《数据结构与算法实验》第 7 次实验

学院： 专业： 年级：
姓名 学号： 日期： 2022 年 4 月 10 日

第一部分 验证和设计实验

一、 实验目的

1. 掌握链队列的存储结构。
2. 验证链队列的存储结构及基本操作的实现。
3. 验证链队列的操作特性。

二、 实验内容

1. **链队列的实现（带头节点）**：建立一个空队列。
   - 实现加入、删除、取队头元素等操作。
   - 给定测试数据，验证抛出异常机制。
2. **循环队列的实现（不带头节点）**： 建立一个循环队列。
   - 用浪费一个数组空间的方法实现。
   - 用不浪费一个数组空间的方法实现。（以此实验为例写实验报告）
   - 用不带头节点的循环链表实现。
3. **火车车厢重排问题**：参考实验书 p192。

三、 设计与编码

1. 本实验用到的理论知识

- **循环队列的实现：不浪费数组空间**

  实现循环队列其核心是判断队列是空或者满。根据课本 p77 和实验书 p45-46，在不浪费数组空间的条件下实现循环队列有两种方法：

① 引入一个 flag 变量，默认为 0，当有元素入队时 flag 记为 1，有元素出队时 flag 记为 0，此时队满的条件变为 front==rear&&flag==1，队空的条件变为 front==rear&&flag==0。

② 引入一个计数器变量 count 来累计队列的长度，每入队一个元素 count 加 1，出队一个元素 count 减 1，此时队满的条件为 count==QueueSize，队空的条件为 count==0.

本次实验选择使用方法②。

·火车车厢重排问题

一列货运列车共有 n 节车厢，每节车厢将停放在不同的车站。假定 n 个车站的编号分别为 1~n，货运列车按照第 n 站至第 1 站的顺序经过这些车站。车厢编号与他们的目的地一样。为了便于从列车上卸掉相应的车厢，必须重排车厢顺序，使得各车厢从前往后按编号 1 到 n 的次序排列。当所有车厢按照这种次序排列时，在每个车站只需卸掉最后一个车厢即可。我们在一个转轨站里完成重拍的工作，在转轨站有一个入轨，一个出轨和 k 个缓冲轨（位于入轨和出轨之间）。

具体实现：用队列来实现车厢重排，如果待排列车厢 c 为待输出车厢则直接出轨。否则，将其移动到缓冲轨时，应该移动到这样的缓冲轨中：该缓冲轨中现有各车厢的编号均小于 c，如果有多个缓冲轨都满足这一条件，那么选择其中缓冲轨尾车厢编号最大的那个缓冲轨，否则选择一个空的缓冲轨（如果存在的话）。每次有等待排列车厢直接出轨的之后，扫描每个缓冲轨的前端，看看还有没有可以出轨的车厢。

2. 算法设计

定义空队列，设计成员数据头指针和尾指针，成员函数包括构造函数、析构函数、入队函数、出队函数、获取队头函数、判断队列是否为空函数。

3. 编码

(1)链队列的实现-LinkQueue.h

```
#ifndef LinkQueue_H
#define LinkQueue_H
const int Maxsize=100;

template <class T>
```

```
struct Node
{
    T data;
    Node<T> *next;
};
template <class T>
class LinkQueue
{
    public:
        LinkQueue();
        ~LinkQueue();
        void EnQueue(T x);
        T DeQueue();
        T GetQueue();
        int Empty();
    private:
        Node<T> *front,*rear;
};
#endif
```

(1)链队列的实现-LinkQueue.cpp

```
#include <iostream>
#include "LinkQueue.h"
//const int Maxsize=100;

template <class T>
LinkQueue<T>::LinkQueue()              //建立空队
{
    Node<T>*s = new Node<T>;
    s -> next = NULL;
    front = rear = s;
}

template <class T>
LinkQueue<T>::~LinkQueue()             //清除
{
    Node<T>*p = NULL;
    while (front != NULL)
    {
        p = front ->next;
        delete front;
        front = p;
    }
```

```
}

template <class T>
void LinkQueue<T>::EnQueue(T x)          //入队
{
    Node<T>*s = new Node<T>;
    s -> data = x;
    s -> next = NULL;
    rear -> next = s;
    rear = s;
}

template <class T>
T LinkQueue<T>::DeQueue()                //出队
{
    if (rear == front) throw"下溢";
    Node<T> *p = front -> next;
    front -> next = p -> next;
    if (p -> next == NULL) rear = front;
    T x = p -> data;
    delete p;                            //在队列中删除队首元素
    return x;                            //输出出队元素
}

template <class T>
T LinkQueue<T>::GetQueue()               //取队首、不删除
{
    if (rear == front) throw"下溢";
    return front -> next -> data;
}

template <class T>
int LinkQueue<T>::Empty()                //判空函数
{
    if (rear == front) return 1;
    else return 0;
}
```

(1)链队列的实现-LinkQueue_main.cpp

```
#include <iostream>
using namespace std;
#include "LinkQueue.cpp"
```

```
int main()
{
    LinkQueue<int> Q;
    cout<<"Q.Empty = "<<Q.Empty()<<endl;
    cout<<"Q.EnQueue(10),Q.EnQueue(15)"<<endl;
    Q.EnQueue(10);
    Q.EnQueue(15);
    cout<<"Q.GetQueue = "<<Q.GetQueue()<<endl;
    cout<<"Q.DeQueue = "<<Q.DeQueue()<<endl;
    cout<<"Q.GetQueue = "<<Q.GetQueue()<<endl;
    cout<<"Q.Empty = "<<Q.Empty()<<endl;
    return 0;
}
```

(2)循环队列的实现（不带头节点）-CirQueue.h

不浪费一个数组空间的方法（其余见源代码文件夹）

```
#ifndef CirQueue_H
#define CirQueue_H
const int QueueSize=100;

template <class T>
class CirQueue
{
    public:
        CirQueue();
        ~CirQueue(){};
        void EnQueue(T x);
        T DeQueue();
        T GetQueue();
        int Empty();
    private:
        T data[QueueSize];
        int front,rear;
        int count;
};
#endif
```

(2)循环队列的实现（不带头节点）-CirQueue.cpp

不浪费一个数组空间的方法（其余见源代码文件夹）

```
#include <iostream>
#include "CirQueue.h"
```

```
template <class T>
CirQueue<T>::CirQueue()                //建立空队
{
    rear = front = QueueSize-1;
    count = 0;
}

template <class T>
void CirQueue<T>::EnQueue(T x)         //入队
{
    if (count == QueueSize) throw "上溢";
    count++;
    rear = (rear+1)%QueueSize;
    data[rear] = x;
}

template <class T>
T CirQueue<T>::DeQueue()               //出队
{
    if (count == 0) throw"下溢";
    count--;
    front = (front+1)%QueueSize;
    return data[front];
}

template <class T>
T CirQueue<T>::GetQueue()              //取队首、不删除
{
    if (rear == front) throw"下溢";
    int i = (front+1)%QueueSize;
    return data[i];
}

template <class T>
int CirQueue<T>::Empty()               //判空函数
{
    if (rear == front) return 1;
    else return 0;
}
```

(2)循环队列的实现（不带头节点）-CirQueue_main.cpp

不浪费一个数组空间的方法（其余见源代码文件夹）

```cpp
#include <iostream>
using namespace std;
#include "CirQueue.cpp"

int main()
{
    CirQueue<int> Q;
    cout<<"Q.Empty = "<<Q.Empty()<<endl;
    cout<<"Q.EnQueue(10),Q.EnQueue(15)"<<endl;
    Q.EnQueue(10);
    Q.EnQueue(15);
    cout<<"Q.GetQueue = "<<Q.GetQueue()<<endl;
    cout<<"Q.DeQueue = "<<Q.DeQueue()<<endl;
    cout<<"Q.GetQueue = "<<Q.GetQueue()<<endl;
    cout<<"Q.Empty = "<<Q.Empty()<<endl;
    return 0;
}
```

(3)火车重排-LinkQueue.h

```cpp
#ifndef LinkQueue_H
#define LinkQueue_H

template <class T>
struct Node
{
    T data;
    Node<T> *next;
};

template <class T>
class LinkQueue
{
    private:
        Node<T> *front, *rear;
    public:
        LinkQueue();
        ~LinkQueue();
        void EnQueue(T x);
        T DeQueue();
        T GetQueue();
        T GetQueuetail();
        int Empty();
};
```

```
#endif
```

(3)火车重排-LinkQueue.cpp

```
#include <iostream>
#include "LinkQueue.h"
//const int Maxsize=100;

template <class T>
LinkQueue<T>::LinkQueue()                //建立空队
{
    Node<T>*s = new Node<T>;
    s -> next = NULL;
    front = rear = s;
}

template <class T>
LinkQueue<T>::~LinkQueue()               //清除
{
    Node<T>*p = NULL;
    while (front != NULL)
    {
        p = front ->next;
        delete front;
        front = p;
    }
}

template <class T>
void LinkQueue<T>::EnQueue(T x)          //入队
{
    Node<T>*s = new Node<T>;
    s -> data = x;
    s -> next = NULL;
    rear -> next = s;
    rear = s;
}

template <class T>
T LinkQueue<T>::DeQueue()                //出队
{
    if (rear == front) throw"下溢";
    Node<T> *p = front -> next;
```

```
    front -> next = p -> next;
    if (p -> next == NULL) rear = front;
    T x = p -> data;
    delete p;                           //在队列中删除队首元素
    return x;                           //输出出队元素
}

template <class T>
T LinkQueue<T>::GetQueue()              //取队首、不删除
{
    if (front == rear) throw"下溢";
    else return front -> next -> data;
}

template <class T>
T LinkQueue<T>::GetQueuetail()
{
    return rear->data;
}

template <class T>
int LinkQueue<T>::Empty()               //判空函数
{
    if (rear == front) return 1;
    else return 0;
}
```

(3)火车重排-LinkQueue_main.cpp

```
#include <iostream>       //引用输入输出流
using namespace std;
#include "LinkQueue.cpp"    //引入成员函数文件


void chongpai(int n,int k,int a[])
{
    LinkQueue <int> Q[k];
    int out=1;
    for(int i=n;i>=1;i--)
    {
        int num=a[i];
        if(num==out)
        {
            cout<<"出轨：  "<<out<<" 直接出轨  "<<endl;
```

```
            out++;

            int t=1,pos;
            while(t==1)
            {
                t=0;
                for(int i=0;i<k;i++)
                {
                    if(Q[i].Empty()==1) continue;
                    if(Q[i].GetQueue()==out)
                    {
                        cout<<"出轨: "<<out<<" 从 "<<i+1<<" 轨道出 "<<endl;
                        t=1;
                        Q[i].DeQueue();
                        out++;
                        break;
                    }
                }
            }
        }
        else
        {
            int before=-1,po=0;
            for(int i=0;i<k;i++)
            {
                if(Q[i].Empty()==1) continue;
                if(Q[i].GetQueuetail()<num&&Q[i].GetQueuetail()>before)
                {
                    before=Q[i].GetQueuetail();
                    po=i;
                }
            }
            if(before==-1)
            {
                int emp=0,oup;
                for(int i=0;i<k;i++)
                {
                    if(Q[i].Empty()==1)
                    {
                        emp=1;
                        oup=i;
                        break;
                    }
                }
```

```
                if(emp==0)
                {
                    cout<<"重排失败"<<endl;
                    return ;
                }
                else
                {
                    Q[oup].EnQueue(num);
                    cout<<"入轨：  "<<num<<" 进入 "<<oup+1<<" 轨道"<<endl;
                }
            }
            else
            {
                Q[po].EnQueue(num);
                cout<<"入轨：  "<<num<<" 进入 "<<po+1<<" 轨道"<<endl;
            }
        }
    }
}
int main()
{
    int n,k;
    int a[100];
    cin>>n>>k;
    for(int i=1;i<=n;i++)
        cin>>a[i];
    chongpai(n,k,a);
    return 0;
}
```

四、 运行与测试

1. 链队列的实现（不带头节点）

```
Q.Empty = 1
Q.EnQueue(10),Q.EnQueue(15)
Q.GetQueue = 10
Q.DeQueue = 10
Q.GetQueue = 15
Q.Empty = 0

--------------------------------
Process exited after 0.2056 seconds with return value 0
请按任意键继续. . .
```

2. 循环队列的实现

```
Q.Empty = 1
Q.EnQueue(10),Q.EnQueue(15)
Q.GetQueue = 10
Q.DeQueue = 10
Q.GetQueue = 15
Q.Empty = 0

--------------------------------
Process exited after 0.2056 seconds with return value 0
请按任意键继续. . .
```

3. 火车重排

```
9
3
5 8 1 7 4 2 9 6 3
入轨：3 进入 1 轨道
入轨：6 进入 1 轨道
入轨：9 进入 1 轨道
入轨：2 进入 2 轨道
入轨：4 进入 2 轨道
入轨：7 进入 2 轨道
出轨：1 直接出轨
出轨：2 从 2 轨道出
出轨：3 从 1 轨道出
出轨：4 从 2 轨道出
入轨：8 进入 2 轨道
出轨：5 直接出轨
出轨：6 从 1 轨道出
出轨：7 从 2 轨道出
出轨：8 从 2 轨道出
出轨：9 从 1 轨道出

--------------------------------
Process exited after 21.1 seconds with return value 0
请按任意键继续. . .
```

```
5
3
1 2 3 4 5
入轨：5 进入 1 轨道
入轨：4 进入 2 轨道
入轨：3 进入 3 轨道
重排失败

--------------------------------
Process exited after 7.829 seconds with return value 0
请按任意键继续. . .
```

五、 总结与心得

通过这次实验，我学习到了链队列和循环的建立和应用，重点在于循环队列的判空和判满，都是建立在对队空和满之前操作的理解之上的。链队列的设计，依旧是带头节点、带头尾指针的结构更好进行编程的操作。

火车重排列问题是一次很好的队列应用的实例。具体实现的时候，难点在于每当有火车出轨的时候进行每个缓冲轨扫描，来寻找是否有下一个适合出轨的火车。

第二部分 设计实验：迷宫问题

一、问题描述

迷宫求解是实验心理学中的一个经典问题，心理学家把一只老鼠从一个无顶盖的大盒子的入口处赶进迷宫，迷宫中设置很多隔壁，对前进方向形成了多处障碍，假设前进的方向有 8 个，分别是上、下、左、右、左上、左下、右下、右上。

二、基本要求

- 设计数据结构储存迷宫；
- 设计存储结构保存入口到出口的通路；
- 设计算法完成迷宫问题的求解。
- 分析算法的时间复杂度。

三、算法设计

1. 数据结构的设计

可以将迷宫定义成一个二维数组，其中元素值为 1 表示有障碍，元素值为 0 表示没有障碍。为了表示四周的围墙，二维数组四周的数组元素均为 1,, 其中双边矩形表示迷宫,1 代表有障碍,0 代表无障碍，前进的方向有 8 个，分别是上、下、左、右、左上、左下、右下、右上。

因为个点有 8 个试探方向，设当前点的坐标是(x,y)，与其相邻的 8 个点的坐标都可根据与该点的相邻方位而得到。可以采用回溯法实现该问题的求解。回溯法是一种不断试探及时纠正错误的搜索方法。从入口出发，按某一方向向前探索，若能走通（未走过的),即某处可

以到达，则到达新点，否则试探下一方向；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向再继续试探，直到所有可能的通路都搜索到，或找到一条通路，或无路可走又返回到入口点在求解过程中，为了保证在任何位置上都能沿原路退回，需要一个后进先出的栈来保存从入口到当前位置的路径。

为了将递归算法改为非递归算法，需要使用一个栈来存储在试探的过程中所走过的路径。一旦需要回退，可以从栈中取得刚才走过位置的坐标和前进方向。栈中需保存一系列三元组以记录这些信息，这些三元组如图 5 所示 当在迷宫中向前试探时，可能同时存在几个允许的前进方向。我们利用三元组记下当前位置和上一步前进的方向，然后根据 8 个不同方向的选择，选择某一个允许的前进方向前进一步，并将活动记录进栈，以记下前进路径。如果该前进方向走不通，则将位于栈顶的活动记录退栈，以在前进路径上回退一步，再尝试其他的允许方向。如果栈空则表示已经退到开始位置。

2.算法设计

•迷宫和标记数组初始化

a[i][j]=1 表示障碍，0 表示道路。对于边界之外的部分补充障碍 1.t[i][j]=1 表示已经到过此处，0 表示没有到过。

•用栈实现递归

先判断起点(1,1)是否有障碍，如果没有，就将(1,1)压入栈中，并标记(1,1)处已经走过。如果有障碍，则程序失败。之后的递归算法如下：

①当栈不空的时候，取出栈顶部的元素（一个三元组结构，有 x,y,d 三个变量），记为当前达到的位置。

② 开始试探当前位置附近的八个位置是否可以走，选择任意一个新位置试探。若当前位置即为终点，那么直接输出栈中的所有点的坐标即为路线，并结束算法；若新位置不是障碍且没有到达过，则将新位置标记为已到达，再将它的坐标和从当前位置的哪个方位来，压入栈中，并把新位置记为当前位置；否则换下一个位置试探。

③ 栈空则算法结束，迷宫无路可走。

四、代码实现

1. 迷宫-LinkStack.h

```
#ifndef SEQSTACK_H
#define SEQSTACK_H

struct Point
{
    int x;
    int y;
    int d;
};

struct Node
{
    Point p;
    Node *next;
};

class LinkStack
{
    public:
        LinkStack();                //初始化栈
        ~LinkStack();               //清空
        void Push(int x,int y,int d);//将元素 x 入栈
        Point Pop();                //将栈顶元素弹出
        Point GetTop();             //取栈顶元素（并不删除）
        int Empty();                //判断栈是否为空
        friend void migong();
    private:
        Node  *top;                 //栈顶指针，指示栈顶元素在数组中的下标
};
#endif
```

1. 迷宫-LinkStack.cpp

```
#include <iostream>
using namespace std;
#include "LinkStack.h"

LinkStack::LinkStack( )
{
    top=NULL;
}
```

```
LinkStack :: ~LinkStack( )
{
    Node *q;
    while (top != NULL)
    {
        q = top;
        top = top->next;
        delete q;
    }
}

void LinkStack::Push(int x,int y,int d)
{
    Node *s;
    s = new Node;
    s -> p.x = x;
    s -> p.y = y;
    s -> p.d=d;
    s -> next=top;
    top=s;
}

Point LinkStack::Pop( )
{
    Point x;
    Node *tp;
    if (top==NULL) throw "ÏÂÒç";
    x = top -> p;
    tp = top;
    top = top -> next;
    delete tp;
    return x;
}

Point LinkStack::GetTop( )
{
    if (top != NULL)  return top -> p;
    else throw "¿Õ";
}

int LinkStack::Empty( )
{
    if(top == NULL) return 1;
```

```
    else return 0;
}
```

1. 迷宫-LinkStack_main.cpp

```
#include <iostream>          //引用输入输出流
using namespace std;
#include "LinkStack.cpp"       //引入成员函数文件

int deltax(int i)
{
    if(i<=3) return 1;
    if(i<=6) return -1;
    else return 0;
}
int deltay(int i)
{
    if(i%3==1) return 1;
    if(i%3==2) return -1;
    if(i%3==0) return 0;
}

void migong()
{
    int n,m,a[100][100];
    LinkStack Q;
    bool t[100][100];
    cout<<"输入迷宫的行数和列数"<<endl;
    cin>>n>>m;
    cout<<"按行依次输入迷宫，0 代表道路，1 代表障碍，用空格隔开："<<endl;
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=m;j++)
        {
            cin>>a[i][j];
            t[i][j]=0;
        }
    }
    for(int j=0;j<=m+1;j++)
    {
        a[n+1][j]=1;
        a[0][j]=1;
    }
    for(int i=0;i<=n+1;i++)
```

```
    {
        a[i][0]=1;
        a[i][m+1]=1;
    }

    if(a[1][1]==1)
    {
        cout<<"失败"<<endl;
        return ;
    }
    Q.Push(1,1,1);
    t[1][1]=1;
    int x,y,d=1;
    while(Q.Empty()==0)
    {
        Point s=Q.Pop();
        x=s.x,y=s.y,d=s.d;
        while(d<=8)
        {
            int newx = x + deltax(d),newy = y + deltay(d);
            if(x==n && y==m)
            {
                while(Q.Empty()==0)
                {
                    Point po=Q.GetTop();
                    cout<<"("<<po.x<<","<<po.y<<") ->";
                    Q.Pop();
                }
                cout<<"("<<1<<","<<1<<");";
                return ;
            }
            if(t[newx][newy]==0 && a[newx][newy]==0)
            {
                t[newx][newy]=1;
                Q.Push(newx,newy,d);
                x=newx;
                y=newy;
                d=1;
            }
            else
            {
                d++;
            }
        }
```

```
    }
    cout<<"失败"<<endl;
    return ;

}

int main()
{
    migong();
    return 0;
}
```

五、运行测试

测试结果如图。

```
输入迷宫的行数和列数
6 8
按行依次输入迷宫，0代表道路，1代表障碍，用空格隔开：
0 0 1 1 0 1 1 1
1 0 0 0 1 1 1 1
0 1 0 0 0 0 0 1
0 1 1 1 0 1 1 1
1 0 0 1 0 0 0 0
0 1 1 0 0 1 1 0
(6,8) ->(5,7) ->(4,5) ->(3,5) ->(2,4) ->(3,3) ->(2,2) ->(1,1);
--------------------------------
Process exited after 71.56 seconds with return value 0
请按任意键继续. . .
```

六、总结与心得

迷宫问题的核心主要体现在用栈实现递归算法，整个程序的设计较为复杂，需要清晰的思路和恰当的方式进行调试。