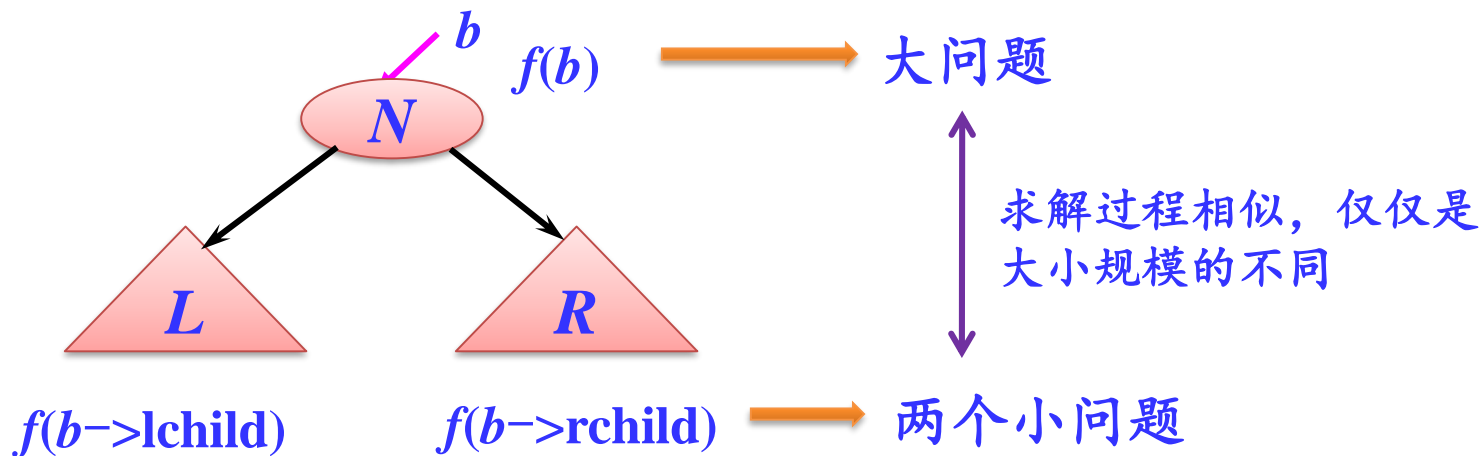


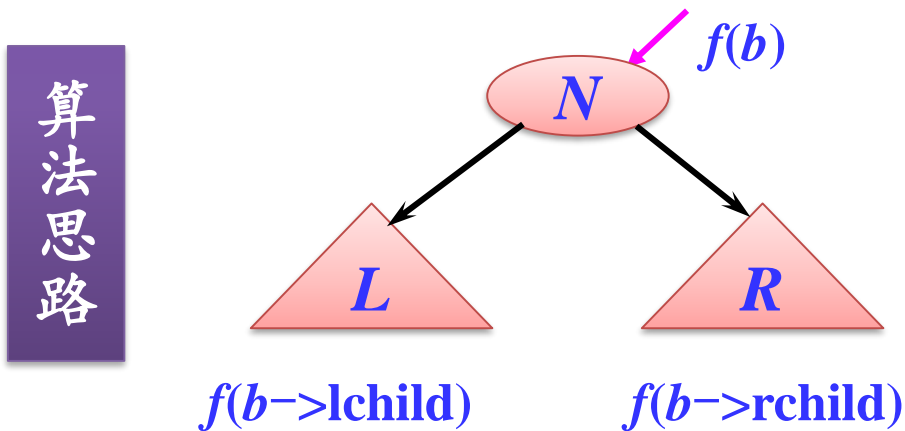
7.6 二叉树遍历的应用

7.6.1 二叉树3种递归遍历算法的应用

基本思路



【例7-4】 假设二叉树采用二叉链存储结构存储，设计一个算法，计算一棵给定二叉树的所有节点个数。



解： 计算一棵二叉树 **b** 中所有节点个数的递归模型 **$f(b)$** 如下：

$$f(b)=0$$

若 **$b=NULL$**

$$f(b)=f(b \rightarrow lchild)+f(b \rightarrow rchild)+1$$

其他情况

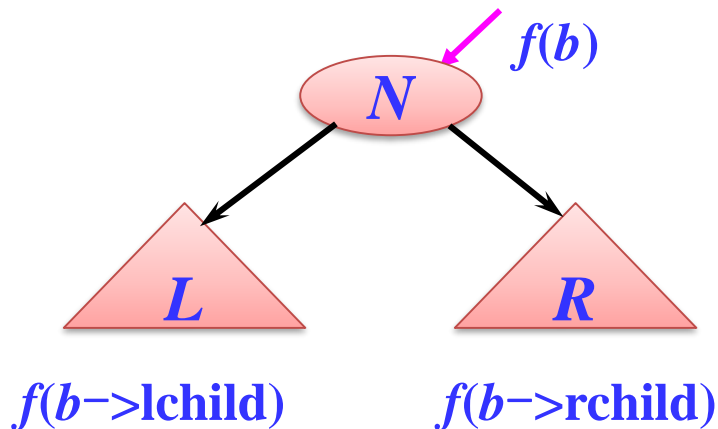
对应的递归算法如下：

```
int Nodes(BTNode *b)
{   int num1,num2;
    if (b==NULL)
        return 0;
    else
        return Nodes(b->lchild)+Nodes(b->rchild)+1
}
```

提示：本例算法可以基于后序遍历的思路。
先左子树、再右子树，最后根节点（计1），

【例7-5】 假设二叉树采用二叉链存储结构存储，设计一个算法，计算一棵给定二叉树的所有叶子节点个数。

算法思路



解： 计算一棵二叉树 b 中所有叶子节点个数的递归模型 $f(b)$ 如下：

$$f(b)=0$$

$$f(b)=1$$

$$f(b)=f(b \rightarrow lchild)+f(b \rightarrow rchild)$$

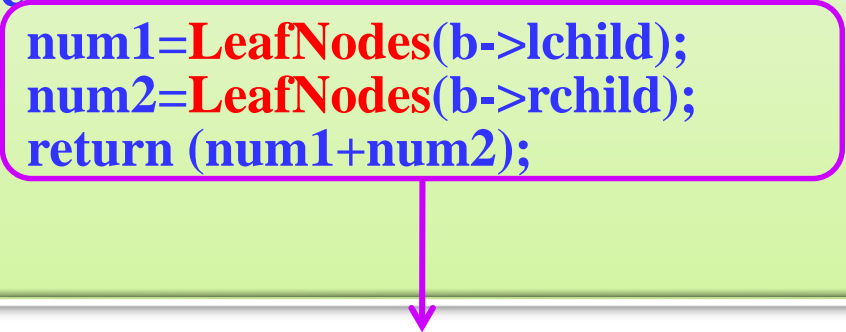
若 $b=NULL$

若 $*b$ 为叶子节点

其他情况

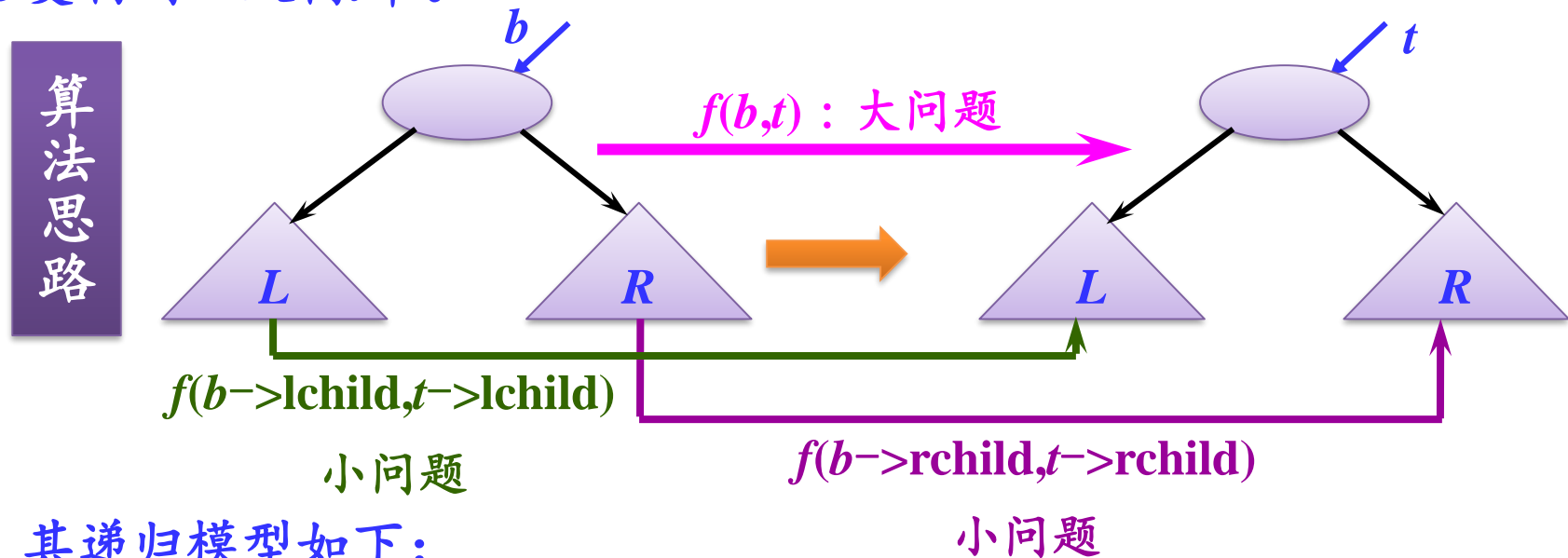
对应的递归算法如下：

```
int LeafNodes(BTNode *b)
{
    int num1,num2;
    if (b==NULL)
        return 0;
    else if (b->lchild==NULL && b->rchild==NULL)
        return 1;
    else
    {
        num1=LeafNodes(b->lchild);
        num2=LeafNodes(b->rchild);
        return (num1+num2);
    }
}
```



提示： 同样本例算法可以基于任何一种遍历算法。
先左子树，再右子树，最后根节点（计0）
是后序遍历的思路。

【例7-6】 假设二叉树采用二叉链存储结构，设计一个算法把二叉树***b***复制到二叉树***t***中。



其递归模型如下：

$f(b,t) \equiv t = \text{NULL}$

$f(b,t) \equiv$ 复制根节点**b*产生**t*节点;

$f(b \rightarrow lchild, t \rightarrow lchild);$

$f(b \rightarrow rchild, t \rightarrow rchild);$

若***b***=NULL

其他情况

对应的递归算法如下：

```
void Copy(BTNode *b,BTNode *&t)
{   if (b==NULL) t=NULL;
    else
    {   t=(BTNode *)malloc(sizeof(BTNode));
        t->data=b->data;           //复制一个根节点*t
        Copy(b->lchild,t->lchild); //递归复制左子树
        Copy(b->rchild,t->rchild); //递归复制右子树
    }
}
```

先根节点、再左子树，最后右子树，
是先序遍历的思路。

【例7-7】 设二叉树采用二叉链存储结构，设计一个算法把二叉树***b***的左、右子树进行交换。要求**不破坏原二叉树**。

解： 本题要求不破坏原有二叉树，实际上就是建立一个新的二叉树***t***，它交换了二叉树***b***的左、右子树。其递归模型如下：

$f(b,t) \equiv t = \text{NULL}$

若 $b = \text{NULL}$

$f(b,t) \equiv$ 复制根节点 $*b$ 产生 $*t$ 节点;

其他情况

$f(b \rightarrow \text{lchild}, t \rightarrow \text{rchild});$

$f(b \rightarrow \text{rchild}, t \rightarrow \text{lchild});$

对应的递归算法如下：

```
void Swap(BTNode *b,BTNode *&t)
{
    if (b==NULL) t=NULL;
    else
    {
        t=(BTNode *)malloc(sizeof(BTNode));
        t->data=b->data;           //复制一个根节点*t
        Swap(b->lchild,t->rchild); //递归交换左子树
        Swap(b->rchild,t->lchild); //递归交换右子树
    }
}
```

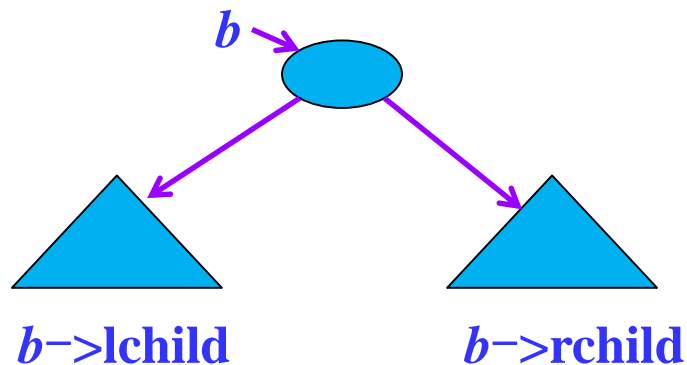
```
void Copy(BTNode *b,BTNode *&t)
{
    if (b==NULL) t=NULL;
    else
    {
        t=(BTNode *)malloc(sizeof(BTNode));
        t->data=b->data;           //复制一个根节点*t
        Copy(b->lchild,t->lchild); //递归复制左子树
        Copy(b->rchild,t->rchild); //递归复制右子树
    }
}
```

两个算法比较

【例7-8】 假设二叉树采用二叉链存储结构，设计一个算法 `Level()` 求二叉树 b 中值为 x 的节点的层次（假设所有节点值唯一）。

设 `Level(b, x, h)` 返回二叉树 b 中 `data` 值为 x 的节点的层次，其中 h 表示 b 所指节点的层数。

当在二叉树 b 中找到 `data` 值为 x 的节点，返回其层次（一个大于0的整数）；若没有找到，返回0。



初始调用: `Level($b, x, 1$)`

↑
 b 指
向根
节点

↑
根节
点的
层次
为1

☑ 空二叉树中找不到值为 x 的节点 \longrightarrow $\text{Level}(b, x, h)=0$ 若 $b=\text{NULL}$

☑ 若 b 指向值为 x 的节点 \longrightarrow $\text{Level}(b, x, h)=h$ 若 $b \rightarrow \text{data}=x$

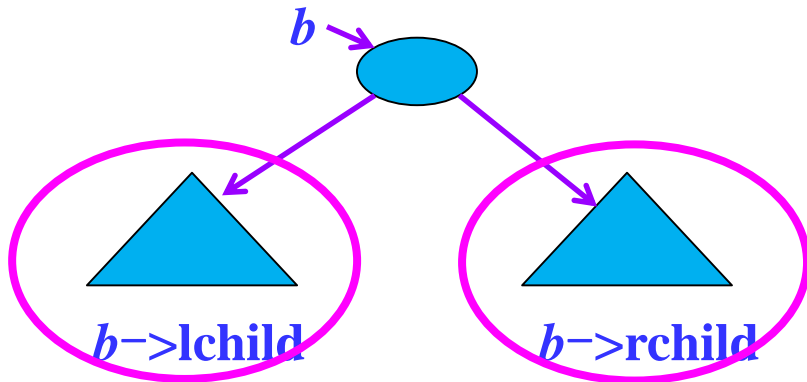
因为假设“ h 表示 b 所指节点的层次”

☑ 对于其他情况:首先在左子树中找。若找到了直接返回。

\longrightarrow $\text{Level}(b, x, h)=l$ 当 $l=\text{Level}(b \rightarrow \text{lchild}, x, h+1) \neq 0$

☑ 否则返回在右子树中的查找结果。

\longrightarrow $\text{Level}(b, x, h)=\text{Level}(b \rightarrow \text{rchild}, x, h+1)$ 其他情况



递归模型 $f(b, x, h)$ 如下:

$$f(b, x, h) = 0$$

当 $b = \text{NULL}$

$$f(b, x, h) = h$$

当 $b \rightarrow \text{data} = x$

$$f(b, x, h) = l$$

当 $l = f(b \rightarrow \text{lchild}, x, h+1) \neq 0$

$$f(b, x, h) = f(b \rightarrow \text{rchild}, x, h+1)$$

其他情况

对应的递归算法如下：

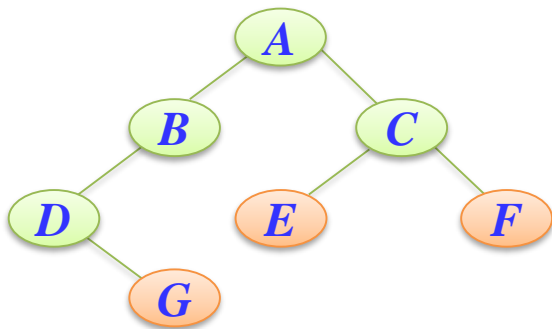
```
int Level(BTNode *b,ElemType x,int h)
//找到*p节点后h为其层次,否则为0
{   if (b==NULL) return 0;           //空树时返回0
    else if (b->data==x) return h;    //找到节点时
    else
    {   l=Level(b->lchild,x,h+1);      //在左子树中查找
        if (l==0)                     //左子树中未找到时在右子树中查找
            return Level(b->rchild,x,h+1);
        else return l;
    }
}
```

注意：基于先序遍历算法思想。

7.6.2 层次遍历算法的应用

【例7-10】 假设二叉树采用二叉链存储结构，设计一个算法输出从根节点到每个叶子节点的逆路径。

例如：



输出结果：

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

解：设计的队列为非环形队列qu，将所有已访问过的节点指针进队，并在队列中保存双亲节点的位置。

```
struct snode
{
    BTreeNode *node;           //存放当前节点指针
    int parent;                 //存放双亲节点在队列中的位置
} qu[MaxSize];                 //定义非环形队列
int front = rear = -1;         //置队列为空队列
```

当找到一个叶子节点时，在队列中通过双亲节点的位置输出根节点到该叶子节点的逆路径。

说明：类似于用队列求解迷宫问题。

对应算法如下：

```
void AllPath(BTNode *b)
{
    struct snode
    {
        BTNode *node;           //存放当前节点指针
        int parent;             //存放双亲节点在队列中的位置
    } qu[MaxSize];             //定义非环形队列
    BTNode *q;
    int front, rear, p;         //定义队头和队尾指针
    front=rear=-1;              //置队列为空队列
    rear++;
    qu[rear].node=b;            //根节点指针进入队列
    qu[rear].parent=-1;         //根节点没有双亲节点
}
```



```
while (front!=rear)           //队列不为空
{   front++;                  //front是当前节点*q在qu中的位置
    q=qu[front].node;         //队头出队列,该节点指针仍在qu中
    if (q->lchild==NULL && q->rchild==NULL) // *q为叶子节点
    {   p=front;               //输出*q到根节点的逆路径序列
        while (qu[p].parent!=-1)
        {   printf("%c->",qu[p].node->data);
            p=qu[p].parent;
        }
        printf("%c\n",qu[p].node->data);
    }
}
```

```
if (q->lchild!=NULL)           /*q节点有左孩子时将其进列
```

```
{    rear++;
```

```
    qu[rear].node=q->lchild;
```

```
    qu[rear].parent=front;      /*q的左孩子的双亲位置为front
```

```
}
```

```
if (q->rchild!=NULL)           /*q节点有右孩子时将其进列
```

```
{    rear++;
```

```
    qu[rear].node=q->rchild;
```

```
    qu[rear].parent=front;      /*q的右孩子的双亲位置为front
```

```
}
```

```
}
```

```
}
```

思考题

如何利用二叉树遍历算法求解实际应用问题？

——本讲完——