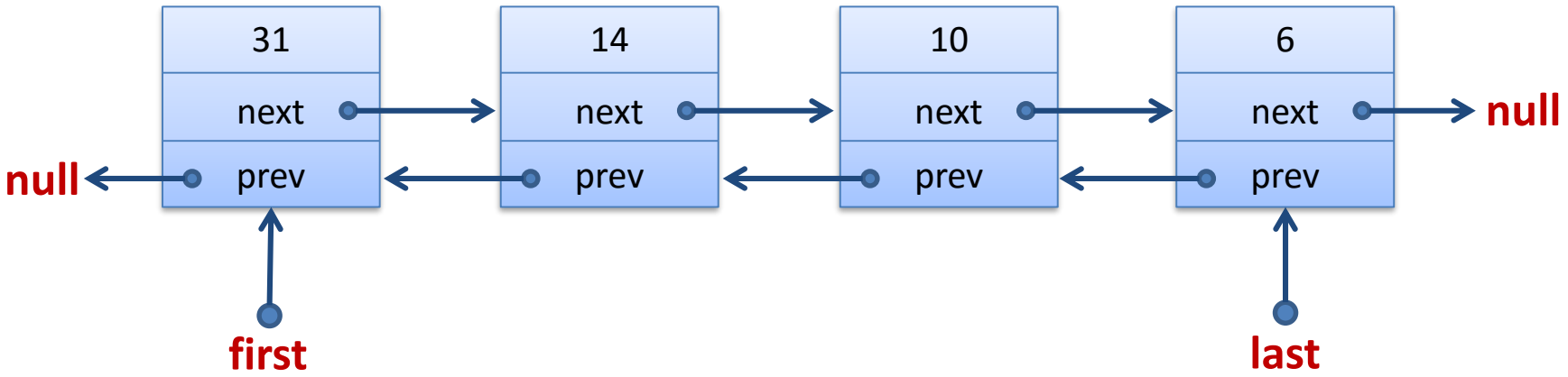
**In all diesen Fällen ist insbesondere eine schnelle *find*-Op. wichig!**

# Verkettete Listen: Struktur

## Einfach verkettete Liste (Singly Linked List):



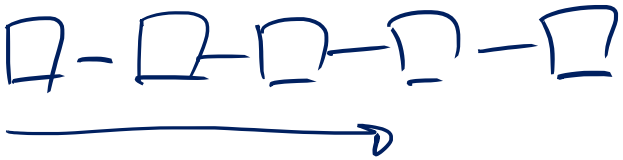

## Doppelt verkettete Liste (Doubly Linked List):



# Dictionary mit verketteten Listen

**Operationen:**

- *create*:
  - lege neue leere Liste an



- *D.insert(key, value)*:
  - füge neues Element vorne ein
  - Annahme: Es gibt noch keinen Eintrag mit dem Schlüssel *key*
- *D.find(key)*:
  - gehe von vorne durch die Liste
- *D.delete(key)*:
  - suche zuerst das Listenelement (wie in *find*)
  - lösche Element dann aus der Liste
  - Bei einfach verketten Listen muss man stoppen, sobald *current.next.key == key* ist!

# Dictionary mit verketteten Listen

**Laufzeiten:**

*create:* $O(1)$

*insert:* $O(1)$

*find:* $O(n)$

*delete:* $O(n)$

Ist das gut?

# Dictionary mit Array

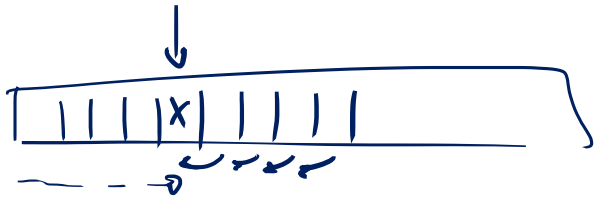
**Operationen:**

- *create*:
  - lege neues Array der Länge *NMAX* an

- *D.insert(key, value)*:
  - füge neues Element hinten an (falls es noch Platz hat)
  - Annahme: Es gibt noch keinen Eintrag mit dem Schlüssel *key*

- *D.find(key)*:
  - gehe von vorne (oder hinten) durch die Elemente

- *D.delete(key)*:
  - suche zuerst nach dem *key*
  - lösche Element dann aus dem Array:



**Man muss alles dahinter um eins nach vorne schieben!**

# Dictionary mit Array

**Laufzeiten:**

*create:* $\Theta(1)$

*insert:* $\Theta(1)$

*find:* $\Theta(n)$

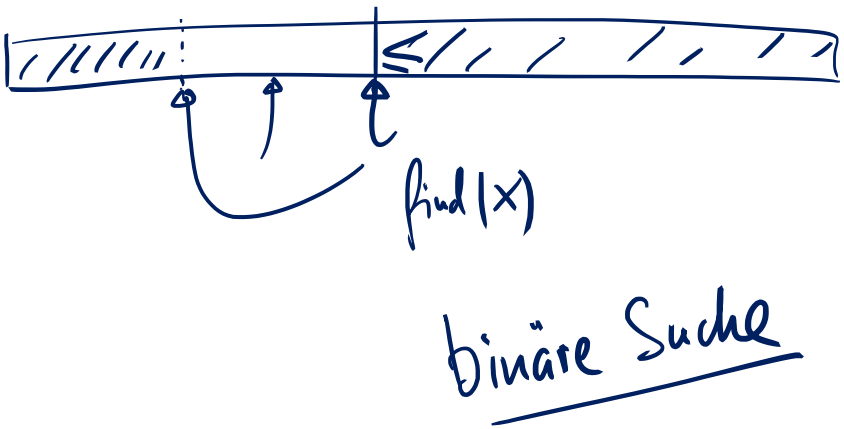
*delete:* $\Theta(n)$

Bessere Ideen?

# Benutze sortiertes Array?

- Teure Operation bei Liste/Array, insbesondere ***find***
- Falls (sobald) sich die Einträge nicht zu sehr ändern, ist *find* die wichtigste Operation!
- Kann man in einem (nach Schlüsseln) sortierten Array schneller nach einem bestimmten Schlüssel suchen?
  - Beispiel: Suche Tel.-Nr. einer Person im Telefonbuch…

**Ideen:**



# Binäre Suche

Benutze Divide and Conquer Idee!

Suche nach der Zahl (dem Key) 19:

| 2 | 3 | 4 | 6 | 9 | 12 | 15 | 16 | 17 | 18 | 19 | 20 | 24 | 27 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

$n$ Elemente

Laufzeit: $O(\log n)$