1、顺序表有关算法

**#define maxsize 50**

**typedef struct{**

**ElemType data[maxsize];**

**int length;**

**}SqList;**

(1)在顺序表i位置插入元素x，成功返回1，失败返回-1

int SeqInsert(int \* seq,int i,int x){

int j;

if(seq->length == maxsize)

return -1; //空间已满

else if(i < 0 || i > seq->length - 1)

return -1; //位置无效

else{

for(j = i;j < seq->length;j++){

seq->data[j+1] = seq->data[j]; //元素后移

}

seq->data[i] = x;

seq->length++; //长度+1

return 1; //插入成功

}

}

(2)删除顺序表的第i个元素，值通过参数返回（DataType型）

int SeqDelete(int \* seq,int i,int \* px){

int j;

if(seq->length == maxsize)

return -1; //空间已满

else if(i < 0 || i > seq->length - 1)

return -1; //位置无效

else{

\*px = seq->data[i]; //将i位置的元素赋值到px

for(j = i;j < seq->length;j++){

seq->data[j] = seq->data[j+1]; //元素向前覆盖

}

seq->length--; //长度+1

return 1; //删除成功

}

}

(3)有一整型顺序表L，元素从位置1开始存放，下列算法实现以第一个元素为基准，

使其前面的元素都比它小，后面的元素均比它大。

int part(SeqList \* L){

int i,j;

int x;

x = L->data[1];

for(i = 2;i <= L->len;i++){

if(L->data[i] < x){

L->data[0] = L->data[i];

//将当前元素前面的元素后移 - 覆盖掉i元素，最前方留出一个空位插入

for(j = i-1;j >= 1;j--){

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

}

L->data[1] = L->data[0]; //移动i位置元素到最前方

}

}

}

2、单链表有关算法

**typedef struct LNode{**

**ElemType data;**

**struct LNode \* next;**

**}LNode,\*LinkList;**

(1)头插法为数组a的n个元素建立一个带头结点的单链表，并返回其头指针

LNode \* createList(int a[],int n){

LNode \* head,\*s;

int i;

head = (LNode \*)malloc(sizeof(LNode));

if(head->next == NULL)

return NULL;

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

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

s->data = a[n-i];

s->next = head->next;

head->next = s;

}

return head;

}

(2)尾插法为数组a的n个元素建立一个带头结点的单链表，返回其头结点

LNode \* createList1(int a[],int n){

LNode \* head,\* tail,\*s;

int i;

head = init(); //链表初始化，init()函数

if(head == NULL)

return NULL;

tail = head;

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

s = (LNode \*)malloc(sizeof(NodeType));

s->data = a[i]; //赋值

s->next = NULL; //将s结点赋NULL，尾插准备

tail->next = s; //s赋值给尾结点的后置结点

tail = s; //s做最尾部结点(tail指向s)

}

return head;

}

(3)已知整型带头结点的单链表H,下列算法实现链表元素逆置,若原链表中元素为1,2

3,4,5;倒置后变成5,4,3,2,1

//基本思想,将head->指空,把链表元素进行为头插插入到新的链表中

void reverse(LNode \* head){

LNode \* p,q;

p = head->next;

head->next = NULL;

while(p != NULL){

q = p;

p = p->next;

q->next = head->next;

head->next = q;

}

}

(4)利用原表结点,将该链表划分成两个链表,一个放正数和零,另一个放置负数

//思想:原表存放数据＞0的结点,将＜0的结点利用尾插法存入head1

LNode \* DivideList(LNode \* head){

LNode \*p,\*q,\*r,\*head1;

head1 = (LNode \*)malloc(sizeof(NodeType));

head1->next = NULL;

r = head1;

p = head;

while(p->next){

if(p->next->data < 0){

q = p->next;

p->next = q->next; //将结点脱链

r->next = q; //尾插

r = q;

}else{

p = p->next;

}

}

r->next = NULL;

return head1;

}

(5)设计算法删除单链表重复结点

void delNode(LNode \* head){

LNode \*p,\*q,\*r;

p = head->next;

if(p == NULL)

return;

while(p->next){

q = p;

while(q->next){

if(q->next->data == p->data){

r = q->next;

q-next = r->next;

free(r);

}

else{

q = q->next;

}

}

p = p->next;

}

}

(6)设计算法删除单链表所有元素

void delAllNode(LNode \* head){

LNode \*p;

p = head;

if(p->next == NULL)

return;

while(p->next){

if(p->next->data == x){

LNode \* r = p->next;

p->next = r->next;

free(r);

}else{

p = p->next;

}

}

}

(7)将两个生序排列的单链表合并为一个升序的单链表

LNode merge(LNode \* head1,LNode \* head2){

LNode \*p,\*head,\*tail;

head->next = NULL;

//带头结点

head1 = head1->next;

head2 = head2->next;

while(head1 && head2){

if(head1->data < head2->data){

p = head1;

head1 = head1->next;

}else{

p = head2;

head2 = head2->next;

}

p->next = NULL; //尾插最尾元素置空

if(!head)

head = tail = p; //head初始化指向了p的头部

else{

tail = tail->next = p; //尾插,tail不断后移到最末结点

}

}

//如果某一链表为空,将非空链表接在flist尾结点后边

if(head1)

p = head1;

else if(head)

p = head2;

tail->next = p; //尾插p结点

return head;

}

(8)将两个升序表合并为降序表

LNode merge(LNode \* head1,LNode \* head2){

LNode \*p,\*r,\*head,\*tail;

head->next = NULL;

//带头结点

head1 = head1->next;

head2 = head2->next;

while(head1 && head2){

if(head1->data < head2->data){

p = head1;

head1 = head1->next;

}else{

p = head2;

head2 = head2->next;

}

p->next = head->next;

head->next = p;

}

//链表head1非空,将head1赋值到head2上再进行头插

//head2非空,依旧对head2操作进行头插

if(head1)

head2 = head1;

while(head2){ //继续进行头插

p = head2;

head2 = head->next;

p->next = head->next;

head->next = p;

}

return head;

}

(9)将元素x插入到单链表元素elm之后;若不存在则插入到最末尾

void insertNode(LNode \* head,int x,int elm){

LNode \* p,\*q;

p = head;

while(p->next){

if(p->next->data == elm){

q = (LNode \*)malloc(sizeof(NodeType));

q->data = x;

q->next = p->next; //插入其之前

p->next = q;

return;

}

p=p->next;

}

//插入到末尾

q = (LNode \*)malloc(sizeof(NodeType));

q->data = x;

q->next = p->next; //插入其之前

p->next = q;

}

(10)将元素x插入到升序单链表中,使插入后单链表依然有序

void insertNode(LNode \* head,int x){

LNode \* p,\*q;

p = head;

while(p->next){

if(p->next->data > x){

q = (LNode \*)malloc(sizeof(NodeType));

q->data = x;

q->next = p->next; //插入其之前

p->next = q;

return;

}

p=p->next;

}

//插入到末尾,如果该元素是最大的

q = (LNode \*)malloc(sizeof(NodeType));

q->data = x;

q->next = p->next; //插入其之前

p->next = q;

}

(11)求单链表最大结点并返回该结点指针,若链表为空,返回空指针

LNode \* getMaxNode(LNode \* head){

LNode \* p,\*q,\*s;

p = head->next;

if(P == NULL)

return NULL; //返回空指针

q = (LNode \*)malloc(sizeof(NodeType));

q->data = p->data;

while(p){

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

q = p; //把最大结点赋值给p

}

p = p->next;

}

return q;

}

3、树有关算法

**①二叉树**

**typedef struct BTree{**

**ElemType data;**

**struct BTree \*lchild,\*rchild;**

**}BTree;**

**②线索二叉树**

**typedef struct ThreadNode{**

**ElemType data;**

**struct ThreadNode \* lchild,\*rchild;**

**int ltag,rtag;**

**}ThreadNode,\*ThreadNode;**

(1)统计树中度为2的结点数(统计分支节点)

int n = 0;

void count(BTree \* bt){

if(bt != NULL){

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

n++;

}

count(bt->lchild);

count(bt->rchild);

}

}

(2)查找树中x元素所在结点,找到就返回该结点指针,否则返回空指针 - 待理解

BTree search(BTree \* bt,int x){

BTree \*p = NULL;

if(bt){

if(bt->data == x)

return bt;

if(bt->lchild != NULL){

p = search(bt->child,x);

if(p)

return p;

}

if(bt->rchild != NULL){

p = search(bt->rchild,x);

if(p)

return p;

}

}

return NULL;

}

Btree \* Search(BStree \* t,int key){

if(t != NULL){

if(t->data == key) return t;

Search(t->lchild,key);

Search(t->rchild,key);

}

}

(3)设计算法统计叶子结点总数

int countLeaf(BTree \* bt){

if(bt == NULL){

return 0;

}

if(bt->lchild == NULL && bt->rchild == NULL){

return 1;

}else{

return countLeaf(bt->lchild) + countLeaf(bt->rchild);

}

}

(4)统计t树的结点总数

void getNodeCount(BTree \* bt){

int n1 = 0,n2 = 0;

if(bt == NULL)

return 0;

else{

n1 = count(bt->lchild); //从左子树遍历得到结点数

n2 = count(bt->rchild); //从右子树遍历得到结点数

return n1 + n2 + 1;

}

}

void Count(BTree \* bt){

if(t == NULL) return 0;

else

return Count(bt->lchild) + Count(bt->rchild) + 1;

}

(5)计算算法计算二叉树的深度

int getTreeDeepth(BTree \* bt){

int ld = 0,rd = 0;

if(!bt)

return 0; //深度为0

if(bt){

ld = getTreeDeepth(bt->lchild);

rd = getTreeDeepth(bt->rchild);

}

return ld > rd ? ld : rd;

}

(6)给定结点s,插入到以t为根的二叉排序树中

void insertNode(BTree \* t,BTree \* s){

if(s->data < t->data){

if(t->lchild == NULL){

t->lchild = s;

}else{

insertNode(t->lchild,s);

}

}else{

if(t->rchild == NULL){

t->rchild = s;

}else{

insertNode(t->rchild,s);

}

}

}

(7)二叉排序树的生成算法

int BT\_insert(BTree \* t,int k){

if(bt==NULL){ //原树为空,构建根节点

t = (BStree)malloc(sizeof(BStree));

t->key = k;

t->lchild = t->rchild = NULL;

return 1; //插入成功

}

else if(k == t->key)

return 0; //存在相同结点插入失败

else if(k < t->key) //插入到左子树

return BT\_insert(t->lchild,k);

else //插入到右子树

return BT\_insert(t->rchild,k);

}

void create\_BST(BTree \* t,int \* a,int n){

t = NULL;

int i = 0;

while(i < n){

BT\_insert(t,a[i]);

i++;

}

}

(8)二叉排序树查找算法

BTree \* bstreeSearch(BTree \* t,int x){

while(t != NULL && key != t->data){

if(x < t->data)

t = t->lchild;

else

t = t->rchild;

}

return t;

}

(9)假设二叉排序树t各个元素值为整型且均不相同,定义结点类型设计算法

按递减次序输出个各元素的值

typedef struct Node{

int data;

struct Node \* lchild,rchild;

}BTree;

void print(BTree \* bt){

if(bt != NULL)

{

print(bt->rchild);

printf("%5d",bt->data);

print(bt->lchild);

}

}

(10)树的先序遍历

void preOrder(BTree \* bt){

if(bt != NULL){

visit(bt); //访问根节点

preOrder(bt->lchild); //先遍历左子树

preOrder(bt->rchild); //先遍历右子树

}

}

(11)树的中序遍历

void inOrder(BTree \* bt){

if(bt != NULL){

inOrder(bt->lchild); //先遍历左子树

visit(bt); //访问根节点

inOrder(bt->rchild); //先遍历右子树

}

}

(12)树的后序遍历

void postOrder(BTree \* bt){

if(bt != NULL){

postOrder(bt->lchild); //先遍历左子树

postOrder(bt->rchild); //先遍历右子树

visit(bt); //访问根节点

}

}

(13)层序遍历

void LevelOrder(BTree \* t){

InitQueue(Q);

BTree p;

EnQueue(Q,t);

while(!IsEmpty(Q)){

DeQueue(Q,p);

visit(p);

if(p->lchild != NULL){

EnQueue(Q,p->lchild);

}

if(p->rchild != NULL){

EnQueue(Q,p->rchild);

}

}

}

3、栈和队列

**①顺序栈**

**typedef struct{**

**ElemType data[MaxSize];**

**int top;**

**}SqStack;**

**②链式栈**

**typedef struct LinkNode{**

**ElemType data;**

**struct LinkNode \* next;**

**}\*LStack;**

**③顺序队列**

**typedef struct{**

**ElemType data[MaxSize];**

**int front,rear;**

**}SqQueue; //循环队列也使用这种存储结构**

**④链式队列**

**typedef struct{**

**ElemType data;**

**struct LinkNode \*next;**

**}LinkNode;**

**typedef struct{**

**LinkNode \*front,\*rear;**

**}LinkQueue;**

(1)循环队列的入队 队尾入

int EnQueue(SqQueue & Q,ElemType x){

if((Q.read + 1)%MaxSize == Q.front)

return 0;

Q.data[Q.rear] = x;

Q.rear = (Q.rear + 1)%MaxSize;

return 1;

}

(2)循环队列的出队 队头出

int DeQueue(SqQueue & Q,ElemType &x){

if(Q.read == Q.front)

return 0;

x = Q.data[Q.front];

Q.front = (Q.front + 1)%MaxSize;

return 1;

}

(3)顺序栈的入栈

bool Push(SqStack & s,ElemType x){

if(S.top == MaxSize - 1) return false;

S.data[++S.top] = x;

return true;

}

(4)顺序栈的出栈

bool Pop(SqStack &s,ElemType &x){

if(S.top == -1) return false;

x = S.data[S.top--];

return true;

}

(5)链式栈的入栈

int Push(LStack \* s,ElemType x)

{

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

p->data = x;

p->next = s->top; //插入

s->top = p;

}

(6)链式栈的出栈

int pop(LStack \* s,ElemType &x)

{

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

p = s->top;

x = p->data;

s->top = s->top->next; //脱链

free(p);

}

4、图

**①邻接矩阵法存储**

**#define MaxVertexNum 100 //最大顶点数**

**typedef char VertexType; //顶点类型定义**

**typedef int EdgeType; //边类型定义**

**typed struct{**

**VertexType Vex[MaxVertexNum]; //顶点表**

**EdgeType Edge[MaxVertexNum][MaxVertexNum]; //邻接矩阵**

**int vexnum,arcnum; //顶点数、弧数**

**}MGraph;**

**②邻接表法存储**

**#define MaxVertexNum 100 //最大顶点数**

**typedef struct ArcNode{**

**int adjvex; //弧所指向的顶点位置**

**struct ArcNode \* next; //指向下一条弧的指针**

**}ArcNode;**

**typedef struct VNode{ //表结点**

**VertexType data; //顶点信息**

**ArcNode \* first; //指向第一条依附于该顶点的弧的指针**

**}VNode,AdjList[MAX\_VERTEX\_NUM];**

**typedef struct{**

**AdjList vertices; //邻接表**

**int vexnum,arcnum; //图的顶点数和弧数**

**}ALGraph;**

(1)图的初始化(邻接矩阵)

void CreateGraph(MGraph &G){

int i,j,k,weight;

int gn,ge;

printf("请输入顶点数和边数:");

scanf("%d %d",&Gn,&Ge);

printf("请输入顶点:");

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

scanf("%d",&vexs[i]);

for(i = 0;i < Ge;i++){

for(j = 0;j < Gn;j++){

G.AdjMatrix[i][j] = infinity;

}

}

printf("请输入边的顶点下标i,j以及权重w :");

for(k = 0;k < Ge;k++){

scanf("%d %d %d",&i,&j,&weight);

G.AdjMatrix[i][j] = weight;

G.AdjMatrix[i][j] = G.AdjMatrix;

}

}

(2)广度优先遍历

bool visit[MAX\_VERTEX\_NUM];

void BFST(Graph G){ //访问标记数组

for(i = 0;i < G.vexnum;i++)

visit[i] = false; //访问标记数组初始化

InitQueue(Q); //初始化辅助队列

for(i = 0;i < G.vexnum;i++){ //从0号结点开始遍历

if(!visit[i])

BFS(G,i);

}

}

void BFS(Graph G,int v){

visit(v); //访问初始结点v

visited(v) = true; //置访问标记

EnQueue(Q,v);

while(!IsEmpty(Q)){

DeQueue(Q,v);

//检测v所有邻接结点

for(w = FirstNeighbor(G,v),w >= 0;w = NextNeighbor(G,v,w))

if(!visited[w]){ //w为v当前未访问的邻接结点

visit(w);

**visited**[w] = true;

EnQueue(Q,w);

}

}

}

(3)深度优先遍历

bool visit[MAX\_VERTEX\_NUM];

void DFST(Graph G){ //访问标记数组

for(i = 0;i < G.vexnum;i++)

visit[i] = false; //访问标记数组初始化

for(i = 0;i < G.vexnum;i++){ //从0号结点开始遍历

if(!visit[i])

DFS(G,i);

}

}

void DFS(Graph G,int v){

visit(v); //访问初始结点v

visited(v) = true; //置访问标记

for(w = FirstNeighbor(G,v),w >= 0;w = NextNeighbor(G,v,w))

if(!visited[w]) DFS(G,w); //w为v当前未访问的邻接结点

}

}

(4)计算顶点i的出度(邻接矩阵)

int getOutDegree(MGraph G,VertexType i){

int j,sum = 0;

for(j = 0;j < G.vernum;j++){

if(AdjMatrix[i][j] == 1){

sum++;

}

}

return sum;

}

(5)计算顶点i的入度(邻接矩阵)

int getInDegree(MGraph G,VertexType i){

int j,sum = 0;

for(j = 0;j < G.vernum;j++){

if(AdjMatrix[j][i] == 1){

sum++;

}

}

return sum;

}

5、排序

(1)简单选择排序

void SelectSort(int \* a,int n){

int i,j,k;

int temp;

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

k = i;

//循环得到待排序序列的最小元素

for(j = i+1;j < n;j++){

if(a[k] > a[j])

k = j;

}

//交换两个元素

temp = a[i];

a[i] = a[k];

a[k] = temp;

}

}

(2)冒泡排序升序排序算法

void bSort(int \* a,int max){

int i,j,flag;

for(i = 0;i < max - 1;i++){

flag = 0;

for(j = 0;j < max - i - 1;j++){

if(a[j] < a[j+1])

{

int temp = a[j];

a[j] = a[j+1];

a[j+1] = temp;

flag = 1;

}

}

if(!flag){ //如果没有发生交换表示后续序列已经有序,结束循环

return 0;

}

}

}

(3)直接插入排序进行升序排序的算法 - 在已排序序列找位置

void in\_sort(int \* a,int n){

int i,j,temp; //temp充当了哨兵

for(i = 1;i < n;i++){ //这里是从第二个元素开始比

temp = a[i];

j = i - 1;

//如果temp小于其之前的元素,让这个元素后移,直到找到合适位置

while(j >= 0 && temp < a[j]){ //降序只需要改 temp > a[j]

a[j+1] = a[j]; //后移

--j;

}

a[j+1] = temp; //插入

}

}

void in\_sort(int \* a,int n){

int i,j,temp;

for(i = 1;i < n;i++){ //这里加if(a[i] < a[i+1])升序可以简化算法

temp = a[i];

for(j = i - 1;j >= 0;j--)

{

a[j+1] = a[j];

}

a[j+1] = temp;

}

}

(4)快速排序算法

①第一种算法

int partition(int \* a,int low,int high){

int pivot = a[low];

while(low < high){

while(low < high && a[high] >= pivot) --high;

a[low] = a[high];

while(low < high && a[low] <= pivot) ++low;

a[high] = a[low];

}

a[low] = pivot;

return low;

}

void quickSort(int \* a,int low,int high){

if(low < high){

int pos = partition(a,low,high);

quickSort(a,low,pos-1);

quickSort(a,pos+1,high);

}

}

②第二种算法

void quicksort(int a[],int left,int right)

{

    int i,j,temp;

    i=left;

    j=right;

    temp=a[left];

    if(left>right)

        return;

    while(i!=j)

    {

        while(a[j]>=temp&&j>i)

            j--;

        if(j>i)

            a[i++]=a[j];

        while(a[i]<=temp&&j>i)

            i++;

        if(j>i)

            a[j--]=a[i];

    }

    a[i]=temp;

    quicksort(a,left,i-1);

    quicksort(a,i+1,right);

}

6、查找

(1)顺序表中二分查找的非递归算法

int search(int \* a,int max,int x){

int low,mid,high;

low = 0;

high = max - 1;

while(low <= high){

mid = (high+low)/2;

if(x < a[mid]){

high = mid-1;

}

else if(x > a[mid]){

low = mid + 1;

}else{

return mid;

}

}

return 0;

}

(2)顺序表二分查找的递归算法

int search(int \* a,int x;int low,int high){

int mid;

if(low > high)

return - 1;

mid = (low + high)/2;

if(x == a[mid]){

return mid;

}

if(x < a[mid])

return (search(a,x,low,mid-1));

return (search(a,x,mid+1,high))

}

(3)二叉排序树查找算法

BTree \* bstreeSearch(BTree \* t,int key){

while(t != NULL && key != t->data){

if(key < t->data) t = t->lchild;

else t = t->rchild;

}

return t;

}