Andere Methoden zur Kollisionsauflösung

Neben Verkettung gibt es viele andere Methoden, Kollision zu behandeln:

- Linear Probing
- Quadratic Probing
- Double Hashing
- Sekundäre Hashtabelle

und viele andere. . .

Universelle Familien von Hashfunktionen

Sei $U = \{0, \ldots, p-1\}$, wobei p eine Primzahl ist.
Es sei $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$.
Wir definieren

$$\mathcal{H} = \{\, h_{a,b} \mid 1 \leq a < p,\ 0 \leq b < p \}$$

Theorem

$\mathcal{H}$ ist eine universelle Familie von Hashfunktionen.

Es seien $x, y \in \{0, \ldots, p-1\}$, $x \neq y$.
Wir wollen zunächst zeigen, daß die Funktion

$$f : (a, b) \mapsto (ax + b \bmod p, ay + b \bmod p)$$

für $a, b \in \{0, \ldots, p-1\}$ injektiv und somit auch bijektiv ist.

$$\begin{aligned}
&(ax + b \bmod p, ay + b \bmod p) = (a'x + b' \bmod p, a'y + b' \bmod p) \\
\Leftrightarrow &(ax + b - b' \bmod p, ay + b - b' \bmod p) = (a'x \bmod p, a'y \bmod p) \\
\Leftrightarrow &(b - b' \bmod p, b - b' \bmod p) = ((a' - a)x \bmod p, (a' - a)y \bmod p) \\
\Leftrightarrow &a' = a \wedge b' = b
\end{aligned}$$

Nach wie vor gelte $x, y \in \{0, \ldots, p-1\}$, $x \neq y$.
Für wieviele Paare (a, b) haben $c_x := ax + b \bmod p$ und $c_y := ay + b \bmod p$ den gleichen Rest modulo m?

Wir haben auf der letzten Folie bewiesen, daß sich für jedes Paar (a, b) ein eindeutiges Paar (c_x, c_y) ergibt. Für ein festes c_x gibt es nur

$$\lceil p/m \rceil - 1 = \left\lfloor \frac{p+m-1}{m} \right\rfloor - 1 \leq \frac{p-1}{m}$$

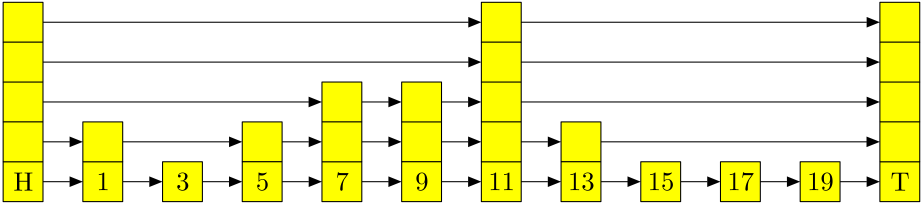
viele mögliche Werte von c_y mit $c_x \equiv c_y \mod m$ und $c_x \neq c_y$.

Weil p verschiedene Werte für c_x existieren, gibt es insgesamt höchstens $p(p-1)/m$ Paare der gesuchten Art.

$$\frac{|\{\, h \in \mathcal{H} \mid h(x) = h(y) \,\}|}{|\mathcal{H}|} \leq \frac{p(p-1)/m}{p(p-1)} \leq \frac{1}{m}$$

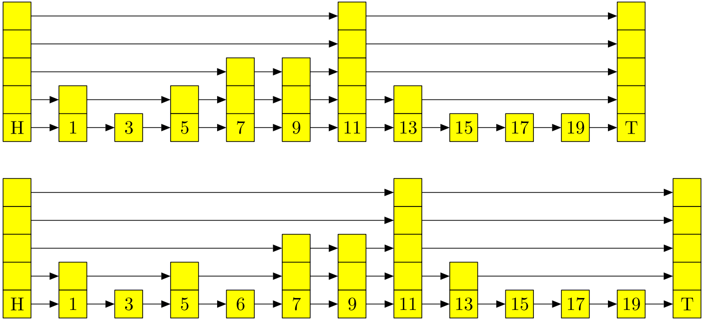
Übersicht

Skip-Lists



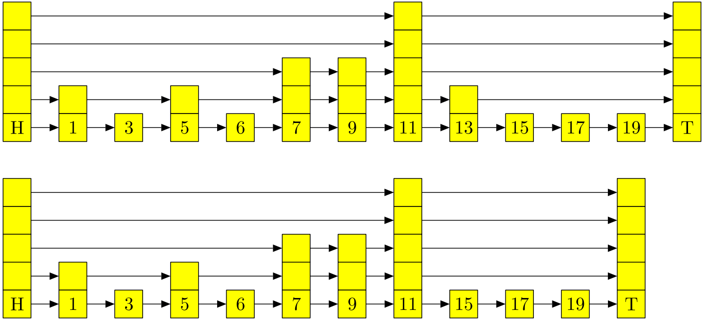
1. Schlüssel nach Größe einsortiert.
2. Es gibt beliebig viele Listen.
3. Anzahl ist geometrisch verteilt.
4. Anzahl ändert sich nicht.
5. Suchen: Von oben nach unten.

Skip-Lists – Einfügen



Einfügen des Elements 6.

Skip-Lists – Löschen



Löschen des Elements 13.

Java

```
public class Skiplist<K extends Comparable<K>, D>
            extends Dictionary<K, D> {
  int size;
  double prob = 0.5;
  Random rand;
  private SkiplistNode<K, D> head;
  private SkiplistNode<K, D> tail;
```

Java

```
public Skiplist() {
  head = new SkiplistNode⟨K, D⟩();
  tail = new SkiplistNode⟨K, D⟩();
  head.succ = new ArrayList⟨SkiplistNode⟨K, D⟩⟩();
  tail.succ = new ArrayList⟨SkiplistNode⟨K, D⟩⟩();
  head.succ.add(tail);
  size = 0;
  rand = new Random();
}
```

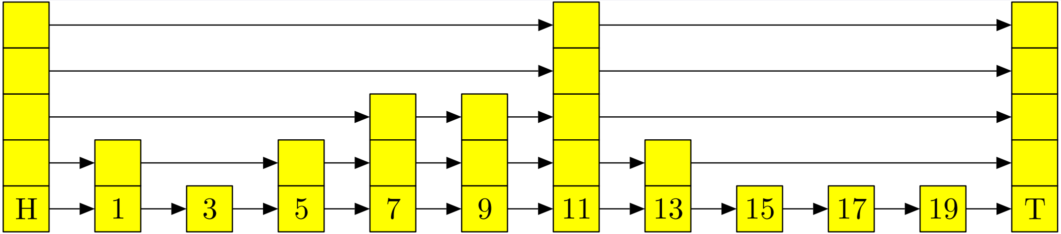
Es gibt noch weitere Konstruktoren, die es erlauben, den Zufallsgenerator und p vorzugeben.

Java

```
public D find(K k) {
  SkiplistNode⟨K, D⟩ n = findnode(k);
  if(n == null) return null;
  return n.getData();
}
```

Java

```
public boolean containsKey(K k) {
  return findnode(k) ≠ null;
}
```



Java

```
SkiplistNode⟨K, D⟩ findnode(K k) {
  SkiplistNode⟨K, D⟩ n = head;
  for(int i = head.succ.size() − 1; i ≥ 0; i−−)
    while(n.succ.get(i) ≠ tail && n.succ.get(i).key.compareTo(k) ≤ 0)
                n = n.getSucc().get(i);
  if(n == head || !n.key.equals(k)) return null;
  return n;
}
```

Java

```
public void insert(K k, D d) {
  delete(k);
  int s = 1;
  while(rand.nextDouble() ≥ prob) s++;
  SkiplistNode⟨K, D⟩ n = new SkiplistNode⟨⟩();
  n.key = k;  n.data = d;
  n.succ = new ArrayList⟨SkiplistNode⟨K, D⟩⟩(s);
  SkiplistNode⟨K, D⟩ m = head;
  for(int i = 0;  i < s;  i++)
    if(i ≥ head.succ.size()) head.succ.add(tail);
  for(int i = head.succ.size() − 1;  i ≥ s;  i−−)
    while(m.succ.get(i) ≠ tail &&
      m.succ.get(i).key.compareTo(k) ≤ 0) m = m.getSucc().get(i);
  for(int i = s − 1;  i ≥ 0;  i−−) m = insertOnLevel(i, m, n);
  size++;
}
```

Java

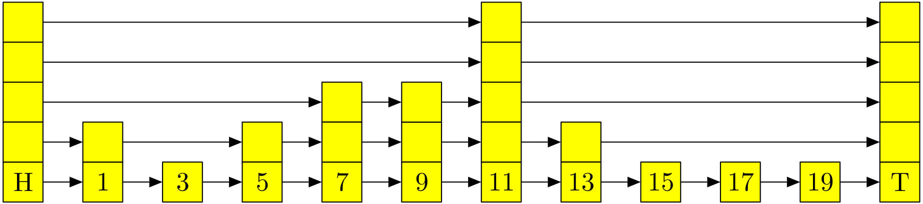
```
SkiplistNode<K, D> insertOnLevel(int i, SkiplistNode<K, D> m,
                                 SkiplistNode<K, D> n) {
  while(m.succ.get(i) ≠ tail &&
    m.succ.get(i).key.compareTo(n.key) < 0) {
      m = m.succ.get(i);
  }
  while(n.succ.size() < i + 1) n.succ.add(null);
  n.succ.set(i, m.succ.get(i));
  m.succ.set(i, n);
  return m;
}
```

Java

```
public void delete(K k) {
  SkiplistNode⟨K, D⟩ n = head;
  if(containsKey(k)) size−−;
  else return;
  for(int i = head.succ.size() − 1; i ≥ 0; i−−) {
    while(n.succ.get(i) ≠ tail &&
          n.succ.get(i).key.compareTo(k) < 0) n = n.succ.get(i);
    if(n.succ.get(i) ≠ tail && n.succ.get(i).key.equals(k))
      n.succ.set(i, n.succ.get(i).succ.get(i));
  }
}
```

Skip-Lists – Analyse

Es sei h die Höhe, n die Anzahl der Schlüssel und p die Wahrscheinlichkeit der Skip-List.



Die erfolgreiche Suche führt entlang eines Wegs, der oben in head beginnt und schlimmstens unten im gesuchten Knoten endet.

Sehen wir uns den Weg rückwärts an!
Wieweit gehen wir im Durchschnitt, bis der Weg nach oben führt?
$\to$ $1/(1-p)$ viele Schritte!
$\to$ Insgesamt $h/(1-p) = O(h)$ Schritte im Erwartungswert.

Skip-Lists – Analyse

Erfolgreiche Suche: $O(h)$

Wie hoch ist eine Skip-Liste mit n Elementen?
Es sei h_i die Höhe des $i$ten Knotens.

$$\Pr[h_i \geq t] = (1-p)^{t-1}$$

$$\Pr[h_i \geq t \text{ für ein } i] \leq n(1-p)^{t-1}$$

Setze $t = -2\log_{1-p}(n) + 1 = 2\log_{1/(1-p)}(n) + 1 = O(\log n)$.

$$\to \Pr[h = O(\log n)] \geq 1 - n(1-p)^{\log_{1-p}(1/n^2)} = 1 - \frac{1}{n}$$

Also gilt $E(h) = O(\log n)(1 - 1/n) + O(n) \cdot 1/n = O(\log n)$.

Skip-Lists – Analyse

Theorem

Skip-Listen unterstützen die Operationen Einfügen, Löschen und Suchen in erwarteter Zeit $O(\log n)$.
Der Speicherverbrauch ist im Erwartungswert $O(n)$.

Beweis.

Löschen benötigt asymptotisch so viel Zeit wie Suchen, also $O(\log n)$.
Einfügen ebenfalls, außer die Höhe nimmt zu. Die Zeit dafür ist aber im Mittel nur $O(1)$, obwohl sie (mit kleiner Wahrscheinlichkeit) unbeschränkt groß werden kann.
Jeder Knoten benötigt im Durchschnitt $O(1)$ Platz, insgesamt ergibt das $O(n)$. $\square$

Skip-Lists – Fragen

Wie lange benötigt eine erfolglose Suche nach einem Element, das größer ist als alle Schlüssel in der Skip-List?

Welchen Fehler darf man bei der Implementierung nicht machen, um dies zu vermeiden:

Die Liste ist fast leer, doch das Einfügen geht sehr, sehr langsam.