Script generated by TTT

Title: T?ubig: GAD (10.05.2012)

Date: Thu May 10 12:29:25 CEST 2012

Duration: 46:59 min

Pages: 18

Übersicht

Sortierte Sequenz

S: sortierte Sequenz

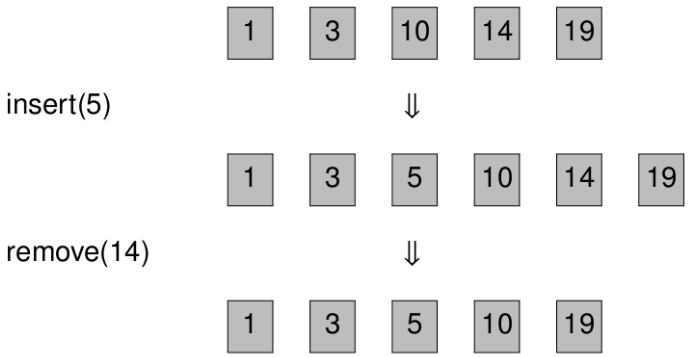
Jedes Element e identifiziert über key(e)

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

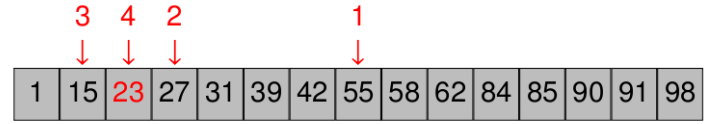
- insert und remove kosten zwar eigentlich nur konstante Zeit, müssen aber wie find zunächst die richtige Position finden
- find auf Sequenz der Länge n kostet $O(n)$ Zeit, damit ebenso insert und remove

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):

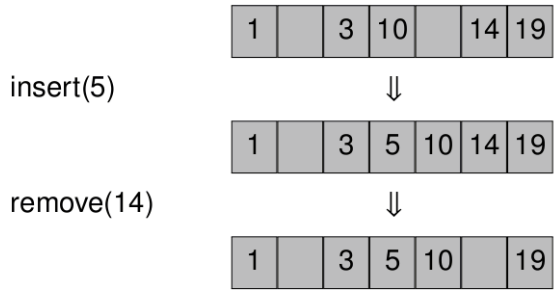


In einer n-elementigen Sequenz kann ein beliebiges Element in maximal $2 + \lfloor \log_2 n \rfloor$ Schritten gefunden werden.

Sortierte Sequenz

Kann man insert und remove besser mit einem Feld realisieren?

⇒ Vorbild Bibliothek: Lücken lassen



Durch geschickte Verteilung der Lücken:

⇒ amortisierte Kosten für insert und remove $\Theta(\log^2 n)$

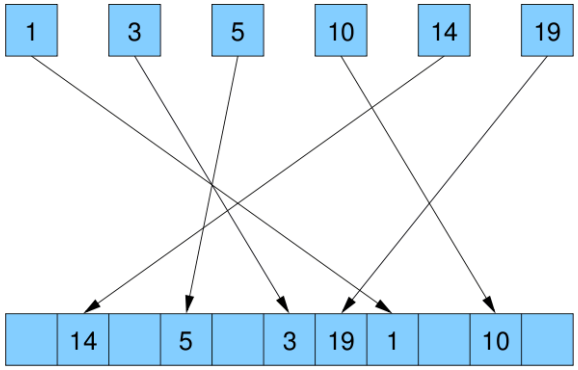
Übersicht

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)
- 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)

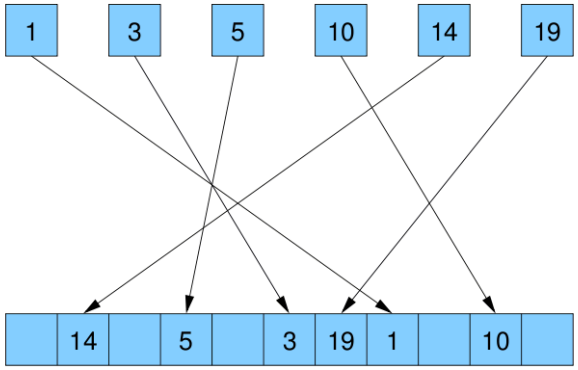
Hashing Hashtabellen

Hashfunktion und Hashtabelle

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

$|\text{Key}| = N$

gespeicherte
Elemente: n



Hashtabelle T

H. Täubig (TUM) GAD SS'12 163 / 631

Hashing Hashtabellen

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))]

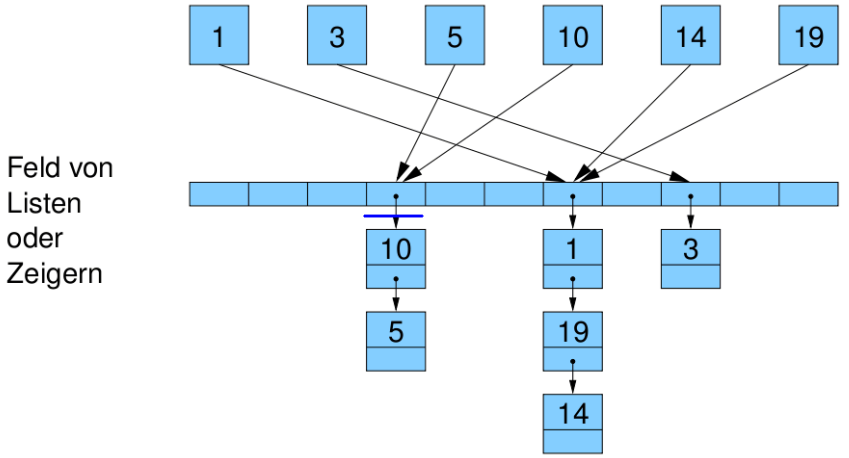
$\Rightarrow$ find, insert und remove in konstanter Zeit
(genauer: plus Zeit für Berechnung der Hashfunktion)

H. Täubig (TUM) GAD SS'12 164 / 631

Übersicht

Dynamisches Wörterbuch

Hashing with Chaining:



unsortierte verkettete Listen
(Ziel: Listen möglichst kurz)

Dynamisches Wörterbuch

Hashing with Chaining:

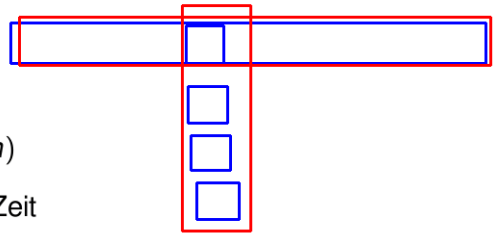
```
List<Elem>[m] T;

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 Menge von mindestens N/m Schlüsseln, die demselben Tabelleneintrag zugeordnet werden
- meistens ist $n < N/m$ und in diesem Fall kann die Suche immer zum Scan aller Elemente entarten

$\Rightarrow$ Auswege

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

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

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);
}
```