

Algorithmen und Datenstrukturen

Sommersemester 2021

Stefan Näher

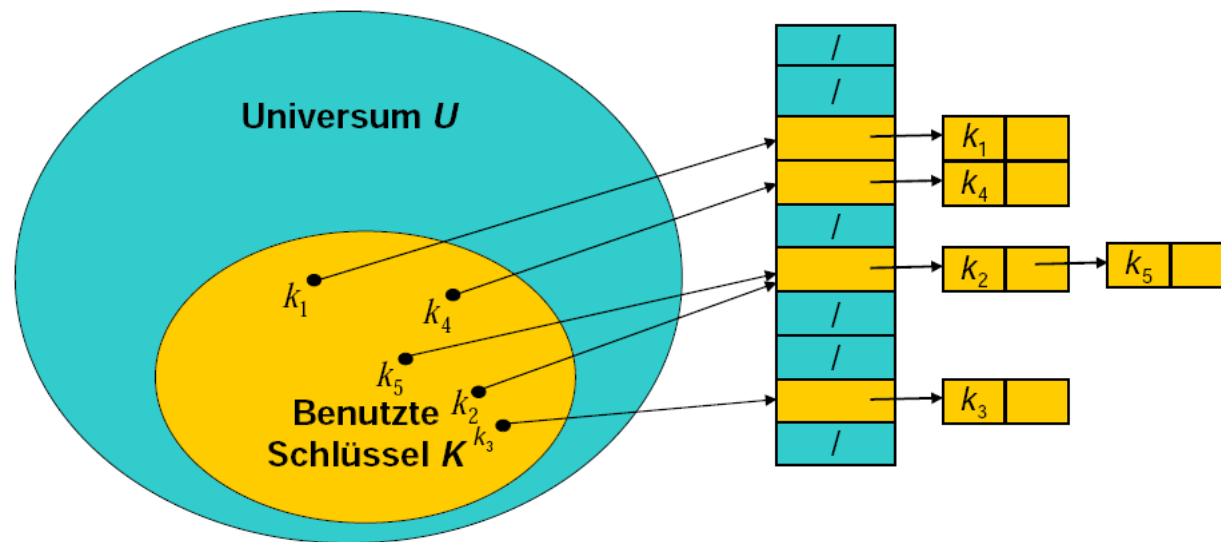
Universität Trier

naeher@uni-trier.de

Vorlesung 10

6. Mai 2021

Hashing mit Listen - Illustration



Hashing mit Verkettung

Betrachte $h : U \rightarrow [0..m - 1]$.

$$\delta_h(x, y) := \begin{cases} 1 & \text{falls } x \neq y \text{ und } h(x) = h(y) \\ 0 & \text{sonst} \end{cases}$$

Wir erweitern auf Schlüsselmengen $S \subseteq U$ mit

$$\delta_h(x, S) = \sum_{y \in S} \delta_h(x, y).$$

Ist S die Menge der abgespeicherten Schlüssel, so ist $\delta_h(x, S)$ die Länge der Liste an der Stelle $h(x)$.

Wie groß ist $\delta_h(x, S)$?

Wir betrachten zwei Fälle:

Im schlechtesten Fall: $n = |S|$, dann generiert Hashing zu linearer Suche.

Im mittleren Fall: Analyse folgt...

Hashing mit Verkettung (mittlerer Fall)

Betrachte $h : U \rightarrow S = \{0, \dots, m - 1\}$.

Annahmen:

- (1) h verteilt S gleichmäßig.
- (2) Die Schlüsselmenge $S = \{x_1, \dots, x_n\}$ wird gemäß Gleichverteilung und unabhängig voneinander “gezogen”

Frage: Wie groß ist $\delta_h(x, S)$ im Mittel ?

Sei $h_i = h(x_i)$. Die ZV h_i nimmt nach Annahmen die Zahlen in $[0..m - 1]$ gleichwahrscheinlich an.

Hashing mit Verkettung (mittlerer Fall) (Forts.)

Wie groß ist die Wahrscheinlichkeit, dass ℓ der Zahlen h_1, \dots, h_n gleich einer bestimmten Zahl z sind ?

Lösung ist Produkt dreier Terme (Bernoulli-Formel):

$\binom{n}{\ell}$: # Möglichkeiten, ℓ Elemente aus n vielen auszuwählen

$m^{-\ell}$: Wahrscheinlichkeit für jedes der ausgewählten Elemente, mit z übereinzustimmen.

$\left(\frac{m-1}{m}\right)^{n-\ell}$: Wahrscheinlichkeit für jedes der nicht-ausgewählten Elemente, mit z nicht übereinzustimmen.

Dieses Produkt ist also die Wahrscheinlichkeit dafür, dass eine feste aber beliebige Liste die Länge ℓ besitzt.

Hashing mit Verkettung (mittlerer Fall) (Forts.)

$$\begin{aligned}
 E[\delta_h(x, \{x_1, \dots, x_n\})] &= \sum_{\ell \geq 0} P(\delta_h(x, \{x_1, \dots, x_n\}) = \ell) \cdot \ell \\
 &= \sum_{\ell \geq 1} \binom{n}{\ell} m^{-\ell} \left(\frac{m-1}{m}\right)^{n-\ell} \cdot \ell \\
 &= \frac{n}{m} \cdot \sum_{\ell \geq 1} \frac{(n-1)!}{(\ell-1)!(n-\ell)!} \left(\frac{1}{m}\right)^{\ell-1} \left(\frac{m-1}{m}\right)^{n-\ell} \\
 &= \frac{n}{m} \cdot \underbrace{\sum_{\ell \geq 1} \binom{n-1}{\ell-1} \left(\frac{1}{m}\right)^{\ell-1} \left(\frac{m-1}{m}\right)^{(n-1)-(\ell-1)}}_{\left(\frac{1}{m} + \frac{m-1}{m}\right)^{n-1} = 1} \\
 &= \frac{n}{m}
 \end{aligned}$$

Hashing mit Verkettung (mittlerer Fall)

Unter den gemachten Annahmen können wir also schlussfolgern, wenn wir den Belegungsfaktor $\beta = \frac{n}{m}$ setzen:

1. Die mittleren Kosten der n -ten Einfügung sind $\mathcal{O}(\beta)$.
 2. Die mittleren Kosten von n Einfügungen sind daher $\mathcal{O}(\beta n)$.
- Der Belegungsfaktor entspricht dabei der mittleren Listenlänge.

Das bedeutet: Mittleres Verhalten von Hashing mit Verkettung ist sehr gut beim Einfügen, nämlich $\mathcal{O}(1)$, wenn β klein ist.

Platzausnutzung ist gut, wenn β nicht zu klein ist, z.B. bei $\beta \approx \frac{1}{4}$ ist beides gewährleistet.

Rehashing

Ungünstige Belegungsfaktoren kann man durch Verdoppeln bzw. Halbieren der Hashtabelle vermeiden.

Dazu **Folge von Hashfunktionen und Tafeln** $h_i : U \rightarrow [0..2^i - 1]$, $i = 1, 2, 3, \dots$
Benutze h_i , solange $2^{i-2} \leq n \leq 2^i$, also $\frac{1}{4} \leq \beta \leq 1$.

Falls $n = 2^i + 1$, verdoppele Tafel: Umspeichern mit h_{i+1}

Falls $n = 2^{i-2} - 1$, halbiere Tafel: Umspeichern mit h_{i-1} .

Eigenschaften:

- (1) Zugriff und Einfügen kosten weiterhin $\mathcal{O}(1)$ im Mittel.
- (2) Übergang von h_i auf h_{i+1} bzw. auf h_{i-1} kostet $\mathcal{O}(n)$.

Verhältnis von Rehash-Kosten zu den übrigen Kosten

Beobachtungen:

(1) Nach einem Rehash ist $\beta = \frac{1}{2}$.

(2) Beginnen wir, mit h_i zu arbeiten, können wir (wegen (1)) wenigstens

$$\min(2^i - 2^{i-1}, 2^{i-1} - 2^{i-2}) \geq 2^{i-2}$$

Operationen mit h_i durchführen, bevor wieder ein Rehash nötig wird.

(3) Wenn wir die Kosten für das Rehash auf diese Operationen umlegen (**amortisierte Analyse**), so benötigt jede dieser Operationen immer noch $\mathcal{O}(1)$ Zeit.

Hinweis: Der benutzte Verdoppelungs- bzw. Halbierungstrick ist oft einsetzbar.

Diskussion der gemachten Annahmen

(1) h verteilt U gleichmäßig

Diese Annahme ist leicht zu erfüllen, z.B. durch die Hashfkt. $h(x) = x \bmod m$

(2) Die Schlüsselmenge S wird gemäß Gleichverteilung und unabhängig voneinander “gezogen”.

Dies ist im Grunde vom “Benutzer” der Hashfunktion abhängig, der aber wiederum die genaue Darstellung von U oft nicht kennt und daher dieses Kriterium nicht erfüllen kann. Untersuchungen an praktischen Beispielen haben jedoch gezeigt, dass die theoretischen Ergebnisse “meist” mit denen aus der Praxis übereinstimmen.

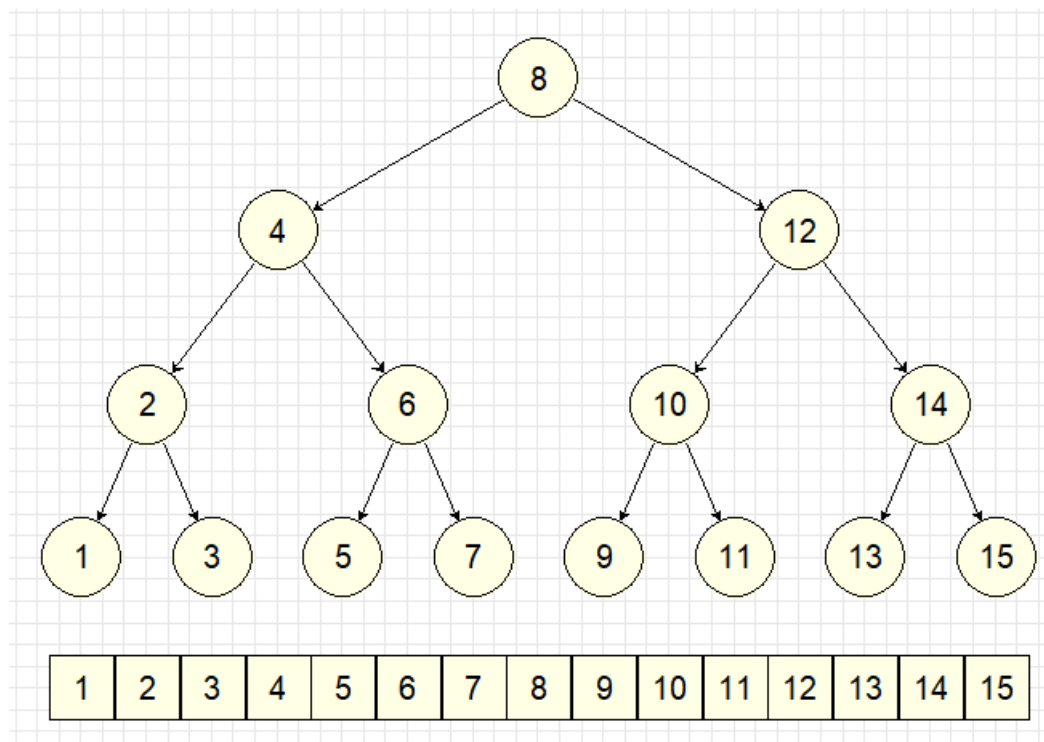
Folgerung aus den gemachten Annahmen

Hashing mit Verkettung gewährleistet, dass die *mittleren Kosten* des Nachschlagens, Einfügens und Löschens eines Elementes $\mathcal{O}(1)$ betragen. Die Kosten im schlechtesten Fall sind jedoch linear d.h. $\mathcal{O}(n)$.

Der Platzbedarf ist ebenfalls linear.

Binäre Suchbäume

Sei $A[1..n]$ ein aufsteigend sortiertes Feld. Ein **binärer Suchbaum** repräsentiert alle möglichen Abläufe der **Binärsuche** auf A .



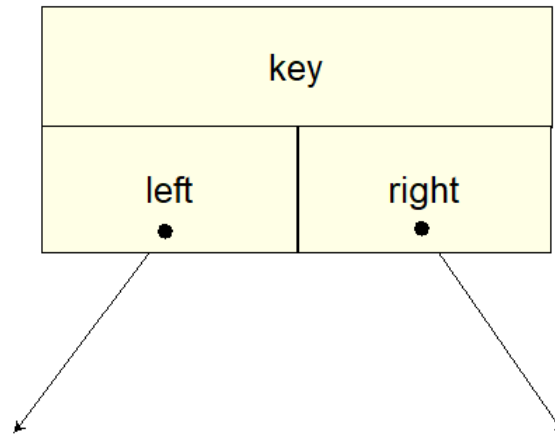
Rekursive Definition

Binärer Suchbaum T für ein Teil-Feld $A[\ell..r]$

Sei $m = \lfloor (\ell + r)/2 \rfloor$ die Position des mittleren Elements.
Dann ist T ein binärer Baum mit

1. die Wurzel v von T speichert das Element $A[m]$
2. der linke Unterbaum von v ist ein binärer Suchbaum für $A[\ell..m - 1]$
3. der rechte Unterbaum von v ist ein binärer Suchbaum für $A[m + 1..r]$

Knoten-Struktur: bin_tree_node



```
class bin_tree_node {  
    int key;  
    bin_tree_node left; // Verweis auf linkes Kind  
    bin_tree_node right; // Verweis auf rechtes Kind  
};
```

$p \leftarrow \mathbf{new} \text{ bin_tree_node};$

$p.\text{key} \leftarrow x;$

$p.\text{left} \leftarrow \dots;$

$p.\text{right} \leftarrow \dots;$

Rekursiver Aufbau

```
1. BUILD_TREE( $\ell$ ,  $r$ )  
   // baut einen binären Suchbaum für  $A[\ell, r]$  auf und gibt die Wurzel zurück  
3. if  $\ell > r$  then return null; fi // leerer Baum  
4.  $m \leftarrow \lfloor (\ell + r)/2 \rfloor$ ;  
5.  $v \leftarrow$  new bin_tree_node;  
6.  $v.\text{key} \leftarrow A[m]$ ;  
6.  $v.\text{left} \leftarrow$  BUILD_TREE( $\ell$ ,  $m - 1$ );  
7.  $v.\text{right} \leftarrow$  BUILD_TREE( $m + 1$ ,  $r$ );  
8. return  $v$ ;
```

Aufbau des binären Suchbaums für das gesamte Feld $A[1..n]$

$\text{root} \leftarrow$ BUILD_TREE(1, n); // Laufzeit: $\mathcal{O}(n)$ (**warum ?**)

Suche in einem binären Suchbaum

LOOKUP(x) liefert den Knoten der x enthält oder null, falls x nicht in T .

```
1. LOOKUP( $x$ )
2.  $p \leftarrow \text{root}$ ;
3. while  $p \neq \text{null} \wedge p.\text{key} \neq x$  do
4.   if  $x < p.\text{key}$ 
5.   then  $p \leftarrow p.\text{left}$ ;
6.   else  $p \leftarrow p.\text{right}$ ;
7. fi ;
8. od;
9. return  $p$ ;
```

Laufzeit: $\mathcal{O}(\text{Höhe}(T))$.

Einfügen in einen binären Suchbaum

INSERT(x): fügt einen Knoten (genauer Blatt) mit Schlüssel x hinzu (falls $x \notin T$).

1. INSERT(x)
2. // zunächst wie LOOKUP: merke den Parent von p in q
3. $p \leftarrow \text{root}; q \leftarrow \text{null};$
5. **while** $p \neq \text{null} \wedge p.\text{key} \neq x$ **do**
6. $q \leftarrow p;$
7. **if** $x < p.\text{key}$
8. **then** $p \leftarrow p.\text{left};$
9. **else** $p \leftarrow p.\text{right};$
10. **fi** ;
11. **od**;
12. **if** $p \neq \text{null}$ **then return** ;**fi** ; // x gefunden (nichts zu tun)

// füge neues Blatt u ein

13. $u \leftarrow \mathbf{new} \text{ bin_tree_node};$

14. $u.\text{key} \leftarrow x;$

15. $u.\text{left} \leftarrow \text{null};$

16. $u.\text{right} \leftarrow \text{null};$

17. **if** $x < q.\text{key}$

18. **then** $q.\text{left} \leftarrow u;$

19. **else** $q.\text{right} \leftarrow u;$

20. **fi**

Laufzeit: $O(\text{Höhe}(T)).$

Löschen in einem binären Suchbaum

DELETE(x) entfernt den Knoten v mit $v.key = x$ aus T

Sei p der Knoten der x enthält und q der Vater von p (null, falls $p = \text{root}$);

Wir unterscheiden 3 Fälle

Fall 1: p ist ein **Blatt** ($p.\text{left} = p.\text{right} = \text{null}$)

Fall 2: p hat **ein Kind** (entweder $p.\text{left} = \text{null}$ oder $p.\text{right} = \text{null}$)

Fall 3: p hat **zwei Kinder** ($p.\text{left} \neq \text{null}$ und $p.\text{right} \neq \text{null}$)