+树状结构：

一棵自然界中的树　　一棵逻辑上的树

图 3-55 树

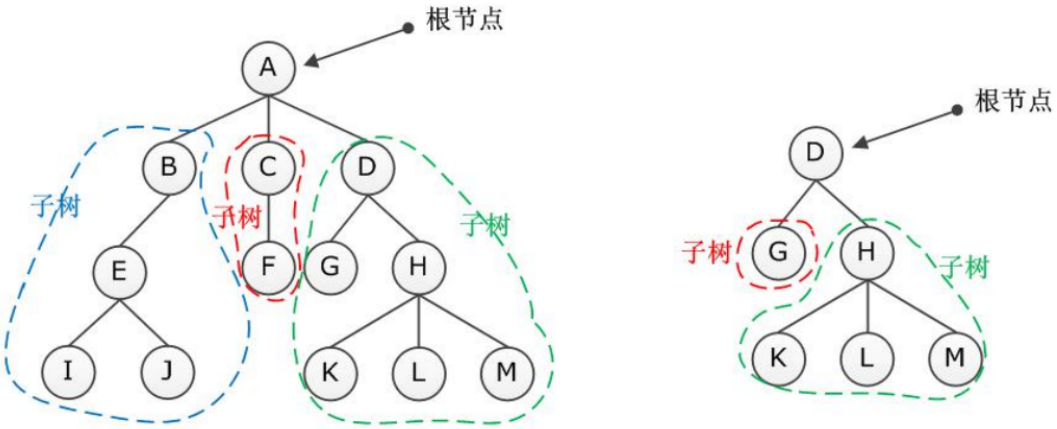


图 3-56 树和子树

A) 双亲（parent）和孩子（children）：一个节点的后继节点被称为该节点的孩子，相应地该节点被称为这些孩子的双亲。比如上图中的 A 是 B、C 和 D 的双亲，而 B、C 和D 都是 A 的孩子。

B) 兄弟（sibling）：拥有共同双亲的节点互为兄弟节点。比如 I 和 J，或 K、L 和 M。

C) 节点的度（degree）：一个节点的孩子个数，称为该节点的度。比如 A 的度为 3，而 C 的度为 1。

D) 节点的层次（level）：人为规定树的根节点的层次为 1，他的后代节点的层次依次加 1。比如节点 A 的层次为 1，而节点 B、C 和 D 的层次均为 2，节点 E、F、G 和 H 为 3，以此类推。

E) 树的高度（height）：树中节点层次的最大值。比如在以 A 为根的树中，节点层次的最大值为 4，因此以 A 为根的树的高度是 4。树的高度也称为树的深度（depth），树的高度是后面衡量一棵树平衡性的重要依据。

F) 终端节点（terminal）：所谓的终端当然指的是最末端的叶子（leaf）节点，严格的定义是度为 0 的节点。比如 I、J、K、L、M 都是叶子。

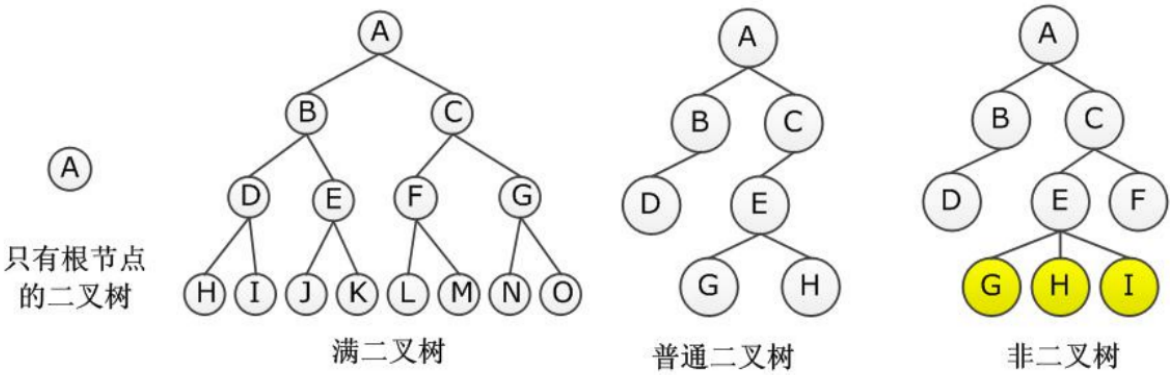
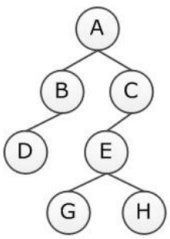


图 3-57 各种二叉树

1. 二叉树是树状结构中比较常见的一个种树状结构， 二叉树的每一个节点的度（节点的孩子个数）最大不可以超过2，如果超过2则不称为二叉树。
2. 必须是一棵有序树，即节点的孩子是有次序的，哪怕只有一个孩子，也要严格区分左右两个分叉
3. 其节点的个数达到了 4 层的最大值 15，即：如果一棵二叉树有 K 层，而他的节点个数达到最大值 2 K（次方） -1 个的话，那么这个二叉树被称为满二叉树

二叉树的遍历：

前序： 先遍历根节点（父母节点），然后遍历左孩子， 最后遍历右孩子。
中序： 先遍历左孩子， 然后遍历根节点， 最后遍历右孩子 。
后序： 先遍历左孩子， 然后遍历右孩子，之后遍历根节点
按层： 按树的每一层来遍历（高度）兄弟节点（使用队列来实现）



先序: A B D C E G H
中序: D B A G E H C
后序: D B G H E C A

笔试题1： 已知一棵二叉树的前序遍历的结果是ABECDFGHIJ，中序遍历的结果是EBCDAFHIGJ，试画出这棵二叉树，并给出这棵二叉树的后序遍历序列。

已知二叉树后序遍历序列是dabec，中序遍历序列是debac，它的前序遍历序列是（cedba）

已知一棵二叉树前序遍历和中序遍历分别为ABDEGCFH和DBGEACHF，则该二叉树的后序遍历为（DGEBHFCA）

若某二叉树的前序遍历访问顺序是abdgcefh，中序遍历访问顺序是dgbaechf，则其后序遍历的结点访问顺序是（gdbehfca）

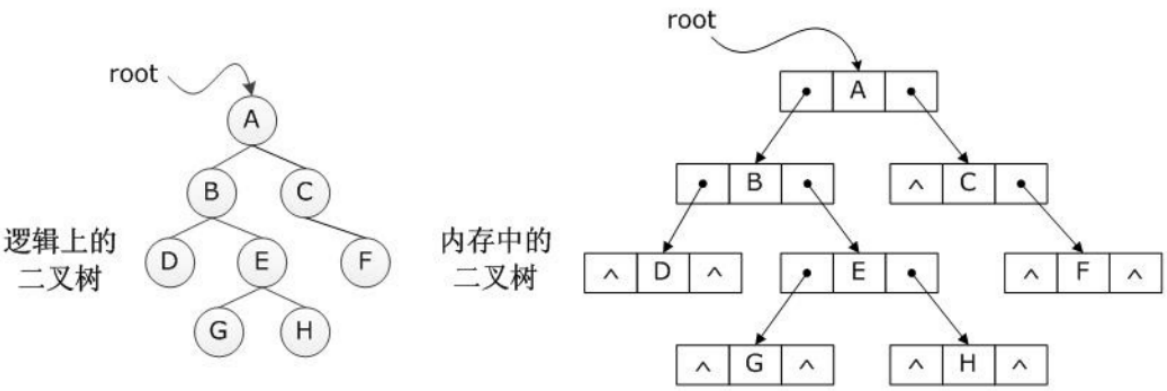
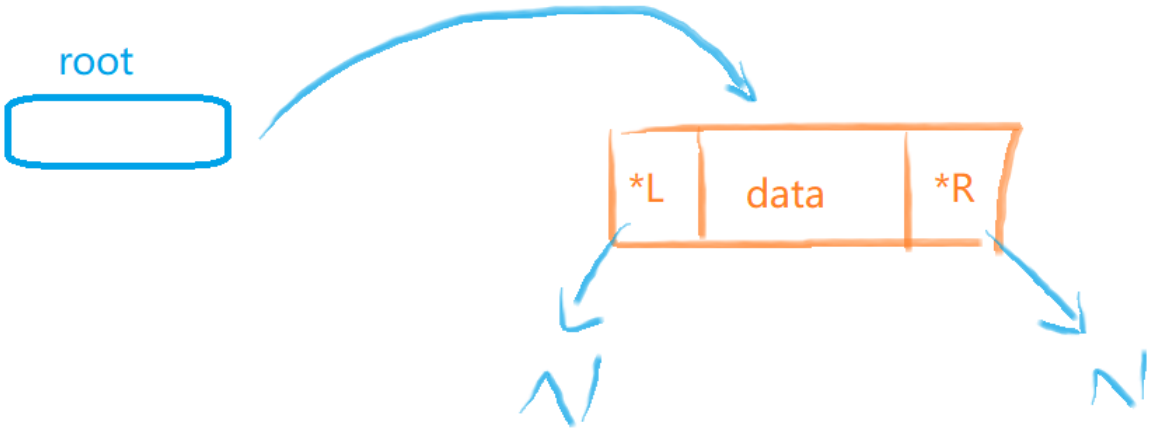


图 3-58 二叉树的链式存储方式

设计节点：


```
// 节点设计
typedef struct tree
{
    int data ;
    struct tree * L  , * R ;
}Tree , *P_Tree ;
```

初始化：




```
1
2  P_Tree new_node_init(int data)
3  {
4      P_Tree new = calloc(1, sizeof(Tree));
5
6      new->data = data ;
7
8      new->L = new->R = NULL ;
9
10     return new ;
11 }
```

添加数据：

```
1
2  P_Tree add_2_tree( P_Tree root , P_Tree new)
3  {
4      if ( root == NULL )
5      {
6          // printf("节点异常！！\n");
7          return new ; // 当root 为空的时候说明后面没有数据，
8                       //当前数据为 相对最大 或 相对最小
9      }
10
```

```
    if (root->data < new->data) // 如果新节点的数据比根节点大则往右走
    {
        root->R = add_2_tree( root->R  ,  new);
    }
    else // 否则往左边走 （递归调用字节 传递的是自己当前的右脚/左脚）
    {
        root->L = add_2_tree( root->L  ,  new);
    }

    return root ;
}

```

中序遍历：

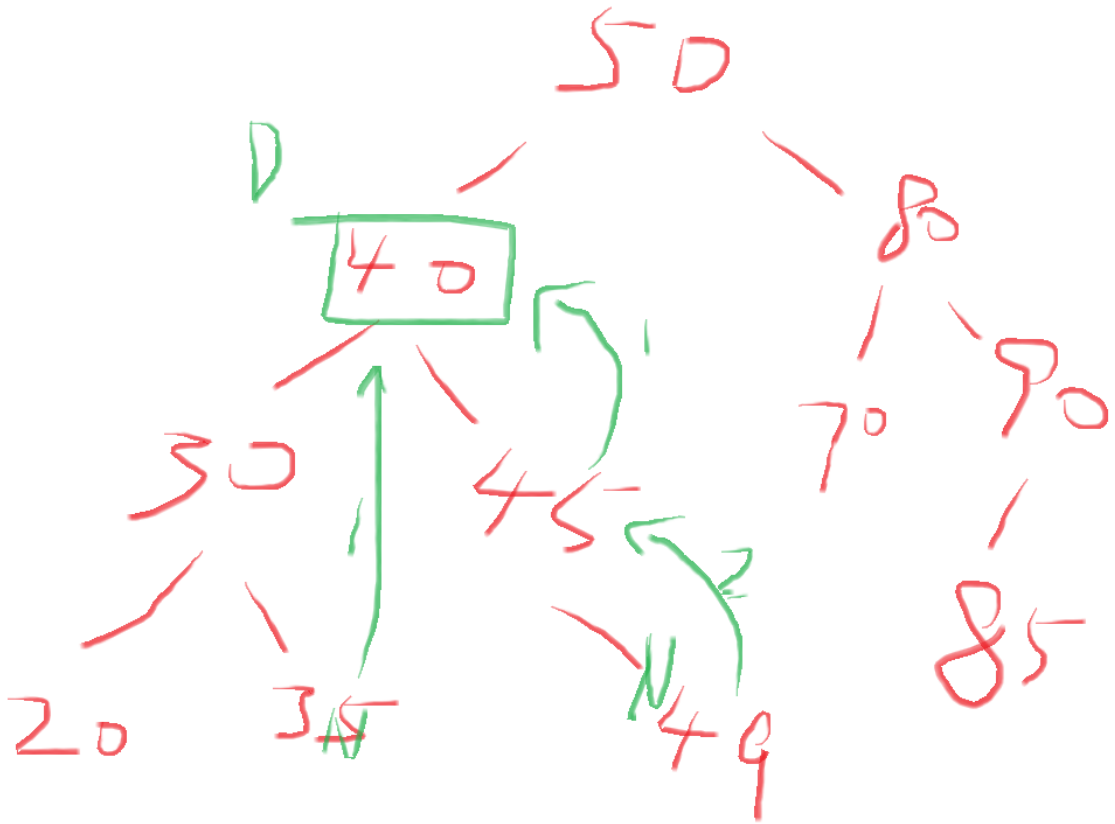
```

void tree_4_each( P_Tree root )
{
    if (root == NULL )
        return ;

    tree_4_each(  root->L ) ;
    printf("%d\t" , root->data) ;
    tree_4_each(  root->R ) ;

    return ;
}
```

删除节点：



思路：

1. 找到需要删除的节点
2. 找一个合适的节点来替换需要删除的节点
    a. 右边最小的节点
    b. 左边最大的节点
3. 删除替换的节点

```
1
2  P_Tree del_4_tree(P_Tree root , int data)
3  {
4      if (root == NULL )
5      {
6          return NULL ;
7      }
8
9      P_Tree tmp = NULL  ;
10
11     if ( data < root->data )
12     {
13         /* 往左边找 */
14         root->L = del_4_tree( root->L   ,  data);
15     }
16     else if (data > root->data )
```

```
    {
        /* 往右边找 */
        root->R = del_4_tree( root->R  ,  data);
    }
    else
    {
        printf("del data: %d \n" , root->data );

        /* 找到 需要删除的节点 */
        if (root->L == NULL && root->R == NULL ) // 判断需要删除的节点是否为叶子节点
        {
            free(root);
            root =  NULL ;
            return root ;
        }
        else if (root->L != NULL ) // 如果不是叶子节点则 检查左脚是否为空 , 如果不为空则在左边
        {
            /* 如果有左孩子则使用左边最大的来替换 */
            for ( tmp = root->L ; tmp->R != NULL ; tmp = tmp->R );// for循环在寻找左边最大
            root->data =  tmp->data ; // 替换数据

            root->L =  del_4_tree( root->L , tmp->data );// 删除替换的节点
        }
        else /* 如果只有右子树则使用右边最小的来替换*/
        {
            for ( tmp = root->R ; tmp->L != NULL ; tmp = tmp->L );// for循环在寻找右边最小
            root->data =  tmp->data ; // 替换数据

            root->R = del_4_tree( root->R , tmp->data );// 删除替换的节点
        }

    }

    return root ; // 返回当前栈空间中的root 指针值
}
```

按层遍历：

遍历方式，叫按层遍历，就是从上到下，从左到右地遍历每一个节点。这种遍历方式需要用到队列逻辑，具体的流程是这样的：

1，创建一个空队列。

2，将树的根节点入队。

3，判断队列是否为空，如果为空，则遍历结束，否则出队队头元素。

4，访问该队头元素。

5，如果该队头元素左孩子不为空，则将其左孩子入队。

6，如果该队头元素右孩子不为空，则将其右孩子入队。

7，重复第 3 步

作业：

1. 尝试销毁二叉树
2. 完善其它遍历方式（先序/后序）