

# Lecture 1

## array, list, stack, queue, big O

吳蔚琪 2022.10.14

# 991 《数据结构与算法》考纲

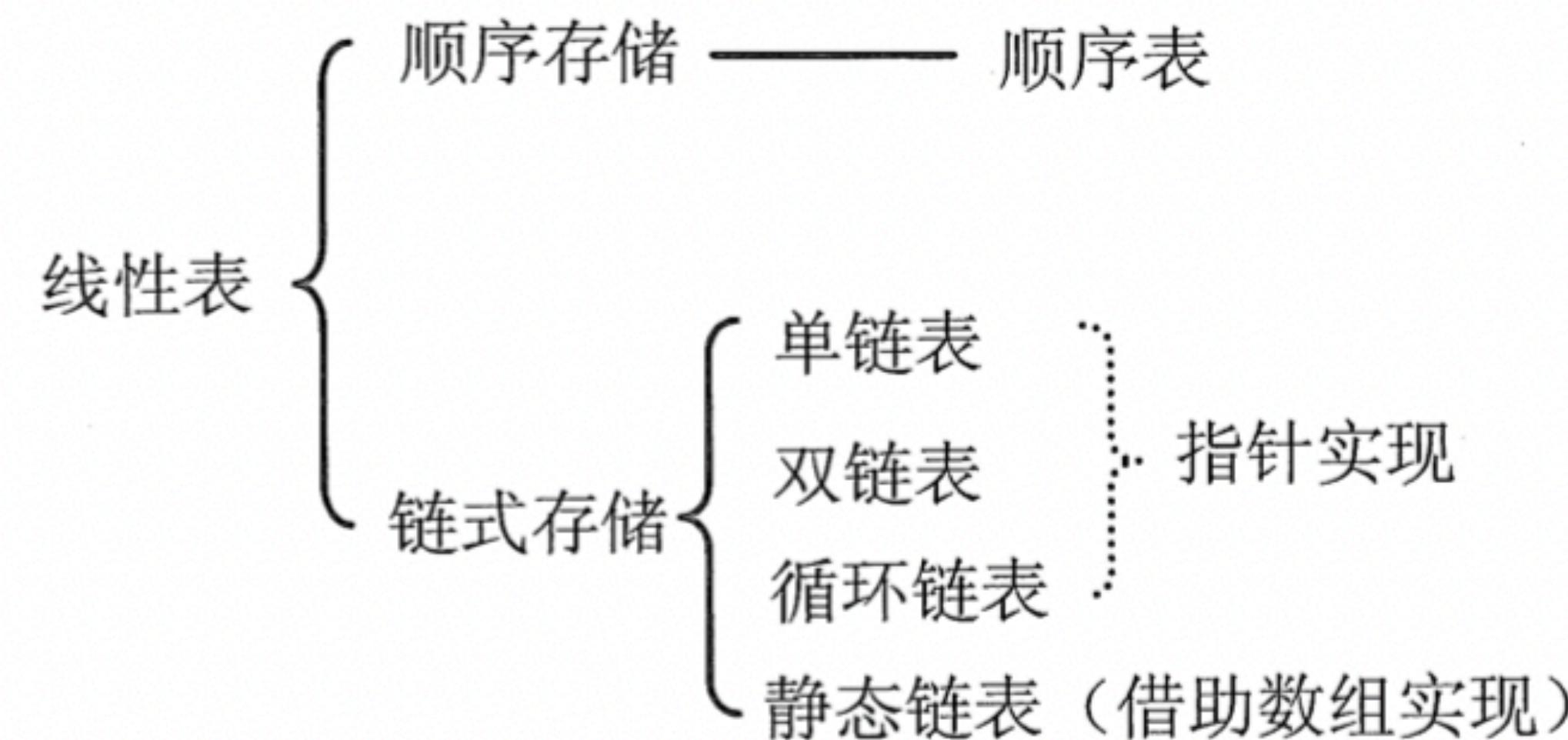
## 2、线性表

- (1) 线性表的定义、基本操作。
- (2) 线性表的实现及应用，包括顺序存储结构、链式存储结构(单链表、循环链表和双向链表)的构造原理，在两种存储结构上对线性表实施的主要的操作(三种链表的建立、插入和删除、检索等)的算法设计与实现。

## 3、栈与队列

- (1) 栈与队列的基本概念、基本操作。
- (2) 栈与队列的顺序存储结构、链式存储结构的构造原理。
- (3) 在不同存储结构的基础上对堆栈、队列实施基本操作（插入与删除等）对应的算法设计与实现。

- 1) 顺序存储。把逻辑上相邻的元素存储在物理位置上也相邻的存储单元中，元素之间的关系由存储单元的邻接关系来体现。其优点是可以实现随机存取，每个元素占用最少的存储空间；缺点是只能使用相邻的一整块存储单元，因此可能产生较多的外部碎片。
- 2) 链式存储。不要求逻辑上相邻的元素在物理位置上也相邻，借助指示元素存储地址的指针来表示元素之间的逻辑关系。其优点是不会出现碎片现象，能充分利用所有存储单元；缺点是每个元素因存储指针而占用额外的存储空间，且只能实现顺序存取。



# Array

# List ADT

An Abstract List (or List ADT) is linearly ordered data (with same data type)

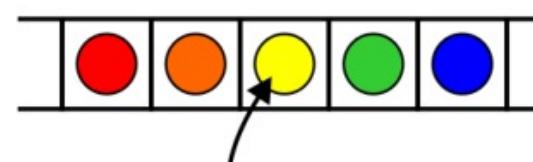
$$( \ A_1 \ A_2 \ \dots \ A_{n-1} \ A_n \ )$$

- The number of elements in the List denotes the length of the List.
- When there is no element it is an empty List.
- The beginning of a List is called the List head; the end of a List is called the List tail.
- The same value may occur more than once.

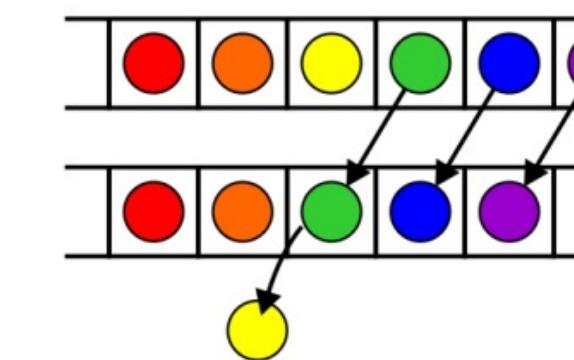
# Operations

Operations at the  $k^{\text{th}}$  entry of the list include:

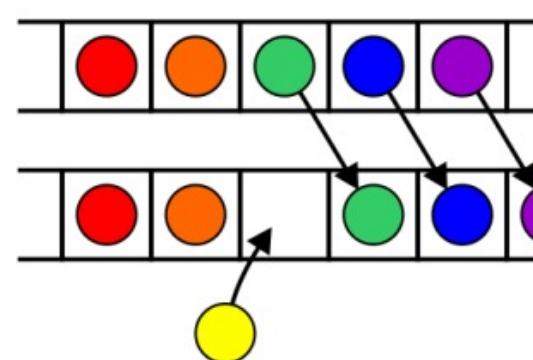
Access to the object



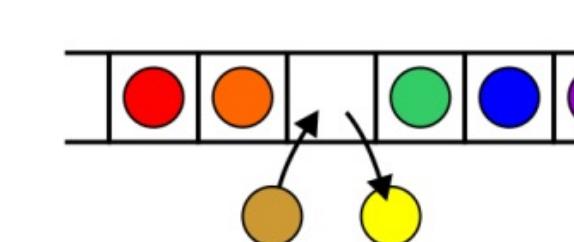
Erasing an object



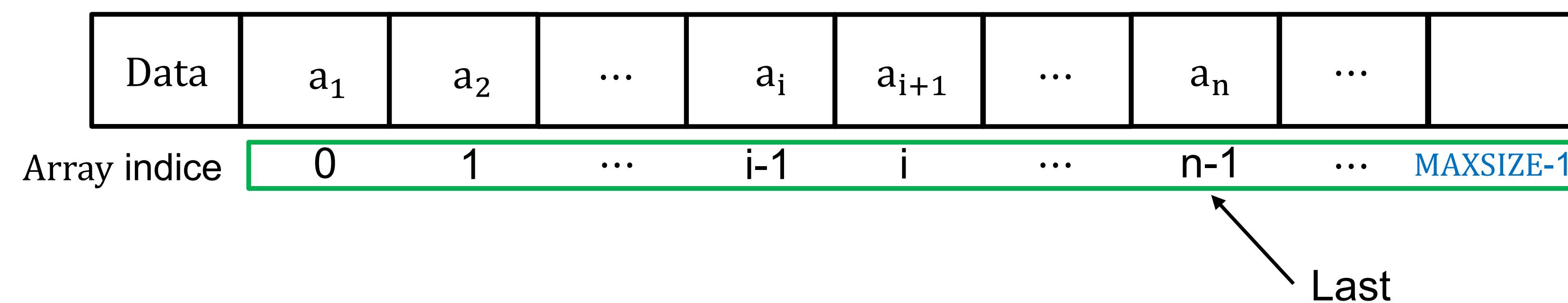
Insertion of a new object



Replacement of the object



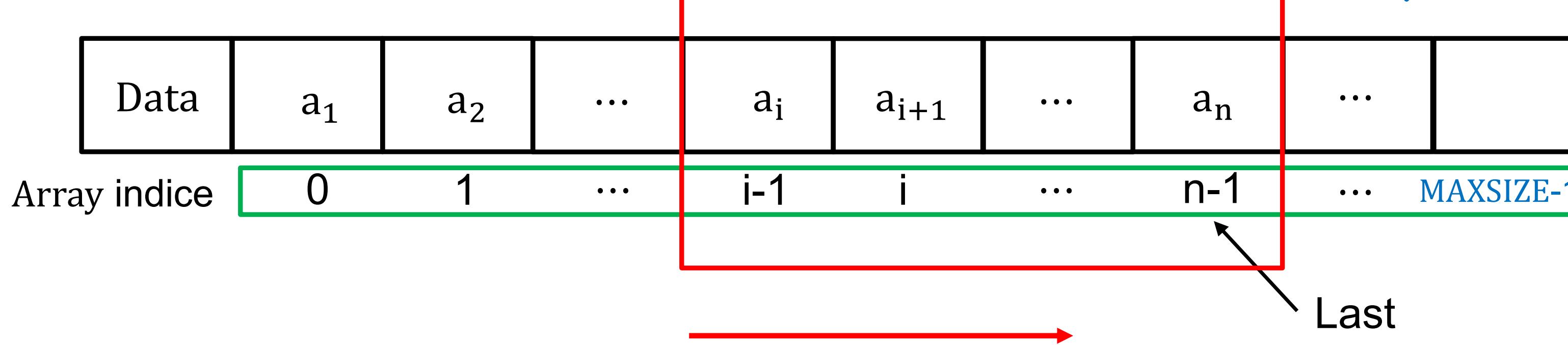
## List based on array



## List based on array

$O(n)$

- Insert element



最好情况：在表尾插入（即  $i = n + 1$ ），元素后移语句将不执行，时间复杂度为  $O(1)$ 。

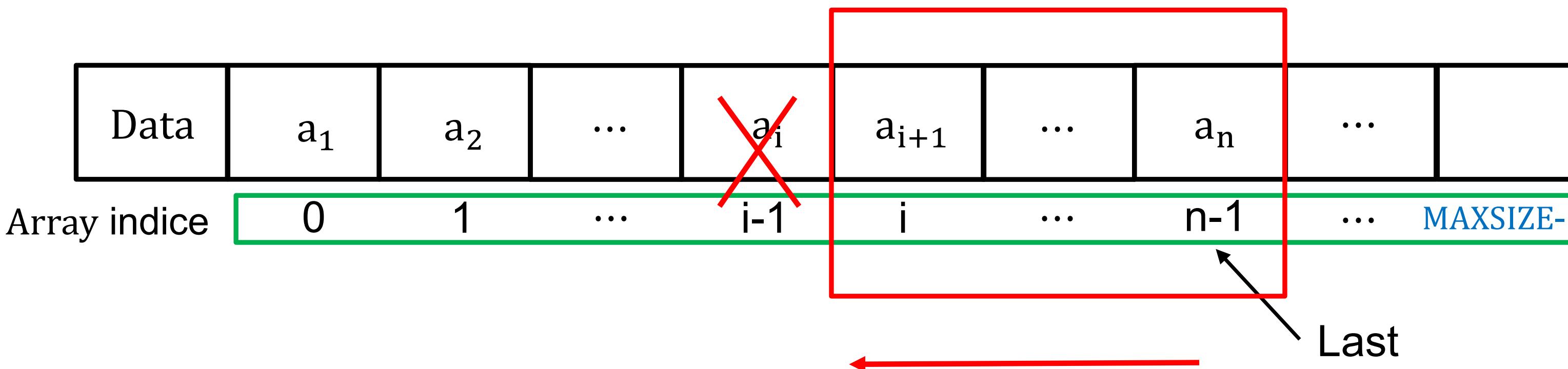
最坏情况：在表头插入（即  $i = 1$ ），元素后移语句将执行  $n$  次，时间复杂度为  $O(n)$ 。

平均情况：假设  $p_i$  ( $p_i = 1/(n+1)$ ) 是在第  $i$  个位置上插入一个结点的概率，则在长度为  $n$  的线性表中插入一个结点时，所需移动结点的平均次数为

$$\sum_{i=1}^{n+1} p_i (n-i+1) = \sum_{i=1}^{n+1} \frac{1}{n+1} (n-i+1) = \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{1}{n+1} \frac{n(n+1)}{2} = \frac{n}{2}$$

因此，线性表插入算法的平均时间复杂度为  $O(n)$ 。

- **Delete element**



最好情况：删除表尾元素（即  $i = n$ ），无须移动元素，时间复杂度为  $O(1)$ 。

最坏情况：删除表头元素（即  $i = 1$ ），需移动除第一个元素外的所有元素，时间复杂度为  $O(n)$ 。

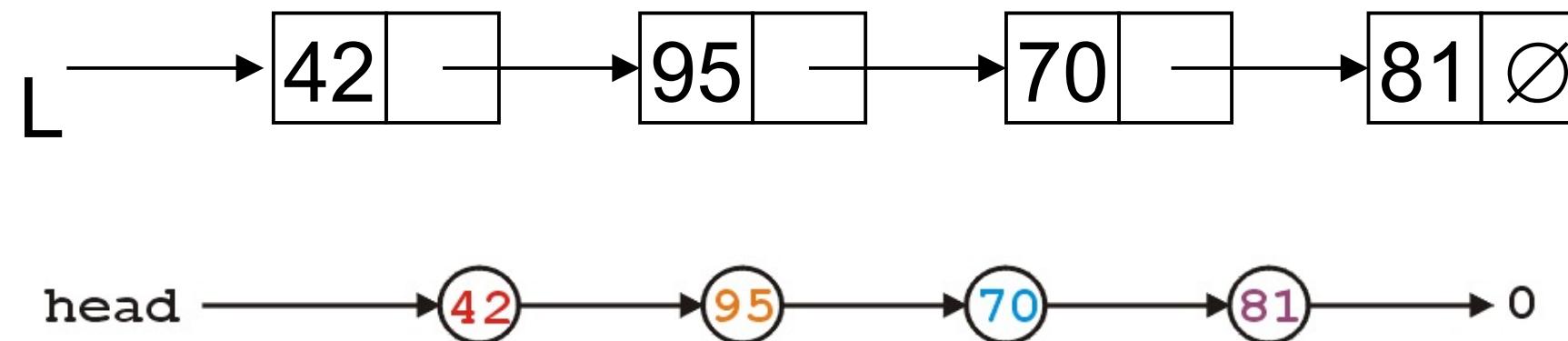
平均情况：假设  $p_i$  ( $p_i = 1/n$ ) 是删除第  $i$  个位置上结点的概率，则在长度为  $n$  的线性表中删除一个结点时，所需移动结点的平均次数为

$$\sum_{i=1}^n p_i (n-i) = \sum_{i=1}^n \frac{1}{n} (n-i) = \frac{1}{n} \sum_{i=1}^n (n-i) = \frac{1}{n} \frac{n(n-1)}{2} = \frac{n-1}{2}$$

因此，线性表删除算法的平均时间复杂度为  $O(n)$ 。

# List

# Node Class



The node must store **data** and a **pointer**:

```
class Node {  
private:  
    int element;  
    Node *next_node;  
public:  
    Node( int = 0, Node * = nullptr );  
  
    int retrieve() const;  
    Node *next() const;  
};
```

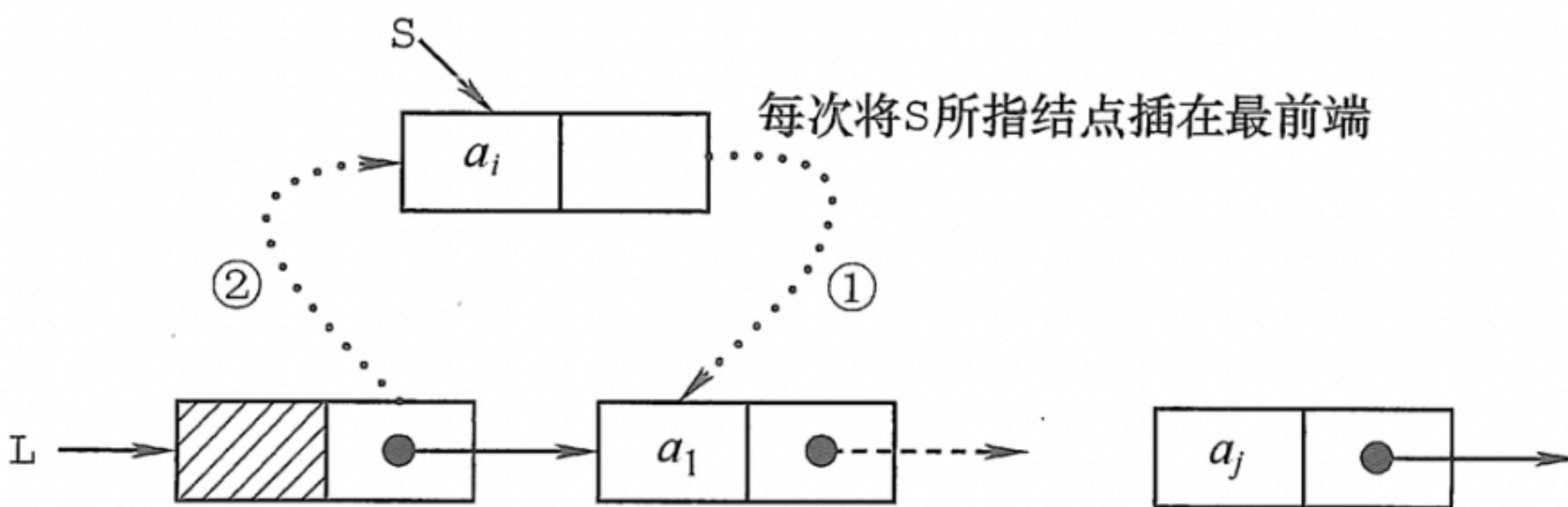


图 2.5 头插法建立单链表

头插法建立单链表的算法如下：

```

LinkList List_HeadInsert(LinkList &L) { //逆向建立单链表
    LNode *s; int x;
    L=(LinkList)malloc(sizeof(LNode)); //创建头结点
    L->next=NULL; //初始为空链表
    scanf("%d", &x); //输入结点的值
    while(x!=9999) { //输入 9999 表示结束
        s=(LNode*)malloc(sizeof(LNode)); //创建新结点①
        s->data=x;
        s->next=L->next;
        L->next=s; //将新结点插入表中, L 为头指针
        scanf("%d", &x);
    }
    return L;
}

```

采用头插法建立单链表时，读入数据的顺序与生成的链表中的元素的顺序是相反的。每个结点插入的时间为  $O(1)$ ，设单链表长为  $n$ ，则总时间复杂度为  $O(n)$ 。

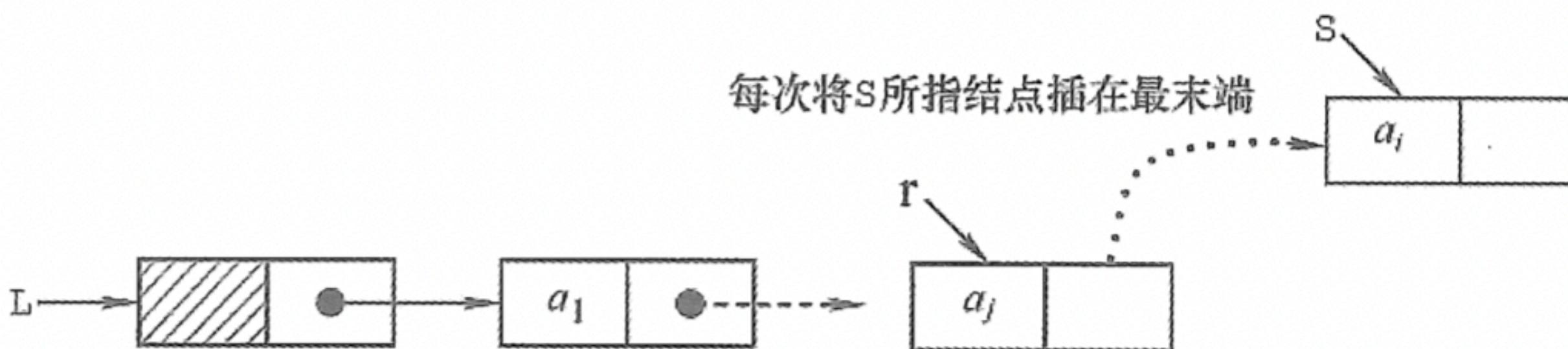


图 2.6 尾插法建立单链表

尾插法建立单链表的算法如下：

```

LinkList List_TailInsert(LinkList &L) { //正向建立单链表
    int x;                                //设元素类型为整型
    L=(LinkList)malloc(sizeof(LNode));
    LNode *s, *r=L;                      //r 为表尾指针
    scanf("%d", &x);                    //输入结点的值
    while(x!=9999) {                     //输入 9999 表示结束
        s=(LNode *)malloc(sizeof(LNode));
        s->data=x;
        r->next=s;
        r=s;                            //r 指向新的表尾结点
        scanf("%d", &x);
    }
    r->next=NULL;                      //尾结点指针置空
    return L;
}

```

### 3. 按序号查找结点值

在单链表中从第一个结点出发，顺指针 next 域逐个往下搜索，直到找到第  $i$  个结点为止，否则返回最后一个结点指针域 NULL。

按序号查找结点值的算法如下：

```
LNode *GetElem(LinkList L, int i) {
    int j=1;                      //计数，初始为 1
    LNode *p=L->next;            //头结点指针赋给 p
    if(i==0)
        return L;                  //若 i 等于 0，则返回头结点
    if(i<1)
        return NULL;                //若 i 无效，则返回 NULL
    while(p&&j<i){              //从第 1 个结点开始找，查找第 i 个结点
        p=p->next;
        j++;
    }
    return p;                      //返回第 i 个结点的指针，若 i 大于表长则返回 NULL
}
```

按序号查找操作的时间复杂度为  $O(n)$ 。

#### 4. 按值查找表结点

从单链表的第一个结点开始，由前往后依次比较表中各结点数据域的值，若某结点数据域的值等于给定值  $e$ ，则返回该结点的指针；若整个单链表中没有这样的结点，则返回 NULL。

按值查找表结点的算法如下：

```
LNode *LocateElem(LinkList L, ElemtType e) {  
    LNode *p=L->next;  
    while(p!=NULL&&p->data!=e) //从第 1 个结点开始查找 data 域为 e 的结点  
        p=p->next;  
    return p; //找到后返回该结点指针，否则返回 NULL  
}
```

按值查找操作的时间复杂度为  $O(n)$ 。

## 5. 插入结点操作

实现插入结点的代码片段如下：

```
①p=GetElem(L, i-1);          //查找插入位置的前驱结点  
②s->next=p->next;          //图 2.7 中操作步骤 1  
③p->next=s;                //图 2.7 中操作步骤 2
```

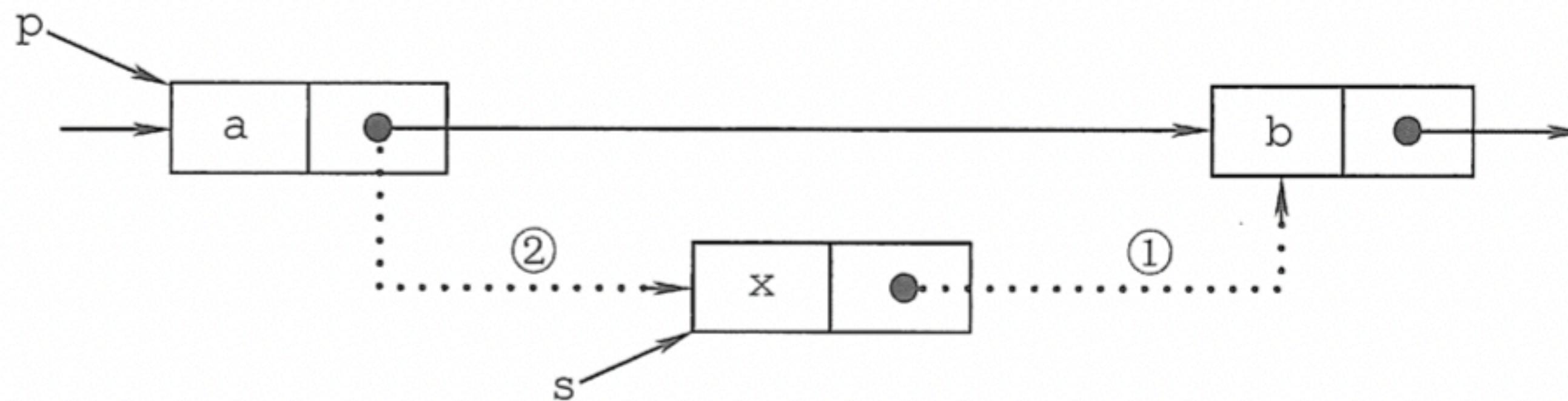


图 2.7 单链表的插入操作

算法中，语句②和③的顺序不能颠倒，否则，当先执行  $p \rightarrow \text{next} = s$  后，指向其原后继的指针就不存在，再执行  $s \rightarrow \text{next} = p \rightarrow \text{next}$  时，相当于执行了  $s \rightarrow \text{next} = s$ ，显然是错误的。本算法主要的时间开销在于查找第  $i-1$  个元素，时间复杂度为  $O(n)$ 。若在给定的结点后面插入新结点，则时间复杂度仅为  $O(1)$ 。

**扩展：对某一结点进行前插操作。**

前插操作是指在某结点的前面插入一个新结点，后插操作的定义刚好与之相反。在单链表插入算法中，通常都采用后插操作。

以上面的算法为例，首先调用函数 GetElem() 找到第  $i-1$  个结点，即插入结点的前驱结点后，再对其执行后插操作。由此可知，对结点的前插操作均可转化为后插操作，前提是单链表的头结点开始顺序查找到其前驱结点，时间复杂度为  $O(n)$ 。

此外，可采用另一种方式将其转化为后插操作来实现，设待插入结点为  $*s$ ，将  $*s$  插入到  $*p$  的前面。我们仍然将  $*s$  插入到  $*p$  的后面，然后将  $p->data$  与  $s->data$  交换，这样既满足了逻辑关系，又能使得时间复杂度为  $O(1)$ 。算法的代码片段如下：

```
//将*s 结点插入到*p 之前的主要代码片段
s->next=p->next;           //修改指针域，不能颠倒
p->next=s;
temp=p->data;                //交换数据域部分
p->data=s->data;
s->data=temp;
```

## 6. 删除结点操作

删除结点操作是将单链表的第  $i$  个结点删除。先检查删除位置的合法性，后查找表中第  $i - 1$  个结点，即被删结点的前驱结点，再将其删除。其操作过程如图 2.8 所示。

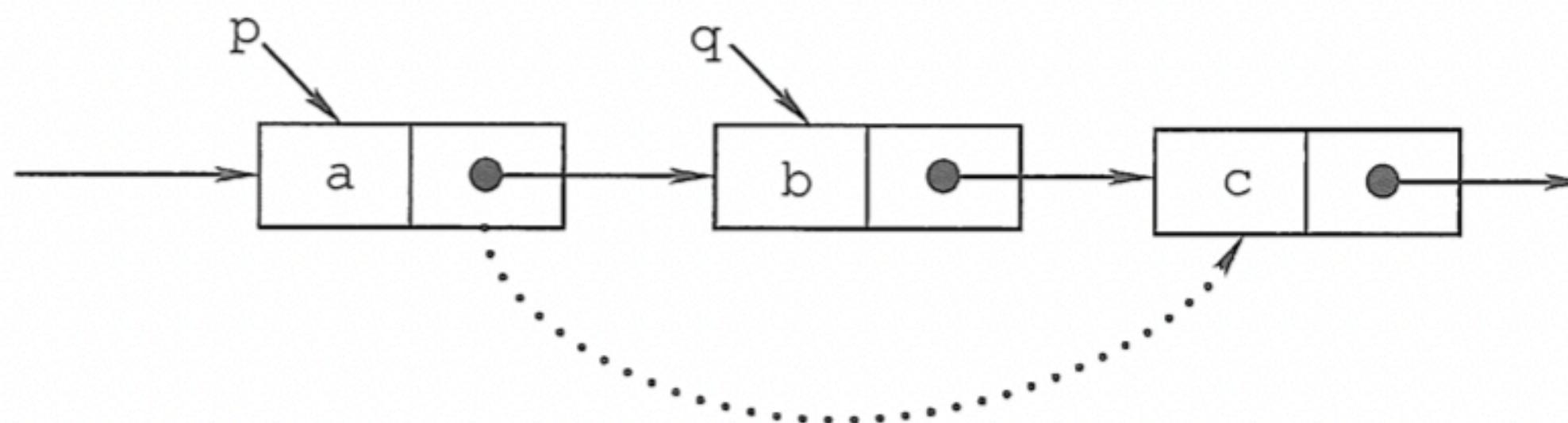


图 2.8 单链表结点的删除

实现删除结点的代码片段如下：

```
p=GetElem(L, i-1);           //查找删除位置的前驱结点  
q=p->next;                  //令 q 指向被删除结点  
p->next=q->next;           //将*q 结点从链中“断开”  
free(q);                     //释放结点的存储空间③
```

和插入算法一样，该算法的主要时间也耗费在查找操作上，时间复杂度为  $O(n)$ 。

**扩展：删除结点 $*p$ 。**

要删除某个给定结点 $*p$ ，通常的做法是先从链表的头结点开始顺序找到其前驱结点，然后再执行删除操作，算法的时间复杂度为  $O(n)$ 。

其实，删除结点 $*p$  的操作可用删除 $*p$  的后继结点操作来实现，实质就是将其后继结点的值赋予其自身，然后删除后继结点，也能使得时间复杂度为  $O(1)$ 。

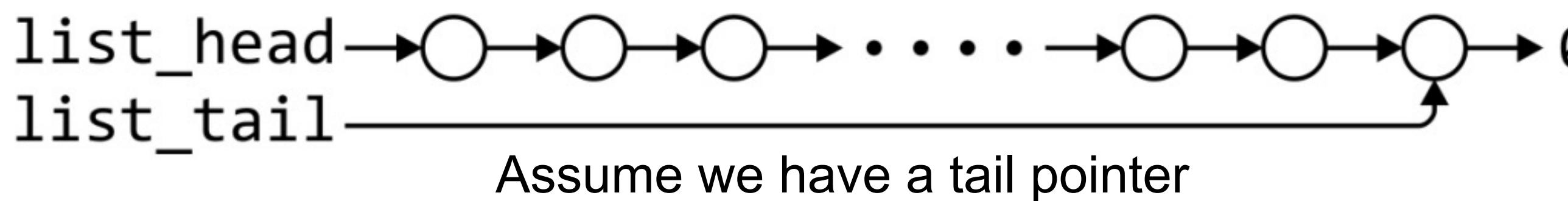
实现上述操作的代码片段如下：

```
q=p->next;           //令 q 指向*p 的后继结点  
p->data=p->next->data; //和后继结点交换数据域  
p->next=q->next;      //将*q 结点从链中“断开”  
free(q);              //释放后继结点的存储空间
```

# Linked list

	Front/1 <sup>st</sup> node	$k^{\text{th}}$ node	Back/ $n^{\text{th}}$ node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$O(n)$	$\Theta(n)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$O(n)$	$\Theta(n)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$O(n)$	$\Theta(n)$

\* These assume we have already accessed the  $k^{\text{th}}$  entry—an  $O(n)$  operation



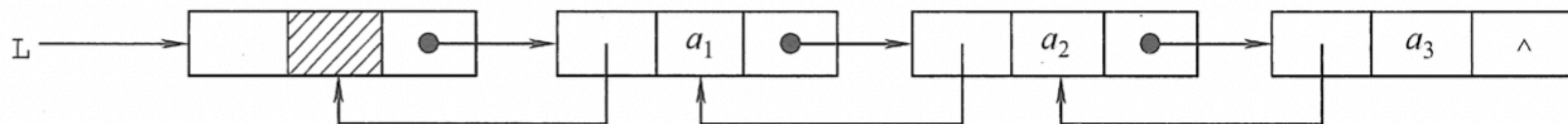


图 2.9 双链表示意图

双链表中结点类型的描述如下：

```

typedef struct DNode{
    ElemType data;           // 定义双链表结点类型
    struct DNode *prior, *next; // 数据域
} DNode, *DLinklist;        // 前驱和后继指针

```

## 1. 双链表的插入操作

在双链表中  $p$  所指的结点之后插入结点  $*s$ ，其指针的变化过程如图 2.10 所示。

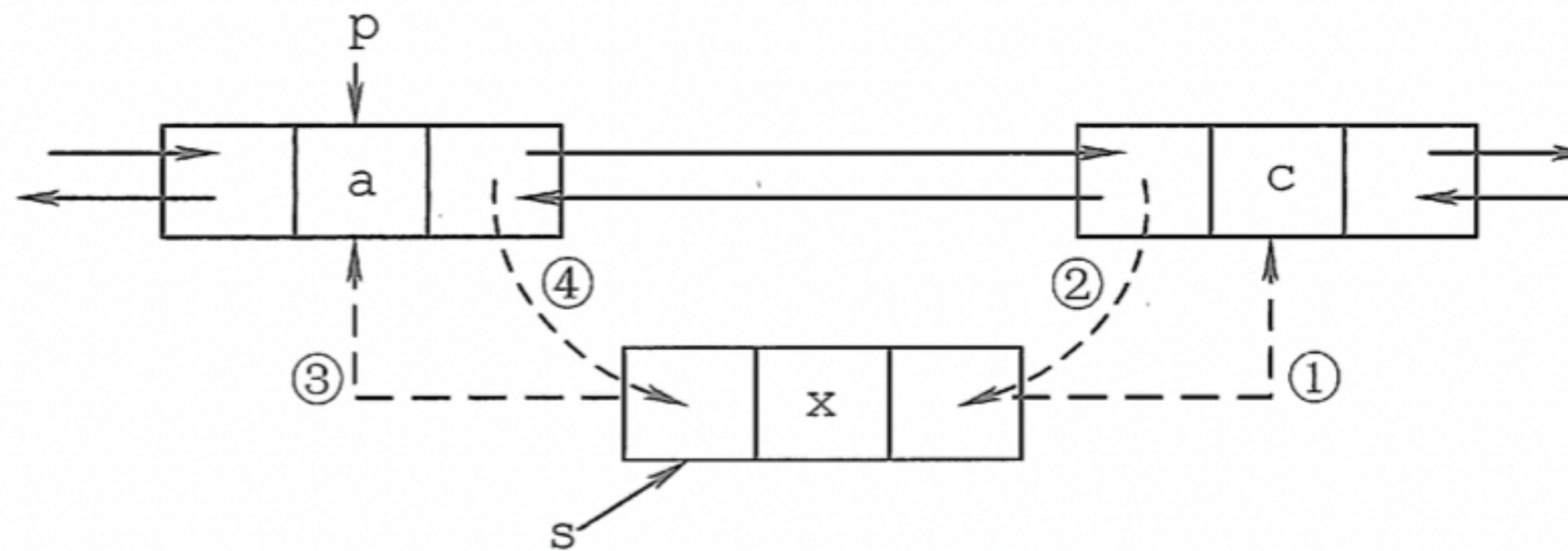


图 2.10 双链表插入结点过程

插入操作的代码片段如下：

```
①s->next=p->next;                                //将结点*s 插入到结点*p 之后  
②p->next->prior=s;  
③s->prior=p;  
④p->next=s;
```

上述代码的语句顺序不是唯一的，但也不是任意的，①和②两步必须在④步之前，否则  $*p$  的后继结点的指针就会丢掉，导致插入失败。为了加深理解，读者可以在纸上画出示意图。若问题改成要求在结点  $*p$  之前插入结点  $*s$ ，请读者思考具体的操作步骤。

## 2. 双链表的删除操作

删除双链表中结点 $*p$  的后继结点 $*q$ ，其指针的变化过程如图 2.11 所示。

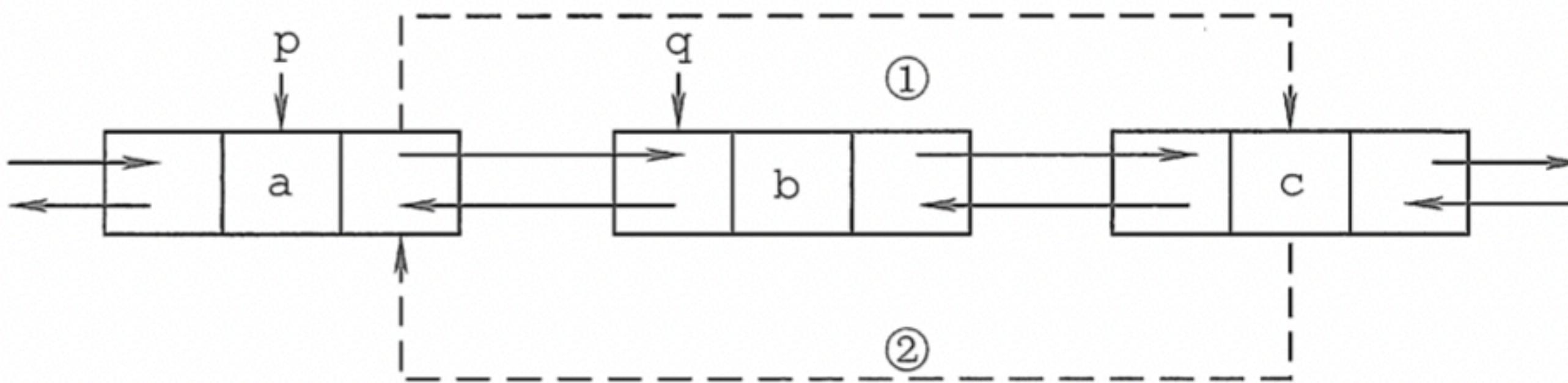


图 2.11 双链表删除结点过程

删除操作的代码片段如下：

```
p->next=q->next;           //图 2.11 中步骤①  
q->next->prior=p;          //图 2.11 中步骤②  
free(q);                    //释放结点空间
```

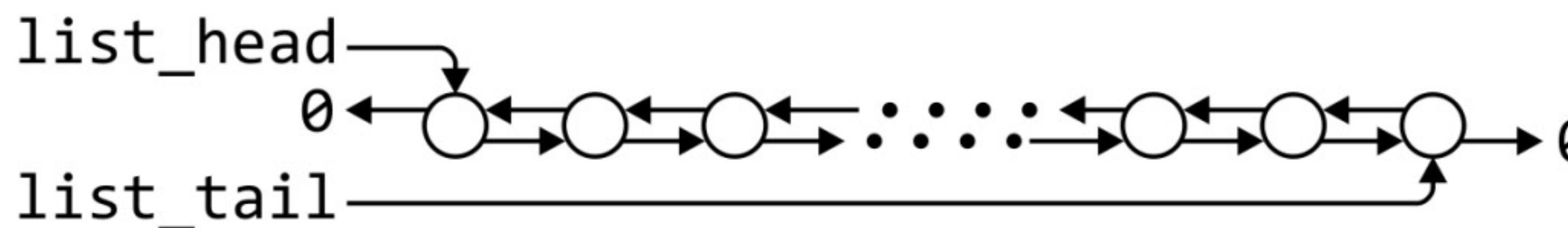
若问题改成要求删除结点 $*q$  的前驱结点 $*p$ ，请读者思考具体的操作步骤。

在建立双链表的操作中，也可采用如同单链表的头插法和尾插法，但在操作上需要注意指针的变化和单链表有所不同。

# Doubly linked lists

	Front/1 <sup>st</sup> node	$k^{\text{th}}$ node	Back/ $n^{\text{th}}$ node
Find	$\Theta(1)$	$O(n)$	$\Theta(1)$
Insert Before	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Insert After	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Replace	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
Next	$\Theta(1)$	$\Theta(1)^*$	n/a
Previous	n/a	$\Theta(1)^*$	$\Theta(1)$

\* These assume we have already accessed the  $k^{\text{th}}$  entry—an  $O(n)$  operation



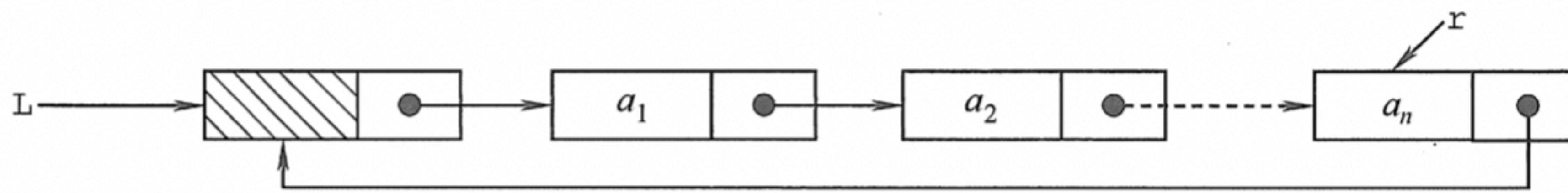


图 2.12 循环单链表

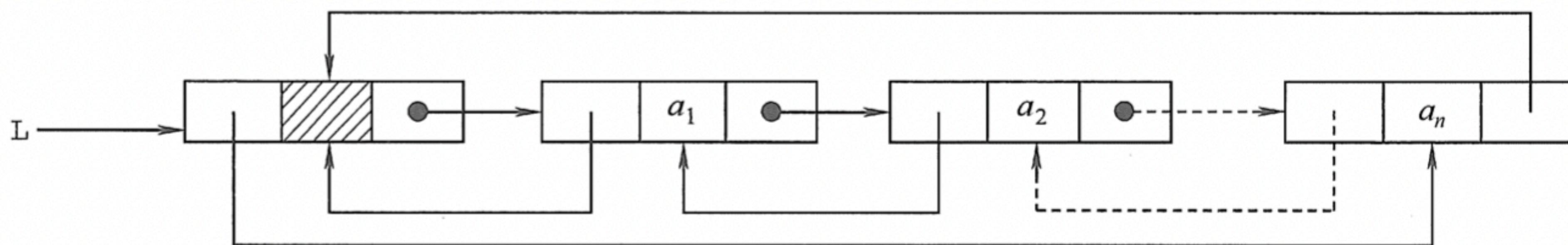


图 2.13 循环双链表

---

## I: Insert Before

---

This operation has been discussed in detail. Given a new data and the target node, create a new node using the new data and insert the new node before the target node. You should implement this function with  $O(1)$  time complexity.

- Assume the target node is neither **head** nor **tail**.
- 

```
void InsertBefore (Node *curr , int data)
{
    Node *newNode = new Node( curr->data, curr->next..... );
    .....curr->next = newNode.....;
    .....curr->data = data.....;
}
```

---

## **2: Remove Duplication**

---

Given a linked list in ascending order, you should remove all the nodes with duplicate data. Traverse the list for at most once. For example:

**Before:**

list: 1 -> 2 -> 2 -> 3 -> 3 -> NULL

**After:**

list: 1 -> 2 -> 3 -> NULL

---

# 快慢指针

## 查找链表指定节点

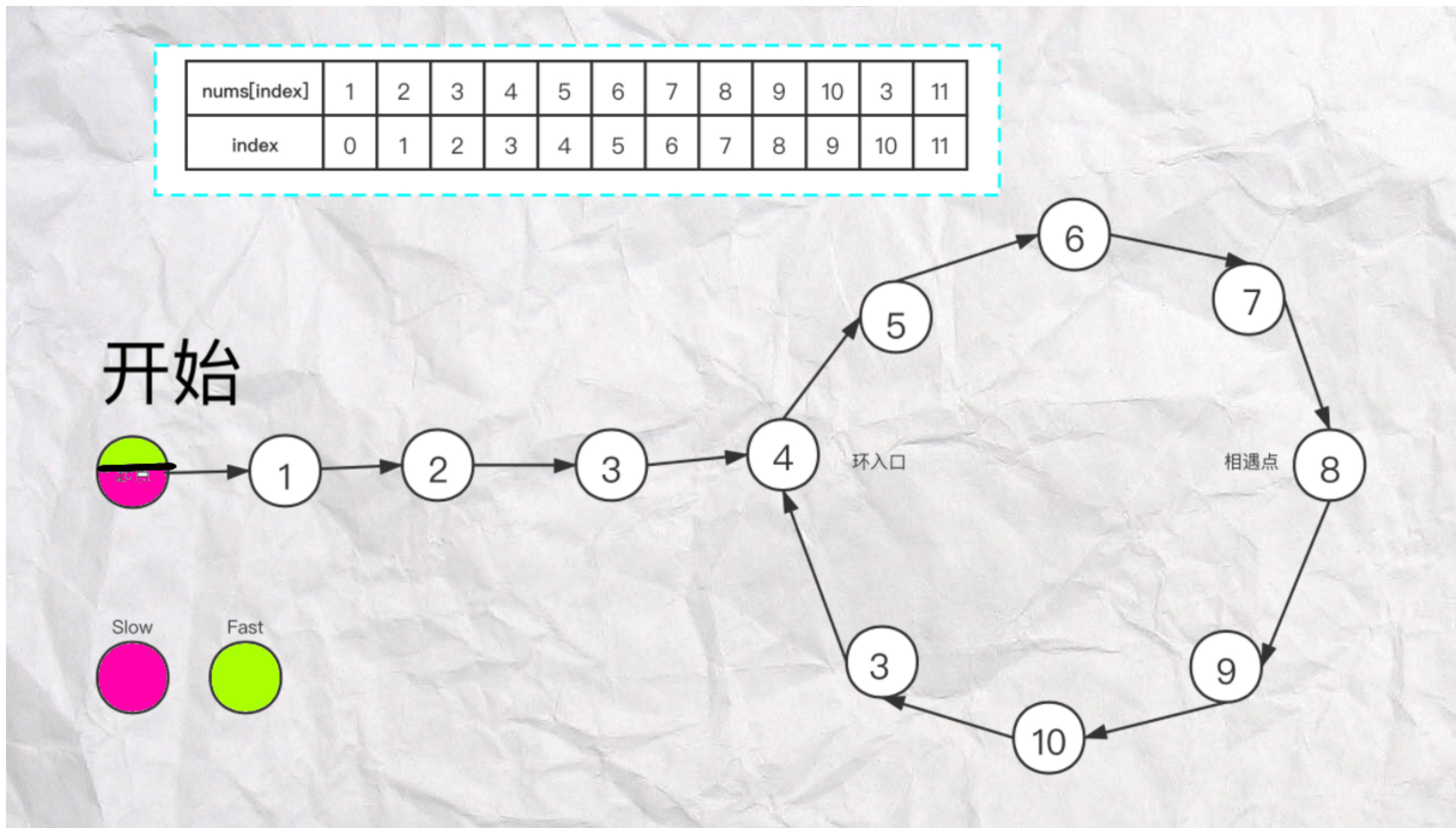
- 查找单链表中位于倒数第k个位置的节点
  - 思路：快慢指针保持固定距离
  - 快慢指针同时指向head节点。接着慢指针不动，快指针向前移动k步。之后快慢指针同速前移动。直至快指针先到终点，此时慢指针，所处的位置就是倒数第k个节点的位置。
- 查找链表中间位置的节点
  - 思路：快慢指针保持固定速度差
  - 快指针每次走两步，慢指针走一步。

```
while fast and fast.next:  
    slow = slow.next  
    fast = fast.next.next
```

# 快慢指针

## Find the Duplicate Number (leetcode 287)

- 给定一个包含  $n + 1$  个整数的数组  $\text{nums}$ ，其数字都在 1 到  $n$  之间（包括 1 和  $n$ ），可知至少存在一个重复的整数。假设  $\text{nums}$  只有一个重复的整数，找出这个重复的数。
- 你设计的解决方案必须不修改数组  $\text{nums}$  且只用常量级  $O(1)$  的额外空间。
- 示例：输入： $\text{nums} = [1,3,4,2,2]$  输出：2
- 思路：考虑每个元素的值和其下标形成的映射，能找到两个不同的索引（但是值相等）指向同一索引，即在链表中成环，环点即是重复元素。-» 快慢指针

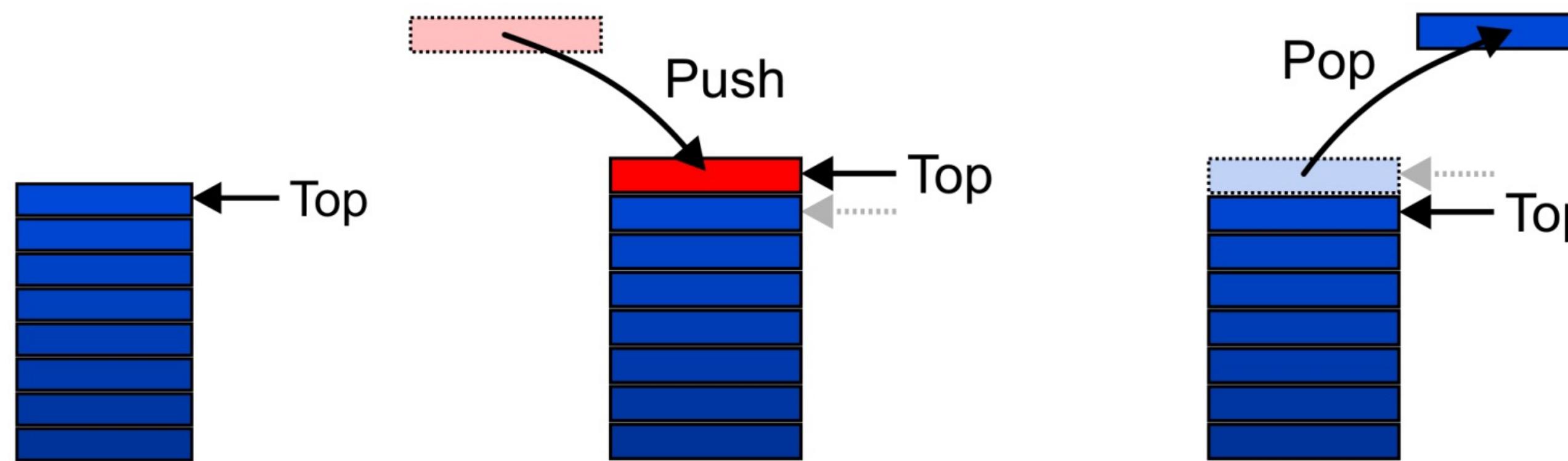


# Stack

# Stack ADT

Also called a *last-in–first-out* (LIFO) behaviour

- Graphically, we may view these operations as follows:



# Array Implementation

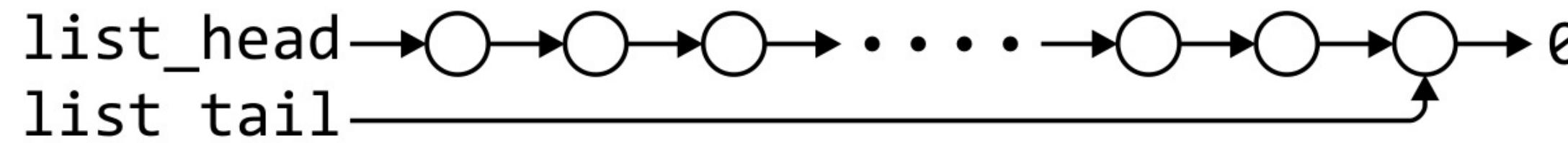
For one-ended arrays, all operations at the back are  $\Theta(1)$



	Front/1 <sup>st</sup>	Back/n <sup>th</sup>
<b>Find</b>	$\Theta(1)$	$\Theta(1)$
<b>Insert</b>	$\Theta(n)$	$\Theta(1)$
<b>Erase</b>	$\Theta(n)$	$\Theta(1)$

# Linked-List Implementation

Operations at the front of a singly linked list are all  $\Theta(1)$



	Front/1 <sup>st</sup>	Back/n <sup>th</sup>
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

The desired behavior of an Abstract Stack may be reproduced by performing all operations at the front

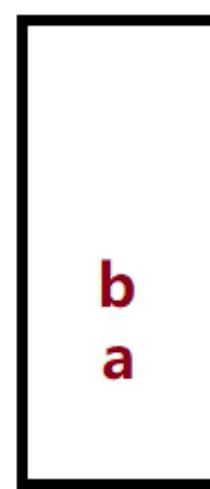
中缀表达式:  $(a + b)^*c - (a + b)/e$

后缀表达式:  $ab + c * ab + e / -$

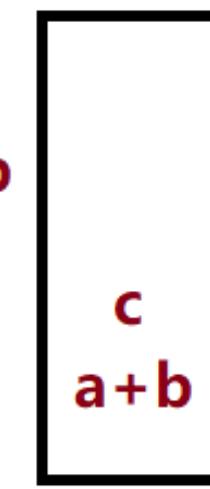
其实可以用栈来实现后缀表达式

$ab + c * ab + e / -$

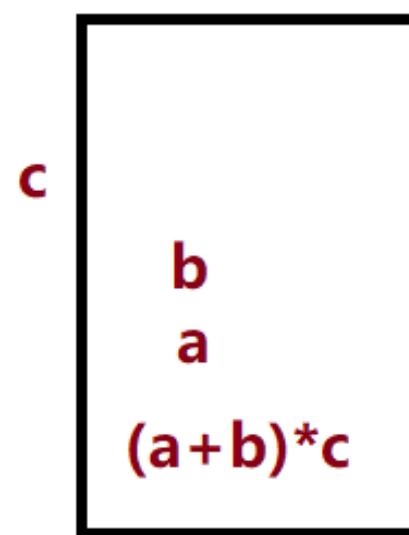
模拟过程



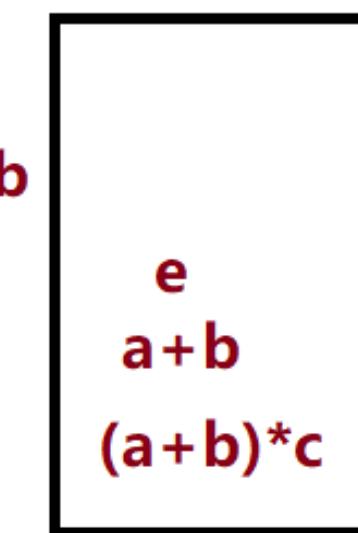
遇到+ 取出ab  
将a+b入栈  
继续向后



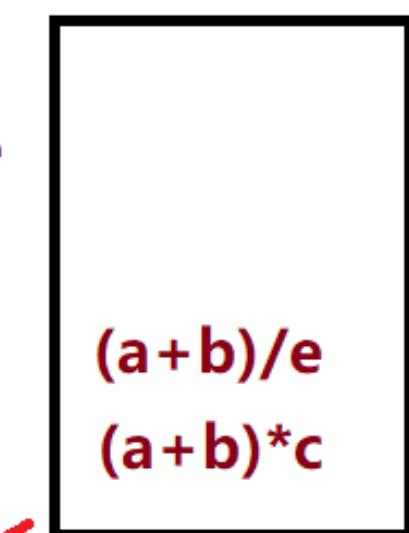
遇到\*，取出(a+b) 和 c  
将  $(a+b)^*c$  入栈  
继续向后



遇到+ 取出ab  
将a+b入栈  
继续向后



遇到/ 取出(a+b)和e  
将  $(a+b)/e$  入栈  
继续向后



$(a+b)^*c - (a+b)/e$

逆波兰表达式 ==> 后缀表达式

我们看着爽的方式

中缀表达式: 操作符位于操作数的中间 --> a + b

后缀表达式: 操作符位于操作数的后面 --> a b +

计算机操作表达式的方式

这里注意先取出的是右操作数,  
后取出的做操作数

最后遇到-

将  $(a+b)/e$  和  $(a+b)^*c$  取出  
将  $(a+b)^*c - (a+b)/e$  入栈  
显然栈顶元素即为最后的结果

## When processing an operator:

1. pop the last two items off the operand stack
2. perform the operation
3. push the result back onto the stack

[https://blog.csdn.net/weixin\\_45818891](https://blog.csdn.net/weixin_45818891)

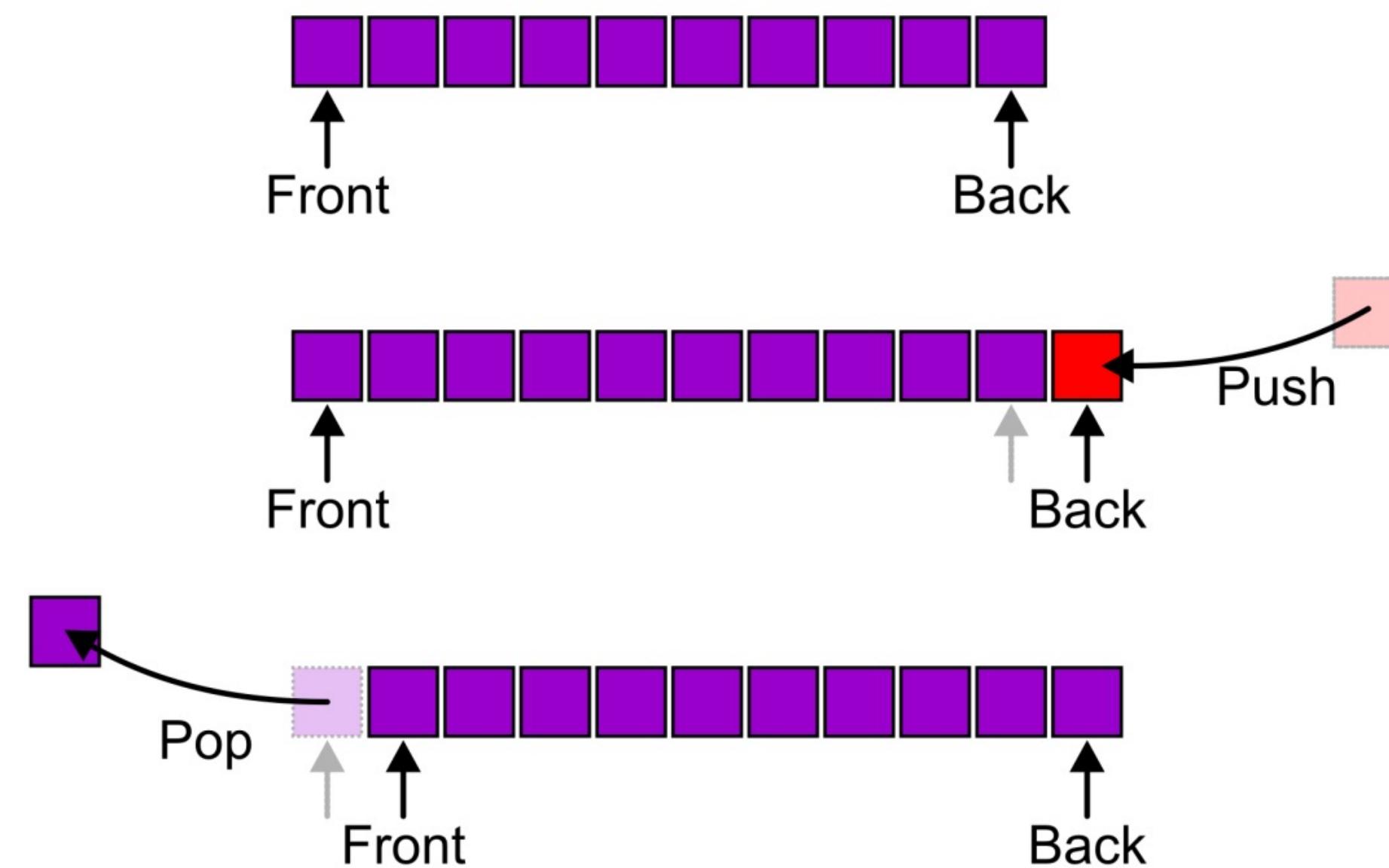
[https://blog.csdn.net/weixin\\_45818891/article/details/110009710](https://blog.csdn.net/weixin_45818891/article/details/110009710)

# Queue

# Queue ADT

Also called a *first-in–first-out* (FIFO) data structure

- Graphically, we may view these operations as follows:



# Array Implementation

A **one-ended array** does not allow all operations to occur in  $\Theta(1)$  time



	Front/1 <sup>st</sup>	Back/n <sup>th</sup>
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(n)$	$\Theta(1)$
Remove	$\Theta(n)$	$\Theta(1)$

# Array Implementation

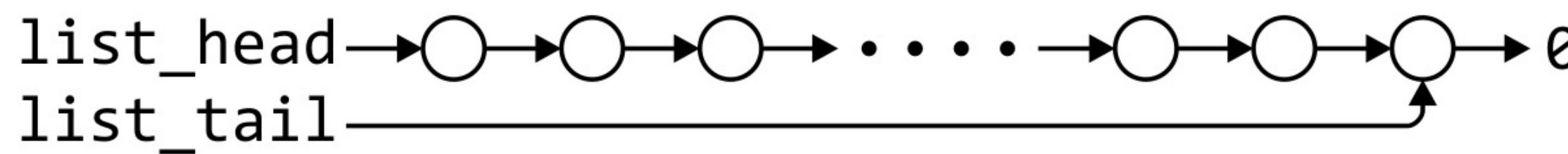
Using a **two-ended array**,  $\Theta(1)$  are possible by pushing at the back and popping from the front



	Front/1 <sup>st</sup>	Back/n <sup>th</sup>
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Remove	$\Theta(1)$	$\Theta(1)$

# Linked-List Implementation

List head/tail → Queue front/back?



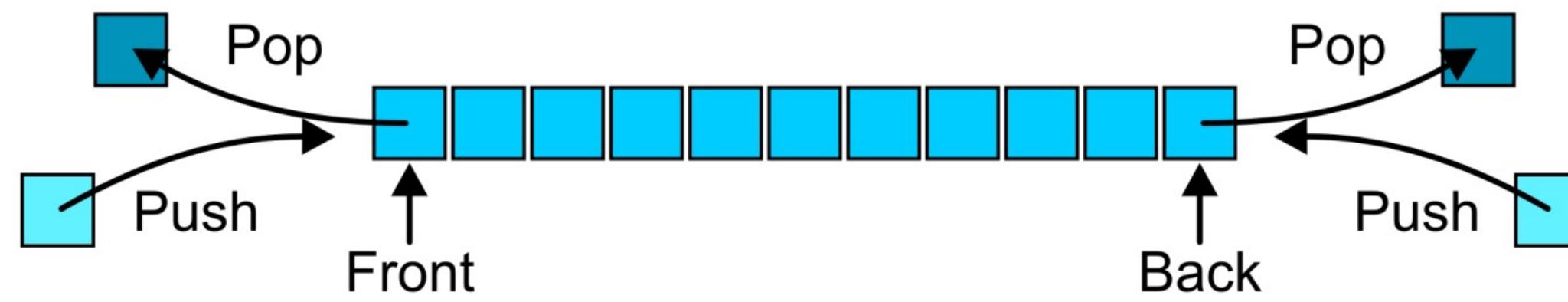
	Front/1 <sup>st</sup>	Back/n <sup>th</sup>
Find	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(1)$	$\Theta(1)$
Erase	$\Theta(1)$	$\Theta(n)$

Removal is only possible at the front with  $\Theta(1)$  run time

The desired behavior of an Abstract Queue may be produced by performing insertions at the back and removal at the front

# Deque ADT

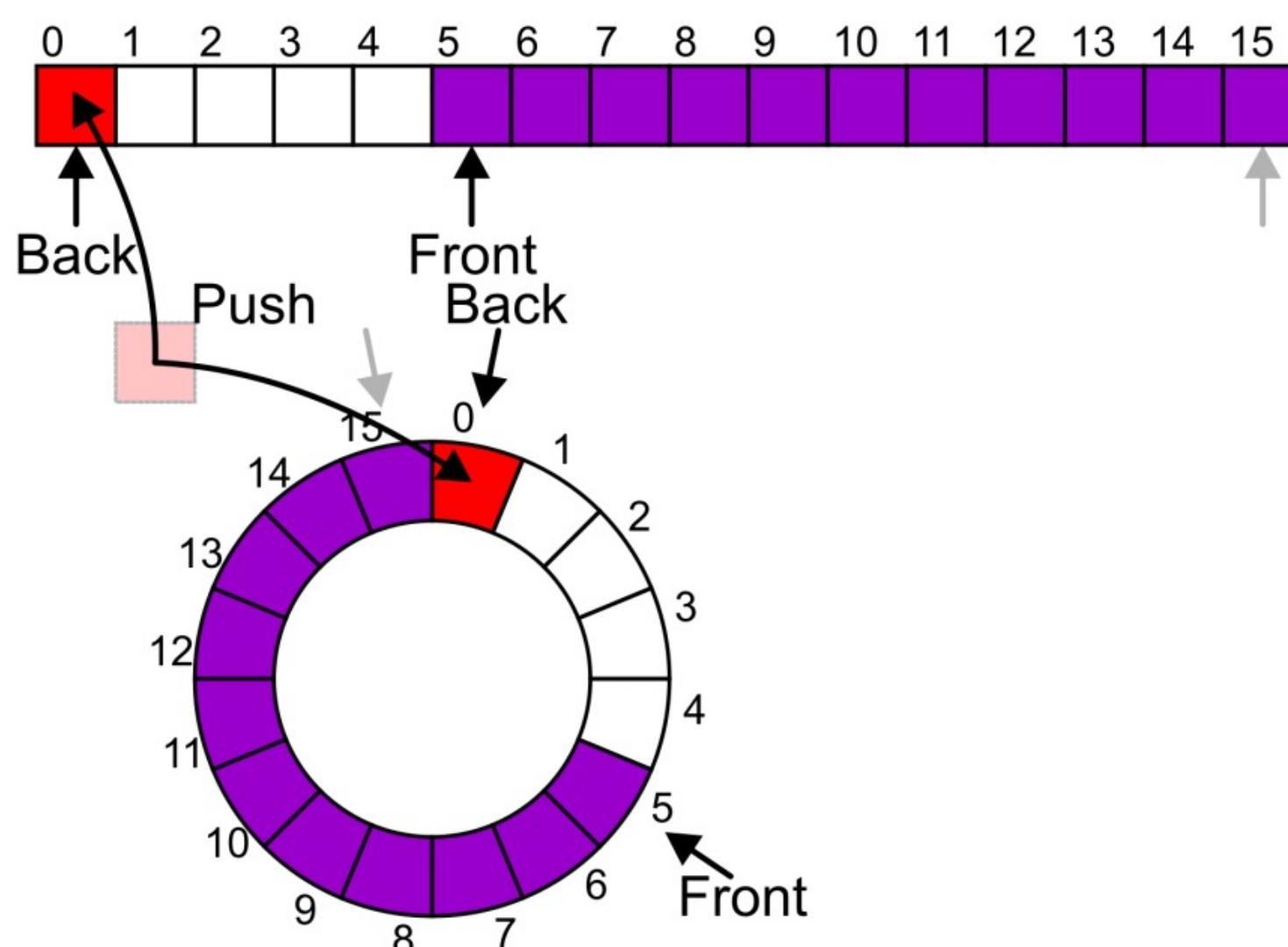
- Deque = Double-ended queue
  - pronounced like "deck"
- Uses an explicit linear ordering
- Allows insertion/removal at both the front and the back of the deque



# Member Functions

Now, the next push may be performed in the next available location of the circular array:

```
++iback;  
if ( iback == capacity() ) {  
    iback = 0;  
}
```



# Big O

We will at times use five possible descriptions

$$f(n) = o(g(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = O(g(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \Theta(g(n))$$

$$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \Omega(g(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) = \omega(g(n))$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

在分析一个程序的时间复杂性时，有以下两条规则：

a) 加法规则

$$T(n) = T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$$

b) 乘法规则

$$T(n) = T_1(n) \times T_2(n) = O(f(n)) \times O(g(n)) = O(f(n) \times g(n))$$

常见的渐近时间复杂度为

$$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

# Master Theorem

假设有递推关系式  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ ，其中  $n$  为问题规模， $a$  为递推的子问题数量， $\frac{n}{b}$  为每个子问题的规模（假设每个子问题的规模基本一样）， $f(n)$  为递推以外进行的计算工作。

$a \geq 1$ ,  $b > 1$  为常数， $f(n)$  为函数， $T(n)$  为非负整数。则有以下结果（分类讨论）：

(1) 若  $f(n) = O(n^{\log_b a - \varepsilon})$ ,  $\varepsilon > 0$ , 那么  $T(n) = \Theta(n^{\log_b a})$

(2) 若  $f(n) = \Theta(n^{\log_b a})$ , 那么  $T(n) = \Theta(n^{\log_b a} \log n)$

(3) 若  $f(n) = \Omega(n^{\log_b a + \varepsilon})$ ,  $\varepsilon > 0$ , 且对于某个常数  $c < 1$  和所有充分大的  $n$  有  $af\left(\frac{n}{b}\right) \leq cf(n)$ , 那么

$T(n) = \Theta(f(n))$ .

- $T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$

Guess  $T(n) = O(n \log n)$ :  $\exists c$ , s.t.,  $T(n) \leq cn \log n$

Substitute:

$$\begin{aligned} T(n) &\leq 2\left(c \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor\right) + n \leq cn \log \frac{n}{2} + n \\ &= cn \log n + (1 - c)n \end{aligned}$$

- $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1$

Guess  $T(n) = O(n)$ :  $\exists c, s.t., T(n) \leq cn$

Substitute:

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor + c \left\lceil \frac{n}{2} \right\rceil + 1 = cn + 1$$

FAILED

- $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1$

Guess  $T(n) = O(n)$ :  $\exists c, d$ , s.t.,  $T(n) \leq cn - d$

Substitute:

$$T(n) \leq \left(c \left\lfloor \frac{n}{2} \right\rfloor - d\right) + \left(c \left\lceil \frac{n}{2} \right\rceil - d\right) + 1 = cn - d + (1 - d)$$



## 2022互助工坊-数据结构



该二维码7天内(10月21日前)有效，重新进入将更新