Script generated by TTT

Title: TÃ¤ubig: GAD (07.05.2013)

Date: Tue May 07 14:17:25 CEST 2013

Duration: 88:11 min

Pages: 34

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

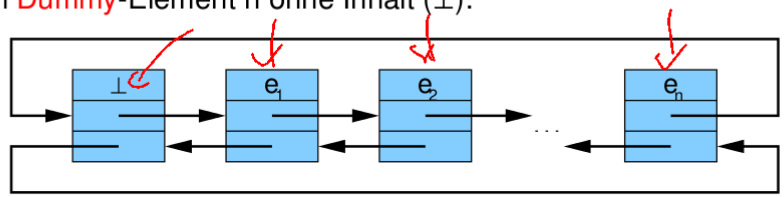
- Gesamtlaufzeit = O(Summe der Tokenlaufzeiten)

Übersicht

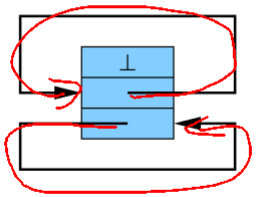
Doppelt verkettete Liste

Einfache Verwaltung:

durch Dummy-Element h ohne Inhalt (⊥):

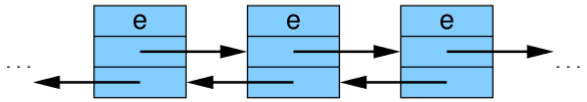


Anfangs:



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

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

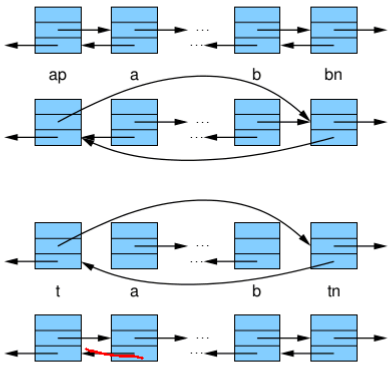
liefert splice(a,b,t)

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

Doppelt verkettete Liste

Methoden

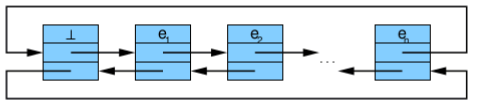
```
(static) splice(Handle a, b, t) {
   // schneide ⟨a, ..., b⟩ heraus
   Handle ap = a→prev;
   Handle bn = b→next;
   ap→next = bn;
   bn→prev = ap;

   // füge ⟨a, ..., b⟩ hinter t ein
   Handle tn = t→next;
   b→next = tn;
   a→prev = t;
   t→next = a;
   tn→prev = b;
}
```

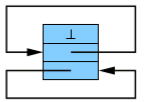


Doppelt verkettete Liste

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. → ⊥
}
```

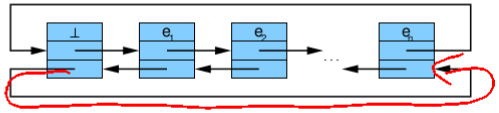




Doppelt verkettete Liste

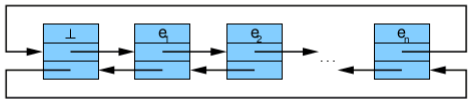
Methoden

```
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

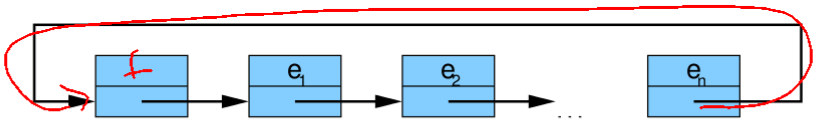
Manipulation ganzer Listen:

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;
}
```

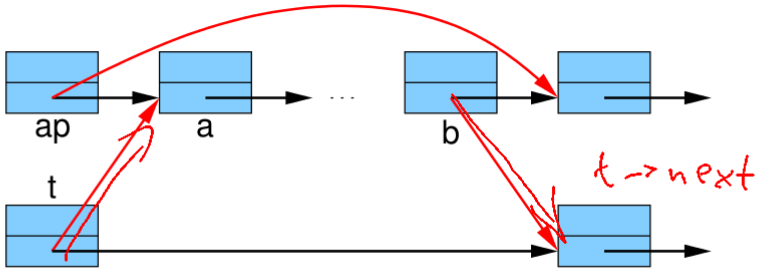



```
class SList<Elem> {
   SItem<Elem> h;
   ... weitere Variablen und Methoden ...
}
```


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

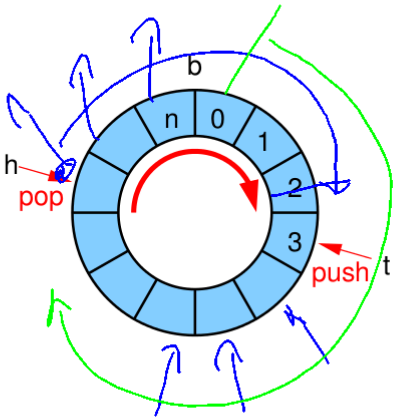
Spezielle Umsetzungen:

- 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
}
```

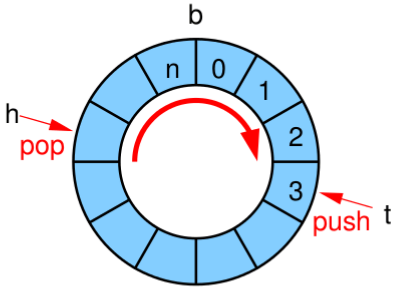
- Queue besteht aus den Feldelementen $h \ldots t-1$
- Es bleibt immer mindestens ein Feldelement frei (zur Unterscheidung zwischen voller und leerer Queue)



Beschränkte Queues

Methoden

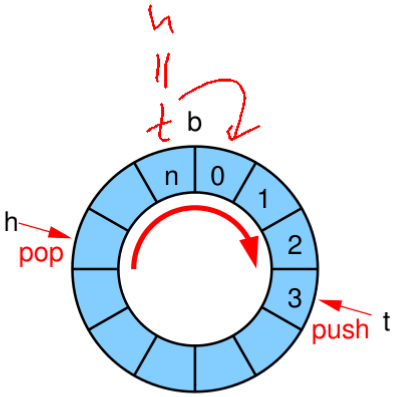
```
boolean isEmpty() {
  return (h==t);
}

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

Sortierte Sequenz

S: sortierte Sequenz

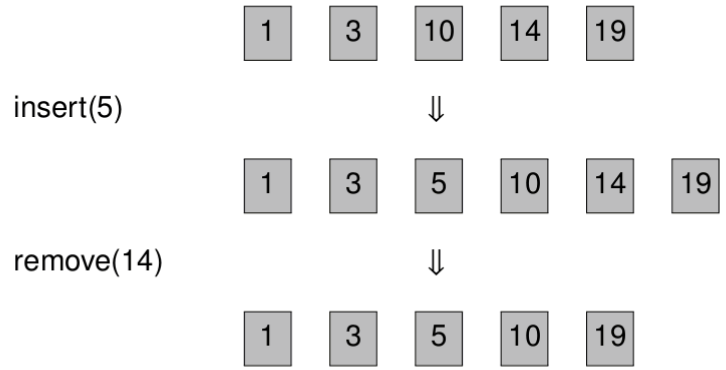
Jedes Element e identifiziert über key(e)

Operationen:

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

Sortierte Sequenz

Problem:
Aufrechterhaltung der Sortierung nach jeder Einfügung / Löschung



Sortierte Sequenz

Realisierung als Liste

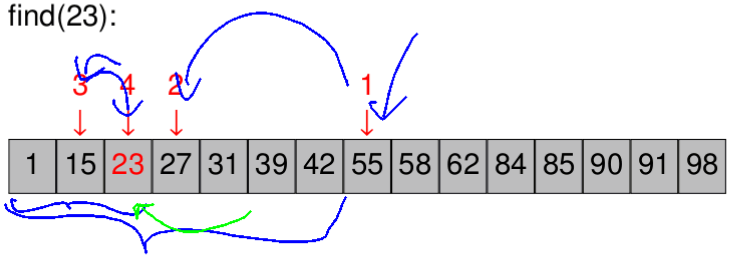
- insert und remove kosten zwar eigentlich nur konstante Zeit, müssen aber wie find zunächst die richtige Position finden
- find auf Sequenz der Länge n kostet $O(n)$ Zeit, damit ebenso insert und remove

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 n-elementigen Sequenz kann ein beliebiges Element mit maximal $2 + \lfloor \log_2 n \rfloor$ Vergleichen gefunden werden.

Übersicht

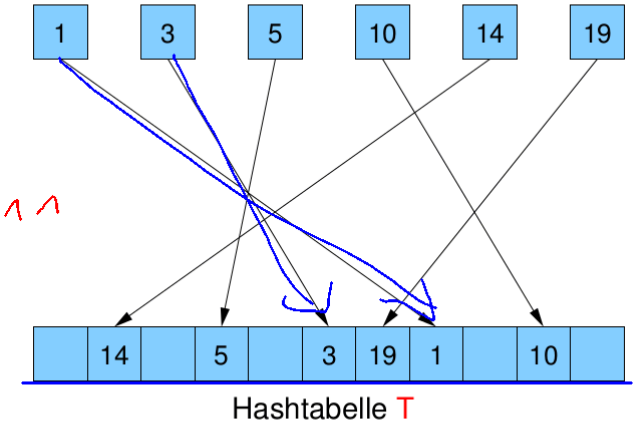
Übersicht

Hashfunktion und Hashtabelle

Hashfunktion h :
Key $\mapsto \{0, \ldots, m-1\}$

$|\text{Key}| = N$

gespeicherte Elemente: $n = 6$



Hashtabelle T

Hashfunktion

Anforderungen:

- schneller Zugriff (Zeiteffizienz)
- platzsparend (Speichereffizienz)
- gute Streuung bzw. Verteilung der Elemente über die ganze Tabelle
- Idealfall: Element e direkt in Tabelleneintrag t[h(key(e))]

$\Rightarrow$ find, insert und remove in konstanter Zeit
(genauer: plus Zeit für Berechnung der Hashfunktion)

Hashing

Annahme: perfekte Streuung

```
insert(Elem e) {
  T[h(key(e))] = e;
}

remove(Key k) {
  T[h(k)] = null;
}

Elem find(Key k) {
  return T[h(k)];
}
```

statisches Wörterbuch: nur find
dynamisches Wörterbuch: insert, remove und find

Übersicht