Kapitel 4: Suchen

- Einfache Suchverfahren
  - Lineare Suche, Binäre Suche, Interpolationssuche
- Suchbäume
  - Binäre Suchbäume, AVL- Bäume, Splay-Bäume, B-Baum, R-Baum
- Hashing

Suche in konstanter Zeit

- Bisher: Statt lineare Suche erlauben Bäume für viele Anwendungen durch geeignete Strukturierung, den Suchaufwand auf $O(\log n)$ zu reduzieren.
- Einfügen, Löschen und Zugriff in $O(\log n)$.
- Wenn wir den Suchraum noch geschickter strukturieren, können wir dann noch schneller suchen?

Bitvektor-Darstellung für Mengen

- Geeignet für kleine Universen U
  $N = |U|$ vorgegebene maximale Anzahl von Elementen
  $S \subseteq U = \{0,1,\dots,N-1\}$
  Suche hier nur als „Ist-Enthalten“-Test

  Verwende Schlüssel ‚i‘ als Index im Bitvektor (= Array von Bits)
  Bitvektor: Bit[i] = 1 wenn $i \in S$
  Bit[i] = 0 wenn $i \notin S$
- Beispiel: $N = \{0,1,2,3,4\}$, $M_1 = \{0,2,3\}$, $M_2 = \{0,1\}$

$$Bit(M_1) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}, \qquad Bit(M_2) = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Bitvektor-Darstellung: Komplexität

- Operationen
  - Insert, Delete $O(1)$ setze/lösche entsprechendes Bit
  - Search $O(1)$ teste entsprechendes Bit
  - Initialize $O(N)$ setze ALLE Bits des Arrays auf 0
- Speicherbedarf
  - Anzahl Bits $O(N)$ maximale Anzahl Elemente
- Problem bei Bitvektor
  - Initialisierung kostet $O(N)$
  - Verbesserung durch spezielle Array-Implementierung
  - Ziel: Initialisierung $O(1)$

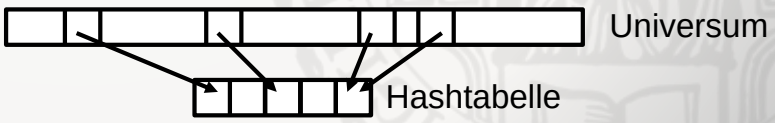
Hashing

- Ziel:
  Zeitkomplexität Suche $O(1)$ - wie bei Bitvektor-Darstellung
  Initialisierung $O(1)$

- Ausgangspunkt
  Bei Bitvektor-Darstellung wird der Schlüsselwert direkt als Index in einem Array verwendet

- Grundidee
  Oft hat man ein sehr großes Universum (z.B. Strings)
  Aber nur eine kleine Objektmenge (z.B. Straßennamen einer Stadt)
  Für die ein kleines Array ausreichend würde

- Idee
  Bilde verschiedene Schlüssel auf dieselben Indexwerte ab.
  Dadurch Kollisionen möglich



Hashing

- Grundbegriffe:
  - U ist das Universum aller Schlüssel
  - $S \subseteq U$ die Menge der zu speichernden Schlüssel mit $n = |S|$
  - T die Hash-Tabelle der Größe m

- Hashfunktion h:
  - Berechnung des Indexwertes zu einem Schlüsselwert x
  - Schlüsseltransformation: $h: U \rightarrow \{0, \ldots, m-1\}$
  - $h(x)$ ist der Hash-Wert von x

- Hashing wird angewendet wenn:
  - $|U|$ sehr groß ist
  - $|S| \ll |U|$ - Anzahl der zu speichernden Elemente ist viel kleiner als die Größe des Universums

Anwendung von Hashing als Prüfziffer

- IBAN (International Bank Account Number): Aufbau einer deutschen IBAN

D	E	x	x	b	b	b	b	b	b	b	b	k	k	k	k	k	k	k	k	k	k
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

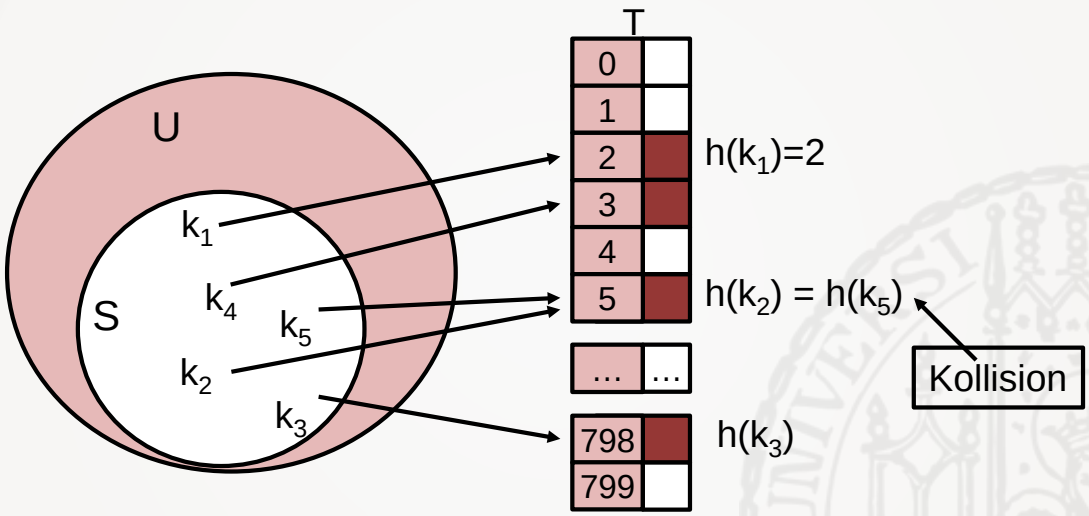
Land | Prüf-ziffern | Bankleitzahl | Kontonummer

Ländercode beachten: $A \rightarrow 10, B \rightarrow 11, \ldots$

- Berechnung der Prüfziffer:
  $$xx = 98 - bbbbbbbbkkkkkkkkkk131400 \; mod \; 97$$
- Universum: 10^{18} Bank-/Kontonummern (theoretisch) möglich
- Hashwerte: $02 \leq xx \leq 98$
- Durch die schnelle Berechnung können viele Fehler bereits bei Eingabe gemeldet werden (z.B. Zahlendreher)

Hashing-Prinzip

- Grafische Darstellung - Beispiel: Studenten



- Gesucht:
  - Hashfunktion, welche die Matrikelnummern möglichst gleichmäßig auf die 800 Einträge der Hash-Tabelle abbildet

Hashfunktion

- Dient zur Abbildung auf eine Hash-Tabelle
  - Hash-Tabelle **T** hat m Plätze (Slots, Buckets)
    - In der Regel $m \ll |U|$ daher Kollisionen möglich
  - Speichern von $|S| = n$ Elementen ($n < m$)
  - Belegungsfaktor $\alpha = n/m$

- Anforderung an eine Hashfunktion
  - $h: domain(K) \rightarrow \{0, 1, \ldots, m-1\}$ soll surjektiv sein.
  - $h(K)$ soll effizient berechenbar sein, idealerweise in $O(1)$.
  - $h(K)$ soll die Schlüssel möglichst gleichmäßig über den Adressraum verteilen, um dadurch Kollisionen zu vermeiden (Hashing = Streuspeicherung).
  - $h(K)$ soll unabhängig von der Ordnung der K sein in dem Sinne, dass in der Domain „nahe beieinander liegende“ Schlüssel auf nicht nahe beieinander liegende Adressen abgebildet werden.

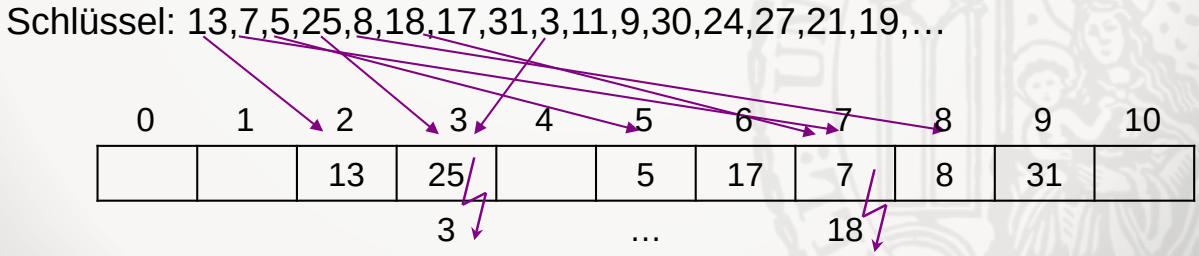
Hashfunktion: Divisionsmethode

- Hashfunktion:
  - $h(k) = K \; mod \; m$ für numerische Schlüssel
  - $h(k) = ord(K) \; mod \; m$ für nicht-numerische Schlüssel

- Konkretes Beispiel für ganzzahlige Schlüssel:

  $h: domain(K) \rightarrow \{0,1, \ldots, m-1\}$ mit $h(K) = K \; mod \; m$

- Sei m=11



Beispiel: Divisionsmethode

- Für Zeichenketten: Benutze die *ord*-Funktion zur Abbildung auf ganzzahlige Werte, z.B.
- Sei m=17: $h : \text{STRING} \mapsto \left(\sum_{i=1}^{\text{len(STRING)}} ord(\text{STRING}[i]) \right) \bmod m$

JAN	→ 25 mod 17 = 8	MAI	→ 23 mod 17 = 6	SEP	→ 40 mod 17 = 6
FEB	→ 13 mod 17 = 13	JUN	→ 45 mod 17 = 11	OKT	→ 46 mod 17 = 12
MAR	→ 32 mod 17 = 15	JUL	→ 43 mod 17 = 9	NOV	→ 51 mod 17 = 0
APR	→ 35 mod 17 = 1	AUG	→ 29 mod 17 = 12	DEZ	→ 35 mod 17 = 1

- Wie sollte m aussehen?
  - $m = 2^d$ → einfach zu berechnen
    $K \bmod 2^d$ liefert die letzten d Bits der Binärzahl K → Widerspruch zur Unabhängigkeit von K
  - m gerade → $h(K)$ gerade ⇔ K gerade → Widerspruch zur Unabhängigkeit von K
  - m Primzahl → hat sich erfahrungsgemäß bewährt

Beispiel Hashfunktion

- Einsortieren der Monatsnamen in die Symboltabelle

$$h(c) = \left(N(c_1) + N(c_2) + N(c_3)\right) \bmod 17$$

0	November
1	April, Dezember
2	März
3	
4	
5	
6	Mai, September
7	
8	Januar

9	Juli
10	
11	Juni
12	August, Oktober
13	Februar
14	
15	
16	

3 Kollisionen

Perfekte Hashfunktion

- Eine Hashfunktion ist perfekt:
  - wenn für $h : U \to \{0, \dots, m-1\}$ mit $S = \{k_1, \dots, k_n\} \subseteq U$ gilt
    $$h(k_i) = h(k_j) \Leftrightarrow i = j$$
  - also für die Menge S keine Kollisionen auftreten
- Eine Hashfunktion ist minimal:
  - wenn $m = n$ ist, also nur genau so viele Plätze wie Elemente benötigt werden
- Im Allgemeinen können perfekte Hashfunktionen nur ermittelt werden, wenn alle einzufügenden Elemente und deren Anzahl (also S) im Voraus bekannt sind (static Dictionary)
  → In der Praxis meist nicht gegeben!

Kollisionen beim Hashing

- Verteilungsverhalten von Hashfunktionen
  - Untersuchung mit Hilfe von Wahrscheinlichkeitsrechnung
  - S sei ein Ereignisraum
  - E ein Ereignis $E \subseteq S$
  - P sei eine Wahrscheinlichkeitsverteilung
- Beispiel: Gleichverteilung
  - einfache Münzwürfe: $S = \{\text{Kopf}, \text{Zahl}\}$
  - Wahrscheinlichkeit für Kopf $P(\text{Kopf}) = \frac{1}{2}$
  - n faire Münzwürfe: $S = \{\text{Kopf}, \text{Zahl}\}^n$
  - Wahrscheinlichkeit für n-mal Kopf $P(n\text{−mal Kopf}) = \left(\frac{1}{2}\right)^n$
    (Produkt der einzelnen Wahrscheinlichkeiten)

Kollisionen beim Hashing

- Analogie zum Geburtstagsproblem (-paradoxon)
  - Wie groß ist die Wahrscheinlichkeit, dass mindestens 2 von n Leuten am gleichen Tag Geburtstag haben?
  - $m = 365$ Größe der Hash-Tabelle (Tage), n = Anzahl Personen

- Eintragen des Geburtstages in die Hash-Tabelle
  - $p(i, m)$ = Wahrscheinlichkeit, dass für das i-te Element eine Kollision auftritt
  - $p(1, m) = 0$ da keine Zelle belegt
  - $p(2, m) = 1/m$ da 1 Zellen belegt
  - ...
  - $p(i, m) = (i - 1)/m$ da (i-1) Zellen belegt

Kollisionen beim Hashing

- Eintragen des Geburtstages in die Hash-Tabelle

  - Wahrscheinlichkeit für keine einzige Kollision bei n Einträgen in eine Hash-Tabelle mit m Plätzen ist das Produkt der einzelnen Wahrscheinlichkeiten

$$P(NoCol|n, m) = \prod_{i=1}^{n} \big(1 - p(i, m)\big) = \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right)$$

  - Die Wahrscheinlichkeit, dass es mindestens zu einer Kollision kommt, ist somit

$$P(Col|n, m) = 1 - P(NoCol|n, m)$$

Kollisionen beim Hashing

- Kollisionen bei Geburtstagstabelle

Anzahl Personen n	$P(Col\|n,m)$
10	0,11695
20	0,41144
…	
22	0,47570
23	0,50730
24	0,53835
…	
30	0,70632
40	0,89123
50	0,97037

- Schon bei einer Belegung von $23/365 = 6\%$ kommt es zu 50% zu mindestens einer Kollision
- Daher Strategie für Kollisionen wichtig
- Wann ist eine Hashfunktion gut?
- Wie groß muss eine Hash-Tabelle in Abhängigkeit zu der Anzahl Elemente sein?

Kollisionen beim Hashing

- Wie muss m in Abhängigkeit zu n wachsen, damit $P(NoCol|n,m)$ konstant bleibt?

$$P(NoCol|n,m) = \prod_{i=0}^{n-1}\left(1-\frac{i}{m}\right)$$

- Durch Anwendung der Logarithmus-Rechenregel kann ein Produkt in eine Summe umgewandelt werden:

$$ab = e^{\ln(ab)} = e^{\ln a + \ln b}$$

$$P(NoCol|n,m) = \exp\left(\sum_{i=0}^{n-1}\ln\left(1-\frac{i}{m}\right)\right)$$

- Logarithmus: $\ln(1-\varepsilon) \approx -\varepsilon$ für kleine ε
- Da $n \ll m$ gilt: $\ln\left(1-\frac{i}{m}\right) \approx -\left(\frac{i}{m}\right)$

Kollisionen beim Hashing

- Auflösen der Gleichung

$$P(NoCol|n,m) \approx \exp\left(-\sum_{i=0}^{n-1} \ln\left(\frac{i}{m}\right)\right) = \exp\left(-\frac{n(n-1)}{2m}\right) \approx \exp\left(-\frac{n^2}{2m}\right)$$

- Ergebnis:
  Kollisionswahrscheinlichkeit bleibt konstant wenn die Größe m der Hash-Tabelle quadratisch mit der Zahl n der Elemente wächst.

Hashing: Umgang mit Kollisionen

- Kollisionen treten auf, wenn zwei Schlüssel den selben Hashwert erhalten und an die gleiche Stelle gespeichert werden müssen.

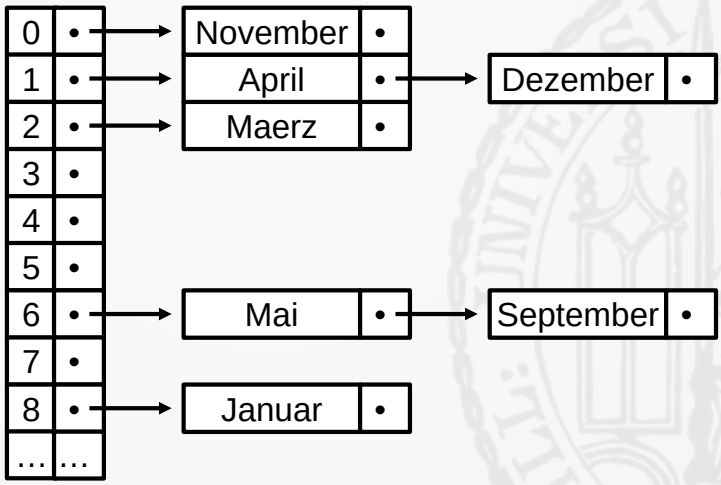
- Kollisionen sind lassen sich nicht vermeiden, deswegen gibt es entsprechende Methoden zur Behandlung.

- Tritt eine Kollision auf, so gibt es zwei populäre Auflösungsstrategien:
  - **Offenes Hashing mit geschlossener Adressierung**
  - **Geschlossenes Hashing mit offener Adressierung**

    Achtung: In der Literatur gerne als Offenes/Geschlossenes Hashing abgekürzt und dann teils vertauscht benutzt!

Offenes Hashing (mit geschlossener Adressierung)

- Speicherung der Schlüssel außerhalb der Tabelle, z.B. als verkettete Liste.
- Bei Kollisionen werden Elemente unter der selben Adresse abgelegt.
- Die externe Speicherstruktur hat großen Einfluss auf Effektivität und Effizienz.



Geschlossenes Hashing (mit offener Adressierung)

- Bei Kollision wird mittels bestimmter Sondierungsverfahren eine freie Adresse gesucht.
- Jede Adresse der Hashtabelle nimmt höchstens einen Schlüssel auf.
- Das Sondierungsverfahren bestimmt die Effizienz, so dass nur wenige Sondierungsschritte nötig sind.

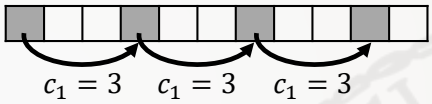
0	November
1	April
2	Maerz
3	Dezember
4	
5	
6	Mai
7	September
8	Januar
…	…

Polynomielles Sondieren

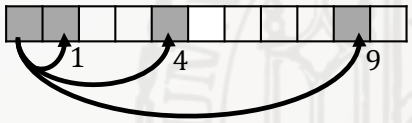
- Für $j = 0 \ldots m$ teste die Hashadressen $h(x, j)$, bis eine freie Adresse gefunden wird:

$$h(x, j) = (h(x) + c_1 j + c_2 j^2 + \cdots) \; mod \; m$$

- Lineares Sondieren
  $h(x, j) = (h(x) + c_1 j) \; mod \; m$



- Quadratisches Sondieren
  $h(x, j) = (h(x) + c_2 j^2) \; mod \; m$



- Problem Clusterbildung: für gleiche Schlüssel werden dieselben Positionen sondiert.

Geschlossenes Hashing: Komplexität

- Anzahl Sondierungsschritte

Einfügen:	$C_{ins}(n, m)$
Suche ohne Erfolg:	$C^-_{search}(n, m)$
Suche mit Erfolg:	$C^+_{search}(n, m)$
Löschen:	$C_{del}(n, m)$

m: Größe der Hash-Tabelle
n: Anzahl der Einträge
$\alpha = \frac{n}{m}$: Belegungsfaktor der Hash-Tabelle

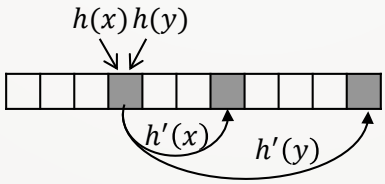
Belegung α	$C^-_{search}(n, m) = C_{Ins}(n, m) \approx \frac{1}{1-\alpha}$	$C^+_{search}(n, m) = C_{Del}(n, m) \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
0,5	≈ 2	≈ 1,38
0,7	≈ 3,3	≈ 1,72
0,9	≈ 10	≈ 2,55
0,95	≈ 20	≈ 3,15

min. n=19 und m=20 damit α=0,95 (bei ganzen Zahlen)

Doppelhashing

- Doppelhashing soll Clusterbildung verhindern, dafür werden zwei unabhängige Hashfunktionen verwendet.

- Dabei heißen zwei Hashfunktionen h und h' unabhängig, wenn gilt
  - Kollisionswahrscheinlichkeit $P\big(h(x) = h(y)\big) = \frac{1}{m}$
  - $P\big(h'(x) = h'(y)\big) = \frac{1}{m}$
  - $P\big(h(x) = h(y) \wedge h'(x) = h'(y)\big) = \frac{1}{m^2}$
- Sondierung mit $h(x, j) = (h(x) + h'(x) \cdot j^2)\ mod\ m$
- Nahezu ideales Verhalten aufgrund der unabhängigen Hashfunktionen



Hashing: Suchen nach Löschen

- Offenes Hashing: Behälter suchen und Element aus Liste entfernen → kein Problem bei nachfolgender Suche
- Geschlossenes Hashing:
  - Entsprechenden Behälter suchen
  - Element entfernen und Zelle als gelöscht markieren
    - Notwendig da evtl. bereits *hinter* dem gelöschten Element andere Elemente durch Sondieren eingefügt wurden
      (In diesem Fall muss beim Suchen über den freien Behälter hinweg sondiert werden)
  - Gelöschte Elemente dürfen wieder überschrieben werden

Hashing: Zusammenfassung

- Anwendung:
  - Postleitzahlen (Statische Dictionaries)
  - IP-Adresse zu MAC-Adresse (i.d.R. im Hauptspeicher)
  - Datenbanken (Hash-Join)

- Vorteil
  - Im *Average Case* sehr effizient: $O(1)$

- Nachteil
  - Skalierung: Größe der Hash-Tabelle muss vorher bekannt sein
    - Abhilfe: Spiral Hashing, lineares Hashing
  - Keine Bereichs- oder Ähnlichkeitsanfragen
    - Lösung: Suchbäume

Suchen: Zusammenfassung

Hashing
- Extrem schneller Zugriff für Spezialanwendungen
  - Bestimmung einer Hashfunktion für die Anwendung
  - Beispiel: Symboltabelle im Compilerbau, Hash-Join in Datenbanken

Binärer Bäum (AVL-Baum, Splay-Baum)
- Allgemeines effizientes Verfahren für Indexverwaltung im Hauptspeicher
  - Bereichsanfragen möglich, da explizit ordnungserhaltend
  - Bei Updates effizienter als sortierte Arrays

B-Baum, B^+-Baum, R-Baum, etc.
- Effiziente Implementierung für die Verwendung von blockorientierten Sekundärspeichern
- B^+-Bäume werden in nahezu allen Datenbanksystemen eingesetzt