数据结构学习

- 结点：数据元素+若干指向子树的分支
    - 叶子结点:度为零的结点
    - 分支结点:度大于零的结点

      孩子结点、双亲结点、兄弟结点、堂兄弟 祖先结点、子孙结点
    - 结点的层次：假设根结点的层次为1，第l层的结点的子树根结点的层次为l+1
- (从根到结点的)路径：由从根到该结点所经分支和结点构成
- 结点的度:分支的个数

- 树的度：树中所有结点的度的最大值
- 树的深度：树中叶子结点所在的最大层次--[结点的层次]

  <u>度和深度不一样</u>

- 森林：m（m≥0）棵<u>互不相交</u>的树的集合

树与二叉树

定义和初始化

```c
/* 二叉树节点结构体定义 */
typedef struct TreeNode {
    int val;                 // 节点值
    int height;              // 节点高度
    struct TreeNode *left;   // 左子节点指针
    struct TreeNode *right;  // 右子节点指针
} TreeNode;

/* 二叉树构造函数 */
TreeNode *newTreeNode(int val) {//定义一个二叉树根节点创建函数，参数是根节点的值，返回一个结构指针-->指向一个结构体
    TreeNode *node;

    node = (TreeNode *)malloc(sizeof(TreeNode));
    node->val = val;
    node->height = 0;
    node->left = NULL;
    node->right = NULL;

    return node;//函数返回一个结构，注意要指针定义才可返回结构指针
}
```

二叉树的遍历

层序遍历

```
/* 层序遍历 */
int *levelOrder(TreeNode *root, int *size) {//参数是根节点和二叉树的大小？
    /* 辅助队列 */
    int front, rear;
    int index, *arr;
    TreeNode *node;
    TreeNode **queue;

    /* 辅助队列----初始化一个列表，用于保存遍历序列，所以queue是二重指针，是个头指针可变的数组，数组里存放节点*/
    queue = (TreeNode **)malloc(sizeof(TreeNode *) * MAX_SIZE);
    // 队列指针
    front = 0, rear = 0;
    // 加入根节点，queue存的是树
    queue[rear++] = root;


    /* 辅助数组----保存节点的值 */
    arr = (int *)malloc(sizeof(int) * MAX_SIZE);
    // 数组指针
    index = 0;
    while (front < rear) {
        // 队列出队，这里front++增加的范围是此结构存储结构的长度
        node = queue[front++];
        // 保存节点值
        arr[index++] = node->val;
        if (node->left != NULL) {   // 左子节点入队
            queue[rear++] = node->left;
        }
        if (node->right != NULL) {  // 右子节点入队
            queue[rear++] = node->right;
        }
        //if 确定rear的上限，就是树的深度
    }
    // 更新数组长度的值
    *size = index;
    arr = realloc(arr, sizeof(int) * (*size));

    // 释放辅助数组空间
    free(queue);
    return arr;

}
```

```
//补充：a[i++]和a[++i]的区别
a[i++]==a[i];i++;
a[++i]==++i;a[i];
```

前中后序遍历

> 可以发现，这三个遍历都是递归的方式

```c
/* 前序遍历 */
void preOrder(TreeNode *root, int *size) {
    if (root == NULL)
        return;
    // 访问优先级：根节点 -> 左子树 -> 右子树
    arr[(*size)++] = root->val;//访问根节点，取值
    preOrder(root->left, size);//访问左子树，直至访问结束
    preOrder(root->right, size);
    //-------左子树访问结束后，最后一次函数访问右子树；倒数第二个函数访问上一个节点的右子树......-------//
}

/* 中序遍历 */
void inOrder(TreeNode *root, int *size) {
    if (root == NULL)
        return;
    // 访问优先级：左子树 -> 根节点 -> 右子树
    inOrder(root->left, size);
    arr[(*size)++] = root->val;
    inOrder(root->right, size);
}

/* 后序遍历 */
void postOrder(TreeNode *root, int *size) {
    if (root == NULL)
        return;
    // 访问优先级：左子树 -> 右子树 -> 根节点
    postOrder(root->left, size);
    postOrder(root->right, size);
    arr[(*size)++] = root->val;
}
```

线索二叉树

> 二叉树的遍历可以理解成对非线性结构进行线性化操作。但是当二叉树以二叉链表作为存储结构时，前驱和后继的寻找显得比较麻烦

```
这是实验四的验收
{1} {0} of {1} course is {data} {5}
 subject the The course structure


Enter the format string: {1}{0} of {1} course is {3} {1001}
Enter the substitute strings('!!' indicates end): subject
words[0]: subject
the
words[1]: the
course
words[2]: course
{data}
```

```
words[3]: {data}
{structure}
words[4]: {structure}
!!
words[5]: !!
Input completed
the subject of the course is {data}
The index is too big!
```

实验--汉诺塔问题

> 汉诺塔问题中，3 个圆盘至少需要移动 7 次，移动 n 的圆盘至少需要操作 2^n-1 次。

在汉诺塔问题中，当圆盘个数不大于 3 时，多数人都可以轻松想到移动方案，随着圆盘数量的增多，汉诺塔问题会越来越难。也就是说，圆盘的个数直接决定了汉诺塔问题的难度，解决这样的问题可以尝试用分治算法，将移动多个圆盘的问题分解成多个移动少量圆盘的小问题，这些小问题很容易解决，从而可以找到整个问题的解决方案。

分治算法

PPT内容总结

(Gemini & deepseek)

第六章：树和二叉树

本章详细介绍了一种非常重要且应用广泛的非线性数据结构——树和其最常用的特例——二叉树。

6.1 树的定义和基本术语

树的定义

树（Tree）是一种由 n（$n \geq 0$）个有限节点组成的数据结构，具有以下特点：

- 当 n=0 时，称为**空树**。
- 当 n>1 时，有且仅有一个特定的节点称为**根**（root）。
- 其余节点可分为 m（$m \geq 0$）个互不相交的有限集 $T_1, T_2, \ldots, T_m$，其中每个集合本身又是一棵树，被称为根的**子树**（Subtree）。

核心特征:

1. 有确定的根节点。
2. 节点之间是**有向关系**，从父节点指向子节点。
3. **有序树**：子树之间存在确定的次序关系。
4. **无序树**：子树之间不存在确定的次序关系。

树与线性结构的对比

特性	线性结构	树型结构
起始	有唯一的“第一个”元素（无前驱）	有唯一的“根”节点（无前驱）
终端	有唯一的“最后一个”元素（无后继）	有多个“叶子”节点（无后继）

特性	线性结构	树型结构
中间	每个元素有唯一前驱和唯一后继	每个节点有唯一前驱和多个后继

基本术语

- **节点（Node）**：数据结构中的基本单元，包含数据元素及指向子树的分支。
- **节点的度（Degree）**：节点拥有的子树（或分支）的数量。
- **树的度**: 树中所有节点度的最大值。
    - **根节点（Root Node）**： 树的第一个节点。
    - **叶子节点（Leaf Node）**：度为 0 的节点，也称为终端节点。
    - **分支节点（Branch Node）**：度大于 0 的节点，也称为非终端节点。

- **孩子（Child）** 和 **双亲（Parent）**：一个节点的子树的根是该节点的孩子，该节点是其孩子的双亲。
- **兄弟（Sibling）**：具有相同双亲的节点互为兄弟。
- **祖先（Ancestor）** 和 **子孙（Descendant）**：从根到某节点的路径上的所有节点都是其祖先。反之，某节点是其子树中所有节点的祖先。

- **节点的层次（Level）**：根的层次为 1，其余节点的层次为其双亲层次加 1。
- **树的深度（Depth）** 或 **高度（Height）**：树中节点的最大层次。
- **森林（Forest）**：m（$m \geq 0$）棵互不相交的树的集合。任何一棵非空树都可以看作是一个二元组 `Tree = (root, F)`，其中 `F` 是根 `root` 的子树森林。

树的基本操作

- **查找类**:
    - `Root(T)`: 求树的根节点。
    - `Value(T, cur_e)`: 求当前节点的值。
    - `Parent(T, cur_e)`: 求当前节点的双亲。
    - `LeftChild(T, cur_e)`: 求当前节点的最左孩子。
    - `RightSibling(T, cur_e)`: 求当前节点的右兄弟。
- **状态类**:
    - `TreeEmpty(T)`: 判断树是否为空。
    - `TreeDepth(T)`: 求树的深度。
- **遍历**:
    - `TraverseTree(T, Visit())`: 按某种方式遍历树的所有节点。
- **构造/修改类**:
    - `InitTree(&T)`: 初始化树。
    - `CreateTree(&T, definition)`: 根据定义构造树。
    - `Assign(T, cur_e, value)`: 给节点赋值。
    - `InsertChild(&T, &p, i, c)`: 将以 `c` 为根的树插入为节点 `p` 的第 `i` 棵子树。
- **销毁类**:
    - `ClearTree(&T)`: 清空树。

- `DestroyTree(&T)`: 销毁树。
- `DeleteChild(&T, &p, i)`: 删除节点 `p` 的第 `i` 棵子树。

6.2 二叉树

二叉树的定义

二叉树（Binary Tree）是 n（$n \geq 0$）个节点的有限集合，它或为空树，或由一个根节点加上两棵分别称为**左子树**（Left Subtree）和**右子树**（Right Subtree）的、互不相交的二叉树组成。

与树的区别:

1. 二叉树的度**最多为 2**，树的度没有限制。
2. 二叉树的子树有**左右之分**，次序不能颠倒，是**有序树**。

二叉树的五种基本形态

1. 空树。
2. 只有根节点。
3. 只有根节点和左子树。
4. 只有根节点和右子树。
5. 根节点、左子树和右子树都存在。

二叉树的重要特性

- **性质1**: 在二叉树的第 i 层上至多有 2^{i-1} 个节点 ($i \geq 1$)。
  - *证明思路*: 使用数学归纳法。$i = 1$ 时成立。假设第 j 层 ($j < i$) 成立，则第 i 层的节点数最多是第 $i - 1$ 层节点数的 2 倍，即 $2 \times 2^{i-2} = 2^{i-1}$。
- **性质2**: 深度为 k 的二叉树上至多含 $2^k - 1$ 个节点 ($k \geq 1$)。
  - *证明思路*: 将各层最大节点数相加: $\sum_{i=1}^{k} 2^{i-1} = 2^0 + 2^1 + ... + 2^{k-1} = 2^k - 1$。
- **性质3**: 对任何一棵二叉树，若它含有 n_0 个叶子节点、n_2 个度为 2 的节点，则必存在关系式: $n_0 = n_2 + 1$。
  - *证明思路*: 设树的总节点数为 $n = n_0 + n_1 + n_2$（n_1 为度为 1 的节点数）。树的总分支数（边数）$b = n_1 + 2n_2$。同时，除根节点外，每个节点都有一个进入的分支，所以 $b = n - 1$。联立两式可得 $n_1 + 2n_2 = n_0 + n_1 + n_2 - 1$，化简即得 $n_0 = n_2 + 1$。
- **性质4**: 具有 n 个节点的**完全二叉树**的深度为 $\lfloor \log_2 n \rfloor + 1$。
  - *证明思路*: 设深度为 k，根据性质2和完全二叉树的定义，有 $2^{k-1} - 1 < n \leq 2^k - 1$，即 $2^{k-1} \leq n < 2^k$。两边取对数得 $k - 1 \leq \log_2 n < k$。因为 k 是整数，所以 $k = \lfloor \log_2 n \rfloor + 1$。
- **性质5**: 对含 n 个节点的**完全二叉树**从上到下、从左到右编号（从 1 开始），对任意节点 i:
  1. 若 $i > 1$，其双亲节点为 $\lfloor i/2 \rfloor$。
  2. 若 $2i > n$，节点 i 无左孩子；否则其左孩子是 $2i$。
  3. 若 $2i + 1 > n$，节点 i 无右孩子；否则其右孩子是 $2i + 1$。

特殊的二叉树

- **满二叉树**: 深度为 k 且含有 $2^k - 1$ 个节点的二叉树。每一层的节点数都达到了最大值。
- **完全二叉树**: 深度为 k，有 n 个节点的二叉树，当且仅当其每一个节点都与深度为 k 的满二叉树中编号从 1 至 n 的节点一一对应。特点是叶子节点只可能出现在最下两层，且最下一层的叶子节点都集中在左边。

完全 ≠ 满

6.3 二叉树的存储结构

1. 顺序存储表示

将二叉树的节点按**层序**存放在一个一维数组中。这种结构适合表示**完全二叉树**，因为不会浪费空间。对于一般的二叉树，会造成大量空间浪费。

- **双亲表示法**: 数组中每个元素不仅存储数据，还存储其双亲节点在数组中的下标。

```c
#define MAX_TREE_SIZE 100

typedef char TElemType; // 假设元素类型为字符

typedef struct BPTNode { //结点的结构
    TElemType data;
    int parent;      // 指向双亲的下标
    char LRTag;      // 左右孩子标志
} BPTNode;

typedef struct {    //树的结构
    BPTNode nodes[MAX_TREE_SIZE];
    int num_node;    // 节点数目
    int root;        // 根节点的位置
} BPTree;
```

2. 链式存储表示

- **二叉链表**: 每个节点包含一个数据域和两个指针域，分别指向其左孩子和右孩子。这是最常用和最直观的表示方法。

```c
typedef char TElemType;

typedef struct BiTNode {
    TElemType data;
    struct BiTNode *lchild, *rchild; // 左右孩子指针
} BiTNode, *BiTree;
```

- **三叉链表**: 在二叉链表的基础上增加一个指向双亲节点的指针域。方便寻找双亲，但增加了空间开销。

```c
typedef char TElemType;

typedef struct TriTNode {
    TElemType data;
    struct TriTNode *lchild, *rchild; // 左右孩子指针
    struct TriTNode *parent;          // 双亲指针
} TriTNode, *TriTree;
```

6.4 二叉树的遍历

遍历（Traversal）是指按某种特定的搜索路径巡访树中的每个节点，使得每个节点均被访问一次，而且仅被访问一次。对于二叉树，主要有四种遍历方式：

1. 先序遍历 (Pre-order Traversal)

- **规则**: 若二叉树为空则空操作，否则：(1) 访问根节点；(2) 先序遍历左子树；(3) 先序遍历右子树 (DLR)。
- **递归实现**:

```
void PreOrderTraverse(BiTree T, void (*Visit)(TElemType e)) {
    if (T) {
        Visit(T->data);                   // 访问根节点
        PreOrderTraverse(T->lchild, Visit); // 递归遍历左子树
        PreOrderTraverse(T->rchild, Visit); // 递归遍历右子树
    }
}
```

2. 中序遍历 (In-order Traversal)

- **规则**: 若二叉树为空则空操作，否则：(1) 中序遍历左子树；(2) 访问根节点；(3) 中序遍历右子树 (LDR)。
- *重要特性*: 中序遍历**二叉排序树**可以得到一个有序的节点序列。
- **递归实现**:

```
void InOrderTraverse(BiTree T, void (*Visit)(TElemType e)) {
    if (T) {
        InOrderTraverse(T->lchild, Visit); // 递归遍历左子树
        Visit(T->data);                    // 访问根节点
        InOrderTraverse(T->rchild, Visit); // 递归遍历右子树
    }
}
```

- **非递归实现**:
  - **算法思想**: 递归的本质是栈，因此可用一个显式的栈来模拟递归过程。
    1. 初始化一个栈，指针 `p` 指向根节点。
    2. 当 `p` 不为空或栈不空时循环：
    3. 如果 `p` 不为空，则将 `p` 压入栈中，并让 `p` 移动到其左孩子 (`p = p->lchild`)。
    4. 如果 `p` 为空（说明左子树已走到尽头），则从栈中弹出一个节点，访问它，然后让 `p` 指向弹出节点的右孩子 (`p = p->rchild`)。
  - **算法实现**:

```
// 辅助函数：从节点T出发，一路向左，将路径上的节点压入栈S，返回最左下方的节点
BiTNode* GoFarLeft(BiTree T, Stack *S){
    if (!T) return NULL;
    while (T->lchild){
        Push(S, T); // 将非最左下的根节点压栈
        T = T->lchild;
    }
    return T; // 返回最左下的节点
```

```
}

// 中序遍历非递归算法
void Inorder_I(BiTree T, void (*visit)(TElemType e)){
   Stack S; // 辅助栈
   InitStack(&S);
   BiTree p = GoFarLeft(T, &S); // 首先找到最左下节点
   while(p){
      visit(p->data); // 访问该节点
      // 如果有右子树，则对右子树重复“一路向左”的操作
      if (p->rchild) {
         p = GoFarLeft(p->rchild, &S);
      }
      // 如果没有右子树，且栈不空，则弹出父节点访问
      else if (!StackEmpty(S)){
        Pop(&S, &p);
      }
      // 遍历结束
      else {
        p = NULL;
      }
    }
}
```

3. 后序遍历 (Post-order Traversal)

- **规则**: 若二叉树为空则空操作，否则：(1) 后序遍历左子树；(2) 后序遍历右子树；(3) 访问根节点 (LRD)。
- *应用*: 常用于计算表达式树的值、释放树的内存空间等。
- **递归实现**:

```
void PostOrderTraverse(BiTree T, void (*Visit)(TElemType e)) {
    if (T) {
        PostOrderTraverse(T->lchild, Visit); // 递归遍历左子树
        PostOrderTraverse(T->rchild, Visit); // 递归遍历右子树
        Visit(T->data);                      // 访问根节点
    }
}
```

4. 层序遍历 (Level-order Traversal)

- **规则**: 从上到下，从左到右，逐层访问节点。
- *实现*: 通常需要借助一个**队列**来实现。

5. 遍历算法的应用举例

要清楚递归的调用过程和实质。

遍历是二叉树很多操作的基础，以下是一些典型应用：

- **应用1：统计叶子结点个数**
  - **算法思想**：采用先序遍历（或其他遍历方式均可）来访问树中的每一个结点。在访问到某个结点时，判断它是否为叶子结点（即其左右孩子指针均为空）。如果是，则将一个通过地址传递的计数器加一。

- **算法实现**:

```c
// 统计以T为根的二叉树中叶子结点的个数
// count为计数器指针，用于返回结果
void CountLeaf(BiTree T, int* count) {
    if (T) {
        // 如果当前结点的左右孩子都为空，则为叶子结点
        if ((!T->lchild) && (!T->rchild)) {
            (*count)++; // 计数器加一
        }
        // 递归访问左子树
        CountLeaf(T->lchild, count);
        // 递归访问右子树
        CountLeaf(T->rchild, count);
    }
}
```

- **应用2：求二叉树的深度**
  - **算法思想**：此问题天然具有递归性，非常适合使用后序遍历的模式。一棵树的深度等于其左子树深度和右子树深度的较大者，再加1（根节点本身占一层）。空树的深度定义为0。
  - **算法实现**:

```c
// 计算以T为根的二叉树的深度
int Depth(BiTree T) {
    int depthval, depthLeft, depthRight;

    // 递归基：如果树为空，深度为0
    if (!T) {
        depthval = 0;
    } else {
        // 递归计算左子树的深度
        depthLeft = Depth(T->lchild);
        // 递归计算右子树的深度
        depthRight = Depth(T->rchild);
        // 树的深度 = 1 + max(左子树深度, 右子树深度)
        depthval = 1 + (depthLeft > depthRight ? depthLeft : depthRight);
    }
    return depthval;
}
```

- **应用3：复制二叉树**
  - **算法思想**：同样采用后序遍历的模式。要复制一棵树，首先要递归地复制其左子树和右子树，得到指向新左、右子树的指针。然后，为根节点申请新的内存空间，并把数据复制过去，最后将新根节点的左右孩子指针分别指向新复制好的左、右子树。
  - **算法实现**:

```c
// 复制一棵以T为根的二叉树，并返回指向新树根的指针
BiTree CopyTree(BiTree T) {
    BiTree newT = NULL;
    BiTree newlptr = NULL;
    BiTree newrptr = NULL;
```

```c

    // 递归基：如果原树为空，复制结果也为空
    if (!T) {
        return NULL;
    }

    // 递归复制左子树
    if (T->lchild) {
        newlptr = CopyTree(T->lchild);
    } else {
        newlptr = NULL;
    }

    // 递归复制右子树
    if (T->rchild) {
        newrptr = CopyTree(T->rchild);
    } else {
        newrptr = NULL;
    }

    // 创建新根节点，并连接复制好的左右子树
    newT = (BiTree)malloc(sizeof(BiTNode));
    if (!newT) exit(1); // 内存分配失败
    newT->data = T->data;
    newT->lchild = newlptr;
    newT->rchild = newrptr;

    return newT;
}
```

- **应用4：建立二叉树**
  - **根据带空指针的先序序列建立**: 如 `A(B(*,D(*,*)),(C(E*,*),*)` 、 `A(B(*,C(*,*)),D(*,*))` 形式（用*表示空树），可以直接通过递归建立。

```c
Status CreateBiTree(BiTree *T) {
    scanf(&ch);
    if (ch=='*') T = NULL;
    else {
      if (!(T = (BiTNode *)malloc(sizeof(BiTNode))))
        return ERROR;
      T->data = ch;              // 生成根结点
      CreateBiTree(T->lchild);   // 构造左子树
      CreateBiTree(T->rchild);   // 构造右子树
    }
    return OK;
} // CreateBiTree
```

  - **根据表达式建立**: 对于 `(a+b)*c-d/e` 这样的中缀表达式，可以通过两个栈（一个操作数栈，一个操作符栈）来构造表达式树，处理核心是运算符的优先级。
    - 先缀表达式：**操作数是叶子节点，运算符是分支节点。**

      算法的基本思想如下：

1. **读取字符**：顺序读取先缀表达式中的一个字符。
2. 判断类型：
    - 如果该字符是**操作数**（如'a', 'b'等），说明它是一个叶子结点。此时，创建一个新的叶子结点，并将该操作数存入其中，然后返回该结点。
    - 如果该字符是**操作符**（如'+', '-', '×', '/'等），说明它是一个分支结点（即子树的根）。此时，创建一个根结点，并将该操作符存入其中。
3. 递归构造子树：
    - 由于操作符后面紧跟着的是它的两个操作数（或子表达式），因此接着递归调用建树函数来构造**左子树**。
    - 左子树构造完成后，继续递归调用建树函数来构造**右子树**。
4. **返回根**：将创建的根结点返回。

```
/* 函数功能：根据全局指针p指向的先缀表达式字符串，递归地创建表达式树。
   参数 T: 指向要创建的树（或子树）根节点的指针的指针。
   返回值: 如果创建成功返回OK，否则返回ERROR。
*/
Status CreateExpTree_Pre(BiTree *T, char **p) {
    char ch = **p; // 读取当前字符
    (*p)++; // 指针后移，准备下次读取

    if (ch == '\0' || ch == ' ') { // 假设空格或字符串末尾为结束符
        *T = NULL;
        return OK;
    }

    // 判断字符类型
    if (ch >= 'a' && ch <= 'z') { // 假设操作数为小写字母
        // 1. 如果是操作数，则为叶子结点
        *T = (BiTNode *)malloc(sizeof(BiTNode));//子一级的递归调用去直接修改父一级结点的lchild或rchild指针域，从而将新建的结点挂接到正确的位置上，下同
        if (!(*T)) return ERROR; // 内存分配失败
        (*T)->data = ch;
        (*T)->lchild = NULL;
        (*T)->rchild = NULL;
    } else { // 2. 如果是操作符，则为分支结点
        *T = (BiTNode *)malloc(sizeof(BiTNode));
        if (!(*T)) return ERROR; // 内存分配失败
        (*T)->data = ch;

        // 3. 递归创建左子树
        CreateExpTree_Pre(&((*T)->lchild), p);

        // 4. 递归创建右子树
        CreateExpTree_Pre(&((*T)->rchild), p);
    }
    return OK;
}
```

- 中缀表达式 `(a+b)*c - d/e` 建树要复杂得多，因为操作符的执行顺序不只取决于其出现位置，还取决于**运算符的优先级**和**括号**。

  可以使用**两个栈**来解决这个问题：

- **操作符栈 (Operator Stack, `S`)**: 存放尚未处理的操作符和左括号。
- **指向结点的指针栈 (Pointer Stack, `PTR`)**: 存放已创建的、指向操作数或子树根节点的指针。

算法的核心思想是模拟人们计算表达式的过程：

1. 初始化: 在操作符栈底放入一个特殊符号（如'#'）作为哨兵，它的优先级最低 。
2. **遍历表达式**: 从左到右扫描中缀表达式字符串。
3. 处理操作数: 如果遇到操作数，则创建一个只包含该操作数的叶子结点，并将其指针压入 `PTR` 栈 。
4. **处理操作符**: 如果遇到操作符（包括 括号），则根据其类型和与 `S` 栈顶操作符的优先级关系进行处理 。
   - 遇到' (': 直接压入 `S` 栈 。
   - 遇到') ': 从 `S` 栈中弹出操作符，并从 `PTR` 栈中弹出两个指针来构建子树，直到遇到' ('为止。这个' ('被弹出并丢弃，表示括号内的表达式已处理完毕 。
   - 遇到其他操作符 (如 '+', '-', '×', '/'): 将当前操作符ch与S栈顶操作符c进行优先级比较 。
     - 若 `precede(c, ch)` 为真（即栈顶 `c` 的优先级**高于或等于**当前 `ch`），则说明栈顶的操作符 `c` 应该先被执行。因此，弹出 `c` 和 `PTR` 栈中的两个指针，构建子树，并将新子树的根指针压回 `PTR` 栈。重复此过程，直到栈顶操作符的优先级低于 `ch`。
     - 若 `precede(c, ch)` 为假，将操作符压入栈，即下面的操作
     - 当循环结束（即栈顶操作符优先级低于 `ch`）后，将当前操作符 `ch` 压入 `S` 栈。
   - **结束处理**: 表达式扫描完毕后，`S` 栈中可能还有剩余的操作符。依次将它们弹出，并与 `PTR` 栈中的指针构建子树，直到 `S` 栈中只剩下哨兵'#'。
   - **完成**: 此时，`PTR` 栈中应仅存一个指针，它就是整个表达式树的根节点。将其弹出即可 。

```
// 需要预先定义 precede(op1, op2) 函数判断优先级，此处略
// 以及两个栈：S (存char) 和 PTR (存BiTree)

void CreateExpTree_Infix(BiTree *T, char exp[]) {
    Stack_OP S; // 操作符栈
    Stack_PTR PTR; // 指针栈

    InitStack_OP(&S); Push_OP(&S, '#'); // 初始化操作符栈并放入哨兵
    InitStack_PTR(&PTR);

    char *p = exp;
    char ch = *p;

    // 循环直到表达式和操作符栈都处理完毕
    while (!(GetTop_OP(S) == '#' && ch == '#')) { // #是表达式结束符
        if (is_operand(ch)) {
            // 1. 如果是操作数，创建叶子结点并压入PTR栈
            BiTree node;
            CrtNode(&node, ch); // 创建叶子结点
            Push_PTR(&PTR, node); // 指针入栈
            p++; ch = *p; // 读取下一个字符
```

```
        } else { // 2. 如果是操作符
            switch (precede(GetTop_OP(S), ch)) {
                case '<': // 栈顶操作符优先级低，当前操作符入栈
                    Push_OP(&S, ch);
                    p++; ch = *p; // 读取下一个字符
                    break;
                case '=': // 优先级相等，通常是括号匹配
                    char temp_op;
                    Pop_OP(&S, &temp_op); // 脱去括号
                    p++; ch = *p; // 读取下一个字符
                    break;
                case '>': // 栈顶操作符优先级高，弹栈并建树
                    char op;
                    BiTree rc, lc, sub_tree;
                    Pop_OP(&S, &op); // 弹出高优先级操作符
                    Pop_PTR(&PTR, &rc); // 弹出右孩子
                    Pop_PTR(&PTR, &lc); // 弹出左孩子
                    CrtSubtree(&sub_tree, op, lc, rc); // 建子树
                    Push_PTR(&PTR, sub_tree); // **新子树的根指针入栈，这个很重要，他告诉我递归的实现
                    // 注意：此时不读取下一个字符，继续用当前ch与新栈顶比较
                    break;
            }
        }
    }
    // 循环结束后，PTR栈中唯一的元素就是树根
    Pop_PTR(&PTR, T);
}

// 辅助函数：创建叶子结点
void CrtNode(BiTree *T, char ch) {
   *T = (BiTNode*)malloc(sizeof(BiTNode));
   (*T)->data = ch;
   (*T)->lchild = (*T)->rchild = NULL;
}

// 辅助函数：创建子树
void CrtSubtree(BiTree *T, char op, BiTree lc, BiTree rc) {
   *T = (BiTNode*)malloc(sizeof(BiTNode));
   (*T)->data = op;
   (*T)->lchild = lc;
   (*T)->rchild = rc;
}
```

- **根据先序和中序序列建立**: 这是确定一棵二叉树的经典方法。
    1. **确定根**: 先序遍历的第一个元素永远是当前（子）树的根。
    2. **划分左右子树**: 在 [中序遍历序列] 中找到这个根，[根左边的所有元素] 构成 [左子树的中序序列]，[右边的所有元素] 构成 [右子树的中序序列]。 **（画图 or 想象一下树的中序遍历）**
    3. **确定子树范围**: 根据 [中序序列] 中 [左、右子树的节点数量]，可以在 [先序序列] 中确定对应 [左、右子树的范围]。 **（先序序列 是 根-左-右，根节点 连着的 左右节点的数量）**

4. **递归构造**: 对左、右子树的先序和中序序列递归地调用此方法，即可构造出整棵二叉树。

- **算法实现**（根据先序和中序序列）：

```c
// pre: 先序序列数组, ino: 中序序列数组
// ps, is: 当前处理的子序列在原数组中的起始下标
// n: 当前处理的子序列的长度
// T: 指向要建立的树根节点的指针的指针
void CrtBT(BiTree* T, char pre[], char ino[], int ps, int is, int n) {
    if (n == 0) {
        *T = NULL; // 子序列长度为0，是空树
        return;
    }

    // 1. 创建根节点
    *T = (BiTNode*)malloc(sizeof(BiTNode));
    (*T)->data = pre[ps]; // 先序序列的第一个元素是根

    // 2. 在中序序列中找到根的位置k，以划分左右子树
    int k = is;
    while(k < is + n && ino[k] != pre[ps]) {
        k++;
    }

    int left_len = k - is; // 左子树的长度
    int right_len = n - left_len - 1; // 右子树的长度

    // 3. 递归构造左子树
    // 左子树的先序序列从 pre[ps+1] 开始
    // 左子树的中序序列从 ino[is] 开始
    CrtBT(&((*T)->lchild), pre, ino, ps + 1, is, left_len);

    // 4. 递归构造右子树
    // 右子树的先序序列从 pre[ps + 1 + left_len] 开始
    // 右子树的中序序列从 ino[k + 1] 开始
    CrtBT(&((*T)->rchild), pre, ino, ps + 1 + left_len, k + 1, right_len);
}
```

6.5 线索二叉树

1. 定义与提出

在二叉链表中，有大量的空指针域 ($n+1$个)。为利用这些空指针域，**线索二叉树**（Threaded Binary Tree）被提出。它利用 [空的] 左/右孩子 [指针域]，存放指向该节点在某种遍历次序下的**前驱**（predecessor）和**后继**（successor）的指针。这种指针称为**线索**（Thread）。

为区分指针和线索，节点结构需增加两个标志位 `LTag` 和 `RTag`:

- `LTag=0` (Link): `lchild` 指向左孩子。
- `LTag=1` (Thread): `lchild` 指向**前驱**。
- `RTag=0` (Link): `rchild` 指向右孩子。
- `RTag=1` (Thread): `rchild` 指向**后继**。

```c
typedef enum { Link, Thread } PointerTag; // Link=0:指针, Thread=1:线索

typedef struct BiThrNode {
    TElemType data;
    struct BiThrNode *lchild, *rchild;
    PointerTag LTag, RTag;
} BiThrNode, *BiThrTree;
```

2. 如何建立线索链表

建立线索链表的过程（线索化）就是在遍历二叉树的同时修改空指针。以中序线索化为例：

- **算法思想**: 在对二叉树进行中序遍历时，附设一个指针 `pre`，始终指向刚刚访问过的节点。当访问当前节点 `p` 时，就可以建立 `pre` 与 `p` 之间的线索关系。
    - 检查 `p` 的左孩子是否为空，若为空，则建立 `p` 到其前驱 `pre` 的线索 (`p->lchild = pre`)。
    - 检查 `pre` 的右孩子是否为空，若为空，则建立 `pre` 到其后继 `p` 的线索 (`pre->rchild = p`)。
    - 访问结束后，更新 `pre` 为 `p`，以便为下一个节点服务。
- **算法实现**:

```c
BiThrTree pre; // 全局变量，始终指向刚访问过的结点

// 对以p为根的二叉树进行中序线索化
void InThreading(BiThrTree p) {
    if (p) {
        InThreading(p->lchild); // 递归，线索化左子树

        // 建立前驱线索
        if (!p->lchild) {
            p->LTag = Thread;
            p->lchild = pre;
        }

        // 建立后继线索
        if (pre && !pre->rchild) {
            pre->RTag = Thread;
            pre->rchild = p;
        }

        pre = p; // 保持pre指向p的前驱

        InThreading(p->rchild); // 递归，线索化右子树
    }
}

// 主函数，建立带头结点的中序线索二叉树
void CreateInThread(BiThrTree *Thrt, BiThrTree T) {
    *Thrt = (BiThrTree)malloc(sizeof(BiThrNode)); // 创建头结点
    (*Thrt)->LTag = Link;
    (*Thrt)->RTag = Thread;
    (*Thrt)->rchild = *Thrt; // 右指针回指
    if (!T) {
        (*Thrt)->lchild = *Thrt; // 若树空，左指针回指
    } else {
        (*Thrt)->lchild = T;
```

```
        pre = *Thrt;
        InThreading(T);
        pre->rchild = *Thrt; // 处理最后一个结点
        pre->RTag = Thread;
        (*Thrt)->rchild = pre; // 头结点的后继是最后一个结点
    }
}
```

3. 遍历线索二叉树

有了线索，遍历不再需要栈。以带头结点的中序线索二叉树为例：

- **算法思想**: 从根节点开始，先沿左孩子指针链找到中序序列的第一个节点。然后，利用后继线索不断访问下一个节点，直到遍历完成。

```
// 遍历中序线索二叉树
void InOrderTraverse_Thr(BiThrTree T, void (*Visit)(TElemType e)) {
  BiThrTree p = T->lchild; // p指向根结点
  while (p != T) {      // 空树或遍历结束时p==T
     // 1. 沿左孩子向下，找到中序序列的第一个（或子树第一个）结点
     while (p->LTag == Link) {
        p = p->lchild;
     }
     Visit(p->data);

     // 2. 沿后继线索访问后继结点，直到一个有右孩子的结点
     while (p->RTag == Thread && p->rchild != T) {
         p = p->rchild;
         Visit(p->data);
     }
     // 3. 转向右子树
     p = p->rchild;
  }
}
```

6.6 树、森林的表示方法

1. 双亲表示法

每个节点存储其数据和其双亲在数组中的位置。这种方法**找双亲**非常方便，但**找孩子**需要遍历整个结构。

```c
#define MAX_TREE_SIZE  100
typedef struct PTNode { //结点结构
    Elem  data;
    int    parent;   // 双亲位置域
} PTNode;

typedef struct {    //树结构
    PTNode  nodes[MAX_TREE_SIZE];
    int    r, n;
    // 根结点的位置和结点个数
} PTree;

```

2. 孩子链表表示法

将每个节点的孩子们用一个单链表连接起来。结构中包含一个节点数组，每个数组元素包含节点数据和指向其孩子链表的头指针。

```c
typedef struct CTNode { //孩子结点结构:
    int          child;
    struct CTNode *next;
} *ChildPtr;

typedef struct {    //双亲结点结构
    Elem    data;
    int    parent;   // 双亲位置域
    ChildPtr  firstchild; // 孩子链的头指针
} CTBox;

typedef struct {    //树结构
    CTBox  nodes[MAX_TREE_SIZE];
    int    n, r;
    // 结点数和根结点的位置
} CTree;

```

3. 孩子兄弟表示法 (二叉链表表示法)

这是一种非常巧妙和常用的方法，可以将任意树转换为唯一的二叉树。

- **规则**:
    - 节点的**第一个孩子**作为其二叉树形态的**左孩子**。
    - 节点的**下一个兄弟**作为其二叉树形态的**右孩子**。
- **优点**: 结构统一，所有对树的操作都可以转化为对二叉树的操作。

```c
typedef struct CSNode{  //节点结构
     Elem          data;
     struct CSNode
           *firstchild, *nextsibling;
} CSNode, *CSTree;
```

森林与二叉树的转换

- **森林 -> 二叉树**:
    1. 将森林中的每棵树用孩子兄弟表示法转换成二叉树。
    2. 将第二棵树的根作为第一棵树根的右孩子，第三棵树的根作为第二棵树根的右孩子，以此类推。
- **二叉树 -> 森林**:
    1. 逆向操作。从根节点开始，沿着右孩子链断开，可以得到多棵树的根。
    2. 对每棵分离出的树，递归地将节点的左子树转换为其孩子，右子树转换为其兄弟。

6.7 树和森林的遍历

- **树的先根遍历**: 访问根节点，然后依次先根遍历各棵子树。这等价于其对应的**二叉树的先序遍历。**
- **树的后根遍历**: 先依次后根遍历各棵子树，然后访问根节点。这等价于其对应的**二叉树的中序遍历。**
- **森林的先序遍历**: 依次对森林中的每一棵树进行先根遍历。
- **森林的中序遍历**: 依次对森林中的每一棵树进行后根遍历。

森林	树	二叉树
先序遍历	先根遍历	先序遍历
中序遍历	**后根遍历**	中序遍历

6.8 哈夫曼树与哈夫曼编码

哈夫曼树 (Huffman Tree)

- **节点路径长度**: 从根到某节点的路径上的分支数目~~（包括根节点但不包括该节点）~~。
- **树的路径长度**：树中叶子结点的路径长度之和。
- **带权路径长度 (WPL)**: 树中所有**叶子节点**的带权路径长度之和，记为 $WPL(T) = \sum w_k l_k$。
- **最优二叉树 (哈夫曼树)**: 在所有含 n 个带权叶子节点的二叉树中，带权路径长度 WPL 最小的树。

构造哈夫曼树 (哈夫曼算法)

1. 根据给定的 n 个权值 { $w_1, w_2, ..., w_n$ } 构成 n 棵二叉树的集合 F，每棵树 i 只有一个带权值为 w_i 的根节点。
2. 在 F 中选取**权值最小**的两棵二叉树，作为 [左右子树] 构造一棵新的二叉树，新树的根节点权值为其左右子树根节点权值之和。
3. 从 F 中删除这两棵树，同时加入新生成的树。
4. 重复步骤 2 和 3，直到 F 中只剩一棵树为止。这棵树就是哈夫曼树。

特点: 权值越大的叶子节点离根越近，权值越小的叶子节点离根越远。

哈夫曼编码 (Huffman Coding)

- **前缀编码**: 任何一个字符的编码都不是另一个字符编码的前缀。
- **构造**:
  1. 将待编码的字符集作为叶子节点，其出现频率作为权值，构造一棵哈夫曼树。
  2. 约定树中从根到叶子节点的路径上，左分支代表 0，右分支代表 1。
  3. 从根到每个叶子节点的路径所构成的 0/1 序列，即为该叶子节点对应字符的哈夫曼编码。
- **优点**: 是一种最优前缀编码，能使编码序列总长度最短。