东南大学·面试&算法讲座

July

东南大学自动化研会

2014-7-16, 晚6:30~9:00

本次讲座大纲

- 笔试面试考什么
- 解决笔试面试题的常用算法
- 常用算法的时间复杂度
  - 包括各类排序算法
- O(N)时间复杂度内能解决的问题
  - 包括KMP，最长回文子串Manacher算法
- 如何学习算法
  - 相互串联
    - 以Trie树、后缀树，贪心、动态规划为例
  - 简单入手，追本溯源
    - 二叉树、红黑树、2-3-4树、B树为例
- 海量数据处理面试题
  - 十种解决之道

笔试面试考什么

笔试偏基础

- 语言基础

```
int hope;
int* hope;
double(*p) [3];
void (*func) ();
```

- 操作系统
  - 线程与进程的区别
  - 产生死锁的条件
    - 如何规避死锁
  - C++内存分配
    - 堆、栈、自由存储区、全局/静态存储区，常量存储区

- 网络协议
  - TCP建立连接的三次握手

- 数据库

- 概率论与数理统计
  - 推荐《数理统计学简史》

基础不够 补基础

- 语言
  - **C** ：
    - 《C 和指针》
    - 《征服C 指针》
  - **C++**：
    - 《C++ Primer》
    - 《STL 源码剖析》
    - 《Effective C++》
    - 《深度探索C++ 对象模型》

面试偏算法

- 数据结构上的增删改查
  - 查找、遍历、排序
- 算法
  - 分治、递归、回溯
  - 贪心、动态规划
- 海量数据处理

一般面试常考的是：①数据结构：字符串、链表、数组、堆、哈希表、树（Trie树、后缀树、红黑树、B树、R树）、图（遍历：BFS、DFS、Dijkstra）；②基于各个数据结构的查找（二分、二叉树）、排序、遍历；③算法：排列组合概率、分治递归回溯、贪心算法、动态规划、海量数据，外加字符串匹配和资源调优。

3月30日 23:02 来自微博 weibo.com 阅读(8.0万) 推广 | (63) | 转发(229) | 收藏 | 评论(38)

基于各个数据结构上的增删改查

- 字符串
  - 字符串库函数的编写，例如atoi 等
  - 字符串查找、翻转、匹配
- 数组
  - 查找（如二分查找、杨氏矩阵查找）
- 链表
  - 翻转、遍历、查找、删除、合并
- Hash表
  - 查找
- 树
  - 遍历（前序、中序、后序）
  - set、map
  - 高级树的查找（红黑树、B树、R树）
- 图
  - 遍历
  - 查找（DFS、BFS）
  - 最短路径算法

知道了考什么，怎么破

笔试面试常用算法

- 穷举（递归回溯）——“万能的”
  - 求n个数的全排列 & 8皇后（N皇后问题）
- 分治
  - 分而治之，然后归并
- 递归回溯
  - DFS
- 空间换时间
  - hashtable
- 巧用数据结构
  - 堆
- 能排序，考虑排序
  - 前后两个指针往中间扫
  - 若已经排好序，想想有无必要二分
- 不能排序
  - 贪心
    - 最小生成树 Prim, Krusal
    - 最短路 dijkstra
  - 动态规划
    - 如 01背包问题，每一步都在决策
- 细节处理
  - 注意边界条件

各类算法的时间复杂度

O(1) 到 O(nlogn)

- O(1)
  - 基本运算， + ， - ， * ， / ，%，寻址
  - **Hash表的期望复杂度**
- O(logn)
  - 二分查找
- $O(n^{1/2})$
  - 枚举约数
- O(n)
  - 线性查找
  - 建立**堆**
- O(nlogn)
  - 归并排序
  - **快速排序的期望复杂度**
  - 基于比较排序的算法下界

$O(n^2)$ 到 $O(n^n)$

- $O(n^2)$
  - 集合里枚举所有二元组、朴素最近点对
- $O(n^3)$
  - 集合里枚举三元组、Floyd最短路、普通矩阵乘法
- $O(2^n)$
  - 枚举全部的子集
- $O(2^n n)$
  - TSP的动态规划算法
- $O(n!)$
  - 枚举全排列
- $O(n^n)$
  - 枚举[1..n]的n维数组的全部元素……
- 总结
  - $O(1) < O(\log n) < O(n^{1/2}) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(2^n n) < O(n!) < O(n^n)$

各种排序算法的时间复杂度

排序方法	最好时间	平均时间	最坏时间	辅助空间	稳定性
直接插入	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
二分插入	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
希 尔		$O(n^{1.25})$		O(1)	不稳定
冒 泡	O(n)	$O(n^2)$	$O(n^2)$	O(1)	稳定
快 速	O(nlgn)	O(nlgn)	$O(n^2)$	O(lgn)	不稳定
直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
堆	O(nlgn)	O(nlgn)	O(nlgn)		不稳定
归 并	O(nlgn)	O(nlgn)	O(nlgn)	O(n)	稳定
基 数	O(d(r+n))	O(d(r+n))	O(d(r+n))	O(rd+n)	稳定

O(N)的时间复杂度能解决什么问题？

O(N)时间内能解决的问题

- 字符串
  - **字符串循环位移**
  - **最长回文子串**

- 数组
  - **寻找最小的K个数**
  - 2-sum
  - 最大连续子数组和
  - **快排的partition**
  - **奇偶排序**
  - **荷兰国旗问题**
  - 完美洗牌问题
  - 最大面积直方图
  - 最大连续乘积子数组

- 查找排序
  - 杨氏矩阵查找
  - **出现次数超过一半的数字**

- 建立堆
- 计数排序
- 二叉查找树的前中后序遍历

- KMP
- Manacher

字符串翻转

- 翻转
  - 定义字符串左旋转操作：把字符串前面的若干个字符移动到字符串尾部,如把字符串 abcdef 左旋转 3 位得到字符串 defabc。

  要求时间复杂度为 O(n)，空间复杂度为 O(1)。

  - 暴力移位

  - 三步翻转（字符串 abcdef -> defabc）
    - X：abc，Y：def；
    - X->X^T，得：abc->cba；Y->Y^T，得：def->fed
    - X^TY^T，得到：cbafed->defabc，即(X^TY^T)^T=YX

寻找最小的k个数

- 输入n个整数，输出其中最小的k个。
  - 例如输入1，2，3，4，5，6，7和8这8个数字，则最小的4个数字为1，2，3和4。

最大子数组最大和

- 题目描述
  - 输入一个整形数组，数组里有正数也有负数。数组中连续的一个或多个整数组成一个子数组，每个子数组都有一个和。求所有子数组的和的最大值。要求时间复杂度为O(n)。
    - i)一维：输入的数组为1, -2, 3, 10, -4, 7, 2, -5，和最大的子数组为3, 10, -4, 7, 2，因此输出为该子数组的和18。
      - 暴力循环
      - O（N）遍历

		1	-2	**3**	**10**	**-4**	**7**	**2**	-5
b	:	0	1	-1	3	13	9	16	18	13
sum	:	0	1	1	3	13	13	16	18	18

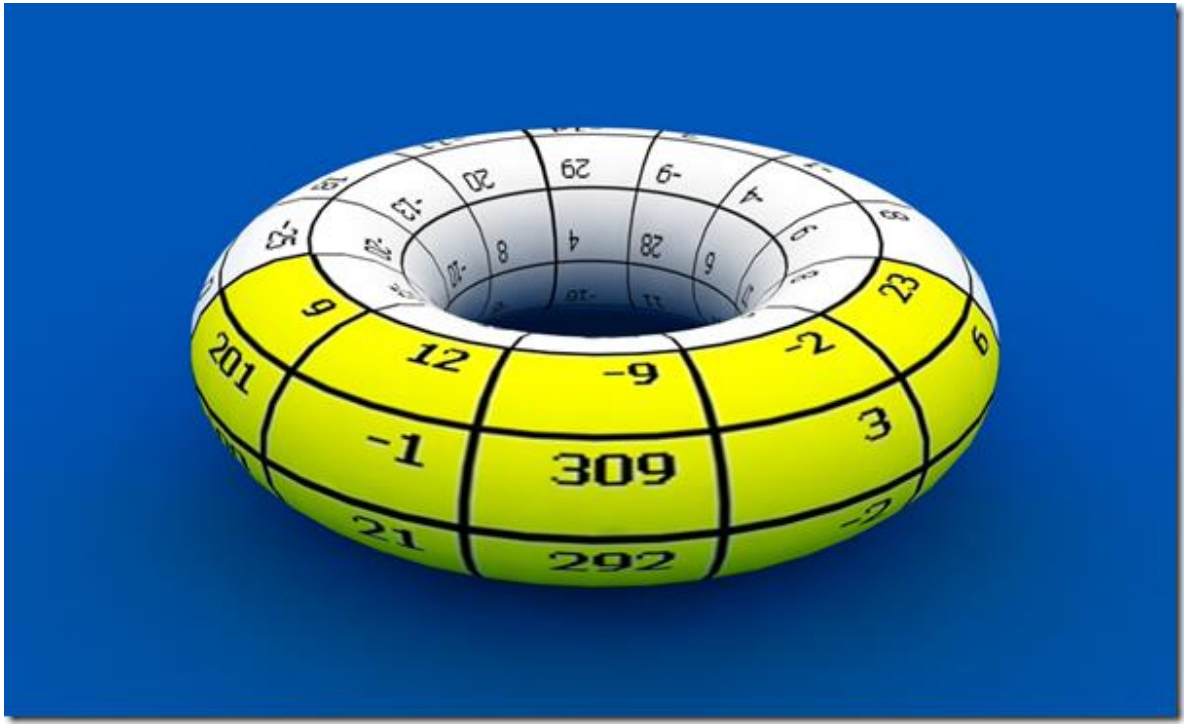
- ii) 二维

	A	B	C	D	E	F
1	3					
2	6					
3	5	6	-3	8	-9	2
4	1	-12	20	0	-3	-5
5	-9	-7	-3	6	7	-1

- iii)子数组” 并不只是一个二维数组或矩形，而是联通的元素(上下或左右相邻即视为联通)

	A	B	C	D	E	F
1	3					
2	6					
3	5	6	-3	8	-9	2
4	1	-12	20	0	-3	-5
5	-9	-7	-3	6	7	-1

- iv)如果是个轮胎呢？



寻找和为定值的两个数

- 输入一个数组和一个数字，在数组中查找两个数，使得它们的和正好是输入的那个数字。
  - 要求时间复杂度是O(N)。如果有多对数字的和等于输入的数字，输出任意一对即可。
  - 例如输入数组1、2、4、7、11、15和数字15。由于4+11=15，因此输出4和11。

- 解答：
  - 百试不厌：暴力穷举
  - 如果无序，先排序，排完序后，i j前后两个指针往中间扫

快速排序算法的一个实现

- 一前一后两指针同往后扫
  - j 指针从第一个元素扫到倒数第二个，遇到比主元小，(i+1)与 j 所指元素互换

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p - 1
3   for j ← p to r - 1
4         do if A[j] ≤ x
5               then i ← i + 1
6                    exchange A[i] <-> A[j]
7  exchange A[i + 1] <-> A[r]
8  return i + 1

QUICKSORT(A, p, r)
1 if p < r
2    then q ← PARTITION(A, p, r)   //关键
3         QUICKSORT(A, p, q - 1)
4         QUICKSORT(A, q + 1, r)
```

快速排序实现之步骤一

```
int partition(int data[],int lo,int hi)
{
 int key=data[hi];  //以最后一个元素，data[hi]为主元
 int i=lo-1;
 for(int j=lo;j<hi;j++)   ///注，j从p指向的是r-1，不是r。
 {
  if(data[j]<=key)
  {
   i=i+1;
   swap(&data[i],&data[j]);
  }
 }
 swap(&data[i+1],&data[hi]);   //不能改为swap(&data[i+1],&key)
 return i+1;
}
```

快速排序实现之步骤二

```
void QuickSort(int data[], int lo, int hi)
{
    if (lo<hi)
    {
        int k = partition(data, lo, hi);
        QuickSort(data, lo, k-1);
        QuickSort(data, k+1, hi);
    }
}
```

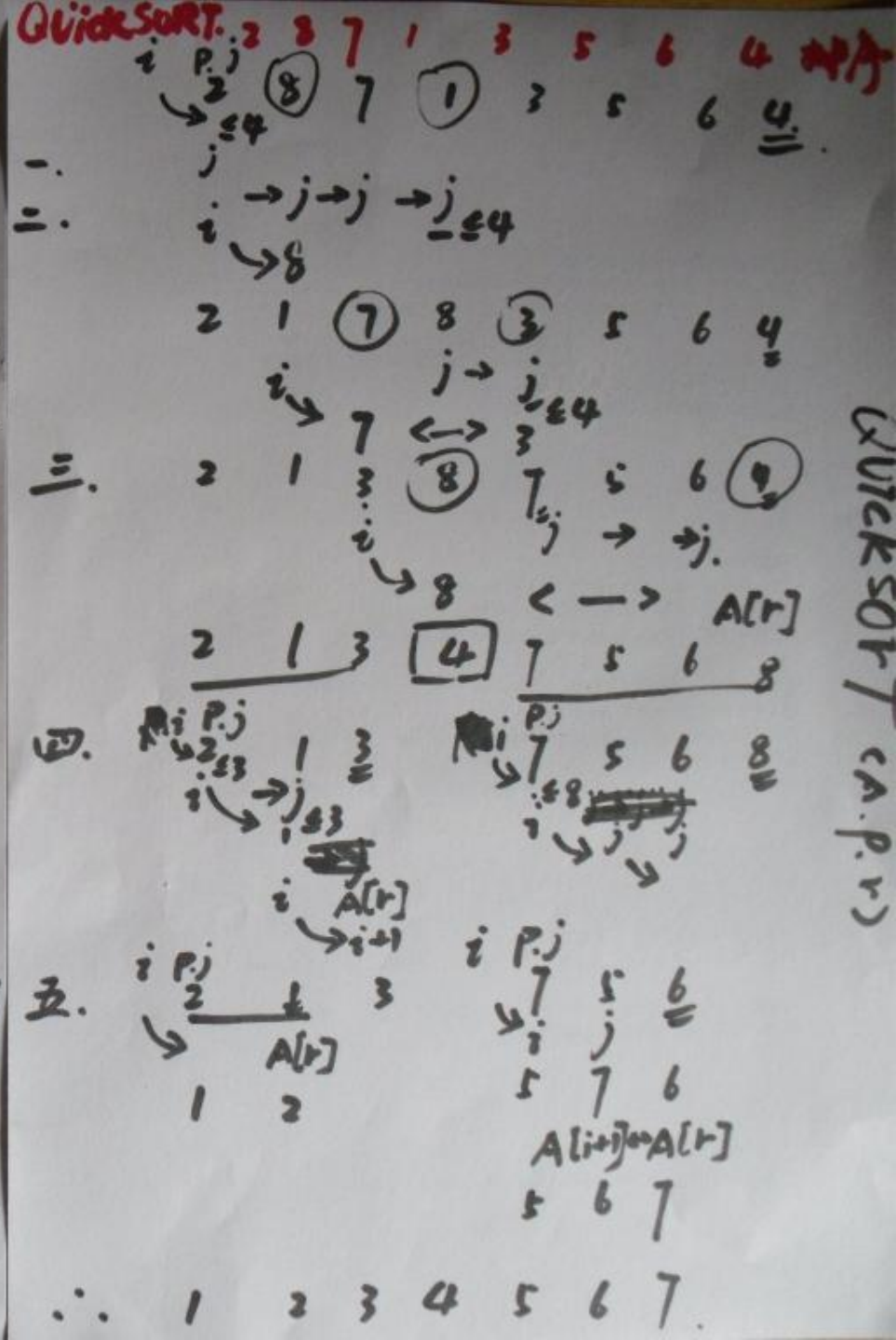
快速排序

```
QUICKSORT(A,p,r)
1. if p<r
2.     then q ← PARTITION(A,p,r)
3.          QUICKSORT(A,p,q-1)
4.          QUICKSORT(A,q+1,r).

PARTITION(A,p,r)
1. x ← A[r]
2. i ← p-1
3. for j ← p to r-1
4.     do if A[j]<=x
5.          then i ← i+1
6.               exchange A[i]↔A[j]
7. exchange A[i+1]↔A[r]
8. return i+1
```

July. 二零--年-月四日

QUICKSORT(A,p,r)



奇偶排序

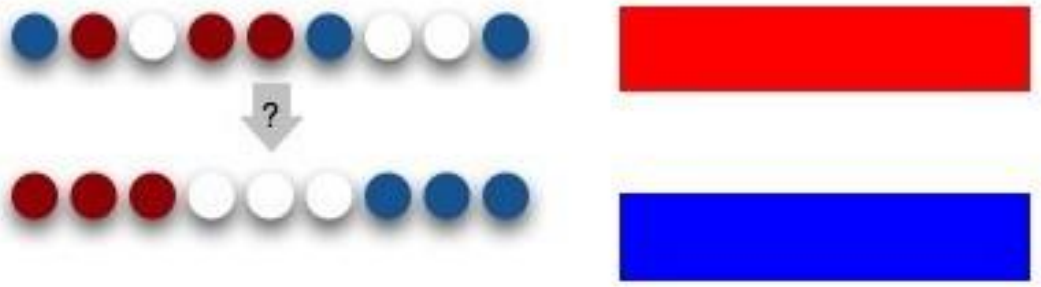
- 题目描述
  - 输入一个整数数组，调整数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。要求时间复杂度为O(n)。

- 解法：
  - 暴力移位
  - 维护两个指针，步骤如下：
    1. 一个指针指向数组的第一个数字，向后移动；
    2. 一个指针指向最后一个数字，向前移动；
    3. 如果第一个指针指向的数字是偶数且第二个指针指向的数字是奇数，我们就交换这两个数字。

- 变形：
  - 给定一个整数数组，其元素分为正数、负数和0，现请把正数放左边，负数放右边，0在中间
    - 荷兰国旗问题

荷兰国旗问题

- 题目描述
  - 相同的球放到一起，让你按顺序输出红白蓝三种颜色的球，可以用012来表示，要求只能扫描一次数组
    - 将乱序的红白蓝三色小球排列成有序的红白蓝三色的同颜色在一起的小球组。



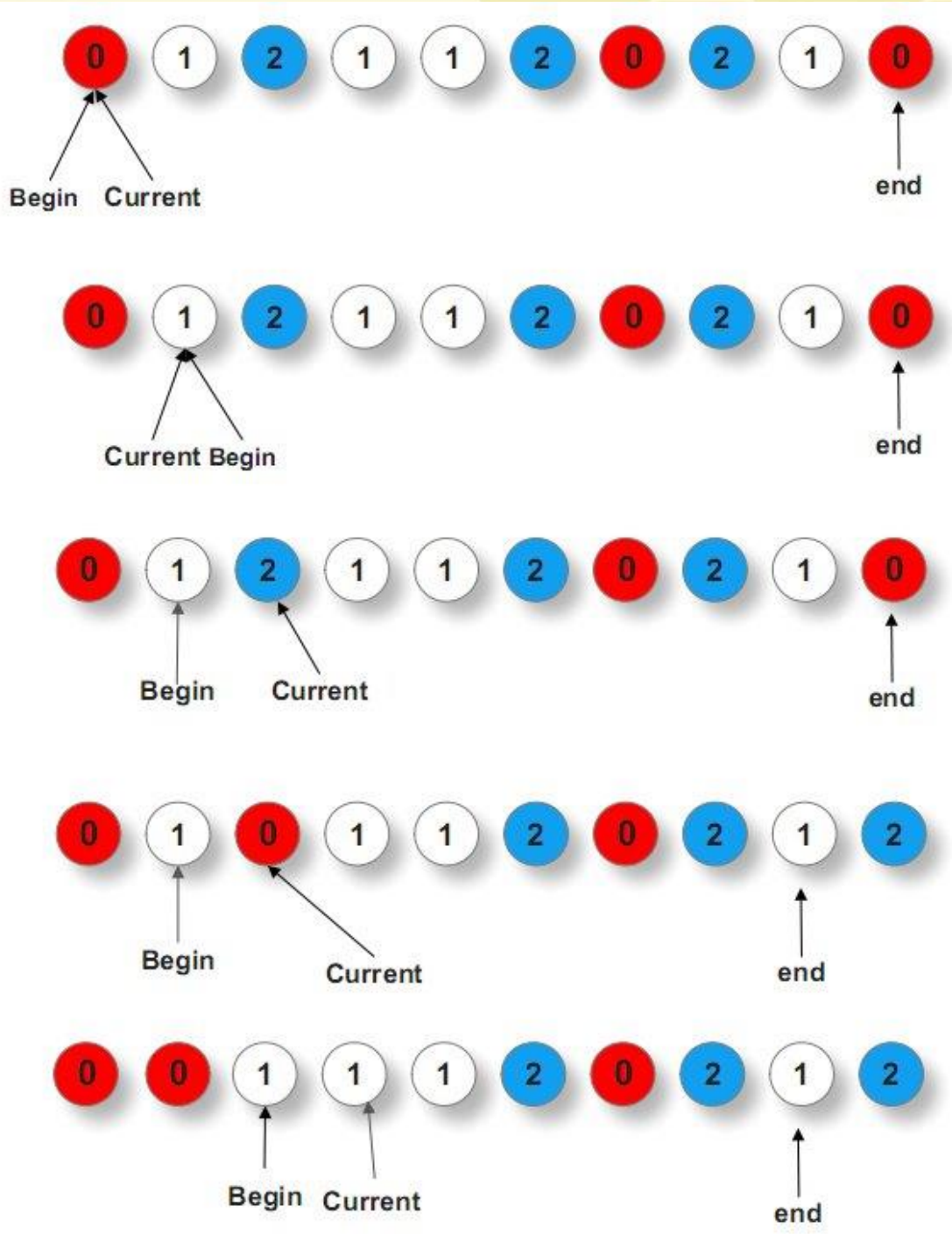
荷兰国旗问题思路

- 类似快排中partition过程，用三个指针，一前begin，一中current，一后end，current遍历 整个数组序列：
  1. current指0，与begin交换，而后current++，begin++；
  2. current指1，不做任何交换（即球不动），而后current++；
  3. current指2，与end交换，而后，current指针不动，end--。

- 快排中的partition过程？

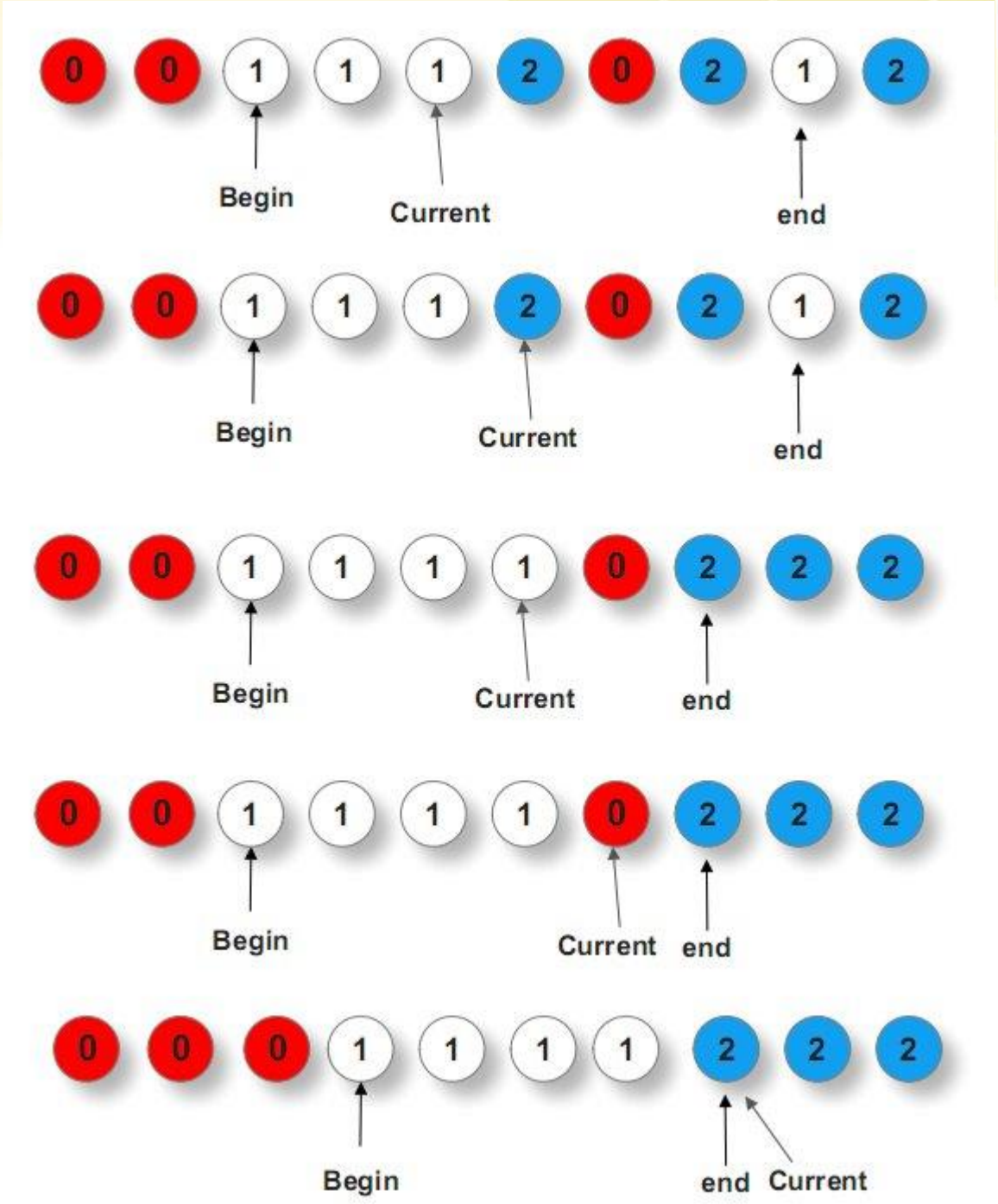
解决步骤1

- current指0，与begin交换
  - 而后current++，begin++
- current指1，不做任何交换
  - 而后current++
- current指2，与end交换
  - 而后，current不动，end--



解决步骤2

- current指0，与begin交换
  - 而后current++，begin++
- current指1，不做任何交换
  - 而后current++
- current指2，与end交换
  - 而后current不动，end--



再进一步

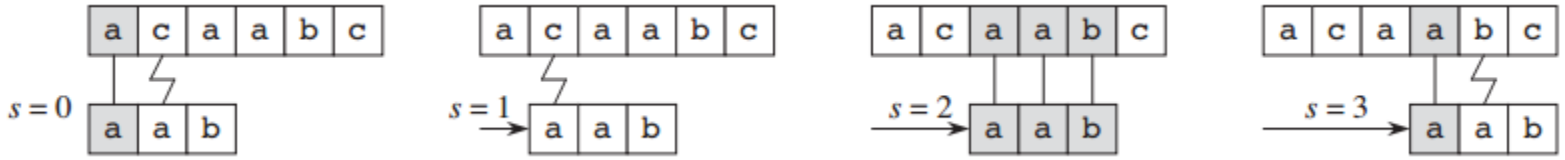
- 题目描述
  - 给定一个未排序的整数数组，有正数和负数，重新排列使得负数在正数前面，并且要求不改变原来的正负数之间的相对顺序
    - 1 7 -5 9 -12 15
    - -5 -12 1 7 9 15

如何降低时间复杂度

- 减少不必要的操作
  - 比如寻找出现次数超过一半的数字
    - 数组排完序后直接输出第N/2处的那个数，不必再统计每个数字的出现次数
- 空间换时间
  - 比如借助Hash表达到快速映射的目的
- 根据问题本身的特性使用对应的技巧
  - 比如KMP算法中，对模式串的预处理，通过实现求解出一个next数组后，后续匹配失配时直接查next数组得到下一次匹配的位置。
- 巧妙的算法
  - 比如最长回文子串Manacher算法

字符串匹配·KMP

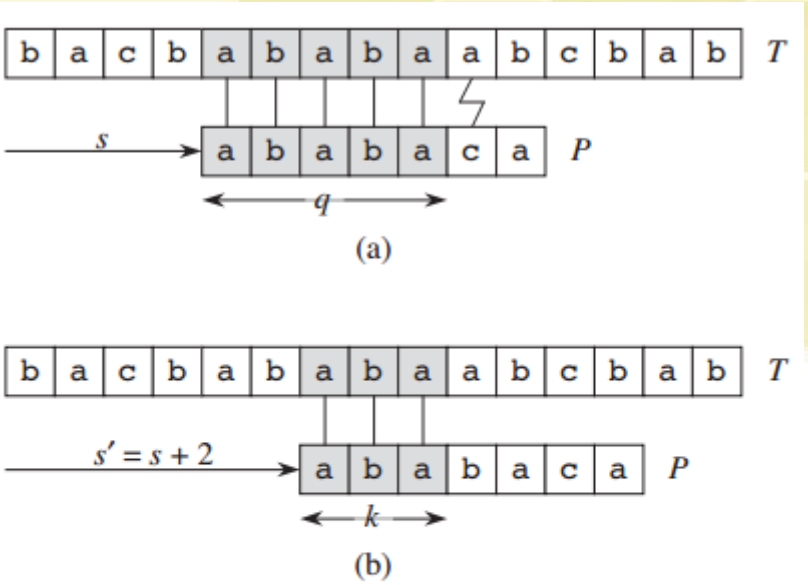
- 朴素算法
  - O（ (n-m+1)m ）

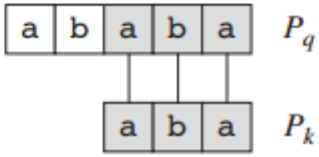


- 有限自动机算法
  - O（n）
- KMP
  - O（n）

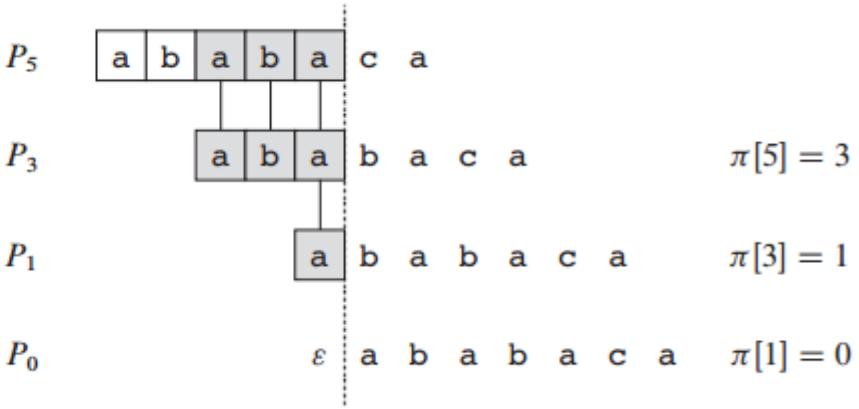
Knuth - Morris - Patt

- KMP
  - O (n)





i	1	2	3	4	5	6	7
$P[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1



求模式串的next

模式串	a	b	a	a	b	c	a	b	a
next	-1	0	0	1	1	2	0	1	2

Next的描述性说法

- 对于Aj =a0 a1 ...aj-1 aj，查找字符串Aj的最大相等k前缀和k后缀。
- 即：查找满足条件的最大的k，使得
  - a0 a1 ...ak-1 ak = aj-k aj-k+1...aj-1 aj
  - 则对于pattern的前j+1序列字符
    - 若pattern[k+1]==pattern[j+1]
      - next[j+1]=next(j)+1
    - 若pattern[k+1]≠pattern[j+1]
      - 如果此时 pattern[next[k] + 1] == pattern[j + 1]，则next[j + 1] = next[k] + 1
      - 否则重复此过程。

求字符串的最长回文子串

- 回文子串的定义：
  - 给定字符串str，若s同时满足以下条件：
    - s是str的子串
    - s是回文串
  - 则，s是str的回文子串。求str中最长的那个回文子串。

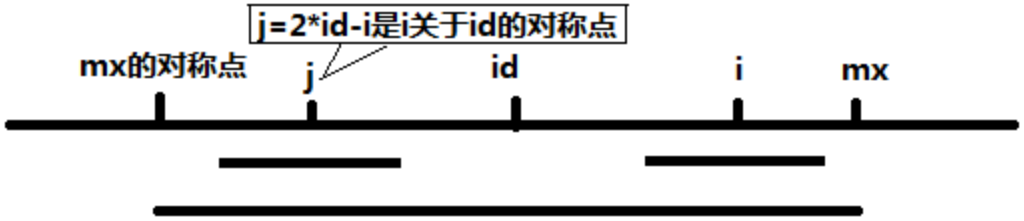
- 解决策略
  - 枚举中心位置，$O(N^2)$
  - Manacher算法，线性O(N)

Manacher 算法step1——预处理

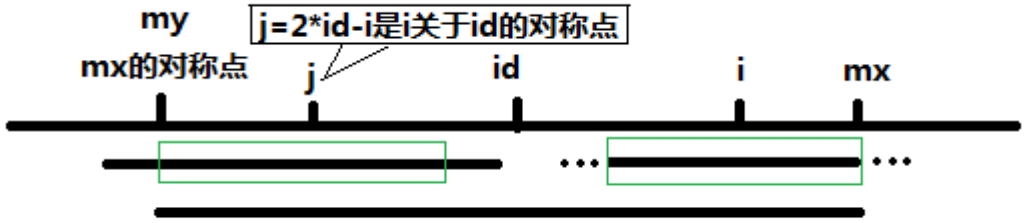
- 每个字符两边插入一个特殊的符号#
  - abbc → #a#b#b#c#
  - aba → #a#b#a#
- 继续插入一个字符
  - 字符串12212321→ S[] = "$#1#2#2#1#2#3#2#1#";
- 用一个数组 P[i] 来记录以字符S[i]为中心的最长回文子串向左/右扩张的长度（包括S[i]）
  - 比如S和P的对应关系
    - S # 1 # 2 # 2 # 1 # 2 # 3 # 2 # 1 #
    - P 1 2 1 2 5 2 1 4 1 2 1 6 1 2 1 2 1
      - 其中，P[i]-1正好是原字符串中回文串的总长度

Manacher 算法step 2——计算P[i]

- 增加两个辅助变量id和mx
  - id表示最大回文子串中心的位置
  - mx则为id+P[id]，也就是最大回文子串的边界。得到一个很重要的结论：
    - **如果mx > i，那么P[i] >= Min(P[2 * id - i], mx - i)**
  - 令j = 2 * id - i，也就是说j是i关于id的对称点。
    - 当 mx - i > P[j] ，以S[j]为中心的回文子串被包含在以S[id]为中心的回文子串中，因 i 和 j 对称，以S[i]为中心的回文子串必然被包含在以S[id]为中心的回文子串中，所以P[i] = P[j]。



    - 当 P[j] >= mx - i ，以S[j]为中心的回文子串不一定被完全包含于以S[id]为中心的回文子串中，但是基于对称性可知，图中两个绿框所包围的部分是相同的，也就是说以S[i]为中心的回文子串，其向右至少会扩张到mx的位置，也就是说 P[i] >= mx - i。



代码——线性时间解决最长回文(Manacher)

```
void Manacher()
{
    int i, mx = 0;
    int id;
    for(i=1; i<n; i++){
        //由上张PPT的说明，可得：
        if( mx > i )
            p[i] = MIN( p[2*id-i], mx-i );
        else
            //mx <= i ，无法对P[i]做更多的假设，直接让P[i] = 1
            p[i] = 1;
        for(; str[i+p[i]] == str[i-p[i]]; p[i]++);
        if( p[i] + i > mx ){
            mx = p[i] + i;
            id = i;
        }
    }
}
```

如何学习算法？

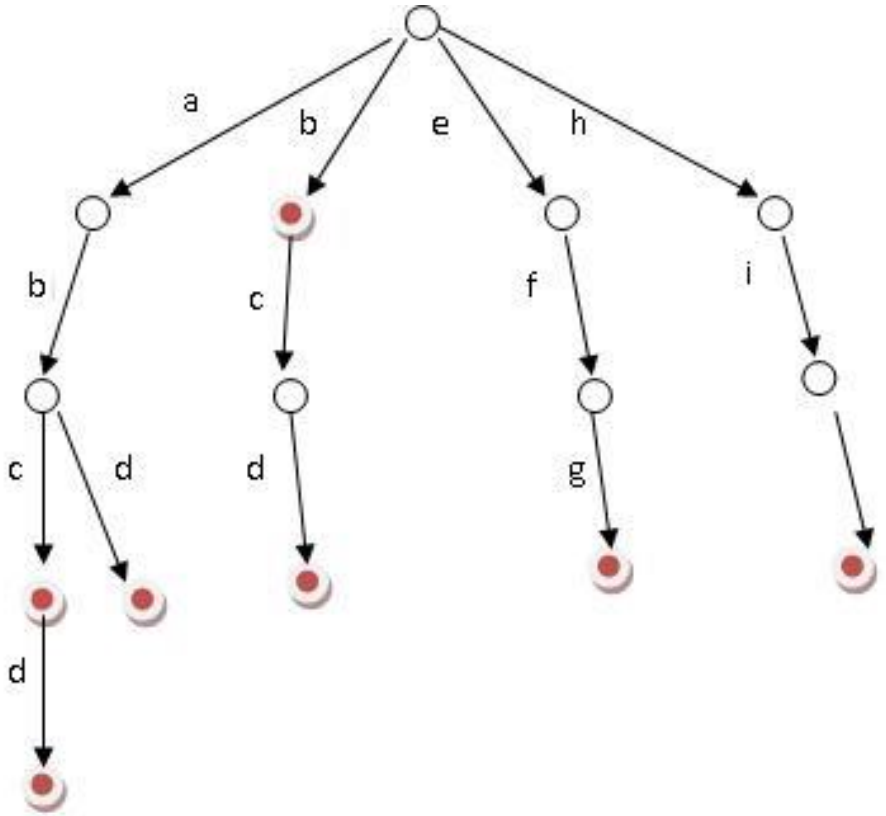
算法学习方法论

- 基础很重要
- 学习什么，心中有大纲
- 算法解决什么问题，解决策略是什么
  - 广搜
    - 一层一层往外遍历，寻找最短路径
    - 策略：队列
  - 最小生成树
    - 最小代价连接所有点
    - 策略：贪心（Prim：贪心+权重队列）
  - Dijkstra
    - 寻找单源最短路径
    - 策略：贪心+非负权重队列
  - Floyd
    - 多结点对的最短路径
    - 策略：动态规划
- 方法论
  - 对比联系
  - 从简单入手，追本溯源

- 要则一：把相关算法串联起来，相互比对
  - 比如trie树、后缀树
  - 贪心、动态规划
- 要则二：从简单入手，追本溯源
  - 二叉树、红黑树、2-3-4树、B树

Trie 树

- 每个节点是一个字母
- 从根到某节点的路径作为一个单词
- 每个节点维护一个boolean值
- 优化：
  - 压缩节点
- 时间复杂度：
  - 查找O(len)
- 空间复杂度： 跟公共前缀有关



存储前缀，统计后缀

- Trie树 + TOP K
  - hashmap+堆，hashmap+堆统计出如10个近似的热词

没这么简单

- （讨论）搜索引擎的智能提示
  - 注意只能解决前缀的问题
  - 如何存储？分布？
    - 有无slave & master?
    - P2P?
  - 如何更新？ 尽量透明？
  - 中文怎么处理？ 化成拼音？
  - 用途：最长前缀匹配！

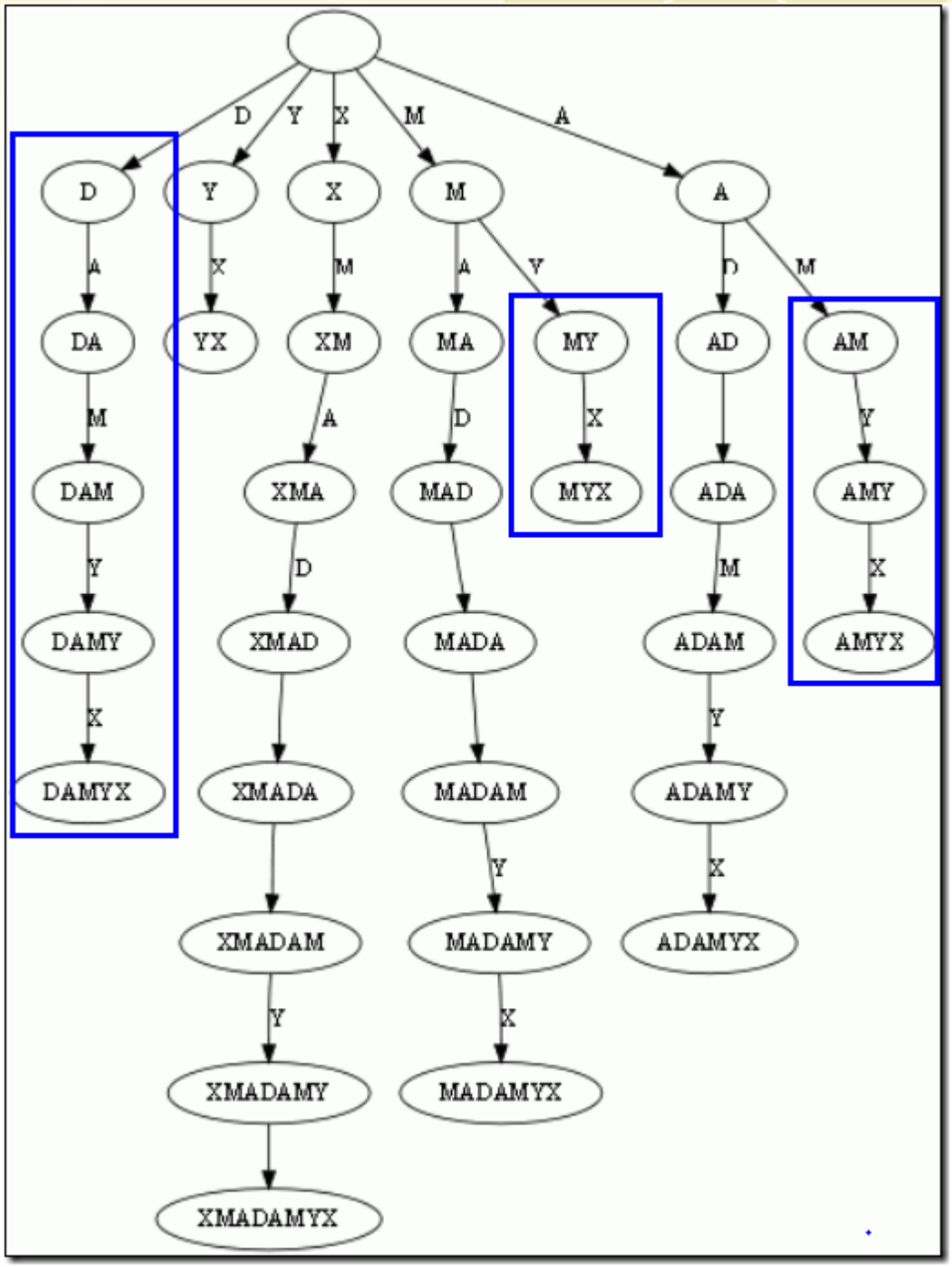
后缀树 1/4

- 以字符串S=XMADAMYX为例，它的长度为8，所以S[1..8], S[2..8], ...，S[8..8]都算S的后缀，我们一般还把空字串也算成后缀。这样，我们一共有如下后缀（对于后缀S[i..n]，我们说这项后缀起始于i）：

S[1..8], XMADAMYX， 也就是字符串本身，起始位置为1
S[2..8], MADAMYX，起始位置为2
S[3..8], ADAMYX，起始位置为3
S[4..8], DAMYX，起始位置为4
S[5..8], AMYX，起始位置为5
S[6..8], MYX，起始位置为6
S[7..8], YX，起始位置为7
S[8..8], X，起始位置为8
空字串，记为$

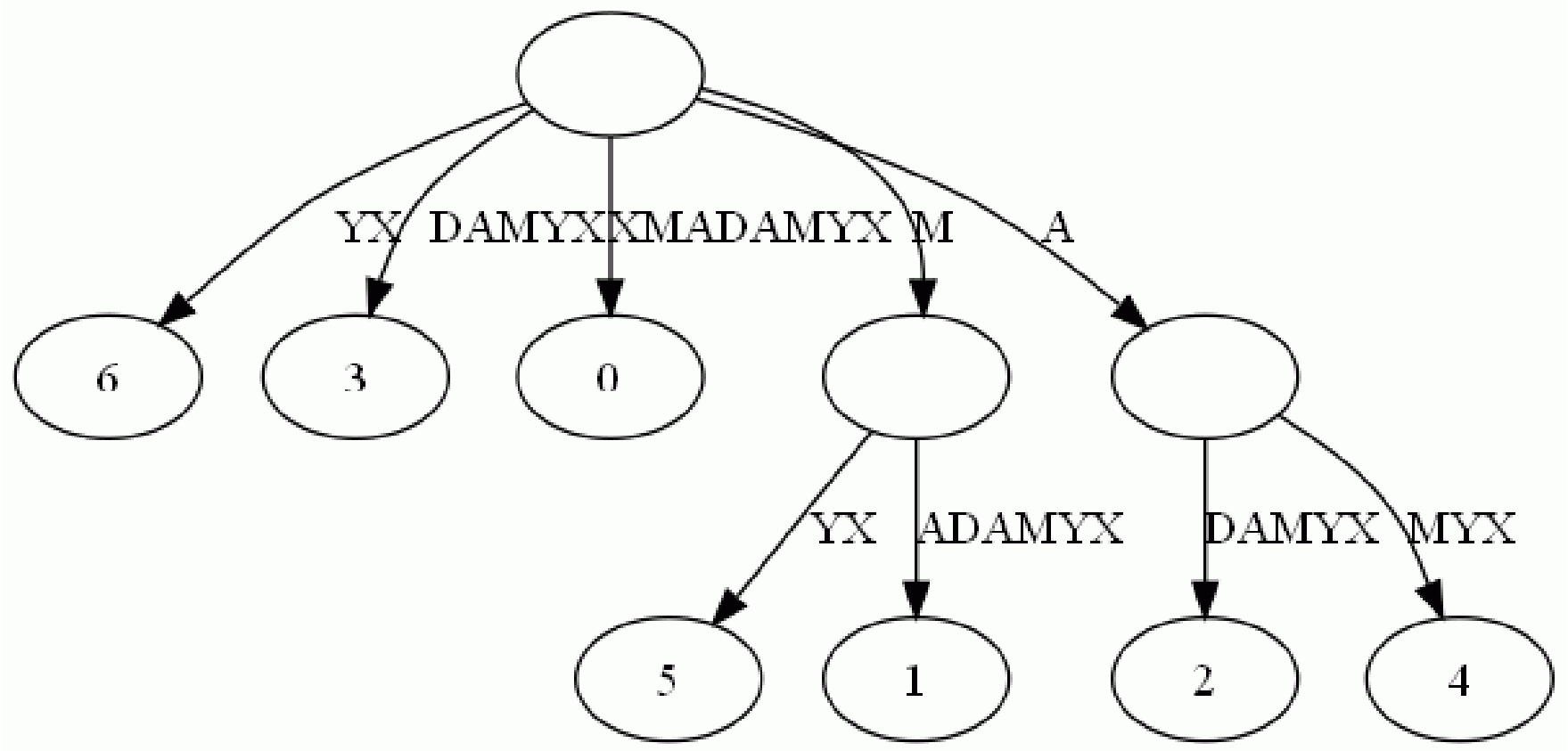
后缀树 2/4

把上面的后缀加入Trie后，
我们得到右边的结构：



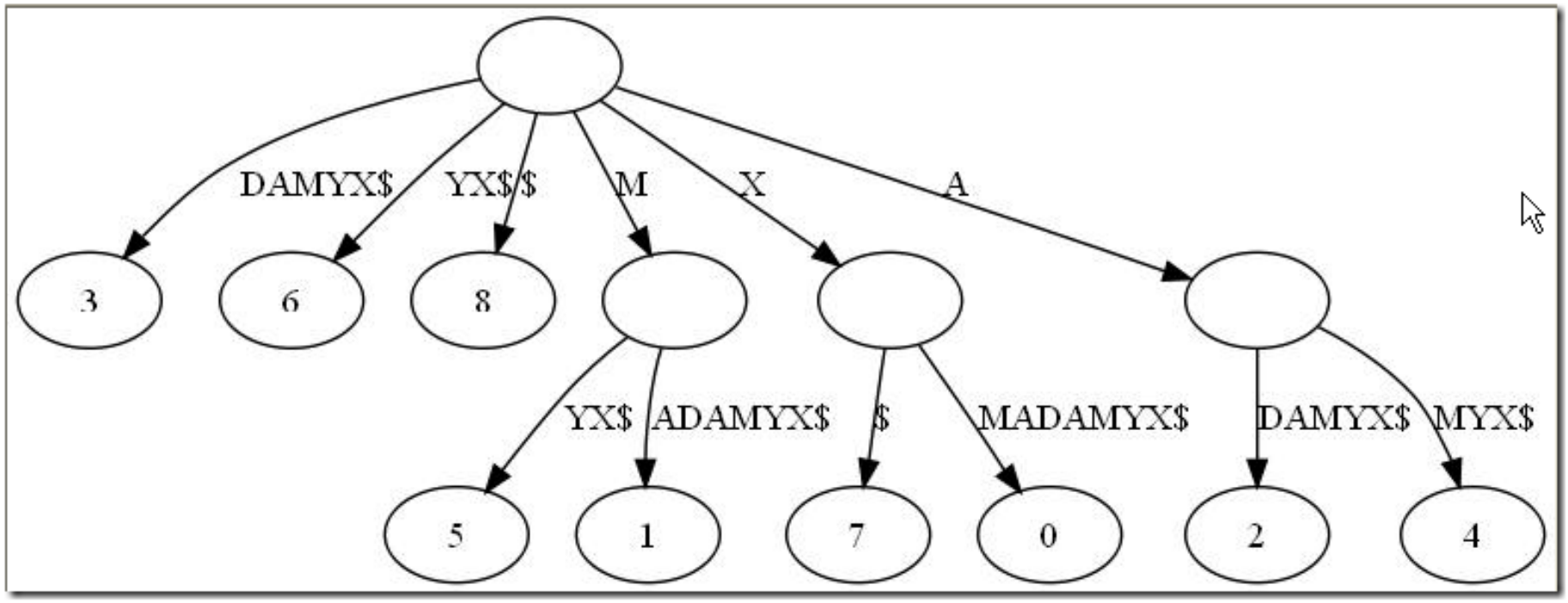
后缀树 3/4

- 把这种没有分叉的路径压缩到一条边
- 并在叶节点上标注上每项后缀的起始位置



后缀树 4/4

- 在待处理的子串后加一个空字串
  - 例如我们处理XMADAMYX前，先把XMADAMYX变为 XMADAMYX$，于是就得到suffix tree--后缀树了



贪心算法

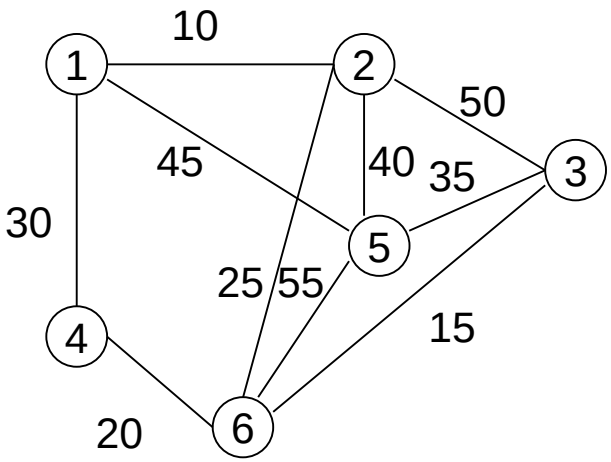
- 特点
  - 容易想到
  - 证明可能困难
  - 应用广泛

- 应用
  - MST (prim, Krusal)
  - Dijkstra
  - Huffman
  - LRU缓存替换机制 （其实最好的机制是换掉最远不使用）
  - 其他问题的启发式——给出近似解

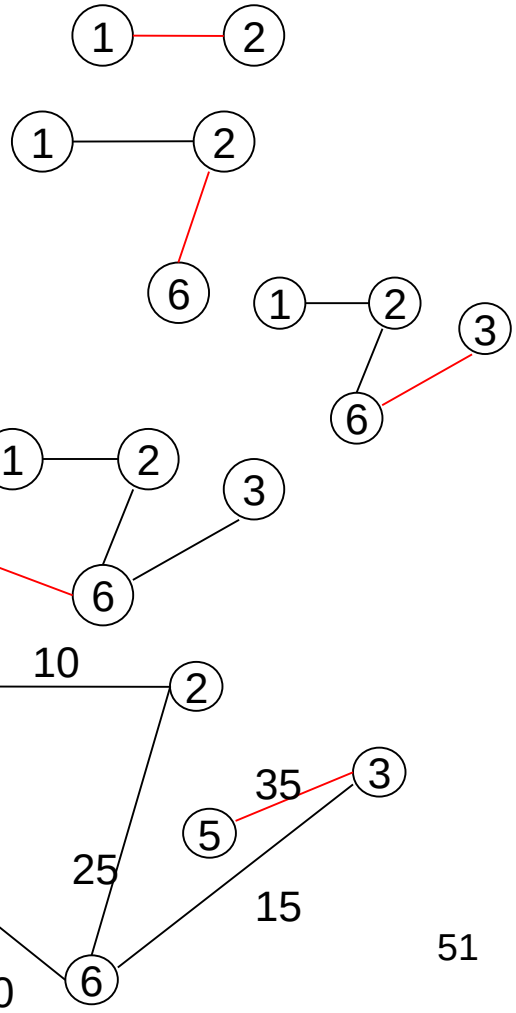
Prim算法

策略：

使得迄今所选择的边的集合A构成一棵树；则将要计入到A中的下一条边(u,v)，应是E中一条当前不在A中且使得A∪{(u,v)}也是一棵树的最小成本边。



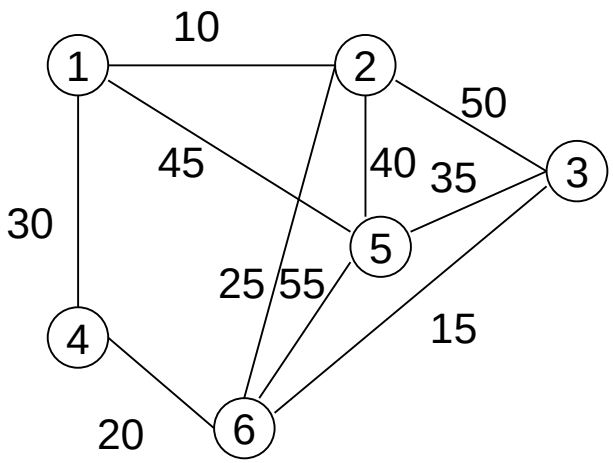
边	成本
(1,2)	10
(2,6)	25
(3,6)	15
(6,4)	20
(3,5)	35



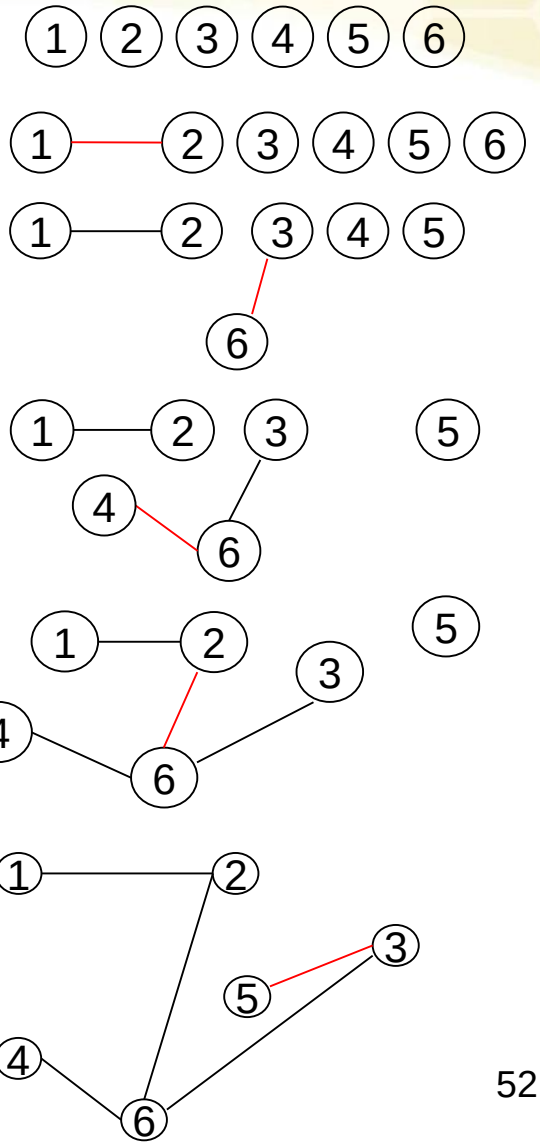
Kruskal算法

策略：

图G的所有边按成本非降次序排列，下一条生成树T中的边是尚未加入树的边中具有最小成本、且和T中现有的边不会构成环路的边。



边	成本
(1,2)	10
(3,6)	15
(4,6)	20
(2,6)	25
(3,5)	35



动态规划

两个简单的例子

- 最短路径：A -> B
  - 经过x1，x2，x3
    - 故枚举所有可能从A到B要经过的路径
    - 选择一条最优
      - 如何求最优
        - » 比较
        - » 如何比较？写DP方程，min之类的
        - » 如何写？练.
        - » 如何练？..

- 比如二维数组最小路径和
  - 一个二维矩阵M*N矩阵matrix中，找出一条路径，只能向右或向下，求路径经过元素之和最小
    - 当前位置(i, j)
      - 上一个位置只可能是(i-1， j) 或 (i, j-1)
        - » 故：path[i][j] = min(path[i][j-1], path[i-1][j]) + matrix[i][j]

寻找和为定值的多个数

- 输入两个整数 n 和 m，从数列1，2，3.....n 中 随意取几个数,使其和等于 m ,要求将其中所有的可能组合列出来。

```
list1.push_front(n);      //典型的01背包问题
find_factor(sum-n, n-1);  //放n，n-1个数填满sum-n
list1.pop_front();
find_factor(sum, n-1);    //不放n，n-1个数填满sum
```

- 动态规划适用条件
  - 最优子结构
  - 独立重叠子问题

交替字符串

- 输入三个字符串s1、s2和s3，判断第三个字符串s3是否由前两个字符串s1和s2交错而成，即不改变s1和s2中各个字符原有的相对顺序
  - 例如当s1 = “aabcc”，s2 = “dbbca”，s3 = “aadbbcbcac”时，则输出true，但如果s3=“accabdbbca”，则输出false。

- 解法
  - 令dp[i][j]代表s3[0...i+j-1]是否由s1[0...i-1]和s2[0...j-1]的字符组成
    - 如果s1当前字符（即s1[i-1]）等于s3当前字符（即s3[i+j-1]），而且dp[i-1][j]为真，那么可以取s1当前字符而忽略s2的情况，dp[i][j]返回真；
    - 如果s2当前字符等于s3当前字符，并且dp[i][j-1]为真，那么可以取s2而忽略s1的情况，dp[i][j]返回真，其它情况，dp[i][j]返回假

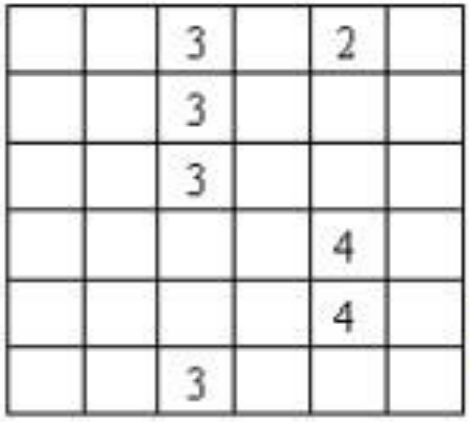
格子取数问题

- 题目
  - 有n*n个格子，每个格子里有正数或者0，从最左上角往最右下角走，只能向下和向右，一共走两次（即从左上角走到右下角走两趟），把所有经过的格子的数加起来，求最大值SUM，且两次如果经过同一个格子，则最后总和SUM中该格子的计数只加一次。

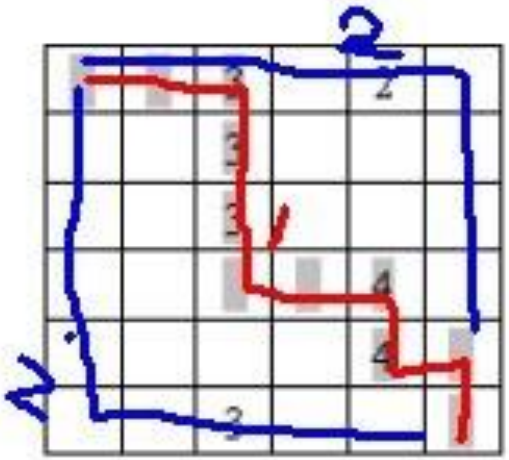
A	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	13	0	0	6	0	0
3	0	0	0	0	7	0	0	0
4	0	0	0	14	0	0	0	0
5	0	21	0	0	0	4	0	0
6	0	0	15	0	0	0	0	0
7	0	14	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0

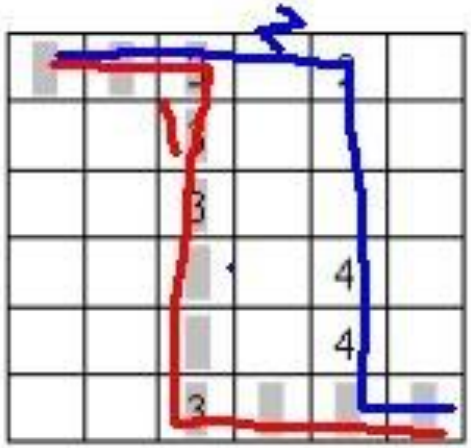
贪心陷阱

- 不顾全局，只看局部，让第一次和第二次的路径都是最优



图一


图二


图三

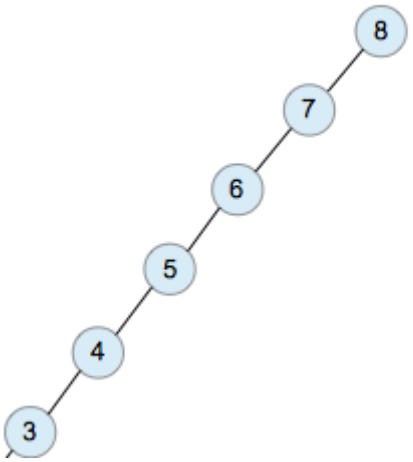
追本溯源

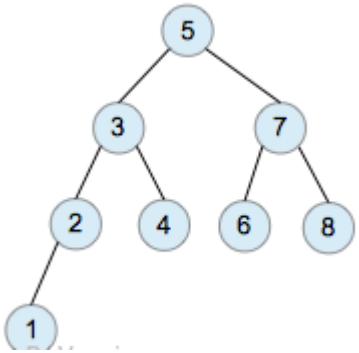
- 红黑树
  - 为何要有RB-Tree
    - 完全平衡完全二叉树
    - 高度平衡AVL树
  - 先理解二叉树的插入、删除
  - 后理解红黑树的插入修复、删除修复

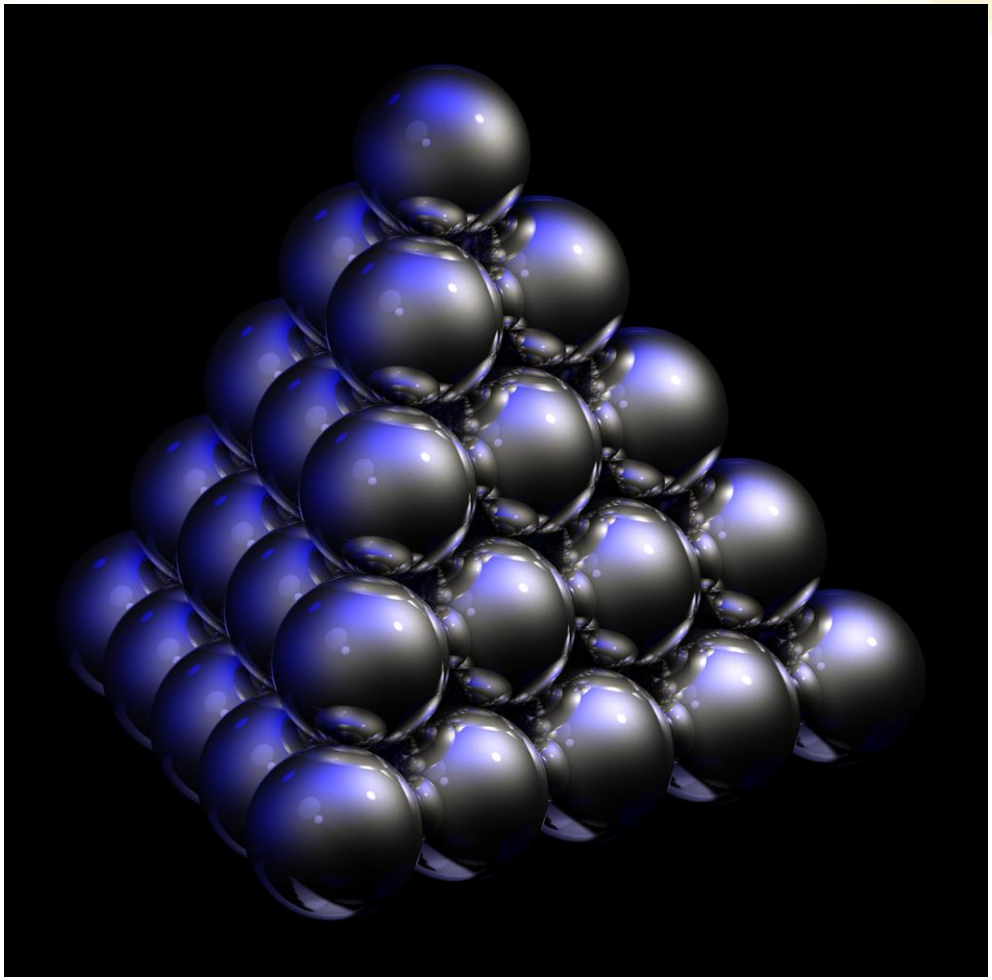
- B树
  - 先学习2-3-4树，理解结点饱和分裂，结点稀缺合并
  - 为何？
    - 因为2-3-4 树在计算机科学中是阶为 4 的B树
    - 意味着什么？
      - 意味着2-3-4树中每个结点的关键字数目是：1-3个
        - （ceil(m / 2)-1）<= n <= m-1
        - m为阶数，即孩子树，等于4

二叉树到完全二叉树

- 树的深度越小，搜索时间logn（n即为树的深度）

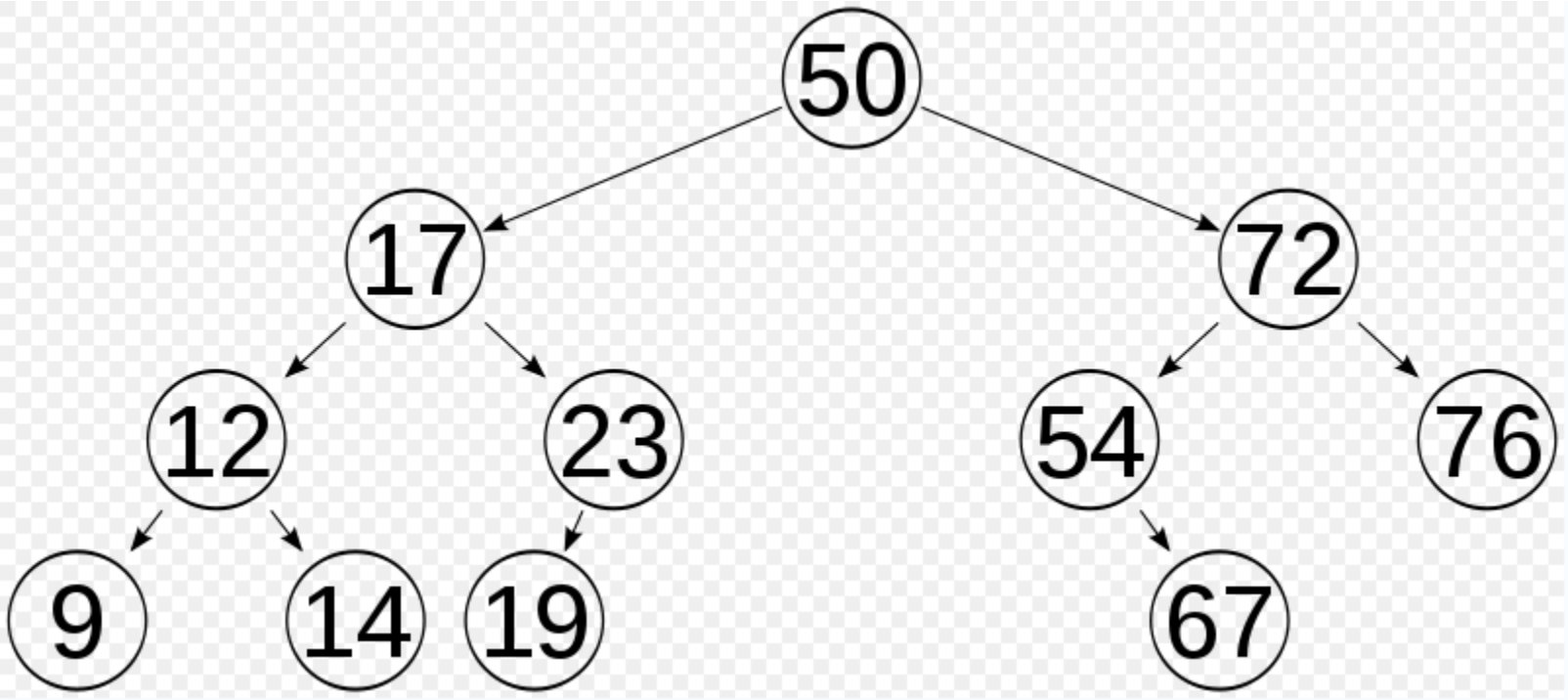






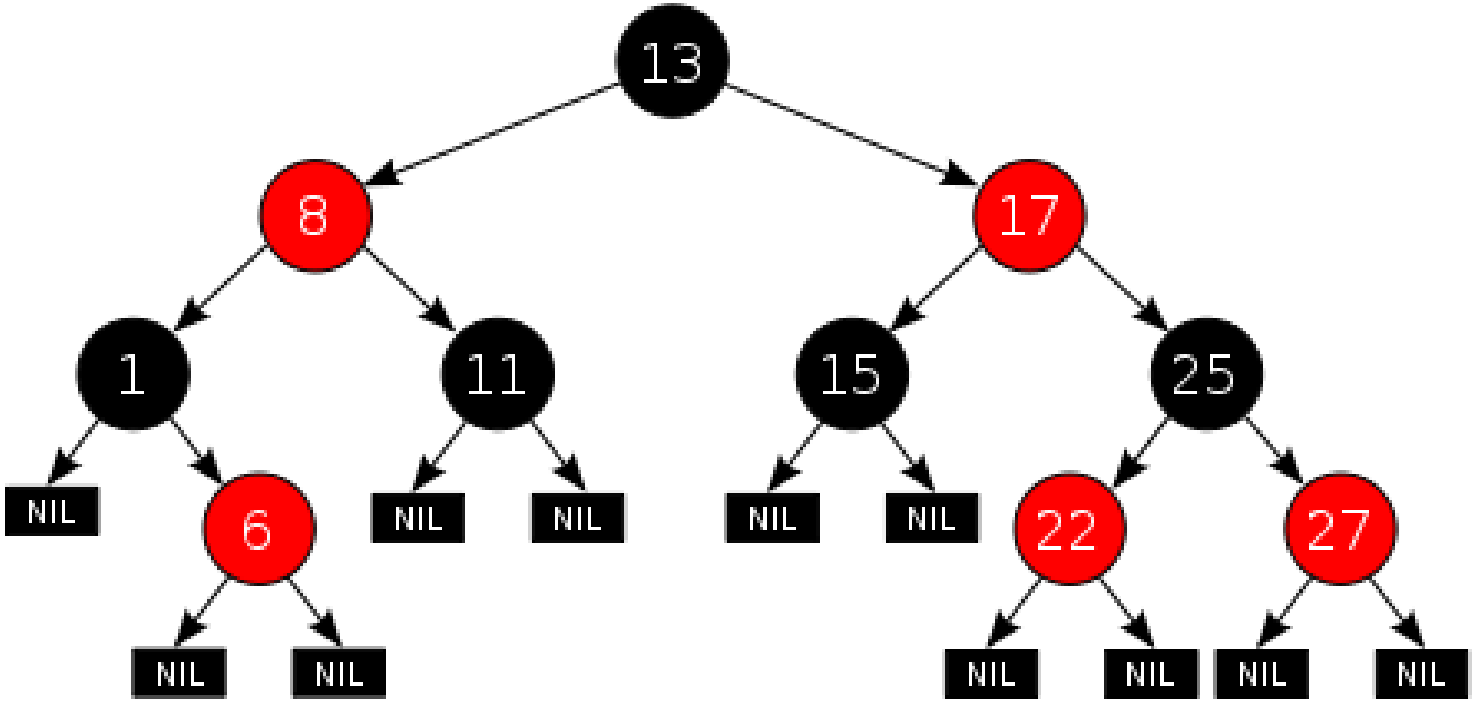
AVL树

- 高度平衡树
  - AVL树中任何节点的两个子树的高度最大差别为一



红黑树的5个性质

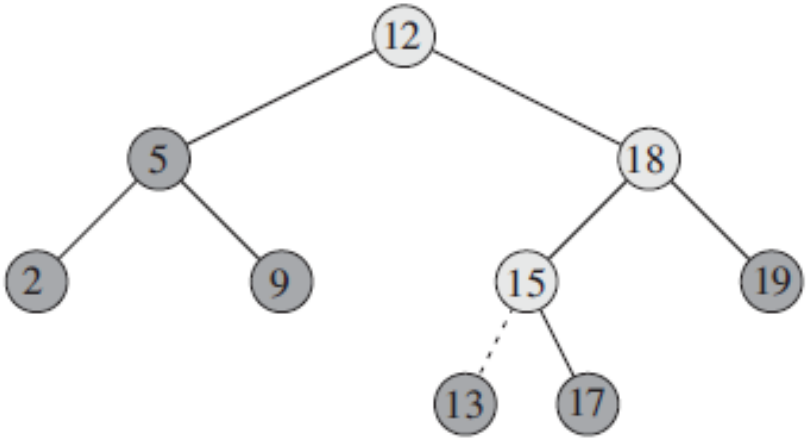
1. 每个结点要么是红的，要么是黑的。
2. 根结点是黑的。
3. 每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。
4. 如果一个结点是红的，那么它的俩个儿子都是黑的。
5. 对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。



二叉树的插入

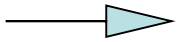
- 插入一个节点

```
TREE-INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z          // tree T was empty
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```



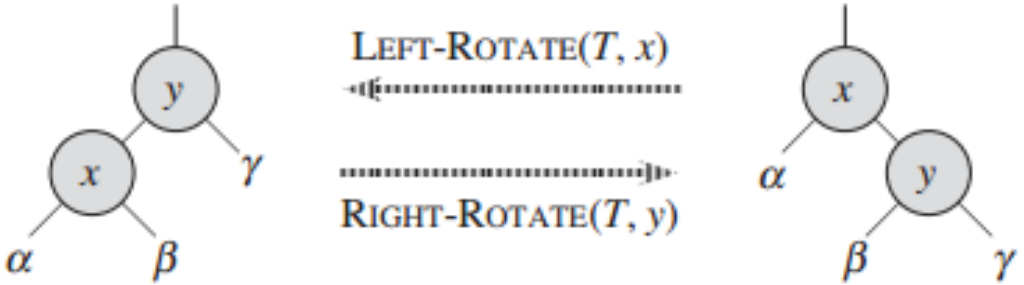
红黑树的插入

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z          // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```



```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == T.nil
10        T.root = z
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

- 插入一个元素后，需要修复，修复有两种手段
  - 重新着色
  - 旋转操作：左旋与右旋



红黑树的插入修复代码

- 3种插入修复情况

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // case 1
 6              y.color = BLACK                      // case 1
 7              z.p.p.color = RED                    // case 1
 8              z = z.p.p                            // case 1
 9          else if z == z.p.right
10                  z = z.p                          // case 2
11                  LEFT-ROTATE(T, z)                // case 2
12              z.p.color = BLACK                    // case 3
13              z.p.p.color = RED                    // case 3
14              RIGHT-ROTATE(T, z.p.p)               // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

插入一个元素而后修复的几种情况 1/3

1）每个结点要么是红的，要么是黑的。
2）根结点是黑的。
3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。
4）如果一个结点是红的，那么它的俩个儿子都是黑的。
5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

- 插入修复情况 [预处理]
  - 插入的是根结点
    - 原树是空树，此情况只会违反性质2
      - 对策：直接把此结点涂为黑色。
  - 插入修复情况
    - 插入的结点的父结点是黑色。
      - 此不会违反性质2和性质4，红黑树没有被破坏。
      - 对策：什么也不做。

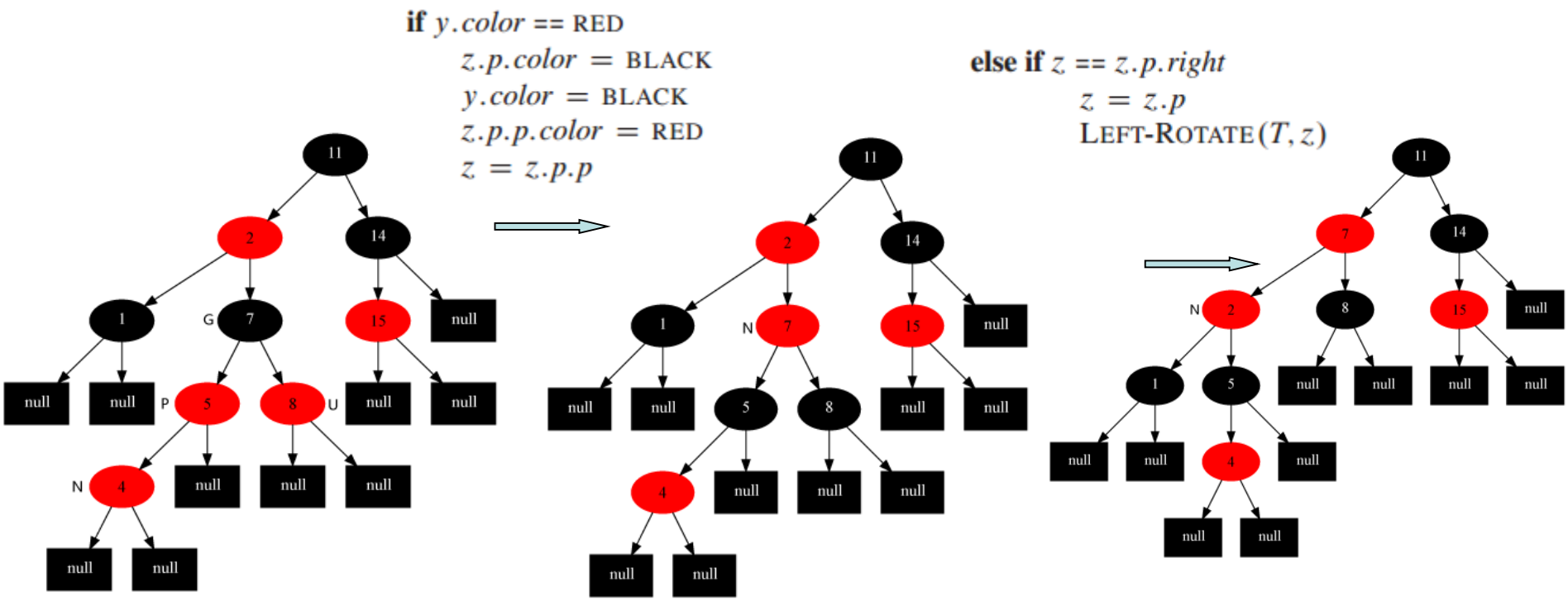
插入一个元素而后修复的几种情况 2/3

1）每个结点要么是红的，要么是黑的。
2）根结点是黑的。
3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。
4）如果一个结点是红的，那么它的俩个儿子都是黑的。
5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

- 插入修复情况1：当前结点的父结点是红色，且叔叔结点（祖父结点的另一个子结点）是红色
  - 对策：将当前节点的父节点和叔叔节点涂黑，祖父结点涂红，把当前结点指向祖父节点，从新的当前节点重新开始算法（此情况3将引发后续的一连锁反应：情况2 和 情况3）

[情况1变成了情况2]

- 插入修复情况2：当前节点的父节点是红色,叔叔节点是黑色，当前节点是其父节点的右子
  - 对策：当前节点的父节点做为新的当前节点，以新当前节点为支点左旋（还没完，在情况3中继续）



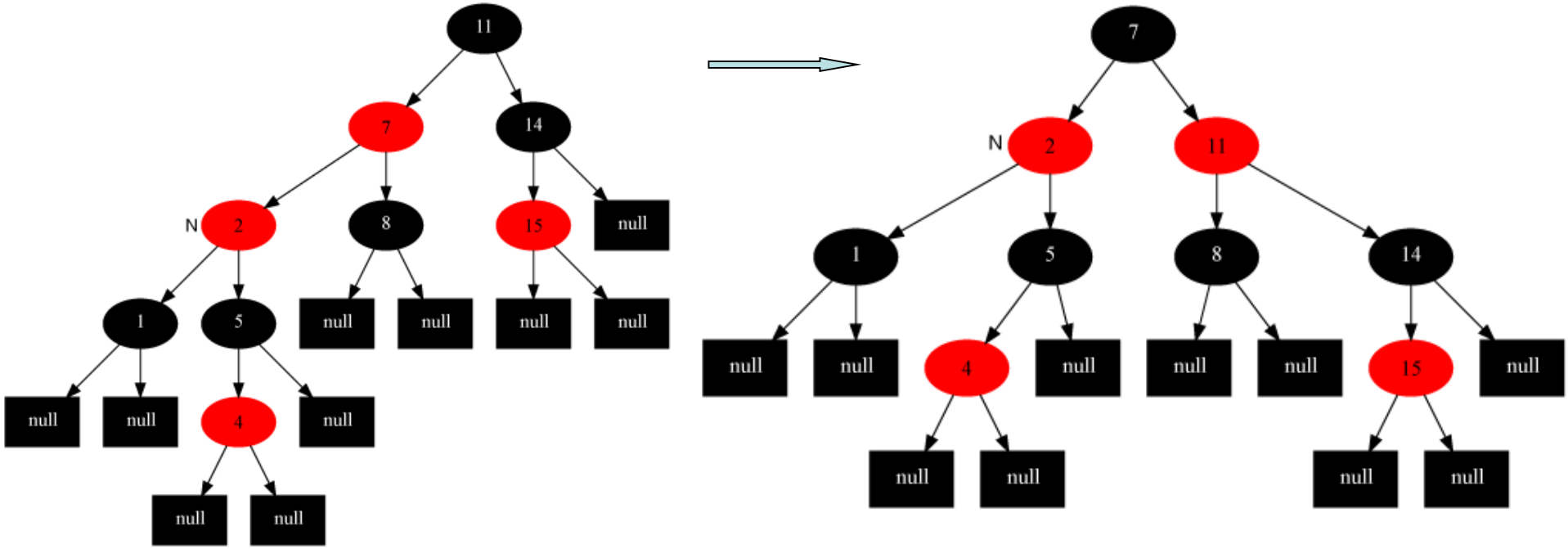
插入一个元素而后修复的几种情况 3/3

1）每个结点要么是红的，要么是黑的。
2）根结点是黑的。
3）每个叶结点（叶结点即指树尾端NIL指针或NULL结点）是黑的。
4）如果一个结点是红的，那么它的俩个儿子都是黑的。
5）对于任一结点而言，其到叶结点树尾端NIL指针的每一条路径都包含相同数目的黑结点。

[接情况2]

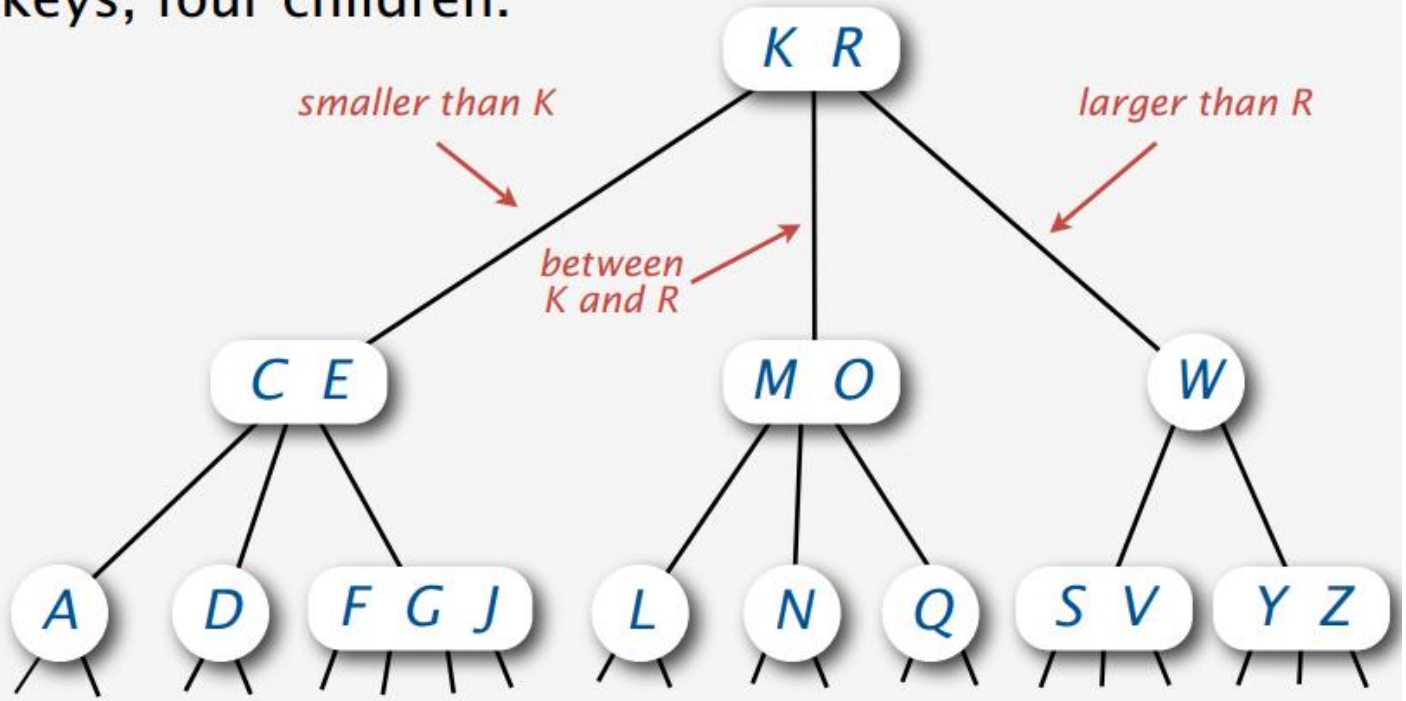
- 插入修复情况3：当前节点的父节点是红色,叔叔节点是黑色，当前节点是其父节点的左子
  - 解法：父节点变为黑色，祖父节点变为红色，在祖父节点为支点右旋 （从情况1到情况3，终于调整完成）

$z.p.color = \text{BLACK}$
$z.p.p.color = \text{RED}$
$\text{RIGHT-ROTATE}(T, z.p.p)$

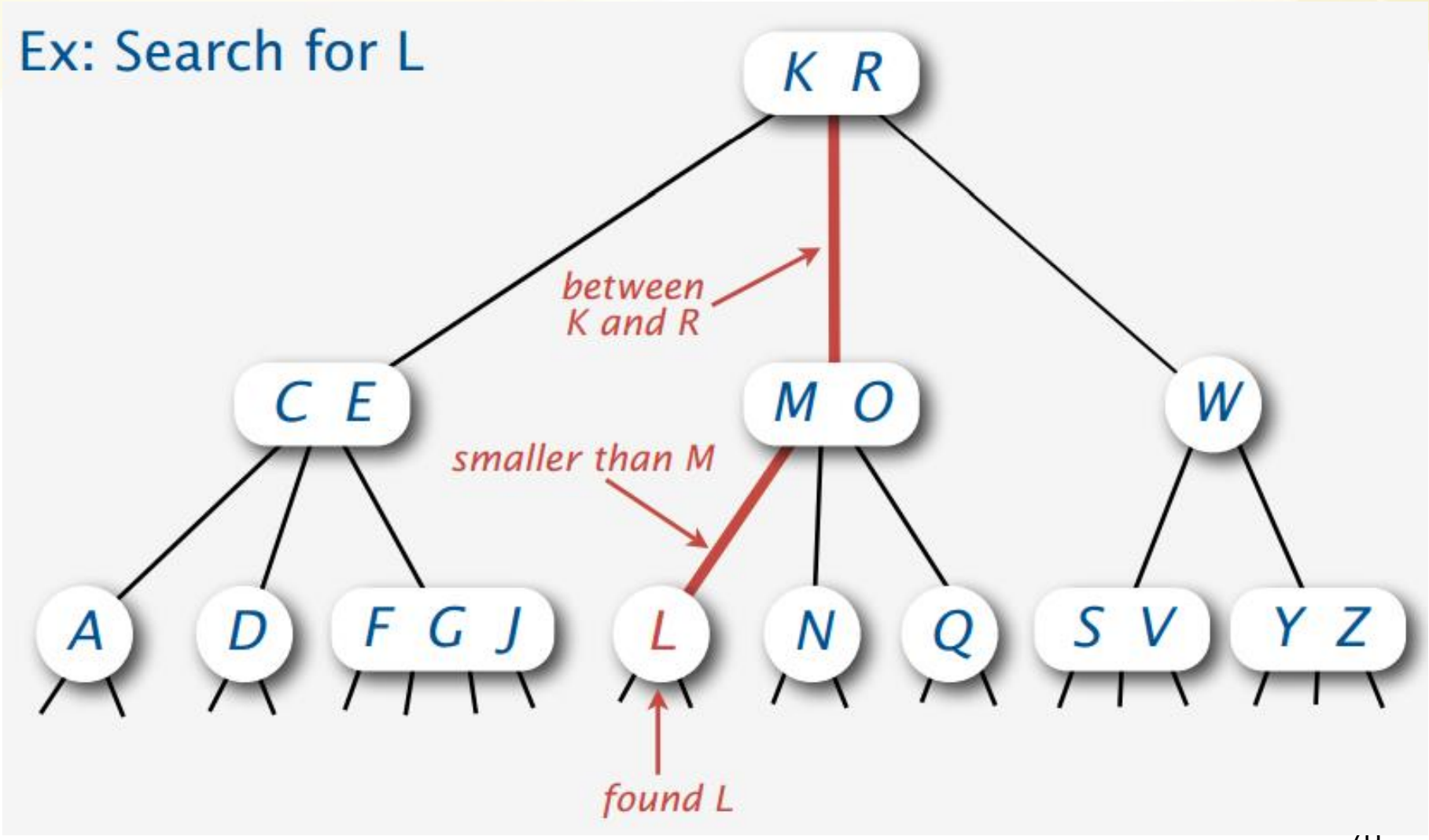


从2-3-4树开始谈起

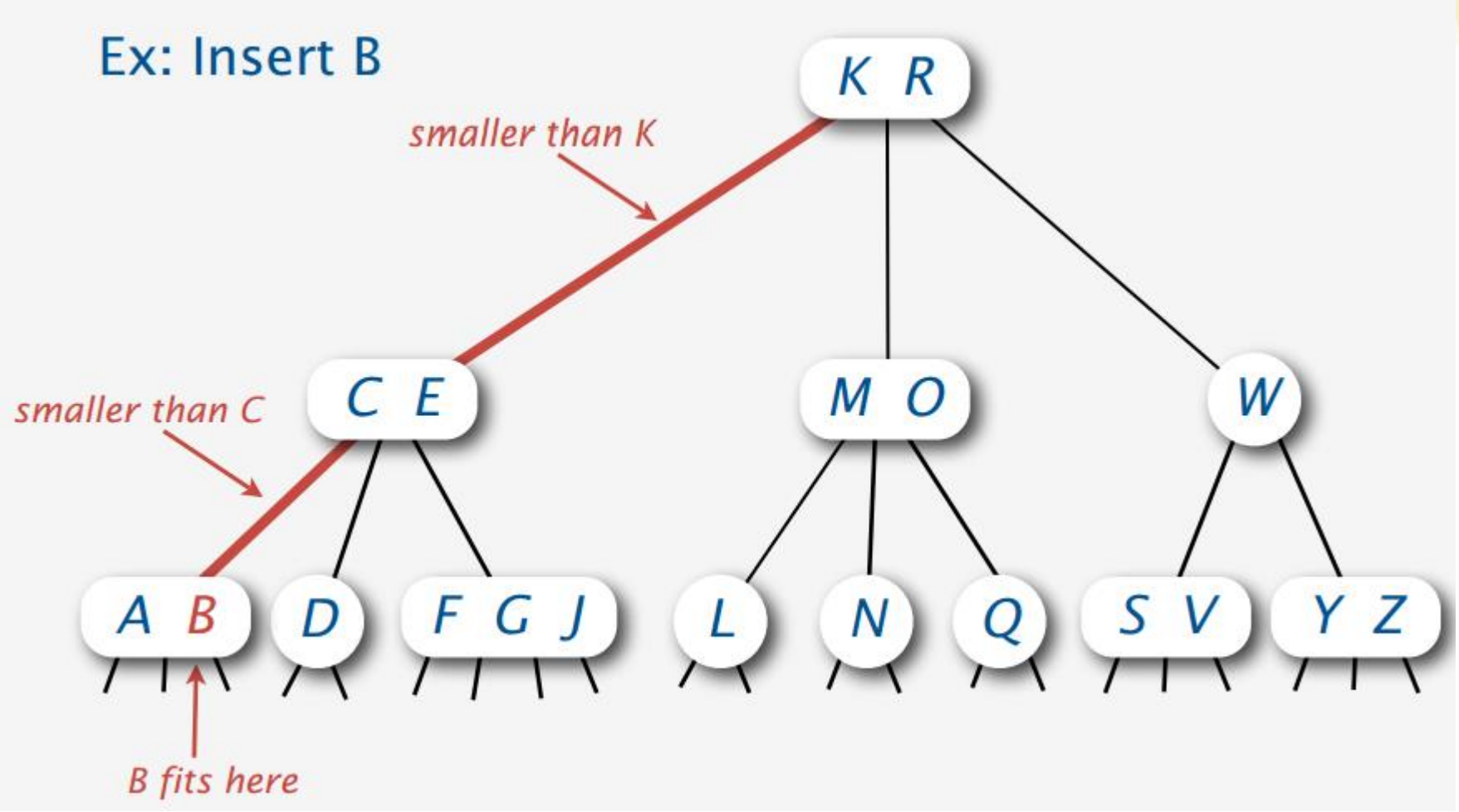
- 2-node: one key, two children.
- 3-node: two keys, three children.
- 4-node: three keys, four children.



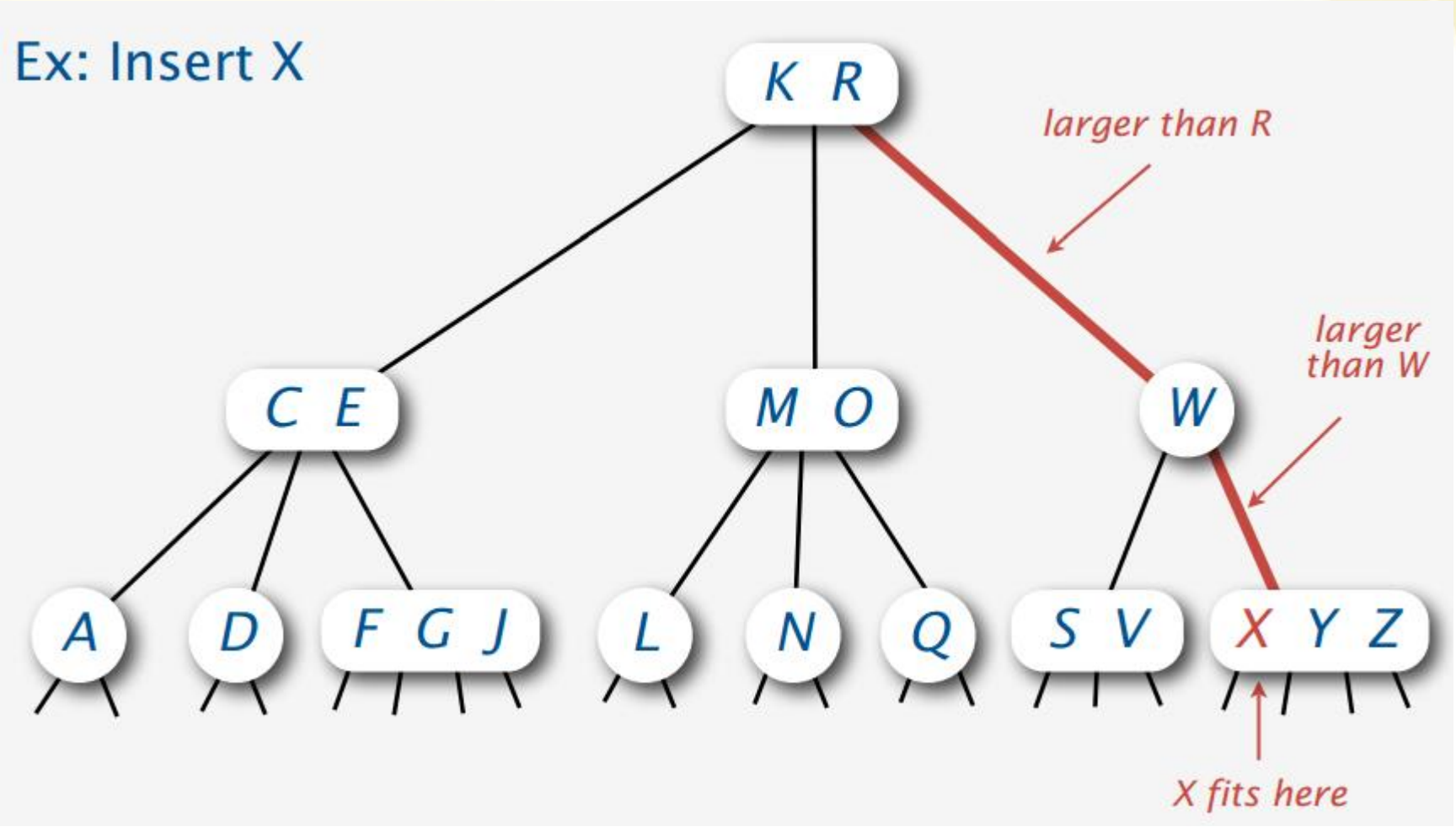
2-3-4树的查找


Ex: Search for L
K R
between
K and R
C E
M O
W
smaller than M
A
D
F G J
L
N
Q
S V
Y Z
found L

2-3-4树的插入 1/3

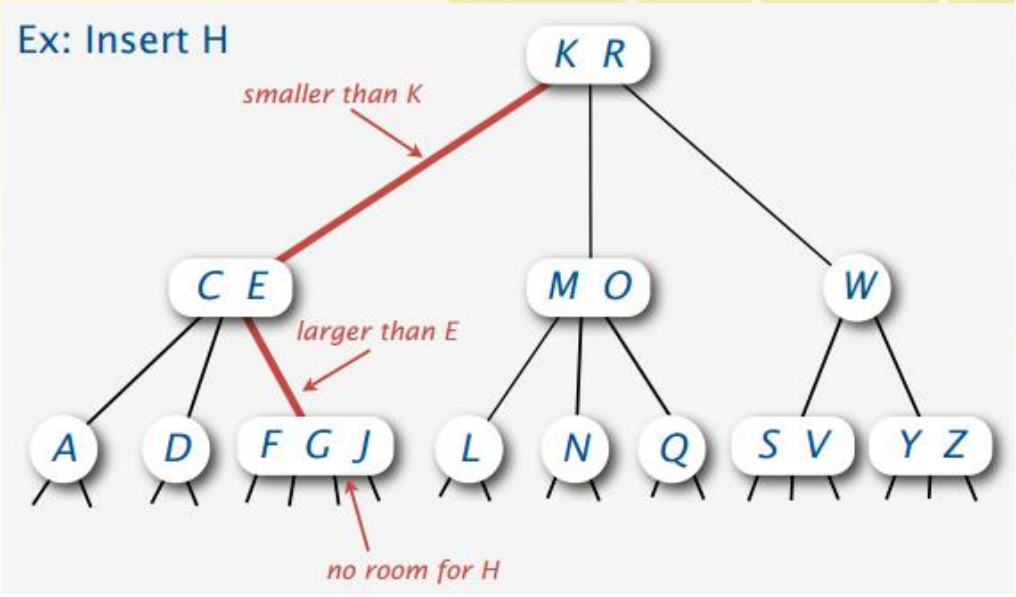

Ex: Insert B
K R
smaller than K
C E
smaller than C
M O
W
A B
D
F G J
L
N
Q
S V
Y Z
B fits here

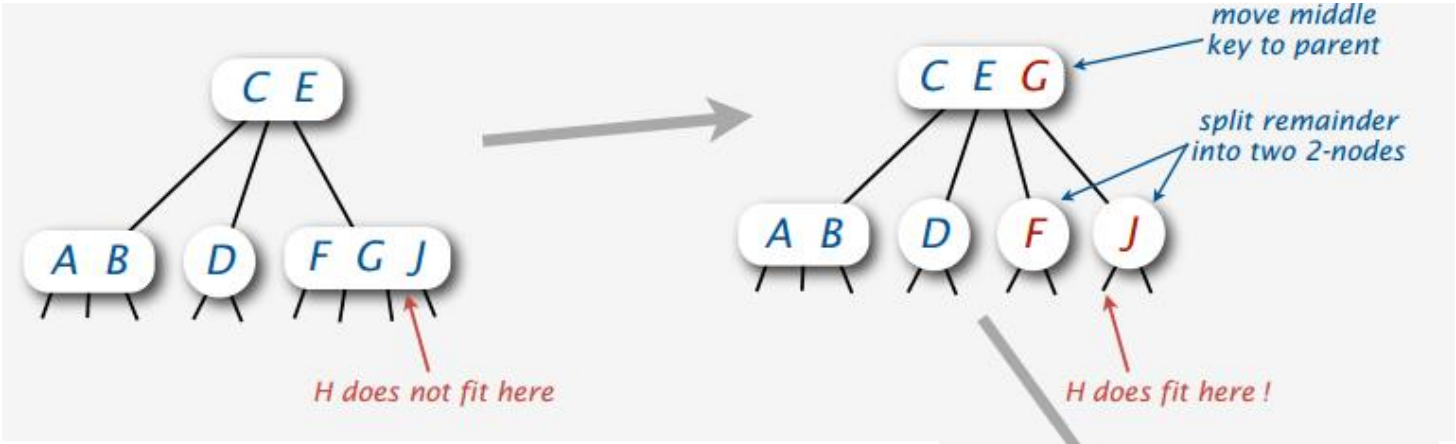
2-3-4树的插入 2/3

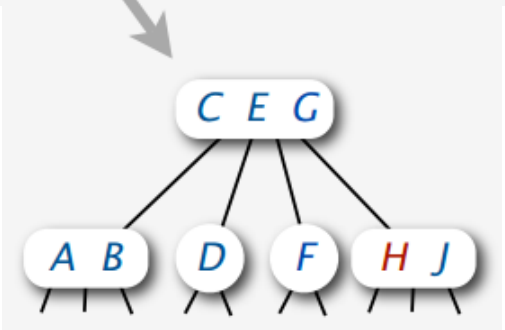

Ex: Insert X
K R
larger than R
larger than W
C E
M O
W
A
D
F G J
L
N
Q
S V
X Y Z
X fits here

2-3-4树的插入 3/3

- 关键字数要超过4时
  - 就要开始分裂
  - 4阶的B树的关键字数满足：
    - 大于等于1，小于等于3

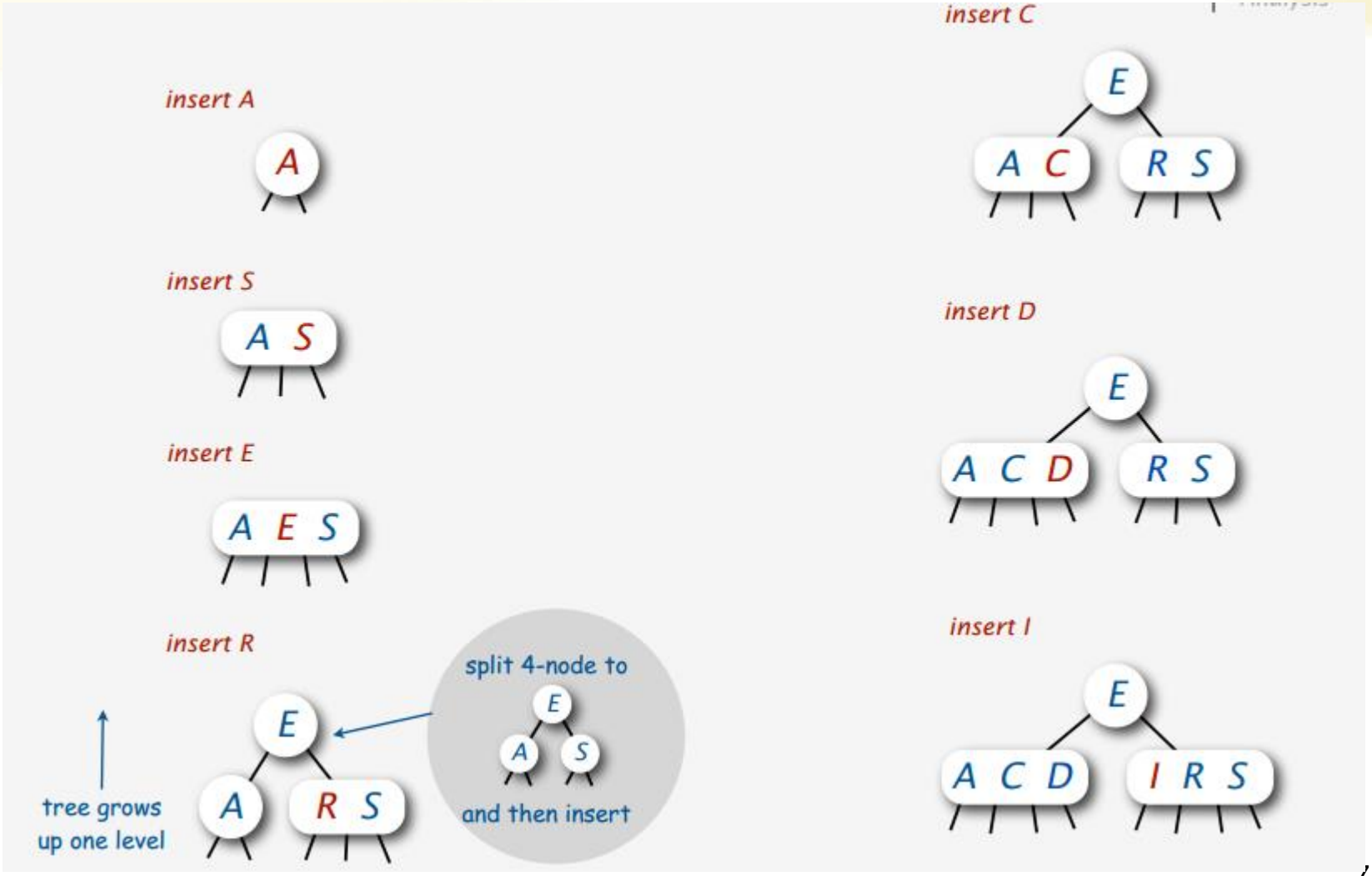






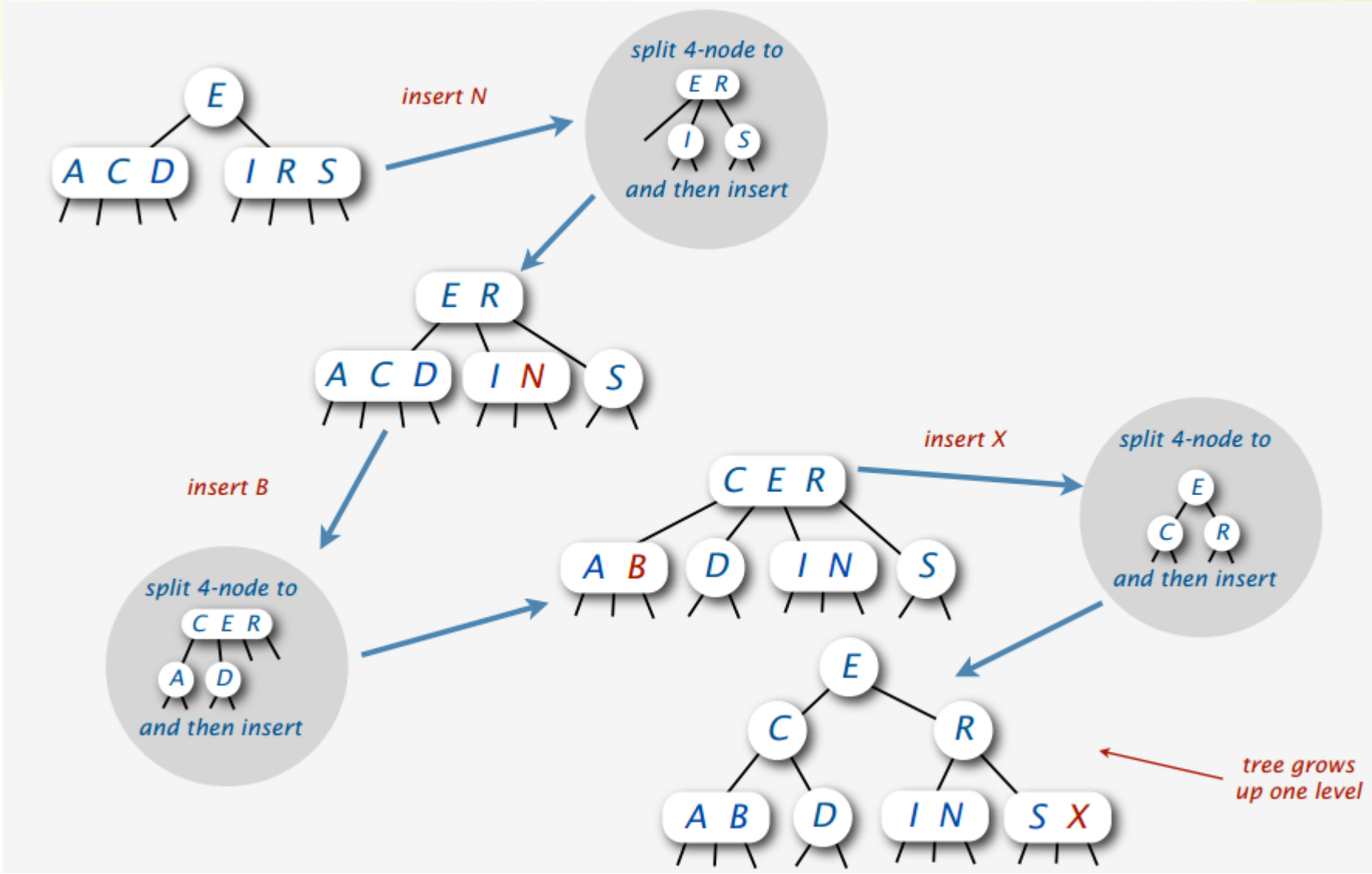
2-3-4树一次完整的插入示例 1/2

- 不断插入多个元素的过程



2-3-4树一次完整的插入示例 1/2

- 接上，继续插入元素N、B、X



- 看过了红黑树的插入
- 看过了2-3-4树分裂
- 接下来，看另外一种新树
  - 它与红黑树最大的区别在于，它的结点可以有许多子女，从几个到几千个

B树

- 出现缘由
  - 二叉查找树结构由于树的深度过大而造成磁盘I/O读写过于频繁，进而导致查询效率低下

一棵m阶的B树满足下列条件：

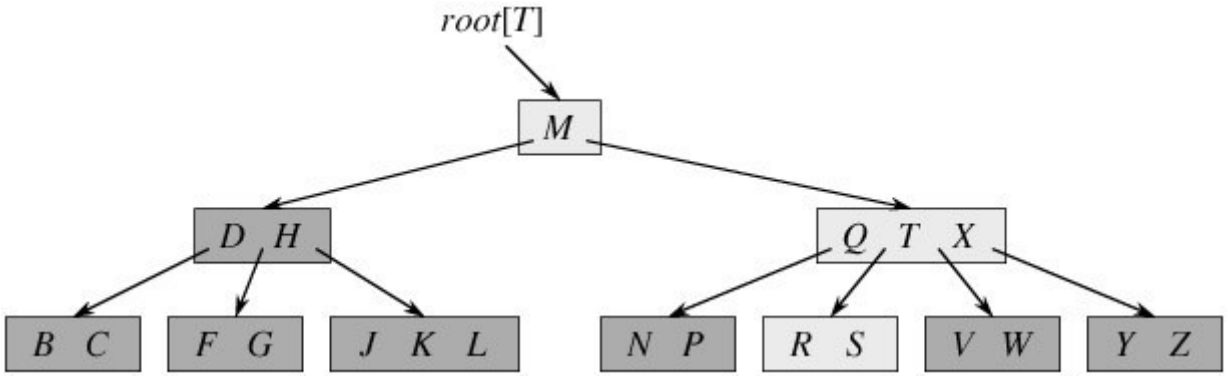
1，每个结点至多有m棵子树。

2，除根结点外，其它每个分支结点至少有 $\lceil m/2 \rceil$ 棵子树。

3，根结点至少有两棵子树(除非B树只包含一个结点)。

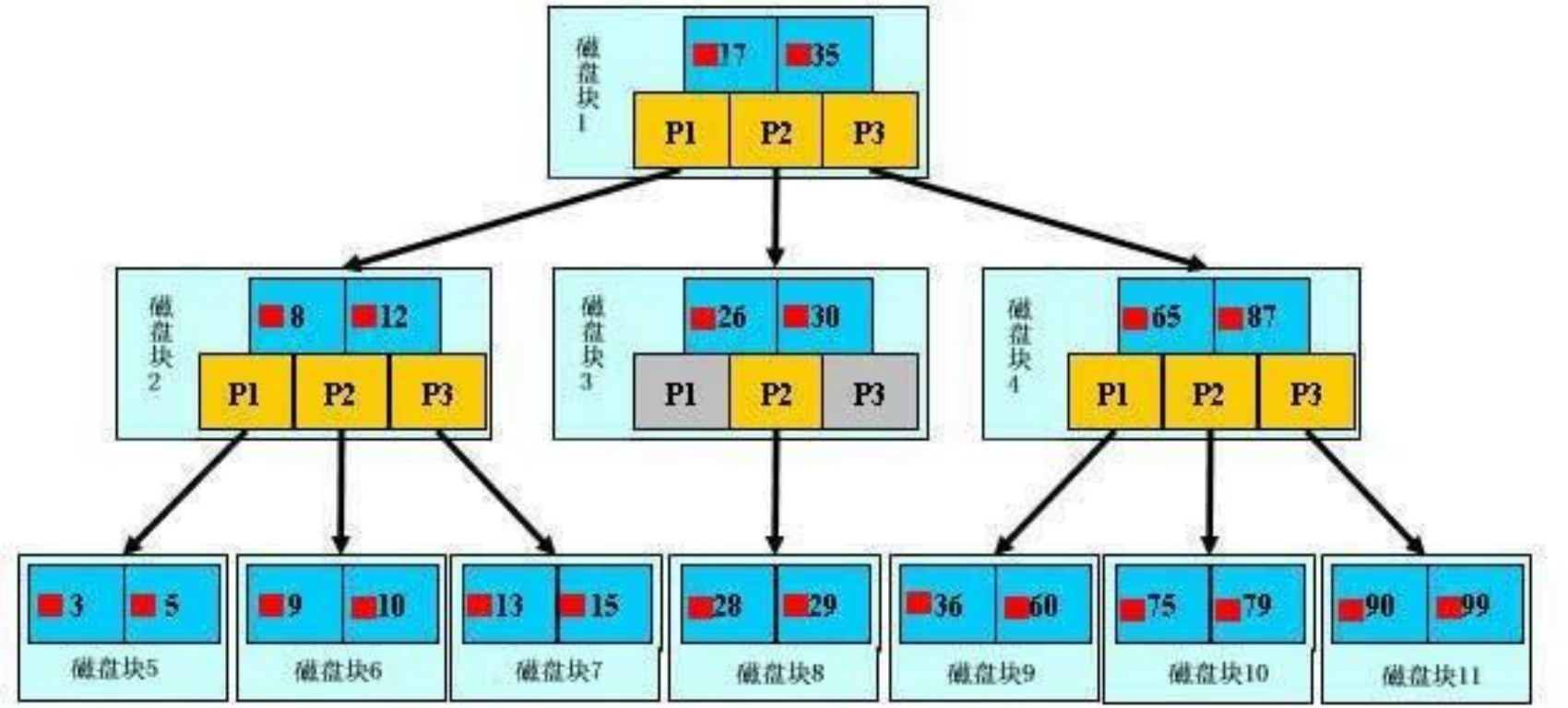
4，所有叶结点在同一层上。B树的叶结点可以看成一种外部结点，不包含任何信息。

5，有j个孩子的非叶结点恰好有j-1个关键码，关键码按递增次序排列。



一棵B树的示例

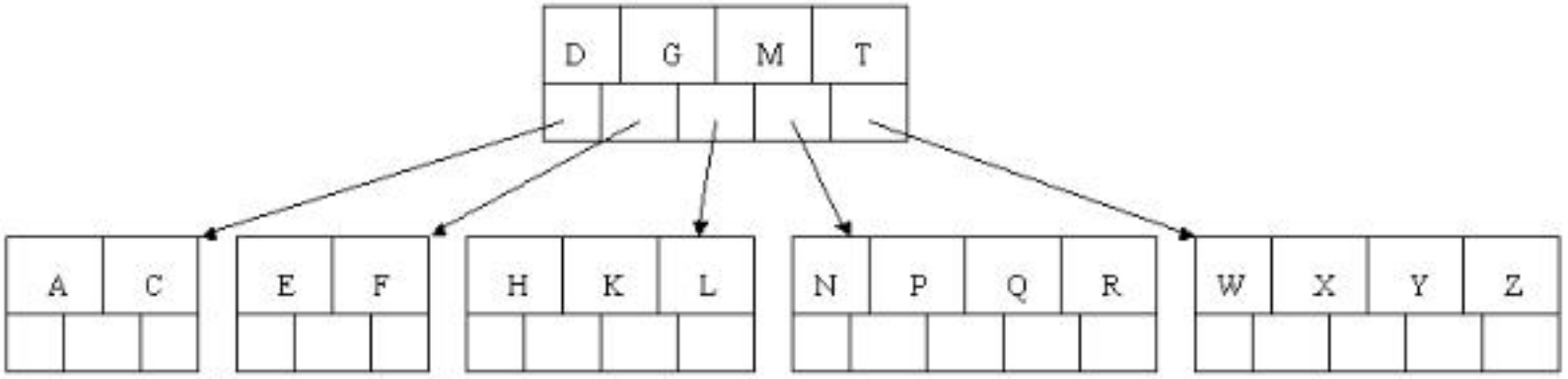
- 查找文件29
  - 一直往下，3次磁盘IO操作和3次内存查找



B树的插入示例 1/5

一棵***5***阶（即树中任一结点至多含有4个关键字，5棵子树）B树

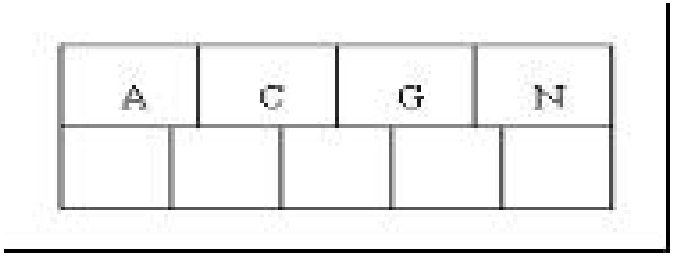
- 根结点至少得有2个孩子
- 5阶，2- 4 个key，3-5 个children
  - B树除根结点之外的结点（包括叶子结点）的关键字的个数n必须满足：
    - （ceil(m / 2)-1）<= n <= m-1
    - 2 <= 关键字数 <= 4
      - » m为孩子数，即子树的数目，等于5



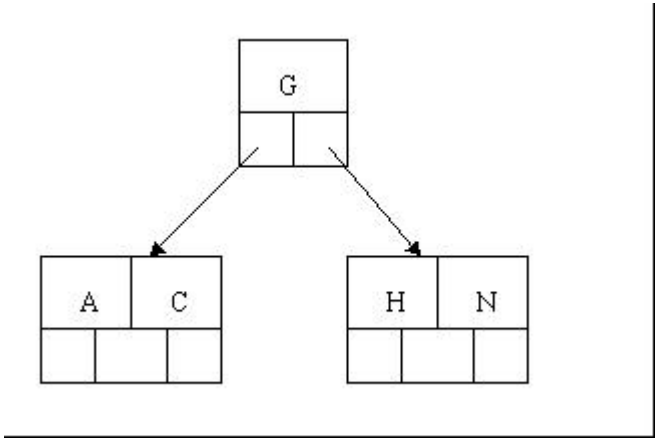
B树的插入示例 2/5

- 插入以下字符字母到一棵空的B 树中
  - 非根结点关键字数小了（小于2个）就合并
  - 大了（超过4个）就分裂）：

  C N G A H E K Q M F W L T Z D P R X Y S
  - ①首先，结点空间足够，4个字母插入相同的结点中：

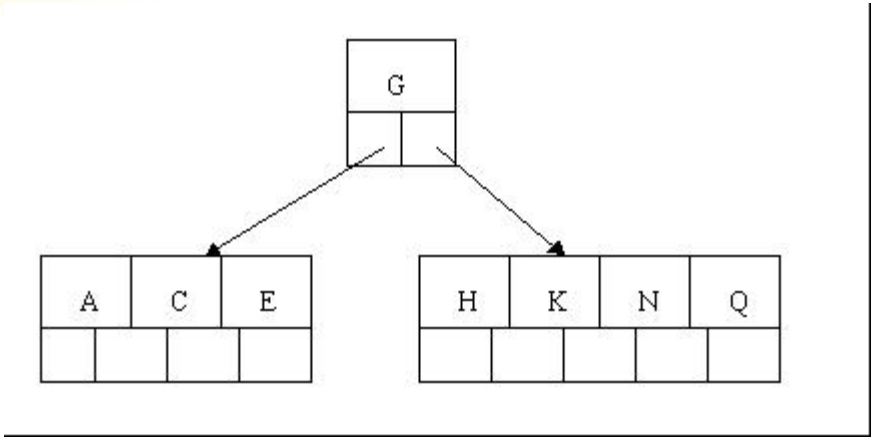


  - ②当咱们试着插入H时，结点发现空间不够，以致将其分裂成2个结点，移动中间元素G上移到新的根结点中，在实现过程中，咱们把A和C留在当前结点中，而H和N放置新的其右邻居结点中：

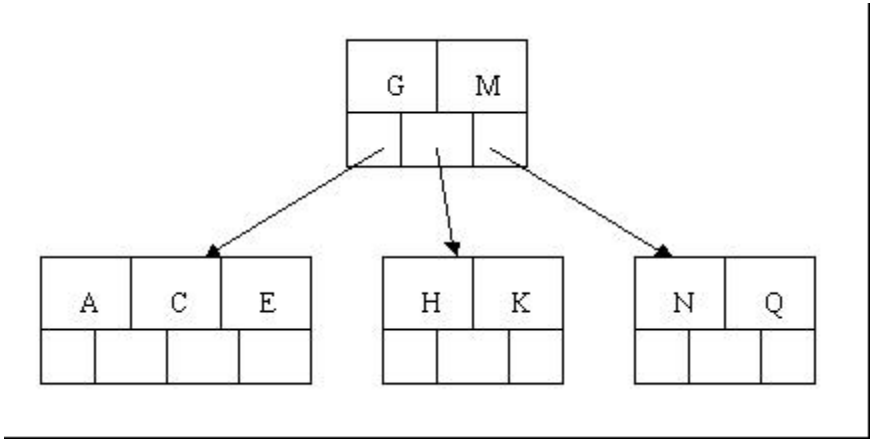


B树的插入示例 3/5

- C N G A H E K Q M F W L T Z D P R X Y S
  - ③当咱们插入E,K,Q时，不需要任何分裂操作：

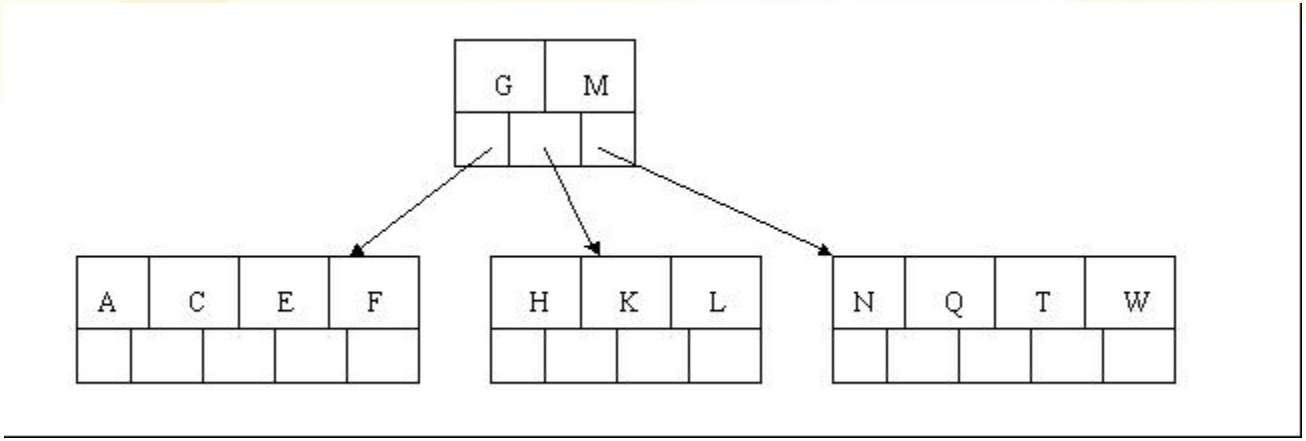


  - ④插入M需要一次分裂，注意M恰好是中间关键字元素，以致向上移到父节点中：

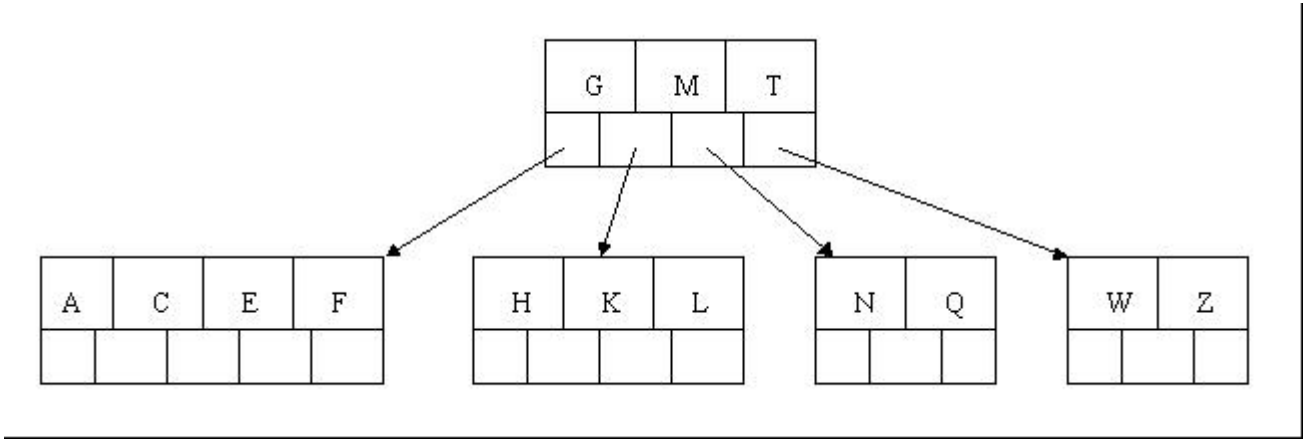


B树的插入示例 4/5

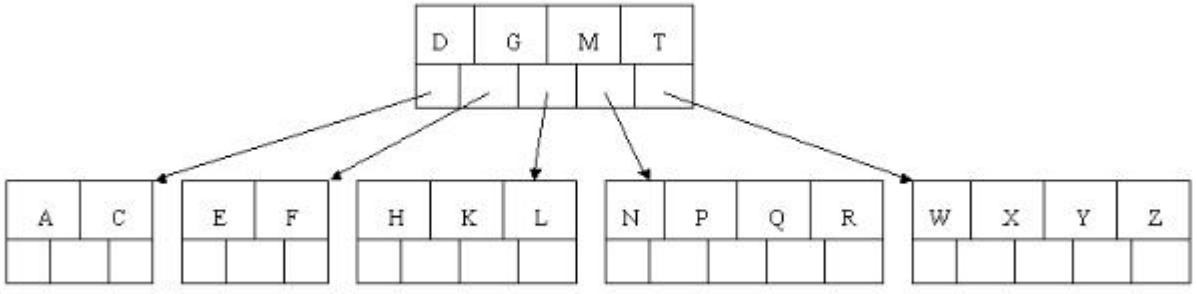
- C N G A H E K Q M F W L T Z D P R X Y S
  - ⑤插入F,W,L,T不需要任何分裂操作：



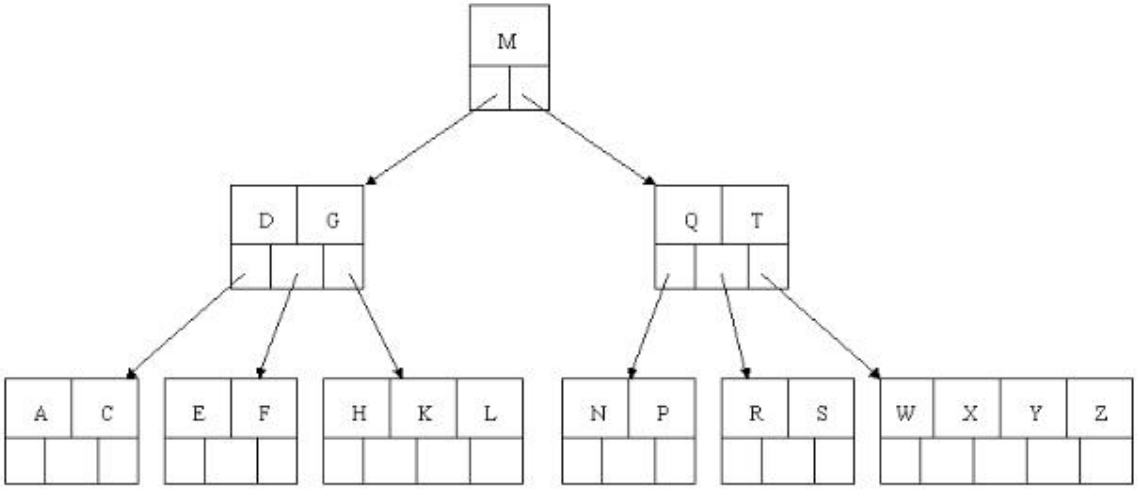
  - ⑥插入Z时，最右的叶子结点空间满了，需要进行分裂操作，中间元素T上移到父节点中，注意通过上移中间元素，树最终还是保持平衡，分裂结果的结点存在2个关键字元素：



- C N G A H E K Q M F W L T Z D P R X Y S
  - ⑦插入D时，导致最左边的叶子结点被分裂，D恰好也是中间元素，上移到父节点中，然后字母P,R,X,Y陆续插入不需要任何分裂操作（别忘了，任一结点最多可有4个关键字）：

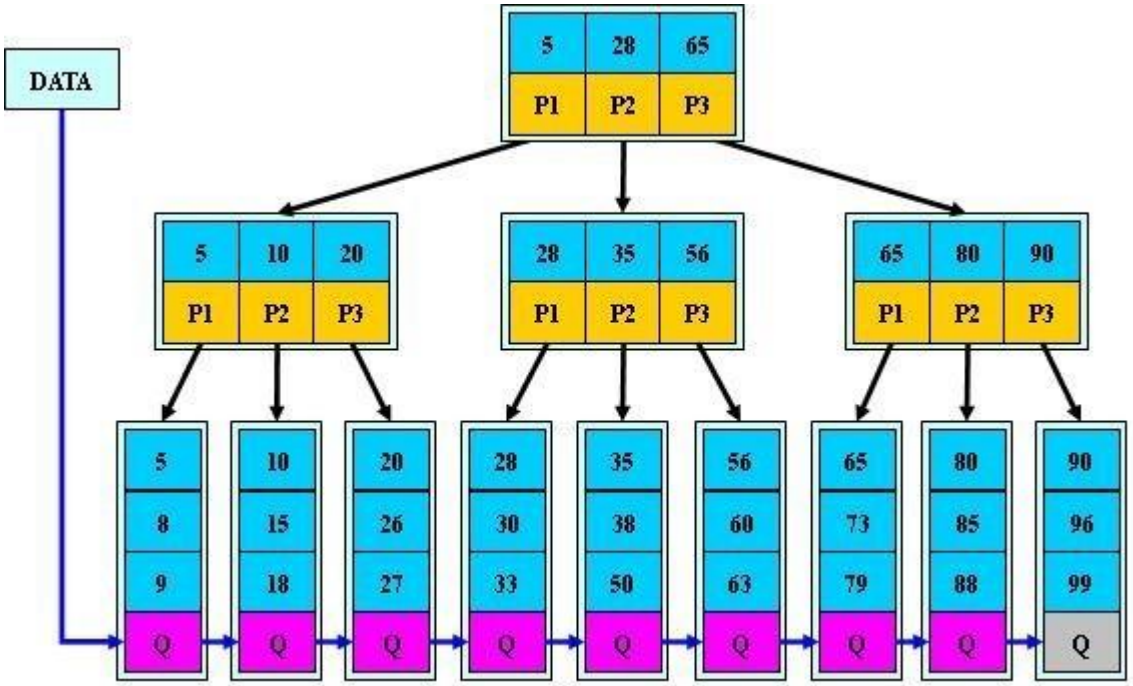


  - ⑧最后，当插入S时，含有N,P,Q,R的结点需要分裂，把中间元素Q上移到父节点中，但是问题来了，因为Q上移导致父结点 “D G M T” 也满了，所以也要进行分裂，将父节点中的中间元素M上移到新形成的根结点中，从而致使树的高度增加一层。



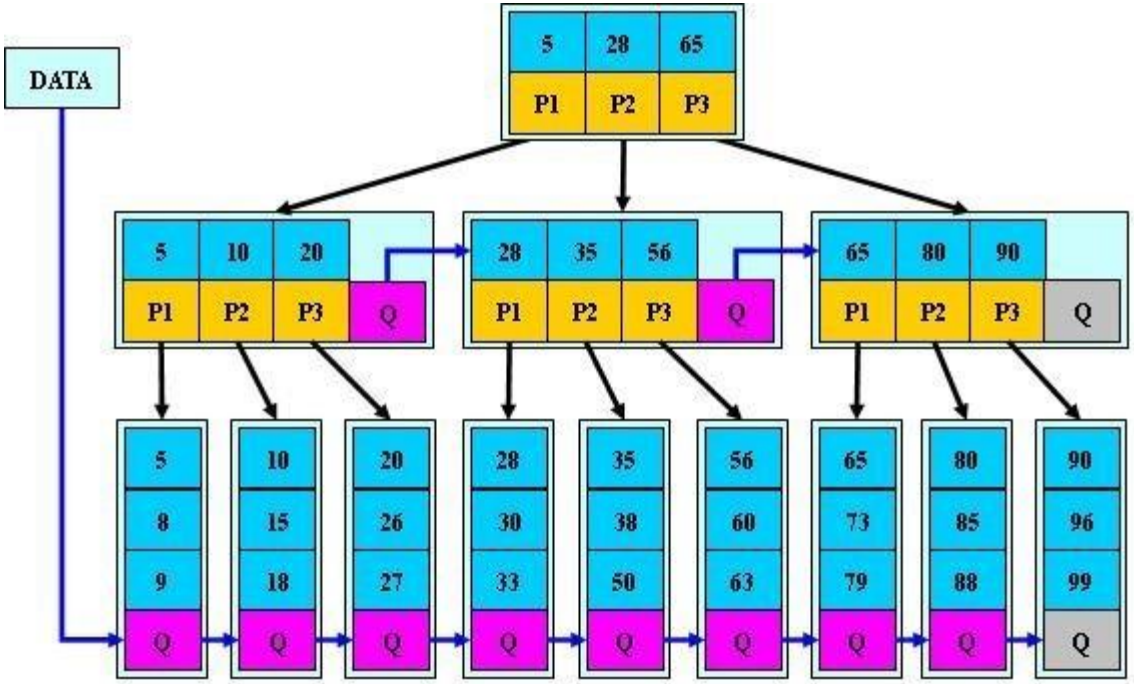
B+ 树

- 一棵m阶的B+树和m阶的B树的异同点在于：
  - 有n棵子树的结点中含有n-1 个关键字
  - 所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。 (而B 树的叶子节点并没有包括全部需要查找的信息)
  - 所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。(而B 树的非终节点也包含需要查找的有效信息)



B* 树

- B*-tree是B+-tree的变体
  - 在B+树的基础上，所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针)；
  - B*树中非根和非叶子结点再增加指向兄弟的指针；
  - B*树定义了非叶子结点关键字个数至少为(2/3)*M，即块的最低使用率为2/3（代替B+树的1/2）



B树、B+树、B*树的区别

- B树，B+树，B*树总结如下：
  - B树：有序数组+平衡多叉树；
  - B+树：有序数组链表+平衡多叉树；
  - B*树：一棵丰满的B+树。

- 总结：
  1. 理解了2-3-4树，便理解了B树
  2. 理解了B树，便理解了B+树
  3. 理解了B+树，便理解了B*树
  4. 当然，还有一种树：空间划分树 - R树

R 树的查找

- 假设我要查询广州市天河区天河城附近一公里的所有餐厅地址怎么办？
  - 打开地图（也就是整个R树），先选择国内还是国外（也就是根结点）；
  - 然后选择华南地区（对应第一层结点），选择广州市（对应第二层结点），
  - 再选择天河区（对应第三层结点），
  - 最后选择天河城所在的那个区域（对应叶子结点，存放有最小矩形），遍历所有在此区域内的结点，看是否满足我们的要求即可。

总结

- 有了二叉树，为何要有AVL平衡树？
- 有了AVL树，为何要有红黑树？
- 有了红黑树，为何要有B树？
- 有了B树，为何要有R树？

海量数据处理

十个密匙

- 哈希分治
- simhash算法
- 外排序
- MapReduce
- 多层划分

- Bitmap
- Bloom Filter

- Trie树
- 数据库
- 倒排索引

万丈高楼平地起

- Reb-Black tree
  - set/map（map同时拥有key和value，set的key就是value）
  - multiset/multimap（允许重复键值）
- hashtable
  - hashmap/hashset/
  - hash_multiset/hash_multimap（允许重复键值）
- 备注：
  - C++ 11标准命名了基于hash函数实现的unordered_set/ unordered_map / unordered_multiset / unordered_multimap
    - 相当于hash_set、hash_ma，和hash_multiset、hash_multimap。

分而治之/hash映射 + hash统计 + 堆/快速/归并排序

- 海量日志数据，提取出某日访问百度次数最多的那个IP
  - 分而治之/hash映射，比如模1000，把整个大文件映射为1000个小文件
  - hash_map(ip，value)来进行频率统计（hash_map对那1000个文件中的所有IP进行频率统计，然后依次找出1000个文件中频率最大的那个IP）
  - 再在这1000个最大的IP中，找出那个频率最大的IP

类似问题变形

- hash函数映射 + hashmap统计 + 堆排
  - 有100W个关键字，长度小于等于50字节。用高效的算法找出top10的热词，并对内存的占用不超过1MB。
  - 假设已有10w个敏感词，现给你50个单词，查询这50个单词中是否有敏感词。
  - 单机5G内存，磁盘200T的数据，分别为字符串，然后给定一个字符串，判断这200T数据里面有没有这个字符串，怎么做？
    - 如果查询次数会非常的多, 怎么预处理？

simHash的具体算法

- 分词
  - 给定一段语句：“CSDN博客结构之法算法之道的作者July”，分词后为：“CSDN/博客/结构/之/法/算法/之/道/的/作者/July”，然后为每个特征向量赋予权值：CSDN(4) 博客(5) 结构(3) 之(1) 法(2) 算法(3) 之(1) 道(2) 的(1) 作者(5) July(5)，其中括号里的数字代表这个单词在整条语句中的重要程度，数字越大代表越重要。

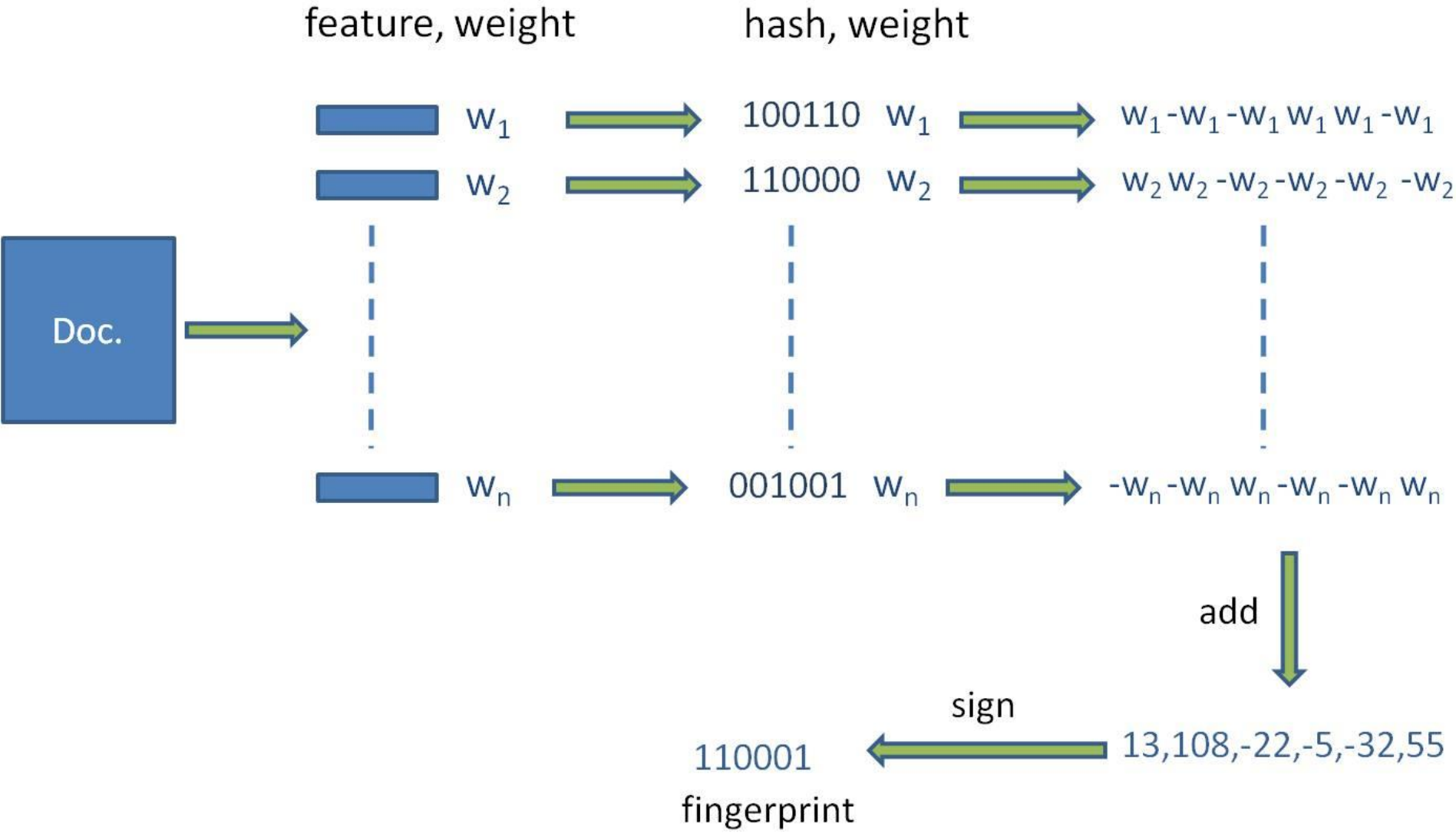
- hash
  - ○通过hash函数计算各个特征向量的hash值，hash值为二进制数01组成的n-bit签名。比如“CSDN”的hash值Hash(CSDN)为100101，“博客”的hash值Hash(博客)为“101011”。就这样，字符串就变成了一系列数字。

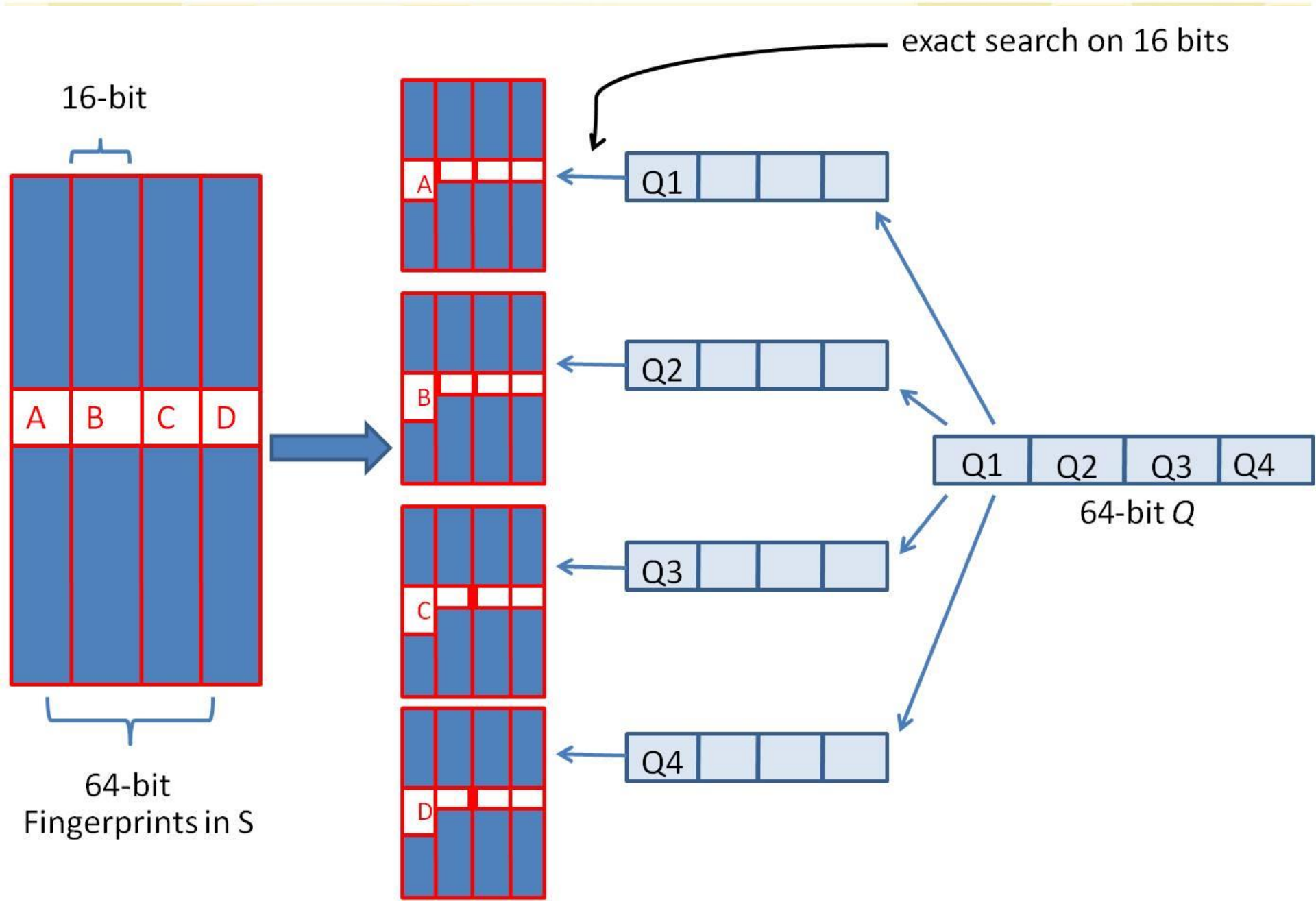
- 加权
  - W = Hash * weight。W(CSDN)=100101*4=4-4-44-44，W(博客)=101011*5=5 -5 5 -5 5 5。

- 合并
  - 将上述各个特征的加权结果累加，变成一个序列串。如：“4+5,-4+-5,-4+5,4+-5,-4+5,4+5”，得到“9,-9,1,-1,1”。

- 降维
  - 对于n位签名的累加结果，如果大于0则置1，否则置0，从而得到该语句的simhash值，最后我们便可以根据不同语句simhash的海明距离来判断它们的相似度。例如把上面计算出来的“9,-9,1,-1,1,9”降维，得到 “101011”，从而形成它们的simhash签名。

Simhash


feature, weight
hash, weight
Doc.
w1
w2
wn
100110 w1
110000 w2
001001 wn
w1 -w1 -w1 w1 w1 -w1
w2 w2 -w2 -w2 -w2 -w2
-wn -wn wn -wn -wn wn
add
sign
13,108,-22,-5,-32,55
110001
fingerprint


exact search on 16 bits
16-bit
A
B
C
D
64-bit
Fingerprints in S
A
B
C
D
Q1
Q2
Q3
Q4
Q1
Q2
Q3
Q4
64-bit Q

算法之上，继续深入！

- 数学
  - 微积分：《微积分概念发展史》
  - 概率论与数理统计：《数理统计学简史》
  - 数学史：《数学恩仇录》《数学与知识的探求》《古今数学思想》《素数之恋》
  - 《矩阵分析与应用》，清华张贤达著；
  - 《凸优化》，作者: Stephen Boyd / Lieven Vandenberghe，原著名: Convex Optimization；
- 搜索 & 推荐
- 自然语言处理
- 数据挖掘 & 机器学习
  - SVM
    - 《支持向量机导论》
    - 《支持向量机通俗导论（理解SVM的三层境界）》

参考资料与继续阅读

- 北京算法班的两个PPT
  - 周六班讲师邹博PPT
  - 周日班讲师曹博PPT
- 《算法导论》
  - 第十二章 二叉搜索树
  - 第十三章 红黑树
- 编程艺术github
  - https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/Readme.md
- 教你透彻了解红黑树
  - https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/08.01.md
- 从B树、B+树、B*树谈到R树
  - http://blog.csdn.net/v_july_v/article/details/6530142

thank you，any question?

contact me
微博：@研究者July