第5章 树和二叉树

5.1 脉络梳理，本章导学

本章知识结构如图 1-5-1 所示。

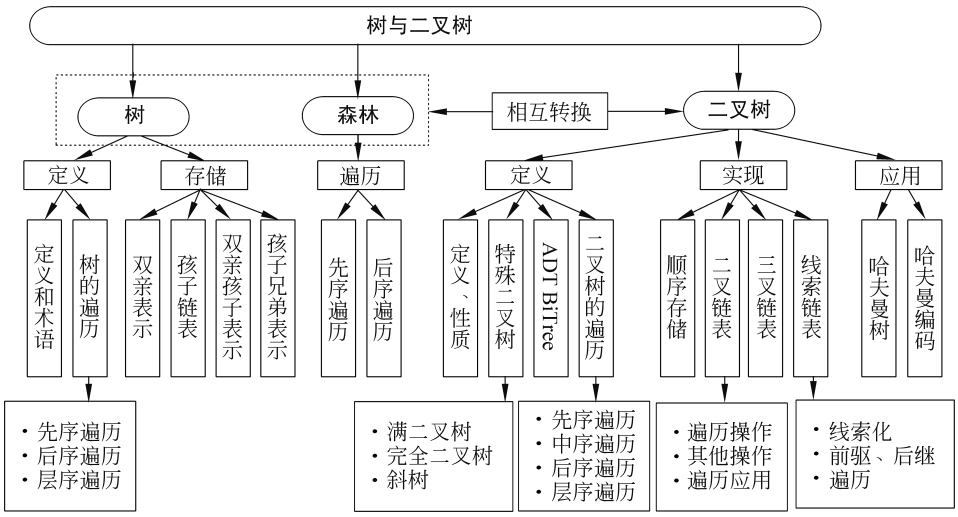

图 1-5-1　第 5 章知识结构图

本章内容涉及树形结构中的树、森林和二叉树及树/森林与二叉树之间的相互转换。

5.1.1 树

树中包括树的定义和存储两部分，其中定义包括树形结构相关术语与树的遍历。

树是 $n(n\geqslant 0)$个数据元素的有限集合。当 $n=0$ 时，称为空树；任意一棵非空树，由根结点和 $m(m>0)$个互不相交的子树组成，其中每棵子树，同样可以看成由子树的根与子树根的子树组成。

树的相关术语包括度（结点的度、树的度），各类结点（叶子、孩子、双

亲、兄弟、堂兄弟、祖先、子孙等)，路径，路径长度，层数，树的深度，有序树，无序树，森林等。

树有 3 种遍历方式：先根(序)遍历、后根(序)遍历、层序遍历。

树的存储方式有多种，主教材中主要介绍了双亲表示(顺序存储)、孩子链表(顺序与链式结合)、双亲孩子表示(顺序与链式结合)和孩子兄弟(链式存储)表示 4 种方法。

5.1.2 森林

森林是 $m(m\geqslant 0)$棵互不相交的树的集合。

森林遍历的方法有两种：先序遍历和后序遍历。先序遍历方法是按树的先序遍历方法依次遍历各棵子树。后序遍历方法是按树的后序遍历方法依次遍历森林的各棵子树。

5.1.3 二叉树

二叉树内容包含 3 个方面：定义、实现和应用。

1. 二叉树的定义

二叉树或为空，或由根与 $m(m=0$、1 或 2)棵分左、右的子树构成，每棵子树也是二叉树。

二叉树有以下 5 个重要的性质。

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

性质 2. 在一棵深度为 k 的二叉树中，最多具有 2^k-1 个结点($k\geqslant 0$)，最少有 k 个结点。

性质 3. 对于一棵非空的二叉树，如果度为 0 结点数为 n_0、度为 2 的结点数为 n_2，则有：$n_0=n_2+1$。

性质 4. 具有 n 个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor+1$ 或$\lceil \log_2(n+1) \rceil$。

性质 5. 完全二叉树中，如果从上至下且从左至右进行 1 至 n 的编号，第 i 个结点的双亲是$\lfloor i/2 \rfloor$个结点，左和右孩子是第 $2i(\leqslant n)$和第 $2i+1(\leqslant n)$个结点。

特殊形态的二叉树有满二叉树、完全二叉树、斜树等，它们因形态特殊而具有特殊的性质。

二叉树抽象的数据类型给出二叉树的逻辑结构(数据元素和数据元素之间的关系)及其上基本操作的定义。遍历是其中重要的操作，二叉树有 4 种遍历方式：先序(根)遍历、中序(根)遍历、后序(根)遍历和层序遍历。

以先左后右为例，先序遍历次序是根、左、右；中序遍历次序是左、根、右；后序遍历次序是左、右、根；层序遍历从上至下、从左往右逐层遍历。

2. 二叉树的实现

二叉树的实现中包含 3 方面的内容：二叉树的存储设计、二叉链表存储的二叉树的操作算法和线索二叉树。

1) 二叉树的存储设计

二叉树的存储方法有顺序存储、二叉链表存储、三叉链表存储等。

二叉树的顺序存储是按层序从左往右，依次把二叉树的数据元素存储到一组连续的

内存中，根据二叉树的性质 5，用数据元素的下标映射元素彼此之间的双亲、孩子关系。

二叉树的二叉链表存储是最常用的一种二叉树链式存储方式。结点定义如下：

```
template <class DT>
struct BTNode
{  DT data;                    //数据域
   BTNode * lchild;            //左孩子指针
   BTNode * rchild;            //右孩子指针
};
```

二叉树的三叉链表存储是在二叉链表存储中加了一个指向双亲的指针。结点定义如下：

```
template <class DT>
struct BTPNode
{  DT data;                //数据域
   BTPNode * parent;       //双亲指针
   BTPNode * lchild;       //左孩子指针
   BTPNode * rchild;       //右孩子指针
};
```

2）二叉链表存储的二叉树的操作算法

二叉树采用二叉链表存储，操作算法如下。

（1）先序、中序、后序遍历操作的递归算法（算法 5.1、算法 5.2、算法 5.3）。

（2）先/中/后根遍历递归操作非递归算法（算法 5.5、算法 5.6、算法 5.7），借用栈模拟遍历中的回溯。

（3）层序遍历操作算法（算法 5.4），借用队列实现。

（4）创建二叉树（算法 5.8）、销毁二叉树（算法 5.9）、结点查询（算法 5.10）、求二叉树的深度（算法 5.11）和结点计数（算法 5.12）等。

二叉树的操作中，遍历操作是基础，其余许多操作都是基于遍历思想的。

由遍历序列确定二叉树：已知中序遍历序列和先序遍历序列、后序遍历序列、层序遍历序列中的任一个可以唯一确定一棵二叉树。

3）线索二叉树

利用二叉树二叉链表存储中的空指针标识遍历中的前驱和后继，形成了线索二叉树。线索二叉树相关的术语与概念有线索、前驱线索、后继线索；线索化、前驱线索化、后继线索化、全序线索化；先序线索化、中序线索化、后序线索化、层序线索化；先序线索二叉树、中序线索二叉树、后序线索二叉树、层序线索二叉树等。

线索二叉树的结点定义如下：

```
template <class DT>
struct BiThrNode
{
    DT data;                   //数据域
    int lflag;                 //左标志域：0 表示 lchild 非线索，1 表示 lchild 线索
```

```
    int rflag;                  //右标志域：0表示 rchild非线索,1表示 rchild线索
    BiThrNode * lchild;         //左指针域
    BiThrNode * rchild;         //右指针域
};
```

线索信息给某些遍历带来方便。例如，基于线索信息，先序线索二叉树上的先序遍历无须回溯，中序线索二叉树上的中序遍历无须回溯。

3. 二叉树的应用

哈夫曼树和哈夫曼编码是二叉树的典型应用。

1）哈夫曼树/最优二叉树

在所有含 n 个叶结点、并带相同权值的 m 叉树中，必存在一棵树的带权路径长度最小的树，称为“最优树”。当 $m=2$ 时，为最优二叉树。哈夫曼给出了最优二叉树的构造方法，最优二叉树也称为哈夫曼树。

2）哈夫曼编码

规定哈夫曼树的左分支为 0，右分支为 1。从根结点到每个叶结点所经过的分支对应的 0 和 1 组成的序列便是该结点对应的字符编码。

5.1.4 树/森林与二叉树的相互转换

1. 树与二叉树的相互转换

树与二叉树的相互转换通过加线、去线与位置调整实现。

2. 森林与二叉树的相互转换

基于树与二叉树的相互转换，并将森林中各棵树的根看作兄弟，可以实现森林与二叉树的相互转换。

3. 树、森林、二叉树的遍历序列关系

树的遍历序列与对应的二叉树的遍历序列之间具有以下对应关系：树的先序遍历≌二叉树的先序遍历、树的后序遍历≌二叉树的中序遍历。

森林的遍历序列与对应的二叉树的遍历序列之间具有以下对应关系：森林的先序遍历≌二叉树的先序遍历、森林的中序遍历 ≌ 二叉树的中序遍历。

5.2 拨云见日，谜点解析

5.2.1 树与二叉树

树和二叉树都是树结构，但二叉树不是树的特例。

（1）定义不同。在“两者都是由 $n(n\geqslant 0)$ 个结点的有限集合”的相同点下，$n>1$ 时，树是由“根和 m 个互不相交的子树组成”，二叉树是由“根和两棵互不相交，分别称为根的左子树和右子树组成”。

（2）二叉树不是度为 2 的树。度为 2 的树，至少有一个度为 2 的结点，且不可能是空树。二叉树可以为空，二叉树的度可以是 0、1 或 2。

(3) 孩子顺序不同。有序树结点的孩子编号为第 1,第 2,以此类推,从左往右或从右往左依次编号,二叉树结点的孩子分左右,即使只有一个孩子结点,也分左、右。

5.2.2 遍历过程

二叉树通过遍历得到一个线性序列,因此,遍历可以看作某个规则下将非线性结构线性化的一个过程。线性结构的特点是有唯一前驱和后继。

下面以图 1-5-2 所示二叉树 BT1 为例,说明各遍历线性序列生成规律。

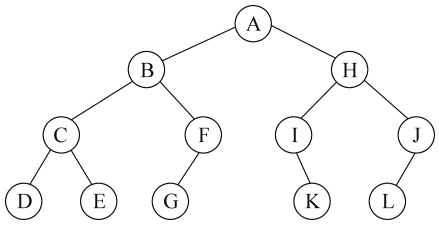

图 1-5-2　遍历示例二叉树 BT1

1. 先序遍历序列

由先序遍历规则可知,先序遍历的第一个结点是树根。

对于其他任一结点 p:

(1) 如果 p 有左孩子,p 的后继为 p->lchild 所指,如 A→B→C→D、F→G、H→I、J→L。

(2) 如果 p 无左孩子有右孩子,p 的后继为 p->rchild 所指,如 I→K。

(3) 如果 p 是叶结点、是双亲的左孩子且双亲有右孩子,p 的后继为双亲的右孩子,如 D→E。

(4) 如果 p 是叶结点、是双亲的左孩子且双亲无右孩子,或 p 是叶结点、是双亲的右孩子,p 的后继是双亲的双亲的右孩子;如果双亲的双亲无右孩子,继续往祖先方向推,如 E→F、G→H、K→J。

最后得到先序遍历序列:A B C D E F G H I K J L。

2. 中序遍历序列

由中序遍历规则可知:D C E B G F A I K H L J。

中序遍历的第一个结点是从根出发左行至极左结点,即结点 D。对于其他任一结点 p:

(1) 如果 p 无右孩子且是双亲的左孩子,p 的后继为其双亲,如 D→C、G→F、L→J。

(2) 如果 p 有右孩子,以 p 的右孩子为出发点的极左结点为 p 有后继,如 C→E、B→G、A→I、I→K、H→L。

(3) 如果 p 无右孩子且是双亲的右孩子,p 的后继为其祖先上第一个是双亲左孩子的祖先结点,如 E→B、F→A、K→H;根的右子树上极右结点为中序遍历的最后一个结点,无后继,如 J。

最终得到中序遍历序列为 D C E B G F A I K H L J。

3. 后序遍历序列

由后序遍历规则可知：

后序遍历的第一个结点是左子树上极左再到极右结点，即从根出发左行至极左后右行至极右，如 D。对于其他任一结点 p：

(1) 如果 p 是双亲的左孩子且双亲有右孩子，转至双亲的右孩子，p 的后继为以此为出发点的左行至极左后右行至极右结点，如 D→E、C→G、B→K、I→L。

(2) 如果 p 是双亲的左孩子且双亲无右孩子，其后继为双亲，如 G→F、L→J。

(3) 如果 p 是双亲的右孩子，其后继为双亲，如 E→C、F→B、K→I、J→H、H→A。

最终得到后序遍历序列为 D E C G F B K I L J H A。

5.2.3 二叉树的先序、中序、后序遍历递归操作

二叉树的先序、中序和后序遍历算法描述如下。

```
//先序遍历
0  template <class DT>
1  void PreOrderBiTree(BTNode<DT> * bt)
2  {  if (bt !=NULL)
3     {  cout<<bt->data;
4        PreOrderBiTree(bt->lchild);
5        PreOrderBiTree(bt->rchild);
6     }
7  }
//中序遍历
0  template <class DT>
1  void InOrderBiTree(BTNode<DT> * bt)
2  {  if (bt !=NULL)
3     {  InOrderBiTree(bt->lchild);
4        cout<<bt->data;
5        InOrderBiTree(bt->rchild);
6     }
7  }
//后序遍历
0  template <class DT>
1  void PostOrderBiTree(BTNode<DT> * bt)
2  {  if(bt !=NULL)
3     {  PostOrderBiTree(bt->lchild);
4        PostOrderBiTree(bt->rchild);
5        cout<<bt->data;
6     }
7  }
```

3 个算法的描述都有 8 行语句，关键语句是 3、4、5，只是顺序上不同。主要区别是访

问语句“cout<<bt->data”的位置。先序遍历算法中，它位于递归遍历左、右子树之前；中序遍历中位于递归遍历左、右子树之间；后序遍历中位于递归遍历左、右子树之后。分别与遍历中 DLR、LDR、LRD 的根的位置相一致。

遍历输出中，对结点的访问的输出结点的值，把访问改成其他操作，即可通过遍历解决其他问题，如创建、销毁、查询、结点计数等。

5.2.4　二叉树创建

二叉树的创建方法有以下 3 种：算法 5.8、基于遍历序列和顺序存储序列。

1. 算法 5.8

主教材中的算法 5.8 给出的二叉树的创建是基于先序遍历思想。先创建根结点，然后递归创建根的左子树，再递归创建根的右子树。将先序遍历递归算法中的访问语句“cout<<bt->data”改成结点创建，即可形成二叉树的创建算法。算法根据用户输入创建二叉树，#表示空。算法描述如下。

```
0  template<class DT>
1  void CreateBiTree(BiTree<DT>&bt)
2  {  cin>>ch;                              //输入结点值
3     if(ch=="#")                           //空值
4        bt=NULL;
5     else                                  //非空值
6     {  bt=new BiTNode;                    //创建结点
7        bt->data=ch;
8        CreateBiTree(bt->lchild);          //递归创建左子树
9        CreateBiTree(bt->rchild);          //递归创建右子树
10    }
11 }
```

需要注意的是，算法虽然是基于先序遍历思想，但是输入的结点序列不是先序遍历序列。因为对每一个非空结点，执行 else 分支语句 6、7、8、9，其中包括 3 个工作：(1)创建新结点；(2)递归创建该结点的左子树；(3)递归创建该结点的右子树。因此，对每一个非空结点，要给出其左、右孩子的结点值，如果没有左、右孩子，用空值#代替。

为了避免出错，可以先把要创建的二叉树的每一个非空结点补全其左、右孩子，结点值用#表示(如图 1-5-3 所示)，然后以被补全后的二叉树先序遍历序列顺序输入。图 1-5-3 所示二叉树输入序列为 AB#DF###CE#G###。

2. 基于遍历序列

已知二叉树的中序遍历序列和先序、后序、层序遍历序列的任一种，可以唯一确定二叉树。操作方法如下。

Step 1. 在先序遍历或后序遍历或层序遍历序列中，获知树或子树根，即：(1)先序遍历序列的第一个结点；(2)后序遍历序列的最后一个结点；(3)层序遍历序列的第一个结点。

Step 2. 在中序遍历序列中，找到树或子树“根”，将中序遍历序列以“根”为分界线分

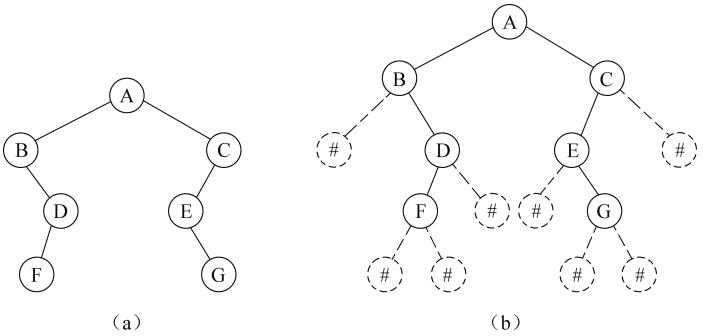

图 1-5-3 创建二叉树示例

为左、右两个子序列。它们分别为树或子树“根”的左、右子树的中序遍历序列。

Step 3. 由序列集合相等的特性在先序或后序或层序遍历序列找到左、右子树的先序或后序或层序遍历序列。

重复 Step 1～Step 3 确定各棵子树，直至叶结点。

上述操作的关键是先序遍历序列或后序遍历序列或层序遍历序列中能确定“根”，在中序遍历序列中能根据“根”划分左子树序列集合和右子树序列集合。如果给出的是除中序遍历序列外的任意两种，如先序遍历序列和后序遍历序列，均无法唯一确定二叉树。

对于完全二叉树，给出一个遍历序列即可唯一确定该树。序列中元素个数可以确定完全二叉树的形态，序列中元素顺序可以确定每个结点的值。例如，一棵完全二叉树先序遍历序列为 ABCDEFGHIJ，则此树有 10 个结点。10 个结点的完全二叉树树形如图 1-5-4(a)所示，先序遍历序列可确定完全二叉树如图 1-5-4(b)所示。

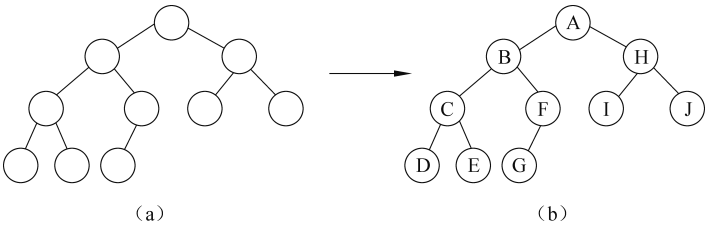

图 1-5-4 遍历序列确定完全二叉树

3. 顺序存储序列

如果采用顺序存储，可以按照完全二叉树的层序序列创建二叉树。需要注意的是，如果是非完全二叉树，需补全缺失的结点。如图 1-5-5(a)所示的二叉树，创建序列为 ABC＃DE＃＃＃F＃＃G。

5.2.5 二叉树问题的递归分析

二叉树由根、根的左子树和根的右子树 3 部分组成，以相同的方式划分子树。因此，对于二叉树的“大问题”，可以变成根、左子树和右子树的“小问题”。由“小问题”与“大问题”的关系，得到递归求解思路。

【例】 算法 5.11，计算二叉树的深度。求二叉树深度等于递归求左子树深度(设为 m)、右子树深度(设为 n)，树的深度为 $\max(m,n)+1$。递归计算定义如下：

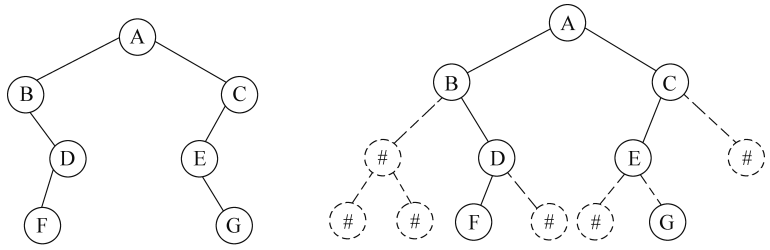

图 1-5-5　顺序存储结点补全示例

$$\begin{cases} \text{depth(BiTreeT)}=0 & \text{T}=\text{NULL} \\ \text{Max}\{\text{depth(T-}>\text{lchild)},\text{depth(T-}>\text{rchild)}\}+1 & \text{其他} \end{cases}$$

求二叉树深度算法，算法描述如下：

```
0  template <class DT>
1  int Depth(BiNode<DT> * bt)
2  {  if(bt==NULL)                          //空树,深度 0
3        return 0;
4     else                                  //非空树
5     {  m=Depth(bt->lchild);               //递归计算左子树深度
6        n=Depth(bt->rchild);               //递归计算右子树深度
7        if(m>n)                            //左子树深
8           return m+1;
9        else                               //右子树深
10          return n+1;
11    }
12 }
```

【例】 算法 5.12，求结点数。二叉树的结点数＝左子树结点数＋右子树结点数＋1，求结点数问题可以变成递归求左、右子树的结点数。结点计数的递归定义如下：

$$\begin{cases} \text{NodeCount(BiTreeT)}=0 & \text{T}=\text{NULL} \\ \text{NodeCount(T-}>\text{lchild)}+\text{NodeCount(T-}>\text{rchild)}+1 & \text{其他} \end{cases}$$

计算二叉树结点个数的算法，算法描述如下。

```
0  template<class DT>
1  int NodeCount(BiTree<DT> bt)
2  {
3     if(bt==NULL)                 //空二叉树,结点个数为 0
4        return 0;
5     else                         //返回左、右子树结点个数和加 1
6        return NodeCount(bt->lchild)+NodeCount(bt->rchild)+1
7  }
```

复制二叉树，可以转变成复制根结点，递归复制左子树和递归复制右子树。

5.2.6 二叉树的先序、中序、后序遍历中结点访问次序

先序、中序和后序遍历分别按照 DLR、LDR 或 LRD 访问二叉树中的所有结点，使得每个结点被访问一次且仅被访问一次。如果绕着二叉树走一遍（见图 1-5-6），会发现：叶结点被经过一次，只有左或右孩子的结点被经过 2 次，左、右孩子双全的结点被经过 3 次。不同的遍历规则，决定了结点在第几次经过时被访问。

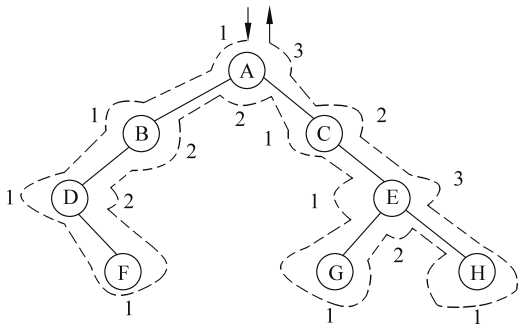

图 1-5-6 遍历路径

先序遍历中，每个结点第 1 次经过时，即被访问。

中序遍历中，一个结点如果没有左孩子，则第 1 次经过时被访问，否则是在从左子树回溯到双亲时（即第 2 次经过）被访问。

后序遍历中，叶结点首次经过时被访问；只有左孩子的结点是从左子树回溯到双亲时（即第 2 次经过）被访问；只有右孩子结点时，是从右子树回溯到双亲时（即第 2 次经过）被访问；左、右孩子双全时，左子树遍历完回溯到双亲转双亲的右子树，遍历完右子树回溯到双亲时（即第 3 次经过）被访问。

5.2.7 先序、中序和后序遍历非递归操作中的入栈结点

二叉树的二叉链表存储中，当选择根的左子树后，只有通过回溯才能得到双亲或右子树结点。因此，按 DLR、LDR、LRD 不同顺序进行的遍历中不可避免地存在回溯。非递归遍历中需用栈保存结点实现回溯。

1. 先序遍历的入栈结点

先序遍历的第一点是树根，如果有左孩子，下一个结点为左孩子，依次继续下去；如果没有左孩子，需回溯到双亲，转向右孩子。因此，非递归先序遍历中，从根一路左行中，需将被访问的结点进栈或其右孩子进栈。如果进栈的是被访问结点，以出栈结点的右孩子为新的遍历起点；如果进栈的是被访问结点的右孩子，以出栈结点作为新的遍历起点。

2. 中序遍历的入栈结点

中序遍历中先访问左子树，然后根，最后右子树。因此，非递归中序遍历中，入栈的是从根开始左行沿途经过的结点。如果被访问结点是双亲的左孩子，通过出栈回溯到该结点的双亲，访问它，然后将它的右孩子作为继续遍历的新起点；如果被访问的是双亲的右