数据结构与算法

平衡树

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

平衡树 (Balanced Tree)

- Definition
  - A *tree* where no *leaf* is *much farther* away from the *root* than any other leaf.

- Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.

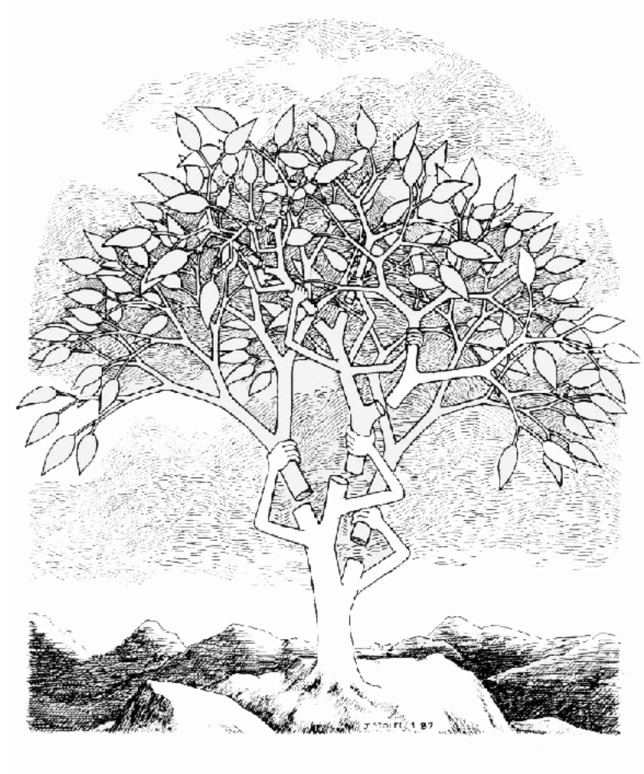
Different Balancing Schemes

- Red-black tree： also known as symmetric binary B-tree，因结点分红、黑两色而得名（1972年由Rudolf Bayer发明，J. Guibas 和 Robert Sedgewick 1978年的一篇论文给出Red-black tree的命名）
- AVL Tree： 得名于发明者名字的首字母缩写，Adelson-Velskii & Landis (1962)
- Top-down 2-3-4 tree： 结点分成2-、3-、4-叉而得名
- Splaying (priority tree)
- ……

伸展树

伸展树

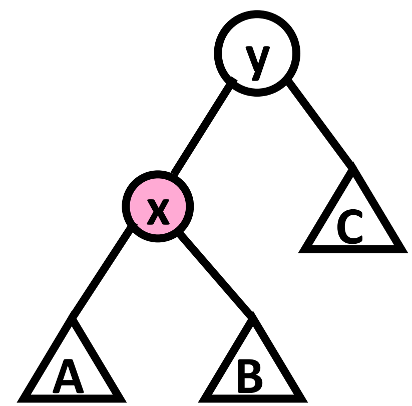
- 一种自组织数据结构
  - 1985年由Daniel Sleator & Robert Tarjan设计
  - 数据随检索而调整位置
  - 汉字输入法的词表
- 并非一个新的数据结构，只是改进BST性能的一组规则
  - 保证访问的总代价不高，达到最令人满意的性能
  - 不能保证最终树高平衡



A Self-Adjusting Search Tree

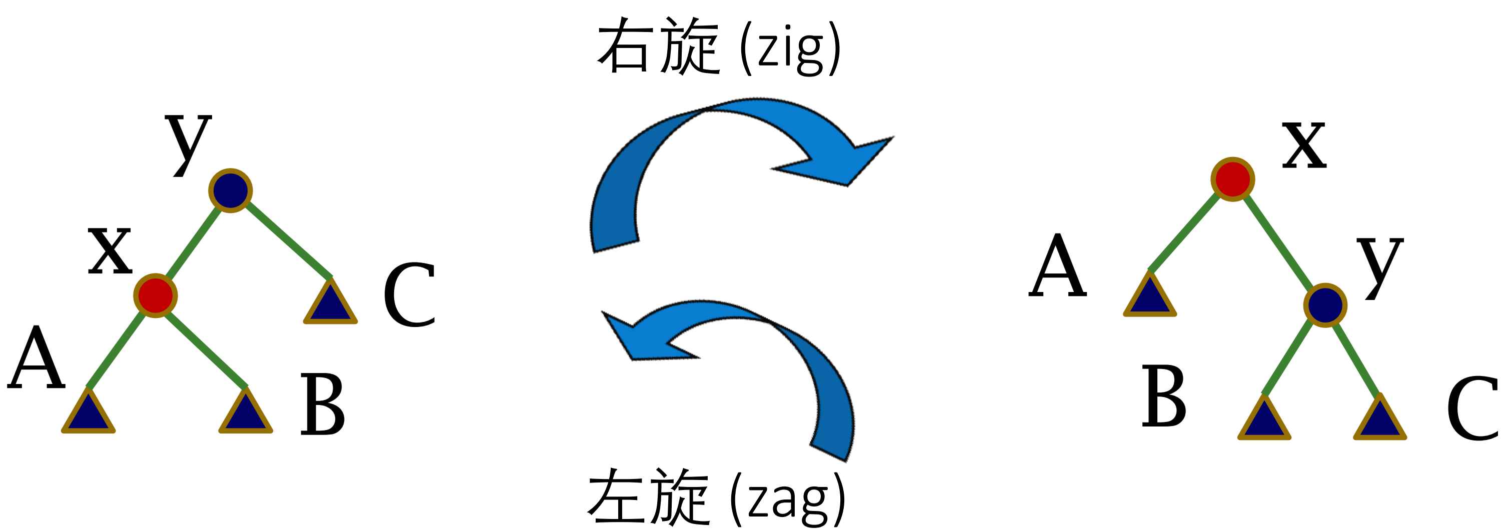
伸展/展开 (splaying)

- 访问一次结点（e.g., x），完成一次称为展开的过程
  - x 被检索或插入时，将结点 x 调整到BST的根结点
  - 删除结点 x 时，将结点 x 的父结点调整到根结点
- 同AVL树的调整，结点 x 的一次展开包括一组旋转(rotation)
  - 调整结点 x、父结点、祖父结点的位置
  - 把 x 移到树结构中的更高层

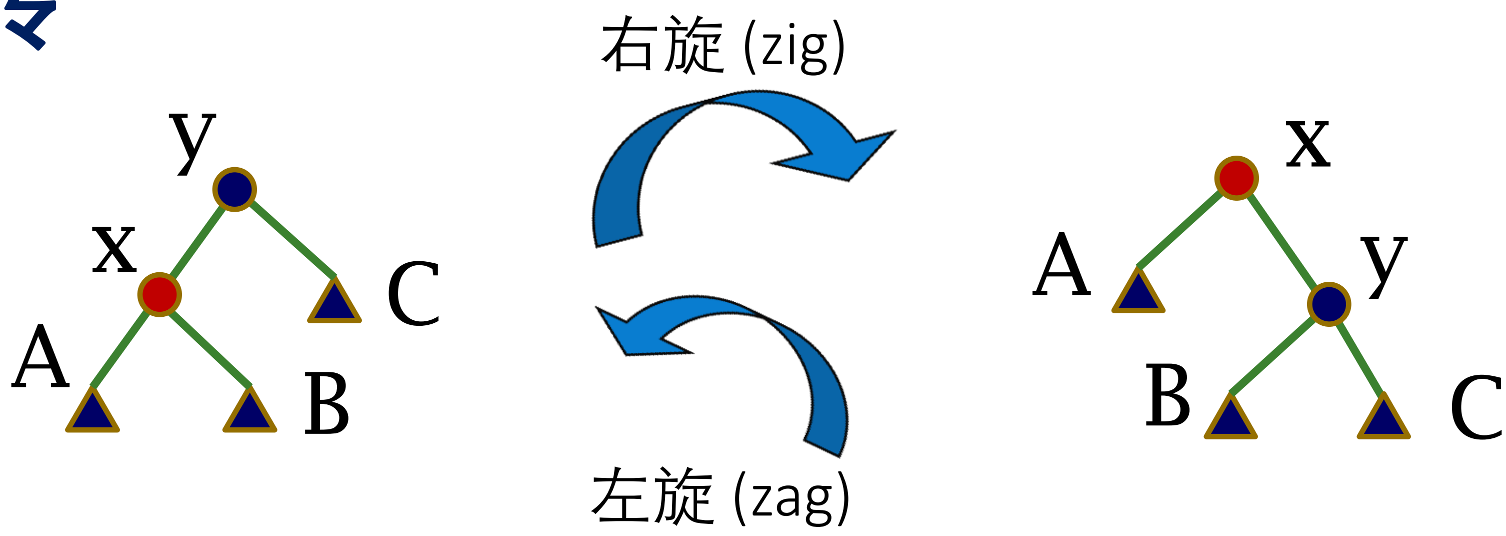


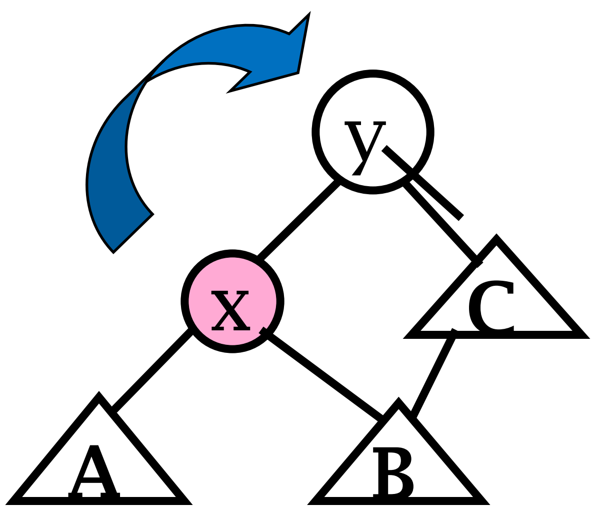
单旋转 (single rotation)

- x 是根结点的直接子结点， y = father(x)
  - 结点 x 与其父结点 y 交换位置
  - 保持BST特性



单旋转





双旋转(double rotation)

- 涉及
  - 结点 x
  - 结点 x 的父结点　　y = father(x)
  - 结点 x 的祖父结点　z = father(y)

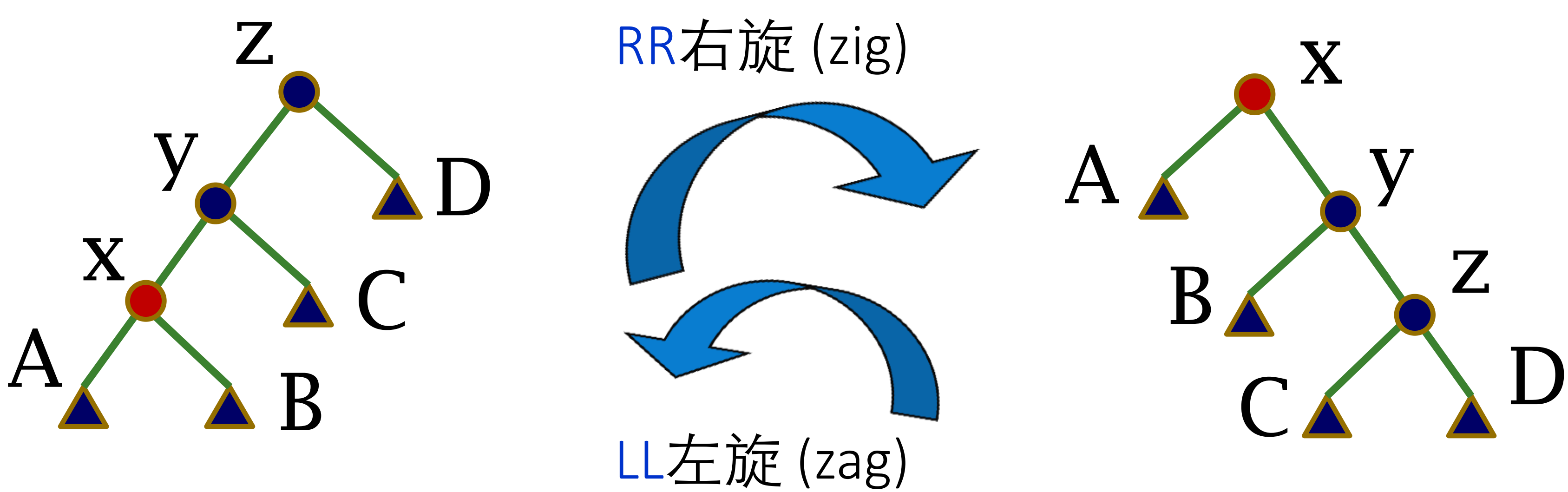
- 目的：　将结点 x 在树结构中 向上移两层

双旋转

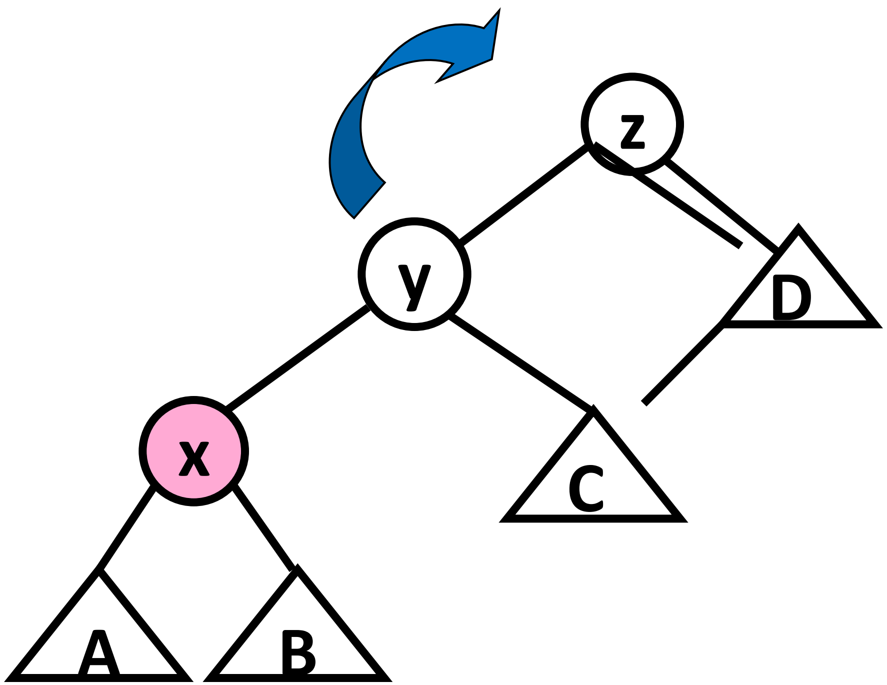
- 分两类
  - 一字形旋转(zig-zig rotation)
    - 也称 同构调整
    （homogeneous configuration）
  - 之字形旋转(zig-zag rotation)
    - 也称 异构调整
    （heterogeneous configuration）

双旋转：一字形旋转

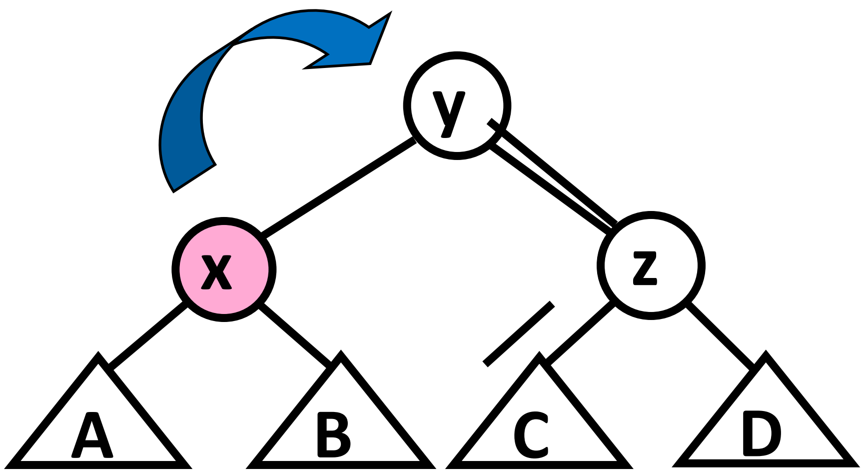
- 表现为LL 和 RR型 双旋：保持 BST 的中序性质
- x、y、z呈一顺
  - 结点 x 是结点 y 的左子结点，y 是结点 z 的左子结点
  - 结点 x 是结点 y 的右子结点，y 是结点 z 的右子结点



双旋转：一字形旋转



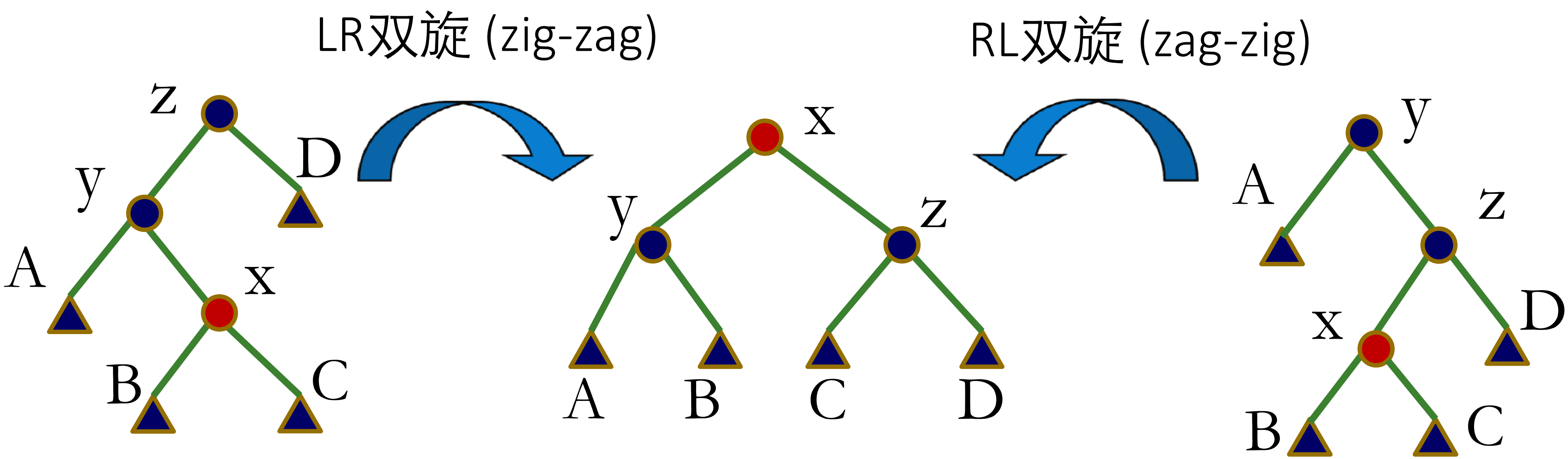
双旋转：一字形旋转



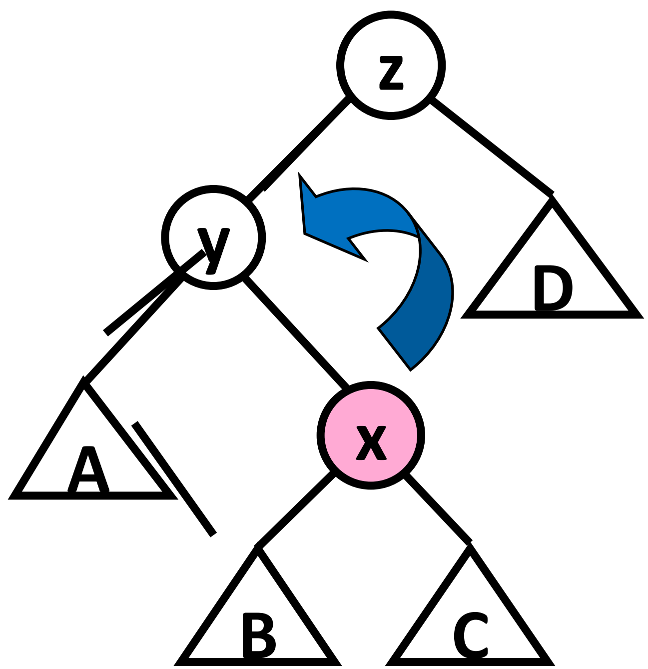
双旋转：之字形旋转

- x、y、z呈 之字 扭结
  1. 结点 x 是结点 y 的左子，y 是结点 z 的右子结点
  2. 结点 x 是结点 y 的右子，y 是结点 z 的左子结点

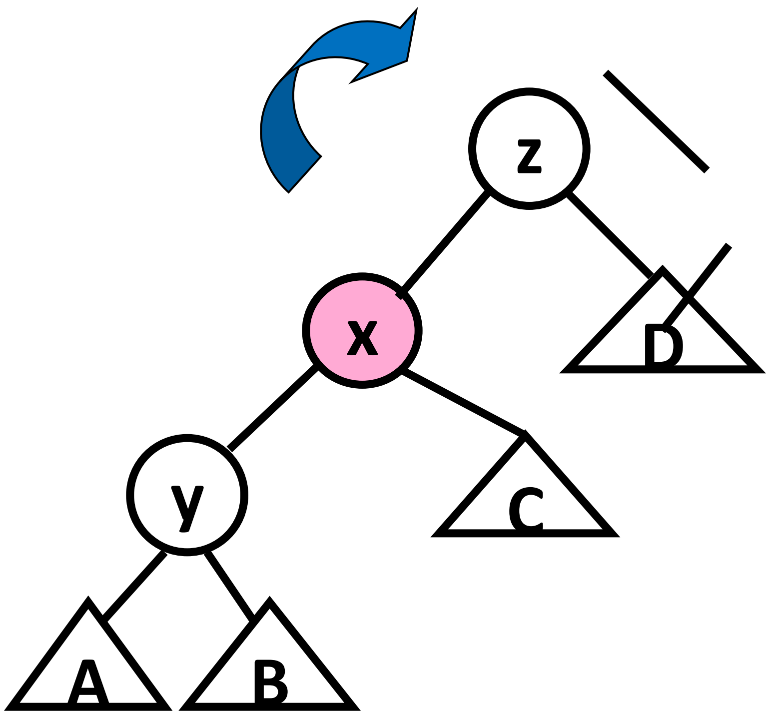
  表现为 LR 和 RL 型 双旋：保持 BST 的中序性质



双旋转：之字形旋转



双旋转：之字形旋转



双旋转：两种旋转的不同作用

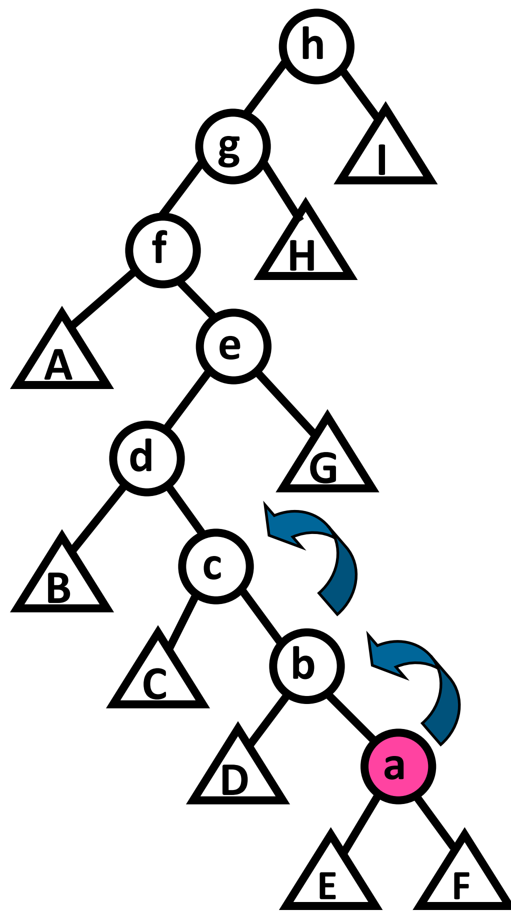
- 之字形旋转
  - 新访问的记录向根结点移动
  - 子树结构高度减 1
  - 树结构趋于平衡
- 一字形提升
  - 通常不会降低树结构的高度
  - 只将新访问的记录向根结点移动

伸展树的调整过程

- 一系列**双旋转**
  - 直到结点 x 到达根结点 或 成为根结点的子结点
- 若结点 x 为根结点的子结点
  - 进行一次**单旋转**使结点 x 成为根结点

- 调整使得树结构趋于平衡
  - 访问频繁的结点靠近树结构的根层
  - **减少访问代价**

伸展树的调整过程



(a-b-c)

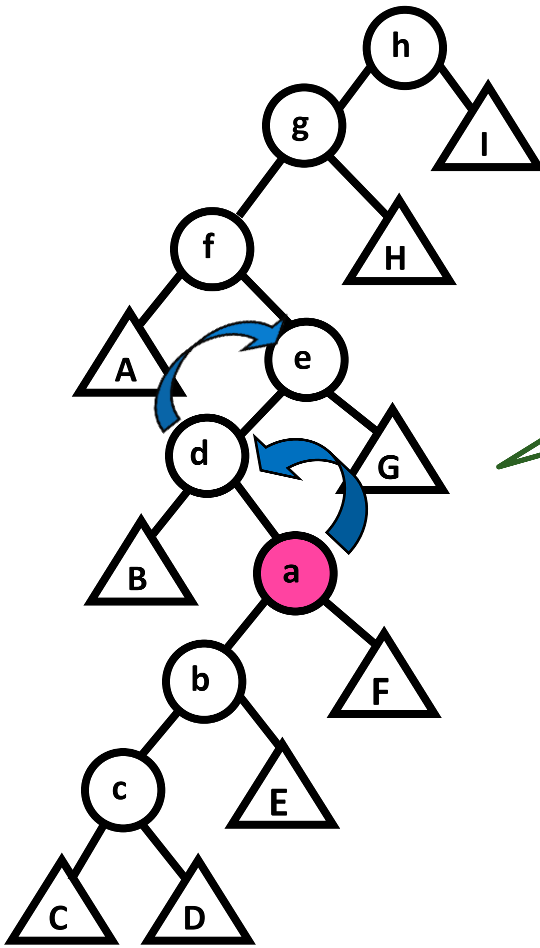
一字形旋转

(b, c)左转

(a, b)左转

伸展树的调整过程

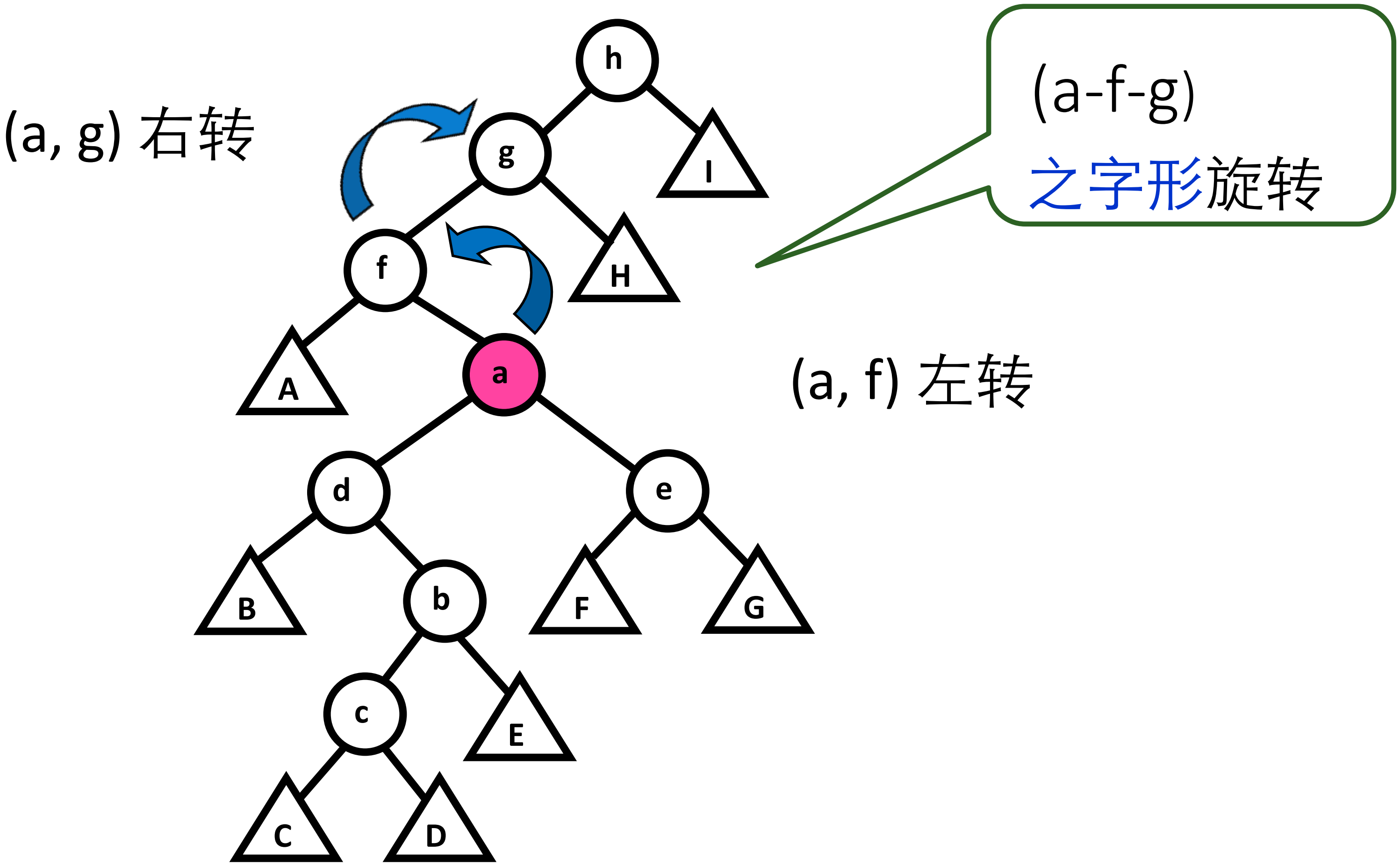
(a, e) 右转



(a-d-e)
之字形调整

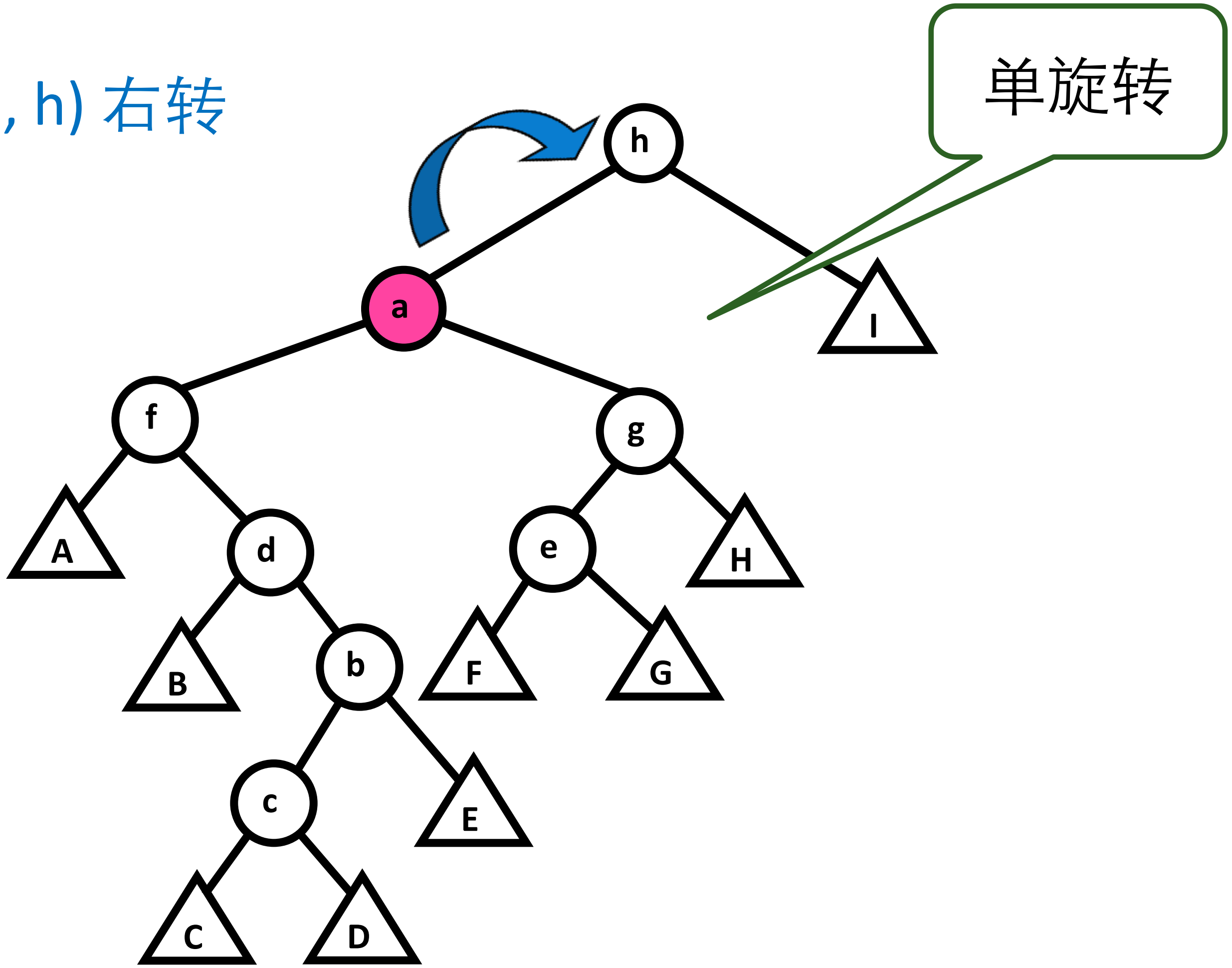
(a, d) 左转

伸展树的调整过程

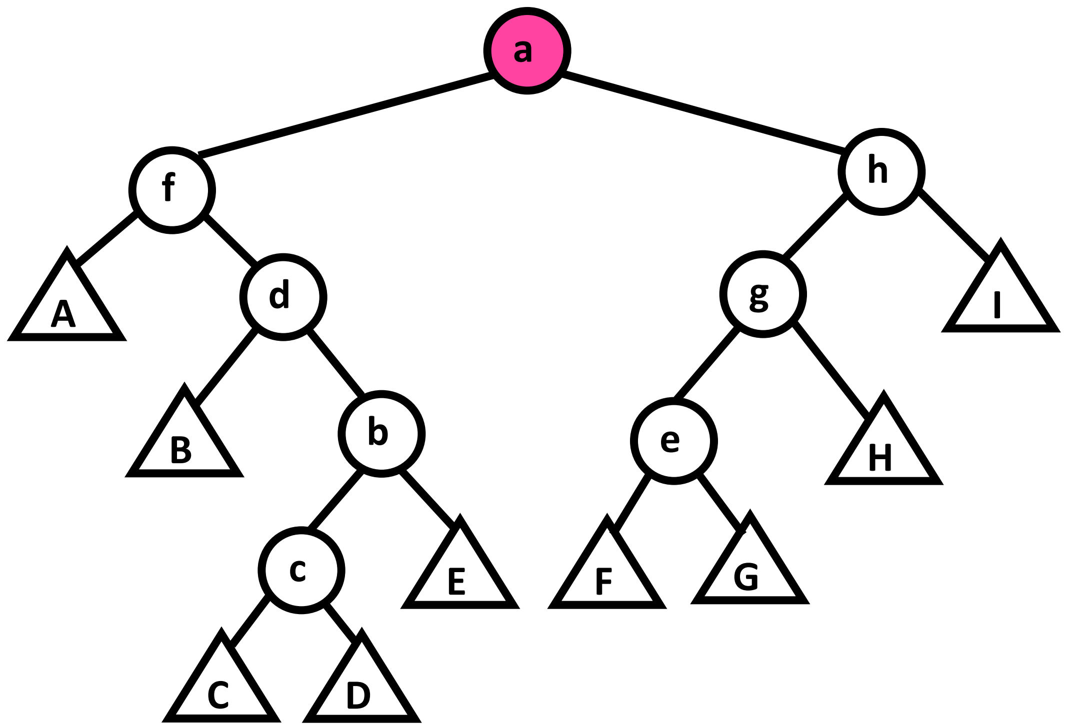


伸展树的调整过程

(a, h) 右转



伸展树的调整过程



伸展树上的基本操作

- find(x, s) : 采用BST检索，后splaying
- insert(x, s) : 采用BST插入，后splaying
- delete(x, s) : 采用BST删除，后splaying
  - 采用BST检索，再合并或分离子结点
  - join(s1, s2)： 两棵树的合并
  - split(x, s)： 将树s根据给定的 x 分成两部分

Splay 树的操作

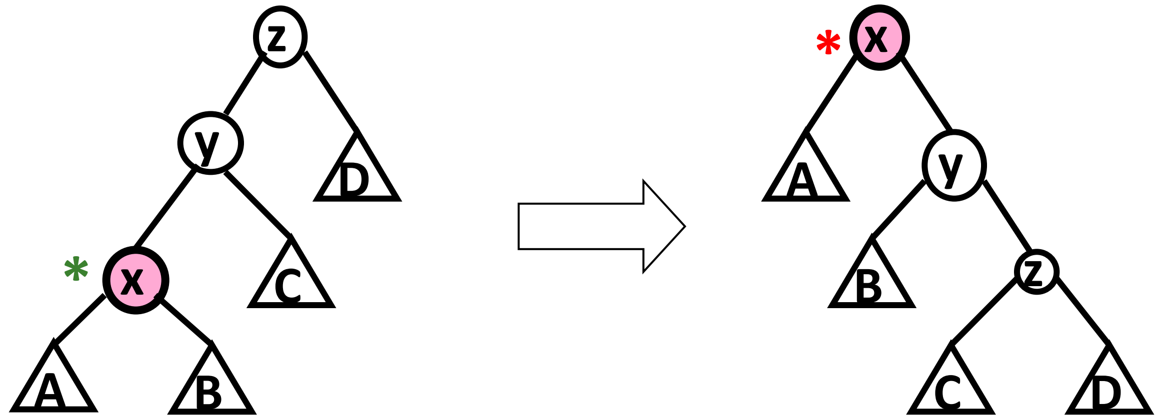
```
struct TreeNode {
    int key;
    ELEM value;
    TreeNode *father, * left, *right;
};

Splay(TreeNode *x, TreeNode *f);          // 把 x 旋转到祖先 f 下面
Splay(x, NULL)；                           // 把 x 旋转为根
Find(int k，  TreeNode *f) ;               // 查询 k
Insert(int k，  TreeNode *f)；             // 插入值 v
Delete(TreeNode *x) ;                     // 删除 x 结点
DeleteTree(TreeNode *x);                  // 删除 x 子树
```

Splay 树的操作

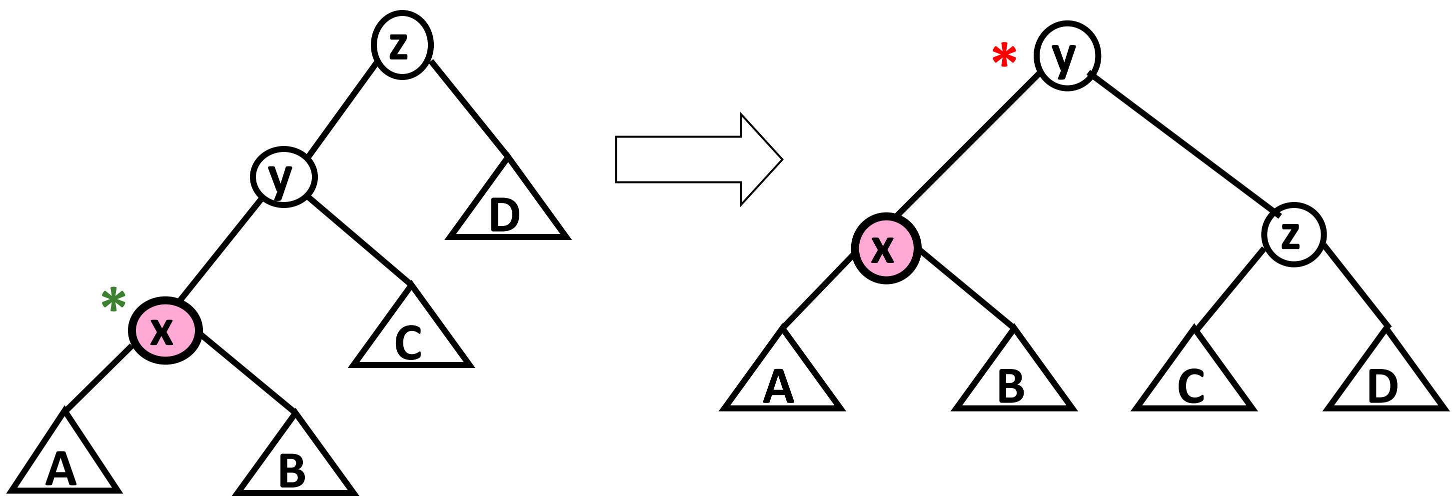
```
void Splay (TreeNode *x, TreeNode *f) {
    while (x->parent != f) {
        TreeNode *y = x-> parent, *z = y-> parent;
        if (y->parent != f )  {                                  // y 不是 f 的子结点
            if (z->lchild == y) {                                // LL or RR
                    if (y->lchild == x)  { Zig(y);  Zig(x); }    // 一字型双右旋
                else  { Zag(x);  Zig(x); }                       // x左旋上来，接着右旋
            } else {                                             // RL or  LR
                    if (y->lchild == x) {  Zig(x);  Zag(x); }    // x右旋上来，接着左旋
                    else  { Zag(y);  Zag(x); }                   // 一字型双左旋
             }
        } else {
            if (y->lchild == x) Zig(x);                          // 右单旋
            else Zag(x);                                         // 左单旋
        }
    }
    if (x->parent == NULL)  Root = x;
}
```

半伸展

普通
一字旋转



半伸展
一字旋转



下一次旋转从 y 开始，而不从 x 开始

伸展树的效率

n 个结点的伸展树

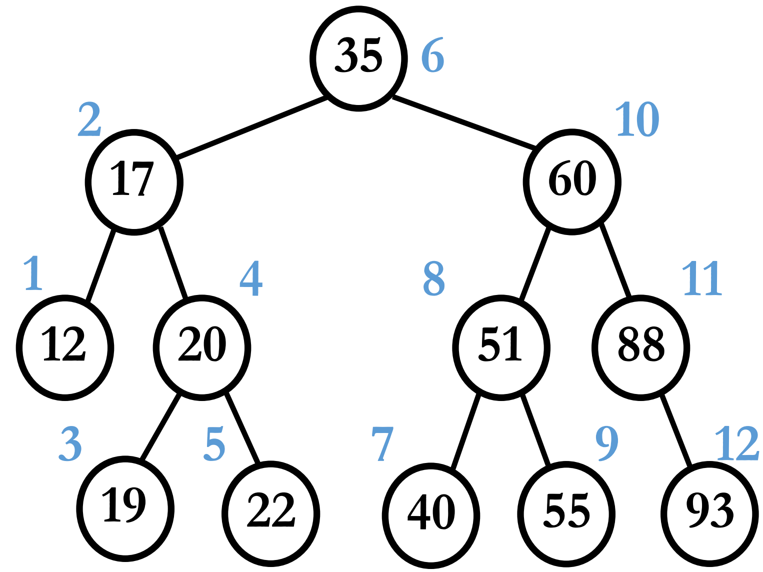
- 进行一组 m 次操作（插入、删除、查找操作），当$m \geq n$ 时，总代价为 O(m log n)
  - 不能保证每一单个操作高效
  - 即，每次访问的平摊代价(*amortized running time*) 为 O(log n)

伸展树的问题

- Splaying is a strategy focusing upon the *elements* rather than the shape of the tree.
  - It may perform well in situation in which *some elements are used much more frequently* than others
  - If the elements near the root are accessed *with about the same frequency* as elements on the lowest levels, then splaying may not be the best choice

伸展树的应用

- 删除大于 u 小于 v 的所有结点
  - 把 u 结点旋转到根
  - 把 v 旋转为 u 的右儿子
  - 删除 v 结点的左子树




```
void DeleteUV(TreeNode* rt, TreeNode* u, TreeNode* v ) {
    Splay(u, NULL);
    Splay(v, u);
    DeleteTree(v->lchild);
    v->lchild = NULL;
}
```

伸展树的应用

- 字典
- 找第k小值
- 求满足 $k_1 \leq key \leq k_2$ 的 所有key值之和
- 最大前缀和（前k大的key值之和为k-前缀和，求最大的k-前缀和）

- 结点中需要维护附加信息
  - 子树的结点个数
  - 子树的所有结点之和

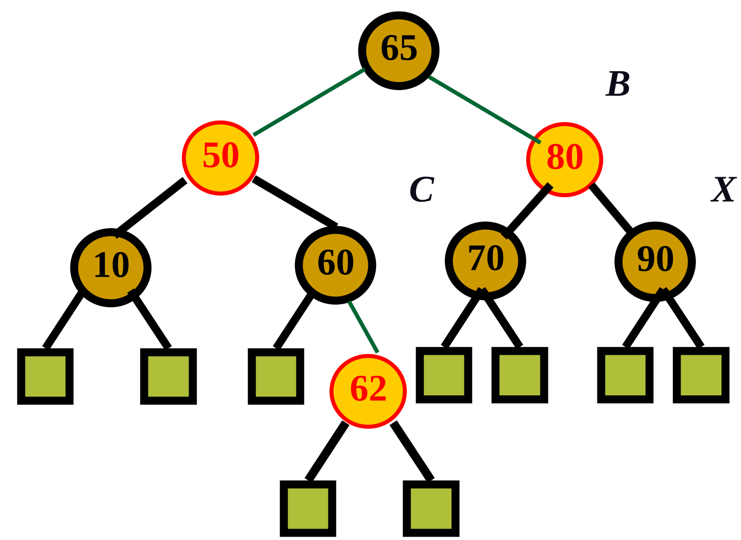
思考

- 初始空的BST中依次插入1，2，3，…， n 形成一棵BST，……

- 请调研 Splay 树的各种应用
- 红黑树、AVL 树和 Splay 树的比较
  - 它们与访问频率的关系?
  - 树形结构与输入数据的顺序关系?
  - 统计意义上哪种数据结构的性能更好?
  - 哪种数据结构最容易编写?

2-3-4 树

- 《算法导论》：
  - 假设将一棵红黑树的每个红结点“吸收”到其黑色父结点中，并将红结点的子女变为黑色父结点的子女（忽略关键字的变化）。当一个黑结点的所有红色子女都被吸收后
  - 其可能的度是多少?
    - 2、3、4
    - 即成为一棵2-3-4树（阶为4的B树）
  - 此结果树的叶结点深度怎样?
    - 就是RB的阶
    - 叶结点等高



2-3-4 & RB-Tree

1. Represent 2-3-4 tree as a BST.
2. Use "internal" red edges for 3- and 4- nodes.

3. Require that 3-nodes be left-leaning.

2-3-4 & RB-Tree

Key Properties

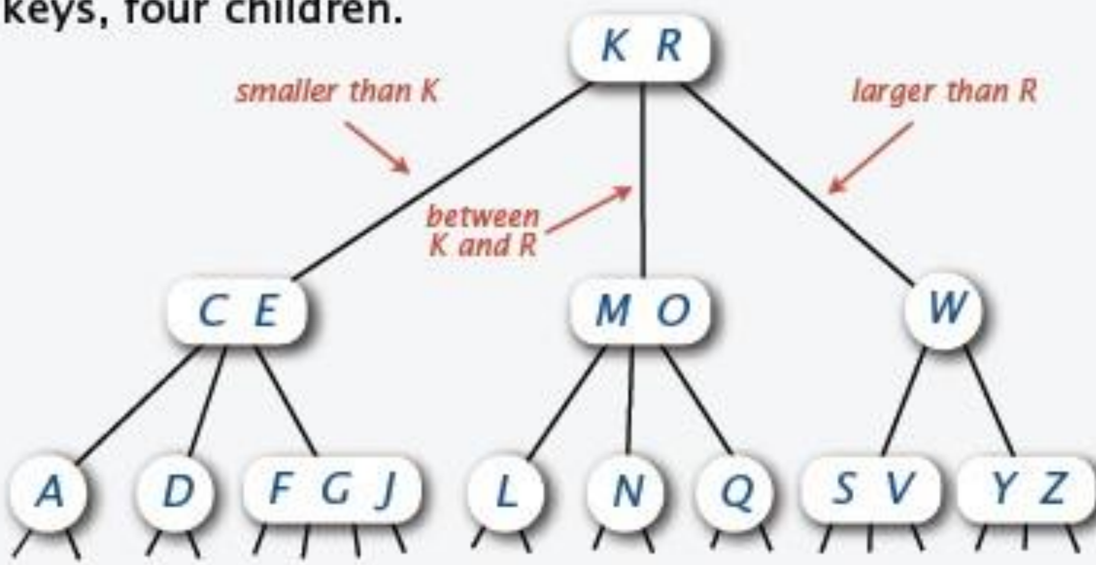
- elementary BST search works
- easy-to-maintain 1-1 correspondence with 2-3-4 trees
- trees therefore have perfect black-link balance

每个结点的颜色定义为 进入该点的边的颜色---→ red-black tree

4阶B树

Allow 1, 2, or 3 keys per node.

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



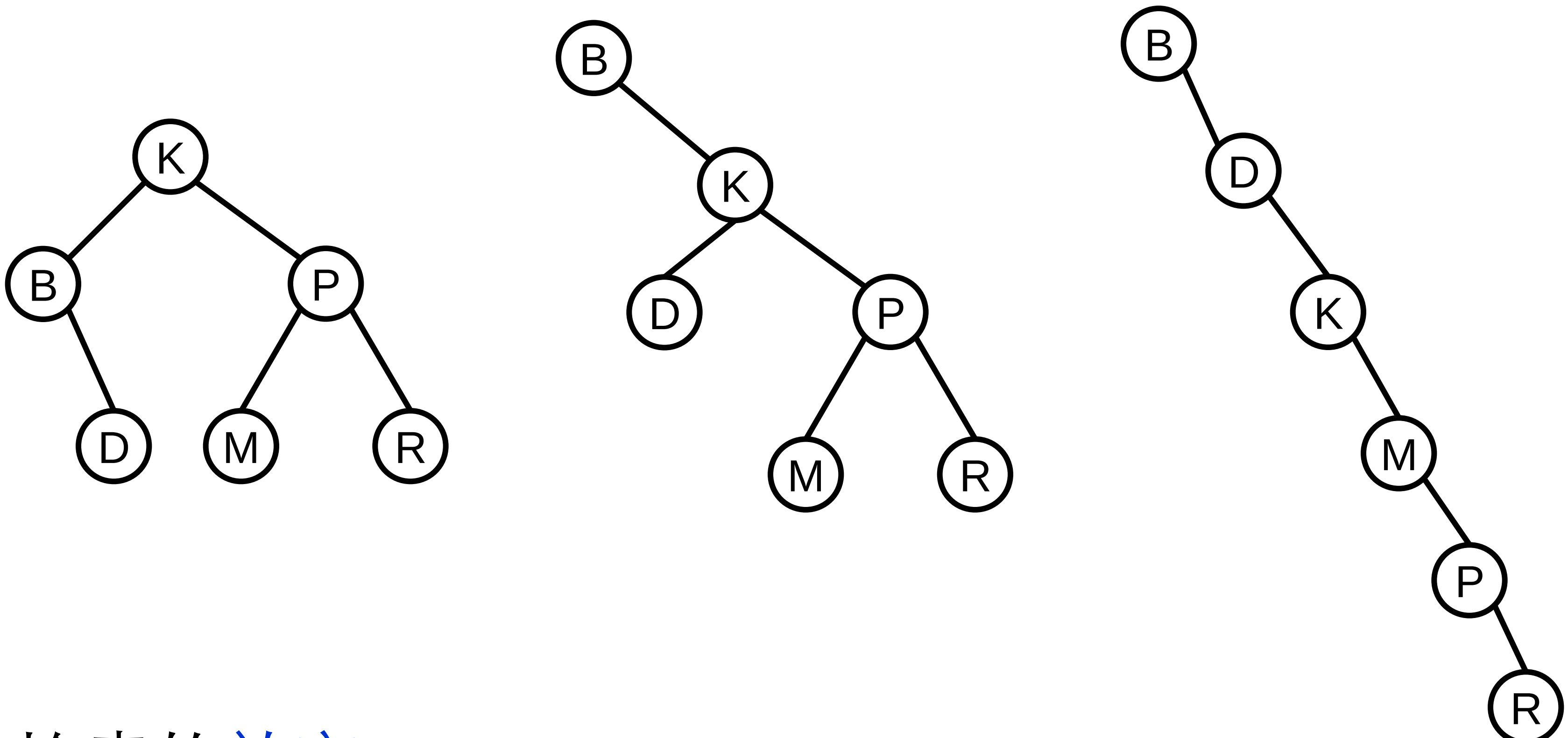
几种平衡机制比较

- AVL树要求完全平衡，高度平衡
  - 树结构与访问频率无关，只与插入、删除的顺序有关
- 伸展树与操作频率相关
  - 根据插入、删除、检索等动态地调整
  - 无需附加信息
- RB-Tree局部平衡，阶平衡
  - 统计性能好于AVL树
  - 增删记录算法性能好、易实现
  - C++ STL的set、multiset、map、multimap都应用了红黑树的变体

索引的效率问题

- 索引(indexing)：把一个关键码与其对应的数据记录的位置相关联
  - （关键码，指针）对，即(key, pointer)
- 按结构分三类索引
  - 线性索引：有序数组、索引顺序文件
  - 散列索引
  - 树型索引：二叉搜索树(BST)、B/B^+树、字符树 ……

二叉搜索树



- 检索的效率　 vs .
- 树的形状、特点　& 结点的检索频率

二叉检索树的效率衡量

- 检索、插入、删除等操作的效率均依赖于二叉检索树的高度 *h*，时间代价为*O(h)*
  - 最佳：高度（尽可能）最小
  - 最差：退化成线性结构

何 **为** 一棵最佳二叉检索树?

如何 **构筑** 一棵最佳二叉检索树?

如何 **保持** 二叉搜索树的最佳特性?

二叉检索树的效率再考察

- 成功的检索：比较次数为关键码（内部结点）所在层数 + 1
- 不成功的检索：比较次数等于其所属的外部结点的层数

故，BST中检索一个关键码的平均比较次数为

$$ASL(n) = \frac{1}{w}[\sum_{i=1}^{n} p_i(l_i + 1) + \sum_{i=0}^{n} q_i l'_i] \qquad W = \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i$$

其中，l_i 是第i个内部结点的层数，l'_i 为第i个外部结点的层数，p_i是被检索第i个内部结点所代表的关键码的频率，q_i是被检索第i个外部结点代表的可能关键码集合的频率

各结点等概率检索情况

$$\frac{p_1}{W}=\frac{p_2}{W}=\ldots=\frac{p_n}{W}=\frac{q_1}{W}=\ldots=\frac{q_n}{W}=\frac{1}{2n+1}$$

$$\begin{aligned}\mathrm{ASL}(n)&=\frac{1}{2n+1}\left(\sum_{i=1}^{n}(l_i+1)+\sum_{i=0}^{n}l'_i\right)\\&=\frac{1}{2n+1}\left(\sum_{i=1}^{n}l_i+n+\sum_{i=0}^{n}l'_i\right)\\&=\frac{1}{2n+1}(I+n+E)\\&=\frac{2I+3n}{2n+1}\end{aligned}$$

- 平均比较次数 *ASL(n)* **最小**的**前提** 是扩充二叉树内部路径长度 *I* 最小

各结点等概率检索情况

- 一棵二叉树里，路径长度为0 结点有且仅有1 个，路径长度为1 结点 至多 2个，路径长度为2 结点至多4个，……

故，有 n 个结点的二叉树其内部路径长度 l 至少等于序列 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4,……的**前n项和**，即

$$\sum_{k=1}^{n}\left\lfloor \log_{2}{}^{k} \right\rfloor$$

即，

$$\sum_{k=1}^{n}\left\lfloor \log_{2} k \right\rfloor = (n+1)\left\lfloor \log_{2} n \right\rfloor - 2^{1+\lfloor \log_{2} n \rfloor} + 2$$

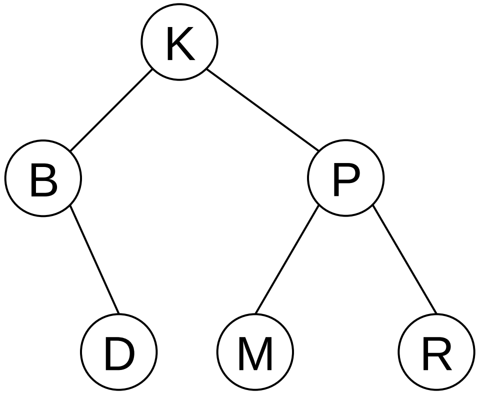
最佳二叉搜索树

检索概率相同时，先对序列进行排序：

B D K M P R

尔后用二分法依次插入这些关键码

```
void balance(int data[], int first, int last) {
    if (first <= last) {
        int middle  = (first + last)/2；
        insert(data[middle]);
        balance(data, first, middle-1);
        balance(data, middle+1, last);
    }
}
```



最佳二叉搜索树

- 问题
  - 额外的数组
  - 新数据的添加可能导致树不再最佳

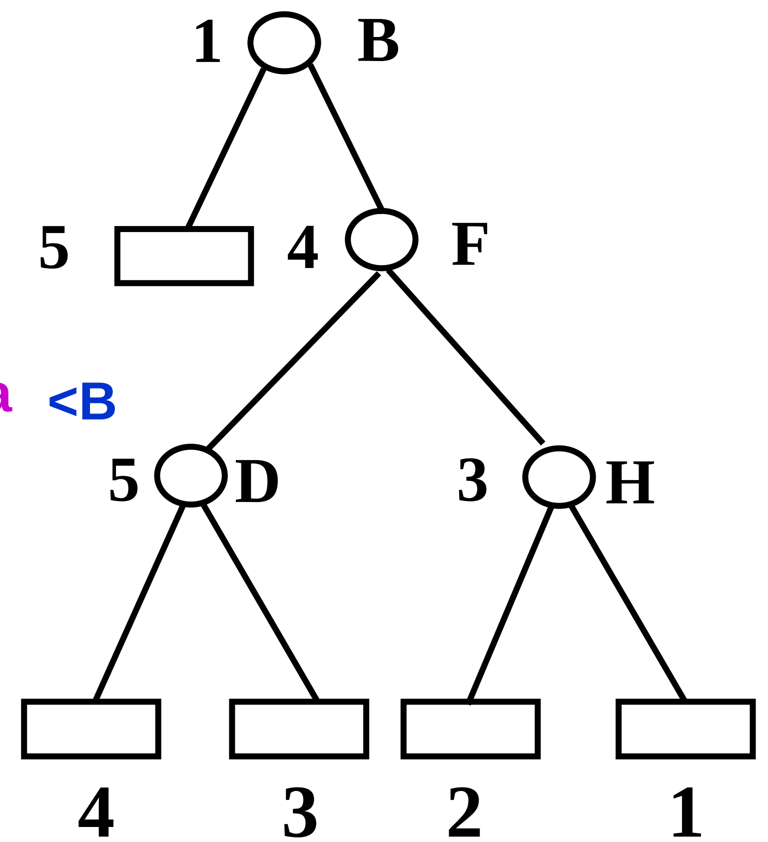
保持最佳二叉搜索树的平衡

- Red-black tree
- AVL Tree
- Top-down 2-3-4 tree
- Splaying (priority tree)
- ……

最佳二叉搜索树

检索**概率不等**时如何构造?

根据**关键码集合**及**检索概率**:
给定一个已排序的带权关键码集合，如何构造一个ASL**最小**的二叉搜索树?



a <B

c (B,D) e (D,F) g (F,H) >=i >H

$$ASL(n)=\frac{1}{W}\left[\sum_{i=1}^{n}p_i(l_i+1)+\sum_{i=0}^{n}q_1 l_i'\right]$$

构造最佳二叉搜索树

- 最佳二叉搜索树
  - 任何子树都是最佳二叉搜索树
  - 具有最佳子结构、重复子结构的动规特点
- 动态规划构造过程
  - 第1步：构造包含1个结点的最佳二叉搜索树
    - 找t(0, 1)，t(1 , 2)，…，t(n-1 , n)
  - 第2步：构造包含2个结点的最佳二叉搜索树
    - 找t(0 , 2)，t(1 , 3)，…，t(n-2 , n)
  - 再构造包含3，4，…个结点的最佳二叉搜索树
  - 最后构造包含n个结点的t(0 , n)

最佳二叉搜索树 t(i, j)

l_x： 内部结点k_x所在层数

l'_x： 外部结点x所在层数

- 根为r(i, j)
  - 内部结点的关键码为k_{i+1}， k_{i+2}， …， k_j (0≤i≤j≤n)
  - 结点的权为 (q_i， p_{i+1}， q_{i+1}， …， p_j， q_j)
- 开销 C(i, j)， 即 $\sum_{x=i+1}^{j} p_x(l_x+1)+\sum_{x=i}^{j} q_x l'_x$
- 权的总和 W(i, j) =

$$p_{i+1} + \ldots + p_j + q_i + q_{i+1} + \ldots + q_j$$

最佳二叉搜索树t(i, j)

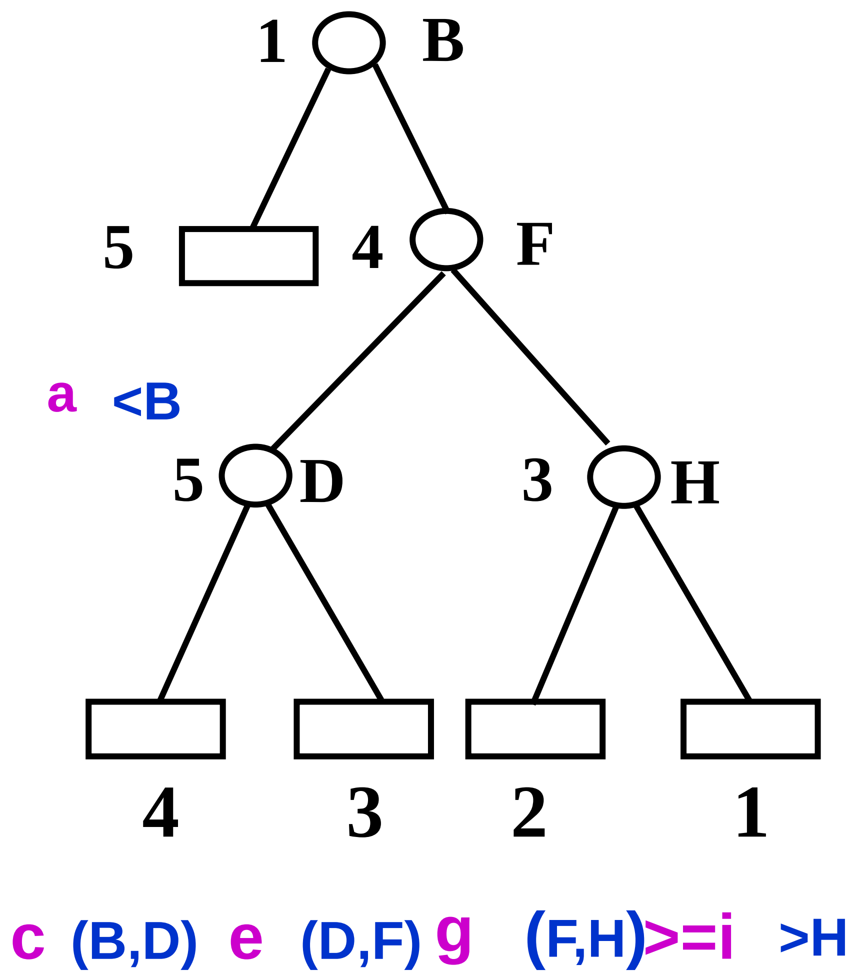
- 以 k_x 为根
  - 左子树包含 k_{i+1}， …， k_{x-1}
    - C(i, x-1)
  - 右子树包含 k_{x+l}， k_{x+2}， …， k_j
    - C(x, j)

$$C(i, j) = W(i, j) + \min_{(i \le x \le j)} (C(i, x-1) + C(x, j))$$

最佳二叉搜索树

关键码：
{ B, D, F, H}

权序列:
(1, 5, 4, 3, 5, 4, 3, 2, 1)



最佳二叉搜索树(i, j)

i \ j	0	1	2	3	4
0	0	1	2	2	2
1		0	2	2	3
2			0	3	3
3				0	4
4					0

r(i ,j)

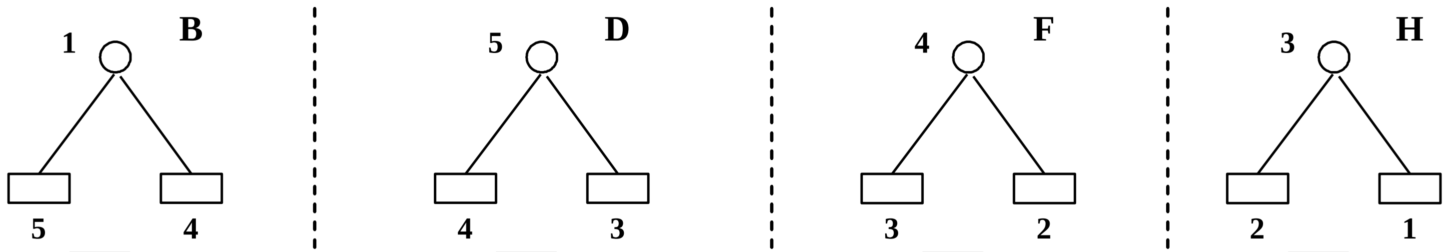
i \ j	0	1	2	3	4
0	0	10	28	43	57
1		0	12	27	40
2			0	9	19
3				0	6
4					0

C(i ,j)

i \ j	0	1	2	3	4
0	5	10	18	21	28
1		4	12	18	22
2			3	9	3
3				3	6
4					1

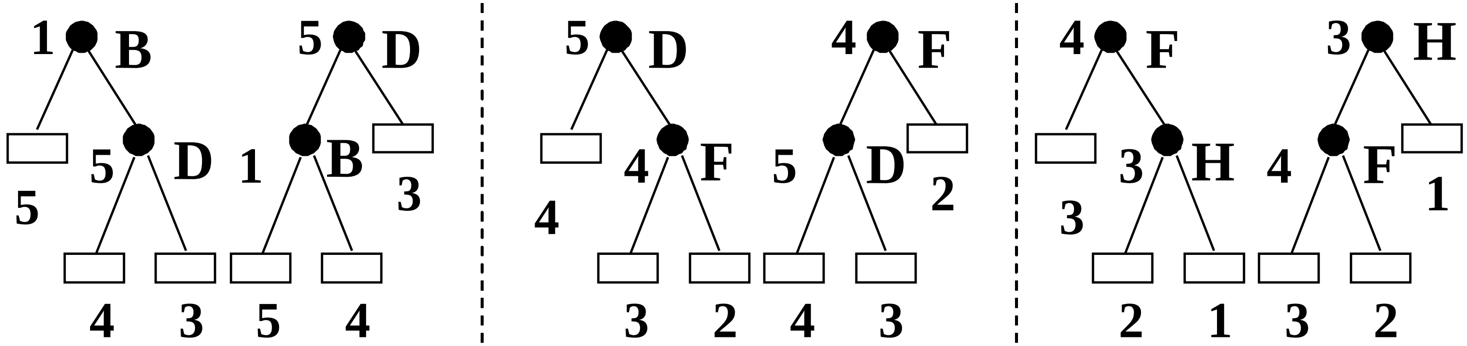
W(i ,j)

第一步



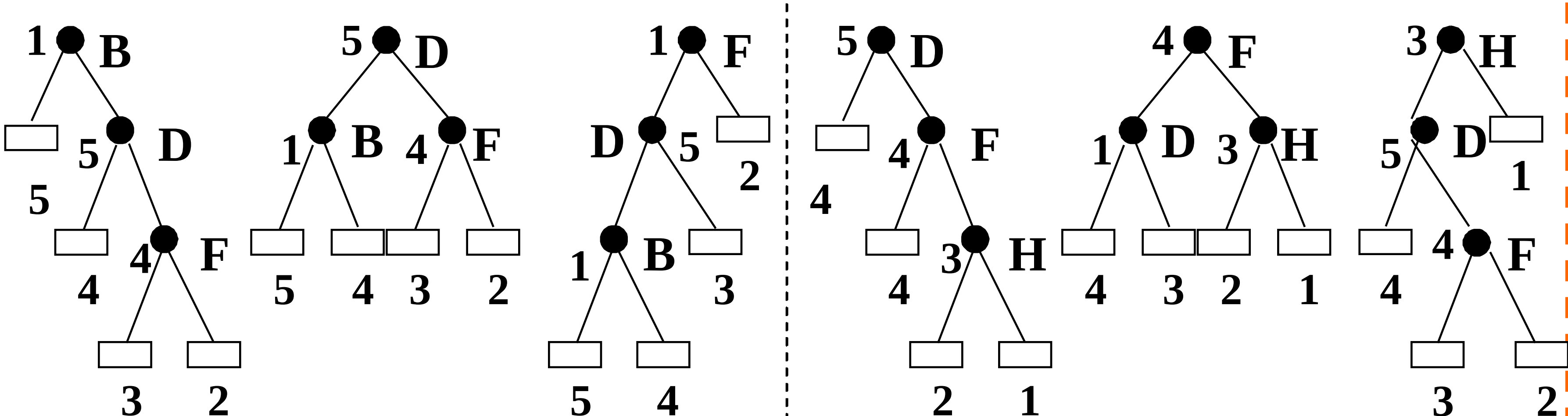
花费	$\boxed{10}$	$\boxed{12}$	$\boxed{9}$	$\boxed{6}$
总权	10	12	9	6

第二步



花费	30	28	27	30	19	22
总权	18	18	18	18	13	13

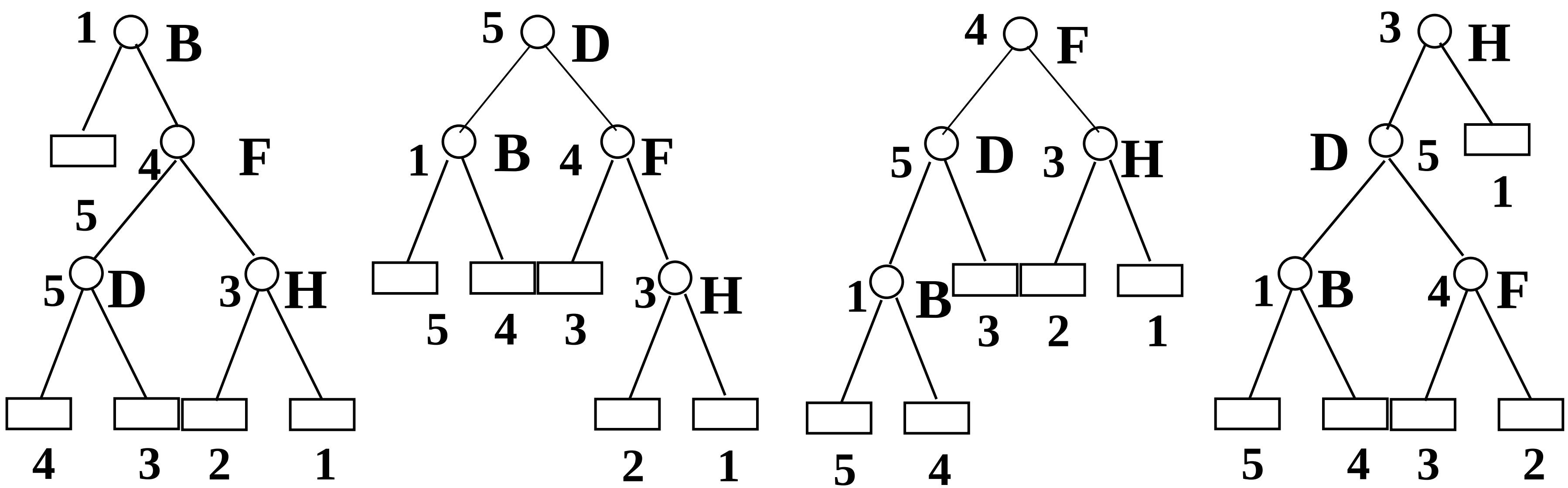
第三步



花费	51	43	52	41	40	49
总权	24	24	24	22	22	24

最佳二叉搜索树 t(i, j)

第四步



花费	68	57	62	71
总权	28	28	28	28

构造最佳二叉搜索树算法

```
void OptimalBST(int a[], int b[], int n, int c[N+1][N+1], int r[N+1][N+1], int w[N+1][N+1]) {
   for (int i=0;i<=n;i++)
      for (int j=0;j<=n;j++)  {                           // 初始化
            c[i][j] = 0;  r[i][j] = 0; w[i][j] = 0;
      }
      for (i = 0; i <=n; i++)  {
            w[i][i] = b[i];
            for(int j = i+1; j <= n; j++)                 // 求出权和w[i..j]
               w[i][j] = w[i][j-1] + a[j] + b[j];
      }
      for (int j=1; j<=n; j++)   {                        // 确定一个结点的BestBST
            c[j-1][j] = w[j-1][j];
            r[j-1][j] = j;
      }
```

构造最佳二叉搜索树算法

```
int m, k0, k;
for (int d=2; d<=n; d++)  {              // 确定d个结点的最佳二叉树
        for (int j = d; j <= n; j++)  {
                i = j - d;
                m = c[i+1][j];
                k0 = i + 1;
                for (k = i+2; k <= j; k++)  {
                        if (c[i][k-1] + c[k][j] < m)  {
                                m = c[i][k-1] + c[k][j];
                                k0 = k;
                        }
                }
                c[i][j] = w[i][j] + m;
                r[i][j] = k0;
        }
}
}
```

如何动态地保持最佳?

- 问题： 静态，经过若干次插入、删除后可能会失去平衡，检索性能变坏

如何动态保持一棵二叉检索树的平衡？

平衡树技术