Inhaltsverzeichnis

1 Datenstrukturen für Sequenzen

Sequenzen

Felder

Listen

Stacks und Queues

Sortierte Sequenzen

* 1. **Sequenzen**

**1.1.1Sequenz: lineare Struktur**

s = ⟨e0, . . . , en−1⟩

(Gegensatz: verzweigte Struktur in Graphen, fehlende Struktur in Hashtabelle)

**1.1.2Klassische Repräsentationen:**

**• (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

**1.1.3Operationen:**

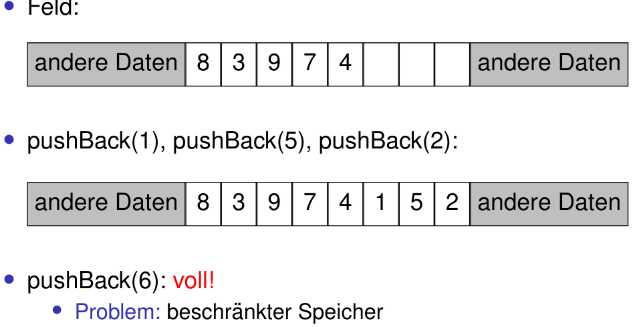
• ⟨e0, . . . , en−1⟩.get(i) = ei

• ⟨e0, . . . , ei−1, ei, . . . , en−1⟩.set(i, e) = ⟨e0, . . . , ei−1, e, . . . , en−1⟩

• ⟨e0, . . . , en−1⟩.pushBack(e) = ⟨e0, . . . , en−1, e⟩

• ⟨e0, . . . , en−1⟩.popBack() = ⟨e0, . . . , en−2⟩

• ⟨e0, . . . , en−1⟩.size() = n

**1.2 Felder(Kettung, Array)**

Problem:

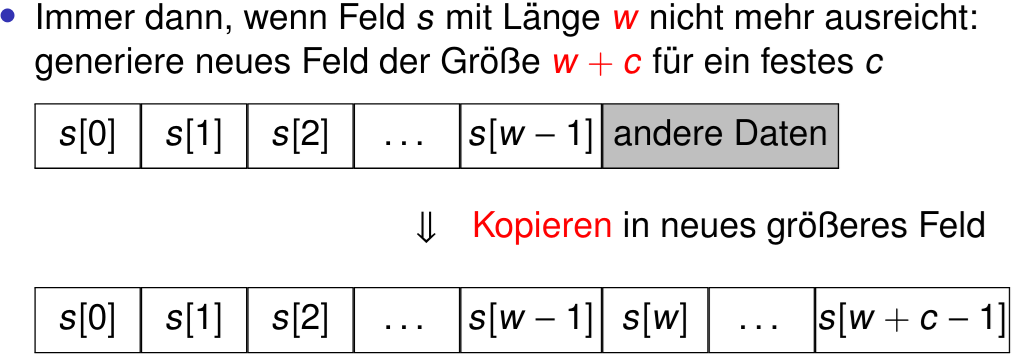
• Beim Anlegen des Felds ist nicht bekannt, wie viele Elemente es enthalten wird

• Nur Anlegen von statischen Feldern mit fester Länge arr.length möglich

**(s = new ElementTyp[arr.length])**

Lösung: Datenstruktur für dynamisches Feld

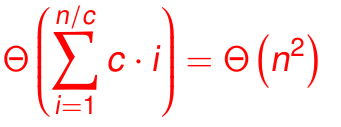
**1.2.1 Dynamische Feld**

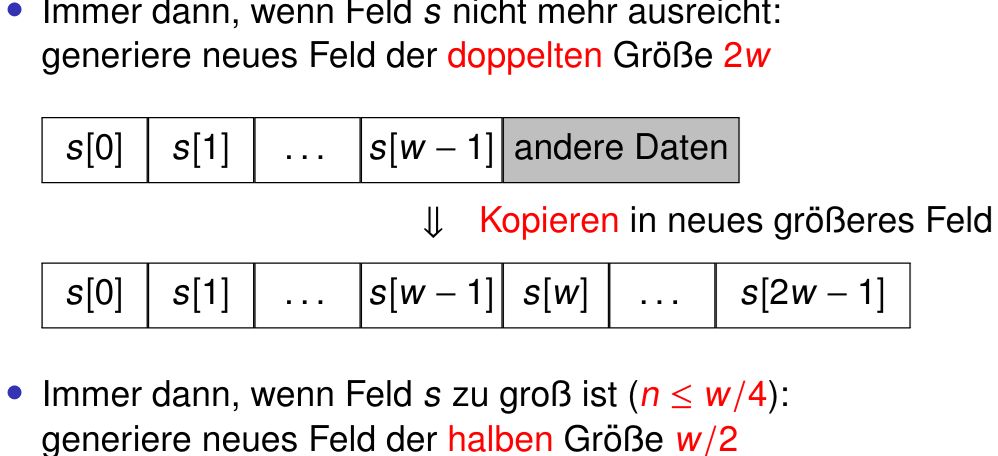
1.2.1.1Erste Idee:

Zeitaufwand für Erweiterung: Θ(w)

Zeitaufwand für n pushBack Operationen:

• Aufwand von Θ(w) nach jeweils c Operationen (wobei w immer größer wird)

• Gesamtaufwand (Start mit leerem Feld):

1.2.1.2Bessere Idee:

1.2.1.3: Implementierung:

Klasse **UArray** mit den Methoden:

• ElementTyp **get**(int i)

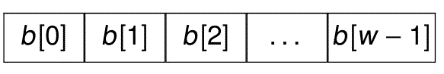
• void **set**(int i, ElementTyp e)

• int **size**()

• void **pushBack**(ElementTyp e)

• void **popBack**()

• void **reallocate**(int new w)

Klasse UArray mit den Elementen:

• β = 2 // Wachstumsfaktor

• α = 4 // max. Speicheroverhead

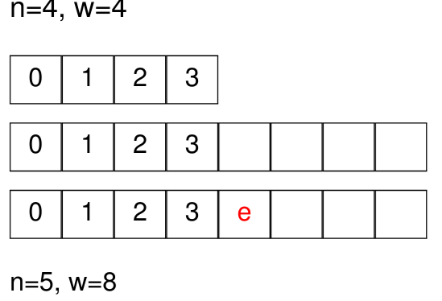
• w = 1 // momentane Feldgröße

• n = 0 // momentane Elementanzahl

• b = new ElementTyp[w] // statisches Feld

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

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

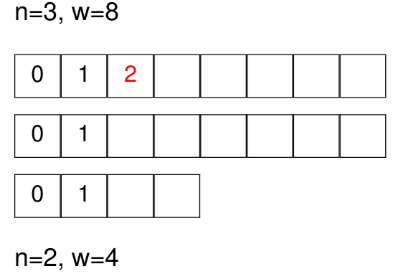
**int size () {return n;}**

**void pushback (ElementTyp e) {**

**if (n==w) {reallocate(β\*w);}**

**b[n]=e;**

**n++;}**

**void popBack () {**

**assert(n>0);**

**n−−;**

**if (α\*n ≤w && n>0) {reallocate(w/β);}**

**}**

**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;**

**}**

1.2.1.4Ü:  
Wieviel Zeit kostet eine Folge von n pushBack/popBack Operationen?

**Erste Idee: WORST CASE**

• einzelne Operation kostet O(n)最坏情况，每次都需要扩容并移动数据

• Schranke kann nicht weiter gesenkt werden, denn reallocate-Aufrufe kosten jeweils Θ(n)

⇒ also Gesamtkosten für n Operationen beschränkt durch n · O(n) = O(n^2) 需要执行n次相同操作

**Zweite Idee: NORMAL CASE**

• 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并不是每次都是满数组

LEMMA:

Betrachte ein anfangs leeres dynamisches Feld s.均摊分析

Jede Folge σ = ⟨σ1, . . . , σn⟩ von pushBack und popBack Operationen auf s kann in Zeit **O(n)** bearbeitet werden. n次操作的复杂度

⇒ **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. 平均下来每次操作的复杂度都是O(1)

1.2.1.5 Formale Verrechnung: Zeugenzuordnung

• reallocate kann eine Vergrößerung oder Verkleinerung sein 重新分配大小可大可小

• reallocate als Vergrößerung von n auf 2n Speicherelemente (erfordert n Kopien):一倍扩容复制原有n个到新的去

es werden die n/2 vorangegangenen pushBack Operationen zugeordnet n个元素的复制成本被之前发生的n/2次（或大约n/2次）pushBack操作所“分摊”。这是因为在动态数组的扩容策略中，每次扩容前通常已经执行了接近当前容量一半的pushBack操作。

• reallocate als Verkleinerung von n auf n/2 Speicherelemente (erfordert n/4 Kopien):

es werden die n/4 vorangegangenen popBack Operationen zugeordnet

⇒ kein pushBack / popBack wird mehr als einmal zugeordnet 不会多次分摊成本

• Idee: verrechne reallocate-Kosten mit pushBack/popBack-Kosten

(ohne reallocate)

• Kosten für pushBack / popBack: O(1)

• Kosten für reallocate(k\*n): O(n) k：扩容倍数

• Konkret:

• Θ(n) Zeugen pro reallocate(k∗n) 每次扩容操作有Θ(n)个“见证者”

• verteile O(n)-Aufwand gleichmäßig auf die Zeugen 平均分摊

• Gesamtaufwand: **O(m**) bei m Operationen

Kontenmethode

• günstige Operationen zahlen Tokens ein

• teure Operationen entnehmen Tokens

• Tokenkonto darf **nie negativ** werden!

Bsp.:

• günstige Operationen zahlen Tokens ein

• pro pushBack 2 Tokens

• pro popBack 1 Token

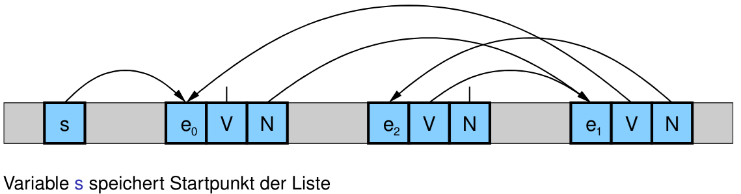
• teure Operationen entnehmen Tokens

• pro reallocate(k∗n) −n Tokens

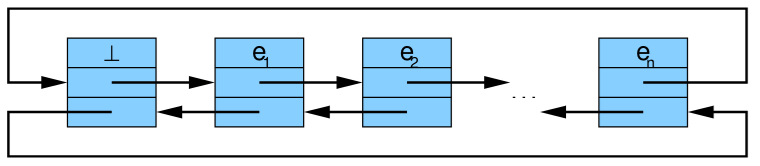
• Tokenkonto darf nie negativ werden!

• Nachweis über Zeugenargument

目标:在执行 push 和 pop 操作时尽可能少地触发 reallocate。这通常是通过使用动态数组的扩展策略来实现的，例如当数组满时，将其大小加倍，而不是仅增加一个元素的空间。这样，可以在多次 push 操作之后才需要进行一次 reallocate，从而减少了昂贵的内存分配和复制操作的次数。

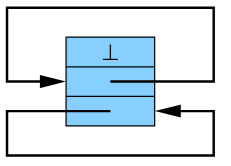
1.3 Listen

1.3.1 Doppelt verkettete Liste



Einfache Verwaltung:

durch Virtuelle-Element h ohne Inhalt (⊥):



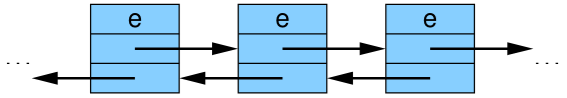
Leere Liste:

1.3.2 Implementierung:

type Handle: Item<Ele>; //声明，Item：名称， Ele：泛型

//结构体定义

type Item<Ele> {

Ele e; //Value

Handle next; //指针，指向下一个

Handle prev; //指针，指向前一个

}//实现

class List<Ele> {

Item<Ele> h; // initialisiert mit ⊥ und Zeigern auf sich selbst //定义了head为起始元素

. . . weitere Variablen und Methoden . . .

}

Invariante: //等价关系，前一个指向的next和后一个的prev都是自己

next.prev == prev.next == this