Script **generated by TTT**

Title: Täubig: GAD (08.05.2012)

Date: Tue May 08 14:31:12 CEST 2012

Duration: 79:21 min

Pages: 41

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

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

Dynamisches Feld

Implementierung

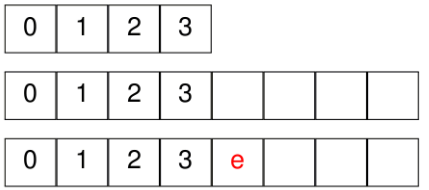
```
ElementTyp get(int i) {
   assert(0≤i<n);
   return b[i];
}

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



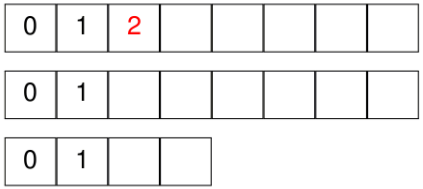
n=5, w=8

Dynamisches Feld

Implementierung

```
void popBack() {
   assert(n>0);
   n−−;
   if (α*n ≤ w ∧ n>0)
      reallocate(β*n);
}
```

n=3, w=8



n=2, w=4

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

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

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.

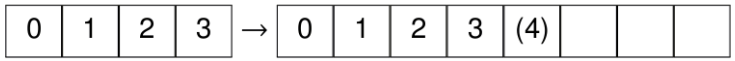
Dynamisches Feld

$\Rightarrow$ 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)

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

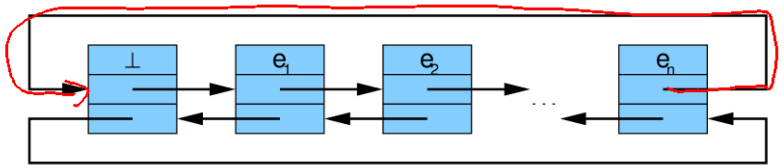
- Idee: verrechne reallocate-Kosten mit pushBack/popBack-Kosten (ohne reallocate)
  - Kosten für pushBack / popBack: $O(1)$
  - Kosten für reallocate(k*n): $O(n)$
- Konkret:
  - $\Theta(n)$ Zeugen pro reallocate(k∗n)
  - verteile $O(n)$-Aufwand gleichmäßig auf die Zeugen
- Gesamtaufwand: $O(m)$ bei m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

Übersicht

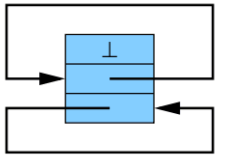
Doppelt verkettete Liste

Einfache Verwaltung:

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



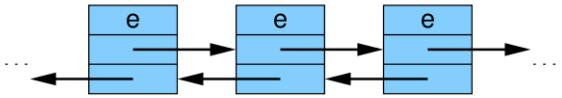
Anfangs:



Doppelt verkettete Liste

type Handle: pointer → 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→prev == prev→next == this

Doppelt verkettete Liste

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

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

Für

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

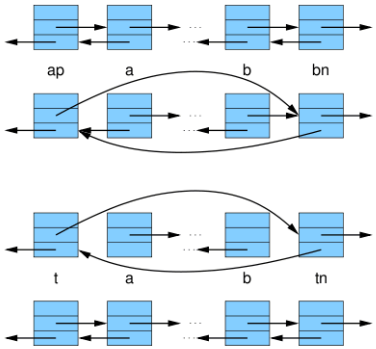
liefert splice(a,b,t)

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

Doppelt verkettete Liste

Methoden

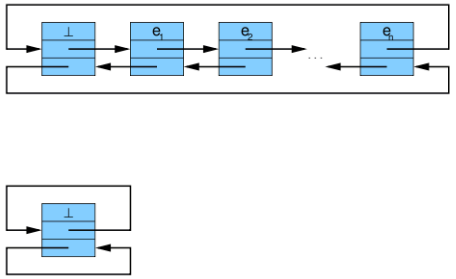
```
(static) splice(Handle a, b, t) {
   // schneide ⟨a,...,b⟩ heraus
   Handle ap = a→prev;
   Handle bn = b→next;
   ap→next = bn;
   bn→prev = ap;

   // 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

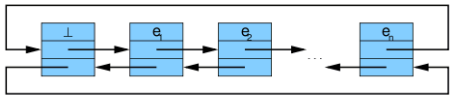
Methoden

```
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

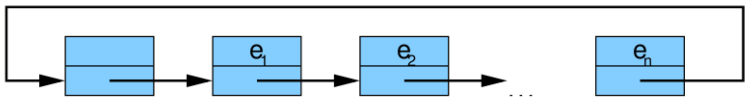
Manipulation ganzer Listen:

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;
}
```

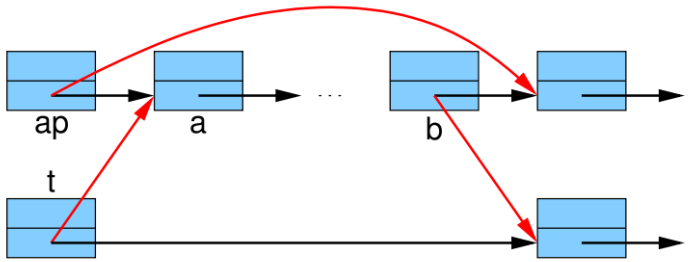



```
class SList<Elem> {
   SItem<Elem> h;
   ... weitere Variablen und Methoden ...
}
```


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!



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

$\Rightarrow$ pushBack in $O(1)$

Übersicht

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

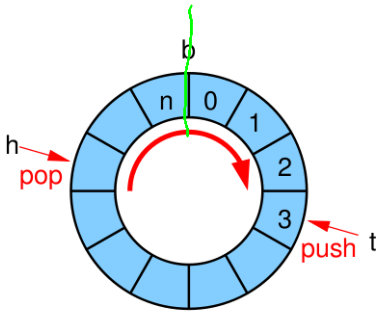
Spezielle Umsetzungen:

- Stacks mit beschränkter Größe ⇒ Bounded Arrays
- Stacks mit unbeschränkter Größe ⇒ 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 ⇒ 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
}
```

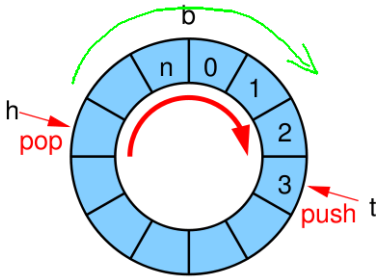
- Queue besteht aus den Feldelementen $h \ldots t-1$
- Es bleibt immer mindestens ein Feldelement frei (zur Unterscheidung zwischen voller und leerer Queue)



Beschränkte Queues

Methoden

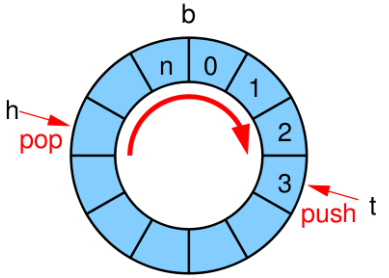
```
boolean isEmpty() {
  return (h==t);
}

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