第五章 树

- 树和森林的概念
- 二叉树
- 二叉树遍历
- 线索化二叉树
- 树与森林
- 堆
- **Huffman树**

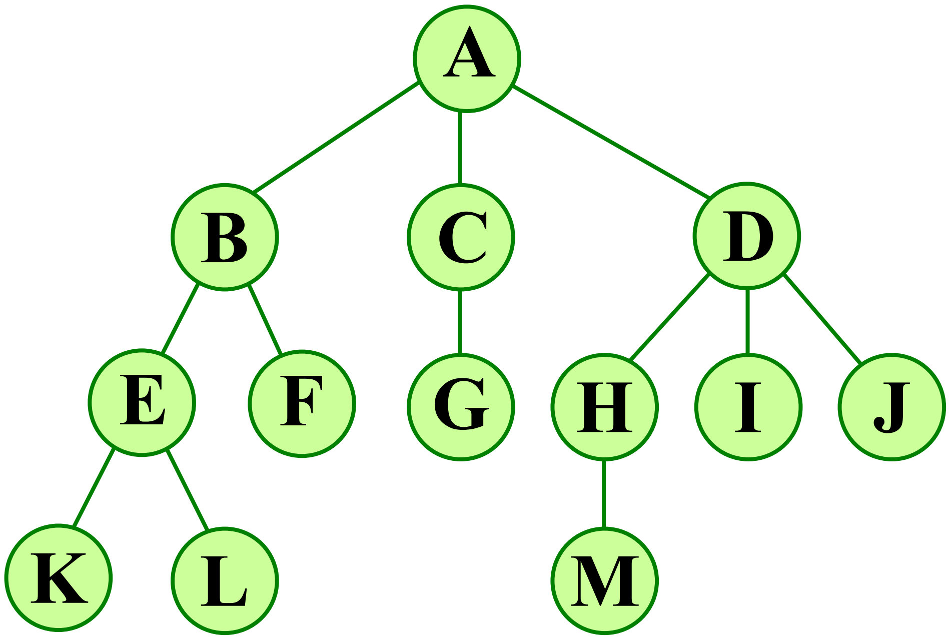
树和森林的概念

- 有根树:
  - 一棵有根树T，简称为树，它是n ($n \geq 0$) 个结点的有限集合。当$n = 0$时，T称为空树；否则，T是非空树。记作

$$T = \begin{cases} \Phi, & n = 0 \\ \{r, T_1, T_2, \ldots, T_m\}, & n > 0 \end{cases}$$

  - r是一个特定的称为根 (root) 的结点，它只有直接后继，没有直接前驱
  - 根以外的其他结点划分为m ($m \geq 0$) 个互不相交的有限集合$T_1, T_2, \ldots, T_m$，每个集合又是一棵树，并且称为根的子树

- 每棵子树的根结点有且仅有一个直接前驱，但可以有0个或多个直接后继



树的基本术语

- 子女：若结点的子树非空，结点子树的根即为该结点的子女。

- 双亲（父亲）：若结点有子女，该结点是子女的双亲（父亲）。

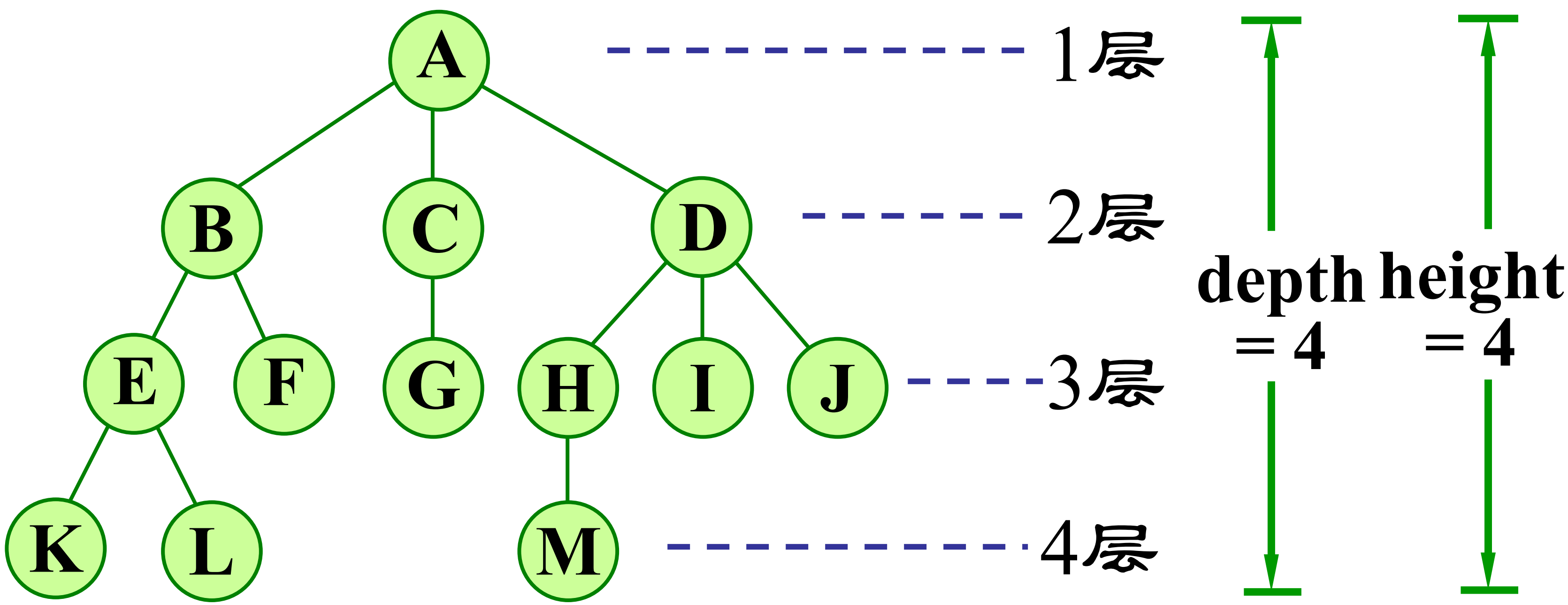
- 兄弟：同一结点的子女互称为兄弟。

- 度：结点的子女个数即为该结点的度；树中各个结点的度的最大值称为树的度。

- 分支结点：度不为0的结点即为分支结点，亦称为非终端结点。
- 叶结点：度为0的结点即为叶结点，亦称为终端结点。
- 祖先：根结点到该结点的路径上的各个结点都是该结点的祖先。
- 子孙：某结点的所有下属结点，都是该结点的子孙。

- 结点的层次：规定根结点在第一层，其子女结点的层次等于它的层次加一。以下类推。

- 结点的深度：结点的深度即为结点的层次；离根最远结点的层次即为树的深度。



- **结点的高度**：规定叶结点的高度为1，其双亲结点的高度等于它的高度加一。
- **树的高度**：等于根结点的高度，即根结点所有子女高度的最大值加一。
- **有序树**：树中结点的各棵子树 $T_1, T_2, \ldots$是有次序的，即为有序树。
- **无序树**：树中结点的各棵子树之间的次序是不重要的，可以互相交换位置。
- **森林**：森林是m（$m \geq 0$）棵树的集合。

树的抽象数据类型

```
template <class T>
class Tree {
//对象: 树是由n (≥0) 个结点组成的有限集合。在
//类界面中的 position 是树中结点的地址。在顺序
//存储方式下是下标型, 在链表存储方式下是指针
//型。T 是树结点中存放数据的类型, 要求所有结
//点的数据类型都是一致的。
public:
    Tree ();
    ~Tree ();
```

BuildRoot (**const** T**&** value)**;**

//建立树的根结点

position FirstChild(position p)**;**

//返回 **p** 第一个子女地址, 无子女返回 **0**

position NextSibling(position p)**;**

//返回 **p** 下一兄弟地址, 若无下一兄弟返回 **0**

position Parent(position p)**;**

//返回 **p** 双亲结点地址, 若 **p** 为根返回 **0**

T GetData(position p)**;**

//返回结点 **p** 中存放的值

bool InsertChild(position p, T**&** value)**;**

//在结点 **p** 下插入值为 **value** 的新子女, 若插

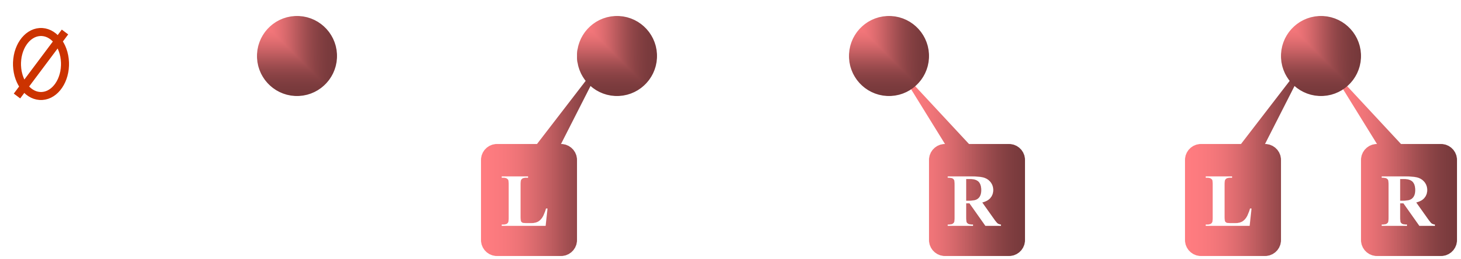
//入失败, 函数返回**false,** 否则返回**true**

```
    bool DeleteChild (position p, int i);
      //删除结点 p 的第 i 个子女及其全部子孙结
      //点, 若删除失败, 则返回false, 否则返回true
    void DeleteSubTree (position t);
      //删除以 t 为根结点的子树
    bool IsEmpty ();
      //判树空否, 若空则返回true, 否则返回false
    void Traversal (void (*visit)(position p));
      //遍历以 p 为根的子树
};
```

二叉树 (Binary Tree)

- <u>二叉树的定义</u>

  一棵二叉树是结点的一个有限集合，该集合或者为空，或者是由一个根结点加上两棵分别称为左子树和右子树的、互不相交的二叉树组成。



二叉树的五种不同形态

二叉树的性质

- <u>性质1</u>　若二叉树结点的层次从 1 开始, 则在二叉树的第 i 层最多有 2^{i-1} 个结点。($i\geqslant 1$)

  [证明用数学归纳法]

- <u>性质2</u>　深度为 k 的二叉树最少有 k 个结点，最多有 2^k-1个结点。($k\geqslant 1$)

  因为每一层最少要有1个结点，因此，最少结点数为 k。最多结点个数借助性质1：用求等比级数前k项和的公式

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

- <u>性质3</u>　对任何一棵二叉树，如果其叶结点有 n_0 个, 度为 2 的非叶结点有 n_2 个，则有

$$n_0 = n_2 + 1$$

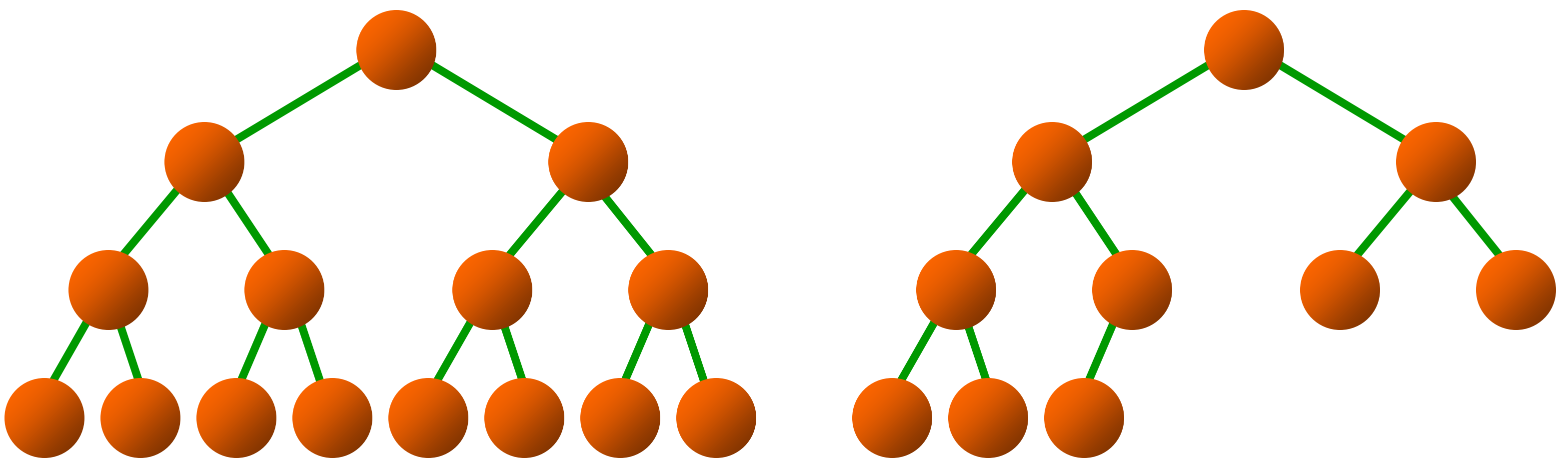
若设度为 1 的结点有 n_1 个，总结点数为 n，总边数为 e，则根据二叉树的定义，

$$n = n_0 + n_1 + n_2 \qquad e = 2n_2 + n_1 = n - 1$$

因此，有 $\cancel{2}n_2 + \cancel{n_1} = n_0 + \cancel{n_1} + \cancel{n_2} - 1$

$$n_2 = n_0 - 1 \Longrightarrow n_0 = n_2 + 1$$

- 定义1 满二叉树 (Full Binary Tree)
  - 深度为 k的满二叉树是有2^k -1个结点的二叉树。
- 定义2 完全二叉树 (Complete Binary Tree)
  - 若设二叉树的深度为 k，则共有 k 层。除第 k 层外，其它各层 (1～k-1) 的结点数都达到最大个数，第k层从右向左连续缺若干结点，这就是完全二叉树。



- 性质4　具有 n ($n \geqslant 0$) 个结点的完全二叉树的深度为 $\lceil \log_2(n+1) \rceil$

设完全二叉树的深度为 k，则有

$$\underbrace{2^{k-1}-1}_{\text{上面}k-1\text{层结点数}} < n \leqslant \underbrace{2^{k}-1}_{\text{包括第}k\text{层的最大结点数}}$$

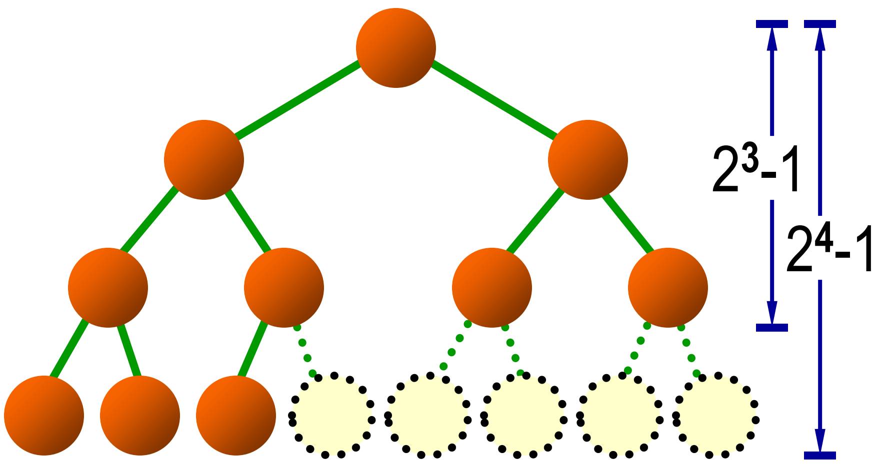
变形 $2^{k-1} < n+1 \leqslant 2^k$

取对数

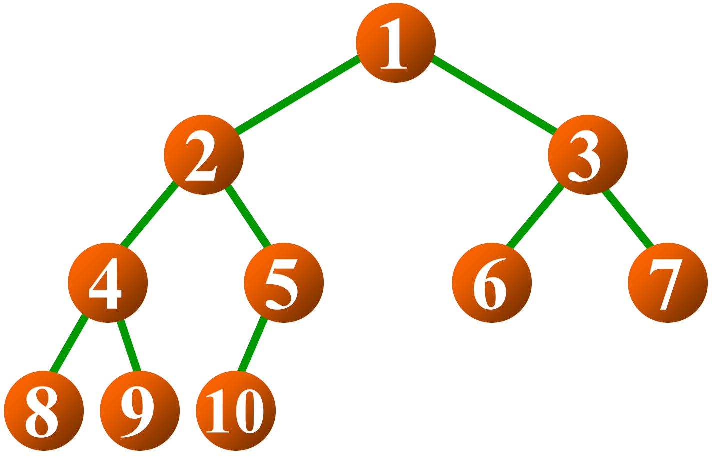
$$k-1 < \log_2(n+1) \leqslant k$$

得到

$$\lceil \log_2(n+1) \rceil = k$$



- 性质5 如将一棵有n个结点的完全二叉树自顶向下，同一层自左向右连续给结点编号1, 2, …, n，则有以下关系：
  - ✓ 若$i = 1$, 则 i 无双亲
  - ✓ 若$i > 1$, 则 i 的双亲为$\lfloor i / 2 \rfloor$
  - ✓ 若$2*i <= n$, 则 i 的左子女为 $2*i$
  - ✓ 若$2*i+1 <= n$, 则 i 的右子女为$2*i+1$
  - ✓ 若 i 为奇数, 且i != 1, 则其左兄弟为$i-1$
  - ✓ 若 i 为偶数, 且i != n, 则其右兄弟为$i+1$



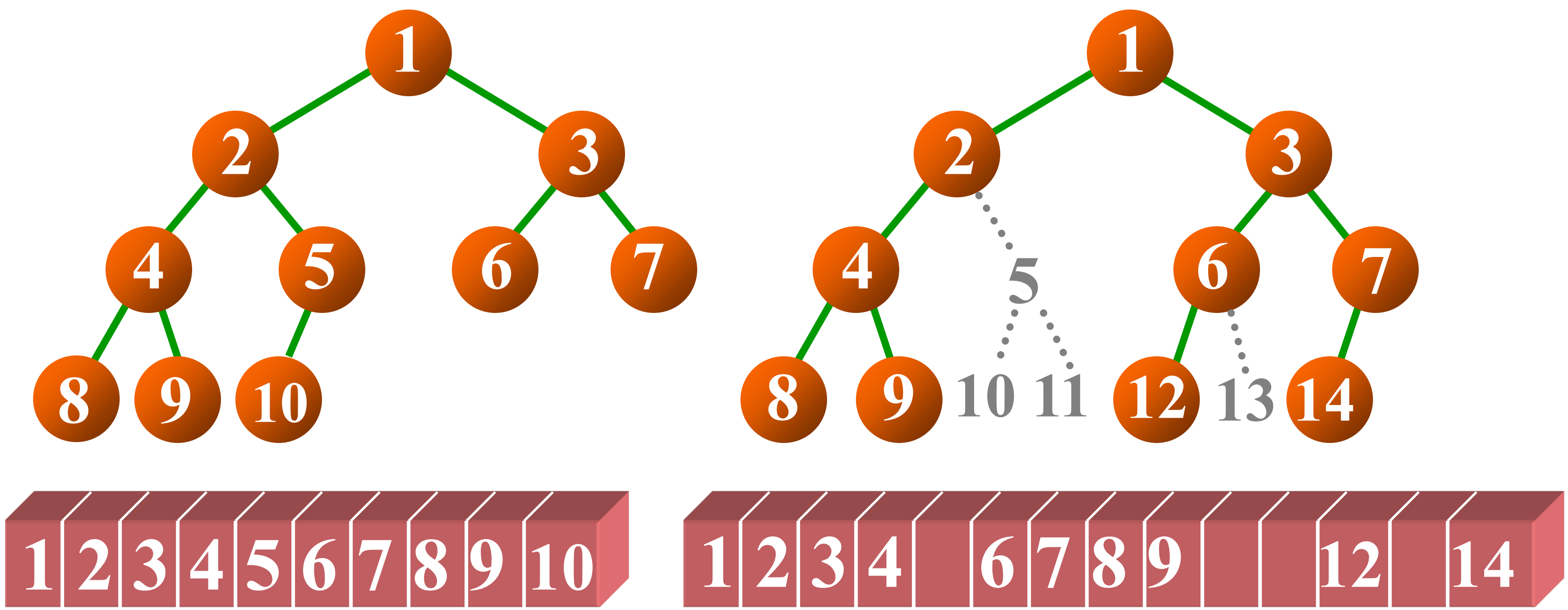
二叉树的抽象数据类型

```
template <class T>
class BinaryTree {
//对象: 结点的有限集合, 二叉树是有序树
public:
    BinaryTree ();                          //构造函数
    BinaryTree (BinTreeNode<T> *lch,
                BinTreeNode<T> *rch, T item);
      //构造函数, 以item为根, lch和rch为左、右子
      //树构造一棵二叉树
    int Height ();                          //求树深度或高度
    int Size ();                            //求树中结点个数
```

```
bool IsEmpty ();                  //判二叉树空否？
BinTreeNode<T> *Parent (BinTreeNode<T> *t);
                                  //求结点 t 的双亲
BinTreeNode<T> *LeftChild (BinTreeNode<T> *t);
                                  //求结点 t 的左子女
BinTreeNode<T> *RightChild (BinTreeNode<T> *t);
                                  //求结点 t 的右子女
bool Insert (T item);             //在树中插入新元素
bool Remove (T item);             //在树中删除元素
bool Find (T& item);              //判断item是否在树中
bool GetData (T& item);           //取得结点数据
```

```
    BinTreeNode<T> *GetRoot ();       //取根
    void PreOrder (void (*visit) (BinTreeNode<T> *t));
        //前序遍历, visit是访问函数
    void InOrder (void (*visit) (BinTreeNode<T> *t));
        //中序遍历, visit是访问函数
    void PostOrder (void (*visit) (BinTreeNode<T> *t));
        //后序遍历, (*visit)是访问函数
    void LevelOrder (void (*visit)(BinTreeNode<T> *t));
        //层次序遍历, visit是访问函数
};
```

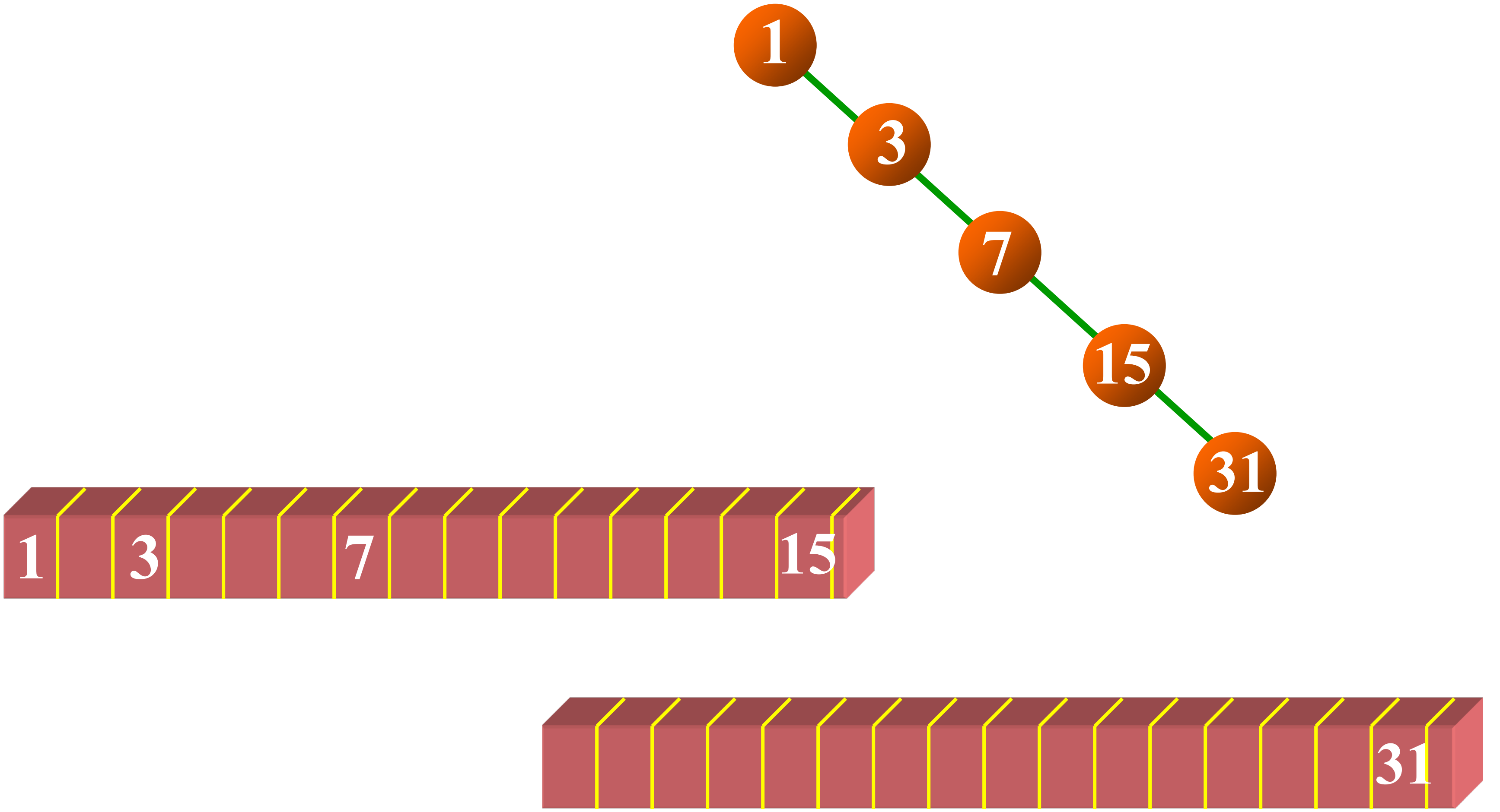
二叉树的顺序表示



完全二叉树
的顺序表示

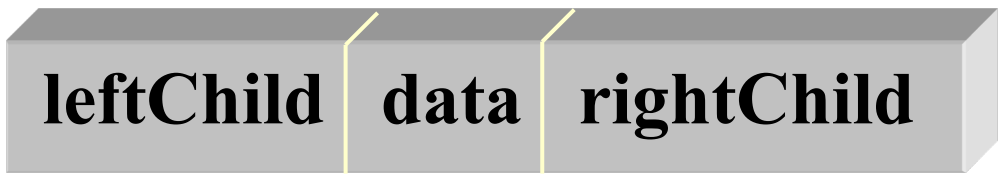
一般二叉树
的顺序表示

极端情形: 只有右单支的二叉树

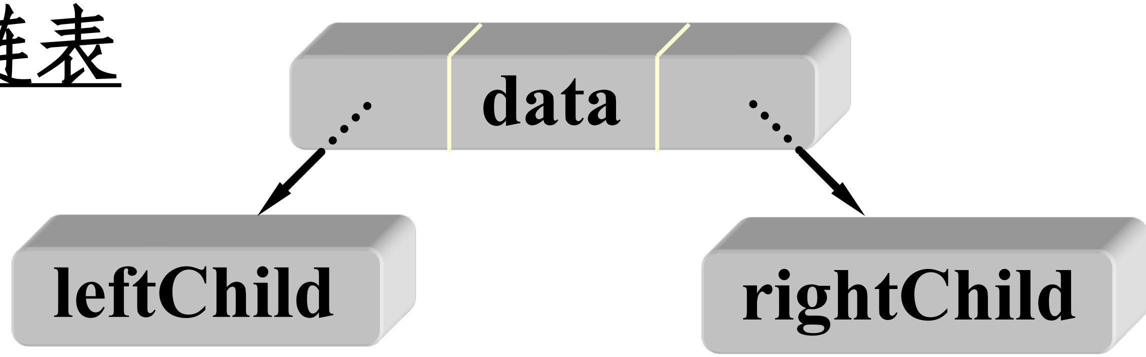


二叉树的链表表示（二叉链表）

- 二叉树结点定义：每个结点有3个成员，**data**域存储结点数据，**leftChild**和**rightChild**分别存放指向左子女和右子女的指针。

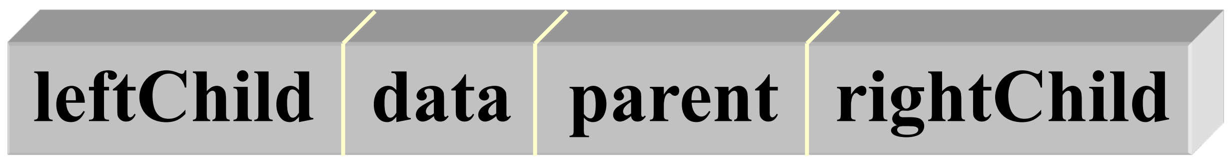


<u>二叉链表</u>

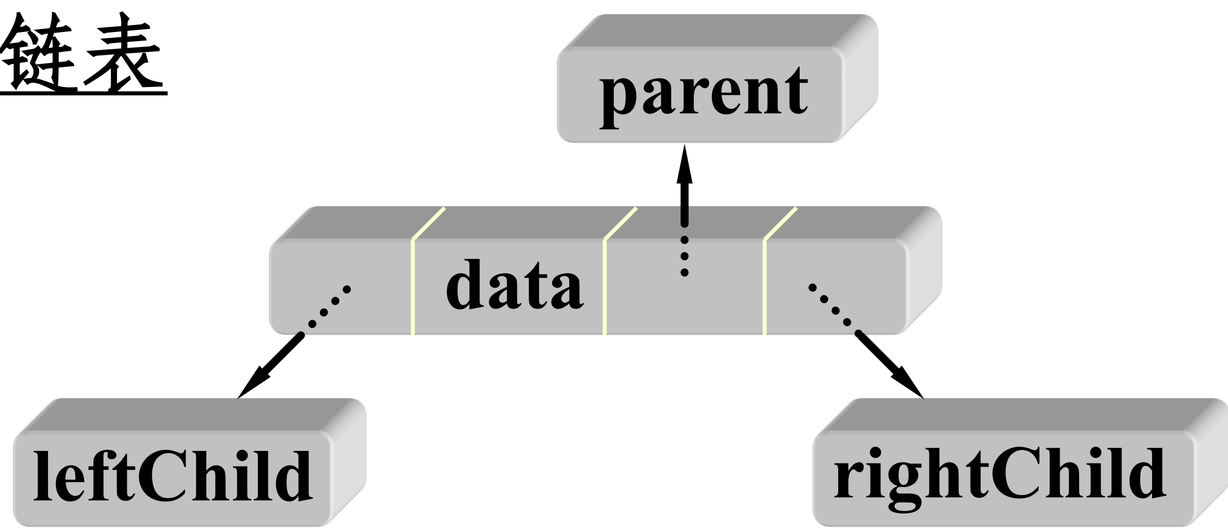


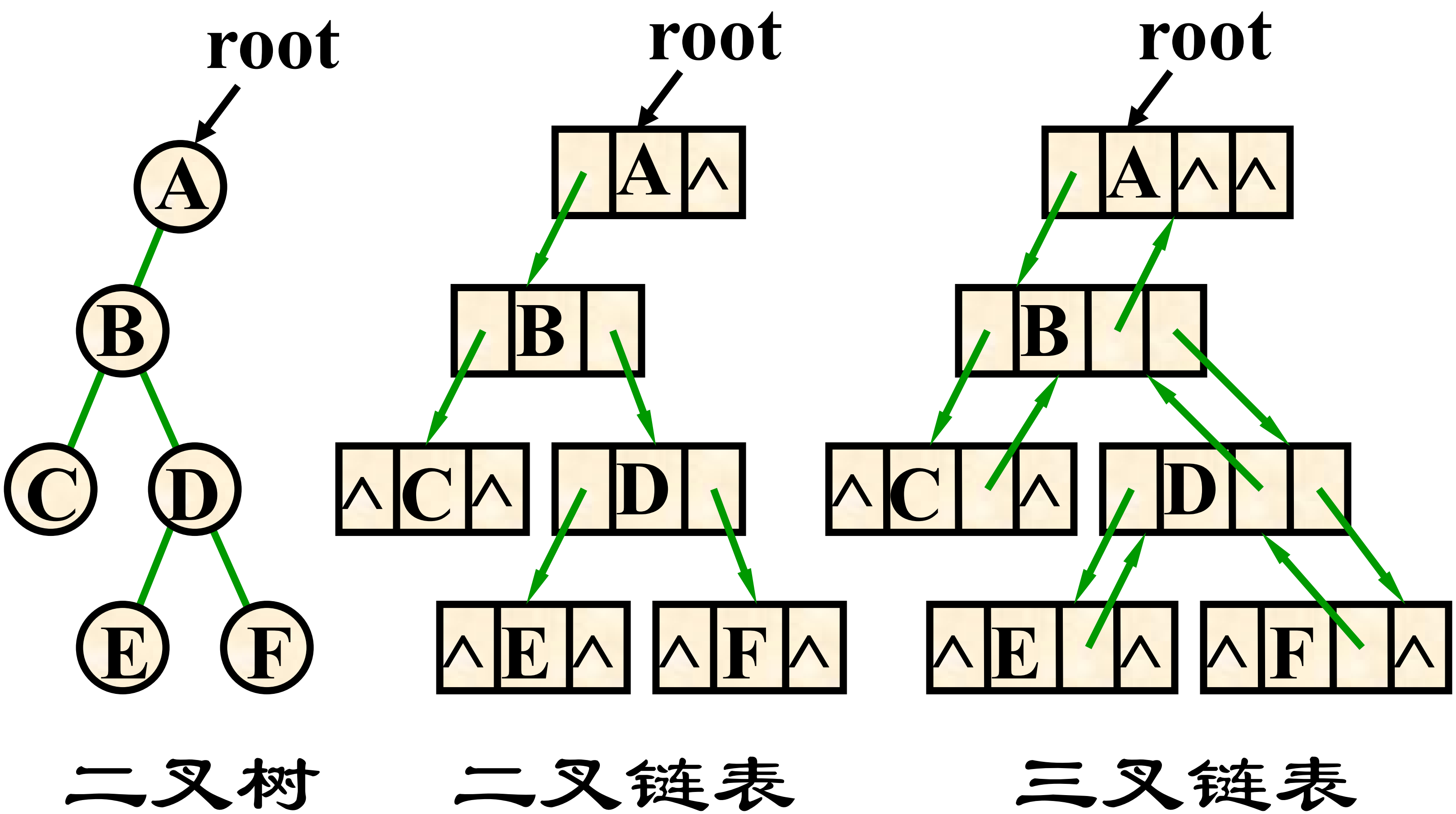
二叉树的链表表示（三叉链表）

- 每个结点增加一个指向双亲的指针parent，使得查找双亲也很方便。

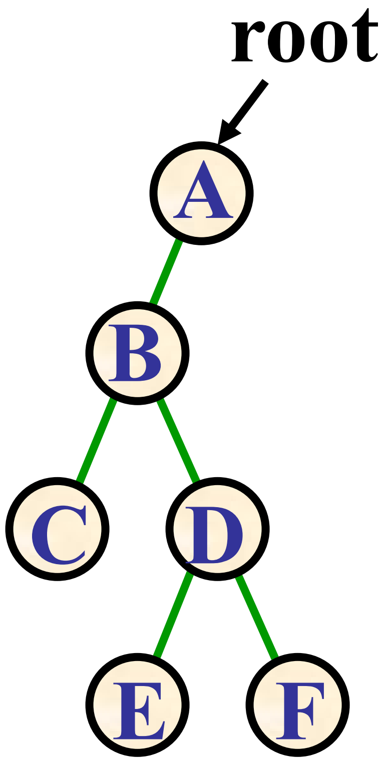


三叉链表





二叉树链表表示的示例



	data	parent	leftChild	rightChild
0	A	−1	1	−1
1	B	0	2	3
2	C	1	−1	−1
3	D	1	4	5
4	E	3	−1	−1
5	F	3	−1	−1

三叉链表的静态结构

二叉树的类定义

```
template <class T>
struct BinTreeNode {          //二叉树结点类定义
    T data;                   //数据域
    BinTreeNode<T> *leftChild, *rightChild;
                              //左子女、右子女链域
    BinTreeNode ()            //构造函数
      { leftChild = NULL;  rightChild = NULL; }
    BinTreeNode (T x, BinTreeNode<T> *l = NULL,
            BinTreeNode<T> *r = NULL)
      { data = x;  leftChild = l;  rightChild = r; }
};
```

```
template <class T>
class BinaryTree {          //二叉树类定义
public:
    BinaryTree () : root (NULL) { }        //构造函数
    BinaryTree (T value) : RefValue(value), root(NULL)
        { }                                 //构造函数
    BinaryTree (BinaryTree<T>& s);     //复制构造函数
    ～BinaryTree () { Destroy(root); }   //析构函数
    bool IsEmpty () { return root == NULL;}
                                         //判二叉树空否
    int Height () { return Height(root); }  //求树高度
    int Size () { return Size(root); }         //求结点数
```

```
BinTreeNode<T> *Parent (BinTreeNode <T> *t)
  { return (root == NULL || root ==  t) ?
      NULL : Parent (root, t); }    //返回双亲结点
BinTreeNode<T> *LeftChild (BinTreeNode<T> *t)
  { return (t != NULL) ? t->leftChild : NULL; }
                                    //返回左子女
BinTreeNode<T> *RightChild (BinTreeNode<T> *t)
  { return (t != NULL) ? t->rightChild : NULL; }
                                    //返回右子女
BinTreeNode<T> *GetRoot () const { return root; }
                                    //取根
```

```
void PreOrder (void (*visit) (BinTreeNode<T> *t))
  { PreOrder (root, visit); }           //前序遍历
void InOrder (void (*visit) (BinTreeNode<T> *t))
  { InOrder (root, visit); }            //中序遍历
void PostOrder (void (*visit) (BinTreeNode<T> *t))
  { PostOrder (root, visit); }          //后序遍历
void LevelOrder (void (*visit)(BinTreeNode<T>
*t));                                   //层次序遍历
int Insert (const T item);              //插入新元素
BinTreeNode<T> *Find (T item) const;    //搜索
```

```
protected:
    BinTreeNode<T> *root;        //二叉树的根指针
    T RefValue;                  //数据输入停止标志
    void CreateBinTree (istream& in,
                BinTreeNode<T> *& subTree);
                                 //从文件读入建树
    bool Insert (BinTreeNode<T> *& subTree,  T& x);
                                 //插入
    void Destroy (BinTreeNode<T> *& subTree);
                                 //删除
    bool Find (BinTreeNode<T> *subTree, T& x);
                                 //查找
```

```
BinTreeNode<T> *Copy (BinTreeNode<T> *r);
                                        //复制
int Height (BinTreeNode<T> *subTree);
                                        //返回树高度
int Size (BinTreeNode<T> *subTree);
                                        //返回结点数
BinTreeNode<T> *Parent (BinTreeNode<T> *
    subTree, BinTreeNode<T> *t);
                                        //返回父结点
BinTreeNode<T> *Find (BinTreeNode<T> *
    subTree, T& x) const;       //搜寻x
```

```
void Traverse (BinTreeNode<T> *subTree,
    ostream& out);                //前序遍历输出
void PreOrder (BinTreeNode<T>& subTree,
    void (*visit) (BinTreeNode<T> *t));
                                  //前序遍历
void InOrder (BinTreeNode<T>& subTree,
    void (*visit) (BinTreeNode<T> *t));
                                  //中序遍历
void PostOrder (BinTreeNode<T>& Tree,
    void (*visit) (BinTreeNode<T> *t));
                                  //后序遍历
```

```
    friend istream& operator >> (istream& in,
          BinaryTree<T>& Tree);    //重载操作：输入
    friend ostream& operator << (ostream& out,
          BinaryTree<T>& Tree);    //重载操作：输出
};
```

部分成员函数的实现

```
template <class T>
BinTreeNode<T> *BinaryTree<T>::
Parent (BinTreeNode <T> *subTree,
        BinTreeNode <T> *t) {
```

```
//私有函数: 从结点 subTree 开始, 搜索结点 t 的双
//亲, 若找到则返回双亲结点地址, 否则返回NULL
    if (subTree == NULL) return NULL;
    if (subTree->leftChild == t ||
            subTree->rightChild == t )
        return subTree;     //找到, 返回父结点地址
    BinTreeNode <T> *p;
    if ((p = Parent (subTree->leftChild, t)) != NULL)
        return p;           //递归在左子树中搜索
    else return Parent (subTree->rightChild, t);
                            //递归在右子树中搜索
};
```

```
template<class T>
void BinaryTree<T>::
Destroy (BinTreeNode<T> * subTree) {
//私有函数: 删除根为subTree的子树
    if (subTree != NULL) {
        Destroy (subTree->leftChild);     //删除左子树
        Destroy (subTree->rightChild);   //删除右子树
        delete subTree;                   //删除根结点
    }
};
```

二叉树遍历

- 二叉树的遍历就是按某种次序访问树中的结点，要求每个结点访问一次且仅访问一次。

- 设访问根结点记作 V

  遍历根的左子树记作 L

  遍历根的右子树记作 R

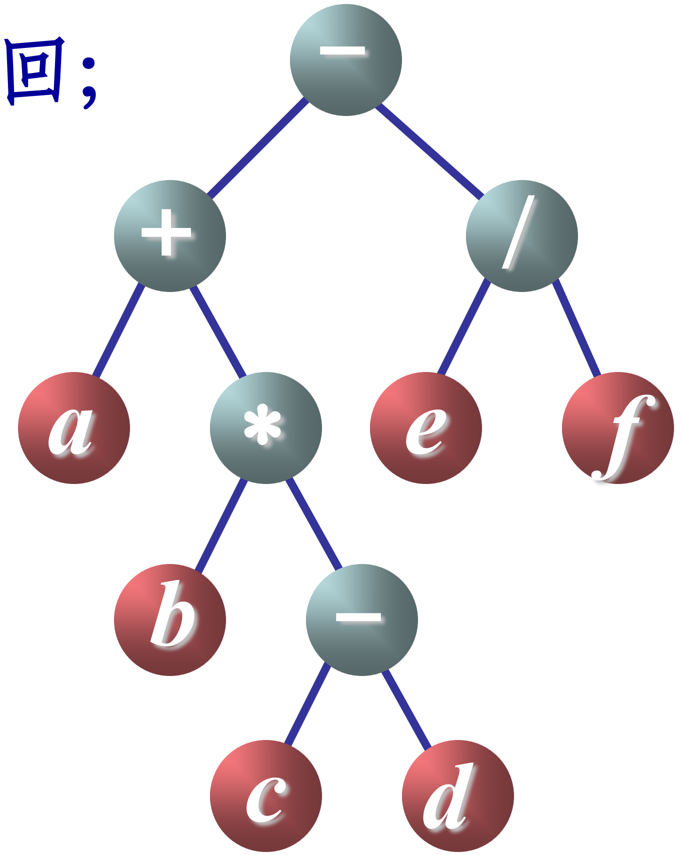
- 则可能的遍历次序有

前序	VLR	镜像	VRL
中序	LVR	镜像	RVL
后序	LRV	镜像	RLV

中序遍历 (Inorder Traversal)

中序遍历二叉树算法的框架是:

- 若二叉树为空，则直接返回;
- 否则
  - 中序遍历左子树 (L);
  - 访问根结点 (V);
  - 中序遍历右子树 (R)。



遍历结果

$$a + b * c - d - e / f$$

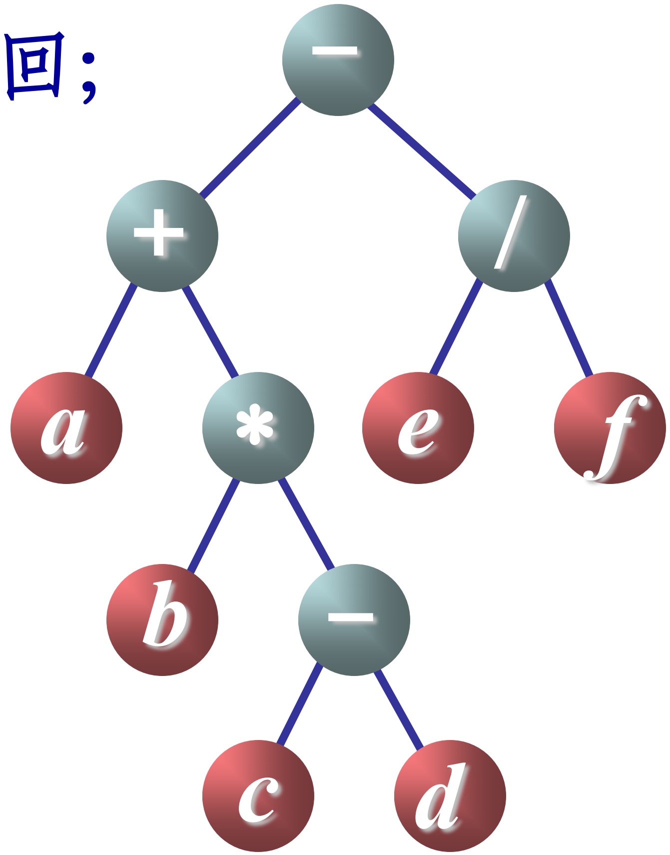
二叉树递归的中序遍历算法

```
template <class T>
void BinaryTree<T>::InOrder (BinTreeNode<T> *
   subTree, void (*visit) (BinTreeNode<T> *t)) {
    if (subTree != NULL) {
        InOrder (subTree->leftChild, visit);
                                        //遍历左子树
        visit (subTree);                //访问根结点
        InOrder (subTree->rightChild, visit);
                                        //遍历右子树
    }
};
```

前序遍历 (Preorder Traversal)

前序遍历二叉树算法的框架是:

- 若二叉树为空，则直接返回;
- 否则
  - 访问根结点 **(V)**;
  - 前序遍历左子树 **(L)**;
  - 前序遍历右子树 **(R)**。



遍历结果

$- + a * b - c\ d\ /\ e\ f$

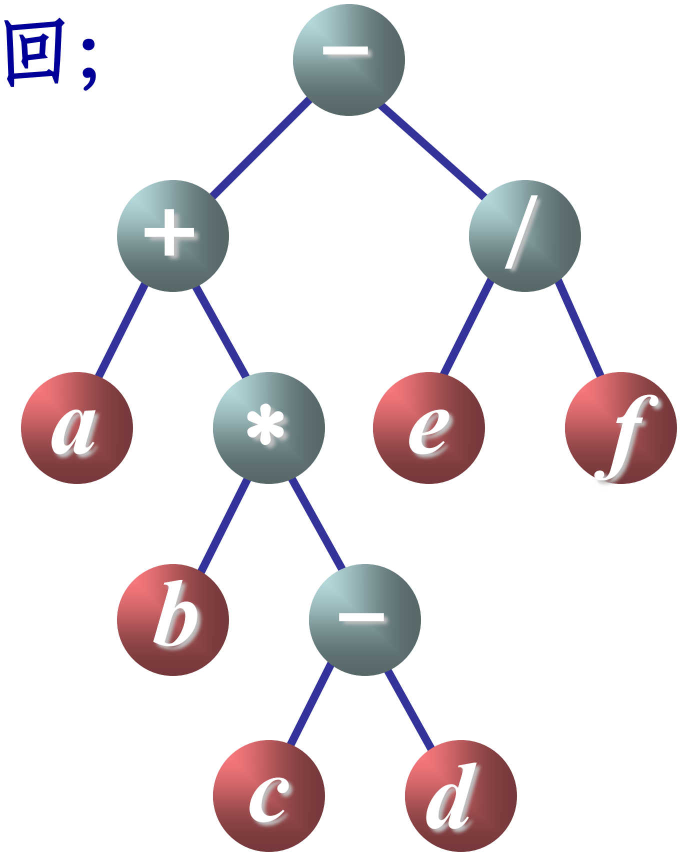
二叉树递归的前序遍历算法

```
template <class T>
void BinaryTree<T>::PreOrder (BinTreeNode<T> *
   subTree, void (*visit) (BinTreeNode<T> *t)) {
    if (subTree != NULL) {
        visit (subTree);                   //访问根结点
        PreOrder (subTree->leftChild, visit);
                                           //遍历左子树
        PreOrder (subTree->rightChild, visit);
                                           //遍历右子树
    }
};
```

后序遍历 (Postorder Traversal)

后序遍历二叉树算法的框架是:

- 若二叉树为空，则直接返回;
- 否则
  - 后序遍历左子树 **(L)**;
  - 后序遍历右子树 **(R)**;
  - 访问根结点 **(V)**。



遍历结果

a b c d **–** * + *e f* / **–**

二叉树递归的后序遍历算法

```
template <class T>
void BinaryTree<T>::PostOrder (BinTreeNode<T> *
   subTree, void (*visit) (BinTreeNode<T> *t ) {
    if (subTree != NULL ) {
        PostOrder (subTree->leftChild, visit);
                                        //遍历左子树
        PostOrder (subTree->rightChild, visit);
                                        //遍历右子树
        visit (subTree);                //访问根结点
    }
};
```

应用二叉树遍历的示例

```
template <class T>
int BinaryTree<T>::Size (BinTreeNode<T> *
    subTree) const {
//私有函数：利用二叉树后序遍历算法计算二叉
//树的结点个数
    if (subTree == NULL) return 0;          //空树
    else return  Size (subTree->leftChild)
                 + Size (subTree->rightChild) + 1;
};
```

应用二叉树遍历的示例

```
template <class T>
int  BinaryTree<T>::Height ( BinTreeNode<T> *
   subTree) const {
//私有函数：利用二叉树后序遍历算法计算二叉
//树的高度或深度
    if (subTree == NULL) return 0;   //空树高度为0
    else {
        int i = Height (subTree->leftChild);
        int j = Height (subTree->rightChild);
        return (i < j) ? j+1 : i+1;
   };
```

利用二叉树前序遍历建立二叉树

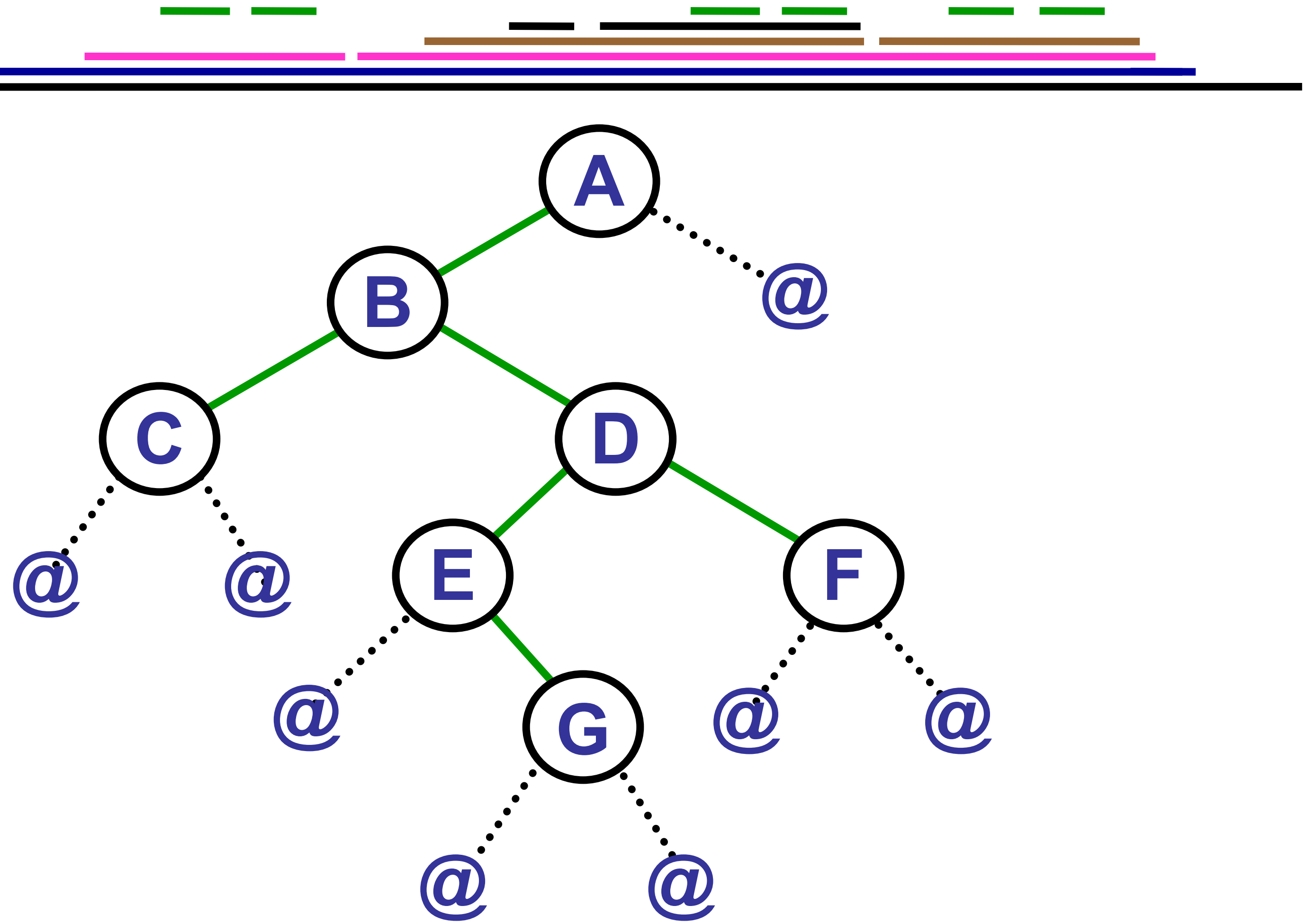
- 以递归方式建立二叉树。

- 输入结点值的顺序必须对应二叉树结点前序遍历的顺序，并约定以输入序列中不可能出现的值作为空结点的值以结束递归，此值在**RefValue**中。例如用“**@**”或用“**-1**”表示字符序列或正整数序列的空结点。

利用二叉树前序遍历建立二叉树

如图所示的二叉树的前序遍历顺序为

A B C @ @ D E @ G @ @ F @ @ @



利用二叉树前序遍历建立二叉树

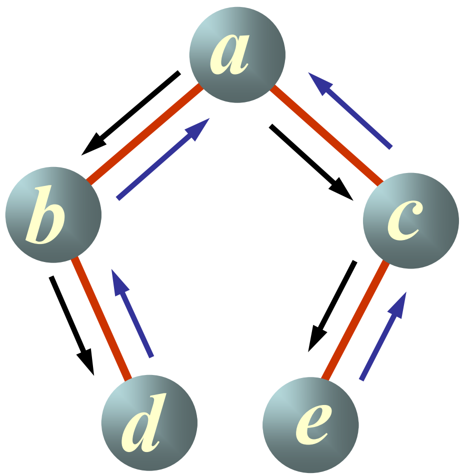
```
template<class T>
void BinaryTree<T>::CreateBinTree (ifstream& in,
   BinTreeNode<T> *& subTree) {
//私有函数: 以递归方式建立二叉树。
    T item;
    if ( !in.eof () ) {                //未读完, 读入并建树
        in >> item;                    //读入根结点的值
        if (item != RefValue) {
            subTree = new BinTreeNode<T>(item);
                                //建立根结点
            if (subTree == NULL)
              {cerr << "存储分配错!" << endl;  exit (1);}
```

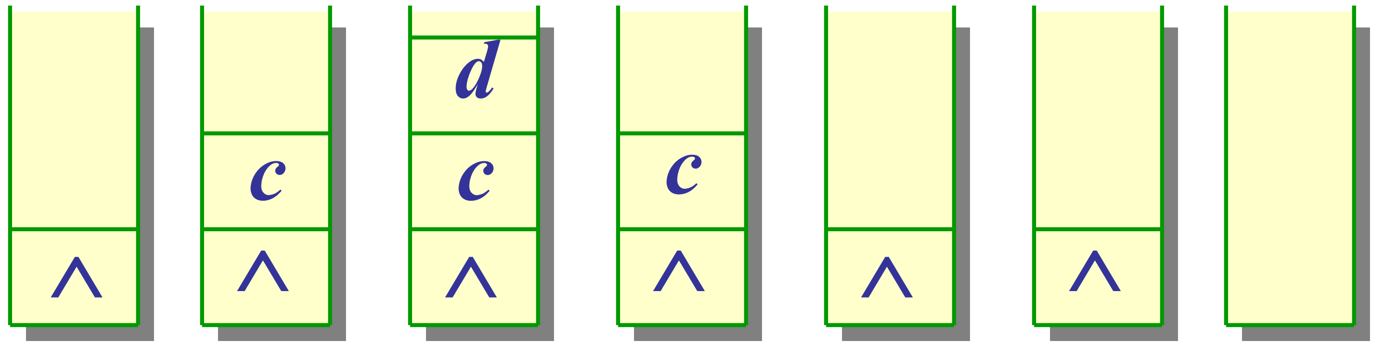
利用二叉树前序遍历建立二叉树

```
        CreateBinTree (in, subTree->leftChild);
            //递归建立左子树

        CreateBinTree (in, subTree->rightChild);
            //递归建立右子树
    }
    else subTree = NULL;
            //封闭指向空子树的指针
  }
};
```

利用栈的前序遍历非递归算法





初始化

访问 a 进栈 c 左进 b

访问 b 进栈 d 左进 空

退栈 d 访问 d 左进 空

退栈 c 访问 c 左进 e

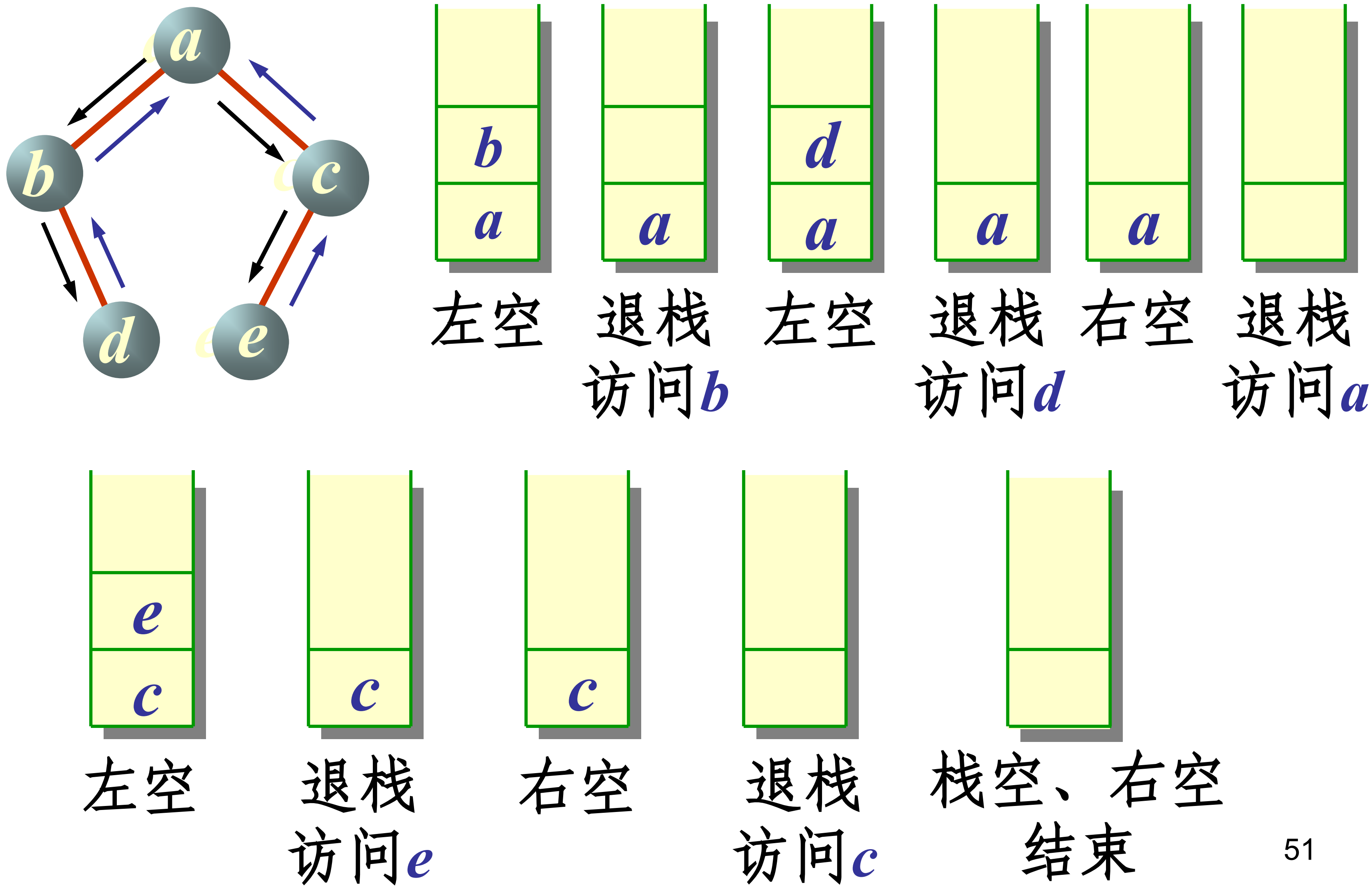
访问 e 左进 空

∧ 退栈 结束

利用栈的前序遍历非递归算法

```
template <class T>
void BinaryTree<T>::
PreOrder (void (*visit) (BinTreeNode<T> *t) ) {
    stack<BinTreeNode<T>*> S;
    BinTreeNode<T> *p = t;
    S.Push (NULL);
    while (p != NULL) {
        visit(p);                        //访问结点
        if (p->rightChild != NULL)
            S.Push (p->rightChild);  //预留右指针在栈中
        if (p->leftChild != NULL)
            p = p->leftChild;      //进左子树
        else S.Pop(p);               //左子树为空
    }
};
```

利用栈的中序遍历非递归算法



利用栈的中序遍历非递归算法

```
template <class T>
void BinaryTree<T>::
InOrder (void (*visit) (BinTreeNode<T> *t)) {
    stack<BinTreeNode<T>*> S;
    BinTreeNode<T> *p = t;
   do {
        while (p != NULL) {    //遍历指针向左下移动
            S.Push (p);              //该子树沿途结点进栈
            p = p->leftChild;
        }
        if (!S.IsEmpty()) {                //栈不空时退栈
            S.Pop (p);  visit (p); //退栈, 访问
            p = p->rightChild;  //遍历指针进到右子女
        }
    } while (p != NULL || !S.IsEmpty ());
};
```

利用栈的后序遍历非递归算法

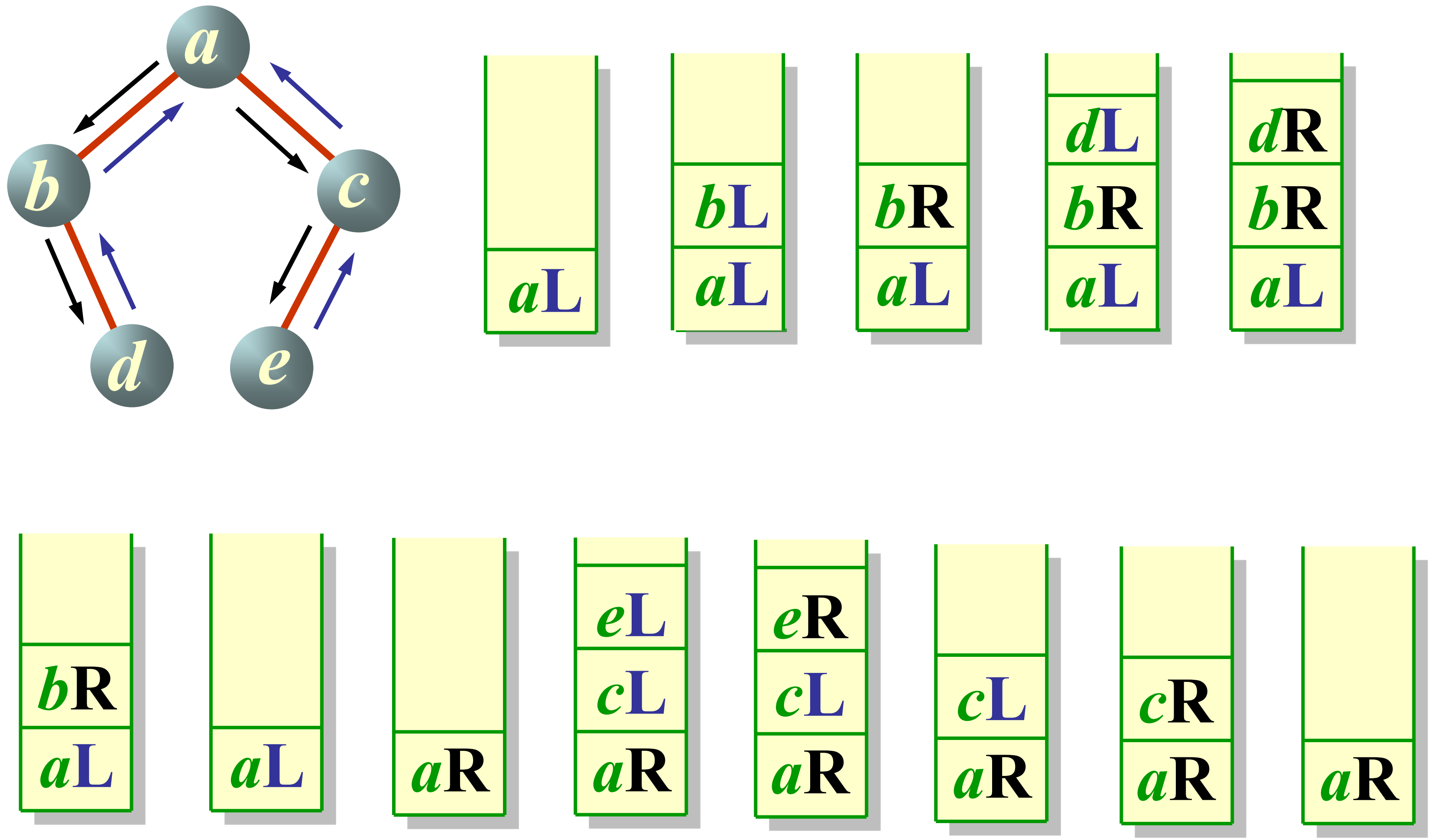
- 在后序遍历过程中所用栈的结点定义

ptr	tag{L,R}

```
template <class T>
struct stkNode {
    BinTreeNode<T> *ptr;          //树结点指针
    enum tag {L, R};              //退栈标记
    stkNode (BinTreeNode<T> *N = NULL) :
        ptr(N), tag(L) { }        //构造函数
};
```

- **tag = L**，表示从左子树退回还要遍历右子树；
  tag = R，表示从右子树退回要访问根结点。

利用栈的后序遍历非递归算法



利用栈的后序遍历非递归算法

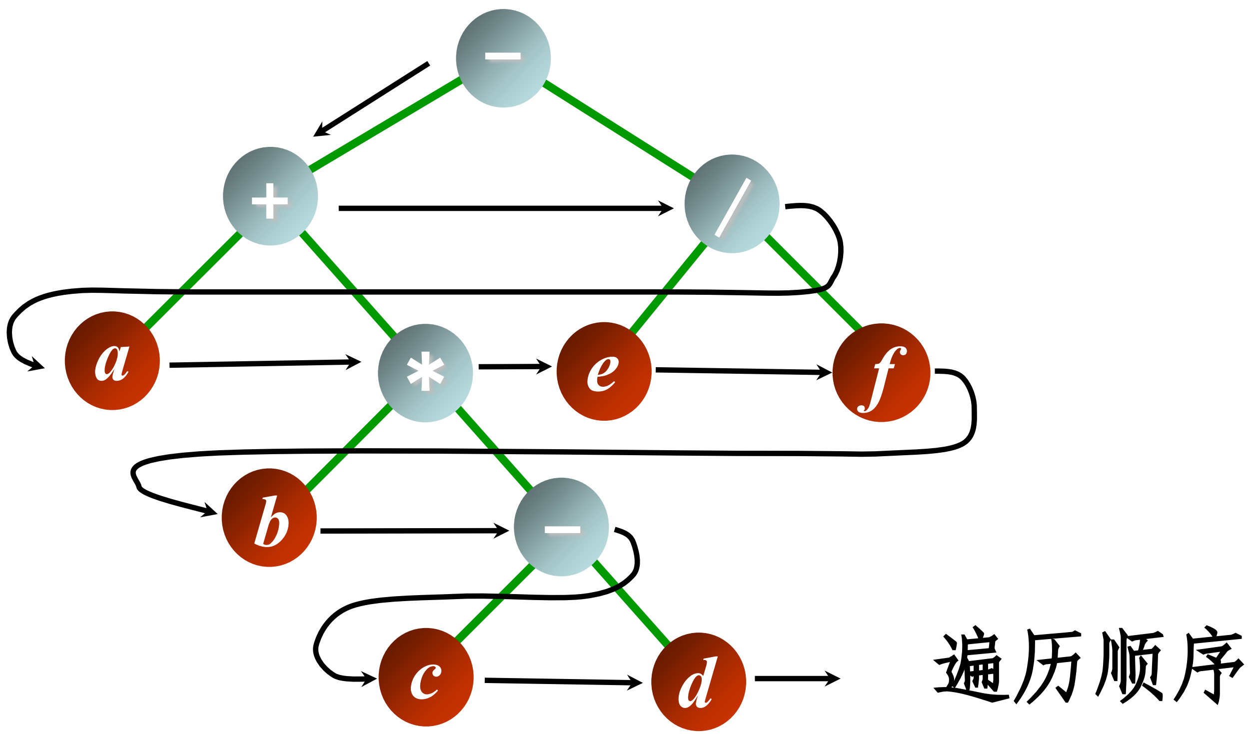
```
template <class T>
void BinaryTree<T>::
PostOrder (void (*visit) (BinTreeNode<T> *t) {
    Stack<stkNode<T>> S;   stkNode<T> w;
    BinTreeNode<T> * p = t;     //p是遍历指针
    do {
        while (p != NULL) {
            w.ptr = p;  w.tag = L;  S.Push (w);
            p = p->leftChild;
        }
        int continue1 = 1;     //继续循环标记, 用于R
```

利用栈的后序遍历非递归算法

```
    while (continue1 && !S.IsEmpty ()) {
        S.Pop (w);  p = w.ptr;
        switch (w.tag) {          //判断栈顶的tag标记
            case L:  w.tag = R;  S.Push (w);
                     continue1 = 0;
                     p = p->rightChild;  break;
            case R:  visit (p);   break;
        }
    }
  } while (!S.IsEmpty ());       //继续遍历其他结点
  cout << endl;
};
```

层次遍历二叉树的算法

- 层次遍历二叉树就是从根结点开始，按层次逐层遍历，如图：

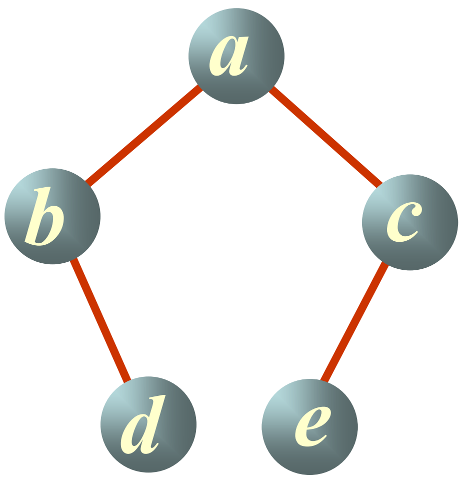


层次遍历二叉树的算法

- 这种遍历需要使用一个先进先出队列，在处理上一层时，将其下一层的结点直接进到队列（的队尾）。在上一层结点遍历完后，下一层结点正好处于队列的队头，可以继续访问它们。

- 算法是非递归的。

层次遍历二叉树的算法



Q	*a*				

*a*进队

Q		*b*	*c*		

*a*出队, 访问*a*
*b*进队, *c* 进队

Q			*c*	*d*	

*b*出队, 访问*b*
*d*进队

Q				*d*	*e*

*c*出队,访问*c*
*e*进队

Q					*e*

*d*出队,访问*d*

Q					

*e*出队,访问*e*

层次遍历二叉树的算法

```
template <class T>
void BinaryTree<T>::
LevelOrder (void (*visit) (BinTreeNode<T> *t)) {
    if (t == NULL) return;
    Queue<BinTreeNode<T> * > Q;
    BinTreeNode<T> *p = t;
    Q.EnQueue (p);
    while (!Q.IsEmpty ()) {
        Q.DeQueue (p); visit (p);
        if (p->leftChild != NULL)    Q.EnQueue (p->leftChild);
        if (p->rightChild != NULL)   Q.EnQueue (p->rightChild);
    }
};
```

- 由给定的前序序列和中序序列能够唯一地确定一棵二叉树

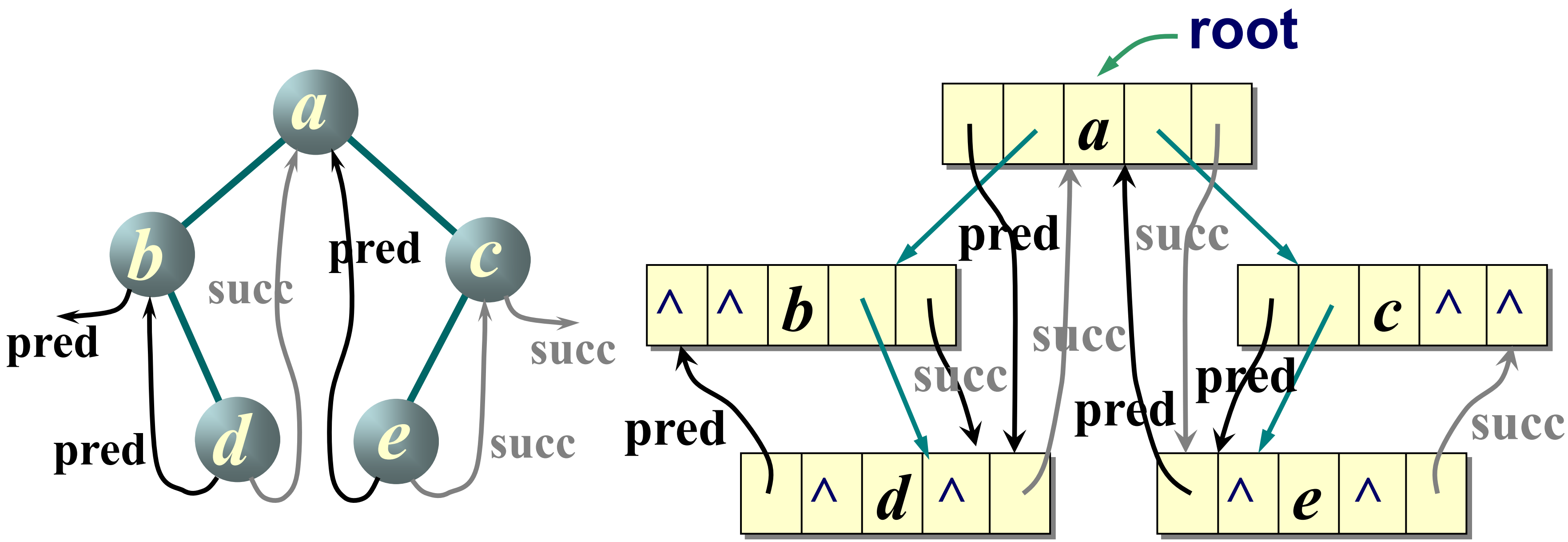
- 例如：假定一棵二叉树的前序序列为**AHBFDECKG**，中序序列为**HBDFAEKCG**

线索化二叉树
(Threaded Binary Tree)

- 又称为穿线树。
- 通过二叉树的遍历，可将二叉树中所有结点的数据排列在一个线性序列中，可以找到某数据在这种排列下它的前驱和后继。
- 希望不必每次都通过遍历找出这样的线性序列。只要事先做预处理，将某种遍历顺序下的前驱、后继关系记在树的存储结构中，以后就可以高效地找出某结点的前驱、后继。

线索 (Thread)

pred	leftChild	data	rightChild	succ



方法一：增加 *Pred* 指针和 *Succ* 指针的二叉树

- 这种设计的缺点是每个结点增加两个指针，当结点数很大时存储消耗较大。

- 对于原来的二叉链表结构，一棵n个结点的二叉树共有$2n$个指针域，而非空的指针域为n-1个，因此，仍有n+1个指针域没有利用起来。

方法二：增加左右线索标志的二叉树

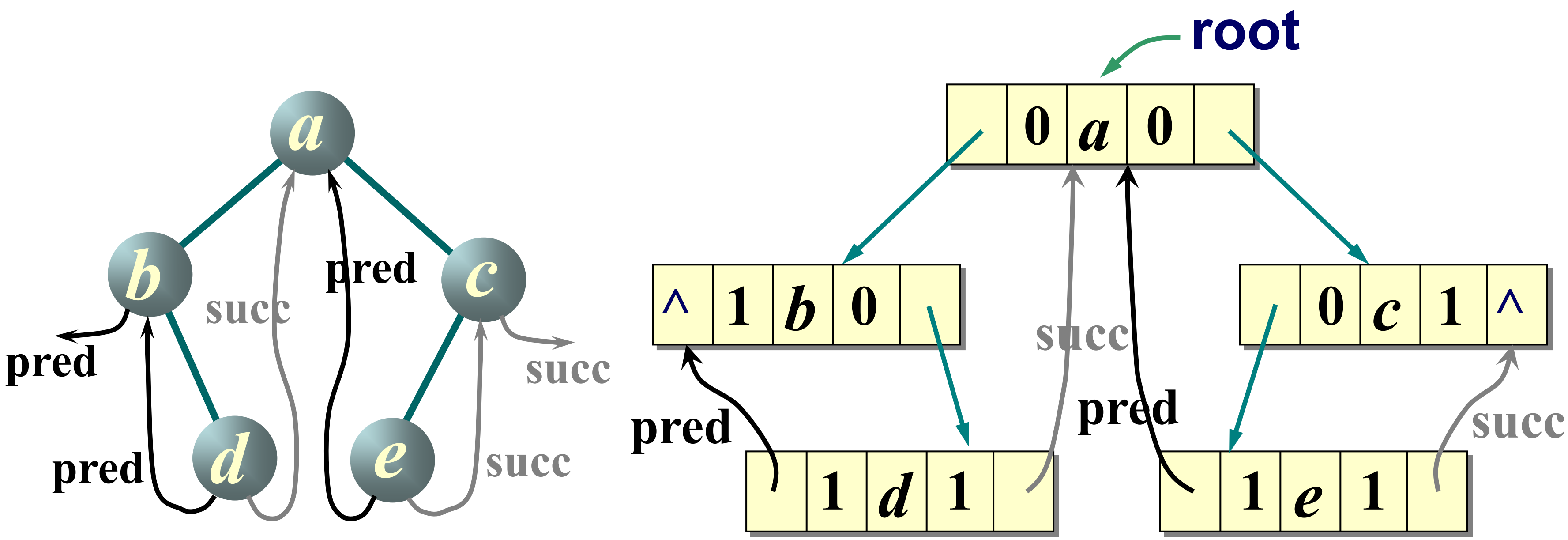
- 改造树结点，将 pred 指针和 succ 指针压缩到 leftChild 和 rightChild 的空闲指针中，并增设两个标志 ltag 和 rtag，指明指针是指示子女还是前驱／后继。后者称为线索。

leftChild	ltag	data	rtag	rightChild

- ltag (或rtag) = 0，表示相应指针指示左子女（或右子女结点）； 当ltag (或rtag) = 1, 表示相应指针为前驱（或后继）线索。

线索化二叉树及其链表表示

leftChild	ltag	data	rtag	rightChild



ltag= 0, *leftChild*为左子女指针

ltag= 1, *leftChild*为前驱线索

rtag = 0, *rightChild*为右子女指针

rtag = 1, *rightChild*为后继线索

线索化二叉树的类定义

```
template <class T>
struct ThreadNode {              //线索二叉树的结点类
    int ltag, rtag;              //线索标志
    ThreadNode<T> *leftChild, *rightChild;
                                 //线索或子女指针
    T data;                      //结点数据
    ThreadNode ( const T item)           //构造函数
        : data(item), leftChild (NULL),
          rightChild (NULL), ltag(0), rtag(0) {}
};
```

```
template <class T>
class ThreadTree {                    //线索化二叉树类
protected:
    ThreadNode<T> *root;          //树的根指针
    void createInThread (ThreadNode<T> *current,
          ThreadNode<T> *& pre);
      //中序遍历建立线索二叉树
    ThreadNode<T> *parent (ThreadNode<T> *t);
      //寻找结点t的双亲结点
public:
    ThreadTree () : root (NULL) { }     //构造函数
```

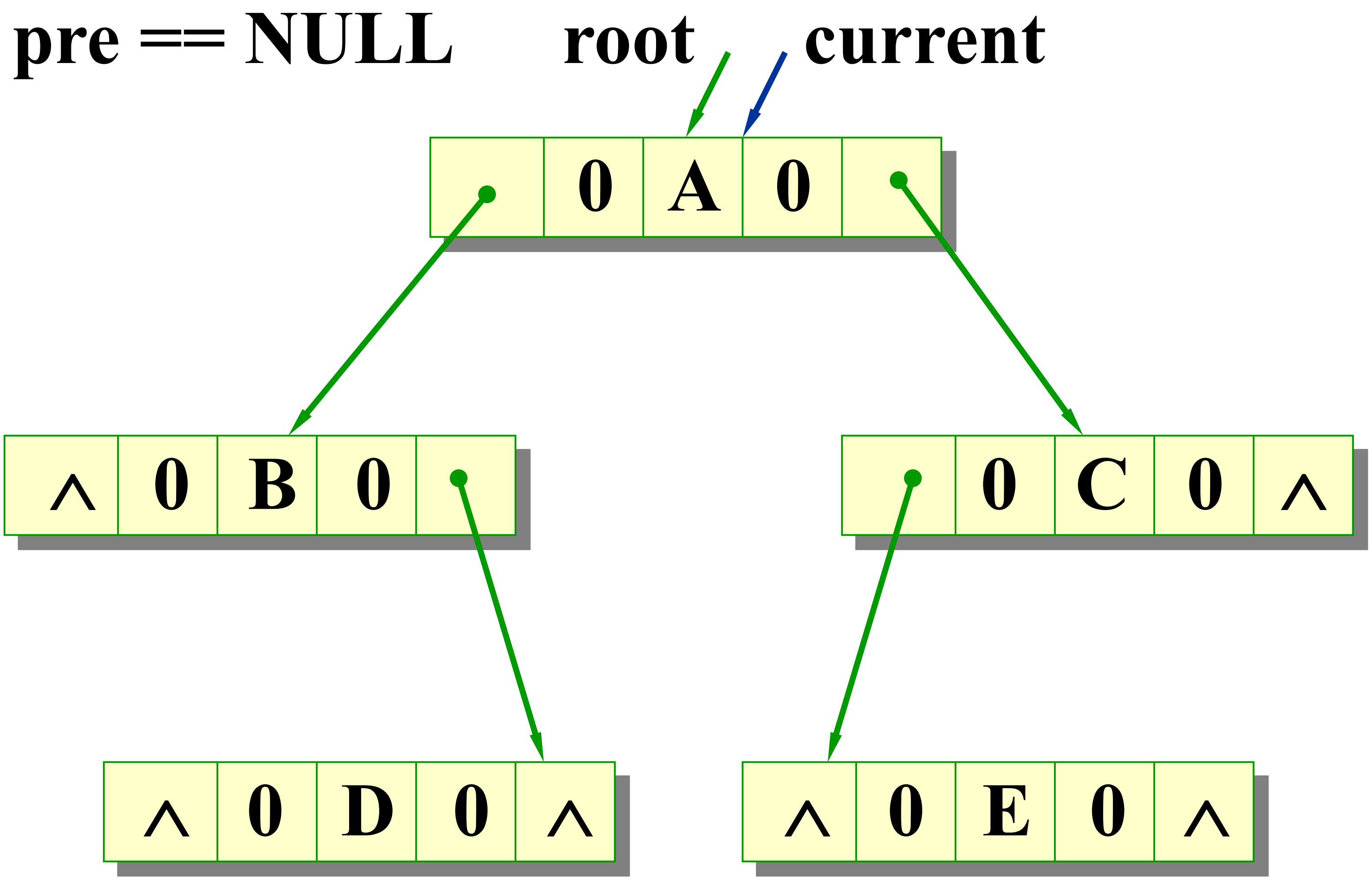
```
void createInThread();        //建立中序线索二叉树
ThreadNode<T> *First (ThreadNode<T> *current);
    //寻找中序下第一个结点
ThreadNode<T> *Last (ThreadNode<T> *current);
    //寻找中序下最后一个结点
ThreadNode<T> *Next (ThreadNode<T> *current);
    //寻找结点在中序下的后继结点
ThreadNode<T> *Prior (ThreadNode<T> *current);
    //寻找结点在中序下的前驱结点
    ………
};
```

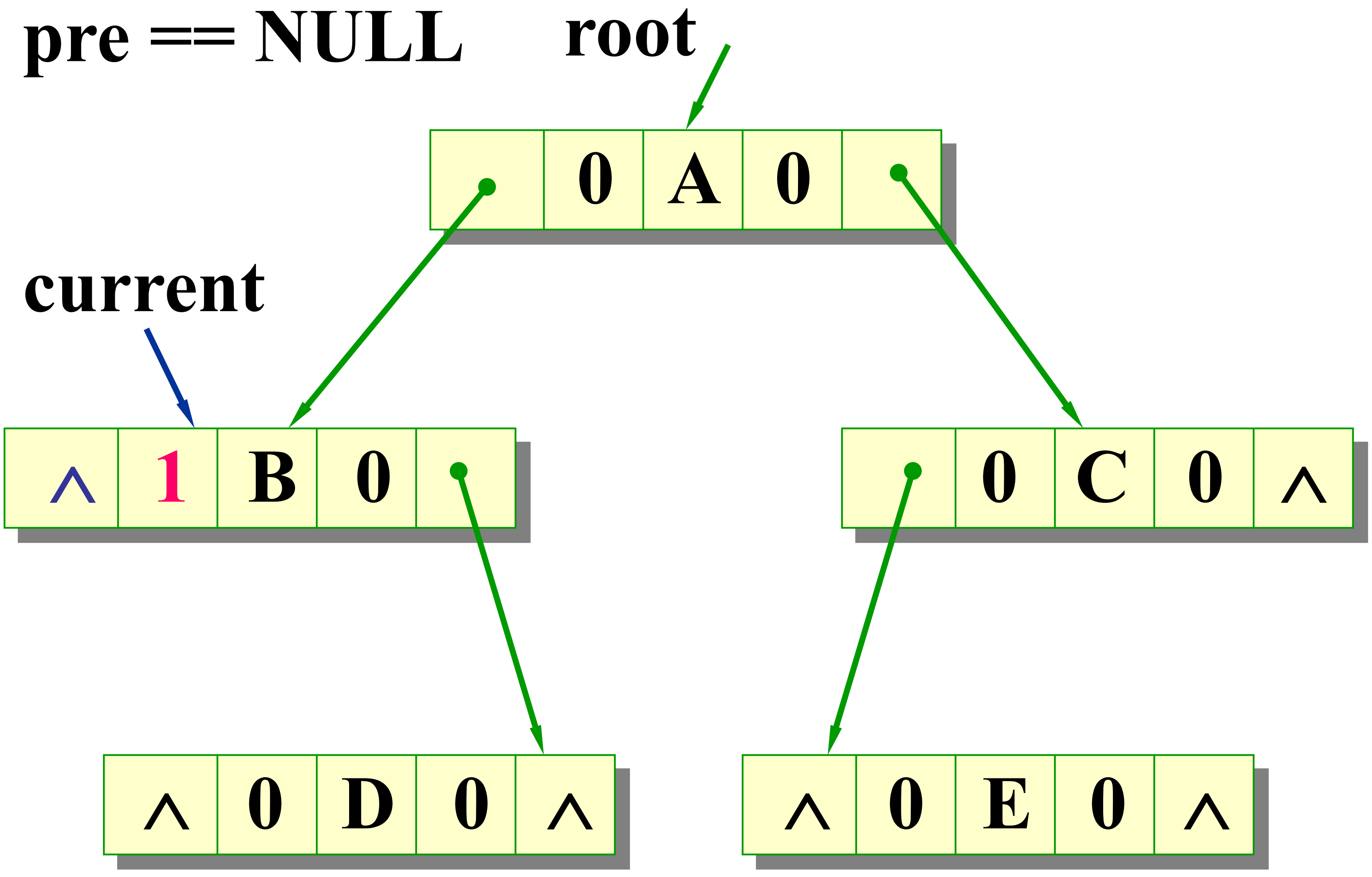
通过中序遍历建立中序线索化二叉树

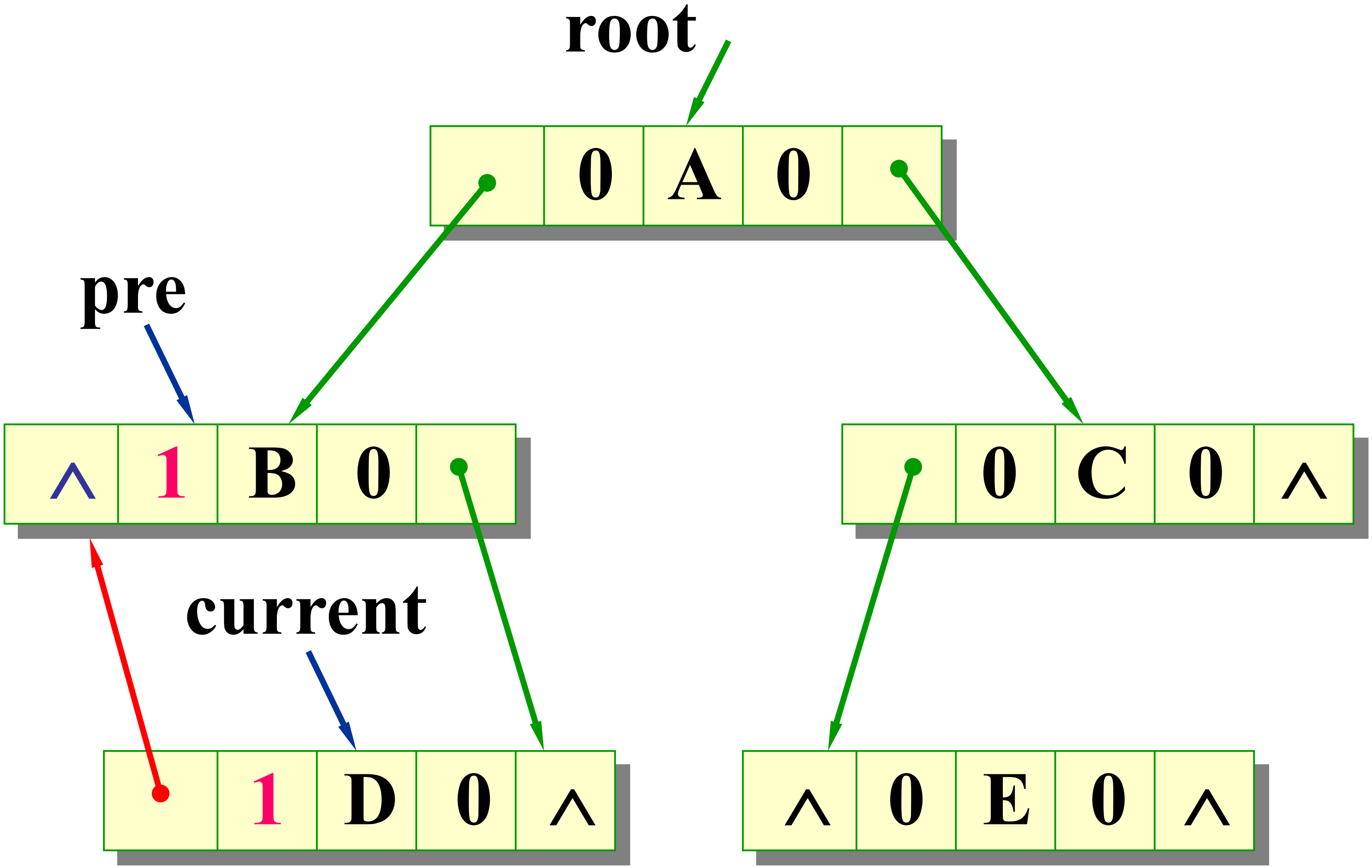
```
template <class T>
void ThreadTree<T>::createInThread () {
    ThreadNode<T> *pre = NULL;   //前驱结点指针
    if (root != NULL) {          //非空二叉树, 线索化
        createInThread (root, pre);
           //中序遍历线索化二叉树
        pre->rightChild = NULL;  pre->rtag = 1;
           //后处理中序最后一个结点
    }
};
```

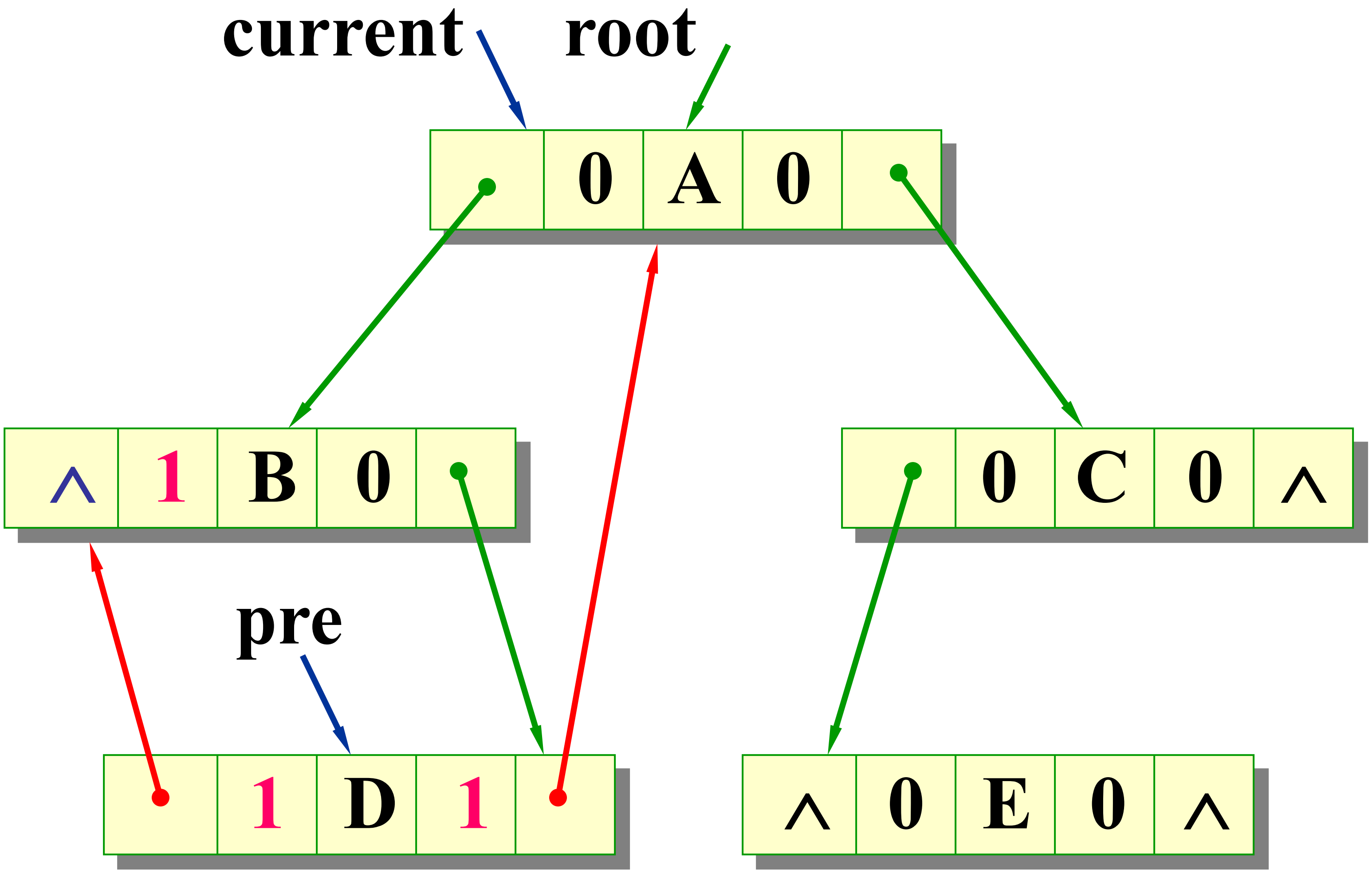
```
template <class T>
void ThreadTree<T>::
createInThread (ThreadNode<T> *current,
   ThreadNode<T> *& pre) {
//通过中序遍历, 对二叉树进行线索化
    if (current == NULL) return;
    createInThread (current->leftChild, pre);
        //递归, 左子树线索化
    if (current->leftChild == NULL) {
        //建立当前结点的前驱线索
        current->leftChild = pre;   current->ltag = 1;
    }
```

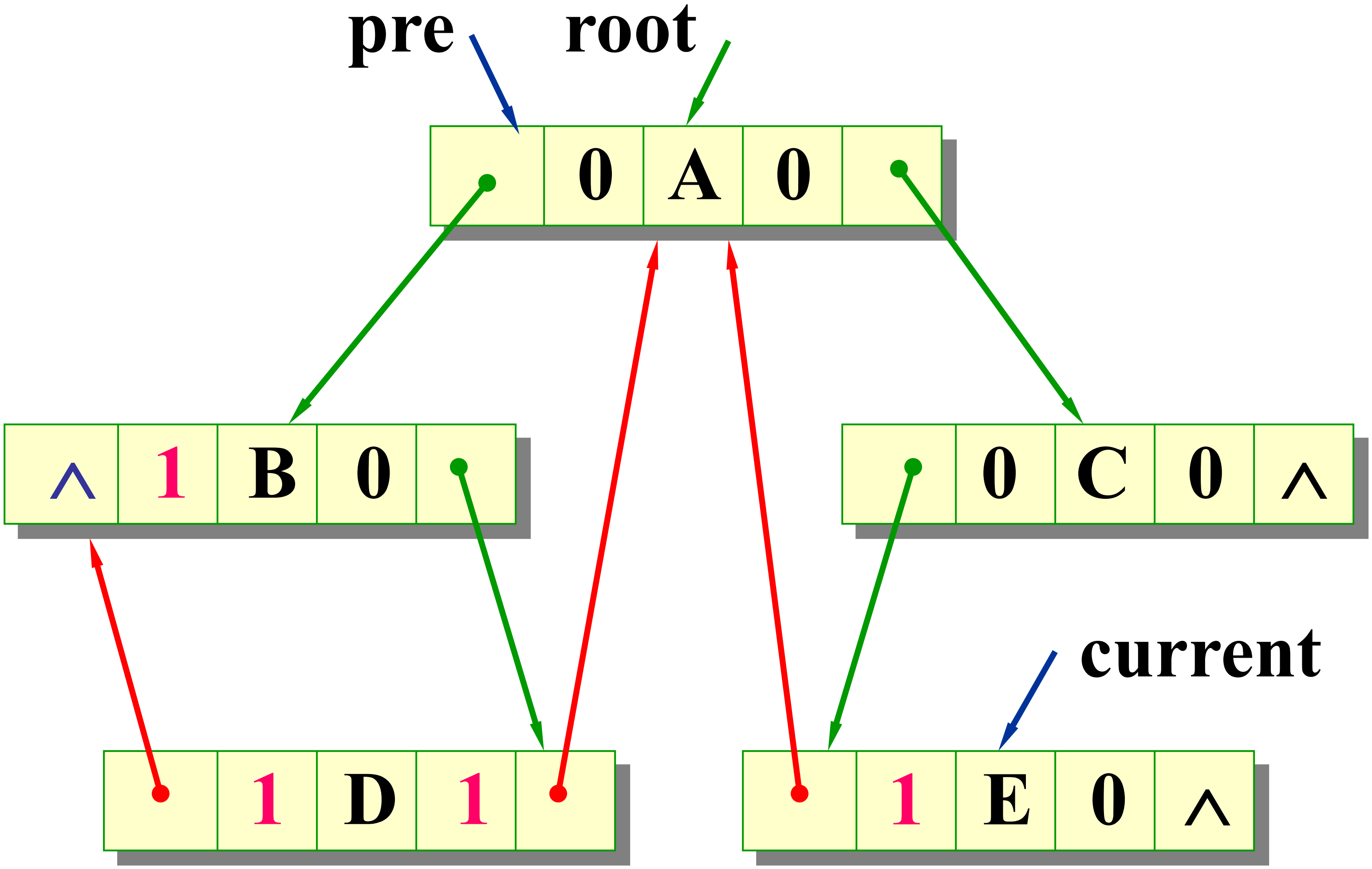
```
if (pre != NULL && pre->rightChild == NULL)
        //建立前驱结点的后继线索
   { pre->rightChild = current;   pre->rtag = 1; }
pre = current;
        //前驱跟上,当前指针向前遍历
createInThread (current->rightChild, pre);
        //递归, 右子树线索化
};
```

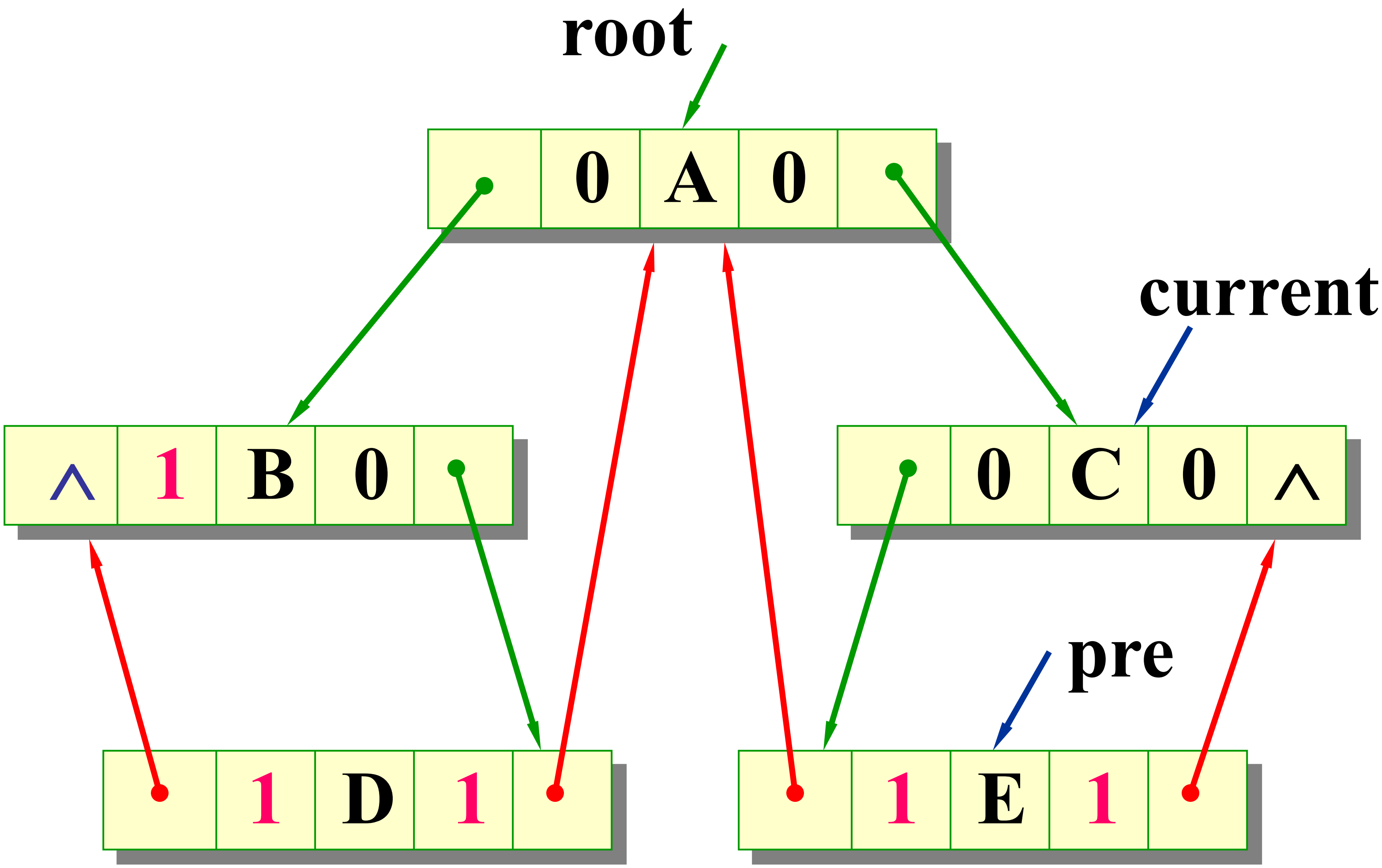

pre == NULL
root
current
0
A
0
∧
0
B
0
0
C
0
∧
∧
0
D
0
∧
∧
0
E
0
∧

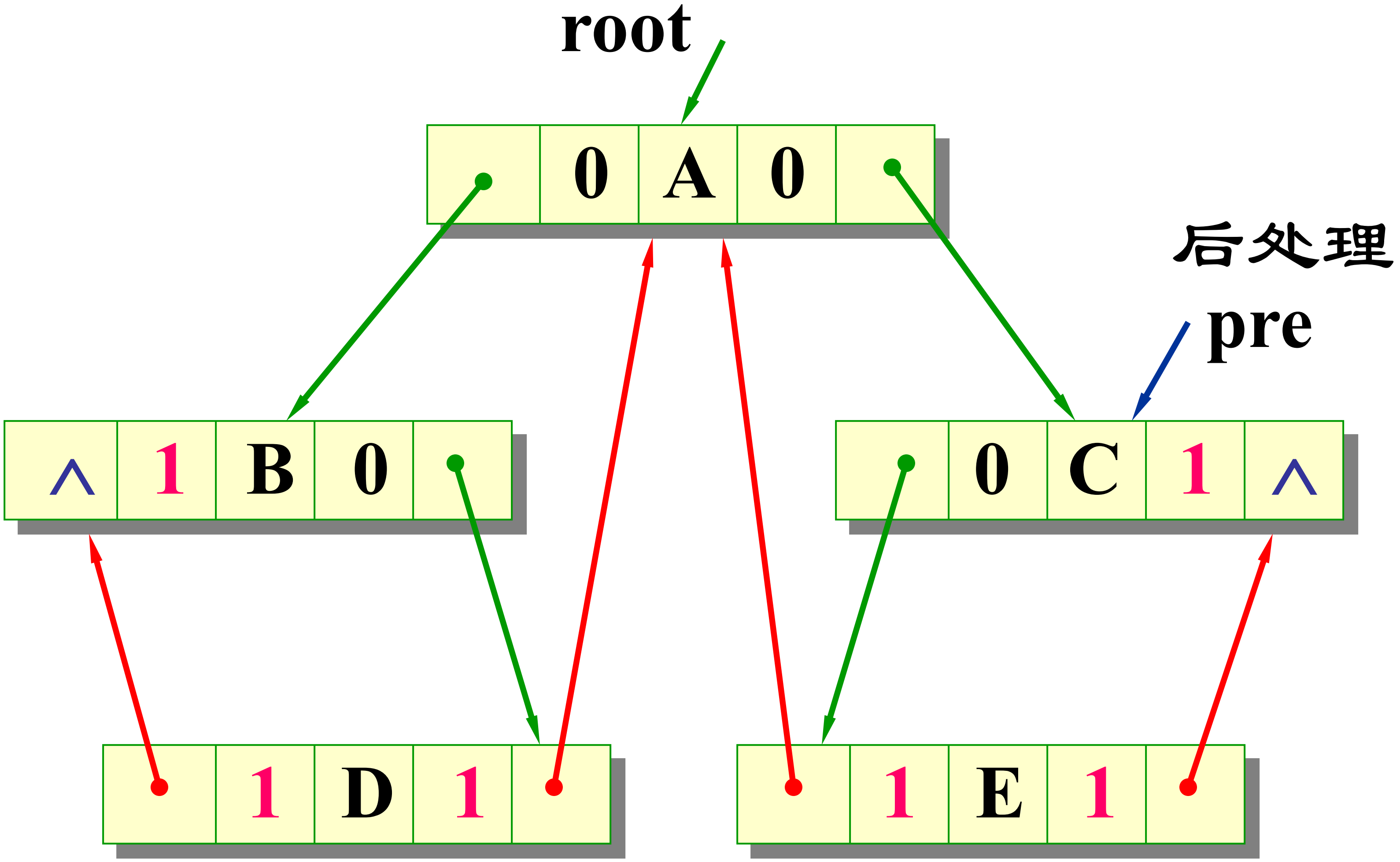

pre == NULL
root
0 A 0
current
∧ 1 B 0
0 C 0 ∧
∧ 0 D 0 ∧
∧ 0 E 0 ∧


root
0 A 0
pre
^ 1 B 0
0 C 0 ^
current
1 D 0 ^
^ 0 E 0 ^

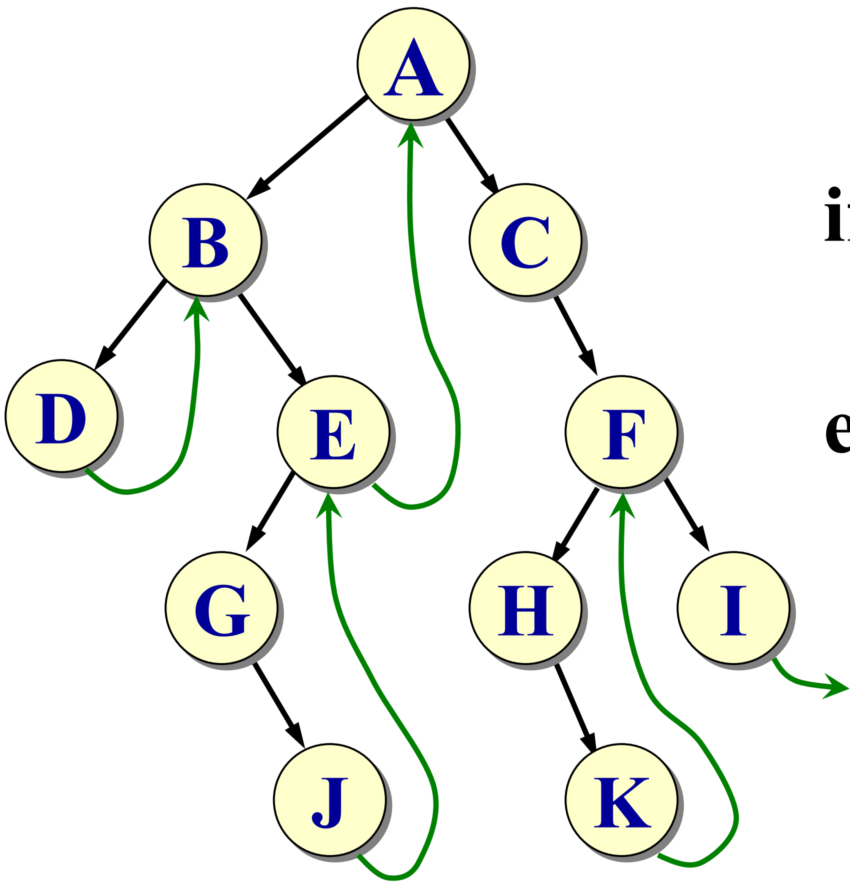

current
root
0
A
0
^
1
B
0
0
C
0
^
pre
1
D
1
^
0
E
0
^


pre
root
0 A 0
^ 1 B 0
0 C 0 ^
current
1 D 1
1 E 0 ^


root
0
A
0
current
∧
1
B
0
0
C
0
∧
pre
1
D
1
1
E
1


root
0
A
0
后处理
pre
∧
1
B
0
0
C
1
∧
1
D
1
1
E
1

寻找当前结点在中序下的后继



if (current−>rtag ==1)

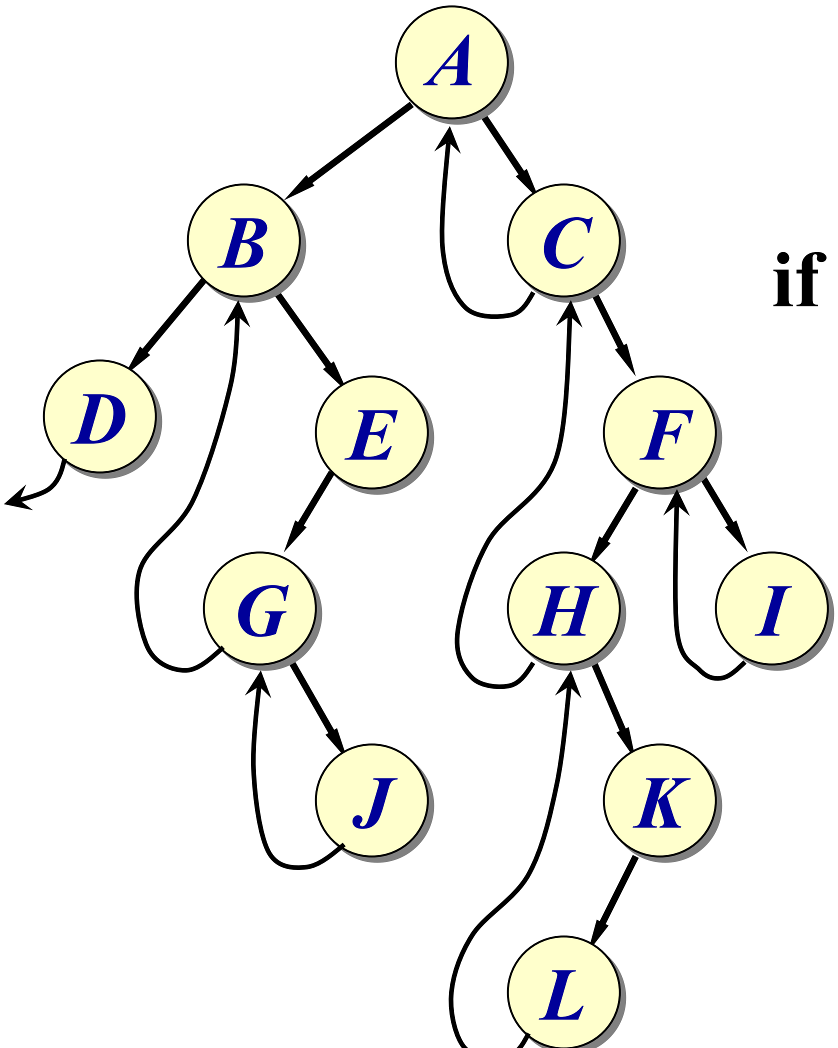
后继为current−>rightChild

else //current−>rtag == 0

后继为当前结点右子树

的中序下的第一个结点

寻找当前结点在中序下的前驱



if (current−>ltag == 1)

前驱为current−>leftChild

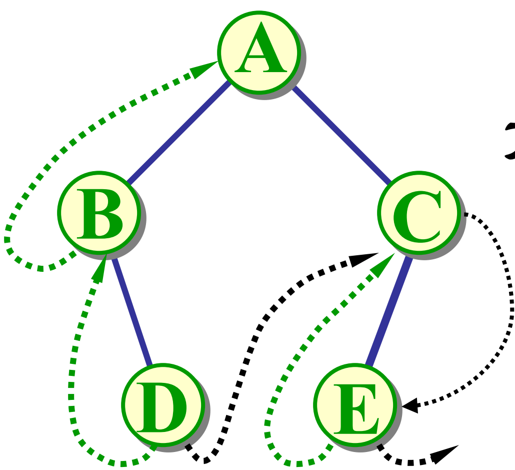
else //current−>ltag == 0

前驱为当前结点左子树

中序下的最后一个结点

前序线索化二叉树

- 在前序线索化二叉树中寻找当前结点的后继



p->ltag==1?

前驱线索 =

≠ 左子女

p->rightChild == NULL?

后继为 p->leftChild

= 无后继

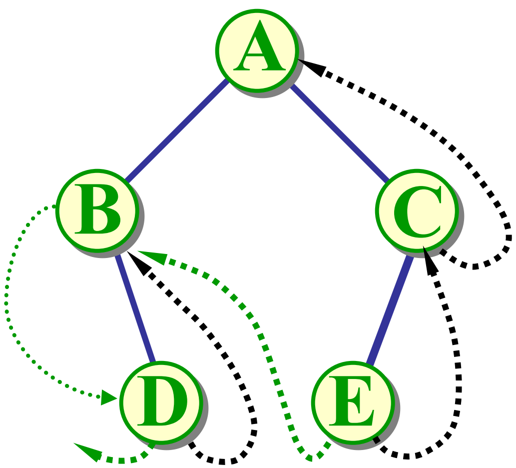
≠ 后继为 p->rightChild

<u>前序序列</u>

A B D C E

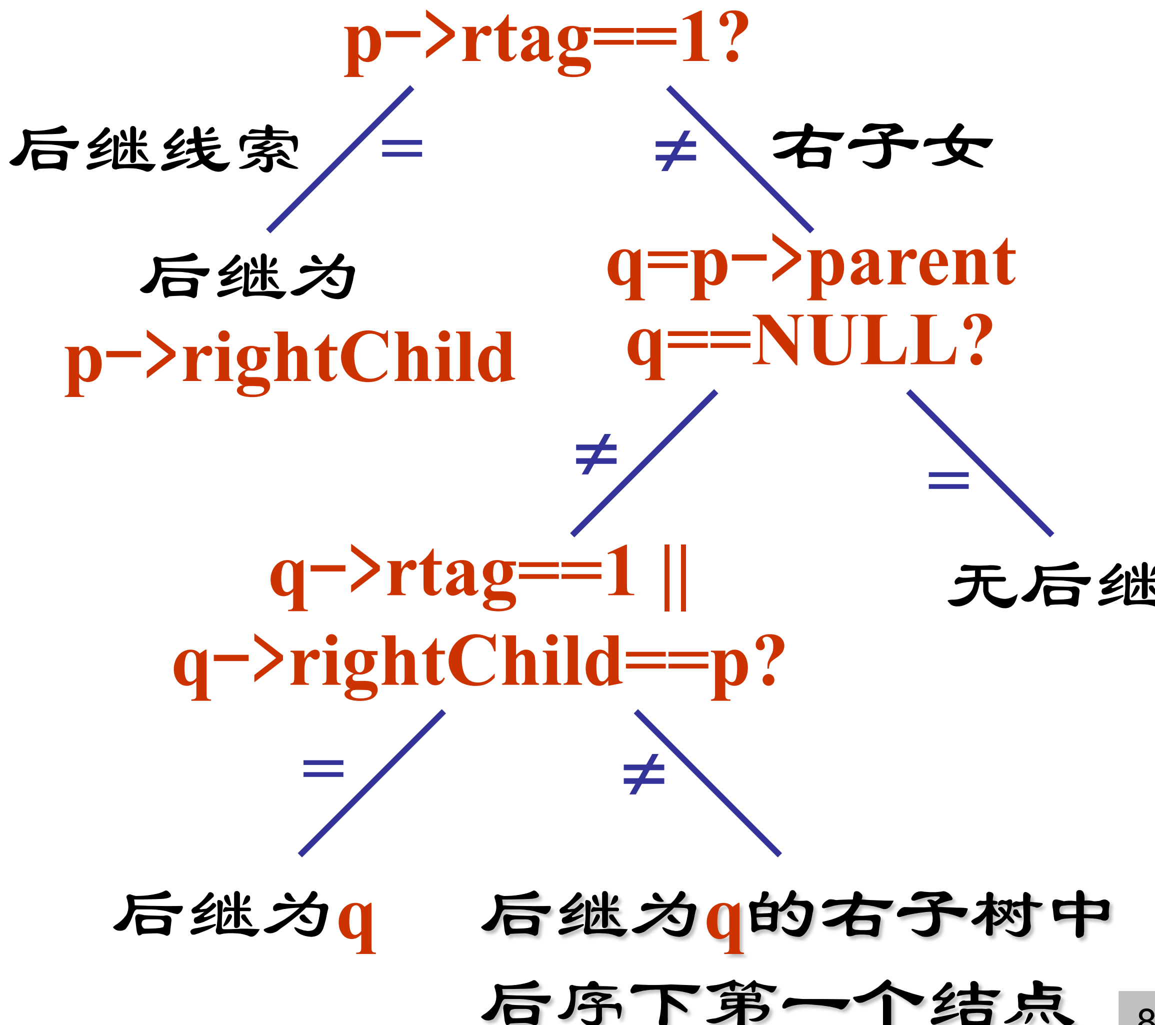
后序线索化二叉树

在后序线索化二叉树中寻找当前结点的后继



后序序列

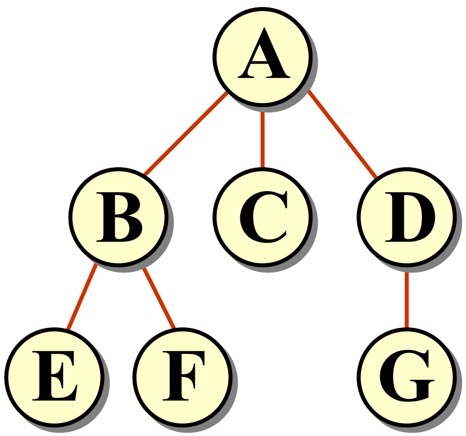
D B E C A

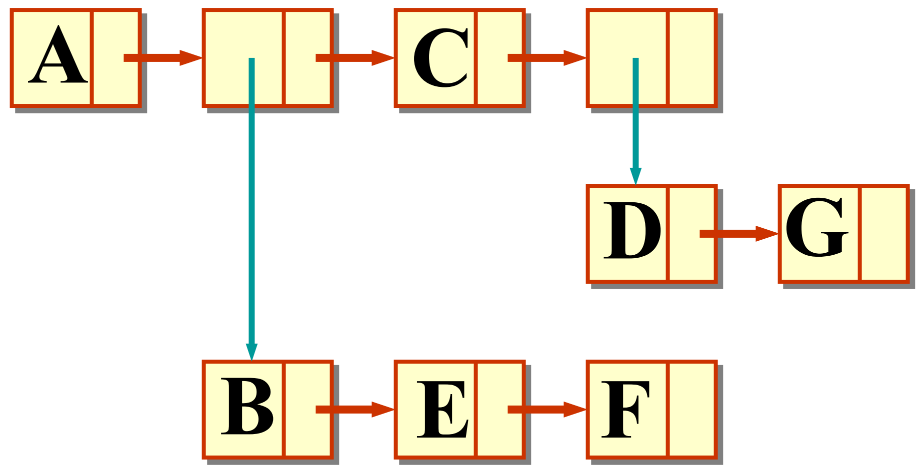


树与森林

树(一般的树)的存储表示

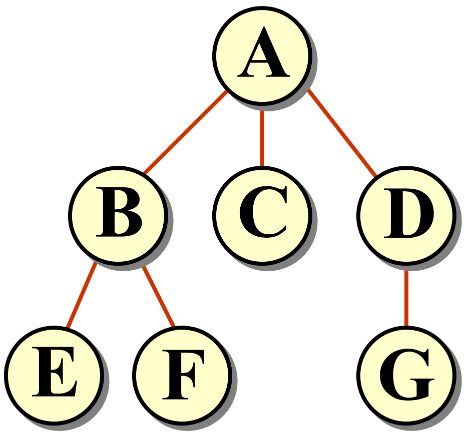
1、广义表表示





A(B(E, F), C, D(G)) 结点的utype域没有画出

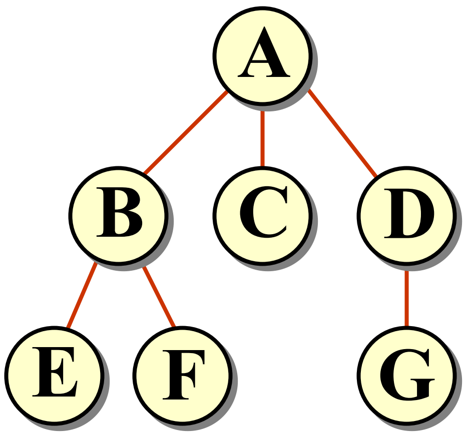
2、双亲表示

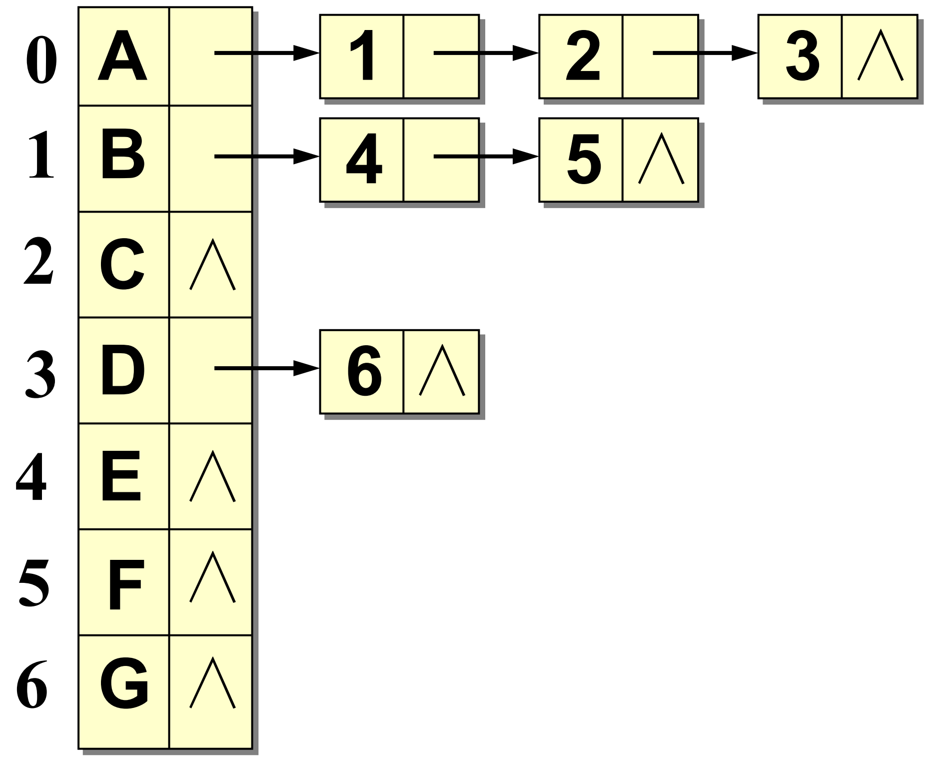


	0	1	2	3	4	5	6
data	A	B	C	D	E	F	G
parent	−1	0	0	0	1	1	3

- 树中结点的存放顺序一般不做特殊要求，但为了操作实现的方便，有时也会规定结点的存放顺序。例如，可以规定按树的前序次序存放树中的各个结点，或规定按树的层次次序安排所有结点。

3、子女链表表示

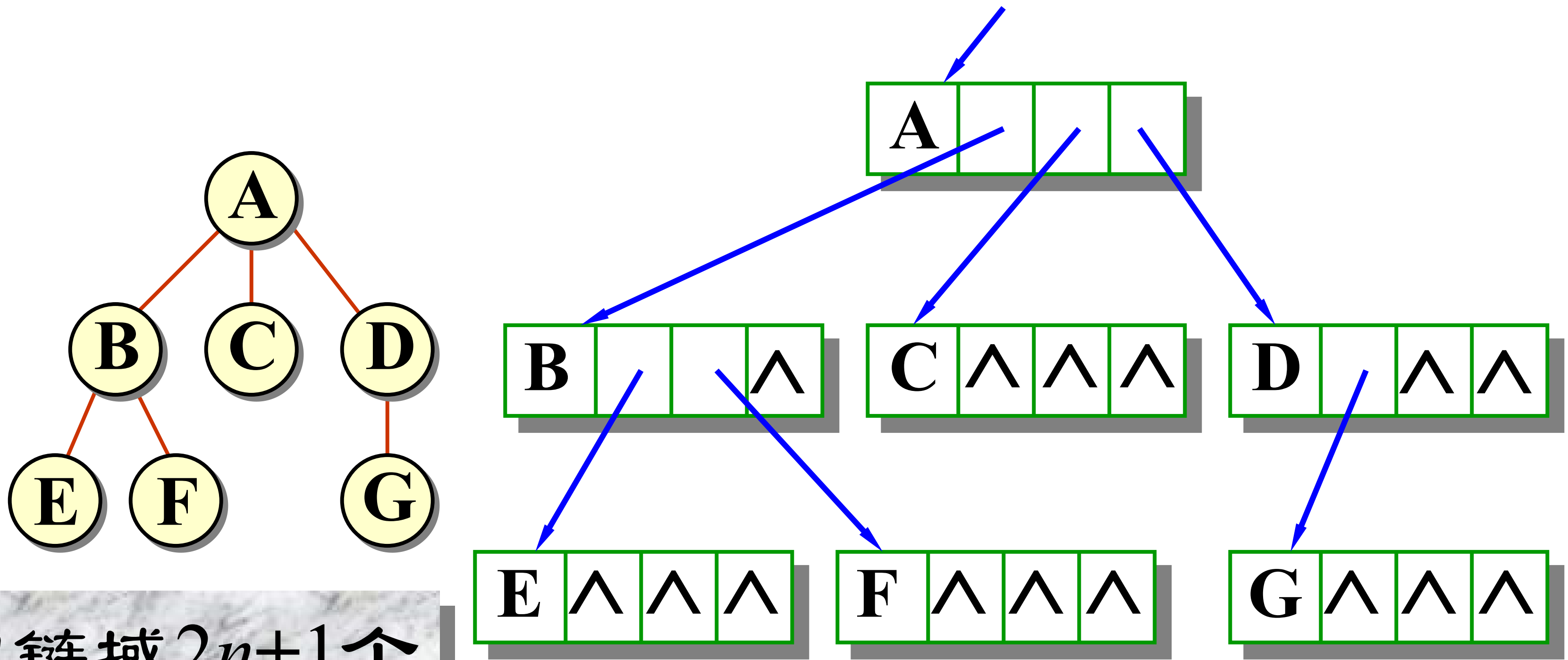




- 无序树情形链表中各结点顺序任意，有序树必须自左向右链接各个子女结点。

4、子女指针表示

- 一个合理的想法是在结点中存放指向每一个子女结点的指针。但由于各个结点的子女数不同，每个结点设置数目不等的指针，将很难管理。
- 为此，设置等长的结点，每个结点包含的指针个数相等，等于树的度（degree）。
- 这保证结点有足够的指针指向它的所有子女结点。但可能产生很多空闲指针，造成存储浪费。



空链域$2n+1$个

等数量的链域

data	$child_1$	$child_2$	$child_3$	·····	$child_d$

5、子女-兄弟表示

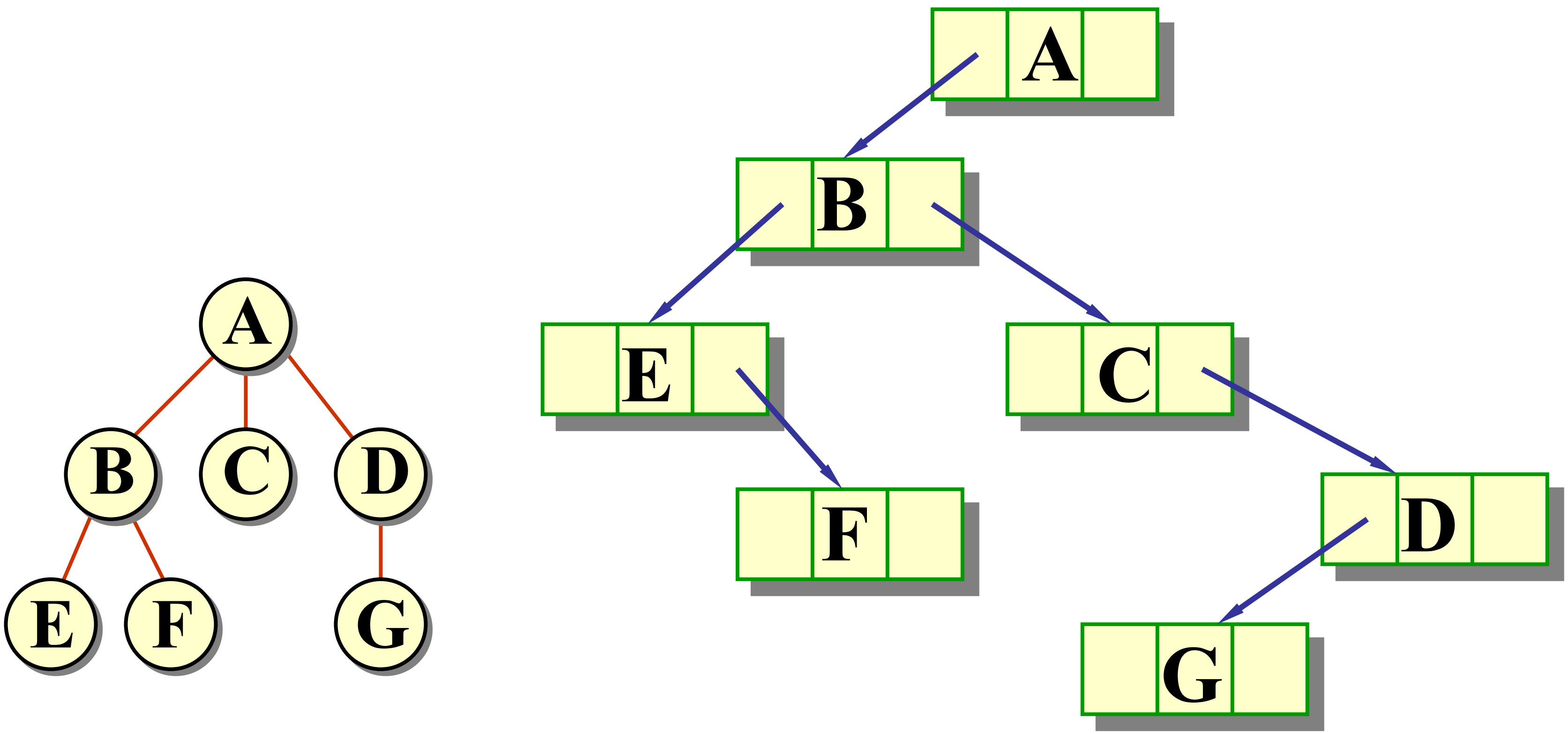
- 也称为树的二叉树表示。结点构造为:

data	firstChild	nextSibling

- **firstChild** 指向该结点的第一个子女结点。无序树时，可任意指定一个结点为第一个子女。
- **nextSibling** 指向该结点的下一个兄弟。任一结点在存储时总是有顺序的。
- 若想找某结点的所有子女，可先找**firstChild**，再反复用 **nextSibling** 沿链扫描。

树的子女-兄弟表示

data	firstChild	nextSibling



用子女-兄弟表示实现的树的类定义

```
template <class T>
struct TreeNode {                          //树的结点类
    T data;                                //结点数据
    TreeNode<T> *firstChild, *nextSibling;
                                           //子女及兄弟指针
   TreeNode (T value = 0, TreeNode<T> *fc = NULL,
      TreeNode<T> *ns = NULL)     //构造函数
      : data (value), firstChild (fc), nextSibling (ns) { }
};
```

```
template <class T>
class Tree {                    //树类
private:
    TreeNode<T> *root, *current;
        //根指针及当前指针
    int Find (TreeNode<T> *p, T value);
        //在以p为根的树中搜索value
    void RemoveSubTree (TreeNode<T> *p);
        //删除以p为根的子树
    bool FindParent (TreeNode<T> *t,
        TreeNode<T> *p);
public:
```

```
    Tree () { root = current = NULL; } //构造函数
    bool Root ();              //置根结点为当前结点
    bool IsEmpty () { return root == NULL; }
    bool FirstChild ();
      //将当前结点的第一个子女置为当前结点
    bool NextSibling ();
      //将当前结点的下一个兄弟置为当前结点
    bool Parent ();
      //将当前结点的双亲置为当前结点
    bool Find (T value);
      //搜索含value的结点, 使之成为当前结点
    ……                         //树的其他公共操作
};
```

子女-兄弟链表常用操作的实现

```
template <class T>
bool Tree<T>::Root () {
//让树的根结点成为树的当前结点
    if (root == NULL) {
        current = NULL;  return false;
    }
    else {
        current = root;  return true;
    }
};
```

```
template <class T>
bool Tree<T>::Parent () {
//置当前结点的双亲结点为当前结点
    TreeNode<T> *p = current;
    if (current == NULL || current == root)
      { current = NULL;  return false; }
        //空树或根无双亲
    return FindParent (root, p);
        //从根开始找*p的双亲结点
};
```

```
template <class T>
bool Tree<T>::
FindParent (TreeNode<T> *t, TreeNode<T> *p) {
//在根为*t的树中找*p的双亲, 并置为当前结点
    TreeNode<T> *q = t->firstChild;     //*q是*t长子
    bool succ;
    while (q != NULL && q != p) {       //扫描兄弟链
        if ((succ = FindParent (q, p)) == true)
            return succ;     //递归搜索以*q为根的子树
        q = q->nextSibling;
    }
```

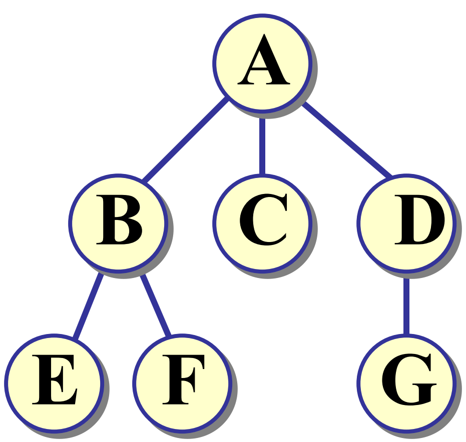
```
    if (q != NULL && q == p) {
        current = t;  return true;
    }
    else { current = NULL;  return false; }   //未找到
};

template <class T>
bool Tree<T>::FirstChild () {
//在树中找当前结点的长子, 并置为当前结点
    if (current && current->firstChild )
       { current = current->firstChild;  return true; }
    current = NULL;  return false;
};
```

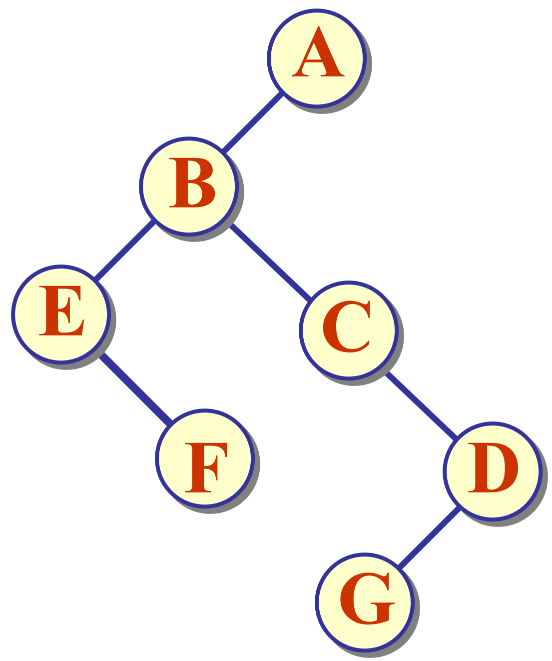
```
template <class T>
bool Tree<T>::NextSibling () {
//在树中找当前结点的兄弟, 并置为当前结点
    if (current && current->nextSibling) {
        current = current->nextSibling;
        return true;
    }
    current = NULL;  return false;
};
```

树的遍历

- 深度优先遍历
  - 先根次序遍历
  - 后根次序遍历
- 广度优先遍历

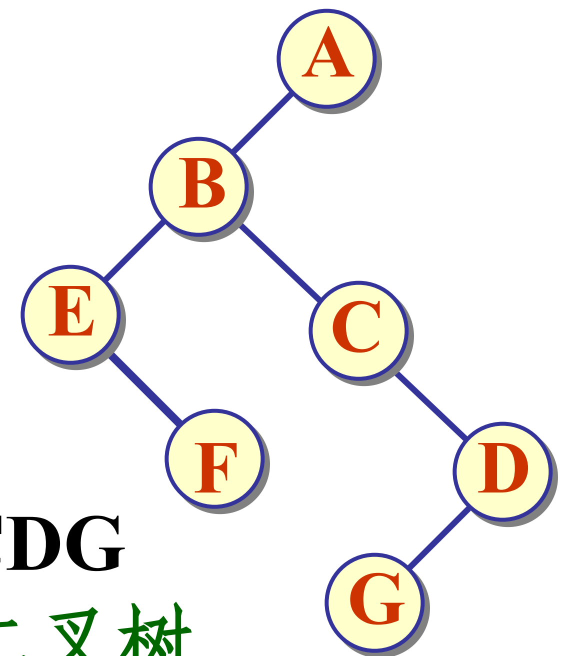


树的二叉树表示



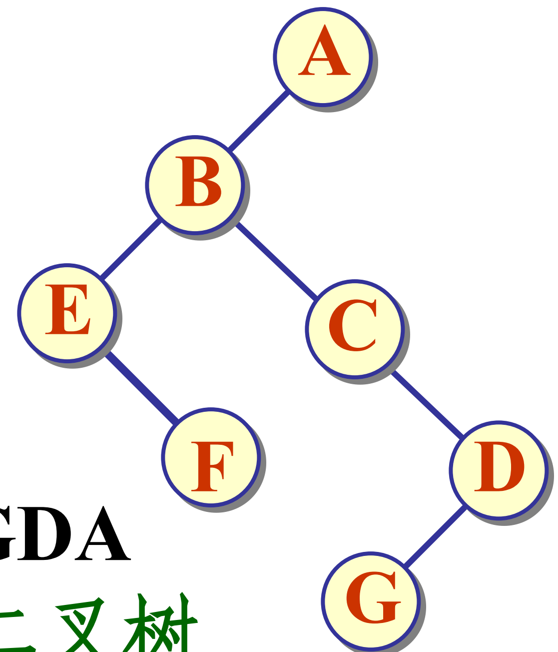
树的先根次序遍历

- 当树非空时
  - 访问根结点
  - 依次先根遍历根的各棵子树
- 树先根遍历 ABEFCDG
- 对应二叉树前序遍历 ABEFCDG
- 树的先根遍历结果与其对应二叉树表示的前序遍历结果相同
- 树的先根遍历可以借助对应二叉树的前序遍历算法实现



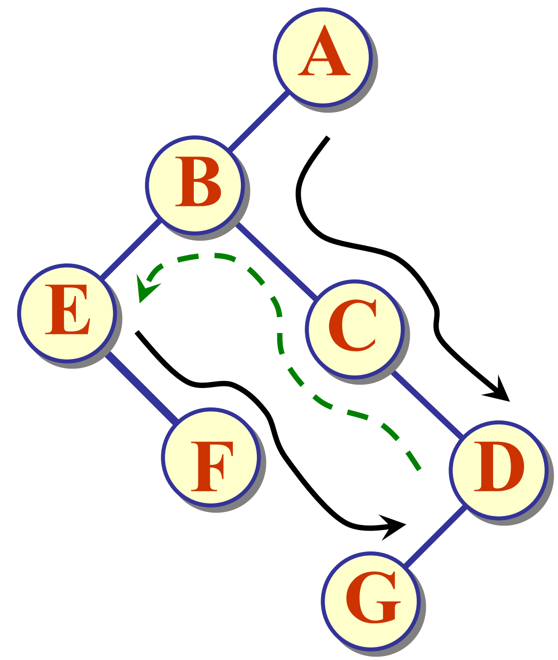
树的后根次序遍历

- 当树非空时
  - 依次后根遍历根的各棵子树
  - 访问根结点
- 树后根遍历 EFBCGDA
- 对应二叉树中序遍历 EFBCGDA
- 树的后根遍历结果与其对应二叉树表示的中序遍历结果相同
- 树的后根遍历可以借助对应二叉树的中序遍历算法实现



广度优先（层次次序）遍历

- 按广度优先次序遍历树的结果
  ABCDEFG
- 遍历算法用到一个队列。

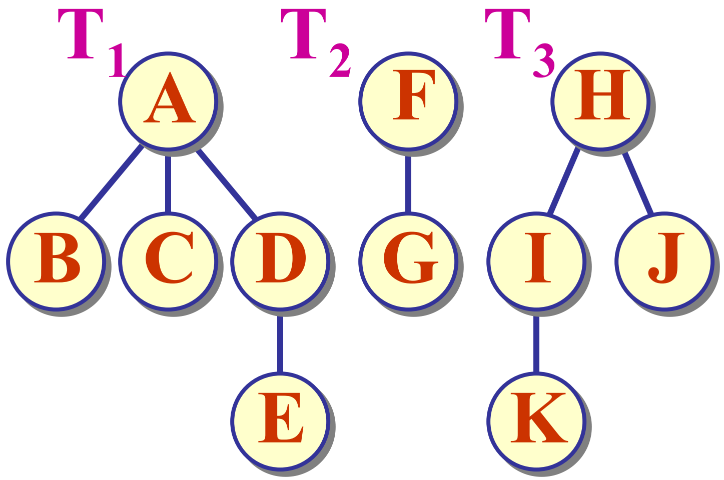


template <**class** T>
void Tree<T>::
LevelOrder(**void** (*visit) (BinTreeNode<T> *t)) {
//按广度优先次序分层遍历树，树的根结点是
//当前指针current。

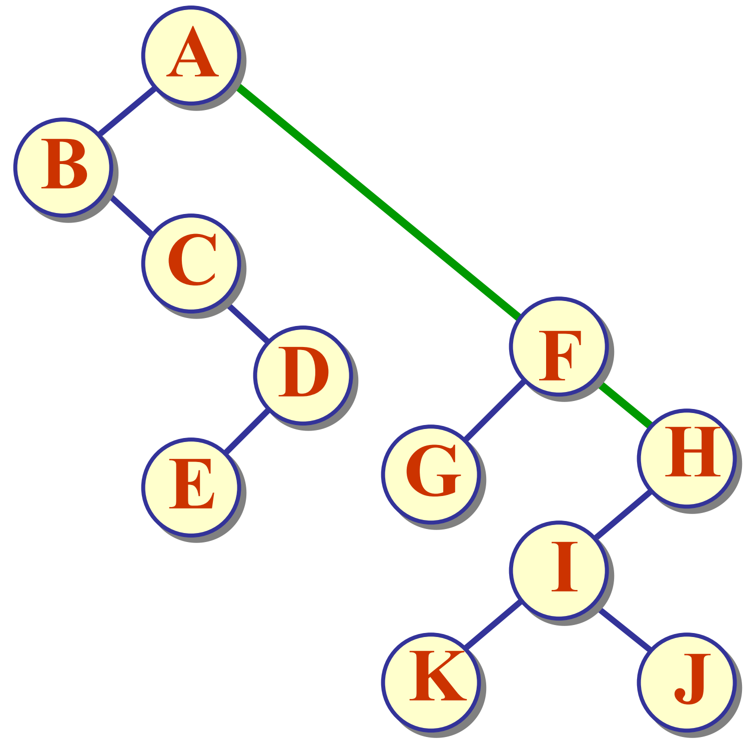
```
    Queue<TreeNode<T>*> Q;
    TreeNode<T> *p;
    if (current != NULL) {          //树不空
        p = current;                    //保存当前指针
        Q.EnQueue (current);            //根结点进队列
        while (!Q.IsEmpty ()) {
            Q.DeQueue (current);        //退出队列
            visit (current);            //访问之
            current = current->firstChild;
            while (current != NULL) {
                Q.EnQueue (current);
                current = current->nextSibling;
            }
        }
        current = p;        //恢复算法开始的当前指针
    }
};
```

森林与二叉树的转换

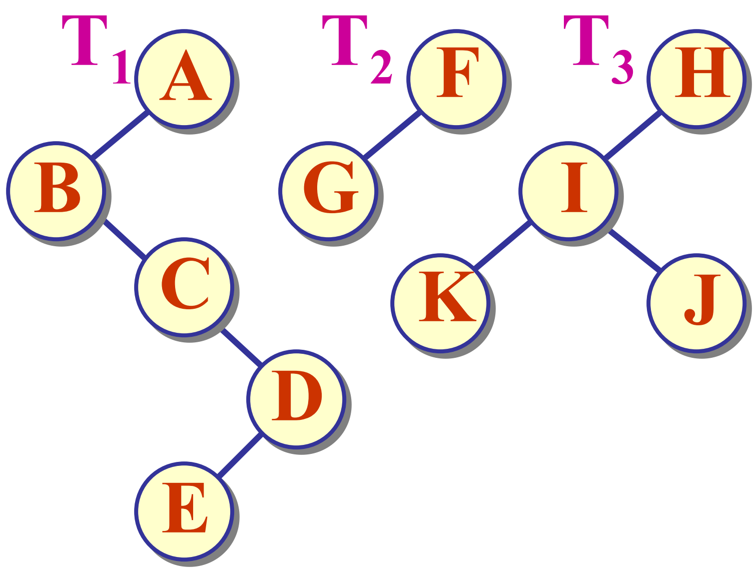
- 将一般树化为二叉树表示就是用树的子女-兄弟表示来存储树的结构。
- 森林与二叉树表示的转换可以借助树的二叉树表示来实现。



3 棵树的森林



森林的二叉树表示



各棵树的二叉树表示

森林转化成二叉树的规则

❶ 若 F 为空，即 $n = 0$，则对应的二叉树 B 为空树。

❷ 若 F 不空，则

✓ 二叉树 B 的根是 F 第一棵树 T_1 的根;

✓ 其左子树为$B\ (T_{11}, T_{12}, \ldots, T_{1m})$，其中，$T_{11}, T_{12}, \ldots, T_{1m}$ 是 T_1 的根的子树;

✓ 其右子树为 $B\ (T_2, T_3, \ldots, T_n)$，其中，$T_2, T_3, \ldots, T_n$ 是除 T_1 外其它树构成的森林。

二叉树转换为森林的规则

❶ 如果 B 为空，则对应的森林 F 也为空。

❷ 如果 B 非空，则

- ✓ F 中第一棵树 T_1 的根为 B 的根;
- ✓ T_1 的根的子树森林 $\{ T_{11}, T_{12}, \ldots, T_{1m} \}$ 是由 B 的根的左子树 LB 转换而来;
- ✓ F 中除了 T_1 之外其余的树组成的森林 $\{ T_2, T_3, \ldots, T_n \}$ 是由 B 的根的右子树 RB 转换而成的森林。

森林的遍历

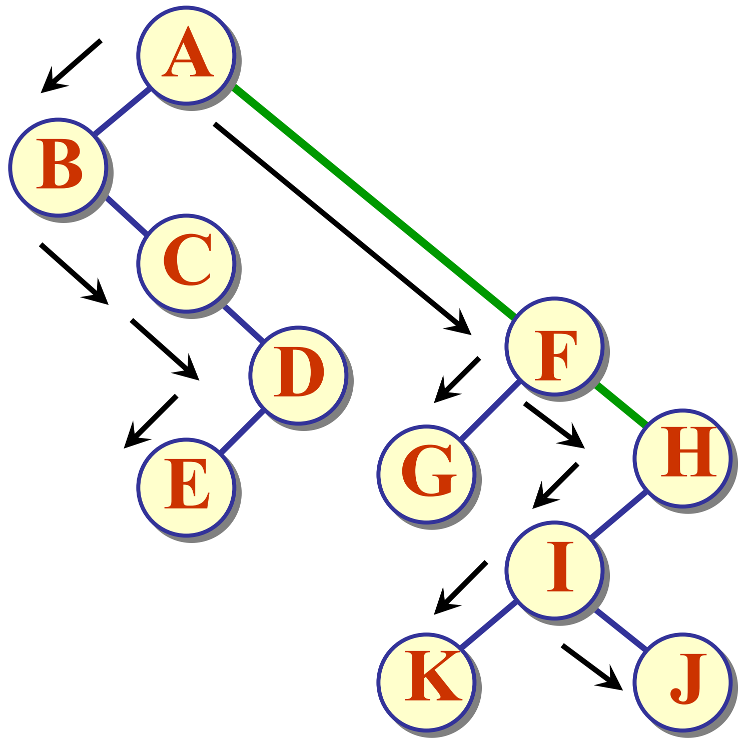
- 森林的遍历也分为深度优先遍历和广度优先遍历，深度优先遍历又可分为先根次序遍历和后根次序遍历。

深度优先遍历

- 给定森林 F，若 $F = \emptyset$，则遍历结束。否则
- 若$F = \{\{T_1 = \{ r_1, T_{11}, \ldots, T_{1k} \}, T_2, \ldots, T_m\}$，则可以导出先根遍历、后根遍历两种方法。其中，$r_1$是第一棵树的根结点，$\{T_{11}, \ldots, T_{1k}\}$是第一棵树的子树森林，$\{T_2, \ldots,T_m\}$是除去第一棵树之后剩余的树构成的森林。

森林的先根次序遍历

- 若森林F = Ø，返回；否则
  - ✓访问森林的根（也是第一棵树的根）r_1;
  - ✓先根遍历森林第一棵树的根的子树森林$\{T_{11}, \ldots, T_{1k}\}$;
  - ✓先根遍历森林中除第一棵树外其他树组成的森林$\{T_2, \ldots, T_m\}$。



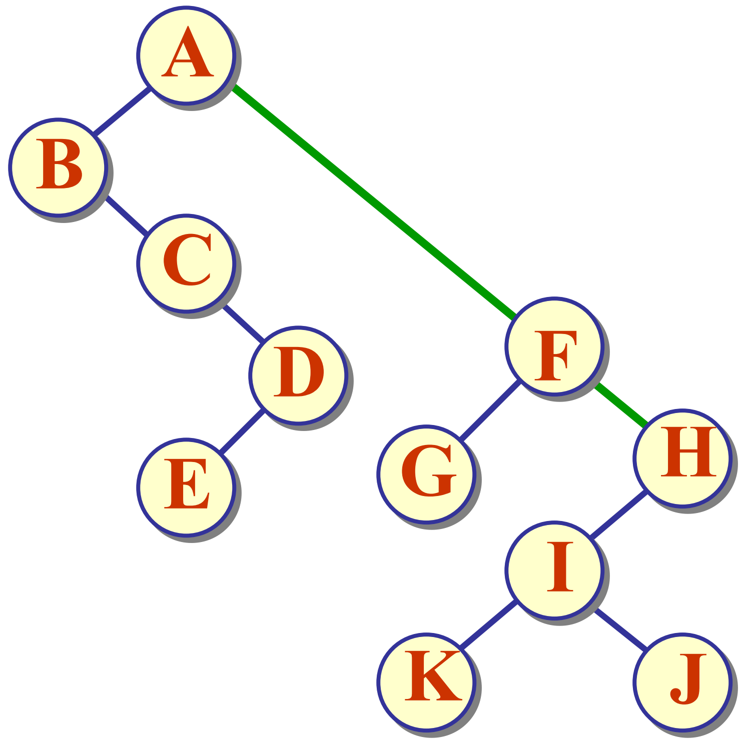
- 森林的先根次序遍历的结果序列

  ABCDE FG HIKJ

- 这相当于对应二叉树的前序遍历结果。

森林的后根次序遍历

- 若森林 $F = \varnothing$，返回；否则
  - ✓ 后根遍历森林 F 第一棵树的根结点的子树森林$\{T_{11}, \ldots, T_{1k}\}$;
  - ✓ 访问森林的根结点 r_1;
  - ✓ 后根遍历森林中除第一棵树外其他树组成的森林$\{T_2, \ldots, T_m\}$。

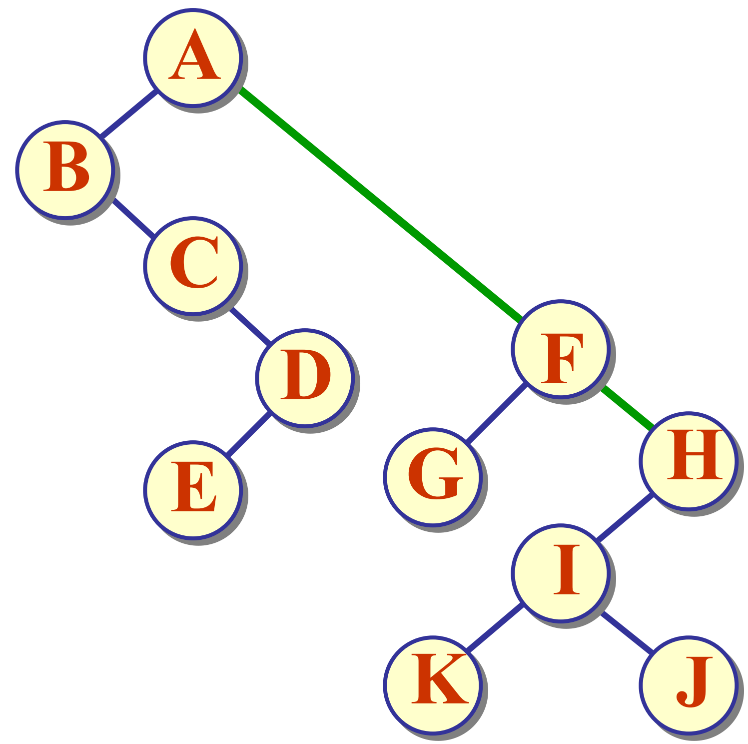


- 森林的后根次序遍历的结果序列

  BCEDA GF KIJH
- 这相当于对应二叉树中序遍历的结果。

广度优先遍历（层次序遍历）

- 若森林 *F* 为空，返回；否则
  - ✓ 依次遍历各棵树的根结点；
  - ✓ 依次遍历各棵树根结点的所有子女；
  - ✓ 依次遍历这些子女结点的子女结点；
  - ✓ ……



AFH BCDGIJ EK

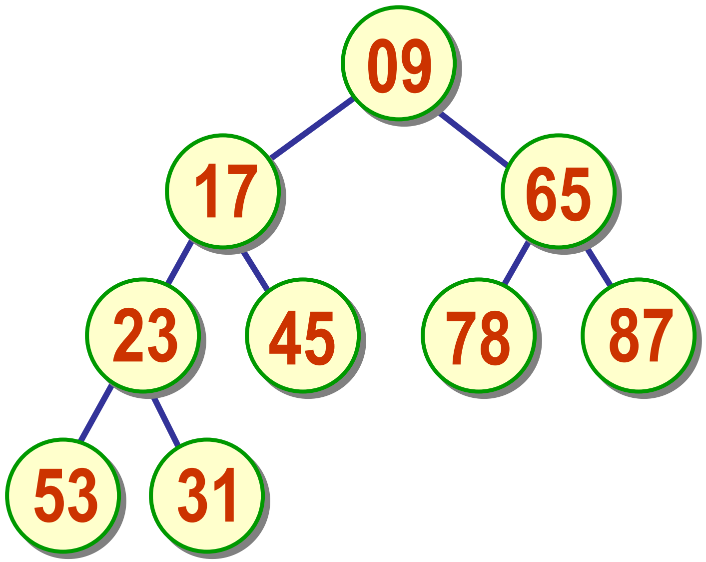
堆（Heap）

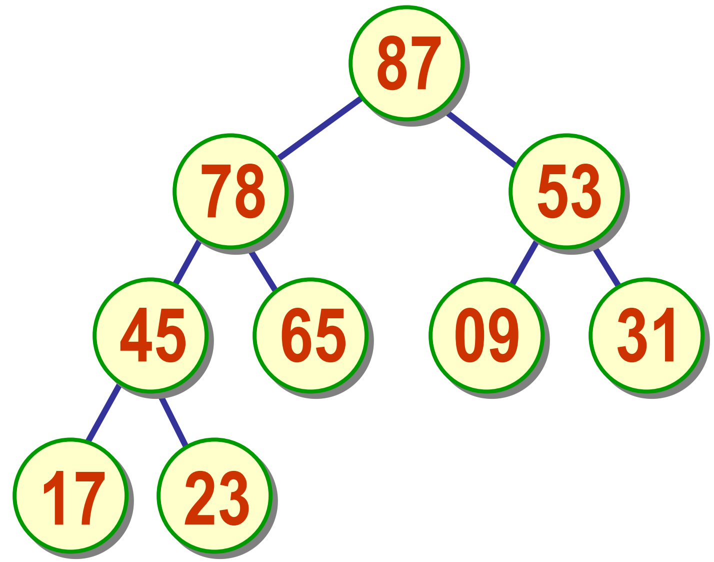
优先级队列

- 每次出队列的是优先权最高的元素。
- 用堆实现其存储表示，能够高效运作。

```
template <class E>
class MinPQ {          //最小优先级队列类的定义
public:
    Virtual bool Insert (E& x) = 0;
    Virtual bool Remove (E& x) = 0;
};
```

堆的定义





完全二叉树顺序表示

$K_i \leqslant K_{2i+1}$ &&

$K_i \leq K_{2i+2}$

完全二叉树顺序表示

$K_i \geqslant K_{2i+1}$ &&

$K_i \geq K_{2i+2}$

堆的元素下标计算

- 由于堆存储在下标从 0 开始计数的一维数组中，因此在堆中给定下标为 i 的结点时

a) 如果 $i = 0$，结点 i 是根结点，无双亲；否则结点 i 的父结点为结点 $\lfloor(i-1)/2\rfloor$;

b) 如果 $2i+1 > n-1$，则结点 i 无左子女；否则结点 i 的左子女为结点 $2i+1$;

c) 如果 $2i+2 > n-1$，则结点 i 无右子女；否则结点 i 的右子女为结点 $2i+2$。

最小堆的类定义

```
template <class E>
class MinHeap : public MinPQ<E> {
//最小堆继承了（最小）优先级队列
public:
    MinHeap (int sz = DefaultSize);      //构造函数
    MinHeap (E arr[], int n);            //构造函数
    ～MinHeap() { delete [ ] heap; }     //析构函数
    bool Insert (E& x);                  //插入
    bool Remove (E& x);                  //删除
```

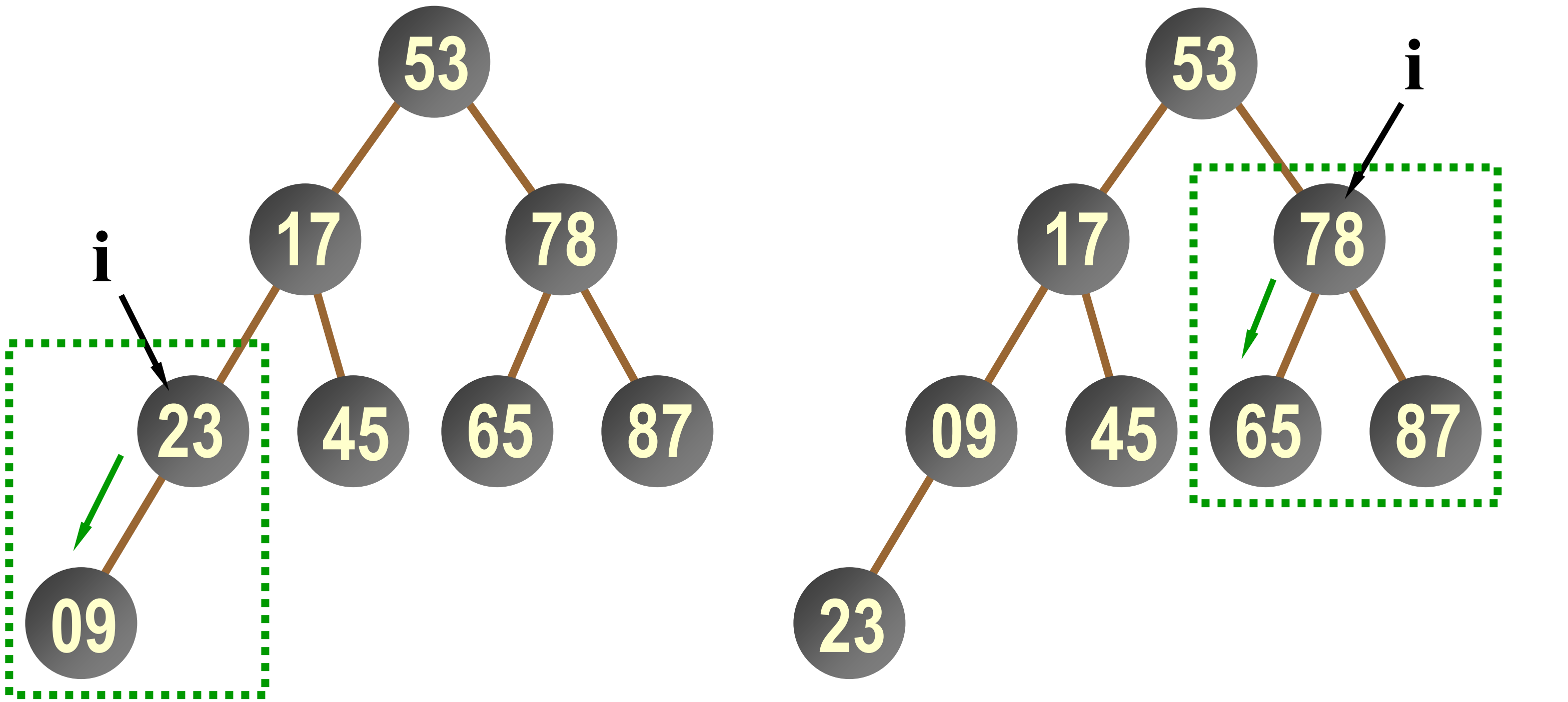
```
    bool IsEmpty () const           //判堆空否
      { return  currentSize == 0; }
    bool IsFull () const            //判堆满否
      { return currentSize == maxHeapSize; }
    void MakeEmpty () { currentSize = 0; }   //置空堆
private:
    E *heap;                    //最小堆元素存储数组
    int currentSize;            //最小堆当前元素个数
    int maxHeapSize;            //最小堆最大容量
    void siftDown (int start, int m);     //调整算法
    void siftUp (int start);              //调整算法
};
```

堆的建立

```
template <class E>
MinHeap<E>::MinHeap (int sz) {
    maxHeapSize = (DefaultSize < sz) ?
                                    sz : DefaultSize;
    heap = new E[maxHeapSize];        //创建堆空间
    if (heap == NULL) {
        cerr << "堆存储分配失败！" << endl;  exit(1);
    }
    currentSize = 0;                  //建立当前大小
};
```

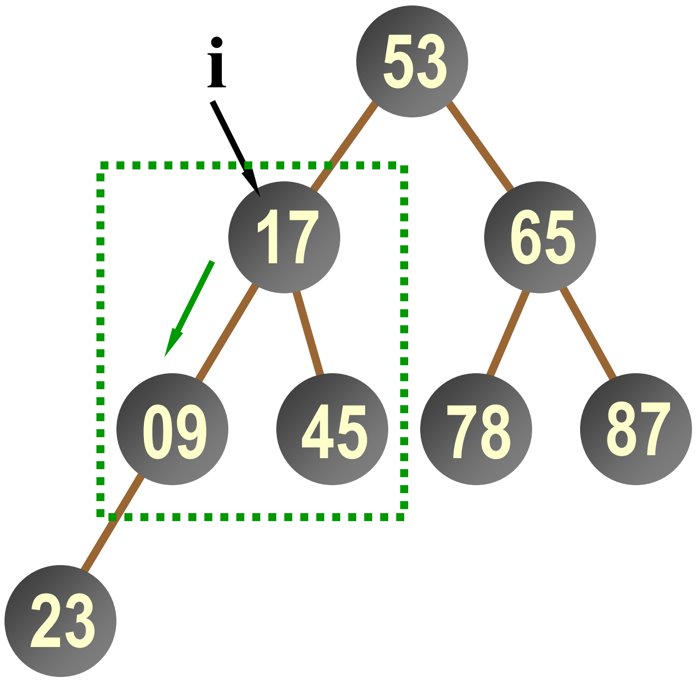
```
template <class E>
MinHeap<E>::MinHeap (E arr[], int n) {
    maxHeapSize = (DefaultSize < n) ? n : DefaultSize;
    heap = new E[maxHeapSize];
    if (heap == NULL) {
      cerr << "堆存储分配失败！" << endl;  exit(1);
    }
    for (int i = 0; i < n; i++) heap[i] = arr[i];
    currentSize = n;            //复制堆数组, 建立当前大小
    int currentPos = (currentSize-2)/2;
                    //找最初调整位置:最后分支结点
    while (currentPos >= 0) {      //逐步向上扩大堆
        siftDown (currentPos, currentSize-1);
        //局部自上向下下滑调整
        currentPos--;
    }
};
```

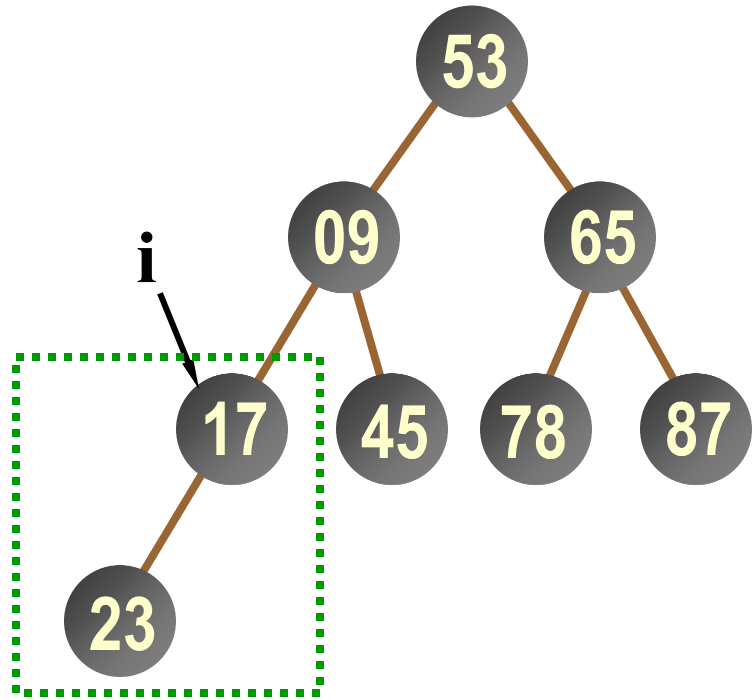
将一组用数组存放的任意数据调整成堆



currentPos = i = 3　　　　currentPos = i = 2

自下向上逐步调整为最小堆

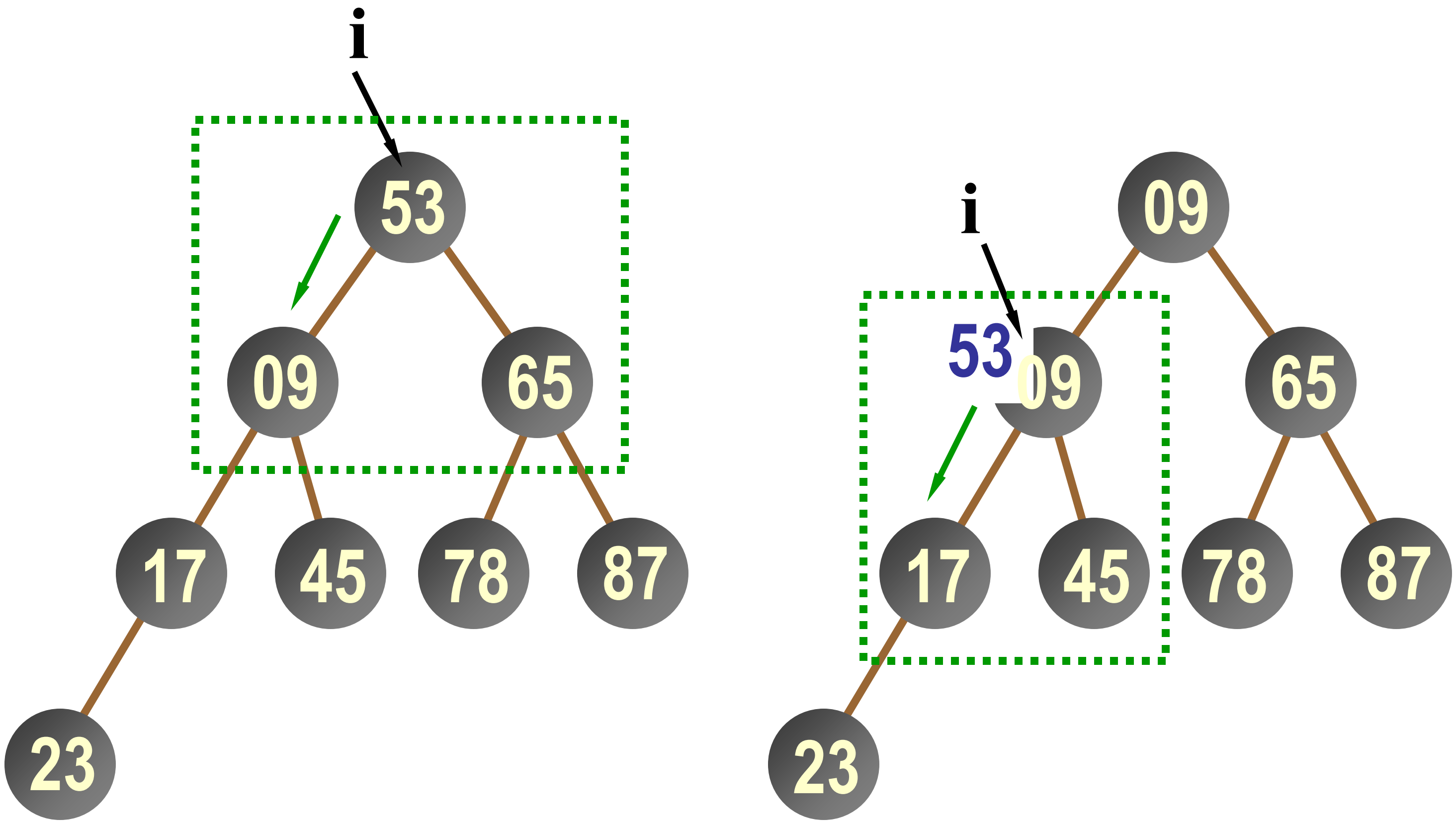




currentPos = i = 1

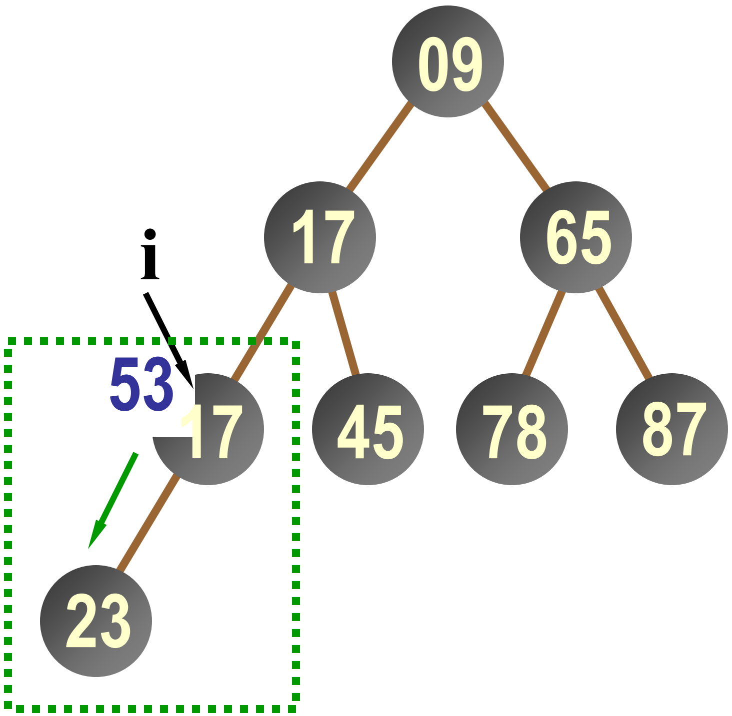
Step 1

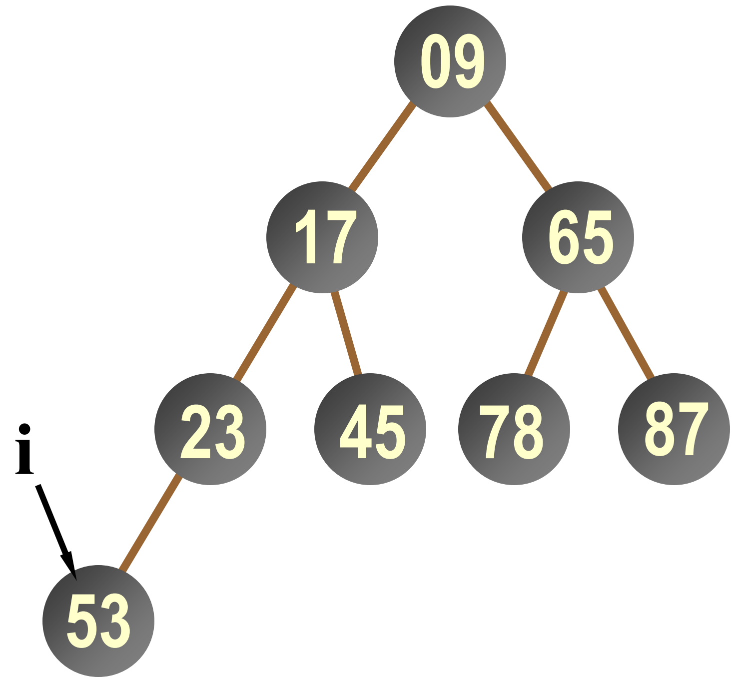
Step 2



currentPos = i = 0

Step 1 **Step 2**





currentPos = i = 0

Step 3

Step 4

最小堆的下滑调整算法

```
template <class E>
void MinHeap<E>::siftDown (int start, int m ) {
//私有函数: 从结点start开始到m为止, 自上向下比较,
//如果子女的值小于父结点的值, 则关键码小的上浮,
//继续向下层比较, 将一个集合局部调整为最小堆。
    int i = start,   j = 2*i+1;          //j是i的左子女位置
    E temp = heap[i];
    while (j <= m) {                     //检查是否到最后位置
        if ( j < m && heap[j] > heap[j+1] ) j++;
                                         //让j指向两子女中的小者
```

```
        if ( temp <= heap[j] ) break;     //小则不做调整
        else { heap[i] = heap[j];  i = j; j = 2*j+1; }
                                      //否则小者上移, i, j下降
    }
    heap[i] = temp;               //回放temp中暂存的元素
};
```

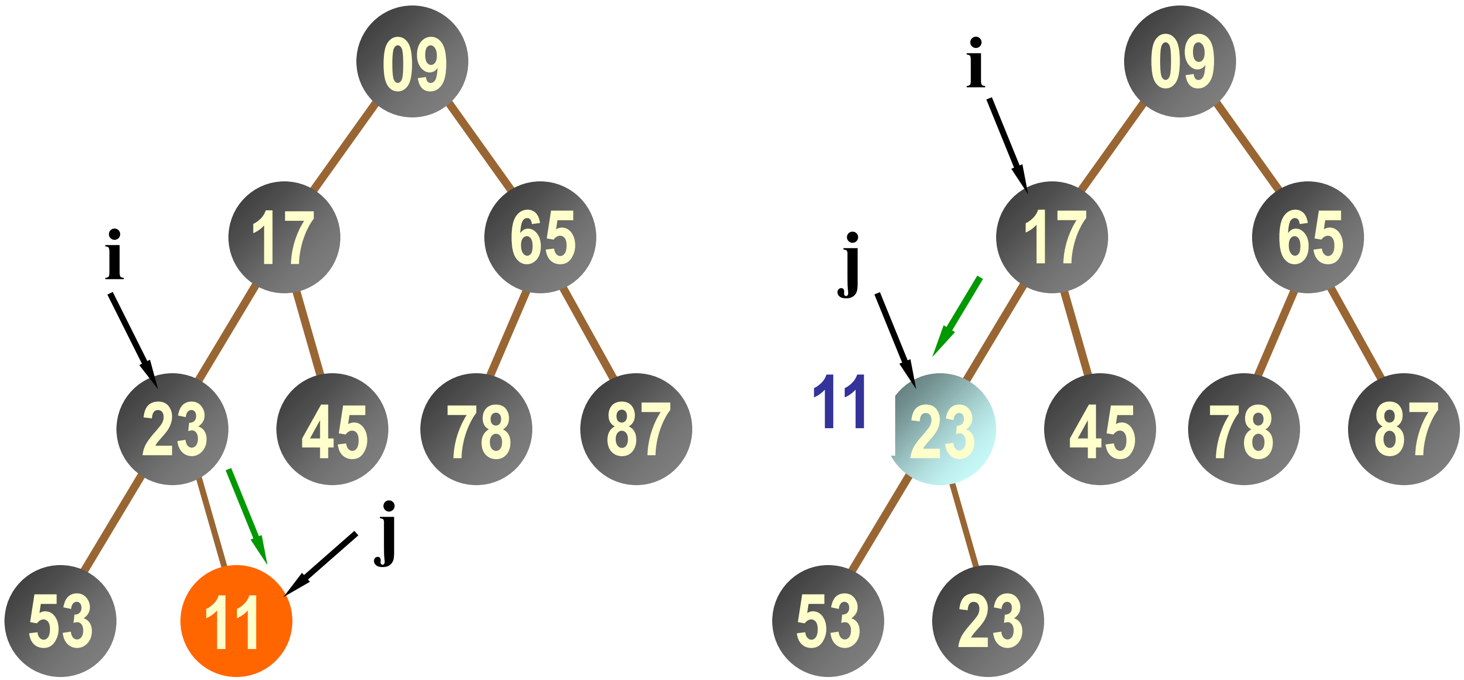
最小堆的插入

- 每次插入都加在堆的最后，再自下向上执行调整，使之重新形成堆，时间复杂性$O(\log_2 n)$。

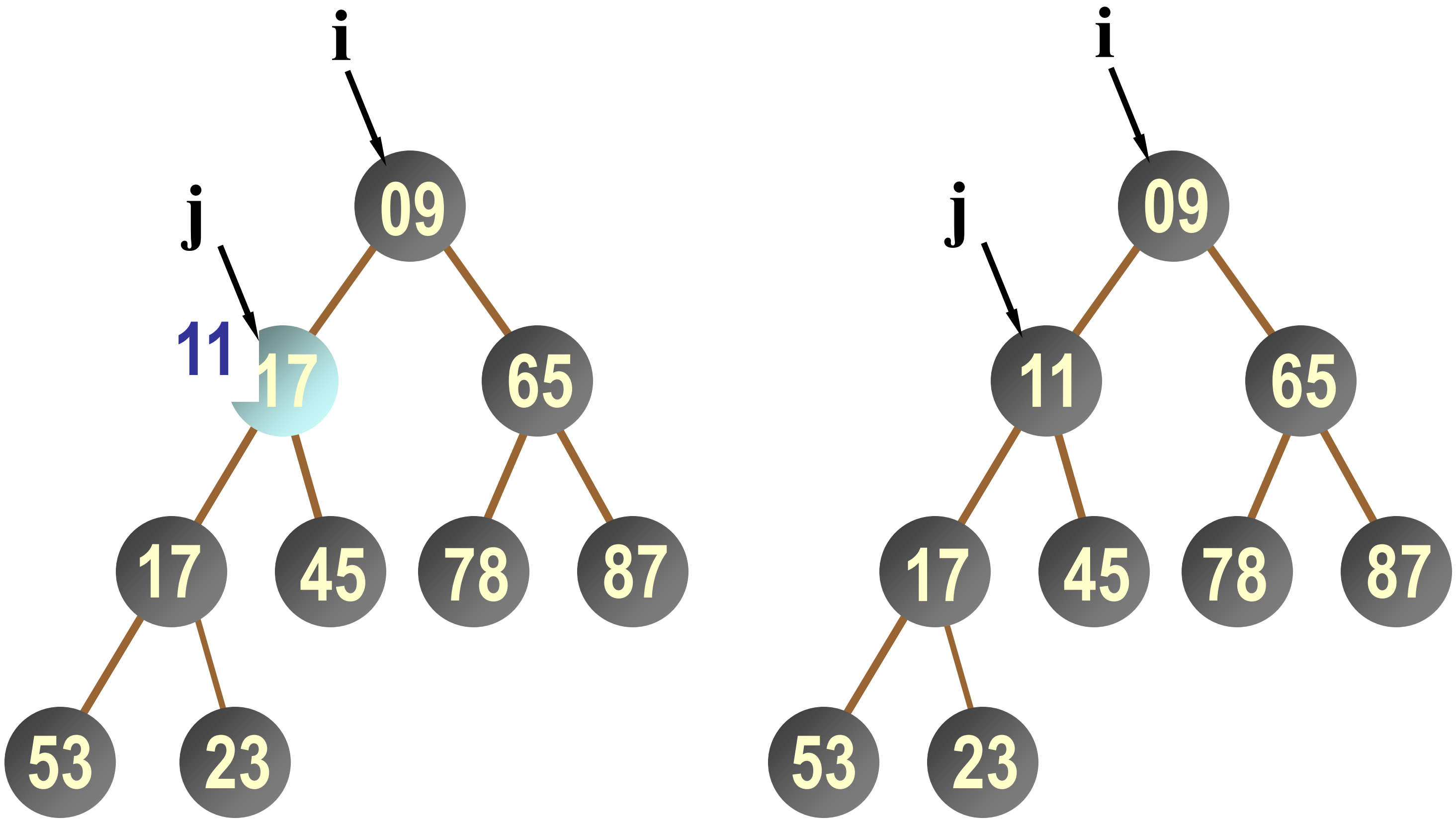
```
template <class E>
bool MinHeap<E>::Insert (const E& x ) {
//公共函数: 将x插入到最小堆中
   if ( currentSize == maxHeapSize )  //堆满
      { cerr << "Heap Full" << endl;  return false; }
   heap[currentSize] = x;                  //插入
   siftUp (currentSize);                   //向上调整
   currentSize++;                          //堆计数加1
   return true;
};
```

```
template <class E>
void MinHeap<E>::siftUp (int start) {
//私有函数: 从结点start开始到结点0为止, 自下向上
//比较, 如果子女的值小于父结点的值, 则相互交换,
//这样将集合重新调整为最小堆。关键码比较符<=
//在E中定义。
    int j = start,  i = (j-1)/2;   E temp = heap[j];
    while (j > 0) {              //沿父结点路径向上直达根
        if (heap[i] <= temp) break;
                                 //父结点值小, 不调整
        else { heap[j] = heap[i];  j = i;  i = (i-1)/2; }
                                 //父结点结点值大, 调整
    }
    heap[j] = temp;                          //回送
};
```

最小堆的向上调整



在堆中插入新元素11


i
j
09
11
17
65
17
45
78
87
53
23
i
j
09
11
65
17
45
78
87
53
23

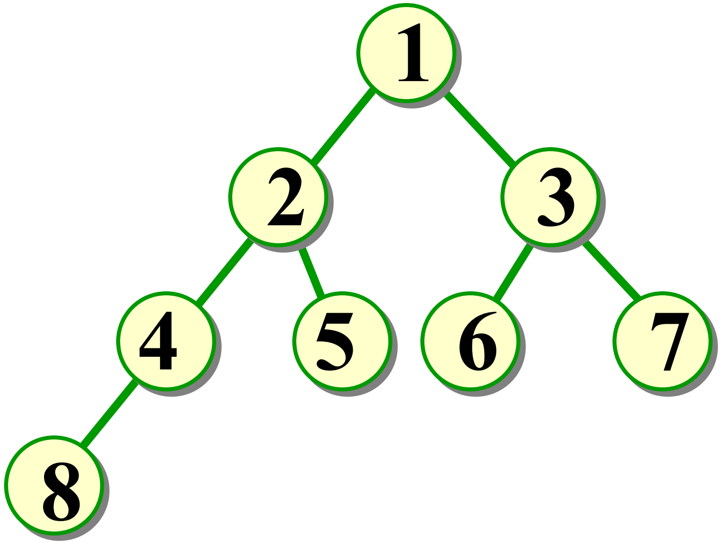
最小堆的删除算法

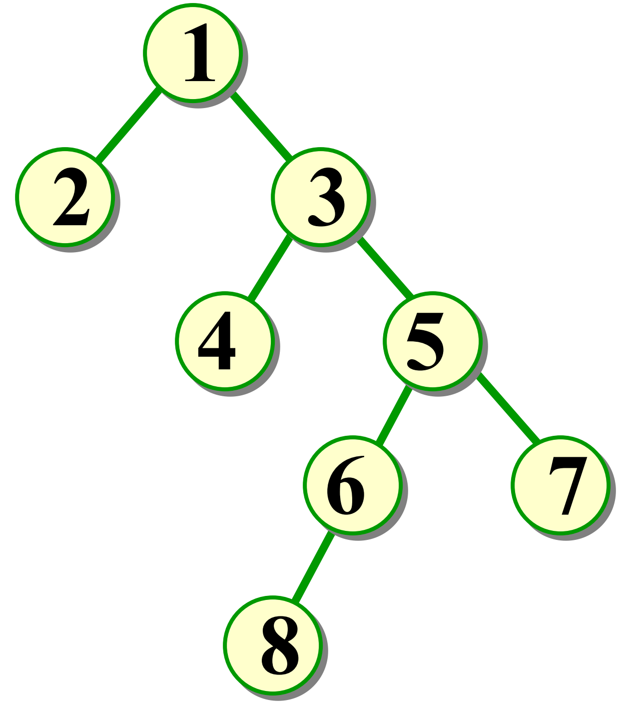
```
template <class E>
bool MinHeap<E>::Remove (E& x) {
    if ( !currentSize ) {                  //堆空, 返回false
        cout << "Heap empty" << endl;  return false;
    }
    x = heap[0];
    heap[0] = heap[currentSize-1];
    currentSize--;
    siftDown(0, currentSize-1);        //自上向下调整为堆
    return true;                       //返回最小元素
};
```

Huffman树

路径长度 (Path Length)

- 两个结点之间的路径长度 **PL** 是连接两结点的路径上的分支数。
- 树的外部路径长度**EPL**是各叶结点（外结点）到根结点的路径长度之和。
- 树的内部路径长度**IPL**是各非叶结点（内结点）到根结点的路径长度之和。
- 树的路径长度 **PL = EPL + IPL**。





IPL = 0+1+1+2 = 4

EPL = 2+2+2+3 = 9

PL = 13

IPL = 0+1+2+3 = 6

EPL = 1+2+3+4 = 10

PL = 16

- n 个结点的二叉树的路径长度不小于下述数列前 n 项的和，即

$$PL = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

$$= 0 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + \cdots$$

- 其路径长度最小者为

$$PL = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

- 完全二叉树满足这个要求。

带权路径长度
(Weighted Path Length, WPL)

- 在很多应用问题中为树的叶结点赋予一个权值，用于表示出现频度、概率值等。因此，在问题处理中把叶结点定义得不同于非叶结点，把叶结点看成“外结点”，非叶结点看成“内结点”。这样的二叉树称为相应权值的扩充二叉树。
- 扩充二叉树中只有度为 2 的内结点和度为 0 的外结点。根据二叉树的性质，有 n 个外结点就有 $n-1$ 个内结点，总结点数为$2n-1$。

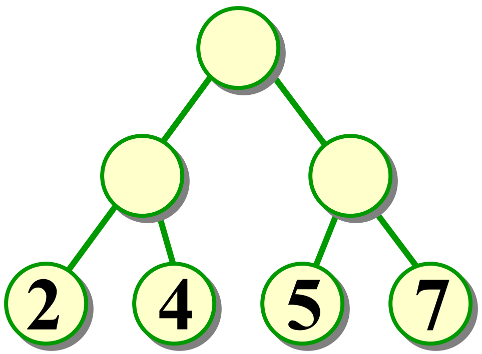
- 若一棵扩充二叉树有 n 个外结点，第 i 个外结点的权值为w_i，它到根的路径长度为l_i，则该外结点到根的带权路径长度为w_i*l_i。

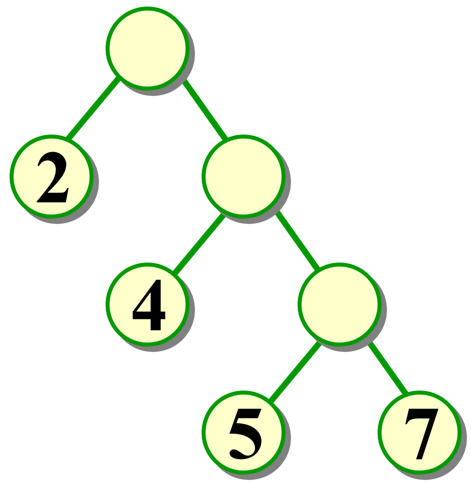
- 扩充二叉树的带权路径长度定义为树的各外结点到根的带权路径长度之和。

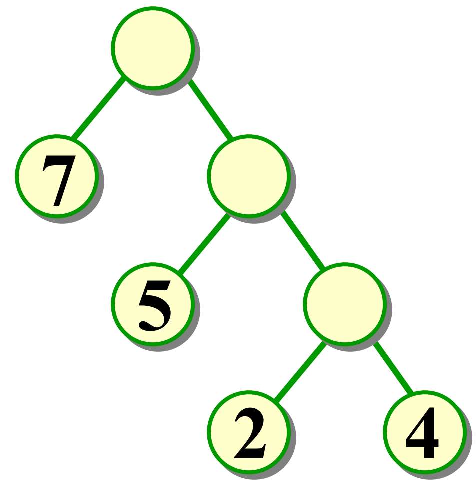
$$WPL = \sum_{i=1}^{n} w_i * l_i$$

- 对于同样一组权值，如果放在外结点上，组织方式不同，带权路径长度也不同。

具有不同带权路径长度的扩充二叉树







WPL = 2*2+ 4*2+5*2+ 7*2 = 36

WPL = 2*1+ 4*2+5*3+ 7*3 = 46

WPL = 7*1+ 5*2+2*3+ 4*3 = 35

带权路径长度达到最小

Huffman树

- 带权路径长度达到最小的扩充二叉树即为Huffman树。
- 在Huffman树中，权值大的结点离根最近。

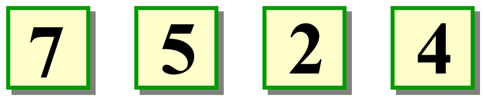
Huffman树的构造算法

1. 给定 n 个权值 $\{w_0, w_1, w_2, \ldots, w_{n-1}\}$，构造具有 n 棵扩充二叉树的森林 $F = \{ T_0, T_1, T_2, \ldots, T_{n-1} \}$，其中每棵扩充二叉树 T_i 只有一个带权值 w_i 的根结点，其左、右子树均为空。

2. 重复以下步骤，直到 F 中仅剩一棵树为止:
   a) 在 F 中选取两棵根结点的权值最小的扩充二叉树，做为左、右子树构造一棵新的二叉树。置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。
   b) 在 F 中删去这两棵二叉树。
   c) 把新的二叉树加入 F。

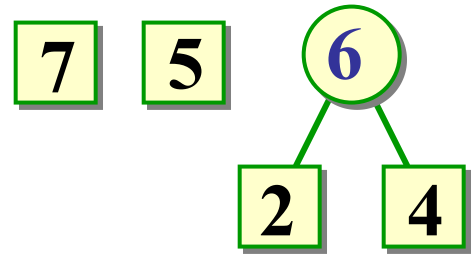
Huffman树的构造过程

F : {7} {5} {2} {4}



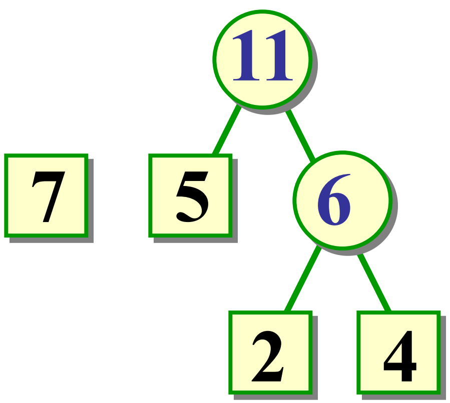
初始

F : {7} {5} {6}



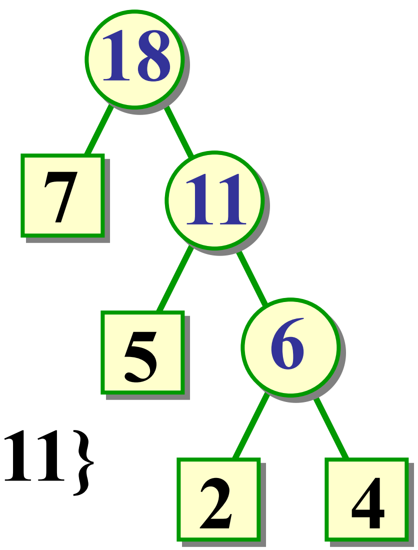
合并{2} {4}

F : {7} {11}



合并{5} {6}

F : {18}



合并{7} {11}

Huffman树的类定义

```
#include "heap.h"
const  int DefaultSize = 20;        //缺省权值集合大小
template <class E>
struct HuffmanNode {                //树结点的类定义
    E data;                         //结点数据
    HuffmanNode<E> *parent;
    HuffmanNode<E> *leftChild, *rightChild;
                       //左、右子女和父结点指针
    HuffmanNode () : parent(NULL), leftChild(NULL),
        rightChild(NULL) { }        //构造函数
```

```
    HuffmanNode (E elem,                    //构造函数
        HuffmanNode<E> *pr = NULL,
        HuffmanNode<E> *left = NULL,
        HuffmanNode<E> *right = NULL)
            : data (elem), parent (pr), leftChild (left),
              rightChild (right) { }
};
```

```
 template <class E>
class HuffmanTree {                    //Huffman树类定义
public:
    HuffmanTree (E w[], int n); //构造函数
    ～HuffmanTree() {deleteTree (root);}  //析构函数
protected:
    HuffmanNode<E> *root;          //树的根
    void deleteTree (HuffmanNode<E> *t);
        //删除以 t 为根的子树
    void mergeTree (HuffmanNode<E>& ht1,
        HuffmanNode<E>& ht2,
        HuffmanNode<E> *& parent) ;
};
```

建立Huffman树的算法

```
template <class E>
HuffmanTree<E>::HuffmanTree (E w[], int n) {
//给出 n 个权值w[0]～w[n-1], 构造Huffman树
    minHeap<E> hp;          //使用最小堆存放森林
    HuffmanNode<E> *parent, &first, &second;
    HuffmanNode<E> *NodeList =
        new HuffmanNode<E>[n];   //森林
    for (int i = 0; i < n; i++) {
        NodeList[i].data = w[i];
        NodeList[i].leftChild = NULL;
```

```
        NodeList[i].rightChild = NULL;
        NodeList[i].parent = NULL;
        hp.Insert(NodeList[i]);     //插入最小堆中
    }
    for (i = 0; i < n-1; i++) {      //n-1趟, 建Huffman树
        hp.Remove (first);           //根权值最小的树
        hp.Remove (second);          //根权值次小的树
        mergeTree (first, second, parent);     //合并
        hp.Insert (*parent);         //重新插入堆中
    }
    root = parent;              //建立根结点
};
```

```
template <class E>
void HuffmanTree<E>::
mergeTree (HuffmanNode<E>& bt1,
        HuffmanNode<E>& bt2,
        HuffmanNode<E> *& parent) {
    parent = new HuffmanNode<E>;
    parent->leftChild = &bt1;
    parent->rightChild = &bt2;
    parent->data.key =
        bt1.root->data.key+bt2.root->data.key;
    bt1.root->parent = bt2.root->parent = parent;
};
```

Huffman编码

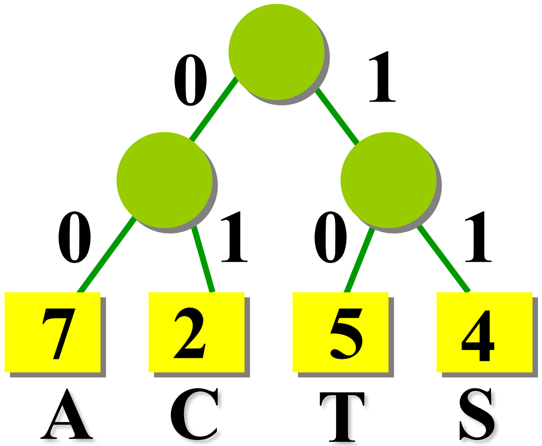
- 主要用途是实现数据压缩。设给出一段报文:

  CAST CAST SAT AT A TASA

- 字符集合是 { **C, A, S, T** }，各个字符出现的频度（次数）是 W = { **2, 7, 4, 5** }。
- 若给每个字符以等长编码（2位二进制足够）

  A : 00　T : 10　C : 01　S : 11

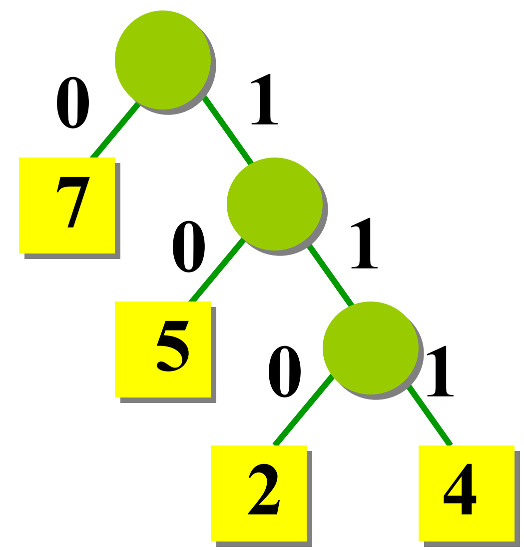
- 则总编码长度为 **(2+7+4+5) * 2 = 36**。
- 能否减少总编码长度，使得发出同样报文，可以用最少的二进制代码?



- 若按各个字符出现的概率不同而给予不等长编码，可望减少总编码长度。
- 各字符出现概率为{ 2/18, 7/18, 4/18, 5/18 },化整为 { 2, 7, 4, 5 }。以它们为各叶结点上的权值, 建立Huffman树。左分支赋 0，右分支赋 1，得Huffman编码(变长编码)。

A : 0　T : 10　C : 110　S : 111

- 它的总编码长度：7*1+5*2+(2+4)*3 = 35。比等长编码的情形要短。
- 总编码长度正好等于Huffman树的带权路径长度WPL。
- Huffman编码是一种前缀编码，即任一个二进制编码不是其他二进制编码的前缀。解码时不会混淆。



Huffman编码树