

Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 5 (2.5.2018)

Abstrakte Datentypen



**UNI
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

Erfahrungen 1. Übung

- Ein paar einzelne haben noch Probleme mit SVN, daphne, flake8, Makefiles, etc. gebraucht
 - Die Übung war unter anderem dazu da, dass Sie sich mit all den Tools auseinandersetzen und “anfreunden” können
- Zum Teil im falschen Verzeichnis, z.T. fehlt das Makefile
 - Unbedingt testen, ob Jenkins fehlerfrei durchläuft
- Durchschnittliche Bearbeitungszeit: die meisten $< 10h$
 - zum Teil 2-5h, zum Teil $> 10h$
 - Einige hatten zu wenig Zeit (Abgabezeit nicht klar)
 - ca 9-10 Stunden pro Übungsblatt sind normal
 - **Aber:** Wenden Sie sich bei Problemen frühzeitig ans Forum!
- Zeitmessung und Schaubilder waren zeitaufwendig
 - Sehe ich ein, das war vermutlich eine einmalige Sache

Probleme bei der Implementierung von Quicksort

- Das richtige Abbruchkriterium beim Divide zu finden war tricky

Probleme mit Rekursion

- Probleme mit Template (wegen Rekursion)

Problem mit Python

- RuntimeError: maximum recursion depth exceeded in comparison
 - Trat bei Quicksort auf, wenn man das erste Element als Pivot nimmt und das Array absteigend vorsortiert ist.
 - Lösung:

```
import sys
sys.setrecursionlimit(10000)
```

Hinweise zu Daphne, etc. auf Webseite:

http://ac.informatik.uni-freiburg.de/teaching/ss_18/info2/InfoUebungen.php

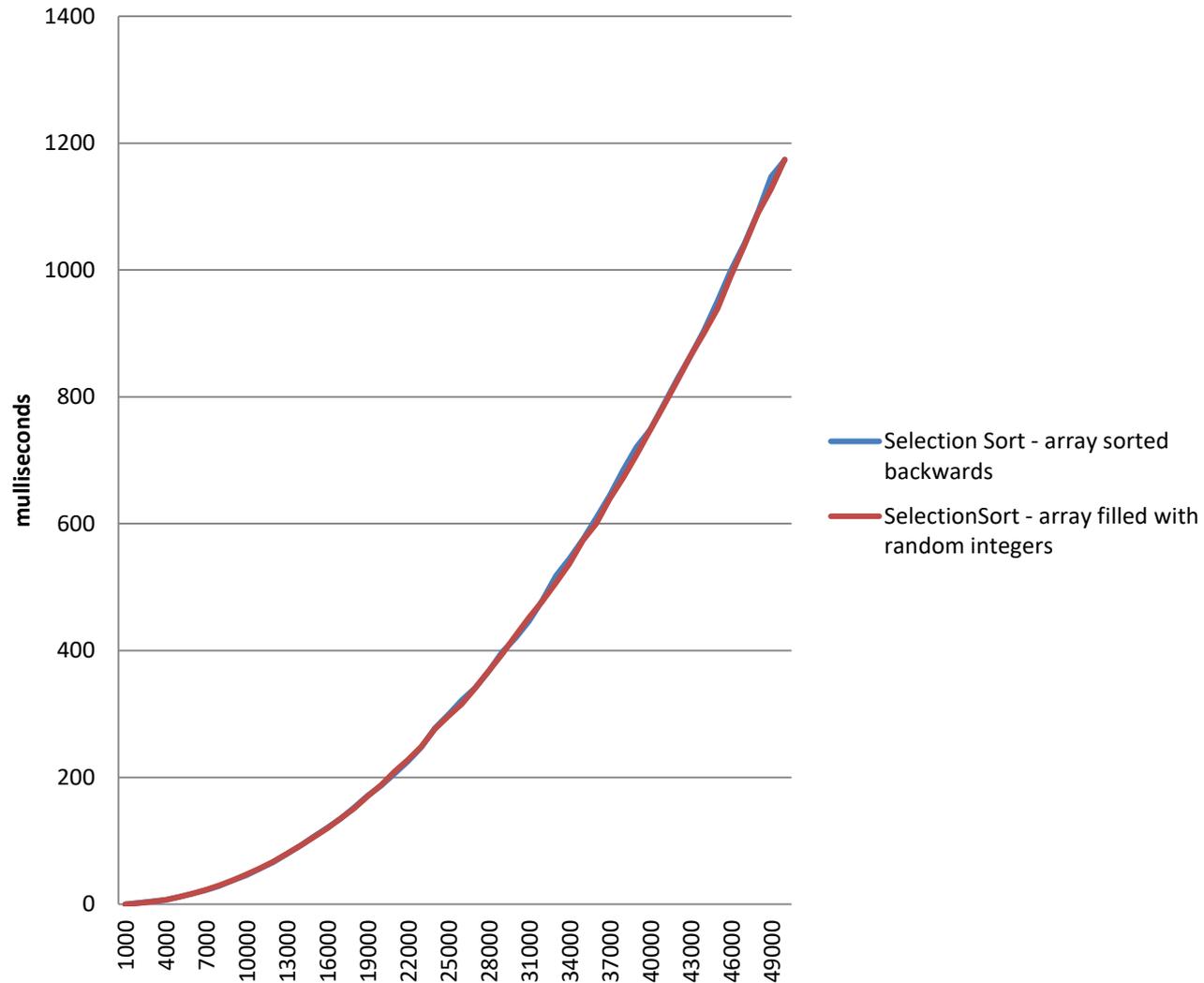
- Einfachste Lösung: Vorlage aus der Vorlesung übernehmen und anpassen
 - Makefile / build.xml einfach übernehmen und jedesmal verwenden
 - Für Java: build.properties jeweils kurz anpassen
 - Für Übung 1 dann einfach Quicksort-Code hinzufügen und Main-File anpassen
 - Wir werden im public-Folder eine Musterlösung bereitstellen:
Auch wenn Sie die Aufgabe nicht gelöst haben, versuchen Sie die Vorlesungsbeispiele und die Musterlösung aus dem SVN (public-Folder) auszuchecken und bei sich auszuprobieren

- Übung 2 (online) ist rein theoretisch...
- Übung 3 wird mind. zur Hälfte praktisch sein...

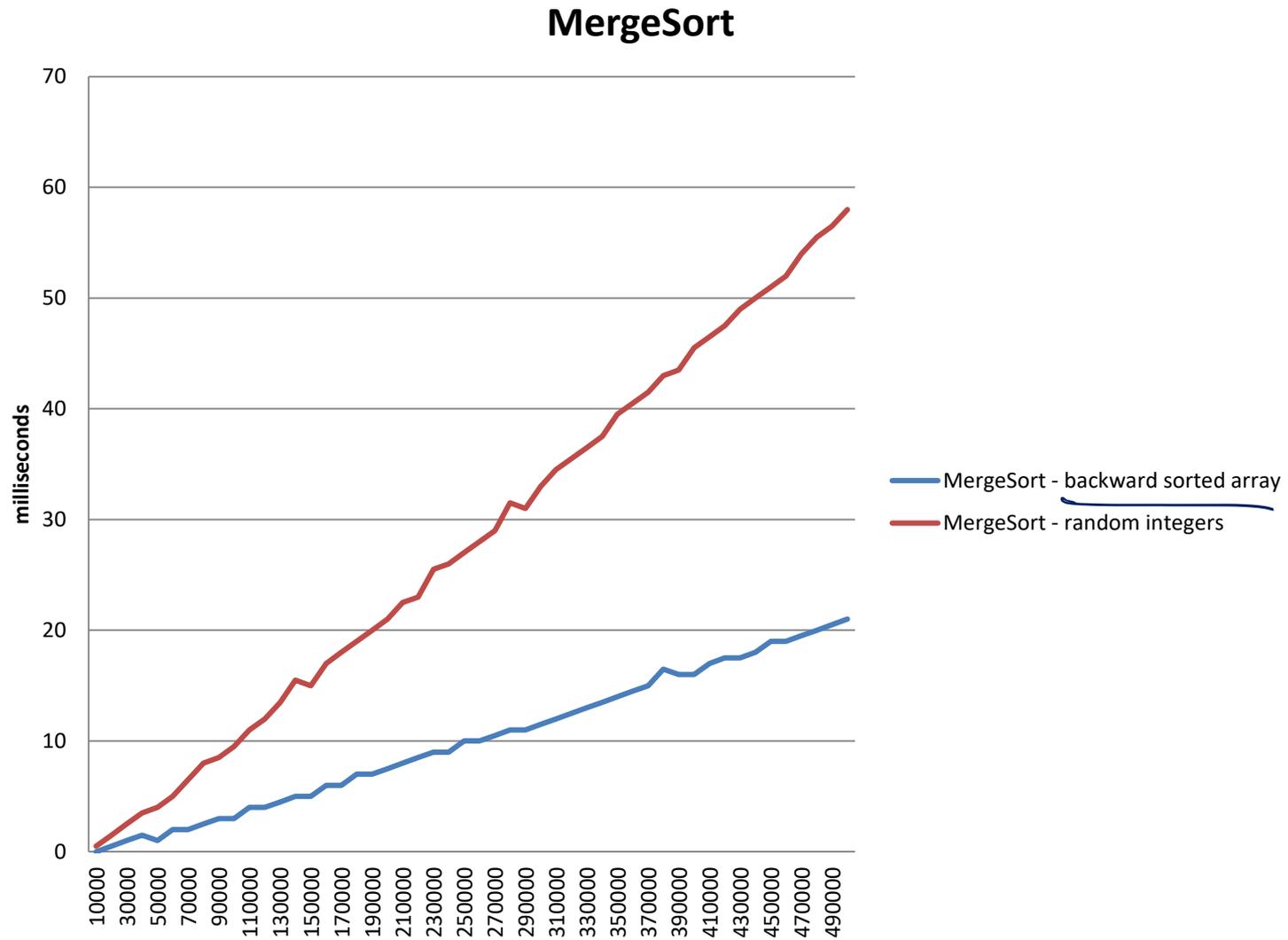
Weitere Programmierübungen:

- Wir werden versuchen, jeweils Python-Grundgerüst zu geben, welches Sie dann entsprechend ergänzen können
- Es ist normal, dass die Dinge nicht immer auf Anhieb funktionieren...
 - Verschwenden Sie allerdings nicht zu viel Zeit, ohne Fortschritt zu machen, sondern wenden Sie sich **frühzeitig** mit Ihren Fragen ans Forum
 - Wenn möglich nicht erst am Sonntag Abend / Nacht!

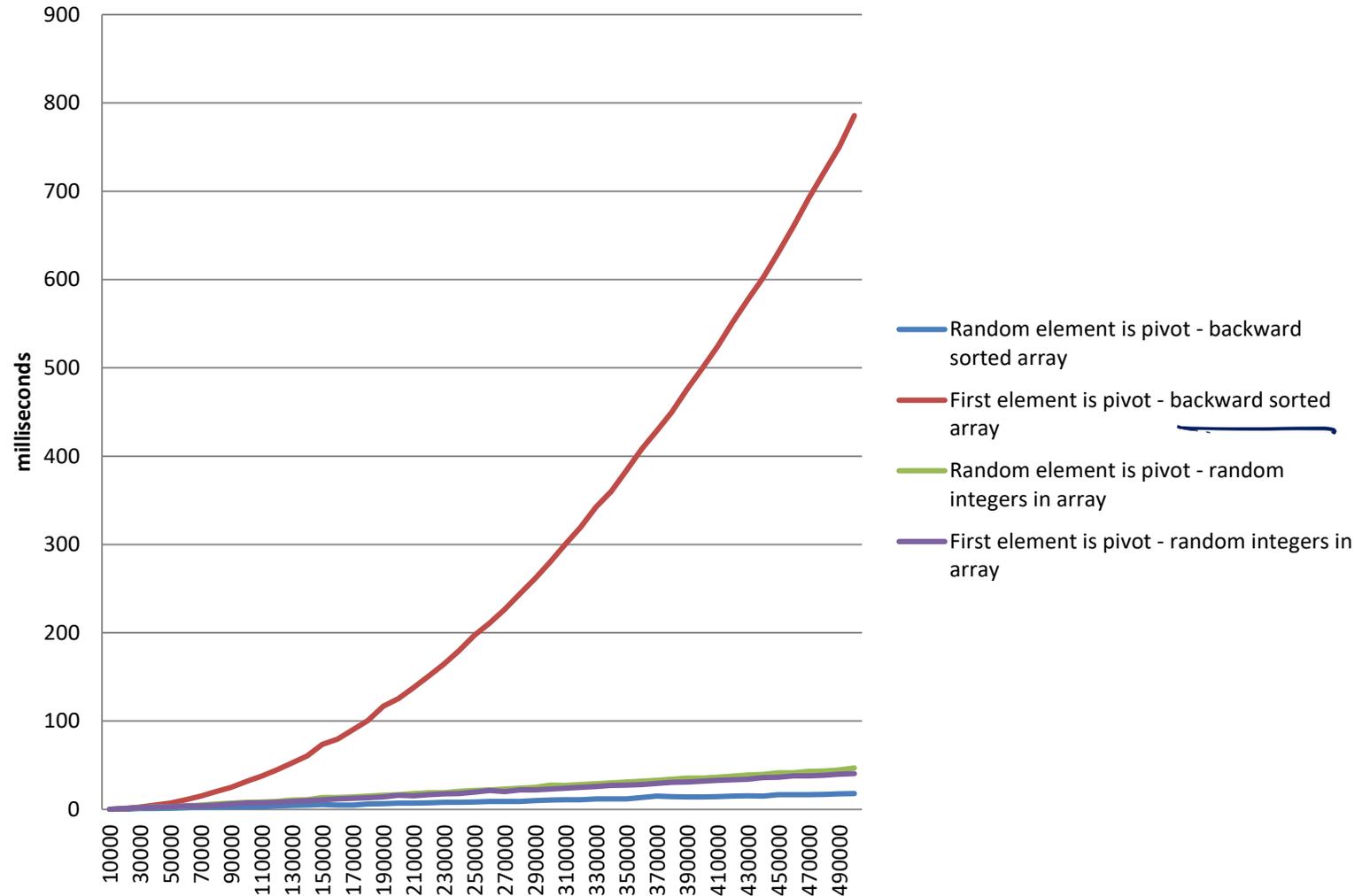
Messungen Selection Sort



Messungen Merge Sort

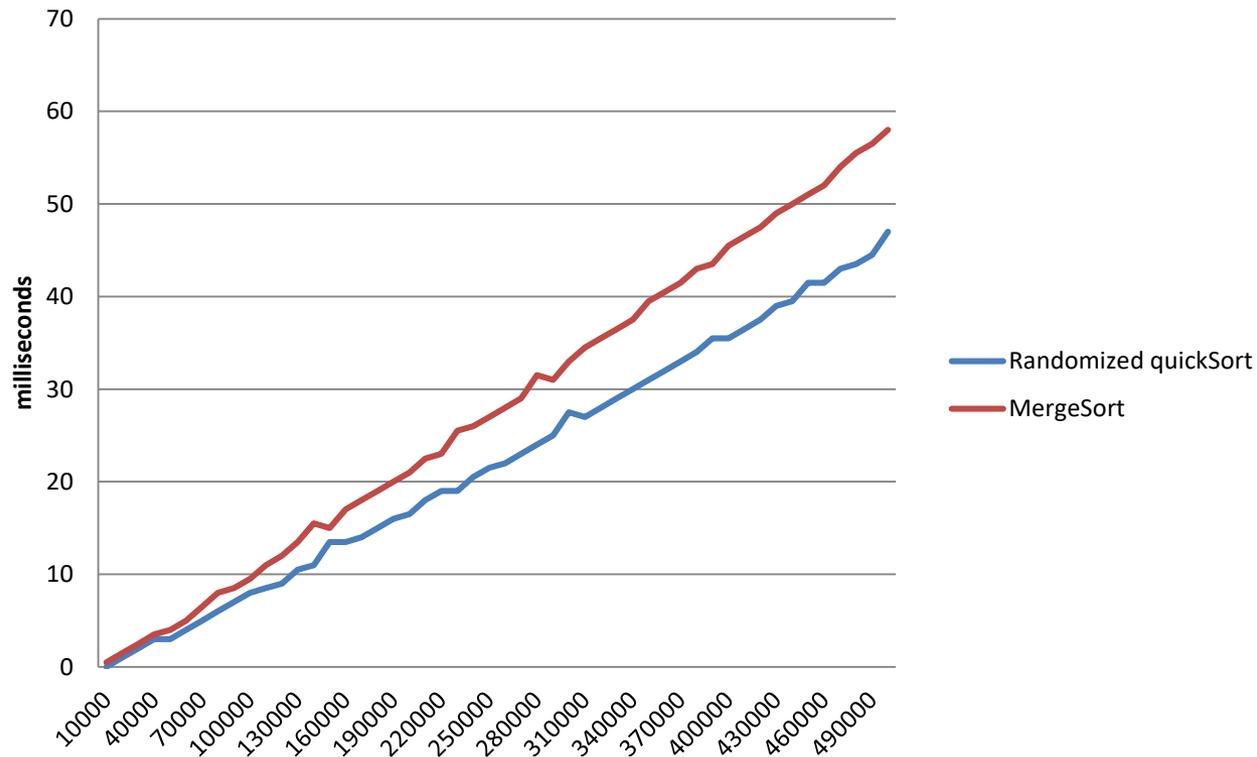


QuickSort



Messungen – Merge Sort vs. Quick Sort

QuickSort vs MergeSort
Array of random integers



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

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.

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

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

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

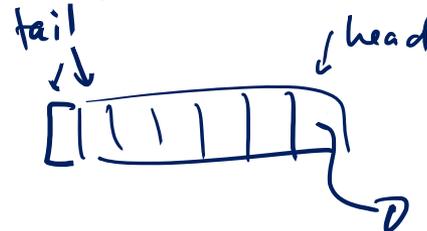
Queue (Warteschlange):

- 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)

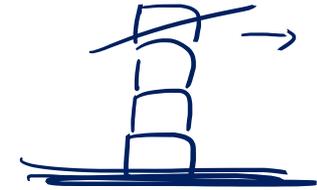


Stack (Stapel):

- 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:

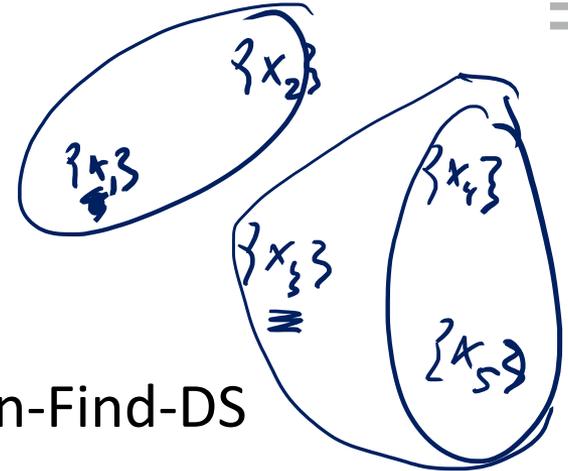
- *create* : erzeugt einen leeren Heap
- *H.insert(x, key)* : fügt Element *x* mit Schlüssel *key* ein
- *H.getMin()* : gibt ^{ein} Element mit kleinstem Schlüssel zurück ← konsistent
- *H.deleteMin()* : löscht Element mit kleinstem Schlüssel
- *H.decreaseKey(x, newkey)* : Falls *newkey* kleiner als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

Union-Find / Disjoint Sets:

- Verwaltet eine Partition von Elementen

Operationen:

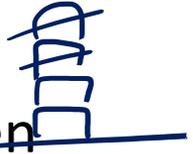
- *create* : erzeugt eine leere Union-Find-DS
- *U.makeSet(x)* : fügt Menge $\{x\}$ zur Partition hinzu
- *U.find(x)* : gibt Menge mit Element x zurück
- *U.union(S1, S2)* : vereinigt die Mengen $S1$ und $S2$



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:

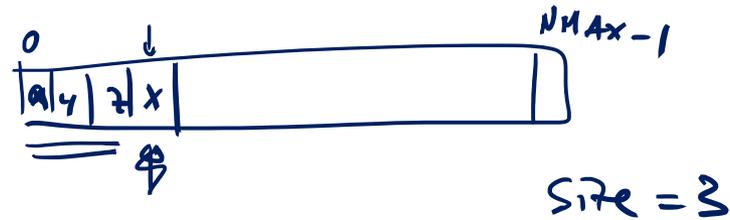
```
stack = new array of length NMAX
```

```
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]
```

Laufzeit (Zeitkomplexität) der Operationen:

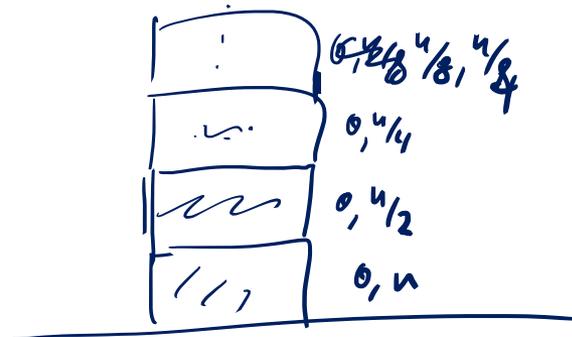
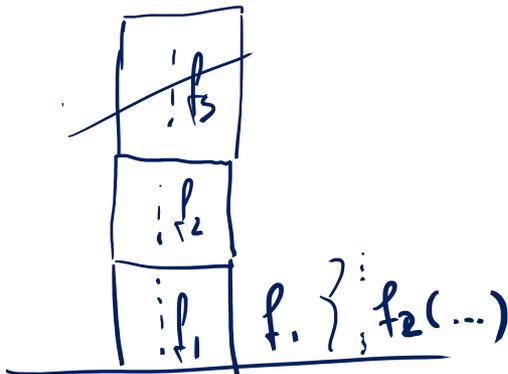
- create: $O(1)$
 - falls man davon ausgeht, dass Speicher in $O(1)$ Zeit alloziert werden kann
- push: $O(1)$
- pop: $O(1)$
- isEmpty: $O(1)$

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: A, B, C
 $push(A), push(B), push(C), pop() \rightarrow C, pop() \rightarrow B, pop() \rightarrow A$
- 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



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)
```

Stack

Einträge:

[l, r, sorted]

False: Anfang

True: nach rek. Aufrufen

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~~ ^{Queue} zu speichern:

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

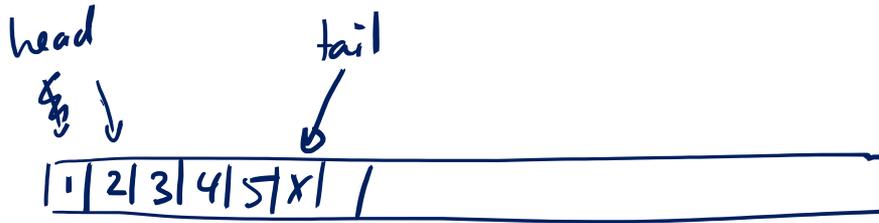
create:

queue = new array of length $NMAX$

head = 0

tail = 0

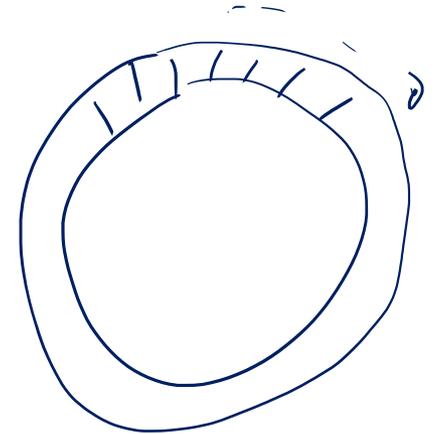
Array-Implementierung Queue



dequeue: move head



empty \iff head = tail



Array-Implementierung Queue

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

↙ ∈ {0, ..., NMAX - 1}

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

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

Laufzeit (Zeitkomplexität) der Operationen:

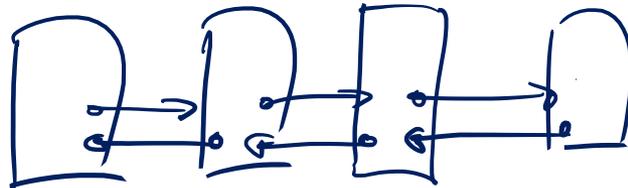
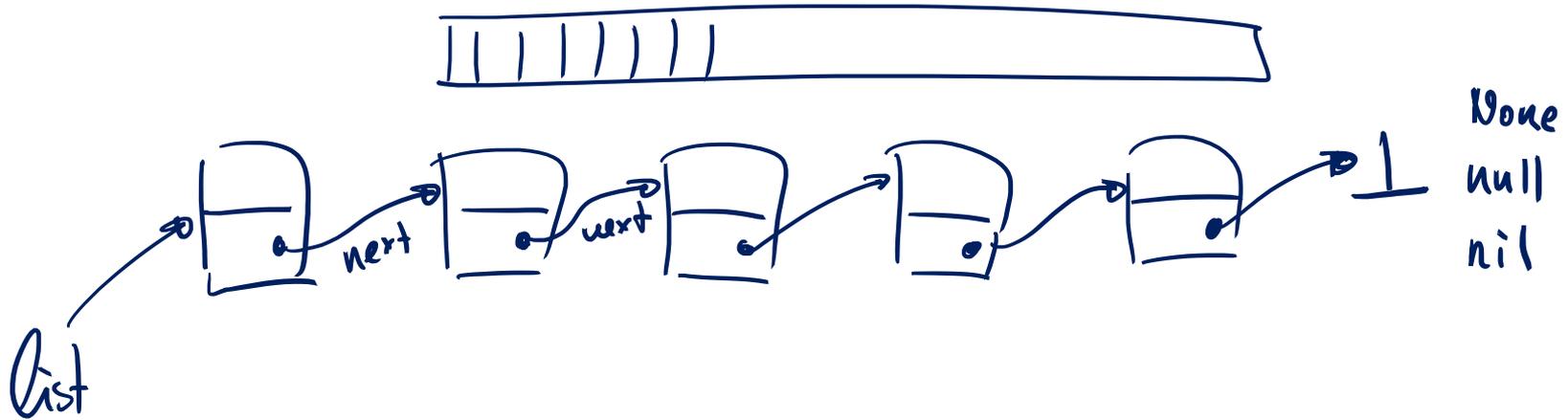
- create: $O(1)$
 - falls man davon ausgeht, dass Speicher in $O(1)$ Zeit alloziert werden kann
- enqueue : $O(1)$
- dequeue : $O(1)$
- isEmpty : $O(1)$

Nachteile der Implementierung:

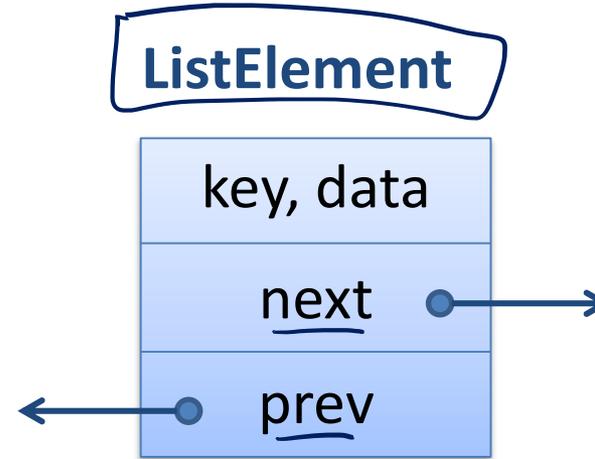
- Speicherverbrauch (space complexity) : $O(NMAX)$
 - man braucht immer gleich viel Speicher, egal wie viele Elemente in der Queue gespeichert sind!
- Die Queue kann nur $NMAX-1$ Elemente aufnehmen...
- Wir werden gleich sehen, wie man beides beheben kann...

Verkettete Listen (Linked Lists)

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



- Klasse, um Listenelemente zu beschreiben



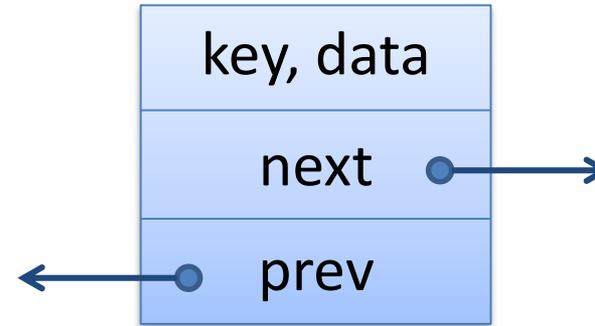
Python:

```
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
```

- Klasse, um Listenelemente zu beschreiben

ListElement



Java:

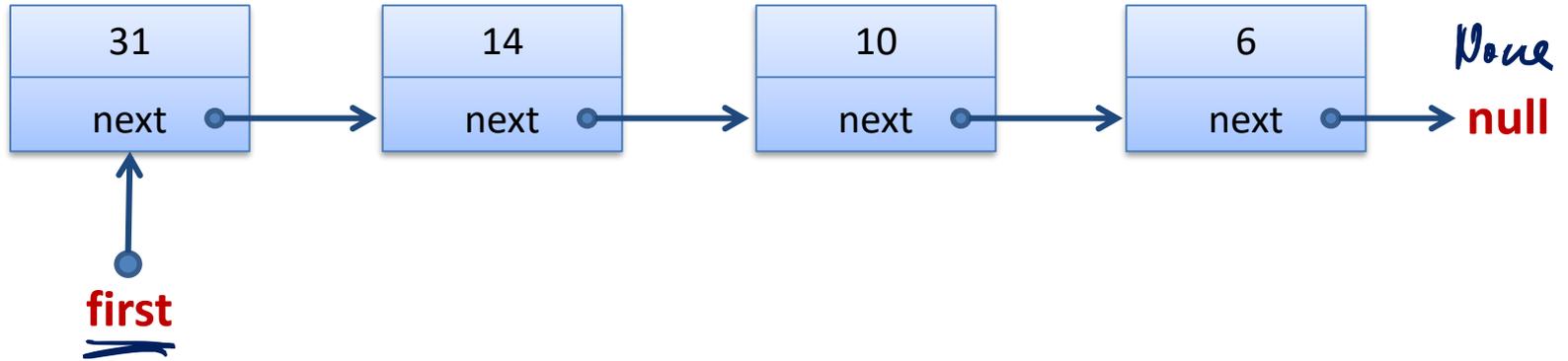
```
public class ListElement {  
    int/String/... key;  
    Object/... data;  
  
    ListElement next;  
    ListElement prev;  
}
```

C++:

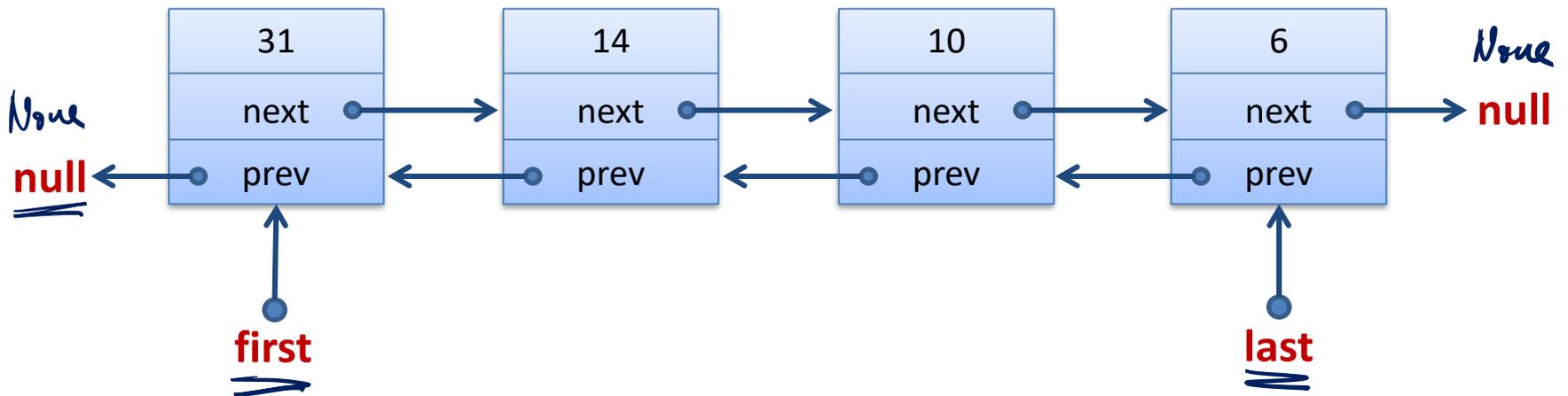
```
class ListElement {  
public/private:  
    int/... key;  
    void*/... data;  
  
    ListElement* next;  
    ListElement* prev;  
}
```

Verkettete Listen: Struktur

Einfach verkettete Liste (Singly Linked List):



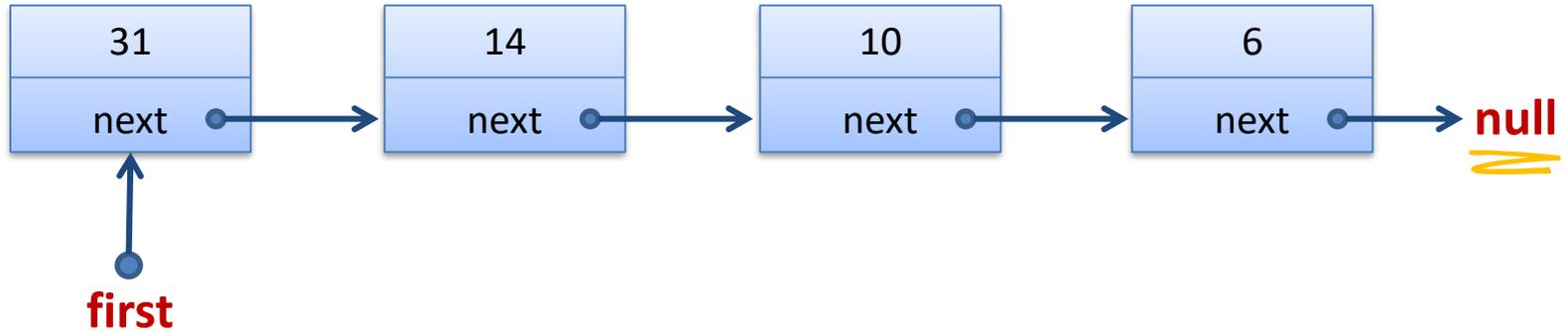
Doppelt verkettete Liste (Doubly Linked List):



Suchen in verketteten Listen

Einfach verkettete Liste (Singly Linked List):

key
next



Finde Element mit Schlüssel x

current = first

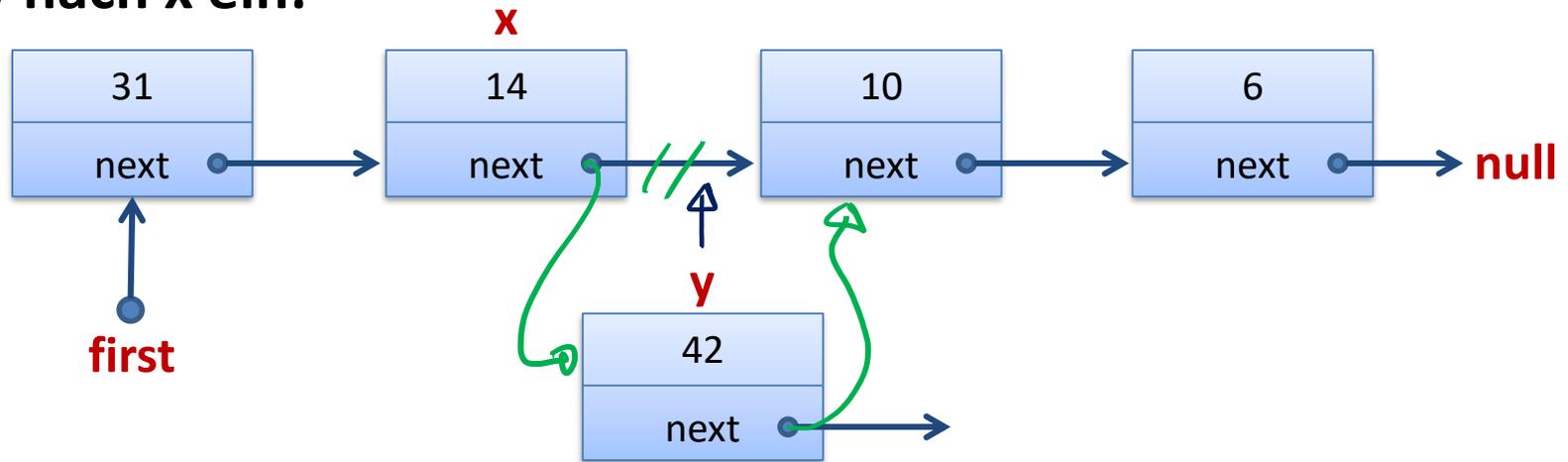
while current != None and
current = current.next

current.key != x :

} Laufzeit
Liste der Länge n
 $O(n)$

Einfügen in einfach verketteten Listen

Füge y nach x ein:



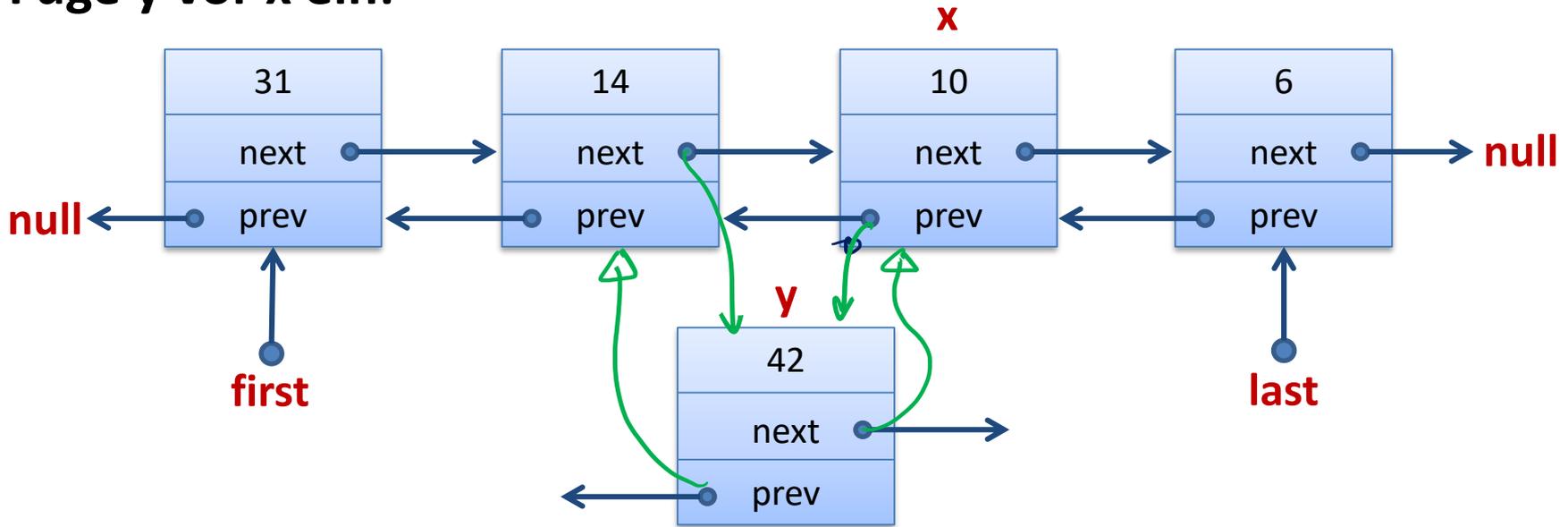
$$y.next = x.next$$

$$x.next = y$$

Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Einfügen in doppelt verketteten Listen

Füge y vor x ein:



$$y.next = x$$

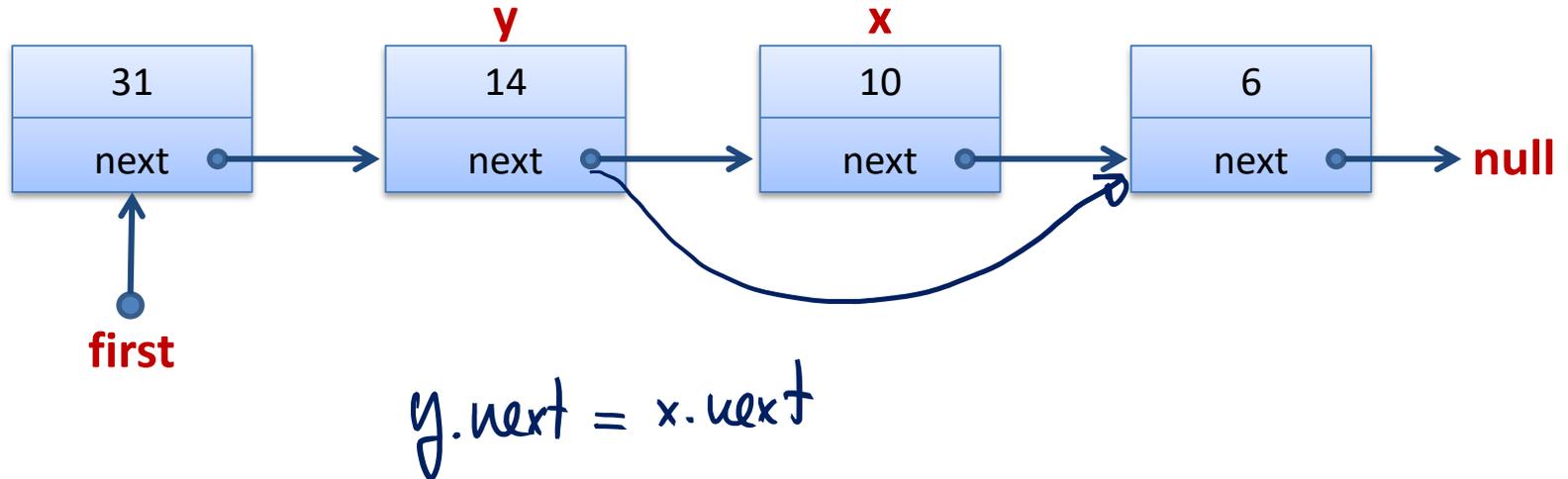
$$y.prev = x.prev$$

$$y.prev.next = y$$

Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Löschen in einfach verketteten Listen

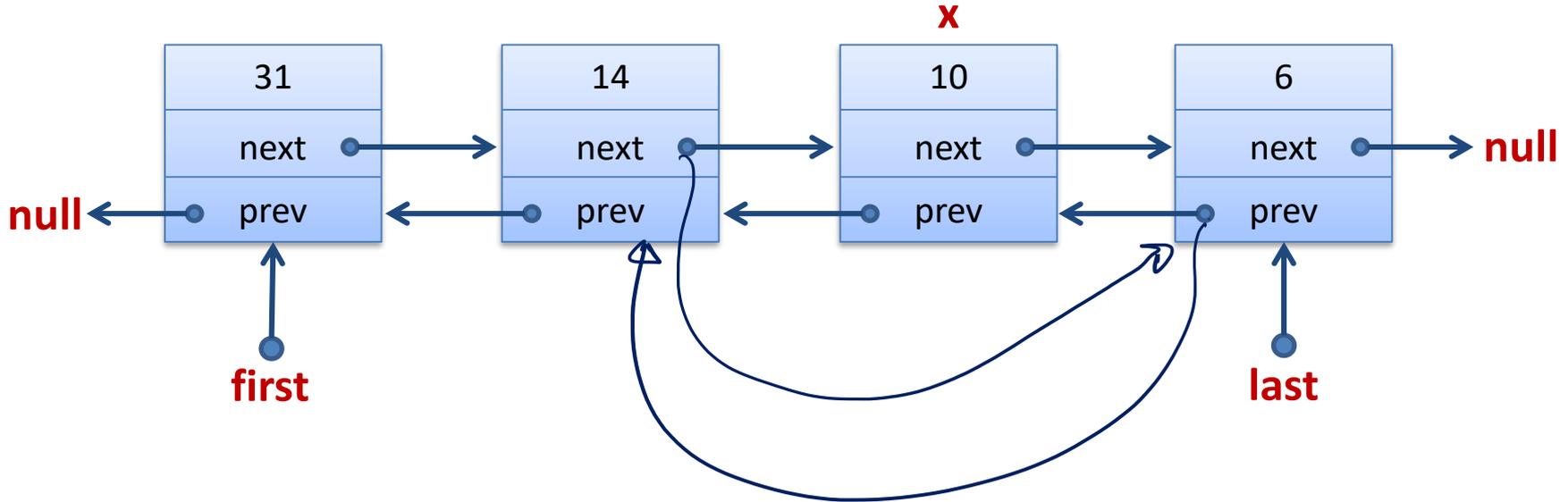
Lösche Element x:



Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Löschen in doppelt verketteten Listen

Lösche Element x:



$$x.\text{prev}.\text{next} = x.\text{next}$$

$$x.\text{next}.\text{prev} = x.\text{prev}$$

Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Annahme: Liste hat **Länge n**

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

Einfügen eines Elements: falls Ref. auf Vorgänger: $O(1)$, sonst $O(n)$

Löschen eines Elements: falls Ref. auf Vorgänger (einf. verk. L.) : $O(1)$
auf Element (doppelt v. L.)

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

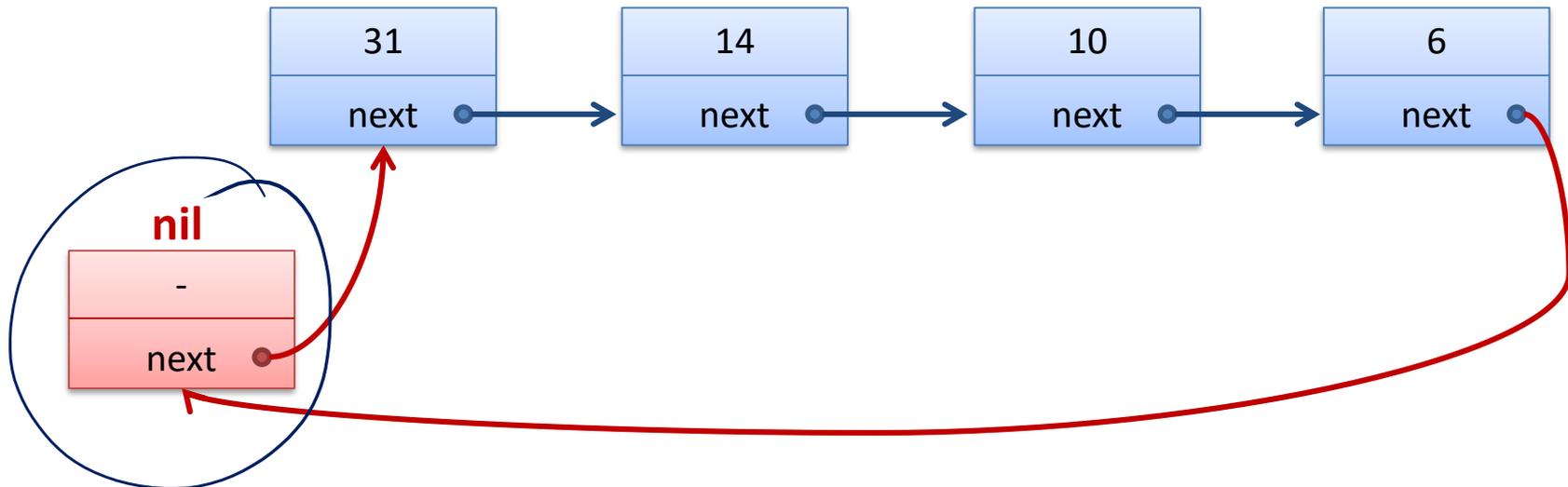
Stack und Queue mit verketteten Listen:

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

Sentinel:

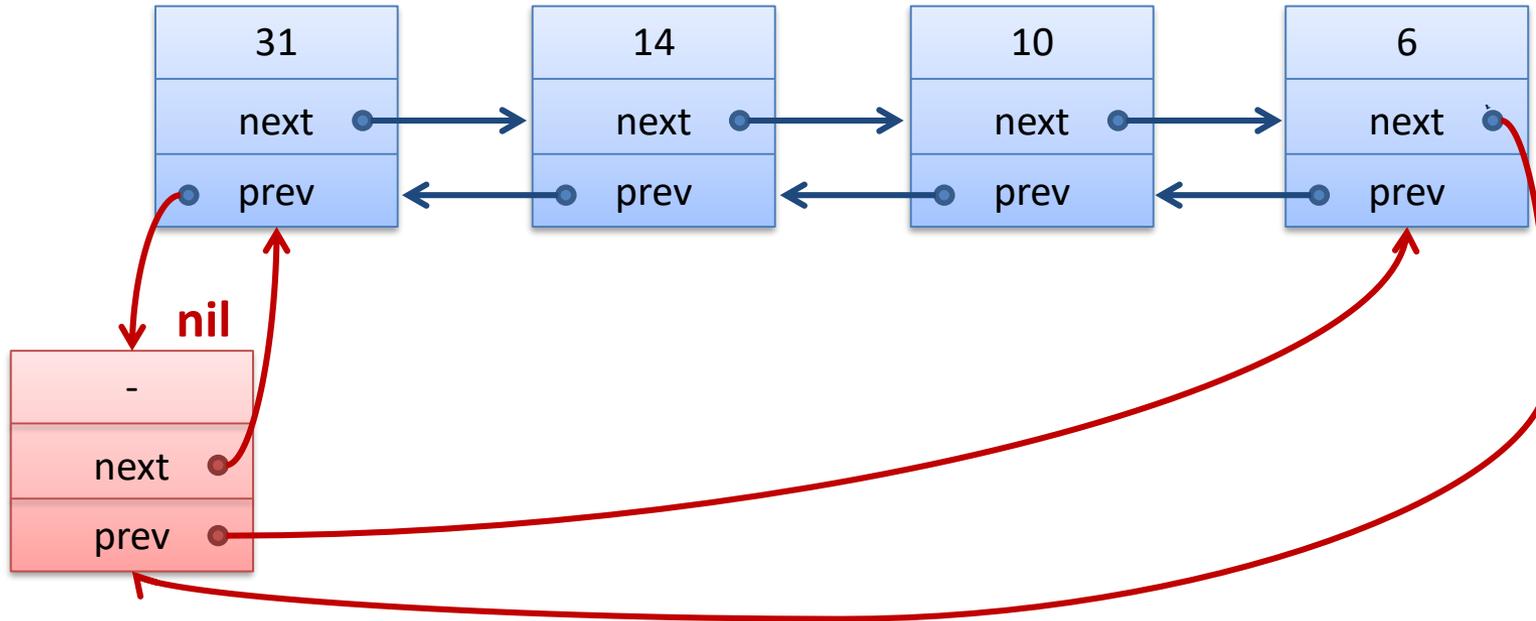
nimmt Rolle von "None" ein

- 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.

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