Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 5 (2.5.2018)

Abstrakte Datentypen

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Erfahrungen 1. Übung

UNI FREIBURG

- 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 auseinandersetzten und “anfreunden” können
- Zum Teil im falschen Verzeichnis, z.T. fehlt das Makefile
  - Unbedingt testen, ob Jenkins fehlerfrei durchläuft
- Durchnittliche 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

Erfahrungen 1. Übung

UNI FREIBURG

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)

Bemerkungen 1. Übung

UNI FREIBURG

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

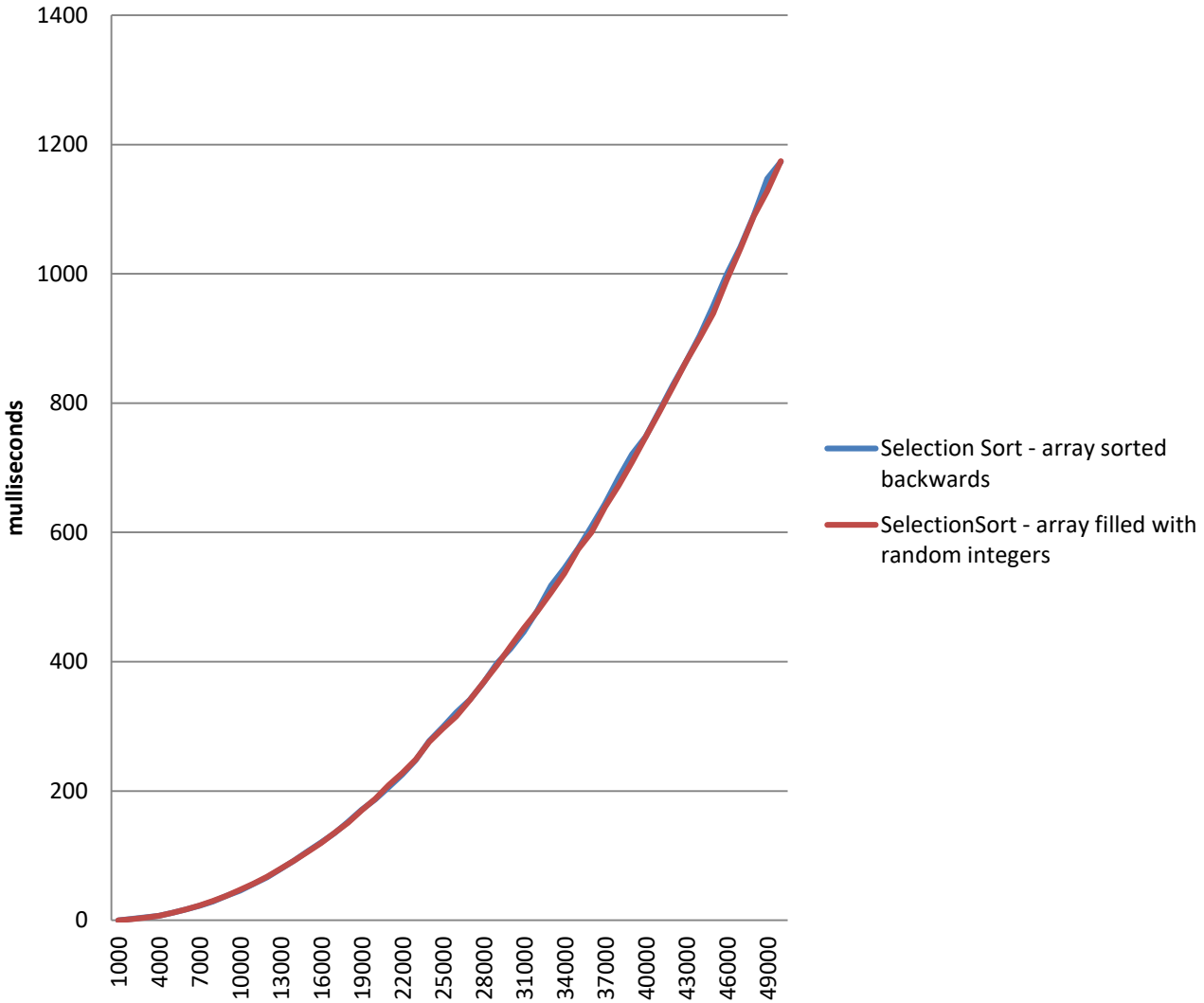
Nächste Übungen

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

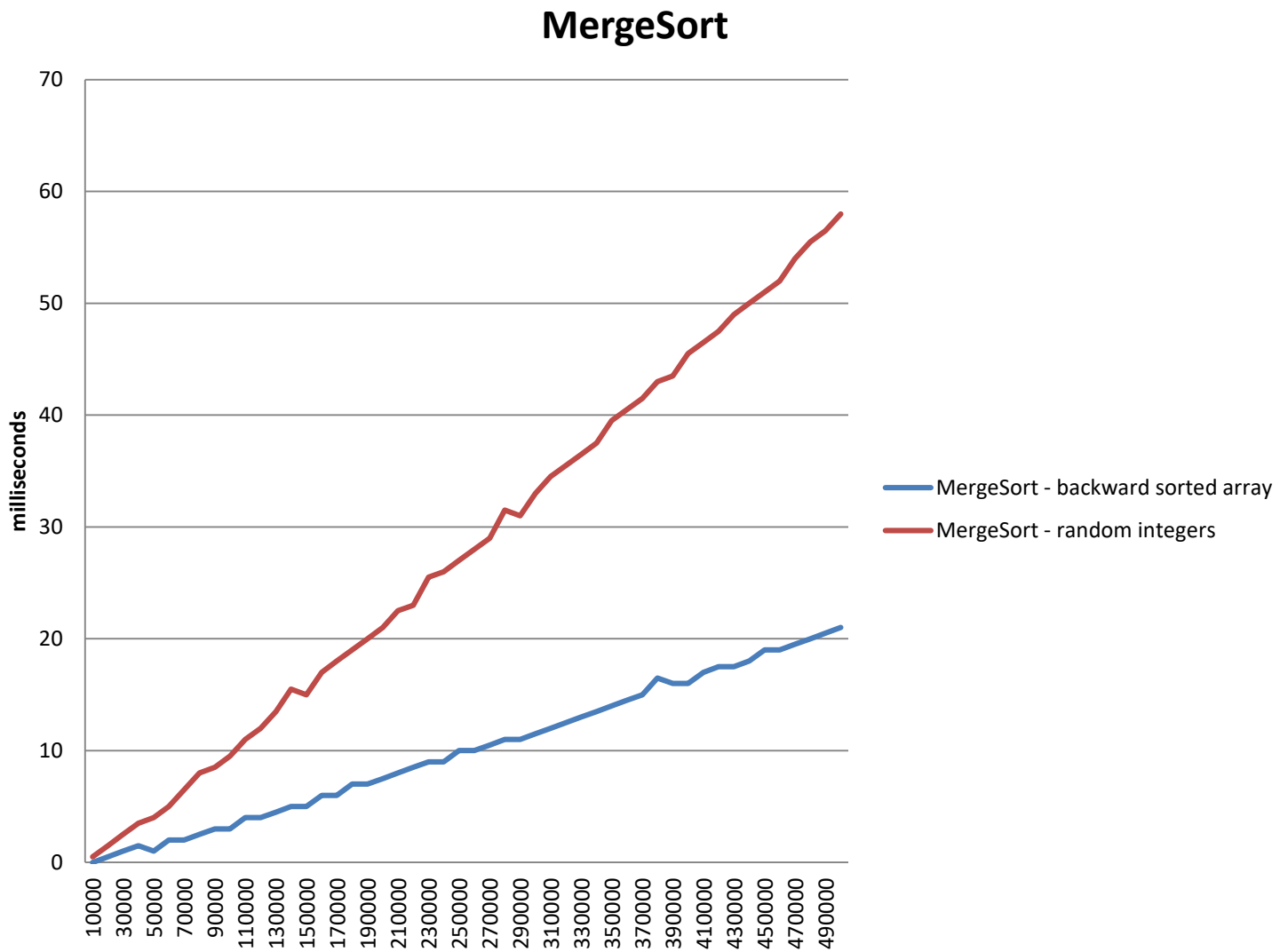
Weitere Programmierübungen:

- 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!

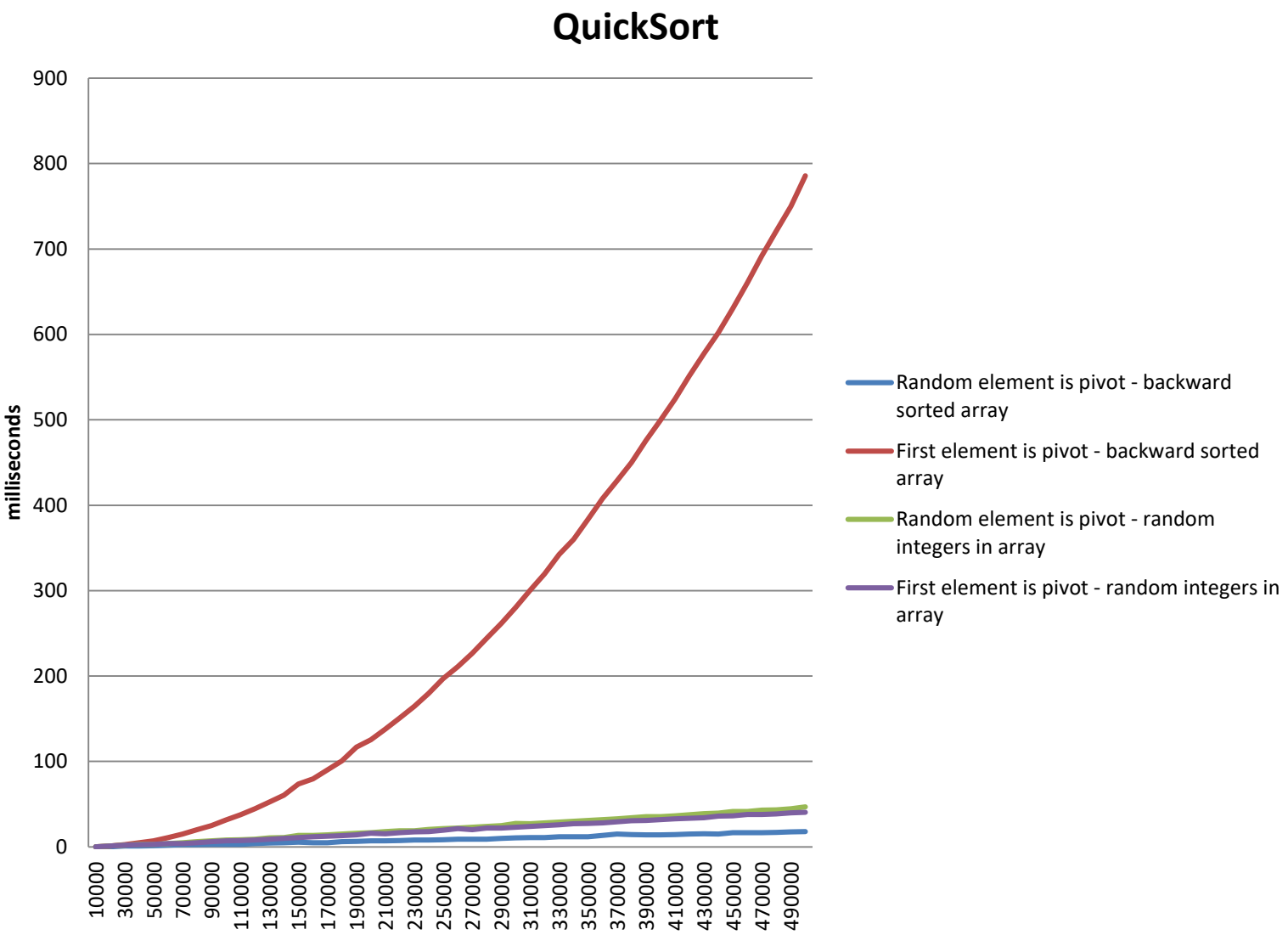
Messungen Selection Sort



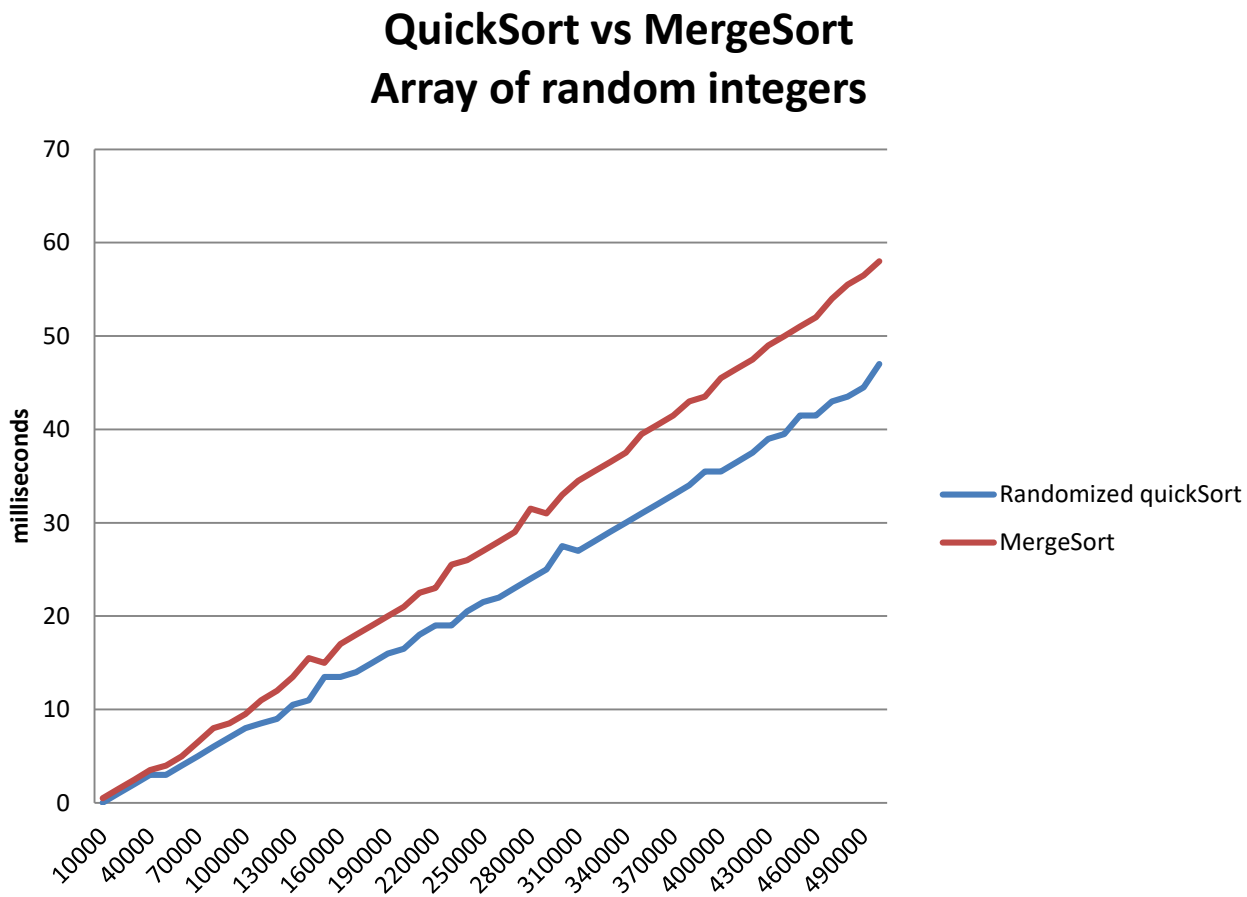
Messungen Merge Sort



Messungen Quick Sort



Messungen – Merge Sort vs. Quick Sort



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.

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

UNI FREIBURG

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

UNI FREIBURG

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

Abstrakte Datentypen : Beispiele

UNI FREIBURG

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) : $O(NMAX)$
  - 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)
```

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 (tail – head) mod NMAX

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
```

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:

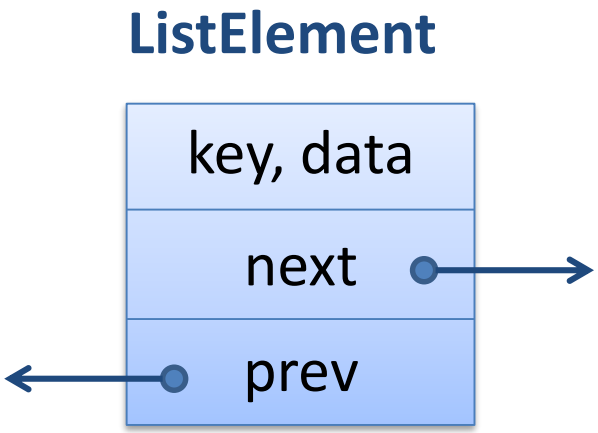
- Speicherverbrauch (space complexity) : $O(NMAX)$
  - man braucht immer gleich viel Speicher, egal wie viele Elemente in der Queue gespeichert sind!
- Die Queue kann nur *NMAX-1* Elemente aufnehmen…
- Wir werden gleich sehen, wie man beides beheben kann…

Verkettete Listen (Linked Lists)

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

Listenelemente

- Klasse, um Listenelemente zu beschreiben

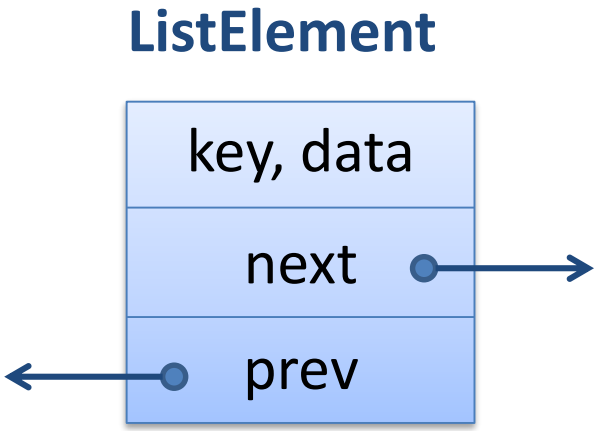


Python:

```
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:

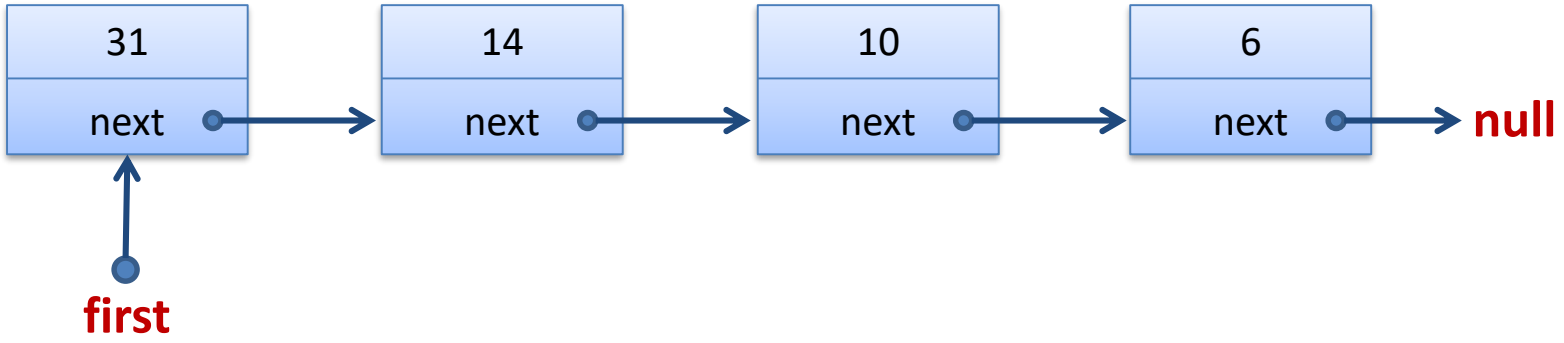
```
public class ListElement {

  int/String/… key;
  Object/… data;

  ListElement next;
  ListElement prev;

}
```

C++:

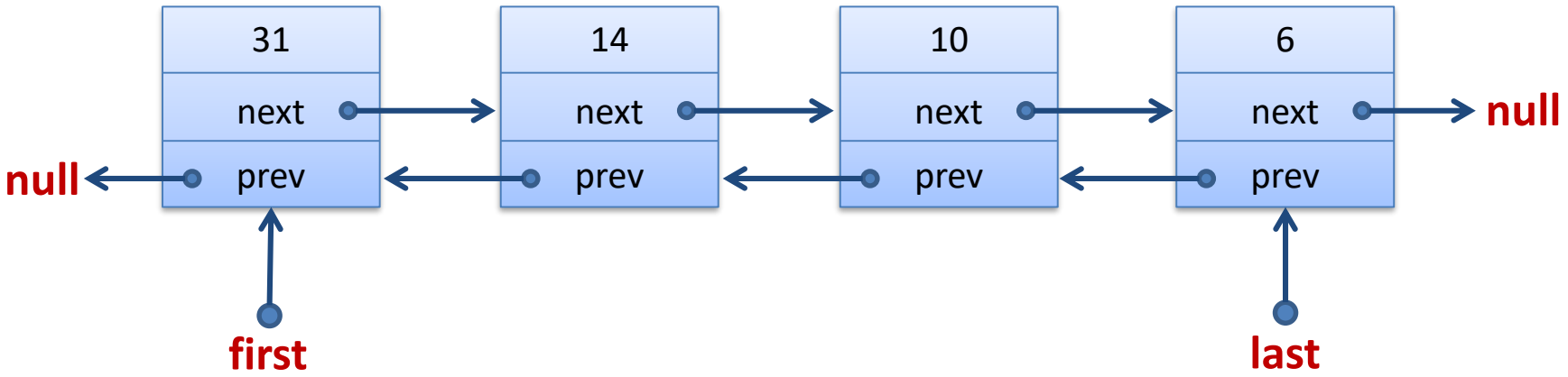
```
class ListElement {
public/private:

  int/… key;
  void*/… data;

  ListElement* next;
  ListElement* prev;

}
```

Verkettete Listen: Struktur

Einfach verkettete Liste (Singly Linked List):



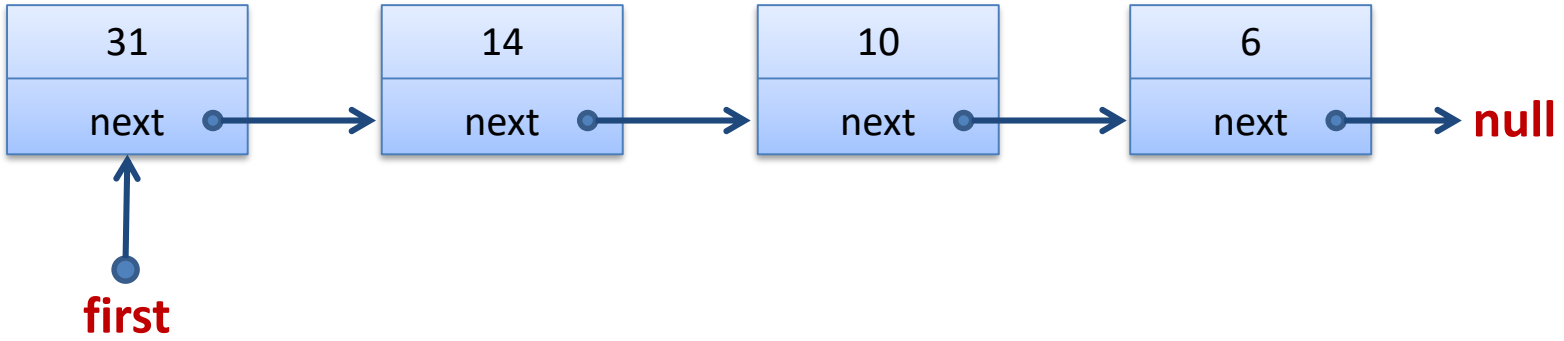
Doppelt verkettete Liste (Doubly Linked List):



Suchen in verketteten Listen

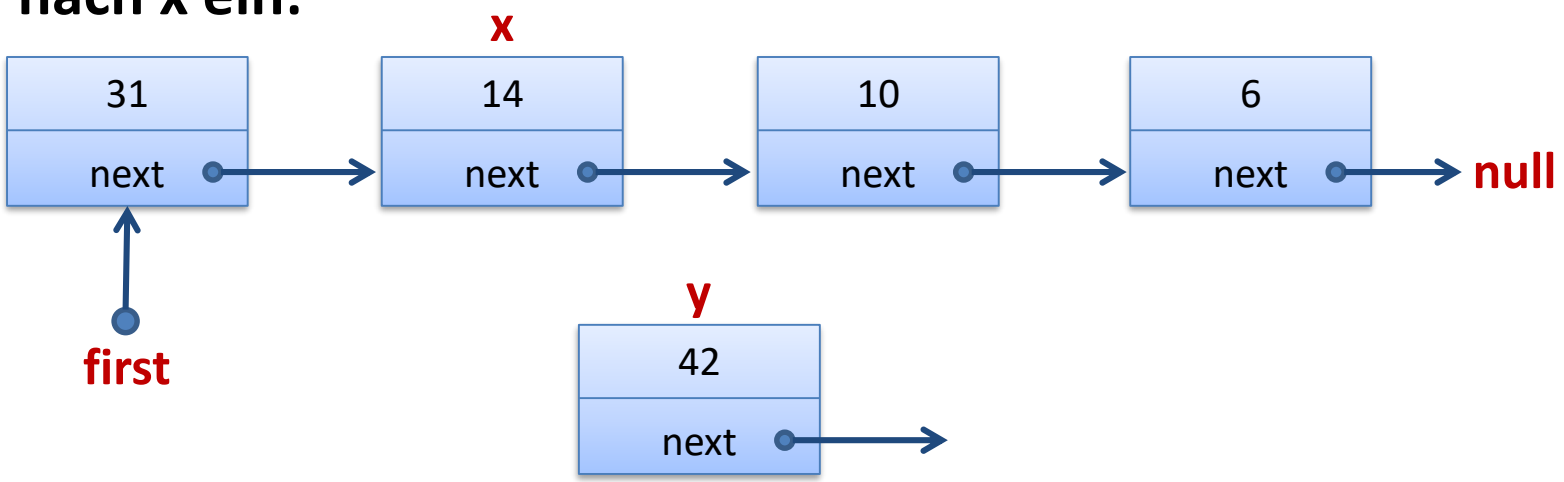
UNI FREIBURG

Einfach verkettete Liste (Singly Linked List):



Einfügen in einfach verketteten Listen

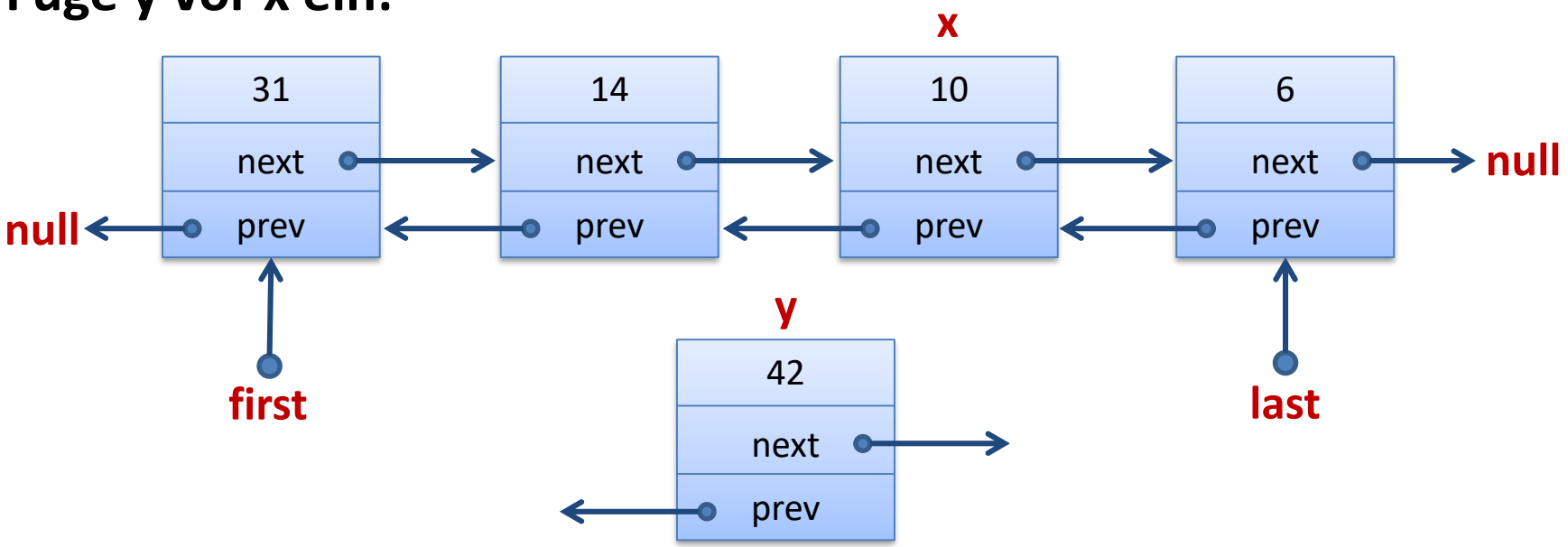
Füge y nach x ein:



Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Einfügen in doppelt verketteten Listen

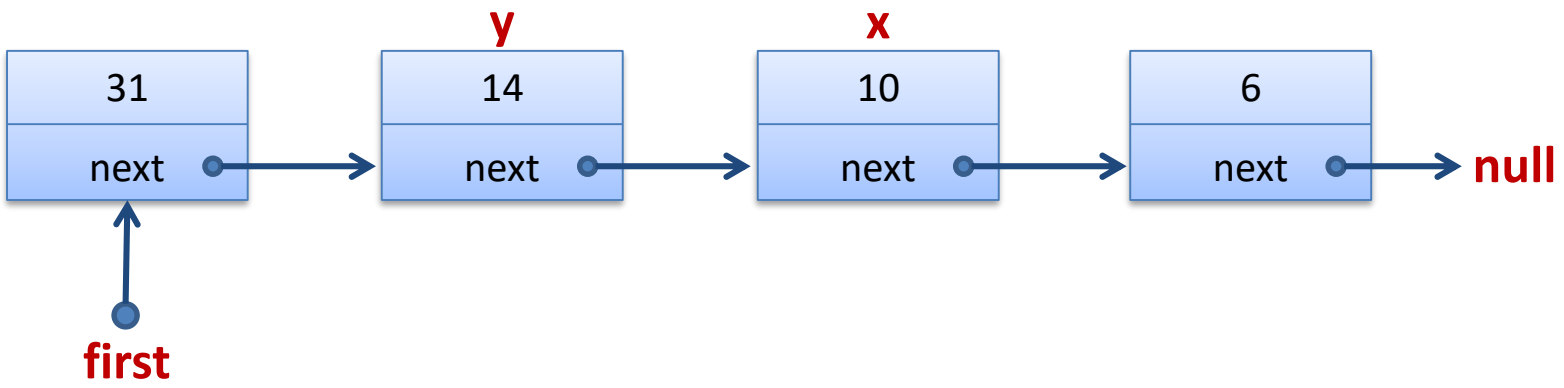
Füge y vor x ein:



Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Löschen in einfach verketteten Listen

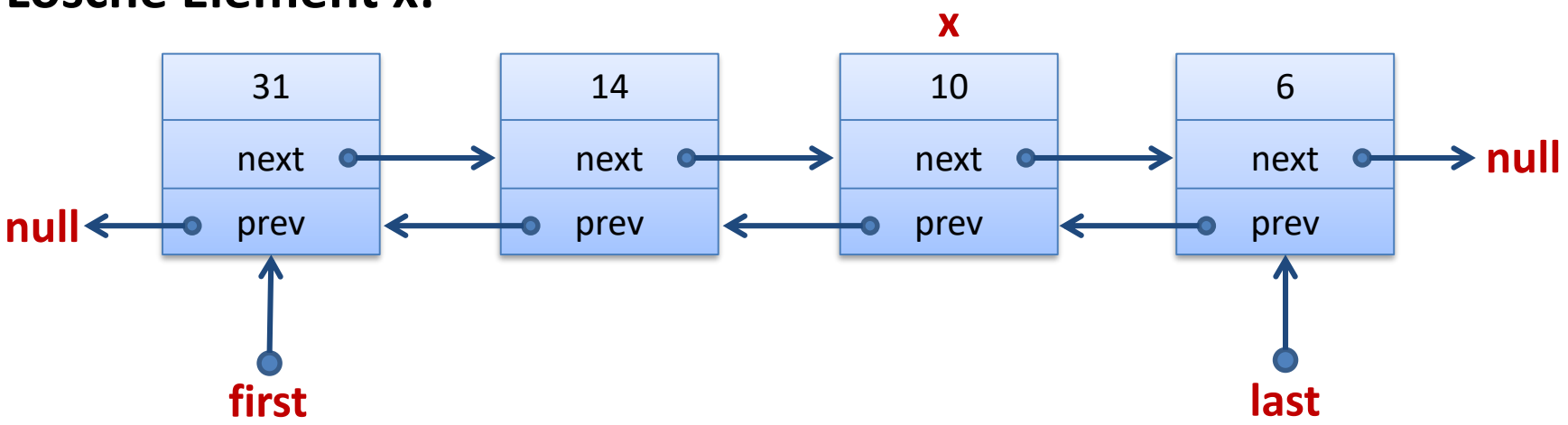
Lösche Element x:



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

UNI FREIBURG

Annahme: Liste hat Länge n

Suche nach Element mit Schlüssel x:

Einfügen eines Elements:

Löschen eines Elements:

Aneinanderhängen (concatenate) von zwei Listen:

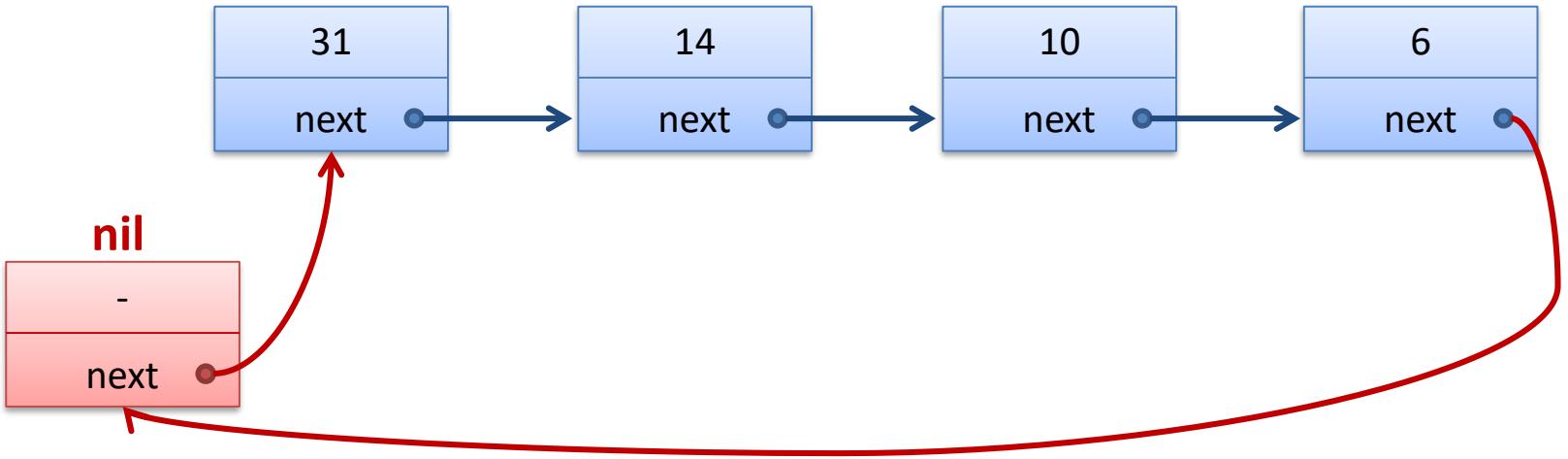
Stack und Queue mit verketteten Listen:

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

Listen mit Sentinel

Sentinel:

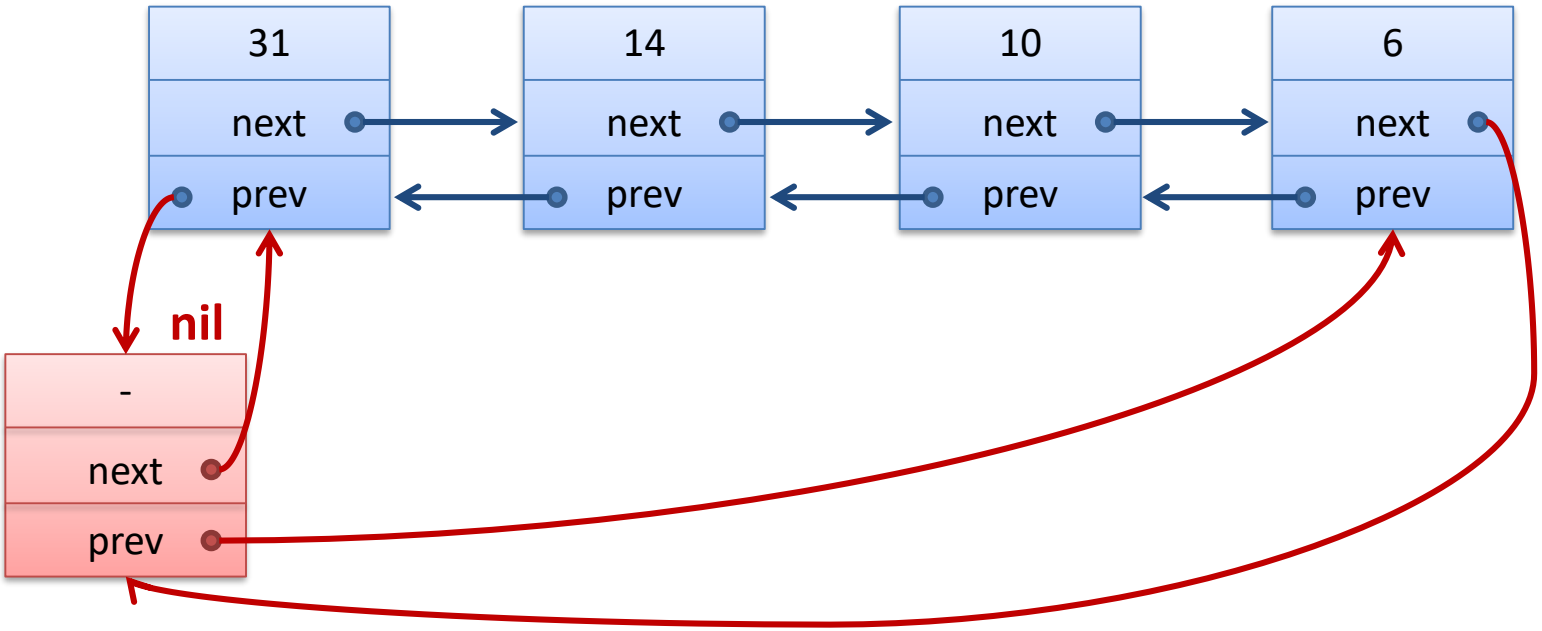
- Ein Dummy-Element, welches Anfang/Ende der Liste bildet



- 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