III. Datenstrukturen

- Dynamische Mengen **Einf. Abschnitt III**
- Binäre Suchbäume, Realisierung durch Rot-Schwarz-Bäume **13**
- B-Bäume **18**
- Hashtabellen. Kollisionauflösung: Verkettung, offene Adressierung **11**

Dynamische Mengen

Eine **dynamische Menge** ist eine Datenstruktur, die Objekte verwaltet, welche einen Schlüssel tragen, und zumindest die folgenden Operationen unterstützt:

- $\text{SEARCH}(S, k)$: liefert (einen Zeiger auf) ein Element in S mit Schlüssel k , falls ein solches existiert; NIL sonst.
- $\text{INSERT}(S, x)$: fügt das Element (bezeichnet durch Zeiger) x in die Menge S ein.

Oft werden weitere Operationen unterstützt, wie etwa

- $\text{DELETE}(S, x)$: löscht das Element (bezeichnet durch Zeiger) x .
- Maximum, Minimum, Sortieren, etc. bei geordneten Schlüsseln.

Typische Anwendung: Symboltabelle in einem Compiler. Schlüssel = Bezeichner, Objekte = (Typ, Adresse, Größe, ...)

Realisierungen durch Bäume

Dynamische Mengen können durch **binäre Suchbäume** realisiert werden.

Notation für binäre Bäume:

- Wurzel des Baumes T gespeichert in $root[T]$.
- Jeder Knoten x hat Zeiger auf die Söhne $left[x]$ und $right[x]$, sowie auf den Vater $p[x]$ (ggf. NIL).

Die Operationen

- TREE-SEARCH(T, k)
- TREE-INSERT(T, x)
- TREE-DELETE(T, x)
- TREE-MINIMUM(T) und TREE-MAXIMUM(T)
- TREE-SUCCESSOR(T, x) und TREE-PREDECESSOR(T, x)

haben sämtlich eine Laufzeit von $O(h)$, wobei h die Tiefe des Baumes T ist.

Balancierte binäre Suchbäume

Damit die Operationen auf binären Suchbäumen effizient sind, sollten diese **balanciert** sein, d.h. die Höhe eines Baumes mit n Knoten ist $O(\log n)$.

Bekanntes Beispiel balancierter Suchbäume: **AVL-Bäume**

Balancieren erfolgt durch **Rotationsoperationen**:

Operation $\text{ROTATE-RIGHT}(T, x)$

Dual dazu ($\leftarrow$) wirkt $\text{ROTATE-LEFT}(T, y)$.

Rot-Schwarz-Bäume

Rot-Schwarz-Bäume: binäre Suchbäume mit zusätzlicher Struktur, die dafür sorgt, dass diese balanciert sind, also die Höhe höchstens $O(\log n)$ ist.

Jeder Knoten wird als innerer Knoten aufgefasst, Blätter sind zusätzliche, leere Knoten (**Wächter**).

Knoten haben ein zusätzliches Feld $color[x]$, mit den Werten RED und BLACK. Wir sprechen von **roten** und **schwarzen** Knoten.

Rot-Schwarz-Eigenschaft:

1. Die Wurzel und alle Blätter sind schwarz.
2. Beide Söhne eines roten Knotens sind schwarz.
3. Für jeden Knoten x gilt:
jeder Pfad von x zu einem Blatt enthält gleich viele schwarze Knoten.

Eigenschaften von Rot-Schwarz-Bäumen

Wir zeigen, dass Rot-Schwarz-Bäume balanciert sind:

Satz:

Die Höhe eines Rot-Schwarz-Baums mit n inneren Knoten ist höchstens $2 \log(n + 1)$.

Daraus folgt:

Die Operationen TREE-SEARCH, TREE-MINIMUM, TREE-MAXIMUM, TREE-SUCCESSOR und TREE-PREDECESSOR benötigen für Rot-Schwarz-Bäume $O(\log n)$ Operationen.

Auch TREE-INSERT und TREE-DELETE laufen auf Rot-Schwarz-Bäumen in Zeit $O(\log n)$, erhalten aber nicht die Rot-Schwarz-Eigenschaft.

~> Für diese benötigen wir spezielle Einfüge- und Löschooperationen.

Deutung von Rot-Schwarz-Bäumen als 2-3-4 Bäume

Man kann einen schwarzen Knoten gedanklich mit den unmittelbar unter ihm liegenden roten Knoten zusammenfassen und erhält dadurch “Makroknoten” mit 2, 3, oder 4 Kindern:

Siehe rotschw.ps

Einfügen in einen Rot-Schwarz-Baum

Intuition: Suche (mit TREE-INSERT den Makroknoten, in den der neue Knoten “gehört”. Dieser hat keine echten Kinder. Ist in diesem Makroknoten noch Platz, so füge den neuen Knoten dort ein. Enthält er bereits drei Einträge (voll), so teile ihn in zwei neue Makroknoten auf. Führt das zu Überlauf im Eltern(makro)knoten, so fahre entsprechend fort.

Einfügen in einen Rot-Schwarz-Baum

Konkrete Realisierung:

Füge neuen Knoten mit TREE-INSERT

Neuer Knoten x wird mit TREE-INSERT eingefügt und **rot** gefärbt.

Problem: Vater $p[x]$ kann auch rot sein $\rightsquigarrow$ Eigenschaft 2 verletzt.

Diese Inkonsistenz wird in einer Schleife durch lokale Änderungen im Baum nach oben geschoben:

- **Fall 1:** $p[x] = \text{left}[p[p[x]]]$:
 - Betrachte x ' Onkel $y = \text{right}[p[p[x]]]$.
 - **Fall 1.1:** y ist rot. Dann färbe $p[x]$ und y schwarz, und $p[p[x]]$ rot.
Aber: $p[p[p[x]]]$ kann rot sein $\rightsquigarrow$ Inkonsistenz weiter oben.
 - **Fall 1.2:** y ist schwarz, und $x = \text{left}[p[x]]$. Dann wird $p[x]$ schwarz und $p[p[x]]$ rot gefärbt, und anschliessend um $p[p[x]]$ nach rechts rotiert.
 $\rightsquigarrow$ keine Inkonsistenz mehr.
 - **Fall 1.3:** y ist schwarz, und $x = \text{right}[p[x]]$. Wird durch eine Linksrotation

Einfügen in einen Rot-Schwarz-Baum

um $p[x]$ auf Fall 1.2 zurückgeführt.

- **Fall 2:** $p[x] = \text{right}[p[p[x]]]$ ist symmetrisch.

Einfügen in einen Rot-Schwarz-Baum

Am Ende wird die Wurzel $root[T]$ schwarz gefärbt.

Bemerkung:

- In den Fällen 1.2 und 1.3 bricht die Schleife sofort ab.
- Schleife wird nur im Fall 1.1 wiederholt, dann ist die Tiefe von x kleiner geworden.
- Rotationen werden nur in 1.2 und 1.3 durchgeführt.

$\leadsto$ Laufzeit ist $O(\log n)$, und es werden höchstens 2 Rotationen durchgeführt.

Löschen aus einem Rot-Schwarz-Baum

Intuition: Suche (mit TREE-DELETE) den Makroknoten (in Blattposition!), aus dem der Knoten zu entfernen ist. Enthält dieser 2 oder 3 Einträge, so entferne dort. Enthält er nur einen Eintrag, so versuche vom linken oder rechten Nachbarn einen Eintrag abzuziehen (geht, falls dort 2 oder 3 Einträge vorhanden sind.). Enthält auch der linke oder rechte Nachbar nur 1 Eintrag, so verschmilz mit Nachbarn zu größerem Makroknoten. Führt das zu Unterlauf im Eltern(makro)knoten, so fahre entsprechend fort.

Zu löschender Knoten z wird mit $\text{TREE-DELETE}(T, z)$ gelöscht.

Erinnerung: $\text{TREE-DELETE}(T, z)$ entfernt einen Knoten y , der höchstens einen (inneren) Sohn hat.

- Hat z höchstens einen Sohn, so ist $y = z$.
- Andernfalls $y \leftarrow \text{TREE-SUCCESSOR}(T, z)$.

Problem: Wenn der entfernte Knoten y schwarz war, ist Eigenschaft 3 verletzt.

Intuition: der (einzige) Sohn x von y erbt dessen schwarze Farbe, und ist jetzt “doppelt schwarz”.

Löschen aus einem Rot-Schwarz-Baum

Das zusätzliche Schwarz wird in einer Schleife durch lokale Änderungen im Baum nach oben geschoben.

Löschen aus einem Rot-Schwarz-Baum

- Ist x rot, so wird es schwarz gefärbt, und die Schleife bricht ab.
Andernfalls unterscheide zwei Fälle:
- **Fall 1:** $x = \text{left}[p[x]]$.
 - Betrachte x ' Bruder $w = \text{right}[p[x]] \neq \text{NIL}$, und unterscheide 2 Fälle:
 - **Fall 1.1:** w ist rot. Dann muss $p[x]$ schwarz sein, also färbe $p[x]$ rot, w schwarz und rotiere links um $p[x]$
 $\leadsto$ reduziert auf Fall 1.2.
 - **Fall 1.2:** w ist schwarz. Es gibt drei weitere Fälle:
 - * **Fall 1.2.1:** Beide Kinder von w sind schwarz. Also können wir w rot färben, und das zusätzliche Schwarz von x zu $p[x]$ versetzen
 $\leadsto$ nächste Iteration.
 - * **Fall 1.2.2:** $\text{left}[w]$ ist rot, $\text{right}[w]$ ist schwarz. Dann färbe w rot, $\text{left}[w]$ schwarz und rotiere rechts um w
 $\leadsto$ reduziert auf Fall 1.2.3.
 - * **Fall 1.2.3:** $\text{right}[w]$ ist rot. Dann vertausche die Farben von w und $p[x]$, färbe $\text{right}[w]$ schwarz, und rotiere links um $p[x]$
 $\leadsto$ Zusätzliches Schwarz ist verbraucht.

Löschen aus einem Rot-Schwarz-Baum

- **Fall 2:** $x = \text{right}[p[x]]$ ist symmetrisch.

Auch beim Löschen wird am Ende die Wurzel $\text{root}[T]$ schwarz gefärbt.

Bemerkung:

- In den Fällen 1.2.2 und 1.2.3 bricht die Schleife sofort ab.
- Schleife wird nur im Fall 1.2.1 wiederholt, dann ist die Tiefe von x kleiner geworden.
- Im Fall 1.1 wird die Tiefe von x größer, aber die Schleife bricht danach im Fall 1.2 sofort ab.

$\leadsto$ Laufzeit ist $O(\log n)$, es werden höchstens 3 Rotationen durchgeführt.

Zusammengefasst: die Operationen SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, INSERT und DELETE können für Rot-Schwarz-Bäume mit Laufzeit $O(\log n)$ realisiert werden.

B-Bäume

B-Bäume verallgemeinern binäre Suchbäume dahingehend, dass in einem Knoten mehrere Schlüssel stehen und ein Knoten mehrere Kinder hat. (Typischerweise jeweils 500-1000).

Dadurch sinkt die Zahl der Knoten, die bei einem Suchvorgang besucht werden müssen, dafür ist aber jedes einzelne Besuchen aufwendiger.

Das ist sinnvoll, wenn die Knoten auf einem Massenspeichermedium abgelegt sind (Plattenstapel o.ä.) wo einzelne Zugriffe recht lange dauern, dafür aber gleich eine ganze *Seite* (z.B.: 1024Byte) auslesen, bzw. schreiben.

B-Bäume: Definition

Die Definition von B -Bäumen bezieht sich auf ein festes $t \in \mathbb{N}$, z.B.: $t = 512$.

Ein B -Baum ist ein Baum mit den folgenden Eigenschaften:

- Jeder Knoten x enthält die folgenden Einträge:
 - die Anzahl $n[x]$ der im Knoten gespeicherten Schlüssel, wobei $n[x] \leq 2t - 1$.
 - die in aufsteigender Reihenfolge gespeicherten Schlüssel:
 $key_1[x] \leq key_2[x], \dots, key_{n[x]}[x]$
- Jeder Knoten x ist entweder ein Blatt oder ein innerer Knoten und hat dann gerade $n[x] + 1$ Kinder.
- Ist x innerer Knoten und nicht die Wurzel, so gilt $n[x] \geq t - 1$.
- Alle Schlüssel, die sich im oder unterhalb des i -ten Kindes eines Knotens x befinden, liegen bzgl. der Ordnung zwischen $key_{i-1}[x]$ und $key_i[x]$. Grenzfall $i = 1$: die Schlüssel sind kleiner als $key_1[x]$; Grenzfall $i = n[x]$: die Schlüssel sind größer als $key_{n[x]}[x]$.

Suchen in B -Bäumen

Beim Suchen vergleicht man den gesuchten Schlüssel k mit den Einträgen der Wurzel. Entweder ist k unter den Wurzeleinträgen, oder man ermittelt durch Vergleich mit den Wurzeleinträgen denjenigen Unterbaum, in welchem k , wenn überhaupt, zu finden ist und sucht in diesem rekursiv weiter.

Natürliche Verallgemeinerung des Suchens in BST.

Einfügen in B -Bäumen

1. Bestimme durch Suche das Blatt x , in welches der neue Schlüssel k aufgrund seiner Ordnungsposition gehört.
2. Falls dort noch nicht zuviele Einträge (also $n[x] < 2t - 1$) vorhanden sind, so füge den neuen Schlüssel dort ein.
3. Andernfalls füge k trotzdem in das Blatt ein (es enthält dann $2t$ Schlüssel) und teile es in zwei Blätter der Größe t und $t - 1$ auf. Der übrige Schlüssel wird dem Elternknoten als Trenner der beiden neuen Kinder hinzugefügt.
4. Führt dies zum Überlauf im Elternknoten, so teile man diesen ebenso auf, etc. bis ggf. zur Wurzel.

Löschen aus B -Bäumen

1. Zunächst ersetzt man ähnlich wie bei BST den zu löschenden Schlüssel durch einen passenden Schlüssel aus einem Blatt. Dadurch kann man sich auf das Entfernen von Schlüsseln aus Blättern beschränken.
2. Kommt es beim Entfernen eines Schlüssels aus einem Blatt b zum Unterlauf (weniger als $t - 1$ Schlüssel), hat aber der rechte Nachbar b' mehr als $t - 1$ Schlüssel, so gibt man den Schlüssel aus dem Elternknoten, der b von b' trennt, zu b hinzu und nimmt den kleinsten Schlüssel in b' als neuen Trenner.
3. Hat der rechte Nachbar b' von b auch nur $t - 1$ Schlüssel, so verschmilzt man b und b' samt dem Trenner im Elternknoten zu einem neuen Knoten mit nunmehr $t - 2 + t - 1 + 1 = 2t - 2$ Schlüsseln. Dem Elternknoten geht dadurch ein Schlüssel verloren, was wiederum zum Unterlauf führen kann. Ein solcher ist durch rekursive Behandlung nach demselben Schema zu beheben.

Mögliche Optimierung

Cormen und auch die Wikipedia empfehlen, beim Einfügen bereits beim Aufsuchen der entsprechenden Blattposition solche Knoten, die das Höchstmaß $2t - 1$ erreicht haben, vorsorglich zu teilen, damit nachher ein Überlauf lokal behandelt werden kann und nicht wieder rekursiv nach oben gereicht werden muss.

Analog würde man beim Löschen Knoten der Mindestgröße $t - 1$ durch Ausgleichen vom Nachbarn oder, falls das nicht möglich, Verschmelzen, vorsorglich verdicken.

Man nimmt hierdurch mehr Spaltungen / Verschmelzungen vor, als unbedingt nötig, spart sich dafür aber Knotenzugriffe. Was im Einzelfall besser ist, ist unklar.

Dynamische Mengen als Hashtabelle

- Direkte Adressierung
- Hashing und Hashfunktionen
- Kollisionauflösung durch Verkettung
- Offene Adressierung
- Analyse der erwarteten Laufzeiten

In diesem Abschnitt werden Arrayfächer beginnend mit 0 nummeriert.

Direkte Adressierung

- Sind die Schlüssel ganze Zahlen im Bereich $0 \dots N - 1$, so kann eine dynamische Menge durch ein **Array A der Größe N** implementiert werden.
- Der Eintrag $A[k]$ ist x falls ein Element x mit Schlüssel k eingetragen wurde.
- Der Eintrag $A[k]$ ist **NIL**, falls die dynamische Menge kein Element mit Schlüssel k enthält.
- Die Operationen SEARCH, INSERT, DELETE werden unterstützt und haben Laufzeit $\Theta(1)$.
- Nachteile: **Enormer Speicherplatz** bei großem N . Nicht möglich, falls keine obere Schranke an Schlüssel vorliegt.

Hash-Tabelle

- Sei U die Menge der Schlüssel, z.B.: $U = \mathbb{N}$.
- Gegeben eine Funktion $h : U \rightarrow \{0, 1, 2, 3, \dots, m - 1\}$, die „Hashfunktion“.
- Die dynamische Menge wird implementiert durch ein Array der Größe m .
- Das Element mit Schlüssel k wird an der Stelle $A[h(k)]$ abgespeichert.

SEARCH(A, k)

1 **return** $A[h(k)]$

INSERT(A, x)

1 $A[h(\text{key}[x])] \leftarrow x$

DELETE(A, k)

1 $A[h(k)] \leftarrow \text{NIL}$

- Zum Beispiel: $U = \mathbb{N}$, $h(k) = k \bmod m$.
- **Problem:** $h(k_1) = h(k_2)$ obwohl $k_1 \neq k_2$ (**Kollision**). Kommt es zu Kollisionen so ist dieses Verfahren **inkorrekt** (also **gar kein Verfahren!**).

Häufigkeit von Kollisionen

- Alle m Hashwerte seien gleichwahrscheinlich.
- Die Wahrscheinlichkeit, dass k zufällig gewählte Schlüssel **paarweise verschiedene Hashwerte** haben ist dann:

$$\begin{aligned} & 1 \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \dots \frac{m-k+1}{m} = \prod_{i=0}^{k-1} \left(1 - \frac{i}{m}\right) \\ & \leq \prod_{i=0}^{k-1} e^{-i/m} \\ & = e^{-\sum_{i=0}^{k-1} i/m} = e^{-k(k-1)/2m} \end{aligned}$$

- Diese Wahrscheinlichkeit wird kleiner als 50% wenn $k \geq 1 + \frac{1}{2}\sqrt{1 + 8m \ln 2}$.
- Beispiel $m = 365$, $h(k)$ = „Geburtstag von k “. Bei mehr als 23 Menschen ist es wahrscheinlicher, dass zwei **am selben Tag Geburtstag** haben, als umgekehrt.
- Kollisionen sind **häufiger als man denkt**.

Kollisionsauflösung durch Verkettung

Um Kollisionen zu begegnen, hält man in jeder Arrayposition eine **verkettete Liste von Objekten**.

- **Suchen** geschieht durch Suchen in der jeweiligen Liste,
- **Einfügen** geschieht durch Anhängen an die jeweilige Liste,
- **Löschen** geschieht durch Entfernen aus der jeweiligen Liste.

SEARCH(A, k)

1 Suche in der Liste $A[h(k)]$ nach Element mit Schlüssel k

INSERT(A, x)

1 Hänge x am Anfang der Liste $A[h(k)]$ ein.

DELETE(A, k)

1 Entferne das Objekt mit Schlüssel k aus der Liste $A[h(k)]$.

Leider ist die Laufzeit jetzt **nicht mehr** $\Theta(1)$.

Lastfaktor

Die Hashtabelle habe m Plätze und enthalte n Einträge.

Der Quotient $\alpha := n/m$ heißt **Lastfaktor**.

Beachte: $\alpha > 1$ ist möglich.

Der Lastfaktor heißt auch **Belegungsfaktor**

Eine Hashtabelle heißt auch **Streuspeichertabelle**.

Analyse von Hashing mit Verkettung

Die Hashwerte seien wiederum uniform verteilt.

Dann werden die Listen im Mittel Länge α besitzen.

Die Operationen SEARCH, INSERT, DELETE haben also jeweils **erwartete** (=mittlere) Laufzeit

$$T \leq c(1 + \alpha)$$

für geeignete Konstante $c > 0$.

Der Summand „1“ bezeichnet den Aufwand für das Berechnen der Hashfunktion und die Indizierung.

Der Summand α bezeichnet die lineare Laufzeit des Durchsuchens einer verketteten Liste.

Hashfunktionen

Seien die einzutragenden Objekte x irgendwie zufällig verteilt.

Die Hashfunktion sollte so beschaffen sein, dass die Zufallsvariable $h(\text{key}[x])$ **uniform verteilt** ist (da ja sonst manche Fächer leer bleiben, während andere überfüllt sind.)

Sind z.B. die Schlüssel in $\{0, \dots, N - 1\}$ uniform verteilt, so ist $h(k) = k \bmod m$ eine gute Hashfunktion.

Sind z.B. die Schlüssel in $[0, 1[$ uniform verteilt, so ist $h(k) = \lfloor mk \rfloor$ eine gute Hashfunktion.

m wie immer die Größe der Hashtabelle.

Die Schlüssel sind meist **nicht uniform** verteilt:

Bezeichner in einer Programmiersprache: `count`, `i`, `max_zahl` häufiger als `zu6fgp98qq`. Wenn `kli` dann oft auch `kli1`, `kli2`, etc.

Nichtnumerische Schlüssel

...müssen vor Anwendung einer „Lehrbuch-Hashfunktion“ zunächst in Zahlen konvertiert werden.

Zeichenketten etwa unter Verwendung der Basis 256:

'p' = 112, 'q' = 116, also "pq" = $112 \cdot 256 + 116 = 28788$.

Divisionsmethode

Wie immer: Schlüssel: $0 \dots N - 1$, Hashwerte: $0 \dots m - 1$.

Hashfunktion: $h(k) = k \bmod m$.

- m sollte keine Zweierpotenz sein, da sonst $h(k)$ nicht von allen Bits (von k) abhängt.
- Ist k eine Kodierung eines Strings im 256er System, so bildet h bei $m = 2^p - 1$ zwei Strings, die sich nur durch eine Transposition unterscheiden, auf denselben Wert ab.
- Eine gute Wahl für m ist eine Primzahl die nicht nahe bei einer Zweierpotenz liegt. Z.B. $n = 2000$, vorauss. Lastfaktor $\alpha = 3$: Tabellengröße $m = 701$ bietet sich an.
- Bei professionellen Anwendungen empfiehlt sich ein Test mit „realen Daten“.

Multiplikationsmethode

Hashfunktion: $h(k) = \lfloor m(kA \bmod 1) \rfloor$ für $A \in]0, 1[$.

Hier $x \bmod 1$ = „gebrochener Teil von x “, z.B.: $\pi \bmod 1 = 0,14159\dots$

Rationale Zahlen A mit kleinem Nenner führen zu Ungleichverteilungen, daher empfiehlt sich die Wahl $A = (\sqrt{5} - 1)/2$ („**Goldener Schnitt**“)^a.

Vorteile der Multiplikationsmethode:

- Arithmetische Progressionen von Schlüssel $k = k_0, k_0 + d, k_0 + 2d, k_0 + 3d, \dots$ werden ebenmäßig verstreut.
- Leicht zu implementieren wenn $m = 2^p$ (hier unproblematisch) und $N < 2^w$, wobei w die Wortlänge ist: Multipliziere k mit $\lfloor A \cdot 2^w \rfloor$. Dies ergibt zwei w -bit Wörter. Vom Niederwertigen der beiden Wörter bilden die p höchstwertigen Bits den Hashwert $h(k)$.

^aDiese Zahl ist die „irrationalste“ überhaupt, da der Nenner als Funktion der Approximationsgüte bei dieser Zahl am stärksten wächst.

Weiterführendes

- **Universelles Hashing**: Zufällige Wahl der Hashfunktion bei Initialisierung der Tabelle, dadurch Vermeidung systematischer Kollisionen, z.B. Provokation schlechter Laufzeit durch böartig konstruierte Benchmarks.
- Gute Hashfunktionen können zur Authentizierung verwendet werden, z.B., MD5 *message digest*.

Offene Adressierung

Man kann auf verkettete Listen verzichten, indem man bei Auftreten einer Kollision eine andere Arrayposition benutzt.

Dazu braucht man eine zweistellige Hashfunktion

$$h : U \times \{0, \dots, m - 1\} \rightarrow \{0, \dots, m - 1\}.$$

INSERT(T, x)

1 $i \leftarrow 0$

2 **while** $i \leq m$ and $h(\text{key}[x], i) \neq \text{NIL}$ **do**

3 $i \leftarrow i + 1$

4 **if** $i \leq m$

5 **then** $T[h(\text{key}[x], i)] = x$

6 **else error** “hash table overflow“

Für jeden Schlüssel k sollte die **Sondierungsfolge** (*probe sequence*)

$$h(k, 0), h(k, 1), h(k, 2), \dots, h(k, m - 1)$$

eine Permutation von $0, 1, 2, \dots, m - 1$ sein, damit jede Position irgendwann sondiert wird.

Offene Adressierung

```
SEARCH( $T, k$ )
1   $i \leftarrow 0$ 
2  repeat
3       $j \leftarrow h(k, i); i \leftarrow i + 1$ 
4  until  $i = m$  or  $T[j] = \text{NIL}$  or  $\text{key}[T[j]] = k$ 
5  if  $i < m$  and  $\text{key}[T[j]] = k$ 
6      then return  $T[j]$ 
7      else return NIL
```

NB: Tabelleneinträge sind Objekte **zuzüglich** des speziellen Wertes NIL.

Z.B. Zeiger auf Objekte oder **Nullzeiger**.

Einschränkung: Bei offener Adressierung ist Löschen etwas umständlich (etwa durch explizites Markieren von Einträgen als “gelöscht”).

Hashfunktionen für offene Adressierung

- **Lineares Sondieren** (*linear probing*):

$$h(k, i) = (h'(k) + i) \bmod m$$

Problem: Lange zusammenhängende Blöcke besetzter Plätze entstehen (*primary clustering*), dadurch oft lange Sondierdauer.

- **Quadratisches Sondieren** (*quadratic probing*):

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m$$

für passende c_1, c_2 sodass $h(k, \cdot)$ Permutation ist.

Quadratisches Sondieren ist wesentlich besser als lineares Sondieren, aber beide Verfahren haben **folgenden Nachteil**:

Wenn $h(k_1, 0) = h(k_2, 0)$, dann $h(k_1, i) = h(k_2, i)$, d.h., kollidierende Schlüssel haben dieselbe Sondierungsfolge.

Insgesamt gibt es nur m verschiedene Sondierungsfolgen (von $m!$ Möglichen!); das führt auch zu Clusterbildung (*secondary clustering*).

Hashfunktionen für offene Adressierung

- Double hashing

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

Jede Sondierungsfolge ist eine **arithmetische Progression**, Startwert und Schrittweite sind durch Hashfunktionen bestimmt.

Damit alle Positionen sondiert werden, muss natürlich $\text{ggT}(h_2(k), m) = 1$ sein. Z.B. m Zweierpotenz und $h_2(k)$ immer ungerade.

Es gibt dann $\Theta(m^2)$ Sondierungsfolgen.

Analyse der Offenen Adressierung

Vereinfachende Annahme: Die Schlüssel seien so verteilt, dass jede der $m!$ Sondierungsfolgen **gleichwahrscheinlich** ist.

Diese Annahme wird durch *double hashing* approximiert aber nicht erreicht.

Satz: In einer offen adressierten Hashtabelle mit Lastfaktor $\alpha = n/m < 1$ ist die zu erwartende Dauer einer erfolglosen Suche beschränkt durch $1/(1 - \alpha)$.

Beispiel: Lastfaktor $\alpha = 0,9$ (Tabelle zu neunzig Prozent gefüllt): Eine erfolglose Suche erfordert im Mittel weniger als 10 Versuche (unabhängig von m, n).

Bemerkung: Dies ist auch die erwartete Laufzeit für eine Insertion.

Beweis des Satzes

Sei X eine Zufallsvariable mit Werten aus $\mathbb{N}$.

Dann ist

$$E[X] := \sum_{i=0}^{\infty} i \Pr\{X = i\} = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$$

Dies deshalb, weil $\Pr\{X \geq i\} = \sum_{j=i}^{\infty} \Pr\{X = j\}$.

Daher ergibt sich für die erwartete Suchdauer D :

$$D = \sum_{i=1}^{\infty} \Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\}$$

$$\Pr\{\text{„Mehr als } i \text{ Versuche finden statt“}\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \leq \alpha^i$$

Also, $D \leq \sum_{i=1}^{\infty} \alpha^i = 1/(1 - \alpha)$.

Analyse der Offenen Adressierung

Satz: In einer offen adressierten Hashtabelle mit Lastfaktor $\alpha = n/m < 1$ ist die zu erwartende Dauer einer erfolgreichen Suche beschränkt durch $(1 - \ln(1 - \alpha))/\alpha$.

Beispiel: Lastfaktor $\alpha = 0,9$: Eine erfolgreiche Suche erfordert im Mittel weniger als 3,67 Versuche (unabhängig von m, n).

Lastfaktor $\alpha = 0,5$: mittlere Suchdauer $\leq 3,39$.

Achtung: All das gilt natürlich nur unter der **idealisierenden** Annahme von *uniform hashing*.

Beweis

Die beim Aufsuchen des Schlüssels durchlaufene Sondierungsfolge ist dieselbe wie die beim Einfügen durchlaufene.

Die Länge dieser Folge für den als $i + 1$ -ter eingefügten Schlüssel ist im Mittel beschränkt durch $1/(1 - i/m) = m/(m - i)$. (Wg. vorherigen Satzes!)

Gemittelt über alle Schlüssel, die eingefügt wurden, erhält man also

$$\frac{1}{n} \sum_{i=0}^{n-1} \frac{m}{m-i} = \frac{1}{\alpha} (H_m - H_{m-n})$$

als obere Schranke für den Erwartungswert der Suchdauer.

Hier ist $H_k = \sum_{i=1}^k 1/i$ die **k -te harmonische Zahl**. Es gilt mit Integralabschätzung $\ln k \leq H_k \leq 1 + \ln k$.

Daraus ergibt sich der Satz.

Zusammenfassung

- Hashing = Speichern von Objekten an Arraypositionen, die aus ihrem Schlüssel **berechnet** werden.
- Die Zahl der Arraypositionen ist i.a. **wesentlich kleiner** als die der Schlüssel.
- Kollisionauflösung durch Verkettung: Jede Arrayposition enthält eine verkettete Liste.
- Offene Adressierung: Bei Kollision wird eine andere Arrayposition sondiert.
- Hashfunktionen für einfaches Hashing: Multiplikationsmethode, Divisionsmethode.
- Hashfunktionen für **offene Adressierung**: Lineares, quadratisches Sondieren. Double Hashing.
- Analyse unter Uniformitätsannahme, Komplexität jeweils als Funktion des **Lastfaktors** α (Auslastungsgrad):
 - Suchen und Einfügen bei einfachem Hashing: $c(1 + \alpha)$
 - Einfügen und erfolgloses Suchen bei offener Adressierung: $\leq 1/(1 - \alpha)$ Versuche
 - Erfolgreiches Suchen bei offener Adressierung: $\leq (1 - \ln(1 - \alpha))/\alpha$ Versuche