

* Wertebuchse:

为什么这个数?

— 1位符号位 + (n-1) 统计数位.

byte: 8 bit. $-128 \rightarrow 127$

short: 16 bit. $-32.000 \rightarrow 32.000$

int : 32 bit. $-2\text{ Milliarden} \rightarrow 2\text{ Milliarden}$

long: 64 bit. $-2^{63} \rightarrow 2^{63} - 1$

* float : 数字后加 f

* code line.: 1. " { " 同行尾风格

2. 不超过2000行. 每行80字符.

3. 缩进: 4个空格 = 1 Tab .

4. 变量在使用时的最小block里定义.

f.

explizit: Typumwandlung: 圆括号.

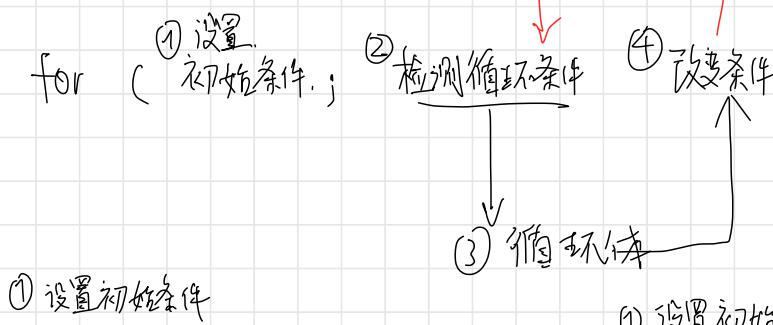
— double explicit int 用去尾法 floor 不用 round.

implizit: 直接转.

Überlant 一溢出.

Konkatenieren (字符串的串联).

char 相加 = 转换为 int 相加. 新一轮.



① 设置初始条件

while (②检测循环条件) ←

③ 循环体

④ 改变条件

① 设置初始条件

do {

 ② 循环体 ←

 ③ 改变条件.

} while (④检测循环条件.)

新一轮循环

十五讲:

类外只允许：“对象.方法”不允许“对象.属性”（通过“getter”来访问，“setter”来修改）
类内才可以“对象.属性”

方法的签名 (Signature)

↳ 包括: 1. 参数 Reihenfolge.

2. 参数 Anzahl

3. 参数 Datentypen

不换都是
直接属性, 不用
this

this 关键字

- 类内部, 方法内部, 属性名与形参名重复了, 用 "this.名字" 表示属性名。
- 类内部, Konstruktor 内部, 调用重名的另一个 Konstruktor, 用 "this.名字 (不同的 signature)" 来指代 another Konstruktor.

"++" 写在变量前到底在哪些情况下有影响?

→ return 前: "return ++": 先 return, 后 ++. > 都会 ++, return 语句就把本行表达式执行完
"return ++": 先 ++, 后 return.

Call by value:

方法的 Übergeberparameter 的数值变量.

传参时只从外界拷贝值, 内部操作不改变外界变量值.

如: 実参为 数组某元素 $a[i]$

- - - 为引用

传参时会拷贝一份引用, 内部操作会改变外界对象.

如: 実参为 数组名 (引用) a .

考试题.

$$x = (5)_{10} \quad y = (9)_{10}$$

$$\hookrightarrow \text{Binar: } x_B = (101)_2$$

$$y_B = (1001)_2$$

十进制 → 二进制

$$\begin{array}{r}
 x_B = 5 : 2 = 2 \ R: 1 \\
 2 : 2 = 1 \ R: 0 \\
 1 : 2 = 0 \ R: 1 \\
 \hline
 (101)_2
 \end{array}$$

↪ bitweise Operation, $x \rightarrow x_B \wedge y_B$

$$a_B = \underline{\text{XOR}}(x_B, y_B) = (1100)_2$$

$$b_B = \text{AND}(x_B, y_B) = (0001)_2$$

$$c_B = \text{OR}(x_B, y_B) = (1101)_2$$

bitwise Operation 过程

$$\begin{array}{r}
 101 \\
 1001 \\
 \hline
 1100
 \end{array}$$

Zahlensystem

移位运算

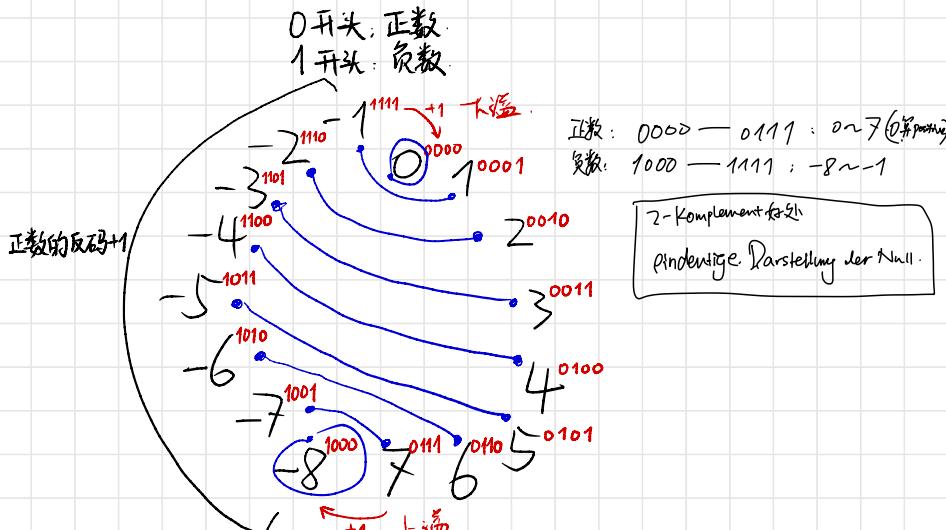
- 右移“>>”n位：除以2的n次方。
- 左移“<<”n位：乘以2的n次方。
- “ $1 \ll n$ ” 记作“1左移n位”

如果是负数，则必须知道数据类型（最高位是符号位1）

再用2-Komplement形式参与移位。

不是移小数点!!!
是反的！

2-Komplement / B-Komplement:



正负数对互为“反码+1”的关系。

0 和最小负数 -8 的 补码是自身。

B七右位移时：正数补0

负数补1 — 反码+1仍闭合

上、下溢出实质都是取 2^n 模运算 放补码与原码等价。

2-Komplement 加减法

- 全是加法，只不过区分正负数。

2-Komplement 加减法

- 转换成 2-Komplement 形式的二进制数。

正数(包括0)直接转。任意进制转二进制的格式?

负数: 其绝对值的 B-1 码
加前导0，取反再加1

e.g. $(-67)_{10} \rightarrow (-67)_B ???$

$$\begin{aligned} 1) & \text{#} (67)_{B-1} \quad (67)_{B-1} = 67 : 2 = 33 \quad R: 1 \\ & \downarrow \\ & 33 : 2 = 16 \quad R: 1 \\ & \downarrow \\ & 16 : 2 = 8 \quad R: 0 \\ & \downarrow \\ & 8 : 2 = 4 \quad R: 0 \\ & \downarrow \\ & 4 : 2 = 2 \quad R: 0 \\ & \downarrow \\ & 2 : 2 = 1 \quad R: 0 \\ & \downarrow \\ & 1 : 2 = \boxed{0} \quad R: 1 \end{aligned}$$

$(1000011)_2$.

2) 补前导0:

$$(67)_{B-1} = 01000011$$

3) 取反+1

$$(-67)_{B-1} = 10111100$$

$$\begin{array}{r} \downarrow +1 \\ (-67)_B = 10111101 \end{array}$$

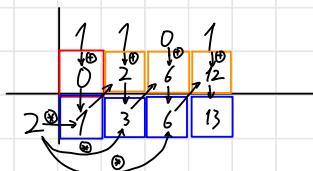
- 两个 2-Komplement 相加,

减转换成加。

就是普通两二进制数相加，各位正常参与相加。

Horners-Schema (任意进制 → 十进制)

e.g. $(1101)_2 \rightarrow ()_{10}$



任何进制 任何数，都写。

上加下，加下来

进制乘口，得到口右上角的口

步骤： 1. 在口处写口

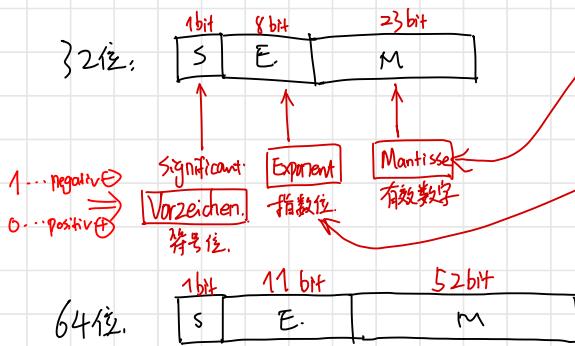
2. 口加上面的数，得到下面第一个口

3. 进制乘口，得到口右上角的口

代替口进行下一轮循环。

Normierte Gleitkommazahlen (IEEE 754)

任意浮点数: $x = S \cdot M \cdot 2^E$



Mantisse 只表示小数点右侧的位数。

但小数点左侧还有一位默默认位是“1”

* 无符号

* Mantisse 最高位对应原数字 (即小数点后第1位)

E 部分是 unsigned (无符号) 的。

但科学计数法指数位有符号

↳ idea: 真实值加上固定的 bias, 再存入 E

	bias
float/8位E	127
double/11位E	1023

如: $1.5 \times 2^{-127} \rightarrow E = -127 + 127 = 0$

$1.5 \times 2^{-10} \rightarrow E = -10 + 127 = 117$

$1.5 \times 2^{10} \rightarrow E = 10 + 127 = 137$

$1.5 \times 2^{128} \rightarrow E = 128 + 127 = 255$

8位E
的表示范围
0~255

$(127)_{10} = (01111111)_2$
Unsigned

Hash

Hash $x \rightarrow y$.

单射: ~~一对一~~ 前域 / 域上多对一
满射: ~~一对一~~ ~~域上多对一~~ ~~一对一~~ ~~域上多对一~~
既不是单射也不是满射: ~~域上多对一~~

nicht-injektiv, 对射(单 + 满).

hashfunktion 是 ~~一对一~~ ~~域上多对一~~ ~~有左~~ ~~是~~ surjektiv (即映射到所有).
有右

\downarrow ~~无~~ Kollision.

解决:

1. offene \rightarrow 2. überlapp / overflow.

(解决办法) $h(x)$ overflow, 避免

$h_1(x), h_2(x), h_3(x) \dots$

$h_m(x)$ 分布冲突

m 个 Behälter.



$h_1(x) = (h(x) + c_1) \bmod m$

c, m 互质.

hash函数 | Laufzeitkomplexität:

m 个 key, m 个 Behälter.

Average: ~~平均数~~, $E[m/n]$ zum.

best:

直接 suchen, Einfügen, Entfernen $\rightarrow O(1)$.

无 Kollision, hashwert \rightarrow 下标, 直接存入 Behälter.

Worst: 全存在 m 个 Behälter, ~~直接存入~~ $O(n)$.

linear Sondierung

$$h_i(x) = (h(x) + c_i) \bmod m$$

Quadrat Sondierung

$$h_i(x) = (h(x) + i^2) \bmod m$$

Bereite Richtung

$$\begin{cases} \text{linear: } -ci + m^2 \\ \text{quadrat: } -i^2 + m^2 \end{cases}$$

Divisionsrest-Methode: $\bmod m$. 若 m 非质数 $\Rightarrow \bmod p \bmod m$.

Mittel-Quadrat-Methode: k^2 数码的中间部分, 23. $k=537$, $k^2 = \underline{\underline{288369}}$

primärkollision.

$h_0(x)$ 矛

sekundärkollision.

$h_k(x)$ 矛

Doppelhashing by sondierung

$$h_{2m} = (h(x) + h'(x) \cdot i^2) \bmod m$$

i)

若且 $D = \frac{1}{m}$ 不对称

~~h(k)~~

OOP

"null": 空 behälter, 空引用.

"undefined": 未定义变量.

Zugriffskontrolle:

public: 公开(本类、子类、包内/包外任意类) 都可以

protected: 本类 子类(无法包内外) 传递给子类的继承的东西.

private: 仅本类. 封装的体现

default: 仅本包(子类非模糊), 包外不可 针对本包访问设计.

Static:

属于类, 不属于 Objekt/Instanz.

e.g. 类名. static 属性/方法.

(!) 静态域不要访问

动态域 ! 除非 static 里创建对象.

(因为 static 方法可以在无实例时调用,
而此时动态域还不存在)

this 关键字:

- 不涉及
继承. {
- 一般方法中(也包括 Konstruktor), this 表示当前对象的引用, e.g. this. attribute, this. method, class anotherObj = this, 避免 Signature 与类 Attribute 重名 但 static 域中不能有 this.
 - 仅在 Konstruktor 里, this(xx), 表示同名构造体,
 - 涉及继承时, 父类中的 this: this(xx) 父类构造体.
this. attribute 父类 attribute.
this. method(e) { 若子类 override, 则子类 method
若无 override, 则父类 method.

父子类同名变量
不涉及
override, 不涉及多态.

Super 关键字 (必涉及继承)

1. 子类构造方法中必须有(隐式/显式) **父类 Constructor**
 1) super():
 • 必须放第一行.
 • 父类没有任何构造器 —— 子类构造器隐式调用super().
 父类只声明default构造器 —— 显式调用.

* Java 规定:
父类全名必须在子类构造器中显式调用!!!



private 又怎么访问?

implement If.

从上到下 / super 重写用

但如何的添加哈?

多态性 Polymorphic.

以下完全是 Vorlesung 和 Tafelübung 的要求!

Tafelübung: 没有跑代码出结果 几乎什么都没说.

Vorlesung: 讲了 Typkonvertierung.

讲了 dynamische Bindung. (但没有讲调用顺序), 局限在父引用 = 子实例,

"父引用, override 方法" 则调用子类方法.

- 子类在类内部调父类成员、方法: 不重名就不加super 直接调.

子类实例在测试类调父类成员、方法: 直接调 (但父类成员方法不能为 private.)

Poly morphic 时, 父类引用调父类同名成员、方法. → 通常调出子类的.

继承

3类 Konstruktor.

UML

- + public
- private
- # protected

attribute : type. = InitialWert. 可无.

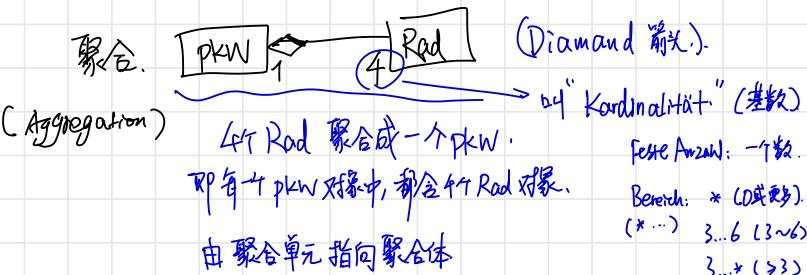
method (Parameter : Typ) : returnTyp.

<<constructor>> method (Parameter : Typ) 无 returnTyp, 一般进构造方法.
 ↓ 加这个更明确

Static 的成员、方法都是下划线如: +get (id: int) : Employee

抽象方法写成斜体.

子类继承, 从子类框用 ↑ (三角箭头) 指到父类框.



组合. 实现 Diamond.



Aufzählung: 0, 1 (或 1)
 (1) 3, 6, 9 (或 6 中的 1)

Exception 处理

注意：程序在碰到
“throw”时会中止，之后的
代码不会执行。try 块中
出现异常的行之后一直到 try
块结束的代码也不会执行。
此时程序会转向 catch 块。

• try {

// try 块放可能引发异常的代码。

// 如果执行 try 不发生异常，则去执行 finally 及 finally 之后代码。

// 如果执行 try 发生异常，则匹配到 catch，匹配到了 / 未匹配到都会执行 finally。

} catch (Exception 类 e) {

// 如果匹配此 Exception 类成功，则执行块内代码，可以输出错误信息、返回方法等。

} finally {

//无论异常是否发生，都会执行 finally 块。finally 块用来作清理工作。

// finally 块可选，finally 和 catch 只存其一。

}

• 何时用 try...catch...finally ?

— 当调用一个方法，不知该方法会不会抛出异常使程序中止时，就把该方法放在 try 块中，并用 catch 块准备捕获该方法抛出的异常。

前提：该方法头部需要声明 (ankündigen) 可能抛出的异常，方法体中出现异常处也要新建 Exception 对象并抛出。

格式如下：

..... + 方法名(参数列表) throws Exception 类 1, Exception 类 2, ... {

 |

 |

 |

 | --- if (...) {

 |

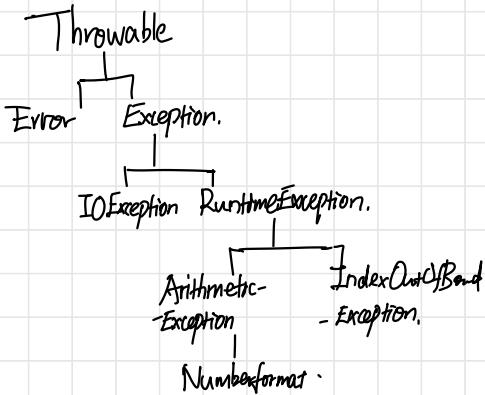
 |

 | throw new Exception 类 1(); // 参数为打印的字符串，后像在

 | // catch System.out.println(e);

如何自定义一个Exception类

Java 异常类的继承关系：



```
public class OwnException extends RuntimeException {  
    private String toPrintInfo;  
    // default构造方法。  
    public OwnException() {}  
    // 带参数构造方法。  
    public OwnException(String toPrintInfo) {  
        this.toPrintInfo = toPrintInfo;  
    }  
    // getter.  
    public String getMessage() {  
        return toPrintInfo;  
    }  
}
```

语法：1. 在某个方法中定义，出现何种情况时抛出。

→ OwnException 实例。 (注意 throw、throws 区别)

方法头要写 throws + 异常类。

throws 在最内层(解释
同时触发异常时)写。

```
public void method1() throws OwnException {  
    // 1. 任何情况  
    // 2. throw new OwnException("提示语");  
    // 3. "throw 之后的语句(直到方法结束)不会执行."
```

2. 在另一个方法中调用 method1. 因为不知道在调用 method1 时.

会不会触发异常.

Final 用 try --- catch 语句, 该方法头不用写 "throws"

public void method2() { }

{ try {

 } } → method2 有时是 main 方法

{ | ... // 出现异常行之后(到 try 块结束) 的语句不会执行.

{ } . Catch < OwnException e > {

{ | ... :

{ }

{ finally {

{ | ... ,

{ }

}

常用 API.

String 类. • A. compareTo(B) : 返回值 0 --- 相等
(<0) ... A < B
(>0) ... A > B.

• A. charAt(int length() - 1)

• A. indexOf(int ch) // ch 第一次出现的位置.
unicode ~~100~~ 99: --

• A. length

• A. substring(int b , int e) // b <= e - 1.

• toLowerCase() . 每个字符都转.

• toUpperCase()

• replace(char oldChar , char newChar) 把所有 oldChar 替换为 newChar

• trim() 去掉前后空格.

Math 类. cos() sin() tan().

log() exp() // e 为底.

random() PI, E.

重写· überschreiben/override

仅与继承相关.

"外貌不变, 核心重写"

方法名, 实现.

返回值(可以是父类返回值的派生类)

final, static, private, Konstruktor 都不能重写.

重写(覆盖) überschreiben (override). 反~父子类 时. 方法相同 signature 无要求

重载(Overload). 本类中. 方法名相同 signature 必须不同

Object 类和 toString()

toString(): 把对象值作为字符串返回

调用顺序:

1. this.show()
2. Super.show()
3. this.show(Super())
4. Super.show(Super())

class A {

```
public String show (D obj){  
    return ("A and D");  
}
```

```
public String show (A obj){  
    return ("A and A");  
}
```

```
}
```

class B {

```
public String show (B obj){  
    return ("B and B");  
}
```

```
public String show (A obj){  
    return ("B and A");  
}
```

```
}
```

class C extends B {}

class D extends B {}

"B and A"

System.out.println (obj2.show (C)); ?

↓ abstract 类: B

Pferd.
- Farbe. String.
- name. String.
anzahl. int
+ wiehern()

>> interface <<
Fabelwesen
+ kannFliegen. boolean()
+ zaubern()

↓ 抽象类

Einhorn

Übersetzung:

public abstract class Pferd {

 private String farbe,
 - - - name;

 protected static int anzahl;

 public abstract void wiehern();
}

 public interface Fabelwesen {

 public static boolean kannFliegen;

 public void zaubern();
 }.

抽象类和接口都没有 constructor.

public class Einhorn extends Pferd implements Farbewesen {

 public Einhorn (String name) { ... },

 @override // 实现 abstract 和接口的方法时也@Override.

 public static wiehern () { ... }

 @Override,

 public void zaubern () { ... }

}.

O-Kalkül.

Wie schnell wächst die Laufzeit mit der Größe der Eingabe

Komplexität (半定量)

语句块的输入量

elementare operation (定量)

赋值, 比较, 传递运算, a[i]

best case, worst case, average case

$\mathcal{O}(n^3) \supset \mathcal{O}(n!) \supset \mathcal{O}(3^n) \supset \mathcal{O}(2^n) \supset \mathcal{O}(n^2) \supset \mathcal{O}(n \lg n) \supset \mathcal{O}(n) \supset \mathcal{O}(\lg n) \supset \mathcal{O}(1)$

Rechenregeln.

常数 1. $\mathcal{O}(f(n)) \pm c = \mathcal{O}(f(n) \pm c) = \mathcal{O}(f(n))$

系数 2. $c \cdot \mathcal{O}(f(n)) = \mathcal{O}(c \cdot f(n)) = \mathcal{O}(f(n))$

$\mathcal{O}(\log_c n) = \mathcal{O}(\log n)$

加法 3. $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

$\mathcal{O}(f(n)) + \mathcal{O}(f(n)) = \mathcal{O}(f(n))$

乘法 4. $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$

嵌套 5. $\mathcal{O}(\mathcal{O}(f(n))) = \mathcal{O}(f(n))$

问法: "Einfachsten Vertreter" 最简形式.

"die niedrigste obere Schrank" 最紧邻高阶.

"einfachsten Verfahren" — 什么 O()

但常常要退化成大或小的，有时还会进错领

$$\ln(3n) = \ln 3 + \ln n. \quad / \text{logan 常常分解开来.}$$

n — 循环度量边界.
i — 循环变量.
m — 循环次数

for 的 Komplexität 看 i 怎么涨

$$i = 2^m < n.$$

嵌套：乘

函数：看 übergebene Parameter (例如: f 里 m 是个
循环体, n 是循环度量界).

$$n = 2^{-m} \geq 0$$

$$m: \log_2 \square$$

O-Notation 是个很粗的东西、不能精确量化.

用途：“niedrigste obere Schranke”
是最高阶.

$$\left\lceil \sum_{i=1}^n \right\rceil n \log(n^4)$$

不严谨，就是“≈”的意思.

$$f: \frac{n}{2} \cdot \log_2 n. \quad \boxed{\quad} \quad g: \boxed{\quad}$$

$O(n)$ --- 假设是，不可能没有一个基本操作。
不可能的。

Recursion

规模小一点的问题。

假设子问题已解决，则用子问题可以很简单地表示当前问题。

iteration, 循环 \rightarrow 终止条件,

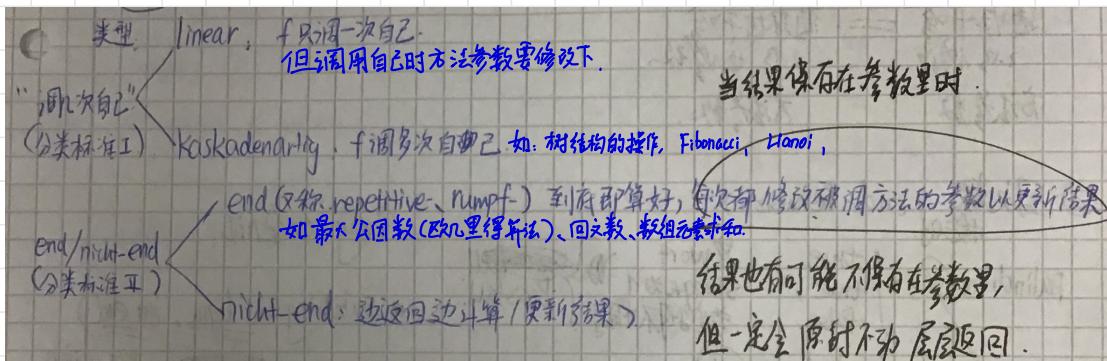
rekursiv, 递归 \rightarrow base case. } 尾递归 \leftrightarrow 循环互相转换的方法。

不同解决大问题，只需直接 $\left\{ \begin{array}{l} 1. \text{如何分解成小问题.} \\ 2. \text{Base case 如何解决.} \end{array} \right.$

需不需要利用Base case 传上来的信息？

怎么利用？

分类



若有： verschachtelte (嵌套), 参数部分也出现递归调用。

verschränkte 互相调用(f 和 g 的)

```

int product = 1;
while (a > 1) {
    product *= a;
    a /= 2;
}
return product * 2;

```

Complexität: $S(n)$?

矩阵乘法:

矩阵

↳ array[]: 1. Element 在数组中 beliebig fix.

2. 每个 Element 有 Zeiger \rightarrow Nachfolger.

3. 可以修改.

4. 可以 beliebig wachsen.

5. 在任何 beliebig Stelle $\exists \cdot / \forall \cdot$.

6. 无 Indizierungsoperator. (下标)

自定义一个 ListItem 类，内含相关操作，或 Java-Collection - List (10. Tafelübung)

结构

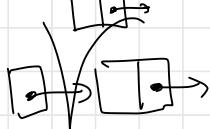
head	fall	- Element
单独一个指针。 (不是 ListItem 实例) $\text{ListItem head} = \text{xxx};$ ↗: head 不是 index=0 的 头部。	ListItem 实例 2个成员, value = xxx; next = null	ListItem 实例 2个成员, value = xxx; next = xxx;

基本操作

* for (ListItem i = head; i != null; i = i.next) {} .

- Einfügen / add.

Fall 1. 在头部插.



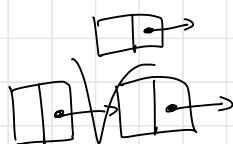
`ListItem newItem = new ListItem();`

初始化值
`newItem.value = xxx;`
`newItem.next = head;` // head 为 null, Element0

`head = newItem`

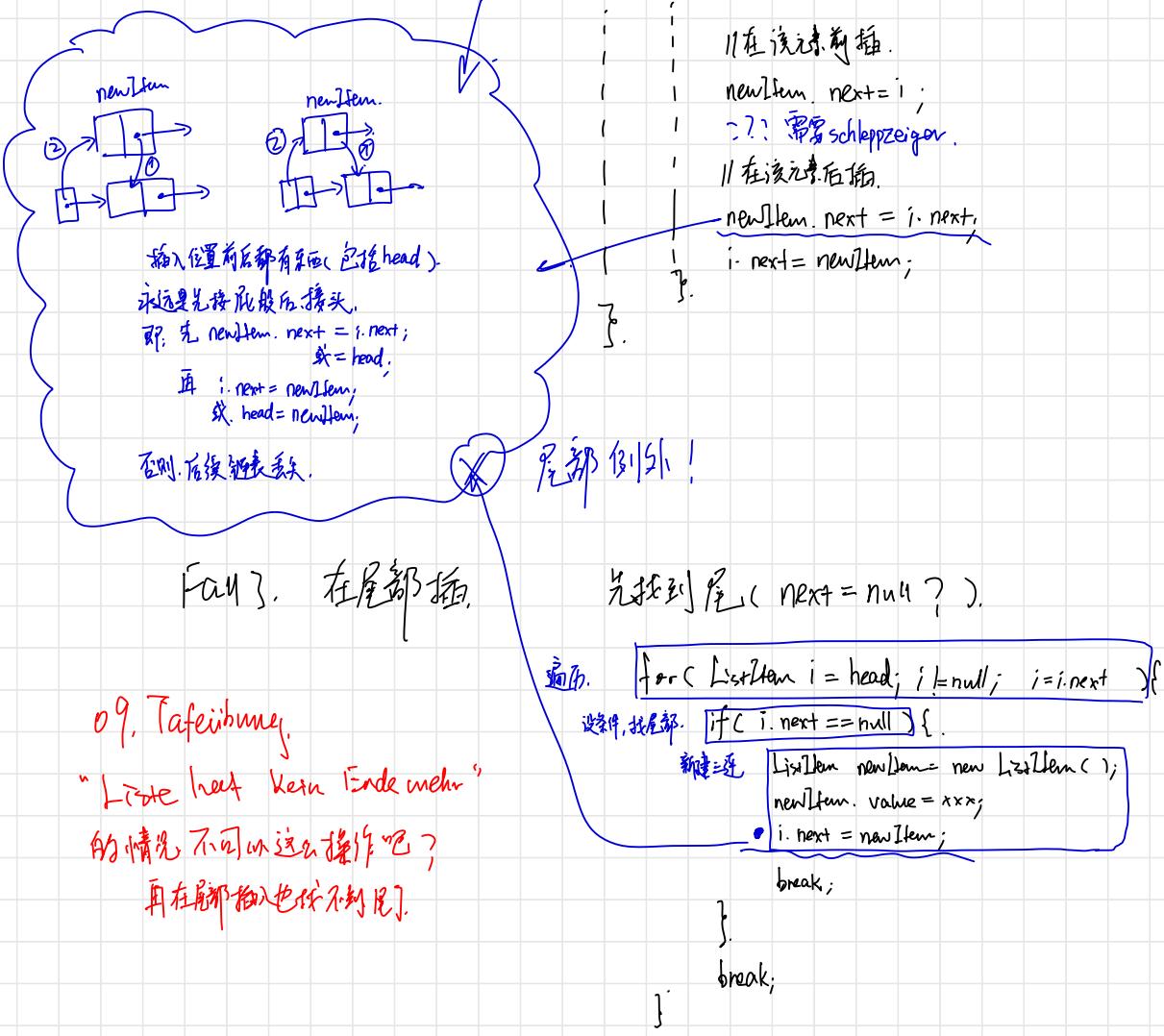
不可颠倒！

Fall 2. 在中间插.



若找到待插入元素, (value == xxx?)
`for(ListItem i = head; i != null; i = i.next) {`
`if (i.value == xxx) { // 替换元素 value? }`
`ListItem newItem = new ListItem();`
`newItem.value = xxx;`

若要在第 n 个处插
`for(int i = 0; i < length; i++)`



自己新建, Listitem 自己再新建 tail-Zeiger? 还是别人已给出?

Java-Collection 的 List 会给出 tail-Zeiger 吗?

Sonderfälle:

1. Head 空.

先写基本情况

Einfügen = head.建立引用

再写 Sonderfall.

Sonderfall 头空.

'if (head == null) { }.

head = newItem;

}.

Länge einer Liste.

rekursiv 一直到 Länge.

- 检验是否存在元素.

循环

```
private boolean exists (int value) {  
    for (ListItem i = head; i != null; i = i.next) {  
        if (i.value == value) {  
            return true;  
        }  
    }  
    return false;  
}.
```

必须写在里边.

里面没法削减规模

递归

```
private boolean exists (int value, ListItem i) {  
    if (i == null) {  
        return false;  
    }  
    if (i.value == value) {  
        return true;  
    }  
    return exists (value, i.next);  
}.
```

Entfernen / Löschen, \Leftarrow 你知道前一个元素. { 1. Schleppzeiger. 2. Doppeltverkettete Liste.

Schleppzeiger.

Entfernen / Löschen 就是 Schleppzeiger.

ListItem prev = null;

Schleppzeiger, 声明在for外.

(是一个实例 / ListItem 方案 2 继承是一个指针.)

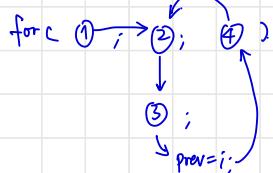
for (ListItem i = head; i != null; i = i.next) {

循环体

}

[prev = i;] schleppzeiger 跟进 / 更新.

每一次循环体执行完后, prev 会更新.
执行下次循环体前 i 会更新.



删除操作
(删掉 i)

prev.next = i.next;

Sonderfälle: Leer, 空指针, 无待删元素

Anfang. 沿着, 第0个没有prev. 且. 不需遍历

$head = (head, head), next?$

Mitte. 元素都是. 1. for 外层 x. schleppzeiger. 2. for 循环. 3. 循环体.
(先if) (先Löschen) 4. prev 变.

Ende. 与步骤同上, 不同点在于循环体内. ①. if ($i.next == null$) .

② 删除操作: $prev.next = null;$

Doppelte verketzte Liste.

含两个指针， prev、next.

- Sortiert Einfügen.

排序插入。

应该前面那部分有什么不同？

必须有schleppzeiger！

```
for (Listitem i = head; i != null; i = i.next) {
```

在 prev 和 i 之间插。

```
| if (!prev < nextItem(i)) {
```

|

```
| newitem.next = i;
```

```
| prev.next = newItem;
```

```
| }.
```

```
| prev = i;
```

```
}
```

Sonderfälle ; Leer. 直接插到 head 后。

Anfangs

没有 prev; if 中比较式不同

和之前插入头部一样。与第 10 页此段后。

newItem.next = head;

head = newItem;

Write. 见示例。

Ende!

Schleppziger: "prev"
插入的时候需要

Generische Klasse

Attribute 可以根据数据类型 \Rightarrow 处理不同 datatype 的 list
可以处理多种数据的.

链表类.

申明 class ListItem<E> {

private E value; // Attribute 类型

public void setValue (E) toSet); // setter 参数类型 \rightarrow 全是链表类: E

public (E) getValue (); // getter 返回值类型

类拥有泛型 T 意味着:

该类储存 T 类的数据 (形成成员变量)

或该类处理 T 类的数据 (用 setter, getter 处理, 或在诸多方法中比大小, 求最大值)

使用

ListItem<String> item = new ListItem<String>();

构造方法后 + <>
(但可以省略)

泛型接口的实现 (继承).

写法 1. interface Info<T> {}.

原来什么字母不重要, 只要是类两个泛型一样.

\Rightarrow 类还是泛型

class Son<T> implements Info<T> {}.

新建对象时待指定.

写法 2. class Son implements Info<String> {}.

子类是 String 型.

新建对象时直接给 String 值.

没必要的写一遍 boundary

type?

如果 interface 的泛型有 bounding type.

子类继承时这么写: class Son<T extends Comparable<T>> implements Info<T> {}

/ 意味 T 类型不明, 但只能是 Comparable 的子类 (那些会比较操作的类)

但是 Comparable<T> 只是个接口, 但我们可以实现它

如: class StringCompare implements Comparable<String>
(...), class DoubleCompare implements Comparable<Double>
(...), class IntCompare implements Comparable<Integer>
....

想要用 int: ArrayList<IntCompare> element1 = new ArrayList<IntCompare>(xxx);
 单 int 类

ArrayList<IntCompare> element2 = new ArrayList<IntCompare>(xxx);

element1.compareTo(element2);

仅有 一个 成员方法

iterator().

// 返回一个 Iterator

迭代器对象.

Iterator 怎么用

Iterable<T>

继承 Iterable<T> 的类
可以用 for each.

for-each 怎么用

首先要继承 接口. Iterator <T>.

尤其适用于 不知元素个数的情况.

for (String s: referenceName) {

 |

 ↓

 元素类型

 |

 ↓
 数值、List、
 LinkedList、ArrayList...
 的名称。
(引用类型变量)

成员方法: T next(). // get 当前 node 的 value.

并聚集下一个.

boolean hasNext(). // 如果下一个非空/还有下一个元素.

List<String> list = new LinkedList<String>();

Iterator<String> it = list.iterator();

while (it.hasNext()) {

 f(it.next()); // 处理 value.

}

Java Collections.

`List<T>` --- 接口. 两种实现类. `LinkedList<T>`, `ArrayList<T>`.
(非抽象)

① `LinkedList<T>`

双向链表.

— 常用方法.

`add` 类: `add(T value)` // 添加到链尾.

`add(int index, T value)` // 添加到 index 处 ? index 前添后添

`addFirst(T value)` // 添加到链头.

`addLast(T value)` // 添加到链尾.

`remove` 类: `remove()` // 删除第一个.

`remove(int index)` // 删除第 index 个.

`remove(T value)` // 删除某个 node.

`clear()` // 清空所有.

`removeFirst()`

`removeLast()`

`get` 类: `get(int index)` // get 指定位置 0 ~ size - 1.

`getFirst()` // get 第一个, 进入表结构

`getLast()`

`indexOf(Object o)` // get 某 node 的位置.

6

— 遍历. (九种著名方式)

① 一般 for 循环.

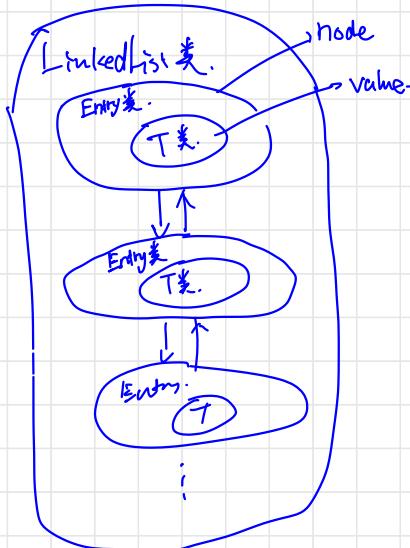
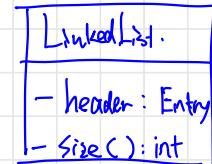
`for (int index = 0; index < list.size(); index++)`

② for-each 循环.

`for (T value : list)` // 容器忽略 Entry 层.

③ 迭代器 Iterator

`LinkedList` 表类的 `size()` 方法.
返回 list 长度.



如果只有 Entry 遍历 list
就很方便.

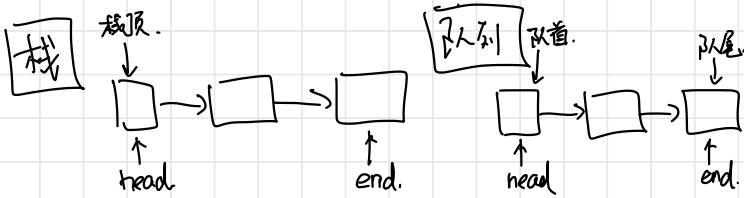
同时 `get(index) + for (index < size)`

④ while (list.removeFirst() != null)

Stack, Queue, Priority Queue 等。

Ringpuffer (环形队列) 是 Vorlesung & Altklausur.

栈表，与 Stack, Queue 关系。



- push = addFirst (value)
- pop = removeFirst()
- enqueue = addLast (value)
- dequeue = removeFirst()

Ausdrucksbaum

post-fix 适用于 stack 的工作原理，先进操作数，再进操作符，一个操作符提出俩操作数，算完结果再入栈，

注意减号. 除号：先出栈的操作数是减数 / 除数。

代码

Vollständig: 圆满, $1+2+4+8+\dots$

Voll: Child数要偶数且为2, 没有单 Nachfolger.

树 → 三大操作查找、添加、删除 + 遍历 + f size .

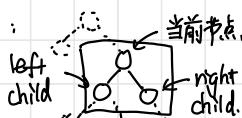
所有数据结构都差不多.

Add.

- Basisfall: 空树(可插)?

Sonderfall: Value 已经存在?

- 核心:



聚焦一个单元. 如果待插 value

比当前节点 value 大, 则往右, 对比 value
与 right child 的 value 的大小; 否则往左.

在与 node 对比完之后, 知道了下一次对
比该往左/右, 但此方向节点为 null, 那么,
value 的插入位置就在这里.

- add 没必要用.

pre./In/Post 遍历每个系统.

只遍历局部即可.

- Add 不会打断既有的树结构

只会作为 Blatt 插在已有树上

```
public void add(TreeNode node, String value){
```

```
    // Sonderfall: 向下寻找(递归)过程中, 发现 value 已经存在  
    if(value.equals(node.value)){  
        throw new ElementExistsException();  
    }
```

// 左? 右.

```
    TreeNode child = (value.compareTo(node.value) > 0) ? node.right :  
                    node.left;
```

```
    // Basisfall: 当前节点为空, 在这里新建节点, 完成最终的“插入”  
    if(child == null){  
        • 当 child != null 时, 可以把 child  
            作为 add() 的参数进行递归,
```

```
        if(< value.compareTo(node.value) > 0){  
            node.right = new TreeNode(value);  
            return;  
        }  
    }
```

因为 child 指向树结构中某一个节点,
但是当 child = null 时,

```
else{  
    node.left = new TreeNode(value);  
    return;  
}
```

```
else{  
    add(child, value);  
    return;  
}
```

child 就不再指向树结构中的某个节点,
必须重新通过本层递归方法 add
的参数 node 重新进入树结构.

简而言之: add 就是深入树的某一支, 一直插到底 (null) 然后新建节点.

find/exist.

- Sondernfall: 空树?

- 核心(递归模块)

Basisfall: value 已经存在 [找到])

找: 往左往右? + 递归.

树的操作里, public

拿一个 value 向往左往右的

都对应同一句话:

往左往右就是为了找到

递归对象 node.left ? node.right

- find 与 add 共性: 都是根据二叉树特点(左<根<右)拿着 value 与节点比较大小然后逐层向下深入.

- find 与 add. Basisfall 的触发条件不一样. find: child.value 等于 value.
add: child 等于 null (到没到底).

```
public boolean find(TreeNode node, int value)  
// 空树?  
if (node == null).  
    return false; // 找不到;  
// Basisfall  
if (node.value == value)  
    return true; // 找到了;
```

// 往左往右?

TreeNode child = (value > node.value)? node.right:
node.left;

return find(child, value);



对 find() 来说, 一杆捅到底 捅出 null 就是不存在该 value. 不需要遍历树的结构
所以可以一直用 child 递归.

remove:

Sondorfell: 空树.

核心算法. (switch).

拿着待删 node 的引用(方法参数)去 check.

• 0个 child, 直接置 null

• 1个 child, 与 child 值交换, 删 child.

• 2个 child, 与 左子树中最大 child 值交换, 遍归删 child.

④ while 在左子树中一直往右找到底 (node == null).

特点: • 没有返回值,

• 最后用 case 0. 1 孩童.

(Borsdell)

• 需要 2 个 Hilfsmethode. numChild 和 replaceInParent.

• remove 前提是已获得待删节点的引用.

private void removec(TreeNode node) throws NoSuchElementException {

// 空树.

if (node == null) {

throw new NoSuchElementException();

return;

switch (numChildc(node)) {

case 0: // 直接置 0

replaceInParent (node, null);

break;

case 1: // 用孩子来代替.

// 左孩子、右孩子?

TreeNode child = (node.left != null) ? node.left : node.right;

replaceInParent (node, child);

break;

case 2: // 用左子树中最大(最右)来代替.

TreeNode maxInLeft = node.left;

// 当前或一直往右找.

while (maxInLeft.right != null) {

maxInLeft = maxInLeft.right;

// "代替" = value copy + 由 maxInLeft (递归)

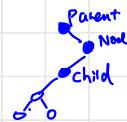
node.value = maxInLeft.value;

remove (maxInLeft);

break;

default:

这是一个仅为 remove() 设计的 Hilfsmethode. 其作用仅在于:



当 node 只有一个 child 时.
将 child 的引用交给 parent.
把 node 抛掉.

当 node 有两个 children, remove 的含义就不同了.
此时应该做的: 在 node 的左子树中找最大的
(即最右的) node, 将其值与待删 node 交换, 再对
该最大节点调用 remove(). (此时往往直接变成 case 0 或
case 1)

或在右子树中找最大的(最左的) 同上操作

前中后序遍历

Pre Order.

子含子含.

PreOrder(node.left);
PreOrder(node.right);

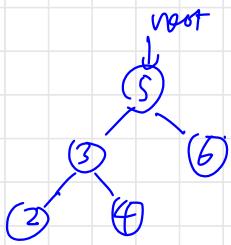
InOrder.

InOrder(node.left);
子含子含.
InOrder(node.right)

PostOrder.

子含子含.

postOrder(node.left);
postOrder(node.right);



```

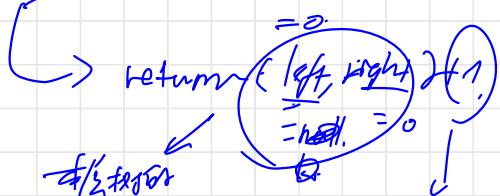
add( node ) {
    // Sonderfall: 空指针?
    if ( node == null )
        return xxx;
    // Sonderfall: 节点?
    if ( -- == - - ? )
        // ...
    ...
}
  
```

return Math.max(getheight (root.left) , getheight (root.right)) + 1;



(3).

return Math.max($\frac{\text{left}}{=1}$, right) + 1



左子节点高度为0
右子节点高度为0
本层最高节点高度为1

① 求树高

Sonderfall: 空树?

核心: 递归得到左子树高.

递归得到右子树高.

返回 $\max(\text{左高}, \text{右高}) + 1$

Public int getHeight(TreeNode node) {

if (node == null)

return 0;

return Math.max(getHeight(node.left), getHeight(node.right)) + 1;

② 求节点数 (算法同求树高)

Sonderfall: 空树.

核心: 递归得左子树节点数.

递归得右子树节点数.

返回. 左节点数 + 右节点数 + 1 (当前).

Public int count(TreeNode node) {

if (node == null)

return 0;

return count(node.left) + count(node.right) + 1;

f.size

却因 cascading

而功

AVL 树旋转

- 总是转三个节点.
- 起点: 新添节点 (单节点) → 转着. 从上往下数 39.
终点: 第一个非空的节点
- \leftarrow grand parent
 \rightarrow parent.
 \Rightarrow child.

$\frac{1}{2} \rightarrow$ 逆时针.



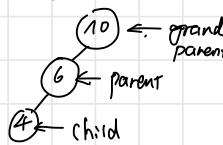
何时?

— “双左”: left child, left subtree.

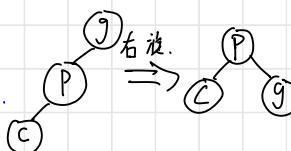
谁旋? 绕谁旋?

— g 旋, 绕 P 旋

$\frac{1}{2} \rightarrow$ 顺时.



可能看它的具体数值大小?
P 的右侧树放哪里?



何时右旋? (找出 g, p, c 三个节点后)

— left child & left subtree (“双左”)

(P 是 g 的 left child) (C 是 P 的 left subtree)

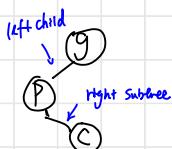
谁旋? 绕谁旋?

— g 旋, 绕 P 旋.

同时双旋?

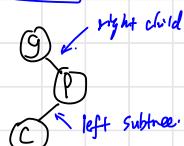
child 和 subtree 不一致 = “Knick”

[LR]

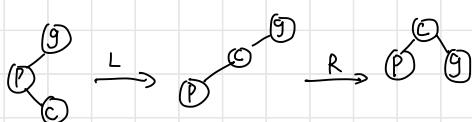


left child, right subtree \rightarrow left-right Rotation

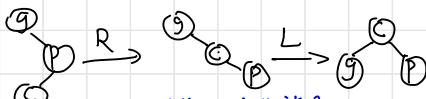
[RL]



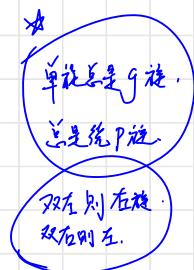
right child, left subtree \rightarrow right-left Rotation.



谁旋? 绕谁旋? — 先 C 左旋, 再 g 右旋, 总是“绕 P 旋转的 Knick”旋



谁旋? 绕谁旋?
— 先 C 右旋, 再 g 左旋, 总是“绕 P ...”



Wrapper Klassen

包装类， primitive 类对应的对像类（内含求整等方法）

Integer 类， Double 类， 属于 referenzdatatyp - 无法用运算符比大小运算 或赋类型转换
但可以用相应的方法来

ADT

Abstract Datentypen. 主要用来描述一种数据结构

通过 Sorten、 Operationen、 Axiome.

都是类名。

大多数数据结构。

“公理”

相当于 operation
(操作/运算) 的
操作数 Operand.

都具备的一些操作。
如: add, remove,
IsEmpty, create.

特定操作对应的结果。
(都是不言自明的魔法，
但可以表达这种数据
结构的性质)

这里补充一点..

比如： 对于 List (链表) 这种数据结构来说，

+ 在这里是一个
自定义的、 用于作链表
节点的类

Sorten: List, T, int, boolean,

Operationen:

create : \rightarrow List.

// create 操作会返回一个 List.

add : List \times T \rightarrow List

// add 对 List 和 T 操作， 返回一个 List.

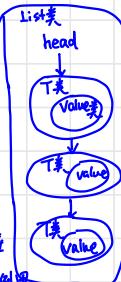
:

Axiome: isEmpty (create) = true.

// create 返回一个空链表， 该链表为空吗？ true.

contains (create, x) = false.

// 空链表中是否存在元素 x? false.

- value 类是我们存储的数据的类型， 如： int, double, String ...
 - Value 被封装在一个节点类 T 的实例里， T 中包含 next, prev 指针。 所有 T 类的实例彼此连接形成链表。
 - 上述类的成员变量 head 是一个 T 类型变量， 它指向链表头。
 - List 类还包含 add, remove, find 等方法。
 - 这一系列的封装、 链接保证了一个 List 类实例 list 内包含了一个完整的链表， 程序可以通过 list.head 进入链表结构， 通过 (list.head).value 来访问链表中保存的数据。
- 

Suche.

Vorlesung 独有内容.

- linear Suche. $O(n)$.

- binär Suche. $O(\log n)$ 但 Folge 必须已排好序.

(二分法)

高中就学过: 在一串排好序的数据中查某元素 c , 先选中间位置 p , 若 $p < s$, 则往右找.
 (aufsteigend) 若 $p > s$, 则往左找.

Sortier 算法.

贪心算法. $\left\{ \begin{array}{l} \text{selection sort. } O(n^2) \\ \text{insertion sort.} \\ \text{bubble sort.} \\ \text{heap sort.} \end{array} \right.$ 分治 $\left\{ \begin{array}{l} \text{Quicksort.} \\ \text{Merge sort.} \end{array} \right.$

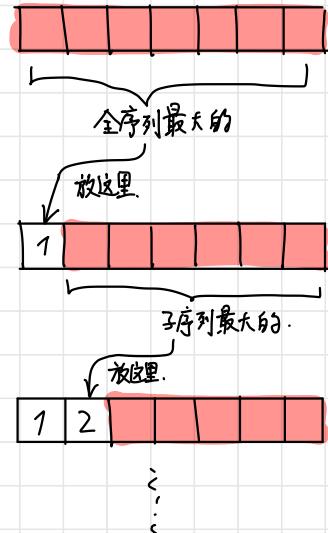
bucket Sort. (桶排序).

radix Sort.

Comparable
 排序算法中 Comparable 可能会很常见.
 Comparable 是一个接口, 含一些比较对大小的方法, 一个封装数据类 Entry 类/功能上等同于 ListItem 类或 Comparable
 若继承了该接口且实现了该接口的方法, 则该 Entry 类实例 比较大小可以通过这些方法.

Selection Sort: 从不断缩小的子序列中选出最大/小的, 然后在 sorted 序列后面.

```
selectionSort( Comparable[] element ) {
    // 不断减小子序列规模.
    for( int i = 0; i < element.length; i++ ){
        T max = element[i];
        // 遍历子序列找 max.
        for( int j = i + 1; j < element.length; j++ ){
            if( element[j].compareTo(max) > 0 ){
                T temp = max;
                max = element[j];
                element[j] = temp;
            }
        }
    }
}
```



insertionsort: 从不断缩小的子序列中选一个出来，插入到 sorted 序列中间。

aufsteigend: 找左小右大插进去。

absteigend: 找左大右小插进去。

这里我的代码模仿 mergesort“插入”部分的写法。

insertionsort (Comparable[] element) {

```
| Comparable[] newElement = new T<E>[element.length]; // 建立新数组来容纳排列好的  
| for ( int i=0; i<element.length; i++ ) { // 第层 for: 不断从左  
|   newElement[i] = element[i]; // 子序列中选元素出来  
|   for ( int j=0; j<i; j++ ) { // 第二层 for:  
|     if ( newElement[i] < newElement[j] ) {  
|       T temp = element[j];  
|       System.arraycopy ( newElement, j, newElement, j+1, (i-1)-j+1 );  
|       newElement[j] = temp;  
|       break;  
|     } else continue;  
|   }  
| }  
| }
```

Mergesort: 先两两归并成 einzelne. 再原路排回去。以下代码是 Vorlesung 上
抄下来的，只是数组实现。

mergeSort (Comparable[] element, int low, int high) {

if (low >= high) // divide 的尽头，子数组头=尾，这时不做

但一定要理解算法，确保用 List 也能写出来。

return; // 任何操作而是返回上层开始二路归并。

int middle = (low+high)/2;

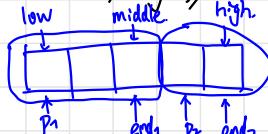
mergeSort (element, low, middle);

mergeSort (element, middle+1, high);

int p1 = low;

int p2 = middle+1;

int end1 = middle;



把传参进来的数组分成两个部分。

divide 到底，认为数组已经被
分为多个已经排序的序列。此时
开始“二路归并”（把两个序
合并成一个有序的操作，降低同理）

“divide”
越深的递归，
通过方法传入的
参数能访问的
数组长度就越短。

第一次执行二路归并操作正好与底层 mergeSort，此时传参进来的数组长度仅为2，排序后返回上层再对长度为4的子数组进行排序。

↖

```

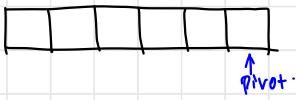
int end2 = high;
while ( (p1 <= end1) && ( p2 <= end2 ) ) {
    if ( element[p1].compareTo( element[p2] ) <= 0 ) {
        p1++;
    } else {
        Comparable temp = element[p2];
        System.arraycopy( element, p1, element, p1+1, end1-p1+1 );
        element[p1] = temp;
        end1++;
        p1++;
        p2++;
    }
}

```

△ 并没有所谓的“合并”操作。
因为排序都是在原数组结构上进行的。
排好本层再排上层就能保证序列最终被排好。

3.

Quick Sort: 规则有点复杂，但理解了规则代码很好写。



1. 指定末元素作为 pivot.
2. 指针从头元素开始向尾移动，指到第一个比 pivot 大的元素时停下来。
3. j 由 i 的位置出发，指到第一个比 pivot 小的元素时停下来，i, j 交换。
4. i 往前走 1 步，（每次交换后皆如此，为的是让 i 指向第一个比 pivot 大的元素，且左侧全是比 pivot 小的元素）。
5. j 继续走，指到比 pivot 小的元素就再换，重复步骤 3 直到 i 指到 pivot。
6. 此时再交换 i, j。此时原序列被分成两个子序列。

Teil 1: 0 ~ i - 1, Teil 2: i ~ j.

quickSort (Comparable[] element, int low, int high) {

```

int pivot = high;
int i = low;
while ( element[i].compareTo( element[pivot] ) < 0 ) {
    i++;
}

int j = i + 1;
while ( j < pivot ) {
    if ( element[j].compareTo( element[pivot] ) < 0 ) {
        // 交换.
    }
}

```

```

    i--;
}
else j++;
}

// 截至这里, j = pivot, 再与 i 交换;
quicksort(element, low, i-1);
quicksort(element, i+1, high);
}

```

Radix Sort.

非比较型. ohne Schlüsselvergleiche.

$O(n \cdot \lg n)$

Schlüssel = Schlüsselwert = Value.

被排序的值!

- 使用队列作为桶: $B_0 \sim B_9$
- 按位拆分整数, 从右向左, 先看个位.
- 从 $0 \sim 9$, 从队伍一队屋, 将元素返回原数组
- 再将个位重新放入一队

代码来源于考试题 (应该是17年).

radixSort (List< Integer > numbers) {

List< Integer > buckets = new ArrayList< > (10); // 根据要求建立十个桶.

for (int i = 0 ; i < 10 ; i++) {

| buckets.add(new ArrayList< Integer > ()); // 每一个桶存放一个队列, 用ArrayList实现.

}

// 8 次循环, 对应 MatrixIndex 的 8 个位 (digit).

for (int digit = 0 ; digit < 8 ; digit++) {

| // 从原 List (numbers) 中取数放入桶中.

| for (Integer m : numbers) {

| | // 根据 m 的第 digit 位, 将 m 放进对应 bucket. o ArrayList.add(m)
| | buckets.get(getDecimalDigit(m, digit)).add(m); 将 m 添加在 ArrayList.

| // 先删掉原 List 内容.

| numbers.clear();

| // 再遍历 bucket, 将元素重新 copy 回 bucket.

| for (int b = 0 ; b < 9 ; b++) {

| | for (Integer m : bucket.get(b))

| | | numbers.add(m);

| }

int.

o buckets.get(\downarrow)
get buckets 第 int 位元素.

o ArrayList.add(m)
将 m 添加在 ArrayList.

bucketSort: (桶排序)

Vorlesung 前只涉及 Integer.

最快 (比Quicksort快). 但最耗空间.
(非比较型) $O(n)$

- 乱序数组 n个数. 均有范围, [0, MAX_VALUE]
- 建立 bucket 数组长度为 MAX_VALUE. (容量够用).
 - $\text{bucket}[value] = i$ (有序, 看代码)
- 若 duplicate, 用 List 实现 bucket. 然后 offenes Hashing?

heapSort

public void heapSort (Comparable[] element)

int length = element.length();

乱序数组

该方法 heapSort.
会先将乱序数组
排列为堆化数组,
再排列为升序/降序
数组

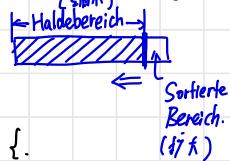
// first : heapify ab i bis Ende (应该是 greedy 体现)
for(int i=n/2; i>0; i--) {
 heapify(element, i, n-1); // i ... 堆化起点
}

往下这支通通“堆化”
n-1 ... 数组尾. 读的底

升序/降序提
最终目标, 堆化
只是中间步骤.

// 把 max 放到数组尾, 作为已经排好序的array. 端 index 不应该超过 n-1
for(int i=n-1; i>0; i--) {
 vertausche(element, 0, i);
 heapify(element, 0, i-1); // 剩余
}

结构重新堆化! "Wiederherstellung"
(缩写)
↓
Haldebereich



Hilfsmethode:

heapify (Comparable[] element, int start, int end) {

// 当前 start 元素的左右 child 的 index: (前提: 数组 index 从 0 开始)

int left = 2 * start + 1;

int right = 2 * start + 2;

↳ heap 特性: links-vollständig.

```

// 如果当前元素只有 left child, 则说明本次堆化已经到底, 交换完后方法可以返回。
if (left < end) && (right > end) {
    // 左右大小, 判断是否交换
    if (element[start].compareTo(element[left]) < 0) {
        vertausche(element, start, left);
    }
}

// 左右 child 都有, 说明未到底, 换完后还需递归。
else if (right < end) {
    int tausche = element[left].compareTo(element[right]) > 0 ? left : right;
    if (element[start].compareTo(element[tausche]) < 0) {
        vertausche(element, start, tausche);
        heapify(element, tausche, end);
    }
}

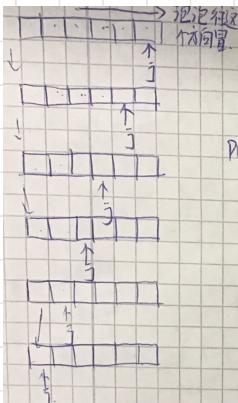
```

```

vertausche( Comparable[] element, int i, int j) {
    Comparable temp = element[i];
    element[i] = element[j];
    element[j] = temp;
}

```

Bubble Sort. 备忘。



如果有一个足够大的元素在之前, 将它与数组中两个比较并不通过交换一定能看到最大值。

* 只要元素变大, 就交换, 往右冒。

```

public static void bubbleSort( Comparable[] element ) {
    boolean vertauscht = false;
    for( int j = element.length - 1; j > 0; j-- ) {
        for( int i = 0; i < j; i++ ) {
            if( element[i].compareTo(element[i+1]) > 0 ) {
                Comparable temp = element[i];
                element[i] = element[i+1];
                element[i+1] = element[i];
                vertauscht = true;
            }
        }
        if( !vertauscht ) // 最后一次循环未执行, 没做交换。
            break;          // 打印 !vertauscht = true
    }
}

```

Graph.

几种数据结构: adjazenzmatrix (邻接矩阵)

adjazenzliste. Kantenliste.

einbettung in ein Feld.

↓
代表含义
element
Array index

A, B, C, D会被转换成 0, 1, 2, 3 来储存.

(A) (B) (C) (D)

↑
4

0 1 2 3 4 5 6 7 8 9

Graph 中的 A 给点的邻接信息 (A 能抵达的其余结点的序号). 储存在 index 从 4 开始的元素中.

B ...

C ...

D ...

其中, 如果从 B 出发谁也到不了, 则 B, C 的储存位置共同一个 index

找最小生成树.

几种算法: Prim.

Kruskal.

(权重排序法)

把边按权重排序, 从小到大挑选.
如果构成回路就跳过

随便选 start Knoten.

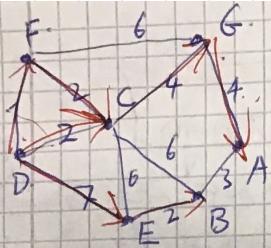
再选与此点相接的最近边

并纳入第二个点, 该两点作为

子 Baum, 在与孩子 Baum 邻接

的所有边中找最小邻接边.

Dijkstra 算法: (找 Kurzest Weg)



Dijkstra 算法:

1. 指定起始点, 第一行为起始点是 D, 其它全是 ∞
2. 看 Startknoten 有无直连各个顶点, 有则更新长度.
选一个最近的作为中间点.
3. 从 Startknoten \rightarrow 中间点到其它各点(不包括 Start 和中间点)
有比上面短的就更新.
- ④ 再选一个最近的作为中间点. . .

A	B	C	D	E
20	∞	∞	0	∞
20	∞	2	2	7
10	∞	2	2	7
10	∞	2	2	7
11.	9	2	0	7

F	G
∞	∞
1	∞

7. 如果新中间点没有更新任何旧路径, 就从旧路径中选一个最长的.
对应的点就是新中间点.
△ 注意看这个上边前连的是谁.
- ↓
- 8 再用这一条新路径去更新旧路径.

以下是 Graph 部分容易忽略的基础知识, 及 13-19 卷子整理出的部分基础知识.

图 - 基本概念

- 有向 (D) 的 Graph 叫 Schicht. 又叫单向图 Graph.
- 节点有直接 Vor-/Nach 从 方向. Vor-/Nach-gänger.

Ungerichtet. 图. $V_i, V_j \in V = (V_i, V_j) \in E$ (边 $(V_j, V_i) \in E$) 对称的. Symmetrisch.

相邻又叫 adjazent, 邻接. Kante $\overline{V_i V_j}$

- zusammenhängend. (两个点之间都有路径 (不是 Kante))
(连通的)
- stark zusammenhängend. (任意两点间任意
(强连通的))

• schwach zusammenhängend. (没有带权的 Zusammenhangswert).

• Spannbaum. & Teilgraph von G. (zusammenhängend,zyklenfrei, baile Knoten).

两种表示形式. Adjazenzmatrix

~~矩阵表示法~~

Adjazenzliste.

Kanteliste

Einbettung in ein Feld.

时间复杂度

Adjazenzmatrix. ① 找 Kante: $O(1)$... 抽象化

② 找 V_i 的 Nachbarn. $O(n)$... \rightarrow V_i 有多少个 ~~孩子~~ 孩子.

③ 找 V_i 的 Vorgänger. $O(n)$

④ 找 Nachgänger. $O(n)$.

Späher.

$n \rightarrow$ Knoten. $m \dashv \dashv$ Kanten.

$(n^2 \text{ Platz}),$ ungerichtet die $\frac{n \cdot (n+1)}{2}$

Adjazenzliste.

抽样.

Späher. $n+m$ - gerichtet

$n + \underline{2m} \dashv \dashv$ ungerichtet
 $\rightarrow n \rightarrow m$ nach
nach $\rightarrow n$

直接 Field 有 indices.

BRUNNEN

Primärkollision,

$b_1(x)$ 接触

drückkollision.

接触

= 288369

~~288369~~

~~bck~~

2. BL. 算法

= ~~288369~~

right

height > size

n Knoten

1

2

3

4

5

6

7

W

S

Z



Enter

Shift

PgDn



Kantentypen:

Speicher: $\exists \text{ mit platz}$: von? nach? gewicht?

Einfüllung in Array: $\forall n \in \mathbb{N}$, index.

Speicher:

从后到前写入。

Hilfslate \rightarrow Zeiger.

Sortierung:

Greedy:

{ Selection $O(n^2)$ insertion. $O(n^2)$ bubble. $O(n^2)$ heap. $O(n \log n)$

divide & conquer

{ merge $O(n \log n)$ { quick $O(n \log n)$ 与算法 $O(n)$:Vergleichsbasiert: (平均上) $\Omega(n \log n)$. $O(n)$ \rightarrow best-case.

Selection.

应该都是根据 max 做的比较, $i+1 + i+2 + \dots + (n-1)$

Insertion.

(平均) $O(n^2)$.

Bubble

(平均) $O(n^2)$.

Heap

HeapSort: $\forall n \in \mathbb{N}$ Einfügen. $O(n \log n)$ $\forall n \in \mathbb{N}$ Entnahmen $O(n \log n)$ (平均) $O(n^2) = O(n \log n)$ - R. Einfügen / Entnahme: $O(\log n)$

Divide & Conquer:

1. "divide": Teile das Problem.

2. "conquer": Teilproblem lösen.

3. "join": Teillösung kombinieren.

merge: easy split, hard join.

quicksort: hard split, easy join.

万年

涉及基础数据问题。

19年. 7月.

Aufgabe 7. 1. nutzt AVL. Höhendifferenz von Knoten $72 - 67 = 2 - 5 = -3$

neue Wahlen, für jeden Knoten, ist value ~~entweder~~ ≤ -2 .
nicht maximale oder nicht minimale ~~Wert~~ Verglichen mit Pfakstelbaum & Rechtsfeuerum.

2.

Häufchen fügt: links-Vollständig

eine Ebene bis auf letzte \rightarrow voll besetzt.

Blätter links wie möglich.

Aufgabe 8.

九种图遍历法 $O(n)$ $E \log V$

Kruskal: $O(\frac{E}{\log V})$ ~~时间~~: $O(n \log n)$, ~~空间~~ $O(n)$ - 最差。

Prim: $O(\frac{E}{\log V})$ ~~时间~~ ~~空间~~

Dijkstra: $O(V^2 + E)$. ~~时间~~ ~~空间~~ ~~最短路径~~, ~~起始顶点~~.

Floyd $O(n^3)$ $i,j: n(n-1) \times$ n^2 Knoten.

18'

Aufgabe 1. vergleich zw. 3 vergleich. Sortierung $\mathcal{O}(n)$? Langzeitkomplexität basiert auf paarweise Vergleich mit Elementen.

Vergleichbasert: $\mathcal{O}(n \log n)$, Langzeit ist hauptsächlich abhängig von wie viel mal mehr Element, desto höher Aufwand (Linear Austrag)

法2.

选择. 搜索算法. $\mathcal{O}(n)$.

Linear. unsortierte Liste = Linear Suche: $\mathcal{O}(n)$

BRUNNEN

Sortierte Array. Binär Suche: $\mathcal{O}(\log n)$.

Linear. $\mathcal{O}(n)$

Binär Suche. $\mathcal{O}(\log n)$

Schnellsorten.
选择排序 $\Theta(n^2)$.

linear-unsortierte Liste. MergeSort: $\Theta(n \log n)$.

Unsortiert Array. (noch bekannt)

$\Theta(n \log n)$

Halde. $\Theta(n \log n)$.

Aufgabe 8.

- Typumwandlung:

implizit: syntatisch,

nicht sichtbar
automatisches Compiler
ausgeführt.

semantisch.

in Java: ~~cast Operator existiert~~
keine Operator existiert
nur Nieder \rightarrow Hohe.

explizit: sichtbar
von Programmierer gemacht.

cast Operator existiert:
in Java "Typecast"
nur Höhenrangfolge \rightarrow Niederwertige.

Wrapper Klasse z.B.: packt eine primitive Datentypen in Objekt ein.

Auswirkung: bei generische Klasse nur Klasse (keine primitive Datentypen).

als Parameter übergeben werden. Wenn man primitive Datentypen speichern möchte,
 $\text{für } <T>$. muss man Wrapper Klassen benutzen.

2018. 2. 19.

Aufgabe 1. Sortieren: Quicksort: für gleich großer Teilfelder.

↓
→ günstig Aufteilung.

Radix-Sort: Schalter für List .

Binärbaum: Pre-Order. Max. $\frac{n(n+1)}{2}$

(zulässig).

图示 Tiefensuche 索不序 Hilfsdatenstruktur ①

图示 kurz weg 图示 Dijkstra. \rightarrow effektiv 但 inefficient.

Spannbaum Tiefendurchgang. (Gewicht ~~无向图~~ 无边的无向图)

kurzweg.
gerichtet
ungerichtet

仅针对有向图

但复杂度高 1. !

O-Kalkül: niedrigste obere Schranke - (\geq 的). 且没有二进制的。
niedrigste untere Schranke .
如果它是大于的，则是 \leq 最近的

RadixSort: wiederholtes BucketSort.

zuerst ~~aus~~, gemäß LSD die Elemente ins verschiedenen Bucket aufteilen.
dann aufnehmen. Dann aufsteigend nach höherwertigen Digit

Wofür Stabilität重要?

- stabil für: die erste-mal Ergebnis kann an nächster-Mal zur Verfügung stellen.
jedes Mal basiert auf das Ergebnis von letztem Mal abhängig.

2016. 10.

jedes Mal basiert auf das Ergebnis von letztem Mal abhängig.

?) einfacher Graph zu Adjazenzlisten korrekt?

又即 Schlüssel, 每一个 Knoten hat Liste mit自己出现的结点.

举例, 每一个 Knoten hat Liste mit自己.

Kanten dr.

2016. 4.

Aufgabe 2. Hash.

offen.

geschlossen.

1. für jede Behälter,
gibt's eine expandierbare Liste

1. jede Behälter
hat festgelegte Plätze. keine neue Plätze belegt.
üblich implementiert als Feld.

2. bei Kollision kein Problem.

2. bei Kollision muss
Sondieren

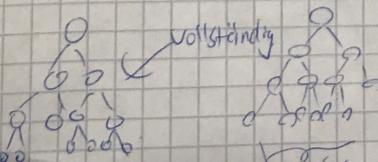
3. List einfach, effizient löschen.

4. dynamische Anwendung.

gesamtzahl nicht begrenzt.

Aufgabe 1. Voll-Baum: innere Knoten \rightarrow 2^{Teilbaum} 个.

底层 Ebene 不是 2^n .



Voll \Rightarrow . Nachfolger
0, 1, 00, 01, 000, 001, 010, 011, 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111.

Vollständig \Rightarrow Voll d Blatt-Baum.

6个, 也是 voll. (且 vollständig).

