**要求代码和实验报告规范，在算法思想中：对实验涉及的数据结构进行有效设计和分析；对算法进行分析并给出时间、空间复杂度的结论；清晰表达实验思路、出现的问题及解决方法。**

一、调试成功程序及说明

1、

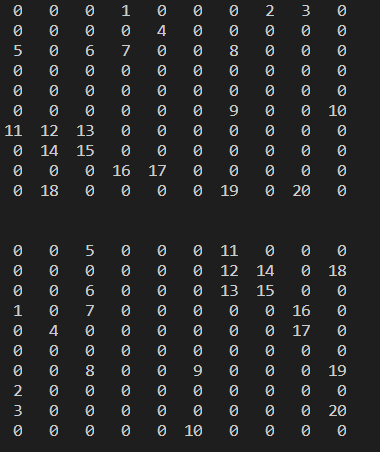
题目：输入稀疏矩阵，建立稀疏矩阵三元组顺序结构，实现矩阵的列序遍历转置和快速转置算法。

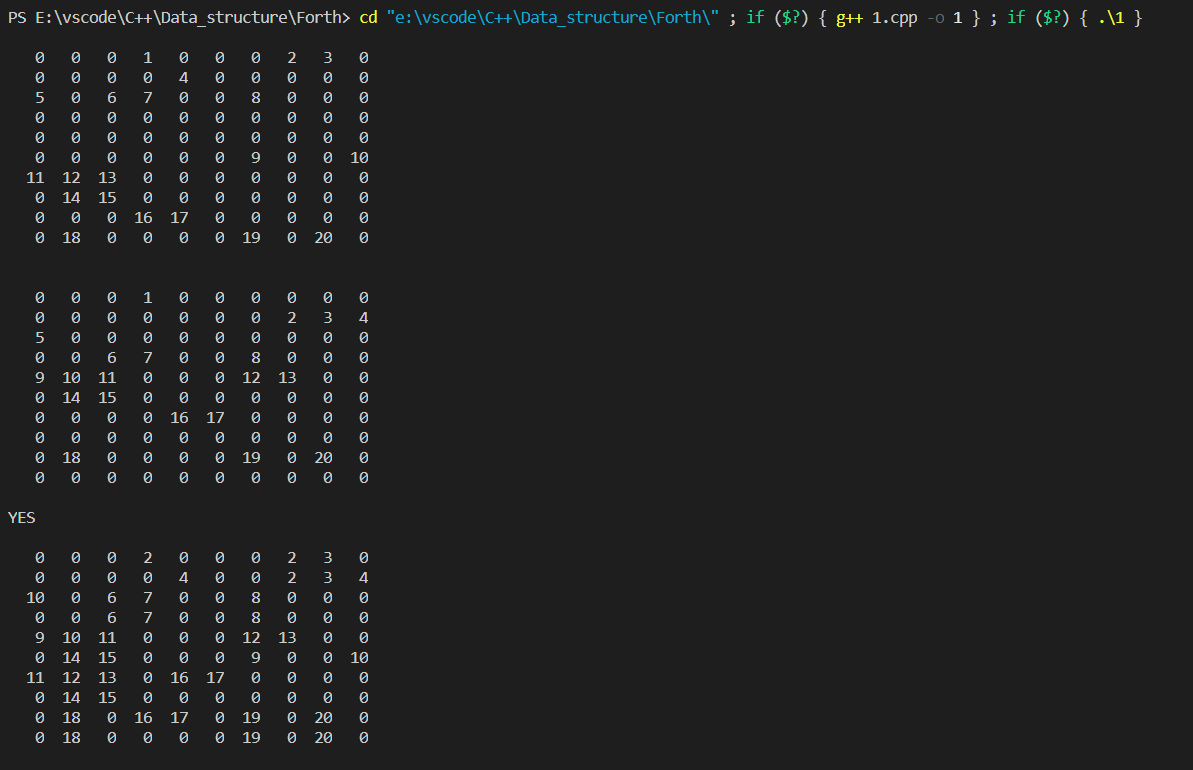
算法思想：

1.先建立基本操作，输入矩阵的方式，储存的结构为课本建议的结构体三元组储存。2.列序遍历的主要思想是先将行和列进行倒置，然后利用一个稳定排序的算法根据行号进行排序，即可获得转置后的结果。时间复杂度为（n\*cols）。

3.快速转置的方法为先对存放的三元组进行一次遍历，记录下每个列号有几个元素，然后根据这一信息，计算出每行开始的位置应该在哪里，第二次遍历三元组数组时候，能够直接查找出当前的元素应该放入的位置标号。时间复杂度为O（cols+n）。

4.实现了矩阵相加的操作，利用两个指针遍历三元组数组，选择标号较小的移动进目标矩阵，两个标号相同则数值相加，然后移入新的矩阵。

运行结果：



结果分析：运行结果正确，实现了矩阵的转置。

附源程序。

#include <iostream>

#include <cstdlib>

#include <string.h>

#include <iomanip>

using namespace std;

#define INITSIZE 256

#define INCRESIZE 128

#define MAXSIZE 512

#define ElemType int

typedef struct

{

int i, j; //矩阵行列标号均从1开始

ElemType value;

}Triple;

typedef struct

{

Triple \*arr;

int Rows, Cols, Nums; //行数为5，标号为1 - 5

int MatrixSize;

}SeqMatrix;

bool operator < (const Triple &a, const Triple &b)

{

if(a.i < b.i)

{

return true;

}

else if(a.i == b.i&& a.j < b.j)

{

return true;

}

else

{

return false;

}

}

bool operator == (const Triple &a, const Triple &b)

{

if ((a.i == b.i)&&(a.j == b.j))

{

return true;

}

else

{

return false;

}

}

bool InitMatrix(SeqMatrix &M)

{

M.arr = (Triple\*)malloc(sizeof(Triple)\*INITSIZE);

if (M.arr == NULL)

{

cout << "Memory is overflow !"<<endl;

exit (0);

}

M.MatrixSize = INITSIZE;

M.Rows = 0;

M.Cols = 0;

M.Nums = 0;

for (int i = 0; i < INITSIZE; i++)

{

M.arr[i].i = -1;

M.arr[i].j = -1;

M.arr[i].value = -1;

}

return true;

}

//初始化矩阵

bool CreateMatrix(SeqMatrix &M, int row, int col, int n, const int \*rows, const int \*cols)

//行数，列数，元素数，行元素数组，列元素数组

{

if (M.arr == NULL)

{

return false;

}

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

{

M.arr[i].i = rows[i];

M.arr[i].j = cols[i];

M.arr[i].value = i+1;

M.Nums ++;

}

M.Rows = row;

M.Cols = col;

return true;

}

//创建矩阵

bool DestroyMatrix(SeqMatrix &M)

{

M.Rows = 0;

M.Cols = 0;

M.Nums = 0;

if(M.arr != NULL)

{

free(M.arr);

M.arr = NULL;

}

return true;

}

//销毁矩阵

void PrintMatrix(SeqMatrix &M)

{

//cout << "[ ";

int row = 1, col = 1;

Triple \*p=M.arr;

int num = 0;

cout << endl;

while(row <= M.Rows)

{

if(p->i == row)

{

while(col <= M.Cols)

{

if((p->j == col)&&(p->i == row))

{

cout << setw(4) << p->value;

p++;

num ++;

}

else

{

cout << setw(4) << "0";

}

col++;

}

cout << endl;

}

else

{

while (col <= M.Cols)

{

cout << setw(4) << "0";

col ++;

}

cout << endl;

}

col = 1;

row ++;

//cout << "now row is "<< row <<" and p->i = "<<p->i<<" and num is "<<num<<endl;

}

cout <<endl;

return;

}

//打印矩阵

void CopyMatrix(const SeqMatrix &A, SeqMatrix &B)

{

B.Cols = A.Cols;

B.Rows = A.Rows;

B.Nums = A.Nums;

while(B.MatrixSize < A.MatrixSize)

{

B.arr = (Triple\*)realloc(B.arr,sizeof(Triple)\*(B.MatrixSize+INCRESIZE));

B.MatrixSize += INCRESIZE;

}

for (int i = 0; i < A.Nums; i++)

{

B.arr[i].i = A.arr[i].i;

B.arr[i].j = A.arr[i].j;

B.arr[i].value = A.arr[i].value;

}

}

//复制矩阵

bool AddMatrix(const SeqMatrix &A, const SeqMatrix &B, SeqMatrix &C)

{

if ((A.Cols != B.Cols)||(A.Rows != B.Rows))

{

return false;

}

C.Cols = A.Cols;

C.Rows = A.Rows;

while((C.MatrixSize < A.MatrixSize)||(C.MatrixSize < B.MatrixSize) )

{

C.arr = (Triple\*)realloc(C.arr,sizeof(Triple)\*(C.MatrixSize+INCRESIZE));

C.MatrixSize += INCRESIZE;

}

int p = 0, q = 0, w = 0; //指向p和q的下标指针

while(p < A.Nums || q < B.Nums)

{

//如果有一个跑完了，另一个直接复制过去即可

if(p == A.Nums)

{

while(q < B.Nums)

{

C.arr[w].value = B.arr[p].value;

C.arr[w].i = B.arr[q].i;

C.arr[w].j = B.arr[q].j;

w++;

q++;

C.Nums++;

}

break;

}

if(q == B.Nums)

{

while(p < A.Nums)

{

C.arr[w].value = A.arr[p].value;

C.arr[w].i = A.arr[p].i;

C.arr[w].j = A.arr[p].j;

w++;

p++;

C.Nums++;

}

cout << "YES"<<endl;

break;

}

//如果当前两个下标之指针指向的位置坐标相同，那么数值直接相加

if((A.arr[p] == B.arr[q]) && (A.arr[p].i != -1))

{

C.arr[w].value = A.arr[p].value + B.arr[q].value;

C.arr[w].i = A.arr[p].i;

C.arr[w].j = A.arr[p].j;

// cout << p << " " << q << " " << w <<endl;

// cout << C.arr[w].value<<endl;

p++;

q++;

}

else if(A.arr[p] < B.arr[q])

{

C.arr[w].value = A.arr[p].value;

C.arr[w].i = A.arr[p].i;

C.arr[w].j = A.arr[p].j;

// cout << p << " " << q << " " << w <<endl;

// cout << C.arr[w].value<<endl;

p++;

}

else

{

C.arr[w].value = B.arr[q].value;

C.arr[w].i = B.arr[q].i;

C.arr[w].j = B.arr[q].j;

q++;

}

w ++;

C.Nums ++;

if(C.Nums >= C.MatrixSize)

{

C.arr = (Triple\*)realloc(C.arr,sizeof(Triple)\*(C.MatrixSize+INCRESIZE));

C.MatrixSize += INCRESIZE;

}

}

return true;

}

bool Column\_order\_traversal\_transpose(SeqMatrix &A, SeqMatrix &B)

{

if (A.Nums == 0)

{

return true; //当A中有0个元素，那么直接返回true,即所有元素都为0

}

B.Cols = A.Rows;

B.Rows = A.Cols;

B.Nums = A.Nums;

int q = 0;

for(int i = 1; i <= A.Cols; i++)

{

for (int p = 0; p<A.Nums; p++)

{

if (A.arr[p].j == i)

{

B.arr[q].i = A.arr[p].j;

B.arr[q].j = A.arr[p].i;

B.arr[q].value = A.arr[p].value;

q++;

}

}

}

return true;

}

//列序遍历转置

bool FastTransposeOfMatrix(SeqMatrix &A, SeqMatrix &B)

{

if (A.Nums == 0)

{

CopyMatrix(A,B);

return true;

}

B.Cols = A.Rows;

B.Rows = A.Cols;

B.Nums = A.Nums;

//若为零矩阵，那么直接进行返回

int rowNum[MAXSIZE];

int rowStart[MAXSIZE];

memset(rowNum, 0, MAXSIZE);

memset(rowStart, 0, MAXSIZE);

for (int i = 0; i < A.Nums; i++)

{

rowNum[A.arr[i].j]++;

}

//两个数组的第0个元素均置为0，与算法需求相符合

for (int i = 1; i <= A.Cols; i++)

{

rowStart[i] = rowNum[i-1]+rowStart[i-1];

}

int j = 0, p = 0;

for(int i = 0; i < A.Nums; i++)

{

j = A.arr[i].j; //j为之前应放置的列

p = rowStart[j]; //p记录的是当前列位置的初始位置

B.arr[p].i = A.arr[i].j;

B.arr[p].j = A.arr[i].i;

B.arr[p].value = A.arr[i].value;

rowStart[j]++; // 这一列的初始位置要加一

}

return true;

}

//矩阵的快速转置

void test1()

{

int rows\_1[100] = {

1,1,1,

2,

3,3,3,3,

6,6,

7,7,7,

8,8,

9,9,

10,10,10};

int cols\_1[100] ={

4,8,9,

5,

1,3,4,7,

7,10,

1,2,3,

2,3,

4,5,

2,7,9};

int rows\_2[100] = {

1,

2,2,2,

3,

4,4,4,

5,5,5,5,5,

6,6,

7,7,

9,9,9};

int cols\_2[100] ={

4,

8,9,10,

1,

3,4,7,

1,2,3,7,8,

2,3,

5,6,

2,7,9};

SeqMatrix A,B,C;

InitMatrix(A);

InitMatrix(B);

InitMatrix(C);

CreateMatrix(A,10,10,20,rows\_1,cols\_1);

CreateMatrix(B,10,10,20,rows\_2,cols\_2);

PrintMatrix(A);

PrintMatrix(B);

AddMatrix(A,B,C);

PrintMatrix(C);

// for (int i = 0; i < C.Nums; i++)

// {

// cout << C.arr[i].i << " "<< C.arr[i].j<<" "<<C.arr[i].value<<endl;

// }

// cout << A.Nums << " "<<B.Nums<<" "<<C.Nums<<endl;

return;

}

int main()

{

test1();

int rows\_1[100]={1,1,1,2,3,3,3,3,6,6,7,7,7,8,8,9,9,10,10,10};

int cols\_1[100] ={4,8,9,5,1,3,4,7,7,10,1,2,3,2,3,4,5,2,7,9};

SeqMatrix A,B;

InitMatrix(A);

InitMatrix(B);

CreateMatrix(A,10,10,20,rows\_1,cols\_1);

PrintMatrix(A);

//Column\_order\_traversal\_transpose(A,B);

FastTransposeOfMatrix(A,B);

PrintMatrix(B);

//FastTransposeOfMatrix(B,A);

//PrintMatrix(A);

return 0;

}

2、

题目：求矩阵的马鞍点。（书P69 7）

算法思想：经过证明，一个矩阵中鞍点至多只有一个。所以可以先将每行的最小值存入一个数组，然后对每行最小值相应的列进行遍历，如果这个数是该列的最大值，那么为矩阵鞍点。

运行结果：



结果分析：结果正确，能够正确的找出矩阵的鞍点。

附源程序。

#include <iostream>

#include <fstream>

#include <cmath>

using namespace std;

int M = 10, N = 10;

int Matrix[100][100];

int main()

{

fstream file ("2.txt",ios::in);

file >> M >> N;

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

{

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

{

file >> Matrix[i][j];

}

}

int row\_min[M+5];

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

{

row\_min[i] = 1;

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

{

if(Matrix[i][row\_min[i]] > Matrix[i][j])

{

row\_min[i] = j;

}

}

}

int ans\_i = -1, ans\_j = -1;

bool ans = false;

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

{

int j = 0;

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

{

if (j == i)

{

continue;

}

if(Matrix[i][row\_min[i]] <= Matrix[j][row\_min[i]])

{

break;

}

}

if(j == N + 1) //如果某一列循环到了最后，那么找到了鞍点

{

ans = true;

ans\_i = i;

ans\_j = j-1;

break; //经过证明，鞍点只有一个,那么找到一个就可以停止了

}

}

cout << ans\_i<< " " << ans\_j<<endl;

return 0;

}

3、

题目：题目背景：某地疫情爆发后，出于“应检尽检”的原则，我们想要通知所有近期经过该高危区域的居民参与核酸检测。

问题描述：想要找出经过高危区域的居民，分析位置记录是一种简单有效的方法。

具体来说，一位居民的位置记录包含t个平面坐标(x1,y1),(x2,y2),…,(xt,yt), 其中(xi,yi) 表示该居民i时刻所在位置。

高危区域则可以抽象为一个矩形区域（含边界），左下角和右上角的坐标分别为(xl,yd)和(xr,yu)，满足xl<xr且yd<yu。

考虑某位居民的位置记录，如果其中某个坐标位于矩形内（含边界），则说明该居民经过高危区域；进一步地，如果其中连续k个或更多坐标均位于矩形内（含边界），则认为该居民曾在高危区域逗留。需要注意的是，判定经过和逗留时我们只关心位置记录中的t个坐标，而无需考虑该居民在i到i+1时刻之间位于何处。

给定高危区域的范围和n位居民过去t个时刻的位置记录，试统计其中经过高危区域的人数和曾在高危区域逗留的人数。

算法思想：根据输入进行判断，如果处于危险期，那么经过置为true，并对连续的经过进行记录，遍历完一个人的所有点的时候，如果最长的连续点是大于k的，那么逗留标识置为true。

运行结果：

结果分析：运行结果正确，通过csp模拟检测。

附源程序。

#include <iostream>

#include <fstream>

using namespace std;

int n,k,t,xl,yd,xr,yu;

int x = 0, y = 0;

int main()

{

ifstream file("3.txt", ios::in);

file >> n >> k >> t;

file >> xl >> yd >> xr >> yu;

int passby = 0, stay = 0, time = 0;

bool in = false;

bool candle = false;

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

{

in = false;

candle = false;

time = 0;

for (int j = 1; j <= t; j++)

{

file >> x >> y;

if(x >= xl && x <= xr && y >= yd && y <= yu)

{

in = true;

time++;

}

else

{

time = 0;

}

if(time >= k)

{

candle = true;

}

}

if(time >= k)

{

candle = true;

}

if(candle)

{

stay++;

}

if(in)

{

passby ++;

}

}

cout << passby <<endl;

cout << stay;

return 0;

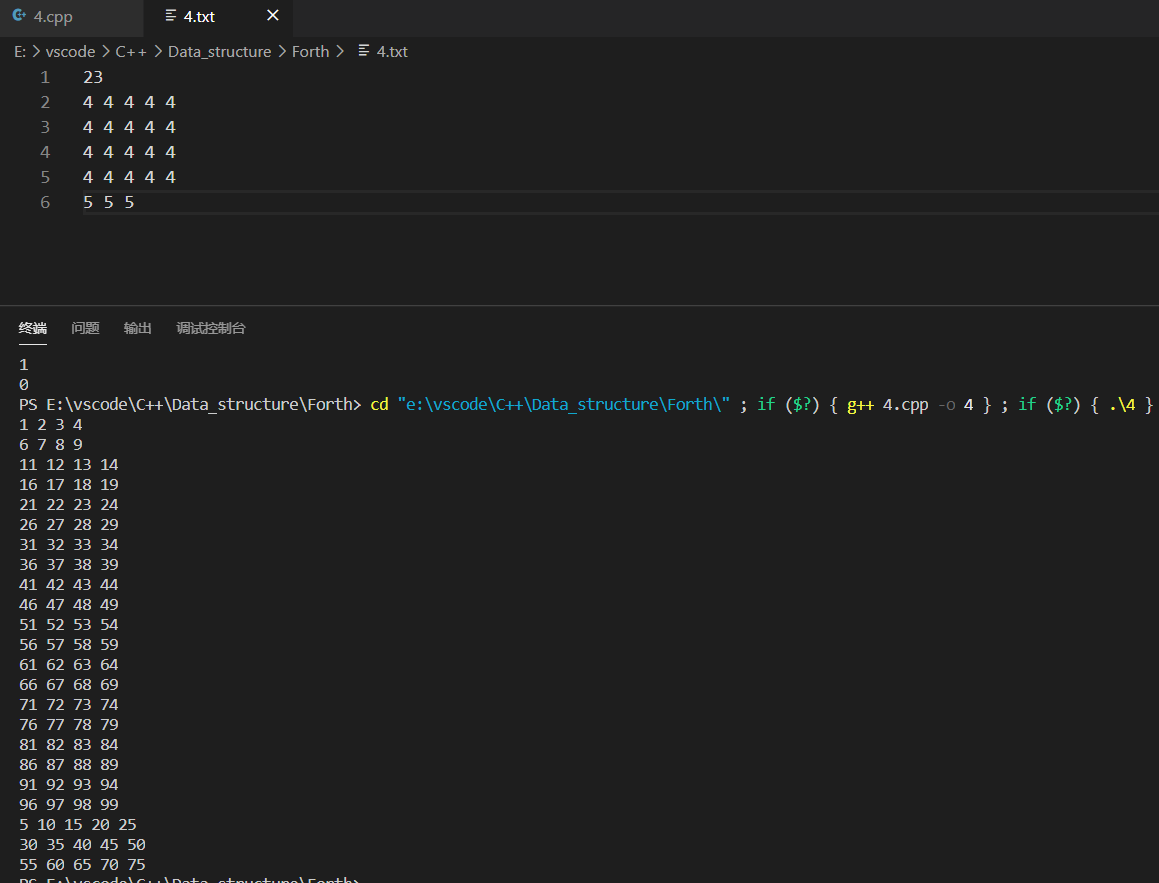
}

4、

题目：问题描述：请实现一个铁路购票系统的简单座位分配算法，来处理一节车厢的座位分配。

算法思想：开设两个数组，一个用来标记100个座位是否做了人，另一个数组大小为20，表示1-20标号每排座位上还剩几个座位，当接收到一个购票请求时，先查询一个符合连续的最小排号进行座位安排，如果所有的排不符合，那么就从最小号的空位开始填补。

运行结果：



结果分析：运行结果正确，通过csp模拟测试检测。

附源程序。

#include <iostream>

#include <fstream>

using namespace std;

bool seat[105]; //表示这个座位是否有人

int series[25]; //表示这个排号有几个连续的座位

int n;

int main()

{

fstream file("4.txt",ios::in);

file >> n;

for(int i = 0; i <= 20; i++)

{

series[i] = 5;

}

int p = 0;

bool flag = false;

while (n--)

{

file >> p;

flag = false;

for (int i = 0; i <20; i++)

{

if(series[i] >= p)

{

flag = true;

register int j=1;

while(seat[i\*5+j])

{

j++;

}

series[i] -= p;

int t = 0;

while( t < p)

{

seat[i\*5 + j + t] = true;

cout << i\*5 + j + t << " ";

//cout << "i= "<<i<<" j= "<<j<<" t = "<<t<<endl;

t++;

}

cout << endl;

break;

}

}

if (!flag)

{

register int t = p;

for (int i = 1; i <= 100; i++)

{

if(!seat[i])

{

seat[i] = true;

cout <<i<< " ";

t --;

}

if(t == 0)

{

break;

}

}

cout << endl;

}

}

return 0;

}

5、

题目：假设以二维数组g(1...m,1...n),表示一个图像区域，g[i,j]表示该区域中点(i,j)

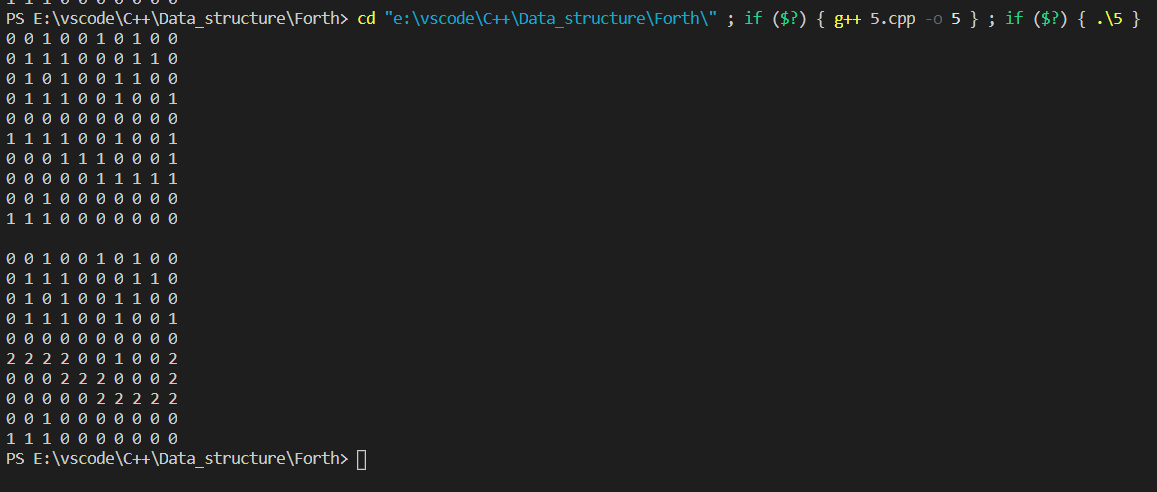
所具有的颜色，其值为从0到k的整数。编写算法置换点(i0,j0)所在区域的颜色。约定和

(i0,j0)同色的上下左右的邻接点为同色区域的点。

算法思想:题目要求对一个点所在区域所有同色的点进行染色，并给出了邻接点的定义，所以我们只需要进行递归。实现了以下两种解法:

1. 利用递归，将当前点染色，递归邻接点。
2. 利用栈(或者队列)，将第一个点染色，并将点入栈，弹出栈顶元素，将弹出元素周围的同色邻接点染色并入栈，直到栈为空。

运行结果：



结果分析：运行结果均正确。

附源程序。

/\*假设以二维数组g(1...m,1...n),表示一个图像区域，g[i,j]表示该区域中点(i,j)

所具有的颜色，其值为从0到k的整数。编写算法置换点(i0,j0)所在区域的颜色。约定和

(i0,j0)同色的上下左右的邻接点为同色区域的点。\*/

#include <iostream>

#include <fstream>

typedef struct Point

{

int x;

int y;

}Point;

#define ElemType Point

#include "Stack.h"

using namespace std;

const int M = 10;

const int N = 10;

char map[M+5][N+5];

void print(ElemType e)

{

cout << e.x <<" "<< e.y <<endl;

}

void readMap( )

{

fstream file("5.txt",ios::in);

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

{

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

{

file >> map[i][j];

}

}

file.close();

return;

}

void printMap()

{

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

{

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

{

cout << map[i][j] << " ";

}

cout << endl;

}

}

void ChangeColorWithRecursion(char toColor, int x, int y)

{

cout << x << " "<< y << endl;

if(x < 1 || x > M || y < 1 || y > N) //在查找的范围之外了

{

return ;

}

char color = map[x][y];

if(map[x][y] == toColor)

{

return;

}

map[x][y] = toColor;

if((x+1 <= M) && (map[x+1][y] == color))

{

ChangeColorWithRecursion(toColor, x+1, y);

}

if((x-1 >= 1) && map[x-1][y] == color)

{

ChangeColorWithRecursion(toColor, x-1, y);

}

if((y+1<=N) && map[x][y+1] == color)

{

ChangeColorWithRecursion(toColor, x, y+1);

}

if((y-1>=1) && map[x][y-1] == color)

{

ChangeColorWithRecursion(toColor, x, y-1);

}

cout << "now x = "<< x <<" and y = "<< y<<endl<<endl;

printMap();

cout << endl<<endl;

return;

}

void ChangeColorWithStack(char toColor, int x, int y)

{

Stack S;

InitStack(S);

char oldColor = map[x][y];

Point p,q;

p.x = x;

p.y = y;

map[x][y] = toColor; //先染色，后压栈

Push(S,p);

while (!StackEmpty(S))

{

Pop(S,q);

//print(q);

//printMap();

if((q.x - 1 >= 1)&&map[q.x - 1][q.y] == oldColor)

{

map[q.x - 1][q.y] = toColor;

p.x = q.x - 1;

p.y = q.y;

Push(S,p);

}

if((q.x + 1 <= M) && map[q.x + 1][q.y] == oldColor)

{

map[q.x + 1][q.y] = toColor;

p.x = q.x + 1;

p.y = q.y;

Push(S,p);

}

if((q.y - 1 >= 1)&&map[q.x][q.y - 1] == oldColor)

{

p.x = q.x;

p.y = q.y - 1;

map[q.x][q.y - 1] = toColor;

Push(S,p);

}

if((q.y + 1 <= M)&&map[q.x][q.y + 1] == oldColor)

{

p.x = q.x;

p.y = q.y + 1;

map[q.x][q.y + 1] = toColor;

Push(S,p);

}

}

return ;

}

int main ()

{

readMap();

printMap();

//ChangeColorWithRecursion('2',1,3);

cout <<endl;

ChangeColorWithStack('2',6,10);

printMap();

return 0;

}

......

二、未调试成功程序及说明

1、

题目：

算法思想：

错误原因：

附源程序。

2、

题目：

算法思想：

错误原因：

附源程序。

......

三、代码行数及小结

