

Datenstrukturen und Algorithmen

Vorlesung 3: Elementare Datenstrukturen (K10)

Joost-Pieter Katoen
(Christian Dehnert)

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

<http://moves.rwth-aachen.de/teaching/ss-15/dsal/>

16. April 2015



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 (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 (II)

Beispiel

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

- ▶ die Vorbedingung: `true` (d. h. leere Aussage) und
- ▶ die Nachbedingung: neuester Eintrag von `s` ist `e`.

- ▶ ADTs sind *durch ihre Spezifikation* festgelegte „Standard“-Komponenten zum Aufbau unserer Algorithmen.

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.

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, Element e)` als Array-Implementierung:

```
1 void push(Stack s, Element 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.

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).

Beispiele für ADTs: Stapel (I)



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 (II)

Stapel (stack)

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

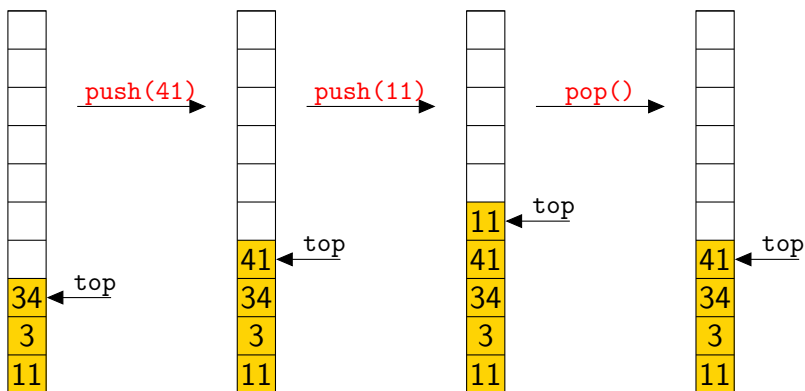
- ▶ **bool** isEmpty(Stack s) gibt **true** zurück, wenn s leer ist und andernfalls **false**.
- ▶ **void** push(Stack s, Element e) fügt das Element e in den Stapel s ein.
- ▶ **Element** pop(Stack s) entfernt das zuletzt eingefügte Element und gibt es zurück; pop(s) benötigt einen nicht-leeren Stapel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

Ein Stapel bietet **LIFO** (last-in first-out) Semantik.

Beispiele für ADTs: Warteschlangen (I)



Stapelimplementierung auf unbeschränkten Arrays



Beispiele für ADTs: Warteschlangen (II)

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, **Element** e) fügt das Element e in die Warteschlange q ein.
- ▶ **Element** dequeue(**Queue** q) entfernt das schon am längsten in der Warteschlange vorhandene Element und gibt es zurück; benötigt daher eine nicht-leere Warteschlange q.

Eine Warteschlange bietet **FIFO** (first-in first-out) Semantik.

Stapelimplementierung auf unbeschränkten Arrays

```

1 bool isEmpty(Stack s) {
2   return (s.top == -1);
3 }

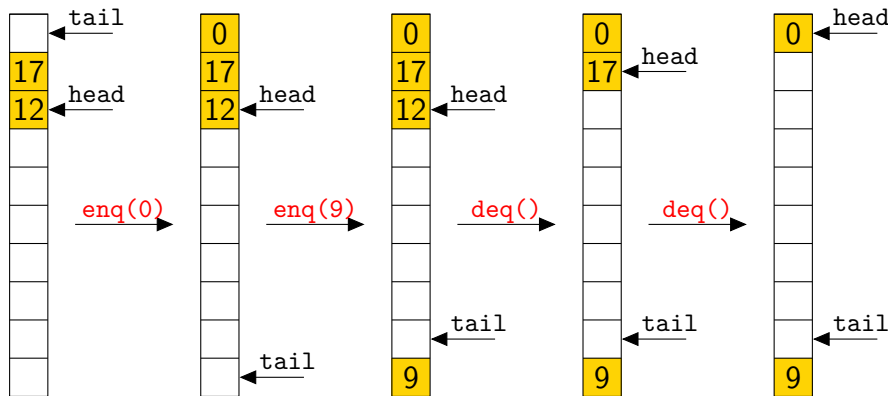
5 void push(Stack s, Element e)
6   {
7     s.top = s.top + 1;
8     s[s.top] = e;
9   }

10 Element pop(Stack s) {
11   Element e = s[s.top];
12   s.top = s.top - 1;
13   return e;
14 }

```

- ▶ 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)



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, Element e, int k)` fügt das Element `e` mit dem Schlüssel `k` in `pq` ein.
- ▶ `Element getMin(PriorityQueue pq)` gibt das Element mit dem kleinsten Schlüssel zurück; benötigt nicht-leere `pq`.

→

Warteschlangenimplementierung auf beschränkten Arrays (II)

```

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

5 void enqueue(
6     Queue q,
7     Element e
8 ) {
9     q[q.tail] = e;
10    q.tail = (q.tail + 1) mod N;
11 }

13 Element dequeue(Queue q) {
14     Element e = q[q.head];
15     q.head = (q.head + 1) mod N;
16     return e;
17 }

```

- ▶ 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) mod N`.

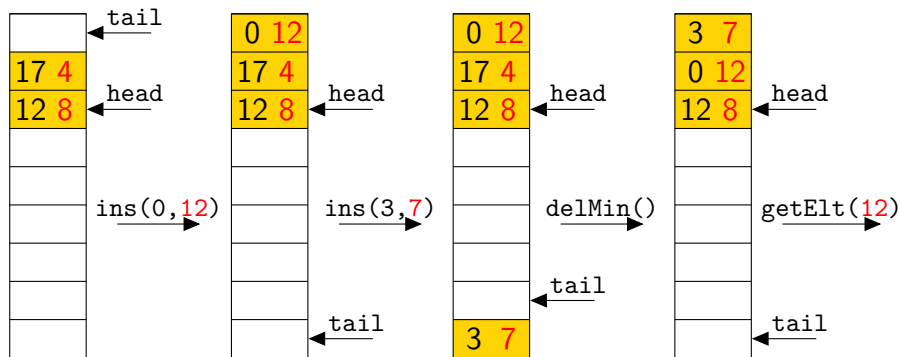
Die Prioritätswarteschlange (II)

Prioritätswarteschlange (priority queue) (Forts.)

- ▶ `void delMin(PriorityQueue pq)` entfernt das Element mit dem kleinsten Schlüssel; benötigt nicht-leere `pq`.
- ▶ `Element 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, Element 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

Zwei Prioritätswarteschlangenimplementierungen

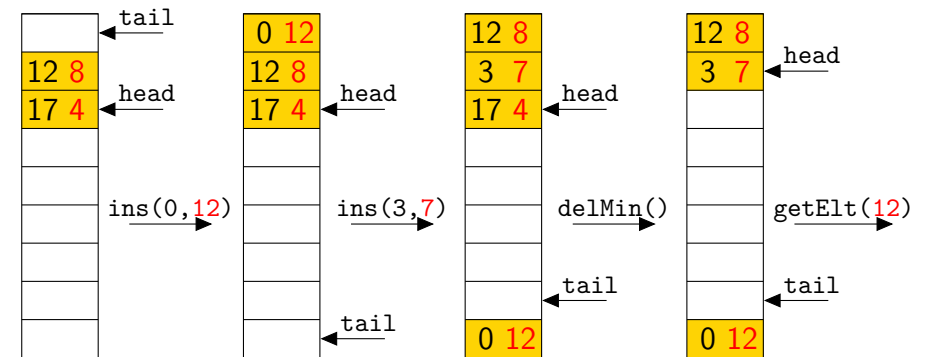
Operation	Implementierung	
	unsortiertes Array	sortiertes Array
isEmpty(pq)	$\Theta(1)$	$\Theta(1)$
insert(pq, e, k)	$\Theta(1)$	$\Theta(n)^*$
getMin(pq)	$\Theta(n)$	$\Theta(1)$
delMin(pq)	$\Theta(n)^*$	$\Theta(1)$
getElt(pq, k)	$\Theta(n)$	$\Theta(\log n)^\dagger$
decrKey(pq, e, k)	$\Theta(n)$	$\Theta(n)^*$

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

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

[†]Mittels binärer Suche.

Prioritätswarteschlange, sortiert, auf beschränkten Arrays



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 (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 , **int** k) gibt das Nachfolgerelement des Elements mit Schlüssel k zurück.
- ▶ **Element** predecessor(**List** l , **int** k) gibt das Vorgängerelement des Elements mit Schlüssel k zurück.

Listen (I)

Liste (ADT)

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

- ▶ **void** insert(**List** l , **Element** x , **int** k) fügt das Element x mit dem Schlüssel k in die Liste ein.
- ▶ **void** remove(**List** l , **int** k) entfernt ein Vorkommen eines Elements mit Schlüssel k aus der Liste, falls mindestens ein solches Element in der Liste existiert.
- ▶ **Element** search(**List** l , **int** k) gibt das Element mit dem übergebenen Schlüssel k zurück und **null** falls es kein derartiges Element gibt.

Listen (III)

Liste (Untertypen)

Die Elementreihenfolge kann auf unterschiedliche Arten definiert werden – wie z. B. durch folgende (strengere) Spezifikationen der insert Operation:

- ▶ **void** insert(**List** l , **Element** x , **int** k) fügt das Element x mit dem Schlüssel k am Anfang der Liste ein (entspricht z. B. **LinkedList** Klasse in Java, wenn man die Methode **addFirst** als insert Operation verwendet).
- ▶ **void** insert(**List** l , **Element** x , **int** k) fügt das Element x mit dem Schlüssel k sortiert nach Schlüssel in die Liste ein (z. B. bei einer Skiplist).
- ▶ **void** insert(**List** l , **Element** x , **int** k) fügt das Element x mit dem Schlüssel k am Ende der Liste ein (entspricht z. B. **List** Interface in Java, wenn man die Methode **add** als insert Operation verwendet).

Wenn keine näheren Angaben gemacht werden, gehen wir in dieser Vorlesung meist von der ersten dieser drei Spezifikationen aus.

Listen umdrehen

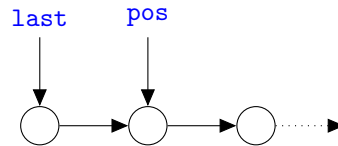
```

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 }

```



Laufzeiten

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

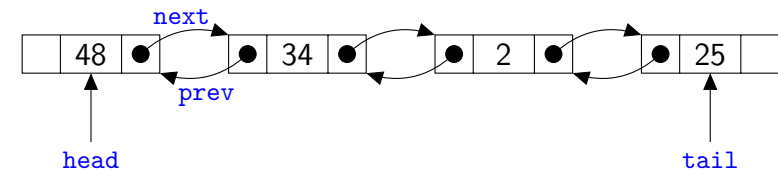
- Suchen eines Schlüssels erfordert einen Durchlauf der gesamten Liste. Gibt es andere Möglichkeiten, die Daten zu organisieren?

Doppelt verkettete Listen

Doppelt verkettete Liste

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

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



Übersicht

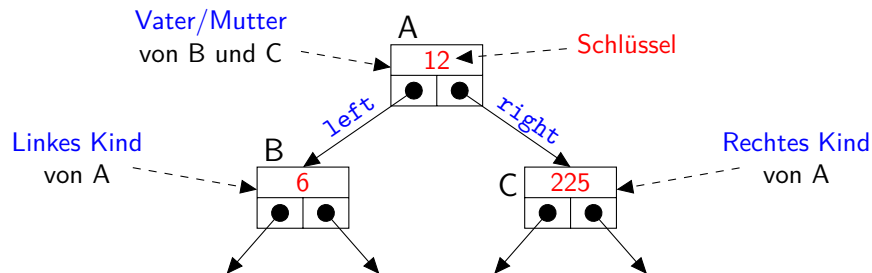
- 1 Abstrakte Datentypen
- 2 Stapel und Warteschlangen
- 3 Verkettete Listen
 - Einfach verkettete Listen
 - Doppelt verkettete Listen
- 4 Binäre Bäume
 - Traversierungen

Binärbäume – Intuition

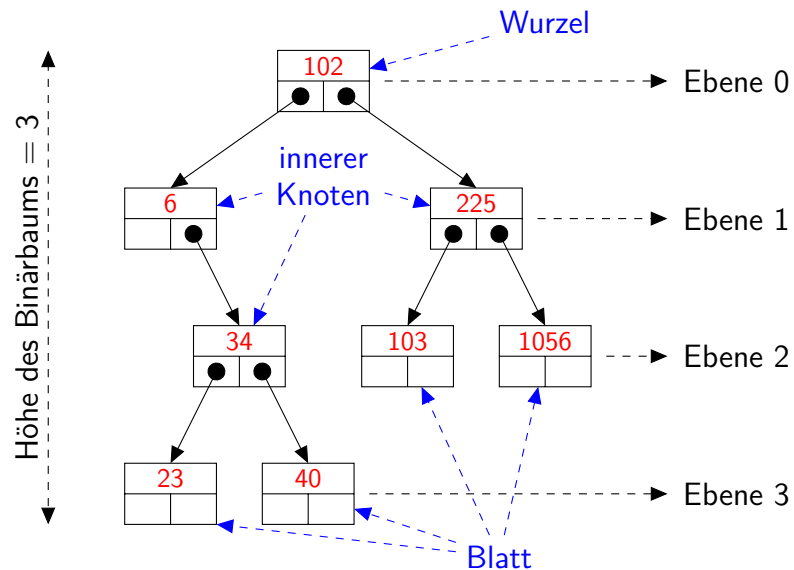
Binärbaum – Intuition

Betrachte einen **binären Baum**:

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



Binärbäume – Begriffe (I)



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) $E \subseteq V \times V$.

- ▶ 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 Knotens ist 1, bzw. 0 bei der Wurzel.
- ▶ Sonderfall: Baum mit $V = E = \emptyset$.

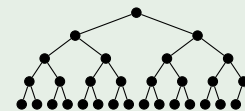
Binärbäume – Begriffe (II)

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** (height) eines Baumes ist die maximale Tiefe seiner Blätter.

Beispiel (Vorteile von binären Bäumen)

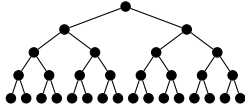
Angenommen, man möchte 31 Elemente vorhalten:



Ebene 0 (Wurzel) enthält 1 Element	Gesamt:
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



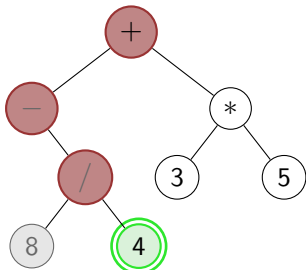
Lemma (Übung)

- ▶ 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 $\lceil \log(n+1) \rceil - 1$ ($\log \equiv \log_2$).

Definition (vollständig)

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

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**.

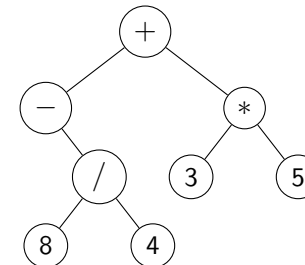
Traversierung

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 welchem Durchlauf man den Knoten selbst (bzw. das dort gespeicherte Element) „besucht“.

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)

$- 8 / 4 + 3 * 5$

Beispiel (Preorder)

$+ - / 8 4 * 3 5$

Beispiel (Postorder – Umgekehrte Polnische Notation (RPN))

$8 4 / - \uparrow 3 5 * +$

$\uparrow$ 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.

Zusammenfassung

- ▶ ADTs: Spezifikation und Implementierung
- ▶ Stapel, Warteschlangen, Prioritätswarteschlangen
- ▶ einfach und doppelt verkettete Listen
- ▶ Binäre Bäume

Preorder, Inorder, Postorder (II)

Satz

Ist von einem (unbekannten) 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

