

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

Übersicht

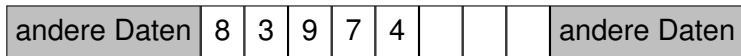
4 Datenstrukturen für Sequenzen

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

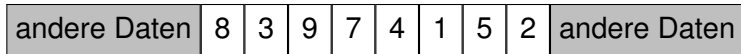
Sequenz als Feld

Problem: beschränkter Speicher

- Feld:



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



- `pushBack(6)`: **voll!**

Sequenz als Feld

Problem:

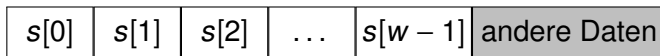
- Beim Anlegen des Felds ist nicht bekannt, wieviele Elemente es enthalten wird
- Nur Anlegen von **statischen** Feldern möglich
(`s = new ElementTyp[w]`)

Lösung: Datenstruktur für **dynamisches** Feld

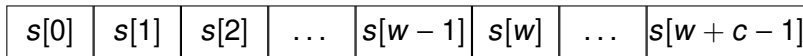
Dynamisches Feld

Erste Idee:

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

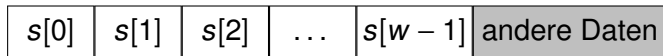


⇓ **Kopieren** in neues größeres Feld

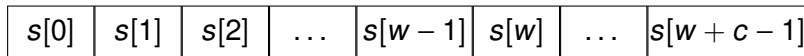


Dynamisches Feld

Zeitaufwand für Erweiterung: $\Theta(w)$



↓ Kopieren in neues größeres Feld



Zeitaufwand für n pushBack Operationen:

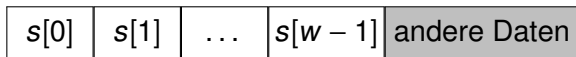
- Aufwand von $\Theta(w)$ nach jeweils c Operationen (wobei w immer größer wird)
- Gesamtaufwand:

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

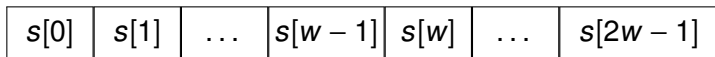
Dynamisches Feld

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 **realloc**(int new_w)

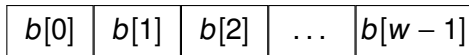
- ElementTyp& [int i]
auch möglich, aber Referenz nur bis zur nächsten Größenänderung des Felds gültig

Dynamisches Feld

Implementierung

Klasse **UArray** mit den Elementen:

- $\beta = 2$ // Wachstumsfaktor
- $\alpha = 4$ // max. Speicheroverhead
- $w = 1$ // momentane Feldgröße
- $n = 0$ // momentane Elementanzahl
- $\mathbf{b} = \text{new ElementTyp}[w]$ // 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( $\beta*n$ );  
    b[n]=e;  
    n++;  
}
```

$n=4, w=4$

0	1	2	3
---	---	---	---

0	1	2	3				
---	---	---	---	--	--	--	--

0	1	2	3	e			
---	---	---	---	---	--	--	--

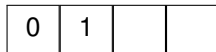
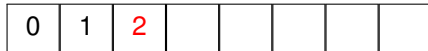
$n=5, w=8$

Dynamisches Feld

Implementierung

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

n=3, w=8



n=2, w=4

Dynamisches Feld

Implementierung

```
void realloc(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 $\Theta(n)$

⇒ 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
- ⇒ 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($k*n$): $O(n)$
- 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($k \cdot n$) $-n$ 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($k*n$) kostet n Tokens
 - ▶ Tokenkosten für reallocate($k*n$): $n-n=0$ Tokens



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

Übersicht

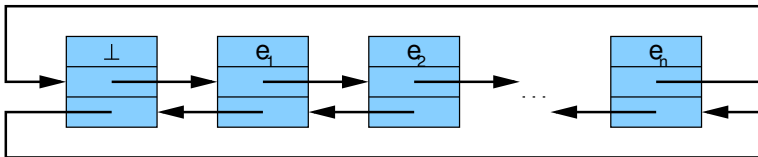
4 Datenstrukturen für Sequenzen

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

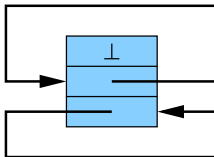
Doppelt verkettete Liste

Einfache Verwaltung:

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



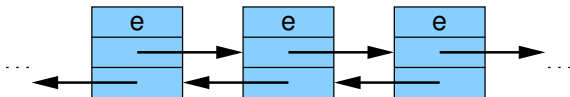
Anfangs:



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  $\perp$  und Zeigern auf sich selbst
  ... weitere Variablen und Methoden ...
}
```

Invariante:

$\text{next} \rightarrow \text{prev} == \text{prev} \rightarrow \text{next} == \text{this}$

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

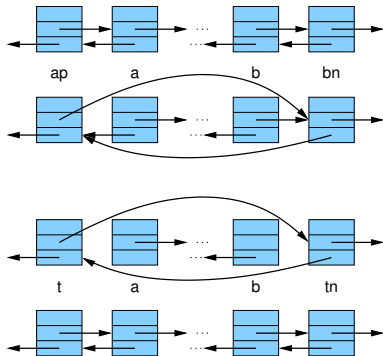
Methoden

```

(static) splice(Handle a, b, t) {
  // schneide  $\langle a, \dots, b \rangle$  heraus
  Handle ap = a→prev;
  Handle bn = b→next;
  ap→next = bn;
  bn→prev = ap;

  // füge  $\langle a, \dots, b \rangle$  hinter t ein
  Handle tn = t→next;
  b→next = tn;
  a→prev = t;
  t→next = a;
  tn→prev = b;
}

```



Doppelt verkettete Liste

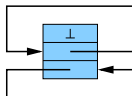
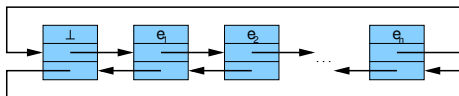
Methoden

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

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

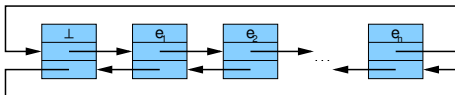
```
Handle first() {
    return h.next;    // evt.  $\rightarrow \perp$ 
}
```

```
Handle last() {
    return h.prev;    // evt.  $\rightarrow \perp$ 
}
```



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:

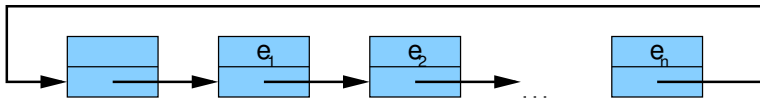
Trick: verwende Dummy-Element

```
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;  
}
```

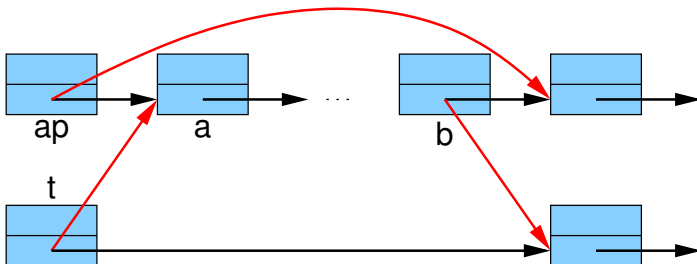


```
class SList<Elem> {  
  SItem<Elem> h;  
  ... weitere Variablen und Methoden ...  
}
```

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!



Einfach verkettete Liste

- findNext sollte evt. auch nicht den nächsten Treffer, sondern dessen **Vorgänger** liefern
(damit man das gefundene SItem auch löschen kann, Suche könnte dementsprechend erst beim Nachfolger des gegebenen SItems starten)
 - auch einige andere Methoden brauchen ein modifiziertes Interface
 - sinnvoll: Pointer zum letzten Item
- ⇒ pushBack in $O(1)$

Übersicht

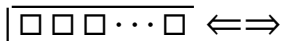
4 Datenstrukturen für Sequenzen

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

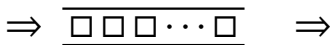
Stacks und Queues

Grundlegende sequenzbasierte Datenstrukturen:

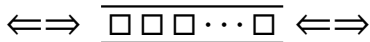
- Stack (Stapel)



- (FIFO-)Queue (Schlange)



- Deque (double-ended queue)



Stacks und Queues

Stack-Methoden:

- pushBack (bzw. push)
- popBack (bzw. pop)
- last (bzw. top)

Queue-Methoden:

- pushBack
- popFront
- first

Stacks und Queues

Warum spezielle Sequenz-Typen betrachten, wenn wir mit der bekannten Datenstruktur für Listen schon alle benötigten Operationen in $O(1)$ haben?

- Programme werden **lesbarer** und **einfacher zu debuggen**, wenn spezialisierte Zugriffsmuster explizit gemacht werden.
- Einfachere Interfaces erlauben eine größere Breite von konkreten Implementationen (hier z.B. **platzsparendere** als Listen).
- Listen sind ungünstig, wenn die Operationen auf dem Sekundärspeicher (Festplatte) ausgeführt werden.

Sequentielle Zugriffsmuster können bei entsprechender Implementation (hier z.B. als Arrays) stark vom **Cache** profitieren.

Stacks und Queues

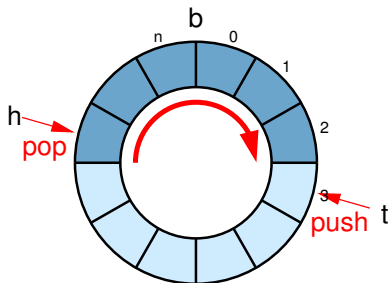
Spezielle Umsetzungen:

- Stacks mit beschränkter Größe \Rightarrow Bounded Arrays
- Stacks mit unbeschränkter Größe \Rightarrow Unbounded Arrays
- oder: Stacks als einfach verkettete Listen
(top of stack = front of list)
- (FIFO-)Queues: einfach verkettete Listen mit Zeiger auf letztes Element (eingefügt wird am Listenende, entnommen am Listenanfang, denn beim Entnehmen muss der Nachfolger bestimmt werden)
- Deques \Rightarrow doppelt verkettete Listen
(einfach verkettete reichen nicht)

Beschränkte Queues

```
class BoundedFIFO<Elem> {
  const int n; // Maximale Anzahl
  Elem[n+1] b;
  int h=0; // erstes Element
  int t=0; // erster freier Eintrag
}
```

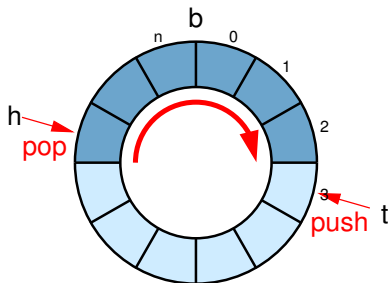
- Queue besteht aus den Feldelementen $h \dots t-1$
- Es bleibt immer mindestens ein Feldelement frei (zur Unterscheidung zwischen voller und leerer Queue)



Beschränkte Queues

Methoden

```
boolean isEmpty() {  
    return (h==t);  
}  
  
Elem first() {  
    assert(!isEmpty());  
    return b[h];  
}  
  
int size() {  
    return (t-h+n+1)% (n+1);  
}
```



Beschränkte Queues

Methoden

```

pushBack(Elem x) {
    assert(size()<n);
    b[t]=x;
    t=(t+1)%(n+1);
}

```

```

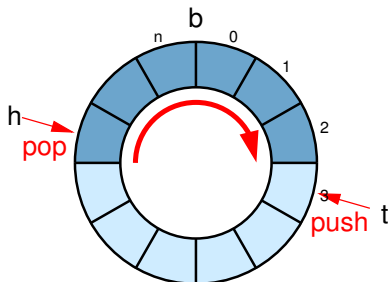
popFront() {
    assert(!isEmpty());
    h=(h+1)%(n+1);
}

```

```

int size() {
    return (t-h+n+1)%(n+1);
}

```



Beschränkte Queues

- Struktur kann auch als **Deque** verwendet werden
- Zirkuläre Arrays erlauben auch den indexierten Zugriff:
Elem Operator [int i] {
 return b[(h+i)%(n+1)];
}
- Bounded Queues / Deques können genauso zu **Unbounded Queues / Deques** erweitert werden wie Bounded Arrays zu Unbounded Arrays

Übersicht

4 Datenstrukturen für Sequenzen

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

Sortierte Sequenz

S: sortierte Sequenz

Jedes Element e identifiziert über $\text{key}(e)$

Operationen:

- $\langle e_1, \dots, e_n \rangle.\text{insert}(e) = \langle e_1, \dots, e_i, e, e_{i+1}, \dots, e_n \rangle$
für das i mit $\text{key}(e_i) < \text{key}(e) < \text{key}(e_{i+1})$
- $\langle e_1, \dots, e_n \rangle.\text{remove}(k) = \langle e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n \rangle$
für das i mit $\text{key}(e_i) = k$
- $\langle e_1, \dots, e_n \rangle.\text{find}(k) = e_i$
für das i mit $\text{key}(e_i) = k$

Sortierte Sequenz

Problem:

Aufrechterhaltung der Sortierung nach jeder Einfügung / Löschung



insert(5)



remove(14)



Sortierte Sequenz

Realisierung als **Liste**

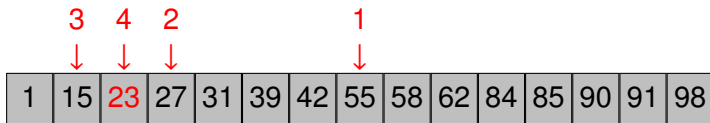
- insert und remove kosten zwar eigentlich nur konstante Zeit, müssen aber wie find zunächst die richtige Position finden
- find auf Sequenz der Länge n kostet $O(n)$ Zeit, damit ebenso insert und remove

Realisierung als **Feld**

- find kann mit binärer Suche in Zeit $O(\log n)$ realisiert werden
- insert und remove kosten $O(n)$ Zeit für das Verschieben der nachfolgenden Elemente

Binäre Suche

find(23):



In einer sortierten Sequenz mit n Elementen kann ein beliebiges Element mit $O(\log n)$ Vergleichen gefunden werden.