树和二叉树

树的定义和基本术语

1、 树是 n 个节点的有限集。 在任意一颗非空树中：（1）有且仅有一个特定的称为根的节点；（2）当 n>1 时，其余节点可分为 m 个互不相交的有限集 T1，T2…，其中每一个集合本身又是一颗树，并且称为根的子树。
2、 树的结构定义是一个递归的定义，即在树的定义中又用到树的概念，它道出了树的固有特性。
3、 树的基本术语：
   a) **树的结点**：树的结点包含**一个数据元素及若干指向其子树的分支**。结点拥有的子树称为**结点的度**。度为 0 的节点称为**叶子或终端结点**。度不为 0 的结点称为**非终端结点或分支结点**。（除根结点之外，分支结点也成为内部结点）。
   b) **树的度：树的度是树内各结点的度的最大值**。
   c) 双亲和孩子：**结点的子树的根**称为该结点的**孩子（child）**；该结点称为孩子的双亲（parent）。同一个双亲的孩子之间互称**兄弟（sibling）**。
   d) 结点的层次：结点的层次从根开始定义。根为第一层，根的孩子为第二层。
   e) **树的深度：树中结点的最大层次称为树的深度（Depth）**（或高度）。
   f) 有序树与无序树：如果将树中结点的各子树堪称从左至右是**有顺序**的（不可互换），则称该树为**有序树**，否则称为**无序树**。
   g) 森林：森林是 m 棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。

二叉树

1、 二叉树的定义：二叉树是一种树型结构，它的特点是**每个结点至多只有两棵子树**。**即二叉树中不存在度大于 2 的结点**。并且，二叉树的子树有左右之分，其次序不能任意颠倒。
2、 抽象数据类型二叉树的定义，详见书 P121
3、 **满二叉树**：一棵深度为 k 且 2^k-1 个结点的二叉树称为满二叉树。
4、 **完全二叉树**：可以对满二叉树的结点进行连续编号：从根节点其，自上而下，从左到右。**深度为 k，有 n 个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应，则称之为完全二叉树**。（满二叉树为完全二叉树）
5、 **二叉树的性质：**
   a) 在二叉树的第 i 层上至多有 2^(i-1)个结点。
   b) 深度为 k 的二叉树至多有 2^k-1 个结点
   c) 对任何一棵二叉树 T，如果其终端结点数为 n0，度为 2 的结点数为 n2，则 n0=n2+1
   d) 具有 n 个结点的完全二叉树的深度为[log2(n)]+1
   e) 如果对一棵有 n 个结点的完全二叉树的结点按层序编号，则对任一结点 i 有
      i. 如果 i=1，则结点 i 是二叉树的根，无双亲；如果 i>1，则其双亲 parent(i)为结点 i/2 再向下取整。

ii. 如果 2i>n，则结点 i 无左孩子（或者说，该结点为叶节点），否则其左孩子的结点为 2i

iii. 如果 2i+1>n，则结点 i 无右孩子，否则其右孩子的结点为 2i+1

6、二叉树的存储结构

a) 顺序存树结构

用一个数组来存储，详见书 P126

b) 链式存储结构

二叉树的结点由一个数据元素和分别指向其左、右子树的两个分支构成，则表示二叉树的链表中的结点至少包含三个域：数据域、左指针和右指针。有时，为了便于寻找双亲（Parent），还可以设置一个指向双亲结点的指针域。

遍历二叉树和线索二叉树

1、遍历二叉树

遍历基本思想：二叉树是由三个基本单元组成：根结点、左子树、右子树。因此遍历这三个部分就遍历了整个树。

a) 先序遍历：①访问根节点②先序遍历左子树③先序遍历右子树

b) 中序遍历：①中序遍历左子树②访问根节点③中序遍历右子树

c) 后序遍历：①后序遍历左子树②后序遍历右子树③访问根节点

最后总结一波：遍历实质上是对一个非线性结构进行线性化操作

例子：先序遍历 c 语言代码

```
void PreOrderTraverse(BinTree *BinT)
{
    if(BinT)
    {
        printf("%d,",BinT->data);
        PreOrderTraverse(BinT->lchild);
        PreOrderTraverse(BinT->rchild);
    }
}
```

2、线索二叉树

引：在有 n 个结点的二叉链表中必定存在 n+1 个空链域（空指针）。由此设想能否利用这些空链域来存放节点的前驱和后继的信息。

（补充：前驱：经过遍历后的前面的元素。后驱：和前驱相反）

作如下规定：若结点有左子树，则其 lchild 指向其左孩子，否则指向其前驱；若有右子树，rchild 指向右孩子，否则指向其后继。为了避免混淆，增加两个标志域：

Lchild	Ltag	Data	Rtag	Rchild

其中：ltag=0 则 lchild 指向左孩子；ltag=1 则 lchild 指向右孩子

以这种结点结构构成的二叉链表作为二叉树的存储结构，叫做**线索链表**，其中指向结点前驱和后继的指针叫做**线索**。加上线索的二叉树称为**线索二叉树**。

如图：

由上述可知，**在线索树上进行遍历，只需要找到序列中的第一个结点，然后以此找到结点后继直至其后继为空即可**。

如何找到节点后继？以中序遍历为例子，叶子结点的右指针为线索。非终端结点的右链为指针，无法从此得到后继信息。但是，根据中序遍历规律可知，**结点的后继是遍历其右子树时访问的第一个结点（右子树最左下的点）。同理，找到前驱的方法就是：遍历左子树最后的结点，即左子树最右下的结点**。

树和森林

1、 树的存储结构

在大量应用中，人们曾使用多种形式的存储结构来表示树。这里，我们介绍三种常用的链表结构

a) 双亲表示法

假设以一组连续空间存储树的结点，同时在每个结点中附设一个指示器指示其双亲结点在链表中的位置。

（优点：可以直接找到结点的双亲（Parent）；缺点：但要是寻找孩子（Child），则需要遍历整个结构）

b) 孩子表示法

把每个结点的孩子结点排列起来，看成是一个线性表，且以单链表作为存储结构，则 n 个结点有 n 个孩子链表。而 n 个头指针又组成一个线性表，为了便于查找，可采用顺序存储结构。（详见书 P136）

c) 孩子兄弟表示法

以二叉链表作为树的存储结构。链表中结点的两个链域分别指向该结点的第一个孩子结点（firstchild）和下一个兄弟结点（nextsibling）。

（优点：这种存储结构便于实现各种树的操作。例如：寻找结点孩子，寻找兄弟等）

2、 森林与二叉树的转换

3、 树和森林的遍历

树与等价问题

哈夫曼树及其应用

1、 定义：**路径长度**：从树中的一个结点到另一个结点之间的分支构成这两个结点之间的**路径**，路径上的分支数目称做路径长度。

2、 **树的路径长度**：从树根到每一结点的路径长度之和。

3、 **树的带权路径长度：**树中所有**叶子结点**的带权路径长度之和，记作 WPL=$\sum_{k=1}^{n} w_k l_k$。

4、 **最优二叉树或哈夫曼树：**一棵有 n 个叶子结点的二叉树，每个叶子结点带权为 wi，**则其中带权路径长度 WPL 最小的二叉树称为最优二叉树（哈夫曼树）**。

现在来看如何构造哈夫曼树

①根据给定的 n 个权值{w1,w2,w3,…}构成 n 棵二叉树的集合 F={T1,T2,…}, 其中每棵二叉树 Ti 中只有一个带权为 wi 的根节点，其左右子树均为空。

②在 F 中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树上根结点的权值之和。

③在 F 中删除这两棵树，同时将新得到的二叉树加入 F 中。

④重复②③，直到 F 中只含有一棵树为止，这就是哈夫曼树。

5、 哈夫曼编码

a) **前缀编码：设计编码的任意一个字符编码都不是另一个字符编码的前缀**。

下面讨论如何得到使电文总长最短的二进制前缀编码。

（提示：将字符出现的频率视为权，之后构造一个哈夫曼树）

具体做法代码（c 语言）

```
#include <stdio.h>

#define MAXBIT 100
#define MAXVALUE 10000
#define MAXLEAF 30
#define MAXNODE MAXLEAF*2 -1

typedef struct
{
    int bit[MAXBIT];
    int start;
} HCodeType; /* 编码结构体 */
typedef struct
{
    int weight;
    int parent;
    int lchild;
    int rchild;
} HNodeType; /* 结点结构体 */

/* 构造一颗哈夫曼树 */
void HuffmanTree (HNodeType HuffNode[MAXNODE], int n)
{
    /* i、j: 循环变量，m1、m2: 构造哈夫曼树不同过程中两个最小权值结点的权值，
    x1、x2: 构造哈夫曼树不同过程中两个最小权值结点在数组中的序号。*/
    int i, j, m1, m2, x1, x2;
    /* 初始化存放哈夫曼树数组 HuffNode[] 中的结点 */
    for (i=0; i<2*n-1; i++)
    {
```

```
            HuffNode[i].weight = 0;
            HuffNode[i].parent =-1;
            HuffNode[i].lchild =-1;
            HuffNode[i].lchild =-1;
        } /* end for */

/* 输入 n 个叶子结点的权值 */
        for (i=0; i<n; i++)
        {
            printf ("Please input weight of leaf node %d: \n", i);
            scanf ("%d", &HuffNode[i].weight);
        } /* end for */

/* 循环构造 Huffman 树 */
        for (i=0; i<n-1; i++)
        {
            m1=m2=MAXVALUE; /* m1、m2 中存放两个无父结点且结点权值最小的两个
结点 */
            x1=x2=0;
        /* 找出所有结点中权值最小、无父结点的两个结点，并合并之为一颗二叉树 */
            for (j=0; j<n+i; j++)
            {
                if (HuffNode[j].weight < m1 && HuffNode[j].parent==-1)
                {
                    m2=m1;
                    x2=x1;
                    m1=HuffNode[j].weight;
                    x1=j;
                }
                else if (HuffNode[j].weight < m2 && HuffNode[j].parent==-1)
                {
                    m2=HuffNode[j].weight;
                    x2=j;
                }
            } /* end for */
/* 设置找到的两个子结点 x1、x2 的父结点信息 */
            HuffNode[x1].parent = n+i;
            HuffNode[x2].parent = n+i;
            HuffNode[n+i].weight = HuffNode[x1].weight + HuffNode[x2].weight;
            HuffNode[n+i].lchild = x1;
            HuffNode[n+i].rchild = x2;

            printf ("x1.weight and x2.weight in round %d: %d, %d\n", i+1,
HuffNode[x1].weight, HuffNode[x2].weight); /* 用于测试 */
```

```
            printf ("\n");
        } /* end for */
} /* end HuffmanTree */

int main(void)
{
        HNodeType HuffNode[MAXNODE]; /* 定义一个结点结构体数组 */
        HCodeType HuffCode[MAXLEAF], cd; /* 定义一个编码结构体数组， 同时定义一个临时变量来存放求解编码时的信息 */
        int i, j, c, p, n;
        printf ("Please input n:\n");
        scanf ("%d", &n);
        HuffmanTree (HuffNode, n);

        for (i=0; i < n; i++)
        {
            cd.start = n-1;
            c = i;
            p = HuffNode[c].parent;
            while (p != -1) /* 父结点存在 */
            {
                if (HuffNode[p].lchild == c)
                    cd.bit[cd.start] = 0;
                else
                    cd.bit[cd.start] = 1;
                cd.start--; /* 求编码的低一位 */
                c=p;
                p=HuffNode[c].parent; /* 设置下一循环条件 */
            } /* end while */

            /* 保存求出的每个叶结点的哈夫曼编码和编码的起始位 */
            for (j=cd.start+1; j<n; j++)
            { HuffCode[i].bit[j] = cd.bit[j];}
                HuffCode[i].start = cd.start;
        } /* end for */

        /* 输出已保存好的所有存在编码的哈夫曼编码 */
        for (i=0; i<n; i++)
        {
            printf ("%d 's Huffman code is: ", i);
            for (j=HuffCode[i].start+1; j < n; j++)
            {
                printf ("%d", HuffCode[i].bit[j]);
            }
```

```
            printf ("\n");
        }
        getch();
        return 0;
    }
```

回溯法与树的遍历

1、 回溯法：回溯法是设计递归过程的一种重要方法，它的求解过程实质上是一个先序遍历一棵“状态树”的过程，只是这棵状态树不是遍历前预先建立的，而是隐含在遍历过程中，但如果认识到这点，很多问题的递归过程设计就迎刃而解了。

树的计数

附 A：自己写的哈夫曼编码程序（c 语言）

```
#include <stdio.h>
#include <stdlib.h>
/****
问题：字符出现概率为 0.05,0.29,0.07,0.08
0.14,0.23,0.03,0.11
试求哈夫曼编码
****/

#define N 20 //8 个字符
#define M 39 //创建十五个树
typedef struct treeNode
{
	int data;
	int exist;
	struct treeNode *lchild,*rchild,*parent;
}HaFuManTree;//定义数据结构

typedef struct codeNode
{
	char hfmcode[N+1];
	int lencode;
	int w;
}CODE;//定义编码数据结构

HaFuManTree hafuman[M];//哈夫曼树结点
CODE hcode[N];
int NUM=0;

void init_system(int *weight);
void findmin2(int *t1,int *t2);
void HaFuManCode();
int findmin(int *minval);
void Combine(HaFuManTree *t1,HaFuManTree *t2,int n);
void HaFuManDecode();
void preorder(HaFuManTree *tr);
void visit(HaFuManTree *tr);
void PrintCode();

int main()
{
	int weight[N]={5,22,7,9,14,12,3,11,1,4,18,15,16,40,32,2,86,66,250,79};//初始化权值
```

```
    init_system(weight);//初始化
    HaFuManCode();
    HaFuManDecode();
    PrintCode();
    return 0;
}

void init_system(int *weight)
{
    for(int i=0;i<N;i++)
    {
        hafuman[i].data=*(weight+i);
        hafuman[i].lchild=NULL;
        hafuman[i].rchild=NULL;
        hafuman[i].parent=NULL;
        hafuman[i].exist=1;
        hcode[i].lencode=0;
    }
    for(int i=N;i<M;i++)
    {
        hafuman[i].data=0;
        hafuman[i].lchild=NULL;
        hafuman[i].rchild=NULL;
        hafuman[i].parent=NULL;
        hafuman[i].exist=0;//不存在
    }
}

int findmin(int *minval)
{
    int min=0;
    int index=0;
    for(int i=0;i<M;i++)
        if(hafuman[i].exist==1)//存在
        {
            min=hafuman[i].data;
            index=i;
        }
    for(int i=0;i<M;i++)
    {
        if(hafuman[i].data<min&&hafuman[i].exist==1)//存在且更小
        {
            min=hafuman[i].data;
            index=i;
```

```
            }
        }
        hafuman[index].exist=0;
        *minval=min;
        return index;
}

void findmin2(int *t1,int *t2)
{
        int minval1,minval2;
        *t1=findmin(&minval1);
        *t2=findmin(&minval2);
        //printf("min:%d,%d\n",minval1,minval2);
}

void Combine(HaFuManTree *t1,HaFuManTree *t2,int n)
{
        //HaFuManTree *tr;
        /*tr=malloc(sizeof(HaFuManTree));
        if(tr==NULL)
        {
                printf("wrong\n");
                exit(1);
        }*/
        hafuman[n].data=t1->data+t2->data;
        hafuman[n].lchild=t1;
        hafuman[n].rchild=t2;
        //t1->parent=&hafuman[n];
        //t2->parent=&hafuman[n];
        //hafuman[n].parent=NULL;
        hafuman[n].exist=1;//存在
}

void HaFuManCode()
{
        for(int i=N;i<M;i++)
        {
                int t1,t2;
                //printf("min:%d,%d\n",t1->data,t2->data);
                findmin2(&t1,&t2);
                //printf("min:%d,%d\n",hafuman[t1].data,hafuman[t2].data);
                Combine(&hafuman[t1],&hafuman[t2],i);//合并
        }
}
```

```
void visit(HaFuManTree *tr)
{
	if(tr->lchild==NULL&&tr->rchild==NULL)
	{
		hcode[NUM].w=tr->data;
		//printf("\nweight=%d\n",hcode[NUM].w);
		for(int i=0;i<hcode[NUM].lencode-1;i++)
		{
			hcode[NUM+1].hfmcode[i]=hcode[NUM].hfmcode[i];
		}
		hcode[NUM].hfmcode[hcode[NUM].lencode--]='\0';
		hcode[NUM+1].lencode=hcode[NUM].lencode;
		NUM++;
	}
	else
	{
		hcode[NUM].lencode--;
	}
	//printf("data:%d\n",tr->data);
}
void preorder(HaFuManTree *tr)
{
	if(tr)
	{


//printf("NUM:%d,length:%d,code:%s\n",NUM,hcode[NUM].lencode,hcode[NUM].hfmcode);
		hcode[NUM].hfmcode[hcode[NUM].lencode++]='0';//左走为 0，长度加 1
		//printf("to                                                    left
NUM:%d,length:%d,code:%s\n",NUM,hcode[NUM].lencode,hcode[NUM].hfmcode);
		preorder(tr->lchild);

		hcode[NUM].hfmcode[hcode[NUM].lencode++]='1';//右走为 1，长度加 1
		//printf("to                                                   right
NUM:%d,length:%d,code:%s\n",NUM,hcode[NUM].lencode,hcode[NUM].hfmcode);
		preorder(tr->rchild);
		visit(tr);
	}
	else
	{
		hcode[NUM].lencode--;
	}
	//返回前长度减一
}
```

```
void HaFuManDecode()
{
	preorder(&hafuman[M-1]);
}
void PrintCode()
{
	for(int i=0;i<N;i++)
	{
		printf("权值为%d 字符的编码为：%s\n",hcode[i].w,hcode[i].hfmcode);
	}
}
```

附 B：自己写的二叉树遍历（c 语言）

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef int TElemtype;
typedef struct BiTNode
{
	TElemtype data;
	struct BiTNode *lchild,*rchild;
}BinTree;

typedef struct Info_BinTree
{
	int number;
	BinTree *head;
}BinTreeList;

void InitBinTree(BinTreeList *BinT);//初始化一个二叉搜索树
BinTree* InsertBinTree(BinTree *BinT,TElemtype indata);//插入数据
void PreOrderTraverse(BinTree *BinT);//先序遍历
BinTree* DeleteBinTree(BinTree *BinT,TElemtype x);//删除节点
BinTree* Findmax(BinTree *BinT);//找到最大的节点，返回一个这样的指针

int main()
{
	BinTreeList BinaryTree;
	int i,n,data,todelete;
	char ch;

	InitBinTree(&BinaryTree);
```

```
	printf("请输入您要存入数据的个数:");
	scanf("%d",&n);
	for(i=0;i<n;i++)
	{
		scanf("%d",&data);
		BinaryTree.head=InsertBinTree(BinaryTree.head,data);
	}
	printf("按任意键开始遍历！\n");
	getch();
	PreOrderTraverse(BinaryTree.head);

	printf("是否删除元素？");
	fflush(stdin);
	ch=getchar();
	if(ch=='y'||ch=='Y')
	{
		printf("请输入您要删除的元素：");
		scanf("%d",&todelete);
		BinaryTree.head=DeleteBinTree(BinaryTree.head,todelete);

		printf("按任意键遍历！");
		getch();
		PreOrderTraverse(BinaryTree.head);
	}

	getch();
	return 0;
}

void InitBinTree(BinTreeList *BinT)
{
	BinTree *BT;
	BT=(BinTree*)malloc(sizeof(BinTree));
	if(!BT)
	{
		printf("分配内存失败！\n");
		getch();
		exit(1);
	}
	BinT->head=NULL;
	BinT->number=0;
}

BinTree* InsertBinTree(BinTree *BinT,TElemtype indata)
```

```
{

    if(!BinT)
    {
        BinT=(BinTree*)malloc(sizeof(BinTree));
        BinT->data=indata;
        BinT->lchild=BinT->rchild=NULL;
    }
    else
    {
        if(indata>BinT->data)
        {
            BinT->rchild=InsertBinTree(BinT->rchild,indata);
        }
        else if(indata<BinT->data)
        {
            BinT->lchild=InsertBinTree(BinT->lchild,indata);
        }
    }
    return BinT;
}

void PreOrderTraverse(BinTree *BinT)
{
    if(BinT)
    {
        printf("%d,",BinT->data);
        PreOrderTraverse(BinT->lchild);
        PreOrderTraverse(BinT->rchild);
    }
}

BinTree* DeleteBinTree(BinTree *BinT,TElemtype x)
{
    BinTree *Tmp;
    if(!BinT)
    {
        printf("元素未找到！");
    }
    else
    {
        if(x>BinT->data)//要删除的大于目前节点，去右边找
        {
            BinT->rchild=DeleteBinTree(BinT->rchild,x);
```

```
        }
        else if(x<BinT->data)
        {
            BinT->lchild=DeleteBinTree(BinT->lchild,x);
        }
        else//元素就是这个节点的
        {
            if(BinT->lchild&&BinT->rchild)//有左右两个孩子，两个办法，从右面拿出来最小的替换 或 左面拿个最大的替换
            {
                Tmp=Findmax(BinT->lchild);
                BinT->data=Tmp->data;
                BinT->lchild=DeleteBinTree(BinT->lchild,BinT->data);//从这个节点的左子树下删除那个最大的
            }
            else//有一个孩子或没有
            {
                Tmp=BinT;
                if(!BinT->lchild)//没有左孩子
                {
                    BinT=BinT->rchild;
                }
                else//没有右孩子或者没孩子
                {
                    BinT=BinT->lchild;
                }
                free(Tmp);
            }
        }
        //printf("删除成功！\n");
    }
    return BinT;
}

BinTree* Findmax(BinTree *BinT)
{
    if(BinT->rchild)
    {
        BinT->rchild=Findmax(BinT->rchild);
    }
    return BinT;
}
```