第5章 树和二叉

知识要点

(1) 掌握一般树的定义与基本术语；
(2) 掌握二叉树的定义、性质及存储结构；
(3) 熟练掌握二叉树的遍历及递归和非递归的遍历算法；
(4) 进一步了解哈夫曼树、二叉查找树、平衡二叉树、堆与优先队列等二叉树的多种应用；
(5) 熟悉树的存储结构及树和森林与二叉树之间的转换方法，了解树和森林的遍历方法。

西安交通大学计算机科学与技术系

2016

5. 1树的定义与基本术语

5.1.1 树的定义

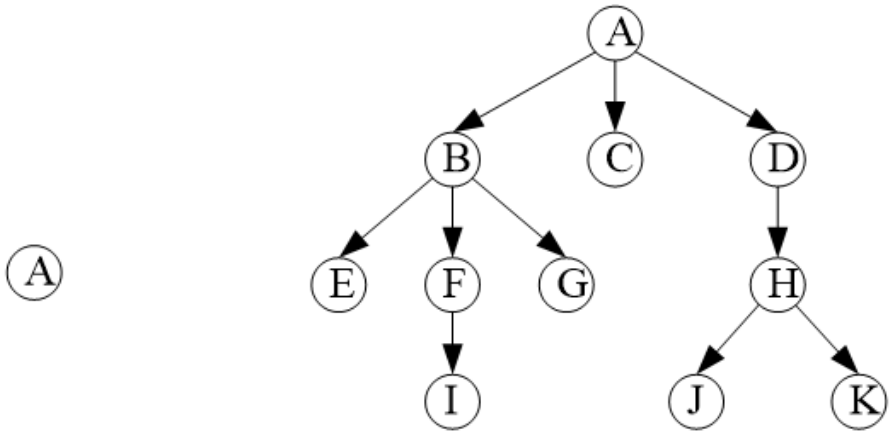
树T是包含n (n ≥ 0)个结点的有限集合。

当n = 0时，树T为空树。

当n > 0时，树T为非空树。在任意一棵非空树，都有：

- (1) 有且仅有一个特定的结点R称为树T的根结点；
- (2) 除根之外的其余结点可被分成m个(m ≥ 0)互不相交的有限集合T_1，T_2，…，T_m，其中每一个子集合本身也是一棵树，称为树T的子树，并且每个子树都有其对应的根结点分别为R_1，R_2，…，R_m，称为树根R的孩子。

□ 树的定义是一个递归定义。



(a) 只有根结点的树　　(b) 一般的树

图 5-1 树的示例

(a)是只有一个根结点的树；

(b)是有11个结点的树T，其中结点A是树的根，除根结点之外，其余结点分成三个互不相交的集合：T_1={B, E, F, G, I}，T_2={C}，T_3={D, H, J, K}，T_1，T_2，T_3都是以结点A为根的子树。这三棵子树本身也是一棵树

5.1.2 相关的基本术语

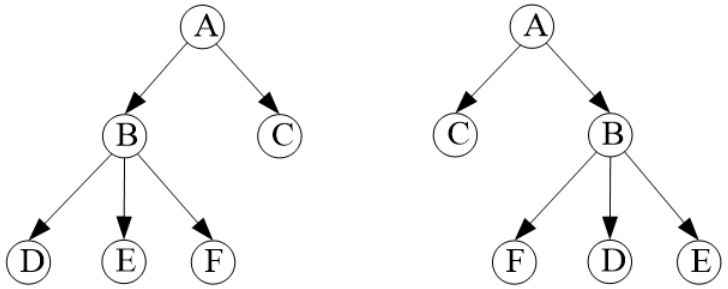


图 5-2 结点顺序不同的两颗树

(1) **结点的度**：指结点所拥有的子树个数。

(2) **树的度**：指树内各结点的度的最大值。

(3) **叶子结点**：指度为0的结点，也称终结点。

(4) **分支结点**：指度不为0的结点，又称非终结点或内部结点。

(5) **孩子结点与双亲结点**：树中结点的子树的根称为根结点的孩子结点或子结点，而该结点又称为其孩子结点的双亲结点。

(6) **兄弟**：同一个双亲的孩子之间互称兄弟结点。

(7) **堂兄弟**：其双亲在同一层的结点之间互称堂兄弟。

(8) **祖先**：一个结点的祖先是从根结点到该结点所经分支上的所有结点。

(9) **子孙**：一个结点的子孙是以该结点为根的子树中的所有结点。

(10) **结点的层次**：结点的层次从根开始算起，根结点的层次为1，其余结点的层次等于其双亲结点的层次加1。

(11) **树的深度**：树中结点的最大层号称为树的深度

(12) **树的高度**：树的总层数成为该树的高度。

(13) **有序树**：如果将树中每个结点的各子树看成是从左至右有顺序的（即不能互换），则称该树为有序树。在有序树中，最左边的子树根称为第一个孩子，最右边的称为最后一个孩子。

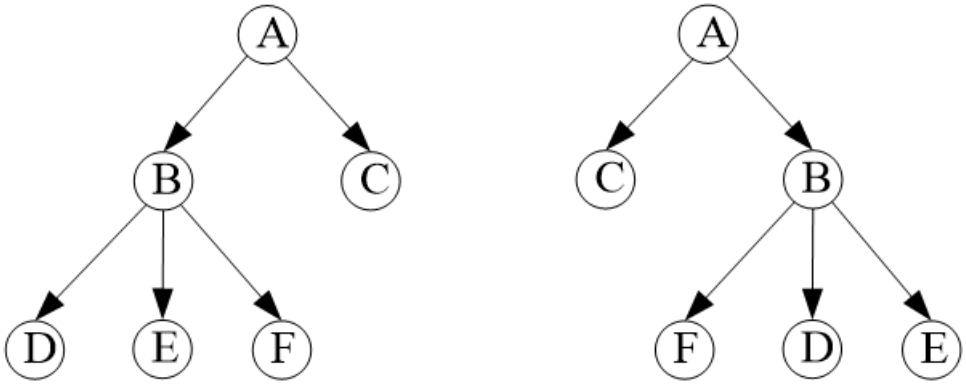


图 5-2 结点顺序不同的两颗树

(14) **无序树**：如果将树中每个结点的各子树看成是无顺序的（即可互换），则称该树为无序树。

(15) **森林**：是指m(m ≥ 0)棵互不相交的树的集合。

注：如果删去一棵树的根结点，剩余的子树就构成了森林；反之，加上一个结点作为树的根，森林就变成了一棵树。

5.2 二叉树的定义、性质和存储结构

二叉树T是包含n($n \geq 0$)个结点的有限集合。

当n = 0时，二叉树T为空二叉树。

当n > 0时，二叉树T为非空二叉树。

在任意一棵非空二叉树中，都有：

(1) 有且仅有一个特定的结点R称为二叉树T的根结点；

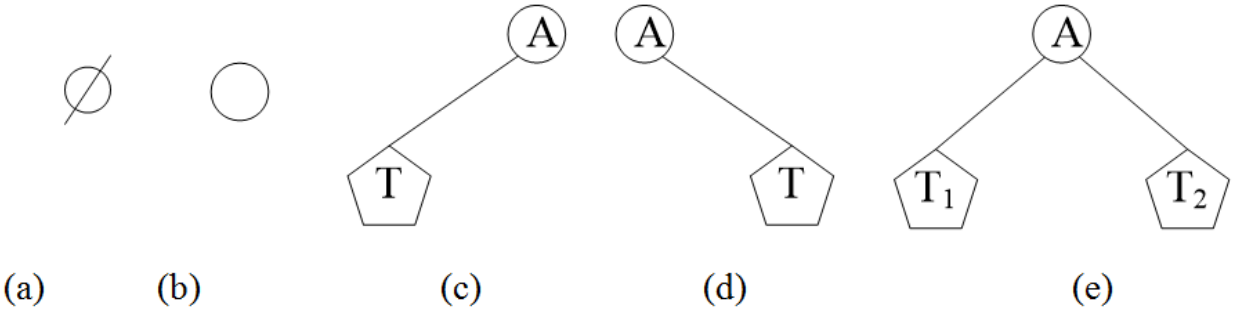
(2) 除根结点之外的其余结点被分成两个互不相交的有限集合T_1，T_2，其中集合T_1，T_2本身也是一棵二叉树，这两棵二叉树分别称为根结点R的左子树和右子树，并且这两棵子树的根分别称为二叉树根结点R的左孩子和右孩子结点。

◆树和二叉树是两种不同的数据结构。

二叉树的定义也是一个递归的定义

注意：每个结点最多只能有两棵子树，并且有左右之分，其次序不能任意颠倒，即使只有一棵子树，也要区分是左子树还是右子树。

二叉树有五种基本形态：



(a)空二叉树　(b)只有根结点的二叉树　(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 5-3　二叉树的五种基本形态

思考题：请问具有3个结点的树和二叉树，各有几种形态？

5.2.2 二叉树的主要性质

性质1：在二叉树的第i层上至多有2^{i-1}个结点(i≥1)。

归纳法可以证明之。

性质2：高度为k的二叉树至多有2^k-1个结点(k≥1)。

$$\sum_{i=1}^{k}(\text{第 i 层上的最大结点数}) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

性质3：对于任意一棵非空二叉树T，如果其叶子结点的个数为n_0，度为2的结点数为n_2，则有：$n_0 = n_2 + 1$。

由于存在：$n = n_0+n_1+n_2$

$n = e+1$

$e = n_1+2n_2$

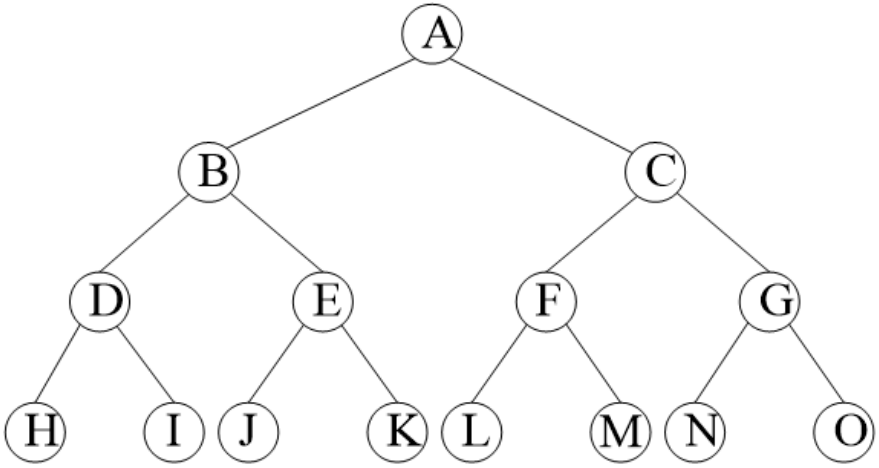
◆ 满二叉树和完全二叉树是二叉树的两种特殊情形。

一棵高度为k且有2^k-1个结点的二叉树称为满二叉树。

- 满二叉树中每一层上的结点数都达到最大值；
- 满二叉树中不存在度数为1的结点；
- 每个分支结点均有两棵高度相同的子树，且叶子都在最下一层上。

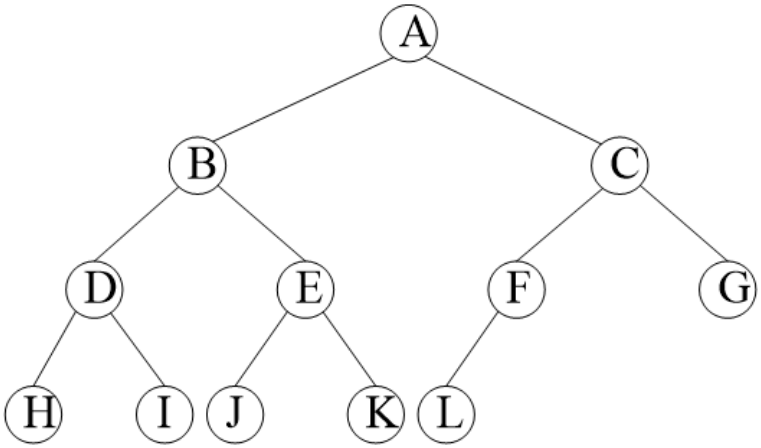
•按照从左到右、从上到下的顺序进行编号，称该编号序列为二叉树的层序编号。

•对高度为k的满二叉树的结点层序编号是从0到2^k - 2进行连续编号。



(a) 满二叉树

- **一棵二叉树每个结点的层序编号与对应位置的满二叉树层序编号完全一致，则称其为完全二叉树。**
- 假设从满二叉树中连续删除m个元素，其层序编号为2^k -1-i ($1 \le i \le m$)，所得到二叉树也被称为完全二叉树。
- 在一棵完全二叉树中至多只有最下面两层上的结点的度数可以小于2，并且最下一层的结点都集中在该层最左边的若干位置上。

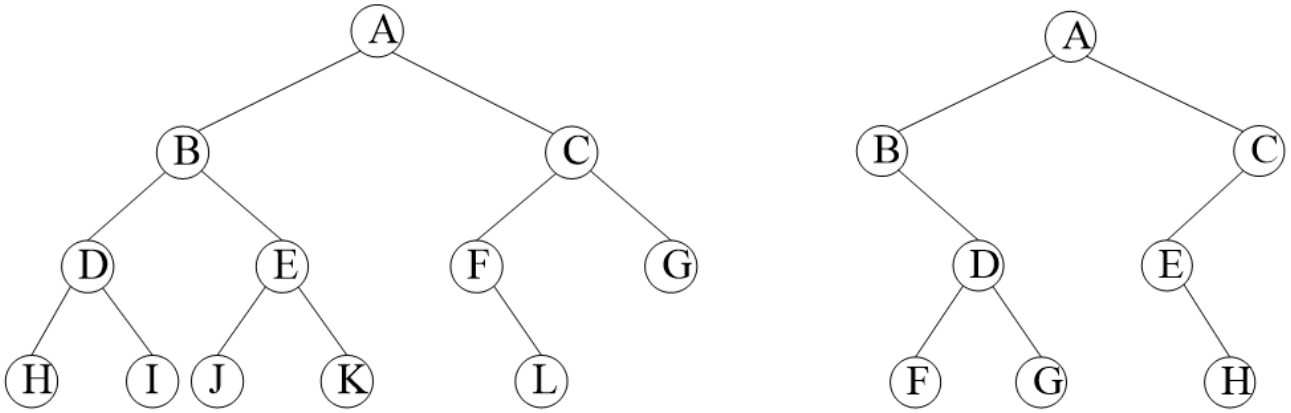


(b) 完全二叉树

- **结论：满二叉树是完全二叉树，但完全二叉树不一定是满二叉树。换句话说，满二叉树是完全二叉树的特殊情况。**

在完全二叉树的最下一层上，从最右边开始连续删去若干结点后得到的二叉树仍然是一棵完全二叉树。

在完全二叉树中，若某个结点没有左孩子，则它一定没有右孩子。



(c) 两棵非完全二叉树

性质4：具有n个结点的完全二叉树的高度k= $\lfloor \log_2 n \rfloor + 1$。

性质5*：如果对一棵有n个结点的完全二叉树（其高度为h）的结点按层序编号（从第1层到第h层，每层从左到右），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，无双亲结点；

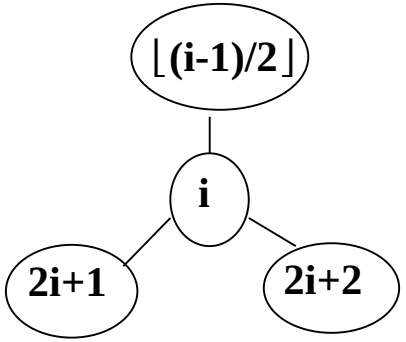
如果i > 0，则其双亲PARENT(i)是结点$\lfloor (i-1)/2 \rfloor$下取整。

(2) 如果$2i+1 \ge n$，则结点i无左孩子（结点i为叶子结点）；

否则，其左孩子LCHILD(i)是结点2i+1。

(3) 如果$2i+2 \ge n$，则结点i无右孩子

否则其右孩子RCHILD(i)是结点2i+2。



5.2.3 二叉树的存储结构

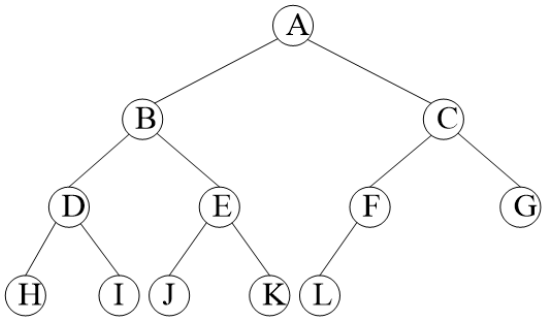
1. 顺序存储结构

顺序存储结构就是将二叉树中的所有结点按照一定的次序存储到一组地址连续的存储单元中。

必须把二叉树中的所有结点安排成一个适当的线性序列，使得结点在这个序列中的相互位置能反映出结点之间的逻辑关系。

◆对于完全二叉树

- 完全二叉树中结点的编号完全反映了结点之间的逻辑关系(性质5)。
- 将完全二叉树上编号为i的结点存储在一维数组中下标为i的分量中。
- 如图5-6(b)为图5-6(a)所示完全二叉树T_1的顺序存储结构。



(a) 完全二叉树 T_1

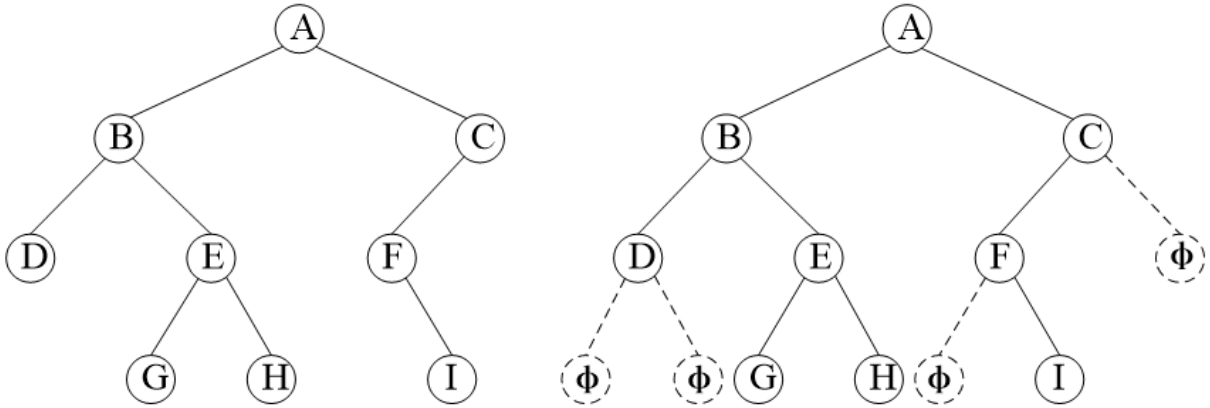
下标	0	1	2	3	4	5	6	7	8	9	10	11
结点	A	B	C	D	E	F	G	H	I	J	K	L

(b) 完全二叉树 T_1 的存储结构

图 5-6 完全二叉树的顺序存储结构

◆ 对于一般二叉树，

- 按照完全二叉树的形式存储树中的结点，即通过添加一些不存在的“空结点”，使之成为一棵完全二叉树。
- 将二叉树的每一个结点与完全二叉树上的结点相对应，存储在一维数组的相应分量中。
- 如图5-7(b)为图5-7(a)的一般二叉树T_2改造后的完全二叉树，图5-7(c)为改造后的二叉树的顺序存储结构，图中以"Φ"表示不存在此结点。



(a) 一般二叉树 T_2　　(b) 图 5-7(a)改造后的二叉树

数组下标：	0	1	2	3	4	5	6	7	8	9	10	11	12
	A	B	C	D	E	F	Φ	Φ	Φ	G	H	Φ	I

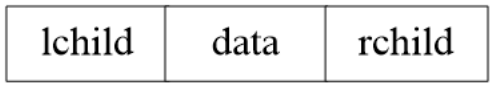
(c) 改造后的二叉树的顺序存储结构

图 5-7 一般二叉树的顺序存储结构

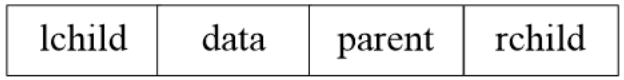
在最坏的情况下，一个深度为k且只有k个结点的单支树（树中不存在度为2的结点）却需要长度为$2^k - 1$的一维数组。

2. 链式存储结构

- 链式存储结构可以解决一般二叉树采用顺序存储结构时造成的空间浪费问题。
- 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。
- 表示二叉树的链表中的结点至少**包含三个域：数据域和左、右指针域。**
- **二叉链表**存储如图5-8(a)，**三叉链表**存储如图5-8(b)。



(a)含有两个指针域的结点结构



(b)含有三个指针域的结点结构

图 5-8 二叉树结点的存储结构

如图5-9所示为单支树的二叉链表。

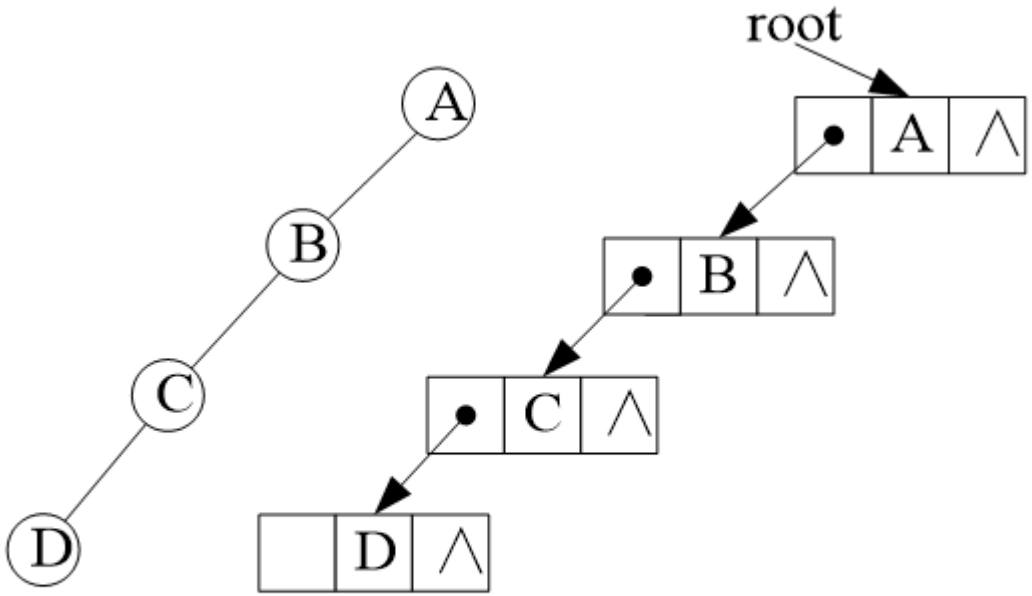
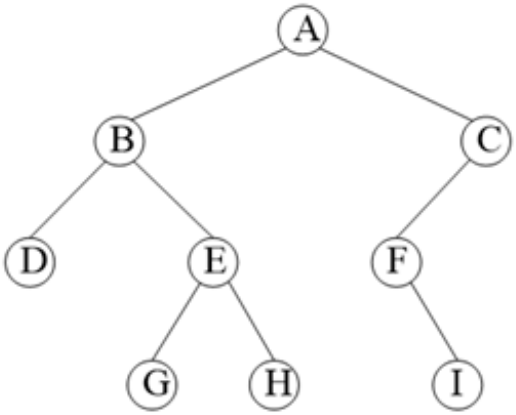
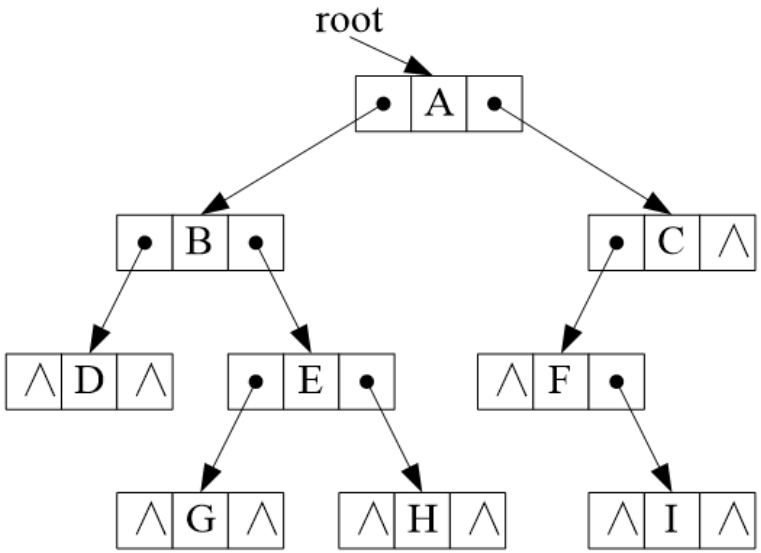
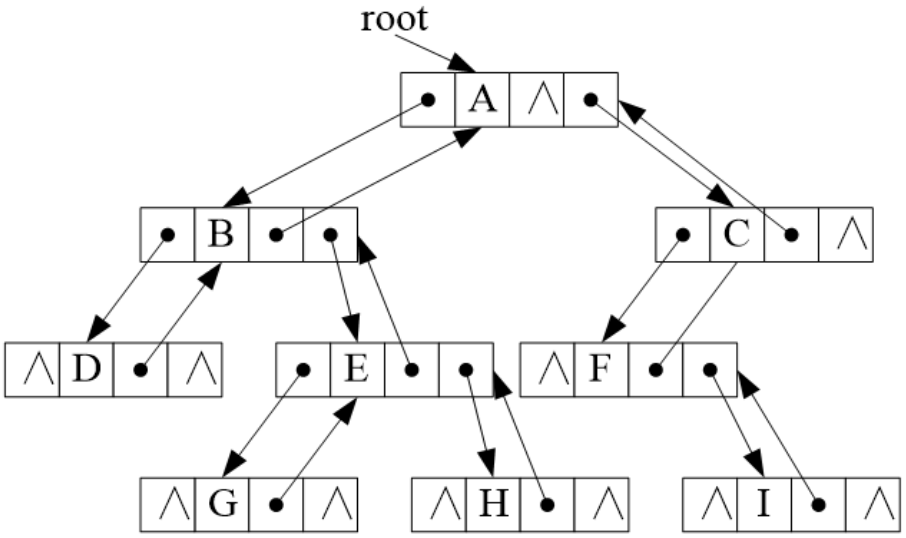


图 5-9 单支树的二叉链表





(a) 二叉链表



(b) 三叉链表

图 5-10 二叉树的链式存储结构

在二叉树的链式存储结构中，通常采用二叉链表来存储，二叉树的二叉链表存储结构的形式可描述如下：

算法5.1：二叉树的结点类的声明

```
template <class T>
class BinaryTreeNode{
private:
    T data;                          //二叉树结点数据域
    BinaryTreeNode<T>*  left;        //二叉树结点指向左子树的指针
    BinaryTreeNode<T>*  right;       //二叉树结点指向左子树的指针
```

```
public:
    BinaryTreeNode();                          //缺省构造函数
    BinaryTreeNode(const T& elem);    //给定数据的构造函数
    BinaryTreeNode(const T& elem, BinaryTreeNode<T>* l, BinaryTreeNode<T>* r);
        //给定数据的左右指针的构造函数
    ~BinaryTreeNode(){};
    T value() const;                                    //返回当前结点的数据
    BinaryTreeNode<T>*  leftchild() const;        //返回当前结点指向左子树的指针
    BinaryTreeNode<T>*  rightchild() const;       //返回当前结点指向右子树的指针
    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;                    //判定当前结点是否为叶结点,若是返回true
};
```

算法5.2：二叉树类定义：

```
template <class T>
class BinaryTree{
protected:
    BinaryTreeNode<T>* root;                          //二叉树根结点指针
public:
    BinaryTree() {root = NULL;}                       //构造函数
    BinaryTree(BinaryTreeNode<T>* r) {root = r;}
    ~BinaryTree() { DeleteBinaryTree(root); };        //析构函数
    bool isEmpty() { return root==NULL; };            //判断二叉树是否为空树
    void visit(BinaryTree<T>& curr){cout <<curr->data << " ";} //访问当前结点
    BinaryTreeNode<T>*& Root() {return root;};        //返回二叉树的根结点
    void CreateTree(const T& data, BinaryTreeNode<T>* left-tree,
     BinaryTreeNode<T>* right-tree);
     //以data作为根结点，ltree作为树的左子树，rtree作为树的右子树，构造一
     棵新二叉树
```

```
    void CreateTree(BinaryTreeNode<T> *&r); //根据先序遍历序列构造二叉树
    void DeleteBinaryTree(BinaryTreeNode<T>* root); //删除二叉树及其子树
    void PreOrder(BinaryTreeNode<T>* root);          //前序遍历二叉树或其子树
    void InOrder(BinaryTreeNode<T>* root);           //中序遍历二叉树或其子树
    void PostOrder(BinaryTreeNode<T>* root);         //后序遍历二叉树或其子树
    void PreOrderWithoutRecusion(BinaryTreeNode<T>* root);//非递归前序遍历
    void InOrderWithoutRecusion(BinaryTreeNode<T>* root); //非递归中序遍历
    void PostOrderWithoutRecusion(BinaryTreeNode<T>* root);//非递归后序遍历
    void LevelOrder(BinaryTreeNode<T>* root);        //按层次遍历二叉树或其子树
};
```

5.3 二叉树的遍历

- 由二叉树的递归定义可知，一棵非空的二叉树是由三个基本部分组成：根结点，左子树和右子树。
- 假设以D、L、R分别表示访问根结点、遍历左子树、遍历右子树，那么就会有六种遍历方案：DLR、LDR、LRD、DRL、RDL、RLD。
- 经常用到的次序总是先左(L)后右(R)，再把根结点穿插于其中，就构成了常见的三种遍历：
  - 先序（根）遍历 (Preorder Traversal)；
  - 中序（根）遍历 (Inorder Traversal)；
  - 后序（根）遍历 (Postorder Traversal)。
- 遍历的次序不同，导致结果不同。

5.3.1 二叉树的先序遍历

先序遍历算法的递归定义如下：

若二叉树为空，则遍历结束；

否则：

(1) 访问根结点；

(2) 先序遍历左子树；

(3) 先序遍历右子树。

即，按照“根—左子树—右子树”的次序递归地遍历二叉树。

算法5.3：先序遍历二叉树的递归算法

```
template<class T>
void BinaryTree<T>::PreOrder(BinaryTreeNode<T>* root){
    if (root == NULL)  return;
    visit(root->value());                    //访问根结点
    PreOrder(root->leftchild());             //先序遍历左子树
    PreOrder(root->rightchild());            //先序遍历右子树
}
```

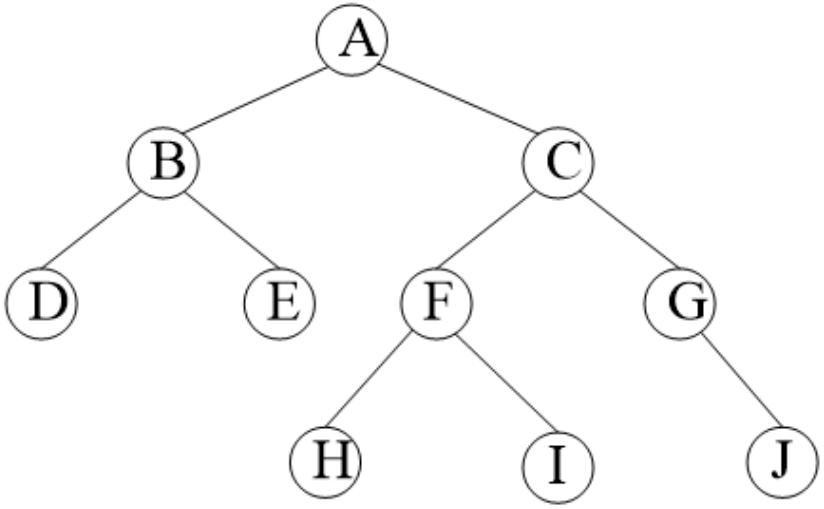


图 5-11 二叉树 T_3

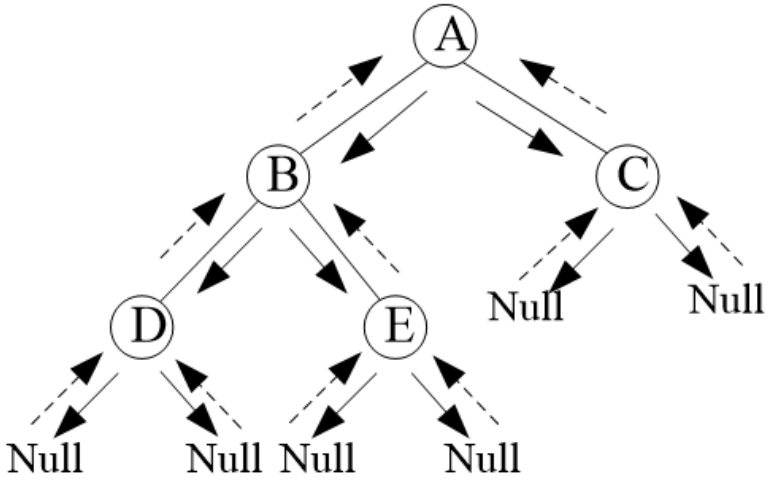


图 5-12 二叉树的遍历轨迹

如图5-11所示二叉树T_3的先序遍历序列为：

ABDECFHIGJ

二叉树遍历的非递归算法：

- 将递归算法改写为非递归算法，需要设置一个堆栈，用以保存结点指针。
- 保存结点的目的有两个：

①访问该结点；

②获得该结点的左右子树，当该结点的左子树遍历完后能够回溯，然后遍历其右子树。

算法5.4：先序遍历二叉树的非递归算法

```
template<class T>
void BinaryTree<T>::PreOrderWithoutRecusion(BinaryTreeNode <T> * root){
    stack<BinaryTreeNode<T>* > tStack;
    BinaryTreeNode<T>* pointer = root;
    while(!tStack.empty() || pointer){
        if (pointer){
            visit(pointer->value()); //访问当前结点
            tStack.push(pointer);    //当前结点地址入栈
            pointer = pointer->leftchild();    //当前链接结构指向左孩子
        } else{        //左子树访问完毕，转向访问右子树
            pointer = tStack.top(); //当前链接结构指向栈顶的元素
            tStack.pop();              //栈顶元素出栈
            pointer = pointer->rightchild(); }
    }
}
```

5.3.2 二叉树的中序遍历

中序遍历的递归定义如下：

若二叉树为空，则遍历结束；

否则：

(1) 中序遍历左子树；

(2) 访问根结点；

(3) 中序遍历右子树。

即，按照“左子树—根—右子树”的次序递归的遍历二叉树。

算法5.5：中序遍历二叉树的递归算法

```
template<class T>
void BinaryTree<T>::InOrder(BinaryTreeNode<T>* root){
    if (root == NULL) return;
    InOrder(root->leftchild());                    //中序遍历左子树
    visit(root->value());                          //访问根结点
    InOrder(root->rightchild());                   //中序遍历右子树
}
```

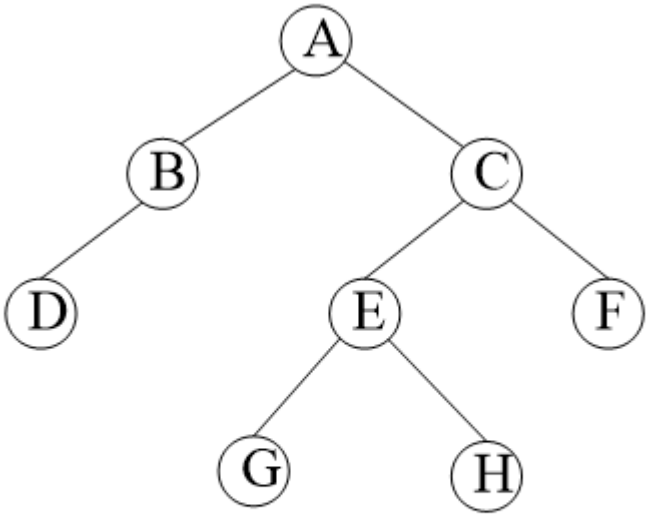


图 5-13 二叉树 T_4

如图5-13所示二叉树T_4的中序遍历序列为：

DBAGEHCF

注意：中序遍历的非递归算法也需要设置一个堆栈

算法5.6：中序遍历二叉树的非递归算法

```
template<class T>
void BinaryTree<T>::InOrderWithoutRecusion(BinaryTreeNode<T>* root){
    stack<BinaryTreeNode<T>* > tStack;
    BinaryTreeNode<T>* pointer = root;
    while(!tStack.empty() || pointer){
        if (pointer){
            tStack.push(pointer);          //当前结点地址入栈
            pointer = pointer->leftchild();   //当前链接结构指向左孩子
        } else{       //左子树访问完毕，转向访问右子树
            pointer = tStack.top();       //当前链接结构指向栈顶的元素
            tStack.pop();                 //栈顶元素出栈
            visit(pointer->value());      //访问当前结点
            pointer = pointer->rightchild();   //指向右孩子
        }
    }
}
```

5.3.3 二叉树的后序遍历

后序遍历的递归定义如下：

若二叉树为空，则遍历结束；

否则：

(1) 后序遍历左子树；

(2) 后序遍历右子树；

(3) 访问根结点。

即，按照“左子树—右子树—根”的次序递归的遍历二叉树。

后序遍历二叉树的递归算法描述如下：

算法5.7：后序遍历二叉树的递归算法

```
template<class T>
void BinaryTree<T>::PostOrder(BinaryTreeNode<T>* root){
    if (root == NULL) return;
    PostOrder(root->leftchild());          //后序遍历左子树
    PostOrder(root->rightchild());         //后序遍历右子树
    visit(root->value());                  //访问根结点
}
```

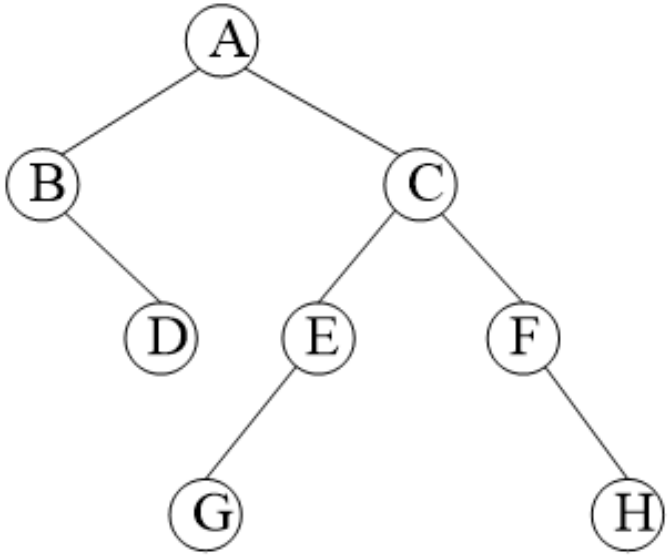


图 5-14 二叉树 T_5

如图5-14所示二叉树T_5的后序遍历序列为：

DBGEHFCA

算法5.8：后序遍历二叉树的非递归算法

```
enum Tag{L, R};
template <class T>
class StackNode{
public:
    BinaryTreeNode<T>* pointer;
    Tag tag;
};
template<class T>
void BinaryTree<T>::PostOrderWithoutRecusion(BinaryTreeNode<T>* root){
      stack<StackNode<T> >  tStack;
      StackNode<T>  Node;
      BinaryTreeNode<T>*  pointer = root;
```

```
    do{
        while (pointer != NULL){     //将左子树中的结点加Tag=L后压入栈中
            Node.pointer = pointer;
            Node.tag = L;
            tStack.push(Node);
            pointer = pointer->leftchild();
        }
        Node = tStack.top();         //栈顶元素出栈
        tStack.pop();
        pointer = Node.pointer;
        if ( Node.tag == R ){        //如果从右子树回来
            visit(pointer->value()); //访问当前结点
            pointer = NULL;          //置pointer为空，以继续弹栈
        } else {                     //如果从左子树回来，Node.tag==L
            Node.tag = R;            //标志域置为R，进入右子树
            tStack.push(Node);
            pointer = pointer->rightchild();
        }
    }while (!tStack.empty() || pointer);
}
```

二叉树的遍历算法与表达式的表示法之间有着密切的联系。

- 先序遍历序列为：-*A+*BCD/EF(波兰式)
- 中序遍历序列为：A*B*C+D-E/F (一般表示)
- 后序遍历序列：ABC*D+*EF/-（逆波兰式）

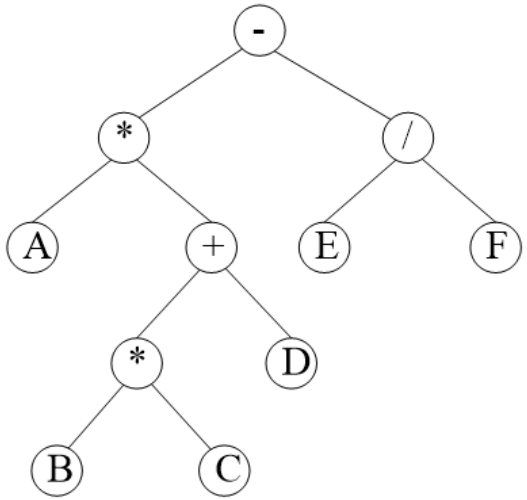


图 5-15 表达式 A*(B*C+D)-E/F 的二叉树表示

二叉树构造算法：根据二叉树先序遍历序列构造二叉链表存储算法。

算法的输入：二叉树的先序遍历序列，但是输入中必须加入空结点以表示空指针的位置，如图5-16所示二叉树T_6，其输入的先序遍历序列为：

ABD##E##CFH##I##G##

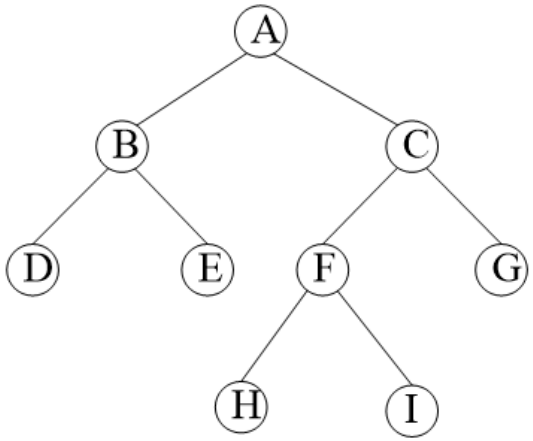


图 5-16 二叉树 T_6

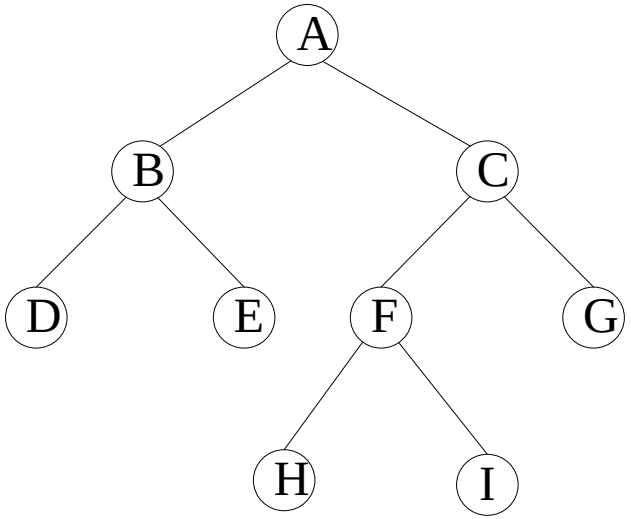
例如，输入的先序遍历序列为：
ABD##E##CFH##I##G##

算法5.10：二叉树的构造算法

```
template<class T>
void BinaryTree<T>::CreateTree(BinaryTreeNode<T> * &r){
    int ch;
    cin >> ch;
    if (ch == '#') r = NULL;
    else{    //读入非空符号
        r = new BinaryTreeNode<T>(ch);              //生成结点
        CreateTree(r->left);                        //构造左子树
        CreateTree(r->right);                       //构造右子树
    }
}
```

逐层遍历二叉树：

按照从顶层到底层的次序访问树中结点，在同一层中，从左到右依次访问。



逐层遍历得到的序列为：ABCDEFGHI。

算法5.11：逐层遍历二叉树

```
template<class T>
void BinaryTree<T>::LevelOrder(BinaryTreeNode<T> *root){
    queue<BinaryTreeNode<T>* > tQueue;
    BinaryTreeNode<T> *pointer = root;
    if (pointer) tQueue.enqueue(pointer);      //根结点入队列
    while (!tQueue.empty()){
        pointer = tQueue.dequeue();            //出队列，获得队列首结点
        visit(pointer->value());               //访问当前结点
        if (pointer->leftchild() != NULL)
            tQueue.enqueue(pointer->leftchild()); //左子树入队列
        if (pointer->rightchild() != NULL)
            tQueue.enqueue(pointer->rightchild()); //右子树入队列
    }
}
```

二叉树遍历算法的复杂度分析:

- 对于有n个结点的二叉树，这四种遍历算法的时间复杂度都为O(n)。
- 所需要的辅助空间为遍历过程中栈的最大容量，即树的高度，$k=\log_2^n+1$，$O(\log^n)$。
- 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) 。

5.4 二叉树应用1：哈夫曼树

哈夫曼(Huffman)树又称**最优二叉树**，

由Huffman在1952年提出，是带权路径长度最短的树。

哈夫曼树定义：假设有n个结点，每个结点的权值分别为$w_1,w_2,\ldots,w_n$，构造一棵以这n个结点为叶子结点的二叉树，则其中**带权路径长度WPL最小的二叉树称为哈夫曼树或最优二叉树**。

5.4.1 哈夫曼树的构造

1. 相关术语：

(1) **路径：** 从树中一个结点到另一个结点之间的分支构成这两个结点间的路径。

二叉树T_7，从结点A到结点F的路径为：A→B→D→F。

(2) **路径长度：** 路径上的分支数称为路径长度。(边数)

结点A到结点F的路径长度为3。

(3) **树的路径长度**：从树的根结点到每个叶子结点的路径长度之和称为树的路径长度。

该树的路径长度为2+3+3+1=9。

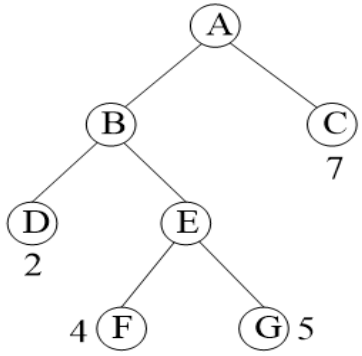


图 5-17 二叉树 T_7

(4) **叶子的带权路径长度：**从该叶结点到树的根结点的路径长度与该结点的权的乘积称为该结点的带权路径长度。

叶子结点G的带权路径长度为：3*5=15。

(5) **树的带权路径长度：**树中所有叶子结点的带权路径长度之和为树的带权路径长度，记作：

$$WPL = \sum_{i=1}^{n} w_i l_i$$

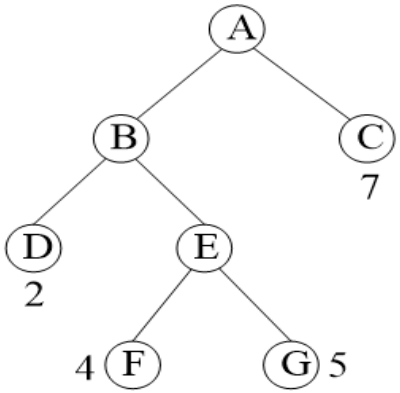
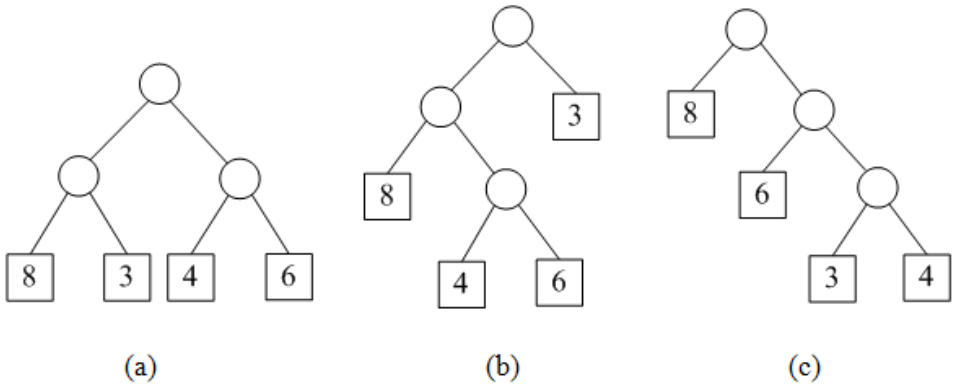


图 5-17 二叉树 T_7

图中树的带权路径长度*WPL*=2*2+4*3+5*3+7*1=38。

(6) **哈夫曼树：**假设有n个结点，每个结点的权值分别为$w_1,w_2,\ldots,w_n$，构造一棵以这n个结点为叶子结点的二叉树，则其中**带权路径长度WPL最小的二叉树称为哈夫曼树或最优二叉树**。



它们的带权路径长度分别为：

(a) $WPL = 8\times2 + 3\times2 + 4\times2 + 6\times2 = 42$

(b) $WPL = 8\times2 + 3\times1 + 4\times3 + 6\times3 = 49$

(c) $WPL = 8\times1 + 3\times3 + 4\times3 + 6\times2 = 41$

其中，5-18(c)所示二叉树的*WPL*值最小，可以证明，它就是哈夫曼树。

构造哈夫曼树的基本思想：

让权值越大的叶子结点离根结点的距离越近，而权值越小的叶子结点离根结点的距离越远。

2. 哈夫曼树的构造过程

基本算法如下：

(1) 由给定的n个权值$\{w_1, w_2, \ldots, w_n\}$构成有n棵二叉树的森林，其中每棵二叉树分别都是只含有一个带权为w_i的单根结点，其左右子树均为空。

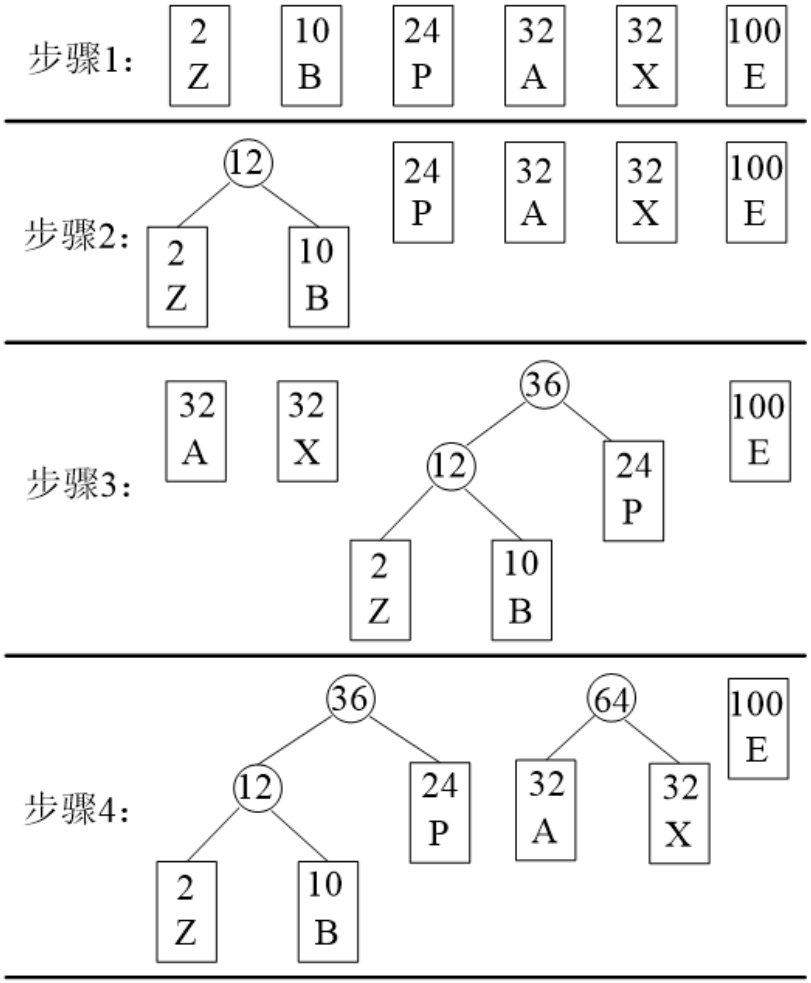
(2) 在森林中选取根结点的权值最小的两棵二叉树，分别作为左右子树构造一棵新的二叉树，新的二叉树的根结点的权值为其左右子树上根结点的权值之和。

(3) 从森林中删除(2)中选取的两棵树，同时将新得到的二叉树加入到森林中，此时森林中还有n-1棵树。

(4) 重复步骤(2)和(3)，直到森林中只剩一棵树为止，该树即为所求得的哈夫曼树。

【例】给定一组字符：{A, B, E, P, X, Z}，

对应的权重：{32, 10, 110, 24, 32, 2}



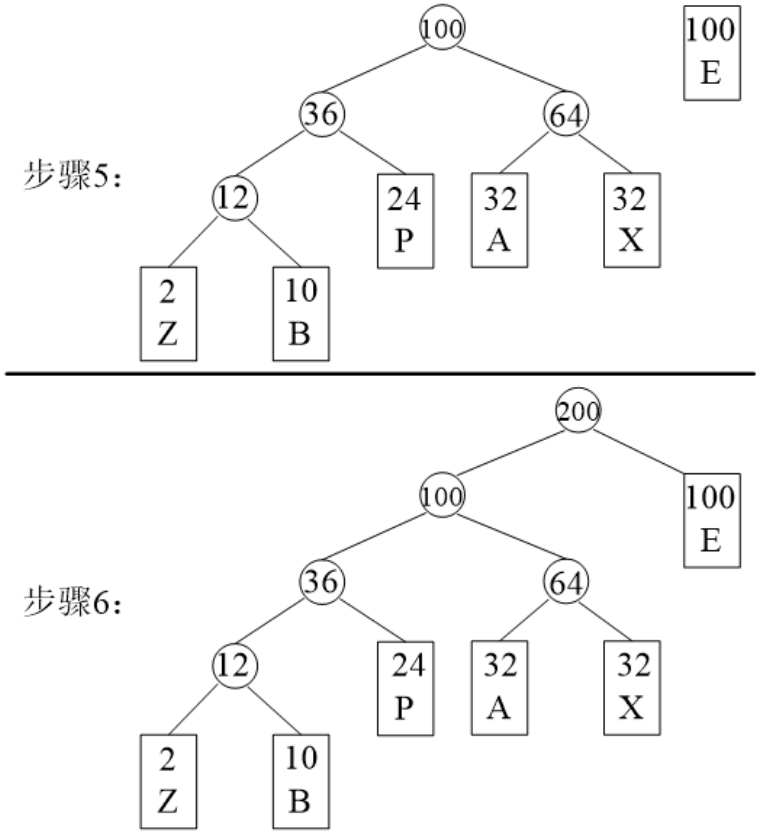


图 5-19 哈夫曼树的构造过程

注意：n个叶子结点构成的哈夫曼树的带权路径长度是确定的，但树形是不唯一的，因为将森林中两棵权值最小的子树合并时，哪棵做左子树，哪棵做右子树并不严格限制。

如图5-20所示的哈夫曼树与图5-19步骤6中所示的哈夫曼树具有相同的带权路径长度，但它们的树形不一样。

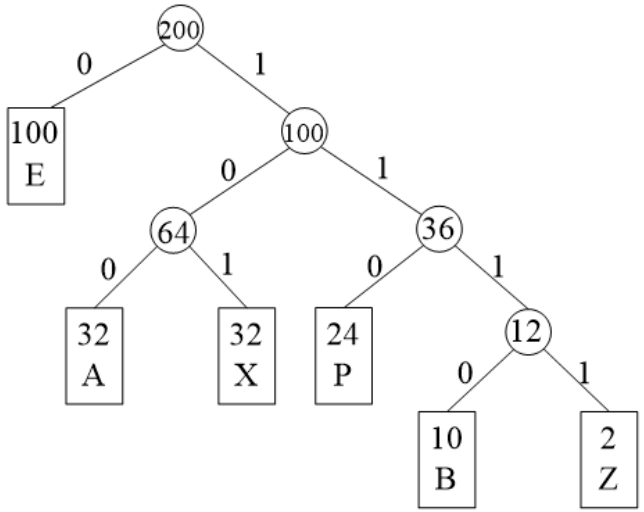


图 5-20 哈夫曼树

3. 哈夫曼树的具体实现

算法5.12：哈夫曼树结点类的定义

```
template<class T>
class HuffmanTreeNode{
    friend class HuffmanTree<T>;
private:
    HuffmanTreeNode<T> *left;
    HuffmanTreeNode<T> *right;
    int wgt;
    T elem;
```

```
public:
    HuffmanTreeNode() {left = right = NULL; wgt = 0;};
    HuffmanTreeNode(HuffmanTreeNode<T> *l, HuffmanTreeNode<T> *r,
      int weight, T element);
    ~HuffmanTreeNode();
    int weight() {return wgt;};
    T& element() {return elem;};
    bool isLeaf(){return !left && !right;};
    void setLeft(HuffmanTreeNode<T> *b){left = b;};
    void setRight(HuffmanTreeNode<T> *b){right = b;};
    HuffmanTreeNode<T>* Left() const {return left;};
    HuffmanTreeNode<T>* Right() const {return right;};
};
```

算法5.13：哈夫曼树类的定义

```
template<class T>
class HuffmanTree{
private:
   HuffmanTreeNode<T> *root;              //HuffmanTree的树根
public:
   HuffmanTree(){root = NULL;};
   HuffmanTree(T element,  int weight);
   HuffmanTree(HuffmanTree<T> *l, HuffmanTree<T> *r, int weight);
   void DeleteTree(HuffmanTreeNode<T> *root);    //删除Huffman树或其子树
   ~HuffmanTree(){ DeleteTree(root); };          //析构函数
   HuffmanTreeNode<T>* Root() {return root;}
   int weight() {return root->wgt;};
};
```

算法5.14：哈夫曼树构造的实现

```
//构造哈夫曼树：element数据存储字符，weight数组存储对应
    权值，n是数据长度
template<class T>
HuffmanTree<T>* BuildHuffTree(T element[], int weight[], int n){
    Priority_queue<HuffmanTree<T>*> QHTree;
    HuffmanTree<T> *lc, *rc, *p;
    //初始化：建立n棵单根树
    for (int i = 0; i < n; i++)
        QHTree.push(new HuffmanTree<T>(element[i],weight[i]));
```

```
//构造哈夫曼树
    while (QHTree.size() > 1){
            rc = QHTree.top();      QHTree.pop(); //取出权值最小树
            lc = QHTree.top();      QHTree.pop(); //取值权值第二小树
            p = new HuffmanTree<T>(lc,rc,lc->weight()+rc->weight());
            QHTree.push(p);  //合并两棵最小树后加入优先队列
    }
    return QHTree.top();
}
```

5.4.2 哈夫曼编码

- 哈夫曼树可以用来设计二进制的前缀编码，既满足前缀编码的条件，译码时不会产生二义性，又能保证报文的总编码长度最短。
- **具体做法为：**
  - ✓首先将字符集中的每个字符的使用频率作为权值构造一棵哈夫曼树；
  - ✓从根结点开始，分别把‘0’或‘1’标于树的每条边上
  - ✓‘0’对应于连接左孩子的边，‘1’对应于连接右孩子的边
  - ✓从根到叶子的路径上‘0’‘1’串即为该叶子哈夫曼编码

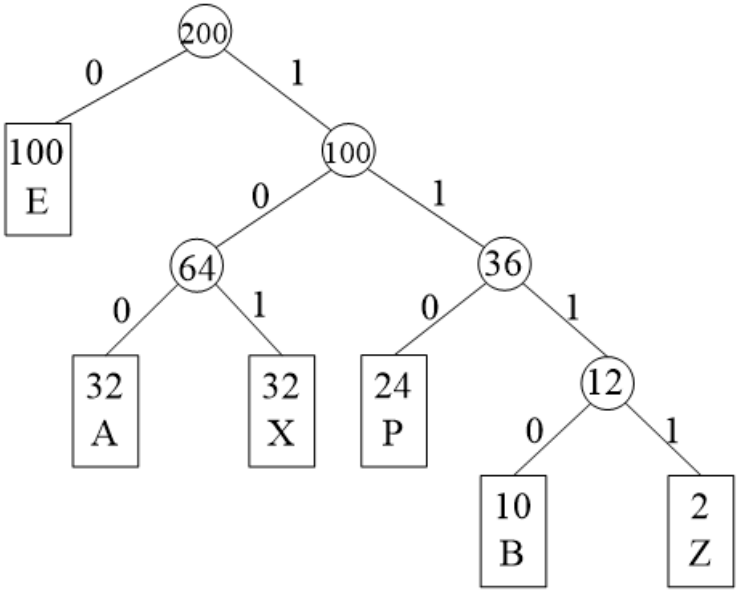


图 5-20 哈夫曼树

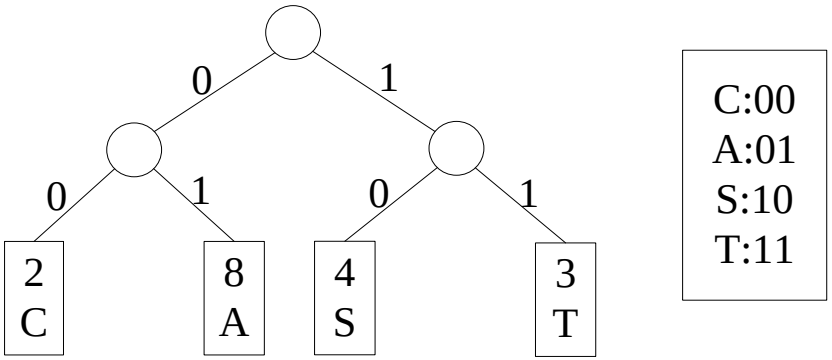
表 5-1 图 5-20 所示哈夫曼树所得字符的对应编码

字符	频率	哈夫曼编码	位
A	32	100	3
B	10	1110	4
E	110	0	1
P	24	110	3
X	32	101	3
Z	2	1111	4

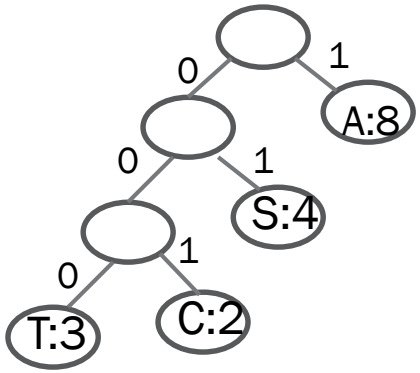
【例子1】：

给定一段报文：CASATTAASCATSAAAS，字符集合是{C, A, S, T}，各个字符出现的频率（次数）是W = {2, 8, 4, 3}。两种编码方式：

第一种编码方式是等长编码，它也是最简单的二进制编码方式，等长编码的每个字符的编码长度都相同（编码长度就是每个编码所含的二进制位数），表示n个不同字符需要位。WPL=2*2+8*2+4*2+3*2=34



第二种编码方式为不等长编码。若报文中可能出现26个不同字符，则等长编码的长度为5。在实际应用当中，各个字符的出现频率或使用次数是不相同的，因此，在传送报文时，为了使编码长度尽可能的短，可以将每个字符的编码设计为不等长的，使用频率高的字符分配一个相对较短的编码，使用频率较低的字符分配一个较长的编码，以优化整个报文的编码长度。



C(2): 001
A(8): 1
S(4): 01
T(3): 000
WPL=2*3+8*1+4*2+3*3=31

算法5.15：哈夫曼编码的实现

```
typedef vector<bool> Huff_Code;   //8 bit code of one char
map<char, Huff_Code> Huff_Dic;  //Huffman code dictionary
//哈夫曼编码
template<class T>
void Huffman_Code(HuffmanTreeNode<T> *r, Huff_Code curcode){
    if (r->isLeaf()){
        Huff_Dic[r->element()] = curcode;
        return ;
    }
    Huff_Code lcode = curcode;
    Huff_Code rcode = curcode;
    lcode.push_back(false);
    rcode.push_back(true);
    Huffman_Code(r->Left(), lcode);
    Huffman_Code(r->Right(), rcode);
}
```

【例子2】：在解决某些判定问题时，利用哈夫曼树可以得到最佳判定算法，一个典型的例子就是将学生百分成绩按分数段分级，由于在实际情况下，学生成绩的等级分布并不是均匀的，因此判定过程的表示也可能不同，判定过程可以用图5-23所示的判定树来表示，(a)和(b)分别为不及格(<60) 学生和70～80分学生最多的情况下的判定树。

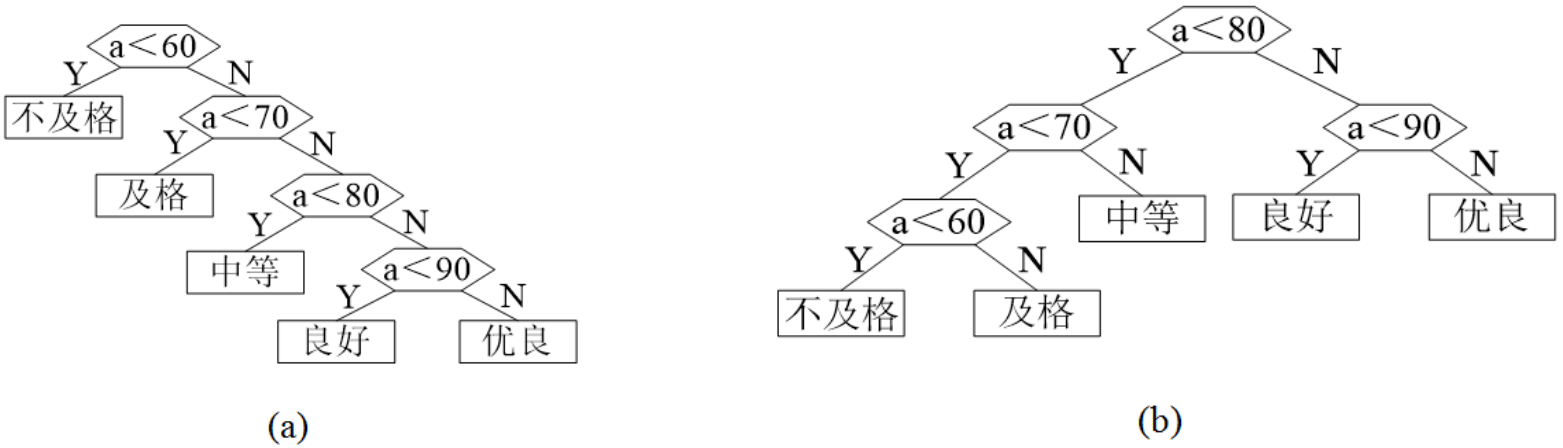


图5-23 判定过程示意图

5.5 二叉树应用2：二叉查找树

二叉查找树是经过一定地组织形成的有特定结构特征的二叉树，支持各种动态集合操作（如插入、删除等）。这些操作的时间复杂度与树的高度成正比。

平均时间复杂度为：$O(\log_2 n)$

最坏情况下(单支树)，最坏时间复杂度将变为：$O(n)$

应用：快速访问数据，建立索引

原理：假设希望存储包含多个域的数据元素，其中的一个域被指定为关键值，搜索就是基于该关键值进行的。

5.5.1二叉查找树的定义

二叉查找树(Binary Search Tree, BST)，又称二叉搜索树。其定义如下：

二叉查找树或者是一棵空树，或者是每个结点具有一个关键值，其**满足以下性质**：

- **(1) 若左子树非空，则左子树上结点的关键字值小于其根结点的值；(left.Key<Key)**
- **(2) 若右子树非空，则右子树上结点的关键字值大于其根结点的值；(right.Key>Key)**
- **(3) 左、右子树均是二叉查找树。**

注意：二叉查找树的定义也是一个递归的定义

图5-24给出了一组对应数值的两颗二叉查找树，

- (a)为(35, 23, 54, 7, 3, 40, 32, 100)；
- (b)为(100, 54, 7, 3, 32, 35, 23, 40)；

两种顺序插入各结点得到的二叉查找树。

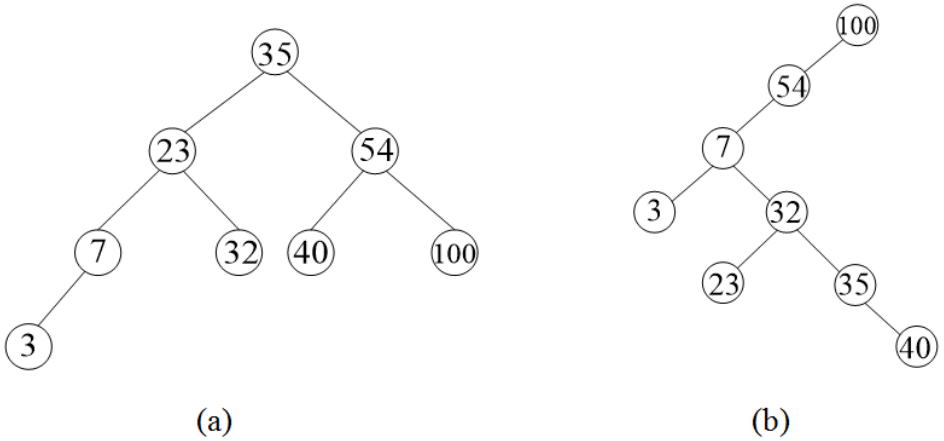


图 5-24 对应同一组数值的两颗不同二叉查找树

- 同一组数值由于插入顺序的不同可以对应多个不同的二叉查找树，而这些二叉查找树的共同特点：

对该树中序遍历会得到一个递增的有序序列。

- 图5-24所示的两棵二叉查找树按照中序遍历得到的有序序列均为：3, 7, 23, 32, 35, 40, 54, 100。

★注：**二叉查找树与二叉排序树的区别：**

二叉查找树中不存在关键值相等的结点，而二叉排序树中允许存在关键值相等的结点，并习惯将相等的结点放在其右子树中。

算法5.16：二叉查找树的声明

```
template <class T>
class BinarySearchTree:public BinaryTree<T>{
public:
    BinarySearchTree(){};
    ~BinarySearchTree(){};
    BinaryTreeNode<T>* Search(BinaryTreeNode<T>* subroot,
     const T  & k);
    void Insert(BinaryTreeNode<T> *&root, const T& k);
    bool Delete(BinaryTreeNode<T> *&root, const T& k);
};
```

5.5.2 二叉查找树的查找

假设查找关键值为k的元素，**二叉查找树的查找方法为：**

①从根结点开始，将给定的值k与根结点的关键值比较。若相等，则查找成功，否则必须查找树的更深一层。

②如果k值小于根结点的值，则只需查找该结点的左子树；

③如果k值大于根结点的值，则只需查找该结点的右子树。

该过程将一直持续到找到k或者遇到叶子结点为止，如果遇到叶子结点仍没有找到k，则k不在该二叉查找树中。

从上述查找方法可知，二叉查找树的高效率表现在其只需要查找两棵子树之一。

图5-25(a)给出了在二叉查找树中查找关键值k等于40的结点的查找路径。

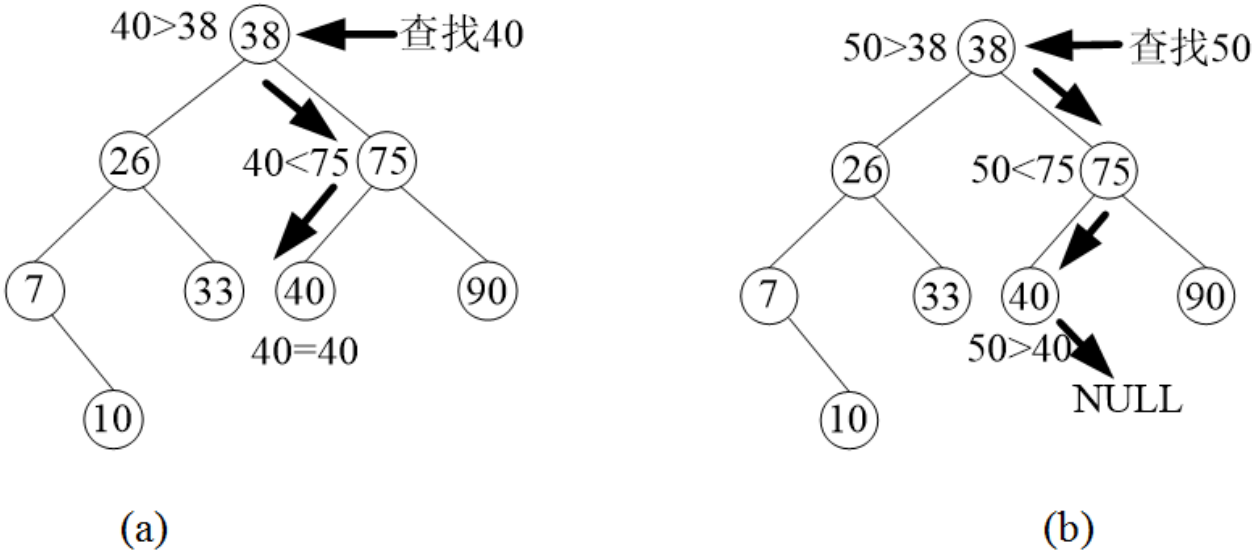


图 5-25 二叉查找树的查找路径

查找过程的算法实现如下：

1）递归算法的实现

算法5.17：二叉查找树递归算法

```
template<class T>
BinaryTreeNode<T>*
    BinarySearchTree<T>::Search(BinaryTreeNode<T>* subroot,
    const T& k){
   if (subroot == NULL) return NULL;
   else if (k > subroot->value())
      return Search(subroot->right, k);     //递归查找右子树
   else if (k < subroot->value())
      return Search(subroot->left, k);      //递归查找左子树
   else  return subroot;                    //相等则查找成功
}
```

2）非递归算法的实现：采用指针

算法5.18：二叉查找树非递归算法

```
template<class T>
BinaryTreeNode<T>* BinarySearchTree<T>::
    Search(BinaryTreeNode<T>* subroot, const T& k){
    BinaryTreeNode<T>* p = subroot;
    while (p != NULL){
        if (k > p->value())                    //查找右子树
            p = p->rightchild();
        else if (k < p->value())               //查找左子树
            p = p->leftchild();
        else return p;                         //查找成功
    }
    return NULL;
}
```

图5-24(a)的平均查找长度ASL为：

$$ASL_a = \frac{1\times1+2\times2+3\times4+4\times1}{8} = \frac{21}{8}$$

图5-24(b)的平均查找长度ASL为：

$$ASL_b = \frac{1\times1+2\times1+3\times1+4\times2+5\times2+6\times1}{8} = \frac{30}{8}$$

结论：

(1) 在BST树中查找次数最长的值 = 树的高度；

(2) 理想情况：二叉查找树比较平衡，查找时的平均查找长度与二叉树的深度有关，复杂度为O(log(n));

(3) 最坏情况：二叉查找树是按照数值递增或递减的顺序依次插入而生成的，是一棵深度等于结点个数n的单支树，其平均查找长度为(n+1)/2，复杂度为O(n)。

5.5.3 二叉查找树的插入

二叉查找树的建立应先从空树开始，要在二叉查找树中插入一个关键值为k的新结点，需定位其在树中的位置，即实现对二叉查找树的搜索，并且必须保证插入后的树仍满足BST的性质。**其插入过程为：**

①若二叉树为空，则生成一个值为k的新结点，并令其为二叉查找树的根；

②若二叉树非空，则将关键值k与根的关键值比较，若相等，则说明树中已有关键值k的结点，无需插入；若k小于根结点的关键值，则将k插入到其左子树中，否则插入到其右子树中。

结论：新插入的结点一定为叶子结点。

插入算法实现如下：

1）递归算法的实现

算法5.19：二叉查找树的插入递归算法

```
template<class T>
void BinarySearchTree<T>::Insert(BinaryTreeNode<T> *&root, const T& k){
   if (root == NULL){
      root = new BinaryTreeNode<T>(k);
      return;
   } else{
      if (k == root->value()) return ; //树中已有关键值为k的结点，无需插入
      else if (k < root->value())
         Insert(root->left, k);
      else  Insert(root->right, k);
   }
}
```

2）非递归算法的实现

算法5.20：二叉查找树的插入非递归算法

```
template<class T>
void BinarySearchTree<T>::Insert(BinaryTreeNode<T> *&root, const T& k){
    if (root == NULL){
        root = new BinaryTreeNode<T>(k);
        return ;
    }
    BinaryTreeNode<T> *p = root, *q, *s;
    q = NULL, s = new BinaryTreeNode<T>(k);
    while (p != NULL){
        q = p;
        if (k > q->value())  p = p->right;
        else  p = p->left;
    }
    if (q == NULL)  { root=s; return ; }
    if (k > q->value())  q->setRightchild(s);
    else  q->setLeftchild(s);
    return ;
}
```

例子：通过二叉查找树的插入可构建二叉查找树，一组关键值(4, 6, 3, 9, 7, 1)，从空树开始，逐个插入结点，生成一棵二叉查找树的过程如图5-26所示。

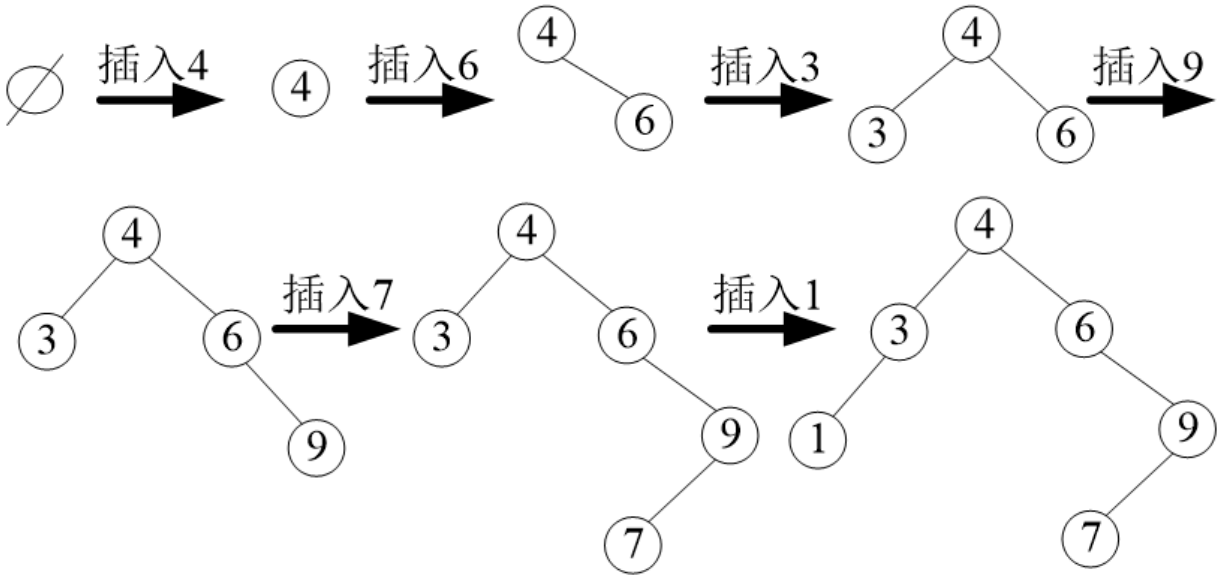


图 5-26 二叉查找树的生成过程

由于同一组关键字按照不同的顺序插入时得到不同的二叉查找树，因此有两种极端情况，如下图5-27所示，(a)插入顺序完全有序，相当于一个线性表，(b)插入顺序部分有序。

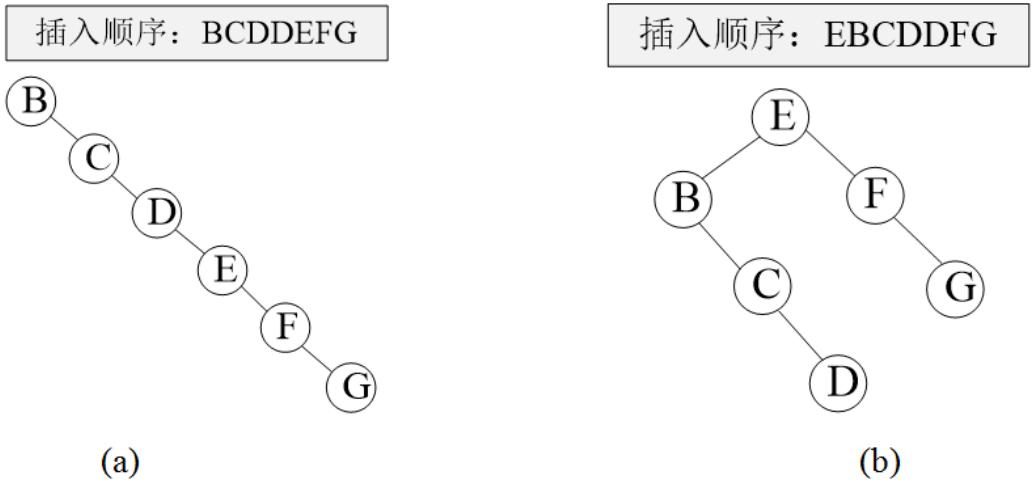


图 5-27 生成二叉查找树时的两种极端情况

结论：插入后形成的二叉查找树的好坏，取决于插入的关键字的顺序。

5.5.4 二叉查找树的删除

(1) 若结点k为叶子结点，由于删去叶子结点不会破坏整棵树的结构，因此只需将k的双亲结点指向它的指针置空，然后删除结点k即可；

如图5-29为图5-28二叉查找树T_8删除关键值为40的结点后得到的二叉查找树。

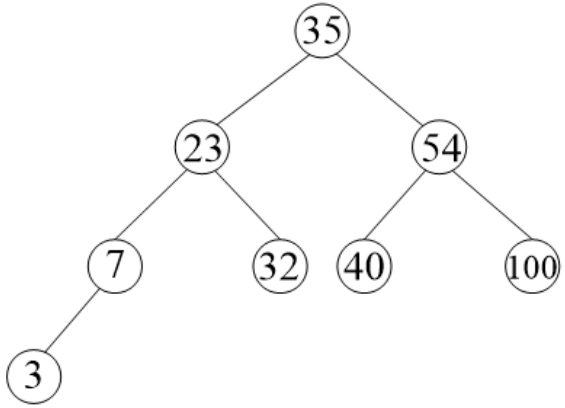


图 5-28 二叉查找树 T_8

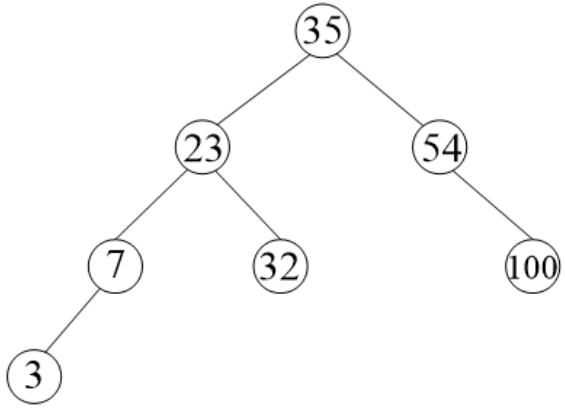


图 5-29 删除结点 40

(2) **若结点k只有左子树（或右子树），**此时可以直接用k的左子树（或右子树）取代k的位置，将k的双亲结点指向k的左子树（或右子树）即可；

如图5-30为图5-28删除关键值为7的结点后得到的二叉查找树。显然该修改也不会破坏二叉查找树的特性。

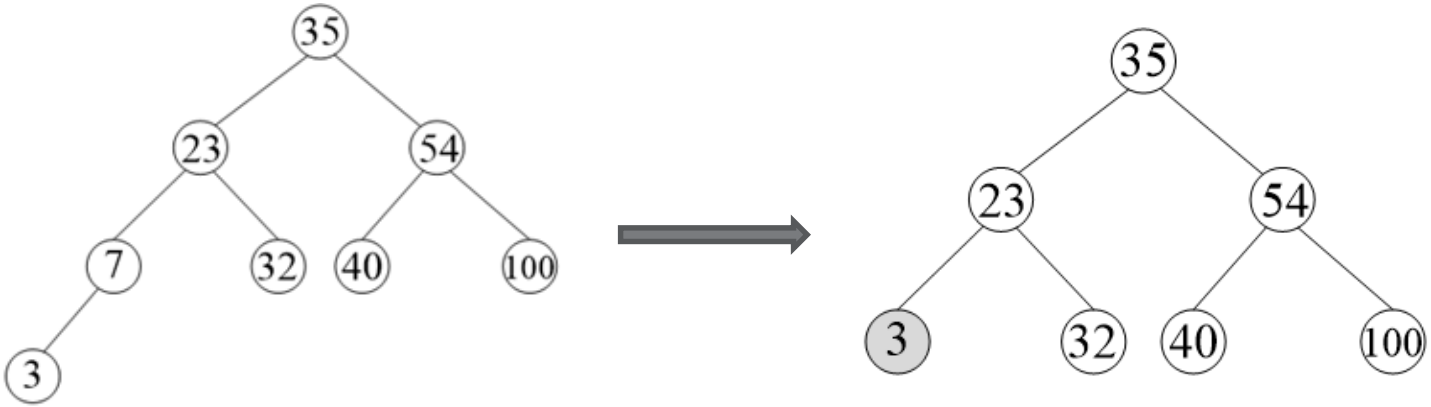


图 5-28 二叉查找树 T_8

图 5-30 删除结点 7

(3)**若结点k的左子树和右子树均不为空**，即K为树的内部结点。为保证二叉查找树的特性，可将该元素替换为中序遍历所得的前驱或后继结点，即它的左子树中的最大元素或右子树中的最小元素。

如要删除图5-28中关键值为35的结点。

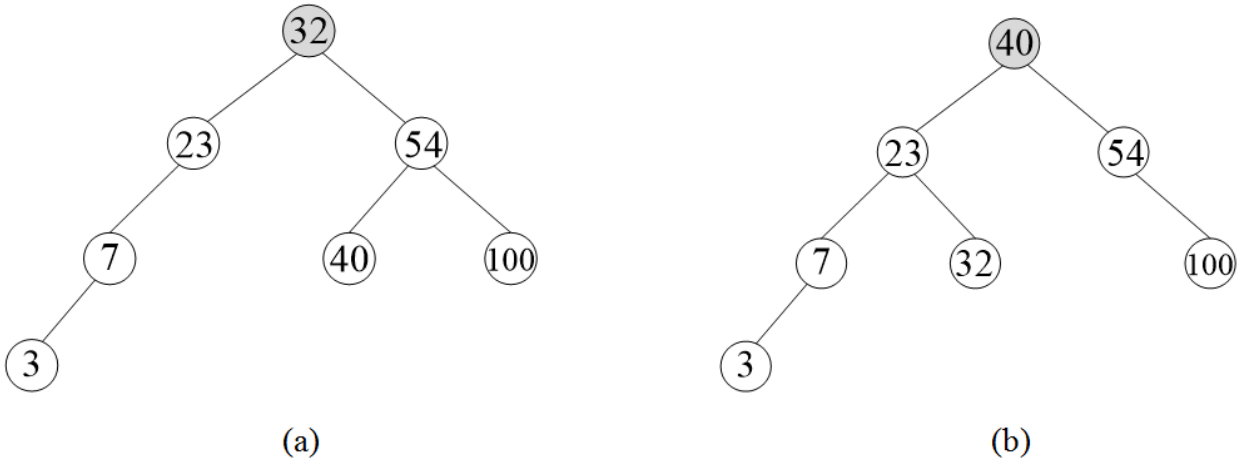


图 5-31 删除结点 35

仔细分析上述三种情况会发现：

- 情况(1)和情况(3)可以统一到情况(2)，
- 情况(1)是情况(2)的特例，
- 情况(3)可以转化为情况(2)。

算法5.21：二叉查找树的删除操作

```
template<class T>
bool BinarySearchTree<T>::Delete(BinaryTreeNode<T> *&root, const T& k){
BinaryTreeNode<T> *pointer = Search(root,k);
   if (pointer == NULL)  return false;        //未找到关键值为k的结点
   BinaryTreeNode<T> *child, *parent = BinaryTree<T>::Parent(pointer);
   BinaryTreeNode<T> *ptr = pointer;      //ptr记录被删除结点pointer
   if (pointer->left && pointer->right){    //若pointer的左右子树均非空
```

```
//找到右子树中的最小值,将情况(3)转化为情况(2)
    parent = pointer;
    pointer = pointer->right;
    while (pointer->left){
        parent = pointer;
        pointer = pointer->left;
    }
}
```

```
//若是情况(2),则child非空；否则child为空
   child = (pointer->left) ? pointer->left : pointer->right;
   if (!parent)  root = child;   //若双亲为空，说明p为根
   else{
      if (parent->left == pointer)  parent->left = child;  //pointer是
       双亲结点的左孩子
      else   parent->right = child;
   }
   if (ptr != pointer)  ptr->data = pointer->data;    //情况(3)
   return true;
}
```

除了替换方法完成删除操作之外，还可采用**截枝方法**实现删除操作。把删除结点的左子树挂到右子树中最小元素后面，成为它的左子树；或把删除结点的右子树挂到左子树中最大元素的后面，使其成为它的右子树，如下图5-32所示。截枝方法可能会大幅改变树的形状，从而影响查找速度。

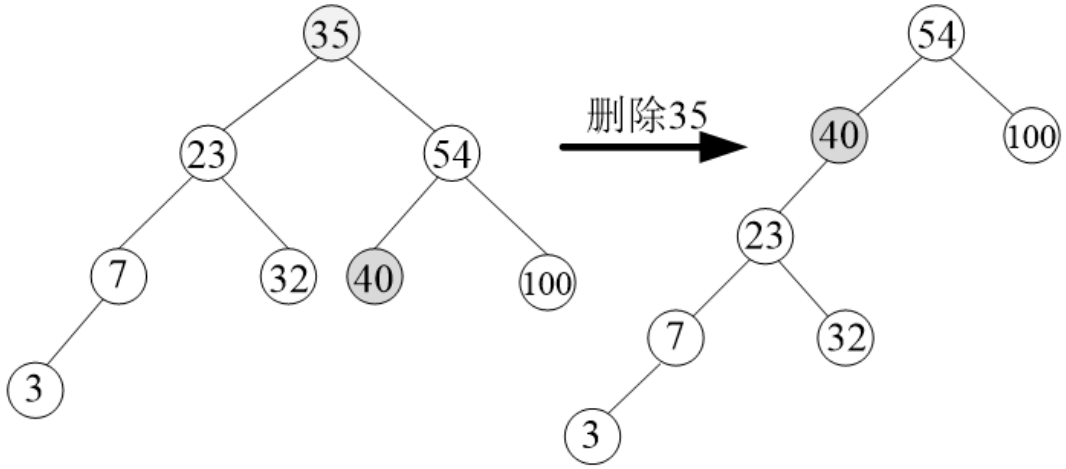


图 5-32 使用截枝方法删除关键值为 35 的结点

算法5.22：截枝方法完成删除操作

```
template<class T>
bool BinarySearchTree<T>::Delete(BinaryTreeNode<T> *&root, const T& k){
    BinaryTreeNode<T> *p, *parent, *ptr, *pointer = Search(root,k);
    if (pointer == NULL)  return false;  //未找到关键值为k的结点
    ptr = pointer; parent = BinaryTree<T>::Parent(pointer); //ptr记住被删除结点
    if (pointer->left == NULL) pointer = pointer->right;
    else  if (pointer->right == NULL) pointer = pointer->left;
          else{       p = pointer->right;
                      while (p->left != NULL) p = p->left;
                      p->left = pointer->left;
                      pointer = pointer->right;
          }
    if (parent == NULL) root = pointer;
    else if (ptr==parent->left) parent->left = pointer;
         else parent->right = pointer;
    delete ptr;
    return true;
}
```

结论：（具有n个结点的BST树）

(1) BST树的查找代价：至少为$\log_2 n$；平均查找代价为$O(\log_2 n)$。

(2) BST树的插入代价：与查找代价相同，平均插入代价为$O(\log_2 n)$。

(3) BST树的删除代价：也同查找代价相同，平均删除代价为$O(\log_2 n)$。

5.6 二叉树应用3：平衡二叉树

5.6.1 平衡二叉树的定义

平衡二叉树(Balanced Binary Tree) 又称为AVL树。

如果将二叉树上的结点的**平衡因子BF**(Balance Factor)定义为该结点的左子树的深度减去它的右子树的深度，BF()表示为：

$$BF(A) = h(A.left()) - h(A.right())$$

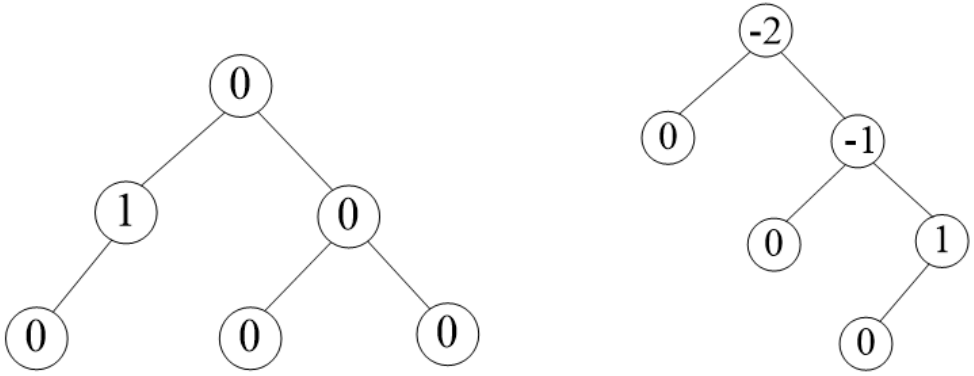
- **平衡二叉树定义**如下：

平衡二叉树或者是一棵空树，或者是满足以下性质的二叉查找树：

(1) 左子树和右子树的深度之差的绝对值不超过1，即$|BF(A)| \leq 1$；

(2) 左子树和右子树都是一棵平衡二叉树，满足递归的定义。

- 平衡二叉树上所有结点的平衡因子只可能是-1、0和1。
- 只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就不是平衡二叉树。



(a) 平衡二叉树　　(b)不平衡二叉树

图 5-33 平衡二叉树与不平衡二叉树及其结点的平衡因子

- 图5-33(b)不是平衡二叉树，结点中的值为结点的平衡因子。
- **结论：** 具有n个结点的AVL树，其深度最大值为1.44log(n)，而通常情况接近于log(n)。
- 平衡二叉树查找的平均时间复杂度为$O(\log_2 n)$。

5.6.2 平衡化旋转

- 一般来说，结点的插入和删除操作可能会影响到平衡二叉树的平衡性。
- 通过对树进行简单的修改来保持树的平衡性，称其为平衡化旋转。
- 平衡化旋转是从离插入或删除结点位置最近的失衡结点的位置开始调整。

一般情况下，假设由于在平衡二叉树上插入结点而失去平衡的最小子树的根结点的指针为A（即A是离插入结点位置最近，且平衡因子绝对值超过1的祖先结点），通常不平衡的状态有两种：

①左高：左子树左高和右子树左高；

②右高：左子树右高和右子树右高。

不平衡可能出现在下面四种情况：

(1) 在A的左孩子的左子树上插入结点；

(2) 在A的右孩子的右子树上插入结点；

(3) 在A的左孩子的右子树上插入结点；

(4) 在A的右孩子的左子树上插入结点。

其中，(1)和(4)为左高情况，(2)和(3)为右高情况。因此，相应的失去平衡后进行调整的规律可以归纳为下面四种情况。

1. LL型平衡旋转法

- LL型是左子树左高的情况，这种情况是由于在结点A的左孩子的左子树上插入结点，结点A的平衡因子由1增至2从而使得以结点A为根的子树失去平衡。
- 此种情况下平衡化旋转的**方法是进行一次顺时针旋转操作。**即将A的左孩子B向右上旋转代替A作为根结点，A向右下旋转成为B的右子树的根结点，而原来B的右子树则变成A的左子树，如图5-34所示。

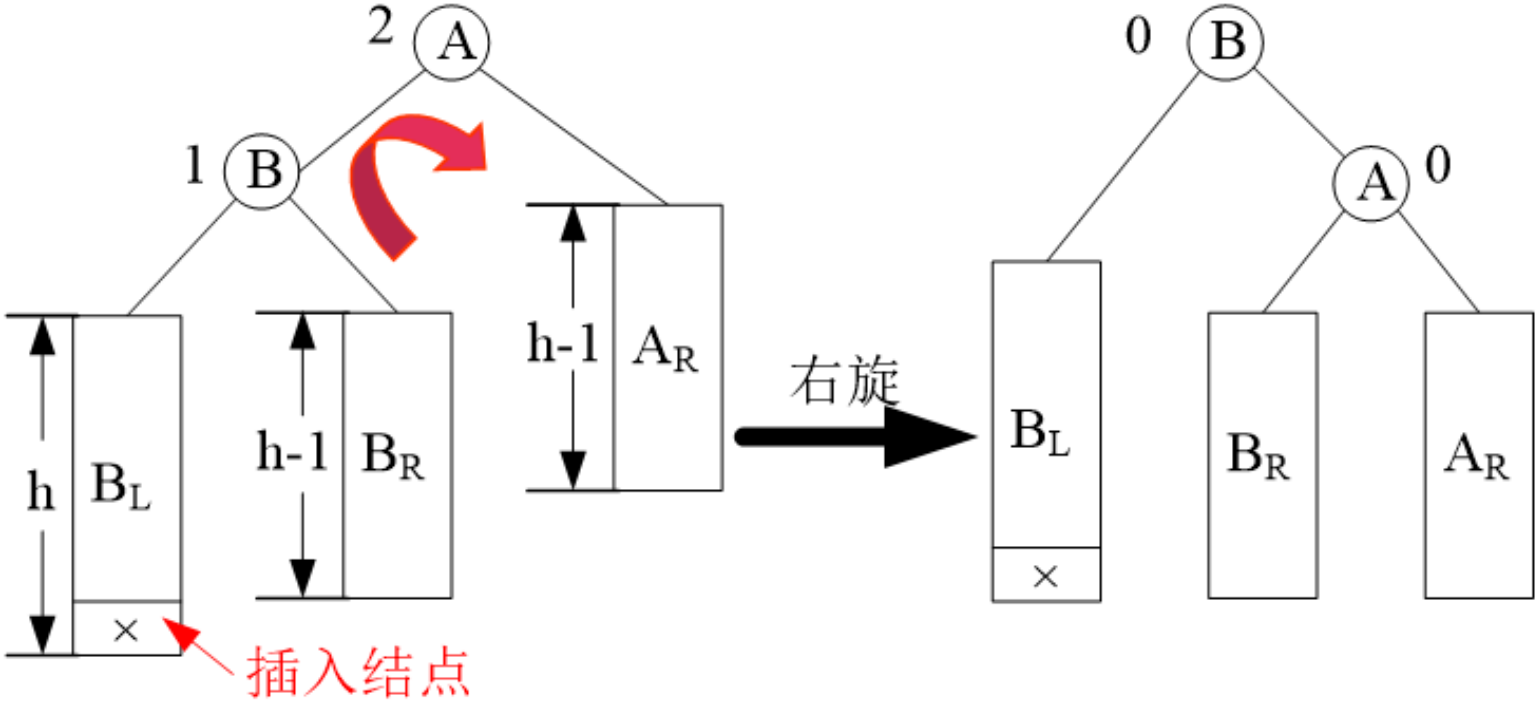


图 5-34 LL 型平衡化旋转示意图

2. RR型平衡旋转法

- 在结点A的右孩子的右子树上插入结点，结点A的平衡因子由-1减至-2，致使结点A失去平衡，因此RR型属于右子树右高。
- 此时RR平衡化旋转的**方法是进行一次逆时针旋转操作。**即将A的右孩子B向左上旋转代替A作为根结点，A向左下旋转成为B的左子树的根结点，而原来B的左子树则变成A的右子树，如图5-35所示。

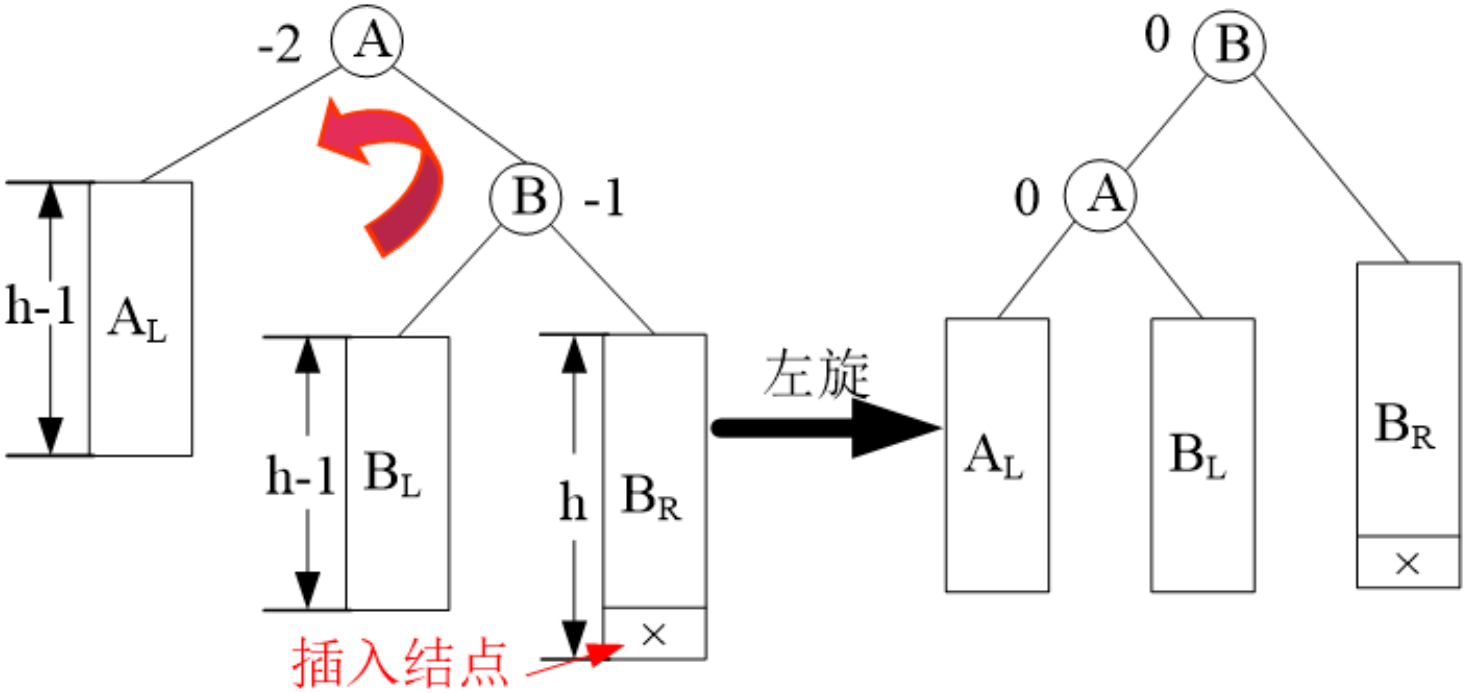


图 5-35 RR 型平衡旋转法示意图

3. LR型平衡旋转法

- LR型为左子树右高，这种情况是由于在结点A的左孩子的右子树上插入结点，使A的平衡因子由1增至2而使以结点A为根的子树失去平衡。
- 设B为A的左孩子，C为B的右孩子。
- 在插入结点前C的平衡因子只能是0，否则C会失去平衡或者C的深度不发生变化。而插入后C的平衡因子的**变化有下列三种情况：**

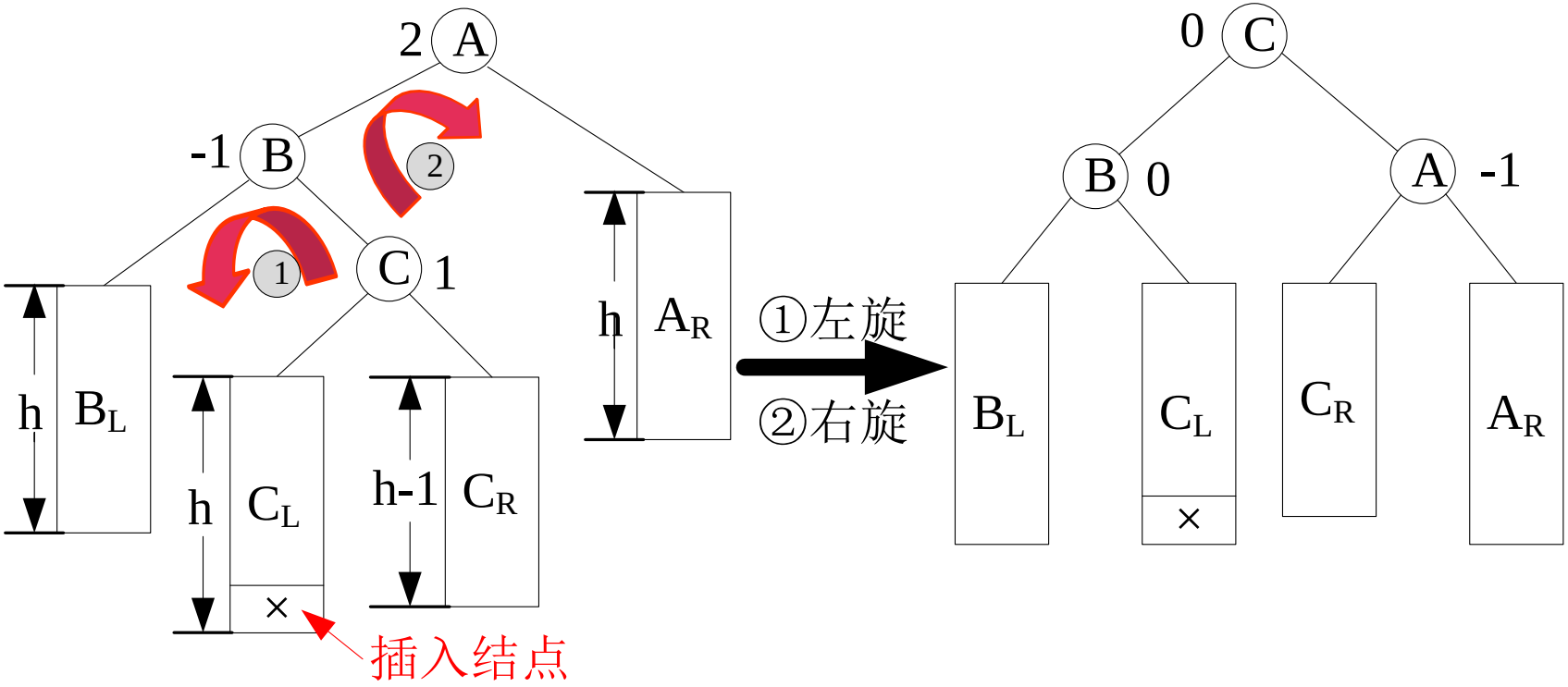
(1) 插入后C的平衡因子为1，即在C的左子树上插入。设此时C的左子树的深度为h，则C的右子树的深度为h-1。由于B的平衡因子是-1，故B的左子树的深度为h，，由于A的平衡因子为2，故A的右子树的深度为h。

(2) 插入后C的平衡因子为-1，即在C的右子树上插入。设此时C的左子树的深度为h，则C的右子树的深度为h+1。由于B的平衡因子是-1，故B的左子树的深度为h+1，，由于A的平衡因子为2，故A的右子树的深度为h+1。

(3) 插入后C的平衡因子为0，即C本身就是插入结点。设此时C的左子树的深度为h，则C的右子树的深度为h。由于B的平衡因子是-1，故B的左子树的深度为h，，由于A的平衡因子为2，故A的右子树的深度为h。

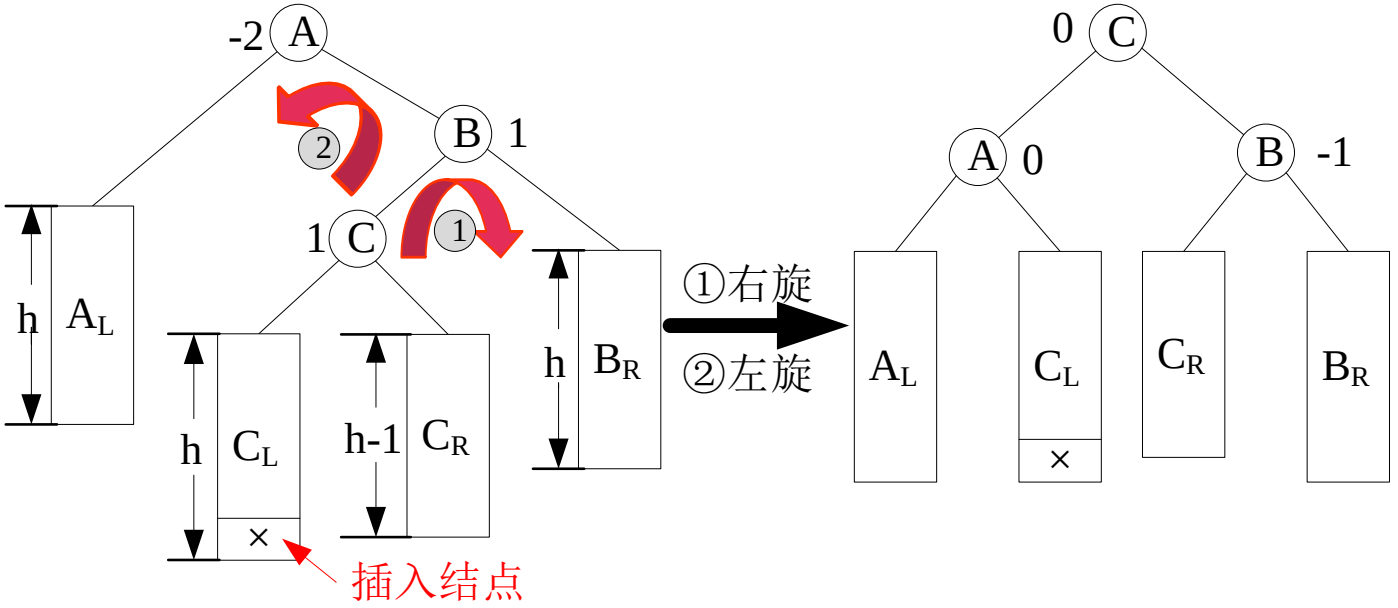
•此时**LR型平衡化旋转的方法：**先对左孩子进行一次逆时针左旋转;再对失衡点进行一次顺时针的右旋转。

•即先将A的左孩子B和B的右孩子C向旋转①;再对失衡点A和新的左孩子C向右旋转②。如图所示。



4. RL型平衡旋转法

RL型为右子树左高情况，这种情况是由于在结点A的右孩子的左子树上插入结点，使A的平衡因子由-1增至-2而使以结点A为根的子树失去平衡。这种情况与LR型旋转情况是对称的。



下面依据前面三种情况来分析一下旋转后各个结点的平衡因子的情况：

(1) 若旋转前C的平衡因子为1，则旋转后A的左子树没有变化，右子树是C原来的左子树，故A的平衡因子为0；B的右子树没有变化，左子树变为C原来的右子树，故B的平衡因子为-1；C的左右子树分别是以A和B为根的子树，因为A子树和B子树的深度均为h+1，故C的平衡因子为0。

(2) 若旋转前C的平衡因子为-1，同理，可以求出旋转后A的平衡因子为1，B的平衡因子为0，C的平衡因子为0。

(3) 若旋转前C的平衡因子为0，同理，根据前面求出的各子树的深度可以求出旋转后A、B、C的平衡因子均为0。

结论：四种不平衡状态下每种状态的平衡因子及平衡化旋转处理情况总结如下：

不平衡状态	平衡因子	平衡化旋转
LL	BF(A)=2 BF(A->left)=1	右旋
RR	BF(A)=-2 BF(A->right)=-1	左旋
LR	BF(A)=2 BF(A->left)=-1	1.在A->left处左旋 2.在A处右旋
RL	BF(A)=-2 BF(A->right)=1	1.在A->right处右旋 2.在A处左旋

5.6.3 平衡二叉树的插入

如图5-38所示，按照关键值序列(3, 10, 24, 65, 32)依次插入结点构造平衡二叉树。

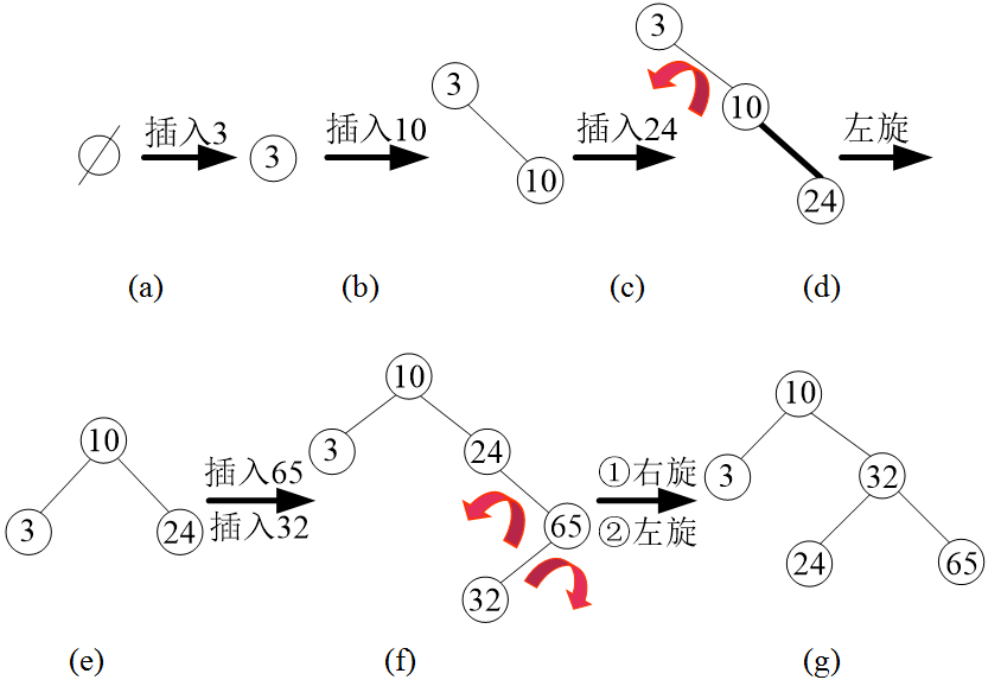


图 5-38 平衡二叉树的生成

在平衡二叉树上插入结点的大致步骤如下：

(1) 按照二叉查找树的插入方法，找到插入新元素的位置。

(2) 修改从新插入结点到根结点路径上的所有祖先结点的平衡因子。

(3) 判断是否产生不平衡，若A结点不平衡，则确定A的不平衡类型并执行相应的旋转处理，根据旋转需要修改相应的平衡因子。然后，插入过程结束。

(4) 若逆向到根结点均未发现不平衡，则插入结束。

5.6.4 平衡二叉树的删除

基本思想：

(1) 使用二叉查找树的删除操作**找到要删除的结点并删除**；

(2) 沿被删除结点的双亲结点到根结点的路径逐层向上回溯，并**追踪平衡因子的变化**；

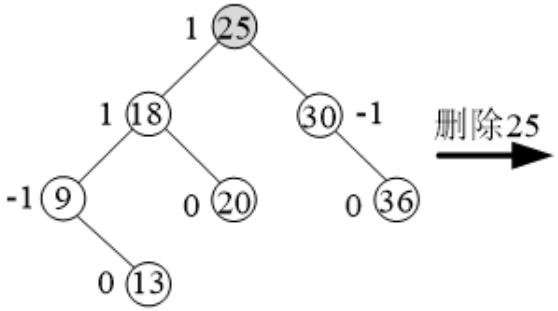
(3) 回溯过程中，一旦发现被删除结点的某个祖先失衡，就要应用平衡化旋转以恢复平衡性。即**左子树删除要调整右子树，右子树删除要调整左子树**。

(4) 继续逆向到根调整平衡因子，**若未发现失衡结点，则结束删除算法。**

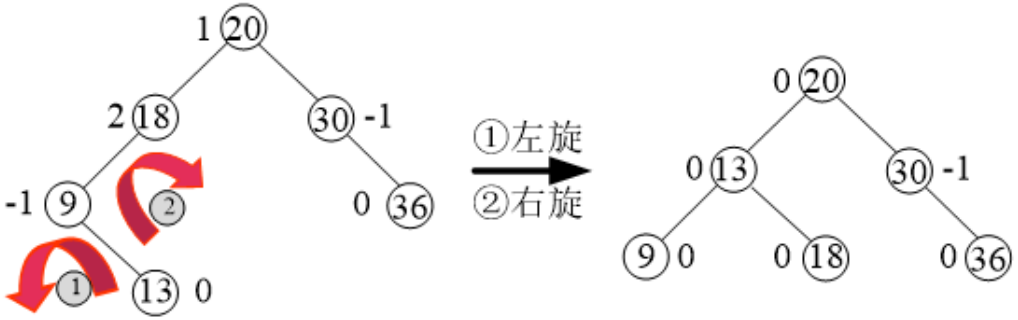
结论：

在平衡二叉树上删除一个结点后，为了恢复平衡性可能需要多次平衡化旋转处理。

图5-39为在图5-39(a)所示平衡二叉树上删除关键值为25的结点的过程：



(a)



(b) (c)

图 5-39 平衡二叉树的删除

从左子树删除和右子树删除是对称的，设B为A的右孩子，以从结点A的**左子树删除结点导致其左子树高度降低**为例，分为以下**三种情况进行分析：**

1. A的右子树右高情况，即A的右子树的平衡因子为-1。

在结点A的左子树上删除结点后，A的左子树的高度降低，结点A的平衡因子由-1变为-2，从而使得以结点A为根的子树失去平衡。这种情况下只需要进行一次逆时针旋转即可，旋转后结点A和B的平衡因子均变为0，其他结点的平衡因子不变，如图5-40所示。

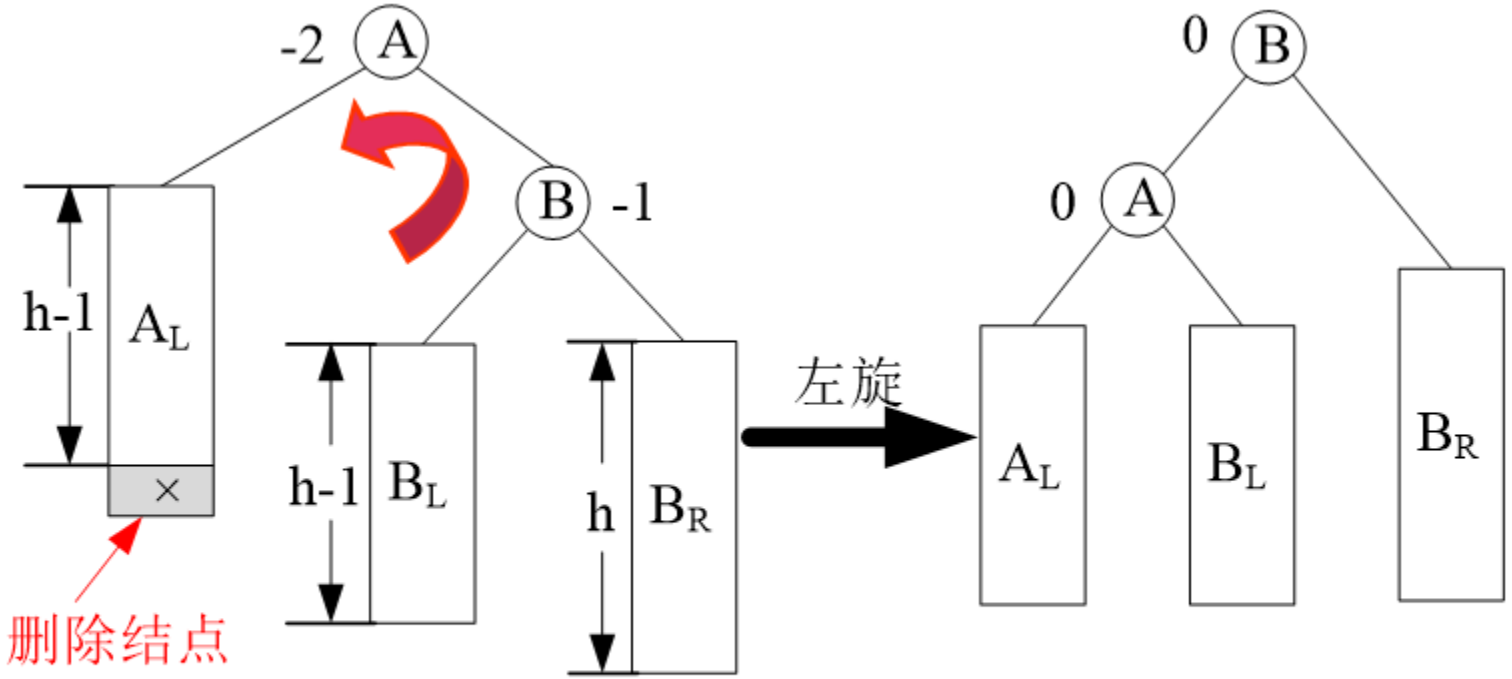


图 5-40 平衡二叉树的删除：右子树右高情况

2. A的右子树等高情况，即A的右子树的平衡因子为0。

在结点A的左子树上删除结点后，A的左子树的高度降低，结点A的平衡因子由-1变为-2，以结点A为根的子树失去平衡。这种情况与右子树右高情况类似，只需进行一次逆时针旋转即可，旋转后结点A的平衡因子变为-1，其右孩子B的平衡因子变为1，其他结点的平衡因子不变，如图5-41所示。

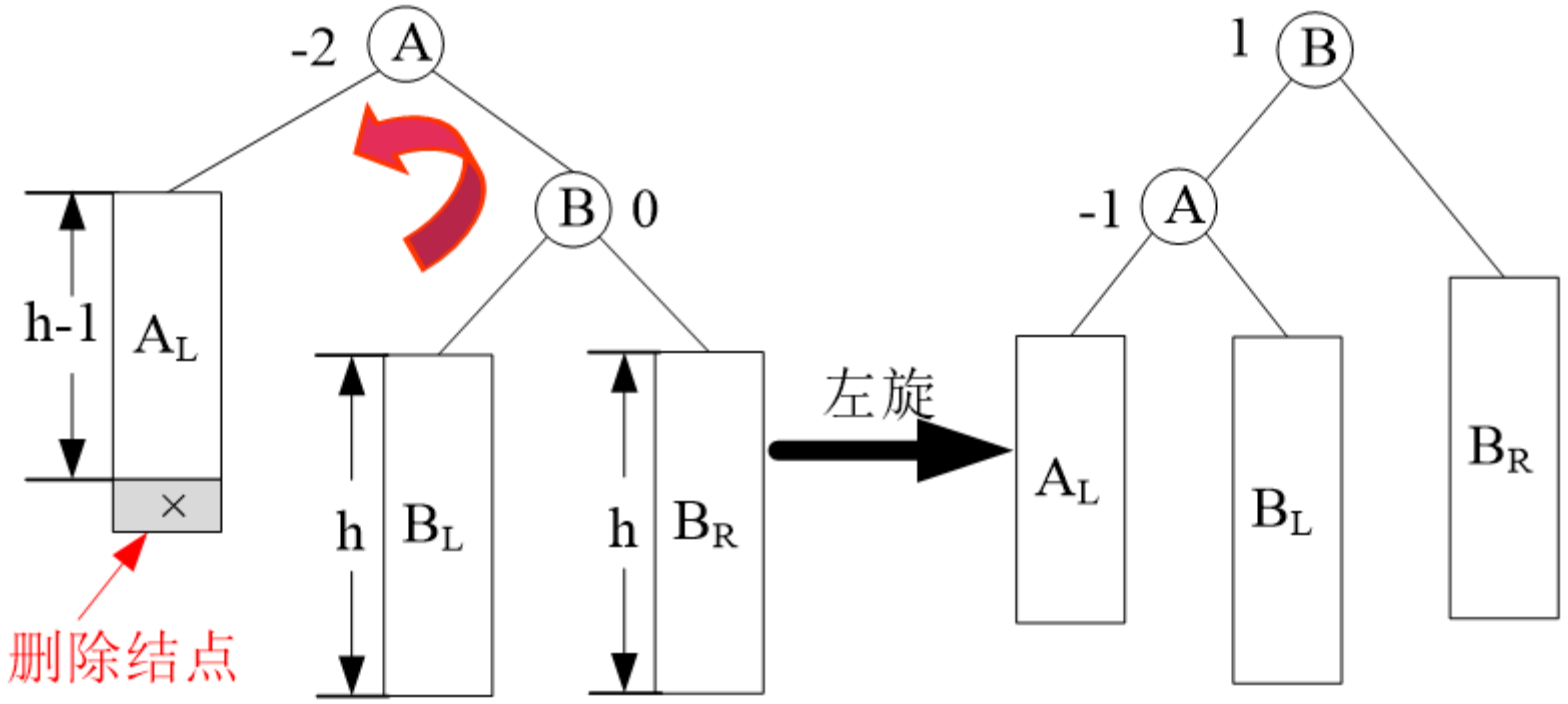


图 5-41 平衡二叉树的删除：右子树等高情况

3. A的右子树左高情况，即A的右子树的平衡因子为1。

在结点A的左子树上删除结点后，A的左子树的高度降低，结点A的平衡因子由-1变为-2，以结点A为根的子树失去平衡。这种情况需要做两次旋转，先将以B为根的子树顺时针旋转为以C为根，再以A进行一次逆时针旋转。需要注意的是，这种情况下B的左孩子C有三种可能的平衡因子：1，-1和0。

当C的平衡因子为1时，经过旋转处理后结点A的平衡因子变为0，其右孩子B的平衡因子变为-1，C的平衡因子变为0，如图5-42所示。

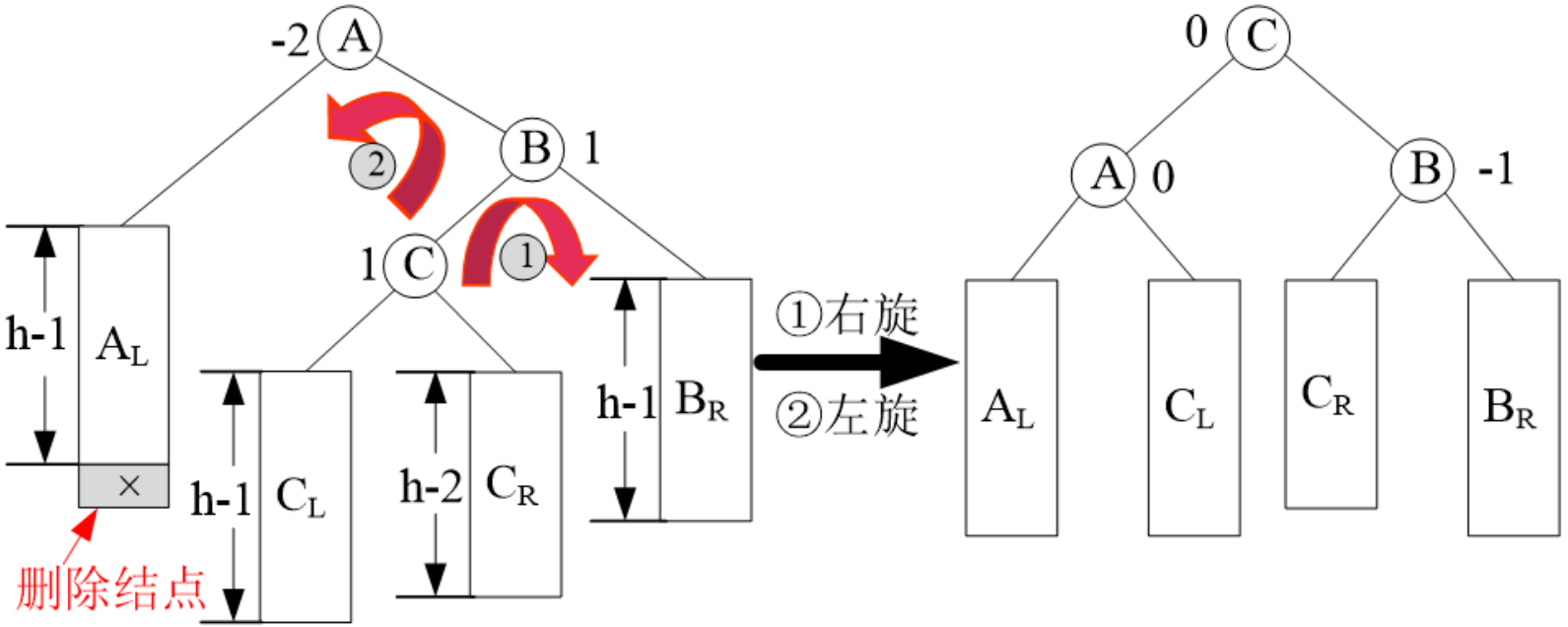


图 5-42 平衡二叉树的删除：右子树左高情况（BF(C) = 1）

当C的平衡因子为-1时，经过旋转处理后结点A的平衡因子变为1，其右孩子B的平衡因子变为0，C的平衡因子变为0，如图5-43所示。

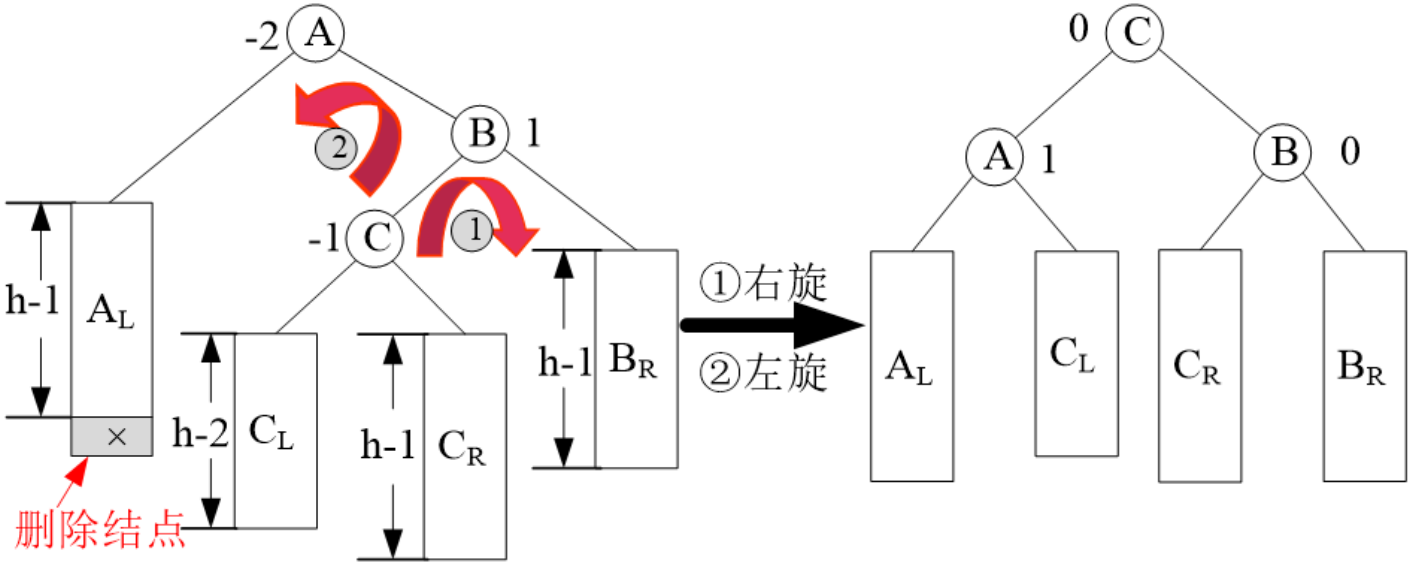


图 5-43 平衡二叉树的删除：右子树左高情况（BF(C) = -1）

当C的平衡因子为0时，经过旋转处理后结点A、B和C的平衡因子都变为0，如图5-44所示。

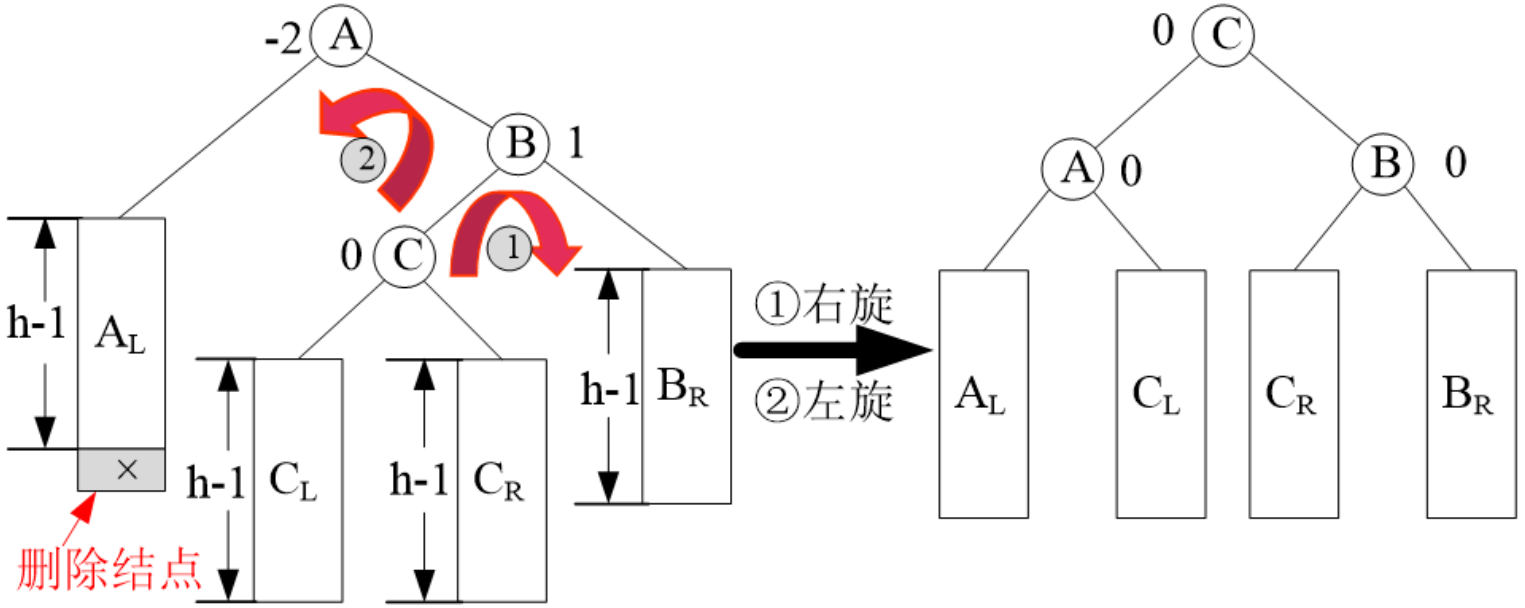


图 5-44 平衡二叉树的删除：右子树左高情况（BF(C) = 0）

5.7 二叉树应用4：堆与优先队列

5.7.1 堆与优先队列的定义与实现

假定每个对象都包含一个关键值，称为对象的优先级，按照优先级或重要性来组织的对象被称为**优先队列**(Priority Queue)。

优先队列可以通过几种已经介绍过的数据结构实现：

(1) 使用有序链表，插入时找到合适位置，时间代价为O(n)；删除元素时代价为O(1)。

(2) 使用简单链表，在表头进行插入，时间代价为O(1)；删除元素时找到最小值需遍历该链表，需O(n)花费时间。

(3) 使用二叉查找树，平均情况下插入和删除操作的时间代价均为O(logn)。但是由于插入和删除操作可能导致二叉查找树失去平衡，这将导致二叉查找树的性能变得很差。

优先队列的存储实现---堆：

为了保证较高的操作效率，提出一种新的数据结构——**堆(Heap)**。

堆的定义如下：

(1) 堆是一棵完全二叉树；

(2) 在一个堆中，对于任意一个非终结点A，A的关键值大于等于（或小于等于）其任意一个孩子结点的关键值。

注：堆分为大顶堆和小顶堆。

有两种不同的堆：大顶堆(Max-Heap)和小顶堆(Min-Heap)。

- **大顶堆**中的每一个非终结点A的关键值都大于或等于其任意一个子结点的关键值：

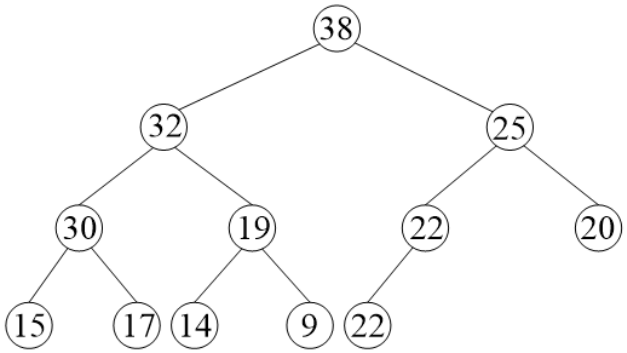
$$\begin{cases} A.key \geq A.leftchild().key \\ A.key \geq A.rightchild().key \end{cases}$$

- **小顶堆**中的每一个非终结点A的关键值都小于或等于其任意一个子结点的关键值：

$$\begin{cases} A.key \leq A.leftchild().key \\ A.key \leq A.rightchild().key \end{cases}$$

堆是一棵完全二叉树，所以可以直接用一个数组来表示，不需要使用指针。如图5-45(b)为大顶堆5-45(a)的数组实现。

堆中元素满足完全二叉树的性质5。如果一个结点在数组中的位置为k，则其左孩子的位置为2k+1，其右孩子的位置为2k+2，而其双亲结点的位置为⌊(k-1)/2⌋。



(a) 大顶堆

数组下标：	0	1	2	3	4	5	6	7	8	9	10	11
	38	32	25	30	19	22	20	15	17	14	9	22

(b) 大顶堆的数组实现

图 5-45 大顶堆及其数组实现

算法5.23：大顶堆的类定义

```
template <class T>
class MaxHeap{
private:
    T* Heap;        //堆的顺序存储区
    int n;          //堆中元素的数目
    int maxsize;    //堆能包含的最大元素数目
public:
    MaxHeap(T * &h, int num, int max){
        Heap = h; n = num; maxsize = max;
        BuildHeap();
    }
    ~MaxHeap(){};
```

```
    int heapsize();                          //返回堆中元素个数
    bool isLeaf(int pos);                    //如果是叶子结点，返回TRUE
    int leftchild(int pos);                  //返回左孩子位置
    int rightchild(int pos);                 //返回右孩子位置
    int parent(int pos);                     //返回双亲结点位置
    void BuildHeap();                        //建堆
    void SiftDown(int pos);                  //筛选法调整
    bool Insert(const T& newKey);            //向堆中插入一个新元素
    T& RemoveMax();                          //删除关键值最大的元素
    bool Remove(int pos, T& node);//删除给定下标的元素
};
```

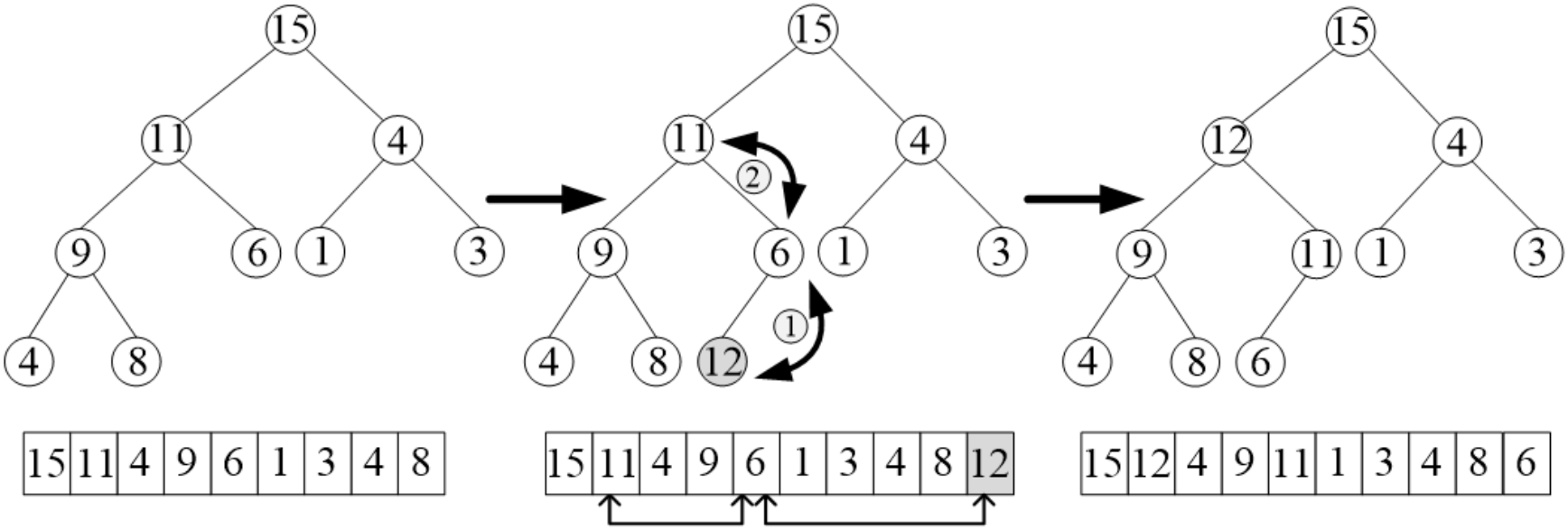
5.7.2 堆的插入和堆顶删除

在有n个元素的堆中**插入一个新元素需要两步**：

(1) 将该元素插入到数组的末尾位置n处；

(2) 插入之后该元素很可能不在正确的位置，需要与其双亲结点进行比较，并将该元素移动到正确的位置。

- 对于一个大顶堆，如果该元素的关键值小于或等于其双亲结点的关键值，说明此时该元素已经位于正确的位置，插入成功；如果该元素的关键值大于其双亲结点的关键值，则交换两个元素的位置，持续上述过程直到该元素位于正确位置。
- 如图5-46所示，在5-46(a)中的大顶堆中插入关键值为12的元素



(a) 大顶堆　　(b) 插入 12　　(c) 交换后的大顶堆

图 5-46 堆的插入

向大顶堆中插入一个新元素的算法如下：

算法5.24：向大顶堆中插入一个新元素

```
template<class T>
bool MaxHeap<T>::Insert(const T& newKey){
    if (n == maxsize) return false;          //堆已满
    int curr = n++, p = parent(curr);
    Heap[curr] = newKey;
    while (curr != 0 && (Heap[p] < newKey)){
        swap(Heap[p], Heap[curr]);      //curr与其双亲结点交换
        curr = p;
        p = parent(curr);
    }
    return true;
}
```

- 实现向堆中插入一个新元素的操作之后，就可以按照把元素一个接一个插入堆中的方法来建堆。
- 但是最坏情况下，插入一个新元素的值可能要从树的最底层一直交换到顶端，这种插入的时间代价为O(logn)。插入n个元素的时间代价就为O(nlogn)。
- 如果在建堆时全部n个关键值都已知，可以使用筛选法(Sifting)更高效的建堆。

对于一个给定的结点R，如果它的左子树和右子树都已经是堆，则可以通过将结点R中的关键值向下筛选的方法将以R为根的树调整为堆。

如图5-47所示，对于结点3，其左右子树均已成大顶堆，只需将结点3通过3次向下筛选，即可将初始根结点为3的树调整为大顶堆。

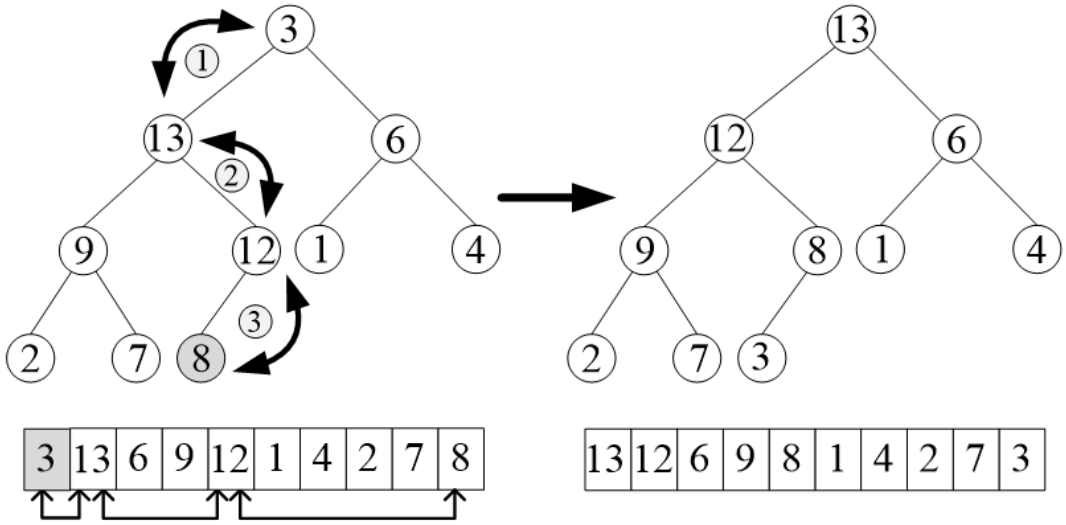


图 5-47 筛选法调整堆

筛选法建堆的步骤：

(1) 将所有n个已知的关键值保存到数组中，此时形成也许不满足堆特性的完全二叉树；

(2) 从最后一个内部结点（该结点位于完全二叉树的倒数第二层，在数组中的位置为⌊n/2⌋-1）开始，用筛选法从右至左从下到上依次调整每个内部节点，直到到达根结点，整棵完全二叉树就成为一个堆。

为给定一组关键值{14, 16, 21, 18, 30, 35}建大顶堆的过程如图5-48所示。

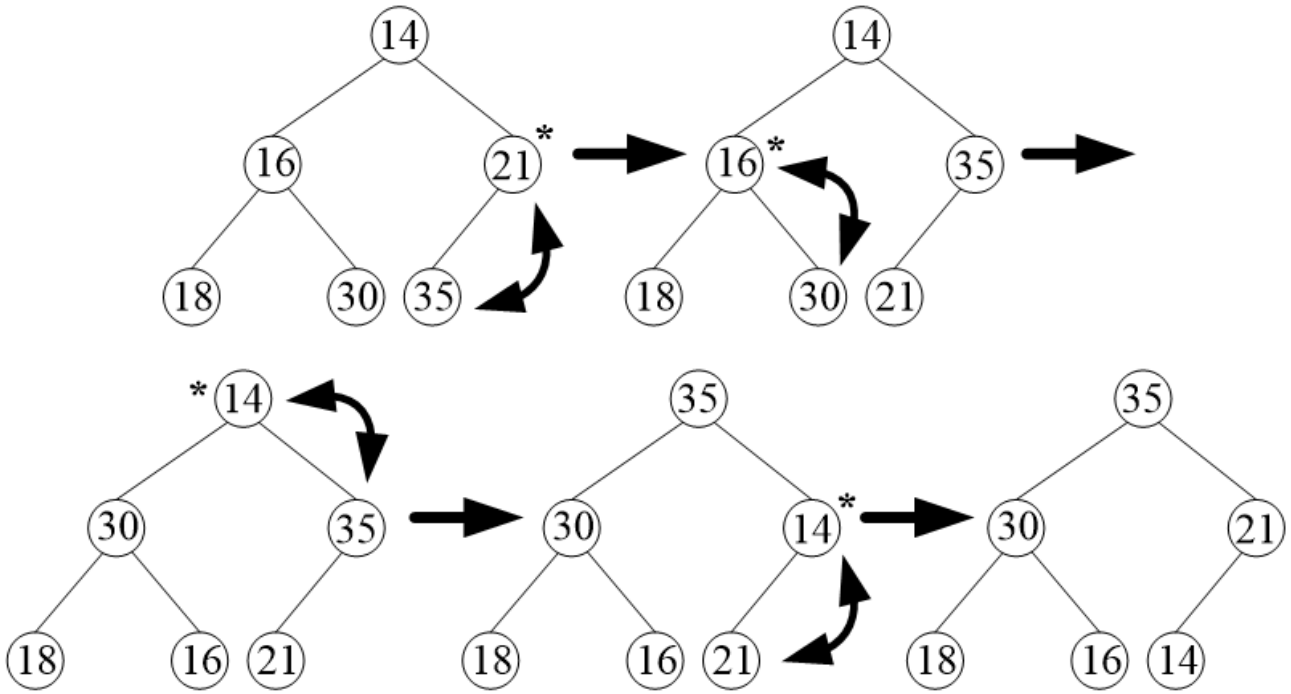


图 5-48 筛选法建堆

筛选法建堆的算法实现如下：

算法5.25：筛选法建堆

```
template<class T>
void MaxHeap<T>::BuildHeap(){
    for (int i = n/2-1; i >= 0; i--)
        SiftDown(i);
}
```

算法5.26：筛选法调整

```
template<class T>
void MaxHeap<T>::SiftDown(int pos){
    if (pos < 0 || pos >= n) cout << "Heap is null or full!" << endl;
    T temp = Heap[pos];
    while (!isLeaf(pos)){
        int lc = leftchild(pos);
        if ((lc < n-1) && Heap[lc] < Heap[lc+1])  lc++;
        if (temp >= Heap[lc]) break;
        Heap[pos] = Heap[lc];
        pos = lc;
    }
    Heap[pos] = temp;
}
```

具有n个结点的完全二叉树的深度为，筛选法调整堆的算法SiftDown的最大代价是结点向下移至树的最底层（即叶子结点层）的层数，因此最坏情况下代价为O(logn)，其中有2log(n)次比较和log(n)次移动。

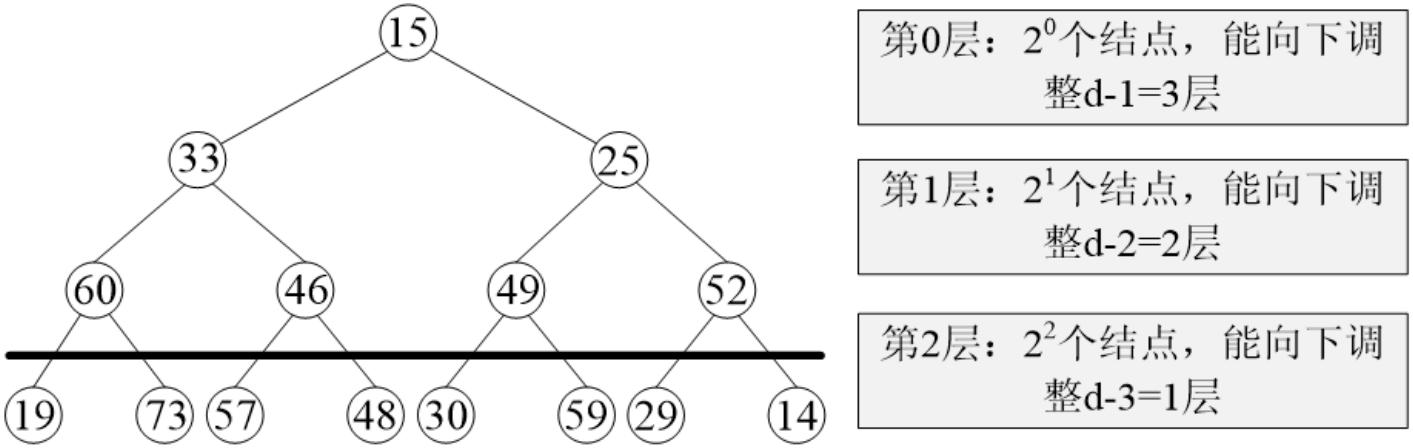


图 5-49 满二叉树及结点最多需调整层数

- 堆还有一个基本操作——删除堆的根结点，即删除堆中最大（或最小）的关键值结点。
- 删除结点后堆的特性可以通过筛选法调整算法SiftDown来保持。(**用最后一个元素替代对顶元素**)
- 如图5-50所示为删除大顶堆中的最大关键值38的过程。

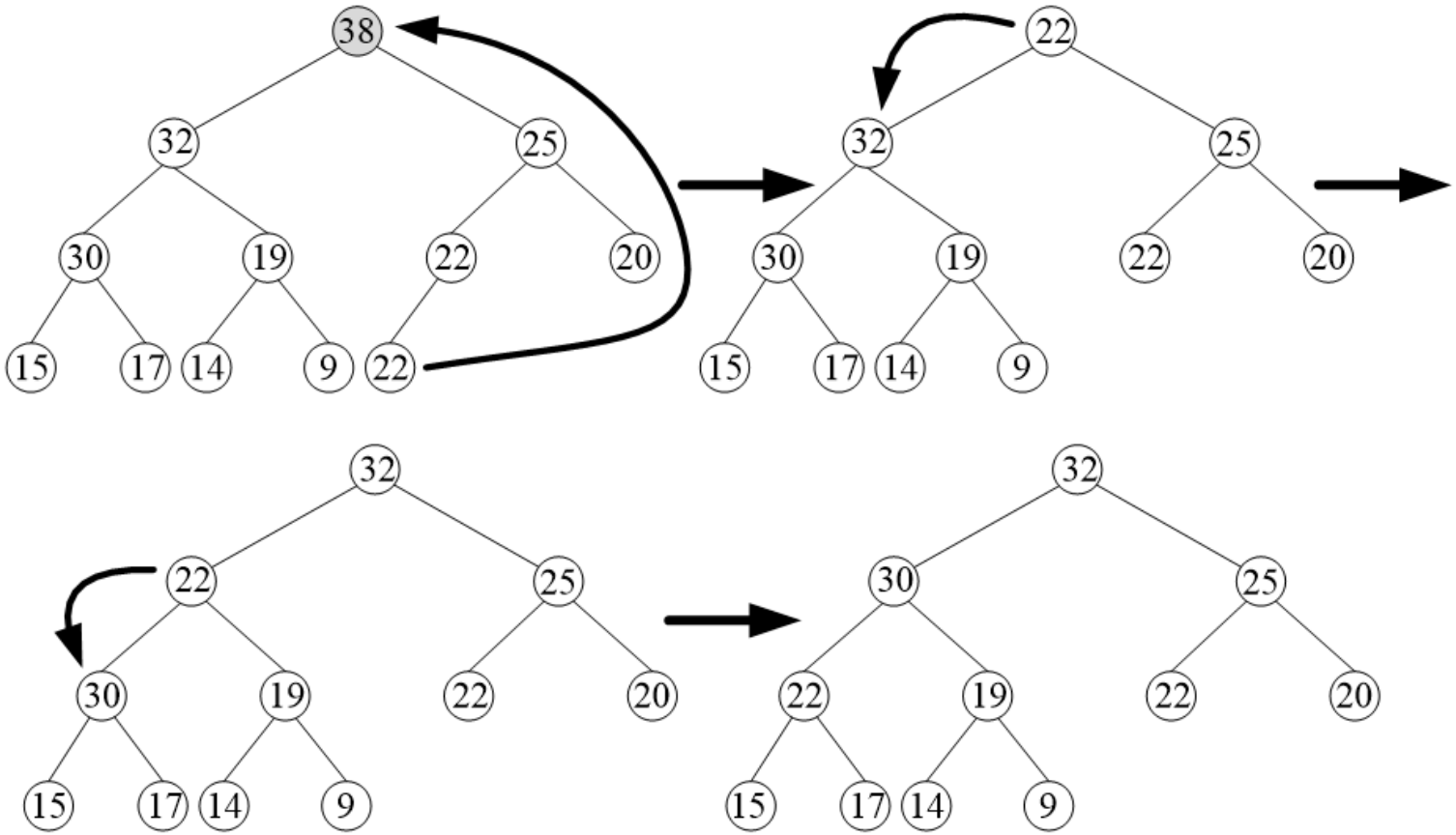


图 5-50 删除大顶堆最大关键值及调整过程

删除大顶堆中最大关键值元素的算法实现如下：

算法5.27 删除大顶堆中最大关键值元素的算法

```
template<class T>
T& MaxHeap<T>::RemoveMax(){
    if (n == 0)          cout << "Removing from empty Heap!" << endl;
    T temp = Heap[0];
    if (--n != 0){
            Heap[0] = Heap[n];
            SiftDown(0);
    }
    return temp;
}
```

注意：在一个有n个结点的堆中，删除根结点后需要向下调整层，故删除最大关键值元素的操作的代价为O(logn)。

5.8 树与森林

5.8.1 树的存储结构

1. 双亲表示法

在树中，除了根结点没有双亲结点外，其余的结点都有唯一的一个双亲结点。在每个结点中附设一个指向其双亲结点的指针来唯一地表示一棵树，其结点结构图如图5-51所示。

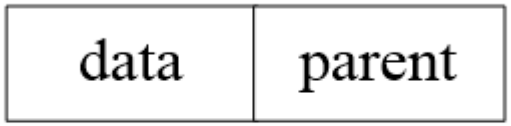
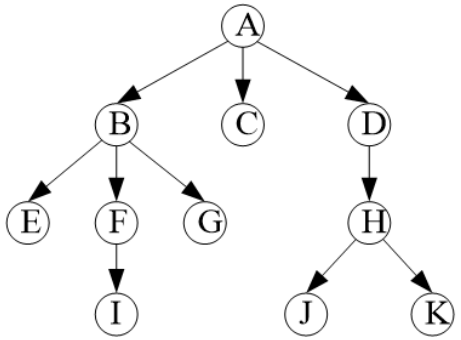


图 5-51 双亲表示法的结点结构图

如图5-52(a)所示树T_9的存储结构如图5-52(b)所示。为简明起见，将父指针表示为双亲结点在数组中位置的下标，注意根结点A无双亲，设其parent的值为-1。



(a) 树 T_9

0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	1
7	H	3
8	I	5
9	J	7
10	K	7

(b) 树的双亲表示法的存储结构

图 5-52 树的双亲表示法的存储结构

双亲表示法是实现树的最简单方法，使用这种方法在寻找树中某一结点的双亲时只需要O(1)时间

2. 孩子表示法

树中的每个结点可能有多个孩子，所以可以对树中的每个结点用一个包含结点本身数据的数据域和多个指向该结点孩子的指针域来表示一棵树，各指针域反映了树中结点与结点之间的关系。其结点结构图如图5-53所示，其中n为树的度。

data	child 1	child 2	…	child n

图 5-53 孩子表示法的结点结构图

图5-54为图5-52(a)所示的树的孩子表示法的结构图。

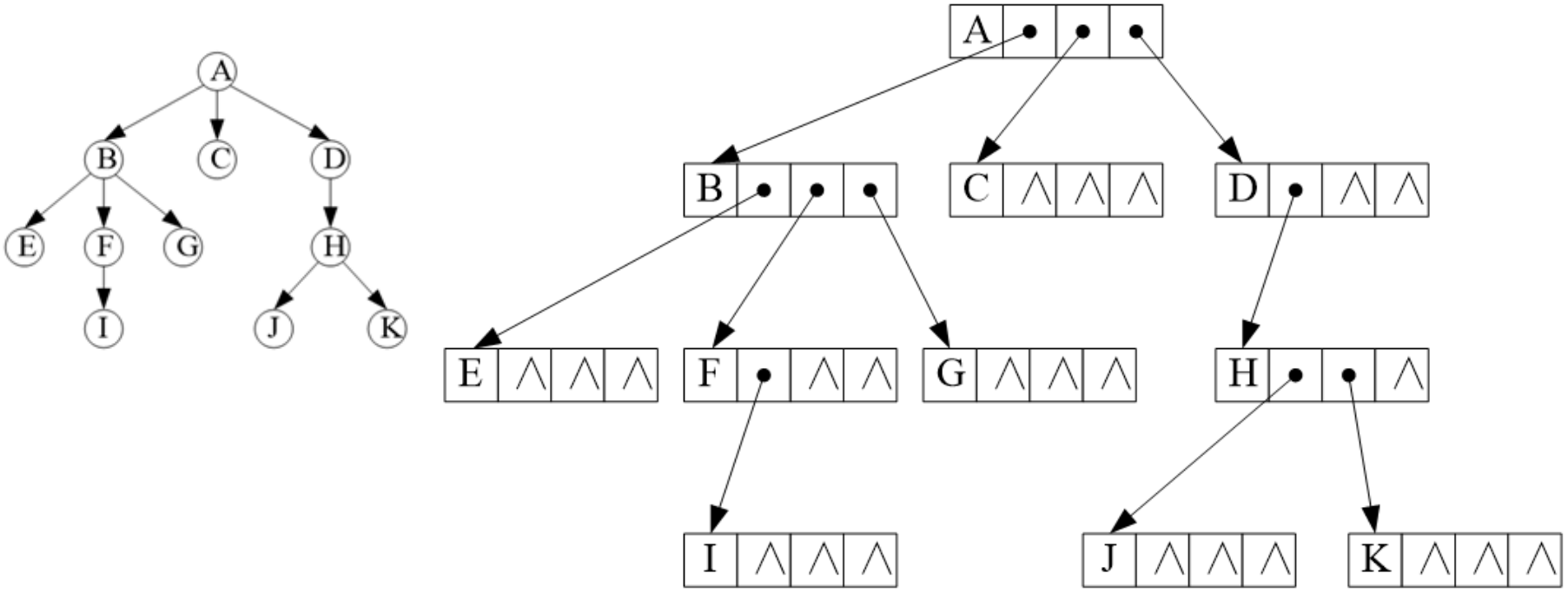


图 5-54 树的孩子表示法的存储结构图

比较好的方法是将每个结点的孩子结点按照从左到右的顺序以单链表的形式存储起来，每个结点包含一个结点数据域以及一个指向子结点链表的指针。图5-55为图5-52(a)所示的树的孩子表示法的链表存储结构。

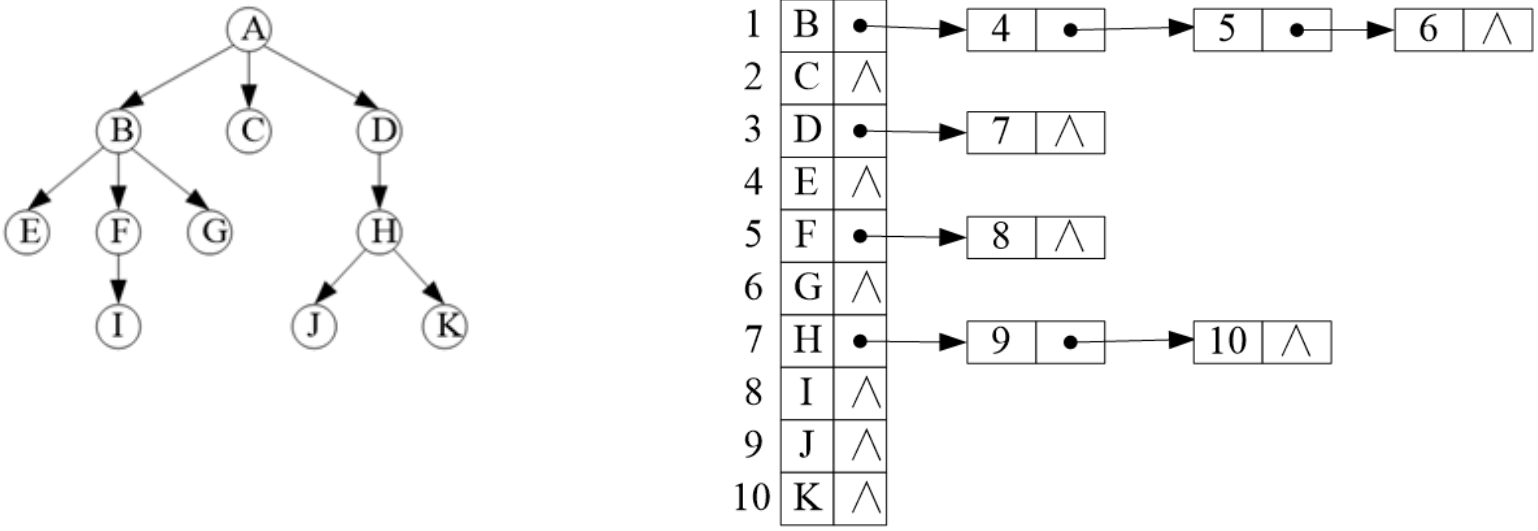


图 5-55 树的孩子表示法的链表存储结构

与双亲表示法相反，孩子表示法在实现涉及孩子结点及其子孙的操作中较为方便，但不便于实现与双亲有关的运算。我们可以将双亲表示法和孩子表示法结合起来，形成双亲孩子表示法。图5-56为图5-52(a)所示的树的双亲孩子表示法的链表存储结构。

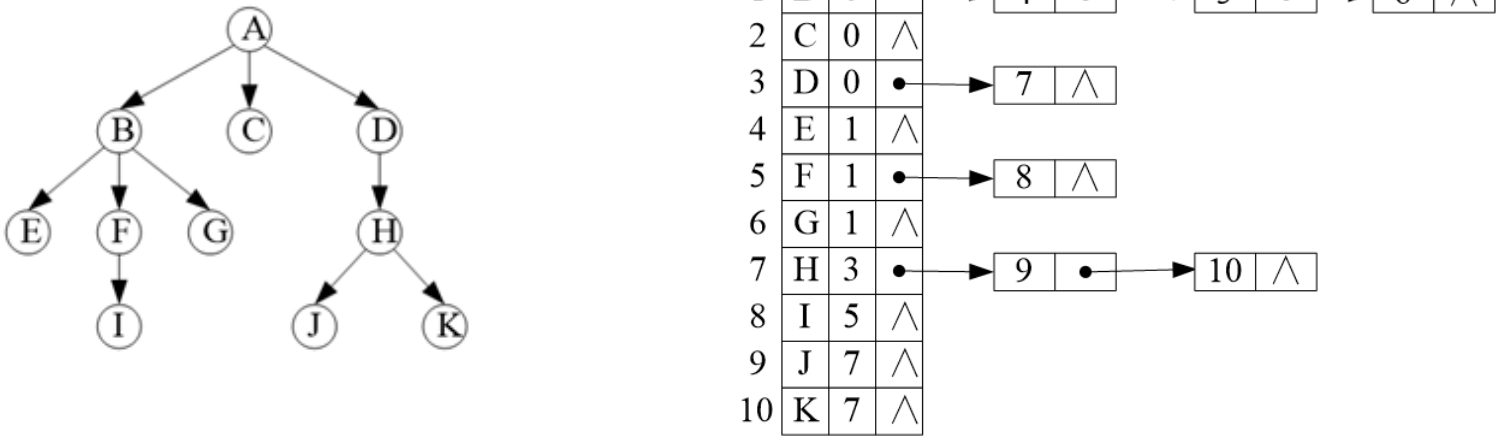


图 5-56 树的双亲孩子表示法的链表存储结构

3. 左孩子/右兄弟表示法

将每个结点用一个包含该结点的数据域、一个指向该结点最左孩子结点的指针域和右兄弟结点的指针域来表示，其结点结构图如图5-57所示。

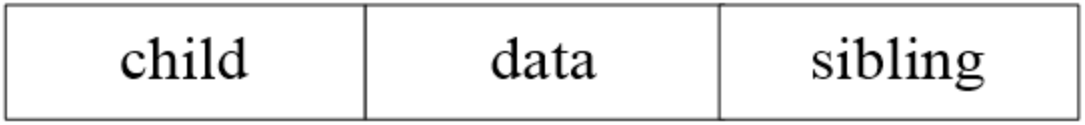
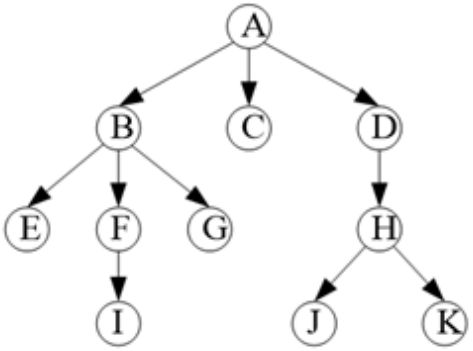


图 5-57 左孩子/右兄弟表示法结点结构图

图5-58为图5-52(a)所示的树的左孩子/右兄弟表示法的链表存储结构。



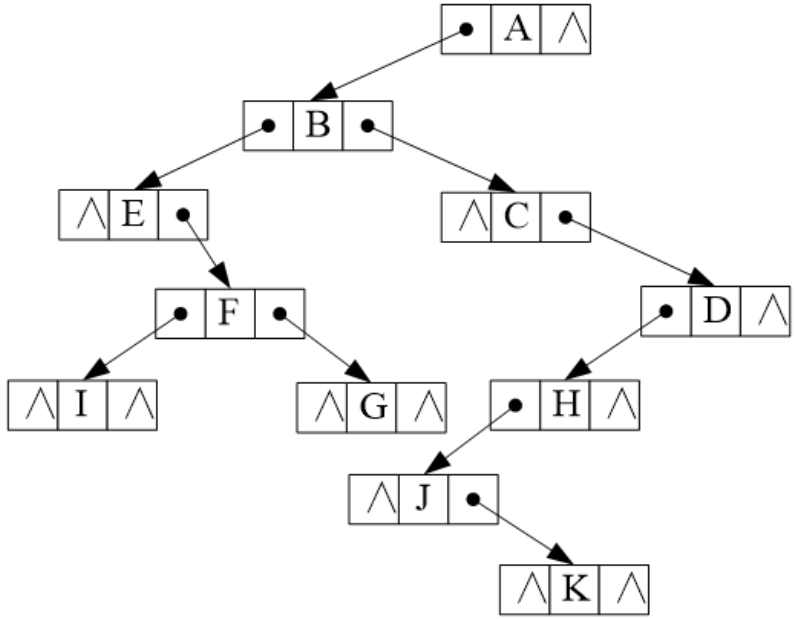


图 5-58 树的左孩子/右兄弟表示法的链式存储结构

5.8.2 树、森林与二叉树的转换

1. 树转换为二叉树

转换方法如下：

1）在所有兄弟结点之间加一条线；

2）对树中的每个结点，只保留它与最左边孩子的连线，删除该结点与其他孩子的连线；

3）以树根为轴心，对结点进行旋转处理即可得到相应的二叉树。

如图5-59所示为将图5-59(a)的一般树转换为二叉树的过程。

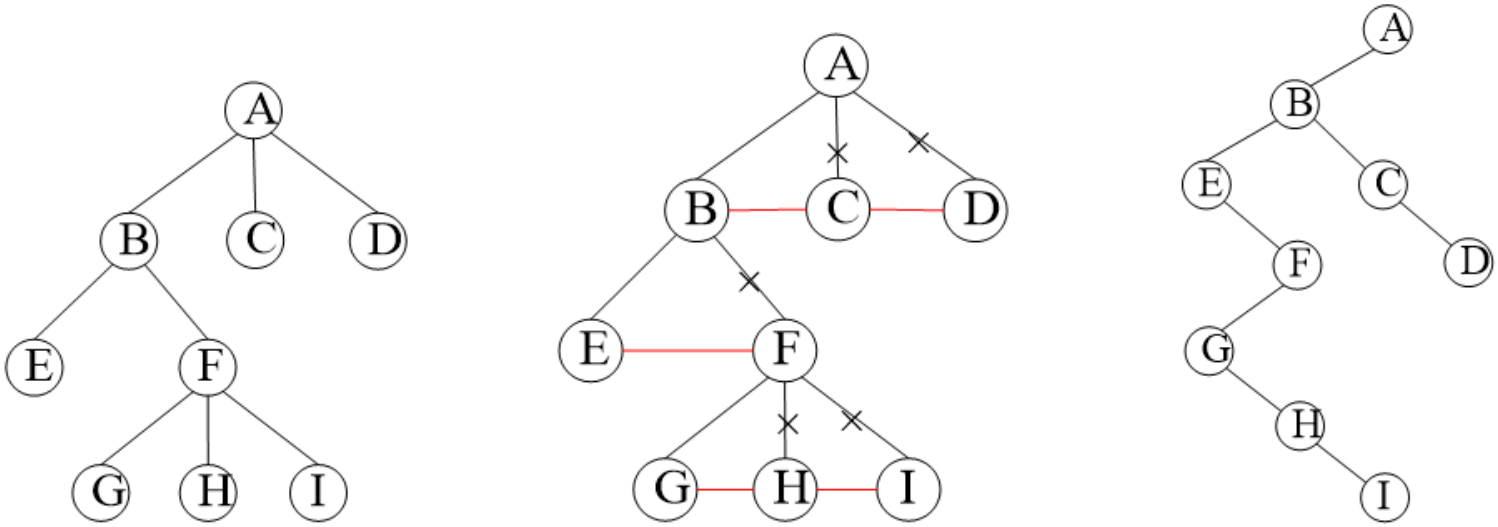


图 5-59 一般树转换为二叉树

2. 森林转换为二叉树

转换方法如下：

1）将森林中的每棵树转换为二叉树；

2）从第一棵二叉树开始，依次将森林中的后一棵二叉树的根结点作为前一棵二叉树根结点的右孩子连接在一起；

3）以第一棵二叉树的树根为轴心，对结点进行旋转处理即可得到相应的二叉树。

如图5-60所示为将图5-60(a)的森林转换为二叉树的过程。

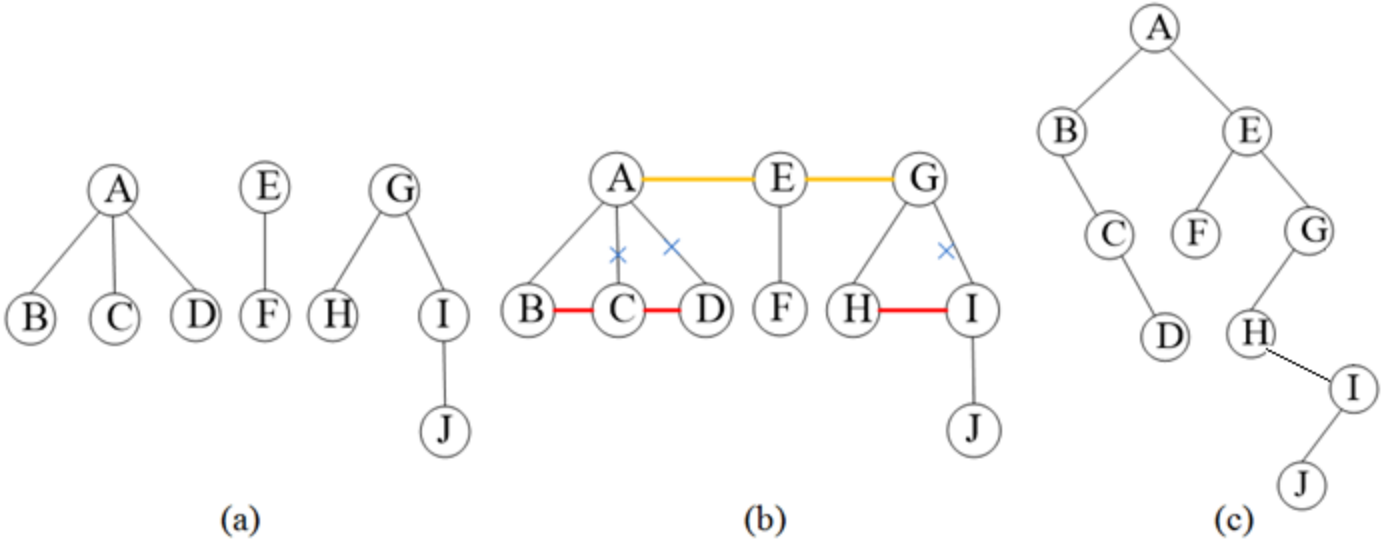


图 5-60 森林转换为二叉树

3. 二叉树转换为树或森林

转换步骤如下：

(1) 若二叉树的树根结点存在右子树，先将其与右子树的连线删除，得到分离的二叉树；

(2) 将分离后的二叉树按照以下步骤转换为树：若某结点的左孩子存在，将左孩子的右孩子，右孩子的右孩子……都作为该结点的孩子结点用线连接起来，然后删除掉原二叉树中所有结点与其右孩子的连线即可；

(3) 整理(2)得到的树即为分离的二叉树转换后的树；

(4) 若剩余二叉树非空，则转到（1）完成森林其余树转换。

如图5-61为二叉树转换为森林的过程。

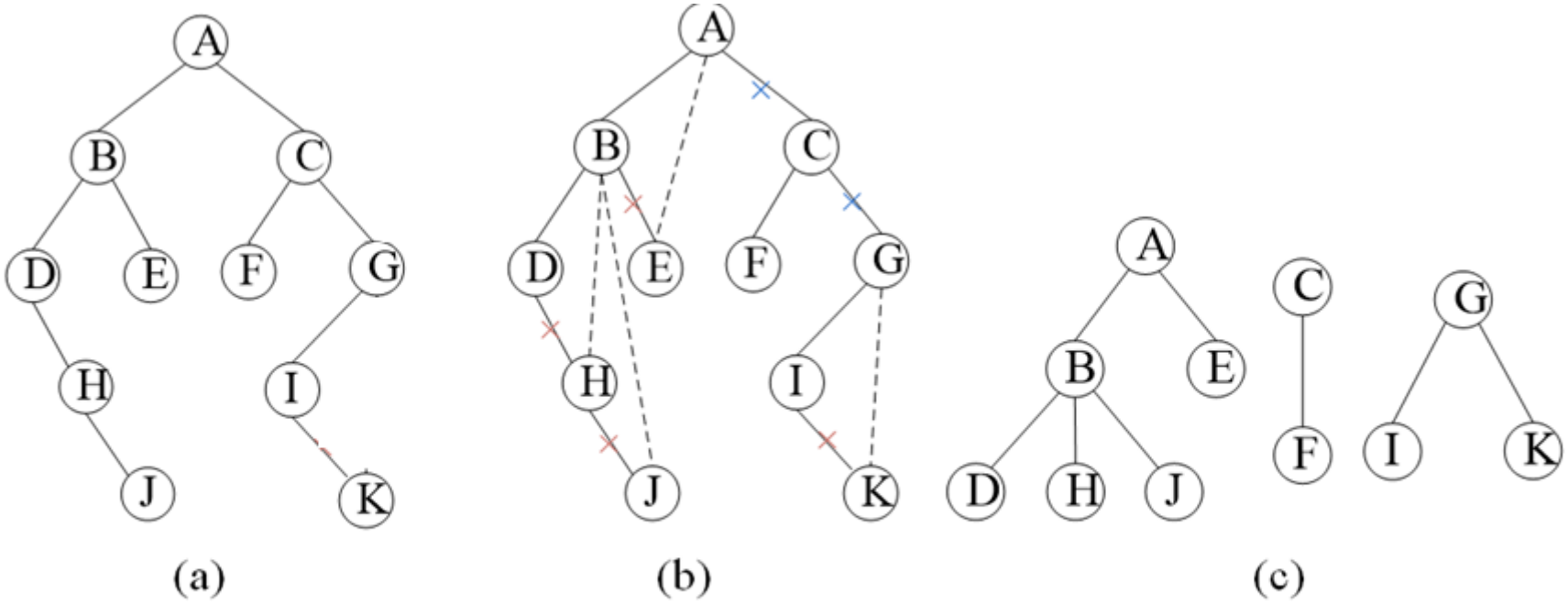


图 5-61 二叉树转换为森林

5.8.3 树与森林的遍历

树有两种遍历方法：先根遍历和后根遍历。设树T如图5-62所示，结点R是根，根的子树从左到右依次为$T_1, T_2, \ldots, T_k$。树的两种遍历方法的定义如下：

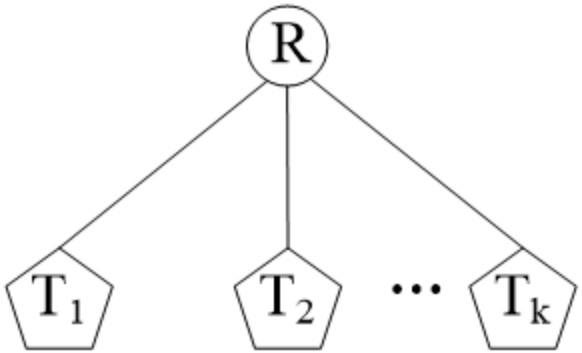


图 5-62 树 T

(1) 先根（序）遍历树

若树T非空，则：

①访问根结点R；

②依次先根遍历根R的各子树$T_1, T_2, \ldots, T_k$。

(2) 后根（序）遍历树

若树T非空，则：

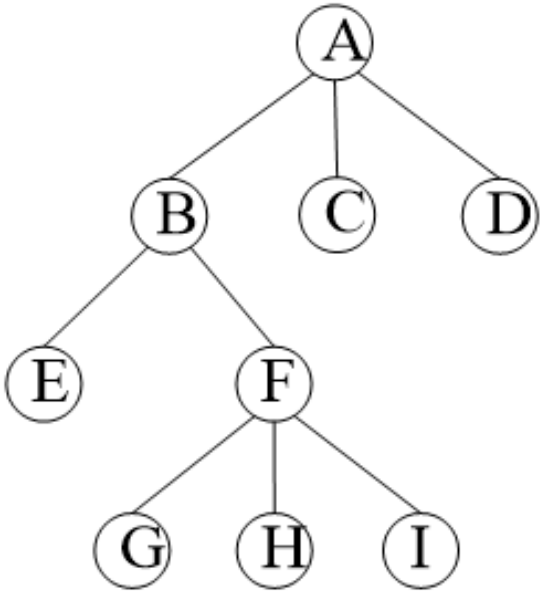
①依次后根遍历根T的各子树$T_1, T_2, \ldots, T_k$；

②访问根结点R。

对图5-59(a)所示的树进行先根遍历和后根遍历，

先根遍历序列：ABEFGHICD

后根遍历序列：EGHIFBCDA



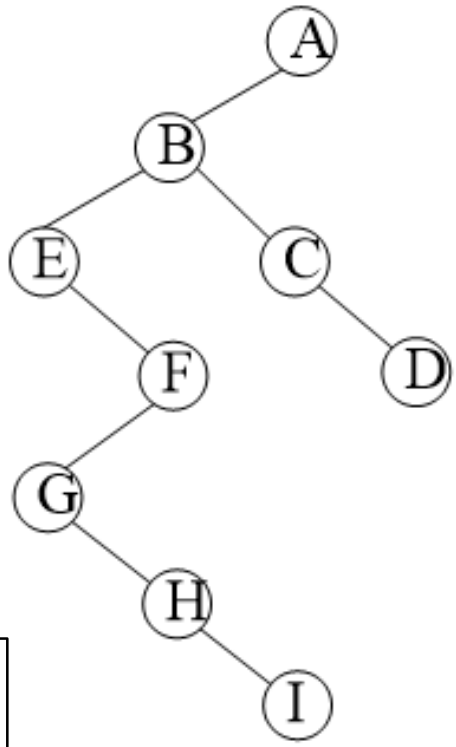
(a)

对图5-59(a)所示的树转换后对应的二叉树如图5-59(c)进行，

先序遍历：ABEFGHICD

中序遍历：EGHIFBCDA

后序遍历：IHGFEDCBA



(c)

对图5-59(a)所示的树进行先根遍历和后根遍历：
先根遍历序列：ABEFGHICD
后根遍历序列：EGHIFBCDA

森林的遍历定义如下：

(1) 先根（序）遍历森林

若森林非空，则：

①访问森林中第一棵树的根结点；

②先根遍历第一棵树中的子树森林；

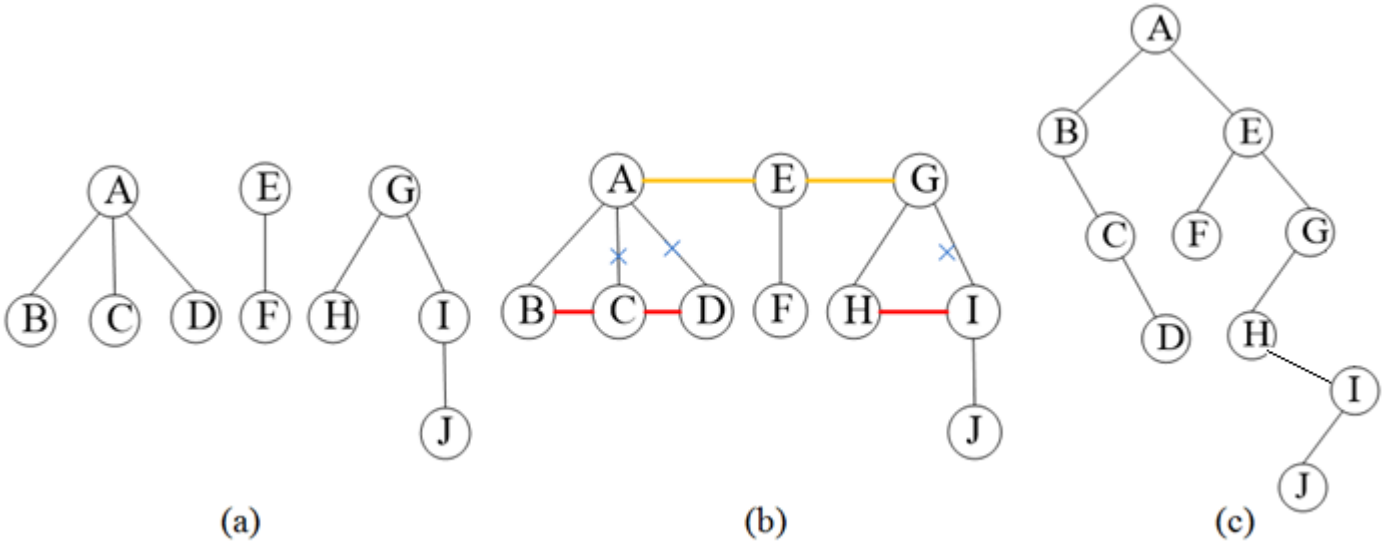
③先根遍历除去第一棵树之后剩余的树构成的森林。

(2) 中根（序）遍历森林

①中根遍历森林中第一棵树的根结点的子树森林；

②访问森林中第一棵树的根结点；

③中根遍历除去第一棵树之后剩余的树构成的森林。



森林： 先根遍历：ABCDEFGIJ

中根遍历：BCDAFEHJIG

森林转换后对应的二叉树： 先序遍历：ABCDEFGHIJ

中序遍历：BCDAFEHJIG