数据结构

第六讲 二叉树

孙猛

http://www.math.pku.edu.cn/teachers/sunm

2017年10月16日

被猜价格	第一次	第二次	第三次	第四次	第五次	第六次	第七次
39	50	25	37	43	40	38	39
82	50	75	88	82			
99	50	75	88	94	97	99	

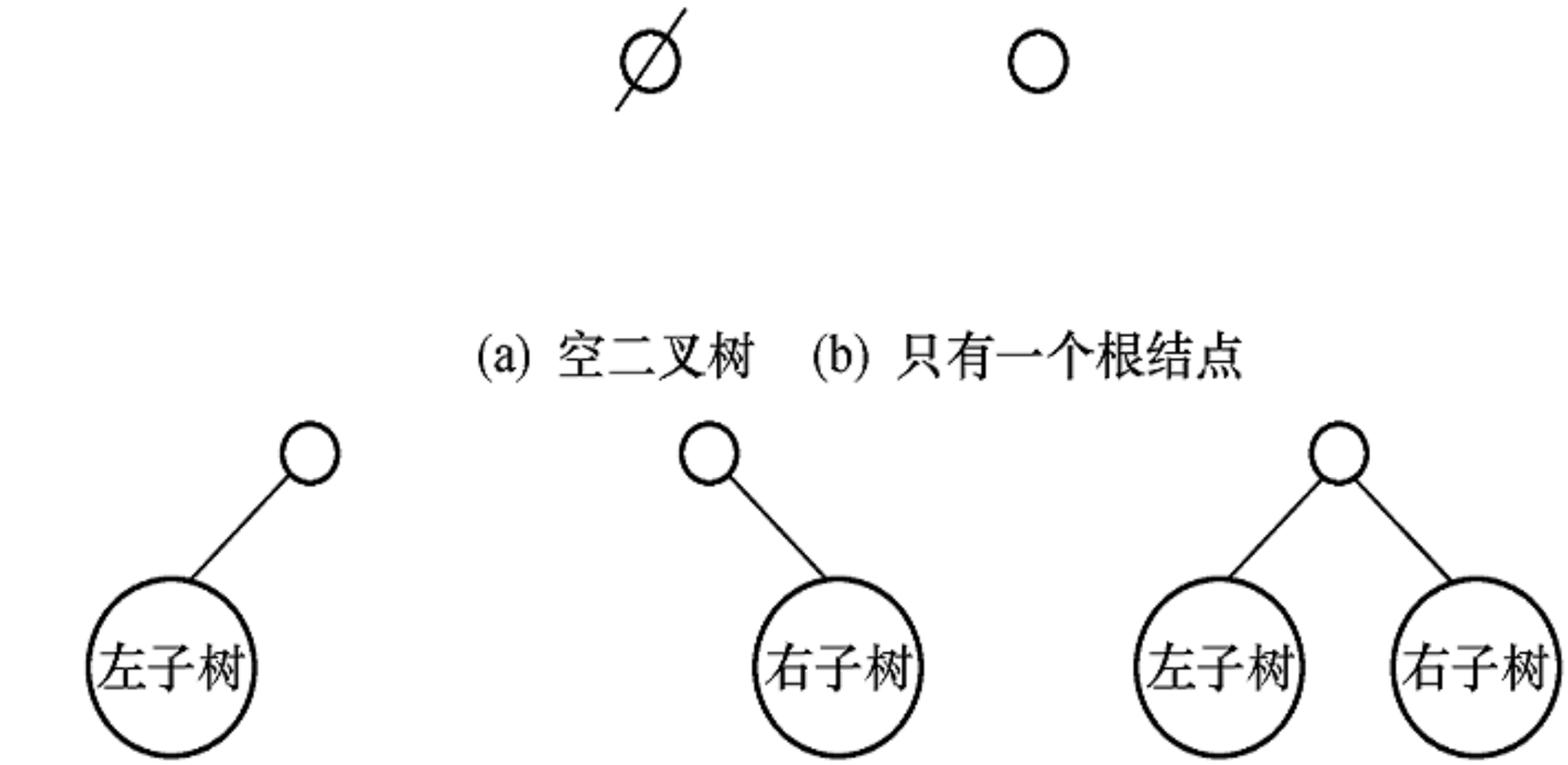
课程内容

- 二叉树及其抽象数据类型
- 二叉树的周游
- 二叉树的实现

二叉树及其抽象数据类型

- 基本概念
  - 二叉树可以定义为结点的有限集合，这个集合或者为空集，或者由一个根及两棵不相交的分别称作这个根的左子树和右子树的二叉树组成。
- 二叉树的定义是个递归定义。
- 二叉树可以是个空集合，这时的二叉树称为空二叉树。
- 二叉树也可以是只有一个结点的集合，这个结点只能是根；它的左子树和右子树均是空二叉树。

二叉树的五种基本形态



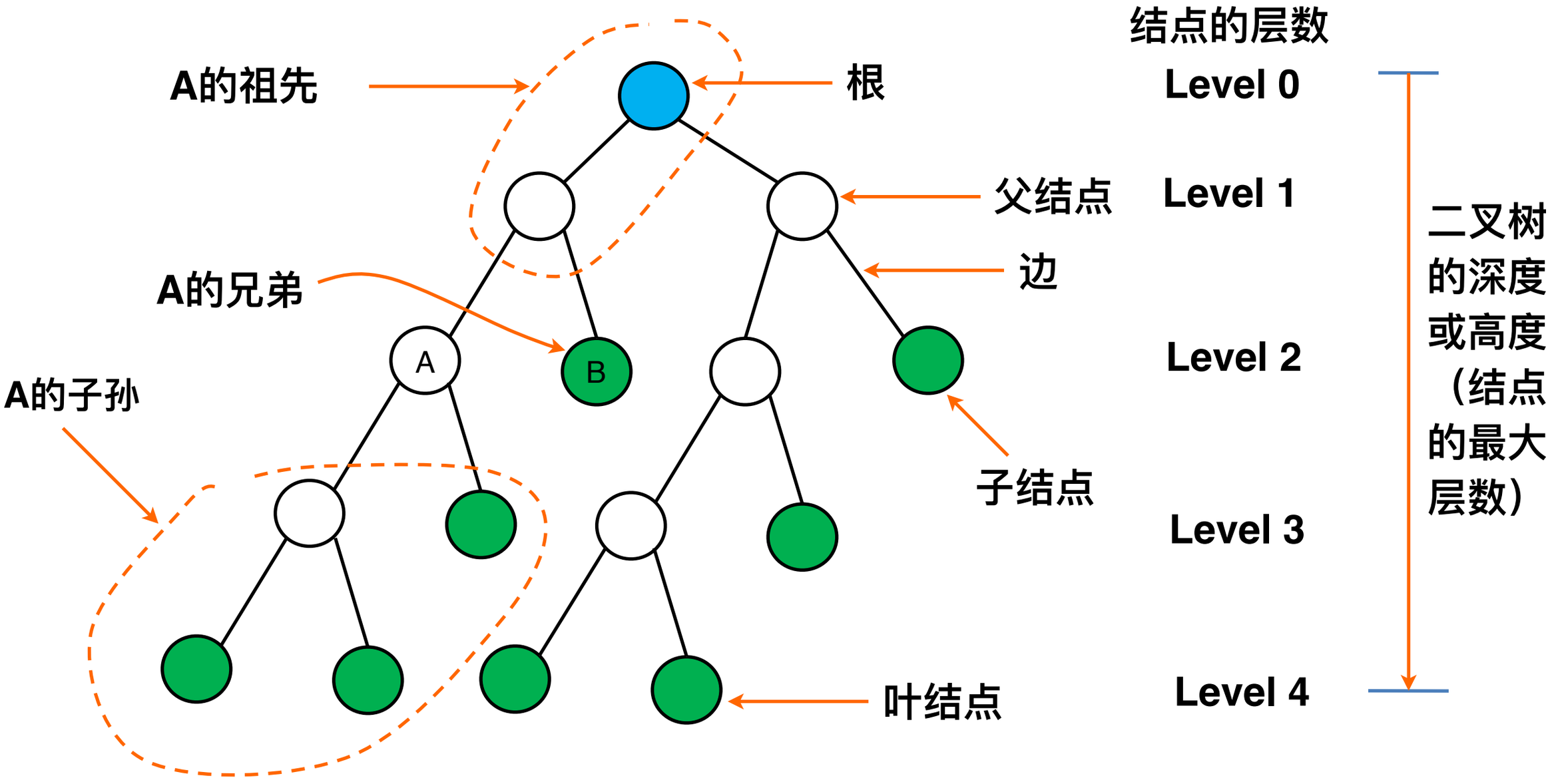
(a) 空二叉树　(b) 只有一个根结点

(c) 有根结点和非空左子树

(d) 有根结点和非空右子树

(e) 有根结点并且左、右子树均为非空

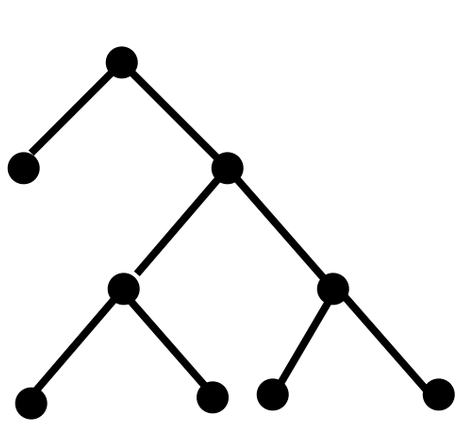
二叉树的基本术语


结点的层数
A的祖先
根
Level 0
父结点
Level 1
A的兄弟
边
A
B
Level 2
A的子孙
子结点
Level 3
叶结点
Level 4
二叉树的深度或高度（结点的最大层数）

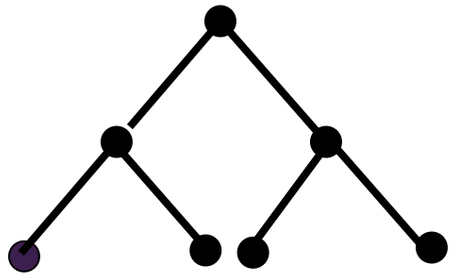
满二叉树和完全二叉树

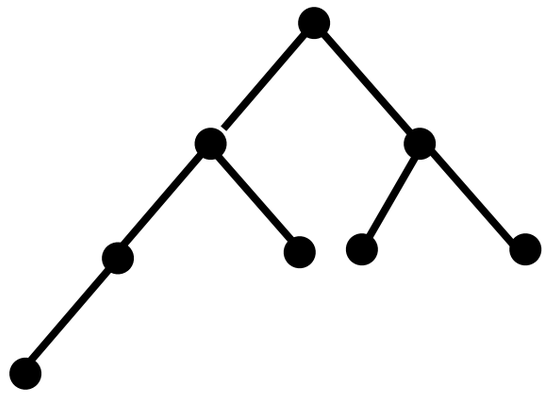
- **满二叉树**：如果一棵二叉树的任何结点或者是树叶，或有两棵非空子树，则此二叉树称作满二叉树。
- **完全二叉树**：如果一棵二叉树至多只有最下面的两层结点度数可以小于2，其余各层结点度数都必须为2，并且最下面一层的结点都集中在该层最左边的若干位置上，则此二叉树称为完全二叉树。
- 完全二叉树不一定是满二叉树。

满二叉树和完全二叉树



满二叉树

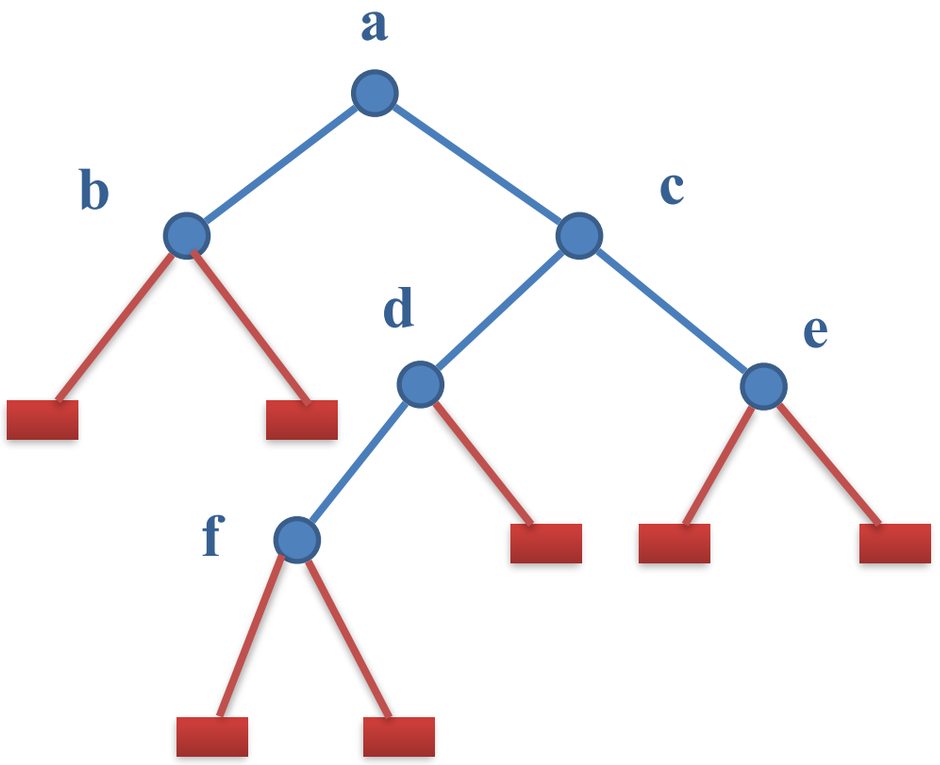




完全二叉树

扩充二叉树和外部结点

- **扩充二叉树**：把原二叉树的结点都变为度数为2的分支结点，也就是说，如果原结点的度数为2，则不变，度数为1，则增加一个分支，度数为0（树叶），则增加两个分支。
- 新增加的结点都用小方框表示，称为**外部结点**，树中原有的结点称为**内部结点**。
- 特别对空二叉树扩充得到只有一个外部结点的扩充二叉树。



外部路径长度和内部路径长度

- 外部路径长度E定义为在扩充的二叉树中从根到每个外部结点的路径长度之和。
- 内部路径长度I定义为在扩充的二叉树中从根到每个内部结点的路径长度之和。

主要性质

- **性质1：在二叉树的 i 层上至多有2^i个结点（$i \geq 0$）。**

- **性质2：高度为k的二叉树中最多有2^{k+1}-1个结点（$k \geq 0$）。**

- **性质3：对于任何一棵二叉树，如果叶结点个数为n_0，度为2的结点个数为n_2，则有$n_0 = n_2 + 1$。**

- **性质4：具有n个结点的完全二叉树的深度k为$\lfloor log_2 n \rfloor$。**

主要性质

- 性质5：对于具有n个结点的完全二叉树，如果按照从上（根结点）到下（叶结点）和从左到右的顺序对二叉树中的所有结点从0开始到$n-1$进行编号，则对于任意的下标为i的结点，有：

① 如果 $i=0$，则它是根结点，没有父结点；如果 $i>0$，则它的父结点的下标为$\lfloor\frac{i-1}{2}\rfloor$；

② 如果$2i+1\leq n\text{-}1$，则下标为 i 的结点的左子结点的下标为 $2i+1$；否则下标为 i 的结点没有左子结点；

③ 如果$2i+2\leq n\text{-}1$，则下标为 i 的结点的右子结点的下标为 $2i+2$；否则下标为 i 的结点没有右子结点。

主要性质

- **性质6：在满二叉树中，叶结点的个数比分支结点个数多1。**
- **性质7：在扩充二叉树中，外部结点的个数比内部结点的个数多1。**
- **性质8：对任意扩充二叉树，外部路径长度E和内部路径长度I之间满足关系：E ＝ I ＋ 2n，其中n是内部结点个数。**

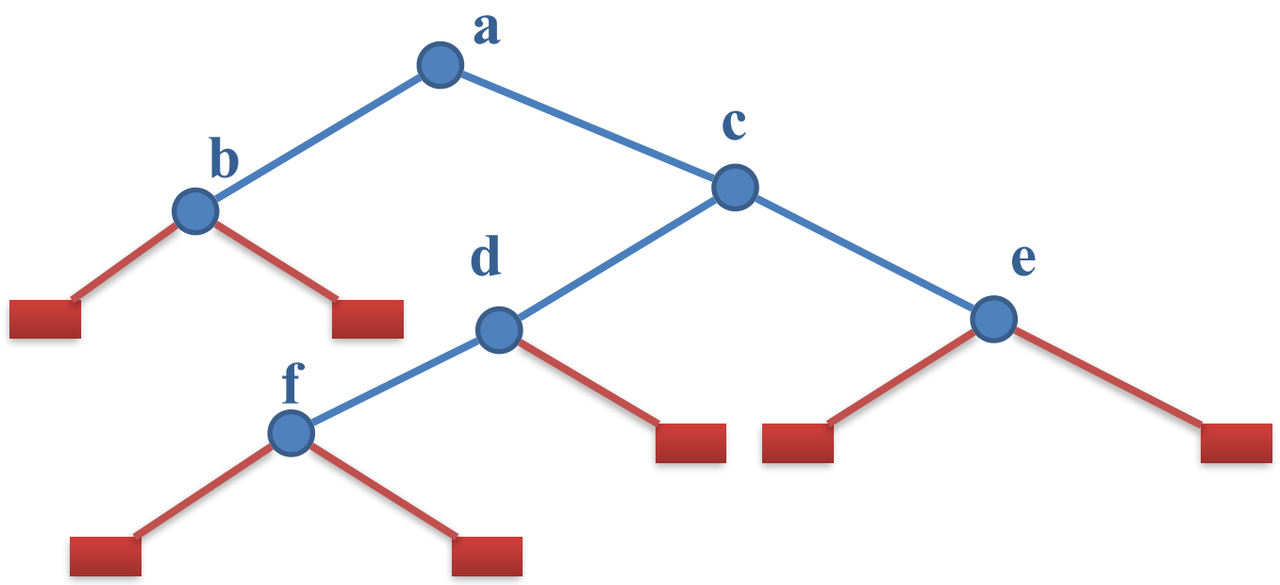
$$E = I + 2n$$

- 如下图的扩充二叉树中：

  E＝2＋2＋4＋4＋3＋3＋3＝21

  I＝0＋1＋1＋2＋2＋3＝9

  n＝6



二叉树的抽象数据类型

ADT BinTree **is**

operations

BinTree binTree (DataType r, BinTree lt, BinTree rt);

Bool is_empty (BinTree t);

DataType root (BinTree t);

BinTree left (BinTree t);

Bintree right (BinTree t);

BinTree set_left (BinTree t, BinTree new);

BinTree set_right (BinTree t, BinTree new);

end ADT BinTree

二叉树的周游

什么是周游?

- **二叉树的周游**是指按某种方式访问二叉树中的所有结点，使每个结点被访问一次且只被访问一次。
  - 深度优先周游
  - 广度优先周游

深度优先周游

- 若以符号D、L、R分别表示根结点、左子树、右子树，则二叉树的周游共有六种方式：DLR，LDR，LRD，DRL，RDL和RLD。

如果限定先左后右，则只能采用前三种周游方式:

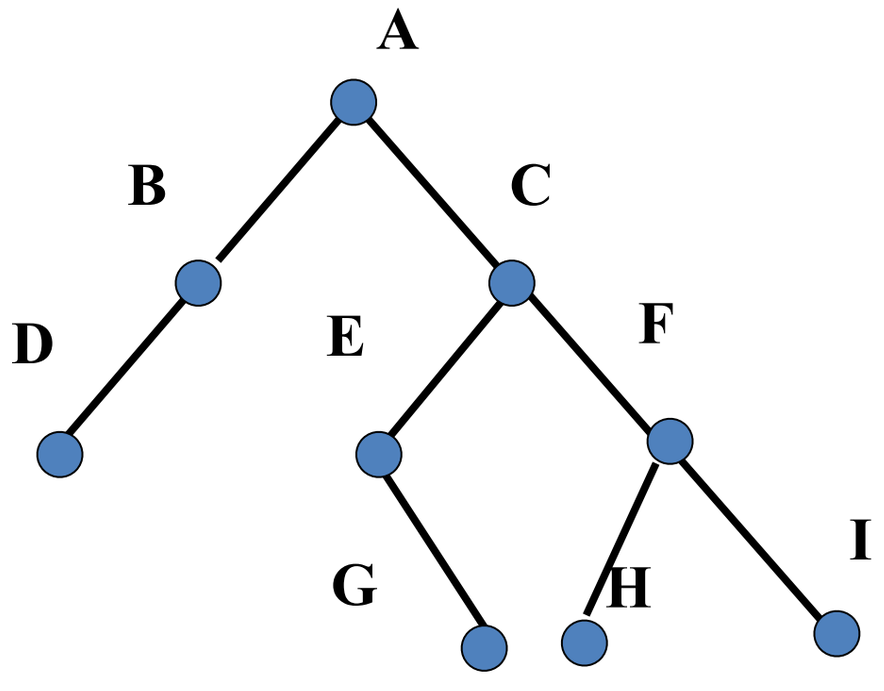
DLR-----先根次序（简称先序或前序）周游

LDR-----中根次序（简称中序或对称序）周游

LRD-----后根次序（简称后序）周游

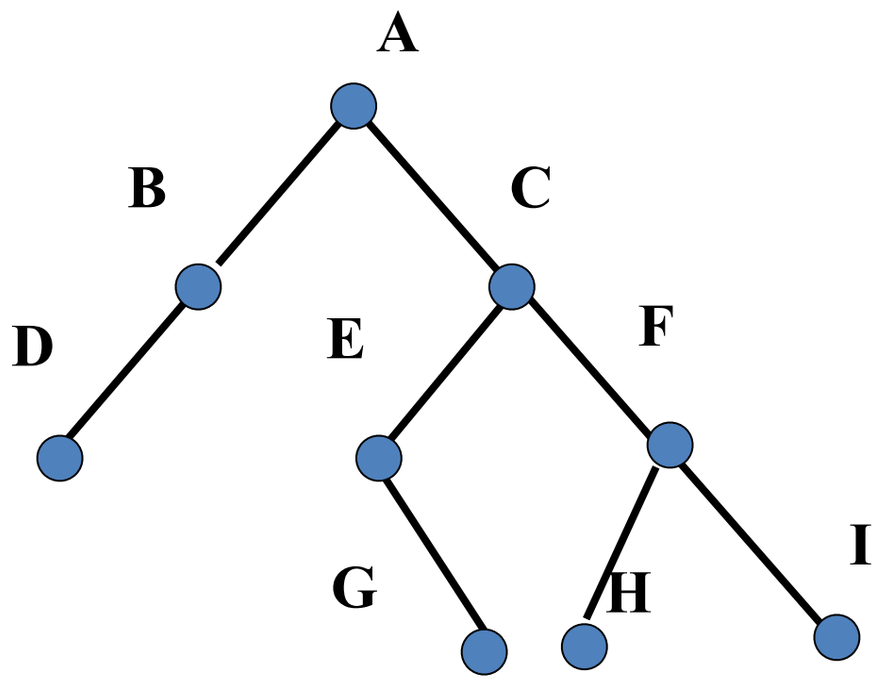
先根序列

- 若二叉树不空，则先访问根；然后按先根次序周游左子树；最后按先根次序周游右子树。
- 下图所示二叉树，先根次序周游得到的结点序列（也称为先根序列）为：A，B，D，C，E，G，F，H，I



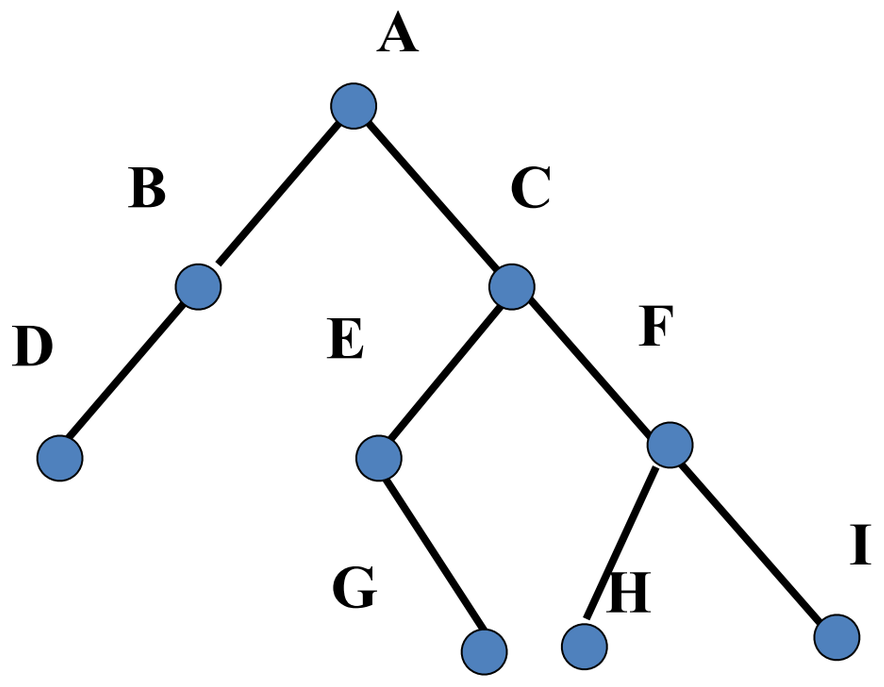
后根序列

- 若二叉树不空，则先按后根次序周游左子树；然后按后根次序周游右子树；最后访问根。
- 前图所示的二叉树，后根次序周游得到的结点序列（也称为后根序列）为：D，B，G，E，H，I，F，C，A



对称(中根)序列

- 若二叉树不空，则先按对称序周游左子树；然后访问根；最后按对称序周游右子树。
- 对前图所示的二叉树，按对称序周游得到的结点序列（也称为对称或中根序列）为：D，B，A，E，G，C，H，F，I

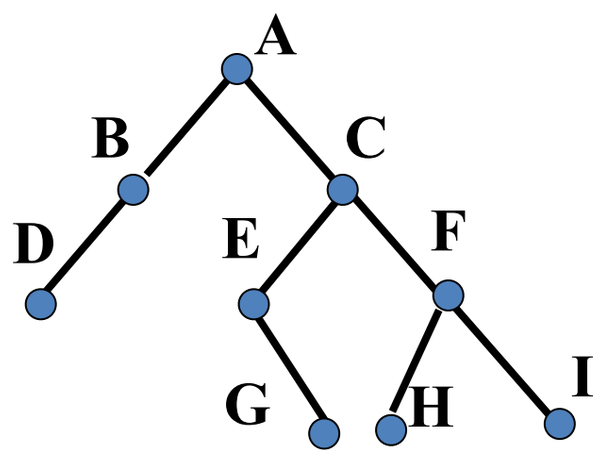


提示

- 对于给定的二叉树，可以唯一确定它的先根序列、后根序列和对称序列。但是反过来，给定一个二叉树的任意一种周游的序列，无法唯一确定这个二叉树。
- 一般而言，如果已知一个二叉树的对称序列，又知道另外一种周游序列（无论是先根序列还是后根序列），就可以唯一确定这个二叉树。
- 先根ABECFDGHIJKL
- 中根EBFCDAIJKHGL

广度优先周游

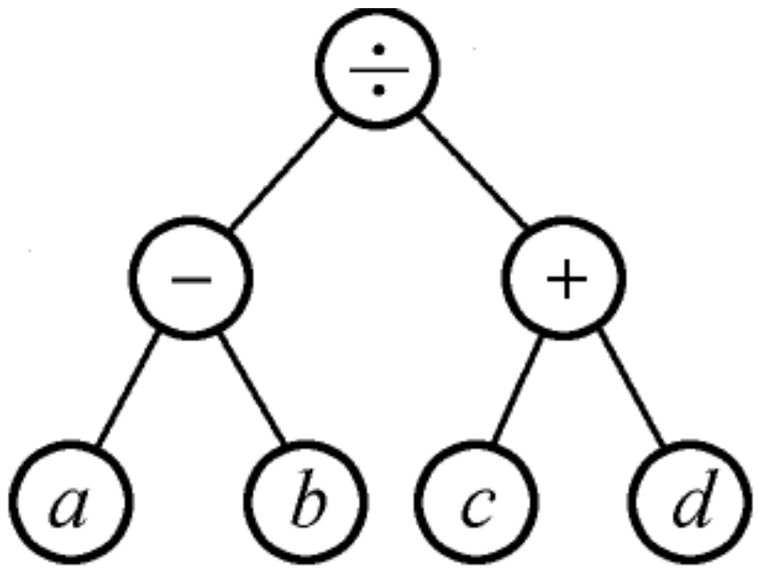
- 若二叉树的高度为h，则从0到h逐层如下处理：从左到右逐个访问存在的结点。
- 广度优先周游一棵二叉树所得到的结点序列，叫作这棵二叉树的层次序列。
- 前图的二叉树，广度优先周游所得到的结点层次序列为：A，B，C，D，E，F，G，H，I



例子

对这个二叉树进行先根、后根和中根周游所得到的线性序列分别为：

先根序列　　÷ － a b ＋ c d
后根序列　　a b － c d ＋ ÷
中根序列　　a － b ÷ c ＋ d



它们也分别被称为表达式的前缀表示、后缀表示和中缀表示。

周游的抽象算法

- 这里给出的算法，都是建立在上节关于抽象数据类型基本运算的基础上。它们不依赖于二叉树的具体存储结构，所以称为抽象算法。

先根次序周游的递归算法

```
void preOrder( BinTree  t) {
   if (t==NULL) return;
   visit(root(t));
   preOrder(leftChild(t));
   preOrder(rightChild(t));
}
```

对称序周游的递归算法

```
void inOrder(BinTree  t){
   if (t==NULL) return;
   inOrder(leftChild(t));
   visit(root(t));
   inOrder(rightChild(t));
}
```

后根次序周游的递归算法

```
void postOrder( BinTree  t) {
   if (t==NULL) return;
   postOrder(leftChild(t));
   postOrder(rightChild(t));
   visit(root(t));
}
```

先根次序周游

- 考虑先根序的非递归遍历算法

  需要用一个栈保存树尚未访问过部分的信息

- 考虑先根序遍历的一种方法。基本想法很简单
  - 先根序，访问遇到的结点并沿左枝下行

    尚未访问的右分支需要记录，将其入栈
  - 遇空树时回溯，取出栈中保存的右分支，像一棵树一样遍历它
- 算法还有一些细节，主要是循环的控制
  - 循环条件：“当前树非空（这棵树需要遍历）或者栈不空（还存在整个树的未遍历部分）”，这时就应该继续循环
  - 在向下检查左分支时把经过结点的右分支入栈（也要用一个循环）
  - 弹出栈中元素（一个右子树）回溯，要做的也是遍历一棵二叉树

先根次序周游的非递归算法

```
void nPreOrder(BinTree  t) {
  Stack  s;                                  /*栈元素的类型是BinTree */
  BinTreeNode* c;
  if (t = = NULL) return;
  s = createEmptyStack();  push(s, t);
  while (!isEmptyStack(s)) {                 /*每当栈不空*/
    c = top(s);   pop(s);                    /*取栈顶，出栈*/
    if (c != NULL) {
      visit(root(c));                        /*访问*/
      push(s, rightChild(c));                /*右子树进栈*/
      push(s, leftChild(c));                 /*左子树进栈*/
    }
  }
}
```

算法的时间代价

- 假设栈的主要操作只要常量时间，算法中每个二叉树恰好进栈、出栈各一次，所以它的时间代价为O(n)，其中n为二叉树中子二叉树（也是结点）的个数。

对称序周游

1. 若当前二叉树不为空时，则沿其左子树尽量前进，在前进过程中，把所经过的二叉树逐个压入栈中，直到左子树为空。
2. 弹出栈顶元素为当前二叉树，并访问该二叉树的根；
3. 如果当前二叉树有右子树，再进入它的右子树（作为当前二叉树），从1开始执行上述过程；
4. 如果当前二叉树没有右子树，但是栈不空，转2。

重复上面的处理，直到当前二叉树没有右子树并且栈也为空时，周游结束。

对称序周游的非递归算法

```
void nInOrder(BinTree  t) {
Stack s= createEmptyStack();     /*栈元素类型是BinTree */
BinTree  c= t;
if (c = = NULL) return;
do {
   while (c != NULL)  {   push(s, c);   c = leftChild(c); }
   c = top(s); pop(s); visit(root(c) );
   c = rightChild(c);
   } while (c != NULL ll !isEmptyStack(s));
}
```

算法的时间代价

- 假设栈的主要操作只要常量时间，算法中每个二叉树恰好进栈、出栈各一次，所以它的时间代价还是O(n)，其中n为二叉树中子二叉树（也是结点）的个数。
- 外表看它是一个双重循环，但时间代价还是线性的。

后根次序周游

1. 首先由该二叉树找到其左子树，周游其左子树，周游完返回到这个二叉树的根；
2. 然后由该二叉树找到其右子树，周游其右子树，周游完再次返回到这个二叉树的根，
3. 最后才能访问该二叉树的根结点。

为此必须在算法中增加对二叉树出栈的判断：

如果是从栈顶二叉树的左子树回来，就直接进入右子树周游；

如果是从栈顶二叉树的右子树回来，就执行出栈，访问该二叉树的根结点。

后根次序周游的非递归算法

```
void nPostOrder2( BinTree  t ) {
   Stack s = createEmptyStack ( );  /*栈中元素的类型是BinTree*/
   BinTree  p = t;
   while ( p != NULL ll !isEmptyStack (s) ) {
    while (p != NULL) {
     push ( s, p );
     p = leftChild (p)?   leftChild (p)：   rightChild(p);
    }                                          /* 循环到当前需要处理的结点*/
   p = top (s); pop (s); visit(root(p));
                                               /* 栈顶二叉树的根是应访问结点*/
   if ( !isEmptyStack (s) && leftChild (top (s)) == p )
      p = rightChild(top (s)) ;                /* 栈不空，且为从左子树退回*/
   else p = NULL;                              /*从右子树回来，退到上一层处理 */
   }
}
```

算法的代价分析

- 假设栈的主要操作只要常量时间，算法中每个二叉树恰好进栈、出栈各一次，所以它的时间代价还是O(n)，其中n为二叉树中子二叉树（也是结点）的个数。
- 外表看它是一个双重循环，但时间代价还是线性的。
- 各种深度周游算法的空间代价主要是栈。最坏情况是O(n)。

广度优先周游

- 根据广度优先周游的思想不难想到，可以利用一个队列实现其算法：

- 首先把二叉树送入队列；其后，每当从队首取出一个二叉树访问根之后，马上把它的子二叉树按从左到右的次序送入队列尾端；重复此过程直到队列为空。

广度优先周游

```
void levelOrder(BinTree  t) {
  BinTree  c, cc;
  Queue  q= createEmptyQueue();
                                          /* 队列元素为BinTree类型*/
  if (t==NULL)  return;                   /*空二叉树返回*/
  c = t;  enQueue(q,c);                   /*将二叉树送入队列*/
  while (!isEmptyQueue(q)) {
    c = frontQueue(q);  deQueue(q);  visit(root( c ));
                                          /*从队列首部取出二叉树并访问*/
    cc = leftChild( c );  if(cc!=NULL)  enQueue(q,cc);
                                          /*将左子树送入队列*/
    cc= rightChild( c );  if(cc!=NULL)  enQueue(q,cc);
                                          /*将右子树送入队列*/
  }
}
```

算法的代价分析

- 每个二叉树进队列一次出队列一次，所以时间代价为O(n)。主要空间代价是需要队列的附加空间。若二叉树结点个数为n，最坏的情况出现在完全二叉树时，需要大约n/2个队列元素的空间。

二叉树的实现

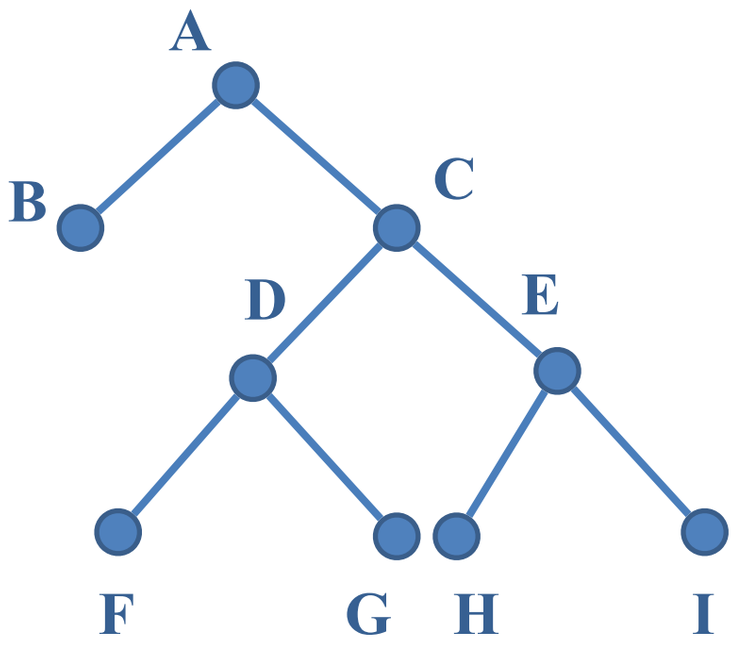
- 顺序表示
- 链接表示
- 线索二叉树

二叉树的顺序表示

- 采用一组连续的存储单元来存放二叉树中的结点。

  - 对于完全二叉树，按照从上（根结点）到下（叶结点）和从左到右的顺序，对二叉树中的所有结点从0到n-1编号，这样存放到一维数组中。只要通过数组元素的下标关系，就可以确定二叉树中结点之间的逻辑关系。

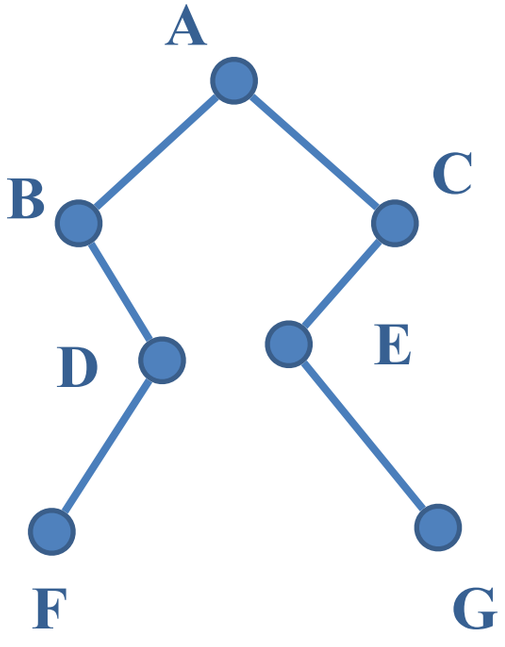
二叉树的顺序表示

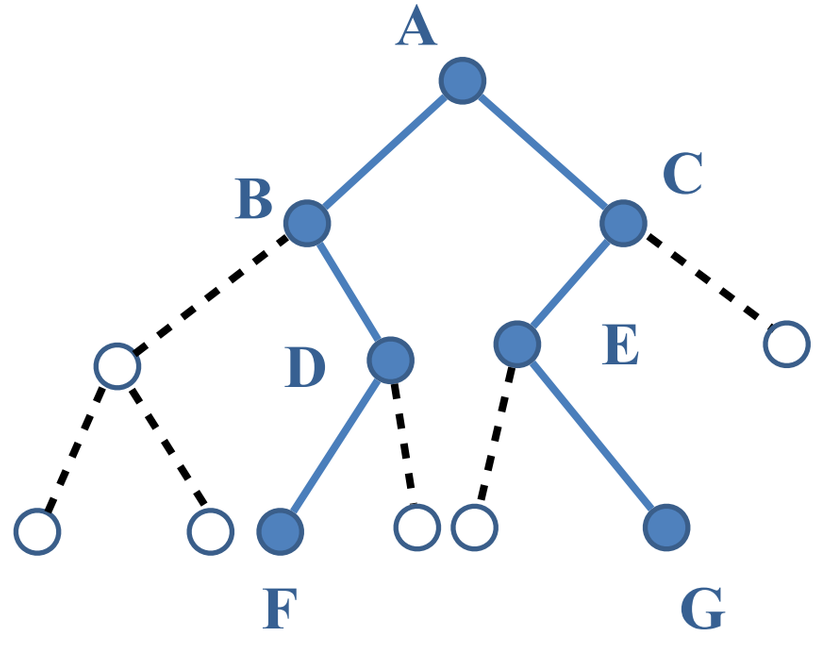


A	B	C	D	E	F	G	H	I

二叉树的顺序表示

- 对于一般的二叉树，如果仍采用顺序表示，首先要对它进行扩充，增加一些并不存在的空结点，使之成为一棵完全二叉树，然后再用一维数组顺序存储。





A	B	C	∧	D	E	∧	∧	∧	F	∧	∧	G

二叉树的顺序表示

```
struct SeqBinTree{        /* 顺序二叉树类型定义 */
   int   MAXNUM        /* 完全二叉树中允许结点的最大个数 */
   int   n;            /* 改造成完全二叉树后，结点的实际个数 */
   DataType  *nodelist;        /* 存放结点的数组 */
};
typedef  struct SeqBinTree  *PSeqBinTree;
                    /*顺序二叉树类型的指针类型*/
```

运算的实现

```
int parent_seq(PSeqBinTree t, int p){        /*返回下标为p的结点的父结点的下标*/
    if (p < 0 ll p >= t->n) return -1;
    return (p - 1) / 2;
}

int leftChild_seq(PSeqBinTree t, int p){     /*返回下标为p的结点的左子结点的下标*/
    if (p < 0 ll p >= t->n) return -1;
    return 2*p + 1;                                  /*可能不存在*/
}

int rightChild_seq (PSeqBinTree t, int p){ /*返回下标为p的结点的右子结点的下标*/
    if (p < 0 ll p >= t->n) return -1;
    return 2 * (p + 1) ;                             /*可能不存在*/
}
```

链接表示

- 二叉树中每个结点对应链接表示中的一个结点。
- 每个结点中，除了存储结点本身的数据外，再设置两个链接字段：llink和rlink，分别存放结点的左子结点和右子结点的位置。
- 当结点的某个子树为空时，则相应的链接为空。
- 这种表示法称为左-右指针表示法。

左-右指针表示法

```
struct BinTreeNode;                    /* 二叉树中结点 */
typedef  struct BinTreeNode  * PBinTreeNode;
                                       /* 结点的指针类型 */
struct BinTreeNode{
  DataType  info;                      /* 数据域 */
  PBinTreeNode  llink;                 /* 指向左子结点 */
  PBinTreeNode  rlink;                 /* 指向右子结点 */
};
```

左-右指针表示法

- 由于递归是二叉树的固有特性，二叉树的许多处理都可以用递归算法来描述，因此，为了运算和参数传递的方便，不再对二叉树进行封装，直接**将二叉树定义为指向结点的指针类型**：

```
typedef  struct BinTreeNode  *BinTree;
```

- 在实际应用中，将二叉树作为参数传递时，可能需要传递二叉树根结点指针的地址，因此，为了说明方便，可以**引入二叉树类型的指针类型**：

```
typedef  BinTree  *PBinTree;
```

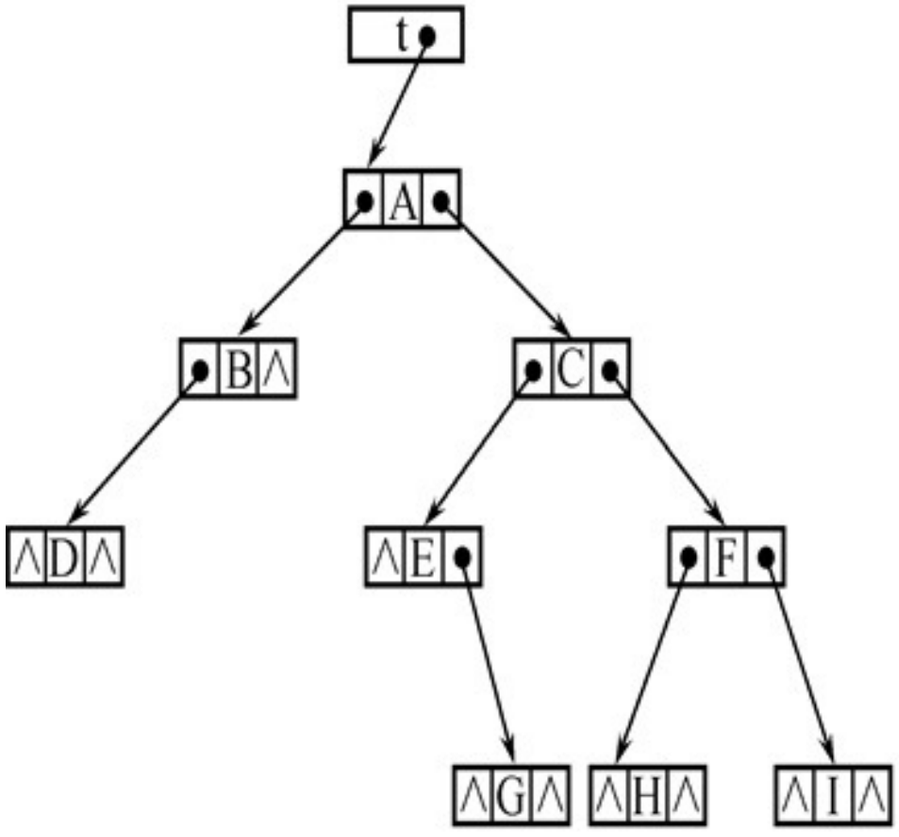
运算的实现

```
/*返回结点p的左子结点的地址*/
PBinTreeNode leftChild_link(PBinTreeNode p){
   if (p == NULL) return NULL;
   return p->llink;
}
/*返回结点p的右子结点的地址*/
PBinTreeNode rightChild_link(PBinTreeNode p){
   if (p == NULL) return NULL;
   return p->rlink;
}
```

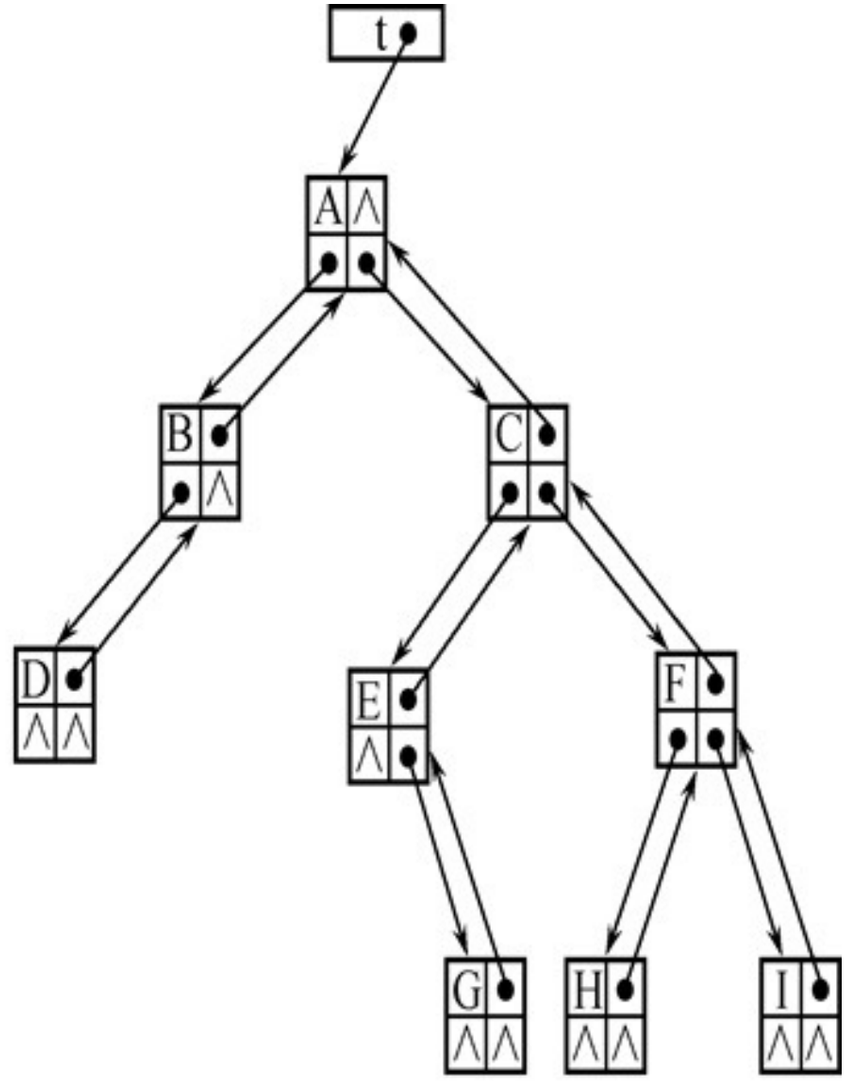
求父结点的操作

- 只能从t出发，使用前面介绍的周游算法，通过算法中的访问函数(visit)，检查当前结点是否所求结点的父结点。最坏的时间代价与周游整个二叉树的代价相同。
- 为了提高求父结点操作的速度，可以采用的另一种链接表示方式是三叉链表表示，即给二叉树中的每个结点增加一个指向父结点的指针域。
- 采用三叉链表表示，既便于查找子结点，又便于查找父结点，但是相对于左－右指针表示而言，它增加了空间开销。

三叉链表表示



(a) 左－右链接表示



(b) 三叉链表表示

线索二叉树

- 线索二叉树是对于左－右指针表示法的一种修改。
- 它利用结点的空的左指针（llink）存储该结点在某种周游序列中的前驱结点的位置；利用结点的空的右指针（rlink）存储该结点在同种周游序列中的后继结点的位置。
- 这种附加的指向前驱结点和后继结点的指针称作**线索**，加进了线索的二叉树左－右指针表示称作**线索二叉树**。

线索二叉树的结点

为区分左右指针和线索，需要在每个结点里增加两个标志位ltag和rtag。

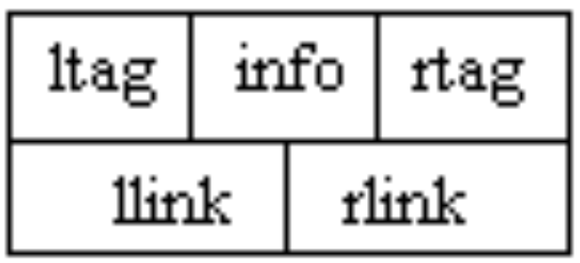
ltag=0，llink是指针，指向结点的左子结点；

ltag=1，llink是线索，指向结点的对称序的前驱结点；

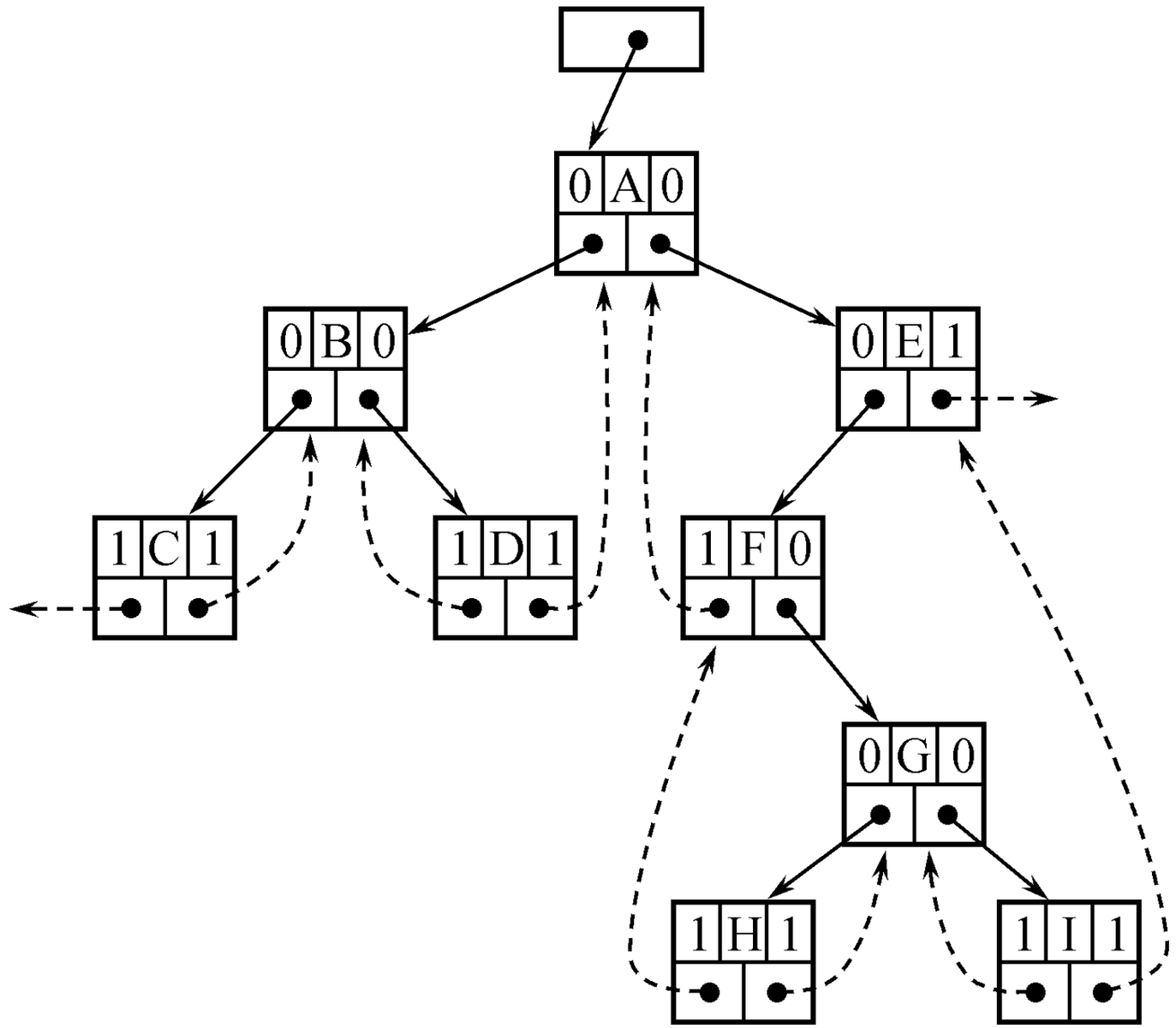
rtag=0，rlink是指针，指向结点的右子结点；

rtag=1，rlink是线索，指向结点的对称序的后继结点。

增加了标志域的结点结构为：



线索二叉树



线索二叉树类型的定义

```
struct ThrTreeNode;          /* 线索二叉树中的结点 */
typedef struct ThrTreeNode * PThrTreeNode;
                              /* 指向线索二叉树结点的指针类型 */
struct ThrTreeNode  {        /* 线索二叉树中结点的定义 */
   DataType info;
   PThrTreeNode  llink, rlink;
   int ltag, rtag;
   };
typedef struct ThrTreeNode * ThrTree; /* 线索二叉树类型的定义 */
typedef ThrTree * PThrTree;      /* 线索二叉树类型的指针类型 */
```

按对称序线索化二叉树

- 在未线索化之前，所有结点的llink和rlink都是指向子结点指针，因此所有ltag和rtag的初始状态都为0。给出一棵二叉树，要将它按对称序线索化，其做法就是按对称序周游此二叉树，在周游的过程中用线索代替空指针。

- 注意与对称序周游二叉树算法的比较！！

按对称序线索化二叉树

```
void thread(ThrTree  t) {
PSeqStack  st = createEmptyStack (M); /*栈元素类型是ThrTree, M一般取t的高度 */
 ThrTree  p,   pr;
if (t==NULL) return ;
p = t;  pr = NULL;
do {
  while (p!=NULL) {push_seq(st,p);p= p->llink;}
  p = top_seq(st);  pop_seq(st);
  if (pr!=NULL) {
      if (pr->rlink==NULL) { pr->rlink = p;pr->rtag = 1; } /*修改前驱结点的右指针*/
      if (p->llink==NULL) { p->llink = pr; p->ltag = 1;}     /*修改该结点的左指针*/
      }
  pr = p;  p = p->rlink;
} while ( !isEmptyStack_seq(st) ll p!=NULL );
}
```

按对称序周游对称序线索二叉树

- 要按对称序周游对称序线索二叉树，首先找到对称序列中的第一个结点，然后依次找到结点的后继结点，直至其后继结点为空即可。
- 第一个结点也很容易找，只要从根结点出发沿着左指针不断往下走，直至左指针为空，到达“最左下”的结点，这就是对称序第一个结点。
- 找任意结点的对称序后继时，也非常容易做：一个结点的右指针字段如果是线索，则它就指向该结点在对称序下的后继；如果不是线索，则它指向该结点右子树的根，而该结点在对称序下的后继应是此右子树的最左下结点。

按对称序周游对称序线索二叉树

```
void nInOrder(ThrTree  t ) {
ThrTree  p= t;
if (t==NULL) return ;
while ( p->llink!=NULL && p->ltag==0 )  p = p->llink;
while (p!=NULL)  {
   visit(*p);
   if ( p->rlink!=NULL && p->rtag==0 ) {        /* 右子树不是线索时 */
        p = p->rlink;
        while (p->llink!=NULL&&p->ltag==0)
             p = p->llink;                              /*顺右子树的左子树一直向下*/
       }
   else   p = p->rlink;
  }
}
```

本讲重点

- 二叉树的概念和主要性质
- 二叉树的周游算法（递归与非递归）
- 二叉树的实现方法