一、基本数据结构

1. **数据结构分类：**

* 线性结构：数据元素一一对应
* 存储结构：顺序存储（地址连续）、链式储存（地址分散 ）
* 常见结构：数组、列表、链表、栈
* 非线性结构：
* 二维数组、多维数组、广义表、树结构、图结构

1. **稀疏数组：**

* 概念：第一行记录：原始数组的行、列、非零元素个数；

第二行往下，记录原始数组中非零元素的：行号、列号、值；

* 作用：用于0元素较多的数组，达到缩小数组的目的。
* eg：



1. **队列：**

* **概念：**队列是一个有序列表，先进先出，可以使用数组、链表实现。
* **应用：**银行排队取号系统；
* **环形队列：**

使用取模的方法，将数组改成环形数组——即：达到环形队列效果。

**使用取模法的环形数组判断方法：**

（1）头（head）、尾（tail）初始值 = 0；

（2）判断队列为满：（tail + 1）% MAXSIZE == header；

（3）判断队列为空：tail == head；

（4）实际存储的数据元素，只能为MAXSIZE-1个；

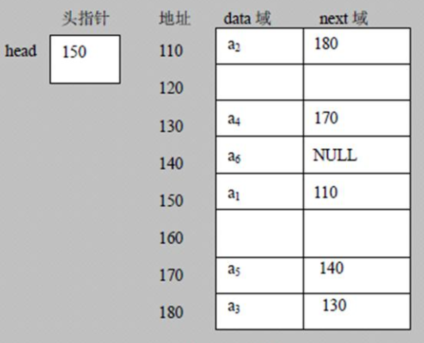
（5）队列中的元素个数：（tail + MAXSIZE - header）% MAXSIZE；



### 链表

1. **单链表：**

* 链表是以节点的方式来存储数据，
* 节点组成：data域 + next域（用于指向下一个节点）；
* 单链表可划分为：带头节点的链表、没有头节点的链表；
* **单链表的结构图：**
* 内存中：



* 逻辑结构：



* **单链表的反向：**

**思路：**将原来的链表最前面的元素，放到反转后链表的最前面，最后将原链表的头部指向反转后的链表。

public void reverse(){  
 ListNode next = null; **//当前节点的下一个节点**  
 ListNode current = header.next; **//原链表的头节点**  
 ListNode reverseHeader = new ListNode(0,"",""); **//反转后链表的头节点**  
 if(current == null){ **//原链表为空** System.*out*.println("LINKED lIST IS NULL !");  
 return;  
 }  
 while (true){   
 if(current == null){  
 break;  
 }  
 next = current.next; **//保存当前原链表的下一个节点** current.next = reverseHeader.next; **//拼接当前节点、反转链表的头部**  
 reverseHeader.next = current;  
 current = next; **//原链表当前节点后移一个位置** }  
 header = reverseHeader; **//原链表头部接到反转后的链表上**  
}

* **单链表的反向输出：**

**思路：**使用栈的性质：先进后出，实现反转打印链表。

public void reversePrint(){  
 ListNode current = header.next;  
 if(current == null){  
 System.*out*.println("LINKED LIST IS NULL !");  
 return;  
 }  
 Stack<ListNode> stack = new Stack<ListNode>();  
 **//放入数据**  
 while (current != null){  
 stack.push(current);  
 current = current.next;  
 }  
  **//取出数据**  
 while (stack.size() > 0){  
 System.*out*.println(stack.pop());  
 }  
}

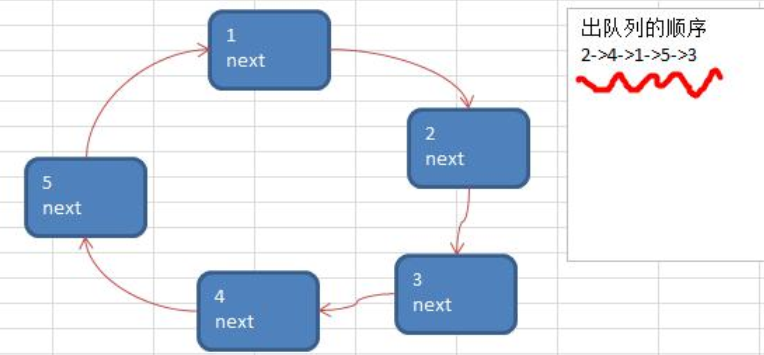
1. **单向循环链表：（约瑟夫环Josephu）**

* **概念：**

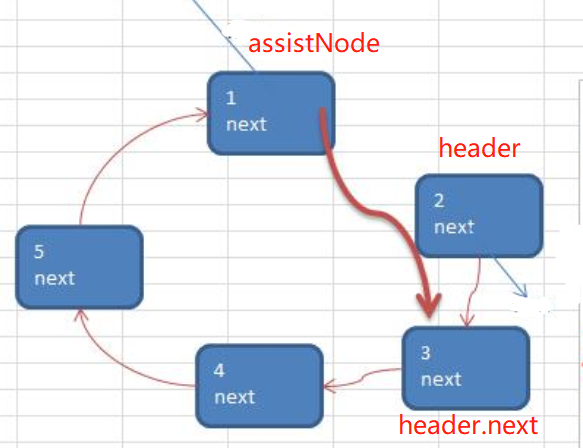
单项链表，首尾节点相接，形成环状链表。

* **约瑟夫问题：**

有1，2，3…n个人围成一圈，从第一个人开始报数，报到m的人出列，下一个人又从1开始报数，以此类推，知道所有人都出列，会产生一个出列的编号。示意图如下：



* **出列的思路：**



* **实现：**

public class Josephu {  
 private SingleListNode **header** = new SingleListNode(0);  
 private int size;  
 **/\*\*  
 \* 添加节点：一次性添加多少个节点  
 \* @param num  
 \*/**  
 public void put(int num){  
 if(num < 1){  
 System.*out*.println("ERROR：The number of Node shuould't < 1");  
 return;  
 }  
 SingleListNode currentNode = null; **// currentNode表示链表的最后一个节点**  
 for(int i = 1; i <= num; i++){  
 SingleListNode newNode = new SingleListNode(i);  
 if(i == 1){  
 header = newNode;  **//加入第一个节点时,构成环状**  
 header.setNext(header);  
 currentNode = header; **//本质上，currentNode就是最后一个节点**  
 }  
 else {  
 currentNode.setNext(newNode); **//后面添加的节点，自动与头节点构成环状** newNode.setNext(header);  
 currentNode = newNode;  
 }  
 size++;  
 }  
 }

**/\*\*  
 \* 节点出圈，获得出列的顺序  
 \* 思路：1、使用辅助节点assistNode：位于first后一个的节点，用于拼接出列后的链表  
 \* 2、每次移动countNum个节点，first所在位置为出列节点  
 \* 3、assistNode、first，同时移动每次移动countNum - 1，因为自身所在位置为1  
 \*/**  
 public void sequence(int startId, int countNum){  
 if(startId > size || startId <= 0 || countNum >= size){  
 System.*out*.println("ERROR; invalid parameters !");  
 return;  
 }  
 SingleListNode **assitNode** = header; **//辅助节点**  
 **//找到辅助节点位置：位于header的后一个节点——链表尾部**  
 while (true){  
 if(assitNode.getNext() == header){  
 break;  
 }  
 assitNode = assitNode.getNext();  
 }  
 **//从第startId节点开始出列：移动assistNode、header节点为位置**  
 while (startId > 1){  
 header = header.getNext();  
 assitNode = assitNode.getNext();  
 startId--;  
 }  
  **//获得出列的顺序**  
 while (true){  
 if(assitNode == header){  
 break;  
 }  
 for(int i = 0; i < countNum - 1; i++){ **//移动countNum-1个位置**  
 header = header.getNext();  
 assitNode = assitNode.getNext();  
 }  
 System.*out*.print(header.getId() + "->");**//出列**  
 header = header.getNext();  
 assitNode.setNext(header);  
 }  
 System.*out*.println("\nTHE LAST NODE = " + header.getId());  
 }

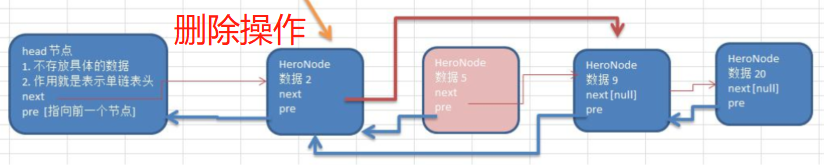
public static void main(String[] args) {  
 Josephu josephu = new Josephu();  
 josephu.put(25);  
 josephu.sequence(2,2);  
 }  
}

1. **双向链表：**

* **特点：**
* 查找方向：可前可后；（单向链表只能往后）
* 可自我删除节点，不需要使用辅助节点；

（单项链表需要前一个节点辅助删除节点）

* **实现：**和单向链表差不多
* **注意：**删除节点时：需要分为链表头、链表尾、其余节点等情况考虑



1. **判定链表是否存在环形**
2. **相交链表的判定**



## 栈：

* **介绍：**

先进后出，限制元素插入、删除只能从一端进行的特殊线性表。

push：入栈； pop：出栈

* **应用：**
* 输入+-\*/的表达式，计算结果
* 处理递归调用：存储下一个指令的地址、参数、变量等
* 表达式的转换：中缀表达式->后缀表达式
* **二叉树遍历：**前序、中序、后序遍历
* 图的深度优先搜索算法
* **实现栈：**
* 使用数组
* 使用单向链表
* **前缀表达式：**波兰表达式
* **概念：**运算符位于操作数之前；
* **eg：**（3+4）\*5-6，**前缀表达式为：**- \* + 3 4 5 6；
* **计算机求解：**从右到左扫描表达式，数字全部入栈。再获取运算符，当遇到一个运算符时，弹出两个数进行计算，再将结果入栈。。。直到扫描完表达式
* **中缀表达式：**
* **概念：**需要判断运算符的优先级，也就是我们常规的计算思路 （一般来说会将中缀表达式转换成后缀表达式）
* **eg：**（3+4）\*5-6，**中缀表达式为：**3 + 4 \* 5 - 6；
* **后缀表达式：**逆波兰表达式——**常用的方法，适合计算机**
* **概念：**与前缀表达式相似，但运算符位于操作数后
* **eg：**（3+4）\*5-6，**后缀表达式为：**3 4 + 5 \* 6 -；

a = 1 + 3，后缀表达式为：a 1 3 + = ；

3\*4+5-4/2+1，后缀表达式为：3 4 \* 5 + 4 2 / - 1 +；

* **计算机求解：**从左到右扫描表达式，遇到数字入栈，遇到运算符，弹出两个数计算，将结果入栈。。。直到表达式扫描完
* **中缀表达式🡪后缀表达式，并计算后缀表达式：**
* **中缀表达式-->列表：**方便后期操作
* **中缀表达式-->后缀表达式：**
* 创建1个栈：存储符号，一个列表：存储数字

**开始遍历表达式：**

* 遇到数字：直接压入数字列表
* 遇到“（”：压入符号栈
* 遇到运算符：和符号栈的栈顶元素比较优先级；
* 当前运算符优先级 <= 栈顶元素：

将栈内元素一直出栈，放到队列（直到，栈顶元素优先级 < 当前运算符，为止）。并且将当前元素入栈；

* 当前运算符优先级 > 栈顶元素：

直接将当前运算符入栈；

* 遇到“）”：将符号栈元素出栈，放到队列，直到遇到“（”为止，并且将该“（”出栈——消除一对括号

**表达式遍历完**

* 将所有符号栈内的元素出栈，放到列表中——得到后缀表达式
* **后缀表达式计算**
* 创建一个栈：存储数字
* 遍历后缀表达式：
* 遇到数字：直接入栈
* 遇到运算符：从栈中弹出两个数字，按照运算符计算，并将结果压入栈中
* 遍历完表达式后，栈中最后一个元素就是计算结果

## 递归：

1. **应用：**

* **迷宫问题：**
* 搜索策略：下右上左、上下右左…
* 不同的搜索策略得到的结果不同
* **八皇后问题：**
* **解决方法：**使用递归（回溯）
* **思路:** 每放一个皇后都需判断与前面皇后的关系：同列、同斜线？
* **解决方式：**使用一个一维数组存储一次解：
* 一维数组的**元素序号**表示：第几个皇后
* 一维数组的**元素值**表示：该皇后放在第几个位置
* **本质：**一位数组可以当成一个平面坐标系使用，可用于判断元素是否在同一直线上，详见：isPass()方法

**元素序号num**：x轴坐标

**元素值val：**y轴坐标

**判断两个皇后是否在同一斜线：**即判断是否在y=x直线上

**判断方法：**|num1 - num2| == |val1 – val2|,

**x轴与y轴的增量是否相等**

* **深度优先搜索：**
* **归并排序：**
* **递归的缺点：**

递归的效率低，占用内存大

## 排序

1. **分类：**

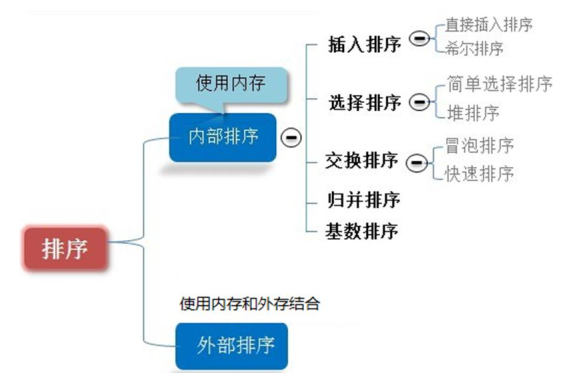
* **内部排序：**

将需要处理的数据全部加载到内存中进行排序；

* **外部排序：**

数据量较大时，无法将数据全部加载到内存中时，需使用外部存储进行排序，例如：使用文件存储数据

* **详细分类：**



1. **时间复杂度：度量程序的运行时间**

* **估算方法：**
* **事后统计：**

需要实际运行程序，运行的计算即需要一样，依赖于硬件，才能得到实际算法的优劣

* **事前估算：**

通过分析算法的时间复杂度得到运行时间，以此判断算法的优劣

* **时间频度：**
* **介绍：**

一个算法中语句执行的次数，称为时间频度（语句频度），记为T（n）

* **应用：**

可以忽略的：常数项、低次项、忽略系数

* **时间复杂度：**
* **T（n）：**算法中的基本操作语句的重复执行次数时问题规模n的某个函数；
* **f（n）：**某个辅助函数，当n趋于无穷时，使得T（n）/f（n）的极限等于不为零的常数；称f（n）是T（n）的**同数量级函数**
* **T（n）=O（f（n））：**O（f（n））为算法的渐进时间复杂度（时间复杂度）
* **常见的时间复杂度：**
* **常数阶O(1):**

eg：int m = 1+2; 或者，

int i = 1;

while(i<**10**){

i=i\*2;}

* **对数阶O(log2n)：**

**eg:** for(int i=1; i<n; i=i\*2){} ，或者

int i = 1;

while(i<n){

i=i\*2;}

**分析：**假设n=1024，则退出while循环需要i>1024;

2^x=n ==> 循环执行次数x=log21024=log2n

* **线性阶O(n)：**

eg：一个for循环

for(int i=0; i<n; i++){}

* **线性对数阶O(nlog2n)：**

eg：for(int i=0; i<n; i++){

j=1;

while(j<n){

j=j\*2;

}

}

* **平方阶O(n^2)：**

eg：两个for循环

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

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

}

}

* **立方阶O(n^3)：**

eg：三个for循环

* **k次方阶O(n^k)：**

eg：k个for循环

* **指数阶O(2^n)：**

eg: for(int i=0; i<2^n; i++){}

* **多项式时间复杂度的关系：**

O(1)<O(logn)<O(n)<O(nlogn)<O(n^2)<O(n^3)<O(2^n)<O(n!)

* **平均时间复杂度：**

所有可能的输入以等概率出现的情况，该算法运行的时间

* **最坏时间复杂度：**

任何输入下，该算法运行时间的上限

* **空间复杂度：**

算法所耗费的空间，快速排序、归并排序、基数排序属于这种情况，一般都是用空间换时间（因此，有redis这种缓存产品）

* **P类问题：**（Polynmial，多项式问题）

O(logn)、O(n)、O(nlogn)、O(n^2)、O(n^3)

* **NP问题：**（Non-Deterministic Polynmial，非确定多项式问题）

O(2^n)、O(n!)

* **时间复杂度计算：（详解）**

1. **计算步骤：**
2. 找出基本语句；
3. 计算基本语句的执行次数的数量级；
4. 用大O记号表示算法的时间性能；

**eg：**

　　for (i=1; i<=n; i++)

　　 x++; ①

　　for (i=1; i<=n; i++)

　 for (j=1; j<=n; j++)

　　 x++; ②

**分析：**

基本语句①：执行n次；

基本语句②：执行n^2次；

整个算法的时间复杂度：O(n + n^2) = O(n^2)

1. **分析法则：**

**算法整体的时间复杂度 = 各个部分时间复杂度之和**

1. **求和法则：（一般用于顺序结构）**

* T1(n)=O(f(n))和 T2(n)=O(g(n)),

则 T1(n)+T2(n) = O(max(f(n), g(n)))

* T1(m)=O(f(m)), T2(n)=O(g(n)),

则 T1(m)+T2(n) = O(f(m) + g(n))

1. **乘法法则：（一般用于循环结构）**

* T1(n)=O(f(n))和 T2(n)=O(g(n)),

则 T1\*T2=O(f(n)\*g(n))

* (所有的常数都记为1)

①：g(n)=O(f(n)),则O(f(n))+ O(g(n))= O(f(n))；

②：O(Cf(n)) = O(f(n)),其中C是一个正常数；

1. **注意：**

循环语句只需考虑循环体中语句的执行次数，忽略该语句中步长加1、终值判别、控制转移等成分，当有若干个循环语句时，算法的时间复杂度是由嵌套层数最多的循环语句中最内层语句的频度f(n)决定。

eg：

a=0;

b=1; ①

for (i=1;i<=n;i++) ②

{

s=a+b;　　　　 ③

b=a;　　　　　 ④

a=s;　　　　　 ⑤

}

**分析：**

语句1的频度：1,

语句2的频度：n,

语句3的频度：n,

语句4的频度：n,

语句5的频度：n,

T(n) = 2+n+3n = 4n+2 = O(n).

1. **时间复杂度常见问题：**
2. **时间复杂度为O(1)：**

算法中语句的执行次数为常数，不受其他参数的影响

1. **不同情况分类：**

* **最好情况：**

算法运行次数最少的情况，该算法的时间复杂度

* **最坏情况：**

算法运行次数最多的情况，该算法的时间复杂度

* **平均情况：**

该算法所有情况的遍历次数相加 / 总情况数



* **空间复杂度计算：（详解）**

1. 和时间复杂度类似，但是空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度。

使用的也是大O记号 ；

1. **计算步骤：**
2. **忽略常数**，用O(1)表示；
3. **递归算法**的空间复杂度=递归深度N\*每次递归所要的辅助空间；
4. 对于**单线程**来说，递归有运行时堆栈，求的是递归最深的那一次压栈所耗费的空间的个数，因为递归最深的那一次所耗费的空间足以容纳它所有递归过程。

**eg：二分查找（递归方法实现）**

int BinarySearchRecursion(int arr[5], int lef, int rig,int aim)

{

int **mid** = lef + (rig - lef) / 2;

if (lef <= rig)

{

if (aim < arr[mid])

{

rig = mid - 1;

BinarySearchRecursion(arr, lef, rig, aim);

}

else if (arr[mid] < aim)

{

lef = mid + 1;

BinarySearchRecursion(arr, lef, rig, aim);

}

else if (aim == arr[mid])

{

return mid;

}

}

else

return -1;

}

**分析：**

时间复杂度 = O(log2n)

空间复杂度 = O(log2n)：每次递归调用都会创建一个变量mid






1. **冒泡排序：**

* **思想：**

相邻的两个数相互比较，依据大小互换位置——每循环一次，都会将当前无序数组中最大（最小）的元素放到数组的最后面。

* **代码实现：**

public void sort(){  
 int temp;  
 for(int i = 0; i < arr.length - 1; i ++){

**//每结束以此循环，都会将最大（最小）的元素放到最后面**  
 for(int j = 0; j < arr.length - 1 - i; j++){   
 if(arr[j] > arr[j+1]){  
 flag = true;  
 temp = arr[j];  
 arr[j] = arr[j+1];  
 arr[j+1] = temp;  
 }  
 }  
 if(!flag){ **//排序提前结束：本次for循环，遍历数组未发生元素互换**  
 break;  
 }  
 else {  
 flag = false;  
 }  
 }  
}

1. **选择排序：**

* **思想：**(找到剩余元素中，最小的元素，插到有序数组的最后面)

从第1个元素开始，找到最小的元素，和arr[0]交换；

从第2个元素开始，找到最小的元素，和arr[1]交换；

从第3个元素开始，找到最小的元素，和arr[2]交换；

……

从第n-2个元素开始，找到最小的元素，和arr[n-2]交换；

一共需要找n-1次最小值

* **代码实现：**

public void sort(){  
 int min;  **//最小值**  
 int minIndex;  **//最小值的索引值**  
  
 for(int i = 0; i < arr.length; i++){  
 min = arr[i];  
 minIndex = i;  
 for(int j =i+1;j < arr.length;j++ ){ **//找到第i个->最后一个元素中,最小的元素**  
 cycleIndex++;  
 if(min > arr[j]){  
 min = arr[j];  
 minIndex = j;  
 }  
 }  
 if(minIndex != i){ **//最小元素与第i个元素互换** arr[minIndex] = arr[i];  
 arr[i] = min;  
 }  
 }  
}

1. **插入排序：**

* **思想：**

将数组分为有序、无序两部分。通过遍历有序数组，将无序数组中第一个数，依据大小，插入到有序数组中;

**缺点：**当无序数组最后一个元素较小时，需要运行的次数变多

* **代码实现:**

public void sort(){  
 int insertVal; **//保存需要插入元素的值**  
 int insertIndex;   
 for(int i = 1; i < arr.length; i++){  
 insertVal = arr[i]; **//保存需要插入元素的值**  
 insertIndex = i - 1; **//有序数组，当前最后一个元素的索引**  
  **//遍历有序数组，寻找插入位置，找到位置后，需要将数组向后移动一个位置**

**//从小到大排序,有序数组是按照：从后向前遍历**  
 while (insertIndex >= 0 && (arr[insertIndex] > insertVal)){   
 arr[insertIndex + 1] = arr[insertIndex]; **//元素往后移动**  
 insertIndex--;  
 }  
  **//将需要插入的元素，插入到有序数组中**  
 arr[insertIndex+1] = insertVal;  
 }  
}

1. **希尔排序：解决插入排序的缺点**

* **思想：**

将数组的元素，按照一定的步长N进行分组，每组分别进行排序。再将步长=N/2，即：缩减分组数量（但每组的元素个数会增加，一开始时，每组只有2个元素），每组再进行排序…直到只剩下一组时，进行最后一次排序，就可以得到最后的排序结果。

* **步骤：**

（1）步长变更

（2）确定每个步长下的分组

（3）对每个分组进行排序：按照元素大小，遍历…

有两种排序方法：

* **交换法：**

**/\*\*  
 \* 交换法排序  
 \* 思路：（1）步长变更  
 \* （2）确定每个步长下的分组  
 \* （3）对每个分组进行排序：按照元素大小，遍历、交换位置  
 \*/**  
public void sortExch(){  
 int temp; **//暂时存储交换的元素**  
  
 **//1、步长:stepSize，每次对半减小**  
 for(int stepSize = arr.length/2; stepSize > 0; stepSize /= 2){  
  **//2、不同步长下的分组：i表示每组第1个元素的索引，无论arr元素个数=奇、偶数，都可以完成所有数据的排序** for(int i = stepSize; i < arr.length; i++){  
 **//3、对每个分组进行排序：每组元素之间的间隔为stepSize，从后往前遍历数组** for(int j = i - stepSize; j >= 0; j -= stepSize){  
 if(arr[j] > arr[j+stepSize]){  
 temp = arr[j];  
 arr[j] = arr[j+stepSize];  
 arr[j+stepSize] = temp;  
 }  
 }  
 }  
 }  
}

* **位移法：**

**/\*\*  
 \* 位移法排序：  
 \* 本质：将交换法中的步骤（3）：元素交换，改为插入排序  
 \*/**  
public void sortOffset(){  
 int temp;  
 int insertVal;  **//保存需要插入元素的值**  
 int insertIndex; **//需要插入元素的索引**  
  **//1、步长**  
 for(int stepSize = arr.length/2; stepSize > 0; stepSize /=2){  
  **//2、分组：从每一组的第1个元素开始，使用插入排序**  
 for(int i = stepSize; i < arr.length; i++){  
  **//3、插入排序**  
 insertVal = arr[i]; **//保存需要插入元素的值**  
 insertIndex = i;  **//需要插入元素的索引**  **//若，当前要插入的值 < 有序数组最后一个元素，才进行元素后移**  
 if(arr[insertIndex] < arr[insertIndex - stepSize]){  
 while (insertIndex - stepSize >= 0 && (insertVal < arr[insertIndex - stepSize])){  
 arr[insertIndex] = arr[insertIndex - stepSize];  
 insertIndex -= stepSize;  
 }  
  **//在有序数组最后面添加当前要插入的值** arr[insertIndex] = insertVal;  
 }**//当前要插入的值 >= 有序数组最后一个元素：不需要修改，因为 insertVal = arr[i],要插入的元素就在该位置**  
 }  
 }  
}

1. **快速排序：是冒泡法的改进，比希尔排序块**

* **思路：**
* 寻找数组中的一个元素，将数组进行切分成左右两边。用该元素作为基准，将作于左右两边的元素进行互换（左边大于该元素的，和右边小于该元素的，进行互换）。

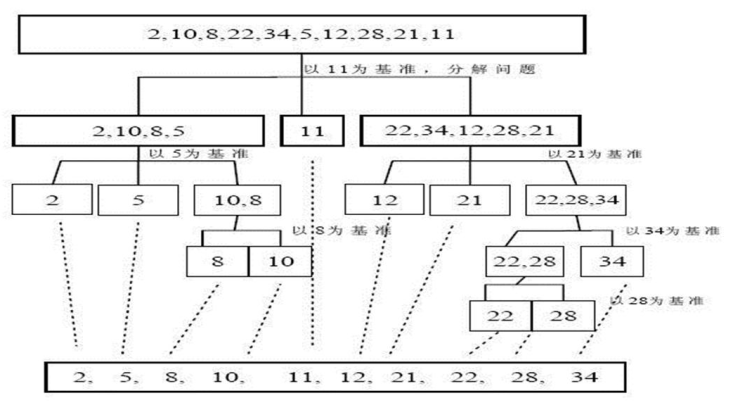
**终止一次遍历的条件：**当前的左元素标签 >= 右元素标签，即：完成一次数组遍历

* 遍历一次数组后，该元素左右两边的元素都满足：

左边元素<该元素；

右边元素>该元素；

* 再将左右两边的元素，按照上面的思路，进行遍历，直到所有的元素排序完成



* **快速排序优化：**

1. 当分割后，左右数组的元素小于某个数时，使用插入排序

因为，在数据量较小的时候，插入排序时间复杂度更低

1. 进行每一次快速排序后，把那些和基准数相等的元素都放到基准数的周围，下次就可以不用对这些元素进行排序了。

* **代码实现：**

public void sort(int[] arr, int left, int right){  
 int l = left;  **//当前数组最左边元素索引**  
 int r = right;  **//当前数组最右边元素索引** int privot = arr[(left + right) / 2]; **//数组的中间元素，作为切分点**  
 int temp = 0;  
  **//0、开始遍历一次数组：while里，进行一次privot左右两边的元素调整**  
 while(l < r){  
  **//1、找到一个arr，左边>privot 和 右边<privot的元素索引**  **//左边>privot**  
 while (arr[l] < privot){  
 l++;  
 }  
  **//右边<privot**  
 while (arr[r] > privot){  
 r--;  
 }

**//终止一次数组遍历的条件：l>=r**

**说明privot左右两边的元素满足：左边全部<privot，右边全部>privot**  
 if(l >= r){  
 break;  
 }

**//2、交换两个元素**  
 temp = arr[l];  
 arr[l] = arr[r];  
 arr[r] = temp;

**//3.1、交换后，如果是privot和arr[l]元素交换**

**（说明，此时privot右边的元素全部大于privot）**  
  **// 需要对右边索引r前移一位：保证privot这个参考点同步移动**  
 if(arr[l] == privot){  
 r--;  
 }  
  **//3.2、交换后，如果是privot和arr[r]元素交换**

**（说明，此时privot左边的元素全部大于privot）；**  **// 需要对右边索引l后移一位：保证privot这个参考点同步移动**  
 if(arr[r] == privot){  
 l++;  
 }  
 }  
  
  **//4、若l = r，说明已经遍历完一次数组，需将l：后移一位，r：前移一位**  
 **//否则后面的递归调用时，会出现栈溢出。  
 //后面的递归调用：不会对privot进行排序，只会对privot两边的元素进行排序**  
 if(l==r){  
 l++;  
 r--;  
 }

**//5、进行递归：对privot两边的元素进行排序**  
  **//对privot左边，进行递归**  
 if(left < r){  
 sort(arr,left,r);  
 }  
  **//对privot右边，进行递归**  
 if(right > l){  
 sort(arr,l,right);  
 }  
}

1. **归并排序：**

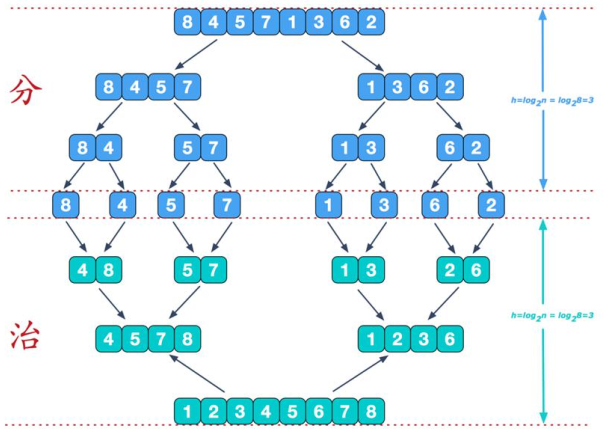
* **思想：**

**采用分治的思想**，将原数组分成若干个最小的数组（每组只有两个元素），再分别进行组合（需要一个缓存数组temp[]，暂存合并之后的结果）

* **分类：**
* **自底向上：**从原数组的0-1，1-2…开始：两两合并 🡪 四四合并 🡪 八八合并…，合并时从排好序的两个数组中最前面元素的大小，拿出较小的元素放到缓存数组——保存这两个数组排序后的数据。

（自底向上的代码量比自顶向下的少一些，也不需要使用递归对原数组进行划分，只需要使用两个for控制索引数组的位置）

* **自顶向下：**从上到下，将原数组划分至只有一个元素的数组，然后对划分的数组进行排序，两个数组合并至tmp缓存数组中，在将temp[]放回原数组的索引位置。



* **代码实现：（自顶向下）**

**/\*\*  
 \* 归并排序：分数组 + 合并数组  
 \*/**  
public void sort(int[] arr, int left, int right, int[] temp){  
 if(left < right){  
 int mid = (left + right) / 2;  
  **//1、左边数组分割：递归**  
 sort(arr,left,mid,temp);  
 **//2、右边数组分割：递归**  
 sort(arr,mid + 1,right,temp);  
  **//3、合并数组：在完成上述的递归分割之后，才开始合并数组**  
 merge(arr,left,mid,right,temp);  
 }  
}

**/\*\*  
 \* 合并数组  
 \*/**  
private void merge(int[] arr, int left, int mid ,int right,int[] temp){  
 int leftArrIndex = left;  **//左边数组的初始索引（包含mid节点）**  
 int rightArrIndex = mid + 1;  **//右边数组的初始索引** int tempIndex = 0; **//temp[]的初始索引**  
  **//1、先合将原数组左右两边能够合并的元素合并起来，放到temp[]中** while (leftArrIndex <= mid && rightArrIndex <= right){  
  **//左、右两边从头开始，挑选较小的元素，放到temp[]中**  
 if(arr[leftArrIndex] <= arr[rightArrIndex]){  
 temp[tempIndex] = arr[leftArrIndex];  
 leftArrIndex++;  
 tempIndex++;  
 }  
 else {  
 temp[tempIndex] = arr[rightArrIndex];  
 rightArrIndex++;  
 tempIndex++;  
 }  
 }  
  **//2、再将左边或右边数组中剩余的元素合并起来，放到temp[]中** while (leftArrIndex <= mid){  
 temp[tempIndex] = arr[leftArrIndex];  
 leftArrIndex++;  
 tempIndex++;  
 }  
 while (rightArrIndex <= right){  
 temp[tempIndex] = arr[rightArrIndex];  
 rightArrIndex++;  
 tempIndex++;  
 }  
  **//3、将temp[]中的结果，填充回原数组arr[]**  **//第一次合并：tempLeft = 0，righ=1 ；tempLeft = 2，righ=3 …；tempLeft = n-2，righ=n-1；  
 //......  
 //最后一次合并：tempLeft = 0，righ=n-1 ；**  
 tempIndex = 0;  
 int tempLeft = left;  **//存储要放到原数组的起始位置**  
 while (tempLeft <= right){  
 arr[tempLeft] = temp[tempIndex];  
 tempLeft++;  
 tempIndex++;  
 }  
}

1. **基数排序：（桶排序）**

* **思想：**

基数排序称为“分配式排序”（桶排序），通过对各个元素的个、十、百、千…位分配到对应的桶数组中，达到排序的目的。属于稳定的排序算法,用空间换时间，消耗的内存较大——因为有10个和元素组一样大小的同数组。一般用在所有元素均为正数的情况，若有负数，需要做改进（取绝对值）。

将数组中所有的数统一成一样的长度，不够长度的前面补零。从最低位个位开始排序：个 🡪 十 🡪 百 🡪 千 🡪 万…需要遍历数组的次数=数组中最大数字的位数，每遍历完一次数组，需将同数组中的元素放回原数组，下一次遍历数组，再从原数组中取出元素。

eg：max(arr) = 1203，则需要遍历4次数组

* **代码实现：**

**/\*\*  
 \* 思路：  
 \* 1、寻找数组中最大的元素-->获取位数digits  
 \* 2、for循环，将原数组按照个、十、百、千...的顺序遍历digits遍，放到桶中（每次遍历完成后，都从桶中取出结果，放回原数组）  
 \*/**  
public void sort(){  
 int[][] bucketArr = new int[10][arr.length]; **//桶数组**  
 int[] bucketArrNum = new int[10];  **//记录每个桶中的元素个数**  
  **//1、获取最大的元素-->得到长度**  
 int max = 0;  
 for(int i = 0; i <arr.length; i++ ){  
 if(max < arr[i])  
 max = arr[i];  
 }  
  
  **//2、进行桶排序：需要遍历 的顺序遍历digits遍 次数组**  
 int maxLenght = (max + "").length();  **//最大元素的位数** int digits = 0;  **//元素某个位：个、十、百、千...** int arrIndex = 0;  **//原数组的索引** for(int i = 0, unit = 1; i < maxLenght; i++, unit \*= 10){  
  **//3、元素放入桶中：按照个、十、百、千...的顺序遍历 digits 遍**  
 for(int j = 0; j < arr.length; j++){  
 digits = arr[j] /unit % 10;  
 bucketArr[digits][bucketArrNum[digits]] = arr[j]; **//元素放入对应的桶中** bucketArrNum[digits]++;  **//记录对应桶中的元素个数** }

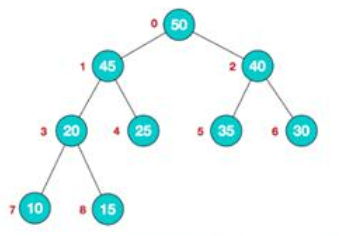
**//4、遍历一次数组后：将元素放回原数组,等待下次遍历**  
 for(int j = 0; j < bucketArrNum.length; j++){  
 if(bucketArrNum[j] != 0){  
 for(int x = 0; x < bucketArrNum[j]; x++){  
 arr[arrIndex++] = bucketArr[j][x];  
 }  
 }  
 bucketArrNum[j] = 0;  **//桶数组取完后，需要清零桶数组的计数** }  
 arrIndex=0;  **//清零原数组的索引值**  
 }  
}

1. **堆排序**

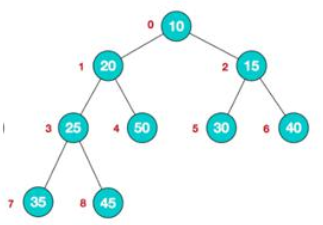
* **思想**：
* 将数组放入二叉树中（可以说是堆，大顶堆、小顶堆）

该二叉树的特点：

**大顶堆：**所有父节点，都大于其左右子树



**小顶堆：**所有父节点，都小于其左右子树



* **数组索引值和二叉树元素的索引值关系：**

数组索引值：index = 0（从0开始），大小：arr.length

二叉树中：

当前节点下的左子树索引值：2\*index + 1

当前节点下的右子树索引值：2\*index + 2

二叉树中，倒数第二层的索引值：[(arr.length/2) - 1]

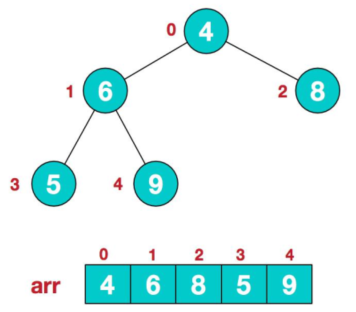
* **如何进行排序：**
* 先将无序数列排成堆（二叉树—大顶堆、小顶堆）
* 再将排好序的堆顶元素和现数组最后一个元素交换，然后再对除最后一个元素外的数组进行堆调整（实际上就是将根节点元素取出）
* 将堆调整后的数组的堆顶元素，和数组倒数第二个元素进行交换，再进行除末尾两个元素外的数组进行堆调整
* .....以此循环，直到所有元素排完

**注：**大顶堆排完之后，结果为从小到大排序

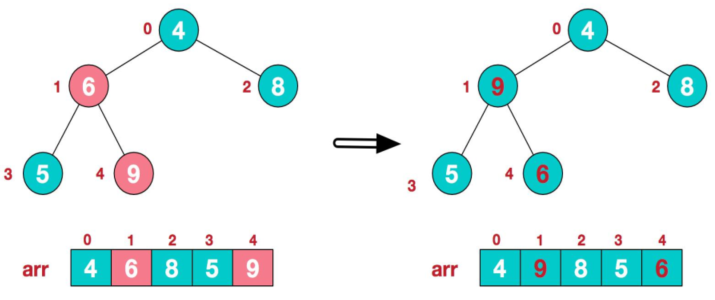
小顶堆排完之后，结果为从大到小排序

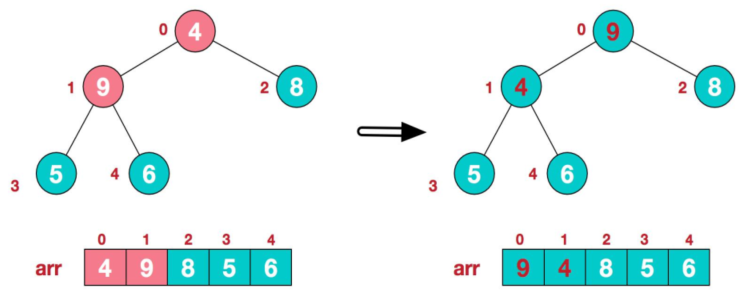
* **示例：堆排序的过程（大顶堆）**

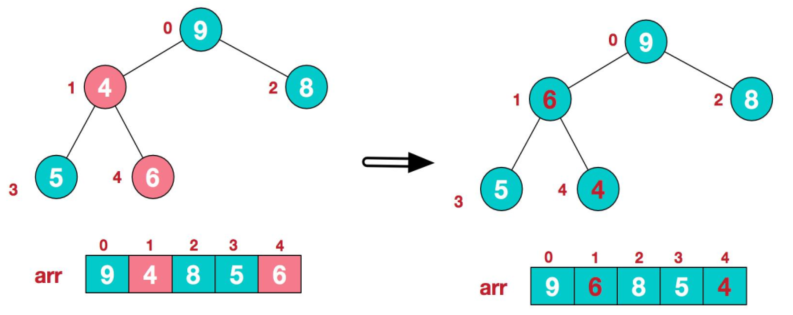
1. **无序队列如下：**



1. **进行局部堆排序：（第一次进行堆排序：自底向上）**

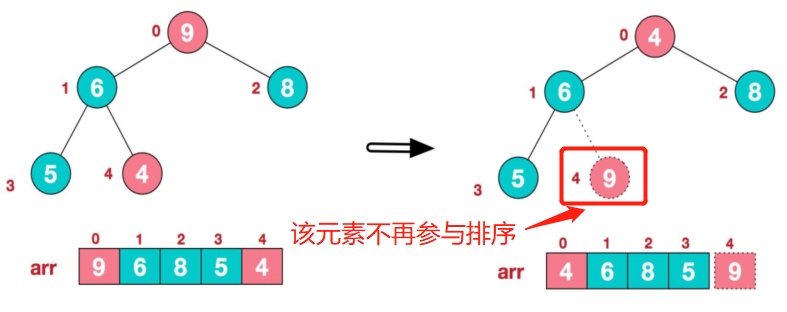


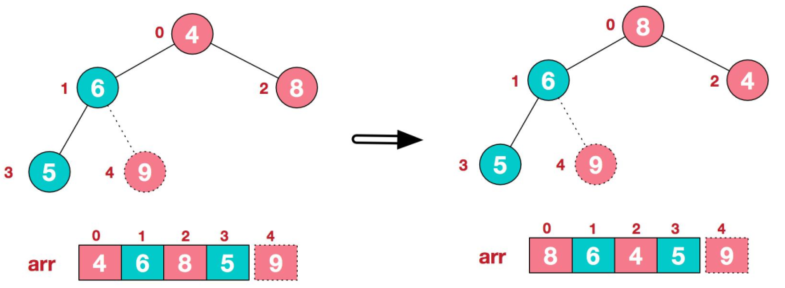




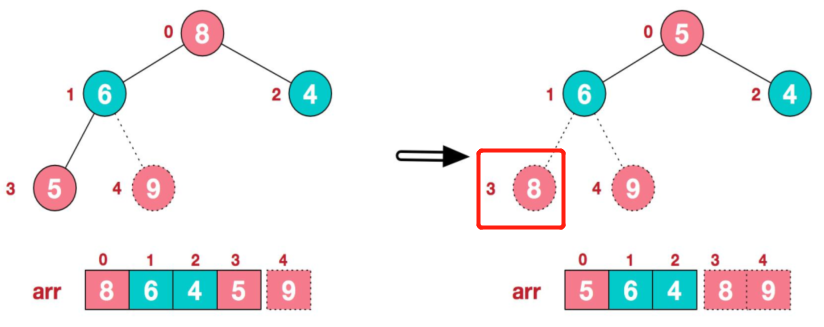
1. **将当前最大元素放到数组末尾，并进行堆调整（自顶向下）**

**（根节点和底层的末端子节点互换位置，然后取出）**

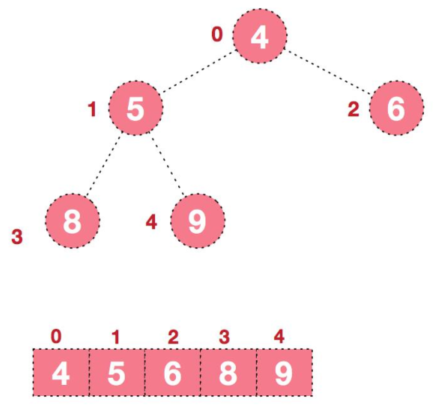




1. **重复上述排序：**



1. **最终结果：**





1. **排序算法总结：**

* **排序算法稳不稳定：**

**稳定：**原数组中，未排序前：a在b前面

排序后：a依然在b前面

**不稳定：**原数组中，未排序前：a在b前面

排序后：a可能在b前后面

* **内排序、外排序：**

**内排序：**所有排序操作都在内存中完成

**外排序：**数据量大，将数据存放在磁盘中，通过磁盘、内存数据交换进行排序；

* **时间复杂度、空间复杂度：**

**时间复杂度：**算法执行耗费的时间

**空间复杂度：**算法执行所占用的内存空间

* **各个排序算法的复杂度：**



## 查找

1. **二分查找：**

* **思想：**

数组必须是有序的，才能使用二分查找。

* **代码实现：**

public List<Integer> searche(int[] arr, int left, int right, int findVal){  
 **//如果满足：终止条件，left > right，退出递归——没有找到元素**  
 if(left > right){  
 return new ArrayList<Integer>();  
 }  
 int midIndex = (left + right) / 2;  
 int midVal = arr[midIndex];  
  **//右递归查找**  
 if(findVal > midVal){  
 return searche(arr,midIndex + 1,right,findVal);  
 }  
 **//左递归查找**  
 else if(findVal < midVal){  
 return searche(arr,left,midIndex - 1,findVal);  
 }  
 **//找到相应的元素：继续搜索该元素左右两边的相同元素**  
 else{  
  **//左查找**  
 int tempIndex = midIndex - 1;  
 while (true){  
 if(tempIndex < 0 || arr[tempIndex] != findVal){  
 break;  
 }  
 res.add(tempIndex);  
 tempIndex--;  
 }  
 **//插入第一次找到的值得索引**  
 res.add(midIndex);  
 **//右查找**  
 tempIndex = midIndex + 1;  
 while (true){  
 if(tempIndex > arr.length || arr[tempIndex] != findVal){  
 break;  
 }  
 res.add(tempIndex);  
 tempIndex++;  
 }  
 return res;  
 }  
}

1. **插值查找：**

* **思想：**

二分查找的改进版，自动定位查找的位置，查找次数较少，一次就可以定位到查找的元素。

* **适用于：数组有序、**数据量大、数据分布均匀的数组，当数据分布不均匀，查找效率并不高。
* **使用的公式：**

 ，说明如下：

* 前提：原数组排序方式：从小到大
* left：当前左索引
* right：当前右索引
* findVal：查找的元素数值
* midIndex：最终定位索引位置
* **代码实现：**

public List<Integer> search(int[] arr, int left, int right, int findVal){  
 **//提前结束查找：没有找到元素**  
 if(left > right || findVal < arr[left] || findVal > arr[right]){  
 return new ArrayList<Integer>();  
 }  
  **//改进公式：定位查找的元素**  
 int midIndex = left + (findVal - arr[left]) \* (right - left) / (arr[right] - arr[left]);  
 int midVal = arr[midIndex];  
 if(findVal > midVal){  
 return search(arr,midIndex + 1,right,findVal);  
 }  
 else if(findVal < midVal){  
 return search(arr,left,midIndex - 1,findVal);  
 }  
 else { **//找到查找的元素**  
  **//左查找**  
 int tempIndex = midIndex - 1;  
 while (true){  
 if(tempIndex < 0 || arr[tempIndex] != findVal){  
 break;  
 }  
 res.add(tempIndex);  
 tempIndex--;  
 }  
  **//插入第一次找到的值得索引**  
 res.add(midIndex);

**//右查找**  
 tempIndex = midIndex + 1;  
 while (true){  
 if(tempIndex > arr.length || arr[tempIndex] != findVal){  
 break;  
 }  
 res.add(tempIndex);  
 tempIndex++;  
 }  
 return res;  
 }  
}

1. **斐波那契查找：（黄金分割法）**

* **思想：**

**前提：**数组有序

和二分查找、插值查找相似，**只不过是改变了中间节点的计算方式**，采用黄金分割点附近的值，作为查找的中间节点，需要对原始数组进行扩充（因为，原始数组，可能长度会小于黄金分割点的数值）。

**斐波那契数列的特点：**

1，1，2，3，5，8，13，21，34，55……

当n趋于无穷大时，

**公式：**

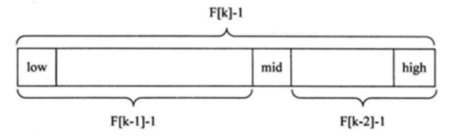
 ，说明：（通常low=0， mid表示黄金分割点）

F表示：斐波那契数列

K表示：第几个斐波那契元素

F(k)-1表示：扩充数组的长度

将数组分成两段：F(k-1)-1、F(k-2)-1 ，（注意：F(k-1) > F(k-2)）





**查找步骤：**

（1）如果与给定关键字相同，则查找成功，返回在表中的位置；

（2）如果给定关键字大，向右查找并**减小2个**斐波那契区间；

（3）如果给定关键字小，向左查找并**减小1个**斐波那契区间；

（4）重复过程，直到找到关键字（成功）或区间为空集（失败）

* **代码实现:**

public int[] getFibonaciiArr(int num){  
 int[] fibo = new int[num];  
 fibo[0] = 1;  
 fibo[1] = 1;  
 for(int i = 2; i < num; i++){  
 fibo[i] = fibo[i-1] + fibo[i-2];  
 }  
 return fibo;  
}

public int **search**(int[] arr, int findVal){  
 int high = arr.length - 1;  
 int low = 0;  
 int fiboIndex = 0; **//斐波那契数列分割值索引**  
 int mid = 0;  
 int[] fiboArr = getFibonaciiArr(20);**//获取斐波那契数列，用于得到分割点**  
  **//1、找到数组的黄金分割点: 获取斐波那契分割数的下标**  
 while (high > fiboArr[fiboIndex] - 1){  
 fiboIndex++;  
 }  
  **//2、数组的黄金分割点——fiboArr[fiboIndex]，可能大于原数组长度，需将原数组扩充**  
 int[] arrTemp = Arrays.*copyOf*(arr,fiboArr[fiboIndex]);  
 for(int i = high + 1; i < arrTemp.length; i++){  
 arrTemp[i] = arr[high]; **//用数组最高位填充**  
 }  
  **//3、查找数据**  
 while (low <= high){  
 mid = low + fiboArr[fiboIndex - 1] - 1;  
 if(findVal < arrTemp[mid]){  **//查找的数在数组的左边**  
 high = mid - 1;  
 fiboIndex--; **//右边查找，需要-1**  
 }  
 else if(findVal > arrTemp[mid]){  **//查找的数在数组的右边**  
 low = mid + 1;  
 fiboIndex -= 2; **//左边查找，需要-2**  
 }  
 else{  **//找到元素**  
 if(mid <= high){  
 return mid;  **//查找的元素等于mid**  
 }  
 else {  
 return high;   
 }  
 }  
 }  
 return -1; **//没找到元素**  
}

1. **哈希表：（散列） 可用于制作缓存产品**

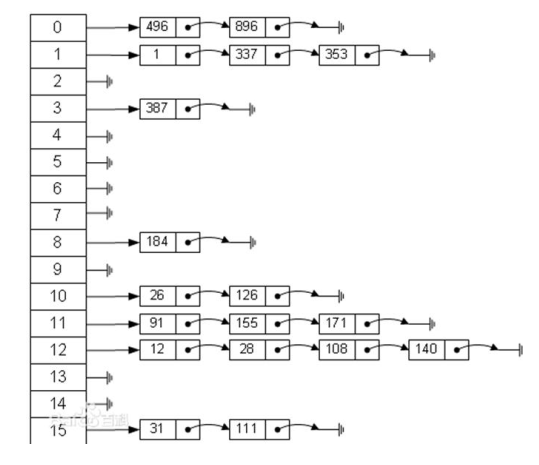
* **思想：**

**哈希表组成：**数组 + 链表

主要是操作哈希码来访问哈希表中对应位置的链表数据。

**关键：**链表有自己的增删改查函数，哈希表类根据所给的id等数据调用链表的增删改查函数来操作数据。

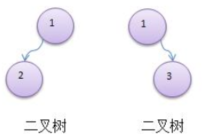
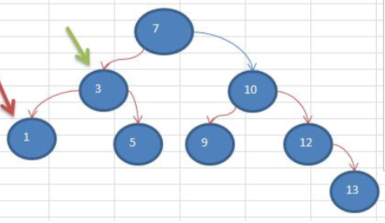
**注意：**初始化哈希表数组时，不仅是初始化数组的大小，还需要初始化数组每个元素对应的链表——给每个元素new一个链表对象。



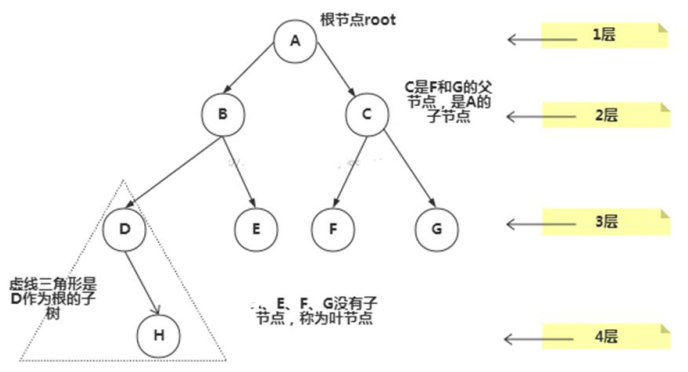
* **代码实现：Java中对应的封装的类为：HashMap**

1. **新建节点Node**
2. **新建单向链表LinkedList：**由节点Node构成，并写出链表元素的增删改查方法；
3. **构建哈希表HashTable：**由数组+LinkedList构成，写出散列函数（计算哈希码的函数），根据id等数据，调用链表的方法，写出哈希表的增删改查方法。
4. **二叉树**

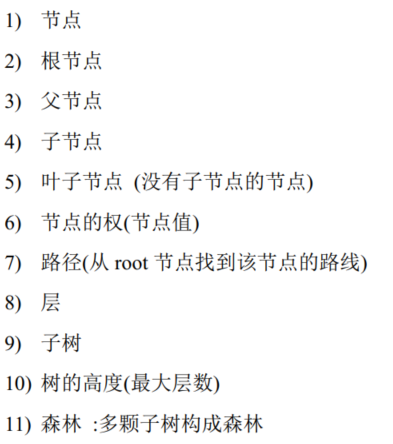
* **思想：**
* **二叉树和数组对比：**
* **数组：**大小固定，使用下标访问数组，访问数组的速度快，但是插入数值时，需要整体移动，效率低。所谓的动态数组，也需要进行扩容——将数组复制到扩容后的数组中去。
* **二叉树：**改进链表查找速度慢的问题，二叉树将将数据排好 （左子树 < 右子树）。树的高度=最大层数



* **二叉树知识：**
* **结构：**

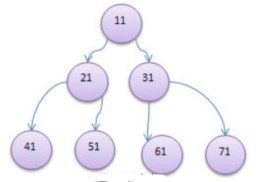


* **名词：**



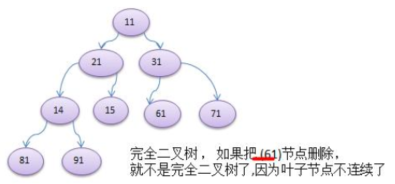
* **满二叉树：**

节点数 = 2n-1个，n为树的最大层数



* **完全二叉树：**

一共k层，除了第k层之外，其他层的节点都达到最大个数，第k层的节点都是左连续（集中在最左边）



* **AVL平衡二叉树：**

左子树和右子树的高度差不超过1.



* **遍历方式： 用递归的方式遍历**
* **前序遍历：**

**父节点** -> 左子树 -> 右子树

* **中序遍历：**

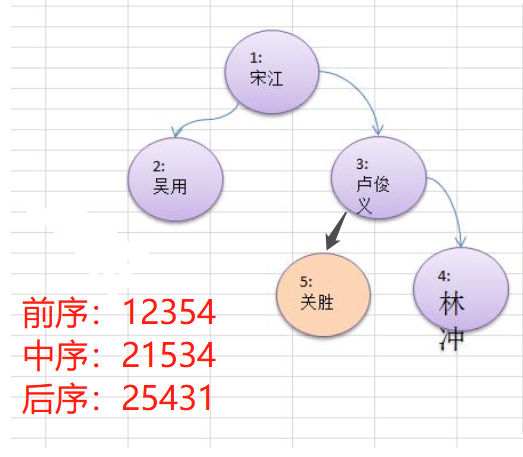
左子树 -> **父节点** -> 右子树

* **后序遍历：**

左子树 -> 右子树 -> **父节点**

**总结：区别在于父节点的位置不同，但左子树比右子树先遍历。**

**都是先遍历到叶节点，再向上判断其他节点是否符合要求。**



* **遍历查找方式：用递归的方式遍历查找**
* **前需遍历查找：**

父节点 -> 左子树 -> 右子树

* **中序遍历查找：**

左子树 -> 父节点 -> 右子树

* **后序遍历查找：**

左子树 -> 右子树 -> 父节点

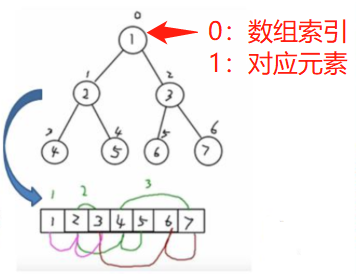
**总结：1、区别在于父节点的位置不同，但左子树比右子树先遍历**

**2、查找：都是先遍历到叶节点，再向上判断其他节点是否**

**符合要求。**

* **顺序存储二叉树：（只考虑完全二叉树）**

一维数组 **<==>** 二叉树，相互转换。



数组起始索引：index=0

顺序二叉树中：

**左子树节点的索引：**2\*index+1

**右子树节点的索引：**2\*index+2

**左叶子节点的父节点的索引：**（index-1）/2

**代码实现：（顺序二叉树）**

**/\*\*  
 \* 前序遍历顺序二叉树（一维数组）  
 \* 调用时：传入0，即：传入数组的第一个元素，作为root节点  
 \*/**  
public void preorderTraversal(int index){  
 if(arr.length == 0){  
 System.*out*.println("Arr is null");  
 return;  
 }  
 **//输出：当前节点（父节点）**  
 System.*out*.println(arr[index]);  
  **//左递归：判断是否还有左子树，（arr.length）为该二叉树的节点个数**  
 if(2\*index + 1 < arr.length){  
 preorderTraversal(2\*index + 1);  
 }  
 **//右递归：判断是否还有右子树，**  
 if(2\*index+2 < arr.length){  
 preorderTraversal(2\*index + 2);  
 }  
}

* **线索化二叉树：**



**将左边的二叉树转换成中序二叉树：**所谓的中序二叉树，就是采用特殊的中序遍历方法的二叉树，遍历得到的结果和普通二叉树的中序遍历结果一样。

* **关于叶节点的几个概念：**

叶节点需要前驱节点、后继节点：

**前驱节点：**该叶节点的左子树指向前一个节点，例如上图，10号节点的前驱节点为3

**后继节点：**该叶节点的右子树指向后一个节点，例如上图，10号节点的后继节点为1

* **代码实现：（二叉树）**
* **构成：**
* **总体上：**和实现HashTable的方式类似，先创建Node节点类，再用Node节点构成二叉树类。
* **方法的实现：（**前序、中序、后序**）**
* **Node节点类：**创建相应的前序、中序、后序等方法。
* **二叉树类：**通过封装、调用Node类的前序、中序、后序等方法🡪得到用户操作的前序、中序、后序等方法。

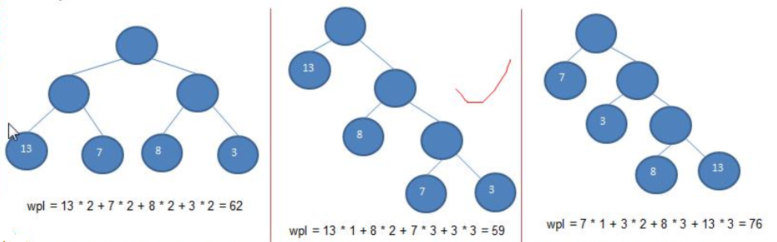
1. **赫夫曼树**

* **思想：**
* **所有叶子节点所构成的一棵树，其带权路径长度最小**（weighted path length--**wpl**），则该树为**赫夫曼树**

**本质：就是将较小的元素（权值）放到底层，较大的权值放到上面。**

* **wpl的计算：**

wpl = sum（叶子节点i的权值 \* （所在的层数-1））



* **代码实现：**
* **实现步骤：**

1. 用ArrayList来存放数组
2. 将ArrayList元素从小到大排序
3. 取前两个元素，构成一棵树，该树的父节点=两个子节点之和
4. 删除数组中的前两个元素（因为，已经用过了），**将父节点的值放到ArrayList最后面，再对ArrayList进行排序**
5. 重复1 - 4步骤，最后剩下一个元素时结束（这个元素为赫夫曼树的根节点）

例如：原数组 ：13,7,8,1,3,29,6 ，其赫夫曼树为：

67

-------

29 38

-------

15 23

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

7 8 10 13

------

4 6

-----

1 3

1. **赫夫曼编码：**

* **思想：**
* **根据数据中字符出现的频率，从大到小开始排序，出现越多的字符，其索引数值越小**。再利用赫夫曼树对字符进行编码，每个字符对应一个编码，并且得到的编码为前缀编码（即：任意字符编码，都不会出现在其他字符编码的前面一部分，也就不会产生二义性）。

* 赫夫曼编码是不稳定的编码，因为当不同字符串出现频率相同时，从大到小排序后，对应字符的索引不确定。进而导致字符的赫夫曼编码不同。但是最终的wpl都是一样的。

* **实现步骤：**

1. 先将字符串—>转成对应的ASCII**（使用.getBytes()方法）**
2. 根据字符的ASCII码，创建赫夫曼树**（此时，ASCII为该树的叶节点）**
3. 统计该字符串中，每个字符出现的频率
4. 自定义：经过左子树:0，右子树:1，**即：赫夫曼编码由0、1组成（或者说赫夫曼编码是按照字节来处理的，因此可以处理所有类型的文件）**
5. 遍历赫夫曼树，找到赫夫曼树每个叶节点，使用**hashMap<String，Byte>记录对应叶节点和所经过左右子树的0、1情况。**

**（hashMap<字符，赫夫曼编码>**，**即：获得字符对应的赫夫曼编码）**

1. **获得字符串的赫夫曼编码（无损压缩）：**将字符串转成ASCII码，然后通过查询对应字符的赫夫曼编码表，获得每个字符的编码，再全部拼装起来就得到该字符串的赫夫曼编码，将所得到的编码按照每8位，取反，转成int型，就获得字符串压缩后的数据。
2. **关于解码：**使用IO流，将数据按照1-6步骤进行压缩。将压缩后的数据存入文件中，再将对应字符的赫夫曼编码放进去（否则无法解压），解压的时候，需要根据对应字符的赫夫曼编码进行解码。获得对应的byte[]数组，再写出到文件即可获得原文件。

* 赫夫曼编码用于压缩，当文件中存在大量重复的数据时，压缩效率高。

1. **二叉排序树（BST：binary sort tree）**

* 思想：
* 对比数组，**二叉排序树添加、查找速度更快。与链表相似。**
* 二叉排序树的特点：任何非叶子节点都有，

**左子节点 < 当前节点 < 右子节点**

**(若有相同值，可存放在右子节点，或左子节点)**

* 将数组{7，3，10，12，5，1，9}变成二叉排序树：



* **二叉排序树，使用中序遍历，就会得到从小到大的排序结果。**

上图，中序遍历：1，2，3，5，7，9，10，12

* **二叉排序树删除节点的情况：**
* **删除叶子节点：**

先找到删除节点和他的父节点；

判断该节点在其父节点的左或右子树；

让父节点的左或右子树 = null；

* **删除只有一颗子树的节点：**

先找到删除节和他的父节点；

判断该节点拥有的子树是左或右子树；

判断该节点在其父节点的左或右子树；

将父节点的左或右子树 = 该节点子树；

* **删除有两颗子树的节点：**

先找到删除节点和他的父节点；

找到删除节点的右子树的最小节点（**即：找右子树的最底层的左子树节点）；并用变量temp保存该值**

删除这个最小节点，并令删除节点的值 = temp；

* **可能存在的问题：**

当用数组{1，2，3，4，5，6}直接添生成二叉排序树，会导致最终得到的树缺失左子树

**（改进：看AVL树）**

1. **平衡二叉树（AVL树）**

* **思想：**
* **实质上：**是二叉排序树的改进版，避免二叉排序树在添加节点时可能出现某一侧的子树缺失现象。

（**左右两棵子数的高度差的绝对值不能超过1**）

* **AVL树比BST树（二叉排序树）多出来的方法：**

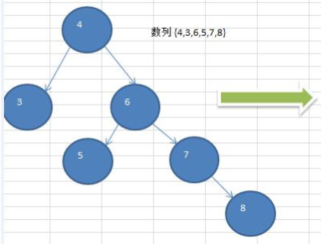
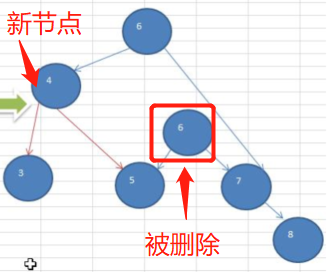
**无论为左右旋转、双旋转，都是在新增一个节点的之后，调用，以保证添加完所有节点之后，最终生成的树是AVL树。**

* **左旋转：**

当该节点的右子树高度 > 该节点的左子树高度时，需左旋转

**步骤：**

public void leftRotate(){  
 **//1、新建节点：其值为当前节点的值** Node newNode = new Node(this.value);  
 **//2、新建节点的左子树 = 当前节点的左子树**  
 newNode.leftNode = this.leftNode;  
 **//3、新建节点的右子树 = 当前节点右子树的左子树**  
 newNode.rightNode = this.rightNode.leftNode;  
 **//4、当前节点的值 = 当前节点右子树的值**  
 this.value = this.rightNode.value;  
 **//5、当前节点的右子树 = 当前节点的右子树的右子树**  
 this.rightNode = this.rightNode.rightNode;  
 **//6、当前节点的左子树 = 新建节点**  
 this.leftNode = newNode;  
}

**叶子节点的特点：**子树**最左边的叶子节点**是该子树的**最小值**

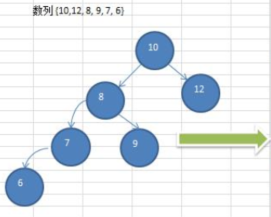
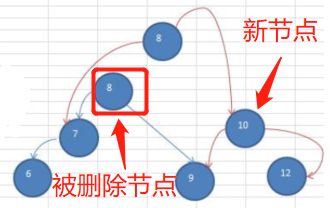
子树**最右边的叶子节点**是该子树的**最大值**

* **右旋转：（和左旋转类似）**

当该节点的左子树高度 > 该节点的右子树高度时，需右旋转

**步骤：**

public void rightRotate(){  
 **//1、新建节点：其值为当前节点的值**  
 Node newNode = new Node(this.value);  
 **//2、新建节点的右子树 = 当前节点的右子树**  
 newNode.rightNode = this.rightNode;  
 **//3、新建节点的左子树 = 当前节点左子树的右子树** newNode.leftNode = this.leftNode.rightNode;  
 **//4、当前节点的值 = 当前节点左子树的值** this.value = this.leftNode.value;  
 **//5、当前节点的右子树 = 新建节点**  
 this.rightNode = newNode;  
 **//6、当前节点的左子树 = 当前节点左子树的左子树**  
 this.leftNode = this.leftNode.leftNode;  
}

* **双旋转：**
* **该节点右子树的左子树高度 > 该节点右子树的右子树高度**

**步骤：**

if(this.rightNode != null && **this.rightNode.leftTreeHight() > this.rightNode.rightTreeHight()**){  
 **//该节点的右子树先进行右旋转**  
 this.rightNode.leftRotate();  
  **//再对该节点进行左旋转**  
 this.leftRotate();  
}

* **该节点左子树的右子树高度 > 该节点左子树的左子树高度**

**步骤：**

if(this.leftNode != null && this.leftNode.rightTreeHight() > this.leftNode.leftTreeHight()){  
 **//该节点的左子树先进性左旋转**  
 this.leftNode.leftRotate();  
 **//再进行右旋转**  
 this.rightRotate();  
}

**双旋转示意图：**



* **如何生成AVL树：**

1. 以二叉排序树为基础，添加新节点
2. 在添加每一个新节点之后，都必须判断整棵树的左右子树高度（以root节点为起点）
3. 进行相应的左右旋转、双旋转操作，再添加新的节点。

* **二叉树的子树高度计算 ：采用递归来计算**

**//计算以该节点为根节点的树高度**  
public int hight(){  
 return Math.*max*(this.leftNode == null ? 0: this.leftNode.hight(),  
 this.rightNode == null ? 0 : this.rightNode.hight()) + 1;  
}

1. **多叉树：2-3树、2-3-4树、B树、B+树、B\*树**

* **思想：弥补二叉树的缺点**
* **二叉树缺点：**

二叉树需要加载到内存，当二叉树节点很多时，构建二叉树需要进行多次的IO操作，对于构建二叉树的速度有影响，并且二叉树的高度很高，导致增删改查的操作速度下降。

* **多叉树：**

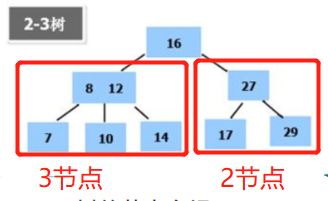
是二叉树的改进版，即：每个节点可以有多个子节点，以此减少树的高度

* **2-3树：**

**特点：**所有的叶子节点都在同一层，由2节点 + 3节点构成2-3树。

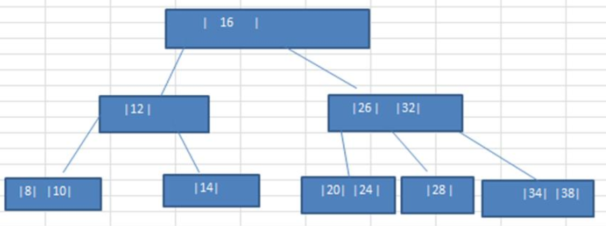
**2节点（3节点），只能同时有2（3）个节点或者同时没有节点。**

插入新节点时，可能需要先上拆、再下拆节点来调整树，使得构建的树满足2-3树的要求。



* **2-3-4树：**

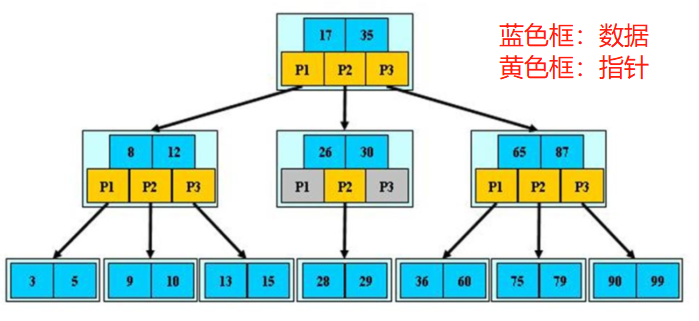
**特点：**2-3树的改进版



* **B树：（Balance 树，B-tree，用于MySQL的索引）**
* B树的阶：节点的最多子节点个数。

eg：2-3树的阶=3， 2-3-4树的阶=4

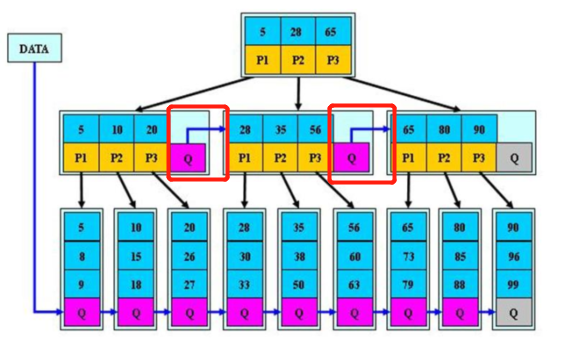
* **B树的每一层都有存放数据**（下图中的蓝色框）
* **B树的搜索：**从root开始，对每个节点内的关键字进行**二分查找**，找到则结束查找，否则一直往下，直到查到叶子节点为止。



* **B+树：**
* B+树是B树的改进版，搜索方式也是**二分查找**。
* **B+树的所有数据都存放在叶子节点**（即：**稠密索引**），非叶子节点的值是数据的索引，即：**稀疏索引**（作用：用于快速定位要查找的数据）
* **eg：**
* **数据：**{5，8，9……，98，96，99}，共27个数据
* **稀疏索引：**将数据的索引分成9组，每组3个数据。先将数组切成3份（5，58，65对应每份的起始值），在对每份数据进行3等分（eg：5，10，20），以此类推。

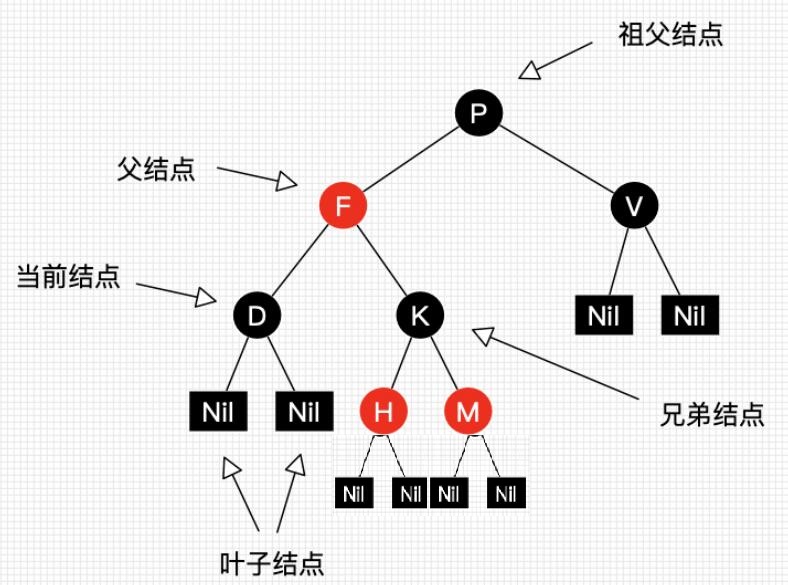


* **B\*树：**
* **B\*树是B+树的改进版，增加了非根、非叶子节点的指向指针。**



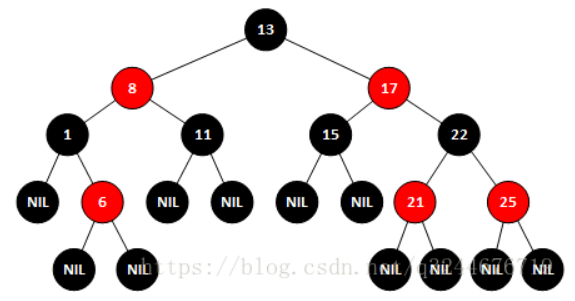


1. **红黑树**
2. 红黑树是一种好友红黑节点、可自平衡的二叉树，具备以下性质：
3. 每个节点不是红色就是黑色；
4. 根节点必须是黑色；
5. 每个叶子节点是黑色的（NIL，叶子节点是null，和普通的二叉树的叶子节点有所区别）
6. 每个红色节点的两个子节点一定是黑色的；
7. **任意节点到每个叶子节点的路径所包含的黑色节点数量相同**。
8. 由5可知：一个节点若有一个黑色子节点，则该节点必定有两个子节点；
9. **红黑树的结构：**



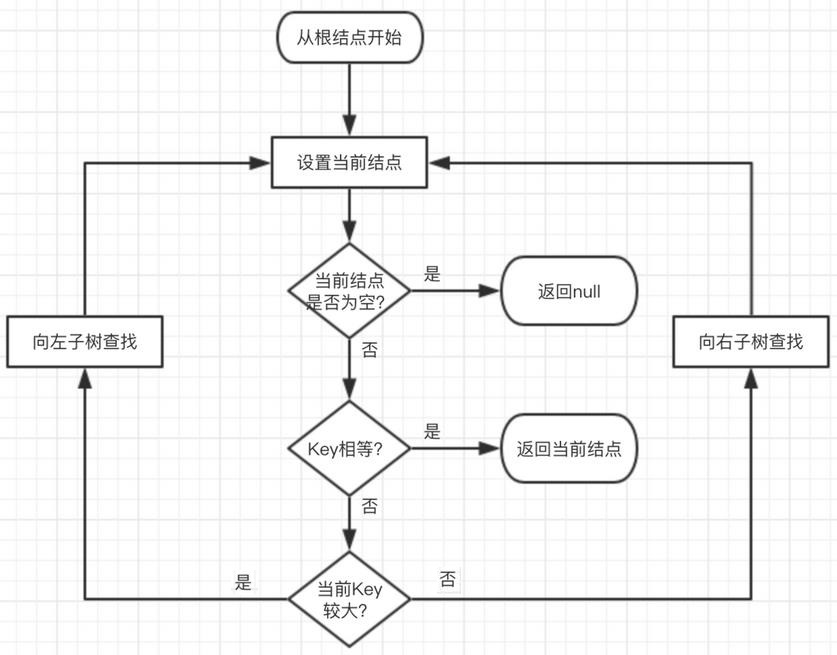
1. **当前节点：**正在遍历的节点；
2. 红黑树并不是完全平衡的（P节点的左右子树层数不相等），但左右子树黑色节点的层数相等，因此称此种红黑树为“**黑色完全平衡**”）；

**示例：**



1. **红黑树的查找**

红黑树的查找不会破坏树的平衡，因此和AVL二叉树的查找方式相同。

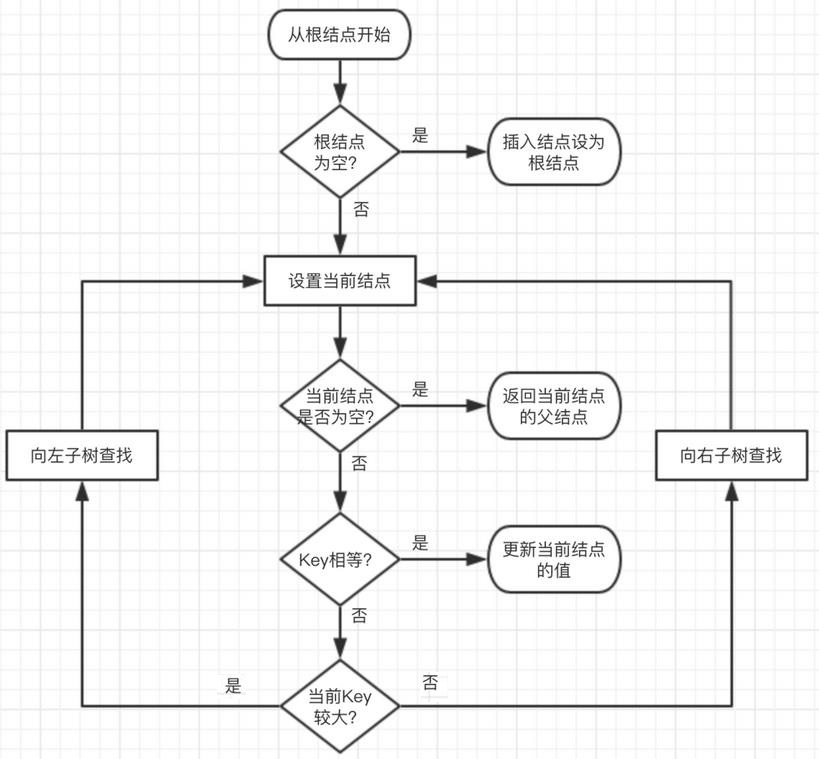


1. **红黑树的插入**

插入操作包含两部分：查找插入位置、插入后进行自平衡。

1. **查找插入位置：**

查找插入位置的父节点的方式和查找操作类似。



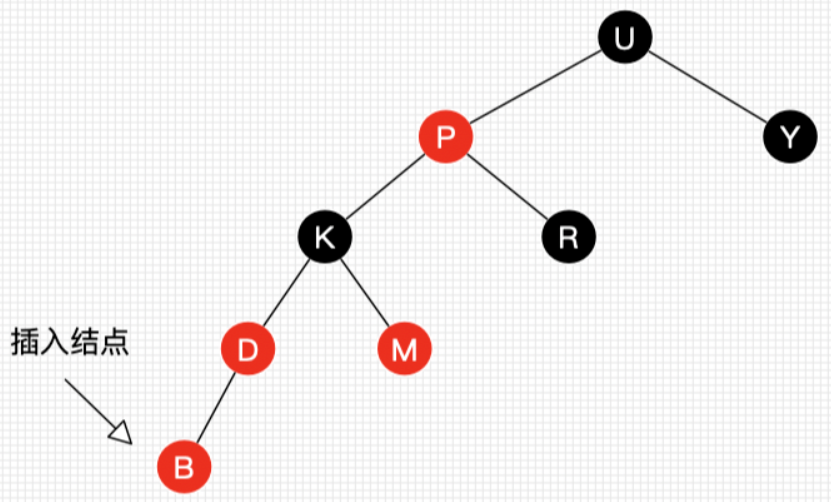
**注意：**

1. **新插入节点是红色的**，因为插入红色节点后，红黑树的黑色完全平衡不会被破坏（如果插入节点是黑色的，就会使插入位置的子树黑色节点 + 1，必须进行自平衡操作）。
2. **所有插入操作都是在叶子节点进行的。**
3. 红黑树的生长是自底向上（普通的二叉树的生长是自顶向下）
4. **插入后，进行自平衡：**

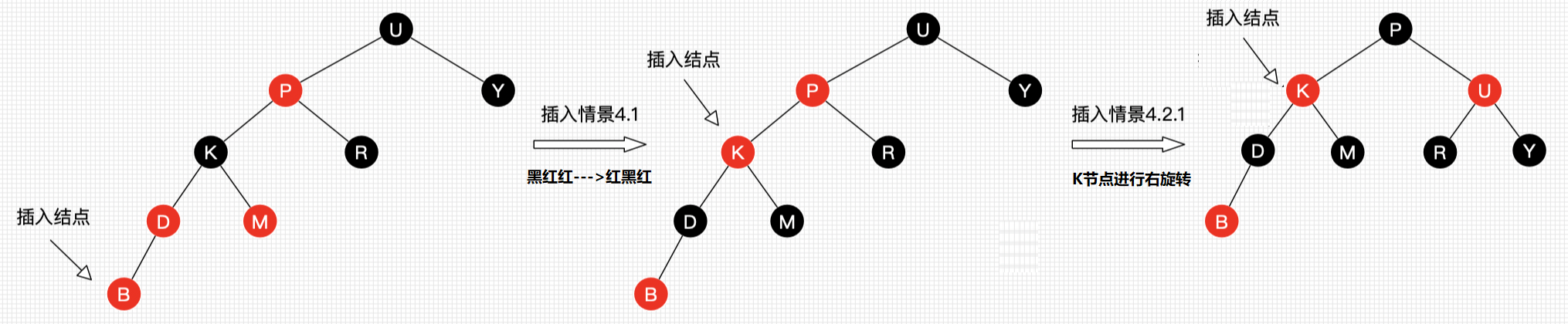
自平衡会涉及到左右旋转，和AVL树的左右旋转一样，只不过需要进行颜色变换、插入节点的变更；



**练习题：绘制插入节点后的自平衡过程**



**答案：**

****

1. **红黑树的删除**

红黑树的删除操作包含两部分：查找目标节点、删除后进行自平衡

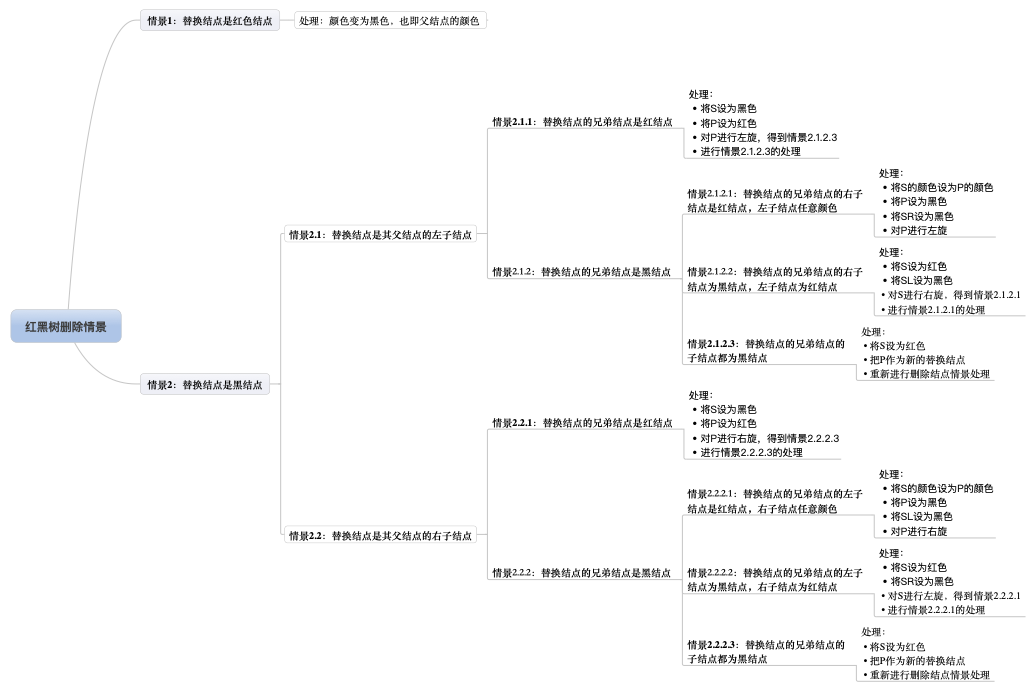
1. **查找操作：**

这与红黑树的查找操作一致。

1. **删除后，进行自平衡：**

删除操作的3中情况：

1. 情景1：若删除结点无子结点，直接删除；
2. 情景2：若删除结点只有一个子结点，用子结点替换删除结点；
3. 情景3：若删除结点有两个子结点，用后继结点（大于删除结点的最小结点）替换删除结点；

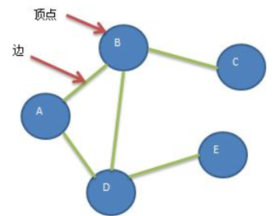


1. **图**

* **思想：**
* **线性表：**局限于只有一个直接前驱、直接后继的关系**（一对一关系）**

**树：**也只有一个直接前驱（父节点） **（一对多、一对一关系）**

**图：**表示一对多的关系



**（无向图）**

* **图的名词：**

**顶点：**上图中的圆圈

**边：**连接顶点的线条

**路径：**权重

**相邻节点：**和当前节点在同一层的节点（不一定相连）

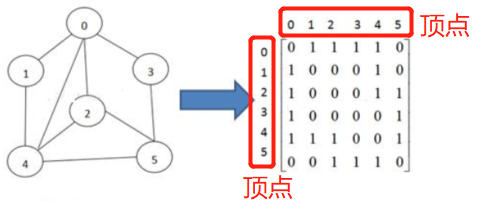
**邻接节点：**和当前节点相连的下一层的节点（一定是相连的）

**有向图：**

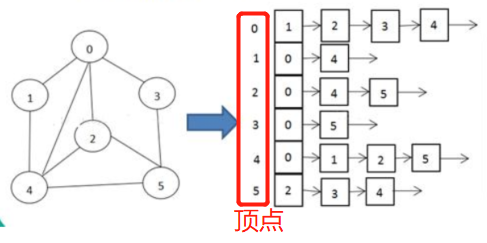
**无向图：**

**带权图：**

* **图的表示方式：（两顶点直接连接：记为1；不直连：记为0）**
* **相邻矩阵：（二维数组）**

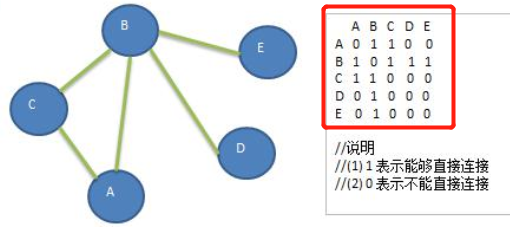


* **邻接表：（一维数组 + 单链表）**



* **遍历图的方法：**
* **深度优先搜索：（可用栈来代替递归）**

深度优先搜索是一个递归过程，从根节点开始，纵向遍历节点，直到遍历到叶子节点，返回上一层继续向下遍历节点…直到所有节点遍历完成。根据邻接矩阵来遍历。**eg：**

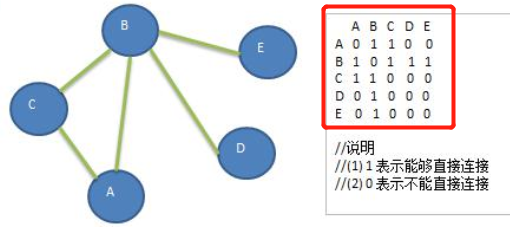


以A为根节点，则深度优先搜索遍历结果：A、B、C、D、E

**深度优先搜索的遍历步骤：（看邻接矩阵，再分析下面的文字）**

* **从第一行开始：**A，其第一个邻接节点为B；
* 跳转到B所在的**第二行：**获取没被访问过的第一个邻接节点C；
* 跳转到C所在的**第三行：**不存在没被访问过的第一个邻接节点；
* 开始递归返回：继续到B第二行，读取D；
* 跳转到**第四行：**D不存在没被访问过的第一个邻接节点；
* 开始递归返回：继续到B第二行，读取E；
* 跳转到**第五行：**E不存在没被访问过的第一个邻接节点；（递归返回，至此结束深度优先搜索）
* **广度优先搜索：（可用队列来代替递归——类似于二叉树层序遍历）**

和分层搜索相似。需要使用队列来保存要进入下层遍历的节点——即：第一个没被访问过的相邻节点。按照列表的先进先出的顺序来访问整棵树。



以A为根节点，则广度优先搜索遍历结果：A、B、C、D、E

**广度优先搜索的步骤：（看邻接矩阵，再分析下面的文字）**

* **第一层：**先从A开始，获取A的第一个邻接节点B（纵向深入下一层），将B存入列表末尾（用于访问B的邻接节点）；
* **第二层：**开始访问和B同层的相邻节点（C），并将C存入列表末尾（用于访问C的邻接节点）；
* **第三层：**开始读取列表最前面的元素（B、C），访问B的邻接节点D、E（都存入列表末尾）；
* **第四层**（不存在）：但是还需要读取、判断D、E是否有下一层。

1. **二分查找：（采用非递归方法）**

* **思想：**

**前提：**查找的数组必须是有序的，才能使用二分查找

二分查找的时间复杂度为O(log2n)，即：找到目标最多需要走log2n步，例如L从100个数中找到30，需要最多查找步数log2100≈7，

因为，26 < 100 < 27

* **代码实现：**

public int search(int[] arr, int findData){  
 int left = 0;  
 int right = arr.length - 1;  
 int mid = (left + right) / 2;  
  
 while(left <= right){  
 if(arr[mid] > findData){  
 right = mid - 1;  
 }  
 else if(arr[mid] < findData){  
 left = mid + 1;  
 }  
 else if(arr[mid] == findData){  
 return mid;  
 }  
 mid = (left + right) / 2;  
 }  
 return -1;  
}

1. **分治算法：**

* **思想：**

将复杂问题分解成简单的问题，将子问题的解合并之后就是原问题的解。**本质：采用递归的思想进行求解**

例如：

* 二分搜索：
* 大整数除法：
* 棋盘覆盖：
* 合并排序：
* 快速排序：
* 归并排序
* 傅里叶变换：
* **汉诺塔：**一共三个塔，初始时，所有的盘都在A塔。将A塔分成两部分，第一部分：除最底下的一个盘之外的盘作为一个整体，第二部分：最底下的一个盘

* **代码：汉诺塔**

public class TowerOfHanoi {  
 int cycleIndex;  
 public void hanoi(int num, char a, char b, char c){  
 cycleIndex++;  
 if(num == 1){  
 System.*out*.println("第 1 个盘从 " + a + "-->" + c);  
 }  
 else{

**//把A塔的num-1个盘，移动到B塔**  
 hanoi(num - 1, a,c,b);

**//把A塔最下面的盘，移到C塔**  
 System.*out*.println("第 " + num + " 个盘从 " + a +"-->" + c);

**//把B塔num-1个盘（所有的盘），移动到C塔**  
 hanoi(num - 1, b,a,c);  
 }  
 }  
  
 public static void main(String[] args) {  
 TowerOfHanoi towerOfHanoi = new TowerOfHanoi();  
 towerOfHanoi.hanoi(3,'A','B','C');  
 System.*out*.println("SYSLE INDEX = " + towerOfHanoi.cycleIndex);  
 }  
}

1. **动态规划**

* **思想：**
* **通过填表得方法来逐步推导，以寻求最优解。**
* **动态规划针对的问题类型：**

1. **计数：**

一般用于求解**有多少种方式**（不是每种方式具体过程）；

1. **求最大、最小值：**

从左上角->右下角路径的最大数字和（二维DP—机器人路径）

1. **存在性：**

博弈游戏（除数博弈、是否有k个数之和为sum等）

* **动态规划的求解步骤：**

**需要一个temp[]数组记录结果，最终的结果在temp[]数组的最后一个元素。**

（1）**确定状态**：

（2）**转移方程**：

（3）**初始条件和边界条件**：

初始条件：一般使数组temp[0] = 0；

边界：什么时候停止、不能获得结果时的处理方式

（4）**计算顺序**：（指：存放结果数组的数据存放顺序）

**选取计算顺序的原则：**当前计算所需的右遍状态数据已经有了。

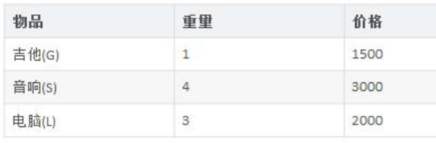
①一维数组索引**（常用）**：从小到大（从左到右）

②一维数组索引：从后往前（从右到左）

③二维数组索引：从左到右 -> 从上到下

* 可以用于解决背包问题：

**问题一：**



背包可负重为N（eg：N=4），求解怎么选取物品，每件物品只能放入一次，使得到得价值最大。

* **0-1背包：**所有的物品只有一件（只能使用一次）

**完全背包：**所有得物品可以无限使用

* **求解上面的问题一：（0-1背包问题，所有物品只用一次）**



* **公式：**

**r:row,行，物品类别**

**c:column，列，不同背包容量：0，1，2，3…最大值**

**v[]:不同物品的价值**

**weight[]：各个物品打重量**

**table[][]：表格**

* **新增物品重量 > 当前背包容量时：**直接使用上一层同列的数据，填入表格

**if(weight[r] > capacity)**

**table[r][c] = table[r-1][c];**

* **新增物品重量 <= 当前背包容量时：**

比较**上一层同列数据** 和 **所加入物品价值 + 上一层目前背包所能放下的最大价值**，选择当中较大的者，填入表格。

（

**为什么只能取dp上一层的组合情况？**

因为，这是0-1背包，所有物品只能用一次。

如果是完全背包（左右物品可以用无限次，那就应该取本行的 + 上一行用一列的）

）

**if(weight[r] <= capacity)**

**table[r][c] =**

**Math.max(table[r-1][c],v[r]+table[r-1][c- weight[r]]);**

* **步骤：**

1. 初始化table表格：必须多出一行、一列，用于求解max中的数据；
2. 按照上面的公式，对表中的单元格进行赋值，**单元格中的数据含义：当前可选用的物品和当前背包容量大小的条件下，该背包的最大价值**

* **代码实现：**

public int optimalSolution(int[] itemWeight, int[] itemValue, int backpackCapacity){  
 int res = 0;  
 int itemCount = itemValue.length; //物品数量  
  
  **//创建表格，存储每个容量下能存放的最大值。**

//行：物品重量， 列：背包容量（0，1，2，3...backpackCapacity）  
 //该表格多出一行、一列（用于计算每个单元格的数据）  
 int[][] maxValueTable = new int[itemValue.length + 1][backpackCapacity + 1];  
  
 **//初始化表格：第一行和第一列都为0**  
 for(int i = 0; i < backpackCapacity; i++){  
 maxValueTable[0][i] = 0;  
 }  
 for(int i = 0; i < itemCount; i++){  
 maxValueTable[i][0] = 0;  
 }  
  
  **//使用动态规划的公式计算 :**

//行：物品重量， 列：背包容量（0，1，2，3...backpackCapacity）  
 for(int rowIndex = 1; rowIndex < itemCount + 1; rowIndex++){  
 for(int columnIndex = 1; columnIndex < backpackCapacity + 1; columnIndex++){  
  **//当前物品超重，无法放入背包，直接使用同列的上一层的值**  
 if(itemWeight[rowIndex - 1] > columnIndex){  
 maxValueTable[rowIndex][columnIndex] = maxValueTable[rowIndex - 1][columnIndex];  
 }  
  **//当前物品可以放入背包：选取最大值放入**

else{  
 maxValueTable[rowIndex][columnIndex] = Math.***max***(

maxValueTable[rowIndex - 1][columnIndex],

itemValue[rowIndex - 1] +  
 maxValueTable[rowIndex - 1][columnIndex - itemWeight[rowIndex - 1]]);  
 return res;  
}

1. **KMP ：用于字符串匹配**

* **思想：**
* 充分利用之前检索、匹配过的数据，加上部分匹配表next[]，进行回溯，通过next数组来查找之前已经匹配过的数据，直接跳过这些数据，进入下一次匹配。以此减少时间，这和暴力破解不太一样。暴力破解每次只能移动一位，直到遍历完所有的原始字符串。这样效率低。

* **KMP检索步骤：**

1. **计算要查找的目标字符串的部分匹配表：**

* 部分匹配表：

字符串的前缀、后缀的最长共有元素的长度

**例如：**

**字符串：ABCDAB**

* **前缀：**A，AB，ABC，ABCD，ABCDA，ABCDA
* **后缀：**BCDAB, CDAB, DAB, AB, B
* **前缀与后缀的共有元素的最大长度 = len(AB) = 2**
* **其部分匹配表为：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 检索词 | A | B | C | D | A | B |
| 部分匹配值 | 0 | 0 | 0 | 0 | 1 | 2 |

1. 检索的字符串为：ABCDABD；

但是目前只匹配了6个字符，D不匹配；



1. 此时需要向后移动下次开始匹配要移动多少个字符：

**使用KMP：**

下次匹配需移动的位数 =**已匹配的字符数–对应的部分匹配值**

= 6 – 2 = 4

（这里移动的是，要查找的目标字符数组“ABCDABD”）

**实际编程中，只需要用for遍历一遍被检索的文档内容，用while来控制目标字符数组的移动位数——实质上，是在寻找、退回到目标字符数组的合适位置（那些已经被匹配的元素将不会在下一轮比较中使用。只有下一轮比较出现不符合时，已匹配的元素才会再次被比较）。**



1. 用for循环遍历文本数据，用while检索需要使用的部分匹配值

* **核心思想：**
* **部分匹配表：**通过遍历**目标字符串**获得对应的部分匹配值
* **使用KMP计算移动的位置：**检索时，遇到不匹配的字符的处理方式，使用部分匹配表，计算需要移动的字符个数。（不再使用暴力破解，每次只移动一位）

* **代码实现：**

**//计算KMP的部分匹配表**  
public int[] next(String source){  
 int[] next = new int[source.length()];  
 next[0] = 0; **//第一个部分匹配值为0**  
 **//index：遍历数组 ，prefix：记录前缀字符和后缀字符相等的个数**  
 for(int index = 1, prefix = 0; index < source.length(); index++){  
  **//计算前缀、后缀共有元素的最大长度**  
 while(prefix > 0 && source.charAt(index) != source.charAt(prefix)){  
 prefix = next[prefix - 1]; **//当前字符不相等，则其部分匹配值 = 前一个**  
 }  
 if(source.charAt(index) == source.charAt(prefix)){  
 prefix++;  
 }  
 next[index] = prefix;  
 }  
 return next;  
}  
  
**//KMP搜索算法**  
public int search(String source, String target, int[] next){  
 for(int i = 0, j = 0; i < source.length(); i++){  
 **//j：记录已经匹配好的字符串长度，当出现不匹配字符时，需要回溯**  
 while(j > 0 && source.charAt(i) != target.charAt(j)){  
 j = next[j-1];  
 }

**//找到完整的匹配字符，返回结果的索引**  
 if(source.charAt(i) == target.charAt(j)){  
 j++;  
 }  
 if(j == target.length()){  
 return i - j + 1;  
 }  
 }  
 return -1;  
}

1. **贪婪算法：（适用于：集合覆盖问题，例如电台匹配地区等）**

* **思想：**
* 每一次遍历数据后，都选取最佳的数据，当满足退出条件后，所得到的数据是接近最佳结果（可能存在多个不同结果），虽然得到的结果不一定的最好的，但一定是接近最好的。
* **贪婪算法和暴力破解的对比：**

暴力破解较为耗时，尤其是当可供选择的结果很多时，多个数据进行组合的有很多种类，这样的计算太大。贪心算法可以在减少计算的同时，又保证得到的数据相对较好。

* **代码实现：**

/\*\*  
 \* @param **broadcasts：需要进行检索的电台集合，每个电台都覆盖特定的区域**  
 \* @param **allAreas：所有的地区集合**  
 \* @return： **返回最佳的电台名称：B2, B3, B4, B5,**  
 \*/  
public ArrayList<String> fit(HashMap<String, HashSet<String>> broadcasts, HashSet<String> allAreas){  
  **//存放未覆盖地区，和当前电台覆盖地区的交集**  
 HashSet<String> intersection = new HashSet<String>();  
 **//当前遍历的电台能够覆盖的区域**  
 HashSet<String> areas;  
 **//覆盖地区最大的电台：键值--索引  
 String maxB = null;**  
 **//存放结果：电台的键值**  
 ArrayList<String> res = new ArrayList<String>();  
 **//当所有地区都被覆盖后，退出查找**  
 while(allAreas.size() != 0){  
 maxB = null;  
  **//1、遍历所有电台，覆盖地区最大的电台的键值** for(String B : broadcasts.keySet()){  
 intersection.clear();  
 areas = broadcasts.get(B);  
 intersection.addAll(areas);  
  **//2、求出该电台和当前剩余的为覆盖地区的交集 （核心）**  
 intersection.retainAll(allAreas);  
  **//3、比较当前电台覆盖的区域 和 当前存在的最大覆盖区域的大小 （核心）**  
 if(intersection.size() > 0 && (maxB == null || intersection.size() > broadcasts.get(maxB).size())){  
 maxB = B;  
 }  
 }

**//4、遍历一次之后，存放结果，直到所有地区被完全覆盖**  
 if(maxB != null){

**//存储单次遍历的最优结果**  
 res.add(maxB);  
  **//5、删除已经覆盖区域 （核心）**  
 allAreas.removeAll(broadcasts.get(maxB));  
 }  
 }  
 return res;  
}

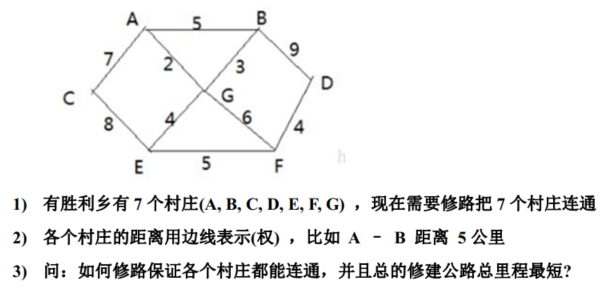
1. **Prim算法：**

**(求解无向加权连通图的最小生成树---联通所有顶点的最短路径)**

* **思想：**
* 本质上，就是最小生成树问题（Minimun cost spanning tree，MST），给定一个无向带权连通图，要**生成一颗所有边的权值总和最小的树**。
* **特点：**N个顶点的树，一共有N-1条边。

**无论顶点如何改变，所生成的最小生成树都不会改变**。

* **应用场景：**



* **实现思路：**

1. 先利用现有顶点数据、邻接矩阵创建图
2. 确定开始遍历的起始顶点
3. 选择该顶点的邻接节点中权重最小的一个顶点，作为一条路径，并标记该顶点已被访问
4. 开始下一轮搜索，遍历这些已被访问的顶点，选择当中权重最小的邻接节点，构成下一条路径
5. 重复3、4步骤，直到所有的顶点都被访问完为止，所构成的树即为：最下生成树。

* **代码实现：**

**/\*\*  
 \* 获取最小生成树  
 \* @param graph：传入原有图，用于构建MST  
 \* @param startVertex：起始顶点  
 \* @return  
 \*/**  
public HashMap<Integer, HashSet<Character>> MST(Graph graph, int startVertex){  
 System.*out*.println("====== MST ======");  
 **//标记被访问过的顶点**  
 int[] isVisited = new int[graph.getVertexNum()];

**//初始时，该起点被访问过**  
 isVisited[startVertex] = 1;   
  
 **//记录两个顶点的索引**  
 int preIndex = -1; //已被访问的顶点索引  
 int postIndex = -1; //没被访问的顶点索引  
  
 **//初始化最短路径数值**  
 int minWeight = 100; //该值为：选择邻接矩阵中最大的数据---表示这两个顶点不相连

**//核心部分**  
  **//开始寻找最短路径：寻找到 graph.getVertexNum() -1 条路径，才算完成最短路径规划**  
 for(int i = 1; i < graph.getVertexNum(); i++){   
 for(int x = 0; x < graph.getVertexNum() ; x++){ **//遍历已被访问的顶点** for(int y = 0; y < graph.getVertexNum(); y++){  **//遍历未被访问的顶点** **//找到一个已被访问顶点的未被访问过的邻接节点，满足权重最小(路径最短)** if(isVisited[x] == 1 && isVisited[y] == 0 && graph.getAdjacentMatrix(x, y) < minWeight){  
 preIndex = x;  
 postIndex = y;  
 }  
 }  
 }

**//打印结果** System.*out*.println(graph.getVertexes(preIndex)+ "--" + graph.getVertexes(postIndex) + "--" + graph.getAdjacentMatrix(preIndex,postIndex));  
  **//标记该节点已被访问**  
 isVisited[postIndex] = 1;  
 minWeight = 100;  
 }  
 return res;  
}

1. **Kruskal算法： （求解加权连通图的最小生成树）**

* **思想：**
* 将所有的边，按照权重，从小到大排序，每次选择一条权重最小的边，并且不和现有最小生成树构成回路。添加到现有最小生成树，直到遍历完所有的边为止。此时构成的树就是Kruskal算法的最小生成树。

* **kruskal和prim对比：**
* **Prim算法：遍历的是现有最小生成树内的顶点**，寻找这些顶点的邻接节点中权重最小的进行添加。需要遍历完整个图的所有顶点。该MST边的数量（edgeNum - 1）
* **Kruskal算法：遍历的是整个图的边（已经按照权重，从小到大排序）**，寻找那些权重小并且没有和现有最小生成树构成回路的的边，添加进最小生成树。该MST边的数量（edgeNum - 1）
* **（最重要的地方）如何判断一条边和当前MST是否构成回路：**

1. 首先，规定没被添加过的顶点，该顶点的终点就是自己。一条边只有两个顶点。
2. 使用一维数组ends[]，来记录当前MST所有顶点的终点信息：

**数组元素索引：**表示该顶点的索引

**数组元素值：**表示该顶点的终点索引

1. **（核心）**在数组ends[]中，检索要添加边的两个顶点的末尾顶点索引所对应的值是否相等？

**相等：**说明构成回路

**不相等：**未构成回路

* **实现步骤：**

1. 创建边的数据结构：一条边有起始顶点、末尾顶点、权重等3个参数；

一幅图：由顶点和带权边构成——无向加权图；

1. 将所有的边，用上面边的数据结构的一维数组来存储，并且按照权重大小，排好序；
2. 遍历所有边，现选择权重小的且与当前MST不构成回路的边，添加到MST中。
3. 重复步骤3，遍历完所有的边，构成的树，即为MST（最小生成树）。

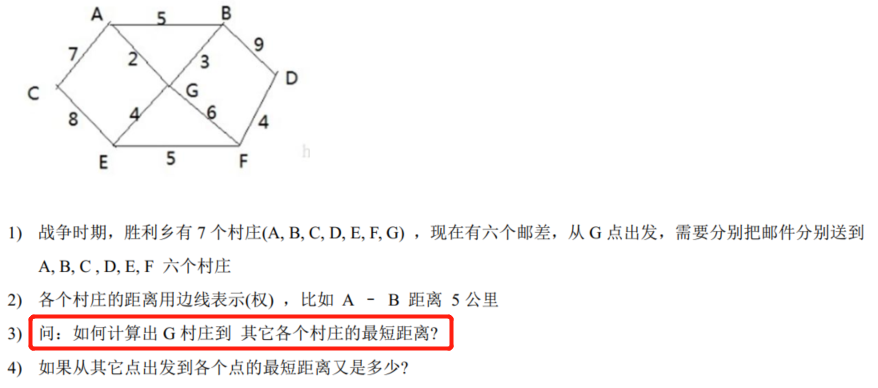
1. **Dijkstra算法（迪科斯切）**

**（求解给定加权图中，求解一个顶点到其他剩余顶点的最短路径，使用广度优先搜索的思想）**

* **思想：**
* 使用**广度优先搜索**思想，通过规定一个顶点作为起始点，按照广度优先搜索的方式，逐层遍历各个顶点，选择每个顶点到起始顶点的最短路径，并不断叠加这个最短路径。当遍历完所有的顶点之后，就可得到每个顶点到第一个顶点的最短路径（获得一个具体数值）。

**本质：**给定一个顶点，搜寻整幅图其余顶点到该顶点的最短路径

* **具体应用：**



1. **Floyd算法：**

**（求解给定加权图中，每个顶点到其余顶点的最短路径，和Dijkstra用处一样，但不同）**

* **思想：**
* 采用中间顶点，来计算起始顶点到末尾顶点的距离：

**len** = distance[起始顶点到中间顶点]+ distance[中间顶点到末尾顶点]

* 即：Floyd涉及到3个顶点：起始顶点、中间顶点、末尾顶点
* **Floyd算法核心：**使用3个for来循环遍历所有的顶点，计算每个顶点到剩余顶点的最短距离。

因此Floyd的时间复杂度较大：n的立方阶

* **实现方式：**使用两张二维数组来保存数据，循环遍历所有顶点，来不断更新这两张表的数据，最终的数据即为最短距离。

**1、前驱表：**起始顶点对应的前驱顶点关系表

**2、距离表：**每个顶点到其他顶点的最短距离表

**for循环的遍历方式：**中间顶点 -> 起始顶点 -> 末尾顶点

**//1、中间顶点**  
for(int mid = 0; mid < dis.length ; mid++){  
 **//2、起始顶点**  
 for(int start = 0; start < dis.length; start++){  
 **//3、末尾顶点**  for(int end = 0; end < dis.length; end++){

**//计算距离，更新前驱表、距离表**

}

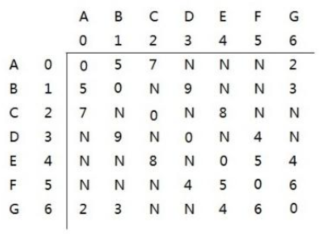
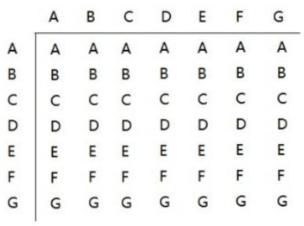
}

}

* **eg：**

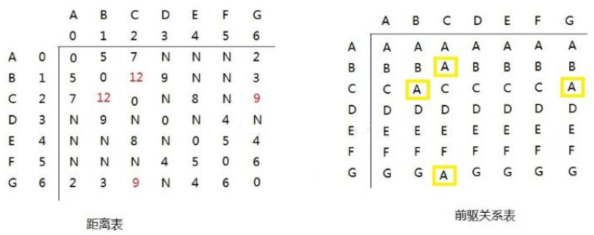


（1）初始时，所有的前驱表，其前驱顶点为自己

距离表 前驱表

（2）假设，选择A作为中间顶点，开始遍历，第一次更新所有顶点的距离表、前驱表，结果如下：



（3）执行完3个for循环，就可以得到结果。（将A-G分别作为中间节点，遍历计算得到最短距离）。

* **代码实现：**

**/\*\*  
 \* 使用Floyd算法求解最短路径  
 \* 复杂度：n立方阶  
 \*/**  
public void floyd(){  
 int len = 0;  
  
 **/\*\*\*\*\*\* 开始计算每个顶点，到其他剩余顶点的最短路径 \*\*\*\*\*\*/**  
  **//1、中间顶点**  
 for(int mid = 0; mid < dis.length ; mid++){  
 **//2、起始顶点**  
 for(int start = 0; start < dis.length; start++){  
 **//3、末尾顶点**  
 for(int end = 0; end < dis.length; end++){  
  **//起始顶点到中介顶点的距离 + 中间顶点到末尾顶点的距离**  
 len = dis[start][mid] + dis[mid][end];  
 if(len < dis[start][end]){  
  **//更新最短距离**  
 dis[start][end] = len;  
  **//更新前驱节点**  
 pre[start][end] = pre[mid][end];  
 }  
 }  
 }  
 }  
}

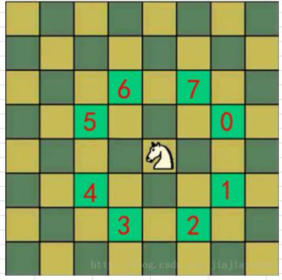
1. **骑士周游（马踏棋盘）**

* **思想：**
* **使用深度优先搜索**，遍历每个路径，直到得到最终结果
* **解决思路：**

1. 8\*8的棋盘相当于是二维数组；
2. 判断当前‘马’所在位置，下一步能走到哪些位置，用ArrayList存储，每往下走一步step计步+1；
3. 若，当前这步已经不能继续往下走，则开始回溯，并且将这步走过的数据清零（防止对下面的回溯产生影响）；
4. **‘马’可以遍历所有格子的判断条件**：深度优先搜索后，步数step=8\*8=64 ，也就说明已经走过所有的格子。

* **注意：**

在棋盘中，马的下一步，最多有8种选择



* **贪心算法的应用：**由于这样的深度优先搜索，耗时巨大，可以结合贪心算法，减少时间。具体操作：

**（贪心算法的精髓：**

1. 每一步都是获取当前最优值（有点像dp）
2. **先排序**，后使用贪心算法获取最优值；

）

在获得当前‘马’的下一步可选择位置时，将这些步骤的下一步的数量进行非递减排序，在从头开始遍历这些步骤。即：每次都选择步骤最少的下一步遍历。

（**非递减排序：**即排序后的数组存在重复元素：1，**2，2，2**，3，5）

（**非递增排序：**即排序后的数组存在重复元素：**9，9**，7，**5，5**，1）