VERERBUNG

Normale Class
Default **Constructor** → Leerer **Constructor** automatisch generiert, wenn kein anderer definiert wurde, setzt alle Variablen zu dessen Default Werten.
Aufrufen von `public` / `protected`, **Methoden** / **Attributen** / **Constructor**, via. `super` → zeigt auf Instanz der Superklasse

```
super(); /* Constructor */    super(10); /* Constructor mit Argumenten */
super.test; /* Variable */    super.test(); /* Methode */
```

ABSTRACT CLASS

```
public abstract class Test {
  public abstract void test1();
  public void test2() {}
}
public class Test2 extends Test {
  @Override public void test1() {} /* Required */
  @Override
  public void test2() {} /* Optional */
}
```

GENERICS

Verwendung von Typen (Klassen und Schnittstellen) als Parameter, bei der Definition von Klassen, Schnittstellen und Methoden.
Damit werden zur Laufzeit Objekte in gleichem Datentypen in Datenstruktur sichergestellt.
- Type Casts sind ungeprüft bei Generics

Anwendungsfälle

```
public static <E> Stack<E> multiPush(E value, int times) { … }
Stack<String> stack1 = multiPush("Hello", 100);  Typ Inferenz
```

Generische Methode = Methode mit Typ-Parameter und Typ-Argument

```
public class MyClass<T> implements Interface<T> { … }
```

Generische Klasse = Klasse mit generischen Parametern

```
public interface MyInterface<T> { … }
```

Generisches Interface = Interface mit generischen Parametern

```
Stack<String> stack1;
```

Generische Klasse, mit konkretem Typ instanziiert, (Funktioniert für statische Variable nicht, werden über mehrere Instanziierungen geteilt, genauer Typ unklar)
Typ-Inferenz: Typ vom Compiler am Kontext erkannt, bei generischen Methoden

Gemischte Argumenttypen
Es wird der nächstmögliche gemeinsamen Basistyp gewählt:
majority(1, 3.14, 1); → T = Number oder Object | Double → T = Number / Object

Vorteile von Generics
- Typsicherheit zur Kompilierzeit: Ein Java-Compiler wendet eine strenge Typüberprüfung auf generischen Code an und gibt Fehler aus, wenn der Code die Typsicherheit verletzt. Die Behebung von Kompilierfehlern ist einfacher als die Behebung von Laufzeitfehlern, die schwer zu finden sein können
- kein explizites casting notwendig, Code Wiederverwendung, Lesbarkeit, Wartbarkeit, Früherkennung von Fehlern

Nicht möglich
- nicht möglich sind instanceof T und statische Parameter (Statische Felder benötigen zur Kompilierungszeit bekannte Typen, aber generische Typen sind erst zur Laufzeit bekannt.)
- Generic kann nicht primitiver Datentyp (int/double) sein
- Instanz erstellen nicht möglich: new T() und auch Array erstellen: new T[]
- **type erasure Einschränkung**: Es können keine Konstruktoren von generischen Typen aufgerufen werden, da deren Typ zur Laufzeit nicht bekannt ist.

TYPE BOUNDS UND TYPE ERASURE

Type Bounds
Typargumente einschränken (Graphic und alle SubTypen)
(Beschränken des Parameter-Typ auf Subtypen einer bestimmten Klasse)

```
class GraphicStack<T extends Graphic> extends Stack<T> {  public void drawAll() { … }  }
class ClassName<T extends Type1 & Interface1 & … >
```

Type Erasure
Für Rückwärtskompatibilität. Typbeschränkungen zur Komplilierzeit, Löschen der Typ-Informationen zur Laufzeit. Methode muss nach Type Erasure immer noch eindeutig sein:

```
<T> T majority(T x, T y, T z); -- Signatur mit Generics
Object majority(Object x, Object y, Object u); -- mit type erasure

public static <T extends Comparable<T>> int findFirstGreaterThan(T[] at, T el) { … } // wird als bytecode ↓ (wegen type bound)
public static int findFirstGreaterThan(Comparable[] at, Comparable el) { … }
```

GENERISCHE KLASSE (VOR UND NACH TYPE ERASURE)

```
Vorher :
class Stack<T> {
    Entry<T> top;
    void push(T value) { … }
    T pop() { … }
    static <T extends Comparable<T>> int find(T elem) { … }
}  String x = s.pop();
Raw Type (nach type erasure) :
class Stack {
    Entry top;
    void push(Object value){ … }
    Object pop() { … }
    static int find(Comparable elem) { … }
}  String x = (String)s.pop();
```

WILDCARDS

	Typ	Lesen	Schreiben	Kompatible Typ-Argumente
Invarianz	C<T>	✓	✓	T
Kovarianz	C<? extends T>	✓	X	T und Subtypen
Kontravarianz	C<? super T>	X	✓	T und Basistypen
Bivarianz	C<?>	X	X	Alle

```
public static <T> List<T> mergeToList(
  Collection<? extends T> c1, Collection<? extends T> c2 ) { … }
```

Elemente aus generischer Collection nehmen → Kovarianz
Elemente in eine Collection einfügen → Kontravariant
Beides mit einer generischen Collection machen → Invarianz

PECS (producer extends, consumers super)

```
<T> void move(
  Stack<? extends T> from, Stack<? super T> to) {
    while (!from.isEmpty()) {
        to.push(from.pop());
    }  }
```

Beispiel ohne Invarianz
Typen sind Invariant, der Typ muss exakt sein bei ArrayList<T>

```
class A {}
class B extends A {}
ArrayList<B> bList = new ArrayList<B>();
ArrayList<A> aList = bList;
```

2 Listen zusammenfügen funktioniert nicht bei Iterable
ArrayList<B> ist keine Unterklasse von ArrayList<A>, auch wenn B Unterklasse von A ist.

```
ArrayList<A> aList = new ArrayList<A>(); aList.addAll(bList); // funktioniert
```

Arrays sind Covariant, Array ist auch ein Array seiner Subtypen. Typ erlaubt auch Subtypen:

```
B[] bArray = new B[10];
A[] aArray = bArray; // Kompiliert
```

Extends bei Consumer geht nicht

```
List<? extends Number> numbers = List.of(1, 2, 3);
numbers.add(2.718); geht nicht, nur add(null) möglich
```

Compiler weiss nicht, welcher Subtyp von Number erlaubt ist, es könnte für Compiler auch eine List<Integer> sein. Deshalb kann keine Zahl hinzugefügt werden. Lösung: **super** verwenden für consumer (add).

BEISPIELE GENERICS

Bivarianz Beispiel (kein lesen und schreiben möglich)

```
public static boolean hasSameSize(List<?> list1, List<?> list2) {
    return list1.size() == list2.size();  }

public class BeispielGenerics<T> {
    public static T statischesFeld; -- kompiliert nicht, static
}

public class GenericsSyntax<T extends Iterable<T>> {
    public void processList(Iterable<? super T> processMe) {
        // … Kompiliert
}  }
```

Generics mit Comparable

```
private static <T extends  Comparable<T>>
T findLargest(Iterable<T> iterable) {
    T biggestElement = iterable.iterator().next();
    for (T item : iterable) {
        if (biggestElement.compareTo(item) < 1) {
            biggestElement = item;
    }  }
    return biggestElement;
}
```

Generisch vergleichen

```
public static <T extends Comparable<T>> int searchBinary (List<? extends T> list, T elem) {
 int compare = elem.compareTo(list.get(pivot));
 elem < pivotEl -1, elem == pivotEl 0, elem > pivotEl 1
}
```

REFLECTION & ANNOTATIONS

Metadaten über das Programm, kein Teil des Programmcodes. Kann auf private Elemente zugreifen. (`@Override`, `@Test`, `@JsonProperty`, `@FunctionalInterface`, `@SuppressWarnings(value="unchecked")`)
Anwendungen Informationen für den Compiler, Verarbeitung zur Compile -und Runtime Implementierung von Annotations, Remote Procedure Call, Serialisierung und Deserialisierung, Object Relational Mapping und Plugin-Systeme (Nachladen einer Klasse)
Marker Annotation Annotation ohne Variablen
Marker Interface Interface ohne Methoden / Variablen / Konstanten (`Serializeable`)

WICHTIGE METHODEN FÜR REFLECTION

Class-Objekt
.getClass() bei Instanz von Klasse, .class() bei Klassentyp/Klassendefinition

```
Class c = "foo".getClass(); // java.lang.String
Class c = Boolean.class; // java.lang.boolean
```

Methoden von Class-Objekt

```
public Method[] getDeclaredMethods()
 throws SecurityException
public Constructor<?>[] getDeclaredConstructors()
  throws SecurityException
public Field[] getDeclaredFields()
  throws SecurityException
```

Beispielanwendung: Testclass.class.getDeclaredFields();

Modifier (check ist Field private)

```
Field f = c.getDeclaredFields()[0];
int m = f.getModifiers();
if (Modifier.isPrivate(m)) {}
oder Lösung:
for (Field f : fields) {
 if (! java.lang.reflect.Modifier.isPrivate(f.getModifiers())) {
   System.out.println(f.getName() + "' is not private.");
   return;
} }
```

Method-Objekt, Methoden von Method

```
public String getName()
public Object invoke(Object obj, Object... args)
  throws IllegalAccessException,
  IllegalArgumentException, InvocationTargetException
public boolean isAnnotationPresent( Class<? extends Annotation>
 annotationClass)
```

VALIDATOR BEISPIEL

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Min {
    int value();
    String message() default "Field must be greater than or equal to {value}";
}
@Min(value = 18, message = "Age must be at least 18")
private int age;
```

Check in Validator<T> Class

```
public List<String> validate(T object) throws
 IllegalAccessException {
 List<String> validationErrors = new ArrayList<>();
 for (Field field: object.getClass().getDeclaredFields()) {
        field.setAccessible(true);
        validateMin(object, field, validErrors);
    }
    return validationErrors;
}
private static <T> void validateMin(T object, Field field,
 List<String> validErrors) throws IllegalAccessException {
 if (field.isAnnotationPresent(Min.class) &&
      field.getType() == int.class) {
        int value = field.getInt(object);
        Min min = field.getAnnotation(Min.class);
        if (value < min.value()) {
            validErrors.add(min.message().replace("{value}",
String.valueOf(min.value())));
} }  }
```

BEISPIEL AUTHOR CLASS

```
@Target(ElementType.TYPE) // TYPE→Klasse, METHOD, FIELD
@Retention(RetentionPolicy.RUNTIME) // COMPILE
public @interface Author {
 String name();
 String date() default "N/A";
}
```

Annotation verwenden

```
@Author(name = "Jasmin Fässler", date = "2025-06-10")
public class AnnotTest { … }
```

Auswahl Annotierter Methoden (Methoden von Method)

```
var methods = AnnotTest.class.getDeclaredMethods();
for (var m : methods) {
  if (m.isAnnotationPresent(Author.class)) { … }
}  var classNameAnnot = m.getAnnotation(Get.class);
```

BEISPIEL PROFILER

Profiler erstellen (Reflection nutzen)
1. Annotationen definieren **2.** Annotierte Methoden erkennen
3. Annotierte Methoden aufrufen und profilen

```
1. Annotation definieren
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME) // oder COMPILE
@Target(ElementType.METHOD) METHOD, TYPE→Klasse, FIELD
public @interface Profile { }

public class Profiler {
2. Annotierte Methoden finden
 public static void profileMethod(
  Object object, Method method, Object[] args
 ) {
  long startTime = System.nanoTime();
  try {
   method.invoke(object, args);
  } catch (IllegalAccessException |
    InvocationTargetException e) {
   logger.error(e.toString());
  }
  long endTime = System.nanoTime();
  long elapsedTime = endTime - startTime;
  logger.info("{} took {} nanoseconds to execute.",
   method.getName(), elapsedTime);
  }
3. Profiling der Methoden
 public static void main(String[] args) {
  ProfileFs testFs = new ProfileFs ();
  int[] array = {5, 2, 8, 1, 93, 3, 33, 1, 333};
  var methods = ProfileFs.class.getDeclaredMethods();
  for (var m : methods) {
   if(m.isAnnotationPresent(Profile.class)) {
    Profiler.profileMethod(testFs,m,new Object[]{array});
   } wenn Method die Annotation Profile hat..
//boolean isAnnotationPresent(Class<? Extends Annotation>..)
} } }
```

BEISPIEL @TEST

```
@Test
void listNodeTest() {
    var listNode2 = new LinkedListNode<>(4, null);
    var listNode = new LinkedListNode<>(3, listNode2);
    assertEquals(4, listNode2.getElement());
    assertEquals(listNode2, listNode.getNext());
}
```

BEISPIEL REFLECTION MOODLE ÜBUNG

```
public static void validateClass(Class<?> clazz) {
  Field[] fields = clazz.getDeclaredFields();
  if (fields.length > 4) {
        System.out.println("Validation failed: class > 4 fields.");
        return;
  }
  for (Field field : fields) {
    if (!java.lang.reflect.Modifier.isPrivate(field.getModifiers())) {
            System.out.println("Validation failed: Field '" +
field.getName() + "' is not private.");
            return;
    }  }
    System.out.println("Validation passed: Class " +
    clazz.getSimpleName() + " meets all conditions.");
}
```

BIG-O NOTATION | THEORETISCHE ANALYSE

Immer worst-case (ungünstigster Fall) betrachten: Einfache Analyse, Vermeidet Unsicherheit, Gut für Anwendung die garantierte Antwortzeiten benötigen, durchschnittliches Verhalten schwerer zu bestimmen, Bester Fall ist nicht interessant/trivial
Wofür geeignet
Analyse welcher Such-Algorithmus für eine Funktionalität mit aufsteigenden Zahlen am schnellsten wäre → Noch nicht implementiert und Einteilung in Komplexitätsklasse einfach, daher Big-O Notation am sinnvollsten

Regeln um Laufzeitskomplexität in Big-O-Notation zu notieren
Falls $f(n)$ Polynom vom Grad d ist, ist $f(n) \in O(n^d)$, d.h.:
1. Alle tieferen Potenzen weglassen
2. Jeweils genaueste Funktion verwenden (tiefst mögliche Potenz)
   $2^n = O(n)$ und $2^n = O(n^2)$ sind beide korrekt, aber besser ist $2^n = O(n)$
3. Konstanten weglassen
4. So stark wie möglich vereinfachen
   $3^n + 5 = O(n)$ statt auch korrekte Lösung: $3^n + 5 = O(3^n)$

Big-O-Notation mit Formel beweisen, Gleichung
f(n) (Algorithmus) ≤ c**g**(n) für n ≥ n_0
c (reelle Konstante) > 0 und n_0 (Ganzzahl-Konstante) ≥ 1
O(n) = O(g(n)) = Komplexitätsklasse = Big-O

Eigenes Beispiel: Beweisen Sie $8n^4+3n^2+4$ ist $O(n^4)$
$f(n) = O(g(n))$ falls: $f(n) \le c \cdot g(n)$ für $n \ge n_0$
$8n^4 + 3n^2 + 4 \le c \cdot n^4$
löse nach c auf $8n^4 + 3n^2 + 4 \le c \cdot n^4 \quad | \div n^4$
$\frac{8n^4+3n^2+4}{n^4} \le c = setze\ n_0 = 1\ und\ n = n_0: \frac{8+3+4}{1} \le c = 15 \le c$
n_0 = 1 und c = 15 (n_0 ist das kleinstmögliche n bei diesem c)
Die Gleichung ist ist wahr für n=1,2,3,4,...

	Name	Beschreibung
1	Konstant	Statement (Aufruf Methode `v.f()` / Setzen Wert `v = null` / Check `v != null`)
log(n)	Logarithmisch	In Hälften teilen (Binäre Suche) `i*=2`
n	Linear	Loops (Aufruf in Schleife) `i++`
nlog(n)	Linearithmische	Sortieralgorithmen (Merge Sort)
2^n	Exponential	Brute Force

Algorithm *arrayMax(A, n)*

	# Operationen	
currentMax ← *A*[0]	1 Indexierung + 1 Zuweisung:	2
for *i* ← 1 **to** *n* − 1 **do**	1 Zuweisung + n (Subtraktion + Test):	$1 + 2n$
if *A*[*i*] > *currentMax* **then**	(Indexierungen + Test) $(n-1)$	$2(n-1)$
currentMax ← *A*[*i*]	(Indexierungen + Zuweisung) $(n-1)$	$0 \mid 2(n-1)$
increment *i*	(Inkrement + Zuweisung) $(n-1)$	$2(n-1)$
return *currentMax*	1 Verlassen der Methode	1

Worst Case: $2 + (1 + 2n) + 2(n-1) + 2(n-1) + 2(n-1) + 1 = 8n - 2$
Best Case: $2 + (1 + 2n) + 2(n-1) + 0 + 2(n-1) + 1 = 6n$

ALGORITHMEN

Benötigte Eigenschaft für Algorithmus (wie Kochrezept)
Beschreibung eines endlichen, deterministischen und allgemeinen Verfahrens unter Verwendung ausführbarer, elementarer Schritte.
- **Endlich** (Termination nach N-Schritten)
- **Deterministisch** (Selbe Eingabe = Selbes Ergebnis)

Problem → Algorithmus → Lösung

ALGORITHMEN PARADIGMEN

Brute-Force Beispiel mit Passwort
26 Zeichen möglich, Passwort hat 8 Stellen, Dann gibt es 26^8 Möglichkeiten. Bei einer Stelle mehr, gibt es 26^9 Möglichkeiten

Greedy-Algorithmen
O-Notation O(n) O(1) O(n^2) | **Problem** Liefert immer eine Lösung, aber nicht unbedingt optimale Lösung | nutzen bei 2^n Problemen um sie mit O(n^2) zu lösen (Rückgeld, Ladendieb, Set covering, Radiostationen)
- Bei jedem Teilschritt die beste Lösung wählen. Versucht das globale Optimum, durch lokales optimieren zu erreichen.
- Verwendet Bewertungsfunktion zur Auswahl der Schritte.
- Einfach, einfach programmierbar, immer 1 (nicht optimale) Lösung

Dynamische Programmierung
Aufteilen in keine Subprobleme und diese nur 1 Mal je lösen und speichern. Verbesserung wenn gleiches Problem mehrmals gelöst wird.
- global optimale Lösung, kann Backtracking, komplexerer Code

Fibonacci-Folge mit dynamic programming (Tabulationsmethode) wird die Zeitkomplexität auf linear reduziert. (rekursiv: O(2^n), dynamisch: O(n))

Divide-and-Conquer (Teile und Herrsche)
Problem rekursiv in kleinere Subprobleme aufteilen. Subprobleme lösen. Lösung zusammenfügen.
Beispiel Binärsuche: kann rekursiv definiert sein | O-Notation $O(\log(n))$

EMPIRISCHE ANALYSE

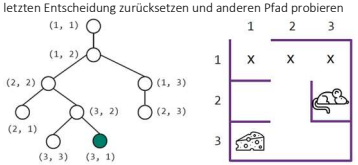
Prinzip Algorithmus wurde implementiert und die Laufzeit für unterschiedliche Eingaben gemessen. Aus diesen Messungen werden Vorhersagen für die Laufzeit für grössere oder kleinere Eingaben getroffen
Beispiel Analyse der Laufzeit einer bestehenden Cloud-Anwendung in ihrer Umgebung → Schon bestehend und schwierig in Komplexitätsklasse einzuteilen dafür eignet sich empirische Analyse besser
Herausforderungen Ergebnisse können durch Störungen in Hardware, Software und Systembelastung verfälscht werden und Algorithmus muss implementiert sein **und** Eingaben beeinflussen Ergebnis

```
StopWatch s = new StopWatch();  s.reset(); s.start(); /* Algorithmus ausführen; */  s.stop();
```

BACKTRACKING (REKURSIV)

- ☐ Anforderung genau lesen, was ist gewünscht vom Code? Aufgaben durchlesen
- ☐ Code im Anhang gut studieren
  - ☐ grobes UML Diagramm um den Code zu verstehen
- ☐ Mit Struktur für Backtracking beginnen

Konzept von Backtracking ist Trial & Error. Falls aktueller Zweig nicht zur Lösung führt, zur letzten Entscheidung zurücksetzen und anderen Pfad probieren



Vorgehen Backtracking
1. Position auf Feld (Schachbrett, Labyrinth) markieren
2. Rekursionsabbruch (Alle Felder besucht, Ausgang gefunden) `return true;`
3. Alle Operationen probieren
   a. Neues Feld / Koordinate festlegen
   b. Überprüfen, ob Feld gültig ist und noch nicht besucht wurde
      i. Wenn Ja: Überprüfen ob rekursiver Aufruf `true` zurückgibt (wird Ziel erreicht?)
      ii. Falls i. Ja: `return true;`
4. Backtracking: Markierung vom Feld entfernen und `return false;`

```
public boolean backtrack(int param1, int param2, ...) {
 1. Base case: check if solution is found, if is goal
 if (...) { return true; }
 2. Check constraints and boundaries (bounds, obstacles, marked) manchmal 2 tests
 if (nicht erlaubt) { return false; }
 3. Mark the current state as visited/part of the path (for example,
 set a cell, add to path, etc.)
 4. Iterate through all possible choices/moves at this step
 for (/* each possible choice/move */) {
 Oder separate if() statements (alle Richtungen) mit Test ob im
 erlaubten Bereich am Anfang von backtrack()
     Problem-specific: make a move or choice (if needed)
     möglicherweise vorher definierte Methoden aufrufen

     Recursion (call itself with new parameters)
     if (backtrack(param1, param2,..)) { return true; }
      // ist nur true, wenn Ziel in Rekursion erreicht wird
     Undo the move/choice (backtrack) no return statement!
 }
 5. Unmark the current state (visited or partial solution)
 (for example, reset a cell, remove from path, etc.)
 6. No solution found at this path: return false;
    // Backtracking/Rekursion ausserhalb testet nächster Weg
}
```

REKURSION

Rekursionsabbruch Werte der Parameter, für die kein rekursiver Aufruf ausgeführt wird. In jeder Rekursion muss es einen **Base Case** geben, welcher die Rekursion nicht weiterführt. Rekursive Aufrufkette muss Base Case erreichen, sonst Infinite Loop
Rekursive Aufrufe Rufen sich selbst wieder auf und bewegen sich Richtung Base Case. Kein festes Maximum für die Anzahl
Lineare Rekursion Ein rekursiver Aufruf startet höchstens einen weiteren rekursiven Aufruf
Binäre Rekursion Rekursiver Aufruf macht höchstens zwei rekursive Aufrufe
Mehrfache Rekursion Rekursiver Aufruf macht mehr als 2 weitere rekursive Aufrufe
Endrekursion Funktion, bei der rekursiver Aufruf letzter Schritt ist. Weniger Speicherbedarf auf Call Stack und kann in iterative Funktion umgewandelt werden

Ohne Endrekursion

```
private static int recsum(int x) {
 if (x == 0) { return 0; }
 else {
// Addition ist letzte Anweisung
   return x + recsum(x - 1);
} }
```

Callstack

```
recsum(3)
3 + recsum(2)
3 + (2 + recsum(1))
3 + (2 + (1 + recsum(0))
3 + (2 + (1 + 0))
3 + (2 + 1)
3 + 3
6
```

Mit Endrekursion

```
private static int tailrecsum(int x, int total) {
  if (x == 0) {
    return total;
  } else {
    // Rekursiver Aufruf ist letzte Anweisung
    return tailrecsum(x - 1, total + x);
} }
```

Iterative Version von Endrekursion

```
private static int tailsum(int x) {
  int total = 0;
  for (; x > 0; x--) {
    total += x;
  }
  return total;
}
```

Callstack

```
tailrecsum(3, 0)
tailrecsum(2, 3)
tailrecsum(1, 5)
tailrecsum(0, 6)
6
```

Rekursion (Palindrom Checker)

```
public static boolean isPalin(String text) {
 String low = text.toLowerCase();
 if (text.length() <= 1){
     return true;
 }
 if (low.charAt(0) == low.charAt(text.length() - 1)) {
    return isPalin(text.substring(1, text.length() - 1));
 }
 return false;
}
```

Rekursion (String umkehren)

```
public static String reverseString(String s){
   if(s.length() <= 1){
       return s;
   } else {
     return reverseString(s.substring(1)) + s.charAt(0);
} }
```

Rekursion (Decimal to binary)

```
public static int decimalToBinary(int decimal) {
    if (decimal == 0) {
        return 0;
    } else {
        return (decimal % 2) + (10 * (decimalToBinary(decimal / 2)));
} }
```

SORTIEREN

Bogo sort: O(∞),O(n*n!), O(?) wegen Zufall

SWAP METHODE

```
private static <T> void swap(T[] arr, int x, int y) {
    var temp = arr[x];
    arr[x] = arr[y];
    arr[y] = temp;
}
```

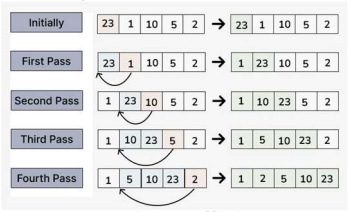
SELECTION SORT

best-case O(n^2), worst case O(n^2) O($\frac{n(n-1)}{2}$)
Von der unsortierten Liste das grösste Element in neue sortierte Liste einfügen und danach das nächst grössere Element aus unsortierten wählen, bis unsortierte Liste leer ist.
Laufzeit unabhängig von Eingabe und wenig Bewegungen im Array.

```
void selectionSort(int[] array) {
    for (int step = 0; step < array.length - 1; step++) {
        int min = step;
        for (int i = step + 1; i < array.length; i++) {
            if (array[i] < array[min]) {
            // (arr[j].compareTo(arr[index]) < 0)
                min = i;
        }   }
        swap(array, step, min); // swaps positions (vorher definiert)
} }
```

INSERTION SORT

best-case O(n), worst case O(n^2) O($\frac{n(n+1)}{2}$)
Element entnehmen und an richtiger Stelle in sortierter Liste einfügen. Gut bei teilweise sortierten Arrays.




```
void insertionSort(int array[]) {
    for (int step = 1; step < array.length; step++) {
        int key = array[step];
        int j = step - 1;
        while (j >= 0 && key < array[j]) {
            array[j + 1] = array[j];
            --j;
        }
        array[j + 1] = key;
}   }
```

BUBBLE SORT

best-case (optimierte Version) O(n), worst case O(n^2) O($\frac{n(n-1)}{2}$)
Durch Liste iterieren und in Zweier-Paaren vergleichen und Positionen vertauschen falls unsortiert. Wiederholen, bis Liste sortiert ist.

```
public static <T extends Comparable<T>> void bubbleSort(T[] arr) {
    for (int i = arr.length; i > 1; i--) {
        for (int j = 0; j < i -1; j++) {
            if (arr[j].compareTo(arr[j+1]) > 0) {
                swap(arr, j, j + 1);
 kein return, weil array ist object=call by reference
}   }   }   }
public static void optimizedBubbleSort(int[] array){
    if(array==null || array.length==0) {
        return;
    }
    boolean isSwapped;
    for (int i = 0; i < array.length - 1; i++) {
        isSwapped = false;
        for (int j = 0; j < array.length -i -1; j++) {
            if(array[j]>array[j+1]){
                int temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
                isSwapped = true;
            }
        }
        if(!isSwapped) { break; } wenn bereits sortiert
} }
```

COUNTING SORT

best-case O(n), worst case O(n+k) k=grösster Wert in der Liste
- Benötigt zusätzliche Speicherstruktur

Zählt Anzahl Vorkommnisse einzelner Elemente (Wert : Anzahl) -> (0 : 2, 1 : 0, 2 : 2, 3 : 3, 4: 0, 5 : 1), wobei der Wert zum Index wird und die Anzahl der Vorkommnisse zum Wert. Schreibt dann aus dieser Wertetabelle die Werte im ursprünglichen Array neu. Wobei er mit dem tiefsten Index beginnt und so viele Werte schreibt wie der Wert an diesem Index zeigt

```
public static void countingSort(int[] data) {
  // first find biggest value k
    int maxValue = data[0];
    for (int i = 1; i < data.length; i++) {
        if (data[i] > maxValue) {
            maxValue = data[i];
    }   }
    int[] count = new int[maxValue + 1];
    for (int i = 0; i < data.length; i++) {
        // Increment the occurrences on the given index
        count[data[i]]++;
    }
    int position = 0;
    for (int i = 0; i <= maxValue; i++) {
        for (int j = 0; j < count[i]; j++) {
            data[position] = i; position++;
} } }
```

SHELL SORT

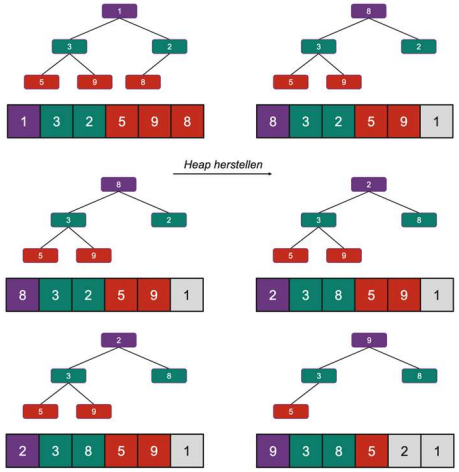
best-case O(nlog(n)), worst case: O(n^2)
Weit auseinander liegende Einträge austauschen, um teilweise sortierte Arrays zu erzeugen. Teilweise sortierte Arrays effizient mit Insertion Sort sortieren.

4-sortieren: Jedes 4te Element vergleichen und swappen falls erstes Element grösser ist als das zweite. Dann evtl. Schritt wiederholen mit jedem 2ten Element. Am Schluss Array mit Insertion Sort sortieren.

```
public static <T extends Comparable<T>> void shellSort(T[] a) {
    int n = a.length;
    // 3x+1 increment sequence:  1, 4, 13, 40...
    int h = 1;
    while (h < n/3) { h = 3*h+1; }
    while (h >= 1) {
        // h-sort the array
        for (int i = h; i < n; i++) {
            for (int j = i;
             j >= h && (a[j].compareTo(a[j-h]) < 0);
            j = j-h) {
                swap(a, j, j - h);
        }   }
        h /= 3;
}   }
```

HEAP SORT

best-case O(nlog(n)), worst case: O(nlog(n))
1. Root mit Element unten rechts tauschen
2. Heap wiederherstellen
3. Root mit Element unten rechts tauschen...




```
public static <T extends Comparable<T>> void heapSort(T[] arrayToSort,
Comparator<T> comparator) {
    var heap = new Heap<T, Void>(comparator);
    for (var i : arrayToSort) {
        heap.insert(i, null);
    }
    for (var i = 0; i < arrayToSort.length; i++) {
        arrayToSort[i] = heap.removeMin().getKey();
    }
}
```

BINÄRER SUCHBAUM (BST) SORTIERUNG

best-case O(1), worst case: O(log(n))

```
int binarySearch(int array[], int x, int low, int high) {   if (high
>= low) {
        int mid = low + (high - low) / 2;
        if (array[mid] == x) {
            return mid; }// If found at mid,return it
        if (array[mid] > x) { // Search the left half
            return binarySearch(array,x, low, mid-1);
        }
        // Search the right half
        return binarySearch(array, x, mid + 1, high);
    }
return -1;  }
```

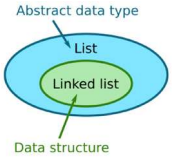
ABSTRAKTER DATENTYP (ADT)

ADT: Abstraktion konkreter Datenstruktur. Beschreibt WAS eine Datenstruktur tut, ohne auf das WIE einzugehen (**Interface**). Beschreibt Attribute (Einträge in einer Liste), Operation auf den Attributen (`pop()`, `push()`, ...), Ausnahme und Fehler (Welche Exception wird wann geworfen)
Datenstruktur: Speichert / Organisiert Daten. Konkrete Implementierung Schnittstelle (ADT)
Ziel Kapselung (Nutzung ausschliesslich über Schnittstelle) / Geheimnisprinzip (Interne Realisierung ist verborgen)
Beispiel Abstrakter Datentyp `Stack` beschreibt Reihe von Funktionen (`pop()`, `push()`). Diese können unteschiedliche implementiert werden (mit Array oder einer Liste)
`null` ist einfügbar bei Listen (ArrayList, LinkedList..), Map (values, 1 key), HashSet

PERFORMANCE / FEATURES

	Finden	Einfügen	Löschen
ArrayList	Langsam	Sehr schnell am Ende	Langsam
LinkedList	Langsam	Sehr schnell an Enden	Sehr schnell an Enden
HashSet	Sehr schnell	Sehr schnell	Sehr schnell
HashMap	Sehr schnell	Sehr schnell	Sehr schnell
TreeSet	Schnell	Schnell	Schnell
TreeMap	Schnell	Schnell	Schnell

finden = contains()



	Indexiert	Sortiert	Duplikate	null-Elem
ArrayList	Ja	Nein	Ja	Ja
LinkedList	Ja	Nein	Ja	Ja
HashSet	Nein	Nein	Nein	Ja
HashMap	Nein	Nein	Key: Nein	Ja
TreeSet	Nein	Ja	Nein	Nein
TreeMap	Nein	Ja	Key: Nein	Key: Nein

Collection	Ordering	Random Access	Key-Value	Duplicate Elements	Null Element	Thread Safety
ArrayList	Yes	Yes	No	Yes	Yes	No
LinkedList	Yes	No	No	Yes	Yes	No
HashSet	No	No	No	No	Yes	No
TreeSet	Yes	No	No	No	No	No
HashMap	No	Yes	Yes	No	Yes	No
Properties	No	Yes	Yes	No	No	Yes
Stack	Yes	No	No	Yes	Yes	Yes

WRAPPERKLASSEN

Wrapper-Klassen
- Wertetypen Referenztypen
- Keine primitiven Datenwerte auf dem Stack
  - byte, short, int, long, float, double, char
  - Dies sind keine Objekte (keine Referenzsemantik)
  - Sondern direkte Werte (Kopiersemantik)

```
Integer.parseInt("1") // String zu int
 auto-unboxing:     auto-boxing:
 int d = b;         Integer b = 5;
```

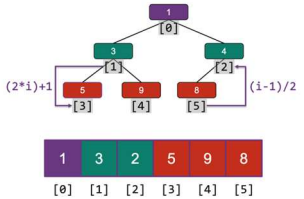
BAUM

Zweidimensionale Datenstruktur, die hierarchische Beziehungen darstellen. Besteht aus Knoten (Objekte des Baums) und Kanten (Relationen zwischen Knoten). Beispiel: Verzeichnisbaum, DOM, Familie

Heap

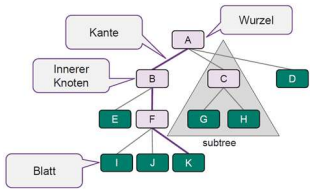
Binärer Baum, immer vollständig, Wurzel enthält immer kleinstes/grösstes Element, wiederholt Wurzelelement entnehmen, bis Array leer ist

Array zu Binären Baum → Breadth-First



Terminologie

Wurzel	Knoten ohne Elternknoten (A)
Innere Knote	Knoten mit mind. einem Kind (A, B, C, F)
Blatt	Knoten ohne Kinder (E, I, J, K, ...)
Vorgängerknoten	Eltern, Grosseltern
Geschwister	Knoten mit selben Eltern
Subtree (Unterbaum)	Baum aus einem Knoten und seinen Nachfolgern
Tiefe eines Knotens	Anzahl Vorgänger (Tiefe von Wurzel=0)
Höhe eines Baums	Maximale Tiefe der Knoten im Baum



Interface

```
public interface Tree<E> extends Iterable<E> {
    Position<E> root();
    Position<E> parent(Position<E> p);
    Iterable<Position<E>> children(Position<E> p);
    int numChildren(Position<E> p);
    boolean isInternal(Position<E> p);
    boolean isExternal(Position<E> p);
    boolean isRoot(Position<E> p);
}
```

BAUMTRAVERSIERUNGEN

Pre-Order (W-L-R)	Knoten wird **vor** seinen Kindern besucht
Post-Order (L-R-W)	Knoten wird **nach** seinen Kindern besucht
In-Order (L-W-R)	Knoten **nach** linkem Subtree und **vor** rechtem Subtree besuchen
Breadth-First	**Alle Knoten einer Stufe** besuchen, bevor Nachfolgeknoten besucht, werden

Traversierungen Code

```
public void preOrder(v) {
    visit(v);
    foreach child w of v {
        preOrder(w);
} }
```

```
public void postOrder(v) {
    foreach child w of v {
        postOrder(w);
    }
    visit(v);
}
```

```
public void inorder(v){
    if hasLeft(v)
        inOrder(left(v));
    }
    visit(v)
    if hasRigth(v){
        inOrder(right(v));
    }
}
```

```
public void breadthFirst()
{
 while Q not empty {
  v = Q.dequeue();
  visit(v);
  foreach child w in
   children(v) {
       Q.enqueue(w);
} } }
```

BINÄRER BAUM

Grundprinzip Spezialform Baum. Jeder Knoten höchstens 2 Kinder
Binärer Suchbaum Kinder sind geordnetes Paar (links kleiner, rechts grösser)
Interface

```
public interface BinaryTree<E> extends Tree<E> {
    Node<E> left(Node<E> p);
    Node<E> right(Node<E> p);
    Node<E> sibling(Node<E> p);
    Node<E> addRoot(E e);
    Node<E> addLeft(Node<E> p, E e);
    Node<E> addRight(Node<E> p, E e);
}
```

BAUM FUNKTIONEN

Tiefe

```
public int depth (Position <E> p) {
 if (isRoot(p)) { return 0; } else { return 1 + depth(parent(p)); }
}
```

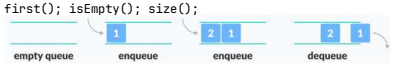
Höhe

```
public int height (Position <E> p) {
    int h = 0;
    for (Position<E> c : children(p)) {
        h = Math.max(h, 1 + height(c));
    }
    return h;
}
```

QUEUE (FIFO)

Queue (Warteschlange), Deque : double ended queue

```
Deque<String> queue = new LinkedList<>();
queue.addLast(elem);       |  queue.addFirst(elem);
queue.removeFirst(elem);   |  queue.removeLast(elem);
first(); isEmpty(); size();
```



Anwendungen Queue: Website-Verkehr, Router/Switches
Zeitkomplexität Enqueue und Dequeue sind beide $O(1)$

PRIORITY QUEUE

Grundprinzip Einfügen von Werten mit Priority k. Liste ist nach k sortiert. Eine Priority Queue kann entweder sortiert oder unsortiert sein

```
insert(K k, V v); removeMin(); min();
```

Anwendungen Dijkstra-Algorithmus, Datenkompression Huffman Code

Methode	Unsorted List	Sorted List
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

ADAPTABLE PRIORITY QUEUE ADT

Gleiches Interface wie Priority Queue aber mehr Methoden

```
remove(e); replaceKey(e, k); replaceValue(e, v);
```

LISTS

ARRAYLIST

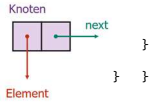
```
ArrayList<String> sList = new ArrayList<>();
List.of(array); Create list out of array
sList.add("00"); Add Element
sList.add(0, "Bsys1"); Add at position 0
String x = sList.get(1); Get at position 1
sList.set(0, "Bsys2"); Replace at position 0
Check if list contains element:
boolean b = sList.contains("CN1");
sList.remove("ICTh"); Remove element
sList.remove(1); Remove at position 1
list.sort(..); z.B. .sort(Comparable::compareTo)
list.addAll(list);   list.size();
```

LINKEDLIST

worst-case: suchen und löschen O(n), einfügen O(1)
- Verkettete Liste der Elemente
- Dynamisch hinzufügbar und entfernbar
- LIFO (stack) und FIFO (queue) möglich
- Kein Umkopieren bei `add()`, `remove()`
- Intern Doppelt verkettet (vorwärts/rückwärts)
  `List<String>firstList = new ArrayList<>();`

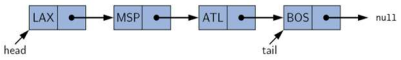
```
List<String> firstList = new LinkedList<>();
add(), remove(anfang/ende) Sehr schnell (O(1))
get(), set(), contains(), remove() Langsam (O(n))

//Gleiche Funktionen wie ArrayList
public class Node<E> {
    private E element;
    private Node<E> next;
public Node(E e, Node<E> n) {
element = e;
next = n;
    public E getElement() {
return element;
```




```
}
}
}
```

Linked List Sequenz von Knoten
Knoten Element + Zeiger zum nächsten Knoten
Anwendungen Hash Tabellen, Graphen



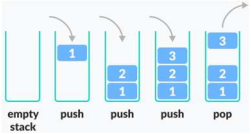
Doppelt-verkettete-Liste Jeder Knoten speichert Verbindung zum Vorgänger und Nachfolger

STACK (LIFO)

Zeitkomplexität: Pop und Push Operationen sind $O(1)$
Kann mithilfe von z.B. Array oder Liste implementiert werden

```
var stack = new Stack<E>();
Entfernt oberstes Element vom Stack:
var popEl = stack.pop();
Fügt Element auf den Stack hinzu:
var pushedItem = stack.push(el);
var isEmpty = stack.empty(); size();
```

Anwendungen Stacks Methodenaufrufe in JVM, Undo, History im Browser, Hilfsdatenstruktur für weitere Algorithmen (Array umkehren)



SETS

Sammlung von Objekten eines Typs. Keine Duplikate, keine Reihenfolge, kein Index-zugriff. Normale for-loop nicht möglich (for-each, iterator)

HashSet

- In Hashtabelle gespeichert
- Elemente geben hashCode() konsistent zu equals()

```
Set<String> otherSet = new HashSet<>();
add(elem); Add one element
remove(elem); Remove the given element
contains(elem); boolean
size();  size of all contained elements
isEmpty(); boolean

String[] a = (String[]) set.toArray();
```

TreeSet

- Sortiert, In Binärbäumen gespeichert
- Elemente implementieren Comparable und equals()

```
Set<String> firstSet = new TreeSet<>();
// Gleiche Funktionen wie HashSet
```

Implementierung:

```
@Override
public boolean add(E e) {
  if (list.contains(e)) {
    return false;
  } else {
    list.add(e);
    return true;
} }
```

```
@Override
public boolean
contains(Object o) {
    return
list.contains(o);
}
```

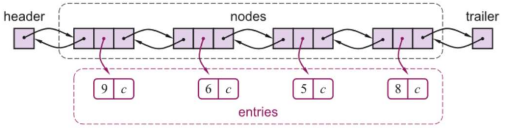
MULTI SET

Grundprinzip Set mit erlaubten Duplikaten, Duplikat equals() oder == überlegen

```
count(elem);  remove(E e, int n);  remove(e) entfernt 1 Element davon
public int add(E element, int occurrences) {
    if (occurrences < 0) {
      throw new IllegalArgumentException("Occurrences must be
      positive.");
    }
    int index = getIndex(element);
    if (index == -1) {
        var newEntry = new MultisetEntry<>(element);
        newEntry.setCount(occurrences);
        elements.add(newEntry);
        return 0;
    } else {
        int currentCount = elements.get(index).getCount();
        int newCount = currentCount + occurrences;
        elements.get(index).setCount(newCount);
        return newCount;
} }
```

MAP

Grundprinzip: Sammlung von eindeutigen Schlüssel-Wert Paaren
Ähnlich wie Set, Mengen von Schlüssel-Wert-Paaren, Schlüssel müssen gleiche Regeln wie Sets erfüllen (Keine Duplikate)
Zeitkomplexität schlechtester Fall von put,get,remove ist O(n)
Implementierung mit unsortierter Liste:



HashMap

- Braucht hashCode() und equals() Methode für sinvolle Schlüssel.

```
Map<Integer, String> map = new HashMap<>();
// Gleiche Funktionen wie TreeMap
```

TreeMap

- Nach Schlüssel sortiert

```
Map<Integer, String> map1 = new TreeMap<>();
put(2000, "Hello"); Add one element
containsKey(2000); // boolean, key in map
containsValue("Hello"); // boolean, value in map
get(2000); // get value of key
size(), remove(key), isEmpty(), clear()
```

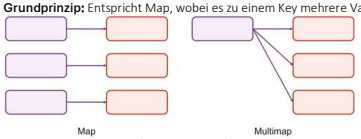
Werte ausgeben von Map

```
for (int number : map1.keySet()) { // all keys
    System.out.println(number);
}
for (String value : map1.values()) { // all values
    System.out.println(value);
}
for (var el : map.entrySet()) { // all key values
    var k = el.getKey();
    var v = el.getValue();
}
for (Map.Entry<Integer, String> number : map.entrySet()) {
    System.out.println(number.getKey());
}
map.values().stream().mapToInt(i -> i).sum();
```

SeparateChainingMap
Map mit LinkedList als Value. (z.b. mit Entry implementiert).

MULTI MAP

Grundprinzip: Entspricht Map, wobei es zu einem Key mehrere Values geben kann.




```
Iterable<V> get(K k); void put(K k, V v); boolean remove(K k, V v);
Iterable<V> removeAll(K k);
```

HASHING

Hash-Maps Menge von Schlüssel-Wert Paaren von denen jeder Schlüssel unique ist. | **Hashing** Hash-Funktion h bildet Schlüssel auf Indexwerte im Intervall [0, n-1] ab und bestimmt für Elemente e die Position h(e) im Feld. Der Hash Algorithmus gibt einen Integer zurück, welcher auf einen Slot der Hash-Tabelle zeigt. | **Ziele** Mit konstantem Aufwand (möglichst schnell) O(n) in einer Collection (Set, Map) finden.

Eigenschaften guter Hashfunktion
- Konsistenz (Trotz wiederholtem Aufruf gleichen Hashcode)
- Effiziente Berechnung, Gleichmässige Verteilung der Schlüssel → Geringe Anomalien

Mögliche Methoden zum Hashen
Divisionsrestverfahren $h(x) = x \bmod 10$
Integer Cast Schlüssel als Integer interpretieren
```
byte[] b = key.getBytes(StandardCharsets.UTF_16);
ByteBuffer wrapped = ByteBuffer.wrap(b);
return wrapped.getInt();
```
Komponentensumme Schlüssel in Komponenten fester Länge unterteilen, Komponenten summieren, Overflow ignorieren
```
int hash = 0;
for (int i = 0; i < s.length(); i++) {
    hash += s.charAt(i); }
```
Polynom-Akkumulation Komponenten bei Komponentensumme unterschiedlich gewichten (Gut für Strings)

$$p(z) = a_0 + a_1 \cdot z + a_2 \cdot z^2 + \cdots + a_{n-1} \cdot z^{n-1}$$

```
int hash = 0;
for (int i = 0; i < s.length(); i++) {
    hash += s.charAt(i) * Math.pow(31,(s.length() - i + 1));  }
return hash;
```
Beispiel Kompressionsfunktion $h2(y) = y \bmod N$ (Grösse der Hash-Tabelle N ist oft Primzahl)
Kollision Mehrere Schlüssel zeigen auf einen Bucket
Überlauf/Overflow Datensatz mit Schüssel heisst Überläufer, wenn durch Hashfunktion zugewiesener Behälter schon voll ist
Doppeltes Hashing Zweite Hashfunktion $d(x)$, falls Kollision entstanden ist $h(x) + j \cdot d(x) \bmod N$, j = Anzahl Kollisionen, N = Tabellengrösse (Primzahl)
Konsistentes Hashing gibt es auch, Gegenteil von dynamischen Hashing.

Lastfaktor

Einträge pro Bucket im Schnitt (Lastfaktor): $\alpha = \frac{n}{N}$ (n: Einträge in der Tabelle, N: Buckets in der Tabelle)

Lineare Suche im Kollisionsbereich: $\frac{n}{N}$

Optimierung: Einträge im Kollisionsbereich ordnen (z.B. nach Zugriffhäufigkeit)

GESCHLOSSENE ADDRESSIERUNG

Behälter/Buckets sind (verkettete) Listen, Platz nicht begrenzt, prinzipiell keine Überläufer.
Die Kollisionseinträge werden in einem Überlaufbereich pro Bucket gespeichert.
Der Lastfaktor kann hier grösser werden als 1.

Separate Chaining
- Resize/Rehash einfacher, da jedes Element im passenden Bucket gespeichert wird
- Geringer Speicheroverhead (Speicherung in verketteten Listen)
- Akzeptable Performance auch bei hohem Lastfaktor

Separate Chaining: Jede Zelle der Hashtabelle zeigt auf Liste, Einfache Umsetzung dafür zusätzliche Datenstruktur und Speicherbedarf
Listen-basierte Map für Kollisionsbereich: get(k) = return A[h(k)].get(k)

OFFENE ADDRESSIERUNG

Überläufer in nächste verfügbare Zelle einfügen, durchsuchen bis leere Zelle gefunden wurde.
Bei Kollisionen wird Platz in benachbarten Buckets gesucht.
Der Lastfaktor kann nicht grösser werden als 1
Sondierungsfolge bestimmt Weg zum Speichern und Wiederauffinden der Überläufer.
Gelöschte Werte dürfen nicht gelöscht werden, sondern nur als gelöscht markiert, um die Sondierungskette nicht zu unterbrechen.
Problem: lange Sondierungsketten

Lineare Sondierung / Linear Probing
- Resize/Rehash aufwändiger, da Sondierreihenfolgen betrachtet werden müssen
- Kein Speicheroverhead (Speicherung erfolgt in-Place)
- Performance verschlechtert sich stark mit wachsendem Lastfaktor

Quadratisches sondieren
um «Primary Clustering» zu lösen. i = sondierungsschritt $s(k, i) = h(k) + ci$ (c = 2)

- Wahrscheinlichkeit einer Kollision abhängig von $\alpha = \frac{n}{N}$
- Wahrscheinlichkeit, dass eine Position belegt ist: α
- Wahrscheinlichkeit, dass eine Position frei ist: $1 - \alpha$

Multiplikation: Wahrscheinlichkeit, dass **beide** Ereignisse eintreten

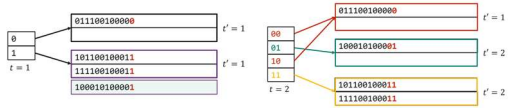
1. Wahrscheinlichkeit, dass die erste Position belegt ist: α
2. Wahrscheinlichkeit, dass die erste Position belegt (α) und die zweite Position frei ($1 - \alpha$) ist: $\alpha * (1 - \alpha)$
3. Wahrscheinlichkeit, dass die ersten beiden Positionen belegt ($\alpha * \alpha$) sind und die dritte Position frei ist: $\alpha^2 * (1 - \alpha)$
- Wahrscheinlichkeit für das Finden einer freien Position ($p = Anzahl\ Sondierungsschritte$): $\alpha^{p-1} * (1 - \alpha)$

DYNAMISCHES HASHING : ERWEITERBARES HASHING

extendible hashing, das mit Binärzahlen
- Resize/Rehash einfacher, da nicht die gesamte Tabelle verschoben werden muss
- Grösserer Speicheroverhead (Bucketstruktur und Directory)
- Gute Performance bei hohem Lastfaktor

Vereinfacht vergrössern der Hashtabelle ohne Reallokation aller Werte. Verwendet erste Bit, binäres Ergebnis Hashfunktion $h(x)$. Überlauf → + 1 Bit
Globale Tiefe und Tiefe von Behälter relevant.

Vorher **Nachher**



Ablauf: Bucket berechnen
- wenn es Platz hat, das Mapping in Bucket einfügen
- wenn `bucket.getLocalDepth() == this.globalDepth` → Directory vergrössern
- wenn `bucket.getLocalDepth() < this.globalDepth` → Bucket aufteilen

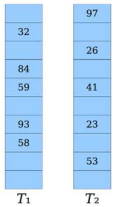
```
private int getIndex(K key){
    var hash = key.hashCode();
    var binaryString = Integer.toBinaryString(hash);
    var index = binaryString.length() - globalDepth;
    var relevantPart = binaryString.substring(Math.max(index, 0));
    return Integer.parseInt(relevantPart,2);
}
```

CUCKOO HASHING

Zwei gleichgrosse Tabellen erstellen. Hashfunktionen: $h1$ und $h2$, die auf die Tabelle abbilden.
Jedes Element befindet sich an Position $h_1(x)$ → T_1 (z.B. x mod 5), $h_2(x)$ → T_2 (z.B. (x div 5) mod 5)
Einfügen von x $T1$ prüfen
- $h1(x)$ leer: Platziere x dort
- $h1(x)$ nicht leer: Verdränge bestehendes Element y und versuche, y in $T2$ einzufügen.
- Wiederhole diesen Vorgang im Wechsel zwischen den beiden Tabellen.

Vorteile
- Konstante Zugriffszeit im erfolgreichen Fall: Nur zwei mögliche Positionen pro Element: O(1). - Keine langen Sondierketten, da maximal 2 Positionen pro Schlüssel geprüft werden müssen.

Nachteile
- Einfügen ist teuer
- Möglicherweise Rehashing nötig, falls ein Zyklus auftritt oder maximale Verdrängungsschritte überschritten werden. (MAX_CUCKOO)




```
public static int MAX_CUCKOO = 10;
private void cuckooHash(K key, V value) {
    K curKey = key; V curValue = value;
    for (int attempt = 0; attempt < MAX_CUCKOO; attempt++) {
     // Hier kommt die Logik, return
    }
    throw new IllegalStateException("Maximum cuckoo attempts reached
(" + MAX_CUCKOO + ")"); }
```

HASHCODE

Zwei "gleiche" Objekte → gleicher HashCode (best practice)
Ungleiche Objekte können aber gleichen HashCode haben
Hashcode wird zusammen mit equals definiert, konsistente Werte wie bei equals, sonst ergeben sich beim Einsatz von Maps Fehler.

```
@Override
public int hashCode() {
    return Objects.hash(firstName, lastName);
    return Objects.hashcode(firstName);
}
```

DESIGN PATTERNS

Motivation: Wiederverwendbare Lösungen für wiederkehrende Probleme verwenden. Vorlagen die in konkreten Problemen angewandt werden können. Lösung ist abstrakt und in neuen Kontexten anwendbar.
Erzeugungsmuster: Abstrahieren Instanziierung (Factory, Singleton, etc.)
Strukturmuster: Zusammensetzung von Klassen und Objekten zu grösseren Strukturen (Adapter, Fassade, etc.)
Verhaltensmuster: Algorithmen und Verteilung von Verantwortung zwischen Objekten (Iterator, Visitor, etc.)

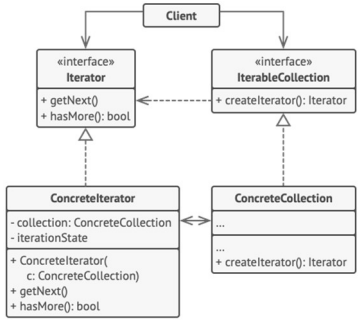
ITERATOR (VERHALTENSMUSTER)

Verhaltensmuster: Elemente einer Collection nach einem gewissen Pattern iterieren, ohne die zugrunde liegende Darstellung (Liste, Stapel, Baum usw.) offenzulegen

```
public interface Iterator<E> {
    boolean hasNext();
    E next() throws NoSuchElementException;
    void remove() throws UnsupportedOperationException;
}
```
Lazy Iterator: Iteration auf originaler Datenstruktur. Niedrigere Speicherkosten $O(1)$. Jedoch bei Änderungen auf Datenstruktur können korrekte Iteration verunmöglichen. Fehl-Verhalten bei unerwarteten Modifikationen der Ausgangs-Datenstruktur

Snapshot Iterator: Originale Datenstruktur nicht verändern. Bei Erzeugung wird eine Kopie der Ausgangs-Datenstrukturen erzeugt. Änderungen auf Ausgangs-Datenstruktur beeinflussen Iteration nicht. Kosten: $O(n)$ Speicher und Laufzeit

```
public SnapShotArrayListIterator(T[] elements, int size) {
    this.elements = Arrays.copyOf(elements, size);
}
```



Verwendung `return new ConcreteIterator(this);`

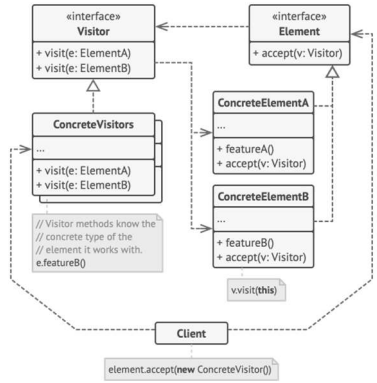
ArrayListIterator
```
public class ArrayListIterator<T> implements Iterator<T> {
    private int currentIndex;
    private final ArrayList<T> arraylist;
    public ArrayListIterator(ArrayList<T> arraylist) {
        this.currentIndex = 0;
        this.arraylist = arraylist;
    }
    @Override
    public boolean hasNext() {
        return currentIndex < arraylist.size();
    }
    @Override
    public T next() {
        return arraylist.get(currentIndex++);
} }
```

Beispiel BinaryTreeIterator (Preorder)
```
public class BinaryTreeIteratorPreorder<T> implements Iterator<T> {
    private final Stack<Node<T>> visiting;
    public BinaryTreeIteratorPreorder (Node<T> root) {
        this.visiting = new Stack<>();
        visiting.push(root);
    }
    @Override
    public boolean hasNext() {
        return !visiting.empty();
    }
    @Override
    public T next() {
        Node<T> node = visiting.pop();
        if (node.getRight() != null) {
            visiting.push(node.getRight();
        }
        if (node.getLeft() != null) {
            visiting.push(node.getLeft();
        }
        return node.getValue();
} }
```

VISITOR (VERHALTENSMUSTER)

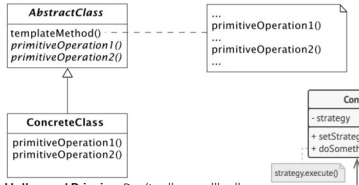
Verhaltensmuster: Trennung von Algorithmen und Datenstrukturen auf denen sie operieren. Ziel ist Algorithmen nicht über Datenstrukturen zu verteilen
Anwendungen: Visitor Pattern beschreibt Operationen auf Elementen einer anderen Objektstruktur. Neue Operationen definieren, ohne Klassen auf denen operiert wird zu ändern. Klassen nicht mit zusätzlichen Operationen verkomplizieren. Zusammenhängende Operationen in einer Klasse zusammenfassen
Beispiel
```
public interface IVisitor {
    void visit(Method t);
    void leave (Method t);
}
public class Visitor implements IVisitor {
    @Override
    public void visit(Method t) {
        System.out.println("Calling method:" + t);
} }
```

```
public interface IMethod {
    void accept(IVisitor visitor);
}
public class Method implements IMethod {
    private final String m_name;
    public Method(String name) {
        m_name = name;
    }
    @Override
    public String toString() {
        return m_name;
    }
    @Override
    public void accept(IVisitor visitor) {
        visitor.visit(this);
        // Execute needed actions of this class here
        visitor.leave(this);
} }
```

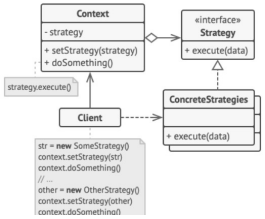


TEMPLATE METHOD (VERHALTENSMUSTER)

Backbone eines Algorithmus definieren, Teilschritte später in Subklassen spezifizieren. Lässt Subklassen Teile des Algorithmus verfeinern, ohne Struktur des Algorithmus zu verändern. Gemeinsame, unveränderliche Teile werden in abstrakter Klasse implementiert (Template). Variablen Schritte werden in Methode ausgelagert
Pattern
- Struktur in `templateMethod()` definieren, enthält gemeinsame, unveränderlichen Teile.
- Variable Schritte in Methode (`primitiveOperation`) auslagern
  - Variable Schritte repräsentieren Hooks, die von Subklassen implementiert werden.
  - Methode kann Default-Implementierung haben.
- Subklassen (`ConcreteClass`) passen Algorithmus durch Überschreiben an



Hollywood Prinzip: „Don't call us, we'll call you"
Anwendungen: Frameworks: Framework implementiert unabhängigen, allgemeinen Teil, Data Mining, Operationen auf Datenstrukturen, Traversierung z.B. Euler Tour
Problem von Template: Wenn Skeleton Code für Problem nicht ausreicht und deshalb angepasst werden muss, dann ist das sehr komplex. Deshalb Wartbarkeit sehr schwierig.



Alternative Strategy Pattern
Macht das Gleiche, einfach über ein Interface.
Geringere Abhängigkeit von Interface und Concrete Strategy. Jedoch weniger Code wiederverwendbarkeit.

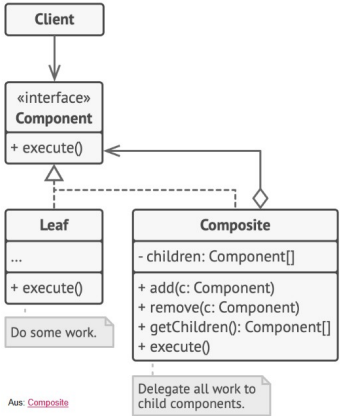
Übung Kochrezept
```
public abstract class Rezept {
  public final void bereiteZu() { // … }
  // abstrakte Methoden
}

public class SpaghettiBolognese extends Rezept {
  // überschreiben der abstrakten Methode von Rezept
}
```

COMPOSITE (STRUKTURMUSTER)

Rekursiv Element suchen, z.B. Schlüssel in Box, und Box kann weitere Boxen enthalten.
Component Interface beschreibt Operationen, die sowohl für einfache als auch für komplexe Elemente des Baums gemeinsam sind.
Leaf ist ein grundlegendes Element eines Baums, das keine Unterelemente besitzt.
Composite ist ein Element, das Unterelemente besitzt: entweder Blätter oder andere Container. Ein Container kennt die konkreten Klassen seiner Kindelemente nicht. Er arbeitet mit allen Unterelementen ausschließlich über die Komponenten-Schnittstelle.
Client arbeitet mit allen Elementen über das Component
Interface. Kann dadurch mit Leaf oder Composite Elementen auf die gleiche Weise arbeiten.

```
// Component interface
interface Component {  void printInfo();  }
// Leaf - Item like Key, Phone, etc.
class Item implements Component {
    private String name;
    public Item(String name) {
        this.name = name;
    }
    @Override
    public void printInfo() {
        System.out.println("Item: " + name);
}   }
// Composite - Box can hold Items or other Boxes
class Box implements Component {
    private List<Component> contents = new ArrayList<>();
    public void add(Component component) {
        contents.add(component);
    }
    public void remove(Component component) {
        contents.remove(component);
    }
    @Override
    public void printInfo() {
        for (Component c : contents) {
            c.printInfo(); // Delegate to children
}   }   }
public void main(String[] args) {
  Item key = new Item("Key");
  Item phone = new Item("Phone");
  Item charger = new Item("Charger");
  Box smallBox = new Box("Small Box");
  smallBox.add(key);
  smallBox.add(phone);
  Box bigBox = new
Box("Big Box");
  bigBox.add(smallBox);
  bigBox.add(charger);
}
```



Aus: Composite

STREAM API IMPORT JAVA.UTIL.STREAM.*;

Zwischenoperationen (verketten möglich)
filter(Predicate) Beispiel: people.stream().filter(p -> p.getAge() >= 18)
map(Function) Projizieren gemäss Function
mapToInt(Function) Proji. auf primitiver Typ (mapToDouble, mapToLong)
sorted() Sortieren mit oder ohne Comparator (z.B. Person::getAge)
distinct() Duplikate werden gelöscht gemäss equals
limit(long n) Erste n Elemente liefern
skip(long n) Erste n Elemente ignorieren
flatMap(Function) map aber Stream von Streams werden «flach»

Terminaloperationen (beenden die Kette)
forEach(Consumer) Beispiel: .forEach(s -> System.out.println(s));
forEachOrdered(Consumer) Erhält die Reihenfolge der Elemente
count() Anzahl Elemente (**long**)
min(), max() Mit Comparator Argument , liefert Optional-Objekte
average() Nur bei int, long, double und liefert Optional-Objekte
sum() Nur bei int, long, double
findAny() Gibt irgendein Element zurück
findFirst() gibt erstes Element zurück
allMatch(Predicate), anyMatch(..), noneMatch(..) boolean | Stream.concat(stream1, stream2)

Weiteres
isPresent() für Optional-Obj., boolean, ob Element vorhanden
isEmpty(): true wenn **kein** Element vorhanden ist
get(): Gibt Element, Exception wenn nicht vorhanden

```
Optional<String> result = people.stream()
  .map(p -> p.getLastName())
  .reduce((name1, name2) -> name1 + name1);
```

COMPARATOR UND COMPARABLE

Gibt zurück, -1, 1, 0
< 0 : this kleiner als other
> 0 : this grösser als other
0 : this gleich other

```
Comparator: arr.sort(comparator); oder
            Collections.sort(arr, comparator);
Comparable (definiert in Objekt): Collections.sort(arr);
```

2 Elemente vergleichen (siehe Methodenreferenz)

```
Person::compareTo ⇔ (p1, p2) -> p1.compareTo(p2)
 Comparator.comparing(Person::getAge));
comparator.compare(v1, v2); mit Comparator comparator
Collections.sort(people, new AgeComparator());
ist gleich wie: people.sort(new AgeComparator());
```

NÜTZLICHE CODES

Main Methode

```
public static void main(String[] args) {…}
x = a ? b : c // ternary operator für Einfaches
```

Offene Parameterliste, Erlaubt beliebige Anzahl Argumente

```
static int sum(int... numbers) {
      // numbers als Array benutzen .length [i]
} // varargs, s = sum(1, 2, 3);
Objects.isNull(myObject); Null check
```

ARRAY

Datenstruktur, um gleichartige Elemente in einer Reihenfolge zu speichern. Inhalt liegt in zusammenhängendem Bereich im Speicher
Gut: Schneller wahlfreier Zugriff, **Schlecht**: Feste Grösse, Einfügen teuer
Vorgegebene Länge (capacity), Kann nur Werte des gegebenen Types enthalten | Kann Basis Datentypen enthalten (**int**, **char**)

```
// Array erstellen
var array = new int[]{1, 2, 3};
int array[] = {1, 2, 3};
var array = new int[3]; // Empty
int length = myArray.length; // ohne Klammern
myArray[0] = 3; // kein .set bei Array
```

Vergleichen mittels **Array.equals(a, b)**

```
Arrays.deepEquals(a, b) Geschachtelte Arrays
for (i = 0; i < a.length; i++){} / enhanced for
```

Liste zu Array

```
Person[] p = peopleStream.toArray(Person[]:: new);
```

Liste mit Integer zu Array

```
list.stream().mapToInt(Integer::intValue).toArray();
```

LISTS, COLLECTIONS

Combine lists of lists to one

```
List<String> collect = list.stream().flatMap(Collection::stream)
    .collect(Collectors.toList());
```

Stream Rückumwandlung (Stream to Collections)

```
List<Person> list = people.toList();
HashSet<Person> set = people.toCollection(HashSet::new)
```

COLLECTIONS MERGE

```
containsAll, addAll, removeAll, retainAll, clear
public static <T> List<T> mergeToList(Collection<? extends T>
coll1, Collection<? Extends T> coll2) {
    var result = new ArrayList<T>();
    merge(coll1, coll2, result);
    return result;
}
public static <T> void merge(Collection<? extends T>
inputColl1, Collection<? extends T> inputColl2, Collection<?
super T> targetColl) {
    targetColl.addAll(inputColl1);
    targetColl.addAll(inputColl2);
}
sum += num.doubleValue(); Number zu Double mit .doubleValue()

Collections merge mit Bivarianz :
public static <T> List<T> filterByType(Collection<?> source,
Class<T> clazz) {
    List<T> result = new ArrayList<>();
    for (Object obj : source) {
        if (clazz.isInstance(obj)) {
            result.add(clazz.cast(obj));
    }   }
    return result;
}
```

STRING, TOSTRING UND STRINGBUILDER

```
    @Override
    public String toString() {
        var sBuilder = new StringBuilder();
        for (E elem : list) {
            sBuilder.append(elem);
            sBuilder.append(System.lineSeparator());
        }
        return sBuilder.toString();
} }
String[] lines = fileContent.toString().split("\n");
.toLowerCase() .length() .charAt(int index)
. substring(int beginIndex, int endIndex) // substring begins
at beginIndex and goes to the character at (endIndex - 1).
String listToString = String.join(", ", listOfStrings);
```

SWITCH CASE

```
int x = 5;
switch(x) {
    case 2:
        System.out.println("2");
        break;
    case 5:
        System.out.println("5");
        break;
    default:
        System.out.println("no match!");
}       Ohne Break → Fallthrough(Nächstes Case wird ausgeführt)
String howMany = switch(x) { // seit Java 14
    case 2 -> ("2");
    case 5 -> ("5");
    default -> ("many");
};
```

LOOPS

```
for (int i = 0; i < list.size(); i++) {…}
enhanced for loop:
for (var el : list) {…}
```

ausgeschrieben so

```
for (Iterator<Typ> it = stringList.iterator(); it.hasNext(); ) {
    String s = it.next();    System.out.println(s);
}
```

oder so:

```
Iterator<Typ> it = stringList.iterator();
while(it.hasNext()) {
    Typ elem = it.next();
    if (!elem.equals("test")) {
        it.remove(); hinzufügen/löschen möglich
    }
}  Keine ConcurrentModificationException mit Iterator
continue; nächster Schleifendurchlauf    break; Loop abbrechen
```

IMPLIZITER CODE = ROT

```
public class Vehicle extends Object {
    private int speed;
    public Vehicle() {
       super();
       speed = 0;
} }
public class Car extends Vehicle {
    private int doors;
    public Car() {
       super();
} }
```

ENUMERATIONS (ENUM)

Fast gleich zu Klasse enthält aber spezifische Werte die Klasse darstellen
Enum mit Konstruktor Beispiel

```
public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY;
    public boolean isWeekend() {
        return this == SATURDAY || this == SUNDAY;
}  }
public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY;
    public boolean isWeekend() {
        return this == SATURDAY || this == SUNDAY;
} }
public enum Weekday {
    MONDAY(true), TUESDAY(true), WEDNESDAY(true),
    THURSDAY(true), FRIDAY(true),
    SATURDAY(false), SUNDAY(false);
    private boolean workDay;

    Weekday(boolean workDay) {// nur privater Konstruktor
        this.workDay = workDay;
    }
    public boolean isWorkDay() {
        return workDay;
} }
Weekday currentDay; // Deklaration
currentDay = Weekday.WEDNESDAY;
if (currentDay == Weekday.MONDAY) { ..}
currentDay = null;
if (currentDay == null) { … }
void getActivity(Weekday day) {
    switch (day) {
        case MONDAY:
            return "consulting";
        default:
            return "weekend";
}   }
Weekday currentDay = …
if (currentDay.isWeekend()) { … }
```

Ein Enum ist ein eigener Datentyp mit endlichem Wertebereich. Parameter Datentyp (hier boolean) kann ersetzt weggelassen werden.

ENUM SW4

```
enum BillCoin {
    ONE_HUNDRED(100, Kind.BILL),
    TWENTY(20, Kind.BILL),
    FIVE(5, Kind.COIN),
    ONE(1, Kind.COIN);
    enum Kind {
        BILL, COIN;
        public String toString() {
            return name().toLowerCase();
        }
    }
    private final int amount;
    private final Kind kind;
    BillCoin(int amount, Kind kind) {
        this.amount = amount;
        this.kind = kind;
    }
    public int getAmount() { return amount;  }
    public Kind getKind() {  return kind;    }
    @Override
    public String toString() {
        return amount + " CHF " + kind;
    }
}
for (BillCoin money : BillCoin.values()) { // do something }
```

CODE

RECORD

Record generiert automatisch Felder (erkannt von den Parametern). Allerdings keine Getter/Setter. Record ist immutable = unveränderbar.
Generiert wird toString(), equals(), hashCode()
Statt Getter .name(), .score()

```
public record RecordClass(String name, int score) {
    @Override
    public String toString() {
        return "(" + name + ", " + score + ")"; }
}
```

RINGBUFFER

```
class RingBuffer implements IRingBuffer {
    private int[] buffer;  private int size;
    private int start;  private int end;
    private boolean isFull;

    public RingBuffer(int size) {
        if (size <= 0) {
            throw new IllegalArgumentException("Size must be >
0");
        }
        this.size = size;
        this.buffer = new int[size];
        this.start = 0;   this.end = 0;  this.isFull = false;
    }
    public void append(int item) {
        buffer[end] = item;
        end = (end + 1) % size;
        if (isFull) {
            start = (start + 1) % size;
        }
        if (end == start) { isFull = true; }
    }
    public List<Integer> getData() {
        List<Integer> items = new ArrayList<>();
        if (!isFull) {
            for (int i = 0; i < end; i++) {
                items.add(buffer[i]);
            }
        } else {
            for (int i = start; i < size; i++) {
```

```
            items.add(buffer[i]);
        }
        for (int i = 0; i < end; i++) {
            items.add(buffer[i]);
    }   }
    return items;
} }
```

LAUFZEITVERHALTEN

```
public static void function(int[] arr) {
    for (int i = 1; i < arr.length + 1; i++) {
        for (int i1 = 1; i1 < arr.length; i1 *= 2) {
            System.out.println(arr[i - 1]);
}   }   }
```

Lösung: $O(n * \log(n))$

```
public static void createPairs(int[] array) {
    for (int value : array) {
        for (int e : array) {
            System.out.println(value + ", " + e);
    }   }
    return null;
}
```

Lösung: $O(n^2)$

```
public static void addToSelf(int[] array) {
    for (int i : array) {
        for (int i1 = 0; i1 < 10000; i1++) {
            i = i + i1;
            System.out.println(i);
    }   }
    return null;
}
```

Lösung: $O(n)$

```
void searchBinaryIterative(int[] sortedArr, int searchEl) {
    int start = 0;
    int end = sortedArr.length - 1;
    while (start <= end) {
        int pivot = start + ((end - start) / 2);
        if (searchEl > sortedArr[pivot]) {
            start = pivot + 1;
        } else if (searchEl<sortedArr[pivot] && start != pivot) {
            end = pivot - 1;
        } else {
            System.out.println(searchEl + " an Position " + pivot + "
enthalten.");
            return null;
    }   }
    return null;
}
```

Lösung: $O(n * \log(n))$

BACKTRACKING BEISPIELE

BACKTRACKING (LABYRINTH, MAZE)

```
public class Rat {
    private int amountOfSteps;
    private final Maze maze;
    public Rat(Maze maze) {
        this.maze = maze;
    }
    public void walk(int x, int y) {
        if (step(x, y)) {
            maze.setField(x, y, State.WALKED);
        }
    }
    // Backtracking method
    public boolean step(int x, int y) {
        amountOfSteps++;
        System.out.println(maze);
        // Return true in case the goal was found
        if (maze.checkField(x, y, State.GOAL)) {
            return true;
        }
    // Return false in case a wall or already walked path is
reached
        if (maze.checkField(x, y, State.WALL)
                || maze.checkField(x, y, State.WALKED)
                || maze.checkField(x, y, State.BACKTRACKED)) {
            return false;
        }
        // Mark current location as walked
        maze.setField(x, y, State.WALKED);
        // Try to go Right
        if (step(x, y + 1)) { return true; }
        // Try to go Up
        if (step(x - 1, y)) { return true; }
        // Try to go Left
        if (step(x, y - 1)) { return true; }
        // Try to go Down
        if (step(x + 1, y)) { return true; }
        /* In Case none of the above is possible because a wall
or an
        already walked path was reached start backtracking */
        // Mark current location as backtracked
        maze.setField(x, y, State.BACKTRACKED);
        // Go back
        return false;
    }
    public int getAmountOfSteps() {
        return amountOfSteps;
}  }
public class Maze {
 public boolean checkField(int x, int y, State state) {
   if ((maze.length <= x || x < 0) || (maze[0].length <= y || y < 0)){
     return State.WALL.getMarkingChar () ==  state.getMarkingChar(); }
   return maze[x][y] == state.getMarkingChar ();
 }
 public void setField(int x, int y, State state) {
    maze[x][y] = state.getMarkingChar();
}  }
public enum State {
    OPEN(' '), WALL('#'), WALKED('o'), BACKTRACKED('.'),
CURRENT_POSITION('X'), GOAL('G'), START('S');
    private char markingChar;
    State(char c) {
        this.markingChar = c;
    }
    char getMarkingChar () {
        return markingChar;
}   }
```

TOWER OF HANOI

```
public void runHanoi(int nofDisks, int source, int target, int
reserve) {
    if (nofDisks == 0) {
        return;
    }
    runHanoi(nofDisks - 1, source, reserve, target);
    runHanoi(nofDisks - 1, reserve, target, source);
}
```

HASHING CODE

OFFENE ADRESSIERUNG (LINEARE SONDIERUNG)

```
private int findAvailableSlot(int indexInHashTable, K key) {
    int probedIndex = indexInHashTable;
    do {
        if(isAvailable(probedIndex)) {
            return probedIndex;
        }
        else if(table[probedIndex].getKey().equals(key)) {
            return probedIndex;
        }
        probedIndex = (probedIndex + 1) % capacity;
    } while (probedIndex != indexHashTable);
    return -1;
}
@Override
public V get(K key) {
    int hashIndex = Math.abs(key.hashCode() % capacity);
    int availableSlot = findAvailableSlot(hashIndex, key);
    if(availableSlot == -1) {
        return null;
    }
    return table[availableSlot].getValue();
}
@Override
public V remove(K key) {
    int hashIndex = Math.abs(key.hashCode() % capacity);
    int indexInHashMap = probeForDeletion(hashIndex, key);
    if(indexInHashMap == -1){ return null; }
    V answer = table[indexInHashMap].getValue();
    table[indexInHashMap] = DELETED;
    return answer;
}
private boolean isAvailable(int i){
    return table[i] == null || table[i] == DELETED;
}
private int probeForDeletion(int hashIndex, K key) {
    int i = hashIndex;
    do {
        if (table[i] == null) {
            return -1;
        } else if (table[i].getKey().equals(key)) {
            return i;
        }
        i = (i + 1) % capacity;
    } while (i != hashIndex);
    return -1;  }
```

HASHING

DYNAMISCHES HASHING (ERWEITERTES HASHING)

```
private int getPosition(Object o, int d) {
    BitSet bits = hashValueToBitSet(o);
    int pos = 0;
    for (int i = 0; i < d; i++) {
        pos *= 2;
        if(bits.get(i)) {
            pos++;
    }   }
    return pos;
}
private void extenIndex() {
    int newSize = 1 << globalDepth;
    Block newIndex[] = new Block[newSize * 2];
    for(int i = 0; i newSize; i++) {
        newIndex[2 * i] = hashIndex[i];
        newIndex[2 * i + 1] = hashIndex[i]
    }
    hashIndex = newIndex;
    globalDepth++;
}
public void add(Object o) {
    int index = getPosition(o, globalDepth);
    Block block = hashIndex[index];
    if(block.elements().contains(o)) { return; }
    while(block.elements().size() == MAXSIZE){
        if(block.getDepth() == globalDepth) {
            extendIndex();
            index = getPosition(o, globalDepth);
        }
        splitBlock(index);
        block = hashIndex[index];
    }
    block.elements().add(o);
}
```

HASHMAP (MULTIMAP)

```
public class HashMultimap<K, V> implements Multimap<K, V> {
  Map <K, List<V>> map = new HashMap<>();
  int numberOfEntries = 0;
  @Override
   public int size() { return numberOfEntries;    }
  @Override
   public boolean isEmpty() { return (numberOfEntries == 0);    }
  @Override
    public Iterable<V> get (K key) {
        List<V> secondary = map.get(key);
        if (secondary != null) { return secondary; }
        return new ArrayList<>();
    }
  @Override
   public void put(K key, V value) {
        List<V> secondary = map.get(key);
        if (secondary == null) {
            secondary = new ArrayList<>();
            map.put(key, secondary);
        }
        secondary.add(value);
        numberOfEntries++;
   }
  @Override
   public boolean remove(K key, V value) {
        boolean wasRemoved = false;
        List<V> secondary = map.get(key);
        if (secondary != null) {
            wasRemoved = secondary.remove(value);
            if (wasRemoved) {
                numberOfEntries--;
                if(secondary.isEmpty()) {
                    map.remove(key);
        } } }
        return wasRemoved;
   }
  @Override
   public Iterable<V> removeAll(K key) {
        List<V> secondary = map.get(key);
        if (secondary != null) {
            numberOfEntries -= secondary.size();
            map.remove(key);
        } else { secondary = new ArrayList<>(); }
        return secondary;
   }
  @Override
   public Iterable<Map.Entry<K, V>> entries() {
        List<Map.Entry<K, V>> result = new ArrayList<>();
        for (Map.Entry<K, List<V>> secondary : map.entrySet()) {
            K key = secondary.getKey();
            for (V value : secondary.getValue()) {
                result.add(new AbstractMap.SimpleEntry<>(key, value));
}    } return result;  }  }
```

CODE SCHREIBEN

- ☐ statt for() stream api genutzt wenn sinnvoll (z.B. .sum())
- ☐ Beispielcode nutzen (z.B. für Klasse die Methode() > Klasse.Methode()
- ☐ Typen korrekt angegeben? Sichtbarkeit bei Klassen? Instanzvariabeln immer «private» deklarieren!
- ☐ korrekter Rückgabetyp
- ☐ bei loop 1. und letztes Element nicht vergessen
- ☐ bei collections wrapperklassen nutzen, nicht int sondern Integer
- ☐ exceptions beachten

Checkliste bei Textantworten:

// - type erasure, type bound, wildcards (generische invarianz) erwähnt
// - lastfaktor erklärt