

Lecture 19: 树与二叉树的应用

19.1 二叉排序树

19.1.1 查找

如何查找

- 左子树结点值<根结点值<右子树结点值
- 若树非空，目标值与根结点的值比较：
 1. 若相等，则查找成功
 2. 若小于根结点，则在左子树上查找，否则在右子树上查找。
 3. 查找成功，返回结点指针。查找失败返回 NULL

代码实现

```
//二叉排序树结点
typedef struct BSTNode{
    int key;
    struct BSTNode *lchild,*rchild;
}BSTNode,*BSTree;
```

```
//在二叉排序树中查找值为key的结点
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL&&key!=T->data){ //若树空或等于根结点值，则结束循环
        if(key<T->data)
            T=T->lchild; //小于，则在左子树上查找
        else
            T=T->rchild; //大于，则在右子树上查找
    }
    return T;
}
//最坏空间复杂度O(1)
```

```
//在二叉排序树中查找值为key的结点(递归实现)
BSTNode *BSTSearch(BSTree T,int key){
    if(T==NULL)
        return NULL; //查找失败
    if(key==T->key)
        return T; //查找成功
    else if(key<T->key)
        return BSTSearch(t->lchild,key); //在左子树中找
    else
        return BSTSearch(t->rchild,key); //在右子树中找
}
```

19.1.2 插入

如何插入

- 若原二叉排序树为空，则直接插入结点。
- 否则，若关键字 k 小于根结点值，则插入到左子树，若关键字 k 大于根结点值，则插入到右子树。
- **注意**：新插入的结点一定是叶子结点
- **最坏空间复杂度** $O(h)$

代码实现

```
//在二叉排序树插入关键字为k的新结点(递归实现)
int BST_Insert(BiTree &T,KeyType k){
    if(T==NULL){        //原树为空，新插入的结点为根结点
        T=(BiTree)malloc(sizeof(BSTNode));
        T->data=k;
        T->lchild=T->rchild=NULL;
        return 1;        //返回1,插入成功
    }
    else if(k==T->data)
        return 0;
    else if(k<T->key)//插入到T的左子树
        return BST_Insert(T->lchild,k);
    else                //插入到T的右子树
        return BST_Insert(T->rchild,k);
}
```

19.1.3 构造与删除

二叉排序树的构造

```
//按照str[]中的关键字序列建立二叉排序树
void Creat_BST(BiTree &T,KeyType str[],int n){
    T=NULL;        //初始时T为空树
    int i=0;
    while(i<n){ //依次将每个关键字插入到二叉排序树中
        BST_Insert(T,str[i]);
        i++;
    }
}
```

删除

- 先搜索找到目标节点：
1. 若被删除结点 z 是叶结点，则直接删除，不会破坏二叉排序树的性质。
 2. 若结点 z 只有一棵左子树或右子树，则让 z 的子树称为 z 父结点的子树，替代 z 的位置。

3. 若结点 z 有左、右两棵子树，则令 z 的直接后继 (或直接前驱) 替代 z ，然后从二叉排序书中删除这个直接后继 (或直接前驱)，这样就转换成了第一或第二种情况。

19.1.4 查找效率分析

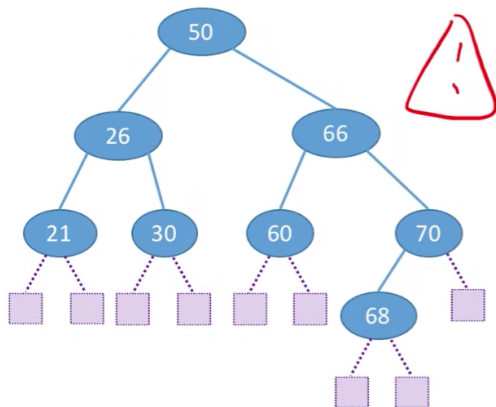
查找长度

- 在查找运算中，需要对比关键字的次数称为查找长度，反映了查找操作时间复杂度
- 最好情况**
 - n 个结点的二叉树最小高度为 $\lfloor \log_2 n \rfloor + 1$
 - 平均查找长度 $O(\log_2 n)$
- 最坏情况**
 - 每个结点只有一个分支，树高 $h = \text{结点数 } n$
 - 平均查找长度 $O(n)$
- 总结：
 - 每层 * 每层的结点数 / 总结点数

查找失败

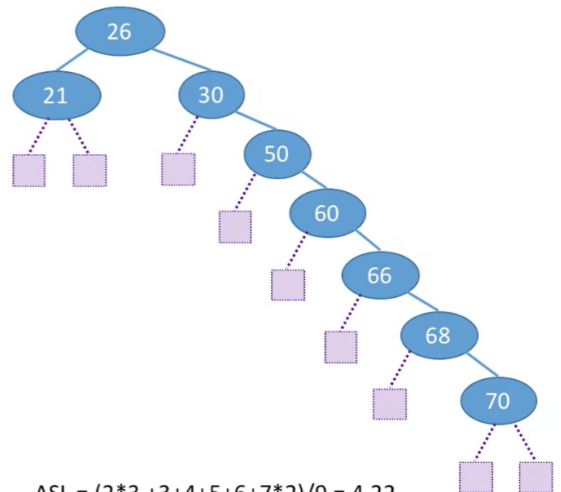
- 公式：
 - $ASL = \sum \frac{\text{本层高度} * \text{本层叶子数}}{\text{补上的叶子数}}$
- 图示：

查找长度——在查找运算中，需要对比关键字的次数称为查找长度。



查找失败的平均查找长度 ASL (Average Search Length)

$$ASL = (3 * 7 + 4 * 2) / 9 = 3.22$$



$$ASL = (2 * 3 + 3 + 4 + 5 + 6 + 7 * 2) / 9 = 4.22$$

19.2 平衡二叉树

19.2.1 定义

平衡二叉树 (Balanced Binary Tree)

- 简称平衡树 (AVL 树)：树上任一结点的左子树和右子树的高度之差不超过 1
- 结点的平衡因子 = 左子树高 - 右子树高

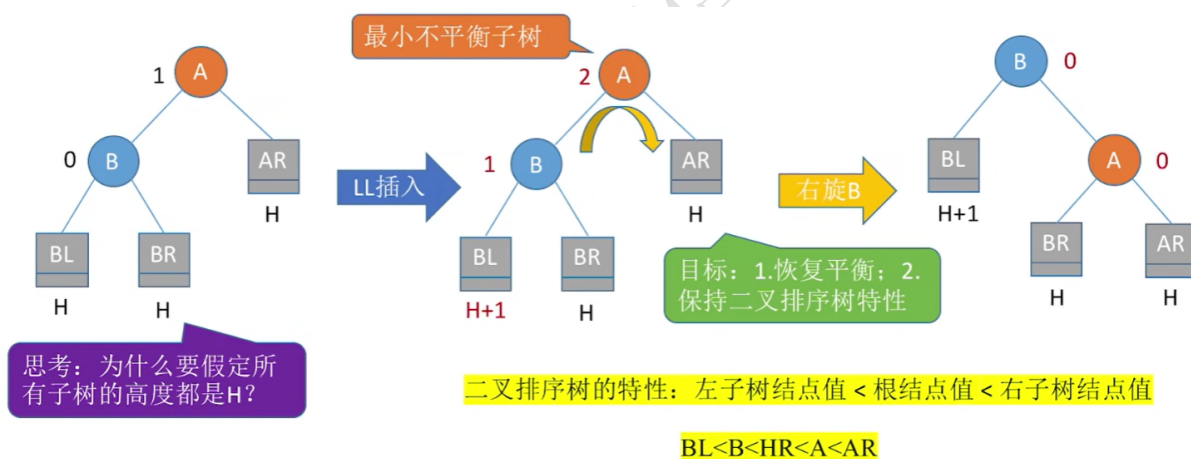
代码实现

```
//平衡二叉树结点
typedef struct AVLNode{
    int key;          //数据域
    int balance;      //平衡因子
    struct AVLNode *lchild,*rchild;
}AVLNode,*AVLTree;
```

19.2.2 调整最小不平衡子树

概念：LL 旋转

- 概念：
 - 由于在 A 的左孩子的左子树中插入导致不平衡引发的旋转；
- 操作
 - LL 平衡旋转 (右单旋转)。
 - 由于在结点 A 的左孩子 (L)的左子树 (L)上插入了新结点，A 的平衡因子由 1 增至 2，导致以 A 为根的子树失去平衡，需要一次向右的旋转操作。
 - 将 A 的左孩子B 向右上旋转替代 A 成为根结点，将A 结点向右下旋转成为 B 的右子树的根结点，而 B 的原右子树则作为 A 结点的左子树。
- 图示：



概念：RR 旋转

- 概念：
 - 由于在 A 的右孩子的右子树中插入导致不平衡
- 操作
 - RR 平衡旋转 (左单旋转)
 - 由于在结点 A 的右孩子 (R)的右子树 (R)上插入了新结点，A 的平衡因子由-1 减至-2，导致以 A 为根的子树失去平衡，需要一次向左的旋转操作。
 - 将 A 的右孩子B 向左旋转代替 A 成为根结点，将A 结点向左下旋转成为 B 的左子树的根结点，而 B 的原左子树则作为 A 结点的右子树

概念：LL 旋转

- 概念：
 - 由于在 A 的左孩子的右子树中插入导致不平衡

- 操作
 - LR 平衡旋转 (先左后右双旋转)
 - 由于在 A 的左孩子 (L) 的右子树 (R) 上插入新结点, A 的平衡因子由 1 增至 2, 导致以 A 为根的子树失去平衡, 需要进行两次旋转操作, 先左旋转后右旋转。
 - 先将 A 结点的左孩子 B 的右子树的根结点 C 向左上旋转提升到 B 结点的位置, 然后再把该 C 结点向右上旋转提升到 A 结点的位置。

概念: LL 旋转

- 概念:
 - 由于在 A 的右孩子的左子树中插入导致不平衡
- 操作
 - RL 平衡旋转 (先右后左双旋转)

19.2.3 调整二叉树总结

1. LL

1. 在 A 的左孩子的左子树中插入导致不平衡
2. **调整** A 的左孩子结点右上旋

2. RR

1. 在 A 的右孩子的右子树中插入导致不平衡
2. **调整** A 的右孩子结点左上旋

3. LR

1. 在 A 的左孩子的右子树中插入导致不平衡
2. **调整** A 的左孩子的右孩子先左上旋再右上旋

4. RL

1. 在 A 的右孩子的左子树中插入导致不平衡
2. **调整** A 的右孩子的左孩子先右上旋后左上旋

注意: 插入操作导致"最小不平衡子树"高度+1, 经过调整后高度恢复

19.2.4 查找效率分析

效率分析

- 概念:
 - 若树高为 h , 则最坏情况下, 查找一个关键字最多需要对比 h 次, 即查找操作的时间复杂度不可能超过 $O(h)$
- 平衡二叉树:
 - 树上任一结点的左子树和右子树的高度之差不超过 1
 - 假设以 n_h 表示深度为 h 的平衡树中含有的最少结点数
 - 则有 $n_0 = 0, n_1 = 1, n_2 = 2$
 - 并且有: $n_h = n_{h-1} + n_{h-2} + 1$
- 可以证明:
 - 含有 n 个结点的平衡二叉树的最大深度为 $O(\log_2 n)$, 平衡二叉树的平均查找长度为 $O(\log_2 n)$

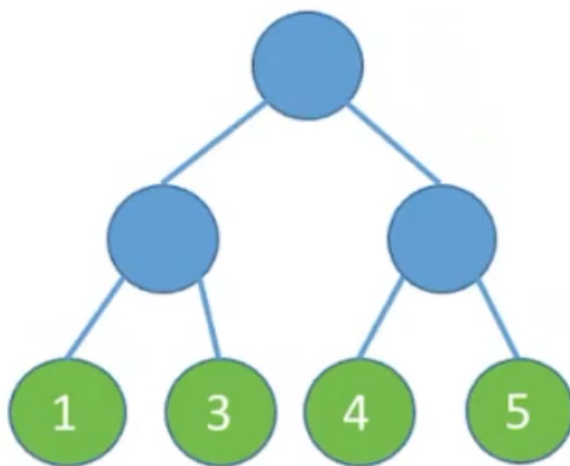
19.3 哈夫曼树

19.3.1 带权路径长度

基本概念

- 结点的权
 - 概念：有某种现实含义的数值
 - 例如：表示结点的重要性等
- 结点的带权路径长度
 - 从树的根到该结点的**路径长度** (经过的边数)与该**结点上权值**的乘积；
- 树的带权路径长度
 - 树中所有**叶结点的带权路径长度之和** (WPL, Weighted Path Length)；
 - 公式：

$$WPL = \sum_{i=1}^n w_i l_i$$



$$WPL = 2 \times 1 + 2 \times 3 + 2 \times 4 + 2 \times 5 = 26$$

- 举例：

19.3.2 哈夫曼树的定义与构造

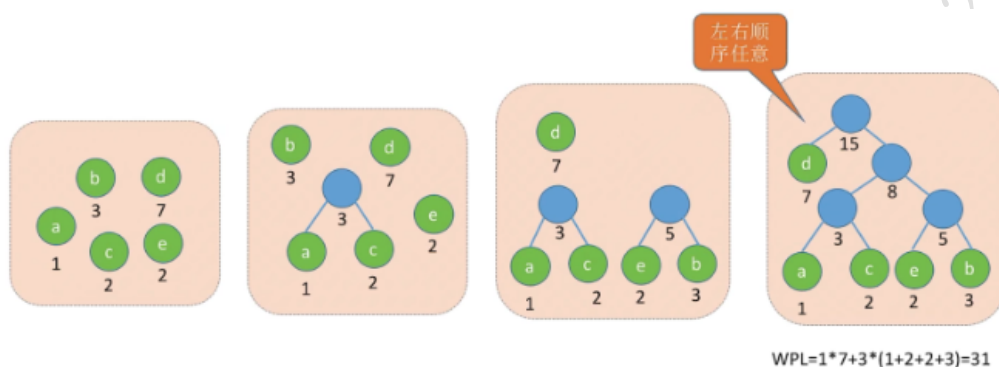
哈夫曼树的定义

- 定义：在含有 n 个带权**叶结点**的二叉树中，其中带权路径长度 (WPL) 最小的二叉树称为哈夫曼树，也称最优二叉树

哈夫曼树的构造

- 构造
 - 前提：假设给定 n 个权值分别为 $W_1, W_2, \dots, W_n$ 的结点进行构造
 - 1. 将这 n 个结点分别作为 n 棵仅含一个结点的二叉树，构成森林 F 。

- 2. 构造一个新结点，从 F 中**选取两棵根结点权值最小的树**作为新结点的左，右子树，并且将新结点的权值置为左，右子树上根结点的权值之和。
- 3. 从 F 中删除刚才选出的两棵树，同时将新得到的树加入 F 中。
- 4. 重复 (2)和 (3)，直至 F 中只剩下一颗树为止；
- 特点
 - 1. 每个初始结点最终都成为叶结点，且权值越小的结点到根结点的路径长度越大；
 - 2. 哈夫曼树的结点总数为 $2n - 1$ ；
 - $2n - 1 = n + (n - 1)$ ，其中 n 为原本就有的 n 个叶子结点， $n-1$ 为新连接得到的结点树，两个节点相连就新增一个结点；
 - 3. 哈夫曼树中不存在度为 1 的结点；
 - 4. 哈夫曼树并不**唯一**，但 WPL 必然相同且为最优；
- 图示：构造过程



19.3.3 哈夫曼编码

基础概念

- 可变长度编码：
 - 概念：允许对不同字符用**不等长**的二进制位表示
- 固定长度编码：
 - 概念：每个字符用**相等**的二进制表示
- 前缀编码：
 - 得到的码当中，任何一个编码都不是另一个编码的前缀，因此当接收到一堆编码过后的数字后，可以准确、无歧义的解码；
 - 若没有一个编码是另一个编码的前缀，则称这样的编码为**前缀编码**；
- 哈夫曼编码：
 - 用构造哈夫曼树的一种方法，构造的一种字符集的编码方案；
 - 字符集中的**每个字符作为一个叶子结点**，各个字符出现的**频度**作为结点的**权值**，根据之前介绍的方法构造哈夫曼树；
 - 哈夫曼编码的结果不唯一；

19.4 并查集

19.4.1 逻辑结构

集合

- 各个集合之间互不相交；
- 代码实现集合的思路：将不同的不相关的集合，放到不同的树下，构成森林；

19.4.1 树与集合

查找操作

- 概念：查到一个元素到底属于哪一个集合；
- 方法：从当前元素出发，一路向上，找到根节点 -> 判断属于哪一个集合；

判同

- 概念：判断两个元素是否属于同一个集合
- 方法：分别查找到两个元素的根，判断根节点是否相同

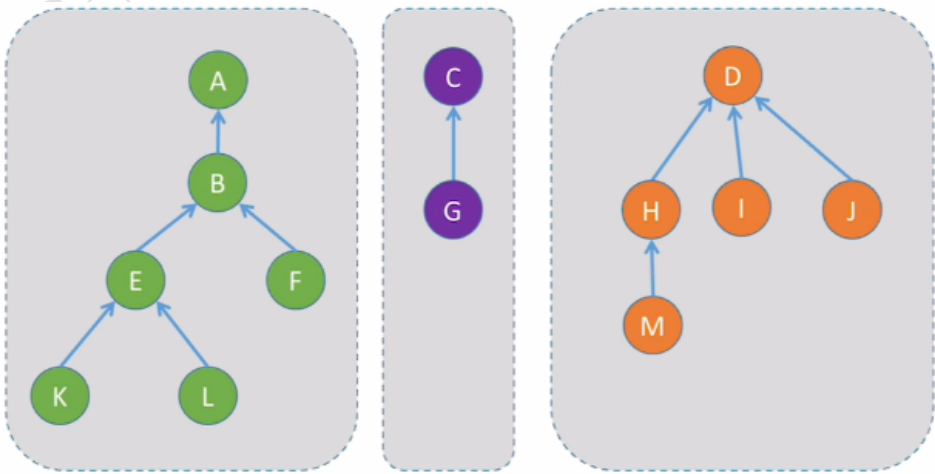
合并操作

- 概念：讲两个互不相交的集合合并；
- 方法：让一棵树成为另一颗树的子树；

19.4.2 存储结构

并查集的存储结构

- 概念：
 - 使用树的双亲表示法进行存储：将每个结点的父节点的 index 存储在一个数组构成的表当中；
 - 使用双亲表示法进行并查集实现好处：十分简单就可以实现并和查 -> 只要对根进行操作，并可以通过 index 轻松的访问父节点；
- 图示：



数据元素	A	B	C	D	E	F	G	H	I	J	K	L
数组下标	0	1	2	3	4	5	6	7	8	9	10	11
S[]	-1	0	-1	-1	1	1	2	3	3	3	4	4

19.4.3 基本操作

集合的基本操作

- 查
 - `Find`：确定一个指定元素所属集合；
- 并
 - `Union`：将两个不交的集合合并为一个
- 注：
 - 并查集 (`Disjoint Set`) 是集合这种逻辑结构的一种具体实现；
 - 只能进行“并”和“查”两种基本操作；

时间复杂度

- `Find`： $O(n)$
 - 和树的高度 h 直接相关；
- `Union`： $O(1)$

代码实现：初始化

- 初始化：先将每个元素都初始化为独立的子集 $\rightarrow S[i] = -1$

```
//并查集的初始化：
#define SIZE 13
int UFsets[SIZE];    //集合元素数组

//初始化并查集
void Initial(int S[]){
    for(int i=0;i<SIZE;i++)
        S[i]=-1;
}
```

代码实现：并和查

```
//查操作，找x所属集合(返回x所属根节点)
int Find(int S[],int x){
    // 循环，寻找到x的根，根的下标对应的值为-1
    while(S[x]>=0)
    {
        x=S[x];
    }

    return x;
}

//并操作
void Union(int S[],int Root1,int Root2){
    //确保当前Root1与Root2是不同的集合
    if(Root1==Root2)
    {
        return;
    }
}
```

```
}
```

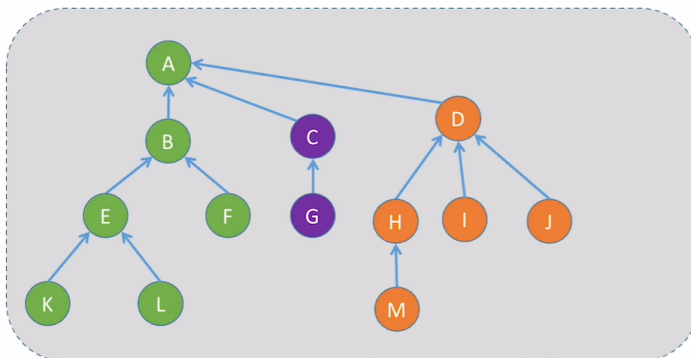
```
//将根Root2连接到另一根Root1下面  
S[Root2]=Root1;
```

```
}
```

19.4.4 对并操作的优化

概念: Union 操作的优化

- 思路
 - 在每次 Union 操作构建树的时候, 尽可能让树不长高;
 - 即: 将小树合并到大树下面;
- 操作
 - 1. 用根节点的绝对值表示树的结点总数;
 - 比如图中 A 的下标为 0, 其存储的值为 -13, 表示 A 下面有 13 个元素;
 - 2. 使用 Union 操作时, 让小树合并到大树;
- 树高:
 - 该方法构造的树高不超过 $\lfloor \log_2 n \rfloor + 1$
- 效率:
 - $O(\log_2 n)$
- 图示



```
//Union “并”操作, 小树合并到大树  
void Union(int S[],int Root1,int Root2){  
    if(Root1==Root2) return;  
    if(S[Root2]>S[Root1]) { //Root2结点数更少  
        S[Root1] += S[Root2]; //累加结点总数  
        S[Root2]=Root1; //小树合并到大树  
    } else {  
        S[Root2] += S[Root1]; //累加结点总数  
        S[Root1]=Root2; //小树合并到大树  
    }  
}
```

数据元素	A	B	C	D	E	F	G	H	I	J	K	L	M
数组下标	0	1	2	3	4	5	6	7	8	9	10	11	12
S[]	-13	0	0	0	1	1	2	3	3	3	4	4	7

代码实现

```
void Union(int S[],int root1,int root2)  
{  
    if(root1 == root2)  
    {  
        return;  
    }  
  
    // root1的值更小, 因此root1下面的结点数更多, 将小树合并到大树  
    if(s[root2] > s[root1])  
    {
```

```

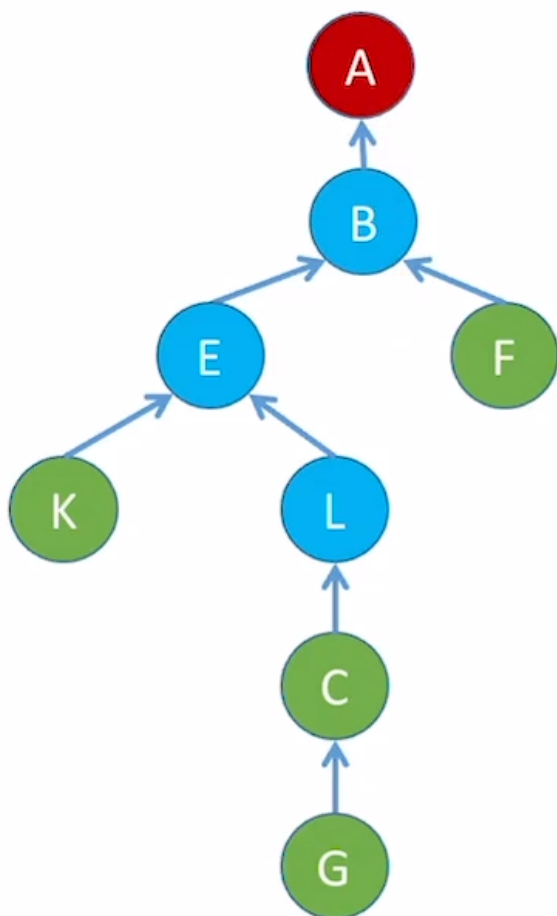
    s[root1] += s[root2];
    s[root2] = root1;
}
else
{
    s[root2] += s[root1];
    s[root1] = root2;
}
}

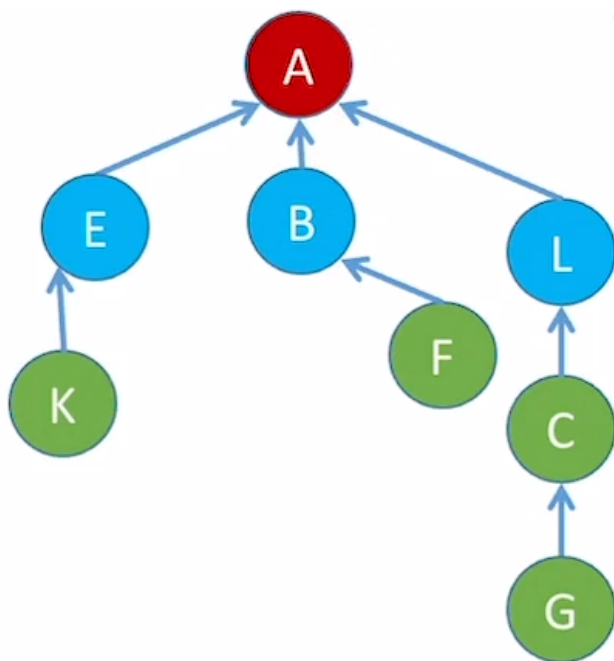
```

19.4.5 对查操作的优化

查找路径

- 概念：
 - 对 `Find` 的查找路径进行优化；
 - 使得查找所需要的路径数变少；
- 方法：
 1. 先找到根；
 2. 分别处理查找路径上的各个顶点，把每个点改在根节点下面；
- 效率：
 - 几乎小于 $O(4)$
- 图示：
 - 把 A 下面蓝色的结点，全部挂到 A 的下面；





代码实现

```

int Find(int S[], int x)
{
    int root = x;
    while(s[root] >= 0)
    {
        // 循环找到路径
        root = S[root];
    }
    while(x != root)
    {
        int t = S[x]; // t指向x的父节点
        S[x] = root; // x直接挂在根节点下
        x = t;
    }

    return root;
}

```

19.4.6 总结

无优化

- Find 操作：
 - $O(n)$
 - 即理论上的最大树高；
- 将 n 个元素 Union 成一个集合： $O(n^2)$
 - 先花 $O(n)$ 找到目标位置，然后对 n 个元素进行合并，因此是 $O(n^2)$

Union 优化

- Find 操作： $O(\log_2 n)$

- 即理论上的最大树高；
- 将 n 个元素 Union 成一个集合：
 - $O(n \log_2 n)$

Find 优化

- Find 操作: $O(\alpha(n))$
- 将 n 个元素 Union 成一个集合: $O(n\alpha(n))$

获取笔记持续更新可加微信: WandD2277