第5章 chapter 5

树

本章导读

树状结构是一种应用很广泛的非线性结构，是一种以分支关系定义的层次结构。本章需要了解树的定义和表示方法，以及树的遍历等操作，最后介绍树结构的几个应用实例。

教学目标

通过本章的学习，要求读者掌握以下内容。

- 树的概念和树的基本操作。
- 二叉树的定义及存储结构。
- 二叉树的遍历。
- 二叉树的应用，包括二叉排序树、哈夫曼树等。
- 树的存储，以及树、森林和二叉树的相互转换。

5.1 树的定义与表示

树状结构是一类重要的非线性结构，是以分支关系定义的层次结构。树状结构在现实生活和计算机领域都有广泛的应用。本节着重介绍树的基本定义和常用术语，以便用户对树状结构有一个全面的理解。

5.1.1 树的定义及基本术语

树是由 $n(n\geqslant 0)$ 个结点组成的有限集合 T。当 $n=0$ 时，称为空树，否则称为非空树。在任一非空树中：

- 有且仅有一个特定的称为根的结点。
- 除根结点之外的其余结点被分成 $m(m\geqslant 0)$ 个互不相交的集合 $T_1,T_2,\cdots,T_m$。且其中每一个集合本身又是一棵树，它们被称为根的子树。

显然，这是一个递归定义，即在树的定义中又用到了树的概念。它反映了树的固有特性，即树中每一个结点都是该树中某一棵子树的根。

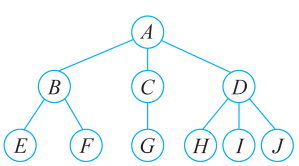


图 5-1 树 T

在如图 5-1 所示的树 T 中，A 是根结点，其余结点分成三个互不相交的子集 $T_1=\{B,E,F\}$，$T_2=\{C,G\}$，$T_3=$

$\{D,H,I,J\}$。T_1、T_2、T_3 都是根结点 A 的子树，B、C、D 分别为这三棵子树的根。而子树本身也是树，按照定义可继续划分，如 T_1 中 B 为根结点，其余结点又可分为两个互不相交的子集 $T_{11}=\{E\}$ 和 $T_{12}=\{F\}$，显然 T_{11}、T_{12} 是只含一个根结点的树（对于 T_2、T_3，可做类似的划分）。由此可见，树中每一个结点都是该树中某一棵子树的根。

下面介绍树结构中常用的术语。

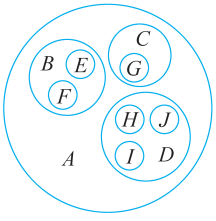
树中结点之间的连线称为分支。结点的子树个数称为结点的度。一棵树中结点度的最大值称为树的度。度为零的结点称为叶子结点或终端结点。度不为零的结点称为分支结点或非终端结点。结点的各子树的根称为该结点的孩子，反之，该结点称为孩子的双亲。同一双亲下的同层结点称为兄弟。将这些关系进一步推广，结点的祖先是从根到该结点所经分支上的所有结点，反之，以该结点为根的子树上所有结点都是此结点的子孙。例如，如图 5-1 所示的树 T 中，B 的度为 2，是分支结点。E 的度为 0，是叶子结点。树的度为 3。B、C、D 是兄弟。A 是 E 的祖先，B 的子孙是 E、F。

结点的层次是从根结点算起的，设根结点在第一层上，则根结点的孩子为第二层。若某结点在第 L 层，则其子树的根就在第 $L+1$ 层。树中结点的最大层次称为树的高度或深度。图 5-1 中树 T 的高度为 3。

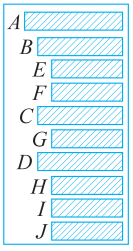
森林是 $n(n\geqslant 0)$ 棵互不相交的树的集合。森林的概念与树非常接近，如果去掉一棵树的根，就得到森林。例如，在图 5-1 中去掉根结点 A，就得到由三棵树 T_1、T_2、T_3 组成的森林。反之，给由 n 棵树组成的森林加一个根结点，就生成一棵树。

5.1.2 树的表示

树的表示方法除了如图 5-1 所示外，还可以用一种表示集合包含关系的文氏图来表示，如图 5-2(a)所示；或用凹入法来表示，如图 5-2(b)所示；或用广义表的形式表示，根作为由子树森林组成的表的名字写在表的左边，如图 5-2(c)所示。



(a) 文氏图表示法



(b) 凹入表示法

(A(B(E, F), C(G), D(H, I, J))

(c) 广义表表示法

图 5-2 树的不同表示法

5.2 二叉树及其遍历

5.2.1 二叉树的定义

二叉树是 $n(n\geqslant 0)$ 个结点的有限集合，它或为空二叉树（$n=0$），或由一个根结点和

两棵分别称为根的左子树和右子树的互不相交的二叉树组成，这是二叉树的递归定义。根据这个定义，可以导出二叉树的5种基本形态，如图5-3所示。其中，图5-3(a)为空二叉树，图5-3(b)为仅有一个根结点的二叉树，图5-3(c)为右子树为空的二叉树，图5-3(d)为左子树为空的二叉树，图5-3(e)为左、右子树均为非空的二叉树。

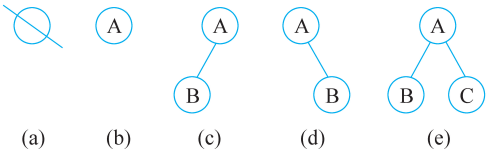


图5-3　二叉树的5种形态

5.2.2　二叉树的重要性质

1. 二叉树的性质

二叉树具有下列重要特性。

性质1　在二叉树的第 i 层上，至多有 2^{i-1} 个结点($i \geqslant 1$)。

可利用归纳法来证明性质1的正确性。根据结点层次的定义，二叉树的根结点在第一层上，当 $i=1$ 时，只有一个根结点，$2^{i-1}=2^0=1$，则上述结论成立。若第 $j-1$ 层上有 2^{j-2} 个结点($1 \leqslant j \leqslant i$)，由于二叉树中每个结点至多有两个孩子结点，若其结点在第 $j-1$ 层，则孩子结点必在第 j 层，故在第 j 层上最多有 $2 \times 2^{j-2}=2^{j-1}$ 个结点。由此，结论成立。

性质2　高度为 k 的二叉树中至多含 2^k-1 个结点($k \geqslant 1$)。

由性质1可见，高度为 k 的二叉树的最大结点数为

$$\sum_{i=1}^{k}(\text{第 } i \text{ 层上的最大结点数}) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

性质3　在任意一棵二叉树 T 中，若其叶子结点数为 n_0，度为1的结点数为 n_1，度为2的结点数为 n_2，由于二叉树中所有结点的度均小于或等于2，所以其结点总数为

$$n = n_0 + n_1 + n_2 \tag{5-1}$$

二叉树中除根结点之外的每个结点都有一个指向其双亲结点的分支，则分支数 B 和结点总数 n 之间存在如下关系。

$$n = B + 1 \tag{5-2}$$

从另一个角度看，这些分支可看成度为1的结点和度为2的结点与它们的孩子结点之间的连线，则分支数 B 和 n_1 及 n_2 之间存在下列关系。

$$B = n_1 + 2n_2$$

代入式(5-2)中得

$$n = n_1 + 2n_2 + 1 \tag{5-3}$$

由式(5-1)和式(5-3)，可得

$$n_0 + n_1 + n_2 = n_1 + 2n_2 + 1$$

化简得

$$n_0 = n_2 + 1$$

2. 完全二叉树和满二叉树的性质

完全二叉树和满二叉树是两种特殊形态的二叉树。

满二叉树：一棵高度为 k 且含有 2^k-1 个结点的二叉树称为满二叉树。对一棵满二叉树，若从第 1 层的根结点开始，自上而下、从左到右对结点进行连续编号，则可给出满二叉树的顺序表示法，如图 5-4 所示。

完全二叉树：若高度为 k、有 n 个结点的二叉树是一棵完全二叉树，当且仅当每个结点都与高度为 k 的满二叉树中编号从 1 到 n 的结点一一对应，称为完全二叉树，如图 5-5 所示。完全二叉树的特点是，除最下面一层外，每一层的结点个数都达到最大值，且最下面一层的结点都集中在该层最左边的若干位置。显然，一棵满二叉树一定是完全二叉树，但一棵完全二叉树不一定是满二叉树。

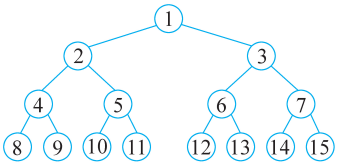


图 5-4 满二叉树

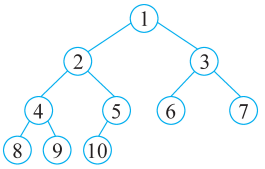


图 5-5 完全二叉树

完全二叉树具有以下两个性质。

性质 4 如果对一棵有 n 个结点的完全二叉树的结点按顺序编号，则对任一结点 $i(1\leqslant i\leqslant n)$ 有以下特性。

- 若 $i\neq 1$，则 i 的双亲结点是结点 $[i/2]$；若 $i=1$，则 i 是根结点，无双亲。
- 若 $2i\leqslant n$，则 i 的左孩子是结点 $2i$；若 $2i>n$，则 i 无左孩子。
- 若 $2i+1\leqslant n$，则 i 的右孩子是结点 $2i+1$；若 $2i+1>n$，则 i 无右孩子。

性质 5 具有 n 个结点的完全二叉树的高度为 $[\log_2 n]+1$。

证明：假设高度为 k，则根据性质 2 和完全二叉树的定义，有

$$2^{k-1}-1<n\leqslant 2^k-1$$

或

$$2^{k-1}<n\leqslant 2^k$$

于是

$$k-1\leqslant \log_2 n<k$$

由于 k 为整数，所以

$$k=[\log_2 n]+1$$

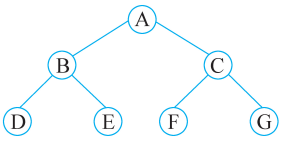
5.2.3 二叉树的存储结构

1. 顺序存储结构

存储二叉树时，是用一组连续的存储单元存储二叉树的数据元素，按满二叉树的结点顺序编号依次存放二叉树中的数据元素。用一维数组 T 存放二叉树时，如图 5-6 所示。

这种存储结构适用于存放完全二叉树和满二叉树。但对一般二叉树，这种存储结构会造成内存的浪费。如图 5-7 所示，在最坏的情况下，一个高度为 k 且只有 k 个结点的

单支树(即二叉树中没有度为2的结点),却需要2^k-1个存储单元。可见,此时二叉树使用顺序存储结构,会浪费较多存储空间。另外,顺序分配时的插入和删除操作是很不方便的,会造成大量结点的移动。因此,二叉树通常采用链式存储结构。

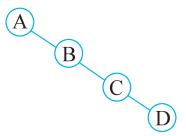


T[7] 0 1 2 3 4 5 6

A	B	C	D	E	F	G

用一维数组存放满二叉树

图 5-6 满二叉树的顺序存储结构



T[15] 0 1 2 3 4 5 6 7 … 14

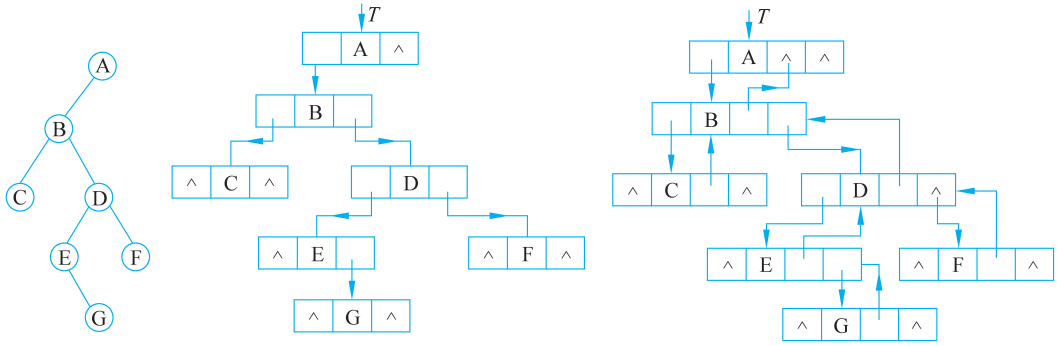
A		B				C		…	D

用一维数组存放一般二叉树

图 5-7 一般二叉树的顺序存储结构

2. 链式存储结构

由于二叉树的每一个结点最多可有左右两棵子树,故链表的结点结构除数据域外还可设两个链域:左孩子域(lchild)、右孩子域(rchild),分别指向其左、右孩子。结点由两个链域组成的链表称为二叉链表。但有时为了便于找到双亲结点,还要另设一个指向双亲的链域,即结点由三个链域组成,此链表称为三叉链表。二叉树 T 的二叉链表表示及三叉链表表示如图5-8所示。



(a) 二叉树　(b) 二叉链表表示　(c) 三叉链表表示

图 5-8 二叉树的链表及三叉链表表示示例

5.2.4 二叉树的遍历

遍历二叉树是指按一定的规律访问二叉树的每一个结点,且每个结点仅被访问一次(访问结点可理解为打印该结点数据域值或其他操作)。遍历是二叉树最重要和最基本的运算,并且有很多实际的应用。遍历对线性结点来说,是一个容易解决的问题。由于二叉树是一个非线性结构,每个结点都可能有两棵子树,所以要找到一个完整的、有规律的走法,以便使二叉树上的结点按被访问的先后顺序排列起来,得到一个线性序列。

分析二叉树的结构特性可知,一棵非空二叉树是由根结点、左子树、右子树三个基本部分组成。若分别令 D、L 和 R 表示访问根结点、遍历左子树和遍历右子树,则可有DLR、LDR、LRD、DRL、RDL、RLD 6种遍历次序。若在左、右子树的遍历次序上限定先左后右,则仅有前三种情况,分别称为前序遍历、中序遍历、后序遍历。

1. 前序遍历

前序遍历的递归定义可描述为：若二叉树不空，则进行下列操作。

- 访问根结点。
- 前序遍历左子树。
- 前序遍历右子树。

若定义二叉树的存储结构为二叉链表，则根据前序遍历的递归定义，可以写出相应的 C 语言算法描述如下。

```
#include <stdio.h>
typedef struct node                    /* 二叉树结点类型的定义 */
{
    char data;
    struct node *lchild;
    struct node *rchild;
}NODE;
void preorder(NODE *t)
{/* 前序遍历二叉树的递归算法 */
  if(t!=NULL)
     {printf("%c\t",t->data);
      preorder(t->lchild);
      preorder(t->rchild);
     }
}
void main()
 {   NODE *t;
     t=create();                       /* create 函数是建立二叉树的函数,见后面 */
     preorder(t);
 }
```

例如，如图 5-9 所示的二叉树，若按前序遍历的方法，则输出结点的序列是(－ * abc)。

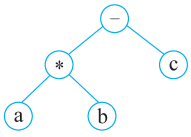


图 5-9 二叉树

2. 中序遍历

中序遍历的递归定义是：若二叉树不空，则进行下列操作。

- 按中序遍历左子树。
- 访问根结点。
- 按中序遍历右子树。

用 C 语言描述的算法如下。

```
#include <stdio.h>
typedef struct node                        /* 二叉树结点类型的定义 */
{
    char data;
    struct node *lchild;
    struct node *rchild;
}NODE;
void inorder(NODE *t)
 {/* 中序遍历二叉树的递归算法 */
  if(t!=NULL)
     { inorder(t->lchild);
```

```
    printf("%c\t",t->data);
    inorder(t->rchild);
    }
}
void main()
{    NODE * t;
    t=create();                          /* create 函数是建立二叉树的函数 */
    inorder(t);
}
```

若对如图 5-9 所示的二叉树进行中序遍历,则输出的结点序列为(a * b－c)。

3. 后序遍历

后序遍历的递归定义为：若二叉树不空,则执行如下操作。

- 按后序遍历左子树。
- 按后序遍历右子树。
- 访问根结点。

用 C 语言描述的算法如下。

```
#include <stdio.h>
typedef struct node                      /* 二叉树结点类型的定义 */
{
    char data;
    struct node * lchild;
    struct node * rchild;
}NODE;
void postorder(NODE * t)
 {/* 后序遍历二叉树的递归算法 */
  if(t!=NULL)
   { postorder(t->lchild);
     postorder(t->rchild);
     printf("%c\t",t->data);
   }
}
void main()
 { NODE * t;
   t=create();                           /* create 函数是建立二叉树的函数 */
   postorder(t);
 }
```

若对如图 5-9 所示的二叉树进行后序遍历,则输出的结点序列为(ab * c－)。

4. 建立二叉树

建立二叉树的方法有很多,这里介绍一种基于前序遍历的构造方法。算法输入的是二叉树的扩充前序序列,即在前序序列中加入空指针。若用“#”表示空指针,要建立如图 5-9 所示的二叉树,其输入的扩充前序序列为(－ * a# #b# #c# #),即可建立相应的二叉链表,用 C 语言描述的算法如下。

```
#include <stdio.h>
#include <malloc.h>
typedef struct node                      /* 二叉树结点类型的定义 */
{
    char data;
```

```
    struct node * lchild;
    struct node * rchild;
}NODE;
NODE * create()                              /* 创建二叉链表 */
{
  NODE * t;
  char  a;
  scanf("%c",&a);
  if(a= ='#')
       t=NULL;
  else
     {
       t=(NODE*)malloc(sizeof(NODE));        /* 申请根结点 * t 空间 */
       t->data=a;                            /* 将结点数据 a 放入根结点的数据域 */
       t->lchild=create();                   /* 建左子树 */
       t->rchild=create();                   /* 建右子树 */
       }
  return(t) ;
}
void main()
{
    NODE * t;
    t=create();                              /* create 函数是建立二叉树的函数 */
}
```

5. 前序遍历的非递归算法

前面讲述了二叉树遍历的递归算法，但在有些算法语言中是不允许递归调用的，所以有必要讨论二叉树遍历的非递归算法。下面就利用栈写出各种遍历二叉树的非递归算法。

使用栈实现前序遍历二叉树的基本思想是：从二叉树的根结点开始，沿左支一直走到没有左孩子的结点为止，在走的过程中访问所遇结点，并把非空右孩子进栈。当找到没有左孩子的结点时，从栈顶退出某结点的右孩子，此时该结点的左子树已遍历完，再按上述过程遍历结点的右子树。如此重复，直到二叉树中的所有结点都访问完毕为止。前序遍历二叉树的非递归算法如下。

```
#define MAX 50
#include <stdio.h>
typedef struct node                          /* 二叉树结点类型的定义 */
{
    char data;
    struct node * lchild;
    struct node * rchild;
}NODE;
void preorder1(NODE * t)
{NODE * p, * s[MAX];
 int top=0;
 p=t;
 if(t==NULL) return;
   do
   {
while(p!=NULL)
      {   printf("%c\t",p->data);
```

```
            if(p->rchild!=NULL) s[top++]=p->rchild;
            p=p->lchild;
         }
      if(top>0)
      p=s[--top];
      }
      while(p||top>0);
}
void main()
{
  NODE *t;
  t=create();                                  /*create函数是建立二叉树的函数*/
  preorder1(t);
}
```

6. 中序遍历二叉树的非递归算法

使用栈实现中序遍历二叉树的基本思想与前序遍历类似,只是在沿左支向前走的过程中将所遇结点进栈,待到遍历完左子树时,从栈顶取出结点并访问、退栈,然后再遍历右子树。中序遍历二叉树的非递归算法如下。

```
#define MAX 50
#include <stdio.h>
typedef struct node                            /*二叉树结点类型的定义*/
{
     char data;
     struct node *lchild;
     struct node *rchild;
}NODE;
void inorder1(NODE *t)
{/*中序遍历二叉树的非递归算法*/
 NODE *p,*s[MAX];
 int top=0;
 p=t;
 do
 {while(p!=NULL)
     {s[top++]=p;
      p=p->lchild;
     }
  if(top>0)
     {p=s[--top];
      printf("%c\t",p->data);
      p=p->rchild;
     }
  }while(top>0||p);
}
void main()
{
    NODE *t;
    t=create();                                /*create函数是建立二叉树的函数*/
    inorder1(t);
}
```

7. 后序遍历二叉树的非递归算法

使用栈实现后序遍历二叉树要比前序及中序遍历复杂一些。每个结点要等到遍历

左、右子树之后才得以访问，所以在遍历左、右子树之前，结点都需要进栈。当它出栈时，需要判断是从遍历左子树后的返回（即刚遍历完左子树，需要继续遍历右子树），还是从遍历右子树后的返回（即刚遍历完右子树，需要访问这个结点）。为了区分同一个结点的两次退栈，设一个标记域 tag，tag＝0 表明结点在遍历左子树时进栈，tag＝1 表明结点在遍历右子树时进栈。结点进、出栈时标记值也同时进、出栈。后序遍历二叉树的非递归算法如下。

```
#define MAX 50
#include <stdio.h>
typedef struct node                          /* 二叉树结点类型的定义 */
{
     char data;
     struct node * lchild;
     struct node * rchild;
}NODE;
void postorder(NODE * t)
{/* 后序遍历二叉树的非递归算法 */
  int s2[MAX],top=0;
  NODE   * p, * s1[MAX];
  p=t;
  do
{ while(p !=NULL)
    {s1[top]=p;s2[top++]=0;
     p=p->lchild;
}
    while((top>0)&&(s2[top-1]= =1))
    {p=s1[--top];
     printf("%c\t",p->data);
}
    if(top>0)
    {s2[top-1]=1;
     p=s1[top-1];
     p=p->rchild;
     }
    }
    while(top>0);
 }
void main()
{
    NODE * t;
    t=create();                              /* create 函数是建立二叉树的函数 */
    postorder(t);
}
```

由于遍历二叉树基本操作是访问结点，则不论按哪一种次序进行遍历，对含 n 个结点的二叉树，其时间复杂度均为 $O(n)$。

8. 由结点前序序列和中序序列构造对应的二叉树

假定二叉树中各结点数据域值均不相同，若给定一棵二叉树结点的前序序列和中序序列，就可以唯一地确定一棵二叉树。下面介绍由这两种序列构造一棵二叉树的方法。

根据前序遍历的定义，前序序列中的第一个元素必为二叉树的根结点。由中序遍历定义可知，中序序列中根结点元素恰为左、右子树的中序序列的分界点，根结点元素把中