Algorithmen und Datenstrukturen

Vorlesung 3

Abstrakte Datentypen, einfache Datenstrukturen, Binäre Suche

Fabian Kuhn

Algorithmen und Komplexität

UNI FREIBURG

Datenstrukturen

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

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

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

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

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

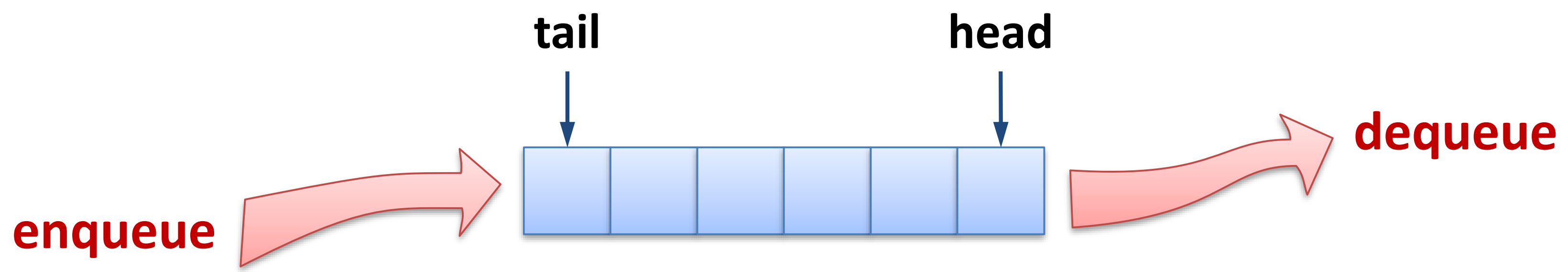
Queue (Warteschlange):

- 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)



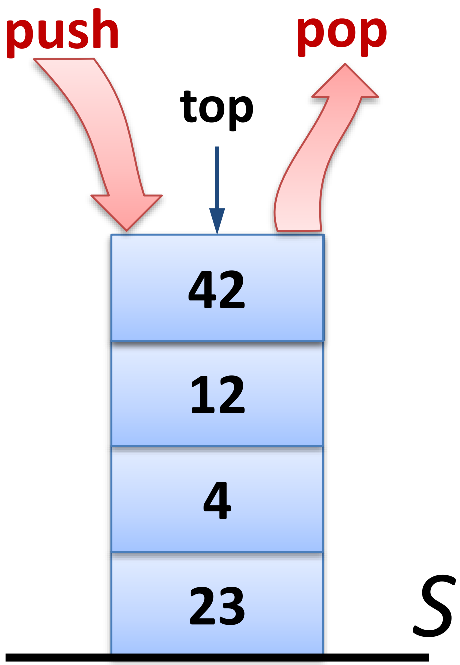
Abstrakte Datentypen : Beispiele

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

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

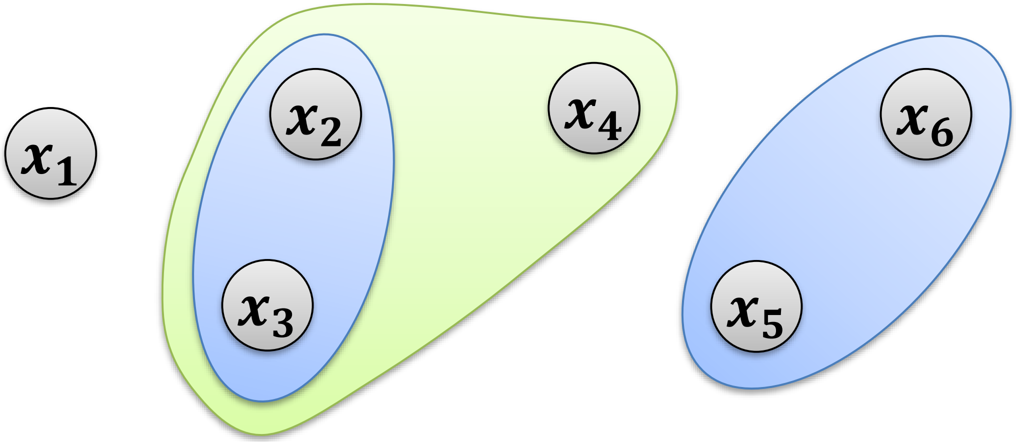
Abstrakte Datentypen : Beispiele

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

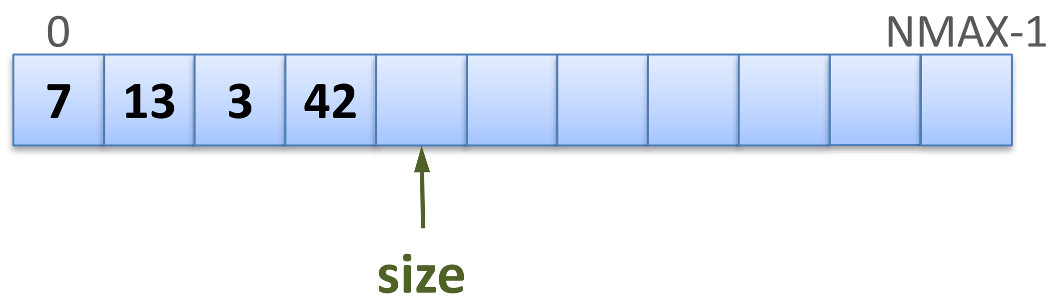
- **Operationen:** *create, push, pop, isEmpty*
- **Annahme:** Stack muss nur für *NMAX* Elemente Platz bieten

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

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…

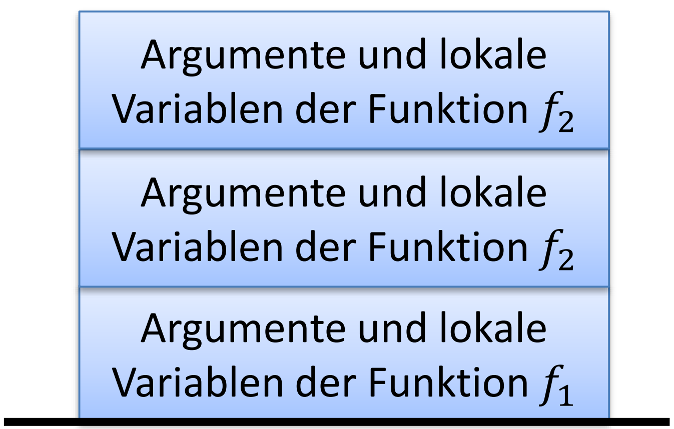
Stack : Anwendungen

- 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 $f_1(x, y)$:
    …
    $f_2(z)$
    …

def $f_2(a)$:
    …
    $f_2(b)$
    …

Array-Implementierung Queue

Versuchen wir den Queue-Datentyp zu implementieren

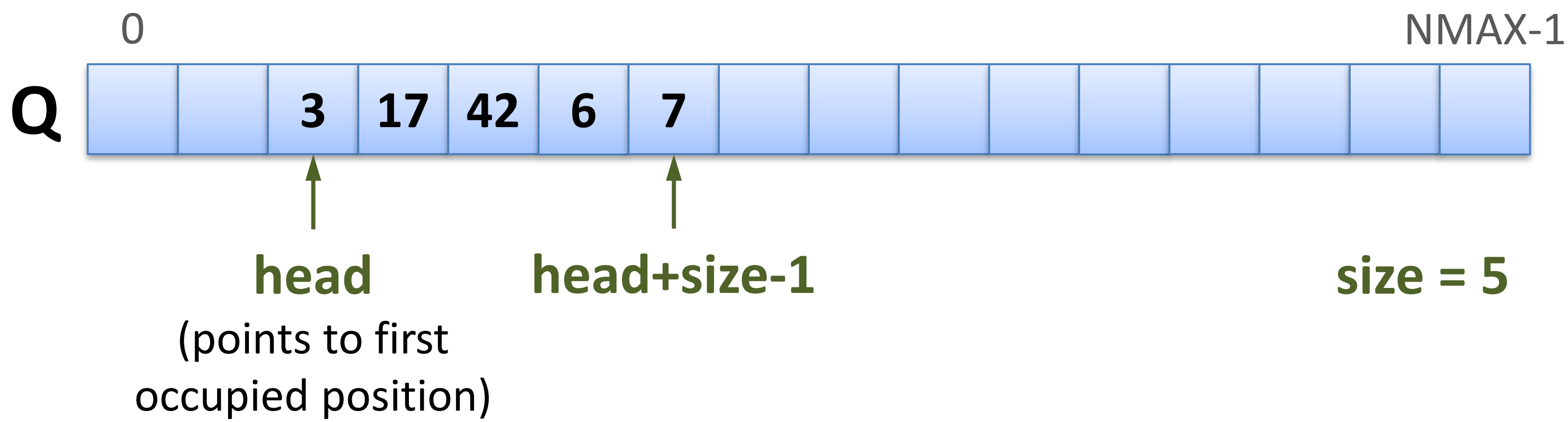
- **Operationen:** *create, enqueue, dequeue, isEmpty*
- **Annahme:** Queue muss nur für *NMAX* Elemente Platz bieten

Variablen, um den Zustand des Stack 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

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:

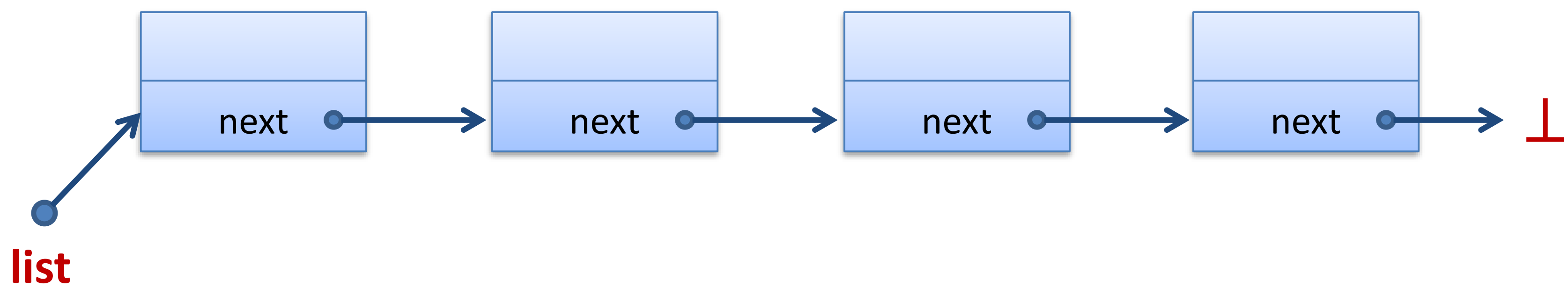
- Speicherverbrauch (space complexity) : $\Theta(NMAX)$
  - man braucht immer gleich viel Speicher, egal wie viele Elemente in der Queue gespeichert sind!
- Die Queue kann nur *NMAX* Elemente aufnehmen…
- Wir werden gleich sehen, wie man beides beheben kann…

Verkettete Listen (Linked Lists)

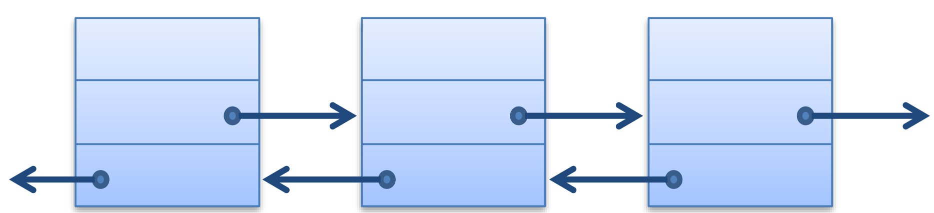
- Datenstruktur, um eine Liste (Sequenz) von Werten zu verwalten

Array:

Verkettete Liste:



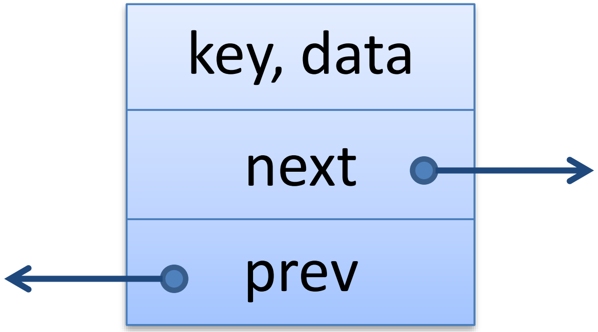
Doppelt verkettete Liste:



Listenelemente

- Klasse, um Listenelemente zu beschreiben

ListElement



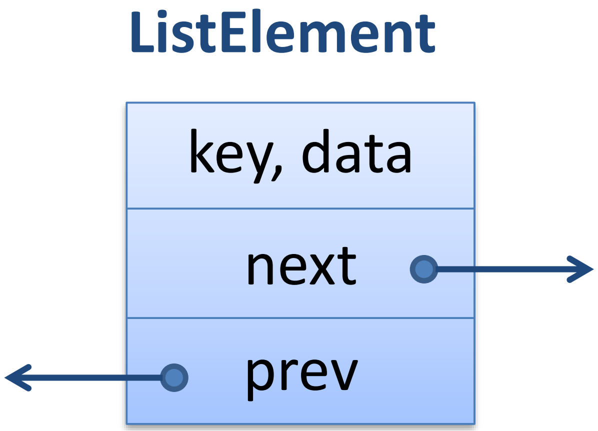
Python:

```
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:

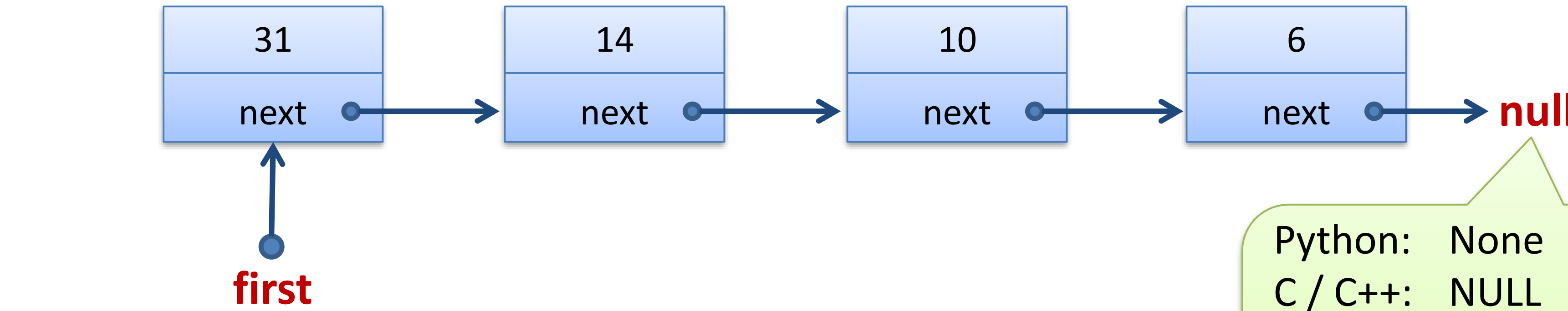
```java
public class ListElement {
   int/String/… key;
   Object/… data;
   ListElement next;
   ListElement prev;
}
```

C++:

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

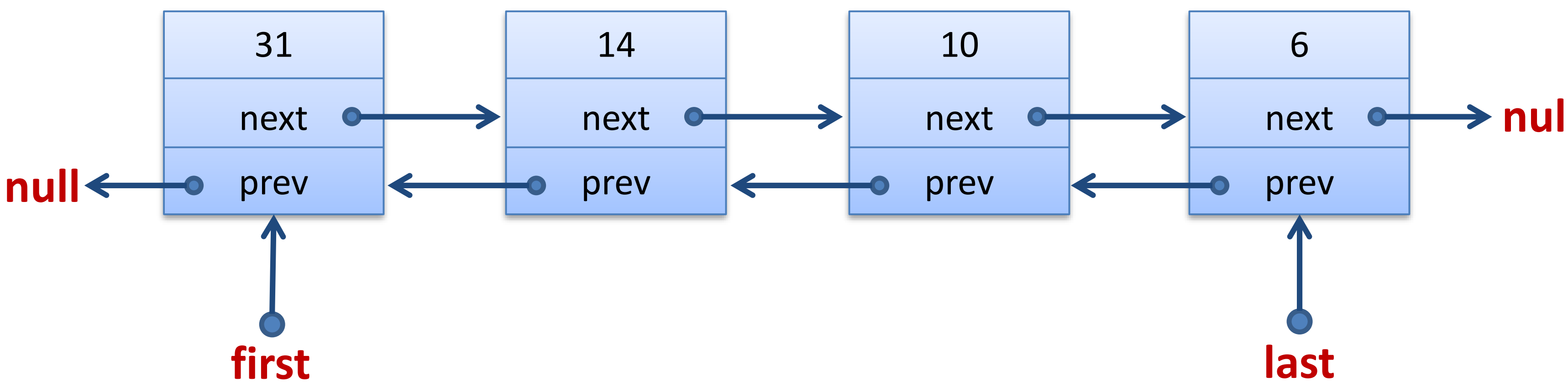
Verkettete Listen: Struktur

Einfach verkettete Liste (Singly Linked List):



Python:	None
C / C++:	NULL
Java:	null
others:	nil
symbol:	⊥

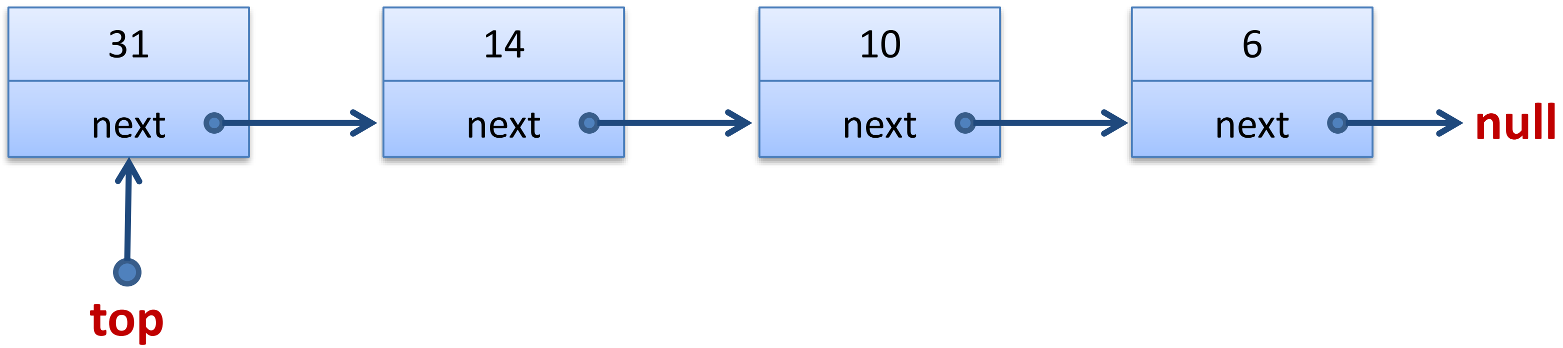
Doppelt verkettete Liste (Doubly Linked List):



Stack und FIFO Queue mit Listen

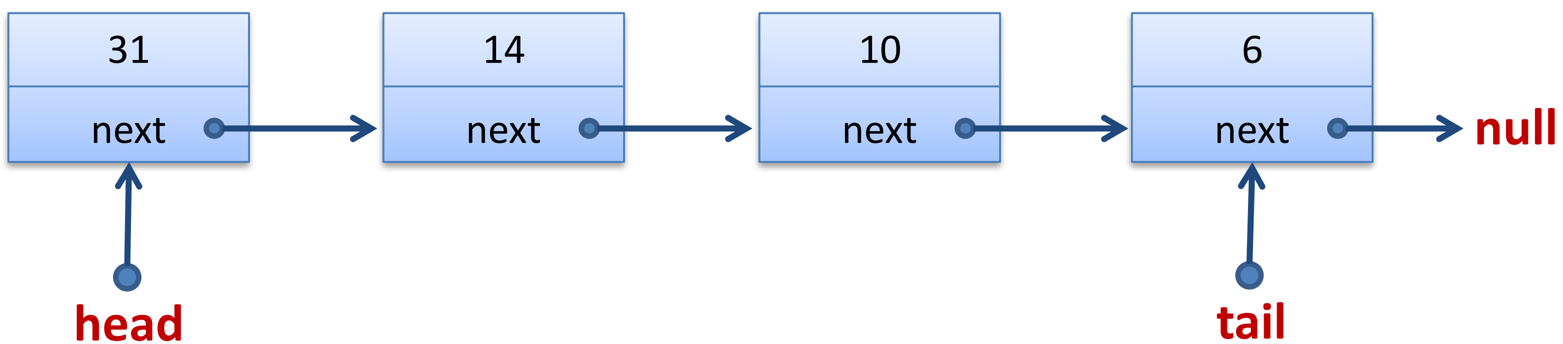
Mit einfach verketteten Listen, alle Operationen in $O(1)$ Zeit

Stack:



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

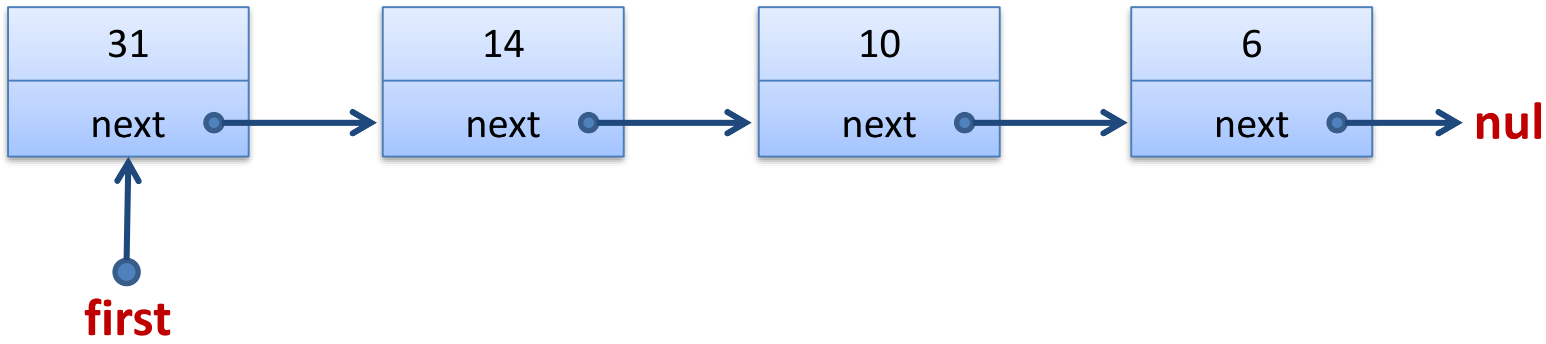
Queue:



- 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):



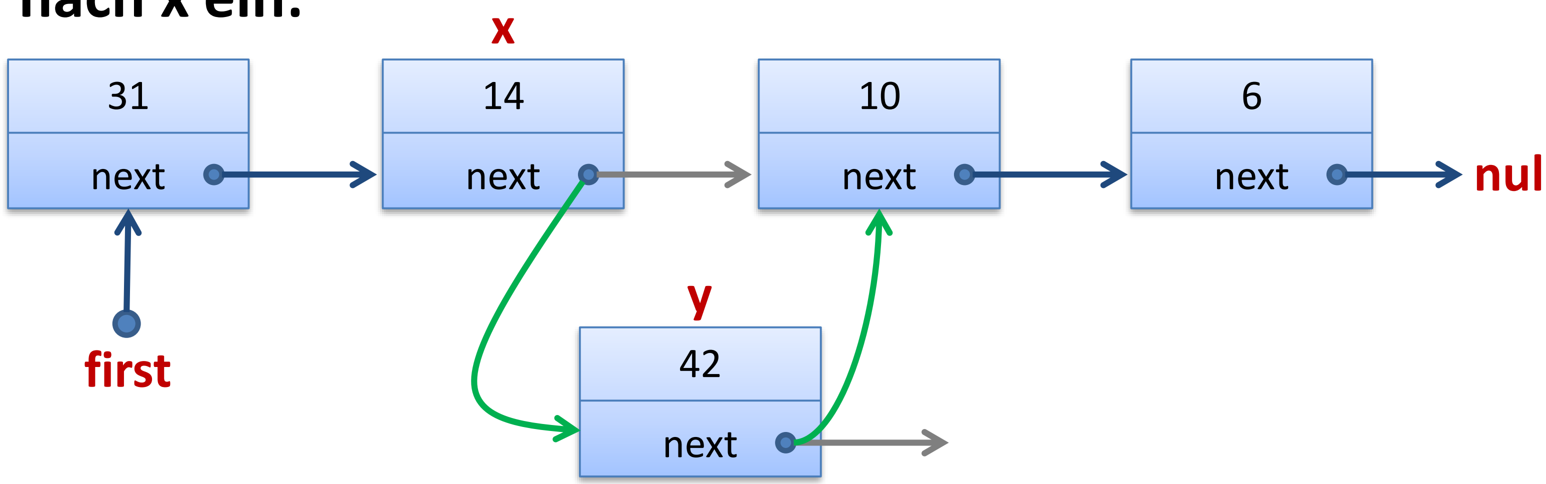
Ziel: Finde Element mit Schlüssel x

```
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:

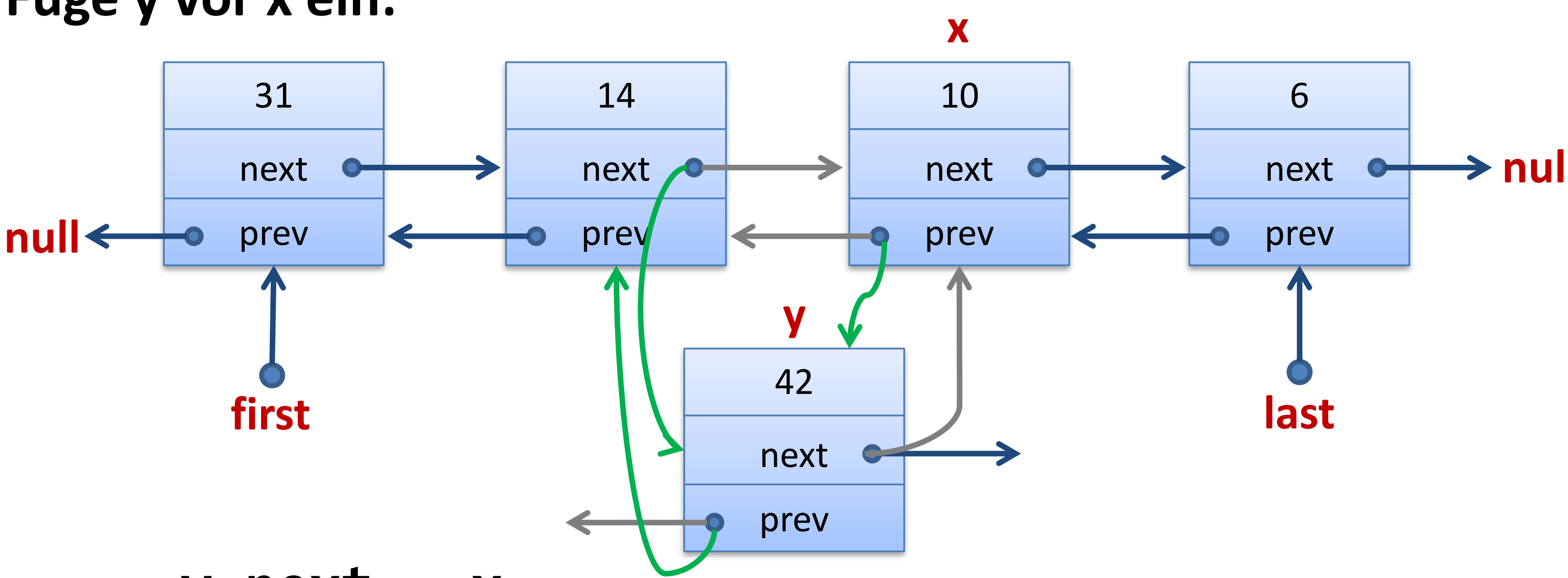



```
y.next = x.next
x.next = y
```

Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Einfügen in doppelt verketteten Listen

Füge y vor x ein:

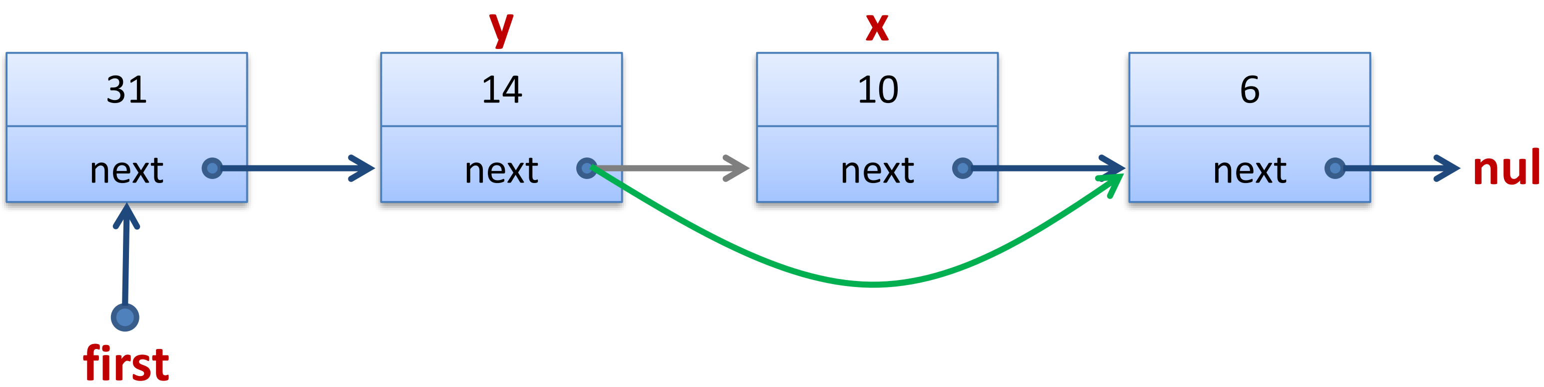



```
y.next = x
y.prev = x.prev
x.prev.next = y
x.prev = y
```

Achtung: Spezialfälle bei Einfügen am Anfang/Ende beachten!

Löschen in einfach verketteten Listen

Lösche Element x:



Annahme: Vorgängerelement y ist gegeben

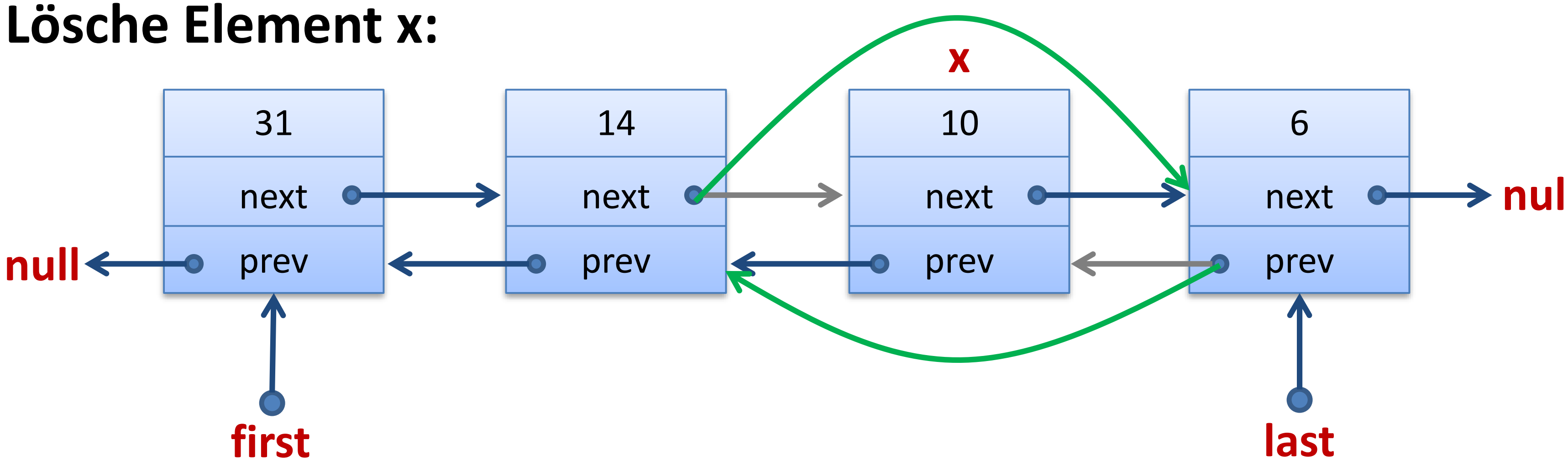
```
y.next = x.next
```

- 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

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

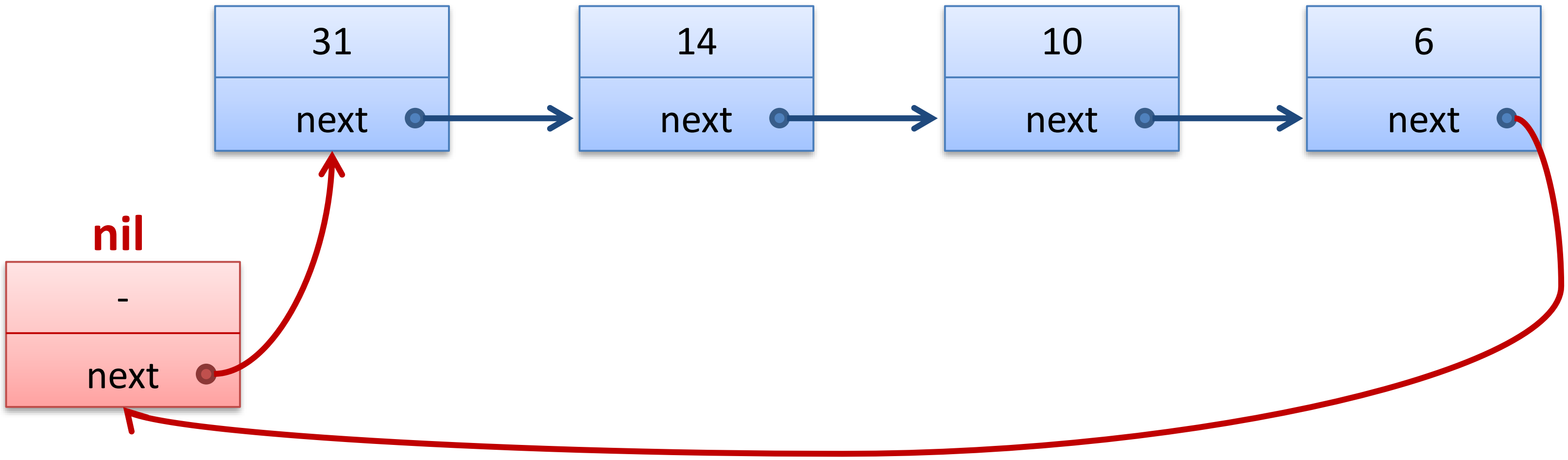
Stack und Queue mit verketteten Listen:

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

Listen mit Sentinel

Sentinel:

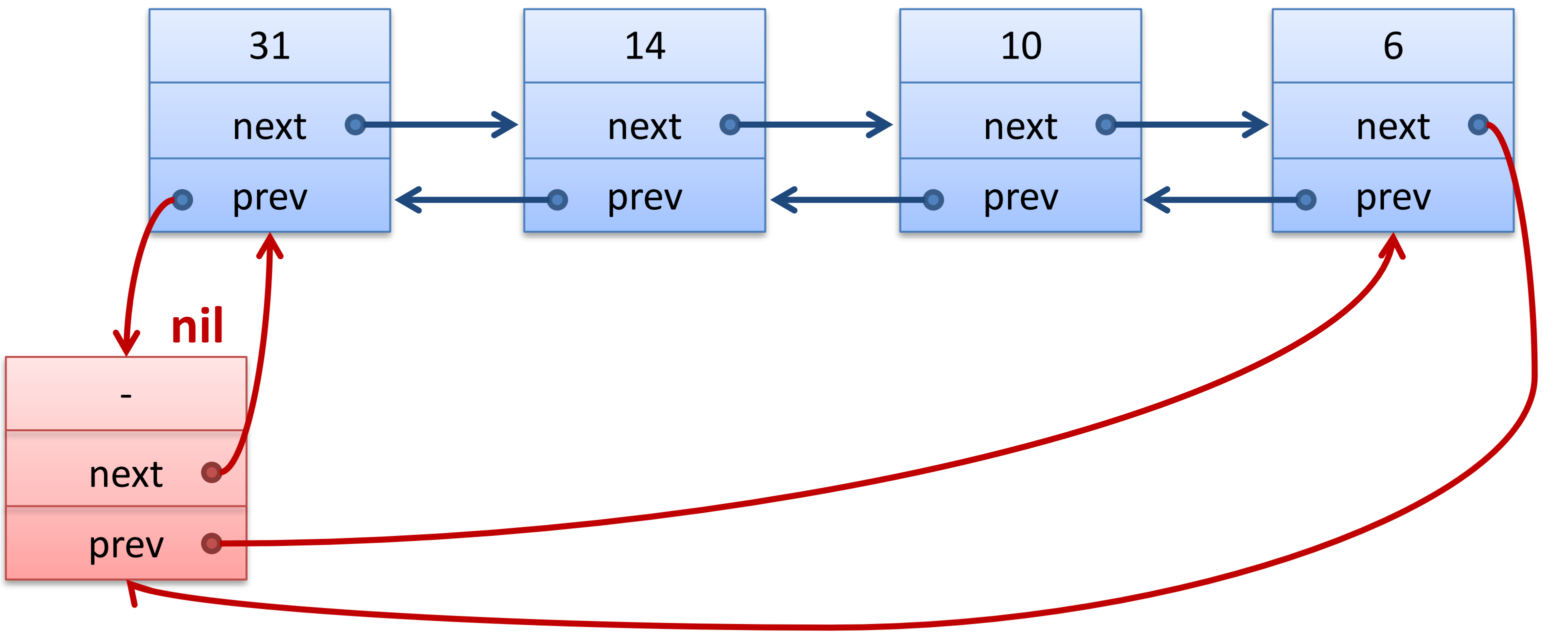
- Ein Dummy-Element, welches Anfang/Ende der Liste bildet



- 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

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

Abstrakte Datentypen : Dictionary

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

- 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!

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: $O(1)$

- Falls man nicht überprüfen muss, ob der Schlüssel schon vorkommt

find: $O(n)$

- Wir müssen möglicherweise über die ganze Liste iterieren

delete: $O(n)$

- Wir müssen möglicherweise über die ganze Liste iterieren

Ist das gut?

- Insbesondere find ist sehr teuer!

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: $O(1)$

insert: $O(1)$

find: $O(n)$

- Wir müssen möglicherweise über das ganze Array iterieren

delete: $O(n)$

- Wir müssen möglicherweise über das ganze Array iterieren und im Worst Case $\Omega(n)$ Werte umkopieren

Bessere Ideen?

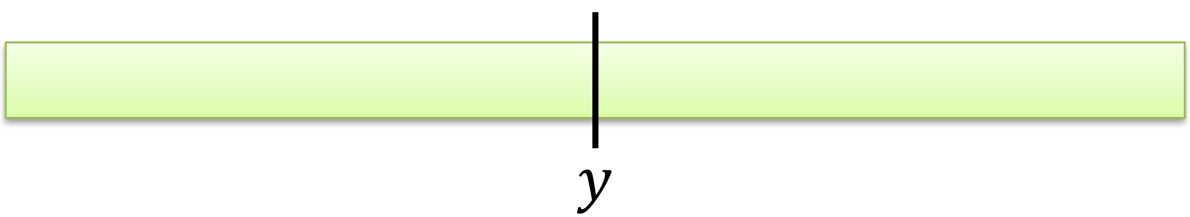
- Insbesondere find ist immer noch sehr teuer!

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 für Suche nach x:

- Wir schlagen Telefonbuch mal ungefähr in der Mitte auf und schauen, ob der Name in der ersten oder in der zweiten Hälfte ist.



Ist $y < x$ oder ist $y > x$ oder ist $y = x$?

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**

Binäre Suche

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

Algorithmus (Array A der Länge n, Suche nach Schlüssel x):

- Behalte linken und rechten Rand l und r, so dass (falls x in A ist)

$$A[l] \leq x \leq A[r]$$

- Am Anfang setzen wir $l = 0$ und $r = n - 1$

- Gehe in die Mitte $m = (l + r)/2$
  - Falls $A[m] = x \implies x$ gefunden!
  - Falls $A[m] < x \implies x$ ist im rechten Teil $\implies l = m + 1$
  - Falls $A[m] > x \implies x$ ist im linken Teil $\implies l = m - 1$

Binäre Suche

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

Algorithmus (Array A der Länge n, Suche nach Schlüssel x):

```
l = 0; r = n – 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then
        l = m + 1
    else if A[m] > x then
        r = m – 1
    else
        l = m; r = m
```

Falls Schlüssel x im Array ist, dann gilt am Schluss $A[l] = x$

Ist der Algorithmus korrekt?

Wie überprüft man das?

- Empirisch: Unit Test oder auch systematischere Tests...
- Formal?
  - Korrektheit ist (meistens) noch wichtiger als Performance!

Hoare Kalkül

- Wir schauen hier nur die Grundideen an
- **Vorbedingung**
  - Bedingung, welche am Anfang (der Methode / Schleife / ...) gilt
- **Nachbedingung**
  - Bedingung, welche am Schluss (der Methode / Schleife / ...) gilt
- **Schleifeninvariante**
  - Bedingung welche am Anfang / Ende jedes Schleifendurchlaufs gilt

Ist der Algorithmus korrekt?


```
l = 0; r = n – 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m – 1
    else l = m; r = m
```

Vorbedingung

- *Array ist am Anfang sortiert, Array hat Länge n*

Nachbedingung

- *Falls x im Array ist, dann gilt $A[l] = x$*

Schleifeninvariante

- *Falls x im Array ist, dann gilt $A[l] \leq x \leq A[r]$*

Ist der Algorithmus korrekt?

Vorbedingung

- *Array ist am Anfang sortiert, Array hat Länge n*

```
l = 0; r = n - 1;
```

Schleifeninvariante

Schleifeninvariante

- *Falls x im Array ist, dann gilt $A[l] \leq x \leq A[r]$*
- Vorbedingung und Zuweisung zu l und r → Schleifeninvariante
  - Invariante gilt am Anfang des ersten Schleifendurchlaufs

Nachbedingung

- *Falls x im Array ist, dann gilt $A[l] = x$*
- Abbruchbedingung while-Schleife $\boxed{\rightarrow l \geq r}$ und damit $\boxed{A[l] \geq A[r]}$
- Falls x im Array ist, dann folgt aus der Schleifeninvariante und da A sortiert ist, dass $\boxed{A[l] = A[r] \text{ und damit } A[l] = x}$

Ist der Algorithmus korrekt?

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m - 1
    else l = m; r = m
```

Schleifeninvariante

- *Falls x im Array ist, dann gilt $A[l] \leq x \leq A[r]$*
  - Die Schleifeninvariante gilt am Anfang der Schleife, sie kann nur ungültig werden, wenn wir die Variablen l und r verändern
  - Wenn wir $l = m + 1$ setzen, dann wissen wir, dass $A[m] < x$, daher gilt danach $A[m + 1] \leq x$ falls x enthalten ist.
  - Analog, wenn wir $r = m - 1$ setzen, dann wissen wir, dass $A[m] > x$, daher gilt danach $x \leq A[m - 1]$ falls x enthalten ist.

Terminiert der Algorithmus?

```
l = 0; r = n - 1;
while r > l do
    m = (l + r) / 2;
    if A[m] < x then l = m + 1
    else if A[m] > x then r = m - 1
    else l = m; r = m
```

- Veränderung der Anz. Elemente $(r - l + 1)$ pro Schleifendurchlauf?
  - $l = m + 1$:
    $$r - (m + 1) + 1 \leq r - \left(\frac{l + r}{2} + \frac{1}{2}\right) + 1 = \frac{r - l + 1}{2}$$
  - $r = m - 1$:
    $$(m - 1) - l + 1 \leq \frac{l + r}{2} - 1 - l + 1 = \frac{r - l}{2} < \frac{r - l + 1}{2}$$
  - Sonst wird x gefunden und $r - l + 1$ wird 1

Laufzeit

UNI FREIBURG

Terminiert der Algorithmus?

- In jedem Schleifendurchlauf wird die Anzahl der Elemente mindestens halbiert.
- Der Algorithmus terminiert!

Laufzeit?

$$T(n) \leq T\left(\left\lfloor n/2 \right\rfloor\right) + c, \qquad T(1) \leq c$$

$$\begin{aligned}
T(n) &\leq T(n/2) + c \\
&\leq T(n/4) + \underbrace{c + c}_{2c} \\
&\leq T(n/8) + 3c \\
&\;\;\vdots \\
&\leq T(n/2^i) + i \cdot c \\
&\;\;\vdots \\
&\leq T(1) + c(\log_2 n) \leq c(\log_2 n + 1)
\end{aligned}$$

Laufzeit Binäre Suche

Der Algorithmus terminiert in Zeit $O(\log n)$.

$$T(n) \leq T(n/2) + c, \quad T(1) \leq c$$

Vermutung: $T(n) \leq c(\log_2 n + 1)$

Verankerung: $n = 1 \quad T(1) \leq c(0+1) = c$ ✓

Schritt: $n > 1 \quad T(n) \leq T(n/2) + c$

$$\leq c(\underbrace{\log_2 \tfrac{n}{2}}_{\log_2 n} + 1) + c$$

$$= c(\log_2 n + 1).$$

Dictionary mit sortiertem Array

Operationen:

- *create*:
  - lege neues Array der Länge *NMAX* an
- *D.find(key)*:
  - **Suche nach *key* mit binärer Suche**
- *D.insert(key, value)*:
  - suche nach *key* und füge neues Element an der richtigen Stelle ein
  - Einfügen: alles dahinter muss um eins nach hinten geschoben werden!
- *D.delete(key)*:
  - suche zuerst nach dem *key* und lösche den Eintrag
  - Löschen: alles dahinter muss um eins nach vorne geschoben werden!

Dictionary mit sortiertem Array

Laufzeiten:

create: $O(1)$

insert: $O(n)$

find: $O(\log n)$

delete: $O(n)$

Können wir alle Operationen schnell machen?

- und das *find* noch schneller?