5-16

int MaxSize = 100;

char BTree[MaxSize];

void preorderNonrecursion(char BTree[], int length)

{

int stack[MaxSize]; //序号栈

int root,lchild,rchild;

root = 0;

stack.push(root);

while(!stack.empty()){

root = stack.top();

stack.pop();

cout << BTree[root];

lchild = root \* 2 + 1;

rchild = root \* 2 + 2;

if(rchild < length) stack.push(rchild);

if(lchild < length) stack.push(lchild);

}

}

5-17

void preorderNonrecursion(BTNode \* bt)

{

if(bt != NULL)

{

BTNode \* Stack[maxSize];

int top = -1;

BTNode \* p;

Stack[++top] = bt;

while(top != -1)

{

p = Stack[top--];

Visit(p);

if(p -> rchile != NULL)

Stack[++top] = p -> rchile;

if(p -> lchild != NULL)

Stack[++top] = p -> lchile;

}

}

}

5-18

递归方法

//preod:先序数组

//ps, pe 先序数组的首元素和末元素索引

//inod：中序数组

//is, ie 中序数组的首元素和末元素索引

bitree \* BPI(datatype preod[], datatype inod[], int ps, int pe, int is, int ie)

{

int m;

bitree \* p;

if(pe < ps)

return NULL;

p = (bitree\*)malloc(sizeof(bitree));

p -> data = preod[ps];

m = is;

while(inod[m] != preod[ps])

m++;

p -> lchild = BPI(preod, inod, ps + 1, ps + m - is, is, m - 1);

p -> rchild = BPI(preod, inod, ps + m - is + 1, pe, m + 1, ie);

return p;

}

5-19

思路：后序遍历非递归算法2

1.沿着根的左孩子，依次入栈，直到左孩子为空

2.取栈顶元素：若其右孩子不空且未被访问过，将右子树执行步骤1；否则，栈顶元素出栈并访问。

void PostOrder(BiTree T)

{

InitStack(S);

BiTree p = T;

BiTree r = NULL;

while(p || !IsEmpty(S))

{

if(p)

{

push(S, p);

p = p -> lchild;

}

else

{

GetTop(S, p);

if(p -> rchild && p -> rchild != r)

p = p -> rchild;

else

{

pop(S, p);

visit(p -> data);

r = p;

p = NULL;

}

}

}

}

typedef struct

{

BiTree t;

int tag;

}stack; // tag = 0 表示左子树被访问，tag = 1 表示右子树被访问

void Search(BiTree bt, ElemType x)

{

stack s[];

int top = 0;

while(bt != NULL || top > 0)

{

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

{

s[++top].t = bt;

s[top].tag = 0;

bt = bt -> lchild;

}

if(bt -> data == x)

{

for(i = 1; i <= top; i++)

{

vidit(s[i].t -> data);

exit(1);

}

}

while(top != 0 && s[top].tag == 1)

top--;

if(top != 0)

{

s[top].tag = 1;

bt = s[top].t -> rchild;

}

}

}

5-20

bool IsComplete(BiTree T)

{

InitQueue(Q);

if(!T)

return 1;

EnQUeue(Q, T);

while(! IsEmpty(Q))

{

DeQueue(Q, p);

if(p)

{

EnQUeue(Q, p -> lchild);

EnQUeue(Q, p -> rchild);

}

else

{

while(! IsEmpty(Q))

{

DeQueue(Q, p);

if(p)

return 0;

}

}

}

return 1;

}

5-21

void ExChangeTree(BiTree \*T) {

BiTree \*temp = new BiTree;

if (T)

{

temp = T->rchild;

T->rchild = T->lchild;

T->lchild = temp;

ExChangeTree(T->lchild);

ExChangeTree(T->rchild);

}

else

{

return;

}

}

5-22

BiTree PreOrderDe(BiTree T,ElemType e,int tag){

//tag作为辅助标志，判断当前结点是否需要删除

if(T)//递归终止条件

if(T->data!=e&&tag==0){//用来判断当前结点值是否为x或者是x的子孙结点

//T不是值为x的结点，且不是x结点的子孙结点

//只进行先序遍历

T->lchild=PreOrderDe(T->lchild,e,0);

T->rchild=PreOrderDe(T->rchild,e,0);

}else{//当前结点为要删除的结点

//T的子孙结点都需要删除，从这里开始，后续传入的tag值都将为1

T->lchild=PreOrderDe(T->lchild,e,1);

T->rchild=PreOrderDe(T->rchild,e,1);

free(T);

T=NULL;

}

return T; //设置这个返回当前结点主要是为了将x的父结点指向x的指针设置为NULL

}