Script **generated by TTT**

Title: Seidl: GAD (27.04.2016)

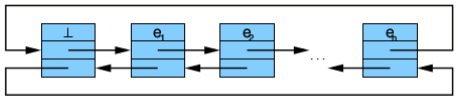
Date: Wed Apr 27 13:21:21 CEST 2016

Duration: 37:01 min

Pages: 16

Doppelt verkettete Liste

Methoden




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

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

void 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)

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

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

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


Doppelt verkettete Liste

Methoden

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

Handle insertBefore(Elem x, Handle b) {
  return insertAfter(x, b.prev);
}

Handle pushFront(Elem x) {   return insertAfter(x, head());   }

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

Doppelt verkettete Liste

Manipulation ganzer Listen:

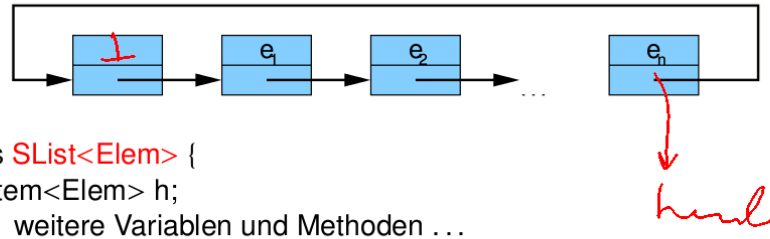
Trick: verwende Dummy-Element

```
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: SItem<Elem>;

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

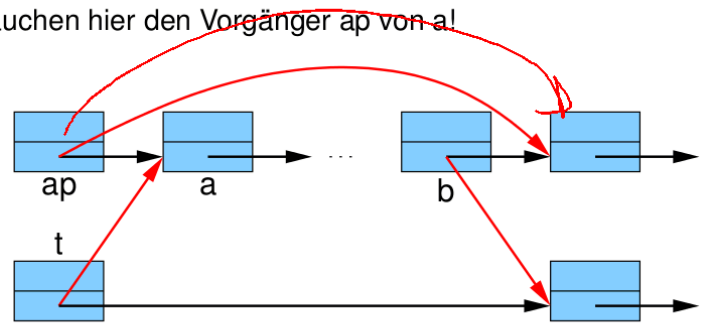



```
class SList<Elem> {
   SItem<Elem> h;
   ... weitere Variablen und Methoden ...
}
```


Einfach verkettete Liste

```
static void 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

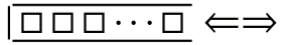
- findNext sollte evt. auch nicht den nächsten Treffer, sondern dessen Vorgänger liefern
  (damit man das gefundene SItem auch löschen kann, Suche könnte dementsprechend erst beim Nachfolger des gegebenen SItems starten)
- auch einige andere Methoden brauchen ein modifiziertes Interface
- sinnvoll: Pointer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

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

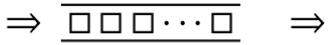
Stacks und Queues

Grundlegende sequenzbasierte Datenstrukturen:

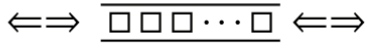
- Stack (Stapel)



- (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 $\mathcal{O}(1)$ 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

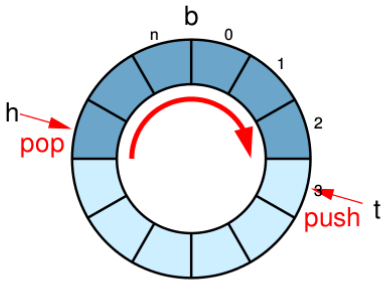
Spezielle Umsetzungen:

- 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)



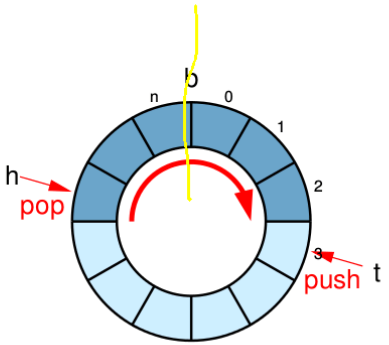
Beschränkte Queues

Methoden

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

Elem first() {
  assert(!isEmpty());
  return b[h];
}

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



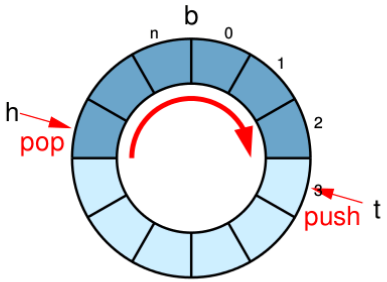
Beschränkte Queues

Methoden

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

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

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



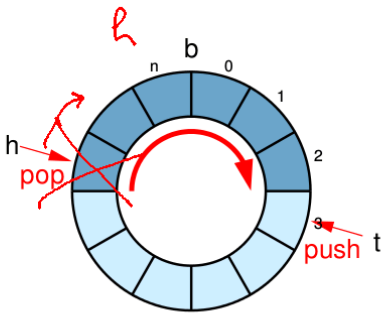
Beschränkte Queues

Methoden

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

void 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