Script **generated by TTT**

Title: Seidl: GAD (12.05.2015)

Date: Tue May 12 13:45:43 CEST 2015

Duration: 147:11 min

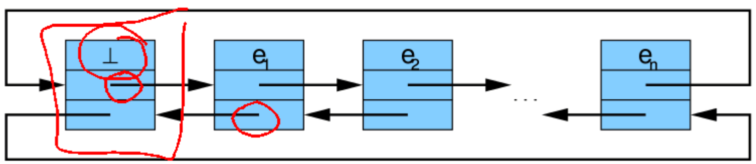
Pages: 75

Übersicht

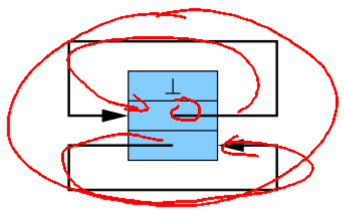
Doppelt verkettete Liste

Einfache Verwaltung:

durch Dummy-Element h ohne Inhalt (⊥):

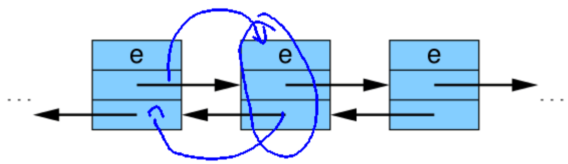


Anfangs:



Doppelt verkettete Liste

```
type Handle: 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:
  - $\langle a, \dots, b \rangle$ 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

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

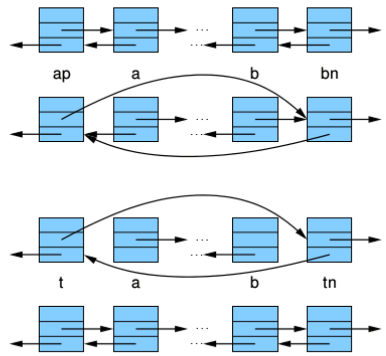
liefert splice(a,b,t)

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

Doppelt verkettete Liste

Methoden

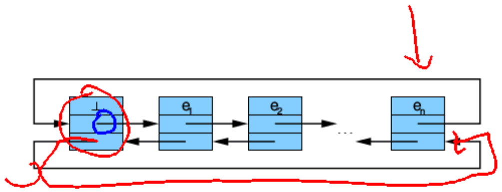
```
(static) splice(Handle a, b, t) {
    // schneide ⟨a, ..., b⟩ heraus
    Handle ap = a.prev;
    Handle bn = b.next;
    ap.next = bn;
    bn.prev = ap;

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

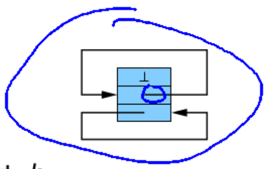


Doppelt verkettete Liste

Methoden

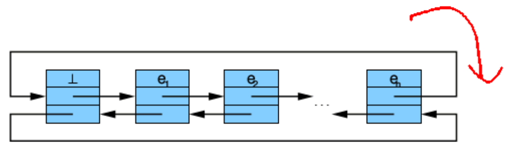
```
Handle head() {
    return h;
}

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

Handle first() {
    return h.next;        // evt. h
}

Handle last() {
    return h.prev;        // evt. h
}
```





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());
}
```

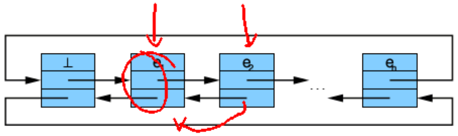

Doppelt verkettete Liste

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

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
}
```

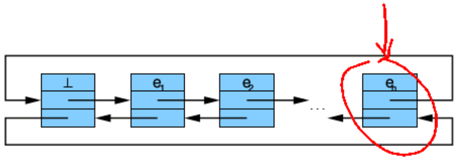

Doppelt verkettete Liste

Methoden

```
(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

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

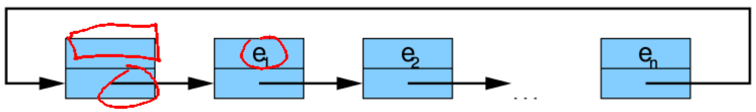
Manipulation ganzer Listen:

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;
}
```

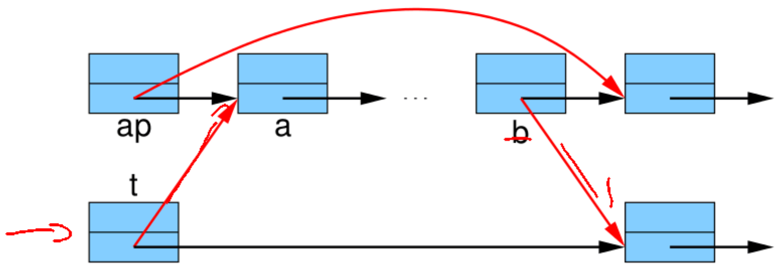



```
class SList<Elem> {
   SItem<Elem> h;
   ... weitere Variablen und Methoden ...
}
```


Einfach verkettete Liste

```
(static) 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

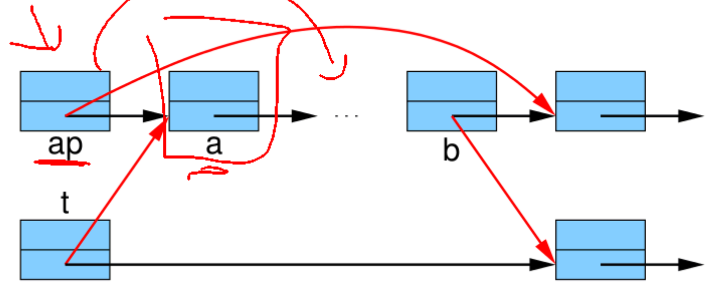
- findNext sollte evt. auch nicht den nächsten Treffer, sondern dessen Vorgänger liefern
  (damit man das gefundene SItem auch löschen kann, Suche könnte dementsprechend erst beim Nachfolger des gegebenen SItems starten)
- auch einige andere Methoden brauchen ein modifiziertes Interface
- sinnvoll: Pointer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

⇒ pushBack in $\mathcal{O}(1)$

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!



Datenstrukturen für Sequenzen | Listen

Einfach verkettete Liste

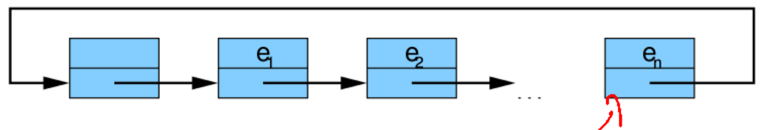
- findNext sollte evt. auch nicht den nächsten Treffer, sondern dessen Vorgänger liefern
  (damit man das gefundene SItem auch löschen kann, Suche könnte dementsprechend erst beim Nachfolger des gegebenen SItems starten)
- auch einige andere Methoden brauchen ein modifiziertes Interface
- sinnvoll: Pointer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

⇒ pushBack in $\mathcal{O}(1)$

H. Seidl (TUM) | GAD | SS'15 | 149

Einfach verkettete Liste

```
type SHandle: SItem<Elem>;

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




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

Übersicht

Stacks und Queues

Grundlegende sequenzbasierte Datenstrukturen:

- Stack (Stapel)

  LIFO

  $|\square\,\square\,\square\cdots\square \Leftarrow\!\Rightarrow$

- (FIFO-)Queue (Schlange)

  $\Rightarrow \square\,\square\,\square\cdots\square \Rightarrow$

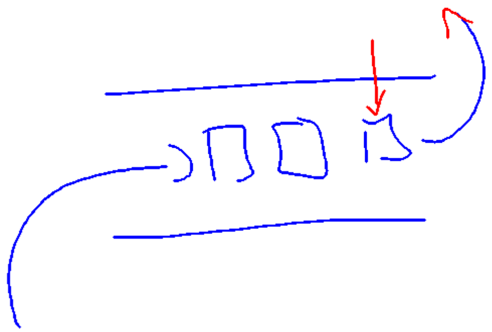
- Deque (double-ended queue)

  $\Leftarrow\!\Rightarrow \square\,\square\,\square\cdots\square \Leftarrow\!\Rightarrow$

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.

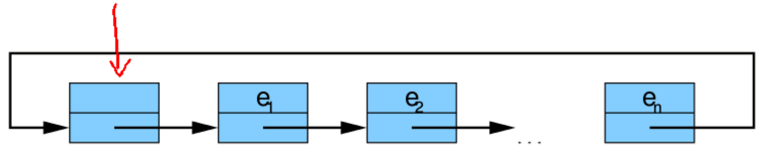
Stacks und Queues

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)

Einfach verkettete Liste

```
type SHandle: SItem<Elem>;

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




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


Datenstrukturen für Sequenzen | Stacks und Queues

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)

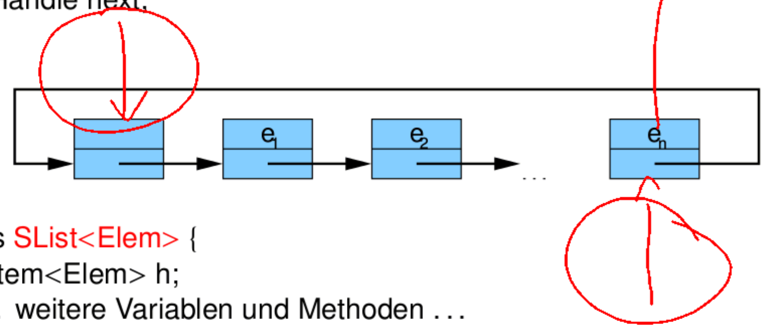
H. Seidl (TUM) | GAD | SS'15 | 154

Datenstrukturen für Sequenzen | Listen

Einfach verkettete Liste

type SHandle: SItem<Elem>;

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

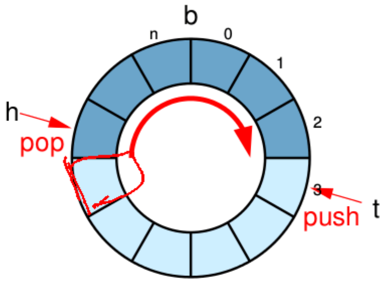


class SList<Elem> {
   SItem<Elem> h;
   ... weitere Variablen und Methoden ...
}

H. Seidl (TUM) | GAD | SS'15 | 147

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
}
```

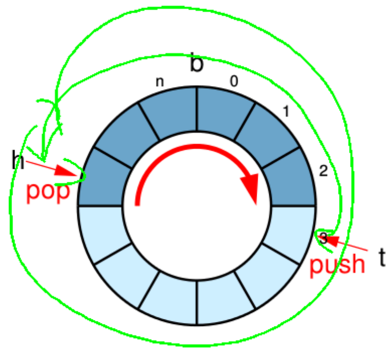


- Queue besteht aus den Feldelementen h...t−1
- Es bleibt immer mindestens ein Feldelement frei (zur Unterscheidung zwischen voller und leerer Queue)

Beschränkte Queues

Methoden

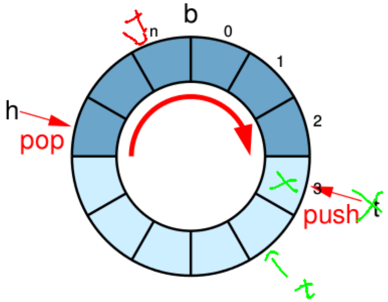
```
boolean isEmpty() {
  return (h==t);
}

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:

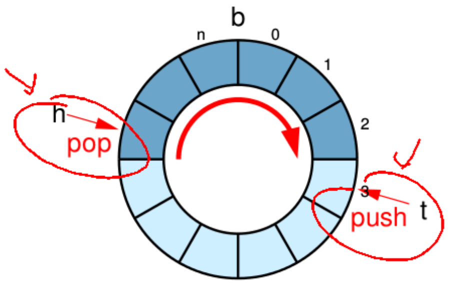
```
Elem Operator [int i] {
  return b[(h+i)%(n+1)];
}
```

- Bounded Queues / Deques können genauso zu Unbounded Queues / Deques erweitert werden wie Bounded Arrays zu Unbounded Arrays

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)];
}
```

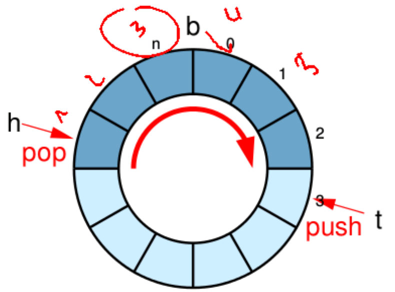
- Bounded Queues / Deques können genauso zu Unbounded Queues / Deques erweitert werden wie Bounded Arrays zu Unbounded Arrays

Datenstrukturen für Sequenzen | Stacks und Queues

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);
}
```

Datenstrukturen für Sequenzen | Stacks und Queues

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

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

Sortierte Sequenz

Problem:
Aufrechterhaltung der Sortierung nach jeder Einfügung / Löschung

	1	3	10	14	19	
insert(5)			⇓			
	1	3	5	10	14	19
remove(14)			⇓			
	1	3	5	10	19	

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):

	3 ↓	4 ↓	2 ↓				1 ↓							
1	15	23	27	31	39	42	55	58	62	84	85	90	91	98

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

Übersicht

5 Hashing
- Hashtabellen
- Hashing with Chaining
- Universelles Hashing
- Hashing with Linear Probing
- Anpassung der Tabellengröße
- Perfektes Hashing
- Diskussion / Alternativen

Assoziative Arrays / Wörterbücher

- Assoziatives Array / Wörterbuch (dictionary) S:
  speichert eine Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel key(e)

Operationen:

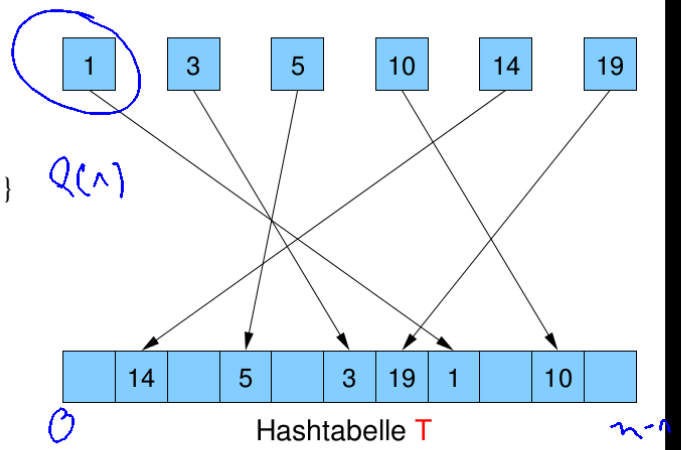
- S.insert(Elem e): $S := S \cup \{e\}$
- S.remove(Key k): $S := S \setminus \{e\}$,
  wobei e das Element mit key(e) $= k$ ist
- S.find(Key k):
  gibt das Element $e \in S$ mit key(e) $= k$ zurück, falls es existiert,
  sonst $\perp$ (entspricht Array-Indexoperator [ ], daher der Name)

Hashfunktion und Hashtabelle

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

$|\text{Key}| = N$

gespeicherte
Elemente: n



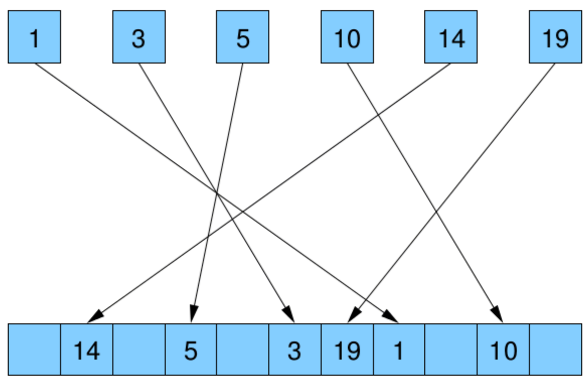
Hashtabelle T

Hashfunktion und Hashtabelle

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

$|\text{Key}| = N$

gespeicherte
Elemente: n



Hashtabelle T

Hashfunktion

Anforderungen:

- schneller Zugriff (Zeiteffizienz)
- platzsparend (Speichereffizienz)
  (z.B. surjektive Abbildung möglicher Schlüssel auf die Adressen)
- gute Streuung bzw. Verteilung der Elemente
  über die ganze Tabelle
- Idealfall: Element e direkt in Tabelleneintrag t[h(key(e))]

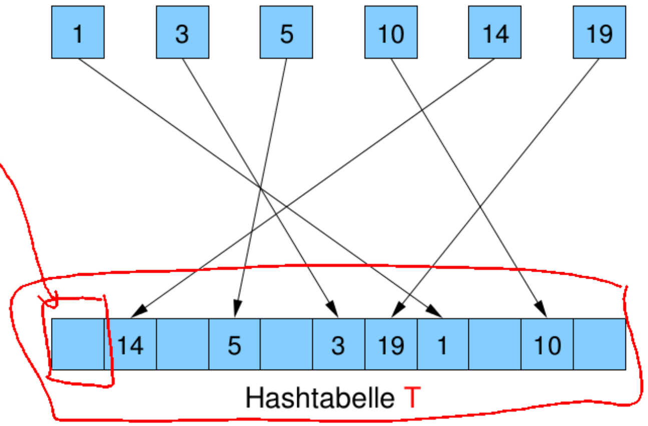
⇒ find, insert und remove in konstanter Zeit
(genauer: plus Zeit für Berechnung der Hashfunktion)

Hashfunktion und Hashtabelle

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

$|\text{Key}| = N$

gespeicherte
Elemente: n



Hashing Hashtabellen

Hashfunktion

Anforderungen:

schneller Zugriff (Zeiteffizienz)
platzsparend (Speichereffizienz)
(z.B. surjektive Abbildung möglicher Schlüssel auf die Adressen)
gute Streuung bzw. Verteilung der Elemente
über die ganze Tabelle
Idealfall: Element e direkt in Tabelleneintrag t[h(key(e))]
⇒ find, insert und remove in konstanter Zeit
(genauer: plus Zeit für Berechnung der Hashfunktion)

H. Seidl (TUM) GAD SS'15 168

Hashing Hashtabellen

Hashfunktion

Anforderungen:

schneller Zugriff (Zeiteffizienz)
platzsparend (Speichereffizienz)
(z.B. surjektive Abbildung möglicher Schlüssel auf die Adressen)
gute Streuung bzw. Verteilung der Elemente
über die ganze Tabelle
Idealfall: Element e direkt in Tabelleneintrag t[h(key(e))]
⇒ find, insert und remove in konstanter Zeit
(genauer: plus Zeit für Berechnung der Hashfunktion)

H. Seidl (TUM) GAD SS'15 168

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

Kollisionen

In der Praxis:

- perfekte Zuordnung zwischen den gespeicherten Schlüsseln und den Adressen der Tabelle nur bei statischem Array möglich
- leere Tabelleneinträge
- Schlüssel mit gleicher Adresse (Kollisionen)

Wie wahrscheinlich ist eine Kollision?

- Geburtstagsparadoxon: In einer Menge von 23 zufällig ausgewählten Personen gibt es mit Wahrscheinlichkeit > 50% zwei Leute, die am gleichen Tag Geburtstag feiern.
- Bei zufälliger Abbildung von 23 Schlüsseln auf die Adressen einer Hashtabelle der Größe 365 gibt es mit Wahrscheinlichkeit > 50% eine Kollision.

Wahrscheinlichkeit von Kollisionen

- Hilfsmittel: $\forall x \in \mathbb{R}: \quad 1 + x \leq \sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x$

$\Rightarrow$ $\forall x \in \mathbb{R}: \quad \ln(1+x) \leq x \quad$ (da $\ln(x)$ monoton wachsend ist)

- Pr[keine Kollision beim i-ten Schlüssel] $= \frac{m-(i-1)}{m}$ für $i \in [1 \ldots n]$

$$\begin{aligned}\Pr[\text{keine Kollision}] &= \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1}\left(1 - \frac{i}{m}\right) \\ &= e^{\left[\sum_{i=0}^{n-1} \ln\left(1-\frac{i}{m}\right)\right]} \leq e^{\left[\sum_{i=0}^{n-1}\left(-\frac{i}{m}\right)\right]} = e^{\left[-\frac{n(n-1)}{2m}\right]}\end{aligned}$$

(da e^x monoton wachsend ist)

$\Rightarrow$ Bei gleichverteilt zufälliger Hashposition für jeden Schlüssel tritt für $n \in \omega(\sqrt{m})$ mit Wahrscheinlichkeit $1 - o(1)$ mindestens eine Kollision auf.

Wahrscheinlichkeit von Kollisionen

$\frac{1}{e^{\frac{[illegible]}{2m}}}$

- Hilfsmittel: $\forall x \in \mathbb{R}: \quad 1 + x \leq \sum_{i=0}^{\infty} \frac{x^i}{i!} = e^x$

$\Rightarrow$ $\forall x \in \mathbb{R}: \quad \ln(1+x) \leq x \quad$ (da $\ln(x)$ monoton wachsend ist)

- Pr[keine Kollision beim i-ten Schlüssel] $= \frac{m-(i-1)}{m}$ für $i \in [1 \ldots n]$

$$\begin{aligned}\Pr[\text{keine Kollision}] &= \prod_{i=1}^{n} \frac{m-(i-1)}{m} = \prod_{i=0}^{n-1}\left(1 - \frac{i}{m}\right) \\ &= e^{\left[\sum_{i=0}^{n-1} \ln\left(1-\frac{i}{m}\right)\right]} \leq e^{\left[\sum_{i=0}^{n-1}\left(-\frac{i}{m}\right)\right]} = e^{\left[-\frac{n(n-1)}{2m}\right]}\end{aligned}$$

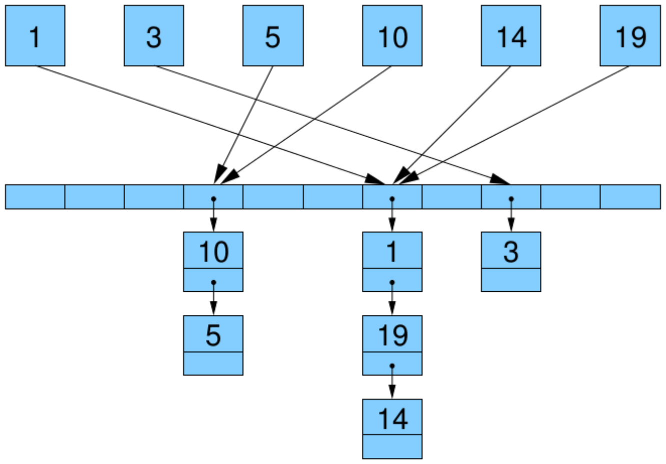
(da e^x monoton wachsend ist)

$\Rightarrow$ Bei gleichverteilt zufälliger Hashposition für jeden Schlüssel tritt für $n \in \omega(\sqrt{m})$ mit Wahrscheinlichkeit $1 - o(1)$ mindestens eine Kollision auf.

Dynamisches Wörterbuch

Hashing with Chaining:

Feld von Listen oder Zeigern



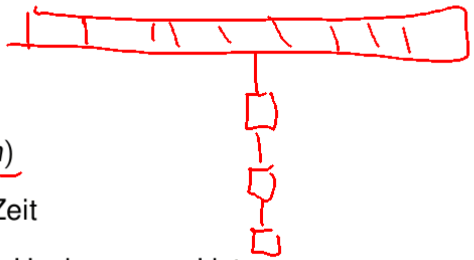
unsortierte verkettete Listen
(Ziel: Listen möglichst kurz)

Dynamisches Wörterbuch

Hashing with Chaining:

```
List<Elem>[m] T;

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

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

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


Hashing with Chaining



- Platzverbrauch: $O(n+m)$
- insert benötigt konstante Zeit
- remove und find müssen u.U. eine ganze Liste scannen
- im worst case sind alle Elemente in dieser Liste

$\Rightarrow$ im worst case ist Hashing with chaining nicht besser als eine normale Liste

Hashing with Chaining

Gibt es Hashfunktionen, die garantieren, dass alle Listen kurz sind?

- nein, für jede Hashfunktion gibt es eine Adresse, der mindestens N/m mögliche Schlüssel zugeordnet sind (erweitertes Schubfachprinzip / pigeonhole principle)
- Meistens ist $n < N/m$ (weil N riesig ist).
- In diesem Fall kann die Suche zum Scan aller Elemente entarten.

$\Rightarrow$ Auswege

- Average-case-Analyse
- Randomisierung
- Änderung des Algorithmus (z.B. Hashfunktion abhängig von aktuellen Schlüsseln)

Hashing with Chaining

$\ell(0) \mid \ell(1) \mid \ell(2)$

Betrachte als Hashfunktionsmenge die Menge aller Funktionen, die die Schlüsselmenge (mit Kardinalität N) auf die Zahlen $0, \ldots, m-1$ abbilden.

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $O(1 + n/m)$.

Unrealistisch: es gibt m^N solche Funktionen und man braucht $\log_2(m^N) = N \log_2 m$ Bits, um eine Funktion zu spezifizieren.

$\Rightarrow$ widerspricht dem Ziel, den Speicherverbrauch von N auf n zu senken!

Hashing with Chaining

Beweis.

- Betrachte feste Position $i = h(k)$ bei remove(k) oder find(k)
- Laufzeit ist Konstante plus Zeit für Scan der Liste also $O(1 + \mathbb{E}[X])$, wobei X Zufallsvariable für Länge von $T[i]$

Hashing with Chaining

Beweis.

- Betrachte feste Position $i = h(k)$ bei remove(k) oder find(k)
- Laufzeit ist Konstante plus Zeit für Scan der Liste also $O(1 + \mathbb{E}[X])$, wobei X Zufallsvariable für Länge von $T[i]$
- Zufallsvariable $X_e \in \{0,1\}$ für jedes $e \in S$
- $X_e = 1 \iff h(\text{key}(e)) = i$
- Listenlänge $X = \sum_{e \in S} X_e$

Hashing with Chaining

Beweis.

- Betrachte feste Position $i = h(k)$ bei remove(k) oder find(k)
- Laufzeit ist Konstante plus Zeit für Scan der Liste also $O(1 + \mathbb{E}[X])$, wobei X Zufallsvariable für Länge von $T[i]$
- Zufallsvariable $X_e \in \{0,1\}$ für jedes $e \in S$
- $X_e = 1 \iff h(\text{key}(e)) = i$
- Listenlänge $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge $\mathbb{E}[X]$

$$= \mathbb{E}\left[\sum_{e \in S} X_e\right] = \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} \left(0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1]\right)$$

$$= \sum_{e \in S} \Pr[X_e = 1] = \sum_{e \in S} 1/m = n/m$$

$\square$

c-universelle Familien von Hashfunktionen

Wie konstruiert man zufällige Hashfunktionen?

Definition

Sei c eine positive Konstante.

Eine Familie H von Hashfunktionen auf $\{0, \dots, m-1\}$ heißt c-universell, falls für jedes Paar $x \neq y$ von Schlüsseln gilt, dass

$$\left|\{h \in H : \; h(x) = h(y)\}\right| \leq \frac{c}{m}|H|.$$

c-universelle Familien von Hashfunktionen

Wie konstruiert man zufällige Hashfunktionen?

Definition

Sei c eine positive Konstante.

Eine Familie H von Hashfunktionen auf $\{0, \dots, m-1\}$ heißt c-universell, falls für jedes Paar $x \neq y$ von Schlüsseln gilt, dass

$$\left|\{h \in H : \; h(x) = h(y)\}\right| \leq \frac{c}{m}|H|.$$

D.h. bei zufälliger Auswahl der Hashfunktion $h \in H$ gilt

$$\forall \{x, y\}_{(x \neq y)} : \quad \Pr[h(x) = h(y)] \leq \frac{c}{m}$$

1-universelle Familien nennt man universell.

Hashing | Universelles Hashing

c-universelle Familien von Hashfunktionen

Wie konstruiert man zufällige Hashfunktionen?

Definition

Sei c eine positive Konstante.

Eine Familie H von Hashfunktionen auf $\{0, \dots, m-1\}$ heißt c-universell, falls für jedes Paar $x \neq y$ von Schlüsseln gilt, dass

$$\left|\{h \in H : \; h(x) = h(y)\}\right| \leq \frac{c}{m}|H|.$$

D.h. bei zufälliger Auswahl der Hashfunktion $h \in H$ gilt

$$\forall \{x, y\}_{(x \neq y)} : \quad \Pr[h(x) = h(y)] \leq \frac{c}{m}$$

1-universelle Familien nennt man universell.

H. Seidl (TUM) | GAD | SS'15 | 180

c-Universal Hashing with Chaining

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion h aus einer c-universellen Familie gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $O(1 + c \cdot n/m)$.

c-Universal Hashing with Chaining

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion h aus einer c-universellen Familie gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $O(1 + c \cdot n/m)$.

Beweis.

- Betrachte festen Schlüssel k
- Zugriffszeit X ist $O(1 + \text{Länge der Liste } T[h(k)])$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$ zeigt an, ob e auf die gleiche Position wie k gehasht wird

c-Universal Hashing with Chaining

Beweis.

- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = h(k)$
- Listenlänge $X = \sum_{e \in S} X_e$
- Erwartete Listenlänge

$$
\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}\left[\sum_{e \in S} X_e\right] \\
&= \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} 0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1] \\
&= \sum_{e \in S} \Pr[X_e = 1] \leq \sum_{e \in S} c/m = n \cdot c/m
\end{aligned}
$$

□

Hashing | Universelles Hashing

c-Universal Hashing with Chaining

Satz

Falls n Elemente in einer Hashtabelle der Größe m mittels einer zufälligen Hashfunktion h aus einer c-universellen Familie gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $O(1 + c \cdot n/m)$.

Beweis.

- Betrachte festen Schlüssel k
- Zugriffszeit X ist $O(1 + \text{Länge der Liste } T[h(k)])$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$ zeigt an, ob e auf die gleiche Position wie k gehasht wird

H. Seidl (TUM) GAD SS'15 181

Beispiele für c-universelles Hashing

$5 \cdot \frac{1}{5} = 1$

Einfache c-universelle Hashfunktionen?

Annahme: Schlüssel sind Bitstrings einer bestimmten Länge

Wähle als Tabellengröße m eine Primzahl

$\Rightarrow$ dann ist der Restklassenring modulo m (also $\mathbb{Z}_m$) ein Körper, d.h. es gibt zu jedem Element außer für die Null genau ein Inverses bzgl. Multiplikation $\mathbb{Z}_5$ $2 \cdot 3 \equiv 1 \mod 5$

- Sei $w = \lfloor \log_2 m \rfloor$.
- unterteile die Bitstrings der Schlüssel in Teile zu je w Bits
- Anzahl der Teile sei k
- interpretiere jeden Teil als Zahl aus dem Intervall $[0, \ldots, 2^w - 1]$
- interpretiere Schlüssel x als k-Tupel solcher Zahlen:

$$\mathbf{x} = (x_1, \ldots, x_k)$$

Familie für universelles Hashing

Definiere für jeden Vektor

$$\mathbf{a} = (a_1, \ldots, a_k) \in \{0, \ldots, m-1\}^k$$

mittels Skalarprodukt

$$\mathbf{a} \cdot \mathbf{x} = \sum_{i=1}^{k} a_i x_i$$

eine Hashfunktion von der Schlüsselmenge in die Menge der Zahlen $\{0, \ldots, m-1\}$

$$h_{\mathbf{a}}(\mathbf{x}) = \mathbf{a} \cdot \mathbf{x} \bmod m$$

Familie für universelles Hashing

Satz

Wenn m eine Primzahl ist, dann ist

$$H = \left\{ h_{\mathbf{a}} : \mathbf{a} \in \{0, \ldots, m-1\}^k \right\}$$

eine [1-]universelle Familie von Hashfunktionen.

Oder anders:
das Skalarprodukt zwischen einer Tupeldarstellung des Schlüssels und einem Zufallsvektor modulo m definiert eine gute Hashfunktion.

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$

$\Rightarrow$ Schlüssel unterteilt in $k = 4$ Teile mit $w = \lfloor \log_2 m \rfloor = 8$ Bits

- Schlüssel sind also 4-Tupel von Integers aus dem Intervall $[0, 2^8 - 1] = \{0, \ldots, 255\}$, z.B. $\mathbf{x} = (11, 7, 4, 3)$

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$

$\Rightarrow$ Schlüssel unterteilt in $k = 4$ Teile mit $w = \lfloor \log_2 m \rfloor = 8$ Bits

- Schlüssel sind also 4-Tupel von Integers aus dem Intervall $[0, 2^8 - 1] = \{0, \ldots, 255\}$, z.B. $\mathbf{x} = (11, 7, 4, 3)$
- Die Hashfunktion wird auch durch ein 4-Tupel von Integers, aber aus dem Intervall $[0, 269 - 1] = \{0, \ldots, 268\}$, spezifiziert z.B. $\mathbf{a} = (2, 4, 261, 16)$

Hashing Universelles Hashing

Familie für universelles Hashing

Beispiel

- 32-Bit-Schlüssel, Hashtabellengröße $m = 269$

$\Rightarrow$ Schlüssel unterteilt in $k = 4$ Teile mit $w = \lfloor \log_2 m \rfloor = 8$ Bits

- Schlüssel sind also 4-Tupel von Integers aus dem Intervall $[0, 2^8 - 1] = \{0, \ldots, 255\}$, z.B. $\mathbf{x} = (11, 7, 4, 3)$
- Die Hashfunktion wird auch durch ein 4-Tupel von Integers, aber aus dem Intervall $[0, 269 - 1] = \{0, \ldots, 268\}$, spezifiziert z.B. $\mathbf{a} = (2, 4, 261, 16)$

$\Rightarrow$ Hashfunktion:

$$h_{\mathbf{a}}(\mathbf{x}) = (2x_1 + 4x_2 + 261x_3 + 16x_4) \bmod 269$$

$$h_{\mathbf{a}}(\mathbf{x}) = (2 \cdot 11 + 4 \cdot 7 + 261 \cdot 4 + 16 \cdot 3) \bmod 269 = 66$$

H. Seidl (TUM) GAD SS'15 186

Eindeutiges a_j

Beweis

- Betrachte zwei beliebige verschiedene Schlüssel $\mathbf{x} = \{x_1, \ldots, x_k\}$ und $\mathbf{y} = \{y_1, \ldots, y_k\}$
- Wie groß ist $\Pr[h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})]$?
- Sei j ein Index (von evt. mehreren möglichen) mit $x_j \neq y_j$ (muss es geben, sonst wäre $\mathbf{x} = \mathbf{y}$)

$\Rightarrow$ $(x_j - y_j) \not\equiv 0 \bmod m$
d.h., es gibt genau ein multiplikatives Inverses $(x_j - y_j)^{-1}$

$\Rightarrow$ gegeben Primzahl m und Zahlen $x_j, y_j, b \in \{0, \ldots m - 1\}$ hat jede Gleichung der Form

$$a_j(x_j - y_j) \equiv b \bmod m$$

eine eindeutige Lösung: $a_j \equiv (x_j - y_j)^{-1} b \bmod m$

Wann wird $h(\mathbf{x}) = h(\mathbf{y})$?

Beweis.

Wenn man alle Variablen a_i außer a_j festlegt, gibt es exakt eine Wahl für a_j, so dass $h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y})$, denn

$$\begin{aligned}
h_{\mathbf{a}}(\mathbf{x}) = h_{\mathbf{a}}(\mathbf{y}) \quad &\Leftrightarrow \quad \sum_{i=1}^{k} a_i x_i \equiv \sum_{i=1}^{k} a_i y_i \mod m \\
&\Leftrightarrow \quad a_j(x_j - y_j) \equiv \sum_{i \neq j} a_i(y_i - x_i) \mod m \\
&\Leftrightarrow \quad a_j \equiv (x_j - y_j)^{-1} \sum_{i \neq j} a_i(y_i - x_i) \mod m
\end{aligned}$$