1.双向循环链表

1.1双向循环链表的特点

双向链表的头和尾节点的pre,next指向NULL，如果一个指针通过next访问到了

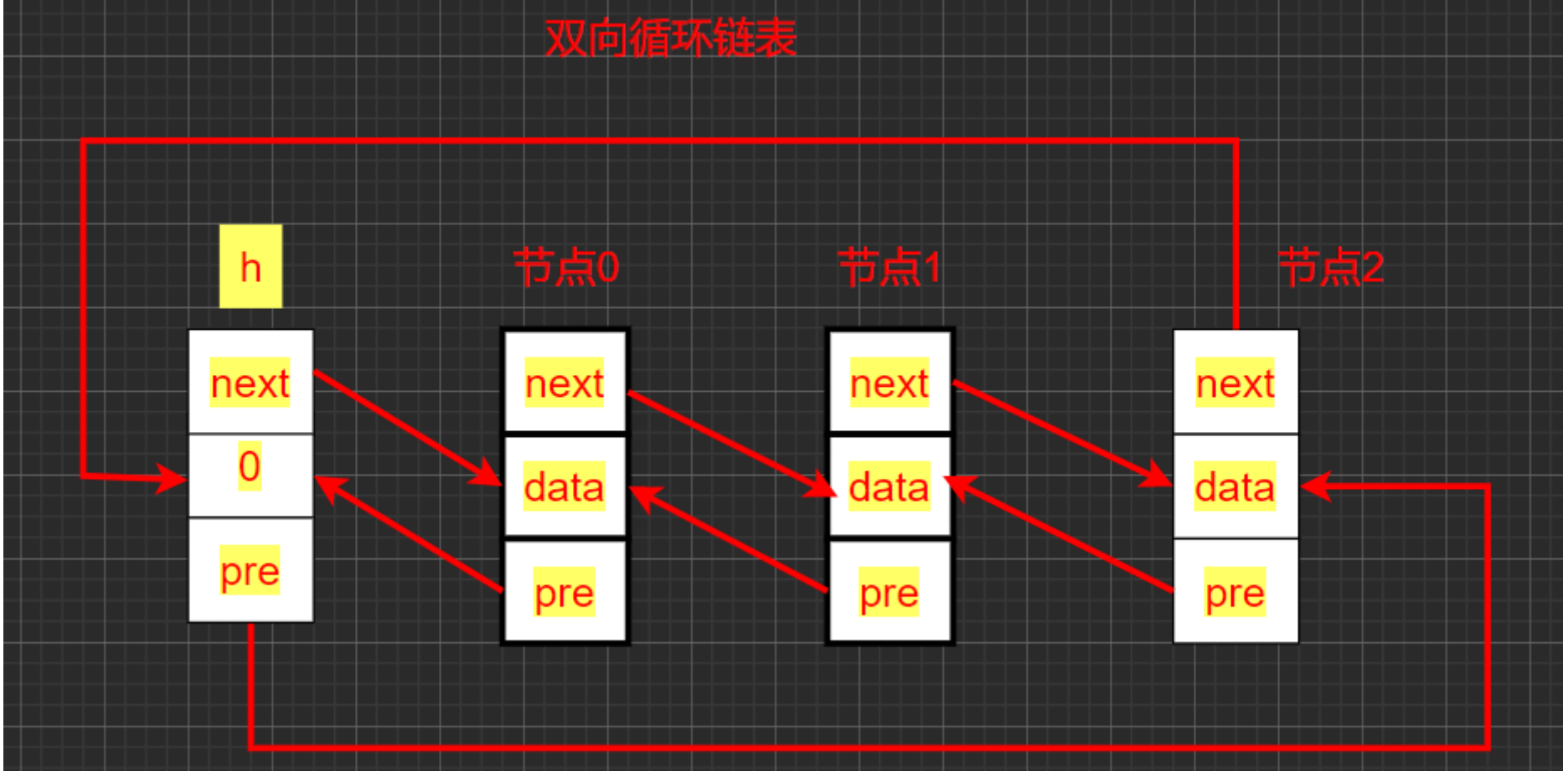
尾结点，它想再次回到头节点就必须通过pre将整条链表走一遍，但是双向循环

循环链表就可以解决这种问题，因为双向循环链表的尾节点next指向的头节点，

头节点的pre指向的是尾节点。

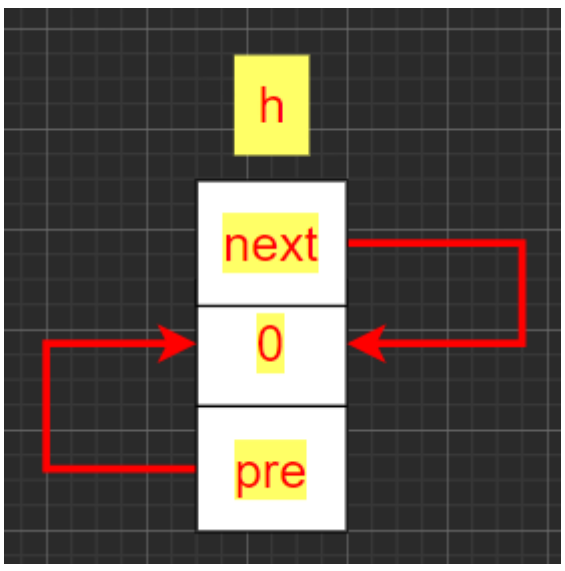
1.2双向循环链表的结构

```c
#define datatype int
typedef struct node{
    datatype data;
    struct node *pre,*next;
}DPloop_t;
```



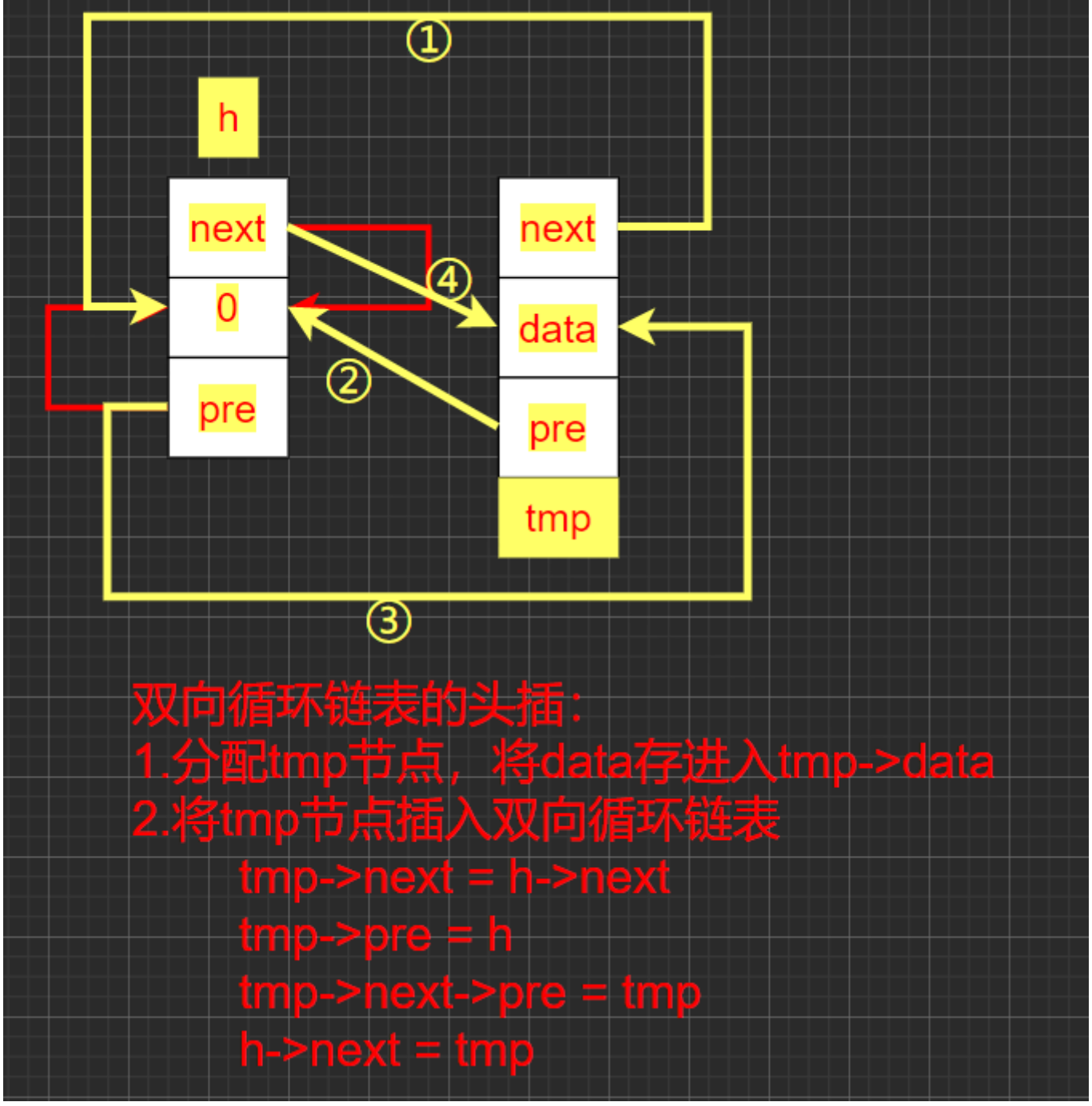
1.3双向循环链表的常见操作

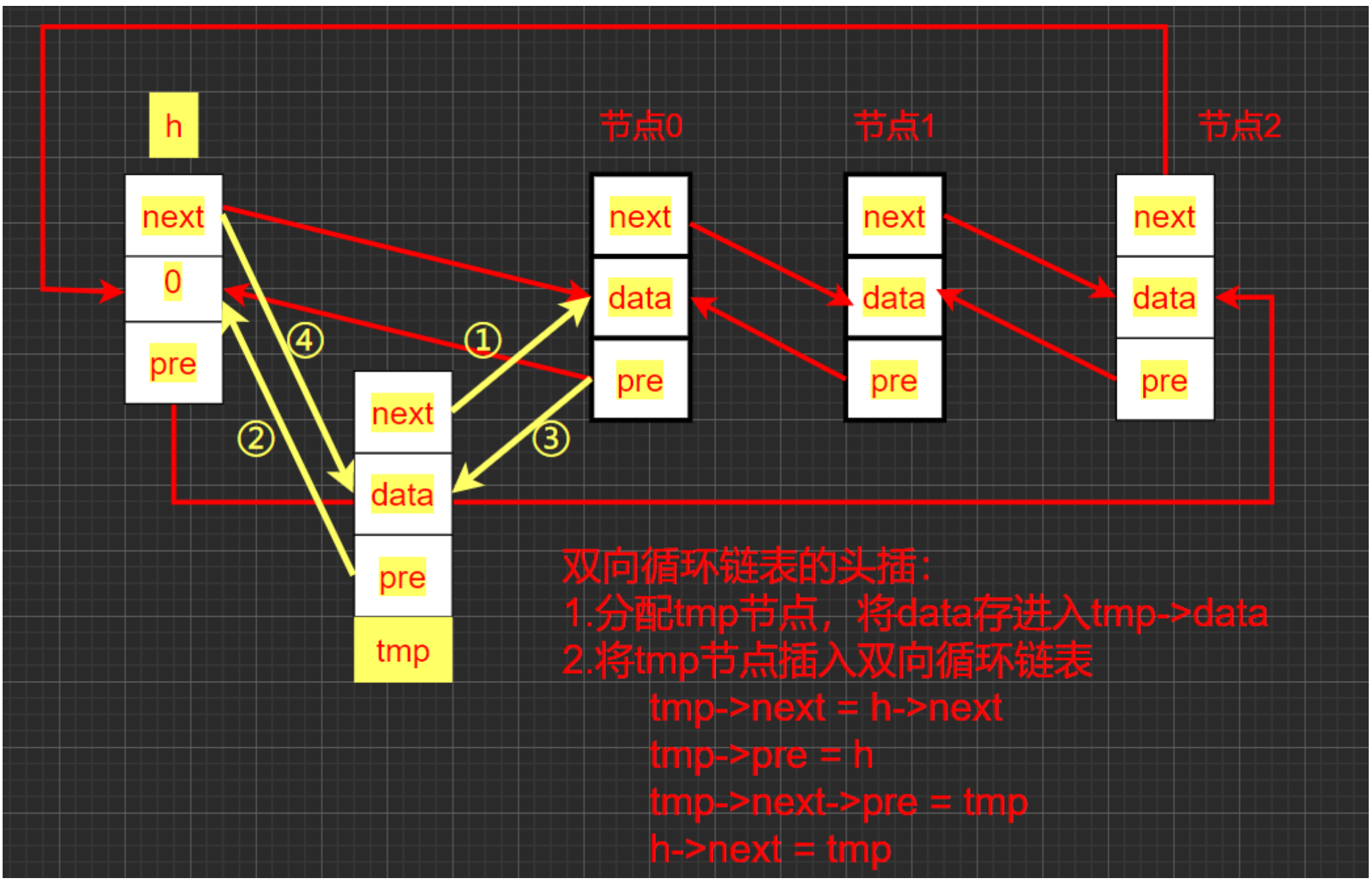
1.3.1双向循环链表的创建




```c
DPloop_t* DPLoopCreate(void)
{
    DPloop_t *h;
    h = (DPloop_t *)malloc(sizeof(*h));
    if(h == NULL){
        printf("%s malloc memory error\n",__func__);
        return NULL;
    }

    h->data = (datatype)0;
    h->next = h->pre = h;

    return h;
}
```

1.3.2双向循环链表的头插法





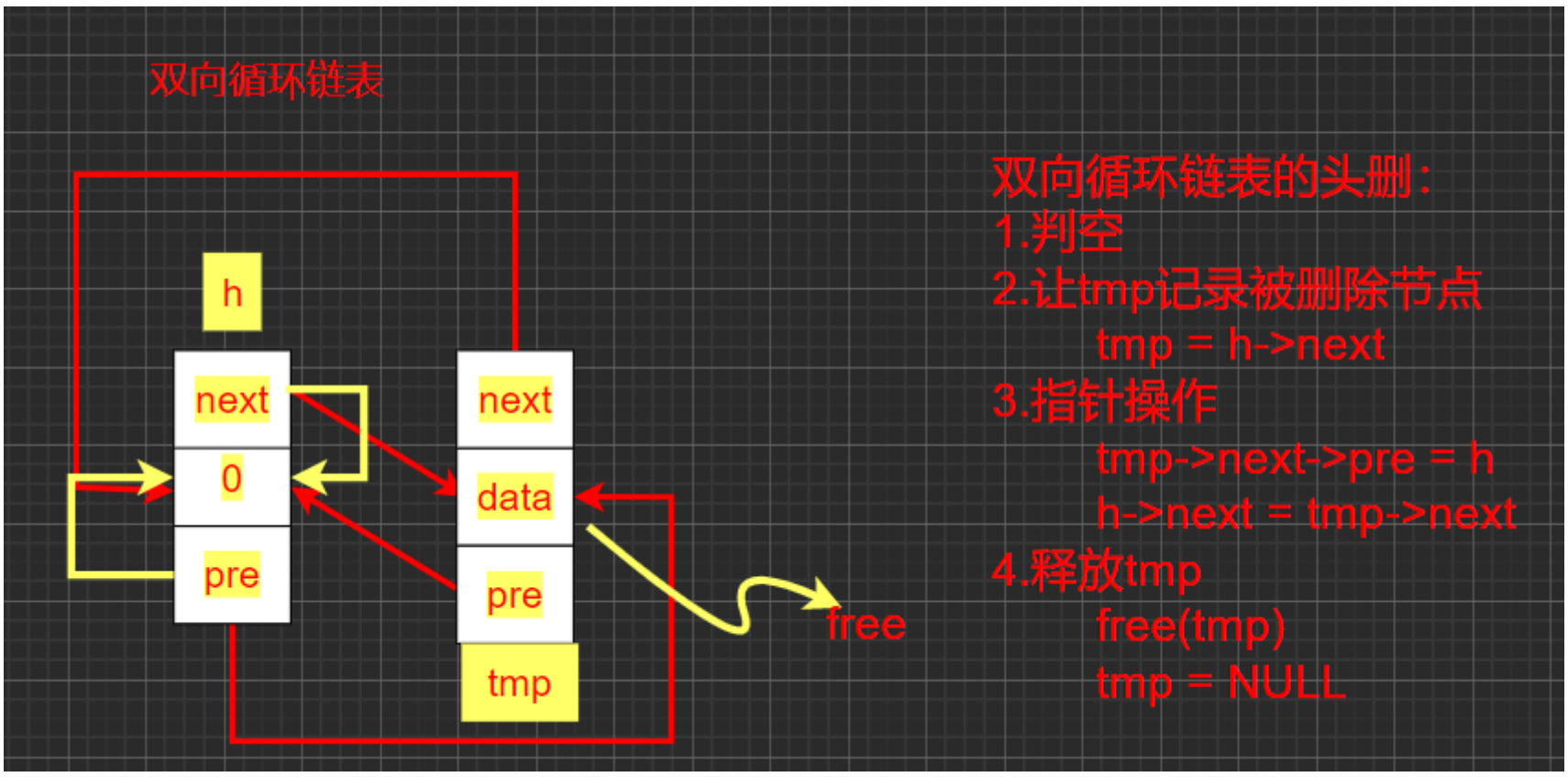

```c
int DPLoopInsertHead(DPloop_t* h, datatype data)
{
    DPloop_t* tmp;
    tmp = (DPloop_t*)malloc(sizeof(*tmp));
    if (tmp == NULL) {
        printf("%s malloc memory error\n", __func__);
        return -1;
    }
    tmp->data = data;

```

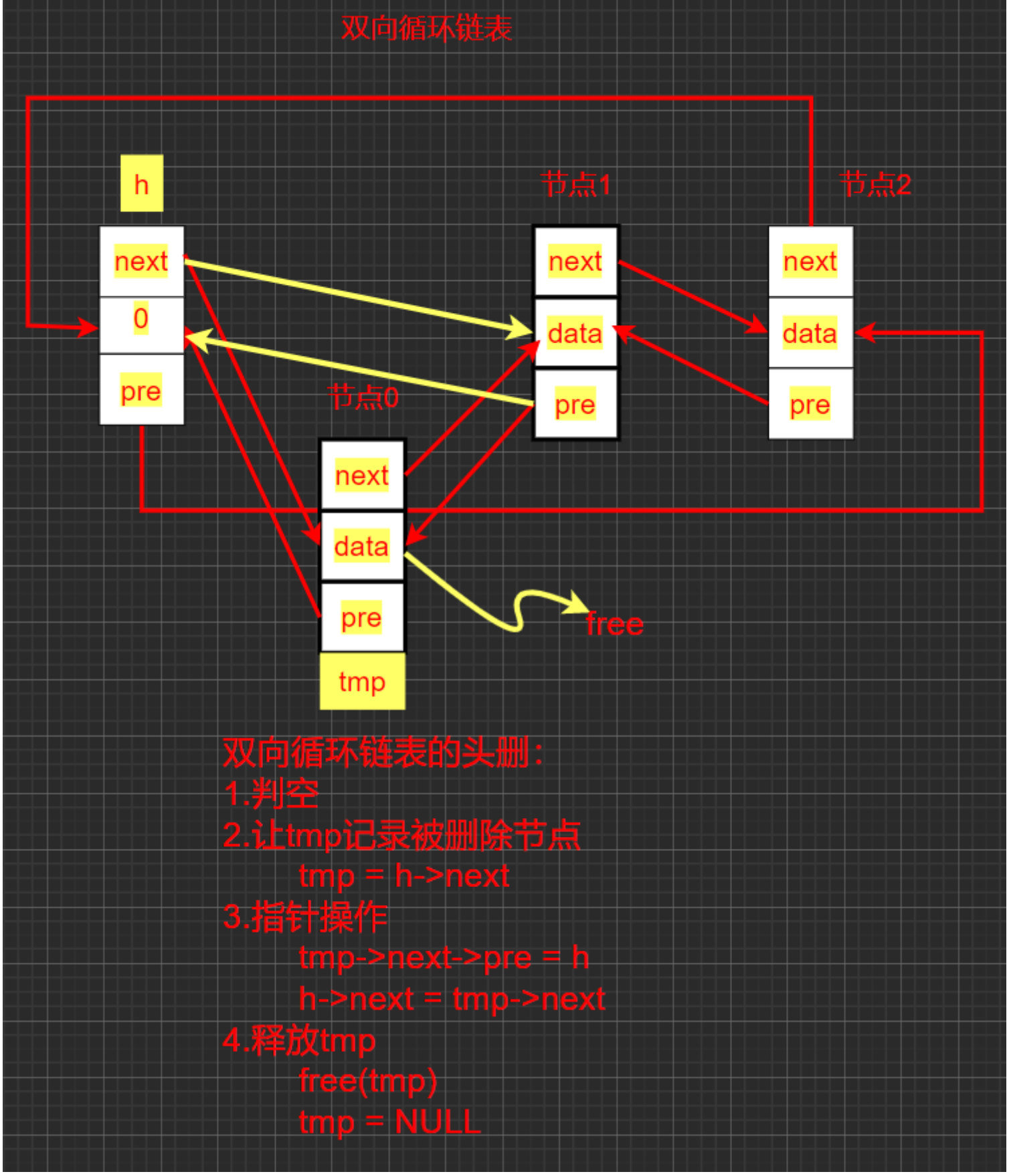
```
    tmp->next = h->next;
    tmp->pre = h;
    tmp->next->pre = tmp;
    h->next = tmp;

    return 0;
}
```

1.3.3双向循环链表判空

```
int DPLoopIsEmpty(DPloop_t* h)
{
    return (h->next == h) ? 1 : 0;
}
```

1.3.4双向循环链表的头删法





1.3.5双向循环链表的遍历

```c
void DPLoopShow(DPloop_t* h)
{
    DPloop_t *th = h;
    while(h->next != th){
        printf("-%d",h->next->data);
        h = h->next;
    }
    printf("-\n");

    th = h;
    while(h->pre != th){
        printf("-%d",h->data);
        h = h->pre;
    }
    printf("-\n");
}
```

1.3.6双向循环链表的查询

```c
datatype DPLoopCheckByPos(DPloop_t* h, int pos)
{
    DPloop_t* th = h;
    if (pos < 0) {
        printf("%s pos left error\n", __func__);
        return (datatype)-1;
    }
    while (h->next != th) {
        if (pos != 0) {
            h = h->next;
            pos--;
        } else {
            // 找到查询的位置
```

```
            return h->next->data;
        }
    }
    printf("%s pos right error\n", __func__);
    return (datatype)-1;
}
```

1.3.7双向循环链表的更新

```
int DPLoopUpdateByPos(DPloop_t* h, int pos, datatype data)
{
    DPloop_t* th = h;
    if (pos < 0) {
        printf("%s pos left error\n", __func__);
        return -1;
    }
    while (h->next != th) {
        if (pos != 0) {
            h = h->next;
            pos--;
        } else {
            // 找到查询的位置
            h->next->data = data;
            return 0;
        }
    }
    printf("%s pos right error\n", __func__);
    return -1;
}
```

1.4双向循环链表的整体代码

DPloop.h

```
#ifndef __DPLOOP_H__
#define __DPLOOP_H__

#include <stdio.h>
#include <stdlib.h>

#define datatype int
typedef struct node {
    datatype data;
    struct node *pre, *next;
} DPloop_t;

DPloop_t* DPLoopCreate(void);
int DPLoopInsertHead(DPloop_t* h, datatype data);
int DPLoopInsertByPos(DPloop_t* h,int pos, datatype data);
int DPLoopIsEmpty(DPloop_t *h);
datatype DPLoopDeleteHead(DPloop_t* h);
datatype DPLoopCheckByPos(DPloop_t* h, int pos);
int DPLoopUpdateByPos(DPloop_t* h, int pos, datatype data);
void DPLoopShow(DPloop_t* h);

#endif
```

DPloop.c

```
#include "DPloop.h"
DPloop_t* DPLoopCreate(void)
{
    DPloop_t* h;
    h = (DPloop_t*)malloc(sizeof(*h));
    if (h == NULL) {
        printf("%s malloc memory error\n", __func__);
        return NULL;
    }

    h->data = (datatype)0;
    h->next = h->pre = h;

    return h;
}
int DPLoopInsertHead(DPloop_t* h, datatype data)
{
    DPloop_t* tmp;
    tmp = (DPloop_t*)malloc(sizeof(*tmp));
    if (tmp == NULL) {
        printf("%s malloc memory error\n", __func__);
        return -1;
```

```
    }
    tmp->data = data;

    tmp->next = h->next;
    tmp->pre = h;
    tmp->next->pre = tmp;
    h->next = tmp;

    return 0;
}
int DPLoopInsertByPos(DPloop_t* h, int pos, datatype data)
{
    DPloop_t* th = h;
    // 1.判断位置（左侧）合法性
    if (pos < 0) {
        printf("%s pos left error\n", __func__);
        return -1;
    }
    // 2.为了让循环多走一次，加上了pos==0
    while (h->next != th || pos == 0) {
        if (pos != 0) {
            h = h->next;
            pos--;
        } else {
            DPloop_t* tmp;
            // 3.分配tmp节点，将数据存入
            tmp = (DPloop_t*)malloc(sizeof(*tmp));
            if (tmp == NULL) {
                printf("%s malloc memory error\n", __func__);
                return -1;
            }
            tmp->data = data;
            // 4.将tmp节点插入双向循环链表中
            tmp->next = h->next;
            tmp->pre = h;
            tmp->next->pre = tmp;
            h->next = tmp;
            return 0;
        }
    }

    printf("%s pos right error\n", __func__);
    return -1;
}
int DPLoopIsEmpty(DPloop_t* h)
{
    return (h->next == h) ? 1 : 0;
}
datatype DPLoopDeleteHead(DPloop_t* h)
{
    DPloop_t* tmp;
    datatype data;

    if (DPLoopIsEmpty(h)) {
        printf("%s is empty error\n", __func__);
        return (datatype)-1;
    }

    tmp = h->next;
    tmp->next->pre = h;
    h->next = tmp->next;

    data = tmp->data;
    if (tmp != NULL) {
        free(tmp);
        tmp = NULL;
    }
    return data;
}
datatype DPLoopCheckByPos(DPloop_t* h, int pos)
{
    DPloop_t* th = h;
    if (pos < 0) {
        printf("%s pos left error\n", __func__);
        return (datatype)-1;
    }
    while (h->next != th) {
        if (pos != 0) {
            h = h->next;
            pos--;
        } else {
            // 找到查询的位置
            return h->next->data;
        }
```

```
    }
    printf("%s pos right error\n", __func__);
    return (datatype)-1;
}
int DPLoopUpdateByPos(DPloop_t* h, int pos, datatype data)
{
    DPloop_t* th = h;
    if (pos < 0) {
        printf("%s pos left error\n", __func__);
        return -1;
    }
    while (h->next != th) {
        if (pos != 0) {
            h = h->next;
            pos--;
        } else {
            // 找到查询的位置
            h->next->data = data;
            return 0;
        }
    }
    printf("%s pos right error\n", __func__);
    return -1;
}
void DPLoopShow(DPloop_t* h)
{
    DPloop_t* th = h;
    while (h->next != th) {
        printf("-%d", h->next->data);
        h = h->next;
    }
    printf("-\n");

    th = h;
    while (h->pre != th) {
        printf("-%d", h->data);
        h = h->pre;
    }
    printf("-\n");
}
```

main.c

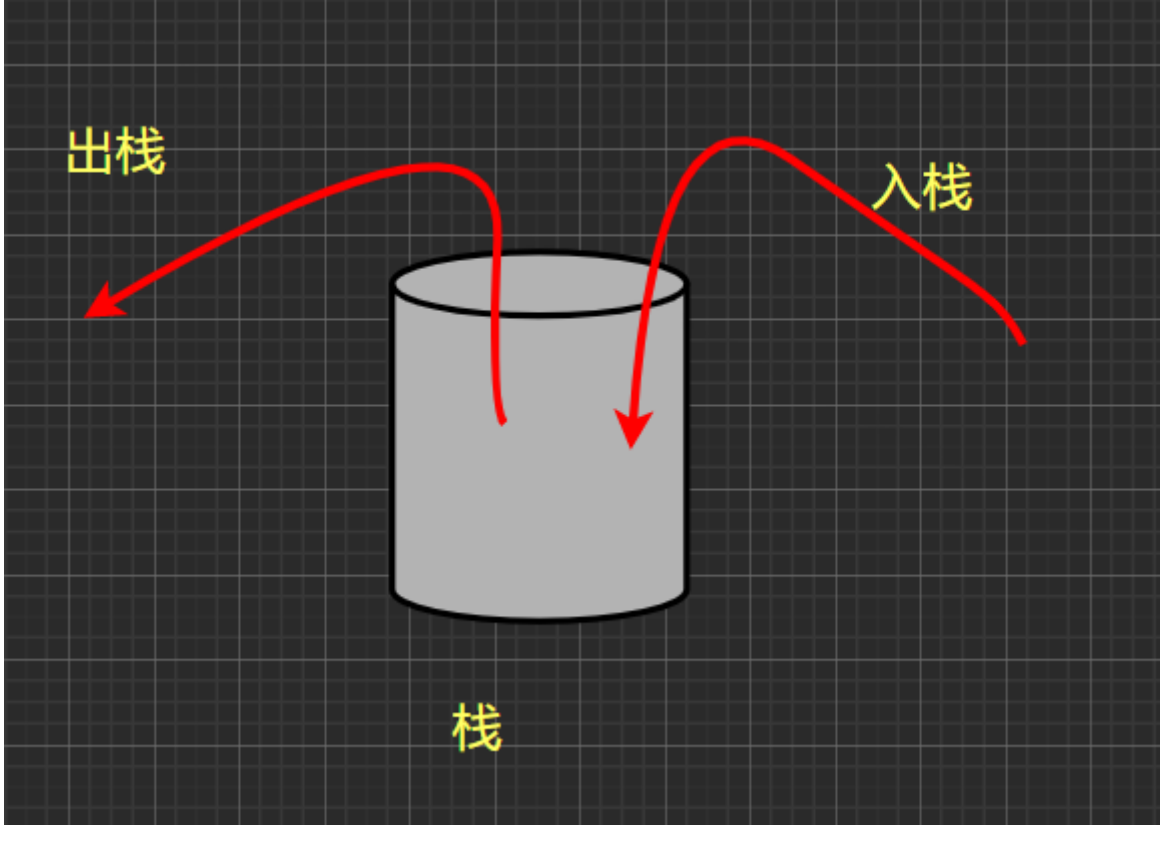
```
#include "DPloop.h"

int main(int argc, const char* argv[])
{
    DPloop_t* h;
    h = DPLoopCreate();
    if (h == NULL)
        return -1;

    // DPLoopInsertHead(h, 44);
    // DPLoopInsertHead(h, 33);
    // DPLoopInsertHead(h, 22);
    // DPLoopInsertHead(h, 11);
    // DPLoopShow(h);

    // DPLoopDeleteHead(h);
    // DPLoopShow(h);

    DPLoopInsertByPos(h,0,11);
    DPLoopInsertByPos(h,1,22);
    DPLoopInsertByPos(h,2,33);
    DPLoopInsertByPos(h,3,44);
    DPLoopShow(h);
    printf("check data = %d\n", DPLoopCheckByPos(h, 2));
    DPLoopUpdateByPos(h, 2, 6666);
    printf("check data = %d\n", DPLoopCheckByPos(h, 2));
    DPLoopShow(h);
    return 0;
}
```

2.栈

2.1栈的特点

栈：它是一种后进先出的数据结构（LIFO）,栈有栈顶和栈底，

栈底是不能够操作的，所有的操作都是通过栈顶完成。



2.2栈的种类

常用的栈有顺序栈和链式栈两种。

顺序栈：它是通过数组实现的。

链式栈：它是通过链表实现的

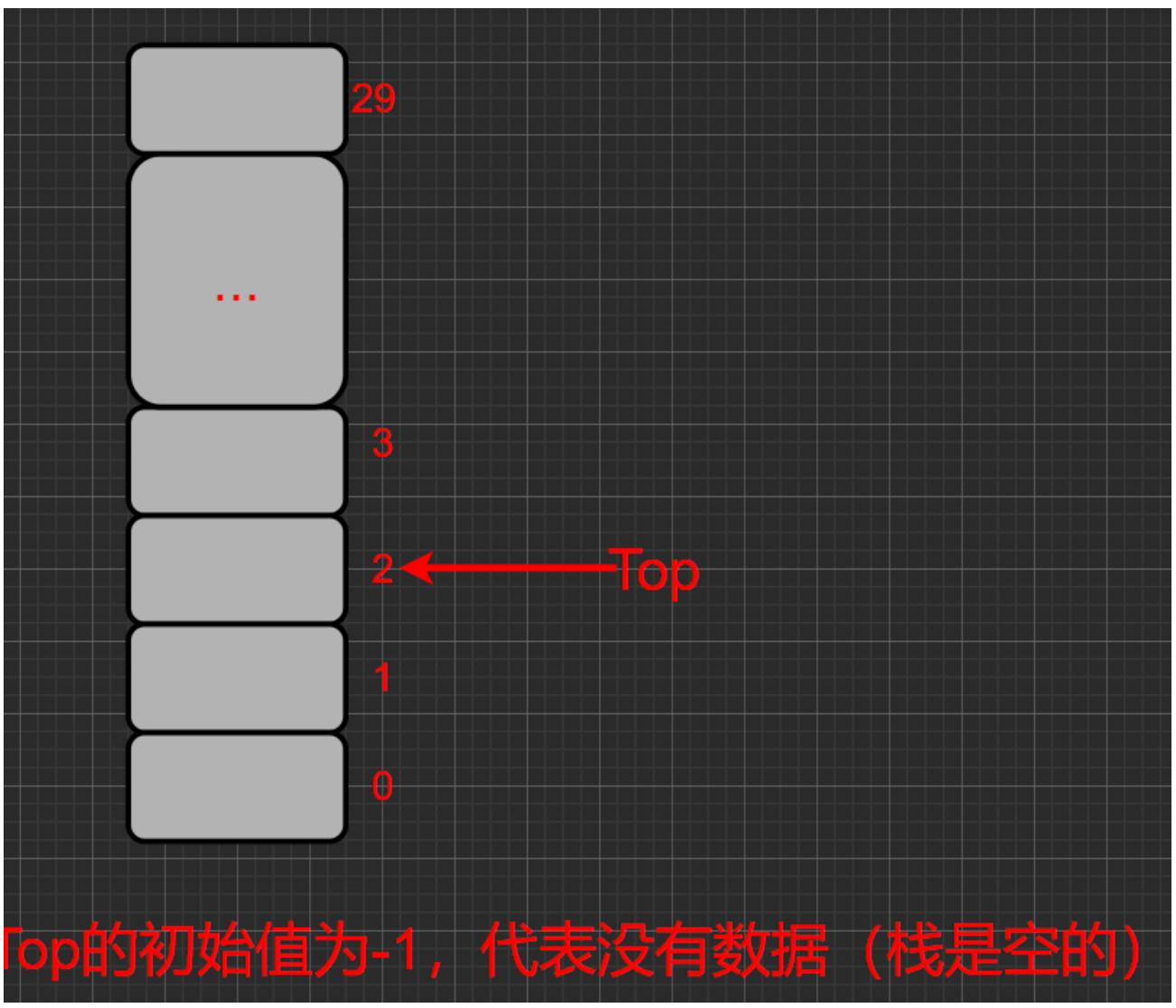
2.3栈的常见操作

1. 创建栈
2. 判满（针对顺序栈）
3. 入栈
4. 判空
5. 出栈

2.4顺序栈

2.4.1顺序栈的结构

```
#define N 30
#define datatype int
typedef struct{
    datatype data[N]; //数据域
    int top;          //栈顶位置
}seqstack_t;
```



2.4.2顺序栈的代码

seqstack.h

```h
#ifndef __SEQSTACK_H__
#define __SEQSTACK_H__

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 30
#define datatype int
typedef struct {
    datatype data[N]; // 数据域
    int top; // 栈顶位置
} seqstack_t;

seqstack_t * SeqStackCreate(void); //顺序栈的创建
int SeqStackIsFull(seqstack_t *h); //判满
int SeqStackPush(seqstack_t *h,datatype data);//入栈
int SeqStackIsEmpty(seqstack_t *h);//判空
datatype SeqStackPop(seqstack_t *h);//出栈
void SeqStackShow(seqstack_t *h);   //遍历

#endif
```

seqstack.c

```c
#include "seqstack.h"

seqstack_t* SeqStackCreate(void)
{
    seqstack_t* h;
    h = (seqstack_t*)malloc(sizeof(*h));
    if (h == NULL) {
        printf("%s malloc memory error\n", __func__);
        return NULL;
    }
    // #include <string.h>
    // 原型：void *memset(void *s, int c, size_t n);
    // 功能：内存填充
    // 参数：
    //      @s:被填充的首地址
    //      @c：被填充的数值
    //      @n:被填充内存大小，单位是字节
    // 返回值：返回s指针
    // 将data数组中的数据全部清零
    memset(h->data, 0, sizeof(datatype) * N);
    h->top = -1; // 空栈

    return h;
}
int SeqStackIsFull(seqstack_t* h)
{
    return (h->top + 1 == N) ? 1 : 0;
}
int SeqStackPush(seqstack_t* h, datatype data)
{
    if (SeqStackIsFull(h)) {
        printf("%s stack is full\n", __func__);
        return -1;
    }

    h->data[++h->top] = data;
    return 0;
}
int SeqStackIsEmpty(seqstack_t* h)
{
    return h->top == -1 ? 1 : 0;
}
datatype SeqStackPop(seqstack_t* h)
{
    if (SeqStackIsEmpty(h)) {
        printf("%s stack is empty\n", __func__);
        return (datatype)-1;
    }
    return h->data[h->top--];
}
void SeqStackShow(seqstack_t* h)
{
    for(int i=0;i<=h->top;i++){
        printf("-%d",h->data[i]);
```

```
    }
    printf("-\n");
}

```

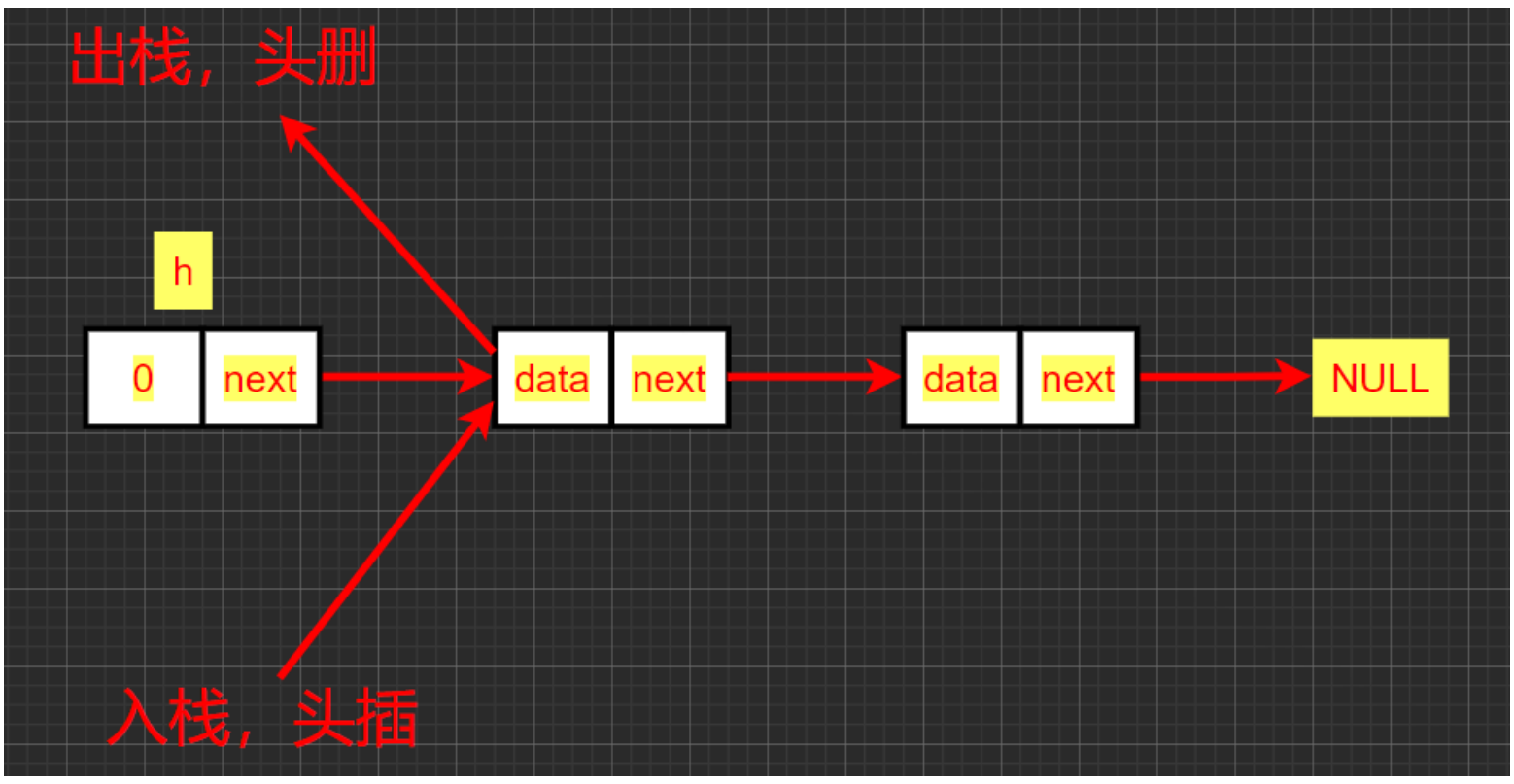
main.c

```
#include "seqstack.h"

int main(int argc, const char* argv[])
{
    seqstack_t* h;

    h = SeqStackCreate();
    if (h == NULL)
        return -1;

    for(int i=0;i<N;i++){
        SeqStackPush(h,i+1);
    }
    SeqStackShow(h);

    for(int i=0;i<N;i++){
        SeqStackPop(h);
        SeqStackShow(h);
    }
    return 0;
}
```

2.5链式栈

2.5.1链式栈的结构

```
#define datatype int
typedef struct node{
    datatype data;
    struct node *next;
}linkstack_t;
```



2.5.2链式栈的整体代码

linkstack.h

```
#ifndef __LINKSTACK_H__
#define __LINKSTACK_H__

#include <stdio.h>
#include <stdlib.h>

#define datatype int
typedef struct node{
    datatype data;
    struct node *next;
}linkstack_t;

linkstack_t *LinkStackCreate(void); //创建链式栈
int LinkStackPush(linkstack_t *h,datatype data); //头插，入栈
int LinkStackIsEmpty(linkstack_t *h);//判空
datatype LinkStackPop(linkstack_t *h);//头删，出栈
void LinkStackShow(linkstack_t *h);//遍历
```

```
#endif
```

linkstack.c

```c
#include "linkstack.h"
linkstack_t* LinkStackCreate(void)
{
    linkstack_t* h;
    h = (linkstack_t*)malloc(sizeof(*h));
    if (h == NULL) {
        printf("%s malloc error\n", __func__);
        return NULL;
    }
    h->data = (datatype)0;
    h->next = NULL;

    return h;
}
int LinkStackPush(linkstack_t* h, datatype data)
{
    linkstack_t* tmp;
    tmp = (linkstack_t*)malloc(sizeof(*tmp));
    if (tmp == NULL) {
        printf("%s malloc error\n", __func__);
        return -1;
    }
    tmp->data = data;

    tmp->next = h->next;
    h->next = tmp;

    return 0;
}
int LinkStackIsEmpty(linkstack_t* h)
{
    return h->next == NULL ? 1 : 0;
}
datatype LinkStackPop(linkstack_t* h)
{
    linkstack_t *tmp;
    datatype data;
    if(LinkStackIsEmpty(h)){
        printf("%s link stack empty\n",__func__);
        return (datatype)-1;
    }
    tmp = h->next;

    h->next = tmp->next;

    data = tmp->data;
    if(tmp!=NULL){
        free(tmp);
        tmp=NULL;
    }

    return data;
}
void LinkStackShow(linkstack_t* h)
{
    while(h->next){
        printf("-%d",h->next->data);
        h = h->next;
    }
    printf("-\n");
}
```

main.c

```c
#include "linkstack.h"

int main(int argc, const char* argv[])
{
    linkstack_t* h;
    h = LinkStackCreate();
    if (h == NULL) {
        return -1;
    }
    LinkStackPush(h, 567);
    LinkStackPush(h, 666);
    LinkStackPush(h, 321);
    LinkStackPush(h, 777);

```

```
    LinkStackShow(h);

    LinkStackPop(h);
    LinkStackShow(h);

    return 0;
}
```

2.6栈的常见笔试题

一个栈的入栈序列为A B C D E 则不可能的输出序列为（ C ）

A. EDCBA

B. DECBA

C. DCEAB

D. ABCDE

3.队列

3.1队列的特点

队列：它是一个先进先出的数据结构（FIFO）,对于队列有一个队列头和

一个队列尾，入队的时候从队尾入队，出队的时候从对头出队。通过front

记录对头，通过rear记录队尾。

3.2队列的种类

顺序队列：它是通过数组实现的

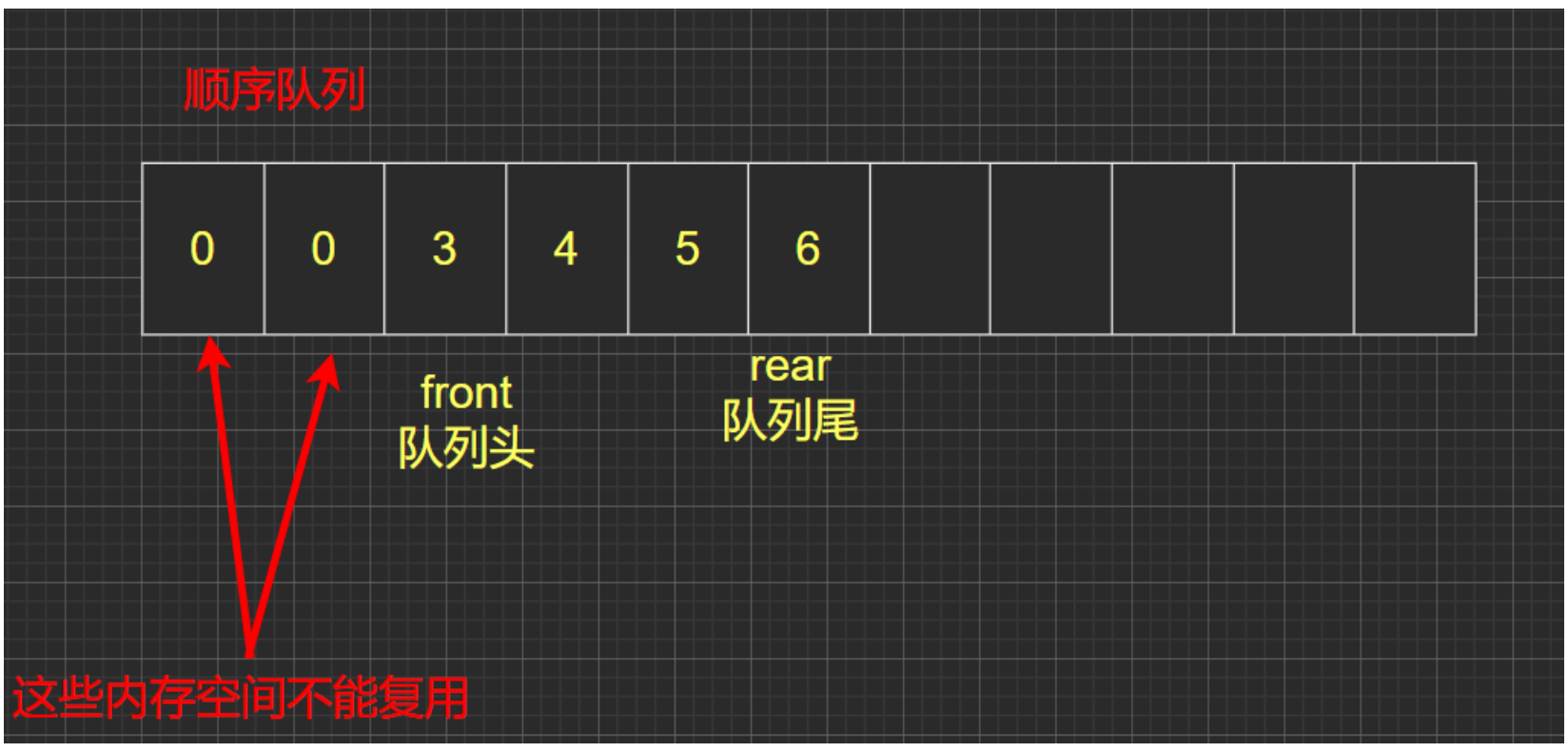
循环队列：它是通过数组实现的

链式队列：它是通过链表实现的

3.3队列的常见操作

1. 队列的创建
2. 判满（数组实现的队列）
3. 入队
4. 判空
5. 出队
6. 遍历

3.4顺序队列



顺序队列是通过数组实现的，通过front记录对头，通过rear记录队尾。入队

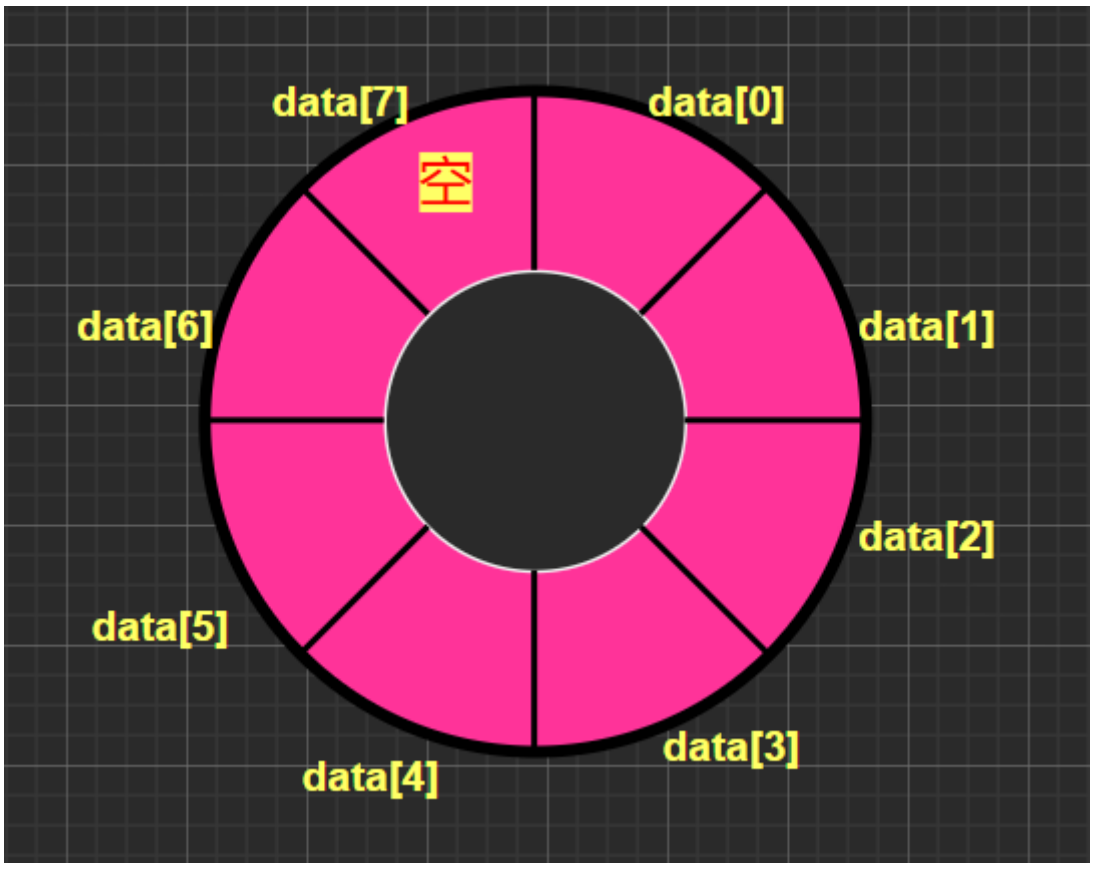
的时候从队尾入，rear向后走。出队的时候从队头出front也是向后走的，front

走过的位置不能复用，所以队列顺序队列在实际的工作开发过程中不适用。

3.5循环队列

3.5.1循环队列的特点

顺序队列内存不能够复用，为了能上内存复用所以设计了循环队列，循环队列能实现循环存储的原因就是那数组下标对数组成员的个数取余完成的。结构图如下：

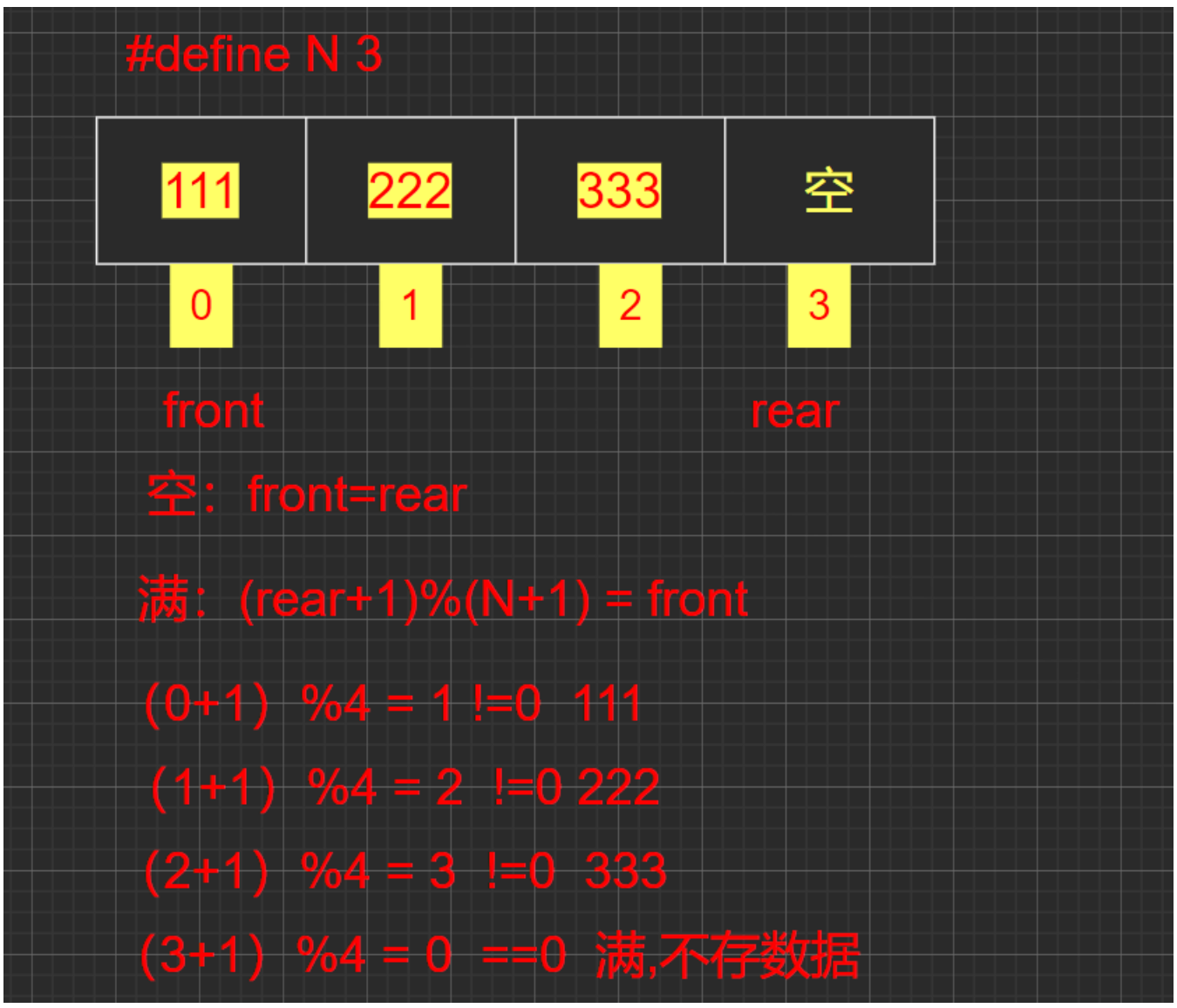


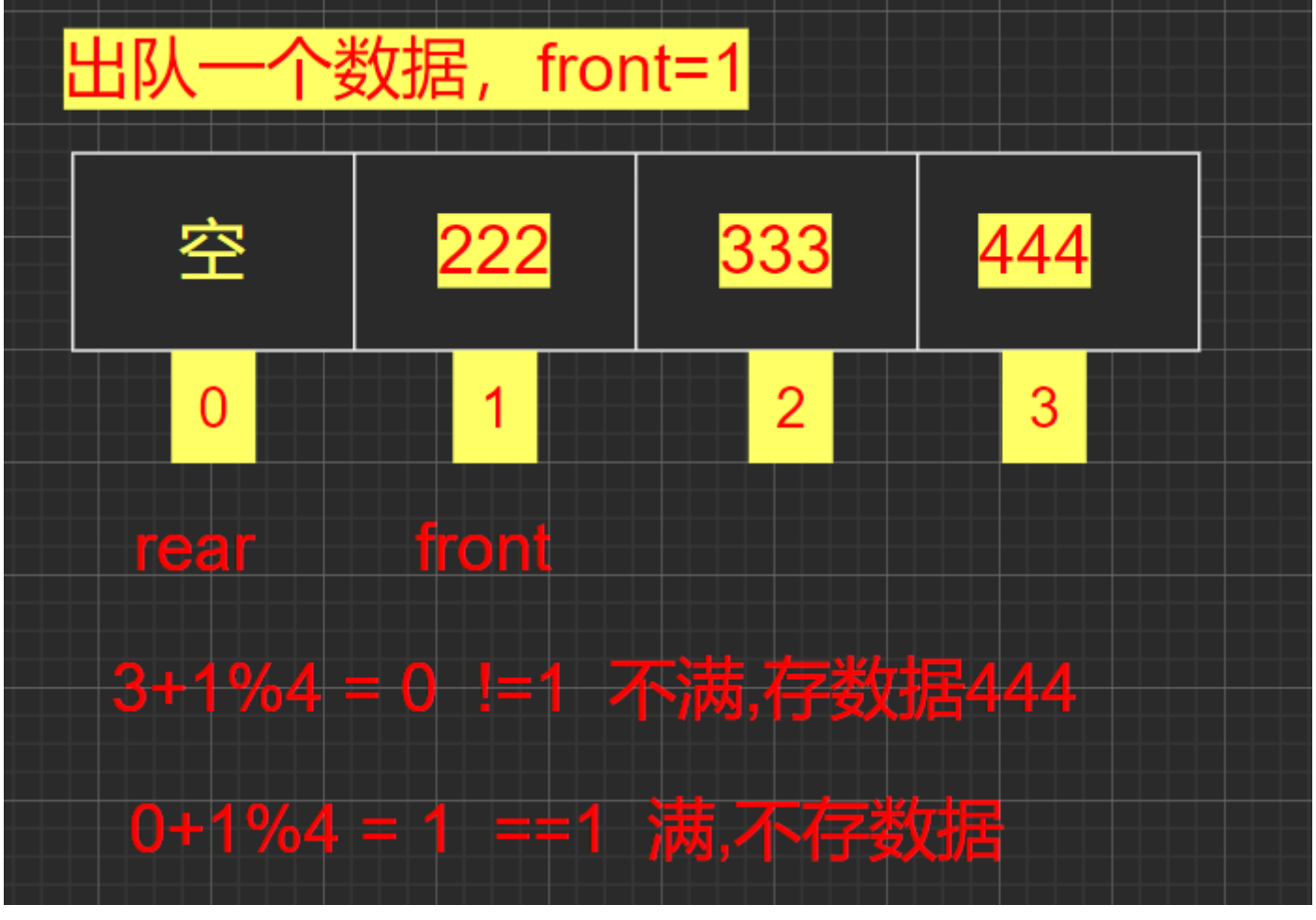
3.5.2循环队列的原理详解

顺序队列内存复用性差，所以引入了循环队列，循环队列能实现循环的思想就是借助数组下标对成员个数取余来实现循环的。但是如果数组成员是N，这个N个都要存数据的话，就没有办法判满和判空，因为判满和判空的条件是一样的。所以为了能够判满和判空专门浪费1块内存空来完成判满和判空。

`判空：front==rear`

`判满：（rear+1）%(N+1)==front` //N能存储数据的个数





3.5.3循环队列的整体代码

loopqueue.h

```h
#ifndef __LOOPQUEUE_H__
#define __LOOPQUEUE_H__

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 5 // 能存入数据的个数
#define datatype int

typedef struct {
    datatype data[N + 1]; // 数组
    int front, rear; // front是队列头的下标，rear是队尾的下标
} loopqueue_t;

loopqueue_t* LoopQueueCreate(void);
int LoopQueueIsFull(loopqueue_t* q);
int LoopQueueEnQueue(loopqueue_t* q, datatype data);
int LoopQueueIsEmpty(loopqueue_t* q);
datatype LoopQueueDeQueue(loopqueue_t* q);
void LoopQueueShow(loopqueue_t* q);
#endif
```

loopqueue.c

```c
#include "loopqueue.h"

loopqueue_t* LoopQueueCreate(void)
{
    loopqueue_t* q;
    q = (loopqueue_t*)malloc(sizeof(*q));
    if (q == NULL) {
        printf("%s malloc error\n", __func__);
        return NULL;
    }
    memset(q->data, 0, sizeof(datatype) * (N + 1));
    q->front = q->rear = 0;

    return q;
}
int LoopQueueIsFull(loopqueue_t* q)
{
    return ((q->rear + 1) % (N + 1) == q->front) ? 1 : 0;
}
int LoopQueueEnQueue(loopqueue_t* q, datatype data)
{
    if (LoopQueueIsFull(q)) {
        printf("%s is full,enter error\n", __func__);
        return -1;
    }
    // 从队尾入队
    q->data[q->rear] = data;
    q->rear = (q->rear + 1) % (N + 1);
```

```
    return 0;
}
int LoopQueueIsEmpty(loopqueue_t* q)
{
    return q->rear == q->front ? 1 : 0;
}
datatype LoopQueueDeQueue(loopqueue_t* q)
{
    datatype data;
    if (LoopQueueIsEmpty(q)) {
        printf("%s is empty,delete error\n", __func__);
        return (datatype)-1;
    }
    // 从队列头出队
    data = q->data[q->front];
    q->data[q->front] = 0;
    q->front = (q->front + 1) % (N + 1);
    return data;
}
void LoopQueueShow(loopqueue_t* q)
{
    for(int i=0;i<N+1;i++){
        printf("-%d",q->data[i]);
    }
    printf("-\n");
}
```

main.c

```
#include "loopqueue.h"
int main(int argc, const char* argv[])
{
    loopqueue_t* q;

    q = LoopQueueCreate();
    if (q == NULL)
        return -1;
    for (int i = 0; i < N; i++) {
        LoopQueueEnQueue(q, i + 1);
    }
    LoopQueueShow(q);

    LoopQueueDeQueue(q);
    LoopQueueEnQueue(q, 6);
    LoopQueueShow(q);

    LoopQueueDeQueue(q);
    LoopQueueShow(q);

    return 0;
}
```

> 循环队列中成员个数：
>
> if(rear>=front){
>
> return (rear-front);
>
> }else{
>
> return (rear-front+N+1);
>
> }

3.6链式队列

3.6.1链式队列的特点

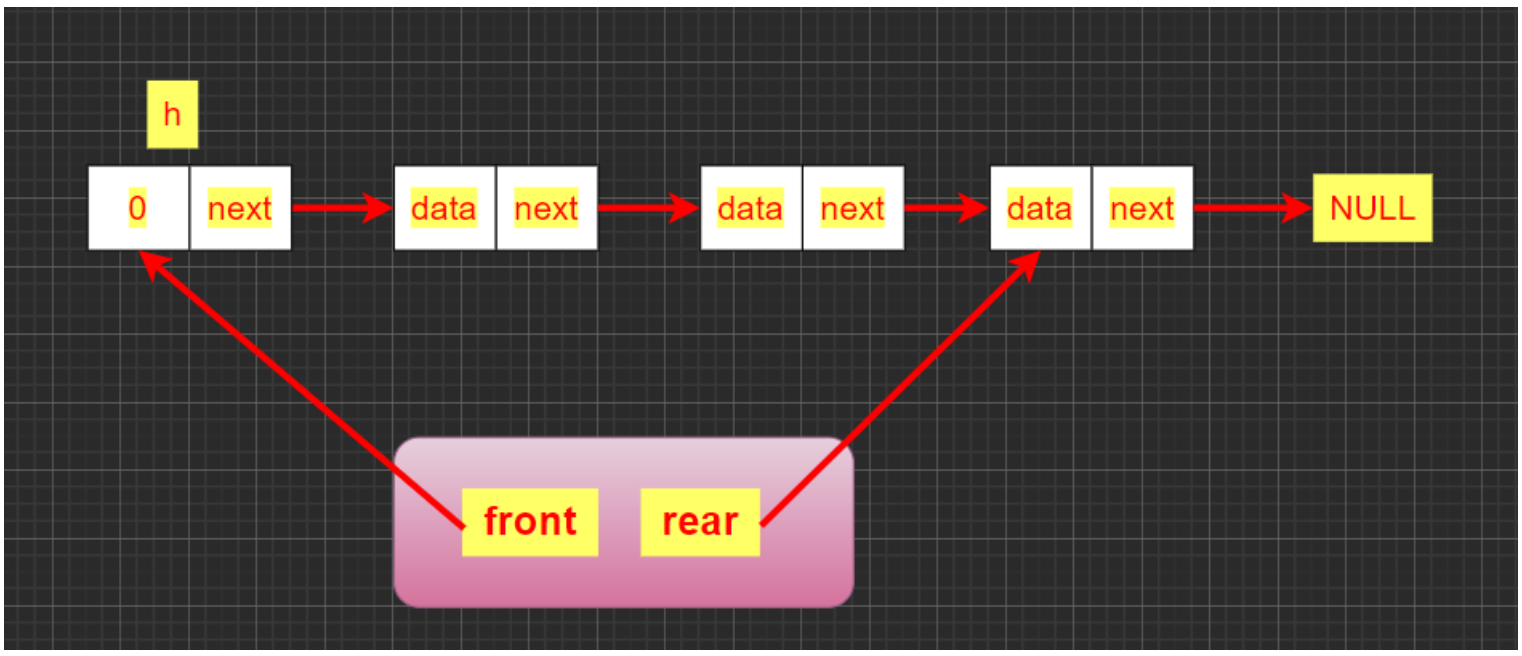
链式队列：链式队列是通过链表实现的，所以成员个数限制。

在链式队列中front指针指向队列的头，rear指针指向队列的尾。

如果通过rear完成，出队通过front完成。

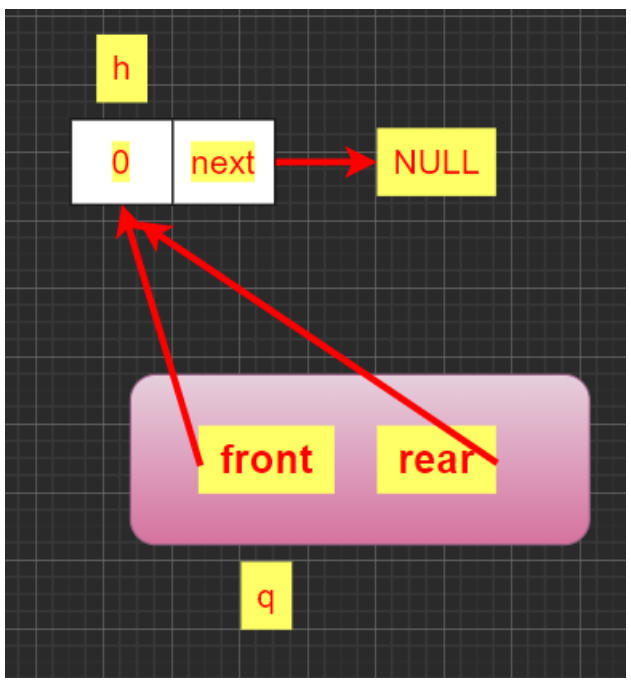
3.6.2链式队列的结构

```c
#define datatype int

typedef struct node{
    datatype data;
    struct node *next;
}node_t;

typedef struct{
    node_t *front,*rear;
}linkqueue_t;
```

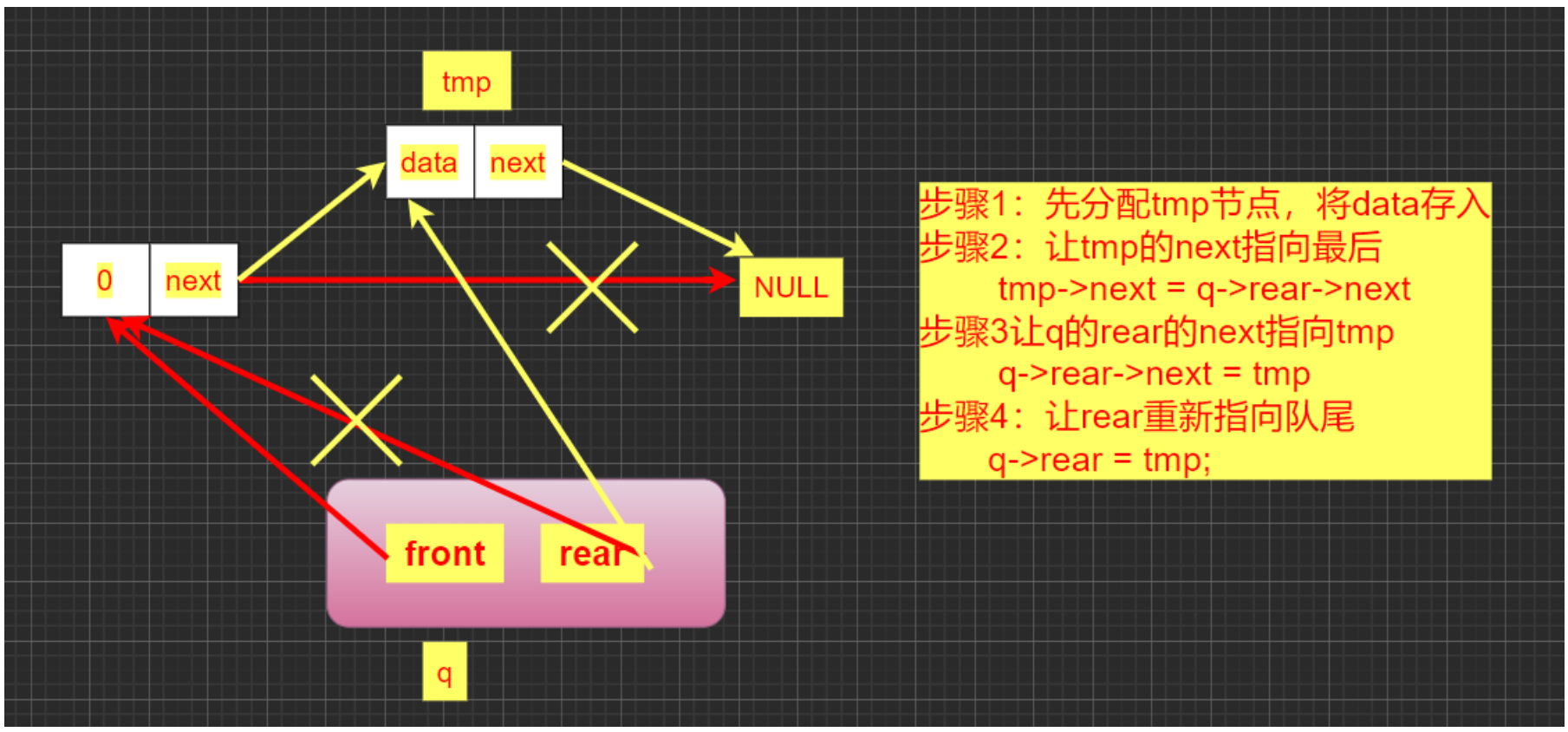


3.6.3链式队列的常见操作

1. 链式队列的创建

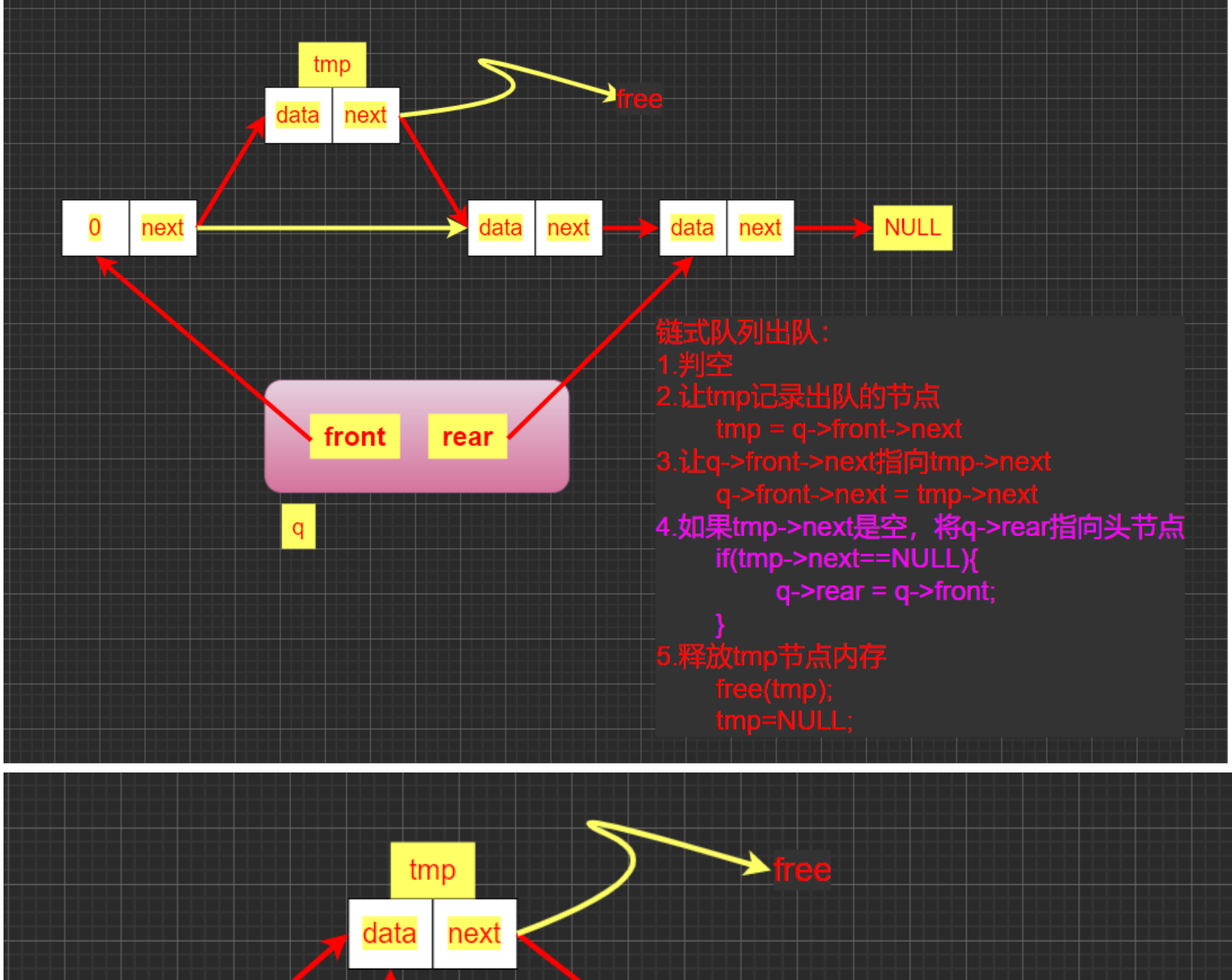


2. 链式队列的入队



3. 链式队列判空

4. 链式队列出队



3.6.4链式队列的整体代码

linkqueue.h

```c
#ifndef __LINKQUEUE_H__
#define __LINKQUEUE_H__

#include <stdio.h>
#include <stdlib.h>

#define datatype int
typedef struct node {
    datatype data;
    struct node* next;
} node_t;

typedef struct {
    node_t *front, *rear;
} linkqueue_t;
```

```
16
17 linkqueue_t* LinkQueueCreate(void);
18 int LinkQueueEnQueue(linkqueue_t* q, datatype data);
19 int LinkQueueIsEmpty(linkqueue_t* q);
20 datatype LinkQueueDeQueue(linkqueue_t* q);
21 void LinkQueueShow(linkqueue_t* q);
22 #endif
```

linkqueue.c

```
1  #include "linkqueue.h"
2
3  linkqueue_t* LinkQueueCreate(void)
4  {
5      node_t* h;
6      linkqueue_t* q;
7      h = (node_t*)malloc(sizeof(*h));
8      if (h == NULL) {
9          printf("%s malloc node_t error\n", __func__);
10         return NULL;
11     }
12     h->data = (datatype)0;
13     h->next = NULL;
14
15     q = (linkqueue_t*)malloc(sizeof(*q));
16     if (q == NULL) {
17         printf("%s malloc linkqueue_t error\n", __func__);
18         return NULL;
19     }
20     q->front = q->rear = h;
21
22     return q;
23 }
24
25 int LinkQueueEnQueue(linkqueue_t* q, datatype data)
26 {
27     node_t* tmp;
28     // 1.分配tmp(node_t*)
29     tmp = (node_t*)malloc(sizeof(*tmp));
30     if (tmp == NULL) {
31         printf("%s malloc node_t memory error\n", __func__);
32         return -1;
33     }
34     tmp->data = data;
35     // 2.将tmp入队
36     tmp->next = q->rear->next;
37     q->rear->next = tmp;
38     // 3.让rear记录当前队列队尾
39     q->rear = tmp;
40
41     return 0;
42 }
43 int LinkQueueIsEmpty(linkqueue_t* q)
44 {
45     return q->front == q->rear ? 1 : 0;
46 }
47
48 datatype LinkQueueDeQueue(linkqueue_t *q)
49 {
50     node_t *tmp;
51     datatype data;
52     if(LinkQueueIsEmpty(q)){
53         printf("%s linkqueue empty\n",__func__);
54         return (datatype)-1;
55     }
56
57     tmp = q->front->next;
58     q->front->next = tmp->next;
59     if(tmp->next == NULL){
60         q->rear = q->front;
61     }
62
63     data = tmp->data;
64     if(tmp!=NULL){
65         free(tmp);
66         tmp = NULL;
67     }
68
69     return data;
70 }
71
72 void LinkQueueShow(linkqueue_t* q)
73 {
74     node_t *h = q->front;
```

```
    while(h->next){
        printf("-%d",h->next->data);
        h = h->next;
    }
    printf("-\n");
}
```

main.c

```
#include "linkqueue.h"

int main(int argc, const char* argv[])
{
    linkqueue_t* q;

    q = LinkQueueCreate();
    if (q == NULL)
        return -1;
    for (int i = 0; i < 6; i++) {
        LinkQueueEnQueue(q, i + 111);
    }
    LinkQueueShow(q);

    LinkQueueDeQueue(q);
    LinkQueueShow(q);

    return 0;
}
```

4.作业

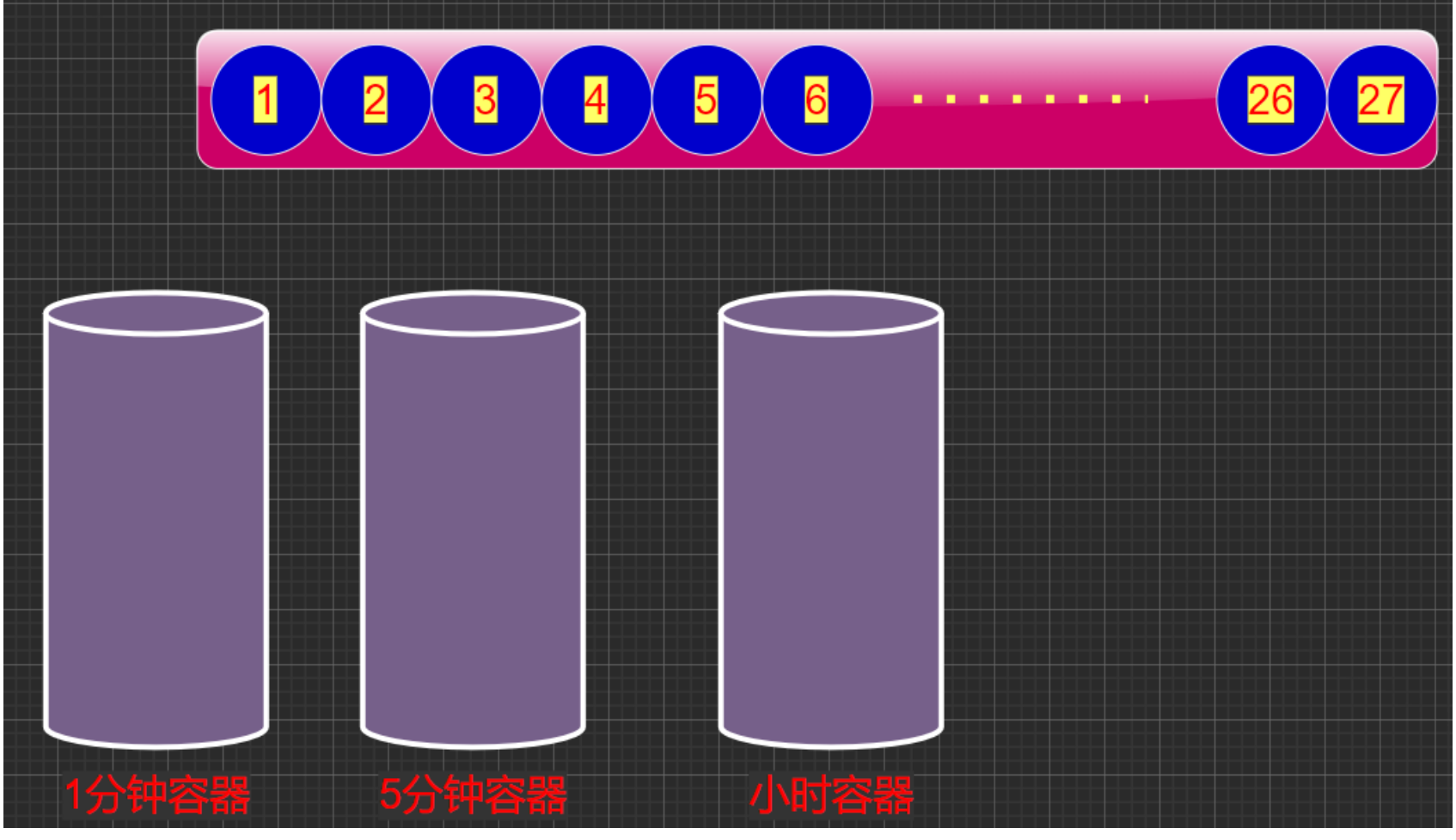
队列和栈结合实例　　结果：33120分钟，23天

问题描述：

球钟是一个利用球的移动来记录时间的简单装置。它有三个可以容纳若干个球的指示器：分钟指示器，五分钟指示器，小时指示器。若分钟指示器中有2个球，五分钟指示器中有6个球，小时指示器中有5个球，则时间为5:32

工作原理：每过一分钟，球钟就会从球队列的队首取出一个球放入分钟指示器，分钟指示器最多可容纳4个球。当放入第五个球时，在分钟指示器的4个球就会按照他们被放入时的相反顺序加入球队列的队尾。而第五个球就会进入五分钟指示器。按此类推，五分钟指示器最多可放11个球，小时指示器最多可放11个球。当小时指示器放入第12个球时，原来的11个球按照他们被放入时的相反顺序加入球队列的队尾，然后第12个球也回到队尾。这时，三个指示器均为空，回到初始状态，从而形成一个循环。因此，该球钟表示时间的范围是从0：00到11：59。

要求：现设初始时球队列的球数为27，球钟的三个指示器初态均为空。问，要经过多久，球队列才能回复到原来的顺序?



seqstack.h

```h
#ifndef __SEQSTACK_H__
#define __SEQSTACK_H__

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define datatype int
typedef struct {
    datatype *data; // 数据域
    int top; // 栈顶位置
} seqstack_t;

seqstack_t * SeqStackCreate(int max);
int SeqStackIsFull(seqstack_t *h,int max);
int SeqStackPush(seqstack_t *h,int max,datatype data);
int SeqStackIsEmpty(seqstack_t *h);
datatype SeqStackPop(seqstack_t *h);
void SeqStackShow(seqstack_t *h);

#endif
```

seqstack.c

```c
#include "seqstack.h"

seqstack_t* SeqStackCreate(int max)
{
    seqstack_t* h;
    h = (seqstack_t*)malloc(sizeof(*h));
    if (h == NULL) {
        printf("%s malloc memory error\n", __func__);
        return NULL;
    }
    h->data = (datatype *)malloc(max*sizeof(datatype));
    if(h->data == NULL){
        printf("malloc data memory error\n");
        return NULL;
    }
    h->top = -1; // 空栈
    return h;
}
int SeqStackIsFull(seqstack_t* h,int max)
{
    return (h->top + 1 == max) ? 1 : 0;
}
int SeqStackPush(seqstack_t* h,int max,datatype data)
{
    if (SeqStackIsFull(h,max)) {
        // printf("%s stack is full\n", __func__);
        return -1;
    }

    h->data[++h->top] = data;
```

```
    return 0;
}
int SeqStackIsEmpty(seqstack_t* h)
{
    return h->top == -1 ? 1 : 0;
}
datatype SeqStackPop(seqstack_t* h)
{
    if (SeqStackIsEmpty(h)) {
        printf("%s stack is empty\n", __func__);
        return (datatype)-1;
    }
    return h->data[h->top--];
}
void SeqStackShow(seqstack_t* h)
{
    for (int i = 0; i <= h->top; i++) {
        printf("-%d", h->data[i]);
    }
    printf("-\n");
}

```

linkqueue.h

```
#ifndef __LINKQUEUE_H__
#define __LINKQUEUE_H__

#include <stdio.h>
#include <stdlib.h>

#define datatype int
typedef struct node {
    datatype data;
    struct node* next;
} node_t;

typedef struct {
    node_t *front, *rear;
} linkqueue_t;

linkqueue_t* LinkQueueCreate(void);
int LinkQueueEnQueue(linkqueue_t* q, datatype data);
int LinkQueueIsEmpty(linkqueue_t* q);
datatype LinkQueueDeQueue(linkqueue_t* q);
void LinkQueueShow(linkqueue_t* q);
int LinkQueueCompare(linkqueue_t *q);
#endif
```

linkqueue.c

```
#include "linkqueue.h"

linkqueue_t* LinkQueueCreate(void)
{
    node_t* h;
    linkqueue_t* q;
    h = (node_t*)malloc(sizeof(*h));
    if (h == NULL) {
        printf("%s malloc node_t error\n", __func__);
        return NULL;
    }
    h->data = (datatype)0;
    h->next = NULL;

    q = (linkqueue_t*)malloc(sizeof(*q));
    if (q == NULL) {
        printf("%s malloc linkqueue_t error\n", __func__);
        return NULL;
    }
    q->front = q->rear = h;

    return q;
}

int LinkQueueEnQueue(linkqueue_t* q, datatype data)
{
    node_t* tmp;
    // 1.分配tmp(node_t*)
    tmp = (node_t*)malloc(sizeof(*tmp));
    if (tmp == NULL) {
        printf("%s malloc node_t memory error\n", __func__);
```

```
        return -1;
    }
    tmp->data = data;
    // 2.将tmp入队
    tmp->next = q->rear->next;
    q->rear->next = tmp;
    // 3.让rear记录当前队列队尾
    q->rear = tmp;

    return 0;
}
int LinkQueueIsEmpty(linkqueue_t* q)
{
    return q->front == q->rear ? 1 : 0;
}

datatype LinkQueueDeQueue(linkqueue_t *q)
{
    node_t *tmp;
    datatype data;
    if(LinkQueueIsEmpty(q)){
        printf("%s linkqueue empty\n",__func__);
        return (datatype)-1;
    }

    tmp = q->front->next;
    q->front->next = tmp->next;
    if(tmp->next == NULL){
        q->rear = q->front;
    }

    data = tmp->data;
    if(tmp!=NULL){
        free(tmp);
        tmp = NULL;
    }

    return data;
}

void LinkQueueShow(linkqueue_t* q)
{
    node_t *h = q->front;

    while(h->next){
        printf("-%d",h->next->data);
        h = h->next;
    }
    printf("-\n");
}

int LinkQueueCompare(linkqueue_t *q)
{
    node_t *h = q->front;

    while(h->next && h->next->next){
        if(h->next->data > h->next->next->data)
            return 0;
        h = h->next;
    }

    return 1;
}
```

ballclock.c

```
#include "linkqueue.h"
#include "seqstack.h"
#define OSMAX 4
#define FSMAX 11
#define HSMAX 11
int main(int argc, const char* argv[])
{
    seqstack_t *os, *fs, *hs;
    linkqueue_t* q;
    int count = 0;
    datatype data;
    // 1.创建三个栈
    if ((os = SeqStackCreate(OSMAX)) == NULL)
        return -1;
    if ((fs = SeqStackCreate(FSMAX)) == NULL)
        return -1;
    if ((hs = SeqStackCreate(HSMAX)) == NULL)
```

```
        return -1;

    // 2.创建一个队列
    if ((q = LinkQueueCreate()) == NULL)
        return -1;
    // 3.让1-27号球入队
    for (int i = 1; i <= 27; i++) {
        LinkQueueEnQueue(q, i);
    }
    // LinkQueueShow(q);

    // 4.从队列中每过1分钟取一个球，向栈中存放
    while (1) {
        // 从队列中出队一个球，就代表一分钟过去了
        data = LinkQueueDeQueue(q);
        // 记录分钟数
        count++;
        // 将球向1分钟的栈存放
        if (SeqStackPush(os, OSMAX, data) == -1) {
            // 将1分钟栈中的球取出，放回队列
            while (!SeqStackIsEmpty(os)) {
                // 出栈入队
                LinkQueueEnQueue(q, SeqStackPop(os));
            }
            // 将球向5分钟的栈存放
            if (SeqStackPush(fs, FSMAX, data) == -1) {
                // 将5分钟栈中的球取出，放回队列
                while (!SeqStackIsEmpty(fs)) {
                    // 出栈入队
                    LinkQueueEnQueue(q, SeqStackPop(fs));
                }
                // 将球向小时的栈存放
                if (SeqStackPush(hs, HSMAX, data) == -1) {
                    // 将小时栈中的球取出，放回队列
                    while (!SeqStackIsEmpty(hs)) {
                        // 出栈入队
                        LinkQueueEnQueue(q, SeqStackPop(hs));
                    }
                    //  将第27号球入队
                    LinkQueueEnQueue(q, data);

                    // 如果是有序的，函数返回1，循环退出
                    if (LinkQueueCompare(q))
                        break;
                }
            }
        }
    }

    printf("分钟数count = %d\n", count);
    return 0;
}
```