第5章 二叉树

主要内容

- 5.1　二叉树的概念
- 5.2　二叉树的抽象数据类型
- 5.3　二叉树的存储结构
- 5.4　二叉搜索树
- 5.5　堆与优先队列
- 5.6　**Huffman**树及其应用
- 5.7　二叉树知识点总结

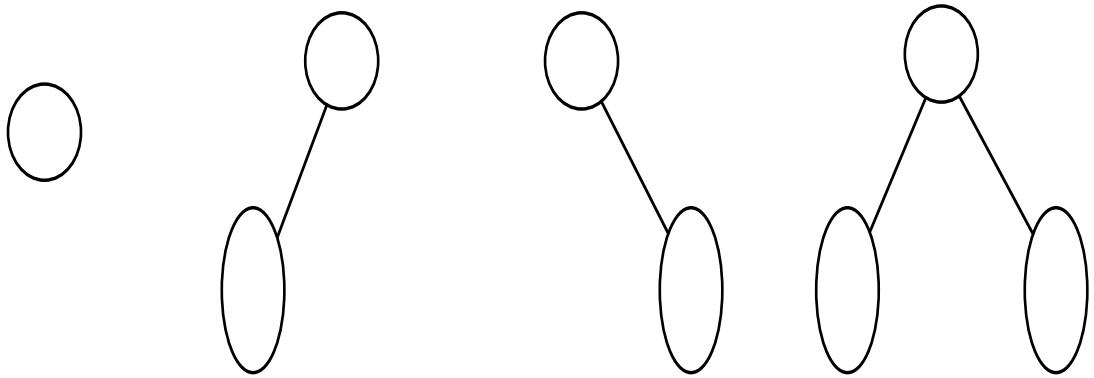
5.1 二叉树的概念

- 5.1.1 二叉树的定义及基本术语
- 5.1.2 满二叉树、 完全二叉树、 扩充二叉树
- 5.1.3 二叉树的主要性质

二叉树的定义

- **二叉树(binary tree)**由结点的有限集合构成，这个有限集合或者为空集(empty)，或者为由一个**根结点(root)**及两棵互不相交、分别称作这个根的**左子树(left subtree)**和**右子树(right subtree)**的二叉树组成的集合。

二叉树的五种基本形态



(a)空二叉树　(b)仅有根的二叉树　(c)右子树为空的二叉树　(d)左子树为空的二叉树　(e)左右子树均非空的二叉树

图5.1 二叉树的五种基本形态

二叉树相关术语（1）

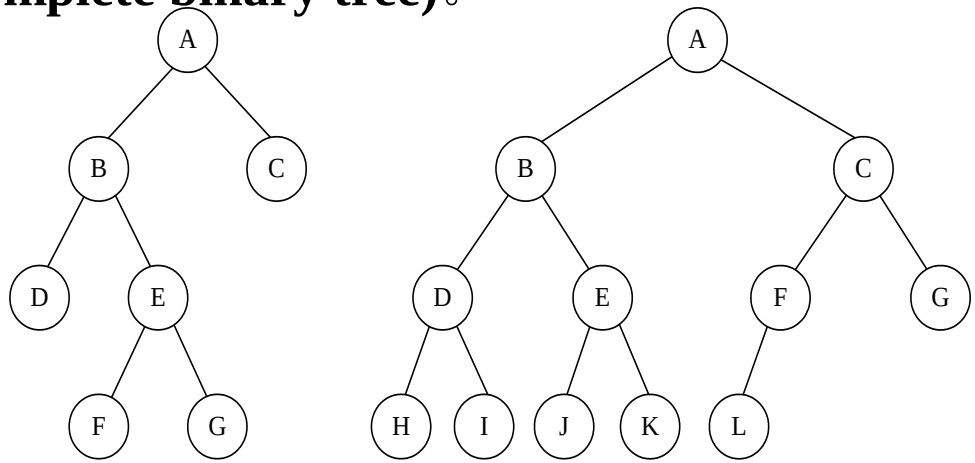
- 二叉树是由唯一的起始结点引出的结点集合。这个起始结点称为根(root)
- 二叉树中的任何非根结点都有且仅有一个前驱结点，称之为该结点的父结点(或称为双亲，parent)。根结点即为二叉树中唯一没有父结点的结点
- 二叉树中的任何结点最多可能有两个后继结点，分别称为左子结点(或左孩子、左子女，left child)和右子结点(或右孩子，右子女，right child)，具有相同父结点的结点之间互称兄弟结点(sibling)
- 二叉树中结点的子树数目称为结点的度(degree)。
- 没有子结点的结点称为叶结点 (leaf，也称“树叶”或“终端结点”)，叶结点的度为0。
- 除叶结点以外的那些非终端结点称为内部结点(或分支结点，internal node)
- 父结点k与子结点k’之间存在一条有向连线<k, k’>，称作边(edge)

二叉树相关术语（2）

- 若二叉树中存在结点序列{k0，k1，…，ks}，使得<k0，k1>，< k1，k2>，…，< ks-1，ks>都是二叉树中的边，则称从结点k0到结点ks存在一条路径(path)，该路径所经历的边的个数称为这条路径的路径长度(path length)。若有一条由 k到达ks的路径，则称k是ks的祖先(ancestor)，ks是k的子孙(descendant)
- 断掉一个结点与其父结点的连接，则该结点与其子孙构成的树就称为以该结点为根的子树(subtree)
- 从根结点到某个结点的路径长度称为结点的层数(level)，根结点为第0层，非根结点的层数是其父结点的层数加1

满二叉树与完全二叉树

- 如果一棵二叉树的任何结点，或者是树叶，或者恰有两棵非空子树，则这棵二叉树称作**满二叉树(full binary tree)**。
- 如果一棵二叉树最多只有最下面的两层结点度数可以小于2，并且最下面一层的结点都集中在该层最左边的连续位置上，则此二叉树称作**完全二叉树(complete binary tree)**。



(A)满二叉树　　(b)完全二叉树

图5.2 满二叉树和完全二叉树示例

完全二叉树

- **完全二叉树的特点是：**
  - 其叶结点只可能在层次最大的两层出现
  - 完全二叉树中由根结点到各个结点的路径长度总和在具有同样结点个数的二叉树中达到了最小，即任意一棵二叉树中根结点到各结点的最长路径一定不短于结点数目相同的完全二叉树中的路径长度

扩充二叉树

- 在二叉树里出现空子树的位置增加空树叶，所形成的二叉树称为**扩充二叉树(extended binary tree)**
- 构造一棵扩充二叉树只需要在原二叉树里度数为1的分支结点下增加一个空树叶，在原二叉树的树叶下面增加两个新的空树叶。
- 扩充二叉树是满二叉树，新增空树叶(以下称为**外部结点**)的个数等于原二叉树的结点(以下称为**内部结点**)个数加1

扩充二叉树

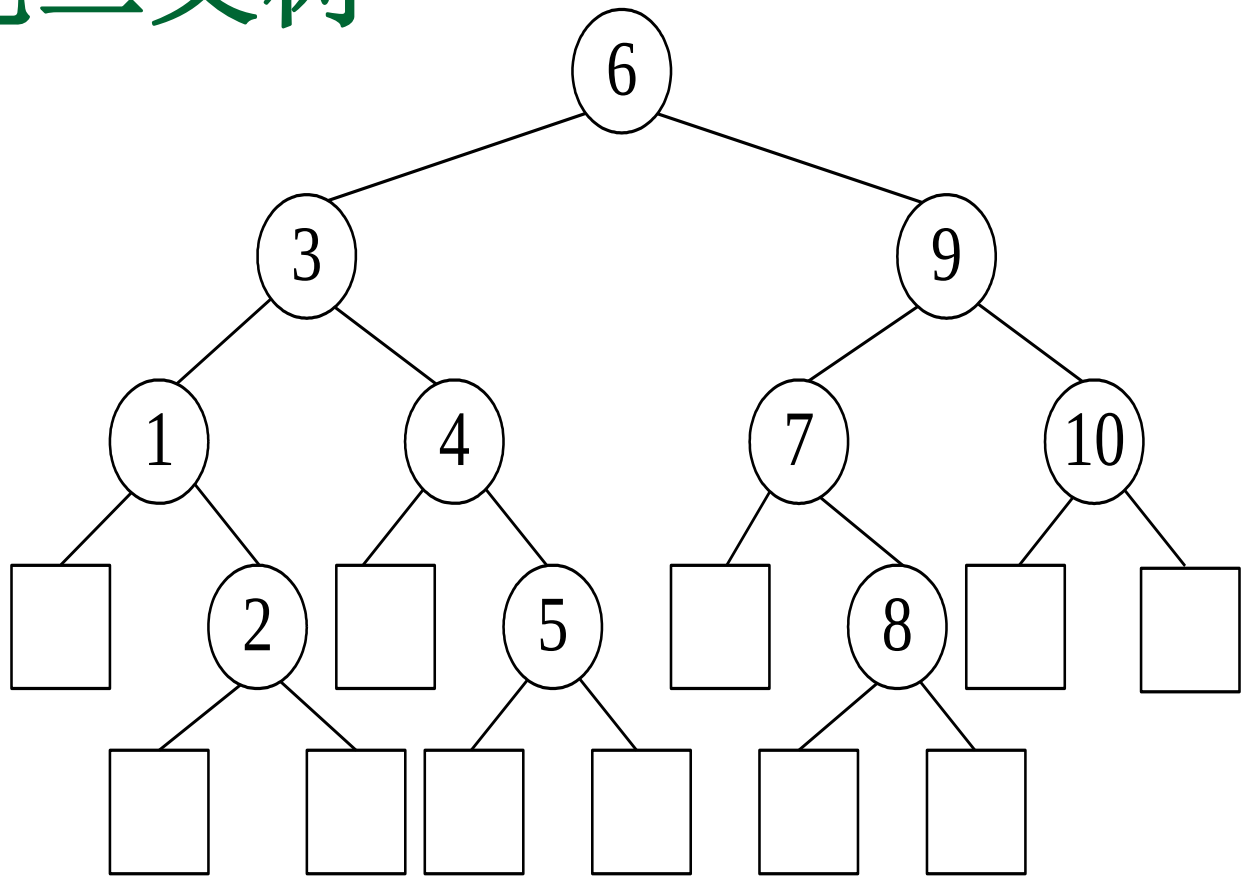
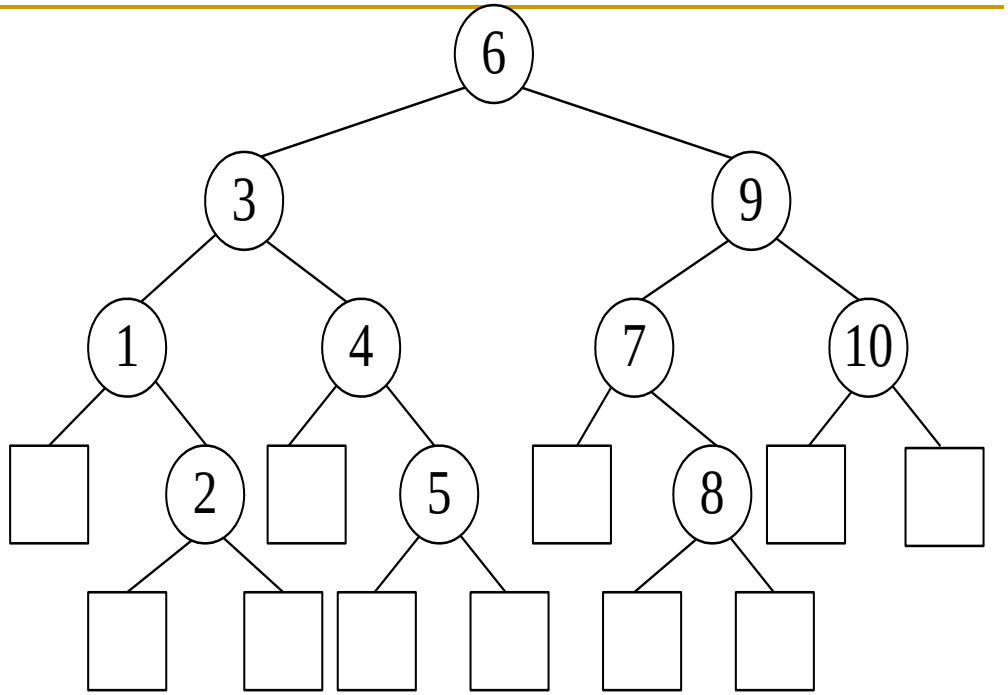


图5.3 扩充二叉树

扩充二叉树

- 从扩充的二叉树的根到每个外部结点的路径长度之和称为**外部路径长度(E)**。
- 扩充的二叉树里从根到每个内部结点的路径长度之和称为**内部路径长度(I)**。
- 外部路径长度E和内部路径长度I满足：E = I + 2n，其中n是内部结点个数。

扩充二叉树



- 例如，在图5.3这个有10个内部结点的扩充二叉树里
- E = 3 + 4 + 4 + 3 + 4 + 4 + 3 + 4 + 4 + 3 + 3= 39
- I = 0 + 1 + 2 + 3 + 2 + 3 + 1 + 2 + 3 + 2 = 19
- E和I两个量之间的关系为 E = I + 2n。

扩充二叉树

- 证明：对内部结点数目进行归纳。
- 当n = 1时，I = 0且E = 2，故E = I + 2n成立。
- 对于有n个内部结点的扩充二叉树此等式已成立，即En = In十2n，现在考虑有n + 1个内部结点的扩充二叉树
  - 删去一个分支结点，该分支结点与根结点的路径长度是K，使之成为有n个内部结点的扩充二叉树。由于删去了一个路径长度为K的内部结点，内部路径长度变为$I_n = I_{n+1}$-K；由于减少了两个路径长度为K + 1的外部结点，增加了一个路径长度为 K的外部结点，外部路径长度变为$E_n = E_{n+1} - 2 (K + 1) + K = E_{n+1} – K - 2$。由前两个等式，有$E_{n+1} = I_n + 2n + K + 2 = I_{n+1} + 2 (n + 1)$。等式E = I+2n得证。

二叉树的主要性质

- 性质1. 在二叉树中，第i层上最多有2^i个结点（$i \geqslant 0$）

  证明：利用数学归纳法

  当$i = 0$时，$2^0 = 1$，只有一个根结点，正确。

  现在假设对所有的j，$1 \leqslant j < i$，命题成立，即第j层上之多有2^j个结点。下面证明当就$j = i$时结论也成立。由归纳假设，第$i - 1$层上最多有2^{i-1}个结点。由于二叉树每个结点的度数最大为2，所以第i层上的最大结点数为第$i - 1$层上的最大结点数的2倍，即2^i个。

二叉树的主要性质

- **性质2. 深度为k的二叉树至多有 $2^{k+1}-1$个结点（k≥0）。其中深度(depth)定义为二叉树中层数最大的叶结点的层数。**

证明：由性质1可知，第i层的最大结点数为2^i，所以

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

二叉树的主要性质

- **性质3. 任何一棵二叉树，若其终端结点数为n_0，度为2的结点数为n_2，则$n_0=n_2+1$。**

证明：设n_1为二叉树中度为1 的结点数。该二叉树的结点总数n为度分别为0，1，2的结点数之和，即

$$n=n_0+n_1+n_2 \qquad \text{（公式5.1）}$$

除根结点外，其余结点都有一条边进入，设边数为e，有n = e + 1。由于这些边是由度为1或2的的结点发出的，所以又有$e=n_1+2n_2$，于是得

$$n=e+1=n_1+2n_2+1 \qquad \text{（公式5.2）}$$

由公式5.1和5.2得　　$n_0+n_1+n_2=n_1+2n_2+1$，

即　　$n_0=n_2+1$

二叉树的主要性质

- **性质4. 满二叉树定理：非空满二叉树树叶数目等于其分支结点数加1。**

证明：满二叉树定理由性质3可直接推出。

- **性质5. 满二叉树定理推论：一个非空二叉树的空子树数目等于其结点数加1。**

证明：设二叉树为T，将其所有空子树换为树叶，记新扩充满二叉树为T'。所有原来T的结点现在是T'的分支结点。根据满二叉树定理，新添加的树叶数目等于T结点个数加1。而每个新添加的树叶对于T的一个空子树。因此T中空子树数目等于T中的结点个数加1

二叉树的主要性质

- **性质6. 有n个结点（n>0）的完全二叉树的高度为[$\log_2(n+1)$]（深度为[$\log_2(n+1)$] - 1）。 其中二叉树的高度(height)定义为二叉树中层数最大的叶结点的层数加1。**

证明：假设高度为h，则根据性质2和完全二叉树的定义，

有 $2^{h-1}-1<n\le 2^{h}-1$ 或 $2^{h-1}<n+1\le 2^{h}$

不等式中各项取对数，于是得到$h-1<\log_2^{n+1}\le h$。因为h为整数，所以h＝[$\log_2(n+1)$]。

二叉树的主要性质

- 性质7. 对于具有n个结点的完全二叉树，结点按层次由左到右编号，则对任一结点i（$0 \le i \le n-1$）有

（1）如果i = 0，则结点i是二叉树的根结点；若i＞0，则其父结点编号是[(i - 1)/2]。

（2）当$2i+1 \le n-1$时，结点i的左子结点是2i + 1，否则结点i没有左子结点。

当$2i+2 \le n-1$时，结点i的右子结点是2i + 2，否则结点i没有右子结点。

（3）当i为偶数且$0 < i < n$时，结点i的左兄弟是结点i - 1，否则结点i没有左兄弟。

当i为奇数且$i+1 < n$时，结点i的右兄弟是结点i + 1，否则结点i没有右兄弟。

二叉树的主要性质

证明：这里证明(2)，(1) 和(3)即可由结论(2)推得。对于i = 0，由完全二叉树的定义，其左孩子的编号是1，如果1>n-1，即不存在编号为1的结点，此时结点i没有左孩子。其右孩子的编号只能是2，如果2>n-1，此时结点i没有右孩子。

对于i>0分两种情况讨论：

(1)设第j层的第一个结点编号为i(此时有$i = 2^j-1$)，则其左孩子必为第j + 1层的第一个结点，其编号为$2^{j+1} - 1 = 2i + 1$，如果2i + 1 > n - 1，那么i没有左孩子；其右孩子必为第j + 1层第二个结点，其编号为2i + 2。

(2)假设第j层的某个结点编号为i，若它有左孩子，那么它的左孩子必然是第j + 1层中的第$2[i - (2^j - 1)]$个，那么其左孩子的编号就是（$2^{j+1} - 1$）+ $2[i - (2^j - 1)] = 2i + 1$；如果结点i有右孩子，那么其右孩子的编号必是2i + 2。

5.2 二叉树的抽象数据类型

- **5.2.1** 抽象数据类型
- **5.2.2** 深度优先周游二叉树
- **5.2.3** 广度优先周游二叉树

5.2.1 抽象数据类型

- 一般情况下需要二叉树的各个结点存储所需要的信息，对二叉树的操作和运算也主要集中在访问二叉树的结点信息上。
  - 例如访问某个结点的左子结点、右子结点、父结点，或者访问结点存储的数据。
- 从二叉树的应用角度来看，有时还需要遍历二叉树的每个结点。

5.2.1 抽象数据类型

- 在二叉树的抽象数据类型中，定义了二叉树基本操作的集合，在具体应用中可以以此为基础进行扩充
- 为了强调抽象数据类型与存储无关，在此并没有具体规定该抽象数据类型的存储方式

5.2.1 抽象数据类型

【代码5.1】二叉树结点的抽象数据类型(ADT)

```
template <class T>
class BinaryTreeNode  {
friend class BinaryTree<T>;                    // 声明二叉树类为结点类的友元类，以便访问私有数据成员
private:
    T  info;                                   // 二叉树结点数据域
public:
    BinaryTreeNode();                          // 缺省构造函数
    BinaryTreeNode(const T& ele);
                                   // 给定数据的构造函数
    BinaryTreeNode(const T& ele,BinaryTreeNode<T> *l,
    BinaryTreeNode<T> *r);         // 子树构造结点
```

5.2.1 抽象数据类型

```
T  value() const;    // 返回当前结点的数据
BinaryTreeNode<T>* leftchild() const;
                        // 返回当前结点左子树
BinaryTreeNode<T>* rightchild() const;
            // 返回当前结点右子树
void  setLeftchild(BinaryTreeNode<T>*) ;
            // 设置当前结点的左子树
void  setRightchild(BinaryTreeNode<T>*) ;
            // 设置当前结点的右子树
void  setValue(const T& val);
                        // 设置当前结点的数据域
bool  isLeaf() const;            // 判断当前结点是否为叶结点
BinaryTreeNode<T>& operator = (const BinaryTreeNode<T>& Node);      //
    重载赋值操作符
};
```

5.2.1 抽象数据类型

【代码5.2】二叉树的抽象数据类型

```
template <class T>
class BinaryTree  {
private:
     BinaryTreeNode<T>* root;
            // 二叉树根结点
public:
   BinaryTree() {root = NULL;};
                        // 构造函数
   ~BinaryTree() {DeleteBinaryTree(root);};
            // 析构函数
   bool isEmpty() const;
                                  // 判定二叉树是否为空树
     BinaryTreeNode<T>* Root(){return root;};
                        // 返回二叉树根结点
```

5.2.1 抽象数据类型

```
BinaryTreeNode<T>* Parent(BinaryTreeNode<T> *current);                    //
    返回当前结点的父结点
    BinaryTreeNode<T>* LeftSibling(BinaryTreeNode<T> *current);           //
    返回当前结点的左兄弟
    BinaryTreeNode<T>* RightSibling(BinaryTreeNode<T> *current);          // 返
    回当前结点的右兄弟
void CreateTree(const T& info, BinaryTree<T>& leftTree, BinaryTree<T>& rightTree);
    // 构造新树
    void PreOrder(BinaryTreeNode<T> *root); // 前序周游二叉树或其子树
    void InOrder(BinaryTreeNode<T> *root);// 中序周游二叉树或其子树
    void PostOrder(BinaryTreeNode<T> *root);// 后序周游二叉树或其子树
    void LevelOrder(BinaryTreeNode<T> *root); // 按层次周游二叉树或其子树
    void DeleteBinaryTree(BinaryTreeNode<T> *root); // 删除二叉树或其子树
};
```

5.2.1 抽象数据类型

- 所谓二叉树的**周游(**或称**遍历，traversal)**是指按照一定顺序依次访问树中所有结点，并使得每一结点仅被访问一次。这里所说的“访问”是指操作，可以理解成对二叉树结点数据成员的处理，比如输出、修改结点的信息等。

- 周游一棵二叉树的过程实际上就是把二叉树的结点放入一个线性序列的过程，也就是对二叉树进行线性化。

5.2.2 深度优先周游二叉树

基于二叉树的递归定义，这三种深度优先周游的递归定义是：

(1) 前序法(tLR次序，preorder traversal)。其递归定义是

访问根结点；

按前序周游左子树；

按前序周游右子树。

(2) 中序法(LtR次序，inorder traversal)。其递归定义是

按中序周游左子树；

访问根结点；

按中序周游右子树。

(3) 后序法(LRt次序，postorder traversal)。其递归定义是

按后序周游左子树；

按后序周游右子树；

访问根结点。

5.2.2 深度优先周游二叉树

深度周游如下二叉树

- 前序序列是：**A B D E G C F H I**
- 中序序列是：**D B G E A C H F I**
- 后序序列是：**D G E B H I F C A**

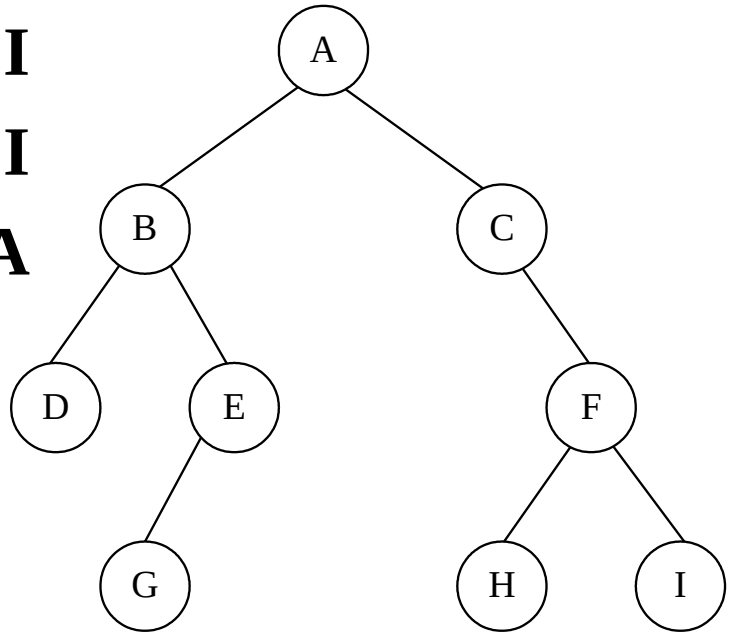


图5.5 二叉树示例

5.2.2 深度优先周游二叉树

- 二叉树的周游算法与表达式的“前缀”和“后缀”表示法之间有着密切的联系。例如图5.6是表达式A+B×(C+D)的二叉树表示。按照前序方式周游，运算符出现在前面，参与运算的对象紧跟在其后，这样就形成了前缀表达式（波兰式）：

  + A × B + C D

- 按照中序方式周游，得到的结果是去掉括号的中缀表达式：

  A + B × C + D

- 下面是后序方式周游的结果，得到的是后缀表达式（逆波兰式）：

  A B C D + × +

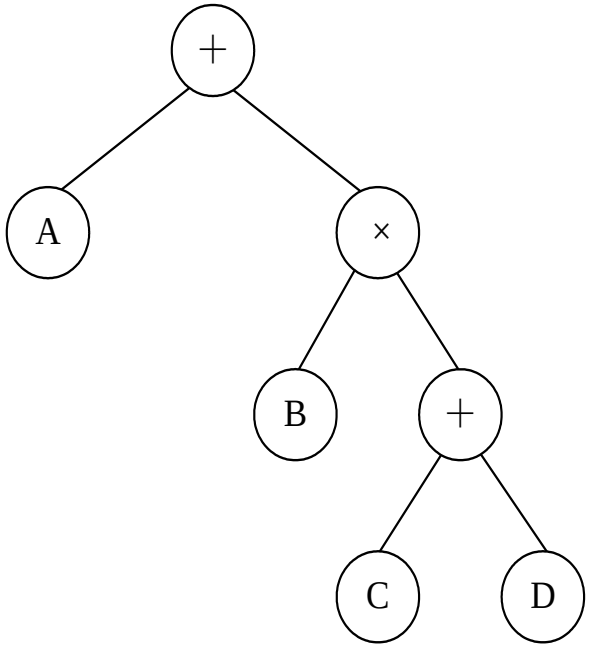


图5.6 表达式树

5.2.2 深度优先周游二叉树

【算法5.3】深度优先周游二叉树或其子树

```
template<class T>
void BinaryTree<T>::PreOrder (BinaryTreeNode<T> *root) {      // 前序周游二叉树或其子树
    if (root != NULL) {
        Visit(root->value());                    // 访问当前结点
        PreOrder(root->leftchild()); // 前序周游左子树
        PreOrder(root->rightchild());            // 前序周游右子树
}
}
template<class T>
void BinaryTree<T>:: InOrder (BinaryTreeNode<T> *root) {      // 中序周游二叉树或其子树
    if (root != NULL) {
        InOrder (root->leftchild());  // 中序周游左子树
```

5.2.2 深度优先周游二叉树

```
    Visit(root->value());
        // 访问当前结点
        InOrder(root->rightchild()); // 中序周游右子树
}
}
template<class T>
void BinaryTree<T>:: PostOrder (BinaryTreeNode<T> *root) {    // 后序周游二叉树或其子树
    if (root != NULL) {
        PostOrder(root->leftchild());          // 后序周游左子树
        PostOrder (root->rightchild());        // 后序周游右子树
        Visit(root->value());
            // 访问当前结点
}
}
```

5.2.2 深度优先周游二叉树

深度优先周游二叉树的非递归算法

- 递归算法虽然简洁，但是有时程序设计不允许用递归，这时就存在如何把递归程序转化成等价的非递归算法的问题
- 解决这个问题的关键是设置一个栈结构。仿照递归算法执行过程中编译栈的工作原理，可以写出非递归周游二叉树的算法

5.2.2 深度优先周游二叉树

非递归前序周游算法的主要思想：

- 每遇到一个结点，先访问该结点，并把该结点的非空右子结点推入栈中，然后周游其左子树
- 左子树周游不下去时，从栈顶托出待访问的结点，继续周游。算法执行过程中，只有非空结点入栈
- 为了算法的简洁，最开始推入一个空指针作为监视哨；当这个空指针被弹出来时，周游就结束了

5.2.2 深度优先周游二叉树

【算法5.4】非递归前序周游二叉树或其子树

```
template<class T>
void BinaryTree<T>::PreOrderWithoutRecursion(BinaryTreeNode<T>
    *root)  {
    using std::stack;
                    // 使用STL中的栈
    stack<BinaryTreeNode<T>* > aStack;
    BinaryTreeNode<T> *pointer = root;
    aStack.push(NULL);
        // 栈底监视哨
    while (pointer) {
        // 或者!aStack.empty()
        Visit(pointer->value());                    // 访问当前结点
```

5.2.2 深度优先周游二叉树

```
    if (pointer->rightchild() != NULL)                    // 非空右孩子入栈
            aStack.push(pointer->rightchild());
            if (pointer->leftchild() != NULL)
            pointer = pointer->leftchild();                        // 左路下降
            else {
    // 左子树访问完毕，转向访问右子树
                        pointer=aStack.top();
    // 获得栈顶元素
                        aStack.pop();
    // 栈顶元素退栈
            }
    }
}
```

5.2.2 深度优先周游二叉树

非递归中序周游算法的主要思想是：

- 每遇到一个结点就把它推入栈，然后去周游其左子树
- 周游完左子树后，从栈顶托出这个结点并访问之
- 然后按照其右链接指示的地址再去周游该结点的右子树

5.2.2 深度优先周游二叉树

【算法5.5】非递归中序周游二叉树或其子树

```
template<class T>
void BinaryTree<T>::InOrderWithoutRecursion(BinaryTreeNode<T> *root)
    {
    using std::stack;
                    // 使用STL中的栈
    stack<BinaryTreeNode<T>* > aStack;
    BinaryTreeNode<T> *pointer = root;
    while (!aStack.empty() || pointer) {
       if (pointer) {
```

5.2.2 深度优先周游二叉树

```
                aStack.push(pointer);
            // 当前指针入栈
                pointer = pointer->leftchild();          // 左路下降
}
    else {          // 左子树访问完毕，转向访问右子树
                pointer = aStack.top();                  // 获得栈顶元素
                aStack.pop();                            // 栈顶元素退栈
                Visit(pointer->value());                 // 访问当前结点
                pointer = pointer->rightchild();         // 指针指向右孩子
    }
  }
}
```

5.2.2 深度优先周游二叉树

在非递归的后序周游算法的主要思想：

- 每遇到一个结点，先把它推入栈中，去周游它的左子树
- 周游完它的左子树后，应继续周游该结点的右子树
- 游完右子树之后，才从栈顶托出该结点并访问它

5.2.2 深度优先周游二叉树

- 解决方案：
  - 由于访问某个结点前需要知道是否已经访问该结点的右子树，因此需要给栈中的每个元素加一个标志位**tag**
  - 标志位用枚举类型**Tags**表示，**tag**为**Left**表示已进入该结点的左子树；**tag**为**Right**表示已进入该结点的右子树

5.2.2 深度优先周游二叉树

【算法5.6】非递归后序周游二叉树或其子树

```
enum Tags{Left,Right};
                                        // 定义枚举类型标志位
template <class T>
class StackElement  {                    // 栈元素的定义
public:
    BinaryTreeNode<T>* pointer;      // 指向二叉树结点的指针
    Tags tag;                                        // 标志位
};
template<class T>
void BinaryTree<T>::PostOrderWithoutRecursion(BinaryTreeNode<T>*
    root)  {
    using std::stack;                          // 使用STL的栈
    StackElement<T> element;
```

5.2.2 深度优先周游二叉树

```
stack<StackElement<T > > aStack;
BinaryTreeNode<T>* pointer;
if (root == NULL)                          // 如果是空树则返回
        return;
else pointer = root;
while (!aStack.empty() || pointer) {
        while (pointer != NULL) {          // 如果当前指针非空则压栈并下降到最左子结点
                element.pointer = pointer;
                element.tag = Left;        // 置标志位为Left, 表示进入左子树
                aStack.push(element);
                pointer = pointer->leftchild();
        }
        element = aStack.top();                    // 获得栈顶元素
```

5.2.2 深度优先周游二叉树

```
            aStack.pop();                                   // 栈顶元素退栈
            pointer = element.pointer;
            if (element.tag == Left) {                      // 如果从左子树回来
                        element.tag = Right;     // 置标志位为Right, 表示进入
    右子树
                        aStack.push(element);
                        pointer = pointer->rightchild();
            }
            else {                                          // 如果从右子树回来
                        Visit(pointer->value());            // 访问当前结点
                        pointer = NULL;      // 置point指针为空，以继续弹栈
            }
    }
}
```

5.2.2 深度优先周游二叉树

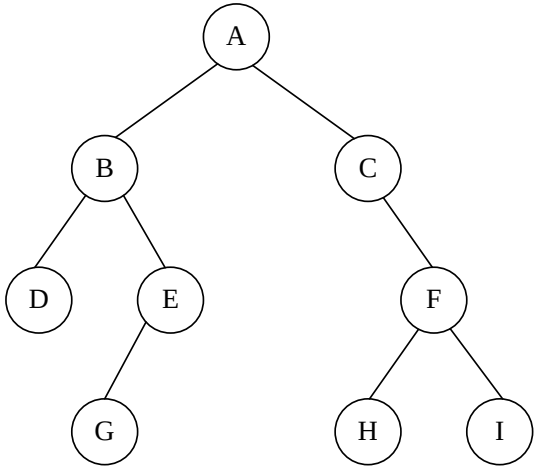
- 对于有n个结点的二叉树，周游完树的每一个元素都需要O(n)时间
- 只要对每个结点的处理（函数Visit的执行）时间是一个常数，那么周游二叉树就可以在线性时间内完成
- 所需要的辅助空间为周游过程中栈的最大容量，即树的高度
- 最坏情况下具有n个结点的二叉树高度为n，则所需要的空间复杂度为O(n)

5.2.3 广度优先周游二叉树

- 对二叉树进行广度优先(层次)周游的过程是：
  - 首先访问第0层，也就是根结点所在的层
  - 然后从左到右依次访问第一层两个结点
  - 以此类推，当第i层的所有结点访问完之后，再从左至右依次访问第i+1层的各个结点

- 对于右图中的二叉树进行广度优先周游的结果为：

A B C D E F G H I



5.2.3 广度优先周游二叉树

【算法5.7】广度周游二叉树及其子树

```
template<class T>
void BinaryTree<T>::LevelOrder(BinaryTreeNode<T> *root)          {
    using std::queue;                                  // 使用STL的队列
    queue<BinaryTreeNode<T>*> aQueue;
    BinaryTreeNode<T> *pointer = root;
    if (pointer)
        aQueue.push(pointer);                  // 根结点入队列
    while (!aQueue.empty()) {                  // 队列非空
        pointer = aQueue.front();       // 获得队列首结点
        aQueue.pop();                              // 当前结点出队列
    Visit(pointer->value());                       // 访问当前结点
        if (pointer->leftchild() != NULL)
            aQueue.push(pointer->leftchild());  // 左子树进队列
        if (pointer->rightchild()!= NULL)
            aQueue.push(pointer->rightchild()); // 右子树进队列
    }
}
```

5.3 二叉树的存储结构

- 5.3.1　二叉树的链式存储结构
- 5.3.2　完全二叉树的顺序存储结构

5.3.1 二叉树的链式存储结构

所谓链式存储方式，是指二叉树的各结点随机的存储在内存空间中，结点之间的关系用指针表示。

- 二叉树的链表的结点包含三个域：数据域和左、右指针域。用info域存储结点的数据元素，另外再设置两个指针域left和right，分别指向结点的左子结点和右子结点
- 当结点的某个子结点为空时，相应的指针为空指针。其结点结构如图5.7(a)所示。有时候为了便于查找二叉树中某个结点的父结点，还可以在结点结构中加入一个指向其父结点的指针域，如图5.7(b)所示

(a) 有两个指针域的结点结构　　(b) 有三个指针域的结点结构

图5.7 二叉树结点的存储结构

5.3.1 二叉树的链式存储结构

利用这两种结点结构所得到的二叉树的存储结构分别称为二叉链表（也称“left-right存储法”）和三叉链表

- 在使用二叉链表存储的二叉树中，如果找某个结点的父结点，那么需要从根结点出发依次巡查
- 在三叉链表表示的二叉树中只需要顺着parent指针就能直接找到该结点的父结点

5.3.1 二叉树的链式存储结构

图5.5 二叉树示例

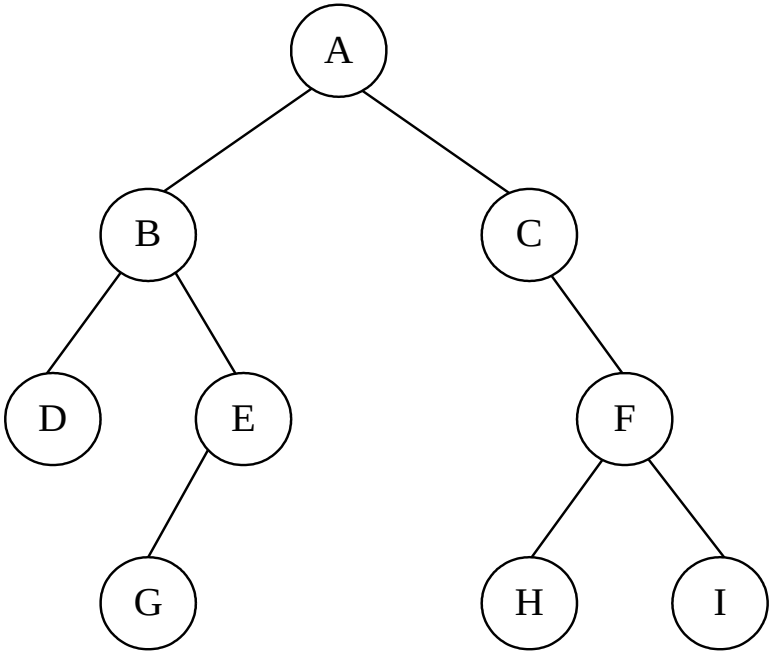
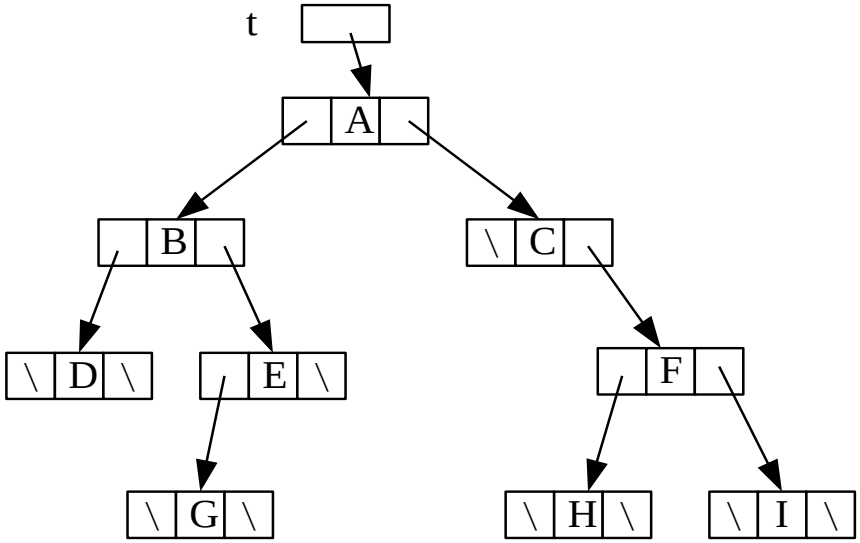


图5.8 图5.5中二叉树的二叉链表存储结构



容易看出，在含有n个结点的二叉链表中有n+1个空链域。可以利用这些空链域存储其它的有用信息，形成其它的链式存储结构。

5.3.1 二叉树的链式存储结构

【算法5.8】二叉树部分成员函数的实现

```
// 用二叉链表实现二叉树需要在代码5.1的BinaryTreeNode类中增加两个私
    有数据成员
private:
BinaryTreeNode<T> *left;                // 指向左子树的指针
BinaryTreeNode<T> *right;               // 指向右子树的指针
template<class T>
bool BinaryTree<T>:: isEmpty() const  {     // 判定二叉树是否为空树
    return ( root != NULL ? false : true);
}
template<class T>
BinaryTreeNode<T>* BinaryTree<T>::Parent(BinaryTreeNode<T>
    *current)  {
```

5.3.1 二叉树的链式存储结构

```
using std::stack;                          // 使用STL中的栈
stack<BinaryTreeNode<T>* > aStack;
BinaryTreeNode<T> *pointer = root;
if (root != NULL && current != NULL) {
        while (!aStack.empty() || pointer) {
                if (pointer != NULL) {
                        if (current == pointer->leftchild() || current ==
pointer->rightchild())
                                return pointer;                          //
如果pointer的孩子是current则返回parent
                        aStack.push(pointer);       // 当前指针入栈
                        pointer = pointer->leftchild();        // 当前指
针指向左孩子
                }
else {                    // 左子树访问完毕，访问右子树
                        pointer = aStack.top();        // 获得栈顶元素
```

5.3.1 二叉树的链式存储结构

```
                    aStack.pop();                    // 栈顶元素退栈
                    pointer = pointer->rightchild();     // 当前指针指向右孩子
                    }
            }
    }
}
// 创建一棵新树，参数info为根结点元素，leftTree和rightTree是不同的两棵树
template<class T>
void BinaryTree<T>:: CreateTree (const T& info, BinaryTree<T>& leftTree,
    BinaryTree<T>& rightTree) {
    root = new BinaryTreeNode<T>(info, leftTree.root, rightTree.root);
            // 创建新树
    leftTree.root = rightTree.root = NULL;              // 原来两棵子树根结点置为空，避免非法访问
}
```

5.3.1 二叉树的链式存储结构

```
template<class T>
void BinaryTree<T>:: DeleteBinaryTree(BinaryTreeNode<T> *root)
    {      // 后序周游删除二叉树
    if (root != NULL) {
          DeleteBinaryTree(root->left);     // 递归删除左子树
          DeleteBinaryTree(root->right); // 递归删除右子树
          delete root;
          // 删除根结点
}
}
```

5.3.2 完全二叉树的顺序存储结构

二叉树的顺序存储是指按照一定次序，用一组地址连续的存储单元存储二叉树上的各个结点元素。

完全二叉树的顺序表示法：

对于一棵具有n个结点的完全二叉树，可以从根结点起自上而下，从左至右地把所有的结点编号，得到一个足以反映整个二叉树结构的线性序列。线性序列里存储的结点就是按照层次周游二叉树得到的排列。

复习：二叉树的主要性质7

- 性质7. 对于具有n个结点的完全二叉树，结点按层次由左到右编号，则对任一结点i（$0 \le i \le n-1$）有

（1）如果i = 0，则结点i是二叉树的根结点；若i＞0，则其父结点编号是[(i - 1)/2]。

（2）当$2i+1 \le n-1$时，结点i的左子结点是2i + 1，否则结点i没有左子结点。

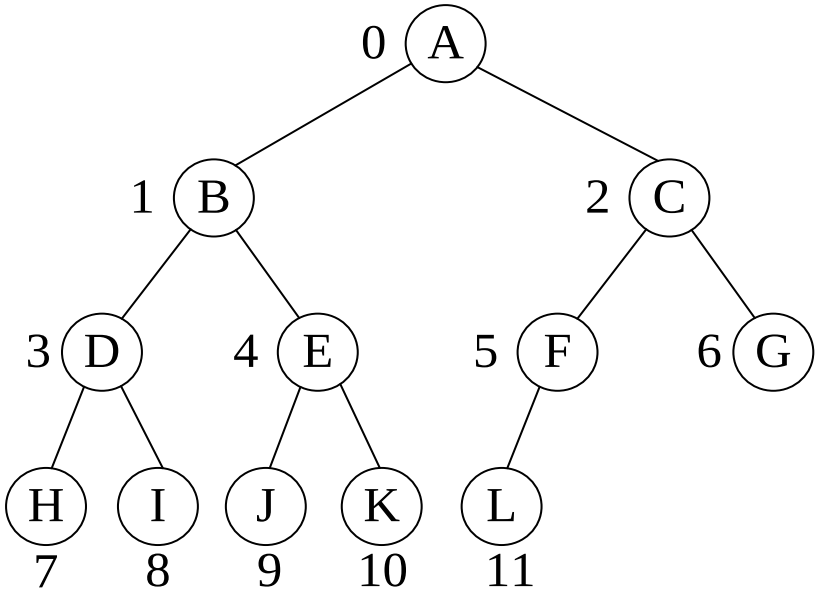
当$2i+2 \le n-1$时，结点i的右子结点是2i + 2，否则结点i没有右子结点。

（3）当i为偶数且$0 < i < n$时，结点i的左兄弟是结点i - 1，否则结点i没有左兄弟。

当i为奇数且$i+1 < n$时，结点i的右兄弟是结点i + 1，否则结点i没有右兄弟。

5.3.2 完全二叉树的顺序存储结构

图5.9 完全二叉树的节点编号



对于任何一个二叉树结点，如果它存储在数组的第i个位置，那么它的左右子结点分别存放在2i + 1和2i + 2的位置，它的父结点的下标为[(i-1) / 2]。

图5.10 完全二叉树的顺序存储结构

如图所示，按层次顺序将一棵n个结点的完全二叉树中的所有结点从0到n-1编号。可以将这棵二叉树编号为i的结点元素存储在一维数组下标为i的分量中。例如左图所示的二叉树的顺序存储结构如右图所示。

5.3.2 完全二叉树的顺序存储结构

- 对于完全二叉树这种特殊情况，结点的层次序列就足以反映整个二叉树的结构。顺序方式是完全二叉树的最简单又最节省空间的存储方式。
- 在不同的二叉树的存储结构中，不仅空间开销有差异，实现二叉树的操作方法也不同。由此在具体应用中采取什么存储结构，除了根据二叉树的形态之外，还应该考虑时间、空间复杂度和算法的简洁性，根据需要而确定。

5.4 二叉搜索树

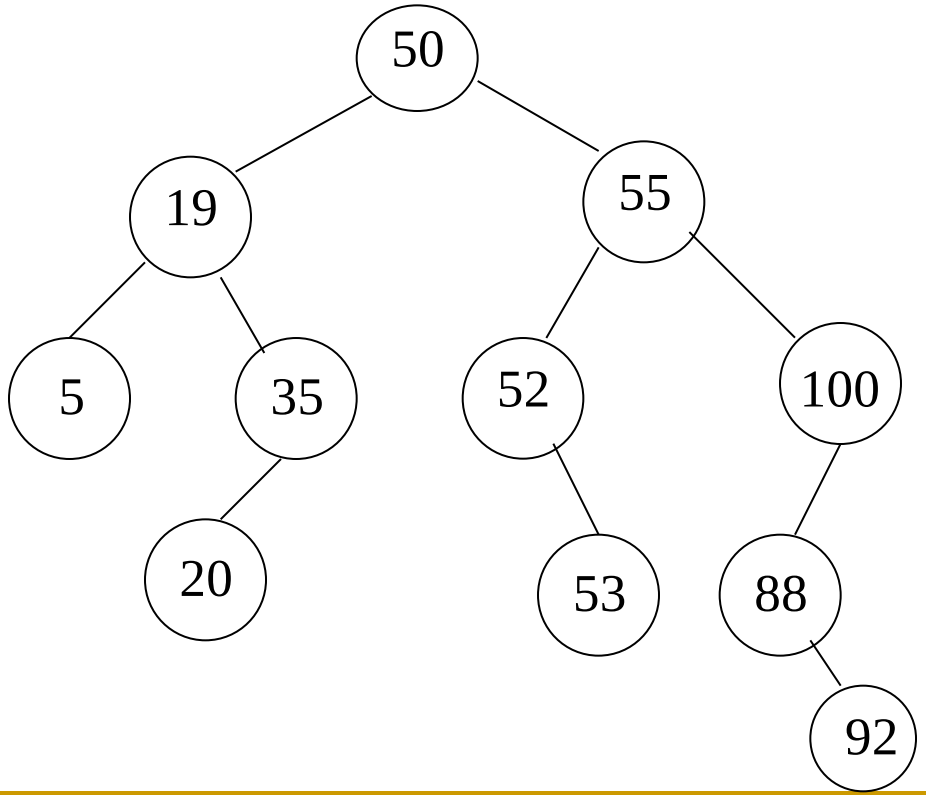
- 二叉搜索树(BST)是一类满足以下属性的特殊二叉树：
  - 二叉搜索树中的每个非空结点表示一个记录
  - 某结点左子树不空，则左子树上所有结点的值均小于该结点的关键码值
  - 若其右子树不为空，则右子树上所有结点的值均大于该结点的关键码值
  - 树中各结点的关键码必须是唯一的
  - 二叉搜索树可以是一棵空树，任何结点的左右子树都是二叉搜索树,按照中序周游整个二叉树得到一个由小到大有序排列

5.4 二叉搜索树

对于关键码集合

K = {50，19，35，55，20，5，100，52，88，53，92}

二叉搜索树的生成过程如图所示：



5.4 二叉搜索树

- 二叉搜索树的高效率在于继续检索时只需要查找两棵子树之一：
  - 将给定值key与根结点的关键码比较，如果key小于根结点的值，则只需要检索左子树
  - 如果key大于根结点的值，就只检索右子树
  - 这个过程一直持续到key被匹配成功或者遇到叶结点为止
  - 如果遇到叶结点仍没有发现key，那么key就不在这棵二叉搜索树中

5.4 二叉搜索树

二叉搜索树的插入算法：

- 需要运用检索方法，查找待插入关键码是否在树中
- 如果存在则不允许插入重复关键码
- 如果直到找到空结点还没有发现重复关键码，那么就把新结点插入到待插入方向作为新的叶结点
- 对于给定的关键码集合，可以从一个空的二叉搜索树开始，按照检索路径搜索这棵二叉树，将关键码一个个插入到相应的叶结点位置，从而动态生成二叉搜索树

5.4 二叉搜索树

- 二叉搜索树插入操作：
  - 将待插入结点的关键码与根结点的关键码相比较，若待插入的关键码小于根结点的关键码，则进入左子树，否则进入右子树。
  - 按照同样的方式沿检索路径直到叶结点，确定插入位置，把待插入结点作为一个新叶结点插入到二叉搜索树中。

5.4 二叉搜索树

【算法5.9】二叉搜索树的结点插入算法

```
template<class T>
void BinarySearchTree<T>::InsertNode(BinaryTreeNode<T> *root ,
    BinaryTreeNode<T> *newpointer)  {
    BinaryTreeNode<T> *pointer = NULL;
    if (root == NULL) {                  // 如果是空树
        Initialize(newpointer);   // 则用指针newpointer作为树根
        return;
    }
    else pointer = root;
    while (pointer != NULL) {
        if (newpointer->value() == pointer->value())     // 如果存在相等的
    元素则不用插入
                return ;
        else if (newpointer->value() < pointer->value()) {  // 如果待插入结
    点小于pointer的关键码值
if (pointer->leftchild() == NULL) {        // 如果pointer没有左孩子
```

```
                        pointer->left = newpointer;
            // newpointer作为pointer的左子树
                        return;
                    }

else        pointer = pointer->leftchild();    // 向左下降
            }
            else {      // 若待插入结点大于pointer的关键码值
                    if (pointer->rightchild() == NULL) {
                            // 如果pointer没有右孩子
                        pointer->right = newpointer;
                        // newpointer作为pointer的右子树
                        return;
                    }
                    else        pointer = pointer->rightchild();
                            // 向右下降
            }
    }
}
```

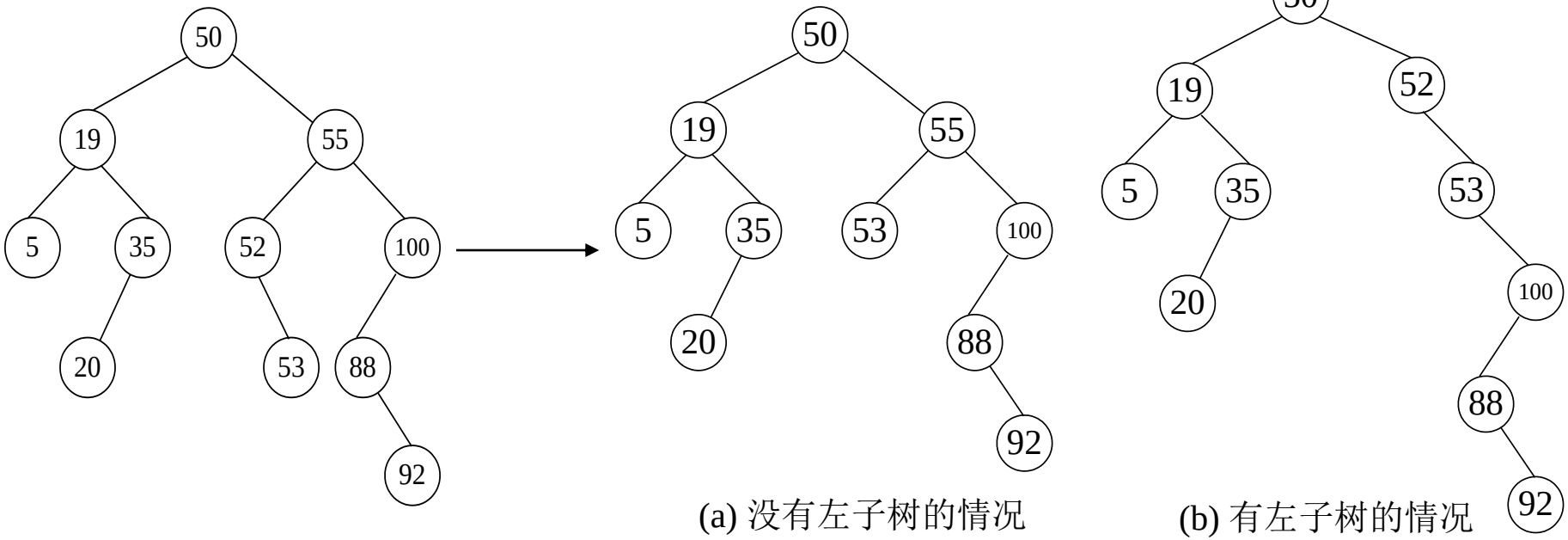
5.4 二叉搜索树

- 对二叉搜索树的检索，每一次只需与结点的一棵子树相比较
- 在执行插入操作时，也不必像在有序线性表中插入元素那样要移动大量的数据，而只需改动某个结点的空指针插入一个叶结点即可
- 与查找结点的操作一样，插入一个新结点操作的时间复杂度是根到插入位置的路径长度，因此在树形比较平衡时二叉搜索树的效率相当高

二叉搜索树的删除

- 设pointer，temppointer是指针变量，其中pointer表示要删除的结点。首先找到待删除的结点pointer，删除该结点的过程如下：
  - 若结点pointer没有左子树，则用pointer右子树的根代替被删除的结点pointer；
  - 若结点pointer有左子树，则在左子树里找到按中序周游的最后一个结点temppointer，把temppointer的右指针置成指向pointer的右子树的根，然后用结点pointer左子树的根代替被删除的结点pointer。

二叉搜索树的删除



(a) 没有左子树的情况　　(b) 有左子树的情况

图5.11 二叉搜索树

图5.12 二叉搜索树的删除示例

二叉搜索树的删除

- 改进的二叉搜索树结点删除算法的思想为：
  - 若结点pointer没有左子树，则用pointer右子树的根代替被删除的结点pointer
  - 若结点pointer有左子树，则在左子树里找到按中序周游的最后一个结点temppointer（即左子树中的最大结点）并将其从二叉搜索树里删除
  - 由于temppointer没有右子树，删除该结点只需用temppointer的左子树代替temppointer，然后用temppointer结点代替待删除的结点pointer

二叉搜索树的删除

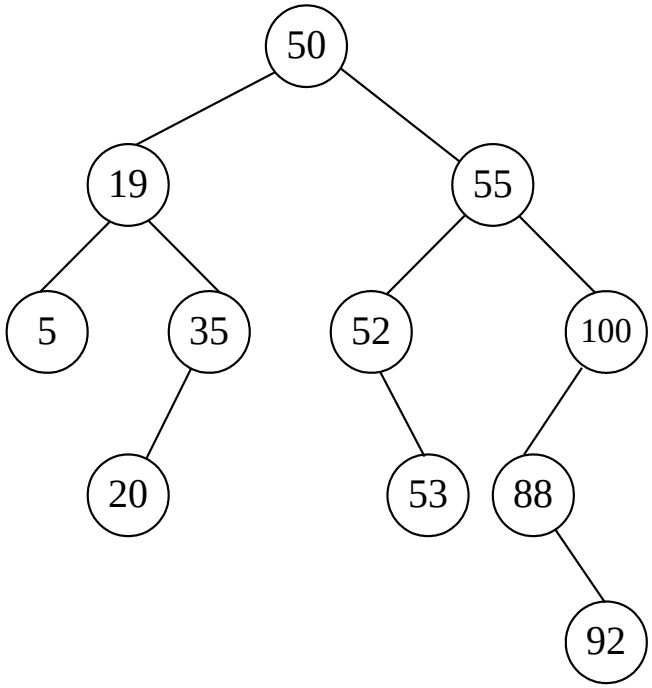


图5.11 二叉搜索树

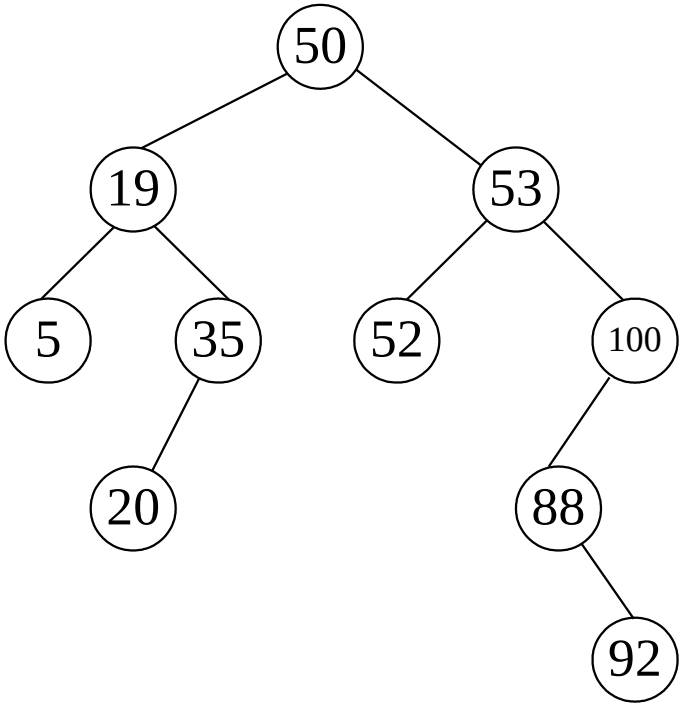


图5.13 改进的二叉搜索树的删除

二叉搜索树的删除

【算法5.10】改进的二叉搜索树的结点删除

```
template <class T>
void BinarySearchTree<T>::DeleteNodeEx(BinaryTreeNode<T> *pointer) {
    if ( pointer == NULL )                          // 若待删除结点不存在则返回
        return;
    BinaryTreeNode<T>  *temppointer;                // 用于保存替换结点
    BinaryTreeNode<T>  *tempparent = NULL;// 用于保存替换结点的父结点
    BinaryTreeNode<T>  *parent = Parent(pointer );
                                                    // 用于保存待删除结点的父结点
    if ( pointer->leftchild() == NULL )       // 如果待删除结点的左子树为空
    temppointer = pointer->rightchild();  // 替换结点赋值为其右子树的根
```

二叉搜索树的删除

```
else {// 待删除结点左子树不空，在左子树中寻找最大结点作为替换结点
        temppointer = pointer->leftchild();
        while (temppointer->rightchild() != NULL ) {/
            / 寻找左子树中的最大结点
                tempparent = temppointer;
                temppointer = temppointer->rightchild();     // 向右下降
        }
        if (tempparent == NULL)
      // 如果替换结点就是被删结点的左子结点
                pointer->left = temppointer->leftchild();
            // 替换结点左子树挂接到被删结点的左子树
        else tempparent->right = temppointer->leftchild();
        // 替换结点的左子树作为其父结点右子树
        temppointer->left = pointer->leftchild();
        // 继承pointer的左子树
        temppointer->right = pointer->rightchild();
         // 继承pointer的右子树
    }
```

二叉搜索树的删除

```
    // 下面用替换结点代替待删除结点
    if (parent == NULL)
            root = temppointer;
    else if ( parent->leftchild() == pointer )
       parent->left = temppointer;
    else parent->right = temppointer;
    delete pointer;                  // 删除该结点
    pointer = NULL;
    return;
}
```

5.5 堆与优先队列

- 5.5.1 堆的定义及其实现
- 5.5.2 优先队列

5.5.1 堆的定义及其实现

- **最小值堆**：最小值堆是一个关键码序列{K_0，K_1，…K_{n-1}}，它具有如下**特性**：
  - $K_i \leq K_{2i+1}$　(i=0，1，…，n/2-1)
  - $K_i \leq K_{2i+2}$

5.5.1 堆的定义及其实现

堆的性质

- 从逻辑的角度来讲，堆是一种树形结构，而且是一种特殊的完全二叉树。此完全二叉树的每个结点对应于序列中的一个关键码，根结点对应于关键码K_0，按层次从左至右依次类推。
- 说其特殊，主要是因为堆序只是局部有序，即最小堆对应的完全二叉树中所有内部结点的值均不大于其左右孩子的关键码值，而一个结点与其兄弟之间没有必然的联系。
- 最小堆不像二叉搜索树那样实现了关键码的完全排序。相比较而言，只有当结点之间是父子关系的时候，才可以确定这两个结点关键码的大小关系。

5.5.1 堆的定义及其实现

关键码序列K = {12，14，15，19，20，17，18，24，22，26}所对应的最小堆形成的完全二叉树形式为图5.14所示：

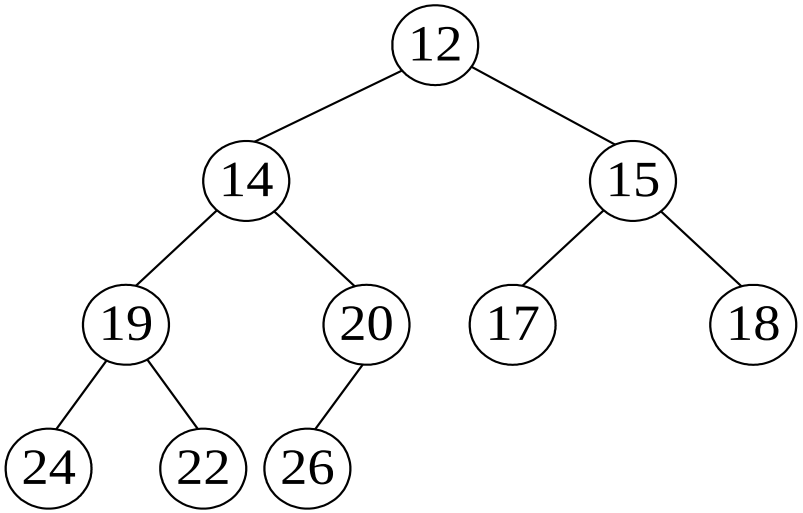
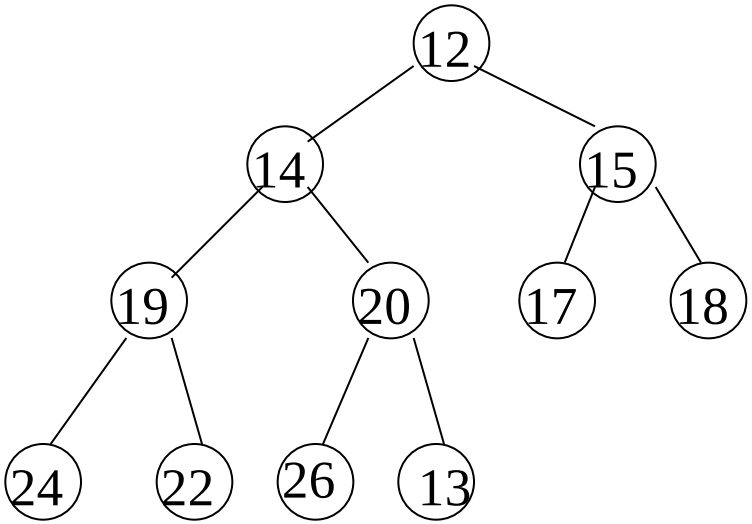


图5.14 最小堆对应的完全二叉树

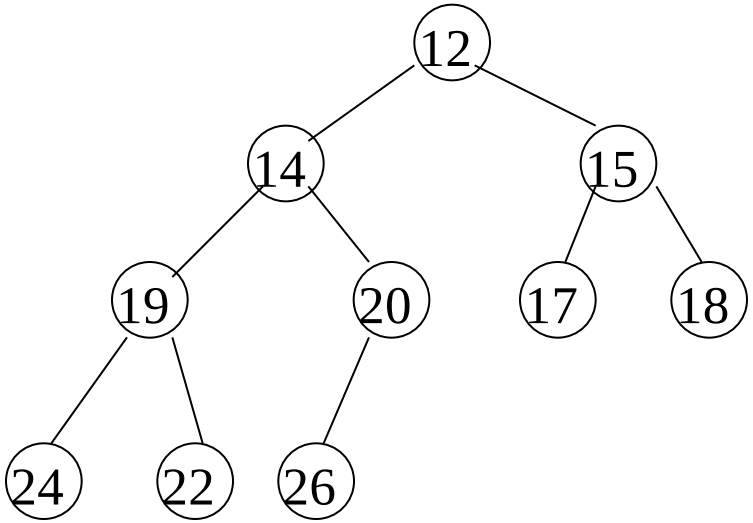
5.5.1 堆的定义及其实现

图5.15 在最小堆5.14中插入元素13



5.5.1 堆的定义及其实现

图5.16 在最小堆5.14中删除元素14



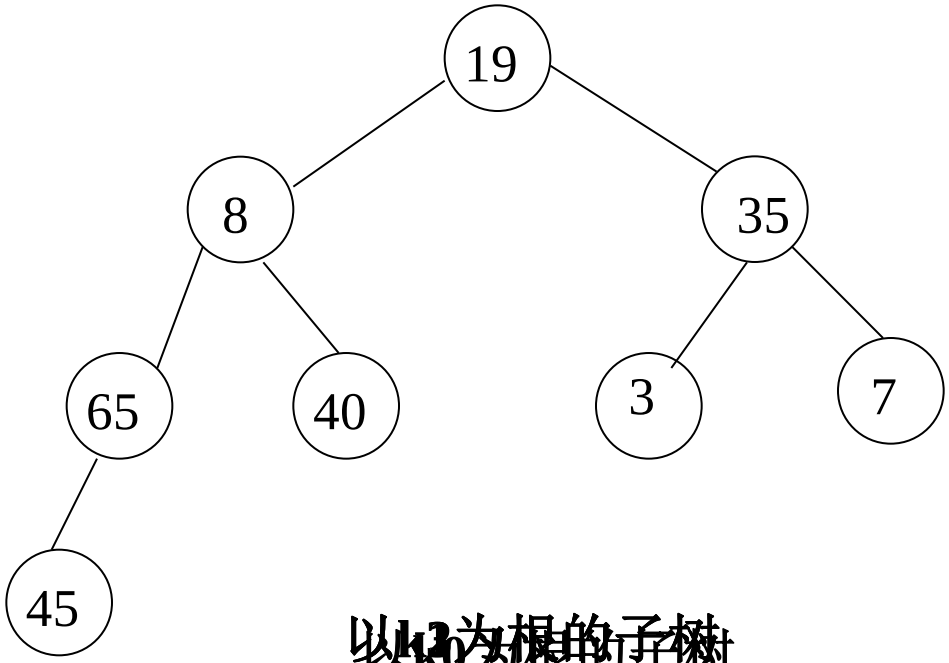
5.5.1 堆的定义及其实现

建堆过程：

- 首先将所有关键码放到一维数组中，这时形成的完全二叉树并不具备最小堆的特性，但是仅包含叶子结点的子树已经是堆
- 即在有n个结点的完全二叉树中，当i >[n/2]-1时，以关键码K_i为根的子树已经是堆。
- 这时从含有内部结点数最少的子树（这种子树在完全二叉树的倒数第二层，此时i = [n/2]-1 ）开始，从右至左依次调整
- 对这一层调整完成之后，继续对上一层进行同样的工作，直到整个过程到达树根时，整棵完全二叉树就成为一个堆了

5.5.1 堆的定义及其实现

对于关键码集合K = {19，8，35，65，40，3，7，45}，用筛选法建堆的过程。其中n = 8，n/2-1=3，所以从K3 = 65开始调整。



以k3为根的子树

以k0为根的子树

35>3 调整

65>45 调整

8<40 无需调整

19>3 调整

图5.17

建堆过程示例

堆的类定义和筛选法(1)

[代码5.11] 堆的类定义和筛选法

```
template <class T>
class MinHeap  {
        // 最小堆类定义
private:
    T *heapArray;
        // 存放堆数据的数组
    int CurrentSize;
        // 当前堆中元素数目
    int MaxSize;                          // 堆所能容纳的最大元素数目
    void swap(int pos_x, int pos_y);      // 交换位置x和y的元素
    void BuildHeap();                     // 建堆
public:
```

堆的类定义和筛选法(2)

```
public:
    MinHeap(const int n);                          // 构造函数，n表示 堆的最大元素数目
    virtual ~MinHeap(){delete []heapArray;};       // 析构函数
    bool isEmpty( );                               // 如果堆空，则返回真
    bool isLeaf(int pos) const;                    // 如果是叶结点，返回TRUE
    int leftchild(int pos) const;                  // 返回左孩子位置
    int rightchild(int pos) const;                 // 返回右孩子位置
    int parent(int pos) const;                     // 返回父结点位置
    bool Remove(int pos, T& node);                 // 删除给定下标的元素
    bool Insert(const T& newNode);                 // 向堆中插入新元素newNode
    T& RemoveMin();                                // 从堆顶删除最小值
    void SiftUp(int position);                     // 从position向上开始调整，使序列成为堆
    void SiftDown(int left);                       // 向下筛选，参数left表示开始处理的数组下标
};
```

堆的类定义和筛选法(3)

```
template<class T>
MinHeap<T>::MinHeap(const int n)  {
        if (n <= 0)
                return;
        CurrentSize = 0;
        MaxSize = n;                                    // 初始化堆容量为n
        heapArray = new T[MaxSize];                     // 创建堆空间
        // 此处进行堆元素的赋值工作
        BuildHeap();
}
template<class T>
bool MinHeap<T>::isLeaf(int pos) const  {
        return (pos >= CurrentSize/2) && (pos < CurrentSize);
}
template<class T>
void MinHeap<T>::BuildHeap()  {
        for (int i = CurrentSize/2-1; i >= 0; i--)              // 反复调用筛选函数
                SiftDown(i);
}
```

堆的类定义和筛选法(4)

```
template<class T>
int MinHeap<T>::leftchild(int pos) const  {
        return 2*pos + 1;                                          // 返回左孩子位置
}
template<class T>
int MinHeap<T>::rightchild(int pos) const  {
        return 2*pos + 2;                                          // 返回右孩子位置
}
template<class T>
int MinHeap<T>::parent(int pos) const  {
        return (pos-1)/2;                                          // 返回父结点位置
}
template <class T>
bool MinHeap<T>::Insert(const T& newNode)  {    // 向堆中插入新元素newNode
        if (CurrentSize == MaxSize)                                // 堆空间已经满
                return FALSE;
        heapArray[CurrentSize] = newNode;
        SiftUp(CurrentSize);                                       // 向上调整
        CurrentSize++;
        return TRUE;
}
```

堆的类定义和筛选法(5)

```
template<class T>
T& MinHeap<T>::RemoveMin()        {                          // 从堆顶删除最小值
    if (CurrentSize == 0) {
        cout<< "Can't Delete";                               // 调用RemoveMin之前，需要判断堆非空
        exit(1) ;
    }
    else  {
        swap(0,--CurrentSize);                               // 交换堆顶和最后一个元素
        if (CurrentSize>1)
            SiftDown(0);                                     // 从堆顶开始筛选
        return heapArray[CurrentSize];
    }
}
template<class T>
bool MinHeap<T>::Remove(int pos, T& node)  {    // 删除给定下标的元素
    if ((pos < 0) || (pos >= CurrentSize))
        return false;
    node = heapArray[pos];
    heapArray[pos] = heapArray[--CurrentSize];               // 用最后的元素值替代删除位置的元素
    if (heapArray[parent(pos)] > heapArray[pos])
        SiftUp(pos);                                         // 当前元素小于父结点，需要上升调整
    else SiftDown(pos);                                      // 当前元素大于父结点，向下筛
    return true;
}
```

堆的类定义和筛选法(6)

```
template<class T>
void MinHeap<T>::SiftUp(int position)  {
    // 从position向上开始调整
     int temppos = position;
     T temp = heapArray[temppos];
     while ((temppos>0) && (heapArray[parent(temppos)]>temp)) {
              heapArray[temppos] = heapArray[parent(temppos)];
              temppos = parent(temppos);
     }
     heapArray[temppos] = temp;
}
```

堆的类定义和筛选法(7)

```
template <class T>
void MinHeap<T>::SiftDown(int left)  {
        int i = left;                                   // 标识父结点
        int j = leftchild (i);                          // 标识关键值较小的子结点
        T temp = heapArray[i];                          // 保存父结点
        while (j < CurrentSize) {                       // 过筛
                if ((j < CurrentSize-1) && (heapArray[j]>heapArray[j + 1]))
                //若有右子节点，且小于左子节点
                        j++;
                // j指向右子结点
                if (temp>heapArray[j]) {  //若父节点大于子节点的值则交换位置
                        heapArray[i] = heapArray[j];
                        i = j;
                        j = leftchild(j);
                }
                else break;  //堆序满足，跳出
        }
        heapArray[i] = temp;
}
```

建堆效率

- 对于n个结点的堆，其对应的完全二叉树的层数为log$n$。
- 设i表示二叉树的层编号，则第i层上的结点数最多为$2^i(i \geq 0)$。
- 建堆的过程中，对每一个非叶子结点都调用了一次SiftDown调整算法，而每个数据元素最多向下调整到最底层，即第i层上的结点向下调整到最底层的调整次数为log$n$ – i。因此，建堆的计算时间为

$$\sum_{i=0}^{\log n} 2^i \cdot (\log n - i)$$

（公式5.3）

令j = logn – i，代入5.3式得

（公式5.4）

$$\sum_{i=0}^{\log n} 2^i \cdot (\log n - i) = \sum_{j=0}^{\log n} 2^{\log n - j} \cdot j = \sum_{j=0}^{\log n} n \cdot \frac{j}{2^j} < 2n$$

建堆效率

- 建堆算法的时间复杂度是O(n)。这就说明可以在线性时间内把一个无序的序列转化成堆序
- 由于堆有log n层深，插入结点、删除普通元素和删除最小元素的平均时间代价和最差时间代价都是O(log n)
- 最小堆只适合于查找最小值，查找任意值的效率不高

5.5.2 优先队列

- 优先队列(priority queue)是一种有用的数据结构。它是0个或多个元素的集合，每个元素都有一个关键码值，执行的操作有查找、插入和删除一个元素。
- 优先队列的主要特点是支持从一个集合中快速地查找并移出具有最大值或最小值的元素。最小优先队列，适合查找和删除最小元素；最大优先队列中，适合查找和删除最大元素。

5.6 Huffman树及其应用

- **5.6.1 Huffman树**
- **5.6.2 Huffman编码**

5.6.1 Huffman树

- 假设有n个权值分别为w_0，w_1，…，w_{n-1}(n≥2)的结点，求带权外部路径长度就是要构造一棵具有n个外部结点的扩充二叉树，每一个外部结点k_i取w_i作为它的权，l_i表示该外部结点的路径长度，则带权外部路径长度可记作

$$\sum_{i=0}^{n-1} w_i \cdot l_i$$

其中带权外部路径长度最小的二叉树称为Huffman树

5.6.1 Huffman树

例如，图5.18中表示了三棵具有4个外部结点的二叉树，各外部结点的权值分别为6，2，3，4。

它们的带权外部路径长度分别为：

(a) 6×2 + 2×2 + 3×2 + 4×2 = 30

(b) 6×2 + 2×3 + 3×3 + 4×1 = 31

(c) 6×1 + 2×3 + 3×3 + 4×2 = 29

其中，5.18(c)中所示的二叉树外部带权路径长度最小。可以验证，它就是一棵Huffman树，也就是说，这棵树在所有的具有6，2，3，4权值的叶结点的二叉树中带权外部路径长度最小。

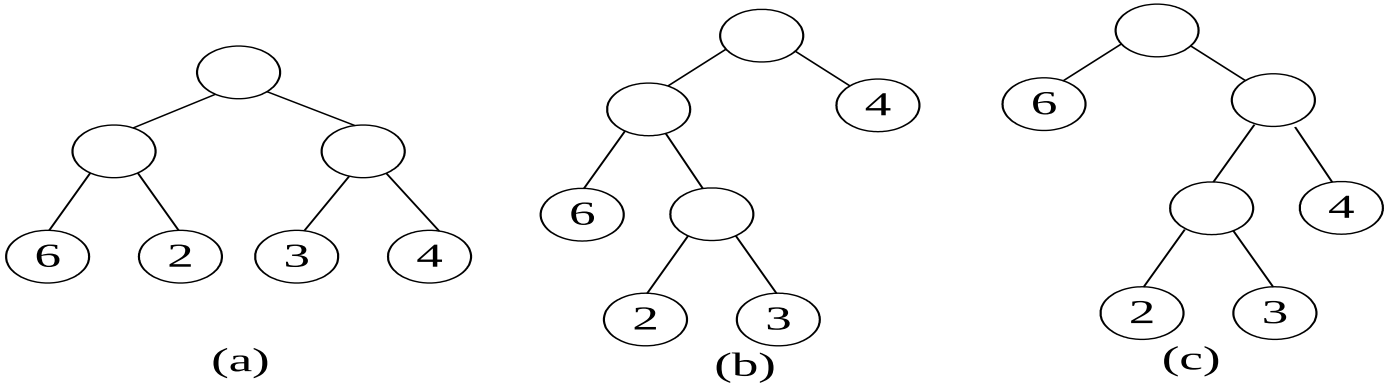


图5.18 具有不同带权外部路径长度的二叉树

5.6.1 Huffman树

建立Huffman编码树：

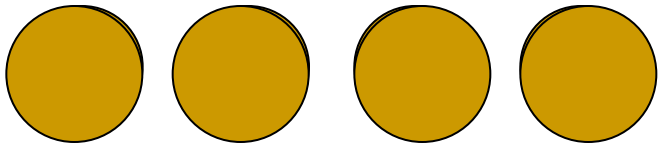
(1) 对于给定的n个权值w_0，w_1，…，w_{n-1}(n≥2)，构成n棵二叉树的集合T = { T_0，T_1，T_2，…，T_{n-1}}，使得每一棵扩充二叉树只具有一个带权为w_i的根结点。

(2) 构造一棵新的扩充二叉树，在集合T中找出两个权值最小的树作为新树根结点的左右子树，把新树根结点的权值赋为其左右子树根结点的和。

(3) 在集合T中删除这两棵树，并把得到的新扩充二叉树加入到集合中。

(4) 重复步骤(2)、(3)的操作，直到集合T中只含有一棵树为止。

5.6.1 Huffman树



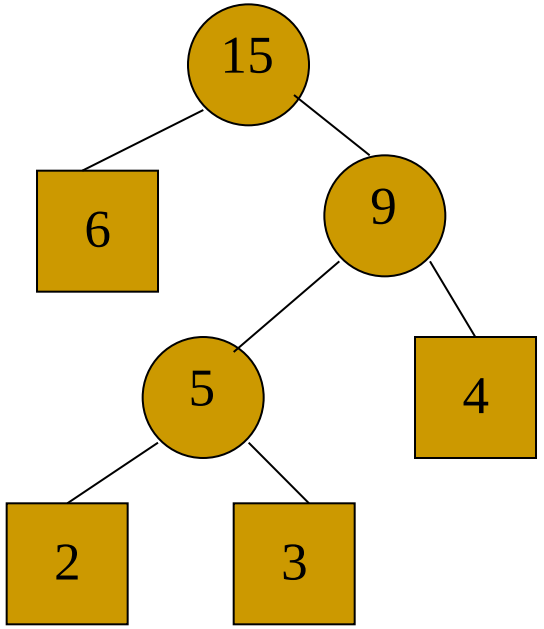


图5.19
Huffman树的构造过程

5.6.1 Huffman树

【代码5.12】 Huffman树的类定义

```
template<class T>
class HuffmanTree  {
private:
    HuffmanTreeNode<T> *root;       // Huffman树的根结点
    void MergeTree ( HuffmanTreeNode<T> &ht1, HuffmanTreeNode<T> &ht2,
    HuffmanTreeNode<T> *parent);
      // 把以ht1和ht2为根的两棵Huffman树合并成一棵以parent为根的二叉树
    void DeleteTree(HuffmanTreeNode<T> *root);
     // 删除Huffman树或其子树
public:
   HuffmanTree(T weight[],int n);
     // 构造Huffman树，参数weight为权值数组，n为数组长度
    virtual ~HuffmanTree(){DeleteTree(root);};                  // 析构函数
};
```

5.6.1 Huffman树

```
template<class T>
HuffmanTree<T>::HuffmanTree(T weight[], int n)  {
        MinHeap< HuffmanTreeNode<T> > heap(n);    // 最小值堆
        HuffmanTreeNode<T> *parent, firstchild, secondchild;
        HuffmanTreeNode<T> *NodeList = new HuffmanTreeNode<T>[n];
        for (int i = 0;i < n;i++) {                                  // 初始化
                NodeList[i].info = weight[i];
                NodeList[i].parent = NodeList[i].left = NodeList[i].right =
    NULL;
                heap.Insert(NodeList[i]);            // 向堆中添加元素
        }
        for (i = 0;i < n-1;i++) {          // 通过n-1次合并建立Huffman树
```

5.6.1 Huffman树

```
            parent = new HuffmanTreeNode<T>;// 申请一个分支结点
            firstchild = heap.RemoveMin();        // 选择权值最小的结点
            secondchild = heap.RemoveMin();     // 选择权值次小的结点
            MergeTree(firstchild,secondchild,parent);
            // 将权值最小的两棵树合并到parent树
            heap.Insert(*parent);
            // 把parent插入到堆中去
            root = parent;
            // Huffman树的根结点赋为parent
        }
        delete []NodeList;
}
```

5.6.2 Huffman编码

- Huffman树的一个重要应用就是解决数据通信中的二进制编码问题。

  设　　$D=\{d_0, \ldots, d_{n-1}\}$，

  $W=\{W_0, \ldots, W_{n-1}\}$

  D为需要编码的字符集合，W为D中各字符出现的频率，要对D里的字符进行二进制编码，使得：

  - $\sum_{i=0}^{n-1} w_i l_i$ 最小，其中，li为第i个字符的二进制编码长度。

- 由此可见，设计电文总长度最短的编码问题就转化成了设计字符出现频率作为外部结点权值的Huffman树的问题。

5.6.2 Huffman编码

- **Huffman编码过程如下：**
  - 用d_0，d_1，…，d_{n-1}作为外部结点构造具有最小带权外部路径长度的扩充二叉树
  - 把从每个结点引向其左子结点的边标上号码**0**，从每个结点引向其右子结点的边标上号码**1**
  - 从根结点到每个叶结点路径上的编号连接起来就是这个外部结点所代表字符的编码。得到的二进制前缀码就称作**Huffman**编码

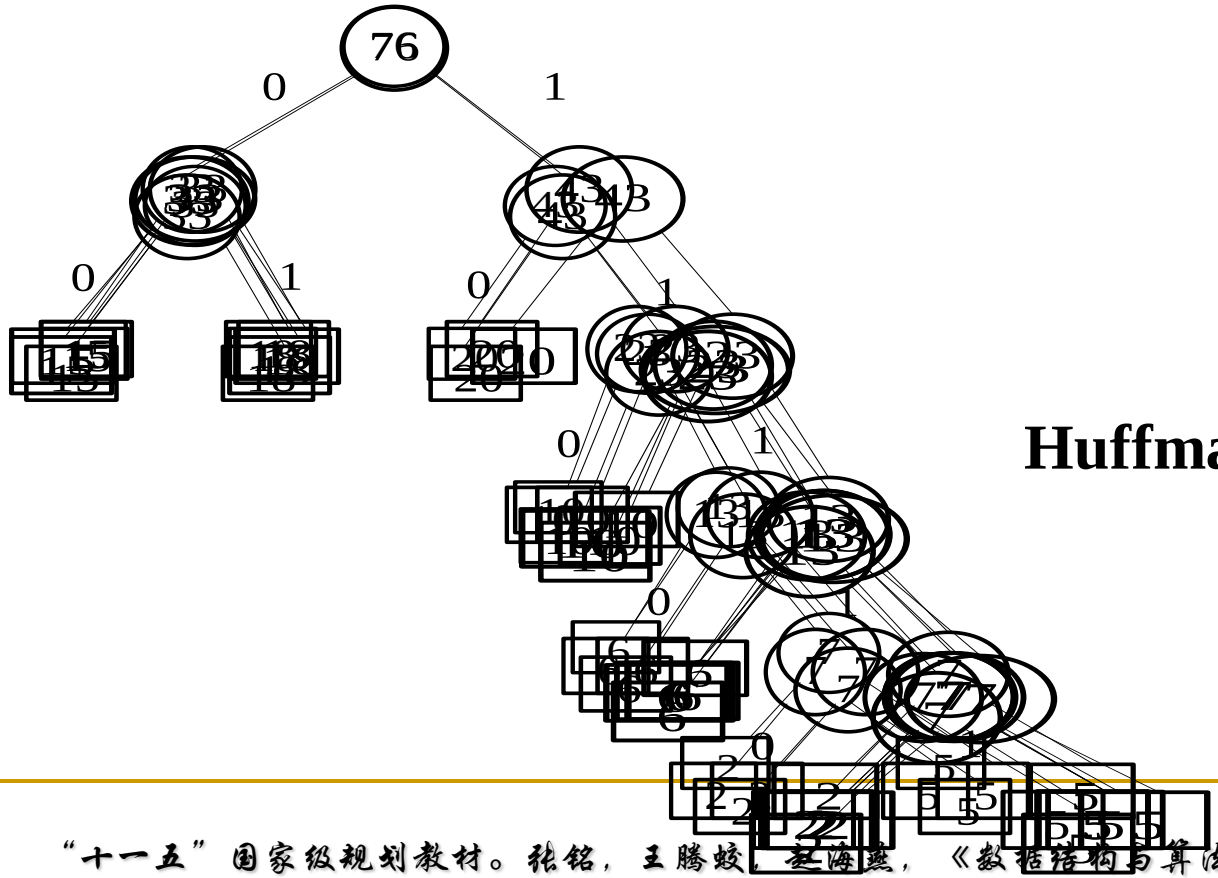
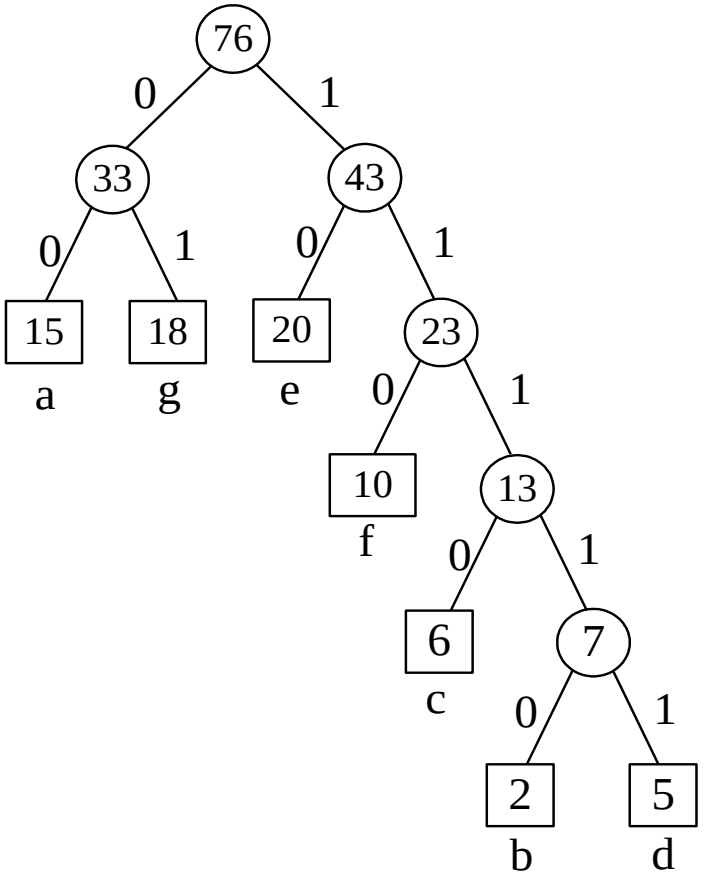


图5.20

Huffman编码示例

5.6.2 Huffman编码

例如，在一个数据通信系统中使用的字符是a，b，c，d，e，f，g，对应的频率分别为15，2，6，5，20，10，18。



各字符的二进制编码为：

a：00　　b：11110　　c：1110

d：11111　e：10　　f：110

g：01

5.6.2 Huffman编码

- 用Huffman算法构造出的扩充二叉树给出了各字符的编码，同时也用来译码
- 从二叉树的根开始，把二进制编码每一位的值与Huffman树边上标记的0，1相匹配，确定选择左分支还是右分支，直至确定一条到达树叶的路径。一旦到达树叶，就译出了一个字符。然后继续用这棵二叉树继续译出其它二进制编码

5.7 二叉树知识点总结

- 二叉树的概念及主要性质
- 二叉树的抽象数据类型及周游方法
- 二叉树的存储结构
- 二叉搜索树及其应用
- 堆的概念、性质与构造
- **Huffman**树的主要思想与具体应用

The End

Thank You!

http://www.jpk.pku.edu.cn/pkujpk/course/sjjg

张铭，王腾蛟，赵海燕

高等教育出版社，2008.6。“十一五”国家级规划教材