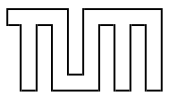
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

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

Sommersemester 2010



Übersicht

Sequenzen

Sequenz:

$$s = \langle e_0, \ldots, e_{n-1} \rangle$$

Zugriff auf Elemente:

- Feldrepräsentation: direkter Zugriff über $s[i]$
  - Nachteil: dynamische Größenänderung schwierig
- Listenrepräsentation: indirekter Zugriff über Nachfolger und/oder Vorgänger
  - Nachteil: Elemente sind u.U. über den gesamten Speicher verteilt

Sequenz als Feld

Operationen:

- $\langle e_0, \ldots, e_{n-1} \rangle [i]$ liefert Referenz auf e_i
- $\langle e_0, \ldots, e_{n-1} \rangle .\mathrm{get}(i) = e_i$
- $\langle e_0, \ldots, e_{i-1}, e_i, \ldots, e_{n-1} \rangle .\mathrm{set}(i, e) = \langle e_0, \ldots, e_{i-1}, e, \ldots, e_{n-1} \rangle$
- $\langle e_0, \ldots, e_{n-1} \rangle .\mathrm{pushBack}(e) = \langle e_0, \ldots, e_{n-1}, e \rangle$
- $\langle e_0, \ldots, e_{n-1} \rangle .\mathrm{popBack}() = \langle e_0, \ldots, e_{n-2} \rangle$
- $\langle e_0, \ldots, e_{n-1} \rangle .\mathrm{size}() = n$

Sequenz als Feld

Problem: beschränkter Speicher

- Feld:

andere Daten	8	3	9	7	4			andere Daten

- pushBack(1), pushBack(5), pushBack(2):

andere Daten	8	3	9	7	4	1	5	2	andere Daten

- pushBack(6): voll!

Sequenz als Feld

Problem:

- Beim Anlegen des Felds ist nicht bekannt, wieviele Elemente es enthalten wird
- Nur Anlegen von statischen Feldern möglich
  ($s =$ new ElementTyp[w])

Lösung: Datenstruktur für dynamisches Feld

Dynamisches Feld

Erste Idee:

- Immer dann, wenn Feld s nicht mehr ausreicht ($n > w - 1$): generiere neues Feld der Größe $w + c$ für ein festes c

$s[0]$	$s[1]$	$s[2]$	$\ldots$	$s[w-1]$	andere Daten

$\Downarrow$ Neues größeres Feld, Kopieren

$s[0]$	$s[1]$	$s[2]$	$\ldots$	$s[w-1]$	$s[w]$	$\ldots$	$s[w+c-1]$

Dynamisches Feld

Zeitaufwand für Erweiterung: $\mathcal{O}(w + c) = \mathcal{O}(w)$

$s[0]$	$s[1]$	$s[2]$	$\ldots$	$s[w-1]$	andere Daten

$\Downarrow$ Neues größeres Feld, Kopieren

$s[0]$	$s[1]$	$s[2]$	$\ldots$	$s[w-1]$	$s[w]$	$\ldots$	$s[w+c-1]$

Zeitaufwand für n pushBack Operationen:

- Aufwand von $\mathcal{O}(w)$ nach jeweils c Operationen
- Gesamtaufwand:

$$\mathcal{O}\left(\sum_{i=1}^{n/c} c \cdot i\right) = \mathcal{O}\left(n^2\right)$$

Dynamisches Feld

Bessere Idee:

- Immer dann, wenn Feld s nicht mehr ausreicht ($n > w - 1$): generiere neues Feld der doppelten Größe $2w$

$s[0]$	$s[1]$	$s[2]$	$\ldots$	$s[w-1]$	andere Daten

$\Downarrow$ Neues größeres Feld, Kopieren

$s[0]$	$s[1]$	$s[2]$	$\ldots$	$s[w-1]$	$s[w]$	$\ldots$	$s[2w-1]$

- Immer dann, wenn Feld s zu groß ist ($n < w/4$): generiere neues Feld der halben Größe $w/2$

Dynamisches Feld

Implementierung als Klasse **UArray** mit den Methoden:

- ElementTyp **get**(int i)
- int **size**()
- void **pushBack**(ElementTyp e)
- void **popBack**()
- void **reallocate**(int new_w)

Dynamisches Feld

Implementierung als Klasse **UArray** mit den Elementen:

- **beta** $= 2$ // Wachstumsfaktor
- **alpha** $= 4$ // max. Speicheroverhead
- **w** $= 1$ // momentane Feldgröße
- **n** $= 0$ // momentane Elementanzahl
- **b** $=$ new ElementTyp[w]

$b[0]$	$b[1]$	$b[2]$	$\ldots$	$b[w-1]$

Dynamisches Feld

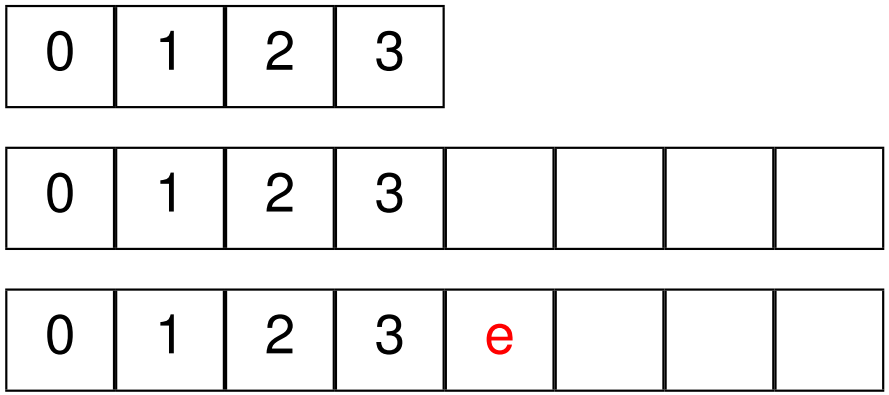
```
ElementTyp get(int i) {
    assert(0<=i && i<n);
    return b[i];
}
```

```
int size() {
    return n;
}
```

Dynamisches Feld

```
void pushBack(ElementTyp e) {
    if (n==w)
        reallocate(beta*n);
    b[n]=e;
    n=n+1; }
```

n=4, w=4

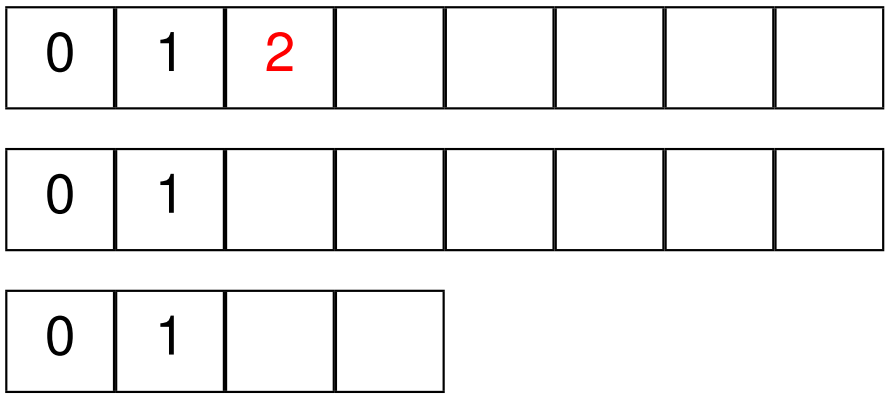


n=5, w=8

Dynamisches Feld

```
void popBack() {
    assert(n>0);
    n=n−1;
    if (alpha*n <= w && n>0)
        reallocate(beta*n);
}
```

n=3, w=8



n=2, w=4

Dynamisches Feld

```
void reallocate(int new_w) {
    w = new_w;
    ElementTyp[ ] new_b = new ElementTyp[new_w];
    for (i=0; i<n; i++)
        new_b[i] = b[i];
    b = new_b;
}
```

Dynamisches Feld

Wieviel Zeit kostet eine Folge von n pushBack-/popBack-Operationen?

Erste Idee:

- einzelne Operation kostet $\mathcal{O}(n)$
- Schranke kann nicht weiter gesenkt werden, denn reallocate-Aufrufe kosten jeweils $\Theta(n)$

$\Rightarrow$ also Gesamtkosten für n Operationen beschränkt durch $n \cdot \mathcal{O}(n) = \mathcal{O}(n^2)$

Dynamisches Feld

Wieviel Zeit kostet eine Folge von n pushBack-/popBack-Operationen?

Zweite Idee:

- betrachtete Operationen sollen direkt aufeinander folgen
- zwischen Operationen mit reallocate-Aufruf gibt es immer auch welche ohne

$\Rightarrow$ vielleicht ergibt sich damit gar nicht die n-fache Laufzeit einer Einzeloperation

Lemma

Betrachte ein anfangs leeres dynamisches Feld s.

Jede Folge $\sigma = \langle \sigma_1, \ldots, \sigma_n \rangle$ von pushBack- und popBack-Operationen auf s kann in Zeit $O(n)$ bearbeitet werden.

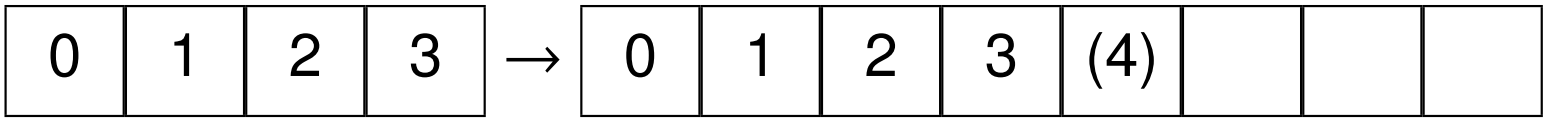
Dynamisches Feld

$\Rightarrow$ nur durchschnittlich konstante Laufzeit pro Operation

- Dabei werden Kosten teurer Operationen mit Kosten billiger Operation verrechnet.
- Man nennt das dann amortisierte Kosten bzw. amortisierte Analyse.

Dynamisches Feld: Analyse

- Feldverdopplung:



- Feldhalbierung:

- nächste Verdopplung: $\geq n$ pushBack-Operationen
- nächste Halbierung: $\geq n/2$ popBack-Operationen

Dynamisches Feld: Analyse

Formale Verrechnung: Zeugenzuordnung

- reallocate kann eine Vergrößerung oder Verkleinerung sein
- reallocate als Vergrößerung auf n Speicherelemente:
  es werden die $n/2$ vorangegangenen pushBack-Operationen zugeordnet
- reallocate als Verkleinerung auf n Speicherelemente:
  es werden die n vorangegangenen popBack-Operationen zugeordnet

$\Rightarrow$ kein pushBack/popBack wird mehr als einmal zugeordnet

Dynamisches Feld: Analyse

- Idee: verrechne reallocate-Kosten mit pushBack/popBack-Kosten (ohne reallocate)
  - Kosten für pushBack / popBack: $\mathcal{O}(1)$
  - Kosten für reallocate(k*n): $\mathcal{O}(n)$
- Konkret:
  - $\Theta(n)$ Zeugen pro reallocate(k$*$n)
  - verteile $\mathcal{O}(n)$-Aufwand gleichmäßig auf die Zeugen
- Gesamtaufwand: $\mathcal{O}(m)$ bei m Operationen

Dynamisches Feld: Analyse

Kontenmethode

- günstige Operationen zahlen Tokens ein
- teure Operationen entnehmen Tokens
- Tokenkonto darf nie negativ werden!

Dynamisches Feld: Analyse

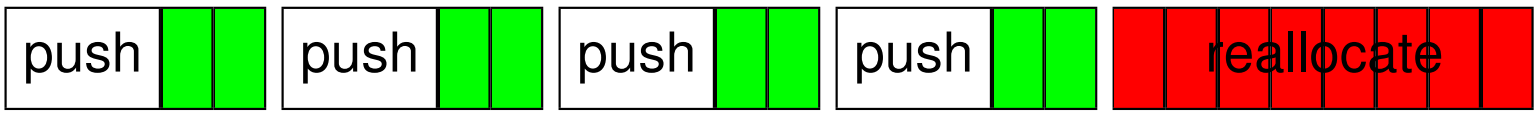
Kontenmethode

- günstige Operationen zahlen Tokens ein
  - pro pushBack 2 Tokens
  - pro popBack 1 Token
- teure Operationen entnehmen Tokens
  - pro reallocate(k∗n) −n Tokens
- Tokenkonto darf nie negativ werden!
  - Nachweis über Zeugenargument

Dynamisches Feld: Analyse

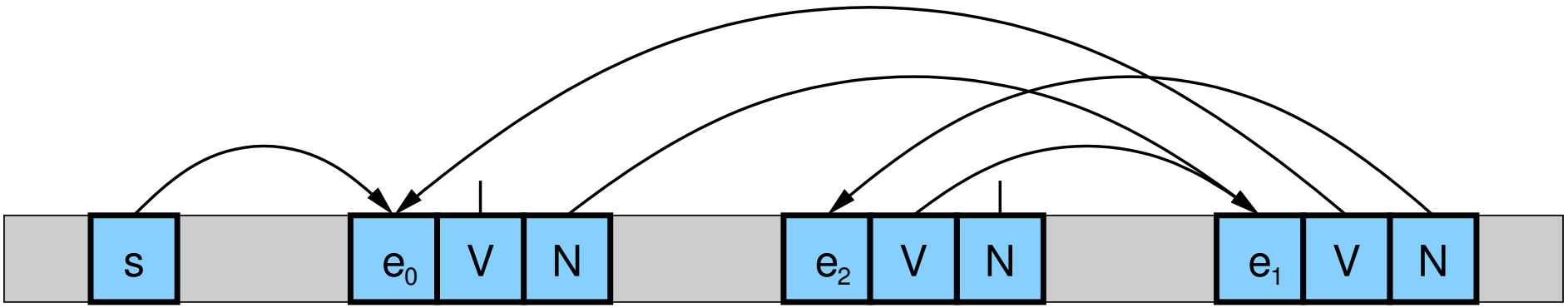
Tokenlaufzeit (Reale Kosten + Ein-/Auszahlungen)

- Ausführung von pushBack / popBack kostet 1 Token
  - Tokenkosten für pushBack: 1+2=3 Tokens
  - Tokenkosten für popBack: 1+1=2 Tokens
- Ausführung von reallocate(k∗n) kostet n Tokens
  - Tokenkosten für reallocate(k∗n): n−n=0 Tokens



- Gesamtlaufzeit = $\mathcal{O}$(Summe der Tokenlaufzeiten)

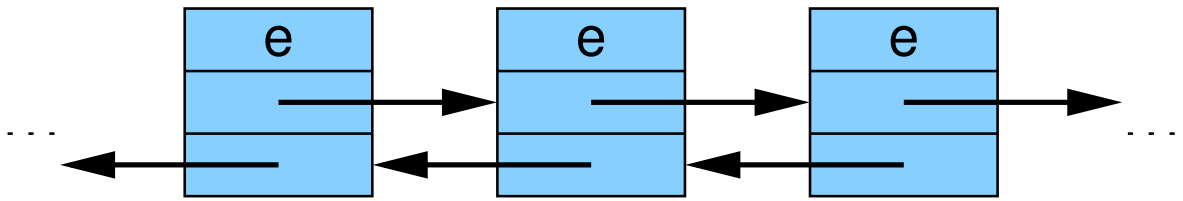
Doppelt verkettete Liste



Variable s speichert Startpunkt der Liste

Doppelt verkettete Liste

```
class Item<Elem> {
    Elem e;
    Item<Elem> next;
    Item<Elem> prev;
}
```




```
class List<Elem> {
    Item<Elem> h;
    ...weitere Variablen und Methoden ...
}
```

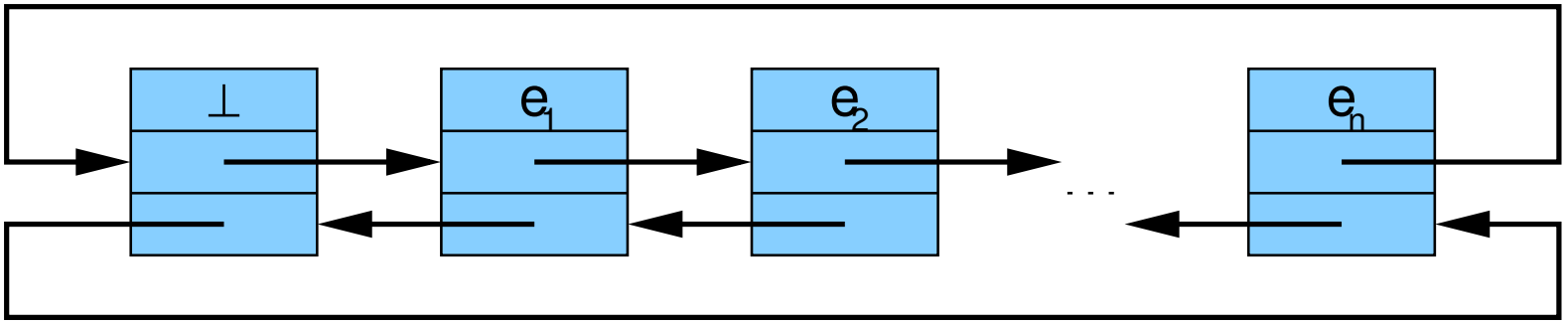
Invariante:

next.prev == prev.next == this

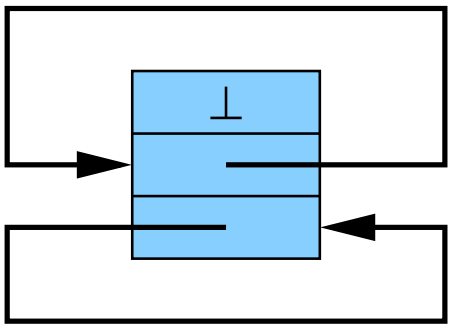
Doppelt verkettete Liste

Einfache Verwaltung:

durch Dummy-Element ohne Inhalt:



Anfangs:



Doppelt verkettete Liste

Zentrale statische Methode: splice

- splice entfernt $\langle a, \ldots, b \rangle$ aus der Sequenz und fügt sie hinter Item t an
- Bedingung:
  - $\langle a, \ldots, b \rangle$ muss eine Teilsequenz sein
  - b nicht vor a
  - t darf nicht in $\langle a, \ldots, b \rangle$ stehen

Für

$$\langle e_1, \ldots a', a, \ldots, b, b', \ldots, t, t', \ldots, e_n \rangle$$

liefert splice(a,b,t)

$$\langle e_1, \ldots a', b', \ldots, t, a, \ldots, b, t', \ldots, e_n \rangle$$