🟦 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)
```

➡️ 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**
- Kollisionen werden angehängt

2. Offene Adressierung

- Lineares Probing
- Quadratisches Probing
- Double Hashing

6️⃣ Besonderheiten / Prüfungsrelevante Hinweise

- **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


# Beispiel
ht = HashTable()
```

```python
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
```