L7：二叉树的存储和实现

吉林大学计算机学院
谷方明
fmgu2002@sina.com

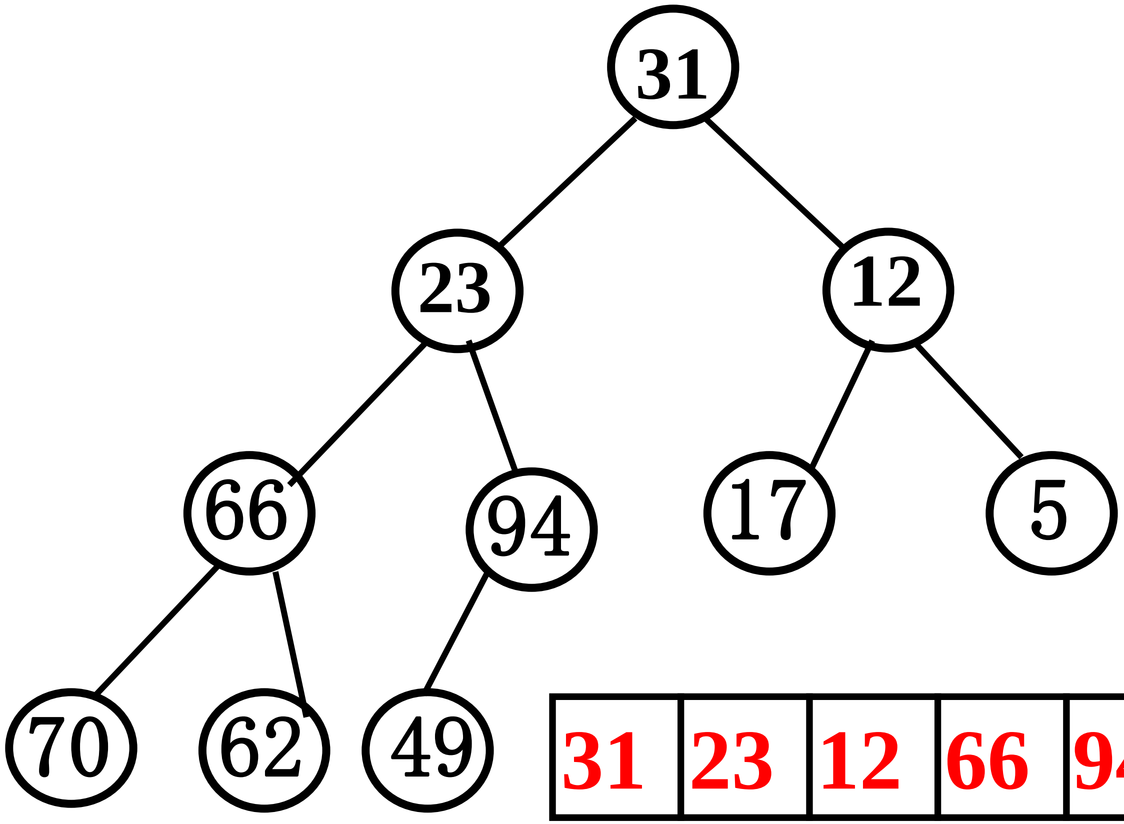
学习目标

- 掌握二叉树的顺序存储和链接存储
- 掌握二叉树的遍历操作
- 掌握二叉树的创建等操作

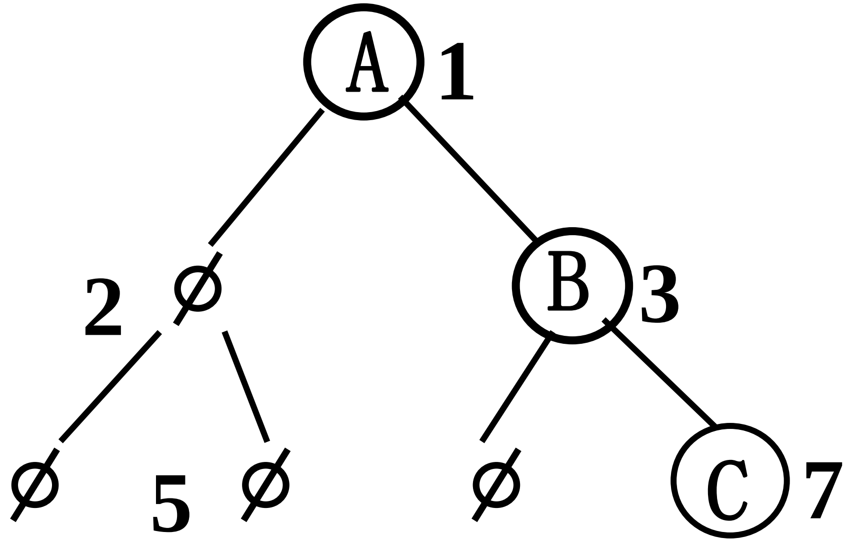
二叉树的顺序存储

- 将二叉树中所有结点存放在一块地址连续的存储空间中，同时反映出二叉树中结点间的逻辑关系。

完全二叉树的顺序存储



31	23	12	66	94	17	5	70	62	49
1	2	3	4	5	6	7	8	9	10



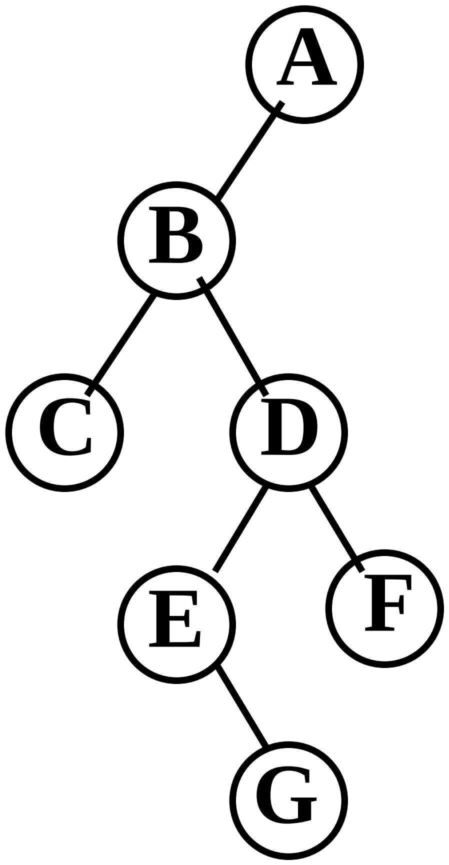
A	∅	B	∅	∅	∅	C
t[1]	t[2]	t[3]	t[4]	t[5]	t[6]	t[7]

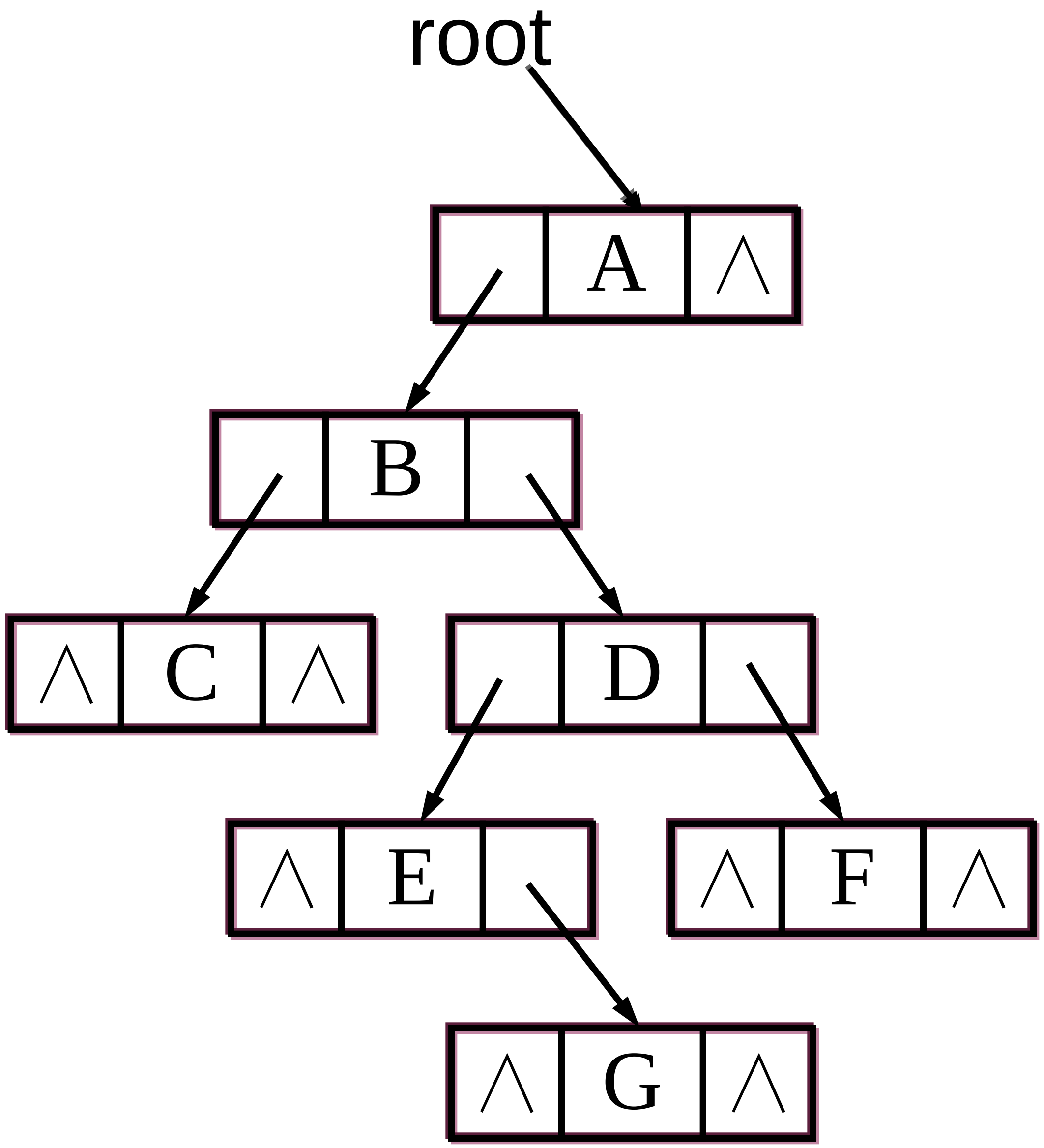
- 一般二叉树也可仿照完全二叉树那样存储。但可能会浪费很多存储空间。

二叉树的链接存储

- 二叉树诸结点被随机存放在内存空间中，结点之间的关系用指针说明。
- 二叉树的结点结构包含三个域：数据域**data**、指针域**left**和指针域**right**，其中左、右指针分别指向该结点的左、右孩子结点。

left	data	right


A
B
C
D
E
F
G

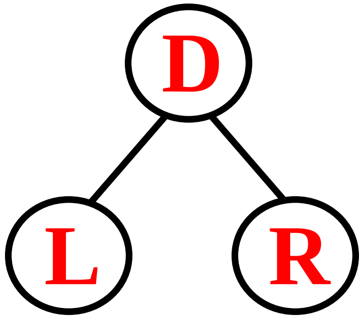

root
A
B
C
D
E
F
G

- 在二叉树的链接存储中，有一个指向根结点的指针，称为根指针。若二叉树为空，根指针为**NULL.**

- 空指针域个数：**2n - (n-1)**

二叉树的遍历

- **二叉树的遍历**：按照一定次序访问树中所有结点，且使每个结点恰好被访问一次。

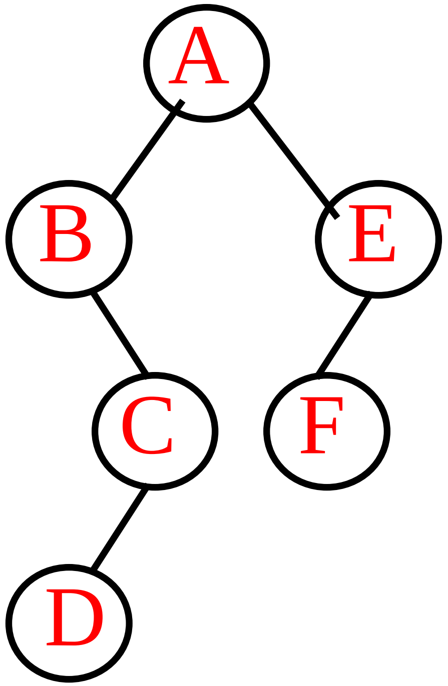


遍历方式
- 先根遍历： DLR
- 中根遍历： LDR
- 后根遍历： LRD

- 先根(中根、后根)序列：以先根(中根、后根)次序遍历二叉树 T，得到 T 之结点的一个序列

先根遍历 (前/先序遍历)

- 若二叉树为空，则返回空；
- 否则
  - ✓ 访问根结点；
  - ✓ 先根遍历左子树；
  - ✓ 先根遍历右子树。



遍历结果

ABCDEF

先根遍历算法

算法PreOrder(t)

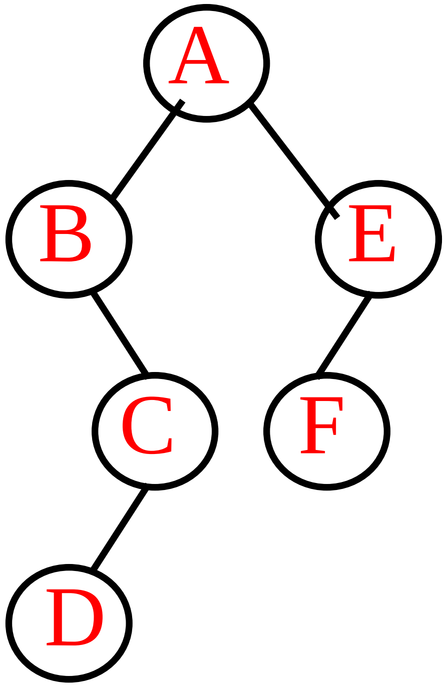
/* 先根遍历 t 指向的树*/

P1 [递归遍历]

```
 if (t != NULL) {
     printf("%d\n“, t->data );
     PreOrder(t->left);
     PreOrder(t->right);
 }
```

中根遍历(中序遍历)

- 若二叉树为空，则空操作；
- 否则
  - ✓ 中根遍历左子树；
  - ✓ 访问根结点；
  - ✓ 中根遍历右子树。



遍历结果

BDCAFE

中根遍历算法

算法InOrder(t)

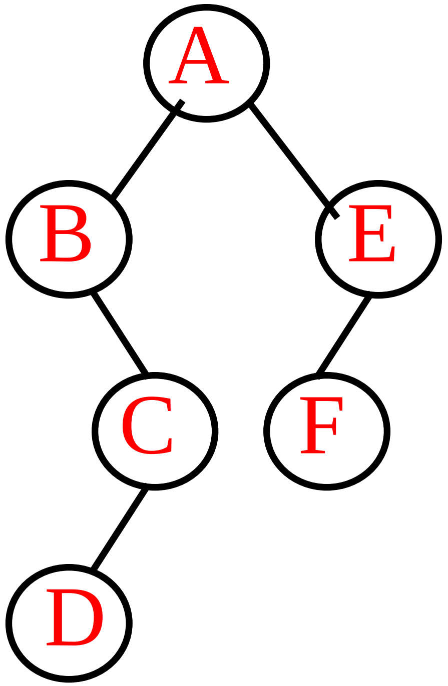
/* 中根遍历 t 指向的树*/

I1 [递归遍历]

```
  if (t != NULL) {
      InOrder(t->left);
      printf("%d\n“, t->data );
      InOrder(t->right);
 }
```

后根遍历 (后序遍历)

- 若二叉树为空，则空操作；
- 否则
  - ✓ 后根遍历左子树；
  - ✓ 后根遍历右子树；
  - ✓ 访问根结点。



遍历结果

DCBFEA

后根遍历算法

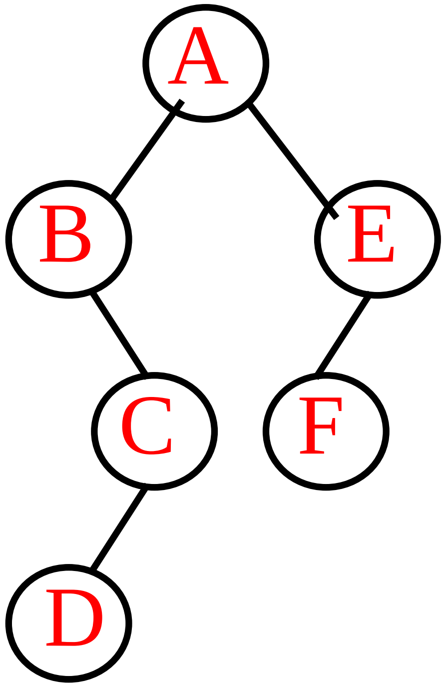
```
算法PostOrder(t)
/* 后根遍历 t 指向的树*/
I1 [递归遍历]
  if (t != NULL) {
      PostOrder(t->left);
      PostOrder(t->right);
      printf("%d\n“, t->data );
}
```

层次遍历

- 层次遍历：按层数由小到大，即从第 0 层开始逐层向下，同层中由左到右的次序访问二叉树的所有结点。

- 例：层次遍历序列为

  ABECFD



层次遍历的实现

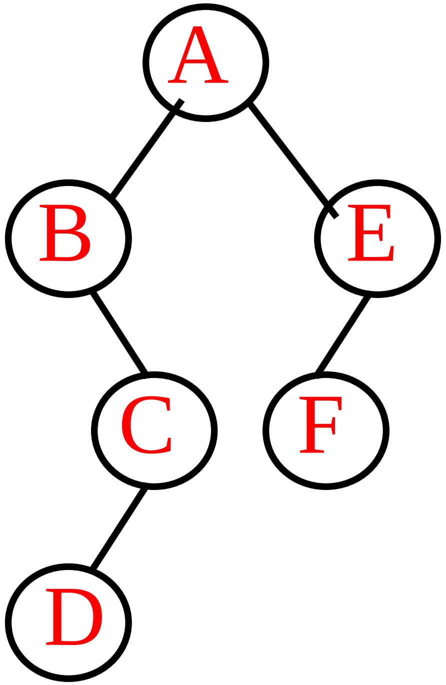
- 需要一个队列辅助

1. 根结点入队.
2. 若队不空，重复本步骤:

   取队头结点并访问;

   若其左指针不空，将其左孩子入队;

   若其右指针不空，将其右孩子入队.



算法LevelOrder (t)

LevelOrder1. [初始化]

CREATEQuene *Q* ;

***p* =*t* ; if (p!=NULL) *Q* ⇐ *p* ;**

LevelOrder 2. [层次遍历]

while (!*Q.empty()*) {

***p* ⇐ Q .**

printf("%d\n“, p->data);

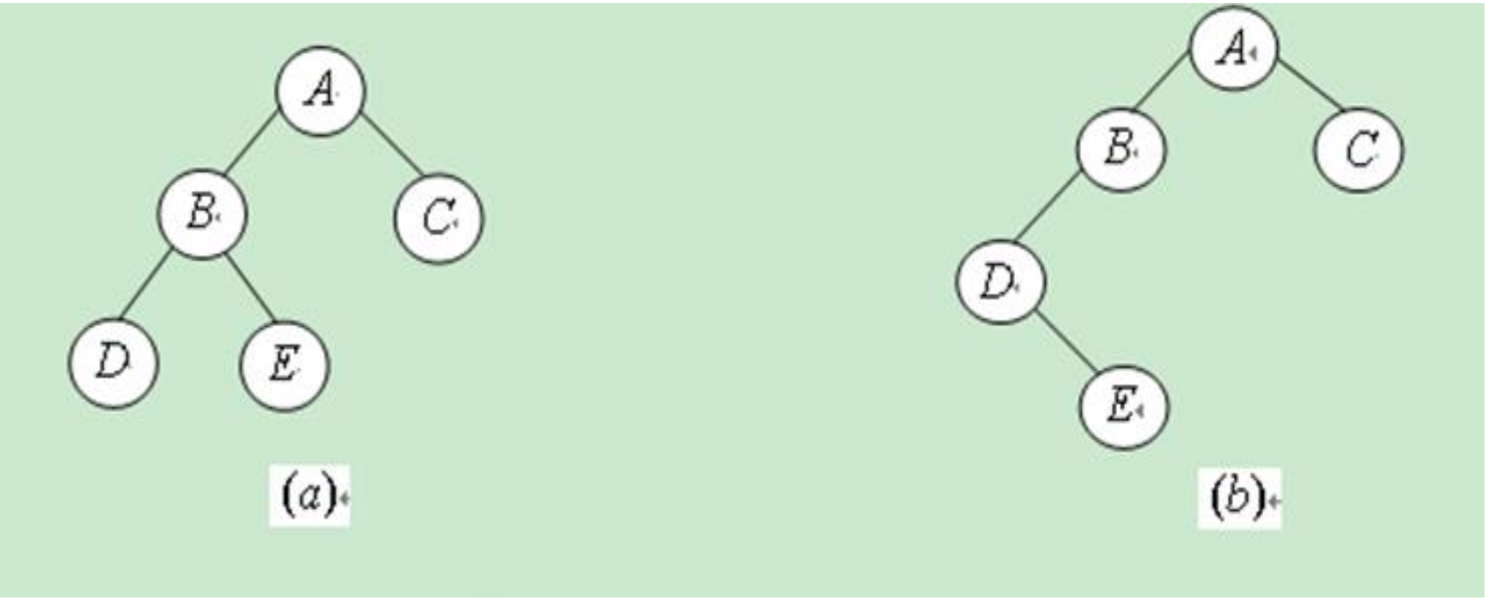
if (p-> *left* != NULL) *Q* ⇐ p -> *left* ;

if (p-> *right* != NULL) *Q* ⇐ p -> *right* ;

} ▌

创建二叉树

- 先根序列不能唯一确定二叉树之结构，两棵不同的二叉树却可能有相同的先根序列。



带空指针的扩展先根序列

- 用“#”表示空指针。图(*a*)所示二叉树的先根序列加入‘#’表示空指针位置后变为*ABD##E###C##*．图(*b*)所示二叉树之先根序列*ABDEC*，加入‘#’表示空指针位置后变为*ABD#E####C##*．

递归算法CreateBinTree

- 以包含空指针信息的扩展先根序列为输入序列，以根指针*t* 为输出参数.
- 当读入‘#’字符时，将其初始化为一个空指针；否则生成一个新结点并初始化其父结点之左、右指针.

算法CBT (tostop . t)

/* 构造以结点*t*为根的二叉树； *tostop* =‘#’ */

CBT1. [读数据]

  scanf(“%c“,&ch); /* 顺序读入序列中的一个符号 */

CBT2. [*ch* = *tostop*?]

  if (*ch* == *tostop*) { *t* =Λ ; return; }

  t ⇐ AVAIL ; t->*data* = *ch* ;

CBT3. [构造左子树]

  CBT (*tostop* , *t* ->*left*);

CBT4. [构造右子树]

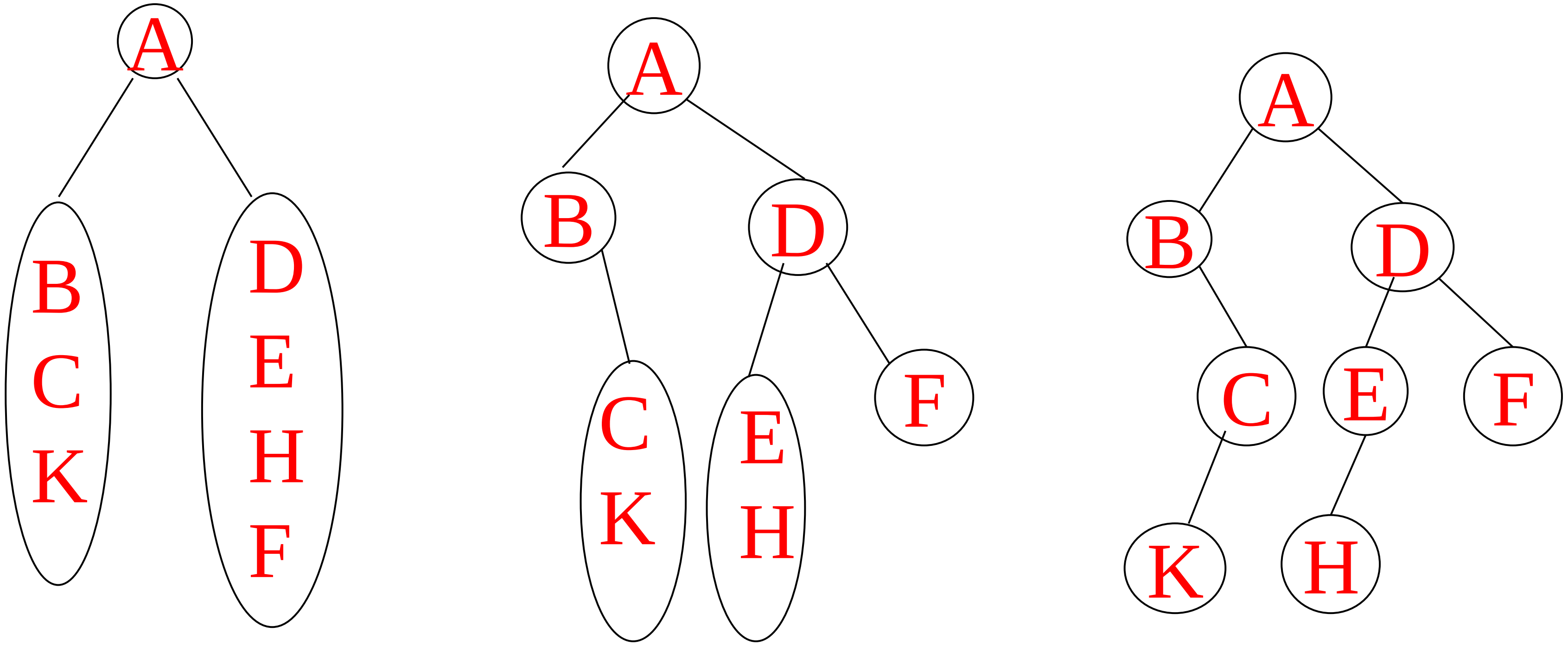
  CBT (*tostop*, *t* -> *right*); ▌

其它建树方法

- 先根序列和中根序列可以唯一确定一棵二叉树。

[例] 先根序列　A B C K D E H F

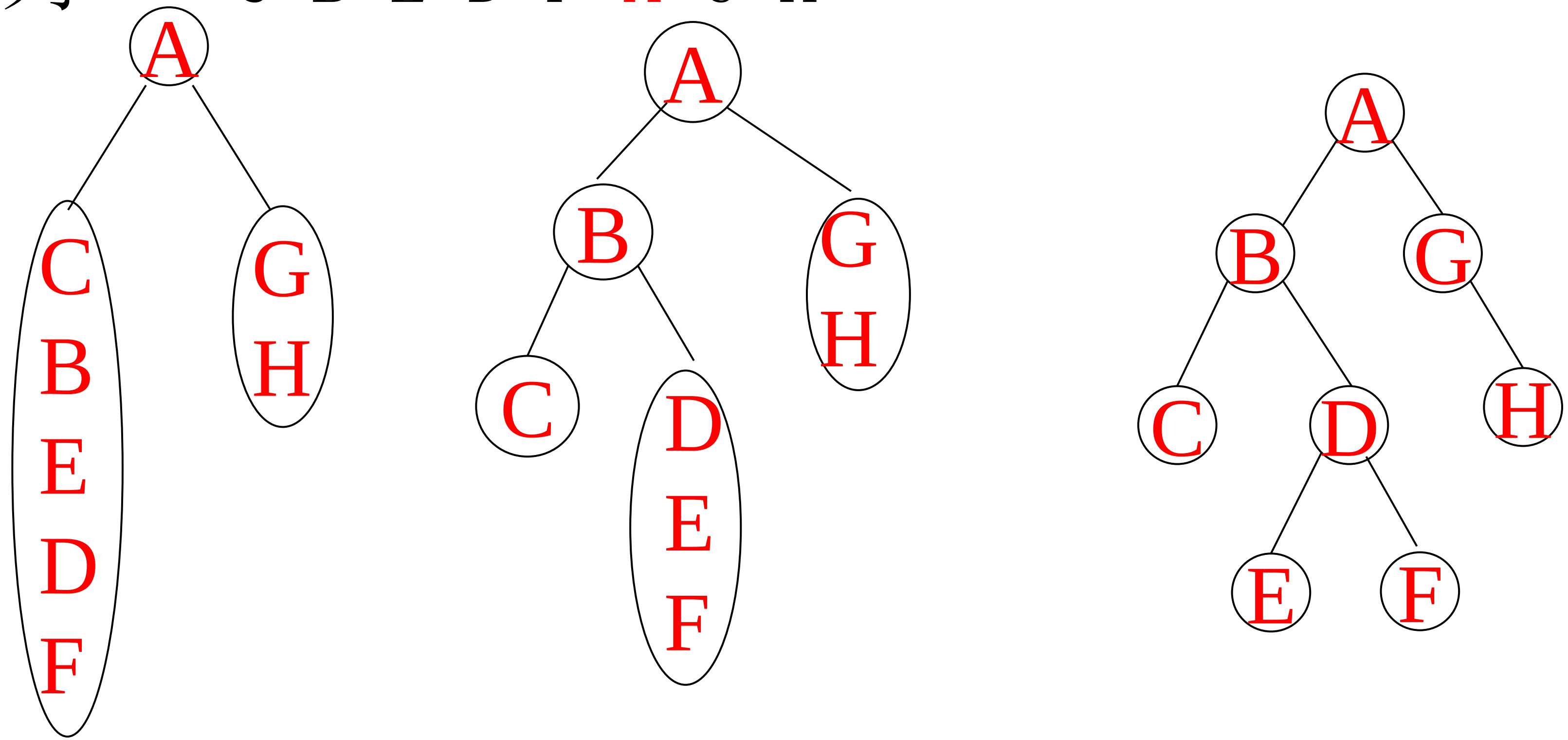
中根序列　B K C A H E D F



- 后根序列和中根序列可以唯一确定一棵二叉树。

［例］ 后根序列　　C E F D B H G A

　　　 中根序列　　C B E D F A G H



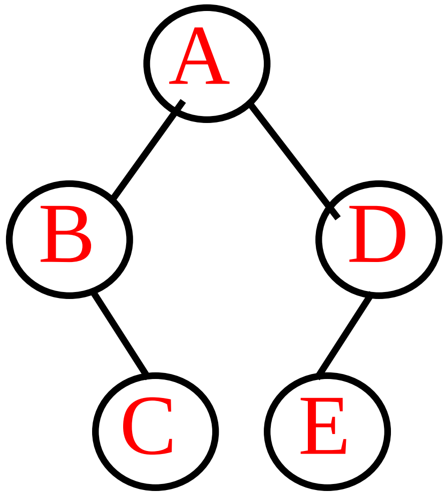
课后思考

1. 给定一棵二叉树 T 的先根序列和后根序列，那么能否由此确定出 T 之结构?

2. 尝试设计其它表示二叉树的方法。

复制二叉树

- 可以按先根遍历、中根遍历或后根遍历的方式复制二叉树。以后根遍历为例进行复制。
- 复制过程：先复制子结点，再复制父结点，将父结点与子结点连接起来。



算法 CopyTree (t . p)

CopyTree1. [递归出口]

if (*t* == NULL) { p = NULL; return; }

CopyTree2. [复制左右子树]

CopyTree(t -> *left* , *newlptr*);

CopyTree(t -> right , *newrptr);*

CopyTree4. [复制根结点]

p **⇐AVAIL.**

p -> *data* = t -> *data* ; p -> *left* = *newlptr ;* p -> *right* = *newrptr;* ▌

搜索父结点

```
算法Father ( t, p . q )
 F1 [判断t是否存在及p是否为根结点]
     if (t==NULL || p==NULL ||  p==t) return q = NULL;
 F2 [若t为所求]
     if ( t->Left(t)==p ||  t->Right(t)==p ) return q = t;
F3 [递归调用]
      Father( t->left, p , qL);
      if (qL!=NULL)  return q=qL ;
      Father(t-> right, p . qR);
      return q=qR;
```

三叉链

- 另一种常用的结点结构包括三个指针域，**parent**域中指针指向其父结点

left	**data**	**parent**	**right**

搜索数据

算法**Find**(*t, item* . *q*)

Find1. [判断*t*是否为空或为所求]

```
    if (t==NULL)  return q = NULL;
    if (t->data == item) return q = t ;
```

Find2. [递归]

```
    Find ( t->left, item , p );
    if( p!=NULL )  return q = p;
    Find ( t->right, item , p );
    return q = p;  ▌
```

插入结点作为某结点的左儿子

算法**InsertLeft** (*t*, *item*, *p*)

I1. [特判]

if (t==*NULL*) return ;

I2. [结点p]

```
Find(t, item, q);
if(q==NULL) return;
```

I3. [插入结点]

```
p->left = q->left ;
q->left = p;
```

▌

释放二叉树

算法Del(*p*)

/* 删除结点*p*及其左右子树 */

Del1. [递归删除]

```
   if( p!=NULL ) {
       Del( p->left  );
       Del( p->right  );
       AVAIL ⇐ p ;
  } ▌
```

删除给定结点及其左右子树

算法**DST** (*t*)

DST1. [特判]

```
if ( t==NULL ) return ;
if ( t==root ) { Del(t) ;  root =NULL ; return ;}
```

DST2. [找*t*的父结点*q*]

```
p=t ;  Father(root, p, q).
```

DST3. [修改*q*的指针域]

```
if ( q!=NULL &&    q->left==p) q-> left=NULL;
if ( q!=NULL &&    q-> right ==p) q-> right=NULL;
```

DST4. [删除*p*及其子树]

```
Del(p);
```

非递归的中根遍历算法

算法NIO(*t*)

NIO1. [初始化]

CREATEStack(*S*) ; *p* = *t* .

NIO2. [入栈]

while(*p* !=NULL) { *S* .push(*p*) ; *p* = *p*->*Left* ; }

NIO3. [栈为空?]

if (S.empty()) return; else *p* = *S.pop* ( ) ;

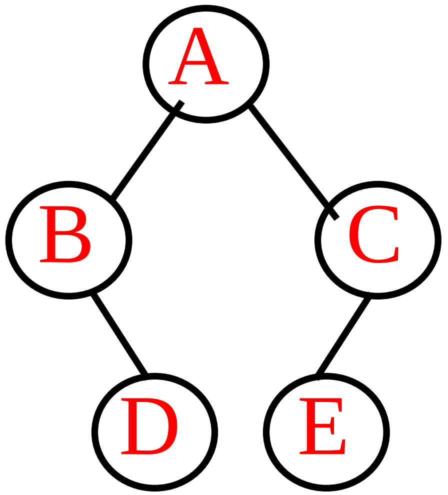
NIO4. [访问*p*，更新*p*]

printf(”%c ”, *p*->*Data*) ; *p* = *p*-> *Right* ;

NIO5. [返回]

goto NIO2 ; //编程时用循环代替goto ■

运行及证明



- 定理5.1：正确性证明

设算法NIO从步骤NIO2开始，p指向一棵有n个结点之二叉树T*的根，此时栈S中有S[1]，S[2]，…，S[m]，m≥0，则步骤NIO2至NIO5将以中根序遍历T*，并最后到达步骤NIO3，同时栈S也恢复到原来值。

非递归的后根遍历算法

- 先根和中根遍历的非递归算法，一个结点仅进栈出栈一次，我们能够判断其输出语句的位置，分别为结点进栈前及出栈后。

- 而后根遍历输出结点的位置应为处理完右子树之后，如果每个结点还是进栈、出栈一次，则无法确定何时输出结点，即其左右子树是否已处理完。

工作栈结点

结点	结点状态 i

结点所处状态i = 0, 1或2：

0：结点及左右子树均未被访问；

1：遍历左子树；

2：遍历右子树。

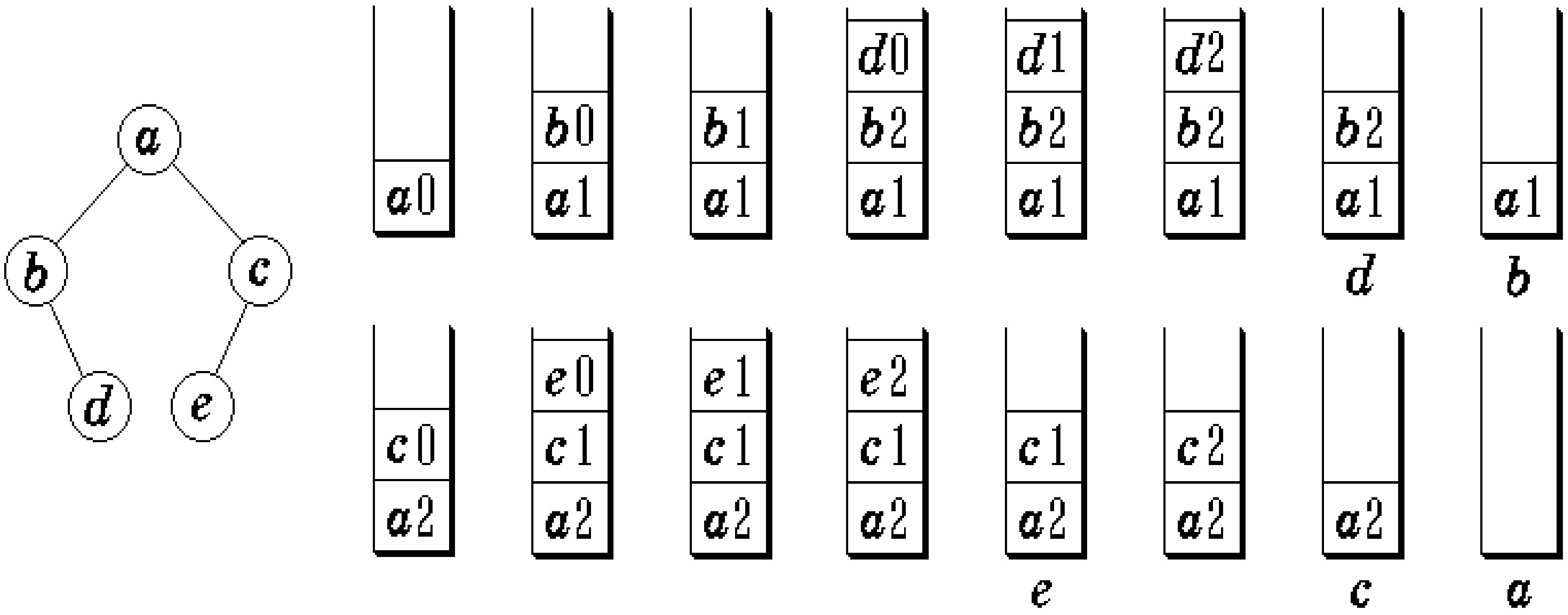
- 树中任一结点q都需进栈三次，出栈三次。第一次出栈是为遍历结点q的左子树，第二次出栈是为遍历结点q的右子树，第三次出栈是为访问结点q .

算法思想

1) 将(根结点,0)压入堆栈。

2) 弹栈，对出栈元素(*p*, *i*)中标号*i*进行判断，

- 若 *i*=0，则将(*p*,1)压入堆栈；若结点*p*的左指针不为空，则将(*Left*(*p*), 0) 压入堆栈，准备遍历其左子树.
- 若 *i*=1，此时已遍历完结点*p*的左子树，则将(*p*,2)压入堆栈；若右指针不为空，则将(*Right*(*p*), 0)压入堆栈，准备遍历其右子树.
- 若 *i*=2，此时已遍历完结点*p*的右子树，访问结点*p*.


a
b
c
d
e
a0
b0
a1
b1
a1
d0
b2
a1
d1
b2
a1
d2
b2
a1
b2
a1
d
a1
b
c0
a2
e0
c1
a2
e1
c1
a2
e2
c1
a2
c1
a2
e
c2
a2
a2
c
a

算法NPostOrder(t)

NPO1[建立堆栈]

```
    CREATStack(S);   S ⇐(t,0);
NPO3[利用栈实现递归]
  while( ! s.empty()) {
      (P,i) ⇐S;
      if ( i==0 ) { S ⇐(P,1);
          if ( p->Left != NULL )   S ⇐(p->Left,0) ; }
      if ( i==1 ) { S ⇐(P,2);
          if ( p->Right != NULL )   S ⇐(p->Right,0) ; }
      if ( i==2 )  printf("%c ", p->Data) ;
  }▌
```

课后思考

- 非递归的先根遍历算法？
- 非递归的后根遍历的其它实现方案？

总结

- 二叉树的存储结构（顺序、链接）
- 二叉树的操作（遍历、创建、……）
- 遍历的迭代算法（中根、后根、先根）

第5章 任务

- 慕课
  - ✓ 在线学习/预习 第 5 章 视频
  - ✓ 在线学习/预习 7.4.2 堆排序

- 作业
  - ✓ P162：5-2，5-4， 5-5，5-6，
    5-8，5-10，5-12，5-14
  - ✓ 在线提交