中国人民大学 信息学院
RENMIN UNIVERSITY OF CHINA SCHOOL OF INFORMATION

程序设计荣誉课程

13. 数据结构2——链表、栈、队列

授课教师：游伟 副教授、孙亚辉 副教授

授课时间：周二14:00 – 15:30，周四10:00 – 11:30（教学三楼3304）

上机时间：周二18:00 – 21:00（理工配楼二层机房）

课程主页：https://www.youwei.site/course/programming

目录

引子：约瑟夫问题（YOJ-604）

【题目描述】 N个小孩围成一圈，每个小孩依次编号1..N。从1号小孩开始报数，数到M的人出圈，再由下一个人重新从1开始报数，数到M的人出圈，依次类推，请问最后留在圈里的孩子的编号是多少。

【样例输入】5 3

【样例输出】4

【样例解释】出圈编号顺序为3->1->5->2，最后留下来的孩子编号为4。

【数据范围】2<=N，M<=1000

引子：约瑟夫问题（YOJ-604）

- 用数组解决约瑟夫问题（方法一：显式删除数组元素）

```
1. #include <stdio.h>

2. #define MAX 1001
3. int number[MAX]; //number[i]代表当前还在圈子里的第i个人原始的编号

4. int main()
5. {
6.     int N, M;
7.     int i, j, k, idx;

8.     scanf("%d %d", &N, &M);

9.     for (i = 1; i <= N; i++) number[i] = i; //初始时，第i个人的编号为i

10.    idx = 1;

11.    for (i = 1; i < N; i++) //执行N-1次出圈操作
12.    {
13.        //第i轮开始，圈子里还剩N-i+1个人；数到最后一个人，又从第一个人开始数起
14.        for (j = 1; j <= M - 1; j++) idx = idx % (N - i + 1) + 1;
15.        //第idx个人出圈，后面的人往前移动一个
16.        for (k = idx; k <= N - i; k++) number[k] = number[k+1];
17.    }

18.    printf("%d\n", number[1]); //N-1次出圈操作后留下最后一个人的编号记录在number[1]
19.}
```

引子：约瑟夫问题（YOJ-604）

- 用数组解决约瑟夫问题（方法二：隐式删除数组元素）

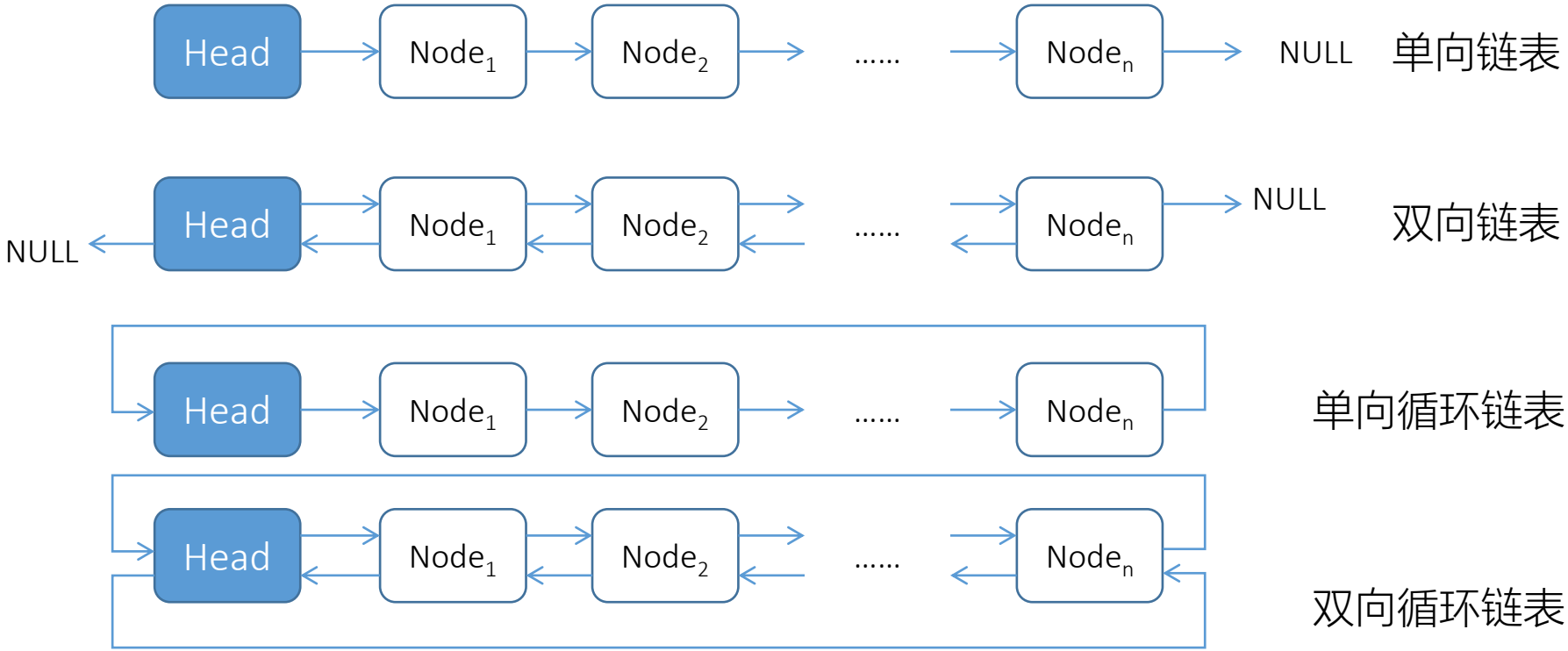
```
1. #include <stdio.h>

2. #define MAX 1001
3. int live[MAX]; //live[i]代表编号为i的人是否还存活在圈子里


4. int main() {
5.     int N, M;
6.     int i, j, idx;

7.     scanf("%d %d", &N, &M);

8.     for (i = 1; i <= N; i++) live[i] = 1; //初始时，N个人都存活在圈子里

9.     idx = 0;

10.    for (i = 1; i < N; i++) //执行N-1次出圈操作
11.    {
12.        for (j = 1; j <= M; j++)
13.          do { idx = idx % N + 1; } while (!live[idx]); //直到找到下一个还存活的人

14.        live[idx] = 0; //设置第idx个人不存活
15.    }

16.    for (i = 1; i <= N; i++) if (live[i]) printf("%d\n", i);
17.}
```

13.1 链表

- 链表 = 动态内存分配 + 结构体 + 指针
  - 按需创建结构体
  - 结构中包含指向同类型结构的指针
  - 通过指针连接成链表，终点是NULL指针
- 链表的特点
  - 链表中每一个元素称为“结点”，每个结点包括两个部分：
    - 数据域：用户需要的实际数据
    - 链域：指向前驱/后继结点的指针
  - 链表中各元素在逻辑上连续，在物理上不一定连续
    - 逻辑上：各元素通过链域构成前驱/后继关系
    - 物理上：各元素在内存中的地址可以是不连续的

13.1.1 基本概念

- 链表的类型
  - 单向链表：每个结点只有一个指向后继结点的指针
  - 双向链表：每个结点有一个指针指向前驱结点和一个指针指向后继结点
  - 循环链表：使最后一个结点的指针指向第一个结点



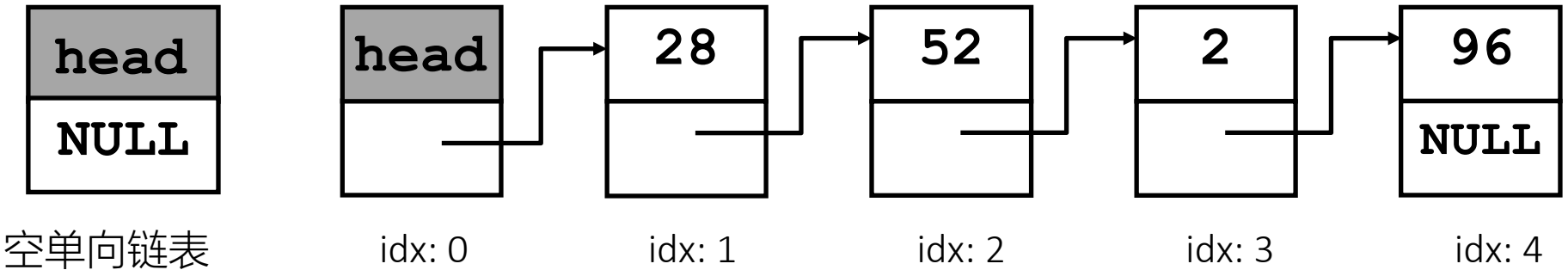
13.1.1 基本概念

- 链表的基本操作
  - 创建结点(CreateNode)：动态分配结构体空间，并对各个成员域赋值
  - 销毁结点(DestroyNode)：动态释放结构体内存空间
  - 初始化链表(Initialize)：创建一个头结点，并让头指针指向它
  - 清理链表(Finalize)：遍历链表，依次释放每个结点的内存空间
  - 检索(Search)：按照给定的结点索引号获得结点指针(Idx2Ptr)；按照给定的结点指针获得结点索引号(Ptr2Idx)
  - 插入(Insert)：在结点k_{i-1}与k_i之间插入一个新的结点k'，使线性表的长度增1，且k_{i-1}与k_i的逻辑关系发生如下变化：插入前，k_{i-1}是k_i的前驱，k_i是k_{i-1}的后继；插入后，新插入的结点k'成为k_{i-1}的后继、k_i的前驱
  - 删除(Delete)：删除结点k_i，使线性表的长度减1，且k_{i-1}、k_i和k_{i+1}之间的逻辑关系发生如下变化：删除前，k_i是k_{i+1}的前驱、k_{i-1}的后继；删除后，k_{i-1}成为k_{i+1}的前驱，k_{i+1}成为k_{i-1}的后继

13.1.2 单向链表

- 单向链表的定义
  - 头结点：在第一个正式结点之前增加一个特殊结点，使链表头指针永远非空，从而简化了插入/删除操作（无需区分头指针空与非空的不同情况）
  - 判断链表非空：head->next != NULL

```
typedef struct LinkNode {
   DataType data;            //可以通过typedef定义DataType
   struct LinkNode *next;    //指向下一个LinkNode类型的结构
} LinkNode;

LinkNode *head; //头结点指针
```



13.1.2 单向链表

- 创建结点

```
1.  LinkNode *CreateNode(DataType data)
2.  {
3.    LinkNode *node;

4.    node = (LinkNode *)malloc(sizeof(LinkNode));
5.    node->data = data;
6.    node->next = NULL;

7.    return node;
8.  }
```

- 销毁结点

```
1.  void DestroyNode(LinkNode *node)
2.  {
3.    //注意：若结构体中的数据域是由动态内存分配产生的，也应释放其内存空间
4.    free(node);
5.  }
```

思考1：为什么Initialize()函数的参数类型是LinkNode **，而Finalize()函数的参数类型是LinkNode *？

思考2：如果Finalize()函数中了绿色部分的代码替换成右边的红色代码会有什么问题？

13.1.2 单向链表

- 初始化链表

```
1.  void Initialize(LinkNode **head)
2.  {
3.      DataType data = 0;
4.      *head = CreateNode(data);
5.  }
```

- 清理链表

```
1.  void Finalize(LinkNode *head)
2.  {
3.      LinkNode *iter, *tmp;

4.      iter = head;

5.      while (iter != NULL)
6.      {
7.          tmp = iter;
8.          iter = iter->next;
9.          DestroyNode(tmp);
10.     }

11. }
```

```
while (iter != NULL)
{
    DestroyNode(iter);
    iter = iter->next;
}
```

```
for (iter = head;
     iter != NULL;
     iter = iter->next)
{
    DestroyNode(iter);
}
```

13.1.2 单向链表

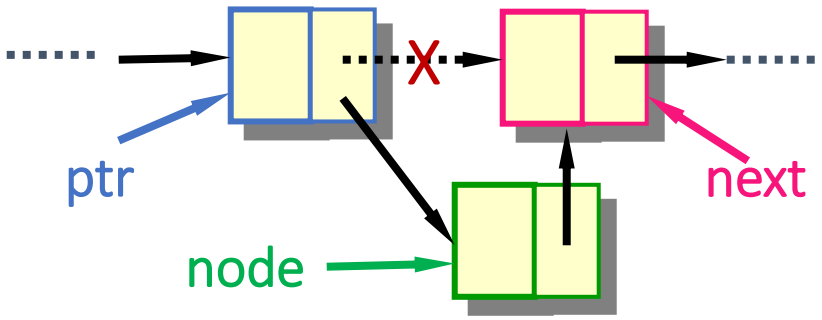
- 检索（Idx2Ptr）

```
1.  LinkNode *Idx2Ptr(LinkNode *head, int idx)
2.  {
3.      LinkNode *ptr = head;
4.      if (idx < 0) return NULL;

5.      while (idx > 0 && ptr != NULL) {
6.          ptr = ptr->next;
7.          idx--;
8.      }
9.      return ptr;
10. }
```

- 检索（Ptr2Idx）

```
1.  int Ptr2Idx(LinkNode *head, LinkNode *ptr)
2.  {
3.      LinkNode *iter = head;
4.      int idx = 0;

5.      while (iter != NULL && iter != ptr)
6.      {
7.          iter = iter->next;
8.          idx++;
9.      }

10.     if (iter != NULL) return idx;
11.     else return -1;
12. }
```

13.1.2 单向链表

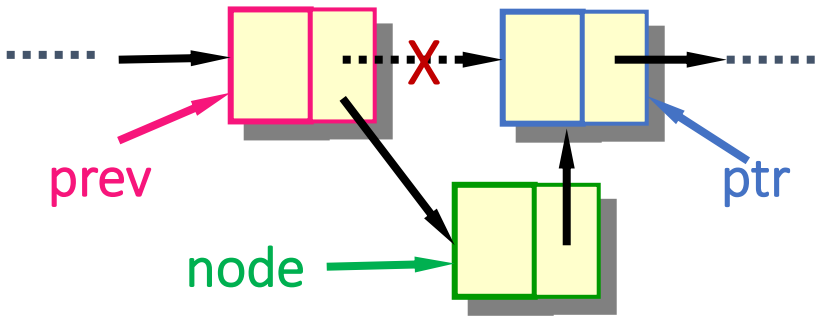


- 插入（后插）：结点node插到结点ptr（序号为idx）的后面

```
1.  void InsertAfterPtr(LinkNode *ptr, LinkNode *node)
2.  {
3.      LinkNode *next = ptr->next;
4.
5.      node->next = next;
6.      ptr->next = node;
7.  }
```

```
1.  void InsertAfterIdx(LinkNode *head, int idx, LinkNode *node)
2.  {
3.      LinkNode *ptr = Idx2Ptr(head, idx);
4.
5.      if (ptr != NULL) InsertAfterPtr(ptr, node);
6.  }
```

13.1.2 单向链表

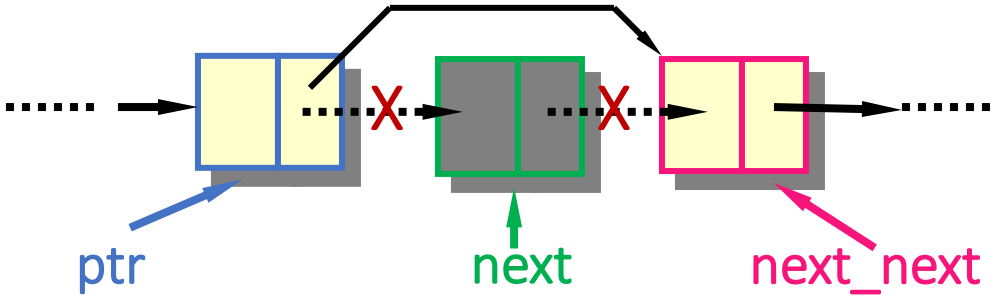


- 插入（前插）：结点node插到结点ptr（序号为idx）的前面

```
1.  void InsertBeforePtr(LinkNode *head, LinkNode *ptr, LinkNode *node)
2.  {
3.      int idx = Ptr2Idx(head, ptr);
4.      LinkNode *prev = Idx2Ptr(head, idx - 1);

5.      if (prev != NULL)
6.      {
7.        node->next = ptr;
8.        prev->next = node;
9.      }
10. }
```

```
1.  void InsertBeforeIdx(LinkNode *head, int idx, LinkNode *node)
2.  {
3.      LinkNode *ptr = Idx2Ptr(head, idx);

4.      if (ptr != NULL) InsertBeforePtr(head, ptr, node);
5.  }
```

13.1.2 单向链表

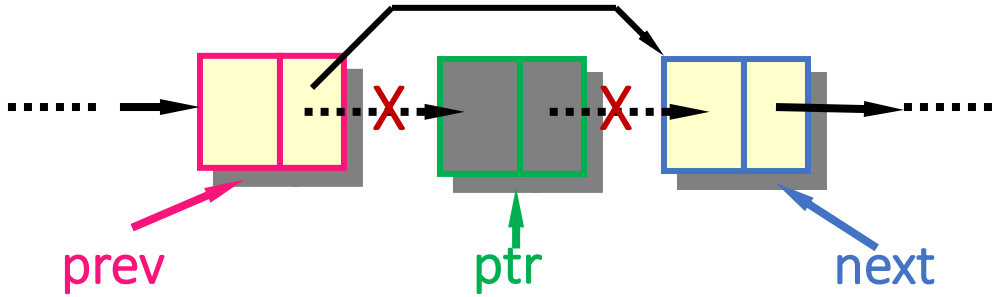


- 删除（下一个结点）

```
1.  void DeleteNextPtr(LinkNode *ptr)
2.  {
3.      LinkNode *next = ptr->next;
4.      LinkNode *next_next = (next != NULL ? next->next : NULL);

5.      if (next != NULL)
6.      {
7.          ptr->next = next_next;
8.          DestroyNode(next);
9.      }
10. }
```

```
1.  void DeleteNextIdx(LinkNode *head, int idx)
2.  {
3.      LinkNode *ptr = Idx2Ptr(head, idx);

4.      if (ptr != NULL) DeleteNextPtr(ptr);
5.  }
```

13.1.2 单向链表



- 删除（当前结点）

```
1.  void DeleteCurrPtr(LinkNode *head, LinkNode *ptr)
2.  {
3.      int idx = Ptr2Idx(head, ptr);
4.      LinkNode *prev = Idx2Ptr(head, idx - 1);
5.      LinkNode *next = ptr->next;

6.      if (prev != NULL)
7.      {
8.          prev->next = next;
9.          DestroyNode(ptr);
10.     }
11. }
```

```
1.  void DeleteCurrIdx(LinkNode *head, int idx)
2.  {
3.      LinkNode *ptr = Idx2Ptr(head, idx);

4.      if (ptr != NULL) DeleteCurrPtr(head, ptr);
5.  }
```

13.1.2 单向链表

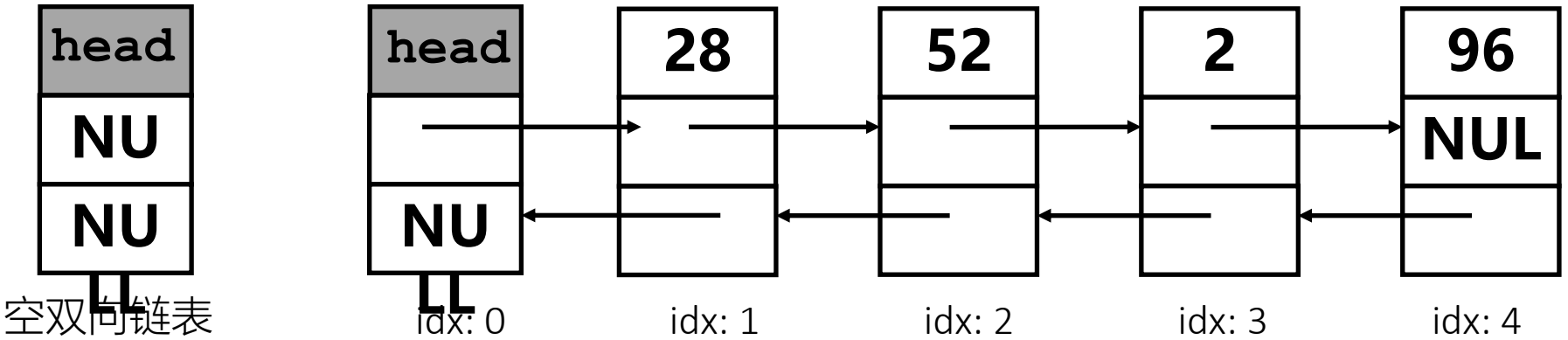
- 使用单向链表解决约瑟夫问题（joseph_single.c）

```
1. typedef int DataType;
2. typedef struct Node
3. {
4.     DataType data;
5.     struct Node *next;
6. } LinkNode;

7. LinkNode *head;

8. LinkNode *CreateNode(DataType data);
9. void DestroyNode(LinkNode *node);
10.void Initialize(LinkNode **head);
11.void Finalize(LinkNode *head);
12.int Ptr2Idx(LinkNode *head, LinkNode *ptr);
13.LinkNode *Idx2Ptr(LinkNode *head, int idx);
14.void InsertAfterPtr(LinkNode *ptr, LinkNode *node);
15.void DeleteCurrPtr(LinkNode *head, LinkNode *ptr);


16.int main() {
17.  int N, M, i;
18.  LinkNode *ptr, *node, *tmp;
19.
20.  scanf("%d %d", &N, &M);

21.  Initialize(&head); //初始化链表
22.  ptr = head;
23.  for (i = 1; i <= N; i++) //尾插法构造链表，结点编号1-N
24.  {
25.    node = CreateNode(i);
26.    InsertAfterPtr(ptr, node);
27.    ptr = node;
28.  }

29.  tmp = head;
30.  while (head->next->next != NULL)//直到链表中仅有一个元素
31.  {
32.    ptr = tmp;

33.    for (i = 1; i <= M; i++) //数M个数
34.    {
35.      tmp = ptr;
36.      ptr = ptr->next;
37.      //如果到达链表末尾，则重新回到开头
38.      if (ptr == NULL) ptr = head->next;
39.    }

40.    DeleteCurrPtr(head, ptr); //删除结点
41.  }

42.  printf("%d\n", head->next->data);

43.  Finalize(head); //清理链表
44.}
```

13.1.3 双向链表

- 双向链表的定义
  - 头结点：在第一个正式结点之前增加一个特殊结点，使链表头指针永远非空，从
    简化了插入/删除操作（无需区分头指针空与非空的不同情况）
  - 判断链表非空：head->next != NULL

```
typedef struct LinkNode {
   DataType data;              //可以通过typedef定义DataType
   struct LinkNode *next;      //指向下一个LinkNode类型的结构
   struct LinkNode *prev;      //指向上一个LinkNode类型的结构
} LinkNode;
LinkNode *head; //头结点指针
```



空双向链表

13.1.3 双向链表

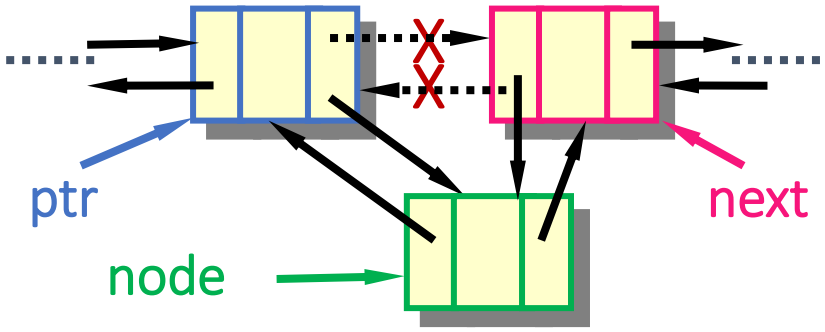
- 与单向链表不同的操作
  - 创建结点
  - 插入（后插/前插）
  - 删除（下一个/当前）
- 创建结点

```
1.  LinkNode *CreateNode(DataType data)
2.  {
3.    LinkNode *node;

4.    node = (LinkNode *)malloc(sizeof(LinkNode));
5.    node->data = data;
6.    node->next = NULL;
7.    node->prev = NULL;

8.    return node;
9.  }
```

13.1.3 双向链表

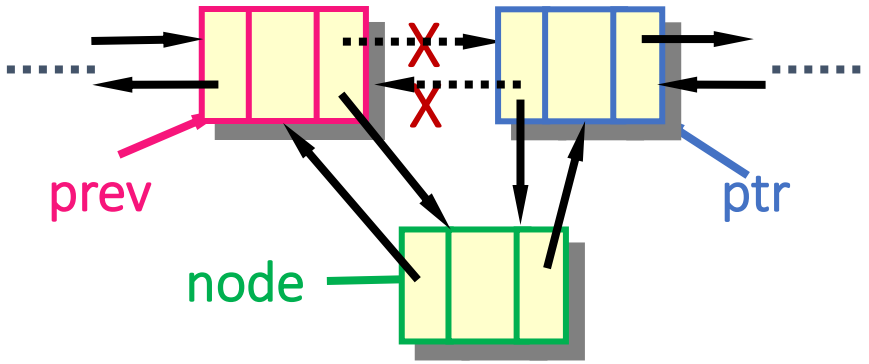
- 插入（后插）：结点node插到结点ptr的后面

```
1.  void InsertAfterPtr(LinkNode *ptr, LinkNode *node)
2.  {
3.      LinkNode *next = ptr->next;

4.      node->prev = ptr;
5.      node->next = next;

6.      ptr->next = node;
7.      if (next != NULL) next->prev = node;
8.  }
```



- 插入（前插）：结点node插到结点ptr的前面

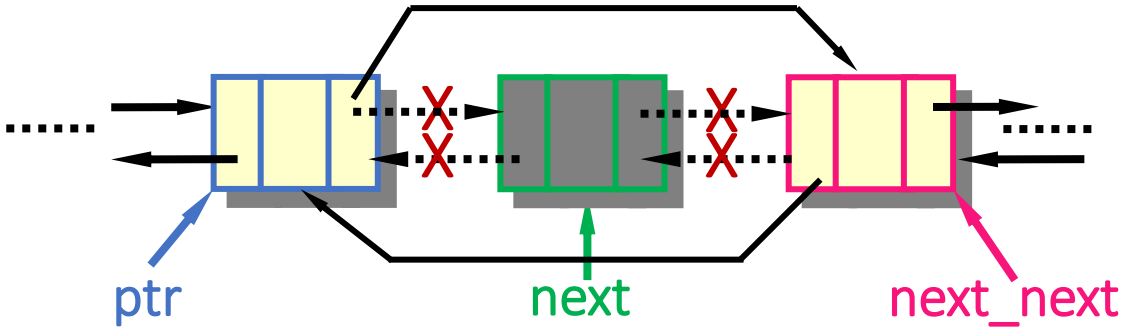
```
1.  void InsertBeforePtr(LinkNode *ptr, LinkNode *node)
2.  {
3.      LinkNode *prev = ptr->prev;

4.      node->prev = prev;
5.      node->next = ptr;

6.      ptr->prev = node;
7.      if (prev != NULL) prev->next = node;
8.  }
```



13.1.3 双向链表

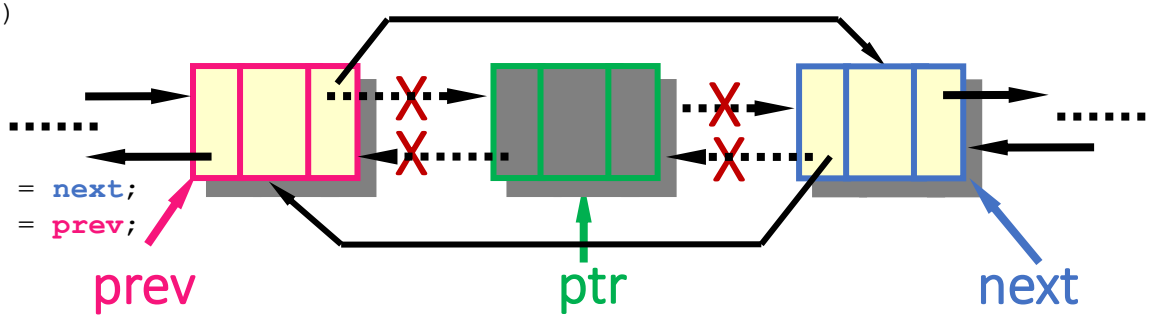
- 删除（下一个结点）

```
1.  void DeleteNextPtr(LinkNode *ptr)
2.  {
3.      LinkNode *next = ptr->next;
4.      LinkNode *next_next = (next != NULL ? next->next : NULL);

5.      if (next != NULL)
6.      {
7.          ptr->next = next_next;
8.          DestroyNode(next);
9.      }
10.     if (next_next != NULL) {
11.         next_next->prev = ptr;
12.     }
13. }
```



- 删除（当前结点）

```
1.  void DeleteCurrPtr(LinkNode *ptr)
2.  {
3.      LinkNode *prev = ptr->prev;
4.      LinkNode *next = ptr->next;

5.      if (prev != NULL) prev->next = next;
6.      if (next != NULL) next->prev = prev;

7.      DestroyNode(ptr);
8.  }
```



13.1.3 双向链表

- 使用双向链表解决约瑟夫问题（joseph_double.c）

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. typedef int DataType;
4. typedef struct LinkNode
5. {
6.     DataType data;
7.     struct LinkNode *next, *prev;
8. } LinkNode;

9. LinkNode *head;

10.LinkNode *CreateNode(DataType data);
11.void DestroyNode(LinkNode *node);
12.void Initialize(LinkNode **head);
13.void Finalize(LinkNode *head);
14.void InsertAfterPtr(LinkNode *ptr, LinkNode *node);
15.void DeleteCurrPtr(LinkNode *ptr);

16.int main() {
17.  int N, M, i;
18.  LinkNode *ptr, *node, *tmp;
19.
20.  scanf("%d %d", &N, &M);

21.  Initialize(&head); //初始化链表
22.  ptr = head;
23.  for (i = 1; i <= N; i++) //尾插法构造链表，结点编号1-N
24.  {
25.    node = CreateNode(i);
26.    InsertAfterPtr(ptr, node);
27.    ptr = node;
28.  }

29.  tmp = head;
30.  while (head->next->next != NULL)//直到链表中仅有一个元素
31.  {
32.    ptr = tmp;

33.    for (i = 1; i <= M; i++) //数M个数
34.    {
35.      tmp = ptr;
36.      ptr = ptr->next;
37.      //如果到达链表末尾，则重新回到开头
38.      if (ptr == NULL) ptr = head->next;
39.    }

40.    DeleteCurrPtr(ptr); //删除结点
41.  }

42.  printf("%d\n", head->next->data);

43.  Finalize(head); //清理链表
44.}
```

13.1.4 数组与链表的比较

- 线性表
  - 定义：由n个类型相同的数据元素组成的有限序列，记为$(a_1, a_2, \ldots, a_n)$
  - 性质：线性表的数据元素之间存在次序关系，a_{i-1}是a_i前驱，a_{i+1}是a_i后继
  - 分类：顺序存储（数组），链式存储（链表）
- 存储方式的比较
  - 数组：存储空间**静态分配**，各元素依次存放在一组**地址连续**的内存单元中。
  - 链表：存储空间**动态分配**，各元素的内存**地址不一定连续**，但**逻辑上连续**。
- 存储密度的比较（结点数据本身所占的存储量/结点结构所占的存储）
  - 数组：存储密度 = 1
  - 链表：存储密度 < 1（特别地，双向链表的存储密度 < 单向链表的存储密度）
- 存取方式的比较
  - 数组：可以随机存取，也可以顺序存取
  - 链表：只能顺序存取

13.1.4 数组与链表的比较

- 操作时间的比较

事实上，链表插入、删除运算的快捷是以空间代价来换取时间。

线性表类型	存取			插入		删除	
	前一个元素	后一个元素	任意元素	后插	前插	当前结点	下一个结点
数组	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)	O(n)
单向链表	O(n)	O(1)	O(n)	O(1)	O(n)	O(n)	O(1)
双向链表	O(1)	O(1)	O(n)	O(1)	O(1)	O(1)	O(1)

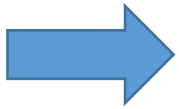
- 优缺点的比较
  - 数组的优点是存储空间利用率高，能支持顺序和随机存取；缺点是插入或删除元素不方便。
  - 链式存储的优点是插入或删除元素很方便，使用灵活；缺点是存储空间利用率低，不支持随机存取。
- 适用场景的比较
  - 数组适宜于做查找这样的静态操作。若线性表的长度变化不大，且主要操作是查找，则采用数组
  - 链表宜于做插入、删除这样的动态操作。若线性表的长度变化较大，且主要操作是插入、删除操作，则采用链表

引子：简单文字编辑器

设计简单文字编辑器，使其具有删除打错字符的功能。

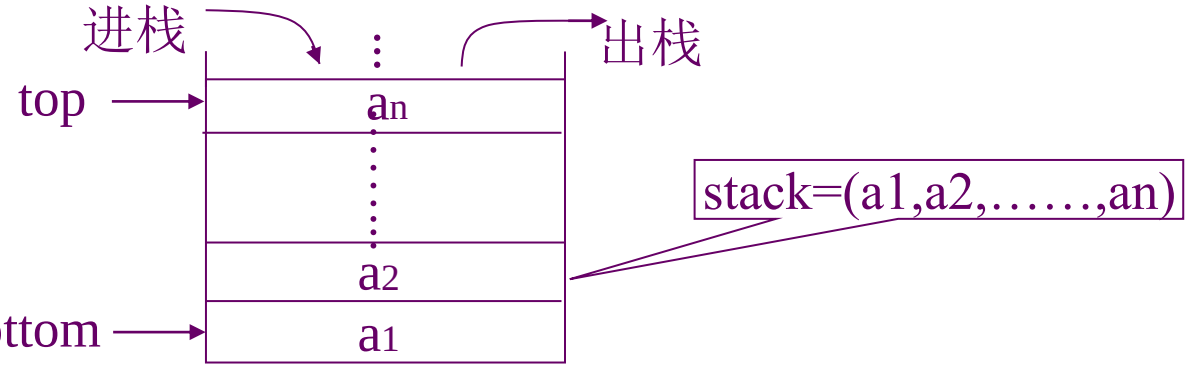
每读入一个字符

?	——	删除前面一个字符	出栈
*	——	删除前面所有字符	清空栈
#	——	输入结束	编辑结束
其它	——	记录输入的字符	入栈

“abc*def?gh#”  “degh”

我们用栈来实现这种功能的文字编辑器

13.2 栈



- 栈的定义和特点
  - 定义：限定仅在表尾进行插入或删除操作的线性表
    - 表尾—栈顶（top）
    - 表头—栈底（bottom）
  - 特点：先进后出（FILO）或者后进先出（LIFO）
- 栈的基本操作
  - 初始化栈(InitStack)：创建一个空栈
  - 清空栈(ClearStack)：清空栈中元素
  - 判断栈空(IsEmpty)：判断栈是否为空栈
  - 判断栈满(IsFull)：判断栈是否为满栈
  - 入栈(Push)：在栈顶插入元素
  - 出栈(Pop)：从栈顶删除元素
  - 取栈顶元素(Top)：取栈顶元素值，栈顶元素不出栈

13.2.1 顺序栈

- 顺序栈的存储结构（使用数组存储数据）

```
#define StackSize 1000
typedef struct {
  DataType data[StackSize];
  int top, cnt;       //cnt: 记录栈中元素个数
} SeqStack;

SeqStack stack;
```

- 约定：
  - --top指向最后一个入栈的元素，初始时设置top = 0
  - top == 0：表示栈空，top == StackSize：表示栈满
  - 栈满时执行入栈操作，会产生上溢错误（overflow）
  - 栈空时执行出栈或取栈顶元素操作，会产生下溢错误（underflow）

13.2.1 顺序栈

思考1：为什么函数InitStack()和ClearStack()的参数类型是SeqStack *?

思考2：为什么函数IsEmpty()和IsFull()的参数类型是SeqStack *?

思考3：为什么清空栈无需把栈的每个元素都回收?

- 初始化栈、清空栈

```
1.  void InitStack(SeqStack *stack)
2.  {
3.      stack->top = 0;
4.      stack->cnt = 0;
5.  }

6.  void ClearStack(SeqStack *stack)
7.  {
8.      stack->top = 0;
9.      stack->cnt = 0;
10. }
```

- 判断栈空、判断栈满

```
1.  int IsEmpty(SeqStack *stack)
2.  {
3.      return (stack->top == 0);
4.  }

5.  int IsFull(SeqStack *stack)
6.  {
7.      return (stack->top == StackSize);
8.  }
```

13.2.1 顺序栈

- 入栈

```
1.  void Push(SeqStack *stack, DataType data)
2.  {
3.      if (IsFull(stack)) { printf("overflow\n"); exit(-1); }
4.      else { stack->data[stack->top++] = data; stack->cnt++; }
5.  }
```

- 取栈顶元素、出栈

```
1.  DataType Top(SeqStack *stack) {
2.      if (IsEmpty(stack)) { printf("underflow\n"); exit(-1); }
3.      else { return stack->data[stack->top-1]; }
4.  }

5.  DataType Pop(SeqStack *stack)
6.  {
7.      DataType ret = Top(stack);
8.      stack->top--;
9.      stack->cnt--;
10.     return ret;
11. }
```

13.2.2 链式栈

- 链式栈的存储结构（使用链表存储数据）

```
typedef struct LinkNode{
  DataType data;
  struct LinkNode *next;
} LinkNode;

typedef struct {
  LinkNode *top;
  int cnt;      //cnt: 记录栈中元素个数
} LinkStack;

LinkStack stack;
```

- 约定：
  - top->next指向最后一个入栈的元素结点，初始时设置top = CreateNode(data)
  - top->next == NULL表示栈空
  - 当没有可以动态分配的内存时，栈满

13.2.2 链式栈

- 初始化栈

```
1.  void InitStack(LinkStack *stack)
2.  {
3.      DataType data = 0;
4.      stack->top = CreateNode(data);
5.      stack->cnt = 0;
6.  }
```

- 清空栈

```
1.  void ClearStack(LinkStack *stack)
2.  {
3.      while (!IsEmpty(stack)) Pop(stack);
4.  }
```

- 判断栈空

```
1.  int IsEmpty(LinkStack *stack)
2.  {
3.      return (stack->top->next == NULL);
4.  }
```

13.2.2 链式栈

- 入栈

```
1.  void Push(LinkStack *stack, DataType data)
2.  {
3.      LinkNode *node = CreateNode(data);
4.      if (node == NULL) { printf("overflow\n"); exit(-1); }
5.      else { InsertAfterPtr(stack->top, node); stack->cnt++; }
6.  }
```

- 取栈顶元素、出栈

```
1.  DataType Top(LinkStack *stack)
2.  {
3.      if (IsEmpty(stack)) { printf("underflow\n"); exit(-1); }
4.      else { return stack->top->next->data; }
5.  }

6.  DataType Pop(LinkStack *stack) {
7.      DataType ret = Top(stack);
8.      DeleteNextPtr(stack->top);
9.      stack->cnt--;
10.     return ret;
11. }
```

简单文字编辑器（顺序栈实现：editor_seq.c）

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. #define StackSize 1000
4. #define ArraySize 1000

5. typedef char DataType;
6. typedef struct
7. {
8.     DataType data[StackSize];
9.     int top;
10.    int cnt;
11.} SeqStack;

12.SeqStack stack;

13.void InitStack(SeqStack *stack);
14.void ClearStack(SeqStack *stack);
15.int IsEmpty(SeqStack *stack);
16.int IsFull(SeqStack *stack);
17.void Push(SeqStack *stack, DataType data);
18.DataType Top(SeqStack *stack);
19.DataType Pop(SeqStack *stack);
```

```
20.int main()
21.{
22.    char ch, result[ArraySize];
23.    int i, cnt = 0;

24.    InitStack(&stack);          //初始化栈

25.    while ((ch = getchar()) != EOF)
26.    {
27.        if (ch == '#') break;    //编辑结束
28.        else if (ch == '*')      //清空栈
29.            ClearStack(&stack);
30.        else if (ch == '?') Pop(&stack); //出栈
31.        else Push(&stack, ch);           //入栈
32.    }

33.    while (!IsEmpty(&stack))
34.        result[cnt++] = Pop(&stack);

35.    for (i = cnt-1; i >= 0; i--)
36.        printf("%c", result[i]);

37.    ClearStack(&stack);      //清空栈
38.}
```

简单文字编辑器（链式栈实现：editor_link.c）

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. #define ArraySize 1000

4. typedef char DataType;
5. typedef struct LinkNode
6. {
7.     DataType data; struct LinkNode *next;
8. } LinkNode;
9. typedef struct {
10.    LinkNode *top; int cnt;
11.} LinkStack;

12.LinkStack stack;

13.void InitStack(LinkStack *stack);
14.void ClearStack(LinkStack *stack);
15.int IsEmpty(LinkStack *stack);
16.void Push(LinkStack *stack, DataType data);
17.DataType Top(LinkStack *stack);
18.DataType Pop(LinkStack *stack);
19.LinkNode *CreateNode(DataType data);
20.void DestroyNode(LinkNode *node);
21.void InsertAfterPtr(LinkNode *ptr, LinkNode *node);
22.void DeleteNextPtr(LinkNode *ptr);
```

```
23.int main()
24.{
25.    char ch, result[ArraySize];
26.    int i, cnt = 0;

27.    InitStack(&stack);                  //初始化栈

28.    while ((ch = getchar()) != EOF)
29.    {
30.        if (ch == '#') break;           //编辑结束
31.        else if (ch == '*')             //清空栈
32.            ClearStack(&stack);
33.        else if (ch == '?')             //出栈
34.            Pop(&stack);
35.        else
36.            Push(&stack, ch);           //入栈
37.    }

38.     while (!IsEmpty(&stack))
39.        result[cnt++] = Pop(&stack);

40.    for (i = cnt-1; i >= 0; i--)
41.      printf("%c", result[i]);

42.    ClearStack(&stack);      //清空栈
43.}
```

引子：最近的请求次数

【题目描述】有一个系统陆续接受到 n 个服务请求，第 i 个请求的到达时间记作t_i。当系统接收到第 i 个请求后，会输出在 $[t_i-m, t_i]$ 这段时间里到达的请求次数 c_i。

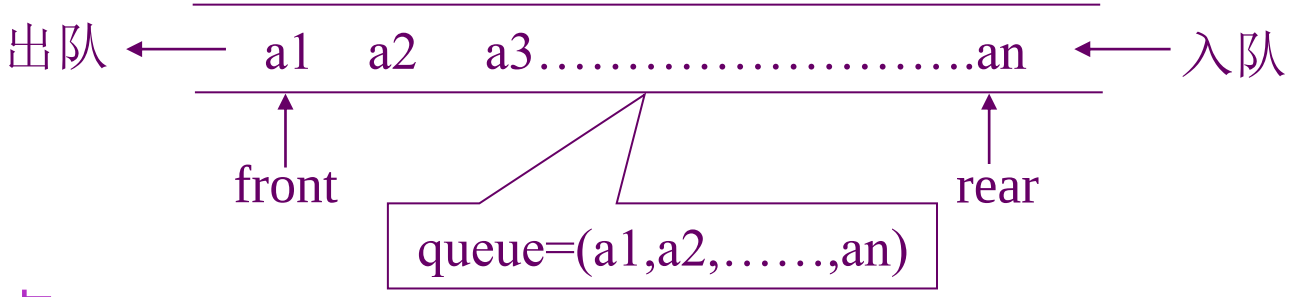
【输入格式】第一行两个正整数 n 和 m，第二行 n 个递增的正整数t_i

【输出格式】n个正整数，c_i $(1\leq i \leq n)$。

【样例输入】5 10 ↵ 1 9 10 15 25

【样例输出】1 2 3 3 2

【样例解释】第4个请求到达时间为15，在[5, 15]这段时间里共有3个请求到达（第2,3,4个请求）；第5个请求到达时间为25，在[15, 25]这段时间共有2个请求到达（第4,5个请求）



13.3 队列

- 队列的定义和特点
  - 定义：限定只能在表的一端进行插入，在表的另一端进行删除的线性表
    - 队尾（rear）——允许插入的一端
    - 队头（front）——允许删除的一端
  - 特点：先进先出(FIFO)
- 队列的基本操作
  - 初始化队列(InitQueue)：创建一个空队列
  - 清空队列(ClearQueue)：清空队列中元素
  - 判断队列空(IsEmpty)：判断队列是否为空队列
  - 入队(Enqueue)：在队尾插入元素
  - 出队(Dequeue)：从队头取出元素
  - 取队头元素(Front)：取队头元素值，队头元素不出队

思考1：rear == QueueSize 时，队列真的满了吗？

13.3.1 顺序队列

- 顺序队列的存储结构（使用数组存储数据）

```
#define QueueSize 1000
typedef struct {
  DataType data[QueueSize];
  int front, rear;
  int cnt;    //cnt: 记录队列中元素个数
} SeqQueue;

SeqQueue queue;
```

- 约定：
  - front指向当前队列中最早入队的元素，初始时设置front = 0
  - --rear指向当前队列中最晚入队的元素，初始时设置rear = 0
  - front == rear：表示队列空，rear == QueueSize：表示队列满
  - 队列满时执行入队操作，会产生上溢错误（overflow）
  - 队列空时执行出队或取队头元素操作，会产生下溢错误（underflow）

13.3.1 顺序队列

- 初始化队列、清空队列

```
1.  void InitQueue(SeqQueue *queue)
2.  {
3.      queue->front = queue->rear = 0;
4.      queue->cnt = 0;
5.  }

6.  void ClearQueue(SeqQueue *queue)
7.  {
8.      queue->front = queue->rear = 0;
9.      queue->cnt = 0;
10. }
```

- 判断队列空、判断队列满

```
1.  int IsEmpty(SeqQueue *queue)
2.  {
3.      return (queue->front == queue->rear);
4.  }

5.  int IsFull(SeqQueue *queue)
6.  {
7.      return (queue->rear == QueueSize);
8.  }
```

13.3.1 顺序队列

- 入队

```
1.  void Enqueue(SeqQueue *queue, DataType data)
2.  {
3.      if (IsFull(queue)) { printf("overflow\n"); exit(-1); }
4.      else { queue->data[queue->rear++] = data; queue->cnt++; }
5.  }
```

- 取队头元素、出队

```
1.  DataType Front(SeqQueue *queue)
2.  {
3.      if (IsEmpty(queue)) { printf("underflow\n"); exit(-1); }
4.      else { return queue->data[queue->front]; }
5.  }

6.  DataType Dequeue(SeqQueue *queue)
7.  {
8.      DataType ret = Front(queue);
9.      queue->front++;
10.     queue->cnt--;
11.     return ret;
12. }
```

13.3.2 链式队列

- 链式队列的存储结构（使用链表存储数据）

```
typedef struct LinkNode{
  DataType data;
  struct LinkNode *next;
} LinkNode;

typedef struct {
  LinkNode *front, *rear;
  int cnt;      //cnt: 记录栈中元素个数
} LinkQueue;

LinkQueue queue;
```

- 约定：
  - front->next指向第一个入队列的元素结点，rear指向最后一个入队的元素结点
  - 初始时，设置front = rear = CreateNode(data)
  - front == rear表示队列空
  - 当没有可以动态分配的内存时，队列满

13.3.2 链式队列

- 初始化队列

```
1.  void InitQueue(LinkQueue *queue)
2.  {
3.      DataType data = 0;
4.      queue->front = queue->rear = CreateNode(data);
5.      queue->cnt = 0;
6.  }
```

- 清空队列

```
1.  void ClearQueue(LinkQueue *queue)
2.  {
3.      while (!IsEmpty(queue)) Dequeue(queue);
4.  }
```

- 判断队列空

```
1.  int IsEmpty(LinkQueue *queue)
2.  {
3.      return (queue->front == queue->rear);
4.  }
```

13.3.2 链式队列

- 入队

```
1.  void Enqueue(LinkQueue *queue, DataType data)
2.  {
3.      LinkNode *node = CreateNode(data);
4.      if (node == NULL) { printf("overflow\n"); exit(-1); }
5.      else { InsertAfterPtr(queue->rear, node); queue->rear = node; queue->cnt++; }
6.  }
```

- 取队头元素、出队

```
1.  DataType Front(LinkQueue *queue)
2.  {
3.      if (IsEmpty(queue)) { printf("underflow\n"); exit(-1); }
4.      else { return queue->front->next->data; }
5.  }

6.  DataType Dequeue(LinkQueue *queue)
7.  {
8.      DataType ret = Front(queue);
9.      if (queue->front->next == queue->rear) queue->rear = queue->front; //临界情况
10.     DeleteNextPtr(queue->front);
11.     queue->cnt--;
12.     return ret;
13. }
```

最近的请求次数（顺序队列实现：service_seq.c)

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. #define QueueSize 1000

4. typedef int DataType;
5. typedef struct
6. {
7.     DataType data[QueueSize];
8.     int front, rear, cnt;
9. } SeqQueue;

10.SeqQueue queue;

11.void InitQueue(SeqQueue *queue);
12.void ClearQueue(SeqQueue *queue);
13.int IsEmpty(SeqQueue *queue);
14.int IsFull(SeqQueue *queue);
15.void Enqueue(SeqQueue *queue, DataType data);
16.DataType Front(SeqQueue *queue);
17.DataType Dequeue(SeqQueue *queue);
```

```
18.int main() {
19.     int n, m, i, t;
20.     DataType x;

21.     scanf("%d %d", &n, &m);

22.     InitQueue(&queue);

23.     for (i = 1; i <= n; i++) {
24.         scanf("%d", &t);
25.         Enqueue(&queue, t);
26.
27.         while (!IsEmpty(&queue))
28.         {    //队头元素不在[t-m, t]范围内就出队
29.             x = Front(&queue);
30.             if (x < t - m) Dequeue(&queue);
31.             else break;
32.         }

33.         printf("%d ", queue.cnt);
34.     }

35.     ClearQueue(&queue);
36.}
```

最近的请求次数（链式队列实现：service_link.c)

```
1. #include <stdio.h>
2. #include <stdlib.h>

3. typedef int DataType;
4. typedef struct LinkNode
5. {
6.     DataType data; struct LinkNode *next;
7. } LinkNode;
8. typedef struct
9. {
10.    LinkNode *front, *rear; int cnt;
11.} LinkQueue;
12.LinkQueue queue;

13.void InitQueue(LinkQueue *queue);
14.void ClearQueue(LinkQueue *queue);
15.int IsEmpty(LinkQueue *queue);
16.void Enqueue(LinkQueue *queue, DataType data);
17.DataType Front(LinkQueue *queue);
18.DataType Dequeue(LinkQueue *queue);
19.LinkNode *CreateNode(DataType data);
20.void DestroyNode(LinkNode *node);
21.void InsertAfterPtr(LinkNode *ptr, LinkNode *node);
22.void DeleteNextPtr(LinkNode *ptr);
```

```
23.int main() {
24.    int n, m, i, t;
25.    DataType x;

26.    scanf("%d %d", &n, &m);

27.    InitQueue(&queue);

28.    for (i = 1; i <= n; i++) {
29.        scanf("%d", &t);
30.        Enqueue(&queue, t);

31.        while (!IsEmpty(&queue))
32.        {   //队头元素不在[t-m, t]范围内就出队
33.            x = Front(&queue);
34.            if (x < t - m) Dequeue(&queue);
35.            else break;
36.        }

37.        printf("%d ", queue.cnt);
38.    }

39.    ClearQueue(&queue);
40.}
```

数据结构小结

栈和队列是操作受限的线性表

- 栈：只能在尾部插入和删除
- 队列：只能在尾部插入，在头部删除