

Script generated by TTT

Title: Täubig: GAD (29.04.2014)

Date: Tue Apr 29 13:55:10 CEST 2014

Duration: 126:35 min

Pages: 71

Übersicht

3 Effizienz

- Effizienzmaße
- Rechenregeln für O -Notation
- Maschinenmodell
- Laufzeitanalyse
- Durchschnittliche Laufzeit
- Erwartete Laufzeit

Average Case Complexity

Uniforme Verteilung:
(alle Instanzen gleichwahrscheinlich)

$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i)$$

Tatsächliche Eingabeverteilung kann in der Praxis aber stark von uniformer Verteilung abweichen.

Dann

$$t(n) = \sum_{i \in \mathcal{I}_n} p_i \cdot T(i)$$

Aber: meist schwierig zu berechnen!

Beispiel: Binärzahl-Inkrementierung

Prozedur increment(A)

Eingabe : Array A mit Binärzahl in $A[0] \dots A[n-1]$,
in $A[n]$ steht eine 0

Ausgabe : inkrementierte Binärzahl in $A[0] \dots A[n]$

```
 $i = 0;$   
while ( $A[i] == 1$ ) do  
  |  $A[i] = 0;$   
  |  $i = i + 1;$   
 $A[i] = 1;$ 
```

Durchschnittliche Laufzeit für Zahl mit n Bits?

Binärzahl-Inkrementierung: Analyse

- \mathcal{I}_n : Menge der n -Bit-Instanzen
- Für die Hälfte (also $\frac{1}{2}|\mathcal{I}_n|$) der Zahlen $x_{n-1} \dots x_0 \in \mathcal{I}_n$ ist $x_0 = \underline{0}$
 \Rightarrow 1 Schleifendurchlauf
- Für die andere Hälfte gilt $x_0 = 1$.
 Bei diesen gilt wieder für die Hälfte (also $\frac{1}{4}|\mathcal{I}_n|$) $x_1 x_0 = \underline{01}$
 \Rightarrow 2 Schleifendurchläufe
- Für den Anteil $(\frac{1}{2})^k$ der Zahlen gilt $x_{k-1} x_{k-2} \dots x_0 = \underline{01 \dots 1}$
 \Rightarrow k Schleifendurchläufe

Durchschnittliche Anzahl Schleifendurchläufe:

$$t(n) = \frac{1}{|\mathcal{I}_n|} \sum_{i \in \mathcal{I}_n} T(i) = \frac{1}{|\mathcal{I}_n|} \sum_{k=1}^n \frac{|\mathcal{I}_n|}{2^k} \cdot k = \sum_{k=1}^n \frac{k}{2^k} \stackrel{?}{=} \mathcal{O}(1)$$

Binärzahl-Inkrementierung: Abschätzung

Lemma

$$\sum_{k=1}^n \frac{k}{2^k} \leq 2 - \frac{n+2}{2^n}$$

Binärzahl-Inkrementierung: Abschätzung

Lemma

$$\sum_{k=1}^n \frac{k}{2^k} \leq 2 - \frac{n+2}{2^n}$$

Beweis

Induktionsanfang:

Für $n = 1$ gilt: $\sum_{k=1}^1 \frac{k}{2^k} = \frac{1}{2} \leq 2 - \frac{1+2}{2^1}$ ✓

Binärzahl-Inkrementierung: Abschätzung

Lemma

$$\sum_{k=1}^n \frac{k}{2^k} \leq 2 - \frac{n+2}{2^n}$$

Beweis

Induktionsanfang:

Für $n = 1$ gilt: $\sum_{k=1}^1 \frac{k}{2^k} = \frac{1}{2} \leq 2 - \frac{1+2}{2^1}$ ✓

Induktionsvoraussetzung:

Für n gilt: $\sum_{k=1}^{n+1} \frac{k}{2^k} \leq 2 - \frac{n+2}{2^n}$

Binärzahl-Inkrementierung: Abschätzung

Beweis.

Induktionsschritt: $n \rightarrow n + 1$

$$\sum_{k=1}^{n+1} \frac{k}{2^k} = \left(\sum_{k=1}^n \frac{k}{2^k} \right) + \frac{n+1}{2^{n+1}}$$

Binärzahl-Inkrementierung: Abschätzung

Beweis.

Induktionsschritt: $n \rightarrow n + 1$

$$\begin{aligned} \sum_{k=1}^{n+1} \frac{k}{2^k} &= \left(\sum_{k=1}^n \frac{k}{2^k} \right) + \frac{n+1}{2^{n+1}} \\ &\leq \underline{2 - \frac{n+2}{2^n}} + \frac{n+1}{2^{n+1}} \quad (\text{laut Ind.vor.}) \end{aligned}$$

Binärzahl-Inkrementierung: Abschätzung

Beweis.

Induktionsschritt: $n \rightarrow n + 1$

$$\begin{aligned} \sum_{k=1}^{n+1} \frac{k}{2^k} &= \left(\sum_{k=1}^n \frac{k}{2^k} \right) + \frac{n+1}{2^{n+1}} \\ &\leq 2 - \frac{n+2}{2^n} + \frac{n+1}{2^{n+1}} \quad (\text{laut Ind.vor.}) \\ &= 2 - \frac{2(n+2)}{2^{n+1}} + \frac{n+1}{2^{n+1}} = 2 - \frac{2n+4-n-1}{2^{n+1}} \\ &= 2 - \frac{n+3}{2^{n+1}} \\ &= 2 - \frac{(n+1)+2}{2^{n+1}} \end{aligned}$$

□

Übersicht

3 Effizienz

- Effizienzmaße
- Rechenregeln für O -Notation
- Maschinenmodell
- Laufzeitanalyse
- Durchschnittliche Laufzeit
- Erwartete Laufzeit

Zufallsvariable

Definition

Für einen Wahrscheinlichkeitsraum mit Ergebnismenge Ω nennt man eine Abbildung $X : \Omega \mapsto \mathbb{R}$ (numerische) **Zufallsvariable**.

Zufallsvariable

Definition

Für einen Wahrscheinlichkeitsraum mit Ergebnismenge Ω nennt man eine Abbildung $X : \Omega \mapsto \mathbb{R}$ (numerische) **Zufallsvariable**.

Eine Zufallsvariable über einer endlichen oder abzählbar unendlichen Ergebnismenge heißt **diskret**.

Zufallsvariable

Definition

Für einen Wahrscheinlichkeitsraum mit Ergebnismenge Ω nennt man eine Abbildung $X : \Omega \mapsto \mathbb{R}$ (numerische) **Zufallsvariable**.

Eine Zufallsvariable über einer endlichen oder abzählbar unendlichen Ergebnismenge heißt **diskret**.

Der Wertebereich diskreter Zufallsvariablen

$$W_X := X(\Omega) = \{x \in \mathbb{R} \mid \exists \omega \in \Omega \text{ mit } X(\omega) = x\}$$

ist ebenfalls endlich bzw. abzählbar unendlich.

Zufallsvariable

Definition

Für einen Wahrscheinlichkeitsraum mit Ergebnismenge Ω nennt man eine Abbildung $X : \Omega \mapsto \mathbb{R}$ (numerische) **Zufallsvariable**.

Eine Zufallsvariable über einer endlichen oder abzählbar unendlichen Ergebnismenge heißt **diskret**.

Der Wertebereich diskreter Zufallsvariablen

$$W_X := X(\Omega) = \{x \in \mathbb{R} \mid \exists \omega \in \Omega \text{ mit } X(\omega) = x\}$$

ist ebenfalls endlich bzw. abzählbar unendlich.

Schreibweise: $\Pr[X = x] := \Pr[X^{-1}(x)] = \sum_{\omega \in \Omega \mid X(\omega) = x} \Pr[\omega]$

Zufallsvariable

Beispiel

Wir ziehen aus einem Poker-Kartenspiel mit 52 Karten (13 von jeder Farbe) eine Karte.

Wir bekommen bzw. bezahlen einen bestimmten Betrag, je nachdem welche Farbe die Karte hat, z.B. 4 Euro für Herz, 7 Euro für Karo, –5 Euro für Kreuz und –3 Euro für Pik.

Wenn wir ein As ziehen, bekommen wir zusätzlich 1 Euro.

Zufallsvariable

Beispiel

Wir ziehen aus einem Poker-Kartenspiel mit 52 Karten (13 von jeder Farbe) eine Karte.

Wir bekommen bzw. bezahlen einen bestimmten Betrag, je nachdem welche Farbe die Karte hat, z.B. 4 Euro für Herz, 7 Euro für Karo, –5 Euro für Kreuz und –3 Euro für Pik.

Wenn wir ein As ziehen, bekommen wir zusätzlich 1 Euro.

$\Omega = \{\heartsuit A, \heartsuit K, \dots, \heartsuit 2, \diamondsuit A, \diamondsuit K, \dots, \diamondsuit 2, \clubsuit A, \clubsuit K, \dots, \clubsuit 2, \spadesuit A, \spadesuit K, \dots, \spadesuit 2\}$.

X sei der Geldbetrag den wir bekommen bzw. bezahlen.

Zufallsvariable

Beispiel

Wir ziehen aus einem Poker-Kartenspiel mit 52 Karten (13 von jeder Farbe) eine Karte.

Wir bekommen bzw. bezahlen einen bestimmten Betrag, je nachdem welche Farbe die Karte hat, z.B. 4 Euro für Herz, 7 Euro für Karo, –5 Euro für Kreuz und –3 Euro für Pik.

Wenn wir ein As ziehen, bekommen wir zusätzlich 1 Euro.

$\Omega = \{\heartsuit A, \heartsuit K, \dots, \heartsuit 2, \diamondsuit A, \diamondsuit K, \dots, \diamondsuit 2, \clubsuit A, \clubsuit K, \dots, \clubsuit 2, \spadesuit A, \spadesuit K, \dots, \spadesuit 2\}$.

X sei der Geldbetrag den wir bekommen bzw. bezahlen.

$W_X = \{-5, -4, -3, -2, 4, 5, 7, 8\}$

$\Pr[X = -3] = \Pr[\spadesuit K] + \dots + \Pr[\spadesuit 2] = 12/52 = 3/13$

Erwartungswert

Definition

Für eine diskrete Zufallsvariable X ist der Erwartungswert definiert als

$$\mathbb{E}[X] := \sum_{x \in W_X} x \cdot \Pr[X = x] = \sum_{\omega \in \Omega} X(\omega) \cdot \Pr[\omega]$$

sofern $\sum_{x \in W_X} |x| \cdot \Pr[X = x]$ konvergiert (absolute Konvergenz).

Erwartungswert

Definition

Für eine diskrete Zufallsvariable X ist der **Erwartungswert** definiert als

$$\mathbb{E}[X] := \sum_{x \in W_X} x \cdot \Pr[X = x] = \sum_{\omega \in \Omega} X(\omega) \cdot \Pr[\omega]$$

sofern $\sum_{x \in W_X} |x| \cdot \Pr[X = x]$ konvergiert (absolute Konvergenz).

Bei endlicher Ereignismenge und gleichwahrscheinlichen Ereignissen entspricht der Erwartungswert dem **Durchschnitt**:

$$\mathbb{E}[X] = \sum_{x \in W_X} x \cdot \Pr[X = x] = \sum_{\omega \in \Omega} X(\omega) \cdot \frac{1}{|\Omega|} = \frac{1}{|\Omega|} \sum_{\omega \in \Omega} X(\omega)$$

Erwartungswert

Beispiel

(Beispiel wie zuvor)

$$\begin{aligned} \mathbb{E}[X] &= 4 \cdot \frac{12}{52} + 5 \cdot \frac{1}{52} + 7 \cdot \frac{12}{52} + 8 \cdot \frac{1}{52} \\ &\quad + (-5) \cdot \frac{12}{52} + (-4) \cdot \frac{1}{52} + (-3) \cdot \frac{12}{52} + (-2) \cdot \frac{1}{52} = \frac{43}{52} \end{aligned}$$

Wir bekommen also im Erwartungswert $\frac{43}{52}$ Euro pro gezogener Karte.



Grundlagen zu diskreter Wahrscheinlichkeitstheorie findet man z.B. in folgendem Buch:

Th. Schickinger, A. Steger
Diskrete Strukturen – Band 2
 (Wahrscheinlichkeitstheorie und Statistik)
 Springer-Verlag, 2001.

Erwartungswert

Beispiel

Münze werfen, bis sie zum ersten Mal Kopf zeigt

Zufallsvariable k : Anzahl der Versuche

k ungerade: Spieler bezahlt etwas an die Bank

k gerade: Spieler bekommt etwas von der Bank

Zufallsvariable X : Gewinnbetrag der Bank

Erwartungswert

Beispiel

Münze werfen, bis sie zum ersten Mal Kopf zeigt

Zufallsvariable k : Anzahl der Versuche

k ungerade: Spieler bezahlt etwas an die Bank

k gerade: Spieler bekommt etwas von der Bank

Zufallsvariable X : Gewinnbetrag der Bank

Variante 1: Spieler bezahlt/bekommt k Euro
 $\mathbb{E}[X]$ existiert (absolute Konvergenz)

Variante 2: Spieler bezahlt/bekommt 2^k Euro
 $\mathbb{E}[X]$ existiert nicht (keine Konvergenz)

Variante 3: Spieler bezahlt/bekommt $\frac{2^k}{k}$ Euro
 $\mathbb{E}[X]$ existiert nicht (Konvergenz, aber keine absolute)

Erwartungswert zusammengesetzter Zufallsvariablen

Satz (Linearität des Erwartungswerts)

Für Zufallsvariablen X_1, \dots, X_n und

$$X := a_1 X_1 + \dots + a_n X_n$$

mit $a_1, \dots, a_n \in \mathbb{R}$ gilt

$$\mathbb{E}[X] = a_1 \mathbb{E}[X_1] + \dots + a_n \mathbb{E}[X_n].$$

Interessant ist für uns vor allem der einfache Fall:

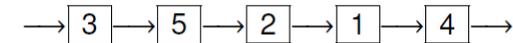
$$X := X_1 + \dots + X_n$$

mit

$$\mathbb{E}[X] = \mathbb{E}[X_1] + \dots + \mathbb{E}[X_n].$$



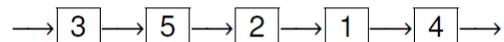
Beispiel: Suche in statischer Liste



- gegeben: Liste mit Elementen $1, \dots, m$
- $\text{search}(i)$: lineare Suche nach Element i ab Listenanfang
 - s_i Position von Element i in der Liste ($1 \hat{=}$ Anfang)
 - p_i Wahrscheinlichkeit für Zugriff auf Element i



Beispiel: Suche in statischer Liste



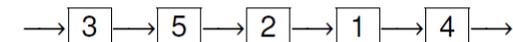
- gegeben: Liste mit Elementen $1, \dots, m$
- $\text{search}(i)$: lineare Suche nach Element i ab Listenanfang
 - s_i Position von Element i in der Liste ($1 \hat{=}$ Anfang)
 - p_i Wahrscheinlichkeit für Zugriff auf Element i

Erwartete Laufzeit der Operation $\text{search}(i)$ mit zufälligem i :

$$\mathbb{E}[T(\text{search}(i))] = O\left(\sum_i p_i s_i\right)$$



Beispiel: Suche in statischer Liste



- gegeben: Liste mit Elementen $1, \dots, m$
- $\text{search}(i)$: lineare Suche nach Element i ab Listenanfang
 - s_i Position von Element i in der Liste ($1 \hat{=}$ Anfang)
 - p_i Wahrscheinlichkeit für Zugriff auf Element i

Erwartete Laufzeit der Operation $\text{search}(i)$ mit zufälligem i :

$$\mathbb{E}[T(\text{search}(i))] = O\left(\sum_i p_i s_i\right)$$

Erwartete Laufzeit $t(n)$ für n Zugriffe bei **statischer** Liste:

$$t(n) = \mathbb{E}[T(n \times \text{search}(i))] = n \cdot \mathbb{E}[T(\text{search}(i))] = O\left(n \sum_i p_i s_i\right)$$



Beispiel: Suche in statischer Liste

Optimale Anordnung?

⇒ wenn für alle Elemente i, j mit $p_i > p_j$ gilt, dass $s_i < s_j$, d.h. die Elemente nach Zugriffswahrscheinlichkeit sortiert sind

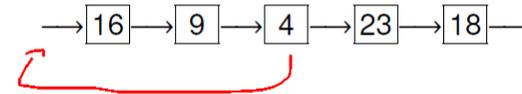
o.B.d.A. seien die Indizes so, dass $p_1 \geq p_2 \geq \dots \geq p_m$

- Optimale Anordnung: $s_i = i$
- Optimale erwartete Laufzeit: $\text{opt} = \sum_i p_i \cdot i$

Einfach: wenn die Zugriffswahrscheinlichkeiten bekannt sind
⇒ optimale erwartete Laufzeit durch absteigende Sortierung nach p_i

Problem: was wenn die Wahrscheinlichkeiten p_i unbekannt sind?

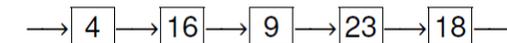
Beispiel: Suche in selbstorganisierender Liste



Move-to-Front Rule:

Verschiebe nach jeder erfolgreichen Suche das gefundene Element an den Listenanfang

Bsp.: Ausführung von `search(4)` ergibt



Beispiel: Suche in selbstorganisierender Liste

Erwartete Laufzeit $t(n)$ bei **dynamischer** Liste:

$$\mathbb{E}[T(\text{search}(i))] = O\left(\sum_i p_i \cdot \underline{\mathbb{E}[s_i]}\right)$$

Satz

Ab dem Zeitpunkt, wo auf jedes Element mindestens einmal zugegriffen wurde, ist die erwartete Laufzeit der `search`-Operation unter Verwendung der **Move-to-Front Rule** höchstens $2 \cdot \text{opt}$.

Beispiel: Suche in selbstorganisierender Liste

Beweis.

Betrachte zwei feste Elemente i und j

t_0 Zeitpunkt der letzten Suchoperation auf i oder j

- bedingte Wahrscheinlichkeit: $\Pr[A | B] = \frac{\Pr[A \wedge B]}{\Pr[B]}$
- $\Pr[C | (C \vee D)] = \frac{\Pr[C \wedge (C \vee D)]}{\Pr[C \vee D]} = \frac{\Pr[C]}{\Pr[C \vee D]}$
- $\Pr[\text{search}(j) \text{ bei } t_0 | \text{search}(i \vee j) \text{ bei } t_0] = \frac{p_j}{p_i + p_j}$
- mit Wsk. $\frac{p_i}{p_i + p_j}$ steht i vor j und mit Wsk. $\frac{p_j}{p_i + p_j}$ steht j vor i

Beispiel: Suche in selbstorganisierender Liste

Beweis.

Betrachte nun nur ein festes Element i

- Definiere Zufallsvariablen $X_j \in \{0, 1\}$ für $j \neq i$:

$$X_j = 1 \Leftrightarrow j \text{ vor } i \text{ in der Liste}$$

- Erwartungswert:

$$\begin{aligned} \mathbb{E}[X_j] &= 0 \cdot \Pr[X_j = 0] + 1 \cdot \Pr[X_j = 1] \\ &= \Pr[\text{letzte Suche nach } i/j \text{ war nach } j] \\ &= \frac{p_j}{p_i + p_j} \end{aligned}$$

Beispiel: Suche in selbstorganisierender Liste

Beweis.

- Listenposition von Element i : $1 + \sum_{j \neq i} X_j$
- Erwartungswert der Listenposition von Element i :

$$\begin{aligned} \mathbb{E}[s_i] &= \mathbb{E}\left[1 + \sum_{j \neq i} X_j\right] \\ &= 1 + \mathbb{E}\left[\sum_{j \neq i} X_j\right] = 1 + \sum_{j \neq i} \mathbb{E}[X_j] \\ \mathbb{E}[s_{i, \text{MTF}}] &= 1 + \sum_{j \neq i} \frac{p_j}{p_i + p_j} \end{aligned}$$

Beispiel: Suche in selbstorganisierender Liste

Beweis.

Erwartete Laufzeit der search-Operation:

$$\begin{aligned} \mathbb{E}[T_{\text{MTF}}] &= \sum_i p_i \left(1 + \sum_{j \neq i} \frac{p_j}{p_i + p_j}\right) \\ &= \sum_i \left(p_i + \sum_{j \neq i} \frac{p_i p_j}{p_i + p_j}\right) = \sum_i \left(p_i + 2 \sum_{j < i} \frac{p_i p_j}{p_i + p_j}\right) \\ &= \sum_i p_i \left(1 + 2 \sum_{j < i} \frac{p_j}{p_i + p_j}\right) \leq \sum_i p_i \left(1 + 2 \sum_{j < i} 1\right) \\ &\leq \sum_i p_i \cdot (2i - 1) < \sum_i p_i \cdot 2i = 2 \cdot \text{opt} \end{aligned}$$

□

Übersicht

- 4 Datenstrukturen für Sequenzen
 - Felder
 - Listen
 - Stacks und Queues
 - Diskussion: Sortierte Sequenzen

Übersicht

- 4 Datenstrukturen für Sequenzen
 - Felder
 - Listen
 - Stacks und Queues
 - Diskussion: Sortierte Sequenzen

Sequenzen

Sequenz: lineare Struktur

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

(Gegensatz: verzweigte Struktur in Graphen, fehlende Struktur in Hashtab.)

Klassische Repräsentation:

- (Statisches) Feld / Array:
 - direkter** Zugriff über $s[i]$
 - Vorteil: Zugriff über Index, homogen im Speicher
 - Nachteil: dynamische Größenänderung schwierig
- Liste:
 - indirekter** Zugriff über Nachfolger / Vorgänger
 - Vorteil: Einfügen / Löschen von Teilsequenzen
 - Nachteil: kein Zugriff per Index, Elemente über Speicher verteilt

Sequenzen

Operationen:

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

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:

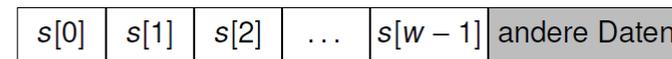
- Beim Anlegen des Felds ist nicht bekannt, wieviele Elemente es enthalten wird
- Nur Anlegen von **statischen** Feldern möglich
(`s = new ElementTyp[w]`)

Lösung: Datenstruktur für **dynamisches** Feld

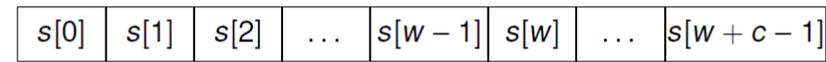
Dynamisches Feld

Erste Idee:

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



⇓ **Kopieren** in neues größeres Feld

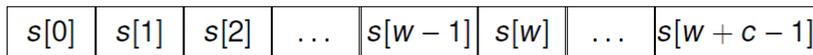


Dynamisches Feld

Zeitaufwand für Erweiterung: $\Theta(w)$



⇓ **Kopieren** in neues größeres Feld



Zeitaufwand für n pushBack Operationen:

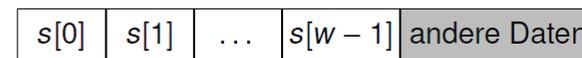
- Aufwand von $\Theta(w)$ nach jeweils `c` Operationen (wobei `w` immer größer wird)
- Gesamtaufwand:

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

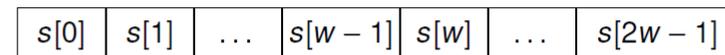
Dynamisches Feld

Bessere Idee:

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



⇓ **Kopieren** in neues größeres Feld



- 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:

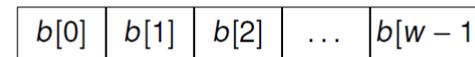
- ElementTyp **get**(int i)
 - **set**(int i, ElementTyp& e)
 - int **size**()
 - void **pushBack**(ElementTyp e)
 - void **popBack**()
 - void **reallocate**(int new_w)
-
- ElementTyp& **[int i]**
auch möglich, aber Referenz nur bis zur nächsten Größenänderung des Felds gültig

Dynamisches Feld

Implementierung

Klasse **UArray** mit den Elementen:

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



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

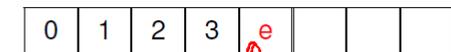
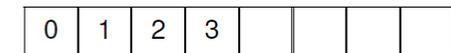
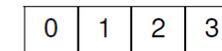
Implementierung

```

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

```

n=4, w=4



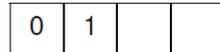
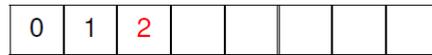
n=5, w=8

Dynamisches Feld

Implementierung

```
void popBack() {
    assert(n>0);
    n--;
    if ( $\alpha \cdot n \leq w \wedge n > 0$ )
        reallocate( $\beta \cdot n$ );
}
```

n=3, w=8



n=2, w=4



Dynamisches Feld

Implementierung

```
void reallocate(int new_w) {
    w = new_w;
    ElementType[] new_b = new ElementType[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 $\Theta(n)$

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



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
- ⇒ vielleicht ergibt sich damit gar nicht die n -fache Laufzeit einer Einzeloperation



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
- ⇒ vielleicht ergibt sich damit gar nicht die n -fache Laufzeit einer Einzeloperation

Lemma

Betrachte ein anfangs leeres dynamisches Feld s .

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



Dynamisches Feld

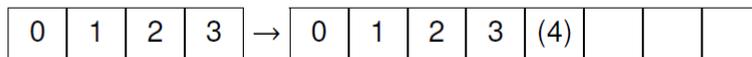
⇒ nur **durchschnittlich konstante** Laufzeit pro Operation

- Kosten teurer Operationen werden mit Kosten billiger Operationen verrechnet.
- Man nennt das dann **amortisierte Kosten** bzw. amortisierte Analyse.
- In diesem Beispiel hätten wir also eine amortisierte Laufzeit von $O(1)$ für die pushBack- und die popBack-Operation.

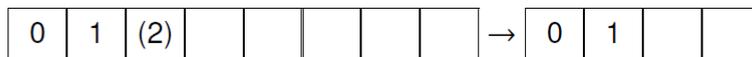


Dynamisches Feld: Analyse

- 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
- ⇒ 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($k \cdot n$): $O(n)$
- Konkret:
 - $\Theta(n)$ Zeugen pro reallocate($k \cdot n$)
 - 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

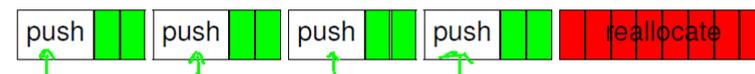
Kontenmethode

- günstige Operationen zahlen Tokens ein
 - pro pushBack 2 Tokens
 - pro popBack 1 Token
- teure Operationen entnehmen Tokens
 - pro reallocate($k \cdot n$) $-n$ 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($k \cdot n$) kostet n Tokens
 - Tokenkosten für reallocate($k \cdot n$): $n-n=0$ Tokens



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

Übersicht

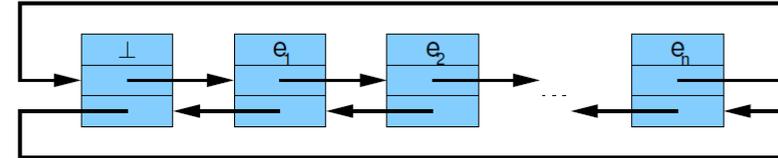
4 Datenstrukturen für Sequenzen

- Felder
- Listen
- Stacks und Queues
- Diskussion: Sortierte Sequenzen

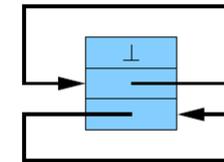
Doppelt verkettete Liste

Einfache Verwaltung:

durch **Dummy**-Element h ohne Inhalt (\perp):



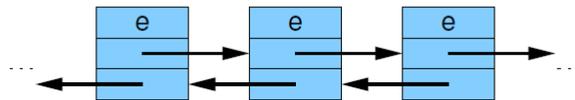
Anfangs:



Doppelt verkettete Liste

type Handle: pointer \rightarrow Item<Elem>;

```
type Item<Elem> {
  Elem e;
  Handle next;
  Handle prev;
}
```



```
class List<Elem> {
  Item<Elem> h; // initialisiert mit ⊥ und Zeigern auf sich selbst
  ... weitere Variablen und Methoden ...
}
```

Invariante:

next \rightarrow prev == prev \rightarrow next == this

Doppelt verkettete Liste

Zentrale statische Methode: splice(Handle a, Handle b, Handle t)

- Bedingung:
 - ▶ $\langle a, \dots, b \rangle$ muss Teilsequenz sein ($a=b$ erlaubt)
 - ▶ b nicht vor a (also Dummy h nicht zwischen a und b)
 - ▶ t nicht in Teilliste $\langle a, \dots, b \rangle$, aber evt. in anderer Liste
- splice entfernt $\langle a, \dots, b \rangle$ aus der Sequenz und fügt sie hinter Item t an

Für

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

liefert splice(a, b, t)

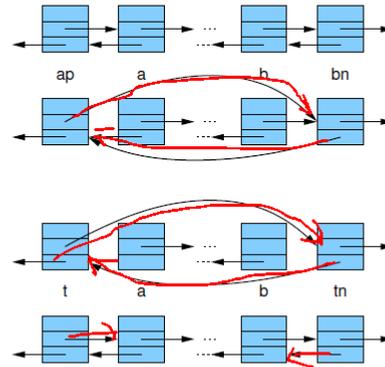
$\langle e_1, \dots, a', b', \dots, t, a, \dots, b, t', \dots, e_n \rangle$

Doppelt verkettete Liste

Methoden

```
(static) splice(Handle a, b, t) {
  // schneide <a, ..., b> heraus
  Handle ap = a->prev;
  Handle bn = b->next;
  ap->next = bn;
  bn->prev = ap;

  // füge <a, ..., b> hinter t ein
  Handle tn = t->next;
  b->next = tn;
  a->prev = t;
  t->next = a;
  tn->prev = b;
}
```



Doppelt verkettete Liste

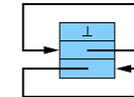
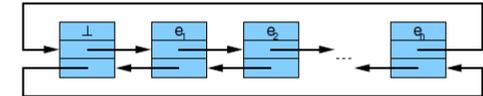
Methoden

```
Handle head() {
  return address of h;
}

boolean isEmpty() {
  return (h.next == head());
}

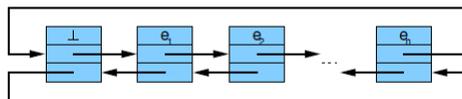
Handle first() {
  return h.next; // evt. → ⊥
}

Handle last() {
  return h.prev; // evt. → ⊥
}
```



Doppelt verkettete Liste

Methoden



```
(static) moveAfter (Handle b, Handle a) {
  splice(b, b, a); // schiebe b hinter a
}
```

```
moveToFront (Handle b) {
  moveAfter(b, head()); // schiebe b ganz nach vorn
}
```

```
moveToBack (Handle b) {
  moveAfter(b, last()); // schiebe b ganz nach hinten
}
```

Doppelt verkettete Liste

Methoden

Löschen und Einfügen von Elementen:
mittels separater Liste **freeList**
⇒ bessere Laufzeit (Speicherallokation teuer)

```
(static) remove(Handle b) {
  moveAfter(b, freeList.head());
}
```

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

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

Doppelt verkettete Liste

Methoden

```
(static) Handle insertAfter(Elem x, Handle a) {
    checkFreeList(); // u.U. Speicher allokieren
    Handle b = freeList.first();
    moveAfter(b, a);
    b → e = x;
    return b;
}
```

```
(static) Handle insertBefore(Elem x, Handle b) {
    return insertAfter(x, b → prev);
}
```

```
pushFront(Elem x) { insertAfter(x, head()); }
```

```
pushBack(Elem x) { insertAfter(x, last()); }
```



Doppelt verkettete Liste

Manipulation ganzer Listen:

Trick: verwende Dummy-Element

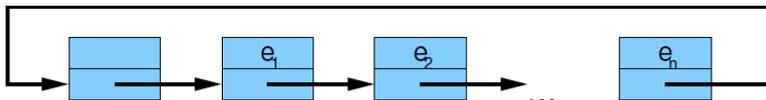
```
Handle findNext(Elem x, Handle from) {
    h.e = x;
    while (from → e != x)
        from = from → next;
    [h.e = ⊥;]
    return from;
}
```



Einfach verkettete Liste

type SHandle: pointer → SItem<Elem>;

```
type SItem<Elem> {
    Elem e;
    SHandle next;
}
```



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



Einfach verkettete Liste

```
(static) splice(SHandle ap, SHandle b, SHandle t) {
    SHandle a = ap → next;
    ap → next = b → next;
    b → next = t → next;
    t → next = a;
}
```

Wir brauchen hier den Vorgänger ap von a!

