第六章 树

1.一棵高度为h的满k叉树有如下性质：根结点所在层次为0；第h层上的结点都是叶子结点；其余各层上每个结点都有k棵非空子树，如果按层次自顶向下，同一层自左向右，顺序从1开始对全部结点进行编号，试问：
(1) 各层的结点个数是多少?
(2) 编号为i的结点的第m个孩子结点（若存在）的编号是多少?
(3) 编号为i的结点有右兄弟的条件是什么？其右兄弟结点的编号是多少?
请简要写出推算过程。

解：
(1)第 i 层的结点个数为 $k^i, i = 0, 1, \ldots, h$，下用数学归纳法证明：
当层数 $n = 0$ 时有 $k^0 = 1$，符合结论。
假设当层数 $n = m$, $m \in N_+$ 时结点个数为 k^m，则当层数 $n = m + 1$ 时结点个数为 $k^m \cdot k = k^{m+1}$ (根据满k叉树的定义)，符合结论。
综上所述，第 i 层的结点个数为 $k^i, i = 0, 1, \ldots, h$。

(2)编号为 i 的结点的第 m 个孩子结点编号为 $k \cdot (i - 1) + m + 1$ ，理由如下：
设编号为 i 的结点在第 j 层，分两种情况讨论：
a. $j = 0, i = 1$
显然若存在子结点，第 m 个子结点的编号为 $k \cdot (i - 1) + m + 1 = m + 1$
b. $j > 0$
先求出第 j 层结点中在 i 结点左侧和右侧的结点数。
第 $0, 1, \ldots, j - 1$ 层的结点总数为

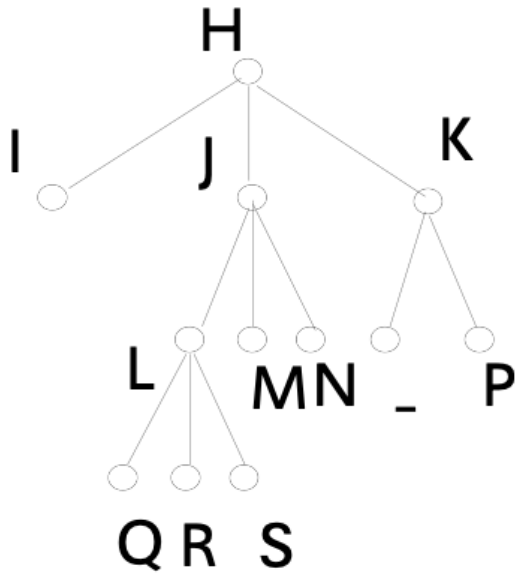
$$\sum_{h=0}^{j-1} k^h = \frac{k^j - 1}{k - 1}$$

故第j层结点中在 i 结点左侧的结点有 $i - \frac{k^j - 1}{k-1} - 1$ 个
同理可得：第 j 层结点中在 i 结点右侧的结点有 $\frac{k^{j+1}-1}{k-1} - i - 1$ 个
故i结点的第 m 个子结点编号为

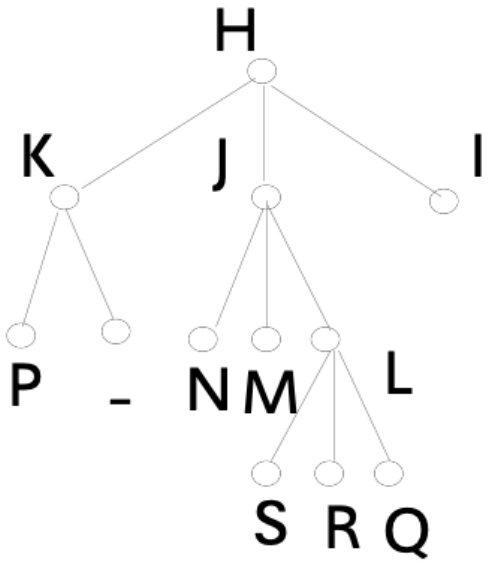
$$k(i - \frac{k^j - 1}{k - 1} - 1) + [i + (\frac{k^{j+1} - 1}{k - 1} - i - 1)] + m + 1 = k(i - 1) + m + 1$$

(3)显然，编号为 i 的结点有右兄弟当且仅当其不是该层的最后一个结点。
设编号为 i 的结点在第 j 层，由(2)知第 j 层最后一个结点编号是 $\frac{k^{j+1}-1}{k-1}$ ，所以当且仅当 $i \neq \frac{k^{j+1}-1}{k-1}$, $j = 0, 1, \ldots$ 时 i 结点有右兄弟，其右兄弟结点的编号为 $i + 1$

2.编写算法，将森林的所有结点的子女倒置（镜面映射）。注意：树是森林的特例



(a) 原树



(b) 倒置后的树

解：

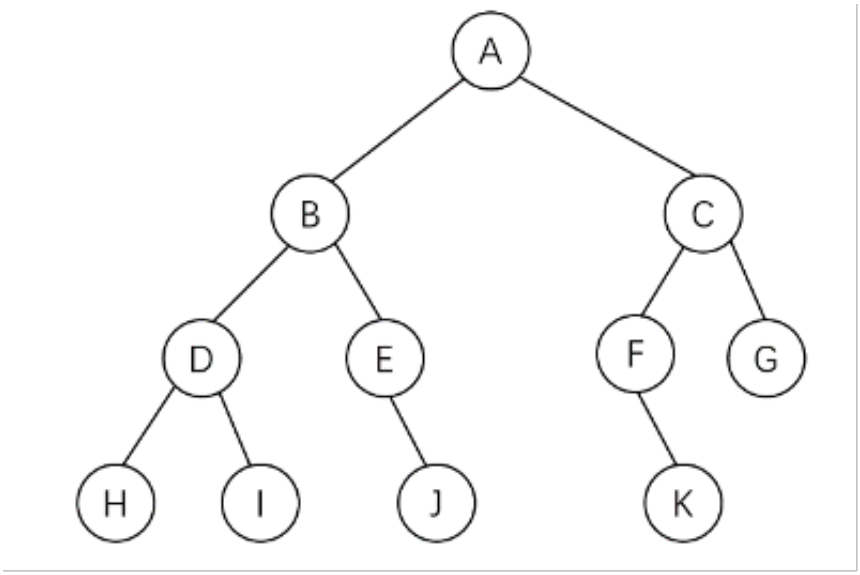
算法描述：按照层次倒置森林，利用递归对当前结点和其所有兄弟结点进行镜面映射。镜面映射时需要从两端向中间遍历所有兄弟结点并互换对称位置上的结点。

```
template<class T>
class TreeNode{
public:
    T val;
    TreeNode<T>*leftMostChild,*rightSibling;//用左子右兄二叉树存储森林
    TreeNode(T v,TreeNode<T>*l,TreeNode<T>*r):
        val(v),leftMostChild(l),rightSibling(r){}
};
template<class T>
void reverse(TreeNode<T>*root){//倒置root所在层的所有结点
    if(!root)//空树不用倒置
        return;
    vector<TreeNode<T>*>rec;//rec存储root的所有兄弟结点
    rec.push_back(root);
    TreeNode<T>*pointer=root;
    while(pointer->rightSibling){
        pointer=pointer->rightSibling;
        rec.push_back(pointer);
    }
    int n=rec.size();
    int i=0,j=n-1;
    while(i<j){
        swap(rec[i],rec[j]);//交换倒置
        i++,j--;
    }
    //更新结点间的位置关系
    root=rec[0];
    for(int i=0;i<n-1;i++){
```

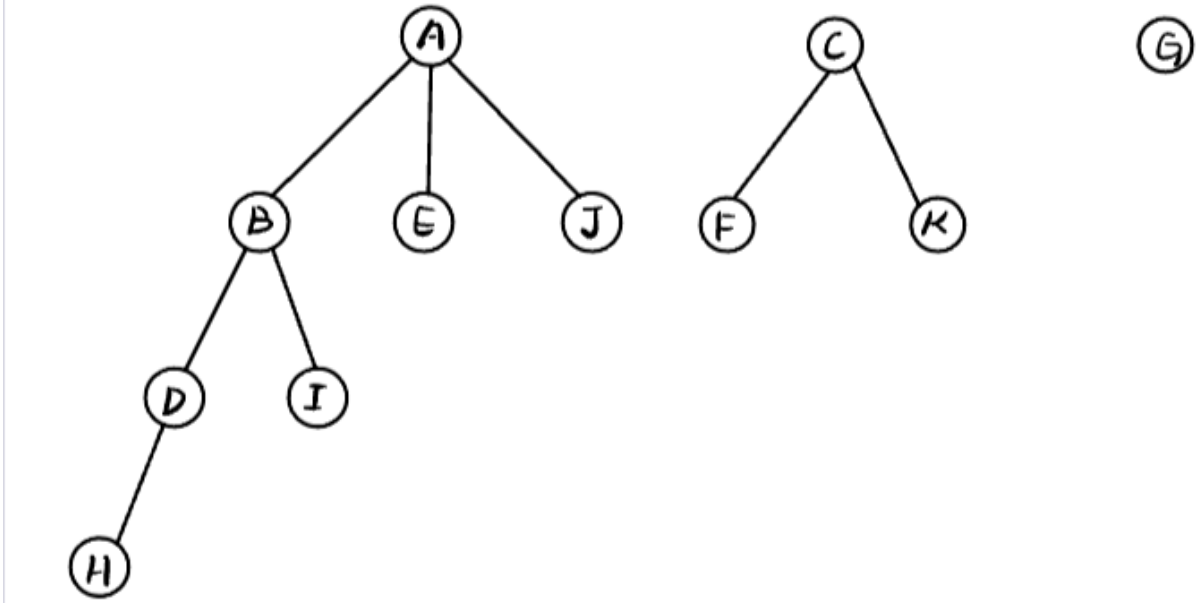
```
            rec[i]->rightSibling=rec[i+1];
        }
        rec[n-1]->rightSibling=nullptr;
        //翻转下一层
        for(int i=0;i<n;i++){
            if(rec[i]->leftMostChild)
                reverse(rec[i]->leftMostChild);
        }
    }
```

复杂度分析：算法按照层次遍历了每一个结点，每一层的翻转操作所需的时间均和该层的结点数成正比，所以平均时间复杂度为 $O(nlogn)$;算法需要的空间与二叉树的高度成正比，每一层递归需要的空间与该层的结点数成正比，所以平均空间复杂度为 $O(nlogn)$

3.给出下图二叉树（对应的森林）的带右链的先根次序存储表示和带度数的后根次序存储表示。



解:
二叉树对应的森林为



带右链的先根次序存储表示为：

data	A	B	D	H	I	E	J	C	F	K	G
index	0	1	2	3	4	5	6	7	8	9	10
rlink	7	5	4	-1	-1	6	-1	10	9	-1	-1
ltag	0	0	0	1	1	1	1	0	1	1	1

带度数的后根次序存储表示为：

data	H	D	I	B	E	J	A	F	K	C	G
index	0	1	2	3	4	5	6	7	8	9	10
degree	0	1	0	2	0	0	3	0	0	2	0

4.编写算法，读入带度数的层次次序存储表示序列，建立一棵森林。
解:
算法描述：利用队列实现算法。倒序遍历带度数的层次次序存储表示序列，由于是层次遍历，所以序列的最后一定存在度数为0的结点，将这些结点依次入队。当读到第一个度数不为0（设其度数为k）的结点时，从队头取出k个结点，将它们的父结点设置为该度数不为0的结点，之后再将该结点入队。依此类推，直到完成遍历。遍历完整个序列后，若队列中仍然存在结点，将这些结点出队即可（这些结点是森林中单独的树）。

```
//算法代码如下:
template<class T>
class arrNode{
public:
    T data;
```

```
    int degree;
};
template<class T>
class TreeNode{
public:
    T data;
    TreeNode*leftMostChild,*rightSibling;//用左子右兄二叉树的形式存储森林
    TreeNode(T val=0,TreeNode*l=nullptr,TreeNode*r=nullptr):
        data(val),leftMostChild(l),rightSibling(r){}
};
template<class T>
TreeNode<T>*buildForest(arrNode<T>*arr,int size){
    queue<TreeNode<T>*>q;//利用队列存储待寻找父结点的结点
    for(int i=size-1;i>=0;i--){//倒序遍历带度数的层次次序存储表示序列
        TreeNode*ptr=new TreeNode(arr[i].data);
        if(arr[i].degree==0)//叶结点，直接入队
            q.push(ptr);
        else{//非叶结点，度数至少为1
            TreeNode<T>* before=q.front();//先取出一个结点，其必然是最右侧的子
结点                                    //层次遍历从上到下，从左到右完成遍历
            q.pop();
            for(int j=1;j<arr[i].degree;i++){//如果度数大于1则继续从队头取
                TreeNode<T>*tmp=q.front();
                q.pop();
                tmp->rightSibling=before;//设置右兄弟
                before=tmp;
            }
            ptr->leftMostChild=before;//before是最左侧的子结点，即为
leftMostChild
            q.push(ptr);//将该结点入队
        }
    }
    TreeNode<T>* before=q.front();//完成遍历后队列中至少有一个结点，这些结点都
是森林的树根
    q.pop();
    while(!q.empty()){
        TreeNode<T>*tmp=q.front();
        q.pop();
        tmp->rightSibling=before;
        before=tmp;
    }
    return before;//最后一个出队的是左子右兄二叉树的根结点
}
```

复杂度分析：设序列中结点个数为 n 。该算法通过对序列的一次倒序遍历实现，因此时间复杂度为 $O(n)$;算法空间上的开销有两方面，一是队列的长度，二是每一个结点空间上的开销。最坏情况下队列的长度为 n ，结点的开销是固定的，与结点个数成正比。因此空间复杂度为 $O(n)$ 。

5.给定n个表示变量之间关系的方程，每个方程 equations[i] 的采用两种不同的形式之一：“a==b”或

“a!=b”。在这里，a 和 b 表示变量名。
要求设计一个使用并查集的算法，使得只有当方程组有解时返回 true，否则返回false。
并且要求说明，在find操作中
(1)不使用优化，1次find的操作时间代价。
(2)使用“重量权衡合并规则”，1次find的操作时间代价。
(3)使用“路径压缩”后，n次find的操作时间代价。
输入：[“a==b”,“b!=a”]
输出：false
解释：没有办法同时满足这两个方程。
输入：[“b==a", "a==b”]
输出：true
解释：我们可以指定 a = 1 且 b = 1 以满足这两个方程。

解：
算法描述：利用并查集完成算法。两次遍历给定的方程：第一次遍历找到所有形如 $a == b$ 的方程，从而找到n个变量在相等关系下形成的划分并将其用并查集存储起来；第二次遍历找到所有形如 $a! = b$ 的方程,对每一个方程在并查集中查找 a 和 b 对应的根结点。若 a 和 b 对应的根结点相同则说明有 $a = b$ 成立，从而得到方程组无解。若成功完成了两次遍历则说明方程组是有解的。

```
//算法代码如下
template <class T>
class TreeNode{//树结点类定义
public:
    T val;
    TreeNode*parent;
    int count;//以该结点为根的树的结点数
    TreeNode(T v,TreeNode*p=nullptr,int c=1):
        val(v),parent(p),count(c);
};

template<class T>
class Tree{//并查集树定义
public:
    vector<TreeNode<T>*>arr;//存放结点的数组
    TreeNode<T>*find(TreeNode<T>*node){//没有优化的find
        TreeNode<T>*pointer=node;
        while(pointer->parent){
            pointer=pointer->parent;
        }
        return pointer;
    }
    TreeNode<T>*isInTree(T val){//判断值是否已经在树中
        int n=arr.size();
        for(int i=0;i<n;i++){
            if(arr[i].val==val){
                return arr[i];
            }
        }
```

```
        return nullptr;
    }
    bool different(TreeNode<T>*tmp1,TreeNode<T>*tmp2){//判断两个根结点是否
相同
        return tmp1!=tmp2;
    }
    void Union(TreeNode<T>*tmp1,TreeNode<T>*tmp2){//没有优化的归并
        if(different(tmp1,tmp2))
            tmp2->parent=tmp1;
    }
};

template <class T>
bool judgeEquations(vector<string>equations){
    int n=equations.size();
    Tree ParTree;//并查集树
    for(int i=0;i<n;i++){//第一次遍历
        string s=equations[i];
        if(s[1]=='='){//"val1==val2"
            TreeNode<T>*tmp1=isInTree(val1);
            TreeNode<T>*tmp2=isInTree(val2);
            if(!tmp1){//val1不在树中，创建结点并加入树
                tmp1=new TreeNode<T>*(val1);
                ParTree.push_back(tmp1);
            }
            if(!tmp2){//val2不在树中，创建结点并加入树
                tmp2=new TreeNode<T>*(val2);
                ParTree.push_back(tmp2);
            }
            tmp1=ParTree.find(tmp1);//寻找根结点
            tmp2=ParTree.find(tmp2);
            ParTree.Union(tmp1,tmp2);//归并
        }
    }
    for(int i=0;i<n;i++){
        string s=equations[i];
        if(s[1]=='!'){//"val1!=val2"
            TreeNode<T>*tmp1=isInTree(val1);
            TreeNode<T>*tmp2=isInTree(val2);
            if(!tmp1&&!tmp2){//无解的必要条件是val1和val2在并查集树中
                tmp1=ParTree.find(tmp1);
                tmp2=ParTree.find(tmp2);
                if(tmp1==tmp2)//在同一个等价类中
                    return false;
            }
        }
    }
    return true;
}
```

复杂度分析:
(1)不使用优化时，在最坏情况下并查集树是一条链，一次find操作需要的时间代价为 $O(n)$ ；最优情况下所有结点都直接与根相连，一次find操作的时间为 $O(1)$ 。
(2)使用重量权衡合并规则时的Union函数为

```
template <class T>
void Tree<T>::Union(TreeNode<T>*tmp1,TreeNode<T>*tmp2){
    if(!different(tmp1,tmp2)){
        if(tmp1->count>=tmp2->count){
            tmp2->parent=tmp1;
            tmp1->count+=tmp2->count;
        }
        else{
            tmp1->parent=tmp2;
            tmp2->count+=tmp1->count;
        }
    }
}
```

每次将小树合并到大树中，这样可以保证将树的高度限制在 $O(logn)$,每次find操作需要的时间是 $O(logn)$ 的。
(3)使用路径压缩的find函数为

```
template <class T>
TreeNode<T>* Tree<T>::find(TreeNode<T>*node){
    if(node->parent==nullptr)
        return node;
    node->parent=find(node->parent);
    return node->parent;
}
```

使用路径压缩后n次find操作的时间复杂度为 $O(n\alpha(n)), \alpha(n) = Ackermann^{-1}(n)$