第5章

二叉树

[5-1] 考查任何一棵二叉树 T。

a) 试证明，对于其中任一节点 v∈T，总有 depth(v) + height(v) ≤ height(T)；

【解答】

对于子树v中的任一节点x，我们将x在该子树中的深度记作$depth_v(x)$。

若将树根节点记作r，则根据定义有：

$$height(v) = \max\{ depth_v(x) \mid x \in subtree(v) \}$$

于是，

$$\begin{aligned} & depth(v) + height(v) \\ = \; & depth(v) + \max\{ depth_v(x) \mid x \in subtree(v) \} \\ = \; & \max\{ depth(v) + depth_v(x) \mid x \in subtree(v) \} \\ = \; & \max\{ depth(x) \mid x \in subtree(v) \} \quad \cdots\cdots (1) \end{aligned}$$

另一方面，

$$\begin{aligned} & height(T) = height(r) \\ = \; & \max\{ depth_r(x) \mid x \in subtree(r) = T \} \\ = \; & \max\{ depth(x) \mid x \in subtree(r) = T \} \quad \cdots\cdots (2) \end{aligned}$$

对比(1)、(2)两式，二者都是取depth(x)的最大值，但前者所覆盖的范围（子树v）是后者所覆盖范围（全树）的子集，因此前者必然不大于后者。

b) 以上取等号的充要条件是什么？

【解答】

由以上分析可见，唯有在全树最深节点属于子树v时，a）中不等式方可取等号；反之亦然。实际上，此时对应的充要条件是：全树最深（叶）节点（之一）是节点v的后代，或者等价地，节点v是全树（某一）最深（叶）节点的祖先。

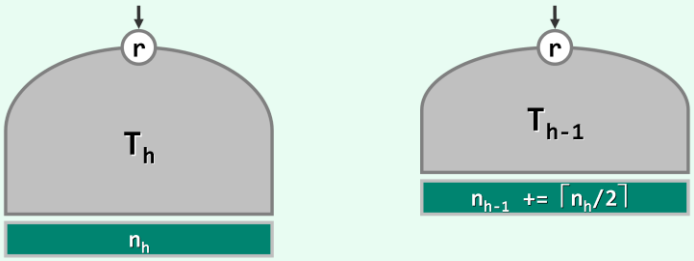
[5-2] 考查任何一棵高度为 h 的二叉树 T，设其中深度为 k 的叶节点有 n_k 个，$0 \le k \le h$。

a) 试证明：$\sum_{k=0}^{h} (n_k / 2^k) \le 1$；

【解答】

采用数学归纳法，对树高h做归纳。在h = 0时，该不等式对单节点的二叉树显然成立。故假定对于高度小于h的二叉树，该不等式均成立，以下考查高度为h的二叉树。

该树在最底层拥有恰好n_h个（叶）节点。如图x5.1所示，若将它们统一删除，则只可能在原次底层（现最底层）增加叶节点，其余更高层的叶节点不增不减。准确地说，若原次底层新增叶节点共计m个，则必有$m \ge \lceil n_h/2 \rceil$；或者反过来等价地，$n_h \le 2m$——取等号当且仅当$n_h$为偶数。



图x5.1 n_h个底层（叶）节点删除后，（次底层叶）节点至少增加$\lceil n_h/2 \rceil$个

经如此统一删除底层（叶）节点之后，所得二叉树的高度为h - 1，故由归纳假设应当满足：

$$\sum_{k=0}^{h-2}(n_k/2^k) + (n_{h-1} + m)/2^{h-1} = \sum_{k=0}^{h-1}(n_k/2^k) + m/2^{h-1} \le 1$$

相应地，对于原树而言应有：

$$\begin{aligned}\sum_{k=0}^{h}(n_k/2^k) &= \sum_{k=0}^{h-1}(n_k/2^k) + n_h/2^h \\ &\le \sum_{k=0}^{h-1}(n_k/2^k) + 2m/2^h \quad \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots (*) \\ &= \sum_{k=0}^{h-1}(n_k/2^k) + m/2^{h-1} \\ &\le 1\end{aligned}$$

b） 以上不等式取等号的充要条件是什么？

【解答】

由以上证明过程可见，题中不等式若欲取等号，则在（沿数学归纳反向）每一步递推中，以上不等式(*)都应取等号，而其充要条件是$n_h = 2m$。换而言之，二叉树每次增加一层，最底层的新叶节点都应该是成对引入的。递推地应用这一规则即不难看出，在如此生成的二叉树中，所有节点的度数必为偶数——亦即所谓的真二叉树（proper binary tree）。

[5-3] 试证明，在二叉树中接入（attachAsLC()或attachAsRC()）或摘除（remove()或secede()）一棵非空子树之后

a） 该子树所有祖先的后代数目（size）必然变化；

【解答】

“祖先-后代”关系是相对的，故这一结论显然成立。

b） 该子树所有祖先的高度（height）可能变化；

【解答】

同样显然成立。

一般地，若在接入/摘除某棵子树之后，某个祖先节点的高度的确因此而增加/降低，则在此时/此前，该祖先节点的最深后代必然都来自于该子树。

c） 对于非该子树祖先的任何节点，高度与后代数目均保持不变。

【解答】

同样地，由“祖先-后代”关系的相对性，后代集合及其总数显然不变。对于这样的节点而言，通往其（包括最深后代在内的）所有后代的通路既然不变，故高度亦保持不变。

[5-4] 考查如教材 121 页代码 5.6 所示的 BinTree::updateHeightAbove(x)算法。

a） 试证明，在逆行向上依次更新 x 各祖先高度的过程中，一旦发现某一祖先的高度没有发生变化，算法即可提前终止；

【解答】

在高度更新过程中，将首个高度不变的节点记作C。

考查任一更高的祖先节点A。若从A通往其最深后代的通路不经过C，则A的高度自然不变。否则，该通路自上而下经过C之后，必然会继续通往C的最深后代。这种情况下，既然C的高度保持不变，则该后代的深度必然也不变——尽管这种后代节点可能不止一个。

b） 试按此思路改进这一算法；

【解答】

在自底而上逐层更新的过程中，一旦当前祖先节点的高度未变，即可立即终止。

c） 如此改进之后，算法的渐进复杂度是否会相应地降低？为什么？

【解答】

在最坏情况下我们仍需一直更新到树根节点，因此就渐进意义而言算法的复杂度并未降低。当然，如此改进之后毕竟可以自适应地减少不必要的更新计算，因此这种改进依然是值得的。

[5-5] 教材 123 页代码 5.9 中的 removeAt()算法，时间复杂度是多少？空间呢？

【解答】

对于待删除子树中的每一节点，该算法都有一个对应的递归实例；反之，算法运行期间出现过的每一递归实例，也唯一对应于某一节点。再注意到，每个递归实例均只需常数时间，故知整体的运行时间应线性正比于待删除子树的规模。

在算法运行过程中的任何时刻，递归调用栈中各帧所对应的节点，自底而上两两构成“父亲-孩子”关系——比如特别地，最底部的一帧对应于（子树的）根节点。而当递归调用栈达到最高时，栈顶一帧必然对应于（子树中的）最深节点。由此可见，该算法的空间复杂度应线性正比于待删除子树的高度。

[5-6] 试证明，若采用 PFC 编码，则无论二进制编码串的长度与内容如何，解码过程总能持续进行——只有最后一个字符的解码可能无法完成。

【解答】

按照教材5.2.2节介绍的PFC解码算法，整个解码过程就是在PFC编码树上的“漫游”过程：最初从根节点出发；此后，根据编码串的当前编码位相应地向左（比特0）或向（比特1）右深入；

一旦抵达叶节点，则输出其对应的字符，并随即复位至根节点。

可见，算法无法继续的唯一可能情况时，在准备向下深入时发现没有对应的分支。然而根据其定义和约束条件，PFC编码树必然是真二叉树（proper binary tree）。也就是说，该算法运行过程中所抵达的每一内部节点，必然同时拥有左、右分支。因此上述情况实际上不可能发生。

[5-7] 因其解码过程不必回溯，PFC 编码算法十分高效。然而反过来，这一优点并非没有代价。试举例说明，如果因为信道干扰等影响致使某个比特位翻转错误，尽管解码依然可进行下去，但后续所有字符的解码都会出现错误。

【解答】

以如教材142页图5.35所示的编码树为例，考查与信息串"AI"相对应的编码串"$_{00}^{011}$"。

若因传输过程中所受干扰，导致接收到的编码串误作"$_{1}^{00}$$_{1}^{1}$"，则仍可正常解码，只不过得到的是错误的信息串"MAMM"。

请注意，依靠该算法本身，并不能检测出此类错误。

当然，这类例子还有很多。读者若有兴趣，不妨自行尝试其它的可能。

[5-8] 在 2.7.5 节我们已经看到，CBA 式排序算法在最坏情况下均至少需要Ω(nlogn)时间，但这并不足以衡量此类算法的总体性能。比如，我们尚不确定，是否在很多甚至绝大多数其它情况下有可能做到运行时间足够少，从而能够使得平均复杂度更低。试证明：若不同序列作为输入的概率均等，则任何 CBA 式排序算法的平均运行时间依然为Ω(nlogn)。（提示：PFC 编码）

【解答】

针对CBA式算法复杂度下界的估计，教材2.7.5节建议的统一方法是：先确定算法所对应的比较树（comparison tree），然后通过输入规模与可能的输出结果（叶节点）数目，推算出最小树高——即算法在最坏情况下所需的运行时间。

此处所需进一步考查的，是在各种输出结果符合某种概率分布的前提下，算法的平均性能。不难理解，实际上这等效于考查比较树中各叶节点的加权平均深度，其中各叶节点的权重取作出现对应输出的概率。

若将比较树与5.5.2节中的最优编码树（optimal encoding tree）做一对照即可看出，这也就是所谓的叶节点平均深度（average leaf depth）。特别地，在各种输出结果概率均等的前提下，对于任一固定的输入规模n，完全二叉树的叶节点平均深度可达到最小——然而即便如此，也至少有Ω(n**log**n)。

[5-9] 考查 5.4.1 节所介绍的各种递归式二叉树遍历算法。若将其渐进时间复杂度记作 T(n)，试证明：

$$T(n) = T(a) + T(n - a - 1) + \mathcal{O}(1) = \mathcal{O}(n)$$

【解答】

这些算法都属于二分递归（binary recursion），但无论如何，每当递归深入一层，都等效于将当前问题（遍历规模为n的二叉树）分解为两个子问题（分别遍历规模为a和n - a - 1的两棵子树）。因此，递归过程总是可以描述为题中的递推关系。在考虑到边界条件（递归基仅需常数时间），即可导出这些算法的运行时间均为$\mathcal{O}$(n)的结论。

[5-10] 试按照消除尾递归的一般性方法，将二叉树先序遍历算法的递归版（教材 124 页代码 5.11）改写为迭代形式。

【解答】

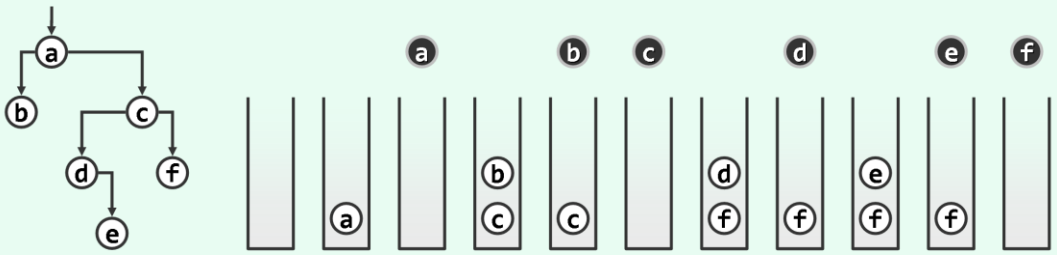
在引入辅助栈之后，可以实现如代码x5.1所示的迭代版先序遍历算法。

```
1 template <typename T, typename VST> //元素类型、操作器
2 void travPre_I1 ( BinNodePosi(T) x, VST& visit ) { //二叉树先序遍历算法（迭代版#1）
3    Stack<BinNodePosi(T)> S; //辅助栈
4    if ( x ) S.push ( x ); //根节点入栈
5    while ( !S.empty() ) { //在栈变空之前反复循环
6       x = S.pop(); visit ( x->data ); //弹出并访问当前节点，其非空孩子的入栈次序为先右后左
7       if ( HasRChild ( *x ) ) S.push ( x->rc ); if ( HasLChild ( *x ) ) S.push ( x->lc );
8    }
9 }
```

代码x5.1 二叉树先序遍历算法（迭代版#1）

请特别留意这里的入栈次序：根据“后进先出”原理，右孩子应先于左孩子入栈。

图x5.2以其左侧的二叉树为例，给出了先序遍历辅助栈从初始化到再次变空的演变过程。



图x5.2 迭代式先序遍历实例（出栈节点以深色示意）

[5-11] 考查教材 5.4.2、5.4.3、5.4.4 和 5.4.5 节所介绍的各种迭代式二叉树遍历算法。

a） 试证明，这些算法都是正确的——亦即，的确会访问每个节点一次且仅一次；

【解答】

纵观这些遍历算法，不难发现以下事实：

只要某个节点能被访问到，则其孩子节点必然也能

由此进一步地可知：只要某个节点能被访问到，则其每个后代节点必然也能。于是特别地，作为根节点的后代，树中的所有节点都能被（起始于根节点的）遍历访问到。

由此可见，这些遍历算法绝不致于遗漏任何节点。

另一方面，我们还需证明，这些算法也不致于重复访问任何节点，它们各自仅被访问一次。

实际上，纵观travPre_I1()算法（教材108页代码x5.1）、travPre_I2()算法（教材127页代码5.14）、travIn_I1()算法（教材129页代码5.15）、travIn_I2()算法（教材130页代码5.17）和travPost_I()算法（教材133页代码5.19），不难看出它们的一项共同点：

每个节点都在且仅在刚刚出栈之后，随即（通过调用visit()）被访问

因此只要进一步注意到，每个节点各自只入栈一次，即可确定每个节点的确至多被访问一次。

迭代式中序遍历算法travIn_I3()（教材131页代码5.18）虽然没有使用栈结构，但也具有类似性质。请读者自行找出规律，并证明同样的结论。

层次遍历算法travLevel()（教材134页代码5.20）采用了队列结构，类似地也不难看出：

每个节点都在且仅在刚刚出队之后，随即（通过调用visit()）被访问

再考虑到每个节点各自只入队一次，即可确定每个节点也至多被访问一次。

b） 试证明，无论递归式或迭代式，这些算法都具有线性时间复杂度；

【解答】

这些算法的运行时间主要消耗于两部分：栈（或队列）操作，以及对节点的访问操作。

根据以上分析，以上操作对每个节点而言各有不过常数次，因此总体而言，这些算法的运行时间都线性正比于二叉树自身的规模。

c） 这些算法的空间复杂度呢？

【解答】

这些算法所占用的空间，主要地无非是用于对辅助结构（栈或队列）的维护。

在travPre_I1()算法（代码x5.1）、travPre_I2()算法（代码5.14）、travIn_I1()算法（代码5.15）和travIn_I2()算法（代码5.17）的运行过程中，树中每一层至多只有一个节点存在栈中，因此栈结构的最大规模不超过二叉树的深度，最坏情况下为$O(n)$。

在travPost_I()算法（代码5.19）的运行过程中，树中每一层至多只有两个节点存在栈中，故栈结构的最大规模不超过二叉树深度的两倍，最坏情况下亦不过$O(n)$。

在travLevel()算法（代码5.20）的运行过程中，队列结构中所存节点的深度相差不超过一，故该结构的最大规模不超过二叉树中任意的相邻两层规模之和，最坏情况下也是$O(n)$。

迭代式中序遍历travIn_I3()算法（代码5.18），需要特别地做一深入讨论。正如教材中所指出的，这里并未（显式地）使用复杂的辅助结构，故表面上看仅需$O(1)$辅助空间。然而相对于功能相同的其它算法，这里却要求每个节点都必须配有parent指针。实际上，借助辅助结构的其它算法，则完全可以不必如此——即便是travPost_I()算法（代码5.19），也可以通过对入栈节点附加标记，以避免使用parent指针）。因此就这一意义严格而言，travIn_I3()算法仍需要$O(n)$的辅助空间。

[5-12] 对比教材中图 5.15 与图 5.16 不难发现，先序遍历与后序遍历在宏观次序上具有极强的对称性。利用这种对称性，试仿照 5.4.2 节所给先序遍历算法的迭代版，实现后序遍历算法更多的迭代版。

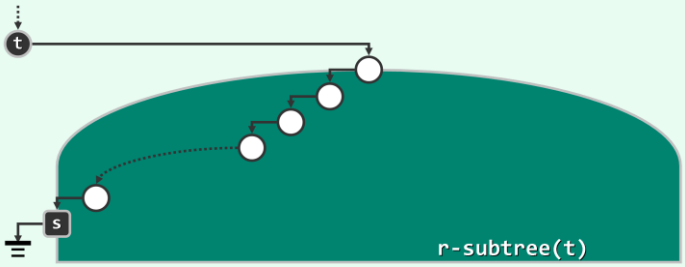
【解答】

请读者参照以上提示，独立完成编码和调试任务。

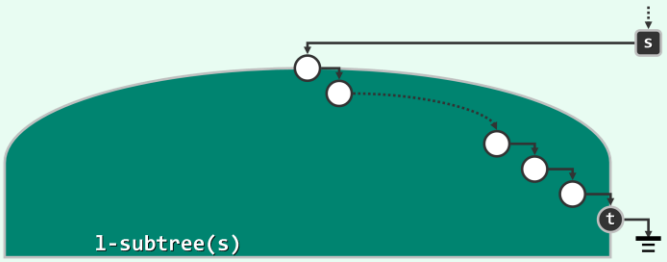
[5-13] 试针对代码 5.16 中 BinNode::succ()算法的两种情况，分别绘出一幅插图以说明其原理及过程。

【解答】

两种可能的情况，分别如图x5.3和图x5.4所示。



图x5.3 BinNode::succ()的情况一：t拥有右后代，其直接后继为右子树中左分支的末端节点s



图x5.4 BinNode::succ()的情况二：t没有右后代，其直接后继为以其为直接前驱的祖先s

需要强调的是，第二种情况涵盖了一种特殊情况：当s并不存在时，t即为全树的最大节点。

请对照教材119页代码5.2，体会宏IsRChild()的定义方式，尤其是如此定义如何能够简化129页代码5.16，同时确保BinNode<T>::succ()接口在上述特殊情况下能够正确地返回NULL。

[5-14] 仿照 BinNode::succ()（教材 129 页代码 5.16），实现二叉树节点直接前驱的定位接口 BinNode::pred()。

【解答】

请读者比照接口BinNode::succ()的实现方式，独立完成。

[5-15] 按照本章实现的迭代式算法（代码 x5.1、代码 5.14、代码 5.15、代码 5.17 和代码 5.19）对规模为 n 的二叉树做遍历，辅助栈的容量各应取作多大，才不致出现中途溢出？

【解答】

根据习题[5-11]的分析结论，最坏情况下（二叉树深度为Ω(n)时）辅助栈必须足以容纳Ω(n)个节点，这也是这些算法空间规模的安全下限。

[5-16] 中序遍历迭代式算法的第三个版本（教材 131 页代码 5.18），需反复地调用 succ()接口以定位直接后继，从而会相应地增加计算成本。

试问，该算法的渐进时间复杂度是否依然保持为 $O(n)$？若是，请给出证明；否则试举一例。

【解答】

该算法的时间复杂度依然还是$O(n)$。

为得出这一结论，只需证明：

无论二叉树规模与结构如何，对succ()接口所有调用所需时间总和不超过$O(n)$

反观教材129页代码5.16不难看出，实际上在这一场合下对succ()算法的调用，其中的if判断语句必然取else分支。因此，算法所消耗的时间应线性正比于其中while循环的步数，亦即其中对parent引用的访问次数。考查该次数，并将规模为n的二叉树所需的最大次数记作P(n)。可以证明，必有$P(n) \le n$。

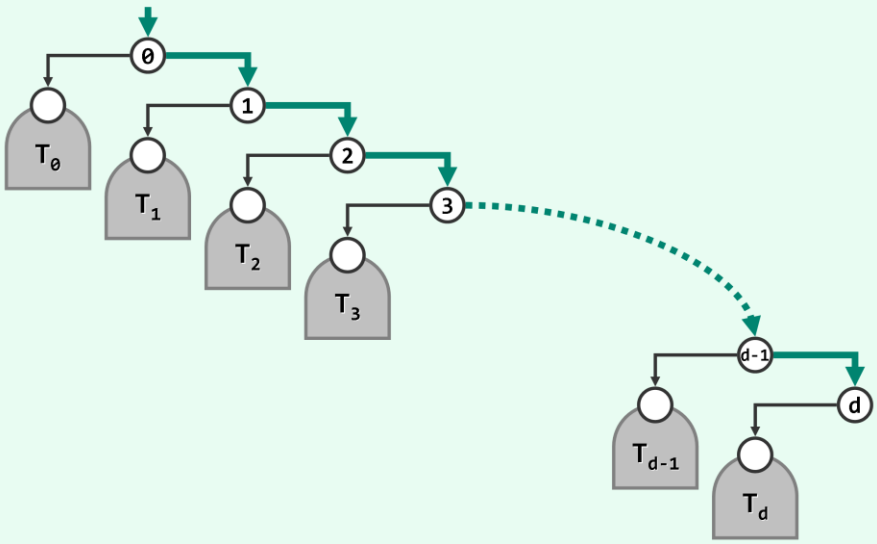
为此，我们对二叉树的高度做数学归纳。作为归纳基，不难验证：对于高度为-1（规模$n = 0$）的空树而言，根本无需访问parent引用，即$P(0) = 0$；对于高度为0（规模$n = 1$）的二叉树而言，只需访问（根节点数值为NULL的）parent引用一次，故有$P(1) = 1$。

因此，以下假设$P(n) \le n$对于高度小于h的所有二叉树均成立，并考查高度为h的二叉树T。

如图x5.5所示，若T的最右侧通路长度为d（$d \le h$），必然可以将T分解为$d + 1$棵子树：

$$\{ T_0, T_1, T_2, T_3, \ldots, T_{d-1}, T_d \}$$

当然，其中的某些子树可能是空树。另外，尽管各子树的高度未必相同，但必然都小于h。因此根据归纳假设，其各自内部的遍历过程中对parent引用的访问次数，应线性正比于其各自的规模。特别地，其中最后一个节点（若子树非空）对应的succ()调用中，最后一个访问的是从子树根节点，联接到全树最右侧通路的parent指针（图中以细线条示意）。请注意，尽管相对于孤立的子树而言，这个parent引用不再是NULL，但并不影响访问次数的统计。



图x5.5 二叉树中序遍历过程中对succ()接口的调用

因此，在所有子树内部的遍历过程中对parent引用的访问，累计不会超过：

$$|T_0| + |T_1| + |T_2| + |T_3| + \ldots + |T_{d-1}| + |T_d| = n - d - 1$$

就全树而言，除此之外尚未统计的，只有在最右侧通路末节点处对succ()的调用。该次调用过程中对parent引用的访问，也就是在图中以粗线条示意者（实际方向应颠倒，向上），不难看出其总数为$d + 1$。与各子树合并统计，总次数不超过：

$$(n - d - 1) + (d + 1) = n$$

[5-17] 考查中序遍历迭代式算法的第三个版本（教材 131 页代码 5.18）。
试继续改进该算法，使之不仅无需辅助栈，而且也无需辅助标志位。

【解答】

一种可行的改进方式，如代码x5.2所示。

```
1  template <typename T, typename VST> //元素类型、操作器
2  void travIn_I4 ( BinNodePosi(T) x, VST& visit ) { //二叉树中序遍历（迭代版#4，无需栈或标志位）
3     while ( true )
4        if ( HasLChild ( *x ) ) //若有左子树，则
5           x = x->lc; //深入遍历左子树
6        else { //否则
7           visit ( x->data ); //访问当前节点，并
8           while ( !HasRChild ( *x ) ) //不断地在无右分支处
9              if ( ! ( x = x->succ() ) ) return; //回溯至直接后继（在没有后继的末节点处，直接退出）
10             else visit ( x->data ); //访问新的当前节点
11          x = x->rc; //（直至有右分支处）转向非空的右子树
12       }
13 }
```

代码x5.2 二叉树中序遍历算法（迭代版#4）

可以看到，这里同样需要调用succ()接口确定回溯的位置。

请读者参照注释，验证该算法的正确性，并就其时、空效率做一分析。

[5-18] 考查实现如 134 页代码 5.20 所示的层次遍历算法，设二叉树共含 n 个节点。

a） 试证明，只要辅助队列 Q 的容量不低于⌈n/2⌉，就不致于出现中途溢出的问题；

【解答】

可以证明：在该算法执行过程中的每一步迭代之后，若当前已经有n个节点入过队，则仍在队中的至多有⌈n/2⌉个——当然，相应地，至少已有⌊n/2⌋个已经出队。

实际上，对该算法稍加观察即不难发现：每次迭代都恰有一个节点出队；若该节点的度数为d（0 ≤ d ≤ 2），则随即会有d个节点入队。通过对已出队节点的数目做数学归纳，即不难证明以上事实。我们将此项工作留给读者。

b） 在规模为 n 的所有二叉树中，哪些的确会需要如此大容量的辅助队列？

【解答】

在算法过程中的任一时刻，辅助队列的规模均不致小于仍应在队列中节点的数目。考查这些节点在目前已入过队的节点中所占的比重。由以上观察结果，可以进一步推知：为使这一比重保持为尽可能大的⌈n/2⌉/n，此前所有出队节点的度数都必须取作最大的2；且中途一旦某个节点只有1度甚至0度，则不可能恢复到这一比重。

由此可见，若果真需要如此大容量的辅助队列，则在最后一个节点入队之前，所有出队节点都必须是2度的。由此可见，其对应的充要条件是，这是一棵规模为n的完全二叉树。

c） 在层次遍历过程中，若 Q 中节点的总数的确会达到这么多，则至多可能达到多少次？

【解答】

按照层次遍历的次序，若将树中各节点依次记作：

$$x_1,\ x_2,\ \ldots,\ x_{\lfloor n/2\rfloor};\ x_{\lfloor n/2\rfloor+1},\ \ldots,\ x_n$$

则其中$x_{1\sim\lfloor n/2\rfloor}$为内部节点，共计$\lfloor n/2\rfloor$个；$x_{\lfloor n/2\rfloor+1\sim n}$为叶节点，共计$\lceil n/2\rceil$个。

根据以上分析，若n为奇数，则必然是一棵真完全二叉树，此时的最大规模$\lceil n/2\rceil=(n+1)/2$仅在$x_{\lfloor n/2\rfloor+1}$处于队首时出现一次。若n为偶数，则只有最后一个内部节点$x_{\lfloor n/2\rfloor}$的度数为1，此时的最大规模$\lceil n/2\rceil=n/2$在$x_{\lfloor n/2\rfloor}$和$x_{\lfloor n/2\rfloor+1}$处于队首时各出现一次。

[5-19] 参考图 5.26（教材 135 页）和图 5.27（教材 135 页）中的实例，考查对规模为 n 的完全二叉树（含满二叉树）的层次遍历。

a） 试证明：在整个遍历过程中，辅助队列的规模变化是单峰对称的，即

{ 0, 1, 2, ..., (n + 1)/2, ..., 2, 1, 0 }（n 为奇数时）或

{ 0, 1, 2, ..., n/2, n/2, ..., 2, 1, 0 }（n 为偶数时）

【解答】

根据上题的分析结论，显然成立。

b） 非完全二叉树的层次遍历过程，是否也可能具有这种性质？为什么？

【解答】

仍由上题的分析结论，在对非完全二叉树的遍历过程中，辅助队列的规模不可能达到$\lceil n/2\rceil$。

[5-20] 在完全二叉树的层次遍历过程中，按入队（亦即出队）次序从 0 起将各节点 X 编号为 r(X)。

a） 试证明：对于任一节点 X 及其左、右孩子 L 和 R（如果存在），必然有

$$r(L) = 2*r(X) + 1$$

$$r(X) = \lfloor (r(L) - 1)/2 \rfloor = (r(L) - 1)/2$$

$$r(R) = 2*r(X) + 2$$

$$r(X) = \lfloor (r(R) - 1)/2 \rfloor = r(R)/2 - 1$$

【解答】

由图10.2（教材287页）中的实例，直接易见。

b） 试证明：任一编号 r ∈ [0, n)都唯一对应于某个节点；

【解答】

由图10.2（教材287页）中的实例，直接易见。

c） 很多应用往往只涉及完全二叉树，此时，如何利用上述性质提高对树的存储和处理效率？

【解答】

将所有节点存入向量结构，各节点X的秩rank(X)即为其编号r(X)。

d） 根据以上编号规则，如何判断任何一对节点之间是否存在“祖先-后代”关系？

【解答】

令s(X) = r(X) + 1，S(X)为s(X)的二进制展开，于是有：

(1) 节点A是D的祖先，当且仅当S(A)是S(D)的前缀。其中特别地，
(2) 节点A是D的父亲，当且仅当S(A)是S(D)的前缀且|S(A)| + 1 = |S(D)|

以图10.2（教材287页）中的节点1、8和18为例，即可验证上述结论：

$s(1) = r(1) + 1 = 2 = 10_{(2)}$

$s(8) = r(8) + 1 = 9 = 1001_{(2)}$

$s(18) = r(18) + 1 = 19 = 10011_{(2)}$

[5-21] 采用“父节点 + 孩子节点”方式表示和实现有根的有序多叉树，隶属于同一节点的孩子节点互为兄弟，且此处的有序性可以理解为“左幼右长”——位置偏左者为弟，偏右者为兄。实际上，这只是现代意义上对“弟”和“兄”的理解，具体到学源上师生关系，可对应于师弟、师兄。

但按中国传统文化，就此的理解与约定却有所不同：凡同辈之间，无论长幼均统一互称为“兄”；而所谓“弟”，则用以指称后辈，大抵相当于“弟子”或“徒弟”。

试问：若照此传统惯例，将“子”改称作“弟”，将“兄弟”统一作“兄”，则多叉树的“父节点 + 孩子节点”表示法，将恰好对应于二叉树的哪种表示法？

【解答】

“长子-兄弟”表示法。

[5-22] 考查借助二叉树，表示（有根有序）多叉树的长子-兄弟表示法：分别以左/右孩子作为长子/兄弟。

a） 试基于 BinTree 模板类（教材 121 页代码 5.5），派生出 Tree 模板类；

b） 试结合应用，测试你的 Tree 模板类。

【解答】

请读者参照教材第5.1.3节关于“长子-兄弟”表示法之原理的讲解，以及这里所给的提示，独立完成设计、编码及测试工作。

[5-23] 试在 BinTree 模板类（教材 121 页代码 5.5）的基础上，扩展 BinTree::swap()接口，在 $O(n)$ 时间将二叉树中每一个节点的左、右孩子（其中之一可能为空）互换，其中 n 为树中的节点总数。

【解答】

在教材所给的递归版先序、中序或后序遍历算法的基础上，在每个递归实例中，交换当前节点的左、右孩子（子树）。

请读者照此思路，独立完成算法的编码和调试。

虽经如上扩充，但每个递归实例渐进地仍然仅需常数时间，故总体时间复杂度依然为$O(n)$。

当然，为了提高实际的运行效率，可以进一步改为迭代形式。此项任务，由读者独立完成。

[5-24] 设二叉树共含 n 个节点，且各节点数据项的类型支持大小比较和线性累加（类似于整数或浮点数）。

a） 试设计并实现一个递归算法，在 $O(n)$时间内判断是否该树中所有节点的数值均不小于其真祖先的数值总和。对于没有真祖先的树根节点，可认为“真祖先”的数值总和为 0；

【解答】

仍以教材所给的递归版先序、中序或后序遍历算法，作为基础框架。

引入一个辅助栈，用以记录从根节点到当前节点的（唯一）通路，当然，沿途节点亦即当前节点的所有祖先。另设一个累加器，动态记录辅助栈中所有节点数据项的总和。

为此，递归每深入一层，即将当前节点压入辅助栈中，同时累计其数据项；反之，递归每返回一层，即弹出辅助栈的顶部节点，并从累加器中扣除其数据项。对于每个当前节点，都将其数据项与累加器做一比较。一旦确认前者小于后者，即可立刻报告“NO”并退出。若直到辅助栈重新变空，都未发生上述情况，即最终可报告“YES”并退出。

请读者照此思路，独立完成算法的编码和调试。

同样地，以上扩充既不致增加递归实例的数量，亦不会增加各递归实例的渐进执行时间，故总体的时间复杂度依然为$O(n)$。

b） 试将以上算法改写为等价的迭代形式，且运行时间依然为 $O(n)$；

【解答】

请读者参照教材中各种迭代版遍历算法的实现方式和技巧，独立完成这一改进。

c） 迭代版需要多少空间？

【解答】

因为省去了由系统隐式维护的递归调用栈，故迭代版只要实现得当，实际占用的空间将大为减少——尽管渐进的空间复杂度依然是线性的。

[5-25] 设二叉树共含 n 个节点，且各节点数据项的类型支持大小比较（类似于整数或浮点数）。

a） 试设计并实现一个递归算法，在 $O(n)$时间内将每个节点的数值替换为其后代中的最大数值。

【解答】

以教材所给的递归版后序遍历算法为基础框架，做必要的扩充。

首先，需要调整接口约定，使每个递归实例都有返回值——亦即，当前节点更新后的数据项。

作为递归基，空树可返回可能的最小值（比如对于整数，可取INT_MIN）。这样，按照后序遍历的次序，只要当前节点的左、右子树均已遍历（左、右孩子的数据项均已更新），即可从二者当中取其更大者，并相应地更新当前节点的数据项，最后再返回更新后的数据项。

请读者照此思路，独立完成算法的编码和调试任务。

同样地，以上扩充既不致增加递归实例的数量，亦不会增加各递归实例的渐进执行时间，故总体的时间复杂度依然为$O(n)$。

b） 试将以上算法改写为等价的迭代形式，且运行时间依然为 $O(n)$；

【解答】

请读者参照教材中各种迭代版后序遍历算法的实现方式和技巧，独立完成这一改进。

c） 迭代版需要多少空间？

【解答】

因为省去了由系统隐式维护的递归调用栈，故迭代版只要实现得当，实际占用的空间将大为减少——尽管渐进的空间复杂度依然是线性的。

[5-26] 设二叉树共含 n 个节点，且各节点数据项的类型支持线性累加（类似于整数或浮点数）。试设计并实现一个递归算法，按照如下规则，在 $O(n)$时间内为每个节点设置适当的数值：

树根为 0

对于数值为 k 的节点，其左孩子数值为 2k + 1，右孩子数值为 2k + 2

【解答】

不难看出，对于完全二叉树，题中的要求实际上等效于，按照层次遍历的次序，为树中的各节点顺序编号。而一般的二叉树作为其子树，各节点的编号也与完全二叉树完全吻合。因此可以教材所给的层次遍历算法为基础框架，并做必要的扩充。

具体地，在根节点首先入队之前，将其数据项置为0。此后的每一步迭代中，若出队节点的编号为k，则入队的左、右孩子节点（若存在）的数值，可分别取作2k + 1、2k + 2。

请读者照此思路，独立完成编码和调试任务。

以上扩充并不会增加各步迭代的渐进执行时间，故总体时间复杂度依然为$O(n)$。

[5-27] 试证明，在考虑字符的出现频率之后，最优编码树依然具有双子性。

【解答】

与不考虑出现频率时的情况相仿，依然可以反证。

假若某棵最优编码树不是真二叉树，则通过收缩变换消除其中的单分支节点，同时平均编码长度亦必然缩短。矛盾。

[5-28] 试证明，5.5.4 节所述 Huffman 编码算法的原理，对任意字符集均成立。

【解答】

可以证明：

由Huffman算法所生成的编码树，的确是最优编码树

为此，可以对字符集的规模|Σ|做归纳。作为归纳基，对单字符集（|Σ| = 1）而言显然。故不妨在|Σ| < n时以上命题均成立，现考查|Σ| = n的情况。

按照Huffman算法的流程，首先从Σ中取出频率最低的两个字符x和y，并将其合并为一个新的超字符z。而在算法此后的执行过程中，可以等效地认为x和y已被删除，并被代以z——亦即，

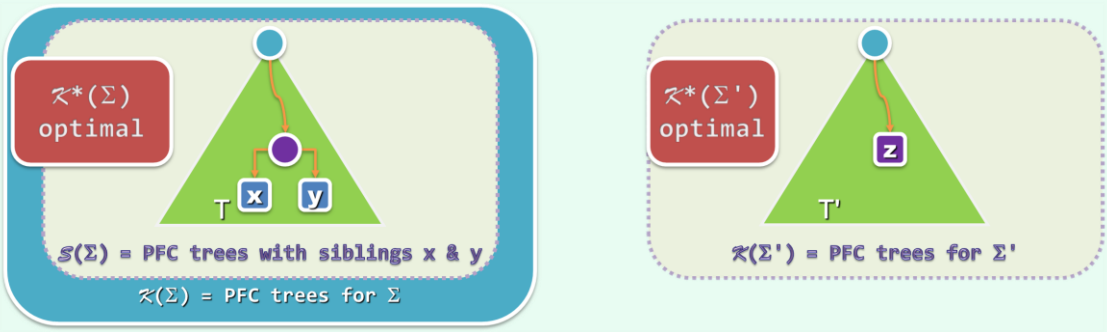
字符集相当于被替换为：

$$\Sigma' = (\Sigma \setminus \{x, y\}) \cup \{z\}$$

其中

$$w(z) = w(x) + w(y)$$

考查字符集Σ和Σ'各自的所有编码树，如图x5.6所示，令其分别构成集合$\mathcal{T}(\Sigma)$与$\mathcal{T}(\Sigma')$，其中的最优编码树分别为其中的子集$\mathcal{T}^*(\Sigma)$与集合$\mathcal{T}^*(\Sigma')$。



图x5.6 频率最低的兄弟节点合并之后，最优编码树必对应于合并之前的最优编码树

鉴于最优编码树必然具有层次性，故在$\mathcal{T}^*(\Sigma)$中只需考虑x和y互为兄弟的那些编码树，不妨令其构成集合$\mathcal{S}(\Sigma)$。于是如图x5.6所示，在按照以上方法统一合并x和y之后，即在$\mathcal{S}(\Sigma)$与$\mathcal{T}(\Sigma')$之间建立起了一一对应的关系。

考查如此对应的任意两棵树$T \in \mathcal{S}(\Sigma)$和$T' \in \mathcal{T}(\Sigma')$，二者的编码总长之差为：

$$n*ald(T) - (n - 1)*ald(T') = w(x) + w(y) = w(z)$$

对于固定的字符集Σ而言，这个差异w(z)即是一个常数。因此，$\mathcal{T}^*(\Sigma')$中的最优编码树，亦必然对应于$\mathcal{S}(\Sigma) \cap \mathcal{T}^*(\Sigma')$中的最优编码树。故归纳假设可以推广至$|\Sigma| = n$的情况。

[5-29] 5.5.4 节针对 Huffman 树构造算法的讲解中，暂时忽略了歧义情况。比如，有些字符的出现频率可能恰好相等；或者虽然最初的字符权重互异，但经过若干次合并之后，森林 F 也可能会出现权重相等的子树。另外，每次选出的一对（超）字符在合并时的左右次序也没有明确说明。

a） 试证明，以上歧义并不影响所生成编码树的最优性，即它仍是 Huffman 编码树之一；

【解答】

上题所给的证明，并未排除字符x和y权重相同的情况，故其结论足以覆盖本题。

b） 参照教材所给代码，了解并总结在实现过程中处理这类歧义的一般性方法。

【解答】

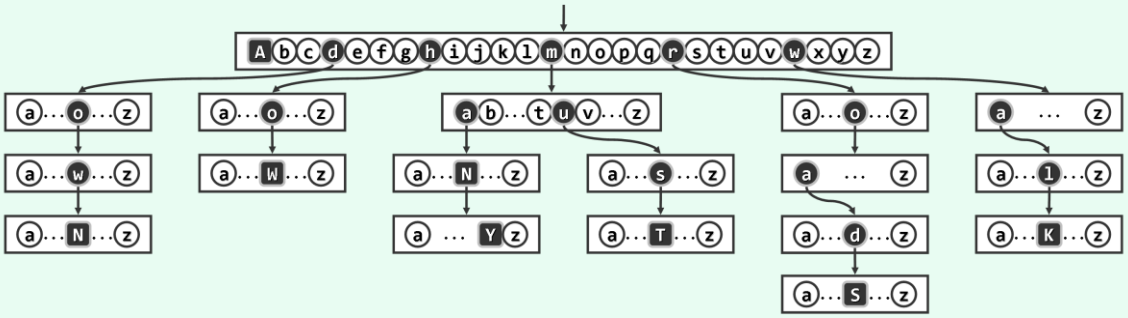
这里实际上是将此类歧义情况的处理，转交给具体实现Huffman森林的数据结构，比如采用列表结构（教材138页代码5.24），或者采用优先级队列结构（教材285页代码10.2）。

目前这些结构对歧义情况均未强制地处理，而多是依照其在逻辑序列中的次序，确定等权重（超）字符的合并次序。反之，若需要显示地消除此类歧义，亦可从这些方面入手。

[5-30] 设字符表为 Σ（|Σ| = r）。任一字符串集 S 都可如图 x5.7 所示，表示为一棵键树（trie）[①]。

键树是有根有序树，其中的每个节点均包含 r 个分支。深度为 d 的节点分别对应于长度为 d 的字符串，且祖先所对应字符串必为后代所对应字符串的前缀。键树只保留与 S 中字符串（及其前缀）相对应的节点（黑色），其余的分支均标记为 NULL（白色）。

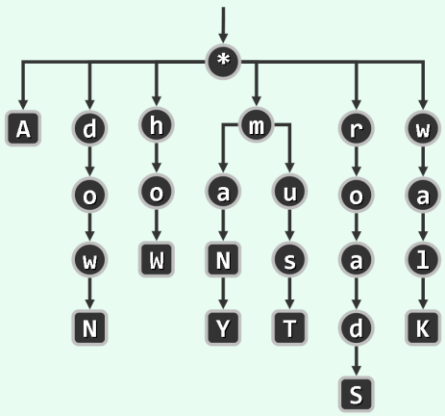
注意，因为不能保证字符串相互不为前缀（如"man"与"many"），故对应于完整字符串的节点（黑色方形、大写字母）未必都是叶子。



图x5.7 字符串集{ "how", "many", "roads", "must", "a", "man", "walk", "down" }对应的键树

试按照如图 x5.7 所示的构思，实现对应的 Trie 模板类。同时要求提供一个接口 find(w)，在 $O(|w|) = O(h)$的时间内判断 S 是否包含字符串 w，其中|w|为该字符串的长度，h 为树高。

（提示：每个节点都实现为包含 r 个指针的一个向量，各指针依次对应于 Σ 中的字符：S 包含对应的字符串（前缀），当且仅当对应的指针非空。此外，每个非空指针都还需配有一个 bool 类型的标志位，以指示其是否对应于 S 中的某个完整的字符串。于是，键树的整体逻辑结构可以抽象为如图 x5.8 所示的形式。其中，黑色方形元素的标志位为 true，其余均为 false。）



图x5.8 键树的紧凑表示与实现

【解答】

请读者根据以上介绍和提示，独立完成编码和调试任务。

[①] 由R. de la Briandais于1959年发明[53]。名字源自单词re**TRIE**val，发音上为区别于tree，改读作[trai]