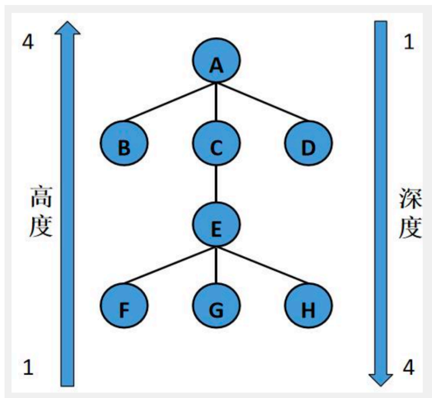
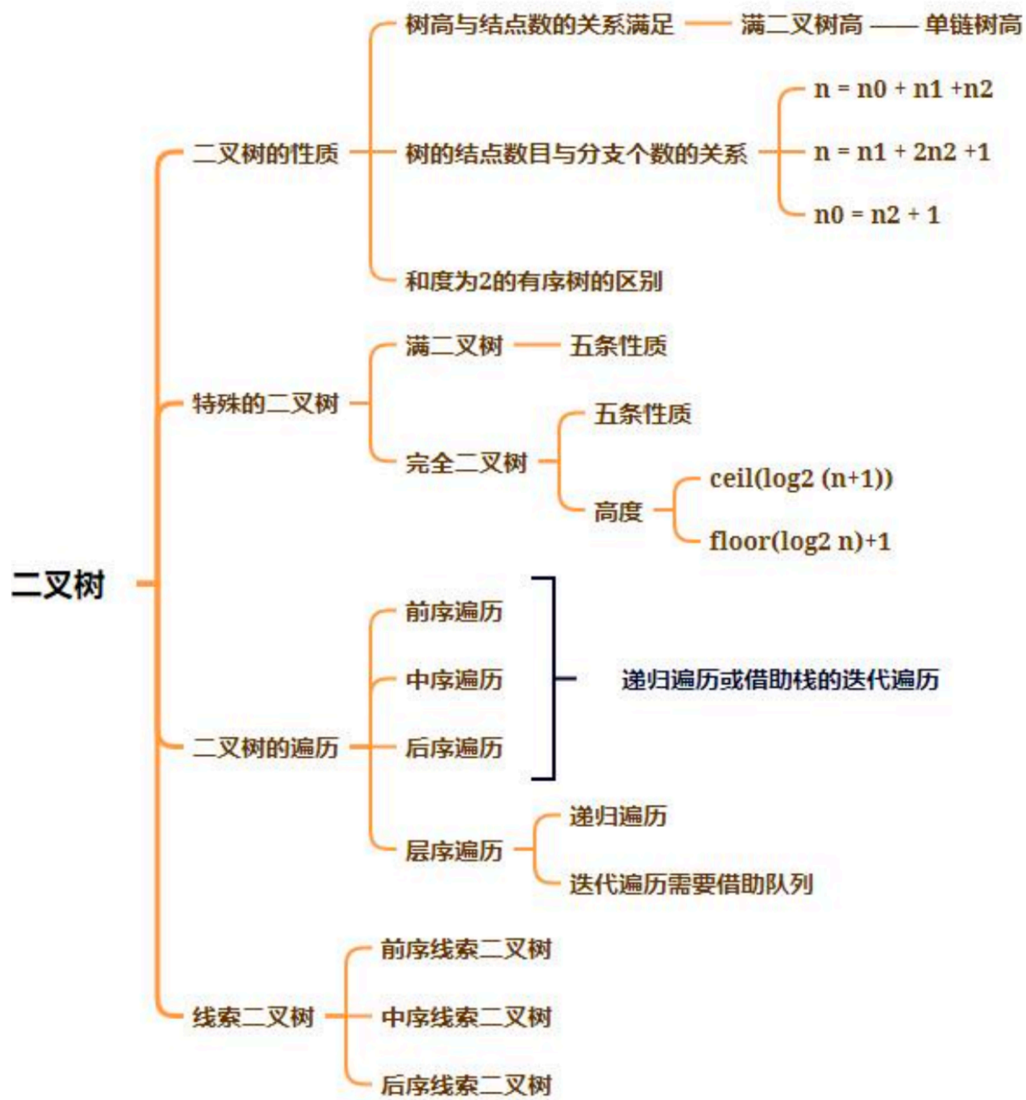
强化_2

树的深度（高度）：树中结点的最大层次，右图中的树的深度为4。
结点的度：一个结点含有的子树的个数称为该结点的度。
树的度：一棵树中，最大的结点的度称为树的度，右图中的树的度为3。
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。
路径：由一系列结点组成的序列，方向为从上到下。
路径长度：路径中包含的边数。

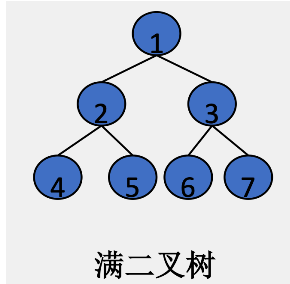


二叉树

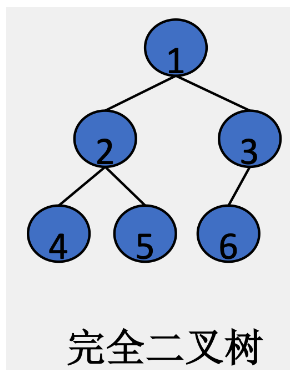


特殊的二叉树

1. 高为h的满二叉树上一共有2h-1个结点。
2. 高为h的满二叉树上，每层都有2h-1个结点。
3. 高为h的满二叉树上，所有的叶子结点都在最后一层。
4. 高为h的满二叉树上，除叶子结点外，每个结点的度都为2.
5. 高为h的满二叉树上，对每个结点从上到下，从左到右进行编号（从1开始），对于任意编号i，若有双亲，则其双亲结点的编号一定是 $\lfloor \frac{i}{2} \rfloor$，若有孩子结点，则左孩子编号为2i，右孩子编号为2i+1。



1. 高为h，有n个结点的完全二叉树上，编号与满二叉树的一一对应。
2. 高为h，有n个结点的完全二叉树上，若结点编号 $i > \lfloor \frac{n}{2} \rfloor$，则该结点一定是叶子结点，否则是非叶子结点。
3. 高为h，有n个结点的完全二叉树上，叶子结点只会处于最后一层和倒数第二层。
4. 高为h，有n个结点的完全二叉树上，只可能存在一个结点度为1并且它肯定只有左孩子没有右孩子。
5. 高为h，有n个结点的完全二叉树上，若n为奇数则所有结点度都为2，若为偶数，则有一个结点度为1。



具有n个（n>0）结点的完全二叉树的高度为 $\lceil log2(n+1) \rceil$ 或 $\lfloor log_2 n + 1 \rfloor$。

证明：设完全二叉树的高度为h，则有 $2^{h-1} - 1 < n \leq 2^h - 1$，或 $2^{h-1} \leq n < 2^h$，得 $2^{h-1} < n + 1 \leq 2^h$，即 $h - 1 < log_2(n+1) \leq h$，故 $h = \lceil log_2(n+1) \rceil$ 或得 $h - 1 \leq log_2 n < h$，故 $h = \lfloor log_2 n \rfloor + 1$。

【例题】高度为h的含有度为1的结点的完全二叉树，最少有多少个结点？最多有多少个结点?

当第h层只有一个结点时，总的结点个数最少，也就是前h-1层是棵满二叉树，此时 $n = 2^{h-1} - 1 + 1 = 2^h - 1$；

当第h层差一个铺满的时候，总的结点个数最多，此时 $n = 2^h - 1 - 1 = 2^h - 2$

注：如果题干没有说带不带度为1的结点，此题就要考虑到满二叉树的情况。

迭代前序序列

```c
int* preorderTraversal(TreeNode* root, int* returnSize) {
    int* res=malloc(sizeof(int)*100);
    *returnSize=0;
    if(root==NULL) {    // 空树
        return res;
    }
    TreeNode* stk[100];
    TreeNode* node=root;
    int top=0;
    while(top>0 || node!=NULL) {
        if(node!=NULL) {
            res[(*returnSize)++]=node->val;
            stk[top++]=node;
            node=node->left;
        }else {
            node=stk[--top];
            node=node->right;
        }
    }
    return res;
}
```

迭代中序序列

```
int* inorderTraversal(TreeNode* root, int* returnSize) {
    int* res=malloc(sizeof(int)*100);
    *returnSize=0;
    if(root==NULL) {    // 空树
        return res;
    }
    TreeNode* stk[100];
    TreeNode* node=root;
    int top=0;
    while(top>0 || node!=NULL) {
        if(node!=NULL) {
            stk[top++]=node;
            node=node->left;
        }else {
            node=stk[--top];
            res[(*returnSize)++]=node->val;
            node=node->right;
        }
    }
    return res;
}
```

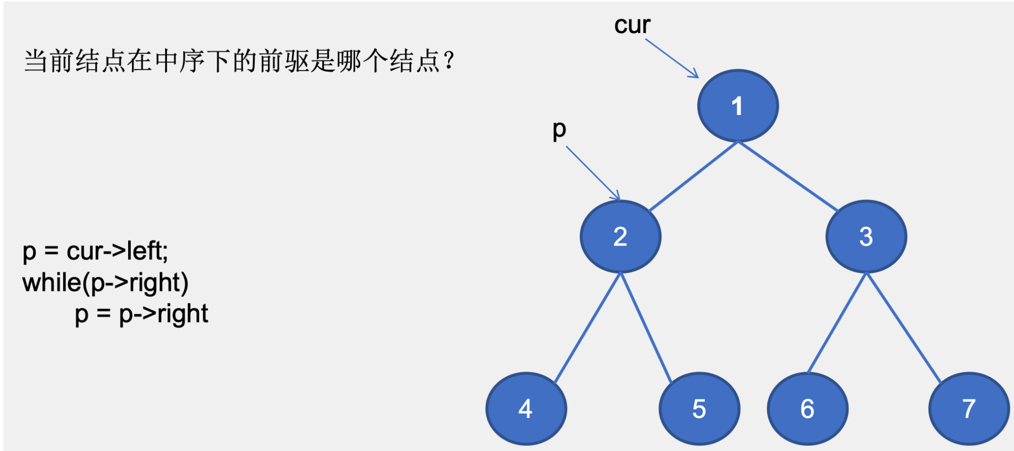
还可以有更加高效的算法完成二叉树的遍历吗?

Morris（莫里斯）遍历

思想：利用二叉树中大量的空闲指针，这里主要是利用一些空闲的右指针来提高遍历的效率。普通的遍历算法，不论递归遍历还是迭代遍历，都需要借助栈这个结构来完成回溯的操作，这里就是优化了这个回溯的操作。

算法：

1. 如果当前结点的左子树为空，向右遍历；否则找到当前结点的左子树中最右的结点（也就是中序遍历下的直接前驱结点）。
2. 找到当前结点的左子树的最右结点后：
    i. 若该结点的右指针为空，则将右指针指向当前结点（也就是这棵左子树的根结点）；
    ii. 若该结点的右指针已经指向当前结点了（已经线索化过了），则将右指针置空。
3. 完成线索化后，当前结点向左遍历一个结点。

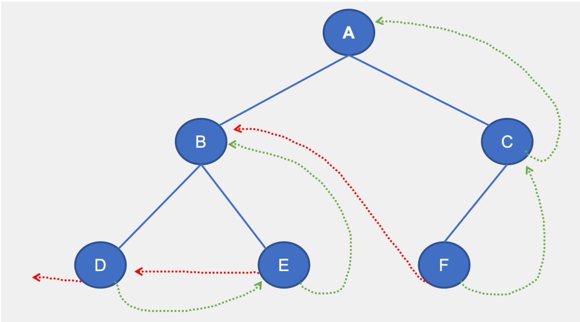



```
if(p->right==NULL) {    // 建立线索
    p->right=cur;
    res[(*returnSize)++]=cur->val;
    cur=cur->left;
    continue;
}
```

```c
int* preorderTraversal(TreeNode* root, int* returnSize) {
    int* res=malloc(sizeof(int)*100);
    *returnSize=0;
    // int index=0;
    if(root==NULL) return res;
    TreeNode *cur=root;
    TreeNode *p=NULL;
    while(cur!=NULL) {
        p=cur->left;
        if(p!=NULL) {   // 当前结点的左子树不空
            while(p->right!=NULL && p->right!=cur) {    // 寻找左子树的最右结点
                p=p->right;
            }
            if(p->right==NULL) {    // 建立线索
                p->right=cur;
                res[(*returnSize)++]=cur->val;
                cur=cur->left;
                continue;
            }else { // 清除线索
                p->right=NULL;
            }
        }else {
            res[(*returnSize)++]=cur->val;
        }
        cur=cur->right;
    }
    return res;
}
```

线索二叉树

根本目的是为了弥补普通的链式存储二叉树难以寻找其直接前驱/后继的缺点，引入线索之后加快查找前驱/后继的速度，但依旧存在弊端。



后序线索二叉树如何找前驱?

1. 该结点为根结点
2. 该结点为双亲的左孩子
    i. 双亲没有右子树
    ii. 双亲有右子树
3. 该结点为双亲的右孩子

线索二叉树	先序线索二叉树	中序线索二叉树	后序线索二叉树
找前驱	×	√	√
找后继	√	√	×

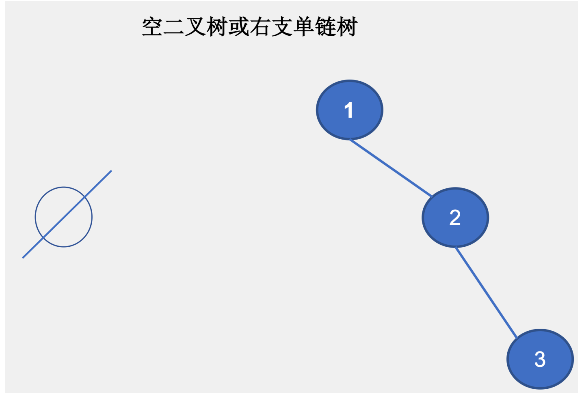
先序线索二叉树找前驱，后序线索二叉树找后继，依旧需要用到栈。

树

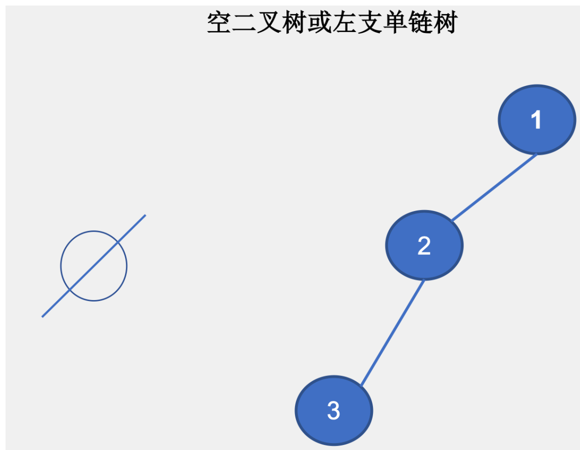
【例题】一棵有n个结点的k（k>=2）叉树, 用链式存储方式表示，有多少个空指针？写出推导过程。

k叉树中每个结点都有k个指针域，所以指针总个数为nk，而n个结点会被n-1个指针指向，所以空指针为 $nk-n+1=n(k-1)+1$

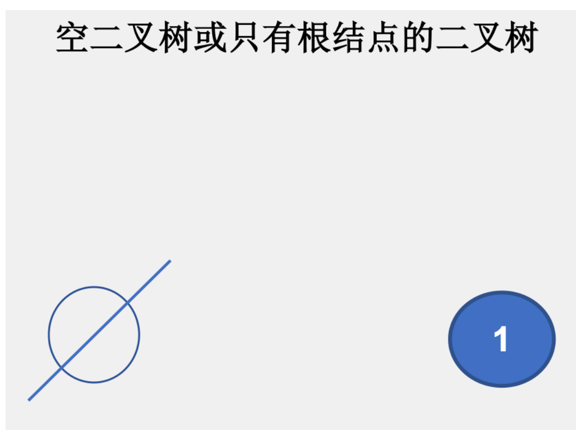
前序序列和中序序列相同的二叉树是什么?



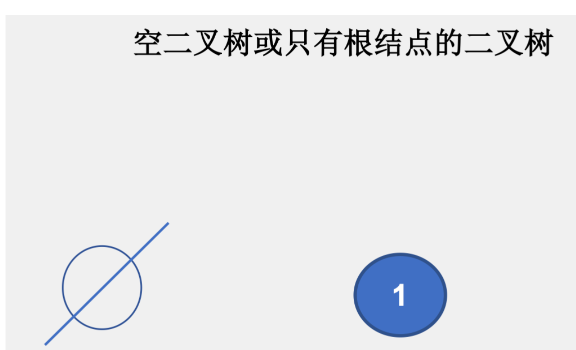
中序序列和后序序列相同的二叉树是什么?



前序序列和后序序列相同的二叉树是什么?



前，中和后序序列相同的二叉树是什么?



【例题】已知二叉树T，求出其深度。先写出结点数据结构定义（值域为int类型)，用c或c++实现该算法。

```
int BTdepth(BiTree *T){ //求二叉树深度
    if(! T){ //空树
        return 0;
    }
    int front=-1, rear=-1; //声明队列
    int last=0, level=0;
    BiTree Q[MaxSize];
    BiTree *p;
    Q[++rear] = T;
    while(front < rear){
        p = Q[++front]; //出队
        if(p->lchild) Q[++rear]=p->lchild; //左孩子入队
        if(p->rchild) Q[++rear]=p->rchild; //右孩子入队
        if(front==last){ //是否为当前层的最后一个结点
            level++;
            last = rear;
        }
    }
    return level;
}
```

【例题】已知二叉树T，求出其结点个数最多的一层的深度。先写出结点数据结构定义(值域为int类型)，用c或c++实现该算法。

```
int widthOfBinaryTree(TreeNode* root, int n) {
    TreeNode *queue=(TreeNode*)malloc(sizeof(TreeNode)*n);
    int front=-1, rear=-1, last=0;
    // maxc=0记录最大宽度，maxl记录最宽层的深度，count记录当前宽度，level记录当前深度
    int maxc=0, max1=0, count=0, level=0;
    TreeNode *p;
    queue[++rear]=root;
    while(front<rear) {
        p=queue[++front];
        count++;
        if(p->left) queue[++rear]=p->left;
        if(p->right) queue[++rear]=p->right;
        if(front==last) {   // 遍历到当前层最后一个结点
            level++;
            if(count>maxc) {    // 更新宽度
                maxc=count;
                max1=level;
            }
            count=0;    // 宽度重置
            last=rear;  // 更新标志位
        }
    }
    free(queue);
    return max1;
}
```

层序遍历模板的代码都可以用来求什么类型的题?

1. 层序遍历，逆层序遍历；
2. 求二叉树的宽度，求最宽的一层位于哪里；
3. 求高度，深度；
4. 求分支节点的个数，判断是否为完全二叉树，满二叉树；
5. 求任何带有层序特点的题；

【例题】现有一棵二叉树，如果你站在这棵树的左侧向右看这棵树，那么你只能看到每一层的第一个结点，从左侧看到的所有结点，我们称之为二叉树的左视图，要求你设计一算法求出二叉树（以root为根结点）的右视图。

首先写出二叉树的数据结构定义（值域为int类型），其次用c或c++实现算法，算法要求返回int数组，其中包含了右视图下按从上到下顺序看到的每一个结点的值。

```c
// Definition for a binary tree node.
typedef struct {
    int val;
    struct TreeNode *left;
    struct TreeNode *right;
}TreeNode;


int* rightSideView(TreeNode* root, int n) {
    TreeNode* queue[n];
    int front=-1, rear=-1;  // 队列初始化
    int last=0, index=0;    // 标志位
    TreeNode *p;
    queue[++rear]=root;
    int *res=malloc(sizeof(int)*n);
    while(front<rear) {
        p=queue[++front];
        if(p->left) queue[++rear]=p->left;      // 左孩子入队
        if(p->right) queue[++rear]=p->right;    // 右孩子入队
        if(front==last) {   // 当前层最右结点
            res[index++]=p->val;
            last=rear;  // 更新标志位
        }
    }
    return res;
}
```