Datensätze und Schlüssel, Suchen nach Schlüssel, Indizieren, Hashtabellen, Kollisionen und deren Behandlung, Hashfunktionen für Strings, dynamische Hashtabellen

8. TABELLEN UND HASHING

Datensätze und Schlüssel

- Annahme: wir wollen eine Menge von Datensätzen verwalten

Id	Marke	Herkunft	Menge	Preis
1	Huerlimann	Zürich	8	3.48
8	Turbinenbräu	Zürich	3	3.45
3	Feldschlösschen	Rheinfelden	3	3.30
9	Ittinger	Ittingen	8	5.65
12	Quöllfrisch	Appenzell	6	3.30

- Wir wollen in einer Menge von Datensätzen nach Schlüssel suchen
- Schlüssel könnte z.B. Id sein oder Markenname, oder sogar Kombinationen z.B. Hersteller, Marke (oben nicht dargestellt)

Datensatz – Implementation

Implementation des Datensatzes: Übersetzung 1:1 in eine Klasse

Id	Marke	Herkunft	Menge	Preis
1	Huerlimann	Zürich	8	3.48
8	Turbinenbräu	Zürich	3	3.45
3	Feldschlösschen	Rheinfelden	3	3.30
9	Ittinger	Ittingen	8	5.65
2	Quöllfrisch	Appenzell	6	3.30

```
public class BeerRecord {
    int id;
    String brand;
    String origin;
    int volume;
    double price;

    BeerRecord (int pid, String pbrand, String porigin, int pvolume, double pprice)
    {
        id = pid; brand = pbrand; origin = porigin; volume = pvolume; price = pprice;
    }
}

...
Container.put(new BeerRecord(1,"Huerlimann","Zürich",8,3.49));
Container.put(new BeerRecord(8,"Turbinenbräu","Zürich",3,3.45));
...
```

```
id: 1
brand:Huerlimann
origin: Zürich
volume: 8
price: 3.49
```

Wie sorgen wir für schnelles Suchen im Container?

Effizientes Suchen nach Schlüssel?

- Erste Beobachtung: bekannte Datenstrukturen helfen nur bedingt:

	Suchen nach Schlüssel	Einfügen, Löschen
Array	$O(n)$ (unsortiert) $O(\log n)$ (sortiert)	$O(n)$
Verkettete Liste	$O(n)$	$O(1)$ (unsortiert) $O(n)$ (sortiert)
Suchbaum	$O(\log n)$ (im Mittel) $O(n)$ (worst case)	$O(\log n)$
AVL Baum	$O(\log n)$ (worst case)	$O(\log n)$

- Und: wie sucht man nach verschiedenen Schlüsseln (id, name)?

Indizieren nach Schlüssel?

- Annahme: sehr begrenzter Wertebereich für die Schlüssel
- Dann könnte man die Datensätze in einem Array speichern

☺ Einfügen: $O(1)$
Suchen: $O(1)$

☹ Platzverschwendung

☹ Annahme unrealistisch

☹ Funktioniert nur für einen speziellen Schlüssel

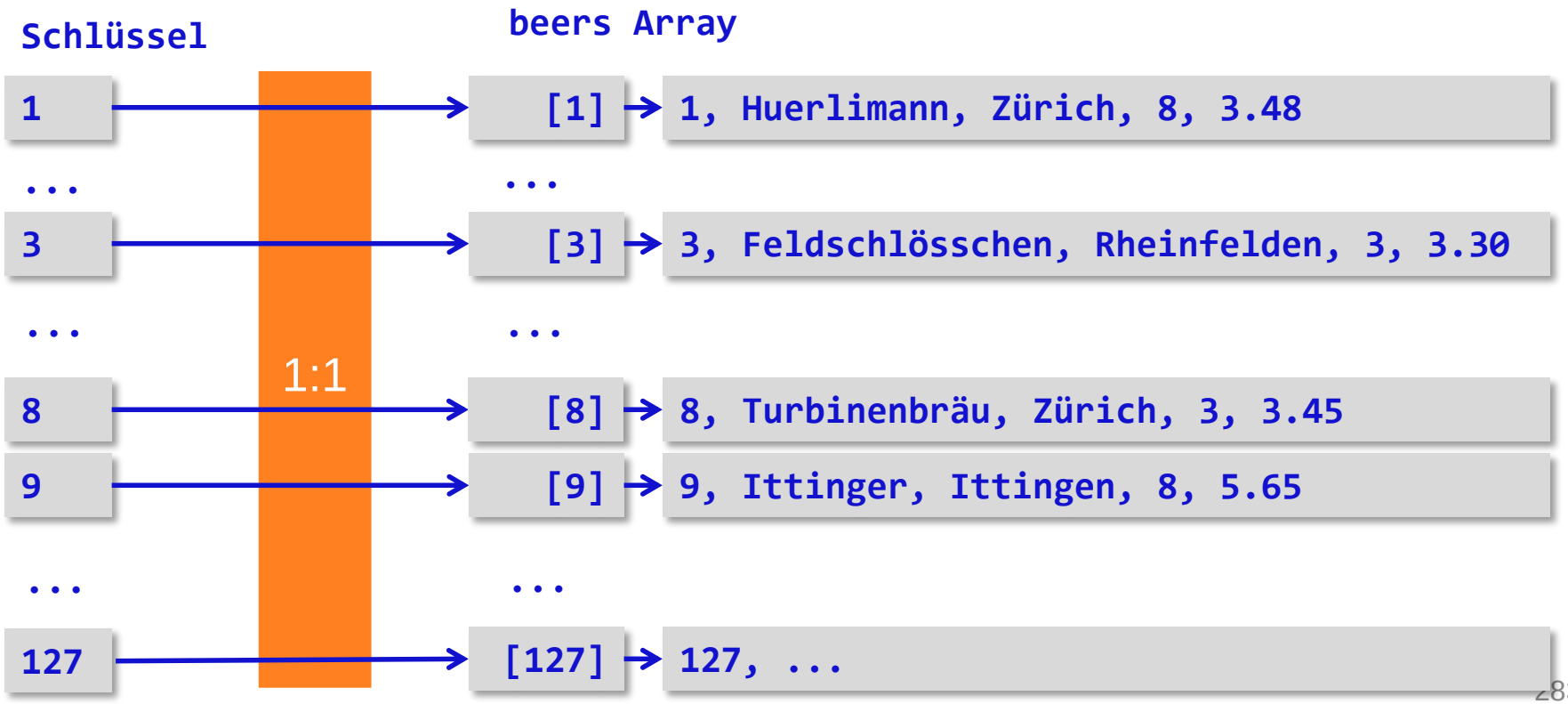
Id	Marke	Herkunft	Menge	Preis
1	Huerlimann	Zürich	8	3.48
2	leer			
3	Feldschlösschen	Rheinfelden	3	3.30
4	leer			
5	leer			
6	leer			
7	leer			
8	Turbinenbräu	Zürich	3	3.45
9	Ittinger	Ittingen	8	5.65
10	leer			
11	leer			
12	Quöllfrisch	Appenzell	6	3.30
13	leer			
...	...			

Indizieren nach Schlüssel

BeerRecord[] beers;

beers = new BeerRecord[128];

```
beers[1] = new BeerRecord(1,"Huerlimann","Zürich",8,3.49));
beers[3] = new BeerRecord(3,"Feldschlösschen", "Rheinfelden",3,3.30));
...
```



Und was nun?

Id	Marke	Herkunft	Menge	Preis
870301	Huerlimann	Zürich	8	3.48
901227	Turbinenbräu	Zürich	3	3.45
040101	Feldschlösschen	Rheinfelden	3	3.30
100802	Ittinger	Ittingen	8	5.65
120101	Quöllfrisch	Appenzell	6	3.30

Problem der direkten Indizierung: Wertebereich der Schlüssel viel grösser als Anzahl erwarteter Datensätze.

Hash Tabellen

- Problem der direkten Indizierung: Wertebereich der Schlüssel viel grösser als Anzahl erwarteter Datensätze.
- Lösung: Hashing
  - Speichere die Datensätze in einem Array,

  verwende jedoch den Schlüssel nicht als Index, sondern

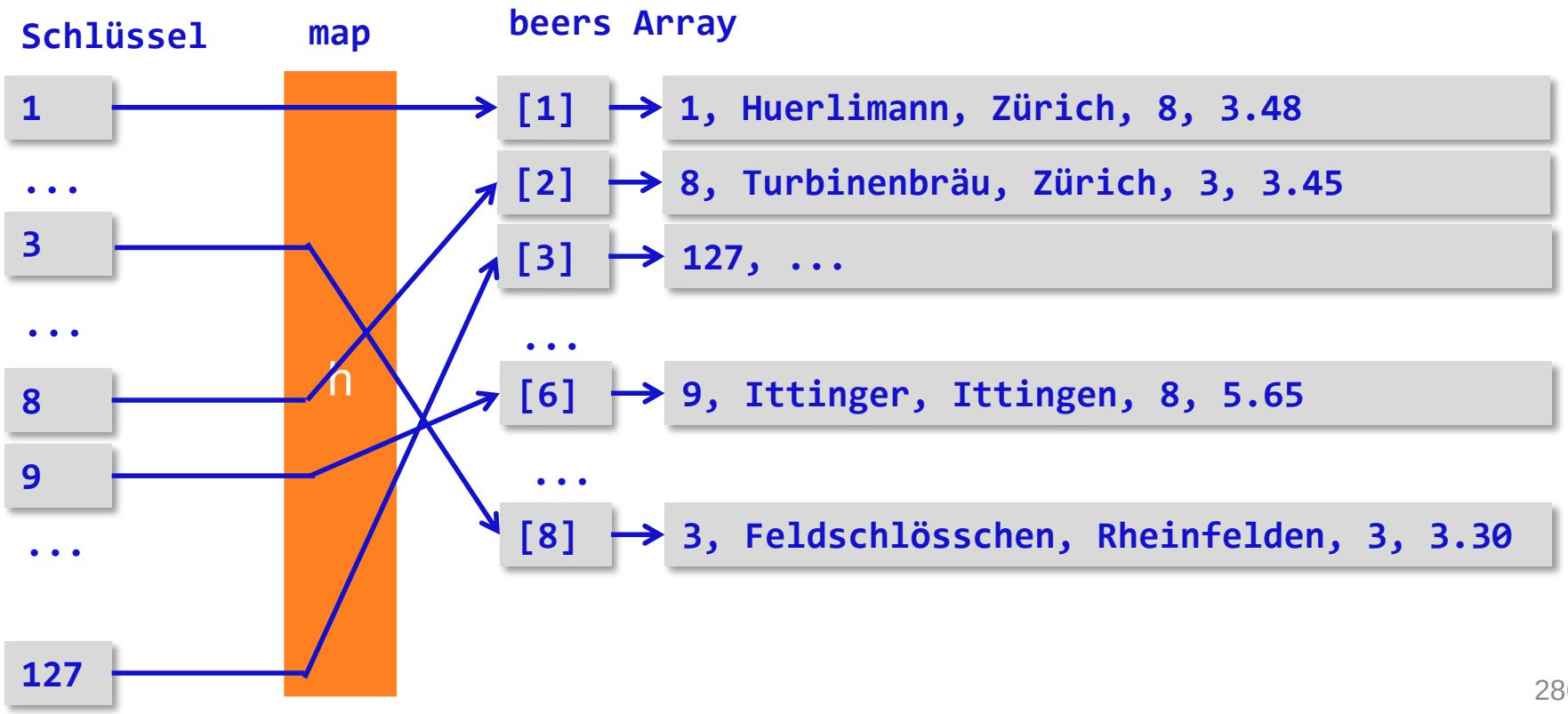
  - verwende eine Hash-Funktion: Abbildung, die zu jedem Schlüssel einen Arrayindex liefert (Schlüssel müssen dann auch nicht mehr ganze Zahlen sein) und
  - behandle Fälle in denen die Abbildung zu verschiedenen Schlüsseln den gleichen Index im Array liefert.
- Die resultierende Datenstruktur heisst Hash-Tabelle

Hash Tabelle

BeerRecord[] beers;

beers = new BeerRecord[128];

```
beers[h(1)] = new BeerRecord(1,"Huerlimann","Zürich",8,3.49));
beers[h(3)] = new BeerRecord(3,"Feldschlösschen", "Rheinfelden",3,3.30));
...
```



Einfache Hash-Funktion int → int

- Für Ganzzahl-Schlüssel ist die folgende Funktion sehr einfach (und oft ausreichend)

$$h(k) = k \bmod n$$

k: Schlüsselwert

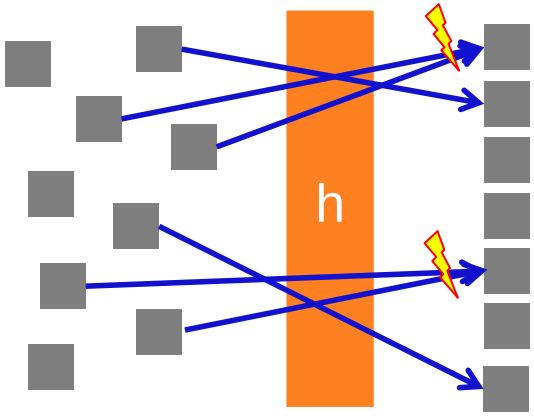
n: Grösse der Tabelle

- Beispiel (mit verschiedenen Werten für n)

Id	h(id), n=5	h(id), n=8	h(id), n=13	h(id), n=100
870301	1	5	3	1
901227	2	3	2	27
040101	1	5	6	1
100802	2	2	0	2
120101	1	5	7	1

Kollisionen

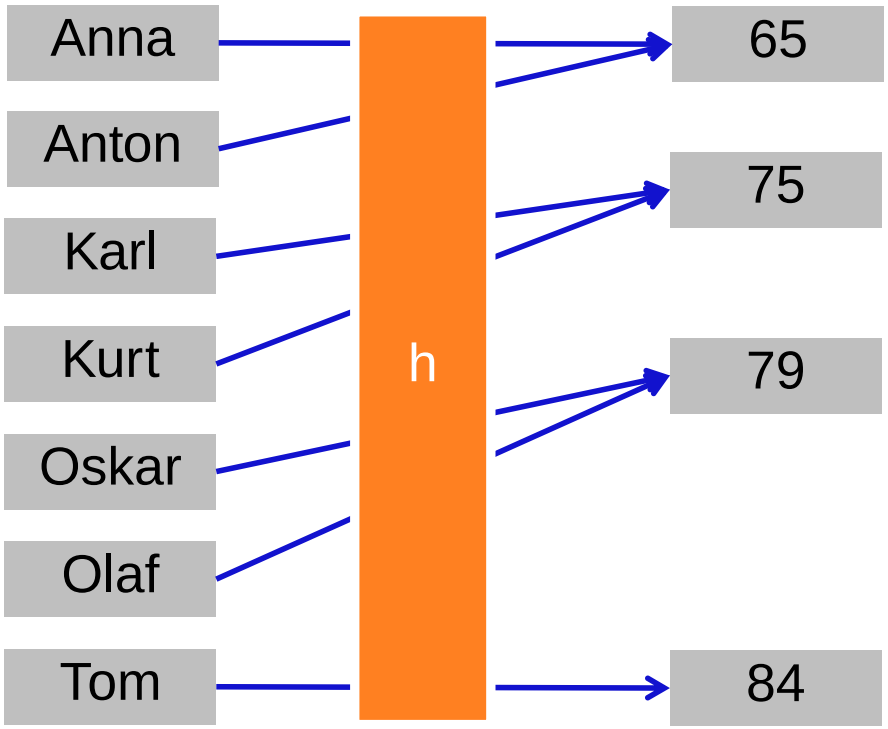
Da der Wertebereich der Schlüssel deutlich grösser ist als der Indexbereich eines Arrays, ist es grundsätzlich unumgänglich, dass mehr als ein Schlüssel auf den gleichen Index abgebildet werden könnte. Einen solchen Fall nennt man *Kollision.*



1. Was macht man, wenn eine Kollision auftritt?
2. Wie sieht eine Hash-Funktion aus, die *möglichst wenige* Kollisionen erzeugt?

Beispiel einer Hash-Funktion für Strings

- Einfache Hash Funktion Name → Index: Ordnungszahl (Unicode) des ersten Buchstaben

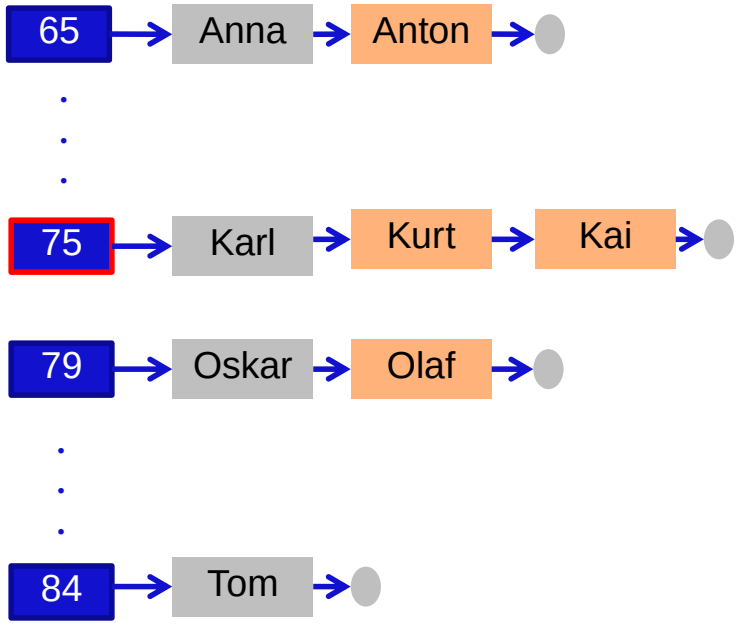


Beispiel zur Illustration der Probleme mit Kollisionen!

Bessere Hash-Funktion für Strings folgt

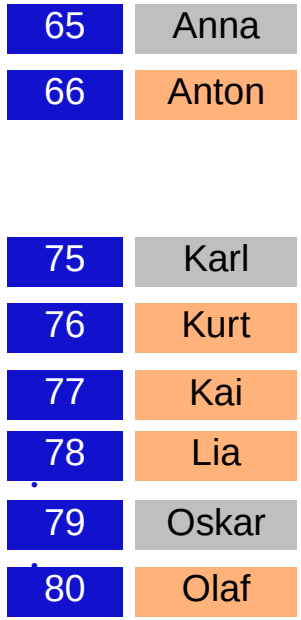
Behandlung von Kollisionen: Verketten der Einträge

- Vorgehen: speichere für jeden Index eine verkettete Liste der Elemente
- Lookup("Kurt"): Durchlaufe Liste bei h("Kurt") bis "Kurt" gefunden
- ☹ separate Datenstruktur
  - Komplizierter
  - Speicherplatzverbrauch für den next-Pointer
- ☹ Nutzt den freien Platz in der Tabelle nicht
- ☺ Funktioniert auch für kleine Tabellen (Tabellenlänge < Anzahl Instanzen), allerdings ist dann Zugriffszeit "weit weg" von O(1)



Behandlung von Kollisionen: Offene Addressierung

- Vorgehen: wenn Platz bei h(Name) schon besetzt, speichere bei einem anderen freien Platz
  - Z.B. lineares Sondieren: nimm den jeweils folgenden Index . (Alternativen: quadratisches Sondieren, double hashing)
  - Ist man mit der Suche beim letzten Index angelangt, fährt man bei 0 weiter ("wrap around"), jedoch maximal n Schritte.



☺ Freier Platz in der Tabelle wird genutzt und immer gefunden

☹ Häufung besetzter Positionen um einen viel genutzten Hashwert herum. Sorgt für Mehraufwand beim Sondieren.

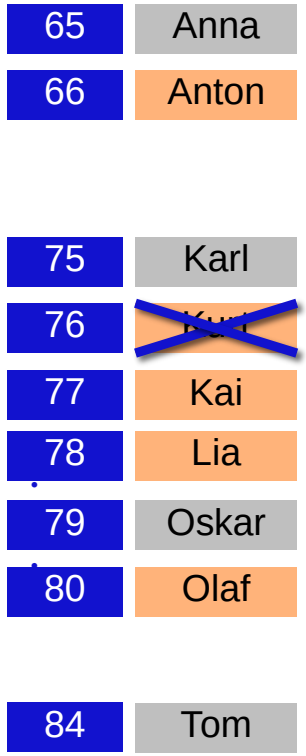
Löschen bei offener Addressierung

Problem

- Löschen eines Eintrages kann zum frühzeitigen Abbruch des Sondierens beim Suchen führen.

Lösung

- Spezialeintrag "gelöscht", welches das Abbrechen des Sondierens unterbindet, ansonsten aber wie ein freier Eintrag wirkt.



Implementation Hashtabelle (Biere)

```
public class BeersHashed {
    BeerRecord[] beers;
    int used;
    BeerRecord free; // special element for deletion

    BeersHashed (){
        beers = new BeerRecord[7]; // initial size
        free = new BeerRecord();
        used= 0;
    }
    // lineares Sondieren mit einfacher Hashfunktion h(id) = id % n
    // Parameter free kann zum erlaubten Suchen nach Spezialelement genutzt werden
    int GetPosition(int id, BeerRecord free)
    {
        int position = id % beers.length; // einfache Hashfunktion
        while (beers[position] != null
                && beers[position].id != id
                && beers[position] != free)
            position = (position+1) % beers.length
        return position;
    }
    ...
```

Implementation: Einfügen und Suchen

```
...

public void Put(BeerRecord beer)
{
    CheckFillLevel();
    int position = GetPosition(beer.id, free);
    if (beers[position] == null)
        ++used;
    beers[position] = beer;
}

public BeerRecord Get(int id)
{
    int position = GetPosition(id, null);
    return beers[position];
}

...
```

erlaubt, dass die Suche nach freiem Element beim Spezialelement free endet

Suche endet entweder beim Element mit id oder bei null

Implementation: Löschen

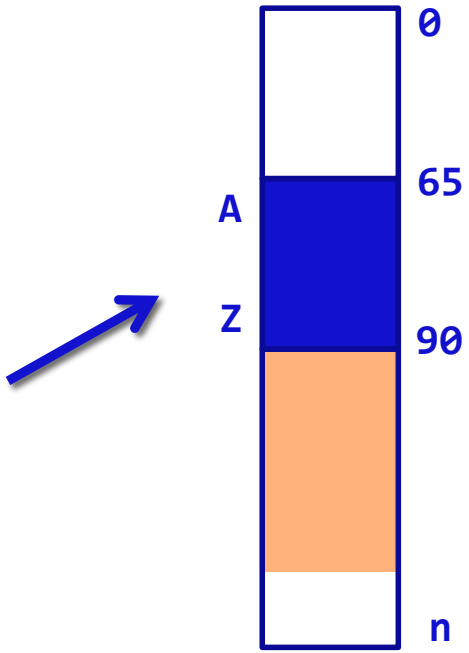
```
    public boolean Delete(int id)
    {
        int position = GetPosition(id, null);
        if (beers[position] != null)
        {
            --used;
            if (beers[(position+1) % beers.length] == null)
                beers[position] = null;
            else
                beers[position] = free;
            return true;
        }
        else
            return false;
    }
}
```

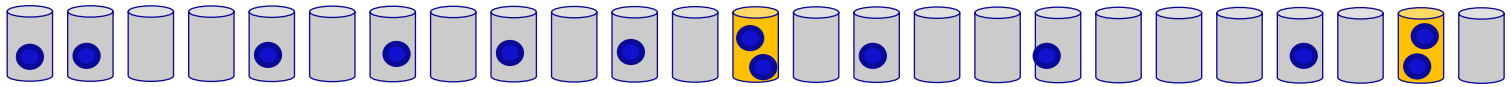
Wenn das nächste Element null ist, darf das zu löschende auch null gesetzt werden, da jede Suche dort enden würde, andernfalls muss das zu löschende Element durch das Spezialelement ersetzt werden

Wie wahrscheinlich sind Kollisionen?

- Wenn die Hash-Funktion schlecht gewählt ist, können Kollisionen sehr wahrscheinlich werden, darüber hinaus werden die Sondierungen sehr teuer
- Beispiel: obige Hash-Funktion

  $\text{Name} \rightarrow \text{Index Anfangsbuchstabe}$
- Ziel: Versuche möglichst eine Gleichverteilung der Schlüsselwerte zu erreichen
- Annahme: Gleichverteilung annähernd erreicht. Wie wahrscheinlich sind dann noch Kollisionen bei einem Füllgrad von, z.B. 10%?



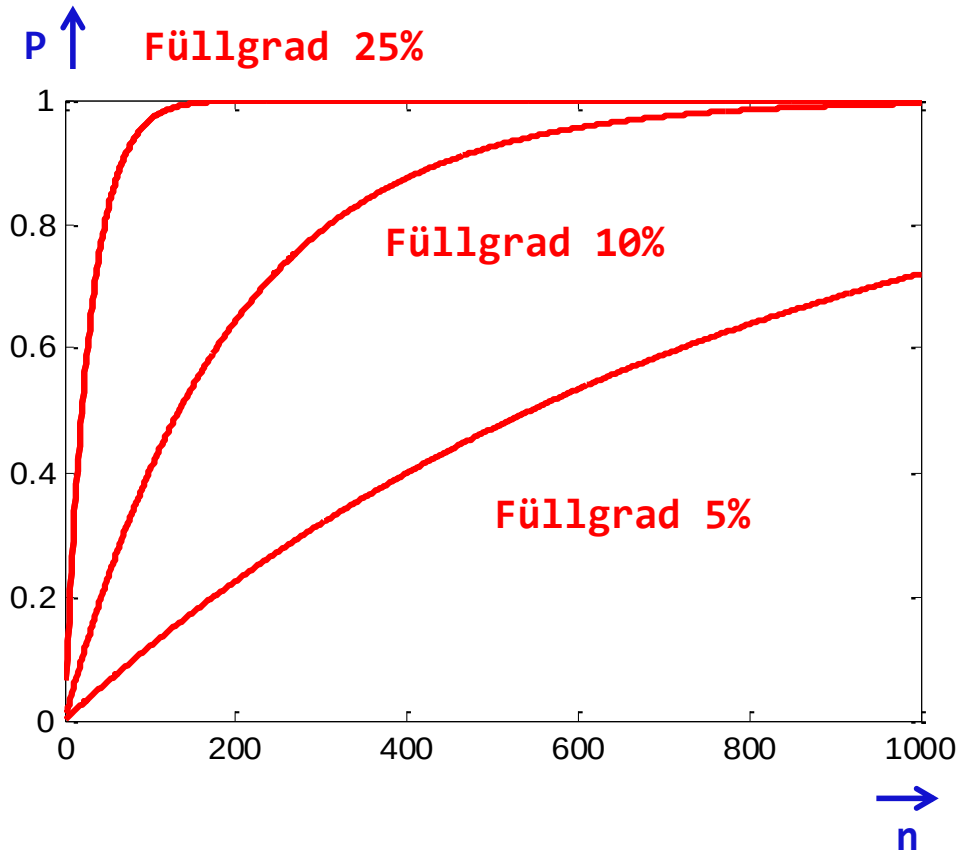


Geburtstagsproblem*

Bei welcher Anzahl Leute haben mit einer Wahrscheinlichkeit von mehr als 50% zwei Leute am selben Tag Geburtstag?

- Antwort: schon ab 23!

Allgemeine Approximation $\mathbb{P}(Kollision) \approx 1 - \left(\frac{n}{n-m}\right)^{n+\frac{1}{2}-m} \cdot e^{-m}$.



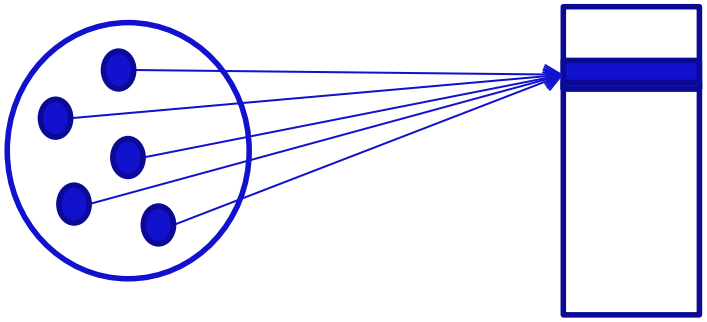
n: Anzahl Indizes (Tage)

m: Anzahl Belegungen (Leute)

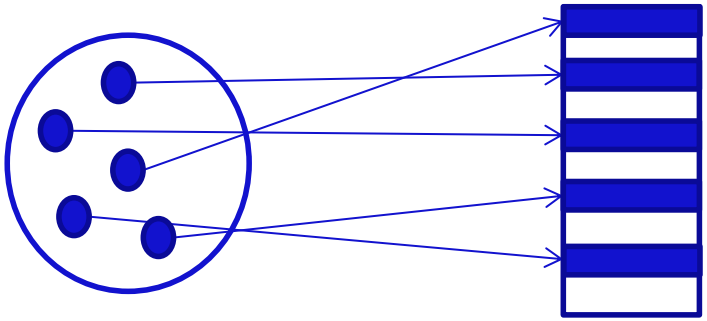
Herleitung: W't für keine Kollision hinschreiben, Approximation der Fakultät mit Stirling-Formel

Hash-Funktion: Kollisionen

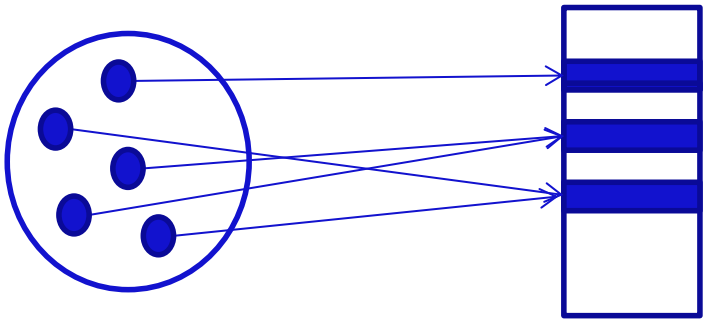
am schlechtesten



am besten



akzeptabel



Hash-Funktionen: Anforderungen

1. Effizient auszurechnen
2. Möglichst volle Information im Schlüssel benutzen
   - Jede Änderung eines Zeichens , einer Ziffer etc. solllte den Hash-Code ändern
   - Annähernde Gleichverteilung auf der Tabelle
3. Funktion: Schlüssel → Tabelle
   - Funktion muss für einen Schlüssel muss immer denselben Hash-Code zurückgeben
   - Wertebereich: Indizes der Tabelle

Unsere einfache Hashfunktion, die nur Anfangsbuchstaben verwendet, verletzt dies deutlich

erreicht man z.B. durch Verwendung einer Modulo-Operation

```
index = h(x) % n
```

Hash-Funktionen für Strings: erster Versuch

Summe der Unicode-Werte der Buchstaben

z.B. h("abc") = 97+98+99

Probleme

1. Permutationen unerkannt h("abc") = h("acb")
2. Ziemlich einfach, gleiche Indizes zu erzeugen h("abc")=h("aad")
3. Kleiner Wertebereich
   z.B. h("a") … h("zzzzz") = 97 .. 600
   vs. 27^5 = 14 Mio. mögliche Kombinationen
   - Entscheidender Nachteil: Vergrösserung der Hash-Tabelle bringt keinen Vorteil

Eine Hash-Funktion für Strings

Gewichtete Summe der Unicode Werte

$$h(s) = s_0 \cdot b + s_1 \cdot b^2 + s_2 \cdot b^3 + \cdots + s_l \cdot b^l$$

s_i: Unicode Wert des i–ten Buchstabens
l: Länge des Wortes
b: Konstante

- Hash-Wert verschiedener Strings ist verschieden.
- Füllt auch grosse Tabellen gut aus.
  - Einschränkung auf die Tabellengrösse durch Verwendung der Modulo- Operation
    `index(s) = h(s) % n`
  - Wahl der Tabellengrösse (n) kann wichtig sein: ist n eine Primzahl, erhält man in der Regel bessere Durchmischung
- Java-Bibliothek verwendet b=31

Dynamische Hashtabellen

Können im Prinzip so implementiert werden wie dynamische Arrays

Einziges Problem: Die Indexberechnung ändert sich mit der Grösse der Tabelle
`index(x) = h(x) % n`

Daher Vergrössern mit Einfügen über Put:

```
int newSize = (beers.length+1)*2;
BeerRecord[] oldBeers = beers;
beers = new BeerRecord[newSize];
used = 0;
for (int i=0; i < oldBeers.length; ++i)
  if (oldBeers[i] != null)
    Put(oldBeers[i]);
```

Hash-Tabellen Effizienz

- Im besten Falle: Einfügen und Suchen O(1)
- Im schlechtesten Falle: Einfügen und Suchen linear
  - O(l) : verkettete Listen, l: Listenlänge
  - O(n): offene Adressierung, n: Tabellengrösse
- O(log n) kann meist übertroffen werden, oft "sehr nahe" an O(1). Wichtig dafür:
  - Gute Wahl der Hash-Funktion
  - Begrenzen des Füllgrads der Hash-Tabelle