线性表

* 1. 顺序表 随机存储 可直接访问某个节点的值
  2. 链性表 顺序存储 只知道头节点的位置，后续的节点的访问需要依次遍历
* 顺序表的一些操作
  1. 初始化与创建

#include

#include

#include

#include

#define SIZE 10 //初始分配量

#define LISTINCREMENT 3 //分配增量

typedef struct{

int \*elem; //存储空间基址

int length; //当前表长

int listsize; //当前分配的存储容量

}Sqlist; //之后再定义顺序表的时候类型为Sqlist

void Init(Sqlist &L){

L.elem = (int \*)malloc(SIZE\*sizeof(int)); //一个表头地址

if(!L.elem) exit(0); //表的存储空间分配失败，无法进行之后的任何操作直接退出

L.length = 0; //表中一开始没有元素，初始长度为0

L.listsize = SIZE //表的初始大小

}

void Create(Sqlist &L,int n){

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

cin>>\*L.elem;

L.length++;

L.elem++; //此处如果不写的话，直接当数组做，可以写L.elem[i]

}

}

//没有特别要求(创建新表存储两个表交并集,并且要求原来两表都在的情况下),将两个函数合并

void Init\_and\_Creat(Sqlist &L，int n){

L.elem = (int \*)malloc(SIZE\*sizeof(int));

if(!L.elem) exit(0);

L.length = 0;

L.listsize = SIZE

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

cin>>\*L.elem;

L.length++;

L.elem++;

}

}

int main(){

int n;cin>>n;Sqlist L; //n为要输入数据的个数

Init\_and\_Create(L,n);

return 0;

}

* 1. 输出

void Printf(SqList&L,int n) { //此处的n可有可无

L.elem = L.elem - L.length; //指针的头必须回到0位置否则没办法正确打印

int i = 0,t=L.length;

cout << "表的元素如下：" << "\t";

for (t; t >= 1; t--) {

cout << \*(L.elem) << "\t";

L.elem++;

}

printf("\n");

}

* 1. 插入

void Insert(SqList& L, int i, int e) {

int\* newbase;int \* q, \* p;

if (iL.length + 1) cout<<"错误";

if (L.length >= L.listsize) { //存储空间不足，再次分配

newbase = (int\*)(realloc)(L.elem, (L.listsize + LISTINCREMENT) \* sizeof(int));

if (!newbase) exit(0);

L.elem = newbase; L.listsize += LISTINCREMENT;

}

q = (L.elem - 1 - (L.length - i)); //q为要插入的位置

p = L.elem - 1; //p为表尾

for (p; p >= q; p--)

\*(p + 1) = \*p; //插入前所有的数据往后移动

\*q = e; //插入

++L.length; //插入后表的长度加一

++L.elem; //使得elem始终保持在表的尾巴

}

* 1. 删除

void Delete(SqList&L,int i,int &e) {

int\* p, \* q;

if (iL.length) cout << "错误";

p = (L.elem - 1 - (L.length-i)); //p为要删除的元素的位置

e = \*p; //用e接收删除的数字

q = L.elem - 1; //q为表最后的元素位置

for (p; p < q;p++)

\*p = \*(p+1); //删除就是从指定位置开始向前覆盖

L.length--; //删除后表的长度-1

L.elem--; //使得elem始终保持在表的尾巴

}

* 1. 置空

void Clear(Sqlist &L){

L.elem = L.elem - L.length;

if(!L.elem) cout<<"错误";

L.length = 0; //表长为0即为空表

}

* 1. 销毁

void Destory(Sqlist &L){

L.elem = L.elem - L.length;

if(!L.elem) cout<<"错误";

free(L.elem)

}

* 1. 查找

void Locate(SqList L, int e) {

int\* p, t;p = L.elem - L.length;

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

if (\*p == e) {

cout << t; break;

}

\*p = \*(p++);

}

if (t > L.length) cout << "表中没有该数字";

}

其他的操作和数组差不多

1. 链性表
   1. 链性表的初始化和创建

#include

#include

#include

using namespace std;

typedef int ElemType;

typedef struct Node {

ElemType data; // 数据域

struct Node\* next; // 指针域

}LNode, \* LinkList;

void Init(LinkList& L) {

L = (LinkList)malloc(sizeof(LNode));

L->next = NULL; //建立一个带头节点的链表

}

void CreatList(LinkList& L, int n) {

LinkList rear = L;

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

LinkList p = (LinkList)malloc(sizeof(Node));

p->next = NULL;

cin >> p->data;

rear->next = p;

rear = p;

}

}

void Printf(LinkList L) {

LinkList p = L->next;

while (p != NULL) {

cout << p->data << " ";

p = p->next;

}

cout << "\n";

}

int main(){

int n;cin>>n;LinkLIst L;

Init(L);CreatList(L);

Printf(L);

return 0;

}

* 1. 输出

void Printf(LinkList L) {

LinkList p = L->next;

while (p != NULL) {

cout << p->data << " ";

p = p->next;

}

cout << "\n";

}

* 1. 插入

bool Insert(LinkList& L, int i, int e) {

LinkList s = (LinkList)malloc(sizeof(Node));

LinkList p = L; int j = 0;

while (p != NULL && j < i-1) {

j++; //向后遍历到要删除的地方

p = p->next;

}

if (!p || j > i - 1) {

cout << "Insert Location Error" << "\n";

return false;

}//下面的顺序不能改变，变了之后链表后面的数据丢失，即断链

s->data = e; //建立一个新的节点储存新数据的值

s->next = p->next; //将新数据加入到链表之中，s的next是插入节点后的节点

p->next = s; //将s与插入之前的节点相连接

return true;

}

* 1. 删除

bool Delete(LinkList& L, int i,int &e) {

LinkList p = L; LinkList q;int j = 0;

while (p != NULL && j < i - 1) {

j++;

p = p->next;

}

if (!(p->next) || j > i - 1) {

cout << "Delete Location Error" << "\n";

return false;

}

q = p->next;

p->next = q->next; //可以是p=p->next->next;

e = q->data;

free(q); //结束后q不存在了要释放 free(p->next)

return true;

}

* 1. 求长度

void Length(LinkList L) {

int cnt = 0;

LinkList p = L->next;

while (p != NULL) {

cnt++； p = p->next;

}

cout << cnt << "\n";

}

* 1. 销毁

void Destroy(LinkList &L){

while(L){ //必须一个节点一个节点去free

q=L.next;

free(L);

L=q;

}

}

1. 顺序栈
   1. 初始化和创建

#include

#include

#include

using namespace std;

#define SIZE 100

#define INCREMENT 10

typedef struct {

int\* base;

int\* top;

int size;

}SqStack;

int n;

void Init(SqStack &S) { //初始化

S.base = (int\*)malloc(SIZE \* sizeof(int));

if (!S.base) exit(0);

S.top = S.base;

S.size = SIZE;

}

* 1. 入栈

void Push(SqStack &S,int e) { //入栈

int\* newbase;

if (S.top - S.base >= S.size) {

newbase = (int\*)realloc(S.base, (S.size + INCREMENT) \* sizeof(int));

if (!newbase) exit(0);

S.base = newbase;

S.top = S.base + S.size;

S.size += INCREMENT;

}

S.top++ = e; //对于栈，此处是对S.top赋值需要加

}

* 1. 出栈

void Pop(SqStack& S, int& e) { //出栈：从栈顶删除 此处其实没删除，只是指针前移

if (S.top == S.base) cout<<"栈为空";

e = \*--S.top;

n--;

}

* 1. 取栈顶元素

void GetTop(SqStack& S, int& e) {

if (S.top == S.base) cout << "栈为空" << endl;

e = \*(S.top - 1);

}

* 1. 清空

void Clear(SqStack& S) {

if (!S.base) cout << "栈不存在";

S.top = S.base;

cout << "栈已经置空";

cout << endl;

}

* 1. 判断空

void JudgeEmpty(SqStack& S) {

if (!S.base) cout << "栈不存在";

if (S.top == S.base) cout << "栈为空";

else cout << "栈不为空";

cout << endl;

}

* 1. 输出

void Printf(SqStack& S, int n) {

SqStack p; p = S;

if (S.top == S.base) cout << "栈为空";

else {

//for (int i = 0; i < n; i++) cout << \*p.base++ << " "; //从栈底开始输出

for (int i = 0; i < n; i++) cout << \*--p.top << " "; //从栈顶开始输出

}

cout << endl;

}

1. 链栈(特殊的链表)
   1. 初始化和创建

typedef struct StackNode{

int data;

struct StackNode\* next;

}SNode,\*LinkStack;

void Init(LinkStack& S) {

S = NULL;

}

* 1. 入栈

void Push(LinkStack& S, int e) { //入栈

LinkStack p = (LinkStack)malloc(sizeof(SNode)); //生成新节点

p->data = e;

p->next = S; //将新结点插入栈顶

S = p; //修改栈顶指针为p

}

* 1. 出栈

void Pop(LinkStack &S,int &e) { //出栈

LinkStack p;

if (S == NULL)cout << "栈为空" << endl;

e = S->data;

p = S; //用p临时保存栈顶元素空间，以备释放

S = S->next; //修改栈顶指针

free(p); //释放原栈顶元素的空间

n--;

}

* 1. 清空

void Clear(LinkStack&S) {

if (S == NULL)cout << "栈已经为空";

else S = NULL;

cout << "栈已经置空";

cout << endl;

}

* 1. 判空

void JudgeEmpty(LinkStack& S) {

if (S == NULL)cout << "栈为空";

else cout << "栈不为空";

cout << endl;

}

* 1. 取栈顶元素

void GetTop(LinkStack& S, int& e) {

if (S == NULL)cout << "栈为空" << endl;

e = S->data;

}

* 1. 输出

void Printf(LinkStack S) {

LinkStack p = S;

while(p!=NULL){

cout << p->data << " ";

p = p->next;

}

cout << endl;

}

1. 链队列
   1. 初始化和创建

typedef struct QNode {

int data;

struct QNode\* next;

}QNode,\*QueuePtr;

typedef struct {

QueuePtr front;

QueuePtr rear;

}LinkQueue;

int Init(LinkQueue& Q) {

Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode));

Q.front->next = NULL;

return 1;

}

* 1. 入队

void En(LinkQueue& Q, int e) { //入队

QueuePtr p = (QueuePtr)malloc(sizeof(QNode));

p->data = e; p->next = NULL;

Q.rear->next = p; //将新节点插入到队尾

Q.rear = p; //修改队尾指针

}

* 1. 出队

int De(LinkQueue& Q, int& e) { //出队:删除Q的队头元素，用e返回值

if (Q.front == Q.rear) return 0;

QueuePtr p = Q.front->next; //p指向队头元素结点

e = p->data; Q.front->next = p->next; //修改链队列头结点指针

if (Q.rear == p)Q.rear = Q.front;//对于链队列只有一个元素节点的情况要同时修改队尾指针

free(p);

return 1;

}

* 1. 取队列头元素

void GetFirst(LinkQueue Q) {

if (Q.front == Q.rear) cout<<"空";

cout << Q.front->next->data << endl;;

}

* 1. 输出

void Printf(LinkQueue Q) {

QueuePtr p = Q.front->next;

while (p) {

cout << p->data << " ";

p = p->next;

}

cout << endl;

}

* 1. 判空

int JudgeEmpty(LinkQueue Q) {

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

cout << "Y" << endl;

return 0;

}

cout << "N" << endl;

return 1;

}

* 1. 清空

int Destroy(LinkQueue& Q) { //从前往后释放空间

while (Q.front) {

Q.rear = Q.front->next;

free(Q.front);

Q.front = Q.rear;

}

return 1;

}

1. 循环队列(顺序队列)
   1. 初始化和创建

#define Maxsize 100

//typedef struct { //将front、rear设置成指针，操作会变得更加复杂

// int\* base;

// int\* front;

// int\* rear;

//}SqQueue;

//int Init(SqQueue& Q) {

// Q.base = (int\*)malloc(sizeof(int));

// if (!Q.base)exit(0);

// Q.front = Q.rear = Q.base;

//}

//int En(SqQueue& Q, int e) {

// int \*p = Q.rear;

// if (Q.rear - Q.base == Maxsize - 1)p = Q.base;

// else p++;

// if (p = Q.front)return 0;

// \*Q.rear = e; Q.rear = p;

//}

typedef struct {

int\* base;

int front; //在非空队列中，头指针始终指向队头元素，尾指针始终指向队尾元素的下一位置

int rear;

}SqQueue;

int Init(SqQueue& Q) {

Q.base = (int\*)malloc(sizeof(int));

if (!Q.base)exit (0);

Q.front = Q.rear = 0;

}

* 1. 入队

int En(SqQueue& Q, int e) {

if ((Q.rear + 1) % Maxsize == Q.front)return 0; //判断是否满了

Q.base[Q.rear] = e;

Q.rear = (Q.rear + 1) % Maxsize; //每次插入新元素时尾指针+1

}

* 1. 出队

int De(SqQueue& Q, int e) {

if (Q.rear == Q.front)return 0; //判断是否为空

e = Q.base[Q.front];

Q.front = (Q.front + 1) % Maxsize; //每次删除元素时头指针+1

return 1;

}

* 1. 取队列头元素

int GetFirst(SqQueue Q) {

if (Q.rear == Q.front)return 0;

cout << Q.base[Q.front] << endl;

return 1;

}

* 1. 判空

int JudgeEmpty(SqQueue Q) {

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

cout << "Y" << endl;

return 0;

}

cout << "N" << endl;

return 1;

}

* 1. 输出

void Printf(SqQueue Q) {

for (int p = Q.front; p < Q.rear; p++)

cout << Q.base[p] << " ";

cout << endl;

}

1. 树
   1. 初始化和创建

char ch;

typedef struct BiTNode {

int data;

struct BiTNode\* lc, \* rc;

}BiTNode,\*BiTree;

void Create(BiTree &T) {

cin >> ch;

if (ch == '\*')T = NULL;

else {

T = (BiTNode\*)malloc(sizeof(BiTNode));

T->data = ch;

Create(T->lc);

Create(T->rc);

}

}

* 1. 前序遍历

void PreOrder(BiTree T) {

if (T) {

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

PreOrder(T->lc);

PreOrder(T->rc);

}

}

* 1. 中序遍历

void InOrder(BiTree T) {

if (T) {

InOrder(T->lc);

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

InOrder(T->rc);

}

}

* 1. 后序遍历

void PostOrder(BiTree T) {

if (T) {

PostOrder(T->lc);

PostOrder(T->rc);

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

}

}

* 1. 统计叶子节点

//1

int Leaf1(BiTree T) {

static int n = 0;

if (T) {

if (T->lc == NULL && T->rc == NULL) n= n + 1;

Leaf1(T->lc);

Leaf1(T->rc);

}

return n;

}

//2

int Leaf2(BiTree T) {

if (T == NULL) return 0;

if (T->lc == NULL && T->rc == NULL) return 1;

return Leaf2(T->lc) + Leaf2(T->rc);//此处若再+1则是全部结点

}

* 1. 统计深度

int Depth(BiTree T) {

int dl, dr, dep;

if (T == NULL) dep = 0;

else {

dl = Depth(T->lc);

dr = Depth(T->rc);

dep = 1 + (dl > dr ? dl : dr);

}

return dep;

}

* 1. 对称交换

void Swift(BiTree &T) {

if (T){

BiTNode \*t = T->lc;

T->lc = T->rc;

T->rc = t;

Swift(T->lc);

Swift(T->rc);

}

}

* 1. 层次遍历

char ch;

typedef struct BiTNode {

int data;

struct BiTNode\* lc, \* rc;

}BiTNode, \* BiTree;

typedef struct QNode {

BiTree data;

struct QNode\* next;

}QNode, \* QueuePtr;

typedef struct {

QueuePtr front;

QueuePtr rear;

}LinkQueue;

void Create(BiTree& T) {

cin >> ch;

if (ch == '\*')T = NULL;

else {

T = (BiTNode\*)malloc(sizeof(BiTNode));

T->data = ch;

Create(T->lc);

Create(T->rc);

}

}

int Init(LinkQueue& Q) {

Q.front = Q.rear = (QueuePtr)malloc(sizeof(QNode));

Q.front->next = NULL;

return 1;

}

void En(LinkQueue& Q, BiTree e) {

QueuePtr p = (QueuePtr)malloc(sizeof(QNode));

p->data = e;

p->next = NULL;

Q.rear->next = p;

Q.rear = p;

}

int De(LinkQueue& Q, BiTree &e) {

if (Q.front == Q.rear)return 0;

QueuePtr p = Q.front->next;

e = p->data;

Q.front->next = p->next;

if (Q.rear == p)Q.rear = Q.front;

free(p);

return 1;

}

int JudgeEmpty(LinkQueue Q) {

if (Q.front == Q.rear)

return 0;

return 1;

}

void LevelOrderTraverse(BiTree T){

LinkQueue Q; BiTree t = 0;

if (T){

Init(Q);

En(Q, T);

while (JudgeEmpty(Q)){

De(Q, t);

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

if (t->lc != NULL)

En(Q, t->lc);

if (t->rc != NULL)

En(Q, t->rc);

}

}

}

* 1. 哈夫曼树

#define Order

typedef struct {

int weight;

int parent, lc, rc;

}HTNode,\*HuffmanTree;

typedef char\*\* HuffmanCode;

int min(HuffmanTree t, int i){

int j, m, k = INT\_MAX;

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

if (t[j].weight < k && t[j].parent == 0)

k = t[j].weight, m = j;

t[m].parent = 1;

return m;

}

void Select(HuffmanTree t, int i, int& s1, int& s2){

s1 = min(t, i);

s2 = min(t, i);

#ifdef Order

if (s1 > s2) {

s1 = s1 + s2;

s2 = s1 - s2;

s1 = s1 - s2;

}

#endif

}

void HuffmanTreeCoding(HuffmanTree &HT,HuffmanCode &HC,int \*w,int n){

HuffmanTree p; char \*cd;

int m, i, s1, s2, c, start, f;

if (n <= 1)return;

m = 2 \* n - 1;

HT = (HuffmanTree)malloc((m + 1) \* sizeof(HTNode));

for (p = HT + 1, i = 1; i <= n; ++i, ++p, ++w) {

p->weight = \*w; p->parent = 0; p->lc = 0; p->rc = 0;

}

for (; i <= m; ++i, ++p) {

p->weight = 0; p->parent = 0; p->lc = 0; p->rc = 0;

}

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

Select(HT, i - 1, s1, s2);

HT[s1].parent = i; HT[s2].parent = i;

HT[i].lc = s1; HT[i].rc = s2;

HT[i].weight = HT[s1].weight + HT[s2].weight;

}

HC = (HuffmanCode)malloc((n + 1) \* sizeof(char\*));

cd = (char\*)malloc(n \* sizeof(char));

cd[n - 1] = '\0';

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

start = n - 1;

for (c = i, f = HT[i].parent; f != 0; c = f, f = HT[f].parent)

if (HT[f].lc == c)

cd[--start] = '0';

else

cd[--start] = '1';

HC[i] = (char\*)malloc((n - start) \* sizeof(char));

strcpy(HC[i], &cd[start]);

}

free(cd);

}

int main() {

/\* 8 5 29 7 8 14 23 3 11\*/

HuffmanTree HT;

HuffmanCode HC;

int\* w, n, i;

cin >> n;

w = (int\*)malloc(n \* sizeof(int));

for (i = 0; i <= n - 1; i++)

scanf("%d", w + i);

HuffmanTreeCoding(HT, HC, w, n);

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

puts(HC[i]);

return 0;

}

1. 图
   1. 邻接矩阵

#define MaxN 20

typedef struct {

char vexs[MaxN];

int arcs[MaxN][MaxN];

int vexnum, arcnum;

}MGraph;

int LocateVex(MGraph& G, char v) {

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

if (G.vexs[i] == v)

return i;

return -1;

}

void CreatUDN(MGraph& G) {

char v1, v2;

int w, i, j, k;

cin >> G.vexnum >> G.arcnum;

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

cin >> G.vexs[i];

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

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

G.arcs[i][j] = MaxInt;

for (k = 0; k < G.arcnum; ++k) {

cin >> v1 >> v2 >> w;

i = LocateVex(G, v1);

j = LocateVex(G, v2);

if (i != -1 && j != -1)

G.arcs[i][j] = w,

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

else {

cout << "ERROR";

break;

}

}

}

void Print(MGraph G) {

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

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

if (G.arcs[i][j] != MaxInt)

cout << G.arcs[i][j] << " ";

else

cout << "∞" << " ";

}

cout << endl;

}

}

* 1. 邻接表

#define MaxN 20

typedef struct ArcNode {

int adjvex;

struct ArcNode\* nextarc;

int info;

} ArcNode;

typedef struct VNode {

char data;

ArcNode\* firstarc;

} VNode, AdjList[MaxN];

typedef struct {

AdjList vertices;

int vexnum, arcnum;

} ALGraph;

int LocateVex(ALGraph& G, char v) {

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

if (G.vertices[i].data == v)

return i;

return -1;

}

void CreatGraph(ALGraph& G){

int i, j, k, w; char v1, v2; ArcNode\* p1, \* p2;

cin >> G.vexnum >> G.arcnum;

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

cin >> G.vertices[i].data;

G.vertices[i].firstarc = NULL;

}

for (k = 0; k < G.arcnum; k++) {

cin >> v1 >> v2 >> w;

i = LocateVex(G, v1);

j = LocateVex(G, v2);

p1 = (ArcNode\*)malloc(sizeof(ArcNode));

p1->adjvex = j; p1->info = w;

p1->nextarc = G.vertices[i].firstarc;

G.vertices[i].firstarc = p1;

p2 = (ArcNode\*)malloc(sizeof(ArcNode));

p2->adjvex = i; p2->info = w;

p2->nextarc = G.vertices[j].firstarc;

G.vertices[j].firstarc = p2;

}

}

void Print(ALGraph G) {

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

cout << G.vertices[i].data<<"\t";

while (G.vertices[i].firstarc != NULL) {

cout << G.vertices[i].firstarc->adjvex << " " << G.vertices[i].firstarc->info << "\t";

G.vertices[i].firstarc = G.vertices[i].firstarc->nextarc;

}

cout << endl;

}

}

1. 查找
   1. 查找表的建立

typedef struct {

int key;

int other;

}SElemType;

typedef struct {

SElemType\* elem;

int length;

}SStable;

void Creat(SStable &ST,int n) {

ST.elem = (SElemType\*)malloc((n + 1) \* sizeof(SElemType));

for (int i = 1; i <= n; i++)

cin >> ST.elem[i].key >> ST.elem[i].other;

ST.length = n;

}

* 1. 顺序查找

void SqSearch(SStable ST,int key) {

int cnt = 0; ST.elem[0].key = key;

int i = ST.length;

while (ST.elem[i].key != key) {

cnt++;

i--;

}

if (i) cout << "Find " << cnt + 1;

else cout << "Not Find " << cnt + 1;

cout << endl;

}

* 1. 折半查找（类似于二分查找）

void BinSearch(SStable ST,int key) {

int low = 1, high = ST.length, mid, cnt = 0, flag = 0;

while (low <= high) {

mid = (low + high) / 2;

cnt++;

if (key == ST.elem[mid].key) {

flag = 1;

break;

}

else if (key < ST.elem[mid].key) high = mid - 1;

else low = mid + 1;

}

if (flag)cout << "Find " << cnt;

else cout << "Not Find " << cnt;

cout << endl;

}

//递归用法

void BinSearch(SStable ST, int key, int low, int high) {

int mid;

if (low <= high) {

mid = (low + high) / 2;

if (key == ST.elem[mid].key) cout << mid;

else if (key < ST.elem[mid].key) BinSearch(ST, key, low, mid - 1);

else BinSearch(ST, key, mid + 1, high);

}

else cout << "Not Find";

}

1. 排序
   1. 冒泡排序

int a[10] ={0},t;

for (int h = 0; h < 10; h++)

scanf\_s("%d", &a[h]);

for (int i = 0; i < 9; i++) {

for (int j = 0; j < 9 - i; j++) {

if (a[j] > a[j + 1]) {

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

}

}

}

* 1. 选择排序

int a[10] = { 0 }, t;

for (int h = 0; h < 10; h++) {

scanf\_s("%d", &a[h]);

for (int i = 0; i < 9; i++) {

for (int j = i+1; j < 10; j++) {

if (a[i] > a[j]) {

t = a[i];a[i] = a[j];a[j] = t;

}

}

}

* 1. 直接排序

int a[10] = { 0 }, t,pmin;

for (int h = 0; h < 10; h++)

scanf\_s("%d", &a[h]);

for (int i = 0; i < 9; i++) {

pmin = i;

for (int j = i + 1; j < 10; j++) {

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

pmin = j;

if (pmin != i) {

t = a[i];a[i] = a[pmin];a[pmin] = t;

}

}

}

* 1. 插入排序

//直接插入法

int a[] = { 4, 7, 1, 2, 5, 8, 9, 10, 3, 6 }; int i, j, m;

for (i = 1; i < 10; i++) {

m = a[i];

j = i - 1;

while (j >= 0 && m > a[j]) { //将某一个数字标记，将其插入到它前面适合的位置，之后后面的元素后移

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

a[j] = m;

j--;

}

}

for (i = 0; i < 10; i++)

printf("%3d", a[i]);

printf("\n");

* 1. 希尔排序
  2. 堆排序
  3. 归并排序
  4. 快速排序

void quicksort(int left, int right){

int i, j, t, temp;

if (left > right) return;

temp = a[left];i = left;j = right; //temp为基准数

while (i != j) {

while (a[j] >= temp && i < j) j--; //必须右边先走

while (a[i] <= temp && i < j) i++;

if (i < j) {

t = a[i];a[i] = a[j];a[j] = t;

}

}

a[left] = a[i]; //基准数归位

a[i] = temp;

quicksort(left,i-1); //递归

quicksort(i+1,right);

}

* 1. 桶排序

//要求桶足够多

int a[1000] = { 0 };

int main() {

int t,n;

scanf\_s("%d\n", &n);

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

scanf\_s("%d", &t);

a[t]++;

}

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

for (int j = 1; j <= a[i]; j++) { //判断桶是不是空桶以及桶中还有几个旗子

printf("%d ", i);

}

}

return 0;

}