

二叉树算法

//前序遍历

```
void preorder(bitree *t){
    bitree *temp;
    temp = t;
    while(temp || !empty()){
        while(temp){
            printf("%4d",temp->data);
            push(temp);
            temp = temp->lchild;
        }
        if(s.top!=0){
            temp = pop();
            temp = temp->rchild;
        }
    }
    printf("\n");
}
```

//中序遍历

```
void inorder(bitree *t){
    bitree *temp = t;
    while(temp || !empty()){
        if(temp){
            push(temp);
            temp = temp->lchild;
        }else{
            temp = pop();
            printf("%4d",temp->data);
            temp = temp->rchild;
        }
    }
    printf("\n");
}
```

//后序遍历

```
void postorder01(bitree *t){
    bitree *temp = t;
    bitree *r=NULL;
    while(temp || !empty()){
        if(temp){
            push(temp);
            temp = temp->lchild;
        }else{
            temp = getTop();
            //printf("栈顶元素:%4d\n",temp->data);
            if(temp->rchild && temp->rchild!=r){
                temp = temp->rchild;
                push(temp);
                temp = temp->lchild;
            }else{
                temp = pop();
                printf("%4d",temp->data);
                r = temp;
                temp = NULL;
            }
        }
    }
    printf("\n");
}
```

```
void postorder02(bitree *t) {
    bitree *temp=t;
    while(temp || !empty()){
        while(temp) {
            temp->count=1;
            push(temp);
            temp = temp->lchild;
        }
        if(!empty()){
            temp = pop();
            if(temp->count==1) {
                temp->count++;
                push(temp);
                temp = temp->rchild;
            }else if(temp->count==2) {
                printf("%4d",temp->data);
                temp=NULL;
            }
        }
    }
    printf("\n");
}
```

```
/**
 * 二叉树的前序遍历非递归 (统一写法)
 */
class Solution{
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        if(root != NULL) st.push(root);
        while( !st.empty() ){
            TreeNode *node = st.top();
            if(node != NULL){
                st.pop();
                if(node->right) st.push(node->right); //右
                if(node->left) st.push(node->left); //左
                st.push(node); //中
                st.push(NULL);
            }else{
                st.pop();
                node = st.top();
                st.pop();
                result.push_back(node->val);
            }
        }
        return result;
    }
};
```

```
/* 普通迭代前序遍历 */  
class Solution {  
public:  
    vector<int> preorderTraversal(TreeNode* root) {  
        stack<TreeNode*> st;  
        vector<int> result;  
        st.push(root);  
        while (!st.empty()) {  
            TreeNode* node = st.top();  
            st.pop();  
            if (node != NULL) result.push_back(node->val);  
            else continue;  
            st.push(node->right);  
            st.push(node->left);  
        }  
        return result;  
    }  
};
```

```
/**
 *  二叉树的中序遍历非递归 (统一写法)
 */
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        if(root != NULL) st.push(root);
        while( !st.empty() ){
            TreeNode *node = st.top();
            if(node != NULL){
                st.pop();
                if(node->right) st.push(node->right);    // 右
                st.push(node);                          // 中
                st.push(NULL);
                if(node->left) st.push(node->left);      // 左
            }else{
                st.pop();
                node = st.top();
                st.pop();
                result.push_back(node->val);
            }
        }
        return result;
    }
};
```

```
/* 普通中序遍历 */  
class Solution {  
public:  
    vector<int> inorderTraversal(TreeNode* root) {  
        vector<int> res;  
        stack<TreeNode*> stk;  
        while (root != nullptr || !stk.empty()) {  
            while (root != nullptr) {  
                stk.push(root);  
                root = root->left;  
            }  
            root = stk.top();  
            stk.pop();  
            res.push_back(root->val);  
            root = root->right;  
        }  
        return res;  
    }  
};
```

```
/**
 * 二叉树的后序遍历非递归 (统一写法)
 */
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result;
        stack<TreeNode*> st;
        if(root != NULL) st.push(root);
        while(!st.empty()){
            TreeNode *node = st.top();
            if(node != NULL){
                st.pop();
                st.push(node);           // 中
                st.push(NULL);
                if(node->right) st.push(node->right); // 右
                if(node->left) st.push(node->left);   // 左
            }else{
                st.pop();
                node = st.top();
                st.pop();
                result.push_back(node->val);
            }
        }
        return result;
    }
};
```



```
/* 通过修改前序, 变为后序 */
```

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        stack<TreeNode*> st;
        vector<int> result;
        st.push(root);
        while (!st.empty()) {
            TreeNode* node = st.top();
            st.pop();
            if (node != NULL) result.push_back(node->val);
            else continue;
            st.push(node->left); // 相对于前序遍历, 这更改一下入栈顺序
            st.push(node->right);
        }
        reverse(result.begin(), result.end()); // 将结果反转之后就是左右中的顺序了
        return result;
    }
};
```

```
/**
```

```
* 二叉树的层序遍历非递归
```

```
*/
```

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        queue<TreeNode*> que;
        if (root != NULL) que.push(root);
        while (!que.empty()) {
            int size = que.size();
            vector<int> vec;
            for (int i=0; i<size; i++) {
                TreeNode *node = que.front();
                que.pop();
                vec.push_back(node->val);
                if (node->left) que.push(node->left);
                if (node->right) que.push(node->right);
            }
            result.push_back(vec);
        }
        return result;
    }
};
```

```

/**
 *   LeetCode-Solution
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector <vector <int>> ret;
        if (!root) return ret;

        queue <TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int currentLevelSize = q.size();
            ret.push_back(vector <int> ());
            for (int i = 1; i <= currentLevelSize; ++i) {
                auto node = q.front(); q.pop();
                ret.back().push_back(node->val);
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
        }
        return ret;
    }
};

/**
 *   PTA
 */
void LevelorderTraversal( BinTree BT ){
    queue<BinTree*> que;
    if(!BT) return;
    que.push(BT);
    while(!que.empty()){
        BinTree *node = que.front();
        visit(node->val);
        if(node->left) que.push(node->left);
        if(node->right) que.push(node->right);
    }
}

```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 * 递归二叉树的深度
 */
class Solution {
public:
    int max(int a, int b){
        return a>b ? a : b;
    }
    int maxDepth(TreeNode* root) {
        if(!root) return 0;
        int left_depth = maxDepth(root->left);
        int right_depth = maxDepth(root->right);
        return max(left_depth, right_depth)+1;
    }
};

/**
 * 给定一个二叉树，检查它是否是镜像对称的。
 * 例如，二叉树 [1,2,2,3,4,4,3] 是对称的。
 */
class Solution {
public:
    bool check(TreeNode *p, TreeNode *q){
        if(!p && !q) return true;
        if(!p || !q) return false;
        return check(p->left, q->right) && check(p->right, q->left) && p->val == q->val;
    }
    bool isSymmetric(TreeNode* root) {
        return check(root, root);
    }
};
```

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 * 给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。
 * 说明：叶子节点是指没有子节点的节点。
 * 示例：
 * 给定如下二叉树，以及目标和 sum = 22
 */
class Solution {
public:
    bool hasPathSum(TreeNode* root, int sum) {
        queue<TreeNode*> node_queue;
        queue<int> val_queue;
        if(root){
            node_queue.push(root);
            val_queue.push(root->val);
        }
        while(!node_queue.empty() && !val_queue.empty()){
            TreeNode *node = node_queue.front(); node_queue.pop();
            int val = val_queue.front(); val_queue.pop();
            if(!node->left && !node->right) {
                if( val==sum )
                    return true;
                continue;
            }
            if(node->left) {
                node_queue.push(node->left);
                val_queue.push(node->left->val + val);
            }
            if(node->right) {
                node_queue.push(node->right);
                val_queue.push(node->right->val + val);
            }
        }
        return false;
    }
};
```

```
/**
 * 《从中序与后序遍历序列构造二叉树》
 * 根据一棵树的中序遍历与后序遍历构造二叉树。
 * 注意：
 * 你可以假设树中没有重复的元素。
 * 例如，给出：中序遍历 inorder = [9,3,15,20,7]，后序遍历 postorder = [9,15,7,20,3]
 * 返回如下的二叉树：
```

```

      3
     / \
    9  20
   /  \
  15   7
```

```
*/
class Solution {
public:
    // 找到Inorder中根节点的位置
    int index;
    int getInorderIndex(vector<int> &inorder, int val){
        for(int i=0;i<inorder.size();i++) {
            if(val==inorder[i]) return i;
        }
        return 0;
    }

    TreeNode* create(vector<int>& inorder, vector<int>& postorder, int left, int right){
        if( left>right ) return nullptr;
        int val=postorder[index];
        TreeNode* root=new TreeNode(val);
        index--;
        int inIndex = getInorderIndex(inorder, val);
        root->right=create(inorder,postorder, inIndex+1, right);    // 后序(左右根)顺序表 index-- 先指向右

        root->left=create(inorder, postorder, left, inIndex-1);
        return root;
    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        if (!postorder.size()) return nullptr;
        // get root index
        index=postorder.size()-1;
        TreeNode* root=create(inorder, postorder, 0, inorder.size()-1);
        return root;
    }
}
```

};

/**

根据一棵树的前序遍历与中序遍历构造二叉树。

注意：

你可以假设树中没有重复的元素。

例如，给出 前序遍历 preorder = [3,9,20,15,7]，中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树：

```

    3
   / \
  9  20
   /  \
  15   7
*/

```

```
class Solution{
```

```
public:
```

```
    int index;
```

```
    int find(vector<int> inorder, int val){
```

```
        for(int i=0; i<inorder.size(); i++){
```

```
            if( inorder[i]==val ){
```

```
                return i;
```

```
            }
```

```
        return -1;
```

```
    }
```

```
}
```

```
TreeNode *create(vector<int> preorder, vector<int> inorder, int left, int right){
```

```
    if(left > right) return nullptr;
```

```
    int val = preorder[index];
```

```
    TreeNode *root = new TreeNode(val);
```

```
    index++;
```

```
    int rootIndexofInorder = find(inorder, val);
```

```
    root->left = create(preorder, inorder, 0, rootIndexofInorder-1); //前序(根左右)遍历序列, index++先
```

```
    root->right = create(preorder, inorder, rootIndexofInorder+1, right);
```

```
    return root;
```

```
}
```

```
TreeNode *buildTree(vector<int>& preorder, vector<int>& inorder){
```

```
    index = 0;
```

```
    return create(preorder, inorder, 0, inorder.size()-1);
```

```
}
```

};

```
//顺序存储结构

/**
 * i的左孩子 ---2i
 * i的右孩子 ---2i+1
 * i的父结点 ---i/2(向下取整)
 * i是否是叶子结点/分支结点 --- i 是否大于 ( n/2(向下取整) )
 * i是否有左孩子 ---2i <= n
 * i是否有右孩子 ---2i+1 <= n
 */

char comm_Ancessor(Tree T,int i,int j){
    if(T.nodes[i].data!='#' && T.nodes[j].data != '#'){
        while(i!=j){
            if(i>j)
                i = i/2;
            else
                j = j/2;
        }
        return T.nodes[i].data;
    }
}
```

```
//判断是否是完全二叉树

/**
    层序遍历 所有节点入队，包括空节点
    空树 return true
    if node == nullptr
        check 后面节点是否为空
        if 后面节点为空
            return false
        return true
*/

class Solution{
public:
    bool check(TreeNode *root){
        queue<TreeNode*> q;
        if(T == nullptr){
            return true;
        }
        q.push(root);
        while(!q.empty()){
            TreeNode *node = q.front();
            q.pop();
            if(node){
                q.push(node->left);
                q.push(node->right);
            }else{
                while(!q.empty()){
                    node = q.front();
                    q.pop();
                    if(p){
                        return false;
                    }
                }
            }
        }
        return true;
    }
};
```



```
// 双分支节点个数

/**
    model:
    f(b)=0                                // 若b=null
    f(b)=f(b->left)+f(b->right)+1        // 若b是双分支结点
    f(b)=f(b->left)+f(b->right)          // 若b是叶结点或单分支节点
*/

int DsonNodes(BinTree T){
    if(T==NULL)
        return 0;
    else if(T->left!=NULL &&T->right!=NULL) return DsonNodes(T->left)+DsonNodes(T->right)+1;
    else return DsonNodes(T->left)+DsonNodes(T->right);
}

/* 交换左右子树 */
/* 后序遍历思想：先交换b的左子树的左右子树，再交换b的右子树的左右子树，最后交换b的左右子树 */
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(!root) return nullptr;
        TreeNode *left = invertTree(root->left);
        TreeNode *right = invertTree(root->right);

        root->left = right;
        root->right = left;
        return root;
    }
};
```

```
/* 前序遍历第k个节点的值 */  
class Solution{  
public:  
    int i=1;  
    char get_N(TreeNode *root, int k){  
        char ch;  
        if(root==NULL)  
            return '#';  
        if(root==k)  
            return root->data;  
        i++;  
        ch = get_N(root->left,k);  
        if(ch!='#')  
            return ch;  
        ch=get_N(root->right,k);  
        if(ch!='#')  
            return ch;  
    }  
};
```

```
// 最近公共祖先
```

```
/**
```

```
 * 具体思路:
```

- (1) 如果当前结点 `root` 等于 `NULL`, 则直接返回 `NULL`
- (2) 如果 `root` 等于 `p` 或者 `q`, 那这棵树一定返回 `p` 或者 `q`
- (3) 然后递归左右子树, 因为是递归, 使用函数后可认为左右子树已经算出结果, 用 `left` 和 `right` 表示
- (4) 此时若`left`为空, 那最终结果只要看 `right`; 若 `right` 为空, 那最终结果只要看 `left`
- (5) 如果 `left` 和 `right` 都非空, 因为只给了 `p` 和 `q` 两个结点, 都非空, 说明一边一个, 因此 `root` 是他们的最近公共祖先
- (6) 如果 `left` 和 `right` 都为空, 则返回空 (其实已经包含在前面的情况中了)

时间复杂度是 $O(n)$: 每个结点最多遍历一次或用主定理, 空间复杂度是 $O(n)$: 需要系统栈空间

```
*/  
  
class Solution {  
public:  
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {  
        if(root==nullptr) return nullptr;  
        if(root==p || root==q) return root;  
  
        TreeNode *left = lowestCommonAncestor(root->left, p, q);  
        TreeNode *right = lowestCommonAncestor(root->right, p, q);  
  
        if(left==nullptr) return right;  
        if(right==nullptr) return left;  
  
        if(right && left) return root;  
  
        return nullptr;  
    }  
};
```

```
// 删除以bt为根的子树
void DeleteXTree(BinTree *bt) {
    if(bt) {
        DeleteXTree(bt->left);
        DeleteXTree(bt->right);
        free(bt);
    }
}

// 在二叉树上查找所有以x为元素的结点，并删除以其为根的子树
void Search(BinTree *bt, ElementType x) {
    BinTree *p;
    Queue q;
    if(bt) {
        if(bt->data==x) {
            DeleteXTree(bt);
            exit(0);
        }
        Init_Queue(&q);
        EnQueue(&q, bt);
        while(!isEmpty(&q)) {
            p=DeQueue(&q);
            if(p->left) {
                if(p->left->data==x) {
                    DeleteXTree(p->left);
                    p->left=NULL;
                }else
                {
                    EnQueue(&q, p->left);
                }
            }
            if(p->right) {
                if(p->right->data==x) {
                    DeleteXTree(p->right);
                    p->right=NULL;
                }else
                {
                    EnQueue(&q, p->right);
                }
            }
        }
    }
}
```

/* 查找结点的所有祖先 */

```
void Search(BinTree *bt, ElemType x) {
    stack s[M];
    int top = 0;
    while(bt || top > 0) {
        while(bt && bt->data != x) {
            s[++top].t = bt;
            s[top].tag = 0;
            bt = bt->left;
        }
        if(bt && bt->data == x) {
            printf("所查结点的所有祖先结点的值为:");
            for(int i = 1; i <= top; i++)
                printf("%4c", s[i].t->data);
            exit(1);
        }
        while(top != 0 && s[top].tag == 1)
            top--;
        if(top != 0) {
            s[top].tag = 1;
            bt = s[top].t->right;
        }
    }
} //结果 1 5
```

/* 递归所有祖先 */

```
int PrintAncestors(BinTree *root, ElemType x)
{
    if (!root) return 0;
    if (root->data == x) return 1;
    //如果子树中可以找到匹配值 那么此节点肯定是祖先结点
    if (PrintAncestors(root->left, x) || PrintAncestors(root->right, x))
    {
        printf("%4c", root->data);
        return 1;
    }
    return 0;
} //打印祖先 结果: 5 1
```

```
/* 二叉树的最大宽度 */
int widthOfBinaryTree(BinTree* root){
    if (root == NULL) return 0;
    int res = 0;
    BinTree *queue[MAXSIZE];
    int front = 0;
    int rear = 0;

    unsigned long long queue_index[MAXSIZE]; // 需要用 unsigned long long

    rear = (rear + 1) % MAXSIZE;
    queue[rear] = root;
    queue_index[rear] = 1;

    while (front != rear) {
        res = max(res, queue_index[rear] - queue_index[(front + 1) % MAXSIZE] + 1);
        int len = (rear - front) % MAXSIZE;
        BinTree *p;
        unsigned long long tmp_index;

        for (int i = 0; i < len; i++) {
            front = (front + 1) % MAXSIZE;
            p = queue[front];
            tmp_index = queue_index[front];

            if (p->left) {
                rear = (rear + 1) % MAXSIZE;
                queue[rear] = p->left;
                queue_index[rear] = tmp_index * 2;
            }
            if (p->right) {
                rear = (rear + 1) % MAXSIZE;
                queue[rear] = p->right;
                queue_index[rear] = tmp_index * 2 + 1;
            }
        }
    }
    return res;
}
```

```
/* 满二叉树 前序序列转换为后序序列 */
```

```
void preTopost(char pre[],int l1,int h1,char post[],int l2,int h2){  
    int half;  
    if(l1<=h1){  
        post[h2]=pre[l1];  
        half=(h1-l1)/2;  
        preTopost(pre,l1+1,l1+half,post,l2,l2+half-1);  
        preTopost(pre,l1+half+1,h1,post,l2+half,h2-1);  
    }  
}
```

```
/* 将叶子结点从左到右连成一个单链表, 表头指向head */
```

```
BinTree *head=NULL;
```

```
BinTree *pre=NULL;
```

```
BinTree *leafNodeToList(BinTree *root){  
    if(root){  
        leafNodeToList(root->left);  
        if(root->left==NULL && root->right==NULL){  
            if(pre==NULL){  
                head=root;  
                pre=root;  
            }else{  
                pre->right=root;  
                pre=root;  
            }  
        }  
        leafNodeToList(root->right);  
        pre->right=NULL;  
    }  
    return head;  
}
```

```
/* 判断两棵树是否相似 */
```

```
int similar_tree(BinTree *t1,BinTree *t2){  
    if(t1==NULL && t2==NULL) return 1;  
    else if(t1==NULL || t2==NULL) return 0;  
    else{  
        return similar_tree(t1->left,t2->left) && similar_tree(t1->right,t2->right);  
    }  
}
```

```
/* 中序线索二叉树里查找指定节点在后序的前驱结点 */
```

```
BinTree *InPostPre(BinTree *t, BinTree *p) {  
    BinTree *q;  
    if(p->rtag==0) q=p->right;  
    else if(p->ltag==0) q=p->left;  
    else if(p->left==NULL)  
        q=NULL;  
    else{  
        while(p->ltag==1&&p->left!=NULL) {  
            p=p->left;  
        }  
        if(p->ltag==0)  
            q=p->left;  
        else  
            q=NULL;  
    }  
    return q;  
}
```

```
/* 将给定的表示式树（二叉树）转换为等价的中缀表达式（添加括号反映操作符计算次序） */
```

```
void BTreeToE(BTree *root) {  
    BTreeToExp(root, 1); //根的高度为1  
}  
  
void BTreeToExp(BTree *root, int deep) {  
    if(root==nullptr) return; // 空节点返回  
    else if(root->left==nullptr && root->right==null) // 若为叶子结点  
        printf(root->data); // 输出操作数，不加括号  
    else{  
        if(deep > 1) printf("("); // 若有子表达式则加 1层括号  
        BTreeToExp(root->left, deep + 1);  
        printf(root->data); // 输出操作符  
        BTreeToExp(root->right, deep + 1);  
        if(deep > 1) printf(")"); // 若有子表达式则加 1层括号  
    }  
}
```