第6章

树和二叉树

Tree and Binary Tree

主讲：顾为兵

第六章　树和二叉树

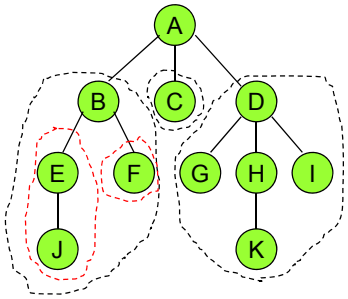
目录

§6.1 树的定义和基本术语

6.1.1 树的定义（P118）

- 树是n（n≥0）个结点的有限集。在任意一棵非空树中，有且仅有一个称为根的结点；其余结点分为m（m≥0）个互不相交的子集，每个子集又是一棵树，称为根的子树。
- 这是一个递归的定义—在树的定义中又用到了树的概念。
- 递归是树的固有特性。

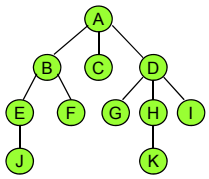
- 树根：A
- 三个互不相交的子集：
  {B，E，F，J}
  {C}
  {D，G，H，I，K}
- 每个子集都是满足树的定义的树，称为A的子树——B子树、C子树、D子树。
- 树根A没有直接前驱；其余结点有且仅有一个直接前驱有，有0个或多个直接后继。



树的特征：层次性和分支性

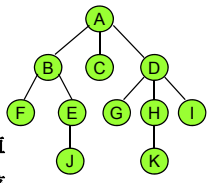
§6.1 树的定义和基本术语

6.1.2 树的术语



- 结点的度：结点的非空子树个数
- 树的度：树中各结点度的最大值
- 分支结点(非终端结点)：度>0的结点
- 叶子结点(终端结点)：度==0的结点
- 孩子：结点子树的根称为该结点的孩子
- 双亲：孩子的前驱结点称为该结点的双亲
- 兄弟：同一个双亲的结点互称为兄弟
- 子孙：以某结点为根的各个子树上的所有结点称为该结点的子孙
- 祖先：从树根到该结点所经过的所有分支结点称为该结点的祖先

树的定义和基本术语 • 树的定义



- 结点的层次：从树根开始自上而下编号，树根的层次为1，其余结点的层次为其双亲的层次+1
- 树的深度（高度）：结点层次的最大值
- 有序树和无序树：若树中所有结点的孩子都“长幼”有序，位置不能互换，称为有序树，否则称为无序树。
- 森林：m（m≥0）个树的集合

树的定义和基本术语 • 树的定义

树的基本运算(教材119)：

初始化空树	`InitTree(&T)`
销毁树	`DestroyTree(&T)`
创建树	`CreateTree(&T, definition)`
清空树	`ClearTree(&T)`
判断空树	`TreeEmpty(T)`
求树的深度	`TreeDepth(T)`
求树根	`Root(T)`
读结点	`Value(T, cur_e)`
写结点	`Assign(&T, cur_e, value)`

树的定义和基本术语 • 树的定义

求双亲	`parent(T, cur_e)`
求长子	`LeftChild(T, cur_e)`
求右邻兄弟	`RightSibling(T, cur_e)`
插入子树	`InsertChild(&T, &p, i, c)`
删除子树	`DeleteChild(&T, &p, i)`
按层次遍历	`TraverseTree(T, visite())`

第六章 树和二叉树

目录

二叉树

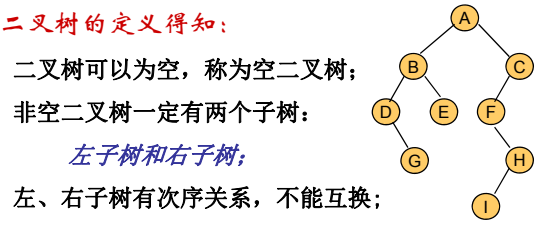
§6.2 二叉树

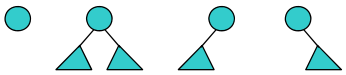
6.2.1 二叉树的定义（P121）

- 二叉树是n(n≥0)个结点的有限集。该集合或者为空，或者由一个根加上两棵互不相交的、分别称为左子树和右子树的二叉树组成。
- 这是一个递归的定义——在二叉树的定义中又用到了“二叉树”的概念。

从二叉树的定义得知：

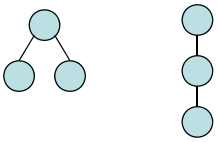
1. 二叉树可以为空，称为空二叉树；
2. 非空二叉树一定有两个子树：
   左子树和右子树；
3. 左、右子树有次序关系，不能互换；
4. 二叉树可以有5种基本形态：



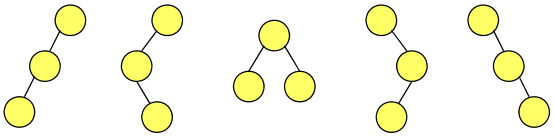
Ø 

5. 二叉树不是结点的度都不超过2的有序树。

三个结点的树有两种不同的形态：



三个结点的二叉树有五种不同的形态：



二叉树的基本运算(教材P121－123)：

初始化空二叉树	InitBiTree(&T)
销毁二叉树	DestroyBiTree(&T)
创建二叉树	CreateBiTree(&T, definition)
清空二叉树	ClearBiTree(&T)
判断空二叉树	BiTreeEmpty(T)
求二叉树深度	BiTreeDepth(T)
求树根	Root(T)
读结点	Value(T, e)
写结点	Assign(T, &e, value)

求双亲	parent(T, e)
求左孩子	LeftChild(T, e)
求右孩子	RightChild(T, e)
求左兄弟	LeftSibling(T, e)
求右兄弟	RightSibling(T, e)
插入子树	InsertChild(T, p, LR, c)
删除子树	DeleteChild(T, p, LR)
先序遍历二叉树	PreOrderTraverse(T, visite())
中序遍历二叉树	InOrderTraverse(T, visite())
后序遍历二叉树	PostOrderTraverse(T, visite())
按层次遍历	levelOrderTraverse(T, visite())

§6.2 二叉树

6.2.2 二叉树的性质 （P123－126）

性质1. 在二叉树的第 i 层上至多有2^{i-1}个结点；

性质2. 深度为k的二叉树上至多有2^k-1个结点

$$\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$$

性质3. 设二叉树中有n_2个度为2个结点，n_1个度为1的结点，n_0个度为0的结点，则有：$n_0=n_2+1$

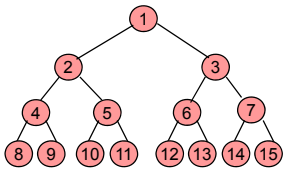
$n=n_0+n_1+n_2$ (1)

树枝数 $=n_0\times 0+n_1\times 1+n_2\times 2=n_1+2n_2$

n=树枝数$+1=n_1+2n_2+1$ (2)

由(1)、(2)可得$n_0=n_2+1$

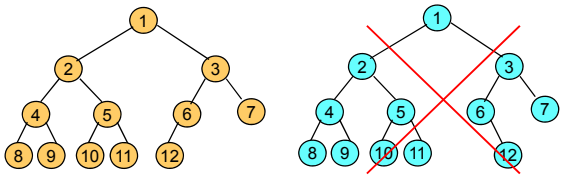
满二叉树：结点数 $n=2^k-1$



完全二叉树：

将深度为k，有n个结点的二叉树自上而下，自左向右进行编号，编号与深度为k的满二叉树中前n个结点一致。

前k－1层是满的，第k层可以不满，但第k层结点集中在左侧



性质4. 具有n个结点的完全二叉树的深度为

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

证明：

设完全二叉树的深度为k，根据完全二叉树的定义有

$2^{k-1}-1 < n \le 2^k-1$ 或 $2^{k-1} \le n < 2^k$

取对数有

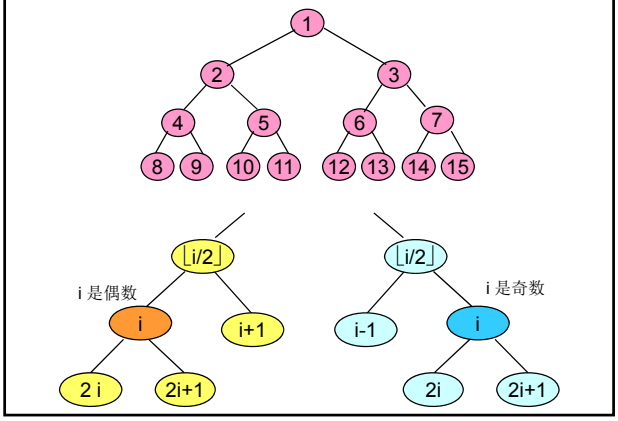
$k-1 < \log_2 n \le k$

∵k是整数，∴$k=\lfloor \log_2 n \rfloor+1$

性质5. 将完全二叉树自上而下，自左向右地编号，树根的编号为1。对于编号为i的结点x有：

1. 若i＝1，则x是根；若i>1，则x的双亲的编号为$\lfloor i/2 \rfloor$；
2. 若x有左孩子，则x左孩子的编号为2i；
3. 若x有右孩子，则x右孩子的编号为2i＋1；
4. 若i为奇数且i>1，则x的左兄弟为i－1；
5. 若i为偶数且i<n，则x的右兄弟为i＋1；

二叉树 • 二叉树的性质



二叉树

§6.2 二叉树

6.2.1 二叉树的定义

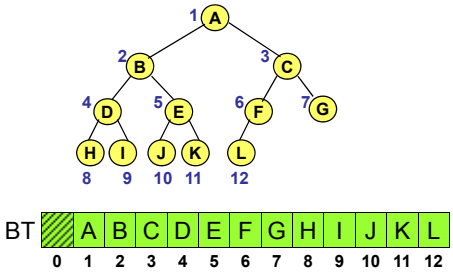
6.2.2 二叉树的性质

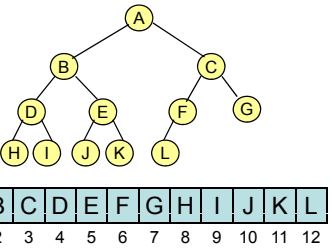
6.2.3 二叉树的存储结构

二叉树 • 二叉树的存储结构

一、二叉树的顺序存储结构

二叉树的性质5为用数组存储完全二叉树提供了依据：





结点在完全二叉树中的编号与数组下标一致，可利用性质5来计算结点的双亲、孩子、兄弟的下标。

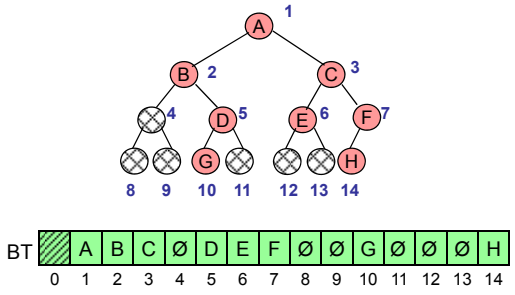
举例：已知结点E的下标是5，容易得知：

E的双亲为BT[2]；　E的左兄弟是E[4]；

E的左孩子是BT[10]；　E的右孩子是BT[11]。

二叉树 • 二叉树的存储结构

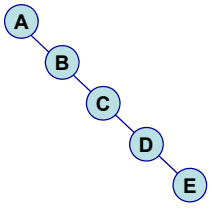
对于普通二叉树，可以将其补足成完全二叉树后再进行编号：



二叉树 • 二叉树的存储结构

这样表示的二叉树可能会造成空间的极大浪费，因而不常采用。

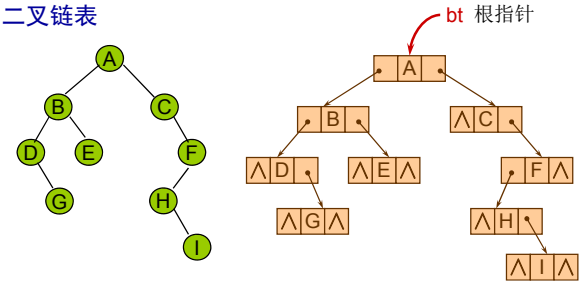
比如有5个结点的右单枝树，需为其分配31个结点空间。



二、二叉树的链式存储结构——二叉链表


```
typedef struct BiTNode{
   ElemType          data;
   struct BiTNode    *lchild, *rchild;
} BiTNode, *BiTree;
```

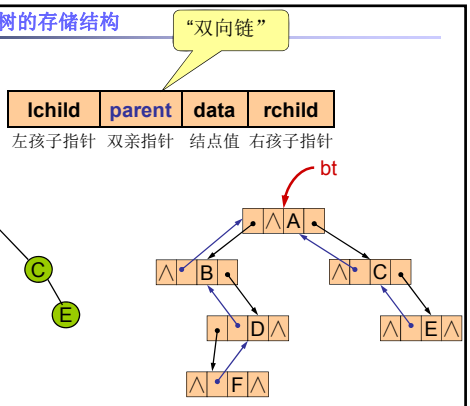

二叉链表



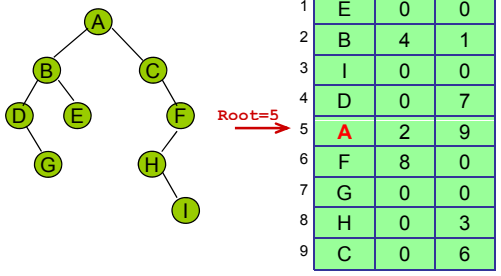
每个非空指针表示一个树枝

n个结点的二叉树共有n+1个空指针

三叉链表



静态二叉链表



	data	lchild	rchild
0			
1	E	0	0
2	B	4	1
3	I	0	0
4	D	0	7
5	A	2	9
6	F	8	0
7	G	0	0
8	H	0	3
9	C	0	6

第六章　树和二叉树

目录

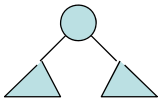
§6.3 遍历二叉树

6.3.1 先序、中序和后序遍历二叉树

遍历：按某种次序访问二叉树的所有结点，且每个结点仅访问一次。

- 根据二叉树的结构：左子树_根_右子树，我们可以把对二叉树的遍历分解为三项子任务：

  访问根 ——D
  遍历左子树 ——L
  遍历右子树 ——R



- 根据这三项任务的执行次序的不同，有六种可能的遍历方案：

  DLR、LDR、LRD　　*先左后右*

  ~~DRL、RDL、RLD~~　　*先右后左*

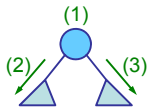
- 如果约定先左后右，则取前三种方案。

根据访问根的时机不同，将这三种遍历方案分别称为：

先根遍历（先序遍历）DLR
中根遍历（中序遍历）LDR
后根遍历（后序遍历）LRD

先序遍历的递归定义：

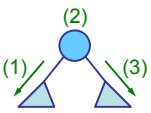
若二叉树为空，则空操作；否则
(1)访问根；
(2)先序遍历左子树
(3)先序遍历右子树




```
void preorder(BiTree bt) {
    if( !bt ) return;
    visite(bt->data);      //访根
    preorder(bt->lchild);
    preorder(bt->rchild);
}
```


中序遍历的递归定义：

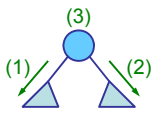
若二叉树为空，则空操作；否则
(1)中序遍历左子树
(2)访问根；
(3)中序遍历右子树




```
void inorder(BiTree  bt) {
    if( !bt ) return;
    inorder(bt->lchild);
    visite(bt->data);      //访根
    inorder(bt->rchild);
}
```

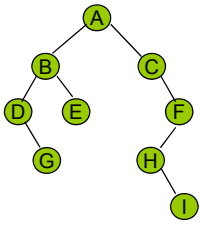

后序遍历的递归定义：

若二叉树为空，则空操作；否则
(1)后序遍历左子树
(2)后序遍历右子树
(3)访问根；




```
void postorder(BiTree  bt) {
    if( !bt ) return;
    postorder(bt->lchild);
    postorder(bt->rchild);
    visite(bt->data);      //访根
}
```

二叉树的遍历



先序遍历序列： A B D G E C F H I

中序遍历序列： D G B E A C H I F

后序遍历序列： G D E B I H F C A

现将三个遍历算法中的visite语句去掉，我们发现这三个算法完全一样。这说明三种遍历的路径是一样的，只是访问根的时机不同而已。

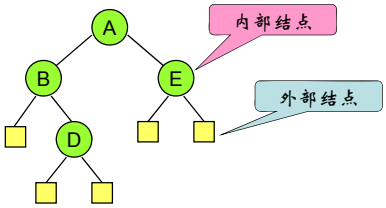
```
void preorder
        (BiTree bt) {
  if( !bt ) return;
  visite(bt->data);
  preorder(bt->lchild);
  preorder(bt->rchild);
}
```

```
void inorder
        (BiTree bt) {
  if( !bt ) return;
  inorder(bt->lchild);
  visite(bt->data);
  inorder(bt->rchild);
}
```

```
void postorder
        (BiTree bt) {
  if( !bt ) return;
  postorder(bt->lchild);
  postorder(bt->rchild);
  visite(bt->data);
}
```


扩展二叉树：将二叉树中的空指针用一个虚结点表示，这样构成的二叉树称为该二叉树的扩展二叉树。

扩展二叉树的虚结点不是原二叉树的一部分，称为外部结点，而原树中结点称为内部结点。



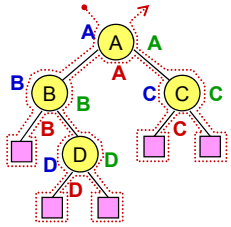
围绕扩展二叉树走一圈，每个内部结点将相遇三次。

若每次访问都是在：

第 1 次相遇时进行，构成先序遍历序列ABDC

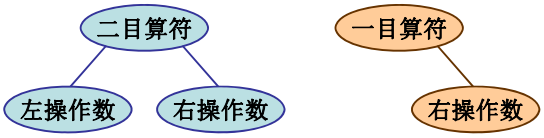
第 2 次相遇时进行，构成中序遍历序列BDAC

第 3 次相遇时进行，构成后序遍历序列DBCA



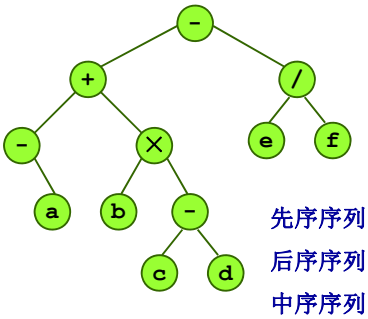
§6.3 遍历二叉树

6.3.2 算术表达式的二叉树表示



二叉树 • 算术表达式的二叉树表示

-a+b×(c-d)-e/f



波兰式

逆波兰式

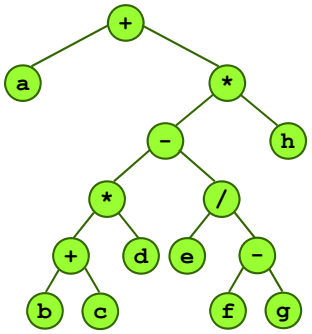
先序序列：-+-a×b-cd/ef

后序序列：a-bcd-×+ef/-

中序序列：-a+b×c-d-e/f

二叉树 • 算术表达式的二叉树表示

a+((b+c)*d-e/(f-g))*h



波兰式：

+a*-*+bcd/e-fgh

逆波兰式：

abc+d*efg-/-h*+

二叉树 • 遍历二叉树

§6.3 遍历二叉树

6.3.1 先序、中序和后序遍历

6.3.2 算术表达式的二叉树表示

6.3.3 二叉树的运算举例

6.3.4 按层次遍历二叉树

6.3.5 创建二叉链表

二叉树 • 遍历二叉树 • 二叉树运算举例

6.3.3 二叉树运算举例

举例1. 计算二叉树中的结点数：

方法一：

```
int Number(BiTree bt){
   //二叉树结点数=1+左子树结点数+右子树结点数
   if(!bt) return 0;    //空二叉树，没有结点
   else { nl=Number(bt->lchild);//计算左子树结点数
          nr=Number(bt->rchild);//计算右子树结点数
          return 1+nl+nr;
        }
}
```

二叉树 • 遍历二叉树 • 二叉树运算举例

计算二叉树中的结点数之方法二：

```
void Number(BiTree bt, int &n) {
 //通过引用参数传递累加器n的值，n在传入时应赋初值0
       if(!bt) return;
       n++;    //累加结点数
       Number(bt->lchild, n);
       Number(bt->rchild, n);
 }
```

二叉树 • 遍历二叉树 • 二叉树运算举例

计算二叉树中的结点数之方法三：

```
int n=0; //n是全局变量
void Number(BiTree bt) {
    //通过全局变量n传递累加器的值，n的初值为0
       if(!bt) return;
       n++;
       Number(bt->lchild);
       Number(bt->rchild);
   }
```

举例2，计算二叉树中叶子的数目，方法一：

```
int Leafs(BiTree bt)   {
   //叶子数=左子树叶子数+右子树叶子数
   if(!bt) return 0;       //空二叉树
   if(! bt->lchild && ! bt->rchild) return 1; //树叶
   ll=Leafs(bt->lchild); //计算左子树的叶子数
   lr=Leafs(bt->rchild); //计算右子树的叶子数
   return ll+lr;
}
```


计算二叉树中叶子的数目，方法二：

```
void Leafs(BiTree bt, int &n){
  //在对二叉树的遍历过程中对叶子数计数，n的初值为0
   if(!bt) return ;
   if(!bt->lchild && !bt->rchild) n++; //叶子
   Leafs(bt->lchild, n);
   Leafs(bt->rchild, n);
}
```


举例3，计算二叉树的深度：

方法一：

```
int Height(BiTree bt) {
   //H=1+Max(左子树深度，右子树深度)
   if(!bt) return 0;
   hl=Height(bt->lchild);  //计算bt的左子树深度
   hr=Height(bt->rchild);  //计算bt的右子树深度
   return 1+(hl>hr ? hl:hr)
}
```


举例3，计算二叉树的深度：

方法二：

```
void Height(BiTree bt, int h, int &depth){
 //h为当前结点的层次值，h的初值为1；
 //depth为当前求得的最大深度，depth的初值为0
   if(bt){
     if(h>depth) depth=h;
     Height(bt->lchild,h+1,depth);//遍历左子树
     Height(bt->rchild,h+1,depth);//遍历右子树
   }//end if
}//Height
```


举例4, 求指针S所指的结点的双亲：

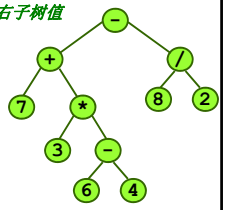
```
void Parent(BiTree bt, BiTree S, BiTree &P){
     //用参数P返回S双亲的指针
     //P对应实参的初值应为NULL
   if(!bt || P!=NULL)return; //P不空表示双亲已经找到
   if(bt->lchild==S||bt->rchild==S) P=bt;  //找到
   else{
      Parent(bt->lchild, S, P); //在左子树上继续找
      Parent(bt->rchild, S, P); //在右子树上继续找
   }
}
```


例5, 打印每个结点的层次数：

```
void level(BiTree bt, int  lev){
  //结点的层次数＝双亲的层次数＋1
  //lev传递双亲的层次，lev所对应的实参的初值应为0
   if(!bt )return;
   lev++;
   printf(bt->data, lev);//打印结点的值和它的层次数
   level(bt->lchild, lev);
   level(bt->rchild, lev);
}
```

举例6：计算算术表达式二叉树的值

```
int  Calculate (BiTree T) {
  //计算表达式二叉树的值，用后序遍历实现
   int nl,nr;
   if(!T->lchild&&!T->rchild) return T->data;
   nl=Calculate(T->lchild); //计算左子树值
   nr=Calculate(T->rchild); //计算右子树值
   switch(T->data) {
      case '+': return nl+nr;
      case '-' : return nl - nr;
      case '*':  return nl * nr;
      case '/':  return nl / nr;
   }
}
```



举例7, 复制二叉树：

```
void CopyTree(BiTree S, BiTree &T){
    //将二叉树S复制到二叉树T，利用后序遍历实现
  if(!S) T=NULL;  //S为空，则T也为空
  else{
     CopyTree(S->lchild, lptr); //复制左子树到lptr
     CopyTree(S->rchild, rptr); //复制左子树到rptr
     T=(BiTree)malloc(sizeof(BiNode));
     T->data=S->data;
     T->lchild=lptr; T->rchild=rptr;
  }//else
}//CopyTree
```


例8. 先序遍历的非递归算法：

```
void PreOrder(BiTree bt){
  if(!bt) return;
  InitStack(S);
  push(S, bt);   //树根的指针进栈
  while(!EmptyStack(S)){
     pop(S, p);
     while(p){  //沿着左链一路向下
        visite(p->data); //访问
        if(p->rchild)push(S,p->rchild);//右孩子进栈
        p=p->lchild;
     }
  }
}
```


中序遍历的非递归算法：

```
void  InOrder(BiTree bt){
  InitStack(S); push(S, bt); //树根的指针进栈
  while( !EmptyStack(S) ){
     while(GetTop(S,p) && p) Push(S, p->lchild);
                          //沿着左链向下到尽头
     pop(S, p);  //空指针出栈
     if(!EmptyStack(S)){
        pop(S, p); visite(p->data);
        push(S, p->rchild);
     } //if
  } //while
}
```


§6.3 遍历二叉树

6.3.4 按层次遍历二叉树

自上而下、自左向右地遍历二叉树时，先被访问的结点的孩子先被访问，故需要借助一个队列来记录已访问过的结点的孩子。

按层次遍历算法思想：

1. 空树，结束。
2. 初始化一个空队列Q，树根入队；
3. 队头e元素出队，访问e；
4. 如果e有左孩子，则左孩子入队；
5. 如果e有右孩子，则右孩子入队；
6. 如果队列不空转3；否则结束。


```
void Traverse(BiTree bt) {
  //按层次遍历二叉树算法
   if( !bt )  return ;          //空树
   InitQueue(Q);                //初始化空队列Q
   EnQueue(Q, bt);              //根入队
   while( !EmptyQueue(Q) ) {
      DeQueue(Q, p);            //队头p出队
      visite(p->data);          //访问p
      if(p->lchild) EnQueue(Q，p->lchild); //p的左孩子入队
      if(p->rchild) EnQueue(Q,p->rchild);  //p的右孩子入队
    }
}
```


§6.3 遍历二叉树

6.3.4 创建二叉链表

CreateBiTree(&T, definition)

扩展二叉树的先序序列

扩展二叉树的后序序列

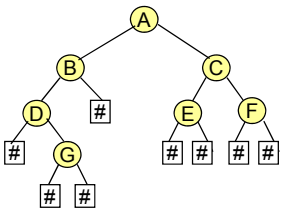
二叉树的先序序列和中序序列

二叉树的后序序列和中序序列

扩展二叉树的按层次遍历序列

已知扩展二叉树的先序序列， 创建二叉树

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




```
BiTree CreateBiTree( ) {
  scanf(ch);
  if(ch=='#') return NULL;
  p=(BiTree)malloc(sizeof(BiNode));
  p->data=ch;
  p->lchild=CreateBiTree( );
  p->rchild=CreateBitree( );
  return p;
}
```

```
BiTree CreateBiTree( ){   //方法1
  scanf(ch);
  if(ch=='#') return NULL;
  p=(BiTree)malloc(sizeof(BiNode));
  p->data=ch;
  p->lchild=CreateBiTree( );
  p->rchild=CreateBitree( );
  return p;
}

                    void CreateBiTree(BiTree &T){//方法2
                       scanf(ch);
                       if(ch=='#') {T=NULL; return;}
                       T=(BiTree)malloc(sizeof(BiNode));
                       T->data=ch;
                       CreateBiTree(T->lchild);
                       CreateBiTree(T->rchild);
                    }
```

6.3.4 构造二叉链表

扩展二叉树的先序序列

扩展二叉树的后序序列

二叉树的先序序列和中序序列

二叉树的后序序列和中序序列

扩展二叉树的按层次遍历序列

已知扩展二叉树的后序序列， 创建二叉树

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

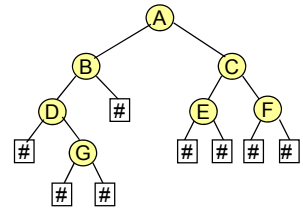
序列的结束符

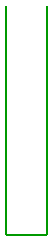
算法思想：

借助一个栈。从左向右扫描后序序列，当遇到#，空指针进栈；当遇到字母，申请一个结点S，弹出两个栈顶，分别作为S的右孩子和左孩子，S进栈；直至序列扫描完

已知扩展二叉树的后序序列， 画出二叉树

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

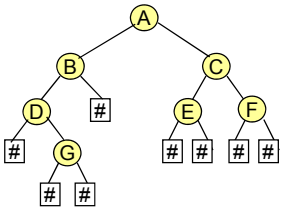




```
void  CreateTree(BiTree &bt) {
//输入扩展二叉树的后序序列，创建二叉树
   InitStack(S);
   while(1){
      scanf(&ch);  //逐个输入后序序列字符
      if(ch==EndMark) break;
      if(ch=='#') push(S, NULL);
      else {
         p=(BiTree)malloc(sizeof(BiNode));
         p->data=ch;
         pop(S,p->rchild); pop(S,p->lchild);
         push(S, p);
      }
   }
   pop(S, bt); //树根留在栈顶
}
```


也可自右向左扫描后序序列，用递归方法实现

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




```
BiTree CreateBiTree( )
{ scanf(ch);  //从右到左输入后序序列
  if(ch=='#') return NULL;
  p=(BiTree)malloc(sizeof(BiNode));
  p->data=ch;
  p->rchild=CreateBiTree( );
  p->lchild=CreateBitree( );
  return p;
}
```


6.3.4 构造二叉链表

扩展二叉树的先序序列

扩展二叉树的后序序列

二叉树的先序序列和中序序列

二叉树的后序序列和中序序列

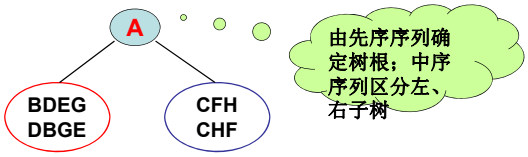
扩展二叉树的按层次遍历序列

二叉树 • 遍历二叉树 • 创建二叉链表

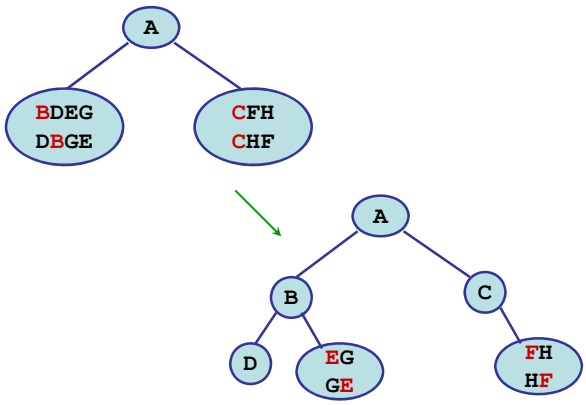
已知二叉树的先序序列为：**ABDEGCFH**，
中序序列为：**DBGEACHF**，画出二叉树。

先序序列 **A** B D E G C F H

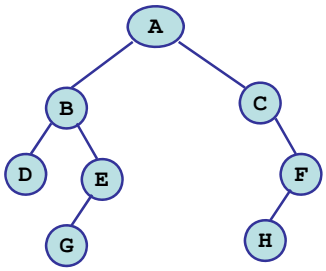
中序序列 D B G E **A** C H F



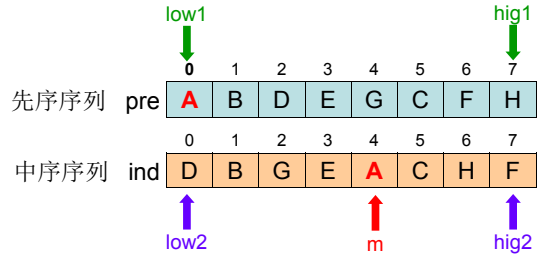
二叉树 • 遍历二叉树 • 创建二叉链表



二叉树 • 遍历二叉树 • 创建二叉链表



二叉树 • 遍历二叉树 • 创建二叉链表



树根：	pre[low1],	ind[m]
左子树：	pre[low1+1..low1+m-low2],	ind[low2..m-1]
右子树：	pre[low1+m-low2+1..hig1],	ind[m+1..hig2]

二叉树 • 遍历二叉树 • 创建二叉链表

```
BiTree  Create (char pre[ ],  char ind[ ],
                int low1, int hig1,  int low2,  int hig2) {
  //根据二叉树的先序序列和中序序列创建二叉链表
   if (low1>hig1) return NULL;
   p=(BiTree)malloc(sizeof(BiNode));
   p->data=pre[ low1];
   m=low2;
   while(pre[ low1] != ind[m] ) m++;        //在中序序列中找根
   p->lchild=Create(pre, ind, low1+1, low1+m-low2, low2, m-1);
   p->rchild=Create(pre,ind, low1+m-low2+1, hig1, m+1, hig2);
   return p;
}
```

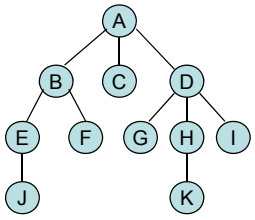
第六章　树和二叉树

目录

§6.4 树和森林

树的存储结构 • 双亲表示法

一、双亲表示法

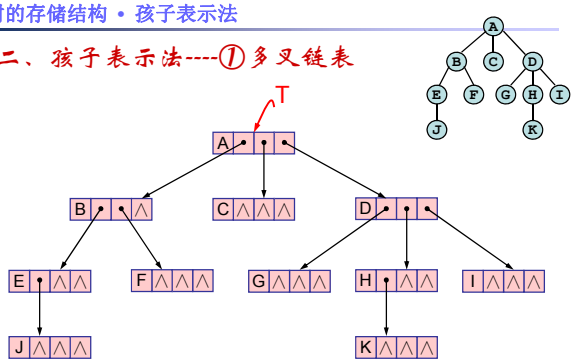


	data	parent
0		
1	E	2
2	B	9
3	J	1
4	D	9
5	K	6
6	H	4
7	G	4
8	F	2
9	A	0
10	C	9
11	I	4

root=9

树的存储结构 • 孩子表示法

二、孩子表示法----①多叉链表

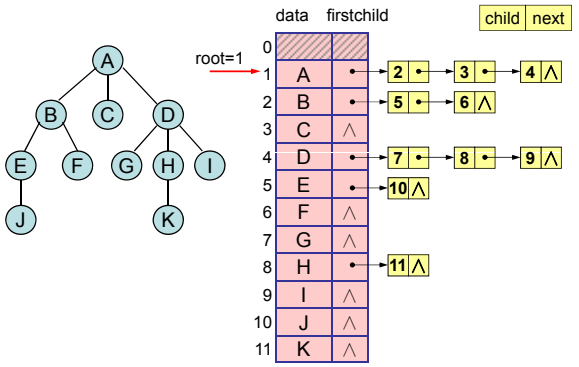


每个结点的指针域个数＝树的度

设树的度为K，空指针数＝n×K－(n－1)＝n×(K－1)＋1

树的存储结构 • 孩子表示法 • 孩子链表

二、孩子表示法----②孩子链表

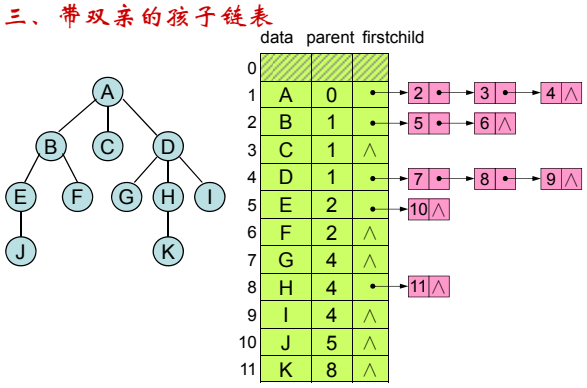


孩子链表存储类型的定义：

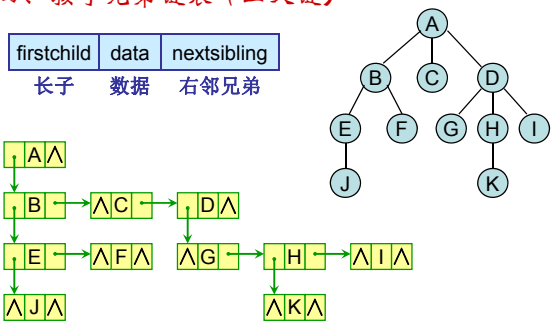
```
typedef  struct  CTNode {   //孩子结点
    int      child;
    struct  CTNode *next;
} *Childptr;

typedef struct  {   //表头结点   data | firstchild
    TElemType   data;
    ChildPtr    firstchild;
} CTBox;

typedef  struct {
    CTBox   nodes[maxsize];
    int     n, root;      //结点数和根的位置下标
} CTree;
```

树的存储结构 • 孩子表示法 • 双亲的带孩子链表

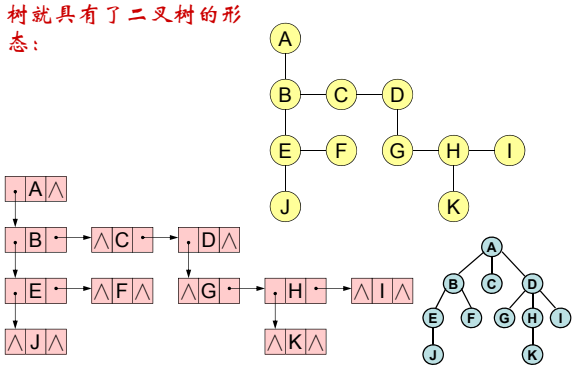
三、带双亲的孩子链表



四、孩子兄弟链表（二叉链）



树用二叉链表表示时，树就具有了二叉树的形态：

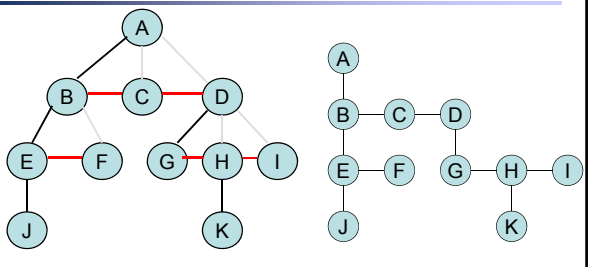


§6.4 树和森林

6.4.2 树、森林与二叉树的关系

1. 树转换成二叉树

树用孩子兄弟链表表示时，就已转化成二叉树了。一棵树可以唯一地转换成一棵右子树为空的二叉树。



树转换成二叉树的直观转换方法：

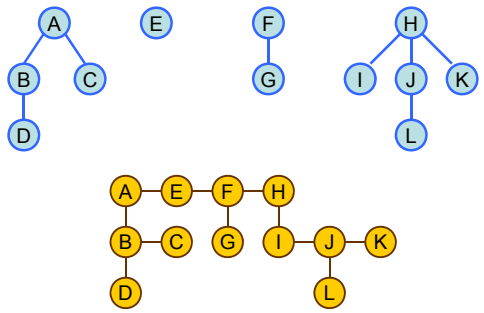
1. 把树中所有的兄弟之间用树枝相连；
2. 每个结点仅保留与长子的连线，删除结点与其余孩子的连线；
3. 以树根为圆心，顺时针转45度。

2. 森林转换成二叉树

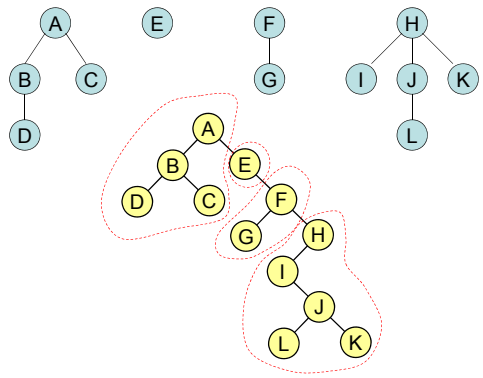
直观的转换方法：

1. 把森林中的每一棵树转换成二叉树
2. 将所有二叉树的树根用线相连
3. 以第一棵二叉树的树根为圆心，顺时针转45度

树、森林与二叉树的关系 • 森林转换成二叉树



树、森林与二叉树的关系 • 森林转换成二叉树



树、森林与二叉树的关系 • 森林转换成二叉树

森林转换成二叉树的形式定义：P138

森林F＝{ T1, T2, ..., Tm } → 二叉树B＝(root, LB,RB) :

(1) 若F为空，则B为空二叉树.

(2) 若F非空，则B的根root是T1的根；

B的左子树由T1的子树森林转换而成；

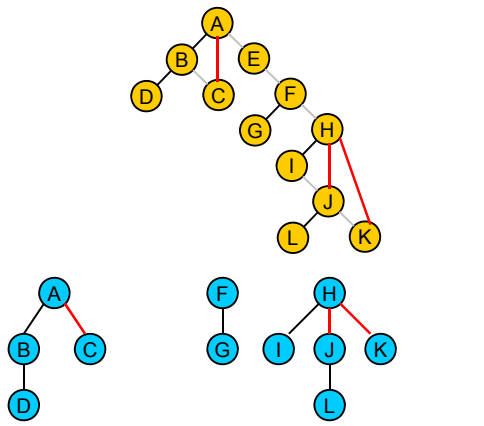
B的右子树由F中除第一棵树以外的剩余树转换而成.

树、森林与二叉树的关系 •二叉树转换成森林

3.二叉树转换成森林

直观的转换方法：

1. 在二叉树中，设B是A的左孩子，则把B的右孩子，右孩子的右孩子，...，都与A用线相连；
2. 去掉所有双亲到右孩子的连线



树、森林与二叉树的关系 • 二叉树换转成森林

二叉树转换成森林的形式定义：P138

二叉树B＝(root,LB, RB) → 森林F＝{ T1, T2, ..., Tm }

(1) 若B为空，则F为空

(2) 若B非空，则F中T1的根是B的根；

T1的根的子树森林由B的左子树LB转换而成；

F中除第一棵树以外的剩余树由B的右子树RB转换而成。

举例：计算树的高度，树用孩子兄弟链表表示

```
int TreeHeight (CSTree T) {
//树高＝1＋max(子树1高，子树2高，...，子树m高)
   int hmax=0, h;
   if(!T) return 0;        //空树
   p=T->firstchild;
   while(p){
      h=TreeHeight(p);     //求以p为根的子树高度
      if(h>hmax) hmax=h;
      p=p->nextsibling;    //p指向下一棵子树的根
   }
   return(1+hmax);
}
```

§6.4 树和森林

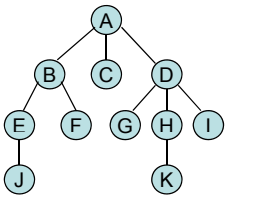
6.4.3 树和森林的遍历

1. 先根遍历树：

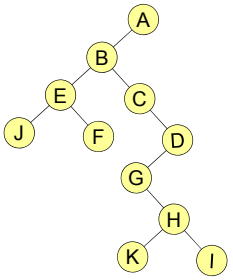
若树不空，则(1)访问根；
(2)从左至右先根遍历根的各个子树。

2. 后根遍历树：

若树不空，则(1)从左至右后根遍历根的各个子树。
(2)访问根。



先根序列：ABEJFCDGHKI
后根序列：JEFBCGKHIDA



先序序列：ABEJFCDGHKI
中序序列：JEFBCGKHIDA
后序序列：JFEKIHGDCBA

先根遍历树，等价于先序遍历由这棵树转换而成的二叉树；

后根遍历树，等价于中序遍历由这棵树转换而成的二叉树；

先序遍历树

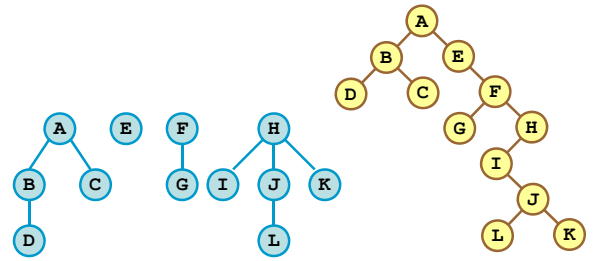
```
void preorder(CTBox tree[ ], int root) {
  //先序遍历树，树用孩子链表表示
    ChildPtr p;                    //辅助指针变量
    if(!root) return;
    printf("%c",tree[root].data);  //访问根
    p=tree[root].firstchild;
    while(p){//沿着孩子链从左至右先序遍历根的各个子树
       preorder(tree, p->child);
       p=p->next;
    }
}
```

3. 先序遍历森林：

若森林不空，则从左至右依次先序遍历各棵树。

4. 后序(中序)遍历森林：

若森林不空，则从左至右依次后序(中序)遍历各棵树。



先序序列：ABDCEFGHIJLK
后序序列：DBCAEGFILJKH

先序序列：ABDCEFGHIJLK
中序序列：DBCAEGFILJKH
后序序列：DCBGLKJIHFEA

举例：计算森林的深度

```
int TreeDepth(CSTree T) {
 //森林用孩子兄弟链表表示
 //深度=max(左子树所指森林的深度+1，右子树所指森林的深度)
   if(!T) return 0;  //空森林
   h1=TreeDepth(T->firstchild);
   h2=TreeDepth(T->nextsibling);
   return max(h1+1, h2);
}
```

第六章 树和二叉树

目录

§6.5 哈夫曼树及其应用

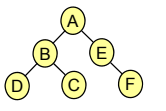
6.5.1 哈夫曼树的定义

路径：从一个结点到另一个结点所经过的分支

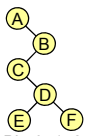
路径长度L：路径上分支的数目

树的路径长度PL：根到每个结点的路径长度之和

$$PL=\sum_{i=1}^{n} li$$ n—结点个数



h=3, PL=0+1+1+2+2+2=8



h=5, PL=0+1+2 +3+4+4=13

哈夫曼树及其应用 • 哈夫曼树的定义

结点的带权路径长度：结点的权值乘结点到根的路径长度$w_i \times l_i$

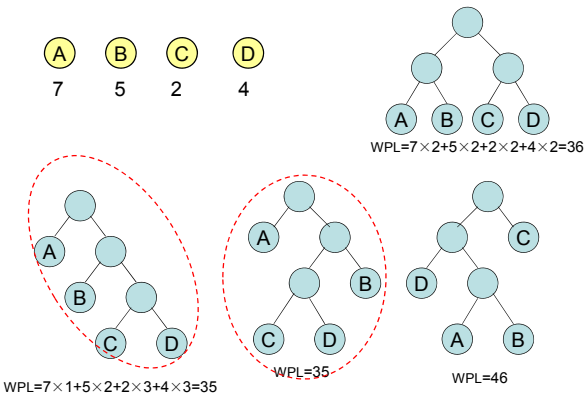
树的带权路径长度WPL：树中所有叶子带权路径长度之和

$$WPL = \sum_{i=1}^{n} wi \times li$$ n—树叶个数

哈夫曼树：由权值为{ w1,w2,...,wn}的n片叶子构成的所有二叉树中，WPL值最小的二叉树。

哈夫曼树又被称为最优二叉树

哈夫曼树及其应用 • 哈夫曼树的定义



哈夫曼树及其应用

§6.5 哈夫曼树及其应用

6.5.1 哈夫曼树的定义

6.5.2 哈夫曼树的构造

6.5.3 哈夫曼编码

哈夫曼树及其应用 • 哈夫曼树的构造

6.4.2 哈夫曼树的构造

1952年，Huffman提出了一个构造最优二叉树的一个精巧算法，被人们称为Huffman算法。

哈夫曼树及其应用 • 哈夫曼树的构造

Huffman算法思想：

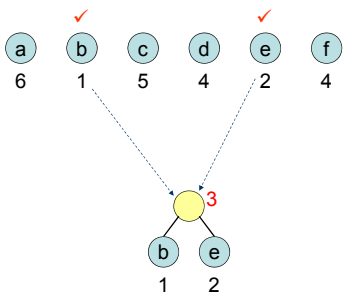
1. 将权值为w_1，w_2，...，w_n的n个叶子构成一个具有n棵树的森林：

F＝ { w_1 w_2 w_3 … w_n }

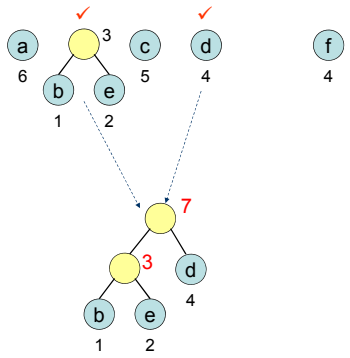
2. 从森林F中选取根权值最小的两棵树，分别作为左子树和右子树，再新添一个结点做为根，合并成一棵新的二叉树，新二叉树根的权值等于左、右子树根权值之和。

3. 重复2，直到F中只剩下一棵树为止，这棵树就是所求的Huffman树。

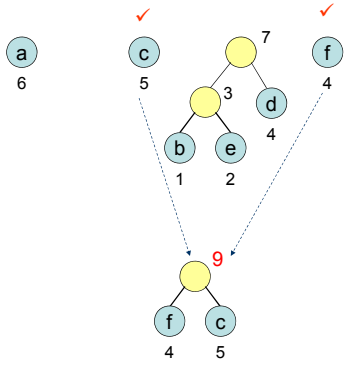
哈夫曼树及其应用 • 哈夫曼树的构造



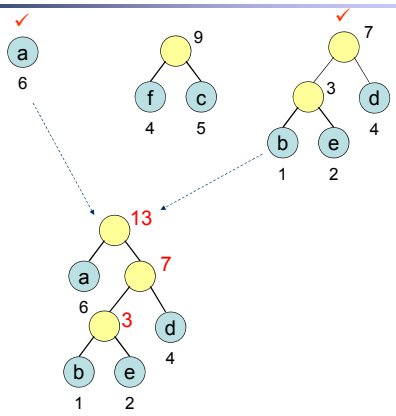
哈夫曼树及其应用 • 哈夫曼树的构造



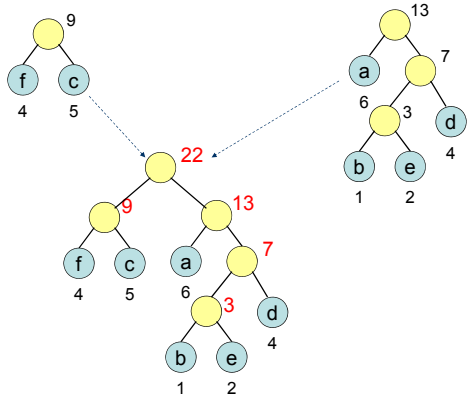
哈夫曼树及其应用 • 哈夫曼树的构造



哈夫曼树及其应用 • 哈夫曼树的构造

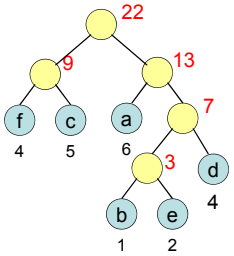


哈夫曼树及其应用 • 哈夫曼树的构造



哈夫曼树及其应用 • 哈夫曼树的构造

WPL=(1 + 2)×4 + 4×3 + (4 + 5 + 6)×2=54



哈夫曼树及其应用 • 哈夫曼树的构造

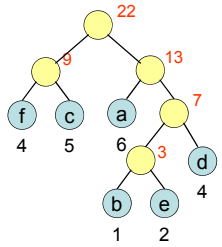
- 构造**n**个叶子的哈夫曼树需要经过**n-1**次合并，每次合并都要增加一个新结点。所以**n**个叶子的哈夫曼树上有且仅有**2n-1**个结点
- 哈夫曼树上不存在度为**1**的结点。我们把这种不存在度为**1**的结点的二叉树称为**严格二叉树**或**正则二叉树**。

$n_0=n_2+1$ （二叉树性质3）

$n=n_0+n_1+n_2=n_0+n_2=n_0+(n_0-1)=2*n_0-1$

哈夫曼树及其应用 • 哈夫曼树的构造

哈夫曼树的存储结构：



	chars
0	
1	a
2	b
3	c
4	d
5	e
6	f

	weight	parent	lchild	rchild
0				
1	6	10	0	0
2	1	7	0	0
3	5	9	0	0
4	4	8	0	0
5	2	7	0	0
6	4	9	0	0
7	3	8	2	5
8	7	10	7	4
9	9	11	6	3
10	13	11	1	8
2n−1= 11	22	0	9	10

哈夫曼树及其应用 • 哈夫曼树的构造

huffman树的存储类型定义：

```
typedef struct{
   unsigned int  weight;
   int  parent,lchild,rchild;
}HTNode, *HuffmanTree;
```

哈夫曼树及其应用 • 哈夫曼树的构造

在这种存储结构上实现Huffman算法：

1. 初始化
2. 进行n−1次合并：
   - 在parent==0的结点中选权值最小的两个结点，下标分别为s1和s2；
   - 填写s1和s2的双亲；
   - 填写新结点的权和左、右孩子的下标

	weight	parent	lchild	rchild
0				
1	6	0	0	0
S1 → 2	1	0	0	0
3	5	0	0	0
4	4	0	0	0
S2 → 5	2	0	0	0
6	4	0	0	0
➡ 7		0	0	0
8		0	0	0
9		0	0	0
10		0	0	0
2n−1= 11		0	0	0

```
void Create_Huffman(HuffmanTree &HT, int *w, int n){
      //构造哈夫曼树，数组w存放权值，n是叶子数
   m=n*2-1;
   HT=(HuffmanTree)malloc(sizeof(HTNode)*(m+1));
   for(i=1; i<=n; i++) HT[i]={w[i],0,0,0};  //初始化HT
   for(i=n+1; i<=m; i++) HT[i]={0,0,0,0};
   for(i=n+1; i<=m; i++){
      select( HT, i-1, s1);
           //在HT[1..i-1]选parent==0的最小权值
      HT[s1].parent=i ;
      select( HT, i-1, s2);
      HT[s2].parent=i ;
      HT[i].weight=HT[s1].weight+HT[s2].weight;
      HT[i].lchild=s1; HT[i].rchild=s2;
   } //for
}
```

哈夫曼树及其应用

§6.5 哈夫曼树及其应用

6.5.1 哈夫曼树的定义

6.5.2 哈夫曼树的构造

6.5.3 哈夫曼编码

哈夫曼树及其应用 • 哈夫曼编码和解码

6.4.3 哈夫曼编码和解码

发送：　电文　⇨⇨⇨ 编码 ⇨⇨⇨ 0,1 序列 (比特流)

接收：　0, 1序列　⇨⇨⇨ 解码 ⇨⇨⇨ 电文

例如：　电文＝“abcdedacafcfadcacfdaef”

字符集＝{a, b, c, d, e, f}

字符出现次数＝{6, 1, 5, 4, 2, 4}

哈夫曼树及其应用 • 哈夫曼编码和解码

电文＝“abcdedacafcfadcacfdaef”

字符集＝{ a, b, c, d, e, f }

编码方案：

	a	b	c	d	e	f	串长	特点
等长编码	000	001	010	011	100	101	66	串长太大
不等长编码	0	1	00	01	10	11	37	有多义性
前缀码	10	1100	01	1101	111	00	56	好

前缀码：任何字符的编码都不是其他字符编码的前缀

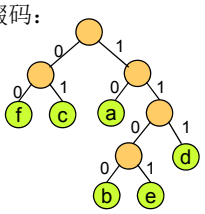
前缀码：10110001110111111010110011011...

哈夫曼树及其应用 • 哈夫曼编码和解码

电文＝“abcdedacafcfadcacfdaef”

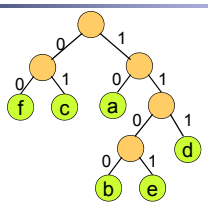
字符集＝{ a, b, c, d, e, f }

利用二叉树可以获得前缀码——以字符集中的字符为叶子，构造一棵二叉树；在左树枝上标0码，右树枝上标1码。从树根到树叶所经历的分支构成了相应叶子字符的前缀码：



a: 10
b: 1100
c: 01
d: 111
e: 1101
f: 00

哈夫曼树及其应用 • 哈夫曼编码和解码



a: 10
b: 1100
c: 01
d: 111
e: 1101
f: 00

- 由于n个叶子能够构造出很多形态各异的二叉树，因而会有多种前缀码方案，
- 取那种呢？
- 当然是能使电文比特流最短的编码方案。

哈夫曼树及其应用 • 哈夫曼编码和解码

比特流长 $= \sum_{i=1}^{n} ci \times li$

n—字符个数

ci—字符在电文中重复出现次数

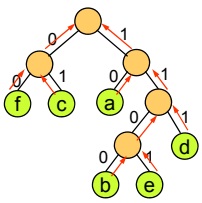
li—串长，根到叶子的路径长度

- 显然，如果ci是权，比特流长度就是二叉树的WPL。
- 哈夫曼树的WPL是最小的，故用哈夫曼树产生前缀码是**最优前缀码**，又称为**哈夫曼编码**。

哈夫曼树及其应用 • 哈夫曼编码和解码

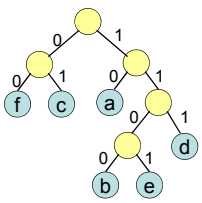
在哈夫曼树上获得前缀码的方法是：

从叶子开始，上溯找树根，经历左分支时产生0码，经历右分支时产生1码，由此可得到每个字符哈夫曼编码的逆序码。



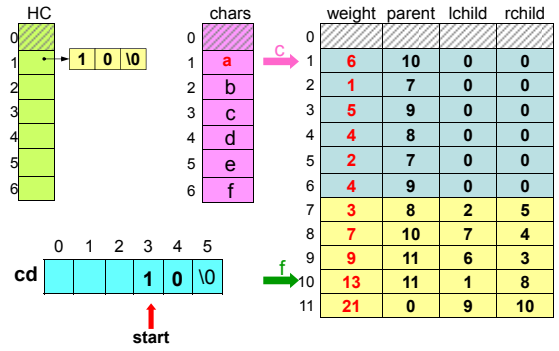
a: 10
b: 1100
c: 01
d: 111
e: 1101
f: 00

哈夫曼树及其应用 • 哈夫曼编码和解码

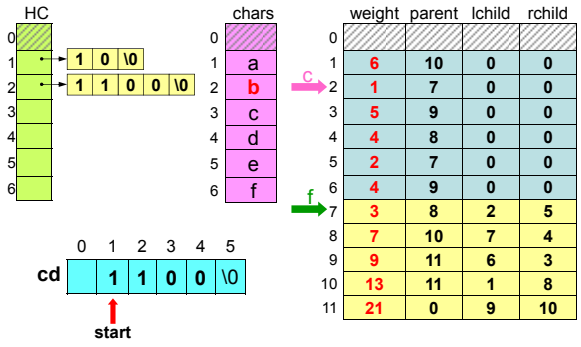


	chars	weight	parent	lchild	rchild
0					
1	a	6	10	0	0
2	b	1	7	0	0
3	c	5	9	0	0
4	d	4	8	0	0
5	e	2	7	0	0
6	f	4	9	0	0
7		3	8	2	5
8		7	10	7	4
9		9	11	6	3
10		13	11	1	8
11		22	0	9	10

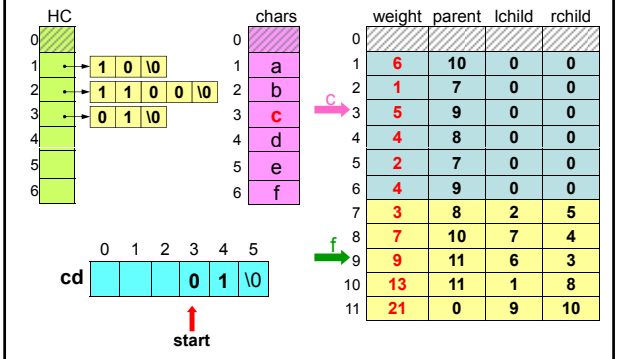
哈夫曼树及其应用 • 哈夫曼编码和解码



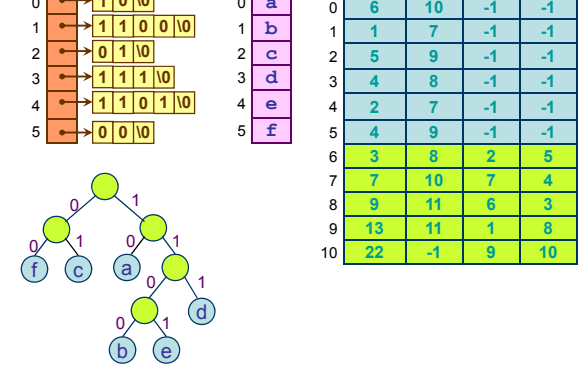
哈夫曼树及其应用 • 哈夫曼编码和解码



哈夫曼树及其应用 • 哈夫曼编码和解码



哈夫曼树及其应用 • 哈夫曼编码和解码




```
void Coding(HuffmanTree HT, char **&HC, int n){
      //已知哈夫曼树 产生每个字符的huffman编码
  char *cd=(char *)malloc(sizeof(char)*n); //辅助数组
  HC=(char **)malloc(sizeof(char *)*(n+1));
  cd[n-1]='\0';
  int start;
  for(i=1; i<=n; i++){
     start=n-1;
     for(c=i,f=HT[i].parent; f!=0; c=f,f=HT[f].parent)
        if(HT[f].lchild==c)cd[--start]='0'//左分支取0码
        else cd[--start]='1'; //右分支字符数组取1码
     HC[i]=(char*)malloc(sizeof(char)*(n-start));
     strcpy(HC[i], &cd[start]);
  }
  free(cd);
}
```

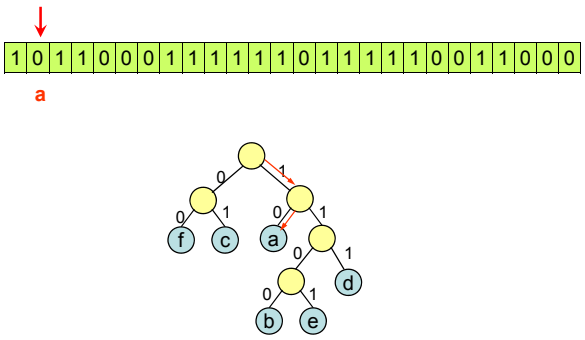
哈夫曼树及其应用 • 哈夫曼编码和解码

Huffman解码：

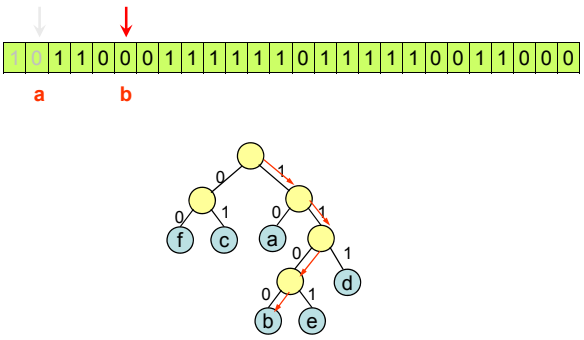
将比特流还原成电文也是在哈夫曼树上实现的：

- 从左至右扫描比特流；
- 自树根开始，逢0沿左链向下，逢1沿右链向下，直到遇到到叶子；
- 还原叶子字符；
- 再回到树根；重复上述过程，直至比特流被扫描完。

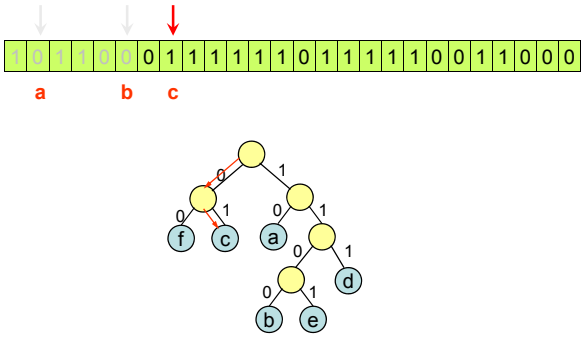
哈夫曼树及其应用 • 哈夫曼编码和解码



哈夫曼树及其应用 • 哈夫曼编码和解码



哈夫曼树及其应用 • 哈夫曼编码和解码




```
void Decoding(HuffmanTree HT, char *bits,
                              char *chars, int n){
  //Huffman解码，bits是比特流串，chars存放字符集字符
   char *p=bits;
   int root=n*2-1,  i=root;
   while(*p!='\0'){
      if(*p=='0')i=HT[i].lchild; //遇0左拐
      else i=HT[i].rchild;       //遇1右拐
      if(HT[i].lchild==0){
          printf("%c", chars[i]);   //打印叶子字符
          i=root;                   //回到树根
      }
      p++;
   }
   if(i!=root&&HT[i].lchild!=0)printf("Error\n");
}
```