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 ***Adaptier-barkeit***(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 <u>adaptiert</u>
- 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

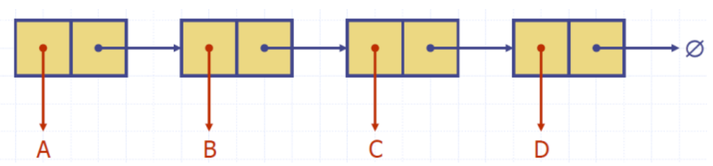
```
Arrays.equals(array, copy);          // true;
Arrays.sort(copy);                   //sortieren
Arrays.equals(array, copy);          //false
Arrays.toString(copy);               //[4,5,23]
```

LINKED-LISTS

GRUNDLAGEN

Einfach verkettete Liste. Sequenz aus Knoten.
Jeder Knoten besitzt:

- Ein Element → den Inhalt
- Ein Link zum nächsten Knoten



LINKED LIST BEISPIEL OHNE GENERICS

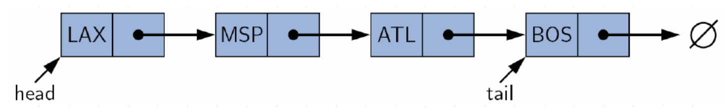
```
public class MyNode {
private Element element;
private MyNode next;
public MyNode(Element e, MyNode n) {
element = e;
next = n; }
public class Element {
private String value;
private int nr;
```

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
- tail 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
```

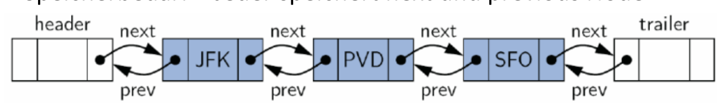
ENTFERNEN DES LETZTEN KNOTENS – O(1)

```
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)
- Speicherbedarf – Jeder speichert next und previous Node

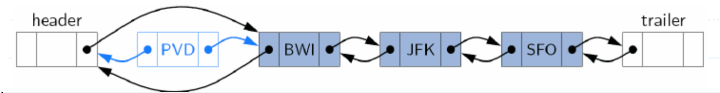


- Header und Trailer spezielle Knoten → ***Sentinels***
- Header und Trailer sind Start-Knoten für die Suche

```
public class DList {
protected int size;
protected DNode header,trailer; // sentinels … }
public class DNode {
protected String element;
protected DNode next, prev; … }
```

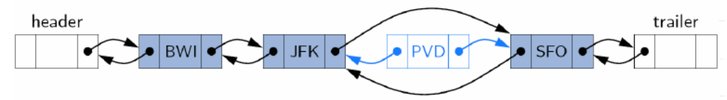
EINFÜGEN DES ERSTEN KNOTENS – O(1)

```
addFirst(v)                  //Methodenaufruf
```



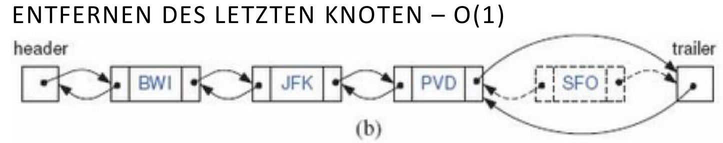

```
w← header.getNext()      //alter erster Knoten merken
v.setNext(w)             //alter 1. Node als next setzen
v.setPrev(header)        //header als previous setzen
w.setPrev(v )            //neuer Knoten als previous
header.setNext(v)        //erster Node next bei header
size = size +1           //Anzahl Nodes um eins erhöhen
```

BESTIMMTER KNOTEN EINFÜGEN – O(1)



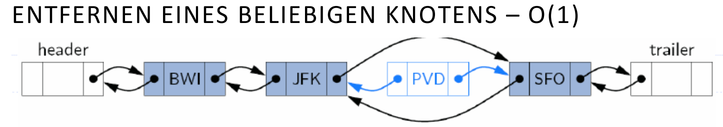

```
addAfter(u, v)           //v nach u einfügen
w← u.getNext( )          //Node nach u speichern
v.setPrev(u)             // u als previous setzen
v.setNext(w)             // w als next setzen
w.setPrev(v)             // v previous von w setzen
u.setNext(v)             // v next von u setzen
size = 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
```

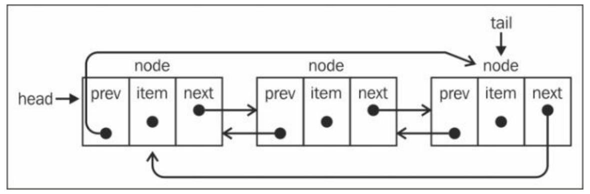
ENTFERNEN EINES BELIEBIGEN KNOTENS – O(1)




```
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

ARITHMETISCHE FOLGEN

Logarithmen:

$\log_b(xy) = \log_b x + \log_b y$

$\log_b(x/y) = \log_b x - \log_b y$

$\log_b x^a = a \cdot \log_b x$

$\log_b a = \log_x a / \log_x b$

- **Exponentiale:**

$a^{(b+c)} = a^b a^c$

$a^{(b-c)} = a^b / a^c$

$a^{bc} = (a^b)^c$

$b = a^{\log_a b}$

$b^c = a^{c \cdot \log_a b}$

Wobei Explizit: $a_n = a_1 + d(n-1)$

BSP: 6-10-14-18

- **Rekursiv:** $a_n = a_{n-1} + 4$ $a_1=6$
- **Iterativ:** $a_n = 6 + \sum_{i=2}^{n} 4$
- **Explizit:** $a_n = 6 + 4(n-1) = 6 + 4n - 4 = 4n + 2$

ARITHMETISCHE REIHEN / SUMMENFORMELN

Allgemeine Summenformel: $s_n = \frac{n(a_1 + a_n)}{2}$

Für a_n explizite Formel(arithmetische Reihe) einfügen.

- **Rekursiv:** ***s_1 = Anfangszahl, $s_n = s_{n-1} + a_n$***
- **Iterativ:** $s_n = \sum_{i=1}^{n} a_n$ **// i als Index verwenden!**
- **Explizit:** $s_n = \frac{n(a_1+a_n)}{2}$

BSP: 6-10-14-18

- **Rekursiv:** ***$s_1 = 6$, $s_n = s_{n-1} + a_n = s_{n-1} + 4n + 2$***
- **Iterativ:** $s_n = \sum_{i=1}^{n} 4i + 2$
- **Explizit:** $s_n = \frac{n(6+4n+2)}{2} = \frac{4n^2+8n}{2} = 2n^2 + 4n$

ALLGEMEINE SUMMENFORMEL FÜR BIG O

Die allgemeine Summenformel:

$$\sum a_i = a_1 m + \frac{m(m-1)}{2} d$$

wobei: m = Anzahl Summanden in der Summe
d = Abstand zwischen den Summanden

Aufgabe 6 (O-Notation)
Bestimmen Sie das Laufzeitverhalten (Anzahl Multiplikationen und Additionen) für die Berechnung des Produktes zweier $n \times n$ Matrizen. Begründen Sie Ihre Aussage.

Allgemeine Berechnung:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Beispiel:

$$\begin{pmatrix} 3 & 4 \\ 2 & 5 \end{pmatrix} \cdot \begin{pmatrix} 6 & 1 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 3\cdot6+4\cdot7 & 3\cdot1+4\cdot8 \\ 2\cdot6+5\cdot7 & 2\cdot1+5\cdot8 \end{pmatrix} = \begin{pmatrix} 46 & 35 \\ 47 & 42 \end{pmatrix}$$

Geben Sie die Laufzeit in *expliziter* Form (als Polynom) und in der *O-Notation* an.

Lösung:
Jede der n Zeilen in der ersten Matrize wird mit jeder der n Spalten der zweiten Matrize verrechnet. Ein solcher Schritt benötigt n Multiplikationen und $n-1$ Additionen, er ist also linear mit $O(n)$ wodurch die Laufzeit der gesamten Prozedur kubisch ist.
Konkret also:
$n^2(n+(n-1)) = 2n^3 - n^2 \in O(n^3)$

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**
- **Quadratisch ≈ n^2**
- **Logarithmisch ≈ log n**
- **Qubisch ≈ n^3**
- **Exponentiell ≈ 2^n**

BENÖTIGTE MATHEMATIK

LAUFZEITVERHALTEN

Das Laufzeitverhalten ist nicht beeinflusst von:

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

BEISPIELE

100n + 1000 → lineare Funktion
$10n^2$ + 100n + 1000 → quadratische Funktion

BIG-OH NOTATION

$f(n)$ ist $O(g(n))$ $\quad (f \in O(g))$

BEISPIELE BIG-OH NOTATION

- 2n + 10 ist O(n)	- $2n^2$ + 4n + 3 ist O(n^2)
- $3n^3$ + 2n2 + n ist O(n^3)	- 2 log n + 4 ist O(log n)

REGELN

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

ASYMPTOTISCHE ALGORITHMUS ANALYSE

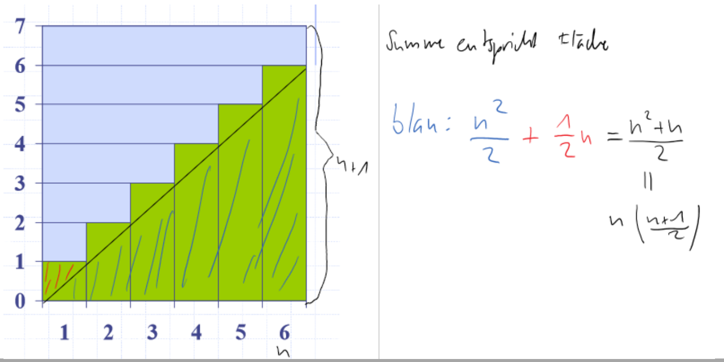
Die asymptotische Analyse eines Algorithmus bestimmt das Laufzeitverhalten in der big-Oh Notation.

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

Pseudocode Details

- Kontrolfluss
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Einrücken, statt Klammern
- Methoden Deklaration
  Algorithm *method* (*arg* [, *arg*…])
  Input … **Output** …
- Methoden Aufruf
  var.method (*arg* [, *arg*…])
- Kommentare
  { das ist ein Kommentar }
- Rückgabewert
  return *expression*
- Ausdrücke — Java
  - ← Zuweisung — =
  - = Gleichheit — ==
  - ≠ Ungleichheit — !=
  - ≤ Kleiner gleich — <=
  - ≥ Grösser gleich — >=
  - ∧ AND — &&
  - ∨ OR — ||
- n^2 Superskripts und andere mathematische Formeln und Formattierungen

ARITHMETISCHE PROGRESSION (N2 + N) / 2



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 auxi-
liary, 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

```
//Fibonacci 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; }
//Fibonacci 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

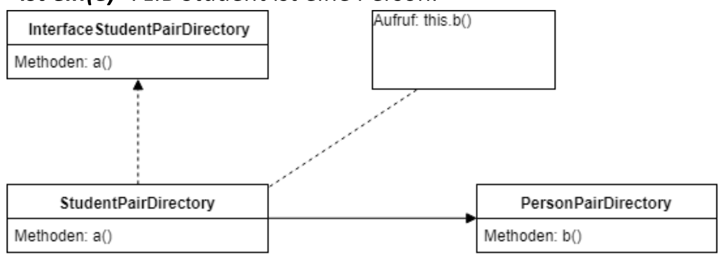
Klasse mit ähnlicher Funktionalität aber anderer Schnittstelle/API

LÖSUNGEN

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

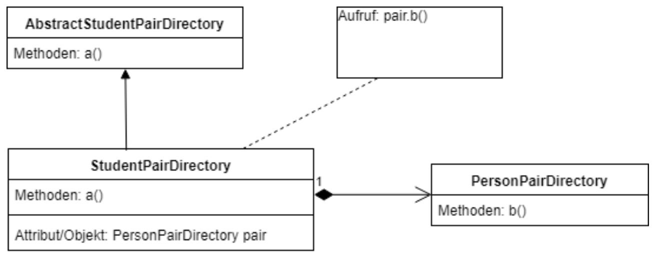
VERERBUNG(KLASSEN-ADAPTER)

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



+	-
+ Ändert einige Methoden der existierenden Klasse und lässt die anderen unverän-dert. + Kein zusätzliches Objekt	- Das ganze Interface der exis-tierenden Klasse ist sichtbar. - Nicht geeignet, um alle Un-terklassen 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 existieren-den Klasse können sehr ein-fach Adaptiert werden	- Jede verwendete Methode der existierenden Klasse muss in der Adapter-Klasse defi-niert 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.

STACK ADT – O(1)

Stack Methode	Singly-Linked-List Methode
size()	list.size()
isEmpty()	list.isEmpty()
push(element)	list.addFirst(element)
pop()	list.removeFirst()
top()	list.first()

- 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)
- Maximale Grösse ist durch Array-Grösse gegeben

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 IllegalStateExcep-
tion {
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 SinglyLinke-
dList<>();
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

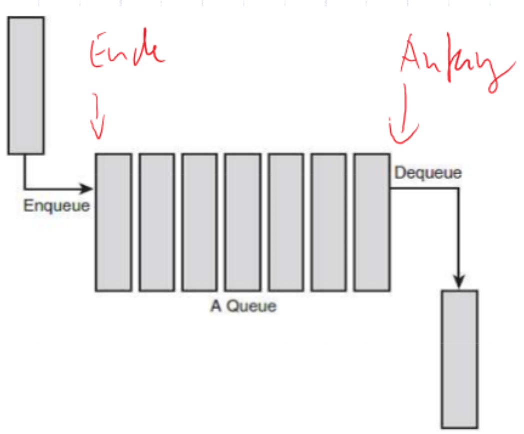
```
public static boolean symbolMatching(char[] klammern) {
Stack<Character> stack = new Stack<>();
for (int i = 0; i < klammern.length; i++) {
        if (klammern[i] == '{' || klammern[i] == '[') {
                stack.push(klammern[i]);}
        if (klammern[i] == '}' || klammern[i] == ']') {
                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;}
```

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]);}
```

QUEUES

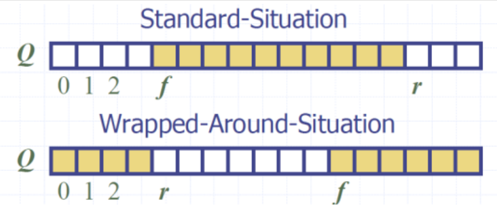
QUEUE ADT



- 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()

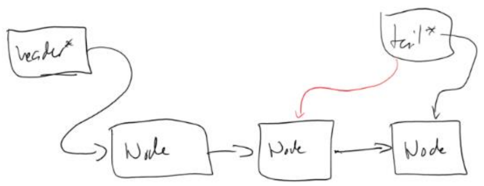
ARRAY-BASIERTE QUEUE

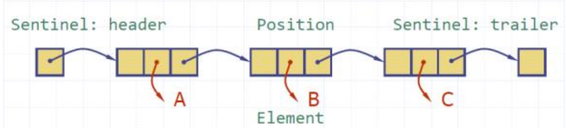


- Benutzung eines Array der Länge ***N*** auf zirkuläre Art und Weise
- Zwei Variabeln gebraucht
  1. ***f*** – Anfang/Front der Queue
  2. ***sz*** – Anzahl der gespeicherten Elemente
- der erste leere Slot wird bestimmt durch:
  - ***r = (f + sz) mod N*** ***//rear***

QUEUE OHNE FESTE GRÖSSE / NODEQUEUE

1. ***Array*** immer vergrössern(in grösseren Array kopieren: z.B mit Faktor 1.5 oder 2)
2. Mit ***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 IllegalStateExcep-
tion();
                } 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].get-
Name());                        if (next == arrayLen - 1)
{
                                        next = 0;
                                } else {
                                        next++;}}}}}
```

Java.UTIL.QUEUE

E **element**()
liefert das erste Element, ohne es zu entfernen.
Wirft *NoSuchElementException.*

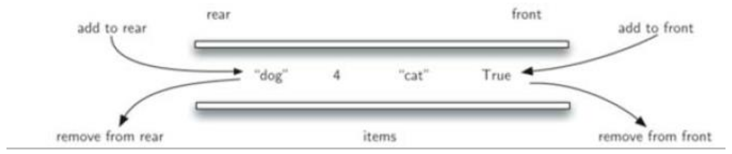
boolean **offer**(E o)
fügt das Element, wenn möglich, in die Queue ein.

E **peek**()
liefert das erste Element, ohne es zu entfernen (*null* wenn leer).

E **poll**()
liefert und entfernt das erste Element (*null* wenn leer).

E **remove**()
liefert und entfernt das erste Element.
Wirft *NoSuchElementException.*

DOUBLE-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:

```
public class DLNode<E> {

   private E element;
   private DLNode<E> next, prev;

   DLNode() { this(null, null, null); }

   DLNode(E e, DLNode<E> p, DLNode<E> n) {
     element = e;
     prev = p;
     next = n;
   }
   ...// setter und getter

 public interface Deque<E> {

   void addFirst(E element);
   void addLast(E element);

   E removeFirst();
   E removeLast();

   E first();
   E last();

   int size();
   boolean isEmpty();

 }
```

PERFOMANCE ANALYSE

Methode	O-Verhalten
size, isEmpty	O(1)
getFirst, getLast	O(1)
addFirst, addLast	O(1)
removeFirst, removeLast	O(1)

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

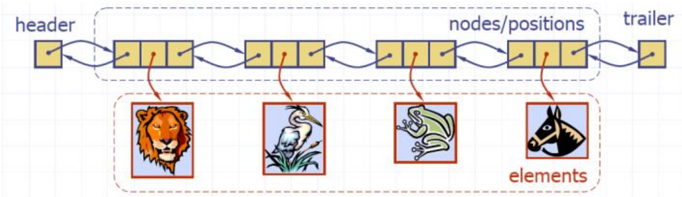
- Modelliert das ***Konzept «Platz/Position»*** in einer Datenstruktur.
- ***Pro Position*** wird ***ein einziges Objekt***(z.B ein Array-Element/Node) abgespeichert.
- Höhere ***Abstraktion*** + Beschränkung auf das ***Wesentliche***

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 IllegalArgu-
mentException;
Position<E> after(Position<E> p) throws IllegalArgumen-
tException;
Position<E> addFirst(E e);
Position<E> addLast(E e);
Position<E> addBefore(Position<E> p, E e) throws Ille-
galArgumentException;
Position<E> addAfter(Position<E> p, E e) throws Ille-
galArgumentException;
E set(Position<E> p, E e) throws IllegalArgumentExcep-
tion;
E remove(Position<E> p) throws IllegalArgumentExcep-
tion;
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 = ne-
wElement; }
```

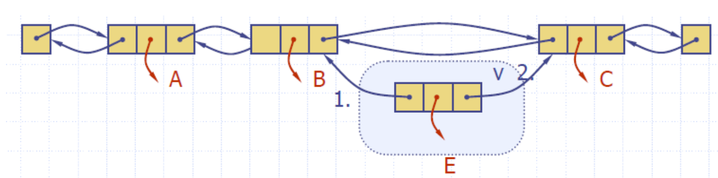
CLASS LINKED-POSITIONAL-LIST

```
public class LinkedPositionalList<E> implements Posi-
tionalList<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 Ille-
galArgumentException {
    if (!(p instanceof Node)) throw new IllegalArgumen-
tException("invalid p");
    Node<E> node = (Node<E>)p;
    if (node.getNext() == null) throw new IllegalArgu-
mentException("p is no longer on the list");
    return node;
}
```

POSITIONAL-LIST (ER)SETZEN

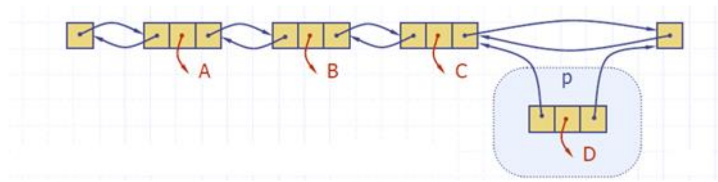
```
public E set(Position<E> p, E e) throws IllegalArgumen-
tException {
Node<E> node = validate(p);
E answer = node.getElement();
node.setElement(e);
return answer;}
```

POSITIONAL-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;}}
```

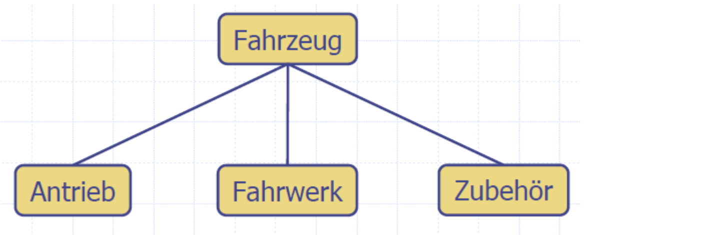
PERFOMANCE

- 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)

Operation	Array	List
size(), isEmpty()	1	1
atIndex(i), get(i)	1	*n*
first(), last(), prev(), next()	1	1
set(p,e)	1	1
set(i,e)	1	*n*
add(i, e), remove(i)	*n*	*n*
addFirst(e), addLast(e)	*n*	1
addAfter(p,e), addBefore(p,e)	*n*	1
remove(p)	*n*	1
indexOf(e)	*n*	*n*

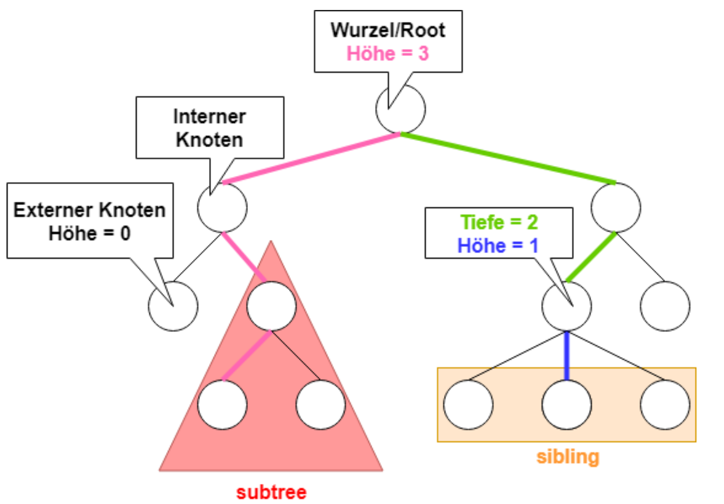
TREES – BÄUME

GRUNDLAGEN



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

TERMINILOGIE – BEGRIFFE



Wurzel(Root) – Knoten <u>ohne</u> Elternknoten(A)

- ***Interner Knoten*** – Knoten mit <u>mind.</u> <u>einem</u> Kind(z.B A oder B)
- ***Externer Knoten(Blatt)*** – Knoten ohne Kinder(z.B E oder K)
- ***Vorgängerknoten*** – Eltern, Grosseltern(alle wo Höhe größer)
- ***Tiefe*** – Anzahl Vorgänger
- ***Höhe eines Knoten***
  - Externe Knoten → 0
  - Interne Knoten → 1 + maximale Höhe Nachfolger-knoten
- ***Höhe eines Baumes*** – Höhe der Wurzel
- ***Nachfolger*** – Kind, Grosskind, etc.
- ***Subtree*** – Baum aus Knoten und seinen Nachfolgern
- ***Sibling:*** Zwillingsknoten

TREE ADT

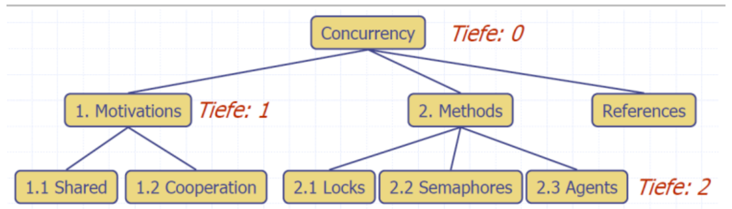
Der Position ADT dient in einem PositionTree als Abstraktion für die Knoten.

- 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

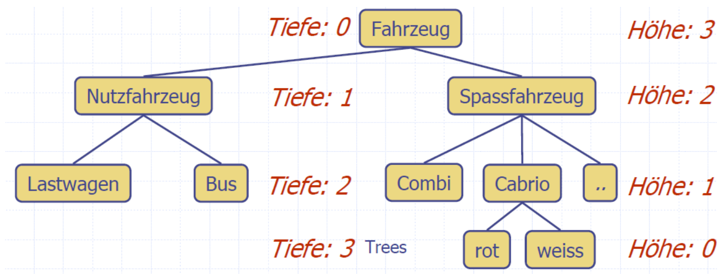
```
public interface Tree<E> extends Iterable<E> {
Position<E> root();
Position<E> parent(Position<E> p) throws IllegalArgu-
mentException;
Iterable<Position<E>> children(Position<E> p) throws Il-
legalArgumentException;
int numChildren(Position<E> p) throws IllegalArgumentEx-
ception;
boolean isInternal(Position<E> p) throws IllegalArgumen-
tException;
boolean isExternal(Position<E> p) throws IllegalArgumen-
tException;
boolean isRoot(Position<E> p) throws IllegalArgumentEx-
ception;
int size();
boolean isEmpty();
Iterator<E> iterator();
Iterable<Position<E>> positions();
```

TIEFE




```
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




```
public 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

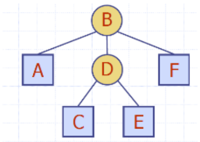
- ***n*** Anzahl Knoten
- ***e*** Anzahl externer Knoten
- ***i*** Anzahl interner Knoten
- ***h*** Höhe

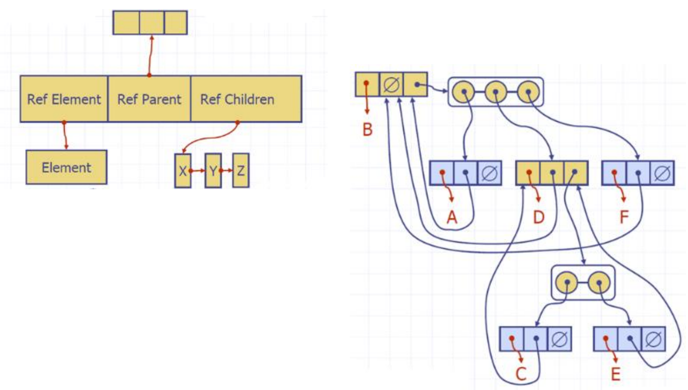
- $e = i + 1 \leftrightarrow i = e - 1$
- $n = i + e = i + i + 1 = 2i + 1 = 2e - 1$
- $e = (n + 1) / 2$
- $i = (n - 1) / 2$
- $e \leq 2^h$
- $h \geq \log_2 e$
- $h \geq \log_2 (n + 1) - 1$
- $n \leq 2^{h+1} - 1$
- $h \leq i$
- $h \leq (n - 1) / 2$
- $n \geq 2h + 1$

SPEICHERVERFAHREN FÜR BÄUME – LINKED LIST

Ein Baumknoten beinhaltet folgendes:

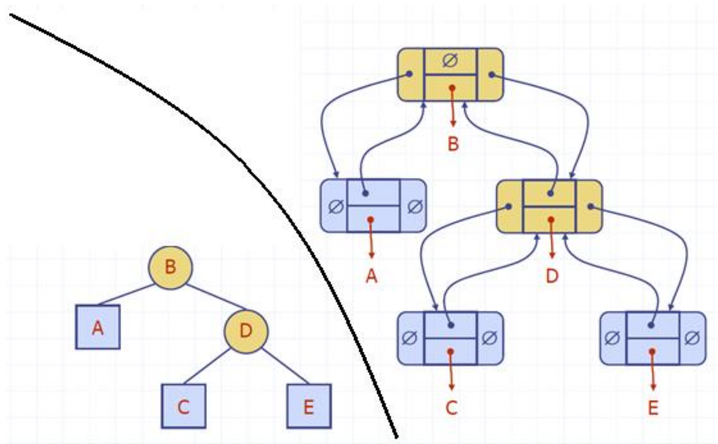
- Element
- Elternknoten
- Sequenz mit Kindknoten





→beliebig viele Children

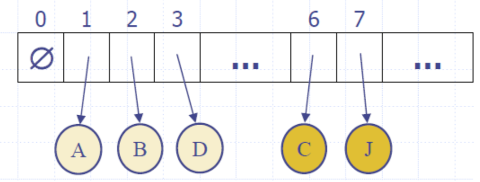
LINKED LIST FÜR BINARRYTREES

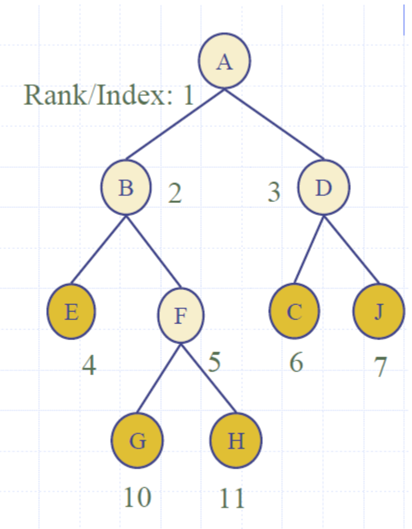


→max zwei Childs

SPEICHERVERFAHREN FÜR BÄUME – ARRAY BASIERT

Die Knoten werden in einem Array gespeichert:





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

- rank(root) = 1
- Falls node linkes Kind des parent(node): rank(node) = 2 * rank(parent(node))
- if node is the right child of parent(node): rank(node) = 2 * rank(parent(node))+1

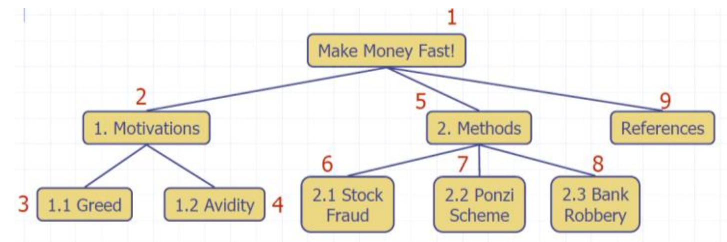
BAUM-TRAVERSIERUNGEN

Bei einer Baum-Traversierung werden alle Knoten eines Baumes auf systematisch Art und Weise besucht werden.
Dabei gibt es verschiedene Traversierungs-Algorithmen:

PREODER

In der Pre-Order Traversierung wird ein ***Knoten <u>vor</u> seinen Nachfolgern*** besucht*:*

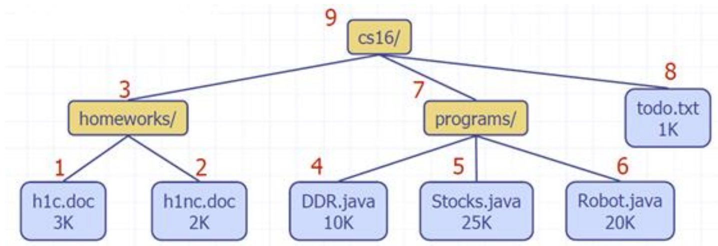
Algorithm *preOrder(v)*
    visit(v)
    for each child *w* of *v*
        preOrder (w)



POSTORDER

In einer Post-Order Traversierung wird ein ***Knoten <u>nach</u> seinen Nachfolgern*** besucht.

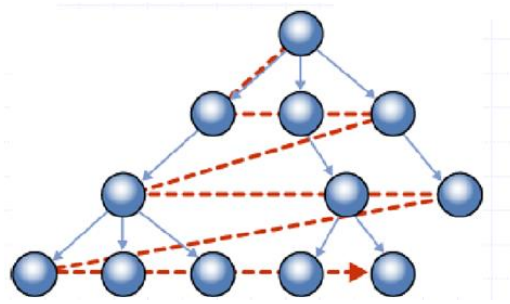
Algorithm *postOrder(v)*
    for each child *w* of *v*
        postOrder (w)
    visit(v)



BREADTH-FIST – BREITENSUCHEN

In einer Breadth-First Traversierung werden ***zuerst alle Knoten einer Tiefe t*** besucht, ***dann alle Knoten der Tiefe t+1***, t+2, usw.

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

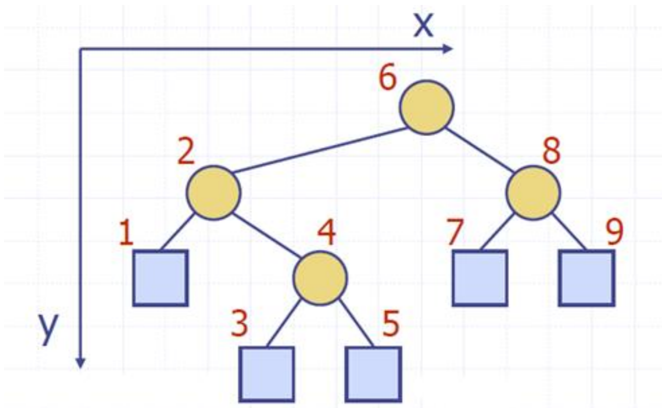


INORDER

In einer Inorder Traversierung wird ein Knoten ***nach seinem linken Subtree und vor seinem rechten Subtree*** besucht.

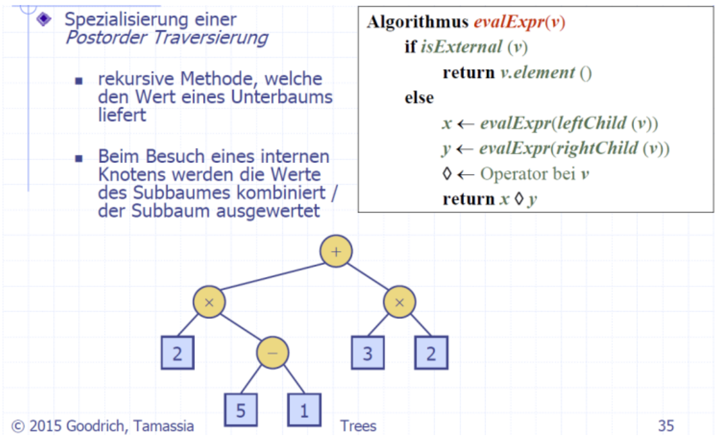
Algorithm *inOrder(v)*
    if *hasLeft(v)*
        inOrder(left(v))
    visit(v)
    if *hasRight(v)*
        inOrder(right(v))

x(v) = inorder Rang von v
y(v) = Tiefe von v

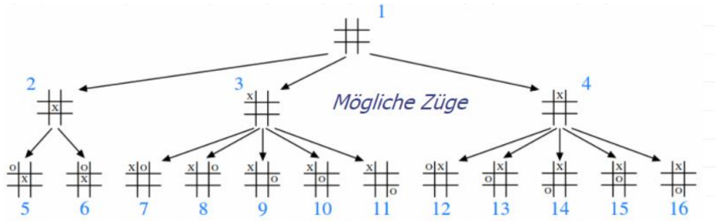


ANWENDUNGSBEISPIELE TRAVERSIERUNGEN

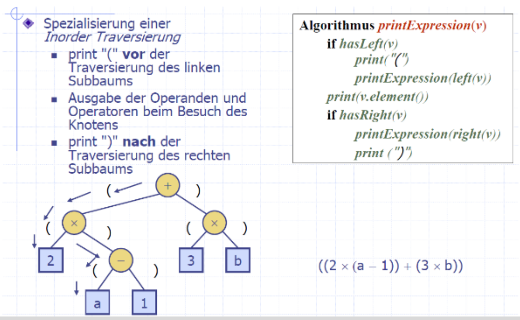
- Preorder →Arithmetische Ausdrücke:



- Postorder → benutzter Speicher in einem Verzeichnis inkl. Subverzeichnissen
- Breadth-First → Tic-Tac-Toe:

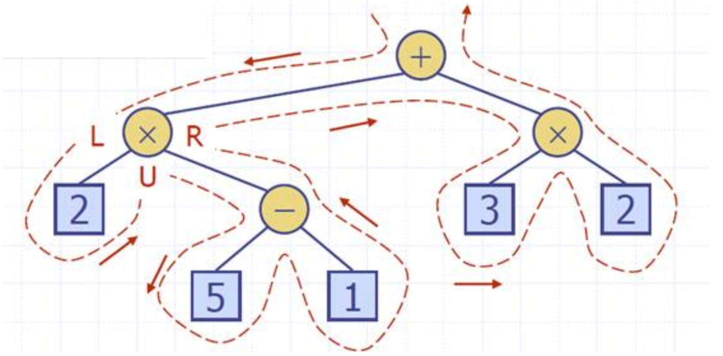


- Inorder→Arithmetische Ausdrücke:



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**



IMPLEMENTIERUNG

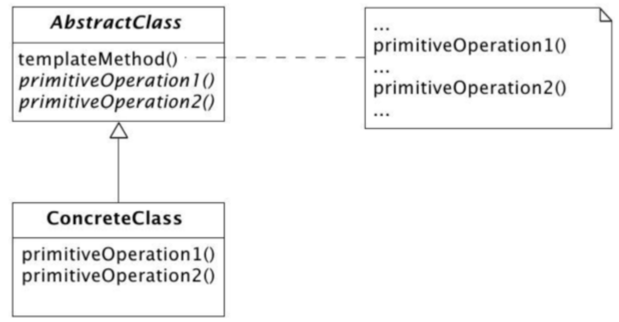
```
public abstract class EulerTour {
  protected BinaryTree tree;
  protected void visitExternal(Position p, Result r) { }
  protected void visitLeft(Position p, Result r) { }
  protected void visitBelow(Position p, Result r) { }
  protected void visitRight(Position p, Result r) { }
  protected Object eulerTour(Position p){
    Result r = new Result();
    if (tree.isExternal(p)) {
      visitExternal(p, r);
    } else {
      visitLeft(p, r);
      r.leftResult = eulerTour(tree.left(p));
      visitBelow(p, r);
      r.rightResult = eulerTour(tree.right(p));
      visitRight(p, r);
      return r.finalResult;
    } …
```

Visit-Methoden verfeinert in Subklassen:

```
public class EvaluateExpression extends EulerTour {

  protected void visitExternal(Position p, Result r) {
    r.finalResult = (Integer) p.element();
  }

  protected void visitRight(Position p, Result r) {
    Operator op = (Operator) p.element();
    r.finalResult = op.operation(
                          (Integer) r.leftResult,
                          (Integer) r.rightResult);
  }

  …
}
```

TEMPLATE METHOD PATERN(SCHABLONENMUSTER)

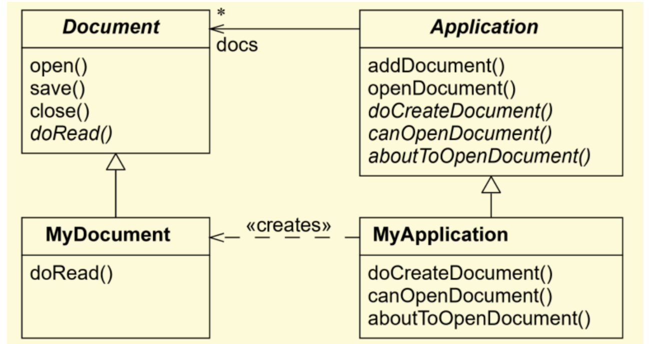
«Hollywood Prinzip» oder auch *«Don't call us, we will call you»*



- 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

MyDocument 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

Beispiel: *Thread 1* *Thread 2*

```
Iterator it =
  list.iterator()
it.hasNext():true
                      list.remove()
it.next()
```

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

JAVA.UTIL.ITERATOR

```
public interface Iterator<E> {

    boolean hasNext();
    E next() throws
              NoSuchElementException,
              ConcurrentModificationException;

    void remove() throws
                  UnsupportedOperationException,
                  IllegalStateException;
}
```

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.

```
public Iterator<E> iterator() {
  return new ElementIterator();
}
```

ELEMENT ITERATOR

```
private class ElementIterator implements Iterator<E> {

  Iterator<Position<E>> posIterator = new PositionIterator();

  public boolean hasNext() { return posIterator.hasNext(); }

  public E next() { return posIterator.next().getElement(); }
                   // return element!

  public void remove() { posIterator.remove(); }
}
```

POSITIONAL ITERATOR

```
private class PositionIterator implements
                            Iterator<Position<E>> {

  /** First Position of the containing list */
  private Position<E> cursor = first();

  /** A Position of the most recent element. */
  private Position<E> recent = null;
private class PositionIterator implements Iterator<Position<E>> {
    /**
     * Tests whether the iterator has a next object.
     * @return true if there are further objects, false otherwise
     */
    public boolean hasNext() { return (cursor != null);  }

    /**
     * Returns the next position in the iterator.
     * @return next position
     * @throws NoSuchElementException: no further elements */
    public Position<E> next() throws NoSuchElementException {
      if (cursor == null) throw new
                    NoSuchElementException("nothing left");
      recent = cursor;
      cursor = after(cursor);
      return recent;
    }
/**
   * Removes the element returned by most recent call to next.
   * @throws IllegalStateException if next has not yet beencalled
   * @throws IllegalStateException if remove was already called */
  public void remove() throws IllegalStateException {
    if (recent == null) throw
           new IllegalStateException("nothing to remove");
     LinkedPositionalList.this.remove(recent);
     // remove from outer list
     recent = null;
     // do not allow remove again until next is called
  }
```

JAVA.UTIL.LISTITERATOR

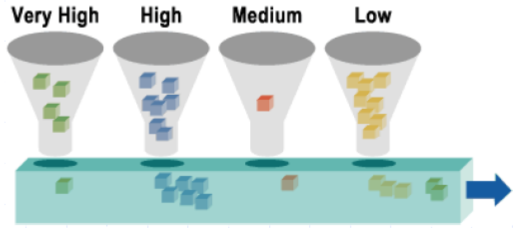
Zusätzliche Funktionen:

• ***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()}

PRIORITY QUEUES



Collection von Einträgen/Entries

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

ENTRY ADT

Entry besteht aus ***Schlüssel-Wert-Paar***(Key-Value)

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

```java
public interface Entry<K,V> {
    public K key();
    public V 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 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***:

compare(a, b): liefert einen Integer i, so dass:

- $i < 0$ falls $a < b$
- $i = 0$ falls $a = b$
- $i > 0$ falls $a > b$

BEISPIEL

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()***

Lexikographischer Vergleich von 2-D Punkten:

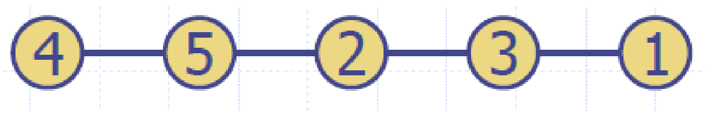
```java
/** Comparator für 2D Punkte bei Standard
    lexikographischer Ordnung. */
public class Lexicographic implements
    Comparator {
  int xa, ya, xb, yb;
  public int compare(Object a, Object b)
   throws ClassCastException {
    xa = ((Point2D) a).getX();
    ya = ((Point2D) a).getY();
    xb = ((Point2D) b).getX();
    yb = ((Point2D) b).getY();
    if (xa != xb)
         return (xa - xb);
    else
         return (ya - yb);
  }
}
```

Point Objekte:

```java
/** Class Point : in der Ebene mit
    Integer Koordinaten */
public class Point2D       {
  protected int x, y; // Koordinaten
  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() {
       return x;
  }
  public int getY() {
       return y;
  }
}
```

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
   a. 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

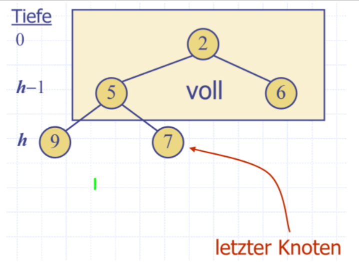
IN-PLACE INSERTION SORT

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



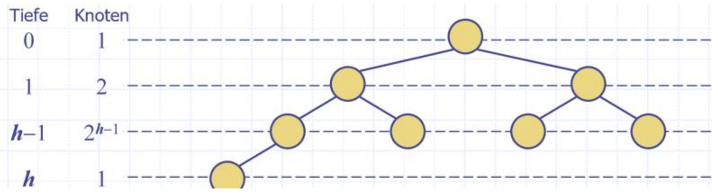
Ein Heap ist ein ***Binärbaum, welcher in seinen Knoten Schlüssel speichert*** und folgende Eigenschaft besitzt: Jeden Knoten v, welcher nicht Wurzel ist gilt:
key(v) ≥ key(parent(v))

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

EIGENSCHAFTEN

Zusätzlich hat ein Heap folgende Eigenschaften:
1. Es sind ***2^i-Knoten auf der Tiefe i*** vorhanden(***volle Levels***)
2. Es wird von ***links her aufgefüllt***
   a. 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.

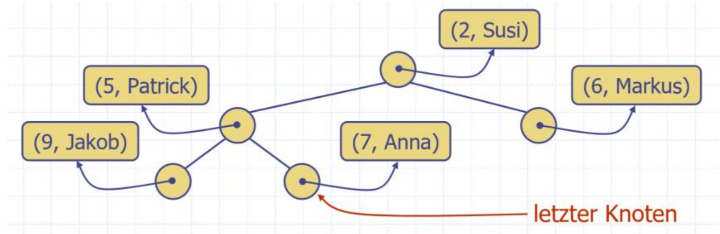
→Sei h die Höhe des Heaps mit n Knoten



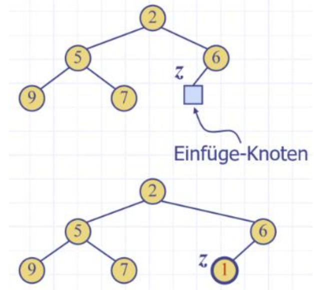
Obere Grenze	**Untere Grenze**
$n = 1 + 2 + 4 + \dots + 2^h = \mathbf{2^{h+1} - 1}$	$\mathbf{n = 2^h}$
h = log(n+1) -1	**h = log(n)**

HEAPS & PRIORITY QUEUE

Mithilfe eines Heaps kann eine Priority Queue implementiert werden:
- Jeder Knoten speichert einen Entry<Key, Element> ab
- Der letzte Knoten wird speziell gemerkt



EINFÜGEN VON ENTRIES IN EINEN HEAP

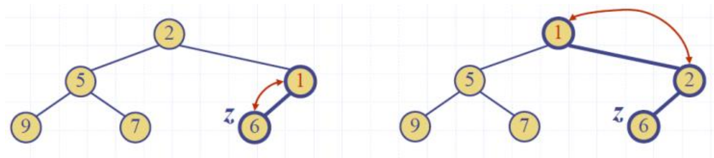


Die insert()-Methode entspricht dem Einfügen eines Schlüssels k in den Heap.

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:

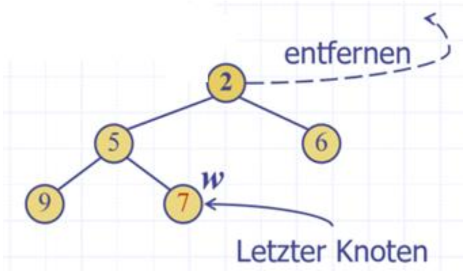


→ Der neu eingefügte Knoten k wird so lange mit dem darüberliegenden Knoten vertauscht, bis der neue Knoten ***entweder die Wurzel*** erreicht hat, oder der ***Elternknoten einen kleineren oder gleichgrossen*** Schlüssel hat.

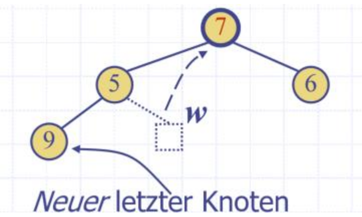
Da der Heap eine Höhe von log(n) benötigt der Algo. ***O(log(n))***

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. Entfernen des letzten Knotens(w)



4. Wiederherstellen 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.

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

HEAP-SORT

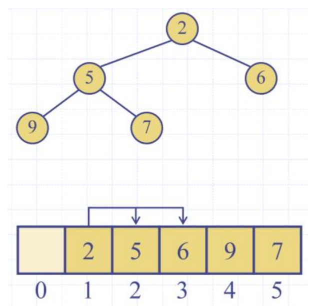
Gegeben: Priority Queue mit n Einträgen mithilfe eines Heaps:
- Speicherbedarf → O(n)
- insert() & removeMin()→O(log(n)
- size((, isEmpty() und min() → O(1)

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: $1000^2 = 10^6$
Heap-Sort: 1000 log(1000) = 10^4
Bei 1000 Elementen ist der Heap-Sort 100x schneller

VECTOR-BASIERTE HEAP-IMPLEMENTIERUNG



Ein 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

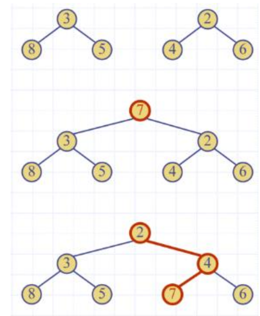
Das Entfernen von Knoten wird mit der Operation removeMin() erreicht und entspricht dem ***Entfernen bei Index 1.***

INPLACE SORTIERUNG

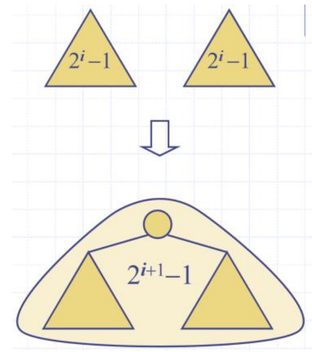
Diese Datenstruktur führt automatisch zu einer inplace-Sortierung.

ZWEI HEAPS ZUSAMMENFÜHREN

Gegeben: zwei Heaps und ein neuer Schlüssel k:
1. K als Wurzel-Schlüssel + zwei Unterbäume
2. Downheap()



BOTTOM-UP HEAP-KONSTRUKTION



Ein 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 2^i-1 Schlüsseln und fügt diese zu einem Heap mit 2^{i+1}-1 Schlüssel zusammenAnzahl Heaps: $(n+1)/2^i$
1. (n+1)/2 Knoten einfügen
2. (n+1)/4 Knoten einfügen
3. (n+1)/8 Knoten einfügen usw.

Bottom-up benötigt ***O(n)*** Zeit.

Bottom-up Heap Konstruktion ist schneller als ***n*** aufeinanderfolgende Einfügen-Operationen in ***O(log(n))*** und beschleunigt in der ersten Phase den Aufbau eines Heaps für einen Heap-Sort.

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

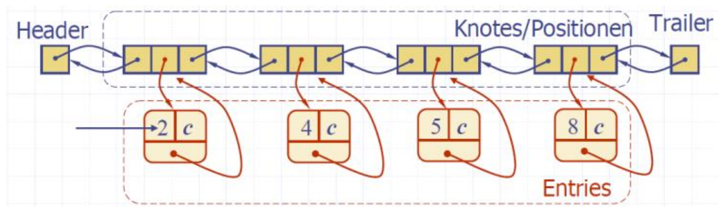
LOCATION-AWARE ENTRIES

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



Eine Lokations-bewusste Listen-Entry ist ein Objekt, welches folgende Informationen abspeichert:

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

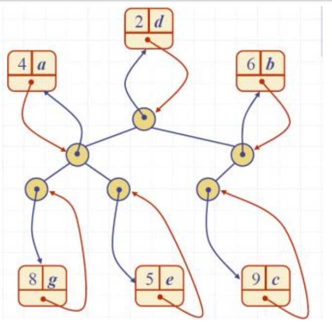
Die Position(oder Array-Zelle) speichert die Entry
Read-only-Operationen(niemand Zugriff auf interne Datenstrukt.)

HEAP IMPLEMENTIERUNG

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***.



PERFORMANCE

Methode	Unsortierte Liste	Sortierte Liste	Heap
size, isEmpty	O(1)	O(1)	O(1)
insert	O(1)	O(n)	O(log n)
min	O(n)	O(1)	O(1)
removeMin	O(n)	O(1)	O(log n)
remove	*O(1)*	*O(1)*	*O(log n)*
replaceKey	O(1)	O(n)	*O(log n)*
replaceValue	O(1)	O(1)	O(1)

MAPS

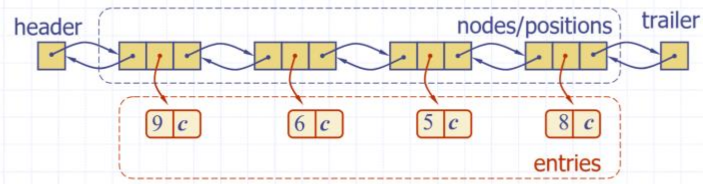
ADT

Eine Map modelliert eine durchsuchbare Collection von Schlüs-sel-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 zu-rück
  - ***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

```
Algorithmus get(k):

B = S.positions()      { B ist Iterator der Positions in S }
while B.hasNext() do
   p = B.next()        { nächste Position in B }
   if p.element().key() = k then
       return p.element().value()     //Wenn gefunden zurückgeben
return null            { es gibt keine Entry zum Schlüssel k }
```

PUT(K,V)-ALGORITHMUS

```
Algorithmus put(k,v):

B  = S.positions()
while B.hasNext() do
   p = B.next()
   if p.element().key() = k  then
        t = p.element().value()
        S.set(p,(k,v)) //Neuen Wert setzen
        return t       {Rückgabe des alten Wertes}
S.addLast((k,v))  // Wenn nicht vorhanden => hinten hinzufügen.
n = n + 1    {n speichert die Anzahl gespeicherte Einträge}
return null  {es gab keinen Eintrag mit Schlüssel k}
```

REMOVE(K)-ALGORITHMUS

```
Algorithmus remove(k):

B = S.positions()
while B.hasNext() do
   p = B.next()
   if p.element().key() = k  then
        t = p.element().value()
        S.remove(p)  //Wenn Element vorhanden => remove
        n = n − 1     {Anzahl Einträge nimmt um 1 ab}
        return t      {Rückgabe des entfernten Werts}
return null           {es gibt keinen Eintrag zum Schlüssel k}
```

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

$$h(x) = x \bmod N$$

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

AUFBAU

Diese Funktionen sind meistens in zwei Teile aufgeteilt:

1. ***Hash-Code: h_1: Keys → Integer***
   a. Schlüssel möglichst zufällig verteilen
2. ***Kompressfunktion: h_2:Integers → [0, N-1]***
   a. ***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:

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

  - Fixer Wert z, Overflow ignoriert
  - Sehr gut für Strings

BEISPIEL POLYNOM-AKKUMULATION

Länge des Strings

*s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]*

Bei s=»ab» → 97 * 31 + 98 = 3105
a ASCII = 97, b ASCII = 98

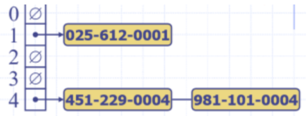
KOMPRESS FUNKTIONEN

- ***Division***
  - $h_2(y) = y \bmod N$
  - N oft Primzahl(Zahlentheorie)
- ***Multiply, Add, Divide(MAD)***
  - $h_2(y) = (ay + b) \bmod N$
  - 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

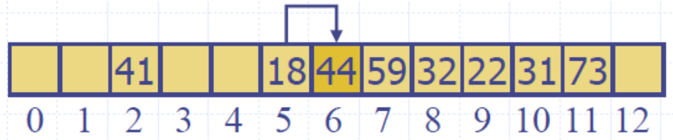


- Behälter sind ***verkettete Listen***
- Jede Zelle der Tabelle zeigt auf eine Liste
- Unbegrenzt
- Selten Überläufer

+ 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
  - $s(k,2)=h(k)+2^2=h(k)+4$,
  - $s(k,3)=h(k)+3^2=h(k)+9$
- ***Alternierendes Sondieren***
  - Es wird abwechselnd davor und dahinter gesucht
  - Vorwärts dann rückwärts(+ dann -)
- ***Alternierendes quadratisches Sondieren***
  - Wie Alternierendes Sortieren, jedoch Schritte quadratisch
  - $s(k,2)=h(k)+2^2= h(k)+4$
  - $s(k,3)=h(k)-3^2=h(k)-9$
- Zufälliges Sondieren
  - Zufallsfunktion implementiert

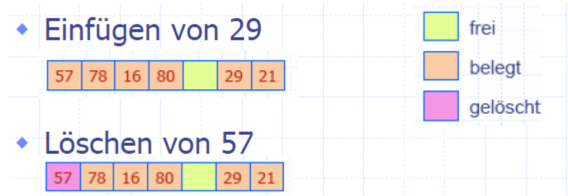
SUCHE MIT LINEARER SONDIERUNG

1. Bei Zelle h(k) wird gestartet
2. Aufeinanderfolgende Zellen durchsuchen, bis:
   a. Eintrag zum Schlüssel k gefunden
   b. Leere Zelle gefunden
   c. 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

	lin. Sondieren		quadr. Sondieren	
α	suche⁺	suche⁻	suche⁺	suche⁻
0.5	1.5	2.5	1.44	2.19
0.9	5.5	50.5	2.85	11.4
0.95	10.5	**200.5**	3.52	22.05
	+	-	+	-

+ erfolgreiche Suche
- Erfolglose Suche

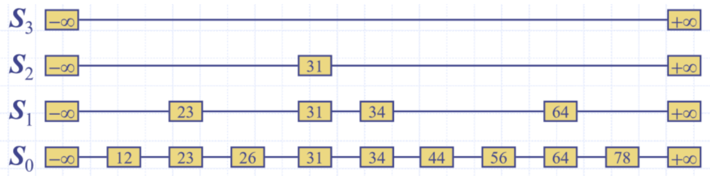
A = n/N
Zahl und Länge der Sondierungsketten(cluster) kleinhalten.
Faustregel Lastfaktor von 0.8 nicht überschritten

SKIPLISTE

GRUNDLAGEN & DEFINITION

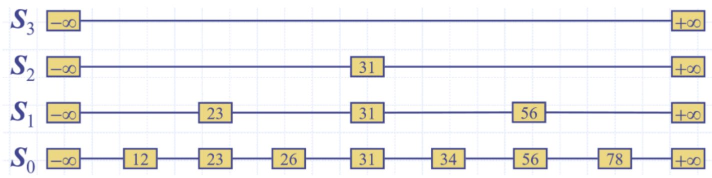
Eine Skip Liste für ein Set S unterschiedlicher (Key, Element)-Paare besteht aus einer ***Serie von ListenI*** S_0, S_1, ..., s_h so dass:

- Jede Liste s_i einen künstlichen Anfangs-(-∞) und End-Knoten(+∞) haben. Die Liste S_h hat nur diese.
  - Diese sind eindeutig & leicht zu prüfen
- S_0 alle Keys von S in aufsteigender Reihenfolge(Teilmengen)



PERFEKTE SKIPLISTE

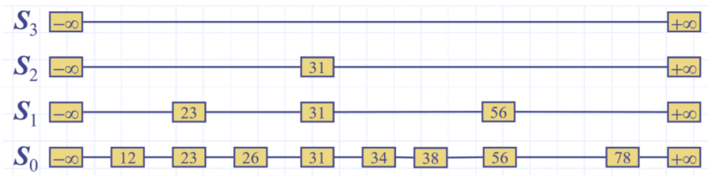
Man nennt eine Skipliste perfekt, wenn jede Liste jeweils ***Knoten in der Mitte der Intervalle der Nachfolgerliste*** enthält.



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

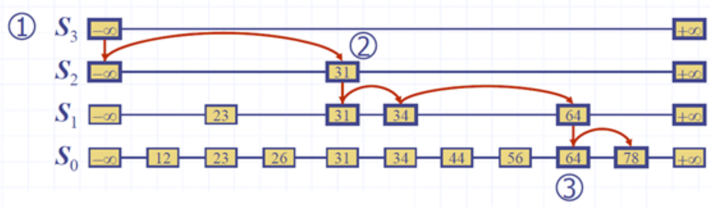
RANDOMISIERTE SKIPLISTE

Man 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.
4. 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

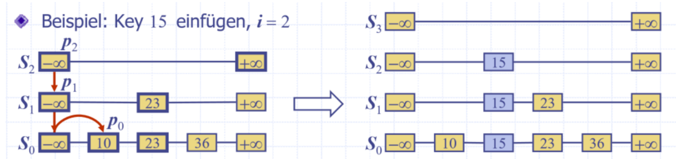
Mithilfe eines randomisierten Algorithmus Eintrag(k,o) einfügen:

1. Erfolglose Suche nach k und ***merke p_0*** 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. Erzeuge Zufalls-Höhe des neuen Turmes(Wahrscheinlichkeit lö1/2^{i+1})

```
public int randheight () {
  /* liefert eine zufällige Höhe zwischen 0 und
     maxHeight */
  height = 0;
  while (rand () % 2 == 0 && height < maxHeight)
      height++;
  return height;
}
```

$$P(\text{randheight} = i) = \frac{1}{2^{i+1}}$$

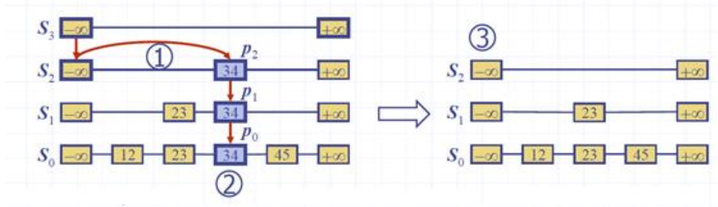
4. Füge p in alle Listen ein, bei welcher die Höhe niedriger sind als die gewählte Höhe(0<i<Höhe)



ENTFERNEN EINER ENTRY

Entry mit Schlüssel x aus Skip Liste entfernen:

1. Positionen p_0, p_1 ,... ,p_i 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

Unsortierte Collection von Elementen, ***ohne Duplikate***

- add(e): Adds the element e to S (if not already present).
- remove(e): Removes the element e from S (if it is present).
- contains(e): Returns whether e is an element of S.
- iterator( ): Returns an iterator of the elements of S.

$$S \cup T = \{e:\ e \text{ is in } S \text{ or } e \text{ is in } T\},$$
$$S \cap T = \{e:\ e \text{ is in } S \text{ and } e \text{ is in } T\},$$
$$S - T = \{e:\ e \text{ is in } S \text{ and } e \text{ is not in } T\}.$$

- addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.
- retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.
- removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

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
$O(n_A+n_B)$

```
Algorithm genericMerge(A, B)
    S ← empty sequence
    while ¬A.isEmpty() ∧ ¬B.isEmpty()
        a ← A.first().element(); b ← B.first().element()
        if a < b
            aIsLess(a, S); A.remove(A.first())
        else if b < a
            bIsLess(b, S); B.remove(B.first())
        else { b = a }
            bothAreEqual(a, b, S)
            A.remove(A.first()); B.remove(B.first())
    while ¬A.isEmpty()
        aIsLess(a, S); A.remove(A.first())
    while ¬B.isEmpty()
        bIsLess(b, S); B.remove(B.first())
    return S
```

Durchschnitt: Element copy when in A and B
Vereinigung: copy all elements no duplicates
Laufzeit: O(n)

MULTISETS

*Set, **mit Duplikaten*** { a, ***b***, ***b***, c, d, e, ***f***, ***f*** }

MULTIMAP

Map, zu ***einem Key mehrere Values***
Ansatz 1: Datenstruktur anpassen
Ansatz 2: Value ist Collection Map<K,List<V>>

Schloss	*Gebäude*
Schloss	*Verriegelung*

MULTIMAP ADT

- get(k): Returns a collection of all values associated with key k in the multimap.
- put(k, v): Adds a new entry to the multimap associating key k with value v, without overwriting any existing mappings for key k.
- remove(k, v): Removes an entry mapping key k to value v from the multimap (if one exists).
- removeAll(k): Removes all entries having key equal to k from the multimap.
- size( ): Returns the number of entries of the multiset (including multiple associations).
- entries( ): Returns a collection of all entries in the multimap.
- keys( ): Returns a collection of keys for all entries in the multimap (including duplicates for keys with multiple bindings).
- keySet( ): Returns a nonduplicative collection of keys in the multimap.
- values( ): Returns a collection of values for all entries in the multimap.

JAVA IMPLEMENTATION

```java
public class HashMultimap<K,V> {
  Map<K,List<V>> map = new HashMap<>();   // the primary map
  int total = 0;                          // total number of entries in the multimap
  /** Constructs an empty multimap. */
  public HashMultimap() { }
  /** Returns the total number of entries in the multimap. */
  public int size() { return total; }
  /** Returns whether the multimap is empty. */
  public boolean isEmpty() { return (total == 0); }
  /** Returns a (possibly empty) iteration of all values associated with the key. */
  Iterable<V> get(K key) {
    List<V> secondary = map.get(key);
    if (secondary != null)
      return secondary;
    return new ArrayList<>();             // return an empty list of values
  }
 /** Adds a new entry associating key with value. */
 void put(K key, V value) {
   List<V> secondary = map.get(key);
   if (secondary == null) {
     secondary = new ArrayList<>();
     map.put(key, secondary);      // begin using new list as secondary structure
   }
   secondary.add(value);
   total++;
 }
 /** Removes the (key,value) entry, if it exists. */
 boolean remove(K key, V value) {
   boolean wasRemoved = false;
   List<V> secondary = map.get(key);
   if (secondary != null) {
     wasRemoved = secondary.remove(value);
     if (wasRemoved) {
       total−−;
       if (secondary.isEmpty())
         map.remove(key);          // remove secondary structure from primary map
     }
   }
   return wasRemoved;
 }
  /** Removes all entries with the given key. */
  Iterable<V> removeAll(K key) {
    List<V> secondary = map.get(key);
    if (secondary != null) {
      total −= secondary.size();
      map.remove(key);
    } else
      secondary = new ArrayList<>();     // return empty list of removed values
    return secondary;
  }
  /** Returns an iteration of all entries in the multimap. */
  Iterable<Map.Entry<K,V>> entries() {
    List<Map.Entry<K,V>> result = new ArrayList<>();
    for (Map.Entry<K,List<V>> secondary : map.entrySet()) {
      K key = secondary.getKey();
      for (V value : secondary.getValue())
        result.add(new AbstractMap.SimpleEntry<K,V>(key,value));
    }
    return result;
  }
}
```