//在链表L中的第i个位置之前插入新的元素e

SeqList ListInsert(SeqList L,int i,int e){

int j;

if(L.length==max) printf("表已经满了，不能插入\n");

else if(i<1||i>L.length) printf("插入的位置不正确\n");

else{

for(j=L.length-1;j>=i-1;j--) L.data[j+1]=L.data[j];

L.data[i-1]=e;

L.length++;

}

return L; }

//删除链表L中第i个元素，并由e返回其值

SeqList ListDelete(SeqList L,int i,ElemType e){

int j;

if(i<1||i>L.length-1) printf("删除位置不正确\n");

else{

e=L.data[i-1];

for(j=i;j<=L.length-1;j++) L.data[j-1]=L.data[j];

L.length--;

printf("%d已经删除\n",e);

}

return L;

}

//在链表中查找元素k

int LocateElem(SeqList L,int k){

int i=0;

while(i<L.length&&L.data[i]!=k) i++;

if(i<L.length){ return i+1; }

else return -1;

}

//求单链表的长度

int LinkedListLength(LinkedList L){

LinkedList p; int j=0; p=L->next;

while(p!=NULL){ j++; p=p->next; return j; } }

//在单链表中第i个数据元素之前插入数据元素e

LinkedList ListInsert(LinkedList L,int i,ElemType e){

LNode \*s; LinkedList p=L; int j=0;

int len=LinkedListLength(L);

if(i>len+1||i<=0){ printf("插入的位置不正确！");

return;

}

else{

while(j<i-1){ p=p->next; j++; }

s=(LinkedList)malloc(sizeof(LNode));

s->data=e;

s->next=p->next;

p->next=s;

return L;

} }

// 单链表中删除第i个元素

LinkedList ListDelete(LinkedList L,int i){

LNode \*s;

LinkedList p=L;

int j=0;

int len=LinkedListLength(L);

if(i>len||i<=0){ printf("删除的位置不正确！"); return; }

else{

while(j<i-1){ p=p->next; j++; }

s=p->next;

p->next=p->next->next;

free(s);

}

return L;

}

SeqList ListInsert(SeqList L, int i, int e)

{

int j;

if(L.length==max) printf("表满，不能插入\n");

else if(i<1||i>L.length) printf("插入位置不正确\n");

else{

for(j=L.length-1;j>=i-1;j--) L.data[j+1]=L.data[j];

L.data[i-1]=e;

L.length++;

}

return L;

}

//删除链表中第i个位置的元素，用e返回

SeqList ListDelete(SeqList L, int i, ElemType e)

{

int j;

if(i<1||i>L.length-1) printf("删除位置不正确\n");

else{

e=L.data[i-1];

for(j=1;j<=L.length-1;j++) L.data[j-1]=L.data[j];

L.length--;

printf("%d已被删除\n",e);

}

return L;

}

//进栈

Stack Push(Stack s,ElemType e){

if(s->top==MaxSize){ printf("栈满，不能插入！"); return s; }

s->data[s->top]=e;

s->top++;

return s;

}

//出栈

ElemType Pop(SqStack \*s){

ElemType e;

if(s->top==s->base) return '0';

s->top--;

e=s->data[s->top];

return e;

}

//取栈顶元素

int GetTop(SqStack \*s,ElemType e){

if(s->top==s->base) return 0;

e=s->data[s->top-1];

return 1;

}

//将数据元素e插入到循环队列Q的队尾

CirQueue Insert\_CirQueue(CirQueue Q,ElemType e)

{

if((Q.rear+1)%MaxSize==Q.front){

printf("队满，不能插入！\n\n");

return Q;

}

Q.Queue\_array[Q.rear]=e;

Q.rear=(Q.rear+1)%MaxSize; //队尾指针向前移动

return Q;

}

//将循环队列的对首元素出队

CirQueue Delete\_CirQueue(CirQueue Q,ElemType x){

if(Q.front==Q.rear){

printf("队空，不能删除！\n\n");

return Q;

}

x=Q.Queue\_array[Q.front];

Q.front=(Q.front+1)%MaxSize; //对首指针向前移动

return Q;

}

//构造3元组

TSMatrix CreatMat(TSMatrix t,ElemType A[N][N]){

int i,j,nums=0;

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

for(j=1;j<=N;j++)

if(A[i-1][j-1]!=0){

nums++;

t.data[nums].r=i; t.data[nums].c=j; t.data[nums].d=A[i-1][j-1];

}

t.data[0].r= N ; //1

t.data[0].c= N; //2

t.data[0].d= nums; //3

return t;

}

//矩阵的转置

TSMatrix FastTransMatrix(TSMatrix a){

TSMatrix b=a;

int p,q,col,k;

int num[MAX\_SIZE+1],copt[MAX\_SIZE+1];

if(a.data[0].d==0)

printf("the TSMatrix A=0\n");

else{

for(col=1;col<=a.data[0].c;++col) num[col]=0 ; //向量num[]初始化为0

for(k=1;k<=a.data[0].d;++k) ++num[a.data[k].c]; //求原矩阵的每一列的非0元素

for(copt[1]=1,col=2;col<=a.data[0].d;++col) copt[col]=copt[col-1]+num[col-1] ; //6

for(p=1;p<=a.data[0].d;++p){

col=a.data[p].c;

q= copt[col]; //7

b.data[q].r=a.data[p].c;

b.data[q].c=a.data[p].r;

b.data[q].d=a.data[p].d;

++copt[col] ; //8

}

}

return b;

}

//二叉树的先序遍历

void PreOrderTraverse(BiTree T)

{

if(T==NULL) return;

printf("%c ",T->data);

PreOrderTraverse(T->lchild);

PreOrderTraverse(T->rchild);

}

//二叉树的中序遍历

void InOrderTraverse(BiTree T)

{

if(T==NULL) return;

InOrderTraverse(T->lchild);

printf("%c ",T->data);

InOrderTraverse(T->rchild);

}

//二叉树的后序遍历

void PostOrderTraverse(BiTree T){

if(T==NULL) return;

PostOrderTraverse(T->lchild);

PostOrderTraverse(T->rchild);

printf("%c ",T->data);

}

//采用顺序存储主串s和模式t，若模式t在主串s中从第pos位置开始有匹配的子串，返回位置，否则返回-1

int IndexString(SString S,SString T,int pos){

int i=pos,j=1;

while(i<=S.str[0]&&j<=T.str[0]){

if(S.str[i]==T.str[j]){ i++; j++; }

else{ i=i-j+2; j=1; }

}

if(j>T.str[0]) return i-T.str[0];//2 匹配，输出位置

else return (-1);

}

void get\_next(SString T,int next[]){

int i=1,j=0;

next[1]=0;

while(i<T.str[0]){

if((j==0)||(T.str[i]==T.str[j]))

{

i++;

j++;

next[i]=j;//3

}

else j=next[j]; //4

}

}

//改进的模式匹配算法 （KMP）

int Index\_KMP(SString S,SString T,int pos)

{

int i=pos,j=1;

int T\_len=T.str[0];

int next[T\_len+1];

get\_next(T,next);

while(i<=S.str[0]&&j<=T.str[0])

{

if(j==0||S.str[i]==T.str[j])

{

i++;

j++;

}

else

{

j=next[j];

}

}

if(j>T.str[0])

return i-T.str[0];

else return (-1);

}

//中序遍历非递归

void InOrderTraverse(BiTree T)

{

BiTree Stack[MAX\_NODE],p=T;

int top=0,bool=1;

if(T==NULL)

printf("Binary Tree is Empty！\n");

else{

do{

while(p!=NULL)

{

Stack[++top]=p;//1

p=p->lchild;

}

if(top==0)

bool=0;

else{

p=Stack[top];

top--;//2

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

p=p->rchild;//3

}

}

while(bool!=0);

}

}

//后序遍历非递归

void PostOrderTraverse(BiTree T){

BiTree S1[MAX\_NODE],p=T;

int S2[MAX\_NODE],top=0,bool=1;

if(T==NULL) printf("Binary Tree is Empty！\n");

else{

do{

while(p!=NULL){ S1[++top]=p; S2[top]=0; p=p->lchild; }

if(top==0) bool=0;

else if(S2[top]==0){ p=S1[top]->rchild; S2[top]=1; }

else{

p=S1[top];

top--;

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

p=NULL;//6

}

}

while(bool!=0);

}

}

void prime(MGraph \*G,int v){

int i,j,k,min;

struct{

char adjvex;

int lowcost;

}closedge[10];

for(i=0;i<G->n;i++){

if(i!=v){

closedge[i].lowcost=G->edgs[v][i];

closedge[i].adjvex=G->vexs[v];

}

}

closedge[i].lowcost=TURE; //标记已经选过的顶点

for(i=1;i<G->n;i++){

min=INFINITY;

for(j=0;j<G->n;j++){

//求当前最短路径

if((closedge[j].lowcost!=TURE)&&(closedge[j].lowcost)<min){

min=closedge[j].lowcost;

k=j;

}

}

printf("(%c,%c,%d) ",closedge[k].adjvex,G->vexs[k],min);

closedge[k].lowcost=TURE;

for(j=0;j<G->n;j++){

if(closedge[j].lowcost!=TURE){

if(G->edgs[k][j]<closedge[j].lowcost){

closedge[j].lowcost=G->edgs[k][j];

closedge[j].adjvex=G->vexs[k];

}

}

}

}

printf("\n\n");

}

//顺序查找

int Seq\_Search(SSTable ST , int key)

{

int p ;

ST.elem[0].key=key; /\* 设置监视哨兵,失败返回0 \*/

for (p=ST.length; !EQ(ST. elem[p].key, key); p--);

return(p) ;

}

//折半查找

int Bin\_Search(SSTable ST,int key)

{

int Low=1,High=ST.length, Mid ;

while (Low<=High)

{

Mid=(Low+High)/2 ;

if(EQ(ST. elem[Mid].key, key)) return(Mid);

else if (LT(ST.elem[Mid].key, key)) Low=Mid+1;

else High=Mid-1;

}

return 0; /\* 查找失败 \*/

}

//简单插入排序

SSTable straight\_insert\_sort(SSTable L){

int i, j ;

for (i=2;i<=L.length;i++){

L.elem[0]=L.elem[i];

j=i-1; /\* 设置哨兵 \*/

while(LT(L.elem[0].key,L.elem[j].key) )

{

L.elem[j+1]=L.elem[j];

j--;

} /\* 查找插入位置 \*/

L.elem[j+1]=L.elem[0]; /\* 插入到相应位置 \*/

}

return L;

}

SSTable ShellInsert(SSTable L,int dk){

//1.前后记录位置的增量

int i,j;

for(i=dk+1;i<=L.length;++i){

if(LT(L.elem[i].key,L.elem[i-dk].key))

{

L.elem[0]=L.elem[i];

for(j=i-dk;j>0&&LT(L.elem[0].key,L.elem[j].key);j-=dk)

L.elem[j+dk]=L.elem[j];

L.elem[j+dk]=L.elem[0];

}

}

return L;

}

//希尔排序

SSTable ShellSort(SSTable L,int dlta[],int t){

//按增量序列dlat[0...t-1]对顺序表L做希尔排序

int k;

for(k=0;k<t;++k)

L=ShellInsert(L,dlta[k]); //一趟增量为dlat[k]插入排序

return L;

}

//冒泡排序

SSTable Bubble\_Sort(SSTable L){

int j,k,flag;

for (j=1;j<L.length;j++) // 共有n-1趟排序

{

flag=TRUE;

for(k=1;k<=L.length-j;k++) // 一趟排序

if (LT(L.elem[k+1].key, L.elem[k].key ) )

{

flag=FALSE ;

L.elem[0]=L.elem[k] ;

L.elem[k]=L.elem[k+1] ;

L.elem[k+1]=L.elem[0] ;

}

if(flag==TRUE)

break ;

}

return L;

}

SSTable simple\_selection\_sort(SSTable L){

int m,n,k;

for(m=1;m<=L.length;m++){

k=m;

for(n=m+1;n<=L.length;n++)

if(LT(L.elem[n].key,L.elem[k].key))

k=n;

if(k!=m){

L.elem[0]=L.elem[m];

L.elem[m]=L.elem[k];

L.elem[k]=L.elem[0];

}

}

return L;

}

int Partition(SSTable \*L , int low, int high)

{

int pivotkey;

L->elem[0]=L->elem[low] ;

pivotkey=L->elem[low].key;

while(low<high)

{

while (low<high&&L->elem[high].key>=pivotkey) --high;

L->elem[low]=L->elem[high];

while (low<high&&L->elem[low].key<=pivotkey) ++low;

L->elem[high]=L->elem[low];

}

L->elem[low]=L->elem[0];

return low;

}

void quick\_Sort(SSTable \*L , int low, int high){

int k;

if(low<high){

k=Partition(L,low,high);

quick\_Sort(L,low,k-1);

quick\_Sort(L,k+1,high);

}

}