

# Algorithmen und Datenstrukturen

Reiner Hüchting

26. April 2023

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

Baumstrukturen

Hashverfahren

# Themenüberblick

## Listen-Datentypen

- Abstrakte Listen-Datentypen

- Konkrete Listen-Datentypen

- Dynamische Arrays

- Verkettete Listen

## Suchverfahren

## Sortierverfahren

## Komplexität

## Baumstrukturen

## Hashverfahren

# Themenüberblick

## Listen-Datentypen

- Abstrakte Listen-Datentypen

- Konkrete Listen-Datentypen

- Dynamische Arrays

- Verkettete Listen

## Suchverfahren

## Sortierverfahren

## Komplexität

## Baumstrukturen

## Hashverfahren

# Listen-Datentypen – Abstrakte Listen-Datentypen

## Gemeinsame Eigenschaften von Listen

- ▶ Elemente gleichen Typs
- ▶ Zugriff auf einzelne Elemente möglich
- ▶ Durchlaufen möglich
- ▶ Operationen: Hinzufügen und Löschen von Elementen

## Abstrakter Datentyp „Liste“

- ▶ Wird durch abstrakte Eigenschaften wie oben definiert.
- ▶ Wird *nicht* durch konkrete Implementierungsdetails definiert.
  - ▶ *nicht* durch die konkrete Anordnung im Speicher
  - ▶ *nicht* durch Performance-Eigenschaften

# Listen-Datentypen – Abstrakte Listen-Datentypen

## Abstrakter Datentyp „Liste“

- ▶ Elemente gleichen Typs
- ▶ Zugriff/Durchlaufen möglich
- ▶ Hinzufügen und Löschen von Elementen

## Konkrete Listen-Datentypen

- ▶ Arrays
- ▶ dynamische Arrays
- ▶ verkettete Listen

# Themenüberblick

## Listen-Datentypen

Abstrakte Listen-Datentypen

**Konkrete Listen-Datentypen**

Dynamische Arrays

Verkettete Listen

Suchverfahren

Sortierverfahren

Komplexität

Baumstrukturen

Hashverfahren

# Listen-Datentypen – Konkrete Listen-Datentypen

## Arrays

- ▶ zusammenhängender Bereich im Speicher
- ▶ Zugriff und Durchlaufen mittels *Pointerarithmetik*

## Vorteile

- ▶ Zugriff/Durchlauf sehr schnell
- ▶ *wahlfreier* Zugriff

## Nachteile

- ▶ keine Größenänderung möglich
- ▶ Einfügen ggf. aufwendig oder unmöglich
- ▶ zusammenhängender Platz nötig



# Listen-Datentypen – Konkrete Listen-Datentypen

## Dynamische Arrays

- ▶ verwendet intern Arrays
- ▶ *wahlfreier* Zugriff
- ▶ fügt die Möglichkeit zur Größenänderung hinzu

## Vorteile

- ▶ Zugriff/Durchlauf sehr schnell
- ▶ Größenänderung möglich

## Nachteile

- ▶ Einfügen ggf. aufwendig
- ▶ zusammenhängender Platz nötig

# Listen-Datentypen – Konkrete Listen-Datentypen

## Verkettete Listen

- ▶ Elemente bestehen aus zwei Teilen:
  - ▶ Daten
  - ▶ Pointer/Referenz auf benachbarte Elemente

## Vorteile

- ▶ Größenänderung möglich
- ▶ Einfügen bei bekannter Position schnell
- ▶ kein zusammenhängender Speicherbereich
  - ▶ dadurch bessere Speicher-Ausnutzung

## Nachteile

- ▶ Durchlauf und Zugriff aufwendig
- ▶ kein *wahlfreier* Zugriff

# Themenüberblick

## Listen-Datentypen

Abstrakte Listen-Datentypen

Konkrete Listen-Datentypen

**Dynamische Arrays**

Verkettete Listen

Suchverfahren

Sortierverfahren

Komplexität

Baumstrukturen

Hashverfahren

# Listen-Datentypen – Dynamische Arrays

## Attribute eines dynamischen Arrays

- ▶ Array für die Daten
- ▶ tatsächliche und maximale Länge

## Zentrale Operation: `realloc`

- ▶ neues Array für die Daten mit neuer Länge erzeugen
- ▶ alle Elemente an die neue Stelle kopieren
- ▶ maximale Länge aktualisieren

## Anhängen von Elementen

- ▶ Element an erste freie Stelle schreiben und Größe aktualisieren
- ▶ ggf. vorher `realloc` durchführen

# Listen-Datentypen – Dynamische Arrays

## Anhängen von Elementen

- ▶ falls Array voll: `realloc` durchführen
- ▶ neues Element an erste freie Stelle schreiben
- ▶ Größe aktualisieren

## Wie viel Speicher sollte bei `realloc` reserviert werden?

- ▶ Antwort: Z.B. immer verdoppeln.
- ▶ Ziel: Der Speicher muss exponentiell wachsen, damit `realloc` nicht zu oft notwendig ist.

# Themenüberblick

## Listen-Datentypen

Abstrakte Listen-Datentypen

Konkrete Listen-Datentypen

Dynamische Arrays

Verkettete Listen

Suchverfahren

Sortierverfahren

Komplexität

Baumstrukturen

Hashverfahren

# Listen-Datentypen – Verkettete Listen

## Attribute eines Listenelements

- ▶ Datensatz
- ▶ Zeiger/Referenzen auf die Nachbarelemente

## Zwei typische Varianten

- ▶ einfach verkettete Liste
- ▶ doppelt verkettete Liste

## Anhängen von Elementen

- ▶ Ende der Liste suchen
- ▶ neues Element anhängen

# Listen-Datentypen – Verkettete Listen

## Anhängen von Elementen

- ▶ Ende der Liste suchen
- ▶ neues Element anhängen

## Markierung des Listen-Endes: Sentinel-Prinzip

- ▶ Verwendung eines *Dummy-Elements*
- ▶ wird nicht für Daten verwendet
- ▶ markiert das Ende der Liste

## Vorteil des Dummy-Elements

- ▶ keine Sonderbehandlung der leeren Liste notwendig



# Listen-Datentypen – Verkettete Listen

## Implementierung der ganzen Liste

- ▶ Ein Listenelement ist gleichzeitig auch eine Liste.
- ▶ **Listen haben eine rekursive Struktur!**
- ▶ Container-Klasse für Liste ist dennoch oft nützlich

## Attribute einer verketteten Liste

- ▶ Zeiger/Referenz auf Anfang der Liste oder Dummy

## Vorteil der Container-Klasse

- ▶ kann Dummy vor Benutzer verstecken
- ▶ manche Operationen einfacher umsetzbar

# Themenüberblick

Listen-Datentypen

Suchverfahren

Lineare Suche

Binäre Suche

Sortiervverfahren

Komplexität

Baumstrukturen

Hashverfahren

# Themenüberblick

Listen-Datentypen

Suchverfahren

Lineare Suche

Binäre Suche

Sortiervverfahren

Komplexität

Baumstrukturen

Hashverfahren

# Suchverfahren – Lineare Suche

Ziel: Finde die Position eines Elements in einer Liste

- ▶ Naiver Ansatz: Durchsuche die Liste Element für Element von Anfang bis Ende.

Vorteil

- ▶ Funktioniert für jede Liste.

Komplexität

- ▶ Linear in der Länge der Liste (Schreibe:  $O(n)$ ).
- ▶ Bei Länge  $n$  müssen im Worst Case alle  $n$  Elemente mit dem gesuchten verglichen werden.

# Themenüberblick

Listen-Datentypen

Suchverfahren

Lineare Suche

Binäre Suche

Sortiervverfahren

Komplexität

Baumstrukturen

Hashverfahren

# Suchverfahren – Binäre Suche

Ziel: Finde die Position eines Elements in einer Liste

- ▶ Ansatz: Vergleiche das mittlere Element mit dem gesuchten.
- ▶ Fahre entweder nur links oder nur rechts der Mitte fort.

Vor- und Nachteile

- ▶ Funktioniert nur für sortierte Listen.
- ▶ Ist erheblich schneller als die lineare Suche.

Komplexität

- ▶ Logarithmisch in der Länge der Liste (Schreibe:  $O(\log n)$ ).
- ▶ In jedem Schritt wird der Suchraum halbiert („**Divide and Conquer**“).
- ▶ Bei Länge  $n$  müssen im Worst Case nur  $\log_2 n$  Elemente mit dem gesuchten verglichen werden.

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortiervverfahren

- Insertion Sort

- Selection Sort

- Bubble Sort

- Quick Sort

- Merge Sort

Komplexität

Baumstrukturen

Hashverfahren

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortiervverfahren

Insertion Sort

Selection Sort

Bubble Sort

Quick Sort

Merge Sort

Komplexität

Baumstrukturen

Hashverfahren



# Sortiervverfahren – Insertion Sort

## Sortieren durch Einfügen

- ▶ Ansatz: Nimm das nächstbeste Element und füge es an der passenden Stelle ein.

## Vorteil

- ▶ schnell für kurze Listen
- ▶ einfach zu verstehen und zu implementieren.
- ▶ typischer Aufräum-Ansatz

## Komplexität

- ▶ Quadratisch in der Länge der Liste (Schreibe:  $O(n^2)$ ).
- ▶ Bei Länge  $n$  müssen  $n$  Elemente einsortiert werden.
- ▶ Jedes Einsortieren dauert bis zu  $n$  Schritte.

# Sortiervverfahren – Insertion Sort

## Sortieren durch Einfügen

- ▶ Ansatz: Nimm das nächstbeste Element und füge es an der passenden Stelle ein.

## typische Implementierung für das Einsortieren eines Elements

- ▶ Füge das nächste Element am Ende der Liste an.
- ▶ Tausche es solange nach links, bis es größer als sein linker Nachbar ist.

## Beobachtung

- ▶ kann sehr effizient **in place** umgesetzt werden.
- ▶ d.h. ohne eine separate Hilfsliste

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortiervverfahren

Insertion Sort

**Selection Sort**

Bubble Sort

Quick Sort

Merge Sort

Komplexität

Baumstrukturen

Hashverfahren

# Sortiervverfahren – Selection Sort

## Sortieren durch Auswählen

- ▶ Ansatz: Suche das kleinste Element aus dem noch unsortierten Teil und hänge es ans Ende der sortierten Liste.

## Vorteil

- ▶ schnell für kurze Listen
- ▶ einfach zu verstehen und zu implementieren.
- ▶ typischer Aufräum-Ansatz

## Komplexität

- ▶ Quadratisch in der Länge der Liste (Schreibe:  $O(n^2)$ ).
- ▶ Bei Länge  $n$  müssen  $n$  Elemente gesucht werden.
- ▶ Jede Suche dauert bis zu  $n$  Schritte.

# Sortiervverfahren – Selection Sort

## Sortieren durch Auswählen

- ▶ Ansatz: Suche das kleinste Element aus dem noch unsortierten Teil und hänge es ans Ende der sortierten Liste.

## typische Implementierung für das Einsortieren eines Elements

- ▶ Suche das kleinste Element im unsortierten Teil der Liste.
- ▶ Vertausche das Element mit dem ersten noch nicht einsortierten.

## Beobachtung

- ▶ kann sehr effizient **in place** umgesetzt werden.
- ▶ d.h. ohne eine separate Hilfsliste

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Insertion Sort

Selection Sort

**Bubble Sort**

Quick Sort

Merge Sort

Komplexität

Baumstrukturen

Hashverfahren

# Sortiervverfahren – Bubble Sort

## Sortieren durch Aufsteigen

- ▶ Ansatz: Tausche nach und nach Elemente nach rechts, wenn sie größer als ihre Nachbarn sind.

## Vorteile

- ▶ schnell für kurze Listen
- ▶ sehr intuitiv
- ▶ Lokales Verhalten: Vergleiche nur benachbarte Elemente.

## Komplexität

- ▶ Quadratisch in der Länge der Liste (Schreibe:  $O(n^2)$ ).
- ▶ Bei Länge  $n$  müssen  $n$  Elemente aufsteigen.
- ▶ Jeder Durchlauf dauert bis zu  $n$  Schritte.

# Sortiervverfahren – Bubble Sort

## Sortieren durch Aufsteigen

- ▶ Ansatz: Tausche nach und nach Elemente nach rechts, wenn sie größer als ihre Nachbarn sind.

## Beobachtung

- ▶ kann sehr effizient **in place** umgesetzt werden.
- ▶ d.h. ohne eine separate Hilfsliste

## Analyse

- ▶ Große Elemente am Anfang steigen schnell auf.
- ▶ Kleine Elemente am Ende sinken nur langsam ab.
- ▶  $\Rightarrow$  langsam bei (fast) umgekehrt sortierten Listen



# Sortiervverfahren – Bubble Sort

## Beobachtung bei BubbleSort

- ▶ Große Elemente steigen schnell auf, kleine sinken langsam ab.

## Weiterentwicklung: CombSort/GapSort

- ▶ Ansatz: Vergleiche und vertausche am Anfang Elemente mit größerem Abstand
- ▶ Komplexität im Best Case:  $O(n \log n)$ .
- ▶ Komplexität im Worst Case:  $O(n^2)$ .

## Weiterentwicklung: CocktailSort

- ▶ Ansatz: Wie bei BubbleSort, aber wechsele die Richtungen ab.
- ▶ Vorteil: Alle Elemente bewegen sich ungefähr gleich schnell.
- ▶ Komplexität im Best und Worst Case:  $O(n^2)$ .

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortiervverfahren

Insertion Sort

Selection Sort

Bubble Sort

**Quick Sort**

Merge Sort

Komplexität

Baumstrukturen

Hashverfahren

# Sortiervverfahren – Quick Sort

## Schnelles Divide and Conquer-Verfahren

- ▶ Ansatz: Sortiere Elemente bzgl eines Referenzelements vor. Sortiere anschließend rekursiv die Teillisten.

## Vorteile

- ▶ schnell für lange Listen
- ▶ Gilt (mit Modifikationen) als das schnellste verfügbare Sortiervverfahren.

## Komplexität

- ▶ Worst Case:  $O(n^2)$ .
- ▶ Average- und Best-Case:  $O(n \log n)$ .
- ▶ Vorsortieren braucht  $n$  Vergleiche.
- ▶ Im Idealfall wird mit jedem Schritt die Liste halbiert.

# Sortiervverfahren – Quick Sort

## Schnelles Divide and Conquer-Verfahren

- ▶ Ansatz: Sortiere Elemente bzgl eines Referenzelements vor. Sortiere anschließend rekursiv die Teillisten.

## Modifikationen

- ▶ Für kurze Listen auf *InsertionSort* ausweichen.
- ▶ Rekursionstiefe begrenzen: Auf *MergeSort* wechseln, um  $O(n \log n)$  im Worst-Case zu garantieren.

## Beobachtung

- ▶ kann sehr effizient **in place** umgesetzt werden.
- ▶ Der Worst-Case ist gerade die umgekehrt sortierte Liste.
- ▶ gut parallelisierbar

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Insertion Sort

Selection Sort

Bubble Sort

Quick Sort

**Merge Sort**

Komplexität

Baumstrukturen

Hashverfahren

# Sortiervverfahren – Merge Sort

## Schnelles Divide and Conquer-Verfahren

- ▶ Ansatz: Halbiere die Liste rekursiv, bis nur noch einzelne Elemente übrig sind. Setze dann sortierte Listen zu längeren sortierten Listen zusammen.

## Vorteile

- ▶ schnell für lange Listen

## Komplexität

- ▶ Worst Case:  $O(n \log n)$ .
- ▶ Mit jedem Schritt wird die Liste halbiert.
- ▶ Zusammensetzen dauert  $n$  Schritte.

# Sortierverfahren – Merge Sort

## Schnelles Divide and Conquer-Verfahren

- ▶ Ansatz: Halbiere die Liste rekursiv, bis nur noch einzelne Elemente übrig sind. Setze dann sortierte Listen zu längeren sortierten Listen zusammen.

## Modifikationen

- ▶ *TimSort*: Identifiziere bereits sortierte Teillisten und spare diese bei der Rekursion aus.
- ▶ Standard-Sortierverfahren in Python

## Beobachtung

- ▶ i.d.R. nicht in place umgesetzt (braucht Hilfsarrays der Länge  $n/2$ )
- ▶ gut für verkettete Listen
- ▶ gut parallelisierbar

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

$O$ -Notation

Optimierung von Algorithmen

Baumstrukturen

Hashverfahren



# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

$O$ -Notation

Optimierung von Algorithmen

Baumstrukturen

Hashverfahren

# Komplexität – O-Notation

## Bisher: Informelle Komplexitätsabschätzungen

- ▶ Laufzeitabschätzungen in Abhängigkeit der Größe einer Datenstruktur
  - ▶ z.B. Länge einer Liste oder Anzahl der Elemente eines Baumes
- ▶ Beobachtung: Laufzeit wird i.d.R. *ungenau* angegeben.
  - ▶ z.B. Schleifendurchläufe zählen, aber nicht die Anzahl der Operationen innerhalb der Schleife
  - ▶ z.B. geschachtelte Schleifen berücksichtigen, hintereinander ausgeführte Schleifen aber nicht

## Ziel: Formalisierung dieser Ungenauigkeiten

- ▶ Wie kommen diese Abschätzungen zustande?
- ▶ Welche Operationen müssen gezählt werden?

# Komplexität – $O$ -Notation

## Beispiel: Maximum einer Liste bestimmen

```
func SearchMax(list []int) int {  
    max := list[0]  
    for _, v := range list {  
        if v > max {  
            max = v  
        }  
    }  
    return max  
}
```

## Komplexität

- ▶  $n$  Schleifendurchläufe (Vergleiche  $v > \text{max}$ )
- ▶ Komplexitätsklasse:  $O(n)$

# Komplexität – O-Notation

Beispiel: Differenz zw. Minimum und Maximum bestimmen

```
func DiffMinMax(list []int) int {  
    return SearchMax(list) - SearchMin(list)  
}
```

## Komplexität

- ▶ Je ein Durchlauf von searchMax und searchMin
  - ▶ Diese haben jeweils lineare Komplexität ( $O(n)$ ).
- ▶ Komplexitätsklasse:  $O(n)$ 
  - ▶ Warum nicht  $O(2n)$ ?

# Komplexität – O-Notation

## Beispiel: Minimale Differenz von Elementen bestimmen

```
func ClosestPair(list []int) int {  
    min := DiffMinMax(list)  
    for i, v1 := range list {  
        for _, v2 := range list[i+1:] {  
            diff := int(math.Abs(float64(v1 - v2)))  
            if diff < min {  
                min = diff  
            }  
        }  
    }  
    return min  
}
```

## Komplexität

- ▶  $n$  Durchläufe der äußeren Schleife,  $\leq n$  mal innere Schleife
- ▶ Komplexitätsklasse:  $O(n^2)$

# Komplexität – $O$ -Notation

## Beobachtungen

- ▶ Komplexitätsklassen geben nur die Größenordnung an.
- ▶ Konstante Faktoren und nicht-dominante Terme werden vernachlässigt.

## Beispiele

$$O(n) = O(2n) = O\left(\frac{n}{2}\right)$$

$$O(n^2) = O(n^2 + n + 1) = O\left(\left(\frac{n}{2}\right)^2\right)$$

$$O(n \log n) = O(2n \log n + 50n)$$

# Komplexität – O-Notation

## Intuition:

- ▶ Der Unterschied zwischen  $O(n)$  und  $O(2n)$  kann durch schnellere Hardware ausgeglichen werden.
- ▶ Ebenso der Unterschied zwischen  $O(n^2)$  und  $O(2(n^2))$ .
- ▶ Der Unterschied zwischen  $O(n)$  und  $O(n^2)$  kann nicht so einfach kompensiert werden.
- ▶ Das Verhalten von Polynomen (Funktionen) wird i.W. vom *Leitterm* bestimmt.

## Ziel bei der Entwicklung:

- ▶ Komplexitätsklasse möglichst klein halten.
- ▶ **Komplexität kann nicht durch Hardware ausgeglichen werden!**
- ▶ Konstante oder lineare Faktoren sind weniger von Bedeutung.

## Definition: $O$ -Komplexität

Gegeben eine Funktion  $f(n)$ , ist  $f(n) = O(g(n))$  genau dann, wenn es eine positive Konstante  $c$  gibt, so dass für alle  $n \geq n_0$  gilt:

$$f(n) \leq c \cdot g(n)$$

## Intuitiv:

- ▶ Falls  $f(n) \geq g(n)$  für alle  $n$  gilt, dann unterscheiden sich die Funktionen nur durch einen konstanten Faktor.
- ▶ Für große  $n$  ist  $g(n)$  eine gute Abschätzung für  $f(n)$ .



# Komplexität – O-Notation

Definition: **O-Komplexität**

$$f \in O(g) \Leftrightarrow \exists_{c>0} \exists_{n_0} \forall_{n \geq n_0} : f(n) \leq c \cdot g(n)$$

Intuitiv:

- ▶ Die Funktion  $f(n)$  wächst nicht schneller als  $g(n)$ .
- ▶ Für **fast alle  $n$**  gilt  $f(n) \leq c \cdot g(n)$ .

Konstante Faktoren sind nicht relevant:

- ▶ Bewegen sich im Bereich der Ungenauigkeit, die durch unterschiedliche Hardware entsteht.
- ▶ Bieten geringes Optimierungspotenzial.
- ▶ Können ggf. durch Hardware ausgeglichen werden.

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

$O$ -Notation

Optimierung von Algorithmen

Baumstrukturen

Hashverfahren

# Komplexität – Optimierung von Algorithmen

Ziel: Prüfung, ob zwei Listen die gleichen Elemente haben

- ▶ Gegeben: Zwei Listen von Zahlen A und B
- ▶ Ergebnis: `true`, falls jedes Element aus Liste A auch in Liste B vorkommt und umgekehrt.

## Naiver Ansatz

- ▶ Durchlaufe Liste A.
  - ▶ Suche jedes Element in Liste B.
- ▶ Wiederhole für Liste B.

## Komplexität

- ▶  $n$  Durchläufe der äußeren Schleife
- ▶ pro Durchlauf:  $\leq n$  Durchläufe der inneren Schleife
- ▶ Komplexitätsklasse:  $O(n^2)$

# Komplexität – Optimierung von Algorithmen

Ziel: Prüfung, ob zwei Listen die gleichen Elemente haben

- ▶ Gegeben: Zwei Listen von Zahlen A und B
- ▶ Ergebnis: true, falls jedes Element aus Liste A auch in Liste B vorkommt und umgekehrt.

## Optimierte Lösung

- ▶ Sortiere beide Listen.
- ▶ Vergleiche die Listen in einem einzigen Durchlauf.

## Komplexität

- ▶ Sortieren:  $O(n \log n)$
- ▶ Vergleichen:  $O(n)$
- ▶ **Gesamt:**  $O(n \log n) + O(n) = O(n \log n)$

# Komplexität – Optimierung von Algorithmen

Ziel: Suche nach dem größten Produkt benachbarter Elemente einer Liste

- ▶ Gegeben: Eine Liste von Zahlen der Länge  $n$ .
- ▶ Ergebnis: Das größte Produkt von  $m$  benachbarten Elementen.

## Naiver Ansatz

- ▶ Durchlaufe die Liste von Stelle 0 bis  $n - m$ .
  - ▶ Von jeder Position aus berechne das Produkt der nächsten  $m$  Elemente.

## Komplexität

- ▶  $n - m + 1 \leq n$  Durchläufe der äußeren Schleife
- ▶ pro Durchlauf:  $m$  Durchläufe der inneren Schleife
- ▶ Komplexitätsklasse:  $O(n \cdot m)$  (für große  $m$  grob  $O(n^2)$ )

# Komplexität – Optimierung von Algorithmen

Ziel: Suche nach dem größten Produkt benachbarter Elemente einer Liste

- ▶ Gegeben: Eine Liste von Zahlen der Länge  $n$ .
- ▶ Ergebnis: Das größte Produkt von  $m$  benachbarten Elementen.

## Optimierter Ansatz

- ▶ Berechne das Produkt der ersten  $m$  Elemente.
- ▶ Durchlaufe die Liste von Stelle  $m$  bis  $n$ .
  - ▶ Multipliziere das Produkt mit dem Element an Stelle  $i$ .
  - ▶ Dividiere das Produkt durch das Element an Stelle  $i - m$ .

## Komplexität

- ▶ Berechnung des Anfangsprodukts:  $O(m)$
- ▶ Schleife  $O(n - m)$
- ▶ Gesamt:  $O(n)$

# Komplexität – Optimierung von Algorithmen

## Ziel: Suche nach einem String in einem Text

- ▶ Gegeben: Ein Text (String) der Länge  $n$  und ein zu suchender String der Länge  $m$ .
- ▶ Ergebnis: Alle Positionen, an denen der Suchstring vorkommt.

## Naiver Ansatz

- ▶ Durchlaufe den gesamten Text.
  - ▶ An jeder Position vergleiche den dortigen Teilstring mit dem gesuchten String.

## Komplexität

- ▶  $n$  Durchläufe der äußeren Schleife
- ▶ pro Durchlauf:  $m$  Schritte für den Vergleich.
- ▶ Komplexitätsklasse:  $O(n \cdot m)$ .

# Komplexität – Optimierung von Algorithmen

## Ziel: Suche nach einem String in einem Text

- ▶ Gegeben: Ein Text (String) der Länge  $n$  und ein zu suchender String der Länge  $m$ .
- ▶ Ergebnis: Alle Positionen, an denen der Suchstring vorkommt.

## Optimierung

- ▶ Durchlaufe den gesamten Text.
  - ▶ An jeder Position vergleiche den dortigen Teilstring mit dem gesuchten String.
  - ▶ Bei Nicht-Übereinstimmung berechne, wie weit gesprungen werden kann.



# Komplexität – Optimierung von Algorithmen

## Ziel: Suche nach einem String in einem Text

- ▶ Gegeben: Ein Text (String) der Länge  $n$  und ein zu suchender String der Länge  $m$ .
- ▶ Ergebnis: Alle Positionen, an denen der Suchstring vorkommt.

## Optimierung bei häufiger Suche

- ▶ Baue einen **Suchindex** auf:
- ▶ Z.B. ein Präfixbaum, der für mögliche Suchbegriffe die Positionen im Text enthält.
- ▶ Kann aus einer Datenbank häufiger Anfragen erstellt werden.
- ▶ Kann nebenbei erstellt und aktualisiert werden.

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

Baumstrukturen

- Binärbäume

- Binäre Suchbäume

- AVL-Bäume

- Heaps

- Präfixbäume

Hashverfahren

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

**Baumstrukturen**

**Binärbäume**

Binäre Suchbäume

AVL-Bäume

Heaps

Präfixbäume

Hashverfahren

# Baumstrukturen – Binärbäume

## Definition: Binärbaum

Für jeden Binärbaum gilt eine der folgenden Möglichkeiten:

- ▶ Der Baum ist leer.
- ▶ Der Baum besteht aus einer Wurzel und zwei Teil**bäumen**.

## Bemerkungen

- ▶ Rekursive Definition beschreibt direkt die Struktur.
- ▶ Verallgemeinerung zu *Bäumen* möglich.
- ▶ Häufig vorkommende Struktur in Mathematik und Informatik.

# Baumstrukturen – Binärbäume

## Beispiele

- ▶ Turnierbäume bei Wettbewerben
- ▶ Wahrscheinlichkeitsbäume
- ▶ Stammbäume
- ▶ Modellierung von Abhängigkeiten

## Anwendung in der Informatik

- ▶ Strukturierung und Sortierung von Daten

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

**Baumstrukturen**

Binärbäume

**Binäre Suchbäume**

AVL-Bäume

Heaps

Präfixbäume

Hashverfahren

# Baumstrukturen – Binäre Suchbäume

## Definition: Binärer Suchbaum

Ein binärer Suchbaum ist ein Binärbaum mit folgenden Eigenschaften:

- ▶ Die Knoten enthalten *Schlüssel* und *Werte* (engl. *key/value*).
- ▶ Auf den Schlüsseln ist eine *totale Ordnung* definiert.
- ▶ Für jeden Knoten gilt:
  - ▶ Die Schlüssel aller Knoten im linken Teilbaum sind kleiner als der Schlüssel der Wurzel.
  - ▶ Die Schlüssel aller Knoten im rechten Teilbaum sind größer als der Schlüssel der Wurzel.
  - ▶ (Gleichheit muss ggf. einer der Seiten zugeschlagen werden.)

## Ziel: Effiziente Implementierung von Listen und Datenbanken

- ▶ Idee: Binärer Suche und effiziente Sortierverfahren direkt in einer Datentstruktur verankern.

# Baumstrukturen – Binäre Suchbäume

## Auffinden von Elementen mit einem bestimmten Suchschlüssel

- ▶ Baum leer: Nicht gefunden.
- ▶ Suchschlüssel in Wurzel gefunden, liefere (Wert der) Wurzel.
- ▶ Suchschlüssel kleiner als Wurzel: Suche im linken Teilbaum.
- ▶ Suchschlüssel größer als Wurzel: Suche im rechten Teilbaum.

## Einfügen von Elementen mit einem bestimmten Suchschlüssel

- ▶ Baum leer: Hier einfügen.
- ▶ Suchschlüssel kleiner Wurzel: Füge in linken Teilbaum ein.
- ▶ Suchschlüssel größer Wurzel: Füge in rechten Teilbaum ein.



## Löschen von Elementen mit einem bestimmten Suchschlüssel

- ▶ Suche das zu löschende Element.
- ▶ Falls Blatt: Entfernen.
- ▶ Ansonsten: Suche den direkten Nachfolger oder Vorgänger.
- ▶ Vertausche Element mit Nachfolger/Vorgänger.
- ▶ Entferne Element aus entsprechendem Teilbaum.

# Baumstrukturen – Binäre Suchbäume

## Verhalten im Optimalfall

- ▶ Linker und rechter Teilbaum in allen Knoten gleich tief.
- ▶ Logarithmisches Verhalten beim Suchen, Einfügen und Löschen.

## Verhalten im Worst Case

- ▶ Eine Seite hat starkes Übergewicht.
- ▶ Extremfall: Jeder Knoten hat nur einen Teilbaum.
- ▶ Der Baum ist dann de facto eine Liste.

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

**Baumstrukturen**

Binärbäume

Binäre Suchbäume

**AVL-Bäume**

Heaps

Präfixbäume

Hashverfahren

# Baumstrukturen – AVL-Bäume

## Ziel: Optimierung von binären Suchbäumen

- ▶ Problem: Bäume können aus der Balance geraten.
- ▶ Lösung: Reorganisieren, wenn das Ungleichgewicht zu groß wird.

## Vorgehensweise

- ▶ Beim Einfügen oder Löschen Struktur des Baumes analysieren.
- ▶ Bei Bedarf Knoten umsortieren, damit linker und rechter Teilbaum jedes Knotens ungefähr gleich groß sind.
- ▶ Problem: Analyse darf nicht zu teuer sein.
- ▶ Frage: Was bedeutet „ungefähr gleich groß“ überhaupt?

# Baumstrukturen – AVL-Bäume

## Definition: Tiefe

Die Tiefe eines Knotens ist die Länge des Pfades von der Wurzel bis zu diesem Knoten.

## Bemerkungen

- ▶ Berechnung rekursiv:
  - ▶ Tiefe der Wurzel: 0.
  - ▶ Tiefe eines Knotens:  $1 + \text{Tiefe des Elternknotens}$
- ▶ Kann beim Einfügen/Löschen nebenbei berechnet werden.
- ▶ Kann bei Bedarf im Knoten gespeichert und gepflegt werden.

# Baumstrukturen – AVL-Bäume

## Definition: Höhe

Die Höhe eines Baums ist die Tiefe des tiefsten Knotens im Baum.

## Bemerkungen

- ▶ Berechnung rekursiv:
  - ▶ Höhe eines Blatts Wurzel: 1.
  - ▶ Höhe eines Knotens:  $1 + \text{Höhe des höheren Kindes}$
- ▶ Kann beim Einfügen/Löschen nebenbei berechnet werden.
- ▶ Kann bei Bedarf im Knoten gespeichert und gepflegt werden.

# Baumstrukturen – AVL-Bäume

## Definition: Balancefaktor

Der Balancefaktor eines Knotens ist die Differenz zwischen der Höhe des rechten und des linken Teilbaums.

## Bemerkungen

- ▶ Kann beim Einfügen/Löschen nebenbei berechnet werden.
- ▶ Gutes Maß für die Ausgeglichenheit des Baumes.

# Baumstrukturen – AVL-Bäume

## Definition: AVL-Baum

Ein AVL-Baum ist ein binärer Suchbaum mit folgender Eigenschaft:

- ▶ Der Balancefaktor jedes Knotens liegt im Intervall  $[-1, 1]$ .

## Umsetzung

- ▶ Beim Einfügen/Löschen Balancefaktoren bestimmen.
- ▶ Nach Einfügen in Teilbaum: Umorganisieren der Knoten.
- ▶ Nach Einfügen in Teilbaum bedeutet: Nach Rekursion.



# Baumstrukturen – AVL-Bäume

## Analyse der Balancefaktoren

Nach Einfügen eines Knotens Balancefaktor prüfen.

- ▶ Falls  $-2$ : Linker Teilbaum zu hoch.
- ▶ Falls  $2$ : Rechter Teilbaum zu hoch.
- ▶ Prüfe Balancefaktor des linken/rechten Teilbaumes.
- ▶ Führe passende *Rotation* durch.

## Ungleichgewichts-Situationen und Rotationen

- ▶ Links-Links
- ▶ Links-Rechts
- ▶ Rechts-Rechts
- ▶ Rechts-Links

# Baumstrukturen – AVL-Bäume

## Verhalten beim Einfügen/Löschen/Suchen

- ▶ Linker und rechter Teilbaum in allen Knoten fast gleich tief.
- ▶ Logarithmisches Verhalten beim Suchen, Einfügen und Löschen.

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

## Baumstrukturen

Binärbäume

Binäre Suchbäume

AVL-Bäume

**Heaps**

Präfixbäume

Hashverfahren

# Baumstrukturen – Heaps

## Erinnerung: Binäre Suchbäume

- ▶ Idee: Optimales Such- und Einfügeverhalten sortierter Listen.
- ▶ Komplexität: Logarithmisches Verhalten wie bei binärer Suche.

Es geht besser, wenn keine perfekte Sortierung benötigt wird!

## Idee: Fordere nur eine partielle Sortierung

- ▶ Knoten müssen größer oder kleiner als ihre Kinder sein.
- ▶ Keine Relation zwischen den Kindern.
- ▶ Beobachtung: Größtes/Kleinstes Element steht an der Wurzel.
- ▶ Ermöglicht sehr schnellen Zugriff auf Wurzel.

# Baumstrukturen – Heaps

## Definition: Vollständiger Binärbaum

Ein vollständiger Binärbaum ist ein Binärbaum mit folgenden Eigenschaften:

- ▶ Jede Ebene ist vollständig besetzt.
- ▶ Nur in der untersten Ebene dürfen Elemente fehlen.
- ▶ Ebenen werden beim Einfügen von links nach rechts aufgefüllt.

## Bemerkungen

- ▶ Ein vollständiger Binärbaum hat keine Lücken.
- ▶ Kann deshalb effizient als Liste gespeichert werden.
- ▶ Einfache Berechnung der Indizes:
  - ▶ Elternknoten an Stelle  $n$
  - ▶ Kinder an Stellen  $2n + 1$  und  $2n + 2$

# Baumstrukturen – Heaps

## Definition: Min-Heap

Ein Min-Heap ist ein vollst. Binärbaum mit folgender Eigenschaft:

- ▶ Jeder Knoten ist kleiner als seine Kinder
- ▶ (Definition *Max-Heap* analog.)

## Einfügen von Elementen

- ▶ Neues Element am Ende einfügen.
- ▶ Aufsteigen lassen, bis es richtig eingeordnet ist.

## Entfernen der Wurzel

- ▶ Wurzel durch letztes Element ersetzen (tauschen).
- ▶ Absteigen lassen, bis es richtig eingeordnet ist.

# Baumstrukturen – Heaps

## Verhalten beim Einfügen/Löschen von Elementen

- ▶ Zugriff auf Wurzel in konstanter Zeit ( $O(1)$ ).
- ▶ Einfügen und Löschen in  $O(\log n)$ .

## Anwendungen

- ▶ *Priority Queues*
- ▶ Routingverfahren, z.B. Navigationssysteme, Netzwerke
- ▶ Optimierungs- und Planungsprobleme
- ▶ effiziente Sortiervverfahren (HeapSort)

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

**Baumstrukturen**

Binärbäume

Binäre Suchbäume

AVL-Bäume

Heaps

**Präfixbäume**

Hashverfahren



# Baumstrukturen – Präfixbäume

## Präfixbäume: Effiziente Speicherung von Zeichenketten

- ▶ Speichere Zeichenketten in einer Baumstruktur ab.
- ▶ Annotiere Knoten mit Eigenschaften dieser Zeichenketten.
- ▶ Zeichenketten mit gemeinsamem Präfix haben gemeinsame Pfade im Baum.
- ▶ Vorteile: Kompression und schnelle Suche.

## Anwendungen

- ▶ Metadaten von Textdokumenten
- ▶ Aufbau eines Suchindex für Texte
- ▶ Aufbau von Wörterbüchern (z.B. für *Predictive Text*)
- ▶ Kompressionsverfahren

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

Baumstrukturen

Hashverfahren

Hashmaps

Hashfunktionen

# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

Baumstrukturen

Hashverfahren

Hashmaps

Hashfunktionen

# Hashverfahren – Hashmaps

## Erinnerung: Binäre Suchbäume

- ▶ effiziente Speicherung von Schlüssel-Wert-Paaren
- ▶ schnelles Einfügen, Löschen und Suchen  
(z.B.  $O(\log n)$  bei AVL-Bäumen)
- ▶ Pointerstrukturen mit bekannten Vor- und Nachteilen

## Erinnerung: Heaps

- ▶ vollständige Binärbäume mit Teil-Sortierung der Elemente
- ▶ Einfügen und Löschen in  $O(\log n)$
- ▶ Zugriff auf Wurzel sogar in  $O(1)$
- ▶ Speicherung als Array, effizienter als binäre Suchbäume

# Hashverfahren – Hashmaps

Neues Ziel: Average-Case-Zugriff auf jeden Schlüssel in  $O(1)$

Wie könnte das gehen?

- ▶ Warum geht der Zugriff auf die Wurzel bei Heaps schnell?
  - ▶ Position ist bekannt und muss nicht gesucht werden!
- ▶ Können wir das für alle Elemente erreichen?
  - ▶ Idee: Verwende ein Array und berechne die Position des Elements aus dem Element selbst.

Welchen Preis müssen wir dafür bezahlen?

- ▶ Erheblich mehr Speicherverbrauch
- ▶ schlechtere Worst-Case-Komplexität

# Hashverfahren – Hashmaps

## Idee zu Hashmaps

- ▶ Speichere Elemente in einem Array (**Hashtabelle**).
- ▶ Berechne Position des Schlüssels mittels einer **Hashfunktion**.

## Eigenschaften der Hashfunktion

- ▶ Eingabe: Das Element bzw. dessen Schlüssel
- ▶ Ergebnis: Die Position des Elements im Array oder eine Zahl, aus der diese Position berechnet werden kann.

# Hashverfahren – Hashmaps

## Idee zu Hashmaps

- ▶ Speichere Elemente in einer Hashtabelle.
- ▶ Berechne Position des Schlüssels mittels einer **Hashfunktion**.

## Beispiele für einfache Hashfunktionen

Eingabe	Ergebnis
ein String	Summe der ASCII-Werte
ein String	nach Position gewichtete Summe der ASCII-Werte
ein Integer	der Wert selbst
ein Integer	(gewichtete) Quersumme
ein Struct	Summe der Hashwerte aller Member

# Hashverfahren – Hashmaps

## Idee zu Hashmaps

- ▶ Speichere Elemente in einer Hashtabelle.
- ▶ Berechne Position des Schlüssels mittels einer **Hashfunktion**.

## Beobachtungen

- ▶ Hashfunktion berechnet Zahlen aus beliebigen Elementen.
- ▶ Ziel-Position kann z.B. durch *Modulo* errechnet werden.
- ▶ Die Hashfunktion ist i.d.R. **nicht injektiv**.
- ▶ D.h. es sind **Kollisionen** möglich.
- ▶ Anforderung: **Berechnung muss schnell gehen!**



# Themenüberblick

Listen-Datentypen

Suchverfahren

Sortierverfahren

Komplexität

Baumstrukturen

Hashverfahren

Hashmaps

Hashfunktionen

# Hashverfahren – Hashfunktionen

## Anforderungen an Hashfunktionen

- ▶ Hashfunktion berechnet Zahlen aus beliebigen Elementen.
- ▶ Kollisionen sind nicht zu vermeiden, sollten aber so selten wie möglich auftreten.
- ▶ Berechnung des Hashes muss schnell gehen.

## Vermeidung von Kollisionen

- ▶ Werte sollten möglichst gleichmäßig verteilt sein.
- ▶ Viel mehr Speicher verwenden als benötigt wird, damit die Wahrscheinlichkeit einer Kollision gering ist.

## Umgang mit Kollisionen

- ▶ **geschlossenes Hashing**: Neue Position berechnen (*Sondieren*)
- ▶ **offenes Hashing** Mehrere Elemente pro Position erlauben
  - ▶ Z.B. als Liste pro Position

# Hashverfahren – Hashfunktionen

## geschlossenes Hashing

- ▶ Berechne so lange neue Hash-Werte, bis eine freie Stelle in der Hashtabelle gefunden wurde.
- ▶ Auch beim Suchen nach Schlüsseln müssen wiederholt Hashes berechnet und die gefundenen Elemente geprüft werden.

## Beispiel: Doppel-Hashing

- ▶ Weiche bei Kollisionen auf eine zweite Hashfunktion aus.
- ▶ Multipliziere die Hash-Werte mit der Anzahl der Versuche.

## Beispiel: Kuckucks-Hashing

- ▶ Verwende zwei Hashtabellen mit zwei Hashfunktionen.
- ▶ Verdränge bei Kollisionen ggf. das vorgefundene Element
- ▶ Füge verdrängte Elemente in die andere Hashtabelle ein.

# Hashverfahren – Hashfunktionen

## Zusammenfassung

- ▶ Speichere Elemente in einer Hashmap, bei der Positionen berechnet werden.
- ▶ Verwende Hashfunktion mit möglichst wenigen Kollisionen.
- ▶ Verwende viel Speicher, um Kollisionen zu vermeiden.
- ▶ Bei Kollisionen verwende finde eine neue Position oder speichere Elemente in Listen.

## Eigenschaften von Hashmaps

- ▶ **Effizienz**: Hashfunktion berechnet schnell Positionen.
- ▶ Average Case:  $O(1)$  für Suchen und Einfügen.
- ▶ Worst Case:  $O(n)$  für Suchen und Einfügen.
  - ▶ geschlossenes Hashing: Ggf. viele Berechnungen notwendig.
  - ▶ offenes Hashing: Ggf. lange Listen an wenigen Positionen.