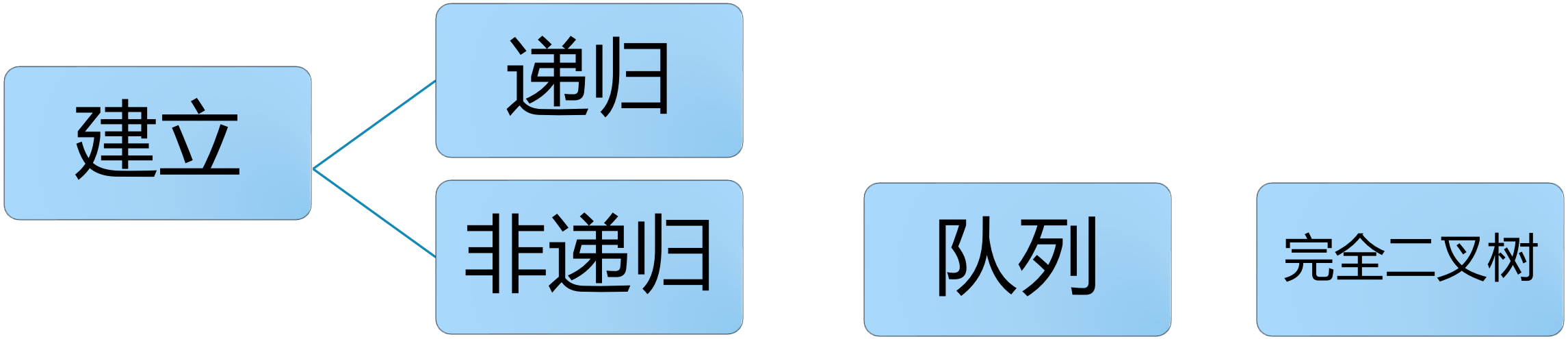
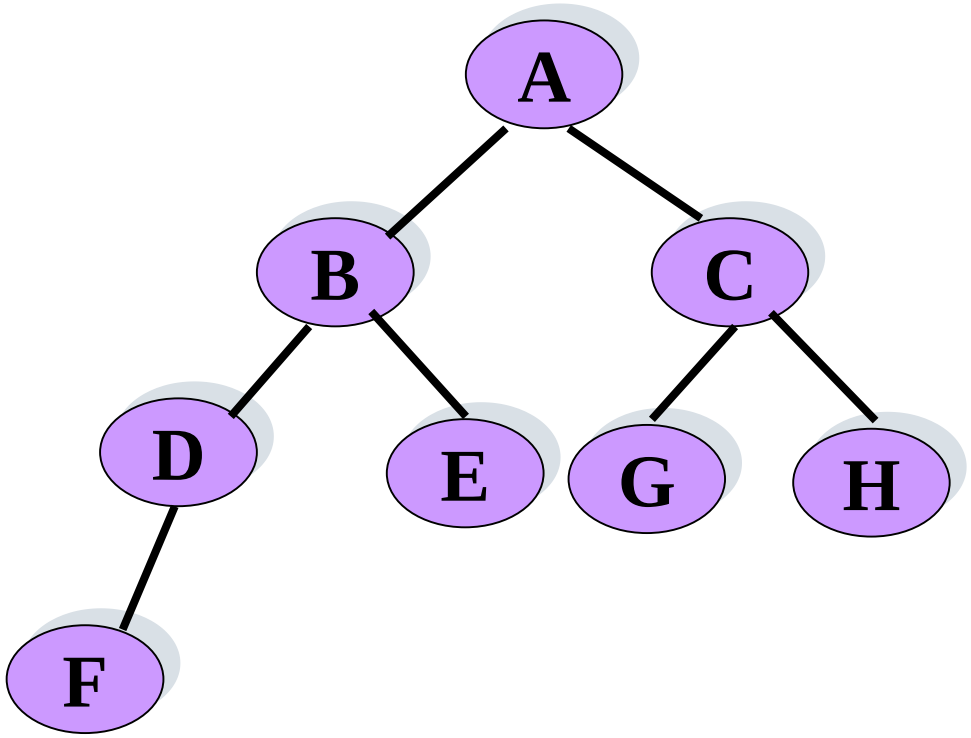
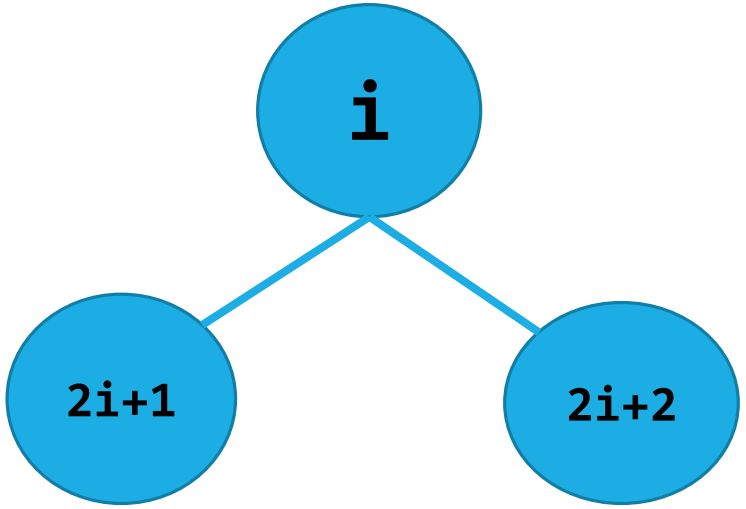
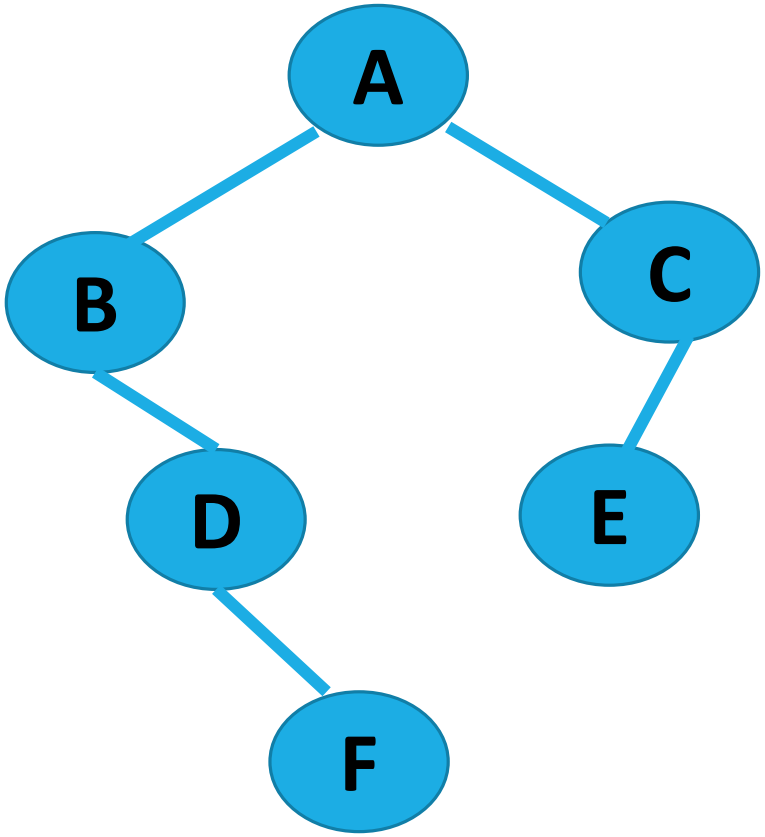
4.10 二叉树的建立和遍历-非递归算法

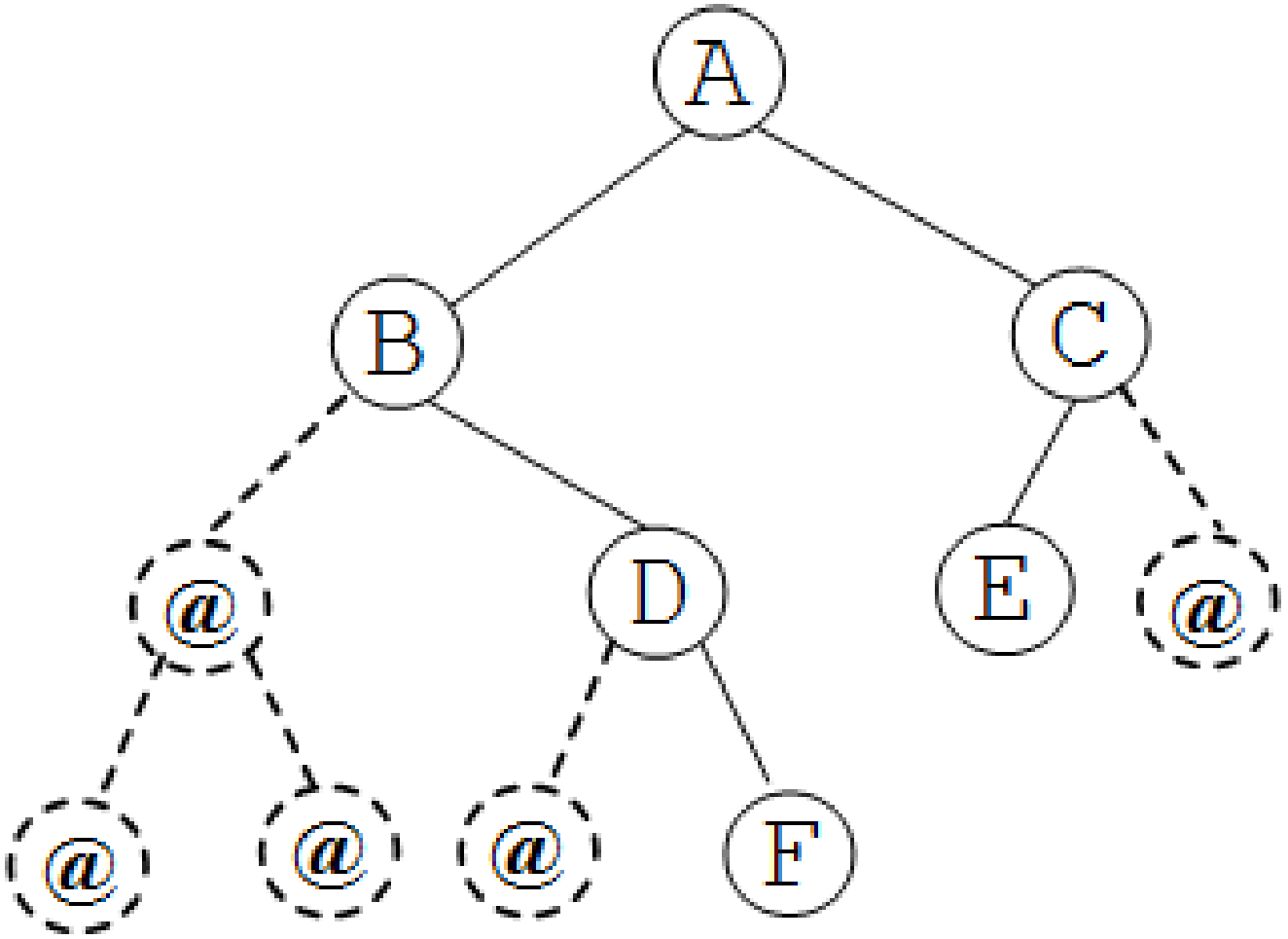


4.10 二叉树的建立和遍历--非递归建立算法






A
B
C
D
E
F


A
B
C
@
D
E
@
@
@
@
F

（1）将二叉树扩充为完全二叉树，输入完全二叉树序列，以#作为结束标志，设置计数器count为-1，用来标识结点的序号。

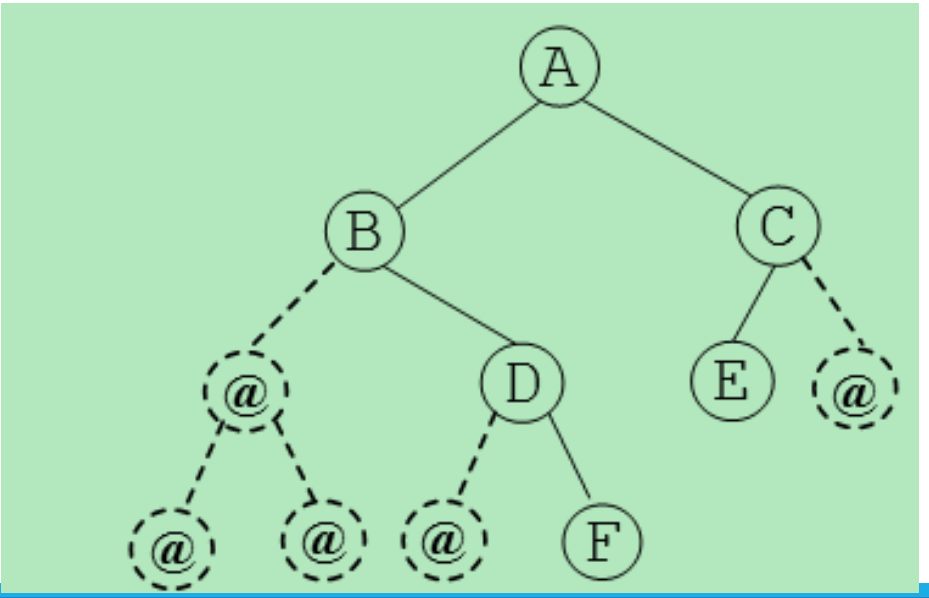
（2）如果输入的不是@，则生成一个新结点s，并对结点的数据域赋值为输入的字符，结点的左右指针赋值为空，结点s入队，计数器count加1；

（3）如果计数器count等于0，这个结点就是根结点，设置二叉树的根bt = s；

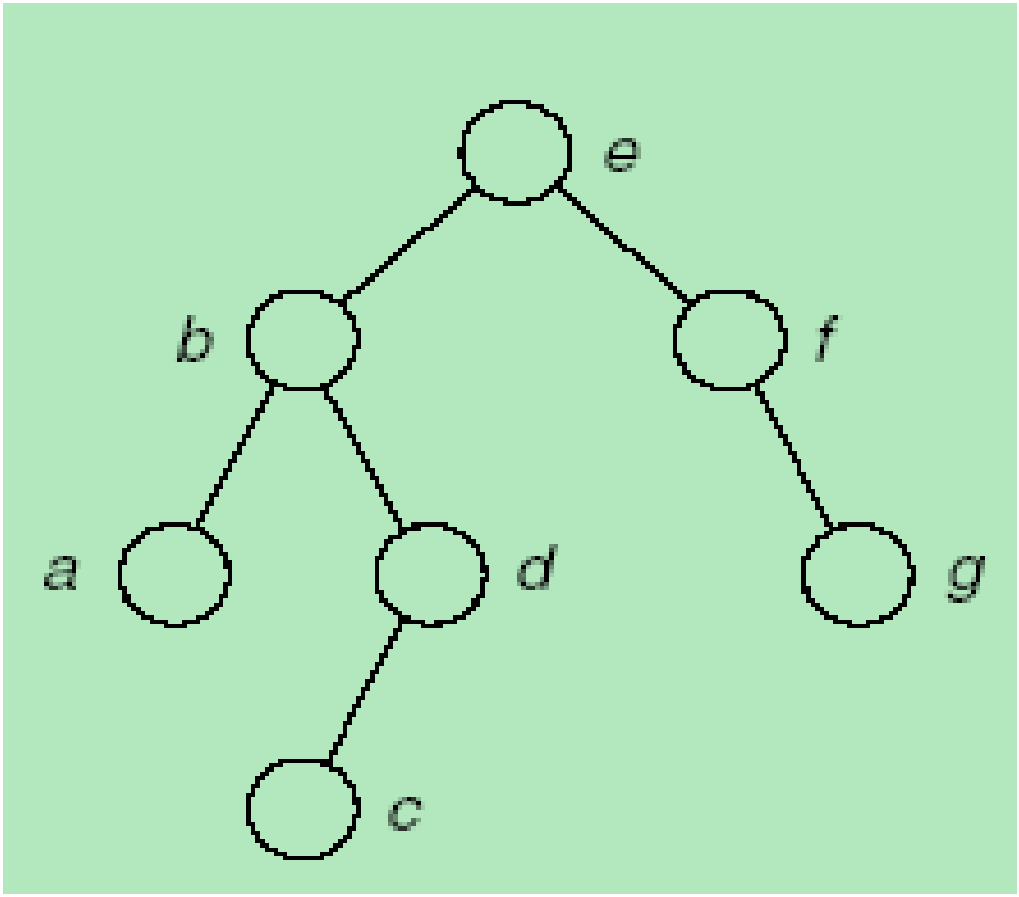
如果count是奇数，则是父结点p（队头结点）的左孩子，即p->leftchild = s;

如果count是偶数，

- 父结点p（队头结点）的右孩子，即p->rightchild = s；
- 队头结点的左右孩子已经处理完毕，出队；



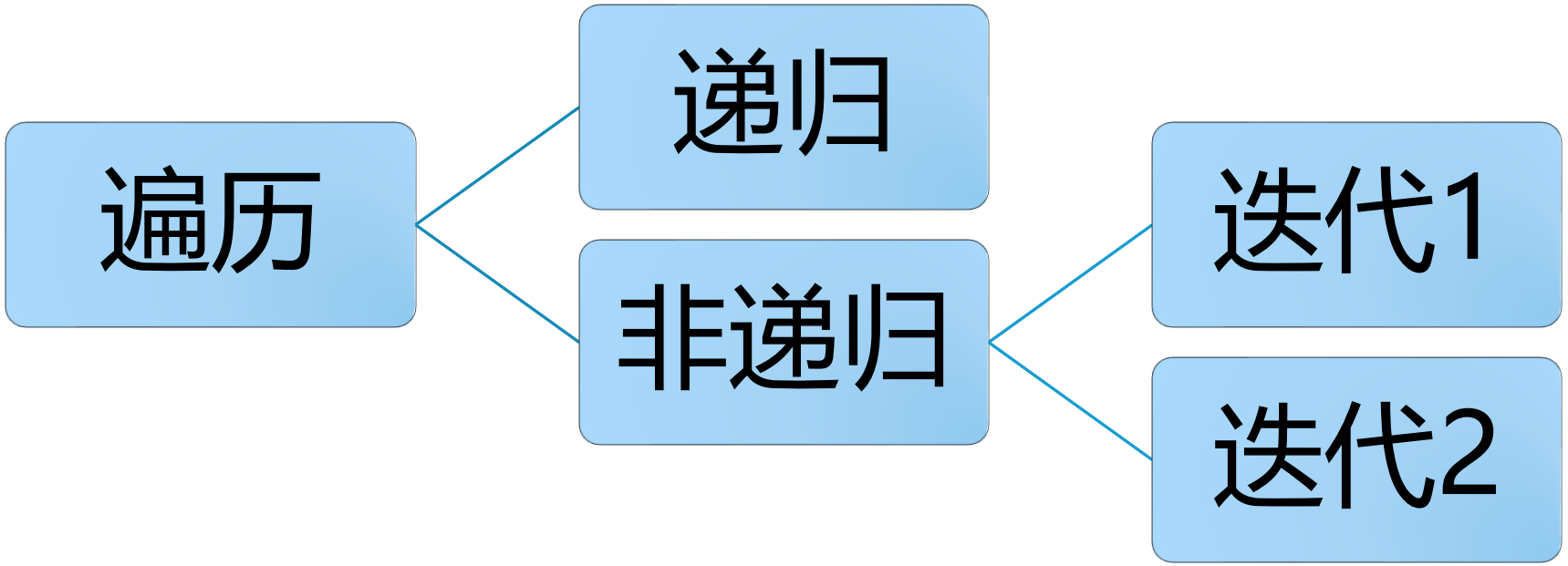
输入序列：ABC@DE@@@@F#


e
b
f
a
d
g
c

4.9 二叉树的建立和遍历—层次遍历算法　　算法4-9

```
1   void LevelOrder(BinTree bt)   //使用队列层次遍历二叉树
2   {
3           BinTree p;
4           LinkQueue queue = SetNullQueue_Link(); //创建空队列
5           if (bt == NULL) return;
6           p = bt;
7           EnQueue_link(queue, bt);                //根结点入队
8           while (!IsNullQueue_Link(queue))     //队列不空，循环执行
9           {
10                  p = FrontQueue_link(queue);      //取队头
11                  DeQueue_link(queue);             //出队
12                  printf("%c ", p->data);
13                  if (p->leftchild != NULL)
14                          EnQueue_link(queue, p->leftchild); //左孩子不空，则入队
15                  if (p->rightchild != NULL)
16                          EnQueue_link(queue, p->rightchild); //右孩子不空，则入队
17          }
18  }
```

二叉树的建立和遍历-非递归算法

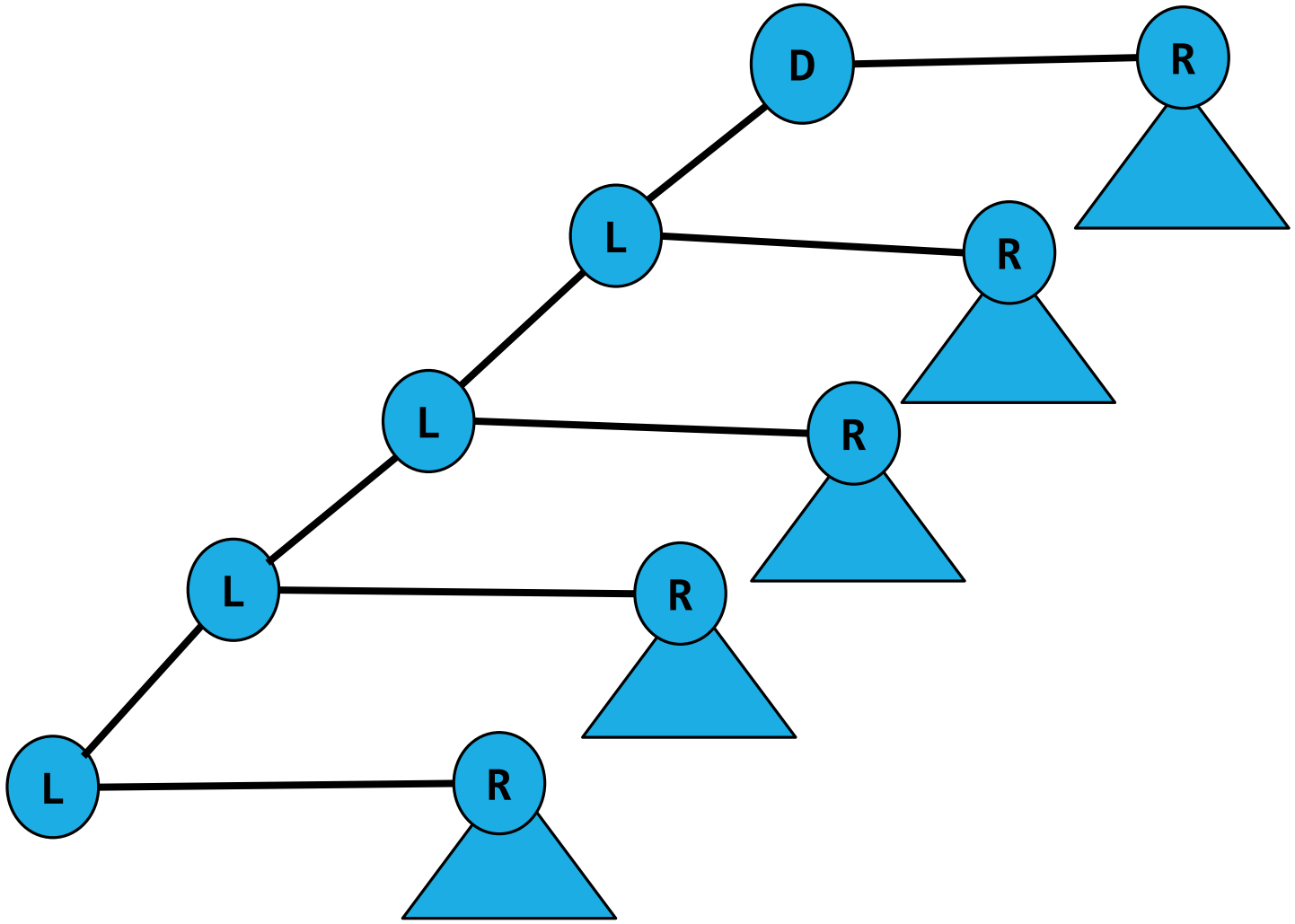


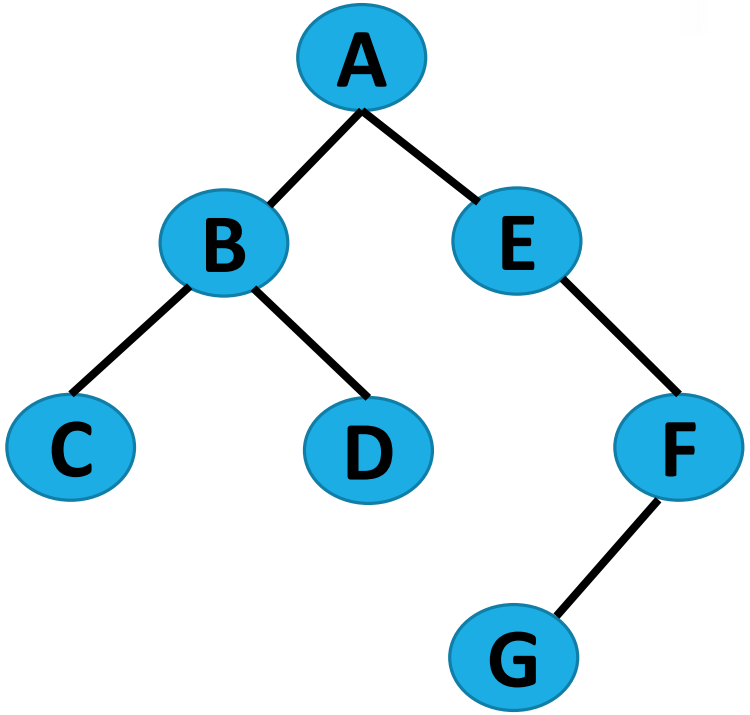
先序遍历的非递归实现：迭代1

算法思路1 先序遍历的过程是按照D->L->R的顺序访问结点。假设每个结点都入栈和出栈一次，并且**出栈的时候访问**，这样对每个结点的左右孩子的进栈的先后顺序应该是**右孩子先入栈然后左孩子入栈。**

（1）从根结点bt开始，将根结点压入栈lstack中；
（2）如果栈lstack不空，从栈lstack中弹出一个元素p，并访问；
（3）接着如果p的右孩子不空，入栈lstack；
（4）p的左孩子不空，入栈lstack；
（5）重复上述过程（2）~（4），直到栈lstack为空结束。

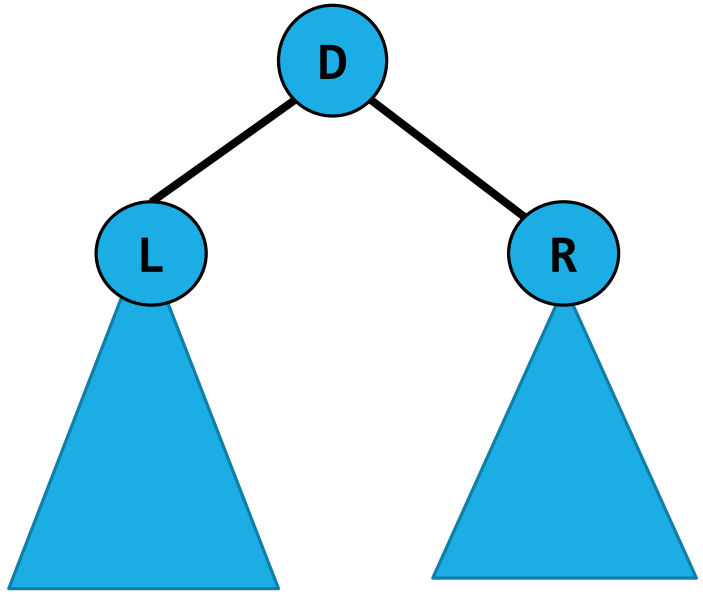
先序遍历的非递归实现：迭代1





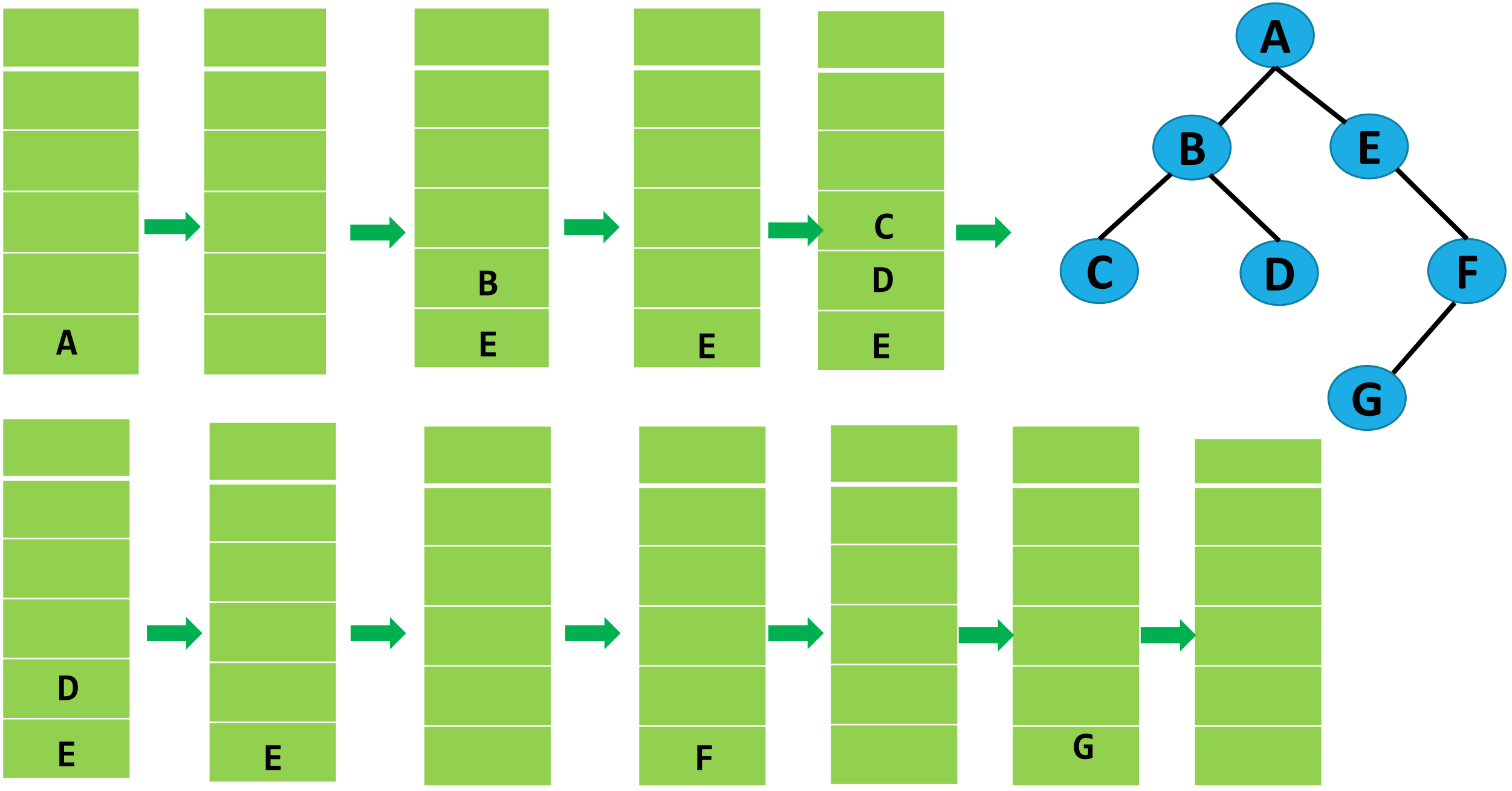
迭代1：先序遍历的非递归实现　　算法4-12

```
1  void PreOrder_NRecursion1(BinTree bt)
2  {
3      LinkStack lstack; //定义链栈
4      lstack = SetNullStack_Link(); //初始化栈
5      BinTreeNode *p;
6      Push_link(lstack, bt); //根结点入栈
7      while (!IsNullStack_link(lstack))
8      {
9          p = Top_link(lstack);
10         Pop_link(lstack);
11         printf("%c", p->data);          //访问结点
12         if (p->rightchild)
13             Push_link(lstack, p->rightchild);
14         if (p->leftchild)
15             Push_link(lstack, p->leftchild);
16     }
17 }
```

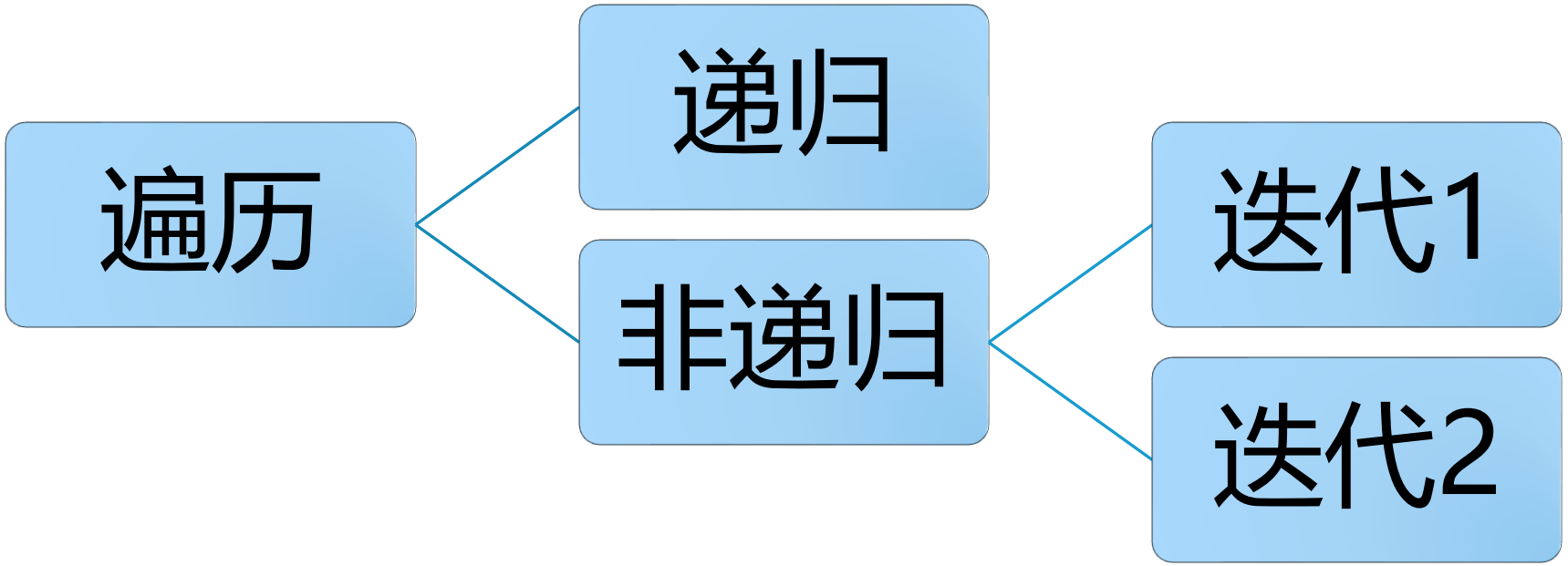


时间复杂度?

O(n)


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

二叉树的遍历算法



先序遍历的非递归实现

算法思路2 先序遍历的过程是按照D->L->R的顺序访问结点。假设考虑只是右孩子入栈，左孩子沿着左分支深入经过的时候就访问，不入栈。从根结点开始，沿着左分支深入、访问，并且将结点的右孩子入栈，直到到达一个空结点。然后检查栈是否为空，栈空结束，栈非空，出栈一个元素，重复上述过程。

（1）从根结点p开始；

（2）如果p不空，则访问它；

（3）接着如果p的右孩子不空，右孩子入栈lstack；

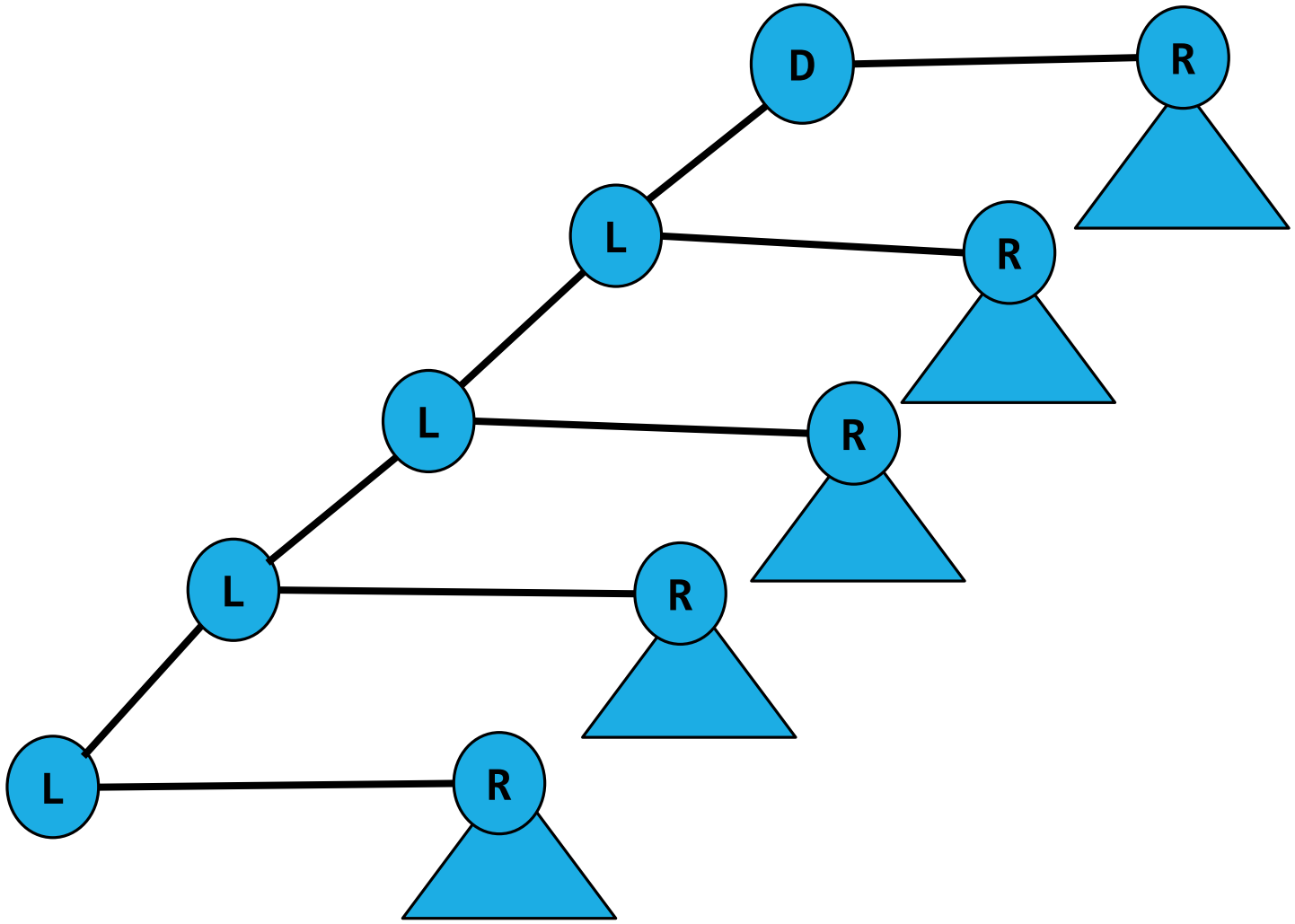
（4）p的左孩子不空，沿着左分支进入p的左子树；

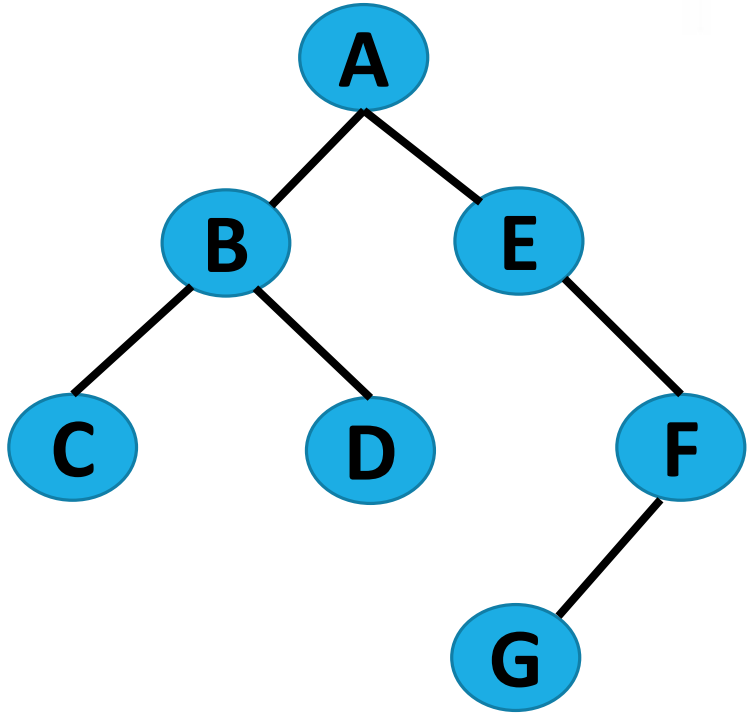
（5）重复上述过程（2）~（4），直到p为空（即沿着左分支深入不下去为止）；

（6）如果lstack栈空算法结束；

（7）栈lstack不空从栈中弹出一个元素，重复上述过程（2）~（5）。

先序遍历的非递归实现：迭代2

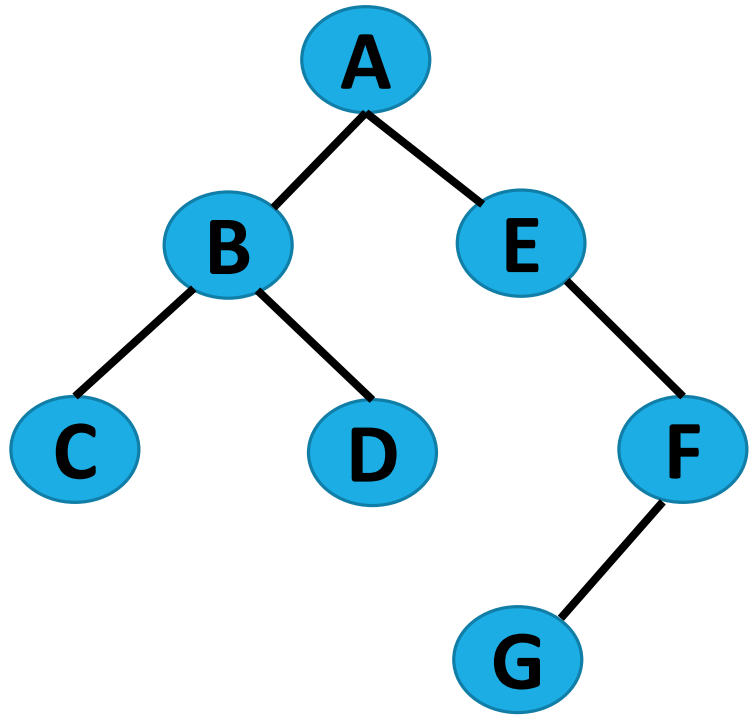




迭代2：先序遍历的非递归实现

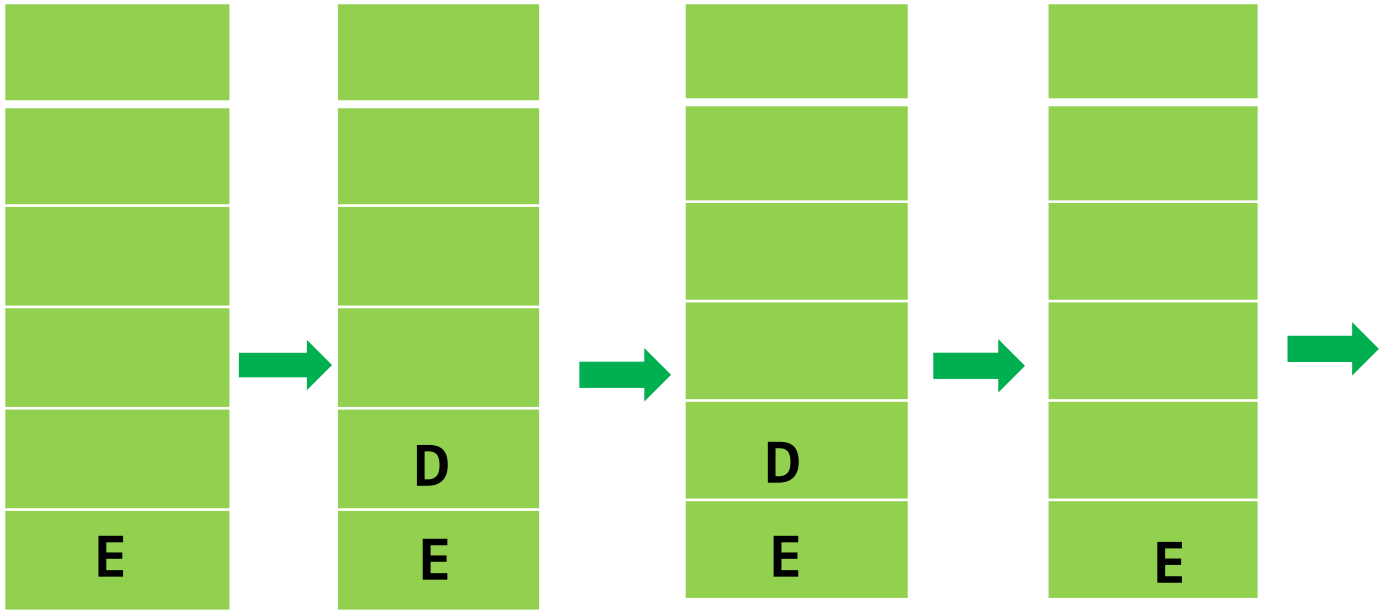
算法4-13

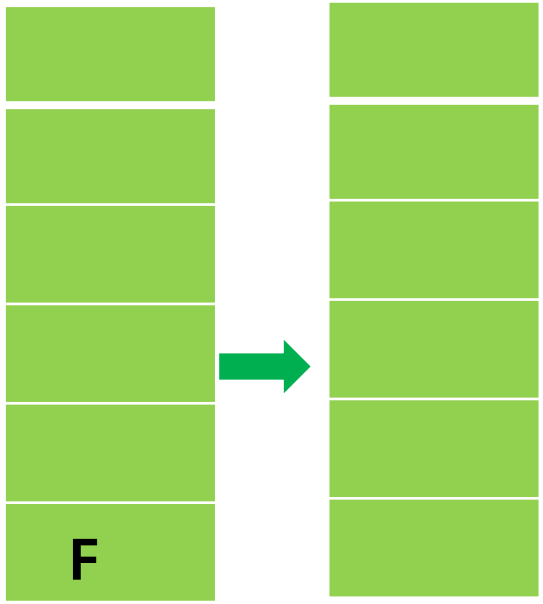
```
1  void PreOrder_NRecursion2(BinTree bt) {
2    LinkStack lstack; //定义链栈
3    BinTreeNode *p = bt;
4    lstack = SetNullStack_Link(); //初始化栈
5    if (bt == NULL) return;
6    Push_link(lstack, bt);
7    while (!IsNullStack_link(lstack)) {
8        p = Top_link(lstack);
9        Pop_link(lstack);
10       while (p) {
11           printf("%c", p->data); //访问结点
12           if (p->rightchild)   //右孩子是空，不用进栈
13                Push_link(lstack, p->rightchild);
14           p = p->leftchild;
15       }
16   }
17 }
```

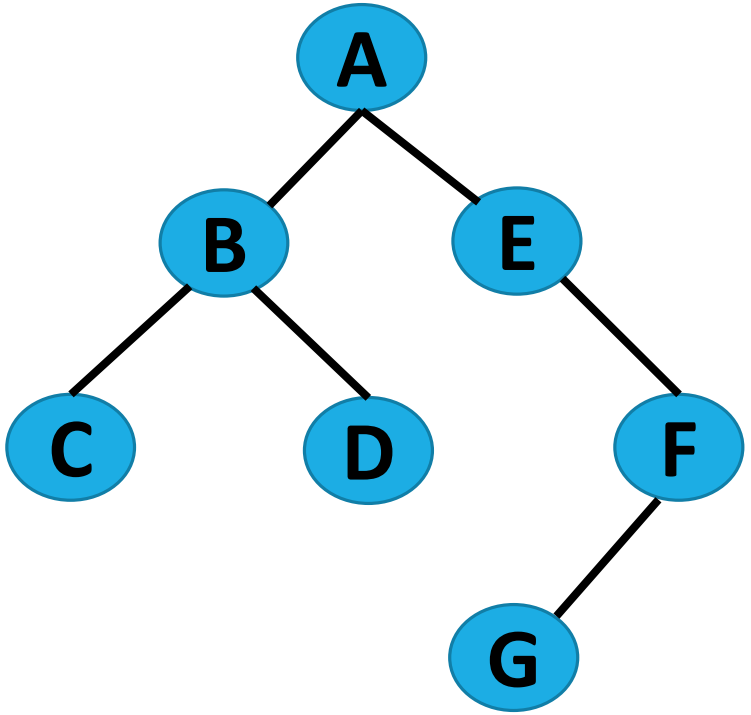


时间复杂度?

O(n)


E
D
E
D
E
E


F


A
B
E
C
D
F
G


举个栗子!!

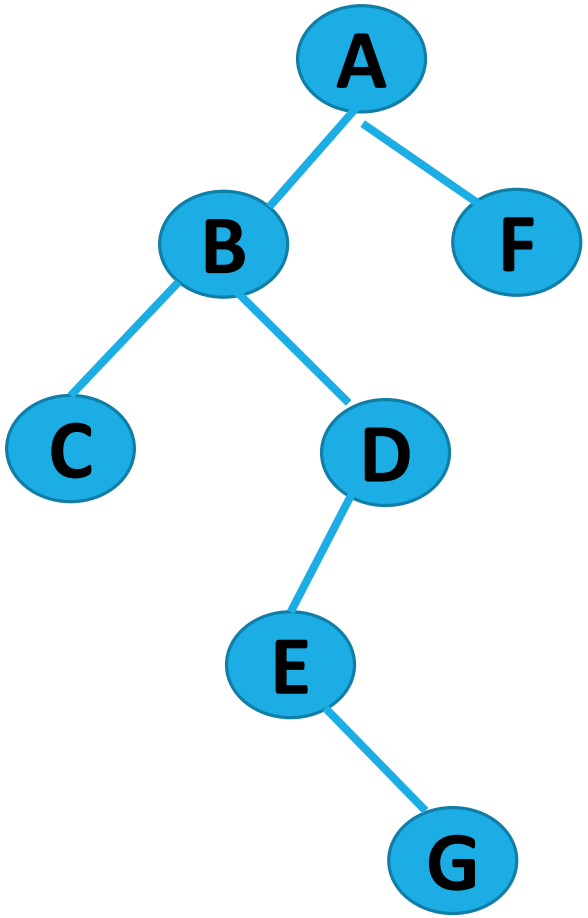
迭代2：算法4-13

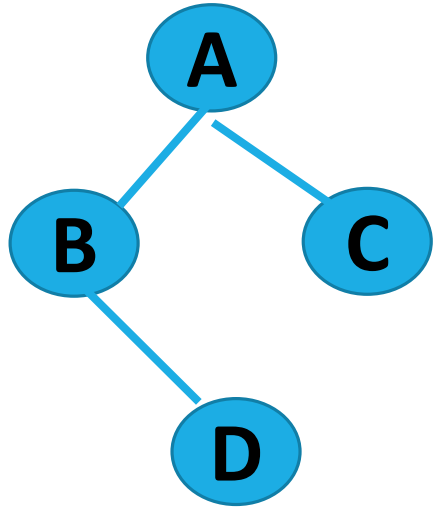
```
1  void PreOrder_NRecursion2(BinTree bt) {
2    LinkStack lstack; //定义链栈
3    BinTreeNode *p = bt;
4    lstack = SetNullStack_Link();
5  if (bt == NULL) return;
6    Push_link(lstack, bt);
7    while (!IsNullStack_link(lstack)) {
8        p = Top_link(lstack);
9        Pop_link(lstack);
10       while (p) {
11           printf("%c", p->data);
12           if (p->rightchild)
13                Push_link(lstack, p->rightch
14           p = p->leftchild;
15       }
16   }
17 }
```

迭代1：算法4-12

```
1  void PreOrder_NRecursion1(BinTree bt)
2  {
3      LinkStack lstack; //定义链栈
4      lstack = SetNullStack_Link(); //初始化栈
5      BinTreeNode *p;
6      Push_link(lstack, bt); //根结点入栈
7      while (!IsNullStack_link(lstack))
8      {
9          p = Top_link(lstack);
10         Pop_link(lstack);
11         printf("%c", p->data);
12         if (p->rightchild)
13           Push_link(lstack, p->rightchild);
14         if (p->leftchild)
15           Push_link(lstack, p->leftchild);
16     }
17 }
```

思考：采用先序迭代1和迭代2方法栈的变化

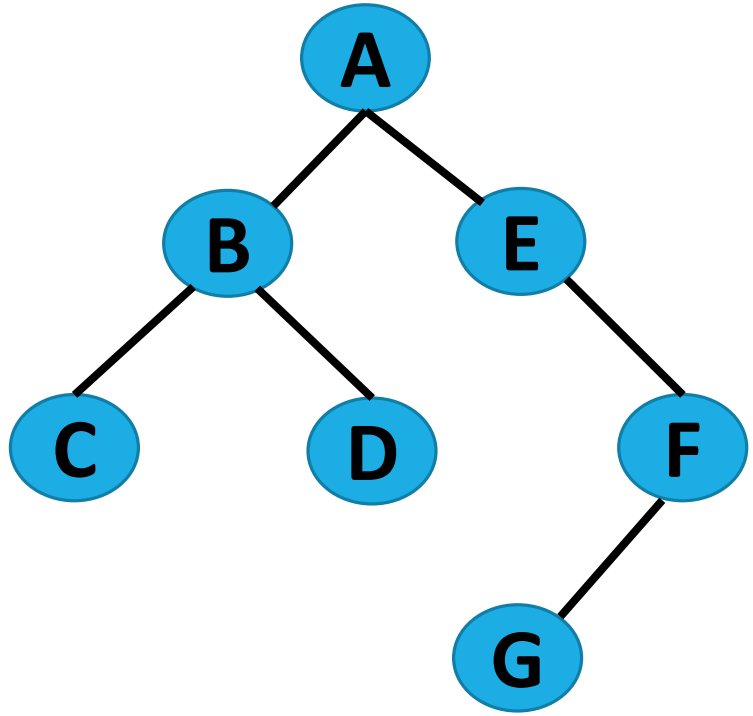




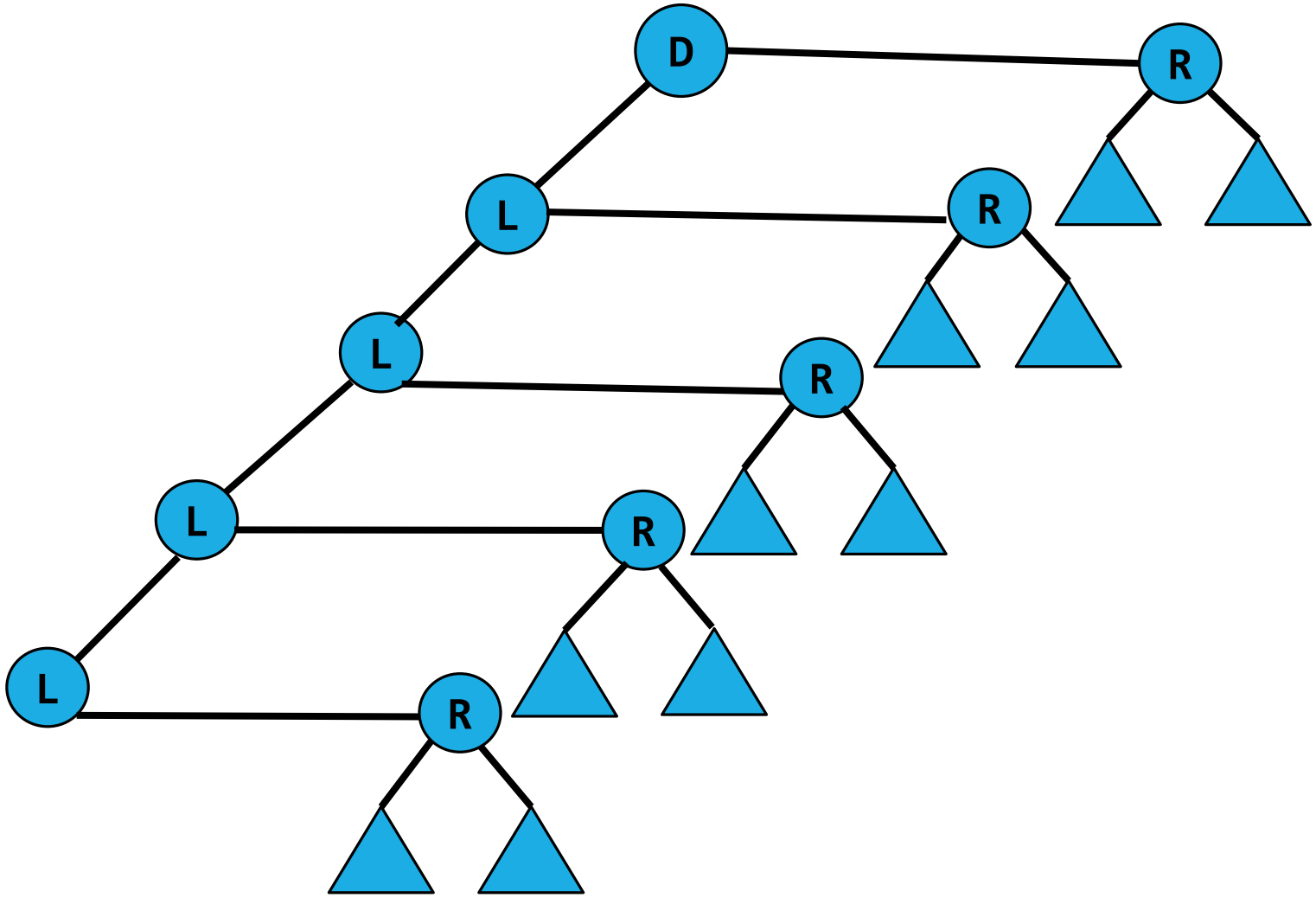
中序遍历的非递归实现

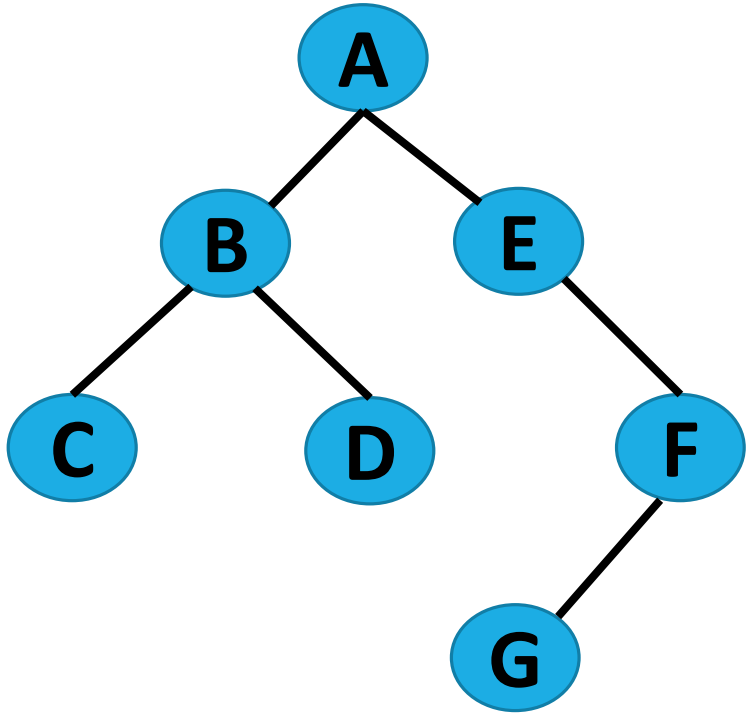
算法思路 中序遍历的过程是按照L->D->R的顺序访问结点。从根结点开始，沿着左分支一直深入将经过的每个结点入栈，直到到达一个空结点。然后出栈一个元素，并访问出栈的元素，接着进入出栈结点的右分支，此时检查出栈结点的右分支以及栈是否为空，如果都为空，算法结束，否则，重复上述过程。

a. p=bt ;
b. p不空，则p入栈lstack;
c. p=leftchild(p);
d. 重复b)~c)，直到p为空;
e. 出栈—>p，访问p;
f. p=rightchild(p);
g. 如果lstack为空同时p也为空，算法结束;
h. 重复上述过程b)~g)



中遍历的非递归实现

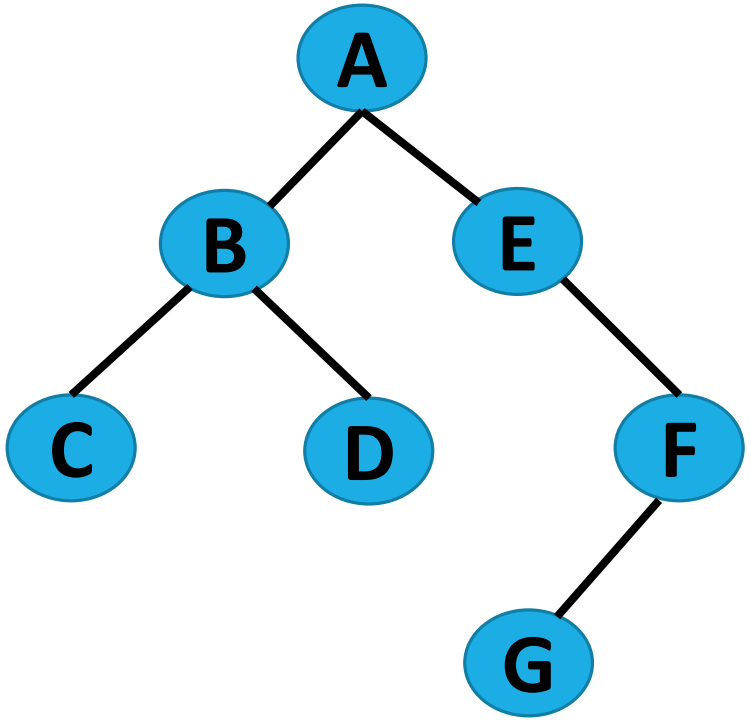




中序遍历的非递归实现

算法4-14

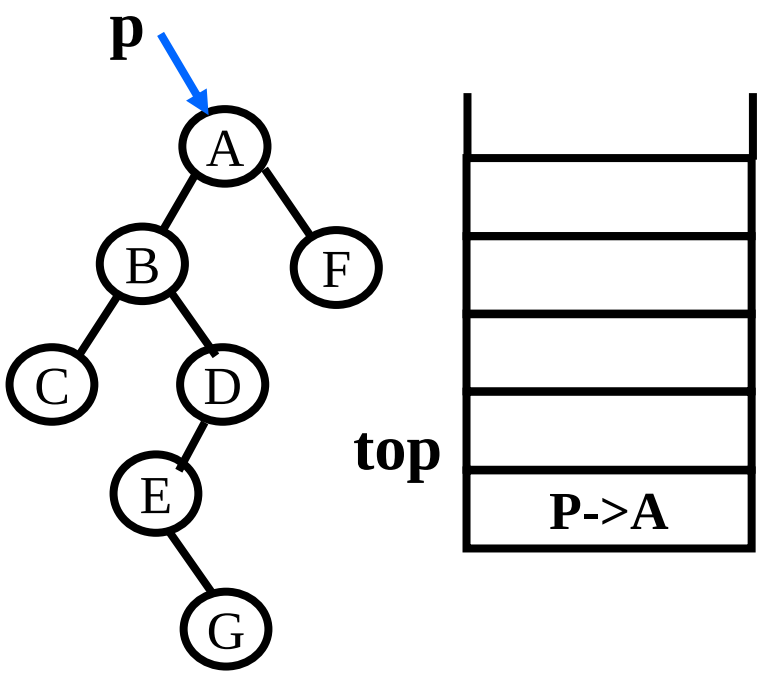
```
1  void InOrder_NRecursion1(BinTree bt)//中序遍历非递归实现
2  {
3          LinkStack lstack; //定义链栈
4          lstack = SetNullStack_Link(); //初始化栈
5          BinTree p;
6          p = bt;
7          if (p == NULL) return;
8          Push_link(lstack, bt); //根结点入栈
9          p = p->leftchild; //进入左子树
10         while (p||!IsNullStack_link(lstack))
11         {
12                 while (p != NULL)
13                 {
14                         Push_link(lstack, p);
15                         p = p->leftchild;
16                 }
17                 p = Top_link(lstack);
18                 Pop_link(lstack);
19                 printf("%c", p->data); //访问结点
20                 p = p->rightchild;//右子树非空，扫描右子树
21         }
22 }
```



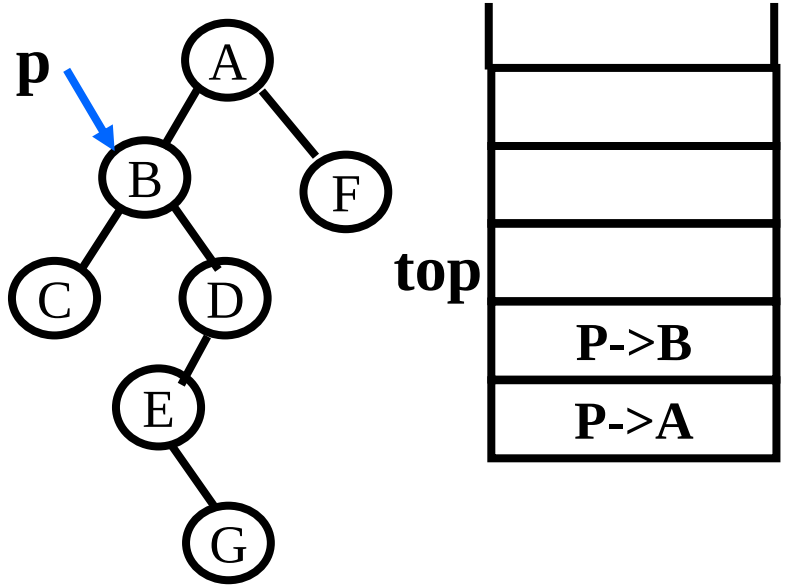
时间复杂度?

O(n)

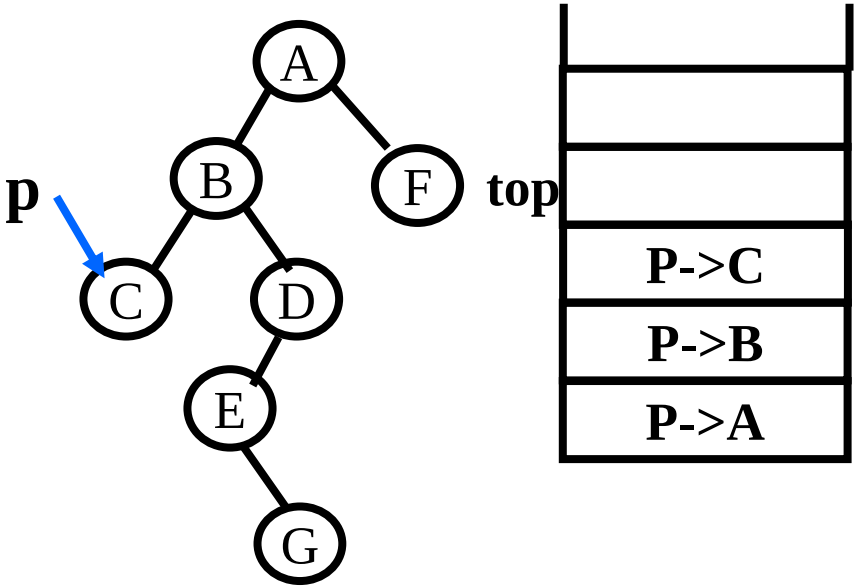
(1)


p
A
B
F
C
D
E
G
top
P->A

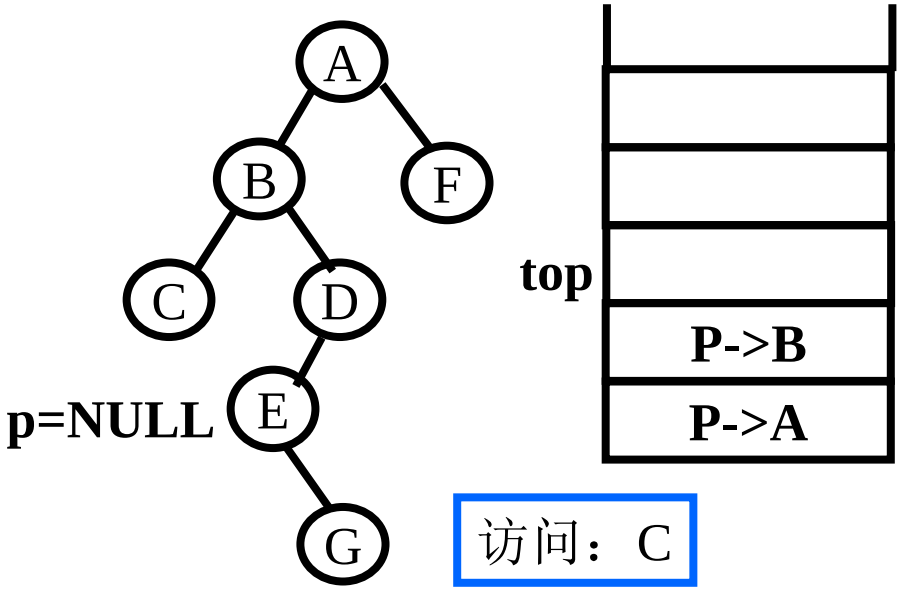
(2)


p
A
B
F
C
D
E
G
top
P->B
P->A

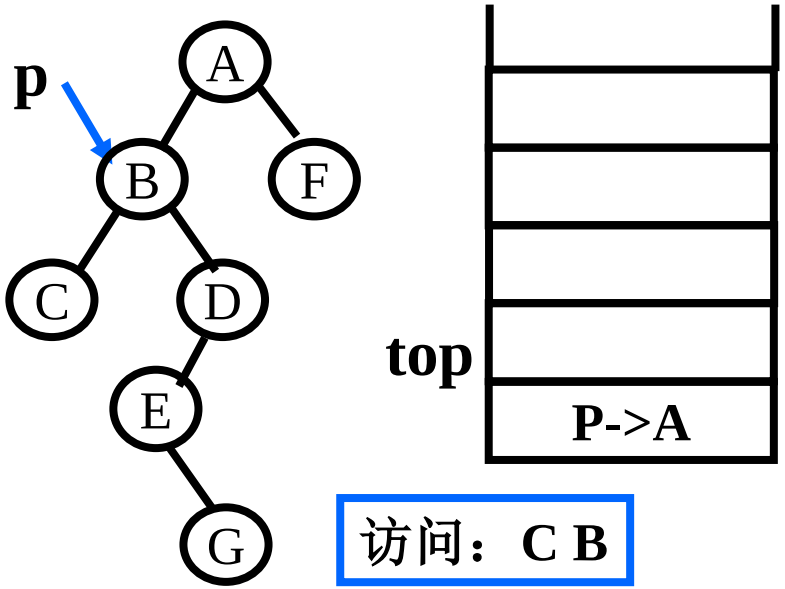
(3)


A
p
B
F
C
D
E
G
top
P->C
P->B
P->A

(4)

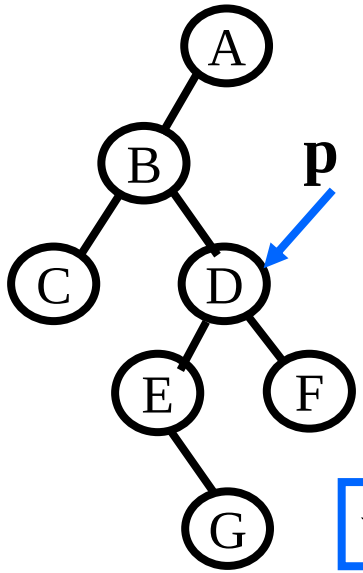

A
B
F
C
D
p=NULL
E
G
top
P->B
P->A
访问：C

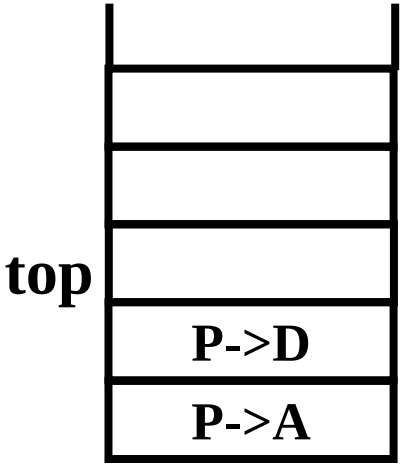
(5)



访问：C B

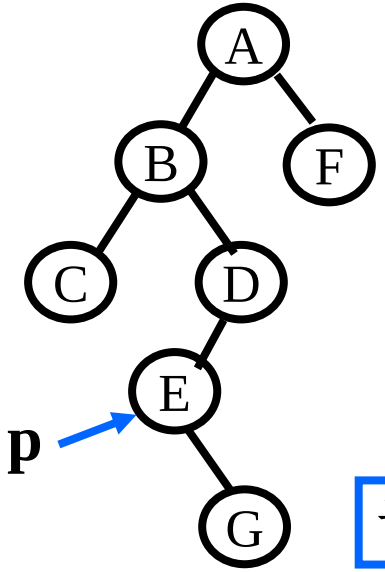
(6)

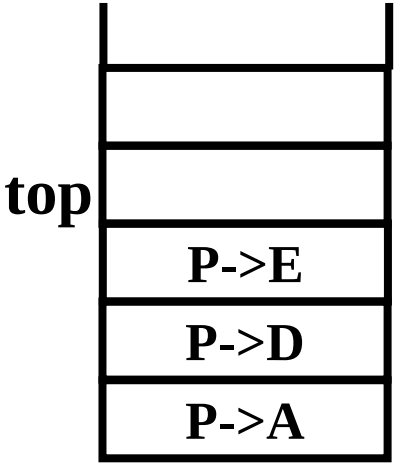




访问：C B

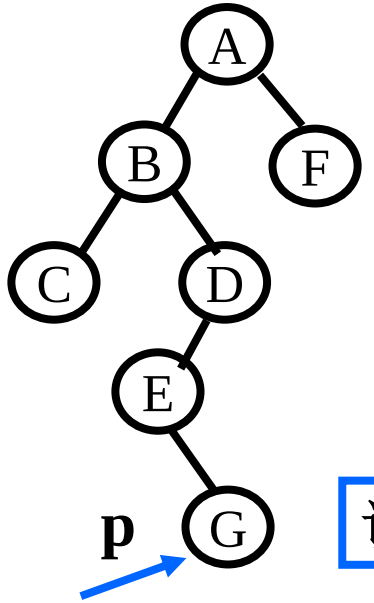
(7)

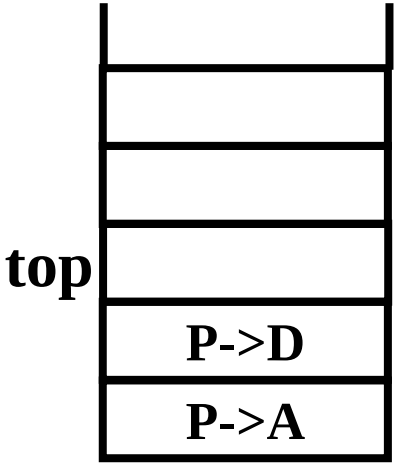




访问：C B

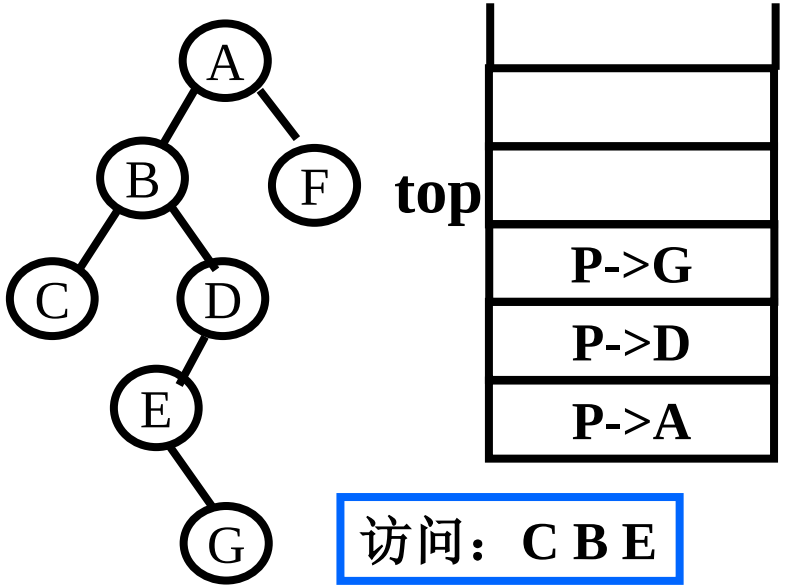
(8)





访问：C B E

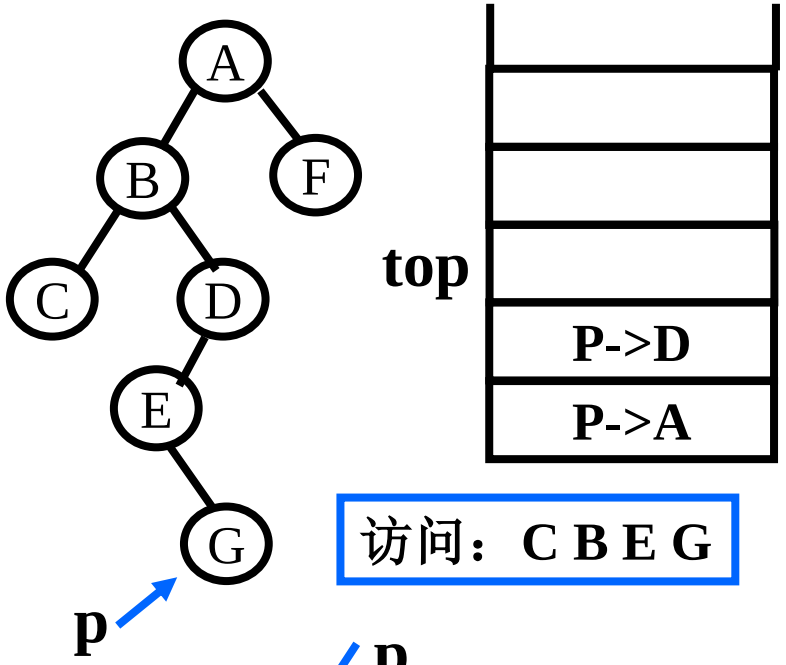
(9)



访问：C B E

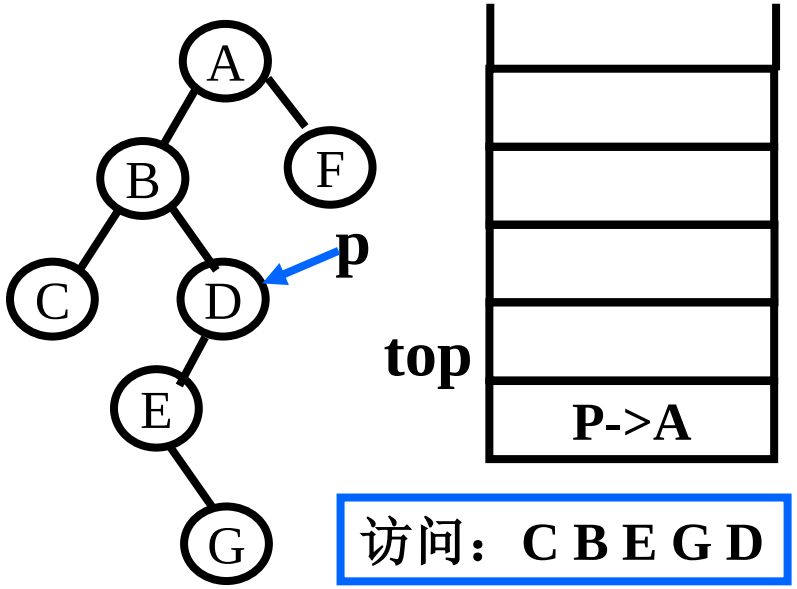
P=NULL

(10)



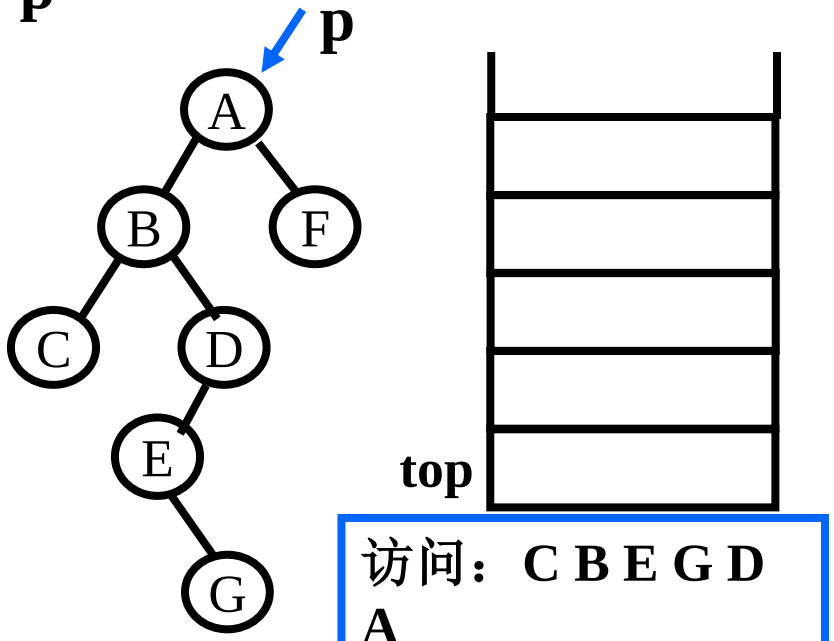
访问：C B E G

(11)



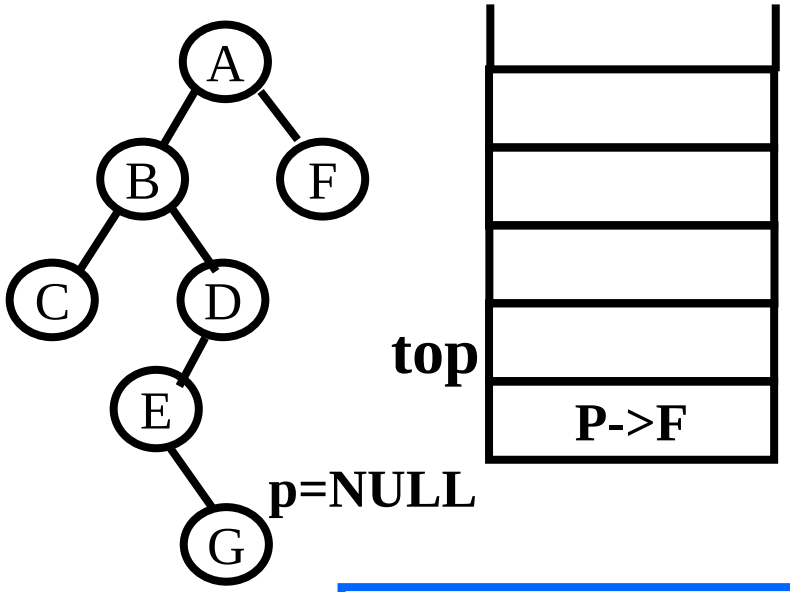
访问：C B E G D

(12)



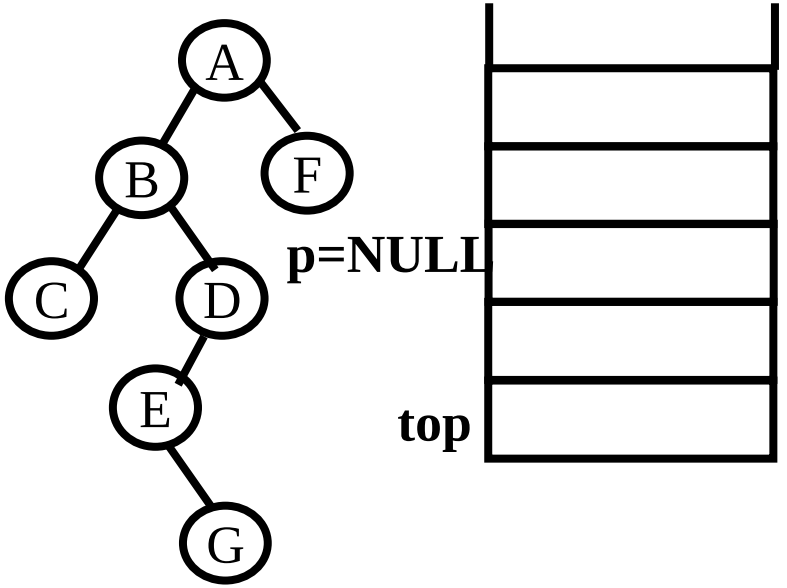
访问：C B E G D A

(13)



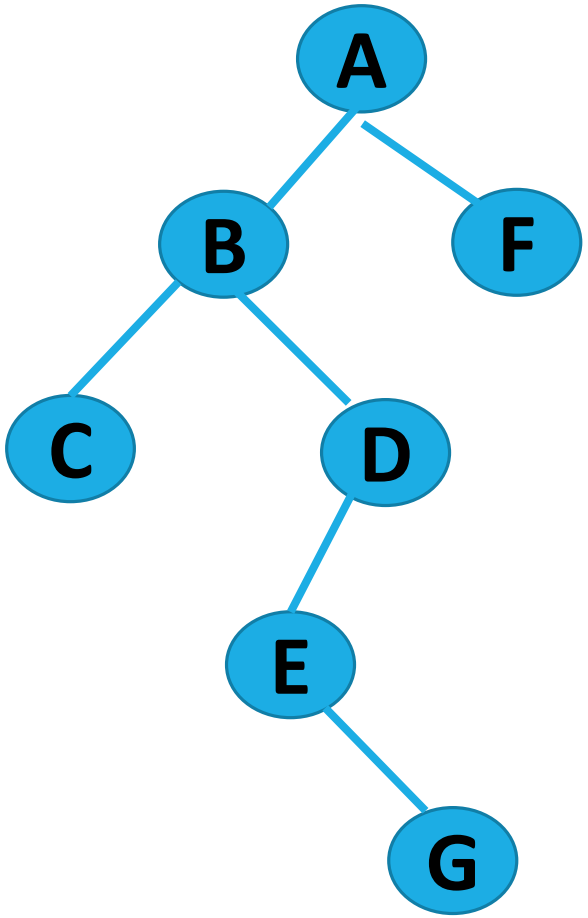
访问：C B E G D A

(14)

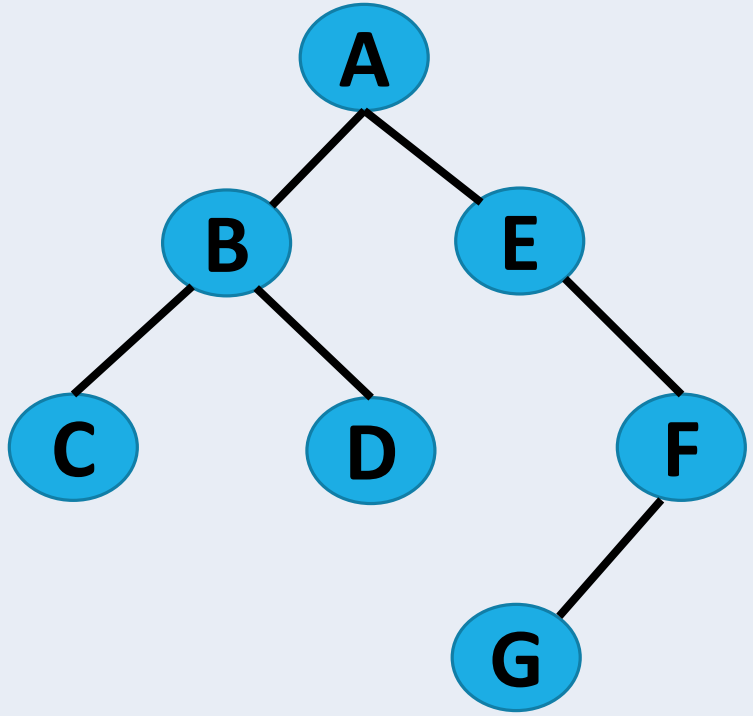


访问：C B E G D A F

思考：采用中序迭代方法栈的变化

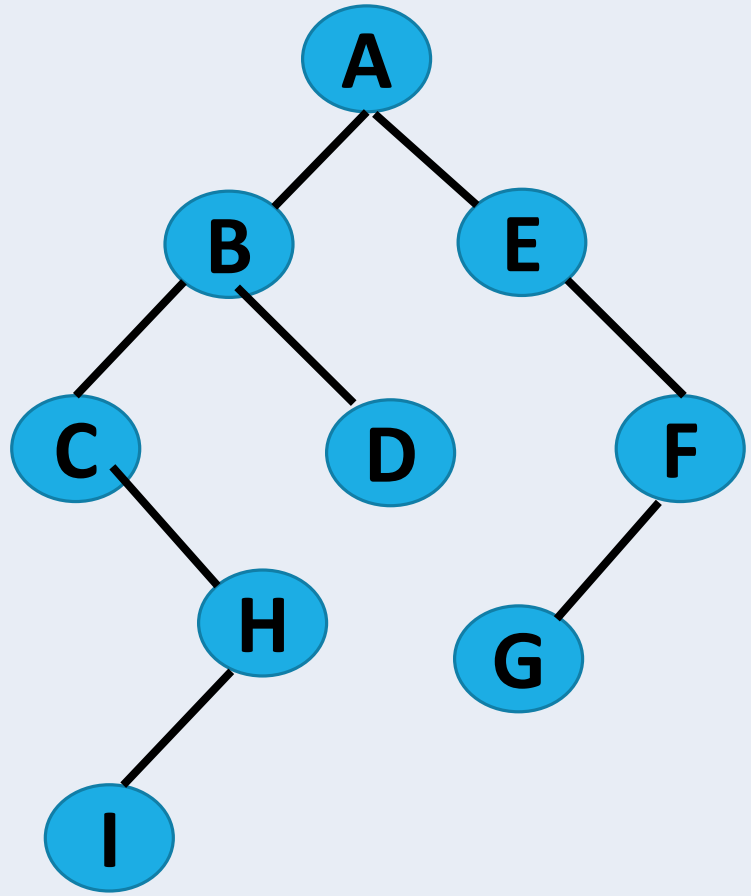


```
1  void PostOrder_NRecursion(BinTree bt){
2    BinTree p = bt;
3    LinkStack lstack; //定义链栈
4    if (bt == NULL) return;
5    lstack = SetNullStack_Link(); //初始化栈
6    while (p != NULL || !IsNullStack_link(lstack))
7    {
8       while (p != NULL) {
9             Push_link(lstack, p);
10            p = p->leftchild? p->leftchild:p->rightchild;
11     }
12     p = Top_link(lstack);
13     Pop_link(lstack);
14     printf("%c", p->data);  //访问结点
15     if(!IsNullStack_link(lstack)&&(Top_link(lstack)->leftchild==p))
16           p = (Top_link(lstack))->rightchild;  //从左子树退回，进入右子树
17     else p = NULL;          //从右子树退回，则退回上一层
18   }
19 }
```



算法4-15

```
1  void PostOrder_NRecursion(BinTree bt){
2    BinTree p = bt;
3    LinkStack lstack; //定义链栈
4    if (bt == NULL) return;
5    lstack = SetNullStack_Link(); //初始化栈
6    while (p != NULL || !IsNullStack_link(lstack))
7    {
8       while (p != NULL) {
9             Push_link(lstack, p);
10            p = p->leftchild? p->leftchild:p->rightchild;
11     }
12     p = Top_link(lstack);
13     Pop_link(lstack);
14     printf("%c", p->data);  //访问结点
15     if(!IsNullStack_link(lstack)&&(Top_link(lstack)->leftchild==p))
16           p = (Top_link(lstack))->rightchild;  //从左子树退回，进入右子树
17     else p = NULL;          //从右子树退回，则退回上一层
18   }
19 }
```



算法4-15

思考：采用后序遍历栈的变化