Script generated by TTT

Title: TÃ¤ubig: GAD (02.05.2013)

Date: Thu May 02 12:10:53 CEST 2013

Duration: 37:48 min

Pages: 21

Dynamisches Feld

Erste Idee:

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

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

$\Downarrow$ Kopieren in neues größeres Feld

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

Dynamisches Feld

Zeitaufwand für Erweiterung: $O(w + c) = O(w)$

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

$\Downarrow$ Kopieren in neues größeres Feld

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

Zeitaufwand für n pushBack Operationen:

- Aufwand von $O(w)$ nach jeweils c Operationen
- Gesamtaufwand:

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

Dynamisches Feld

Bessere Idee:

- Immer dann, wenn Feld s nicht mehr ausreicht:
  generiere neues Feld der doppelten Größe $2w$

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

$\Downarrow$ Kopieren in neues größeres Feld

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

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

Dynamisches Feld

Implementierung

Klasse **UArray** mit den Methoden:

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

- ElementTyp& **[**int i**]**
  auch möglich, aber Referenz nur bis zur nächsten Größenänderung des Felds gültig

Dynamisches Feld

Implementierung

Klasse **UArray** mit den Elementen:

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

$b[0]$	$b[1]$	$b[2]$	...	$b[w-1]$

Dynamisches Feld

Implementierung

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

set(int i, ElementTyp& e) {
   assert(0≤i<n);
   b[i] = e;
}

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


Dynamisches Feld

Implementierung

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

n=4, w=4

0	1	2	3

0	1	2	3				

0	1	2	3	e			

n=5, w=8

Dynamisches Feld

Implementierung

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

set(int i, ElementTyp& e) {
   assert(0≤i<n);
   b[i] = e;
}

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


Dynamisches Feld

Implementierung

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

n=4, w=4

0	1	2	3

0	1	2	3				

0	1	2	3	e			

n=5, w=8

Dynamisches Feld

Implementierung

```
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 $O(n)$
- Schranke kann nicht weiter gesenkt werden, denn reallocate-Aufrufe kosten jeweils $\Theta(n)$

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

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

Datenstrukturen für Sequenzen Felder

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.

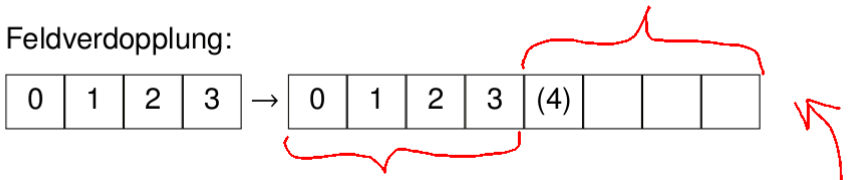
Dynamisches Feld

$\Rightarrow$ nur durchschnittlich konstante Laufzeit pro Operation

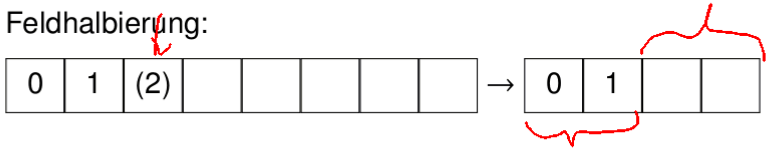
- Kosten teurer Operationen werden mit Kosten billiger Operationen verrechnet.
- Man nennt das dann amortisierte Kosten bzw. amortisierte Analyse.

Dynamisches Feld: Analyse

- Feldverdopplung:



- Feldhalbierung:



- nächste Verdopplung: nach $\geq n$ pushBack-Operationen
- nächste Halbierung: nach $\geq n/2$ 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: $O(1)$
  - Kosten für reallocate(k*n): $O(n)$
- Konkret:
  - $\Theta(n)$ Zeugen pro reallocate(k∗n)
  - verteile $O(n)$-Aufwand gleichmäßig auf die Zeugen
- Gesamtaufwand: $O(m)$ bei m Operationen

Dynamisches Feld: Analyse

Kontenmethode

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

Dynamisches Feld: Analyse

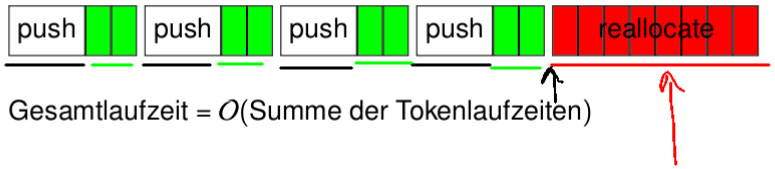
Kontenmethode

- 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 = O(Summe der Tokenlaufzeiten)

Übersicht