数据结构与算法
（六）
高级数据结构基础

杜育根

Ygdu@sei.ecnu.edu.cn

本章内容

介绍树的基本概念，并介绍一个在竞赛中很常见的数据结构——并查集的原理和用法，最后重点介绍二叉树概念、二叉树存储与遍历，以及典型的二叉树应用。

一、树的概念

二、并查集

- 合并的优化、路径压缩

三、二叉树

- 二叉树的存储和遍历
- 二叉搜索树、Treap树、伸展树Splay

一、树的基本概念

树（tree）

- 树状图是一种数据结构，它是由n（n>=0）个有限结点组成一个具有层次关系的集合。把它叫做“树”是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：
- 每个结点有零个或多个子结点；没有父结点的结点称为根结点；每一个非根结点有且只有一个父结点；除了根结点外，每个子结点可以分为多个不相交的子树；

定义

定义一：树（tree）是包含n（n>=0）个结点的有穷集，其中：

（1）每个元素称为结点（node）；

（2）有一个特定的结点被称为根结点或树根（root）。

（3）除根结点之外的其余数据元素被分为m（m≥0）个互不相交的集合T1，T2，……Tm-1，其中每一个集合Ti（1<=i<=m）本身也是一棵树，被称作原树的子树（subtree）。

定义二：树是由根结点和若干颗子树构成的。树是由一个集合以及在该集合上定义的一种关系构成的。集合中的元素称为树的结点，所定义的关系称为父子关系。父子关系在树的结点之间建立了一个层次结构。在这种层次结构中有一个结点具有特殊的地位，这个结点称为该树的根结点，或称为树根。

定义三（递归定义）：

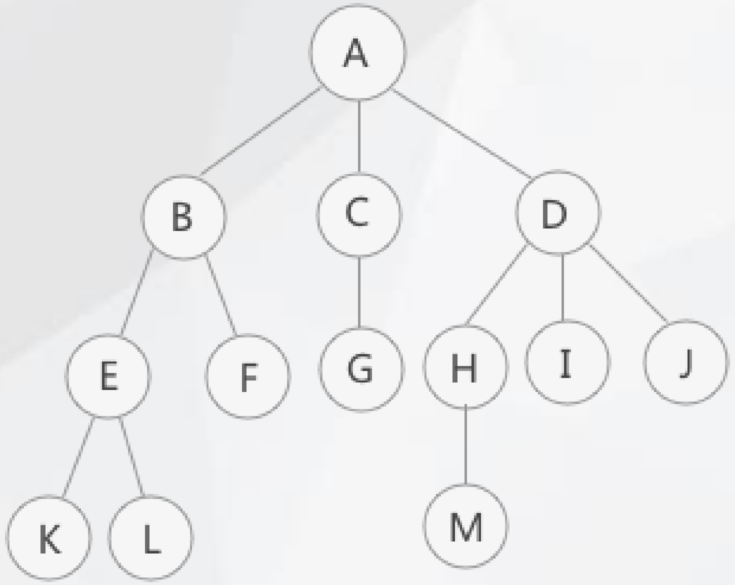
（1）单个结点是一棵树，树根就是该结点本身。

（2）设T1,T2,..,Tk是树，它们的根结点分别为n1,n2,..,nk。用一个新结点n作为n1,n2,..,nk的父亲，则得到一棵新树，结点n就是新树的根。我们称n1,n2,..,nk为一组兄弟结点，它们都是结点n的子结点。我们还称T1,T2,..,Tk为结点n的子树。

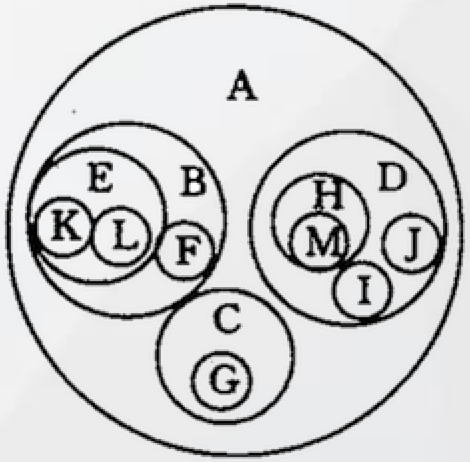
树的相关概念

- 空集合也是树，称为**空树**。空树中没有结点。
- **结点的度**：一个结点含有的子结点的个数称为该结点的度；
- **叶结点或终端结点**：度为0的结点称为叶结点；
- **非终端结点或分支结点**：度不为0的结点；
- **双亲结点或父结点**：若一个结点含有子结点，则这个结点称为其子结点的父结点；
- **孩子结点或子结点**：一个结点含有的子树的根结点称为该结点的子结点；
- **兄弟结点**：具有相同父结点的结点互称为兄弟结点；
- **树的度**：一棵树中，最大的结点的度称为树的度；
- **结点的层次**：从根开始定义起，根为第1层，根的子结点为第2层，以此类推；
- **树的高度或深度**：树中结点的最大层次；
- **堂兄弟结点**：双亲在同一层的结点互为堂兄弟；
- **结点的祖先**：从根到该结点所经分支上的所有结点；
- **子孙**：以某结点为根的子树中任一结点都称为该结点的子孙。
- **森林**：由m（m>=0）棵互不相交的树的集合称为森林；

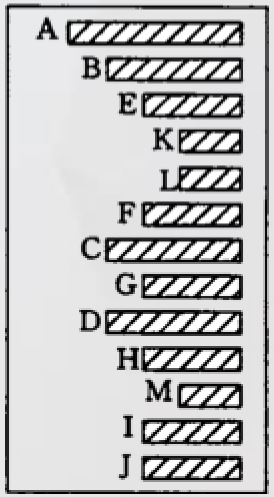
树的表示



一、用图表示



三、以嵌套的集合的形式表示（集合之间绝不能相交，即图中任意两个圈不能相交）

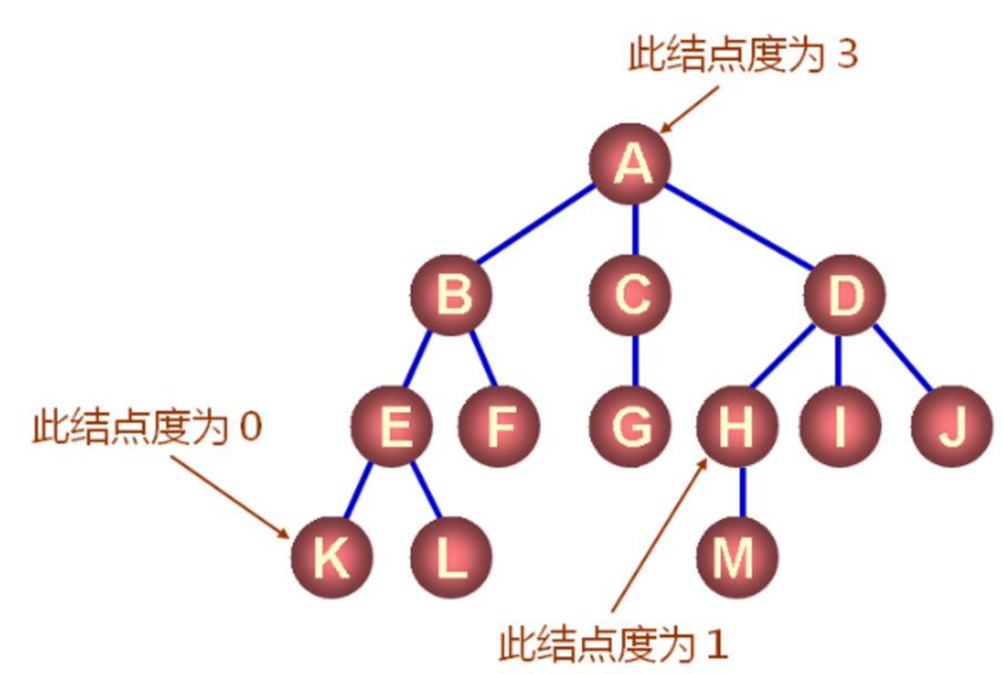


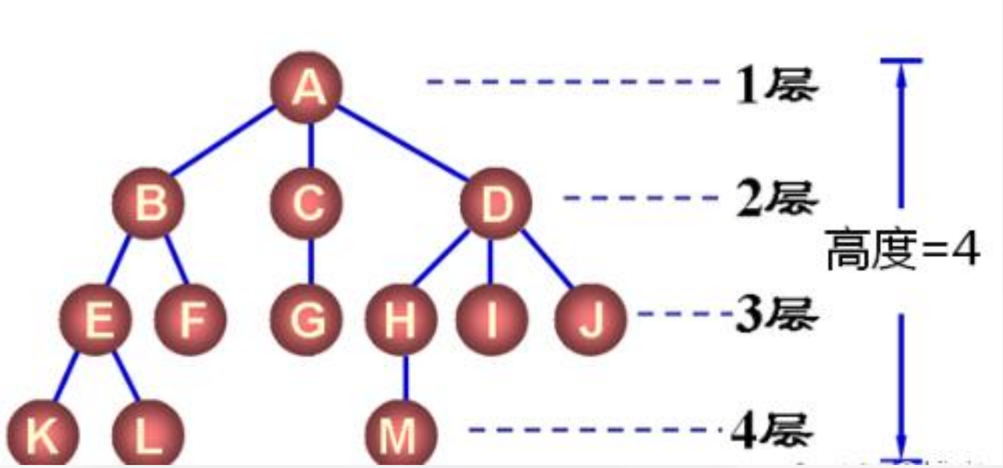
四、凹入表示法，表示方式是：最长条为根结点，相同长度的表示在同一层次。例如 B、C、D 长度相同，都为 A 的子结点，E 和 F 长度相同，为 B 的子结点，K 和 L 长度相同，为 E 的子结点，依此类推。

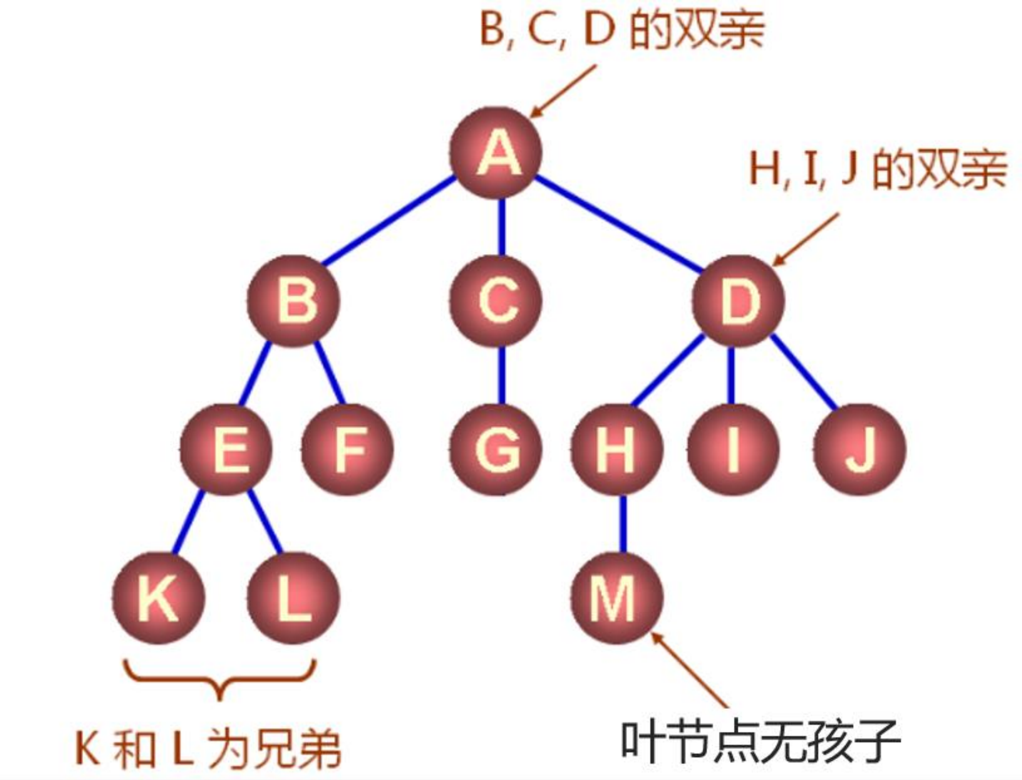
二、用广义表表示：

(A , (B (E (K , L) , F) , C (G) , D (H (M) , I , J)))

概念图示


此结点度为 3
A
B
C
D
E
F
G
H
I
J
K
L
M
此结点度为 0
此结点度为 1


A
B
C
D
E
F
G
H
I
J
K
L
M
1层
2层
3层
4层
高度=4


B, C, D 的双亲
H, I, J 的双亲
A
B
C
D
E
F
G
H
I
J
K
L
M
K 和 L 为兄弟
叶节点无孩子

树的种类

- 无序树：树中任意节点的子结点之间没有顺序关系，这种树称为无序树,也称为自由树;
- 有序树：树中任意节点的子结点之间有顺序关系，这种树称为有序树；
- 二叉树：每个节点最多含有两个子树的树称为二叉树；
- 完全二叉树
- 满二叉树
- 哈夫曼树：带权路径最短的二叉树称为哈夫曼树或最优二叉树；

树的三种最重要的遍历

- 树的3种最重要的遍历方式分别称为前序遍历、中序遍历和后序遍历。
- 以这3种方式遍历一棵树时，若按访问结点的先后次序将结点排列起来，就可分别得到树中所有结点的前序列表，中序列表和后序列表。
- 设T它以n为树根，树根的子树从左到右依次为T1,T2,..,Tk，那么有：
- 对T进行前序遍历是先访问树根n,然后依次前序遍历T1,T2,..,Tk。
- 对T进行中序遍历是先中序遍历T1，然后访问树根n，接着依次对T2,T3,..,Tk进行中序遍历。
- 对T进行后序遍历是先依次对T1,T2,..,Tk进行后序遍历，最后访问树根n。

一些特殊的树：

- **表达式树**：树叶是操作数，如常量或变量。其他的节点是操作符。
- **二叉查找树**：对于树中的每个节点X，它的左子树中所有关键字值小于X的关键字值，它的右子树中所有关键字值大于X的关键字值。
- **AVL树（平衡二叉树或平衡树）**：每个节点的左子树和右子树的高度最多差1的二叉查找树，并且左右两个子树也都是一棵平衡二叉树。
- **红黑树**：是一种自平衡二叉树，在平衡二叉树的基础上每个节点又增加了一个颜色的属性，节点的颜色只能是红色或黑色。
- **伸展树**：它保证从空树开始任意连续M次对树的操作最多花费O(M*logN)时间。
- **B-树**：一棵m阶B树(balanced tree of order m)是一棵平衡的m路搜索树。
- **Trie树（单词查找树，字典树）**：字典树是一种以树形结构保存大量字符串。以便于字符串的统计和查找，经常被搜索引擎系统用于文本词频统计。
- **后缀树**：是用来支持有效的字符串匹配和查询。

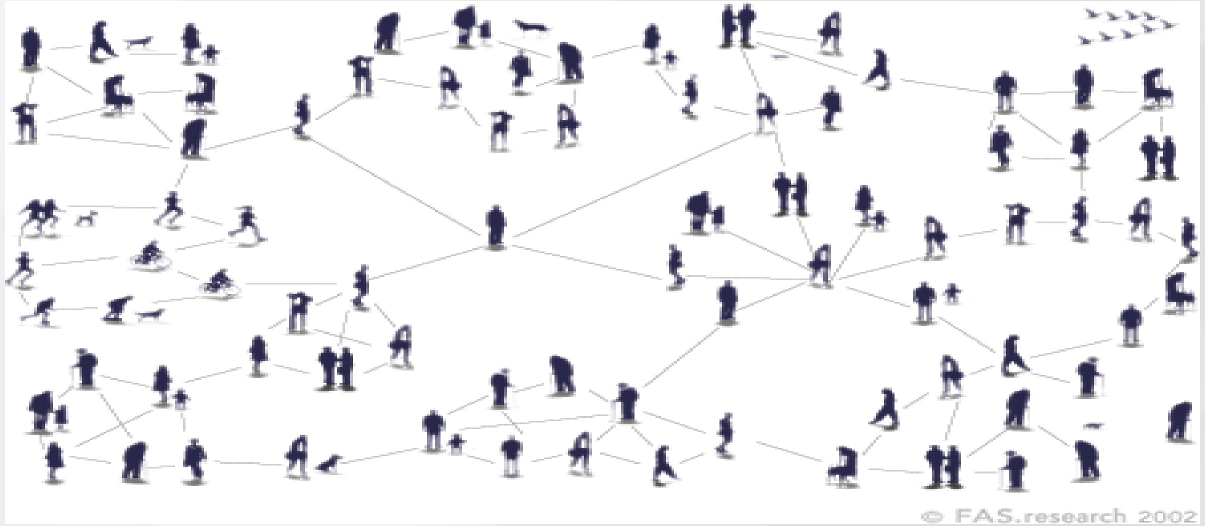
二、并查集

并查集

- 并查集，顾名思义，合并 查找 集合
- 这是一种树型的数据结构，用于处理一些不相交集合（Disjoint Sets）的合并和查询问题。在使用中常常以森林来表示。
- 并查集也是用来维护集合的，和前面学习的set不同之处在于，并查集能很方便地同时维护很多集合。如果用set来维护会非常的麻烦。并查集的核心思想是记录每个结点的父亲结点（代表所在集合）是哪个结点。

应用背景：“朋友群”

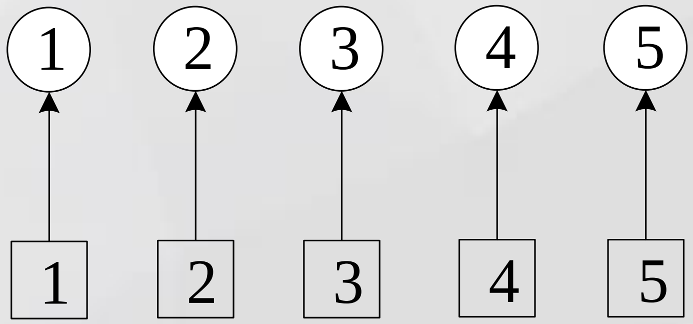
- 一个城市中有n个人，他们属于不同的朋友群；
- 已知这些人的关系，假设1号、2号是朋友关系，1号、3号也是朋友关系，那么他们都属于一个关系群；
- 问有多少关系群，每人属于哪个关系群。
- 用**并查集**可以很简洁地表示这个关系。



（1）初始化

- 定义**father[i]是结点i有关系的父节点**，并查集。
- 初始化：令father[i]=i，则认为这个结点是当前集合根结点。“一人一群，我就是我！”

father[i]	1	2	3	4	5
i	1	2	3	4	5

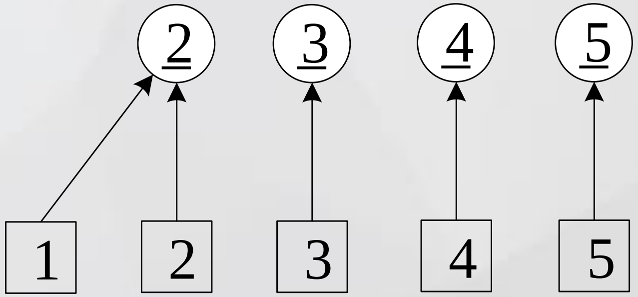


(2）合并

例：加入第一个朋友关系(1, 2)。

- **在并查集s中，把结点1合并到结点2，也就是把结点1的父节点1改成结点2的父节点2。**

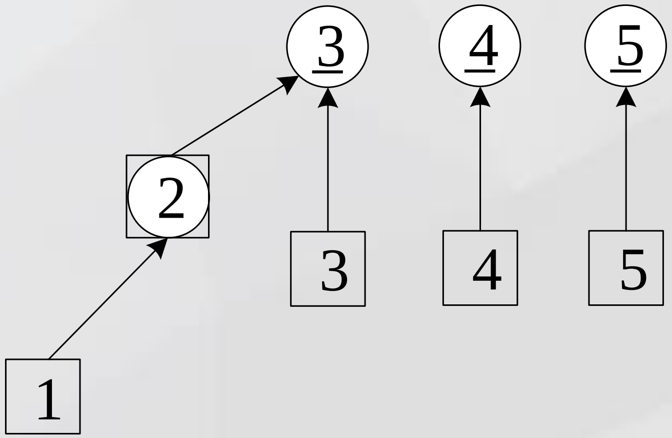
father[i]	2	2	3	4	5
i	1	2	3	4	5



加入第二个朋友关系(1, 3)：

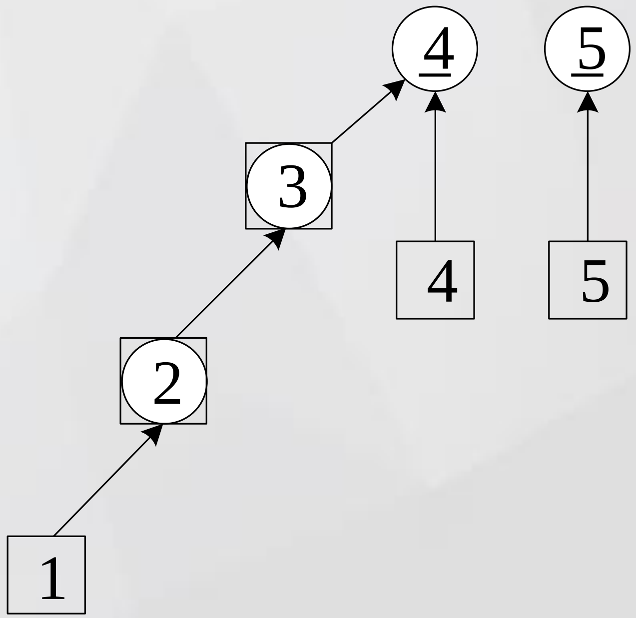
- 查找结点1的父节点，是2，递归查找元素2的父节点是2；
- 把元素2的集2合并到结点3的集3。此时，结点1、2、3都属于一个集。

father[i]	2	**3**	3	4	5
i	1	2	3	4	5



加入第三个朋友关系(2, 4)：

father[i]	2	3	**4**	4	5
i	1	2	3	4	5



（3）查找

- 查找元素的集，是一个递归的过程，直到元素的值和它的集相等，就找到了根结点的集。
- 这棵搜索树，可能很深，复杂度是O(n)，变成了一个链表，出现了树的“退化”现象。

并查集操作

1）初始化：初始的时候每个结点各自为一个集合，father[i] 表示结点 i的父亲结点，如果father[i]=i，则认为这个结点是当前集合根结点。

```
void init() {
    for (int i = 1; i <= n; ++i)
        father[i] = i;
}
```

2) 查找：查找结点所在集合的根结点，结点 x的根结点必然也是其父亲结点的根结点。

```
int get(int x) {        // return x==father[x]? x:get(father[x]);
    if (father[x] == x)  // x 结点就是根结点
        return x;
    return get(father[x]); // 返回父结点的根结点
}
```

3) 合并：将两个元素所在的集合合并在一起，通常，合并之前先判断两个元素是否属于同一集合。

```
void merge(int x, int y) {
    x = get(x);
    y = get(y);
    if (x != y) // 不在同一个集合
        father[y] = x;      //或者father[y]=father[x];
}
```

复杂度

- 前面的并查集的复杂度实际上在有些极端情况会很慢。比如树的结构正好是一条链，那么最坏情况下，每次查询的复杂度达到了O(n)。这并不是我们期望的结果。
- 合并的搜索深度是树的长度，复杂度最坏也是O(n)。
- 性能差。

- 能优化吗?

  目标：优化之后，复杂度 < O(logn)。

合并的优化

- 合并元素x和y时，先搜到它们的根结点；
- 合并这两个根结点：把一个根结点的父节点改成另一个根结点。
- 这两个根结点的高度不同，把高度较小的集合并到较大的集上，能减少树的高度。

合并优化

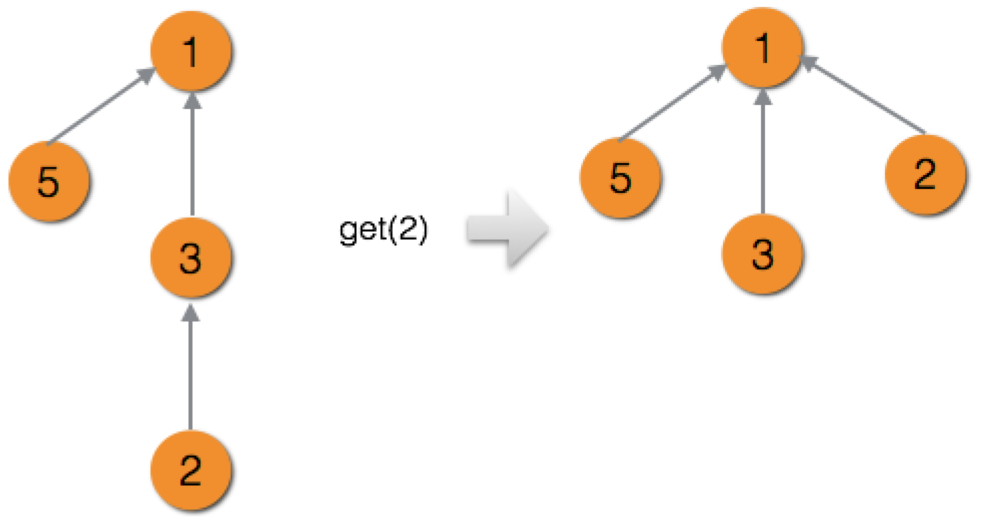
```
int height[maxn];         //用height[i]定义元素i的高度
void init (){
   for(int i = 1; i <= maxn; i++){
        father[i] = i;
        height[i]=0;       //初始化树的高度
   }
}
void merge(int x, int y){    //优化合并操作
    x = get(x);
    y = get(y);
    if (height[x] == height[y]) {
        height[x] = height[x] + 1;  //合并，树的高度加1
        father[y] = x;
    }
    else{      //把矮树并到高树上，高树的高度保持不变
        if (height[x] < height[y])  father[x] = y;
        else   father[y] = x;
    }
}
```

路径压缩

- 路径压缩 的思想是，我们只关心每个结点的父结点，而并不太关心树的真正的结构。这样我们在一次查询的时候，可以把查询路径上的所有结点的father[i] 都赋值成为根结点。只需要在我们之前的查询函数上面进行很小的改动。

```
int get(int x) {
    if (father[x] == x) { // x 结点就是根结点
        return x;
    }
    return father[x] = get(father[x]);
// 返回父结点的根结点，并令当前结点父结点
//直接为根结点
}
```

下面图片是路径压缩前后的对比。



- 优化：沿路径返回时，顺便把i所属的父节点改成根结点。下次再搜，复杂度是O(1)。
- 路径压缩：整个搜索路径上的元素，在递归过程中，从元素i到根结点的所有元素，它们所属的父节点都被改为根结点。路径压缩不仅优化了下次查询，而且也优化了合并，因为合并时也用到了查询。
- 路径压缩在实际应用中效率很高，其一次查询复杂度平摊下来可以认为是一个常数。并且在实际应用中，我们基本都用带路径压缩的并查集。

路径压缩：非递归实现

如果数据规模太大，用递归担心爆栈，可以这样写：

```
int get(int x){
    int r = x;
    while ( father[r] != r ) r=father[r]; //找到x的根结点r
    int i = x, j;
    while(i != r){
        j = father[i]; //用临时变量j记录
        father[i]= r ; //把路径上元素x的父节点改为根结点
        i = j;           //当前节点上升一级
    }
    return r;
}
```

例：hdu 1213 How Many Tables

- 有n个人一起吃饭，有些人互相认识。认识的人想坐在一起，而不想跟陌生人坐。例如A认识B，B认识C，那么A、B、C会坐在一张桌子上。给出认识的人，问需要多少张桌子。
- 输入格式：首行整数T（1 <= T <= 25）开头，该整数表示测试用例的数量。然后是T测试用例。每个测试用例均以两个整数N和M（1 <= N，M <= 1000）开头。N表示朋友的数量，朋友从1到N标记。然后跟随M行。每行包含两个整数A和B（A！ = B），这意味着朋友A和朋友B彼此认识。两种情况之间将有一个空白行。
- 输出格式：对于每个测试用例，只需输出至少需要多少张桌。请勿打印任何空格。
- 样本输入
- 2
- 5 3
- 1 2
- 2 3
- 4 5

- 5 1
- 2 5
- 样本输出
- 2
- 4

```cpp
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1050;
int father[maxn];
int height[maxn];

void init(){  //初始化
   for(int i = 1; i <= maxn; i++){
        father[i] = i;
        height[i]=0;                    //树的高度
   }
}

int get(int x){  //查询
    if(x == father[x]) return father[x];
    father[x] = get(father[x]);   //路径压缩
}
```

```cpp
void merge(int x, int y){        //优化合并操作
    x = get(x);
    y = get(y);
    if (height[x] == height[y]) {
        height[x] = height[x]+1; //树的高度加一
        father[y] = x;//合并
    }
    else{   //把矮树并到高树上，高的树的高度不变
        if (height[x] < height[y])  father[x] = y;
        else   father[y] = x;
    }
}
```

```
int main (){
    int t, n, m, x, y;
    cin >> t;
    while(t--){
        cin >> n >> m;
        init();
        for(int i = 1; i <= m; i++){
            cin >> x >> y;
            merge(x, y);
        }
        int ans = 0;
        for(int i = 1; i <= n; i++)   //统计有多少个集
            if(father[i] == i)
                ans++;
        cout << ans <<endl;
    }
    return 0;
}
```

带权并查集

- 所谓带权并查集，是指结点存有权值信息的并查集。并查集以森林的形式存在，而结点的权值，大多是记录该结点与祖先关系的信息。比如权值可以记录该结点到根结点的距离。
- 例题：在排队过程中，初始时，一人一列。一共有如下两种操作：
- 合并：令其中的两个队列 A,B合并，也就是将队列 A排在队列 B的后面
- 查询：询问某个人在其所在队列中排在第几位

例题解析　我们不妨设 size[]为集合中的元素个数，dist[]为元素到队首的距离，合并时，dist[A.root]需要加上 size[B.root]]（每个元素到队首的距离应该是到根路径上所有点的 dist[]求和），size[B.root]size需要加上 size[A.root]（每个元素所在集合的元素个数只需查询该集合中根的 size[x.root]。

1) 初始化:

```
void init() {
  for(int i = 1; i <= n; i++) {
        father[i] = i, dist[i] = 0, size[i] = 1;
    }
}
```

2) 查找

查找元素所在的集合，即根结点。

```
int get(int x) {
    if(father[x] == x)  return x;
    int y = father[x];
    father[x] = get(y);
    dist[x] += dist[y];
// x 到根结点的距离等于 x 到之前父亲结点距离加上之前父亲结点到根结点的距离
    return father[x];
 }
```

路径压缩的时候，不需考虑 size[]，但 dist[] 需要更新成到整个集合根的距离。

3) 合并

将两个元素所在的集合合并为一个集合。 通常来说，合并之前，应先判断两个元素是否属于同一集合，这可用上面的“查找”操作实现。

```
void merge(int a, int b) {
    a = get(a);
    b = get(b);
    if(a != b) {  // 判断两个元素是否属于同一集合
        father[a] = b;
        dist[a]  = size[b];
        size[b]  += size[a];
    }
}
```

通过小小的改动，我们就可以查询并查集这一森林中，每个元素到祖先的相关信息。

练习题：朋友

- 在社交的过程中，通过朋友，也能认识新的朋友。在某个朋友关系图中，假定 A 和 B 是朋友，B 和 C 是朋友，那么 A 和 C 也会成为朋友。即，我们规定朋友的朋友也是朋友。现在，已知若干对朋友关系，询问某两个人是不是朋友。请编写一个程序来解决这个问题吧。
- 输入格式：第一行：三个整数 n,m,p(n≤5000,m≤5000,p≤5000)分别表示有n 个人，m 个朋友关系，询问p 对朋友关系。接下来 m 行：每行两个数 Ai、Bi,1≤Ai,Bi≤n，表示Ai 和 Bi具有朋友关系。接下来 p 行：每行两个数，询问两人是否为朋友。
- 输出格式：输出共 p 行，每行一个Yes或No。表示第 i个询问的答案为是否朋友。
- 样例输入
- 6 5 3
- 1 2
- 1 5
- 3 4
- 5 2
- 1 3
- 1 4
- 2 3
- 5 6

样例输出

Yes

Yes

No

练习题：网络交友

- 在网络社交的过程中，通过朋友，也能认识新的朋友。在某个朋友关系图中，假定 A 和 B 是朋友，B 和 C 是朋友，那么 A 和 C 也会成为朋友。即，我们规定朋友的朋友也是朋友。现在要求你每当有一对新的朋友认识的时候，你需要计算两人的朋友圈合并以后的大小。
- 输入格式：第一行：一个整数 n(n≤5000)，表示有 n 对朋友认识。
- 接下来 n 行：每行输入两个名字。表示新认识的两人的名字，用空格隔开。（名字是一个首字母大写后面全是小写字母且长度不超过 20 的串）。
- 输出格式：对于每一对新认识的朋友，输出合并以后的朋友圈的大小。
- 样例输入
- 3
- Fred Barney
- Barney Betty
- Betty Wilma
- 样例输出
- 2
- 3
- 4

练习题：找出所有谎言

小明有很多卡片，每张卡片正面上印着“剪刀”，“石头”或者“布”三种图案中的一种，反面则印着卡片的序号。“剪刀”，“石头”和“布”三种构成了一个有趣的环形，“剪刀”可以战胜“布”，“布”可以战胜“石头”，“石头”可以战胜“剪刀”。

现有 N 张卡片，以 1-N 编号。每张卡片印着“剪刀”，“石头”，“布”中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 张卡片所构成的关系进行描述：第一种说法是“1 X Y”，表示 X 号卡片和 Y 号卡片是同一种卡片。第二种说法是“2 X Y”，表示 X 号卡片可以战胜 Y 号卡片。

小明对 N张卡片，用上述两种说法，一句接一句地说出 K 句话，这K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。 1) 当前的话与前面的某些真的话冲突，就是假话； 2) 当前的话中 X 或 Y 的值比 N 大，就是假话； 3) 当前的话表示 X 能战胜 X，就是假话。

你的任务是根据给定的 N 和 K 句话，计算假话的总数。

输入格式：第一行是两个整数N(1≤N≤50,000) 和K(0≤K≤100,000)，以一个空格分隔。以下 K 行每行是三个正整数 D，X，Y，两数之间用一个空格隔开，其中 D 表示说法的种类。 若 D=1，则表示 X 和 Y 是同一种卡片。 若 D=2，则表示 X 能战胜 Y。

输出格式：只有一个整数，表示假话的数目。

样例输入

100 7

1 101 1

2 1 2

2 2 3

2 3 3

1 1 3

2 3 1

1 5 5

样例输出

2

3

4

练习题：接龙

- 小明在玩一种接龙的游戏，小明有30000张卡片分别放在30000列，每列依次编号为 1,2,...,30000。同时，小明也把每张卡片依次编号为1,2,...,30000。游戏开始，小明让让第 i 张卡片处于第 i(i=1,2,...,30000) 列。然后小明会发出多次指令，每次调动指令 M i j会将第 i张卡片所在的队列的所有卡片，作为一个整体（头在前尾在后）接至第 j张卡片所在的队列的尾部。小明还会查看当前的情况，发出 C i j 的指令，即询问电脑，第 i张卡片与第 j张卡片当前是否在同一个队列中，如果在同一列中，那么它们之间一共有多少张卡片。
- 聪明的你能不能编写程序处理小明的指令，以及回答小明的询问呢?
- 输入格式：第一行有一个整数 T（1≤T≤500000），表示总共有 T 条指令。以下有 T 行，每行有一条指令。指令有两种格式：M i j：i和 j是两个整数（1≤i,j≤30000），表示指令涉及的卡片编号。你需要让第 i张卡片所在的队列的所有卡片，作为一个整体（头在前尾在后）接至第 j张卡片所在的队列的尾部，输入保证第 i号卡片与第 j号卡片不在同一列。C i j：i 和j是两个整数（1≤i,j≤30000），表示指令涉及的卡片编号。该指令是小明的询问指令。
- 输出格式：如果是小明调动指令，则表示卡片排列发生了变化，你的程序要注意到这一点，但是不要输出任何信息；如果是小明的询问指令，你的程序要输出一行，仅包含一个整数，表示在同一列上，第 i号卡片与第 j号卡片之间的卡片数目（不包括第 i张卡片和第 j张卡片）。如果第 i号卡片与第 j号卡片当前不在同一个队列种中，则输出 -1。

样例输入	样例输出
4 M 2 3 C 1 2 M 2 4 C 4 2	-1 1

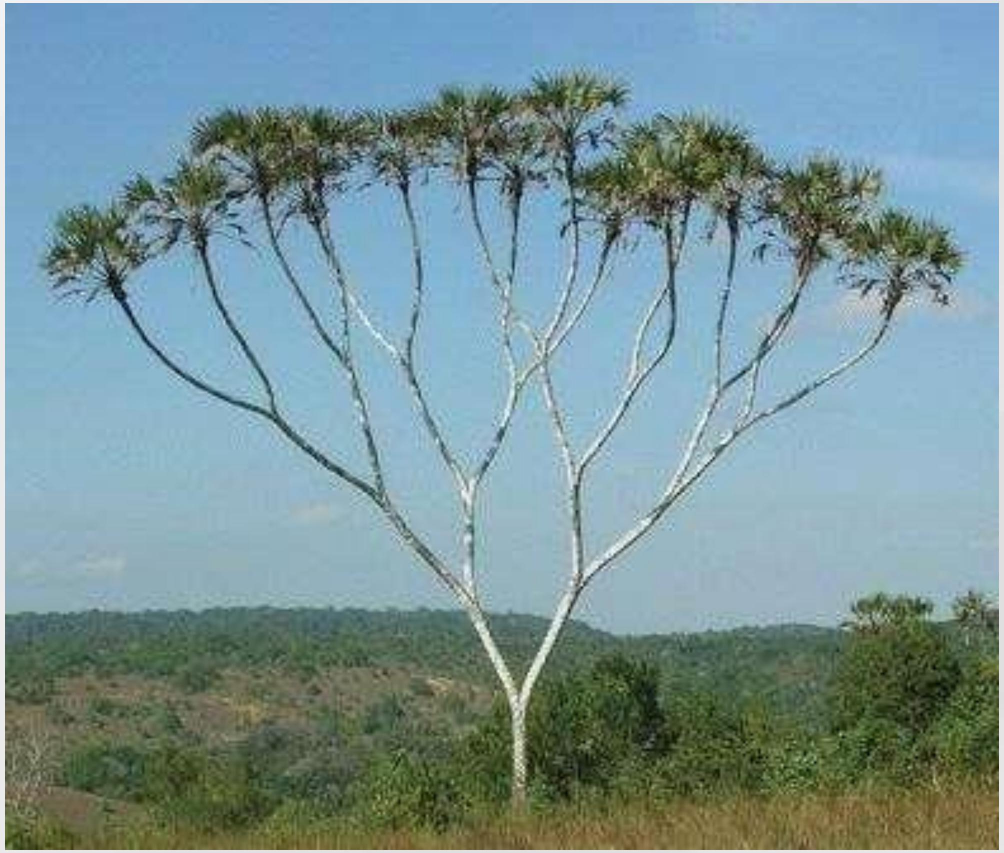
并查集习题

poj 2524 Ubiquitous Religions，并查集简单题。
poj 1611 The Suspects，简单题。
poj 1703 Find them, Catch them。
poj 2236 Wireless Network。
poj 2492 A Bug's Life。
poj 1988 Cube Stacking。
poj 1182食物链，经典题。
hdu 3635 Dragon Balls。
hdu 1856 More is better。
hdu 1272 小希的迷宫。
hdu 1325 Is It A Tree。
hdu 1198 Farm Irrigation。
hdu 2586 How far away，最近公共祖先，并查集+深搜。
hdu 6109 数据分割，并查集+启发式合并。

三、二叉树

◆ 二叉树

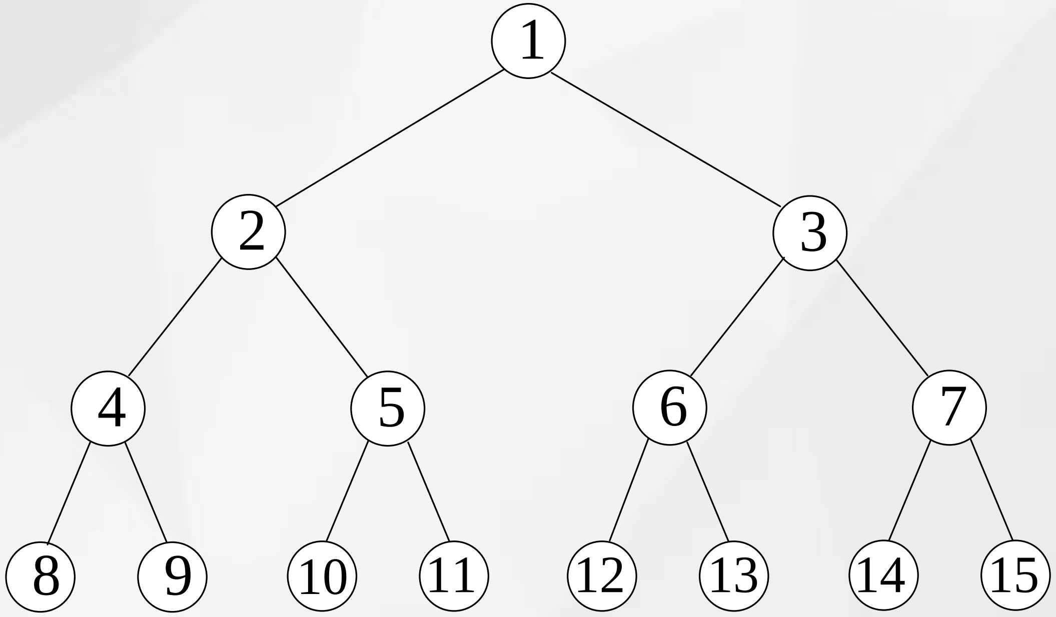
- 二叉树的存储
- 二叉树的遍历
- 二叉搜索树
- Treap树
- Splay树



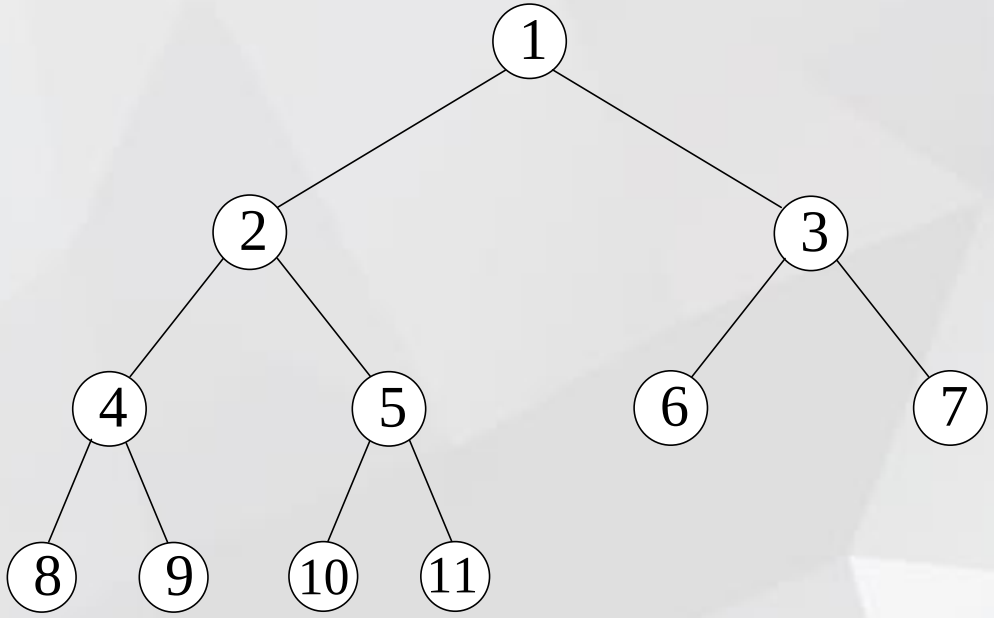
• 二叉树的性质

- 每个结点最多有两个子结点：左孩子、右孩子。以它们为根的子树称为左子树、右子树。
- 二叉树的第i层，最多有2^{i-1}个结点。
- 如果每一层的结点数都是满的，称为**满二叉树**。
- 如果满二叉树只在最后一层有缺失，并且缺失的编号都在最后，那么称为**完全二叉树**。

满二叉树



完全二叉树



二叉树的存储

1.用指针实现。

```
struct node{
    int value;      //结点的值
    node *l, *r;    //指向左右子结点
};
```

2. 用数组实现。

• 二叉树的遍历

1. 宽度优先遍历：一层层地遍历二叉树。用队列实现。

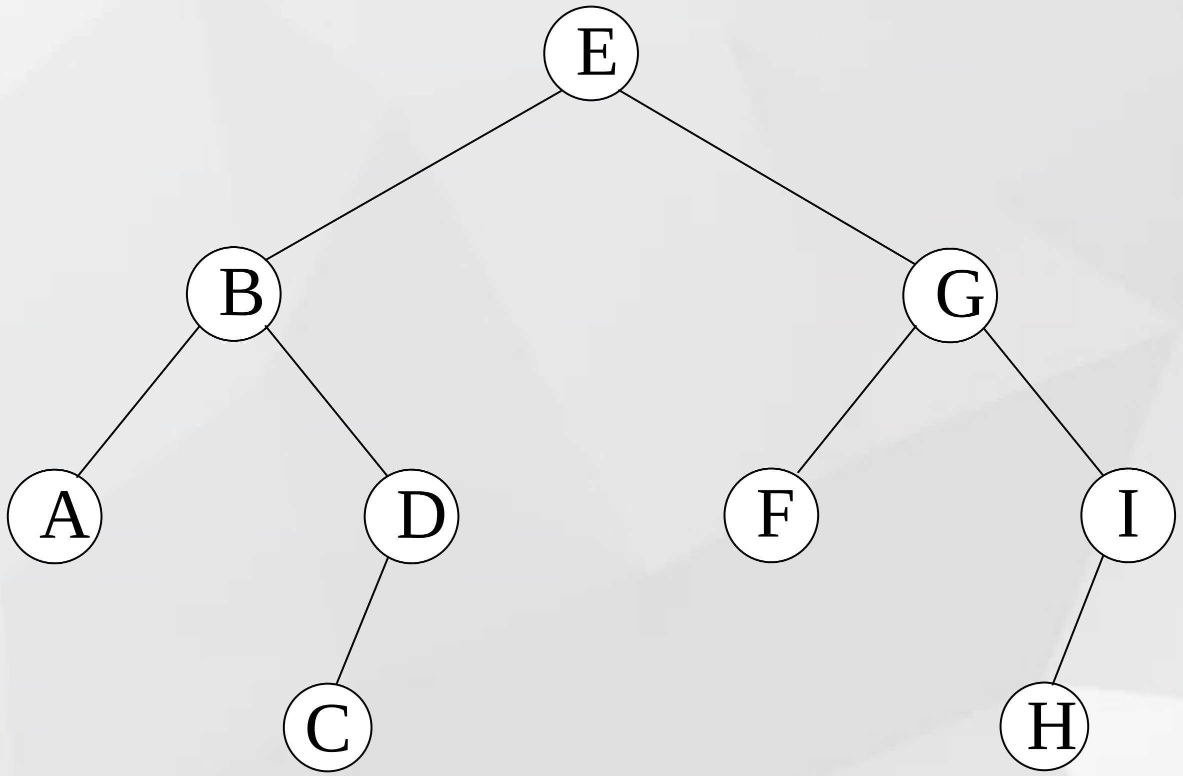
2. 深度优先遍历：更常用的遍历方法。

二叉树的深度优先遍历

- 先（根）序遍历
- 中（根）序遍历
- 后（根）序遍历

先（根）序遍历

- 按父、左儿子、右儿子的顺序访问：
- EBADCGFIH
- 先序遍历的第一个结点是根

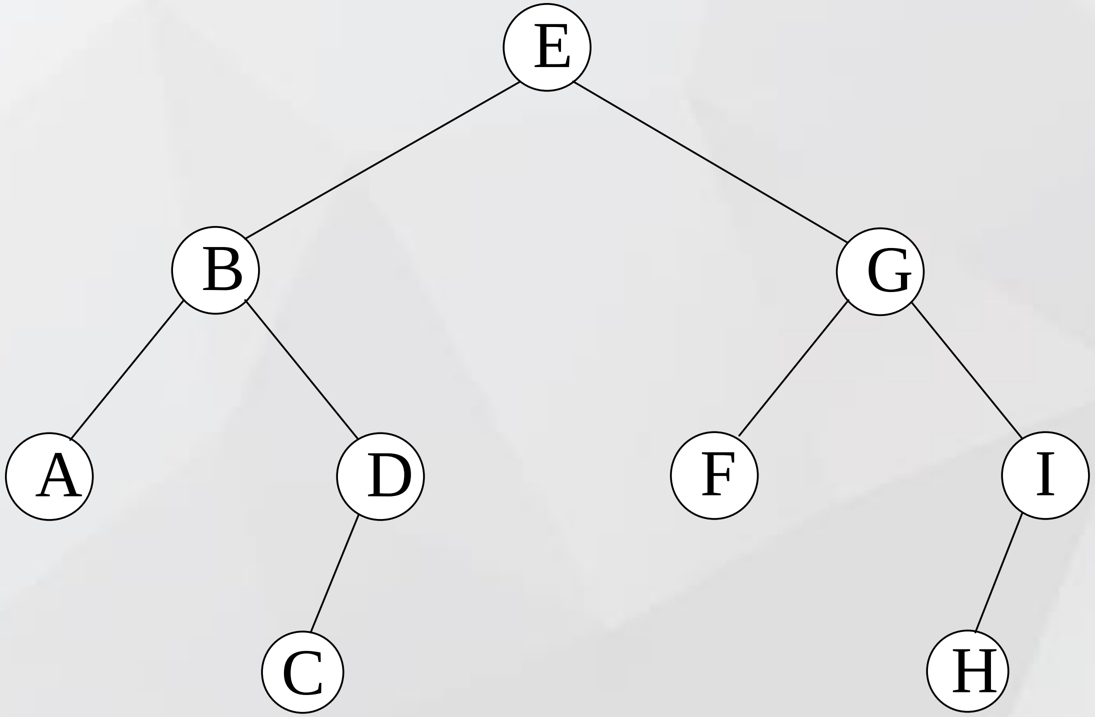


先序遍历的代码：

```
void preorder (node *root){
     cout << root ->value;   //输出
     preorder (root ->l);    //递归左子树
     preorder (root ->r);    //递归右子树
}
```

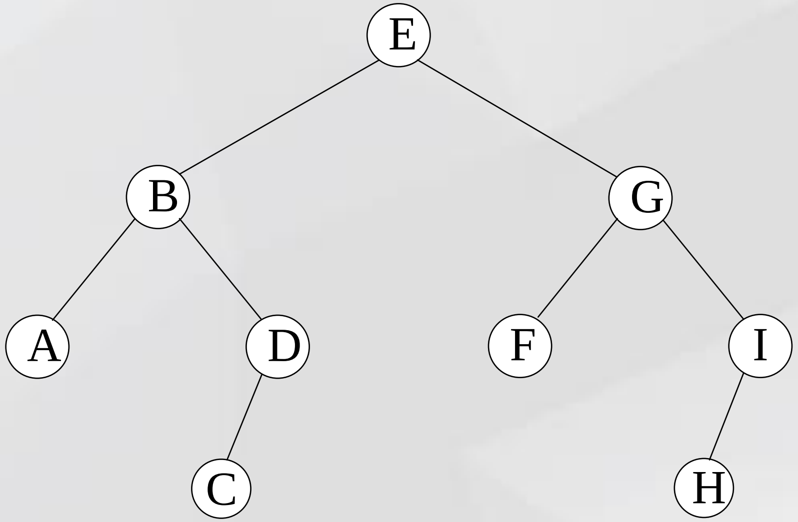
中（根）序遍历

- 按左儿子、**父**、右儿子的顺序访问：
- ABCDEFGHI
- **思考**：结果为什么是字典序？



中序遍历的特点

- ABCDEFGHI
- 返回的结果：根结点左边的点都在左子树上，右边的都在右子树上。
- 例如：E是根，E左边的“ABCD”在它的左子树上；
- 例如：在子树“ABCD”上，B是子树的根，那么“A”在它的左子树上，“CD”在它的右子树上。

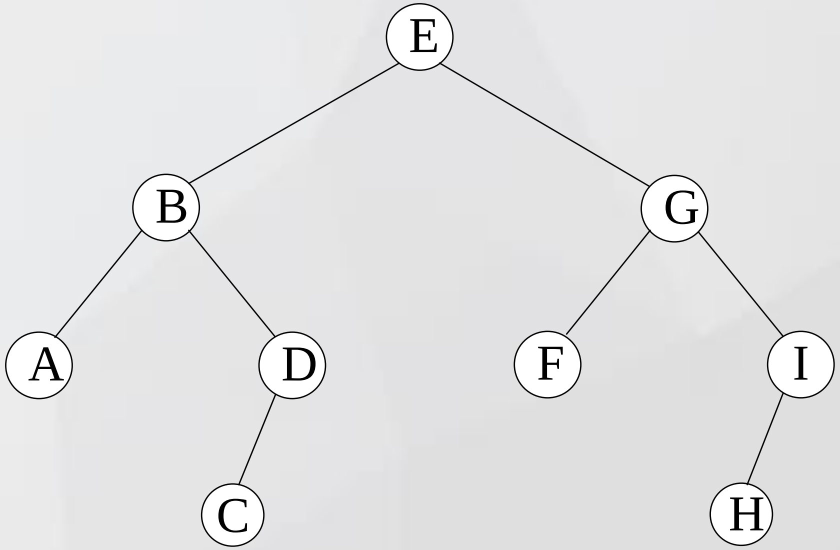


中序遍历代码：

```
void preorder (node *root){
    preorder (root ->l);     //递归左子树
        cout << root ->value;   //输出
        preorder (root ->r);    //递归右子树
}
```

后（根）序遍历

- 按左儿子、右儿子、**父**的顺序访问：
- ACDBFHIGE
- 后序遍历的最后一个结点是根



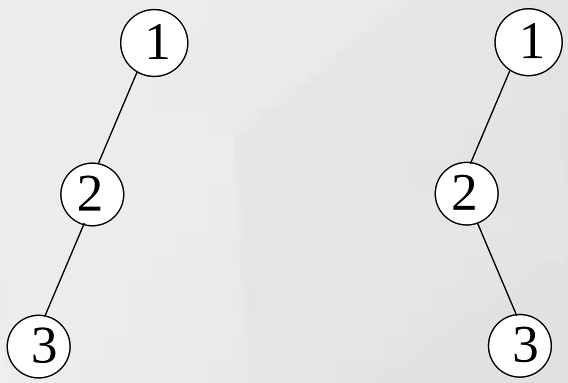
后序遍历代码：

```
void preorder (node *root){
    preorder (root ->l);    //递归左子树
    preorder (root ->r);    //递归右子树
    cout << root ->value;   //输出


}
```

三种遍历的关系

- 已知二叉树的：“中序遍历+先序遍历”，或者“中序遍历+后序遍历”，都能确定一棵树。
- 但是只有“先序遍历+后序遍历”，不能确定一棵树。例如下图，它们的先序遍历都是"1 2 3"，后序遍历都是"3 2 1"。



例题： hdu 1710 Binary Tree Traversals

输入二叉树的先序和中序遍历序列，求后序遍历。

（1）样例输入

先序：1 2 4 7 3 5 8 9 6

中序：4 7 2 1 8 5 9 3 6

（2）样例输出

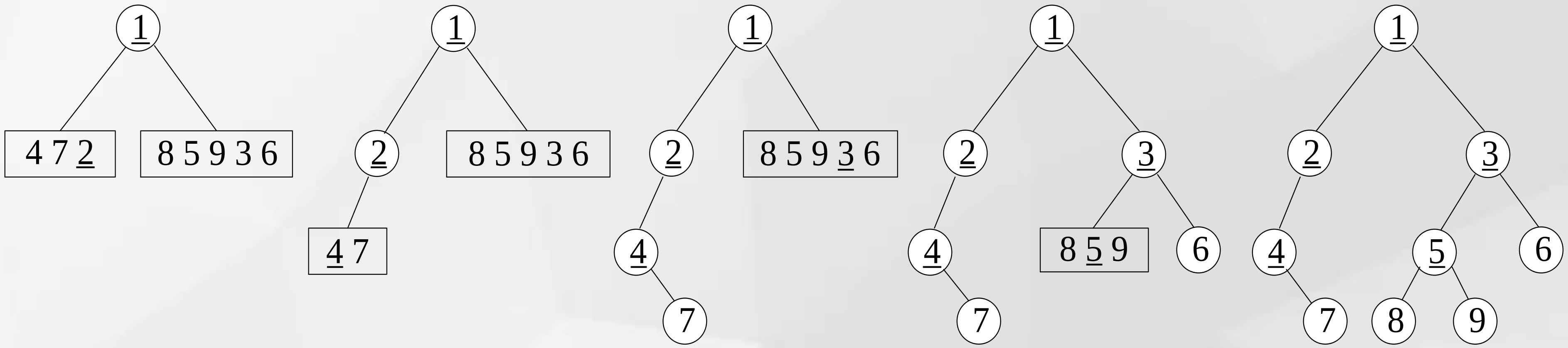
后序：7 4 2 8 9 5 6 3 1

思路

先序：1 2 4 7 3 5 8 9 6　　　中序：4 7 2 1 8 5 9 3 6

（1）先序遍历的第一个数是整棵树的根，例如样例中的“1”。再对照中序遍历，“1”左边的“4 7 2”都在根的左子树上，右边的“8 5 9 3 6”都在根的右子树上。得到下图最左子图。

（2）递归上述过程。



- **代码。**
- **注意其中的函数：**

  先序遍历preorder()

  中序遍历inorder()

  后序遍历postorder()

◆ 二叉搜索树

- BST（Binary Search Tree，二叉搜索树）

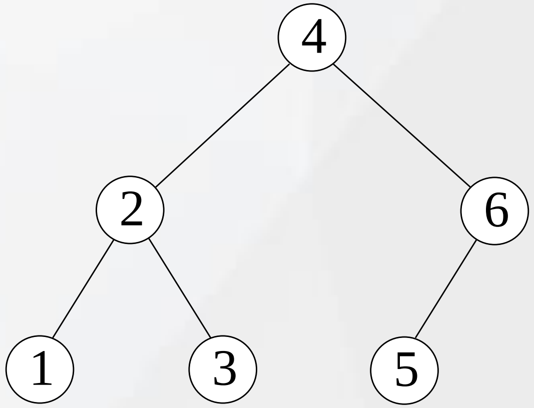
（1）每个元素有唯一的键值，这些键值能比较大小。

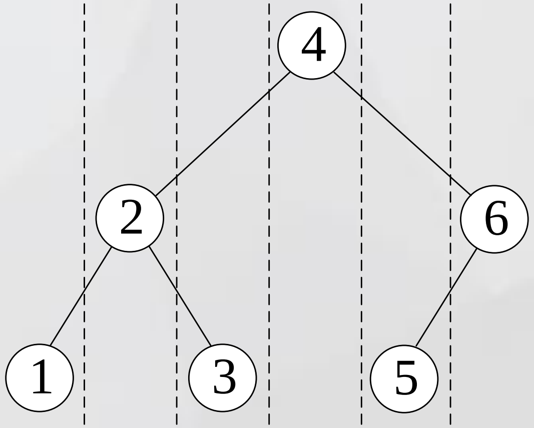
（2）任意一个结点的键值，比它的左子树所有结点的键值大，比它的右子树所有结点的键值小。

- 在BST上，以任意结点为根结点的一棵子树，仍然是BST。

中序遍历与BST

- 用**中序遍历**可以得到BST的有序排列。
- 右图的虚线把结点隔开，结点正好按从小到大的顺序被虚线隔开了。





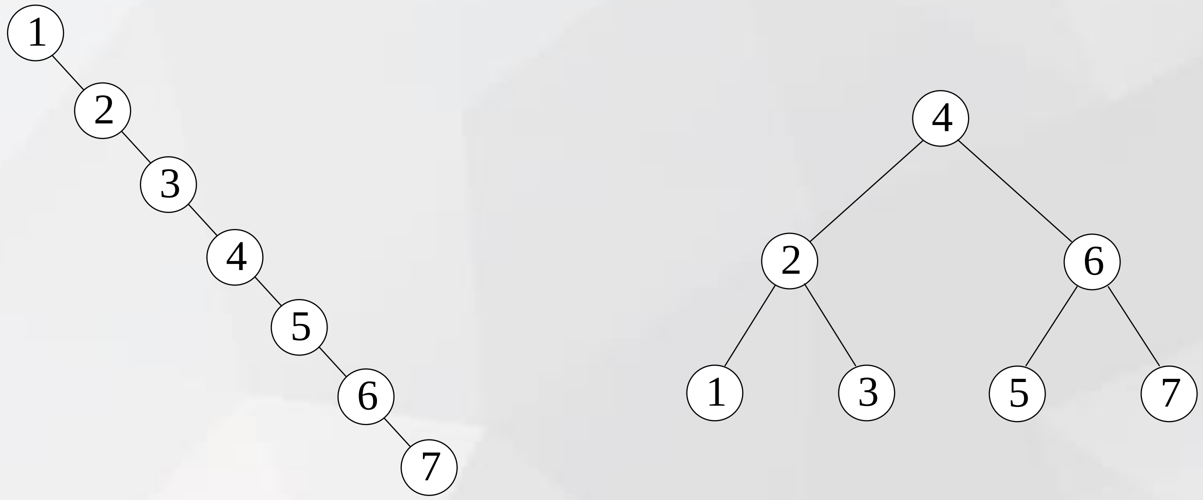
BST的插入、查询、删除、遍历

简单的插入方法：

以第1个数据x为根结点，然后逐个插入其它数据。

如果数据y比根结点x小，就往x的左子树插，否则就往右子树插；如果子树为空就直接放到这个空位，如果非空，就与子树的值进行比较，再进入子树的下一层。

- 这个简单的建树方法，可能导致一个很差的BST。
- 例如{1, 2, 3, 4, 5, 6, 7}，按顺序插入，会全部插到右子树上，见下面左图。
- 右图是期望的BST，它很平衡。



查询、删除、遍历

- 查询：类似于建树过程，递归。
- 删除：删除一个结点x后，剩下的部分应该仍然是一个BST。
- 遍历：中序遍历。

什么是好的BST算法?

- BST的优劣，取决于它是否平衡。
- 如何实现一个平衡的BST？由于无法提前安排元素的顺序，所以只能在建树之后，通过**动态调整**，使得它变得平衡。
- BST算法的区别，就在于用什么办法调整。

- BST算法有：AVL树、红黑树、Splay树、Treap树、SBT树等。

- STL与BST。STL的set和map是用二叉搜索树（红黑树）实现的，检索和更新的复杂度是O(logn)。

习题

- hdu 3999 The order of a Tree，模拟BST的建树和访问。
- hdu 3791 二叉搜索树，模拟BST。
- poj 2418 Hardwood Species，用map快速处理字符串。

◆ Treap树

- Treap =Tree + Heap，树和堆的结合

- 每个结点有2个值：（1）键值；（2）优先级。
- 对于键值来说，这棵树是BST；对于优先级来说，这棵树是一个堆。

- 借助**优先级**这个工具，Treap**简单地**实现了BST的平衡。

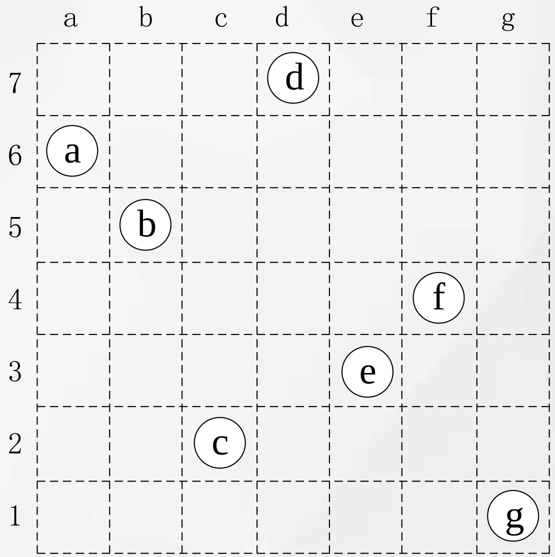
Treap树有唯一的形态

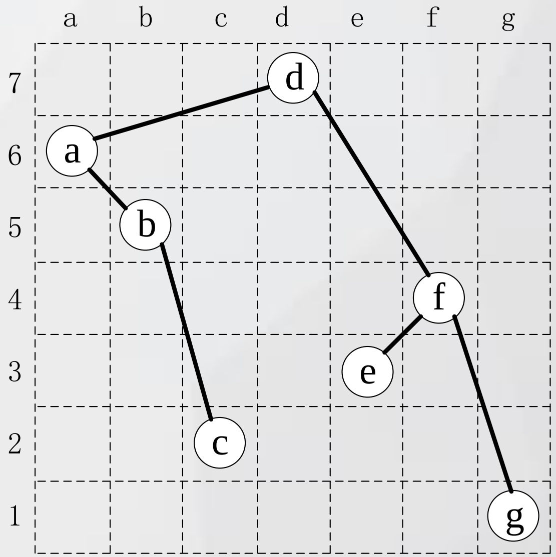
- 令每个结点的优先级互不相等，那么整棵树的形态是唯一的，和元素的插入顺序没有关系。

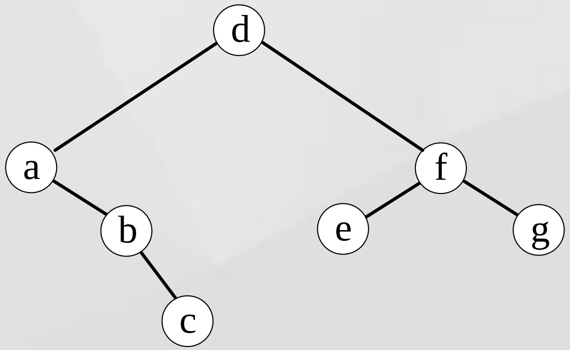
- **例如：**

键值：{a, b, c, d, e, f, g}

优先级：{6, 5, 2, 7, 3, 4, 1}。







(1)键值和优先级 **(2)建树** **(3)形成的Treap树**

Treap树如何解决平衡问题?

- 合理分配结点的优先级，可以得到一个比较平衡的BST。
- 一个简单的分配方法：随机。对每个结点的优先级进行随机赋值，生成的Treap树的形态也是随机的。
- 虽然不能保证每次生成的Treap树是平衡的，但是期望的插入、删除、查找的时间复杂度都是O(logn)的。

Treap树的插入

- **把新结点x插入到Treap树，分两步：**

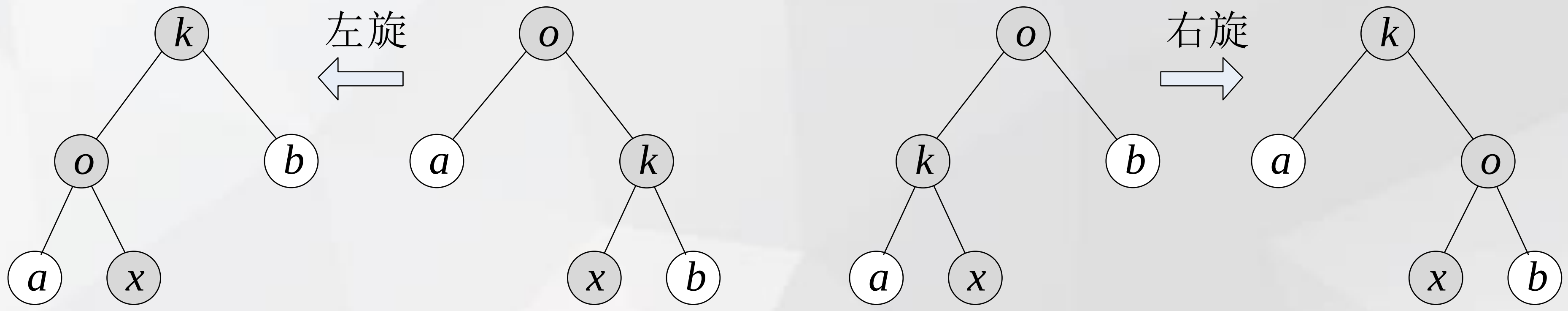
  （1）先把x按键值大小插入到合适的子树上。

  （2）给x随机分配一个优先级，如果x的优先级违反了堆的性质，即它的优先级比父结点高，那么进行调整，让x往上走，替代父结点，最后得到一个新的Treap树。

调整的技巧：旋转

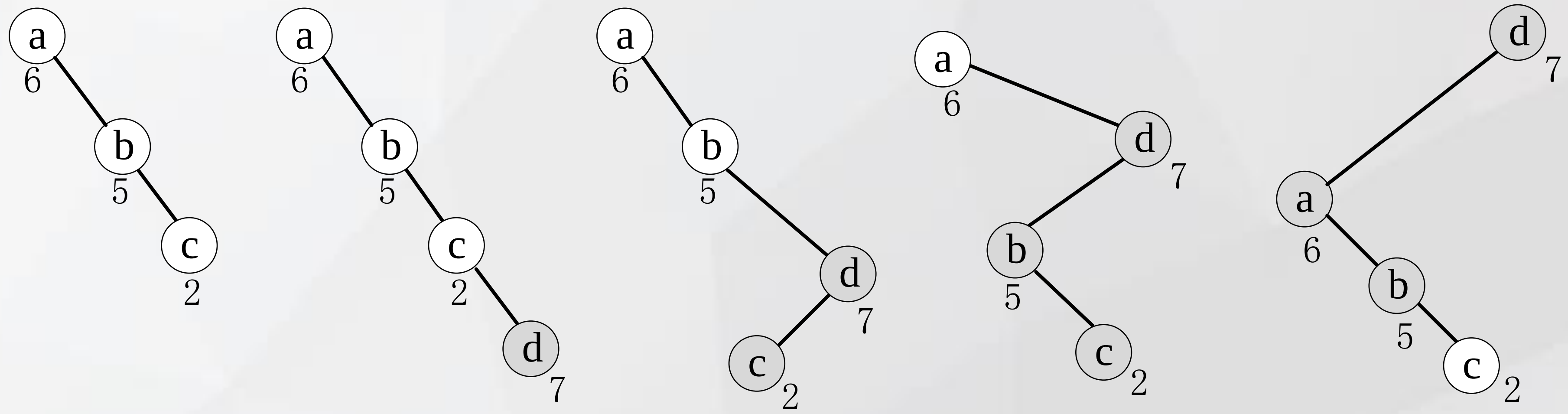
把*k*旋转到根：

```
void rotate(Node* &o,int d){    //d=0左旋转，d=1右旋
      Node *k=o->son[d^1];      //d^1与1-d等价，但是更快
      o->son[d^1]=k->son[d];    //图中的x
      k->son[d]=o;
      o=k;                      //返回新的根
}
```



例：新结点的插入和调整

- 图(2)插入d点，按朴素的插入方法插入到底部
- 图(3)d的优先级比父结点c高，左旋，上升；
- 图(4)d的优先级比新的父结点b高，继续左旋上升；
- 图(5)再次左旋上升，完成了新的Treap树。



(1)初始态　(2)插入d　(3)d左旋　(4)d左旋　(5)d左旋

Treap树的删除

- 待删除的结点x是叶子结点：直接删除。
- 待删除的结点x有子结点：找到优先级最大的子结点，把x向相反的方向旋转，也就是把x向树的下层调整，直到x被旋转到叶子结点，然后直接删除。

分裂与合并

- 把一棵树分裂成两棵树，或者把两棵树合并成一棵。
- Treap树做这样的操作，比较繁琐。

- 一般用Splay树做分裂与合并。

Treap的应用：名次树

hdu 4585 Shaolin

少林寺的第一个和尚是方丈，作为功夫大师，他规定每个加入少林的年轻和尚，要选一个老和尚来一场功夫战斗。每个和尚有一个独立的id和独立的战斗等级grade，新和尚可以选择跟他的战斗等级最接近的老和尚战斗。

方丈的id是1，战斗等级是10^9。他丢失了战斗记录，不过他记得和尚们加入少林的早晚顺序。请帮他恢复战斗记录。

输入：第一行是一个整数n，0 <n <=100,000，和尚的人数，但不包括大师本人。下面有n行，每行有两个整数k，g，表示一个和尚的id和战斗等级，0<= k ,g<=5,000,000。和尚以升序排序，即按加入少林的时间排序。最后一行用0表示结束。

输出：按时间顺序给出战斗，打印出每场战斗中新和尚和老和尚的id。

样例输入:

3

2 1

3 3

4 2

0

样例输出:

2 1

3 2

4 2

- 分析：先对老和尚的等级排序，加入一个新和尚时，找到等级最接近的老和尚，输出老和尚的id。
- n比较大，总复杂度需要是O(nlogn)的。
- 有多种解法，这里给出2种： STL map、Treap树。

(1) STL map

```
#include <bits/stdc++.h>
using namespace std;
map <int, int> mp;           //it->first是等级，it->second是id
int main(){
    int n;
    while (~scanf("%d",&n) && n){
            mp.clear();
            mp[1000000000]=1;   //方丈1，等级1000000000
            while(n--){
                int id,g;
                scanf("%d%d",&id,&g); //新和尚id，等级是g
                mp[g]=id;           //新和尚进队
                int ans;
                map<int,int>::iterator it = mp.find(g); //找到排好序的位置
                if (it == mp.begin())  ans=(++it)->second;
                else{
                    map<int,int> :: iterator it2=it;
                    it2--; it++;        //等级接近的前后两个老和尚
                    if (g -it2->first <= it->first - g)
                        ans=it2->second;
                    else  ans=it->second;
                }
                printf("%d %d\n",id,ans);
            }
    }
    return 0;
}
```

（2）Treap树代码

- **代码中包括了Treap树的常用操作：**

  定义结点struct Node

  旋转ratate()

  插入insert()

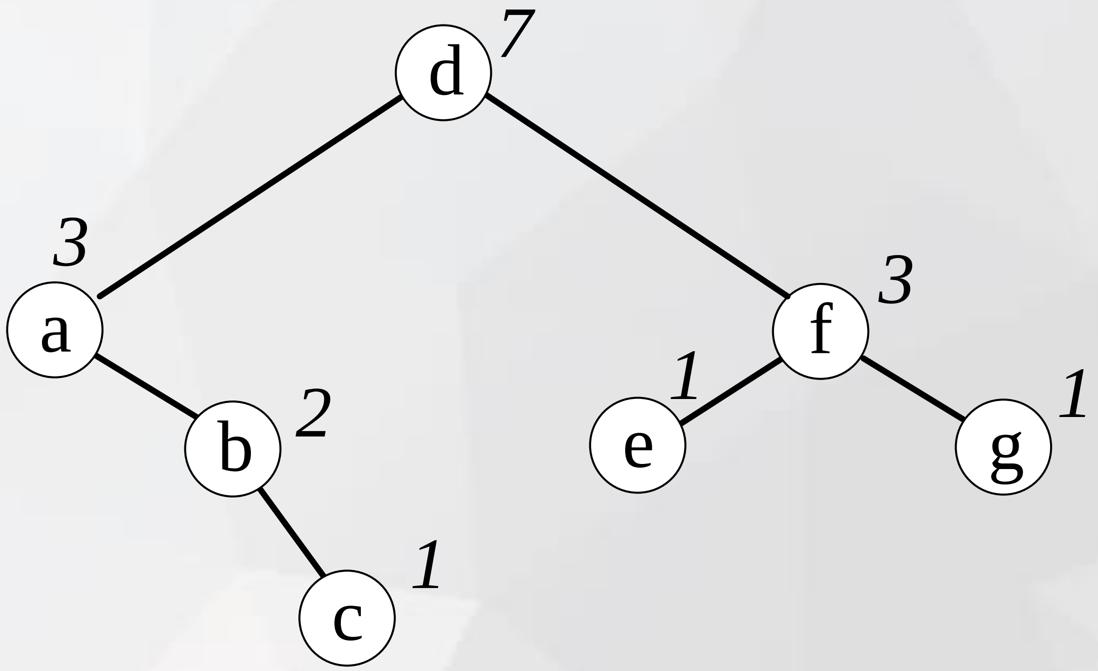
  找第k大数kth()，复杂度是O(logn)的

  查询某个数find() ，复杂度是O(logn)的

名次树

- kth()和find()：与名次树问题有关。
- 名次树的两个功能：找到第k大的元素；查询元素x的名次，即x排名第几。
- 这两个功能的实现，借助于给结点增加的一个size值。

- 一个结点的size值，是以它为根的子树的结点总数量
- 下图所示的名次树。图中结点上标注的数字就是这个结点的size。



◆ 伸展树Splay

- Splay树是一种BST树，它的查找、插入、删除、分割、合并等操作，复杂度都是O(logn)的。
- 最大的特点：可以把某个结点往上旋转到指定位置，特别是可以旋转到根的位置，成为新的根结点。
- 一个应用背景：如果需要经常查询和使用一个数，那么把它旋转到根结点，下次访问它，只需要查一次就找到了。

对比Splay和Treap

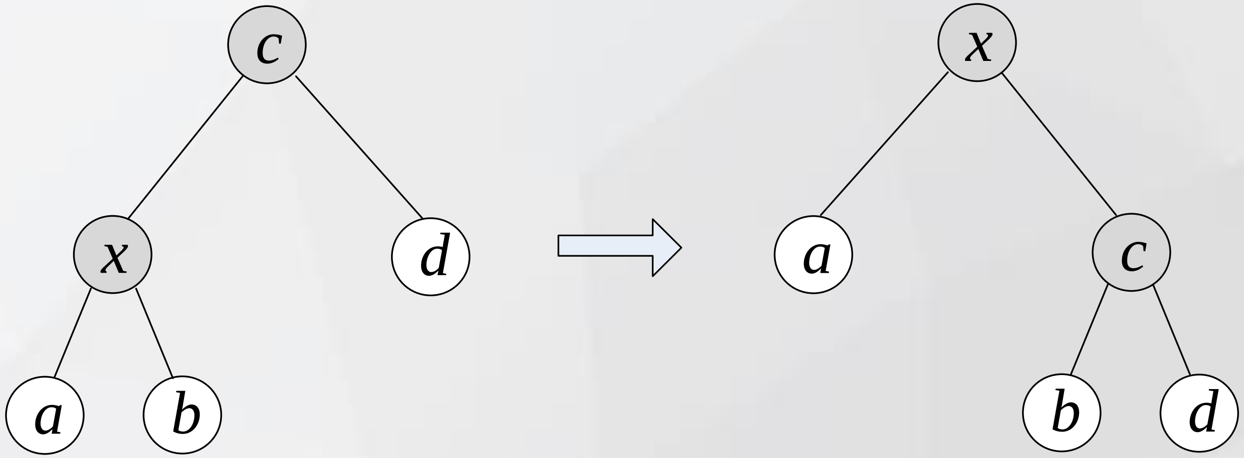
（1）Splay树允许把任意结点旋转到根，而Treap不能，因为它的形态是固定的。

（2）需要分裂和合并时，Splay树的操作非常简便。

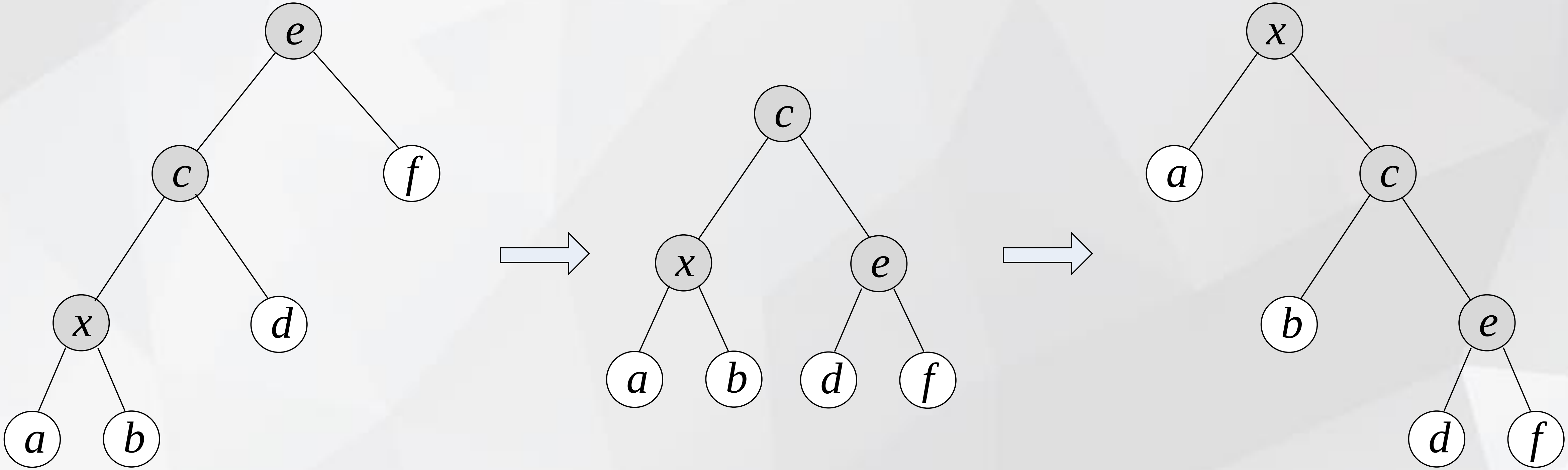
Splay的核心：把结点旋转到根（提根）

分3种情况。

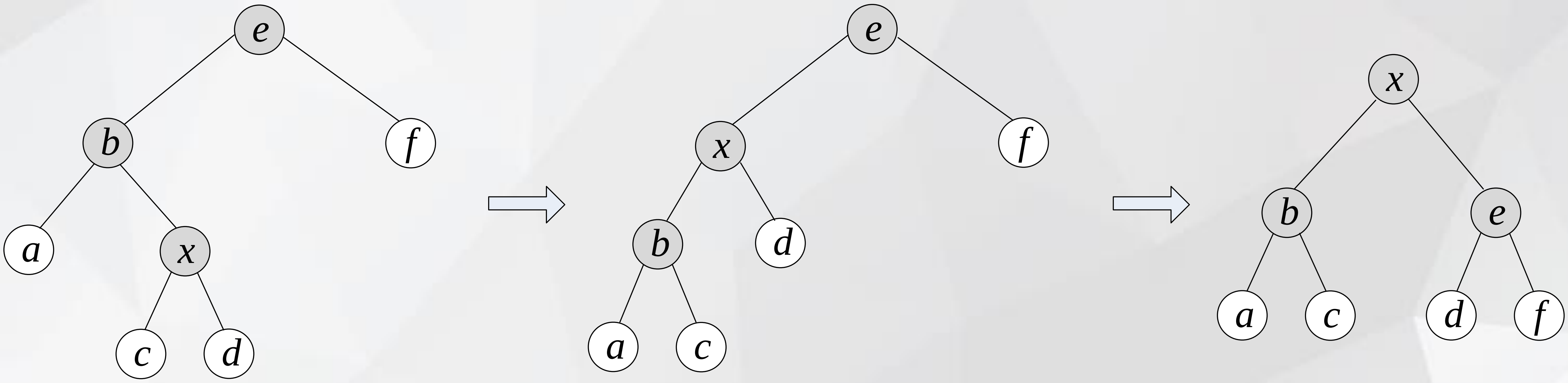
- 情况（1）：*x*的父结点就是根，只需要旋转一次。



❍ 情况（2）：x的父结点不是根，x、x的父结点、x的祖父结点，三点共线。



- 情况（3）：*x*、*x*的父结点、*x*的祖父结点，三点不共线。

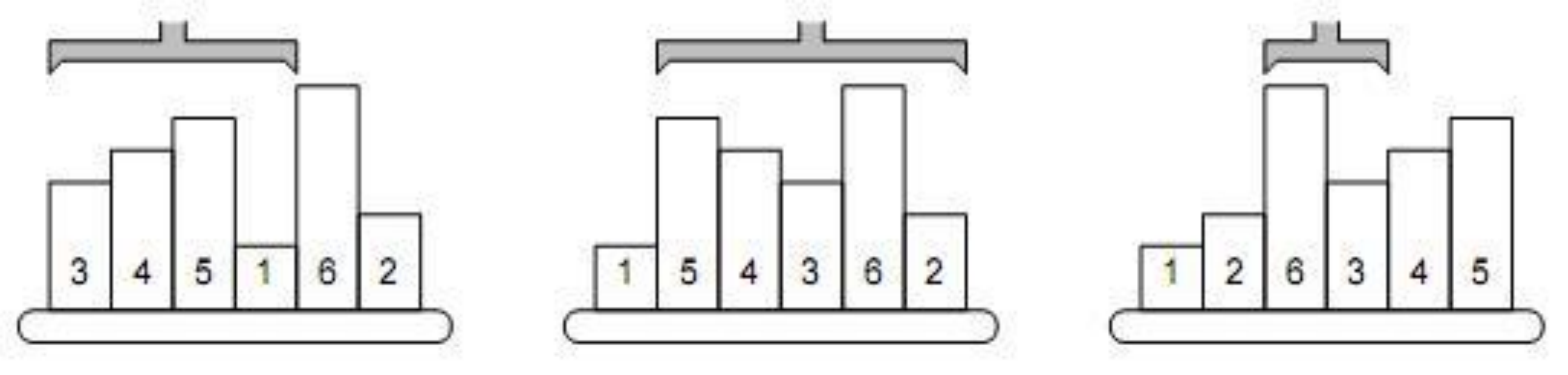


复杂度

- 旋转一次的时间是个常数。
- 把*x*从所在的深度提到根，总复杂度是多少?
- 如果是平衡二叉树，最深的结点深度是O(logn)，那么总复杂度就是O(logn)。
- 在**均摊**意义上，可以把Splay提根操作的复杂度看成是O(logn)。

例题：hdu 1890 Robotic Sort

有n个数字（图中的高度是数字大小），1 ≤n≤100000，用一个机械臂帮忙排序，其方法如下图。左图中，用机械臂夹住第一个数和最小的数，翻转，变成中图的样子，最小的数就处于第一个位置。然后对中图用同样的方法找第二小的数。继续这个过程直到结束。



输入一些数字，输出第i次翻转之前，第i大数的位置。

样例输入：3 4 5 1 6 2

样例输出：4 6 4 5 6 6

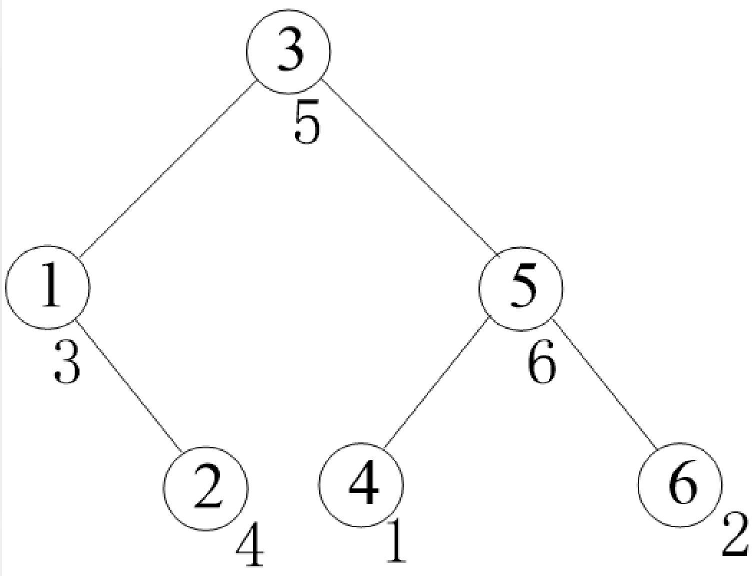
思路

- 找到第i大的数，翻转它左边的数（不包括已经处理过的比它小的数），右边的保持不变。
- 如果用模拟法编程，复杂度约$O(n^3)$，会TLE。
- 需要O(nlogn)的方法。能否借助splay?

（1）建树

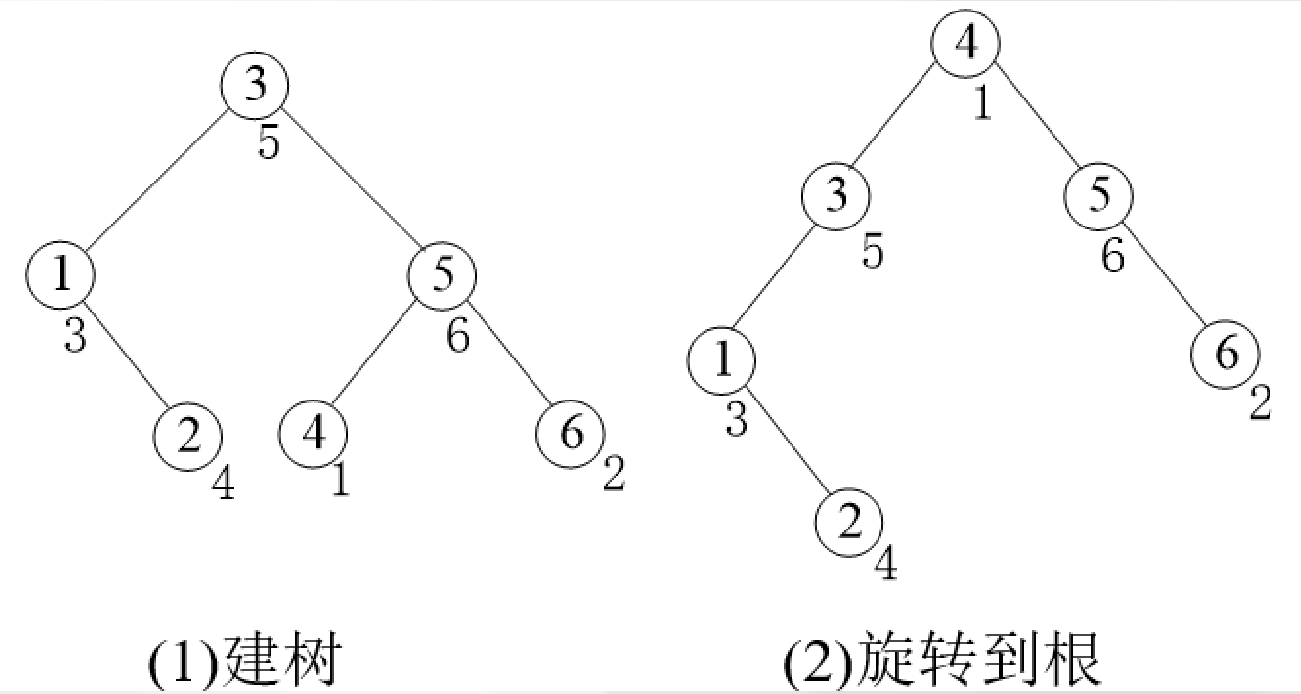
圆圈内数字是初始位置，圆圈旁边数字是题目给出的序列。

根据中序遍历，它是题目的样例3 4 5 1 6 2。



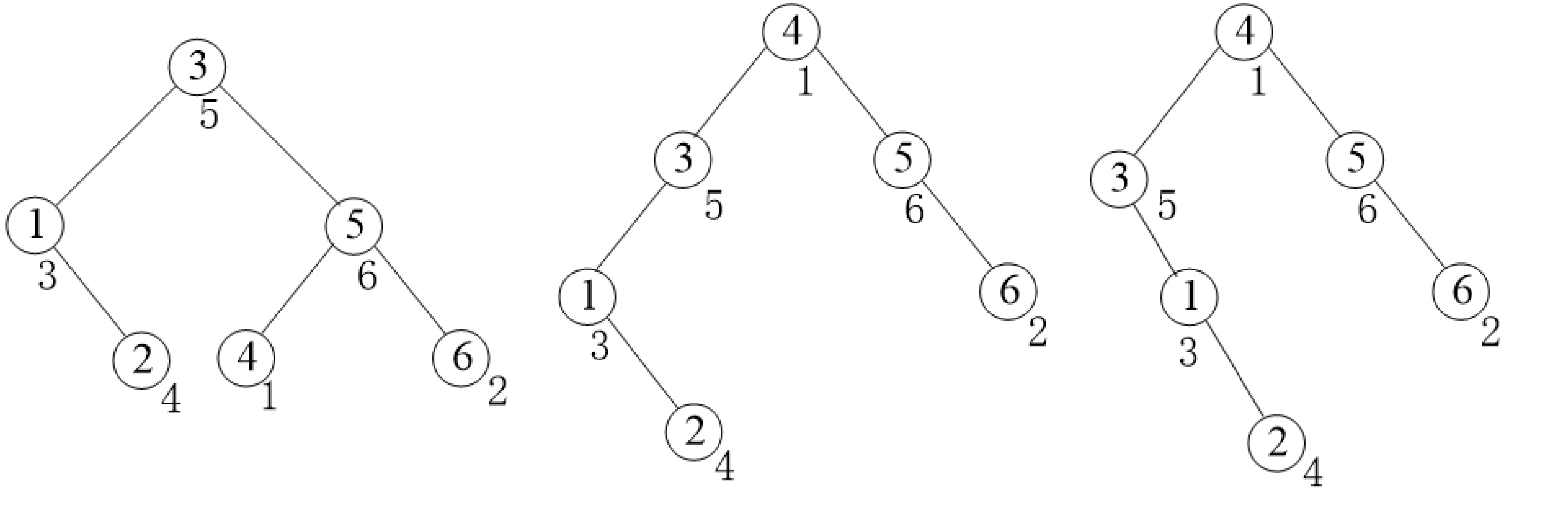
（2）用Splay旋转到根

- 找到最小的数（以1为例），用Splay把它旋转到根。它左子树的大小就是数列中排在它左边的数的个数，也就是题目的输出。



（3）翻转左子树

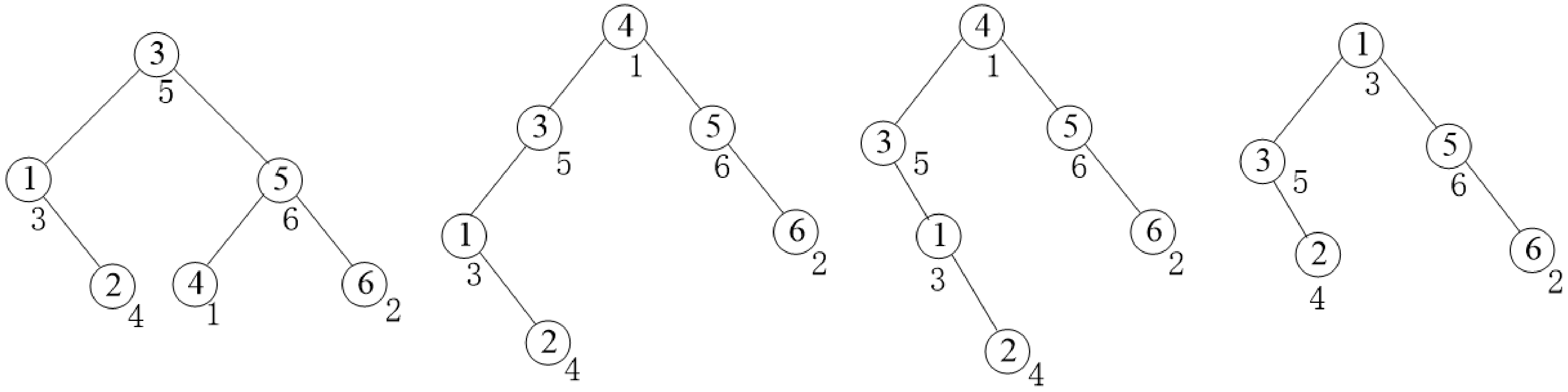
- 模拟题目中的机械臂翻转。但是，如果每次都完全翻转左子树，时间必然超时。这里从线段树得到启发，用标记的方式记录翻转情况，减少直接操作的次数，等Splay操作的时候再处理。图(3)中只翻转了结点3，对结点3做标记，而它的子树1、2保持不变。



(1)建树　(2)旋转到根　(3)处理左子树的翻转

（4）删除根

- 在树上删除最小数。删除过程中，根据标记进行子树的翻转。最后的结果见图(4)，是去掉了最小数的树，第一次处理结束。



(1)建树　(2)旋转到根　(3)处理左子树的翻转　(4)删除根

代码

- 建树：buildtree()
- 旋转到根：splay()
- 标记：update_rev()

- 代码中去掉update_rev()、pushup()、pushdown()，就是纯的Splay代码。

BST习题

- hdu 1622，建二叉树
- hdu 3999，二叉树遍历
- hdu 3791，BST
- hdu 4453，splay基本题
- hdu 3726，离线处理+splay，经典题。用Treap树也能做。