第6章 树与森林

**一、复习要点**

本章主要介绍了树与森林、二叉树的定义、性质、操作和相关算法的实现。特别是二叉树的遍历算法，它们与许多以此为基础的递归算法都必须认真学习。因为树的先根遍历次序与对应二叉树表示的前序遍历次序一致，树的后根遍历次序与对应二叉树的中序遍历次序一致，因此可以据此得出树的遍历算法。线索化二叉树是直接利用二叉链表的空链指针记入前驱和后继线索，从而简化二叉树的遍历。堆是一种二叉树的应用，可以用它作为优先级队列的实现。它的存储表示是完全二叉树的顺序存储方式，它的定义不要求堆中的数据有序，但要求双亲结点与子女结点必须满足某种关系。本章最后讨论霍夫曼树。这种树是扩充二叉树，要求在外结点上带有权值，在构造霍夫曼树时必须注意一个新结点的左子女上所带的权值小于右子女上所带的权值，这不是霍夫曼树必须这样，而是实现算法造成这种结果。此外，作为霍夫曼树的应用，引入霍夫曼编码。通常让霍夫曼树的左分支代表编码“0”，右分支代表编码“1”，得到霍夫曼编码。这是一种不等长编码，可以有效地实现数据压缩。

本章复习的要点是：

**1、**基本知识点

要求理解树和森林的定义，树的抽象数据类型，二叉树的定义，二叉树的性质，二叉树的抽象数据类型，二叉树的数组表示和链表存储表示。要求掌握二叉树的遍历，包括中序遍历、前序遍历、后序遍历方法，要求理解二叉树的计数方法及从二叉树遍历结果得到二叉树的方法。对于线索化二叉树，要求理解什么是线索，中序线索化二叉树的结构特性及寻找某结点的前驱和后继的方法。此外，需要理解堆的定义及其实现的方法，本章只考虑用完全二叉树的顺序存储来实现。还需要理解堆的建立，插入与删除过程。要求掌握树/森林与二叉树的转换，树的遍历方法。最后要求掌握霍夫曼树的实现方法及霍夫曼编码的概念。

**2、**算法设计

⮚ 建立二叉树的递归算法。

⮚ 前序、中序、后序遍历二叉树的递归算法。

⮚ 使用栈的前序、中序、后序遍历的非递归算法。

⮚ 统计二叉树结点个数，二叉树叶结点个数，二叉树高度的递归算法。

⮚ 自左向右链接二叉树叶结点的递归算法。

⮚ 判断两棵二叉树相等和交换二叉树左、右子女指针的递归算法。

⮚ 通过二叉树的遍历建立前序线索化二叉树和中序线索化二叉树的算法。

⮚ 中序线索化二叉树上的中序遍历算法。

⮚ 前序线索化二叉树上的前序遍历算法。

⮚ 后序线索化二叉树上的后序遍历算法。

⮚ 利用堆实现优先级队列的操作。

⮚ 堆的自上向下和自下向上的调整算法。

⮚ 堆的插入与删除算法。

⮚ 树的先根、后根、层次遍历算法（基于树的二叉树表示）。

**二、难点与重点**

1、树：树的定义、树的基本运算

* 树的分层定义是递归的
* 树中结点个数与高度的关系

2、二叉树：二叉树定义、二叉树的基本运算

* 二叉树性质、二叉树中结点个数与高度的关系、不同种类的二叉树棵数
* 完全二叉树的顺序存储、完全二叉树的双亲、子女和兄弟的位置
* 二叉树的前序·中序·后序遍历的递归算法和使用栈的非递归算法
* 二叉树的层次遍历算法
* 中序线索化二叉树、前驱与后继的查找方法、建立中序线索化二叉树的算法

3、霍夫曼树：霍夫曼树的构造方法、霍夫曼编码、带权路径长度的计算

* 霍夫曼树是带权路径长度最小的扩充二叉树
* 构造霍夫曼树时，按构造算法，每次具最小关键码的子树是根的左子树，具次小关键码的子树是根的右子树
* 在构造过程中，新二叉树按根的权值加入到森林的最后

4、树与森林

* 树的广义表表示、树的双亲表示、树的左子女-右兄弟表示
* 树、森林与二叉树的对应关系
* 树的先根·后根·层次遍历算法

5、堆：堆的定义、堆的插入与删除算法

* 形成堆时用到的向下调整算法
* 形成堆时的调整过程及比较次数的上界估计
* 堆插入时用到的向上调整算法

**三、教材中习题的解析**

6-1 写出用广义表表示法表示的树的类声明，并给出如下成员函数的实现：

(1) **operator** >> ( ) 接收用广义表表示的树作为输入，建立广义表的存储表示；

(2) 复制构造函数 用另一棵表示为广义表的树初始化一棵树；

(3) **operator** == ( ) 测试用广义表表示的两棵树是否相等；

(4) **operator** << ( ) 用广义表的形式输出一棵树；

(5) 析构函数 清除一棵用广义表表示的树。

【解答】

#**include** <iostream.h>

**#define** maxSubTreeNum 20**;**  //最大子树(子表)个数

**class** GenTree**;** //GenTree类的前视声明

**class** GenTreeNode **{** //广义树结点类的声明

**friend class** GenTree**;**

**private:**

**int** utype; //结点标志：=0, 数据; =1, 子女

GenTreeNode \* nextSibling**;** //utype=0, 指向第一个子女;

//utype=1或2, 指向同一层下一兄弟

**union {** //联合

**char** RootData**;** //utype=0, 根结点数据

**char** Childdata**;** //utype=1, 子女结点数据

GenTreeNode \*firstChild**;** //utype=2, 指向第一个子女的指针

**}**

**public:**

GenTreeNode ( **int** tp, **char** item ) **:** utype (tp), nextSibling (NULL)

**{ if** ( tp == 0 ) RootData = item**; else** ChildData = item**; }**

//构造函数：构造广义表表示的树的数据结点

GenTreeNode (GenTreeNode \*son = NULL ) **:** utype (2), nextSibling (NULL), firstChild ( son ) **{ }**

//构造函数：构造广义表表示的树的子树结点

**int** nodetype ( ) **{ return** utype**;** **}** //返回结点的数据类型

**char** GetData ( ) **{ return** data**;** **}** //返回数据结点的值

GenTreeNode \* GetFchild ( ) **{ return** firstChild**; }** //返回子树结点的地址

GenTreeNode \* GetNsibling ( ) **{ return** nextSibling**; }** //返回下一个兄弟结点的地址

**void** setInfo ( **char** item ) **{** data = item**; }** //将结点中的值修改为item

**void** setFchild ( GenTreeNode \* ptr ) **{** firstChild = ptr**; }** //将结点中的子树指针修改为ptr

**void** setNsinbilg ( GenTreeNode \* ptr ) **{** nextSibling = ptr**; }**

**};**

**class** GenTree **{** //广义树类定义

**private:**

GenTreeNode \*first**;** //根指针

**char** retValue**;** //建树时的停止输入标志

GenTreeNode \*Copy ( GenTreeNode \* ptr )**;**  //复制一个ptr指示的子树

**void** Traverse ( GenListNode \* ptr )**;** //对ptr为根的子树遍历并输出

**void** Remove ( GenTreeNode \*ptr )**;** //将以ptr为根的广义树结构释放

**friend int** Equal ( GenTreeNode \*s**,** GenTreeNode \*t )**;** //比较以s和t为根的树是否相等

**public:**

GenTree ( )**;**  //构造函数

GenTree ( GenTree**&** t )**;** //复制构造函数

~GenTree ( )**;** //析构函数

**friend int operator** == ( GenTree**&** t1**,** GenTree**&** t2 )**;** //判两棵树t1与t2是否相等

**friend istream& operator** >> ( **istream&** in, **GenTree&** t )**;** //输入

**friend ostream& operator <<** ( **ostream&** out, **GenTree&** t )**;** //输出

**}**

(1) **operator** >> ( ) 接收用广义表表示的树作为输入，建立广义表的存储表示

**istream& operator** >> ( **istream&** in, **GenTree&** t ) **{**

//友元函数, 从输入流对象in接受用广义表表示的树，建立广义表的存储表示t。

t.ConstructTree ( in, retValue )**;**

**return** in**;**

**}**

**void** GenTree **::** ConstructTree ( **istream&** in, **char&** value ) **{**

**//**从输入流对象in接受用广义表表示的非空树，建立广义表的存储表示t。

Stack <GenTreeNode\* > st (maxSubTreeNum)**;** //用于建表时记忆回退地址

GenTreeNode \* p, q, r**;** **Type** ch**;**

**cin** >> value**; //**广义树停止输入标志数据

**cin** >> ch**;** first = q = **new** GenTreeNode ( 0, ch )**;**  //建立整个树的根结点

**cin** >> ch**;** **if** ( ch == ‘(’ ) st.Push ( q )**;** //接着应是‘(’, 进栈

**cin** >> ch**;**

**while** ( ch != value ) **{** //逐个结点加入

**switch** ( ch ) **{**

**case** ‘(’ **:** p = **new** GenTreeNode **<Type>** ( q )**;** //建立子树, p->firstChild = q

r = st.GetTop( )**;** st.Pop( )**;** //从栈中退出前一结点

r->nextSibling = p**;** //插在前一结点r之后

st.Push( p )**;** st.Push( q )**;**  //子树结点及子树根结点进栈

**break;**

**case** ‘)’ **:** q->nextSibling = NULL**;** st.pop( )**;** //子树建成, 封闭链, 退到上层

**if** ( st.IsEmpty ( ) == 0 )q = st.GetTop( )**;** //栈不空, 取上层链子树结点

**else return** 0**;** //栈空, 无上层链, 算法结束

**break;**

**case** ‘,’ **:** **break;**

**default :** p = q**;**

**if** ( isupper (ch) ) q **= new** GenTreeNode ( 0, ch )**;** //大写字母, 建根结点

**else** q = **new** GenTreeNode ( 1, ch )**;** //非大写字母, 建数据结点

p->nextSibling = q**;** //链接

**}**

**cin** >> ch**;**

**}**

**}**

(2) 复制构造函数 用另一棵表示为广义表的树初始化一棵树；

GenTree **::** GenTree ( **const** GenTree**&** t ) **{** //共有函数

first = Copy ( t.first )**;**

**}**

GenTreeNode\* GenTree **::** Copy ( GenTreeNode \*ptr ) **{**

//私有函数，复制一个ptr指示的用广义表表示的子树

GenTreeNode \*q = NULL**;**

**if** ( ptr != NULL ) **{**

q = **new** GenTreeNode ( ptr->utype, NULL )**;**

**switch** ( ptr->utype ) **{** //根据结点类型utype传送值域

**case** 0 **:** q->RootData = ptr->RootData**;** **break;** //传送根结点数据

**case** 1 **:** q->ChildData = ptr->ChildData**;** **break;** //传送子女结点数据

**case** 2 **:** q->firstChild = Copy ( **ptr**->firstChild )**;** **break;** //递归传送子树信息

**}**

q->nextSibling = Copy ( ptr->nextSibling )**;** //复制同一层下一结点为头的表

**}**

**return** q**;**

**}**

(3) **operator** == ( ) 测试用广义表表示的两棵树是否相等

**int operator** == ( GenTree**&** t1**,** GenTree**&** t2 ) **{**

//友元函数 **:** 判两棵树t1与t2是否相等, 若两表完全相等, 函数返回1, 否则返回0。

**return** Equal ( t1.first, t2.first )**;**

**}**

**int** Equal ( GenTreeNode \*t1, GenTreeNode \*t2 ) **{**

**//**是GenTreeNode的友元函数

**int** x**;**

**if** ( t1 == NULL **&&** t2 == NULL ) **return** 1**;** //表t1与表t2都是空树, 相等

**if** ( t1 != NULL **&&** t2 != NULL **&&** t1->utype == t2->utype ) **{** //两子树都非空且结点类型相同

**switch** (t1->utype ) **{** //比较对应数据

**case** 0 **:** x =( t1->RootData == t2->RootData ) **?** 1 **:** 0**;** //根数据结点

**break;**

**case** 1 **:** x = ( t1->ChildData == t2->ChildData ) **?** 1 **:** 0**;** //子女数据结点

**break;**

**case** 2 **:** x = Equal ( t1->firstChild, t2->firstChild )**;** //递归比较其子树

**}**

**if** ( x ) **return** Equal ( t1->nextSibling, t2->nextSibling )**;**

//相等, 递归比较同一层的下一个结点; 不等, 不再递归比较

**}**

**return** 0**;**

**}**

(4) **operator** << ( ) 用广义表的形式输出一棵树

**ostream& operator <<** ( **ostream&** out, **GenTree&** t ) **{**

//友元函数, 将树t输出到输出流对象out。

t.traverse ( out, t.first )**;**

**return** out**;**

**}**

**void** GenTree **::** traverse ( **ostream&** out, GenTreeNode \* ptr ) **{**

//私有函数, 广义树的遍历算法

**if** ( ptr != NULL ) **{**

**if** ( ptr->utype == 0 )out << ptr->RootData << ‘(’**;** //根数据结点

**else if** ( ptr->utype == 1 ) **{** //子女数据结点

out **<<** ptr->ChildData**;**

**if** ( ptr->nextSibling != NULL ) out << ‘,’**;**

**}**

**else {** //子树结点

traverse ( ptr->firstChild )**; //**向子树方向搜索

**if** ( ptr->nextSibling != NULL ) out << ‘,’**;**

**}**

traverse ( ptr->nextSibling )**;** //向同一层下一兄弟搜索

**}**

**else** out << ‘)’**;**

**}**

(5) 析构函数 清除一棵用广义表表示的树

GenTree **::** ~ GenTree ( ) **{**

//用广义表表示的树的析构函数, 假定first ≠ NULL

Remove ( first )**;**

**}**

**void** GenTree **::** Remove ( GenTreeNode \*ptr ) **{**

GenTreeNode \* p**;**

**while** ( ptr != NULL ) **{**

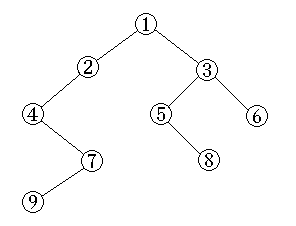
p = ptr->nextSibling**;**

**if** ( p->utype == 2 ) Remove ( p->firstChild )**; //**在子树中删除

ptr->nextSibling = p->nextSibling**;** **delete** ( p )**;** //释放结点p

**}**

**}**



6-2 列出右图所示二叉树的叶结点、分支结点和每个结点的层次。

【解答】

二叉树的叶结点有⑥、⑧、⑨。分支结点有①、②、③、④、⑤、⑦。 结点①的层次为0；结点②、③的层次为1；结点④、⑤、⑥的层次为2；结点⑦、⑧的层次为3；结点⑨的层次为4。

6-3 使用 (1) 顺序表示和 (2) 二叉链表表示法，分别画出右图所示二叉树的存储表示。

【解答】

0 1 2 3 4 5 6 7 8 9

10 11 12 13 14 15 16 17 18

① ② ③ ④ ⑤ ⑥ ⑦

⑧ ⑨

顺序表示

④

⑤

⑦

⑨

⑧

∧

∧

∧

∧

∧

∧

∧

∧

①

②

③

⑥

∧

∧

二叉链表表示

6-4 用嵌套类写出用链表表示的二叉树的类声明。

【解答】

#**include** <iostream.h>

**template <class Type> class** BinaryTree **{**

**private:**

**template <class Type> class** BinTreeNode **{**

**public:**

BinTreeNode<**Type**> \*leftChild**,** \*rightChild**;**

**Type** data**;**

**}**

**Type** RefValue**;**

BinTreeNode<**Type**> \* root**;**

BinTreeNode<**Type**> \*Parent ( BinTreeNode<**Type**> \*start, BinTreeNode<**Type**> \*current )**;**

**int** Insert ( BinTreeNode<**Type**> \*current, **const Type&** item )**;**

**int** Find ( BinTreeNode<**Type**> \*current, **const Type&** item ) **const;**

**void** destroy ( BinTreeNode<**Type**> \*current )**;**

**void** Traverse ( BinTreeNode<**Type**> \*current, **ostream&** out ) **const;**

**public:**

BinaryTree ( ) **:** root ( NULL ) **{ }**

BinaryTree ( **Type** value ) **:** RefValue ( value ), root ( NULL )**{ }**

~BinaryTree ( ) **{** destroy (root)**; }**

BinTreeNode ( ) **:** leftChild ( NULL ), rightChild ( NULL ) **{ }**

BinTreeNode ( **Type** item ) **:** data ( item ), leftChild ( NULL ), rightChild ( NULL ) **{ }**

**Type&** GetData ( ) **const { return** data**; }**

BinTreeNode<**Type**>\* GetLeft ( ) **const { return** leftChild**; }**

BinTreeNode<**Type**>\* GetRight ( ) **const { return** rightChild**; }**

**void** SetData ( **const Type&** item )**{** data = item**; }**

**void** SetLeft ( BinTreeNode<**Type**> \*L ) **{** leftChild = L**; }**

**void** SetRight ( BinTreeNode<**Type**> \*R )**{** RightChild =R**; }**

**int** IsEmpty ( ) **{ return** root == NULL **?** 1 **:** 0**; }**

BinTreeNode<**Type**> \*Parent ( BinTreeNode<**Type**> \*current )

**{ return** root == NULL || root == current **?** NULL **:** Parent ( root, current )**; }**

BinTreeNode<**Type**> \* LeftChild ( BinTreeNode<**Type>** \*current )

**{ return** current != NULL **?** current->leftChild **:** NULL**; }**

BinTreeNode<**Type**> \* RighttChild ( BinTreeNode<**Type**> \*current )

**{ return** current != NULL **?** current->rightChild **:** NULL**; }**

**int** Insert ( **const Type&** item )**;**

BinTreeNode<**Type**> \* Find ( **const Type&** item )**;**

BinTreeNode<**Type**> \* GetRoot ( ) **const { return** root**; }**

**friend istream& operator** >> ( **istream&** in, BinaryTree<**Type**>**&** Tree )**;** //输入二叉树

**friend ostream& operator** << ( **ostream&** out, BinaryTree<**Type**>**&** Tree )**;** //输出二叉树

**}**

6-5 在结点个数为n (n>1)的各棵树中，高度最小的树的高度是多少？它有多少个叶结点？多少个分支结点？高度最大的树的高度是多少？它有多少个叶结点？多少个分支结点？

【解答】

结点个数为n时，高度最小的树的高度为1，有2层；它有n-1个叶结点，1个分支结点；高度最大的树的高度为n-1，有n层；它有1个叶结点，n-1个分支结点。

6-6 试分别画出具有3个结点的树和3个结点的二叉树的所有不同形态。

【解答】

具有3个结点的树 具有3个结点的二叉树

6-7 如果一棵树有n1个度为1的结点, 有n2个度为2的结点, … , nm个度为m的结点, 试问有多少个度为0的结点? 试推导之。

【解答】

总结点数 n = n0 + n1 + n2 + … + nm

总分支数 e = n-1 = n0 + n1 + n2 + … + nm-1

= m\*nm + (m-1)\*nm-1 + … + 2\*n2 + n1

则有 

6-8 试分别找出满足以下条件的所有二叉树:

(1) 二叉树的前序序列与中序序列相同;

(2) 二叉树的中序序列与后序序列相同;

(3) 二叉树的前序序列与后序序列相同。

【解答】

(1) 二叉树的前序序列与中序序列相同：空树或缺左子树的单支树；

(2) 二叉树的中序序列与后序序列相同：空树或缺右子树的单支树；

(3) 二叉树的前序序列与后序序列相同：空树或只有根结点的二叉树。

6-9 若用二叉链表作为二叉树的存储表示，试针对以下问题编写递归算法：

(1) 统计二叉树中叶结点的个数。

(2) 以二叉树为参数，交换每个结点的左子女和右子女。

【解答】

(1) 统计二叉树中叶结点个数

**int** BinaryTree**<Type> ::** leaf ( BinTreeNode**<Type>** \* ptr ) **{**

**if** ( ptr == NULL ) **return** 0**;**

**else if** ( ptr->leftChild == NULL **&&** ptr->rightChild == NULL) **return** 1**;**

**else** **return** leaf ( ptr->leftChild ) + leaf ( ptr->rightChild )**;**

**}**

(2) 交换每个结点的左子女和右子女

**void** BinaryTree**<Type> ::** exchange ( BinTreeNode**<Type>** \* ptr ) **{**

BinTreeNode**<Type>** \* temp**;**

**if** ( ptr->leftChild != NULL || ptr->rightChild != NULL ) **{**

temp = ptr->leftChild**;**

ptr->leftChild = ptr->rightChild**;**

ptr->rightChild = temp**;**

exchange ( ptr->leftChild )**;**

exchange ( ptr->rightChild )**;**

**}**

**}**

6-10 一棵高度为h的满k叉树有如下性质: 第h层上的结点都是叶结点, 其余各层上每个结点都有k棵非空子树, 如果按层次自顶向下, 同一层自左向右, 顺序从1开始对全部结点进行编号, 试问:

(1) 各层的结点个数是多少?

(2) 编号为i的结点的父结点(若存在)的编号是多少?

(3) 编号为i的结点的第m个孩子结点(若存在)的编号是多少?

(4) 编号为i的结点有右兄弟的条件是什么? 其右兄弟结点的编号是多少?

(5) 若结点个数为 n, 则高度h是n 的什么函数关系?

【解答】

(1) ki ( i = 0, 1, ……, h )

(2) 

(3) ( i-1)\*k + m + 1

(4) ( i-1 ) % k ≠ 0 或  时有右兄弟，右兄弟为i + 1。

1. h = logk (n\*(k-1)+1)-1 (n = 0时h = -1 )

6-11 试用判定树的方法给出在中序线索化二叉树上：

【解答】

(1) 搜索指定结点的在中序下的后继。

设指针q指示中序线索化二叉树中的指定结点，指针p指示其后继结点。

q->rightThread == 1?

=

≠

q->rightChild == NULL ?

=

q无后继

≠

q的后继为q->rightChild

q的后继为q的右子树中中序下的第一个结点

找q的右子树中在中序下的第一个结点的程序为：

p = q->rightChild**;**

**while** ( p->leftThread == 1 ) p = p->leftChild**;** // p即为q的后继

(2) 搜索指定结点的在前序下的后继。

q->leftThread == 0 ?

=

q的后继为q->leftChild

≠

q->rightThread == 0 ?

=

q的后继为q->rightChild

≠

p = q**;**

**while** ( p->rightThread == 1 **&&**

p->rightChild != NULL ) p = p->rightChild**;**

**if** ( p->rightChild ==NULL ) q无后继**;**

**else**的后继为p->rightChild

(3) 搜索指定结点的在后序下的后继。

( p = parent (q ) ) == NULL ?

=

q的后继为p

≠

p->rightThread == 1 || p->rightChild == q ?

=

≠

q的后继为p的右子树中后序下的第一个结点

q无后继

可用一段遍历程序来实现寻找p的右子树中后序下的第一个结点：即该子树第一个找到的叶结点。找到后立即返回它的地址。

6-12 已知一棵完全二叉树存放于一个一维数组T[n]中，T[n]中存放的是各结点的值。试设计一个算法，从T[0]开始顺序读出各结点的值，建立该二叉树的二叉链表表示。

【解答】

**template <class Type>** //建立二叉树

**istream& operator** >> ( **istream&** in, BinaryTree<**Type**>**&** t ) **{**

**int** n**;**

**cout** << "Please enter the number of node **:** "**;** in >> n**;**

**Type** \*A = **new Type**[n+1]**;**

**for** ( **int** i = 0**;** i < n**;** i++ ) in >> A[i]**;**

t. ConstructTree( T, n, 0, ptr )**;**  //以数组建立一棵二叉树

**delete** [ ] A**;**

**return** in**;**

**}**

**template <class Type>**

**void** BinaryTree<**Type**> **::** ConstructTree ( **Type** T[ ], **int** n, **int** i, BinTreeNode<**Type**> \***&** ptr ) **{**

//私有函数 **:** 将用T[n]顺序存储的完全二叉树, 以i为根的子树转换成为用二叉链表表示的

//以ptr为根的完全二叉树。利用引用型参数ptr将形参的值带回实参。

**if** ( i >= n )ptr **=** NULL**;**

**else {**

ptr = **new** BinTreeNode<**Type**> ( T[i] )**;** //建立根结点

ConstructTree ( T, n, 2\*i+1, ptr->leftChild )**;**

ConstructTree ( T, n, 2\*i+2, ptr->rightChild )**;**

**}**

**}**

6-13 试编写一个算法，把一个新结点l作为结点s的左子女插入到一棵中序线索化二叉树中，s原来的左子女变成l的左子女。

【解答】

**template<class Type>**

**void** ThreadTree<**Type> ::** leftInsert ( ThreadNode<**Type**> \*s, ThreadNode<**Type**> \* l ) **{**

**if** ( s != NULL **&&** l != NULL ) **{**

l->leftChild = s->leftChild**;** l->leftThread = s->leftThread**;** //预先链接

l->rightChild = s**;** l->rightThread = 1**;** //新插入结点\*l的后继为\*s

s->leftChild = l**;** s->leftThread = 0**;**  //\*l成为\*s的左子女

**if** ( l->leftThread *==* 0 ) **{** //\*l的左子女存在

ThreadNode<**Type**> \* p = l->leftChild**;**

**while** ( p->rightThread *==* 0 ) //找\*l的左子树中序下最后一个结点

p = p->rightChild**;**

p->rightChild = l**;** //该结点的后继为\*l

**}**

**}**

**}**

6-14 写出向堆中加入数据4, 2, 5, 8, 3, 6, 10, 14时，每加入一个数据后堆的变化。

【解答】

以最小堆为例：

2

2

2

4

4

4

5

5

5

2

4

4

4

2

3

8

8

2

2

2

2

3

5

3

5

5

3

3

5

10

6

4

8

10

4

8

6

4

8

4

8

6

14

6-16 请画出右图所示的树所对应的二叉树。

【解答】

1

1

2

2

5

4

4

3

3

对应二叉树

5

7

6

6

8

8

7

6-17 在森林的二叉树表示中，用llink存储指向结点第一个子女的指针，用rlink存储指向结点下一个兄弟的指针，用data存储结点的值。如果我们采用静态二叉链表作为森林的存储表示，同时按森林的先根次序依次安放森林的所有结点，则可以在它们的结点中用只有一个二进位的标志ltag代替llink，用rtag代替rlink。并设定若ltag = 0，则该结点没有子女，若ltag ≠ 0，则该结点有子女；若rtag = 0，则该结点没有下一个兄弟，若rtag ≠ 0，则该结点有下一个兄弟。试给出这种表示的结构定义，并设计一个算法，将用这种表示存储的森林转换成用llink - rlink表示的森林。

【解答】

A

F

B

H

F

A

对应二叉树

H

G

C

I

J

G

D

C

B

I

D

K

E

J

K

E

1 -1 -1 4 -1 6 -1 8 9 -1 -1

A B C D E F G H I K J

5 2 3 -1 -1 7 -1 -1 10 -1 -1

0 1 2 3 4 5 6 7 8 9 10

*llink*

*data*

*rlink*

森林的左子女-右兄弟表示的静态二叉链表

0 1 2 3 4 5 6 7 8 9 10

1 0 0 1 0 1 0 1 1 0 0

A B C D E F G H I K J

1 1 1 0 0 1 0 0 1 0 0

*ltag*

*data*

*rtag*

森林的双标记表示

1. 结构定义

**template <class Type> class** LchRsibNode **{** //左子女-右兄弟链表结点类的定义

**protected:**

**Type** data**;** //结点数据

**int** llink, rlink**;** //结点的左子女、右兄弟指针

**public:**

LchRsibNode ( ) **:** llink(-1), rlink(-1) **{ }**

LchRsibNode ( **Type** x ) **:** llink(-1), rlink(-1), data(x) **{ }**

**}**

**template <class Type> class** DoublyTagNode **{** //双标记表结点类的定义

**protected:**

**Type** data**;** //结点数据

**int** ltag, rtag**;** //结点的左子女、右兄弟标记

**public:**

DoublyTagNode ( ) **:** ltag(0), rtag(0) **{ }**

DoublyTagNode ( **Type** x ) **:** ltag(0), rtag(0), data(x) **{ }**

**}**

**template <class Type> class** staticlinkList //静态链表类定义

**: public** LchRsibNode**<Type>, public** DoublyTagNode **<Type>{**

**private:**

LchRsibNode<**Type**> \*V**;** //存储左子女-右兄弟链表的向量

DoublyTagNode **<Type>** \*U**;** //存储双标记表的向量

**int** MaxSize**,** CurrentSize**;** //向量中最大元素个数和当前元素个数

**public:**

dstaticlinkList ( **int** Maxsz ) **:** MaxSize ( Maxsz ), CurrentSize (0) **{**

V = **new** LchRsibNode <**Type**> [Maxsz]**;**

U = **new** DoublyTagNode <**Type**> [Maxsz]**;**

**}**

1. 森林的双标记先根次序表示向左子女-右兄弟链表先根次序表示的转换

**void** staticlinkList<**Type**> **::** DtagF-LchRsibF ( ) **{**

Stack<**int**> st**; int** k**;**

**for** ( **int** i = 0**;** i < CurrentSize**;** i++ ) **{**

**switch** ( U[i].ltag ) **{**

**case** 0 **: switch** ( U[i].rtag ) **{**

**case** 0 **:** V[i].llink = V[i].rlink = -1**;**

**if** ( st.IsEmpty ( ) == 0 )

**{** k = st.GetTop ( )**;** st.Pop ( )**;** V[k].rlink = i + 1**; }**

**break;**

**case** 1 **:** V[i].llink = -1**;** V[i].rlink = i + 1**; break;**

**}**

**break;**

**case** 1 **: switch** ( U[i].rtag ) **{**

**case** 0 **:** V[i].llink = i + 1**;** V[i].rlink = -1**; break;**

**case** 1 **:** V[i].llink = i + 1**;**  st.Push ( i )**;**

**}**

**}**

**}**

**}**

6-18 二叉树的双序遍历(Double-order traversal)是指：对于二叉树的每一个结点来说，先访问这个结点，再按双序遍历它的左子树，然后再一次访问这个结点，接下来按双序遍历它的右子树。试写出执行这种双序遍历的算法。

【解答】

**template <class Type>**

**void** BinaryTree<**Type**> **::** Double\_order ( BinTreeNode<**Type**> \*current )**{**

**if** ( current != NULL ) **{**

**cout** << current->data << ' '**;**

Double\_order ( current->leftChild )**;**

**cout** << current->data << ' '**;**

Double\_order ( current->rightChild )**;**

**}**

**}**

6-19 已知一棵二叉树的前序遍历的结果是ABECDFGHIJ, 中序遍历的结果是EBCDAFHIGJ, 试画出这棵二叉树。

【解答】

当前序序列为ABECDFGHIJ，中序序列为EBCDAFHIGJ时，逐步形成二叉树的过程如下图所示：

A

A

A

A

F

B

B

F

F

B

G

E

C

G

E

C

HIGJ

CD

E

FHIGJ

H

D

J

HI

J

D

EBCD

I

6-20 已知一棵树的先根次序遍历的结果与其对应二叉树表示(长子-兄弟表示)的前序遍历结果相同, 树的后根次序遍历结果与其对应二叉树表示的中序遍历结果相同。试问利用树的先根次序遍历结果和后根次序遍历结果能否唯一确定一棵树? 举例说明。

【解答】

因为给出二叉树的前序遍历序列和中序遍历序列能够唯一地确定这棵二叉树，因此，根据题目给出的条件，利用树的先根次序遍历结果和后根次序遍历结果能够唯一地确定一棵树。

例如，对于题6-16所示的树

1

1

2

2

对应二叉树

3

5

4

4

3

5

6

7

6

8

7

8

对应二叉树的前序序列为1, 2, 3, 4, 5, 6, 8, 7；中序序列为3, 4, 8, 6, 7, 5, 2, 1。

原树的先根遍历序列为 1, 2, 3, 4, 5, 6, 8, 7；后根遍历序列为3, 4, 8, 6, 7, 5, 2, 1。

6-21 给定权值集合**{**15, 03, 14, 02, 06, 09, 16, 17**}**, 构造相应的霍夫曼树, 并计算它的带权外部路径长度。

【解答】

(Ⅰ)

05

17

16

09

06

14

15

F：

17

16

09

06

02

14

15

02

03

03

(Ⅱ)

17

16

09

14

15

20

16

14

15

(Ⅲ)

17

11

09

11

06

05

02

03

06

05

(Ⅴ)

33

20

29

02

03

29

16

17

20

(Ⅳ)

16

17

11

09

15

14

14

11

09

15

05

06

06

05

02

03

02

03

82

(Ⅶ)

49

33

(Ⅵ)

49

33

17

29

20

16

17

16

29

20

14

11

09

15

11

09

15

14

06

05

06

05

03

02

03

02

此树的带权路径长度WPL = 229。

6-22 假定用于通信的电文仅由8个字母c1, c2, c3, c4, c5, c6, c7, c8组成, 各字母在电文中出现的频率分别为5, 25, 3, 6, 10, 11, 36, 4。试为这8个字母设计不等长Huffman编码, 并给出该电文的总码数。

【解答】

已知字母集 **{** c1, c2, c3, c4, c5, c6, c7, c8 **}**，频率 **{**5, 25, 3, 6, 10, 11, 36, 4 **}**，则Huffman编码为

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| c1 | c2 | c3 | c4 | c5 | c6 | c7 | c8 |
| 0110 | 10 | 0000 | 0111 | 001 | 010 | 11 | 0001 |

电文总码数为 4 \* 5 + 2 \* 25 + 4 \* 3 + 4 \* 6 + 3 \* 10 + 3 \* 11 + 2 \* 36 + 4 \* 4 = 257

**1**

**0**

100

**1**

39

61

**1**

**0**

**0**

**1**

**1**

**0**

**0**

**C2**

17

22

36

25

**C7**

7

11

11

10

**1**

**1**

**0**

**0**

**C6**

**C5**

5

6

3

4

**C4**

**C1**

**C8**

**C3**

6-23 给定一组权值: 23, 15, 66, 07, 11, 45, 33, 52, 39, 26, 58, 试构造一棵具有最小带权外部路径长度的扩充4叉树, 要求该4叉树中所有内部结点的度都是4, 所有外部结点的度都是0。这棵扩充4叉树的带权外部路径长度是多少?

【解答】

权值个数n = 11，扩充4 叉树的内结点的度都为4，而外结点的度都为0。设内结点个数为n4，外结点个数为n0，则可证明有关系n0 = 3 \* n4 + 1。由于在本题中n0 = 11≠3 \* n4 +1，需要补2个权值为0的外结点。此时内结点个数n4 = 4。

仿照霍夫曼树的构造方法来构造扩充4叉树，每次合并4个结点。

66

58

52

45

39

33

26

23

15

11

70

0

0

0

0

11

70

52

39

33

45

18

23

15

26

169

82

66

58

375

此树的带权路径长度WPL = 375 + 82 + 169 + 18 = 644。

**四、其他练习题**

6-24 填空题

(1) 对于一棵具有n个结点的树，该树中所有结点的度数之和为\_\_\_\_\_\_\_。

(2) 假定一棵三叉树的结点个数为50，则它的最小高度为\_\_\_\_\_\_\_，最大高度为\_\_\_\_\_\_\_。

(3) 一棵高度为h的四叉树中，最多含有\_\_\_\_\_\_\_结点。

(4) 在一棵三叉树中，度为3的结点数有2个，度为2的结点数有1个，度为1的结点数为2个，那么度为0的结点数有\_\_\_\_\_\_\_个。

(5) 一棵高度为5的满二叉树中的结点数为\_\_\_\_\_\_\_个，一棵高度为3的满四叉树中的结点数为\_\_\_\_\_\_\_\_个。

(6) 在一棵二叉树中，假定度为2的结点有5个，度为1的结点有6个，则叶子结点数有\_\_\_\_\_\_\_\_个。

(7) 对于一棵含有40个结点的理想平衡树，它的高度为\_\_\_\_\_\_\_\_。

(8) 若对一棵二叉树从0开始进行结点编号，并按此编号把它顺序存储到一维数组a中，即编号为0的结点存储到a[0]中，其余类推，则a[i]元素的左孩子元素为\_\_\_\_\_\_\_\_，右孩子元素为\_\_\_\_\_\_\_\_，双亲元素（i ≥ 1）为\_\_\_\_\_\_\_\_。

(9) 对于一棵具有n个结点的二叉树，对应二叉链表中指针总数为\_\_\_\_\_\_\_\_个，其中\_\_\_\_\_\_\_\_个用于指向子女结点，\_\_\_\_\_\_\_\_个指针空闲着。

(10) 在一棵高度为h的理想平衡树中，最少含有\_\_\_\_\_\_\_个结点，最多含有\_\_\_\_\_\_\_\_ 个结点。

(11) 在一个堆的顺序存储中，若一个元素的下标为i，则它的左子女元素的下标为\_\_\_\_\_\_\_\_，右子女元素的下标为\_\_\_\_\_\_\_\_。

(12) 在一个最小堆中，堆顶结点的值是所有结点中的\_\_\_\_\_\_\_\_，在一个最大堆中，堆顶结点的值是所有结点中的\_\_\_\_\_\_\_\_。

(13) 当向一个最小堆插入一个具有最小值的元素时，该元素需要逐层\_\_\_\_\_\_\_\_调整，直到被调整到\_\_\_\_\_\_\_\_位置为止。

(14) 当从一个最小堆中删除一个元素时，需要把\_\_\_\_\_\_\_\_元素填补到\_\_\_\_\_\_\_\_位置，然后再按条件把它逐层\_\_\_\_\_\_\_\_调整。

(15) 在霍夫曼编码中，若编码长度只允许小于等于4，则除了已对两个字符编码为0和10外，还可以最多对\_\_\_\_\_\_\_\_个字符编码。【解答】

(1) n-1 (2) 4, 49 (3) (4h+1-1) / 3 (4) 6

(5) 63, 85 (6) 6 (7) 5 (8) 2\*i+1, 2\*i+2, ⎡(i-1)/2⎤

(9) 2n, n-1, n+1 (10) 2h, 2h+1-1 (11) 2i+1, 2i+2 (12) 最小者, 最大者

(13) 向上, 堆顶 (14) 最后, 堆顶, 向下 (15) 4

6-25 n个结点可构造出多少种不同形态的二叉树? 若有3个数据1, 2, 3，输入它们构造出来的中序遍历结果都为1, 2, 3的不同二叉树有哪些?

****【解答】

有 种。当n=3时，中序遍历都为1, 2, 3的不同二叉树有5种：

②③③①①

② ① ② ③

① ③ ① ② ③ ②

6-26 判断下列叙述的对错。如果正确，在题前打“√”，否则打“×”。

(1) 二叉树是树的特殊情形。

(2) 若有一个结点是二叉树中某个子树的中序遍历结果序列的最后一个结点，则它一定是该子树的前序遍历结果序列的最后一个结点。

(3) 若有一个结点是二叉树中某个子树的前序遍历结果序列的最后一个结点，则它一定是该子树的中序遍历结果序列的最后一个结点。

(4) 若有一个叶子结点是二叉树中某个子树的中序遍历结果序列的最后一个结点，则它一定是该子树的前序遍历结果序列的最后一个结点。

(5) 若有一个叶子结点是二叉树中某个子树的前序遍历结果序列的最后一个结点，则它一定是该子树的中序遍历结果序列的最后一个结点。

【解答】

(1) √ (2) × (3) × (4) √ (5) ×

6-27 下面是一个二叉树的前序遍历的递归算法。

**void** PreOrder ( BinTreeNode \*t ) **{**

**if** ( t != NULL ) **{** //递归结束条件

**cout** << t->data**;** //访问(输出)根结点

PreOrder ( t->leftChild )**;** //前序遍历左子树

PreOrder ( t->rightChild )**;** //前序遍历右子树

**}**

**}**

1. 改写PreOrder算法，消去第二个递归调用PreOrder (t->rightChild )**;**

(2) 利用栈改写PreOrder算法，消去两个递归调用。

【解答】

(1) 消去第二个递归语句时，视第一个递归语句为一般语句，按尾递归处理。

**void** PreOrder ( BinTreeNode \*t ) **{**

**while** ( t != NULL ) **{ //**按尾递归改为循环

**cout** << t->data**;**

PreOrder ( t->leftChild )**;**

t = t->rightChild**; //**向右子树循环

**}**

**}**

(2) 定义一个栈，在访问某一个结点时保存其右、左子女结点的地址。下一步将先从栈中退出右子女结点，对其进行遍历，然后从栈中退出左子女结点，对其进行遍历。

**#include** <iostream.h>

**#include** “stack.h”

#**include** “BinTree.h”

**void** PreOrder ( BinTreeNode \*t ) **{**

BinTreeNode \* p**;**

Stack S**;** S.push ( t )**;**

**while** ( ! S.IsEmpty ( ) ) **{**

p = S.GetTop ( )**;**  S.Pop ( )**;**

**cout** << p->getData ( )**;**

if ( p->rightChild ( ) != NULL ) S.push ( p->rightChild ( ) )**;**

if ( p->leftChild ( ) != NULL ) S.push ( p->leftChild ( ) )**;**

**}**

**}**

6-28 设二叉树采用二叉链表表示，指针root指向根结点，试编写一个在二叉树中查找值为*x*的结点，并打印该结点所有祖先结点的算法。在此算法中，假设值为*x*的结点不多于一个。

【解答】

此算法采用后序的非递归遍历形式。因退栈时需要区分其左、右子树是否已经遍历，故在结点进栈时附带有一个标志，＝0，进入左子树，＝1，进入右子树。用栈S1保存结点指针ptr，用栈S2保存标志tag。这两个栈的进栈与退栈是同步的。

#**include** <iostream.h>

#**include** “stack.h”

**template <class Type> void** BinaryTree **<Type> ::**

Find-Print ( BinTree **<Type>** **&** BT**;** **Type** x ) **{**

Stack < BinTreeNode <**Type**> \* > S1**;** Stack **<int>** S2**;**

BinTreeNode **<Type>** \* t = BT.root**; int** i, top**;**

**while** ( t != NULL **&&** t->data != x **||** ! S1.IsEmpty( ) ) **{**

**while** ( t != NULL **&&** t->data != x ) **{** //寻找值为x的结点

S1.Push ( t )**;** S2.Push ( 0 )**;** //进栈, 作向左子树遍历标志

t = t->leftChild**;** //转向左子树

**}**

**if** ( t != NULL **&&** t->data == x ) **{**  //找到值为x的结点

**while** ( ! S1.IsEmpty( ) ) **{ cout <<** S1.GetTop( )->data **<< “ << “;** S1.Pop( )**; }**

**break;**

**}**

**else {** //未找到值为x的结点

**while** ( !S2.IsEmpty( ) **&&** S2.GetTop ( ) == 1 ) **{** S1.Pop ( )**;** S2.Pop ( )**; }**

**if** ( ! S2.IsEmpty( ) ) **{**

S2.Pop( )**;** S2.Push (1)**; //**改向右子树遍历的标志

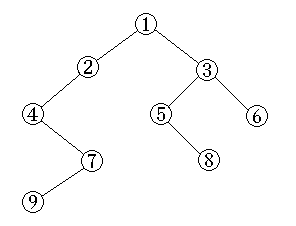
t = S1.GetTop( )->rightChild; //转向右子树

**}**

**}**

**}**

**}**



例如，搜索值为9的结点

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  | → | ⑦ | 0 |
| → | ④ | 0 |  | ④ | 1 |
|  | ② | 0 |  | ② | 0 |
|  | ① | 0 |  | ① | 0 |

S1 S2 S1 S2

6-29 设一棵二叉树以二叉链表表示，试以成员函数形式编写有关二叉树的递归算法：

(1) 统计二叉树中度为1的结点个数。

(2) 统计二叉树中度为2的结点个数。

(3) 统计二叉树中度为0（叶结点）的结点个数。

(4) 统计二叉树的高度。

(5) 统计二叉树的宽度，即在二叉树的各层上，具有结点数最多的那一层上结点总数。

(6) 从二叉树中删去所有叶结点。

(7) 计算二叉树中指定结点 \*p 所在层次。

(8) 计算二叉树中各结点中的最大元素的值。

(9) 以前序次序输出一棵二叉树所有结点的数据值及结点所在层次。

【解答】

(1) 统计二叉树中度为1的结点个数。

**template <class Type>**

**int** BinaryTree<**Type> ::** Degrees1 ( BinTreeNode**<Type>** \* t ) **const {**

**if** ( t == NULL ) **return** 0**;**

**if** ( t->leftChild != NULL **&&** t->rightChild == NULL **||**

t->leftChild == NULL **&&** t->rightChild != NULL ) **return** 1**;**

**return** Degrees1 ( t->leftChild, k ) + Degrees1 ( t->rightChild, k )**;**

**}**

(2) 统计二叉树中度为2的结点个数。

**template <class Type>**

**int** BinaryTree<**Type> ::** Degrees2 ( BinTreeNode**<Type>** \* t ) **const** **{**

**if** ( t == NULL ) **return** 0**;**

**if** ( t->leftChild != NULL **&&** t->rightChild != NULL ) return 1**;**

**return** Degrees2 ( t->leftChild, k ) + Degrees2 ( t->rightChild, k )**;**

**}**

(3) 统计二叉树中度为0（叶结点）的结点个数。

**template <class Type>**

**int** BinaryTree<**Type> ::** leaves ( BinTreeNode**<Type>** \* t ) **const** **{**

**if** ( t == NULL ) **return** 0**;**

**if** ( t->leftChild == NULL **&&** t->rightChild == NULL ) **return** 1**;**

**return** leaves ( t->leftChild ) + leaves ( t->rightChild )**;**

**}**

(4) 统计二叉树的高度。

**template <class Type>**

**int** BinaryTree**<Type>** **::** height ( BinTreeNode<**Type>** \*t ) **{**

**if** ( t *==* NULL ) **return** –1**;**

**int** hl = height ( t->leftChild )**;**

**int** hr = height ( t->rightChild )**;**

**return** 1+ ( hl > hr **?** hl **:** hr )**;**

**}**

(5) 统计二叉树的宽度，即在二叉树的各层上，具有结点数最多的那一层上结点总数。

本题的想法是：先用前序遍历求出每一层的宽度，再求出最大宽度，即树的宽度。

① 求每一层宽度的算法：

**template <class Type>**

**int** BinaryTree**<Type>** **::** levelNumber ( BinTreeNode<**Type>** \*t, **int** a[ ], **int** h ) **{**

//求以\*t为根的子树中各层的宽度, 存放在a[ ] 中, h是 \*t所在层次号,

//要求在主程序中将a[h]初始化为0

**if** ( t != NULL ) **{** //若空树, 不统计

a[h] += 1**;** //第h层宽度加一

levelNumber ( t->leftChild, a, h+1 )**; //**递归统计左子树中各层宽度

levelNumber ( t->rightChild, a, h+1 )**; //**递归统计右子树中各层宽度

**}**

**}**

② 求二叉树的宽度的算法

**template <class Type>**

**int** BinaryTree**<Type>** **::** width ( BinTreeNode<**Type>** \*t ) **{**

**int** a[n+1], h = 0, i, wid**;**

**for** ( i = 0**;** i <= n**;** i++ ) a[i] = 0**; //**统计数组a[ ]初始化

levelNumber ( t**,** a**,** h )**; //**调用求各层宽度的算法

wid = a[0]**;**

**for** ( i = 1**;** i <= n**;** i++ )

**if** ( a[i] > wid ) wid = a[i]**;**

**return** wid**;**

**}**

(6) 从二叉树中删去所有叶结点。

**template <class Type>**

**void** BinaryTree<**Type**> **::** defoliate ( BinTreeNode<**Type**> \***&** t ) **{**

**if** ( t *==* NULL ) **return;**

**if** ( t->leftChild *==* NULL **&&** t->rightChild *==* NULL )

**{ delete** t**;** t = NULL**; }**

**else {**

defoliate ( t->leftChild )**;**

defoliate ( t->rightChild )**;**

**}**

**}**

(7) 计算二叉树中指定结点 \*p 所在层次。

**template <class Type>**

**int** BinaryTree**<Type>** **::** level ( BinTreeNode**<Type>** \* t, BintreeNode**<Type>** \*p ) **{**

**if** ( t *==* NULL ) **return** -1**;**

**if** ( t *==* p ) **return** 0**;**

**if** ( ( leftSubTreelevel = level ( t->leftChild, p ) ) > -1 )

**return** 1 + leftSubTreelevel**;**

**if** ( ( rightSubTreelevel = level ( t->rightChild, p ) ) > -1 )

**return** 1 + rightSubTreelevel**;**

**return** -1**;**

**}**

(8) 计算二叉树中各结点中的最大元素的值。

**template <class Type>**

**Type** BinaryTree<**Type> ::** MaxValue ( BinTreeNode**<Type>** \* t, **Type** max ) **{**

**if** ( t != NULL ) **{**

**if** ( t->data > max ) max = t->data**;**

max = MaxValue ( t->leftChild, max )**;**

max = MaxValue ( t->rightChild, max )**;**

**}**

**return** max**;**

**}**

(9) 以前序次序输出一棵二叉树所有结点的数据值及结点所在层次。

**#include** <iostream.h>

**template <class Type>**

**void** BinaryTree **<Type> ::** nodePrint ( BinTreeNode<**Type**> \*t**, int** i ) **{**

**if** ( t != NULL ) **{**

**cout <<** t->data <<“,” << i << **endl;**

nodePrint ( t->leftChild, i+1 )**;**

nodePrint ( t->rightChild, i+1 )**;**

**}**

**}**

6-30 已知一棵具有n个结点的完全二叉树被顺序存储于一维数组的A[n]中，试编写一个算法打印出编号为i的结点的双亲和所有孩子。

【解答】

**template <class Type>** **void** Request ( **Type** T[ ], **int** n, **int** i ) **{**

**if** ( i > n ) **{**

**cerr** **<<** "编号为" << i << "的结点不存在！" << **endl; exit** (1)**;**

**}**

**cout** << "Current Node : " << T[i] << **endl;**

**int** j = (i-1)/2**;** //下标为j的结点是下标为i结点的双亲

**if** ( j >= 0 ) **cout** << "parent : " << T[j] << **endl;**

**else cout** << "二叉树的根结点没有双亲！" << **endl;**

**if** ( 2 \* i +1 < n ) **cout** << "left Child : " << T[2\*i+1] << **endl;**

**else cout <<** "编号为" << i << "的结点没有左子女！" << **endl;**

if ( 2 \* i +2 < n ) **cout** << "right Child : " << T[2\*i+2] << endl;

else cout << "编号为" << i << "的结点没有右子女！" << **endl;**

**}**

6-31 试编写一个算法，将用二叉链表表示的完全二叉树转换为二叉树的顺序（数组）表示。

【解答】

将完全二叉树所有结点从根开始，自顶向下，同一层自左向右连续编号，根结点的编号为0。

**template <class Type>**

**void** linkedToSequent ( BinTreeNode**<Type>** \*t**, Type** a[ ]**, int** i ) **{**

//指针t是完全二叉树的根指针。

**if** ( t != NULL ) **{**

a[i] = t->getData( )**;**

linkedToSequent ( t->getLeftChild( ), a, 2\*i+1 )**;**

linkedToSequent ( t->getRightChild( ), a, 2\*i+2 )**;**

**}**

**}**

主程序调用方式 linkedToSequent ( T.root**,** a**,** 0 )**;**

6-32 一棵完全二叉树存放于一个一维数组T[n]中，T[n]中存放的是各结点的值。试编写一个算法，从T[0]开始顺序读出各结点的值，建立该二叉树的二叉链表表示。

【解答】

**template <class Type>**

**istream& operator** >> ( **istream&** in, BinaryTree<**Type**>**&** t ) **{**

**int** n**;**

**cout <**< "Please enter the number of node : "**;** in >> n**;**

**Type** \*A = **new Type**[n]**;**

**for** ( **int** i = 0**;** i < n**;** i++ ) in >> A[i]**;**

ConstructTree ( A, n, 0, t )**;** //以数组建立一棵二叉树

**delete** [n]A**;**

**return** in**;**

**}**

**template <class Type> void** BinaryTree<**Type> ::**

ConstructTree ( **Type** T[ ]**, int** n**, int** i**,** BinTreeNode<**Type>** \***&** ptr ) **{**

//将用T[n]顺序存储的完全二叉树, 以i为根的子树转换成为用二叉链表表示的

//以ptr为根的完全二叉树。利用引用型参数ptr将形参的值带回实参。

**if** ( i >= n ) ptr = NULL**;**

**else {**

ptr = **new** BinTreeNode<**Type**> ( T[i] )**;** //建立根结点

ConstructTree ( T, n, 2\*i+1, ptr->leftChild )**; //**递归建立左子树

ConstructTree ( T, n, 2\*i+2, ptr->rightChild )**; //**递归建立右子树

**}**

**}**

6-33 设用二叉链表表示一棵二叉树，试采用输入广义表表示建立二叉树。具体规定如下：

(1) 树的根结点作为由子树构成的表的表名放在表的最前面；

A

C

B

D

E

F

G

(2) 每个结点的左子树和右子树用逗号隔开。若仅有右子树没有左子树，逗号不能省略。

(3) 在整个广义表表示输入的结尾加上一个特殊的符号（例如“#”）表示输入结束。

例如，对于如右图所示的二叉树，其广义表表示为

A ( B ( D, E ( G, ) ), C ( , F ) )

【解答】

此算法的基本思路是：依次从保存广义表的字符串ls中输入每个字符。若遇到的是字母（假定以字母作为结点的值），则表示是结点的值，应为它建立一个新的结点，并把该结点作为作为左子女（当k = 1）或右子女（当k = 2）链接到其双亲结点上。若遇到的是左括号”(”，则表明子表的开始，将k置为1；若遇到的是右括号”)”，则表明子表结束。若遇到的是逗号”,”，则表示以左子女为根的子树处理完毕，应接着处理以右子女为根的子树，将k置为2。

在算法中使用了一个栈s，在进入子表之前，将根结点指针进栈，以便括号内的子女链接之用。在子表处理结束时退栈。

下面给出建立二叉树的算法。

#**include** <iostream.h>

#**include** “stack.h”

**void** BinaryTree **::** CreateBinTree ( BinTreeNode \***&** BT, **char** ls ) {

Stack<BinTreeNode \*> S**;**

BT = NULL**;** //置空二叉树

BinTreeNode \*p, \*q**; int** k**;**

**istream** ins (ls)**;** //把串ls定义为输入字符串流对象ins

**char** ch**;**  ins >> ch**;** //从ins顺序读入一个字符

**while** ( ch != ‘#’ ) **{** //逐个字符处理，直到遇到‘#’为止

**switch** ( ch ) **{**

**case** ‘(’ **:** S.push ( p )**;** k=1**;**  **break;**

**case** ‘)’ **:** S.pop ( )**; break;**

**case** ‘,’ **:** k=2**; break;**

**default :** p = **new** BinTreeNode ( ch )**;**

**if** ( BT == NULL ) BT = p**;**

**else {**

q = S.GetTop ( )**;** S.Pop ( )**;**

**if** ( k==1 ) q->leftChild = p**;**

**else** q->rightChild = p**;**

S.Push ( q )**;**

**}**

**}**

ins >> ch**;**

**}**

**}**

6-34 以二叉链表为存储表示，试编写一个算法，用括号形式 key ( LT, RT ) 输出二叉树的各个结点。其中，key是根结点的数据， LT和RT是括号形式的左子树和右子树。要求空树不打印任何信息，一个结点的树的打印形式是x，而不应是 (x, ) 的形式。

【解答】

**#include** <iostream.h>

**template <class Type> void** printBinTree ( BinTreeNode**<Type>** \*t ) **{**

**if** ( t != NULL ) **{**

**cout** << t->getData ( )**;**

**if** ( t->getLeftChild ( ) != NULL || t->getRightChild ( ) != NULL ) **cout** << “(”**;**

printBinTree ( t->getLeftChild ( ) )**;**

**if** ( t->getRightChild ( ) != NULL ) **cout** << “,”;

printBinTree ( t->getRightChild ( ) )**;**

**cout <<** “)”**;**

**}**

**}**

6-35 请回答下列关于堆（Heap）的一些问题：

(1) 堆的存储表示是顺序的，还是链接的？

(2) 设有一个最小堆, 即堆中任意结点的关键码均小于或等于它的左子女和右子女的关键码。其具有最大值的元素可能在什么地方？

(3) 对n个元素进行初始建堆的过程中，最多做多少次数据比较（不用大O表示法）？

【解答】

(1) 堆是以完全二叉树的顺序（数组）表示来存放的，所以其存储表示是顺序的。

(2) 在最小堆中具有最大值的元素在堆的某一个叶子结点上。

(3) 若设堆中有n个结点，且2k-1 ≤ n < 2k，则对应的完全二叉树有k层。在第i层上的结点数 ≤ 2i (i = 0, 1, …, k-1)。在第一个形成初始堆的**for**循环中对每一个非叶结点调用了一次堆调整算法FilterDown ( )，因此该循环所用的计算时间为：



其中，i是层序号，2i是第i层的最大结点数，(k-i-1)是第i层结点能够移动的最大距离。



6-36 从供选择的答案中选择与下面有关二叉树和森林的叙述中各括号相匹配的词句，将其编号填入相应的括号内。

(1) 设二叉树有n个结点且根结点处于第0层，则其高度为（ A ）。

(2) 设高度为h的空二叉树的高度为-1，只有一个结点的二叉树的高度为0，若设二叉树只有度为2和度为0的结点，则该二叉树中所含结点至少有（ B ）个。

(3) 设森林F中有4棵树，第1、2、3、4棵树的结点个数分别为n1、n2、n3、n4，当把森林F转换成一棵二叉树后，其根结点的右子树中有（ C ）个结点。

(4) 设森林F中有4棵树，第1、2、3、4棵树的结点个数分别为n1、n2、n3、n4，当把森林F转换成一棵二叉树后，其根结点的左子树中有（ D ）个结点。

(5) 将含有82个结点的完全二叉树从根结点开始顺序编号，根结点为第0号，其他结点自上向下，同一层自左向右连续编号。则第40号结点的双亲结点的编号为（ E ）。

供选择的答案

A： ① n-1 ② ⎡log2(n+1)⎤ -1 ③ ⎣log2n⎦ +1 ④ 不确定

B： ① 2h ② 2h -1 ③ 2h +1 ④ h +1

C~D：① n1-1 ② n1+n2+n3 ③ n2+n3+n4 ④ n1

E： ① 20 ② 19 ③ 81 ④ 80

【解答】

A. ④ B. ③ C. ③ D. ① E. ②

6-37 一棵树的存储结构可以采用双亲表示法，即父指针数组表示法。试给出相应的类定义。其中，每个树结点包含两个成员：数据域data和双亲指针parent；树则有一个树结点数组NodeList[MaxSize]，MaxSize表示该数组的最大结点个数，size是当前结点个数，current指示最近操作结点位置，即当前指针。

【解答】

下面给出用双亲（父指针）表示的树和树结点类定义。

**template <class Type> class** Tree**;**

**template <class Type> class** TreeNode **{** //树结点类

**friend class** Tree**<Type>;**

**private:**

**Type** data**;** //结点数据

**int** parent**;**  //父结点指针 (用结点下标表示)

**};**

**template <class Type> class** Tree **{** //树类

**private:**

TreeNode**<Type>** \*NodeList**;** //结点表

**int** Size, MaxSize**;** //当前结点个数及结点表结点最大个数

**int** current**;** //当前结点指针

**public:**

Tree( **int** sz)**;** //构造函数

~Tree ( ) **{ delete** [ ] NedeList**;** **}** //析构函数

**int** Root ( )**;** //搜索根结点

**void** BuildRoot ( **const Type&** value )**;** //建立根结点

**int** FirstChild ( )**;** //搜索当前结点的第一个孩子

**int** NextSibling ( )**;** //搜索当前结点的下一个兄弟

**int** Parent ( )**;** //搜索当前结点的父结点

**Type** getData ( )**;** //检索当前结点数据成员的值

void setData ( **const Type&** value )**;** //修改当前结点数据成员的值

**int** InsertChild ( **const Type&** value )**;** //在当前结点下插入新孩子

**int** DeleteChild ( **int** i )**;** //删除当前结点的第i个孩子

**void** DeleteSubTree ( )**;** //删除以当前结点为根的子树

**int** IsEmpty ( ) **{ return** Size == 0**; }** //判树空否

**};**

**template <class Type>** Tree**<Type> ::** Tree ( **int** sz ) **:** MaxSize (sz) **{**

//构造函数, 建立父指针数组并初始化。

NodeList = **new** TreeNode[MaxSize]**;** //创建结点表

Size = 0**;** current = -1**;**

**}**

**template <class Type> int** Tree**<Type> ::** Root ( ) **{**

//搜索根结点

**if** ( Size != 0 ) **{** current = 0**;** **return** 1**; }**

current = -1**;** **return** 0**;**

**}**

**template <class Type> int** Tree**<Type> ::** BuildRoot ( **const Type&** value ) **{**

//建立树的根结点。树的根在NodeList[0]。

NodeList[0].data = value**;** NodeList[0].parent = -1**;** //根结点

Size = 1**;** current = 0**;**

**}**

**template <class Type> int** Tree**<Type> ::** FirstChild ( ) **{**

//函数执行后当前指针指到当前结点的第一个孩子结点并返回1, 若无孩子, 则当前指针为-1

//且函数返回0。

**int** i = current+1**;**  //从当前位置开始找孩子

**while** ( i < Size **&&** NodeList[i].parent != current ) i++**;**

**if** ( i < Size ) **{** current = i**;** **return** 1**;** **}**  //当前指针指到孩子结点并返回1

current = -1**; return** 0**;**

**}**

**template <class Type> int** Tree**<Type> ::** NextSibling ( ) **{**

//函数执行后当前指针指到当前结点的下一个兄弟结点并返回1, 若无兄弟, 则当前指针为-1

//且函数返回0。

**int** i = current+1**;**  //从当前位置开始找下一个兄弟

**while** ( i < Size **&&** NodeList[i].parent != NodeList[current].parent ) i++**;**

**if** ( i < Size ) **{** current = i**;** **return** 1**;** **}** //当前指针指到下一个兄弟结点, 并返回1

current = -1**;** **return** 0**;**

**}**

**template <class Type> int** Tree**<Type> ::** Parent ( ) **{**

//函数执行后当前指针指到当前结点的父结点并返回1, 若无父结点, 则当前指针为 -1且函

//数返回0。

**if** ( current < Size **&&** current > 0 ) **{** current = NodeList[current].parent**;** **return** 1**;** **}**

current = -1**;**  **return** 0**;**

**}**

**template <class Type> Type** Tree**<Type> ::** getData ( ) **{**

//函数返回当前结点中存放的值。

**if** ( current != -1 ) **return** NodeList[current].data**;**

**else return** NULL**;**

**}**

**template <class Type> int** Tree**<Type> ::** InsertChild ( **const Type&** value ) **{**

//在树中当前结点下插入数据为value的新孩子结点, 若父指针数组已满, 则不能插入, 函数

//返回0**;** 若插入成功, 则函数返回1。

**if** ( Size < MaxSize ) **{**

NodeList[++Size].data = value**;**  //值赋给孩子

NodeList[Size].parent = current**;** //链入父结点链

**return** 1**;**

**}**

**else return** 0**;**

**}**

**template <class Type> int** Tree**<Type> ::** DeleteChild ( **int** i ) **{**

//删除树中当前结点下的第i个孩子及其全部子孙结点, 若该结点的第i个孩子结点不存在,

//则函数返回0; 若删除成功, 则函数返回1。

**int** p = current**,** k = FirstChild ( )**;** //找当前结点p的第一个孩子

**if** ( !k ) **{** current = p**;** **return** 0**;** **}** //若未找到, 则退出

**int** j = 1**;**

**while** ( k **&&** j < i ) **{** j++**;**  k = NextSibling ( )**;** **}** //寻找当前结点的第i个兄弟

**if** ( !k ) **{** current = p**; return** 0**;** **}** //未找到

DeleteSubTree ( )**;** //找到, 删除以current为根的子树

current = p**;**  **return** 1**;**

**}**

**template <class Type> void** Tree**<Type> ::** DeleteSubTree ( ) **{**

//删除以当前结点为根结点的子树。

**if** ( current != -1 ) **{**

**int** t = current**;** k = FirstChild ( )**;**  //找当前结点的第一个孩子

**while** ( k ) **{** //找到

**int** p = current**;** //p记下当前的孩子

k = NextSibling ( )**;** **int** q = current**;** //q记下下一个孩子

current = p**;** DeleteSubTree ( )**;** //删除current为根的子树

current = q**;**

**}**

k = t+1**;**

**while** ( k < Size ) **{** //修改

**if** ( NodeList[k].parent > t ) NodeList[k].parent*--***;**

NodeList[k-1].parent = NodeList[k].parent**;**

NodeList[k-1].data = NodeList[k].data**;**

k++**;**

**}**

Size*--***;**

**}**

**}**

6-38 假设一棵树的存储结构采用双亲表示法，双亲指针数组为int parent[MaxSize]，其中MaxSize表示双亲指针数组的最大结点个数。树中各个结点按先根遍历次序存放，根结点存于parent[0]。试编写一个函数，计算p所指结点和q所指结点的最近公共祖先结点。

【解答】

这是一个二重循环。外层循环从结点 p向双亲方向循环，每变动一个结点，即对从结点 q到根的路径上各结点进行检测，遇到外层循环当前标定的结点即终止。此结点即为所求。

**int** CommonAncestry ( **int** parent[ ], **int** MaxSize, **int** p, **int** q ) **{**

**int** i, j;

**for** ( i = p**;** i != -1**;** i = parent[i] )

**for** ( j = q**;** j != -1**;** j = parent[j] )

**if** ( i == j ) **return** i**;**

**}**

6-39 可以用缩格（或移行）的文本形式（Indented Text）来表示一棵树的结点数据。例如，下面图（a）所示的树的缩格文本形式如图（b）所示。试设计一个算法，将用左子女-右兄弟链表表示的树用缩格文本形式输出。

A

B

C

D

E

F

G

H

I

J

K

A

B

C

D

E

F

G

H

I

J

K

(a) 树 (b) 树的缩格文本形式

【解答】

下面给出树的左子女-右兄弟表示的类声明。

**template <class Type> class** Tree**;**

**template <class Type> class** TreeNode **{** //树的结点类

**friend class<Type>** Tree**;**

**private:**

**Type** data**;** //数据

TreeNode**<Type>** \*firstChild, \*nextSibling**;** //子女及兄弟指针

**public:**

TreeNode ( **Type** value = 0, TreeNode**<Type>** \*fc = NULL, TreeNode**<Type>** \*ns = NULL )

：data (value), firstChild (fc), nextSibling (ns) **{ }** //构造函数

**Type** getData ( ) **{ return** data**; }** //取得结点数据

TreeNode**<Type>** \* getFirstChild ( ) **{ return** firstChild**; }** //取得第一个子女地址

TreeNode**<Type>** \* getNextSibling ( ) **{ return** nextSibling**; }** //取得下一个兄弟地址

**void** setData ( **Type** x ) **{** data = x**; }** //修改结点数据

**void** setFirstChild ( TreeNode**<Type>** \* fc ) **{** firstChild = fc**; }** //修改第一个子女地址

**void** setNextSibling ( TreeNode**<Type>** \* ns ) **{** nextSibling = ns**; }**//修改下一个兄弟地址

**};**

**template <class Type> class** Tree **{** //树类

**private:**

TreeNode**<Type>** \*root, \*current**;** //根指针及当前指针

**void** PreOrder ( **ostream &** out, TreeNode**<Type>** \*p )**;**

**int** Find ( TreeNode**<Type>** \*p, **Type** target )**;**

**void** RemovesubTree ( TreeNode**<Type>** \*p )**;**

**int** FindParent ( TreeNode**<Type>** \*t, TreeNode**<Type>** \*p )**;**

**public:**

Tree ( ) **{** root = current = NULL**;** **}** //构造函数, 建立空树

**void** BuildRoot ( **Type** rootVal )**;**

**int**Root ( )**;** //寻找根，使之成为当前结点

**int** FirstChild ( )**;** //寻找当前结点的第一个子女

**int** NextSibling ( )**;** //寻找当前结点的下一个兄弟

**int** Parent ( )**;** //寻找当前结点的双亲结点

//树的其他公共操作

……

**}**

将以左子女-右兄弟链表表示的树用缩格文本形式打印出来的算法如下：

#**include** <iostream.h>

#**include** “Tree.h”

**template <class Type> void** indentedText ( TreeNode**<Type>**\*t, **int** k ) **{**

**if** ( t != NULL ) **{**

**for** ( **int** i = 0**;** i < k**;** i++ ) **cout** << “ ”**;**

**cout** << t->getData ( )**;**

t = t->getFirstChild ( )**;**

**while** ( t != NULL ) **{**

indentedText ( t, k+1 )**;**

t = t->getNextSibling ( )**;**

**}**

**}**