Generics

Terminologie

Generische Klasse

Klasse mit generischen Parametern

```
class Stack<T> {
// ...
}
```

Generischer Typ

Klasse die mit konkretem Typ instanziiert wird.

```
Stack<String> stack1;
Stack<Integer> stack2;
Stack<Person> stack3;
Stack<double[]> stack4;
Stack<Object> stack5;
```

- Kann ein statisches Attribut generisch sein?
-
```
public class MobileDevice<T> {
    private static T os;
    //..
}
```
  - **Nein**
    - Statisches Attribut wird zwischen Instanzen geteilt
    - Wäre unklar, welchen Typ das statische Attribut tatsächlich hat

Generische Klassen

Typ-Parameter

Platzhalter für generischen Typ

```
<T>
```

Typ-Argument

Typ bei Einsatz angeben

```
Stack<String> stack1;
Stack<Integer> stack2;
```

Generische Methoden

- Typ-Variable innerhalb spezifischer Methode
- Aufrufer bestimmt Typ-Argument

```
public <E> Stack<E> multiPush(E value, int times) {
    var result = new Stack<E>();
    ...
```

Weitere Beispiele (super / extends)

`class GraphicStack<T extends Graphic>`	`Stack<? super Graphic> stack;`
Jeder **subtyp** von `Graphic` ist zugelassen `GraphicStack<Rectangle>` → OK `GraphicStack<Circle>` → OK `GraphicStack<Graphic>` → OK `GraphicStack<String>` → Fail `GraphicStack<Object>` → Fail,	Jeder **super-typ** ist zugelassen `stack = new Stack<Object>();` → OK `stack = new Stack<Rectangle>();` → Fail

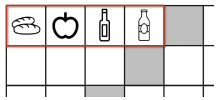
Listen und Arrays

Arrays

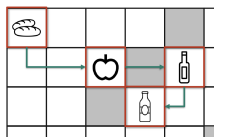
- Speichern gleichartige Objekte (Elemente)
- Zugriff über Index
- **Arrays speichern Referenzen auf Objekte** → ändert sich referenziertes Objekt, so ändert sich Inhalt des Arrays

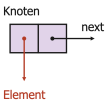
Arrays im Speicher

Array wächst → muss in neuen Block kopiert werden (neue Grösse [*1.5 Java-Standard])



LinkedList im Speicher





Auswahl Datenstruktur

Wovon kann die Wahl abhängig gemacht werden? → Laufzeiten von verschiedenen Use-Cases betrachten

Laufzeiten-Vergleich

	Lesen	**Element einfügen**
Array	$O(1)$	$O(n)$
Liste	$O(n)$	$O(1)$

Algorithmenparadigmen

Algorithmus: Präzise endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer Schritte

Backtracking

- Konzept:
  - Trial & Error
  - Aktueller Zweig führt nicht zur Lösung:
    - Zurück zur letzten Entscheidung (Backtracking)
    - Anderen Pfad probieren

Implementierung Pseudocode

1. Ziel erreicht:
   a) Lösungspfad aktualisieren
   b) True zurückgeben
2. Wenn (x, y) bereits Teil des Lösungspfades
   a) False zurückgeben
3. (x, y) als Teil des Lösungspfades markieren
4. In X-Richtung suchen: →
5. Keine Lösung: In Y-Richtung abwärts suchen: ↓
6. Keine Lösung: Zurück in X-Richtung suchen: ←
7. Keine Lösung: Aufwärts in Y-Richtung suchen: ↑
8. Immer noch keine Lösung: (x, y) aus Lösungspfad entfernen // Backtracking
9. False zurückgeben

Cookbook

1. Position auf Feld (Schachbrett/Labyrinth) markieren
2. Rekursionsabbruch (z.B. alle Felder besucht, Ausgang gefunden)
   - `return true;`
3. Alle Optionen probieren
   a) Neues Feld / Koordinate festlegen
   b) Überprüfen ob Feld gültig ist & noch nicht besucht wurde
      i. Wenn ja → prüfen ob rekursiver Aufruf `true` zurückgibt
         A. Wenn ja → `return true;`
4. Backtracking: Markierung von Feld entfernen & `return false;`

Knights-Tour

1. Feld markieren
2. Abbruchsbedinung: Position so gross wie Anzahl Felder → Alle Felder besucht
3. Alle möglichen Züge mit Springer probieren und neue (X|Y) berechnen
4. Überprüfen ob neues Feld innerhalb des Spielbretts & ob Feld noch nicht besucht
5. Überprüfen ob rekursiver Aufruf Abbruchsbedingung erreicht. Wenn ja: `return true;`
6. Falls beide Bedingungen (4, 5) nicht erfüllt sind → Markierung von Feld entfernen und mit `return false;` backtracken

Java

```
public static boolean knightTour(int[][] visited, int x, int y,
      int pos) {
    visited[x][y] = pos;

    if (pos >= N * N) {
        return true;
    }
    for (int k = 0; k < 8; k++) {
        int newX = x + row[k];
        int newY = y + col[k];

        if (isValid(newX, newY) && visited[newX][newY] == 0) {
            if (knightTour(visited, newX, newY, pos+1)) {
                return true;
            }
        }
    }

    visited[x][y] = 0;
    return false;
}

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

Rekursion

- Rekursionsabbruch (Base Case)
  - mindestens 1 base case
  - Rekursive Kette muss base case erreichen (abbrechen)
- Ausführung bewegt sich Richtung base case

NDigitNums

```
public static void findNDigitNums(String currentNumber, int n,
      int target, List<Integer>
    resultList) {
    if (n > 0 && target >= 0) {
        char digit = '0';
        if (currentNumber.equals("")) {
            digit = '1';
        }
        while (digit <= '9') {
            String num = currentNumber + digit;
            int newTarget = target - (digit - '0');
            findNdigitNums(num, n-1, newTarget, resultList);
            digit++;
        }
    } else if (n == 0 && target == 0) {
        resultList.add(Integer.valueOf(currentNumber));
    }
}
```

Fakultät

Funktion

$$f(n) = \begin{cases} 1 & if\ n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

Java Beispiel

```
static int recursiveFactorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * recursiveFactorial(n-1);
}
```

Endrekursion

Wenn linear rekursive Methode als **letzten Schritt** rekursiven Aufruf ausführt

- können leicht in nicht-rekursive Algorithmen umgewandelt werden
- nicht-rekursive Algorithmen benötigen meist weniger Ressourcen

Java Beispiel

```
private static int tailrecsum(int x, int total) {
    if (x == 0)
        return total;
    else
        return tailrecsum(x-1, total+x);
}
```

Greedy

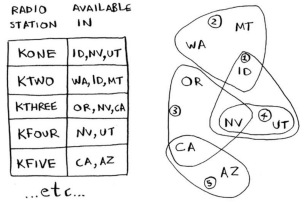
- Konzept: Es soll bestimmte Eigenschaft minimiert / maximiert werden
- Ansatz:
  - in jedem Teilschritt **möglichst viel** erreichen
  - in jedem Teilschritt **möglichst optimale** Lösung wählen

Rückgeld Beispiel

- Preis: 7 CHF
- Bezahlt mit: 15 CHF
- Ziel: Rückgeld mit minimaler Anzahl Münzen
- Vorgehen:
  - Nehme grösste Münze unter Zielwert und ziehe sie von diesem ab
  - Verfahre bis Zielwert = 0

Set-Covering Problem

Beispiel Abdeckung Radiostationen in den USA



- Optimaler Algorithmus
  - Alle Teilmengen der Stationen aufzählen
  - Minimale Anzahl Stationen wählen
  - **Problem:** 2^n mögliche Kombinationen (Potenzmenge)

→ **Es gibt keinen Algorithmus, der Problem schnell genug löst**

Alternative

- Wähle Sender, der meisten Sender abdeckt, die noch nicht abgedeckt sind
- Widerholen, bis alle Sender abgedeckt sind

```
public static void calculateSolution(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);
}
```

Divide-and-Conquer / Teile-und-Herrsche

- Problem aufteilen
- Kleinere Probleme lösen
- Rekursive Rückführung des Problems auf identisches mit kleinerer Eingabemenge

Binärsuche

Iterative Implementierung

```
public static void searchBinary(int[] intArr, int start, int end,
      int searchElement) {
    int pivot = start + ((end - start) / 2);
    if (intArr.length == 0) {
        System.out.println("Array leer.");
        return;
    }
    if (pivot >= intArr.length){
        System.out.println(searchElement + " nicht im Array
              enthalten.");
        return;
    }
    if (searchElement > intArr[pivot]) {
        System.out.println(start + " " + end + " " + pivot);
        searchBinary(intArr, pivot + 1, end, searchElement);
    } else if (searchElement < intArr[pivot] && start != pivot) {
        searchBinary(intArr, start, pivot - 1, searchElement);
    } else if(searchElement == intArr[pivot]) {
        System.out.println(searchElement + " an Position " +
              pivot + " enthalten.");
    } else{
        System.out.println(searchElement + " nicht im Array
              enthalten.");
    }
}
```

Rekursive Implementierung

```
int binarySearch(int array[], int x, int low, int high) {
    if (high >= low) {
        int mid = low + (high - low) / 2;
        if (array[mid] == x) // Wenn Element gefunden:
              zurueckgeben
            return mid;
        if (array[mid] > x) // In der linken Haelfte suchen
            return binarySearch(array, x, low, mid - 1);
```

```
        return binarySearch(array, x, mid + 1, high); // In der
                rechten Haelfte suchen
    }
    return -1;
}
```

Mergesort

→ siehe Sortieralgorithmen → Mergesort

Dynamische Programmierung

Fibonacci Beispiel **Iterativ**

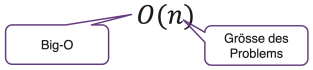
```
static void fibonacciIterativ(int n) {
    int num1 = 0
    int num2 = 1;
    int counter = 0;

    while (counter < n) {
        System.out.print(num1 + " ");

        int num3 = num2 + num1;
        num1 = num2;
        num2 = num3;
        counter++;
    }
}
```

Fibonacci Beispiel **Dynamisch**

```
static int fibonacciDynamisch(int n) {
    int f[] = new int[n + 2];

    int i;
    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++) {
        f[i] = f[i - 1] + f[i - 2];
    }

    return f[n];
}
```

Analyse von Algorithmen

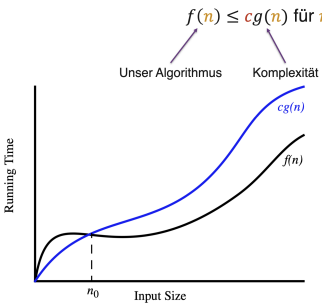
O-Notation / Big-O-Notation



$f(n)$ ist $O(g(n))$ falls

reelle, positive Konstante $c > 0$, Ganzzahl-Konstante $n_0 \geq 1$, so dass

$f(n) \leq cg(n)$ für $n \geq n_0$



- Regeln
  - Falls $f(n)$ Polynom vom Grad d ist, gilt $f(n) \in O(n^d)$, d.h:
    - Alle tieferen Potenzen weglassen
    - Alle Konstanten weglassen
  - Jeweils optimalste Funktion verwenden (tiefst mögliche Potenz)
  - So stark wie möglich vereinfachen
    - $3n + 5$ ist $O(n)$, statt $3n + 5$ ist $O(3n)$
- Häufige Funktionen
  - konstant ≈ 1
  - logarithmisch $\approx \log(n)$
  - linear $\approx n$
  - N-log-N $\approx n * \log(n)$
  - quadratisch $\approx n^2$
  - kubisch $\approx n^3$
  - exponentiell $\approx 2^n$
- big-O
  - $f(n)$ ist $O(g(n))$ falls $f(n)$ asymptomatisch **kleiner oder gleich** $g(n)$ ist: $f(n) \leq cg(n)$ für $n \geq n_0$
- big-Omega
  - $f(n)$ ist $\Omega(g(n))$ falls $f(n)$ asymptomatisch **grösser oder gleich** $g(n)$ ist: $f(n) \geq c * g(n)$ für $n \geq n_0$
- big-Theta
  - $f(n)$ ist $\Theta(g(n))$ falls $f(n)$ asymptomatisch **gleich** $g(n)$ ist: $c'g(n) \leq f(n) \leq c''g(n)$ für $n \geq n_0$

Laufzeit-Beweis Beispiel

Beweisen Sie: $8n^4 + 3n^2 + 4$ ist $O(n^4)$

Gesucht: $c > 0$ und $n_0 \geq 1$, so dass $8n^4 + 3n^2 + 4 \leq cn^4$ für $n \geq n_0$

$8n^4 + 3n^2 + 4 \leq cn^4$
$\Leftrightarrow 3n^2 + 4 \leq cn^4 - 8n^4$
$\Leftrightarrow 3n^2 + 4 \leq n^4(c - 8)$
$\Leftrightarrow \frac{3n^2+4}{c-8} \leq n^4$
$\Rightarrow c = 9$
$\Leftrightarrow 3n^2 + 4 \leq n^4$
$\Leftrightarrow 4 \leq n^4 - 3n^2$
$\Rightarrow n = 2$

Operationen Zählen

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 + 2*n*
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: $2 + (1 + 2n) + 2(n-1) + 0 + 2(n-1) + 1 = 6n$

Empirie / Empirische Analyse

Ablauf Empirie

1. Algorithmus implementieren
2. Algorithmus mit unterschiedlichen Eingaben ausführen
3. Ergebnisse aufzeichnen und vergleichen

```
long startTime = System.currentTimeMillis()
/* Algorithmus */
long endTime = System.currentTimeMillis();
long elapsed = endTime - startTime;
```

Herausforderungen: Empirischer Vergleich von Algorithmen

- Datengrundlage (Testdaten) sollte repräsentativ sein
- Abhängigkeiten vom System → z.B Memory, CPU-Leistung, etc.

Wichtig: Abgrenzung Empirische Analyse, Theoretische Analyse mit O-Notation

Sortieralgorithmen

- Elemente mit übergebener Funktion in logische Ordnung bringen

Swap-Methode für Algorithmen

```
private static void swap(int[] array, int i, int j) {
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

Bogosort

```
void bogo(int[] arr) {
    int shuffle = 1;
    for(; !isSorted(arr); shuffle++)
        shuffle(arr);
}

void shuffle(int[] arr) {
    int i = arr.length - 1;
    while (i > 0) {
        swap (arr, i--, (int) (Math.random() * i));
}
```

Worst Case: *Infinite*
Average Case: $n * n!$
Best Case: n

Insertionsort

- Elementarer Sortieralgorithmus
- Eignet sich für kleinere Datenmengen oder für Einfügen in eine bereits sortierte Liste

```
public static void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0 && less(a[j], a[j - 1]); j--) {
            swap(a, j, j - 1);
        }
    }
}
```

Worst Case: $O(n^2)$
Average Case: $O(n^2)$
Best Case: $O(n)$ (Wenn Liste bereits sortiert oder fast alle Objekte sind etwa am richtigen Ort)

Selection Sort

- Elementarer Sortieralgorithmus

```
static void selectionSort(int[] array) {
    int marker = array.length - 1;
    while (marker >= 0) {
        int max = 0;
        for (int i = 1; i <= marker; i++) {
            if (array[i] > array[max]) {
                max = i;
            }
        }
        swap(array, marker, max);
        marker--;
    }
}
```

Worst Case: $O(n^2)$
Average Case: $O(n^2)$
Best Case: $O(n^2)$

Bubblesort

```
bubbleSort(Array a) {
    for (n = a.size; n > 1; n--) {
        for (i = 0; i < n - 1; i++) {
            if (a[i] > a[i + 1]) {
                a.swap(i, i + 1);
            }
        }
    }
}
```

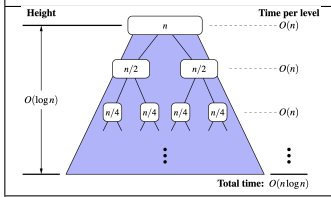
Worst Case: $O(n^2)$
Average Case: $O(n^2)$
Best Case: $O(n)$ (Wenn Array bereits nach Sortierkriterium sortiert ist)

Mergesort

- Divide and Conquer (Teile und Herrsche)
- Rekursiver Sortieralgorithmus
- Zusätzlicher Speicherbedarf beim Merge
- Quicksort überlegen, aber benötigt viel Speicherplatz

```
int[] merge(int[] leftArray, int[] rightArray) {
    int targetPos = 0; int leftPos = 0; int rightPos = 0;
    while (leftPos < leftLen && rightPos < rightLen) {
        int leftValue = leftArray[leftPos];
        int rightValue = rightArray[rightPos];
        if (leftValue <= rightValue) {
            target[targetPos++] = leftValue;
            leftPos++;
        } else {
            target[targetPos++] = rightValue;
            rightPos++;
        }
    }
    while (leftPos < leftLen) {
        target[targetPos++] = leftArray[leftPos++];
    }
    while (rightPos < rightLen) {
        target[targetPos++] = rightArray[rightPos++];
    }
    return target;
}
```

```
private int[] mergeSort(int[] elements, int left, int right) {
    if (left == right)
        return new int[] {elements[left]};

    int middle = left + (right - left) / 2;
    int[] leftArray = mergeSort(elements, left, middle);
    int[] rightArray = mergeSort(elements, middle + 1, right);
    return merge(leftArray, rightArray);
}
```



Worst Case: $O(n * log(n))$
Average Case: $O(n * log(n))$
Best Case: $O(n * log(n))$

Quicksort

- Kein zusätzlicher Speicherbedarf

1. Pivot wählen
2. Array in 2 Unter-Arrays teilen → Elemente kleiner als Pivot und Elemente grösser als Pivot
3. Quicksort rekursiv auf beiden Unter-Arrays aufrufen
4. Elemente des ersten Unter-Array, Pivot und Elemente des zweiten Unter-Arrays zurückgeben

Nico Fehr | Ostschweizer Fachhochschule

Idee

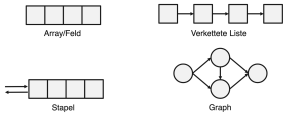
```
def quicksort(array):
    if len(array) < 2:
        return array
    else:
        pivot = array[0]
        less = [i for i in array[1:] if i <= pivot]
        grater = [i for i in array[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

```
public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quicksort(arr, low, pi - 1);
        quicksort(arr, pi + 1, high);
    }
}

public static int partition(int[] arr, int start, int end) {
    int pivotElement = arr[start];
    int pivotIndex = start + 1;
    for (int i = start + 1; i < end; ++i) {
        if (arr[i] <= pivotElement) {
            swap(arr, i, pivotIndex);
            ++pivotIndex;
        }
    }
    swap(arr, start, pivotIndex - 1);
    return pivotIndex - 1;
}
```

Worst Case: $O(n^2)$ (Wenn das Pivot Element immer das letzte Element einer Liste ist und die Liste eigentlich schon sortiert ist und die Liste eigentlich schon sortiert ist. Die Teillisten würden nur um 1 kleiner werden bei der Rekursion. Im besten Fall wählt man Pivot so, dass die Teillisten links und rechts ungefähr gleich gross sind.)
Average Case: $O(n * log(n))$
Best Case: $O(n * log(n))$

Datenstruktur

Beispiele für Datenstrukturen



ADT (abstract data type)

Unterschied zur **Datenstruktur**:

- abstrakter Datentyp (ADT) → eine Abstraktion einer konkreten Datenstruktur
- Datenstruktur → Realisierung / Implementierung eines ADT
- Datenstruktur → Speichern und organisieren Daten

Definition

- ADT beschreibt
  - Attribute
  - Operation auf Attributen
  - Ausnahmen und Fehler
- Beschreibung des **Was**, aber nicht des **Wie**

LinkedList<E>

`void add(E item)`	item am Ende einfügen
`void add(int pos, E item)`	item an Position pos einfügen
`boolean contains(E item)`	Enthält Liste item?
`int size()`	Gibt Anzahl items in Liste zurück
`boolean isEmpty()`	Ist Liste leer?
`E get(int pos)`	Element an Position pos zurückgeben
`E remove(int pos)`	Element an Position pos entfernen und zurückgeben

Doubly-Linked-List

- Jeder Knoten speichert Verbindung zum Vorgänger & Nachfolger
- header & trailer als Start-Knoten für Suche
  - header & trailer: Sentinels / Guards

Stack

- Anwendung: Parser / Parsen von verschachtelten Klammern (())([()])

```
public interface Stack<E> {
    int size();
    boolean isEmpty();
    void push(E element);
    E top();
    E pop();
}
```

`resize()` statt Exception

```
public void push(Item item) {
    if (n == a.length) resize();
    a[++n] = item;
}

private void resize() {
    int oldSize = data.length;
    int newSize = oldSize * RESIZE_FACTOR;

    E[] temp = (E[]) new Object[newSize];
    for (int i = 0; i < oldSize; i++) {
        temp[i] = data[i];
    }
    data = temp;
}
public class ArrayStack<E> implements Stack<E> {
    private E[] data;
    private int t = -1;

    public ArrayStack(int capacity){
        data = (E[]) new Object[capacity];
    }

    public int size() {
        return (t + 1);
    }

    public boolean isEmpty() {
        return (t == -1);
    }

    public void push(E element) throws IllegalStateException {
        if (size() == data.length)
            throw new IllegalStateException ("Stack is full!");
        data[++t] = element;
    }

    public E top() {
        if (isEmpty()) return null;
        return data[t];
    }

    public E pop() {
        if (isEmpty()) return null;
        E element = data[t];
        data[t--] = null;
        return element;
    }
}
```

Queue

```
public interface Queue<E> {
    int size();
    boolean isEmpty();
    void enqueue(E element);
    E first();
    E dequeue();
}
...
public void enqueue(E element) {
    if (this.storedElements == this.capacity) {
        throw new IllegalStateException();
    } else {
        int r = (this.frontElement + this.storedElements) %
                this.capacity;
        this.data[r] = element;
        this.storedElements++;
    }
}
...
public E dequeue() {
    if (this.isEmpty()) {
        return null;
    } else {
        E elem = this.data[this.frontElement];
        this.frontElement = (this.frontElement + 1) %
                this.capacity;
        this.storedElements--;
        return elem;
    }
}
```

Priority Queue

Entry und Priority Queue ADT

```
public interface PriorityQueue<K, V> {
    int size();
    boolean isEmpty();
    Entry<K, V> insert(K key, V value);
    Entry<K, V> min();
    Entry<K, V> removeMin();
}

public interface Entry<K,V> {
    public K key();
    public V value();
}
```

Impl mit **unsortierter** Liste: `insert()`

```
@Override
public Entry<K,V> insert(K key, V value) {
    checkKey(key);
    Entry<K,V> newest = new PriorityQueueEntry<>(key, value);
    list.add(newest);
    return newest;
}
```

Impl mit **unsortierter** Liste: `removeMin()` Impl mit **unsortierter** Liste: `removeMin()`

```
@Override
public Entry<K,V> removeMin() {
    if (list.isEmpty()) {
        return null;
    }
    var entry = findMin();
    list.remove(entry);
    return entry;
}

private Entry<K,V> findMin() {
    Entry<K,V> small = list.get(0);
    for (Entry<K,V> walk : list) {
        if (compare(walk, small) < 0) {
            small = walk;
        }
    }
    return small;
}
```

Impl mit **sortierter** Liste: `insert()`

```
public Entry<K, V> insert(K key, V value) {
    var newest = new PriorityQueueEntry<>(key, value);
    if (list.size() == 0) {
        list.add(newest);
        return newest;
    }
    Entry<K,V> walk = list.get(list.size() - 1);
    int index = 0;
    for (index = list.size() - 1; index >= 0 && compare(newest,
          walk) > 0; index--) {
        walk = list.get(index);
    }
    if (index == -1) {
        list.add(newest);
    } else {
        list.add(index, newest);
    }
    return newest;
}
```

Impl mit **sortierter** Liste: `removeMin()`

```
@Override
public Entry<K, V> removeMin() {
    return list.remove(0);
}
```

Priority Queue sortieren

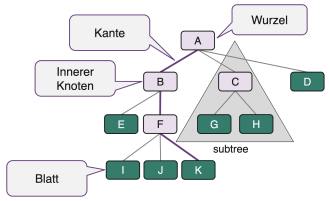
```
public static <E> void pqSort(ArrayList<E> sourceList,
      PriorityQueue<E, ?> pQueue) {
    int n = sourceList.size();
    for (int j = 0; j < n; j++) {
        E element = sourceList.remove(0);
        pQueue.insert(element, null);
    }
    for (int j = 0; j < n; j++) {
        E element = pQueue.removeMin().getKey();
        sourceList.add(element);
    }
}
```

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

Trees / Bäume

- Zweidimensionale Datenstruktur
- Repräsentiert hierarchische Beziehungen
- Besteht aus...
  - **Knoten:** Objekte des Baums
  - **Kanten:** Relationen zwischen Knoten

Terminologie



- **Wurzel:** Knoten ohne Elternknoten (A)
- **Innerer Knoten:** Knoten mit mind. einem Kind (A, B, C, F)
- **Blatt:** Knoten ohne Kinder (E, I, J, K, usw.)
- **Vorgängerknoten:** Eltern, Grosseltern, ...
- **Geschwister:** Knoten mit selben Eltern

Tiefe

- **Tiefe** von v: Vorfahren von v (ohne v)

Tiefe: Implementierung

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

Binary Tree / Binärer Baum

- Innere Knoten: Höchstens 2 Kinder
- Kinder eines Knotens sind geordnetes Paar (links, rechts)

ADT in Java

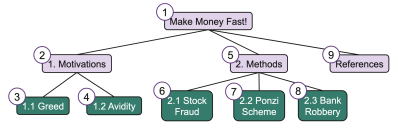
```
public interface BinaryTree<E> extends Tree<E> {
    Position<E> left(Position<E> p);
    Position<E> right(Position<E> p);
    Position<E> sibling(Position<E> p);
    Position<E> addRoot(E e);
    Position<E> addLeft(Position<E> p, E e);
    Position<E> addRight(Position<E> p, E e);
}
```

Traversierungen

- Knoten systematisch besuchen
- Algorithmen
  - Preorder
  - Postorder
  - Breadth-First
  - Inorder

Preorder (W-L-R)

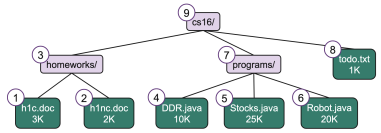
- Knoten wird **vor** seinen Kindern besucht




```
algorithm preOrder(v)
    visit(v)
    for each child w of v
        preOrder(w)
```

Postorder (L-R-W)

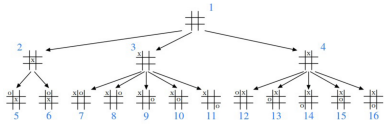
- Knoten wird **nach** seinen Nachfolgern besucht




```
algorithm postOrder(v)
    for each child w of v
        postOrder(w)
    visit(v)
```

Breadth-First

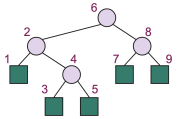
- **Alle Knoten einer Stufe** besuchen, bevor Nachfolgeknoten besucht werden



```
Algorithm breadthFirst()
//Initialize queue Q containing root
while Q not empty
    v = Q.dequeue()
    visit(v)
    for each child w in children(v)
        Q.enqueue(w)
```

Inorder (L-W-R)

- Knoten **nach** linken Subtree und **vor** rechtem Subtree besuchen
- Darstellung von binären Bäumen
  - $x(v)$ = inorder Rang von v
  - $y(v)$ = Tiefe von v



```
algorithm inOrder(v)
    if hasLeft(v)
        inOrder(left(v))
    visit(v)
    if hasRight(v)
        inOrder(right(v))
```

Auswertung von arithmetischen Ausdrücken / Euler-Tour

```
public int eulerPath(Node<E> node) {
    Result result = new Result();
    if (node == null) {
        return 0;
    }
    if (node.isLeaf()) {
        visitLeaf(node, result);
    } else {
        visitLeft(node, result);
        result.leftResult = eulerPath(node.getLeft());
        visitBelow(node, result);
        result.rightResult = eulerPath(node.getRight());
        visitRight(node, result);
    }
    return result.finalResult;
}
```

Heapsort

- Idee des Algorithmus
  - Wurzel enthält immer kleinstes Element
  - Wiederholt Wurzelelement entnehmen bis Array leer ist

Definition Heap

- Binärer Baum mit folgenden Eigenschaften
  - Baum ist vollständig
  - Schlüssel jedes Knoten kleiner oder gleich als Schlüssel seiner Kinder

Algorithmus

```
algorithm HeapSort(F)
    Eingabe: zu sortierende Folge F der Laenge n

    Ueberfuehre F in Heap;
    l := n; /* Letztes Element */
    while l > 1 do
        Vertausche F[1] und F[l];
        Versenke F[1] im Heap F[1 ... l - 1];
        l := l - 1
    od
```

Java Implementation

```
public static void percolate(Comparable[] arrayToSort, int
      startIndex, int last) {
    int i = startIndex;
    while(hasLeftChild(i, last)) {
        int leftChild = getLeftChild(i);
        int rightChild = getRightChild(i);
        int exchangeWith = 0;

        if(arrayToSort[i].compareTo(arrayToSort[leftChild]) > 0) {
            exchangeWith = leftChild;
        }
        if(rightChild <= last && arrayToSort[leftChild]
                .compareTo(arrayToSort[rightChild]) > 0) {
            exchangeWith = rightChild;
        }
        if (exchangeWith == 0 ||
              arrayToSort[i].compareTo(arrayToSort[exchangeWith])
              <= 0) {
            break;
        }
        swap(arrayToSort, i, exchangeWith);
        i = exchangeWith;
    }
}

public static void heapSort(Comparable[] arrayToSort) {
    int i;
    heapifyMe(arrayToSort);
    for (i = arrayToSort.length - 1; i > 0; i--) {
        swap(arrayToSort, 0, i); // Erstes Element mit letztem
              Tauschen
        percolate(arrayToSort, 0, i - 1); // Heap wiederherstellen
    }
}
```

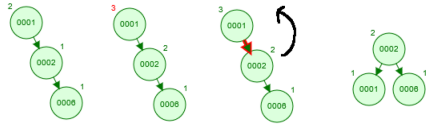
BinarySearchTree - $O(\log(n))$

Rotations

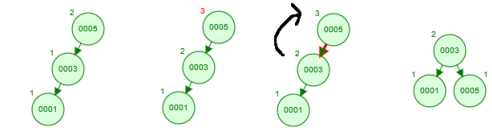
Rotationen erklärt

Hinweis: - `L Rotation -> RR Rotation - R Rotation -> LL Rotation`

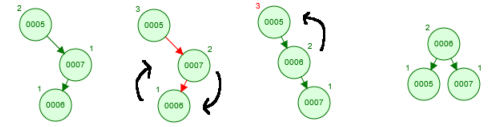
RR Rotation
Eine Drehung gegen den Uhrzeigersinn auf die untere Kante des Knoten, der nicht balanciert ist.



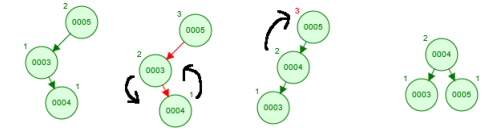
LL Rotation
Eine Drehung im Uhrzeigersinn auf die untere Kante des Knoten, der nicht balanciert ist.



RL Rotation
RL Rotation = LL Rotation + RR Rotation
Da in diesem Fall der Wert 7 grösser ist, als der Wert 6 muss zuerst eine `LL Rotation` auf der unteren Kante des Knoten 7 durchgeführt werden. Danach hat man wieder die Ausgangslage einer `RR Rotation`.



LR Rotation
LR Rotation = RR Rotation + LL Rotation
Da in diesem Fall der Wert 4 grösser ist, als der Wert 3 muss zuerst eine `RR Rotation` auf der unteren Kante des Knoten 4 durchgeführt werden. Danach hat man wieder die Ausgangslage einer `LL Rotation`.



```
package ch.ost.oop.ex.task4;

import java.util.Iterator;
import java.util.List;
import java.util.Random;

import static java.lang.Math.max;

public class BinarySearchTree<E extends Comparable<E>> implements
       Iterable<E> {

    private Node<E> root;

    public void insertElement(E value) {
        root = insertElement(value, root);
    }

    private Node<E> insertElement(E value, Node<E> node) {
        if (node == null) {
            node = new Node<>(value);
        } else if (value.compareTo(node.getValue()) < 0) {
            node.setLeft(insertElement(value, node.getLeft()));
            if (getHeight(node.getLeft()) -
                getHeight(node.getRight()) == 2) {
                if (value.compareTo(node.getLeft().getValue()) <
                      0) {
                    node = rotateWithLeftChild(node);
                } else {
                    node = doubleWithLeftChild(node);
                }
            }
        } else if (value.compareTo(node.getValue()) > 0) {
            node.setRight(insertElement(value, node.getRight()));
            if (getHeight(node.getRight()) -
                getHeight(node.getLeft()) == 2) {
                if (value.compareTo(node.getRight().getValue()) >
                      0) {
                    node = rotateWithRightChild(node);
                } else {
                    node = doubleWithRightChild(node);
                }
            }
        }
        node.setHeight(getMaxHeight(getHeight(node.getLeft()),
              getHeight(node.getRight())) + 1);

        return node;

    }

    // (LR) Links-Rechts-Rotation
    private Node<E> rotateWithLeftChild(Node<E> node2) {
        Node<E> node1 = node2.getLeft();
        node2.setLeft(node1.getRight());

        node1.setRight(node2);

        node2.setHeight(getMaxHeight(getHeight(node2.getLeft()),
              getHeight(node2.getRight())) + 1);
        node1.setHeight(getMaxHeight(getHeight(node1.getLeft()),
              node2.getHeight()) + 1);

        return node1;
    }

    // (RL) Rechts-Links-Rotation
    private Node<E> rotateWithRightChild(Node<E> node1) {
        Node<E> node2 = node1.getRight();
        node1.setRight(node2.getLeft());

        node2.setLeft(node1);

        node1.setHeight(getMaxHeight(getHeight(node1.getLeft()),
              getHeight(node1.getRight())) + 1);
        node2.setHeight(getMaxHeight(getHeight(node2.getRight()),
              node1.getHeight()) + 1);

        return node2;
    }

    // (LL) Links-Links-Rotation
    private Node<E> doubleWithLeftChild(Node<E> node3) {
        node3.setLeft(rotateWithRightChild(node3.getLeft()));

        return rotateWithLeftChild(node3);
    }

    // (RR) Rechts-Rechts-Rotation
    private Node<E> doubleWithRightChild(Node<E> node1) {
        node1.setRight(rotateWithLeftChild(node1.getRight()));
        return rotateWithRightChild(node1);
    }

    private int getHeight(Node<E> node) {
        return node == null ? -1 : node.getHeight();
    }

    private int getMaxHeight(int leftNodeHeight, int
          rightNodeHeight) {
        return max(leftNodeHeight, rightNodeHeight);
    }

    public int size() {
        return size(root);
    }

    private int size(Node<E> head) {
        if (head == null) return 0;
        else {
            int length = 1;
            length += size(head.getLeft());
            length += size(head.getRight());
            return length;
        }
    }

    public Node<E> find(E value) {
        Node<E> current = root;
        while (current != null) {
            if (current.getValue().compareTo(value) == 0) {
                break;
            }
            current = current.getValue().compareTo(value) < 0 ?
                  current.getRight() : current.getLeft();
        }
        return current;
    }

    public Node<E> getRoot() {
        return root;
    }


    @Override
    public Iterator<E> iterator() {
        return new TreeIteratorPreorder<>(root);
    }


    public static void main(String[] args) {

        BinarySearchTree<Integer> bst = new BinarySearchTree<>();

        List<Integer> list = List.of(21, 11, 5, 1, 0, 4, 9, 10,
              13, 12, 18, 15, 20, 30, 25, 23, 22, 28, 43, 42,
              34, 47, 44, 48);
        for (Integer integer : list) {
            bst.insertElement(integer);
        }

        for (Integer integer : bst) {
            System.out.println(integer);
        }
    }

}
```

Node

```
package ch.ost.oop.ex.task4;

class Node<E> {
    private final E value;
    private Node<E> left, right;
    private int height;

    public Node(E value) {
        this.value = value;
    }

    public E getValue() {
        return value;
    }

    public Node<E> getLeft() {
        return left;
    }

    public void setLeft(Node<E> left) {
        this.left = left;
    }
```

```
    public Node<E> getRight() {
        return right;
    }

    public void setRight(Node<E> right) {
        this.right = right;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }
}
```

TreeIteratorPreorder

```
package ch.ost.oop.ex.task4;

import java.util.Iterator;
import java.util.Stack;

public class TreeIteratorPreorder<T> implements Iterator<T> {

    private Node<T> root;
    private final Stack<Node<T>> visiting;

    public TreeIteratorPreorder(Node<T> root) {
        visiting = new Stack<>();
        this.root = root;
    }

    @Override
    public boolean hasNext() {
        return root != null;
    }

    @Override
    public T next() {
        if (visiting.empty()) {
            visiting.push(root);
        }
        Node<T> node = visiting.pop();

        if (node.getRight() != null) {
            visiting.push(node.getRight());
        }
        if (node.getLeft() != null) {
            visiting.push(node.getLeft());
        }

        if (visiting.empty()) {
            root = null;
        }
        return node.getValue();
    }
}
```

Hashing

Eigenschaften guter Hashfunktionen

- Konsistenz (gleicher Input → gleicher Output)
- Effiziente Berechnung
- Gleichmässige Verteilung der Schlüssel

Integer Cast

- Gut → Anzahl bits erlaubt Interpretation als Integer
- Schlecht → Schlüssel ist länger

Komponentensumme

- Bits des Keys in Komponenten fixer länge (16/32 bits) unterteilen
  - Komponenten summieren, Overflow ignorieren
- Gut für Schlüssel fixer Länge, >= Anzahl bits von Integer

Polynom-Akkumulation

- Hashing von Werten der Form $(x_0, x_1, \ldots, x_{n-1})$
- Polynom
  - $p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}$
  - für fixes z
- Gut für Strings
  - Mit $z = 33$ max. 6 Kollisionen bei 50.000 englischen Wörtern
- `String.hashCode()`
  - `s[0] * 31 ^ (n - 1) + s[1] * 31 ^ (n - 2) + ... + s[n - 1]`

Kollisionen (inkl. Behandlung)

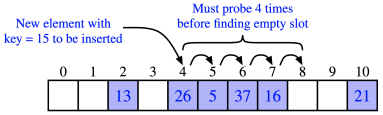
Geschlossene Addressierung

- Behälter sind (verkettete) Listen
- Platz nicht begrenzt, prinzipiell keine Überläufer

Separate Chaining

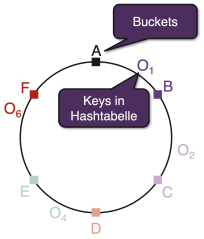
- Jede Zelle der Tabelle zeigt auf Liste
- Gut: Einfach
- Schlecht: Zusätzliche Datenstruktur und Speicherbedarf
- Lineare Suche im Bucket: $O\frac{n}{N}$, n = Einträge in Tabelle, N = Buckets

Offene Addressierung



- Für Überläufer in anderen Behältern Platz suchen (Sondierung)
- Sondierungsfolge bestimmt Weg zum Speichern & Finden der Überläufer
- erfordert besondere Behandlung der Lösch-Operationen
- Wird element gelöscht, kann Sondierungsfolge für anderen Datensatz unterbrechen
- Beim Sondieren werden Behälter mit gelöschten Datensätzen wie belegte Behälter zum weiterhangeln benutzt

Konsistentes Hashing



- Keys werden auf Ring angeordnet
- im Uhrzeigersinn werden Keys dem nächsten Bucket zugeordnet
- Änderungen wirken sich nur auf direkte Nachbarn aus

Hashing in Java

- Wenn für Klasse `equals()` überschrieben wird muss auch `hashCode()` überschrieben werden

IntelliJ IDEA Standard `hashCode()` Implementierung

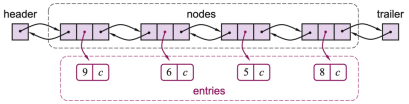
```
@Override
public int hashCode() {
    int result = (int) (id ^ (id >>> 32));
    result = 31 * result + name.hashCode();
    result = 31 * result + email.hashCode();
    return result;
}
```

Gleichheit

- Referenzvergleich: `equals()` standardmässig Referenzvergleich (referential equality)
  - `a.equals(b)` vs `a == b`
- Inhaltsvergleich: `equals()` überschreiben (structural equality)
  - Bei String implementiert (bei anderen Datentypen nicht!)

Maps

Map mit Unsortierter Liste



Get

```
public V get(K key) {
    var iter = this.list.iterator();
    while (iter.hasNext()) {
        var node = iter.next();
        if (node.getKey().equals(key)) {
            return node.getValue();
        }
    }
    return null;
}
```

Put

```
public V put(K key, V value) {
    var iter = this.list.iterator();
    while (iter.hasNext()) {
        var node = iter.next();
        if (node.getKey().equals(key)) {
            V t = node.getValue();
            node.setValue(value);
            return t;
        }
    }
    this.list.addLast(new ListNode<K, V>(key, value));
    return null;
}
```

Remove

```
public V remove(K key) {
var iter = this.list.iterator();
    while (iter.hasNext()) {
        var node = iter.next();
        if (node.getKey().equals(key)) {
            V val = node.getValue();
            list.remove(node);
            return val;
        }
    }
    return null;
}
```

Multimap

- Ähnlich aufgebaut wie Map: speichert <k, v>-Paare
- Multimap kann zu Schlüssel k mehrere Einträge aufnehmen
  - <k, v> und <k, v'>
    - Schloss → Gebäude
    - Schloss → Verriegelung
- Lösungsansatz
  - Ansatz 1: Anpassen der Datenstruktur
  - Ansatz 2: Schlüssel k verweist auf Collection mit Werten k

Interface

```
package ch.ost.oop2.hashmultimap;
import java.util.Map;

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();
}
```

Multimap Impl

```
public class HashMultimap<K, V> implements Multimap<K, V> {
    Map <K, List<V>> map = new HashMap<>();

    int numberOfEntries = 0;
    public HashMultimap() {};

    @Override
    public int size() {
        return numberOfEntries;
    }

    @Override
    public boolean isEmpty() {
        return (numberOfEntries == 0);
    }
    // ...
}
```

Put

```
@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++;
}
```

Get

```
@Override
public Iterable<V> get (K key) {
    List<V> secondary = map.get(key);
    if (secondary != null) {
        return secondary;
    }
    return new ArrayList<>();
}
```

Remove

```
@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;
}
```

SeparateChainingMap

```
package ch.ost.oop.ex12.task1;
import java.util.*;

public class SeparatChainingMap<K, V> {
    List<Entry<K, V>> entries;
    int maxSize;
    int size;

    public SeparatChainingMap() {
        maxSize = 10;
        this.entries = new ArrayList<>();
        for (int i = 0; i < maxSize; i++) {
            entries.add(null);
        }
    }

    private int getListIndex(K key) {
        return Math.abs(Objects.hashCode(key) % maxSize);
    }

    public V get(K key) {
        Entry<K, V> entry = getEntry(key);
        if (entry == null) {
            return null;
        }
        return entry.getValue();
    }

    private Entry<K, V> getEntry(K key) {
        int index = getListIndex(key);
        Entry<K, V> current = entries.get(index);
        while (current != null) {
            if (current.getKey().equals(key)) {
                return current;
            }
            current = current.getNext();
        }

        return null;
    }

    public V put(K key, V value) {
        int index = getListIndex(key);
        Entry<K, V> newEntry = new Entry<>(key, value);
        Entry<K, V> current = entries.get(index);
        if (current == null) {
            entries.set(index, newEntry);
            size++;
        } else {
            while (current != null) {
                if (current.getKey().equals(key)) {
                    V before = current.getValue();
                    current.setValue(value);
                    expandTable();
                    return before;
                }
                current = current.getNext();
            }
            current = entries.get(index);
            newEntry.setNext(current);
            entries.set(index, newEntry);
            size++;
        }
        expandTable();
```

```
        return null;
    }

    private void expandTable() {
        if (1.0d * size / maxSize > 0.7) {
            List<Entry<K, V>> tmp = entries;
            entries = new ArrayList<>();
            maxSize *= 2;
            for (int i = 0; i < maxSize; i++) {
                entries.add(null);
            }
            for (Entry<K, V> entry : tmp) {
                while (entry != null) {
                    put(entry.getKey(), entry.getValue());
                    entry = entry.getNext();
                }
            }
        }
    }

    public V remove(K key) {
        int index = getListIndex(key);
        Entry<K, V> current = entries.get(index);
        if (current == null) {
            return null;
        }
        if (current.getKey().equals(key)) {
            V val = current.getValue();
            current = current.getNext();
            entries.set(index, current);
            size--;
            return val;
        } else {
            Entry<K, V> prev = null;
            while (current != null) {

                if (current.getKey().equals(key)) {
                    prev.setNext(current.getNext());
                    size--;
                    return current.getValue();
                }
                prev = current;
                current = current.getNext();
            }
            return null;
        }
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public Set<K> keySet() {
        Set<K> set = new HashSet<>();
        for (Entry<K, V> entry : entries) {
            if (entry != null) {
                Entry<K, V> current = entry;
                while (current != null) {
                    set.add(current.getKey());
                    current = current.getNext();
                }
            }
        }
        return set;
    }

    public Collection<V> values() {
        List<V> list = new ArrayList<>();
        for (Entry<K, V> entry : entries) {
            if (entry != null) {
                Entry<K, V> current = entry;
                while (current != null) {
                    list.add(current.getValue());
                    current = current.getNext();
                }
            }
        }
        return list;
    }

    public Collection<Entry<K, V>> entrySet() {
        List<Entry<K, V>> list = new ArrayList<>();
        for (Entry<K, V> entry : entries) {
            if (entry != null) {
                Entry<K, V> current = entry;
                while (current != null) {
                    list.add(current);
                    current = current.getNext();
                }
            }
        }
        return list;
    }

    public void print() {
        for (Entry<K, V> entry : entries) {
            if (entry == null) {
                System.out.println("null");
                continue;
            }
            while (entry != null) {
                System.out.print(entry);
                System.out.print(" --> ");
                entry = entry.getNext();
            }
            System.out.println();
        }
    }
}
```

Entry

```
package ch.ost.oop.ex12.task1;
import java.util.Objects;

public class Entry<K, V> {
    private final K key;
    private V value;
    private Entry<K, V> next;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }

    public Entry<K, V> getNext() {
        return next;
    }

    public void setNext(Entry<K, V> next) {
        this.next = next;
    }

    @Override
    public String toString() {
        return "Entry{" +
                "key=" + key +
                ", value=" + value +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Entry<?, ?> entry = (Entry<?, ?>) o;
        return Objects.equals(key, entry.key) &&
               Objects.equals(value, entry.value);
    }

    @Override
    public int hashCode() {
        return Objects.hash(key, value);
    }
}
```

File-IO

FileReader

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

Systemabhängiges Character Set

-1 = end of file

16-bit char

`FileReader: new FileReader(f) ⇔ new InputStreamReader(new FileInputStream(f))`

FileWriter

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

Append

String schreiben

Einzelnen char schreiben

BufferedReader - Read Lines

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