Datenstrukturen und Algorithmen

Vorlesung 3: Elementare Datenstrukturen (K10)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

`https://moves.rwth-aachen.de/teaching/ss-18/dsal/`

23. April 2018

Übersicht

Übersicht

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

Eingabe, Ausgabe ; nicht wie

Unterscheide zwischen

1. Spezifikation des ADTs: wie sich die Datenobjekte verhalten, und
2. 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)

Eingabe, Ausgabe

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.

$\Rightarrow$ Grundlage für die Argumentation über die Korrektheit des ADTs.

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.

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.

$\Longrightarrow$ 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.

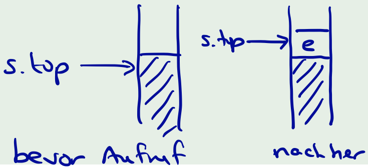
Verschiedene Implementierungen von ADTs *der selben Spezifikation* ermöglichen es uns die Performance zu optimieren.

$\Rightarrow$ Grundlage für die Argumentation über die Effizienz des ADTs.

Beispiel

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

```
void push(Stack s, Element e)
    {
  s.top = s.top + 1;
  s[s.top] = e;
}
```



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

Beispiele für ADTs: Stapel (I)

Beispiele für ADTs: Stapel (II)

Vorbedingung:
true

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.

Vorbedingung

Beispiele für ADTs: Warteschlangen (I)

Beispiele für ADTs: Warteschlangen (II)

Warteschlange (queue)

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

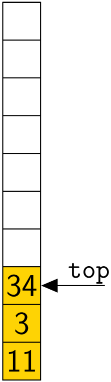
- `bool isEmpty(Queue q)` gibt `true` zurück, wenn `q` leer ist, andernfalls `false`.
- `void enqueue(Queue q, Element e)` fügt das Element `e` in die Warteschlange `q` ein.
- `Element dequeue(Queue q)` entfernt das schon am längsten in der Warteschlange vorhandene Element und gibt es zurück; benötigt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

Vorbedingung

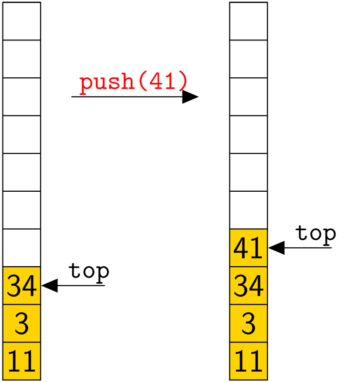
Eine Warteschlange bietet FIFO (first-in first-out) Semantik.

Stapelimplementierung auf unbeschränkten Arrays

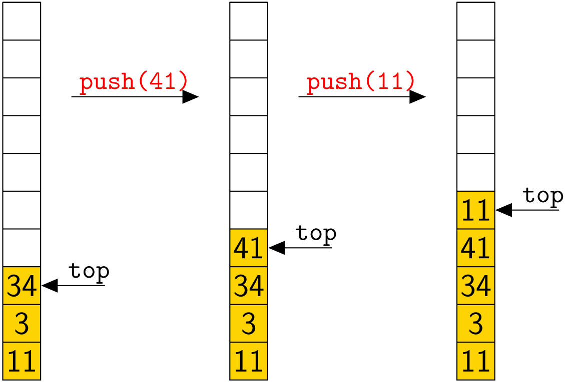
Datenstruktur
für die Implementierung



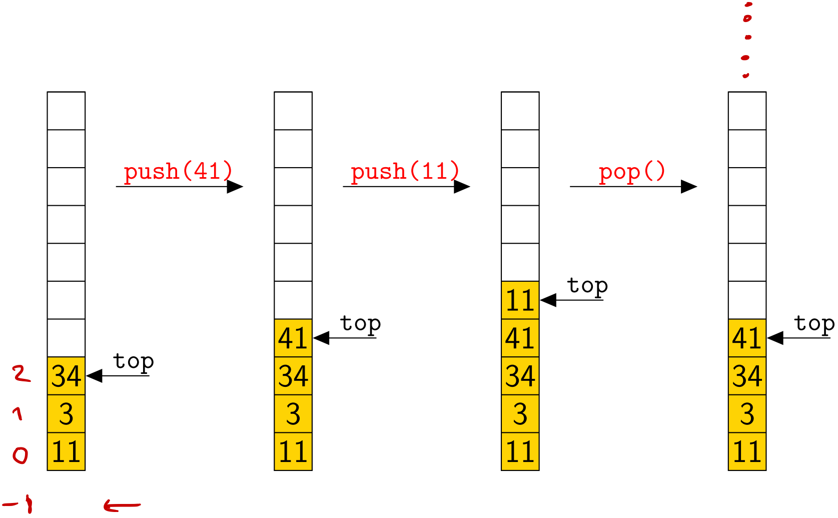
Stapelimplementierung auf unbeschränkten Arrays



Stapelimplementierung auf unbeschränkten Arrays

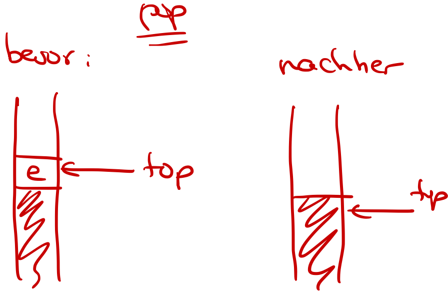


Stapelimplementierung auf unbeschränkten Arrays



Stapelimplementierung auf unbeschränkten Arrays

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

void push(Stack s, Element e)
    {
  s.top = s.top + 1;
  s[s.top] = e;
}

Element pop(Stack s) {
  Element e = s[s.top];
  s.top = s.top - 1;
  return e;
}
```



Stapelimplementierung auf unbeschränkten Arrays

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

void push(Stack s, Element e)
    {
  s.top = s.top + 1;
  s[s.top] = e;
}

Element pop(Stack s) {
  Element e = s[s.top];
  s.top = s.top - 1;
  return e;
}
```

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

Stapelimplementierung auf unbeschränkten Arrays

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

void push(Stack s, Element e) {
  s.top = s.top + 1;
  s[s.top] = e;
}

Element pop(Stack s) {
  Element e = s[s.top];
  s.top = s.top - 1;
  return e;
}
```

~~if isEmpty(s) {~~

~~...~~

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

→ Vorbedingung

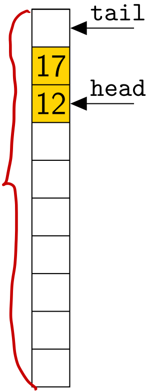
s sei nicht leer

Stapelimplementierung auf unbeschränkten Arrays

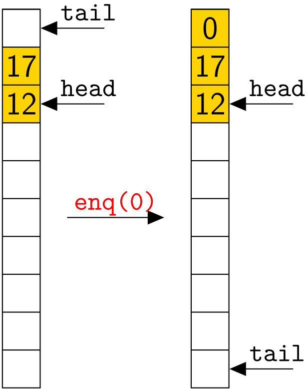
```
bool isEmpty(Stack s) {
  return (s.top == -1);
}

void push(Stack s, Element e) {
  s.top = s.top + 1;
  s[s.top] = e;
}

Element pop(Stack s) {
  Element e = s[s.top];
  s.top = s.top - 1;
  return e;
}
```

- 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.

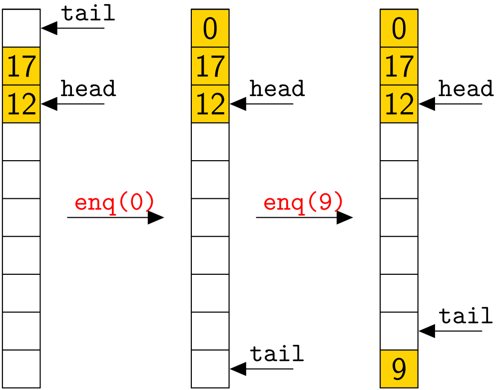
Warteschlangenimplementierung auf beschränkten Arrays (I)



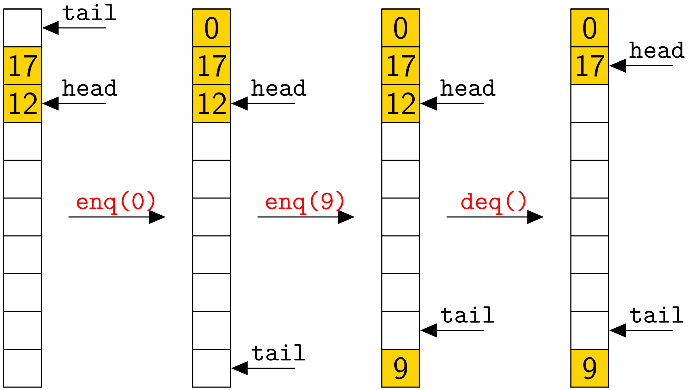
Warteschlangenimplementierung auf beschränkten Arrays (I)



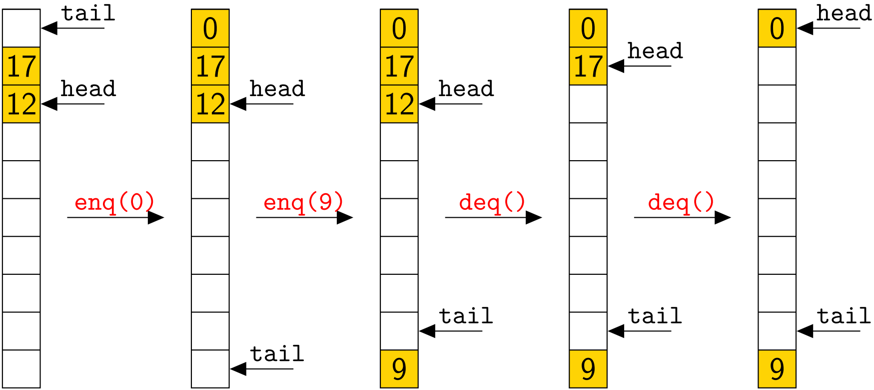
Warteschlangenimplementierung auf beschränkten Arrays (I)



Warteschlangenimplementierung auf beschränkten Arrays (I)



Warteschlangenimplementierung auf beschränkten Arrays (I)



Warteschlangenimplementierung auf beschränkten Arrays (II)

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

void enqueue(
  Queue q,
  Element e
) {
  q[q.tail] = e;
  q.tail = (q.tail + 1) mod N;
}

Element dequeue(Queue q) {
  Element e = q[q.head];
  q.head = (q.head + 1) mod N;
  return e;
}
```

- Arraygröße N.

$N = 10$

Warteschlangenimplementierung auf beschränkten Arrays (II)

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

void enqueue(
  Queue q,
  Element e
) {
  q[q.tail] = e;
  q.tail = (q.tail + 1) mod N;
}

Element dequeue(Queue q) {
  Element e = q[q.head];
  q.head = (q.head + 1) mod N;
  return e;
}
```

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

Warteschlangenimplementierung auf beschränkten Arrays (II)

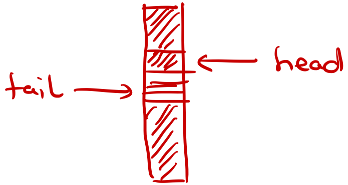
```
bool isEmpty(Queue q) {
  return (q.head == q.tail);
}

void enqueue(
  Queue q,
  Element e
) {
  q[q.tail] = e;
  q.tail = (q.tail + 1) mod N;
}

Element dequeue(Queue q) {
  Element e = q[q.head];
  q.head = (q.head + 1) mod N;
  return e;
}
```

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

Warteschlangenimplementierung auf beschränkten Arrays (II)

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

void enqueue(
  Queue q,
  Element e
) {
  q[q.tail] = e;
  q.tail = (q.tail + 1) mod N;
}

Element dequeue(Queue q) {
  Element e = q[q.head];
  q.head = (q.head + 1) mod N;
  return e;
}
```

- 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 (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.

element	key
3	71
7	23

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`.

→

Die Prioritätswarteschlange (II)

Prioritätswarteschlange (priority queue) (Forts.)

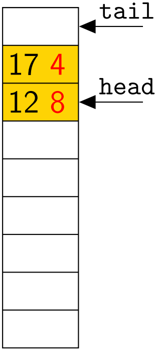
- `void delMin(PriorityQueue pq)` entfernt das Element mit dem kleinsten Schlüssel, benötigt nicht-leere `pq`.
- `Element getElt(PriorityQueue pq, int k)` gibt das Element `e` mit dem Schlüssel `k` aus `pq` zurück; `k` muss in `pq` enthalten sein.
- `void decrKey(PriorityQueue pq, Element e, int k)` 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, ...

Vorbedingung 1

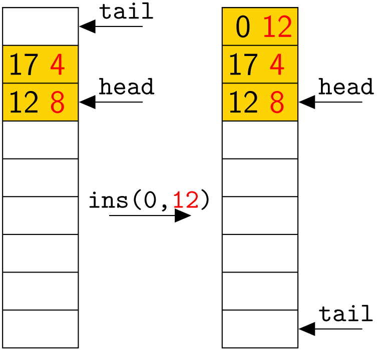
Vorbedingung 2

Prioritätswarteschlange, unsortiert, auf beschränkten Arrays



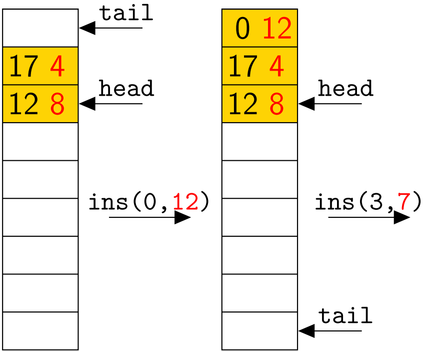
schwarz = Element; rot = Schlüssel

Prioritätswarteschlange, unsortiert, auf beschränkten Arrays



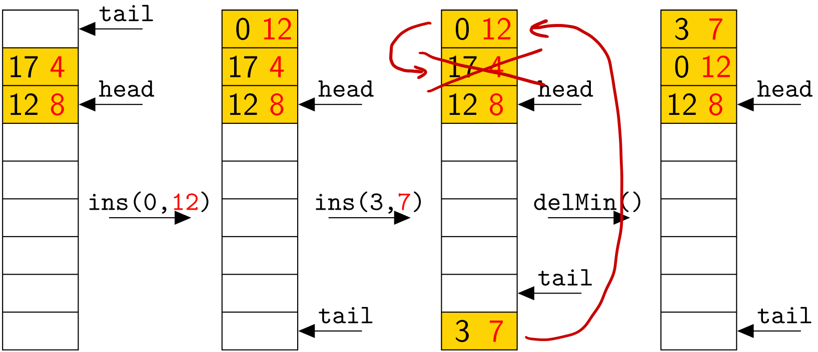
schwarz = Element; rot = Schlüssel

Prioritätswarteschlange, unsortiert, auf beschränkten Arrays



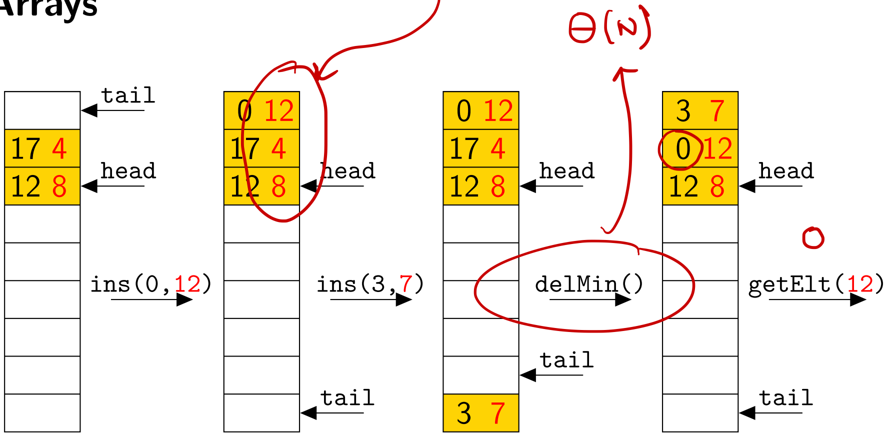
schwarz = Element; rot = Schlüssel

Prioritätswarteschlange, unsortiert, auf beschränkten Arrays



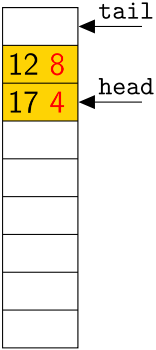
schwarz = Element; rot = Schlüssel

Prioritätswarteschlange, unsortiert, auf beschränkten Arrays

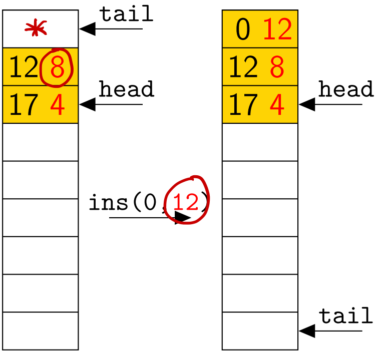


schwarz = Element; rot = Schlüssel

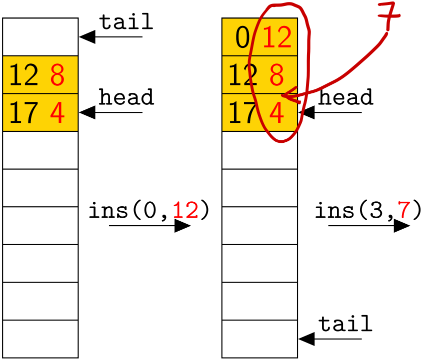
Prioritätswarteschlange, sortiert, auf beschränkten Arrays



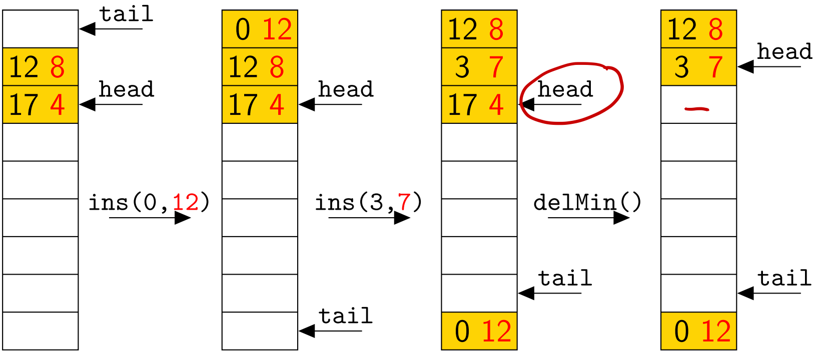
Prioritätswarteschlange, sortiert, auf beschränkten Arrays



Prioritätswarteschlange, sortiert, auf beschränkten Arrays

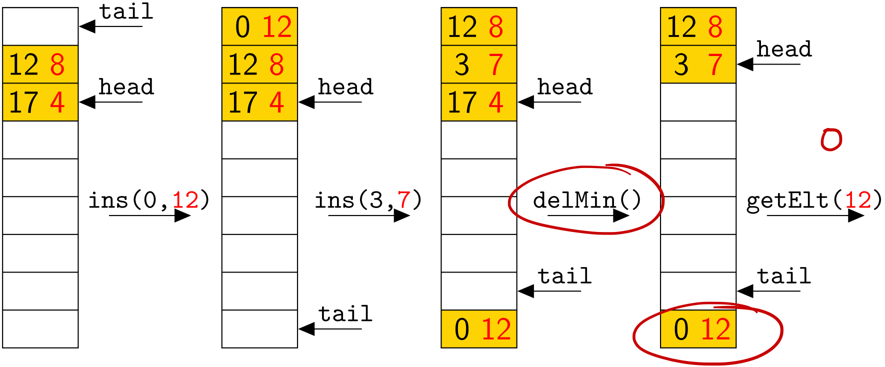


Prioritätswarteschlange, sortiert, auf beschränkten Arrays



Prioritätswarteschlange, sortiert, auf beschränkten Arrays

$\Theta(1)$



Zwei Prioritätswarteschlangenimplementierungen

	Implementierung	
Operation	unsortiertes Array	sortiertes Array
`isEmpty(pq)`		
`insert(pq,e,k)`		
`getMin(pq)`		
`delMin(pq)`		
`getElt(pq,k)`		
`decrKey(pq,e,k)`		

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

*

†

Zwei Prioritätswarteschlangenimplementierungen

	Implementierung	
Operation	unsortiertes Array	sortiertes Array
`isEmpty(pq)`	$\Theta(1)$	$\Theta(1)$
`insert(pq,e,k)`		
`getMin(pq)`		
`delMin(pq)`		
`getElt(pq,k)`		
`decrKey(pq,e,k)`		

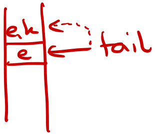
- In Vorlesung 8 (Heapsort) werden wir eine weitere Implementierung kennenlernen.

*

†

Zwei Prioritätswarteschlangenimplementierungen

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



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

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

†

Zwei Prioritätswarteschlangenimplementierungen

	Implementierung	
Operation	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)`		
`getElt(pq,k)`		
`decrKey(pq,e,k)`		

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

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

†

Zwei Prioritätswarteschlangenimplementierungen

	Implementierung	
Operation	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)`		
`decrKey(pq,e,k)`		

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

*Beinhaltet das Verschieben aller Elemente „rechts" von `k`.
†

Zwei Prioritätswarteschlangenimplementierungen

	Implementierung	
Operation	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)`		

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

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

†Mittels binärer Suche.

Zwei Prioritätswarteschlangenimplementierungen

	Implementierung	
Operation	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.

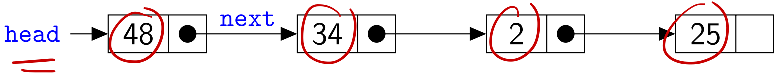
Übersicht

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 (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 (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 (III)

Liste (Untertypen)

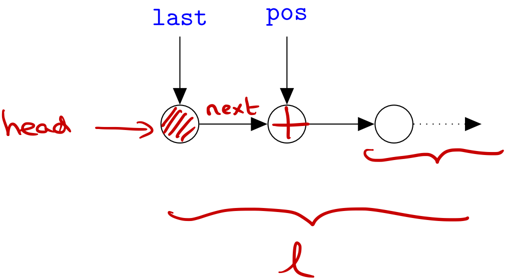
Die Elementreihenfolge kann auf unterschiedliche Arten definiert werden – wie z. B. durch folgende (strengere) Spezifikationen der `insert` Operation:

- `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 Schlüsseln 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.

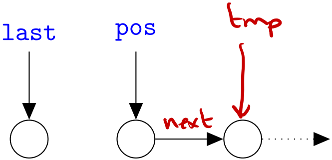
Listen umdrehen

```
void reverse(List l) {
  last = l.head;
  pos  = last.next;
  last.next = null;

  while(pos != null){
    tmp = pos.next;
    pos.next = last;
    last = pos;
    pos = tmp;
  }

  l.head = last;
}
```



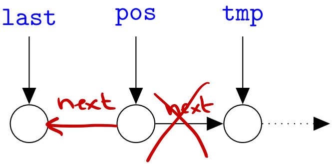
Listen umdrehen

```
void reverse(List l) {
  last = l.head;
  pos  = last.next;
  last.next = null;

  while(pos != null){
    tmp = pos.next;
    pos.next = last;
    last = pos;
    pos = tmp;
  }

  l.head = last;
}
```



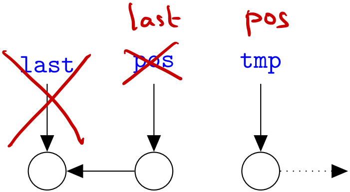
Listen umdrehen

```
void reverse(List l) {
  last = l.head;
  pos  = last.next;
  last.next = null;

  while(pos != null){
    tmp = pos.next;
    pos.next = last;
    last = pos;
    pos = tmp;
  }

  l.head = last;
}
```



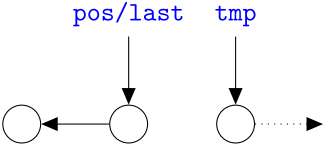
Listen umdrehen

```
void reverse(List l) {
  last = l.head;
  pos  = last.next;
  last.next = null;

  while(pos != null){
    tmp = pos.next;
    pos.next = last;
    last = pos;
    pos = tmp;
  }

  l.head = last;
}
```



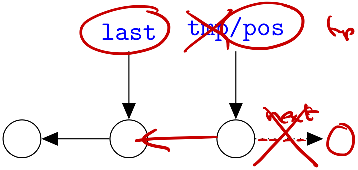
Listen umdrehen

```
void reverse(List l) {
  last = l.head;
  pos  = last.next;
  last.next = null;

  while(pos != null){
    tmp = pos.next;
    pos.next = last;
    last = pos;
    pos = tmp;
  }

  l.head = last;
}
```



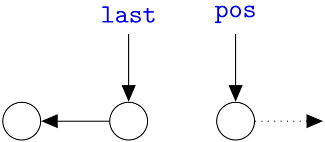
Listen umdrehen

```
void reverse(List l) {
  last = l.head;
  pos  = last.next;
  last.next = null;

  while(pos != null){
    tmp = pos.next;
    pos.next = last;
    last = pos;
    pos = tmp;
  }

  l.head = last;
}
```



Listen umdrehen

```
void reverse(List l) {
  last = l.head;
  pos  = last.next;
  last.next = null;

  while(pos != null){
    tmp = pos.next;
    pos.next = last;
    last = pos;
    pos = tmp;
  }

  l.head = last;
}
```

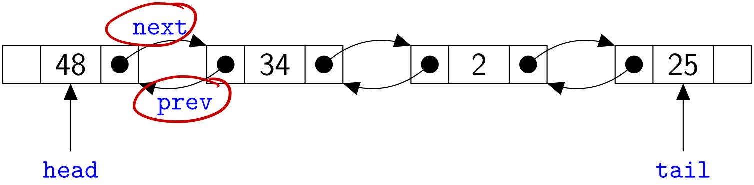


Doppelt verkettete Listen

Doppelt verkettete Liste

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

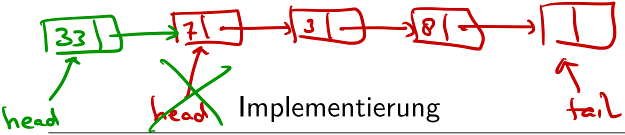
- Elemente besitzen neben Schlüssel und Nachfolgender einen weiteren Zeiger auf das vorherige Element.
- Zusätzlicher Zeiger `tail` zeigt auf das letzte Element der Liste



Laufzeiten

	Implementierung	
Operation	einfach verkette Liste	doppelt verkettete Liste
`insert(L,x,k)`		
`remove(L,k)`		
`search(L,k)`		
`minimum(L)`		
`maximum(L)`		
`successor(L,k)`		
`predecessor(L,k)`		

Laufzeiten



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

Laufzeiten

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

Laufzeiten

	Implementierung	
Operation	einfach verkette 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)`		
`maximum(L)`		
`successor(L,k)`		
`predecessor(L,k)`		

Laufzeiten

	Implementierung	
Operation	einfach verkette 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)`		
`successor(L,k)`		
`predecessor(L,k)`		

Laufzeiten

	Implementierung	
Operation	einfach verkette 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)`		
`predecessor(L,k)`		

Laufzeiten

	Implementierung	
Operation	einfach verkette 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)`		

Laufzeiten

	Implementierung	
Operation	einfach verkette 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)`	$\Theta(n)$	$\Theta(n)$

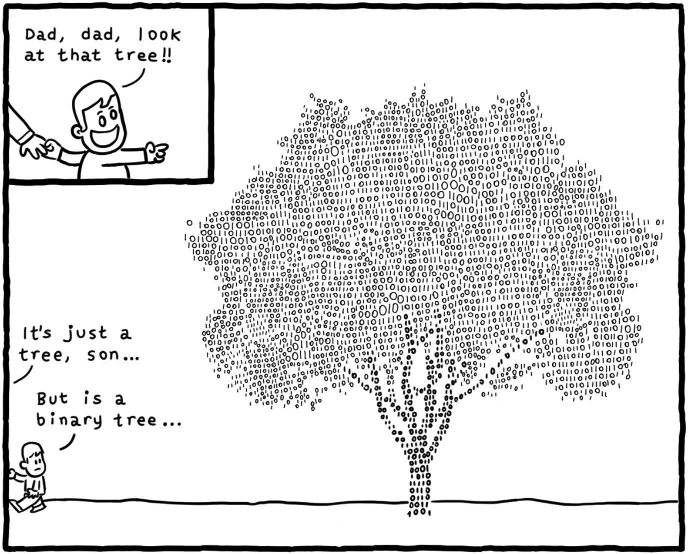
Laufzeiten

	Implementierung	
Operation	einfach verkette 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?

Übersicht

Binärbäume



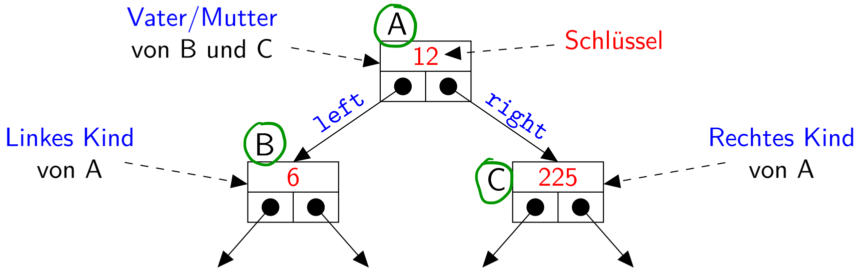
Daniel Stori {turnoff.us}

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:

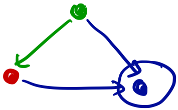


Binärbäume – Definition

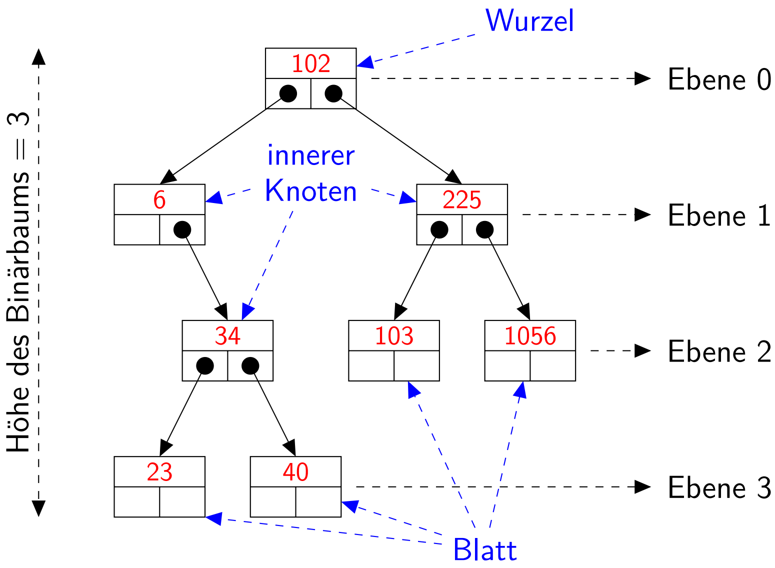
Definition (Binärbaum)

Ein Binärbaum (binary tree) ist ein gerichteter, zykelfreier 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. binär
- Der Ausgangsgrad (out-degree) jedes Knotens ist höchstens 2.
- Der Eingangsgrad eines Knoten ist 1, bzw. 0 bei der Wurzel.
- Sonderfall: Baum mit $V = E = \varnothing$.



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

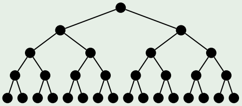
Binärbäume – Begriffe (II)

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

Beispiel (Vorteile von binären Bäumen)

Angenommen, man möchte 31 Elemente vorhalten:



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

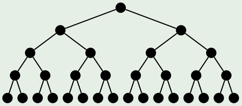
Binärbäume – Begriffe (II)

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

Beispiel (Vorteile von binären Bäumen)

Angenommen, man möchte 31 Elemente vorhalten:



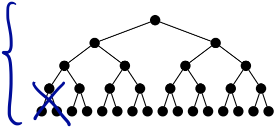
	Gesamt:
Ebene 0 (Wurzel) enthält 1 Element	
Ebene 1 enthält 2 Elemente	3
Ebene 2 enthält 4 Elemente	7
Ebene 3 enthält 8 Elemente	15
Ebene 4 enthält 16 Elemente	31

$\Rightarrow$ Ein Element kann in 5 Schritten (statt 31) erreicht werden.

Einige Fakten über binäre Bäume

$2^5 - 1$

$= 31$



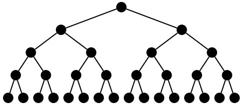
$\leftarrow 4$

$\leq 2^4$

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



$h=4$

$2^{4+1}-1=31$

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)

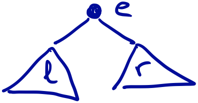
Ein Binärbaum heißt vollständig, wenn er bei Höhe h alle $2^{h+1}-1$ Knoten enthält.

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

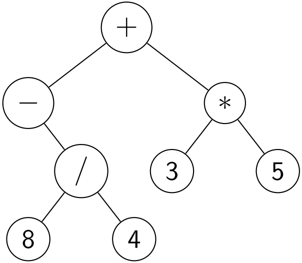


l, r, e

l, e, r

e, l, r

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

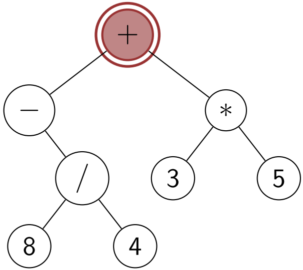
l
e
r

Beispiel

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

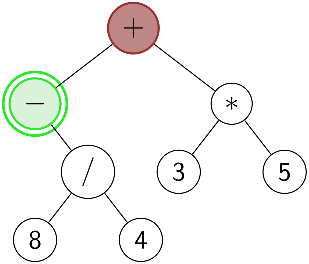
Beispiel

(

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

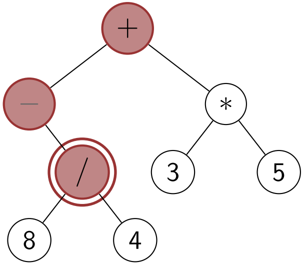
Beispiel

$((-$

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

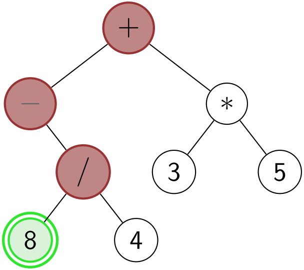
Beispiel

$((-($

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

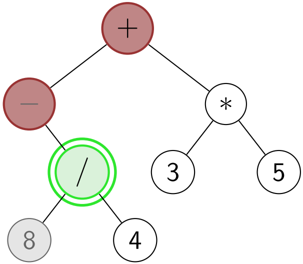
Beispiel

$((-(8$

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

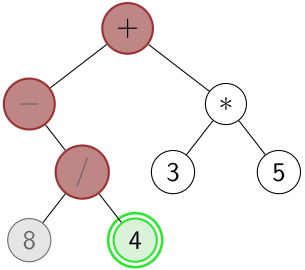
Beispiel

$((-(8/$

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

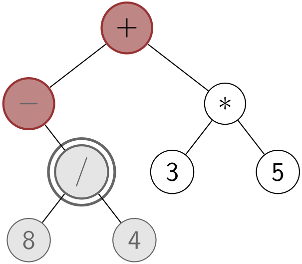
Beispiel

$((-(8/4$

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

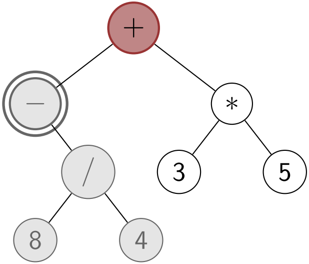
Beispiel

$((-(8/4)$

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

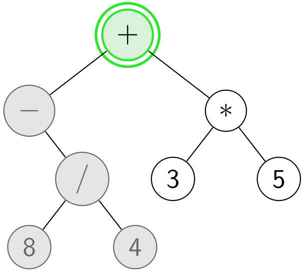
Beispiel

$((-(8/4))$

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

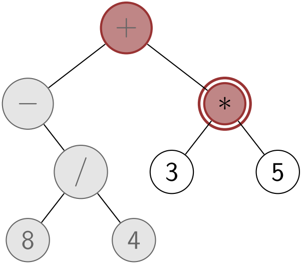
Beispiel

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

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

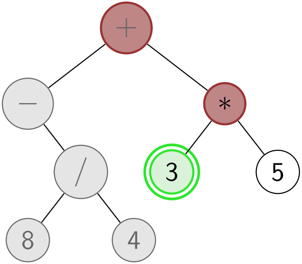
Beispiel

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

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

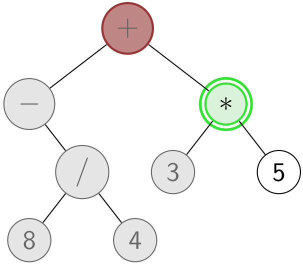
Beispiel

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

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

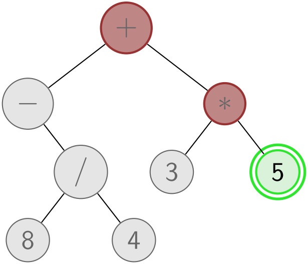
Beispiel

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

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

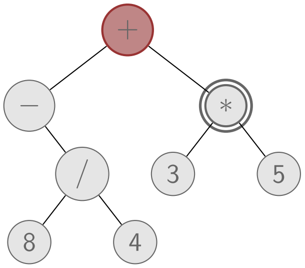
Beispiel

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

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

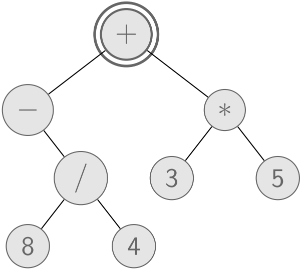
Beispiel

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

Linearisierung

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

Inorder-Traversierung




```
void inorder(Node node) {
  if (node != null) {
    "("
    inorder(node.left);
    print(node);
    inorder(node.right);
    ")"
  }
}
```

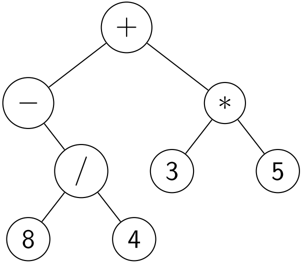
Beispiel

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

Linearisierung

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

Preorder, Inorder, Postorder (I)

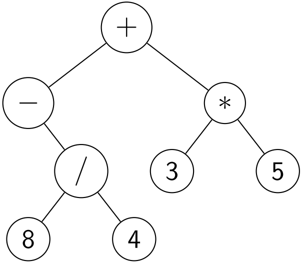



```
void inorder(Node node) {
  if (node != null) {
    inorder(node.left);
    print(node);
    inorder(node.right);
  }
}
```

Beispiel (Inorder)

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

Preorder, Inorder, Postorder (I)




```
void preorder(Node node) {
  if (node != null) {
    print(node);
    preorder(node.left);
    preorder(node.right);
  }
}
```

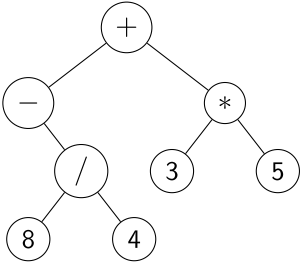
Beispiel (Inorder)

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

Beispiel (Preorder)

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

Preorder, Inorder, Postorder (I)




```
void postorder(Node node) {
  if (node != null) {
    postorder(node.left);
    postorder(node.right);
    print(node);
  }
}
```

Beispiel (Inorder)

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

Beispiel (Preorder)

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

Beispiel (Postorder – Umgekehrte Polnische Notation (RPN))

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

† neg

Preorder, Inorder, Postorder-Traversierung

```
void preorder(Node node) {
  if (node != null) {
    visit(node);
    preorder(node.left);
    preorder(node.right);
  }
}

void inorder(Node node) {
  if (node != null) {
    inorder(node.left);
    visit(node);
    inorder(node.right);
  }
}
```

```
void postorder(Node node) {
  if (node != null) {
    postorder(node.left);
    postorder(node.right);
    visit(node);
  }
}
```

Komplexität

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

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, 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

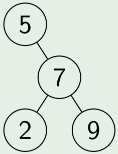
Preorder, Inorder, Postorder (II)

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



Zusammenfassung

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

Nächste Vorlesung

Nächste Vorlesung

Freitag 27. April, 13:15 (H01). Bis dann!