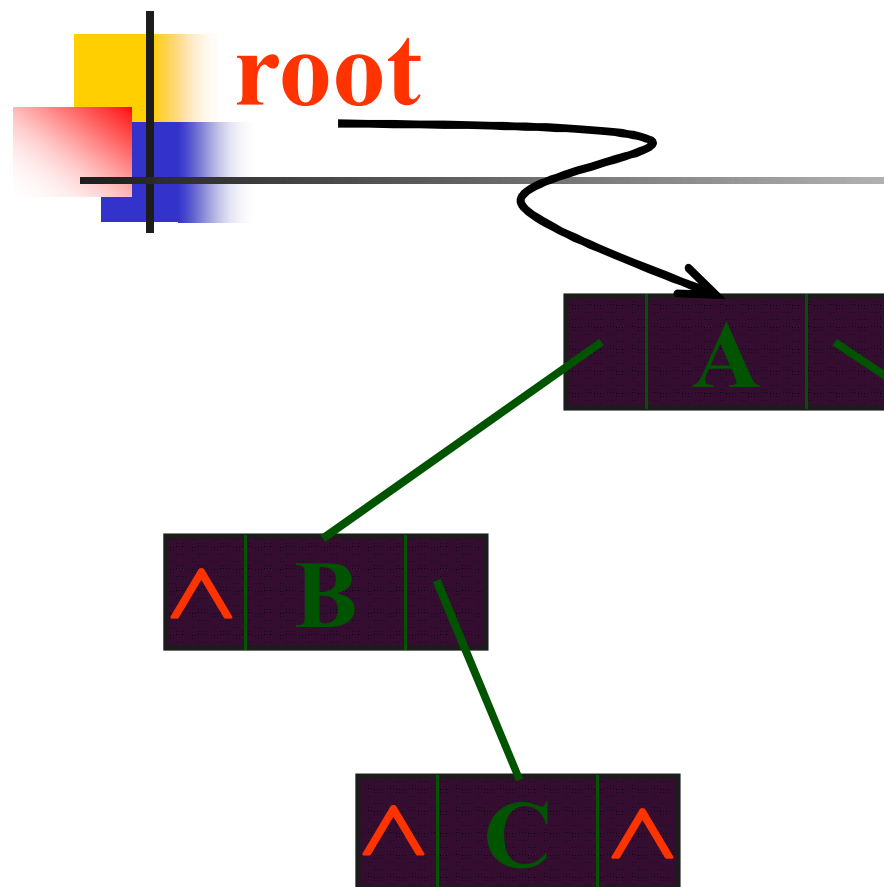


# 线索二叉树



空指针数量?

先序: ABCDEF

中序: BCADFE

后序: CBFEDA



## 线索二叉树定义

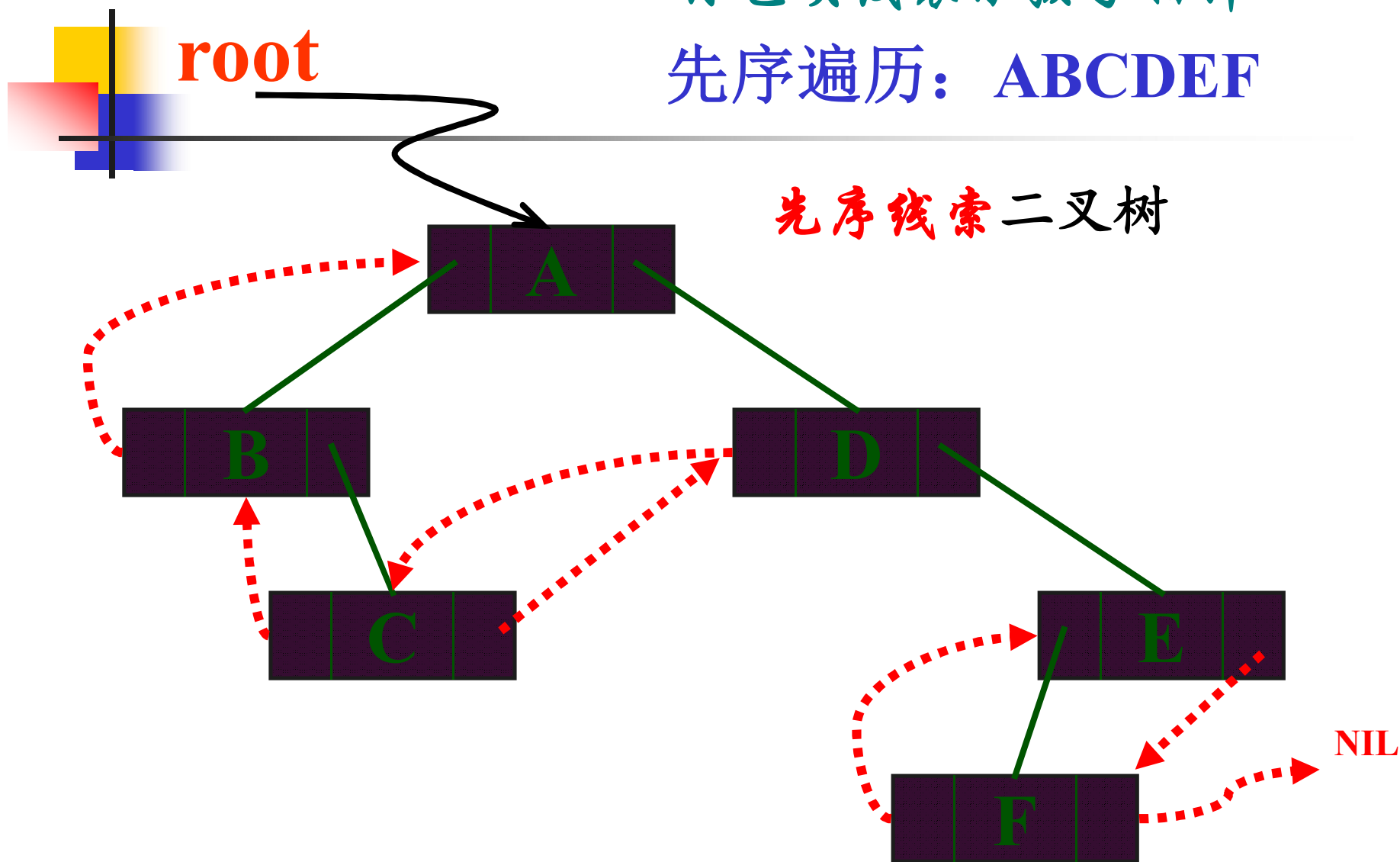
---

- $n$ 个结点的二叉链表中含有 $n+1$ 个空指针域。利用二叉链表中的空指针域，存放指向结点在某种遍历次序下的前趋和后继结点的指针（这种附加的指针称为"线索"）。

红色虚线代表线索

绿色实线表示孩子指针

先序遍历: ABCDEF





## 线索二叉树定义

---

- 加上了线索的二叉链表称为**线索链表**，相应的二叉树称为**线索二叉树** (Threaded BinaryTree)。
- 根据线索性质的不同，线索二叉树可分为：  
**先序线索**二叉树、  
**中序线索**二叉树、  
**后序线索**二叉树。

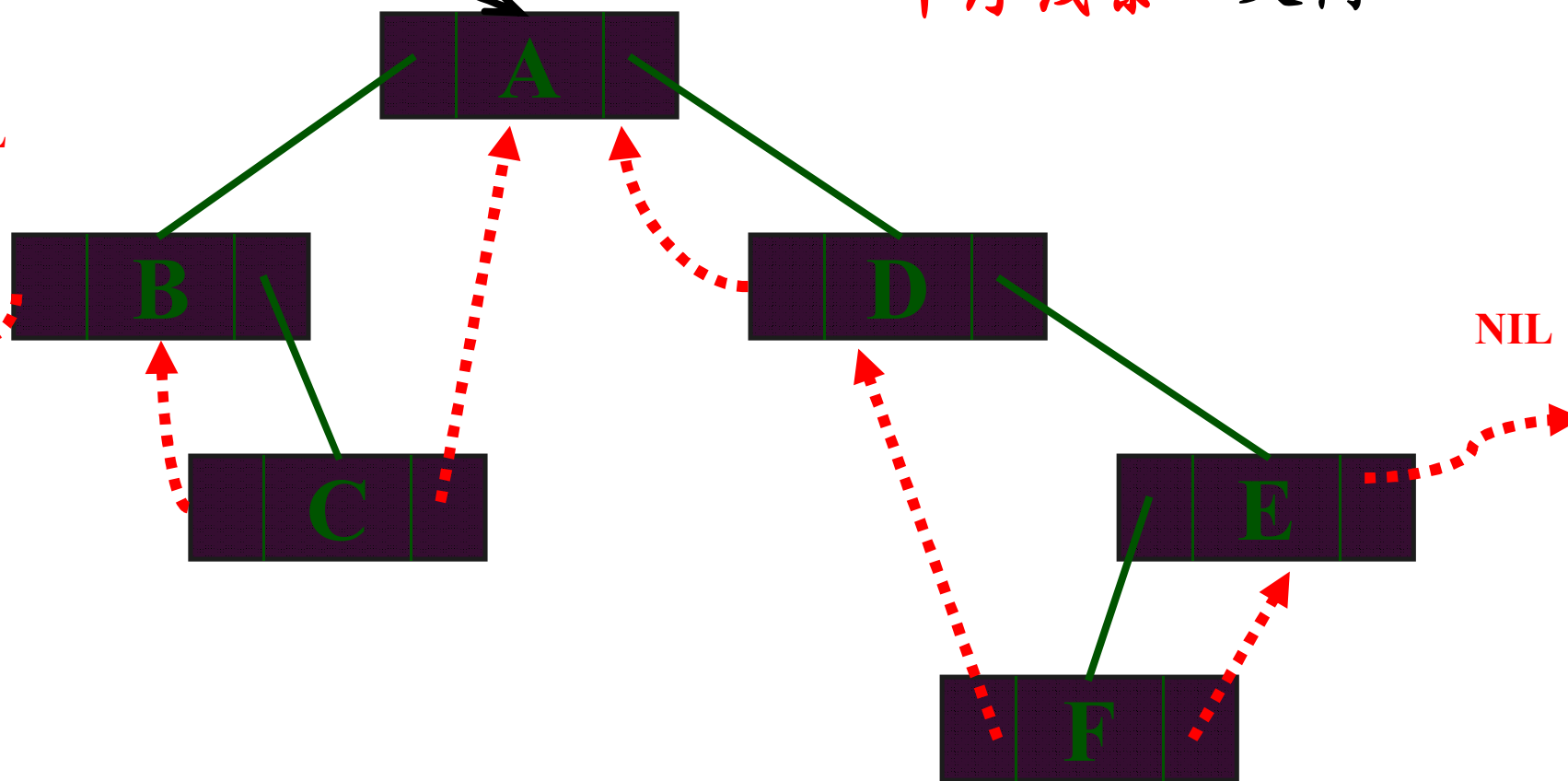
root

中序遍历: BCADFE

中序线索二叉树

NIL

NIL





## 线索链表存在的问题

- 指针非空如何区分是**孩子指针**还是**线索**？
- 在结点结构中增加标志域**LTag**和**RTag**来指示lc和rc指针域中存放的是线索还是孩子指针

lc	LTag	data	RTag	rc
----	------	------	------	----

约定：



lc	LTag	data	RTag	rc
----	------	------	------	----

若该结点的左子树不空，则lc指向其左子树，  
且LTag的值为“指针 Link”； 否则，lc指向  
其“前驱”，且LTag的值为“线索 Thread”。



若该结点的右子树不空，则rc指向其右子树，  
且RTag的值为“指针 Link”； 否则，rc指向  
其“后继”，且RTag的值为“线索 Thread”。

如此定义的二叉树的存储结构称作“线索链表”。



## 线索链表的类型描述:

```
typedef enum { Link, Thread } PointerTag;
```

```
// Link==0:指针, Thread==1:线索
```

```
typedef struct BiThrNod {
```

```
TElemType    data;
```

```
struct BiThrNode *lc, *rc;// 左右指针
```

```
PointerTag  LTag, RTag;  // 左右标志
```

```
} BiThrNode, *BiThrTree;
```

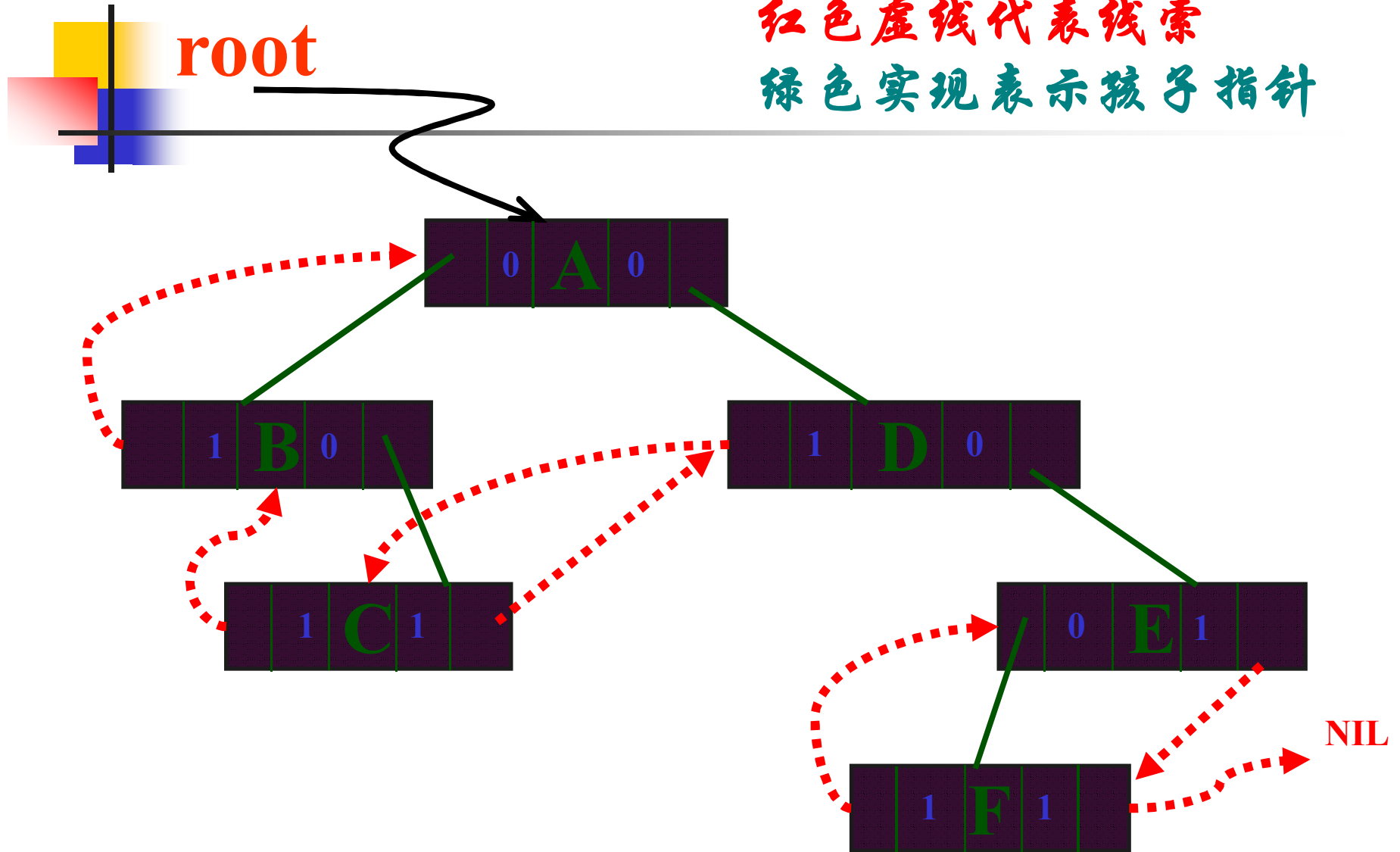




# 先序线索二叉树

红色虚线代表线索

绿色实线表示孩子指针





## 二叉树的线索化

---

- 将二叉树变为线索二叉树的过程称为线索化。
- 按某种次序将二叉树线索化的实质是：  
按该次序遍历二叉树，在遍历过程中用线索取代空指针。



## 二叉树的中序线索化

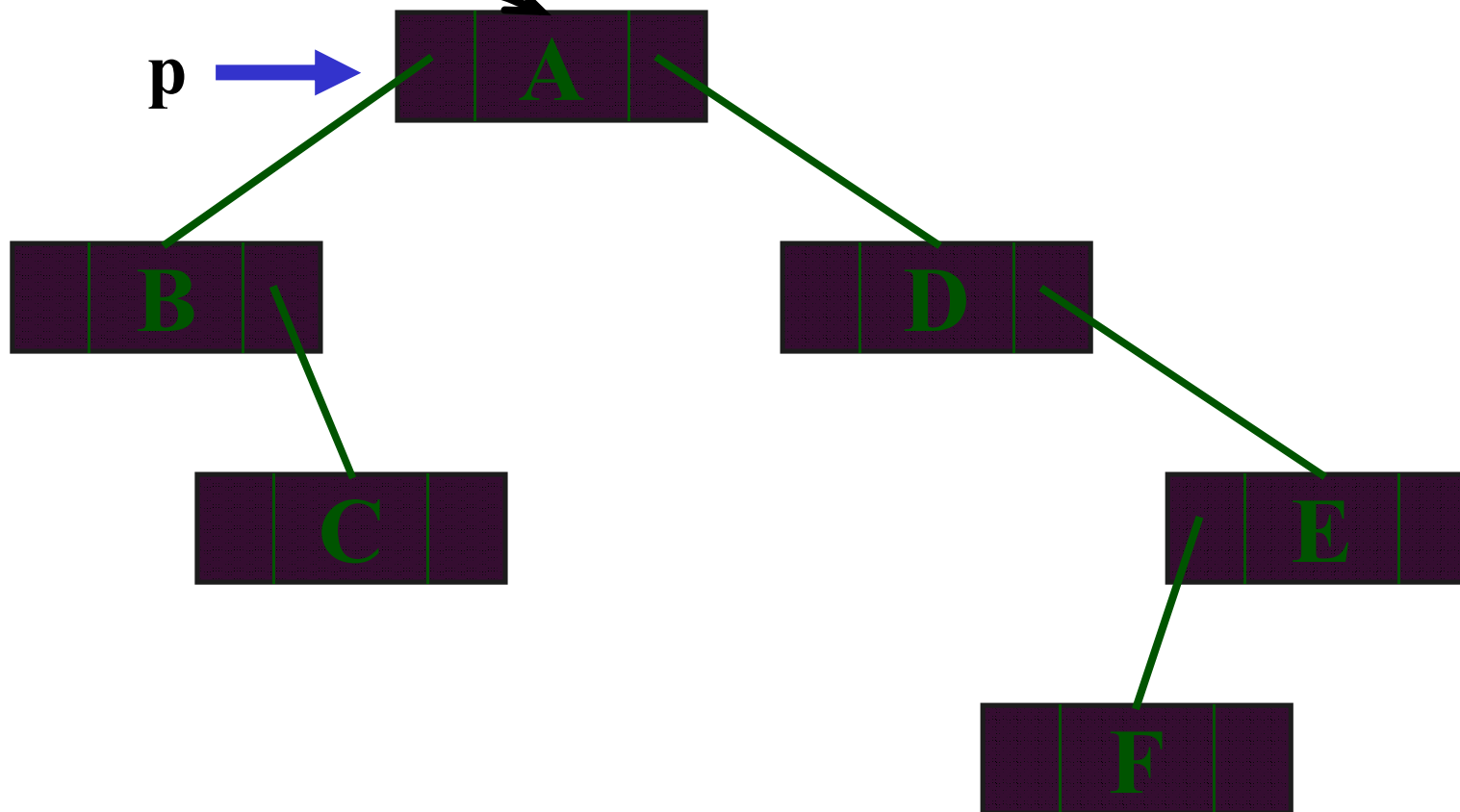
- 分析：算法与中序遍历算法类似。只需要将遍历算法中访问结点的操作具体化为建立正在访问的结点与其非空中序前趋结点间线索。
- 算法应附设一个指针pre始终指向刚刚访问过的结点（pre的初值应为NULL），而指针p指示当前正在访问的结点。结点\*pre是结点\*p的前趋，而\*p是\*pre的后继。

thrt

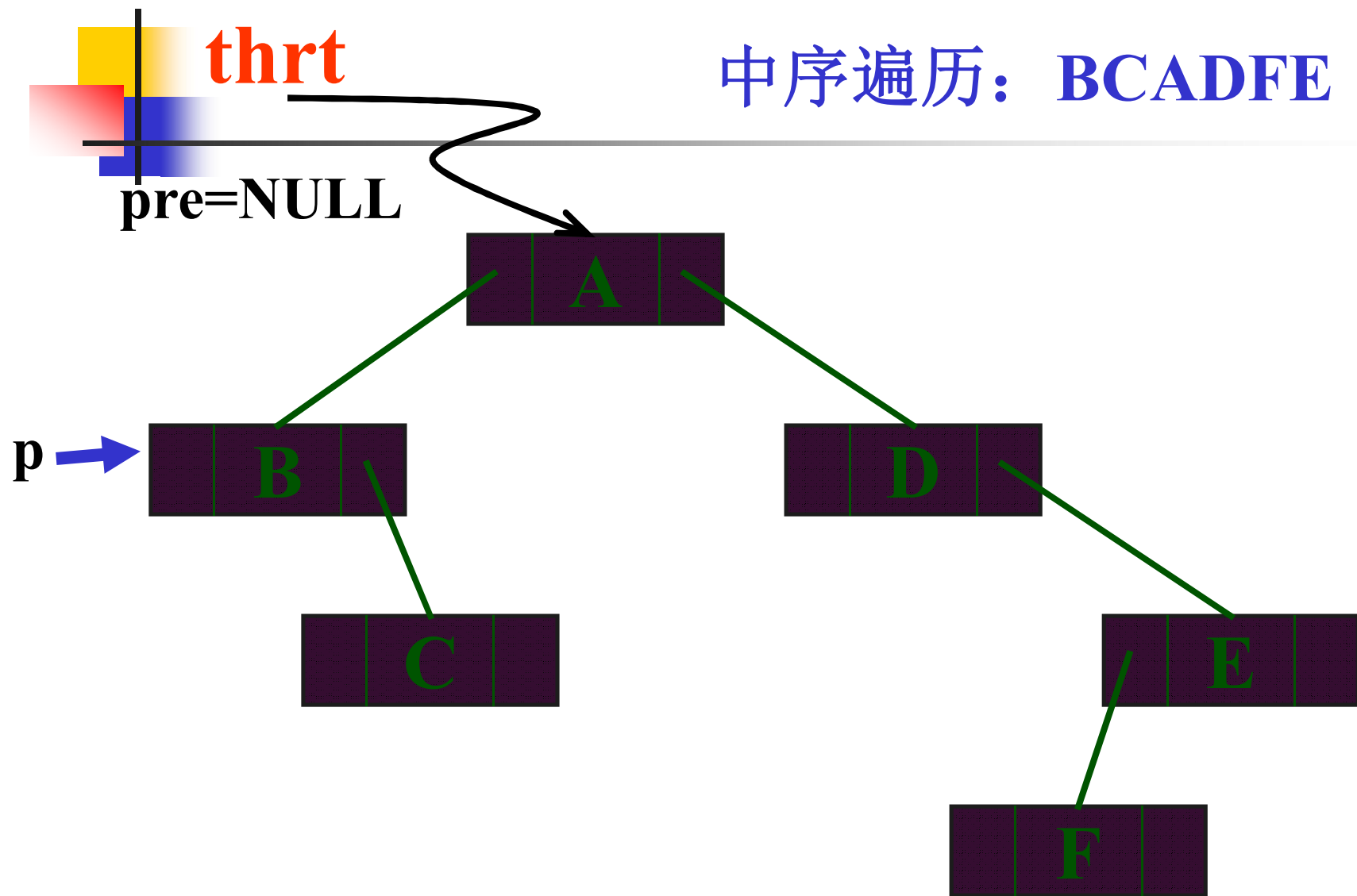
中序遍历: BCADFE

pre=NULL

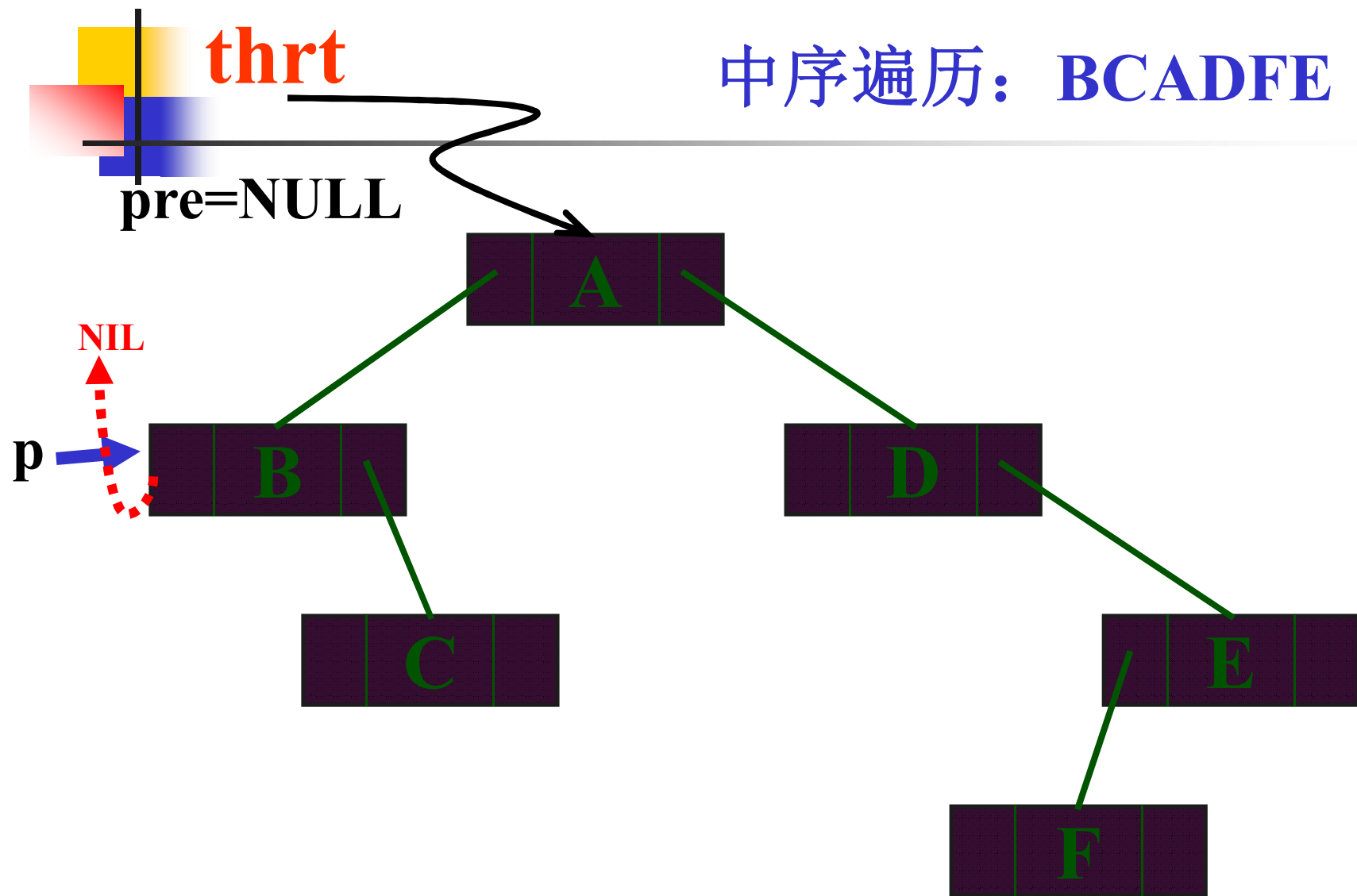
p

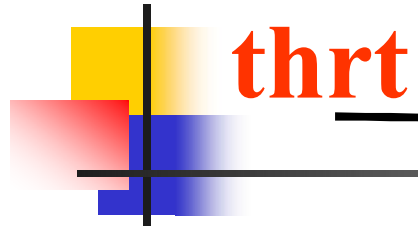


中序遍历: BCADFE



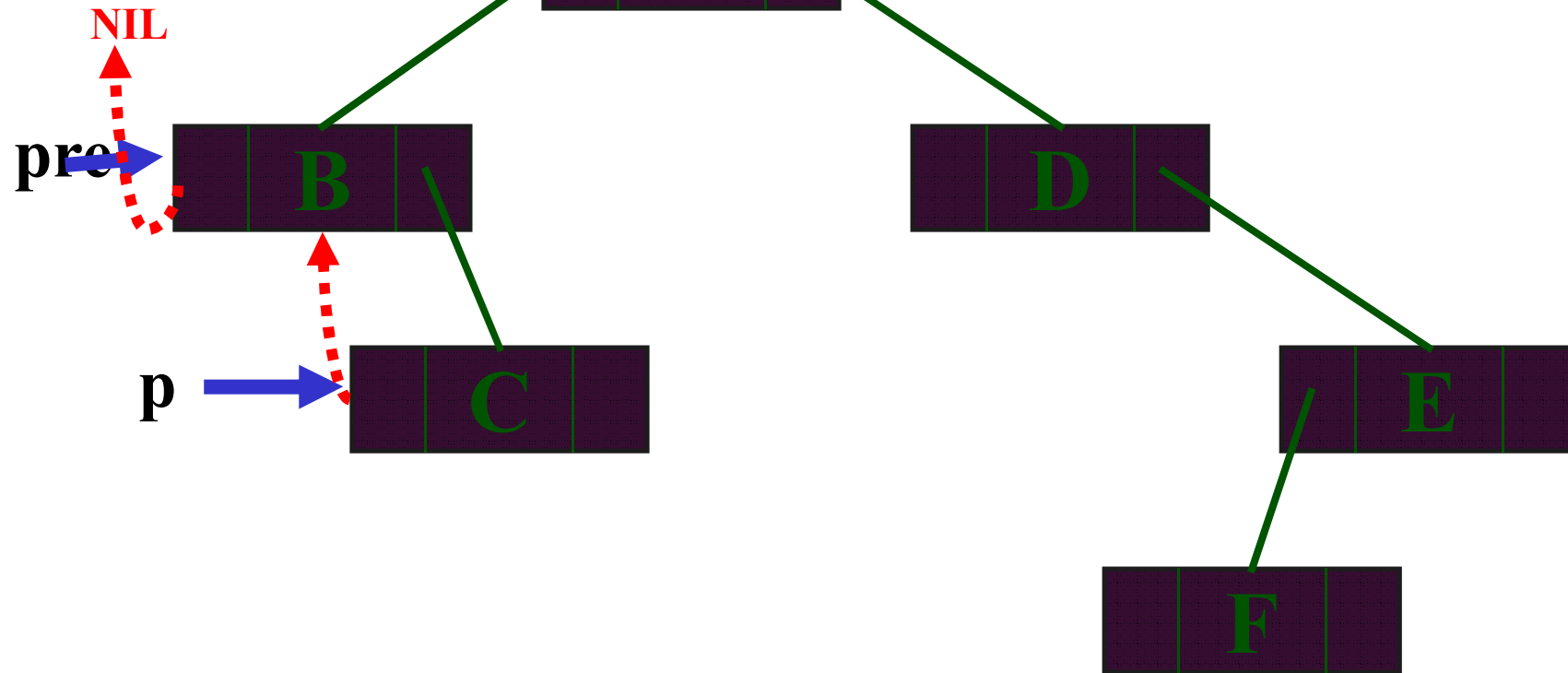
中序遍历: BCADFE





thrt

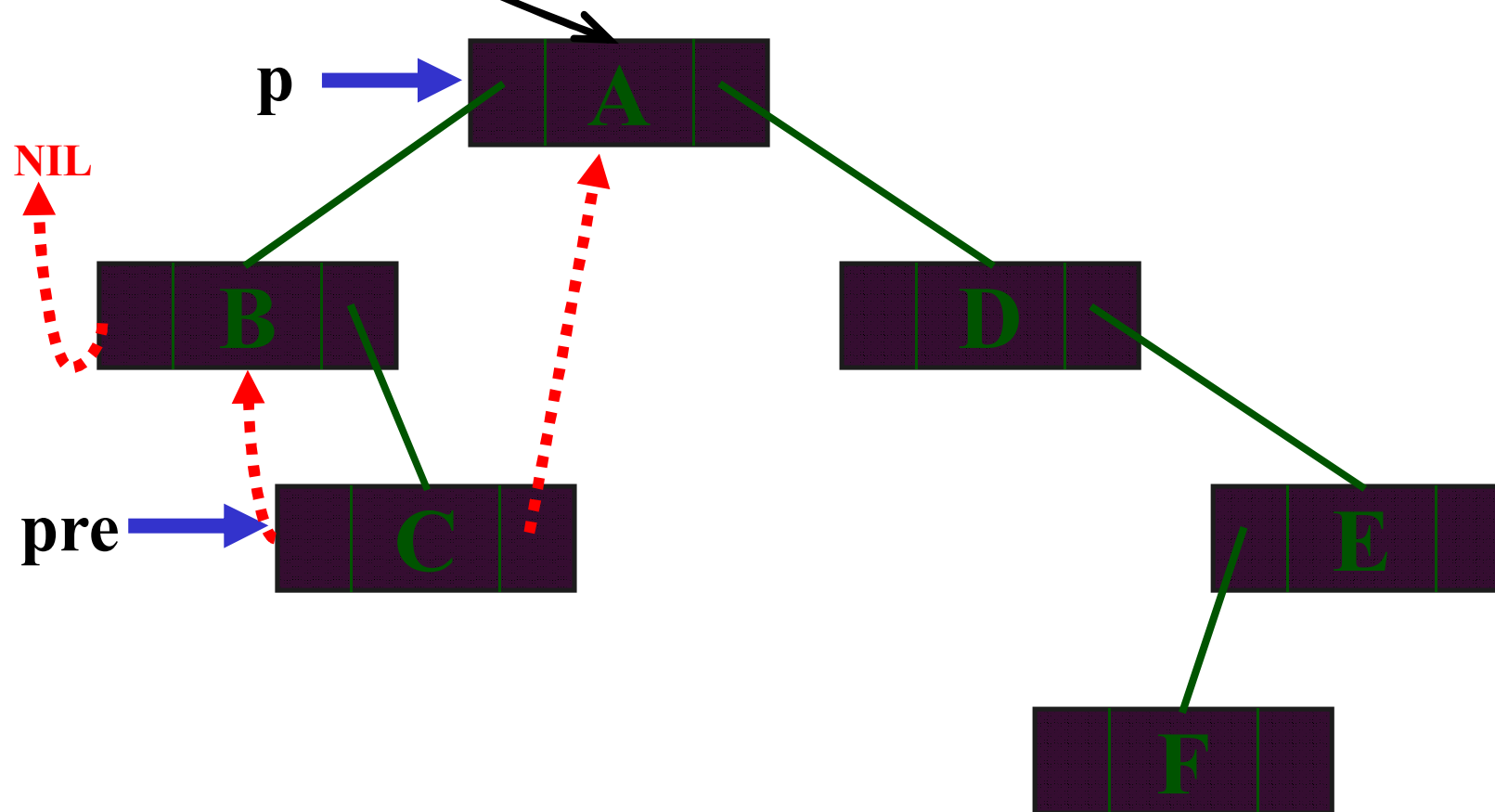
中序遍历: BCADFE





thrt

中序遍历: BCADFE

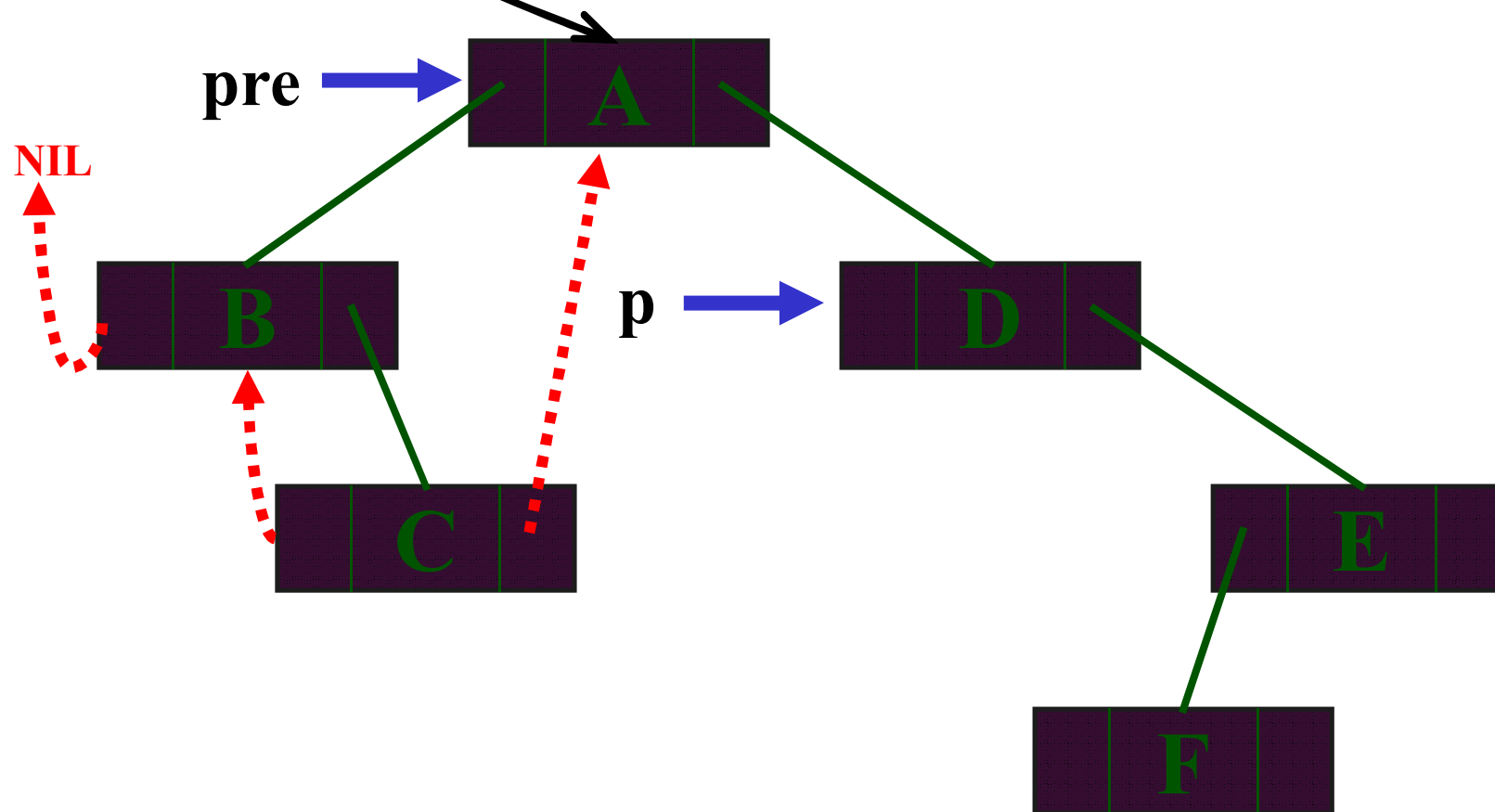






thrt

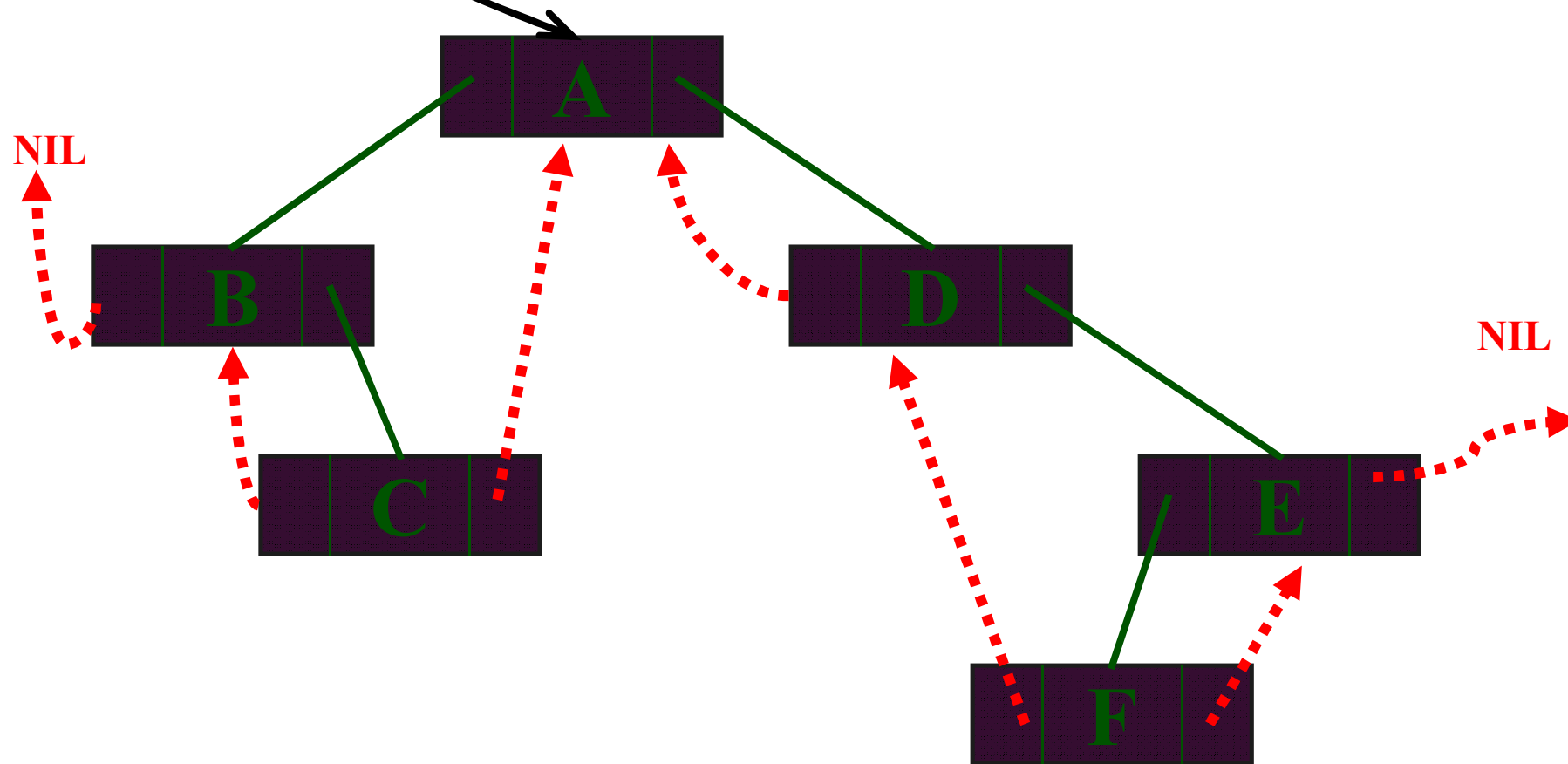
中序遍历: BCADFE





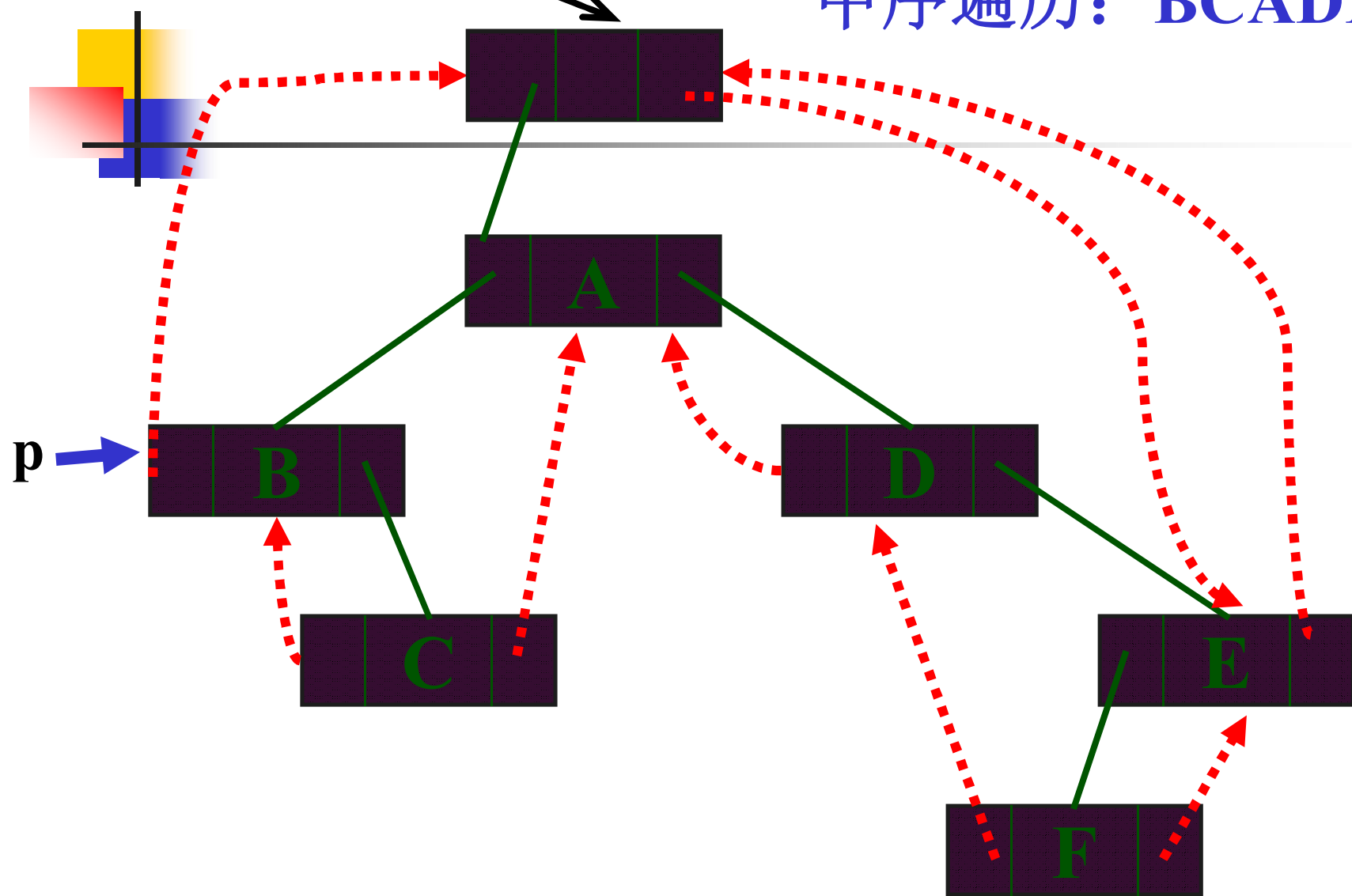
thrt

中序遍历: BCADFE



thrt

中序遍历: BCADFE



**void InorderThreading(BiThrTree &Thrt, BiThrTree T)**

**{ Thrt=(BiThrTree)malloc(sizeof(BiThrNode));**

**if(!Thrt)exit(overflow);**

**Thrt->LTag=Link ; Thrt->RTag= Thread ;**

**Thrt->rc=Thrt ;**

**if(!T) Thrt->lc=Thrt;**

**else**

**{ Thrt->lc=T; pre=Thrt;**

**InThreading(T);**

**pre->rc=Thrt;pre->RTag=Thread;**

**Thrt->rc=pre;**

**}**

**}**

**void InThreading(BiThrTree p)**



**{ if(p)**

**~~{ InThreading(p->lc);~~**

**if(!p->lc)**

**{p->LTag=Thread; p->lc=pre;}**


**if(!pre->rc)**

**{ pre->RTag=Thread; pre->rc=p;}**

**pre=p;**

**InThreading(p->rc); }**

**}**



## 线索链表的遍历算法：

由于在线索链表中添加了遍历中得到的  
“**前驱**”和“**后继**”的信息，从而简化  
了遍历的算法：

```
for ( p = firstNode(T); p; p = Succ(p) )  
    Visit (p);
```

- 说明：
1. **firstNode(T)** 函数功能是取线索二叉树的第一个访问结点；
  2. **Succ(p)** 函数功能是取线索二叉树的p结点的直接后继结点
  3. 先序线索二叉树、中序线索二叉树、后序线索二叉树的  
**firstNode(T)**和**Succ(p)**函数实现方法不同

# 对**中序**线索化链表的遍历算法

## ※ **中序遍历的第一个结点** ?

— 左子树上处于“**最左下**”（没有左子树）的结点。

```
BiThrTree firstNode(BiThrTree T)
{
    p=T;
    while(p->LTag=Link)p=p->lc;
    return p;
}
```

## ※ **在中序线索化链表中结点的后继** ?

**若**无右子树，**则**为后继线索所指结点；

**否则**为对其右子树进行中序遍历时访问的第一个结点。



**BiThrTree Succ(BiThrTree p)**

---

```
{  
    if (p->RTag==Thread) return p->rc;  
    else  
    {  
        p= p->rc;  
        while (p->LTag==Link) p = p->lc;  
        return p;  
    }  
} )//在中序线索二叉树中查找结点p的中序遍历的直接后继
```