**线性表**

7. 设计一个算法，将元素x插入到一个有序（从小到大排序）顺序表的适当位置上，并保持有序性。

解：通过比较在顺序表 L 中找到插入 x 的位置 i，将该位置及后面的元素均后移一个位置，将 x 插入到位置 i 中，最后将 L 的长度增 1。对应的算法如下：

void Insert(SqList \*&L,ElemType x)

{

int i=0,j;

while (i<L->length && L->data[i]<x)

i++;

for (j=L->length-1;j>=i;j--)

L->data[j+1]=L->data[j];

L->data[i]=x;

L->length++;

}

8. 假设一个顺序表 L 中所有元素为整数，设计一个算法调整该顺序表，使其中所有小于零的元素放在所有大于等于零的元素的前面。

解：先让 i、j 分别指向顺序表 L 的第一个元素和最后一个元素。当 i<j 时循环：i 从前

向后扫描顺序表 L，找大于等于 0 的元素，j 从后向前扫描顺序表 L，找小于 0 的元素，当

i<j 时将两元素交换（思路参见《教程》例 2.4 的解法一）。对应的算法如下：

void fun(SqList \* \* &L)

{ int i=0,j=L->length-1;

while (i<j)

{ while (L->data[i]<0)

i++;

while (L->data[j]>=0)

j--;

if (i<j) //L->data[i]与 L->data[j]交换

swap(L->data[i], L->data[j]);

}

}

10. 在结点类型为 DLinkNode 的双链表中，给出将 p 所指结点（非尾结点）与其后继结点交换的操作。

答：将 p 所指结点（非尾结点）与其后继结点交换的操作如下：

q=p->next; //q 指向结点 p 的后继结点

if (q->next!=NULL) //从链表中删除结点 p

q->next->prior=p;

p->next=q->next;

p->prior->next=q; //将结点 q 插入到结点 p 的前面

q->prior=p->prior;

q->next=p;

p->prior=q;

11. 有一个线性表(a 1 ，a 2 ，…，a n )，其中 n≥2，采用带头结点的单链表存储，头指针为 L，每个结点存放线性表中一个元素，结点类型为（data，next），现查找某个元素值等于 x 的结点指针，若不存在这样的结点返回 NULL。分别写出下面 3 种情况的查找语句。要求时间尽量少。

（1）线性表中元素无序。

（2）线性表中元素按递增有序。

（3）线性表中元素按递减有序。

答：（1）元素无序时的查找语句如下：

p=L->next;

while (p!=NULL && p->data!=x) p=p->next;

if (p==NULL) return NULL;

else return p;

（2）元素按递增有序时的查找语句如下：

p=L->next;

while (p!=NULL && p->data<x ) p=p->next;

if (p==NULL || p->data>x) return NULL;

else return p;

（3）元素按递减有序时的查找语句如下：

p=L->next;

while (p!=NULL && p->data>x) p=p->next;

if (p==NULL || p->data<x) return NULL;

else return p;

12. 设计一个算法，将一个带头结点的数据域依次为 a 1 、a 2 、…、a n （n≥3）的单链表的所有结点逆置，即第一个结点的数据域变为 a n ，第 2 个结点的数据域变为 a n-1 ，…，尾结点的数据域为 a 1 。

解：首先让 p 指针指向首结点，将头结点的 next 域设置为空，表示新建的单链表为空表。用 p 扫描单链表的所有数据结点，将结点 p 采用头插法插入到新建的单链表中。对应的算法如下：

void Reverse( LinkNode \*&L L) )

{ LinkNode \*p=L->next,\*q;

L->next=NULL;

while (p!=NULL) //扫描所有的结点

{ q=p->next; //q 临时保存 p 结点的后继结点

p->next=L->next; //总是将 p 结点作为首结点插入

L->next=p;

p=q; //让 p 指向下一个结点

}

}

14. 设计一个算法在带头结点的非空单链表 L 中第一个最大值结点（最大值结点可能有多个）之前插入一个值为 x 的结点。

解：先在单链表 L 中查找第一个最大值结点的前驱结点 maxpre，然后在其后面插入值为 x 的结点。对应的算法如下：

void Insertbeforex( LinkNode \*&L,ElemType x)

{ LinkNode \*p=L->next,\*pre=L;

LinkNode \*maxp=p,\*maxpre=L,\*s;

while (p!=NULL)

{ if (maxp->data<p->data)

{ maxp=p;

maxpre=pre;

}

pre=p; p=p->next;

}

s=(LinkNode \*)malloc(sizeof(LinkNode));

s->data=x;

s->next=maxpre->next;

maxpre->next=s;

}

15. 设有一个带头结点的单链表 L，结点的结构为（data，next），其中 data 为整数元素，next 为后继结点的指针。设计一个算法，首先按递减次序输出该单链表中各结点的数据元素，然后释放所有结点占用的存储空间，并要求算法的空间复杂度为 O(1)。

解：先对单链表 L 的所有结点递减排序（思路参见《教程》例 2.8），再输出所有结点值，最后释放所有结点的空间。对应的算法如下：

void Sort( LinkNode \*&L) // 对单链表 L L 递减排序

{ LinkNode \*p,\*q,\*pre;

p=L->next->next; //p 指向第 2 个数据结点

L->next->next=NULL;

while (p!=NULL)

{ q=p->next;

pre=L;

while (pre->next != NULL && pre->next->data > p->data)

pre=pre->next;

p->next=pre->next; //在结点 pre 之后插入 p 结点

pre->next=p;

p=q;

}

}

void fun( LinkNode \*&L) // 完成本题的算法

{ printf("排序前单链表 L:");

DispList(L); //调用基本运算算法

Sort(L);

printf("排序后单链表 L:");

DispList(L); //调用基本运算算法

printf("释放单链表 L\n");

DestroyList(L); //调用基本运算算法

}

16. 设有一个双链表 h，每个结点中除有 prior、data 和 next 三个域外，还有一个访问频度域 freq，在链表被起用之前，其值均初始化为零。每当进行 LocateNode(h，x)运算时，令元素值为 x 的结点中 freq 域的值加 1，并调整表中结点的次序，使其按访问频度的递减序排列，以便使频繁访问的结点总是靠近表头。试写一符合上述要求的 LocateNode 运算的算法。

解：在 DLinkNode 类型的定义中添加整型 freq 域，给该域初始化为 0。在每次查找到一个结点 p 时，将其 freq 域增 1，再与它前面的一个结点 pre 进行比较，若 p 结点的 freq域值较大，则两者交换，如此找一个合适的位置。对应的算法如下：

bool LocateNode(D LinkNode \*h,ElemType x)

{ DLinkNode \*p=h->next,\*pre;

while (p!=NULL && p->data!=x)

p=p->next; //找 data 域值为 x 的结点 p

if (p==NULL) //未找到的情况

return false;

else //找到的情况

{ p->freq++; //频度增 1

pre=p->prior; //结点 pre 为结点 p 的前驱结点

while (pre!=h && pre->freq<p->freq)

{ p->prior=pre->prior;

p->prior->next=p; //交换结点 p 和结点 pre 的位置

pre->next=p->next;

if (pre->next!=NULL) //若 p 结点不是尾结点时

pre->next->prior=pre;

p->next=pre;

pre->prior=p;

pre=p->prior; //q 重指向结点 p 的前驱结点

}

return true;

}

}

**栈和队列**

11. 设从键盘输入一个序列的字符 a 1 、a 2 、„、a n 。设计一个算法实现这样的功能：若a i 为数字字符，a i 进队，若 a i 为小写字母时，将队首元素出队，若 a i 为其他字符，表示输入结束。要求使用环形队列。

解：先建立一个环形队列 qu，用 while 循环接收用户输入，若输入数字字符，将其进队；若输入小写字母，出队一个元素，并输出它；若为其他字符，则退出循环。本题算法如下：

void fun()

{

ElemType a,e;

SqQueue \*qu; //定义队列指针

InitQueue(qu);

while (true)

{ printf("输入 a:");

scanf("%s",&a);

if (a>='0' && a<='9') //为数字字符

{ if (!enQueue(qu,a))

printf(" 队列满,不能进队\n");

}

else if (a>='a' && a<='z') //为小写字母

{ if (!deQueue(qu,e))

printf(" 队列空,不能出队\n");

else

printf(" 出队元素:%c\n",e);

}

else

break; //为其他字符

}

DestroyQueue(qu);

}

12. 设计一个算法，将一个环形队列（容量为 n，元素下标从 0 到 n-1）的元素倒置。例如，图 3.2（a）中为倒置前的队列（n=10），图 3.2（b）中为倒置后的队列。

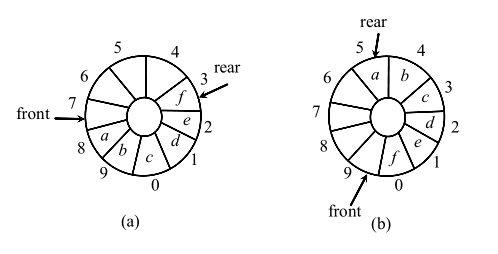


图 3.2 一个环形队列倒置前后的状态

解：使用一个临时栈 st，先将 qu 队列中所有元素出队并将其进栈 st，直到队列空为止。然后初始化队列 qu（队列清空），再出栈 st 的所有元素并将其进队 qu，最后销毁栈 st。对应的算法如下：

void Reverse(SqQueue \*&qu)

{ ElemType e;

SqStack \*st;

InitStack(st);

while (!QueueEmpty(qu)) //队不空时,出队并进栈

{ deQueue(qu,e);

Push(st,e);

}

InitQueue(qu); //队列初始化

while (!StackEmpty(st)) //栈不空时,出栈并将元素入队

{ Pop(st,e);

enQueue(qu,e);

}

DestroyStack(st);

}

13. 编写一个程序，输入 n（由用户输入）个 10 以内的数，每输入 i（0≤i≤9），就把它插入到第 i 号队列中。最后把 10 个队中非空队列，按队列号从小到大的顺序串接成一条链，并输出该链的所有元素。

解：建立一个队头指针数组 quh 和队尾指针数组 qut，quh[i]和 qut[i]表示 i 号（0≤i≤9）队列的队头和队尾，先将它们所有元素置为 NULL。对于输入的 x，采用尾插法将其链到 x号队列中。然后按 0～9 编号的顺序把这些队列中的结点构成一个不带头结点的单链表，其首结点指针为 head。最后输出单链表 head 的所有结点值并释放所有结点。对应的程序如下：

#include <stdio.h>

#include <malloc.h>

#define MAXQNode 10 //队列的个数

typedef struct node

{ int data;

struct node \*next;

} QNode;

void Insert(QNode \* quh [],QNode \* qut [],int x) //将 将 x x 插入到相应队列中

{ QNode \*s;

s=( QNode \*)malloc(sizeof( QNode )); //创建一个结点 s

s->data=x; s->next=NULL;

if (quh[x]==NULL) //x 号队列为空队时

{ quh[x]=s;

qut[x]=s;

}

else //x 号队列不空队时

{ qut[x]->next=s; //将 s 结点链到 qut[x]所指结点之后

qut[x]=s; //让 qut[x]仍指向尾结点

}

}

void Create(QNode \* quh [],QNode \* qut []) // 根据用户输入创建队列

{ int n,x,i;

printf("n:");

scanf("%d",&n);

for (i=0;i<n;i++)

{ do

{ printf("输入第%d 个数:",i+1);

scanf("%d",&x);

} while (x<0 || x>10);

Insert(quh,qut,x);

}

}

void DestroyList(QNode \*&head) // 释放单链表

{ QNode \*pre=head,\*p=pre->next;

while (p!=NULL)

{ free(pre);

pre=p; p=p->next;

}

free(pre);

}

void DispList(QNode \*head) // 输出单链表的所有 结点

{ printf("\n 输出所有元素:");

while (head!=NULL)

{ printf("%d ",head->data);

head=head->next;

}

printf("\n");

}

QNode \*Link(QNode \* quh [],QNode \* qut []) // 将非空队列链接起来并输出

{ QNode \*head=NULL,\*tail; //总链表的首结点指针和尾结点指针

int i;

for (i=0;i<MAXQNode;i++) //扫描所有队列

if (quh[i]!=NULL) //i 号队列不空

{ if (head==NULL) //若 i 号队列为第一个非空队列

{ head=quh[i];

Tail=qut[i];

}

else //若 i 号队列不是第一个非空队列

{ tail->next=quh[i];

tail=qut[i];

}

}

tail->next=NULL;

return head;

}

int main()

{ int i;

QNode \*head;

QNode \*quh[MAXQNode],\*qut[MAXQNode]; //各队列的队头 quh 和队尾指针 qut

for (i=0;i<MAXQNode;i++)

quh[i]=qut[i]=NULL; //置初值空

Create(quh,qut); //建立队列

head=Link(quh,qut); //链接各队列产生单链表

DispList(head); //输出单链表

DestroyList(head); //销毁单链表

return 1;

}

**数组**

10. 设计一个算法，计算一个三元组表表示的稀疏矩阵的对角线元素之和。

解：对于稀疏矩阵三元组表 a，从 a.data[0]开始查看，若其行号等于列号，表示是一个对角线上的元素，则进行累加，最后返回累加值。算法如下：

bool diagonal( TSMatrix a,ElemType &sum)

{ sum=0;

if (a.rows!=a.cols) //行号不等于列号，返回 false

{ printf("不是对角矩阵\n");

return false;

}

for (int i=0;i<a.nums;i++)

if (a.data[i].r==a.data[i].c) //行号等于列号

sum+=a.data[i].d;

return true;

}

**树和二叉树**

10. 假设二叉树中每个结点的值为单个字符，设计一个算法将一棵以二叉链方式存储的二叉树 b 转换成对应的顺序存储结构 a。

解：设二叉树的顺序存储结构类型为 SqBTree，先将顺序存储结构 a 中所有元素置为‘#’（表示空结点）。将 b 转换成 a 的递归模型如下：

f(b，a，i)  a[i]='#'; 当 b=NULL

f(b，a，i)  由 b 结点 data 域值建立 a[i]元素; 其他情况

f(b->lchild，a，2\*i);

f(b->rchild，a，2\*i+1)

调用方式为：f(b，a，1)（a 的下标从 1 开始）。对应的算法如下：

void Ctree(BTNode \*b b, , SqBTree a a ,int i)

{ if (b!=NULL)

{ a[i]=b->data;

Ctree(b->lchild,a,2\*i);

Ctree(b->rchild,a,2\*i+1);

}

else

a[i]='#';

}

11. 假设二叉树中每个结点值为单个字符，采用顺序存储结构存储。设计一个算法，求二叉树 t 中的叶子结点个数。

解：用 i 遍历所有的结点，当 i 大于等于 MaxSize 时，返回 0。当 t[i]是空结点时返回 0；当 t[i]是非空结点时，若它为叶子结点，num 增 1；否则递归调用 num1=LeftNode(t，2\*i)求出左子树的叶子结点个数 num1，再递归调用 num2=LeftNode(t，2\*i+1)求出右子树的叶子结点个数 num2，置 num+=num1+num2。最后返回 num。对应的算法如下：

int LeftNode(SqBTree t,int i)

{ //i 的初值为1

int num1,num2,num=0;

if (i<MaxSize)

{ if (t[i]!='#')

{ if (t[2\*i]=='#' && t[2\*i+1]=='#')

num++; //叶子结点个数增 1

else

{ num1=LeftNode(t,2\*i);

num2=LeftNode(t,2\*i+1);

num+=num1+num2;

}

return num;

}

else return 0;

}

else return 0;

}

12. 假设二叉树中每个结点值为单个字符，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 b 中的所有单分支结点个数。

解：计算一棵二叉树的所有单分支结点个数的递归模型 f(b)如下：

f(b)=0 若 b=NULL

f(b)=f(b->lchild)+f(b->rchild)+1 若 b 结点为单分支

f(b)=f(b->lchild)+f(b->rchild) 其他情况

对应的算法如下：

int SSonNodes(BTNode \*b)

{ int num1,num2,n;

if (b==NULL)

return 0;

else if ((b->lchild==NULL && b->rchild!=NULL) ||(b->lchild!=NULL && b->rchild==NULL))

n=1; //为单分支结点

else

n=0; //其他结点

num1=SSonNodes(b->lchild); //递归求左子树中单分支结点数

num2=SSonNodes(b->rchild); //递归求右子树中单分支结点数

return (num1+num2+n);

}

上述算法采用的是先序遍历的思路。

13. 假设二叉树中每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法求二叉树 b 中最小值的结点值。

解：设 f(b，min)是在二叉树 b 中寻找最小结点值 min，其递归模型如下：

f(b，min)  不做任何事件 若 b=NULL

f(b，min)  当 b->data<min 时置 min=b->data; 其他情况

f(b->lchild，min); f(b->rchild，min);

对应的算法如下：

void FindMinNode(BTNode \*b,char &min)

{ if (b->data<min)

min=b->data;

FindMinNode(b->lchild,min); //在左子树中找最小结点值

FindMinNode(b->rchild,min); //在右子树中找最小结点值

}

void MinNode(BTNode \*b) // 输出最小 结点 值

{ if (b!=NULL)

{ char min=b->data;

FindMinNode(b,min);

printf("Min=%c\n",min);

}

}

14. 假设二叉树中每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法将二叉链 b1 复制到二叉链 b2 中。

解：当 b1 为空时，置 b2 为空树。当 b1 不为空时，建立 b2 结点（b2 为根结点），置b2->data=b1->data；递归调用 Copy(b1->lchild，b2->lchild)，由 b1 的左子树建立 b2 的左子树；递归调用 Copy(b1->rchild，b2->rchild)，由 b1 的右子树建立 b2 的右子树。对应的算法如下：

void Copy(BTNode \*b1,BTNode \*&b2)

{ if (b1==NULL)

b2=NULL;

else

{ b2=(BTNode \*)malloc(sizeof(BTNode));

b2->data=b1->data;

Copy(b1->lchild,b2->lchild);

Copy(b1->rchild,b2->rchild);

}

}

15. 假设二叉树中每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，求二叉树 b 中第 k 层上叶子结点个数。

解：采用先序遍历方法，当 b 为空时返回 0。置 num 为 0。若 b 不为空，当前结点的层次为 k，并且 b 为叶子结点，则 num 增 1，递归调用 num1=LevelkCount(b->lchild，k，h+1)求出左子树中第 k 层的结点个数 num1，递归调用 num2=LevelkCount(b->rchild，k，h+1)求出右子树中第 k 层的结点个数 num2，置 num+=num1+num2，最后返回 num。对应的算法如下：

int LevelkCount(BTNode \*b,int k,int h)

{ //h 的初值为 1

int num1,num2,num=0;

if (b!=NULL)

{ if (h==k && b->lchild==NULL && b->rchild==NULL)

num++;

num1=LevelkCount(b->lchild,k,h+1);

num2=LevelkCount(b->rchild,k,h+1);

num+=num1+num2;

return num;

}

return 0;

}

int Levelk left (BTNode \*b,int k // 返回二叉树 b b 中第 k k 层上叶子 结点 个数

{

return LevelkCount(b,k,1);

}

16. 假设二叉树中每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，判断值为 x 的结点与值为 y 的结点是否互为兄弟，假设这样的结点值是唯一的。

解：采用先序遍历方法，当 b 为空时直接返回 false；否则，若当前结点 b 是双分支结点，且有两个互为兄弟的结点 x、y，则返回 true；否则，递归调用 flag=Brother(b->lchild，x，y)，求出 x、y 在左子树中是否互为兄弟，若 flag 为 true，则返回 true；否则递归调用Brother(b->rchild，x，y)，求出 x、y 在右子树中是否互为兄弟，并返回其结果。对应的算法如下：

bool Brother(BTNode \*b,char x,char y)

{ bool flag;

if (b==NULL)

return false;

else

{ if (b->lchild!=NULL && b->rchild!=NULL)

{ if ((b->lchild->data==x && b->rchild->data==y) ||

(b->lchild->data==y && b->rchild->data==x))

return true;

}

flag=Brother(b->lchild,x,y);

if (flag==true)

return true;

else

return Brother(b->rchild,x,y);

}

}

17. 假设二叉树中每个结点值为单个字符，采用二叉链存储结构存储。设计一个算法，采用先序遍历方法求二叉树 b 中值为 x 的结点的子孙，假设值为 x 的结点是唯一的。

解：设计 Output(p)算法输出以 p 为根结点的所有结点。首先在二叉树 b 中查找值为 x的结点，当前 b 结点是这样的结点，调用 Output(b->lchild)输出其左子树中所有结点，调用Output(b->rchild)输出其右子树中所有结点，并返回；否则，递归调用 Child(b->lchild，x)在左子树中查找值为 x的结点，递归调用Child(b->rchild，x)在右子树中查找值为 x的结点。对应的算法如下：

void Output(BTNode \*p) //输出以p为根结点的子树

{ if (p!=NULL)

{ printf("%c ",p->data);

Output(p->lchild);

Output(p->rchild);

}

}

void Child(BTNode \*b,char x) // 输出 x x 结点 的子孙

{ if (b!=NULL)

{ if (b->data==x)

{ if (b->lchild!=NULL)

Output(b->lchild);

if (b->rchild!=NULL)

Output(b->rchild);

return ;

}

Child(b->lchild,x);

Child(b->rchild,x);

}

}

18. 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 的左、右子树进行交换。要求不破坏原二叉树。并用相关数据进行测试。

解：交换二叉树的左、右子树的递归模型如下：

f(b，t)  t=NULL 若 b=NULL

f(b，t)  复制根结点 b 产生结点 t; 其他情况

f(b->lchild，t1); f(b->rchild，t2);

t->lchild=t2; t->rchild=t1

对应的算法如下（算法返回左、右子树交换后的二叉树）：

#include "btree.cpp" //二叉树基本运算算法

BTNode \*S S wap(BTNode \*b)

{ BTNode \*t,\*t1,\*t2;

if (b==NULL)

t=NULL;

else

{ t=( BTNode \*)malloc(sizeof( BTNode));

t->data=b->data; //复制产生根结点 t

t1=Swap(b->lchild);

t2=Swap(b->rchild);

t->lchild=t2;

t->rchild=t1;

}

return t;

}

或者设计成如下算法（算法产生左、右子树交换后的二叉树 b1）：

void Swap1(BTNode \*b,BTNode \*&b1)

{ if (b==NULL)

b1=NULL;

else

{ b1=( BTNode \*)malloc(sizeof( BTNode));

b1->data=b->data; //复制产生根结点 b1

Swap1(b->lchild,b1->rchild);

Swap1(b->rchild,b1->lchild);

}

}

设计如下主函数：

int main()

{ BTNode \*b,\*b1;

CreateBTree(b,"A(B(D(,G)),C(E,F))");

printf("交换前的二叉树:"); DispBTree(b); printf("\n");

b1=Swap(b);

printf("交换后的二叉树:"); DispBTree(b1); printf("\n");

DestroyBTree(b);

DestroyBTree(b1);

return 1;

}

程序执行结果如下：

交换前的二叉树:A(B(D(,G)),C(E,F))

交换后的二叉树:A(C(F,E),B(,D(G)))

19. 假设二叉树采用二叉链存储结构，设计一个算法判断一棵二叉树 b 的左、右子树是否同构。

解：判断二叉树 b1、b2 是否同构的递归模型如下：

f(b1，b2)=true b1=b2=NULL

f(b1，b2)=false 若 b1、b2 中有一个为空，另一个不为空

f(b1，b2)=f(b1->lchild，b2->lchild) 其他情况

& f(b1->rchild，b2->rchild)

对应的算法如下：

bool Symm(BTNode \*b b 1,BTNode \*b b 2) //判断二叉树b1 和b2是否同构

{ if (b1==NULL && b2==NULL)

return true;

else if (b1==NULL || b2==NULL)

return false;

else

return (Symm(b1->lchild,b2->lchild) & Symm(b1->rchild,b2->rchild));

}

bool Symmtree(BTNode \*b) // 判断二叉树的左、右子树是否同构

{ if (b==NULL)

return true;

else

return Symm(b->lchild,b->rchild);

}

20. 假设二叉树以二叉链存储，设计一个算法，判断一棵二叉树 b 是否为完全二叉树。

解：根据完全二叉树的定义，对完全二叉树按照从上到下、从左到右的次序遍历（层

次遍历）应该满足：

（1）某结点没有左孩子，则一定无右孩子。

（2）若某结点缺左或右孩子（一旦出现这种情况，置 bj=false），则其所有后继一定

无孩子。

若不满足上述任何一条，均不为完全二叉树（cm=true 表示是完全二叉树，cm=false 表

示不是完全二叉树）。对应的算法如下：

bool CompBT ree (BTNode \*b)

{ BTNode \*Qu[MaxSize],\*p; //定义一个队列，用于层次遍历

int front=0,rear=0; //环形队列的队头队尾指针

bool cm=true; //cm 为真表示二叉树为完全二叉树

bool bj=true; //bj 为真表示到目前为止所有结点均有左右孩子

if (b==NULL) return true; //空树当成特殊的完全二叉树

rear++;

Qu[rear]=b; //根结点进队

while (front!=rear) //队列不空

{ front=(front+1)%MaxSize;

p=Qu[front]; //出队结点 p

if (p->lchild==NULL) //p 结点没有左孩子

{ bj=false; //出现结点 p 缺左孩子的情况

if (p->rchild!=NULL) //没有左孩子但有右孩子,违反(1),

cm=false;

}

else //p 结点有左孩子

{ if (!bj) cm=false; //bj 为假而结点 p 还有左孩子，违反(2)

rear=(rear+1)%MaxSize;

Qu[rear]=p->lchild; //左孩子进队

if (p->rchild==NULL)

bj=false; //出现结点 p 缺右孩子的情况

else //p有左右孩子,则继续判断

{ rear=(rear+1)%MaxSize;

Qu[rear]=p->rchild; //将 p 结点的右孩子进队

}

}

}

return cm;

}