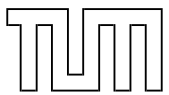
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2010

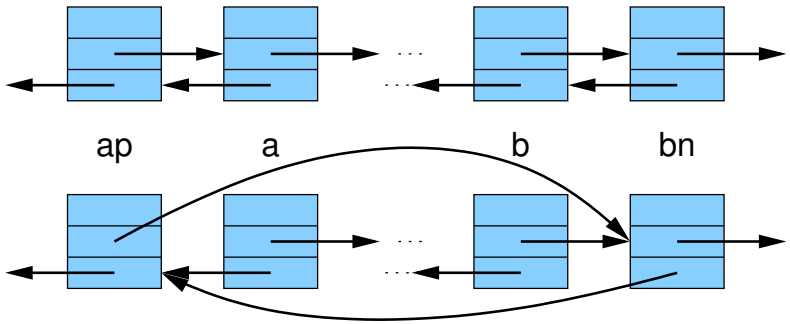


Übersicht

Doppelt verkettete Liste

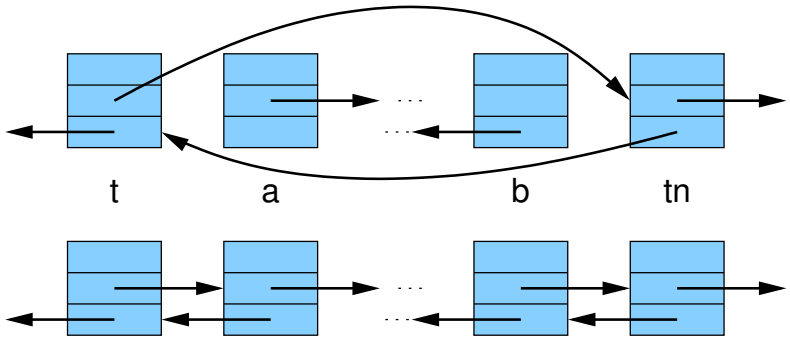
// schneide $\langle a, \ldots, b \rangle$ heraus

```
Item<Elem> ap = a.prev;
Item<Elem> bn = b.next;
ap.next = bn;
bn.prev = ap;
```



// füge $\langle a, \ldots, b \rangle$ hinter t ein

```
Item<Elem> tn = t.next;
b.next = tn;
a.prev = t;
t.next = a;
tn.prev = b;
```

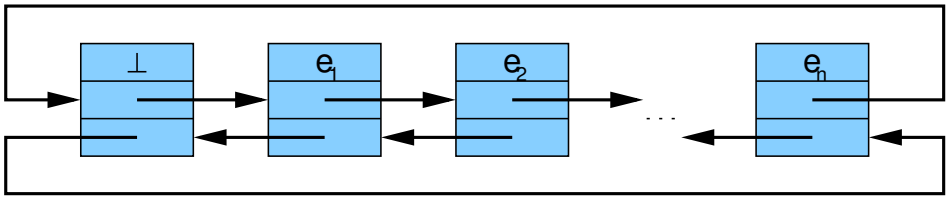


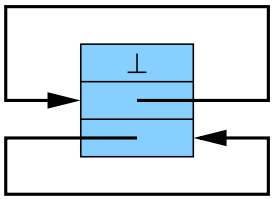
Doppelt verkettete Liste

h: Item mit null

Methoden:

```
Item<Elem> head() {
    return h;
}
boolean isEmpty() {
    return (h.next == h);
}
Item<Elem> first() {
    return h.next;          // evt. ⊥
}
Item<Elem> last() {
    return h.prev;          // evt. ⊥
}
```

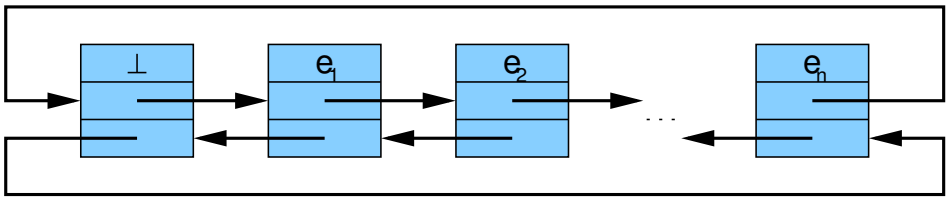




Doppelt verkettete Liste

h: Item mit null

Methoden:




```
static void moveAfter (Item<Elem> b, Item<Elem> a) {
    // schiebe b hinter a
    splice(b, b, a);
}

void moveToFront (Item<Elem> b) {
    // schiebe b ganz nach vorn
    moveAfter(b, head());

    // analog moveToBack
}
```

Doppelt verkettete Liste

Löschen und Einfügen von Elementen:

mittels separater Liste freeList
⇒ bessere Laufzeit (Speicherallokation teuer)

```
static void remove(Item<Elem> b) {
    moveAfter(b, freeList.head());
}

void popFront() {
    remove(first());
}

void popBack() {
    remove(last());
}
```

Doppelt verkettete Liste

```
static Item<Elem> insertAfter(Elem e, Item<Elem> a) {
    checkFreeList();       // u.U. Speicher allokieren
    Item<Elem> it = freeList.first();
    moveAfter(it, a);
    it.e = e;
    return it;
}

static Item<Elem> insertBefore(Elem e, Item<Elem> b) {
    return insertAfter(e, b.prev);
}
```

Doppelt verkettete Liste

```
void pushFront(Elem e) {
    insertAfter(e, head());
}

void pushBack(Elem e) {
    insertAfter(e, last());
}
```

Doppelt verkettete Liste

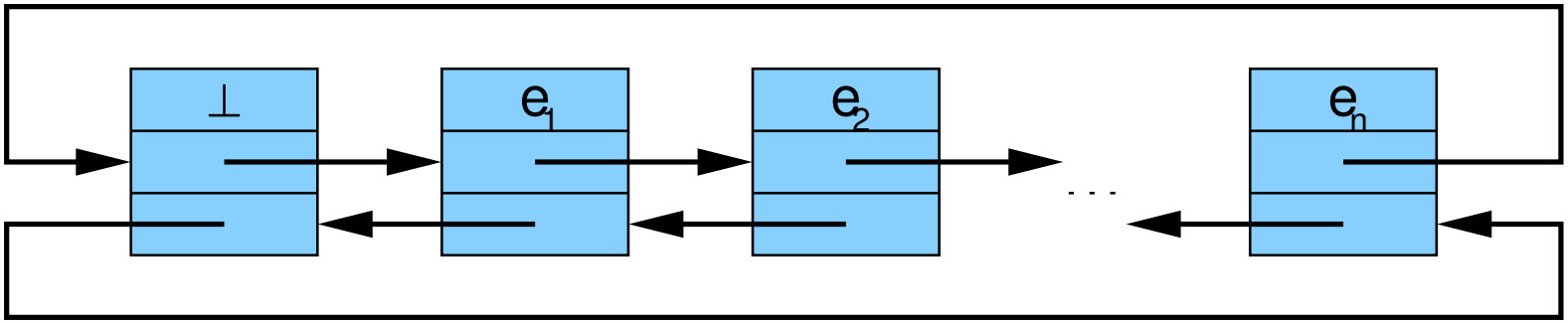
Manipulation ganzer Listen:

Trick: verwende Dummy-Element

```
Item<Elem> findNext(Elem x, Item<Elem> from) {
    h.e = x;
    while (from.e != x)
        from = from.next;
    h.e = null;
    return from;
}
```

Einfach verkettete Liste

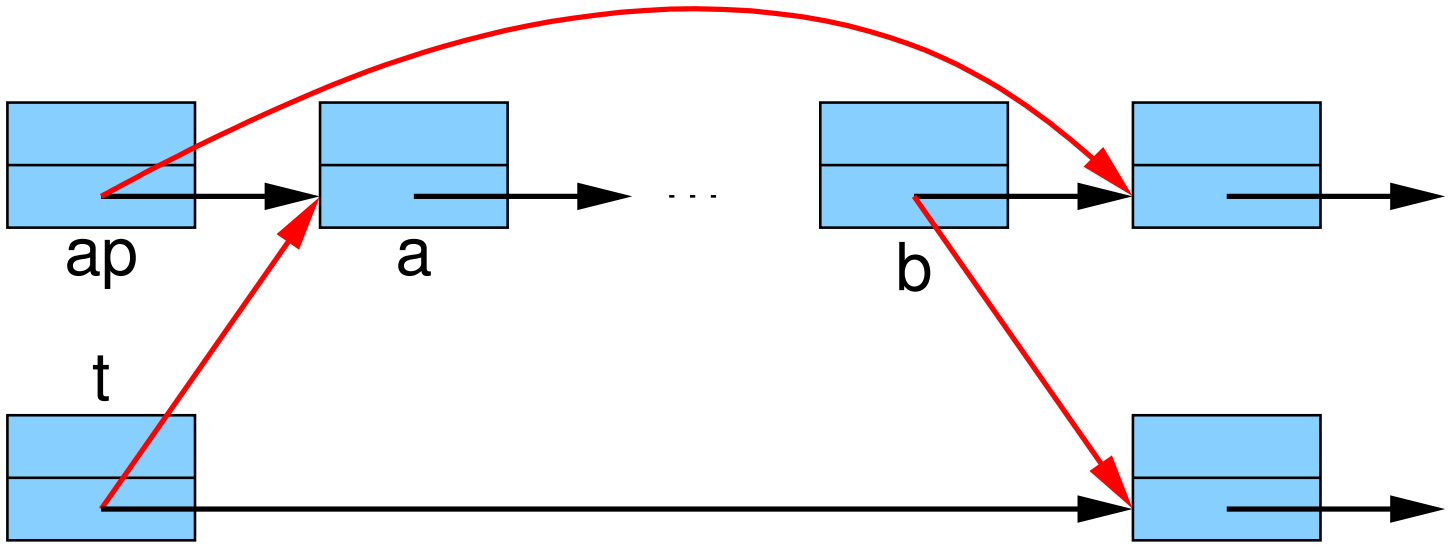
```
class SItem<Elem> {
    Elem e;
    SItem<Elem> h;
}
```




```
class SList<Elem> {
    SItem<Elem> h;
    . . . weitere Variablen und Methoden . . .
}
```

Einfach verkettete Liste

```
static void splice(SItem ap, SItem b, SItem t) {
    SItem a = ap.next;
    ap.next = b.next;
    b.next = t.next;
    t.next = a;
}
```



Einfach verkettete Liste

- findNext sollte evt. auch nicht den nächsten Treffer, sondern dessen Vorgänger liefern
  (damit man das gefundene Item auch löschen kann, Suche könnte dementsprechend erst beim Nachfolger des gegebenen Items starten)

- auch einige andere Methoden brauchen ein modifiziertes Interface

- sinnvoll: Pointer zum letzten Item

$\Rightarrow$ pushBack in $\mathcal{O}(1)$