🟦 Hashing & Hash-Tabellen

1️⃣ Grundidee

Hashing ist eine Technik, um Daten mithilfe einer **Hash-Funktion** schnell in einer Tabelle zu speichern und wiederzufinden.

Eine **Hash-Tabelle** ordnet jedem Schlüssel einen **Index** zu:

```
Index = hash(key)
```

Abgespeichert wird jeweils der Wert. Die Hashfunktion liefert den Index der Tabelle, wo der Wert abgelegt werden soll.

➡️ Ziel: **konstante Zugriffszeit O(1)** im Durchschnitt.

2️⃣ Voraussetzungen

- Eine **Hash-Funktion**, die Schlüssel auf Tabellenindizes abbildet
- Speicherplatz für die Hash-Tabelle
- Strategie zur **Kollisionsbehandlung**

3️⃣ Laufzeiten & Eigenschaften

Eigenschaft	Wert
Suche (Average)	O(1)
Einfügen (Average)	O(1)
Löschen (Average)	O(1)
Worst Case	O(n)
Speicherbedarf	O(n)
In-place	nein
Stabil	–

Hinweis: Der Worst Case tritt auf, wenn viele Schlüssel auf denselben Index abgebildet werden.

4️⃣ Hash-Funktion

Eine gute Hash-Funktion sollte:

- **deterministisch** sein
- Schlüssel **gleichmäßig verteilen**
- schnell berechenbar sein

Beispiel:

```
hash(key) = key mod m
```

5️⃣ Kollisionen & Kollisionsbehandlung

Was ist eine Kollision?

Zwei verschiedene Schlüssel werden auf **denselben Index** abgebildet.

Typische Strategien (klassischer Prüfungsstoff)

1. Verkettung (Chaining)

- Jeder Tabellenplatz enthält eine **Liste**
  - Bei Kollisionen werden alle Einträge in der Liste gespeichert
- Kollisionen werden angehängt
  - Beispiel: Wenn Index 3 bereits belegt ist, wird der neue Eintrag einfach an die Liste an Index 3 angehängt.

2. Offene Adressierung

- Lineares Probing (Lineares Sondieren)
  - Suche nach dem nächsten freien Platz => wenn index bereits belegt ist, wird der nächste Index verwendet. Ist der letzte Index des Arrasy ebenfalls bereits belegt, wird am Anfang des Arrays weitergesucht.
- Quadratisches Probing (Quadratisches Sondieren)
  - Suche nach dem nächsten freien Platz mit quadratischer Schrittweite => wenn index bereits belegt ist, wird der Index + i^2 verwendet (i = Anzahl der Versuche). Ist der letzte Index des Arrasy ebenfalls bereits belegt, wird am Anfang des Arrays weitergesucht.
- Double Hashing (Doppeltes Hashing)
  - Verwenden einer zweiten Hash-Funktion zur Bestimmung des nächsten Index => wenn index bereits belegt ist, wird der Index + hash2(key) verwendet. Ist der letzte Index des Arrasy ebenfalls bereits belegt, wird am Anfang des Arrays weitergesucht.

Performanceunterschied bei unterschiedlichen Hash-Funktionen

Wenn die Hashfunktionen schlecht gewählt sind und viele Kollisionen verursachen, kann die Performance deutlich schlechter ausfallen (bis zu O(n) im Worst Case). Eine gute Hashfunktion minimiert Kollisionen und sorgt für eine gleichmäßige Verteilung der Schlüssel.

Beispiel aus Musterprüfung: h(x) = (2x + 3) mod 19 <=> h'(x) = (3x + 5) mod 19
Da 2 und 3 beide teilerfremd zu 19 sind, wird eine gleichmäßige Verteilung der Schlüssel erreicht. Also keine wesentliche Änderung der Performance.

Mathematische Begründung:\

- 19 ist eine Primzahl.\
- **gcd(2, 19) = 1** und **gcd(3, 19) = 1** (2 und 3 sind teilerfremd zu 19).\
- Beide Hashfunktionen erzeugen eine vollständige Abbildung auf die Indizes 0 bis 18.\
- Daher bleibt die Verteilung der Schlüssel über die Tabelle gleichmäßig, und die Performance ändert sich nicht wesentlich.

Wenn jedoch eine Hashfunktion verwendet wird, die nicht teilerfremd zur Tabellengröße ist, kann dies zu einer schlechten Verteilung führen und die Performance verschlechtern.

Beispiel: h(x) = (6x + 3) mod 18\

- 18 ist keine Primzahl.\
- gcd(6, 18) = 6 (also nicht gleich 1).\
- Diese Hashfunktion kann nur Indizes erzeugen, die Vielfache von 3 sind (0, 3, 6, 9, 12, 15).\
- Dadurch werden viele Indizes in der Tabelle ungenutzt bleiben, was zu einer schlechten Verteilung der Schlüssel führt und die Performance erheblich verschlechtert.

6️⃣ Besonderheiten / Prüfungsrelevante Hinweise

- Load Factor: Wie voll ist die Tabelle?
- **Load Factor α = n / m** (Anzahl Elemente / Tabellengröße)
- Hoher Load Factor → mehr Kollisionen
- Rehashing bei Überschreiten eines Schwellwerts
- Hash-Tabellen sind **nicht sortiert**

7️⃣ Vor- und Nachteile

Vorteile

- sehr schneller Zugriff

- ideal für Nachschlagen (Dictionaries, Sets)
- einfache API

Nachteile

- keine Ordnung der Elemente
- Worst Case O(n)
- Speicher-Overhead

🧠 Merksatz für die Prüfung

Hashing ermöglicht schnellen Zugriff über Schlüssel, erfordert aber gute Hash-Funktionen und Kollisionsstrategien.

8️⃣ Python-Implementierung (Hash-Tabelle mit Verkettung)

In [1]:
```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                return
        self.table[index].append((key, value))

    def search(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None

    def delete(self, key):
        index = self._hash(key)
        for i, (k, _) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                return True
        return False
```

```python
# Beispiel
ht = HashTable()
ht.insert("Alice", 25)
ht.insert("Bob", 30)

print(ht.search("Alice"))  # 25
print(ht.search("Bob"))  # 30
```

```
25
30
```

🟦 Bloomfilter

1️⃣ Grundidee

Ein **Bloomfilter** ist eine **probabilistische Datenstruktur**, mit der effizient geprüft werden kann, ob ein Element **sicher nicht enthalten** oder **möglicherweise enthalten** ist.

- Sehr **speichereffizient**
- Erlaubt **False Positives**, aber **keine False Negatives**
- Basierend auf **mehreren Hash-Funktionen** und einem Bit-Array

---

2️⃣ Voraussetzungen

- Ein **Bit-Array** der Größe m
- **k Hash-Funktionen**
- Hash-Funktionen liefern gleichmäßig verteilte Indizes

3️⃣ Laufzeiten & Eigenschaften

Eigenschaft	Wert
Einfügen	O(k)
Abfrage	O(k)
Löschen	nicht möglich
Speicherbedarf	O(m)
In-place	nein
Stabil	–

Hinweis: k = Anzahl Hash-Funktionen, m = Größe des Bit-Arrays

---

4️⃣ Funktionsweise (Schritt-für-Schritt)

Einfügen eines Elements

1. Element wird mit **k Hash-Funktionen** gehasht
2. Die resultierenden Indizes werden im Bit-Array auf **1** gesetzt

Abfrage eines Elements

- Ist **mindestens ein Bit = 0** → Element **sicher nicht enthalten**
- Sind **alle Bits = 1** → Element **möglicherweise enthalten**

5️⃣ Schritt-für-Schritt-Beispiel

Bit-Array (m = 10):

```
[0 0 0 0 0 0 0 0 0 0]
```

Einfügen von "Alice" (Hash-Indizes: 2, 5, 7):

```
[0 0 1 0 0 1 0 1 0 0]
```

Abfrage von "Bob" (Hash-Indizes: 1, 5, 9):

- Bit an Index 1 = 0 → **Bob sicher nicht enthalten**

Abfrage von "Alice":

- Alle Bits = 1 → **Alice möglicherweise enthalten**

6️⃣ Besonderheiten / Prüfungsrelevante Hinweise

- **False Positives** sind möglich
- **False Negatives sind unmöglich**
- Je größer m und je besser k gewählt wird, desto kleiner die Fehlerwahrscheinlichkeit
- Sehr häufige Prüfungsfrage: *Warum kann man nicht löschen?*

7️⃣ Vor- und Nachteile

Vorteile

- extrem speichereffizient

- sehr schnelle Abfragen
- ideal für große Datenmengen

Nachteile

- keine exakte Mitgliedschaft
- kein Löschen möglich
- False Positives

🧠 Merksatz für die Prüfung

Ein Bloomfilter kann sicher sagen, dass ein Element nicht enthalten ist, aber nur probabilistisch, dass es enthalten sein könnte.

🎱 Python-Implementierung (einfacher Bloomfilter)

In [2]:
```python
class BloomFilter:
    def __init__(self, size=10, hash_count=3):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = [0] * size

    def _hashes(self, item):
        hashes = []
        for i in range(self.hash_count):
            hashes.append(hash((item, i)) % self.size)
        return hashes

    def add(self, item):
        for index in self._hashes(item):
            self.bit_array[index] = 1

    def contains(self, item):
        return all(self.bit_array[index] == 1 for index in self._hashes(item))


# Beispiel
bf = BloomFilter(size=10, hash_count=3)
bf.add("Alice")

print(bf.contains("Alice"))  # True (wahrscheinlich)
print(bf.contains("Bob"))  # False (sicher)
```

```
True
False
```