Algorithmen und Datenstrukturen

Vorlesung 3

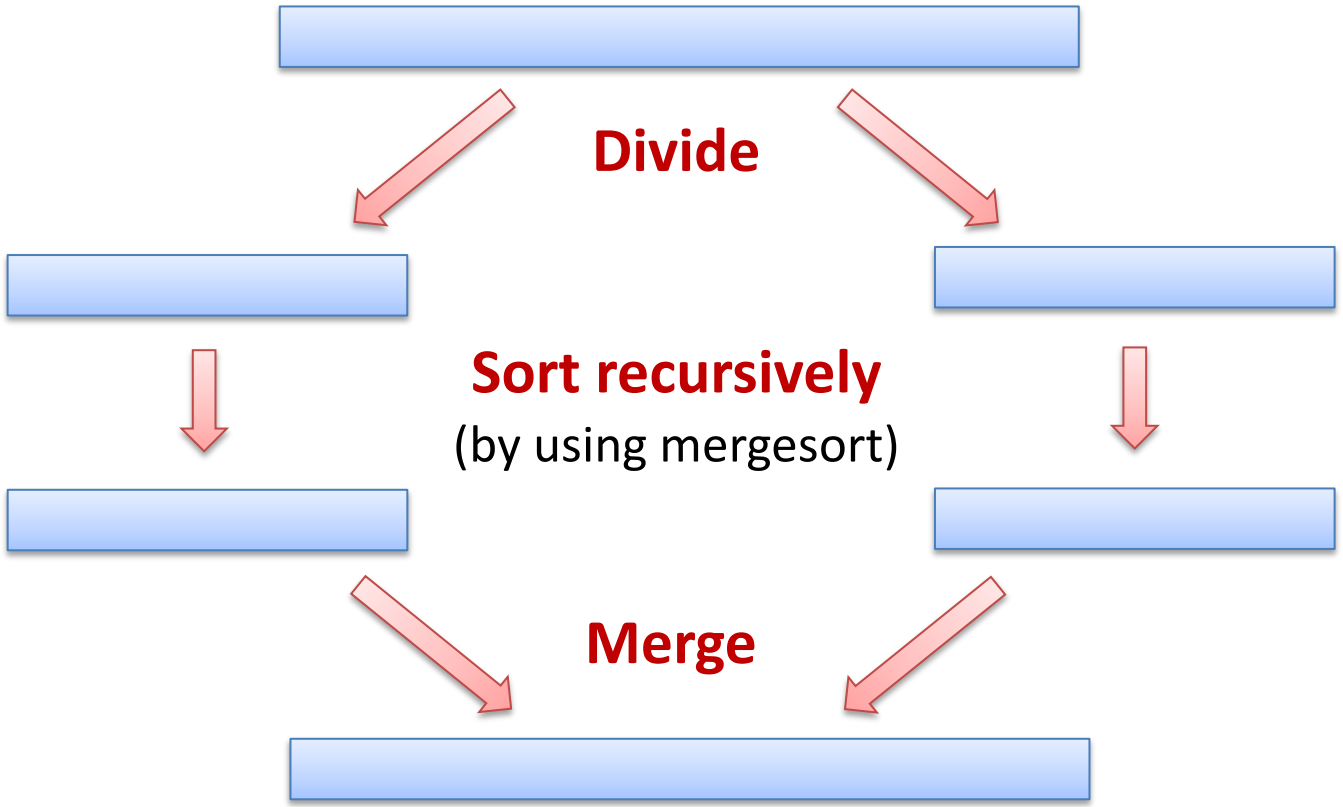
Sortieren III, Abstrakte Datentypen, einfache Datenstrukturen

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Analyse Merge Sort



- Divide ist trivial → Kosten: $O(1)$
- Rekursives Sortieren: Werden wir gleich noch anschauen...
- Merge: Das werden wir uns zuerst anschauen...

Analyse Merge Sort

Laufzeit $T(n)$ setzt sich zusammen aus:

- Divide und Merge: $O(n)$
- 2 rekursive Aufrufe zum Sortieren von $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ Elementen

Rekursive Formulierung von $T(n)$:

- Es gibt eine Konstante $b > 0$, so dass

$$T(n) \leq T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + b \cdot n, \qquad T(1) \leq b$$

- Wir machen uns das Leben ein bisschen einfacher und ignorieren das Auf- und Abrunden:

Annahme: n Zweierpotenz

$$\boldsymbol{T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + b \cdot n, \qquad T(1) \leq b}$$

Zusammenfassung Analyse Merge Sort

UNI FREIBURG

Die Laufzeit von Merge Sort ist $T(n) \in O(n \cdot \log n)$.

- wächst fast linear mit der Grösse der Eingabe…

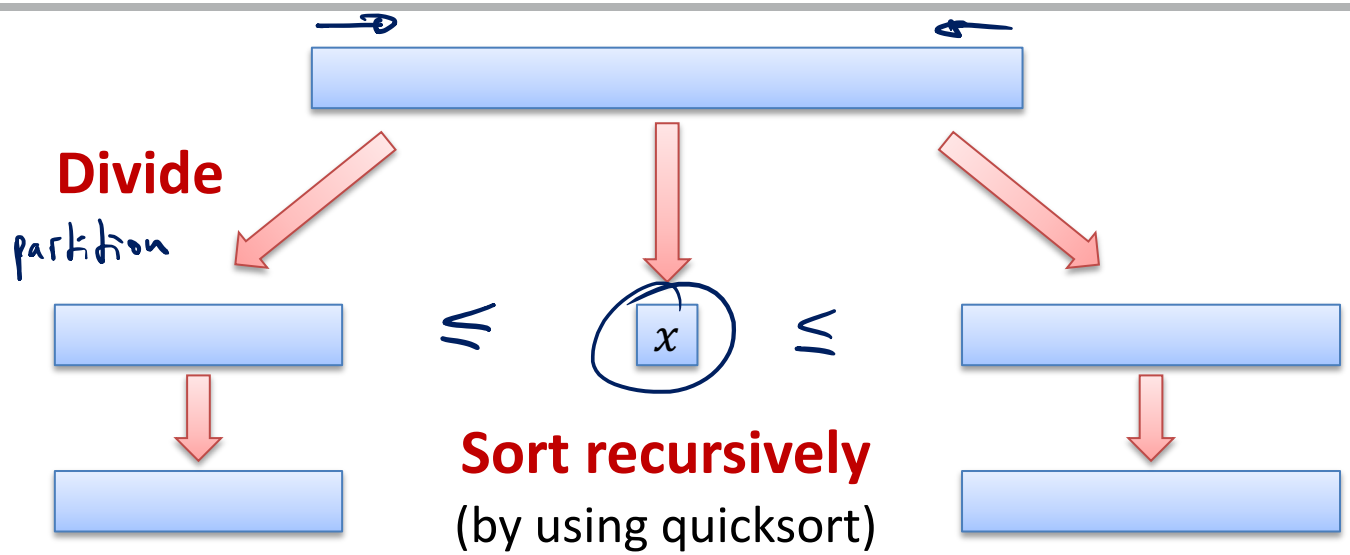
Wie gut ist das?

- Beispielrechnung:
  - Nehmen wir wieder an, 1 Grundop. = 1 ns
  - Wir sind aber ein bisschen konservativer als vorher und nehmen

$$T(n) = 10 \cdot n \log n$$

Eingabegrösse n	4 Bytes p. Zahl	Laufzeit $T(n) = 10 \cdot n \log n$	n^2
$2^{10} \approx 10^3$ Zahlen	$\approx 4\text{KB}$	$10 \cdot 10 \cdot 2^{10} \cdot 10^{-9}\ \text{s} \approx 0.1\ \text{ms}$	$1\ \text{ms}$
$2^{20} \approx 10^6$ Zahlen	$\approx 4\text{MB}$	$10 \cdot 20 \cdot 2^{20} \cdot 10^{-9}\ \text{s} \approx 0.2\ \text{s}$	$16.7\ \text{min}$
$2^{30} \approx 10^9$ Zahlen	$\approx 4\text{GB}$	$10 \cdot 30 \cdot 2^{30} \cdot 10^{-9}\ \text{s} \approx 5.4\ \text{min}$	31.7 Jahre
$2^{40} \approx 10^{12}$ Zahlen	$\approx 4\text{TB}$	$10 \cdot 40 \cdot 2^{40} \cdot 10^{-9}\ \text{s} \approx 122\ \text{h}$	$> 10^7$ Jahre

Quick Sort : Analyse



- Laufzeit hängt davon ab, wie gut die Pivots sind
- Laufzeit, um Array der Länge n zu sortieren, falls das Pivot in Teile der Grösse λn und $(1-\lambda)n$ partitioniert:

$$\boldsymbol{T(n) = T(\lambda n) + T\big((1-\lambda)n\big) + \text{"Pivotsuche + Divide"}}$$

$\mathcal{O}(n)$

- **Divide:**
  - Wir gehen einmal von beiden Seiten über's Array mit konstanten Kosten pro Schritt → Zeit, um Array der Länge n zu partitionieren: $O(n)$

Quick Sort : Analyse

Falls wir in $O(n)$ Zeit ein Pivot finden können, welches das Array in Teile der Grösse λn und $(1-\lambda)n$ unterteilt:

- Es gibt eine Konstante $b > 0$, so dass

$$T(n) \leq T(\lambda n) + T\big((1-\lambda)n\big) + b \cdot n, \qquad T(1) \leq b$$

Extremfall I) $\lambda = 1/2$ (best case):

$$T(n) \leq 2T\left(\frac{n}{2}\right) + bn, \qquad T(1) \leq b$$

- Wie bei Merge Sort: $T(n) \in O(n \log n)$

Extremfall II) $\lambda n = 1$, $(1-\lambda)n = n-1$ (worst case):

$$T(n) = T(n-1) + bn, \qquad T(1) \leq b$$

Quick Sort : Worst Case Analyse

Extremfall II) $\lambda n = 1, (1-\lambda)n = n-1$ (worst case):

$$T(n) = T(n-1) + bn, \qquad T(1) \leq b$$

In dem Fall, ergibt sich $T(n) \in \Theta(n^2)$:

$T(n) \leq T(n-1) + bn$

$\leq T(n-2) + b(n-1) + b(n)$

$\leq T(n-3) + b(n-2 + n-1+n)$

$\vdots$

$\leq T(n-k) + b(n-k+1 + \dots + n)$

$\vdots$

$\leq T(1) + b(2+3+\dots+n)$

$\leq b(1+\dots+n)$

$= b \cdot \frac{n\cdot(n+1)}{2} \in \Theta(n^2)$

Vermutung: $T(n) \leq b\,\frac{n(n+1)}{2}$

Verankerung: $T(1) \leq b\,\frac{1\cdot 2}{2} = b$ ✓

$(n=1)$

Schritt: $(n-1 \to n)$

$T(n) \leq T(n-1) + b\cdot n$

Ind.-vorauss.: $T(n-1) \leq b\,\frac{(n-1)n}{2}$

$\leq b\left(\frac{(n-1)n}{2} + n\right) = b\cdot\frac{n(n+1)}{2}$ ✓

Quick Sort mit zufälligem Pivot

UNI FREIBURG

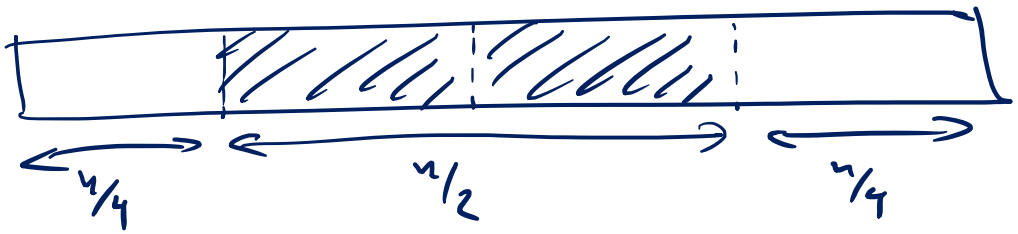
Aufteilung bei zufälligem Pivot:

- Laufzeit $\boldsymbol{T(n) = O(n \log n)}$ für alle Eingaben
  - allerdings nur im Erwartungswert, bzw. mit sehr grosser Wahrscheinlichkeit

Intuition:

- Mit Wahrscheinlichkeit $1/2$, haben die Teile Grösse $\geq n/4$, so dass

$$T(n) \quad \leq \quad T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + bn$$



Quick Sort mit zufälligem Pivot

UNI FREIBURG

Aufteilung bei zufälligem Pivot:

- Laufzeit $\boldsymbol{T(n) = O(n \log n)}$ für alle Eingaben
  - allerdings nur im Erwartungswert, bzw. mit sehr grosser Wahrscheinlichkeit

Analyse:

- Werden wir hier nicht tun
  - siehe z.B. Cormen et al. oder die Algorithmentheorie-Vorlesung
- Mögl. Vorgehen, Rekursion mit Erwartungswerten hinschreiben:

$$\mathbb{E}[T(n)] \leq \mathbb{E}[T(N_L) + T(n - N_L)] + bn$$

Sortieren : Untere Schranke

UNI FREIBURG

Aufgabe: Sortiere Folge $a_1, a_2, \dots, a_n$

- Ziel: benötigte (worst-case) Laufzeit nach unten beschränken

Vergleichsbasierte Sortieralgorithmen

- Vergleiche sind die einzige erlaubte Art, die relative Ordnung von Elementen zu bestimmen
- Das heisst, das Einzige, was die Reihenfolge der Elemente in der sortierten Liste beeinflussen kann, sind Vergleiche der Art

$$a_i = a_j, a_i \leq a_j, a_i < a_j, a_i \geq a_j, a_i > a_j$$

- Nehmen wir an, die Elemente sind paarweise verschieden, dann reichen Vergleiche der Art $a_i \leq a_j$

- 1 solcher Vergleich ist eine Grundoperation

Vergleichsbasierte Sortieralgorithmen

UNI FREIBURG

Alternative Sichtweise

- Jedes Programm (für einen deterministischen, vgl.-basierten Sortieralg.) kann in eine Form gebracht werden, in welcher jede if/while/…-Bedingung von folgender Form ist:

$$\textbf{if } (a_i \leq a_j) \textbf{ then } \ldots$$

- In jeder Ausführung eines Algorithmus, induzieren die Resultate dieser Vergleiche eine Abfolge von T/F (true/false) Werten:

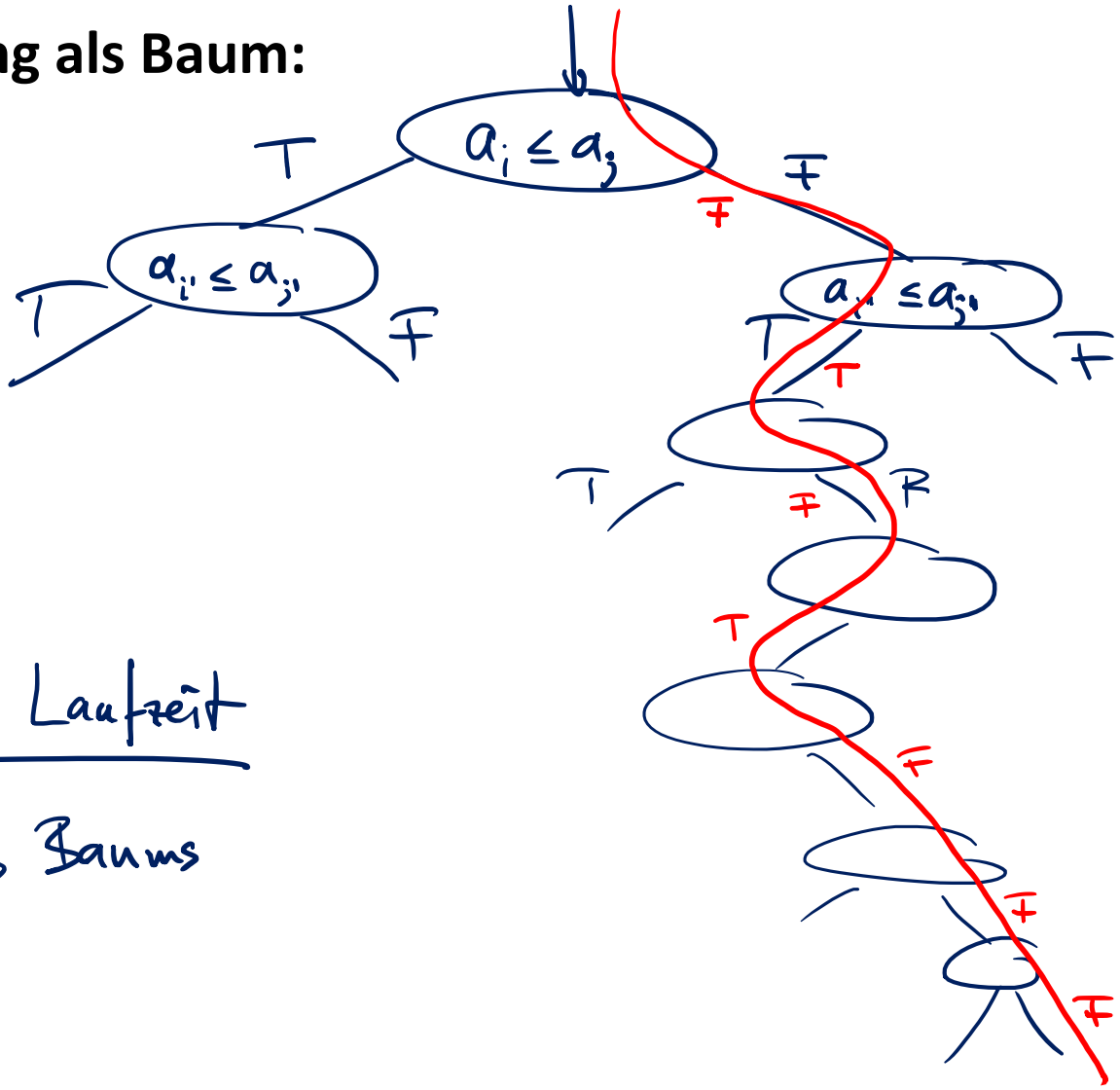
$$\mathbf{TFFTTTFTFFTTFFFFFTFTTT} \ldots$$

- Diese Abfolge bestimmt in eindeutiger Weise, wie die Elemente umgeordnet werden. (für deterministische Algorithmen)

- Unterschiedliche Eingaben der gleichen Werte, müssen daher zu unterschiedlichen T/F-Sequenzen führen!

Vergleichsbasierte Sortieralgorithmen

Ausführung als Baum:



Worst-case Laufzeit

Höhe des Baums

Vgl.-Basiertes Sortieren : Untere Schranke I

- Bei vergleichsbasierten Sortieralgorithmen hängt die Ausführung nur von der Ordnung der Werte in der Eingabe, nicht aber von den eigentlichen Werten ab
  - Wir beschränken und auf Eingaben, bei denen die Werte unterschiedlich sind.
- O.b.d.A. können wir deshalb annehmen, dass wir die Zahlen $1, \dots, n$ sortieren müssen.
- Verschiedene Eingaben müssen verschieden bearbeitet werden.
- Verschiedene Eingaben erzeugen verschiedene T/F-Folgen
- Laufzeit einer Ausführung $\geq$ Länge der erzeugten T/F-Folge
- Worst-Case Laufzeit $\geq$ Länge der längsten T/F-Folge: $\geq$
  - Wir wollen eine untere Schranke
  - Zählen der Anz. mögl. Eingaben → wir benötigen so viele T/F-Folgen...

Vgl.-Basiertes Sortieren : Untere Schranke I

Anzahl Mögliche Eingaben (Anfangsreihenfolgen):

$$n! = n(n-1)(n-2)(n-3)\cdot \ldots \cdot 3\cdot 2\cdot 1$$

$\geq (n/2)^{n/2}$

Anzahl T/F-Folgen der Länge $\leq k$:

Länge $= k$: 2^k — T/F (1) T/F (2) ... T/F (k)

$$2^1 + 2^2 + \ldots + 2^k < 2^{k+1}$$

Theorem: Jeder det. Vergleichs-basierte Sortieralgorithmus benötigt im Worst Case mindestens $\Omega(n \cdot \log n)$ Vergleiche.

Laufzeit $\leq T$

$$2^{T+1} \overset{!}{\geq} n!$$

$$T+1 \geq \log_2(n!)$$

$$T \in \Omega(n \cdot \log n)$$

$$\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$$

$$\frac{n}{2} \cdot \log_2\left(\frac{n}{2}\right) \leq \log_2(n!) \leq n \cdot \log_2 n$$

$$\downarrow$$

$$\log(n!) = \Theta(n \cdot \log n)$$

Sortieren in Linearzeit

UNI FREIBURG

- Mit Vergleichs-basierten Algorithmen nicht möglich
  - Untere Schranke gilt auch mit Randomisierung...

- Manchmal geht's schneller
  - wenn wir etwas über die Art der Eingabe wissen und ausnützen können

- Beispiel: Sortiere n Zahlen $a_i \in \{0,1\}$:
  1. Zähle Anzahl Nullen und Einsen in $O(n)$ Zeit
  2. Schreibe Lösung in Array in $O(n)$ Zeit

Counting Sort

Aufgabe:

- Sortiere Integer-Array A der Länge n
- Wir wissen, dass für alle $i \in \{0, \dots, n-1\}$, $A[i] \in \{0, \dots, k\}$

Algorithmus:

```
1: counts = new int[k+1]          // new int array of length k
2: for i = 0 to k do counts[i] = 0
3: for i = 0 to n-1 do counts[A[i]]++
4: i = 0;
5: for j = 0 to k do
6:    for l = 1 to counts[j] do
7:       A[i] = j; i++
```

$O(k)$

$O(n)$

$O(n+k)$

Zusammenfassung Sortieralgorithmen

UNI FREIBURG

- **Selection Sort, Insertion Sort**
  worst case: $\Theta(n^2)$
- **Merge Sort**
  worst case (und best case): $\Theta(n \log n)$
- **Quick Sort**
  worst case (fixed pivot): $\Theta(n^2)$, average case: $O(n \log n)$
  worst case, randomized: $O(n \log n)$
- **Radix Sort** (für positive ganze Zahlen, siehe Übungen)
  worst case: $O(n \log_n M)$, wobei M die grösste Zahl ist
- **Lower Bound**
  "comparison-based" Algorithmen: $\Omega(n \log n)$

Datenstrukturen

UNI FREIBURG

Algorithmen

- Wie löst man ein gegebenes Problem effizient
- Ziel: möglichst geringe Komplexität
  - kurze Laufzeit / kleiner Speicherverbrauch
  - asymptotisch, abhängig von der Problemgrösse

Datenstrukturen

- Wie können Daten so abgespeichert werden, dass der Zugriff möglichst effizient ist
- Hängt von den Operationen ab, welche unterstützt werden sollen!
- Ermöglicht schnelle Algorithmen
- Benötigt schnelle Algorithmen, um die Operationen optimal auszuführen

Abstrakte Datentypen

UNI FREIBURG

Abstrakter Datentyp:

- Spezifikation, welche Art von Daten verwaltet werden können
- Spezifikation der Operationen, um auf die Daten zuzugreifen
  - inkl. der Semantik der Operation

Datenstruktur:

- Bestimmte Art, einen abstrakten Datentypen zu implementieren
- Je nach Implementierung können die gleichen Operationen verschiedene Laufzeiten (Komplexität) haben.

Wir werden nun zuerst kurz die wichtigsten abstrakten Datentypen diskutieren …

Abstrakte Datentypen : Beispiele

UNI FREIBURG

Array:

- Verwaltet eine Menge von Elementen (des gleichen Typs)

Operationen:

- *create(n)* : erzeugt ein Array der Länge n
- *A.get(i)* : gibt das Element an Position i zurück
- *A.set(x, i)* : schreibt Element x an Position i
- *A.size()* : gibt die Länge des Arrays zurück (nicht immer dabei)

Bei dynamischen Arrays (können Grösse verändern):

- *A.append(x)* : hängt Element x hinten an
- *A.deleteLast()* : löscht letztes Element

Abstrakte Datentypen : Beispiele

UNI FREIBURG

Dictionary: (auch: Maps, assoziative Arrays)

- Verwaltet eine Menge 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

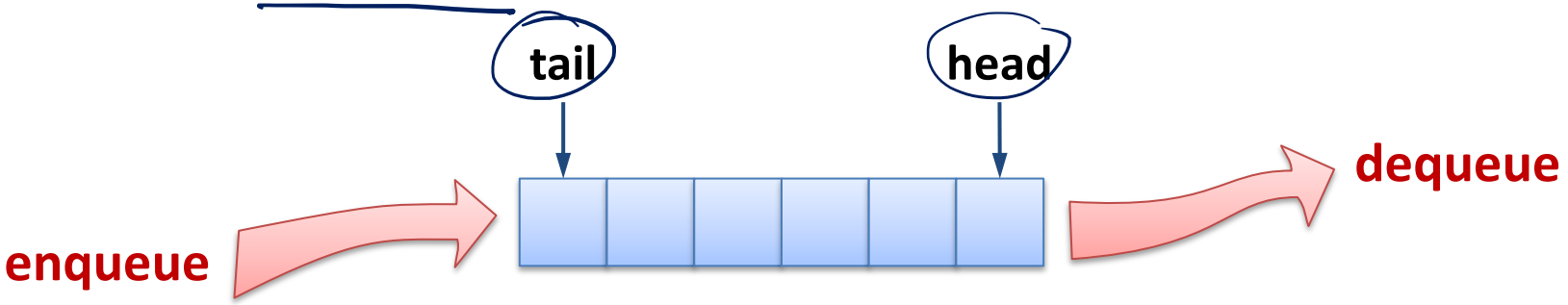
Queue (Warteschlange):

- 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

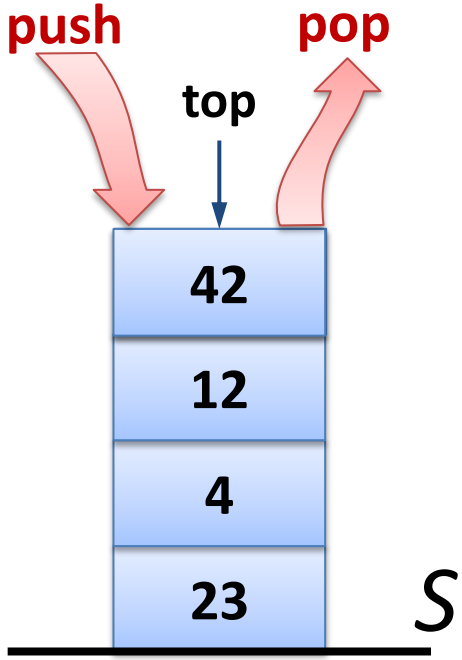
Stack (Stapel):

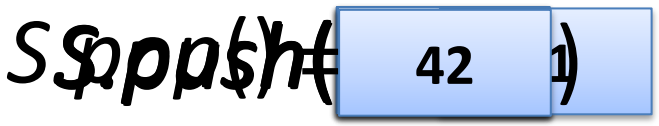
- Verwaltet eine Menge (“Sequenz”) von Werten

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.getMin()* und *H.deleteMin()* müssen konsistent sein
- *H.decreaseKey(x, newkey)* : Falls *newkey* kleiner als der aktuelle Schlüssel von *x* ist, wird der Schlüssel von *x* auf *newkey* gesetzt

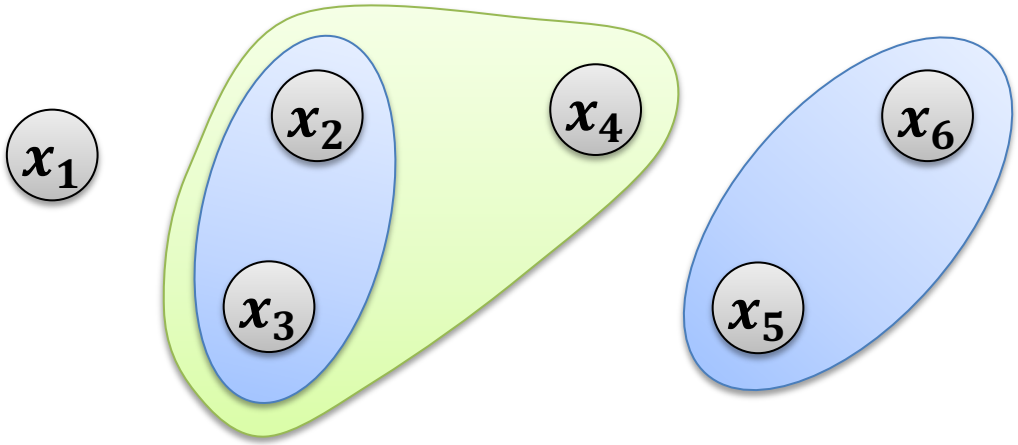
Abstrakte Datentypen : Beispiele

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

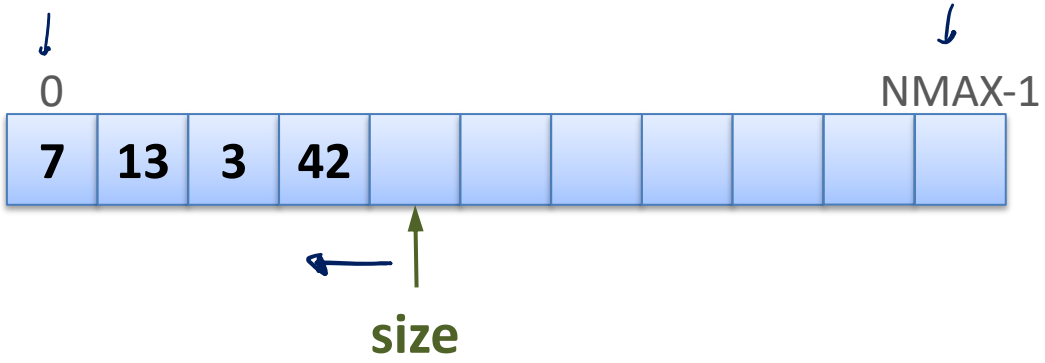
Variablen, um den Zustand des Stack zu speichern:

- $stack$: Array der Länge *NMAX*
- $size$: 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

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) : $\Theta(NMAX)$
  - 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…

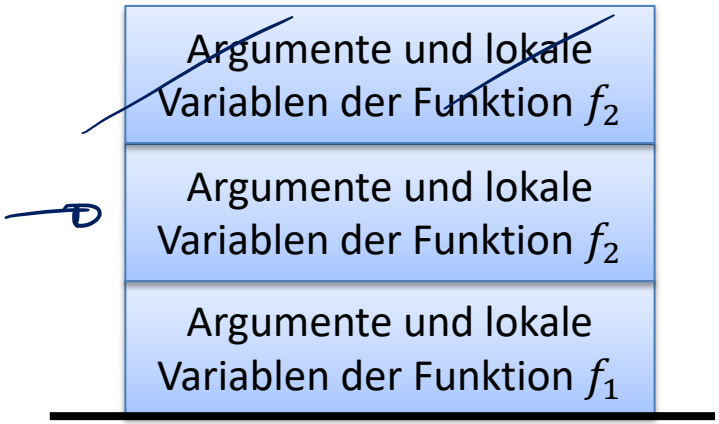
Stack : Anwendungen

- Umdrehen einer Sequenz: A, B, C

  push(A), push(B), push(C), pop() → C, pop() → B, pop() → A

- 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




```
def f1(x,y):
    ...
    f2(z)
    ...
```

```
def f2(a):
    ...
    f2(b)
    ...
```

Array-Implementierung Queue

Versuchen wir den Queue-Datentyp zu implementieren

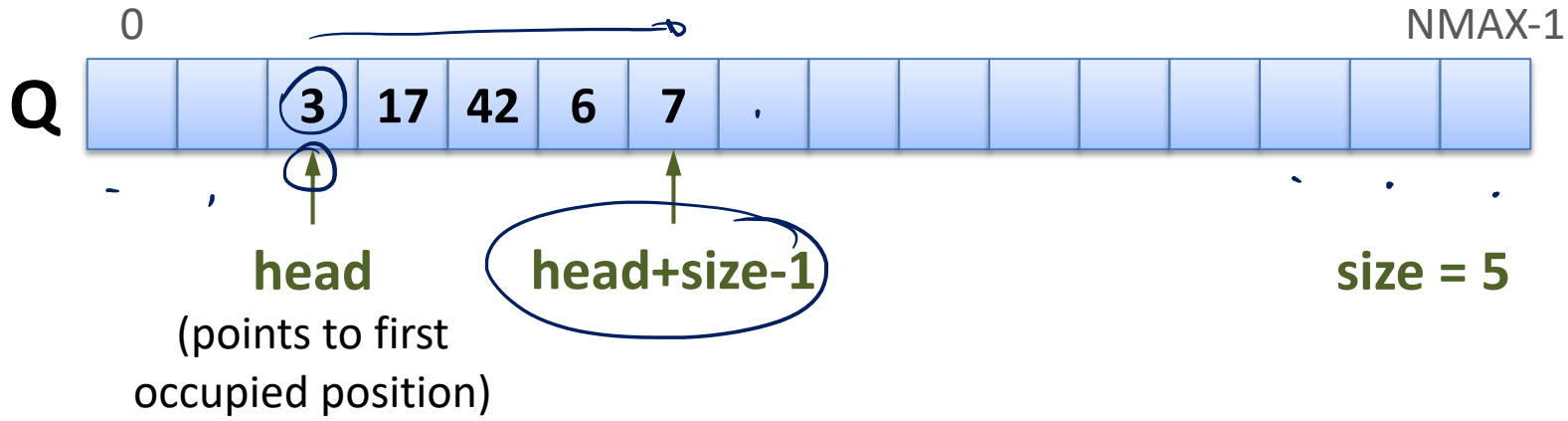
- **Operationen:** *create, enqueue, dequeue, isEmpty*
- **Annahme:** Queue muss nur für *NMAX* Elemente Platz bieten

Variablen, um den Zustand der Queue zu speichern:

- $queue$: Array der Länge *NMAX*
- $head$: Position des vordersten Elements (zyklisch)
  - falls Queue nicht leer ist.
- $size$: Anzahl der Elemente in der Queue

```
create:
  queue = new array of length NMAX
  head  = 0
  size  = 0
```

Array-Implementierung Queue



- Q.dequeue() gibt Element an Pos. head zurück, falls Q nicht leer ist
- Q.enqueue(x) fügt Element x an Pos. head + size ein

- Array wird zyklisch verwendet:

Array-Implementierung Queue

```
S.isEmpty():
  return (size == 0)

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

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

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:

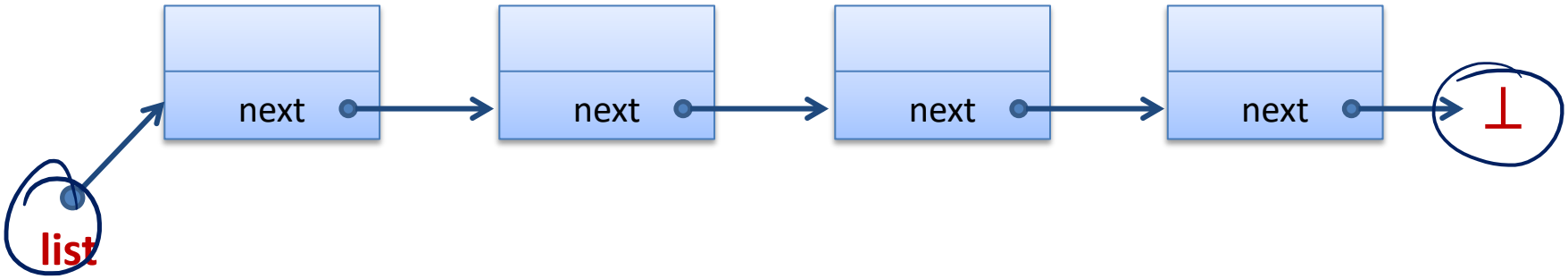
- Speicherverbrauch (space complexity) : $\Theta(NMAX)$
  - man braucht immer gleich viel Speicher, egal wie viele Elemente in der Queue gespeichert sind!
- Die Queue kann nur *NMAX* Elemente aufnehmen…
- Wir werden gleich sehen, wie man beides beheben kann…

Verkettete Listen (Linked Lists)

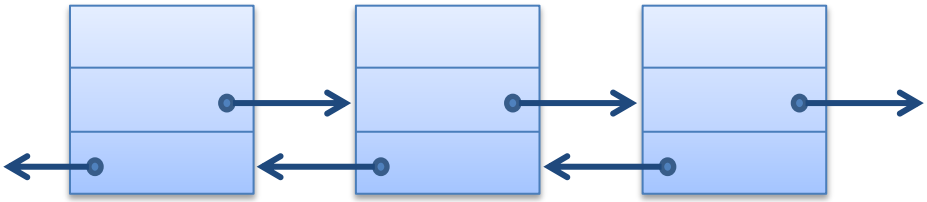
- Datenstruktur, um eine Liste (Sequenz) von Werten zu verwalten

Array:

Verkettete Liste:

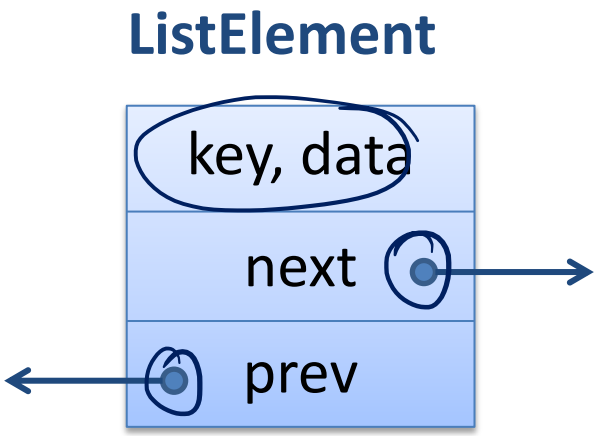


Doppelt verkettete Liste:



Listenelemente

- Klasse, um Listenelemente zu beschreiben

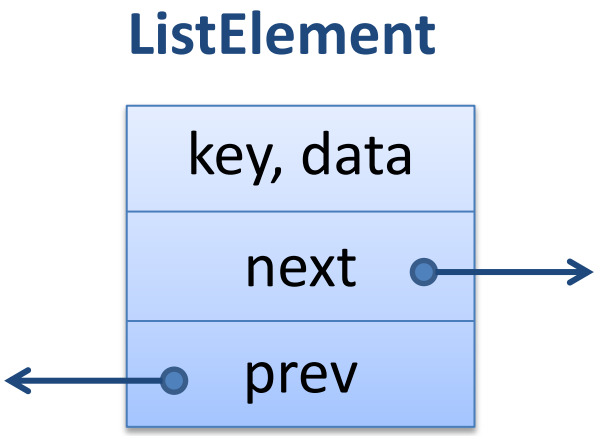


Python:

```
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:

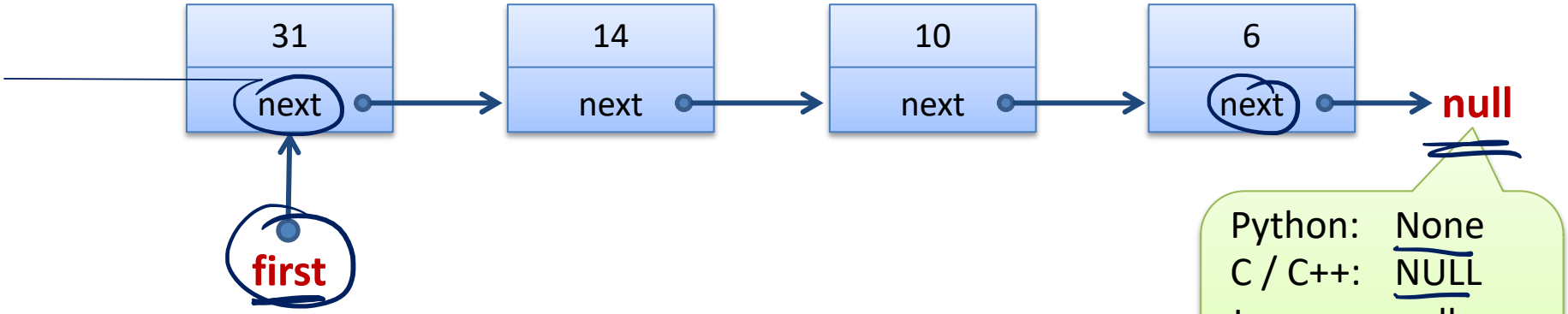
```
public class ListElement {

  int/String/… key;
  Object/… data;

  ListElement next;
  ListElement prev;

}
```

C++:

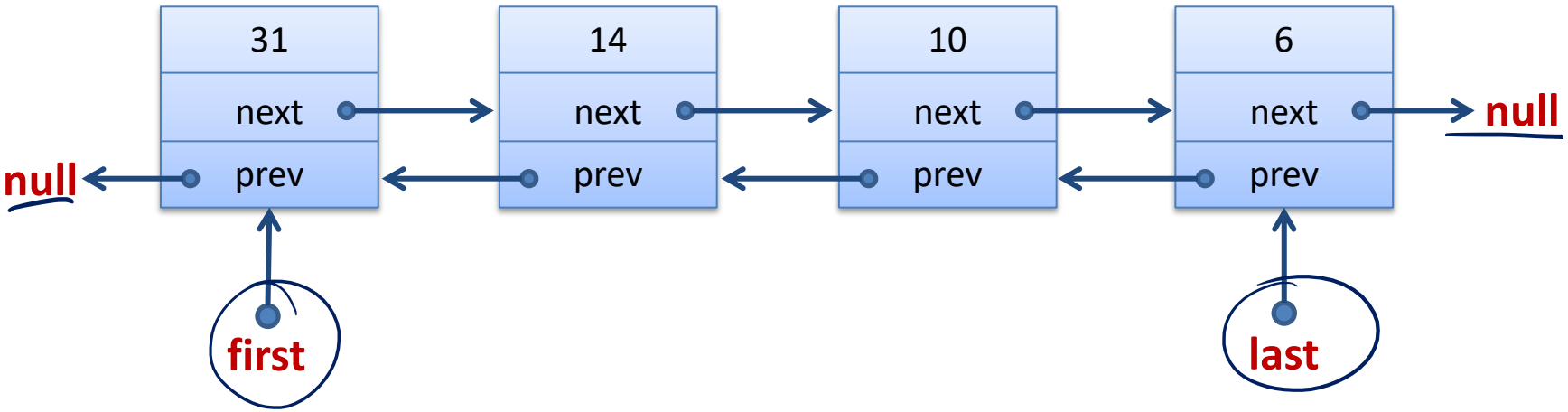
```
class ListElement {
public/private:

  int/… key;
  void*/… data;

  ListElement* next;
  ListElement* prev;

}
```

Verkettete Listen: Struktur

Einfach verkettete Liste (Singly Linked List):



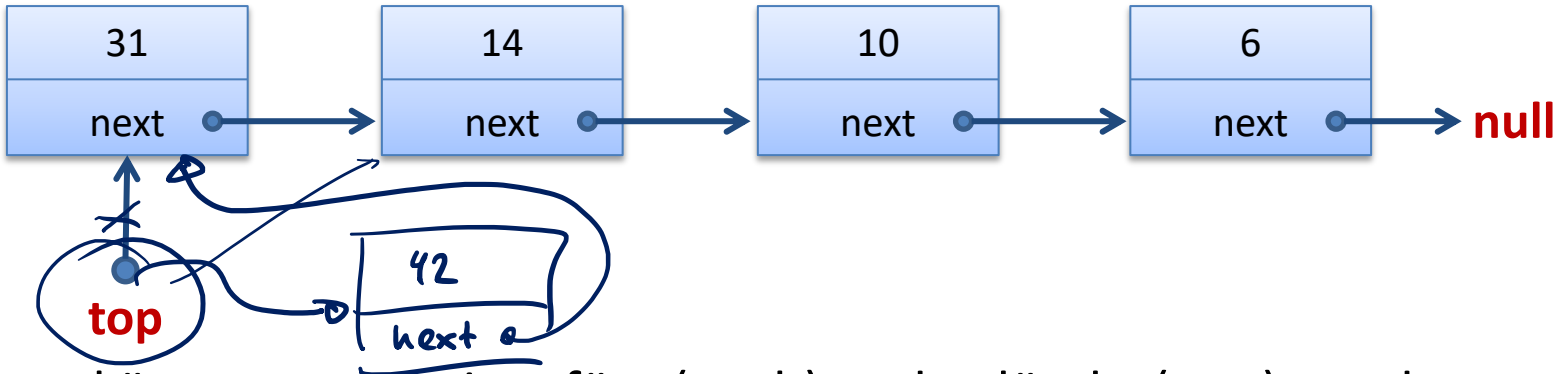
Doppelt verkettete Liste (Doubly Linked List):



Stack und FIFO Queue mit Listen

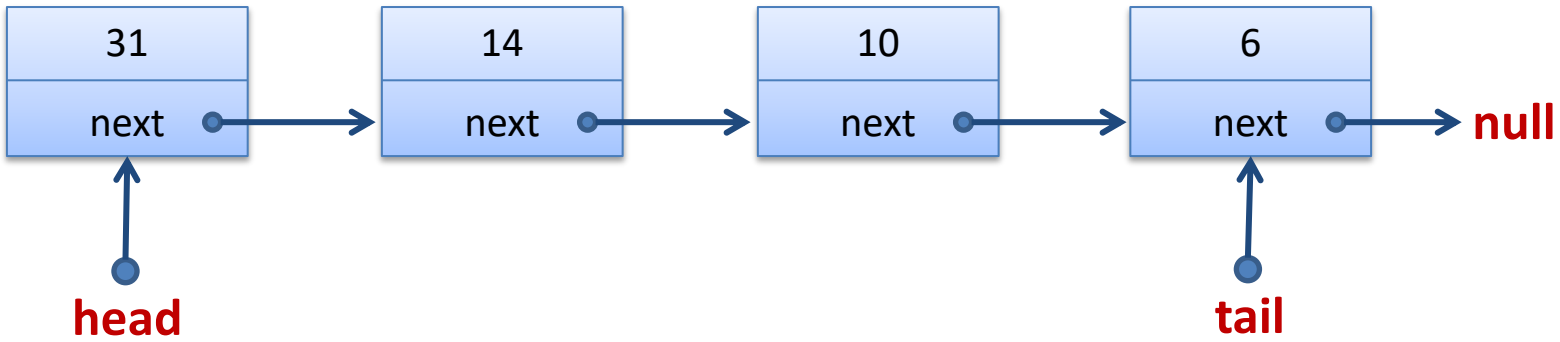
Mit einfach verketteten Listen, alle Operationen in $O(1)$ Zeit

Stack:



- Elemente können vorne eingefügt (push) und gelöscht (pop) werden

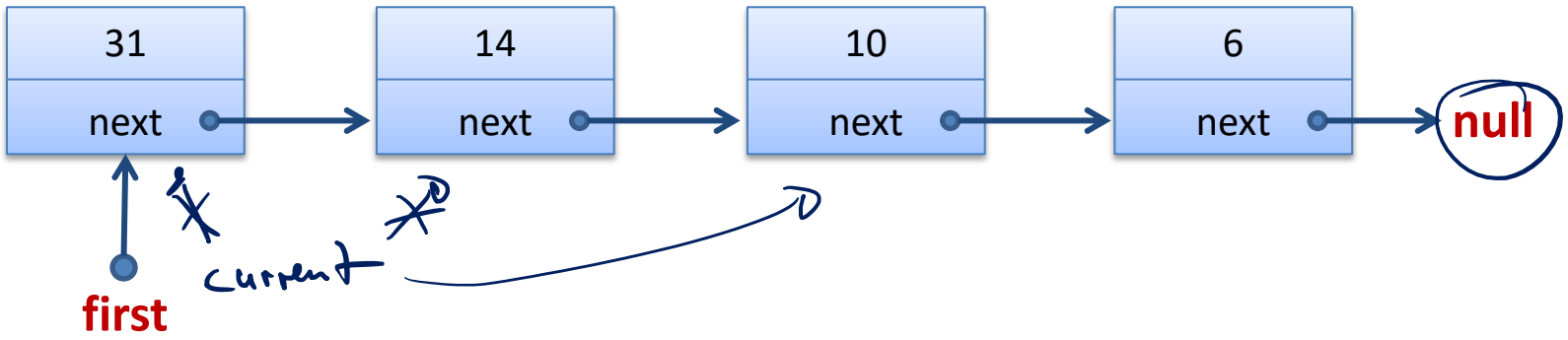
Queue:



- enqueue: füge am Ende der Liste (tail) ein neues Element ein
- dequeue: lösche Element am Anfang der Liste (head)

Suchen in verketteten Listen

Einfach verkettete Liste (Singly Linked List):



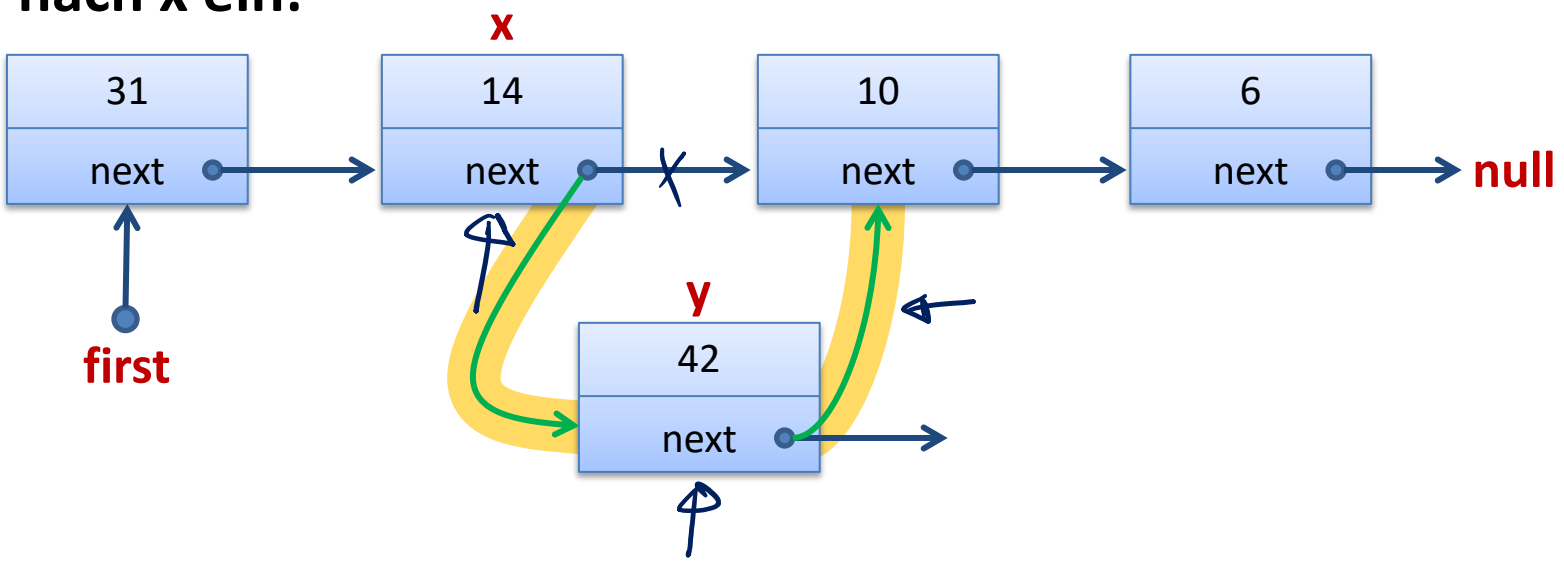
Ziel: Finde Element mit Schlüssel x

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

Laufzeit: Liste der Länge n : $O(n)$

Einfügen in einfach verketteten Listen

Füge y nach x ein:

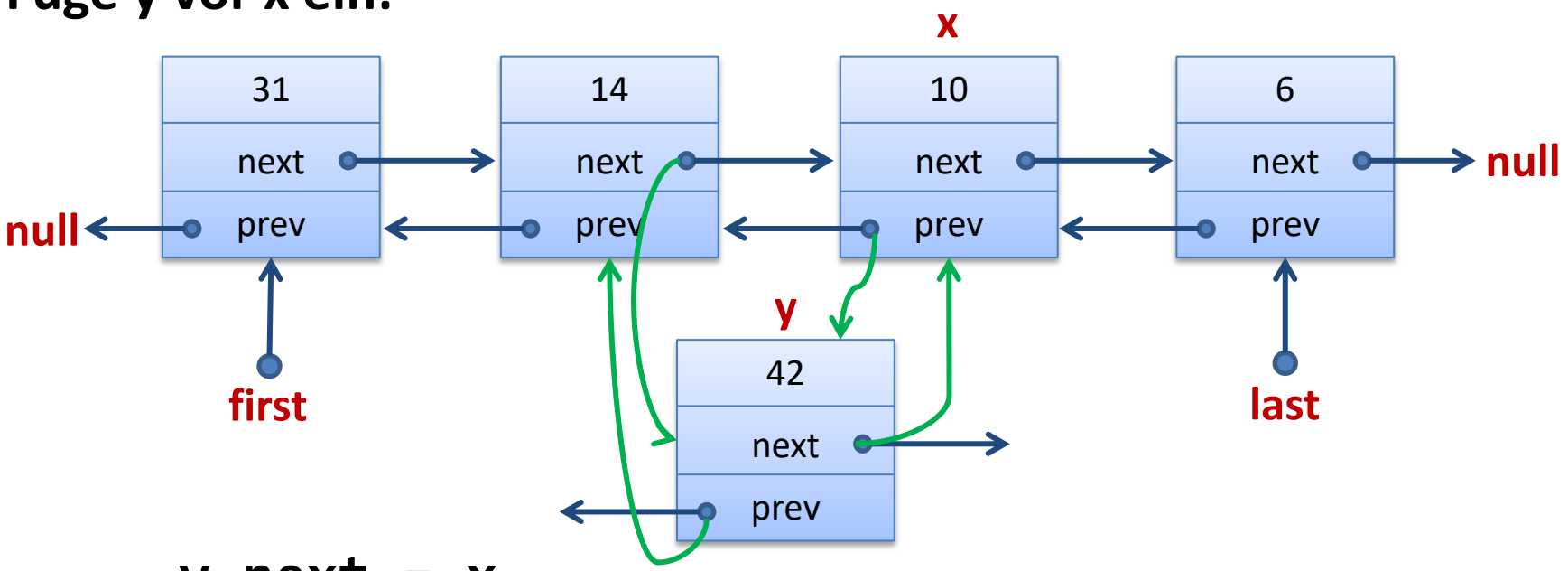



```
y.next = x.next
x.next = y
```

Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Einfügen in doppelt verketteten Listen

Füge y vor x ein:

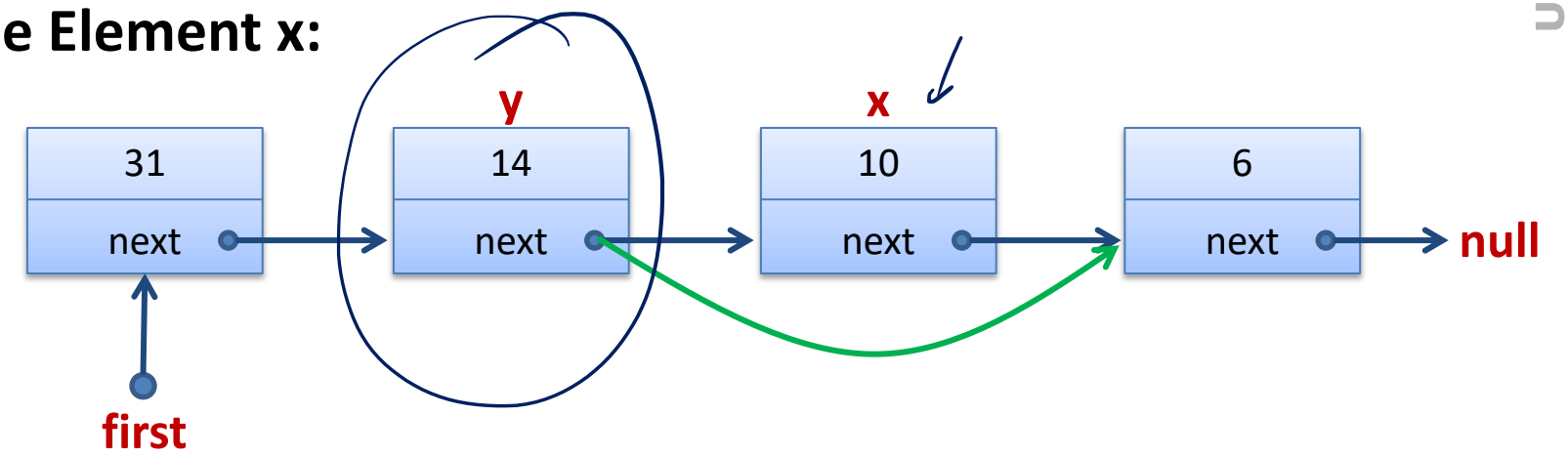



```
y.next = x
y.prev = x.prev
x.prev.next = y
x.prev = y
```

Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Löschen in einfach verketteten Listen

Lösche Element x:



Annahme: Vorgängerelement y ist gegeben

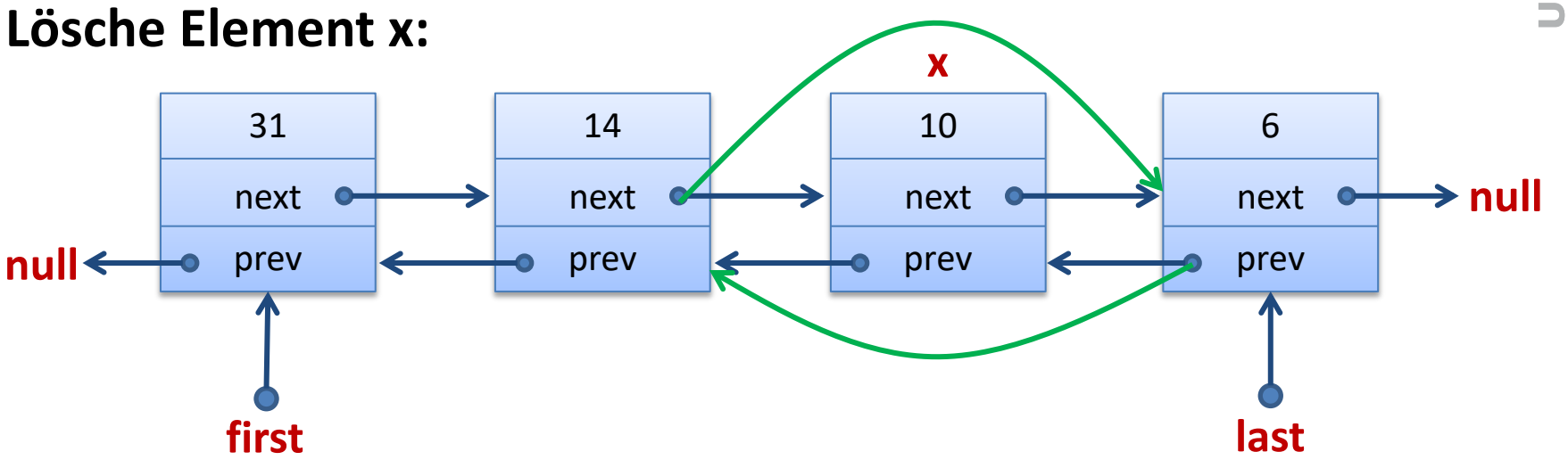
```
y.next = x.next
```

- Bei C++ müsste man jetzt den Speicher von Element x noch freigeben, bei Python / Java macht das der Garbage Collector

Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Löschen in doppelt verketteten Listen

Lösche Element x:




```
x.prev.next = x.next
x.next.prev = x.prev
```

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: $O(1)$

- Falls Ref. auf Vorgänger gegeben, sonst $O(n)$

Löschen eines Elements: $O(1)$

- Falls Ref. auf Vorgänger (einfach verk. Listen) oder Element (doppelt verk. Listen) gegeben, sonst $O(n)$

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

- Falls last-Pointer auf erste Liste gegeben

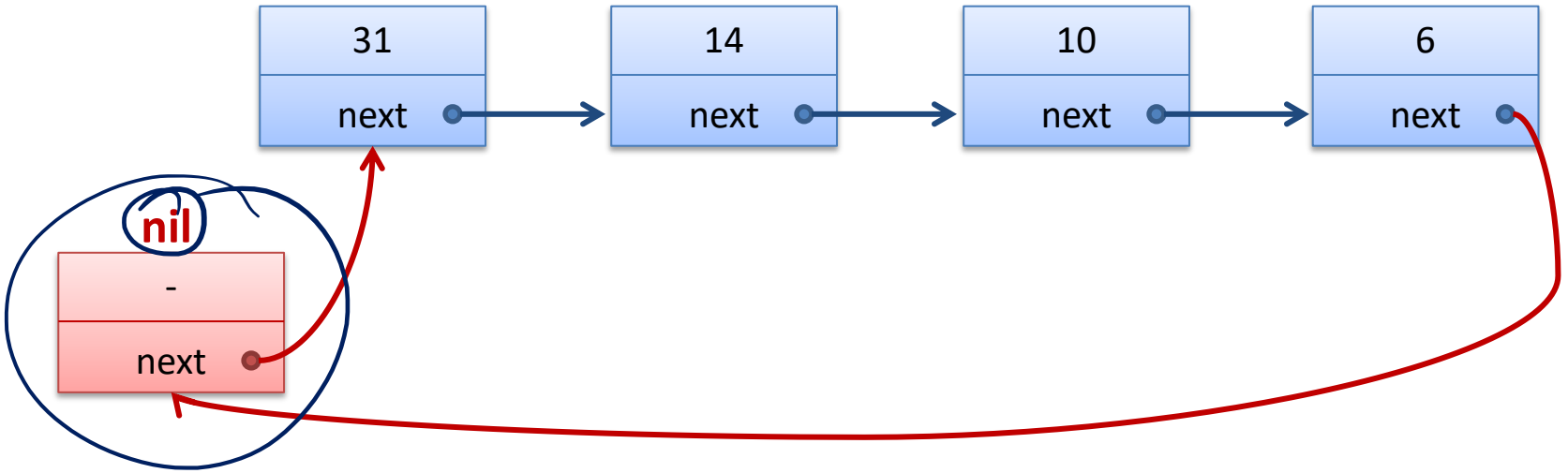
Stack und Queue mit verketteten Listen:

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

Listen mit Sentinel

Sentinel:

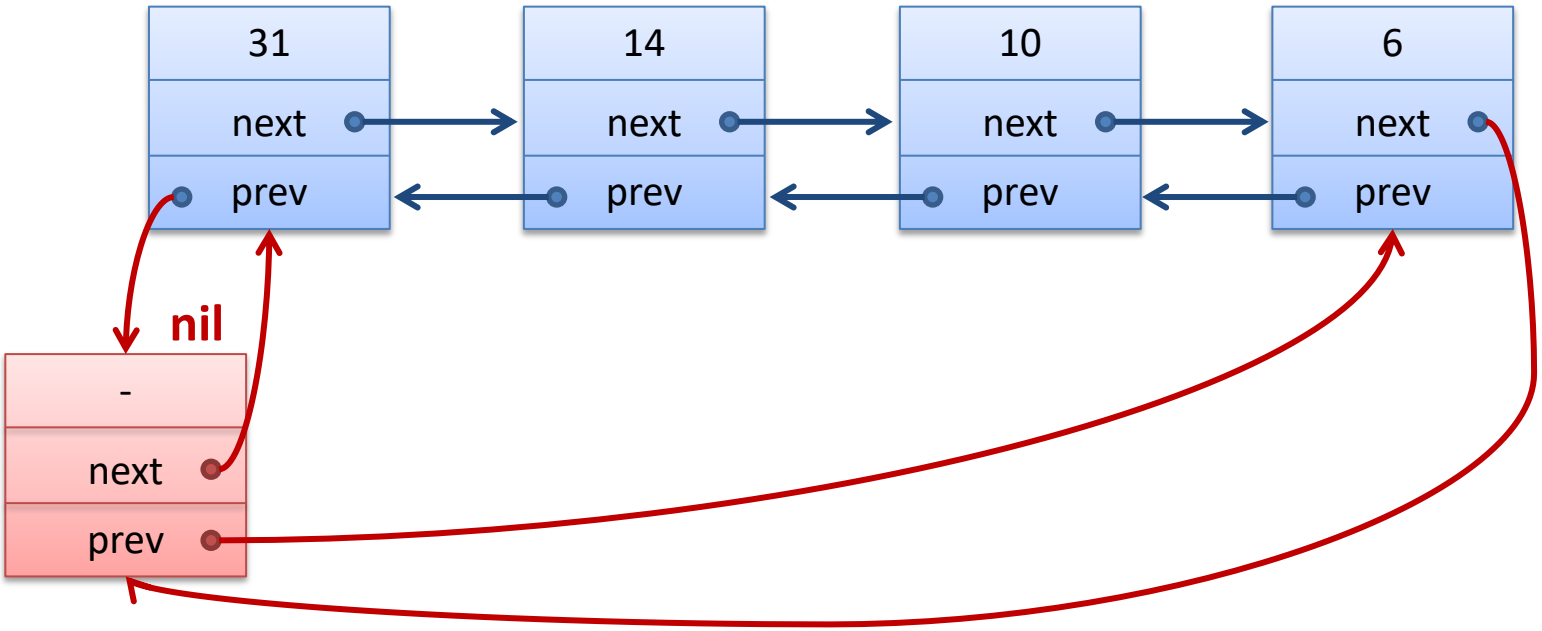
- Ein Dummy-Element, welches Anfang/Ende der Liste bildet



- 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 vor allem da Sinn, wo sie den Code wirklich vereinfachen