Algorithmus

Ein Algorithmus ist eine **Anleitung** (instruction set) zur Lösung einer Aufgabenstellung, die so präzise formuliert ist (durch eine Programmiersprache oder Prosa), dass sie «mechanisch» ausgeführt werden kann.

Bsp: ggT

```
int ggT(int a, int b) {
   if (a > b) return ggT(a - b, b);
   else if (a < b) return ggT(a, b - a);
   else return a;
}
```

```
if (a == 0) return b;
while (b != 0) {
   c = a % b;
   a = b;
   b = c;
}
return a;
```

Eigenschaften von Algorithmen

1. **Determiniertheit:** Identische Eingaben führen stets zu identischen Ergebnissen.
2. **Determinismus:** Ablauf des Verfahrens ist an jedem Punkt fest vorgeschrieben (keine Wahlfreiheit).
3. **Terminierung:** Für jede Eingabe liegt das Ergebnis nach endlich vielen Schritten vor.
4. **Effizienz:** «Wirtschaftlichkeit» des Aufwands relativ zu einem vorgegebenen Massstab (z.B. Laufzeit, Speicherplatzverbrauch).

Jedes Programm repräsentiert einen bestimmten Algorithmus.
Kein Programm ohne Algorithmus

Abstrakte Datentypen (ADT)

Einfache Datentypen: byte, short, int, long, float, double, char, boolean
Referenz Datentypen (Pointers):
Array, String, Objects, Wrapperklassen (→ Autoboxing)
Abstrakte Datentypen (ADT):
Stacks, Collections, List, Queues, Hashtable, Trees, etc.

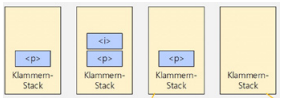
Kapselung / Information Hiding:
Informationen über die Implementierung wird verborgen.
Für jeden ADT ist eine **Schnittstelle (Interface)** definiert.

Stack

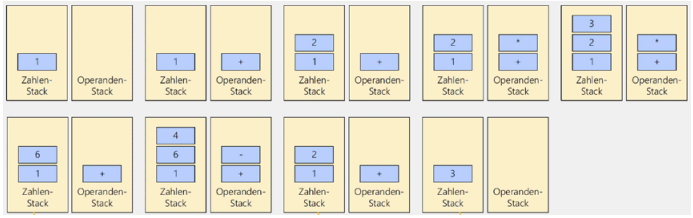
LIFO (Last In, First Out) oder FILO (First In, Last Out)
Operationen: push, pop, peek

Implementation: Mit der **Liste** oder einem **Array**
kann ein **Stack** variabler Grösse implementiert werden.

Anwendungsbeispiele:
- Überprüfen, ob die öffnenden und schliessenden Tags in einem HTML-Ausdruck korrekt gesetzt sind: ein kleiner Test



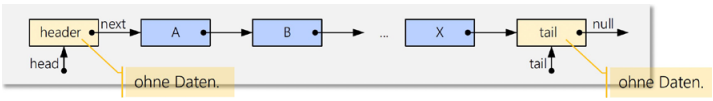
- Parser: Punkt-Vor-Strich-Regel: 1 + 2 * 3 - 4



- Turingmaschine

Lists (LinkedList)

Ist im Prinzip ein Array variabler grösse.
Operationen: add(obj), add(obj, pos), get(pos), remove(pos), remove(obj), contains(obj)



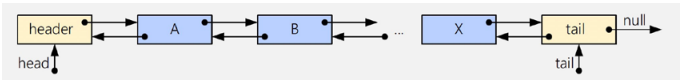
Header- und der Tail-Knoten werden in eigenen Variablen gespeichert.
Vorteile eines Tail-Knotens: Das Ende und Grösse der Liste ist jederzeit bekannt.

Rekursive Methode um die Elemente einer einfach verketteten Liste in umgekehrter Reihenfolge auszugeben:

```
def reverse(n, visitor):
    if not n.has_next():
        visitor.visit(n.value)
    else:
        reverse(n.next, visitor)
```

Doppelte verkette Listen:

Probleme bei **einfach verkette Listen:**
- Traversierung nur in eine Richtung möglich.
- get() nahe Listenende kostet viel Aufwand.



Zeitkomplexität
get, remove: **O(n)** für **unsortiert** oder **O(1)** für **bekanntes Objekt** (Referenz) oder **O(log n)** wenn **sortiert mit Binärsuche.**
add: O(n)
Traversierung: O(n)
Mittels Iterator (it.hasNext(), it.next())
Es wird ein interner Zeiger festgelegt auf das aktuelle Element.

Sortierte Listen:

Elemente müssen das Comparable Interface implementiert haben.

Comparable: Objekte gleichen Typs können sich miteinander vergleichen.
Comparator: Ist eine Art Callback, dass spezifische Vergleiche ermöglicht.
Es können sogar beliebige Objekte verglichen werden (selten).
Collections.sort(list, **(o1, o2) -> ...**);

ArrayList vs LinkedList
ArrayList: Elemente in einem Array (als Memory-Block) gespeichert.
Automatische Speicherverwaltung: System.arraycopy(...)

ArrayList
- Implementation als Array
- Mutationen (arraycopy) langsam: O(n)
- direkter Zugriff schnell: O(1)

LinkedList
- Implementation mittels Referenzen
- schneller für Mutationen: O(n)
- langsamer bei direktem Zugriff: O(n)

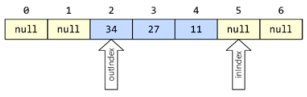
Queue

FIFO (First In, First Out)
Speichert Objekte in einer **(Warte-)Schlange.**

Operationen: enque/add/offer, deque/poll, peek
Andwendungsbeispiele: Drucker, gestaffelt Resourcenzugriffe
Implementation: Mittels **Liste** oder **Array**

PriorityQueue

Operationen: ..., enque(Object x, int **priority**)
Implementation:
- **Array** mit zwei Cursors **(Ring Buffer)**
- **Sortierter Liste**



Anwendungen:
- Scheduling von Prozessen in Betriebssystemen
- Taskliste nach Prioritäten geordnet

Sets

Geordnete oder ungeordnete Menge **ohne Duplikate.**

Implementation:
- **HashSet:** bei grossen Datenbeständen (etwas) effizienter
- **TreeSet:** speichert die Elemente in **geordneter** Reihenfolge

Operationen:
contains
containsAll: Testet für eine Teilmengenbeziehung
addAll: Führt Union aus
retainAll: Führt Intersection aus
removeAll: Führt Difference aus

Anwendungsbeispiele:
- Mengenoperationen
- Keine doppelte Elemente

Generics

Generics ermöglichen die Festlegung der Typen zur Übersetzungszeit.

Collections-Klassen akzeptieren nur Referenztypen / Wrapperklassen.
Typparamter: List**<T>**

Wildcard <?>

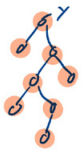
Wenn der genaue Typ nicht bekannt sein muss.

Subtypisierung:

– super (Lower Bound = lässt weniger Klassen zu)
List<? **super X**>
Klasse X oder alle Oberklassen.

Problem: Typinformation geht verloren.
Macht nur Sinn, wenn genauer Typ nicht bekannt sein muss.

– extends (Upper Bound = lässt mehr Klassen zu)
List<? **extends Y**>
Alle Klassen, die Klasse Y implementieren/erweitern.



Type Erasure

Java entfernt die Typeninformation vollständig zur **Laufzeit**.
Aus Box<T> wird zur Laufzeit Box<Object>.

Nachteile
- Keine Typenprüfung möglich: **if (e instanceof List<Integer>)** oder **a.getClass() == b.getClass()**
  Zur Laufzeit kann nicht mehr z.B. zwischen LinkedList<String> und LinkedList<Integer> unterschieden werden. Beide haben zur Laufzeit den Typ **List<Object>**
- Cast sind nicht überprüfbar: **E e = (E)o;**
- Keine Arrays von **T[] a = new T[10];**
- Kein Objekt-Instanzierung möglich: **T t = new T();**
  Lösung:
```
<T> T foo(Class<T> clazz) {
    return (T)clazz.getDeclaredConstructor().newInstance();
}
String s = foo(String.class);
```

Raw Types
Verwendung eines generischen Typs ohne Typparameters Angabe.
List list = new ArrayList<>(); anstelle von List<String> list = new ArrayList<>();
List list ist äquivalent zu List<Object> list

Rohtypen wurden beibehalten, um die Abwärtskompatibilität aufrechtzuerhalten (**Interoperabilität**).

O-Notation / Komplexitätsklassen / Zeitkomplexität

Die O-Notation zeigt die **obere Schranke** (Grenze)
der **Zeitkomplexität (Laufzeit / Operationen).**

O(f(*n*)) beschreibt das **Wachstum** in Bezug auf *n* (Anzahl Elemente).

O(1)	konstanter Aufwand
O(log n)	logarithmischer Aufwand
O(n)	linearer Aufwand
O(n·log n)	linear-logarithmischer A.
$O(n^2)$	quadratischer Aufwand
$O(n^k)$, k>1	polynomialer Aufwand
$O(k^n)$	exponentieller Aufwand
O(n!)	faktorieller Aufwand

Nicht in **Polynomialzeit** berechenbar.
Anzahl mögliche Kombinationen oder Permutationen.

Weshalb interessiert nur die asymptotische Laufzeit?
Laufzeit eines Algorithmus ist besonders für **grosse Eingaben** interessant.
Ausschlaggebend für Kompläxität sind Loops.

P ≙ Lösung *finden* in Polynomzeit
NP ≙ Lösung *verifizieren* in Polynomzeit

NP – Nichtdeterministisch polynomiell

- Probleme mit **nichtdeterministischer** polynomieller Zeitkomplexität.
- Nicht von einer deterministischer* TM in Polynomzeit lösbar, aber in Polynomzeit **verifizierbar**. Nicht Polynomzeit: **O(2^n)** oder **O(n!)**

P – Polynomiell

- Alle von einer deterministischer TM in **Polynomzeit** lösbar: bis **O(n^k)**
  Es gibt ein **Entscheidungsverfahren (While-Programm).**

* deterministisch: Alle Übergänge in einer TM sind bestimmt.

O(2^n)
```
fn recursive(n)
 recursive(n-1)
 recursive(n-1)
```

O(!n)
```
fn recursive(n)
 for n
  recursive(n-1)
  recursive(n-1)
```

O(k^n)
```
fn recursive(n)
 for n
  recursive(n-1)
```

O(x · y · z)
```
for x
 for y
  for z
```

Schleifen-Index ändert sich nicht linear:

```
i = 1;
while (i <= n) {
    s;          s = Statement; O(1)
    i = i * 2;
}
```

Aufwand in O-Notation:

```
O(log n)

i nimmt Werte 1, 2, 4, 8, … an.
Es sind 1 + log2n Durchläufe.
```

Schleifen-Indizes hängen voneinander ab:

```
for (i = 0; i <n; i ++) {
    for (j = 0; j < i; j++) {
        s;          s = Statement; O(1)
    }
}
```

Aufwand in O-Notation:

```
O(n^2)

Die innere Schleife wird
1, 2, 3, 4, … n mal durchlaufen.
s wird n(n+1)/2 mal ausgeführt.
```

```
int n = r;
while (n > 0) {
    n = n / 3;
}
```

```
O(log n)
```

```
for (int i = 0; i < n; i++) {
    for (int j = n; j > 0; j = j/2) {
        s;          s = Statement; O(1)
    }
}
```

```
O(n) * O(log n) = O(n * log n)
```

Worst-Case (Big-O)
$f \in \mathcal{O}(g)$ *f* wächst asymptotisch **maximal** so schnell wie *g*.
Obere Schranke. Bsp: Falls *f(x) < g(x)* dann *f = O(g)*

Average-Case (Theta)
$f \in \Theta(g)$ *f* und *g* sind asymptotisch gleich (*f* und *g* wachsen gleich schnell)

Best-Case (Omega)
$f \in \Omega(g)$ *f* wächst asymptotisch **mindestens** so schnell wie *g*.
Untere Schranke

Rekursion

Ein Algorithmus / eine Datenstruktur heisst rekusiv definiert, wenn er/sie sich selbst enthält. Z.B. Methoden, die sich selbst aufrufen.

Rekursive Methoden können **immer in nichtrekursive,** iterative (= Schleifen) Methoden umgewandelt werden.

Nachteile:

- **Stackoverflow:** Für jeden Methodenaufruf wird ein neues Stackframe benötigt (Speicher für Methoden-Variablen).
- **Indirekte Rekusion:** Mehrere Methoden rufen sich gegenseitig auf. Häufge Fehlerquelle
- Weniger effizientes **Laufzeitverhalten**.

Äquivalent zur **vollständigen Induktion.** Unterscheiden zwischen zwei Fällen:

1. **Verankerung (Basisfall) → terminierend**
   Z.B. fak(0) = 1
2. **Induktionsschritt (Allgemeiner Fall):**
   Z.B. fak(n) = n * fak(n–1) für n > 0
   Man zerlegt das Problem, bis man auf den Basis Fall kommt.

Vorlage (**endrekusives** Programm)

```
public int p(int n) {
   if (basecase) {
      // behandelt Basis Fall
   } else {
      p(n-1); // behandelt allg. Fall
   }
}
```

Endrekusiv: Wenn rekusiver Aufruf letzte Aktion ist und keine vorangehenden Variablen zugewiesen (allocated) wurden.
Vorteil: Kein zusätzlicher Speicherplatz zur Verwaltung der Rekursion wird benötigt.

Endrekursive Programme dürfen **keinen rekursiven Aufruf** in einem **if-Statement** haben, nur in einem else-Statement.

Direkte Rekursion: eine Methode ruft sich selber wieder auf
Indirekte Rekursion: 2 oder mehrere Methoden rufen sich gegenseitig auf. Häufige Fehlerquelle.

Hanoi

Schritte: $O(2^n - 1)$ wobei n = Anzahl Scheiben
1. bewege Stapel (n-1) von A nach C
2. bewege grösste Scheibe von A nach B
3. bewege Stapel (n-1) von C nach B

Fibonacci

Definition: F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2) für n > 2

$O(2^n)$

```
public int fib(int n) {
   if (n == 0) return 0;
   else if (n == 1) return 1;
   else return fib(n-1) + fib(n-2);
}
```

O(n)

```
def fib(n):
    if n < 2: return n
    a = 0 # c[-2]
    b = 1 # c[-1]
    c = 0 # current
    for i in range(2, n + 1):
        c = a + b
        a = b
        b = c
    return c
```

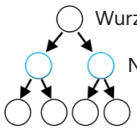
Snowflake

```
double ang = 360. / sides;
for (int i = 0; i < sides; i++) {
  snowflake(depth, 1);
  turtle.turn(-ang);
}

void snowflake(int stufe, double dist) {
  if (stufe == 0) {
    turtle.move(dist);
  } else {
    stufe--;
    dist = dist / 3;
    snowflake(stufe, dist);
    turtle.turn(60);
    snowflake(stufe, dist);
    turtle.turn(-120);
    snowflake(stufe, dist);
    turtle.turn(60);
    snowflake(stufe, dist);
  }
}
```

Bäume

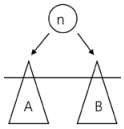
Beispiele von Bäumen (nicht unbedingt Binär):
Dateisystem, HTML-Dokument, Polnische Notation



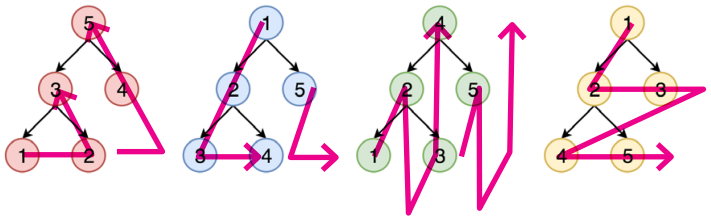
Wurzel (root)
Nachfolger (child), Geschwisterknoten
Blattknoten (leaf)

Gewicht/Grösse: Anzahl Knoten total
Höhe/Tiefe: Anzahl Ebenen

Traversierung



Postorder (A, B, n) **Preorder** (n, A, B) **Inorder** (A, n, B) **Levelorder** n, a0, b0, a1, a2, b1, b2, ...



Inorder = **Aufsteigende Reihenfolge für sortierter Binärbaum.**

Interval-Traversierung: Postorder-Traversierung in einem Wertebereich (Interval), z.B. im Interval [2, 4]

```
// Iterator
Iterator it = new InOrderIterator(tree.root);
while(it.hasNext()) visitor.visit(it.next());
String result = visitor.toString();

// Visitor pattern
Treverser traverser = new Treverser(tree.root);
traverser.iterateInOrder(visitor); // implements Iterator
String result = visitor.toString();
```

Visitor Pattern

Verarbeitet die einzelnen Elemente eines Bauems.
Idee: Baum bestimmt die Traversierung nicht. Die Verarbeitung der Knoten wird an eine Visitor-Klasse ausgelagert.

```
interface Item {
    public int accept(ItemVisitor visitor);
}

class Book implements Item {
    private int price;

    @Override
    public int accept(ItemVisitor visitor)  {
        return visitor.visit(this);
    }
}

interface ItemVisitor {
    int visit(Book book);
    int visit(Fruit fruit);
}

class ItemVisitorImpl implements ItemVisitor {
    @Override
    public int visit(Book book) {
        return book.getPrice();
    }
}

Item[] items = new Item[]{ book, fruit };
ItemVisitor visitor = new ItemVisitorImpl();

int sum = 0;
for(ItemElement item : items) {
    sum += item.accept(visitor);
}
```

```
class MyVisitor<T> implements Visitor<T> {
    StringBuilder output;

    MyVisitor() {
        output = new StringBuilder();
    }

    // Called for each element in the tree
    @Override
    public void visit(T s) {
        output.append(s);
    }

    public String toString() {
        return output.toString();
    }
}

Visitor<String> visitor = new MyVisitor<>();
tree.traversal().inorder(visitor);
String result = visitor.toString();
```

Primzahl

```
def is_prime(n):
    if n < 2: return False
    divisors = [2, 3, 5, 7]
    if n in divisors: return True
    for i in divisors:
        if n % i == 0: return False
        # if n - np.floor(n / i) * i == 0: return False
    return True
```

Binärbaum (unsortiert, unbalanciert)

Max. Knoten unterste Ebene: $2^{(h-1)}$
Max. Knoten: $2^h - 1$
Min. Knoten: h (entarteter Baum, entsprich einer Liste)
Mindesthöhe:
⌊ $\log_2$(max. Anzahl Blattknoten) ⌋ + 1
⌈ $\log_2$(max. Anzahl Knoten + 1) ⌉
Maximalhöhe: Anzahl Blattknoten (= entarteter Baum)

voll (full)
Jeder Knoten (ausser Blatt) besitzt **zwei Kinder.**

vollständig / komplett (complete)
Alle Ebenen bis auf letzte voll gefüllt und letzte Ebene ist **linksbündig**

voll und vollständig
Alle Ebenen sind voll gefüllt

Übersicht

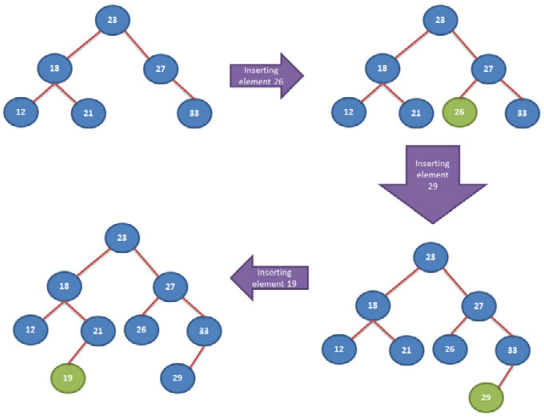
- **Binärbaum:** unsortiert unbalanciert
- **Suchbaum:** sortiert unbalanciert = sortierter Binärbaum
- **AVL-Baum:** sortiert balanciert = balancierter Suchbaum

Suchbaum (sortiert, unbalanciert)

Suchen: O($\log_2$ n), n = Anzahl Elemente
Bei einem (vollständigem) sortiertem Binärbaum.
1000 Elemente → max. 10 Schritte. 1 Mio. Elemente → max. 20 Schritte.

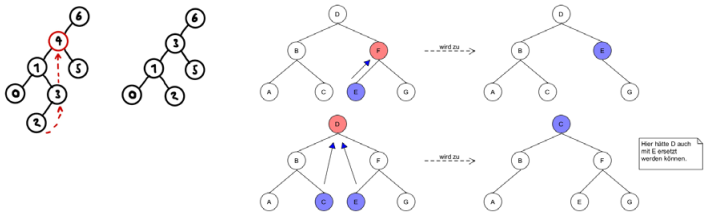
Worstcase Suchen: O(n). Beim Einfügen von neuen Elementen werden diese unten angehängt. Der Baum degeneriert zu einer «Liste» (= unbalanciert).

Einfügen:



Reihenfolge in **Inorder**-Traversierung = Aufsteigende Reihenfolge

Löschen:



So bleibt der Baum möglichst ausgeglichen.
Es gilt: **Ersatzknoten muss <= gelöschter Knoten**

Problem: Neue Knoten können nur unten angehängt werden.
Im schlimmsten Fall degeneriert der Baum zur Liste.
Suche ist dann im schlimmsten Fall **O(n).**

Ziel ist eine möglichst geringe Tiefe.

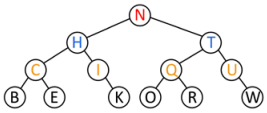
Duplikate: Keine neuen Knoten werden angleget, sondern es wird eine **Liste** bzw. ein interner **Zähler** der Elemente geführt.

Traversierung
Inorder = Aufsteigende Reihenfolge für sortierter Binärbaum.

Balancierte Bäume

Bsp. Balancierten Binär-Baum erstellen aus «Thequickbrown»
1. Sortieren
2. Sukzessive Teilung der Hälften

B C E H I K N O Q R T U W



Beim **Einfügen, Löschen** oder Ändern muss ein **vollständig balancierter** Baum möglicherweise ausbalanciert bzw. neu-erschaffen werden. **Aufwand: O(n)**

Kompromisslösung → AVL-Baum

AVL-Baum

Balancierter binärer Suchbaum

Ausgleichedingung: Für jeden Knoten gilt: Die **Differenz** der Höhe der beiden Teilbäume darf **höchstens 1 sein.**

Beim **Einfügen, Löschen** oder Ändern wird der Baum **re-balanciert,** damit die Ausgleichedingung erhalten bleibt. Eine Höhendifferenz von 3 ist nicht möglich, weil nach jedem Einfügen/Löschen ausbalanciert wird.

Degenerierung zu einer Liste ist unmöglich.

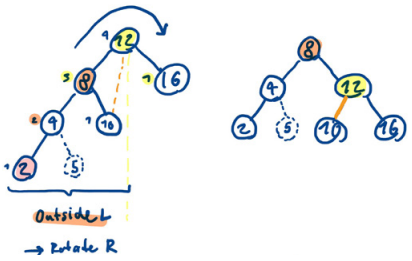
Suche und **Einfügen: O(log n)**

Balancierung und Rotation

Zeitkomplexität **O(1)**

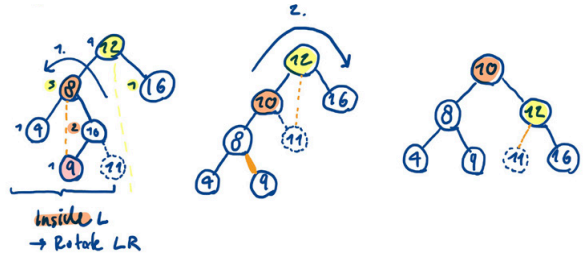
Outside Left → Rotation R
Outside Right → Rotation L
Inside Left → Rotation LR
Inside Right → Rotation RL

Bsp: Es wird von Knoten «12» ausgegangen. «2» wird eingefügt.
Jeder Knoten führt Buch, wie tief die daruntergelegenen Teilbäume sind.




```
// rotateR
Node newRoot = oldRoot.left; (8)
oldRoot.left = newRoot.right; (10)
newRoot.right = oldRoot; (12)
oldRoot.height = max(height(oldRoot.left), height(oldRoot.right)) + 1;
newRoot.height = max(height(newRoot.left), newRoot.height) + 1;
return newRoot;
```

Bsp: Es wird von Knoten «12» ausgegangen. «9» wird eingefügt.




```
// rotateLR
oldRoot.left = rotateL(oldRoot.left);
return rotateR(oldRoot);
```

B-Baum

Vollständig balancierter Baum (nicht-binär).
Vollständig = Alle Blätter haben die gleiche Tiefe (bis auf letzte voll gefüllt)

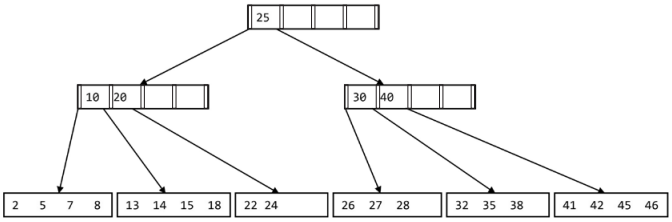
Anwendungen:
- **Filesystem/HD:** Daten können blockweile gelesen werden. Vergleich: Binärbäume eignen sich nur für Strukturen im Hauptspeicher (RAM).
- **Datenbanken**

Möglichst **breiter** Baum mit **geringer Tiefe.**
Disk- oder Speicher-Zugriffe werden minimiert.

- Ordnung *n* = Kinder (Verweise)
- Jeder Knoten enthält max. *n* Kinder (Verweise/Folgeknoten)
- Jeder Knoten enthält min. $\lfloor (n\text{-}1)/2 \rfloor$ und max. *n*-1 Werte (Schlüssel).
- Ausser Wurzel hat 1 bis *n*-1 Werte
- Innerhalb eines Knoten sind alle Werte (Schlüssel) **sortiert**

Suche: $O(\log_{\text{Ordnung}-1}(\text{Max. Anzahl Werte})$
Tiefe: $\approx \lfloor \log_{\text{Ordnung}}(\text{Max. Anzahl Werte}) \rfloor$
Max. Anzahl Elemente: $\sum n^{(\text{Level}-1)}$ Für jedes Level gibt es maximal $n^{(\text{Level}-1)}$

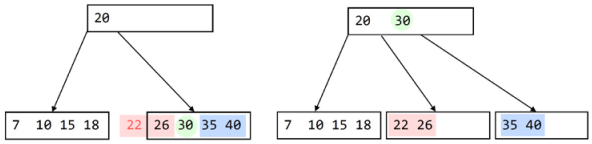
Bsp. mit Ordnung *n* = 5



Einfügen

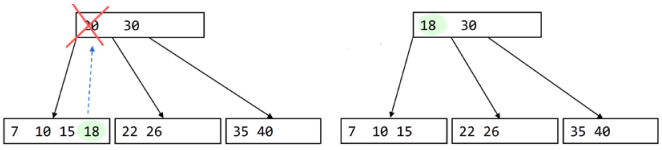
Eingefügt wird **immer in den Blättern** (solange Platz).
Ist ein Knoten bereits voll gibt es ein **Überlauf** nach dem einfügen
→ mittleres Element «heraufziehen»

Überlauf → Teilung: 22 wird eingefügt.



Der zweite Knoten muss **geteilt** werden.

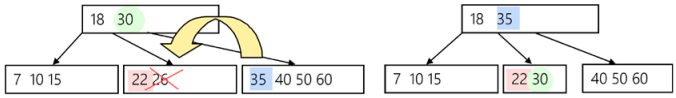
Löschen



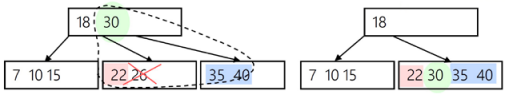
Unterlauf → Rotation oder Merge: 20 wird gelöscht.

Bei **Unterlauf,** d.h. Blatt enthält weniger als $\lfloor (n\text{-}1)/2 \rfloor$ Werte. Zwei Szenarien:

1. **Rotation** wenn Nachbarsblatt genügen Werte hat.



2. **Merge** wenn Nachbarsblatt nicht genügen Werte hat.



B⁺-Baum

Informationen sind nur in den Blättern gespeichert.

2-3-4-Baum

Spezialfall des B-Baums mit Ordnung n = 4
Jeder Knoten bestzt min. zwei und max. 4 Kinder.

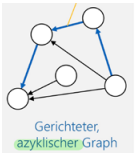
Graphen

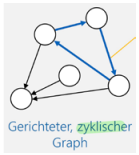
G = (V, E) V = Knoten (Vertices), E = Kanten (Edges)

Anwendungen:
- Kürzeste Verbindung
- Maximaler Durchsatz (Maximal Flow)
- Kürzester Weg
- Reihenfolge bestimmen (Topological Sort)

Dichte des Graphen (= Verhältnis): Anzahl Kanten / Anzahl möglicher Kanten
dünn: Knoten > Kanten
dicht: Knoten < Kanten

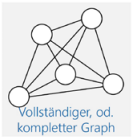
Azyklisch / zyklisch

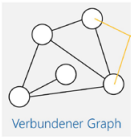




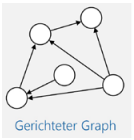
Vollständig (komplett): Jeder Knoten ist direkt mit jedem anderem Knoten verbunden. Anzahl Kanten: **n(n–1) / 2**

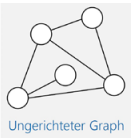
Verbunden: Jeder Knoten ist erreichbar





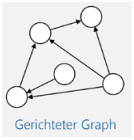
Gerichtet / Ungerichtet:

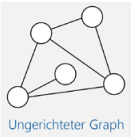




Kanten in einem ungerichtetem Graphen eigentlich zeigen in beide Richtungen.

Gewichtet / Netwerk

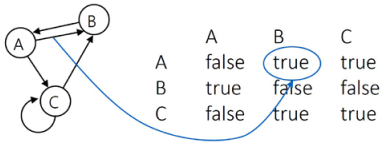




Implementationen

Adjazenz-Liste
Jeder Knoten führt eine Liste zu benachbarten Knoten (= Liste von Listen).

Adjazenz-Matrix



	A	B	C
A	false	true	true
B	true	false	false
C	false	true	true

Anwendung:
- **Gewichteter** Graph: **Gewichtung** der Kanten zwischen Knoten.
- **Gerichteter** Graph: **Verbidnung** der Kanten zwischen Knoten.

Vergleich

	Vorteile	Nachteile
Adjazenz-Matrix	Feststellen, wer mit wem verbunden ist, ist sehr effizient.	hoher Platzbedarf und teure Initialisierung: wachsen quadratisch mit $O(n^2)$.
Adjazenz-Liste	Es wird kein Speicher «verschwendet», Bedarf ist proportional zu n+m.	Effizienz der Kantensuche abhängig von der mittleren Anzahl ausgehender Kanten pro Knoten.

n ist dabei die Knotenzahl und **m** die Kantenzahl eines Graphen G.

Traversierungen

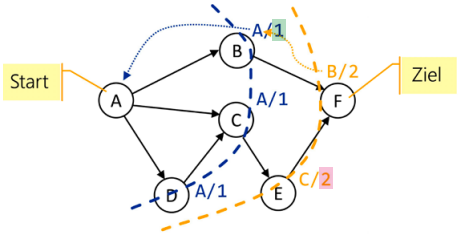
1. Tiefensuche (depth-first search)
= entspricht **Preorder**-Traversierung bei Bäumen.

Bei einem **zyklischen** Graphen müssen die bereits besuchten Knoten markiert werden.

2. Breitensuche (breadth-first search)
= entspricht **Levelorder**-Traversierung bei Bäumen.

Breadth First Search (Breitensuche)

Kürzester Pfad für ungewichtete oder ungewichte Graphen.
Alle Knoten werden besucht: O(Kanten + Knoten²) ≈ **O(Knoten²)**



Distanz und **von welchen Knoten** aus besucht wird, wird eingetragen wenn:
- Knoten noch nicht besucht ist, oder
- Distanz kleiner als bestehende Distanz ist

Vom Endpunkt kann dann rückwärts der kürzeste Pfad gebildet werden.
Leicht Perfomanter ist aber der **Dijkstras Algorithmus.**

Dijkstras Algorithmus

Kürzester Pfad für ungewichtete oder ungewichte Graphen.
Ist ein **gieriger Algorithmus.**

Unterschied zum vorherigem Algorithmus:
Die Knoten werden nicht zufällig besucht sondern nach Gewichtung.

O(Kanten + Knoten²) ≈ **O(Knoten²)**
Worst case: vollständiger (kompletter) Graph (alles miteinander verbunden)

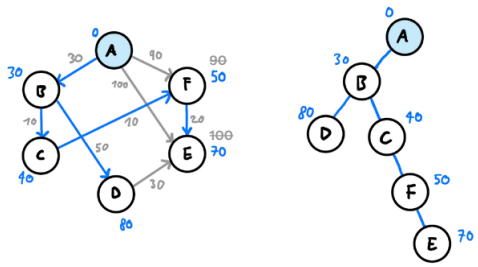
Berechnet ausgehend von einem Knoten der kürzeste Pfad zu allen anderen Knoten.

Der Graph wird in 3 Listen/Gruppen aufgeteilt:
- Besuchte Knoten (Distanz bekannt / bereits berechnet)
- Aktuell benachbart und unbesuchte Knoten
- Unbesuchte Knoten

1. Berechne für alle benachbarten unbesuchten Knoten die neuen Gewichte.
2. Besuche unter allen benachbarten Knoten denjegigen mit der kleinsten Gewichtung (kürzeste Distanz) → **Tiefensuche** (Preorder).

Soll nur der minimale Pfad für einen bestimmten Knoten besucht werden, kann die Suche abgebrochen werden, sobald dieser besucht wurde.

Das Resultat ist ein **Baum** mit dem **kürzesten Weg zu jedem Knoten** im Graphen. Bsp. ausgehend von Knoten A.



Gierige Algorithmen (Greedy)

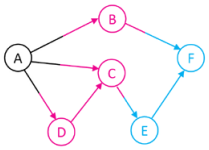
- **Gradient-Descent / Gradientenverfahren**
- **Dijkstras Algorithmus**

Wählt den Folgezustand aus, der **zum Zeitpunkt** der Wahl **den grössten Gewinn verspricht.**

Schnell, können aber in lokalen Maxima/Minima stecken bleiben.
→ Lösung: Stochastische Suchverfahren

Topologisches Sortieren

Nur für **gerichtete, azyklische** Graphen (DAG).
Es wird eine lineare Anordnung der Knoten gesucht.



Nach Anzahl eingehenden Kanten sortieren.
A(0), D(1), B(1), C(2), E(1), F(2)

Bsp: Kanten geben die **Abhängigkeiten** zwischen Modulen in einem Programm an. Die Topologisches **Sortierung** zeigt eine mögliche **Compilationsreihenfolge**.

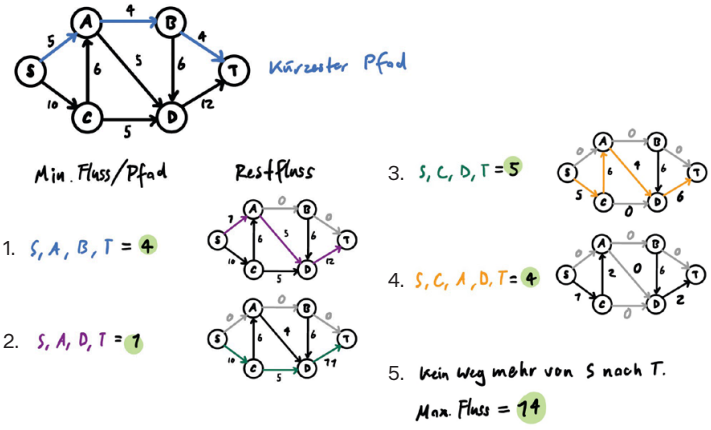
Maximaler Fluss

Wieviel kann maximal A nach Z fliessen?

Was in einen Knoten hinein fliesst, muss auch wieder heraus.
Es gibt mehrere Lösungen mit demselben Fluss.

Anwendungen:
- Verkehrsleitsystem
- Stromleittechnik
- Computernetzwerk

Ford-Fulkerson-Algorithmus



NP-Probleme

Für all diese Probleme entsteht ein **Entscheidungsbaum.**

Backtracking / Trial und Error

Nur für **Bäume** oder **azyklische Graphen.**

O(n!) oder **O(kⁿ)** bzw. **O(**max. Anzahl Kindsknoten $^{\text{max. Tiefe}}$**)**

Tiefensuche (depth-first search, **Preorder**)

Problem:
- Zeit und rechenintensiv
- Lösung liefern nicht zwingend das beste Resultat.

Traveling Salesman Problem: O(n!)
Jeder Knoten in einem Graph soll genau einmal besucht werden.
Greedy-Algorithmus (Approximation): O(n² · log(n²))

Mausproblem: O(kⁿ) bzw. **O(**max. Anzahl Kindsknoten $^{\text{max. Tiefe}}$**)**

Springerproblem (Schach): **O($8^{n \cdot n}$)**, 8 = Zugmöglichkeiten, n = Brettgrösse
Springer soll nacheinander alle Felder genau einmal besuchen.

Damenproblem (Schach): **O(!n)**, n = Anzahl Damen
Eine Stellung für acht Damen, so dass keine zwei Damen sich gegenseitig schlagen können.

Rucksackproblem: O(2^n) = **Potenzmenge**, n = Anzahl Gegenstände

Anwendungen

- Transportwesen (Beladung eines Kontainers)
- Stundenplan
- Spiele: Bester Zug im Schach

Zielfunktion

Die Zielfunktion *f*(Knoten) weist einem Knoten einen geschätzten besten Wert zu. Die Idee einer Zielfunktion ist, dass durch **Pruning** nur ein **Teilbaum** des **Entscheidungsbaums** untersucht werden muss.

Mittels Zielfunktoin lässt sich ein Problem der **kombinatorischen Explosion eindämmen**. Es wird immer die bestmögliche Lösung gefunden.

Heuristik: Methoden, die mit begrenzten Informationen und wenig Zeit dennoch zu praktikablen Lösungen kommen:

Kombination aus Breitensuche und Tiefensuche.

Obere Schranke
= Zielfunktion. Liefert die bestmögliche Lösung eines Pfades zum aktuellen Zeitpunkt.

Best-first Search
Die Obere-Schranke bestimmt welche Kind-Knoten zuerst besucht werden.

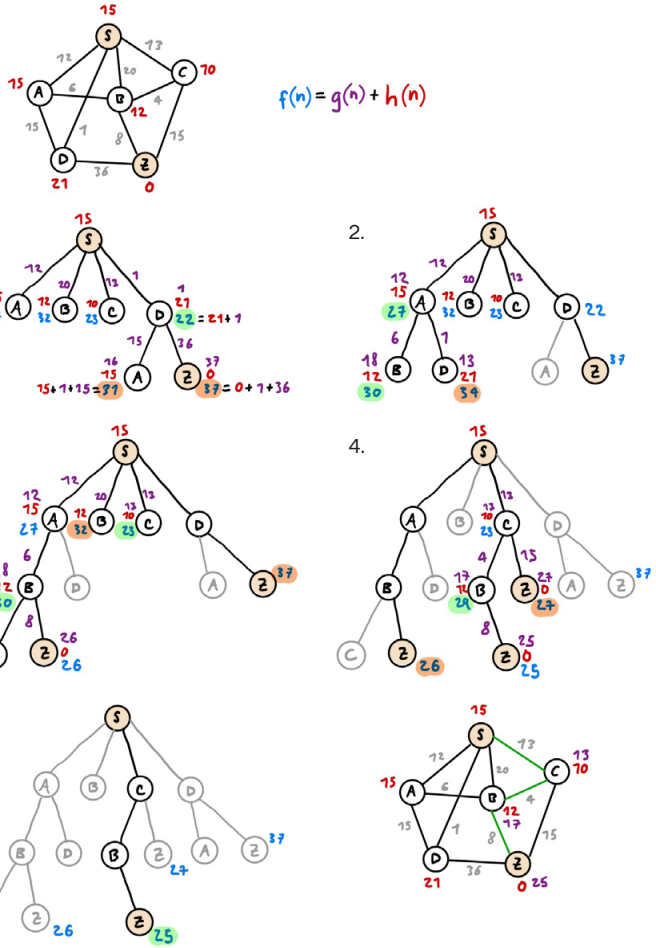
Pruning / Bound
Durch Festlegen von Schranken können Teilbäume abgeschnitten werden, wenn es bereits eine bessere gefundene Lösung gibt, als die mittles **oberen Schranke** geschätze Lösung.

Branch
Aufteilen der Lösung in mehrere, einfachere Teilschritte, z.b in einem Baum.

A*-Algorithmus

Kürzester Pfad. **O(Knoten²)**

f(n) = **Zielfunktion,** Summe aller Kosten von Start nach n: **f(n) = g(n) + h(n)**
h(n) = Heuristischer **Wert von n nach Ziel** (z.B. Luftlinie)
g(n) = **Kosten zwischen** zwei benachbarten **Knoten**.



Dijkstra vs. A*

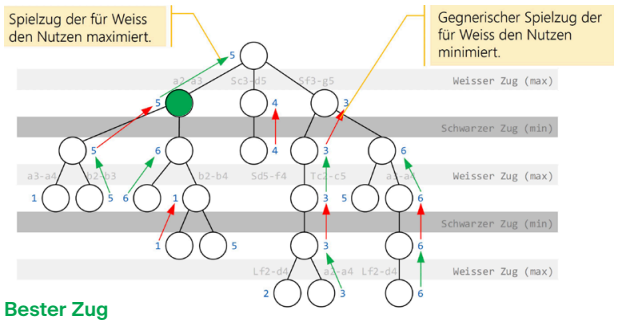
- Dijkstra ist in der Praxis etwas langsamer als A*
- Der Baum von A* enthält nur den **optimalen Weg**
  Der Suchbaum von Dijkstra hingegen umfasst den gesamten Graphen.

Minimax-Algorithmus

Andwendung: Spiel mit abwechselnden Zügen.
Der Minimax Algorithmus untersucht den vollständigen Suchbaum.
Bei jedem Zug wird ein neuer Baum erstellt.

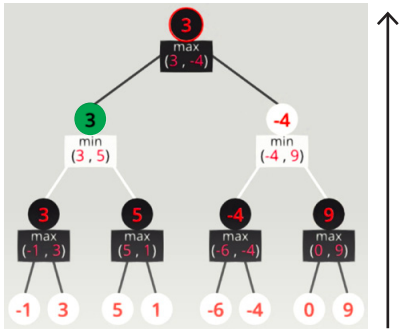
Es wird ein **Entscheidungsbaum aller möglichen Züge** erstellt.
Jedem Knoten (= Spielpositon) wird einen Wert zugewiesen.
Von den Blattknoten wird nun abwechslungsweise maximiert und minimiert.
Weiss versucht zu maximieren, Schwarz zu minimieren.

Bsp, Weiss ist am Zug:



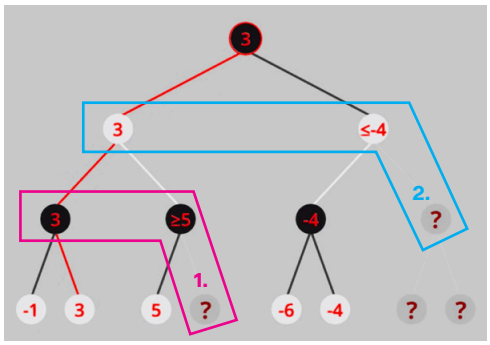
Bester Zug

Bsp. Schwarz ist am Zug.



Alpha-Beta-Pruning

Optimierter Minimax Algorithmus durch Pruning.



1. Muss nicht untersucht werden, weil Weiss min(3, 5) = 3 wählen wird.

2. Muss nicht untersucht werden weil Schwarz max(3, –4) = 3 wählen wird.

Postorder

Horizont-Effekt

O(Mögliche Züge Anzahl durchgeführte Züge **)**
Bsp: 10 Mögliche Züge, n = Anzahl durchgeführte Züge → $O(10^n)$.
d.h. bei jedem weiteren Zug muss 100-mal mehr berechnet werden.

Die gefundene Lösung kann sich beim nächsten Schritt als schlecht erweisen.

Wahl der Datenstruktur

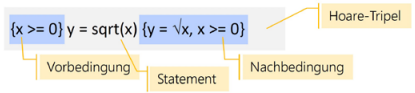
Daten müssen:

– **sortiert** sein (für schnelle Suche)	→ **Tree** (z.B. AVL-Baum, Sortierbaum)
– **nicht sortiert** sein	→ **HashMap** (Duplikate erlaubt) **HashSet** (ohne Duplikate)
– **keine Duplikate** enthalten	→ **Set** (z.B. HashSet, TreeSet)
– eine **klare Reihenfolge** haben	→ **Queue** (ohne Priorität) **PriorityQueue** (mit Priorität)
– **keine klare Reihenfolge** haben	→ **LinkedList** (für schnelles einfügen) **ArrayList** (für schnelles lesen)

Binäres Suchen

Vorbedingung: Aussage, die **vor** der Ausführung gilt.
Invariante: Aussage, die **während** dem ganzen Zeitraum gelten muss.
Nachbedingung: Aussage, die **nach** der Ausführ gilt.

Hoare-Tripel



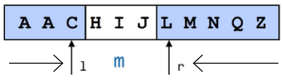
Bsp. Invariante

$$\{\forall x_i; i < k; x_i \neq S\} \wedge k = 0$$

Bsp: {x >= 0} x = x + 1 {x >= 1}

Wert in einem Arrays suchen

Das Arrays muss **sortiert** sein. **O(log n)**

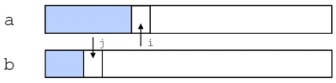


Wiederhole bis Wert S gefunden oder nicht gefunden:

1. nehme `m` als Index zwischen `l` und `r`
2. falls `a[m] == S` → gefunden (Abbruch)
3. falls `a[m] < S` → `l = m`
4. falls `a[m] > S` → `r = m`
5. falls `l + 1 >= r` → nicht gefunden

Werte in mehreren Arrays suchen

Die Arrays müss **sortiert** sein.



`b[j] < a[i]` → `j` um 1 erhöhen
`b[j] > a[i]` → `i` um 1 erhöhen
`b[j] = a[i]` → gefunden Wert zurückgeben.

Zeitkomplexität

Sortierter Array

- Einfügen und Löschen: O(n/2) → O(n)
- Binäres Suchen: $O(\log_2(n))$ → O(log(n))

Lineare sortierte Liste

- Einfügen und Löschen: O(n/2) → O(n)
- Suchen: O(n/2) → O(n)

Sortierter Binärbaum (balanciert)

- Einfügen und Löschen: $O(\log_2(n))$ → O(log(n))
- Suchen: $O(\log_2(n))$ → O(log(n))

Hashing

Idee: Jedem Wert wird mit einem eindeutigen Schlüssel assoziert (z.B. AHV-Nr. → Person). Problem: Wertebereich der Schlüssel kann gross sein. Lösung: Mit einer Hash-Funktion wird ein **grosser Wertebereich auf einen kleineren abgebildet.**

Bsp. einfache Hash-Funktion: h(k) = k mod M

Hashing Probleme

- Geeignete Hash-Funktion finden. Ziel: Möglichst regelmässige Verteilung
- **Kollision:** Zwei Schlüssel können den gleichen Hash-Wert liefern.
  → linear / quadratic probing

Einfügen → put(key, value)

```
index = hash(key)
while(keys[index] != null && keys[index] != key) index++;
values[index] = value;
```

Suchen → get(key)

```
index = hash(key)
while(keys[index] != key) index++;
return values[index];
```

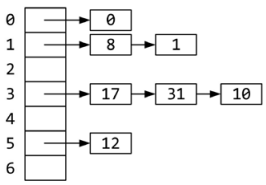
Aufwand **Einfügen/Suchen: O(1)**

Kollisionsauflösung

Kollision = gleiche Hashwerte für unterschiedliche Objekte.

Load-Faktor λ = Anzahl Einträge / Anzahl Plätze (Verhältnis)
Sagt aus, wie stark der Hash-Bereich belegt ist.

Überlauflisten (Separate Chaining)



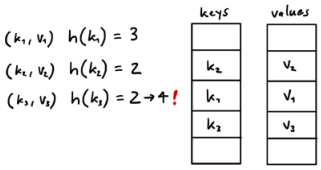
- Nachteil: Overhead durch Verwendung einer weiteren Datenstruktur.
- Vorteil: Funktioniert auch noch bei Load-Faktor λ nahe oder gleich 1.

Anzahl Einträge / Anzahl Plätze

Open Addressing

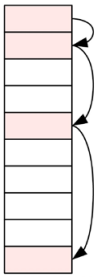
Vorraussetzung: λ < 0.8, sonst ist Performance schlecht.

Linear Probing
Schlüssel-Wert wird in den nächst freien Platz gespeichert.



Problem: **Primary-Clustering**
= Mehr Kollision an gewissen stellen.

Quadratic Probing
Bessere Performance und weniger Primary-Clustering als linear probing.



F + n²
F + 1² = F + 1
F + 2² = F + 4
F + 3² = F + 9
F + 4² = F + 16
…

Löschen

1. Es muss ge-**rehasht** werden.
2. oder gelöschte Zellen als gelöscht markieren.

Java Implementation

- **HashSet** (unsoriterte Menge ohne Dublikate)
- **HashMap**<K, V> (unsoritert)
- **TreeMap**<K, V> (sortierte)

Vorteile:

- Suchen Einfügen in Hash-Tabellen sehr effizient.
- «Einfache» binäre Bäume können bei ungünstigen Inputdaten degenerieren, Hash-Tabellen kaum.
- Der Implementationsaufwand für Hash-Tabellen ist geringer als derjenige für ausgeglichene, binäre Bäume.

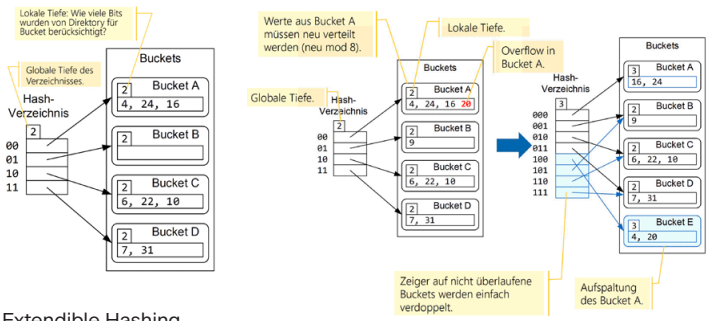
Nachteile:

- Das kleinste oder grösste Element lässt sich nicht einfach finden.
- Geordnete Ausgabe ist nicht möglich.
- Die Suche nach Werten in einem bestimmten Bereich oder das Finden z.B. eines Strings, wenn nur der Anfang bekannt ist, ist nicht möglich.

→ Geeignet wenn die Reihenfolge nicht wichtig ist.

Extendible Hashing

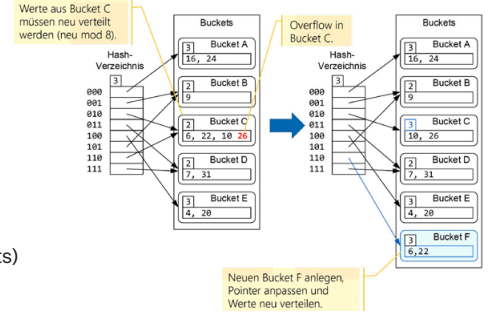
Bei Overflow wenn Lokale Tiefe = Globale Tiefe



Extendible Hashing nutzt eine **dynamisch** wachsende und schrumpfende **Hash-Struktur.**

Vorteil:
- Buckets müssen nicht hintereinander in Memory liegen.
- Kleines Hash-Verzeichnis (letzte Bits)

Bei Overflow wenn Lokale Tiefe < Globale Tiefe



Suchen in Texten

Java

indexOf(String)
Brute-Force-Allgorithmus

1. Sucht nach der Position des Anfangsbuchstaben.
2. Veruscht das restliche Wort zu matchen

Vorteil: Keine Initialisierungskosten
und Speicheraufwände.
Nachteil: nur für kurze Strings geeignet

O(m · n)

Knuth-Morris-Pratt-Algorithmus

Suchen in Texten
O(n + m)

Problemstellung: Um die Länge des abgesuchten String verschieben.
Problem: Subpatterns werden nicht erkannt. Bsp:

```
A A A I I I
A A I
    A A I   Subpattern übersprungen!
```

Idee: Pattern zurückverschieben um allfällige Subpattern (Präfix) zu erkennen.

Phase 1 (Pre-processing)

Next-Tabelle erstellen

Vorgehen

```
l=0, p=1, next[0]=0
–  Match: n[p]=l+1 ,p++, l++
– !Match:
  – l == 0: n[p]=0, p++
  – l != 0: l=n[l–1]
```

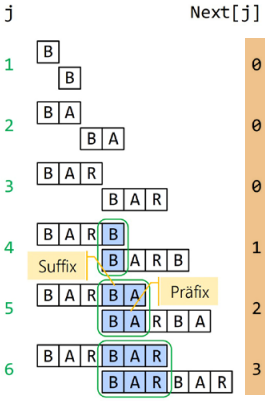
Beim Erstellen der Next-Tabelle werden Präfix und Suffix der Subpattern des Suchmusters (Patterns) verglichen.

Phase 2

Textsuche

Anhand der Next-Tabelle wird bei einem Match (Postfix) zurückverschoben.

Veranschaulichung:
B A R B A R A



Letzter Buchstabe (A) wird ignoriert

Bsp:

Phase 1
Next-Table

n	a	n	o	
l	p			
0				
l		p		
0	0			
	l		p	
0	0	1		
l			p	
0	0	1		
l				p
0	0	1	0	

Phase 2
Textsuche

n	e	n	a	n	a	n	o
n							
	n						
		n	a	n			
				←	n		
				n			
				n	a	n	o

Match-Länge = 1, Verschub = **0**

Match-Länge = 3, Verschub = **1**

Next-Table

1	2	3	4	Match-Länge
0	0	1	0	Verschub nach Links

Inverter Index

Indexdatenstrukur: Bildet Inhalt (Wort, Zahl, etc.) auf seine Position in einem Dokument ab: (Dokument, Position))

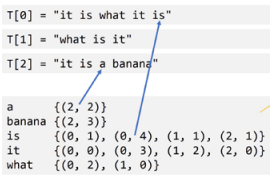
Andendungen: Suchmaschinen, Wieviele Links verweisen auf das Dokument

Implementtaion: Der invertierte Index kann z.B. als **B-Baum** implementiert werden.

Suche: Ist der invertierte Index unsortiert, dann ist die Suchzeit, falls der Index **nicht mittels Hashing** implementiert ist, **O(n);** n = Länge des Index.

Mittels invertiertem Index lässt sich die **Häufigkeit von einzelnen Wörter** bestimmen.

Mach Sinn wenn: Anzahl Index (Key) < Anzahl Tuples (Werte)
Suche: **O(Anzahl Wörter)**



Problem: Wörter müssen **normalisiert** sein (keine Rechtschreibefehler)
→ Levenshtein-Distanz

Levenshtein-Distanz

$L(w_1, w_2)$ gibt die **minimale Anzahl** von **Einfüge- Lösch- und Ersetz-Operationen** an, die man braucht um aus dem Wort w_1 das Wort w_2 zu bilden.

Operationen: insert(c), update(c → d), delete(c)

Ähnliche Wörter mit Schreibfehlern können so gesucht werden.

O(n · m)

Bsp: Sunday → Saturday

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Gleich

+0	+1
+1	min

Ungleich

+1	+1
+1	min

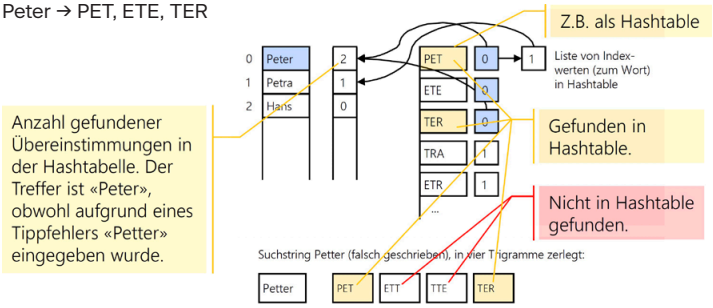
insert(a)
insert(t)
update(n → r)

D[i, j] = min
- D[i-1, j-1] + 0; falls A[i] = B[j]
- D[i-1, j-1] + 1; falls update(A[i]->B[j])
- D[i, j-1] + 1; falls insert(B[j])
- D[i-1, j] + 1; falls delete(A[i])

Trigramm-Suche

Anwendung:
- Fehlertolerante Suche.
- Effizient für grosse Datenmengen

Idee: Wort wird in 3er-Gruppen unterteil, die als Index dienen.
Peter → PET, ETE, TER



Soundex (Phonetische Suche)

Algorithmus zur Indizierung von Wörtern und Phrasen nach ihrem Klang.
Jeder Konsonant (Sprachlaut) kriegt eine Nummer zugewiesen.
Jeder Soundex-Code besteht aus einem Buchstaben gefolgt von drei Ziffern.
Britney → BRTN → **B635**

Reguläre Ausdrücke

Suchen nach Mustern → State-Maschine (Endliche Automaten)

Sortieren Übersicht

Inplace-Verfahren: Es wird fast kein zusätzlicher Speicherbedarf benötigt.

Externer Sortiervorgang: Wenn nicht alle Datensätze gleichzeitig im Arbeitsspeicher gehalten werden können → Merge Sort

Sortierschlüssel: Kriterien, nach denen Datensätze sortiert oder gesucht werden. z.B. «Alter» = nicht eindeutig, d.h. viele haben den gleichen Schlüssel. Besser als Tupel (Name, Vorname, Alter) = eindeutig.

Collation (= Einsortierungsregel): Länderspezifische Sortierreihenfolge, z.B. ä vor a. → Collections.sort(list, **new Locale("de", "CH")**)

Stabilität: Ein Sorieralgorithmus heisst stabil (stable) wenn die Reihenfolge bei gleichwertiger Elemente (= gleiche Schlüssel) beibehalten wird.

	n	Worst-Case	Avg-Case	Best-Case	Stabilität	Inplace
Bubble Sort	< 1000	$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	stabil	ja
Selection Sort	< 1000	$O(n^2)$	$\Theta(n^2)$	$\Omega(n^2)$	instabil*	ja
Insertion-Sort	< 1000	$O(n^2)$	$\Theta(n^2)$	$\Omega(n)$	stabil	ja
Quick Sort	> 1000	$O(n^2)$	$\Theta(n \log n)$	$\Omega(n \log n)$	instabil	ja
Merge Sort	> 1000	$O(n \log n)$	$\Theta(n \log n)$	$\Omega(n \log n)$	stabil	nein
Distribution Sort		$O(n)$	$\Theta(n)$	$\Omega(n)$	stabil	nein

* Selection Sort kann modifiziert werden um stabil zu sein.
Heap-Sort?

Bubble Sort

Benachbarte Elemente werden so lange vertauscht, bis alles sortiert ist: Vergleichen und vertauschen.

1. Durchgang: Grösstes Element ist ganz oben. n-1 durchlaufen
2. Durchgang: Zweitgrösstes Element ist oben. n-2 durchlaufen
3. ...

Worst case: Elemente liegen in umgekehrt sortierter Reihenfolge vor.
Summelformel: (n-1) + (n-2) + ... + 2 + 1 = $n \cdot (n-1) \cdot ½$ = $O(n^2)$

Optimierung: Bei jedem Durchgang testen, ob etwas vertauscht wurde.

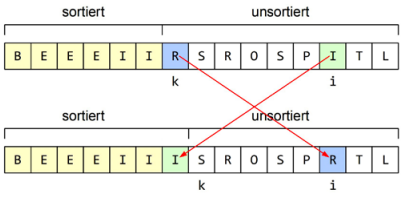
Selection Sort

Idee: In einen sortierten und unsortierten Bereich aufteilen.

Suche das **kleinste Element** im unsortierten Bereich und vertausche es mit dem ersten Element im unsortierten Bereich. Sortierer Bereich um 1 erweitern.

Vorteil zu Bubble Sort: Weniger swap-Anweisungen.

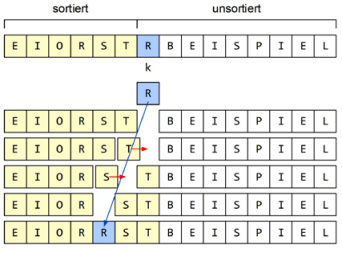
Instabil weil durch Swapen könnte ein Element mit gleichem Schlüssel hinter das andere verschoben werden.



Insertion Sort

Idee: In einen sortierten und unsortierten Bereich aufteilen.
Analog Selection Sort. Anwendung: Wenn **Elemente** einem bereits sortiertem Array **eingefügt** werden sollten.

Das erste Element aus dem unsortiertem Bereich im sortiertem Bereich einordnen. Sortierer Bereich um 1 erweitern.



Quick Sort

Inplace-Verfahren (braucht keinen zusätzlichen Speicheplatz)
Worst-case: Es bleibt bei einer Partition (weil Pivot ungünstig gewählt) = Selection Sort.

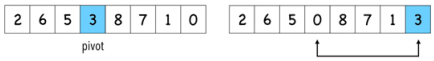
Worst Case: **O(n²)** wenn **Pivot** immer schlecht gewählt. Nicht besser als **O(n log n)** weil Sortieren (Schritt 2.) mit **Vergleich und Swap** ist **O(n)**.

Quicksort ist eine gute Wahl zum sortieren **grösserer Mengen** (>1000) weil wegen der **Rekursion** einen realitv **grossen Overhead** besteht.

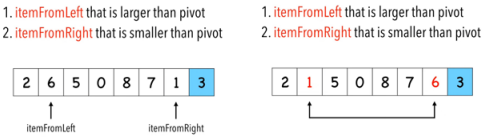
Idee: **Teile und Herrsche:** Diese Algorithmen sind typischerweise rekursiv
1. In Teilmenge zerlegen, 2. Sortieren, 3. Vereinigen

1. Pivotwahl «Median-of-three»
Sortiere erstes, letztes und mittleres Element. Neues mittleres ist unser Pivot. Ziel ist, dass beide Teilmengen möglichst gleich gross sind.

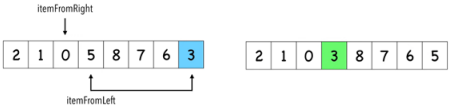
2. Sortieren:
Pivot mit letztem Element vertauschen:



Element von **links** her das **grösser** als Pivot ist vertauschen mit Element von **recht** her das **kleiner** als Pivot ist:



Wiederhole bis sich die Indexe der beiden Elemente kreuzen.
Linkes Element mit Pivot austauschen.



Wiederhole Vorgang **rekursiv** mit beidem Teilmengen links und rechts des Pivots.

Distribution Sort

Schnellster Algorithmus. Kommt ohne Vergleiche aus.

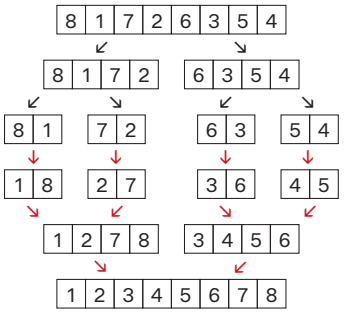
1. Elemente werden gemäss ihrem Wert/Schlüssel direkt in Array eingefügen (distributed).
2. Löcher im Array entfernen.

Allgemeines Hashing geht nicht, weil der Hash-Wert nichts über den Inhalt und Grösse des Elements aussagt.

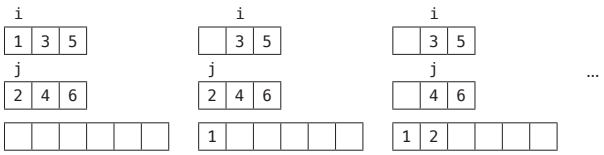
Nachteil: Nur bei Schlüssel mit kleinen Wertebereich

Merge Sort

Für interne und **externe** Sortierverfahren.
Kein Inplace Verfahren.

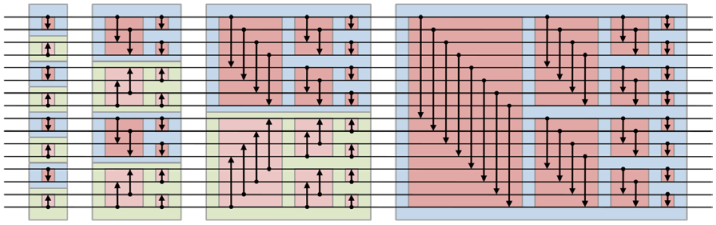


Merge (Zusammenfügen):
Teilmengen müssen bereits **sortiert** sein, können aber theroteisch mit irgendeinem Algorithmus sortiert werden.



Parallel Bitonic Merge Sort

Can be used on the GPU.
The following is a bitonic sorting network with 16 inputs (must be power of 2):



Pointer References

Hard/Strong Reference

An object can't be garbage collected if it's reachable through any strong reference.

```
List<String> list = new ArrayList<>;
list = null; // can now be collected
```

Soft Reference

All soft references to objects reachable only by soft reference should be cleared out before the OutOfMemoryError exception is thrown.

```
SoftReference<List<String>> listReference
  = new SoftReference<List<String>>(new ArrayList<String>());
List<String> list = listReference.get();
if (list == null) { /* object was already cleared */ }
```

Weak Reference

Inner value of a weak references isn't prevented from being collected. From the perspective of garbage collection, they could not exist at all.

Ein Objekt muss allenfalls nochmals neu erzeugt werden, wenn der GC dieses bereits in der Zwischnzeit gelöscht hat und ich dieses weiterhin benötige.

Garbage Collectors

Java Memory Model

Folie 13, S.22

Stack

Heap

Meta Space (Statics)

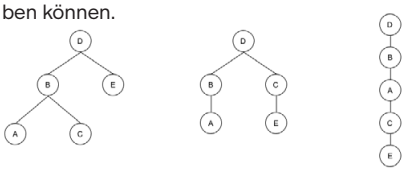
Not subject for the exams

Threads
Dynamische Speicherverwaltung
Rot-Schwarz-Bäume

Fragen

Kann man aus der Pre- und Postorder-Reihenfolge einen Baum immer eindeutig wiederherstellen?

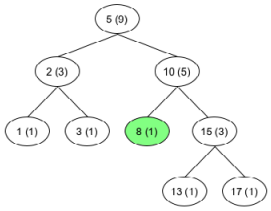
Nein, da verschiedene Bäume die gleiche Pre-Order bzw. Post-Order ergeben können.



Alle drei dieser Bäume haben für die Pre-Order Traversierung: D, B, A, C, E

In einem binären Suchbaum mit ungerade vielen Elementen (n = 2k+1) steht an jedem Knoten, wieviele Knoten der gesamte Teilbaum hat. Wie kann man mit dieser Information den Median bestimmen?

Der Median ist das Element in der **Mitte** bei der **Inorder**-Traversierung.
Die totale Anzahl Elemente kann bei der Wurzel abgelesen werden.



Entwurfsmuster für Algorithmen

- Divide-and-Conquer (Dynamic Programming)
- Greedy
- Backtracking
- Rekursion (direkt, indirekt, End-Rekursion)
- Heuristische Approximationsverfahren

Algorithmus mit Laufzeit

- $O(1)$ konstanter Aufwand
  - Einfügen eines Elements an erster Stelle in einer `LinkedList`
- $O(\log(n))$ logarithmischer Aufwand
  - Suche im AVL Baum
- $O(n)$ linearer Aufwand
  - Suche in `LinkedList`
- $O(n \cdot \log(n))$ linear-logarithmischer A.
  - Merge Sort
  - Quick Sort
- $O(n^2)$ quadratischer Aufwand
  - verschachtelte Schleife
  - Bubble Sort
- $O(n^k)$, $k > 1$ polynomialer Aufwand
  - Ellipsoidmethode
- $O(k^n)$ exponentieller Aufwand
  - Turm von Hanoi (rekursiv)
- $O(n!)$ faktorieller Aufwand
  - Traveling Salesman

Wie kann man die Fibonacci-Zahlen iterativ berechnen?

O(n)

```
def fib(n):
    if n < 2: return n
    a = 0 # c[-2]
    b = 1 # c[-1]
    c = 0 # current
    for i in range(2, n + 1):
        c = a + b
        a = b
        b = c
    return c
```

Wie lautet die **Rekursionsgleichung** für die **maximale Anzahl Vergleiche** bei **binärer Suche (sortierter Array).**

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

T(n) = Maximale Anzahl Vergleiche

+ 1 weil in jedem Schritt das Mittellement des Arrays mit dem gesuchten Element **vergleichen**.

Basisfall: F(0) = 0

Unterschiede und Gemeinsamkeiten zwischen **AVL-Trees** und **Hash-Tables.**

Gemeinsamkeiten: Beide sind abstrakte Datenstrukturen zur Speicherung von Daten. **Unterschiede:** AVL-Bäume sind balancierte binäre Suchbäume, Hash-Tabellen basieren auf Hashfunktionen und erlauben schnelleren Zugriff.

Gegeben ist eine einfach-verkettete Liste OHNE tail-Pointer. Welche Datenstruktur sollte man damit NICHT implementieren, einen Stack oder eine Queue?

Bei einem Stack ist der tail-Pointer irrelevant, da das Hinzufügen und Entfernen (Push und Pop) immer am Anfang erfolgt.

Wenn die **Queue** so implementiert ist, dass das nächste Element am Ende in der Liste ist, dauert es länger dieses letzte Element zu finden.

Sind folgende Funktionen endrekursiv?

```
public int foo(int n) {
    assert(n >= 0);
    if (n == 0) {
        return 1;
    } else {
        return n + foo(n - 1);
    }
}
```

Nicht endrekursiv, da als letzte Operation eine Addition ausgeführt wird.

```
public int foo(int m, int n) {
    assert(m >= 0);
    assert(n >= 0);
    if (m == 0) {
        return n + 1;
    } else if (n == 0) {
        return foo(m - 1, 1);
    } else {
        return foo(m - 1, foo(m, n - 1));
    }
}
```

Nicht endrekursiv, da Endrekursive Programme keine rekursiven Aufruf in einem if-Statement haben dürfen.

Worin unterscheiden sich Implementierungen von pre- in- und postorder?

Bei allen wird **zuerst der linke Teilbaum** und dann der Rechte Teilbaum traversiert. Unterschied ist die **Reihenfolge.**

```
def preorder(node, visitor):
    if node not null:
        visitor.visit(node.element)
        preorder(node.left, visitor)
        preorder(node.right, visitor)

def postorder(node, visitor):
    if node not null:
        postorder(node.left,visitor)
        postorder(node.right,visitor)
        visitor.visit(node.element)

def inorder(node, visitor):
    if node not null:
        inorder(node.left, visitor)
        visitor.visit(node.element)
        inorder(node.right, visitor)
```

Wie hoch ist ein binärer Baum mit n Elementen mindestens und hoechstens?

Mindesthöhe: $\lceil \log_2(n+1) \rceil$ oder $\lfloor \log_2(\text{max. Anzahl Blattknoten}) \rfloor + 1$
Maximalhöhe: Anzahl Blattknoten (= entarteter Baum)

Wieviele Knoten (Pages) hat ein B-Baum der Ordnung k und Höhe h höchstens?

Max. Anzahl Elemente: $\sum n^{(\text{Level}-1)}$
Für jedes Level gibt es maximal $n^{(\text{Level}-1)}$ Elemente.

Wie wird die Höhe eines B-Baumes erhöht?

- Indem genügend Elemente hinzugefügt werden.
- Indem der Baum mit einer verringerten Ordnung neu-erstellt wird.

In beiden Fällen muss natürlich der Baum weiterhin balanciert bleiben.

Definieren Sie einen Sortieralgorithmus

Sortiert Elemente (ab -oder aufsteigend) in

- Numerische Ordnung: Basierend auf den Schlüsselwerte.
  Der Inhalt der Daten spielt dabei keine Rolle.
- Alphabetische Ordnung: Basierend auf Kanonisierung von Daten
  (= länderspezifische alphabetische Sortierung)

Wie beeinflusst bei Quicksort die Wahl des Pivotelements das Laufzeitverhalten, wenn das Array schon sortiert ist?

Wenn das Pivotelement das erste oder letzte Element des Arrays ist und das Array bereits sortiert ist → $O(n^2)$

Eine Folge von Zahlen die aufsteigend sortiert werden soll, ist bereits absteigend sortiert. Wie viele Vergleiche und Vertauschungen brauchen Bubblesort und Selection Sort?

Bubble-Sort

- Vergleiche: $n(n-1)/2$
- Vertauschungen: $n(n-1)/2$

Selection-Sort

- Vergleiche: $n(n-1)/2$
- Vertauschungen: $n/2$

Findet ein Greedy-Algorithmus immer eine optimale Lösung? Wenn nein, kennen Sie ein Beispiel?

Nein.

Wählt den Folgezustand aus, der **zum Zeitpunkt** der Wahl **den grössten Gewinn verspricht.**

Schnell, können aber in lokalen Maxima/Minima stecken bleiben.

Hashing mit linearem oder quadratischem Sondieren: Um den Eintrag in der Tabelle am Index h zu löschen, schreiben Sie map[h] = null. Warum ist das falsch und was müssen Sie stattdessen tun?

Falls an dieser Stelle bereits eine Kollision passiert ist, würde dies ein «Loch» in der Kette bedeuten und die anderen Elemente können so nicht mehr gefunden werden. Beim Löschen muss ge-rehasht werden und die kollidierten Elemente an diese Stelel gebracht werden.

Sie machen Hashing mit linearem oder quadratischen Sondieren. Sollten Sie den Füllgrad der Tabelle beim Löschen eines Elements nach unten korrigieren oder nicht? Warum?

Nein, das würde ein komplettes re-hashing aller Daten bedeuten, anstelle nur die kollidierten Daten an dieser Stelle zu re-hashen.