第六章 树和二叉树

6.1 树的类型定义

数据对象 D：D 是具有相同特性的数据元素的集合。

数据关系 R：若 D 为空集，则称为空树；

否则:

⑴在 D 中存在唯一的称为根的数据元素 root,

⑵ 当n>1 时，其余结点可分为m (m>0) 个互不相交的有限集 T_1, T_2, …, T_m，其中每一棵子集本身又是一棵符合本定义的树，称为根root的子树。

基本操作：

查找：

Root(T); Value(T, cur_e); Parent(T, cur_e);

LeftChild(T, cur_e); RightSibling(T, cur_e);

TreeEmpty(T); TreeDepth(T);

TraverseTree(T, Visit());

插入：

InitTree(&T); CreateTree(&T, definition);

Assign(T, cur_e, value); InsertChild(&T, &p, i, c);

DestroyTree(&T);

删除：

ClearTree(&T); DestroyTree(&T);

DeleteChild(&T, &p, i);

有向树 ：

1） 有确定的根；

2） 树根和子树根之间为有向关系

有序树和无序树的区别在于：

子树之间是否存在次序关系？

和线性结构的比较

线性结构	树结构
第一个数据元素(无前驱)	根结点(无前驱)
最后一个数据元素(无后继)	多个叶子结点(无后继)
其它数据元素 (一个前驱、一个后继)	树中其它结点 (一个前驱、多个后继)

基本术语

结点:数据元素 + 若干指向子树的分支

结点的度:分支的个数

树的度:树中所有结点的度的最大值

叶子结点:度为零的结点

分支结点:度大于零的结点

从根到结点的路径：

孩子结点、双亲结点、兄弟结点、祖先、子孙

结点的层次:假设根结点的层次为 1,第 /层的结点的子树根结点的层次为 /+1

树的深度：树中叶子结点所在的最大层次

森林：是 m（m≥0）棵互不相交的树的集合

任何一棵非空树是一个二元组

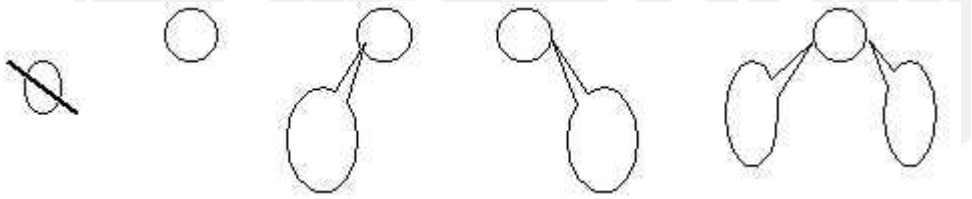
Tree = （root，F）

其中：root 被称为根结点，F 被称为子树森林

6.2 二叉树的类型定义

二叉树或为**空树**；或是由一个**根结点**加上**两棵**分别称为**左子树**和右子树的、**互不相交**的**二叉树**组成。

二叉树的五种基本形态：



二叉树的主要基本操作：

查找：

Root(T); Value(T, e); Parent(T, e);

LeftChild(T, e); RightChild(T, e);

LeftSibling(T, e); RightSibling(T, e);

BiTreeEmpty(T); BiTreeDepth(T);

PreOrderTraverse(T, Visit());

InOrderTraverse(T, Visit());

PostOrderTraverse(T, Visit());

LevelOrderTraverse(T, Visit());

插入：

InitBiTree(&T); Assign(T, &e, value);

CreateBiTree(&T, definition);

InsertChild(T, p, LR, c);

删除：

ClearBiTree(&T); DestroyBiTree(&T);

DeleteChild(T, p, LR);

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}个结点(i≥1)

性质 2 ：深度为k的二叉树上至多含 2^k-1 个结点（k≥1）

性质 3 ： 对任何一棵二叉树，若他含有n_0个叶子结点、n_2个度为 2 的结点，则必存在关系式：$n_0 = n_2+1$

两类特殊的二叉树：

满二叉树：指的是深度为*k*且含有 2^k-1 个结点的二叉树

完全二叉树：树中所含的 n 个结点和满二叉树中**编号为 1 至 n 的结点**一一对应

性质 4 ：具有 *n* 个结点的完全二叉树的**深度**为 $\lfloor \log_2 n \rfloor + 1$

性质 5 ：若对含 n 个结点的二叉树从上到下且从左至右进行 1 至 n 的编号，则对二叉树中任意一个编号为 i 的结点：

(1) 若 i=1，则该结点是二叉树的根，无双亲，

否则，编号为 $\lfloor i/2 \rfloor$ 的结点为其双亲结点；

(2) 若 2i>n，则该结点无左孩子，

否则，编号为 2i 的结点为其左孩子结点；

(3) 若 2i+1>n，则该结点无右孩子结点，

否则，编号为 2i+1 的结点为其右孩子结点。

6.3 二叉树的存储结构

一、 二叉树的顺序存储表示

```
#define MAX_TREE_SIZE 100 // 二叉树的最大结点数

typedef TElemType SqBiTree[MAX_TREE_SIZE]; // 0 号单元存储根结点

SqBiTree bt;
```

显然，这种顺序存储结构仅适用于完全二叉树。因为，在最坏的情况下，一个深度为 k 且只有 k 个结点的单支树（树中不存在度为 2 的结点）却需要长度为 2^k-1 的一维数组。

二、二叉树的链式存储表示

1. 二叉链表

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

2. 三叉链表

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

3. 双亲链表

```
typedef struct BPTNode {
TElemType data;
int *parent; // 指向双亲的指针
char LRTag; // 左、右孩子标志域
} BPTNode
typedef struct {
BPTNode nodes[MAX_TREE_SIZE];
int num_node; // 结点数目
} BPTree
```

4. 线索链表

6.4 二叉树的遍历

一、问题的退出

顺着某一条搜索路径巡访二叉树中的结点，使得每个结点均被访问一次，而且仅被访问一次。

“访问”的含义可以很广，如：输出结点的信息等。

“遍历”是任何类型均有的操作，对线性结构而言，只有一条搜索路径(因为每个结点均只有一个后继)，故不需要另加讨论。而二叉树是非线性结构，每个结点有两个后继，则存在如何遍历即按什么样的搜索路径遍历的问题。

对“二叉树”而言，可以有三条搜索路径：

1．先上后下的按层次遍历；

2．先左（子树）后右（子树）的遍历；

3．先右（子树）后左（子树）的遍历。

二、先左后右的遍历算法

先（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

（1）访问根结点；

（2）先序遍历左子树；

（3）先序遍历右子树。

中（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

（1）中序遍历左子树；

（2）访问根结点；

（3）中序遍历右子树。

后（根）序的遍历算法：

若二叉树为空树，则空操作；否则，

（1）后序遍历左子树；

（2）后序遍历右子树；

（3）访问根结点。

三、算法的递归描述

```
void Preorder (BiTree T, void( *visit)(TElemType& e))
{ // 先序遍历二叉树
if (T) {
visit(T->data); // 访问结点
Preorder(T->lchild, visit); // 遍历左子树
Preorder(T->rchild, visit); // 遍历右子树
}
}
```

四、中序遍历算法的非递归描述

```
BiTNode *GoFarLeft(BiTree T, Stack *S){
if (!T ) return NULL;
while (T->lchild )
{
Push(S, T);
T = T->lchild;
}
return T;
}
// 非递归中序遍历的算法
```

```
void Inorder_I(BiTree T, void (*visit)(TelemType& e))
{
Stack *S;
t = GoFarLeft(T, S); // 找到最左下的结点
while(t){
visit(t->data);
if (t->rchild)
t = GoFarLeft(t->rchild, S);
else if ( !StackEmpty(S )) // 栈不空时退栈
t = Pop(S);
else // 栈空表明遍历结束
t = NULL;
}
}
```

五、遍历算法的应用举例:

1、统计二叉树中叶子结点的个数(先序遍历)

```
void CountLeaf (BiTree T, int& count)
{
if ( T )
{
if ((!T->lchild)&& (!T->rchild))
count++;
CountLeaf( T->lchild, count); // 统计左子树中叶子结点个数
```

```
CountLeaf( T->rchild, count); // 统计右子树中叶子结点个数
}
}
```

2、求二叉树的深度(后序遍历)

```
int Depth (BiTree T )
{
if ( !T )
depthval = 0;
else
{
depthLeft = Depth( T->lchild );
depthRight= Depth( T->rchild );
depthval = 1 +
(depthLeft> depthRight?depthLeft:depthRight);
}
return depthval;
}
```

3、复制二叉树(后序遍历)

```
// 生成一个二叉树的结点
BiTNode *GetTreeNode(TElemType item,
BiTNode *lptr , BiTNode *rptr ){
if (!(T = (BiTNode*)malloc(sizeof(BiTNode))))
```

```
exit(1);
T-> data = item;
T-> lchild = T-> rchild = NULL;
return T;
}
BiTNode *CopyTree(BiTNode *T)
{
if (!T )
return NULL;
if (T->lchild )
newlptr = CopyTree(T->lchild);
else newlptr = NULL;
if (T->rchild )
newrptr = CopyTree(T->rchild);
else newrptr = NULL;
newnode = GetTreeNode(T->data, newlptr, newrptr);
return newnode;
}
```

4、建立二叉树的存储结构

按给定的先序序列建立二叉链表

```
Status CreateBiTree(BiTree &T) {
// 按先序次序输入二叉树中结点的值（一个字符），空格字符
// 表示空树，构造二叉链表表示的二叉树 T。
```

```
scanf(&ch);
if (ch==' ') T = NULL;
else {
if (!(T = (BiTNode *)malloc(sizeof(BiTNode))))
exit(OVERFLOW);
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
```

按给定的表达式建相应二叉树

- **由先缀表示式建树**

例如:已知表达式的先缀表示式

-×+ a b c / d e

- **由原表达式建树**

例如:已知表达式 **(a+b)×c – d/e**

```
void CrtExptree(BiTree &T, char exp[] ) {
// 建立由合法的表达式字符串确定的只含二元操作符的
// 非空表达式树，其存储结构为二叉链表
InitStack(S); Push(S,'#' );
InitStack(PTR);
```

```
p = exp; ch = *p;
while (!(GetTop(S)=='#' && ch=='#')) {
if (!IN(ch, OP)) CrtNode( t, ch ); // 建叶子结点
else {
switch (ch) {
case '(' : Push(S, ch); break;
case ')' : {
Pop(S, c);
while (c!='(' )
{ CrtSubtree( t, c); Pop(S, c) }
break; }
defult : {
while(!Gettop(S, c) && ( precede(c,ch)))
{ CrtSubtree( t, c); Pop(S, c); }
if ( ch!='#' ) Push( S, ch);
break;
} // defult
} // switch
} // else
if ( ch!='#' ) { p++; ch = *p;}
} // while
Pop(PTR, T);
} // CrtExptree
```

其中的建子树算法分别如下：

```
void CrtNode(BiTree& T, char ch) {
// 建叶子结点^T，结点中的数据项为操作数 ch
T = (BiTNode*)malloc(sizeof(BiTNode));
T->data = ch;
T->lchild = T->rchild = NULL
Push( PTR, T);
}
void CrtSubtree (Bitree& T, char c) {
// 建子树^T，其中根结点的数据项为操作符 c
T = (BiTNode*)malloc(sizeof(BiTNode));
T->data = c;
Pop(PTR, rc); T->rchild = rc;
Pop(PTR, lc); T->lchild = lc;
Push(PTR, T);
}
```

- **由表达式的先缀和中缀表示式建树**

例如：已知表达式的

先缀表示式：- × + a b c / d e

中缀表示式：a + b × c – d / e

分析可得：**根**结点是 ' – ' **左**子树是 ' a+ b× c '

右子树是 ' d / e '

6.5 线索二叉树

1. 何谓线索二叉树？

遍历二叉树的结果是，求得结点的一个线性序列。

指向该线性序列中的“前驱”和“后继”的**指针**，称作“**线索**”

包含“线索”的存储结构，称作“**线索链表**”；

与其相应的二叉树，称作“**线索二叉树**”

对线索链表中结点的约定：

在二叉链表的结点中增加两个标志域，并作如下规定：

若该结点的左子树不空，

则 lchild 域的指针指向其左子树，且左标志域的值为 0；

否则，lchild 域的指针指向其“前驱”，且左标志的值为 1。

若该结点的右子树不空，

则 rchild 域的指针指向其右子树，且右标志域的值为 0；

否则，rchild 域的指针指向其“后继”，且右标志的值为 1。

线索链表的结构描述：

```
typedef enum { Link, Thread } PointerThr;
// Link==0:指针，Thread==1:线索
typedef struct BiThrNode{
TElemType data;
struct BiThrNode *lchild, *rchild; // 左右指针
PointerThr LTag, RTag; // 左右标志
} BiThrNode, *BiThrTree;
```

二、线索链表的遍历算法：

```
for ( p = firstNode(T); p; p = Succ(p) )
Visit(p);
```

中序线索化链表的遍历算法：

※中序遍历的第一个结点 ？

※在中序线索化链表中结点的后继 ？

```
Status InOrderTraverse_Thr(BiThrTree T, Status (*Visit)(TElemType e))
{
// T 指向头结点，头结点的左链 lchild 指向根结点，头结点的右链 lchild
// 指向中序遍历的最后一个结点。中序遍历二叉线索链表表示的二叉树，
// 对每个数据元素调用函数 Visit。
p = T->lchild; // p 指向根结点
while (p != T) { // 空树或遍历结束时，p==T
while (p->LTag==Link) p = p->lchild;
if (!Visit(p->data)) return ERROR; // 访问其左子树为空的结点
while (p->RTag==Thread && p->rchild!=T) {
p = p->rchild; Visit(p->data); // 访问后继结点
}
p = p->rchild; // p 进至其右子树根
}
return OK;
} // InOrderTraverse_Thr
```

三、如何建立线索链表？

在中序遍历过程中修改结点的左、右指针域，以保存当前访问结点的“前驱”和“后继”信息。

遍历过程中，附设指针 pre,并始终保持指针 pre 指向当前访问的、指针 p 所指结点的前驱

```
Status InOrderThreading(BiThrTree &Thrt, BiThrTree T) {
// 中序遍历二叉树 T，并将其中序线索化，Thrt 指向头结点。
if (!(Thrt = (BiThrTree)malloc(sizeof(BiThrNode)))) exit (OVERFLOW);
Thrt->LTag = Link; Thrt->RTag =Thread; // 建头结点
Thrt->rchild = Thrt; // 右指针回指
if (!T) Thrt->lchild = Thrt; // 若二叉树空，则左指针回指
else {
Thrt->lchild = T; pre = Thrt;
InThreading(T); // 中序遍历进行中序线索化
pre->rchild = Thrt; pre->RTag = Thread; // 最后一个结点线索化
Thrt->rchild = pre;
}
return OK;
} // InOrderThreading
void InThreading(BiThrTree p) {
if (p) {
InThreading(p->lchild); // 左子树线索化
if (!p->lchild) { p->LTag = Thread; p->lchild = pre; } // 建前驱线索
if (!pre->rchild) { pre->RTag = Thread; pre->rchild = p; } // 建后继线索
```

```
pre = p; // 保持 pre 指向 p 的前驱
InThreading(p->rchild); // 右子树线索化
}
} // InThreading
```

6.6 树和森林的表示方法

树的三种存储结构

一、双亲表示法：

```
#define MAX_TREE_SIZE 100
```

结点结构：

```
typedef struct PTNode {
Elem data;
int parent; // 双亲位置域
} PTNode;
```

树结构：

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

二、孩子链表表示法：

孩子结点结构：

```
typedef struct CTNode {
int child;
```

```
struct CTNode *next;
} *ChildPtr;
```

双亲结点结构

```
typedef struct {
Elem data;
ChildPtr firstchild; // 孩子链的头指针
} CTBox;
```

树结构：

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

三、树的二叉链表(孩子-兄弟）存储表示法

```
typedef struct CSNode{
Elem data;
struct CSNode *firstchild, *nextsibling;
} CSNode, *CSTree;
```

森林和二叉树的对应关系

设森林 $F = (T_1, T_2, \cdots, T_n)$;

$T_1 = (root, t_{11}, t_{12}, \cdots, t_{1m})$;

二叉树 B =(LBT, Node(root), RBT);

则**由森林转换成二叉树**的转换规则为：

若 F = Φ，则 B = Φ；

否则，

由 ROOT(T_1) 对应得到 Node(root);

由 (t_{11}, t_{12}, …, t_{1m}) 对应得到 LBT;

由 (T_2, T_3,…, T_n) 对应得到 RBT.

由二叉树转换为森林的转换规则为:

若 B = Φ, 则 F = Φ;

否则,

由 Node(root) 对应得到 ROOT(T_1);

由LBT 对应得到 (t_{11}, t_{12}, …, t_{1m});

由RBT 对应得到 (T_2, T_3, …, T_n)

由此,树的各种操作均可对应到二叉树的操作来完成

换言之,和树对应的二叉树,其左、右子树的概念改变成:

左是孩子,右是兄弟

6.7 树和森林的遍历

树的遍历可有三条搜索路径:

先根(次序)遍历:

若树不空,则先访问根结点,然后依次先根遍历各棵子树。

后根(次序)遍历:

若树不空,则先依次后根遍历各棵子树,然后访问根结点。

按层次遍历:

若树不空,则自上而下自左至右访问树中每个结点。

森林的遍历

先序遍历(对森林中的每一棵树进行先根遍历)

若森林不空，则访问森林中第一棵树的根结点；

先序遍历森林中第一棵树的子树森林；

先序遍历森林中(除第一棵树之外)其余树构成的森林。

中序遍历(对森林中的每一棵树进行后根遍历)

若森林不空，则中序遍历森林中第一棵树的子树森林；

访问森林中第一棵树的根结点；

中序遍历森林中(除第一棵树之外)其余树构成的森林。

树的遍历和二叉树遍历的对应关系 ？

树的先根遍历 对应 二叉树的 ？遍历

树的后根遍历 对应 二叉树的 ？遍历

树遍历算法的应用

举例如下：

一、求树的深度的算法：

```
int TreeDepth(CSTree T) {
if(!T) return 0;
else {
h1 = TreeDepth( T->firstchild );
h2 = TreeDepth( T->nextsibling);
return(max(h1+1, h2));
}
} // TreeDepth
```

二、输出树中所有从根到叶子的路径的算法：

```
void AllPath( Bitree T, Stack& S ) {
// 输出所有从根到叶的路径
if (T) {
Push( S, T->data );
if (!T->Left && !T->Right ) PrintStack(S);
else {
AllPath( T->Left, S );
AllPath( T->Right, S );
}
Pop(S);
}
} // AllPath
void OutPath( Bitree T, Stack& S ) {
// 输出所有从根到叶的路径
while ( !T ) {
Push(s, T->data );
if ( !T-lchild ) Printstack(s);
else OutPath( T->lchild, s );
Pop(s);
T = T->rchild;
} // while
} // OutPath
```

三、建树的存储结构的算法：

```
void CreatTree( CSTree &T ) {
// 按自上而下自左至右的次序输入双亲-孩子的有序
// 对，建立树的二叉链表。以一对空格字符作为结束标
// 志，根结点的双亲为“空”。
T = NULL;
for( scanf(&fa, &ch); ch!=' '; scanf(&fa, &ch);) {
p = GetTreeNode(ch);
EnQueue(Q,p);
if (fa ==' ' ) T = p;
else {
GetHead(Q,s);
while (s->data != fa ) {
DeQueue(Q,s); GetHead(Q,s);
}
if (!(s->firstchild)) { s->firstchild = p; r = p; }
else { r->nextsibling = p; r = p; }
}
}
} // CreateTree
```

6.8 哈夫曼树与哈夫曼编码

一、最优树的定义

结点的路径长度定义为：从根结点到该结点的路径上分支的数目。

树的路径长度定义为：树中每个结点的路径长度之和。

树的带权路径长度定义为：树中所有叶子结点的带权路径长度之和

$WPL(T) = \sum w_k l_k$ (对所有叶子结点)

在所有含 n 个叶子结点、并带相同权值的 m 叉树中，必存在一棵其带权路径长度取最小值的树，称为“**最优树**”

二、如何构造最优树

哈夫曼最早研究出一个带有一般规律的算法：

以二叉树为例：

(1) 根据给定的n个权值$\{w_1, w_2, \cdots, w_n\}$，构造$n$棵二叉树的集合$F = \{T_1, T_2, \cdots, T_n\}$，其中每棵二叉树中均只含一个带权值为$w_i$的根结点，其左、右子树为空树；

(2) 在 F中选取其根结点的权值为最小的两棵二叉树，分别作为左、右子树构造一棵新的二叉树，并置这棵新的二叉树根结点的权值为其左、右子树根结点的权值之和；

(3) 从 F中删去这两棵树，同时加入刚生成的新树；

(4) 重复(2)和(3)两步，直至 F中只含一棵树为止。

三、前缀编码

指的是，任何一个字符的编码都不是同一字符集中另一个字符的编码的前缀。

哈夫曼编码是一种最优前缀编码。

学 习 要 点

1. 熟练掌握二叉树的结构特性，了解相应的证明方法。

2. 熟悉二叉树的各种存储结构的特点及适用范围。

3. 遍历二叉树是二叉树各种操作的基础。实现二叉树遍历的具体算法与所采用的存储结构有关。不仅要熟练掌握各种遍历策略的递归和非递归算法，了解遍历过程中“ 栈 ”的作用和状态，而且能灵活运用遍历算法实现二叉树的其它操作。层次遍历是按另一种搜索策略进行的遍历。

4. 理解二叉树线索化的实质是建立结点与其在相应序列中的前驱或后继之间的直接联系，熟练掌握二叉树的线索化过程以及在中序线索化树上找给定结点的前驱和后继的方法。二叉树的线索化过程是基于对二叉树进行遍历，而线索二叉树上的线索又为相应的遍历提供了方便。

5. 熟悉树的各种存储结构及其特点，掌握树和森林与二叉树的转换方法。建立存储结构是进行其它操作的前提，因此读者应掌握 1 至 2 种建立二叉树和树的存储结构的方法。

6. 学会编写实现树的各种操作的算法。

7. 了解最优树的特性，掌握建立最优树和哈夫曼编码的方法。