Wörterbücher

Der abstrakte Datentyp „Wörterbuch“

Ein **Wörterbuch** für eine gegebene Menge S besteht aus den folgenden Operationen:

- **insert**(x): Füge x zu S hinzu, d.h. setze $S = S \cup \{x\}$.
- **remove**(x): Entferne x aus S, d.h. setze $S = S - \{x\}$.
- **lookup**(x): Finde heraus, ob x in S liegt, und wenn ja, greife gegebenfalls auf den Datensatz von x zu.

- In einer Firmendatenbank werden Kundendaten in der Form (Kundenummer, Info) abgespeichert.
- Die Kundennummer stellt den Schlüssel x dar.
  - **insert**(x): Füge den Datensatz eines neuen Kunden mit Kundenummer x ein.
  - **remove**(x): Entferne den Datensatz des entsprechenden Kunden.
  - **lookup**(x): Greife auf den Datensatz des Kunden mit Kundennummer x zu.

Suchmaschinen

Suchmaschinen müssen Stichworte und Webseiten verwalten und zu jedem Stichwort alle relevanten Webseiten auflisten.

- Für jedes Stichwort s muss ein Wörterbuch der für s relevanten Webseiten aufgebaut werden.
  - Neue Webseiten sind gegebenenfalls einzufügen
  - und alte, verschwundene Webseiten sind zu entfernen.
- Für jede Webseite w müssen die Stichworte gesammelt werden, für die w relevant ist:
  - Sollte w entfernt werden,
    kann w schnell, für jedes seiner Stichworte entfernt werden.

Es gibt mehrere Milliarden Webseiten.

Welche Daten sollten im schnellen Speicher und welche Daten im langsamen Speicher gehalten werden?

Datenstrukturen für Wörterbücher

- Wie sollten **statische Wörterbücher**, also Wörterbücher die nur lookup benutzen, implementiert werden?
  - Sortiere die gespeicherten Schlüssel und führe eine lookup-Operation mit Binärsuche in logarithmischer Zeit durch
  - Oder aber wir haben sogar eine schnell berechenbare Namens- funktion, um die Position eines jeden Schlüssels zu bestimmen.

  Leider sind die interessanten Wörterbücher **dynamisch**.
- Können wir Heaps benutzen?
  - Das Einfügen **gelingt mühelos**,
  - das Suchen ist aber **extrem mühselig**. (Warum?)

Im Gegensatz zu „starren“ Arrays benötigen wir Datenstrukturen, die schnell modifiziert werden können.

Binäre Suchbäume

Binäre Suchbäume

T sei ein geordneter binärer Baum. Jeder Knoten v von T speichert ein Paar

$$\textbf{Daten}(v) = (\textbf{Schlüssel}(v), \textbf{Info}(v)).$$

T heißt **binärer Suchbaum**, wenn T die folgenden Eigenschaften hat:

(a) Für jeden Schlüsselwert x gibt es höchstens einen Knoten v mit Schlüssel $(v) = x$.

(b) Für jeden Knoten $\mathbf{v}$, jeden Knoten $\mathbf{v}_{\textbf{links}}$ im linken Teilbaum von v und jeden Knoten $\mathbf{v}_{\textbf{rechts}}$ im rechten Teilbaum von $\mathbf{v}$ gilt

$$\textbf{Schlüssel}(\mathbf{v}_{\textbf{links}}) < \textbf{Schlüssel}(\mathbf{v}) < \textbf{Schlüssel}(\mathbf{v}_{\textbf{rechts}}).$$

Binäre Suchbäume unterstützen die binäre Suche!

Binäre Suchbäume: lookup(x)

(1) Sei r die Wurzel des binären Suchbaums. Setze $v = r$.
/* Wir beginnen die Suche an der Wurzel. */

(2) Wenn wir am Knoten v angelangt sind, vergleichen wir x und **Schlüssel**(v):

- **x** $=$ **Schlüssel**(v): Wir haben den Schlüssel gefunden.
- **x** $<$ **Schlüssel**(v): Wir suchen im linken Teilbaum weiter.
- **x** $>$ **Schlüssel**(v): Wir suchen im rechten Teilbaum.

Lookup benötigt Zeit $\Theta(t)$,
wobei t die Tiefe des Knotens ist, der den Schlüssel x speichert.

Die Struktur Knoten

```
typedef struct Knoten
    { schluesseltyp schluessel; infotyp info;
    //schluesseltyp und infotyp sind vorher spezifizierte Typen.

    Knoten *links, *rechts;

    Knoten (schluesseltyp s, infotyp i, Knoten *l, Knoten *r)
    { schluessel = s; info = i; links = l; rechts = r; }
    //Konstruktor. };
```

Die Klasse Binärbaum

```
class bsbaum
  {private:
      Knoten *Kopf;
  public:
      bsbaum ( ) { Kopf = new Knoten (0,0,0,0); }
      // Konstruktor.
      // Kopf->rechts wird stets auf die Wurzel zeigen.
      Knoten *lookup (schluesseltyp x);
      void insert (schluesseltyp x, infotyp info);
      void remove (schluesseltyp x);
      void inorder ( ); };
```

Die Funktion lookup

```
Knoten *bsbaum::lookup (schluesseltyp x)
 { Knoten *Zeiger = Kopf->rechts;
 while ((Zeiger != 0) && (x != Zeiger->schluessel))
     Zeiger = (x < Zeiger->schluessel) ? Zeiger->links : Zeiger->rechts;
 return Zeiger; };
```

Binäre Suchbäume: Insert

Zuerst suche nach *x*.

- Sollten wir *x* finden, überschreibe den alten Info-Teil,
- sonst füge den Schlüssel dort ein, wo die Suche scheitert.

```
void bsbaum::insert (schluesseltyp x, infotyp info)
  {Knoten *Eltern, *Zeiger;
  Eltern = Kopf; Zeiger = Kopf->rechts;

  while ((Zeiger != 0) && (x != Zeiger->schluessel))
      {Eltern = Zeiger;
      Zeiger = (x < Zeiger->schluessel) ?
                  Zeiger->links : Zeiger->rechts; }

  if (Zeiger == 0)
      {Zeiger = new Knoten (x, info, 0, 0);
      if (x < Eltern->schluessel) Eltern->links = Zeiger;
      else Eltern->rechts = Zeiger; }
  else Zeiger->info = info; }
```

Binäre Suchbäume: Remove

Zuerst suche den Schlüssel x.

Wenn die Suche im Knoten v endet und

- wenn v ein Blatt ist: Entferne v.
- Wenn v genau ein Kind w hat: Entferne v und mache den Elternknoten von v zum Elternknoten von w.
- Wenn v zwei Kinder hat: Ersetze v durch den kleinsten Schlüssel s im rechten Teilbaum von v.
  - Der Knoten u speichere den Schlüssel s.
  - u ist als linkester Knoten im rechten Teilbaum leicht zu finden.
  - u hat kein linkes Kind und kann damit sofort entfernt werden.

Binäre Suchbäume: Sortieren

Wir können mit binären Suchbäumen auch sortieren:

- Zuerst füge alle Schlüssel in einen leeren Suchbaum ein.
- Danach bestimme die sortierte Reihenfolge durch einen **Inorder-Durchlauf**.

Die Operationen eines binären Suchbaums

Die Operationen **insert** und **remove** beginnen mit einer Suche nach dem Schlüssel.

- **remove** setzt den Suchprozess mit einer Suche nach dem kleinsten Schlüssel im rechten Teilbaum fort.

(a) **lookup, insert** und **remove** benötigen Zeit proportional zur Tiefe des Baums.

(b) Die Folge `insert`(1, info), `insert`(2, info), ... `insert`(n, info) erzeugt einen Baum der (maximalen) Tiefe $n-1$.

(c) Die minimale Tiefe ist $\lfloor \log_2 n \rfloor$, die maximale Tiefe $n-1$.
Wie groß ist die erwartete Tiefe?

Im worst-case große Tiefe, trotzdem, ...

- Die **erwartete Tiefe** und damit die erwartete Zeit für eine erfolgreiche Suche ist **logarithmisch**.

  Also ist die erwartete Zeit für lookup, insert und remove logarithmisch.
- Trotzdem ist die worst-case Laufzeit **intolerabel**.

Und die Konsequenz?

- Wir arbeiten weiter mit binären Suchbäumen,
- garantieren aber durch zusätzliche Operationen, dass der Baum

  tiefen-balanciert

  bleibt.

AVL-Bäume

AVL-Bäume (**A**delson-**V**elski and **L**andis, 1962)

Ein binärer Suchbaum heißt **AVL-Baum**, wenn für jeden Knoten v mit linkem Teilbaum $\mathbf{T_L(v)}$ und rechtem Teilbaum $\mathbf{T_R(v)}$

$$|\ \mathbf{Tiefe(T_L(v)) - Tiefe(T_R(v))}\ | \leq \mathbf{1}$$

gilt. $\mathbf{b(v)} := \text{Tiefe}(T_L(v)) - \text{Tiefe}(T_R(v))$ ist der **Balance-Grad** von v.

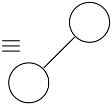
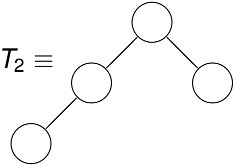
Definiere die Tiefe des leeren Baums als -1.

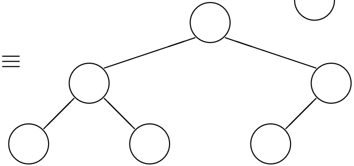
Für AVL-Bäume ist stets $b(v) \in \{-1, 0, 1\}$.

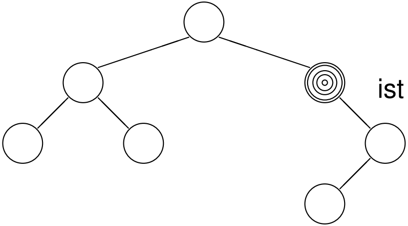
Die **zentralen Fragen**:

- Können wir stets Schlüssel so einfügen, dass der Absolutbetrag des Balance-Grads höchstens Eins ist?
- Wie tief kann ein AVL-Baum mit n Knoten werden?

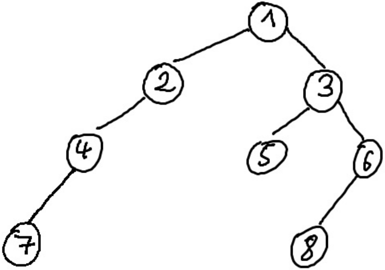
Beispiele und Gegenbeispiele für AVL-Bäume

$T_0 \equiv$  $T_1 \equiv$  $T_2 \equiv$ 

und $T_3 \equiv$  sind AVL-Bäume.

 ist kein AVL-Baum.

An welchen Knoten ist die AVL-Eigenschaft verletzt?



Auflösung: 2

Die Tiefe von AVL-Bäumen

$\mathbf{min(t)}$ sei die minimale Knotenzahl,
die ein AVL-Baum der Tiefe t mindestens besitzen muss.

- $\mathbf{min(0)} = 1$ und $\mathbf{min(1)} = 2$.
- Und es gilt die Rekursion $\mathbf{min(t)} = \mathbf{min(t-1)} + \mathbf{min(t-2)} + 1$.
  - Wenn ein AVL-Baum die Tiefe t besitzt, dann muss ein Teilbaum die Tiefe $t-1$ besitzen und hat mindestens $\mathbf{min(t-1)}$ Knoten.
  - Der andere Teilbaum hat mindestens Tiefe $t-2$ und besitzt deshalb mindestens $\mathbf{min(t-2)}$ Knoten.

Mit induktivem Argument folgt $\mathbf{min(t)} \geq \mathbf{2^{t/2}}$.

- Die Behauptung ist richtig für $t = 0$ und $t = 1$.
- $\mathbf{min(t+1)} = \mathbf{min(t)} + \mathbf{min(t-1)} + \mathbf{1}$
  $\geq \mathbf{2^{t/2}} + \mathbf{2^{(t-1)/2}} + \mathbf{1} \geq \mathbf{2} \cdot \mathbf{2^{(t-1)/2}} = \mathbf{2^{(t+1)/2}}$.

Die Tiefe eines AVL-Baums mit n Knoten ist höchstens $\mathbf{2} \cdot \log_{\mathbf{2}} \mathbf{n}$.

Lookup, Remove und Insert

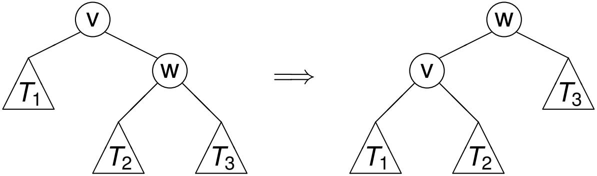
- Da AVL-Bäume logarithmische Tiefe haben, ist die Laufzeit einer **Lookup-Operation** höchstens logarithmisch.
- Wir drücken uns um die **remove-Operation** herum:
  - Wir führen nur eine **lazy remove** Operation aus: Markiere einen gelöschten Knoten als entfernt ohne ihn tatsächlich zu entfernen.
  - Wenn allerdings mehr als 50 % aller Knoten markiert sind, dann beginnt ein **Großreinemachen**:
    - Ein neuer AVL-Baum wird aus den nicht markierten Knoten des alten Baumes durch Insert-Operationen aufgebaut.

  Die Laufzeit für den Neuaufbaus ist groß, aber gegen die **vielen blitzschnellen** remove-Operationen **amortisiert**.

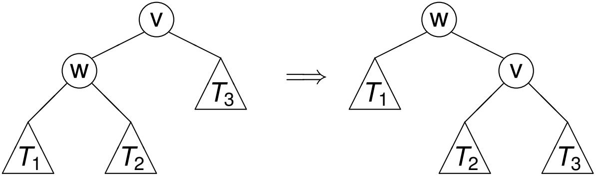
Kritisch ist die Implementierung der **insert-Operation**.

Rotationen

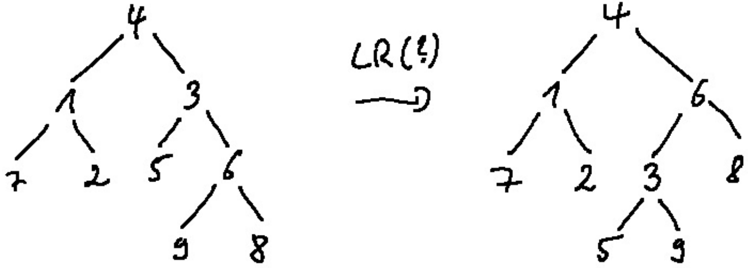
In einer **Linksrotation** ersetzt ein rechtes Kind den Elternknoten.
Der Elternknoten wird zum linken Kind.



Rechtsrotationen sind entsprechend definiert.



An welchem Knoten x wurde eine Linksrotation LR(x) ausgeführt?



Auflösung: 3

Die Insert-Operation

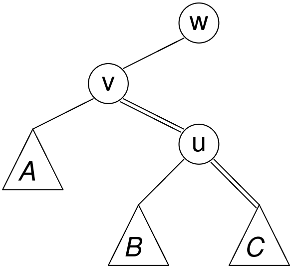
Um den Schlüssel x einzufügen, suche zuerst nach x und füge x am Ende einer erfolglosen Suche ein.

- An welchen Knoten ist jetzt möglicherweise die AVL-Eigenschaft verletzt?
  - Nur Knoten des **Suchpfads**, also des Pfads von der Wurzel zum frisch eingefügten Blatt, können betroffen sein!
  - Wir laufen deshalb den Suchpfad möglicherweise ganz zurück, um die Balance-Eigenschaft zu reparieren.

Die Situation:

- Wir sind bis zum Knoten **u** zurückgelaufen. Die AVL-Eigenschaft gilt für u und alle Nachfahren von u.
- Wenn wir die Reparatur fortsetzen müssen, müssen wir uns als Nächstes um den Elternknoten **v** von u kümmern.
- **w** bezeichne den Großelternknoten von u.

Der Zick-Zick Fall (1/2)



Fallannahme: Ein neues Blatt wurde im Teilbaum von u eingefügt und es gilt $\text{Tiefe}(C) \geq \text{Tiefe}(B)$ nach Einfügung.

- Die Tiefe des Teilbaums von u muss um 1 angewachsen sein, denn ansonsten können wir die Reparatur beenden.
- Sei d die neue, um 1 größere Tiefe des Teilbaums von u.

Der Zick-Zick Fall (2/2)

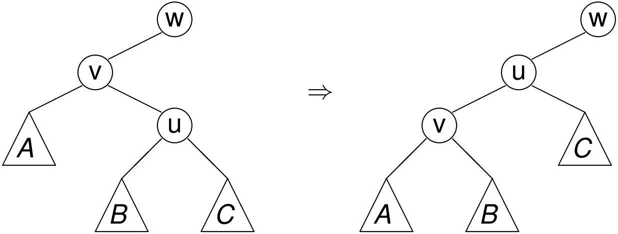
- Tiefe$(A) \geq d + 1$ ist unmöglich, da sonst $b(v) \geq 2$ vor Einfügen des neuen Blatt gilt.
- Wenn Tiefe$(A) = d$, dann brauchen wir nur den Balance-Grad $b(v) = 0$ neu zu setzen.
  - Die Reparatur kann abgebrochen werden, da der Teilbaum mit Wurzel v seine Tiefe nicht verändert hat.
- Wenn Tiefe$(A) = d - 1$, dann setze $b(v) = -1$.

  Diesmal müssen wir die Reparatur in w fortsetzen:
  Die Tiefe des Teilbaums mit Wurzel v ist um 1 angestiegen.
- Der Fall Tiefe$(A) \leq d - 3$ kann nicht auftreten, da sonst $b(v) \leq -2$ vor Einfügen des neuen Blatts gilt.

Der Fall Tiefe$(A) = d - 2$ ist kritisch.

Tiefe(A) $= d - 2$

Führe eine **Linksrotation** in v durch.



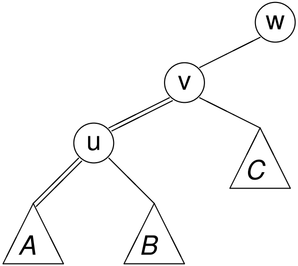
- Tiefe(B) $\leq$ Tiefe(C) gilt nach Fallannahme: Tiefe(C) $= d - 1$ folgt.
- Da die AVL-Eigenschaft in u gilt, folgt

$$d - 2 = \text{Tiefe}(C) - 1 \leq \text{Tiefe}(B) \leq \text{Tiefe}(C) = d - 1.$$

- Die AVL-Eigenschaft gilt somit nach der Rotation für u und v.

  Setze $b(u)$ und $b(v)$ entsprechend und fahre fort, wenn der neue Teilbaum von u tiefer ist als der alte Teilbaum von v.

Der Zack-Zack Fall

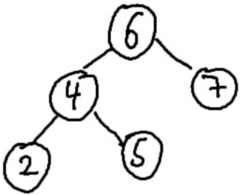


Fallannahme: Ein neues Blatt wurde im Teilbaumvon u eingefügt und Tiefe(A) $\geq$ Tiefe(B) gilt nach Einfügung.

Der Zack-Zack Fall wird wie der Zick-Zick Fall behandelt.

Wo wird wie rotiert? demogr.

Wir fügen Schlüssel 3 in folgenden AVL-Baum ein.

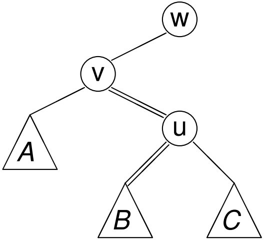


Welche Rotation wird an welchem Knoten ausgeführt?

- (1) LR(2)
- (2) RR(4)
- (3) RR(6)
- (4) LR(4)
- (5) LR(6)

Auflösung: (3) RR(6)

Der Zick-Zack Fall



Fallannahme: Ein neues Blatt wurde im Teilbaum mit Wurzel u eingefügt und $\text{Tiefe}(B) > \text{Tiefe}(C)$ gilt nach Einfügung.

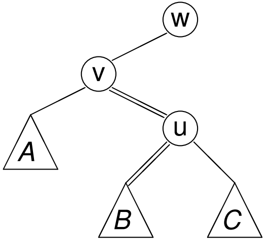
- Die Reparatur muss nur dann fortgesetzt werden, wenn die Tiefe des Teilbaums von u um 1 angestiegen ist.
- Sei d die neue Tiefe des Teilbaums von u.
  Wie im Zick-Zick Fall ist nur der Fall $\text{Tiefe}(A) = d - 2$ kritisch.

Tiefe(A) $= d - 2$

- Da $d - 1 =$Tiefe(B) $>$ Tiefe(C) $= d - 2$, folgt

$$\text{Tiefe}(A) = \text{Tiefe}(C) = d - 2 \text{ und } \text{Tiefe}(B) = d - 1.$$

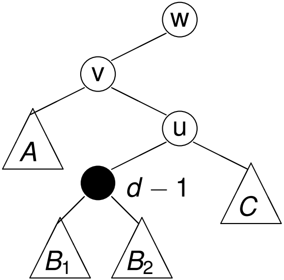
- Eine Linksrotation in v ist keine Reparatur:

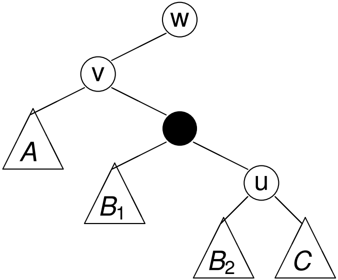


B wandert vom rechten zum linken Teilbaum und die AVL-Eigenschaft bleibt verletzt, da Tiefe (A) $=$ Tiefe(C).

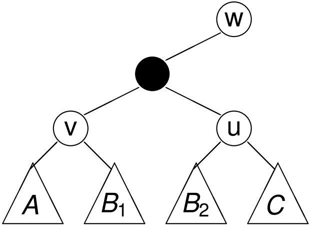
Tiefe(A) $= d - 2$

Zuerst eine **Rechtsrotation** in u



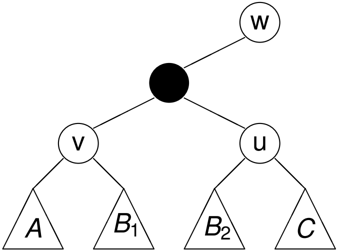


und dann



eine **Linksrotation** in v

Nach der Doppelrotation



- Die Tiefe ist um 1 gesunken, denn $\text{Tiefe}\,(A) = \text{Tiefe}(C) = d - 2$ und $d - 3 \leq \text{Tiefe}(B_1), \text{Tiefe}(B_2) \leq d - 2$.
- Die Tiefe des schwarzen Knotens stimmt jetzt mit der Tiefe d von v vor dem Einfügen des neuen Blatts überein.
- Nach Setzen der neuen Balance-Grade kann die Reparatur abgebrochen werden.

Zusammenfassung

Die Operationen **lookup** und **insert** haben worst-case Laufzeit $O(\log_2 n)$ für AVL-Bäume mit n Knoten.

- Wir haben nur den Zack-Zick Fall ausgelassen, der analog zum Zick-Zack Fall zu behandeln ist.
- Mit AVL-Bäumen können wir schnell sortieren:
  - Füge n Schlüssel in Zeit $O(n \cdot \log_2 n)$ ein
  - und führe dann einen Inorder-Traversal in linearer Zeit aus.

(a, b)-Bäume:
Wörterbücher für Externspeicher

Die (a, b)-Eigenschaft

Es gelte $\mathbf{a \geq 2}$ und $\mathbf{b \geq 2a - 1}$.

Ein Baum T hat die **(a, b)-Eigenschaft**, falls

- alle Blätter von T die gleiche Tiefe haben,
- alle Knoten **höchstens b** Kinder besitzen und
- die Wurzel **mindestens zwei** Kinder hat,
  während alle sonstigen Knoten **mindestens a** Kinder haben.

Interessant sind Bäume mit der (a, b)-Eigenschaft für große Werte von a und b, wenn Daten auf einem Externspeicher abgelegt sind:

- Die Tiefe wird dementsprechend klein sein und
- **wenige** der **sehr langsamen** Zugriffe auf den Externspeicher genügen.

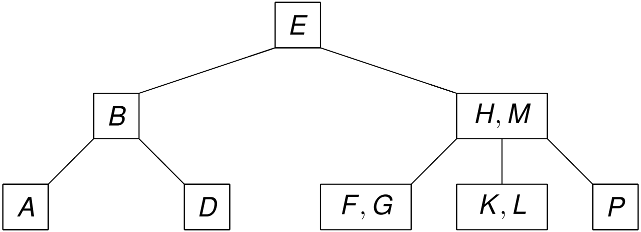
Die Suchstruktur von (a, b)-Bäumen

T ist ein **$(\mathbf{a}, \mathbf{b})$-Baum** für die Schlüsselmenge S,
bzw. ein **B-Baum** für $\mathbf{b = 2a - 1}$, falls gilt:

- T hat die (a, b)-Eigenschaft.
- Jeder Schlüssel in S wird in genau einem Knoten von T gespeichert und jeder Knoten speichert die ihm zugewiesenen Schlüssel in aufsteigender Reihenfolge.
  - Jeder Knoten mit k Kindern speichert genau $k - 1$ Schlüssel.
  - Ein Blatt speichert höchstens $b - 1$ Schlüssel und mindestens $a - 1$ Schlüssel.
- Falls der innere Knoten v die Schlüssel $x_1, ..., x_c$ (mit $x_1 < x_2 < \cdots < x_c$ und $c \leq b - 1$) speichert, dann
  - speichert der linkeste (bzw. rechteste) Teilbaum nur Schlüssel aus dem Intervall $(-\infty, x_1)$ (bzw. (x_c, ∞)).
  - Der i.te Teilbaum (für $2 \leq i \leq c$) speichert nur Schlüssel aus dem Intervall (x_{i-1}, x_i).

Ein Beispiel

Für $a = 2$ und $b = 3$ erhalten wir **2-3 Bäume:**



Die Schlüssel der inneren Knoten helfen in der Suche:

- Auf der Suche nach Schlüssel K suche im rechten Teilbaum weiter, denn $E < K$.
- Da $H < K < M$ muss das mittlere Blatt aufgesucht werden.

Die Tiefe von (a, b)-Bäumen

T sei ein (a, b)-Baum mit n_k Knoten, der n_s Schlüssel speichert.
Dann gilt $n_k < n_s$, und für $\mathrm{Tiefe}(T) \geq 2$:

$$\begin{aligned} & \log_b(n_k) - 1 \\ < {} & \log_b(n_s) - 1 < \mathrm{Tiefe}(T) < \log_a\left(\frac{n_k - 1}{2}\right) + 1 \\ & \qquad\qquad\qquad\qquad < \log_a\left(\frac{n_s - 1}{2}\right) + 1. \end{aligned}$$

Die Tiefe von (a, b)-Bäumen

- Die Tiefe ist minimal, wenn jeder Knoten genau b Kinder hat.
  - In Tiefe t können wir damit höchstens $n_k \leq 1 + b + ... + b^t = \frac{b^{t+1}-1}{b-1}$ Knoten erreichen. Jeder Knoten enthält höchstens $b-1$ Schlüssel, also $n_s \leq n_k \cdot (b-1) \leq b^{t+1} - 1 < b^{t+1}$
  - Also folgt $b^{t+1} > n_s$ und damit $t > \log_b(n_s) - 1$.
- Die Tiefe ist maximal, wenn die Wurzel zwei Kinder und jeder innere Knoten a Kinder hat.
  - Wir erhalten also in Tiefe t mindestens $n_k \geq 1 + 2(1 + ... + a^{t-1}) = 1 + 2 \cdot \frac{a^t-1}{a-1}$ Knoten.
  - Also folgt $n_k \geq 1 + 2 \cdot \frac{a^t-1}{a-1}$, beziehungsweise $\frac{a^t-1}{a-1} \leq \frac{n_k-1}{2}$. Aber $a^{t-1} < \frac{a^t-1}{a-1}$ gilt für $t \geq 2$ und damit $t < \log_a\left(\frac{n_k-1}{2}\right) + 1$.

Knotenanzahl von (a, b)-Bäumen demogr.

Wieviele Knoten muss ein $(3, 7)$-Baum mit Tiefe 3 mindestens haben?

- (1) 3
- (2) 7
- (3) 10
- (4) 21
- (5) 27
- (6) 40

Auflösung: (5) 27

Lookup(x)

Benutze die den inneren Knoten zugeordneten Schlüssel, um den Schlüssel x zu lokalisieren.

Es genügen $\text{Tiefe}(T) + 1 < \log_a \frac{n_s - 1}{2} + 2$ Speicherzugriffe.

- Zum Beispiel: Wähle a als ein Megabyte und n_s als ein Terabyte. Also

$$a = 10^6 \text{ und } n_s = 10^{12}.$$

- Dann genügen **weniger** als $\log_{10^6} 10^{12} + 2$ Zugriffe und damit reichen **drei** Speicherzugriffe.
- Wenn n ein Petabyte ($n_s = 10^{15}$) ist, dann reichen **vier** Zugriffe. Dasselbe gilt sogar für ein Exabyte ($n_s = 10^{18}$).

Für die lookup Operation in einem (a, b)-Baum mit n_s Schlüsseln genügen **weniger als** $\boldsymbol{\log_a \frac{n_s - 1}{2} + 2}$ Speicherzugriffe.

Insert(x)

Zuerst suche nach x.

1. Wenn x gefunden wird, dann überschreibe den Info-Teil, ansonsten endet die Suche in einem **Blatt** v.
2. Füge x in die sortierte Folge der Schlüssel von v ein.

- **Fall 1**: v hat jetzt höchstens $b - 1$ Schlüssel.
  Wir sind fertig, da die (a, b)-Eigenschaft erfüllt ist.
- **Fall 2**: v hat jetzt b Schlüssel $x_1 < ... < x_b$:
  Die (a, b)-Eigenschaft ist verletzt.
  - Ersetze v durch zwei Knoten v_{links} (mit den Schlüsseln $x_1, \ldots, x_{\lceil b/2 \rceil - 1}$) und v_{rechts} (mit den Schlüsseln $x_{\lceil b/2 \rceil + 1}, \ldots, x_b$).
  - Es ist $2a - 1 \leq b$. Also $a - 1 \leq \lfloor \frac{b+1}{2} \rfloor - 1 = \lceil \frac{b}{2} \rceil - 1 \leq \lfloor \frac{b}{2} \rfloor$: v_{links}, v_{rechts} besitzen die notwendige Mindestzahl von Schlüsseln.
  - Der Schlüssel $x_{\lceil b/2 \rceil}$ unterscheidet zwischen v_{links} und v_{rechts}.
    Füge $x_{\lceil b/2 \rceil}$ rekursiv im Elternknoten von v ein.

Wann erhöht sich die Tiefe des (a, b)-Baums?

- Wir fügen zuerst in einem Blatt ein,
  - spalten dann ggf. die Schlüssel unter zwei neuen Knoten auf und
  - fügen **rekursiv** einen trennenden Knoten beim Elternknoten ein.
- Wenn die Wurzel bereits $b - 1$ Schlüssel speichert und einen trennenden Schlüssel zusätzlich erhält, dann
  - muss sie in zwei Knoten aufgespalten werden.
  - Der trennende Schlüssel der beiden neuen Knoten wird zum einzigen Schlüssel der neuen Wurzel.
    - Wir haben erlaubt, dass die Wurzel zwei oder mehr Kinder hat, um diesen Fall abzufangen.
- Auch der Grund für die Bedingung $\mathbf{2 \cdot a - 1 \leq b}$ ist klar:
  - Die Aufspaltung eines Knotens mit b Schlüsseln in zwei Knoten mit legaler Schlüsselzahl muss möglich sein.

Remove(x)

Zuerst müssen wir nach x suchen.

- Angenommen wir finden x in dem **inneren Knoten** v:
  - Wir suchen den kleinsten Schlüssel y mit $x \leq y$.
  - y befindet sich im linkesten Blatt ℓ des entsprechenden Teilbaums von v. Wir ersetzen den Schlüssel x in v durch y.
- Setze $v = \ell$ und entferne x: Das Blatt v verliert einen Schlüssel.

- **Fall 1:** v hat jetzt mindestens $a - 1$ Schlüssel.
  Wir sind fertig, da die (a, b)-Eigenschaft erfüllt ist.
- **Fall 2:** v hat jetzt $a - 2$ Schlüssel.
  1. Zuerst begibt sich Knoten v auf „**Schlüsselklau**“ und stiebitzt, wenn möglich, einen Schlüssel von seinem Elternknoten.
  2. Sollte dies nicht möglich sein, wird v mit einem Geschwisterknoten **fusioniert**.

Schlüsselklau und Fusion

Der Knoten v habe die Schlüssel $x_1 < \cdots < x_{a-2}$.

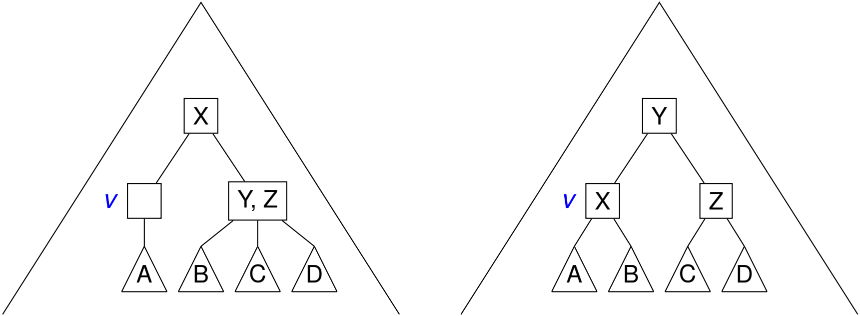
- **Fall 2.1**: Der linke oder rechte Geschwisterknoten hat mindestens a Schlüssel.
  - Z.B. hat der rechte Geschwisterknoten v' die Schlüssel $y_1 < \cdots < y_{a-1} < \cdots < y_{k'}$.
  - Der Schlüssel z des Elternknotens trenne v und v', also $x_{a-2} < z < y_1$.
    1. v klaut z und hat damit $a - 1$ Schlüssel.
    2. Schlüssel z wird durch Schlüssel y_1 ersetzt. Fertig!
- **Fall 2.2:** Beide Geschwisterknoten besitzen nur $a - 1$ Schlüssel.
  - Der rechte Geschwisterknoten v' hat die $a - 1$ Schlüssel $y_1 < \cdots < y_{a-1}$. Verschmelze v' und v.
  - Der bisher trennende Schlüssel z des Elternknotens ist einzufügen
    1. Der fusionierte Knoten hat $a - 1 + a - 2 + 1 = 2a - 2 \leq b - 1$ Schlüssel und die Höchstanzahl wird nicht überschritten.
    2. Der Schlüssel z ist rekursiv aus dem Elternknoten zu entfernen.

Schlüsselklau für innere Knoten

Angenommen wir haben die Remove-Operation rekursiv ausgeführt und haben einen inneren Knoten v erreicht.

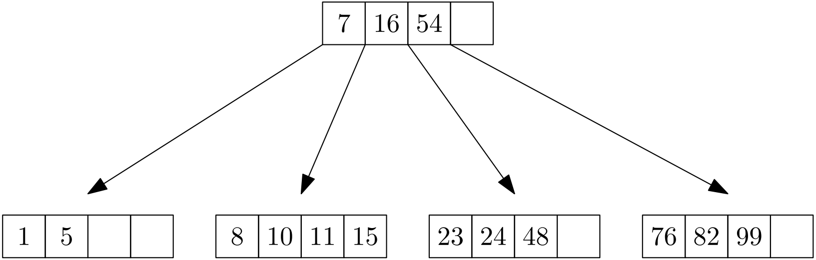
- v hat Schlüssel durch Verschmelzung zweier Kinder verloren.
- Ein Geschwisterknoten von v speichere mindestens a Kinder.

Das Ergebnis des Schlüsselklaus für 2-3 Bäume:



Beispiel

Beispiel $(3,5)$-Baum



Zusammenfassung für (a, b)-Bäume

Sei T ein (a, b)-Baum. Dann genügen

- $\ast$ $\text{Tiefe}(T) + 1$ Speicherzugriffe für eine lookup-Operation,
- $\ast$ $2 \cdot (\text{Tiefe}(T) + 1)$ Speicherzugriffe für eine insert-Operation
- $\ast$ $5 \cdot (\text{Tiefe}(T) + 1)$ Speicherzugriffe für eine remove-Operation.

Beachte, dass $\text{Tiefe}\ (T) < \log_a(\frac{n_k - 1}{2}) + 1$ für Bäume mit n_k Knoten gilt.

- **Lookup**: Der Weg von der Wurzel zu einem Blatt besteht aus $\text{Tiefe}(T) + 1$ Knoten.
- **Insert**: Die Knoten des Suchpfads werden zuerst gelesen und, wenn zu groß, aufgespalten.
- **Remove**: $\text{Tiefe}(T) + 1$ Zugriffe genügen auf dem Weg nach unten und bis zu $4\ \text{Tiefe}(T) + 1$ Zugriffe zurück auf dem Weg nach oben,
  - nämlich das Schreiben des Knotens,
  - das Lesen von zwei Geschwisterknoten und
  - das Schreiben eines Geschwisterknotens.

Hashing

Die Bitvektor-Datenstruktur

Das Wörterbuchproblem wird einfacher, wenn die Menge U der möglicherweise einzufügenden Daten in den Hauptspeicher passt.

- Benutze die **Bitvektor-Datenstruktur**:

  In einem booleschen Array wird für jedes Element $u \in U$ in der Zelle $f(u)$ vermerkt, ob u präsent ist.

  Bis auf die Berechnung von $f(u)$ gelingt damit die Ausführung einer lookup-, insert- oder remove-Operation in konstanter Zeit!
- Selbst bei einem kleinen Universum U ist aber die Bestimmung einer geeigneten Funktion f möglicherweise schwierig.
- Zudem ist in praktischen Anwendungen im Allgemeinen das Universum der möglichen Schlüssel zu groß:

  Wenn Nachnamen als Schlüssel verwandt werden, und selbst wenn nur Nachnamen der Länge höchstens 10 auftreten, gibt es $26^{10} \geq 2^{10} \cdot 10^{10} \geq 10^3 \cdot 10^{10} = 10^{13}$, also mehr als 10 Billionen mögliche Schlüssel!

Hashing

Hashing gehört zu den Datenstrukturen mit der schnellsten erwarteten Laufzeit.

- Sei U die Menge aller möglichen Schlüssel und sei **m** die Größe einer im Hauptspeicher abgespeichernten **Hashtabelle**.
- Eine Funktion

$$h : U \to \{0, 1, ..., m-1\}$$

  heißt eine **Hashfunktion**.
- Zum Beispiel können wir insert (x, info) implementieren, indem wir
  - $h(x) = i$ berechnen und
  - $(x, info)$ in Zelle i der Tabelle eintragen.

  Aber was passiert bei einer **Kollision**, wenn also Zelle i bereits besetzt ist?

Wir beschreiben zwei Hashing-Verfahren,
Hashing mit Verkettung und **Hashing mit offener Adressierung**.

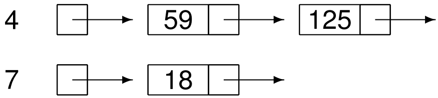
Hashing mit Verkettung

Hashing mit Verkettung

Für jede Zelle i wird eine anfänglich leere Liste angelegt.

- Jede Liste wird sortiert gehalten.
- Für **lookup(x)**: Durchlaufe die Liste von $h(x)$.
- Für **insert(x)** und **remove(x)**: Führe die insert- und remove-Operation für einfach-verkettete Listen aus.

Beispiel: Wähle $\mathbf{h}(\mathbf{x}) = (\mathbf{x} \mod \mathbf{11})$ als Hashfunktion. Die Operationen **insert(59)**, **insert(18)** und **insert**(125) führen auf die Tabelle



lookup (26) benötigt nur einen Suchschritt: Schlüssel 59 wird gefunden und es wird geschlossen, dass 26 nicht präsent ist.

Hashfunktionen

Die Wahl der Hashfunktion

Jeder Schlüssel x wird als Binärzahl dargestellt.
Wir können also annehmen, dass x eine natürliche Zahl ist.

- Eine beliebte und gute Wahl ist $\mathbf{h(x) = x \mod m}$.
  - $h(x)$ kann schnell berechnet werden,
  - aber die Wahl von m ist **kritisch**!
- Wenn m eine Zweierpotenz ist und wenn die Schlüssel Zeichenketten sind, dann werden alle Zeichenketten mit gleicher Endung auf dieselbe Zelle gehasht.
  - Häufig auftretende Endungen provozieren viele Kollisionen und damit lange Listen.
  - Die Bearbeitungszeit der einzelnen Operationen wächst!

Wähle stattdessen Primzahlen mit großem Abstand zur nächsten Zweierpotenz.

Hashing mit offener Adressierung,
wir hashen direkt in die Hashtabelle

Hashing mit offener Adressierung

Wir arbeiten mit einer Folge

$$h_0, ..., h_{m-1} : U \to \{0, ..., m-1\}$$

von Hashfunktionen. Setze $i = 0$.

(1) Wenn die Zelle $h_i(x)$ frei ist, dann füge x in Zelle $h_i(x)$ ein.
(2) Ansonsten setze $i = i + 1$ und gehe zu Schritt (1).

Die Anzahl der Fehlversuche „sollte" ansteigen, wenn die Hashtabelle voll wird. Was ist in einem solchen Fall zu tun?

- Sobald die Tabelle mindestens halb voll ist, dann lade die Tabelle in eine doppelt so große Tabelle.
- Die Zeit für die Reorganisation wird durch die schnellere Bearbeitung der Operationen **amortisiert**.

Wie sollen die einzelnen Operationen implementiert werden?

Implementierung von Lookup, Insert und Remove

- **lookup** und **insert** lassen sich für jede Folge von Hashfunktionen leicht implementieren.
- **Kopfzerbrechen** bereitet **remove**: Wird nach Einfügen des Schlüssels x in Zelle $h_1(x)$ der Schlüssel in Zelle $h_0(x)$ entfernt, dann hat die Operation lookup (x) ein Problem.
  - Ist x nicht da, weil Zelle $h_0(x)$ leer ist oder ist weiterzusuchen?
  - Bringe eine „entfernt“ Markierung nach Löschen des Schlüssels in Zelle $h_0(x)$ an.
  - Die erwartete Laufzeit einer erfolglosen Suche wird anwachsen.

Vermeide Hashing mit offener Adressierung, wenn viele Daten entfernt werden.

Hashing mit offener Adressierung: Welche Hashfunktionen?

Lineares Austesten

In der Methode des **linearen Austestens** wird die Folge

$$\mathbf{h_i(x) = (x + i) \mod m}$$

benutzt: Also wird die jeweils nächste Zelle untersucht.

+ Für jeden Schlüssel x wird jede Zelle in der Folge $h_0(x), \ldots, h_{m-1}(x)$ „getestet".
- Lineares Austesten führt zur **Klumpenbildung**.
  - Angenommen, die Daten besetzen ein Intervall $\{i, i+1, \ldots, j-1, j\}$ von Zellen.
  - Wenn ein weiterer Schlüssel x mit $h_0(x) \in \{i, +1, \ldots, j-1, j\}$ eingefügt wird, dann wird x am Ende des Intervalls eingefügt.
  - Das Intervall wächst und dementsprechend steigt der Aufwand für die einzelnen Operationen.

Lineares Austesten demogr.

Betrachte lineares Austesten mit Hashfunktion $h_i(x) = (x + i) \mod 7$.
Sei die Hashtabelle $H[0..6]$ belegt mit den drei Einträgen

$H[3] = 16, \quad H[4] = 10 \quad$ und $\quad H[2] = 23$.

In welcher Reihenfolge wurden die Zahlen eingefügt?

- (1) 16, 10, 23
- (2) 16, 23, 10
- (3) 10, 16, 23
- (4) 10, 23, 16
- (5) 23, 10, 16
- (6) 23, 16, 10

Auflösung: (6) 23, 16, 10

Doppeltes Hashing

Wir benutzen zwei Hashfunktionen f und g und verwenden die Folge

$$\mathbf{h_i(x) = (f(x) + i \cdot g(x)) \mod m}.$$

- Die Klumpenbildung wird vermieden.
- Man erhält gute Ergebnisse bereits für

$$\mathbf{f(x) = x \mod m} \quad \text{und} \quad \mathbf{g(x) = m^* - (x \mod m^*)}.$$

  - Wähle m als Primzahl und fordere $m^* < m$. Dann ist $g(x)$ niemals Null.
  - Wenn nun $h_i(x) = h_j(x)$, d.h.

$$f(x) + i \cdot g(x) \mod m \quad = \quad f(x) + j \cdot g(x) \mod m,$$

    dann $(i - j) \cdot g(x) = 0 \mod m$. Daraus folgt $i = j$, da m prim.
    Also: $i \neq j$ bedeutet $h_i(x) \neq h_j(x)$.

Im doppelten Hashing werden alle Zellen getestet.

Wie schnell ist Hashing mit Verkettung?

Wie schnell ist Hashing mit Verkettung?

Annahme: Es befinden sich **n** Schlüssel in einer Tabelle mit **m** Schlüsseln. Wir sagen, dass $\lambda = \frac{\mathbf{n}}{\mathbf{m}}$ der **Auslastungsfaktor** der Tabelle ist.

Wie schnell wird eine insert(x), remove(x) oder lookup(x) Operation ausgeführt?

- Bestenfalls ist die Liste für $h(x) = i$ leer und wir erreichen eine **konstante Laufzeit**.
- Schlimmstenfalls sind alle n Schlüssel auf die Liste von i verteilt und die **worst-case Laufzeit** $\Theta(\mathbf{n})$ folgt.

Weder best-case noch worst-case Laufzeit scheinen verlässliche Voraussagen der tatsächlichen Laufzeit zu sein. Die Vorbelegung der Tabelle spielt eine Rolle!

Laufzeit Hashing mit Verkettung demogr.

Für $h(x) = x \mod 7$ seien vier Einträge in der Tabelle gespeichert. Nun wird Schlüssel 8 eingefügt.

Welche Vorbelegung ergibt für Schlüssel 8 die größte Einfügezeit?

- (1) 0,2,4,6
- (2) 0,16,24,32
- (3) 1,3,15,17

Auflösung: (3)

Wir sollten die **erwartete Laufzeit** betrachten.

Die erwartete Laufzeit (1/2)

Wir machen die folgenden Annahmen:

- jedes Element $x \in U$ hat die Wahrscheinlichkeit $\frac{1}{|U|}$ als Operand in einer Operation aufzutreten.
- Die Hashfunktion h **streut** die Schlüssel regelmäßig, d.h. $|\{x \in U \mid h(x) = i\}| \in \left\{\lfloor \frac{|U|}{m} \rfloor, \lceil \frac{|U|}{m} \rceil\right\}$ gilt für jedes i.

- Die Hashfunktion $\mathbf{h(x) = (x \bmod m)}$ erfüllt die Streubedingung.
- Die Wahrscheinlichkeit $\mathbf{p_i}$, dass ein zufällig gezogener Schlüssel auf die Zelle i gehasht wird, ist höchstens

$$\mathbf{p_i \leq \frac{\lceil \frac{|U|}{m} \rceil}{|U|} \leq \frac{\frac{|U|}{m} + 1}{|U|} = \frac{1}{m} + \frac{1}{|U|}}.$$

Die erwartete Länge der Liste für Zelle i ist $\mathbf{p_i \cdot n}$.

Die erwartete Laufzeit (2/2)

Die erwartete Länge L einer *beliebigen* Liste ist

$$
\begin{aligned}
L &= \sum_{i=0}^{m-1} \frac{\text{erwartete Länge der Liste von Zelle } i}{m} \\
&= \sum_{i=0}^{m-1} \frac{p_i \cdot n}{m} = \frac{n}{m} \cdot \sum_{i=0}^{m-1} p_i = n/m = \lambda.
\end{aligned}
$$

Hashing mit Verkettung: Zusammenfassung

- Die erwartete Länge einer Liste für Hashing mit Verkettung stimmt mit dem Auslastungsfaktor λ überein.
- Die erwartete Laufzeit einer insert-, remove- oder lookup-Operation ist höchstens $O(1) + \lambda$

    Werte die Hashfunktion aus und durchlaufe die Liste.

+ Hashing mit Verkettung ist ein hochgradig praxis-taugliches Verfahren.
- Aber, durch die Verwendung von Listen, und damit durch die Verwendung von Zeigern, ersteht zusätzlicher Speicherbedarf.

Wie schnell ist Hashing mit offener Adressierung?

Die erwartete Laufzeit einer erfolglosen Suche (1/2)

Die Annahmen:

- Jeder Schlüssel $x \in U$ tritt mit Wahrscheinlichkeit $\frac{1}{|U|}$ als Operand einer Operation auf.
- Für jedes $x \in U$ ist die Folge

$$(h_0(x), h_1(x), ..., h_{m-1}(x)) = \pi_x$$

  eine Permutation von $\{0, 1, ..., m-1\}$ und
- jede Permutation π_x tritt für $\frac{|U|}{m!}$ Schlüssel $x \in U$ auf.

Wie lange müssen wir auf einen Erfolg, eine freie Zelle, warten?

Die erwartete Laufzeit einer erfolglosen Suche (2/2)

Man stelle sich vor, dass wir einen Schlüssel zufällig ziehen. Nach der Annahme ist jede Permutation getesteter Zellen gleichwahrscheinlich.

- Der Auslastungsfaktor ist λ.
- Die Wahrscheinlichkeit im 1. Versuch eine freie Zelle zu finden ist $1 - \lambda$ und steigt sogar in nachfolgenden Versuchen an, da bereits getestete aber besetzte Zellen nicht mehr getestet werden.
- Wie lange müssen wir auf einen Erfolg warten, wenn die Erfolgswahrscheinlichkeit eines einzigen Versuchs mindestens $p = 1 - \lambda$ ist?
  - Mit Wahrscheinlichkeit höchstens $(1-p)^k \cdot p$ werden genau $k+1$ Versuche benötigt.
  - Die erwartete Zeit bis zum ersten Erfolg beträgt höchstens
  $$\sum_{k=0}^{\infty}(k+1) \cdot (1-p)^k \cdot p = \frac{1}{p}.$$

$p = 1 - \lambda \Rightarrow$ Die erwartete Anzahl getesteter Zellen ist $\frac{1}{p} = \frac{1}{1-\lambda}$.

Hashing mit offener Adressierung: Zusammenfassung

Der Auslastungsfaktor sei λ.

Zur Erinnerung: **Hashing mit Verkettung** besitzt für alle Operationen eine erwartete Laufzeit von höchstens $O(1) + \lambda$.

- Wegen der Klumpenbildung des **linearen Austestens** werden im Durchschnitt $\frac{1}{2} \cdot (1 + \frac{1}{(1-\lambda)^2})$ Zellen getestet. Allerdings ist lineares Austesten „cache-freundlich".
- Die erwartete Laufzeit einer erfolglosen Suche für **doppeltes Hashing** ist höchstens $\frac{1}{1-\lambda}$.
  - Der Auslastungsfaktor für das lineare Austesten oder das doppelte Hashing sollte nicht zu groß werden:
  - Lade in eine doppelt so große Tabelle um, wenn $\lambda > 1/2$.

Universelles Hashing

Universelles Hashing

- Für jede Hashfunktion, ob für

  Hashing mit Verkettung oder Hashing mit offener Addressierung

  kann eine worst-case Laufzeit von $\Theta(n)$ erzwungen werden.

  Wir müssen also „Glück“ haben, dass unsere Operationen kein worst-case Verhalten zeigen.
- Stattdessen arbeiten wir mit einer Klasse H von Hashfunktionen:
  - Zu Beginn wählen wir zufällig eine Hashfunktion $h \in H$ und
  - führen Hashing mit Verkettung mit der Hashfunktion h durch.

- Warum „sollte“ ein solches Verfahren funktionieren?
  - Eine einzelne Hashfunktion ist durch eine bösartig gewählte Operationenfolge zum Scheitern verurteilt,
  - aber die meisten Hashfunktion werden diese Operationenfolge mit Bravour meistern.
- Was ist eine geeignete Klasse H?

c-universelle Klassen von Hashfunktionen

Eine Menge $H \subseteq \{h \mid h : U \to \{0, ..., m-1\}\}$ ist c-universell, falls

$$\frac{|\{h \in H \mid h(x) = h(y)\}|}{|H|} \leq \frac{c}{m}$$

für alle $x, y \in U$ mit $x \neq y$ gilt.

Wenn H c-universell ist, dann gibt es keine zwei Schlüssel, die mit Wahrscheinlichkeit größer als $\frac{c}{m}$ auf die gleiche Zelle hashen.

- Gibt es c-universelle Klassen von Hashfunktionen für kleine Werte von c und
- können wir dann **jede** Folge von lookup-, insert- und remove-Operationen **hochwahrscheinlich** schnell ausführen?

Eine c-universelle Klasse

Sei $U = \{0, 1, 2, ..., p - 1\}$ für eine Primzahl p.

(a) Dann ist

$$H = \{h_{a,b} \mid 0 \leq a, b < p, \;\; h_{a,b}(x) = ((ax + b) \mod p) \mod m\}$$

c-universell mit $c = (\lceil \frac{p}{m} \rceil \,/\, \frac{p}{m})^2$.

(b) **Jede** Folge von n Operationen benötigt für eine c-universelle Klasse die erwartete Zeit höchstens

$$n\left(1 + \frac{c}{2} \cdot \frac{n}{m}\right).$$

Unser **Motto**:

- „Erst durchschütteln" ($x \mapsto y = (ax + b) \mod p$) und
- dann „hashen" ($y \mapsto y \mod m$).

Wörterbücher: Wann welche Datenstruktur?

Datenstrukturen für Wörterbücher

1. **Listen**:
   - \- Die Lookup-Operation dauert viel zu lange!
   - \+ Wichtige Einsatzgebiete sind z.B. „Adjazenzlisten für Graphen“.
   - \+ Passen sich ideal der Größe der Datenmenge an wie etwa im Fall der „Darstellung dünnbesetzter Matrizen“.
2. **Binäre Suchbäume**:
   - \+ Gute erwartete Laufzeit.
   - \+ Ermöglicht die Binärsuche und ist „Ausgangspunkt“ für AVL-Bäume.
   - \- Schlechte worst-case Laufzeit und relativ viel Speicherplatz.
3. **AVL-Bäume**:
   - \+ Die worst-case Laufzeit ist logarithmisch.
   - \- Relativ viel Speicherplatz notwendig für Zeiger und Balance-Information.

Datenstrukturen für Wörterbücher: Die Champions

1. **Hashing mit Verkettung**:
   - \+ hat die sehr schnelle erwartete Laufzeit $O(1) + \lambda$,
   - \- aber verlangt relativ viel Speicher.
   - +/- Die worst-case Laufzeit ist schlecht, aber gutes Verhalten in praktischen Anwendungen.
   - \+ Universelles Hashing: Schnelle erwartete Laufzeit für **jede** Folge von Operationen!
2. **Hashing mit offener Adressierung**:
   - \+ ist mit erwarteter Laufzeit $O(1/(1-\lambda))$ etwas langsamer als Hashing mit Verkettung,
   - \- aber der Auslastungsfaktor λ muss klein sein!
   - +/- Sehr „speicherplatz-freundlich“, mit schlechter worst-case Laufzeit, aber guter Leistung für kleine λ.
3. **(a,b)-Bäume**:
   - \+ Unschlagbar in Anwendungen für langsame Speicher,
   - \- werden aber von Hashing und AVL-Bäumen „geschlagen“, wenn die Daten in einen schnellen Speicher passen.