

1 Randomized Data Structures

1.1 Skip Lists

Idee – skip Lists

- Einfügen von „Express-Liste“ mit einigen Elementen
- Beginne mit Suche in der Express-Liste mit weniger Elementen
- Falls das suchende Element kleiner als nächstes Element in Express-Liste \Rightarrow weiter nach rechts
- Falls nicht \Rightarrow Eine Stufe nach unten wandern und dort weiter suchen

Mögliche Verbesserung

- Zusätzliche Stufen an Express-Listen

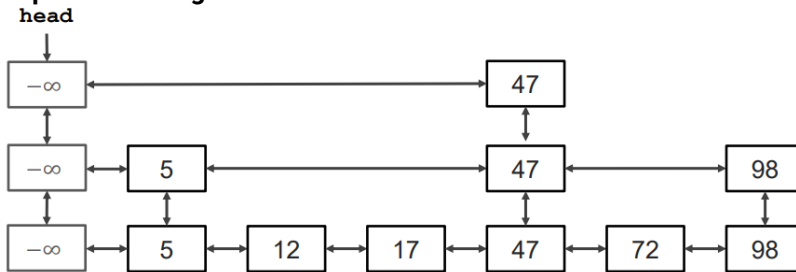
Anwendung

- Gut für parallele Verarbeitung z.B. Multicore-Systeme (Einfügen und Löschen)
- Dafür logarithmische Laufzeit nur im Durchschnitt

Auswahl von Elementen

- Abhängig von einer gewählten Wahrscheinlichkeit p
- Element kommt mit Wahrscheinlichkeit p in übergeordnete Liste
- Höhe: $h = O(\log_{\frac{1}{p}} n)$
- Anzahl Elemente: $n \Rightarrow pn \Rightarrow p^2n \Rightarrow \dots$ (unten nach oben)

Implementierung



L.head – erstes/oberstes Element der Liste
L.height – Höhe der Skiplist
x.key – Wert
x.next – Nachfolger
x.prev – Vorgänger
x.down – Nachfolger Liste unten
x.up – Nachfolger Liste oben
nil – kein Nachfolger / leeres Element

Abbildung 1: Beispiel Skip List

Suche

Laufzeit ist von Expresslisten abhängig

search(L, k)

```
1 current = L.head;
2 WHILE current != nil DO
3   IF current.key == k THEN
4     return current;
5   IF current.next != nil AND current.next.key <= k THEN
6     current = current.next;
7   ELSE
8     current = current.down;
9 return nil;
```

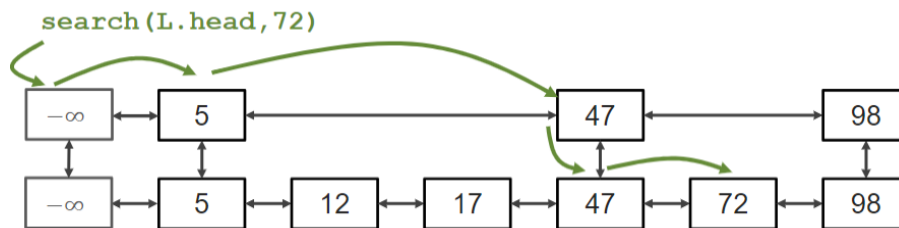


Abbildung 2: Beispiel Suche in einer Skip List

Einfügen

- Füge auf unterster Ebene ein
- Evtl. auf höheren Ebenen mit zufälliger Wahl mithilfe von p auf jeder Ebene
- falls ein Element nicht auf die nächst höhere Ebene gelangt, gelangt es auch nicht auf andere höhere Ebenen (Abbruch des Auswahlprozesses)

Löschen

- Entferne Vorkommen des Elements aus allen Ebenen

Laufzeiten

Einfügen $\Theta(\log_{\frac{1}{p}} n)$

Löschen $\Theta(\log_{\frac{1}{p}} n)$

Suchen $\Theta(\log_{\frac{1}{p}} n)$

- O -Notation versteckt konstanten Faktor $\frac{1}{p}$
- Speicherbedarf im Durchschnitt: $\frac{n}{1-p}$

1.2 Hashtables

Idee – Hashtable

- Hashfunktion sollte gut verteilen
- $h(x)$ sollte uniform sein
- Unabhängig im Intervall $[0, T.length - 1]$ verteilt
- Einfügen mit konstant vielen Array-Operationen
- Kollisionsauflösung z.B. mithilfe von LinkedLists
- Neue Elemente werden vorne angefügt
- Konstante Anzahl an Array-Operationen
- Soviele Schritte wie die Liste lang ist
- Uniforme Hashfunktion

$\Rightarrow \frac{n}{T.length}$ Einträge pro Liste

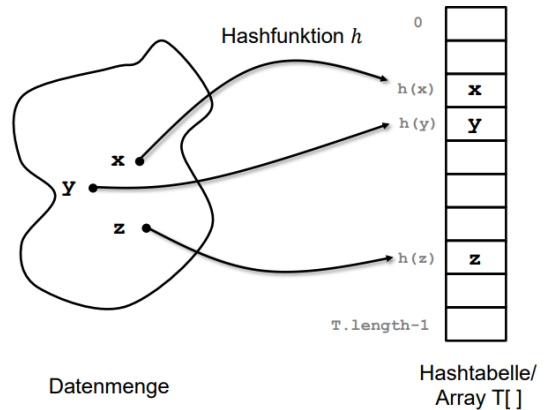


Abbildung 3: Beispiel Hashfunktion

Hash-Funktionen

Universelle Hash-Funktion

- Wähle zufällige $a, b \in [0, p - 1]$, p prim, $a \neq 0$
- $h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod T.length$

Kryptographische Hash-Funktionen

- MD5, SHA-1, SHA-2, SHA-3
- $h(x) = MD5(x) \bmod T.length$

Hashtables vs. Bäume

- | | |
|------------|---|
| Hashtables | <ul style="list-style-type: none"> • nur Suche nach bestimmten Wert möglich • meist gröSSer als zu erwartende Anzahl Einträge |
| Bäume | <ul style="list-style-type: none"> • schnelles Traversieren zu Nachbarn möglich • Bereichssuche möglich |

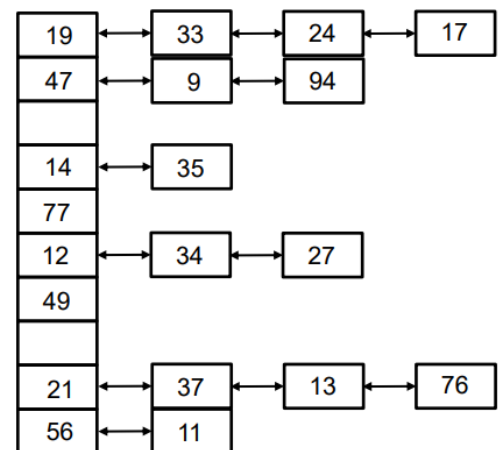
Laufzeiten

Einfügen $\Theta(1)$

Löschen $\Theta(1)$

Suchen $\Theta(1)$

- Für $T.length = n$ ergibt sich konstante Laufzeit
- (Im Durchschnitt, beim Einfügen sogar im Worst-Case)
- Speicherbedarf i.d.R. höher als n, meist ca. $1,33 \cdot n$



Hashtabelle/
Array T[]

Abbildung 4: Beispiel Hashtabelle

1.3 Bloom-Filter

Idee – Bloom-Filter

Speicherschonende Wörterbücher mit kleinem Fehlerpotenzial z.B. Vermeidung von schlechten Passwörtern

1. Abspeichern aller schlechten Passwörter in kompakter Form
2. Prüfe, ob eingegebenes Passwort im Bloom-Filter

z.B. Erkennen von schädlichen Websites (Chrome früher)

Erstellen

- n Elemente x_0, \dots, x_{n-1}
- m Bits-Speicher z.B. als Bit-Array
- k gute Hash-Funktionen H_0, \dots, H_{k-1} mit Bildbereich $0, 1, \dots, m-1$
- Empfohlene Wahl: $k = \frac{m}{n} \cdot \ln 2$ (Fehlerrate von ca. 2^{-k})

Code:

```
initBloom(X, BF, H) // H Array of hash functions
```

```
1 FOR i = 0 TO BF.length - 1 DO
2   BF[i] = 0;
3 FOR i = 0 TO X.length - 1 DO
4   FOR j = 0 TO H.length - 1 DO
5     BF[H[j](X[i])] = 1;
```

1. Initialisiere Array mit "0er-Einträgen
2. Schreibe für jedes Element in jede Bit-Position $H_0(x_i), \dots, H_{k-1}(x_i)$ eine 1

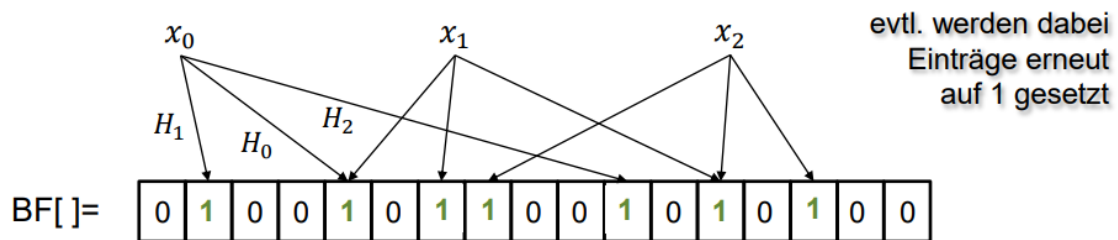


Abbildung 5: Beispiel Bloom Filter

Suche

```
searchBloom(BF, H, y)
```

```
1 result = 1;  
2 FOR j = 0 TO H.length - 1 DO  
3   result = result AND BF[H[j](y)];  
4 return result;
```

- Gibt an, dass y im Wörterbuch, falls alle k Einträge für y in $BF = 1$ sind

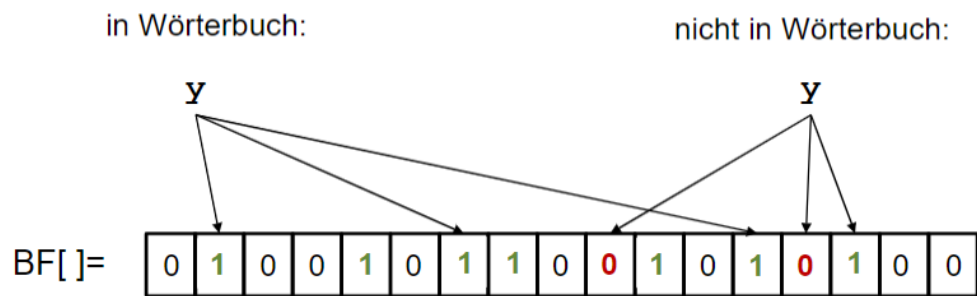


Abbildung 6: Beispiel Suche im Bloom Filter

- Eventuell "false positives" (1, obwohl y nicht im Wörterbuch)
 - Passiert, falls die Einträge vorher von anderen Werten getroffen wurden
 - Daher gute Hashfunktionen und Filtergröße nicht zu klein