Übung Algorithmen und Datenstrukturen

Sommersemester 2017

Patrick Schäfer, Humboldt-Universität zu Berlin

Agenda

1. **Sortierte Listen**
2. Stacks & Queues
3. Teile und Herrsche

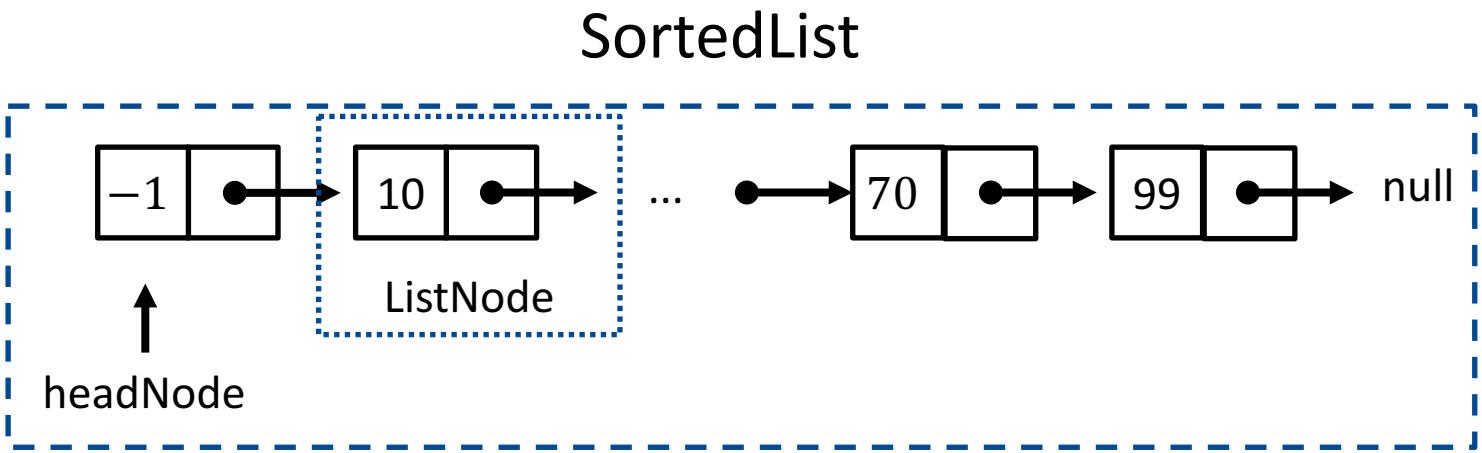
Nächste Woche: Vorrechnen (first-come-first-served)

- Gruppe 5 13-15 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr5U1/
- Gruppe 6 15-17 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr6U1/

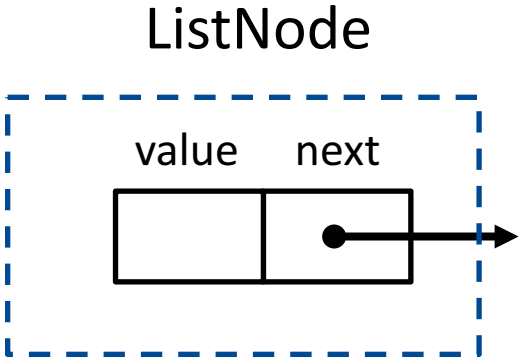
Übung: https://hu.berlin/algodat17

Vorlesung: http://hu.berlin/vl_algodat17

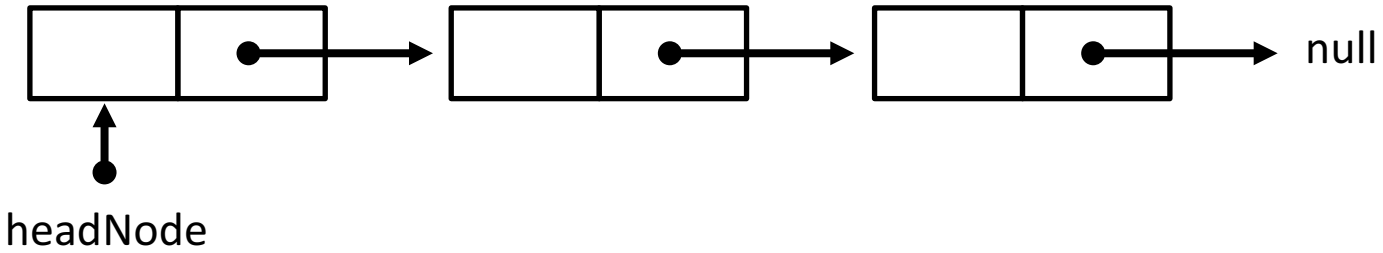
Übungsaufgabe 2.1:
Einfach verkettete, sortierte Liste



ListNode.java



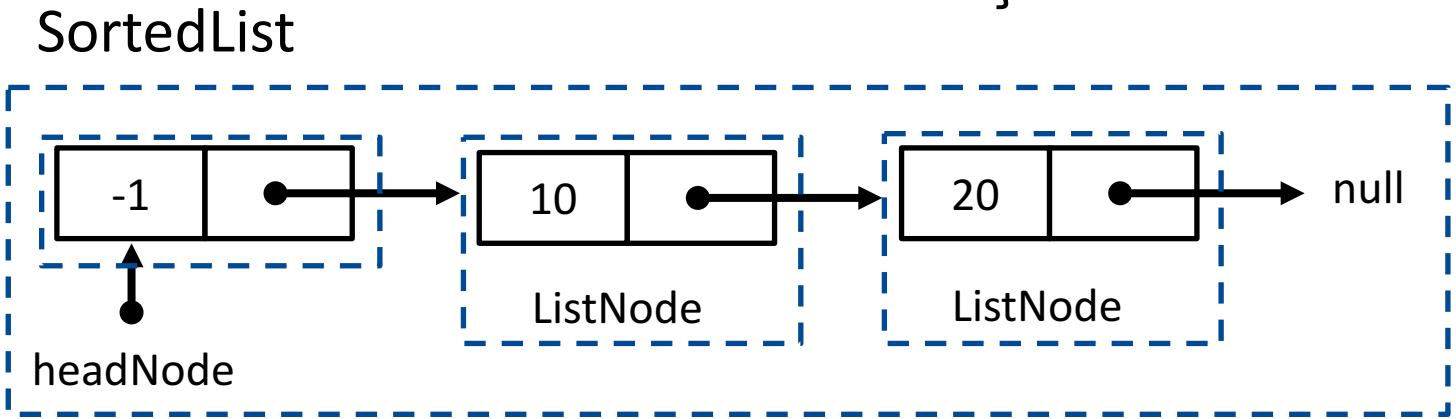

```java
public class ListNode {
  // Der Wert des Knotens
  public int value;

  // Zeiger auf den Nachfolger
  public ListNode next;
}
```



SortedList.java

- Die Liste enthält einen Zeiger headNode auf den ersten Knoten, wobei dieser Zeiger in der leeren Liste auf null zeigt

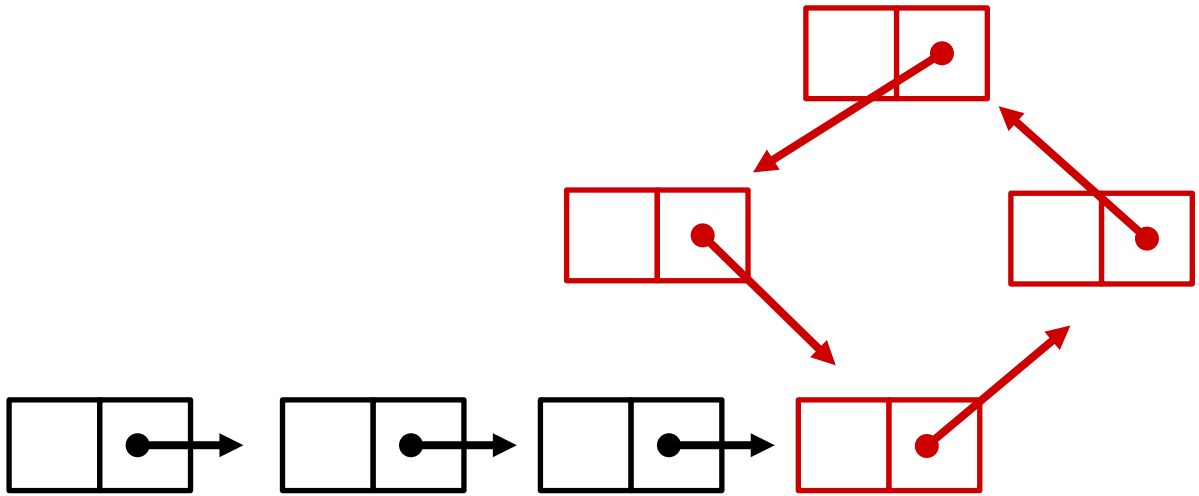
```
public class SortedList {
  // der erste Knoten der Liste
  private ListNode headNode;

  public class ListNode {
    // Der Wert des Knotens
    public int value;

    // Zeiger auf den Nachfolger
    public ListNode next;
  }

}
```



Kreise suchen

Gegeben sei eine einfach verkettete Liste. Nun möchte man seine Liste dahingehend überprüfen, ob es (z.B. durch einen Bug in der Implementierung) zu Schleifen gekommen ist.

Notieren Sie Ihren Algorithmus in Pseudocode und begründen Sie, warum seine Laufzeit in O(n) liegt.



Kreise suchen

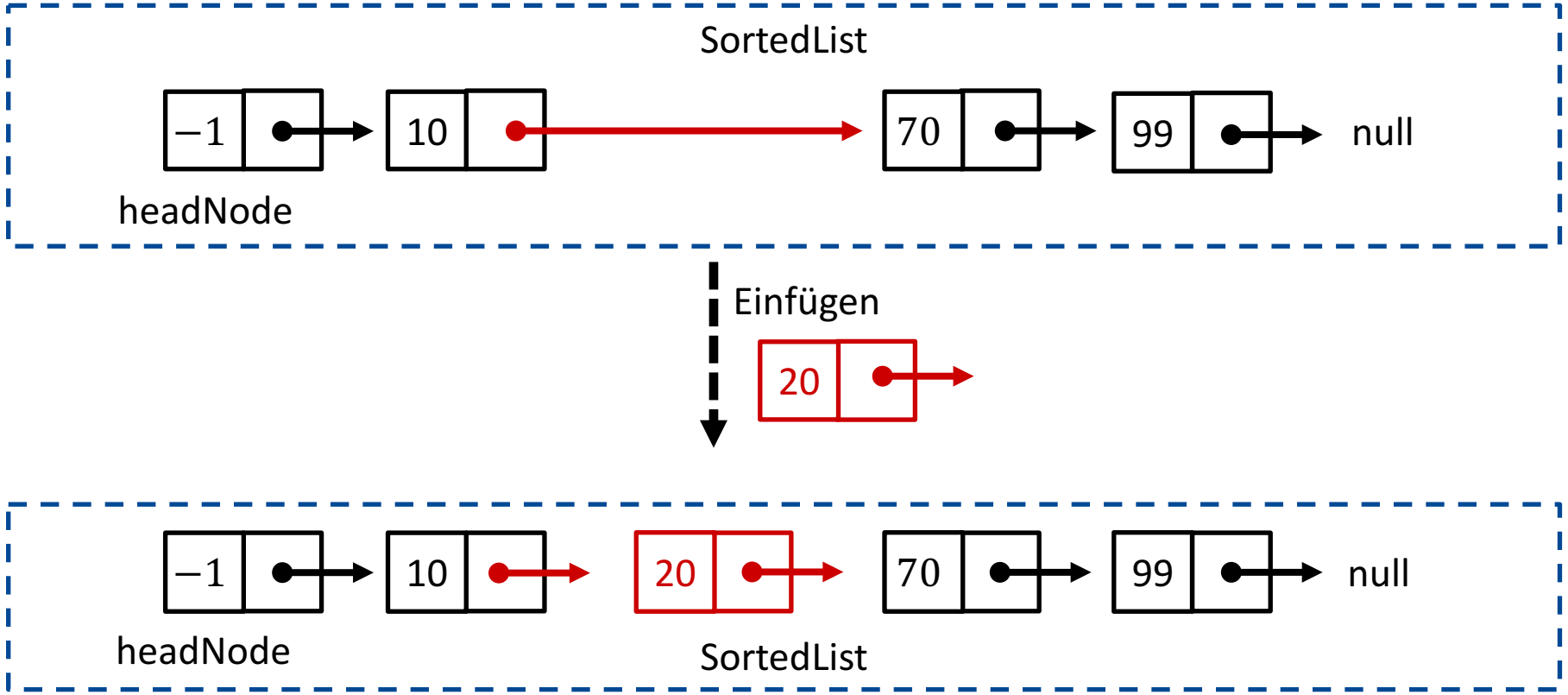
Algorithmus $findeKreis(l)$

Input: Liste l
Output: true, falls Kreis existiert

```
(1)   slow := l.headNode
(2)   fast := l.headNode
(3)   while (fast ≠ NULL and fast.next ≠ NULL) do
(4)     slow := slow.next        // ein Schritt
(5)     fast := fast.next.next   // zwei Schritte
(6)     if (slow = fast) then
(7)       return true;
(8)     end if
(9)   end while
(10)  return false;
```

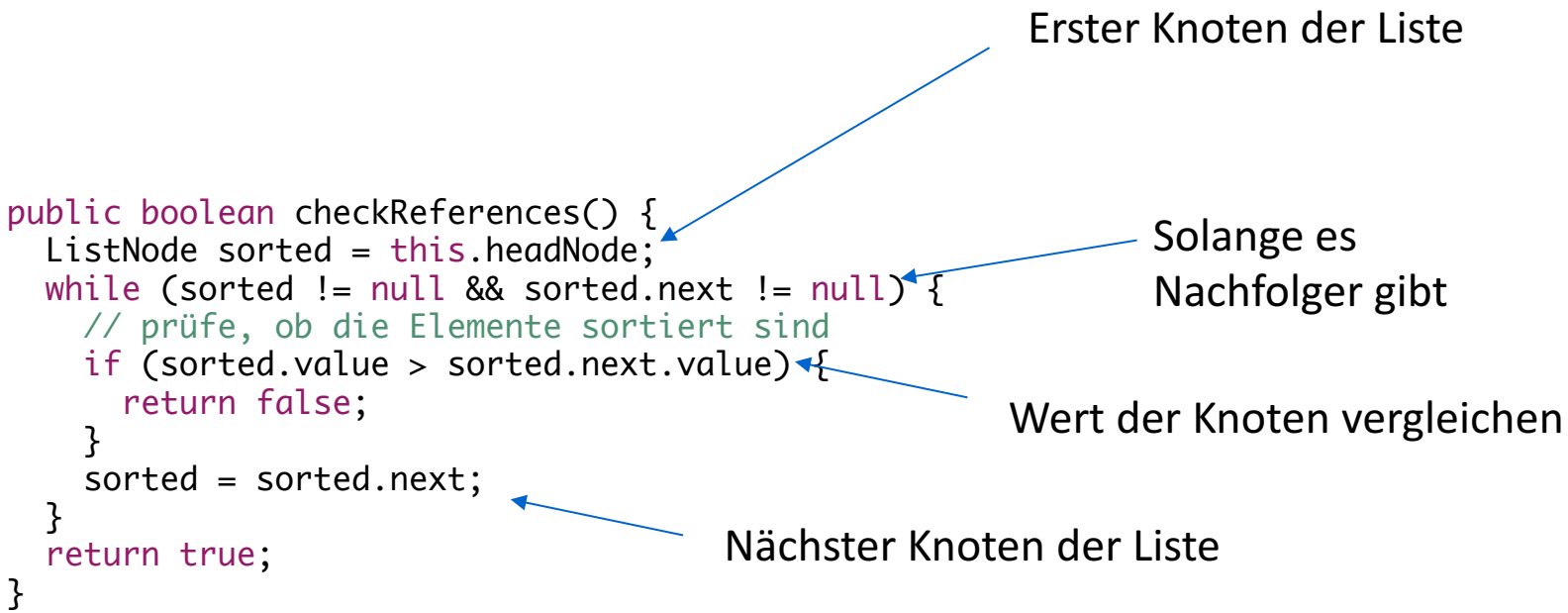
- Zwei Pointer durchlaufen die List. In jeder Runde geht der eine Pointer einen Schritt und der zweite Pointer zwei Schritte.
- Sobald der schnelle Pointer den langsamen Pointer überrundet gibt es eine Schleife.
- Falls der schnelle Pointer bei NULL ankommen ist, gibt es keine Schleife.

Sortiert einfügen



Iterieren von Konten der Liste

- Ihr könnt euch an den Hilfsmethoden in SortedList.java zum Iterieren der Liste orientieren.

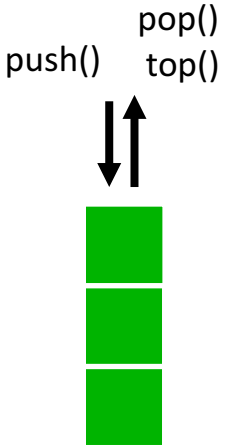



```
public boolean checkReferences() {
  ListNode sorted = this.headNode;
  while (sorted != null && sorted.next != null) {
    // prüfe, ob die Elemente sortiert sind
    if (sorted.value > sorted.next.value){
      return false;
    }
    sorted = sorted.next;
  }
  return true;
}
```

Agenda

1. Sortierte Listen
2. **Stacks & Queues**
3. Teile und Herrsche

Stacks und Queues



- Stack / Stapel: Abstrakter Datentyp mit last-in, first-out (LIFO) Semantik
  - push(value): legt das Element oben auf den Stack.
  - pop(): entfernt das oberste Element vom Stack und gibt es zurück.
  - top() / peek(): gibt das oberste Element zurück (ohne es zu entfernen).
  - isEmpty(): gibt „true“ zurück, falls der Stack leer ist.

  - In Java z.B. java.util.Stack

- Queue / Warteschlange: Abstrakter Datentyp mit first-in, first-out (FIFO) Semantik
  - enqueue(value): hängt das Element an das Ende der Queue an.
  - dequeue(): entfernt das erste Element vom Anfang der Queue und gibt es zurück.
  - head(): liefert das erste Element vom Anfang der Queue ohne es zu entfernen.
  - isEmpty(): gibt „true“ zurück, falls die Queue leer ist.

  - In Java z.B. java.util.ArrayDeque, java.util.LinkedList

Rangieren (Stacks & Queues)

Die Abbildungen zeigen Eisenbahngleise, welche einen Stack bzw. eine Queue mit Überholspur darstellen.

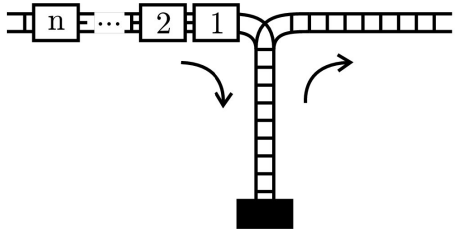


Abbildung 1: Stack

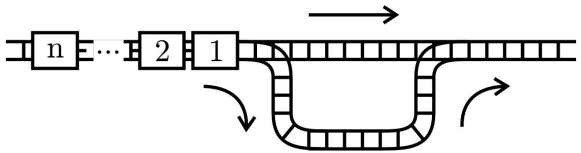


Abbildung 2: Queue

Auf der linken Seite stehen die absteigend nummerierten Waggons n bis 1, die auf dem Rangierwerk so umgestellt werden müssen, dass sie rechts in einer neuen Zusammenstellung herausfahren können. Gibt es für einen Zug mit n = 5 Waggons (5, 4, 3, 2, 1) Rangiermöglichkeiten, so dass die folgenden Waggon-Zusammenstellungen auf der rechten Seite entstehen?

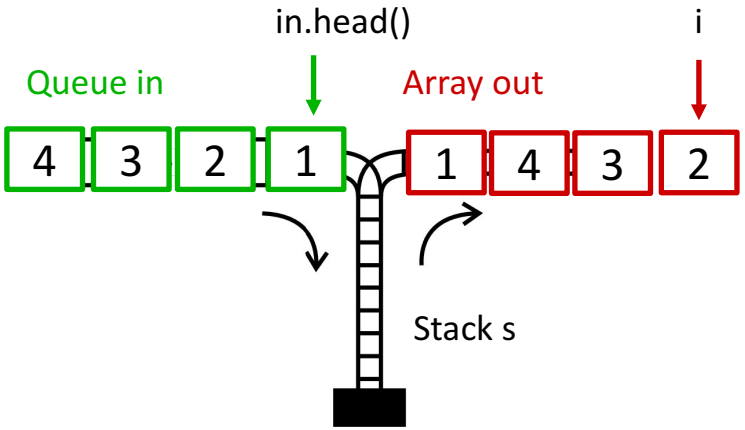
- 5,4,3,2,1 Stack: Ja. Queue: Ja.
- 5,3,1,2,4 Stack: Nein. Queue: Nein.
- 4,2,5,3,1 Stack: Nein. Queue: Ja.
- 1,4,5,3,2 Stack: Ja. Queue: Nein.

Rangieren

Algorithmus $rangiere\ (in, out)$

Input: Int-Queue in, Int-Array out der Länge n
Output: true, falls Reihenfolge existiert

```
(1)   Int-Stack s;
(2)   i := n;
(3)   while i > 0 do
(4)     // Abstellgleis
(5)     while not in.isEmpty() and s.top() < out[i] do
(6)       s.push(in.dequeue());
(7)     end while
(8)     if s.pop() ≠ out[i] then
(9)       return false;
(10)    else
(11)      i := i-1;
(12)    end if
(13)  end while
(14)  return true;
```



Stack: push(), pop(), top(), isEmpty()
Queue: enqueue(), dequeue(), head(), isEmpty()

Wohlgeformte Klammerausdrücke (Stacks & Queues)

Wie findet man heraus, ob ein gegebener String aus öffnenden und schließenden Klammern ein wohlgeformter Klammerausdruck ist?

Ausdruck	wohlgeformt?
([ ])	Ja
(([ ] [ ]))	Ja
( ]) [	Nein
) (	Nein
(( )	Nein

Wohlgeformte Klammerausdrücke

Algorithmus $wohlgeformt(c)$

Input: Character-Array c der Länge |c|=n
Output: true, falls der Ausdruck wohlgeformt ist.

```
(1)   Character-Stack s;
(2)   for i := 1 to n do
(3)     switch (c[i])
(4)       case  '(':
(5)         s.push( ')' );
(6)         break;
(7)       case '[‘:
(8)          s.push( ']' );
(9)          break;
(10)      default:
(11)        if s.isEmpty() or s.pop() ≠ c[i] then
(12)          return false;
(13)        end if
(14)    end switch
(15)  end for
(16)  return s.isEmpty();
```

Alternative: Öffnende & schließende Klammern zählen

Agenda

1. Sortierte Listen
2. Stacks & Queues
3. **Teile und Herrsche**

k-kleinstes Element

- Gegeben ein *unsortiertes* Array von n Elementen, finde das k-kleinste Element.
  - z.B.: Median: k = $\lceil n/2 \rceil$, für n ungerade

- Beispiel, k=4:

- Triviale Lösungen:
  - Sortieren und k-tes Element zurückgeben.
    - $\Theta(n \log n)$ (MergeSort).
  - k-mal das Minimum bestimmen / entfernen.
    - $\Theta(k\ n)$ (SelectionSort), $O(n^2)$

- Besser Lösung?

Triviale-Lösung

Algorithmus $kSmallest(A, k)$

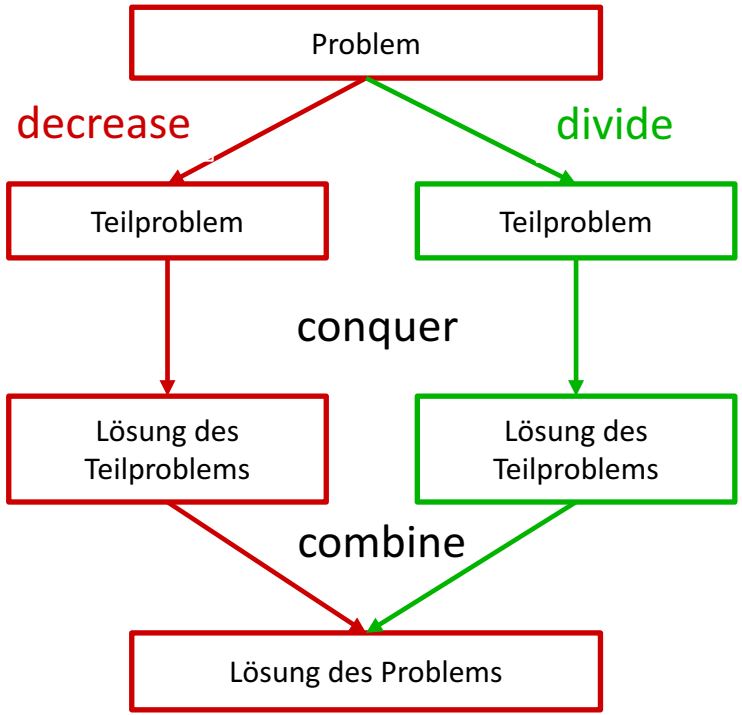
Input: Int-Array A mit $n > 1$ Objekten, k
Output: k-kleinstes Element von A

```
(1)   for i := 1 to k do
(2)     minIndex := i;
(3)     for j := i+1 to n do
(4)       if list[j] < list[minIndex] then
(5)         minIndex := j;
(6)       end if
(7)     end for
(8)     swap list[i] and list[minIndex];
(9)   end for
(10)  return list[k];
```

Teile und Herrsche

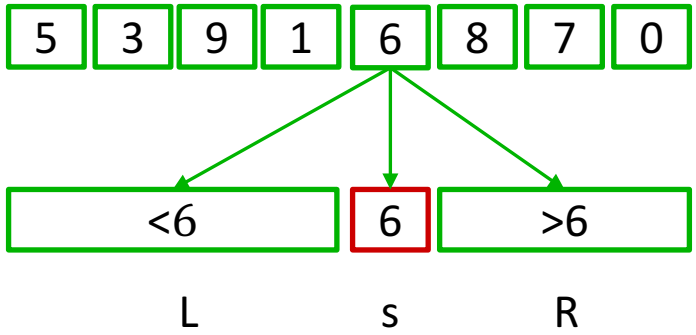
- Paradigma für den Entwurf von effizienten Algorithmen.
- Das Problem wird in kleinere, einfachere Teilprobleme zerlegt, bis man die Lösung (be-)herrscht. Anschließend wird aus den Teillösungen die Gesamtlösung abgeleitet.

1) Decrease and Conquer:
   - Es wird genau ein Teilproblem gelöst (z.B. Binäre Suche).
2) Divide and Conquer:
   - Es werden mindestens zwei Teilprobleme gelöst (z.B. MergeSort, QuickSort).



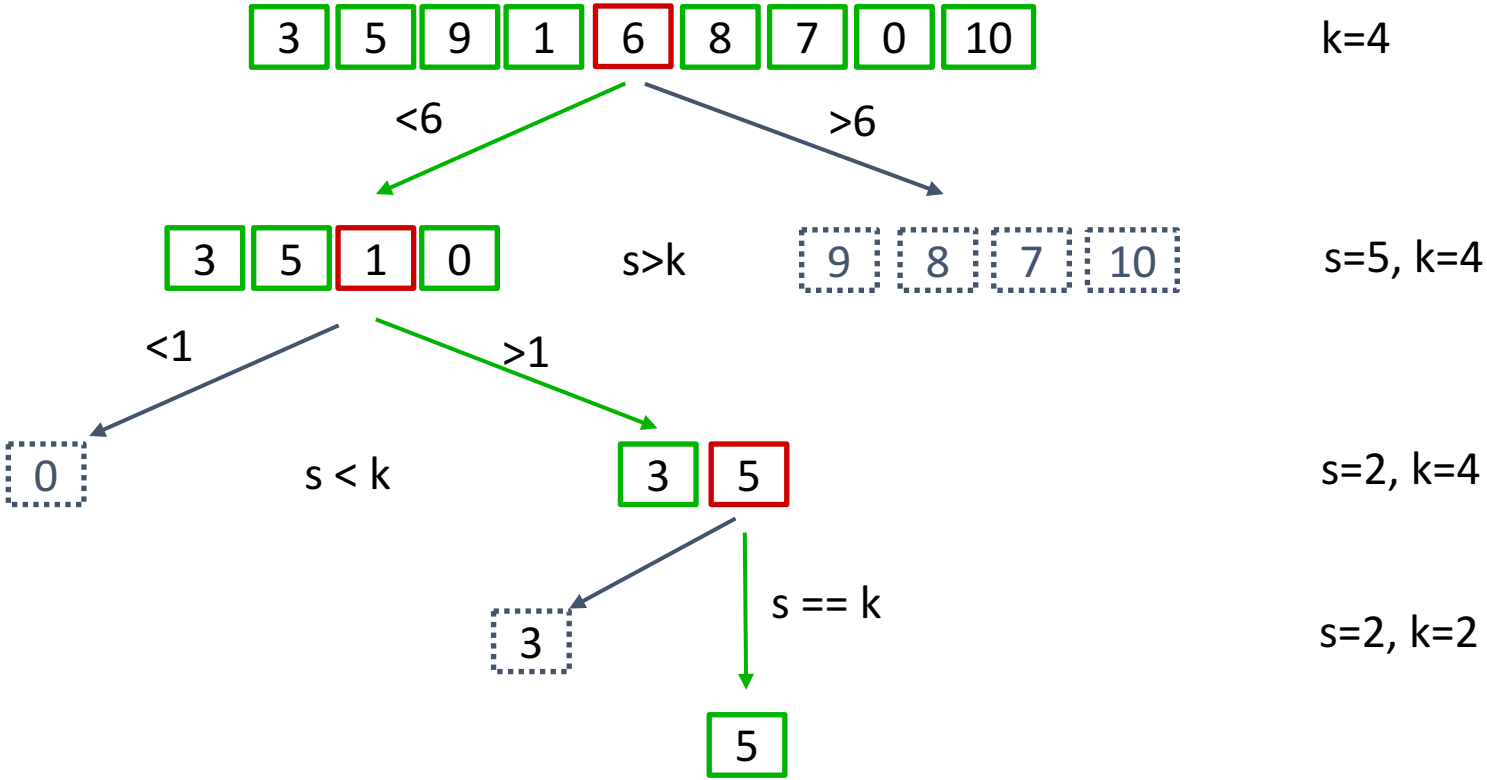
Decrease and Conquer: Idee

- Wähle ein Pivot-Element und partitioniere die Liste in zwei Teile L und R (kleiner und größer als Pivot-Element):



- Sei s die Position des Pivot Elements:
  - Falls s = k, gib das Pivot-Element zurück.
  - Falls s > k, suche das k-kleinste Element in L
  - Falls s < k, suche das (k-s)-kleinste Element in R

- Beispiel, k=4:



Decrease and Conquer

Algorithmus $kSmallest(A, k)$

Input: Int-Array A mit $n > 1$ Objekten, Position k
Output: k-kleinstes Element von A

```
(1)   if k >|A| then
(2)     return null;
(3)   end if
(4)   p = ⌈|A|/2⌉ ;                 // Index des Pivot-Elements
(5)   A_L , A_R := partition(A, p, k);   // Teile A anhand des Pivot-Elements
(6)   s := |A_L|+1
(7)   if s = k then
(8)     return A[s];
(9)   else if s < k then
(10)    return kSmallest(A_L, k);
(11)  else
(12)    return kSmallest(A_R, k-s);
(13)  end if
```

Laufzeit

- Best Case:
  - Eine Iteration : $O(1)$
- Worst Case:
  - $O(n^2)$
  - Wann tritt der ein?
- Average Case:
  - $O(n)$ (Ohne Beweis)
  - Intuition: jeweils ungefähr halbieren der Elemente

$$\approx n\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots\right) = n\sum_{i=0}^{\log n}\left(\frac{1}{2}^{n}\right) \leq \frac{n}{1-\frac{1}{2}} = 2n \in O(n)$$

(Geometrische Reihe)

Nächste Woche: Vorrechnen!

- Jeder muss eine Aufgabe vorrechnen.
- Ihr dürft euch aussuchen, was ihr wann vorrechnen wollt (first-come-first-served):
  - Gruppe 5 13-15 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr5U1/
  - Gruppe 6 15-17 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr6U1/