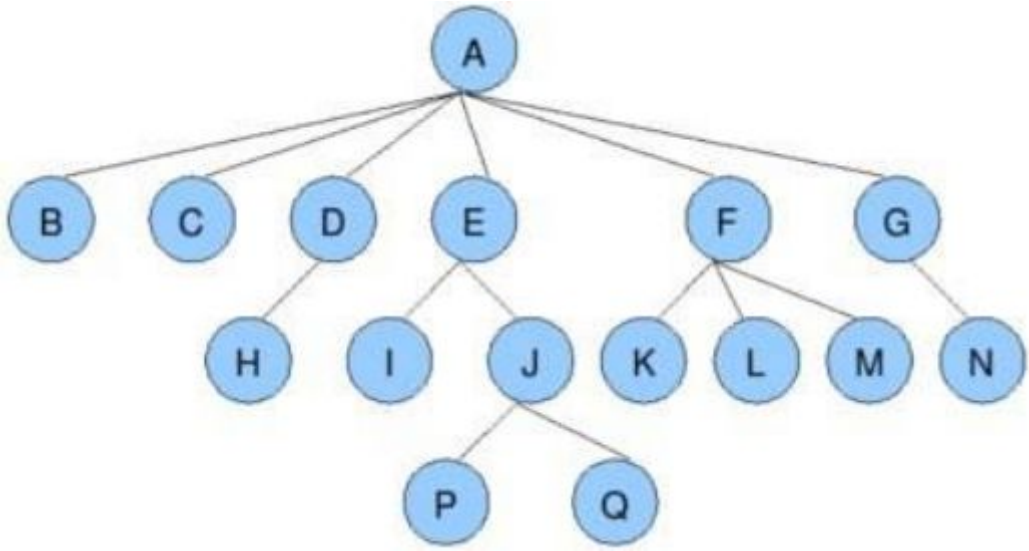
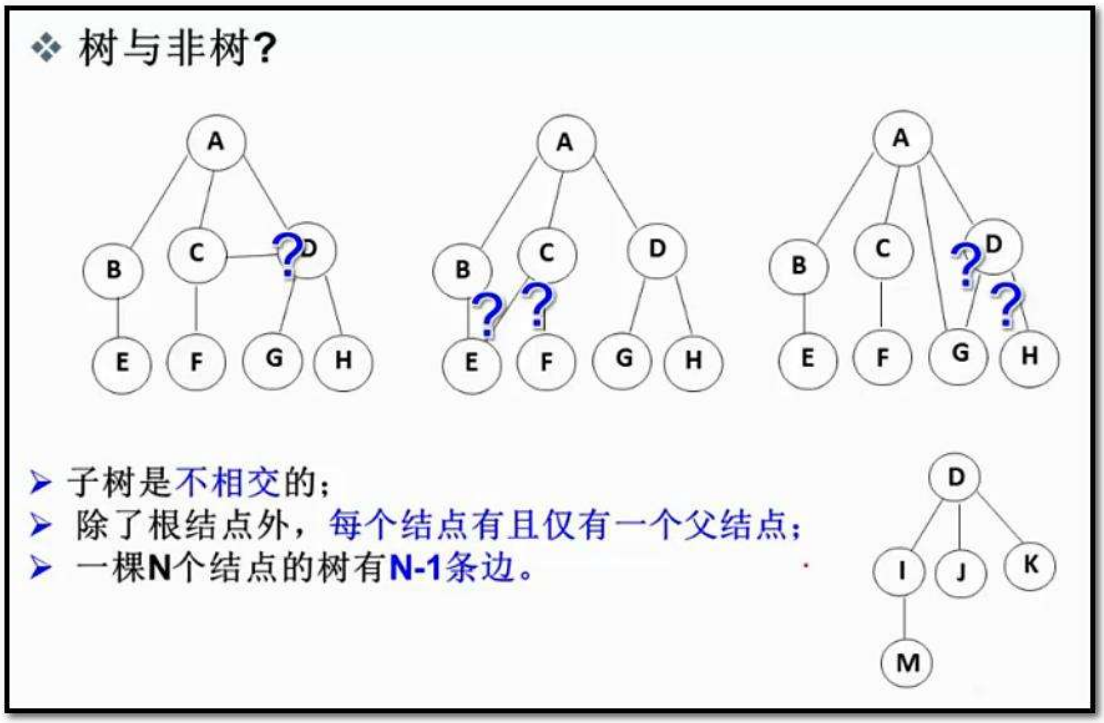
1.树概念及结构(了解)

1.1树的概念



- 节点的度：一个节点含有的子树的个数称为该节点的度； 如上图：A的为6
- 叶节点或终端节点：度为0的节点称为叶节点； 如上图：B、C、H、I...等节点为叶节点
- 非终端节点或分支节点：度不为0的节点； 如上图：D、E、F、G...等节点为分支节点
- 双亲节点或父节点：若一个节点含有子节点，则这个节点称为其子节点的父节点； 如上图：A是B的父节点
- 孩子节点或子节点：一个节点含有的子树的根节点称为该节点的子节点； 如上图：B是A的孩子节点
- 兄弟节点：具有相同父节点的节点互称为兄弟节点； 如上图：B、C是兄弟节点
- 树的度：一棵树中，最大的节点的度称为树的度； 如上图：树的度为6
- 节点的层次：从根开始定义起，根为第1层，根的子节点为第2层，以此类推；
- 树的高度或深度：树中节点的最大层次； 如上图：树的深度为4（高度是3，空树的高度是-1）
- 节点的祖先：从根到该节点所经分支上的所有节点；如上图：A是所有节点的祖先
- 子孙：以某节点为根的子树中任一节点都称为该节点的子孙。如上图：所有节点都是A的子孙
- 森林：由m（m>0）棵互不相交的多颗树的集合称为森林；（数据结构中的学习并查集本质就是一个森林）



1.2树的表示

树结构相对线性表就比较复杂了，要存储表示起来就比较麻烦了，实际中树有很多种表示方式，如：双亲表示法，孩子表示法、孩子兄弟表示法等等。我们这里就简单的了解其中最常用的**孩子兄弟表示法。**

左孩子右兄弟

```
typedef int DataType;
struct Node
{
    struct Node* _firstChild1;    // 第一个孩子结点
    struct Node* _pNextBrother;   // 指向其下一个兄弟结点
    DataType _data;               // 结点中的数据域
};
```

双亲表示法：存双亲的下标

1.3树在实际中的运用（表示文件系统的目录树结构）

2.二叉树概念及结构

2.1概念

一棵二叉树是结点的一个有限集合，该集合或者为空，或者是由一个根节点加上两棵别称为左子树和右子树的二叉树组成。

二叉树的特点：

1. 每个结点最多有两棵子树，即二叉树不存在度大于2的结点。
2. 二叉树的子树有左右之分，其子树的次序不能颠倒。

2.2特殊的二叉树：

1. 满二叉树：一个二叉树，如果每一个层的结点数都达到最大值，则这个二叉树就是满二叉树。也就是说，如果一个二叉树的层数为K，且结点总数是(2^k) -1 ，则它就是满二叉树。

2. 完全二叉树：完全二叉树是效率很高的数据结构，完全二叉树是由满二叉树而引出来的。对于深度为K的，有n个结点的二叉树，当且仅当其每一个结点都与深度为K的满二叉树中编号从1至n的结点一一对应时称之为完全二叉树。 要注意的是满二叉树是一种特殊的完全二叉树。(最后一层不满，但是最后一层从左往右都是连续的)

> 完全二叉树的特点：
> ①叶子节点只可能出现在最后两层
> ②度为1的结点个数为0或1
> ③具有n个结点的完全二叉树的深度为floor(log2n)+1或ceil(log2(n+1))

2.3 二叉树的存储结构

二叉树一般可以使用两种结构存储，一种顺序结构，一种链式结构。

二叉树的性质

1. 若规定根节点的层数为1，则一棵非空二叉树的第i层上最多有2^(i-1) 个结点.
2. 若规定根节点的层数为1，则深度为h的二叉树的最大结点数是2^h- 1.
3. 对任何一棵二叉树，如果度为0其叶结点个数为 n0，度为2的分支结点个数为 n2,则有n0＝n2＋1（叶子节点数，比度为2的结点数多一个）
4. 若规定根节点的层数为1，具有n个结点的满二叉树的深度，h=LogN

二叉树性质相关选择题练习

1. 某二叉树共有 399 个结点，其中有 199 个度为 2 的结点，则该二叉树中的叶子结点数为（ ）

A 不存在这样的二叉树

B 200

C 198

D 199

2.在具有 2n 个结点的完全二叉树中，叶子结点个数为（ ）

A n

B n+1

C n-1

D n/2

解析：假设度为0的有X0个，度为1的有X1个，度为2的有X2个，X0+X1+X2=2n,又因为X0=X2+1，所以2X0+X1-1=2n，完全二叉树度为1的结点个数为0或1，X1可能是1，也可能是0

3.一棵完全二叉树的节点数位为531个，那么这棵树的高度为（ ）

A 11

B 10

C 8

D 12

2.3.1 顺序存储：

顺序结构存储就是使用**数组来存储**，一般使用数组**只适合表示完全二叉树**，因为不是完全二叉树会有空间的浪费。而现实中使用中只有堆才会使用数组来存储，关于堆我们后面的章节会专门讲解。**二叉树顺序存储在物理上是一个数组，在逻辑上是一颗二叉树。**

2.3.2 链式存储：

二叉树的链式存储结构是指，用链表来表示一棵二叉树，即用链来指示元素的逻辑关系。 通常的方法是链表中每个结点由三个域组成，数据域和左右指针域，左右指针分别用来给出该结点左孩子和右孩子所在的链结点的存储地址 。链式结构又分为二叉链和三叉链，当前我们学习中一般都是二叉链，后面课程学到高阶数据结构如红黑树等会用到三叉链

```
// 二叉链
struct BinaryTreeNode
{
    struct BinTreeNode* pLeft;   // 指向当前节点左孩子
    struct BinTreeNode* pRight; // 指向当前节点右孩子
    BTDataType _data; // 当前节点值域
```

```
}

// 三叉链 -- AVL树，红黑树
struct BinaryTreeNode
{
    struct BinTreeNode* pParent; // 指向当前节点的双亲
    struct BinTreeNode* pLeft;   // 指向当前节点左孩子
    struct BinTreeNode* pRight; // 指向当前节点右孩子
    BTDataType _data; // 当前节点值域
};
```

3.二叉树链式结构的实现

二叉树链式结构的遍历

前序/中序/后序的递归结构遍历：

是根据访问结点操作发生位置命名

1. NLR：前序遍历(Preorder Traversal 亦称先序遍历)——访问根结点的操作发生在遍历其左右子树之前。
2. LNR：中序遍历(Inorder Traversal)——访问根结点的操作发生在遍历其左右子树之中（间）。
3. LRN：后序遍历(Postorder Traversal)——访问根结点的操作发生在遍历其左右子树之后。

由于被访问的结点必是某子树的根，**所以N(Node)、L(Left subtree)和R(Right subtree)又可解释为根、根的左子树和根的右子树。**NLR、LNR和LRN分别又称为先根遍历、中根遍历和后根遍历。（深度优先遍历），一般都是递归

分治算法：分而治之，大问题分成类似子问题，子问题再分成子问题...直到子问题不可再分隔

```
前序遍历：A B D NULL NULL E NULL NULL C NULL NULL
中序遍历：NULL D NULL B NULL E NULL A NULL C NULL
后序遍历：NULL NULL D NULL NULL E B NULL NULL C A
```

```
前序遍历：A B NULL D F NULL NULL NULL C E NULL NULL NULL
中序遍历：NULL B NULL F NULL D NULL A NULL E NULL C NULL
后序遍历：NULL NULL NULL F NULL D B NULL NULL E NULL C A
```

```
typedef struct BinaryTreeNode
{
```

```
    struct BinaryTreeNode* left;
    struct BinaryTreeNode* right;
    BTData data;
}BTNode;
//前序遍历
void PrevOrder(BTNode* root)
{
    if (root == NULL)
    {
        printf("NULL ");
        return;
    }

 printf("%c ", root->data);
 PrevOrder(root->left);
 PrevOrder(root->right);
}

//中序遍历
void InOrder(BTNode* root)
{
    if (root == NULL)
    {
        printf("NULL ");
        return;
    }
    InOrder(root->left);
    printf("%c ", root->data);
    InOrder(root->right);
}

//后序遍历
void PostOrder(BTNode* root)
{
    if (root == NULL)
    {
        printf("NULL ");
        return;
    }
    PostOrder(root->left);
```

```
    PostOrder(root->right);
    printf("%c ", root->data);

}
```

层序遍历：

除了先序遍历、中序遍历、后序遍历外，还可以对二叉树进行层序遍历。设二叉树的根节点所在层数为1，层序遍历就是从所在二叉树的根节点出发，首先访问第一层的树根节点，然后从左到右访问第2层上的节点，接着是第三层的节点，以此类推，自上而下，自左至右逐层访问树的结点的过程就是层序遍历。（广度优先遍历），队列实现

```
//层序遍历 -- 上一层出的时候，带下一层结点进
void LevelOrder(BTNode* root)
{
    Queue q;
    QueueInit(&q);
    if (root)
        QueuePush(&q, root);

    while (!QueueEmpty(&q))
    {
        //根先进队，再出队
        BTNode* front = QueueFront(&q);
        QueuePop(&q);
        printf("%c ", front->data);

        if (front->left)
        {
            QueuePush(&q, front->left);
        }
        if (front->right)
        {
            QueuePush(&q, front->right);
        }
    }
    printf("\n");
    QueueDestroy(&q);
}
```

选择题

1.某完全二叉树按层次输出（同一层从左到右）的序列为 ABCDEFGH 。该完全二叉树的前序序列为（）

A ABDHECFG

B ABCDEFGH

C HDBEAFCG

D HDEBFGCA

2.二叉树的先序遍历和中序遍历如下：先序遍历：EFHIGJK;中序遍历：HFIEJKG.则二叉树根结点为（）

A E

B F

C G

D H

3.设一课二叉树的中序遍历序列：badce，后序遍历序列：bdeca，则二叉树前序遍历序列为____。

A adbce

B decab

C debac

D abcde

求节点个数

```
//结点个数（遍历思维）--递归
int TreeSize(BTNode* root,int* psize)
{
    if (root == NULL)
        return;
    else
        ++(*psize);

    TreeSize(root->left, psize);
    TreeSize(root->right, psize);

}

```

```
//分治算法（后序思维）--递归
int TreeSize2(BTNode* root)
{
    return root == NULL ? 0 : TreeSize2(root->left) + TreeSize2(root->right) + 1;
}

//叶子结点的个数--递归
int TreeLeafSize(BTNode* root)
{
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 1;
    return TreeLeafSize(root->left) + TreeLeafSize(root->right);
}
```

二叉树常见OJ题练习

1. 二叉树的前序遍历144. 二叉树的前序遍历 - 力扣（LeetCode）

```
int TreeSize(BTNode* root)
{
    return root == NULL ? 0 : TreeSize(root->left) + TreeSize(root->right) + 1;
}

//void _PreOrder(BTNode* root, int* a, int i)
//{
//      if (root == NULL)
//              return;
//      a[i] = root->data;
//      ++i;
//      //每一层递归函数都有一个i，下一层放了值，++i，不会影响上一层函数中的i
//      _PreOrder(root->left, a, i);
//      _PreOrder(root->right, a, i);
//}

void _PreOrder(BTNode* root, int* a, int* pi)
{
    if (root == NULL)
```

```
        return;
    a[*pi] = root->data;
    ++(*pi);
    _PreOrder(root->left, a, pi);
    _PreOrder(root->right, a, pi);
}


int* preorderTraversal(BTNode* root, int* returnSize)
{
    int size = TreeSize(root);
    //开辟大小为树的结点个数的数组
    int* a = (int*)malloc(size * sizeof(int));
    int i = 0;
    _PreOrder(root, a, &i);
    *returnSize = size;
    return a;
}
```

2. 二叉树的中序遍历：94. 二叉树的中序遍历 - 力扣（LeetCode）
3. 二叉树的后序遍历：145. 二叉树的后序遍历 - 力扣（LeetCode）
4. 二叉树的最大深度：Loading Question... - 力扣（LeetCode）

```
int maxDepth(BTNode* root)
{
    if (root == NULL)
        return 0;
    int leftDepth = maxDepth(root->left);
    int rightDepth = maxDepth(root->right);
    return  leftDepth > rightDepth ? leftDepth + 1:rightDepth +1;
}
```

5. 平衡二叉树：力扣 (leetcode.cn)

```
bool isBalanced(BTNode* root)
{
    if (root == NULL)
        return true;
    int leftDepth = maxDepth(root->left);
```

```
    int rightDepth = maxDepth(root->right);
    return  abs(leftDepth - rightDepth) < 2
        && isBalanced(root->left)
        && isBalanced(root->right);//都满足才满足，一个不满足就不满足
}
```

6. 二叉树的层序遍历：102. 二叉树的层序遍历 - 力扣（LeetCode）

清华面试上机题：

```
//给先序遍历重构二叉树
typedef struct BinaryTreeNode
{
    struct BinaryTreeNode* left;
    struct BinaryTreeNode* right;
    char data;
}BTNode;

BTNode* CreateTree(char* a, int* pi)
{
    if (a[*pi] == '#')
    {
        ++(*pi);
        return NULL;
    }
    BTNode* root = (BTNode*)malloc(sizeof(BTNode));
    if (root == NULL)
    {
        printf("Malloc fail!\n");
        exit(-1);
    }
    root->data = a[*pi];
    ++(*pi);
    //递归
    root->left = CreateTree(a, pi);
    root->right= CreateTree(a, pi);
    return root;
}

//中序遍历
```

```c
void InOrder(BTNode* root)
{
    if (root == NULL)
        return;
    InOrder(root->left);
    printf("%c ", root->data);
    InOrder(root->right);
}


int main()
{
    char str[100];
    scanf("%s", str);
    int i = 0;
    BTData* root = CreateTree(str, &i);
    return 0;
}
```