Übung Algorithmen und Datenstrukturen

Sommersemester 2017

Marc Bux, Humboldt-Universität zu Berlin

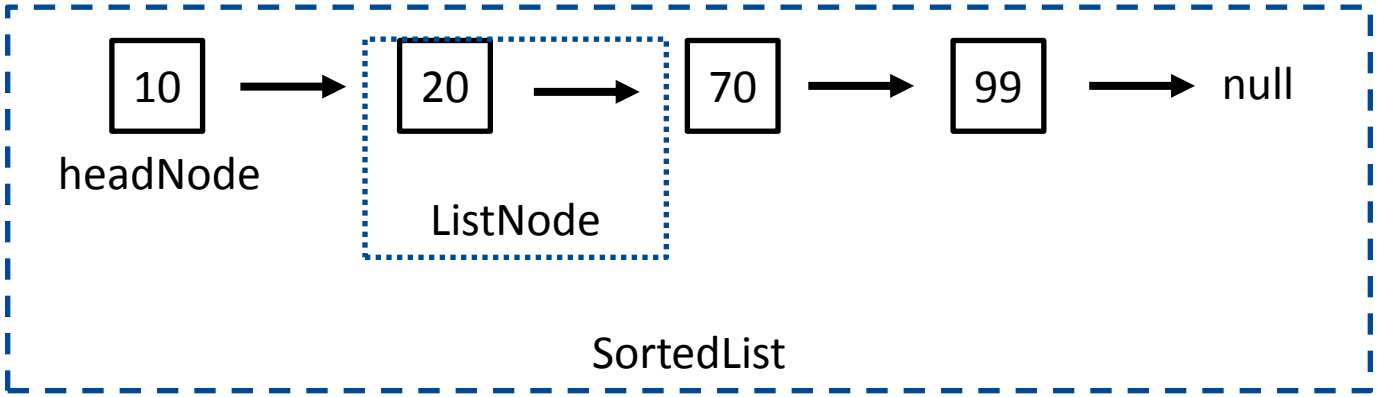
Agenda

1. (Sortierte) Listen
2. Stacks & Queues
3. Datenstrukturen
4. Rekursion und vollständige Induktion

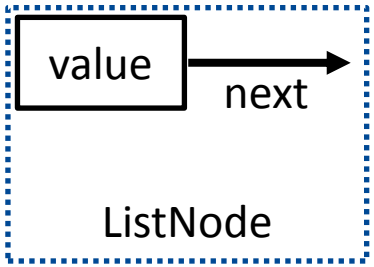
Agenda

1. (Sortierte) Listen
2. Stacks & Queues
3. Datenstrukturen
4. Rekursion und vollständige Induktion

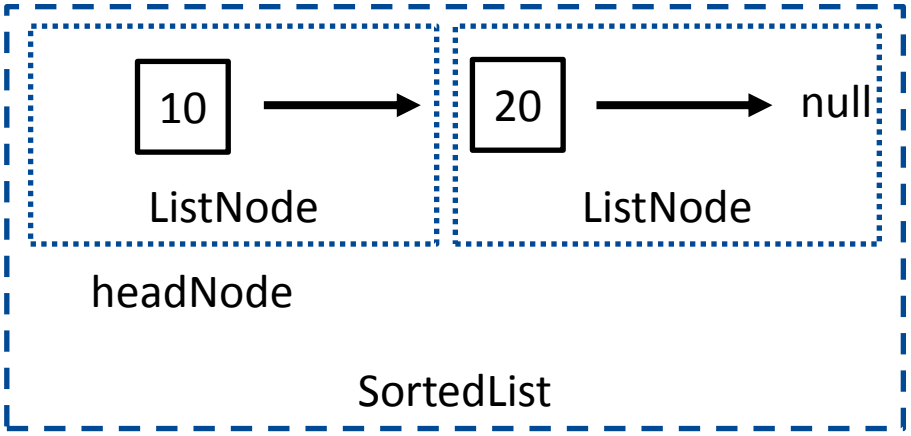
Einfach verkettete (sortierte) Liste



ListNode und SortedList

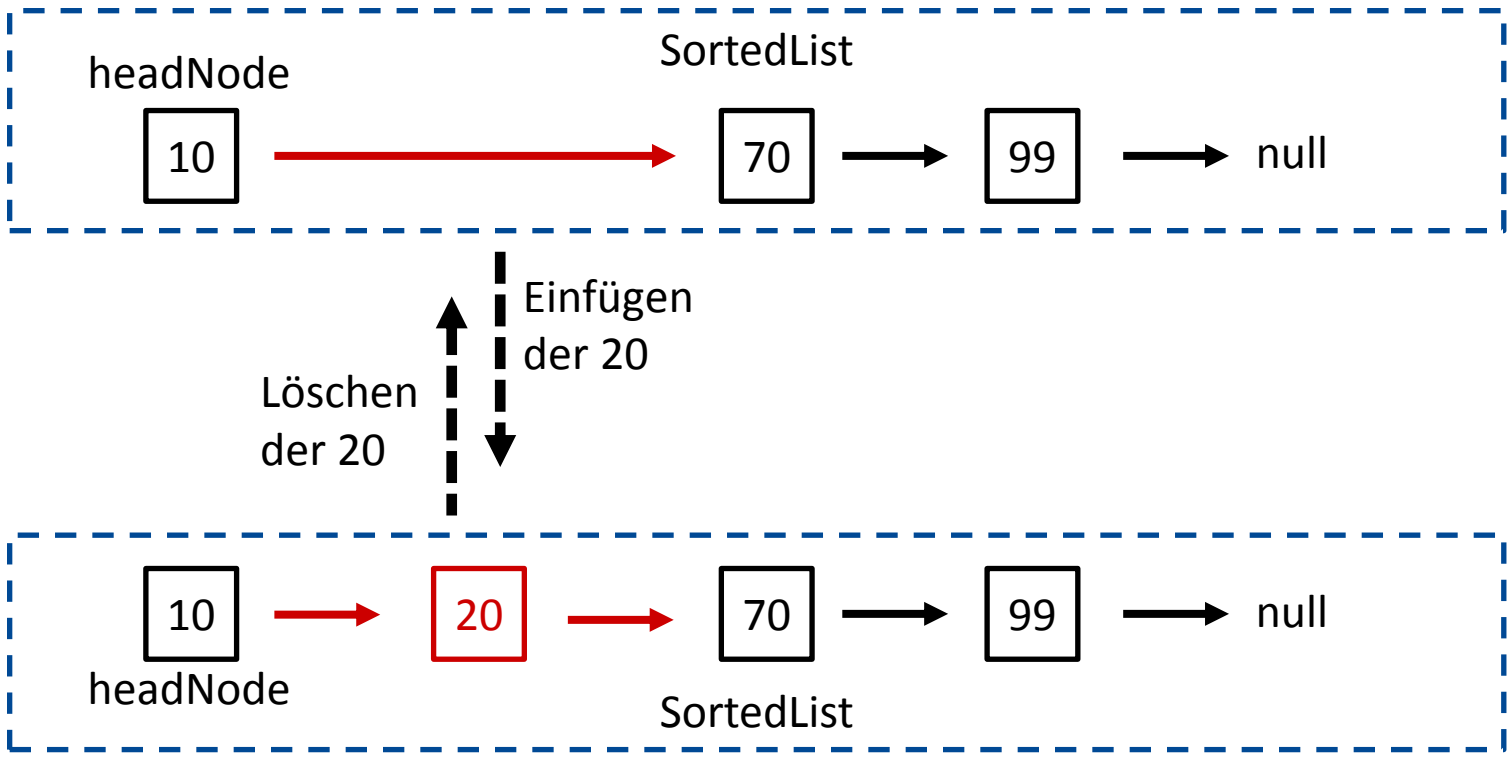



```
public class ListNode {
  // Der Wert dieses Knoten
  public int value;

  // Zeiger auf den Nachfolger
  public ListNode next;

}
```



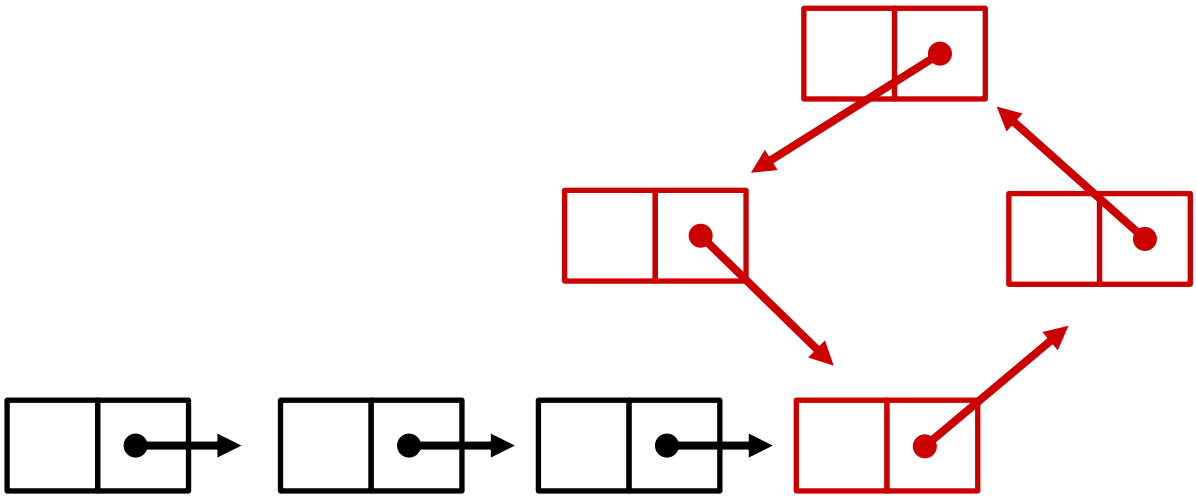

```
public class SortedList {

  // der erste Knoten der Liste
  public ListNode headNode;

}
```

Einfügen


SortedList
headNode
10
70
99
null
Einfügen
der 20
Löschen
der 20
10
20
70
99
null
headNode
SortedList

Übungsaufgabe: 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. Wie geht das möglichst effizient?



Lösung: Kreise suchen

- Zwei Pointer durchlaufen die Liste.
- Der eine Pointer geht immer einen Schritt, der andere zwei.
- Falls der langsame Pointer den schnellen Pointer irgendwann einholt, so gibt es eine Schleife, ansonsten nicht.

Algorithmus $findeKreis(l)$

Input: Liste list
Output: true, falls list einen Kreis enthält; ansonsten false

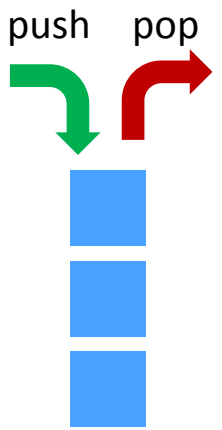
```
(1)  slow := list.headNode;
(2)  fast := list.headNode;
(3)  while (fast ≠ null and fast.next ≠ null) do
(4)     slow := slow.next;          // ein Schritt
(5)     fast := fast.next.next;     // zwei Schritte
(6)     if (slow = fast) then
(7)        return true;
(8)     endif
(9)  endwhile
(10) return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

Agenda

1. (Sortierte) Listen
2. Stacks & Queues
3. Datenstrukturen
4. Rekursion und vollständige Induktion

Stacks und Queues

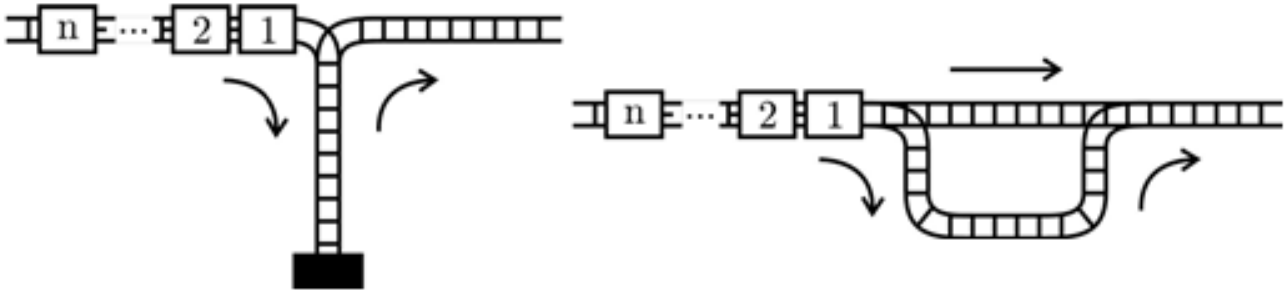
- Stack / Stapel: dynamische last-in, first-out (LIFO) Datenstruktur
  - push(value): legt den Wert value oben auf dem Stack ab
  - pop(): entfernt den obersten Wert vom Stack und gibt ihn zurück
  - top() bzw. peek(): liefert den obersten Wert vom Stack ohne ihn dabei zu entfernen
  - isEmpty(): gibt *true* genau dann zurück, wenn der Stack leer ist



- Queue / Warteschlange: dynamische first-in, first-out (FIFO) Datenstruktur
  - enqueue() bzw. add(value): hängt den Wert value an das Ende der Queue an
  - dequeue() bzw. remove(value): entfernt den ersten Wert vom Anfang der Queue und gibt ihn zurück
  - head() bzw. peek(): liefert den ersten Wert vom Anfang der Queue zurück ohne ihn dabei zu entfernen
  - isEmpty(): gibt *true* genau dann zurück, wenn die Queue leer ist

Übungsaufgabe: Rangieren

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



- 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?

1) 5,4,3,2,1 — Stack: Ja; Queue: Ja
2) 5,3,1,2,4 — Stack: Nein; Queue: Nein
3) 4,2,5,3,1 — Stack: Nein; Queue: Ja
4) 1,4,5,3,2 — Stack: Ja; Queue: Nein

Übungsaufgabe: Stacks und Queues

Es soll eine Queue Q unter Verwendung von zwei Stacks $S1$, $S2$ mit den Funktionen `push(value)`, `pop()`, `top()` und `isEmpty()` und keiner sonstigen Datenstruktur implementiert werden.

1. Entwerfen Sie Algorithmen für die folgenden vier Funktionen einer Datenstruktur Queue (nur unter Verwendung zweier Stacks mit den vier o.g. Methoden, d.h., es darf keine weitere interne Listenstruktur verwendet werden). Die Algorithmen sollten insgesamt möglichst effizient sein. Notieren Sie Ihre Algorithmen als Pseudocode.
   - `enqueue(value)`: Hängt den Wert `value` an das Ende der Queue an.
   - `dequeue()`: Entfernt den ersten Wert vom Anfang der Queue und gibt ihn zurück. Wenn die Queue leer ist, wird der Wert `null` zurückgegeben.
   - `head()`: Liefert den ersten Wert vom Anfang der Queue zurück ohne ihn dabei zu entfernen. Wenn die Queue leer ist, wird der Wert `null` zurückgegeben.
   - `isEmpty()`: Gibt `true` zurück, falls die Queue keine Werte enthält und `false` andernfalls.
2. Analysieren Sie die Laufzeit für die vier genannten Funktionen in Abhängigkeit der Größe der Queue, d.h. der Anzahl n ihrer Werte. Nehmen Sie dabei an, dass alle Funktionen der Stacks in $\mathcal{O}(1)$ laufen.

Agenda

1. (Sortierte) Listen
2. Stacks & Queues
3. Datenstrukturen
4. Rekursion und vollständige Induktion

Datenstrukturen

- elementarer (bzw. primitiver) Datentyp: begrenzter Wertebereich
  - Beispiele: Integer, Character, Boolean, Double, …
- Datenstruktur: Objekt zur Speicherung, Verwaltung und Bereitstellung von Daten
  - Beispiele: Arrays, Listen, Stacks, Queues, …
  - unterschiedliche Datenstrukturen erlauben unterschiedliche Operationen mit unterschiedlicher Effizienz
  - dynamische DS: kann beliebig vergrößert und verkleinert werden

Datenstruktur	Einfügen	Suchen	Java-Interface	Java-Klassen (Auswahl)
Array / String	N/A	$O(n)$		String
Liste	$O(1)$	$O(n)$	List	LinkedList, ArrayList
Stack	$O(1)$	$O(n)$		Stack
Queue	$O(1)$	$O(n)$	Queue	LinkedList

Agenda

1. (Sortierte) Listen
2. Stacks & Queues
3. Datenstrukturen
4. Rekursion und vollständige Induktion

Übungsaufgabe: Rekursion

Gegeben Sei der folgende Algorithmus:

Algorithmus $M(i)$

Input: $i \in \mathbb{N}$
Output: Zahl x

```
(1)  if i = 0 then
(2)      return i;
(3)  else
(4)      return i · M(i − 1);
(5)  endif
```

1. Was berechnet der Algorithmus $M(i)$? Geben Sie hierzu eine Formel an.
2. Analysieren Sie die Laufzeit des Algorithmus.
3. Beweisen Sie die Korrektheit des Algorithmus.

Vollständige Induktion

- Beweismethode für den Beweis einer Aussage $A(n)$ über natürliche Zahlen, d.h. $n \in \mathbb{N}$
- Induktionsanfang (IA): Beweise, dass $A(0)$ wahr ist
- Induktionsschritt (IS): Beweise, dass wenn $A(k)$ wahr ist (IV), auch $A(k+1)$ (IB) wahr sein muss.
  - Induktionsvoraussetzung (IV): Die Aussage $A(k)$ ist wahr für ein bestimmtes $k \in \mathbb{N}$
  - Induktionsbehauptung (IB): Die Aussage $A(k+1)$ ist wahr (unter Verwendung der IV)
- Induktionsschluss: Gültigkeit der Aussage $A(n)$ für alle $n \in \mathbb{N}$ folgt aus IA und IS
- vgl. Dominoeffekt
- Wichtiges Werkzeug für Korrektheitsbeweise rekursiver Algorithmen

Korrektheitsbeweis

Behauptung: $M(i) = 0$

Beweis durch vollständige Induktion über i:

IA (i $= 0$): $M(i) \overset{(2)}{=} i = 0$

IS ($i \to i + 1$):

IV: $M(i) = 0$

IB: $M(i + 1) = 0$

$$M(i+1) \overset{(4)}{=} (i+1) \cdot M\big((i+1) - 1\big)$$
$$= (i+1) \cdot M(i) \overset{\text{IV}}{=} (i+1) \cdot 0 = 0$$ ■

Übungsaufgabe: Rekursion

Betrachten Sie den folgenden rekursiven Algorithmus **P**:

Algorithmus P(A, i, j)

Input: Array A der Länge $|A| = n$ von Zahlen aus $\{0, 1, \ldots, 9\}$,
natürliche Zahlen i, j mit $1 \leq i \leq n + 1$
Output: eine natürliche Zahl

```
1: if i = n + 1 then
2:     return j;
3: else
4:     return P(A, i + 1, 10j + A[i]);
5: end if
```

1. Was berechnet der Algorithmus **P** bei Eingabe des Arrays $[1, 2, 3]$ sowie i=1 und j=0? Was berechnet der Algorithmus **P** allgemein bei Eingabe eines Arrays A mit n Zahlen aus $\{0, 1, \ldots, 9\}$ sowie $i = 1$ und $j = 0$?

2. Analysieren Sie die Laufzeit des Algorithmus in Abhängigkeit von n.

3. Geben Sie einen iterativen Algorithmus für das Berechnungsproblem als Pseudocode an.

Ausblick

- zu nächster Woche:
  - Mit Aufgabenblatt 2 auseinandersetzen
- nächste Woche:
  - Aufgabenblatt 1 durchrechnen