20140603
————————————————————————
ADT是抽象数据类型，为一组数据模型，加上一组操作，不涉及具体的储存方式，就像是用户使用的产品（黑箱），只考虑抽象层面

DS是数据接收，是根据特定语言实现ADT的算法，设计复杂度和各种具体机制。

（计算幂次方的方法：
pow(a,b)
we can devide b into binary type and pow(a,2^n) = pow(a,2^(n-1))
so we can do it in O(log2 n), each time rotate 1 time.
）

vector 向量数据结构，可以对不同数据类型操作，并且封装了许多操作，比如remove(r),sort(),search(e)无序去重,deduplicate(),uniquify()有序去重,traverse()遍历

search(n) 返回不大于n的元素的最后的rank
put(a,b) 将rank为a的数改为b

平均复杂度：根据数据结构个操作出现概率分布进行加权平均，每个操作作为独立事件，割裂了相关性与连贯性

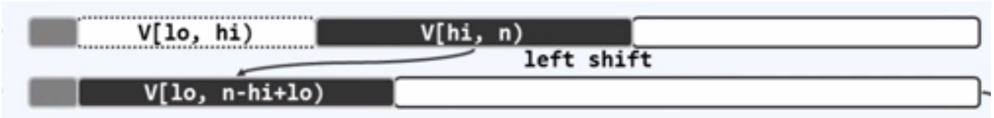
分摊复杂度：将数据结构连续地实施足够多的操作，所需总体成本分摊至单次操作。对一系列操作整体考量，更加精确。

vector 插入算法：
从固定插入位置开始，先扩容，然后从rank = n downto 固定位置k
vector[n] = vector[n-1]逐个后移
最后执行插入。

vector 删除：
区间删除：
删除从lo 到 hi 的区间

则执行如下：
_elem[lo++] = elem[hi++]

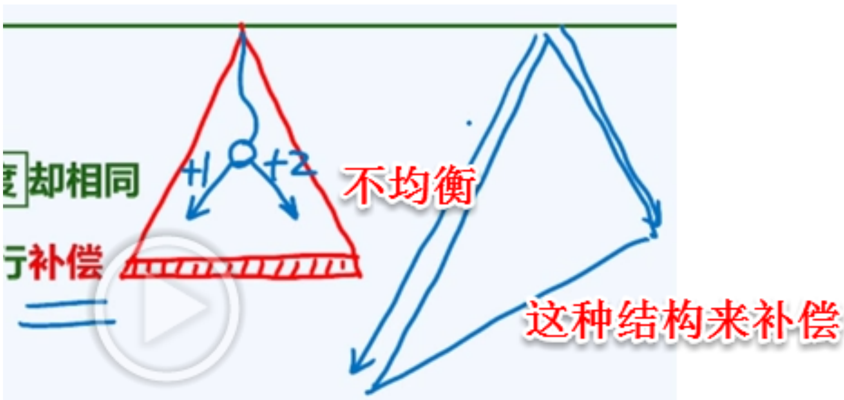
即从前向后逐次移位
不能更改次序，因为如果要删除的区间与想向前移动的区间有重复区间，从后向前的话会覆盖掉重复区间的部分。



有序相向量的唯一化
低效版：
有序向量，重复的元素构成一个区间，因此每个区间保留一个即可。
于是可以从前向后依次检查来逐个删除相同元素
O(N^2)

高效版：
成批删除雷同元素，将他们视为一个整体。
i = j = 0
如果二者不相等，则_elem[++i] = _elem[++j]，相当于忽略了所有重复元素（没有直接显示调用删除）。

Fabonacci查找法：
由于二分查找在大于、小于和等于的情况下比较次数不同，所以造成内部比较次数的不平均，因此为了平衡比较和查找次数，可以构建一个看起来不平衡的数列来平衡比较次数。这就是Fabonacci查找法




```
while(lo < hi)
{
    while (hi - ol < fib.get()) fib.previous();
    rank mi = lo + fib.get() - 1;
    if (e < A[mi]) hi = mi;
    else if (A[mi] < e) lo = mi + 1;
```

    else return mi;
}
return -1;

证明：运用递推*

——————————————————————————————
二分查找的改进版：
为了隐藏比较次数，只比较一次：
while(1 < hi - lo)
{
rank mi = (lo + hi) >> 1 ;
(e < A[mi]) ? hi = mi : lo = mi; (不再是 mi + 1)
}出口时 hi = lo + 1
return (e == A[lo]) ? lo : -1;
只是平均情况减少，但最好情况增加。

最优化版：如果不在其中，还可返回最近的小于此元素的坐标

```
template <typename T> static Rank binSearch(T* A, T const& e, Rank lo, Rank hi) {
    while (lo < hi) { //不变性：A[0, lo) <= e < A[hi, n)
        Rank mi = (lo + hi) >> 1; //以中点为轴点，经比较后确定深入
        (e < A[mi]) ? hi = mi : lo = mi + 1; //[lo, mi)或(mi, hi)
    } //出口时，A[lo = hi]为大于e的最小元素
    return --lo; //故lo - 1即不大于e的元素的最大秩
}
```

冒泡排序改进：可以记录前面是否有逆序对，如果没有就证明不用再排序了
再改进：可以增加一个last，也就是最后一个逆序对的位置，然后以后就只用排开始到last位置的元素了。

——————————————————————————————

第三章：列表
双向列表：两个哨兵：header trailer（-1 和 n）
列表的查找：没有高效方法，只能按位置依次向后找。

列表的排序：
选择排序

一直找当前最大的。
以及 **插入排序**
一次找一个，然后进行对比排序。
逆序对个数：可以决定插入排序的复杂度
设想，当一个元素前面有n个比他大的元素
则在插入排序的时候，会从后向前比较n次。
一次这时候设总逆序对为I，复杂度就为O(I+N)

第四章：队列与栈

应用：进制转换

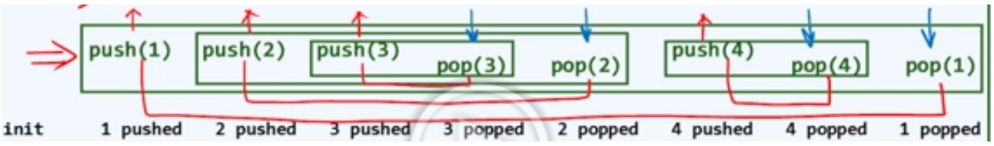
```
void convert( Stack<char> &S, __int64 n, int base ) {
    static char digit[] = //新进制下的数位符号，可视base取值范围适当扩充
        { '0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F' };
    while (n > 0) { //由低到高，逐一计算出新进制下的各数位
        S.push( digit[n % base] ); //余数（对应的数位）入栈
        n /= base; //n更新为其对base的除商
    }
}
```

括号匹配

栈混洗：将A栈中元素移动到B栈中，通过中间栈S，规定：只能移动到S再全部移动到B
那么有多少种方法呢？打个比方，1先入栈，然后又入了一些，则此时让一些栈，则当1出栈时，假设前面已经进入了k个元素，则后面n-k个只能排列在B的后n-k个位置上
所以这样推算，也就得到了递推关系：S(N) = ∑S(K-1)S(N-K)
这样一个递推数叫做Catalan数，其结果为 (2n)!/(n!(n+1)!)
推导过程详见：
http://blog.sina.com.cn/s/blog_497689ad01000azu.html

判断是否是一个栈混洗
O（n)的算法：直接借助A,B,S模拟混洗过程，运用贪心的原则，如果每次S.pop()时候已经空了或者需要弹出的元素不在S最顶端，则是非法的。

栈混洗与括号的联系：
一个栈混洗的过程可以表示为一个合法的括号表达式：



每次push看作是(，pop看作是)
由此可以预见：有多少种栈混洗结果，n对括号可以构成的表达式（合法）也就有多少种。

中缀表达式求值：
将数字压入栈，并且当遇到运算符时，先判断之前是否有运算符的优先级比当前运算符更大，如果是则首先进行上一个运算符的运算，然后递推进行判断再前一个运算符（运算级相同也要运算），直到不成立，就接着压栈。直到最后到头（压尽元素）。
引入两个栈。

优先级操作符的比较：制表

```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
   //            |-------------- 当前运算符 --------------|
   //              +    -    *    /    ^    !    (    )    \0
   /* --  + */   '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /*  |  - */   '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /* 栈  * */   '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 顶  / */   '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 运  ^ */   '>', '>', '>', '>', '>', '<', '<', '>', '>',
   /* 算  ! */   '>', '>', '>', '>', '>', '>', ' ', '>', '>',
   /* 符  ( */   '<', '<', '<', '<', '<', '<', '<', '=', ' ',
   /*  |  ) */   ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
   /* -- \0 */   '<', '<', '<', '<', '<', '<', '<', ' ', '='
```

遇到小于则入栈，遇到大于则计算（栈顶元素 < > = 当前元素）

》》》》》》》》》》》》》》》》》》
部分习题笔记：
1.归并排序的算法复杂度证明：O(nlogn)

教材中已针对该算法，给出了如下边界条件及递推方程：

$$T(1) = O(1)$$
$$T(n) = 2 \times T(n/2) + O(n)$$

或等价地

$$T(n)/n = T(n/2)/(n/2) + O(1)$$

以下若令：

$$S(n) = T(n)/n$$

则有：

$$\begin{aligned} S(1) &= O(1) \\ S(n) &= S(n/2) + O(1) \\ &= S(n/4) + O(2) \\ &= \ldots \\ &= S(n/2^k) + O(k) \\ &= O(\log n) \end{aligned}$$

于是有：

$$\begin{aligned} T(n) &= n \cdot S(n) \\ &= O(n\log n) \end{aligned}$$

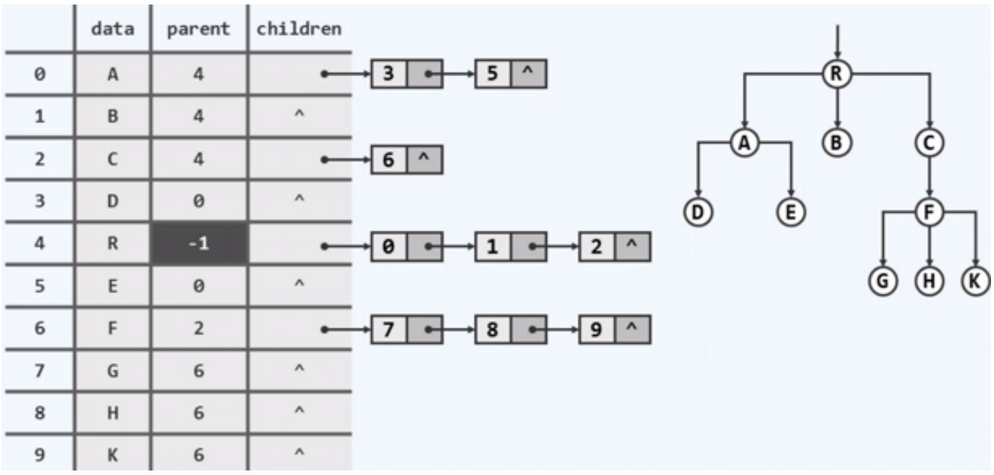
归并排序的边界条件及递推方程，在算法复杂度分析中非常典型，以上解法也极具有代表性。因此，读者不妨记住这一递推模式，并在今后作为基本结论直接应用。

2.归并排序的优化：
对于两段已经排好顺序并且合起来也已经有序的情况，我们只需要增加一条语句：
if (a[mi-1] <= a[mi]) merge(lo,mi,hi);

3.链表访问的优化：
由于对数据结构的操作往往都限定于一个较小的子集，所以可以将每次查找的链表元素移到首元素。在这种情况下，经常被访问的元素将集中在前端，大大提高访问效率。

第四章 树

1 树的表示方法：
可以将各个节点组成数组结构，包含孩子节点数据集与父节点的标号，如果有某个节点孩子节点，那么此节点里面的孩子节点数据集（可以为列表或者向量）就存储又大到小的孩子。再存储此节点的父亲节点。这时候向下查找与孩子的数目线性相关，向上查找与深度有关。



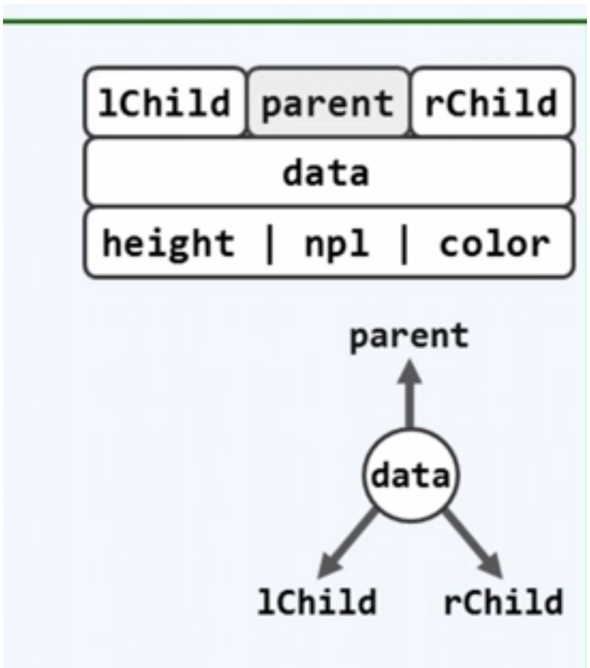
	data	parent	children
0	A	4	3 → 5 ^
1	B	4	^
2	C	4	6 ^
3	D	0	^
4	R	-1	0 → 1 → 2 ^
5	E	0	^
6	F	2	7 → 8 → 9 ^
7	G	6	^
8	H	6	^
9	K	6	^

改进：每个节点主需要记录两个引用：纵向的firstchild以及横向的nextsibling（相邻兄弟），这时候储存结构的规整性大大增加。

2 二叉树：
真二叉树：每个节点都是2个度或者0个度（如果不够则在下面补满2个孩子）。更加规整（实际操作其实是假想的，并不存在）。

如何用二叉树来 描述多叉树
将长子作为左节点，次子作为右结点。

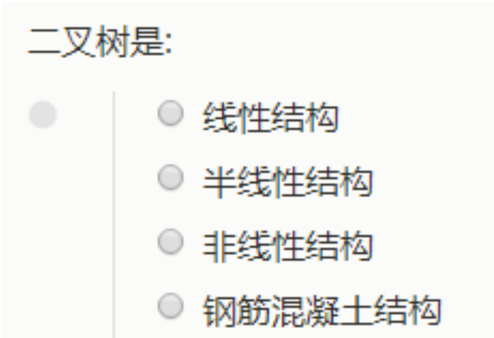
二叉树的表示：
binode类：表示树的结点

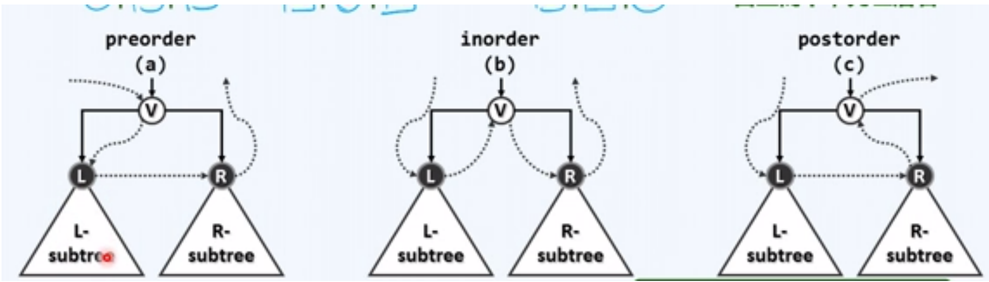


父亲、左右孩子、高度、颜色（红黑树）、npl（左式堆）

父亲、孩子均为引用。

树形结构最重要的就是遍历
定义一个树类，里面有树根（节点类）、树的高度、规模、判空函数、各种遍历方法、子树的删除插入与分离等。

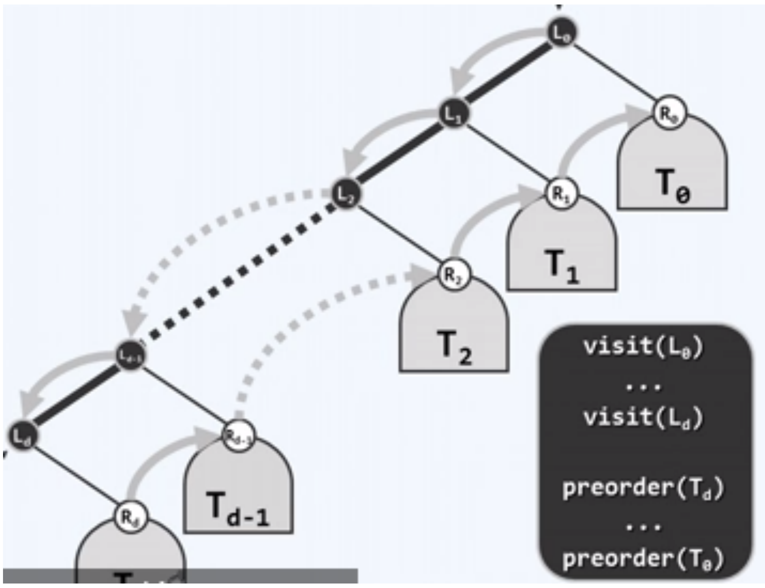
前序遍历：
。。。。。。。


VLR顺序
中序：LVR
后序：LRV
即V（父亲）的顺序在哪里就是什么序遍历


先序遍历：
visit(x->data);
traverse(x->lchild，visit);
traverse(x->rchild，visit);

O(N)复杂度
但是递归在运行栈中占用空间很大。
所以非常有必要从递归改为迭代。
注意：尾递归
化解为迭代：
利用栈的方法，既然需要先处理左边，就先把右边压入栈

新的构思：
先便利左侧链，在自下而上遍历右子树。



每次访问左孩子，然后将右孩子一次压入栈。

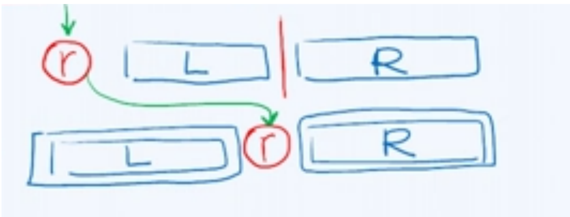
想法：其实也可以先储存A,B表示当前层的两个节点，对A（左）节点判断有无孩子，如果有，则访问，并且A = LA, B = RA; 反之访问B结点，并且A = LB, B = RB;如果都没有，B为父节点的右兄弟节点，然后再次寻找。（自编）

对于**中序遍历**：
同样构建一个栈，存储右子树，
不同的是，左子树一直遍历，同时一直压入栈，直到尽头，取节点值
这时候再在返回后再pop弹出栈顶节点的值，然后将此节点右结点变为活跃节点继续进行左子树遍历。

对于**后序遍历**：
首先将根压栈（因为根没有右结点），然后依次：如果有左节点，就把右结点压栈，再将左节点，如果没有，就把右结点压栈，直到为空；然后对当前节点进行pop，如果pop！=父节点，那么就代表pop的是右结点，就向右结点下方继续遍历，如果是，就直接pop然后输出；

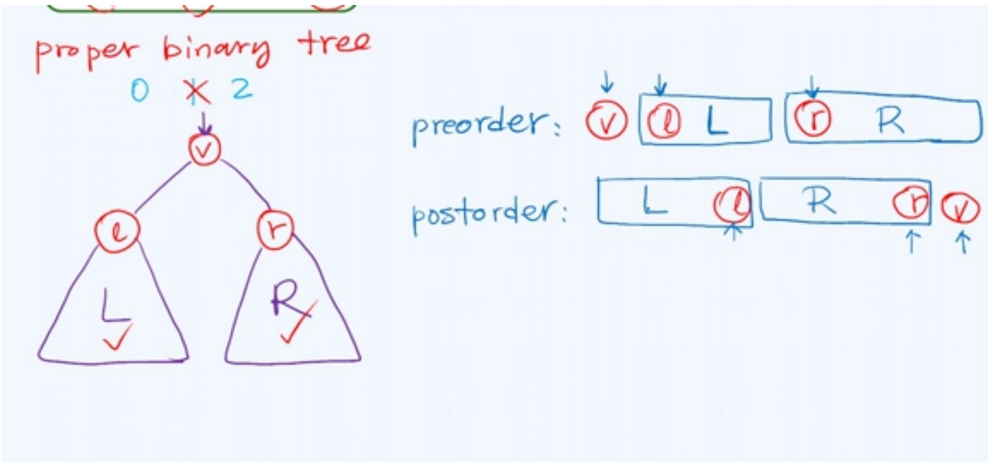
我的方法：先向左搜索到头，然后一路push，最后的直接输出，将当前节点改为父节点的右结点，在一路push，最后push后子节点已经为空，返回。输出。然后再将top（此时出来的为父节点）的节点与刚刚输出的节点比较，如果相同，证明父节点的所有子节点已经遍历完了，就再pop输出父节点，然后继续push；如果不相同，证明刚刚输出的是左端，右结点还没有遍历，遍历右结点。直到栈空。

树的重构：
中序遍历+先序/后序遍历 任意一种 便可以忠实还原原来的树形结构。
（分而治之，找到左右子树）



证法：归纳假设

对于真二叉树，可以使用先序+后序还原。（分而治之）



测试题：
并查集：

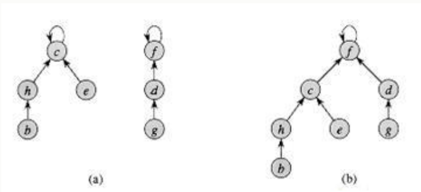
并查集是一种用于表示不相交集合的数据结构，支持以下操作：

- Union(x, y): 将元素x和y所在的集合合并
- Find(x): 返回元素x所在集合（实际上是返回该集合的一个代表元）

一种基本的实现是将每一个集合中的元素组织成一棵有根树，集合中的元素即树中的节点，选取树根为该集合的代表元，而整个并查集就是由若干棵树组成的森林。接口实现的方法是：

- Union(x, y): 将x所在树的根节点的父亲设为y所在树的根节点，从而将它们合并成一棵树
- Find(x): 返回节点x所在树的根节点。

例子：下图中的并查集原先有两棵表示集合的树{c,h,b,e}和{f,d,g}，调用Union(h, f)后得到了右边的树，如果此时再调用Find(e)会返回f。



第六章：图
第一部分 图的表示
邻接矩阵
邻接表：每个顶点有一个链表，链表储存了他的相邻接点（出度）
也可用数组的方式来表示，一个数组A[]表示点，A[1]表示A的第一个相邻接点，一个数组B[]表示A[]中每个点的临界点个数，比如B[N]表示A[N]的临界点个数，然后提取A[N]相邻接点信息时可以这样做：
IF (B[I++] != -1) A[B[I]] ......

第二部分：BFS广度优先搜索
用一个队列来维护，与二叉树层次遍历相似，先入队的先出队操作，对其进行相邻边搜索，并且对未标记的邻接点标记。
*BFS搜索从S->A点的路径是最短路径（等权图下）

第三部分：DFS深度优先搜索
任意找一个点，随机找一个邻接点然后访问，随后递归地进行访问（即随机访问目前点的下一个相邻点），知道访问不能（没有相邻点或者所有相邻点都已被访问，做了标记），然后一步一步退回，直到当前节点还有其他相邻点未被访问，那么继续向内扩展。

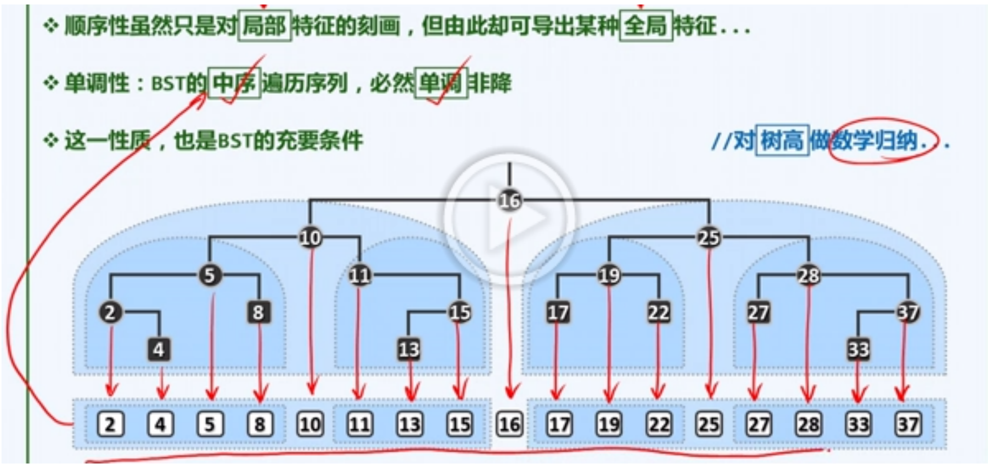
第七章：二叉搜索树 (BST)

优点：既是列表的列表（二叉树），又体现了向量的优点。

循关键码访问：对数据项的访问通过关键码，关键码之间可以比较大小(将词条中比较操作重载）。

有序性：对一个节点，左子树每个节点都不比他关键码大，对称地，右边子树每个元素不都比他的关键码小。

对BST做中序遍历，必然是单调非降序列。
只需要考察所有节点的垂直投影：



只要垂直投影单调非降，则满足BST

接口：可以直接从bintree派生而来
只需要派生查找、插入、删除三个接口。

BST的查找：
查找类似于二分查找，对于关键码，如果大于则走右孩子，小于则走左孩子，直到找到或者没找到。此时返回一个节点，并且将此节点的父节点也存起来，方便后面的插入操作。o(logn)

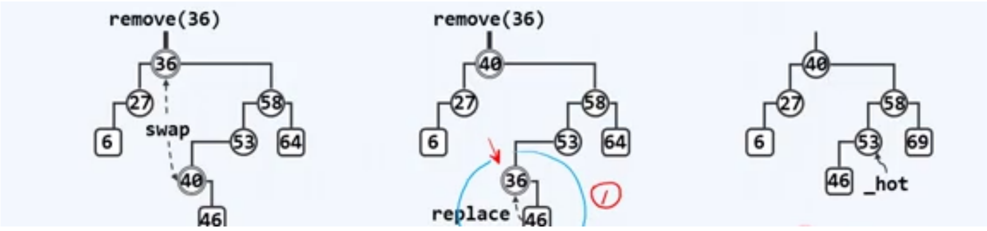
BST的插入：

BST的删除：
1.最简单的情况：没有左子树或者右子树
此时可以直接将左节点或者右结点与要删除的节点交换，对于没有左右结点的节点，可以通过配置哨兵的方式来同样以此实现。

2.复杂的情况：双结点

对于双节点，可以先找到这个节点的直接后继（即按照中序遍历向后第一个节点），它的直接后继一定是比它大的最小元素，因此可以与之进行交换，交换后待删除结点一定是没有节点或者只有右结点的情况，就可以转到情况1 进行操作。

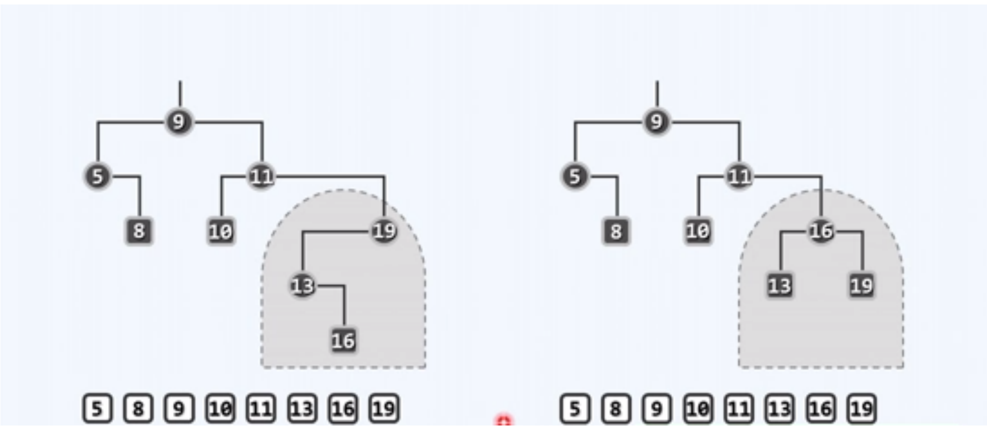
反思：注意不同结构、不同思路之间的融会贯通以及转换。
反思2: 任何复杂的问题都是由简到繁进行解答的，而且繁琐的方面一般都是由简单的解法组合变换而成的。因此要从简单入手，逐步解决。



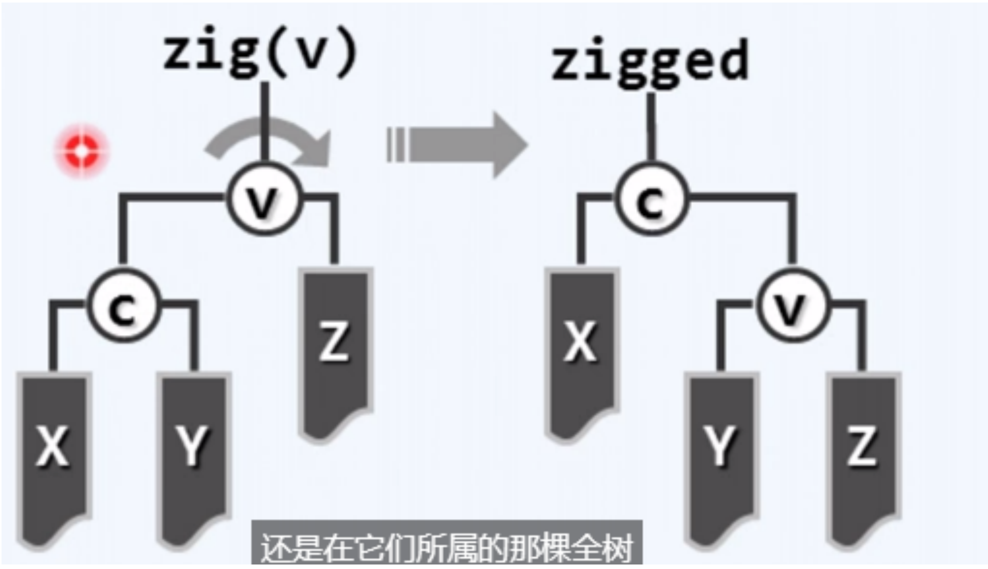
平衡性与等价性测量：
对于n个互异的数，一共有catelan(N)种情况，其高度平均为根号N

如何达到理想平衡（高度最低）？
兄弟子树高度越接近，高度越低。
引理：由n个结点组成的二叉树，其高度不低于log2N（理想平衡）
高度渐进地不超过logn——适度平衡，渐进意义上的理想高度——平衡二叉搜索树（BBST）。

等价BST：不同的二叉搜索树可以对应相同的中序遍历序列。
如何等价？两个原则：上下可变，左右不乱。



等价变换：
第一类：**ZIG** （顺时针）



第二类：**ZAG**（逆时针）
相反操作

AVL树
对于每个节点左右子树高度不相差1的树称为~

平衡因子：对于一个节点，它的左子树减去右子树的高度。

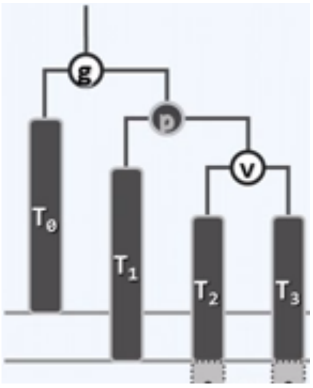
高度为h的AVL树，至少包含
S(h) = fib(h+3)-1个节点

证明：递推
S(h) = S(h - 1) + S(h - 2) + 1
S(h) + 1 = [S(h - 1) + 1] + [S(h - 2) + 1]
以上是fabonacci式

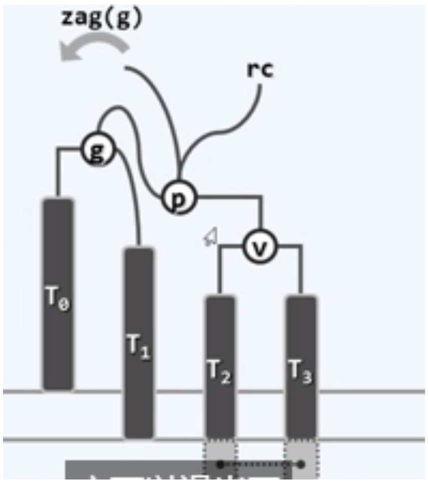
接口：
平衡因子：左子树-右子树高度
定义理想平衡以及AVL平衡
由BST派生接口：查找
重写：插入与删除

插入与删除操作：
对于插入，最多有O(logn)个节点发生变化。
而对于删除，最多有O(1)节点发生变化。

如何实现？zig-zag旋转
首先：遇到了插入后平衡因子绝对值大于1的情况。



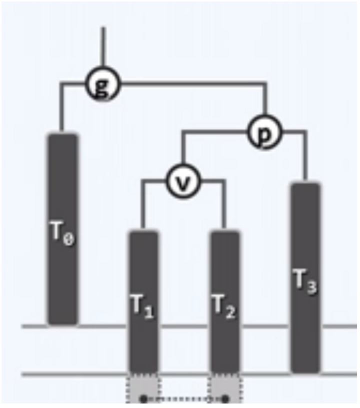
这时候进行逆时针旋转，找哪一个点呢？
我们知道，插入引起的失衡是通过引起祖先一代一代失衡而使得整树失衡的过程，所以只要根绝在源头，也就是最先失衡的（最低级的）祖先，把它调整为平衡，整个树就又恢复平衡了。
因此需要将它与引起它失衡的子节点交换。
例如此图：P和V交换
具体操作：



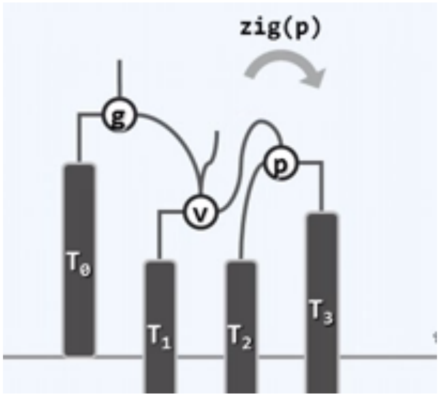
用一个临时引用rc指向p，然后将p的左儿子交给g，之后gp交换，p替代g成为祖先。

这种交换成为 zigzig
对应zagzag为顺时针交换。

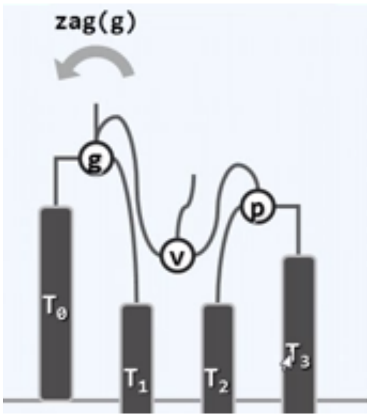
对于zigzag，也就是之字形交换（还有zagzig）



麻烦一些。
具体方法：先zig旋转



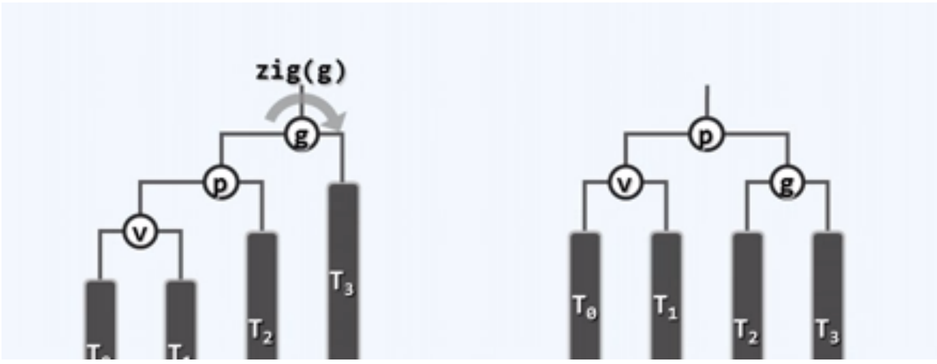
再次zag旋转



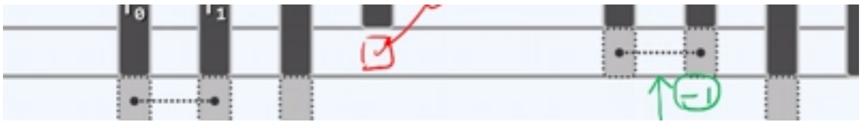
也就是说，先平衡小的，再平衡大的。
如果左子树高度太大，便使用zig
如果右子树高度太大，便使用zag平衡。

操作：先插入（BST），然后依次向上查找祖先，找到第一个失衡祖先，进行旋转调整。

删除算法：

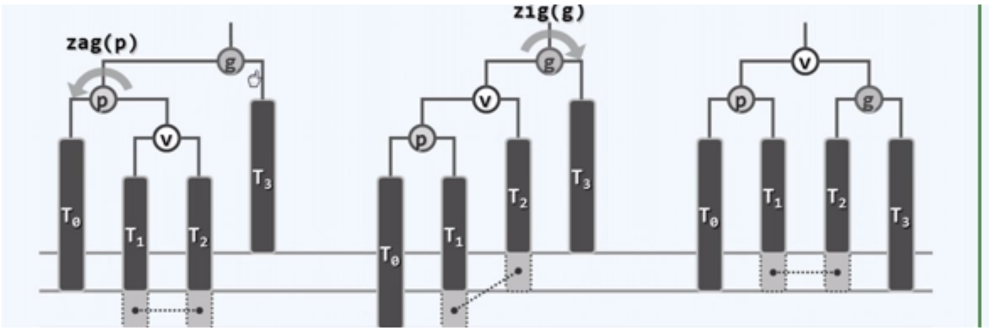


依然进行旋转。
此时的情况是，祖孙三代都是左或右孩子比较多。
当旋转过后，如果T2 存在孩子，这时调整后的新树高度不变



但是如果T2 不存在孩子，则调整后子树的高度减少1，可能会造成失衡。
因此这时候可能最多需要做O(logn) 次调整。

当三代节点更多孩子的个数不是朝一个方向排列，也就是按照之字形排列，

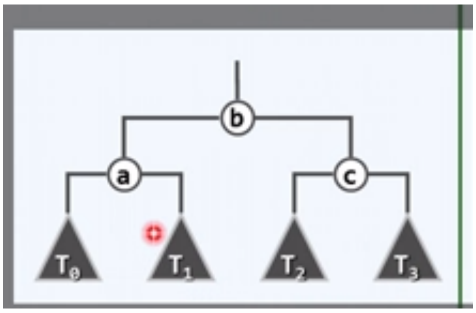


那么就需要先转换为一边倒的类型I，在进行节点旋转。但此时高度减少了1.

——————————————————————————
真正的操作，其实大可不必按照理解的思路进行旋转
可以类比一个魔方，如果是组装工人，不需要根据规则旋转，而是直接组装。

将需要旋转的祖孙三代，g p v
按照中序遍历重新命名 a < b < c
然后对于g p v 的孩子们（不超过四个）
按照中序遍历次序，从新命名
T0 < T1 < T2 < T3

于是可以重新组装



3+4重构。

对于重命名的规则，可以分四种情况分别列举。

至此，删除与插入操作（包括子操作旋转操作）迎刃而解。

AVL树有什么缺点呢?

需要单独记录平衡因子，需要改造元素结构并且额外封装。

单次动态调整之后，全书的拓扑结构变化量可高达Ω(logn)

数据结构（下）

第八章 高级搜索树

一 伸展树

对于一些元素，很可能要经常对其进行访问，或者对与其相邻的一些元素经常访问，因此可以进行优化。

（与链表进行比较： 某些链表的元素会被集中访问，因此每次访问一个元素可以将之移动到最前端，可以提高访问效率。）

回到BST，可以将某一段时间内经常访问的元素移动到树根的位置。（利用zig 与 zag 旋转）

但这样会遇到只有一个狭长分支的最坏情况。

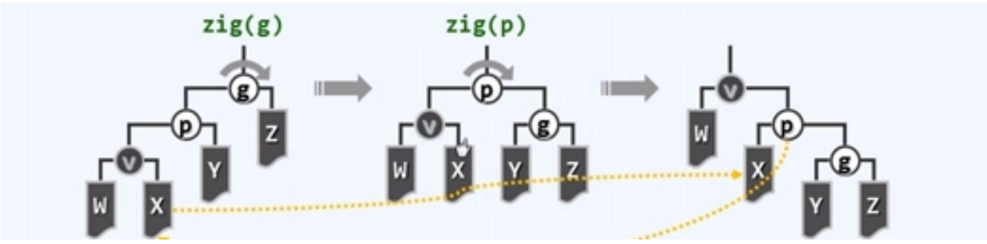
怎么优化呢?

通过zig zag旋转（AVL树的双旋调整）

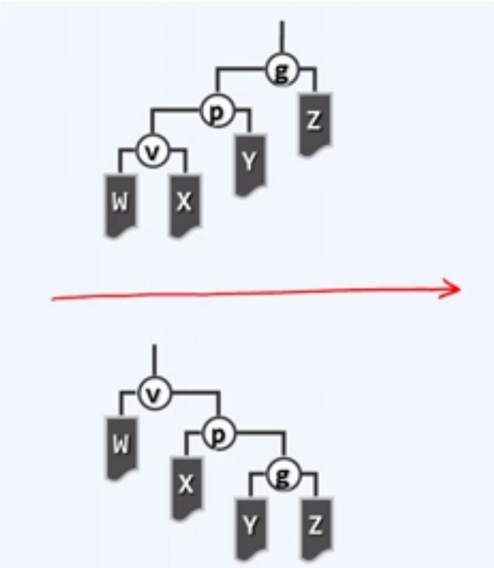
通过zig-zag 或者 zag-zag

（相当于不直接旋转所寻找的那个节点在的一级，而是从他的父节点下手旋转）

当需要zigzig或者zagzag（也就是在分支在一侧的情况）
旋转与avl树相同。

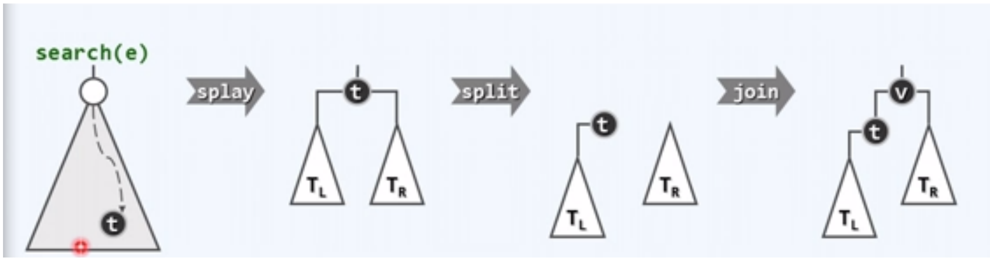


代码实现:
splay函数:
分情况：1，当前孩子为右孩子还是做孩子
　　　　2， 当前孩子的父亲为右孩子还是左孩子
分情况进行zig 、 zag 旋转。
旋转细节：同avl，可以直接进行拆开重接，而非生搬硬套模拟旋转过程



注：还需要考虑没有祖父节点的情况（即只需要直接旋转）
查找函数：查找此节点存在否，若存在，则返回splay后的节点，否则splay与要查找值相似的节点并返回其接口。

插入:
　首先调用search（即首先将相似节点splay到了根节点，然后进行插入）因此可以直接在根节点处进行插入。



删除算法：

依然是调用了search()，所以根节点必为树根

因此直接在树根处进行删除，然后选择左右任意子树中最小的节点作为根节点。

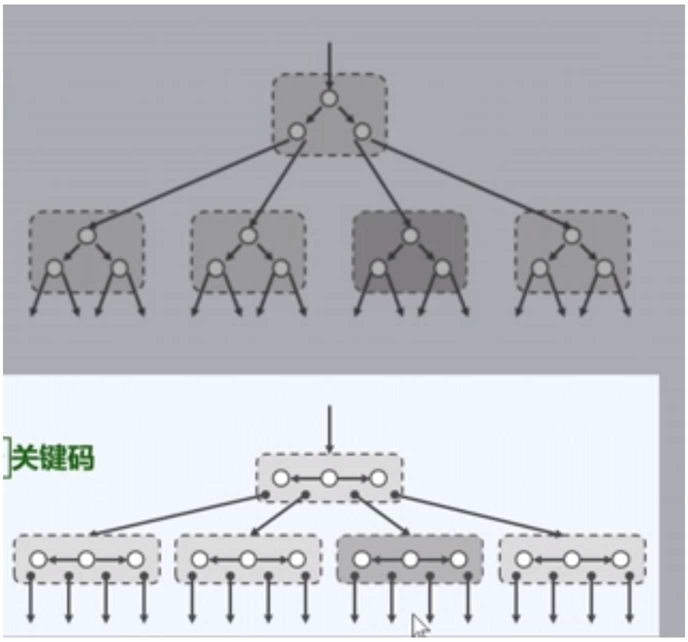
B树：

每个节点未必只有两个分差

底层节点的深度相同

更宽、更矮。

超级节点的概念：多个关键码位于一个节点。

例如：

为什么如此（这样与上图不是等价吗？）

优点：针对外部查找大大提高了输入输出（I/O）效率

每次访问一个关键码可以读出一组数据，相比于一个节点一个数据，这样的效率提高了很多。

例如有1G的数据，单次查找一个数据，

如果二叉树，则需要log(2,10^9)=30 次，而如果以关键节点有256个关键码

的B树，只需要查找$\log(256,10^9)<4$ 次。

B树的阶:

阶数=路数

B树的外部节点（也就是叶子节点的下一个节点（空节点））深度相同，且即B树的高度。

B树的命名:

内部节点：不超过m-1个关键码

不超过m个分支

内部节点的分支数:

对于树根：$2<= n+1$

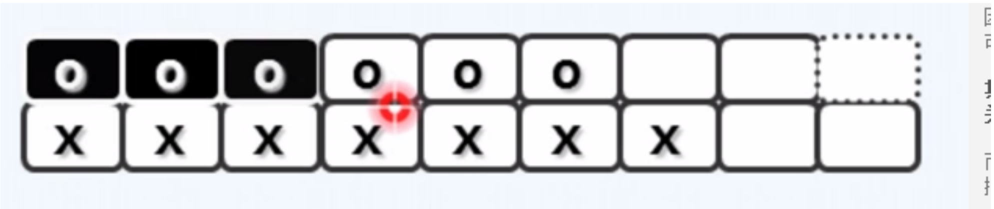
其余节点：（取上界）$m/2 <= n+1$

B树每个节点可以视为线性序列

那么怎样表示呢?

我们可以用向量来表示

比如:



第一个向量存放n个关键码

第二个存放相对应的n+1个引用。

含有N个关键码的M阶B树，最大高度为

$h<= 1+\log(m/2)[(n+1)/2] = O(\log N)$

提示：内部节点尽可能瘦（节点数位m/2上界）而根节点可以只分两支。

最小高度：$h >= \log(n)(N+1) = \Omega(\log mN)$

B树的插入算法:

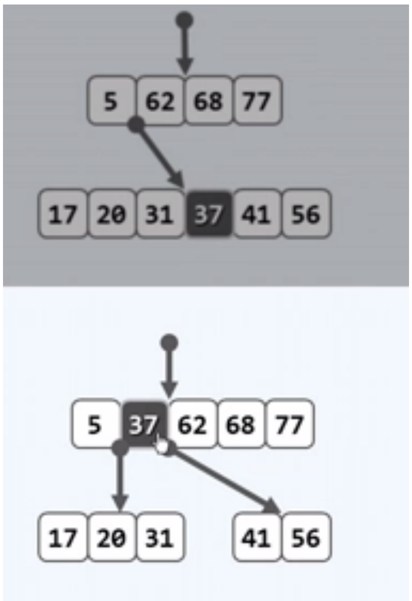
先查找，然后返回一个最近不大于插入值的关键码位置

然后关键码+1，分支也对应加一。

如果违反了约定，即插入后关键码数目上溢，则需要处理。（分裂）

此时从此组关键码中位数分裂，并将中位数关键码向上追寻，并且插入至左上两关键码的中间。

如图：



如果父节点也上溢，那么再次操作。
当然，根节点也需要有相应的上溢处理。
如果根节点也发生上溢，则也取出中位数，使之成为新的根节点。

导致B树增高（此时为两个分支）

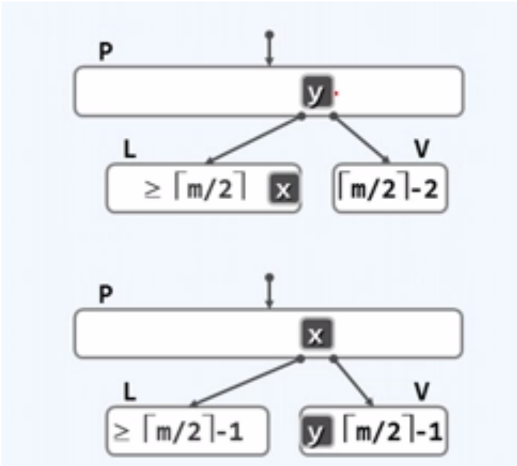
————————————————————————

B树的删除

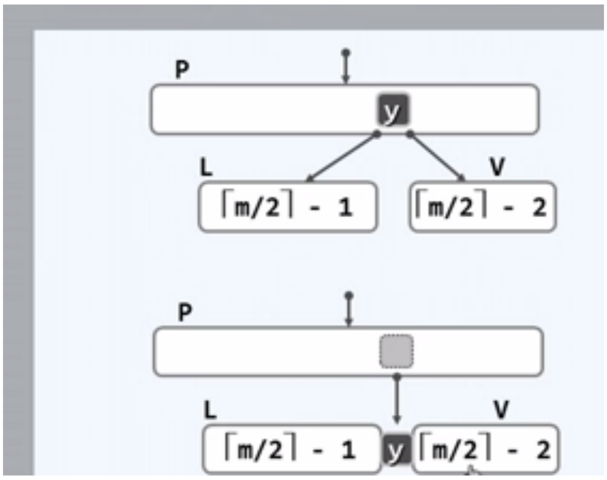
首先进行查找。如果找到，如果它不是叶节点，则在他的右子树中一直向左找到其直接后继，然后互换位置，进行删除。此时可能会发生关键码数目下溢，需要进行旋转。

旋转：

如果在其左边、右边的兄弟子树有足够多的关键码，则可以进行旋转

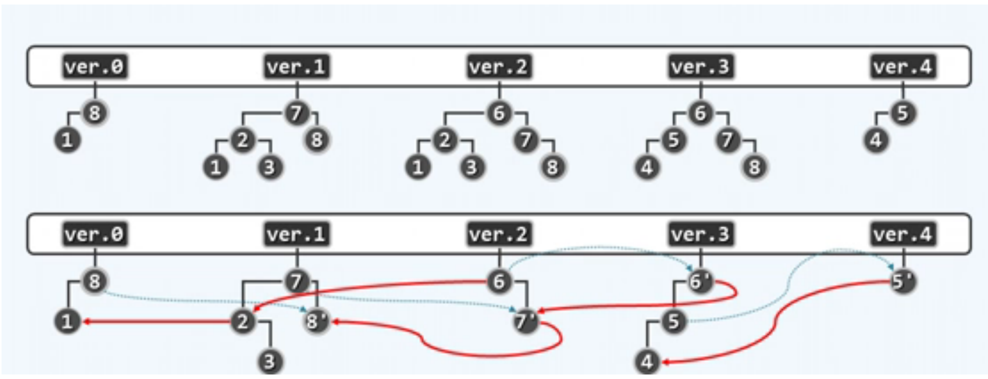


但是如果没有兄弟满足有足够多的关键码（或者不存在），则需要双方合并（此时左右兄弟至少存在其一）将上方的父节点合并下来。



父节点也可能发生下溢，如法炮制进行处理。

一种可以记录历史版本的树



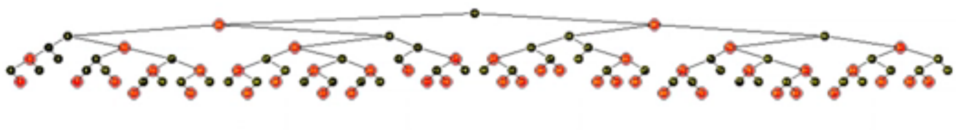
例如，红色线表示相邻版本可以保留的数据

蓝线表示更新的位置。

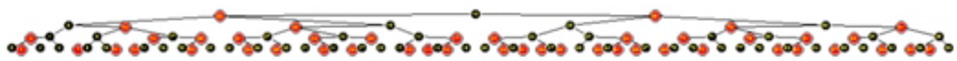
红黑树：

1树根必为黑色
2所有外部节点也必为黑色
3其余节点：红色节点只能有黑色孩子。（红父红子必为黑）
4从任何一个外部节点到根的黑色节点数目相等（黑深度）

提升变换：
将所有红色孩子提升至与父节点同一级的高度
结果不可能有两个红色节点相邻

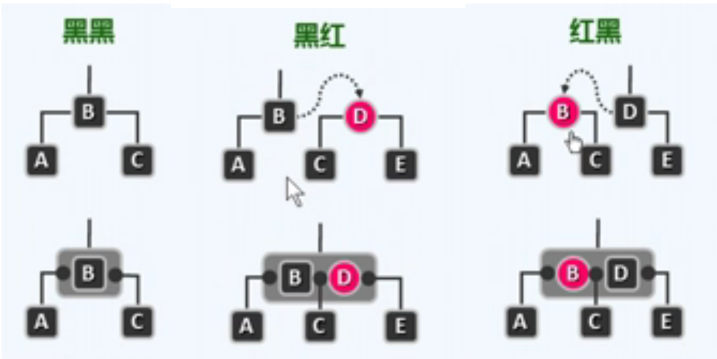


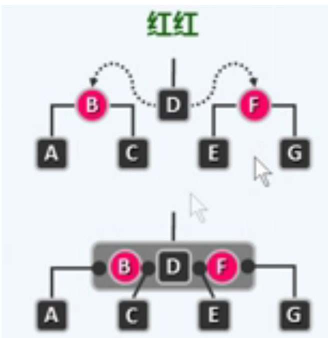
经过提升：



可以发现，所有底层节点的深度相同。
这也是为什么要求黑深度相等的原因。

本质：相当于一棵B树的拉伸
（2,4）树==红黑树
每一种提升后的红黑节点组合都相当于一个4阶B树





每一个节点最多拥有四个分支。

红黑树的高度：**特指黑高度。**

插入：

先将带插入值插入到确定位置，然后染为红色，此时第三条性质即其父亲必为黑色可能不满足。

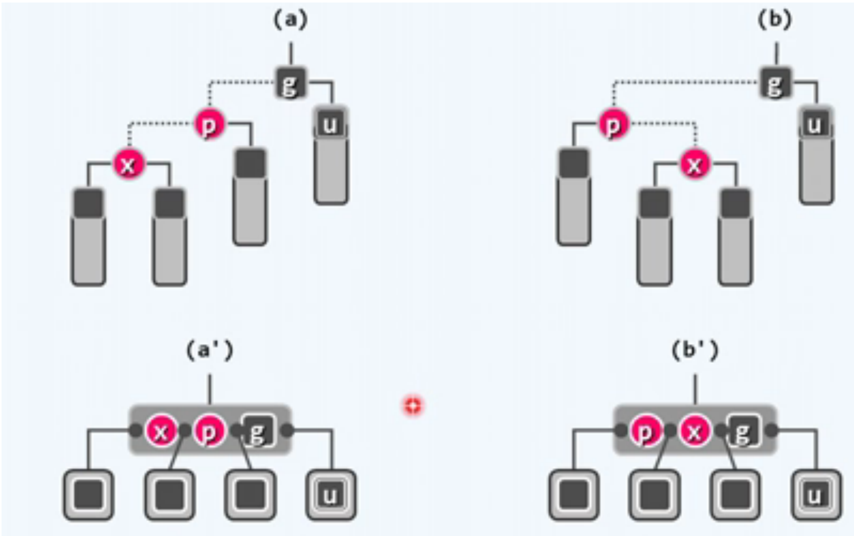
如何修复？

考察其父亲兄弟节点即插入节点叔父节点的颜色。

当叔父节点为黑色：

有四种情况：zigzag zigzig zagzag zagzig

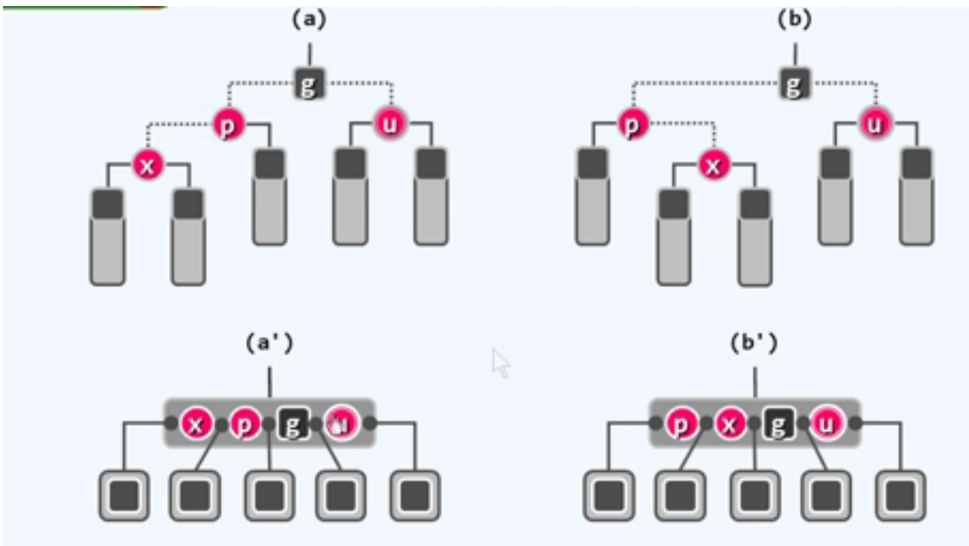
只考虑前两种情况



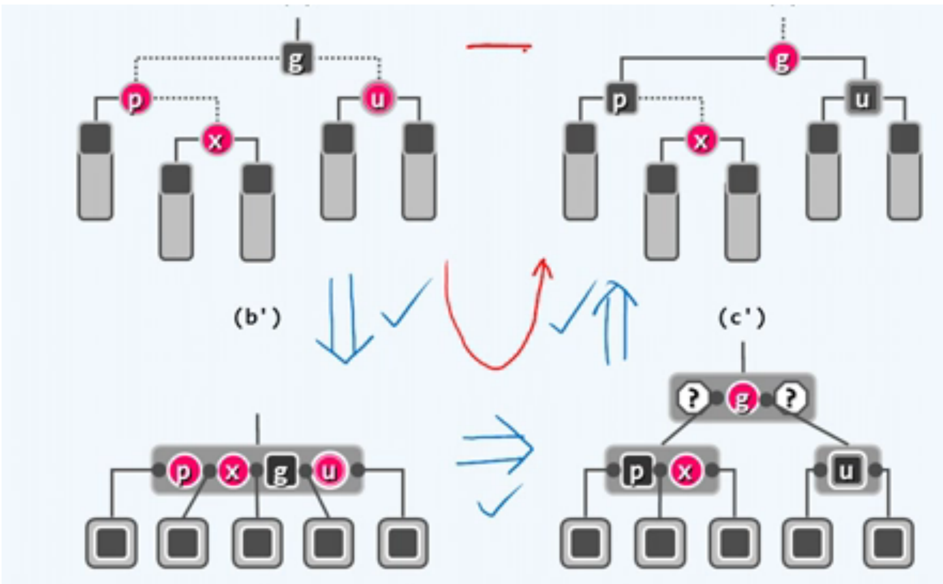
先收缩一下。

这时调整只需要重新染色。

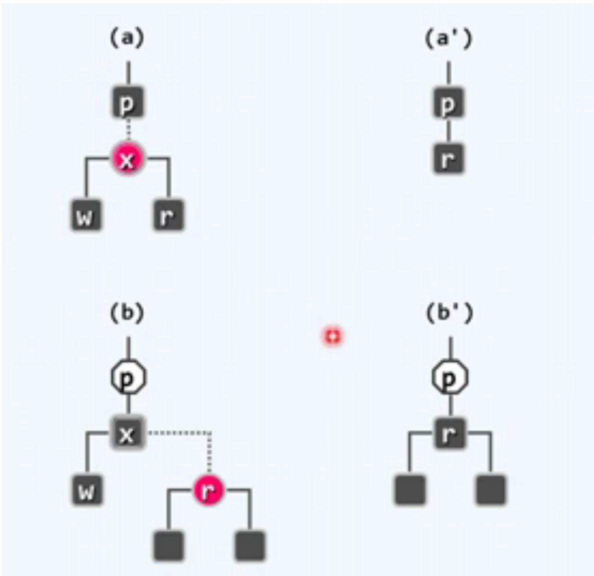
当叔父节点为红色：



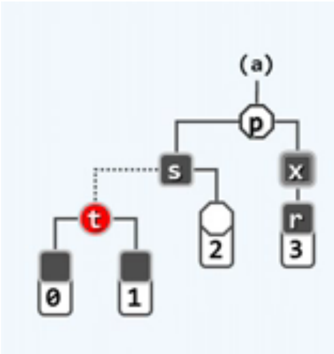
向上合并，形成非法的上溢（B树）
进行上溢调整分裂。
　　找到居中关键码，上移



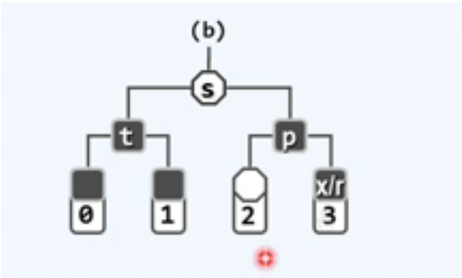
删除操作：
　　当删除节点与它的替代者至少有一个是红色的时候，可以直接删除（+重新染色）



但当两个节点都是黑色的时候，删除可能会破坏第4条性质（黑深度）
调整算法：联系到B树：合并后相当于下溢缺陷
　　因此与B树的调整算法相似。
　　分为四种情况：
　　1.当删除节点兄弟节点为黑，并且其兄弟节点至少有一个红孩子

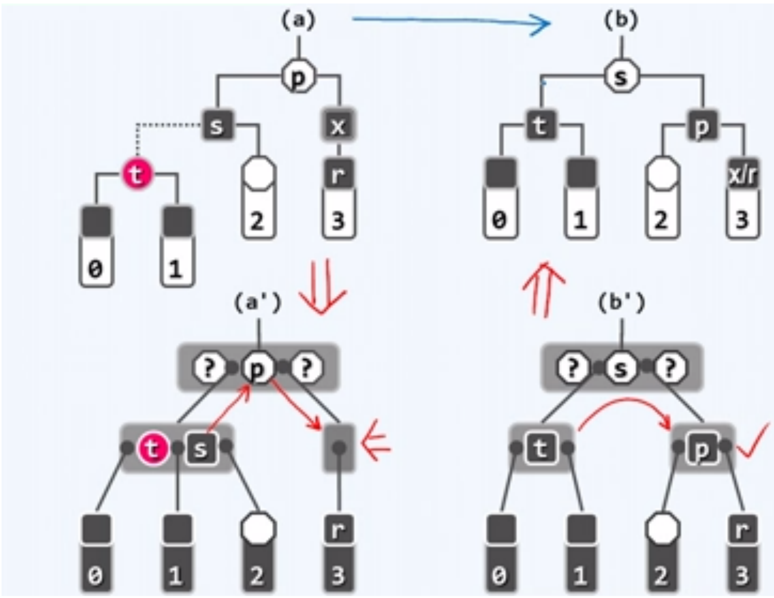


调整为:



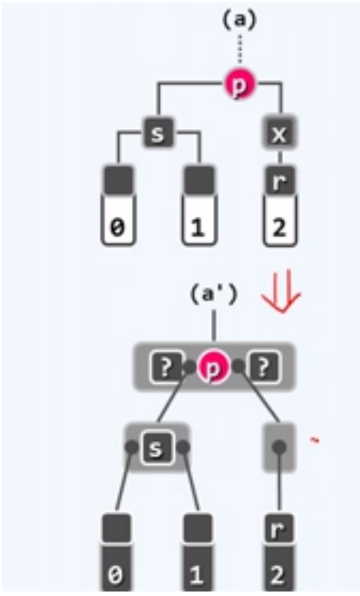
此时s继承P的颜色

实质：观察B树的旋转

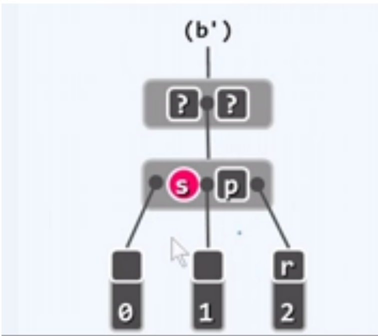


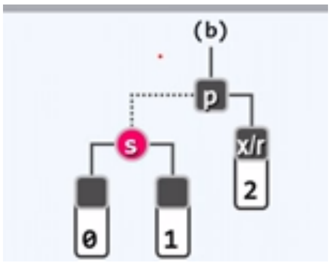
2.当兄弟节点为黑，并且它的两个孩子也为黑的情况:

（1）如果父节点P为红色:



此时无法旋转，只能合并。



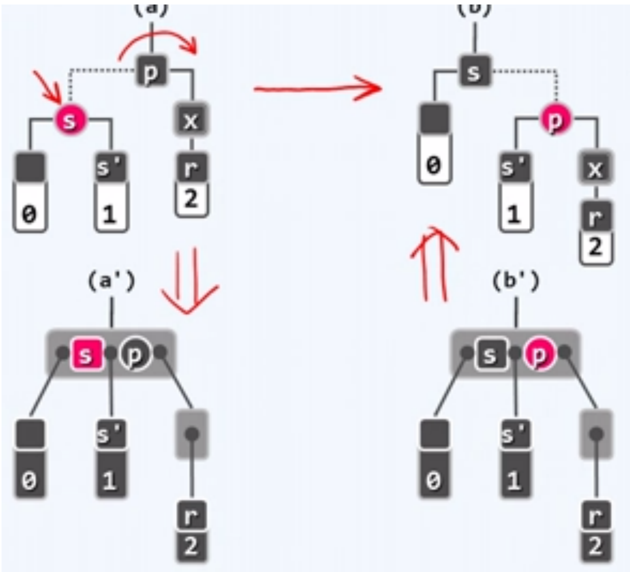
最终变为红黑树：

此时不会再次发生下溢（因为p的颜色为红色）

（2）如果父亲节点P为黑色：

经过上方的合并之后，可能会再次发生下溢，直到树根。

3.兄弟节点为红色的情况：



需要经过旋转并且颜色变换来回到前面的情况。并且只会转到父节点P为红色的情况。

第九章 词典
散列表

桶bucket
　直接存放或者间接指向一个词条
　桶数组/散列表： 容量为M， N < M << R
　通过一个函数将任意关键码通过hash函数转换为散列表中的某一个桶单元。
　优点：压缩了空间大小。

hash函数的设计：
　hash(key) = key % M (M 为散列表长)
特殊情况：
　有时候会产生多个key值在一个桶中的冲突。（鸽巢原理）

函数优秀的特点：
　　（1）确定
　　（2）快速
　　（3）满射
　　（4）均匀

四种函数：
一 除余法：

将M设为素数，可以更加平均。

二 MAD法：

除余法的改进。

除余法的缺陷：（一） 零点为不动点（只有0映射）

（二） 相邻关键码的散列地址也相邻（不在均匀）

MAD = multiply - add - divide

取M为素数， a > 0, b > 0, a % M != 0

hash(key) = (a * key + b) % M

三 数字分析：

抽取key中的某几位构成地址，比如取十进制的奇数位

改进： 去key的平方的中间若干位，构成地址。

四 折叠法：

总结：散列函数越是随机、越没有规律就越好。

五 随机数法：

可以将hash的设计套用伪随机数的方法。

关键码转换：

当关键码为数字时，不难将之转换为整数

但当关键码位字符串时，可运用多项式法。

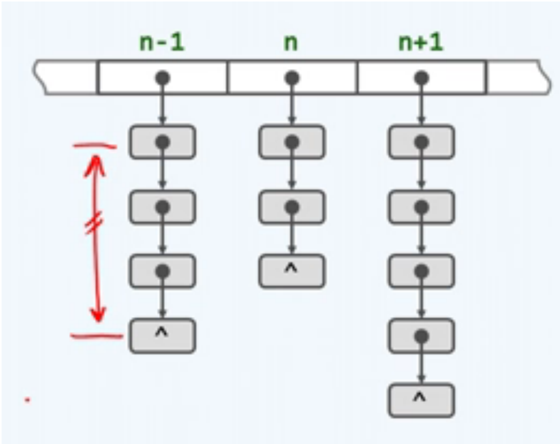
为什么要用如此复杂的方法?

因为如果用每个字母代表一个数字，然后相加求和的话，会导致很多冲突出现（很多字符串字符出现相同，只是顺序不同，或者字符串的和相同）。

每个桶存储多个数据的方法：

利用列表，每一个桶一个列表，动态申请内存

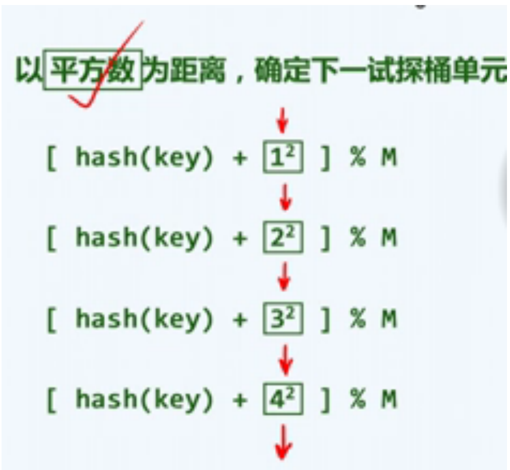
优点：无需预先分配大小，灵活方便。

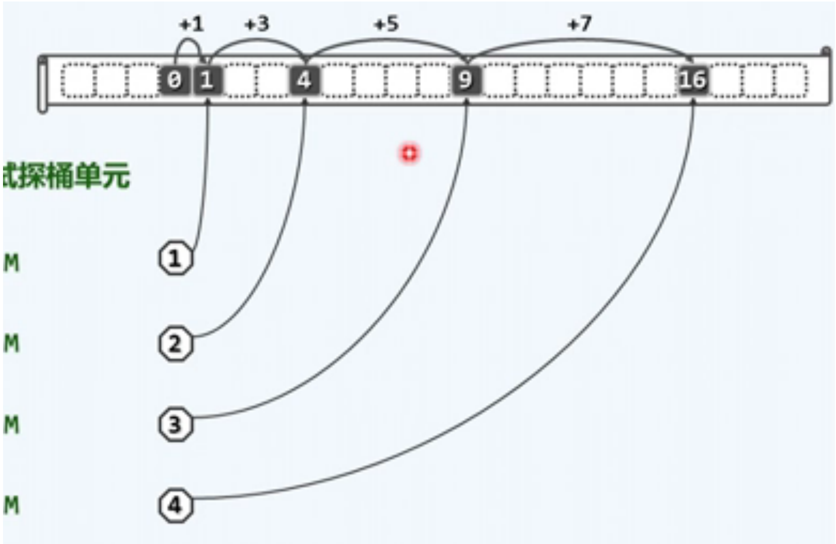


但这种方法缺点也很大：空间未必连续分布，因此缓存几乎失效；并且动态分配内存需要额外时间。

开放定址策略：
每一个词条对应一个桶序列，按照优先级排序。
即如果优先级最高的桶有空位，则直接放入，如果没有，则向后移，直到找到第一个有空位的桶。
在删除的时候，直接在桶中做一个删除标记，当查找的时候直接略过去，当添加的时候如果找到有删除标记则直接添加。

平方试探：每一次试探不是线性向后移动，而是按照平方来向后移动。





但这时会发生有的空桶怎么也搜索不到而不被利用的情况

定理：在M（素数）个桶单元中，如果用平方探测法，则只会利用M/2取上整个桶。

改进：双向平方探测

先+1^2 再 -1^2 再 +2^2 ……以此类推

± i^2		-36	-25	-16	-9	-4	-1	0	1	4	9	16	25	36
M	5					1	4	0	1	4				
	7				5	3	6	0	1	4	2			
	11		8	6	2	7	10	0	1	4	9	5	3	
	13	3	1	10	4	9	12	0	1	4	9	3	12	10

如果将双向试探的位置罗列出来 大致应该是这样

这样的方法对特定的素数有效但对其他素数依然无法遍历所有空桶。

4K+3类的素数，可以用此方法。

而4K+1无法。

双平方定理：

任意素数P可以表示为一对整数的平方和，当且仅当P % 4 = 1

$$(u^2 + v^2)\cdot(s^2 + t^2) = (us + vt)^2 + (ut - vs)^2$$

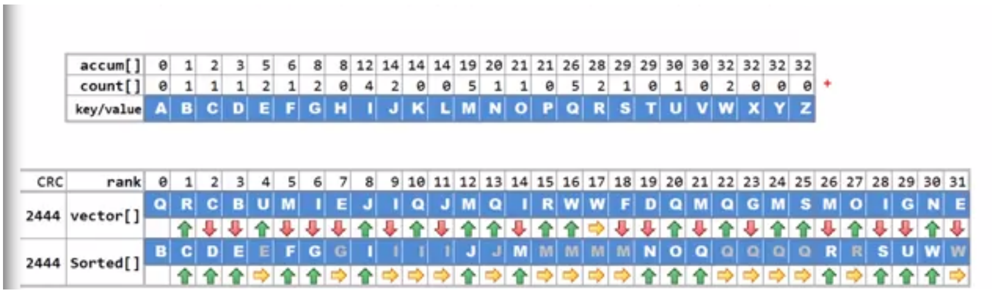
$$(2^2 + 3^2)\cdot(5^2 + 8^2) = (10 + 24)^2 + (16 - 15)^2$$

————————————————————————————

桶排序、计数排序

大数据+小范围
数据的取值范围为[0,M)
运用散列表
例如：26个字母的排序
建立26个桶
首先每一个桶中有一个计数器，记录字母出现的次数
只要遍历一次字符集，只要遇到一个字母，就将对应的桶计数器加一，这样一遍即结束。

在遍历的同时，记录字母的累计值
这样就可以得到相应的排序位置。



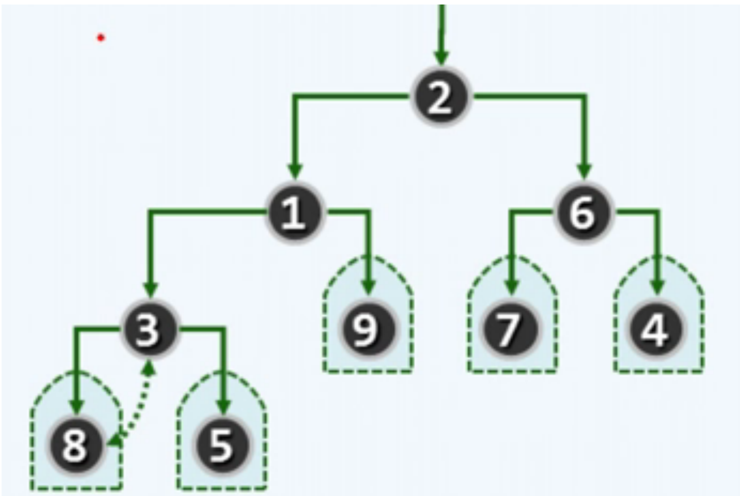
第十章 堆

完全二叉树
类似于AVL树，平衡因子非负
在逻辑上等同于完全二叉树
在物理上借助向量实现
依照层次遍历次序彼此对应
父节点（如果有的话）为（i-1）/2
左孩子：1+i*2
右孩子：（1+i）*2
完全二叉堆
堆序性：
每个节点的数值都不超过他的父亲。因此最大元必然是根节点。

新节点的插入：上滤
将它作为末元素接入向量
与他的父亲比较，如果大于父亲，则互换位置（以此类推）

节点删除：
由于堆的特殊性，我们只需要准备删除根（最大元素）的算法：

将根删除，并且将末节点移到根的位置，与他的孩子比较，将最大的元素移到根的位置，再次交换，直到满足序列性。

交换的优化：
可以将待交换节点备份，与父（子）节点逐一比较，然后只让父（子）节点逐一下移（上移），再把备份节点直接移动到适当位置，这样复杂度为logn+2,少于3logn（直接交换）
如何建堆?
从小到大，由每一个叶子节点开始，进行堆的两两合并（通过下滤）


从n/2-1的节点开始，每一个局部子树根节点进行下滤（从n/2取下整——1）

——————————————————————————
堆排序：
反复的交换、下滤直到堆变空。
先把最大的（根元素）与末元素交换，再删除，然后再次下滤，这时堆的规模便减少1.

第十一章 串
提供接口：
长度、按秩取值、截断、连接、比较大小、判断字串是否存在
KMP匹配：
跳过一部分已匹配字符
制作查询表，每次遇到一个失配条件，就查询失配位置的值并且移到对应位置。
如何制表?
只需要在失配的时候修改模式串中的位置j
而不需要修改文本串的位置i
制表：每一个字符对应一个匹配失败后的位置，在构造时，通过前置哨兵

（值为-1），从0开始，i表示模式串，j表示匹配串，不妨这样考虑：
目前已经匹配到了i = n，即p[i] = j ,考虑n+1，如果 i = j+1 则令当前匹配j+1，且 p[++i] = ++j,如果不相等，则考虑p[p[i]]，这样递推下去，直到有相等的匹配。如果不，则会递推到哨兵即-1,这是p[++i] = ++j, 即等于首元素。

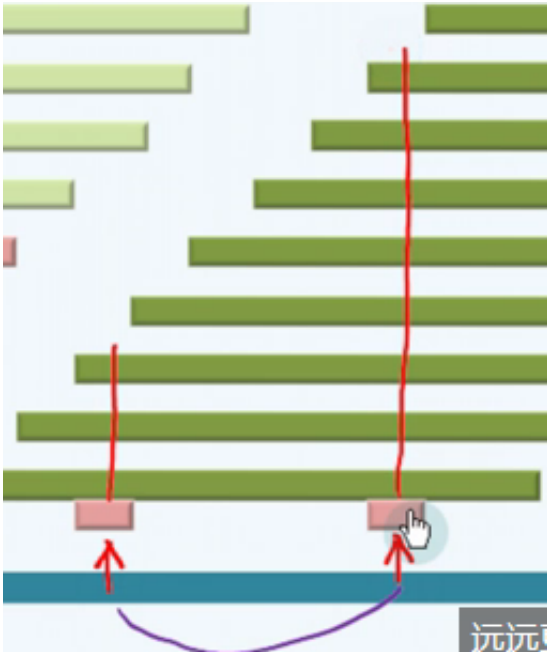
优化：当++i = ++j 时，直接跳过，直到不相等，因为如果相等相当于白白匹配，仍然不会匹配成功。

————————————————————————————

BM_BC算法

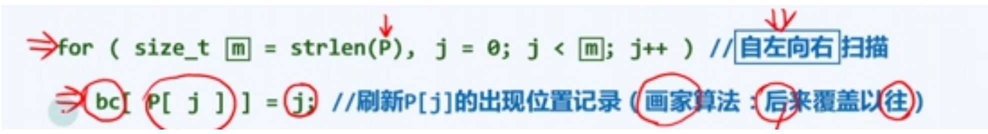
关注点：首先让失败出现，也就是加速失败的出现，使得正确的位置更早到来。
而在模式串中位置越靠后，匹配失败几率越大

在比对失败，移动到下一个匹配位置时，对靠后位置的比对能够排除更多的对齐位置。



因此可以首先比对最后一个字符。如果匹配位置的文本元素没有出现在模式串中，可以大胆移过这个位置（从末尾匹配的失败是大概率的）。如果出现了但是不匹配，则可以移动到最靠后的相应的模式串中的元素。之后递推。

可以通过预处理，制作BC表，首先将所有表项的初值设为-1（作为哨兵）采用画家策略，记录所有位置的最靠右的秩。



```
for ( size_t m = strlen(P), j = 0; j < m; j++ ) //自左向右扫描
    bc[ P[ j ] ] = j; //刷新P[j]的出现位置记录（画家算法：后来覆盖以往）
```

对于容易失败的匹配，BM算法更适合。（比如字母表范围更大的时候）
但是最差情况时，可能退化到蛮力算法的水平。

————————————————————————————

BM_GS算法
通过从末尾匹配，可以积累到一段相匹配的字符串，尽管会发生不匹配的情况，但是我们可以利用最后那段已经匹配的字符串（利用经验）来改进BM算法。
一种优化方式：目前不匹配的字符，在移动之后不能还是这个字符（与KMP的优化一样）。

Karp-Robin匹配法

思想：将字符串对应为一个特定的数字，并且可以双向变换。
例如：十进制的串可以直接视为一个自然数
而至于其他的字符串，每个字符串都对应一个D进制的自然数。
例如，A-Z可以看作26进制。
如此，串与串的比对已经转换为了数字大小的比对。

如果模式串很长，那么它所对应的自然数就会非常大，这样会造成溢出。因此需要用到Hash的算法。

利用模余散列，可以进行一个一个比较。如果发生冲突，则进行严格比对。
如何将字符串转换为数字这一过程加速呢？（因为每次比对都需要转换）
其实，相邻两个字串之间只差了首尾两个字符，因此这之间的转换有联系。

第十二章 排序
快速排序：

普通方法：选定一个定轴点，然后从后往前直到有元素小于此点，交换位置之后再从前向后，直到有元素大于此点，交换，以此类推，直到确定轴点位置。
变种1：随即选定一个元素与首元素交换位置作为定轴点，然后采用L+H+U的顺序，即L表示小于定轴点的区间，H表示大于的区间，U表示还没有比较的区间。

初始化之后，从U中选取，没遍历一个元素，即比较大小，如果大于，则不用管直接向后移动一个元素（此时已经自动加入到了H），如果小于，则只需要将H的第一个元素与这个元素进行交换（不需要将H整体向后移动）。并且准备两个指针进行记录三块区域的分界线就好。

——————————————————————————————
K选择问题

选取一组数中第K小的数
1.扫描众数：即在一组数中出现次数>半数
此时也就是说，中位数=众数

在高效中位数算法未知之前，必须搜索频繁数（即出现次数最多的数）。
考虑向量A，若在向量A的前缀P（|P|为偶数）中，元素x出现的次数恰为半数，则A有众数当且仅当对应的后缀A-P有众数m，且m就是A的众数

于是可以推导出一个算法：设置一个计数器，每次初始化为1，表示当前前缀中有c个数与首元素相等，然后向后迭代，如果与首元素相等，则c++，不相等则c--，直到c = 0（也就是与首元素相等的有一半元素），那么这时候可以将之排除，重新换到下一个前缀，同时将众数改为新的前缀的首元素。
这样进行下去之后，众数只能是最终那个序列的频繁数。

2.选取第K小的数：
QuickSelection：
通过快排的过程，一次排序过后，可以确定定轴点的秩，然后可以通过k与此秩的比较大小，将前段或者后段剪掉，精简快排过程，最终得到想要的元素。

更好的方法：先将数据集分成若干块，找到这若干块的中位数，在找这些中位数的中位数（利用以上方法（众数）），然后以此数为基数，分为三块，并且在利用QuickSelection进行选择，这样会减少不确定性，使得问题最终确定在线性范围。

——————————————————————————————
shell排序

将一维的数据看作一个矩阵，进行**逐列**排序，使其越来越窄，直到成为一列。
如何视为一个矩阵？不必可以开一个二维数组，只需要利用内存中存储二维数组的方法：寻秩访问，就可以了。例如第二行就是A[i+2*n]

各列内部排序如何实现？采用输入敏感方法，保证有序性可持续改善，而且成本低廉——插入排序。
插入排序的时间取决于逆序对的总数，因此在逐步减少列数的过程中，逆序对在总体上是逐步减小的。