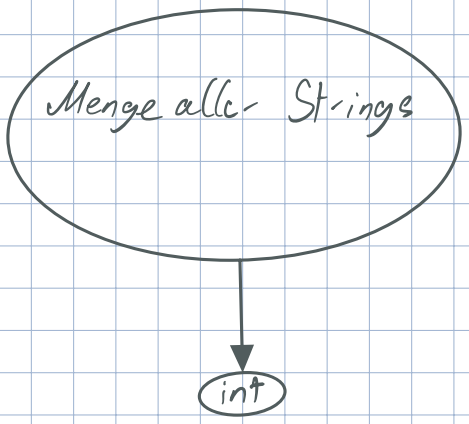
Motivation:

Schlüssel	Wert
"Tom"	0664 12345
"Hans"	0699 98765
"Paul"	...

Tripel, Paar, Eintrag

tab["Tom"]

val = tab[42]



Hash-Funktion

Abbildung von großen Wertebereich (z.B. Menge aller unkurzen Strings)

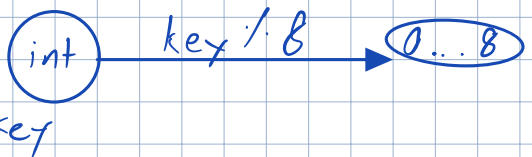
"Funktion"

auf einen kleinen Bereich (z.B. integer)

Ziel:
- gleichmäßige Streuung (wenig Kollisionen)
- einfach u. schnell zu berechnen

Forschungsgebiet machen oft ansprechend

Einfacher Fall



Lange Schlüssel (z.B. String)

A	65	01000001
n	110	01101110
t	116	01110100
o	111	01101111
n	110	01101110
		10100100111100 = 2716 %

```
int hash(String key) {
    int adr = 0;
    for(int i=0; i< key.length(); i++) {
        adr = (2* adr +key.charAt(i)) % tabSize;
    }
    return adr;
}
```

⇒ sehr gute Hash-Funktion

⇒ ABER bei langen Schlüssel langsam

Alternative:

```
key.charAt(0) = 17+key.charAt(key.length()-1)+key.length())
        % wertebereich
```

hash("Anton") == (65 + 17 + 110 + 5) % 499 == 222

Hashfunktion h = str.length() (← sehr schlecht Hashfunktion)

I)

tab["Anton"] = 1234

@

hash("Anton") = 5

@

5 % 4 = 1 ← gültiger Index

tab[1] = 1234

	key	data
[0]		
[1]	"Anton"	1234
[2]		
[3]		

II/ tab["Berta"] = 9876

↓

hash("Berta") = 5

↓

5 % 6 = 1

tab[1] = 9876 ↯ Kollision

Zusammenfassung Hash-Tabellen:

tab["Anton"] = 1234;

Performance wie Array

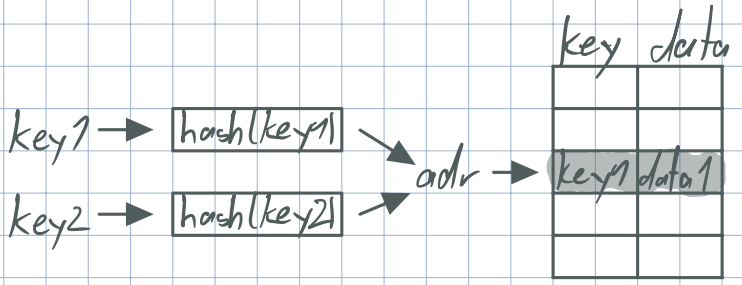
```
HashMap tab = new HashMap();
tab.put("Anton", 1234);
tab.get("Anton");
```

Kollisionstrategie:

Kollision:

hash(key1) == hash(key2)

Zum Ablegen von Kollisionen Schlüssel in Hashtabelle mitspeichern



eintragen

```
if(tab[adr].key == null){
    tab[adr].key = key;
    tab[adr].data = data;
}
```

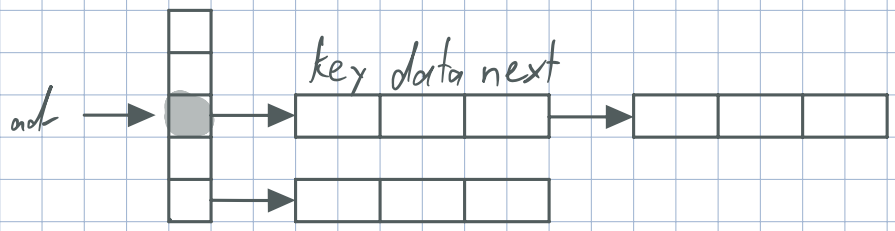
suchen:

```
if(tab[adr].key.equals(key){
    //gefunden
    data = tab[adr].data;
}else{
    // anderswo suchen
}
```

große Tabelle → wenige Kollisionen (schnell aber speicheraufwendig)
kleine Tabelle → viele Kollisionen (speichereffizient aber langsam)

Seperate Chaining

Idee: Alle Werte mit gleicher Hashadresse in gemeinsamer Liste halten




```
Data get (String key) {
    Node p = tab[hash(key)];
    while (p != null && !p.key.equals(key)) p = p.next;
    if (p != null) return p.data; else return null;
}

void put (String key, Data data) {
    int adr = hash(key);
    Node p = new Node(key, data);
    p.next = tab[adr]; tab[adr] = p; // erlaubt hier Duplikate
}
```

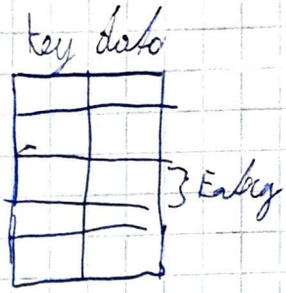
n Schlüssel, Tabellengröße m ⇒ Zugriffszeit O(n/m)

get() und put() muss man können —> Test

Linear Probing

Idee:

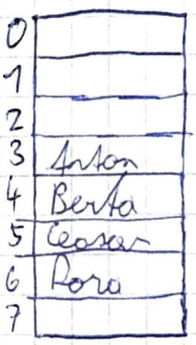
- Schlüssel und Wert direkt in Hashtabelle halten
- Wenn Kollision auf Adresse a ⇒ nächste Adresse a+1 probieren




```
void put (String key, Data data) {
  int a = hash(key);
  while (tab[a] != null)   // while besetzt
  a = (a+1) % tabSize;
  tab[a] = new Entry (key, data);
}
```

Endlosschleife, wenn Liste voll!

Schlüssel	Hashcode
Anton →	3
Berta →	4
Ceasar →	3
Dora →	5



⇒ Cluster
⇒ Viele Kollisionen

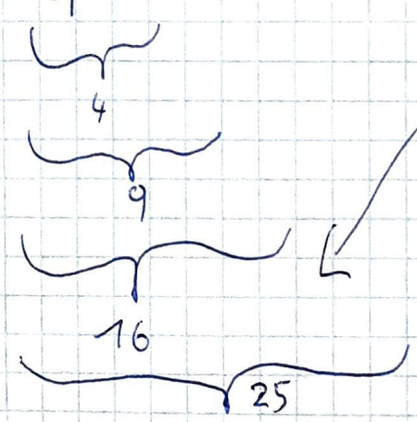
Quadratic Hashing

Idee: Versuch i auf $(a+i^2)$

Nachteil: Nur hälfte der Tabelle wird besucht

a, a+1, a+4, a+9, a+16, ...

1 3 5 7 11 // ungerade Zahlen



a	d
0	1
1	3
4	5
2	7
2	9
4	11
1	13
0	15

```
void put (String key, Data data) {
  int a = hash(key);
  int d = 1;
  while (tab[a] != null) {
    a = (a+d) % tabSize;
    d = d+2;
  }
  tab[a] = neue Entry (key, data)
}
```

Quadratic Probing

Trick um alle Listenplätze zu besuchen:

d: -tabSize ..... +tabSize in 2er Schritten
tabSize muss Primzahl sein

```
void put (String key, Data data) {
int a = hash (key);
int d = - tabSize;
while (tab[a] != null) {
  d = d+2;
  a=(a + Math. abs (d)) % tabSize;
}
  tab[a] = new Entry (key, data);
}
```

a	d
0	-7
5	-5
1	-3
2	-1
3	1
6	3
4	5
4	7

Wenn Hashcode überschreiben ⇒ equals auch überschreiben