[图有关的 2](#_Toc106912220)

[拓扑排序 2](#_Toc106912221)

[关键路径 4](#_Toc106912222)

[最短路径 7](#_Toc106912223)

[Dijkstar(单元最短路径) 7](#_Toc106912224)

[Floyd（所有点对的最短路径） 8](#_Toc106912225)

[最小生成树 9](#_Toc106912226)

[Kruscal（最小生成树） 9](#_Toc106912227)

[Prime（最小生成树） 11](#_Toc106912228)

[排序 13](#_Toc106912229)

[插入排序 13](#_Toc106912230)

[（数组）直接插入排序 13](#_Toc106912231)

[（链表）直接插入排序 14](#_Toc106912232)

[（数组）链式直接插入排序 15](#_Toc106912233)

[（数组）折半插入排序 17](#_Toc106912234)

[希尔排序 18](#_Toc106912235)

[交换形式的排序 19](#_Toc106912236)

[快速排序 19](#_Toc106912237)

[冒泡排序 20](#_Toc106912238)

[基数排序 21](#_Toc106912239)

[链式基数排序，搬家 21](#_Toc106912240)

[归并排序 24](#_Toc106912241)

[归并排序实现递归 24](#_Toc106912242)

[归并排序 25](#_Toc106912243)

[选择排序 26](#_Toc106912244)

[普通选择排序 26](#_Toc106912245)

[堆排序 26](#_Toc106912246)

[串匹配 28](#_Toc106912247)

[Kmp 28](#_Toc106912248)

[链表 29](#_Toc106912249)

[（带表头结点）单链表及其一些操作 29](#_Toc106912250)

[（无表头结点）单链表及其一些操作 32](#_Toc106912251)

[（应用）带表头结点的单链表表示多项式 37](#_Toc106912252)

[树有关的 40](#_Toc106912253)

[二叉树大杂烩 40](#_Toc106912254)

[二叉排序树里面的二叉树 45](#_Toc106912255)

[二叉排序树里的二叉树简化版 49](#_Toc106912256)

[栈和队列 52](#_Toc106912257)

[栈 52](#_Toc106912258)

[用数组实现的栈及其一些操作 52](#_Toc106912259)

[用单链表（不带表头结点）实现的栈（链栈） 54](#_Toc106912260)

[队列 56](#_Toc106912261)

[用数组实现队列 56](#_Toc106912262)

[循环队列 57](#_Toc106912263)

[走迷宫 60](#_Toc106912264)

[栈走迷宫 60](#_Toc106912265)

[队列走迷宫 67](#_Toc106912266)

### 图有关的

拓扑排序

#include<bits/stdc++.h>

using namespace std;

struct edge{

int end;

edge \* next\_edge;

};

struct vertex{

int ind;

edge\* first\_edge;

};

int main(){

// freopen("in.txt","r",stdin);

// 建图

vertex G[100];

for(int i = 0;i<100;++i){

G[i].ind = 0;

G[i].first\_edge = NULL;

}

int n,m;

puts("请输入节点数n和边数m");

cin>>n>>m;

int u,v;

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

cin>>u>>v;

edge \* now = new edge;

now->end = v;

now->next\_edge = G[u].first\_edge;

G[u].first\_edge = now;

}

//计算入度

int inde;

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

edge \* p = G[i].first\_edge;

while(p){

G[p->end].ind++;

p = p->next\_edge;

}

}

//拓扑

int index\_tuopu = 0;

int tuopu[100];

int que[100];

int head = 0;

int tail = 0;

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

if(G[i].ind == 0){

que[tail] = i;

tail++;

}

}

while(head != tail){ //head < tail

int now = que[head];

head++;

tuopu[index\_tuopu] = now;

index\_tuopu++;

edge \* p = G[now].first\_edge;

while(p){

G[p->end].ind--;

if(G[p->end].ind == 0){

que[tail] = p->end;

tail++;

}

p = p->next\_edge;

}

}

for(int i=0;i<index\_tuopu;++i){

cout<<tuopu[i]<<" ";

}

cout<<endl;

return 0;

}

关键路径

#include<bits/stdc++.h>

using namespace std;

const int INF = 0x3f3f3f3f;

struct edge{

int end;

int w;

int Ee;

int El;

edge \* next\_edge;

};

struct vertex{

int ind;

int outd;

int Ve;

int Vl;

edge\* first\_edge;

};

//int max(int a,int b){

// return a>b?a:b;

//}

struct path{

int s;

int e;

int w;

};

int main(){

//建图

vertex G[100];

for(int i = 0;i<100;++i){

G[i].ind = 0;

G[i].outd = 0;

G[i].first\_edge = NULL;

}

int n,m;

puts("请输入节点数n和边数m");

cin>>n>>m;

int u,v,w;

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

cin>>u>>v>>w;

edge \* now = new edge;

now->end = v;

now->w = w;

now->next\_edge = G[u].first\_edge;

G[u].first\_edge = now;

G[u].outd++;

}

//计算入度

int inde;

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

edge \* p = G[i].first\_edge;

while(p){

G[p->end].ind++;

p = p->next\_edge;

}

}

//拓扑

int que[100], head = 0, tail = 0; //队列

int sta[100], top = 0; // 出队元素入的栈

//ini

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

G[i].Ve = 0;

G[i].Vl = INF;

}

// 起点入队

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

if(G[i].ind == 0){

que[tail] = i;

tail++;

}

}

// 求每个点的最早发生时间 Ve //实质为拓扑排序的过程

while(head < tail){

int now = que[head++];

sta[top++] = now;

edge \*p = G[now].first\_edge;

while(p){

p->Ee = G[now].Ve;

G[p->end].Ve = max(G[p->end].Ve,G[now].Ve+p->w);

// if(G[now].Ve+p->w > G[p->end].Ve) G[p->end].Ve = G[now].Ve+p->w;

G[p->end].ind--;

if(G[p->end].ind == 0) que[tail++] =p->end;

p = p->next\_edge;

}

}

//找终点

// int end = -1;

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

// if(G[i].outd == 0){

// end = i;

// break;

// }

// }

int end = que[tail-1];

int T = G[end].Ve;

G[end].Vl = T;

path key\_path[200]; int key\_path\_index = 0;

while(top){ // top != 0;

int now = sta[--top];

edge \*p = G[now].first\_edge;

while(p){

p->El = G[p->end].Vl - p->w;

if(p->Ee == p->El) { //判断这条p所指向的边是不是关键路径

path e;

e.s = now;

e.e = p->end;

e.w = p->w;

key\_path[key\_path\_index++] = e;

}

G[now].Vl = min(G[now].Vl,G[p->end].Vl-p->w);

p = p->next\_edge;

}

}

puts("这些路径是关键路径");

for(int i = 0; i<key\_path\_index; i++){

printf("%d -> %d w = %d\n",key\_path[i].e,key\_path[i].s,key\_path[i].w);

}

return 0;

}

##### 最短路径

Dijkstar(单元最短路径)

#include<bits/stdc++.h>

using namespace std;

const int INF = 0x3f3f3f3f;

int get\_D\_min\_pos(int \*D,int n,bool \*vis){

int min\_ = INF;

int pos = -1;

//遍历D数组,找到最小值及其位置pos

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

if(vis[i] == false){

if(min\_ > D[i]){

min\_ = D[i];

pos = i;

}

}

}

return pos;

}

int main(){

// freopen("in.txt","r",stdin);

int G[100][100];

puts("请输入有多少个节点n,有多少条边m,并输入这n条边的信息");

int n;

int m;

int u,v,w;

cin>>n;

cin>>m;

//ini

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

cout<<i<<endl;

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

G[i][j] = INF;

}

}

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

cin>>u>>v>>w;

G[u][v] = w;

G[v][u] = w;

}

// 起点为1

int D[100];

int p[100];

bool vis[100]; //判断节点i是否最短路径已经确定

//ini

for(int i=1; i<=n; i++) D[i] = INF;

for(int i=1; i<=n; i++) p[i] = -1;

for(int i=1; i<=n; i++) vis[i] = false;

D[1] = 0;

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

//找出D数组中的最小值

int t = get\_D\_min\_pos(D,n,vis);

vis[t] = true; //节点t的最短路径确定,vis数组标记为true

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

if(G[t][i] != INF){

if(D[t]+G[t][i] < D[i]){

D[i] = D[t] + G[t][i];

p[i] = t;

}

}

}

}

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

printf("从起点1到节点%d的最短距离是%d\n",i,D[i]);

}

return 0;

}

Floyd（所有点对的最短路径）

#include<iostream>

#define maxn 100

using namespace std;

const int INF = 0x3f3f3f3f;

int main(){

int G[maxn][maxn];

int n,m;

puts("请输入点数n和边数m，以及这m条边（无向边，点的编号从1开始）");

cin>>n>>m;

//初始化G

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

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

if(i == j) G[i][j] = 0;

else G[i][j] = INF;

}

}

int u,v,w;

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

cin>>u>>v>>w;

G[u][v] = w;

G[v][u] = w;

}

for(int k=1; k<=n; k++){

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

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

G[i][j] = min(G[i][j],G[i][k]+G[k][j]);

}

}

}

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

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

if(G[i][j] == INF) printf("%-5s ","INF");

else printf("%-5d ",G[i][j]);

}

puts("");

}

return 0;

}

##### 最小生成树

Kruscal（最小生成树）

#include<bits/stdc++.h>

#include<algorithm>

using namespace std;

struct edge{

int u;

int v;

int w;

};

bool cmp(const edge& a,const edge& b){

return a.w < b.w;

}

int get\_ancester(int\* F,int node){

int t = node;

while(F[t] != -1){

t = F[t];

}

return t;

}

int main(){

edge e[1000];

puts("请输入边的数量n，以及这n条边的信息");

int n;

cin>>n;

int u,v,w;

//读图

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

cin>>u>>v>>w;

e[i].u = u;

e[i].v = v;

e[i].w = w;

}

//按边的权重从小到大排序 注意考试的时候可能不可以用STL里的sort，所以可以自己写排序，或者每一次for一遍所有边找出最短的边

sort(e+1,e+1+n,cmp);

//

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

cout<<"i = "<<i<<"u : "<<e[i].u<<" v : "<< e[i].v << "w : "<< e[i].w<<endl;

}

// define F ,ini

int F[2000];

for(int i=0; i<2000; i++) F[i] = -1;

int sum = 0;

// mist 储存

edge mist[1000];

int index = 0;

//

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

edge now = e[i];

int u = now.u;

int v = now.v;

int w = now.w;

int ancester\_u = get\_ancester(F,u);

int ancester\_v = get\_ancester(F,v);

if(ancester\_u == ancester\_v) continue;

else {

F[ancester\_u] = ancester\_v;

sum += w;

mist[index] = now;

index++;

// mist[index++] = now;

}

}

//out

cout<<"MIST 的权值和为"<<sum<<endl;

cout<<"将这几条边相连可以得到最小生成树"<<endl;

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

cout<<"u = "<<mist[i].u<<"v = "<<mist[i].v <<"w = "<<mist[i].w<<endl;

}

return 0;

}

Prime（最小生成树）

#include<iostream>

#define maxn 100

using namespace std;

const int INF = 0x3f3f3f3f;

int get\_D\_min(int D[],bool vis[],int n){

int pos,min\_ = INF;

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

if(!vis[i] && min\_ > D[i]){

min\_ = D[i];

pos = i;

}

}

return pos;

}

int main(){

int G[maxn][maxn];

int D[maxn];

int F[maxn];

bool vis[maxn];

int n,m;

puts("请输入点数n和边数m，以及这m条边（无向边，点的编号从1开始）");

cin>>n>>m;

//初始化G

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

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

if(i == j) G[i][j] = 0;

else G[i][j] = INF;

}

}

int u,v,w;

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

cin>>u>>v>>w;

G[u][v] = w;

G[v][u] = w;

}

//初始化D，vis

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

D[i] = INF;

F[i] = -1;

vis[i] = false;

}

int MIST = 0;

D[1] = 0;

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

int p = get\_D\_min(D,vis,n);

vis[p] = true;

MIST += D[p];

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

if(!vis[j] && G[p][j] < D[j]){ //只有对于没有纳入MIST的点才进行更改

D[j] = G[p][j];

F[j] = p;

}

}

}

cout<<"最小生成树的权值为"<<endl;

cout<<MIST<<endl;

return 0;

}

### 排序

##### 插入排序

（数组）直接插入排序

#include<iostream>

#define maxn 10000

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.直接插入排序就是每次将一个数插入到有个排好序的区间中，实现起来也非常的容易

2.这里是将数组从小到大排序

3.这里我们采用的是含有哨兵的，注意我们使用哨兵，只是为了我们的代码更加的简洁，在算法复杂度上面并没有什么优化，此处的哨兵便是将要插入的元素放在R[0]位置，那么到了0位置

一定会满足条件

4.实际上这里不用哨兵代码也是一样的简洁

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(){

int R[maxn]; //maxn 宏定义

int n;

puts("输入数组的大小n，以及这n个数");

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

//将后面的数往前插入

for(int i=2; i<=n; i++){ //将R[i]上面的数插到前面 [1,i-1]这个有序区间里面去

if(R[i] < R[i-1]){ //只有当R[i] < R[i-1]的时候我们采用把这个往前面插。不然R[i]本身就在现在正确的位置上面

R[0] = R[i];

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

if(R[0] < R[j]) R[j+1] = R[j];

else { //此时R[j] <= R[0]，所以把R[0]放在R[j+1]这个上面

R[j+1] = R[0];

break;

}

}

}

}

//

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

（链表）直接插入排序

#include<iostream>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.实际上的链表的插入排序也就是一个单链表操作

2.思想同数组的插入排序是一样的，我们将每一个节点一次往前面插入，最后形成的链表就是排好序的链表

3.因为有这种插入操作，我们最好使用带表头节点的单链表，代码实现起来会更加的容易一点

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct Node{ //链表的节点

int v;

Node\* next;

};

int main(){

Node\* L = NULL;

//将表头节点申请出来

Node\* q = new Node;

q->v = -1;

q->next = NULL;

L = q;

//输入数据

puts("请输入元素的个数，以及这n个元素的值");

int n,val;

cin>>n;

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

cin>>val;

//申请节点

Node\* t = new Node;

t->v = val;

t->next = NULL;

//头插入

t->next = L->next;

L->next = t;

}

//将链表分成两段，一段为L指向的（这一段代表已经排好序的）， 一段为p指向的（这一段代表待排序的）

Node\* p = L->next;

L->next = NULL;

//开始排序，将p中的每一个节点插入到L的这一条链中

Node\* next;

Node\* t; //用于遍历L指向的链表

while(p){

next = p->next; //先记录下p节点后面是哪一个节点，不然之后无法往后移动，因为此时p所指向的节点会被插入到 L 中的某一个位置

t = L; // 使用t来遍历L所指向的链表

while(t->next && (t->next)->v < p->v) t = t->next;

//p指向的节点插入到t所指向的节点的后面

p->next = t->next;

t->next = p;

//p移动到下一个节点

p = next;

}

//看看排序的效果

Node\* r = L->next;

while(r){

cout<<r->v<<" ";

r = r->next;

}

cout<<endl;

return 0;

}

（数组）链式直接插入排序

#include<iostream>

#define maxn 1000

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.这个是链式的直接插入排序，实际上是对数组的直接插入排序的优化

2.对数组直接插入排序的优化是不用每一次插入都进行多次赋值，而是当形成L数组后每一个点最多只进行一次赋值，而L数组就相当与一条另外一种意义上的链表

3.这个优化主要是在于当待排序的节点的大小很大时，这个赋值操作的速度便会慢下来（因为当节点很大的时候，一次节点的赋值就对相当多的变量进行了赋值），所以

我们利用L数组实现类似链表的链式排序，实际上是对下标进行了排序

4.这个根据L数组复原的过程就是老师上课讲的搬家

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(){

int R[maxn];

int L[maxn];

int n;

// 先输入数据

puts("请输入元素的个数n，以及这n个元素");

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

//将链建立起来

for(int i=0; i<n; i++) L[i] = i+1; //形成这样逻辑上的链 0->1->2->3->4->.....->n->-1

L[n] = -1; //最后一个点的下一个L[n] = -1

// 对着一条链排序，实际上何链表的直接插入排序是一样的

int p = L[0];

L[0] = -1;

int t;

int next;

while(p != -1){

next = L[p];

t = 0;

while(L[t] != -1 && R[L[t]] < R[p]) t = L[t];

L[p] = L[t];

L[t] = p;

p = next;

}

//现在L数组里面就储存了排好序的元素的先后关系，如果新开一个数组，我们可以很简单的解决这一个问题，但是如果不允许新开数组的话，我们只能用老师讲的搬家

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//这是新开数组的方法，我觉得应该很好理解了吧，但是我们需要浪费一倍的空间，所以在某些情况下并不是很可取，所以才有了后面所谓的“搬家”

int temp[maxn];

int index = 1;

p = L[0];

while(p != -1){

temp[index++] = R[p];

p = L[p];

}

for(int i=1; i<=n; i++) R[i] = temp[i];

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这个就是老师讲了很久的搬家，想法确实很精妙的，应该好好理解理解！！！

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int i = 1;

p = L[0];

while(p != -1){

if(p == i){

i++;

p = L[p];

}

else if(p > i){

// 交换 swap 写成函数好看一点

int m = R[i];

R[i] = R[p];

R[p] = m;

//交换所带来的的赋值

int next = L[p]; //因为L[p]后面被赋值为l[i]所以要提前记录一下，不然找不到p后面是谁了

L[p] = L[i]; //带着i后面的这一天链

L[i] = p; //i被搬动到了p号位置

//往后面走

i++;

p = next;

}

else {

while(p < i) p = L[p]; //找到p正确的位置

}

}

//输出一下结果

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

（数组）折半插入排序

#include<iostream>

#define maxn 100010

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.折半插入排序，就是将插入排序中在有序区间中找要插入位置的过程用二分查找来完成，使这一步的复杂度将为logn

2.但实际上因为插入排序需要搬动数据，导致这个排序的最坏复杂度还是达到了n2的级别

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(){

int R[maxn];

int n;

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

for(int i=2; i<=n; i++){

int low = 1,high = i-1,mid;

while(low <= high){

mid = (low+high)/2;

if(R[i] < R[mid]) high = mid-1;

else low = mid+1;

}

//要加R[i]放在high+1的位置，所以将[high+1,i-1]上的数据都向后移动一位

int t = R[i];

for(int j = i-1; j>=high+1; j--) R[j+1] = R[j];

R[high+1] = t;

}

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

希尔排序

#include<iostream>

#define maxn 100010

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.希尔排序也是插入排序的一种，当其间隔选取为1的时候就是普通插入排序

2.它能加速的地方在于，在进行间隔为1的插入排序之前，我先进行间隔为其他更大的值的排序，这样我可能可以保证后面的排序的移动次数会很少，其实比较玄学，但是书上讲这个复杂度的证明

是一个很难的问题，我们就不考虑了，就直接实现

3.对于增量序列要注意的是，最后一个增量一定要为1，且增量序列中的值没有除1以外的公因子

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(){

int d[10] = {9,8,7,3,1} ;//这个间隔的数组由自己确定，但是一定要保证最后一个间隔一定要为1，且中间最好不要存在倍数（我觉得有倍数存在的话，间隔大的那一次没什么用）

int R[maxn];

// puts("输入元素的个数n以及这n个元素");

int n;

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

for(int m=0; m<5; m++){

int delta = d[m]; //取出间隔

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

int t = R[i];

int j;

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

if(t < R[j]) R[j+delta] = R[j];

else break;

}

R[j+delta] = t;

}

}

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

##### 交换形式的排序

快速排序

#include<iostream>

#define maxn 100010

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.快速排序也属于交换的排序

2.虽然叫做快排但是不优化的就很慢

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void fast\_sort(int R[],int s,int e){

int l = s,r = e;

if(l >= r) return ;

int t = R[l];

while(l < r){

while(l < r){ //从右边扔向左边

if(R[r]<t){

R[l] = R[r];

break;

}

r--;

}

while(l < r){

if(R[l] >= t){ //从左边扔向右边

R[r] = R[l];

break;

}

l++;

}

}

int gap = l;

R[gap] = t;

fast\_sort(R,s,gap-1);

fast\_sort(R,gap+1,e);

return ;

}

int main(){

int n;

int R[maxn];

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

fast\_sort(R,1,n);

//

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

冒泡排序

#include<iostream>

#define maxn 100010

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.冒泡排序就是每一轮通过相邻元素的交换，将当前最大的元素（或者最小的元素）像冒泡一样浮动到数组的最后面

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void swap(int\* p,int\* q){

int t = \*p;

\*p = \*q;

\*q = t;

return ;

}

int main(){

int R[maxn];

int n;

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

for(int i=n; i>=2; i--){

bool flag = false;

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

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

swap(&R[j],&R[j+1]);

flag = true;

}

}

if(!flag) break;

}

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

##### 基数排序

链式基数排序，搬家

#include<bits/stdc++.h>

using namespace std;

int get\_figure(int x){

int base = 1;

for(int i=1; ;i++){

base = base\*10;

if(x/base == 0) return i;

}

}

int get\_digit(int x,int p){//x 的 第p位数

int base = 1;

for(int i=1; i<=p-1; i++){

base = base\*10;

}

return x/base%10;

}

struct Bucket{

int f;

int e;

};

int main(){

int n;

int R[1000];

//读入数据

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

// 建链

int L[1000];

for(int i=0; i<n; i++) L[i] = i+1;

L[n] = -1;// L[n] = 0;

// 最多有多少位

int max\_figure = 0;

for(int i=1; i<=n; i++) max\_figure = max(max\_figure,get\_figure(R[i]));

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

// if(max\_figure < get\_figure(R[i])) max\_figure = get\_figure(R[i]);

// }

//建桶

Bucket T[10];

//

for(int i=1; i<=max\_figure; i++){

//将桶清空

for(int i=0; i<=9; i++) T[i].f = T[i].e = -1;

//

int p = L[0];

while(p != -1){

int t = get\_digit(R[p],i);

//将R[p]这个数放入t号桶

if(T[t].f == -1){

T[t].f = p;

T[t].e = p;

}

else{

L[T[t].e] = p;

T[t].e = p;

}

p = L[p];

}

//每个桶的元素链起来起来

int pre = 0;//当前链的最后一个是第几号元素

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

if(T[i].f != -1){

L[pre] = T[i].f;

pre = T[i].e;

}

}

L[pre] = -1;

}

// int p = L[0];

// while(p != -1){

// cout<<R[p]<<" ";

// p = L[p];

// }

// cout<<endl;

// cout<<"R[] = ? "<<endl;

// for(int i=1; i<=n; i++) cout<<R[i]<<" ";cout<<endl;

//

// 新开的数组

// int temp[1000];

// int index = 1;

// int p = L[0];

// while(p != -1){

// temp[index++] = R[p];

// p = L[p];

// }

// for(int i=1; i<=n; i++) cout<<temp[i]<<" "; cout<<endl;

// for(int i=1; i<=n; i++) R[i] = temp[i];

// for(int i=1; i<=n; i++) cout<<R[i]<<" "; cout<<endl;

// 只在R中进行交换操作 - “搬家”

int i = 1;

int p = L[0];

while(p != -1){

if(i == p){

p = L[p];

i++;

}

else if(i < p){

int p\_next = L[p];

swap(R[i],R[p]);

L[p] = L[i];

L[i] = p;

p = p\_next;

i++;

}

else if(i > p){

while(p < i) p = L[p];

}

}

for(int i=1; i<=n; i++) cout<<R[i]<<" ";cout<<endl;

return 0;

}

##### 归并排序

归并排序实现递归

#include<bits/stdc++.h>

using namespace std;;

const int maxn = 1000;

void merge(int R[],int tem[],int s,int m,int e){

int pa = s;

int pb = m+1;

int index = s;

// 正常归并排序步骤

while(pa <= m && pb <= e){

if(R[pa] <= R[pb]) tem[index++] = R[pa++];

else tem[index++] = R[pb++];

}

while(pa <= m) tem[index++] = R[pa++];

while(pb <= e) tem[index++] = R[pb++];

// copy

for(int i=s; i<=e; i++) R[i] = tem[i];

return ;

}

void gb\_sort(int R[],int tem[],int s,int e){ // 对区间[s,e]排好序

if(s == e) return ;

int m = (s+e)/2;// 还可以写成 m = (s+e)>>1;

gb\_sort(R,tem,s,m);

gb\_sort(R,tem,m+1,e);

merge(R,tem,s,m,e);//将区间[s,m],[m+1,e]合并

return ;

}

int main(){

int n;

int R[maxn];

int tem[maxn];

//输入数据

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

gb\_sort(R,tem,1,n);

//康康结果

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

归并排序

#include<bits/stdc++.h>

using namespace std;

const int maxn = 1e5+10;

// 归并排序 非递归实现

// 从小到大排序

void merge(int R[],int tem[],int s,int m,int e){ // 将R数组[s,m]区间和[m+1,e]区间合并,(此时[s,m],[m+1,e]已经各自有序)

int pa = s;

int pb = m+1;

int index = s;

// 正常归并排序步骤

while(pa <= m && pb <= e){

if(R[pa] <= R[pb]) tem[index++] = R[pa++];

else tem[index++] = R[pb++];

}

while(pa <= m) tem[index++] = R[pa++];

while(pb <= e) tem[index++] = R[pb++];

// copy

for(int i=s; i<=e; i++) R[i] = tem[i];

return ;

}

int main(){

int n; // 数组的大小

int R[maxn];

int tem[maxn]; //归并排序必备的临时数组,就是老师讲的 R1数组

// 输入数据

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

// for循环合并区间

for(int len = 1; len<n; len <<= 1){ //len可以理解成对于当前数组R 每len 的区间已经有序了,所以我是想合并[1,1+len-1]与[1+len,len\*2]

for(int i = 1; i+len<=n; i+=len\*2){

merge(R,tem,i,i+len-1,min(i+len\*2-1,n)); // 仔细理解这一句话,想明白归并排序非递归实现就想明白大半了

}

}

// 让我们康康归并排序的结果怎么样

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

##### 选择排序

普通选择排序

#include<iostream>

#define maxn 100010

using namespace std;

const int INF = 0x3f3f3f3f;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.这是普通选择排序，想法十分简单，就是每一次选择当前最大的数放在最后

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void swap(int \*p,int \*q){

int t = \*p;

\*p = \*q;

\*q = t;

return ;

}

int main(){

int n;

int R[maxn];

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

for(int i=n; i>=2; i--) {

int pos,m = -INF; // 0x3f3f3f3f是一个很大的数

for(int j=1; j<=i; j++){ //循环找到[1,i]区间中最大的数的位置(pos)

if(m < R[j]){

m = R[j];

pos = j;

}

}

swap(&R[i],&R[pos]);

}

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

堆排序

#include<iostream>

#define maxn 100010

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.堆排序我们也把他认为是选择排序的一种，因为它排序也是每次将当前最大的数放在最后，同普通选择排序一样。

2.但堆排序的效率很优秀，在与他的选择过程利用了堆，每次选择之后维护堆的过程只有logn，所以其最坏复杂度为nlogn

3.堆排序要注意的是为了这个二叉堆的结构，我们储存的数据必须从1开始

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void swap(int \*p,int \*q){

int t = \*p;

\*p = \*q;

\*q = t;

return ;

}

void mt(int R[],int rt\_id,int max\_son){

for(int son = rt\_id\*2; son <= max\_son; son \*= 2){

if((son+1)<=max\_son && R[son] < R[son+1]) son = son+1;

if(R[son] > R[rt\_id]) {

swap(&R[son],&R[rt\_id]);

rt\_id = son;

}

else break;

}

return ;

}

int main(){

int n;

int R[maxn];

cin>>n;

for(int i=1; i<=n; i++) cin>>R[i];

for(int i=n/2; i>=1; i--) mt(R,i,n);

for(int i=n; i>=2; i--){ // 将当前最大的数放在i号位置

swap(&R[1],&R[i]);

mt(R,1,i-1);

}

//输出

for(int i=1; i<=n; i++) cout<<R[i]<<" ";

cout<<endl;

return 0;

}

### 串匹配

Kmp

#include<iostream>

#include<cstring>

#define maxn 1000

using namespace std;

void get\_moveto(char\* s,int moveto[]){

int lenth = strlen(s+1);

moveto[1] = 0;

for(int i=2; i<=lenth; i++){

int j = moveto[i-1];

while(j && s[j] != s[i-1]) j = moveto[j];

moveto[i] = j+1;

}

return ;

}

int main(){

puts("请输入模式串P和匹配串S，将返回S串在P串中的位置");

char p[maxn];

char s[maxn];

int moveto[maxn];

scanf("%s",p+1);

scanf("%s",s+1);

get\_moveto(s,moveto);

//匹配

int i=1,j=1;

int len\_p = strlen(p+1);

int len\_s = strlen(s+1);

bool flag = false; //标志没有找到

while(i<=len\_p){

if(j == 0 || p[i] == s[j]){

i++;

j++;

}

else j = moveto[j];

if(j == len\_s+1){

cout<<"在P串中找到了S串，从P串的"<<i-len\_s<<"位置开始出现了S串"<<endl;

flag = true;

break;

}

}

if(!flag) cout<<"没有找到S串"<<endl;

return 0;

}

### 链表

（带表头结点）单链表及其一些操作

#include<cstdio>

#include<iostream>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.这是带表头节点的单链表，最好可以把这个和不带表头节点的单链表进行对比，这样便能很好的体会带表头节点的单链表对代码的简化

2.大部分代码和没有表头节点差不多，但注意一些细节，比如说传入的参数，我们还用有的函数传入引用或者说指针类型吗？

3.你会发现为什么代表头节点的单链表会少很多的判断！

4.这部分的代码的注释我就写的没有不带表头节点的单链表的那么详细了！

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

实现增删查改四个函数，以及链表的翻转函数reverse()

更多的其他的链表操作见 带表头节点的单链表 表示多项式！

\*/

typedef struct Node{

int data;

Node\* next;

} Node,\*List;

bool Linsert(List L,int pos,int val){ //注意这里就直接传List L就行了，就没有传List& L了

Node\* t = new Node;

t->data = val;

t->next = NULL;

//

Node\* p = L;

for(int i=0; p&&i<pos-1; i++) p = p->next;

if(p){

t->next = p->next;

p->next = t;

}

else return false; //说明插入位置不合法，返回false

return true;

}

bool Lremove(List L,int pos,int\* rem){

Node\* p = L;

for(int i=0; p&&i<pos-1; i++) p = p->next;

if(p && p->next){

//删除节点p->next

Node\* d = p->next;

p->next = d->next;

\*rem = d->data;

delete d;

}

else return false; //p 和 p->next 其中任一个为空都说明无法删除，即删除位置不合法

return true;

}

Node\* Lat(List L,int pos){ //返回这个位置上的节点的指针，如果不存在则返回空

Node\* p = L;

for(int i=0; p&&i<pos; i++) p = p->next;

return p;

}

bool Lmodify(List L,int pos,int new\_val){

Node\* p = L;

for(int i=0; p&&i<pos; i++) p = p->next;

if(p) p->data = new\_val;

else return false; // p为空，说明修改位置不合法，返回false

return true;

}

void reverse(List L){//将链表反转

Node\* pre = NULL;

Node\* now = L->next;

Node\* next;

while(now){

next = now->next;

now->next = pre;

pre = now;

now = next;

}

L->next = pre;

return ;

}

void disp(List L){

Node\* p = L->next;

while(p){

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

p = p->next;

}

return ;

}

int main(){

//建立起链表，注意此时带表头节点

List L;

L = new Node; //指向表头节点

L->data = -1; //表头节点无意义，我们将表头节点的值赋为-1，这个值由你自己来定

L->next = NULL; //next置空

//我们可以插入一些数据

int n,val;

puts("输入元素的个数n，以及这n个元素");

cin>>n;

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

cin>>val;

Linsert(L,1,val);// 头插入

}

cout<<"输入的链表L为 : ";

disp(L);

cout<<endl;

//反转链表reverse

cout<<"将链表反转"<<endl;

reverse(L);

disp(L);

cout<<endl;

//举一些例子

//在2号位置插入一个999

cout<<"在2号位置插入一个999"<<endl;

Linsert(L,2,999);

disp(L);

cout<<endl;

//将3号节点删掉

cout<<"将3号节点删掉" <<endl;

int r;

Lremove(L,3,&r);

disp(L);

cout<<endl;

//查询4号位置是什么

cout<<"查询4号位置是什么"<<endl;

Node\* ret = Lat(L,4);

if(ret) cout<<"是 "<<ret->data<<endl;

else cout<<"没有4号位置"<<endl;

//将2号位置的值改为88888

cout<<"将2号位置的值改为88888"<<endl;

Lmodify(L,2,88888);

disp(L);

cout<<endl;

return 0;

}

（无表头结点）单链表及其一些操作

#include<cstdio>

#include<iostream>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

注意事项：

1.注意这个是不带表头节点的单链表，注意和有表头节点的单链表对比!!!

两者对比，你应该可以理解这个表头节点（无实际意义的节点）的好处了

2.为了节省时间，像什么创建单链表的函数我就不单独写了（事实上这些函数也就一句话 List L = NULL）

3.这个链表我默认的下边是从1开始，即 1号元素就是第一个元素，若你习惯从0开始（即0是第一个元素的话），做相应的更改即可

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct Node{ //链表的节点的定义

int data;

Node\* next;

}Node,\*List;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.这部分是往单链表中插入元素的操作

实际上是一个insert函数便可以解决的

但是可能到时候会考什么叫你写头插入，尾插入，所以我就把这两个函数单独拿出来写一遍

但要明白，所谓头插入，尾插入，本质就是insert函数中 pos 的值

3.注意为什么我这里的L传的引用类型（ 或者说用c语言的话要传List\*类型（注意List\*类型实际是Node\*\*类型） ） ，

这里从一个方面体现了不带表头节点的单链表比带表头节点所欠缺的东西（不带表头节点的单链表是什么辣鸡qwq）

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void head\_insert(List& L, int val){

//常规操作，将节点先申请出来

Node\* t = new Node;

t->data = val;

t->next = NULL;

//将该节点放在头部

t->next = L;

L = t; // 传引用的原因（或者c语言：传List\*的原因） 注意！！！

return ;

}

void tail\_insert(List& L,int val){

//常规操作，将节点先申请出来

Node\* t = new Node;

t->data = val;

t->next = NULL;

//将节点放在尾部

// 先找到尾巴在哪里

Node\* p = L; //使用指针p来寻找

while(p && p->next){

p = p->next;

}

//循环结束的时候我们就找到了尾巴在那里，尾巴就是p，所以我们要将节点放在p指向节点的后面

// 但是注意我们循环中还写了一个 p && p->next中判断p是否为空的条件，这个是为了防止本身L就是一个空链表，即L == NULL，那么此时的尾插入实际上就是赋值

if(p) p->next = t; //p不为空，说明p是尾节点，那么我们将节点插入到p的后面

else L = t; //p为空，说明L本身就是空的，所以尾插入到L的链表中，实际上就是 L = t;

return ;

}

bool Linsert(List& L,int pos,int val){ //将值val插入到链表L的pos位置上即插入的节点变为链表L的第pos个节点，若pos位置不合法则返回false代表插入失败，同时不做任何操作

//常规操作，先将节点申请出来

Node\* t = new Node;

t->data = val;

t->next = NULL;

//特判是不是要插入到头部，即pos = 1

if(pos == 1) {

t->next = L;

L = t;

}

else {

//找到要插入的位置，实际上是pos位置前面的那个节点

Node\*p = L; //用p来找到位置

for(int i=1; i<pos-1 && p; i++){

p = p->next;

}

if(p){ //节点插入在p和p->next之间

t->next = p->next;

p->next = t;

}

else { //p为空，说明要插入的位置不合法

return false;

}

}

return true;

}

//Lsize()

int Lsize(List L){ //返回链表L的元素的大小

int cnt = 0;

Node\* p = L;

while(p){

cnt++;

p = p->next;

}

return cnt;

}

//reverse()

void reverse(List& L){

Node\* pre = NULL;

Node\* now = L;

Node\* next;

while(now){

next = now->next;

now->next = pre;

pre = now;

now = next;

}

L = pre;

return ;

}

void disp(List L){ //打印链表

Node\* p = L;

while(p){

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

p = p->next;

}

return ;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这几个就是增（insert函数前面已经实现了）删查改函数，大致思路和insert函数都差不多，可以自己实现来练手，

这几个函数返回值，传参都可以自己设计，不要拘泥与这个设计

bool remove(List L,int pos,int\* del) 删除链表L第pos位置上的节点，并将删除的节点的值通过指针del带出来，返回是否删除成功

Node\* at(List L,int pos) 返回 指向链表L第pos位置上节点的指针，没有pos位置上的节点就返回空

bool modify(List L,int pos,int new\_val) 将链表第pos位置上的节点的值改为new\_val，修改成功就返回true，否则返回false

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Node\* Lat(List L,int pos){

Node\* p = L;

for(int i=1; p&&i<pos; i++) p = p->next;

return p;

}

bool Lremove(List& L,int pos,int\* del){ //注意为什么传引用

//特判是不是删除了头节点

if(pos == 1){

if(L == NULL) return false;//本身就是空节点，删除失败

else {

Node\* t = L;

L = L->next;

\*del = t->data;//把删除的节点的值带出来

delete t;//删掉节点

}

}

else {

Node\* p = L;//找到要删除的节点的前面一个节点（即其前驱节点）

for(int i=1; p&&i<pos-1; i++) p = p->next;

if(p && p->next){

Node\* t = p->next;

p->next = t->next;

\*del = t->data;

delete t;

}

else return false;//p或者p->next中有一个为空，都表明无法删除，即删除失败

}

return true;

}

bool Lmodify(List L,int pos,int new\_val){

Node\* p = L;

for(int i=1; p&&i<pos; i++) p = p->next;

if(p) p->data = new\_val;

else return false;//p为空，说明当前链表中根本就没有pos这个位置上的节点，所以修改失败，返回false

return true;

}

int main(){

//创建一个单链表，空链表

List L = NULL; // L的类型使List，实际上就是Node\*类型

//现在我们往这个链表里面加入一些数据

int n,val;

puts("请输入元素的个数n，以及这n个元素");

cin>>n;

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

cin>>val;

head\_insert(L,val); //头插入

//尾插入 则调用 tail\_insert(L,val)

//或者是insert(L,1,val)

}

// 我们可以输出一下链表中元素的个数 用 Lsize() 函数

// 可以想想我们一直按头插入插入进去，这个时候输出链表应该是什么样子！是不是会跟你想的不一样

cout<<"链表L中元素的个数为"<<Lsize(L)<<endl;

cout<<"此时链表 : ";

disp(L);

cout<<endl;

// 将单链表L反转 reverse()函数，可能会考，也可能不会，看看吧

reverse(L);

cout<<"链表反转之后 : ";

disp(L);

cout<<endl;

// 举几个使用的例子

// Lat

Node\* p = Lat(L,2);

if(p) cout<<"at 2 "<<p->data<<endl;

//Lremove

int dele;

bool f = Lremove(L,2,&dele);

if(f) cout<<dele<<" is removed"<<endl;

else cout<<"fail to remove"<<endl;

disp(L); cout<<endl;

//Lmodify

int new\_val = 100;

f = Lmodify(L,2,new\_val);

if(f){

Node\* t = Lat(L,2);

if(t) cout<<"节点2 "<<"has been modified to "<<t->data<<endl;

}

else cout<<"fail to modify"<<endl;

disp(L); cout<<endl;

return 0;

}

（应用）带表头结点的单链表表示多项式

#include<cstdio>

#include<cstdlib>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.这是使用带表头节点的单链表表示多项式

2.实际上就是单链表的一些操作

3.coffecient表示这一项的系数，exponent是这一项的指数

4.Polynomial实际上就是前面写链表的时候的那个 L

5.malloc\_Node() 函数不用管，实际上这是我写的时候为了偷懒，所以把申请节点的操作最为一个函数独立出来了，其实就是正常的申请节点

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct Node{

int coffecient;

int exponent;

Node\* next;

} Node,\*Polynomial;

void InitPolynomial(Polynomial& a){ //初始化，实际上就是建立表头节点

Node\* p = (Node\*)malloc(sizeof(Node));//申请出来表头节点

p->coffecient = -1;

p -> exponent = -1;

p -> next = NULL;

a = p;

return ;

}

Node\* malloc\_Node(const int coffecient,const int exponent){ //这个函数不用管

Node\* ret = new Node;

ret -> coffecient = coffecient;

ret -> exponent = exponent;

ret -> next = NULL;

return ret;

}

void orderinsert(const int coffecient,const int exponent,Polynomial a){ //将这一项(coffecient,exponent)这一项插入到正确的位置，按照降序插入

Node\* p = a;

while(p -> next && exponent<(p->next)->exponent)p = p->next; //找到要插入的位置，插入到p节点的后面

if(p -> next && (p -> next)->exponent == exponent){ //判断如果是指数相等推出的，那么要先合并，再做相应的操作

p->next->coffecient += coffecient;

if(!(p->next->coffecient)){ //合并完了没有了，删除该节点

Node\* del = p->next;

p -> next = del -> next;

delete del;

}

}

else{ //此时p为改该多项式末尾，或者p->next->exponent < exponent，不管哪一种情况，但是插入到p节点后面

Node\* q = malloc\_Node(coffecient,exponent);

q -> next = p -> next;

p -> next = q;

}

return ;

}

void read(Polynomial a){ //读入多项式

int n;

scanf("%d",&n);

int coffecient,exponent;

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

scanf("%d%d",&coffecient,&exponent);

if(coffecient) orderinsert(coffecient,exponent,a);

}

return ;

}

void destroy(Polynomial a){ //销毁多项式

Node\* p = a;

while(p){

Node\* del = p;

p = p->next;

free(del);

}

return ;

}

void disp(Polynomial a){ //打印多项式

Node\*p = a->next;

while(p){

printf("(%d,%d)",p->coffecient,p->exponent);

p = p->next;

}

puts("");

return ;

}

void Polynomial\_Plus(Polynomial a,Polynomial b){ //将两个多项式相加

Node\* pa = a;

Node\* pb = b->next;

//相加的逻辑就是将b中的每一项插入到a的对应位置就行了

while(pb){ //while(pb)遍历b的每一项，将其加入a中

while(pa->next && pb->exponent < pa->next->exponent) pa = pa->next;

if(pa->next && pa->next->exponent == pb->exponent){

pa->next->coffecient += pb->coffecient;

if(!(pa->next->coffecient)){

Node\* del = pa->next;

pa->next = del->next;

delete del;

}

}

else{

Node\* q = malloc\_Node(pb->coffecient,pb->exponent);

q->next = pa->next;

pa->next = q;

}

pb = pb->next;

}

return ;

}

void reverse(Polynomial a){ // 反转多项式，实际上就是反转链表

Node\* pre = NULL;

Node\* now = a->next;

Node\* next;

while(now){

next = now->next;

now->next = pre;

pre = now;

now = next;

}

L->next = pre;

return ;

}

int main(){

puts("请输入多项式的项数n,并在后面输入n个二元组,表示每一项的系数以及其指数");

Polynomial a;

InitPolynomial(a);

read(a);

puts("该多项式为");

disp(a);

destroy(a);

puts("请输入多项式a的项数n,并在后面输入n个二元组,表示每一项的系数以及其指数");

InitPolynomial(a);

read(a);

puts("请输入多项式b的项数n,并在后面输入n个二元组,表示每一项的系数以及其指数");

Polynomial b;

InitPolynomial(b);

read(b);

puts("这是多项式a和b的和");

Polynomial\_Plus(a,b);

disp(a);

destroy(a);

return 0;

}

### 树有关的

二叉树大杂烩

#include<iostream>

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.二叉树的建立

2.二叉树的三种遍历的递归实现以及非递归实现 以及层序遍历

3.一些简单的应用，如求节点数，求树高,等等还有很多，反正大部分都是递归的过程

4.根据先序，中序构建二叉树

5.用单链表将二叉树的叶子节点链起来（这个我就不写了，实际上就是找到叶子节点，之后的就是链表操作了）

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct Node{

char c;

Node \*l,\*r;

}Bnode,\*Btree;

void create(Btree& T){

char c;

scanf("%c",&c);

if(c == ' ') {

T = NULL;

return ;

}

T = new Bnode;

T->c = c;

T->l = NULL;

T->r = NULL;

create(T->l);

create(T->r);

return ;

}

//三种遍历顺序 递归实现

void pre\_traverse(Btree T){

if(T == NULL) return ;

cout<<T->c<<" ";

pre\_traverse(T->l);

pre\_traverse(T->r);

return ;

}

void inorder\_traverse(Btree T){

if(T == NULL) return ;

inorder\_traverse(T->l);

cout<<T->c<<" ";

inorder\_traverse(T->r);

return ;

}

void suf\_traverse(Btree T){

if(T == NULL) return ;

suf\_traverse(T->l);

suf\_traverse(T->r);

cout<<T->c<<" ";

return ;

}

//三种遍历顺序 非递归（用栈实现）

struct Status{

Bnode\* p;

bool r\_search;

};

void pre\_traverse\_stack(Btree T){

Bnode\* s[1000];

int tail = 0;

Bnode\* w = T;

while(tail || w){

while(w){

cout<<w->c<<" ";

s[tail++] = w;

w = w->l;

}

if(tail){

w = s[--tail];

w = w->r;

}

}

return ;

}

void inorder\_traverse\_stack(Btree T){

Bnode\* s[1000];

int tail = 0;

Bnode\* w = T;

while(tail || w){

while(w){

s[tail++] = w;

w = w->l;

}

if(tail){

w = s[--tail];

cout<<w->c<<" ";

w = w->r;

}

}

return ;

}

void suf\_traverse\_stack(Btree T){

Status s[1000]; //因为后续遍历需要将左右儿子都输出之后在输出自己，所以必须要有一个量可以表明这一点

int tail = 0;

Bnode\* w = T;

Status e;

while(tail || w){

while(w){

e.p = w;

e.r\_search = false;

s[tail++] = e;

w = w->l;

}

if(tail){

e = s[--tail];

w = e.p;

if(e.r\_search == false){

w = w->r;

e.r\_search = true;

s[tail++] = e;

}

else {

cout<<w->c<<" ";

w = NULL;

}

}

}

return ;

}

//层序遍历

void sequence\_traverse(Btree T){

Bnode\* q[1000];

int head = 0,tail = 0;

Bnode\* p = T;

q[tail++] = p;

while(tail - head){

p = q[head++];

cout<<p->c<<" ";

if(p->l) q[tail++] = p->l;

if(p->r) q[tail++] = p->r;

}

return ;

}

//求节点数（其他求树高这些我也不写了，代码差不多，包括什么求最值这些）

int get\_size(Btree T){

if(T == NULL) return 0;

int l\_size = get\_size(T->l);

int r\_size = get\_size(T->r);

return l\_size + r\_size + 1;

}

int get\_height(Btree T){

if(T == NULL) return 0;

int l\_h = get\_height(T->l);

int r\_h = get\_height(T->r);

int max\_ = (l\_h>r\_h)?l\_h:r\_h;

return max\_+1;

}

// 根据先序以及中序构建二叉树

int find(char in[],int l,int r,char x){ //在序列in[l.....r]中找到x的位置并返回这个位置

int ret;

for(int i=l; i<=r; i++){

if(in[i] == x){

ret = i;

break;

}

}

return ret;

}

void get\_tree(Btree& T,char pre[],char in[],int& rt,int l,int r){ //建立起以pre[rt]为根，中序序列为[l,r]的树

if(l > r){

T = NULL;

return ;

}

int gap = find(in,l,r,pre[rt]);

T = new Bnode;

T->c = pre[rt];

T->l = NULL;

T->r = NULL;

rt++;

get\_tree(T->l,pre,in,rt,l,gap-1);

get\_tree(T->r,pre,in,rt,gap+1,r);

return ;

}

int main(){

Btree T = NULL;

create(T);

//

cout<<"这是先中后序遍历结果（递归实现的）"<<endl;

pre\_traverse(T); cout<<endl;

inorder\_traverse(T); cout<<endl;

suf\_traverse(T); cout<<endl;

//

cout<<"这是先中后序遍历结果（非递归实现的）"<<endl;

pre\_traverse\_stack(T); cout<<endl;

inorder\_traverse\_stack(T); cout<<endl;

suf\_traverse\_stack(T); cout<<endl;

//

cout<<"这是层序遍历的结果"<<endl;

sequence\_traverse(T); cout<<endl;

//

cout<<"节点数为"<<get\_size(T)<<endl;

cout<<"树的高度"<<get\_height(T)<<endl;

//

cout<<"请输入元素的个数n以及先序序列及其相应的中序序列（注意没有重复元素，以字符形式输入）"<<endl;

int n;

char pre[1000];

char in[1000];

cin>>n;

for(int i=1; i<=n; i++) cin>>pre[i];

for(int i=1; i<=n; i++) cin>>in[i];

Btree P = NULL;

int rt = 1;

get\_tree(P,pre,in,rt,1,n);

cout<<"这个二叉树的后续序列是"<<endl;

suf\_traverse(P);

return 0;

}

二叉排序树里面的二叉树

#include<bits/stdc++.h>

using namespace std;

typedef struct node{

int data;

node \*l,\*r;

} Bnode,\* Btree; // 节点,Btree是指向Bnode的指针

bool find(Bnode\* T,int x,Bnode\*\* f,Bnode\*\* c){

Bnode\* q = NULL; //q记录p节点的父亲

Bnode\* p = T; // 当前比较的节点p

if(!T){ //空树

\*f = q;

\*c = p;

return false;

}

else { //开始寻找x

while(p){

if(p->data == x) { //p节点的值就是x，找到了！

\*f = q;

\*c = p;// c = &child

return true;

}

else { // p所指向的节点不是的所有查找的节点

q = p;

// 判断是走向左儿子还是右儿子

if(x < p->data) p = p->l;

else p = p->r;

}

}

}

//到这里还没找到说明在这颗树里面没有x

\*f = q;

\*c = p;

return false;

}

// 向树 T中插入x，但考虑到T指针可能被更改，所以传入T的地址即（&T）

void insert(Bnode\*\* pT,int x){

Bnode\* fa = NULL;

Bnode\* child = NULL;

bool flag = find(\*pT,x,&fa,&child);

if(!flag){ //没找到，所以进行插入insert

// 因为要插入节点了，我们先把这个节点申请出来

Bnode\* p = new Bnode;

p->data = x;

p->l = NULL;

p->r = NULL;

// 插入在正确的位置

if(fa == NULL){ //fa就等于空，所以说明插入在空树里面，那我们把根节点改成p就行了

\*pT = p;

}

else { //不是空树，那我们要将p节点接在fa节点的正确位置，是左儿子还是右儿子

if(x < fa->data) fa->l = p;

else fa->r = p;

}

}

return ;

}

void remove(Bnode\*\* pT,int x){ //在T这个树里面删去值为x的节点

Bnode\* fa = NULL;

Bnode\* child = NULL;

bool flag = find(\*pT,x,&fa,&child);

if(flag){ //如果找到了x，那么我们才进行删除

if(fa == NULL){ // 如果找到了，并且找到的节点（child）并没有父亲（fa == NULL）说明什么，是不是说明要删除的节点child就是根节点

if(child->l == NULL && child->r == NULL){ // child左右儿子都没有

\*pT = NULL;

}

else if(child->l != NULL && child->r == NULL){ // 左支

\*pT = child->l;

}

else if(child->l == NULL && child->r != NULL){ // 右支

\*pT = child->r;

}

else if(child->l != NULL && child->r != NULL){ // 双支

\*pT = child->l; //将左支挂上去

Bnode\* p = child->l; //用p来找到左支的最右边的端点

while(p->r) p = p->r;

p->r = child->r; // 然后将右支挂在左支的最右边

}

}

else if(fa != NULL){ //要删除的节点不是根节点

if(child->data < fa->data){ // 说明child是fa的左儿子

if(child->l == NULL && child->r == NULL){ // child左右儿子都没有

fa->l = NULL;

}

else if(child->l != NULL && child->r == NULL){ // 左支

fa->l = child->l;

}

else if(child->l == NULL && child->r != NULL){ // 右支

fa->l = child->r;

}

else if(child->l != NULL && child->r != NULL){ // 双支

fa->l = child->l; //将左支挂上去

Bnode\* p = child->l; //用p来找到左支的最右边的端点

while(p->r) p = p->r;

p->r = child->r; // 然后将右支挂在左支的最右边

}

}

else { // 说明child是fa的右儿子

if(child->l == NULL && child->r == NULL){ // child左右儿子都没有

fa->r = NULL;

}

else if(child->l != NULL && child->r == NULL){ // 左支

fa->r = child->l;

}

else if(child->l == NULL && child->r != NULL){ // 右支

fa->r = child->r;

}

else if(child->l != NULL && child->r != NULL){ // 双支

fa->r = child->l; //将左支挂上去

Bnode\* p = child->l; //用p来找到左支的最右边的端点

while(p->r) p = p->r;

p->r = child->r; // 然后将右支挂在左支的最右边

}

}

}

//注意结束以后delete child

delete child;

}

return ;

}

void inorder\_traver(Bnode\* p){

if(p == NULL) return ;

inorder\_traver(p->l);

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

inorder\_traver(p->r);

return ;

}

int main(){

Bnode\* T = NULL;

puts("输入元素的个数n，以及这n个元素");

int n;

cin>>n;

// 输入n个数据

int v;

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

cin>>v;

insert(&T,v);

}

puts("这是中序遍历的结果");

inorder\_traver(T);cout<<endl;

int re;

puts("请输入要插入的元素的值");

cin>>re;

insert(&T,re);

puts("插入这个元素值之后的结果");

inorder\_traver(T);cout<<endl;

puts("请输入要删除的元素的元素值");

cin>>re;

remove(&T,re);

puts("插入之后的结果");

inorder\_traver(T);cout<<endl;

return 0;

}

二叉排序树里的二叉树简化版

#include<bits/stdc++.h>

using namespace std;

typedef struct node{

int data;

node \*l,\*r;

} Bnode,\* Btree; // 节点,Btree是指向Bnode的指针

bool find(Bnode\* T,int x,Bnode\*\* f,Bnode\*\* c){

Bnode\* q = NULL; //q记录p节点的父亲

Bnode\* p = T; // 当前比较的节点p

if(!T){ //空树

\*f = q;

\*c = p;

return false;

}

else { //开始寻找x

while(p){

if(p->data == x) { //p节点的值就是x，找到了！

\*f = q;

\*c = p;// c = &child

return true;

}

else { // p所指向的节点不是的所有查找的节点

q = p;

// 判断是走向左儿子还是右儿子

if(x < p->data) p = p->l;

else p = p->r;

}

}

}

//到这里还没找到说明在这颗树里面没有x

\*f = q;

\*c = p;

return false;

}

// 向树 T中插入x，但考虑到T指针可能被更改，所以传入T的地址即（&T）

void insert(Bnode\*\* pT,int x){

Bnode\* fa = NULL;

Bnode\* child = NULL;

bool flag = find(\*pT,x,&fa,&child);

if(!flag){ //没找到，所以进行插入insert

// 因为要插入节点了，我们先把这个节点申请出来

Bnode\* p = new Bnode;

p->data = x;

p->l = NULL;

p->r = NULL;

// 插入在正确的位置

if(fa == NULL){ //fa就等于空，所以说明插入在空树里面，那我们把根节点改成p就行了

\*pT = p;

}

else { //不是空树，那我们要将p节点接在fa节点的正确位置，是左儿子还是右儿子

if(x < fa->data) fa->l = p;

else fa->r = p;

}

}

return ;

}

void remove(Bnode\*\* pT,int x){ //在T这个树里面删去值为x的节点

Bnode\* fa = NULL;

Bnode\* child = NULL;

bool flag = find(\*pT,x,&fa,&child);

if(flag){ //如果找到了x，那么我们才进行删除

Bnode\*\* com = NULL;

//用com指向了要要改变的变量

if(fa == NULL) com = pT;

else if(child->data < fa->data) com = &(fa->l);

else com = &(fa->r);

if(child->l == NULL && child->r == NULL) \*com = NULL;

else if(child->l != NULL && child->r == NULL) \*com = child->l;

else if(child->l == NULL && child->r != NULL) \*com = child->r;

else if(child->l != NULL && child->r != NULL){ // 双支

\*com = child->l; //将左支挂上去

Bnode\* p = child->l; //用p来找到左支的最右边的端点

while(p->r) p = p->r;

p->r = child->r; // 然后将右支挂在左支的最右边

}

//注意结束以后delete child

delete child;

}

return ;

}

void inorder\_traver(Bnode\* p){

if(p == NULL) return ;

inorder\_traver(p->l);

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

inorder\_traver(p->r);

return ;

}

int main(){

Bnode\* T = NULL;

puts("输入元素的个数n，以及这n个元素");

int n;

cin>>n;

// 输入n个数据

int v;

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

cin>>v;

insert(&T,v);

}

puts("这是中序遍历的结果");

inorder\_traver(T);cout<<endl;

int re;

puts("请输入要插入的元素的值");

cin>>re;

insert(&T,re);

puts("插入这个元素值之后的结果");

inorder\_traver(T);cout<<endl;

puts("请输入要删除的元素的元素值");

cin>>re;

remove(&T,re);

puts("插入之后的结果");

inorder\_traver(T);cout<<endl;

return 0;

}

### 栈和队列

##### 栈

用数组实现的栈及其一些操作

#include<cstdio>

#include<iostream>

#define maxn\_capacity 1000

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.这是栈的定义以及其一些基本操作

2.因为这是专门给栈写的，所以有关操作的代码我都写成了函数，但实际上是不用写成函

数的，平常使用栈的时候我们可以就一行代码把栈定义出来，包括其有关操作都可以一行代码实现！通常我们写代码就写成这种就够了，没必要进行一次封装！

所以这个栈的代码看看怎么实现这几个操作就好了!!!!!!!!!在后面，你会明显的发现我们并没有使用这个自己封装好的栈！！！

3.对于栈的实现，我们可以使用数组实现（此时栈的大小就被固定了，或者说容量就被固定了），我们也可以使用链表的

形式实现栈（称为链栈），此时栈的容量便不固定（或者说容量不固定），一个是静态的一个是动态的。但不管用哪种方法实现栈，

栈本身的含义一直没有变，那就是后进先出的顺序，我们可以用这个来进行搜索（具体就是深度优先搜索，变的只是栈能储存数据的大小

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

对于定义栈的函数就不写了，实际上就是定义一个数组，不管你是new（或者是malloc）出来的数组，还是在主函数里面定义的数组，反正就是一个数组

对于一个栈，一些基本操作就是

top()取出栈顶的元素

pop()弹出栈顶的元素

push()将元素放入栈

get\_size()返回栈此时储存的数据的个数

empty()返回此时栈是否为空

full() 返回栈此时是否已经满了

\*/

struct Stack{

//定义一个栈 s，假设这个栈储存的数据为int型的，那我们就定义一个int类型的数组即可，如果要存储其他类型的数据，定义成结构体数组即可

int data[maxn\_capacity];//这个maxn\_capacity是由你自己去定义这个栈的大小，这就是数组栈的局限，数组栈的最大大小在定义的时候就固定了

//这个maxn\_capacity可以使用宏定义，也可以定义成const int 类型的全局变量，两者都可以，此处我们让maxn\_capacity为1000

int tail; //对于栈，它还需要一个与之配套的tail下标（也可以理解成是这个数组的指针）,注意tail要初始化为0

};

//这几个函数的设计由自己去构建

bool top(Stack& s,int& ret){ //栈顶元素放在ret中返回回去

if(!s.tail) return false; // 栈此时为空，没有栈顶元素

ret = s.data[s.tail-1];

return true;

}

bool pop(Stack& s){

if(!s.tail) return false;

else s.tail--;

return true;

}

bool full(Stack& s){

return s.tail == maxn\_capacity;

}

bool push(Stack& s,int val){

if( full(s) ) return false;

else s.data[s.tail++] = val;

return true;

}

int get\_size(Stack& s){

return s.tail;

}

bool empty(Stack& s){

return s.tail == 0;

}

int main(){

Stack s; //定义一个栈s

//初始化 tail = 0

s.tail = 0;

//输入一些元素试试

puts("输入元素的数量n，以及这n个元素");

int n,val;

cin>>n;

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

cin>>val;

push(s,val);

}

puts("弹出这些元素并输出出来");

int ret;

while(!empty(s)){

top(s,ret);

pop(s);

cout<<ret<<" ";

}

cout<<endl;

return 0;

}

用单链表（不带表头结点）实现的栈（链栈）

#include<cstdio>

#include<cstdlib>

#include<iostream>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.这是使用不带表头节点的单链表实现的栈

2.应该知道的是，用数组实现与用链表实现只是实现栈的储存结构不同，栈的意义是没有变的，一样是后进先出的顺序

3.实际上这份代码就是对单链表的一些操作，不过对于栈的push()相当于对这个链表的头插入，pop()相

当于从头部删除，top()相当于这个链表的第一个节点的值

4.链栈看看相应的实现就好了，实际上单链表的操作如果熟练的话，链栈只用知道栈的意义便可以很轻松的实现出来

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef struct Node{

int data;

Node\* next;

} sNode,\*link\_stack;

sNode\* Malloc\_Node(int v,sNode\* next){ //申请一节点sNode,sNode->data = v,sNode -> next = next

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

p->data = v;

p->next = next;

return p;

}

bool top(link\_stack s,int& ret){

if(s) ret = s->data;

else return false;

return true;

}

void push(link\_stack& s,int v){ // 将元素v放入链栈s，实际上就是头插入

sNode\* p = Malloc\_Node(v,s);

s = p;

return ;

}

void pop(link\_stack& s){ // 将链栈s的top元素弹出,若s为空则不做操作，实际上就是删去头节点

if(s) {

sNode\* del = s;

s = s->next;

free(del);

}

return ;

}

bool empty(link\_stack s){

return s == NULL;

}

void ini(link\_stack& s){ // 初始化链栈s,s = NULL;

s = NULL;

return ;

}

void disp(link\_stack& s){ // 从栈s的栈顶打印栈中的元素

sNode\* p = s;

while(p){

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

p = p->next;

}

puts("");

return ;

}

int main(){

puts("输入元素,并输入0表示输入结束");

int v;

link\_stack s;

ini(s);

while(1){

cin>>v;

if(!v) break;

push(s,v);

}

puts("栈中元素");

disp(s);

puts("pop出两个元素之后");

pop(s);

pop(s);

disp(s);

return 0;

}

##### 队列

用数组实现队列

#include<iostream>

#define maxn\_capacity 100000

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.队列和栈的区别就是，队列是先进先出的

2.实现的时候用一个数组和两个小标head，tail就可以了

3.同队列一样，我们实际上使用的时候不用把队列封装成这样，这些对应操作的函数都是一行代码就可以解决的

4.所以也只用看看实现就好，后面具体应用的时候，比如走迷宫，比如层

序遍历这些，我们都只用定义一个数组，一个head一个tail就可以解决了！！

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct queue{

int data[maxn\_capacity]; //maxn\_capacity决定了这个队列最大的储存数量，这里我们使用的是宏定义，认为最大为100000

int head,tail; //head指向当前队列的第一个元素，tail指向第一个不存数据的位置，即 [head,tail) 的范围内表示当前队列中的元素

};

void ini(queue& q){//队列的初始化

q.head = q.tail = 0;

return ;

}

bool empty(queue& q){ // 返回是否队列q为空

return q.head == q.tail;

}

bool push(queue& q,int v){ //从队尾加入，如果装不下了，就返回false

if(q.tail < maxn\_capacity) q.data[q.tail++] = v;

else return false;

return true;

}

bool front(queue& q,int& ret){ //ret中带出队首元素，如果队列为空，即head == tail，则返回false

if(!empty(q)) ret = q.data[q.head];

else return false;

return true;

}

bool pop(queue& q){ //弹出队首元素，如果队列为空，则弹出失败，返回false

if(!empty(q)) q.head++;

else return false;

return true;

}

int main(){

queue q;

ini(q);

//输入一些元素

cout<<"请输入元素的数量n，以及这n个元素"<<endl;

int n,val;

cin>>n;

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

cin>>val;

push(q,val);

}

cout<<"将队列中的元素全部弹出"<<endl;

int r;

while(!empty(q)){

front(q,r);

pop(q);

cout<<r<<" ";

}

cout<<endl;

return 0;

}

循环队列

#include<iostream>

#define maxn\_capacity 100000

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.这个循环队列与普通队列在大体上是一样的，主要是在达到maxn\_capacity的时候回到0开始，即循环起来

2.实际上的循环就是利用取模，容量为maxn\_capacity的循环队列可以在[0,maxn\_capacity-1]位置储存数据，

3.看看相应函数与不循环的队列的差别

4.注意判满的条件

5.看完之后，你会发现所谓循环队列，他就是在尝试能否能使用已经被弹出元素的空间，而不像前面的普通数组一样，tail一直在 tail++ head也一直在head++

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct loop\_queue{

int data[maxn\_capacity]; //maxn\_capacity 100000

int head,tail;

};

void ini(loop\_queue& q){ //初始化 即head = tail = 0

q.head = q.tail = 0;

return ;

}

bool empty(loop\_queue& q){

return q.head == q.tail;

}

bool full(loop\_queue& q){

/\*

每一次往队列里面加入元素的位置应该是q.tail处，放完元素之后

tail应该变为 (q.tail+1)%maxn\_capacity，如果此时tail加1且取模之后发现与head相等，说明此时队列已经慢了，不能往里面放元素了

我们相当于浪费一个空间来判定是否此时队列已经满了

比如 循环队列的maxn\_capacity为5

此时head = 0，tail = 4，如果要往这里面放入元素，则放在tail（4号位置上），放完之后tail = (4+1)%5 = 0，此时tail == head了

尽管4号位置本身是没有放入元素的，但是我们认为此时循环队列还是满了，因为4号位置放完元素之后tail就等于head了，所以我们认为此时不能

再往循环队列中加入元素

做这样的目的主要是要是判队列为空（tail == head）和判队列为满的条件分开，不然我们无法在循环队列中判断到底是空还是满

!!!在这种意义下，容量为maxn\_capacity的循环队列实际上只能储存这maxn\_capacity-1 个数据，因为有一个空间不会被使用

\*/

return (q.tail+1)%maxn\_capacity == q.head;

}

bool push(loop\_queue& q,int val){

if(!full(q)) {

q.data[q.tail] = val;

q.tail = (q.tail+1)%maxn\_capacity;

}

else return false; //此时队列已经满了，无法再放入元素

return true;

}

bool top(loop\_queue& q,int& ret){

if(!empty(q)) ret = q.data[q.head];

else return false; //此时队列为空，根本就没有首元素

return true;

}

bool pop(loop\_queue& q){

if(!empty(q)){

q.head = (q.head+1)%maxn\_capacity; //始终要注意，此时队列是循环起来的，所以一定要记得取模

}

else return false; //队列为空，根本就没有元素可以弹出

return true;

}

int main(){

loop\_queue q;

ini(q);

//输入一些元素试试

cout<<"请输入元素的个数n，以及这n个元素"<<endl;

int n,val;

cin>>n;

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

cin>>val;

push(q,val);

}

//弹出这些元素

cout<<"弹出这些元素"<<endl;

int r;

while(!empty(q)){

top(q,r);

pop(q);

cout<<r<<" ";

}

cout<<endl;

return 0;

}

##### 走迷宫

栈走迷宫

#include<iostream>

#define maxn\_row 100

#define maxn\_col 100

#define maxn\_capacity maxn\_row\*maxn\_col

using namespace std;

/\*

第一种find\_one\_path() 见15行

第二种find\_one\_path() 见74行

第一种find\_all\_paths() 见141行

第二种find\_all\_paths() 见201行

\*/

struct Status{ //栈里面储存的数据，我们将其称为状态，栈里面储存了状态，栈里面储存的所有状态表示了我从起点走到当前位置的路径，包括在每个点是往哪一个方向前进的

int x,y,d;

};

bool is\_ok(int G[][maxn\_col],bool vis[][maxn\_col],int x,int y){

return G[x][y] == 0 && vis[x][y] == false;

}

//// @第一种 find\_one\_path函数 ， 我更倾向于第二种find\_one\_path的逻辑（见71行）

//void find\_one\_path(int G[][maxn\_col],int n,int m,int sx,int sy,int ex,int ey,const int dir[][2]){

// bool vis[maxn\_row][maxn\_col]; //vis数组是用来标记当前已经走过的点

// Status s[maxn\_capacity]; //把栈定义出来，注意栈的大小应该至少为地图的方格数，即maxn\_row\*maxn\_col，或者说用户输入的地图的n，m，那么最大至少为n\*m

// int tail = 0;

// //初始化 vis

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

// for(int j=1; j<=m; j++){

// vis[i][j] = false;

// }

// }

// /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 走迷宫的逻辑就是我的栈里面储存我走到当前点(now\_x,now\_y)的路径上的点以及其状态（就是方向）

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//

// Status e;

// // 起点入栈

// e.x = sx;

// e.y = sy;

// e.d = -1;

// s[tail++] = e;

// vis[sx][sy] = true;

//

// int now\_x = 0,now\_y = 0; //当前所在的位置

// while(tail){

// if(is\_ok(G,vis,now\_x,now\_y)){ //当前位置可以走则继续走

// if(now\_x == ex && now\_y == ey){ //走到了终点，输出这条路径

// puts("找到了一条路径");

// for(int i=0; i<tail; i++) printf("(%d,%d) ",s[i].x,s[i].y); //！！最好不要这样输出，尽管很方便，但是我觉得还是老老实实的倒叙输出

// printf("(%d,%d)\n",ex,ey); //把终点补上

// return ;

// }

// //没走到终点继续走

// e.x = now\_x;

// e.y = now\_y;

// e.d = 0;

// s[tail++] = e;

// vis[now\_x][now\_y] = true;

//

// now\_x = now\_x+dir[0][0];

// now\_y = now\_y+dir[0][1];

// }

// else {

// while(tail){ //退栈，找到第一个4个方向没有被搜索完的

// e = s[--tail]; //注意这里与找所有路径时的vis的区别！！！

// if(e.d < 3) break;

// }

// if(e.d < 3){

// e.d++;

// //走到下一步

// now\_x = e.x + dir[e.d][0];

// now\_y = e.y + dir[e.d][1];

// s[tail++] = e; //入栈

// }

// }

// }

// return ;

//}

//// @第二种find\_one\_path()函数，我认为这个想法更加的清晰一些 ， 具体看自己的理解

//void find\_one\_path(int G[][maxn\_col],int n,int m,int sx,int sy,int ex,int ey,const int dir[][2]){

// bool vis[maxn\_row][maxn\_col]; //vis数组是用来标记当前已经走过的点

// Status s[maxn\_capacity]; //把栈定义出来，注意栈的大小应该至少为地图的方格数，即maxn\_row\*maxn\_col，或者说用户输入的地图的n，m，那么最大至少为n\*m

// int tail = 0;

// //初始化 vis

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

// for(int j=1; j<=m; j++){

// vis[i][j] = false;

// }

// }

// /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 走迷宫的逻辑就是我的栈里面储存我走到当前点(now\_x,now\_y)的路径上的点以及其状态（就是方向）

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//

// Status e;

// // 起点入栈

// e.x = sx;

// e.y = sy;

// e.d = -1;

// s[tail++] = e;

// vis[sx][sy] = true;

//

// int now\_x = 0,now\_y = 0; //当前所在的位置

// while(tail){

// //找到下一步往哪里走，即(now\_x,now\_y)应该是多少

// e = s[--tail];

// if(e.d == 3) continue;

// e.d++;

// now\_x = e.x + dir[e.d][0];

// now\_y = e.y + dir[e.d][1];

// s[tail++] = e;

// //

// if(now\_x == ex && now\_y == ey){ //找到了，输出这一条路径

// /\*

// for(int i=0; i<tail; i++) printf("(%d,%d) ",s[i].x,s[i].y);

// 这样输出路径是可以的，就直接是正确的顺序，但是对于一个栈而言，我们这样做有点不合规矩，因为一个栈

// 我们是不应该取访问除栈顶以外的元素的，所以这里我们就直接倒叙输出

// \*/

// puts("倒叙输出这一条路径");

// printf("(%d,%d) ",ex,ey);

// while(tail){

// e = s[--tail];

// printf("(%d,%d) ",e.x,e.y);

// }

// puts("");

// return ;

// }

// if(is\_ok(G,vis,now\_x,now\_y)){ //如果这一点可以走，那么我们就将其加入栈中

// e.x = now\_x;

// e.y = now\_y;

// e.d = -1;

// s[tail++] = e;

// vis[now\_x][now\_y] = true;

// }

// }

// return ;

//}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.找所有路径的函数其实就是找一条路径的函数做一些修改

2.找所有路径的代码与找一天路径差不多，细节在于vis，找到了不要返回（这是找所有路的关键）

3.我仍然觉得实现起来第二种的逻辑更加的清晰，实现起来代码量也更小，具体看自己的理解

4.(now\_x,now\_y)仍然表示当前走到的位置

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// @第一种find\_all\_paths()的写法

void find\_all\_paths(int G[][maxn\_col],int n,int m,int sx,int sy,int ex,int ey,const int dir[][2]){

bool vis[maxn\_row][maxn\_col]; //vis数组是用来标记当前已经走过的点

Status s[maxn\_capacity]; //把栈定义出来，注意栈的大小应该至少为地图的方格数，即maxn\_row\*maxn\_col，或者说用户输入的地图的n，m，那么最大至少为n\*m

int tail = 0;

int num = 0; // 统计路径的数量

//初始化 vis

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

for(int j=1; j<=m; j++){

vis[i][j] = false;

}

}

Status e;

// 起点入栈

e.x = sx;

e.y = sy;

e.d = -1;

s[tail++] = e;

vis[sx][sy] = true;

int now\_x = 0,now\_y = 0; //当前所在的位置

while(tail){

if(is\_ok(G,vis,now\_x,now\_y) && !(now\_x == ex && now\_y == ey)){ //(now\_x,now\_y)可以走并且不是终点

e.x = now\_x;

e.y = now\_y;

e.d = 0;

s[tail++] = e;

vis[e.x][e.y] = true;

//走到下一步

now\_x = now\_x + dir[0][0];

now\_y = now\_y + dir[0][1];

}

else { //(now\_x,now\_y)不可走或者是(now\_x,now\_y)是终点

if(now\_x == ex && now\_y == ey){ //找到一条路径，输出

num++;

for(int i=0; i<tail; i++) printf("(%d,%d) ",s[i].x,s[i].y);

printf("(%d,%d)",ex,ey);

puts("");

}

while(tail){ //退栈，找到第一个四个方向没有搜索完的状态e

e = s[--tail];

vis[e.x][e.y] = false;

if(e.d < 3) break;

}

if(e.d < 3){

//走向下一步

e.d++;

now\_x = e.x + dir[e.d][0];

now\_y = e.y + dir[e.d][1];

//当前状态入栈

s[tail++] = e;

vis[e.x][e.y] = true;

}

}

}

printf("一共有%d条路径\n",num);

return ;

}

//// @第二种find\_all\_paths()的写法

//void find\_all\_paths(int G[][maxn\_col],int n,int m,int sx,int sy,int ex,int ey,const int dir[][2]){

//

// bool vis[maxn\_row][maxn\_col]; //vis数组是用来标记当前已经走过的点

// Status s[maxn\_capacity]; //把栈定义出来，注意栈的大小应该至少为地图的方格数，即maxn\_row\*maxn\_col，或者说用户输入的地图的n，m，那么最大至少为n\*m

// int tail = 0;

// int num = 0; // 统计路径的数量

// //初始化 vis

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

// for(int j=1; j<=m; j++){

// vis[i][j] = false;

// }

// }

// Status e;

// // 起点入栈

// e.x = sx;

// e.y = sy;

// e.d = -1;

// s[tail++] = e;

// vis[sx][sy] = true;

//

// int now\_x = 0,now\_y = 0; //当前所在的位置

// while(tail){

// //找到下一步走哪里 即 (now\_x,now\_y)的值是多少

// e = s[--tail];

// vis[e.x][e.y] = false; //第一处不一样

// if(e.d == 3) continue; //该点所有方向都被搜索了，直接continue，继续弹出下一个状态

// e.d++;

// now\_x = e.x + dir[e.d][0];

// now\_y = e.y + dir[e.d][1];

// s[tail++] = e;

// vis[e.x][e.y] = true; //第二处不一样

//

// //

// if(now\_x == ex && now\_y == ey){ //找到了，输出这一条路径

// /\*

// for(int i=0; i<tail; i++) printf("(%d,%d) ",s[i].x,s[i].y);

// 这样输出路径是可以的，就直接是正确的顺序，但是对于一个栈而言，我们这样做有点不合规矩，因为一个栈

// 我们是不应该取访问除栈顶以外的元素的

// \*/

// num++; // 计数加一

// //因为在这里我们为了输出所有的路径，需要保留栈里面的信息，不能让栈直接退栈把元素都退完了，所以我们就直接输出，虽然很不好qwq

// for(int i=0; i<tail; i++)printf("(%d,%d) ",s[i].x,s[i].y); //我们能这样做的原因是我们直接使用数组模拟栈，所以输出栈里面所有的元素是很轻松的，但是总感觉有点不好

// printf("(%d,%d)",ex,ey);

// puts("");

// //仅仅输出，不要return

// continue; //第三处不一样！！！ 不要return来直接结束函数，而是应该继续找下一条路径，所以continue就行了

// }

// if(is\_ok(G,vis,now\_x,now\_y)){ //如果这一点可以走，那么我们就将其加入栈中

// e.x = now\_x;

// e.y = now\_y;

// e.d = -1;

// s[tail++] = e;

// vis[now\_x][now\_y] = true;

// }

// }

// printf("一共有%d条路径\n",num);

// return ;

//}

int main(){

const int dir[4][2] = {{-1,0},{0,1},{1,0},{0,-1}};//4个方向 上下左右

int G[maxn\_row][maxn\_col]; //定义出地图 maxn\_row 1000,maxn\_col 1000 int n,m; //地图的大小

int sx,sy,ex,ey; // (sx,sy)表示入口坐标 (ex,ey)表示出口坐标

int n,m;

puts("请输入地图的行数n以及列数m,以及这个地图，规定1代表此处不可走，0代表此处可走");

//用户输入地图

cin>>n>>m;

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

for(int j=1; j<=m; j++){

cin>>G[i][j];

}

}

puts("请输入入口和出口的坐标，请保证入口，出口是合法的");

cin>>sx>>sy>>ex>>ey;

//为了程序更加的简洁，我们在地图的最外面加一圈不可走的点

for(int i=0; i<=n+1; i++) G[i][0] = G[i][m+1] = 1;

for(int j=0; j<=m+1; j++) G[0][j] = G[n+1][j] = 1;

//

// puts("其中一条可行路径");

// find\_one\_path(G,n,m,sx,sy,ex,ey,dir);

puts("所有路径");

find\_all\_paths(G,n,m,sx,sy,ex,ey,dir);

return 0;

}

队列走迷宫

#include<iostream>

#include<cstdio>

#define maxn\_row 100

#define maxn\_col 100

using namespace std;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1.用队列走迷宫，就是层序搜索，或者叫广度优先搜索（BFS）

2.广度优先搜索就是一层一层的搜索，从这一点出发，你便会发现，我第一次搜索到终点，那么这一条路径一定是长度最短的

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct Status{

int x,y;

int from;//代表自己这个状态是由哪一个状态来的

};

void find\_one\_shortest\_path(int G[][maxn\_col],int n,int m,int sx,int sy,int ex,int ey,const int dir[][2]){

bool vis[maxn\_row][maxn\_col];

Status q[maxn\_row\*maxn\_col];

int head = 0, tail = 0;

//起点入队

Status e;

e.x = sx;

e.y = sy;

e.from = -1;

q[tail++] = e;

vis[sx][sy] = true;

int nx,ny;

Status next;

while(tail - head){

e = q[head++];

if(e.x == ex && e.y == ey){ //走到了终点，结束

//这样输出是倒序的

printf("(%d,%d) ",ex,ey);

while(e.from != -1){

e = q[e.from];

printf("(%d,%d) ",e.x,e.y);

}

puts("");

return ;

}

for(int i=0; i<=3; i++){

nx = e.x + dir[i][0];

ny = e.y + dir[i][1];

if(G[nx][ny] == 0 && vis[nx][ny] == false){

//入队

next.x = nx;

next.y = ny;

next.from = head-1;

q[tail++] = next;

vis[nx][ny] = true;

}

}

}

return ;

}

int main(){

const int dir[4][2] = {{-1,0},{0,1},{1,0},{0,-1}};//4个方向 上下左右

int G[maxn\_row][maxn\_col]; //定义出地图 maxn\_row 1000,maxn\_col 1000 int n,m; //地图的大小

int sx,sy,ex,ey; // (sx,sy)表示入口坐标 (ex,ey)表示出口坐标

int n,m;

puts("请输入地图的行数n以及列数m,以及这个地图，规定1代表此处不可走，0代表此处可走");

//用户输入地图

cin>>n>>m;

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

for(int j=1; j<=m; j++){

cin>>G[i][j];

}

}

puts("请输入入口和出口的坐标，请保证入口，出口是合法的");

cin>>sx>>sy>>ex>>ey;

//为了程序更加的简洁，我们在地图的最外面加一圈不可走的点

for(int i=0; i<=n+1; i++) G[i][0] = G[i][m+1] = 1;

for(int j=0; j<=m+1; j++) G[0][j] = G[n+1][j] = 1;

//

find\_one\_shortest\_path(G,n,m,sx,sy,ex,ey,dir);

return 0;

}