第5章

二叉树

通过前面的章节，我们已经了解了一些基本的数据结构。根据其实现方式，这些数据结构大致可以分为两种类型：基于数组的实现与基于链表的实现。正如我们已经看到的，就其效率而言，二者各有长短。具体来说，前一实现方式允许我们通过下标或秩，在常数的时间内找到目标对象；然而，一旦需要对这类结构进行修改，那么无论是插入还是删除，都需要耗费线性的时间。反过来，后一实现方式允许我们借助引用或位置对象，在常数的时间内插入或删除元素；但是为了找出居于特定次序的元素，我们却不得不花费线性的时间，对整个结构进行遍历查找。能否将这两类结构的优点结合起来，并回避其不足呢？本章所讨论的树结构，将正面回答这一问题。

在此前介绍的这些结构中，元素之间都存在一个自然的线性次序，故它们都属于所谓的线性结构（linear structure）。树则不然，其中的元素之间并不存在天然的直接后继或直接前驱关系。不过，正如我们马上就要看到的，只要附加某种约束（比如遍历），也可以在树中的元素之间确定某种线性次序，因此树属于半线性结构（semi-linear structure）。

无论如何，随着从线性结构转入树结构，我们的思维方式也将有个飞跃；相应地，算法设计的策略与模式也会因此有所变化，许多基本的算法也将得以更加高效地实现。以第7章和第8章将要介绍的平衡二叉搜索树为例，若其中包含n个元素，则每次查找、更新、插入或删除操作都可在$O(\log n)$时间内完成——相对于线性结构，几乎提高了一个线性因子（习题[1-9]）。

树结构有着不计其数的变种，在算法理论以及实际应用中，它们都扮演着最为关键的角色。之所以如此，是因得益于其独特而又普适的逻辑结构。树是一种分层结构，而层次化这一特征几乎蕴含于所有事物及其联系当中，成为其本质属性之一。从文件系统、互联网域名系统和数据库系统，一直到地球生态系统乃至人类社会系统，层次化特征以及层次结构均无所不在。

有趣的是，作为树的特例，二叉树实际上并不失其一般性。本章将指出，无论就逻辑结构或算法功能而言，任何有根有序的多叉树，都可等价地转化并实现为二叉树。因此，本章讲解的重点也将放在二叉树上。我们将以通讯编码算法的实现这一应用实例作为线索贯穿全章。

§5.1 二叉树及其表示

5.1.1 树

■ **有根树**

从图论的角度看，树等价于连通无环图。因此与一般的图相同，树也由一组顶点（vertex）以及联接与其间的若干条边（edge）组成。在计算机科学中，往往还会在此基础上，再指定某一特定顶点，并称之为根（root）。在指定根节点之后，我们也称之为有根树（rooted tree）。此时，从程序实现的角度，我们也更多地将顶点称作节点（node）。

■ 深度与层次

由树的连通性，每一节点与根之间都有一条路径相联；而根据树的无环性，由根通往每个节点的路径必然唯一。因此如图5.1所示，沿每个节点v到根r的唯一通路所经过边的数目，称作v的深度（depth），记作depth(v)。依据深度排序，可对所有节点做分层归类。特别地，约定根节点的深度depth(r) = 0，故属于第0层。

■ 祖先、后代与子树

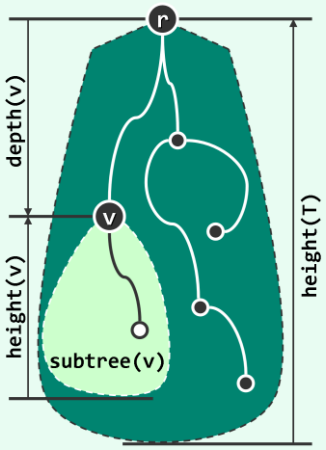


图5.1 有根树的逻辑结构

任一节点v在通往树根沿途所经过的每个节点都是其祖先（ancestor），v是它们的后代（descendant）。特别地，v的祖先/后代包括其本身，而v本身以外的祖先/后代称作真祖先（proper ancestor）/真后代（proper descendant）。

节点v历代祖先的层次，自下而上以1为单位逐层递减；在每一层次上，v的祖先至多一个。特别地，若节点u是v的祖先且恰好比v高出一层，则称u是v的父亲（parent），v是u的孩子（child）。

v的孩子总数，称作其度数或度（degree），记作deg(v)。无孩子的节点称作叶节点（leaf），包括根在内的其余节点皆为内部节点（internal node）。

v所有的后代及其之间的联边称作子树（subtree），记作subtree(v)。在不致歧义时，我们往往不再严格区分节点（v）及以之为根的子树（subtree(v)）。

■ 高度

树T中所有节点深度的最大值称作该树的高度（height），记作height(T)。

不难理解，树的高度总是由其中某一叶节点的深度确定的。特别地，本书约定，仅含单个节点的树高度为0，空树高度为-1。

推而广之，任一节点v所对应子树subtree(v)的高度，亦称作该节点的高度，记作height(v)。特别地，全树的高度亦即其根节点r的高度，height(T) = height(r)。

5.1.2 二叉树

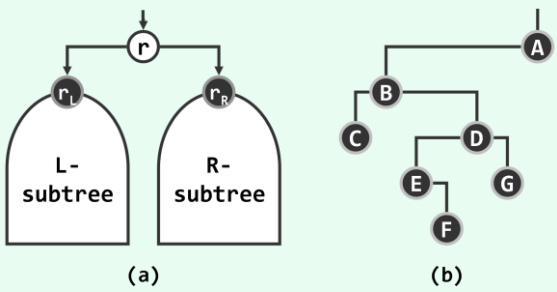


图5.2 二叉树：(a)逻辑结构；(b)实例

如图5.2所示，二叉树（binary tree）中每个节点的度数均不超过2。

因此在二叉树中，同一父节点的孩子都可以左、右相互区分——此时，亦称作有序二叉树（ordered binary tree）。本书所提到的二叉树，默认地都是有序的。

特别地，不含一度节点的二叉树称作真二叉树（proper binary tree）（习题[5-2]）。

5.1.3 多叉树

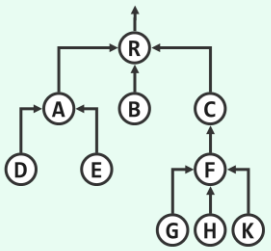
一般地，树中各节点的孩子数目并不确定。每个节点的孩子均不超过k个的有根树，称作k叉树（k-ary tree）。本节将就此类树结构的表示与实现方法做一简要介绍。

■ 父节点

由如图5.3(a)实例不难看出，在多叉树中，根节点以外的任一节点有且仅有一个父节点。

因此可如图5.3(b)所示，将各节点组织为向量或列表，其中每个元素除保存节点本身的信息（data）外，还需要保存父节点（parent）的秩或位置。可为树根指定一个虚构的父节点-1或NULL，以便统一判断。

如此，所有向量或列表所占的空间总量为$O(n)$，线性正比于节点总数n。时间方面，仅需常数时间，即可确定任一节点的父节点；但反过来，孩子节点的查找却不得不花费$O(n)$时间访遍所有节点。

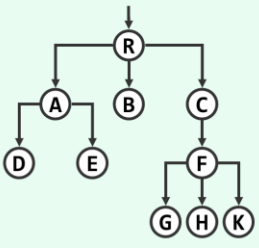


(a)

	data	parent
0	R	-1
1	A	0
2	B	0
3	C	0
4	D	1
5	E	1
6	F	3
7	G	6
8	H	6
9	K	6

(b)

图5.3 多叉树的“父节点”表示法

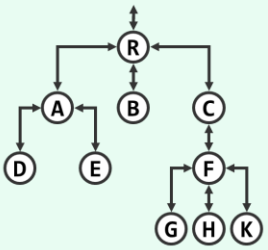


(a)

	data	children
0	A	3 → 5 → ^
1	B	^
2	C	6 → ^
3	D	^
4	R	0 → 1 → 2 → ^
5	E	^
6	F	7 → 8 → 9 → ^
7	G	^
8	H	^
9	K	^

(b)

图5.4 多叉树的“孩子节点”表示法



(a)

	data	parent	children
0	A	4	3 → 5 → ^
1	B	4	^
2	C	4	6 → ^
3	D	0	^
4	R	-1	0 → 1 → 2 → ^
5	E	0	^
6	F	2	7 → 8 → 9 → ^
7	G	6	^
8	H	6	^
9	K	6	^

(b)

图5.5 多叉树的“父节点 + 孩子节点”表示法

■ 孩子节点

若注重孩子节点的快速定位，可如图5.4所示，令各节点将其所有的孩子组织为一个向量或列表。如此，对于拥有r个孩子的节点，可在$O(r + 1)$时间内列举出其所有的孩子。

■ 父节点 + 孩子节点

以上父节点表示法和孩子节点表示法各有所长，但也各有所短。为综合二者的优势，消除缺点，可如图5.5所示令各节点既记录父节点，同时也维护一个序列以保存所有孩子。

尽管如此可以高效地兼顾对父节点和孩子的定位，但在节点插入与删除操作频繁的场合，为动态地维护和更新树的拓扑结构，不得不反复地遍历和调整一些节点所对应的孩子序列。然而，向量和列表等线性结构的此类操作都需耗费大量时间，势必影响到整体的效率。

■ 有序多叉树 = 二叉树

解决上述难题的方法之一，就是采用支持高效动态调整的二叉树结构。为此，必须首先建立起从多叉树到二叉树的某种转换关系，并使得在此转换的意义下，任一多叉树都等价于某棵二叉树。当然，为了保证作为多叉树特例的二叉树有足够的能力表示任何一棵多叉树，我们只需给多叉树增加一项约束条件——同一节点的所有孩子之间必须具有某一线性次序。

仿照有序二叉树的定义，凡符合这一条件的多叉树也称作有序树（ordered tree）。幸运的是，这一附加条件在实际应用问题中往往自然满足。以互联网域名系统所对应的多叉树为例，其中同一域名下的分支通常即按照字典序排列。

■ 长子 + 兄弟

由图5.6(a)的实例可见，有序多叉树中任一非叶节点都有唯一的“长子”，而且从该“长子”出发，可按照预先约定或指定的次序遍历所有孩子节点。因此可如图(b)所示，为每个节点设置两个指针，分别指向其“长子”和下一“兄弟”。

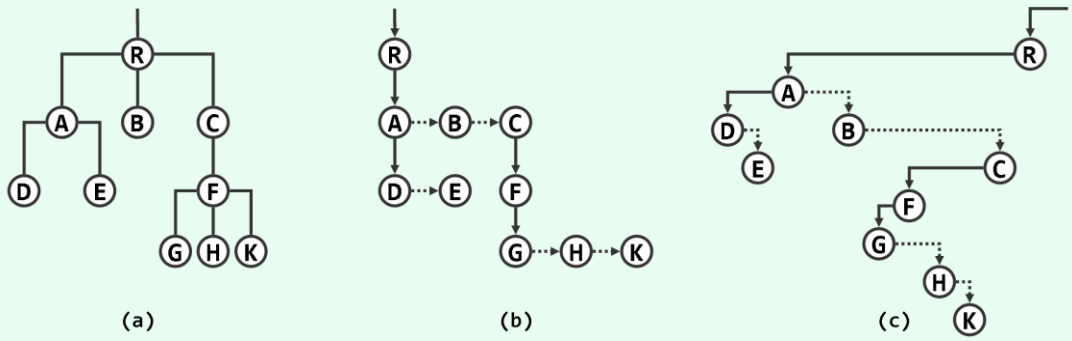


图5.6 多叉树的“长子 + 兄弟”表示法（(b)中长子和兄弟指针，分别以垂直实线和水平虚线示意）

现在，若将这两个指针分别与二叉树节点的左、右孩子指针统一对应起来，则可进一步地将原有序多叉树转换为如图(c)所示的常规二叉树。

在这里，一个饶有趣味的现象出现了：尽管二叉树只是多叉树的一个子集，但其对应用问题的描述与刻画能力绝不低于后者。实际上以下我们还将进一步发现，即便是就计算效率而言，二叉树也并不逊色于一般意义上的树。反过来，得益于其定义的简洁性以及结构的规范性，二叉树所支撑的算法往往可以更好地得到描述，更加简捷地得到实现。二叉树的身影几乎出现在所有的应用领域当中，这也是一个重要的原因。

§5.2 编码树

本章将以通讯编码算法的实现作为二叉树的应用实例。通讯理论中的一个基本问题是，如何在尽可能低的成本下，以尽可能高的速度，尽可能忠实地实现信息在空间和时间上的复制与转移。在现代通讯技术中，无论采用电、磁、光或其它任何形式，在信道上传递的信息大多以二进制比特的形式表示和存在，而每一个具体的编码方案都对应于一棵二叉编码树。

5.2.1 二进制编码

在加载到信道上之前，信息被转换为二进制形式的过程称作编码（encoding）；反之，经信道抵达目标后再由二进制编码恢复原始信息的过程称作解码（decoding）。

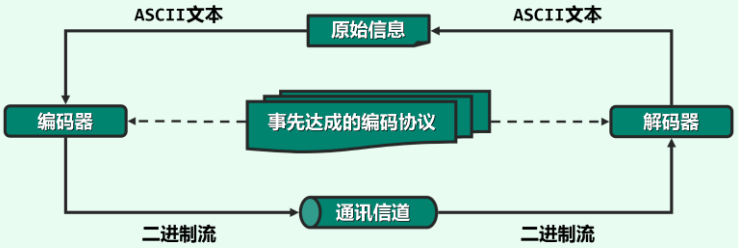


图5.7 完整的通讯过程由预处理、编码和解码阶段组成

如图5.7所示，编码和解码的任务分别由发送方和接收方分别独立完成，故在开始通讯之前，双方应已经以某种形式，就编码规则达成过共同的约定或协议。

■ 生成编码表

原始信息的基本组成单位称作字符，它们都来自于某一特定的有限集合Σ，也称作字符集（alphabet）。而以二进制形式承载的信息，都可表示为来自编码表Γ = { 0, 1 }*的某一特定二进制串。从这个角度理解，每一编码表都是从字符集Σ到编码表Γ的一个单射，编码就是对信息文本中各字符逐个实施这一映射的过程，而解码则是逆向映射的过程。

表5.1 Σ = { 'A', 'E', 'G', 'M', 'S' }的一份二进制编码表

字符	A	E	G	M	S
编码	00	01	10	110	111

表5.1即为二进制编码表的实例。编码表一旦制定，信息的发送方与接收方之间也就建立起了一个约定与默契，从而使得独立的编码与解码成为可能。

■ 二进制编码

现在，所谓编码就是对于任意给定的文本，通过查阅编码表逐一将其中的字符转译为二进制编码，这些编码依次串接起来即得到了全文的编码。比如若待编码文本为"MESSAGE"，则根据由表5.1确定的编码方案，对应的二进制编码串应为"$_{110}^{M}{}_{01}^{E}{}_{111}^{S}{}_{111}^{S}{}_{00}^{A}{}_{10}^{G}{}_{01}^{E}$"[①]。

表5.2 二进制解码过程

二进制编码	当前匹配字符	解出原文
110011111111001001	M	M
01111111001001	E	ME
111 11001001	S	MES
111001001	S	MESS
01001	A	MESSA
1001	G	MESSAG
01	E	MESSAGE

① 这里对各比特位做了适当的上下移位，以便读者区分各字符编码串的范围；在实际编码中，它们并无“高度”的区别

■ 二进制解码

由编码器生成的二进制流经信道送达之后，接收方可以按照事先约定的编码表（表5.1），依次扫描各比特位，并经匹配逐一转译出各字符，从而最终恢复出原始的文本。

仍以二进制编码串"$_{110}{}^{01}{}_{111}{}^{111}{}_{00}{}^{10}{}_{01}$"为例，其解码过程如表5.2所示。

■ 解码歧义

请注意，编码方案确定之后，尽管编码结果必然确定，但解码过程和结果却不见得唯一。

表5.3 Σ = { 'A', 'E', 'G', 'M', 'S' }的另一份编码表

字　符	A	E	G	M	S
编码	00	01	10	11	111

比如，上述字符集Σ的另一编码方案如表5.3所示，与表5.1的差异在于，字符'M'的编码由"110"改为"11"。此时，原始文本"MESSAGE"经编码得到二进制编码串"$_{11}{}^{01}{}_{111}{}^{111}{}_{00}{}^{10}{}_{01}$"，但如表5.4左侧和右侧所示，解码方法却至少有两种。

表5.4 按照表5.3“确定”的编码协议，可能有多种解码结果

二进制编码	当前匹配字符	解出原文
1101111111001001	M	M
01111111001001	E	ME
111111001001	S	MES
111001001	S	MESS
001001	A	MESSA
1001	G	MESSAG
01	E	MESSAGE

二进制编码	当前匹配字符	解出原文
1101111111001001	M	M
01111111001001	E	ME
111111001001	M	MEM
1111001001	M	MEMM
11001001	M	MEMMM
001001	A	MEMMMA
1001	G	MEMMMAG
01	E	MEMMMAGE

进一步推敲之后不难发现，按照这份编码表，有时甚至还会出现无法完成解码的情况。

■ 前缀无歧义编码

解码过程之所以会出现上述歧义甚至错误，根源在于编码表制订不当。这里的解码算法采用的是，按顺序对信息比特流做子串匹配的策略，因此为消除匹配的歧义性，任何两个原始字符所对应的二进制编码串，相互都不得是前缀。比如在表5.3中，字符'M'的编码（"11"）即为字符'S'的编码（"111"）的前缀，于是编码串"111111"既可以解释为：

"SS" = "$_{111}{}^{111}$"

也可以解释为

"MMM" = "$_{11}{}^{11}{}_{11}$"

反过来，只要各字符的编码串互不为前缀，则即便出现无法解码的错误，也绝对不致歧义。这类编码方案即所谓的“前缀无歧义编码”（prefix-free code），简称PFC编码。此类编码算法，可以明确地将二进制编码串，分割为一系列与各原始字符对应的片段，从而实现无歧义的解码。得益于这一特点，此类算法在整个解码过程中，对信息比特流的扫描不必回溯。

那么，PFC编码的以上特点，可否直观解释？从算法角度看，PFC编码与解码过程，又该如何准确描述？从数据结构角度看，这些过程的实现，需要借助哪些功能接口？支持这些接口的数据结构，又该如何高效率地实现？以下以二叉树结构为模型，逐步解答这些疑问。

5.2.2 二叉编码树

■ 根通路与节点编码

任一编码方案都可描述为一棵二叉树：从根节点出发，每次向左（右）都对应于一个0（1）比特位。于是如图5.8所示，由从根节点到每个节点的唯一通路，可以为各节点v赋予一个互异的二进制串，称作根通路串（root path string），记作rps(v)。当然，|rps(v)| = depth(v)就是v的深度。

若将Σ中的字符分别映射至二叉树的节点，则字符x的二进制编码串即可取作rps(v(x))。以下，在不致引起混淆的前提下，不再区分字符x和与之对应的节点v(x)。于是，rps(v(x))可简记作rps(x)；depth(v(x))可简记作depth(x)。

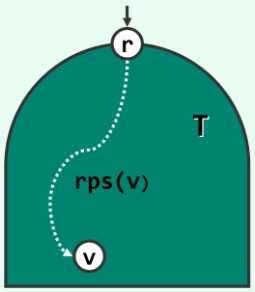


图5.8 二叉树中每个节点都由根通路串唯一确定

■ PFC编码树

仍以字符集Σ = { 'A', 'E', 'G', 'M', 'S' }为例，表5.1、表5.3所定义的编码方案分别对应于如图5.9左、右所示的二叉编码树。

易见，rps(u)是rps(v)的前缀，当且仅当节点u是v的祖先，故表5.3中编码方案导致解码歧义的根源在于，在其编码树（图5.9(b)）中字符'M'是'S'的父亲。

反之，只要所有字符都对应于叶节点，歧义现象即自然消除——这也是实现PFC编码的简明策略。比如，图5.9(a)即为一种可行的PFC编码方案。

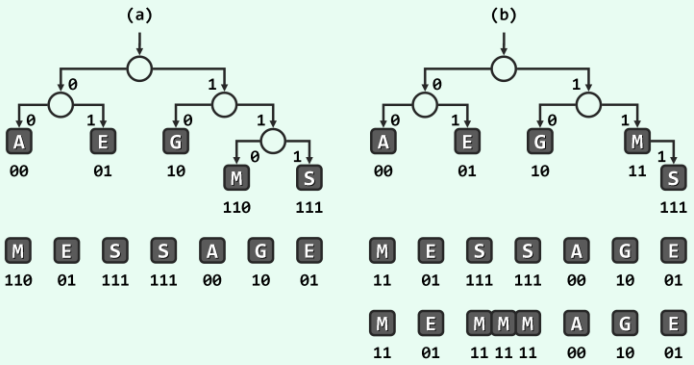


图5.9 Σ = { 'A','E','G','M','S' }两种编码方案对应的二叉编码树

■ 基于PFC编码树的解码

反过来，依据PFC编码树可便捷地完成编码串的解码。依然以图5.9(a)中编码树为例，设对编码串"$_{110}{}^{01}{}_{111}{}^{111}{}_{00}{}^{10}{}_{01}$"解码。从前向后扫描该串，同时在树中相应移动。起始时从树根出发，视各比特位的取值相应地向左或右深入下一层，直到抵达叶节点。比如，在扫描过前三位"110"后将抵达叶节点'M'。此时，可以输出其对应的字符'M'，然后重新回到树根，并继续扫描编码串的剩余部分。比如，再经过接下来的两位"01"后将抵达叶节点'E'，同样地输出字符'E'并回到树根。如此迭代，即可无歧义地解析出原文中的所有字符（习题[5-6]）。

实际上，这一解码过程甚至可以在二进制编码串的接收过程中实时进行，而不必等到所有比特位都到达之后才开始，因此这类算法属于在线算法。

■ PFC编码树的构造

由上可见，PFC编码方案可由PFC编码树来描述，由编码树不仅可以快速生成编码表，而且直接支持高效的解码。那么，任意给定一个字符集Σ，如何构造出PFC编码方案呢？

为此，需要首先解决二叉树本身作为数据结构的描述和实现问题。

§5.3 二叉树的实现

作为图的特殊形式，二叉树的基本组成单元是节点与边；作为数据结构，其基本的组成实体是二叉树节点（binary tree node），而边则对应于节点之间的相互引用。

5.3.1 二叉树节点

■ BinNode模板类

以二叉树节点BinNode模板类，可定义如代码5.1所示。

```
#define BinNodePosi(T) BinNode<T>* //节点位置
#define stature(p) ((p) ? (p)->height : -1) //节点高度（与“空树高度为-1”的约定相统一）
typedef enum { RB_RED, RB_BLACK} RBColor; //节点颜色

template <typename T> struct BinNode { //二叉树节点模板类
// 成员（为简化描述起见统一开放，读者可根据需要进一步封装）
   T data; //数值
   BinNodePosi(T) parent; BinNodePosi(T) lc; BinNodePosi(T) rc; //父节点及左、右孩子
   int height; //高度（通用）
   int npl; //Null Path Length（左式堆，也可直接用height代替）
   RBColor color; //颜色（红黑树）
// 构造函数
   BinNode() :
      parent ( NULL ), lc ( NULL ), rc ( NULL ), height ( 0 ), npl ( 1 ), color ( RB_RED ) { }
   BinNode ( T e, BinNodePosi(T) p = NULL, BinNodePosi(T) lc = NULL, BinNodePosi(T) rc = NULL,
             int h = 0, int l = 1, RBColor c = RB_RED ) :
      data ( e ), parent ( p ), lc ( lc ), rc ( rc ), height ( h ), npl ( l ), color ( c ) { }
// 操作接口
   int size(); //统计当前节点后代总数，亦即以其为根的子树的规模
   BinNodePosi(T) insertAsLC ( T const& ); //作为当前节点的左孩子插入新节点
   BinNodePosi(T) insertAsRC ( T const& ); //作为当前节点的右孩子插入新节点
   BinNodePosi(T) succ(); //取当前节点的直接后继
   template <typename VST> void travLevel ( VST& ); //子树层次遍历
   template <typename VST> void travPre ( VST& ); //子树先序遍历
   template <typename VST> void travIn ( VST& ); //子树中序遍历
   template <typename VST> void travPost ( VST& ); //子树后序遍历
// 比较器、判等器（各列其一，其余自行补充）
   bool operator< ( BinNode const& bn ) { return data < bn.data; } //小于
   bool operator== ( BinNode const& bn ) { return data == bn.data; } //等于
};
```

代码5.1 二叉树节点模板类BinNode

这里，通过宏BinNodePosi来指代节点位置，以简化后续代码的描述；通过定义宏stature，则可以保证从节点返回的高度值，能够与“空树高度为-1”的约定相统一。

■ 成员变量

如图5.10所示，BinNode节点由多个成员变量组成，它们分别记录了当前节点的父亲和孩子的位置、节点内存放的数据以及节点的高度等指标，这些都是二叉树相关算法赖以实现的基础。

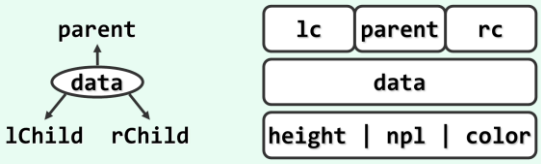


图5.10 BinNode模板类的逻辑结构

其中，data的类型由模板变量T指定，用于存放各节点所对应的数值对象。lc、rc和parent均为指针类型，分别指向左、右孩子以及父节点的位置。如此，既可将各节点联接起来，也可在它们之间漫游移动。比如稍后5.4节将要介绍的遍历算法，就必须借助此类位置变量。当然，通过判断这些变量所指位置是否为NULL，也可确定当前节点的类型。比如，v.parent = NULL当且仅当v是根节点，而v.lc = v.rc = NULL当且仅当v是叶节点。

后续章节将基于二叉树实现二叉搜索树和优先级队列等数据结构，而节点高度height在其中的具体语义也有所不同。比如，8.3节的红黑树将采用所谓的黑高度（black height），而10.3节的左式堆则采用所谓的空节点通路长度（null path length）。尽管后者也可以直接沿用height变量，但出于可读性的考虑，这里还是专门设置了一个变量npl。

有些种类的二叉树还可能需要其它的变量来描述节点状态，比如针对其中节点的颜色，红黑树需要引入一个属于枚举类型RB_Color的变量color。

根据不同应用需求，还可以针对节点的深度增设成员变量depth，或者针对以当前节点为根的子树规模（该节点的后代数目）增设成员变量size。利用这些变量固然可以加速静态的查询或搜索，但为保持这些变量的时效性，在所属二叉树发生结构性调整（比如节点的插入或删除）之后，这些成员变量都要动态地更新。因此，究竟是否值得引入此类成员变量，必须权衡利弊。比如，在二叉树结构改变频繁以至于动态操作远多于静态操作的场合，舍弃深度、子树规模等变量，转而在实际需要时再直接计算这些指标，应是更为明智的选择。

■ 快捷方式

在BinNode模板类各接口以及后续相关算法的实现中，将频繁检查和判断二叉树节点的状态与性质，有时还需要定位与之相关的（兄弟、叔叔等）特定节点，为简化算法描述同时增强可读性，不妨如代码5.2所示将其中常用功能以宏的形式加以整理归纳。

```
1  /*****************************************************************************************
2   * BinNode状态与性质的判断
3   *****************************************************************************************/
4  #define IsRoot(x) ( ! ( (x).parent ) )
5  #define IsLChild(x) ( ! IsRoot(x) && ( & (x) == (x).parent->lc ) )
6  #define IsRChild(x) ( ! IsRoot(x) && ( & (x) == (x).parent->rc ) )
7  #define HasParent(x) ( ! IsRoot(x) )
8  #define HasLChild(x) ( (x).lc )
9  #define HasRChild(x) ( (x).rc )
10 #define HasChild(x) ( HasLChild(x) || HasRChild(x) ) //至少拥有一个孩子
11 #define HasBothChild(x) ( HasLChild(x) && HasRChild(x) ) //同时拥有两个孩子
```

```
#define IsLeaf(x) ( ! HasChild(x) )

/************************************************************************************************
 * 与BinNode具有特定关系的节点及指针
 ************************************************************************************************/
#define sibling(p) /*兄弟*/ \
   ( IsLChild( * (p) ) ? (p)->parent->rc : (p)->parent->lc )

#define uncle(x) /*叔叔*/ \
   ( IsLChild( * ( (x)->parent ) ) ? (x)->parent->parent->rc : (x)->parent->parent->lc )

#define FromParentTo(x) /*来自父亲的引用*/ \
   ( IsRoot(x) ? _root : ( IsLChild(x) ? (x).parent->lc : (x).parent->rc ) )
```

代码5.2 以宏的形式对基于BinNode的操作做一归纳整理

5.3.2 二叉树节点操作接口

由于BinNode模板类本身处于底层，故这里也将所有操作接口统一设置为开放权限，以简化描述。同样地，注重数据结构封装性的读者可在此基础之上自行修改扩充。

■ **插入孩子节点**

```
template <typename T> BinNodePosi(T) BinNode<T>::insertAsLC ( T const& e )
{ return lc = new BinNode ( e, this ); } //将e作为当前节点的左孩子插入二叉树

template <typename T> BinNodePosi(T) BinNode<T>::insertAsRC ( T const& e )
{ return rc = new BinNode ( e, this ); } //将e作为当前节点的右孩子插入二叉树
```

代码5.3 二叉树节点左、右孩子的插入

可见，为将新节点作为当前节点的左孩子插入树中，可如图5.11(a)所示，先创建新节点；再如图(b)所示，将当前节点作为新节点的父亲，并令新节点作为当前节点的左孩子。这里约定，在插入新节点之前，当前节点尚无左孩子。

右孩子的插入过程完全对称，不再赘述。

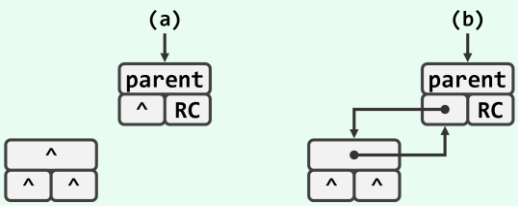


图5.11 二叉树节点左孩子插入过程

■ **定位直接后继**

稍后在5.4.3节我们将会看到，通过中序遍历，可在二叉树各节点之间定义一个线性次序。相应地，各节点之间也可定义前驱与后继关系。这里的succ()接口，可以返回当前节点的直接后继（如果存在）。该接口的具体实现，将在129页代码5.16中给出。

■ **遍历**

稍后的5.4节，将从递归和迭代两个角度，分别介绍各种遍历算法的不同实现。为便于测试与比较，不妨将这些算法的不同版本统一归入统一的接口中，并在调用时随机选择。

```
template <typename T> template <typename VST> //元素类型、操作器
void BinNode<T>::travIn ( VST& visit ) { //二叉树中序遍历算法统一入口
   switch ( rand() % 5 ) { //此处暂随机选择以做测试，共五种选择
      case 1: travIn_I1 ( this, visit ); break; //迭代版#1
      case 2: travIn_I2 ( this, visit ); break; //迭代版#2
      case 3: travIn_I3 ( this, visit ); break; //迭代版#3
      case 4: travIn_I4 ( this, visit ); break; //迭代版#4
      default: travIn_R ( this, visit ); break; //递归版
   }
}
```

代码5.4 二叉树中序遍历算法的统一入口

比如，中序遍历算法的五种实现方式（其中travIn_I4留作习题[5-17]），即可如代码5.4所示，纳入统一的BinNode::travIn()接口。其余遍历算法的处理方法类似，不再赘述。

5.3.3 二叉树

■ BinTree模板类

在BinNode模板类的基础之上，可如代码5.5所示定义二叉树BinTree模板类。

```
#include "BinNode.h" //引入二叉树节点类
template <typename T> class BinTree { //二叉树模板类
protected:
   int _size; BinNodePosi(T) _root; //规模、根节点
   virtual int updateHeight ( BinNodePosi(T) x ); //更新节点x的高度
   void updateHeightAbove ( BinNodePosi(T) x ); //更新节点x及其祖先的高度
public:
   BinTree() : _size ( 0 ), _root ( NULL ) { } //构造函数
   ~BinTree() { if ( 0 < _size ) remove ( _root ); } //析构函数
   int size() const { return _size; } //规模
   bool empty() const { return !_root; } //判空
   BinNodePosi(T) root() const { return _root; } //树根
   BinNodePosi(T) insertAsRoot ( T const& e ); //插入根节点
   BinNodePosi(T) insertAsLC ( BinNodePosi(T) x, T const& e ); //e作为x的左孩子（原无）插入
   BinNodePosi(T) insertAsRC ( BinNodePosi(T) x, T const& e ); //e作为x的右孩子（原无）插入
   BinNodePosi(T) attachAsLC ( BinNodePosi(T) x, BinTree<T>* &T ); //T作为x左子树接入
   BinNodePosi(T) attachAsRC ( BinNodePosi(T) x, BinTree<T>* &T ); //T作为x右子树接入
   int remove ( BinNodePosi(T) x ); //删除以位置x处节点为根的子树，返回该子树原先的规模
   BinTree<T>* secede ( BinNodePosi(T) x ); //将子树x从当前树中摘除，并将其转换为一棵独立子树
   template <typename VST> //操作器
   void travLevel ( VST& visit ) { if ( _root ) _root->travLevel ( visit ); } //层次遍历
   template <typename VST> //操作器
   void travPre ( VST& visit ) { if ( _root ) _root->travPre ( visit ); } //先序遍历
```

```
24    template <typename VST> //操作器
25    void travIn ( VST& visit ) { if ( _root ) _root->travIn ( visit ); } //中序遍历
26    template <typename VST> //操作器
27    void travPost ( VST& visit ) { if ( _root ) _root->travPost ( visit ); } //后序遍历
28    bool operator< ( BinTree<T> const& t ) //比较器（其余自行补充）
29    { return _root && t._root && lt ( _root, t._root ); }
30    bool operator== ( BinTree<T> const& t ) //判等器
31    { return _root && t._root && ( _root == t._root ); }
32 }; //BinTree
```

代码5.5 二叉树模板类BinTree

■ 高度更新

二叉树任一节点的高度，都等于其孩子节点的最大高度加一。于是，每当某一节点的孩子或后代有所增减，其高度都有必要及时更新。然而实际上，节点自身很难发现后代的变化，因此这里不妨反过来采用另一处理策略：一旦有节点加入或离开二叉树，则更新其所有祖先的高度。请读者自行验证，这一原则实际上与前一个等效（习题[5-3]）。

在每一节点v处，只需读出其左、右孩子的高度并取二者之间的大者，再计入当前节点本身，就得到了v的新高度。通常，接下来还需要从v出发沿parent指针逆行向上，依次更新各代祖先的高度记录。这一过程可具体实现如代码5.6所示。

```
1 template <typename T> int BinTree<T>::updateHeight ( BinNodePosi(T) x ) //更新节点x高度
2 { return x->height = 1 + max ( stature ( x->lc ), stature ( x->rc ) ); } //具体规则，因树而异
3
4 template <typename T> void BinTree<T>::updateHeightAbove ( BinNodePosi(T) x ) //更新高度
5 { while ( x ) { updateHeight ( x ); x = x->parent; } } //从x出发，覆盖历代祖先。可优化
```

代码5.6 二叉树节点的高度更新

更新每一节点本身的高度，只需执行两次getHeight()操作、两次加法以及两次取最大操作，不过常数时间，故updateHeight()算法总体运行时间也是$\mathcal{O}$(depth(v) + 1)，其中depth(v)为节点v的深度。当然，这一算法还可进一步优化（习题[5-4]）。

在某些种类的二叉树（例如8.3节将要介绍的红黑树）中，高度的定义有所不同，因此这里将updateHeight()定义为保护级的虚方法，以便派生类在必要时重写（override）。

■ 节点插入

二叉树节点可以通过三种方式插入二叉树中，具体实现如代码5.7所示。

```
1 template <typename T> BinNodePosi(T) BinTree<T>::insertAsRoot ( T const& e )
2 { _size = 1; return _root = new BinNode<T> ( e ); } //将e当作根节点插入空的二叉树
3
4 template <typename T> BinNodePosi(T) BinTree<T>::insertAsLC ( BinNodePosi(T) x, T const& e )
5 { _size++; x->insertAsLC ( e ); updateHeightAbove ( x ); return x->lc; } //e插入为x的左孩子
6
7 template <typename T> BinNodePosi(T) BinTree<T>::insertAsRC ( BinNodePosi(T) x, T const& e )
```

```
8 { _size++; x->insertAsRC ( e ); updateHeightAbove ( x ); return x->rc; } //e插入为x的右孩子
```

代码5.7 二叉树根、左、右节点的插入

insertAsRoot()接口用于将节点插入空树中，当然，该节点随即也应成为树根。

一般地，如图5.12(a)所示，若二叉树T中某节点x的右孩子为空，则可通过T.insertAsRC()接口为其添加一个右孩子。为此可如图(b)所示，调用x->insertAsRC()接口，将二者按照父子关系相互联接，同时通过updateHeightAbove()接口更新x所有祖先的高度，并更新全树规模。

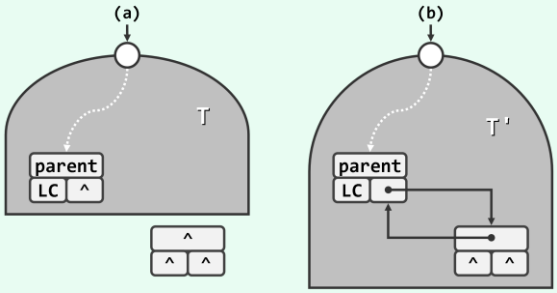


图5.12 右节点插入过程：(a)插入前；(b)插入后

请注意这里的两个同名insertAsRC()接口，它们各自所属的对象类型不同。

左侧节点的插入过程与此相仿，可对称地调用insertAsLC()完成。

■ 子树接入

如代码5.8所示，任一二叉树均可作为另一二叉树中指定节点的左子树或右子树，植入其中。

```
1  template <typename T> //二叉树子树接入算法：将S当作节点x的左子树接入，S本身置空
2  BinNodePosi(T) BinTree<T>::attachAsLC ( BinNodePosi(T) x, BinTree<T>* &S ) { //x->lc == NULL
3     if ( x->lc = S->_root ) x->lc->parent = x; //接入
4     _size += S->_size; updateHeightAbove ( x ); //更新全树规模与x所有祖先的高度
5     S->_root = NULL; S->_size = 0; release ( S ); S = NULL; return x; //释放原树，返回接入位置
6  }
7
8  template <typename T> //二叉树子树接入算法：将S当作节点x的右子树接入，S本身置空
9  BinNodePosi(T) BinTree<T>::attachAsRC ( BinNodePosi(T) x, BinTree<T>* &S ) { //x->rc == NULL
10    if ( x->rc = S->_root ) x->rc->parent = x; //接入
11    _size += S->_size; updateHeightAbove ( x ); //更新全树规模与x所有祖先的高度
12    S->_root = NULL; S->_size = 0; release ( S ); S = NULL; return x; //释放原树，返回接入位置
13 }
```

代码5.8 二叉树子树的接入

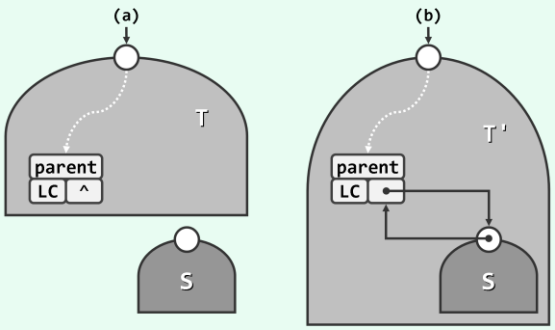


图5.13 右子树接入过程：(a)接入前；(b)接入后

如图5.13(a)，若二叉树T中节点x的右孩子为空，则attachAsRC()接口首先将待植入的二叉树S的根节点作为x的右孩子，同时令x作为该根节点的父亲；然后，更新全树规模以及节点x所有祖先的高度；最后，将树S中除已接入的各节点之外的其余部分归还系统。

左子树接入过程与此类似，可对称地调用attachAsLC()完成。

■ 子树删除

子树删除的过程，与如图5.13所示的子树接入过程恰好相反，不同之处在于，需要将被摘除子树中的节点，逐一释放并归还系统（习题[5-5]）。具体实现如代码5.9所示。

```
template <typename T> //删除二叉树中位置x处的节点及其后代，返回被删除节点的数值
int BinTree<T>::remove ( BinNodePosi(T) x ) { //assert: x为二叉树中的合法位置
   FromParentTo ( *x ) = NULL; //切断来自父节点的指针
   updateHeightAbove ( x->parent ); //更新祖先高度
   int n = removeAt ( x ); _size -= n; return n; //删除子树x，更新规模，返回删除节点总数
}
template <typename T> //删除二叉树中位置x处的节点及其后代，返回被删除节点的数值
static int removeAt ( BinNodePosi(T) x ) { //assert: x为二叉树中的合法位置
   if ( !x ) return 0; //递归基：空树
   int n = 1 + removeAt ( x->lc ) + removeAt ( x->rc ); //递归释放左、右子树
   release ( x->data ); release ( x ); return n; //释放被摘除节点，并返回删除节点总数
}
```

代码5.9 二叉树子树的删除

■ 子树分离

子树分离的过程与以上的子树删除过程基本一致，不同之处在于，需要对分离出来的子树重新封装，并返回给上层调用者。具体实现如代码5.10所示。

```
template <typename T> //二叉树子树分离算法：将子树x从当前树中摘除，将其封装为一棵独立子树返回
BinTree<T>* BinTree<T>::secede ( BinNodePosi(T) x ) { //assert: x为二叉树中的合法位置
   FromParentTo ( *x ) = NULL; //切断来自父节点的指针
   updateHeightAbove ( x->parent ); //更新原树中所有祖先的高度
   BinTree<T>* S = new BinTree<T>; S->_root = x; x->parent = NULL; //新树以x为根
   S->_size = x->size(); _size -= S->_size; return S; //更新规模，返回分离出来的子树
}
```

代码5.10 二叉树子树的分离

■ 复杂度

就二叉树拓扑结构的变化范围而言，以上算法均只涉及局部的常数个节点。因此，除了更新祖先高度和释放节点等操作，只需常数时间。

§5.4 遍历

对二叉树的访问多可抽象为如下形式：按照某种约定的次序，对节点各访问一次且仅一次。与向量和列表等线性结构一样，二叉树的这类访问也称作遍历（traversal）。遍历之于二叉树的意义，同样在于为相关算法的实现提供通用的框架。此外，这一过程也等效于将半线性的树形结构，转换为线性结构。不过，二叉树毕竟已不再属于线性结构，故相对而言其遍历更为复杂。

为此，以下首先针对几种典型的遍历策略，按照代码5.1和代码5.5所列接口，分别给出相应的递归式实现；然后，再分别介绍其对应的迭代式实现，以提高遍历算法的实际效率。

5.4.1 递归式遍历

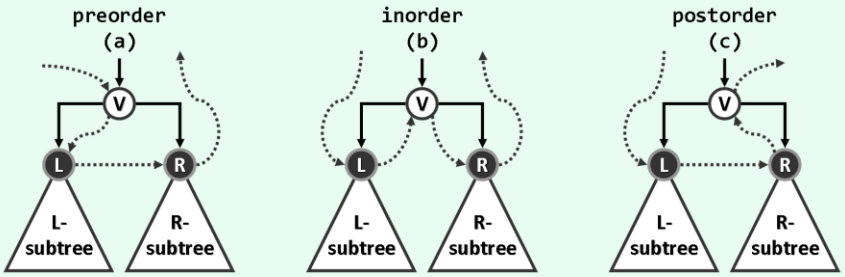


图5.14 二叉树遍历的全局次序由局部次序规则确定

二叉树本身并不具有天然的全局次序，故为实现遍历，首先需要在各节点与其孩子之间约定某种局部次序，从而间接地定义出全局次序。按惯例左孩子优先于右孩子，故若将节点及其孩子分别记作V、L和R，则如图5.14所示，局部访问的次序有VLR、LVR和LRV三种选择。根据节点V在其中的访问次序，这三种策略也相应地分别称作先序遍历、中序遍历和后序遍历，分述如下。

■ 先序遍历

得益于递归定义的简洁性，如代码5.11所示，只需数行即可实现先序遍历算法。

```
1 template <typename T, typename VST> //元素类型、操作器
2 void travPre_R ( BinNodePosi(T) x, VST& visit ) { //二叉树先序遍历算法（递归版）
3    if ( !x ) return;
4    visit ( x->data );
5    travPre_R ( x->lc, visit );
6    travPre_R ( x->rc, visit );
7 }
```

代码5.11 二叉树先序遍历算法（递归版）

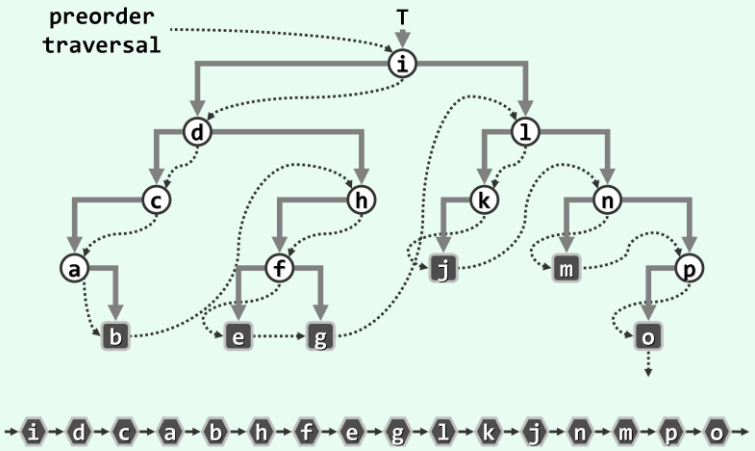


图5.15 二叉树（上）的先序遍历序列（下）

为遍历（子）树x，首先核对x是否为空。若x为空，则直接退出——其效果相当于递归基。反之，若x非空，则按照先序遍历关于局部次序的定义，优先访问其根节点x；然后，依次深入左子树和右子树，递归地进行遍历。实际上，这一实现模式也同样可以应用于中序和后序遍历。

如图5.15所示，经遍历之后，可将节点按某一线性次序排列，称作遍历（生成）序列。

■ 后序遍历

```
1 template <typename T, typename VST> //元素类型、操作器
2 void travPost_R ( BinNodePosi(T) x, VST& visit ) { //二叉树后序遍历算法（递归版）
3    if ( !x ) return;
4    travPost_R ( x->lc, visit );
5    travPost_R ( x->rc, visit );
6    visit ( x->data );
7 }
```

代码5.12 二叉树后序遍历算法（递归版）

仿照以上先序遍历的模式，可如代码5.12所示实现递归版后序遍历算法。

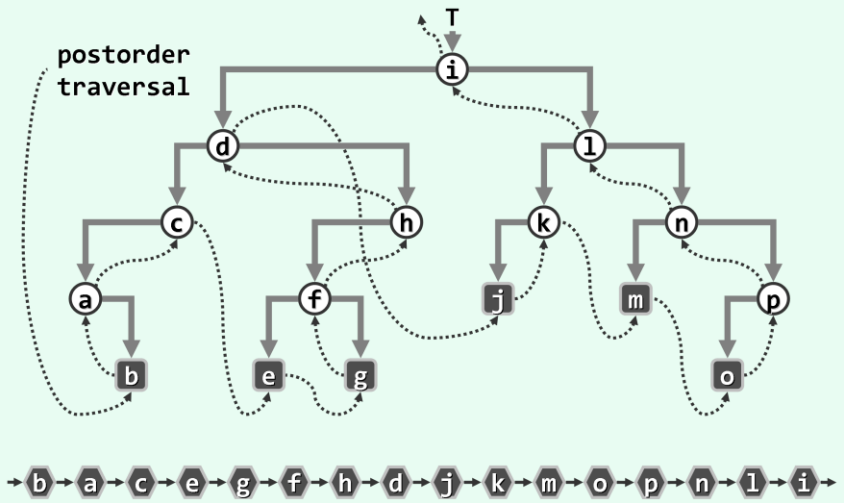


图5.16 二叉树的后序遍历序列

按照后序遍历规则，为遍历非空的（子）树x，将在依次递归遍历其左子树和右子树之后，才访问节点x。对于以上二叉树实例，其完整的后序遍历过程以及生成的遍历序列如图5.16所示。与图5.15做一对比可见，先序遍历序列与后序遍历序列并非简单的逆序关系。

■ 中序遍历

再次仿照以上模式，可实现递归版中序遍历算法如代码5.13所示。

```
1 template <typename T, typename VST> //元素类型、操作器
2 void travIn_R ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历算法（递归版）
3    if ( !x ) return;
4    travIn_R ( x->lc, visit );
5    visit ( x->data );
6    travIn_R ( x->rc, visit );
7 }
```

代码5.13 二叉树中序遍历算法（递归版）

按照中序遍历规则，为遍历非空的（子）树x，将依次递归遍历其左子树、访问节点x、递归遍历其右子树。以上二叉树实例的中序遍历过程以及生成的遍历序列，如图5.17所示。

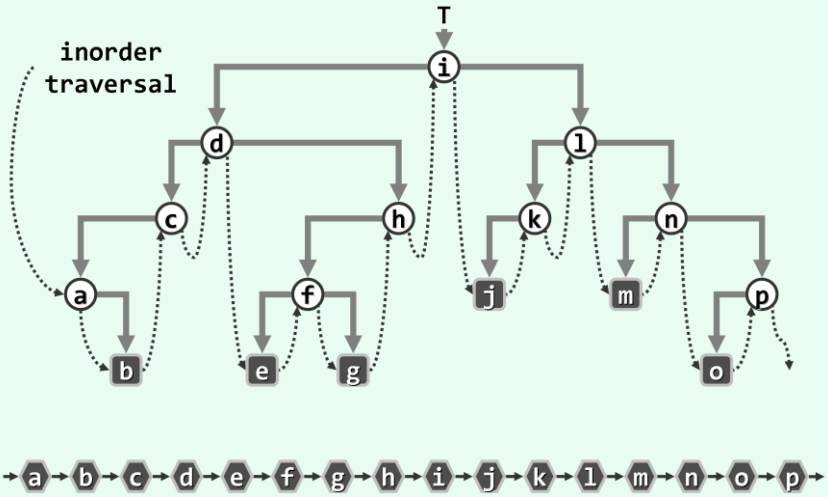


图5.17 二叉树的中序遍历序列

与以上的先序和后序遍历序列做一对比不难发现，各节点在中序遍历序列中的局部次序，与按照有序树定义所确定的全局左、右次序完全吻合。这一现象并非巧合，在第7章和第8章中，这正是搜索树及其等价变换的原理和依据所在。

5.4.2 *迭代版先序遍历

无论以上各种递归式遍历算法还是以下各种迭代式遍历算法，都只需渐进的线性时间（习题[5-9]和[5-11]）；而且相对而言，前者更加简明。既然如此，有何必要介绍迭代式遍历算法呢？

首先，递归版遍历算法时间、空间复杂度的常系数，相对于迭代版更大。同时，从学习的角度来看，从底层实现迭代式遍历，也是加深对相关过程与技巧理解的有效途径。

- **版本1**

观察先序遍历的递归版（代码5.11）可发现，其中针对右子树的递归属于尾递归，左子树的则接近于尾递归。故参照消除尾递归的一般性方法，不难将其改写迭代版（习题[5-10]）。

- **版本2**

很遗憾，以上思路并不容易推广到非尾递归的场合，比如在中序或后序遍历中，至少有一个递归方向严格地不属于尾递归。此时，如下另一迭代式版本的实现思路，则更具参考价值。

如图5.18所示，在二叉树T中，从根节点出发沿着左分支一直下行的那条通路（以粗线示意），称作最左侧通路（leftmost path）。若将沿途节点分别记作L_k，k = 0, 1, 2, ..., d，则最左侧通路终止于没有左孩子末端节点L_d。若这些节点的右孩子和右子树分别记作R_k和T_k，k = 0, 1, 2, ..., d，则该二叉树的先序遍历序列可表示为：

```
preorder(T) =
    visit(L0),    visit(L1), ...,          visit(Ld);
    preorder(Td), ...,       preorder(T1), preorder(T0)
```

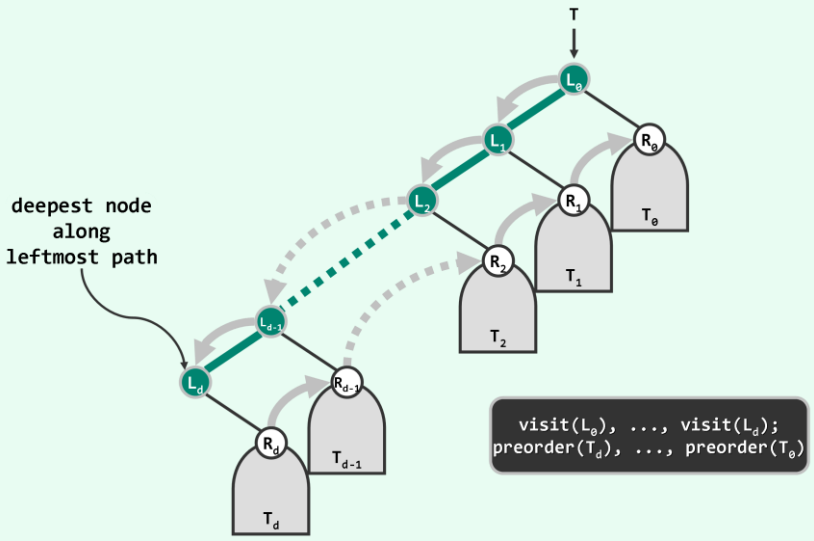


图5.18 先序遍历过程：先沿最左侧通路自顶而下访问沿途节点，再自底而上依次遍历这些节点的右子树

也就是说，先序遍历序列可分解为两段：沿最左侧通路自顶而下访问的各节点，以及自底而上遍历的对应右子树。基于对先序遍历序列的这一理解，可以导出以下迭代式先序遍历算法。

```
1  //从当前节点出发，沿左分支不断深入，直至没有左分支的节点；沿途节点遇到后立即访问
2  template <typename T, typename VST> //元素类型、操作器
3  static void visitAlongLeftBranch ( BinNodePosi(T) x, VST& visit, Stack<BinNodePosi(T)>& S ) {
4     while ( x ) {
5        visit ( x->data ); //访问当前节点
6        S.push ( x->rc ); //右孩子入栈暂存（可优化：通过判断，避免空的右孩子入栈）
7        x = x->lc;  //沿左分支深入一层
8     }
9  }
10
11 template <typename T, typename VST> //元素类型、操作器
12 void travPre_I2 ( BinNodePosi(T) x, VST& visit ) { //二叉树先序遍历算法（迭代版#2）
13    Stack<BinNodePosi(T)> S; //辅助栈
14    while ( true ) {
15       visitAlongLeftBranch ( x, visit, S ); //从当前节点出发，逐批访问
16       if ( S.empty() ) break; //直到栈空
17       x = S.pop(); //弹出下一批的起点
18    }
19 }
```

代码5.14 二叉树先序遍历算法（迭代版#2）

如代码5.14所示，在全树以及其中每一棵子树的根节点处，该算法都首先调用函数VisitAlongLeftBranch()，自顶而下访问最左侧通路沿途的各个节点。这里也使用了一个辅助栈，逆序记录最左侧通路上的节点，以便确定其对应右子树自底而上的遍历次序。

5.4.3 *迭代版中序遍历

如上所述，在中序遍历的递归版本（125页代码5.13）中，尽管右子树的递归遍历是尾递归，但左子树绝对不是。实际上，实现迭代式中序遍历算法的难点正在于此，不过好在迭代式先序遍历的版本2可以为我们提供启发和借鉴。

■ 版本1

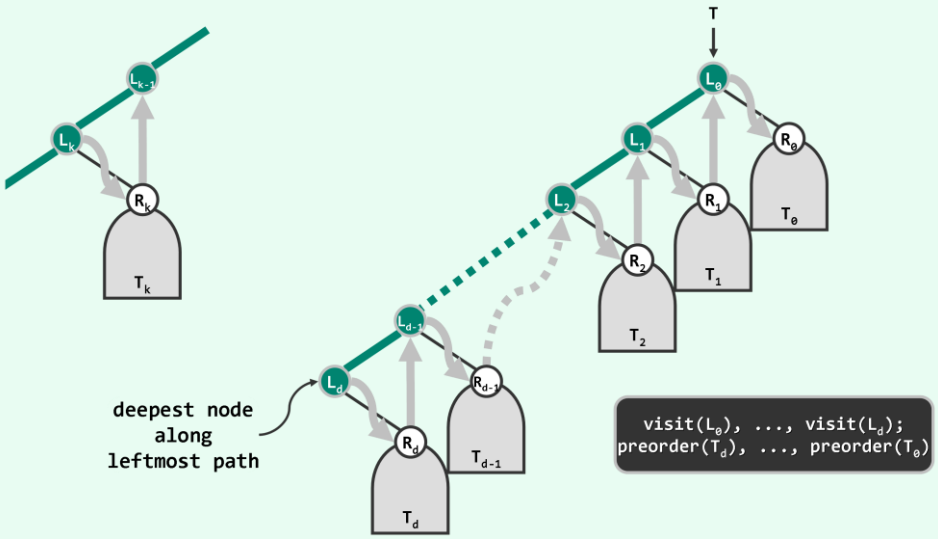


图5.19 中序遍历过程：顺着最左侧通路，自底而上依次访问沿途各节点及其右子树

如图5.19所示，参照迭代式先序遍历版本2的思路，再次考查二叉树T的最左侧通路，并对其中的节点和子树标记命名。于是，T的中序遍历序列可表示为：

$$\begin{aligned} \text{inorder}(T) = {} & \text{visit}(L_d), \text{inorder}(T_d); \\ & \text{visit}(L_{d-1}), \text{inorder}(T_{d-1}); \\ & \ldots, \ldots; \\ & \text{visit}(L_1), \text{inorder}(T_1); \\ & \text{visit}(L_0), \text{inorder}(T_0) \end{aligned}$$

也就是说，沿最左侧通路自底而上，以沿途各节点为界，中序遍历序列可分解为d + 1段。如图5.19左侧所示，各段彼此独立，且均包括访问来自最左侧通路的某一节点L_k，以及遍历其对应的右子树T_k。

基于对中序遍历序列的这一理解，即可导出如代码5.15所示的迭代式中序遍历算法。

```
1 template <typename T> //从当前节点出发，沿左分支不断深入，直至没有左分支的节点
2 static void goAlongLeftBranch ( BinNodePosi(T) x, Stack<BinNodePosi(T)>& S ) {
3    while ( x ) { S.push ( x ); x = x->lc; } //当前节点入栈后随即向左侧分支深入，迭代直到无左孩子
4 }
5
6 template <typename T, typename VST> //元素类型、操作器
7 void travIn_I1 ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历算法（迭代版#1）
```

```
8    Stack<BinNodePosi(T)> S; //辅助栈
9    while ( true ) {
10      goAlongLeftBranch ( x, S ); //从当前节点出发，逐批入栈
11      if ( S.empty() ) break; //直至所有节点处理完毕
12      x = S.pop(); visit ( x->data ); //弹出栈顶节点并访问之
13      x = x->rc; //转向右子树
14   }
15 }
```

代码5.15 二叉树中序遍历算法（迭代版#1）

在全树及其中每一棵子树的根节点处，该算法首先调用函数goAlongLeftBranch()，沿最左侧通路自顶而下抵达末端节点L_d。在此过程中，利用辅助栈逆序地记录和保存沿途经过的各个节点，以便确定自底而上各段遍历子序列最终在宏观上的拼接次序。

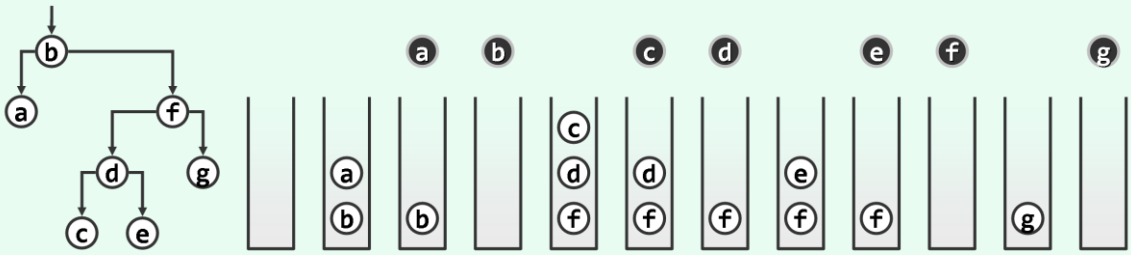


图5.20 迭代式中序遍历实例（出栈节点以深色示意）

图5.20以左侧二叉树为例，给出了中序遍历辅助栈从初始化到再次变空的演变过程。

■ 直接后继及其定位

与所有遍历一样，中序遍历的实质功能也可理解为，为所有节点赋予一个次序，从而将半线性的二叉树转化为线性结构。于是一旦指定了遍历策略，即可与向量和列表一样，在二叉树的节点之间定义前驱与后继关系。其中没有前驱（后继）的节点称作首（末）节点。

对于后面将要介绍的二叉搜索树，中序遍历的作用至关重要。相关算法必需的一项基本操作，就是定位任一节点在中序遍历序列中的直接后继。为此，可实现succ()接口如代码5.16所示。

```
1  template <typename T> BinNodePosi(T) BinNode<T>::succ() { //定位节点v的直接后继
2     BinNodePosi(T) s = this; //记录后继的临时变量
3     if ( rc ) { //若有右孩子，则直接后继必在右子树中，具体地就是
4        s = rc; //右子树中
5        while ( HasLChild ( *s ) ) s = s->lc; //最靠左（最小）的节点
6     } else { //否则，直接后继应是“将当前节点包含于其左子树中的最低祖先”，具体地就是
7        while ( IsRChild ( *s ) ) s = s->parent; //逆向地沿右向分支，不断朝左上方移动
8        s = s->parent; //最后再朝右上方移动一步，即抵达直接后继（如果存在）
9     }
10    return s;
11 }
```

代码5.16 二叉树节点直接后继的定位

这里，共分两大类情况。若当前节点有右孩子，则其直接后继必然存在，且属于其右子树。此时只需转入右子树，再沿该子树的最左侧通路朝左下方深入，直到抵达子树中最靠左（最小）的节点。以图5.20中节点b为例，如此可确定其直接后继为节点c。

反之，若当前节点没有右子树，则若其直接后继存在，必为该节点的某一祖先，且是将当前节点纳入其左子树的最低祖先。于是首先沿右侧通路朝左上方上升，当不能继续前进时，再朝右上方移动一步即可。以图5.20中节点e为例，如此可确定其直接后继为节点f。

作为后一情况的特例，出口时s可能为NULL。这意味着此前沿着右侧通路向上的回溯，抵达了树根。也就是说，当前节点全树右侧通路的终点——它也是中序遍历的终点，没有后继。

■ 版本2

代码5.15经进一步改写之后，可得到如代码5.17所示的另一迭代式中序遍历算法。

```
1  template <typename T, typename VST> //元素类型、操作器
2  void travIn_I2 ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历算法（迭代版#2）
3     Stack<BinNodePosi(T)> S; //辅助栈
4     while ( true )
5        if ( x ) {
6           S.push ( x ); //根节点进栈
7           x = x->lc; //深入遍历左子树
8        } else if ( !S.empty() ) {
9           x = S.pop(); //尚未访问的最低祖先节点退栈
10          visit ( x->data ); //访问该祖先节点
11          x = x->rc; //遍历祖先的右子树
12       } else
13          break; //遍历完成
14 }
```

代码5.17 二叉树中序遍历算法（迭代版#2）

虽然版本2只不过是版本1的等价形式，但借助它可便捷地设计和实现以下版本3。

■ 版本3

以上的迭代式遍历算法都需使用辅助栈，尽管这对遍历算法的渐进时间复杂度没有实质影响，但所需辅助空间的规模将线性正比于二叉树的高度，在最坏情况下与节点总数相当。

为此，可对代码5.17版本2继续改进，借助BinNode对象内部的parent指针，如代码5.18所示实现中序遍历的第三个迭代版本。该版本无需使用任何结构，总体仅需$\mathcal{O}(1)$的辅助空间，属于就地算法。当然，因需要反复调用succ()，时间效率有所倒退（习题[5-16]）。

```
1  template <typename T, typename VST> //元素类型、操作器
2  void travIn_I3 ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历算法（迭代版#3，无需辅助栈）
3     bool backtrack = false; //前一步是否刚从右子树回溯——省去栈，仅O(1)辅助空间
4     while ( true )
5        if ( !backtrack && HasLChild ( *x ) ) //若有左子树且不是刚刚回溯，则
6           x = x->lc; //深入遍历左子树
7        else { //否则——无左子树或刚刚回溯（相当于无左子树）
```

```
8           visit ( x->data ); //访问该节点
9           if ( HasRChild ( *x ) ) { //若其右子树非空，则
10             x = x->rc; //深入右子树继续遍历
11             backtrack = false; //并关闭回溯标志
12          } else { //若右子树空，则
13             if ( ! ( x = x->succ() ) ) break; //回溯（含抵达末节点时的退出返回）
14             backtrack = true; //并设置回溯标志
15          }
16       }
17 }
```

代码5.18 二叉树中序遍历算法（迭代版#3）

可见，这里相当于将原辅助栈替换为一个标志位backtrack。每当抵达一个节点，借助该标志即可判断此前是否刚做过一次自下而上的回溯。若不是，则按照中序遍历的策略优先遍历左子树。反之，若刚发生过一次回溯，则意味着当前节点的左子树已经遍历完毕（或等效地，左子树为空），于是便可访问当前节点，然后再深入其右子树继续遍历。

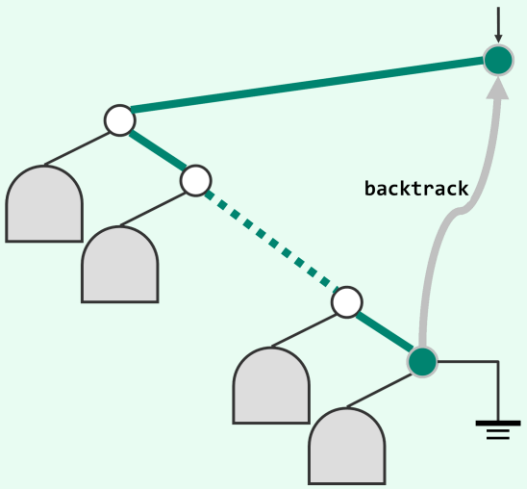


图5.21 中序遍历过程中，在无右孩子的节点处需做回溯

每个节点被访问之后，都应转向其在遍历序列中的直接后继。按照以上的分析，通过检查右子树是否为空，即可在两种情况间做出判断：该后继要么在当前节点的右子树（若该子树非空）中，要么（当右子树为空时）是其某一祖先。如图5.21所示，后一情况即所谓的回溯。

请注意，由succ()返回的直接后继可能是NULL，此时意味着已经遍历至中序遍历意义下的末节点，于是遍历即告完成。

5.4.4 *迭代版后序遍历

在如代码5.12所示后序遍历算法的递归版本中，左、右子树的递归遍历均严格地不属于尾递归，因此实现对应的迭代式算法难度更大。不过，仍可继续套用此前的思路和技巧。我们思考的起点依然是，此时首先访问的是哪个节点？

如图5.22所示，将树T画在二维平面上，并假设所有节点和边均不透明。于是从左侧水平向右看去，未被遮挡的最高叶节点v——称作最高左侧可见叶节点（HLVFL）——即为后序遍历首先访问的节点。请注意，该节点既可能是左孩子，也可能是右孩子，故在图中以垂直边示意它与其父节点之间的联边。

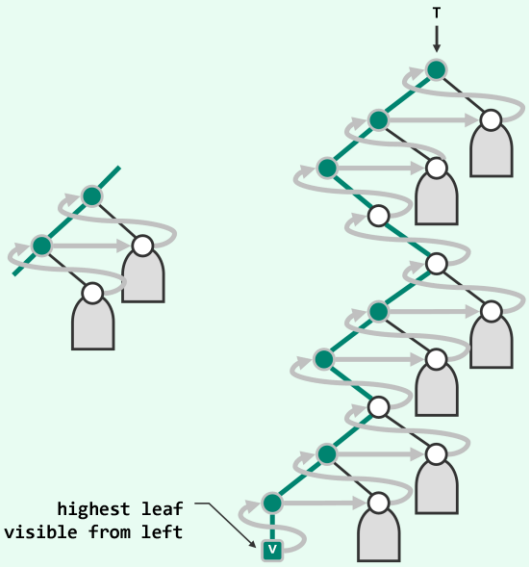


图5.22 后序遍历过程也可划分为模式雷同的若干段

考查联接于v与树根之间的唯一通路（以粗线示意）。与先序与中序遍历类似地，自底而上地沿着该通路，整个后序遍历序列也可分解为若干个片段。每一片段，分别起始于通路上的一个节点，并包括三步：访问当前节点，遍历以其右兄弟（若存在）为根的子树，以及向上回溯至其父节点（若存在）并转入下一片段。

基于以上理解，即可导出如代码5.19所示的迭代式后序遍历算法。

```
1  template <typename T> //在以S栈顶节点为根的子树中，找到最高左侧可见叶节点
2  static void gotoHLVFL ( Stack<BinNodePosi(T)>& S ) { //沿途所遇节点依次入栈
3     while ( BinNodePosi(T) x = S.top() ) //自顶而下，反复检查当前节点（即栈顶）
4        if ( HasLChild ( *x ) ) { //尽可能向左
5           if ( HasRChild ( *x ) ) S.push ( x->rc ); //若有右孩子，优先入栈
6           S.push ( x->lc ); //然后才转至左孩子
7        } else //实不得已
8           S.push ( x->rc ); //才向右
9     S.pop(); //返回之前，弹出栈顶的空节点
10 }
11
12 template <typename T, typename VST>
13 void travPost_I ( BinNodePosi(T) x, VST& visit ) { //二叉树的后序遍历（迭代版）
14    Stack<BinNodePosi(T)> S; //辅助栈
15    if ( x ) S.push ( x ); //根节点入栈
16    while ( !S.empty() ) {
```

```
17        if ( S.top() != x->parent ) //若栈顶非当前节点之父（则必为其右兄），此时需
18           gotoHLVFL ( S ); //在以其右兄为根之子树中，找到HLVFL（相当于递归深入其中）
19        x = S.pop(); visit ( x->data ); //弹出栈顶（即前一节点之后继），并访问之
20     }
21  }
```

代码5.19 二叉树后序遍历算法（迭代版）

可见，在每一棵（子）树的根节点，该算法都首先定位对应的HLVFL节点。同时在此过程中，依然利用辅助栈逆序地保存沿途所经各节点，以确定遍历序列各个片段在宏观上的拼接次序。

图5.23以左侧二叉树为例，给出了后序遍历辅助栈从初始化到再次变空的演变过程。

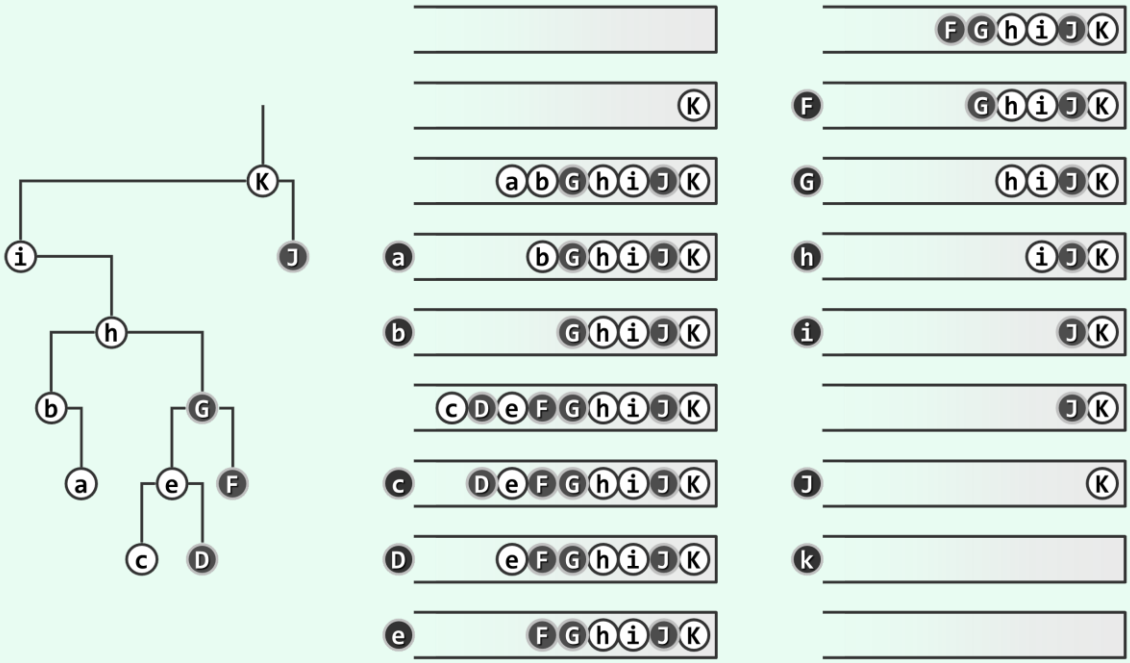


图5.23 迭代式后序遍历实例（出栈节点以深色示意，发生gotoHLVFL()调用的节点以大写字母示意）

请留意此处的入栈规则。在自顶而下查找HLVFL节点的过程中，始终都是尽可能向左，只有在左子树为空时才向右。前一情况下，需令右孩子（若有）和左孩子先后入栈，然后再转向左孩子。后一情况下，只需令右孩子入栈。因此，在主函数travPost_I()的每一步while迭代中，若当前节点node的右兄弟存在，则该兄弟必然位于辅助栈顶。按照后序遍历约定的次序，此时应再次调用gotoHLVFL()以转向以该兄弟为根的子树，并模拟以递归方式对该子树的遍历。

5.4.5 层次遍历

在所谓广度优先遍历或层次遍历（level-order traversal）中，确定节点访问次序的原则可概括为“先上后下、先左后右”——先访问树根，再依次是左孩子、右孩子、左孩子的左孩子、左孩子的右孩子、右孩子的左孩子、右孩子的右孩子、...，依此类推。

当然，有根性和有序性是层次遍历序列得以明确定义的基础。正因为确定了树根，各节点方可拥有深度这一指标，并进而依此排序；有序性则保证孩子有左、右之别，并依此确定同深度节点之间的次序。

为对比效果，同样考查此前图5.15、图5.16和图5.17均采用的二叉树实例。该树完整的层次遍历过程以及生成的遍历序列，如图5.24所示。

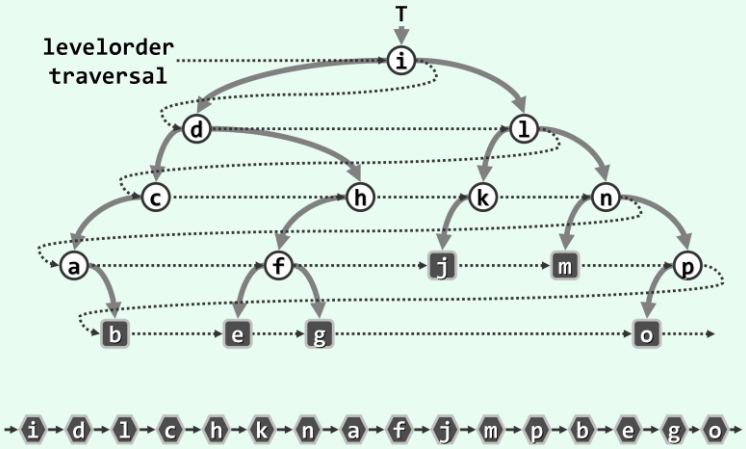


图5.24 二叉树的层次遍历序列

■ 算法实现

此前介绍的迭代式遍历，无论先序、中序还是后序遍历，大多使用了辅助栈，而迭代式层次遍历则需要使用与栈对称的队列结构，算法的具体实现如代码5.20所示。

```
1  template <typename T> template <typename VST> //元素类型、操作器
2  void BinNode<T>::travLevel ( VST& visit ) { //二叉树层次遍历算法
3     Queue<BinNodePosi(T)> Q; //辅助队列
4     Q.enqueue ( this ); //根节点入队
5     while ( !Q.empty() ) { //在队列再次变空之前，反复迭代
6        BinNodePosi(T) x = Q.dequeue(); visit ( x->data ); //取出队首节点并访问之
7        if ( HasLChild ( *x ) ) Q.enqueue ( x->lc ); //左孩子入队
8        if ( HasRChild ( *x ) ) Q.enqueue ( x->rc ); //右孩子入队
9     }
10 }
```

代码5.20 二叉树层次遍历算法

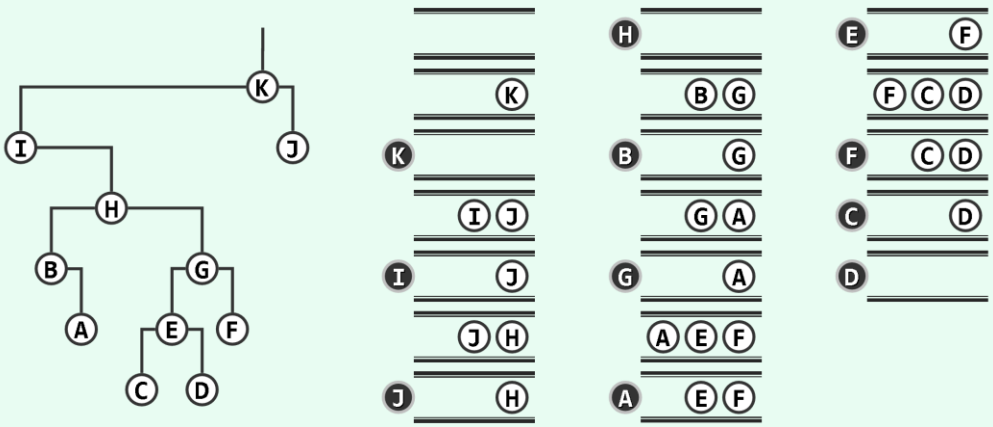


图5.25 层次遍历实例（出队节点以深色示意）

初始化时先令树根入队，随后进入循环。每一步迭代中，首先取出并访问队首节点，然后其左、右孩子（若存在）将顺序入队。一旦在试图进入下一迭代前发现队列为空，遍历即告完成。

图5.25以左侧二叉树为例，给出了层次遍历辅助队列从初始化到再次变空的演变过程。

■ 完全二叉树

反观代码5.20，在层次遍历算法的每一次迭代中，必有一个节点出队（而且不再入队），故累计恰好迭代n次。然而，每次迭代中入队节点的数目并不确定。若在对某棵二叉树的层次遍历过程中，前$\lfloor n/2 \rfloor$次迭代中都有左孩子入队，且前$\lceil n/2 \rceil - 1$次迭代中都有右孩子入队，则称之为完全二叉树（complete binary tree）。

图5.26给出了完全二叉树的实例，及其一般性的宏观拓扑结构特征：叶节点只能出现在最底部的两层，且最底层叶节点均处于次底层叶节点的左侧（习题[5-18]和[5-19]）。由此不难验证，高度为h的完全二叉树，规模应该介于2^h至$2^{h+1} - 1$之间；反之，规模为n的完全二叉树，高度$h = \lfloor \log_2 n \rfloor = O(\log n)$。另外，叶节点虽不致少于内部节点，但至多多出一个。以图5.26左侧的完全二叉树为例，高度h = 4；共有n = 20个节点，其中内部节点和叶节点各10个。

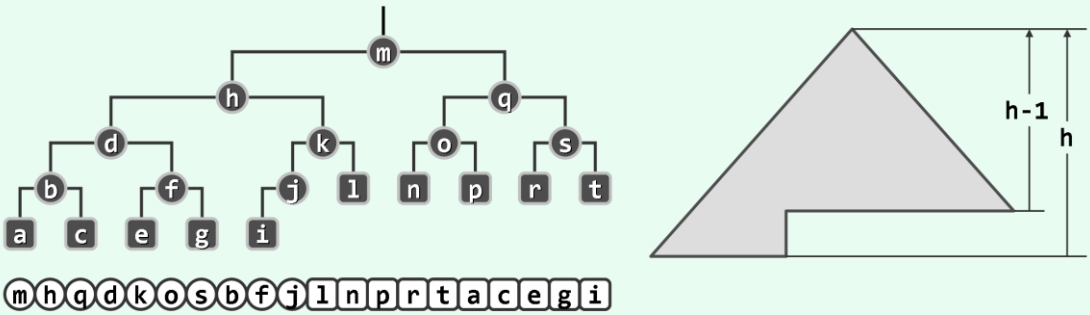


图5.26 完全二叉树实例及其宏观结构

得益于以上特性，完全二叉树可以借助向量结构，实现紧凑存储和高效访问（习题[5-20]）。

■ 满二叉树

完全二叉树的一种特例是，所有叶节点同处于最底层（非底层节点均为内部节点）。于是根据数学归纳法，每一层的节点数都应达到饱和，故将称其为满二叉树（full binary tree）。

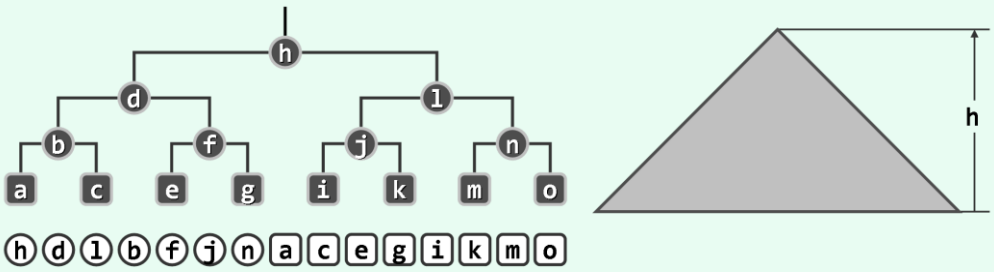


图5.27 满二叉树实例及其宏观结构

类似地不难验证，高度为h的满二叉树由$2^{h+1} - 1$个节点组成，其中叶节点总是恰好比内部节点多出一个。图5.27左侧即为一棵包含n = 15个节点、高度h = 3的满二叉树，其中叶节点8个，内部节点7个；右侧则给出了满二叉树的一般性宏观结构。

§5.5 Huffman编码

5.5.1 PFC编码及解码

以下基于二叉树结构，按照图5.28的总体框架，介绍PFC编码和解码算法的具体实现。

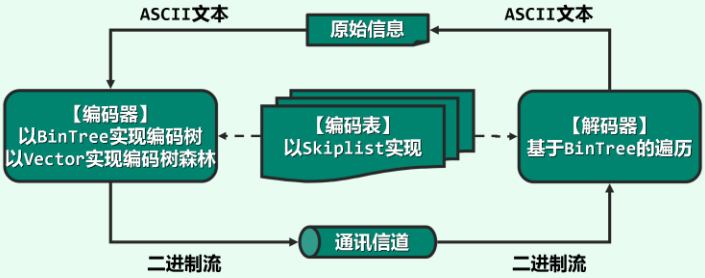


图5.28 为实现PFC编码和解码过程所需的数据结构和算法

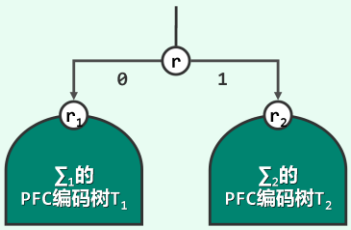


图5.29 子集的PFC编码树合并后，即是全集的一棵PFC编码树

如图5.29所示，若字符集Σ_1和Σ_2之间没有公共字符，且PFC编码方案分别对应于二叉树T_1和T_2，则通过引入一个根节点合并T_1和T_2之后所得到的二叉树，就是对应于$\Sigma_1 \cup \Sigma_2$的一种PFC编码方案。请注意，无论T_1和T_2的高度与规模是否相等，这一性质总是成立。

利用上述性质，可自底而上地构造PFC编码树。首先，由每一字符分别构造一棵单节点二叉树，并将它们视作一个森林。此后，反复从森林中取出两棵树并将其合二为一。如此，经$|\Sigma| - 1$步迭代之后，初始森林中的$|\Sigma|$棵树将合并成为一棵完整的PFC编码树。接下来，再将PFC编码树转译为编码表，以便能够根据待编码字符快捷确定与之对应的编码串。至此，对于任何待编码文本，通过反复查阅编码表，即可高效地将其转化为二进制编码串。

与编码过程相对应地，接收方也可以借助同一棵编码树来记录双方约定的编码方案。于是，每当接收到经信道传送过来的编码串后，（只要传送过程无误）接收方都可通过在编码树中反复从根节点出发做相应的漫游，依次完成对信息文本中各字符的解码。

■ 总体框架

以上编码和解码过程可描述为代码5.21，这也是同类编码、解码算法的统一测试入口。

```
1  int main ( int argc, char* argv[] ) { //PFC编码、解码算法统一测试入口
2     PFCForest* forest = initForest(); //初始化PFC森林
3     PFCTree* tree = generateTree ( forest ); release ( forest ); //生成PFC编码树
4     PFCTable* table = generateTable ( tree ); //将PFC编码树转换为编码表
5     for ( int i = 1; i < argc; i++ ) { //对于命令行传入的每一明文串
6        Bitmap codeString; //二进制编码串
7        int n = encode ( table, codeString, argv[i] ); //将根据编码表生成（长度为n）
8        decode ( tree, codeString, n ); //利用编码树，对长度为n的二进制编码串解码（直接输出）
9     }
10    release ( table ); release ( tree ); return 0; //释放编码表、编码树
11 }
```

代码5.21 基于二叉树的PFC编码

■ 数据结构的选取与设计

如代码5.22所示，这里使用向量实现PFC森林，其中各元素分别对应于一棵编码树；使用9.2节将要介绍的跳转表式词典结构实现编码表，其中的词条各以某一待编码字符为关键码，以对应的编码串为数据项；使用位图Bitmap（习题[2-34]）实现各字符的二进制编码串。

```
/*****************************************************************************************
 * PFC编码使用的数据结构
 *****************************************************************************************/
#include "../BinTree/BinTree.h" //用BinTree实现PFC树
typedef BinTree<char> PFCTree; //PFC树

#include "../Vector/Vector.h" //用Vector实现PFC森林
typedef Vector<PFCTree*> PFCForest; //PFC森林

#include "../Bitmap/Bitmap.h" //使用位图结构实现二进制编码串
#include "../Skiplist/Skiplist.h" //引入Skiplist式词典结构实现
typedef Skiplist<char, char*> PFCTable; //PFC编码表，词条格式为：(key = 字符, value = 编码串)

#define  N_CHAR  (0x80 - 0x20) //只考虑可打印字符
```

代码5.22 实现PFC编码所需的数据结构

以下，分别给出各功能部分的具体实现，请读者对照注解自行分析。

■ 初始化PFC森林

```
PFCForest* initForest() { //PFC编码森林初始化
   PFCForest* forest = new PFCForest; //首先创建空森林，然后
   for ( int i = 0; i < N_CHAR; i++ ) { //对每一个可打印字符[0x20, 0x80)
      forest->insert ( i, new PFCTree() ); //创建一棵对应的PFC编码树，初始时其中
      ( *forest ) [i]->insertAsRoot ( 0x20 + i ); //只包含对应的一个（叶、根）节点
   }
   return forest; //返回包含N_CHAR棵树的森林，其中每棵树各包含一个字符
}
```

代码5.23 初始化PFC森林

■ 构造PFC编码树

```
PFCTree* generateTree ( PFCForest* forest ) { //构造PFC树
   srand ( ( unsigned int ) time ( NULL ) ); //这里将随机取树合并，故先设置随机种子
   while ( 1 < forest->size() ) { //共做|forest|-1次合并
      PFCTree* s = new PFCTree; s->insertAsRoot ( '^' ); //创建新树（根标记为"^"）
      Rank r1 = rand() % forest->size(); //随机选取r1，且
      s->attachAsLC ( s->root(), ( *forest ) [r1] ); //作为左子树接入后
      forest->remove ( r1 ); //随即剔除
      Rank r2 = rand() % forest->size(); //随机选取r2，且
```

```
9        s->attachAsRC ( s->root(), ( *forest ) [r2] ); //作为右子树接入后
10       forest->remove ( r2 ); //随即剔除
11       forest->insert ( forest->size(), s ); //合并后的PFC树重新植入PFC森林
12    }
13    return ( *forest ) [0]; //至此，森林中尚存的最后一棵树，即全局PFC编码树
14 }
```

代码5.24 构造PFC编码树

■ 生成PFC编码表

```
1  void generateCT //通过遍历获取各字符的编码
2  ( Bitmap* code, int length, PFCTable* table, BinNodePosi ( char ) v ) {
3     if ( IsLeaf ( *v ) ) //若是叶节点
4        { table->put ( v->data, code->bits2string ( length ) ); return; }
5     if ( HasLChild ( *v ) ) //Left = 0
6        { code->clear ( length ); generateCT ( code, length + 1, table, v->lc ); }
7     if ( HasRChild ( *v ) ) //right = 1
8        { code->set ( length ); generateCT ( code, length + 1, table, v->rc ); }
9  }
10
11 PFCTable* generateTable ( PFCTree* tree ) { //构造PFC编码表
12    PFCTable* table = new PFCTable; //创建以Skiplist实现的编码表
13    Bitmap* code = new Bitmap; //用于记录RPS的位图
14    generateCT ( code, 0, table, tree->root() ); //遍历以获取各字符（叶节点）的RPS
15    release ( code ); return table; //释放编码位图，返回编码表
16 }
```

代码5.25 生成PFC编码表

■ 编码

```
1  int encode ( PFCTable* table, Bitmap& codeString, char* s ) { //PFC编码算法
2     int n = 0;
3     for ( size_t m = strlen ( s ), i = 0; i < m; i++ ) { //对于明文s[]中的每个字符
4        char** pCharCode = table->get ( s[i] ); //取出其对应的编码串
5        if ( !pCharCode ) pCharCode = table->get ( s[i] + 'A' - 'a' ); //小写字母转为大写
6        if ( !pCharCode ) pCharCode = table->get ( ' ' ); //无法识别的字符统一视作空格
7        printf ( "%s", *pCharCode ); //输出当前字符的编码
8        for ( size_t m = strlen ( *pCharCode ), j = 0; j < m; j++ ) //将当前字符的编码接入编码串
9           '1' == * ( *pCharCode + j ) ? codeString.set ( n++ ) : codeString.clear ( n++ );
10    }
11    return n; //二进制编码串记录于codeString中，返回编码串总长
12 }
```

代码5.26 PFC编码

■ 解码

```
13 void decode ( PFCTree* tree, Bitmap& code, int n ) { //PFC解码算法
14    BinNodePosi ( char ) x = tree->root(); //根据PFC编码树
15    for ( int i = 0; i < n; i++ ) { //将编码（二进制位图）
16       x = code.test ( i ) ? x->rc : x->lc; //转译为明码并
17       if ( IsLeaf ( *x ) ) { printf ( "%c", x->data ); x = tree->root(); } //打印输出
18    }
19 }
```

代码5.27 PFC解码

■ 优化

在介绍过PFC及其实现方法后，以下将就其编码效率做一分析，并设计出更佳的编码方法。

同样地，我们依然暂且忽略硬件成本和信道误差等因素，而主要考查如何高效率地完成文本信息的编码和解码。不难理解，在计算资源固定的条件下，不同编码方法的效率主要体现于所生成二进制编码串的总长，或者更确切地，体现于二进制码长与原始文本长度的比率。

那么，面对千变万化、长度不一的待编码文本，从总体上我们应该按照何种尺度来衡量这一因素呢？基于这一尺度，又该应用哪些数据结构来实现相关的算法呢？

5.5.2 最优编码树

在实际的通讯系统中，信道的使用效率是个很重要的问题，这在很大程度上取决于编码算法本身的效率。比如，高效的编码算法生成的编码串应该尽可能地短。那么，如何做到这一点呢？在什么情况下能够做到这一点呢？以下首先来看如何对编码长度做“度量”。

■ 平均编码长度与叶节点平均深度

由5.2.2节的结论，字符x的编码长度|rps(x)|就是其对应叶节点的深度depth(v(x))。于是，各字符的平均编码长度就是编码树T中各叶节点的平均深度（average leaf depth）：

$$ald(T) = \sum_{x\in\Sigma} |rps(x)| / |\Sigma| = \sum_{x\in\Sigma} depth(x) / |\Sigma|$$

以如图5.9(a)所示编码树为例，字符'A'、'E'和'G'的编码长度为2，'M'和'S'的编码长度为3，故该编码方案的平均编码长度为：

$$ald(T) = (2 + 2 + 2 + 3 + 3) / 5 = 2.4$$

既然ald(T)值是反映编码效率的重要指标，我们自然希望这一指标尽可能地小。

■ 最优编码树

同一字符集的所有编码方案中，平均编码长度最小者称作最优方案；对应编码树的ald()值也达到最小，故称之为最优二叉编码树，简称最优编码树（optimal encoding tree）。

对于任一字符集Σ，深度不超过|Σ|的编码树数目有限，故在其中ald()值最小者必然存在。需注意的是，最优编码树不见得唯一（比如，同层节点互换位置后，并不影响全树的平均深度），但从工程的角度看，任取其中一棵即可。

为导出最优编码树的构造算法，以下需从更为深入地了解最优编码树的性质入手。

■ 双子性

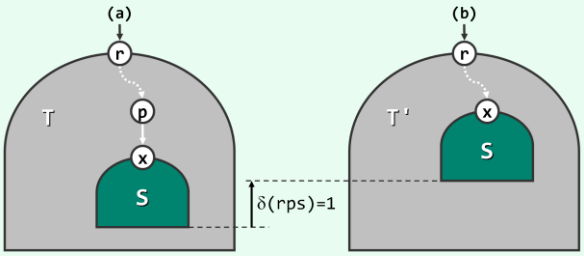


图5.30 最优编码树的双子性

首先，最优二叉编码树必为真二叉树：内部节点的左、右孩子全双（习题[5-2]）。

若不然，如图5.30(a)所示假设在某棵最优二叉编码树T中，内部节点p拥有唯一的孩子x。于是如图(b)，此时只需将节点p删除并代之以子树x，即可得到原字符集的另一棵编码树T'。

不难看出，除了子树x中所有叶节点的编码长度统一缩短1层之外，其余叶节点的编码长度不变，因此相对于T，T'的平均编码长度必然更短——这与T的最优性矛盾。

■ 层次性

最优编码树中，叶节点位置的选取有严格限制——其深度之差不得超过1。

为证明这一重要特性，可如图5.31(a)假设，某棵最优二叉编码树T含有深度之差不小于2的一对叶节点x和y。不失一般性设x更深，并令p为x的父亲。于是由双子性，作为内部节点的p必然还有另一孩子q。

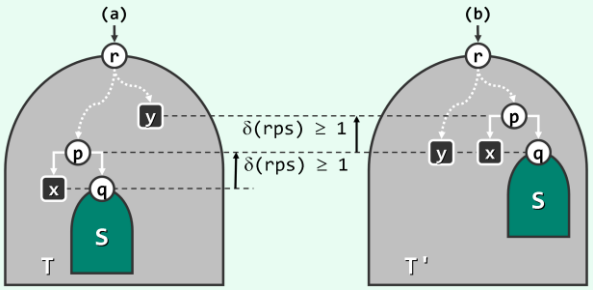


图5.31 最优编码树的层次性

如图(b)所示，令叶节点y与子树p互换位置，从而得到一棵新树T'。易见，T'依然是原字符集的一棵二叉编码树。更重要的是，就深度而言，除了x、y以及子树q中的叶节点外，其余叶节点均保持不变。其中，x的提升量与y的下降量相互抵消，而子树q中的叶节点都至少提升1层。因此相对于T，T'的平均编码长度必然更短——这与T的最优性矛盾。

以上的节点位置互换是一种十分重要的技巧，藉此可从任一编码树出发，不断提高编码效率，直至最优。以图5.32为例，对同一字符集Σ = { 'A', 'I', 'M', 'N' }，左、右两棵编码树的ald()值均为9/4，而经一次交换转换为居中的编码树后，ald()值均降至8/4。

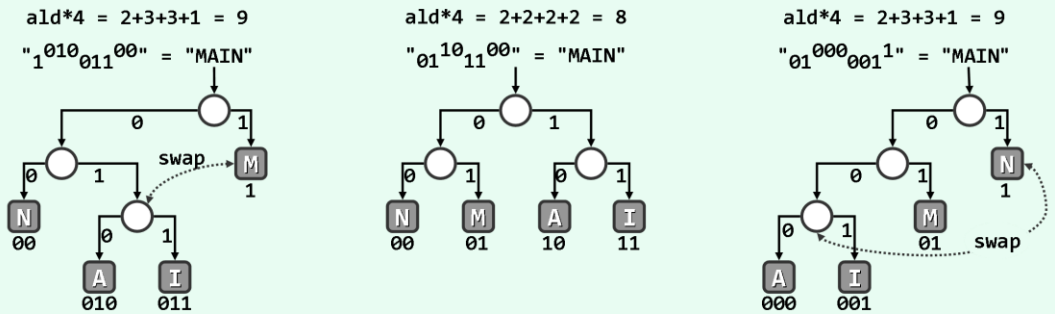


图5.32 通过节点交换提高编码效率完全二叉树与满二叉树

■ 最优编码树的构造

由上可知，最优编码树中的叶节点只能出现于最低两层，故这类树的一种特例就是真完全树。

由此，可以直接导出如下构造最优编码树的算法：创建一棵规模为2|Σ| - 1的完全二叉树T，再将Σ中的字符任意分配给T的|Σ|个叶节点。

仍以字符集 Σ = { 'A', 'E', 'G', 'M', 'S' } 为例，只需创建包含2 × 5 - 1 = 9个节点的一棵完全二叉树，并将各字符分配至5个叶节点，即得到一棵最优编码树。再适当交换同深度的节点，即可得到如116页图5.9(a)所示的编码树——由于此类节点交换并不改变平均编码长度，故它仍是一棵最优编码树。

5.5.3 Huffman编码树

■ 字符出现概率

以上最优编码树算法的实际应用价值并不大，除非Σ中各字符在文本串中出现的次数相等。遗憾的是，这一条件往往并不满足，甚至不确定。为此，需要从待编码的文本中取出若干样本，通过统计各字符在其中出现的次数（亦称作字符的频率），估计各字符实际出现的概率。

当然，每个字符x都应满足p(x) ≥ 0，且同一字符集Σ中的所有字符满足$\Sigma_{x\in\Sigma}$p(x) = 1。

表5.5 在一篇典型的英文文章中，各字母出现的次数

字符	A	B	C	D	E	F	G	H	I	J	K	L	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
次数	623	99	239	290	906	224	136	394	600	5	56	306	586	622	148	10	465	491	732	214	76	164	16	139	13

实际上，多数应用所涉及的字符集Σ中，各字符的出现频率不仅极少相等或相近，而且往往相差悬殊。以如表5.5所示的英文字符集为例，'e'、't'等字符的出现频率通常是'z'、'j'等字符的数百倍。这种情况下，应该从另一角度更为准确地衡量平均编码长度。

■ 带权平均编码长度与叶节点带权平均深度

若考虑字符各自的出现频率，则可将带权平均编码长度取作编码树T的叶节点带权平均深度（weighted average leaf depth），亦即：

$$wald(T) = \sum_{x\in\Sigma} p(x)\cdot|rps(x)|$$

以字符集Σ = { 'A', 'I', 'M', 'N' }为例，若各字符出现的概率依次为2/6、1/6、2/6和1/6（比如文本串"MAMANI"），则按照图5.33的编码方案，各字符对wald(T)的贡献分别为：

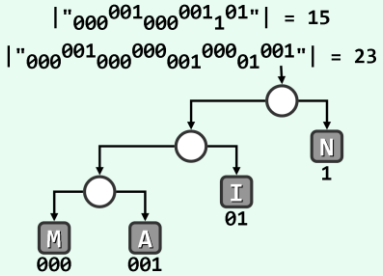


图5.33 考虑字符出现频率，以平均带权深度衡量编码效率

$$3\times(2/6) = 1;\quad 2\times(1/6) = 1/3;\quad 3\times(2/6) = 1;\quad 1\times(1/6) = 1/6$$

相应地，这一编码方案对应的平均带权深度就是：

$$wald(T) = 1 + 1/3 + 1 + 1/6 = 2.5$$

若各字符出现的概率依次为3/8、1/8、4/8和0/8（比如文本串"MAMMAMIA"），则有

$$wald(T) = 3\times(3/8) + 2\times(1/8) + 3\times(4/8) + 1\times(0/8) = 2.875$$

■ 完全二叉编码树 ≠ wald()最短

那么，wald()值能否进一步降低呢？仍然以Σ = { 'A', 'I', 'M', 'N' }为例。

我们首先想到的是前节提到的完全二叉编码树。如图5.34所示，由于此时各字符的编码长度都是2，故无论其出现概率具体分布如何，其对应的平均带权深度都将为2。

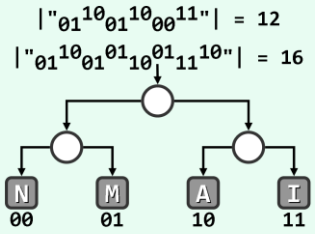


图5.34 若考虑出现频率，完全二叉树或满树未必最优

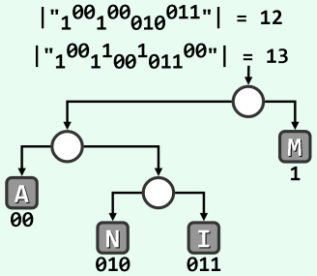


图5.35 若考虑出现频率，最优编码树往往不是完全二叉树

然而，在考虑各字符出现概率的不同之后，某些非完全二叉编码树的wald()值却可能更小。以图5.35的二叉编码树为例，当各字符频率与"MAMANI"相同时，wald(T) = 2，与图5.34方案相当；但当字符频率与"MAMMAMIA"相同时，wald(T) = 1.625，反而更优。

■ 最优带权编码树

若字符集Σ中各字符的出现频率分布为p()，则wald()值最小的编码方案称作Σ（按照p()分布的）的最优带权编码方案，对应的编码树称作最优带权编码树。当然，与不考虑字符出现概率时同理，此时的最优带权编码树也必然存在（尽管通常并不唯一）。

为得出最优带权编码树的构造算法，以下还是从分析其性质入手。一方面不难验证，此时的最优编玛树依然必须满足双子性。另一方面，尽管最优编玛树不一定仍是完全的（比如在图5.35中，叶节点的深度可能相差2层以上），却依然满足某种意义上的层次性。

■ 层次性

具体地，若字符x和y的出现概率在所有字符中最低，则必然存在某棵最优带权编码树，使x和y在其中同处于最底层，且互为兄弟。为证明这一重要特性，如图5.36(a)所示任取一棵最优带权编码树T。根据双子性，必然可以在最低层找到一对兄弟节点a和b。不妨设它们不是x和y。

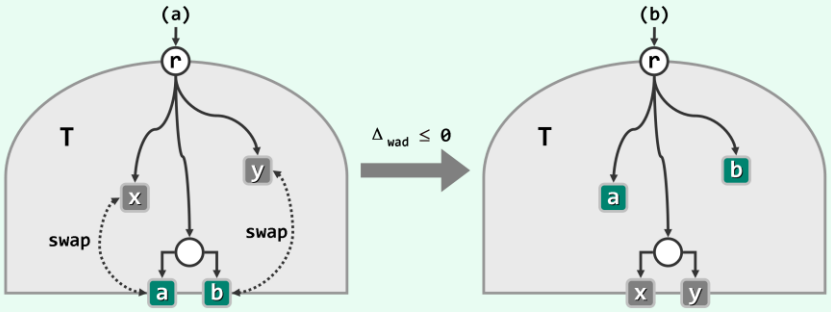


图5.36 最优编码树的层次性

现在，交换a和x，再交换b和y，从而得到该字符集的另一编码树T'，x和y成为其中最低层的一对兄弟。因字符x和y权重最小，故如此交换之后，wald(T')不致增加。

于是根据T的最优性，T'必然也是一棵最优编码树。

5.5.4 Huffman编码算法

■ 原理与构思

设字符x和y在Σ中的出现概率最低。考查另一字符集Σ' = (Σ\{x, y}) ∪ {z}，其中新增

字符z的出现概率取作被剔除字符x和y之和，即p(z) = p(x) + p(y)，其余字符的概率不变。任取Σ'的一棵最优带权编码树T'，于是根据层次性，只需将T'中与字符z对应的叶节点替换为内部节点，并在其下引入分别对应于x和y的一对叶节点，即可得到Σ的一棵最优带权编码树。

仍以142页图5.35中字符集Σ = { 'A', 'I', 'N', 'M' }为例，设各字符出现的频率与编码串'MAMMAMIA"吻合，则不难验证，图5.37左侧即为Σ的一棵最优带权编码树T。

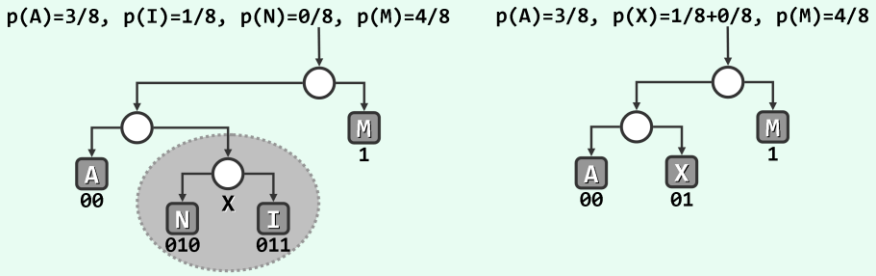


图5.37 最优编码树中底层兄弟节点合并后，依然是最优编码树

现在，将其中出现频率最低的'N'和'I'合并，代之以新字符'X'，且令'X'的出现频率为二者之和 1/8 + 0/8 = 1/8 。若T中也做相应的调整之后，则可得图5.37右侧所示的编码树T'。不难验证，T'是新字符集Σ' = { 'A', 'X', 'M' }的一棵最优带权编码树；反之，在T'中将'X'拆分为'N'和'I'后，亦是Σ的一棵最优带权编码树（习题[5-28]）。

■ 策略与算法

因此，对于字符出现概率已知的任一字符集Σ，都可采用如下算法构造其对应的最优带权编码树：首先，对应于Σ中的每一字符，分别建立一棵单个节点的树，其权重取作该字符的频率，这|Σ|棵树构成一个森林𝓕。接下来，从𝓕中选出权重最小的两棵树，创建一个新节点，并分别以这两棵树作为其左、右子树，如此将它们合并为一棵更高的树，其权重取作二者权重之和。实际上，此后可以将合并后的新树等效地视作一个字符，称作超字符。

这一选取、合并的过程反复进行，每经过一轮迭代，𝓕中的树就减少一棵。当最终𝓕仅包含一棵树时，它就是一棵最优带权编码树，构造过程随即完成。

以上构造过程称作Huffman编码算法[②]，由其生成的编码树称作Huffman编码树（Huffman encoding tree）。需再次强调的是，Huffman编码树只是最优带权编码树中的一棵。

■ 实例

表5.6 由6个字符构成的字符集Σ，以及各字符的出现频率

字符	A	B	C	D	E	F
频率	623	99	239	290	906	224

考查表5.6所列由6各字符构成的字符集Σ。为构造与之相对应的一棵Huffman编码树，在初始化之后共需经过5步迭代，具体过程如图5.38(a~f)所示。

② 由David A. Huffman于1952年发明

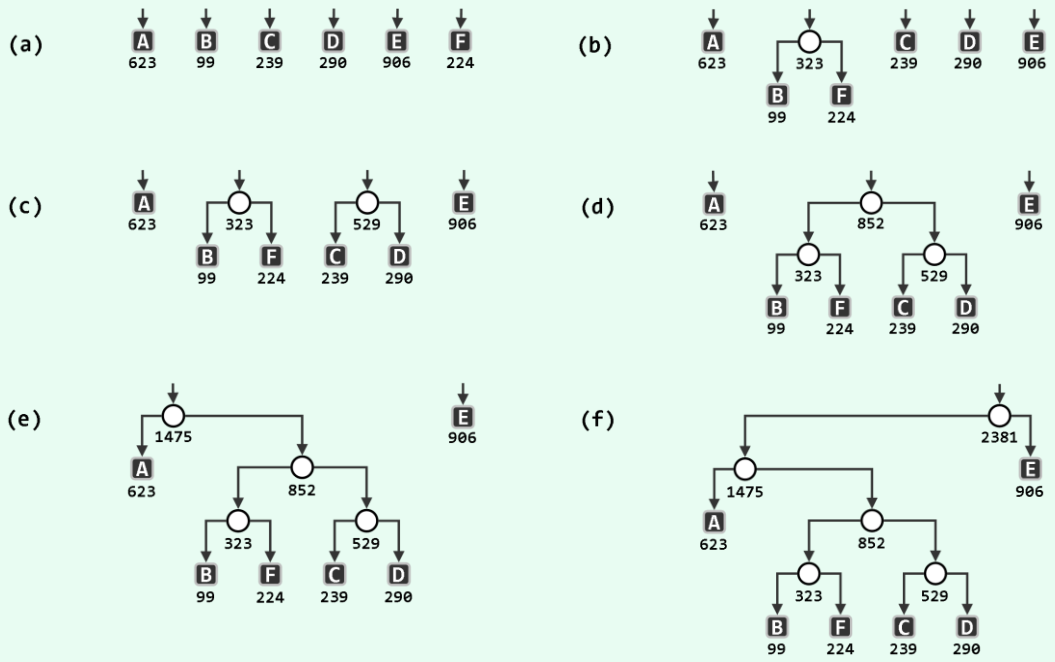


图5.38 Huffman树构造算法实例

请注意，以上构造过程不见得是确定的，在选取、合并子树时都可能出现歧义。幸运的是，这一问题不难解决（习题[5-29]）。

以下，我们分别介绍Huffman编码算法各个环节的具体实现。

■ 总体框架

以上编码和解码过程可描述为代码5.28，这也是同类编码、解码算法的统一测试入口。

```
/*****************************************************************************************
 * 无论编码森林由列表、完全堆还是左式堆实现，本测试过程都可适用
 * 编码森林的实现方式采用优先级队列时，编译前对应的工程只需设置相应标志：
 *    DSA_PQ_List、DSA_PQ_ComplHeap或DSA_PQ_LeftHeap
 *****************************************************************************************/
int main ( int argc, char* argv[] ) { //Huffman编码算法统一测试
   int* freq = statistics ( argv[1] ); //根据样本文件，统计各字符的出现频率
   HuffForest* forest = initForest ( freq ); release ( freq ); //创建Huffman森林
   HuffTree* tree = generateTree ( forest ); release ( forest ); //生成Huffman编码树
   HuffTable* table = generateTable ( tree ); //将Huffman编码树转换为编码表
   for ( int i = 2; i < argc; i++ ) { //对于命令行传入的每一明文串
      Bitmap* codeString = new Bitmap; //二进制编码串
      int n = encode ( table, codeString, argv[i] ); //将根据编码表生成（长度为n）
      decode ( tree, codeString, n ); //利用Huffman编码树，对长度为n的二进制编码串解码
      release ( codeString );
   }
```

```
17     release ( table ); release ( tree ); return 0; //释放编码表、编码树
18 }
```

代码5.28 基于二叉树的Huffman编码

■ （超）字符

如前所述，无论是输入的字符还是合并得到的超字符，在构造Huffman编码树过程中都可等效地加以处理——就其本质而言，相关信息无非就是对应的字符及其出现频率。

```
1 #define  N_CHAR  (0x80 - 0x20) //仅以可打印字符为例
2 struct HuffChar { //Huffman（超）字符
3     char ch; int weight; //字符、频率
4     HuffChar ( char c = '^', int w = 0 ) : ch ( c ), weight ( w ) {};
5 // 比较器、判等器（各列其一，其余自行补充）
6     bool operator< ( HuffChar const& hc ) { return weight > hc.weight; } //此处故意大小颠倒
7     bool operator== ( HuffChar const& hc ) { return weight == hc.weight; }
8 };
```

代码5.29 HuffChar结构

因此如代码5.29所示，可相应地定义一个HuffChar类。对于经合并生成的超字符，这里统一用'^'表示，同时其权重weight设为被合并字符的权重之和。作为示例，这里字符集取ASCII字符集在[0x20, 0x80)区间内的子集，包含所有可打印字符。

另外，为便于超字符做权重的比较和判等，这里还重载了相关的操作符。

■ 数据结构的选取与设计

如代码5.30所示，可借助BinTree模板类定义Huffman编码树类型HuffTree。

```
1 #define HuffTree BinTree<HuffChar> //Huffman树，由BinTree派生，节点类型为HuffChar
```

代码5.30 Huffman编码树结构

如代码5.31所示，可借助List模板类定义Huffman森林类型HuffForest。

```
1 #include "../List/List.h" //用List实现
2 typedef List<HuffTree*> HuffForest; //Huffman森林
```

代码5.31 Huffman森林结构

如代码5.32所示，可借助位图类Bitmap（习题[2-34]）定义Huffman二进制编码串类型HuffCode。

```
1 #include "../Bitmap/Bitmap.h" //基于Bitmap实现
2 typedef Bitmap HuffCode; //Huffman二进制编码
```

代码5.32 Huffman二进制编码串

作为PFC编码表的一种，Huffman编码表与代码5.22一样，自然可以由跳转表实现。作为对后面第9章中词典结构的统一测试，这里选择了与跳转表接口相同的散列表结构（9.3节），并基于该结构实现HuffTable类型。

```
#include "../Hashtable/Hashtable.h" //用HashTable实现
typedef Hashtable<char, char*> HuffTable; //Huffman编码表
```

代码5.33 Huffman编码表

如代码5.33所示，可以9.3节将要介绍的Hashtable结构来实现HuffTable。其中，词条的关键码key（即带编码的字符）为字符类型char，数值value（即字符对应的二进制编码串）为字符串类型char*。

■ 字符出现频率的样本统计

如代码5.34所示，这里通过对样例文本的统计，对各字符的出现频率做出估计。

```
int* statistics ( char* sample_text_file ) { //统计字符出现频率
   int* freq = new int[N_CHAR];  //以下统计需随机访问，故以数组记录各字符出现次数
   memset ( freq, 0, sizeof ( int ) * N_CHAR ); //清零
   FILE* fp = fopen ( sample_text_file, "r" ); //assert: 文件存在且可正确打开
   for ( char ch; 0 < fscanf ( fp, "%c", &ch ); ) //逐个扫描样本文件中的每个字符
      if ( ch >= 0x20 ) freq[ch - 0x20]++; //累计对应的出现次数
   fclose ( fp ); return freq;
}
```

代码5.34 Huffman算法：字符出现频率的样本统计

为方便统计过程的随机访问，这里使用了数组freq。样例文件（假设存在且可正常打开）的路径作为函数参数传入。文件打开后顺序扫描，并累计各字符的出现次数。

■ 初始化Huffman森林

```
HuffForest* initForest ( int* freq ) { //根据频率统计表，为每个字符创建一棵树
   HuffForest* forest = new HuffForest; //以List实现的Huffman森林
   for ( int i = 0; i < N_CHAR; i++ ) { //为每个字符
      forest->insertAsLast ( new HuffTree ); //生成一棵树，并将字符及其频率
      forest->last()->data->insertAsRoot ( HuffChar ( 0x20 + i, freq[i] ) ); //存入其中
   }
   return forest;
}
```

代码5.35 初始化Huffman森林

■ 构造Huffman编码树

```
HuffTree* minHChar ( HuffForest* forest ) { //在Huffman森林中找出权重最小的（超）字符
   ListNodePosi ( HuffTree* ) p = forest->first(); //从首节点出发查找
   ListNodePosi ( HuffTree* ) minChar = p; //最小Huffman树所在的节点位置
   int minWeight = p->data->root()->data.weight; //目前的最小权重
   while ( forest->valid ( p = p->succ ) ) //遍历所有节点
      if ( minWeight > p->data->root()->data.weight ) //若当前节点所含树更小，则
         {  minWeight = p->data->root()->data.weight; minChar = p;  } //更新记录
```

```
8     return forest->remove ( minChar ); //将挑选出的Huffman树从森林中摘除，并返回
9  }
10
11 HuffTree* generateTree ( HuffForest* forest ) { //Huffman编码算法
12    while ( 1 < forest->size() ) {
13       HuffTree* T1 = minHChar ( forest ); HuffTree* T2 = minHChar ( forest );
14       HuffTree* S = new HuffTree();
15       S->insertAsRoot ( HuffChar ( '^', T1->root()->data.weight + T2->root()->data.weight ) );
16       S->attachAsLC ( S->root(), T1 ); S->attachAsRC ( S->root(), T2 );
17       forest->insertAsLast ( S );
18    } //assert: 循环结束时，森林中唯一（列表首节点中）的那棵树即Huffman编码树
19    return forest->first()->data;
20 }
```

代码5.36 构造Huffman编码树

根据以上的构思，generateTree()实现为一个循环迭代的过程。如代码5.36所示，每一步迭代都通过调用minHChar()，从当前的森林中找出权值最小的一对超字符，将它们合并为一个更大的超字符，并重新插入森林。每迭代一次，森林的规模即减一，故共需迭代n - 1次，直到只剩一棵树。minHChar()每次都要遍历森林中所有的超字符（树），所需时间线性正比于当时森林的规模。因此总体运行时间应为：

$$\mathcal{O}(n) + \mathcal{O}(n - 1) + \ldots + \mathcal{O}(2) = \mathcal{O}(n^2)$$

■ 生成Huffman编码表

```
1  static void //通过遍历获取各字符的编码
2  generateCT ( Bitmap* code, int length, HuffTable* table, BinNodePosi ( HuffChar ) v ) {
3     if ( IsLeaf ( *v ) ) //若是叶节点（还有多种方法可以判断）
4        {  table->put ( v->data.ch, code->bits2string ( length ) ); return;  }
5     if ( HasLChild ( *v ) ) //Left = 0
6        { code->clear ( length ); generateCT ( code, length + 1, table, v->lc ); }
7     if ( HasRChild ( *v ) ) //Right = 1
8        { code->set ( length ); generateCT ( code, length + 1, table, v->rc ); }
9  }
10
11 HuffTable* generateTable ( HuffTree* tree ) { //将各字符编码统一存入以散列表实现的编码表中
12    HuffTable* table = new HuffTable; Bitmap* code = new Bitmap;
13    generateCT ( code, 0, table, tree->root() );
14    release ( code ); return table;
15 };
```

代码5.37 生成Huffman编码表

■ 编码

```
1  // 按编码表对Bitmap串做Huffman编码
```

```
int encode ( HuffTable* table, Bitmap* codeString, char* s ) {
   int n = 0; //待返回的编码串总长n
   for ( size_t m = strlen ( s ), i = 0; i < m; i++ ) { //对于明文中的每个字符
      char** pCharCode = table->get ( s[i] ); //取出其对应的编码串
      if ( !pCharCode ) pCharCode = table->get ( s[i] + 'A' - 'a' ); //小写字母转为大写
      if ( !pCharCode ) pCharCode = table->get ( ' ' ); //无法识别的字符统一视作空格
      printf ( "%s", *pCharCode ); //输出当前字符的编码
      for ( size_t m = strlen ( *pCharCode ), j = 0; j < m; j++ ) //将当前字符的编码接入编码串
         '1' == * ( *pCharCode + j ) ? codeString->set ( n++ ) : codeString->clear ( n++ );
   }
   printf ( "\n" ); return n;
} //二进制编码串记录于位图codeString中
```

代码5.38 Huffman编码

■ 解码

```
// 根据编码树对长为n的Bitmap串做Huffman解码
void decode ( HuffTree* tree, Bitmap* code, int n ) {
   BinNodePosi ( HuffChar ) x = tree->root();
   for ( int i = 0; i < n; i++ ) {
      x = code->test ( i ) ? x->rc : x->lc;
      if ( IsLeaf ( *x ) ) {  printf ( "%c", x->data.ch ); x = tree->root();  }
   }
} //解出的明码，在此直接打印输出；实用中可改为根据需要返回上层调用者
```

代码5.39 Huffman解码