[Object-Oriented design 1](#_Toc521759416)

[Array – Speicherplatz fix -> Statisch 1](#_Toc521759417)

[Linked-lists 1](#_Toc521759418)

[Folgen & Reihen + Summenformel 2](#_Toc521759419)

[aNALYSE VON aLGORITHMEN 2](#_Toc521759420)

[Rekursion 2](#_Toc521759421)

[Design patterns 3](#_Toc521759422)

[Stack – Stapel – LIFO Prinzip 3](#_Toc521759423)

[Queues 4](#_Toc521759424)

[List & Iterators 5](#_Toc521759425)

[Trees – bäume 6](#_Toc521759426)

[Iteratoren 8](#_Toc521759427)

[Priority Queues 9](#_Toc521759428)

[Heaps 10](#_Toc521759429)

[Adaptable Priority Queues 11](#_Toc521759430)

[Maps 12](#_Toc521759431)

[Hash Tabellen 13](#_Toc521759432)

[Skipliste 14](#_Toc521759433)

[Sets, multisets & multimaps 15](#_Toc521759434)

# Object-Oriented design

## OO-Design ziele

### Software soll sein:

* Robust
* Adaptierbar
* Wiederverwendbar

### oo-design Prinzipien

* Abstraktion
* Kapselung
* Modularität

### Klasse und Vererbung

* Klasse kapselt Daten, definiert Schnittstelle zur Benutzung von Objekten
* Methoden erlauben sicheren Zugriff(z.B Validierung)

🡪 **robuster**

* Vererbung unterstützt ***Wiederverwendbarkeit*** und ***Adaptierbarkeit***(Polymorphismus/Überschr. Von Meth.)

### Generische Klassen – Generics

* Generische Klassen unterschiedlich typisiert   
  🡪 ***Wiederverwendbarkeit***
* Falsche Anwendung zum Kompilationszeitpunkt(z.B Typ-Fehler – List<Integer>, kein casten nötig) 🡪***Robustheit***

### Algorithmen

* Rekursion, Divide-and-Coquer, Brute-Force, Greedy-Method, Dynamic-Programming

### Design Patterns

* Generische Lösung für typische Software Design Probleme
* Abstrakte Pattern auf konkretes Problem adaptiert
* Adapter, Iterator, Template, Composite, Decorator

### Overloading, Overriding, Dynamic Dispatch

|  |  |
| --- | --- |
| class Base {  void copyTo(Object other) {  // Method 1  }  void copyTo(Base other) {  // Method 2 | class Sub extends Base {  void copyTo(Base other) {  // Method 3  }  void copyTo(Sub other) {  // Method 4 |

|  |  |
| --- | --- |
| **überladene-Methoden**:  copyTo(Object) geerbt, copyTo(Base), copyTo(Sub) | b.copyTo(o); Methode 4  b.copyTo(b); Methode 3  b.copyTo(s); Methode 3 |
| **überschriebene Methoden:**  copyTo(Base) | s.copyTo(o); Methode 1  s.copyTo(b); Methode 3  s.copyTo(s); Methode 4 |
| ***1. Compiler: Overloading statischer(Methoden anhand Signatur)***  ***2. Laufzeitsystem: Ist die gewählte Methode überschrieben?(Dynamic Dipatch) 🡪 Methode anhand des dynamischen Types.*** | |

# Array – Speicherplatz fix -> Statisch

* Speicher für gleich(artige) Objekte 🡪 Vererbung beachten
* Referenz auf Objekte gespeichert, Achtung bei Ref-Änderung

### add Score

public void addScore(player newEntry) {

int score = newEntry.getScore();

if (numEntries < capacity ||

score > board[numEntries - 1].getScore()) {

if (numEntries < capacity) {

numEntries++;}

int j = numEntries - 1;

while (j > 0 && board[j - 1].getScore() < score) {

board[j] = board[j - 1];

j--;}

board[j] = newEntry;}

else {

System.out.println("There is no new highscore!"); }

### Remove Score

public player removeScore(int index) {

if (index < 0 || index >= numEntries) {

throw new IndexOutOfBoundsException("Wrong index!");}

player temp = board[index];

for (int j = index; j < numEntries - 1; j++) {

board[j] = board[j + 1];}

numEntries--;

board[numEntries] = null;

return temp;

### Insertion Sort – aufsteigende Sortierung

int length = data.length;

for (int k = 1; k < length; k++) {

int current = data[k];

int j = k;

while (j > 0 && data[j - 1] > current) {

data[j] = data[j - 1];

j--; }

data[j] = current;

### Hilfreiche funktionen

Arrays.equals(array, copy); // true;

Arrays.sort(copy); //sortieren

Arrays.equals(array, copy); //false

Arrays.toString(copy); //[4,5,23]

# Linked-lists

### Grundlagen

Einfach verkettete Liste. Sequenz aus Knoten.

Jeder Knoten besitzt:

* Ein Element 🡪 den Inhalt
* Ein Bild, das Shoji, Uhr enthält.

  Mit sehr hoher Zuverlässigkeit generierte BeschreibungEin Link zum nächsten Knoten

### Linked List Beispiel ohne Generics

public class MyNode {

private Element element;

private MyNode next;

public MyNode(Element e, MyNode n) {

element = e;

next = n; }

public class Element {

private String value;

private int nr;

### Linked List Beispiel Mit Generics

public class Node*<E>* {

private *E* element;

private Node<E> next; …

### Einfach verkettete Liste(Singly-linked-list)

* head als Einstiegspunkt
* Ein Bild, das Uhr, Objekt, Gerät enthält.

  Mit hoher Zuverlässigkeit generierte Beschreibungtail als letzter Knoten

#### Einfügen am head – O(1)

addFirst(v ) //Methodenaufruf

v.setNext(head) //neuere Kopf auf alten verlinken

head ← v //head zeigt nun auf alten head

size ← size + 1 //Anzahl der Knoten + 1

#### Einfügen am Tail – O(1)

addLast(v ) //Methodenaufruf

v.setNext(null) //Neuer Knoten Next = null

tail.setNext(v) //Alter Tail auf neuen verlinken

tail ← v //Neuer Tail setzten

size ← size + 1 //Anzahl der Knoten + 1

#### Entfernen des ersten Knotens – O(1)

removeFirst( )

if head == null //Throw EmptyListException

t ← head //Head tempotär speichern

head ← head.getNext( )//Head auf nächsten Node linken

t.setNext(null) //Alter Head aus Liste entfernen

size ← size ‐1 //Anzahl der Knoten - 1

#### Entfernen des Letzten Knotens – O(1)

removeLast( )

if head == null //Throw EmptyListException

t ← head //Head temporär speichern

head ← head.getNext( )//Head auf nächsten Node linken

t.setNext(null) //Alter Head aus Liste entfernen

size ← size ‐1 //Anzahl der Knoten - 1

while (next != null) {

System.*out*.print(next.getElement().toString());

next = next.getNext(); }

### Doubly-Linked-List

+ Man kann in beide Richtungen suchen(nicht wie bei SLL)

Ein Bild, das Objekt, Uhr enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung- Speicherbedarf – Jeder speichert next und previous Node

* Header und Trailer spezielle Knoten 🡪 ***Sentinels***
* Header und Trailer sind Start-Knoten für die Suche

public class DList {

protected int size;

protected DNode header,trailer; // sentinels … }

public class DNode {

protected String element;

protected DNode next, prev; … }

#### Einfügen des Ersten Knotens – O(1)

addFirst(*v* ) //Methodenaufruf

*w* ← header.getNext() //alter erster Knoten merken

*v*.setNext(*w* ) //alter 1. Node als next setzen

*v*.setPrev(header) //header als previous setzen

w.setPrev(v ) //neuer Knoten als previous

header.setNext(*v* ) //erster Node next bei header

size = size +1 //Anzahl Nodes um eins erhöhen

#### Ein Bild, das Objekt enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungBestimmter Knoten Einfügen – O(1)

addAfter(*u*, *v* ) //v nach u einfügen

*w* ← *u*.getNext( ) //Node nach u speichern

*v*.setPrev(*u* ) // u als previous setzen

*v*.setNext(*w* ) // w als next setzen

*w*.setPrev(*v* ) // v previous von w setzen

*u*.setNext(*v* ) // v next von u setzen

size = size +1 //Anzahl Nodes um eins erhöhen

#### Entfernen des letzten Knoten – O(1)

removeLast() //Methodenaufruf

if size == 0 //Throw EmptyListException

*v* ← trailer.getPrev() //letzter Node in v speichern

*u* ← *v.*getPrev) //vorletzter Node in u

trailer.setPrev(*u*) //previous auf vorletzer Knoten

*u*.setNext(trailer) //trailer als next bei vorletzem

*v*.setPrev(null) //prev. bei letztem Element=null

*v*.setNext(null) //v ganz aus Liste auslinken

size = size ‐1 //Anzahl Nodes dekrementieren

#### Ein Bild, das Objekt enthält. Mit hoher Zuverlässigkeit generierte BeschreibungEntfernen eines Beliebigen Knotens – O(1)

remove(v) //Methodenaufruf

*u* ← *v*.getPrev() //previous in u speichern

w ← *v*.getNext() //next in w speichern

*w*.setPrev(*u* ) //previous von w auf u umhängen

*u*.setNext(*w* ) //next von u auf w umhängen

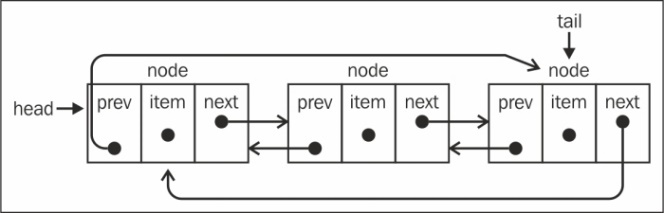
*v*.setPrev(null) //v previous auf null setzen

*v*.setNext(null) //v next auf null(komplett entf.)

size = size ‐1 //Anzahl Nodes dekrementieren

### Circularly-Linked-Lists

* Letzer Knoten ist mit erstem verbunden



# Folgen & Reihen + Summenformel

## Ein Bild, das Gerät enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungArithmetische Folgen

Wobei Explizit: an = a1 +d(n-1)

### BSP: 6-10-14-18

* **Rekursiv:**  an = an-1 + 4 a1=6
* **Iterativ:**
* **Explizit:**  an = 6 + 4(n-1) = 6 +4n -4 = 4n + 2

## Arithmetische Reihen / Summenformeln

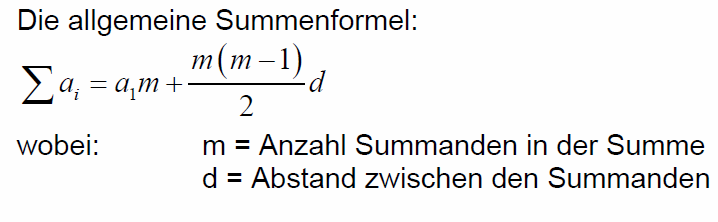
Für an explizite Formel(arithmetische Reihe) einfügen.

* **Rekursiv:** ***s1 = Anfangszahl, sn = sn-1 + an***
* **Iterativ**:  **// i als Index verwenden!**
* **Explizit:**

### BSP: 6-10-14-18

* **Rekursiv:** ***s1 = 6, sn = sn-1 + an = sn-1 + 4n + 2***
* **Iterativ**:
* **Explizit: = = 2n2 + 4n**

## Ein Bild, das Screenshot enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungAllgemeine Summenformel Für Big o



# Analyse von Algorithmen

Da durchschnittliches Verhalten oft schwierig zu bestimmen ist, konzentriert man sich auf das schlechteste Verhalten ***worst case***.

### Theoretische analyse / pseudo code

Alle möglichen Eingaben berücksichtigt, unabhängig von HW/SW

### Wichtige funktionen

|  |  |
| --- | --- |
| * **Konstant ≈ 1** | * **N-Log-N ≈ n log n** |
| * **Linear ≈ n** | * **Qubisch ≈ n3** |
| * **Quadratisch ≈ n2** | * **Exponentiell ≈ 2n** |
| * **Logarithmisch ≈ log n** |  |

### Ein Bild, das Shoji, Kreuzworträtsel enthält. Mit hoher Zuverlässigkeit generierte BeschreibungBenötigte Mathematik

### Laufzeitverhalten

Das Laufzeitverhalten ist nicht beeinflusst von:

* Konstanten Faktoren
* Tieferen Potenzen(die höchste Potenz überwiegt)

#### Beispiele

100n + 1000 🡪 lineare Funktion

10n2 + 100n + 1000 🡪 quadratische Funktion

#### Ein Bild, das Text enthält. Mit hoher Zuverlässigkeit generierte BeschreibungBig-oh notation

##### Beispiele Big-oh Notation

|  |  |
| --- | --- |
| * 2n + 10 ist O(n) | * 2n2 + 4n + 3 ist O(n2) |
| * 3n3 + 2n2 + n ist O(n3) | * 2 log n + 4 ist O(log n) |

##### Regeln

1. Polynom vom Grad d ist O(nd)
2. Tiefst mögliche Potenz verwenden
3. So stark wie möglich vereinfachen – Konstanten weglassen

##### Asymptotische Algorithmus Analyse

Die asymptotische Analyse eines Algorithmus bestimmt das

Laufzeitverhalten in der big-Oh Notation.

1. Worst case bestimmen
2. Der Algorithmus «Testalg» läuft in O(n) Zeit / hat ein O(n) Zeitverhalten

Ein Bild, das Text enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

### Arithmetische Progression (n2 + n) / 2

Ein Bild, das Text, Karte enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

# Rekursion

**Rekursion**: eine Methode ruft sich selbst auf.

Achtung: Baut evtl. viele Call Stacks auf!

Methodenaufrufe sind teuer!

Unter Umständen müssen weitere Parameter definiert werden

## Inhalte

### Verankerung(base case)

Wert der aktuellen Parameter, für die kein rekursiver Aufruf ausgeführt wird.  
Man spricht auch oft von ***Abbruchbedingung***.

### Rekursive Aufrufe

Rufen die aktuelle Methode wieder auf, soll so definiert sein, dass er die Ausführung ***in Richtung base case*** bewegt.

### Beispiel Fakultät Rekursiv

public static int recursiveFactorial(int n) {

if (n == 0) // base case

{

return 1;

}

else {

return n \* *recursiveFactorial*(n - 1)}}

### Divide & Conquer – Teile und Herrsche

Lösungsansatz: Ein Problem so lange in einfache ***Teilprobleme*** zu zerlegen, bis man auf lösbare Probleme stösst.

|  |  |
| --- | --- |
| * Fibonacci | * Sortieren (z.B. Quicksort) |
| * Finden eines Elements in einer sortierten Liste | * Euklid'scher Algorithmus |
| * "English Ruler" | * Tower of Hanoi |
| * Koch Kurve |  |

#### Beispiel – max char

public static char maximum(char[] w, int s, int f) {

if(s==f) return w[s]; //Base Case

int m = (s+f)/2; //Rekursion: teile

char c1 = maximum(w, s, m); //max in linker Hälfte

char c2 = maximum(w, m+1, f); //max rechter Hälfte

//Zusammenfügen: herrsche

if(c1<c2) return c2; //max über alles returnt

return c1;

}

Oder Rekursives Quadrieren(O(log(n))

### Endrekursion – tail recursion

Der **letzte Schritt** einer rekursiven Methode ist der rekursive Aufruf. Können leicht in Iterationen umgewandelt werden. Rechen

intensiv, da bei jedem Aufruf ein Stack-Frame initialisiert wird.

#### Beispiel - SUM

int sum(int n, int s) {

if(n == 0) return s;

else return sum(n - 1, n + s); //letzer Aufruf-> Endrek

}

### Binäre Rekursion

Wenn ***zwei rekursive Aufrufe in «non-base case» Aufrufen*** ausgeführt werden.

#### Beispiel – English ruler

static void ruler(int l, int r, int h) {

int m = (l + r) / 2;

if (h > 0) {

*ruler*(l, m, h - 1);

*mark*(h);

*ruler*(m, r, h - 1);

}

}

static void mark(int h) {

for (int i = 0; i < h; i++) {

System.*out*.print('‐');

}

System.*out*.println();

}

#### Beispiel- - tower of hanoi

//Tower of Hanoi

public class TowerOfHanoi {

public static void main (String[] args){

int amountOfDisks = 3;

doTowers(amountOfDisks, 'A', 'B', 'C');

} public void solve(int n, String start, String auxiliary, String end){

if(n==1){

Sysout("Disk-" + n +"from" + start + "to" + end); } else {

solve(n-1, start, end, auxiliary);

Sysout("Disk-" + n +"from" + start + "to" + end);

solve(n-1, auxiliary, start, end); }}}

### beispiel - Fibonacci

//Fibonaci Iterativ  
public int iterativerfibonaci (int n){  
int prev = 1;  
int now = 1;  
if (n <= 1){  
return n;  
} for (int i = 2; i < n; i++){  
int tmp = now;  
now += prev;  
prev = tmp;  
} return now; }  
//Fibonaci Rekursiv  
public int rekursiverfibonaci (int n){  
if (n == 0){  
return 0;  
} else if (n == 1){  
return 1; } else {  
return rekursiverfibonaci(n - 1) + rekursiverfibonaci(n - 2); }}}

//Fibonacci Verbessert Rekursiv

public static long[] fibonacciGood(int n){

if (n==1) {

long[] answer = {n, 0};

return answer;

} else {

long[] temp = fibonacciGood(n-1);

long[] answer = {temp[0]+temp[1], temp[0]};

return answer; }}

# Design patterns

Lösungsansatz eines typischen Software Design Problems

***Best Practice***, ergeben aus besonders gelungenen Programmen

### Inhalt

|  |  |
| --- | --- |
| Mustername | Problemabschnitt/Beschreib. |
| Lösungsabschnitt. / Konzept | Kosequenzabschnitt(+/-) |

### Adapter Pattern – Wrapper Klasse

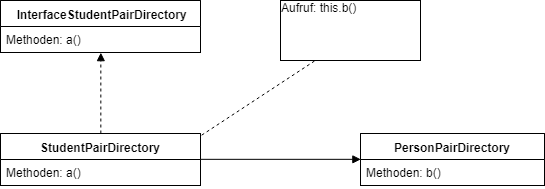
#### Situation

Klasse mit ähnlicher Funktionalität aber anderer Schnittstelle/API

#### LösungEN

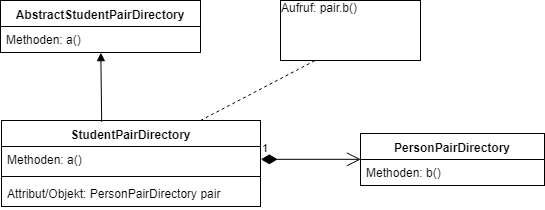
* ***Zwischenobjekt***(Wrapper-Klasse), welches Abbildung der Schnittstell/API durchführt
* ***Neue Klasse***(Adapter), welche Methoden der existierenden Klasse verwendet um die Schnittstelle zu implementieren

##### Vererbung(Klassen-adapter)

***«ist ein(e)»***. z.B Student ist eine Person.

|  |  |
| --- | --- |
| + Ändert einige Methoden der existierenden Klasse und lässt die anderen unverändert.  + Kein zusätzliches Objekt | - Das ganze Interface der existierenden Klasse ist sichtbar.  - Nicht geeignet, um alle Unterklassen der existierenden  Klasse gleichzeitig anzupassen |

##### Komposition(Objekt-Adapter)



***«Ist ein Teil von»***. Z.B Miene ist Teil von Bleistift

public class StudentPairDirectory extends

AbstractStudentPairDirectory {

protected PersonPairDirectory directory;

*/\* versteckte Instanz der zu adaptierenden Klasse \*/*

public StudentPairDirectory() {

directory = new PersonPairDirectory();}

public Student findOther(Student s) {

return (Student) directory.findOther(s);}

|  |  |
| --- | --- |
| + Unterklassen der existierenden Klasse können sehr einfach Adaptiert werden | - Jede verwendete Methode der existierenden Klasse muss  in der Adapter-Klasse definiert werden.  - Zusätzliches Objekt nötig |

# Stack – Stapel – LIFO Prinzip

## Abstrakte Datentypen – ADTs

Abstraktion einer konkreten Datenstruktur und spezifiziert

|  |  |
| --- | --- |
| * Datenfelder/Attribute | * Operationen/Methoden auf/mit Attributen |
| * Ausnahmen & Fehler der Methoden | |

Es ist sozusagen eine erweiterte API.

## Ein Bild, das Text enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungStack ADT – O(1)

* Speichert ADT speichert beliebige Objekte
* Einfügen/Löschen ***nach LIFO/Stapel-Prinzip(last-in-first-out)***
* ***Push()*** – Element auf den Stapel legen
* ***Pop()*** – entfernen und zurückgeben des obersten Elements
* ***Top()*** – liefert das oberste(zuletzt eingefügte) Element ohne entfernen
* ***Size()*** – liefert die Anzahl gespeicherte Elemente
* ***isEmtpy()*** – zeigt an, ob Elemente gespeichert sind

## java.util.Stack<E>

* ***emtpy()*** – zeigt an, ob Elemente gespeichert sind
* ***peek()*** – wie top()
* ***search(Object o)*** – Gibt Position auf dem Stack zurück

Achtung! ***EmptyStackException*** bei top() & pop() möglich!

## Array-basierter Stack

### Grundlagen

* Einfügung von links nach rechts.
* Index des obersten Elements gespeichert z.B t
* Size = t + 1
* t muss mit -1 initialisiert werden
* Wenn t = array.length -1 🡪 SelbstDefinierteException
* O(n) Speicher & Operationen O(1)
* Ein Bild, das Screenshot enthält.

  Mit hoher Zuverlässigkeit generierte BeschreibungMaximale Grösse ist durch Array-Grösse gegeben

## Span-BerechnunG

public static void span2(int[] array) {

int[] output = new int[array.length];

Stack<Integer> stack = new Stack<>();

for(int i = 0; i < array.length; i++){

while(!(stack.isEmpty()) && array[stack.lastElement()] <= array[i]){

stack.pop();}

if(stack.isEmpty()){

output[i] = i + 1;}

else{

output[i] = i - stack.lastElement();}

stack.push(i);

System.out.println(output[i]);}

### Implementierung

public class ArrayStack<E> implements Stack<E>{

private E[] data;

private int t = -1; // Top-of-Stack

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;}

## Listen-basierter Stack

### Grundlagen

### Implementierung – Adapter pattern(Objekt)

public class LinkedStack<E> implements Stack<E> {

private SinglyLinkedList<E> list = new SinglyLinkedList<>();

public int size() {return list.size();}

public boolean isEmpty() {return list.isEmpty();}

public E top() {return list.first();}

public void push(E element) {list.addFirst(element);}

public E pop() {return list.removeFirst();}

}

## Beispiel – kammern-Matchig Algorithmus

public static boolean symbolMatching(char[] klammern) {

Stack<Character> stack = new Stack<>();

for (int i = 0; i < klammern.length; i++) {

if (klammern[i] == '{' || klammern[i] == '[') {

stack.push(klammern[i]);}

if (klammern[i] == '}' || klammern[i] == ']') { if (stack.empty()){

System.out.println("Wrong!");

return false;}

if (stack.pop() != klammern[i]) {

System.out.println("Wrong ! ");

return false;}}}

if (stack.empty()) {

System.out.println("True!");

return true;

} else {

System.out.println("Wrong !");

return false;}

# Queues

## Ein Bild, das Musik enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungQueue ADT

* Einfügen/löschen nach dem FIFO(***first-in-first-out***) Prinzip
* ***Einfügen am Ende – Entnehmen am Anfang***
* ***enqueue(Object)*** – ***Einfügen*** eines Elementes am Ende der Queue
* ***dequeue() –*** ***Entfernen*** und zurückgeben des Elements vom Anfang der Queue
* ***first() –*** liefert das erste Element, ohne dieses zu entfernen
* ***size() –*** liefert die Anzahl gespeicherte Elements
* ***isEmpty() –*** zeigt an, ob Elemente vorhanden sind

***dequeue und first gegen null zurück, wenn isEmpty()***

## Ein Bild, das Uhr enthält. Mit hoher Zuverlässigkeit generierte BeschreibungArray-basierte Queue

* Benutzung eines Array der Länge ***N*** auf zirkuläre Art und Weise
* Zwei Variabeln gebraucht

1. ***f*** – Anfang/Front der Queue
2. ***sz*** – Anzahl der gespeicherten Elemente

* der erste leere Slot wird bestimmt durch:
  + ***r = (f + sz) mod N //rear***

## qUEUE OHNE FESTE gRÖSSE / NodeQueue

1. ***Array*** immer vergrössern(in grösseren Array kopieren: z.B mit Faktor 1.5 oder 2)
2. Ein Bild, das Text, Karte, Uhr enthält.

   Mit hoher Zuverlässigkeit generierte BeschreibungEin Bild, das Text enthält.

   Mit hoher Zuverlässigkeit generierte BeschreibungMit ***Nodes*** arbeiten:

***Problem*** beim Arbeiten mit Nodes 🡪 Beim Entfernen muss tail immer umgehängt werden 🡪 Iteration O(n)!

Ebenfalls kostet die Instanziierung von Node viel!

### Implementierung ArrayBased

public class ArrayBasedQueue {

Person[] array;

int size = 0;

int front = 0;

int r = 0;

int arrayLen;

public ArrayBasedQueue(int capacity) {

array = new Person[capacity];

arrayLen = array.length;}

public void enqueue(Person toAdd) {

if (size == arrayLen) {

throw new IllegalStateException();

} else {

r = (front + size) % arrayLen;

array[r] = toAdd;

size++;}}

public Person dequeue() {

if (size == 0) {

return null;

} else {

Person o = array[front];

array[front] = null;

front = (front + 1) % arrayLen;

size--;

return o;}}

public void print() {

int next = front;

{

while (array[next] != null) System.out.println(array[next].getName()); if (next == arrayLen - 1) {

next = 0;

} else {

next++;}}}}}

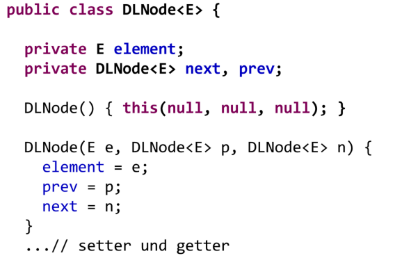
## Ein Bild, das Screenshot enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungJava.util.Queue

## Ein Bild, das Screenshot enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungDouble-Ended Queues

### Deque ADT

* FIFO(first-in-first-out)-Prinzip
* Einfügen am Anfang(front) oder Ende(rear) der Queue
* ***addFirst()*** – einfügen eines Elements am Anfang der Queue
* ***addLast()*** – einfügen eines Elementes am Ende der Queue
* ***removeFirst()*** – entferne das Erste Element der Deque
* ***removeLast()*** – entferne das letzte Element der Deque
* ***first()*** – liefert das erste Element ohne dieses zu entfernen
* ***last()*** - liefert das letzte Element ohne dieses zu entfernen

### Implementierung – BeispieL

Kann einfach mit einer Double Linked List implementiert werden. Dabei sind alle Operationen O(1) da direkter Zugriff:

## Ein Bild, das Screenshot enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungPerfomance Analyse

# List & Iterators

## List ADT

* FIFO(first-in-first-out) & LIFO(last-in-fist-out)-Prinzip
* ***size() –*** liefert die Anzahl Elemente in der Liste
* ***isEmpty() –*** true, falls Liste leer
* ***get(i) –*** gibt das Element an der Stelle i der Liste zurück
* ***set(i, e) –*** ersetzt das Element an der Stelle i durch e und liefert das alte Element zurück
* ***add(i,e) –*** fügt ein Element an der Stelle i in die Liste ein und verschiebt alle nachfolgenden Elemente um eins nach hinten
* ***remove(i) -*** entfernt das Element an der Stelle i und gibt es zurück

## Array-List

Liste, welche im Hintergrund ein Array verwendet. Zusätzlich:

* ***add(e)*** – Fügt das Element zu hinterst ein
* ***indexOf(e)*** – Gibt die Stelle vom Element zurück
* Übergerodnetes System size wegen Performance führen

ArrayList<Bottle> bottles = new ArrayList<>();

### Perfromance analyse

* O(n) Speicherplatz
* ***size(), isEmpty() get() und set() O(1)*** wegen Random Access und size als Übergeordnetes System
* ***add(i,e) & remove(i) O(n)***, wegen Shift-Operation

### Variable Array-basierte Impenentierung

Strategien und das Array dynamisch wachsen zu lassen:

1. ***Inkrementelle Strategie*** - Vergrössert Array um ***Konstante c***
2. ***Verdoppelungs-Strategie*** Verdoppelt der Arraygrösse

Amortisierungszeit(T(n)/n): 1. O(n) 2.O(1)

## java.util.ArrayList<E>

* ***Standard Kapazität 10*** Elemente
* ***Vergrösserung*** immer um 50% also ***Faktor 1.5***

## Positional-Lists / Node-Lists

### Position ADT

* Modelliert das ***Konzept «Platz/Position»*** in einer Datenstruktur.
* ***Pro Position*** wird ***ein einziges Objekt***(z.B ein Array-Element/Node) abgespeichert.
* Höhere ***Abstraktion*** +Beschränkung auf das ***Wesentliche***

### Node-List / Positional-List ADT

* Sequenz von Positionen mit beliebigen Objekten

|  |  |
| --- | --- |
| * ***size()*** | * ***isEmpty()*** |
| * ***first()*** | * ***last()*** |
| * ***before(p)*** | * ***after(p)*** |
| * ***E set(p, e)*** | * ***E remove(p)*** |
| * ***Position addFirst(e)*** | * ***Position addLast(e)*** |
| * ***Position addBefore(p, e)*** | * ***Position addAfter(p, e)*** |
| * ***Iterator<E> iterator()*** | ***Iterable<Position<E>> positions()*** |

#### Implementierung

Einfachste Implementierung mittels ***doppelt verketteter Liste:***

Diese beinhaltet folgendes:

* Element
* Link auf Vorgängerknoten(***previous***)
* Link auf den nächsten Knoten(***next***)
* Beachte: Es gibt speziellen ***header*** und ***trailer*** Nodes

##### Interface Position Implementierung

public interface Position<E> {

E getElement() throws IllegalStateException;}

##### Interface Positional-List

public interface PositionalList<E> extends Iterable<E> {

int size();

boolean isEmpty();

Position<E> first(); //Erste Position oder null

Position<E> last(); //Letzte Position oder null

Position<E> before(Position<E> p) throws IllegalArgumentException;

Position<E> after(Position<E> p) throws IllegalArgumentException;

Position<E> addFirst(E e);

Position<E> addLast(E e);

Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException;

Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException;

E set(Position<E> p, E e) throws IllegalArgumentException;

E remove(Position<E> p) throws IllegalArgumentException;

Iterator<E> iterator();

Iterable<Position<E>> positions();

##### Class Node Impementation

public class Node<E> implements Position<E> {

private Node<E> prev, next;

private E element;// Element stored in this position

/\*\* Constructor \*/

public Node(E elem, Node<E> newPrev, Node<E> newNext) {

element = elem;

prev = newPrev;

next = newNext;

}

public E getElement() throws IllegalStateException {

if ((prev == null) || (next == null))

throw new IllegalStateException("Position ..");

return element;

}

public Node<E> getNext() { return next; }

public Node<E> getPrev() { return prev; }

public void setNext(Node<E> newNext) { next = newNext;}

public void setPrev(Node<E> newPrev) { prev = newPrev;}

public void setElement(E newElement) { element = newElement; }

##### Class Linked-Positional-List

public class LinkedPositionalList<E> implements PositionalList<E> {

private Node<E> header;

private Node<E> trailer;

private int size = 0;

public LinkedPositionalList() {

header = new Node<>(null, null, null);

trailer = new Node<>(null, header, null);

header.setNext(trailer);

}

private Position<E> position(Node<E> node) {

return node; //if node = header or trailer ret null

}

public int size() { return size; }

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

public Position<E> first() {

return position(header.getNext());

}

public Position<E> last() {

return position(trailer.getPrev());

}

public Position<E> before(Position<E> p) {

Node<E> node = validate(p);

return position(node.getPrev());

}

public Position<E> after(Position<E> p) {

Node<E> node = validate(p);

return position(node.getNext());

}

private Node<E> validate(Position<E> p) throws IllegalArgumentException {

if (!(p instanceof Node)) throw new IllegalArgumentException("invalid p");

Node<E> node = (Node<E>)p;

if (node.getNext() == null) throw new IllegalArgumentException("p is no longer on the list");

return node;

}

##### Positional-List (er)Setzen

public E set(Position<E> p, E e) throws IllegalArgumentException {

Node<E> node = validate(p);

E answer = node.getElement();

node.setElement(e);

return answer;}

##### Ein Bild, das Karte, Text enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungPositional-List EInfügen

private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {

Node<E> newest = new Node<>(e, pred, succ);

pred.setNext(newest);

succ.setPrev(newest);

size++;

return newest;}

public Position<E> addFirst(E e) {

return addBetween(e, header, header.getNext());}

public Position<E> addLast(E e) {

return addBetween(e, trailer.getPrev(), trailer);}

public Position<E> addBefore(Position<E> p, E e) {

Node<E> node = validate(p);

return addBetween(e, node.getPrev(), node);}

public Position<E> addAfter(Position<E> p, E e) {

Node<E> node = validate(p);

return addBetween(e, node, node.getNext()); }

##### Positional-List löschen

public E remove(Position<E> p) {

Node<E> node = validate(p);

Node<E> pred = node.getPrev();

Node<E> succ = node.getNext();

pred.setNext(succ);

succ.setPrev(pred);

size--;

E answer = node.getElement();

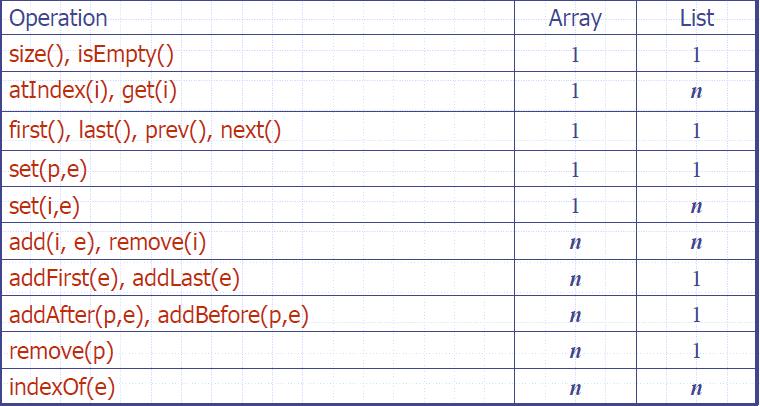
node.setElement(null);

node.setNext(null);

node.setPrev(null);

return answer;}}

##### Perfomance

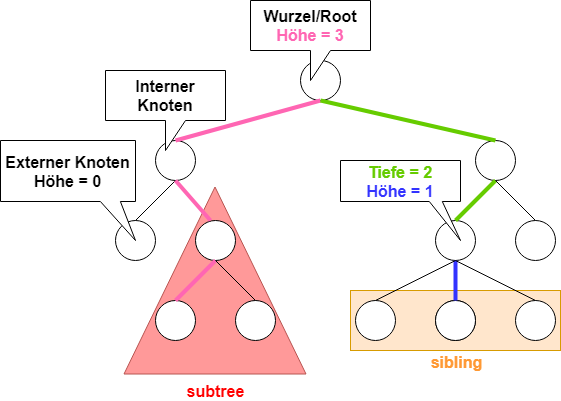
* O(n) Speicher
* Node-List Operationen O(1) sofern man an Position p
* getElement der Position benötigt O(1)
* Suchen nach einer Position O(n)

# Trees – bäume

## Ein Bild, das Text, Karte enthält. Mit sehr hoher Zuverlässigkeit generierte Beschreibunggrundlagen

* Abstrakte, hierarchische Strukturen
* Knoten, welche in Eltern-Lind Relation stehen
* Organigramm, Dateisystem

## Terminilogie – Begriffe

***Wurzel(Root)*** – Knoten ohne Elternknoten(A)

* ***Interner Knoten*** – Knoten mit mind. einem Kind(z.B A oder B)
* ***Externer Knoten(Blatt)*** – Knoten ohne Kinder(z.B E oder K)
* ***Vorgängerknoten*** – Eltern, Grosseltern(alle wo Höhe grö[ß](https://de.wikipedia.org/wiki/%C3%9F)er)
* ***Tiefe –*** Anzahl Vorgänger
* ***Höhe eines Knoten***
  + Externe Knoten 🡪 0
  + Interne Knoten 🡪 1 + maximale Höhe Nachfolgerknoten
* ***Höhe eines Baumes –*** Höhe der Wurzel
* ***Nachfolger –*** Kind, Grosskind, etc.
* ***Subtree –*** Baum aus Knoten und seinen Nachfolgern
* ***Sibling:*** Zwillingsknoten

## Tree ADT

Der Position ADT dient in einem PositionTree als Abstraktion für die Knoten.

|  |  |
| --- | --- |
| ***- Position root()*** | ***- Position parent(p)*** |
| ***- PositionList children(p)*** | ***- Integer numChildren(p)*** |
| ***- Boolean isInternal(p)*** | ***- Boolean isExternal(p)*** |
| ***- Boolean isRoot(p)*** | ***- Integer size()*** |
| ***- Boolean isEmpty()*** | ***- Iterator iterator()*** |

### Tree Interface

public interface Tree<E> extends Iterable<E> {

Position<E> root();

Position<E> parent(Position<E> p) throws IllegalArgumentException;

Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;

int numChildren(Position<E> p) throws IllegalArgumentException;

boolean isInternal(Position<E> p) throws IllegalArgumentException;

boolean isExternal(Position<E> p) throws IllegalArgumentException;

boolean isRoot(Position<E> p) throws IllegalArgumentException;

int size();

boolean isEmpty();

Iterator<E> iterator();

Iterable<Position<E>> positions();

### Ein Bild, das Text, Karte enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungTiefe

public static <E> int depth(Tree<E> T, Position<E> v) {

if (T.isRoot(v))

return 0;

else

return 1 + depth(T, T.parent(v));}

-------------------------------------------------------

Beziehungsweise im PositionTree :

-------------------------------------------------------

public int depth(Position<E> v) {

if (isRoot(v))

return 0;

else

return 1 + depth(v.getParent());}

### Höhe

Ein Bild, das Text, Karte enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibungpublic int height(Position<E> v) {

int h = 0;

for (Position<E> w: children(v) )

h = Math.max(h, 1 + height(w));

return h;}

## Binäre bäume

Ein binärer Baum hat folgende Eigenschaften:

1. Jeder interne Knoten ***max. zwei Kinder***
2. ***Kinder*** eines Knotens sind ***geordnetes*** ***Paar***(links & rechts)
3. ***Echter Binärbaum 🡪*** interne Knoten ***genau zwei Kinder***

### Binärbaum - Binarytree ADT

Erweiterung zum Tree ADT mit:

|  |  |
| --- | --- |
| ***- Position left(v)*** | ***- Position right(v)*** |
| ***- Position sibling(v)*** Interface binarytree | |

public interface BinaryTree<E> extends Tree<E> {

public Position<E> left(Position<E> v);

public Position<E> right(Position<E> v);

public Position<E> sibling(Position<E> v);}

### Abstract-Binarytree

public abstract class AbstractBinaryTree<E>

extends AbstractTree<E> implements BinaryTree<E> {

@Override

public Position<E> sibling(Position<E> p) {

Position<E> parent = parent(p);

if (parent == null) return null; // p must be the root

if (p == left(parent)) // p is a left child

return right(parent); // (right child, might be null)

else // p is a right child

return left(parent); // (left child, might be null)

}

@Override

public int numChildren(Position<E> p) {

int count=0;

if (left(p) != null)

count++;

if (right(p) != null)

count++;

return count;

}

@Override

public Iterable<Position<E>> children(Position<E> p) {

List<Position<E>> snapshot = new ArrayList<>(2);

// max capacity of 2: left, right

if (left(p) != null)

snapshot.add(left(p));

if (right(p) != null)

snapshot.add(right(p));

return snapshot;}}

### Eigenschaften binäre Bäume

Ein Bild, das Text enthält.

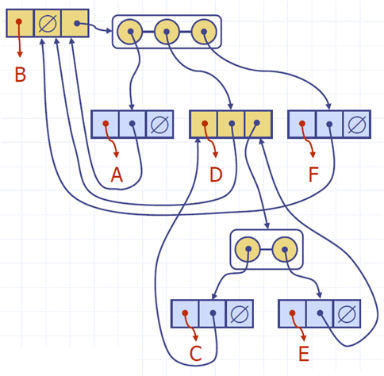
Mit sehr hoher Zuverlässigkeit generierte Beschreibung

* ***n*** Anzahl Knoten
* ***e*** Anzahl externer Knoten
* ***i*** Anzahl interner Knoten
* ***h*** Höhe

## Ein Bild, das Text enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungSpeicherverfahren für bäume – linked list

Ein Baumknoten beinhaltet folgendes:

* Element
* Elternknoten
* Sequenz mit Kindknoten



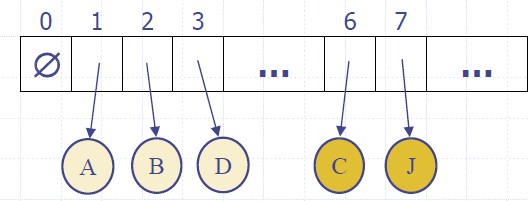
🡪***beliebig viele Children***

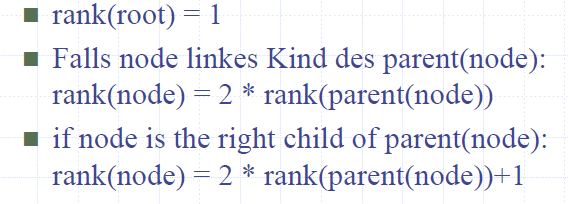
## linked list für binarrytrees

🡪***max zwei Childs***

## Speicherverfahren für Bäume – array basiert

Ein Bild, das Text, Karte enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungDie Knoten werden in einem Array gespeichert:

* ***Rank(root) = 1***
* ***Linkes kind: 2 \*rank(parent(node))***
* ***Rechtes Kind: 2 \*rank(parent(node)) + 1***

## Baum-Traversierungen

Bei einer Baum-Traversierung werden alle Knoten eines Baumes auf systematisch Art und Weise besucht werden.

Dabei gibt es verschiedene Traversierungs-Algorithmen:

### preoder

***Ein Bild, das Flasche enthält.

Mit hoher Zuverlässigkeit generierte Beschreibung***In der Pre-Order Traversierung wird ein ***Knoten vor seinen Nachfolgern*** besucht***:***

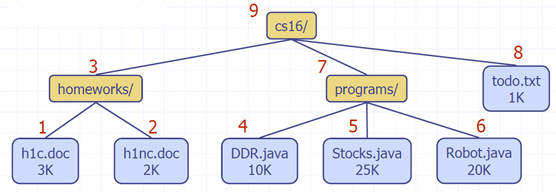
Ein Bild, das Text, Karte enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

### Postorder

***Ein Bild, das Flasche enthält.

Mit hoher Zuverlässigkeit generierte Beschreibung***In einer Post-Order Traversierung wird ein ***Knoten nach seinen Nachfolgern*** besucht.



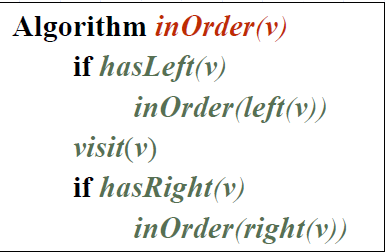
### breadth-fist – Breitensuchen

Ein Bild, das drinnen, Flasche enthält.

Mit hoher Zuverlässigkeit generierte BeschreibungIn einer Breadth-First Traversierung werden ***zuerst alle Knoten einer Tiefe t*** besucht, ***dann alle Knoten der Tiefe t+1***, t+2, usw.

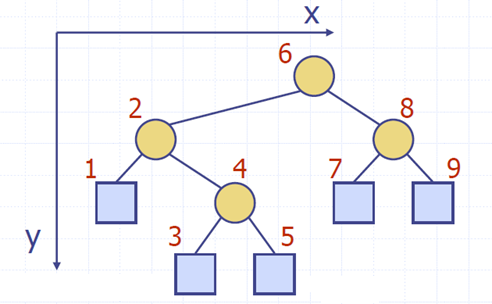
### 

### inorder

In einer Inorder Traversierung wird ein Knoten ***nach seinem***

***linken Subtree und vor seinem rechten Subtree*** besucht.

Ein Bild, das Objekt enthält.

Mit hoher Zuverlässigkeit generierte Beschreibung

## Anwendungsbeispiele Traversierungen

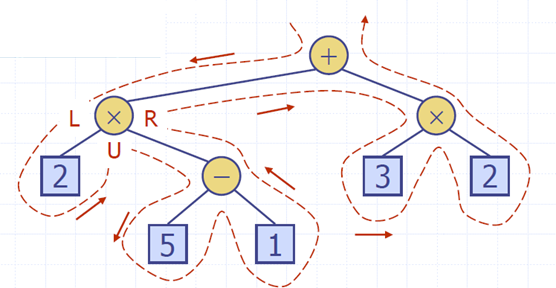
* Ein Bild, das Text, Karte enthält.

  Mit sehr hoher Zuverlässigkeit generierte BeschreibungPreorder 🡪Arithmetische Ausdrücke:
* Postorder 🡪 benutzter Speicher in einem Verzeichnis inkl. Subverzeichnissen
* Ein Bild, das Objekt, Antenne enthält.

  Mit sehr hoher Zuverlässigkeit generierte BeschreibungBreadth-First 🡪 Tic-Tac-Toe:
* Ein Bild, das Text, Karte enthält.

  Mit sehr hoher Zuverlässigkeit generierte BeschreibungInorder🡪Arithmetische Ausdrücke:

## Euler Tour Traversierung

* Generische Traversierung binärer Bäume
* Die Preorder, Postorder und Inorder Traversierungen sind ***Spezialfälle*** der Euler Traversierung
* Jeder Knoten wird drei Mal besucht
* Einmal von **links (preorder): L**
* Einmal von **unten (inorder): U**
* Einmal von **rechts (postorder): R**

### Ein Bild, das Text enthält. Mit hoher Zuverlässigkeit generierte BeschreibungImplementierung

*Visit-Methoden verfeinert in Subklassen:*

Ein Bild, das Screenshot, Text enthält.

Mit hoher Zuverlässigkeit generierte Beschreibung

## template method patern(Schablonenmuster)

*«Hollywood Prinzip»* oderauch *«Don’t call us, we will call youEin Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung»*

* Definieren eines Rumpfes eines Algorithmus, wobei ***einige Teilschritte erst später in Subklassen spezifiziert*** werden.
* ***Einige Teile unveränderlich, andere anpassbar***
* ***Gemeinsame Teile*** in ***abstrakte Klasse implementiert***
* ***Variable Teile*** implementiert in ***konkreten Klassen***
* Die übergeordnete Methode ruf bei Bedarf die untergeordnete, verfeinerte Methode auf.
* Vor allem bei Frameworks eingesetzt

### Beispiel

Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungMyDocument und MyApplication beinhalten die variablen Teile – die verfeinerten Methoden.

# Iteratoren

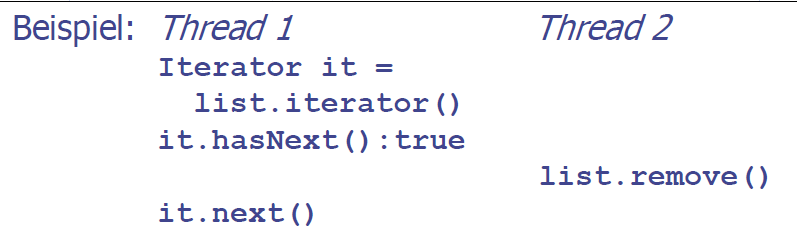
|  |  |
| --- | --- |
| * ***Boolean hasNext()*** | * ***Element next()*** |

## Snapshot-iterator

Beim Erzeugen des Iterators wird eine ***Kopie*** der Ausgangs-Datenstruktur erzeugt.

|  |  |
| --- | --- |
| + Änderungen beim Original keinen Einfluss | * Hohe Kosten 🡪 O(n) |

## lazy-iterator

Es wird auf der ***Original-Datenstruktur*** iteriert.

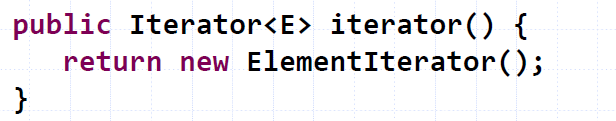
|  |  |
| --- | --- |
| + niedrige Kosten 🡪 O(1) | * Strukturänderungen verunmöglichen Iteration |

🡪***Lösung***: Iterator enthält selber ***Manipulations-Methoden*** wie z.B. remove()[Bei Fehlverhalten gibt’s eine Exception]

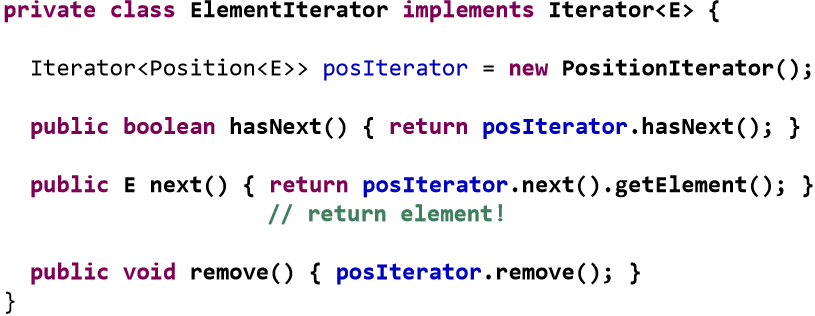
### Ein Bild, das drinnen enthält. Mit sehr hoher Zuverlässigkeit generierte Beschreibungjava.util.iterator

## Grundsätzliche Implementierung

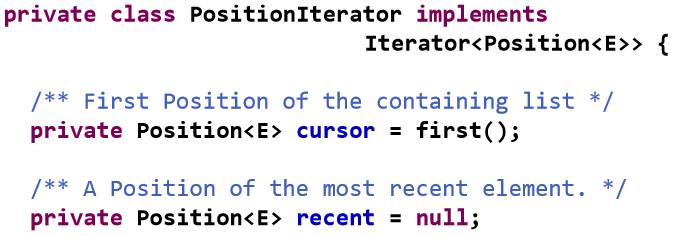
Typischerweise wird der ***Iterator als private innere Klasse*** implementiert. Dann hat man eine ***iterator-Methode***, welche einen spezifischen Iterator zurückgibt.

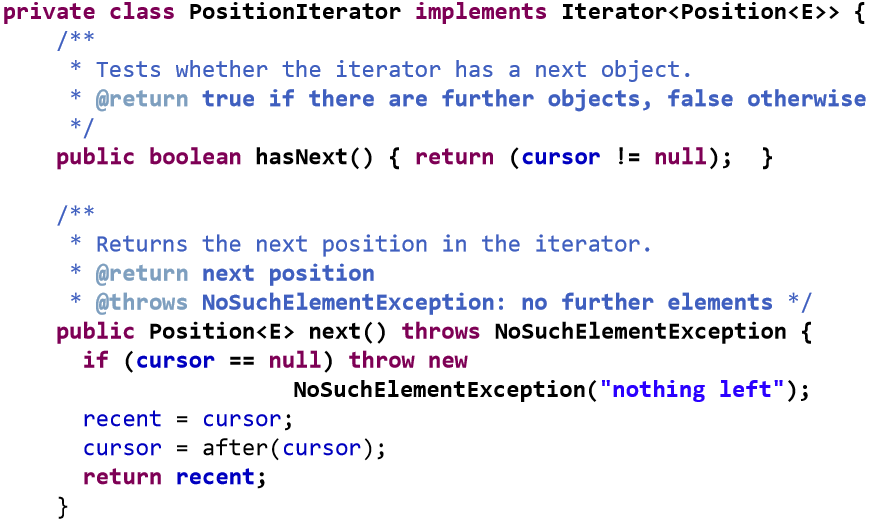
Somit wird erreicht, dass der Iterator ***direkt*** auf die Elemente der zu iterierenden Klasse ***zugreifen*** kann.

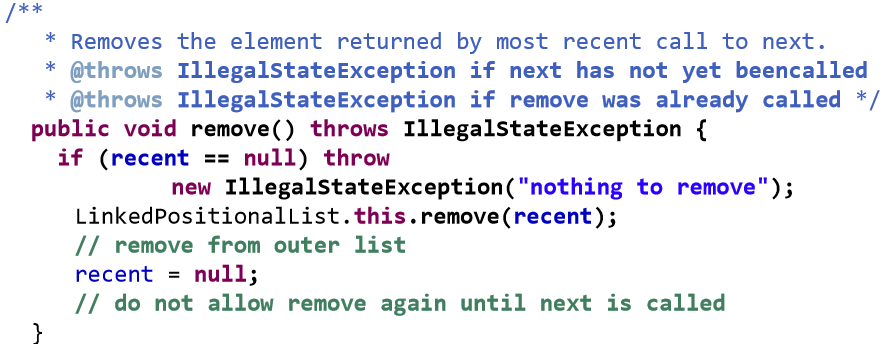
### Element Iterator



### Positional Iterator







## java.util.listiterator

Zusätzliche Funktionen:

|  |  |
| --- | --- |
| * ***void add(E e)*** | * ***Boolean hasPrevious()*** |
| * ***int nextIndex()*** | * ***E previous()*** |
| * ***int previousIndex()*** | * ***void remove()*** |
| * ***set(E e)*** |  |

### Anwendung

Bottle.next().remove()

Bottle.previous().remove()

If(nextIndex == 1) { next().remove()}

# Ein Bild, das Tasse enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungPriority Queues

Collection von Einträgen/Entries

|  |  |
| --- | --- |
| * ***insert(k, v)*** | * ***removeMin()*** |
| * ***min()*** | * ***size()*** |
| * ***isEmpty()*** |  |

## Entry adt

Ein Bild, das Flasche enthält.

Mit hoher Zuverlässigkeit generierte BeschreibungEntry besteht aus ***Schlüssel-Wert-Paar***(Key-Value)

|  |  |
| --- | --- |
| * ***key()*** | * ***value()*** |

### Mathematisches Konzept

Ordnungs-Relation: ≤

* ***Reflexiv***: x ≤ x
* ***Antisymmetrisch***: x ≤ y ^ y ≤ x => x = y
* ***Transitiv***: x ≤ y ^ y ≤ x => x ≤ y

Zwei verschiedene Entries ***gleicher Key möglich***.

## Comparator adt

Ein Bild, das Text enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungEin Comparator wird eingesetzt um zwei Objekte gemäs iener Vollständigen Ordnungsrelation zu ***vergleichen***.

Eine Priority Queue benutzt einen Comparator(als Hilfs-ADT) um die ***Schlüssel zweier Entries zu vergleichen***:

### Ein Bild, das Screenshot enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungBeispiel

## Sortieren mithilfe einer Priority Queue

Mithilfe einer Priority Queue kann man ganz einfach vergleichbare Elemente sortieren(Brute Force Methode):

1. Einfüge der Elemente – ***insert(e)***
2. Entfernen in sortierter Reihenfolge – ***removeMin()***

Je nach Implementierung ist mit verschiedenen Laufzeiten zu rechnen:

### unsortierte Liste

***Insert(e) 🡪O(1)***

***removeMin() & min() 🡪 O(n)***

#### Selection-Sort

Der Selection-Sort-Algorithmus baut auf einer Priority Queue mit unsortierter Liste auf:

1. Einfügen von n Elementen: O(n)
2. Entfernen von n Elementen
   1. N removeMin()🡪 n + … + 2 + 1 🡪 n(n+1)/2

**🡪*Selection Sort benötigt O(n2) Zeit***

Weil beim removeMin() Mal durchiterieren ist der ***Best- & Worst-Case identisch:*** n+(n(n+1)/2)

### sortierte Liste

***Insert(e) 🡪O(n)***

***removeMin() & min() 🡪 O(1)***

#### Insertion-Sort

Der Insertion-Sort Algorithmus baut auf einer sortieren Liste auf:

1. n insert()-Operationen 🡪 1 + 2 + … + n🡪n(n+1)/2
2. n removeMin()-Operationen[1+1+1+1…=n] 🡪O(n)

🡪 ***Insertion-Sort benötigt O(n2) Zeit***

Aber es gib erhebliche **Unterschiede beim Best und Worst-Case**:

* ***Worst Case: n(n+1) / 2 + n***
* ***Best Case, wenn absteigend sortierter Input: n + n***
* ***Speicher*** bei LinkedList 🡪 n bei Hilfsstrukturen(Array) 🡪2n

##### In-place Insertion Sort

Dieser Algorithmus ***agiert nur auf der Input-Datenstruktur*** und verzichtet so auf externe Datenstrukturen:

Ein Teil der Eingabesequenz gilt als Priority Queue:

* Speichert der erste Teil der Sequenz schrittweise sortierte Elemente
* Mithilfe von ***swaps*** wird die Sequenzweise modifiziert
  + Es wird noch eine Variabel gebraucht/verwendet
* Vergleiche Insertion Sort Algorithmus bei den Arrays.

# Heaps

## Grundlagen

Ein Bild, das Text, Karte enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungEin Heap ist ein ***Binärbaum, welcher in seinen Knoten Schlüssel speichert*** und folgende Eigenschaft besitzt:

Jeden Knoten v, welcher nicht Wurzel ist gilt:

***key(v) ≥ key(parent(v))***

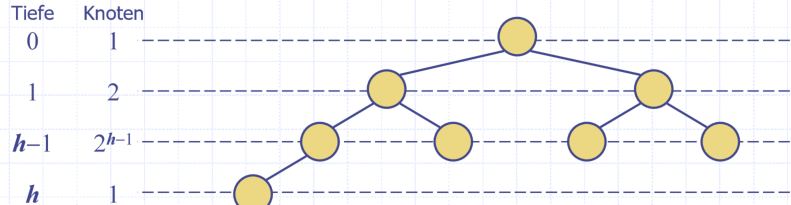
* Je grösser die Tiefe, desto grösser der Schlüsselwert. Die Vorfahren haben immer grössere Schlüssel!

## Eigenschaften

Zusätzlich hat ein Heap folgende Eigenschaften:

1. Es sind ***2i-Knoten auf der Tiefe i*** vorhanden(***volle*** ***Levels***)
2. Es wird von ***links her aufgefüllt***
   1. Auf der Tiefe h-1 befinden sich die internen Knoten links von den externen Knoten.
3. ***Maximal ein Knoten mit einem Kind*** 🡪 dieser Knoten muss ein ***linkes*** ***Kind*** sein
4. Der ***letzte Knoten ist der weitesten rechts stehende Knoten*** auf der grössten Tiefe.

🡪Sei h die Höhe des Heaps mit n Knoten



|  |  |
| --- | --- |
| Obere Grenze | Untere Grenze |
| *n = 1 + 2 + 4 + … + 2h =2h+1 - 1* | ***n = 2h*** |
| *h = log(n+1) -1* | ***h = log(n)*** |

## Heaps & Priority Queue

Mithilfe eines Heaps kann eine Priority Queue implementiert werden:

* Jeder Knoten speichert einen Entry<Key, Element> ab
* Ein Bild, das Text, Karte enthält.

  Mit sehr hoher Zuverlässigkeit generierte BeschreibungDer letzte Knoten wird speziell gemerkt

### einfügen von Entries in einen Heap

Ein Bild, das Text, Karte enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

Die insert()-Methode entspricht dem Einfügen eines Schlüssels k in den Heap.

1. Der ***Einfügeknoten finden***(z)werden(letzter Knoten deshalb gemerkt)
2. ***Schlüssel k in z abspeichern***
3. ***Heap-Eigenschaften überprüfen*** und gegebenfalls wieder herstellen(***Up-/Downheap***)

#### Upheap

Nach Einführen eines neues Schlüssels k könnten die Heap-Ordnungseigenschaften verletzt sein. Der upheap-Alhgorithmus stellt die Heap-Ordnungseigeschaft folgendermassen wieder her:

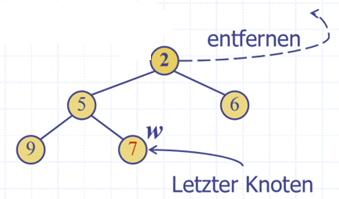
* Ein Bild, das Karte, Text enthält.

  Mit sehr hoher Zuverlässigkeit generierte BeschreibungDer neu eingefügte Knoten k wird so lange mit dem darüberliegenden Knoten vertauscht, bis der neue Knoten ***entweder die Wurzel*** erreicht hat, oder der ***Elternknoten einen kleineren oder gleichgrossen*** Schlüssel hat.

Da der Heap eine Höhe von log(n) benötigt der Algo. ***O(log(n))***

### Entfernen von Entries in einen Heap

Die removeMin-Methode entspricht dem ***Entfernen des Wurzel-Knotens*** eines Heaps:

1. Entfernen des Wurzel -Schlüssels (Minimum)
2. Wurzel-Schlüssel mit letztem Knoten(w) ersetzen.
3. Ein Bild, das Text, Karte enthält.

   Mit sehr hoher Zuverlässigkeit generierte BeschreibungEntfernen des letzten Knotens(w)
4. Ein Bild, das Text, Karte enthält.

   Mit hoher Zuverlässigkeit generierte BeschreibungWiederherstellen der Heap-Ordnung abwärts(Down-Heap)

#### downheap

Nach Ersetzung des Wurzel-Schlüssels wird die Heap-Eigenschaft vermutlich verletzt. Der Down-Heap stellt die Heapordnungseigenschaft folgendermassen wieder her:

* Der neue Wurzel-Schlüssel k wird solange mit dem jeweiligen kleineren Kind getauscht, bis er einen Blattknoten erreicht hat oder die Kinder alle einen Schlüssel haben, welcher grösser oder gleich dem Schlüssel k sind.

Ein Bild, das Text, Karte enthält.

Mit hoher Zuverlässigkeit generierte Beschreibung

Da der Heap eine Höhe von log(n) benötigt der Algo. ***O(log(n))***

### Heap-sort

Gegeben: Priority Queue mit n Einträgen mithilfe eines Heaps:

* Speicherbedarf 🡪 O(n)
* insert() & removeMin()🡪O(log(n)
* size((, isEmpty() und min() 🡪 O(1)

Um eine Sequenz von n-Elementen zu sortieren braucht es:

1. n-insert() – O( log(n))
2. n-removes() – O(log(n))

🡪2n\*log(n) = ***O(n log(n)) Zeit***

#### Vergleich Heap-Sort – Quadratische Algorithmen

Der Heap-Sort ist viel schneller als quadratische Algorithmen wie beispielsweise Insertion & Selection-Sort:

1000 Elemente

Insertion/Selection Sort: 10002 = 106

Heap-Sort: 1000 log(1000) = 104

***Bei 1000 Elementen ist der Heap-Sort 100x schneller***

### Vector-basierte Heap-Implementierung

Ein Bild, das Uhr enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungEin Heap mit n-Knoten kann auch mittels einem ***Vektors der Länge n+1*** realisiert werden: Es gelten folgende Eigenschaften für den Knoten mit Index i:

* der ***linke Kindknoten*** wird bei Index ***2i*** gespeichert
* der ***rechte Kindknoten*** wird bei Index ***2i + 1*** gespeichert

#### insert -Neue Knoten einfügen

Neue Knoten werden an ***Stelle/Index n+1 eingefügt***.

#### removemin – knoten entfernen

Das Entfernen von Knoten wird mit der Operation removeMin() erreicht und entspricht dem ***Entfernen bei Index 1.***

#### Inplace Sortierung

Diese Datenstruktur führt automatisch zu einer inplace-Sortierung.

### Ein Bild, das Karte, Text, Himmel enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungZwei Heaps zusammenführen

Gegeben: zwei Heaps und ein neuer Schlüssel k:

1. K als Wurzel-Schlüssel + zwei Unterbäume
2. Downheap()

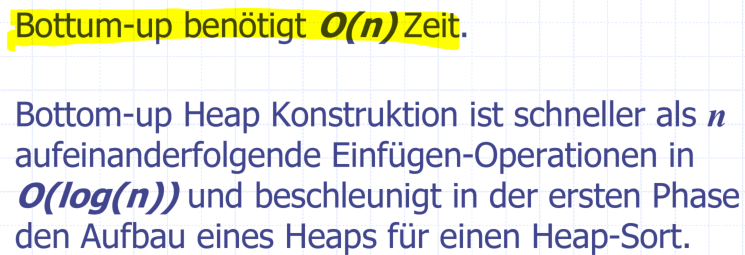
### Bottom-Up Heap-Konstruktion

Ein Bild, das Schild, draußen, Gebäude enthält.

Mit hoher Zuverlässigkeit generierte BeschreibungEin Heap, welcher aus n-Schlüssel besteht, lässt sich am besten Bottom-UP(vom Grund auf) aufbauen:

Das Prinzip funktioniert folgendermassen:

🡪***Man nimmt zwei kleine Heaps mit 2i-1 Schlüsseln und fügt diese zu einem Heap mit 2i+1-1 Schlüssel zusammen***Anzahl Heaps: (n+1)/2i

1. (n+1)/2 Knoten einfügen
2. (n+1)/4 Knoten einfügen
3. (n+1)/8 Knoten einfügen usw.

Wenn n keine zweier Potenz -1 einfach mit Dummy-Daten auffüllen.

# Adaptable Priority Queues

Neue Methoden:

|  |  |
| --- | --- |
| ***remove(e)***  Entfernt Entry e aus P und liefert e zurück | ***replaceKey(e,k)***  Der Schlüssels des Entries e wird durch k ersetzt und der alte Schlüssel zurückgegeben |
| ***replaceValue(e,v)***  Der Wert der Entry e wird durch v ersetzt und der alte Wert zurückgegeben | ***insert(k,v)***  Gibt neu den Entry zurück |

## location-Aware Entries

Um die neuen Operationen effizient zu implementieren müssen die Entries schnell und effizient lokalisiert werden können.

Eine Lokations-bewusste Entry identifiziert und verfolgt die Lokation ihrer(Key, Value) Objekte innerhalb einer Datenstruktur.

***Die Datenstruktur wiess selber, wo sich Einträge befinden 🡪O(1)***

* ***Entries werden von der zu Grunde liegenden Datenstruktur generiert und deren Position an Benutzer geliefert***

***🡪=O(1) wird erreicht mit einer Position, welche auf die Entries zeigen.***

## listen basierte implementierung

Ein Bild, das Text, Karte enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungEine Lokations-bewusste Listen-Entry ist ein Objekt, welches folgende Informationen abspeichert:

* ***Schlüssel***
* ***Wert***
* ***Position(Index) des Items in der Liste***

Die Position(oder Array-Zelle) speichert die Entry

Read-only-Operationen(niemand Zugriff auf interne Datenstrukt.)

## Heap Implementierung

Ein Bild, das Text, Karte enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

Eine Lokations-bewusste Heap-Entry ist ein Objekt, welches folgende Inhalte speichert:

* ***Schlüssel***
* ***Wert***
* ***Position der Entry***(im zugrunde liegenden Heap)

Umgekehrt speichert eine ***Heap-Position eine Entry***.

## Ein Bild, das Text enthält. Mit hoher Zuverlässigkeit generierte BeschreibungPerformance

# Maps

## ADt

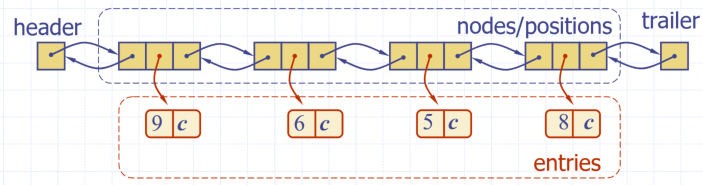
Eine Map modelliert eine durchsuchbare Collection von Schlüssel-Wert Entries.

Einer der wichitgsten Unterschiede zu den anderen Schlüssel-Wert-Datenstrukturen ist, dass ***pro Key nur ein Entry*** erlaubt ist.

BSP: Adressbuch, Einfache Datenbanken

* ***get(k)***
  + ***key vorhanden***🡪 Wert zurück
  + ***key nicht vorhanden🡪*** null zurück
* ***put(k,v)***
  + ***key vorhanden***🡪 Wert ersetzten + alter Wert zurück
  + ***key nicht vorhanden***🡪 neuen Entry hinzufügen + null retourniert
* ***remove(k)***
  + ***key vorhanden***🡪 Entry entfernt + dazugehöriger Wert zurück
  + ***key nicht vorhanden***🡪 null zurück
* ***size(), isEmpty() -*** selbsterklärend
* ***keySet()*** ***–*** liefert iterierbare Collection mit allen Schlüsseln
* ***values()*** –liefert iteriebare Collection mit allen Werten
* ***entrySet()*** – liefert iterierbare Collection mit allen Entries

## Listen-basierte Implementierung

Mithilfe einer unsortierten(Entries in beliebiger Reihenfolge) Liste kann eine MAP-ADT implementiert werden:

put(n),get(n), remove() 🡪O(n)

🡪Implementierung ***nur bei kleinen Listen sinnvoll***.

### get(k)-Algorithmus

### Ein Bild, das Text enthält. Mit sehr hoher Zuverlässigkeit generierte Beschreibungput(k,v)-Algorithmus

### Ein Bild, das Text enthält. Mit sehr hoher Zuverlässigkeit generierte Beschreibungremove(k)-Algorithmus

### Sentinal Trick

Mit ***Einführen eines zusätzliche Knotens am Ende*** der Liste kann man die Abfragen um die Hälfte reduzieren:

Anstatt jedes Mal hasNext() zu prüfen einfach so:

trailer.key = k

next = header.next()

while(next.key() != k)

{

next = next.getNext();

}

If(next != trailer)

{

return next.value() ;

}

return null ;

# Hash Tabellen

## Hash-Tabelle

Eine Hashtabelle für einen gegebenen Key-Typus besteht aus:

* Hash-Funktion h
* Array(genannt Tabelle) der Grösse N

***Anomalien***: Leerstellen, Kollisionen(denselben Hashcode)

Hash-Funktion heisst ***perfekt***, wenn keine Kollisionen vorhanden

## Hash-funktion

Eine Hash-Funktion h bildet ***Keys auf Integers*** ab.

### beispiel

Der Integer ***h(x) nennt man Hashwert*** des Keys x.

### Aufbau

Diese Funktionen sind meistens in zwei Teile aufgeteilt:

1. ***Hash-Code: h1: Keys 🡪 Integer***
   1. Schlüssel möglichst zufällig verteilen
2. ***Kompressfunktion: h2:Integers 🡪 [0, N-1***]
   1. ***Schlüssel in ein fixes Intervall transformiert***

### Hash-codes

* ***Memory Adresse***
  + Standard in Java
  + Die Memory Adresse des Schlüssel-Objekts wird als Integer interpretiert
  + Allg. gute Wahl ausser für numeris. Werte & Strings
* ***Integer Cast***
  + Schlüssel wird als Integer Zahl interpretiert
  + Gut, solange Anzahl Bits Interpretation als Integer erlaubt
* ***Komponentensumme***
  + Bits der Schlüssels in Komponenten mit fixer Länge(16/32Bit=unterteilt und summiert(Overflow ignoriert)
  + Gut für Schlüssel mit fixer Länge
* ***Polynom-Akkumulation***
  + Bits des Schlüssels in Sequenz von Komponenten gleicher fixer Länge(8,16, 32 Bits) zerlegt. Daraus das Polynom berechnet:
  + Fixer Wert z, Overflow ignoriert
  + Sehr gut für Strings

#### Ein Bild, das Shoji enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungBeispiel polynom-Akkumulation

Bei s=»ab» 🡪 97 \* 31 + 98 = 3105

a ASCII = 97, b ASCII = 98

### Kompress Funktionen

* ***Division***
  + ***h***2 (***y***) = ***y*** mod ***N***
  + N oft Primzahl(Zahlentheorie)
* ***Multiply, Add, Divide(MAD)***
  + ***h2 (y) = (ay + b) mod N***
  + a & b nichtnegative Integer, a mod N ≠ 0

### Kollisionsbehandlung

Ein Datensatz s mit Schlüsselwert w heisst Überläufer, wenn der durch h(w) zugewiesene Behälter bereits belegt ist.

🡪***Überläufer*** treten auf, wenn der errechnete ***Hashwert*** ***bereits*** ***in Benutzung*** ist.

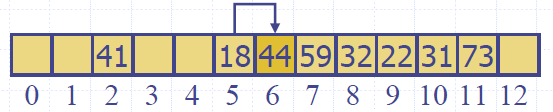
#### geschlossene Addressierung - offenes HashVerfahren - Separate Chaining

* Behälter sind ***verkettete Listen***
* Jede Zelle der Tabelle zeigt auf eine Liste
* Unbegrenzt
* Selten Überläufer

|  |  |
| --- | --- |
| + Verkettung einfach | - zusätzliche Datenstruktur |

#### Offene Addressierung – geschlossenes hashverfahren

Für kollidierende Elemente wird ein ***Platz in der Nähe*** gesucht.

Dabei gibt es verschiedene Sondierungsverfahren.

Inspizierte Zelle = ***probe***

Kollidierende Items = ***cluster***

#### Sondierfunktionen s(k,i)

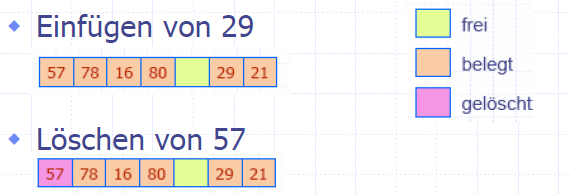
* ***Lineares Sondieren*** (linear probing):
  + linear nach einer freien Stelle suchen;
  + s(k,1)=h(k)+1, s(k,2)=h(k)+2, ...
* ***Lineares negatives Sondieren***
  + Rückwärts linear Minus statt Plus
* ***Quadratisches Sondieren***
  + Quadratische Funktion
  + s(k,2)=h(k)+22=h(k)+4,
  + s(k,3)=h(k)+32=h(k)+9
* ***Alternierendes Sondieren***
  + Es wird abwechselnd davor und dahinter gesucht
  + Vorwärts dann rückwärts(+ dann -)
* ***Alternierendes quadratisches Sondieren***
  + Wie Alternierendes Sortieren, jedoch Schritte quadratisch
  + s(k,2)=h(k)+22= h(k)+4
  + s(k,3)=h(k)-32=h(k)-9
* Zufälliges Sondieren
  + Zufallsfunktion implementiert

#### Suche mit linearer Sondierung

1. Bei Zelle h(k) wird gestartet
2. Aufeinanderfolgende Zellen durchsuchen, bis:
   1. Eintrag zum Schlüssel k gefunden
   2. Leere Zelle gefunden
   3. N Zellen durchsucht(return null)

#### Löschen mit linearer Sondierung

Datensatz wird als ***gelöscht markiert(DEFUNCT)***, weil ansonsten die ***Sondierungsfolge*** für einen anderen Datensatz ***unterbrochen*** werden kann. Es wird zum «Weiterhangeln» benutzt.

Allgemein gibt es die Markierungen «***frei***», «***belegt***» oder «***gelöscht***»

## Doppeltes Hashing - Kollisionsvermeidung

Doppeltes Hashing benutzt eine zweite Hash-Funktion d(k):

***(h(k) + jd(k)) mod N***

Für j = Anzahl Kollisionen

k: d(k) != 0, Tabellegrösse N muss Primzahl sein

Kompressfunktion:

***d(k) = q - k mod q***

wobei q < N & q ist Primzahl

## Performance von Hashing

### Schlimmster fall – worst case

* Alle Elemente führen zu Kollisionen
* Suchen, Einfügen und Löschen 🡪 O(n)
* Lastfaktor α= n /N bestimmt Zeitverhalten
* Hashwerte gleichverteilt 🡪 Einfügen 1/(1- α)

### Erwartetes Resultat

* Alle Map-Operationen in Hash-Tabelle O(1)
* Praxis: sehr schnell, falls Lastfaktor nicht nahe bei 100%
* Rehash automatisch

### Analsyse der offenen Adressierung

+ erfolgreiche Suche

* Erfolglose Suche

Α = n/N

Zahl und Länge der Sondierungsketten(cluster) kleinhalten.

Faustregel Lastfaktor von 0.8 nicht überschritten

# Skipliste

## Grundlagen & Definition

Eine Skip Liste für ein Set S unterschiedlicher (Key, Element)-Paare besteht aus einer ***Serie*** ***von*** ***ListenI*** S0, S1, …., sh so dass:

* Jede Liste si einen künstlichen Anfangs-(-∞) und End-Knoten(+∞) haben. Die Liste Sh hat nur diese.
  + Diese sind eindeutig & leicht zu prüfen
* S0 alle Keys von S in aufsteigender Reihenfolge(Teilmengen)

## Ein Bild, das Uhr enthält. Mit sehr hoher Zuverlässigkeit generierte BeschreibungPerfekte Skipliste

Ein Bild, das Uhr, Wand enthält.

Mit hoher Zuverlässigkeit generierte BeschreibungMan nennt eine Skipliste perfekt, wenn jede Liste jeweils ***Knoten in der Mitte der Intervalle der Nachfolgerliste*** enthält.

* Bei jeder ***Mutation*** muss die Liste ***reorganisiert*** werden und ist dann eventuell nicht mehr perfekt.

## randomisierte Skipliste

Ein Bild, das Uhr, Shoji enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungMan nennt eine Skipliste randomisiert, wenn die Knoten der Listen nicht in der Mitte der Intervalle der Nachfolgerlisten sind.

## Suchen

1. Start🡪 erste Position in Topliste
2. An einer Position p vergleicht max x mit y=key(next(p))
   * ***X = y: return element(next(p))***
   * ***X > y: scan forward***
   * ***X < y: drop down***
3. Falls auf dem Boden angelangt und wieder «drop down» dann soll null zurückgegeben werden

## Randomisierte Algorithmen

* Laufzeit hängt von Ergebnis ab(z.B Münzwurf)
* Worst-case meist sehr hohe Laufzeit aber sehr unwahrscheinlich

## Einfügen einer Entry

Mithilfe eines randomisierten Algorithmus Eintrag(k,o) einfügen:

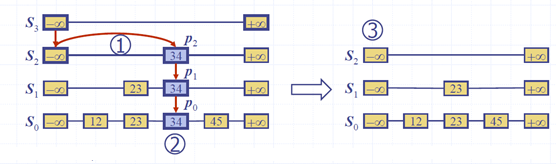
1. Erfolglose Suche nach k und ***merke p0*** mit grösstem Schlüssel, welcher kleiner ist als k.
2. Füge neuen Knoten p mit Schlüssel k ***nach*** ***p0*** ein, wähle dabei eine ***zufällige Höhe***.
3. Ein Bild, das Screenshot enthält.

   Mit hoher Zuverlässigkeit generierte BeschreibungErzeuge Zufalls-Höhe des neuen Turmes(Wahrscheinlichkeit lö1/2i+1)
4. Ein Bild, das Text, Uhr enthält.

   Mit hoher Zuverlässigkeit generierte BeschreibungFüge p in alle Listen ein, bei welcher die Höhe niedriger sind als die gewählte Höhe(0<i<Höhe)

## Entfernen einer Entry

Entry mit Schlüssel x aus Skip Liste entfernen:

1. Positionen p0, p1 ,… ,pi zum Key x finden.
2. Nun die gefunden Positionen entfernen
3. Alle bis auf eine Liste entfernen, welche nur Spezial-Keys haben

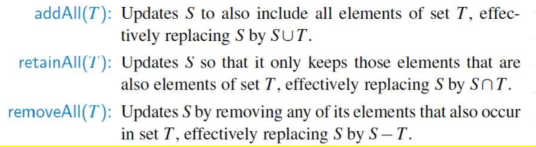
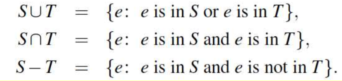
## Speicherplatz/Performance-Analyse

* O(n) Speicher
* Suchen, Einfügen, löschen O(log(n)) Zeit

Viel Speicher aber dafür schnelle Suche!

# Sets, multisets & multimaps

## Set

Ein Bild, das drinnen enthält.

Mit hoher Zuverlässigkeit generierte BeschreibungUnsortierte Collection von Elementen, ***ohne*** ***Duplikate***

### Speichern eines Sets in einer Liste

Ein Set implementiert mit einer Liste führt zu:

***Element nach definierter Ordnung gespeichert***

***Speicher ist O(n)***

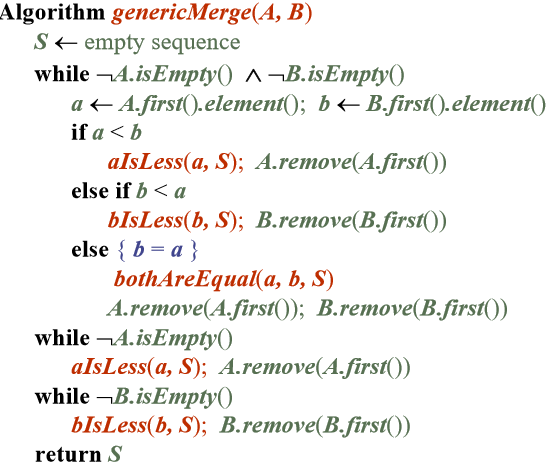
### Generisches Mischen

Mischen von zwei Listen A und B.

genericMerge()

aIsLess, bIsLess, bothAreEqual

O(nA+nB)



***Durchschnitt***: Element copy when in A and B

***Vereinigung***: copy all elements no duplicates

Laufzeit: O(n)

## multisets

*******Set,* ***mit Duplikaten***

## Multimap

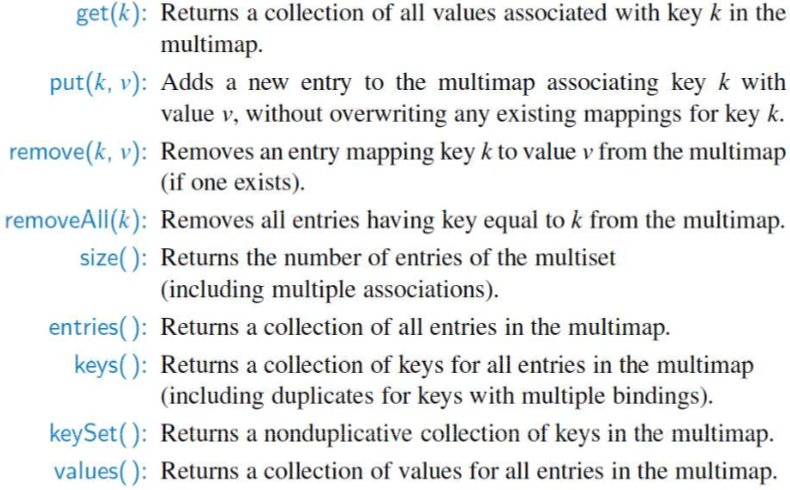
Ein Bild, das Shoji, Gebäude enthält.

Mit sehr hoher Zuverlässigkeit generierte BeschreibungMap, zu ***einem Key mehrere Values***

Ansatz 1: Datenstruktur anpassen

Ansatz 2: Value ist Collection Map<K,List<V>>

### multimap adt



### Java implementation

