

《数据结构》上机报告

2019 年 10 月 22 日

姓名：马家昱 学号：1950509 班级：计科 1 班 得分：200

实验题目	队列	
问题描述	队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（ front ）进行删除操作，而在表的后端（ rear ）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列的一种特殊情形是循环队列，即将队列首尾相连。	
基本要求	1. （p1）实现循环的基本操作，包括循环队列的创建，入队，出队，判队空，判队满，队列的遍历。 2. （p2）队列的应用：统计矩阵中相邻的非零元素的区域数。 3. 实现一个消息队列。	
	已完成基本内容（序号）：	1, 2, 3
选做要求	1. 程序增添适当的注释，程序的书写采用缩进格式。 2. 程序拥有一定的健壮性，对非法的输入有一定的报错信息。	
	已完成选做内容（序号）	1, 2
数据结构设计	第一题，循环队列的基本操作中，从底层搭建了循环队列，即在内存中开辟了指定大小的、连续的存储单元，当出现“假满”时，将存储空间的首尾相连，从而实现空间利用的最大化。 第二题，队列的应用中，直接使用了 STL 标准模板库中的队列。 消息队列的实现，使用了 STL 标准模板库中的队列。	

1. 队列的基本操作

循环队列类及其初始化

```
class Queue { //循环队列类
public:
    int queueSize; //容量
    int length; //元素个数
    int* front; //指向队头
    int* rear; //指向队尾
    int* spaceFront; //指向分配空间的首地址
    int* spaceRear; //指向分配空间的末地址
public:
    Queue(int size); //初始化函数，容量为参数
    int dequeue(); //出队
    bool enqueue(int elem); //入队
    void printQueue(); //遍历打印
};
```

```
Queue::Queue(int size) { //队列的初始化
    queueSize = size;
    length = 0;
    spaceFront = spaceRear = front = rear = (int*)malloc(sizeof(int) * queueSize);
    for (int i = 0; i < queueSize - 1; i++) //定位指向分配空间末地址的指针
        spaceRear++;
}
```

Bool enqueue (int elem) 入队，当假满时将 front 指针指向存储空间首地址。

```
bool Queue::enqueue(int elem) { //入队
    if (length == queueSize) //队满
        return false;
    *rear = elem;
    if (rear == spaceRear) //如果头指针指向分配空间末地址，则跳转至分配空间的首地址
        rear = spaceFront;
    else rear++; //否则直接向上移动一位
    length++;
    return true;
}
```

Int dequeue () 出队，返回 front 指向的当前元素。当 front 指针指向存储空间末尾时，跳转至其首地址

```
int Queue::dequeue() { //出队
    if (length == 0) //空队
        return -1;
    int result = *front;
    if (front == spaceRear) //如果尾指针指向分配空间末地址，则跳转至分配空间的首地址
        front = spaceFront;
    else front++; //否则直接向上移动一位
    length--;
    return result;
}
```

Void printQueue () 遍历打印

```

void Queue::printQueue() {
    int* cur = front;
    for (int i = 0; i < length; i++) {
        cout << *cur << ' ';
        if (cur == spaceRear) //如果当前指针指向分配空间末地址，则跳转至分配空间的首地址
            cur = spaceFront;
        else cur++;
    }
}

```

2. 队列的应用

Void inputMatrix () 矩阵的输入

```

void inputMatrix() { //矩阵的输入
    cin >> row >> column;
    for (int i = 1; i <= row; i++)
        for (int j = 1; j <= column; j++)
            cin >> matrix[i][j];
}

```

bool isOnBoard(int x, int y) 判断 x、y 处的细胞是否在边界上

```

bool isOnBoard(int x, int y) { //x, y处的细胞是否在边界上
    if (x == 1 || x == row || y == 1 || y == column)
        return true;
    else return false;
}

```

bool isCellInMatrix(int x, int y) 判断 x、y 处是否是矩阵内的细胞

```

bool isCellInMatrix(int x, int y) { //x, y处是否是矩阵内的细胞
    if ((x > 0) && (x <= row) && (y > 0) && (y <= column) && (matrix[x][y]))
        return true;
    else return false;
}

```

Void cellNumber () 遍历矩阵，搜索遇到的第一个细胞

```

void cellNumber() { //遍历矩阵，对遇到的第一个细胞进行搜索
    for (int i = 1; i <= row; i++)
        for (int j = 1; j <= column; j++)
            if (matrix[i][j])
                singleCellSearch(i, j);
}

```

Void singleCellSearch (int x, int y) 完成对连成一片的所有细胞的搜索

```

void singleCellSearch(int x, int y) {
    int boardCellNum = 0; //该细胞在边界上所占据的数目
    int totalCellNum = 0; //该细胞总的所占的数目
    queue<location> Q; //存放坐标的队列
    num++; //细胞数目++
    if (isOnBoard(x, y)) //若在边界, 边界数目++
        boardCellNum++;
    totalCellNum++; //总数目++
    matrix[x][y] = 0; //已搜索到的点置零
    Q.push(location(x, y)); //该位置入队
    do { //对四个方向进行搜索, 若为细胞, 则将其位置入队, 然后将已经搜索过周围的坐标出队
        int newX, newY;
        for (int i = 0; i < 4; i++) {
            newX = Q.front().x + dx[i];
            newY = Q.front().y + dy[i];
            if (isCellInMatrix(newX, newY)) {
                if (isOnBoard(newX, newY)) //若在边界, 边界数目++
                    boardCellNum++;
                totalCellNum++; //总数目++
                Q.push(location(newX, newY));
                matrix[newX][newY] = 0;
            }
        }
        Q.pop();
    } while (!Q.empty()); //队空时停止搜索
    if (boardCellNum == totalCellNum) //如果边界数目等于总数目, 则说明所有细胞均在边界上, num--;
        num--;
}

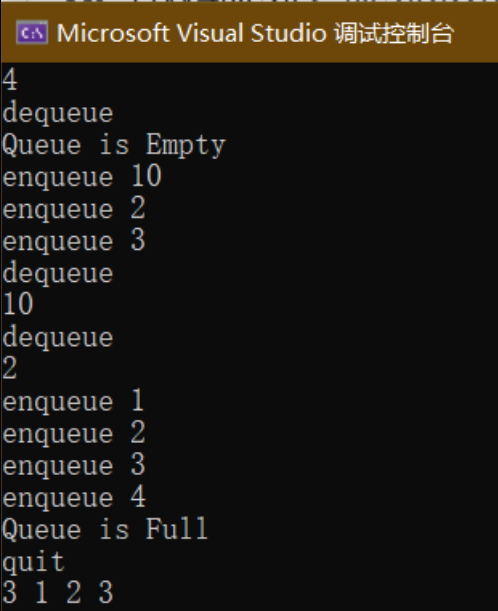
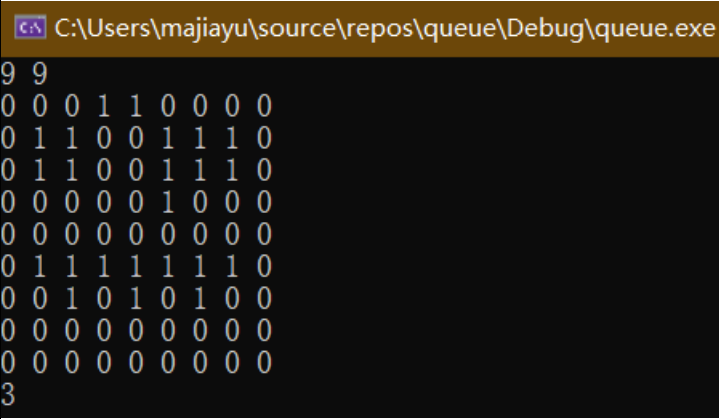
```

3. 消息队列

```

1  #include <iostream>
2  #include <queue>
3  #include <string>
4  using namespace std;
5
6  queue<string> MQ; //总消息队列
7  queue<string> A; //系统A
8  queue<string> B; //系统B
9
10 string str[3000];
11
12 int main() {
13     for (int i = 0; i < 3000; i++) {
14         str[i] = "helloWorld";
15         MQ.push(str[i]);
16     }
17
18     for (; !MQ.empty() && A.size() <= 1000;) {
19         A.push(MQ.front());
20         MQ.pop();
21     }
22
23     for (; !MQ.empty() && B.size() <= 1000;) {
24         B.push(MQ.front());
25         MQ.pop();
26     }
27 }

```

开发环境	Windows 10 Microsoft visual studio 2019
调试分析	<p>(运行结果截图)</p> <p>队列的基本操作：</p>  <p>队列的应用：</p> 
心得体会	<p>(对整个实验过程做出总结，对重要的算法做出性能分析。)</p> <p>第一题队列的基本操作中，循环队列的创建，入队，出队，判队空，判队满操作的时间复杂度均为 $O(1)$，队列的遍历时间复杂度为 $O(n)$。</p> <p>第二题队列的应用中，在对矩阵的每个元素都要进行分析，搜索相邻的格子时，需要对四个方向进行搜索，因此时间复杂度为 $O(n^3)$。</p> <p>通过本次实验，我从底层构建了循环队列的数据结构，并且完成了出队、入队和遍历的基本操作，使我对这种数据结构的底层逻辑更加了解，而不是仅仅停留在会用的层次上。在第二题队列的应用中，我复习了广度优先搜索的基本算法，应用队列完成了实际问题，使我对队列的应用场景有了初步的认识。</p>

