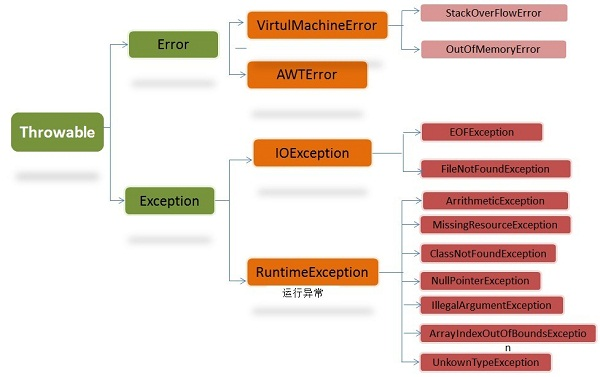


# Java

异常



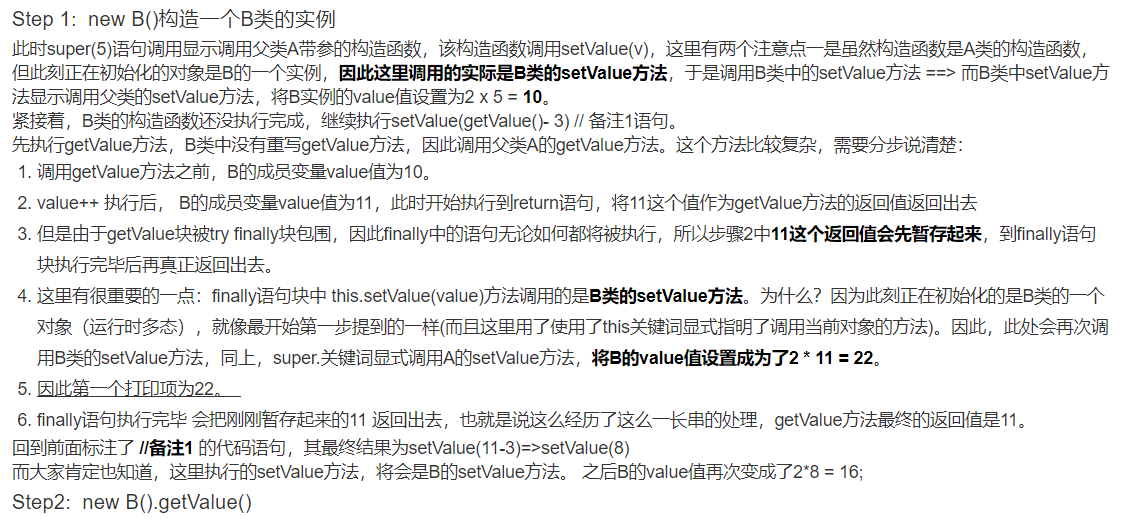
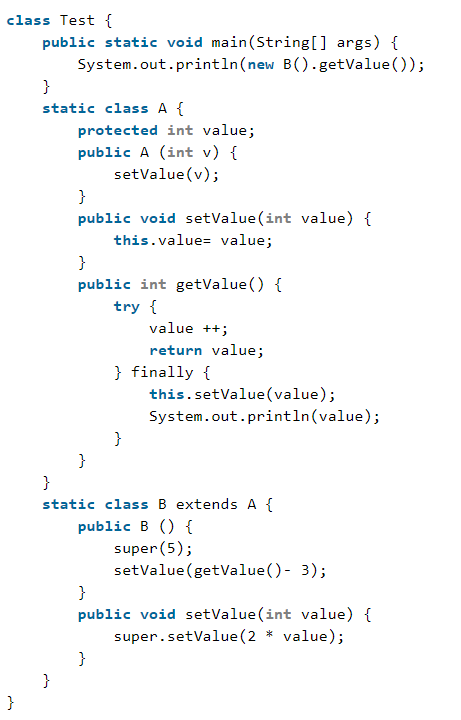
垃圾回收机制

JVM考点

内存泄漏

反射

内存：堆、栈、方法区





构造函数，析构函数

# 数据结构

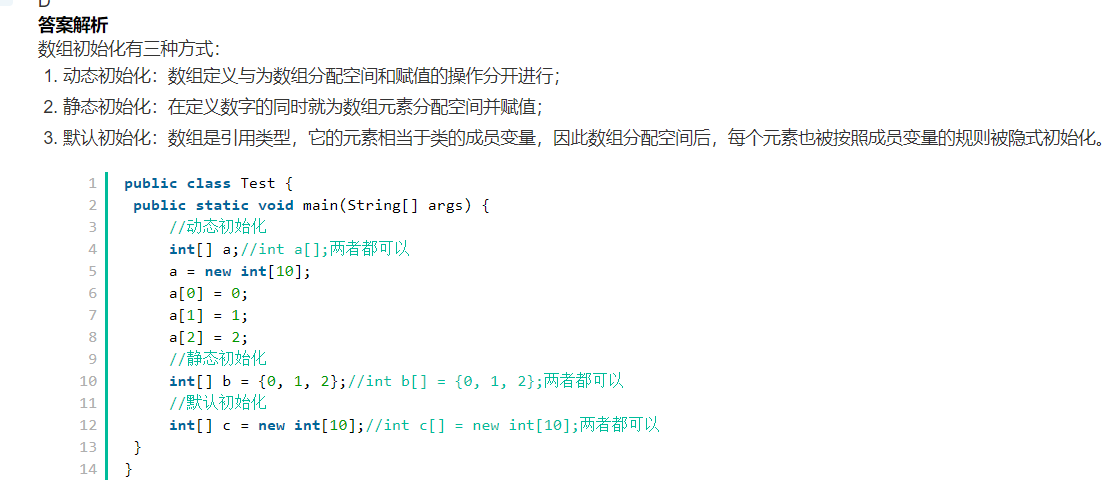
数据结构包括：物理结构（存储方式）和逻辑结构

物理结构包括：顺序（线性）、链式、索引、散列表

逻辑结构包括：集合、线性、树状、图状

## 数组

* 数组属于线性结构，可以使用顺序存储或链式存储
* 三元组进行转置时不仅要行列互换，而且原本按行排列要变成按列排列
* 数组只能一个个元素单个赋值
* 数组初始化



注意：int[] a等价于int a[]

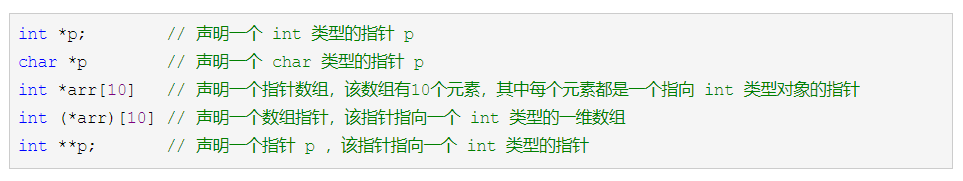
* 指针和数组（[参考](https://blog.csdn.net/cherrydreamsover/article/details/81741459)）

指针大小只和系统有关，数组大小和数据类型有关（[使用sizeof比较](https://blog.csdn.net/cherrydreamsover/article/details/81589838)）

指针数组&数组指针：

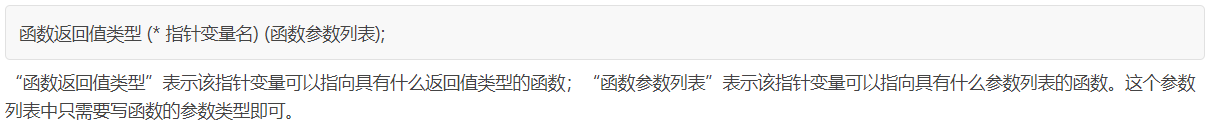
[]的优先级比\*高🡪int \*arr[8]是数组，int (\*arr)[8]是指针

指针：



int \*p=\*a[0];p++; //那么最终\*p=a[1]

函数指针：



# include <stdio.h>

int Max(int, int); //函数声明

int main(void)

{

int(\*p)(int, int); //定义一个函数指针

int a, b, c;

p = Max; //把函数Max赋给指针变量p, 使p指向Max函数

printf("please enter a and b:");

scanf("%d%d", &a, &b);

c = (\*p)(a, b); //通过函数指针调用Max函数

printf("a = %d\nb = %d\nmax = %d\n", a, b, c);

return 0;

}

int Max(int x, int y) //定义Max函数

{

int z;

if (x > y)

{

z = x;

}

else

{

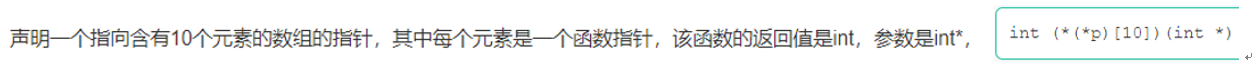
z = y;

}

return z;

}

例题：



## 字符串

sizeof表示字符串时包括结束符’\0’,使用strlen时不包括

字符串的相关方法：

转义字符：\t（不是/t）

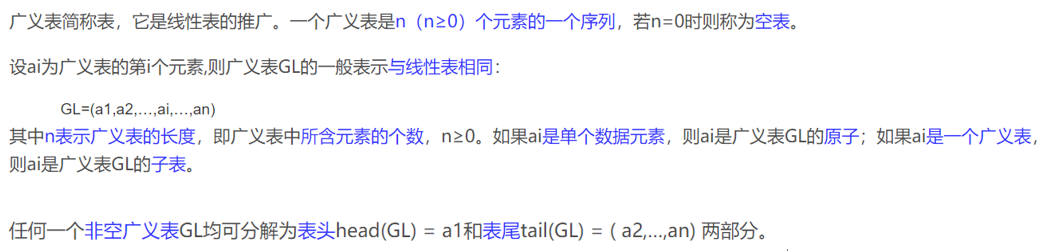
KMP算法

字符串查找

字符串匹配

## 链表

## 广义表



## 栈

## 队列

## 树

### 二叉树

B+树、红黑树、AVL树、二叉搜索树、二叉排序树、平衡树

树的旋转，左旋与右旋

树的高度是指从树的根节点到最低的叶节点所需要的步数，或者更正式的定义：高度是指节点之间的边的最大值。一个高度为 h 的堆有 h+1 层。

哈夫曼树的带权路径的确定

完全二叉树（深度为 k ，有 n 个结点的二叉树当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时，称为完全二叉树。）

满二叉树（堆不保证节点的个数正好能构成满二叉树）

二叉排序树（最小堆只保证父节点比孩子节点小，并不是二叉排序树）

平衡二叉树（二叉平衡树肯定是一颗二叉排序树，堆不是二叉排序树）

深度优先和广度优先

### 非二叉树

## 图

深度优先，广度优先

dijkstra

最短路径

## 哈希

Hash ：散列，通过关于键值(key)的函数，将数据映射到内存存储中一个位置来访问。这个过程叫做Hash，这个映射函数称做散列函数，存放记录的数组称做散列表(Hash Table),又叫哈希表。（[参考](https://blog.csdn.net/yt618121/article/details/81162836)）

**常见的hash函数：**

直接定址法、数字分析法、平方取中法、折叠法、随机数法、除留余数法、旋转法

**解决hash冲突：**

开放地址法（闭hash）、链表法（开hash）、线性探测法、二次探测法、公共溢出区、再散列

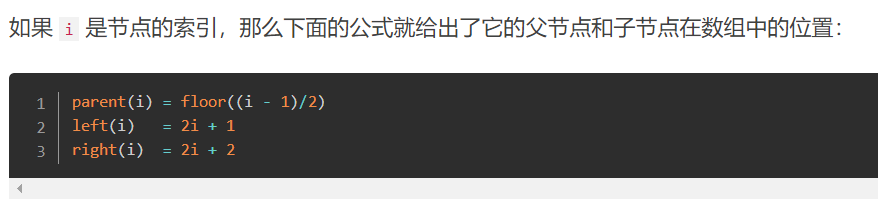
装填因子=元素个数/表长

哈希表平均查找长度与解决冲突的方法和装填因子有关，与表长无关

在内存中查找效率高的是hash，在磁盘中查找效率高的是B+树，内存中也可以使用STL\_Map（红黑树）进行查找，但是效率比前者低

## 堆

堆：数组实现的二叉树，且必须是完全二叉树（[参考](https://www.jianshu.com/p/6b526aa481b1)）



最大（最小）堆也叫大根堆（小根堆）：父节点比子节点大

堆的插入和删除

如何建堆？

# 算法

## 排序

考虑因素：稳定、时间复杂度、空间复杂度

### 插入排序

### 冒泡排序

### 快速排序

### 基数排序

### 堆排序

堆排序时间复杂度O（nlogn），建堆时间复杂度O（n），重建堆时间复杂度O（logn）

最大堆

拓扑排序

## 查找

查找分为静态查找和动态查找，也可以分为有序查找和无序查找

考点：平均查找长度

### 顺序查找（无序查找）

**原理：**将目标值和查询序列的元素挨个依次比较

**平均查找长度：**（n+1）/2

**最大查找长度：**n

**时间复杂度：**O（n）

public static int SequenceSearch(int[] arr,int value) {

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

if(value == arr[i]) {

return i;

}

}

return -1;

}

### 二分查找（有序查找）

**原理：**将有序序列的中间值和查找值进行比较，中间值把序列分为两个序列，然后再将查找值与对应序列的中间值进行比较，如此递归。（限制条件：顺序存储的有序列表）

**平均查找长度：**太复杂，略

**最大查找长度：**log2n

**时间复杂度：**O（logn）

递归方式：

public static int BinarySearch(int[] arr, int low, int high, int value) {

int mid= (low + high) / 2;

if (low <= high) {

if (value > arr[mid])

return BinarySearch(arr, mid + 1, high, value);

if (value < arr[mid])

return BinarySearch(arr, low, mid - 1, value);

if (value == arr[mid])

return mid;

}

return -1;

}

非递归方式：

public static int BinarySearch(int[] arr, int value) {

int low = 0;

int high = arr.length-1;

int mid;

while (low <= high) {

mid = (low + high) / 2;

if (value > arr[mid])

low = mid + 1;

if (value < arr[mid])

high = mid - 1;

if (value == arr[mid])

return mid;

}

return -1;

}

### 插值查找（有序查找）

**原理：**二分查找的优化，二分查找为对半查找，插值查找则是根据值的大小进行调整自适应mid进行查找（适用于关键字分布均匀的情况）



**最大查找长度：**log2n

**时间复杂度：**O（logn）

public static int InsertValueSearch(int[] arr, int left, int right, int findVal) {

//注意：findVal < arr[0] 和 findVal > arr[arr.length - 1] 必须需要

//否则我们得到的 mid 可能越界

if (left > right || findVal < arr[left] || findVal > arr[right]) {

return -1;

}

// 求出mid, 自适应

int mid = left + (right - left) \* (findVal - arr[left]) / (arr[right] - arr[left]);

if (findVal > arr[mid]) { // 说明应该向右边递归

return InsertValueSearch(arr, mid + 1, right, findVal);

} else if (findVal < arr[mid]) { // 说明向左递归查找

return InsertValueSearch(arr, left, mid - 1, findVal);

} else {

return mid;

}

}

### 斐波那契查找（有序查找）

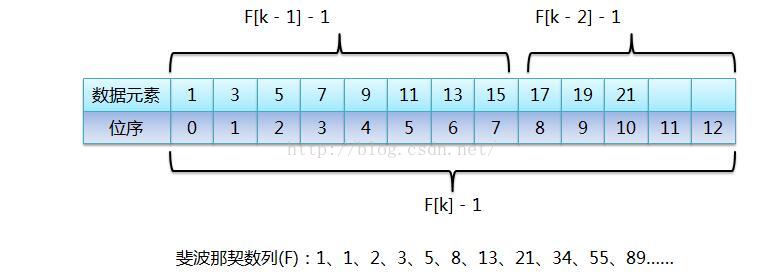
**原理：**二分查找的另一个优化，使用黄金比例的概念进行mid确定，优点是确定mid时不使用乘除法，只使用加减法。

**最大查找长度：**log2n

**时间复杂度：**O（logn）

斐波那契数列：1，1，2，3，5，8，13，21

划分方式：（划分前需要对不够的部分进行补充）



public class Test {

public static int maxSize = 20;

public static void main(String[] args) {

int[] arr = { 1, 8, 10, 89, 100, 123};

System.out.println("index="+fibSearch(arr, 123));

}

// 构建斐波那契数列

// 非递归

public static int[] fib() {

int[] f = new int[maxSize];

f[0] = 1;

f[1] = 1;

for (int i = 2; i < maxSize; i++) {

f[i] = f[i - 1] + f[i - 2];

}

return f;

}

// 斐波那契查找算法（非递归）

/\*\*

\*

\* @param a

\* 数组

\* @param key

\* 要查找的值

\* @return 返回对应下标

\*/

public static int fibSearch(int[] a, int key) {

int low = 0;

int high = a.length - 1;

int k = 0;// 表示斐波那契分割数值的下标

int mid = 0;// 存放mid值

int f[] = fib();// 获取斐波那契数列

// 获取斐波那契分割数值的下标

while (high > f[k] - 1) {

k++;

}

// 因为f[k]值可能大于数组的长度。因此我们需要Arrays类，构造一个新的数组，并指向temp[]

// 不足的部分会使用0填充

int[] temp = Arrays.copyOf(a, f[k]);

// 实际需要使用a数组的最后的数填充temp

for (int i = high + 1; i < temp.length; i++) {

temp[i] = a[high];

}

// 使用while循环来处理，找到key

while (low <= high) {

mid = low + f[k - 1] - 1;

if (key < temp[mid]) {// 向数组的前面查找

high = mid - 1;

// 说明：k--

// 1.全部元素=前面元素+后面元素

// 2.f[k]=f[k-1]+f[k-2]

// 因为前面有f[k-1]个元素可以继续拆分f[k-1]=f[k-2]+f[k-3]

// 即在f[k-1]的前面继续查找k--

// 即下次循环mid=f[k-1-1]-1

k--;

} else if (key > temp[mid]) {// 向数组后面查找

low = mid + 1;

// 说明：k-=2

// 1.全部元素=前面的元素+后边元素

// 2.f[k]=f[k-1]+f[k-2]

// 3.因为后面我们有f[k-2]所以可以继续拆分f[k-1]=f[k-3]+f[k-4]

// 4.即在f[k-2]的前面进行查找k-=2

// 5.即下次循环mid=f[k-1-2]-1

k -= 2;

} else {// 找到

// 需要确定返回的是哪个下标

if (mid < high) {

return mid;

} else {

return a.length-1;

}

}

}

return -1;

}

}

### 树表查找

**二叉查找树：**左子树<根节点<右子树

**最大查找长度：**n

**时间复杂度：**O（logn）

**平衡树：**任意节点的左右子树高度差不超过1

**红黑树：**略

### 分块查找

**原理：**顺序查找的改进算法，先确定记录在哪一块，然后在对应块进行顺序查找

### 哈希查找

**原理：**构建哈希表进行查找

### 动态查找？？？

## 递归

### 汉诺塔问题

## 复杂度



## 图的遍历

析构函数

机器指令和微程序

向量中断

运算顺序：淡云一笔安洛三赋（单目>算数运算符>移位>比较>按位>逻辑>三目>赋值>逗号）

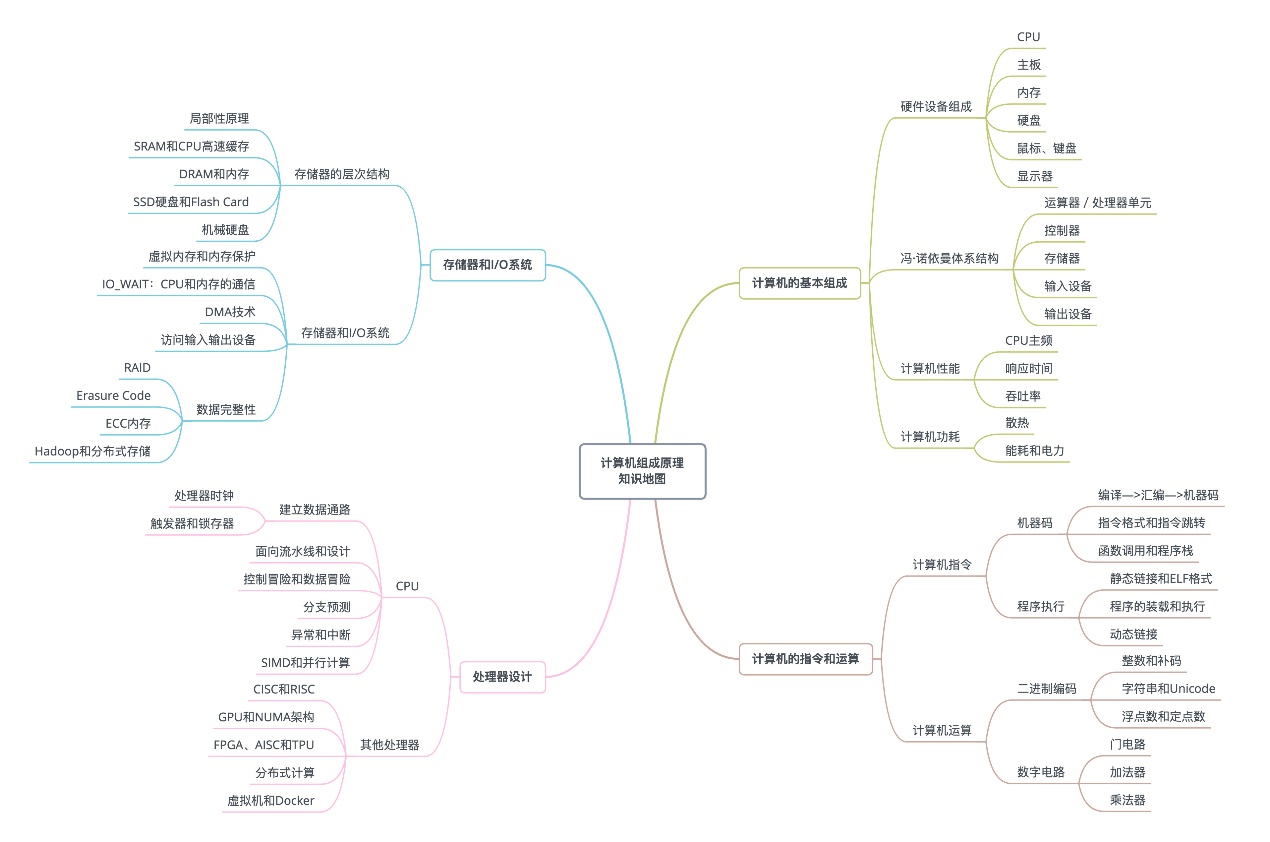


java枚举类型

# 数据库

视图、事件、备份的用法

# 计算机组成原理



# 计算机网络

# 操作系统