# File I/o

## Byte Streams

Lesen Byteweise (8-Bit) Daten ein

**Input Beispiel:**

Diagram

Description automatically generatedFileInputStream in = new FileInputStream("myFile.data");  
int value = in.read();  
while (value >= 0) {  
 byte b = (byte) value;  
 *// work with b* value = in.read();  
}  
in.close();

**Output Beispiel:**

FileOutputStream out = new FileOutputStream("test.data");  
while (…) {  
 byte b = …;  
 out.write(b);  
}  
out.close();

## Character Streams

Unicode mit 16-Bit (UTF-16) codiert

**File-Reader Beispiel:**

try (FileReader reader = new FileReader("quotes.txt")) {  
 int value = reader.read();  
 while (value >= 0) {  
 char c = (char) value;  
 *// use character* value = reader.read();  
 }  
}

**File-Writer Beispiel:**

try (FileWriter writer = new FileWriter("test.txt", true)) {  
 writer.write("Hello!");  
 writer.write("\n");  
}

**Zeilenweise lesen:**

try (BufferedReader reader = new BufferedReader(new FileReader("quotes.txt")) {  
 String line = reader.readLine();  
 while (line != null) {  
 System.*out*.println(line);  
 line = reader.readLine();  
 }  
}

## serialisierung / deserialisierung

**Serialisierung:** Zustand eines Objekts in Bytestrom codieren durch Serializable-Interface Implementation. Ansonsten wird ein NotSerialitableException geworfen.

class Person implements Serializable {  
 private String firstName;  
 private String lastName;  
 *// …*}

**Deserialisierung:** Objekt aus Bytestrom wiederherstellen

Diagram

Description automatically generatedFileInputStream fstream = new FileInputStream("serial.bin");  
try (ObjectInputStream ois = new ObjectInputStream(fstream)) {  
 Person p = (Person) ois.readObject();  
 *//…*}

**Serial Version UID:** Mögliche Probleme bei Deserialisierung wenn Implementation sich ändert. Lösung durch Versionsangabe, welche vor der Deserialisierung überprüft wird und InvalidCastException erzeugt falls Versionsnummer nicht übereinstimmen.

class Person implements Serializable {  
 private static final long *serialVersionUID* = 1L;  
 *//…*}

# generics

Mit Hilfe von Generics können Typen (Klassen und Schnittstellen) bei der Definition von Klassen, Schnittstellen und Methoden als Parameter verwendet werden. Ähnlich wie die bekannteren formalen Parameter, die in Methodendeklarationen verwendet werden, bieten Typparameter eine Möglichkeit, denselben Code mit unterschiedlichen Eingaben wiederzuverwenden. Der Unterschied besteht darin, dass die Eingaben für Formalparameter Werte sind, während die Eingaben für Typparameter Typen sind.

**Generische Klasse:**

public class MyClass<T> { … }

**Generische Methode:**  
public <E> Stack<E> multiPush(E value, int times) {  
 Stack result = new Stack<E>();  
 for (int i = 0; i < times; i++) {  
 result.push(value);  
 }  
 return result;  
}

**Typ-Parameter:** Platzhalter für generischen Typ

**Typ-Argument:** Typ bei Einsatz angeben

A picture containing text

Description automatically generated

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Typ | Lesen | Schreiben | Kompatible Typ-Argumente |
| Invarianz | C<T> | ✓ | ✓ | T |
| Covarianz | C<? extends T> | ✓ | ✗ | T und Subtypen |
| Contravarianz | C<? super T> | ✗ | ✓ | T und Basistypen |
| Bivarianz | C<?> | ✗ | ✗ | Alle |

**Vorteile:**

1. Strengere Typüberprüfungen 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.
2. Keine Casts notwendig

## type bound

Typ Parameter können gebunden werden sodass nur bestimmte Typen erlaubt sind. Alle Subtypen können verwendet werden.

class GraphicStack<T extends Graphic> {  
 public void drawAll() {  
 *// …* }  
}

Mehrere Typen können mit dem & verbunden werden. Jedoch können ab dem zweiten Type nur noch von Interaces verwendet werden.

class ClassName<T extends Type1 & Type2 & … >

## type erasure

Typ Informationen werden bewusst entfernt und vor dem Kompilieren durch Object ersetzt. Damit kann Rückwärtskompatibilität garantiert werden. Generische Informationen separat in Metadaten gespeichert und bleiben Compiler bekannt.

# Algorithmen

## Greedy

In jedem Teilschritt die möglichst optimale Lösung wählen um möglichst viel zu erreichen. Optimiert lokal um global zu optimieren.

Graphical user interface, text

Description automatically generated

O-Notation:

**Problem:** Liefert nicht unbedint optimale Lösung.

## Binärsuche

Problem in kleinere Probleme aufteilen und rekursive Rückführung des Problems mit kleinerer Eingabemenge. Funktioniert nur wenn Elemente nach einer logischen Funktion sortiert sind.

O-Notation:

## backtracking

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

Diagram

Description automatically generatedDiagram

Description automatically generated with medium confidence

**Vorgehen:**

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

## empirische analyse

Algorithmus implementieren und mit unterschiedlichen Eingaben ausführen. Ergebnisse aufzeichnen und vergleichen.

**Lineare Suche:**

public int lineareSuche(int gesucht, int[] daten) {  
 for (int i = 0; i < daten.length; i++) {  
 if (daten[i] == gesucht) {  
 return i;  
 }  
 }  
 return -1;  
}

**Binäre Suche:**

int binarySearch(int array[], int x, int low, int high) {  
 if (high >= low) {  
 int mid = low + (high - low) / 2;  
 if (array[mid] == x) *// Lösung gefunden* return mid;  
 if (array[mid] > x) *// Linken Hälfte suchen* return binarySearch(array, x, low, mid - 1);  
 return binarySearch(array, x, mid + 1, high); }  
 return -1;  
}

Ergebnisse können durch Störungen in Hardware, Software und Systembelastung verfälscht werden. Algorithmus muss implementiert sein.

# 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 Infinte Loop.

**Rekursive Aufrufe:** Rufen Methode wieder auf und bewegen sich Richtung Base Case

**Lineare Rekursion:** Ein rekursiver Aufruf startet höchstens einen weiteren rekursiven Aufruf

**Binäre Rekursion:** Rekursiver Aufruf macht höchstens zwei rekursive Aufrufe.

**Endrekursion:** Funktion, bei der rekursiver Aufruf letzter Schritt ist. Weniger Speicherbedarf auf Call Stack und kann in iterative Funktion umgewandelt werden.

**Teile und Herrsche:** Problem aufteilen, in kleinere Probleme aufteilen und Funktion rekursiv mit kleinerer Eingabemenge aufrufen.

## Big-O Notation

A picture containing text, person, outdoor, different

Description automatically generatedMit Big-O Notation lässt sich Laufzeit und Speicherverbrauch eines Algorithmus mittels algebraische Terme beschreiben. Konzentration auf den Worst Case.

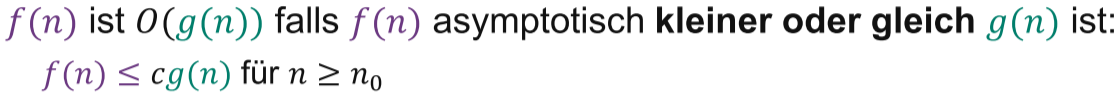
Diagram

Description automatically generated

**Chart, line chart

Description automatically generated**

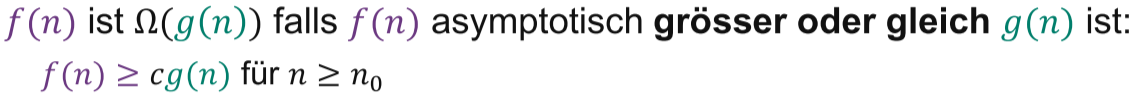
**Omega Notation:** Ω-Notation stellt die untere Grenze der Laufzeit eines Algorithmus dar. Sie gibt also die bestmögliche Komplexität eines Algorithmus an.



Chart, line chart

Description automatically generated

**Theta Notation:** Die Theta-Notation umschließt die Funktion von oben und unten. Da sie die obere und untere Schranke der Laufzeit eines Algorithmus darstellt, wird sie zur Analyse der durchschnittlichen Komplexität eines Algorithmus verwendet.



Chart, line chart

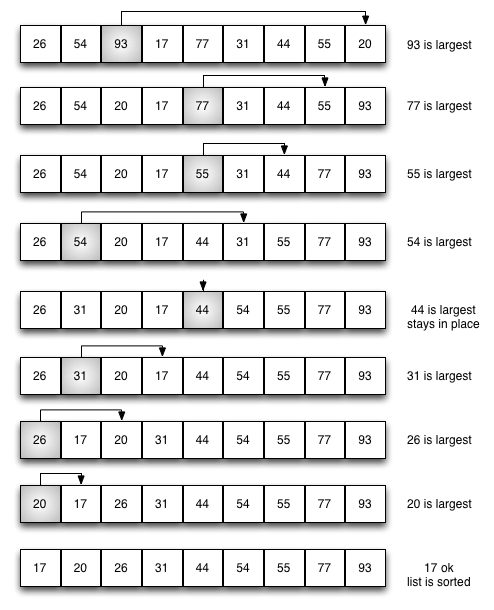
Description automatically generated

|  |  |  |
| --- | --- | --- |
| Wachstumsrate | Name | Beschreibung |
|  | Konstant | Statement (1 Linie Code) |
|  | Linear | In Hälfte teilen (Binäre Suche) |
|  | Logarithmisch | Loop |
|  | Linearithmische | Sortieralgorithmen (Merge Sort) |
|  | Quadratisch | Doppelter Loop |
|  | Kubisch | Dreifacher Loop |
|  | Exponential | Brute Force |

# sortieren

## selection sort

Von der unsortierten Liste das grösste Element in neue List einfügen und danach das nächscht grössere Element wählen bis unsortiere Liste leer ist.



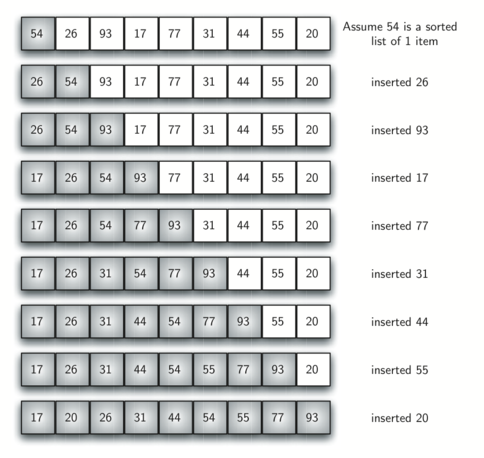
A picture containing diagram

Description automatically generated

**Laufzeit:**

## insertion sort

Durch jedes Element in unsortierten List iterieren und in korrekt sortierte Position der neuen Liste einfügen.



A picture containing diagram

Description automatically generated

**Laufzeit:**

## Bubble sort

Durch Liste iterieren und in je zweier Paare vergleichen und Positionen vertauschen falls unsortiert. Wiederholen bis Liste sortiert ist.

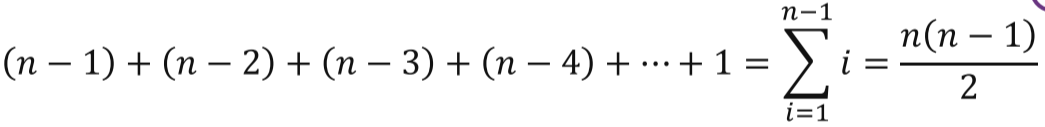
A screenshot of a computer

Description automatically generated with low confidence

etc.

A screenshot of a computer

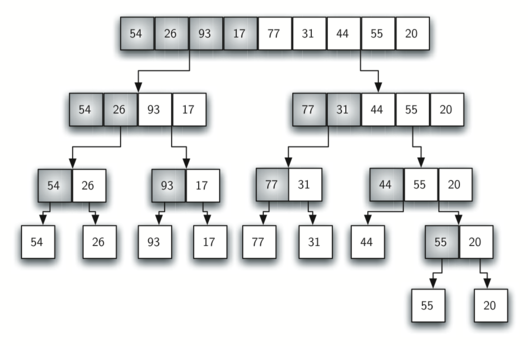
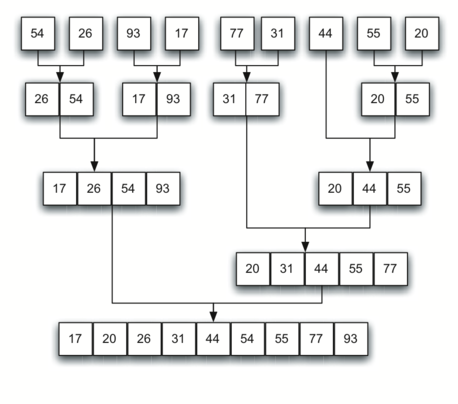
Description automatically generated with low confidence

****

**Laufzeit:**

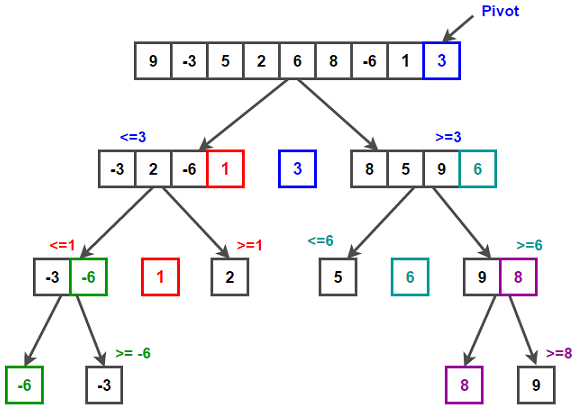
## merge sort

Liste in der Hälfte aufteilen bis nur noch zwei Elemente in Liste sind und diese sortieren. Mit anderer gleich grosser Liste zusammenfügen und sortieren.



**Laufzeit:**

## quick sort

List mit einem Pivot teilen und restliche Elemente in zwei Unter-Arrays teilen, die entweder grösser oder kleiner als Pivot sind. Quicksort rekursiv auf beide Unter-Arrays aufrufen.

**Laufzeit:**  bis zu

## merge sort vs. quick sort

Table

Description automatically generated

# abstrakter datentyp (adt)

Ein ADT ist eine Abstraktion einer konkreten Datenstruktur. Sie beschreibt die Attribute, Operationen auf den Attributen und die Ausnahmen und Fehler. Beschreibt WAS, aber nicht WIE. Beschreibt Datenstrukture unabhängig von konkreter Implementierung. Datenstruktur implementiert Schnittstelle (ADT).

Diagram

Description automatically generated**Ziel:** Kapselung (Nutzung ausschliesslich über Schnittstelle) und Geheimnisprinzip (Interne Realisierung ist verborgen)

**Array vs List:**

|  |  |
| --- | --- |
| Array | List |
| Kann multidimensional sein | Kann nur eindimensional sein |
| Länge statisch und fixe Länge | Dynamische Länge |
| Schneller wegen fixer Länge | Langsamer wegen dynamischer Länge |
| Type-unsafe da keine Generics | Type-safe da Generics möglich |

## stack

**Grundprinzip:** First In, Last out

Graphical user interface

Description automatically generated with medium confidence**Interface:**

|  |  |
| --- | --- |
| void push(E elem) | Element auf Stapel legen |
| E pop() | Oberstes Element entfernen und zurückgeben |
| E top() | Liefert zuletzt eingefügtes Element ohne dies zu entfernen |
| int size() | Zahl gespeicherter Elemente |
| boolean isEmpty() | Zeigt ob Stack leer ist |

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

Icon

Description automatically generated

**Zeitkomplexität:** Pop und Push Operationen sind beide

## linked list

Eine einfach verkettete Liste besteht aus Sequenz von Knoten, von denen jeder aus einem Element und einem Zeiger zum nächsten Knoten besteht.



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;  
 }  
}

**Anwendungen:** Undo Funktionalität, Hash Tabellen, Graphen

**Doubly-Linked-List:** Jeder Knoten speichert Verbindung zum Vorgänger und Nachfogler

A picture containing text, clock, gauge

Description automatically generated

**Zeitkomplexität:**

|  |  |  |
| --- | --- | --- |
|  | Schlechtester Fall | Bester Fall |
| Suchen |  |  |
| Einfügen |  |  |
| Löschen |  |  |

## queue

**Grundprinzip:** First In, First out (FIFO)

**Interface:**

|  |  |
| --- | --- |
| void enqueue(E elem) | Element am Ende der Queue einfügen |
| E dequeue() | Element vom Anfang entfernen und zurückgeben |
| E first(); | Liefert erstes Element, ohne es zu entfernen. |
| int size(); | Zahl gespeicherter Elemente |
| boolean isEmpty(); | Ist Queue leer? |



**Anwendungen:** Website-Verkehr, Router/Switches, Playlist in Spotify

**Zeitkomplexität:** Enqueue und Dequeue sind beide

## priority queue

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

**Interface:**

|  |  |
| --- | --- |
| insert(K k, V v) | Fügt Eintrag mit Schlüssel k und Wert v ein |
| removeMin() | Entfernt Eintrag mit kleinstem Schlüssel und gibt ihn zurück |
| min() | Liefert Eintrag mit kleinstem Schlüssel, ohne ihn zu entfernen |
| int size() | Anzahl Elemente in Queue |
| boolean isEmpty() | Sind Elemente in der Queue? |

**Anwendungen:** Dijkstra-Algorithmus, Datenkompression in Huffman Code

**Zeitkomplexität:**

Table

Description automatically generated

## Baum

**Grundprinzip:** Zweidimensionale Datenstruktur welche eine hierarchische Beziehung darstellt

**Terminologie:**

|  |  |
| --- | --- |
| Begriff | Beschreibung |
| Wurzel | Knoten ohne Elternknoten (A) |
| Innere Knote | Knoten mit mind. einem Kind (A, B, C, D) |
| Blatt | Knoten ohne Kinder (E, I, J, K, ...) |
| Vorgängerknoten | Eltern, Grosseltern |
| Geschwister | Knoten mit selben Eltern |
| Subtree | Baum aus einem Knoten und seinen Nachfolgern |
| Tiefe eines Knotens | Anzahl Vorgänger |
| 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);  
}

**Anwendungen:** Hierarchie

**Diagram

Description automatically generated**

## binärer Baum

**Grundprinzip:** Spezialform von einem Baum. Jedoch besitzt jeder Knoten nur noch 2 Knoten. Jeder Knoten trägt einen Schlüssel, Alle Schlüssel im linken Teilbaum sind kleiner und alle Schlüssel im rechten Teilbaum sind grösser.

**Interface:**

**A picture containing clipart, clock

Description automatically generated**public interface BinaryTree<E> extends Tree<E> {  
 Position<E> left(Position<E> p);  
 Position<E> right(Position<E> p);  
 Position<E> sibling(Position<E> p);  
 Position<E> addRoot(E e);  
 Position<E> addLeft(Position<E> p, E e);  
 Position<E> addRight(Position<E> p, E e);  
}

**Graphical user interface, application

Description automatically generated**A picture containing clock, watch

Description automatically generated**Anwendungen:** Sortieralgorithmen

**Traversierung:**

|  |  |  |
| --- | --- | --- |
| Methode | Beschreibung | Reihenfolge |
| Preorder (W-L-R) | Knoten wird vor seinen Kindern besucht | A, B, C, D, E |
| Postorder (L-R-W) | Knoten wird nach seinen Nachfolgern besucht | A, C, B, E, D |
| Inorder (L-W-R) | Knoten nach linkem Subtree und vor rechtem Subtree besuchen | D, B, A, C, E |
| Breadth-First | Alle Knoten einer Stufe besuchen bevor Nachfolgeknoten besucht werden | D, B, E, A, C |

**Zeitkomplexität:**

|  |  |  |
| --- | --- | --- |
|  | Schlechtester Fall | Bester Fall |
| Suchen |  |  |
| Einfügen |  |  |
| Löschen |  |  |

## hash maps / hashing

**Grundprinzip:** Menge von Schlüssel-Wert Paaren von denen jeder Schlüssel unique ist.

**Interface:**

|  |  |
| --- | --- |
| insert(K k, V v) | Fügt Eintrag mit Schlüssel k und Wert v ein |
| removeMin() | Entfernt Eintrag mit kleinstem Schlüssel und gibt ihn zurück |
| min() | Liefert Eintrag mit kleinstem Schlüssel, ohne ihn zu entfernen |
| int size() | Anzahl Elemente in Queue |
| boolean isEmpty() | Sind Elemente in der Queue? |

**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. Ziel ist, Werte mit O(1) lesen zu können.

Diagram

Description automatically generatedChart, diagram

Description automatically generated

**Anomalien:** Leerstellen sind Platzverschwendung, Ein Hash-Code für mehrere Einträge ergeben Kollisionen

Table

Description automatically generated

**Mögliche Methoden zum Hashen:**

1. Divisionsrestverfahren
2. Integer Cast  
   Schlüssel als Integer interpretieren. Gute Wahl, solange Anzahl Bits Interpretation als Integer erlaubt.
3. Komponentensumme  
   Jedes Zeichen des Keys als ASCII und summieren. Wenn Reihenfolge der Zeichen wichtig ist, kann man das Polynom verwenden, wobei a ein Zeichen ist.

**Eigenschaften guter Hashfunktion:** Konsistenz, Effiziente Berechnung, Gleichmässige Verteilung der Schlüssel

**Wichtig:** Wenn equals() überschrieben wird muss im Normalfall hashcode() auch überschrieben werden, sonst ergeben sich beim Einsatz von Maps Fehler.

**Erweiterung:** 2 Teilfunktionen für Hashing verwenden. Hash-Funktion soll Schlüssel möglichst zufällig verteilen. Kompressionsfunktion soll Schlüssel in fixex Intervall transformieren.

A picture containing graphical user interface

Description automatically generated

**Beispiel Kompressionsfunktion:** (Grösse der Hash-Tabelle ist oft Primzahl)

**Gleichheit:** Referenzvergleich, Inhaltsvergleich

**Kollision:** Menge der Eingabewerte übersteigt Grösse der Hashtabelle. Mehrere Schlüssel werden auf einen Behälter abgebildet. Datensatz s mit Schlüssel k heisst Überläufer, wenn durch h(w) zugewiesener Behälter belegt ist.

**Geschlossene Addressierung:** Jede Zelle der Tabelle zeigt auf Liste, Einfache Implementation, jedoch zusätzliche Datenstruktur und Speicherbedarf erforderlich. Suchen wird verlangsamt, da auch in der Liste linear iteriert werden muss.

Diagram, schematic

Description automatically generated

**Offene Addressierung:** Platz in anderem Behälter suchen, Überläufer in nächste verfügbare Zelle der Tabelle einfügen (Linear Probing). Inspizierte Zellen werden als probe bezeichnet.

Diagram

Description automatically generated

**Sondierung:**

|  |  |
| --- | --- |
| Linear positiv | Linear vorwärts nach freier Stelle suchen |
| Linear negativ | Linear rückwärts nach freier Stelle suchen |
| Quadratisch | Mit quadratischer Funktion nach freier Stelle suchen |
| Alternierend | Nach freier Stelle abwechselnd davor und dahinter suchen |
| Alternierend Quadratisch | Mit quadratischer Funktion abwechselnd nach freier Stelle suchen |

## multi maps

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

Icon, rectangle

Description automatically generated

**Interface:**

public interface Multimap<K, V> {  
 public Iterable<V> get(K key);  
 public void put(K key, V value);  
 public boolean remove(K key, V value);  
 public Iterable<V> removeAll(K key);  
 public Iterable<Map.Entry<K, V>> entries();  
 public int size();  
 public boolean isEmpty();  
}

# design patterns

**Motivation:** Wiederverwendbare Lösungen für wiederkehrende Probleme verwenden. Vorlagen die in konkreten Problemen angewandt werden können. Lösung ist abstrakt un in neuen Kontexten anwendbar.

**Elemente:**

|  |  |
| --- | --- |
| Mustername | Benennung des Patterns |
| Problem | Beschreibung wann Pattern anzuwenden ist |
| Lösung | Elemente aus denen Pattern besteht |
| Konsequenzen | Vor- und Nachteile bei Anwendung des Patterns |

**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

Ein Iterator ist ein Verhaltensmuster, mit dem man Elemente einer Collection nach einem gewissen Pattern iterieren kann, ohne die zugrunde liegende Darstellung (Liste, Stapel, Baum usw.) offenzulegen.

public interface Iterator<E> {  
 boolean hasNext();  
 E next() throws NoSuchElementException;  
 void remove() throws UnsupportedOperationException;  
}

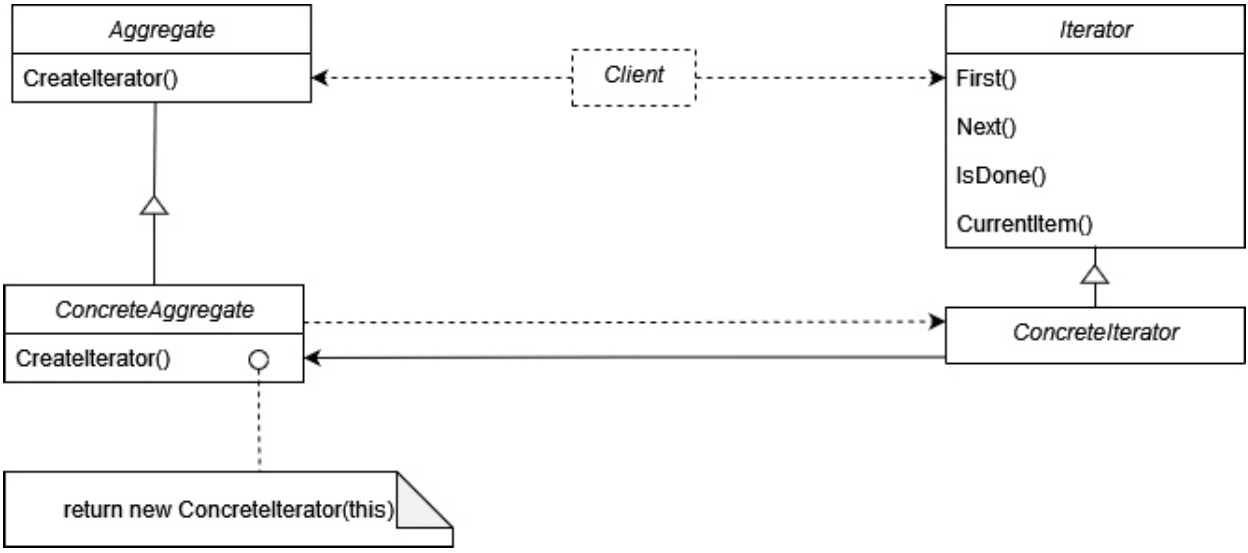
**Lazy Iterator:** Iteration auf originaler Datenstruktur. Niedrigere Speicherkosten . Jedoch bei Änderungen auf Datenstruktur können korrekte Iteration verunmöglichen. Fehl-Verhalten bei unerwarteten Modifikationen der Ausgangs-Datenstruktur

Table

Description automatically generated

**Snapshot Iterator:** Originale Datenstruktur nicht verändern. Bei Erzeugung wird eine Kopie der Ausgans-Datenstrukturen erzeugt. Änderungen auf Ausgangs-Datenstruktur beinflussen iteration nicht. Kosten: Speicher und Laufzeit

public SSArrListIter(T[] elem, int size, Comparator<T> comp) {  
 this.elements = copy(elements, size);  
 Arrays.*sort*(this.elements, comp);  
 this.comparator = comp;  
}



## visitor

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.

Diagram

Description automatically generated

## template method

Backbone eines Algorithmus definieren, Teilschritte später in Subklassen spezifizieren. Lässt Subklassen Teile des Algorithmus verfeinern, ohne Strukture des Algorithmus zu verändern.

Gemeinsame, unveränderliche Teile werden in abstrakter Klasse implementiert (Template). Variablen Schritte werden in Methode ausgelagert.

Diagram

Description automatically generated

**Anwendungen:** Frameworks

# beispiele

## for mit iterator

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

## 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:

*/\*  
 Antwort: O(1). Konstante Laufzeit  
 Es wir nur ein Wert verglichen. Die Laufzeit wird nicht grösser, wenn der Wert grösser wird  
 \*/*public static Void createPairs(int[] array) {  
 for (int value : array) {  
 for (int e : array) {  
 System.*out*.println(value + ", " + e);  
 }  
 }  
 return null;  
}

*/\*  
 Gewisse Funktionen returnen 'Void' anstatt void. Dies ist nur damit man die Funktinen als Functional Interfaces  
 benutzen kann im Visualizer und haben sonst keine Einwirkung.  
 \*/*public static boolean isEven(int value) {  
 return value % 2 == 0;  
}

*/\*  
 Antwort: O(n^2). Quadratische Laufzeit  
 Der äussere Loop läuft n mal. Für jedes durchlaufen des ässueren Loops läuft der innere Loop n mal -> O(n \* n)  
 \*/*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;  
}

*/\*  
 Antwort: O(n). Lineare Laufzeit  
 Der äussere Loop läuft n mal durch. Der innere Loop läuft 10'000 mal durch für jedes Element.  
 Diese 10'000 mal sind jedochabhängig von n und nicht signifikant deshalb -> O(n)  
 \*/*public static Void searchBinaryIterative(int[] sortedArray, int searchElement) {  
 int start = 0;  
 int end = sortedArray.length - 1;  
 while (start <= end) {  
 int pivot = start + ((end - start) / 2);  
 if (searchElement > sortedArray[pivot]) {  
 start = pivot + 1;  
 } else if (searchElement < sortedArray[pivot] && start != pivot) {  
 end = pivot - 1;  
 } else {  
 System.*out*.println(searchElement + " an Position " + pivot + " enthalten.");  
 return null;  
 }  
 }  
 return null;  
}

*/\*  
 Antwort: O(log n). Logarithmische Laufzeit  
 Bei jedem durchlauf wird die Anzahl der Elemente, welche durchlaufen werden halbiert.  
 Wie oft n halbiert werden muss, ist dann ld(n) -> O(log n)  
 ld = Logarithmus dualias. Also Logarithmus zur Basis von 2  
 In der O Notation wird die Basis nicht hingeschrieben, darum O(log n)  
 \*/*public static Void printArrayFrontAndBackwards(int[] array) {  
 for (int j : array) {  
 System.*out*.println(j + ' ');  
 }  
 System.*out*.println();  
  
 for (int i = array.length - 1; i >= 0; i--) {  
 System.*out*.println(array[i] + ' ');  
 }  
 return null;  
}

*/\*  
 Antwort: O(n). Lineare Laufzeit  
 Die Funktion dauert 2n mal. Was in der O Notation O(n) geschrieben wird.  
  
 \*/*public static void main(String[] args) {  
 int[] a = new int[]{1, 2, 67, 435, 354, 345, 346, 347, 34, 53, 45, 6, 567, 8, 85};  
 a = Arrays.*stream*(a).sorted().toArray();  
 *searchBinaryIterative*(a, 345);  
}

Text

Description automatically generated

## laufzeitanalyse

Beweisen Sie: ist

Gesucht: und , sodass für

Lösung:

## To Array/collection

Zu Array:  
Person[] p = peopleStream.toArray(Person[]:: new)   
Zu Collection:  
List<Person> l = st.collect(Collectors.toList());

## generics

class X {}  
class Y extends X {}  
class Z extends Y {}

public class Main {  
 public static void print(Y y) {  
 System.*out*.println("y");  
 }  
 public static void print(Z z) {  
 System.*out*.println("z");  
 }  
 public static void main(String[] args) {  
 *print*((Y) new Y());*//Ausgabe: y  
 print*((Z) new Z());*//Ausgabe: z  
 print*((Y) new Z());*//Ausgabe: y  
 print*((Z) new Y());*//Ausgabe: ClassCastEx* }  
}

class Graphic {  
 void moveTo(Graphic other) {  
 System.*out*.println("Method 1");  
 }  
}  
class Circle extends Graphic {  
 void moveTo(Graphic other) {  
 System.*out*.println("Method 2");  
 }  
 void moveTo(Circle other) {  
 System.*out*.println("Method 3");  
 }  
}  
Circle c = new Circle();  
Graphic g = new Circle();

|  |  |
| --- | --- |
| c.moveTo(g); | Method 2 |
| c.moveTo(c); | Method 3 |
| g.moveTo(c); | Method 2 |
| g.moveTo(g); | Method 2 |
| ((Graphic)c).moveTo(g); | Method 2 |
| ((Circle)g).moveTo(c); | Method 3 |

## selection sort

void selectionSort(int array[]) {  
 int size = array.length;  
 for (int step = 0; step < size - 1; step++) {  
 int min = step;  
 for (int i = step + 1; i < size; i++) {  
 *// To sort in descending order, change > to <* if (array[i] < array[min]) {  
 min = i;  
 }  
 }int temp = array[step];  
 array[step] = array[min];  
 array[min] = temp;  
 }  
}

## Insertion sort

void insertionSort(int array[]) {  
 for (int step = 1; step < array.length; step++) {  
 int key = array[step];  
 int j = step - 1;  
 *// For descending order, change < to >* while (j >= 0 && key < array[j]) {  
 array[j + 1] = array[j];  
 --j;  
 }array[j + 1] = key;  
 }  
}

## bubble sort

void bubbleSort(int[] array) {  
 int size = array.length;  
 for (int i = 0; i < size - 1; i++)  
 for (int j = 0; j < size - i - 1; j++)  
  
 *// change > to < to sort in descending order* if (array[j] > array[j + 1]) {  
 *// swapping occurs if elements  
 // are not in the intended order* int temp = array[j];  
 array[j] = array[j + 1];  
 array[j + 1] = temp;  
 }  
}

## merge sort

public static void mergeSort(int[] a, int n) {  
 if (n < 2) {  
 return;  
 }  
 int mid = n / 2;  
 int[] l = new int[mid];  
 int[] r = new int[n - mid];  
  
 for (int i = 0; i < mid; i++) {  
 l[i] = a[i];  
 }  
 for (int i = mid; i < n; i++) {  
 r[i - mid] = a[i];  
 }  
 *mergeSort*(l, mid);  
 *mergeSort*(r, n - mid);  
  
 *merge*(a, l, r, mid, n - mid);  
}  
  
public static void merge(  
 int[] a, int[] l, int[] r, int left, int right) {  
  
 int i = 0, j = 0, k = 0;  
 while (i < left && j < right) {  
 if (l[i] <= r[j]) {  
 a[k++] = l[i++];  
 } else {  
 a[k++] = r[j++];  
 }  
 }  
 while (i < left) {  
 a[k++] = l[i++];  
 }  
 while (j < right) {  
 a[k++] = r[j++];  
 }  
}

## quick sort

public void quickSort(int arr[], int begin, int end) {  
 if (begin < end) {  
 int partitionIndex = partition(arr, begin, end);  
  
 quickSort(arr, begin, partitionIndex-1);  
 quickSort(arr, partitionIndex+1, end);  
 }  
}  
  
private int partition(int arr[], int begin, int end) {  
 int pivot = arr[end];  
 int i = (begin-1);  
  
 for (int j = begin; j < end; j++) {  
 if (arr[j] <= pivot) {  
 i++;  
  
 int swapTemp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = swapTemp;  
 }  
 }  
  
 int swapTemp = arr[i+1];  
 arr[i+1] = arr[end];  
 arr[end] = swapTemp;  
  
 return i+1;  
}

## heap sort

public void sort(int arr[]) {  
 int n = arr.length;  
  
 *// Build max heap* for (int i = n / 2 - 1; i >= 0; i--) {  
 heapify(arr, n, i);  
 }  
  
 *// Heap sort* for (int i = n - 1; i >= 0; i--) {  
 int temp = arr[0];  
 arr[0] = arr[i];  
 arr[i] = temp;  
  
 *// Heapify root element* heapify(arr, i, 0);  
 }  
}  
  
void heapify(int arr[], int n, int i) {  
 *// Find largest among root, left child and right child* int largest = i;  
 int l = 2 \* i + 1;  
 int r = 2 \* i + 2;  
  
 if (l < n && arr[l] > arr[largest])  
 largest = l;  
  
 if (r < n && arr[r] > arr[largest])  
 largest = r;  
  
 *// Swap and continue heapifying if root is not largest* if (largest != i) {  
 int swap = arr[i];  
 arr[i] = arr[largest];  
 arr[largest] = swap;  
  
 heapify(arr, n, largest);  
 }  
}

## Binärsuche

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

## backtracking (near light)

static final int *N* = 5;  
static final int[] *row* = {2, 1, -1, -2, -2, -1, 1, 2, 2};  
static final int[] *col* = {1, 2, 2, 1, -1, -2, -2, -1, 1};

private static boolean isValid(int x, int y) {  
 return x >= 0 && y >= 0 && x < N && y < N;  
}

public static boolean knightTour(int[][] visited, int x, int y, int pos) {  
 *// Feld markieren* visited[x][y] = pos;  
  
 *// Abbruchsbedingung* if (pos >= *N* \* *N*) {  
 return true;  
 }  
 *// Alle Züge probieren* for (int k = 0; k < 8; k++) {  
 int newX = x + *row*[k];  
 int newY = y + *col*[k];  
  
 *// Ist neue Feld im Spielbrett und unbesucht* if (*isValid*(newX, newY) && visited[newX][newY] == 0) {  
 *// Wird Abbruchsbedingung rekursiv erreicht?* if (*knightTour*(visited, newX, newY, pos + 1)) {  
 return true;  
 }  
 }  
 }  
 *// Falls 4, 5 nicht erfüllt sind, backtracking* visited[x][y] = 0;  
 return false;  
}

## backtracking (labyrinth)

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 wall or already walked path is reached* if (maze.checkField(x, y, State.*WALL*) || maze.checkField(x, y, State.*WALKED*)) {  
 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;  
 } *// Mark current location as backtracked* maze.setField(x, y, State.*BACKTRACKED*);  
  
 *// Go back* return false;

## stack

class Stack {  
 private int arr[];  
 private int top;  
 private int capacity;  
  
 Stack(int size) {  
 arr = new int[size];  
 capacity = size;  
 top = -1;  
 }  
  
 public void push(int x) {  
 if (isFull()) {  
 System.*out*.println("OverFlow Program Terminated");  
 System.*exit*(1);  
 }  
 System.*out*.println("Inserting " + x);  
 arr[++top] = x;  
 }  
  
 public int pop() {  
 if (isEmpty()) {  
 System.*out*.println("STACK EMPTY");  
 System.*exit*(1);  
 }  
 return arr[top--];  
 }  
  
 public int size() {  
 return top + 1;  
 }  
  
 public Boolean isEmpty() {  
 return top == -1;  
 }  
  
 public Boolean isFull() {  
 return top == capacity - 1;  
 }  
}

## queue

public class Queue {  
 int SIZE = 5;  
 int items[] = new int[SIZE];  
 int front, rear;  
  
 Queue() {  
 front = -1;  
 rear = -1;  
 }  
  
 boolean isFull() {  
 return front == 0 && rear == SIZE - 1;  
 }  
  
 boolean isEmpty() {  
 return front == -1;  
 }  
  
 void enQueue(int element) {  
 if (isFull()) {  
 System.*out*.println("Queue is full");  
 } else {  
 if (front == -1)  
 front = 0;  
 rear++;  
 items[rear] = element;  
 System.*out*.println("Inserted " + element);  
 }  
 }  
  
 int deQueue() {  
 int element;  
 if (isEmpty()) {  
 System.*out*.println("Queue is empty");  
 return -1;  
 } else {  
 element = items[front];  
 if (front >= rear) {  
 front = -1;  
 rear = -1;  
 } else {  
 front++;  
 }  
 System.*out*.println("Deleted -> " + element);  
 return (element);  
 }  
 }  
  
}

## rekursion (palindrom checker)

public static boolean isPalin(String text) {  
 String low = text.toLowerCase(Locale.*ROOT*);  
 if (text.length() == 0 || 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 (decimal to binary)

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

## Binärer baum

class BinarySearchTree {  
 class Node {  
 int key;  
 Node left, right;  
  
 public Node(int item) {  
 key = item;  
 left = right = null;  
 }  
 }  
  
 Node root;  
  
 BinarySearchTree() {  
 root = null;  
 }  
  
 void insert(int key) {  
 root = insertKey(root, key);  
 }  
  
 *// Insert key in the tree* Node insertKey(Node root, int key) {  
 *// Return a new node if the tree is empty* if (root == null) {  
 root = new Node(key);  
 return root;  
 }  
  
 *// Traverse to the right place and insert the node* if (key < root.key)  
 root.left = insertKey(root.left, key);  
 else if (key > root.key)  
 root.right = insertKey(root.right, key);  
  
 return root;  
 }  
  
 void inorder() {  
 inorderRec(root);  
 }  
  
 void inorderRec(Node root) {  
 if (root != null) {  
 inorderRec(root.left);  
 inorderRec(root.right);  
 }  
 }  
  
 void deleteKey(int key) {  
 root = deleteRec(root, key);  
 }  
  
 Node deleteRec(Node root, int key) {  
 *// Return if the tree is empty* if (root == null)  
 return root;  
  
 *// Find the node to be deleted* if (key < root.key)  
 root.left = deleteRec(root.left, key);  
 else if (key > root.key)  
 root.right = deleteRec(root.right, key);  
 else {  
 *// If the node is with only one child or no child* if (root.left == null)  
 return root.right;  
 else if (root.right == null)  
 return root.left;  
  
 *// If the node has two children  
 // Place the inorder successor in position of the node to be deleted* root.key = minValue(root.right);  
  
 *// Delete the inorder successor* root.right = deleteRec(root.right, root.key);  
 }  
  
 return root;  
 }  
  
 *// Find the inorder successor* int minValue(Node root) {  
 int minv = root.key;  
 while (root.left != null) {  
 minv = root.left.key;  
 root = root.left;  
 }  
 return minv;  
 }  
}

## 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;  
}

## Greedy

Diagram

Description automatically generated

public static void calcSolution(HashSet<String> statesNeeded, HashMap<String, HashSet<String>> stations) {  
 var finalStations = new HashSet<String>();  
 while (!statesNeeded.isEmpty()) {  
 String bestStation = "";  
 var statesCovered = new HashSet<String>();  
 for (String station : stations.keySet()) {  
 var covered = new HashSet<String>(statesNeeded);  
 covered.retainAll(stations.get(station));  
 if (covered.size() > statesCovered.size()) {  
 bestStation = station;  
 statesCovered = covered;  
 }  
 }  
 statesNeeded.removeAll(statesCovered);  
 finalStations.add(bestStation);  
 }  
 System.*out*.println(finalStations);  
}

public static void main(String[] args) {  
 VendingMachine vendingMachine = new VendingMachine();  
 try (Scanner scanner = new Scanner(System.*in*)) {  
 System.*out*.print("Amount to return in CHF = ");  
 List<BillCoin> returnList = vendingMachine.calculateChange(scanner.nextInt());  
 returnList.forEach(System.*out*::println);  
 }  
}

public List<BillCoin> calculateChange(int amount) {  
 List<BillCoin> changeList = new ArrayList<>();  
 for (BillCoin billCoin : BillCoin.*values*()) {  
 int quantity = amount / billCoin.amount;  
 for (int i = 0; i < quantity; i++) {  
 changeList.add(billCoin);  
 }  
 amount = amount % billCoin.amount;  
 }  
 return changeList;  
}

## Hashing

**Divisionsrestverfahren:**

**Komponentensumme:**

public int hash(String s) {  
 int hash = 0;  
 for (int i = 0; i < s.length(); i++) {  
 hash = (R \* hash + s.charAt(i)) % modulo;  
 }  
 return hash;  
}

**Polynom-Akkumulation:**

int hash = (((day \* R + month) % m) \* R + year) % m;

## DEKLARATIONEN

int x = 3;  
int y = 2;  
double z = 2;  
int i = 0;  
int[] a = {0, 1};

|  |  |  |
| --- | --- | --- |
| **Ausdruck** | **Ergebniswert** | **Ergebnistyp** |
| -x / y | -1 | Int |
| -x / z | -1.5 | Double |
| 1 + x \* y | 7 | Int |
| i++ > 0 | False | Boolean |
| x > y && a[y] > 0 | ArrayIndexOutOfBoundsEx | Boolean |
| x < y && a[y] < 0 | False | Boolean |
| x > y || a[y] > 0 | True | Boolean |
| x < y || a[y] < 0 | ArrayIndexOutOfBoundsEx | Boolean |
| a[1] % a[0] | ArithmeticException | Int |
| a[0] % a[1] | 0 | int |

|  |  |
| --- | --- |
| **Ausdruck** | **Wert für Ausdruck == true** |
| a + a \* 2 == 12 | a = 4 |
| b / 2 == b | b = 0 |
| !c | c = false |
| d.length == d[0] | d = new int[] { 1 } |
| e != 0 && e / 2 == (e + 1) / 2 | e = jede ungerade negative int |
| f + "" != f | f = "abc" (jeder String) |
| g + 1 == g + 2 | g = Float/Double  .POSITIVE\_INFINITY |
| h != h | h = Float/Double.NaN |

|  |  |  |
| --- | --- | --- |
| **Ausdruck** | **Ergebniswert** | **Ergebnistyp** |
| 1 + 3 / 2 | 2 | Int |
| 1.0 + 3 / 2 | 2.0 | double |
| 1 + 3.0 / 2 | 2.5 | double |
| 1 + 3 / 0 | ArithmeticEx div by 0 | int |
| 1 + 3.0 / 0 | Double.POSITIVE\_INFINITY | double |
| 1 + 3.0 / 0.0 | Double.POSITIVE\_INFINITY | double |

## vollständige induktion

Text, letter

Description automatically generated

Text

Description automatically generated