

# Datenstrukturen und Algorithmen

## Vorlesung 3: Elementare Datenstrukturen (K10)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2  
Software Modeling and Verification Group

<http://www-i2.informatik.rwth-aachen.de/i2/dsa112/>

13. April 2012

# Übersicht

- 1 Abstrakte Datentypen
- 2 Stapel und Warteschlangen
- 3 Verkettete Listen
  - Einfach verkettete Listen
  - Doppelt verkettete Listen
- 4 Binäre Bäume
  - Traversierungen

# Übersicht

- 1 Abstrakte Datentypen
- 2 Stapel und Warteschlangen
- 3 Verkettete Listen
  - Einfach verkettete Listen
  - Doppelt verkettete Listen
- 4 Binäre Bäume
  - Traversierungen

# Abstrakte Datentypen

## Abstrakter Datentyp (ADT)

Ein abstrakter Datentyp besteht aus:

- ▶ Einer Datenstruktur (Menge von Werten) und
- ▶ einer Menge von Operationen darauf.  
(z. B. Konstruktor, Zugriffs- und Bearbeitungsfunktionen)

## Beispiele

Baum, Kellerspeicher (stack), Liste, Warteschlange (queue),  
Prioritätswarteschlange (priority queue), Wörterbuch ...

# Datenkapselung

Unterscheide zwischen

**Spezifikation des ADTs:** wie sich die Datenobjekte **verhalten**, und

**Implementierung:** wie dieses Verhalten programmtechnisch erreicht wird.

# Datenkapselung

Unterscheide zwischen

**Spezifikation des ADTs:** wie sich die Datenobjekte **verhalten**, und

**Implementierung:** wie dieses Verhalten programmtechnisch erreicht wird.

## Datenkapselung (data encapsulation)

Dieses Paradigma wird *Kapselung* (oder: Datenabstraktion) genannt:

- ▶ Daten sind außerhalb des ADT nur über wohldefinierte Operationen zugänglich.
- ▶ Die Repräsentation der Daten ist nur für die Implementierung relevant.

# Spezifikation von ADTs (I)

## Spezifikation eines ADTs

- ▶ Beschreibt **wie** sich die Operationen auf den Daten **verhalten**;
- ▶ nicht jedoch die interne Repräsentation der Daten,
- ▶ genauso wenig wie die Implementierung der Operationen.

# Spezifikation von ADTs (I)

## Spezifikation eines ADTs

- ▶ Beschreibt **wie** sich die Operationen auf den Daten **verhalten**;
- ▶ nicht jedoch die interne Repräsentation der Daten,
- ▶ genauso wenig wie die Implementierung der Operationen.

Beschreibung der Auswirkung von Operationen durch logische Aussagen:

## Vorbedingung (precondition)

Aussage, die vor Aufruf der Operation gelten muss.  
(Verpflichtung des Benutzers!)

## Nachbedingung (postcondition)

Aussage, die als Ergebnis der Operation gelten wird.

⇒ Grundlage für die Argumentation über die **Korrektheit** des ADTs.



# Spezifikation von ADTs (II)

## Beispiel

Die Operation `void push(Stack s, int e)` hat

- ▶ die Vorbedingung: `true` (d. h. leere Aussage) und
  - ▶ die Nachbedingung: oberster Eintrag von `s` ist `e`.
- 
- ▶ ADTs sind *durch ihre Spezifikation* festgelegte „Standard“-Komponenten zum Aufbau unserer Algorithmen.

# Implementierung von ADTs

## Implementierung eines ADTs

- ▶ Beschreibt die interne **Repräsentation** der Daten, und
- ▶ die genaue **Implementierung** der Operationen.

# Implementierung von ADTs

## Implementierung eines ADTs

- ▶ Beschreibt die interne **Repräsentation** der Daten, und
- ▶ die genaue **Implementierung** der Operationen.

Verschiedene Implementierungen von ADTs *der selben Spezifikation* ermöglichen es uns die **Performance zu optimieren**.

⇒ Grundlage für die Argumentation über die **Effizienz** des ADTs.

# Implementierung von ADTs

## Implementierung eines ADTs

- ▶ Beschreibt die interne **Repräsentation** der Daten, und
- ▶ die genaue **Implementierung** der Operationen.

Verschiedene Implementierungen von ADTs *der selben Spezifikation* ermöglichen es uns die **Performance zu optimieren**.

⇒ Grundlage für die Argumentation über die **Effizienz** des ADTs.

## Beispiel

Die Operation `push(Stack s, int e)` als Array-Implementierung:

```
1 void push(Stack s, int e) {  
2     s.top = s.top + 1;  
3     s[s.top] = e;  
4 }
```

# Effizienz von Implementierungen

*Die Effizienz einer ADT-Implementierung ist entscheidend.*

1. Die **Zeitkomplexität** der Operationen auf dem ADT.
  - ▶ Einfügen von Elementen,
  - ▶ Löschen von Elementen,
  - ▶ Suchen von Elementen.
2. Die **Platzkomplexität** der internen Datenrepräsentation.

# Effizienz von Implementierungen

*Die Effizienz einer ADT-Implementierung ist entscheidend.*

1. Die **Zeitkomplexität** der Operationen auf dem ADT.
  - ▶ Einfügen von Elementen,
  - ▶ Löschen von Elementen,
  - ▶ Suchen von Elementen.
2. Die **Platzkomplexität** der internen Datenrepräsentation.

Üblicherweise ein Kompromiss zwischen Zeit- und Platzeffizienz:

- ▶ Schnelle Operationen benötigen in der Regel zusätzlichen Speicherplatz.
- ▶ Platzsparende Repräsentationen führen oft zu langsameren Operationen.

## Beispiel

Implementierungen einer Prioritätswarteschlange (später).

# Übersicht

- 1 Abstrakte Datentypen
- 2 **Stapel und Warteschlangen**
- 3 Verkettete Listen
  - Einfach verkettete Listen
  - Doppelt verkettete Listen
- 4 Binäre Bäume
  - Traversierungen

# Beispiele für ADTs: Stapel

## Stapel (stack)

Ein **Stapel** (Kellerspeicher) speichert eine Ansammlung von Elementen und bietet folgende Operationen:

- ▶ `bool isEmpty(Stack s)` gibt `true` zurück, wenn `s` leer ist und andernfalls `false`.
- ▶ `void push(Stack s, int e)` fügt das Element `e` in den Stapel `s` ein.
- ▶ `int pop(Stack s)` entfernt das zuletzt eingefügte Element und gibt es zurück; `pop(s)` benötigt einen nicht-leeren Stapel `s`.



# Beispiele für ADTs: Warteschlangen

## Warteschlange (queue)

Eine **Warteschlange** speichert eine Ansammlung von Elementen und bietet folgende Operationen:

- ▶ `bool isEmpty(Queue q)` gibt `true` zurück, wenn `q` leer ist, andernfalls `false`.
- ▶ `void enqueue(Queue q, int e)` fügt das Element `e` in die Warteschlange `q` eine.
- ▶ `int dequeue(Queue q)` entfernt das schon am längsten in der Warteschlange vorhandene Element und gibt es zurück; benötigt daher eine nicht-leere Warteschlange `q`.

# Beispiele für ADTs: Warteschlangen

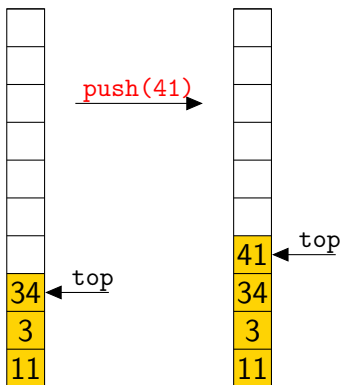
## Warteschlange (queue)

Eine **Warteschlange** speichert eine Ansammlung von Elementen und bietet folgende Operationen:

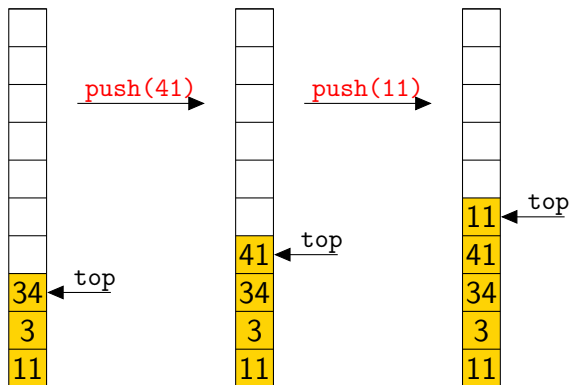
- ▶ `bool isEmpty(Queue q)` gibt `true` zurück, wenn `q` leer ist, andernfalls `false`.
- ▶ `void enqueue(Queue q, int e)` fügt das Element `e` in die Warteschlange `q` eine.
- ▶ `int dequeue(Queue q)` entfernt das schon am längsten in der Warteschlange vorhandene Element und gibt es zurück; benötigt daher eine nicht-leere Warteschlange `q`.

Ein Stapel bietet **LIFO** (last-in first-out) Semantik, eine Warteschlange **FIFO** (first-in first-out).

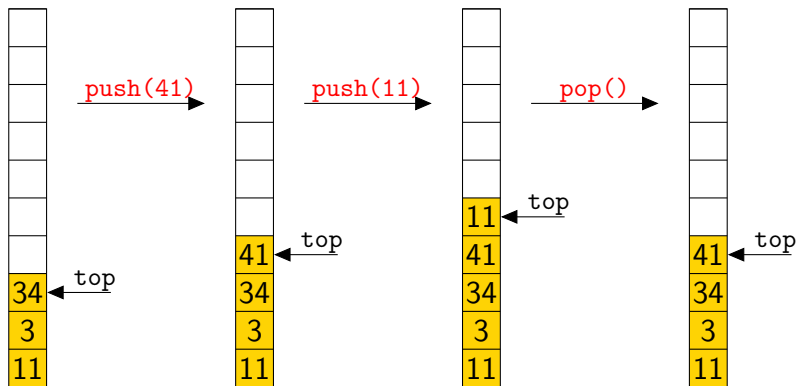
# Stapelimplementierung auf unbeschränkten Arrays



# Stapelimplementierung auf unbeschränkten Arrays



# Stapelimplementierung auf unbeschränkten Arrays



# Stapelimplementierung auf unbeschränkten Arrays

---

```
1 bool isEmpty(Stack s) {  
2     return (s.top == -1);  
3 }  
  
5 void push(Stack s, int e) {  
6     s.top = s.top + 1;  
7     s[s.top] = e;  
8 }  
  
10 int pop(Stack s) {  
11     s.top = s.top - 1;  
12     return s[s.top + 1];  
13 }
```

---

# Stapelimplementierung auf unbeschränkten Arrays

---

```
1 bool isEmpty(Stack s) {  
2     return (s.top == -1);  
3 }  
  
5 void push(Stack s, int e) {  
6     s.top = s.top + 1;  
7     s[s.top] = e;  
8 }  
  
10 int pop(Stack s) {  
11     s.top = s.top - 1;  
12     return s[s.top + 1];  
13 }
```

---

► Die Laufzeit ist jeweils  $\Theta(1)$ .

# Stapelimplementierung auf unbeschränkten Arrays

---

```
1 bool isEmpty(Stack s) {  
2     return (s.top == -1);  
3 }  
  
5 void push(Stack s, int e) {  
6     s.top = s.top + 1;  
7     s[s.top] = e;  
8 }  
  
10 int pop(Stack s) {  
11     s.top = s.top - 1;  
12     return s[s.top + 1];  
13 }
```

---

- ▶ Die Laufzeit ist jeweils  $\Theta(1)$ .
- ▶ In pop muss der Fall eines leeren Stapels nicht berücksichtigt werden. Warum?



# Stapelimplementierung auf unbeschränkten Arrays

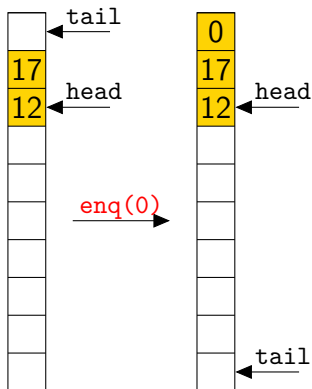
---

```
1 bool isEmpty(Stack s) {  
2     return (s.top == -1);  
3 }  
  
5 void push(Stack s, int e) {  
6     s.top = s.top + 1;  
7     s[s.top] = e;  
8 }  
  
10 int pop(Stack s) {  
11     s.top = s.top - 1;  
12     return s[s.top + 1];  
13 }
```

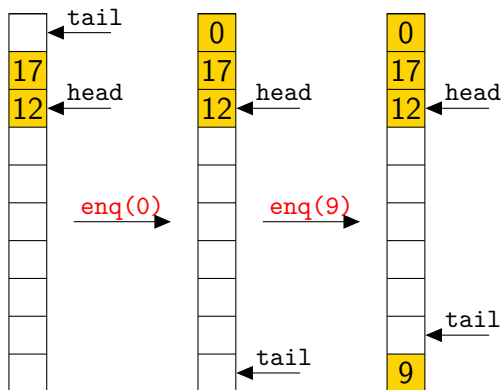
---

- ▶ Die Laufzeit ist jeweils  $\Theta(1)$ .
- ▶ In pop muss der Fall eines leeren Stapels nicht berücksichtigt werden. Warum?
- ▶ Eine Implementierung als verkettete Liste vermeidet eine a priori Festlegung der Arraygröße.

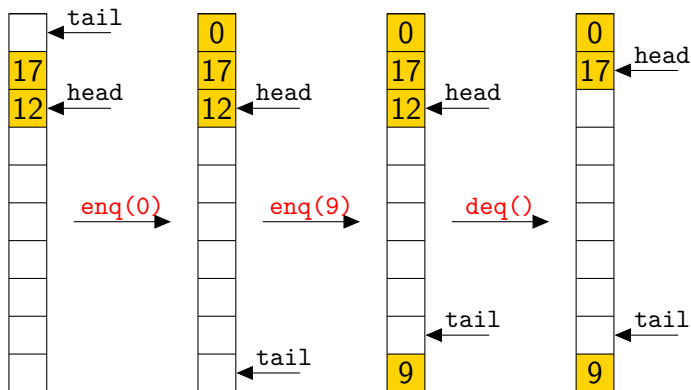
# Warteschlangenimplementierung auf beschränkten Arrays (I)



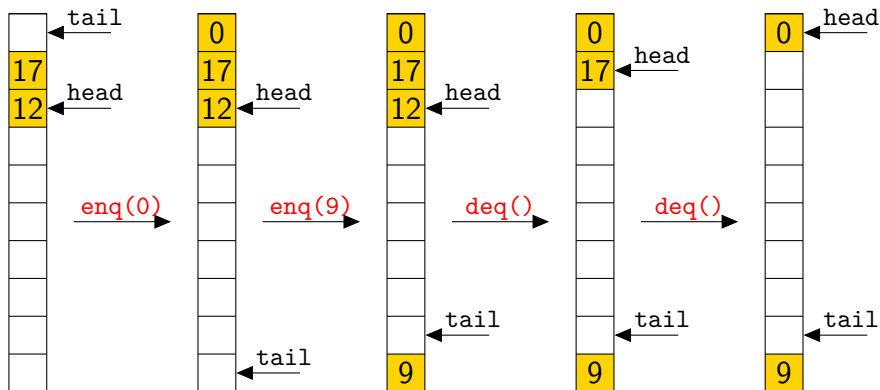
# Warteschlangenimplementierung auf beschränkten Arrays (I)



# Warteschlangenimplementierung auf beschränkten Arrays (I)



# Warteschlangenimplementierung auf beschränkten Arrays (I)



# Warteschlangenimplementierung auf beschränkten Arrays (II)

---

```
1 bool isEmpty(Queue q) {  
2     return (q.head == q.tail);  
3 }
```

► Arraygröße N.

```
5 void enqueue(Queue q, int e) {  
6     q[q.tail] = e;  
7     q.tail = (q.tail + 1) mod N;  
8 }
```

```
10 int dequeue(Queue q) {  
11     int e = q[q.head];  
12     q.head = (q.head + 1) mod N;  
13     return e;  
14 }
```

---

# Warteschlangenimplementierung auf beschränkten Arrays (II)

---

```
1 bool isEmpty(Queue q) {  
2     return (q.head == q.tail);  
3 }  
  
5 void enqueue(Queue q, int e) {  
6     q[q.tail] = e;  
7     q.tail = (q.tail + 1) mod N;  
8 }  
  
10 int dequeue(Queue q) {  
11     int e = q[q.head];  
12     q.head = (q.head + 1) mod N;  
13     return e;  
14 }
```

---

- ▶ Arraygröße  $N$ .
- ▶ Die Laufzeit ist jeweils  $\Theta(1)$ .

# Warteschlangenimplementierung auf beschränkten Arrays (II)

---

```
1 bool isEmpty(Queue q) {  
2     return (q.head == q.tail);  
3 }  
  
5 void enqueue(Queue q, int e) {  
6     q[q.tail] = e;  
7     q.tail = (q.tail + 1) mod N;  
8 }  
  
10 int dequeue(Queue q) {  
11     int e = q[q.head];  
12     q.head = (q.head + 1) mod N;  
13     return e;  
14 }
```

---

- ▶ Arraygröße  $N$ .
- ▶ Die Laufzeit ist jeweils  $\Theta(1)$ .
- ▶ Der Einfachheit halber werden Überläufe nicht abgefangen.



# Warteschlangenimplementierung auf beschränkten Arrays (II)

---

```
1 bool isEmpty(Queue q) {
2     return (q.head == q.tail);
3 }

5 void enqueue(Queue q, int e) {
6     q[q.tail] = e;
7     q.tail = (q.tail + 1) mod N;
8 }

10 int dequeue(Queue q) {
11     int e = q[q.head];
12     q.head = (q.head + 1) mod N;
13     return e;
14 }
```

---

- ▶ Arraygröße  $N$ .
- ▶ Die Laufzeit ist jeweils  $\Theta(1)$ .
- ▶ Der Einfachheit halber werden Überläufe nicht abgefangen.
- ▶ Die Queue ist voll gdw.  
 $q.head == (q.tail + 1) \bmod N$ .

# Die Prioritätswarteschlange (I)

- ▶ Betrachte Elemente, die mit einem **Schlüssel** (key) versehen sind.
- ▶ Jeder Schlüssel sei höchstens an ein Element vergeben.
- ▶ Schlüssel werden als Priorität betrachtet.
- ▶ Die Elemente werden nach ihrer Priorität sortiert.

# Die Prioritätswarteschlange (I)

- ▶ Betrachte Elemente, die mit einem **Schlüssel** (key) versehen sind.
- ▶ Jeder Schlüssel sei höchstens an ein Element vergeben.
- ▶ Schlüssel werden als Priorität betrachtet.
- ▶ Die Elemente werden nach ihrer Priorität sortiert.

## Prioritätswarteschlange (priority queue)

Eine **Prioritätswarteschlange** speichert eine Ansammlung von Elementen und bietet folgende Operationen:

- ▶ `bool isEmpty(PriorityQueue pq)` gibt `true` zurück, wenn pq leer ist, andernfalls `false`.
- ▶ `void insert(PriorityQueue pq, int e, int k)` fügt das Element e mit dem Schlüssel k in pq ein.
- ▶ `int getMin(PriorityQueue pq)` gibt das Element mit dem kleinsten Schlüssel zurück; benötigt nicht-leere pq.



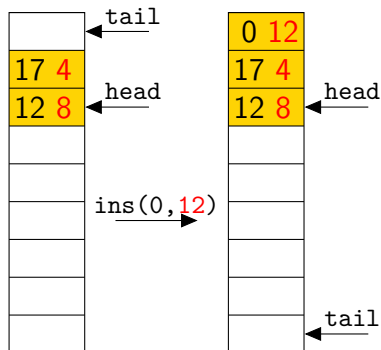
# Die Prioritätswarteschlange (II)

## Prioritätswarteschlange (priority queue) (Forts.)

- ▶ `void delMin(PriorityQueue pq)` entfernt das Element mit dem kleinsten Schlüssel; benötigt nicht-leere `pq`.
- ▶ `int getElt(PriorityQueue pq, int k)` gibt das Element `e` mit dem Schlüssel `k` aus `pq` zurück; `k` muss in `pq` enthalten sein.
- ▶ `void decrKey(PriorityQueue pq, int e, int k)` setzt den Schlüssel von Element `e` auf `k`; `e` muss in `pq` enthalten sein.  
`k` muss außerdem kleiner als der bisherige Schlüssel von `e` sein.

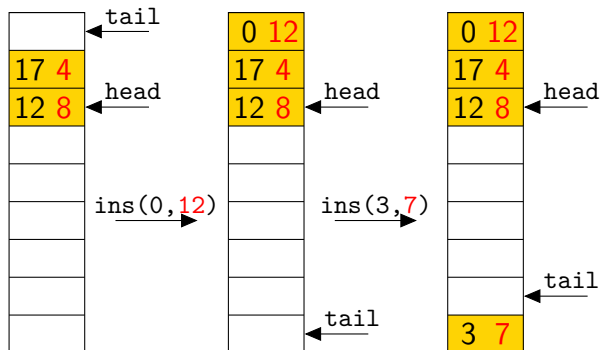
Wichtige Datenstruktur für Greedy-Algorithmen,  
Diskrete-Event-Simulationen, ...

# Prioritätswarteschlange, unsortiert, auf beschränkten Arrays



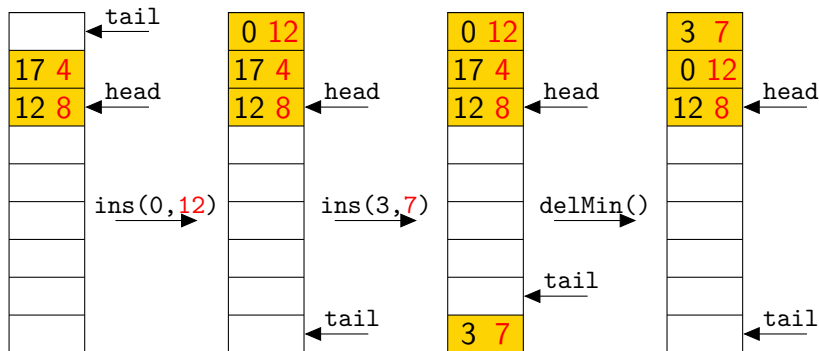
schwarz = Element; rot = Schlüssel

# Prioritätswarteschlange, unsortiert, auf beschränkten Arrays



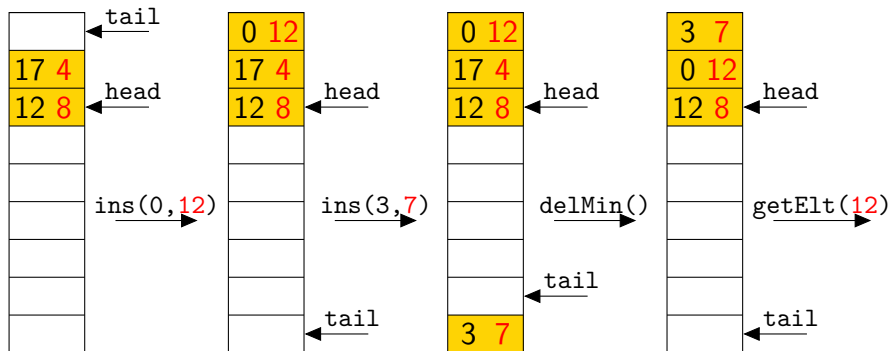
schwarz = Element; rot = Schlüssel

# Prioritätswarteschlange, unsortiert, auf beschränkten Arrays



schwarz = Element; rot = Schlüssel

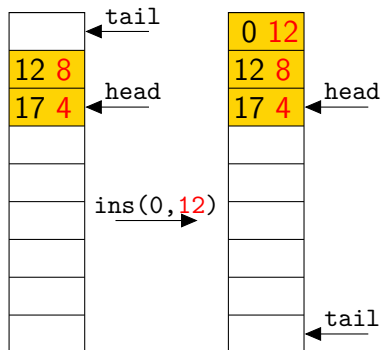
# Prioritätswarteschlange, unsortiert, auf beschränkten Arrays



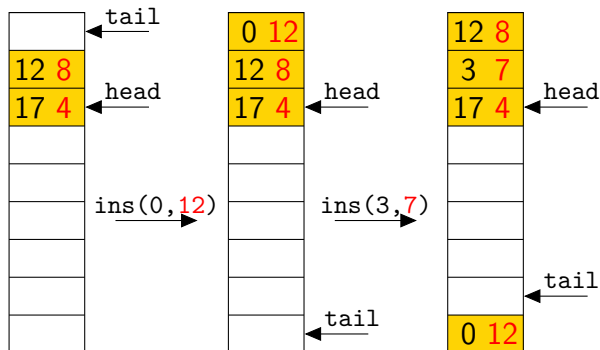
schwarz = Element; rot = Schlüssel



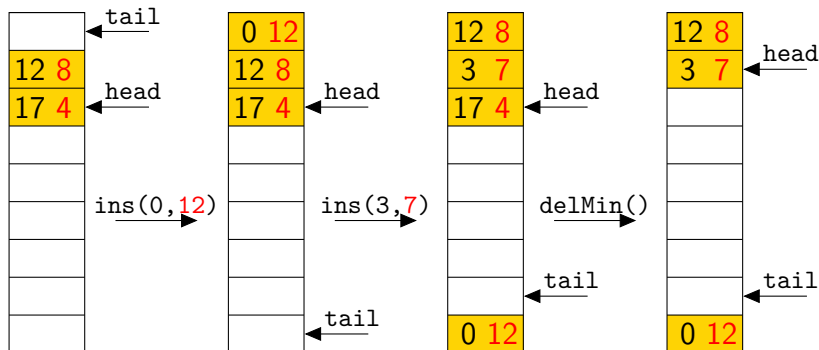
# Prioritätswarteschlange, sortiert, auf beschränkten Arrays



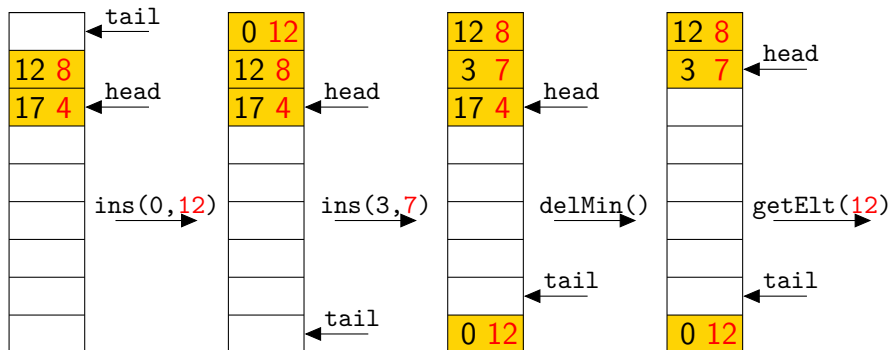
# Prioritätswarteschlange, sortiert, auf beschränkten Arrays



# Prioritätswarteschlange, sortiert, auf beschränkten Arrays



# Prioritätswarteschlange, sortiert, auf beschränkten Arrays



# Zwei Prioritätswarteschlangenimplementierungen

Operation	Implementierung	
	unsortiertes Array	sortiertes Array
<code>isEmpty(pq)</code>	$\Theta(1)$	$\Theta(1)$
<code>insert(pq, e, k)</code>	$\Theta(1)$	$\Theta(n)^*$
<code>getMin(pq)</code>	$\Theta(n)$	$\Theta(1)$
<code>delMin(pq)</code>	$\Theta(n)^*$	$\Theta(1)$
<code>getElt(pq, k)</code>	$\Theta(n)$	$\Theta(\log n)^\dagger$
<code>decrKey(pq, e, k)</code>	$\Theta(n)$	$\Theta(\log n)^\dagger$

- In Vorlesung 8 (Heapsort) werden wir eine weitere Implementierung kennenlernen.

\*Beinhaltet das Verschieben aller Elemente „rechts“ von  $k$ .

$^\dagger$ Mittels binärer Suche.

# Übersicht

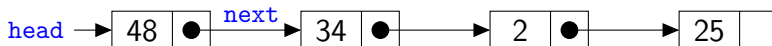
- 1 Abstrakte Datentypen
- 2 Stapel und Warteschlangen
- 3 Verkettete Listen
  - Einfach verkettete Listen
  - Doppelt verkettete Listen
- 4 Binäre Bäume
  - Traversierungen

# Einfach verkettete Listen

## Einfach verkettete Liste

Eine **einfach verkettete Liste** ist eine rekursive, dynamische Datenstruktur.

- ▶ Elemente bestehend aus Schlüssel  $k$  sowie Zeiger auf ein nachfolgendes Element
- ▶ Listen können dynamisch erweitert werden
- ▶ `head` zeigt auf das erste Element der Liste



# Listen (I)

## Liste

Eine **Liste** speichert eine Ansammlung von Elementen mit fester Reihenfolge und bietet folgende Operationen:

- ▶ `void insert(List l, Element x)` fügt das Element `x` an den Anfang der Liste ein.
- ▶ `void remove(List l, Element x)` entfernt das Element `x` aus der Liste.
- ▶ `Element search(List l, int k)` gibt das Element mit dem übergebenen Key `k` zurück und `null` falls es kein derartiges Element gibt.



# Listen (II)

## Liste (Forts.)

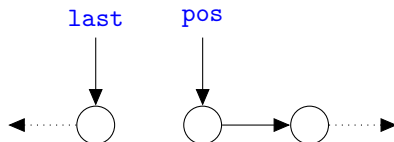
- ▶ `Element minimum(List l)` gibt das Element mit dem kleinsten Schlüssel zurück.
- ▶ `Element maximum(List l)` gibt das Element mit dem höchsten Schlüssel zurück.
- ▶ `Element successor(List l, Element x)` gibt das Nachfolgerelement von Element `x` zurück.
- ▶ `Element predecessor(List l, Element x)` gibt das Vorgängerelement von Element `x` zurück.

# Listen umdrehen

---

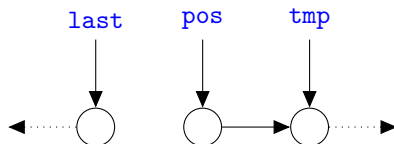
```
1 void reverse(List l) {  
2   last = l.head;  
3   pos = last.next;  
4   last.next = null;  
  
6   while(pos != null){  
7     tmp = pos.next;  
8     pos.next = last;  
9     last = pos;  
10    pos = tmp;  
11  }  
  
13  l.head = last;  
14 }
```

---



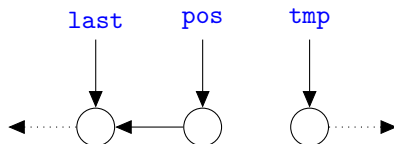
# Listen umdrehen

```
1 void reverse(List l) {  
2   last = l.head;  
3   pos = last.next;  
4   last.next = null;  
  
6   while(pos != null){  
7     tmp = pos.next;  
8     pos.next = last;  
9     last = pos;  
10    pos = tmp;  
11  }  
  
13  l.head = last;  
14 }
```



# Listen umdrehen

```
1 void reverse(List l) {  
2   last = l.head;  
3   pos = last.next;  
4   last.next = null;  
  
6   while(pos != null){  
7     tmp = pos.next;  
8     pos.next = last;  
9     last = pos;  
10    pos = tmp;  
11  }  
  
13  l.head = last;  
14 }
```

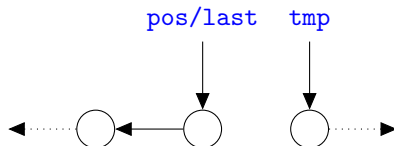


# Listen umdrehen

---

```
1 void reverse(List l) {  
2   last = l.head;  
3   pos = last.next;  
4   last.next = null;  
  
6   while(pos != null){  
7     tmp = pos.next;  
8     pos.next = last;  
9     last = pos;  
10    pos = tmp;  
11  }  
  
13  l.head = last;  
14 }
```

---

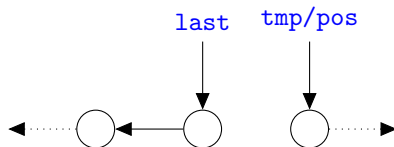


# Listen umdrehen

---

```
1 void reverse(List l) {  
2   last = l.head;  
3   pos = last.next;  
4   last.next = null;  
  
6   while(pos != null){  
7     tmp = pos.next;  
8     pos.next = last;  
9     last = pos;  
10    pos = tmp;  
11  }  
  
13  l.head = last;  
14 }
```

---

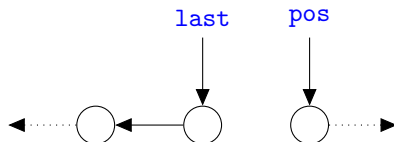


# Listen umdrehen

---

```
1 void reverse(List l) {  
2   last = l.head;  
3   pos = last.next;  
4   last.next = null;  
  
6   while(pos != null){  
7     tmp = pos.next;  
8     pos.next = last;  
9     last = pos;  
10    pos = tmp;  
11  }  
  
13  l.head = last;  
14 }
```

---

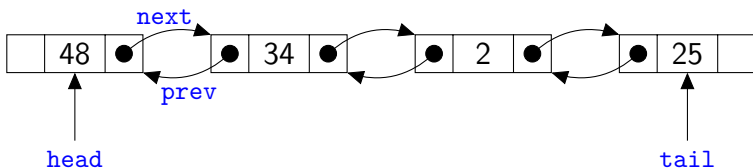


# Doppelt verkettete Listen

## Doppelt verkettete Liste

Eine **doppelt verkettete Liste** kann sowohl vorwärts als auch rückwärts durchlaufen werden. Sie implementiert den ADT List.

- ▶ Elemente besitzen neben Schlüssel und Nachfolgender einen weiteren Zeiger auf das vorherige Element.
- ▶ Zusätzlicher Zeiger `tail` zeigt auf das letzte Element der Liste





# Laufzeiten

## Implementierung

Operation	Implementierung	
	einfach verkettete Liste	doppelt verkettete Liste
insert(L,x)	$\Theta(1)$	$\Theta(1)$
remove(L,x)	$\Theta(n)$	$\Theta(1)$
search(L,k)	$\Theta(n)$	$\Theta(n)$
minimum(L)	$\Theta(n)$	$\Theta(n)$
maximum(L)	$\Theta(n)$	$\Theta(n)$
successor(L,x)	$\Theta(1)$	$\Theta(1)$
predecessor(L,x)	$\Theta(n)$	$\Theta(1)$

# Laufzeiten

## Implementierung

Operation	Implementierung	
	einfach verkettete Liste	doppelt verkettete Liste
insert(L,x)	$\Theta(1)$	$\Theta(1)$
remove(L,x)	$\Theta(n)$	$\Theta(1)$
search(L,k)	$\Theta(n)$	$\Theta(n)$
minimum(L)	$\Theta(n)$	$\Theta(n)$
maximum(L)	$\Theta(n)$	$\Theta(n)$
successor(L,x)	$\Theta(1)$	$\Theta(1)$
predecessor(L,x)	$\Theta(n)$	$\Theta(1)$

- Suchen eines Schlüssels erfordert einen Durchlauf der gesamten Liste. Gibt es andere Möglichkeiten, die Daten zu organisieren?

# Übersicht

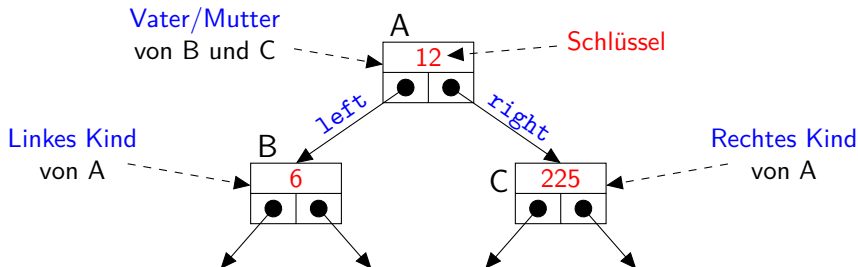
- 1 Abstrakte Datentypen
- 2 Stapel und Warteschlangen
- 3 Verkettete Listen
  - Einfach verkettete Listen
  - Doppelt verkettete Listen
- 4 Binäre Bäume
  - Traversierungen

# Binärbäume – Intuition

## Binärbaum – Intuition

Betrachte einen **binären Baum**:

- ▶ Jedes Element bekommt zwei Zeiger (left und right) zu den nachfolgenden Elementen.
- ▶ Man erhält in etwa folgende Datenstruktur:



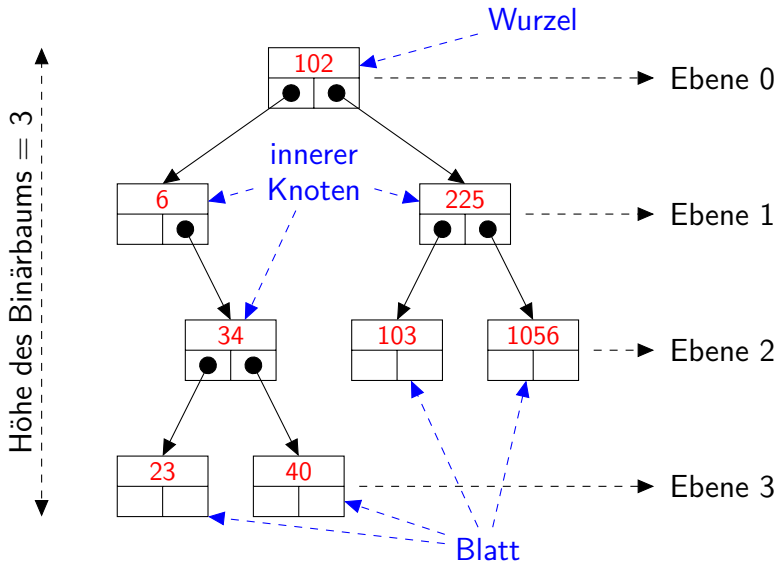
# Binärbäume – Definition

## Definition (Binärbaum)

Ein **Binärbaum** (binary tree) ist ein gerichteter, zyklfreier Graph  $(V, E)$  mit **Knoten** (nodes, allgemein: vertices)  $V$  und **gerichteten Kanten** (edges)  $E \in V \times V$ .

- ▶ Es gibt genau einen ausgezeichneten Knoten, die **Wurzel** (root).
- ▶ Alle Kanten zeigen von der Wurzel weg.
- ▶ Der Ausgangsgrad (out-degree) jedes Knotens ist höchstens 2.
- ▶ Der Eingangsgrad eines Knoten ist 1, bzw. 0 bei der Wurzel.
- ▶ Sonderfall: Baum mit  $V = E = \emptyset$ .

# Binärbäume – Begriffe (I)



# Binärbäume – Begriffe (II)

## Definition (Binärbaum – Begriffe)

- ▶ Ein Knoten mit leerem linken und rechten Teilbaum heißt **Blatt** (leaf).
- ▶ Die **Tiefe** (depth) (auch: Ebene / level) eines Knotens ist sein Abstand, d. h. die Pfadlänge, von der Wurzel.
- ▶ Die **Höhe** (level) eines Baumes ist die maximale Tiefe seiner Blätter.

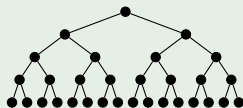
# Binärbäume – Begriffe (II)

## Definition (Binärbaum – Begriffe)

- ▶ Ein Knoten mit leerem linken und rechten Teilbaum heißt **Blatt** (leaf).
- ▶ Die **Tiefe** (depth) (auch: Ebene / level) eines Knotens ist sein Abstand, d. h. die Pfadlänge, von der Wurzel.
- ▶ Die **Höhe** (level) eines Baumes ist die maximale Tiefe seiner Blätter.

## Beispiel (Vorteile von binären Bäumen)

Angenommen, man möchte 31 Elemente vorhalten:



Ebene 0 (Wurzel) enthält 1 Element	<u>Gesamt:</u>
Ebene 1 enthält 2 Elemente	3
Ebene 2 enthält 4 Elemente	7
Ebene 3 enthält 8 Elemente	15
Ebene 4 enthält 16 Elemente	31



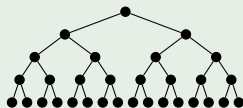
# Binärbäume – Begriffe (II)

## Definition (Binärbaum – Begriffe)

- ▶ Ein Knoten mit leerem linken und rechten Teilbaum heißt **Blatt** (leaf).
- ▶ Die **Tiefe** (depth) (auch: Ebene / level) eines Knotens ist sein Abstand, d. h. die Pfadlänge, von der Wurzel.
- ▶ Die **Höhe** (level) eines Baumes ist die maximale Tiefe seiner Blätter.

## Beispiel (Vorteile von binären Bäumen)

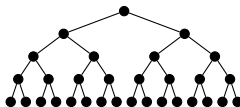
Angenommen, man möchte 31 Elemente vorhalten:



Ebene 0 (Wurzel) enthält 1 Element	<u>Gesamt:</u>
Ebene 1 enthält 2 Elemente	3
Ebene 2 enthält 4 Elemente	7
Ebene 3 enthält 8 Elemente	15
Ebene 4 enthält 16 Elemente	31

⇒ Ein Element kann in 5 Schritten (statt 31) erreicht werden.

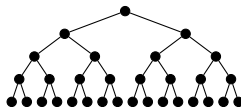
# Einige Fakten über binäre Bäume



## Lemma (Übung)

- ▶ Ebene  $d$  enthält höchstens  $2^d$  Knoten.
- ▶ Ein Binärbaum mit Höhe  $h$  kann maximal  $2^{h+1} - 1$  Knoten enthalten.
- ▶ Enthält er  $n$  Knoten, dann hat er mindestens Höhe  $\lceil \log(n + 1) \rceil - 1$  ( $\log \equiv \log_2$ ).

# Einige Fakten über binäre Bäume



## Lemma (Übung)

- ▶ Ebene  $d$  enthält höchstens  $2^d$  Knoten.
- ▶ Ein Binärbaum mit Höhe  $h$  kann maximal  $2^{h+1} - 1$  Knoten enthalten.
- ▶ Enthält er  $n$  Knoten, dann hat er mindestens Höhe  $\lceil \log(n + 1) \rceil - 1$  ( $\log \equiv \log_2$ ).

## Definition (vollständig)

Ein Binärbaum heißt **vollständig**, wenn er bei Höhe  $h$  alle  $2^{h+1} - 1$  Knoten enthält.

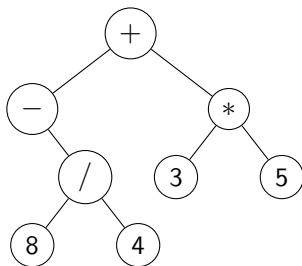
# Traversierung

## Traversierung

Eine **Traversierung** ist ein Baumdurchlauf mit folgenden Eigenschaften:

1. Die Traversierung beginnt und endet an der Wurzel.
2. Die Traversierung folgt den Kanten des Baumes. Jede Kante wird genau zweimal durchlaufen: Einmal von oben nach unten und danach von unten nach oben.
3. Die Teilbäume eines Knotens werden in festgelegter Reihenfolge (zuerst linker, dann rechter Teilbaum) besucht.
4. Unterschiede bestehen darin, bei welchen Durchlauf man den Knoten selbst (bzw. das dort gespeicherte Element) „besucht“.

# Inorder-Traversierung



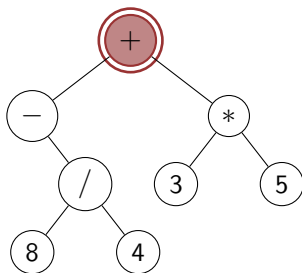
```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

## Beispiel

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

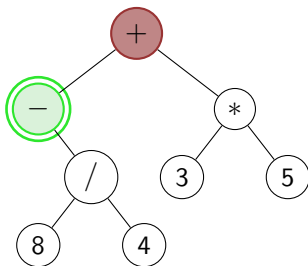
## Beispiel

(

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

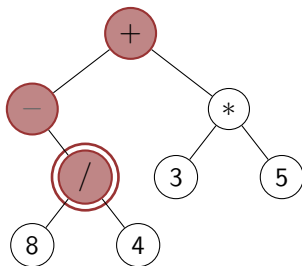
## Beispiel

((-

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

## Beispiel

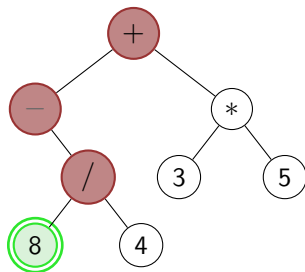
((-(

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.



# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

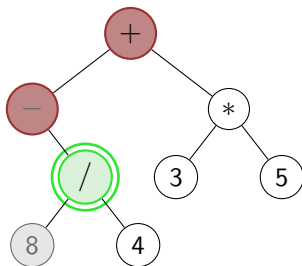
## Beispiel

((-(8

## Linearisierung

Eine Aufzählung aller Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

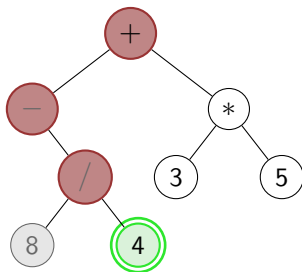
## Beispiel

((-(8/

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

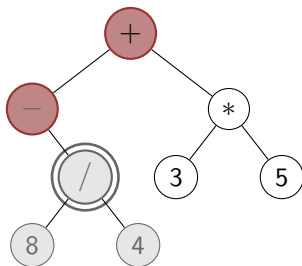
## Beispiel

$((-(8/4$

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

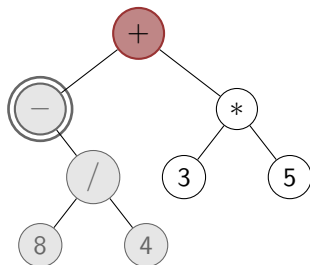
## Beispiel

$((-(8/4)$

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

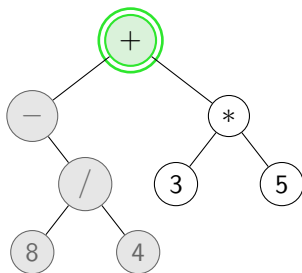
## Beispiel

$((-(8/4))$

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

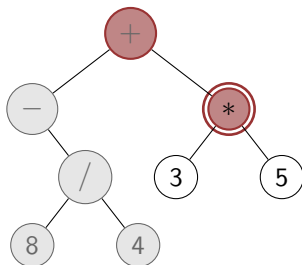
## Beispiel

$((-(8/4)) +$

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

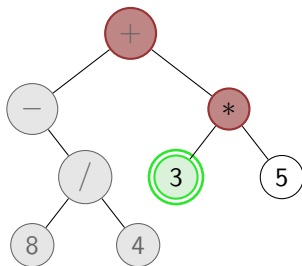
## Beispiel

$((-(8/4)) + ($

## Linearisierung

Eine Aufzählung aller Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

## Beispiel

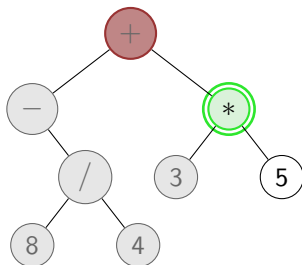
$((-(8/4)) + (3$

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.



# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

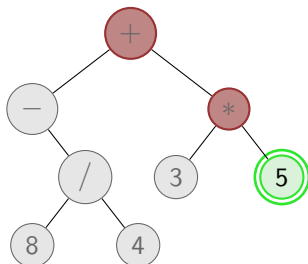
## Beispiel

$((-(8/4)) + (3 * 5))$

## Linearisierung

Eine Aufzählung aller Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

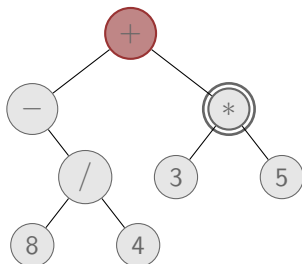
## Beispiel

$((-(8/4)) + (3 * 5$

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

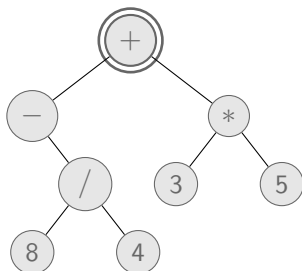
## Beispiel

$((-(8/4)) + (3 * 5))$

## Linearisierung

Eine Aufzählung aller Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Inorder-Traversierung



```
1 void inorder(Node node) {  
2     if (node != null) {  
3         "("  
4         inorder(node.left);  
5         print(node);  
6         inorder(node.right);  
7         ")"  
8     }  
9 }
```

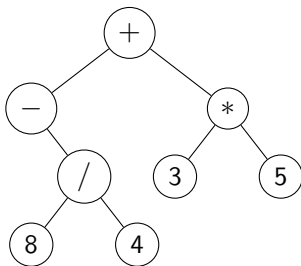
## Beispiel

$((-(8/4)) + (3 * 5))$

## Linearisierung

Eine Aufzählung alle Elemente eines Baumes in der Reihenfolge einer bestimmten Traversierung (ohne Klammern) nennt man **Linearisierung**.

# Preorder, Inorder, Postorder (I)



---

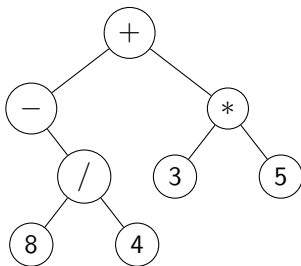
```
1 void inorder(Node node) {  
2     if (node != null) {  
3         inorder(node.left);  
4         print(node);  
5         inorder(node.right);  
6     }  
7 }
```

---

## Beispiel (Inorder)

$- 8 / 4 + 3 * 5$

# Preorder, Inorder, Postorder (I)



---

```
1 void preorder(Node node) {  
2     if (node != null) {  
3         print(node);  
4         preorder(node.left);  
5         preorder(node.right);  
6     }  
7 }
```

---

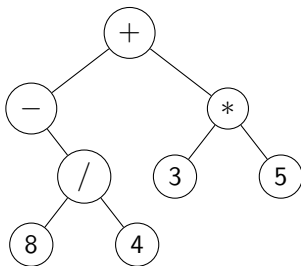
## Beispiel (Inorder)

$- 8 / 4 + 3 * 5$

## Beispiel (Preorder)

$+ - / 8 4 * 3 5$

# Preorder, Inorder, Postorder (I)




---

```

1 void postorder(Node node) {
2     if (node != null) {
3         postorder(node.left);
4         postorder(node.right);
5         print(node);
6     }
7 }

```

---

## Beispiel (Inorder)

$- 8 / 4 + 3 * 5$

## Beispiel (Preorder)

$+ - / 8 4 * 3 5$

## Beispiel (Postorder – Umgekehrte Polnische Notation (RPN))

$8 4 / -^{\dagger} 3 5 * +$

---

$^{\dagger}$ neg

# Preorder, Inorder, Postorder-Traversierung

```
1 void preorder(Node node) {  
2     if (node != null) {  
3         visit(node);  
4         preorder(node.left);  
5         preorder(node.right);  
6     }  
7 }
```

```
9 void inorder(Node node) {  
10    if (node != null) {  
11        inorder(node.left);  
12        visit(node);  
13        inorder(node.right);  
14    }  
15 }
```

```
16 void postorder(Node node) {  
17     if (node != null) {  
18         postorder(node.left);  
19         postorder(node.right);  
20         visit(node);  
21     }  
22 }
```

## Komplexität

$\Theta(n)$ , wobei  $n$  die Anzahl der Knoten ist.



# Preorder, Inorder, Postorder (II)

## Satz

*Ist von einem (unbekannte) Binärbaum mit eindeutigen Werten sowohl die Inorder-Linearisierung als auch entweder die Preorder- oder die Postorder-Linearisierung gegeben, dann ist der Baum eindeutig bestimmt.*

## Beispiel (Rekonstruktion aus Inorder- und Preorder-Linearisierung)

Inorder: 5 2 7 9

# Preorder, Inorder, Postorder (II)

## Satz

*Ist von einem (unbekannte) Binärbaum mit eindeutigen Werten sowohl die Inorder-Linearisierung als auch entweder die Preorder- oder die Postorder-Linearisierung gegeben, dann ist der Baum eindeutig bestimmt.*

## Beispiel (Rekonstruktion aus Inorder- und Preorder-Linearisierung)

Inorder: 5 2 7 9

Preorder: 5 7 2 9

# Preorder, Inorder, Postorder (II)

## Satz

*Ist von einem (unbekannte) Binärbaum mit eindeutigen Werten sowohl die Inorder-Linearisierung als auch entweder die Preorder- oder die Postorder-Linearisierung gegeben, dann ist der Baum eindeutig bestimmt.*

## Beispiel (Rekonstruktion aus Inorder- und Preorder-Linearisierung)

Inorder: 5 2 7 9

Preorder: 5 7 2 9

