第二部分 树

- **树形结构是处理具有层次关系的数据元素**
- **这部分将介绍**
  - **树**
  - **二叉树**
  - **堆**

第6章 树

- **树的概念**
- **二叉树**
- **表达式树**
- **哈夫曼树与哈夫曼编码**
- **树和森林**

树的概念

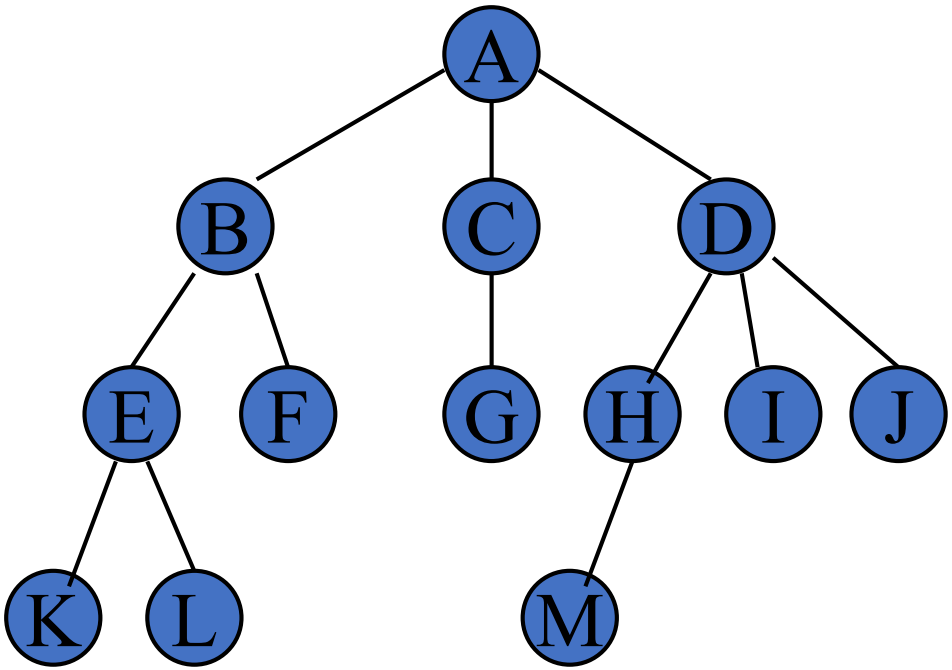
- **树的定义**
- **树的术语**
- **树的运算**

树的定义

树是n（n≥1）个结点的有限集合T，并且满足：

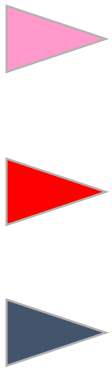
(1)有一个被称之为根(root)的结点

(2)其余的结点可分为m(m≥0)个互不相交的集合T_1，T_2，…，T_m，这些集合本身也是一棵树，并称它们为根结点的子树(Subree)。每棵子树同样有自己的根结点。



树的概念

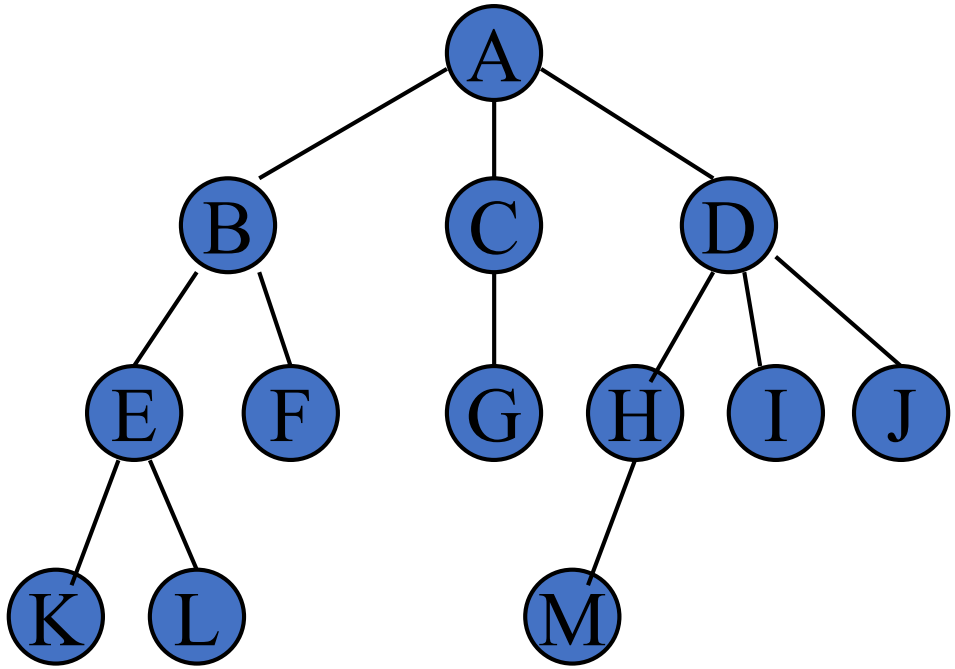
- **树的定义**
- **树的术语**
- **树的运算**



树的术语

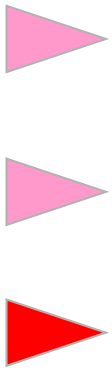
- 根结点、叶结点、内部节点
- 结点的度和树的度
- 儿子结点
- 父亲结点
- 兄弟结点
- 祖先结点
- 子孙结点
- 结点所处层次
- 树的高度

- 有序树
- 无序树
- 森林



树的概念

- **树的定义**
- **树的术语**
- **树的运算**



树的常用操作

- 建树create()：创建一棵空树
- 清空clear()：删除树中的所有结点
- 判空IsEmpty()：判别是否为空树
- 找根结点root()：找出树的根结点值；如果树是空树，则返回一个特殊值
- 找父结点parent(x)：找出结点x的父结点值；如果x不存在或x是根结点，则返回一个特殊值
- 找子结点child(x,i)：找结点x的第i个子结点值; 如果x不存在或x的第i个儿子不存在，则返回一个特殊值
- 剪枝remove(x,i)：删除结点x的第i棵子树
- 遍历traverse()：访问树上的每一个结点

树的抽象类

```
template<class T>
class tree {
public:
    virtual void clear() = 0;
    virtual bool isEmpty() const = 0;
    virtual T root(T flag) const = 0;
    virtual T parent(T x, T flag) const = 0;
    virtual T child（T x, int i, T flag) const = 0;
    virtual void remove(T x, int i) = 0;
    virtual void traverse() const = 0;
};
```

第五章 树

- **树的概念**
- **二叉树**
- **表达式树**
- **哈夫曼树与哈夫曼编码**
- **树和森林**

二叉树

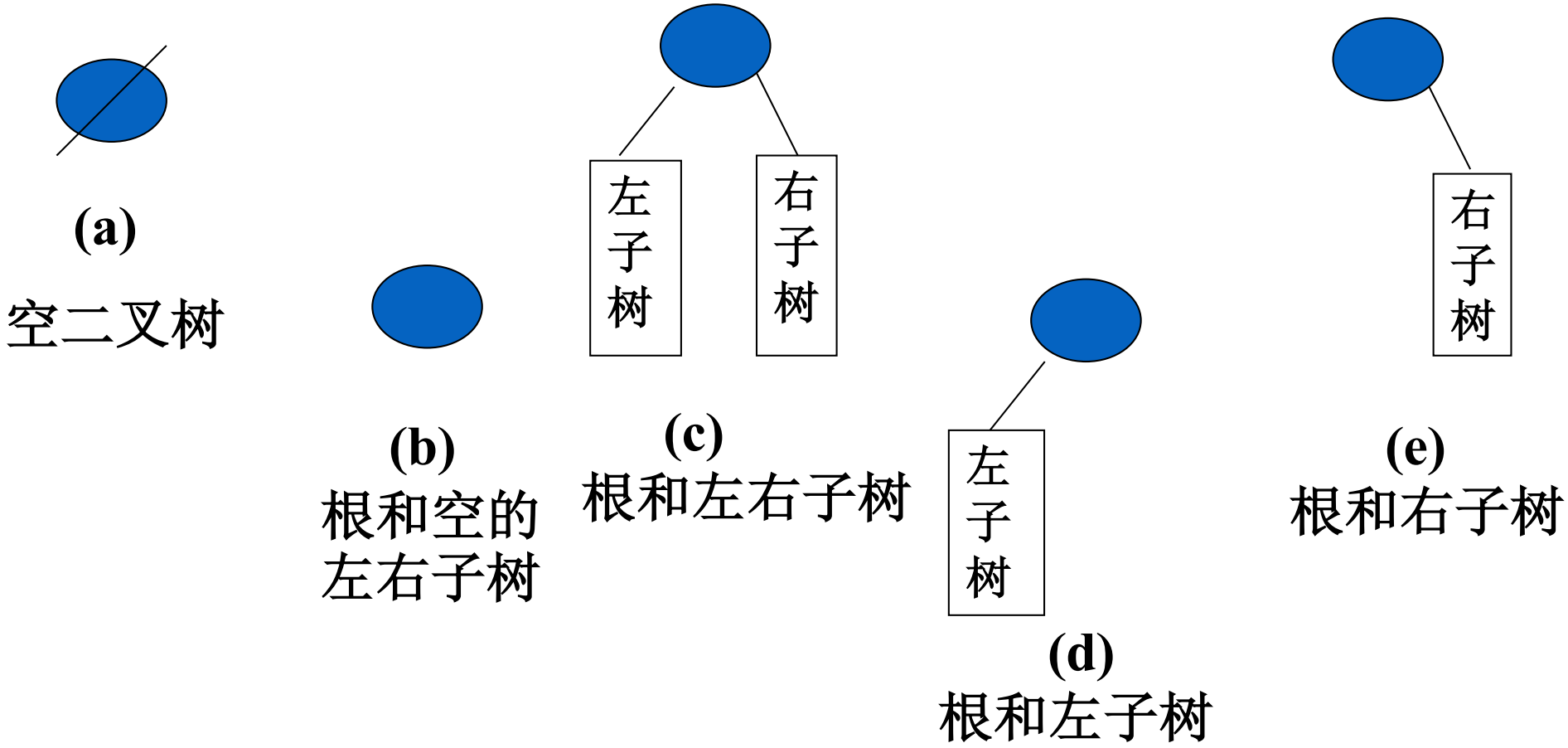
- **二叉树的概念**
- **二叉树的性质**
- **二叉树的基本运算**
- **二叉树的遍历**
- **二叉树的实现**
- **二叉树类**
- **二叉树遍历的非递归实现**

二叉树的定义

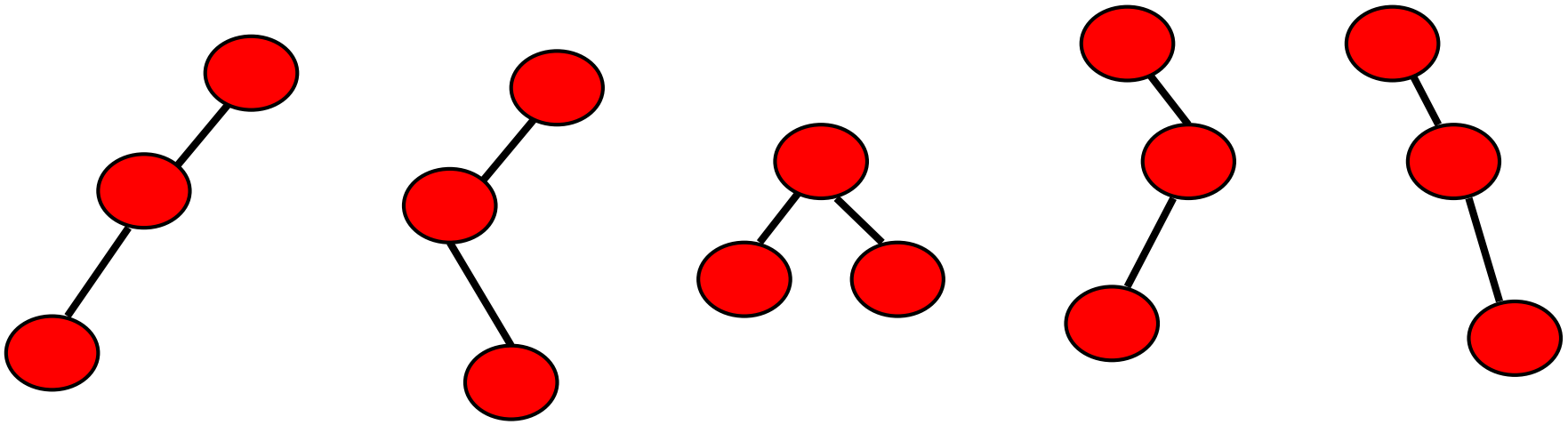
二叉树（Binary Tree）是结点的有限集合，它或者为空，或者由一个根结点及两棵互不相交的左、右子树构成，而其左、右子树又都是二叉树。

注意：二叉树必须严格区分左右子树。即使只有一棵子树，也要说明它是左子树还是右子树。交换一棵二叉树的左右子树后得到的是另一棵二叉树。

二叉树的基本形态

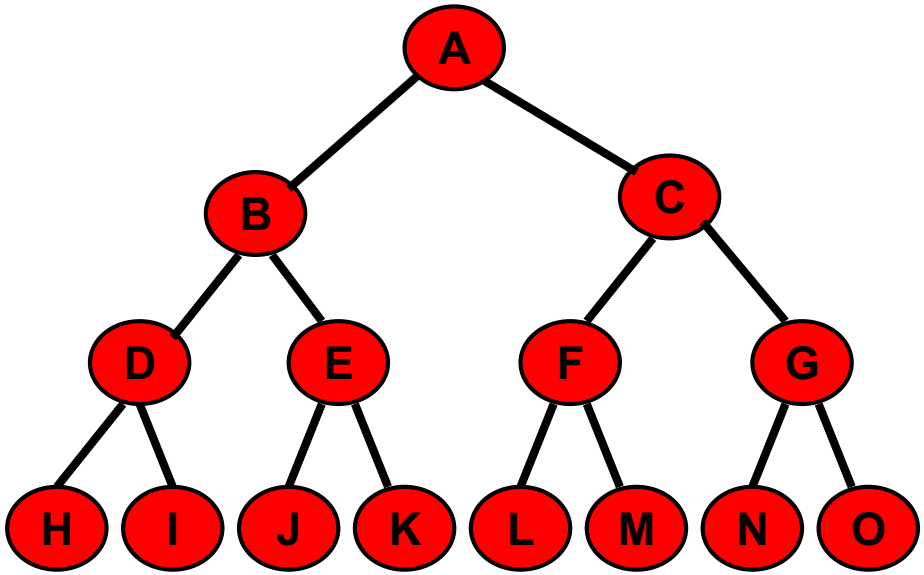


结点总数为3 的所有二叉树的不同形状

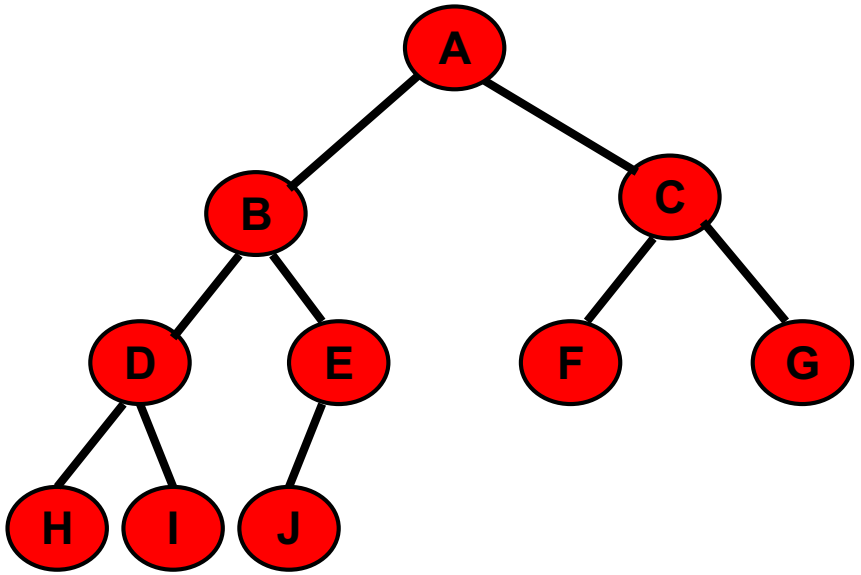


满二叉树

- **一棵高度为k并具有2^k - 1个结点的二叉树称为满二叉树。**
- **一棵二叉树中任意一层的结点个数都达到了最大值**



满二叉树实例



不是满二叉树

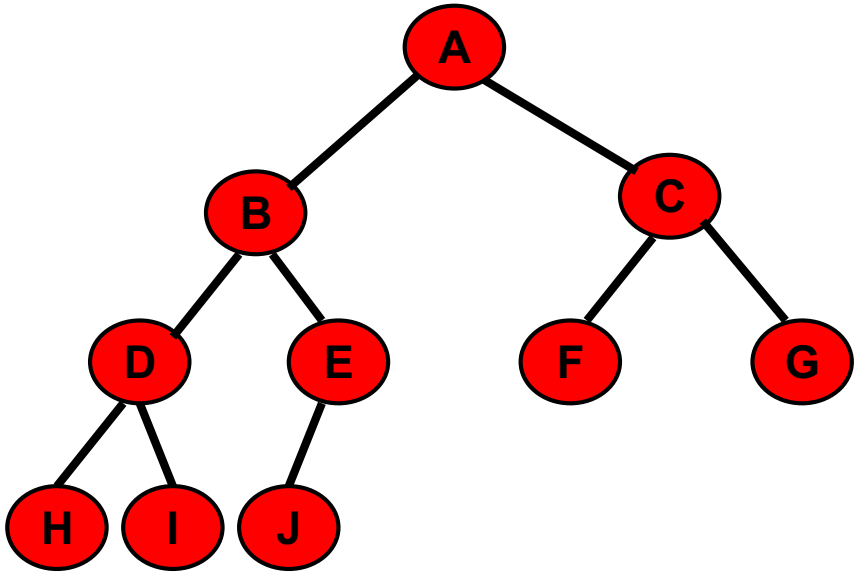
完全二叉树

在满二叉树的最底层自右至左依次(注意：不能跳过任何一个结点)去掉若干个结点得到的二叉树也被称之为完全二叉树。满二叉树一定是完全二叉树，但完全二叉树不一定是满二叉树。

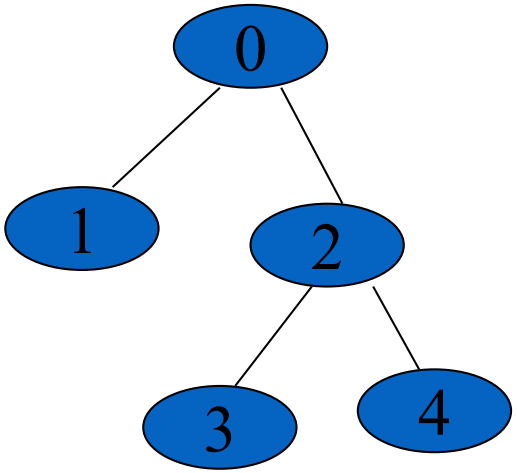
完全二叉树的特点是：

（1）所有的叶结点都出现在最低的两层上。

（2）对任一结点，如果其右子树的高度为k，则其左子树的高度为k或k + 1。



完全二叉树



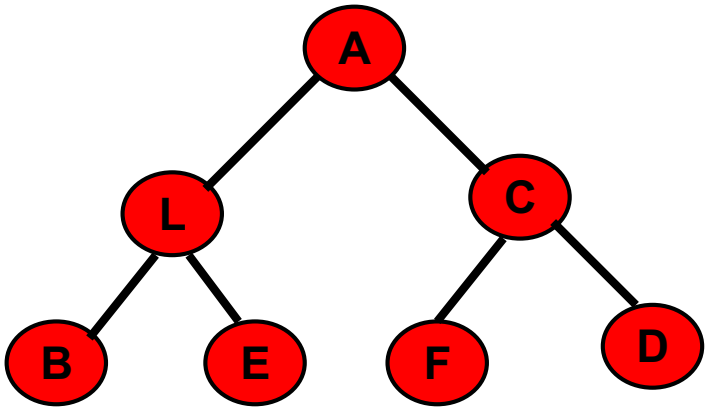
非完全二叉树

二叉树

- **二叉树的概念**
- **二叉树的性质**
- **二叉树的基本运算**
- **二叉树的遍历**
- **二叉树的实现**
- **二叉树类**
- **二叉树遍历的非递归实现**

二叉树的性质1

一棵非空二叉树的第 i 层上最多有2^{i-1}个结点（i≥1）。



1层：结点个数 $2^{1-1}=1$个

2层：结点个数 $2^{2-1}=2$ 个

3层：结点个数 $2^{3-1}=4$ 个

证明：当i=1时，只有一个根结点，$2^{i-1}=2^0=1$,命题成立。

假定对于所有的j，$1\le j\le k$,命题成立。

即第j层上至多有2^{j-1}个结点。

由归纳假设可知，第j = k层上至多有2^{k-1}个结点。

若要使得第k+1层上结点数为最多，那么，第k层上的每个结点都必须有二个儿子结点。

因此，第k+1层上结点数最多为为第k层上最多结点数的二倍，即：$2\times 2^{k-1}=2^{k+1-1}=2^k$。

所以，命题成立。

二叉树的性质2

一棵高度为k的二叉树，最多具有$2^k - 1$个结点。

证明：这棵二叉树中的每一层的结点个数必须最多。

根据性质1，第i层的结点数最多为等于2^{i-1}，

因此结点个数 N 最多为：

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

二叉树的性质3

对于一棵非空二叉树，如果叶子结点数为n_0，度数为2的结点数为n_2，则有:

$n_0 = n_2 + 1$ 成立。

性质3证明

证明：设n为二叉树的结点总数，n_1为二叉树中度数为1的结点数。

二叉树中所有结点均小于或等于2，所以有：

$$n = n_0 + n_1 + n_2$$

再看二叉树中的树枝(父结点和儿子结点之间的线段)总数。

在二叉树中，除根结点外，其余结点都有唯一的一个树枝进入本结点。

设B为二叉树中的树枝数，那么有下式成立：

$$B = n - 1$$

这些树枝都是由度为1和度为2的结点发出的，

一个度为1的结点发出一个树枝，一个度为2的结点发出两个树枝，所以有：

$$B = n_1 + 2n_2$$

因此，　$n_0 = n_2 + 1$

二叉树的性质4

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

证明 根据完全二叉树的定义和性质2可知，高度为k的完全二叉树的第一层到第k-1层具有最多的结点个数，总数为 $2^{k-1}-1$个，第k层至少具有一个结点，至多有2^{k-1}个结点。因此，

$$2^{k-1} - 1 < n \leq 2^k - 1$$

即 $$2^{k-1} \leq n < 2^k$$

对不等式取对数，有

$$k - 1 \leq \log_2 n < k$$

由于k是整数，所以有：$k = \lfloor \log_2 n \rfloor + 1$

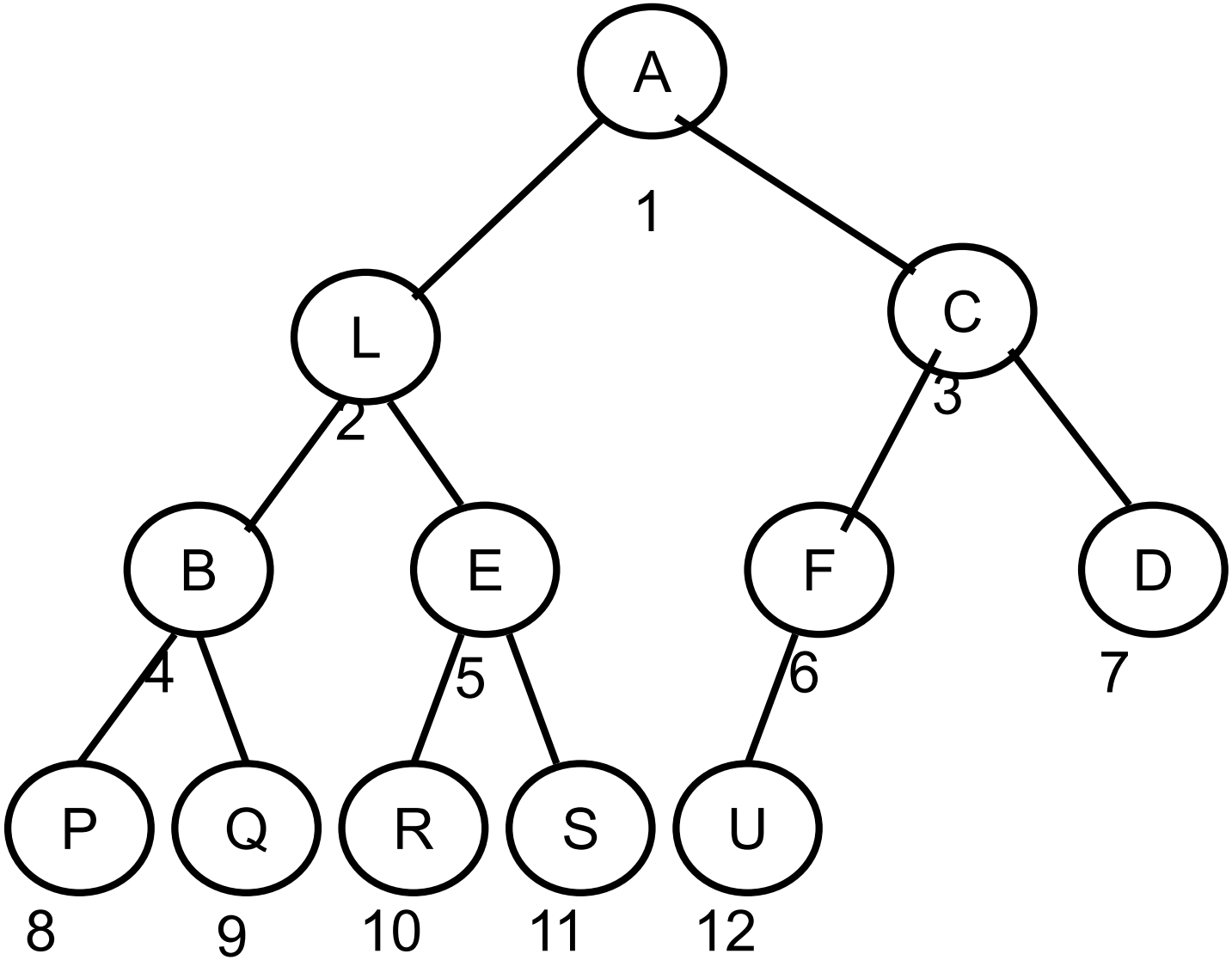
二叉树的性质5

如果对一棵有n个结点的完全二叉树中的结点按层自上而下（从第1层到第⌊log2n⌋+1层），每一层按自左至右依次编号。若设根结点的编号为1。则对任一编号为i的结点（1≤i≤n),有：

（1）如果i＝1，则该结点是二叉树的根结点；如果i>1，则其父亲结点的编号为⌊i/2⌋。

（2）如果2i ＞ n，则编号为i的结点为叶子结点，没有儿子；否则，其左儿子的编号为2i。

（3）如果2i + 1 ＞ n，则编号为i的结点无右儿子；否则，其右儿子的编号为2i+1。


A
1
L
2
C
3
B
E
F
D
4
5
6
7
P
Q
R
S
U
8
9
10
11
12

二叉树

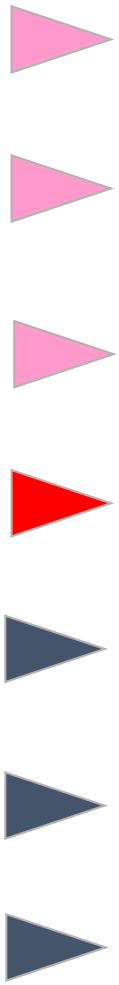
- **二叉树的概念**
- **二叉树的性质**
- **二叉树的基本运算**
- **二叉树的遍历**
- **二叉树的实现**
- **二叉树类**
- **二叉树遍历的非递归实现**

二叉树常用操作

- 建树create()：创建一棵空的二叉树
- 清空clear()：删除二叉树中的所有结点
- 判空IsEmpty()：判别二叉树是否为空树
- 找根结点root()：找出二叉树的根结点值；如果树是空树，则返回一个特殊值
- 找父结点parent(x)：找出结点x的父结点值；如果x不存在或x是根，则返回一个特殊值
- 找左孩子lchild(x)：找结点x的左孩子结点值；如果x不存在或x的左儿子不存在，则返回一个特殊值
- 找右孩子rchild(x)：找结点x的右孩子结点值；如果x不存在或x的右儿子不存在，则返回一个特殊值
- 删除左子树delLeft(x)：删除结点x的左子树
- 删除右子树delRight(x)：删除结点x的右子树
- 遍历traverse()：访问二叉树上的每一个结点

二叉树

- **二叉树的概念**
- **二叉树的性质**
- **二叉树的基本运算**
- **二叉树的遍历**
- **二叉树的实现**
- **二叉树类**
- **二叉树遍历的非递归实现**



二叉树的遍历

- **二叉树的遍历讨论的是如何访问到树上的每一个结点**
- **在线性表中，可以沿着后继链访问到所有结点。而二叉树是有分叉的，因此在分叉处必须确定下一个要访问的节点：是根结点、左结点还是右结点**
- **根据不同的选择，有三种遍历的方法：前序、中序和后序**
- **还有一种遍历方法是层次遍历**

前序遍历

- **如果二叉树为空，则操作为空**
- **否则**
  - **访问根结点**
  - **前序遍历左子树**
  - **前序遍历右子树**

中序遍历

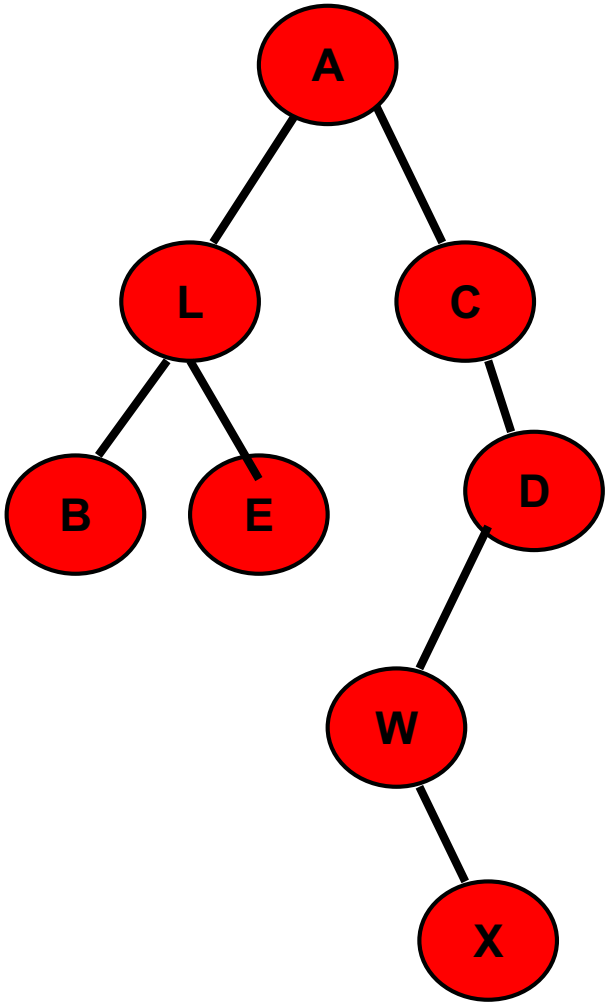
- **如果二叉树为空，则操作为空**
- **否则**
  - **中序遍历左子树**
  - **访问根结点**
  - **中序遍历右子树**

后序遍历

- **如果二叉树为空，则操作为空**
- **否则**
  - **后序遍历左子树**
  - **后序遍历右子树**
  - **访问根结点**

层次遍历

- 先访问根结点，然后按从左到右的次序访问第二层的结点。在访问了第k层的所有结点后，再按从左到右的次序访问第k+1层。以此类推，直到最后一层。



前序：A、L、B、E、 C、D、W、X

中序：B、L、E、A、 C、W、X、D

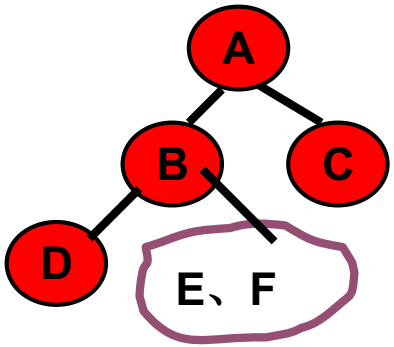
后序：B、E、L、X、 W、D、C、A

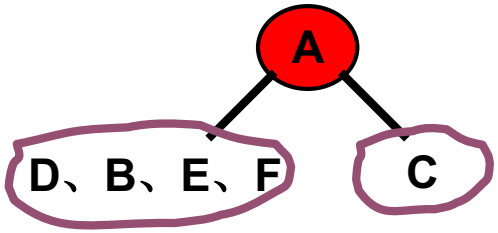
层次：A、L、C、B、E、D、W、X

前序 + 中序 唯一确定一棵二叉树

前序: A、B、D、E、F、C

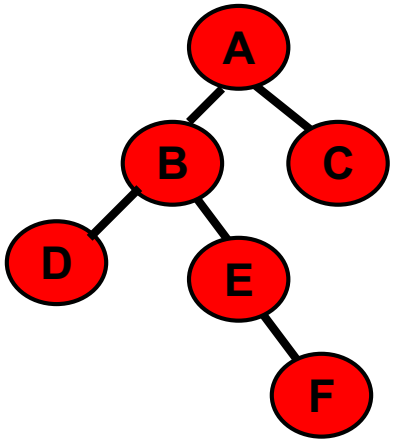
中序: D、B、E、F、A、C

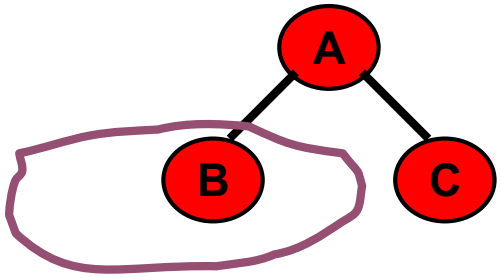




前序: E、F

中序: E、F





前序: B、D、E、F

中序: D、B、E、F

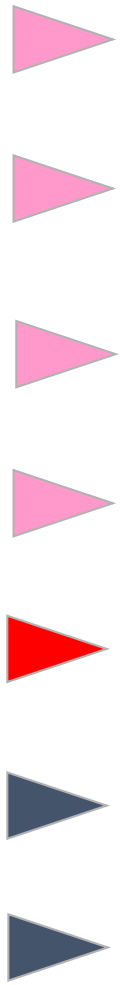
- 同理，由二叉树的后序序列和中序序列同样可以唯一地确定一棵二叉树
- 但是，已知二叉树的前序遍历序列和后序遍历序列却无法确定一棵二叉树。比如：已知一棵二叉树的前序遍历序列为A、B，后序遍历序列为B、A，我们虽然可以很容易地得知结点A为根结点，但是无法确定结点B是结点A的左儿子还是右儿子，因为B无论是结点A的右儿子还是左儿子都是符合已知条件的。

二叉树抽象类

```
template<class T>
class bTree {
public:
    virtual void clear() = 0;
    virtual bool isEmpty() const = 0;
    virtual T Root(T flag) const = 0;
    virtual T parent(T x，  T flag) const = 0;
    virtual T lchild  (T x, T flag) const = 0;
    virtual T rchild  (T x, T flag) const = 0;
    virtual void delLeft(T x) = 0;
    virtual void delRight(T x) = 0;
    virtual void preOrder() const = 0;
    virtual void midOrder() const = 0;
    virtual void postOrder() const= 0;
    virtual void levelOrder() const = 0;
};
```

二叉树

- **二叉树的概念**
- **二叉树的性质**
- **二叉树的基本运算**
- **二叉树的遍历**
- **二叉树的实现**
- **二叉树类**
- **二叉树遍历的非递归实现**

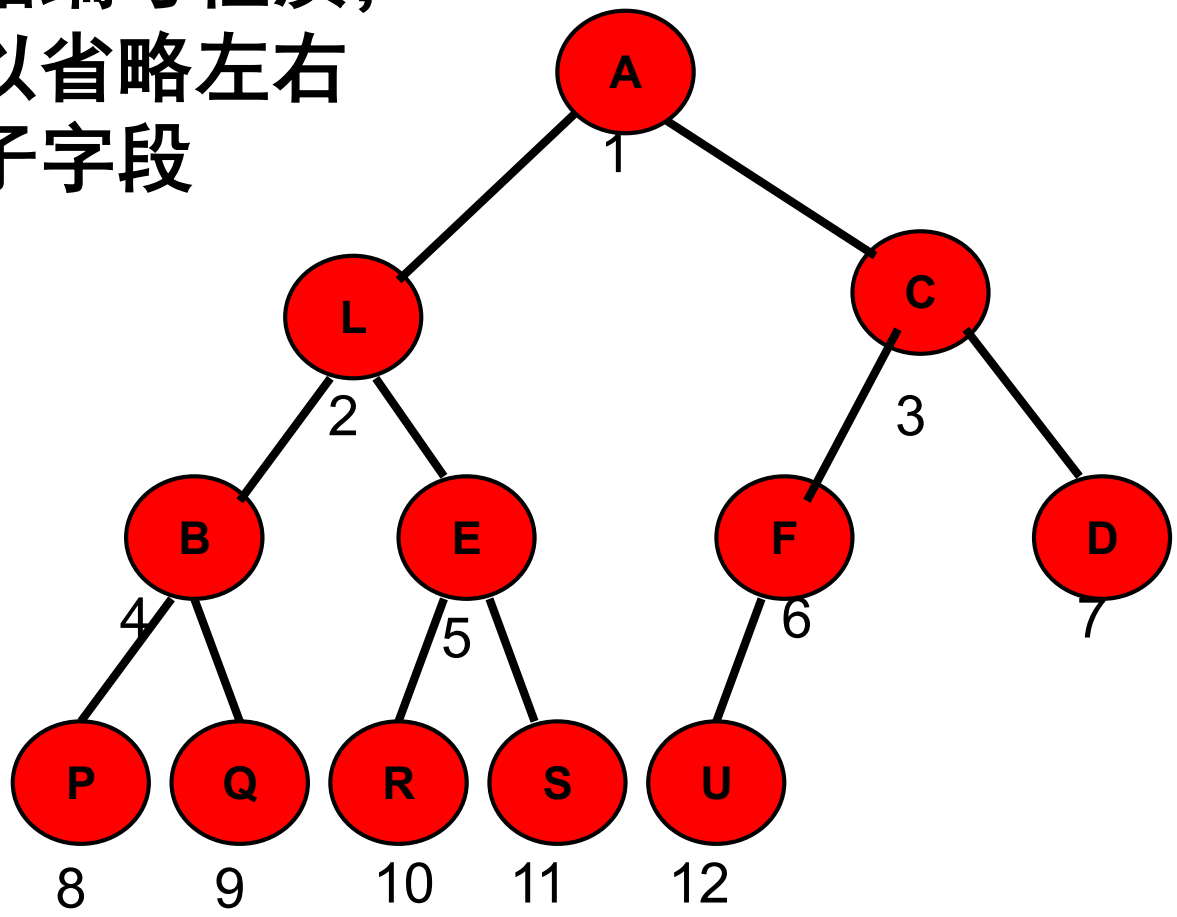


二叉树的实现

- **顺序实现**
- **链接实现**

完全二叉树的顺序存储

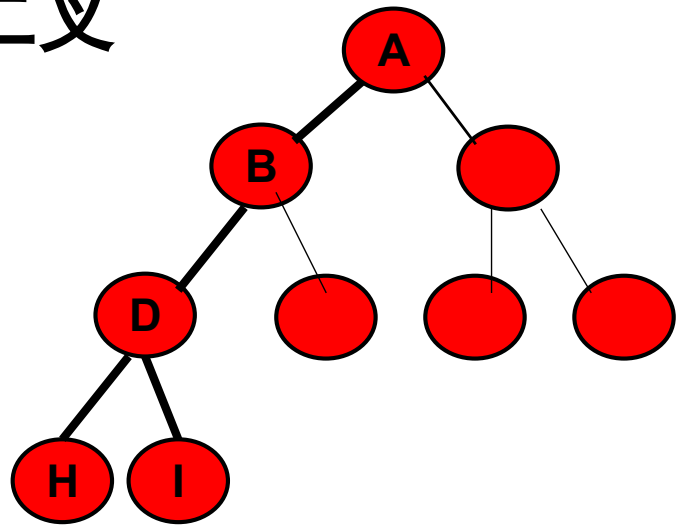
根据编号性质，可以省略左右孩子字段



0	
1	A
2	L
3	B
4	B
5	E
6	F
7	D
8	P
9	Q
10	R
11	S
12	U

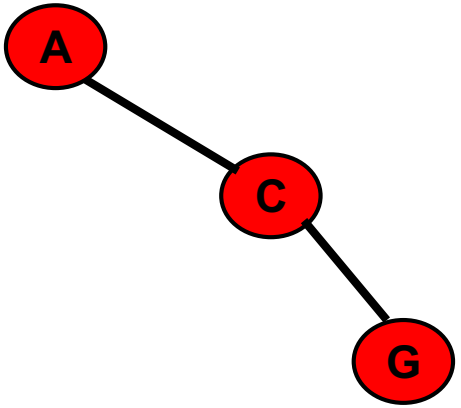
普通二叉树的顺序存储

将普通的树修补成完全二叉树



0	
1	A
2	B
3	∧
4	D
5	∧
6	∧
7	∧
8	H
9	I

右单支树的实例



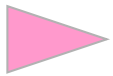
0	
1	A
2	∧
3	C
4	∧
5	∧
6	∧
7	G

顺序实现的特点

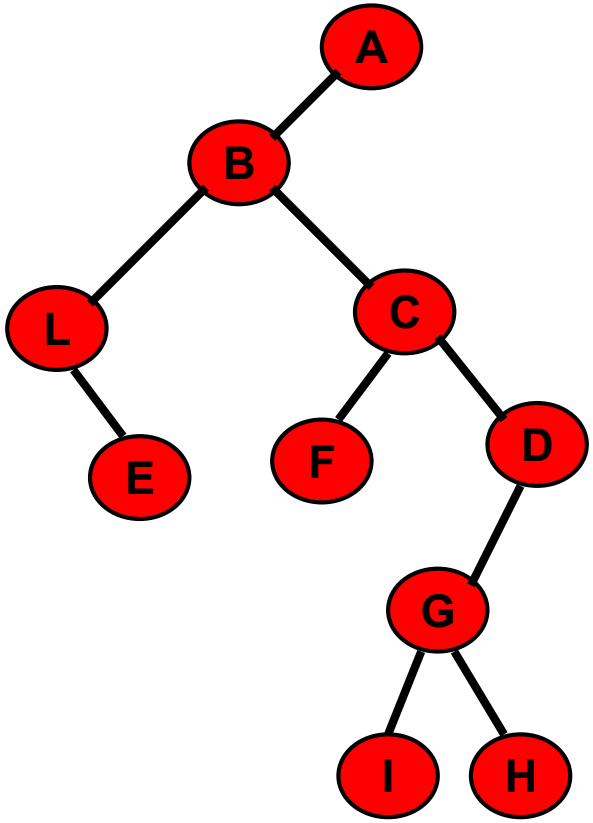
- **存储空间的浪费。**
- **一般只用于一些特殊的场合，如静态的并且结点个数已知的完全二叉树或接近完全二叉树的二叉树。**

二叉树的实现

- **顺序实现**
- **链接实现**



链接存储结构



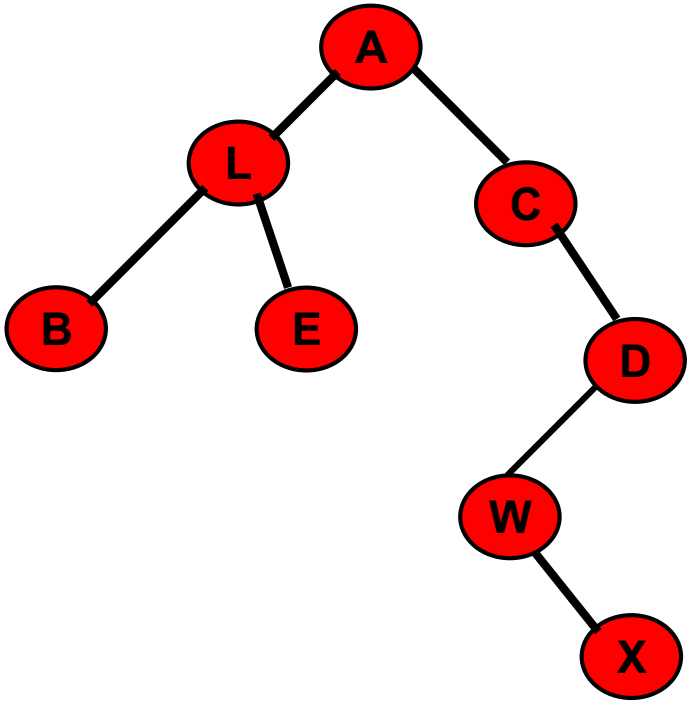
标准形式：（二叉链表）

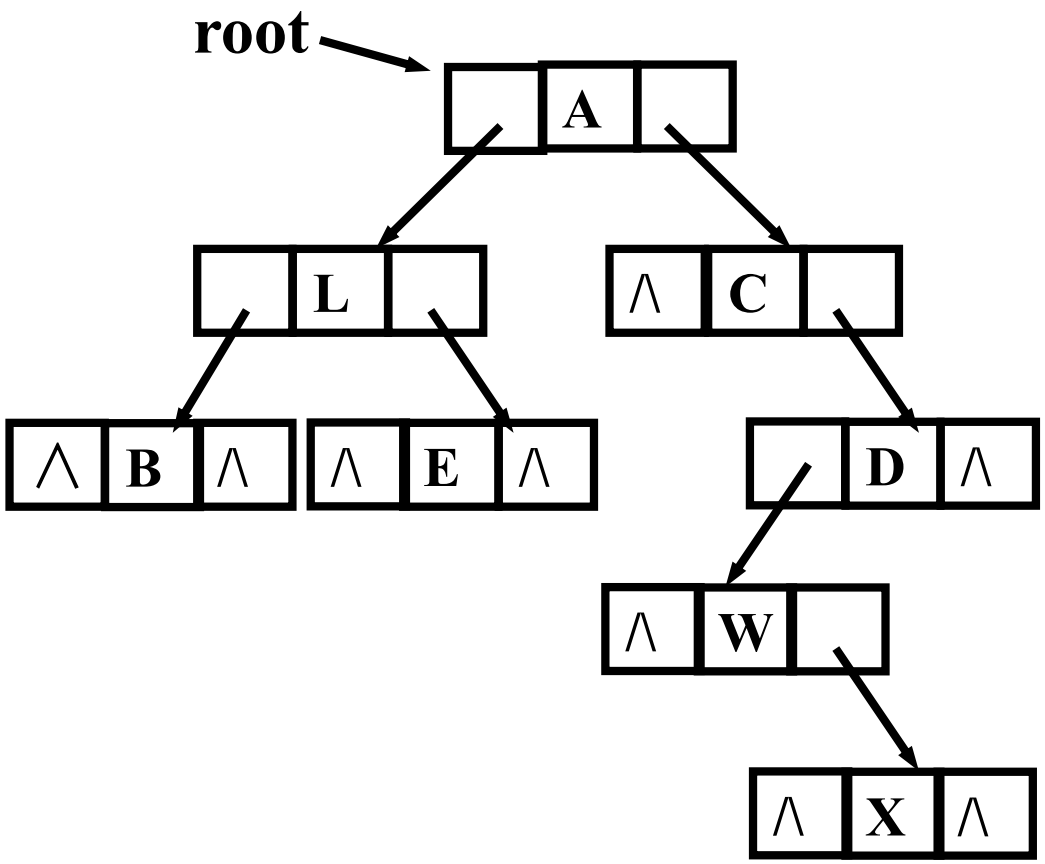
LEFT	DATA	RIGHT

广义标准形式：（三叉链表）

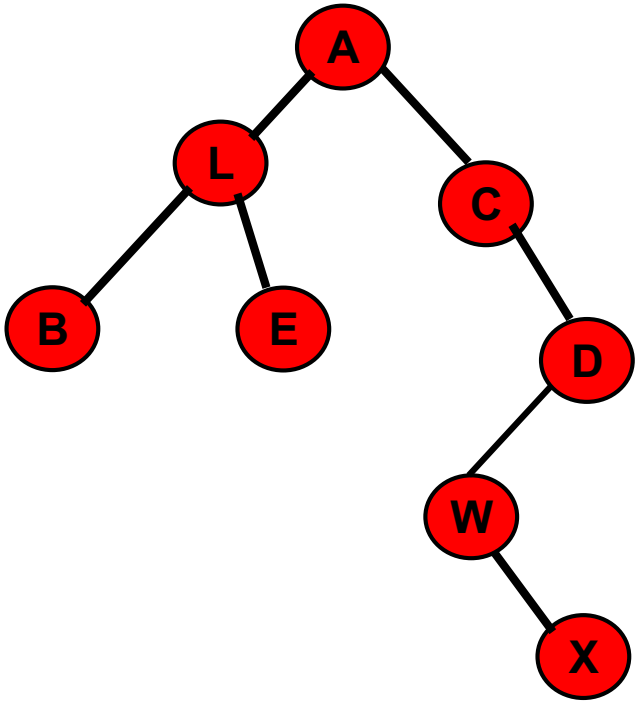
LEFT	DATA	RIGHT	PARENT

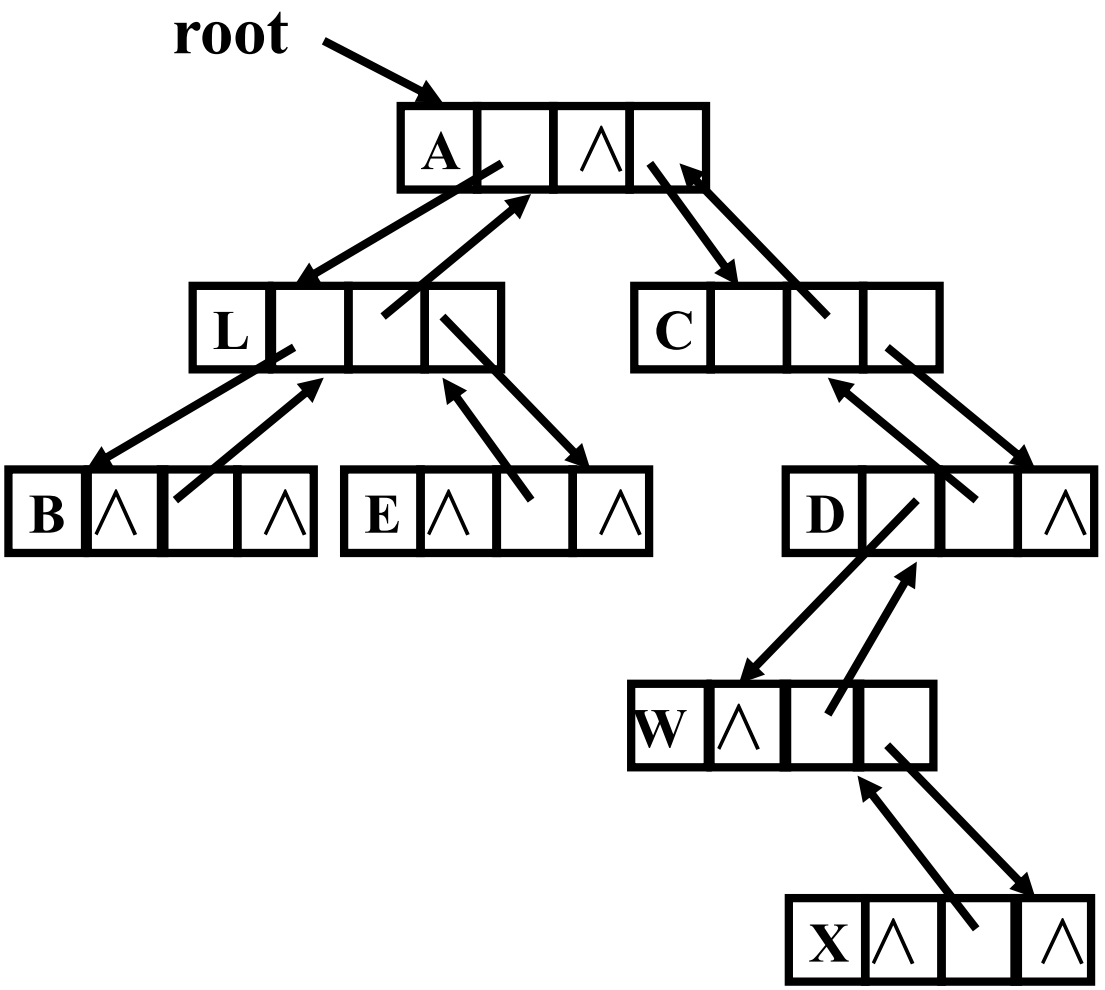
标准形式的实例





广义标准形式的实例





二叉树类

- 由于二叉树的顺序实现仅用于一些特殊的场合。大多数情况下，二叉树都是用二叉链表实现，所以仅介绍用二叉链表实现的二叉树类。

二叉树类的设计

- **标准的链接存储由两个类组成：结点类和树类。**
- **和线性表的实现类似，这个结点类是树类专用的，因此可作为树类的私有内嵌类。**

结点类Node的设计

- **存储和处理树上每一个结点的类。**
- **数据成员包括：结点的数据及左右孩子的指针。**
- **结点的操作包括：**
  - **构造和析构**

二叉树类的设计

- **树的存储：存储指向根结点的指针**
- **树的操作：**
  - **树的标准操作**
  - **增加了一个建树的函数**

递归函数的设计

- **对于二叉树类的用户来说，他并不需要知道这些操作时用递归函数实现的。**
- **对用户来说，调用这些函数并不需要参数**
- **但递归函数必须有一个控制递归终止的参数**
- **设计时，我们将用户需要的函数原型作为公有的成员函数。每个公有成员函数对应一个私有的、带递归参数的成员函数。公有函数调用私有函数完成相应的功能。**

二叉树类的定义

```
template<class T>
class binaryTree : public bTree<T> {
  friend void printTree(const binaryTree &t, T flag);
private:
    struct Node {                          //二叉树的结点类
        Node  *left , *right ;
        T data;
      Node() : left(NULL), right(NULL) { }
      Node(T item, Node *L = NULL, Node * R =NULL ) :
                                          data(item), left(L), right(R) {}
      ~Node() {}
    };
    Node *root;
```

```
public:
        binaryTree() : root(NULL) {}
        binaryTree(T x) { root = new Node(x); }
        ~binaryTree();
        void clear() ;
        bool isEmpty() const;
        T Root(T flag) const;
        T lchild(T x, T flag) const;
        T rchild(T x, T flag) const;
        void delLeft(T x) ;
        void delRight(T x);
        void preOrder() const;
        void midOrder() const;
        void postOrder() const;
        void levelOrder() const;
        void createTree(T flag);
        T parent(T x, T flag) const { return flag; }
```

```
    private:
        Node *find(T x, Node *t ) const;
        void clear(Node *&t) ;
        void preOrder(Node *t)  const;
        void midOrder(Node *t) const;
        void postOrder(Node *t)  const;
};
```

二叉树基本运算的实现

- **构造函数：将指向根结点的指针设为空指针就可以了，即root = NULL。**
- **isEmpty()：只需要判别root即可。如果root等于空指针，返回true，否则，返回false。**
- **root()：返回root指向的结点的数据部分。如果二叉树是空树，则返回一个特殊的标记。**

```
template<class T>
bool binaryTree<T>::isEmpty() const
{
        return root == NULL;
}

template <class T>
T binaryTree<T>::Root(T flag) const
{
        if (root == NULL) return flag;
        else return root->data;
}
```

clear()

- **从递归的观点看，一棵二叉树由三部分组成：根结点、左子树、右子树。删除一棵二叉树就是删除这三个部分。**
- **根结点的删除很简单，只要回收根结点的空间，把指向根结点的指针设为空指针。**
- **如何删除左子树和右子树呢？记住左子树也是一棵二叉树，右子树也是一棵二叉树，左右子树的删除和整棵树的删除过程是一样的。**

```
template<class T>
void binaryTree<T>::clear(binaryTree<T>::Node *&t)
{
      if (t == NULL)  return;
      clear(t->left);
      clear(t->right);
      delete t;
      t = NULL;
}

template<class T>
void binaryTree<T>::clear()
{
         clear(root);
}
```

析构函数

- 释放存储元素的所有结点，这可以由clear函数实现

```
template <class T>
binaryTree<T>::~binaryTree()
{
        clear(root);
}
```

二叉树的遍历

前序：

访问根结点；

If（左子树非空）前序遍历左子树；

If（右子树非空）前序遍历右子树；

其他两种遍历只要调整一下次序即可。

preOrder()

```
template<class T>
void binaryTree<T>::preOrder(binaryTree<T>::Node *t)  const
{      if (t == NULL) return;
      cout << t->data << ' ';
      preOrder(t->left);
      preOrder(t->right);
}

template<class T>
void binaryTree<T>::preOrder()  const
{    cout << "\n前序遍历：";
     preOrder(root);
}
```

midOrder()

```
template<class T>
void binaryTree<T>::midOrder(binaryTree<T>::Node *t)
  const
{    if (t == NULL) return;
     midOrder(t->left);
     cout << t->data << ' ';
       midOrder(t->right);
}

template<class T>
void binaryTree<T>::midOrder() const
{    cout << "\n中序遍历：";
      midOrder(root);
}
```

postOrder()

```
template<class T>
void binaryTree<T>::postOrder(binaryTree<T>::Node *t)
  const
{    if (t == NULL) return;
     postOrder(t->left);
       postOrder(t->right);
     cout << t->data << ' ';
}

template<class T>
void binaryTree<T>::postOrder()  const
{      cout << "\n后序遍历：";
       postOrder(root);
}
```

层次遍历

```
template<class T>
void binaryTree<T>::levelOrder() const
{
  linkQueue< Node * > que;
    Node *tmp;

    cout << "\n层次遍历：";
  que.enQueue(root);

   while (!que.isEmpty()) {
       tmp = que.deQueue();
       cout << tmp->data << ' ';
       if (tmp->left) que.enQueue(tmp->left);
       if (tmp->right) que.enQueue(tmp->right);
    }
}
```

- lchild(x)：返回结点x的left值
- rchild(x)：返回结点x的right值
- delLeft(x)：对左子树调用clear函数删除左子树，然后将结点x的left置为NULL。
- delRight(x)：对右子树调用clear函数删除右子树，然后将结点x的right置为NULL。
- 这4个函数都必须先找到存储x的结点，这是由私有函数find完成

Find(x)

- find函数采用了前序遍历查找x。
- 首先检查根结点是否存放的是x。如果是则返回根结点地址
- 对左子树递归调用find函数。
- 如果返回值是空指针，说明x不在左子树上，于是再对右子树递归调用find函数，返回调用的结果。

```
template <class T>
binaryTree<T>::Node *binaryTree<T>::
        find(T x, binaryTree<T>::Node *t) const
{
        Node *tmp;
        if (t == NULL) return NULL;
        if (t->data == x) return t;
        if (tmp = find(x, t->left) ) return tmp;
        else return find(x, t->right);
}
```

delLeft(T x)

```
template <class T>
void binaryTree<T>::delLeft(T x)
{
    Node *tmp = find(x, root);
    if (tmp == NULL) return;
    clear(tmp->left);
}
```

delRight(T x)

```
template <class T>
void binaryTree<T>::delRight(T x)
{
    Node *tmp = find(x, root);
    if (tmp == NULL) return;
    clear(tmp->right);
}
```

lchild(T x, T flag)

```
template <class T>
T binaryTree<T>::lchild(T x, T flag) const
{
    Node * tmp = find(x, root);
    if (tmp == NULL || tmp->left == NULL) return flag;

    return tmp->left->data;
}
```

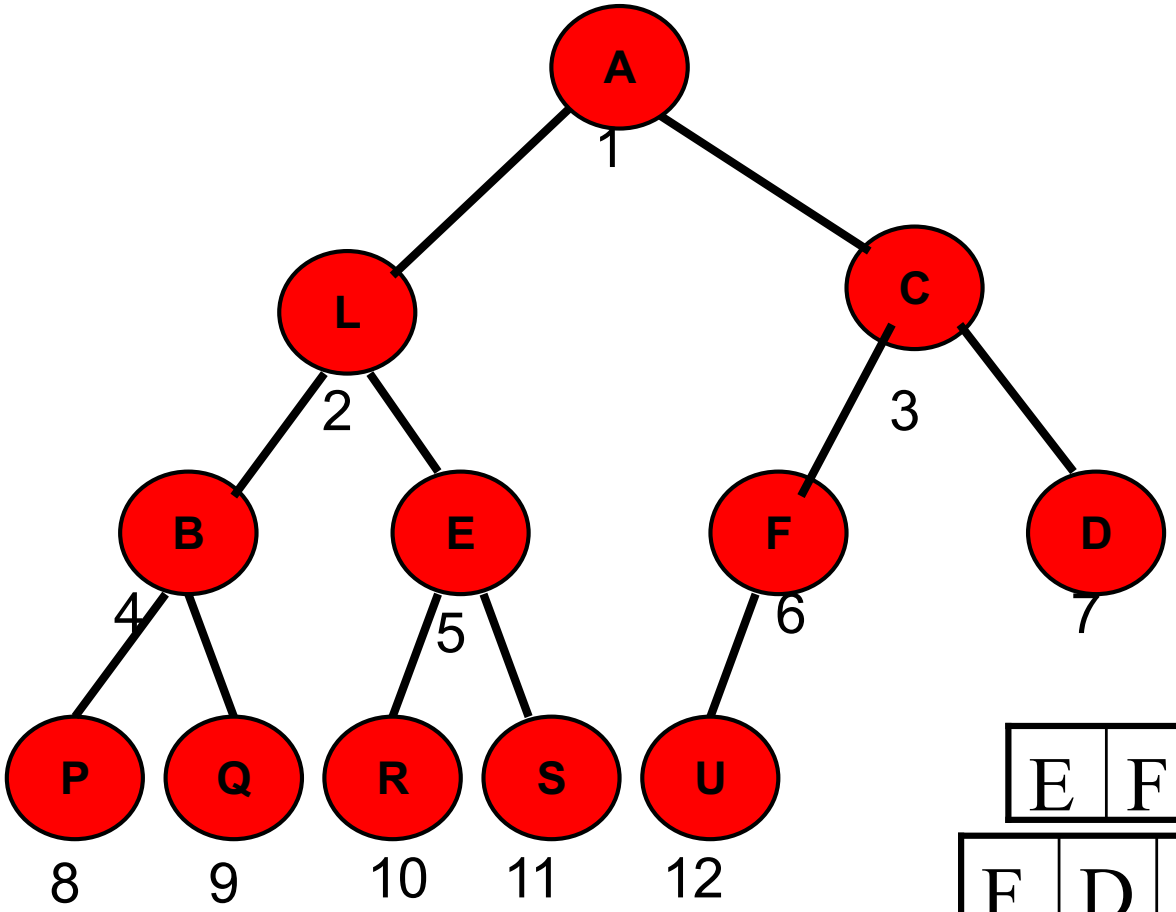
rchild(T x, T flag)

```
template<class T>
T binaryTree<T>::rchild(T x, T flag) const
{
    Node *tmp = find(x, root);
   if (tmp == NULL || tmp->right == NULL) return flag;

    return tmp->right->data;
}
```

创建一棵树

- **创建过程：**
  - **先输入根结点的值，创建根节点**
  - **对已添加到树上的每个结点，依次输入它的两个儿子的值。如果没有儿子，则输入一个特定值**
- **实现工具：**
  - **使用一个队列，将新加入到树中的结点放入队列**
  - **依次出队。对每个出队的元素输入它的儿子**

队列的变化



A			

L	C		

C	B	E	

B	E	F	D

E	F	D	P	Q			

F	D	P	Q	R	S		

D	P	Q	R	S	U		

P	Q	R	S	U			

Q	R	S	U				

R	S	U					

S	U						

U							

createTree

```
template <class Type>
void BinaryTree<Type>::createTree(Type flag)
{ linkQueue< Node * > que;
  Node *tmp;
  Type x, ldata, rdata;

  //创建树，输入flag表示空
  cout << "\n输入根结点：";
  cin >> x;
  root = new Node(x);
  que.enQueue(root);
```

```
while (!que.isEmpty()) {
  tmp = que.deQueue();
  cout << "\n输入" << tmp->data
          << "的两个儿子(" << flag
          << "表示空结点)：";
    cin >> ldata >> rdata;
  if (ldata != flag)
       que.enQueue(tmp->left = new Node(ldata));
  if (rdata != flag)
      que.enQueue(tmp->right = new Node(rdata));
  }
  cout << "create completed!\n";
}
```

printTree

- 以层次遍历的次序输出每个结点和它的左右孩子

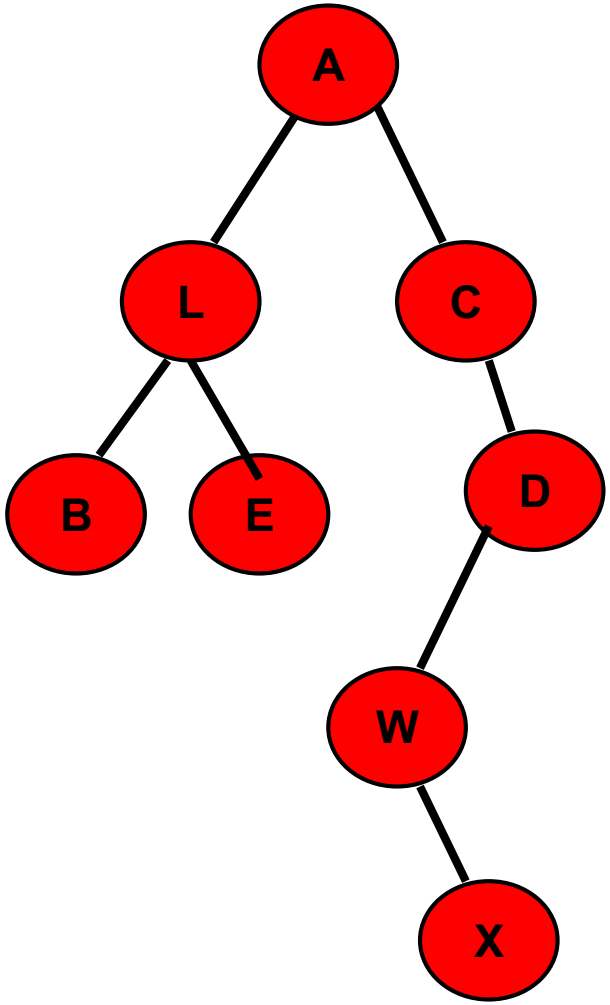
```
template <class T>
void printTree(const binaryTree<T> &t, T flag)
{       linkQueue<T> q;
        q.enQueue(t.root->data);
        cout << endl;
        while (!q.isEmpty()) {
                char p, l, r;
                p = q.deQueue();
                l = t.lchild(p,flag);
                r = t.rchild(p,flag);
                cout << p << "  " << l  << "  " << r << endl;
                if (l != '@') q.enQueue(l);
                if (r != '@') q.enQueue(r);
        }
}
```

二叉树类的应用

- 定义了二叉树对象tree
- 调用createTree输入这棵二叉树
- 执行前序、中序、后序和层次遍历
- 输出这棵树
- 删除L的左子树，删除C的左右子树
- 输出这棵树

```
int main()
{
        binaryTree<char> tree;

        tree.createTree('@');
        tree.preOrder();            tree.midOrder();
        tree.postOrder();           tree.levelOrder();
        printTree(tree, '@');
        tree.delLeft('L');
        tree.delRight('C');   tree.delLeft('C');
        printTree(tree, '@');
        return 0;
}
```

输入为：



输出为：

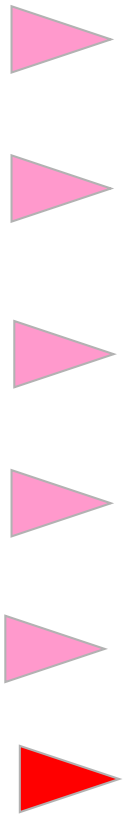
前序遍历：A L B E C D W X
中序遍历：B L E A C W X D
后序遍历：B E L X W D C A
层次遍历：A L C B E D W X

A L C
L B E
C @ D
B @ @
E @ @
D W @
W @ X
X @ @

A L C
L @ E
C @ @
E @ @

二叉树

- **二叉树的概念**
- **二叉树的性质**
- **二叉树的基本运算**
- **二叉树的遍历**
- **二叉树的实现**
- **二叉树遍历的非递归实现**



二叉树遍历的非递归实现

- **递归是程序设计中强有力的工具。**
- **递归程序结构清晰、明了、美观，**
- **递归程序的弱点：它的时间、空间的效率比较低。**
- **所以在实际使用中，我们常常希望使用它的非递归版本**
- **二叉树的遍历也是如此。尽管二叉树遍历的递归函数非常简洁，但有时我们还是希望使用速度更快的非递归函数。**

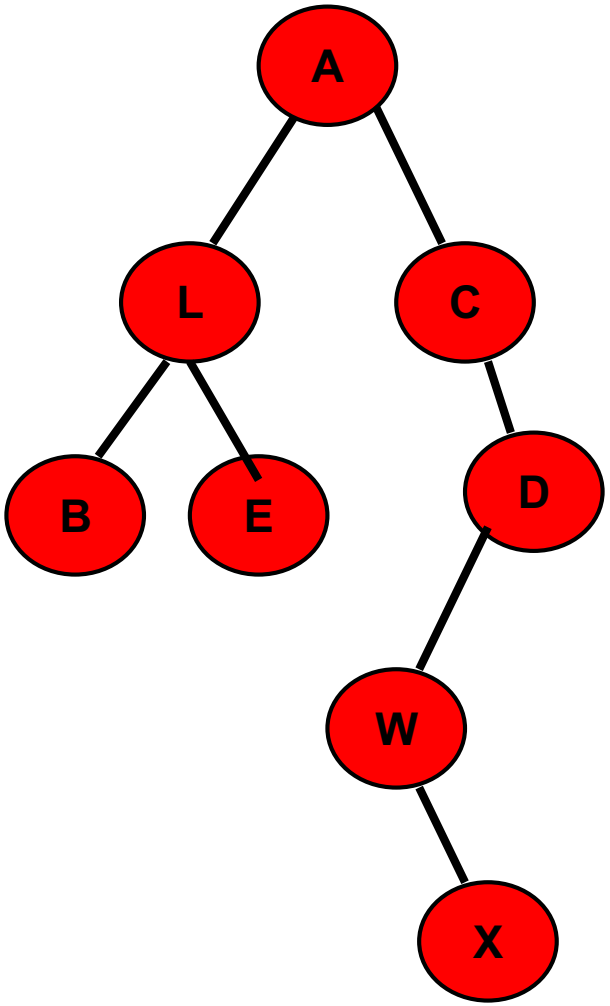
二叉树遍历的非递归实现

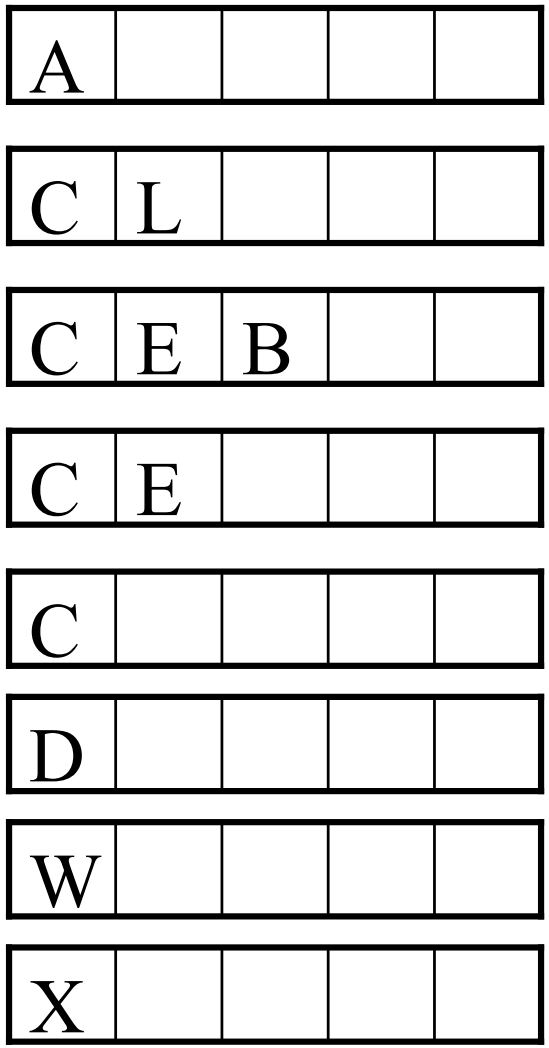
- **前序遍历**
- **中序遍历**
- **后序遍历**

前序遍历的非递归实现

- **前序遍历第一个被访问的结点是根结点，然后访问左子树，最后访问右子树。**
- **可以设置一个栈，保存将要访问的树的树根。**
- **开始时，把二叉树的根结点存入栈中。然后重复以下过程，直到栈为空：**
  - **从栈中取出一个结点，输出根结点的值；**
  - **然后把右子树，左子树放入栈中**

前序遍历的过程





输出：

A

L

B

E

C

D

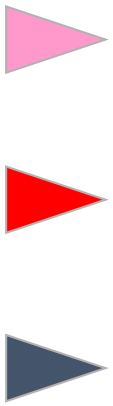
W

X

```
template <class Type>
void BinaryTree<Type>::preOrder() const
{ linkStack<Node *> s;
  Node *current;

  cout << "前序遍历: ";
  s.push( root );
  while ( !s.isEmpty() )   {
          current = s.pop();
          cout << current->data;
           if ( current->right != NULL ) s.push( current->right );
          if ( current ->left != NULL ) s.push( current->left );
         }
}
```

二叉树遍历的非递归实现

- **前序遍历**
- **中序遍历**
- **后序遍历**



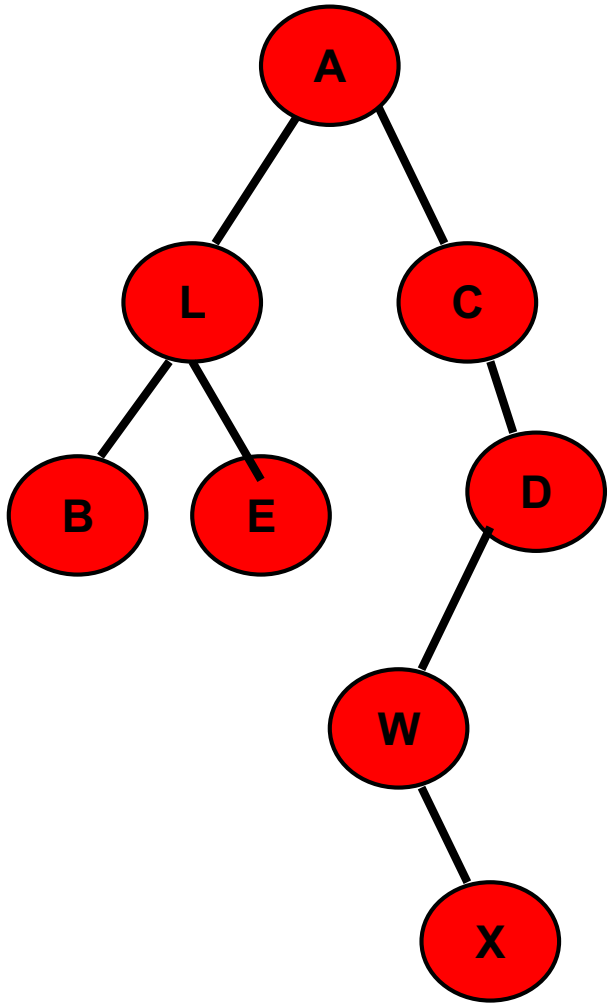
中序遍历的非递归实现

- **采用一个栈存放要遍历的树的树根**
- **中序遍历中，先要遍历左子树，接下去才能访问根结点，因此，当根结点出栈时，我们不能访问它，而要访问它的左子树，此时要把树根结点暂存一下。**
- **存哪里？由于左子树访问完后还要访问根结点，因此仍可以把它存在栈中，接着左子树也进栈。此时执行出栈操作，出栈的是左子树。左子树问结束后，再次出栈的是根结点，此时根结点可被访问。根结点访问后，访问右子树，则将右子树进栈。**

栈元素的设计

- **在中序遍历中根结点要进栈两次。**
- **当要遍历一棵树时，将根结点进栈。**
- **根结点第一次出栈时，它不能被访问，必须重新进栈，并将左子树也进栈，表示接下去要访问的是左子树。**
- **根结点第二次出栈时，才能被访问，并将右子树进栈，表示右子树可以访问了。**
- **在中序遍历时不仅要记住需要访问哪一棵树，而且还必须记住根结点是在第几次进栈。**

中序遍历的过程



A			
0			

A	L		
1	0		

A	L	B	
1	1	0	

A	L	B	
1	1	1	

A	L		
1	1		

A	E		
1	0		

A	E		
1	1		

A			
1			

C			
0			

C			
1			

D			
0			

D	W		
1	0		

D	W		
1	1		

D	X		
1	0		

D	X		
1	1		

D			
1			

输出： B L E A C W X D

StNode类的定义

```
struct  StNode
{
  Node *node;
  int TimesPop;
  StNode ( Node *N = NULL ):node(N), TimesPop(0) {  }
 };
```

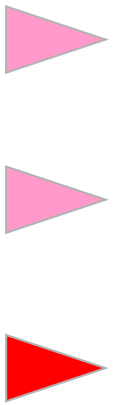
中序遍历的非递归实现

```
template <class Type>
void BinaryTree<Type>::midOrder() const
{ linkStack<StNode> s;
  StNode current(root);

  cout << "中序遍历: ";
  s.push(current);
```

```
while (!s.isEmpty()) {
    current = s.pop();
      if ( ++current.TimesPop == 2 )  {
         cout << current.node->data;
         if ( current.node->right != NULL )
              s.push(StNode(current.node->right ));}
    else { s.push( current );
              if ( current.node->left != NULL )
               s.push(StNode(current.node->left) );
          }
     }
}
```

二叉树遍历的非递归实现

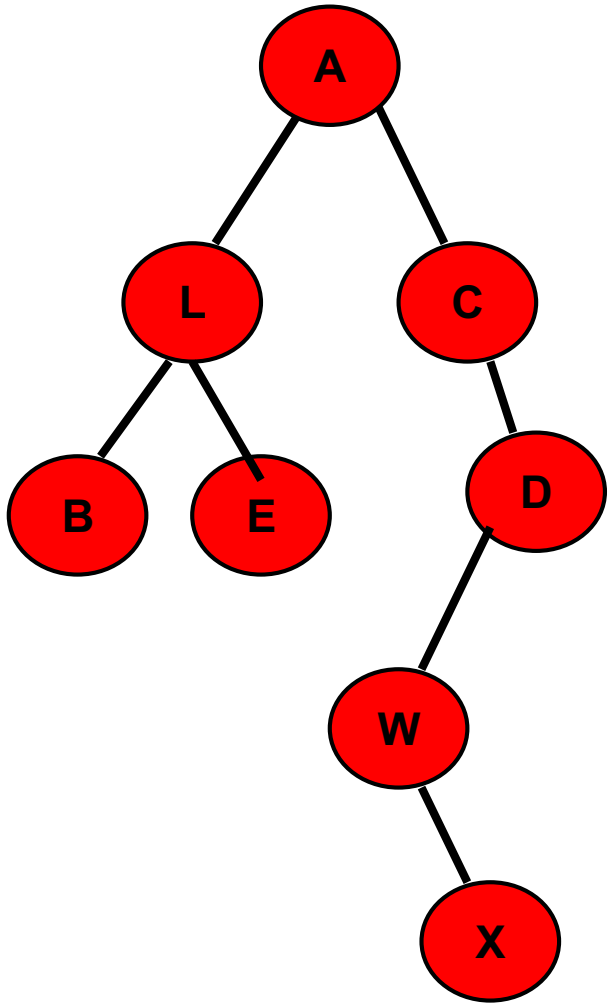
- **前序遍历**
- **中序遍历**
- **后序遍历**



后序遍历的非递归实现

- **将中序遍历的非递归实现的思想进一步延伸，可以得到后序遍历的非递归实现。**
- **当以后序遍历一棵二叉树时，先将树根进栈，表示要遍历这棵树。**
- **根结点第一次出栈时，根结点不能访问，应该访问左子树。于是，根结点重新入栈，并将左子树也入栈。**
- **根结点第二次出栈时，根结点还是不能访问，要先访问右子树。于是，根结点再次入栈，右子树也入栈。**
- **当根结点第三次出栈时，表示右子树遍历结束，此时，根结点才能被访问。**

后序遍历的过程



A			
0			

A	L		
1	0		

A	L	B	
1	1	0	

A	L	B	
1	1	1	

A	L	B	
1	1	2	

A	L	E	
1	2	0	

A	L	E	
1	2	1	

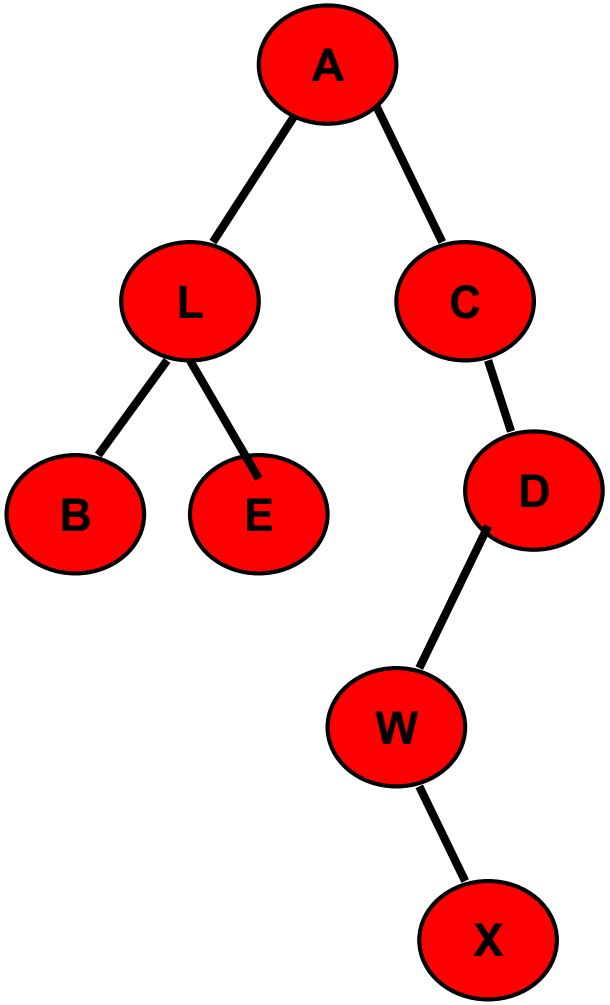
A	L	E	
1	2	2	

A	L		
1	2		

A	C		
2	0		

A	C		
2	1		

输出： B E L



A	C	D		
2	2	0		

A	C	D	W	
2	2	1	2	

A	C	D	W	
2	2	1	0	

A	C	D		
2	2	2		

A	C	D	W	X
2	2	1	2	0

A	C			
2	2			

A	C	D	W	X
2	2	1	2	1

A				
2				

A	C	D	W	X
2	2	1	2	2

输出： **B L E X W D C A**

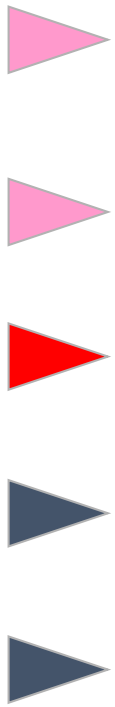
后序遍历的非递归实现

```
template <class Type>
void BinaryTree<Type>::postOrder() const
{
        linkStack< StNode > s;
        StNode current(root);
      cout << "后序遍历: ";
        s.push(current);
```

```
while (!s.isEmpty())
  {  current = s.pop();
     if ( ++current.TimesPop == 3 )
         {cout << current.node->data; continue;}
     s.push( current );
     if ( current.TimesPop == 1 )
          { if ( current.node->left != NULL )
              s.push(StNode( current.node->left) );
         }
        else {
          if ( current.node ->right != NULL )
                    s.push(StNode( current.node->right ) );
    }
  }
}
```

第五章 树

- **树的概念**
- **二叉树**
- **表达式树**
- **哈夫曼树与哈夫曼编码**
- **树和森林**



表达式树

- **算术表达式可以表示为一棵二叉树，如:**

  (4-2)*(10+(4+6)/2)+2

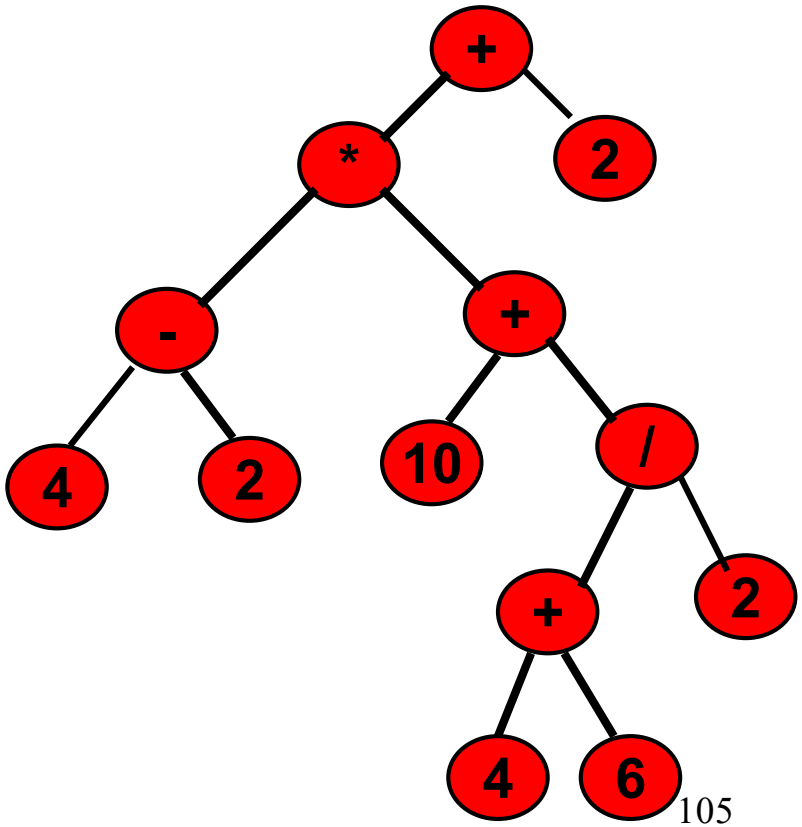
- **对这棵树后序遍历可**

  得到结果

- **设计一个类，**

  利用表达式树计算由

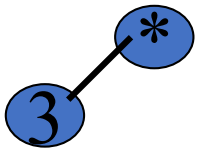
  四则运算组成的表达式



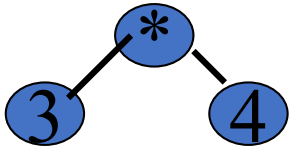
树的构建过程

3*4+5*7*9+8　构建左节点3

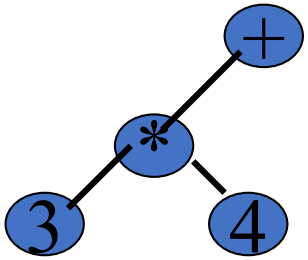
4+5*7*9+8　构建根节点



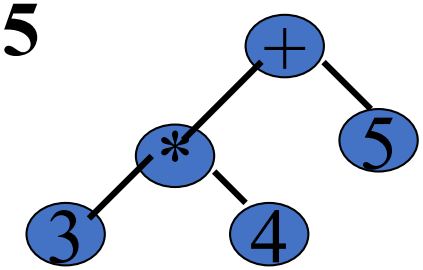
4+5*7*9+8　构建右节点4

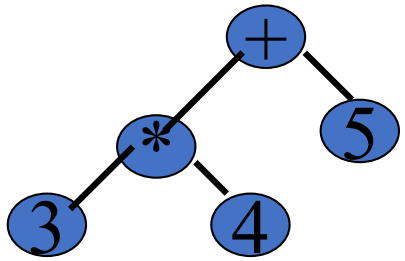


+5*7*9+8　构建根节点+，原树作为左子树

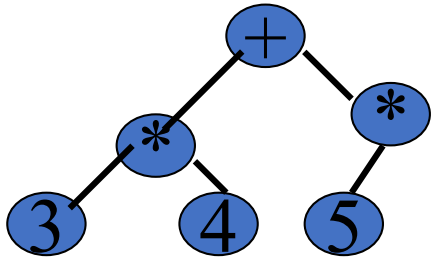


5*7*9+8　构建右节点5

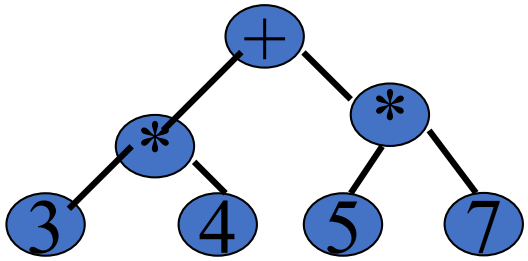




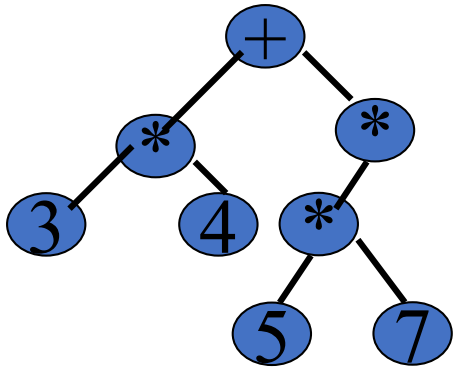
*7*9+8　下移到右节点，构建根节点*，原来的右节点作为它的左节点



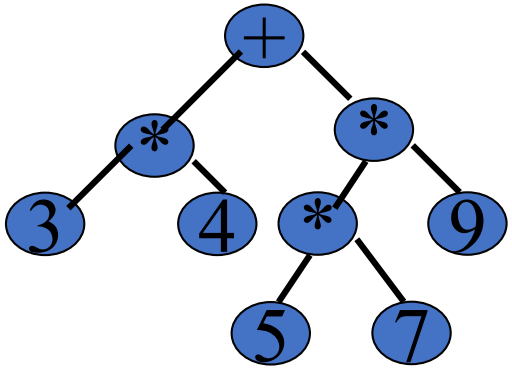
7*9+8　构建右节点7



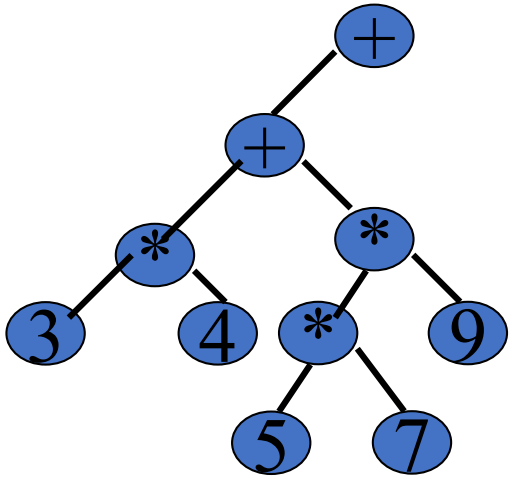
9+8　创建根，原树作为左子树



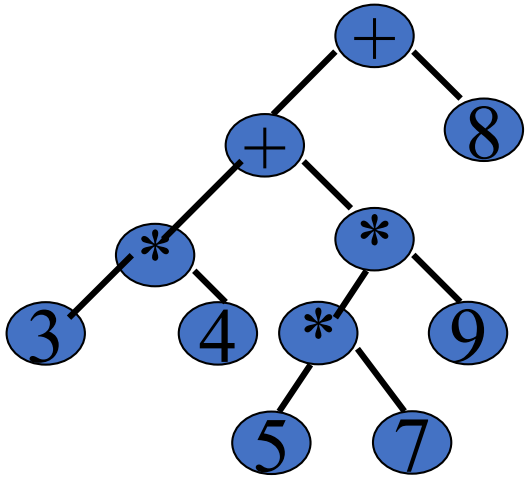
9+8　9作为右子树



+8　上移到根。创建根+，原树作为左子树



8　8作为左节点



构建过程总结

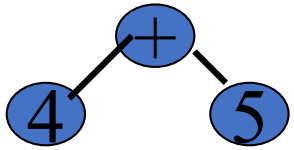
- **顺序扫描中缀表达式**
- **当扫描到的是运算数：先检查当前的表达式树是否存在。如果不存在，则表示扫描到的是第一个运算数，将它作为树根。如果树存在，则将此运算数作为前一运算符的右孩子。**
- **如果扫描到的是+或-：将它作为根结点，原来的树作为它的左子树。**

构建过程总结　续

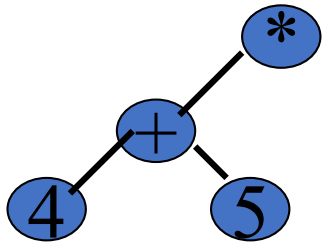
- **如果扫描到的是*或/：则与根结点比较。如果根结点也是*或/，则根结点应该先执行，于是将当前运算符作为根结点，原来的树作为左子树。如果根结点是+或-，则当前运算符应该先运算，于是将它作为右子树的根，原来的右子树作为它的左子树。**
- **在遇到运算数时，如何知道它前面的运算符是谁？这只需要判别根结点有没有右孩子。如果没有右孩子，则运算数是根结点的右运算数，否则就是根结点右孩子的右运算数。**

构建过程（括号的处理）

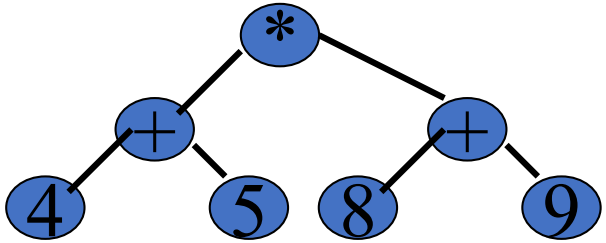
(4+5)*(8+9)+10 遇到括号，将括号内的子表达式构建一棵子树作为整个表达式的一个运算数



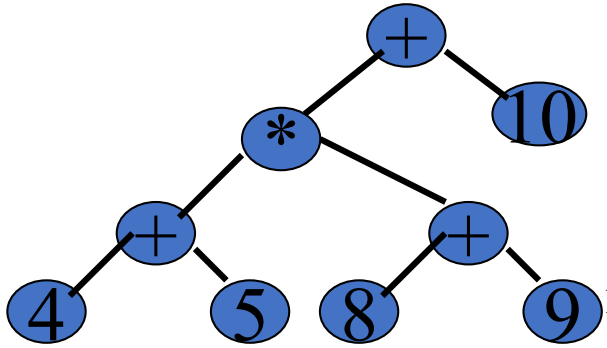
***(8+9)+10 *作为根节点，括号内的子树作为左子树**



(8+9)+10括号内的子表达式构建一棵子树作为整棵树的右子树



+10 +作为根节点，原树作为左子树，10作为右子树



表达式树类的设计

- **数据成员：指向树根节点的指针**
- **公有成员函数：**
  - **构造函数：调用create从表达式构建一棵树**
  - **result：计算表达式的结果，用后序遍历过程**
- **私有成员函数：**
  - **Create**
  - **带有递归参数的result函数**
  - **getToken：create函数所用的子函数，用于从表达式中获取一个语法单位**

结点的设计

- **在表达式树中，每个叶子结点保存的是一个运算数，每个非叶结点保存的是一个运算符。**
- **结点的数据部分应该包括两个部分：结点的类型和值。**

calc类的定义

```
class calc {
  enum Type {  DATA, ADD, SUB, MULTI, DIV, OPAREN,
                    CPAREN, EOL  };
  struct node {
      Type type;
      int data;
      node *lchild, *rchild;

      node(Type t, int d = 0, node *lc = NULL, node *rc = NULL)
            {type = t; data = d; lchild = lc; rchild = rc;}
      };

      node *root;
```

```
public:
    calc( char *s ) { root = create( s ); }
    int result()
        { if ( root == NULL ) return 0;
          return result( root );}
private :
        node *create( char *&s );
        Type getToken( char *&s, int &value );
       int result( node *t );
};
```

私有create函数的实现

```
calc::node  *calc::create(char *&s)
{calc::node *p, *root = NULL;
 Type returnType;
 int value;
 while (*s)
   { returnType = getToken(s, value);
     switch (returnType)
       { case DATA: case OPAREN:
         if (returnType == DATA)
                  p = new node(DATA, value);
           else   p = create(s);
         if (root != NULL)
              if (root->rchild == NULL) root->rchild = p;
                   else root->rchild->rchild = p;
         break;
```

```
  case CPAREN: case EOL:  return root;
  case ADD:  case SUB:
      if (root == NULL) root = new node(returnType,0, p);
       else root = new node(returnType,0, root);
      break;
  case MULTI:  case DIV:
      if (root == NULL) root = new node(returnType,0, p);
       else if (root->type == MULTI || root->type == DIV)
                    root = new node(returnType,0, root);
              else root->rchild = new node
                                  (returnType,0, root->rchild);
    }
 }
 return root;
}
```

getToken

```
calc::Type calc::getToken(char *&s, int &data)
{ char type;
  while (*s == ' ') ++s;
  if (*s >= '0  ‘  &&  *s <= '9')
    { data = 0;
       while (*s >= '0'  &&  *s <= '9')
            { data = data * 10 + *s - '0'; ++s;}
          return DATA;
     }
```

```
if (*s == '\0') return EOL;
  type = *s; ++s;
  switch(type)
      { case '+': return ADD;
        case '-': return SUB;
        case '*': return MULTI;
        case '/': return DIV;
        case '(': return OPAREN;
        case ')': return CPAREN;
        default: return EOL;
      }
 }
```

私有的result函数的实现

```
int calc::result(calc::node *t)
{int num1, num2;
 if (t->type == DATA) return t->data;
 num1 = result(t->lchild);
 num2 = result(t->rchild);
 switch(t->type)
    { case ADD: t->data = num1 + num2;  break;
      case SUB:  t->data = num1 - num2; break;
      case MULTI:  t->data = num1 * num2; break;
      case DIV:  t->data = num1 / num2;
    }
 return t->data;
}
```

Calc类的应用

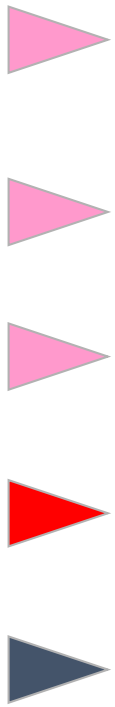
```
int main()
{calc exp(" 2*3+(1 * 2*3+6*6) * (2+3)/5 + 2/2   ");
 cout << exp.result() << endl;
 return 0;
}
```

Calc类的特点

- **使用时和基于栈实现的calc类完全一样**
- **缺点**
  - **没有考虑表达式不正确的情况**
  - **没有考虑乘方运算**

第五章 树

- **树的概念**
- **二叉树**
- **表达式树**
- **哈夫曼树与哈夫曼编码**
- **树和森林**



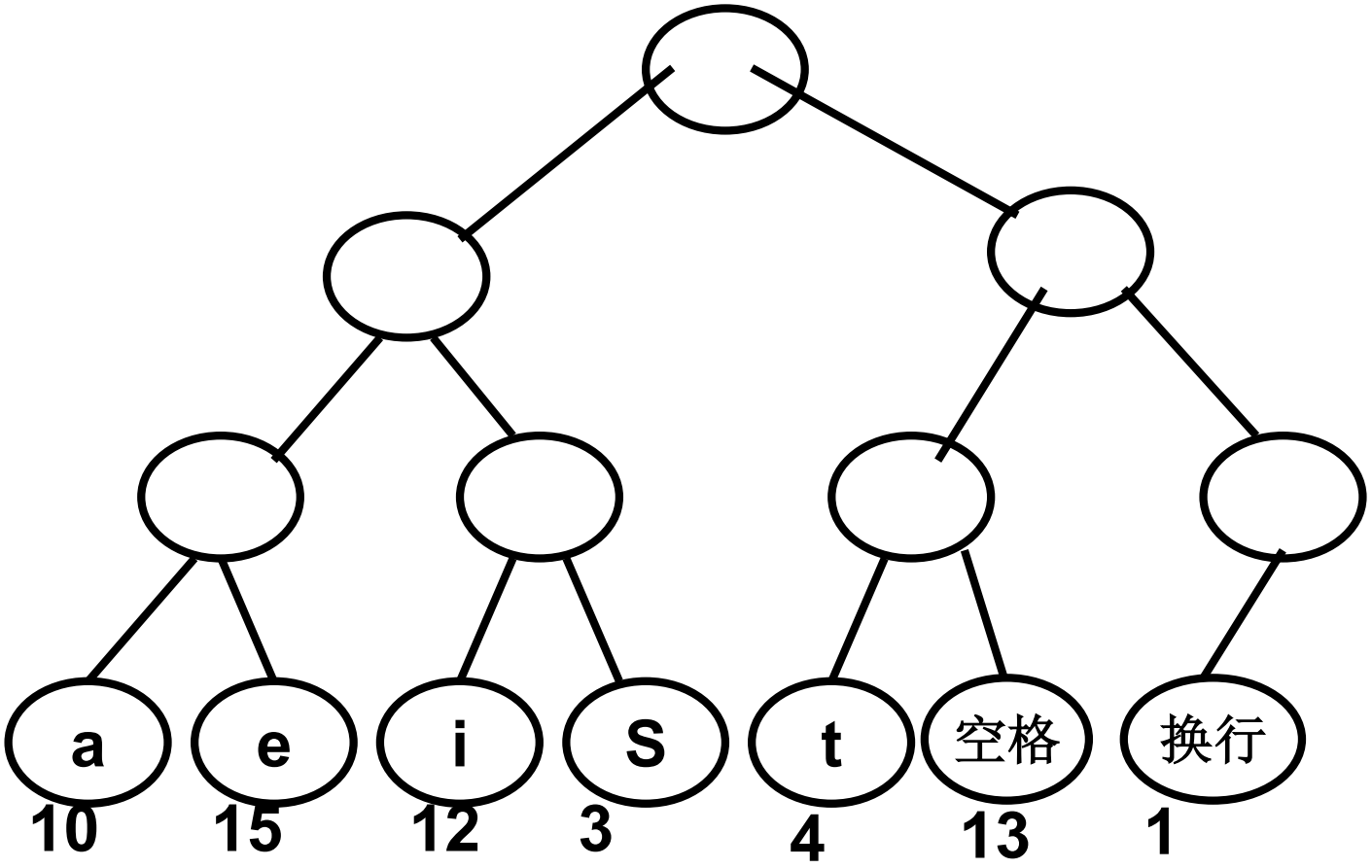
字符的机内表示

- **在计算机中每个字符是用一个编码表示**
- **大多数编码系统都采用等长编码，如ASCII编码**
- **例如在某段文本中用到了下列字符，括号中是它们出现的频率：a(10), e(15), i(12), s(3), t(4), 空格(13), 换行(1)。如采用定长编码，7个不同的字符至少要用3位编码。**

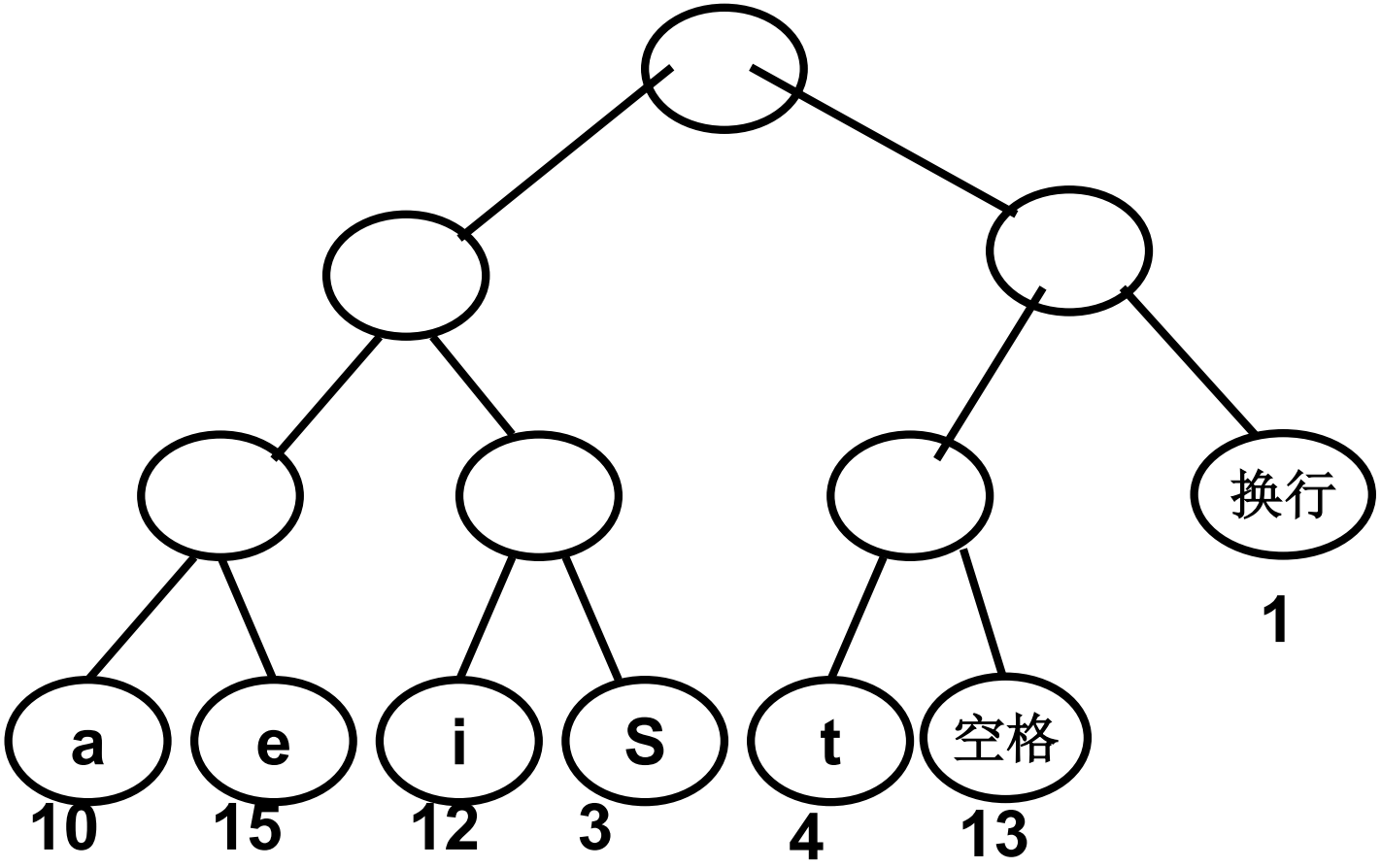
字符	编码	出现频率	占用空间（bit）
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
空格	101	13	39
换行	110	1	3

总存储量：3*（10+15+12+3+4+13+1）= 3*58 = 174 bit

这个编码可以对应成如下的完全二叉树，左枝为0，右枝为1



很显然，将换行上移一层可以减少存储量



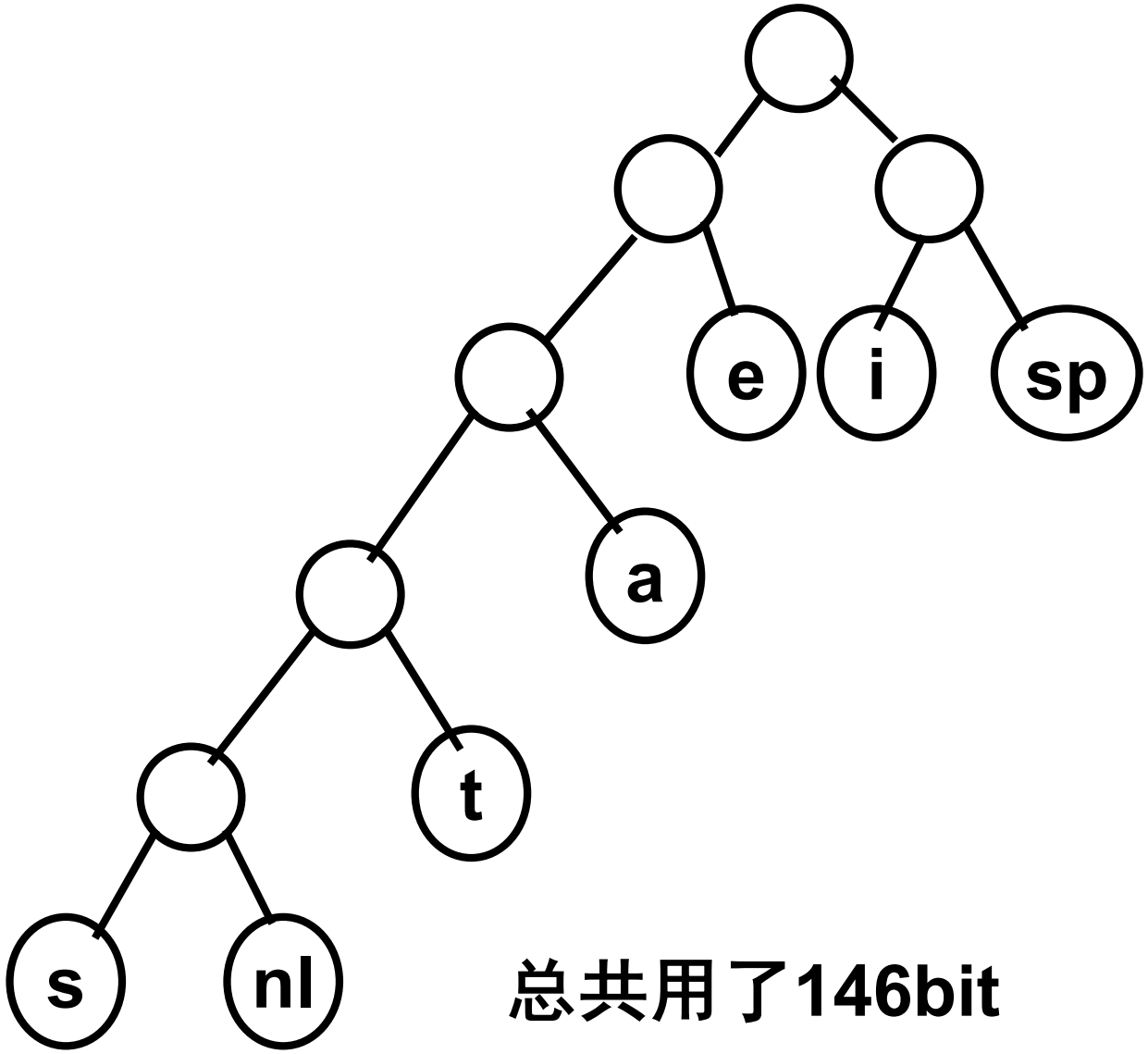
不等长编码可以减少存储量！！！

前缀编码

- **字符只放在叶结点中**
- **字符编码可以有不同的长度**
- **由于字符只放在叶结点中，所以每个字符的编码都不可能是其他字符编码的前缀**
- **前缀编码可被惟一解码**

哈夫曼树

- **哈夫曼树是一棵最小代价的二叉树，在这棵树上，所有的字符都包含在叶结点上。**
- **要使得整棵树的代价最小，显然权值大的叶子应当尽量靠近树根，权值小的叶子可以适当离树根远一些。**



总共用了146bit

哈夫曼编码

A	001
E	01
I	10
S	00000
T	0001
Sp	11
nl	00001

哈夫曼算法

1、给定一个具有n个权值{ $w_1, w_2, \ldots\ldots w_n$ }的结点的集合

F = { $T_1, T_2, \ldots\ldots T_n$ }

2、 初始时，设集合 A = F。

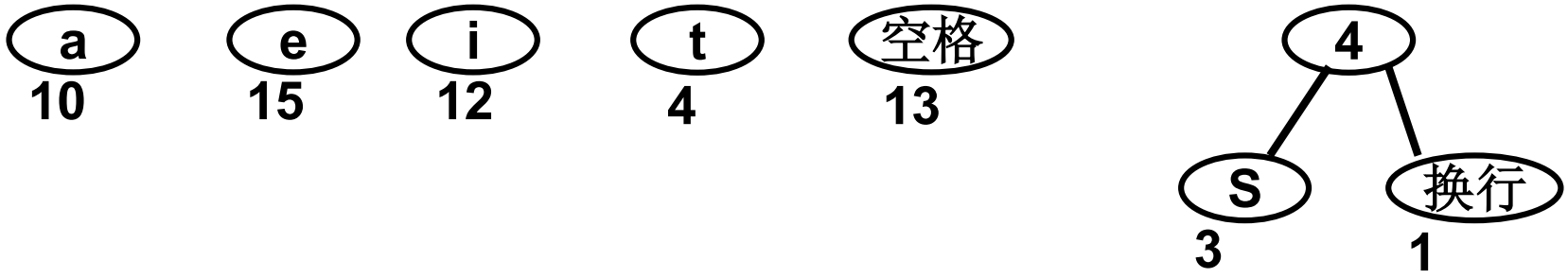
3、 执行 i = 1 至 n -1 的循环，在每次循环时执行以下操作

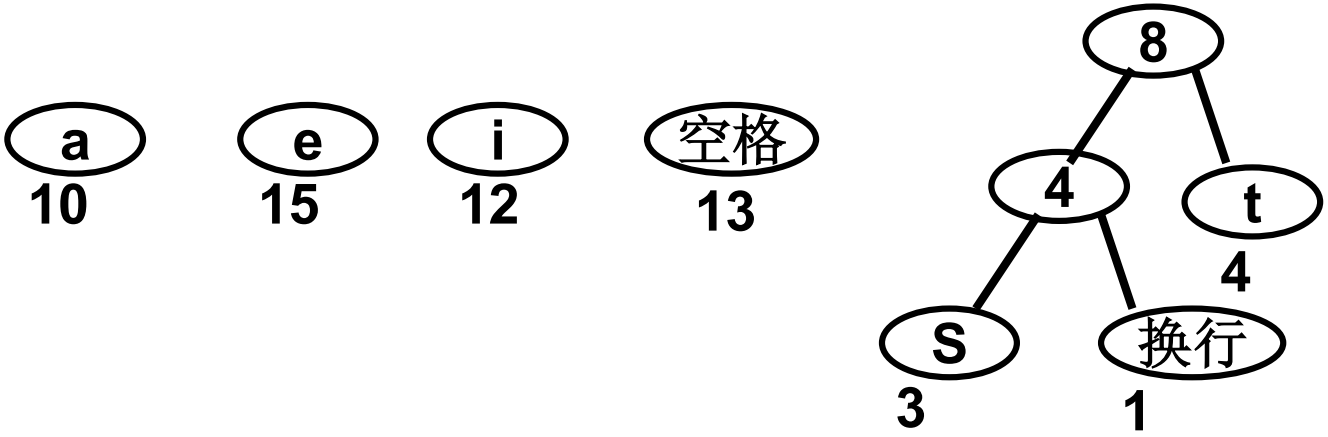
从当前集合中选取权值最小、次最小的两个结点，以这两个结点作为内部结点 b_i 的左右儿子，b_i 的权值为其左右儿子权值之和。

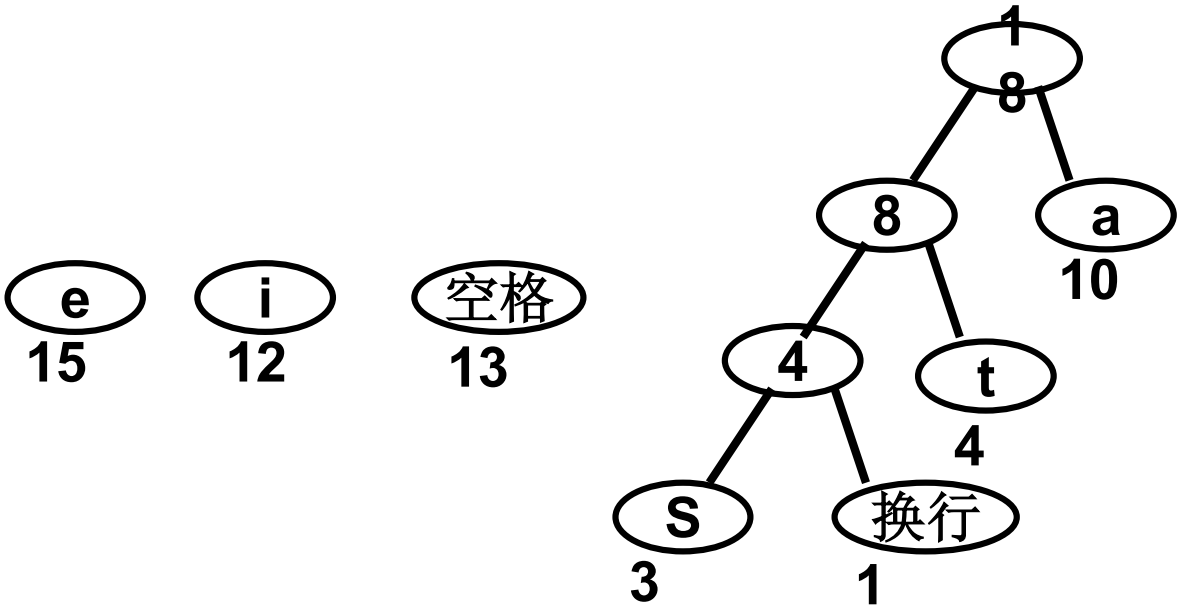
在集合中去除这两个权值最小、次最小的结点，并将内部结点 b_I 加入其中。这样，在集合A中，结点个数便减少了一个。

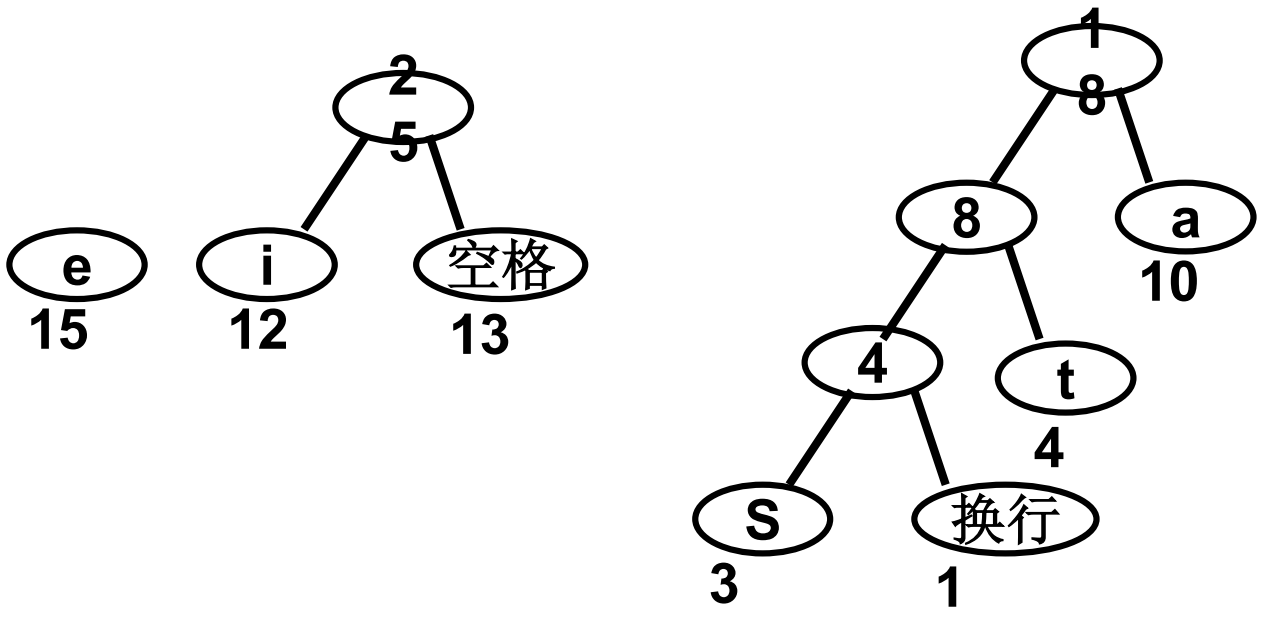
这样，在经过了**n-1** 次循环之后，集合**A**中只剩下了一个结点，这个结点就是根结点。

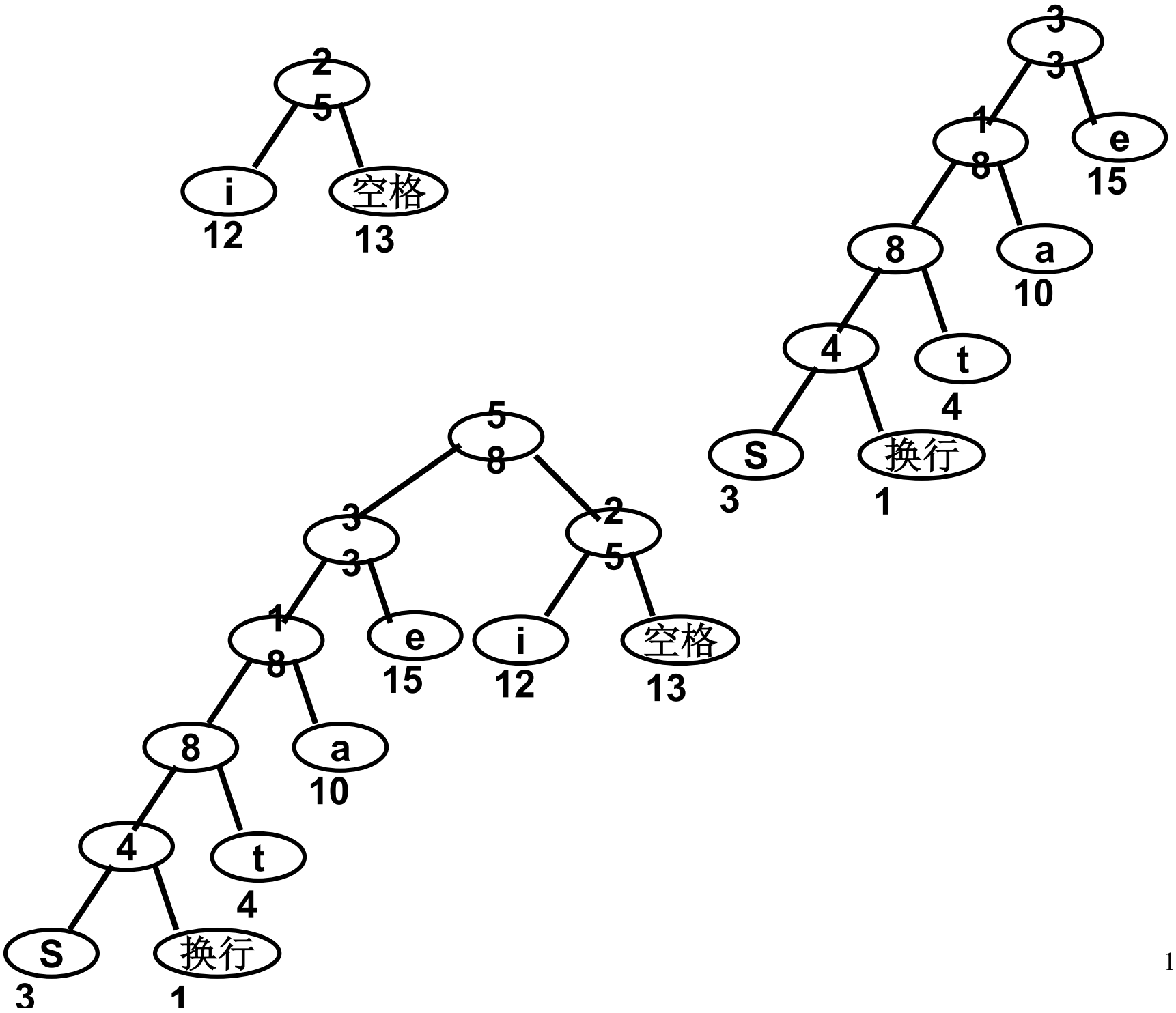
a(10), e(15), i(12), s(3), t(4), 空格(13), 换行(1)。






e
15
i
12
空格
13
1
8
8
a
10
4
t
4
S
3
换行
1


2
5
e
15
i
12
空格
13
1
8
8
a
10
4
t
4
S
3
换行
1


2
5
i
12
空格
13
3
3
1
8
e
15
8
a
10
4
t
4
S
3
换行
1
5
8
3
3
2
5
1
8
e
15
i
12
空格
13
8
a
10
4
t
4
S
3
换行
1

哈夫曼编码的生成

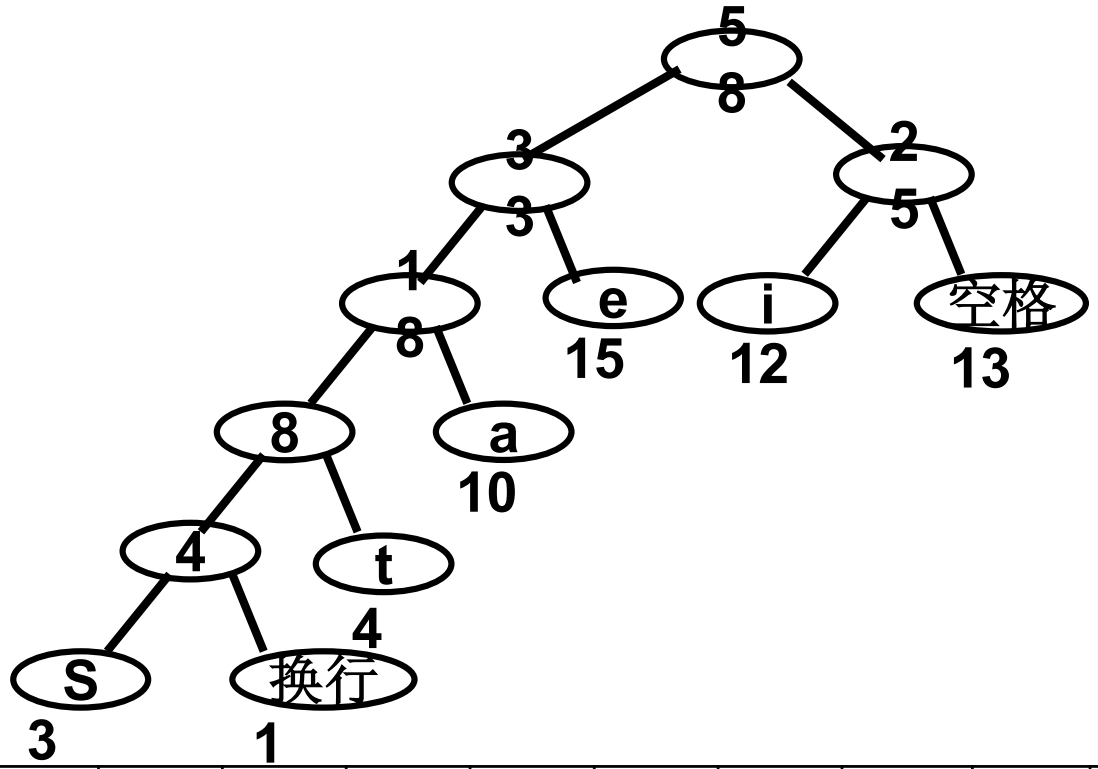
- **每个字符的编码是根节点到该字符的路径**
- **左枝为0，右枝为1**

哈夫曼树类的实现

- **为了便于找出一组符号的哈夫曼编码，我们可以定义一个哈夫曼树类。**
- **哈夫曼树类的对象可以接受一组符号以及对应的权值，并告知每个符号对应的哈夫曼编码。因此，哈夫曼树类应该有两个公有的成员函数：**
  - **构造函数：接受一组待编码的符号以及它们的权值，构造一棵哈夫曼树。**
  - **GetCode函数根据保存的哈夫曼树为每个叶结点生成哈夫曼编码。**

哈夫曼树的存储

- **在哈夫曼树中，每个要编码的元素是一个叶结点，其它结点都是度数为2的节点**
- **一旦给定了要编码的元素个数，由$n_0 = n_2 + 1$可知哈夫曼树的大小为2n-1**
- **哈夫曼树可以用一个大小为2n的数组来存储。0节点不用，根存放在节点1。叶结点依次放在n+1到2n的位置**
- **每个数组元素保存的信息：结点的数据、权值和父结点和左右孩子的位置**



值								a	e	i	s	t	sp	nl
权		58	33	25	18	8	4	10	15	12	3	4	13	1
父			1	1	2	4	5	4	2	3	6	5	3	6
左		2	4	9	5	6	10							
右		3	8	12	7	11	13							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

生成过程

值								a	e	i	s	t	sp	nl
权		58	33	25	18	8	4	10	15	12	3	4	13	1
父			1	1	2	4	5	4	2	3	6	5	3	6
左		2	4	9	5	6	10							
右		3	8	12	7	11	13							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

编码的产生

对每个结点，从叶子往根推进，是左枝加0，是右枝加1

值								a	e	i	s	t	sp	nl
权		58	33	25	18	8	4	10	15	12	3	4	13	1
父			1	1	2	4	5	4	2	3	6	5	3	6
左		2	4	9	5	6	10							
右		3	8	12	7	11	13							
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

生成a的代码：结点4的右孩子（1），结点4是结点2的左孩子（01），结点2是结点1的左孩子（001）

哈夫曼树类

- **存储设计**
  - **结点的表示：结点的数据、权值和父结点和左右孩子的位置**
  - **哈夫曼树的存储：一个结点数组以及一个整型数据成员，保存数组的大小。**
- **操作**
  - **构建一棵哈夫曼树：构造函数实现。**
    - **给出节点数据数组，权值数组和数据个数**
  - **获取树上节点的哈夫曼编码**
    - **返回一个数组，数组的元素由数据和编码两部分组成的**

```
template <class Type>
class hfTree{
private:
    struct  Node
  {  Type data; //结点值
       int weight;   //结点的权值
       int parent, left, right;
   };

    Node *elem;
    int length;
```

```
  public:
      struct hfCode {
         Type data;
         string code;
       };

      hfTree(const Type *x, const int *w, int size);
      void getCode(hfCode result[ ]);
      ~hfTree() {delete [ ] elem;}
};
```

构造函数

```
template <class Type>
hfTree<Type>::hfTree(const Type *v, const int *w, int size)

{  const int MAX_INT = 32767;
   int min1, min2;    //最小树、次最小树的权值
   int x, y;     //最小树、次最小树的下标

   //置初值
   length = 2 * size;
   elem = new Node[length];
   for (int i = size; i < length; ++i)
       { elem[i].weight = w[i - size];
         elem[i].data = v[i - size];
        elem[i].parent = elem[i].left = elem[i].right = 0;
       }
```

```
// 构造新的二叉树
 for (i = size - 1; i > 0; --i)
   { min1 = min2 = MAX_INT; x = y = 0;
     for (int j = i + 1; j < length; ++j)
       if (elem[j].parent == 0)
           if (elem[j].weight < min1)
                   { min2 = min1; min1 = elem[j].weight;
                     x = y; y = j; }
                 else if (elem[j].weight < min2)
                         { min2 = elem[j].weight; x = j; }
       elem[i].weight = min1 + min2;
       elem[i].left = x; elem[i].right = y; elem[i].parent = 0;
       elem[x].parent = i; elem[y].parent = i;
    }
}
```

getCode的伪代码

```
getCode(hfCode<Type> result[])
 {for (int i = size; i < length; ++i)
   {result[i - size].data = elem[i].data;
    result[i - size].code = "";
    p = elem[i].parent; s = i;
  while (p不等于0) {
       if (p的左孩子是 == s) result[i - size].code 前添加 ‘0’ ;
          else result[i - size].code 前添 ‘1’ ;
       移到上一层 ;
  }
   }
}
```

getCode代码

```
template <class Type>
void hfTree<Type>::getCode(hfCode result[])
 { int size = length / 2;
   int p,s; // s是追溯过程中正在处理的结点，p是s的父结点下标
   for (int i = size; i < length; ++i)
     {  result[i - size].data = elem[i].data;
        result[i - size].code = "";
        p = elem[i].parent; s = i;
          while (p) {
           if (elem[p].left == s)
                  result[i - size].code = '0' + result[i - size].code;
              else result[i - size].code = '1' + result[i - size].code;
           s = p; p = elem[p].parent;
          } } }
```

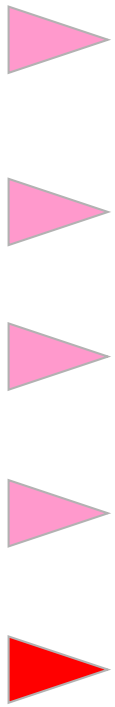
哈夫曼类的使用

- **为下列符号集生成哈夫曼编码：a(10), e(15), i(12), s(3), t(4), d(13), n(1)**

```
int main()
{ char ch[] = {"aeistdn"};
  int w[] = {10,15,12,3,4,13,1};
  hfTree<char> tree(ch, w, 7);
  hfTree<char>::hfCode result[7];
  tree.getCode(result);
  for (int i=0; i< 7; ++i)
      cout << result[i].data << ' '
           << result[i].code << endl;
  return 0;
}
```

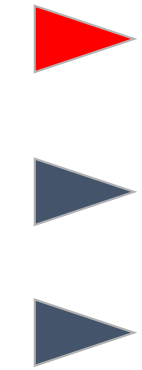
第五章 树

- **树的概念**
- **二叉树**
- **表达式树**
- **哈夫曼树与哈夫曼编码**
- **树和森林**



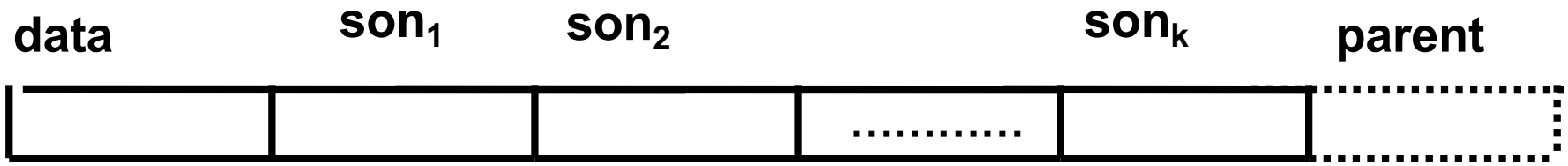
树和森林

- 树的存储实现
- 树的遍历
- 树、森林和二叉树



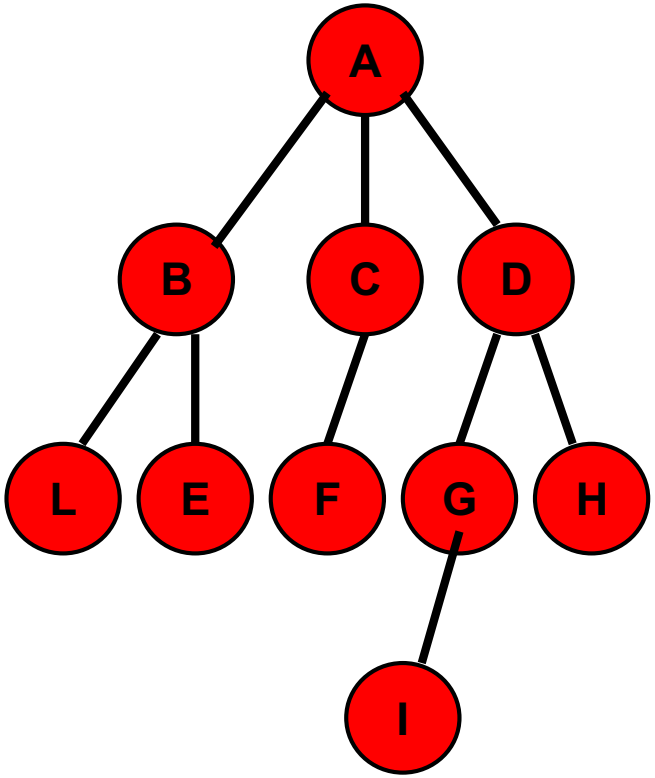
树的存储结构

标准形式：树中的每个结点除有数据字段之外，还有 **K** 个指针字段；其中 **K** 为树的度。



•广义标准形式：在标准形式的基础上，再增加指向父亲结点的指针场。

E.g: 度数 K ＝ 3 的树的广义标准存储



	值	s1	s2	s3	p
0	A	1	2	3	-1
1	B	4	5	-1	0
2	C	6	-1	-1	0
3	D	7	8	-1	0
4	L	-1	-1	-1	1
5	E	-1	-1	-1	1
6	F	-1	-1	-1	2
7	G	9	-1	-1	3
8	H	-1	-1	-1	3
9	I	-1	-1	-1	7

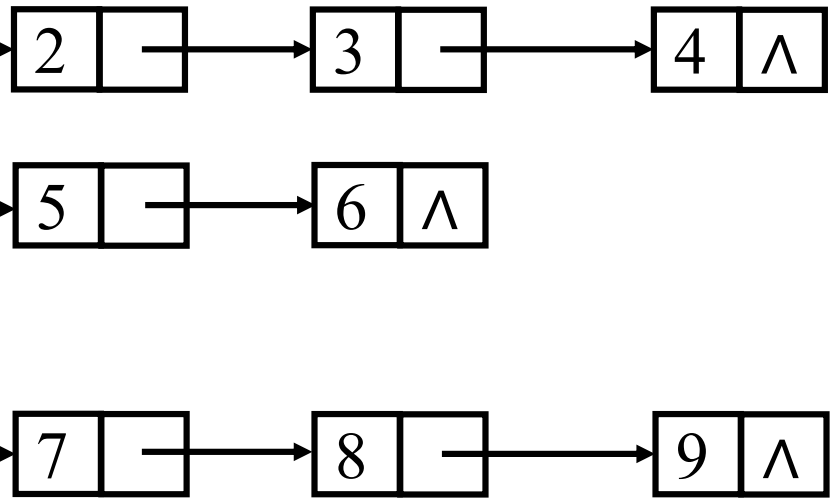
-1 表示空

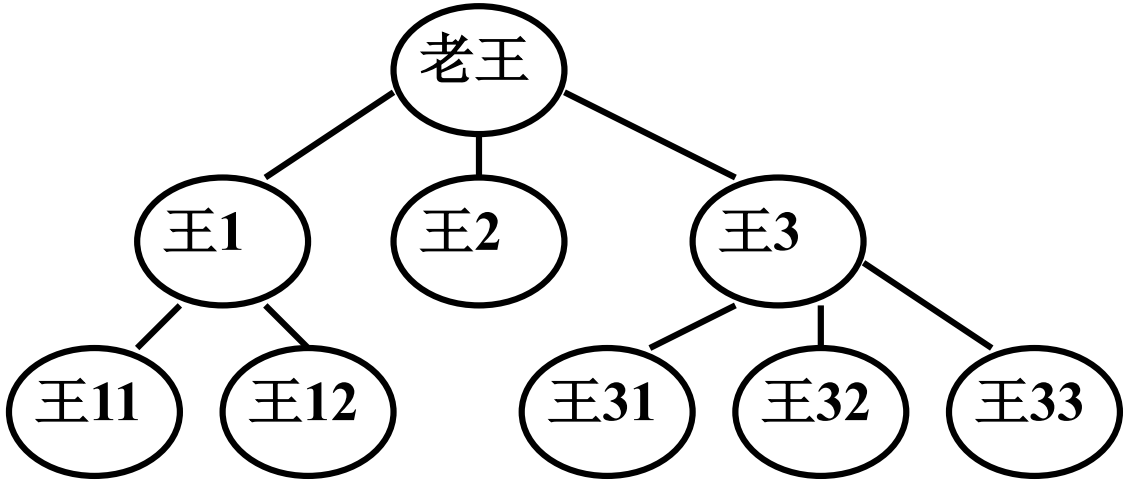
缺点：空指针字段太多，多达 (K -1)× n + 2 个。

孩子链表示法

- **将每个结点的所有孩子组织成一个链表。**
- **树的节点由两部分组成：**
  - **存储数据元素值的数据部分**
  - **指向孩子链的指针**
- **如果树的所有结点存放在一个数组中，这个数组称为表头数组。这种存储方式称为静态的孩子链表。**
- **将树的所有结点组织成一个链表，称为动态的孩子链表**

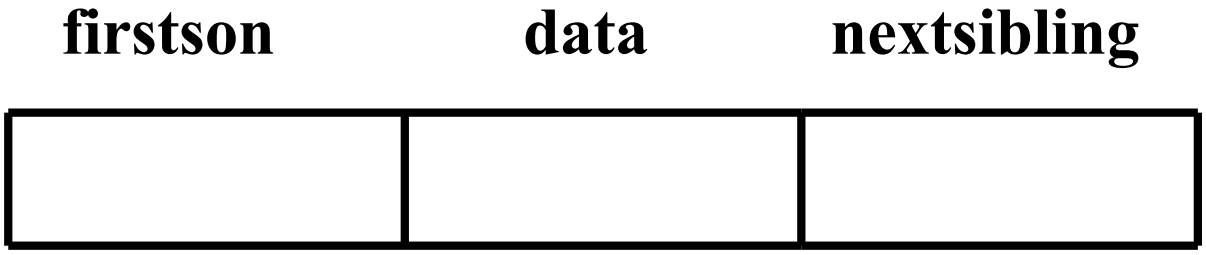
	数据	指针
1	老王	
2	王1	
3	王2	∧
4	王3	
5	王11	∧
6	王12	∧
7	王31	∧
8	王32	∧
9	王33	∧

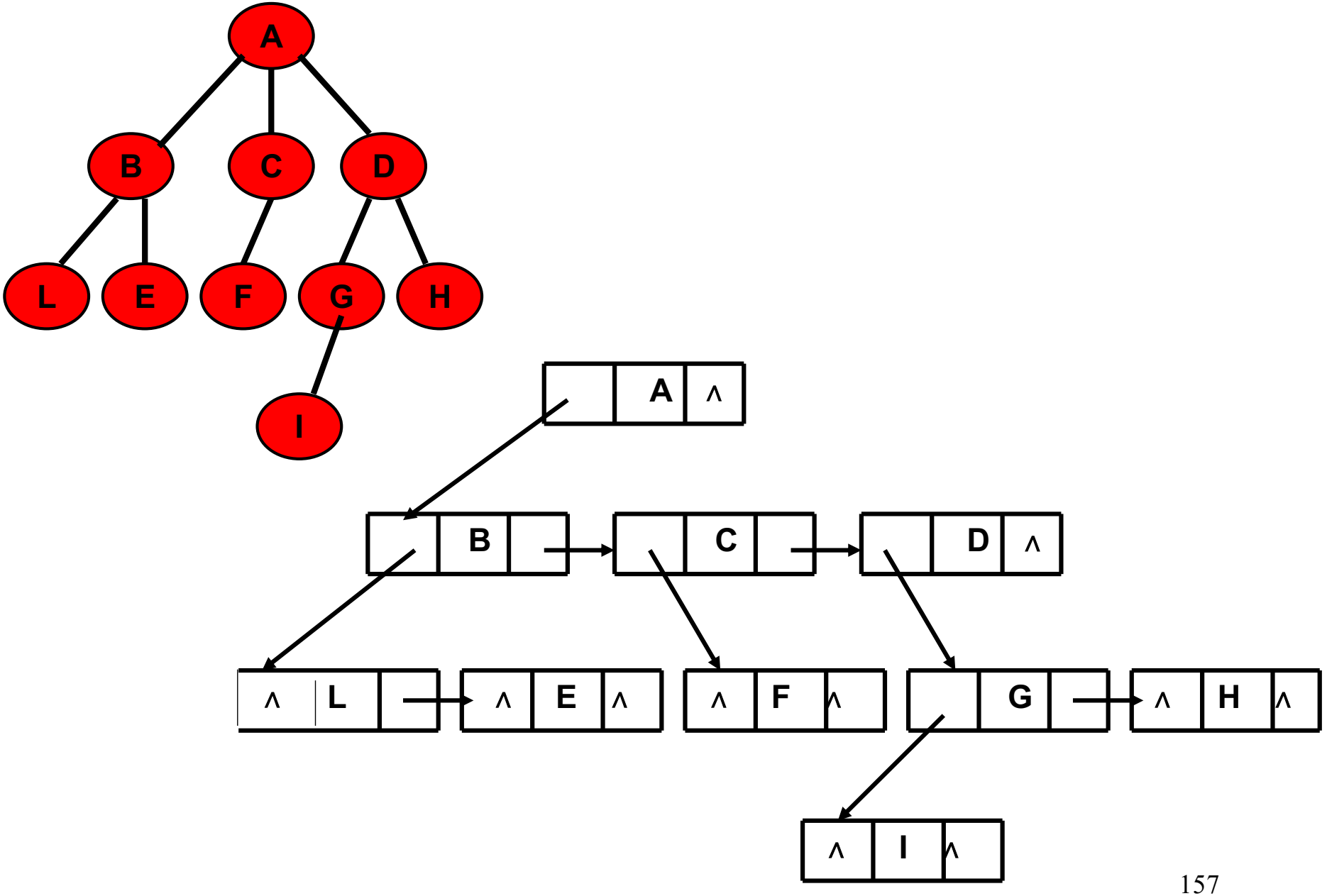




孩子兄弟链表示法

实质上是用二叉树表示一棵树。树中的每个结点有数据字段、指向它的第一棵子树树根的指针字段、指向它的兄弟结点的指针字段。

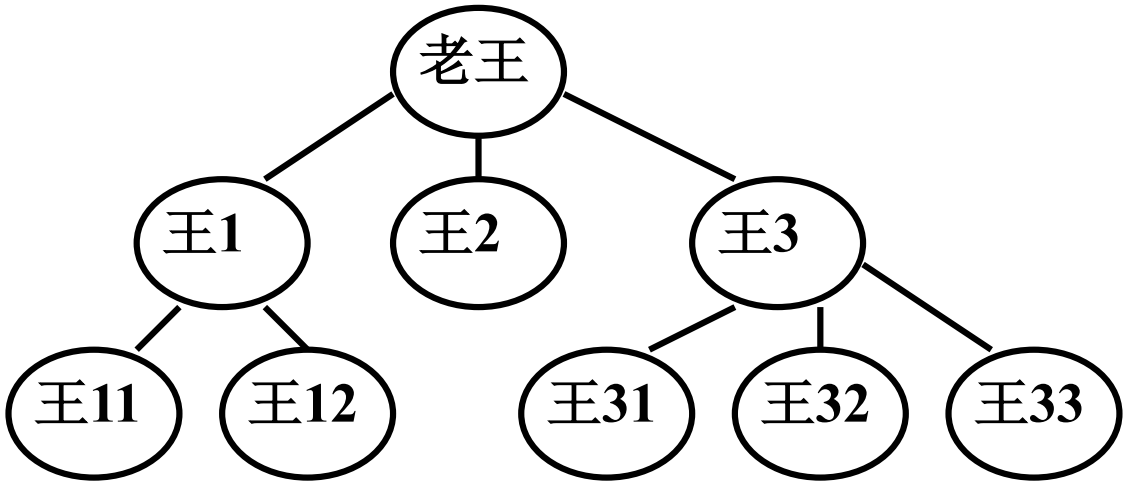



A
B
C
D
L
E
F
G
H
I
A
^
B
C
D
^
^
L
^
E
^
^
F
^
G
^
H
^
^
I
^

双亲表示法

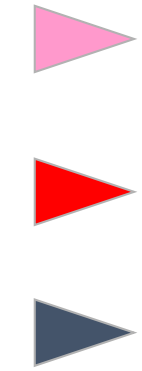
- **通过指向父结点的指针将树中所有的结点组织在一起**
- **在双亲表示法中，每个结点由两部分组成：**
  - **存储数据元素的数据字段**
  - **存储父结点位置的父指针字段**
- **这种表示法对求指定结点的祖先的操作很方便，但对求指定结点的子孙则不方便。只适合某些应用，如不相交集的存储**

	数据	父结点
1	老王	0
2	王1	1
3	王2	1
4	王3	1
5	王11	2
6	王12	2
7	王31	4
8	王32	4
9	王33	4



树和森林

- 树的存储实现
- 树的遍历
- 树、森林和二叉树



前序遍历

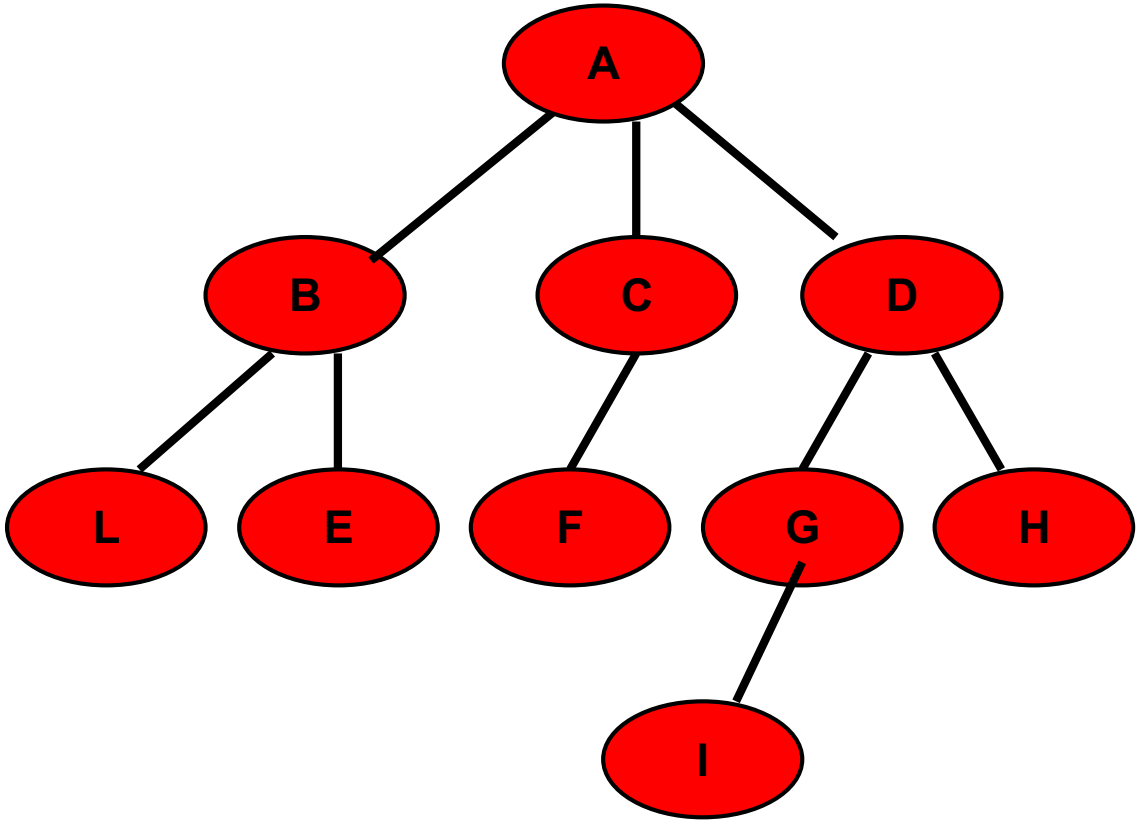
- **访问根结点；**
- **前序遍历根结点的第一棵子树、前序遍历它的第二棵子树、……、前序遍历它的最后一棵子树。**

后序遍历

- **后序遍历根结点的第一棵子树、后序遍历它的第二棵子树、……、后序遍历它的最后一棵子树。**
- **访问根结点；**

层次遍历

- **访问根结点；**
- **若第i层已被访问，且第i+1层的结点尚未被访问，则从左到右依次访问第i+1层的结点。**



前序：A、B、L、E、C、F、D、G、I、H

后序：L、E、B、F、C、I、G、H、D、A

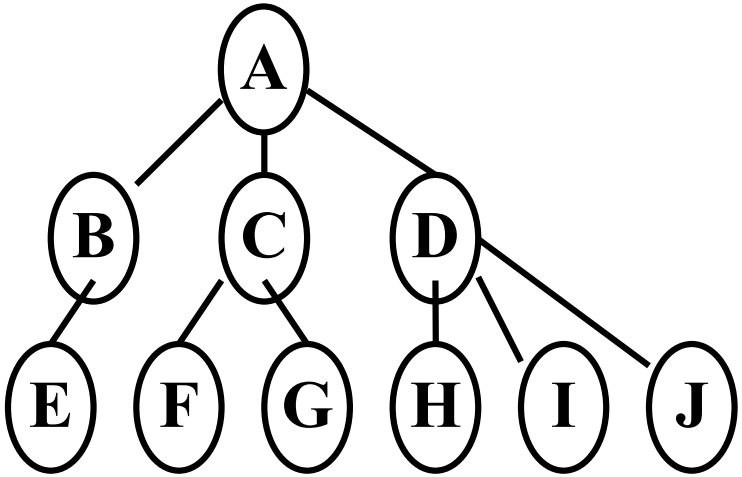
层次：A、B、C、D、L、E、F、G、H、I

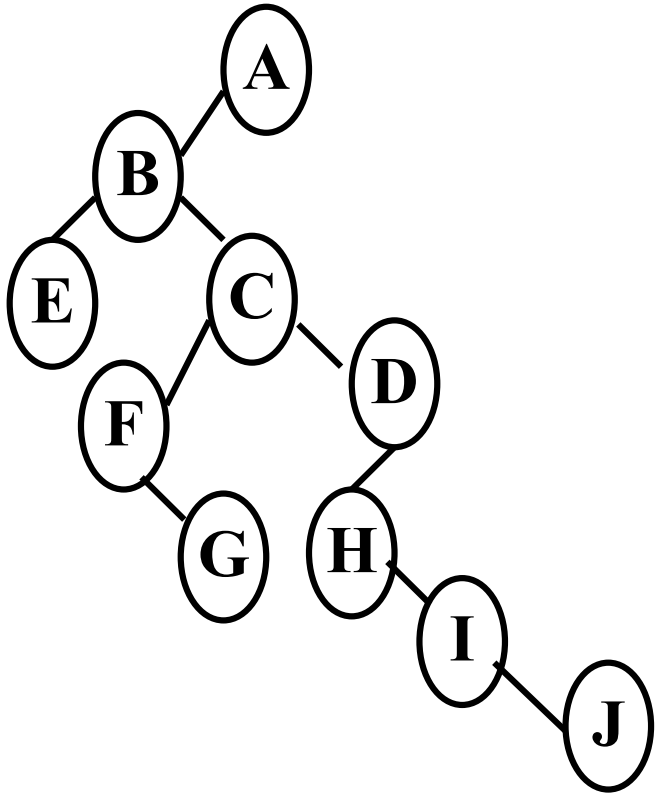
树和森林

- 树的存储实现
- 树的遍历
- 树、森林和二叉树

树、森林和二叉树

- 二叉树是一种结构最简单、运算最简便的树形结构。但对很多问题来讲，其自然的描述形态是树或森林。
- 树的孩子兄弟链表示法就是将一棵树表示成二叉树的形态，这样就可以将二叉树中的许多方法用在树的处理中。


A
B
C
D
E
F
G
H
I
J

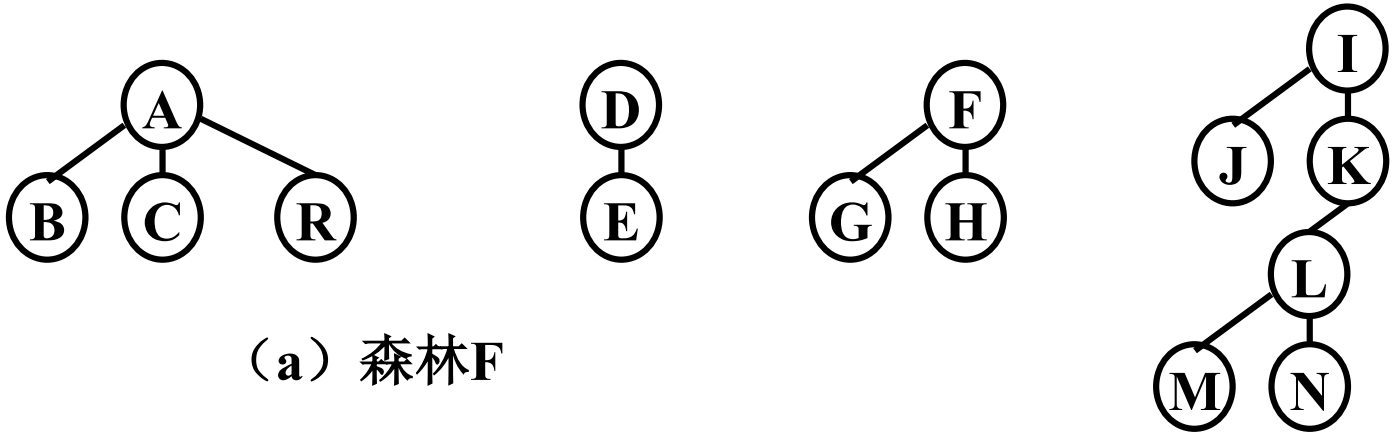

A
B
E
C
F
D
G
H
I
J

森林

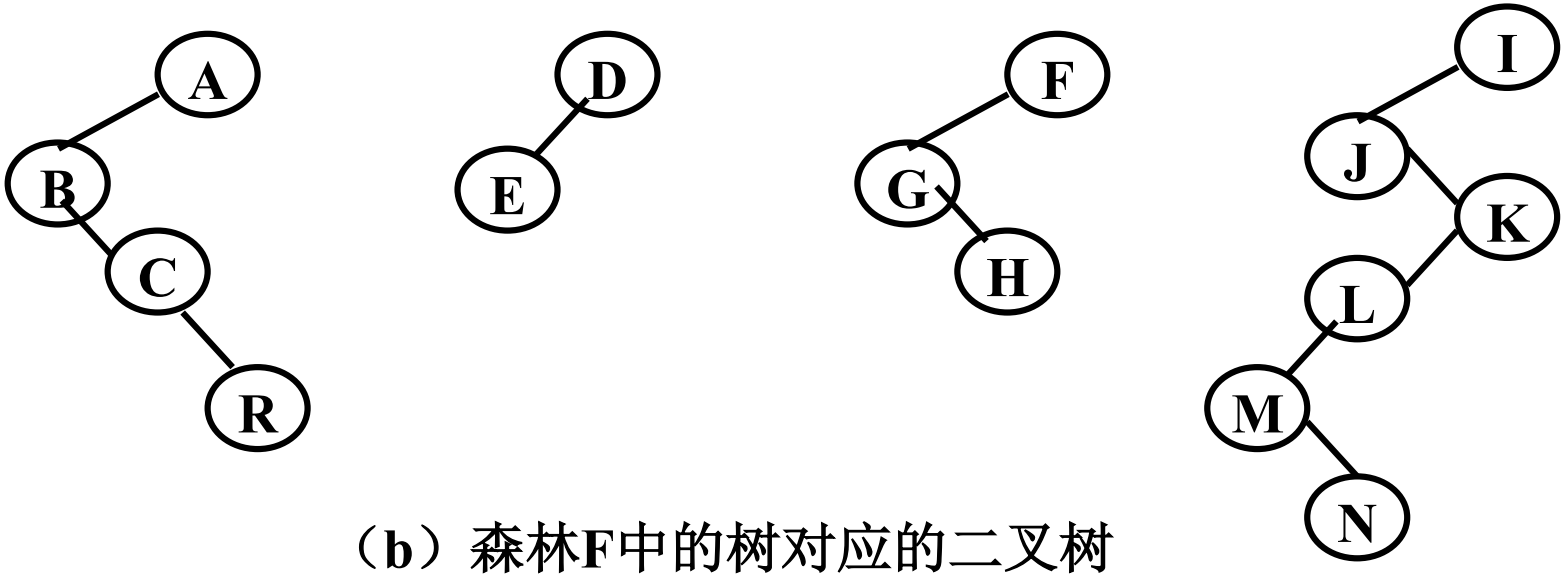
- **森林通常定义为树的集合或树的序列。**
- **森林的存储：存储一个森林要包括两方面的内容**
  - **存储森林中的每一棵树**
  - **表示这些树是属于同一个森林。**

森林的二叉树存储

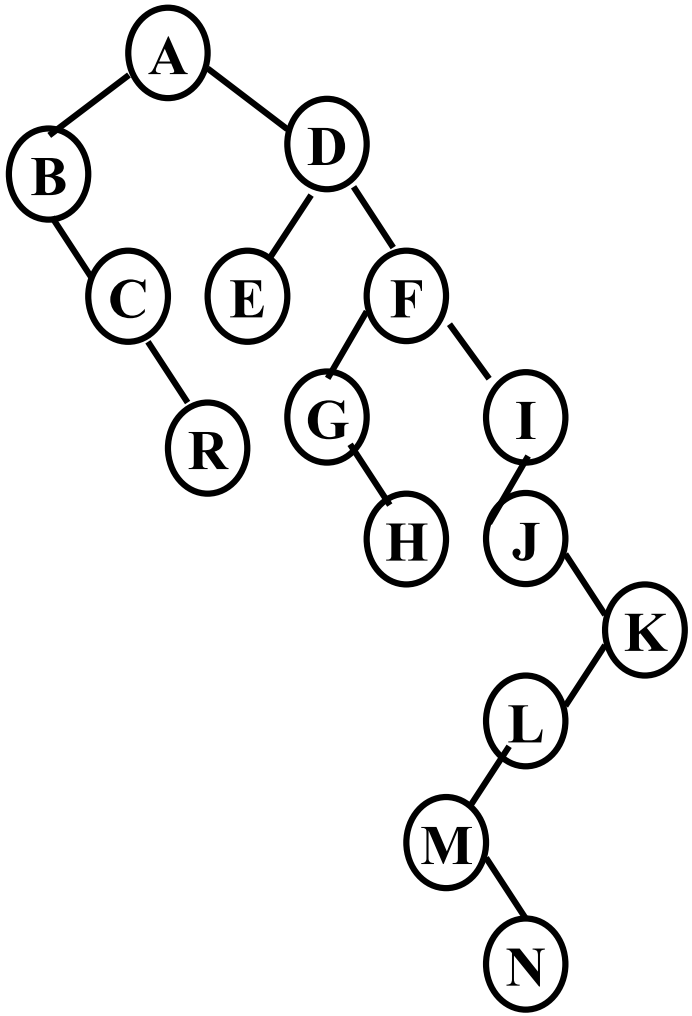
- **将每棵树Ti转换成对应的二叉树Bi；**
- **将Bi作为Bi-1根结点的的右子树。**



（a）森林F



（b）森林F中的树对应的二叉树



森林F中的树对应的二叉树

总结

- **本章是数据结构的重点之一，也是本书许多后续章节的基础。**
- **本章主要介绍了树形结构的基本概念，详细讨论了一种重要的数据结构 — 二叉树的设计和实现。在此基础上介绍了二叉树的两种应用 — 表达式树和哈夫曼树。**
- **最后，本章还介绍了如何处理普通的树形结构以及森林。普通的树形结构和森林的处理方法是将它们转换成二叉树来处理**

作业

第7章 优先级队列

- **基本的优先级队列**
- **二叉堆**
- **D堆**
- **归并优先级队列**
- **STL中的优先级队列**
- **排队系统的模拟**

基本的优先级队列

- 结点之间的关系是由结点的优先级决定的，而不是由入队的先后次序决定。优先级高的先出队，优先级低的后出队。这样的队列称为优先级队列。
- 优先级队列的操作与普通的队列相同

优先级队列的简单实现

- **利用现有的队列结构。有两种方法可以处理优先级**
  - **入队时，按照优先级在队列中寻找合适的位置，将新入队的元素插入在此位置。出队操作的实现保持不变。**
  - **入队操作的实现保持不变，将新入队的元素放在队列尾。但出队时，在整个队列中查找优先级最高的元素，让它出队。**

时间性能分析

- **第一种方案的入队操作的时间复杂度是O（N），出队操作的时间复杂度是O（1）。**
- **第二种方案的入队操作的时间复杂度是O（1），出队操作的时间复杂度是O（N）。**

第6章 优先级队列

- **基本的优先级队列**
- **二叉堆**
- **D堆**
- **归并优先级队列**
- **STL中的优先级队列**
- **排队系统的模拟**

二叉堆

堆是一棵完全二叉树，且满足下述关系之一

$k_i \le k_{2i}$ 且 $k_i \le k_{2i+1}$ $(i=1,2,\ldots,\lfloor n/2 \rfloor)$

或者：

$k_i \ge k_{2i}$ 且 $k_i \ge k_{2i+1}$ $(i=1,2,\ldots,\lfloor n/2 \rfloor)$

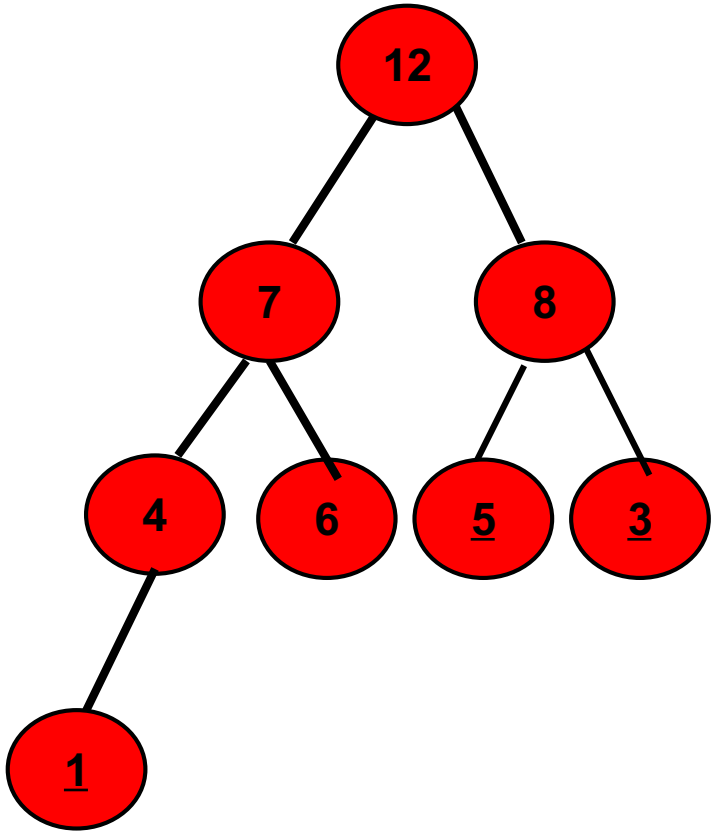
其中，下标是树按层次遍历的次序

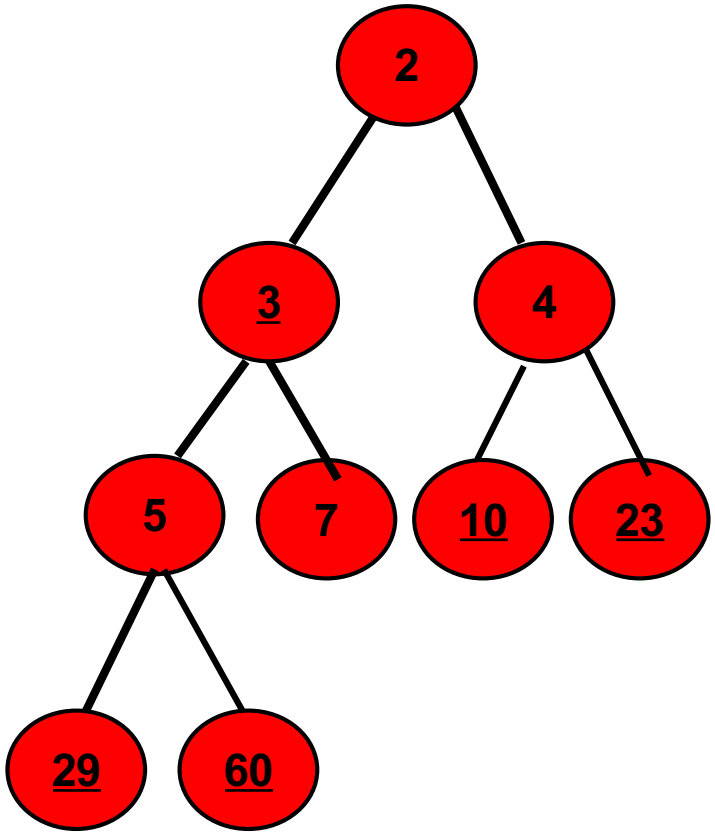
满足前一条件的称为最小化堆。例如:序列

{ 2,3,4,5,7,10,23,29,60 } 是最小化堆

满足后一条件的称为最大化堆。例如:序列

{ 12,7,8,4,6,5,3,1} 是最大化堆





最大化堆

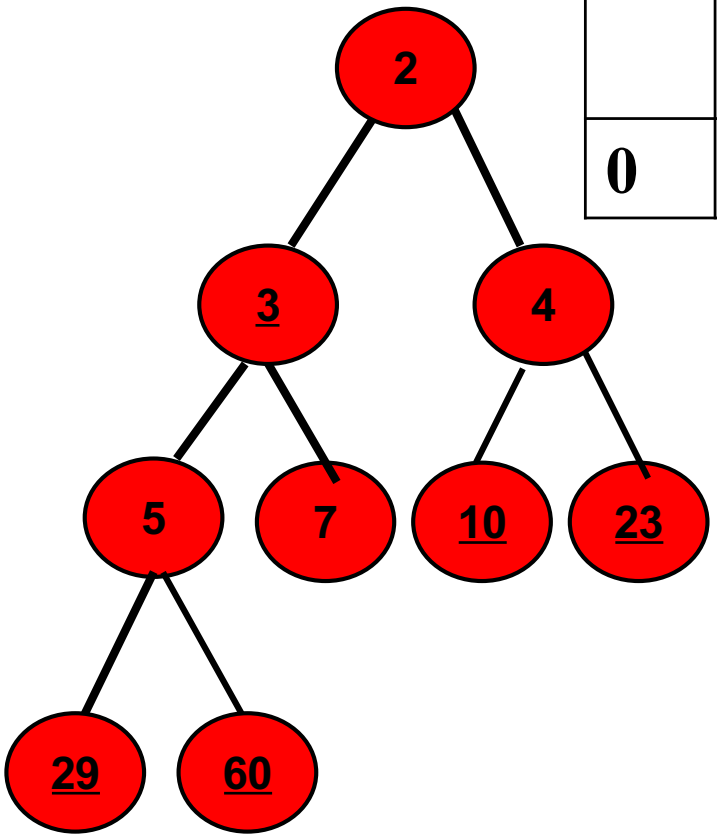
最小化堆

二叉堆的特性

- **结构性：符合完全二叉树的结构**
- **有序性：满足父节点小于子节点（最小化堆）或父节点大于子节点（最大化堆）**
- **以下的讨论都以最小化堆为例**

二叉堆的存储

- **可以采用顺序存储**
- **二叉堆的有序性可以很容易地通过下标来反映**



	2	3	4	5	7	10	23	29	60
0	1	2	3	4	5	6	7	8	9

基于二叉堆的优先级队列

- **如果数值越小，优先级越高，则可以用一个最小化堆存储优先级队列**
- **在最小化堆中，最小元素是根元素，而根结点永远存放在数组的下标为1的元素中。**
  - **获取队头元素的操作就是返回下标为1的数组元素值**
  - **出队操作就是删除下标为1的数组元素中的值**
  - **入队操作就是在数组的末尾添加一个元素，但添加后要调整元素的位置，以保持堆的有序性**

优先级队列类

- **数据成员：用一个动态数组**
- **成员函数：实现队列类的所有操作**

优先级队列类的定义

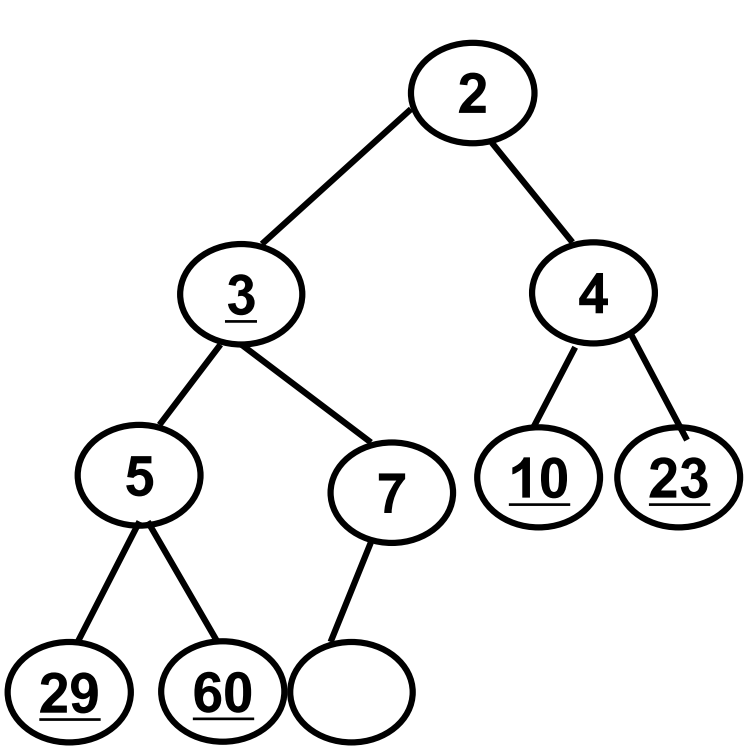
```
template <class Type>
class priorityQueue:public queue<Type>
{private:
    int currentSize;
    Type *array;
    int maxSize;

    void doubleSpace();
    void buildHeap( );
    void percolateDown( int hole );
```

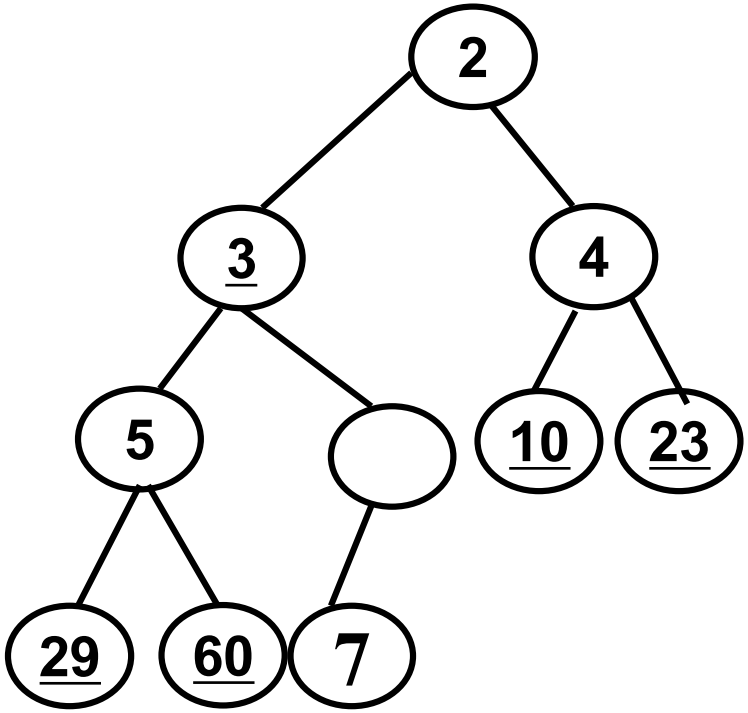
```
public:
    priorityQueue( int capacity = 100 )
    { array = new Type[capacity];
        maxSize = capacity;
        currentSize = 0;}
    priorityQueue( const Type data[], int size );
  ~priorityQueue() { delete [] array; }
    bool isEmpty( ) const { return currentSize ==
  0; }
    void enQueue( const Type & x );
    Type deQueue();
    Type getHead() { return array[1]; }
};
```

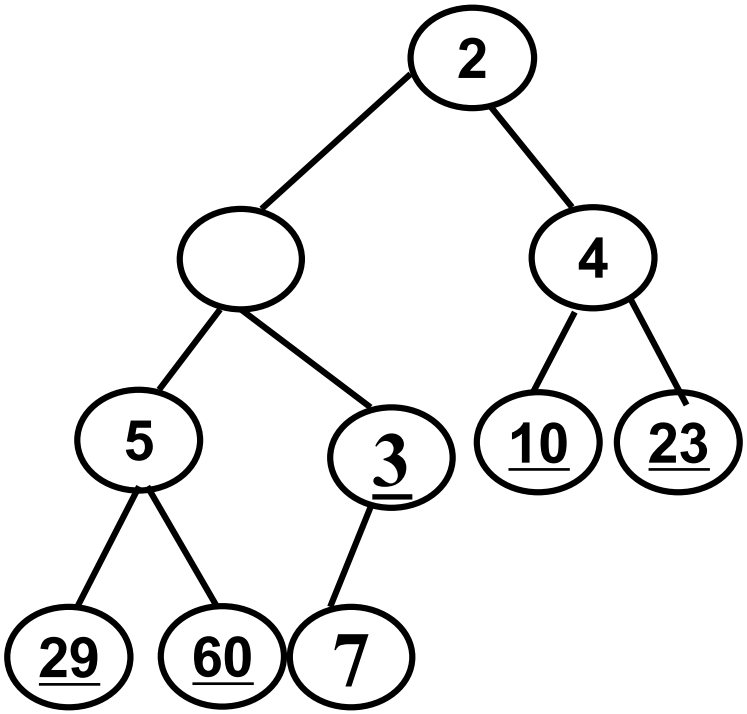
enQueue（x）

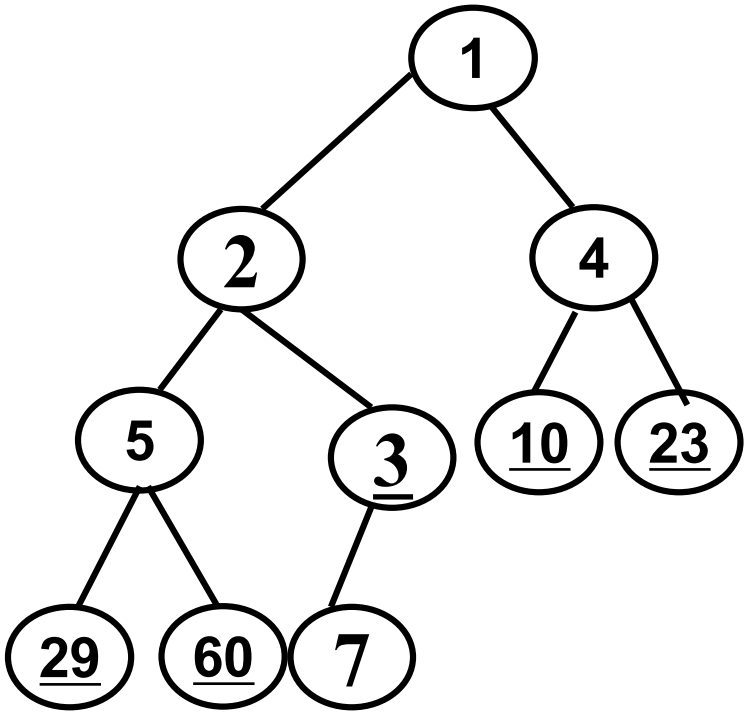
- **enQueue操作是在堆中插入一个新元素**
- **堆的插入是在具有最大序号的元素之后插入新的元素或结点，否则将违反堆的结构性。**
- **如果新元素放入后，没有违反堆的有序性，那么操作结束。否则，让该节点向父节点移动，直到满足有序性或到达根节点。**
- **新节点的向上移动称为向上过滤(percolate up)**

在如下的堆中插入元素1的过程：






2
4
5
3
10
23
29
60
7


1
2
4
5
3
10
23
29
60
7

enQueue过程

```
template <class Type>
void priorityQueue<Type>::enQueue( const Type & x )
{  if( currentSize == maxSize - 1 )  doubleSpace();

  // 向上过滤
  int hole = ++currentSize;
  for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
          array[ hole ] = array[ hole / 2 ];
  array[ hole ] = x;
}
```

enQueue的时间效率

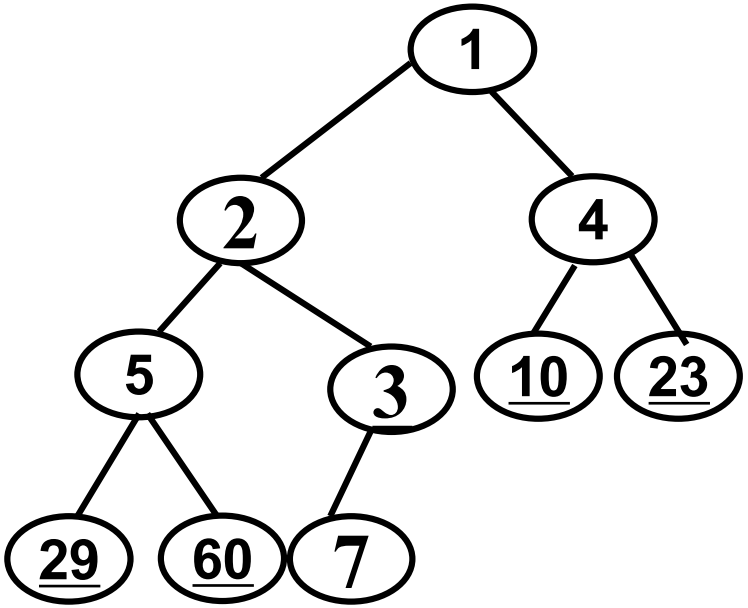
- **最坏情况是对数的**
- **平均情况，过滤会提前结束。有资料表明，平均是2.6次比较，因此元素平均上移1.6层。**

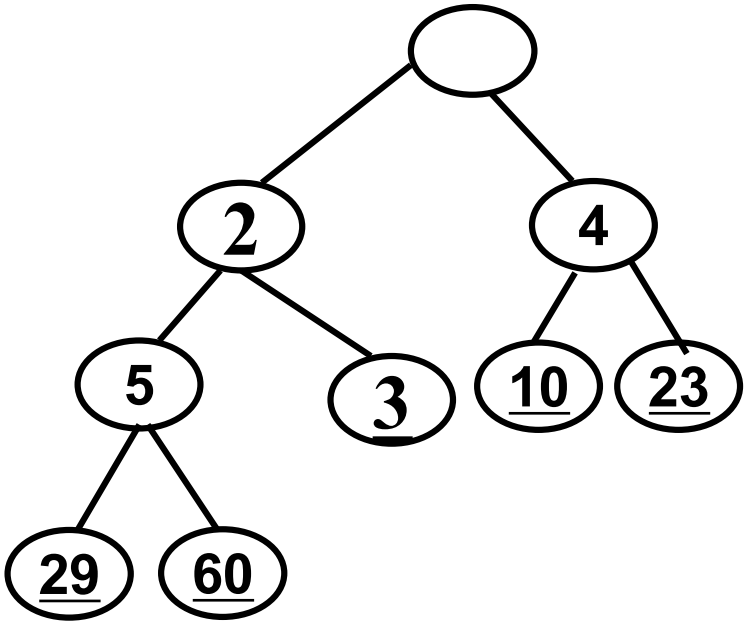
DeQueue 操作

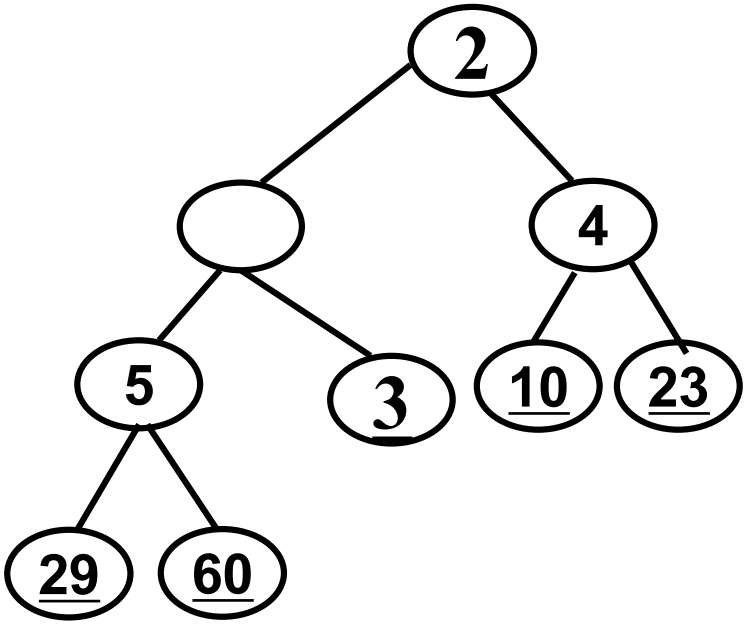
- **当最小元素被删除时，在根上出现了一个空结点。堆的大小比以前小1，堆的结构性告诉我们，最后一个结点应该删掉。**
- **如果最后一项可以放在此空结点中，就把它放进去。然而，这通常是不可能的。**
- **我们必须玩与插入操作相同的游戏：把某些项放入空结点，然后移动空结点。仅有的区别在于：对DeQueue操作，空结点是往下移动。**

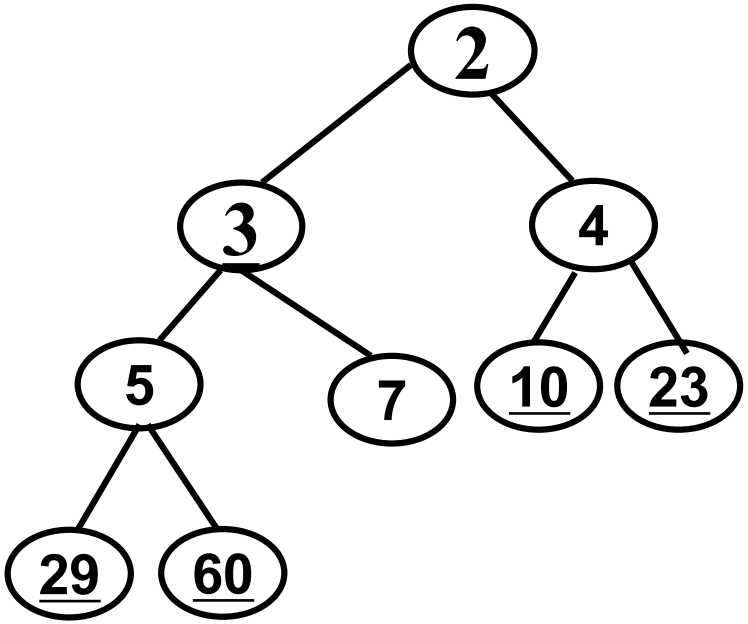
向下过滤过程

- **找到空结点的一个较小的子结点，如果该儿子的值小于我们要放入的项，则把该儿子放入空结点，把空结点往下推一层，重复这个动作，直到该项能被放入正确的位置。**


1
2
4
5
3
10
23
29
60
7


2
4
5
3
10
23
29
60


2
4
5
3
10
23
29
60


2
3
4
5
7
10
23
29
60

deQueue()

```
 template <class Type>
Type priorityQueue<Type>::deQueue()
{ Type minItem;
  minItem = array[ 1 ];
  array[ 1 ] = array[ currentSize-- ];
  percolateDown( 1 );
  return minItem;
 }
```

向下过滤

```
template <class Type>
void priorityQueue<Type>::percolateDown( int hole )
{  int child;
   Type tmp = array[ hole ];

   for( ; hole * 2 <= currentSize; hole = child )
     {  child = hole * 2;
        if( child != currentSize && array[ child + 1 ] < array[ child ] )
              child++;
        if( array[ child ] < tmp )   array[ hole ] = array[ child ];
              else    break;
      }
   array[ hole ] = tmp;
}
```

deQueue操作的性能

- **因为树有对数的深度，在最坏情况下，deQueue是一个对数时间的操作。**
- **根据堆的有序性，堆中最后一个结点的值一般都是比较大的。因此，向下过滤很少有提前一或二层结束的，所以deQueue操作平均也是对数时间。**

建堆

- **可以看成N次连续插入，其时间应该是在O(NlogN)时间内完成。**
- **事实上，在构造过程中，我们并不关心每个元素加入后堆的状态，我们关心的是N个元素全部加入后的最后状态，最后的状态是要保证堆的有序性。至于中间过程中的有序性是否成立并不重要。**
- **有了这个前提后，我们能将构造堆的时间复杂度降到O（N）**

建堆过程

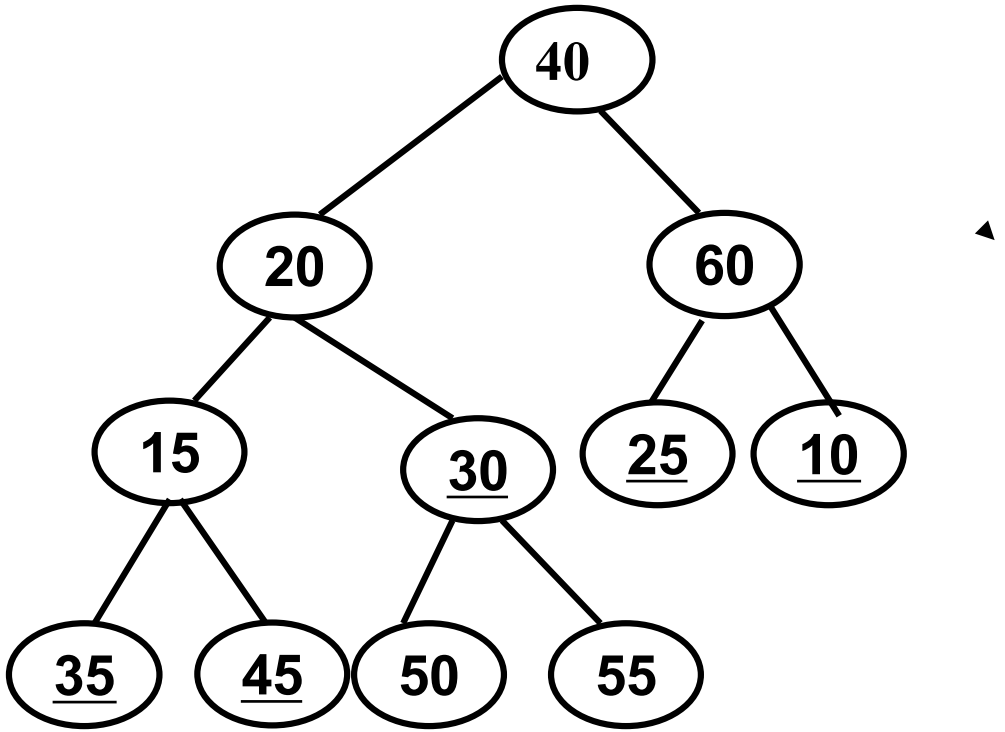
- **利用堆的递归定义**
- **如果函数buildHeap可以将一棵完全二叉树调整为一个堆 ，那么，只要对左子堆和右子堆递归调用buildHeap。至此，我们能保证除了根结点外，其余的地方都建立起了堆的有序性。然后对根结点调用percolateDown，以创建堆的有序性。**

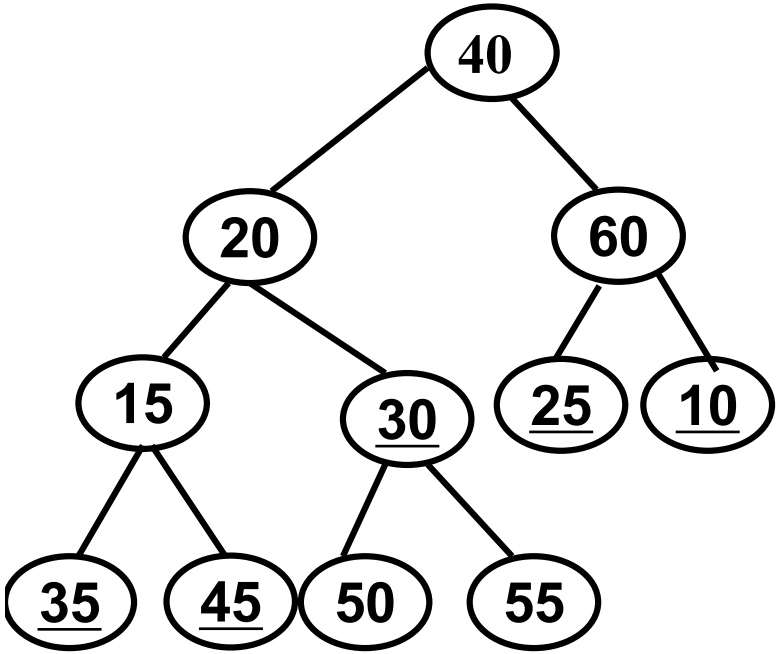
建堆过程的非递归实现

- **如果我们以逆向层次的次序对结点调用percolateDown，那么在percolateDown（i）被处理时，所有结点i的子孙都已经调用过percolateDown。**
- **注意，不需要对叶结点执行percolateDown。因此，我们是从编号最大的非叶结点开始。**

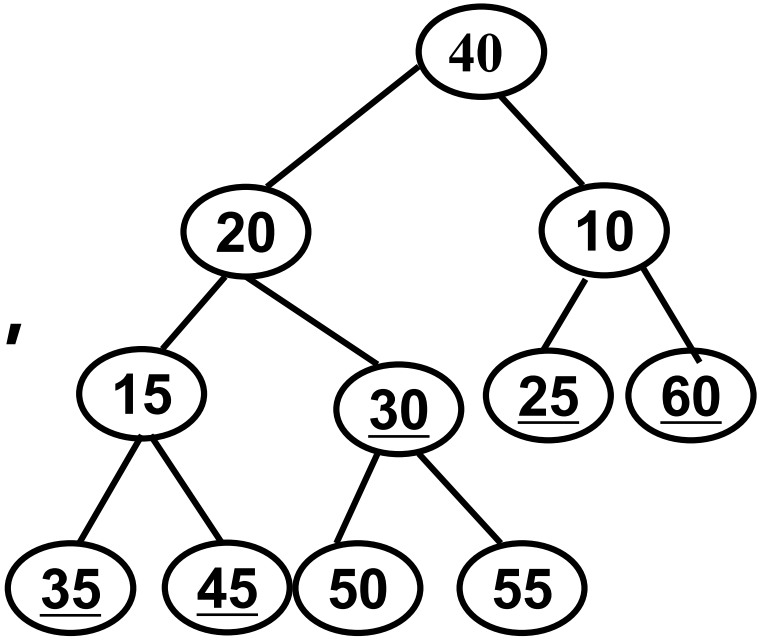
例如，给出的数据初值为40，20，60，15，30，25，10，35，45，50，55，构造一个最小化堆

首先，将它看成是一棵完全二叉树，然后把它调整成一个堆

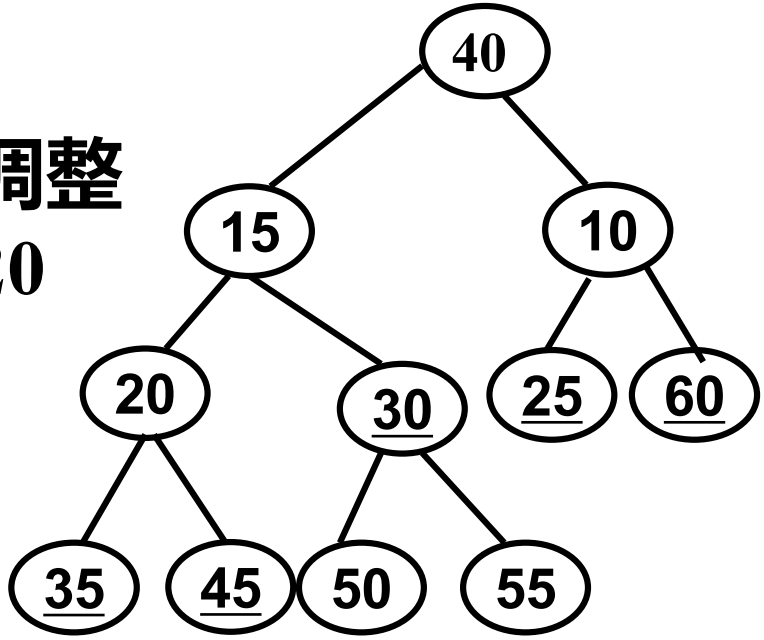




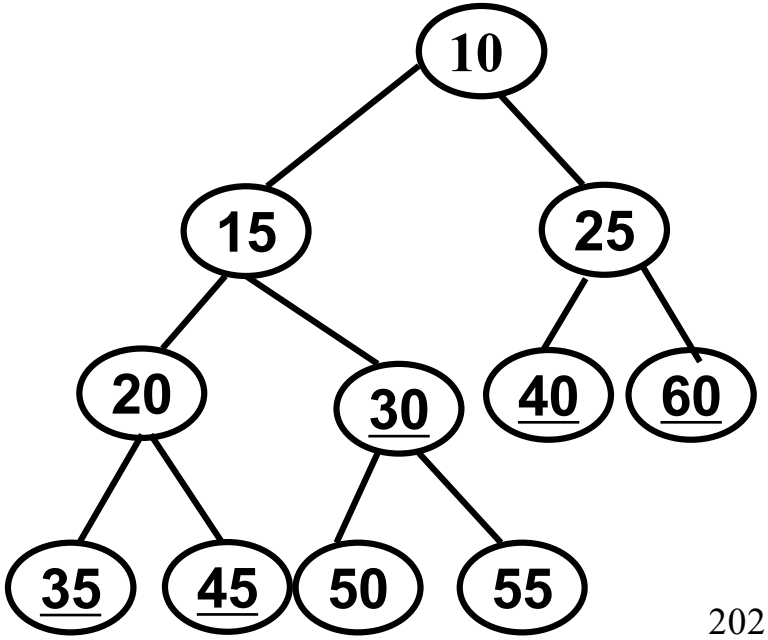
30和15没有违反堆的有序性，接着调整60



调整20



调整40



建堆总结

- **自下往上调整每一个子堆**
- **在调整每个堆时，假设除根以外，所有节点满足堆的定义**
- **根结点的调整和删除时一样，可以通过调用percolateDown实现**

建堆

```
 template <class Type>
void priorityQueue<Type>::buildHeap( )
{  for ( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}
```

带有初始数据的构造函数的实现

```
template <class Type>
priorityQueue<Type>::priorityQueue( const
      Type *items, int size )
     : maxSize(size + 10 ),  currentSize(size)
{  array = new Type[maxSize];
   for( int i = 0; i < size; i++ )
            array[ i + 1 ] = items[ i ];
  buildHeap();
}
```

建堆的时间代价分析

- **建堆的时间是O(N)的。**
- **高度为h的节点（叶节点为0），在percolateDown中交换的最大次数是h。**
- **建堆的时间是所有节点的调整时所需交换次数之和，即所有节点的高度之和。**

- **定理：对于一棵高度为h，包含了$N = 2^{H+1} - 1$个结点的满二叉树，结点的高度和为$N - H - 1$**
- **证明：高度为h的结点有一个，高度为h-1的结点有2个，高度为h-2的结点有2^2个，高度为h-i的节点有2^i个。因此，所有节点的高度和为：**

$$s = \sum_{i=0}^{h} 2^i (h-i)$$

$$= h + 2(h-1) + 4(h-2) + 8(h-3) + 16(h-4) + \ldots 2^{h-1}(1)$$

$$2s = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots + 2^h(1)$$

$$2s - s = -h + [2h - 2(h-1)] + [4(h-1) - 4(h-2)] + \ldots + 2^{h-1} + 2^h$$

$$= -h + 2 + 4 + 8 + \ldots + 2^{h-1} + 2^h$$

$$= (2^{h+1} - 1) - (h+1) = N - h - 1$$

第6章 优先级队列

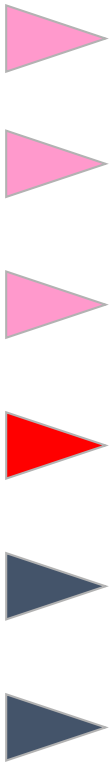
- **基本的优先级队列**
- **二叉堆**
- **D堆**
- **归并优先级队列**
- **STL中的优先级队列**
- **排队系统的模拟**

D-堆

- 每个节点有d个儿子，这样生成的堆比较矮。
- 插入：$O(\log_d N)$
- 删除：需要在d个元素中找出最小的，时间复杂度为：$O(d\log_d N)$
- 优点：插入快
- 缺点：删除慢
- 用途：
  - 插入比删除多的队列
  - 队列太大，内存放不下，要放在外存的时候

第6章 优先级队列

- **基本的优先级队列**
- **二叉堆**
- **D堆**
- **归并优先级队列**
- **STL中的优先级队列**
- **排队系统的模拟**



归并优先级队列

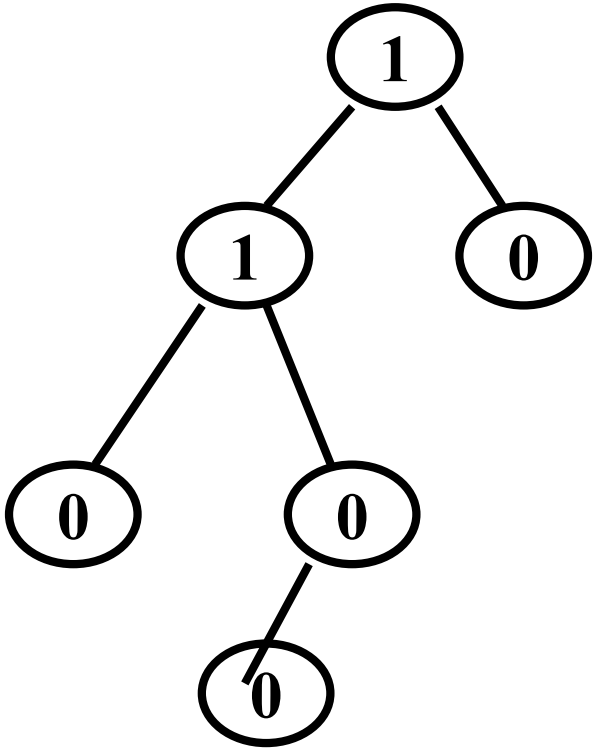
二叉堆能有效地支持优先级队列的入队和出队操作，但不能有效地支持两个优先级队列的归并。能有效地支持优先级队列归并的数据结构有左堆、斜堆和二项堆等

- 左堆
- 斜堆
- 二项堆

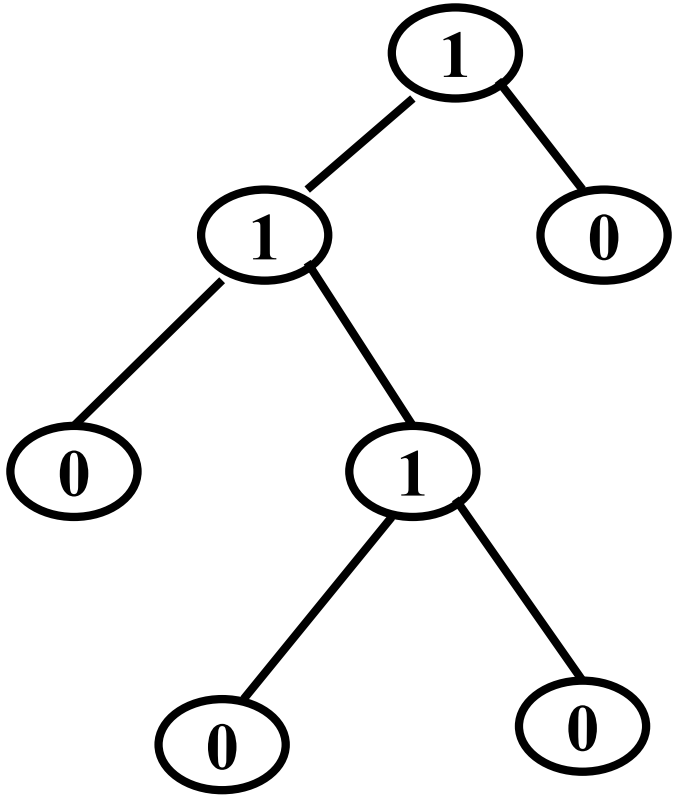
左堆

- **满足堆的有序性，但平衡稍弱一些的堆**
- **定义：空路径长度(npl)**

  Npl(x)为x到不满两个孩子的节点的最短路径。具有0个或一个孩子的节点的npl为0，npl(NULL) = -1
- **左堆：对每个节点x，左孩子的npl不小于右孩子的npl**
- **显然，左堆是向左倾斜的堆。**



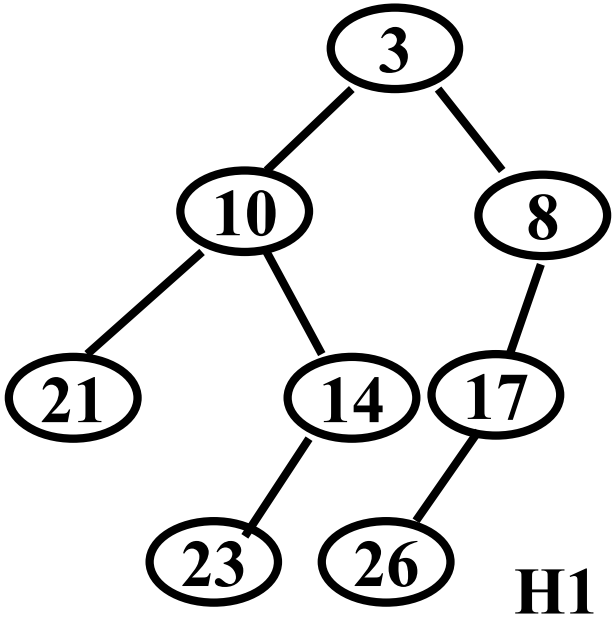
左堆



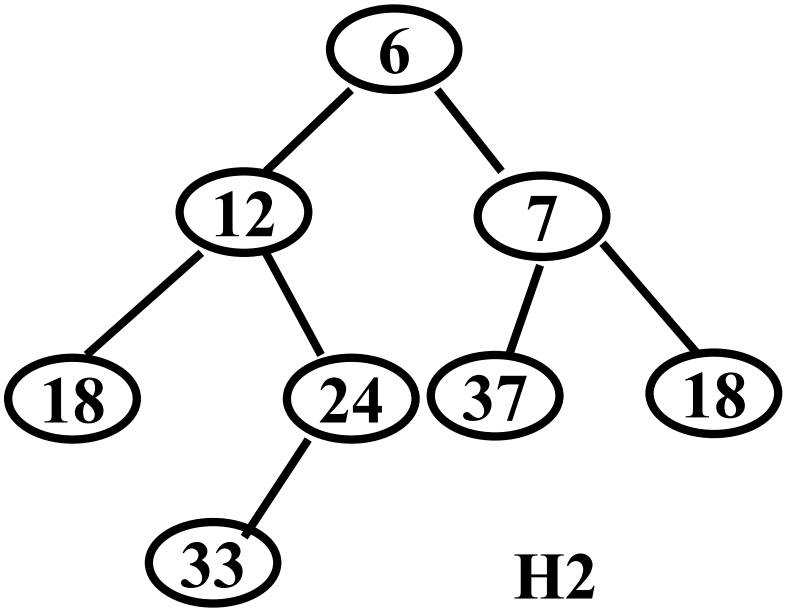
非左堆

左堆的主要操作—归并

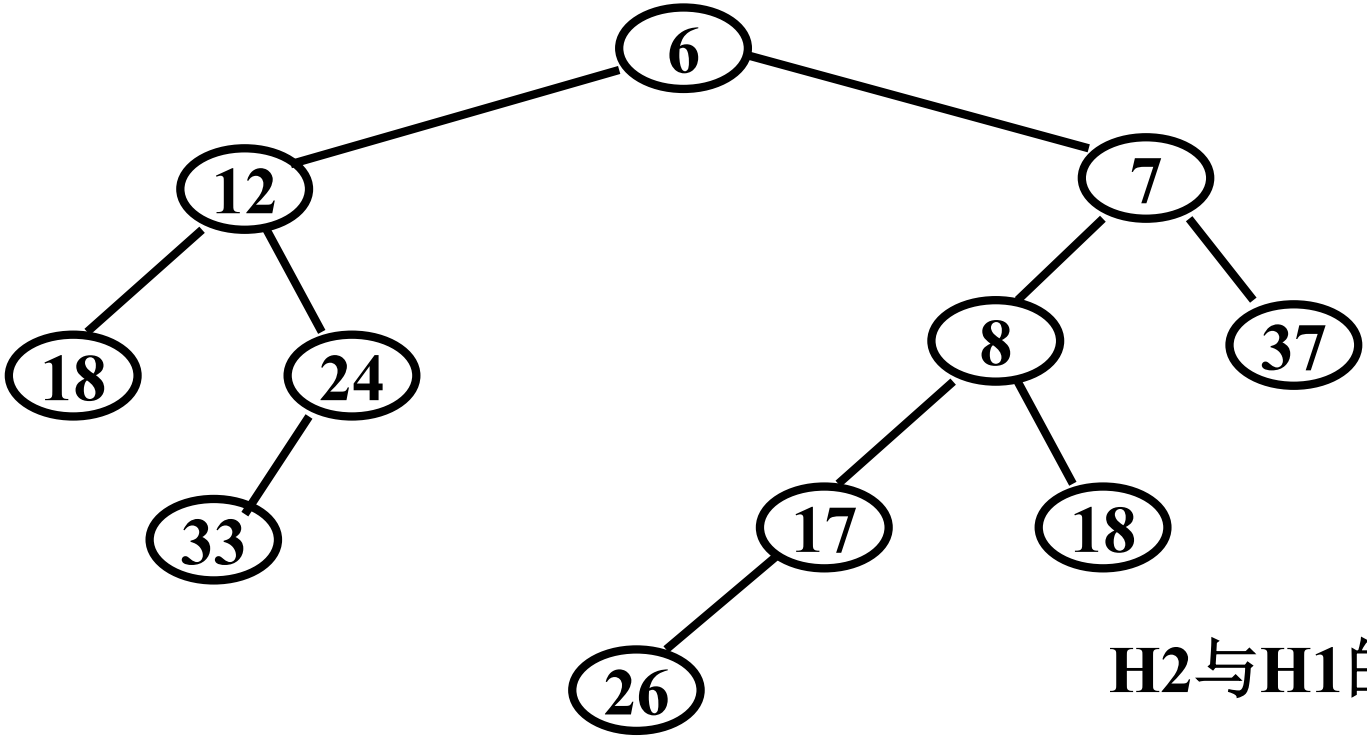
- **采用递归的方法**
  - **将根节点稍大的堆与另一个堆的右子树归并**
  - **如产生的堆违反了左堆的定义，则交换左右子树**
  - **递归的终止条件：当某个堆为空时，另一个堆就是归并的结果**



H1



H2



H2与H1的右子堆归并

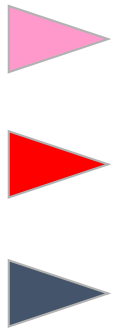
左堆的入队和出队操作

- **入队可以看成是归并的特例。将入队元素看成是指有一个元素的左堆，归并两个左堆就形成了最终的结果。**
- **出队操作的实现也很简单。删除了根结点后，这个堆就分裂成两个堆。把这两个堆重新归并起来就是原始的队列中执行出队运算后的结果。**

归并优先级队列

二叉堆能有效地支持优先级队列的入队和出队操作，但不能有效地支持两个优先级队列的归并。能有效地支持优先级队列归并的数据结构有左堆、斜堆和二项堆等

- 左堆
- 斜堆
- 二项堆

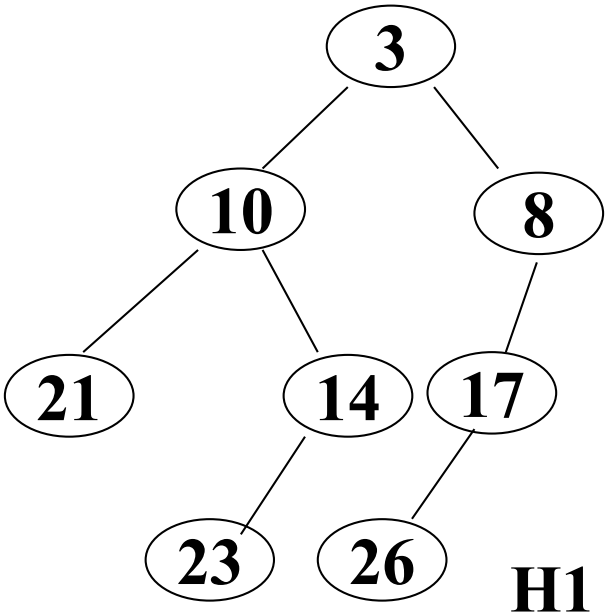


斜堆

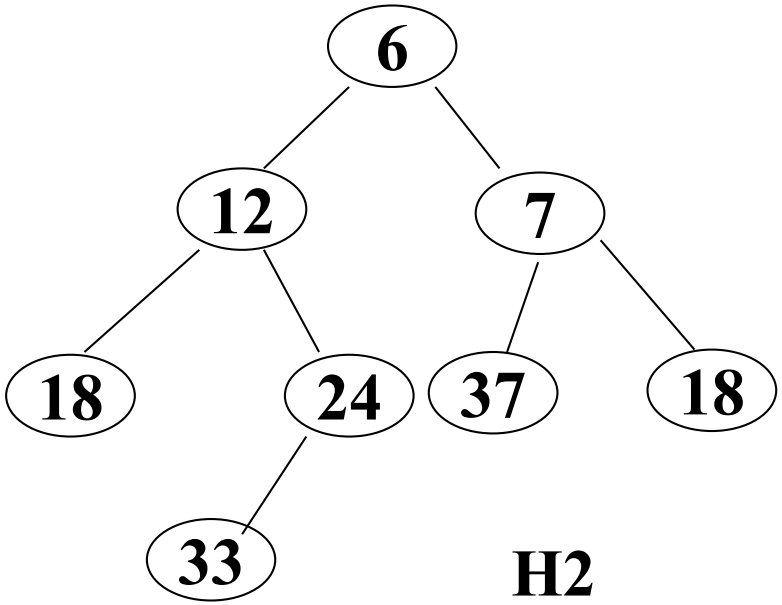
- **斜堆是自调整的左堆。它满足堆的有序性，但不满足堆的结构性。不需要维护npl。因此，右路径可以任意长。**
- **最坏的时间复杂性O(N)，但对M个连续的操作，最坏的运行时间是O(MlogN)。因此，每个操作由均摊的O(logN)的时间复杂度。**
- **它的操作比左堆简单。**

斜堆的归并

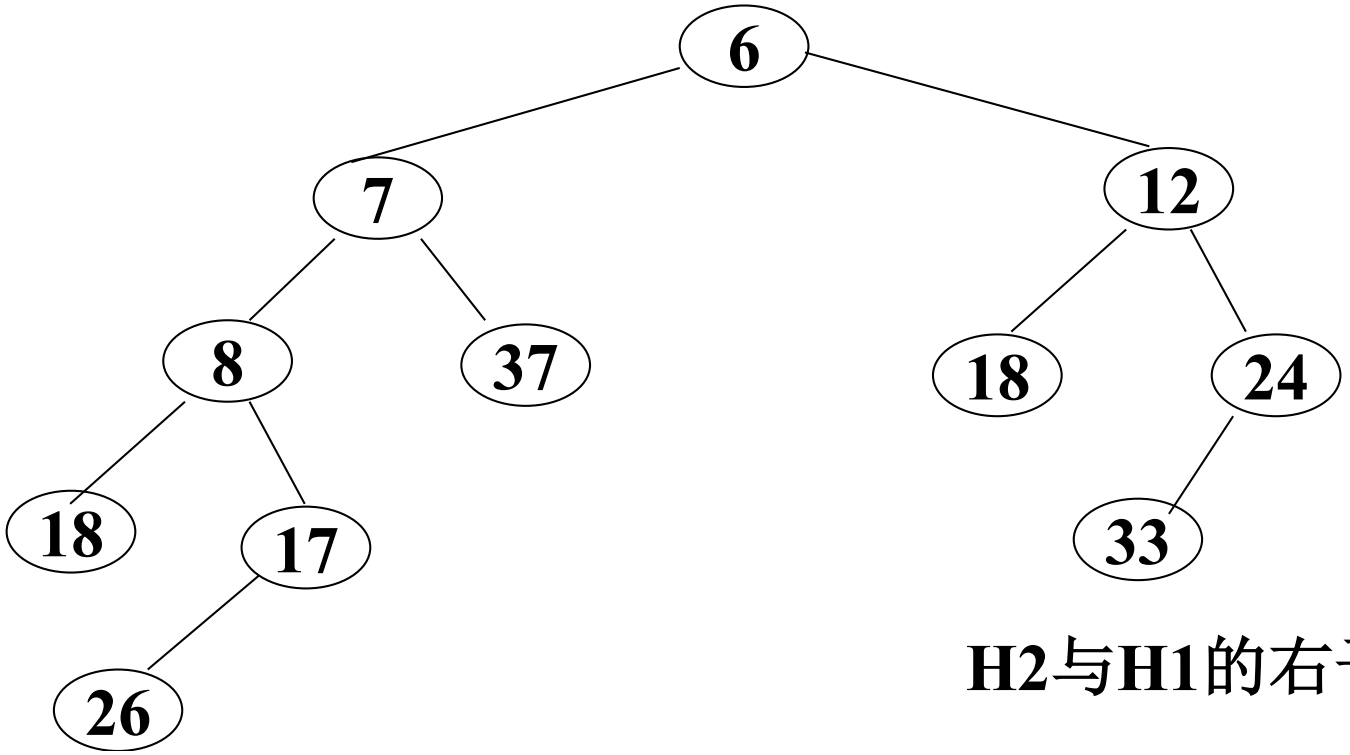
- **类似于左堆，只是在左堆中，归并后左右子堆的交换是有条件的；而在斜堆中，是无条件的，必须交换。**



H1



H2



H2与H1的右子堆归并

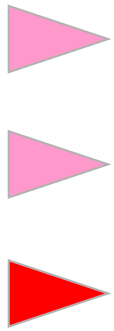
斜堆的优点

- **不需要保存npl**
- **不需要维护npl**
- **不需要测试npl以决定是否要交换左右子堆**

归并优先级队列

二叉堆能有效地支持优先级队列的入队和出队操作，但不能有效地支持两个优先级队列的归并。能有效地支持优先级队列归并的数据结构有左堆、斜堆和二项堆等

- **左堆**
- **斜堆**
- 二项堆



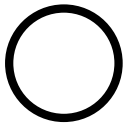
二项堆

- 二项堆**支持插入、删除和归并操作。最坏情况下的时间复杂性是O(logN)，但平均的插入时间是一个常量。**
- 二项堆**表示为一个**二项**树的集合。**

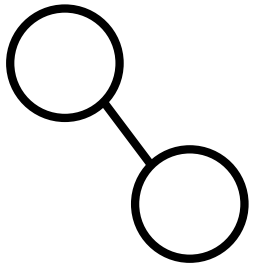
二项堆树

- 二项堆**树是一棵普通的树，不是二叉树。**
- **高度为0的**二项堆**树是单个节点，高度为k的**二项堆**树B_k是将一棵B_{k-1}加到另一棵B_{k-1}的根上形成的。**
- 二项堆**树满足堆的有序性**

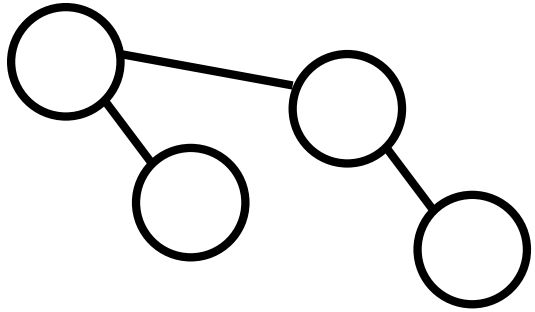
B_0



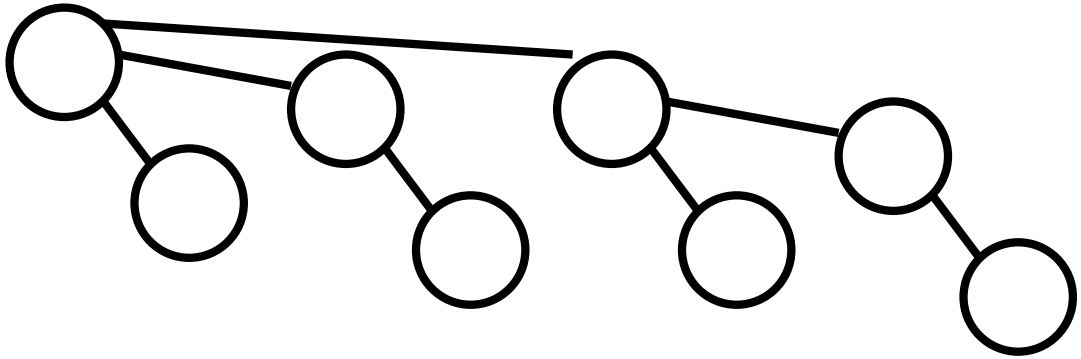
B_1



B_2



B_3



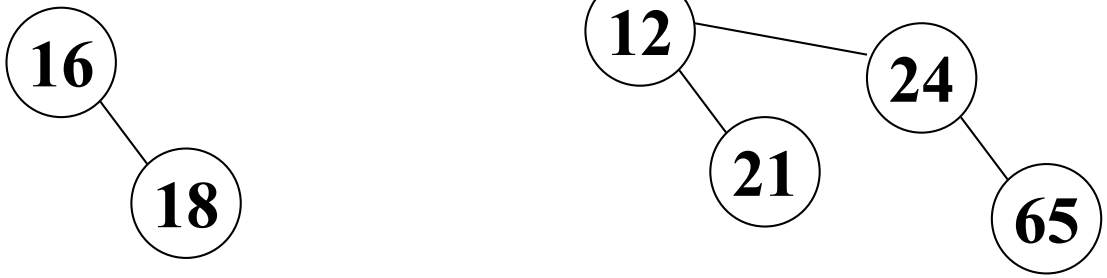
二项堆里树B_k的特性

- **B_k有2^k个节点**
- **第d层的节点数是二项堆系数** $\binom{k}{d}$

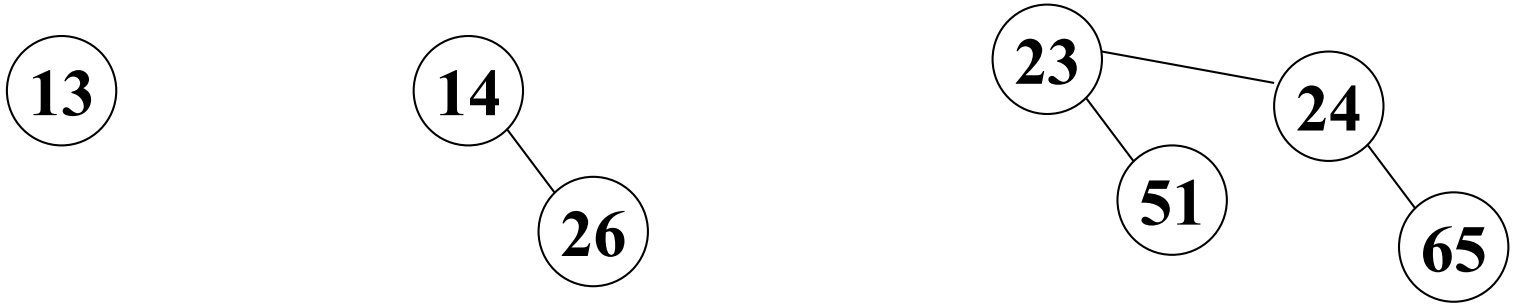
优先级队列的表示

- **把优先级队列表示为**二项堆**树的集合。每个高度至多有一棵**二项堆**树。这样，对于给定的元素个数，这个集合是唯一的，即元素个数的二进制表示。**
- **如13个元素，可表示为1101。即该集合由B_3、B_2和B_0组成**

六个元素的**二项堆：**

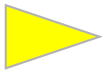


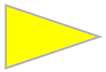
七个元素的**二项堆：**



二项堆的操作

- **归并**
- **入队**
- **出队**

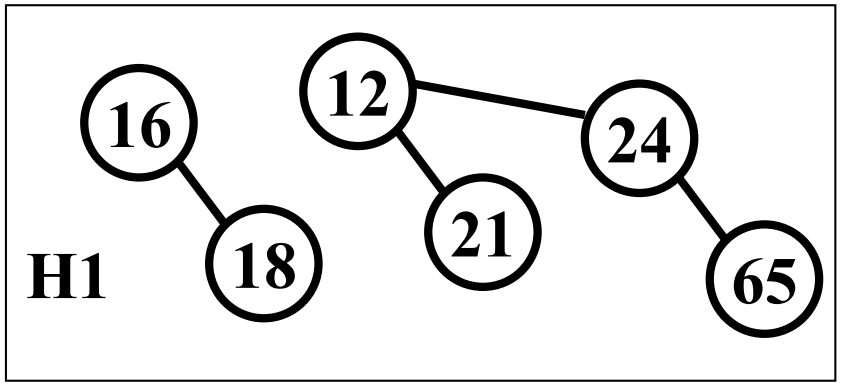


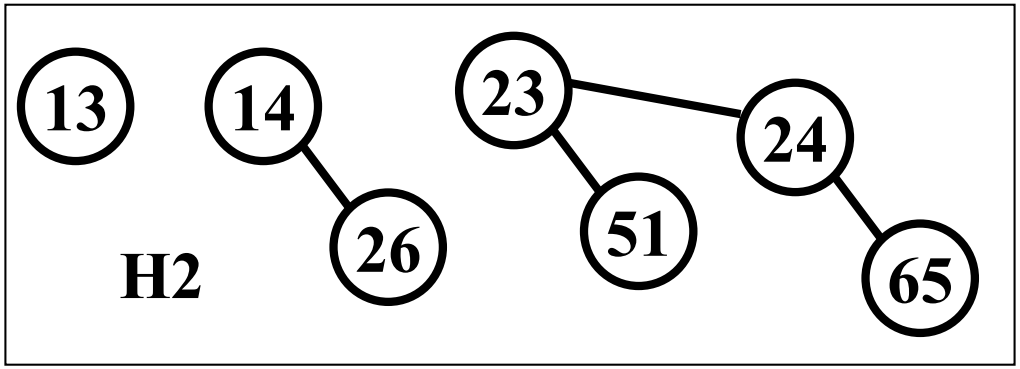


归并操作

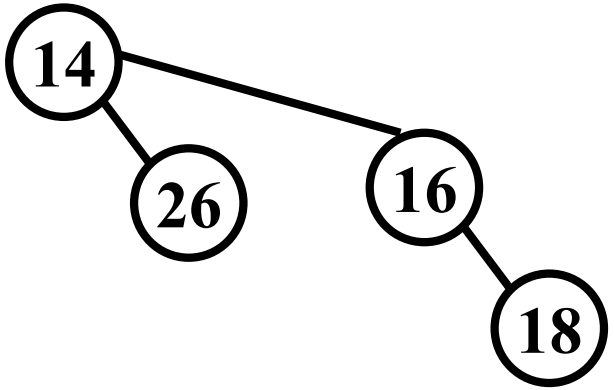
- **由低到高依次归并两个优先级队列中高度相同的树。如果由于前一次归并而出现三棵高度相同的树时，留下一棵，归并其余两棵。**
- **高度相同的树的归并：将根节点大的作为根节点小的树的子树。**
- **归并的时间效益：N个元素的队列有logN棵树，因此最坏情况为O(logN)。**

归并以下两个队列：



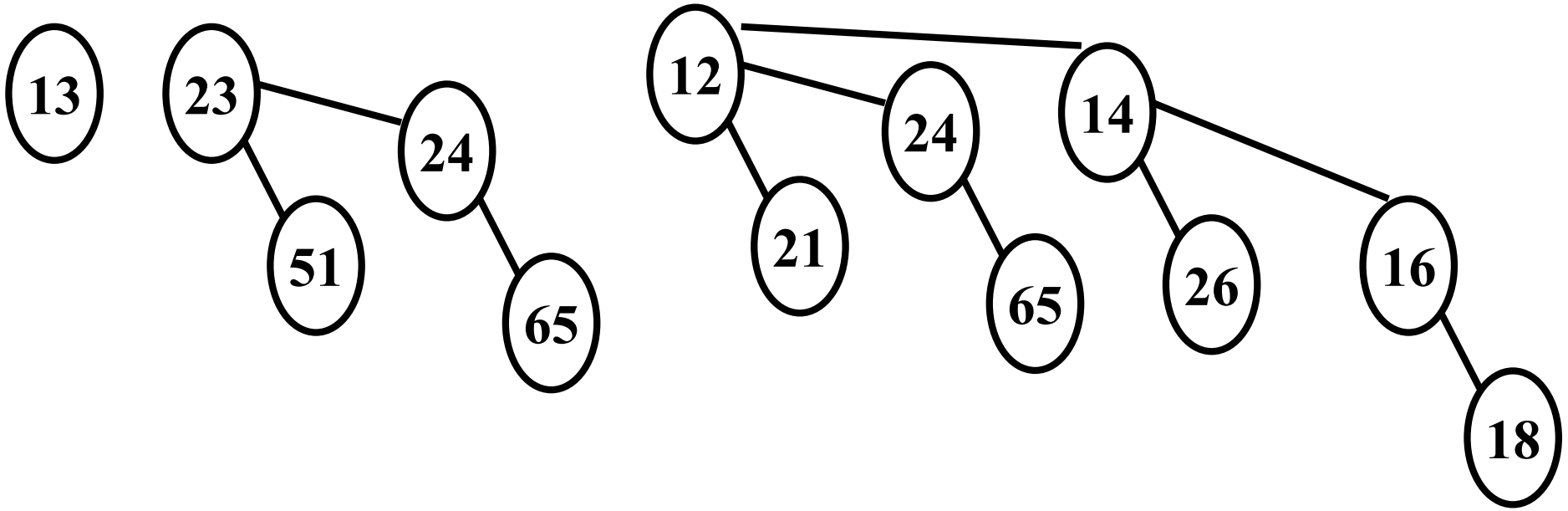


- 归并B_0：由于只有H2有B_0，所以无需归并
- 归并B_1：形成以下的树



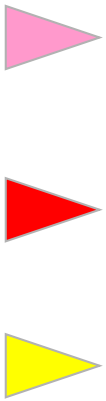
- 现在有三棵B2，留下一棵，归并其余两棵

最后的队列：



二项堆的操作

- **归并**
- **入队**
- **出队**

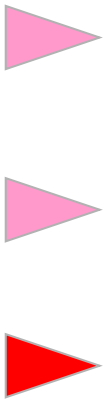


插入

- **插入是归并的特例**
- **为被插入节点形成一棵单节点的树组成的集合，归并两个集合**
- **时间效益：最坏情况为O(logN)，相当于二进制加法中的加1，但每次都有进位的情况。一般进位进到中间的某一位会终止。即当原先集合中缺少B_i时，则归并i次，由于每棵树的出现是等概率的，因此平均归并两次就能结束。**

贝努里队列的操作

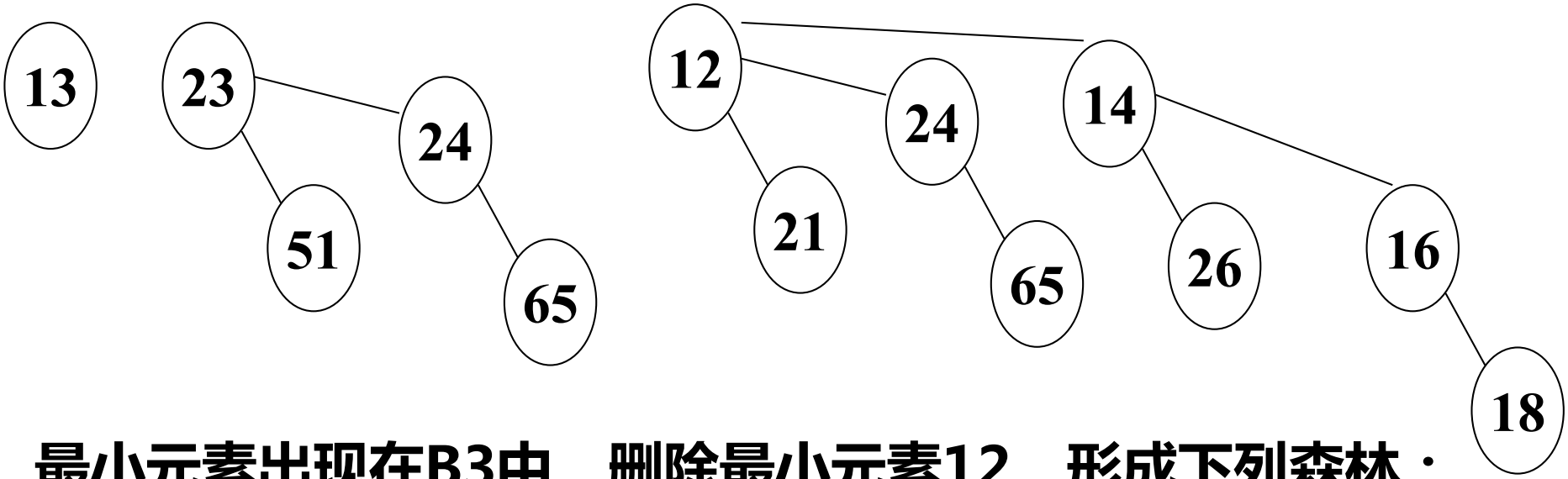
- **归并**
- **入队**
- **出队**



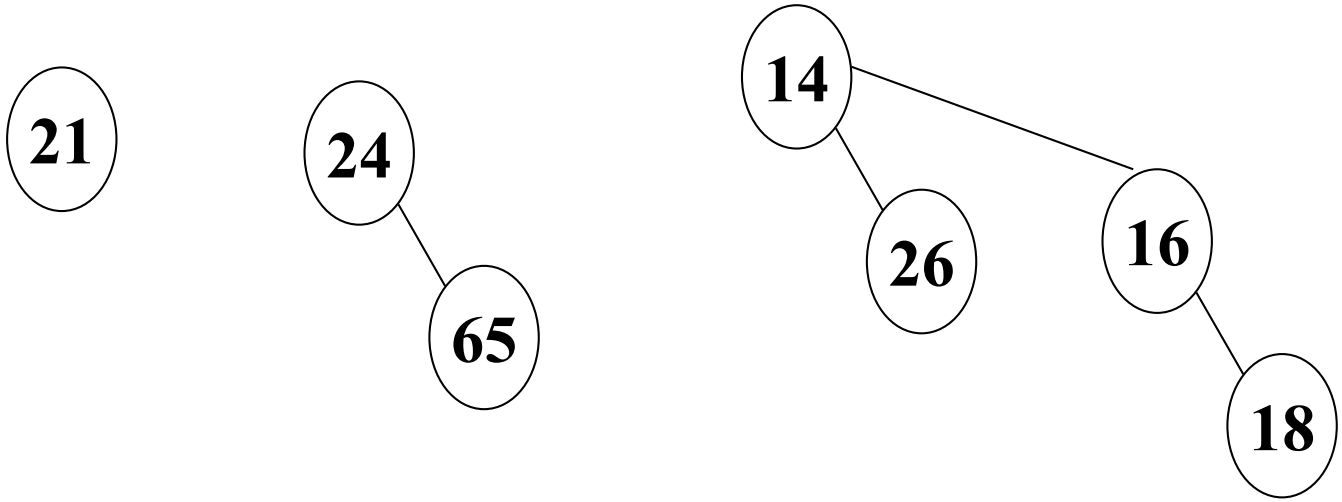
删除

- **找出具有最小根值的树T**
- **将T从原先的集合中删掉**
- **在T中删除根节点**
- **归并T与原先的集合**

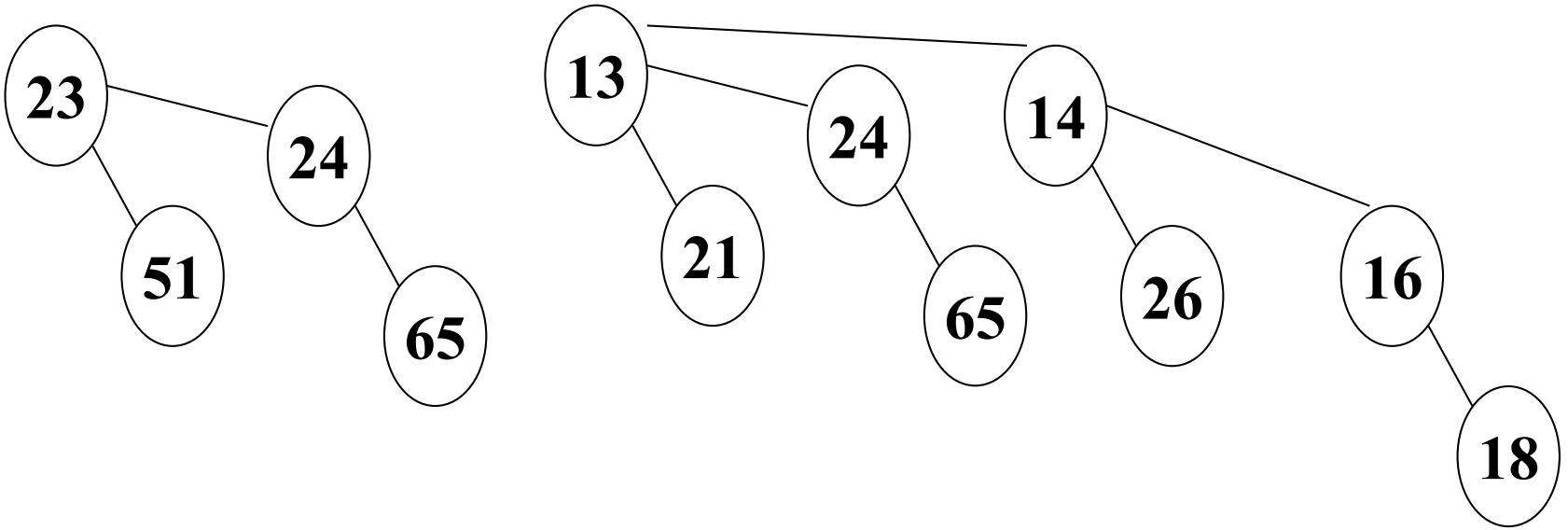
在以下的队列中删除最小元素：



最小元素出现在B3中，删除最小元素12，形成下列森林：



归并两个森林：



二项堆的时间性能

- **归并：N个元素的队列至多有logN棵树，每两棵树的归并只需要常量的时间。因此最坏情况的时间复杂度为O(logN)。但可以证明平均情况的时间复杂度是常量的。**
- **入队操作的平均时间复杂度是O（1）的**
- **出队操作：首先在队列中找出根结点值最小的树。这需要花费O（logN）的时间。然后又要归并两个优先级队列，又需要O（logN）的时间。所以删除操作的时间复杂度是O（logN）的**

第6章 优先级队列

- **基本的优先级队列**
- **二叉堆**
- **D堆**
- **归并优先级队列**
- **STL中的优先级队列**
- **排队系统的模拟**

STL中的优先级队列

- **头文件：queue**
- **类模版：priority_queue**
- **实现方式：二叉堆**
- **主要成员：**
  - **Void push(const Object &x)**
  - **Const Object &top() const**
  - **Void pop()**
  - **Bool empty()**
  - **Void clear()**

使用实例

```
#include <iostream>
#include <queue>
using namespace std;
int main()
{ priority_queue<int> q;
  q.push(10); q.push(1); q.push(5); q.push(8);
  q.push(0); q.push(4);  q.push(9); q.push(7);
  q.push(3); q.push(6); q.push(2);
  while (!q.empty()) {cout << q.top() << "  ";
      q.pop();}
  return 0;
}
```

第6章 优先级队列

- **基本的优先级队列**
- **二叉堆**
- **D堆**
- **归并优先级队列**
- **STL中的优先级队列**
- **排队系统的模拟**

单服务台的排队系统

- **在单服务台系统中，先到达的顾客先获得服务，也肯定先离开。到达和离开的次序是一致的。**
- **事件处理的次序是：顾客1到达、顾客1离开、顾客2到达、顾客2离开、……、顾客n到达、顾客n离开**
- **只需要一个普通队列保存顾客到达信息**

多服务台的排队系统

- **在多服务台系统中，先到达的顾客先获得服务，但后获得服务的顾客可能先离开；**
- **事件处理的次序可能是：顾客1到达、顾客2到达、顾客2离开、顾客3到达、顾客1离开、顾客3离开……、顾客n到达、顾客n离开、……**
- **发生时间早的事件先处理，发生时间晚的事件后处理。因而需要一个优先级队列存放事件。事件的优先级就是发生的时间**

模拟过程

- **模拟开始时，产生所有的到达事件，存入优先级队列。**
- **模拟器开始处理事件：**
  - **从队列中取出一个事件。这是第一个顾客的到达事件。生成所需的服务时间。当前时间加上这个服务时间就是这个顾客的离开时间。生成一个在这个时候离开的事件，插入到事件队列。**
  - **同样模拟器从队列中取出的事件也可能是离开事件，这时只要将这个离开事件从队列中删去，为他服务的服务台变成了空闲状态，可以继续为别的顾客服务。**

多服务台的排队系统过程

产生CustomNum个顾客的到达事件，

按时间的大小存入事件队列；

置等待队列为空；

置所有柜台为空闲；

设置等待时间为0；

```
While （事件队列非空）{
    队头元素出队；   设置当前时间为该事件发生的时间；
   switch(事件类型)
     { case 到达：if (柜台有空)
                        { 柜台数减1； 生成所需的服务时间 ；
                          修改事件类型为 “离开”
                          设置事件发生时间为当前时间加上服务时间；
                          重新存入事件队列；}
                       else 将该事件存入等待队列；
       case 离开：if （等待队列非空）
                        { 队头元素出队； 统计该顾客的等待时间；
                          生成所需的服务时间 ；修改事件类型为 “离开”
                          设置事件发生时间为当前时间加上服务时间；
                          存入事件队列；    }
                       else 空闲柜台数加1；
     }
}
计算平均等待时间； 返回；
}
```

模拟类

- **设计一个模拟类，告诉它顾客到达时间间隔的分布、服务时间的分布、模拟的柜台数以及想要模拟多少个顾客。能提供在本次服务中顾客的平均等待时间是多少。**
- **因此这个类应该有两个公有函数：构造函数接受用户输入的参数，另外一个就是执行模拟并最终给出平均等待时间的函数avgWaitTime。这个类要保存的数据就是模拟的参数。**

模拟类的定义

```
class simulator{
      int noOfServer;  //服务台个数
      int arrivalLow;   //到达间隔时间的下界
      int arrivalHigh;  //到达间隔时间的上界
      int serviceTimeLow;  //服务间隔时间的下界
      int serviceTimeHigh;  //服务间隔时间的上界
      int customNum;       //模拟的顾客数
      struct eventT
       { int  time;  //事件发生时间
         int type;  //事件类型。0为到达，1为离开
         bool operator<(const eventT &e) const
                {return time < e.time;} } ;
public:
      simulator();
      int avgWaitTime();
};
```

构造函数的实现

```
simulator::simulator()
{    cout << "请输入柜台数：";      cin >> noOfServer;
     cout <<  “请输入到达时间间隔的上下界
                （最小间隔时间  最大间隔时间）：";
     cin >> arrivalLow >> arrivalHigh;
     cout <<  “请输入服务时间的上下界
                （服务时间下界  服务时间上界）：";
     cin >> serviceTimeLow >> serviceTimeHigh;
     cout << "请输入模拟的顾客数：";
      cin >> customNum;
      srand(time(NULL));
}
```

avgWaitTime()

```
int simulator::avgWaitTime()
{ int serverBusy = 0;  // 正在工作的服务台数
  int currentTime ;    // 记录模拟过程中的时间
  int totalWaitTime = 0;
                //模拟过程中所有顾客的等待时间的总和
  linkQueue<eventT> waitQueue;  //顾客等待队列
  priorityQueue<eventT> eventQueue;  //事件队列
  eventT currentEvent;
```

```
//生成初始的事件队列
  int i;
  currentEvent.time = 0;
  currentEvent.type = 0;
  for (i=0; i<customNum; ++i)
    { currentEvent.time += arrivalLow
              + (arrivalHigh -arrivalLow +1) *
               rand() / (RAND_MAX + 1);
      eventQueue.enQueue(currentEvent);
    }
```

```
//模拟过程
  while (!eventQueue.isEmpty())
    {currentEvent = eventQueue.deQueue();
     currentTime = currentEvent.time;
     switch(currentEvent.type)
     {case  0:  //处理到达事件
                    break;
      case  1:   //处理离开事件
   } //switch结束
 } //while结束
 return totalWaitTime / customNum;
}
```

处理到达事件

```
if  (serverBusy != noOfServer)
      { ++serverBusy;
        currentEvent.time += serviceTimeLow +
             (serviceTimeHigh - serviceTimeLow
 +1)
             * rand() / (RAND_MAX + 1);
        currentEvent.type = 1;
        eventQueue.enQueue(currentEvent);
 }
 else waitQueue.enQueue(currentEvent);
```

处理离开事件

```
if (!waitQueue.isEmpty())
   { currentEvent = waitQueue.deQueue();
        totalWaitTime += currentTime -
  currentEvent.time;
        currentEvent.time = currentTime +
  serviceTimeLow
              + (serviceTimeHigh - serviceTimeLow +1) *
               rand() / (RAND_MAX + 1);
      currentEvent.type = 1;
      eventQueue.enQueue(currentEvent);
    }
  else --serverBusy;
```

simulator类的使用

```
int main()
{
  simulator sim;
  cout << "平均等待时间：" 
       << sim.avgWaitTime() << endl;
  return 0;
}
```

某次执行结果

请输入柜台数：4

请输入到达时间间隔的上下界（最小间隔时间 最大间隔时间）：0 2

请输入服务时间的上下界（服务时间下界 服务时间上界）：2 7

请输入模拟的顾客数：1000

平均等待时间：61

总结

- **优先级队列是程序设计中常用的一个工具。**
- **本章介绍了一种优先级队列的优秀的实现方法 —— 二叉堆。**
- **还介绍了一些能够实现优先级队列归并的堆的概念，**
- **最后。介绍了优先级队列的一个重要应用，即多服务台的排队系统的模拟。**

作业