**约瑟夫环实验报告**

1. **需求分析**

约瑟夫环（Joseph）问题的一种描述是：编号为1，2，3，…，n的n个人按顺时针方向围坐一圈，每个人持有一个密码（正整数），一开始任选一个正整数作为报数值m，从第一个人开始按顺时针方向自1开始顺序报数，报到m时停止报数，报m的人出列，将他的密码作为新的m值，从他在顺时针方向上的下一个人开始重新从1报数，如此下去，直到所有人出列为止。试设计一个程序求出出列顺序。

1. **概要设计**

约瑟夫环问题仍为线性表的应用，解题过程大致如下：

1. 建立表的存储结构（本次实验采用的是无头结点的双向循环链表存储结构），获取初始m的值
2. 建立m值并根据数值m的大小选择指针移动的方向，查找出列数据元素的位置，删除该数据元素并获取新的m值
3. 重复步骤2直到表空

针对约瑟夫环问题，采用无头节点的双向循环链表进行

报数时只需要用循环遍历对应个数，即可得到对应出队顺序，而后在循环链表中删除该节点并把他的data作为新的m。

当链表为空时，报数结束。

约瑟夫环实验的封装函数：

RecycList(int nums[], int length); //建立双向循环链表

int Node\_delete(int length, int m, Node<T> \*p); //删除链表中的结点

void Node\_move(int length, int m); //合理移动双向循环链表中的指针

Node<T> \*get\_phead(void); //获得链表中的头结点

void clear(); //清空链表

bool empty(); //判空

1. **详细设计**

为高效地描述算法过程，实现线性表抽象数据类型的数据结构（存储结构）选择不带头结点的双向循环链表，如下图所示：

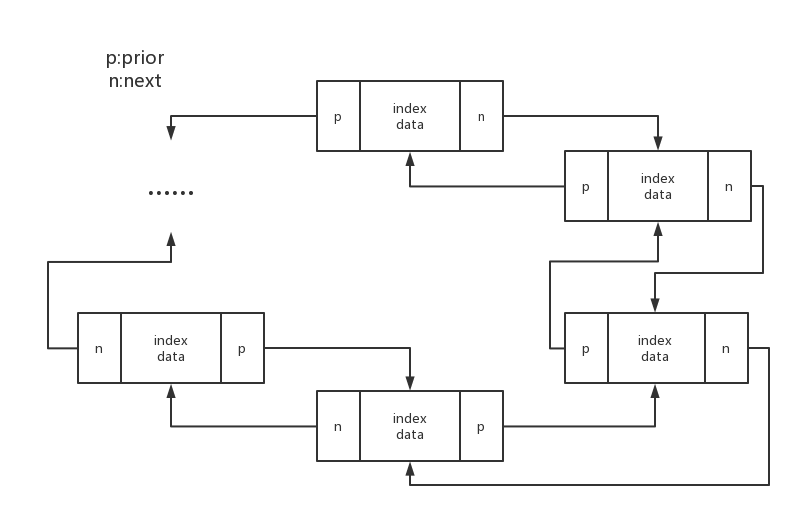


图3.1 无头结点的双向循环链表

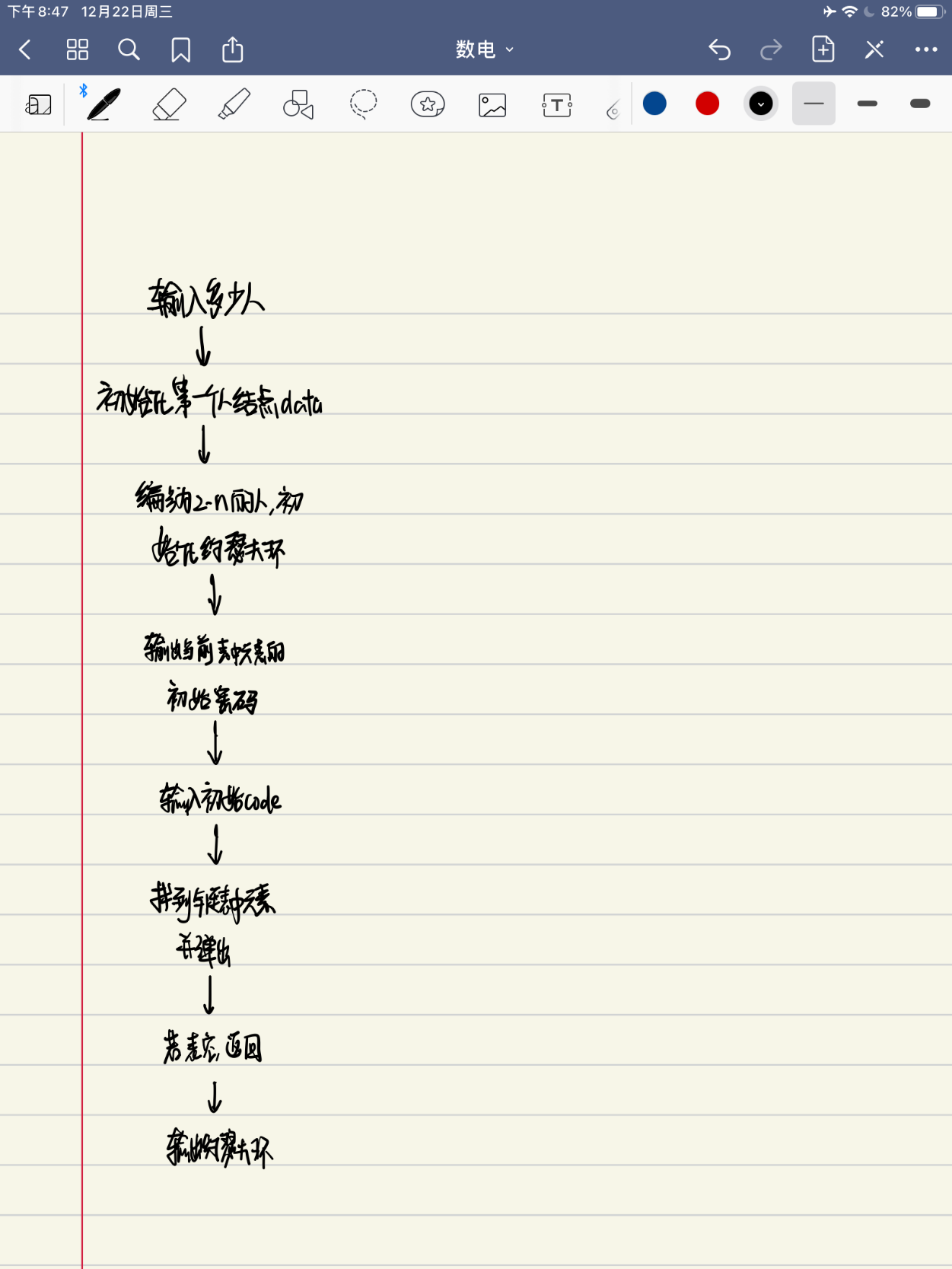


图3.2 约瑟夫环程序实现流程图

用不带头结点的双向循环链表实现的线性表抽象数据类型 **双向循环链表底层结构.cpp** 文档如下：

**1、程序中必要的声明**

#include <iostream>

#include <iomanip>

#include <algorithm>

using namespace std;

template <typename T>

class Reversed\_Basic\_LinkList;

template <typename T>

ostream &operator <<(ostream& out, Reversed\_Basic\_LinkList<T>& rhs);

**2、Reversed\_Basic\_LinkList类的完整构建**

**//构造函数**

Reversed\_Basic\_LinkList(){

phead = new Node(0, 0, NULL);

phead->pre = phead;

phead->next = phead;

}

**//清空链表操作**

void clear()

{

Node \*p = phead;

while (phead->next != phead)

{

p = phead->next;

phead->pre->next = p;

p->pre = phead->pre;

phead = p;

}

delete phead;

phead = NULL;

}

**//析构函数**

virtual ~Reversed\_Basic\_LinkList(){

clear();

}

**//求不带头结点的循环单链表长度**

int Reversed\_Basic\_LinkList\_length()

{

if (!phead)

return 0;

int count = 1;

Node \*p = phead;

while (p->next != p)

{

p = p->next;

count++;

}

return count;

}

**//获取循环链表中第i位元素**

bool GetElem(Reversed\_Basic\_LinkList L, T &e, int i)

{

if ( i < 1 || !L.phead )

return false;

Node \*p = L.phead;

if ( i == 1 )

e = p->data;

else{

int length = Reversed\_Basic\_LinkList\_length();

if (i > length) return false;

else{

while (i > 1){

p = p->next;

i--;

}

e = p->data;

}

}

return true;

}

**//在不带头结点的循环单链表中第i个数据元素之前插入新的数据元素e**

bool ListInsert(Reversed\_Basic\_LinkList &L, int i, T e){

if (i < 1 || !L.phead)

return false;

Node \*p = L.phead;

int length = Reversed\_Basic\_LinkList\_length();

if (i > length) return false;

else{

int j = 0;

if (i < length - i){

while (j < i - 1){

p = p->next;

j++;

}

}else{

while (j < length - i - 1){

p = p->pre;

j++;

}

}

Node \*rub;

rub = new Node(p->pre->number+1, e, NULL);

rub->pre = p->pre;

p->pre->next = rub;

rub->next = p;

p->pre = rub;

}

}

**//在非空循环单链表中删除第i个数据元素**

bool ListDelete(Reversed\_Basic\_LinkList &L, int i, T &e )

{

int length = Reversed\_Basic\_LinkList\_length();

Node \*p;

p = L.phead;

if ( i < 1 || !L.phead )

return false;

if (i == 1 && length == 1){

delete p;

p = NULL;

}else{

if (i > length) return false;

else{

int j = 0;

if (i < length - i){

while (j < i - 1){

p = p->next;

j++;

}

}else{

while (j < length - i - 1){

p = p->pre;

j++;

}

}

// 删除第i个位结点

Node \*rub = p->next;

p->next = rub->next;

rub->next->pre = rub->pre;

delete rub;

}

}

return true;

}

**//从第一个位置起查找与e匹配的数据元素，若存在则返回该数据元素的位置**

int LocateElem(Reversed\_Basic\_LinkList &L, T &e, bool( \*Compare )(T &e1, T &e2))

{

if (! L.head)

return 0;

Node \*p = L.phead;

int cnt = 1;

while (p->next != p && !Compare(p->data, e)){

cnt++;

p->next = p;

}

if (Compare(p->data, e))

return cnt;

else

return 0;

}

**//依序对单链表中的每个数据元素进行遍历，遍历到每个数据元素时调用函数visit()一次且仅一次**

void ListTraverse(Reversed\_Basic\_LinkList &L, void( \*Visit )(T &e))

{

Node \*p = L.head;

if (p){

while (p->next != p){

Visit(p->data);

p = p->next;

}

Visit(p->data);

}

}

**//复制构造函数 深拷贝**

Reversed\_Basic\_LinkList(const Reversed\_Basic\_LinkList <T> &rhs)

{

phead = new Node(0, 0, NULL);

Node \*p = phead;

Node \*q = rhs.phead;

Node \*rub;

while(rhs.phead != NULL)

{

rub = new Node(q->number, q->data, NULL);

p->next = rub;

rub->pre = p;

rub->next = phead;

phead->pre = rub;

p = rub;

q = q->next;

}

rub = new Node(q->number, q->data, NULL);

p->next = rub;

rub->pre = p;

rub->next = phead;

phead->pre = rub;

}

**//移动构造函数 浅拷贝**

Reversed\_Basic\_LinkList(Reversed\_Basic\_LinkList <T> &&rhs) noexcept{

phead = rhs.phead;

rhs.phead = NULL;

}

**//empty 判空**

bool empty()

{

if(phead) return false;

else return true;

}

**//复制赋值运算符**

Reversed\_Basic\_LinkList &operator =(const Reversed\_Basic\_LinkList &rhs)

{

if (this == &rhs) return \*this;

if (phead) delete[]phead;

Node \*p = phead;

Node \*q = rhs.head;

Node \*rub;

while(rhs.phead != NULL)

{

rub = new Node(q->number, q->data, NULL);

p->next = rub;

rub->pre = p;

rub->next = phead;

phead->pre = rub;

p = rub;

q = q->next;

}

rub = new Node(q->number, q->data, NULL);

p->next = rub;

rub->pre = p;

rub->next = phead;

phead->pre = rub;

return \*this;

}

**//移动赋值运算符**

Reversed\_Basic\_LinkList &operator =(Reversed\_Basic\_LinkList <T> &&rhs) noexcept{

Node \*p = this->phead;

this->phead = rhs.phead;

rhs.phead = p;

return \*this;

rhs.clear();

}

**//输出流重载**

friend ostream &operator <<(ostream &os, Reversed\_Basic\_LinkList<T> &rhs)

{

Node \*p = rhs.phead;

Node \*q = p;

while (p->next != p){

os<<"第"<<p->number<<"位玩家出局，其所携带的密码数是"<<p->data;

q = p->next;

delete p;

p = q;

}

os<<"第"<<p->number<<"位玩家出局，其所携带的密码数是"<<p->data;

return os<<endl;

}

**//私有成员**

private:

struct Node{

T number;

T data;

Node \*pre;

Node \*next;

Node(){} //无参构造

Node(Node \*p, Node \*q):pre(p), next(q){}

Node(T x, T y, Node \*p, Node \*q):number(x), data(y), pre(p), next(q){}

};

Node \*phead;

**约瑟夫环实验具体实现代码及其分析如下：**

1. **约瑟夫环实验的封装函数：**

**//定义一个类模板结构体**

template <typename T>

struct Node{

Node(int \_num, int \_data):num(\_num), data(\_data), pre(NULL), next(NULL){}

int num;

T data;

Node<T> \*pre, \*next;

};

**//建立双向循环链表**

RecycList(int nums[], int length){

phead = new Node<T> (0, nums[0]);

if (phead == NULL){

cout<<"There is no empty space.";

return;

}

phead->next = phead;

phead->pre = phead;

Node<T> \*p = phead;

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

Node<T> \*rub = new Node<T>(i, nums[i]);

if (rub == NULL){

cout<<"There is no empty space.";

return;

}

p->next = rub, rub->pre = p;

rub->next = phead, phead->pre = rub;

p = rub;

}

}

**//删除链表中的结点**

int Node\_delete(int length, int m, Node<T> \*p){

m = (p->data - 1) % length;

p->pre->next = p->next;

p->next->pre = p->pre;

delete p;

return m;

}

**//合理移动双向循环链表中的指针**

void Node\_move(int length, int m){

Node<T> \*p;

p = phead;

if (m <= 0) return;

else m = (m - 1) % length;

while (p->next != p){

if (length >= m \* 2){

//判断条件:length - m >= m 指针向后移动,否则向前移动

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

cout<<p->num+1<<endl;

length--;

Node<T> \*q = p->next;

m = Node\_delete(length, m, p);

p = q;

}else{

for (int i = 1; i <= length - m; i++) p = p->pre;

cout<<p->num+1<<endl;

length--;

Node<T> \*q = p->next;

m = Node\_delete(length, m, p);

p = q;

}

}

cout<<p->num+1<<endl;

delete p;

}

**//获得链表中的头结点**

Node<T> \*get\_phead(void){

return phead;

}

**//清空链表**

void clear(){

while (phead->next != phead){

Node<T> \*p = phead->next;

delete phead;

phead = p;

}

}

**//判空**

bool empty(){

return phead == NULL;

}

**//私有成员**

private:

Node<T> \*phead;

1. **Main函数**

int main(){

//玩家个数

int players;

cout<<"请输入玩家个数: ";

cin>>players;

//存放每个玩家的密码数的数组

int code\_nums[players];

cout<<"请按序输入每位玩家的密码数: ";

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

cin>>code\_nums[i];

//cout<<code\_nums[i]<<"\n";

}

RecycList<int> \*L=new RecycList<int> (code\_nums, players);

Node<int> \*p=L->get\_phead(), \*pre, \*next;

int m;

cout<<"请输入游戏中初始的踩雷数字: ";

cin>>m;

cout<<"在约瑟夫环中的玩家淘汰顺序: "<<endl;

L->Node\_move(players, m);

return 0;

}

1. **编程要点及调试分析**
2. 在建表过程中，插入的第一个结点与插入其他结点有所不同
3. 为准确将指针定位在特定结点的前驱位置，设置了一个辅助移动指针，与原链表中phead指针进行配合，进行删除结点的操作。
4. 为过滤不必要的循环次数，使用模运算，以确保m为任意值时，指针的下一个位置定位都能在链表的一次循环内完成。具体实现：先对m的值进行处理：m = (m - 1) % length, 提高程序运行效率；再根据处理后m的大小决定指针移动的方向：当m大于剩余人数的一半时，向前遍历length - m个节点，反之向后遍历m个节点。
5. 由于使用了模运算，算法的复杂度是O（n^2）
6. **测试数据以及用户手册**

程序经Dev C++及Visual Studio Code等编译器编译，运行环境为Windows操作系统，进入程序运行后即交互显示文本方式的用户界面，用户使用过程可参照提示进行。用户手册略。

在进行测试时，由于输入的玩家人数以及对应的密码数是随机的，所以将使用两组测试数据进行测试。

执行约瑟夫环.cpp文件，代入测试数据后，程序的运行结果如下：

**测试1：**

请输入玩家个数：5

游戏中初始的踩雷数字：2

每个玩家各自对应的密码数：各自玩家在数组中的序号加10

1

4

2

0

3

**测试2：**

请输入玩家个数：10

游戏中初始的踩雷数字：7

每个玩家各自对应的密码数：各自玩家在数组中的序号加10

6

3

9

5

0

8

2

1

4

7

经检验，测试结果正确。