Datenstrukturen

Andreas Hohenauer

Algorithmen und Datenstrukturen

12. September 2025

Grundbegriffe

- Einfachster Datentyp: Variable
- Komplexere Datentypen: Klassen und Objekte
- Organisation mehrerer Elemente: Datenstrukturen

Grundbegriffe

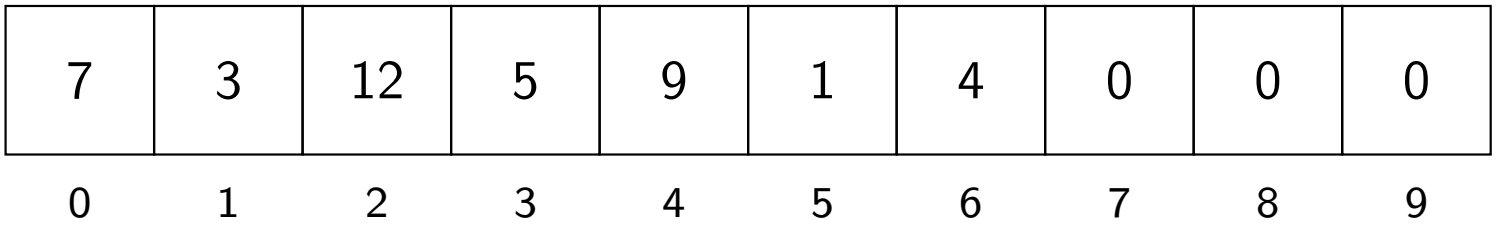
- Lineare Datenstrukturen
  - Array
  - Lists
  - Stack
  - Queue
- Dictionaries / Maps
- Sets (Mengen)
- Graphenbasierte Datenstrukturen:
  - Bäume
  - allgemeine Graphen

Array

- Zusammenhängende Anordnung mehrerer Elemente im Speicher
- Zugriff auf bestimmte Position mit Index $i = 0, 1, 2, \ldots$
- Bei vielen Sprachen existiert Array als primitiver Datentyp
- Operationen wie *Einfügen* oder *Löschen* werden dann in höheren Datentypen umgesetzt.
- Beispiel Java: Array `[]` versus `ArrayList`

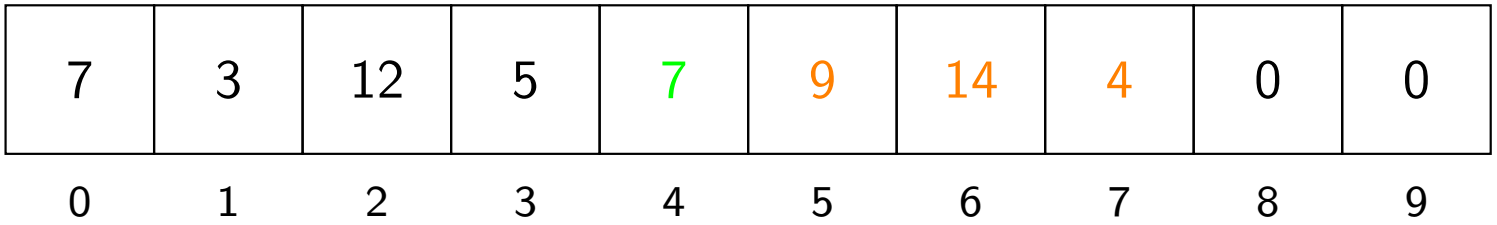
Array

Array mit 10 Elementen (Index unter den Zellen).



Array

Array nach dem Einfügen eines Elements 7 an der Stelle mit Index 4. Die Elemente 9, 14, und 4 wurden um eine Position nach rechts verschoben.



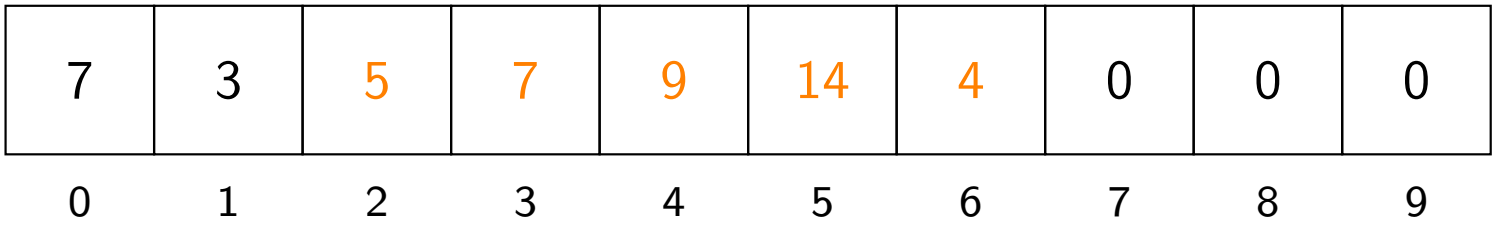
Array

Nun soll das Element an der Stelle mit Index 2 gelöscht werden.

7	3	12	5	7	9	14	4	0	0
0	1	2	3	4	5	6	7	8	9

Array

Die Elemente 12, 5, 7, 9, 14 und 4 wurden um eine Position nach links verschoben.



Array (Analyse)

Laufzeit der wichtigsten Operationen:

- Zugriff mit Index: $O(1)$
- Einfügen: $O(n)$
- Entfernen: $O(n)$

Array (Python)

Interne Umsetzung von list in Python

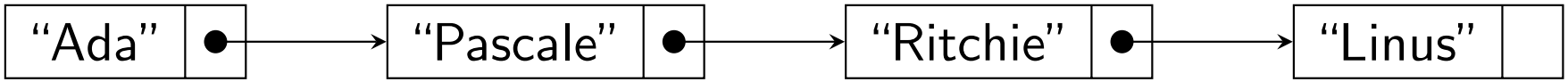
- Intern ist eine `list` ein dynamisches Array (also im Speicher ein zusammenhängender Block).
- Beim Erweitern wird manchmal mehr Speicher reserviert als gerade benötigt, um häufige Kopieraktionen zu vermeiden.
- Indexzugriff O(1), aber das Einfügen in der Mitte O(n) (weil alle folgenden Elemente nach rechts verschoben werden müssen).

Operationen und ihre Komplexität

- lst.append(x) $\rightarrow$ Am Ende einfügen, amortisiert O(1).
- lst.insert(i, x) $\rightarrow$ An Position i einfügen, O(n), weil Elemente verschoben werden.
- lst[i] $\rightarrow$ Zugriff per Index, O(1).
- lst.remove(x) oder del lst[i] $\rightarrow$ ebenfalls O(n) im Worst Case (weil Verschieben nötig ist).

List

- Elemente liegen verteilt im Speicher
- Werden miteinander zur Zeiger/Pointer verkettet
- Einfach- vs. doppelt verketteter Liste



List (Analyse)

Laufzeit der wichtigsten Operationen:

- Zugriff Index: $O(n)$
- Einfügen an beliebiger Stelle: $O(n)$
- Löschen: $O(n)$
- Zugriff/Einfügen/Löschen mit Iterator: $O(1)$

List (Python)

- Standardmäßig gibt es in Python keine verkettete Liste.
- Jedoch bietet die Bibliothek `collections.deque` eine Implementierung.
- deque ist eine doppelt verkettete Liste, optimiert für Einfügen/Entfernen am Anfang/Ende in $O(1)$.
- Aber: Kein effizienter Random Access (Indexzugriff ist $O(n)$).

Stack

Ein **Stack** folgt dem Prinzip **Last In — First Out (LIFO)**: Das zuletzt eingefügte Element wird als erstes wieder entfernt. Typische Operationen sind:

- `push`: Einfügen eines Elements oben auf den Stack.
- `pop`: Entfernen des obersten Elements.
- `peek`/`top`: Zugriff auf das oberste Element, ohne es zu löschen.

Stacks lassen sich einfach mit Arrays oder verketteten Listen realisieren und werden z. B. bei Funktionsaufrufen (Call Stack) oder in Undo-Mechanismen eingesetzt.

Queue

Eine **Queue** (Warteschlange) folgt dem Prinzip **First In – First Out (FIFO)**: Das zuerst eingefügte Element wird auch als erstes wieder entfernt. Typische Operationen sind:

- `enqueue`: Einfügen eines Elements am Ende der Queue.
- `dequeue`: Entfernen des ersten Elements.
- `front`: Zugriff auf das erste Element, ohne es zu löschen.

Queues werden oft in Betriebssystemen (Prozessplanung), Netzwerken (Datenpaketverwaltung) oder Druckersystemen verwendet.

List/Queue (Python)

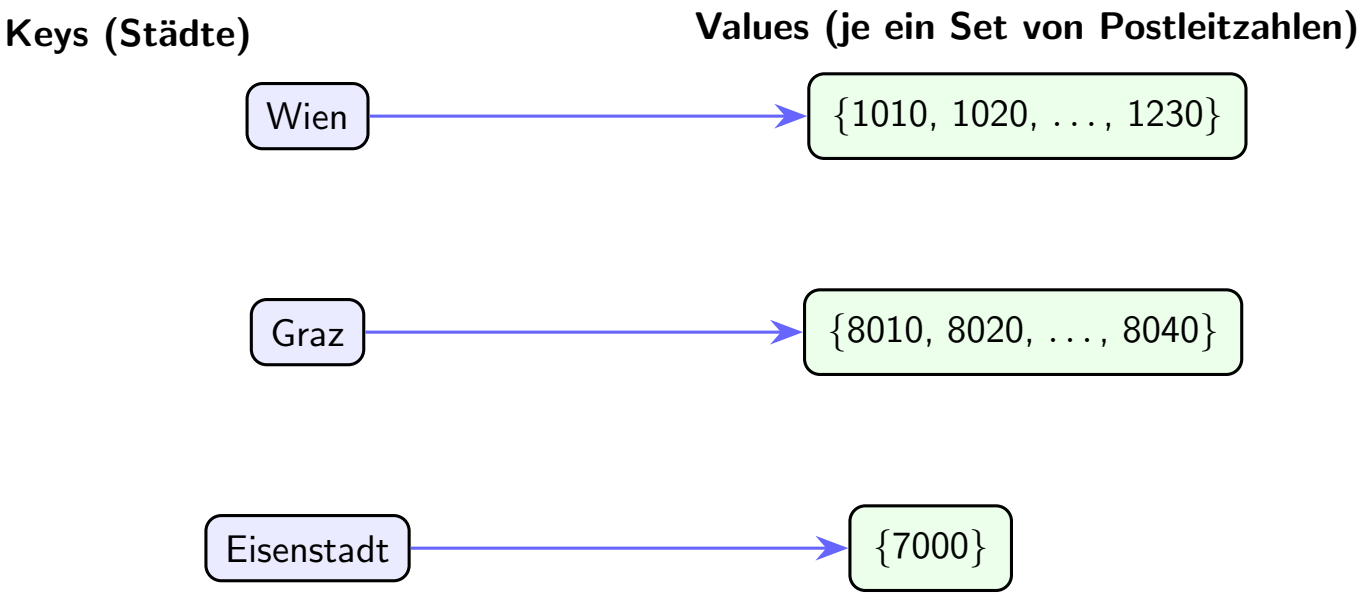
- In Python lässt sich ein **Stack** einfach mit einer normalen Liste implementieren:
  - `append()` entspricht `push`.
  - `pop()` entfernt das oberste Element.
- Eine **Queue** lässt sich mit `collections.deque` effizient umsetzen:
  - `append()` fügt ein Element hinten an (`enqueue`).
  - `popleft()` entfernt das vorderste Element (`dequeue`).
- Vorteil von `deque`: Operationen am Anfang *und* Ende laufen in $O(1)$.

Dictionaries/Maps

- Dictionaries oder Maps **Schlüssel – Wert**-Paare.
- Oft möchte man gespeicherte Objekte *schnell* aufgrund eines *Schlüssels* (id, name, ...) finden, bzw. darauf zugreifen.
- Beispiel:
  - Finde Mitarbeiter mit Personalnummer
  - Finde Immobilie mittels Anzeigennummer
- Das Suchen in Arrays, ArrayLists etc. ist aufwändig, da die Datenstruktur zunächst durchsucht werden muss.
- **Ziel:** Auffinden eines Elementes aufgrund des **Schlüssels (Key)** in (quasi) konstanter Zeit $O(1)$, d.h. ohne viele Elemente in der Datenstruktur betrachten zu müssen.

Dictionaries/Maps

Beispiel: Es werden Städten (=Schlüssel) die Mengen von Postleitzahlen (=Werte) zugeordnet. Der Wert ist hier also eine Menge (Datenstruktur Set).



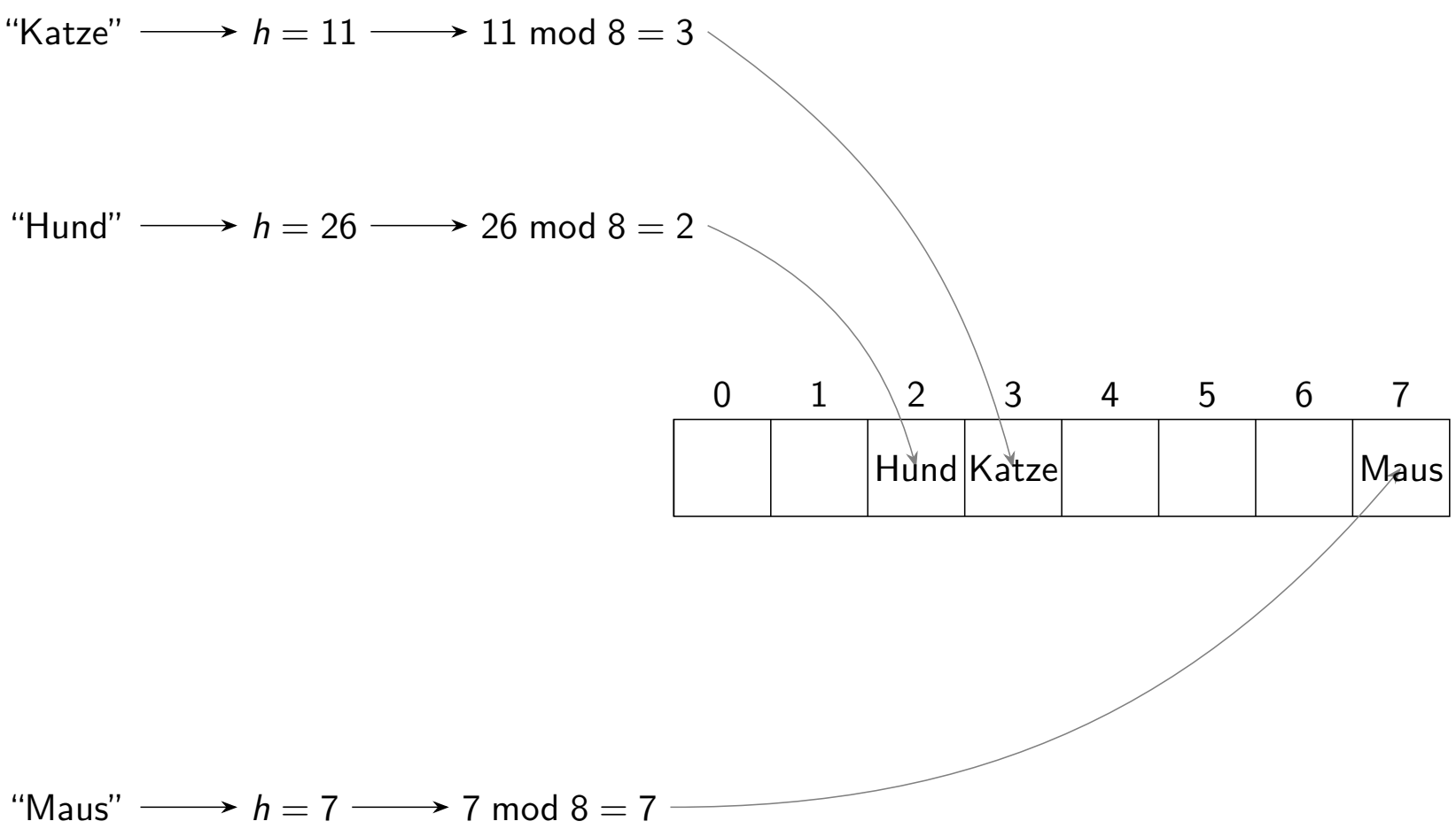
Hashfunktionen und Hashtabellen

- Mittels *Hashfunktionen* wird zu einem Schlüssel ein Hashwert berechnet, der als Index in einer *Hashtabelle* dient.
- Schlüssel (z.B. Strings) sollen schnell auf Speicherplätze (Buckets) einer Tabelle abgebildet werden.
- Dazu wird eine *Hashfunktion* h verwendet, die einen Schlüssel k auf einen ganzzahligen Index $h(k) \in \{0, \ldots, m-1\}$ (Tabellengröße m) projiziert.
- Hashtabellen verbinden eine (große) Array-Struktur fester Länge m mit einer Hashfunktion. Jeder Eintrag (Bucket) kann (abhängig vom Verfahren) direkt ein Element oder eine kleine Sekundärstruktur (Liste/Baum) enthalten. Ziel: Erwartete $O(1)$ Zeit für Suchen, Einfügen und Löschen.

Anforderungen an eine Hashfunktion

- Deterministisch: Gleicher Schlüssel $\Rightarrow$ gleicher Hashwert.
- Effizient: Berechnung in (amortisiert) $O(1)$ und sehr schnell (wenige CPU-Operationen).
- Gleichmäßige Verteilung: Schlüssel sollen möglichst gleich über $\{0, \ldots, m-1\}$ verteilt werden (minimiert Kollisionen).
- Ähnliche Schlüssel $\Rightarrow$ ähnliche Hashwerte (Vermeidung von Clustern).
- Stabil bzgl. Lebensdauer der Tabelle (aber: in manchen Sprachen kann sich Hash zwischen Programmläufen absichtlich ändern, z.B. Python wegen Sicherheit).

Grundidee einer Hashtabelle



Hashtabellen: Grundbegriffe

- *Kollision*: Zwei verschiedene Schlüssel $k_1 \neq k_2$ mit $h(k_1) = h(k_2)$.
- *Lastfaktor* (load factor) $\alpha = \frac{n}{m}$ mit $n =$ Anzahl gespeicherter Elemente.
- Rehashing: Vergrößern der Tabelle (typisch Faktor 2) und Neuverteilen aller Elemente, sobald α einen Schwellwert überschreitet.

Hashfunktionen

Hashfunktion

$$h(k) = h_{\text{raw}}(k) \bmod m$$

Dabei produziert h_{raw} einen großen Integer (z.B. durch Kombination von Zeichen / Feldern), das Modulo faltet den Wert in den Tabellenbereich.

Beispiel: Einfache Beispiele:

- Strings: Polynomielles Rolling-Hash $h_{\text{raw}}(s_0 s_1 \dots s_{L-1}) = \sum_{i=0}^{L-1} s_i \cdot B^{L-1-i}$ mit Basis B (z.B. 131, 257) und ggf. Reduktion mit großem Prim P vor Modulo m.
- Integer: Mischung (Bit-Mixing), z.B. Multiplikation mit großer ungerader Konstante und Bit-Shifts.

Kollisionsbehandlung (Überblick)

- *Getrennte Verkettung* (Chaining): Jeder Bucket hält eine (kurze) Liste / Baum der kollidierenden Elemente.
- *Offene Adressierung* (Sondieren): Bei Kollisionen wird versucht das Element an einer anderen Stelle im Array abzulegen (Sondieren).

Hashfunktionen

Qualität der Hashfunktion:

- Schlechte Verteilung führt zu langen Listen (Chaining) bzw. langen Suchfolgen (Sondieren) $\Rightarrow$ Performance-Verlust.
- Eine (annähernd) uniforme Verteilung hält erwartete Kosten pro Operation bei $O(1)$ solange α kontrolliert bleibt.

Immutabilität von Schlüsseln:

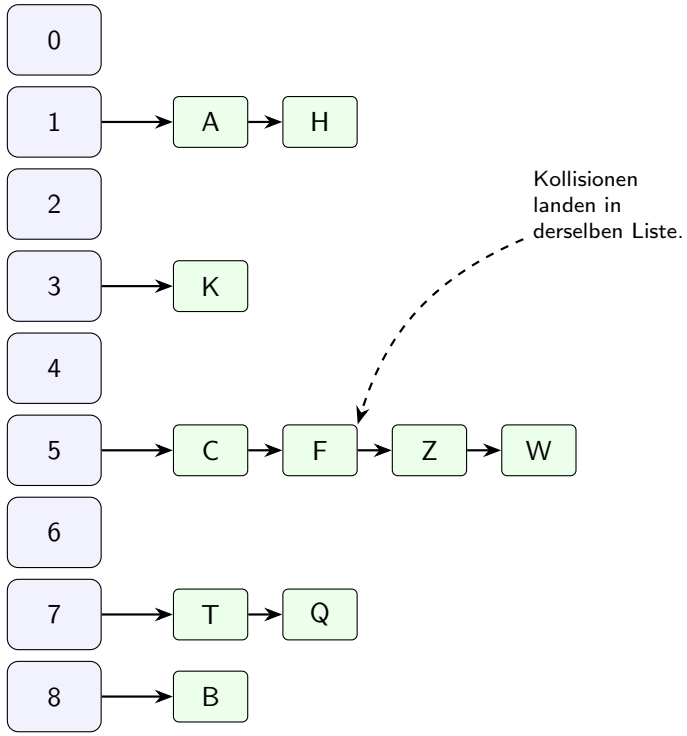
Ändert sich ein Schlüssel nach dem Einfügen (z.B. veränderbares Objekt mit hash-basiertem Wert), kann er nicht mehr korrekt gefunden werden. Daher müssen Schlüssel (logisch) unveränderlich sein oder ihr Hash / Vergleich darf sich nicht ändern.

Methode 1: Verkettung der Überläufer

- Array $T[0 .. m-1]$; jeder Bucket hält z.B. eine (kurze) verkettete Liste der dort liegenden Schlüssel/Werte.
- Insert: Finde $i = h(k)$, hänge (k,v) an Liste $T[i]$ an (falls Schlüssel schon da: Wert ersetzen).
- Lookup: Suche in Liste $T[h(k)]$ nach Schlüssel.
- Delete: Entferne Element aus dieser Liste.
- Erwartete Länge einer Liste: $\alpha = n/m$ (Lastfaktor) bei guter Hashfunktion $\Rightarrow$ erwartete Kosten $O(1 + \alpha)$.

Methode 1: Verkettung der Überläufer

Hashtabelle mit Verkettung (Chaining): Jeder Bucket zeigt (falls nicht leer) auf den Kopf einer Liste kollidierender Elemente. Die erwartete Listenlänge entspricht dem Lastfaktor $\alpha = n/m$.



Methode 2: Offene Hashverfahren (Sondieren)

Die offenen Hashverfahren verfolgen eine andere Strategie im Falle von Kollisionen. Statt die kollidierenden Elemente in einer Liste zu speichern, werden sie in der Tabelle selbst an der nächsten geeigneten freien Stelle abgelegt. Die nächste geeignete Stelle zu berechnen wird als Sondieren bezeichnet.

Bei belegtem Platz wird eine neuer Platz wie folgt berechnet

$$h_i(k) = (h(k) + f(i)) \bmod m, \quad i = 0, 1, 2, \ldots$$

Verwendet man als Funktion $f(i) : i$, so nennt man dies *lineares Sondieren*. Verwendet man hingegen $f(i) = c_1 i + c_2 i^2$, wird dies als *quadratisches Sondieren* bezeichnet.

Methode 3: Double Hashing

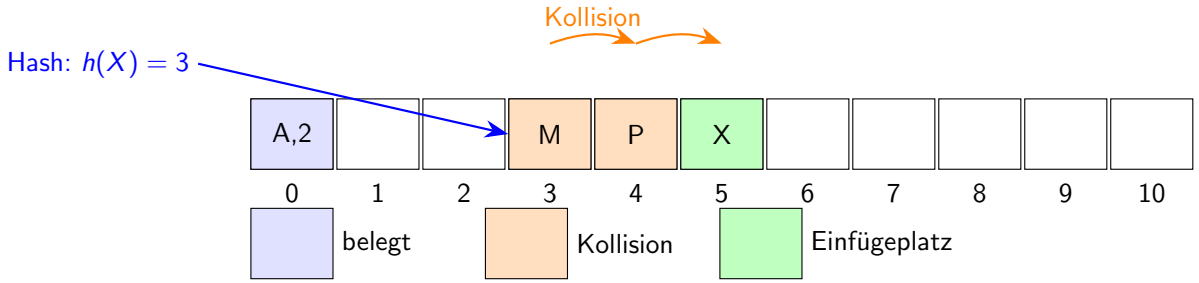
Eine weitere Möglichkeit ist Double Hashing: $h_i(k) = h_1(k) + i \cdot h_2(k)$. Hierbei werden zwei verschiedene Hashfunktionen h_1 und h_2 verwendet, um die Sondierfolge zu bestimmen. Dies kann helfen, Kollisionen weiter zu reduzieren und eine bessere Verteilung der Elemente in der Hashtabelle zu erreichen. Die Wahl von h_2 ist hierbei jedoch kritisch, ebenso ist das Verfahren aufwändiger als lineares oder quadratisches Sondieren.

Vor- und Nachteile der Verfahren:

- Chaining: Einfach, Performance robust, benötigt Zeiger/Overhead.
- Offene Adressierung: Cache-freundlich (alles im Array), aber sensibler gegenüber hoher Auslastung; Clusterbildung möglich.

Illustration: lineares Sondieren

Einfügen von Schlüssel X mit Hash 3: Position 3 ist belegt, lineares Sondieren prüft 4 (auch belegt), findet 5 frei und legt dort ab. Die Anzahl besuchter Buckets hängt vom Cluster vor der Zielposition ab; geringe Auslastung minimiert durchschnittliche Sondenlänge.



Beispiel: lineares Sondieren

$$m = 8, \qquad h'(k) = k \bmod 8$$

Einzufügende Schlüssel (in dieser Reihenfolge): $10, 19, 31, 22, 14, 16$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

Schrittweises Einfügen (lineares Sondieren)

- 10: $h'(10) = 2 \Rightarrow$ Slot 2 frei $\Rightarrow$ einfügen.
- 19: $h'(19) = 3$ frei $\Rightarrow$ Slot 3.
- 31: $h'(31) = 7$ frei $\Rightarrow$ Slot 7.
- 22: $h'(22) = 6$ frei $\Rightarrow$ Slot 6.
- 14: $h'(14) = 6$ Kollision bei 6 (22) $\Rightarrow$ prüfe 7 (31, belegt) $\Rightarrow$ prüfe 0 (frei) $\Rightarrow$ Slot 0.
- 16: $h'(16) = 0$ Kollision bei 0 (14) $\Rightarrow$ prüfe 1 (frei) $\Rightarrow$ Slot 1.

Endzustand der Hashtabelle

Index	0	1	2	3	4	5	6	7
Wert	14	16	10	19			22	31

Beispiel: lineares Sondieren

Beobachtung: Die Clusterbildung zeigt sich daran, dass die Kollision von 14 (Start bei 6) die Sondenfolge bis zum Anfang (Wrap-Around) verlängert; 16 trifft danach sofort erneut auf eine belegte Zelle. Längere zusammenhängende belegte Abschnitte wachsen schneller (Primär-Cluster).

Beispiel: quadratisches Sondieren

$m = 8, \qquad h'(k) = k \bmod 8, \qquad h_i(k) = \left(h'(k) + i + i^2\right) \bmod 8 \quad (i = 0, 1, 2, \ldots;\ c_1 = c_2 = 1)$

Einzufügende Schlüssel (in dieser Reihenfolge): $10, 19, 31, 22, 14, 16$

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

Schrittweises Einfügen (quadratisches Sondieren)

- 10: $h'(10) = 2$ frei $\Rightarrow$ Slot 2.
- 19: $h'(19) = 3$ frei $\Rightarrow$ Slot 3.
- 31: $h'(31) = 7$ frei $\Rightarrow$ Slot 7.
- 22: $h'(22) = 6$ frei $\Rightarrow$ Slot 6.
- 14: $h'(14) = 6$ belegt (22). Sondenfolge: $i = 1 : 6 + 1 + 1 = 8 \equiv 0$ frei $\Rightarrow$ Slot 0.
- 16: $h'(16) = 0$ belegt (14). Sondenfolge: $i = 1 : 0 + 1 + 1 = 2$ belegt (10); $i = 2 : 0 + 2 + 4 = 6$ belegt (22); $i = 3 : 0 + 3 + 9 = 12 \equiv 4$ frei $\Rightarrow$ Slot 4.

Endzustand der Hashtabelle

Index	0	1	2	3	4	5	6	7
Wert	14		10	19	16		22	31

Beispiel: quadratisches Sondieren

Beobachtung: Kollisionen (z.B. Startindex 6 für 22 und 14) führen zu Sprüngen, die kein zusammenhängendes primäres Cluster erzeugen (Slots: $6 \rightarrow 0$). Dennoch teilen alle Schlüssel mit gleichem Start-Hash dieselbe Sondenfolge (sekundäre Cluster). Mit $m = 8$ (Potenz von 2) muss geeignete Wahl von c_1, c_2 getroffen werden, um genügend unterschiedliche Slots zu erreichen.

Python: Voraussetzungen

- Objekte sind als Dictionary-Schlüssel nutzbar, wenn sie `__hash__` und `__eq__` konsistent implementieren.
- Eingebaute unveränderliche Typen (int, str, tuple mit unveränderlichen Elementen) sind hashbar.

Zusammenfassung und Analyse

Laufzeitanalyse von Dictionaries:

- Erfolgreiche Suche: amortisiert $O(1)$
- Einfügen: amortisiert $O(1)$
- Löschen: amortisiert $O(1)$
- Worst Case (pathologisch viele Kollisionen): $O(n)$

Konsistenzanforderungen:

- Hash und Gleichheit müssen konsistent sein: $k_1 = k_2 \Rightarrow h(k_1) = h(k_2)$.
- Schlüssel dürfen sich nach Einfügen nicht (logisch) ändern.

Zusammenfassung:

Eine gute Hashfunktion ist schnell, verteilt Schlüssel gleichmäßig und minimiert so Kollisionen. Kollisionsbehandlung (Chaining oder offene Adressierung) plus kontrollierter Lastfaktor sichern amortisierte $O(1)$-Operationen für Insert, Lookup und Delete.

Python-Dicts

Python-Dictionaries und -Sets nutzen eine optimierte Form offener Adressierung mit perturbierter Sondenfolge und speichern (kompakt) Hash und Schlüssel/Wert zusammen. Löschungen hinterlassen spezielle Marker, periodisches Resizing räumt auf. Iterationsreihenfolge (Einfügereihenfolge stabil) ist implementierungs- bedingt, aber praktisch nutzbar.

```
# Erzeugen
d = {}                                      # leeres Dict
person = {"name": "Ada", "age": 37}

# Zugriff / Einfügen / Überschreiben
person["city"] = "London"                   # Einfügen
person["age"] = 38                          # Überschreiben
print(person["name"])                       # Direktzugriff
```

Python-Dicts

```
# Sicherer Zugriff
print(person.get("email"))                  # -> None
print(person.get("email", "kein Eintrag"))

# Existenztest
if "age" in person:
    ...

# Entfernen
del person["city"]                          # KeyError falls fehlt
age = person.pop("age")                     # entfernt + liefert Wert
val = person.pop("email", None)             # Default falls Key fehlt
k, v = person.popitem()                     # entfernt letztes (LIFO, 3.7+)

# Iteration
for k in person: ...                        # Keys
for k, v in person.items(): ...
for v in person.values(): ...
```

Sets

- Ein *Set* entspricht einer mathematischen Menge.
- Gleiche Elemente können in einem Set nur einmal enthalten sein.
- In einem Set kann man (im Gegensatz zur `ArrayList`) nicht mittels Index auf bestimmte Elemente zugreifen.
- Operationen auf Sets:
  - Hinzufügen eines Elementes
  - Entfernen eines Elementes
  - Überprüfen ob ein Element im Set enthalten ist

Anmerkung: Die konkrete Umsetzung der oben definierten Anforderungen an die Datenstruktur Set kann auf verschiedene Arten erfolgen. Eine sehr gebräuchliche und effiziente Methode ist die Verwendung von Hashtabellen aber auch baum-basierte Sets sind möglich.

Sets mit Hashtabelle

- Jedes Element wird durch seinen Hashwert in eine bestimmte Position (Bucket) der Hashtabelle eingefügt.
- Dies ermöglicht einen schnellen Zugriff auf die Elemente, da die Suche, das Hinzufügen und das Entfernen im Durchschnitt in konstanter Zeit $O(1)$ erfolgen können.
- Vergleiche mit naiver Implementierung als Liste: jeder Zugriff $O(n)$ Zeit

Datentyp `set` (intern als hash-basiertes Set umgesetzt)
Wichtige Operationen:

- `s = 1, 2, 3` → Erzeugung durch Literal
- `s = set([1, 2, 3])` → Erzeugung aus Liste
- `s.add(x)` → fügt ein Element hinzu (tut nichts, wenn schon vorhanden).
- `s.remove(x)` → entfernt ein Element, KeyError, falls nicht vorhanden.
- `s.discard(x)` → entfernt ein Element, ohne Fehler, falls nicht vorhanden.
- `s.pop()` → entfernt & gibt ein beliebiges (meist erstes) Element zurück.
- `s.clear()` → leert das Set.

Eigenschaften:
Die Elemente sind im Set einzigartig. Es gibt keine garantierte Reihenfolge (bis Python 3.6 war die Iterationsreihenfolge zufällig, seit Python 3.7 ist sie stabil und entspricht der Einfüge-Reihenfolge – ähnlich wie LinkedHashSet in Java, aber das ist nur ein Implementierungsdetail, kein Ordnungsversprechen). Zugriff, Einfügen und Entfernen sind im Schnitt O(1).

Set mittels Bäumen

- Umsetzung des *Set*-Konzeptes mittels *Bäumen*.
- Elemente können in $O(\log n)$ gesucht, eingefügt und gelöscht werden.
- Zusatznutzen: die Elemente sind immer sortiert!
- Zugriff auf nächst- größeres/kleineres Element schnell möglich

In Python gibt es standardmäßig kein TreeSet, jedoch kann man die Funktionalität mit der Bibliothek `sortedcontainers` nachbilden.

Laufzeitananlyse

Laufzeitanalyse verschiedener Operationen bei Größe n der Datenstruktur:

Operation	Array(List)	Verkettete Liste	Hash-Set	Tree-Set
add (end)	$O(1)$	$O(1)$	$O(1)$	$O(\mathrm{ld}\ n)$
remove	$O(n)$	$O(n)$	$O(1)$	$O(\mathrm{ld}\ n)$
remove (iterator)	$O(n)$	$O(1)$	$O(1)$	$O(\mathrm{ld}\ n)$
get (index)	$O(1)$	$O(n)$	$O(1)$	$O(\mathrm{ld}\ n)$
access (position)	$O(1)$	$O(n)$		
find (element)	$O(n)$	$O(n)$	$O(1)$	$O(\mathrm{ld}\ n)$
insert (position)	$O(n)$	$O(n)$		
insert (iterator)	$O(n)$	$O(1)$		

Laufzeitananlyse

- In der Tabelle scheint das Hash-Set dem Tree-Set eindeutig überlegen zu sein.
- Das Tree-Set bietet jedoch den Vorteil, dass die enthaltenen Elemente effizient sortiert ausgegeben werden können, und es benötigt wesentlich weniger zusätzlichen Speicher als das Hash-Set.
- Außerdem Bei Hashtabelle Reorganisation notwendig, wenn Auslastung zu groß wird. Aufwand $O(n)$, amortisiert $O(1)$.