# Informatik II

Vorlesung am D-BAUG der ETH Zürich

## Vorlesung 5, 21.3.2016

**Online-Statistik: Zirkulärer Buffer, Provisional Means**
**Hashtabellen: Datensatz, Hash-Funktion, Kollision, Lineares Sondieren, Dynamische Hashtabellen**

# **Fallstudie**: Online-Statistik

Ziel: Klasse / Objekt, welches Daten konsumiert und zu jeder Zeit Statistiken, z.B. Mittelwert, Varianz, Median (etc.) ausgeben kann

```
Statistics s = new Statistics(maxSize);
...

// Viele Datenpunkte einfügen
s.Put(data);

...
System.out.println(s.Mean());
System.out.println(s.Variance());
System.out.println(s.Median());
```

# Angestrebtes Interface

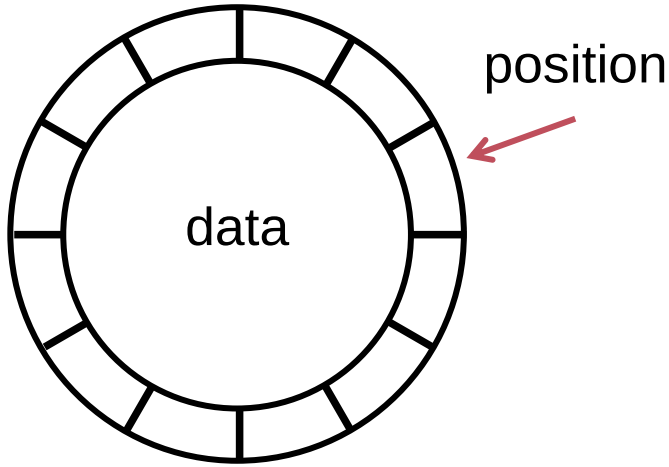
```
public class Statistics {

   // Konstruktor
   public Statistics()

   // Hinzufügen von Daten
   public void Put(double value)

   // Rückgabe des Mittelwertes
   public double Mean()

   // Rückgabe der geschätzten Varianz
   public double Variance()
}
```

Mögliche Implementationen?

- wachsendes Array
  (ähnlich wie in der Übung)
- **Zirkulärer Puffer**
- **Online-Algorithmus**

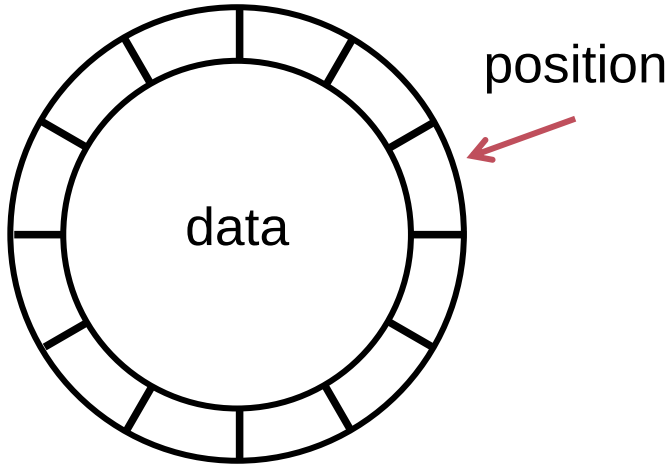
# Zirkulärer Puffer

```
public class Statistics {
   private double data[];          // Zirkulärer Puffer
   private int n;                  // Füllgrad
   private int pos;                // momentane Position

   // Konstruktor
   Statistics(int maxSize) {
      n = 0;
      pos = 0;
      data = new double[maxSize];
   }

   ...
}
```



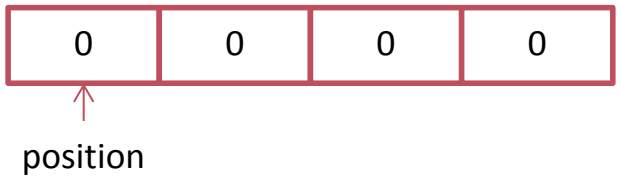
# Einfügen

```
public class Statistics {
  public Statistics() {...}

  // füge Wert in die den zirkulären Puffer ein
  public void Put(double value){
    data[pos] = value;
    if (n < data.length)
      n++;
    pos = (pos + 1) % data.length;
  }
...
}
```



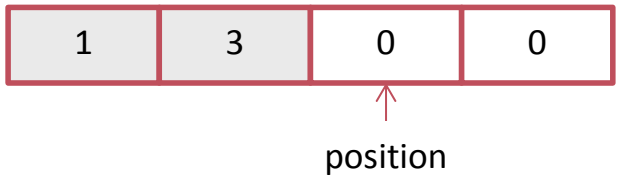
# Szenario

```
Statistics s = new Statistics();
```



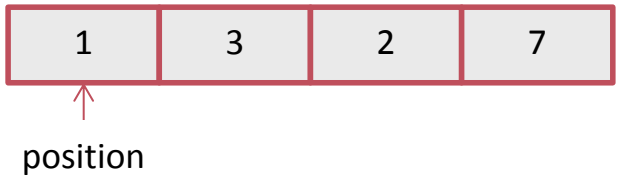

n= 0

```
s.put(1);
s.put(3);
```



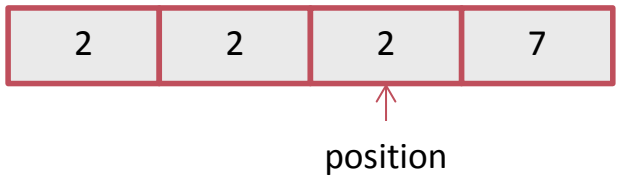

n= 2

```
s.put(2);
s.put(7);
```




n= 4

```
s.put(2);
s.put(2);
```




n= 4

```
// füge Wert ein
public void Put(double value){
   data[pos] = value;
   if (n < data.length)
      n++;
   pos = (pos + 1) % data.length;
}
```

# Mittelwert

```
public class Statistics {
  public Statistics() {...}
  public void Put(double value){...}

  public double Mean() {
    double sum = 0;
    for (int i=0; i<n; ++i)
      sum += data[i];
    return sum / n;
  }

  ...
}
```

n Schritte

# Varianz

```
public class Statistics {
   public Statistics() {...}
   public void Put(double value){...}
   public double Mean() {...}

   public double Variance() {
      if (n > 1) {
         double ssq = 0;
         double mean = Mean();
         for (int i = 0; i < n; ++i)
            ssq += (data[i]-mean) * (data[i]-mean);
         return ssq / (n-1);
      }
      else
         return 0;
   }
}
```

2 * n Schritte

# Frage

Ist der Algorithmus oben für sehr grosse Datensätze im Sinne eines online-Algorithmus geeignet?

Antwort: nein, Online-Algorithmen halten nützliche Zwischenergebnisse und berechnen nicht jedes mal neu.

# Effizienz

Obige Implementation ist nicht sehr effizient.

Wenn `Mean` oder `Variance` oft, z.B. nach jedem Eintrag eines Wertes, abgefragt wird, dann lohnt sich der *Provisional Means* Algorithmus

Wenn $m_n = \frac{1}{n}\sum_{i=1}^{n} x_i$, dann $m_{n+1} = m_n + \frac{x_{n+1} - m_n}{n+1}$

Für $s_n = \sum_{i=1}^{n}(x_i - m_n)^2$ analog: $s_{n+1} = s_n + (x_{n+1} - m_n) \cdot (x_{n+1} - m_{n+1})$

Damit fällt sogar die Notwendigkeit zum Speichern des Arrays weg

# Provisional Means

```
public class Statistics {
   private int n = 0;
   private double mean = 0;
   private double ssq = 0;

   // Einfügen und Update
   public void Put(double value){
      n++;
      double oldMean = mean;
      mean = oldMean + (value - oldMean) / n;
      ssq = ssq + (value - oldMean) * (value - mean);
   }

   ...
}
```

# Mittelwert und Varianz

```
public class Statistics {
   public void Put(double value){

   public double Mean(){
      return mean;
   }

   public double Variance() {
      if (n>1)
         return ssq / (n-1);
      else
         return 0;
   }
}
```

"1 Schritt"

"1 Schritt"

# Fragen

Können wir das auch mit dem Median machen?

Antwort: nein, deutlich komplizierter

Wenn $s(x,k)$ denn k-grössten Wert des Datensatzes $x$ mit Länge $n$ bezeichnet $(1 \leq k \leq n)$, so ist der Median definiert als

$$s\left(x, \frac{n+1}{2}\right), \quad \text{falls n ungerade und}$$

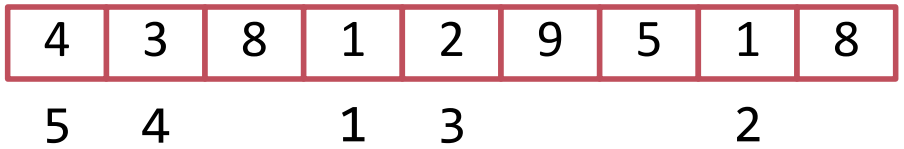
$$\frac{1}{2} \cdot \left(s\left(x, \frac{n}{2}\right) + s\left(x, \frac{n+1}{2}\right)\right), \quad \text{falls n gerade}$$

Was ist ein guter (On-line) Algorithmus für die Berechnung des Medians?

# Median

Bestimmung des Medians ist ein *Auswahlproblem*

Naiv: Bestimmung des grössten / kleinsten Elements in n Schritten durch Traversieren aller Elemente.




Iterative Anwendung dieses Verfahrens unter Streichung des vorherigen Minimums liefert Median in

$\frac{n+1}{2} \cdot n$ Schritten (wenn $n$ ungerade)

Bessere Ideen?

# Median

Wir wissen schon aus Informatik I, dass *Mergesort* $n \log n$ Schritte (Präzisierung folgt!) benötigt.

**Schneller als naive Methode**:

- Sortieren des Arrays (Mergesort) und nachfolgend
- Auswahl des mittleren Elements (Array-Zugriff)

Nachteil

- Verändert die Daten oder benötigt Kopie der Daten

Vorteil

- Ist generisch: direkt für Auswahl des $k$-kleinsten Elements verwendbar

# On-line Median?

Beobachtung: für zukünftige Updates des Medians sind immer alle Daten relevant.

- Es gibt keine Abkürzung wie beim Mittelwert.
- "Beweis": jeder Datenpunkt kann irgendwann zum Median werden.

Idee

- Daten werden grundsätzlich sortiert eingefügt. Geht das effizient?
- Wir kommen darauf später noch einmal zurück

# Hash-Tabellen

möglichst einfach

# Datensatz

| Name | Telefon | Addresse |
|---|---|---|
| Mickey Maus | 4711 | Bahnhofstr |
| Minnie Maus | 94725 | Hauptstr |
| Donald Duck | 95627 | Hauptstr |
| … | … | … |

```
public class Friend {
   String name;
   String telephone;
   String address;

   public Friend(String Name, String Telephone, String Address) {
      name = Name;
      telephone = Telephone;
      address = Address;
   }

   public String toString(){
      return name + ", " + telephone + ", " + address;
   }
}
```

# Gesucht

Methode zum schnellen Nachschlagen von Datensatz nach Name

# Telefonbuch ?

```
class MyFriends{
   Friend[] data;
   int friends;

   public MyFriends(int maxFriends) {
      data = new Friend[maxFriends]; friends = 0;
   }

   // Annahme: f ist noch nicht im Telefonbuch
   public void Add(Friend f){
      if (friends = data.length) Grow();
      data[friends] = f;
      friends ++;
   }
   ...
}
```

# Telefonbuch ?

```
class MyFriends{
   ... //   s.o.

   public Friend Get (String name){
      for (int i = 0; i<friends; ++i)
         if (data[i].name.Equals(name))
            return data[i];
      return null;
   }

}
```

Schlecht: lange Suchzeiten
worst case O(data.length)

# Abhilfe?

Sortiert einfügen → Schnellere (binäre) Suche

Aber: Einfügen aufwändig (Übung 4!).

| # | Name | Telefon | Addresse |
|---|---|---|---|
| 0 | Donald Duck | 95627 | Hauptstr |
| 1 | Goofy | 007 | Sihlquai |
| 2 | Mickey Maus | 4711 | Bahnhofstr |
| 3 | Minnie Maus | 94725 | Hautpstr |
| … | … | … | … |
| n-1 | Zira | 8888 | Hauptstr |

# Anderer Ansatz

**Was benötigen wir eigentlich?**

Antwort: Schnelle Abbildung Name (= Schlüssel) → Index

# Angenommen...

wir hätten unendlich grossen Speicherplatz

```
Friend[] friends = new Friend[∞];
```

und könnten mit beliebig grossen Zahlen umgehen.

```
Friend friend = friends[81237372123323388880033];
```

**(Wie) könnten wir das Problem dann lösen?**

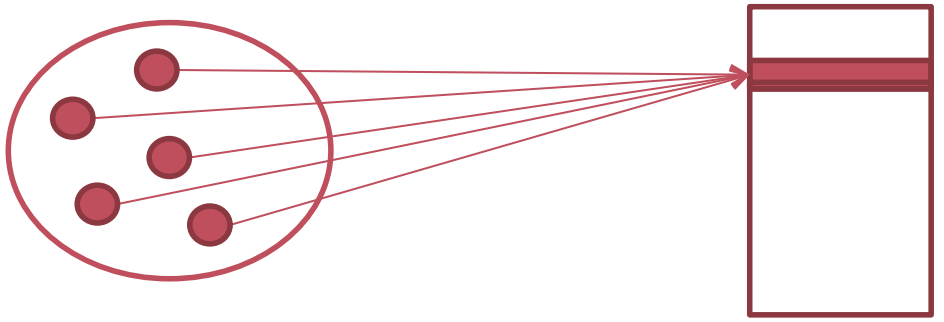
Antwort: mit einer eindeutigen Zuordnung Name → Index.

# Hash-Funktion

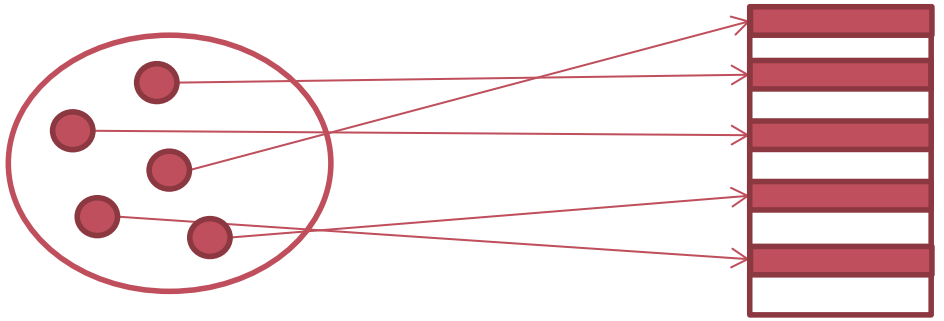
Der Versuch, zu jedem Schlüssel (hier: Name) einen möglichst eindeutigen Index zuzuordnen.

# Hash-Funktion

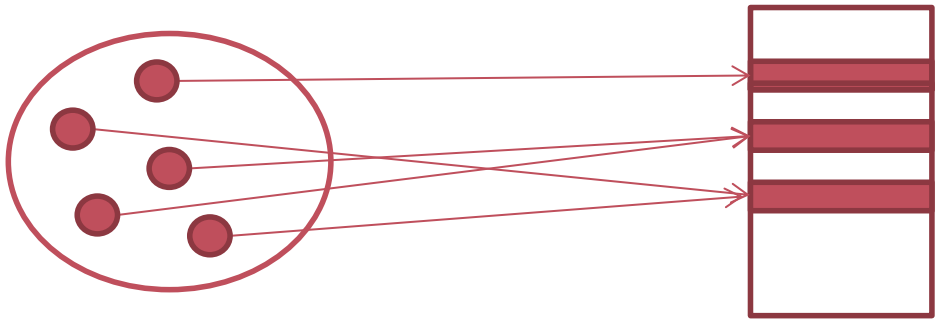
am schlechtesten



am besten

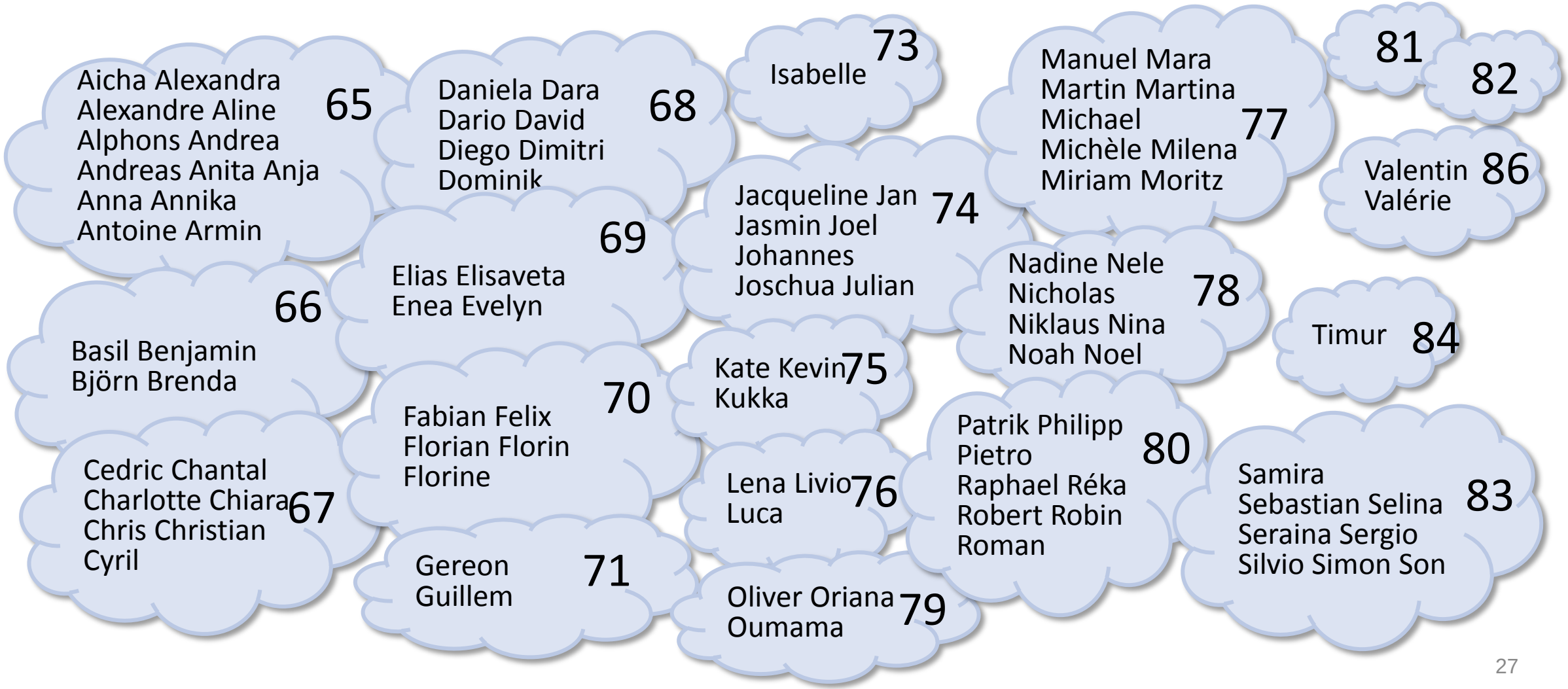


akzeptabel



# Ziemlich schlecht

## Index = Unicode-Index des Anfangsbuchstabens eines Namens



# Etwas besser

## Index = Summe der Buchstabenindices

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 281 | Jan | 304 | Son | 376 | Dara | 377 | Enea | 378 | Anja | 382 | Anna |
| 384 | Lena | 385 | Mara | 388 | Nele | **389** | **Kate** | **389** | **Luca** | **390** | **Nina** |
| **390** | **Noah** | 394 | Joel | 398 | Noel | 470 | Aicha | 488 | David | 488 | Diego |
| 489 | Aline | 491 | Basil | 493 | Anita | 494 | Elias | 495 | Dario | **503** | **Armin** |
| **503** | **Kukka** | 504 | Felix | 505 | Chris | 506 | Robin | **509** | **Kevin** | **509** | **Roman** |
| **515** | **Cyril** | **515** | **Livio** | 518 | Simon | 519 | Réka | 529 | Timur | **577** | **Fabian** |
| **577** | **Fabian** | 584 | Chiara | 586 | Cedric | 587 | Andrea | 588 | Brenda | 591 | Nadine |
| 594 | Annika | 598 | Milena | 602 | Oriana | 604 | Selina | 605 | Samira | 607 | Miriam |
| 608 | Gereon | 608 | Oumama | **610** | **Jasmin** | **610** | **Manuel** | 611 | Julian | 617 | Sergio |
| 618 | Florin | **619** | **Martin** | **619** | **Patrik** | 619 | Patrik | 622 | Robert | 625 | Oliver |
| 625 | Oliver | 627 | Evelyn | 627 | Pietro | 630 | Silvio | 642 | Björn | 645 | Moritz |
| 686 | Daniela | 691 | Michael | 691 | Michael | 699 | Chantal | 701 | Raphael | 702 | Andreas |
| 707 | Seraina | **715** | **Dominik** | **715** | **Florian** | 716 | Martina | 717 | Joschua | 718 | Antoine |
| **719** | **Florine** | **719** | **Guillem** | 722 | Dimitri | 725 | Alphons | 726 | Philipp | 727 | Niklaus |
| 801 | Isabelle | 804 | Benjamin | 817 | Nicholas | 822 | Johannes | 826 | Michèle | 833 | Valentin |
| 844 | Valérie | 912 | Alexandra | 916 | Alexandre | 922 | Sebastian | 926 | Elisaveta | 933 | Christian |
| 934 | Charlotte | 1025 | Jacqueline | | | | | | | | |

**Kollision:**

"Dominik" ↦ 715

"Florian" ↦ 715

# Ziemlich gut

## Hashfunktion aus Übung 3

$$h_b(s) = \sum_{i=0}^{l-1} s_i \cdot b^i = s_0 + s_1 \cdot b + s_2 \cdot b^2 + \cdots + s_{l-1} \cdot b^{l-1}$$

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

z.B. mit **b=100**:

**Anna ↦ 971111065**

**Jacqueline ↦ 102110609021813999774**

**für grosses b: eindeutige Zuordnung, keine Kollisionen.**

# Computerspeicher ist endlich...

Zuordnung auf endlichen Indexbereich $0..m-1$ :

$$h_{b,m}(s) = \left( \sum_{i=0}^{l-1} s_i \cdot b^i \right) \boldsymbol{mod\ m}$$

Mit gut gewähltem b und genügend grossem m sind Kollisionen selten, offensichtlich aber nicht zu vermeiden.

# Nebenbemerkung zur korrekten Berechnung von h

**Aus der Lösung zu Übung 3:**

```
final static int b = 31;

public static int ComputeHash(int m, String s) {
	int sum = 0;
	int B = b;
	for (int k = 0; k<s.length(); ++k)
	{
		sum = (sum + s.charAt(k) * B) % m;
		B = (B * b) % m;
	}
	return sum;
}
```

**(a+b) % m = (a % m + b % m) % m**
**(a*b) % m = (a % m * b % m) % m**

# Wie funktioniert eine Hastabelle?

## 1. Array für Schlüssel und Daten

```
class Friends{
    String[] key;  // Schlüssel
    Entry[] data;  // Daten

    public Friends(){
        final static int initSize = 10;
        data = new Entry[initSize];
        key = new String[initSize];
    }

    ...

}
```

# Wie funktioniert eine Hastabelle?

## 2. Funktion zur Berechnung des Array-Index

```
class Friends{
     ...

     // return the hash value corresponding to key name
     int Hash(String k){
          final int b=31;
          int value = 0;
          int m = key.length;
          for (int i = 0; i < k.length(); ++i){
               value = (value * b % m + k.charAt(i)) % m;
          }
          return value;
     }
}
```

**Hash Funktion genau wie oben, mit m = key.length;**

| k | Hash(k) [m=10] |
|---|---|
| Hannes | 1 |
| Niklas | 6 |
| Katrin | 9 |
| Tobias | 6 |
| Florian | 5 |

# Wie funktioniert eine Hastabelle?

## 3. Datenzugriff im Array: vorerst ohne Kollisionsbehandlung

```
class Friends{
    ...
    public void Set(String k, Entry f){
        int index = Hash(k);
        key[index] = k;
        data[index] = f;
    }

    public Entry Get(String k){
        return data[Hash(k)];
    }
}
```

**Falsch, sobald Kollisionen auftreten!**

| index | key | data |
|---|---|---|
| 0 | null | null |
| **1** | **"Hannes"** | **"Hannes", "007", ....** |
| 2 | null | null |
| 3 | null | null |
| 4 | null | null |
| 5 | null | null |
| **6** | **"Niklas"** | **"Niklas","008", ...** |
| 7 | null | null |
| 8 | null | null |
| **9** | **"Katrin"** | **"Katrin", "4711", ...** |
| 10 | null | null |

# Wie entdeckt man eine Kollision?

int index = Hash(k);

**Drei Fälle:**

1. key[index] = null
   Am index ist noch nichts gespeichert.
   key[Hash("Florian")] ==null
2. key[index].equals(k)
   Am index ist derselbe Schlüssel vorhanden: keine Kollision
   key[Hash("Niklas")] == "Niklas"
3. !key[index].equals(k)
   Am index ist bereits ein anderer Schlüssel gespeichert.
   key[Hash("Tobias")] == "Niklas"

| index | k |
|---|---|
| 0 | null |
| **1** | **"Hannes"** |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| **6** | **"Niklas"** |
| 7 | null |
| 8 | null |
| **9** | **"Katrin"** |
| 10 | null |

# Wie behandelt man eine Kollision?

Eine Möglichkeit ist "Lineares Sondieren":

Es wird am index = Hash(k) gestartet.

Solange key[index] eine Kollision darstellt, wird der nächste Index gewählt

# Beispiel Einfügen

| index | key |
|---|---|
| 0 | null |
| 1 | null |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| 8 | null |
| 9 | null |

| index | key |
|---|---|
| 0 | null |
| **1** | **"Hannes"** |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| 8 | null |
| 9 | null |

| index | key |
|---|---|
| 0 | null |
| **1** | **"Hannes"** |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| **6** | **"Niklas"** |
| 7 | null |
| 8 | null |
| 9 | null |

| index | key |
|---|---|
| 0 | null |
| **1** | **"Hannes"** |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| **6** | **"Niklas"** |
| **7** | **"Tobias"** |
| 8 | null |
| 9 | null |

| k | Hash(k) |
|---|---|
| Hannes | 1 |
| Niklas | 6 |
| Katrin | 9 |
| Tobias | 6 |
| Florian | 5 |

set("Hannes",...) set("Niklas",...) set("Tobias",...)

# Lineares Sondieren

```
class Friends{
    String[] key;  // Schlüssel
    Entry[] data;  // Daten
    ...

    int Probe(String k){
        int index = Hash(k);
        while(key[index] != null
                && !key[index].equals(k))
            index = (index + 1) % key.length;
        return index;
    }
}
```

| index | key |
|---|---|
| 0 | null |
| **1** | **"Hannes"** |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| **6** | **"Niklas"** |
| **7** | **"Tobias"** |
| 8 | null |
| 9 | null |

**friends.Probe("Tobias") --> 7**

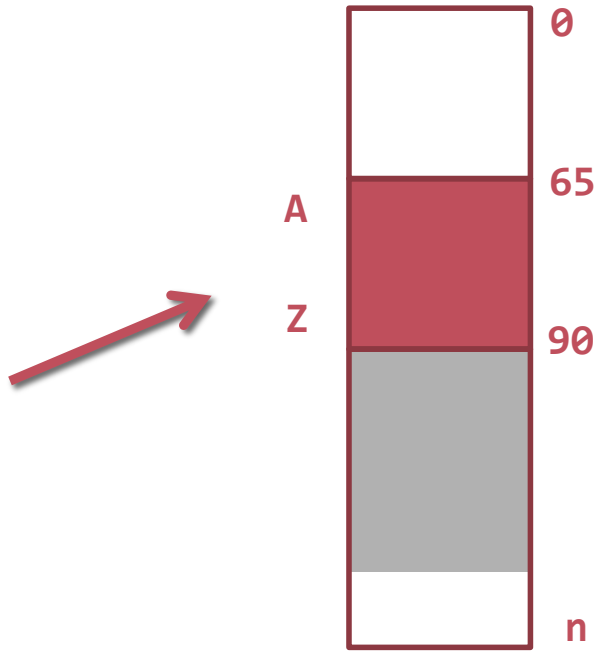
"wrap-around"

# Tabelle mit Sondieren

```
class Friends{
    String[] key;  // Schlüssel
    Entry[] data;  // Daten
    ...
    int Probe(String k);

    public void Set(String k, Entry f){
        int index = Probe(k);
        key[index]= k;
        data[index] = f;
    }

    public Entry Get(String k){
        return data[Probe(k)];
    }
}
```

# Wie wahrscheinlich sind Kollisionen?

Wenn die Hash-Funktion schlecht gewählt ist, können Kollisionen sehr wahrscheinlich werden. Sondieren wird dann teuer.




Beispiel: obige Hash-Funktion

$$\text{Name} \rightarrow \text{Index Anfangsbuchstabe}$$

Ziel: Versuche möglichst eine Gleichverteilung der Schlüsselwerte zu erreichen

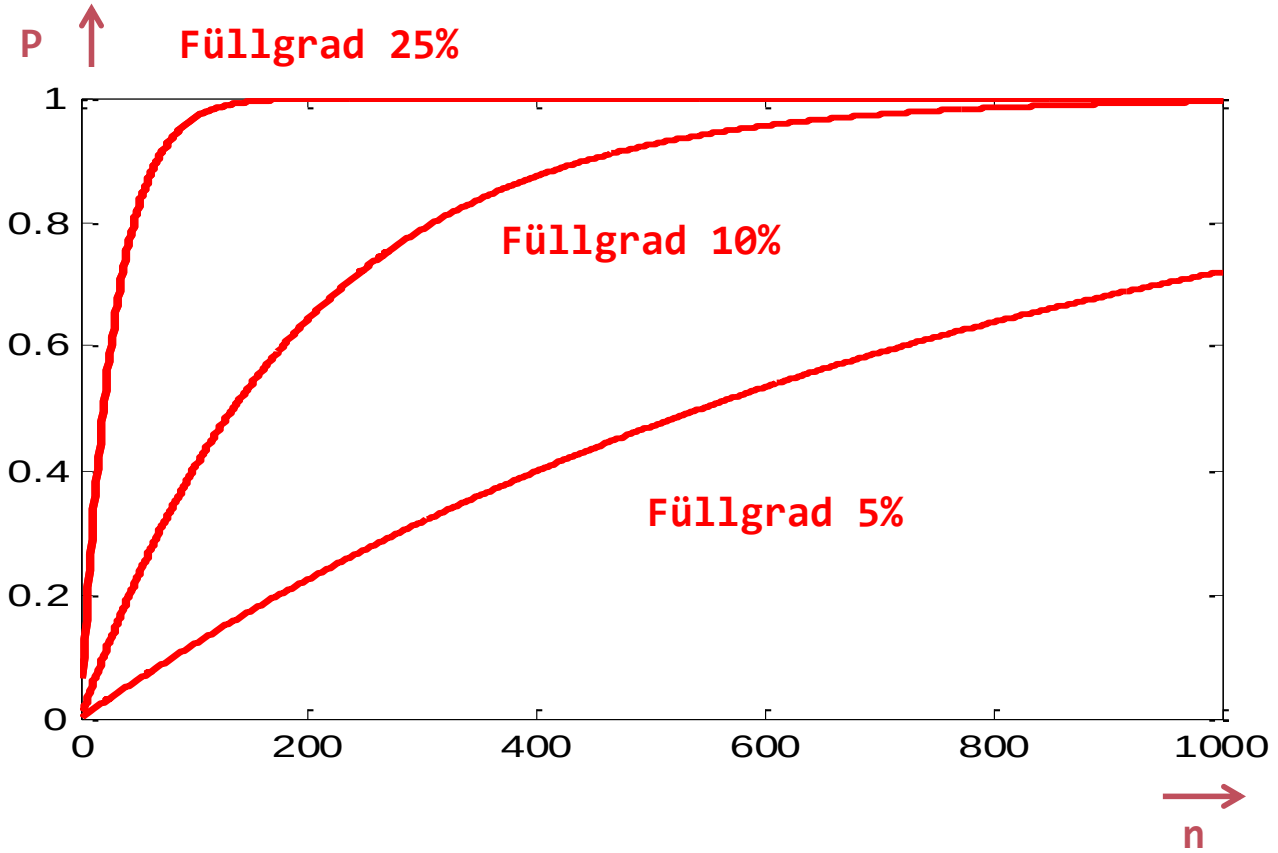
Annahme: Gleichverteilung annähernd erreicht. Wie wahrscheinlich sind dann noch Kollisionen bei einem Füllgrad von, z.B. 10%?

# 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

# Dynamische Hashtabellen

Können im Prinzip so implementiert werden wie dynamische Arrays.

Einziges Problem: Die Indexberechnung ändert sich mit der Grösse der Tabelle.

```
void Grow()
{
   Entry[] oldData = data;
   String[] oldKey = key;
   data = new Entry[data.length*2];
   key = new String[key.length*2];
   for (int i = 0; i<oldData.length; ++i)
      if (oldKey[i] != null)
         Set(oldKey[i], oldData[i]);
}
```

# 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.

| index | key |
|---|---|
| 0 | null |
| **1** | **"Hannes"** |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| **6** | **null** |
| **7** | **"Tobias"** |
| 8 | null |
| 9 | null |

| index | key |
|---|---|
| 0 | null |
| **1** | **"Hannes"** |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| **6** | **""** |
| **7** | **"Tobias"** |
| 8 | null |
| 9 | null |

# Zusammengefasst

Hash-Tabellen funktionieren, wenn eine gute Hash-Funktion vorhanden ist, welche verschiedene Schlüsselwerte möglichst auf verschiedene Array-Indizes abbildet.

Darüber hinaus muss der Indexbereich (das Array) gross genug sein, damit Kollisionen unwahrscheinlich werden.

Sind Kollisionen hinreichend unwahrscheilich, so findet man Elemente in der Hashtabelle in O(1)!

# Generische Hashtabellen in Java

Klasse für HashTabelle java.util.HashMap

```
HashMap<String, Integer> map; // Abbildung String → Integer
```

```
map.put("abc",3);
map.put("xyz",100);
int i = map.get("abc"); // i = 3
int j = map.get("xyz"); // j = 100
```