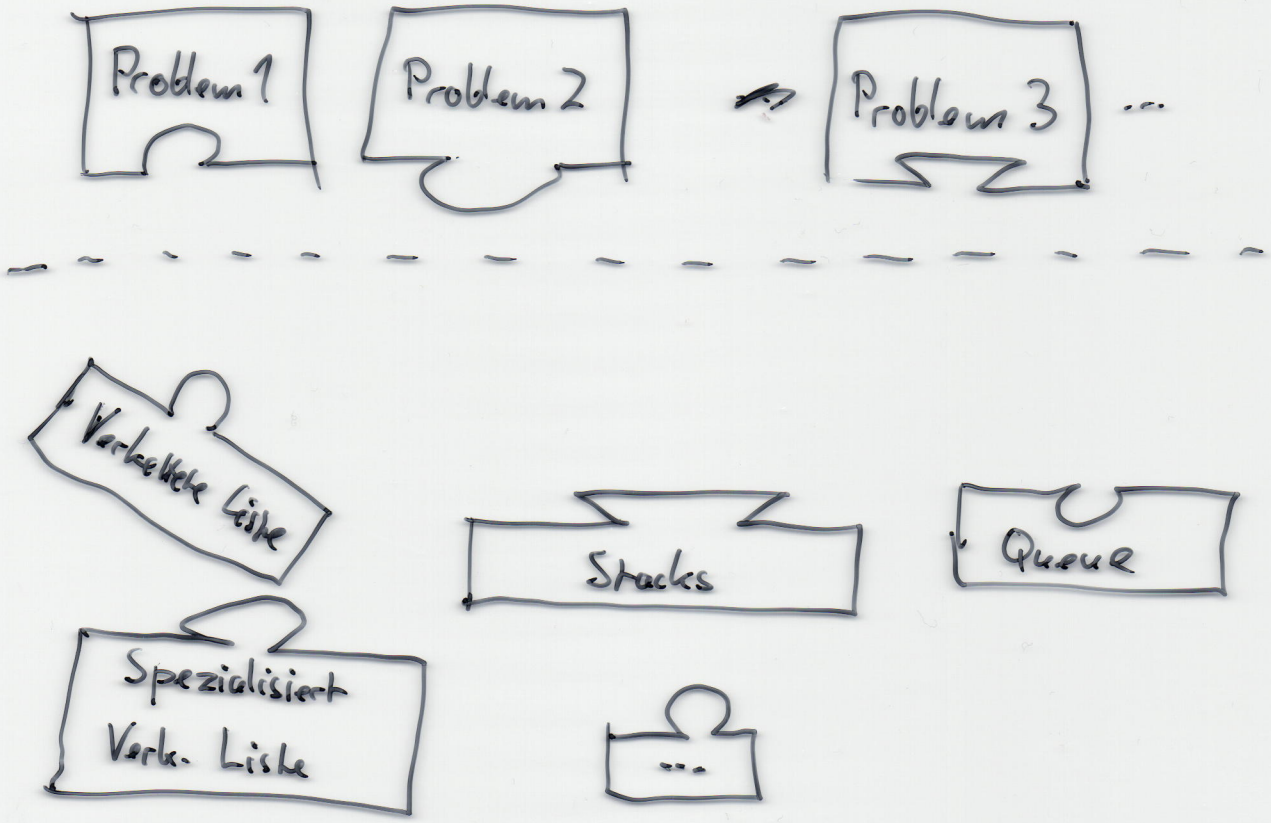
Datenstrukturen

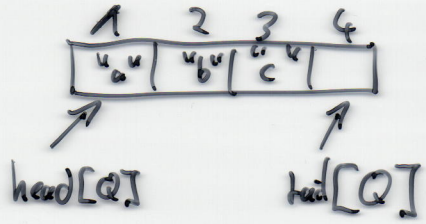
Welche ist die am häufigsten verwendete Datenstruktur?



- Es gibt nicht „die beste" DS
- Für konkrete Anwendung i.d.R. schon!

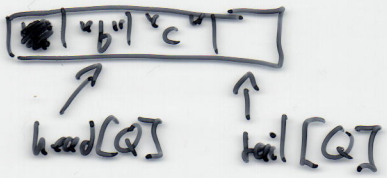
Queues

FIFO

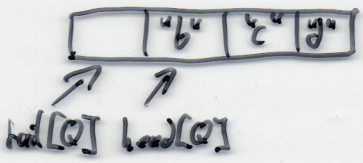


voll! warum? Ein Element mehr und es ist nicht von der leeren Queue unterscheidbar:

DEQUEUE(Q) :



ENQUEUE(Q, "j")



ENQUEUE(Q, "a"): Overflow Error ↯

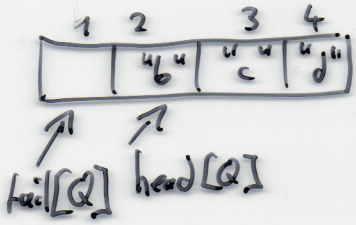
Bisher nicht behandelt! Lösungsansätze?

(1) Array vergrößern!

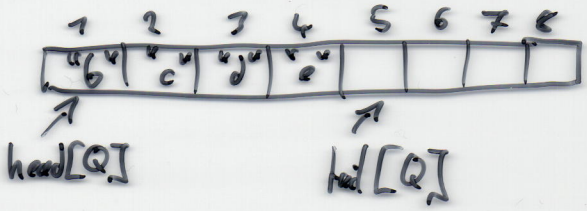
(2) Einfach verk. Liste

(3) Queue von Queues

(1) Array vergrößern



ENQUEUE(Q, "e"):



Vorteile

- Kein Overflow

Nachteile

- $O(n)$ statt $O(1)$ bei ENQUEUE
- u.U. hoher Speicherbedarf

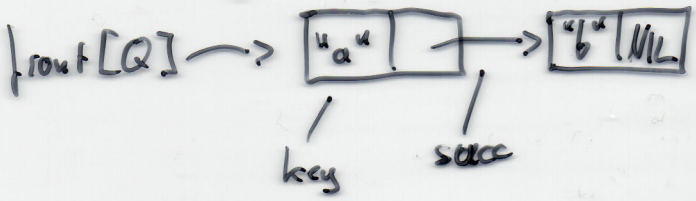
Queue mit Array und Überlaufbehandlung

```
1:  function ENQUEUE(Q, x)
2:      Q[tail[Q]] ← x                                  ▷ Einfügen wie bisher
3:      if tail[Q] = length[Q] then
4:          tail[Q] ← 1
5:      else
6:          tail[Q] ← tail[Q] + 1
7:      end if

8:      if tail[Q] = head[Q] then                       ▷ Überlauf behandeln
9:          arr ← new array(2 · length[Q])
10:         arr[1 . . . length[Q]] ← Q[head[Q] . . . tail[Q] − 1]
11:         Q ← arr
12:         head[Q] ← 1
13:         tail[Q] ← length[Q] + 1
14:         length[Q] ← 2 · length[Q]
15:     end if
16: end function
```

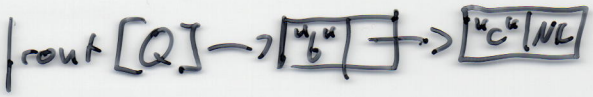
(1) Queue mit einfach verketteter Liste

Idee:



ENQUEUE(Q, "c"):

DEQUEUE(Q)

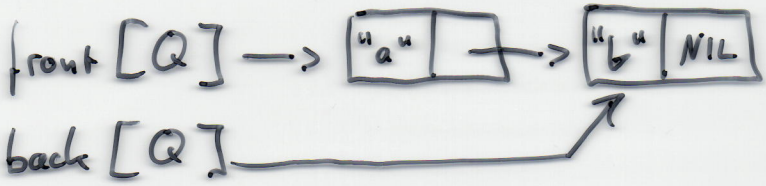


Laufzeit ENQUEUE: $O(n)$, da das letzte Element gesucht werden muss (Zeilen 14–17).

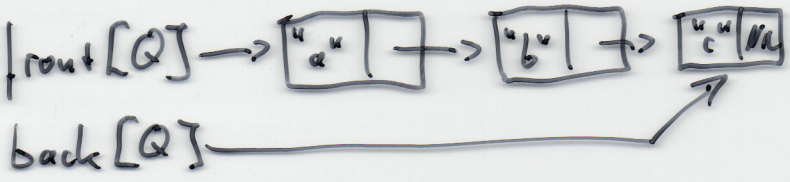
Geht das besser?

Ja!

(1') Queue mit einfach verketteter Liste und Pointer auf letztes Element



ENQUEUE(Q, "c"):



Laufzeit von ENQUEUE? $O(1)$

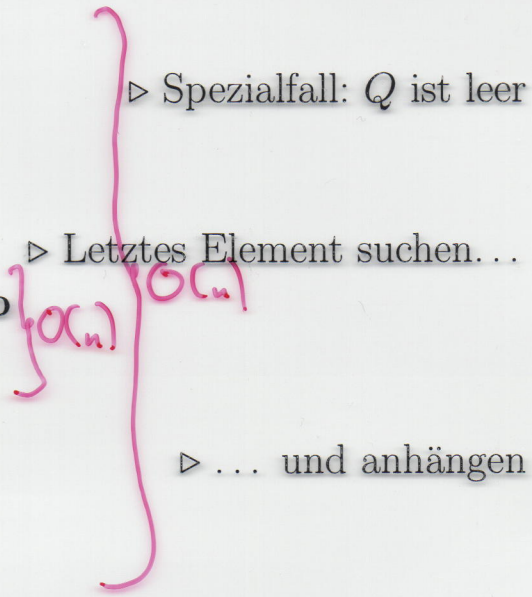
Queue mit einfach verketteter Liste

```
1: function EMPTY(Q)
2:     return front[Q] = NIL
3: end function

4: function FRONT(Q)
5:     return key[front[Q]]
6: end function

7: function DEQUEUE(Q, x)
8:     front[Q] ← succ[front[Q]]
9: end function

10: function ENQUEUE(Q, x)
11:     if front[Q] = NIL then                    ▷ Spezialfall: Q ist leer
12:         front[Q] ← (x, NIL)
13:     else
14:         current ← front[Q]                    ▷ Letztes Element suchen...
15:         while succ[current] ≠ NIL do
16:             current ← succ[current]
17:         end while
18:         succ[current] ← (x, NIL)              ▷ ... und anhängen
19:     end if
20: end function
```



Queue mit einfach verketteter Liste II

```
1: function EMPTY(Q)
2:     return front[Q] = NIL
3: end function

4: function FRONT(Q)
5:     return key[front[Q]]
6: end function

7: function DEQUEUE(Q)
8:     front[Q] ← succ[front[Q]]
9:     if front[Q] = NIL then          ▷ Q ist leer, Update von back[Q]
10:        back[Q] ← NIL
11:    end if
12: end function

13: function ENQUEUE(Q, x)
14:    if back[Q] = NIL then           ▷ Spezialfall: Q ist leer
15:        front[Q] ← back[Q] ← (x, NIL)
16:    else                            ▷ Q ist nicht leer
17:        succ[back[Q]] ← (x, NIL)
18:        back[Q] ← succ[back[Q]]
19:    end if
20: end function
```

O(1) O(1) O(1)

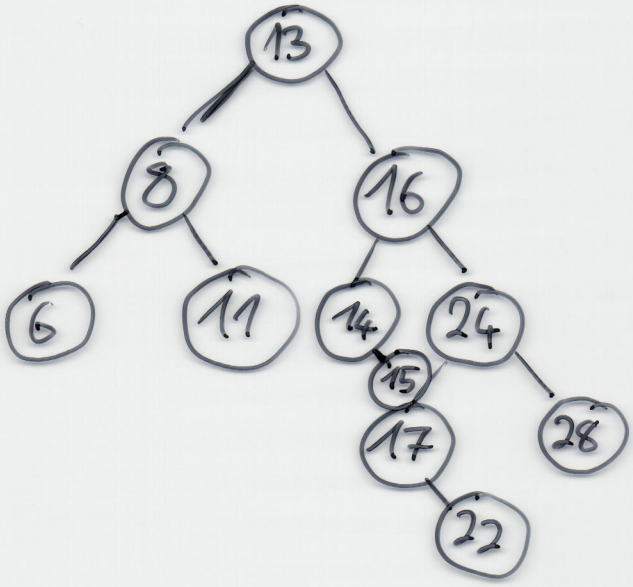
Umsetzung	empty()	front()	dequeue()	enqueue()
(1) Array mit Überlaufbehandlung	$O(1)$	$O(1)$	$O(1)$	$O(n)$
(2) Einfach verk. Liste	$O(1)$	$O(1)$	$O(1)$	$O(n)$
(1') " " " + back-Pointer	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Insgesamt: Richtige DS für richtige Aufgabe wählen!

(oder anpassen!)

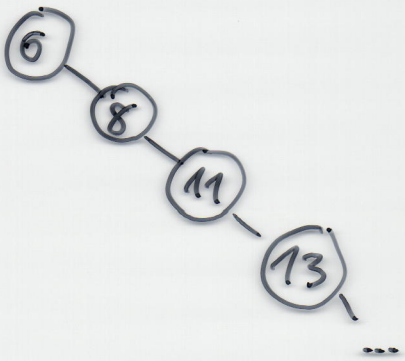
Binärbäume

Einfügen: 13, 8, 6, 16, 11, 24, 14, 17, 22, 28, 15



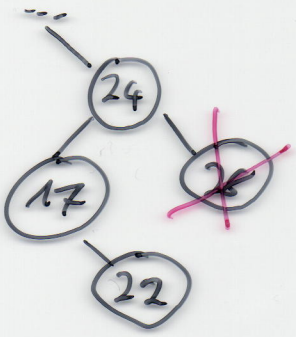
Welche Rolle spielt die Reihenfolge, in der wir einfügen?

6, 8, 11, 13, ...



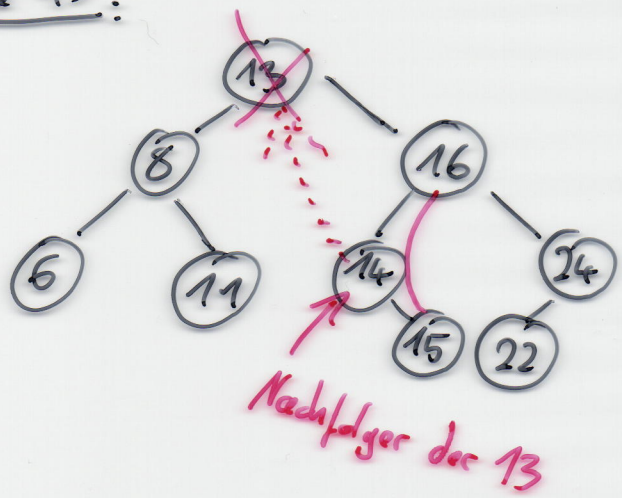
degeneriert!
-> VL/GU 2014
AVL-Bäume!

Lösche die 28 : Einfach rausnehmen, hat keine Kinder.



Lösche 17: Hat nur ein Kind, löschen wie in verketteter Liste

Lösche 13:



Nicht so einfach, hat 2 Kinder.

Kann aber durch ihren Nachfolger ersetzt werden (im rechten Teilbaum). Der Nachfolger kann kein linkes Kind haben, ist also einfach zu entfernen (s.o.).