第5章 栈与队列

本章学习目标

- 掌握栈、队列和双端队列的概念
- 掌握栈、队列和双端队列的数组描述
- 了解栈、队列和双端队列的链式描述

栈(Stack)与队列(Queue)是两种使用频率很高的经典的数据结构,在计算机学科的多个领域(如编译器、操作系统、计算机网络等)得到了广泛的应用。本质上,栈与队列是受限的线性表,由于它们是如此重要,因此栈与队列是独立实现的数据结构。

5.1 栈

栈是限制 add 和 remove 操作只能发生在表头或表尾的线性表。发生操作的一端叫作栈顶(Top),另一端叫作栈底(Bottom)。栈就像餐厅的一摞盘子,只能从顶部取用盘子,放回盘子也只能放回顶部。习惯上,add 操作叫作压栈(Push),remove 操作叫作出栈(Pop)。

栈具有后进先出(Last In First Out,LIFO)的特点。假设栈已有 3 个数据 A、B、C,栈底在下,栈顶在上,如图 5.1(a)所示。执行压栈后 D 进入栈,D 成为栈顶,如图 5.1(b)所示。执行出栈后 D 离开了栈,C 成为栈顶,即后压栈的数据先出栈,如图 5.1(c)所示。

图 5.1 压栈和出栈

栈常用于处理这样的问题:问题一引发了问题二,问题二引发了问题三,解决了问题三后,再返回问题二,继续处理问题二,解决了问题二后再处理问题一。计算机程序的方法调用就使用了栈,调用方法时一般将实参、返回地址、方法返回值等压栈,以保证调用者和被调用者之间交换数据,并且被调用者执行完毕后,能返回调用者。

IStack 接口定义了栈:

```
public interface IStack<T> {
    void clear();
    boolean isEmpty();
    int size();
    T peek();
    void push(T x);
    T pop();
}
```

- clear：删除栈的全部数据，使栈成为空栈。
- isEmpty：若栈为空栈，则返回真，否则返回假。
- size：返回栈的数据个数。
- peek：若栈为空栈，则返回 null，否则返回栈顶。
- push：将数据压栈，作为新的栈顶，若没有空间容纳数据，则抛出异常 StackOverflowError。
- pop：删除并返回栈顶，若栈为空，则抛出异常 NoSuchElementException。

5.1.1 栈的数组描述

栈的数组描述利用数组元素下标的次序关系表示数据压栈的先后关系，数组的右端为栈顶，如图 5.2 所示。图 5.2(a)是空栈，图 5.2(b)的 C 是栈顶，A 是栈底。变量 top 表示下一个压栈的数据应存入的位置，即栈顶的右邻。

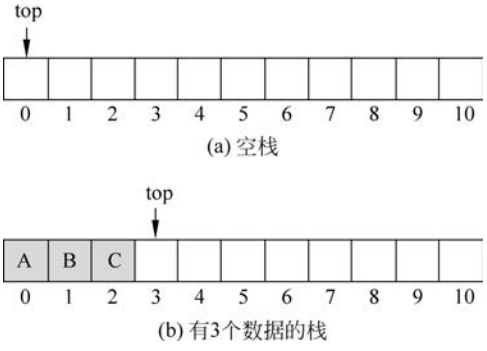

图 5.2 栈的数组描述

1. CStack 类

CStack 类是使用数组描述实现的栈。

```
1   public class CStack<T> implements IStack<T>, Iterable<T> {
2       private Object[] elements;
3       private int top;
4   }
```

数组 elements 用于存储栈的数据，字段 top 等于栈顶右邻的下标，top 也是栈的数据个数。

2. 构造器

构造器构造空栈：top＝0。

```
1   public CStack(int maxSize) {
```

```
2        elements = new Object[maxSize];
3    }
```

代码第 2 行申请了大小为 maxSize 的数组用于存放数据。

3. isEmpty 方法和 size 方法

isEmpty 方法和 size 方法使用变量 top 完成预定的操作。

```
1    public boolean isEmpty() {
2        return top == 0;
3    }
4    public int size() {
5        return top;
6    }
```

4. peek 方法

peek 方法返回栈顶，若栈为空栈，则返回 null。

栈顶存储于数组元素 elements[top－1]中，代码如下：

```
1    public T peek() {
2        return top == 0 ? null : (T) elements[top - 1];
3    }
```

5. push 方法

push 方法将数据压栈。因为 peek 方法返回 null 表示空栈，所以压栈的数据不能为 null。

```
1    public void push(T x) {
2        Objects.requireNonNull(x);
3        if (top == elements.length)
4            throw new StackOverflowError("full stack");
5        elements[top++] = x;
6    }
```

代码第 2 行判断数据 x 是否为 null，若为 null，则抛出异常 NullPointerException。第 3 行判断栈是否已满，若栈满，则抛出异常。第 5 行将数据 x 入栈并作为栈顶，同时更新 top，确保 top 是栈顶右邻的下标。

6. pop 方法

pop 方法将栈顶出栈并返回栈顶，若栈为空栈，则返回 null。

```
1    public T pop() {
2        if (top == 0)
3            throw new NoSuchElementException();
4        @SuppressWarnings("unchecked")
5        T result = (T) elements[ --top];
6        elements[top] = null;
7        return result;
8    }
```

代码第 2 行判断是否为空栈，若为空栈，则抛出异常。第 5 行取出栈顶，并从栈移除栈顶。由于 top 是栈顶的右邻，--top 的第一个作用是定位栈顶的位置，第二个作用是移除旧栈顶，明确新栈顶的下标。由于 elements[top]仍然引用已出栈的旧栈顶，因此，为了防止内存泄漏，第 6 行的语句 elements[top]＝null 切断了这个引用。

7. clear 方法

clear 方法清空栈的数据，清空后，栈的数据个数为 0，即 top=0。

如果只令 top=0，则会造成内存泄漏。如图 5.2(b)所示，因为 elements[0]、elements[1]、elements[2]仍然引用 A、B、C 对象，所以垃圾回收器无法收回 A、B、C 占用的内存空间。

```
1   public void clear() {
2       while (top != 0) {
3           elements[ -- top] = null;
4       }
5   }
```

代码第 2、3 行逐一切断栈与原有数据之间的联系，做到逻辑上清空了栈，物理上也清空了栈。

8. 迭代器

迭代器按照从栈顶到栈底的顺序返回全部数据，代码请参考本书配套资源中的 project，不再赘述。

9. 性能分析

使用数组描述实现栈的代码很简单，clear 方法的时间复杂度为 $O(n)$，其他方法的时间复杂度为 $O(1)$。所有方法的空间复杂度均为 $O(1)$。

5.1.2 栈的链式描述

栈的链式描述使用结点存储数据，利用结点之间的引用关系表示数据压栈的先后关系，这些结点构成了单向链表，如图 5.3 所示。

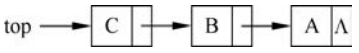

图 5.3 栈的链式描述

1. LinkedStack 类

LinkedStack 类是使用链式描述实现的栈。字段 top 引用栈顶结点，字段 size 记录栈的数据个数，用于保证 size 方法的时间复杂度为 $O(1)$。空栈的判断条件是 size==0 或 top==null，一般使用后者。

以下是 LinkedStack 的声明：

```
1   public class LinkedStack<T> implements IStack<T>, Iterable<T> {
2       Node<T> top;
3       int size;
4       static class Node<T> {
5           T data;
6           Node<T> next;
7           Node(Te, Node<T> p) {
8               data = e;
9               next = p;
10          }
11      }
12  …
13  }
```

嵌套类 Node 的字段 data 存储压栈的数据，字段 next 引用另一个 Node 对象，被引用者先于引用者压栈。

2. 构造器

构造器构造空栈：top＝null，size＝0。

```
1    public LinkedStack() {
2    }
```

LinkedStack 类其他方法的代码留作习题。

5.2 队列

队列是限制 add 操作只能发生在表尾，remove 操作只能发生在表头的线性表。习惯上，add 操作称作入队（Offer），remove 操作称为出队（Poll）。入队的一端叫作队头（Front），另一端叫作队尾（Rear）。队列就像餐厅购买食物所排的队，后来的人排在队尾，只有队头的人能获得服务，购买食物后离开队。

队列具有先进先出（First In First Out，FIFO）的特点。假设队已有 3 个数据 A、B、C，队头在左，队尾在右，如图 5.4(a)所示。执行出队后，队头 A 出队，如图 5.4(b)所示。执行入队后，D 成为队尾，如图 5.4 (c)所示。

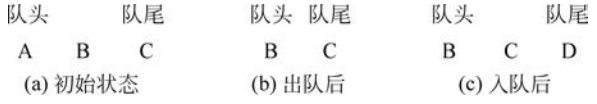

图 5.4 出队和入队

紧缺资源的分配一般采用先到先得的分配策略，队列常用于解决此类问题。操作系统分配共享打印机时会建立打印机队列，申请者需要排队。网络交换机建立数据包的接收队列和发送队列，汇聚到交换机的数据包需要排队。

IQueue 接口定义了队列：

```
public interface IQueue<T> {
    void clear();
    boolean isEmpty();
    int size();
    T peek();
    boolean offer(T x);
    T poll();
}
```

- clear：删除队列的全部数据，使之成为空队。
- isEmpty：若队列为空队，则返回真，否则返回假。
- size：返回队列的数据个数。
- peek：若为空队，则返回 null，否则返回队头。
- offer：如果不违反空间限制，则入队。数据成功入队后，返回真，否则返回假。
- poll：若为空队，则返回 null，否则队头出队，返回队头。

5.2.1　队列的数组描述

队列的数组描述利用数组元素下标的次序关系表示数据入队的先后关系，数组的左端为队头，右端为队尾，如图 5.5 所示。变量 front 表示队头的位置，rear 表示下一个入队的数据应存入的位置，即队尾的右邻。出队后，front＝front＋1，入队后，rear＝rear＋1。

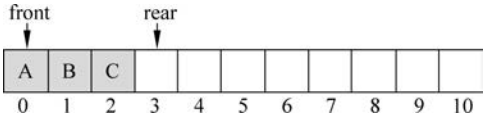

图 5.5　队列的数组描述

队列的数组描述面临假溢出问题。图 5.5 的队经过 3 次出队，8 次入队后，如图 5.6(a)所示，rear 已经越过数组的右边界，不能再存入数据，似乎已经处于队满状态，但数组的左端尚有空闲的空间可以接纳数据，实际上队列处于未满状态，这就是假溢出现象。

为了解决这个问题，引入了循环队列的概念。循环队列不再是一条直线，而是一个圆，即循环队列是首尾相连的队列。相应的，数组下标也看作是循环的，例如，图 5.6(b)的数组的下标为 0，1，2，…，10，0，即下标为 10 的数组元素的右邻是下标为 0 的数组元素。将图 5.6(a)的队列看作循环队列，入队两个数据后的结果如图 5.6(b)所示。

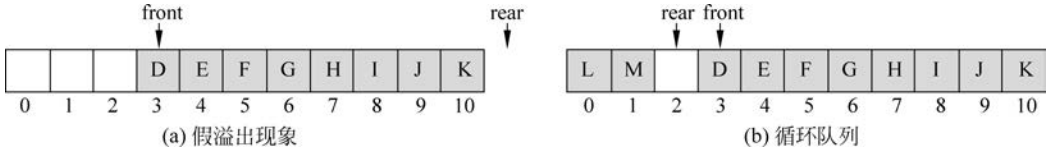

图 5.6　假溢出现象及循环队列

假设数组的长度为 m，为了实现数组下标的循环，出队后，front＝(front＋1) % m，入队后，rear＝(rear＋1) % m。

将 front 和 rear 想象成在运动场的环形跑道上进行 3000 米比赛的两个运动员，假设 rear 的实力较强，如果 rear 追上了 front，则发生了套圈现象。调换 rear 和 front 的角色，如果发生套圈现象，则是 front 追上了 rear。

队列的数组描述将(rear＋1) % m＝＝front 作为队满的条件，即如果 rear 再跑一步就追上了 front，图 5.6(b)所示的循环队列就处于队满的状态。如果 front 追上了 rear，即 front＝＝rear 成立，队就处于空队状态。例如，将图 5.5 视为循环队列，执行 3 次出队后，front 就追上了 rear，队列成为空队。

由 front 和 rear 可计算出队列的数据个数。front 和 rear 的相对位置有两种情形，第一种情形，front 在 rear 的左边，即 rear－front≥0，如图 5.5 所示，此时，数据个数 size 为：

$$\text{size}=\text{rear}-\text{front}=(\text{rear}-\text{front}+m)\ \%\ m$$

第二种情形，front 在 rear 的右边，即 rear－front≤0，如图 5.6(b)所示，此时，数据个数 size 为：

$$\text{size}=\text{rear}+m-\text{front}=(\text{rear}-\text{front}+m)\ \%\ m$$

无论哪种情形，都可以使用相同的算式计算数据个数。

1. CQueue 类

CQueue 类是使用数组描述实现的队列。

```
1   public class CQueue<T> implements IQueue<T>, Iterable<T> {
2       private Object[] elements;
3       private int front;
4       private int rear;
5       …
6   }
```

数组 elements 用于存放队列的数据，字段 front 是存储队头的数组元素的下标，字段 rear 是存储队尾的数组元素的右邻的下标。

另外，队列的数组描述要维护不变式：空队时，各数组元素为 null。

2. 构造器

构造器构造空栈。

```
1   public CQueue(int maxSize) {
2       elements = new Object[maxSize];
3   }
```

代码第 2 行创建了长度为 maxSize 的数组用于存放数据，数组 elements 的各元素取默认值 null。front 和 rear 都取默认值。条件 front==rear 成立，符合空栈的定义。

3. isEmpty 和 size 方法

isEmpty 检测队空的条件 front==rear 是否成立。size 方法根据上面推导的算式计算数据个数。

```
1   public boolean isEmpty() {
2       return front == rear;
3   }
4   public int size() {
5       return (rear - front + elements.length) % elements.length;
6   }
```

4. peek 方法

peek 方法返回队头。若队列为空队，则返回 null。

```
1   public T peek() {
2   // if (front == rear)
3   // return null;
4       return (T) elements[front];
5   }
```

代码第 4 行返回队头，队头存储于 elements[front]。前面介绍的构造器和后面介绍的 poll 和 clear 方法共同维护了不变式：空队时各数组元素为 null。空队时，无论 front 取何值，elements[front]始终为 null。因此，可省略第 2、3 行的语句。

5. offer 方法

offer 方法向队列增加数据，新增的数据作为队尾。若因为某些原因无法将数据入队，则返回 false。

```
1   public boolean offer(T x) {
2       if (x == null)
3           return false;
4       if ((rear + 1) % elements.length == front)
5           return false;
6       elements[rear] = x;
7       rear = (rear + 1) % elements.length;
8       return true;
9   }
```

因为 peek 方法返回 null 作为队空的标识，所以 null 不能作为数据入队。代码第 2 行检查入队的数据，如果为 null，则不能入队，按照约定，第 3 行返回 false。第 4 行检查队是否已满，如果队满，则数据不能入队，按照约定，第 5 行返回 false。第 6 行将数据入队，第 7 行设置 rear，使其为队尾的右邻，因数据已经入队，第 8 行返回 true。

6. poll 方法

poll 方法返回队头，并从队列移除队头。若队列是空队，则返回 null。

```
1   public T poll() {
2   // if (front == rear)
3   // return null;
4       T result = (T) elements[front];
5       if (result != null) {
6           elements[front] = null;
7           front = (front + 1) % elements.length;
8       }
9       return result;
10  }
```

代码第 4 行取出队头，若为 null，则表明是空队，第 9 行返回 null。否则，移除这个数据，第 6 行 elements[front]=null，既防止了内存泄漏，又维护了空队时各数组元素为 null 的不变式，第 7 行调整 front，从队列删除这个数据。

7. clear 方法

clear 方法清空队的数据，使队列成为空队。

```
1   public void clear() {
2       while (front != rear) {
3           elements[front] = null;
4           front = (front + 1) % elements.length;
5       }
6   }
```

代码第 2～5 行的 while 语句将各数组元素置为 null，既防止了内存泄漏，又维护了空队时各数组元素为 null 的不变式。循环结束后，条件 front==rear 成立，满足了空队的要求。

8. 迭代器

迭代器按照从 front 到 rear 的顺序返回队列的全部数据。

```
1   class Itr implements Iterator<T> {
2       int cursor;
3       Itr() {
4           cursor = front;
```

```
5        }
6        public boolean hasNext() {
7            if (cursor != rear)
8                return true;
9            return false;
10       }
11       @SuppressWarnings("unchecked")
12       public T next() {
13           T result = (T) elements[cursor];
14           cursor = (cursor + 1) % elements.length;
15           return result;
16       }
17   }
```

代码第 4 行，设 cursor 为 front 的替身，第 14 行让 cursor 追赶 rear，第 7 行判断 cursor 是否追上了 rear，若追上，则已经遍历了队列的全部数据，第 9 行返回 false，否则第 8 行返回 true。

9. 性能分析

使用数组描述实现队列的代码很简单，clear 方法的时间复杂度为 $O(n)$，其他方法的时间复杂度为 $O(1)$。所有方法的空间复杂度为 $O(1)$。

5.2.2 队列的链式描述

队列的链式描述使用结点存储数据，利用结点之间的引用关系表示数据入队的先后关系，这些结点构成了单向链表。变量 front 引用队头结点，变量 rear 引用队尾结点，如图 5.7 所示。

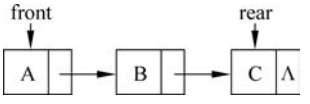

图 5.7 队列的链式描述

1. LinkedQueue 类

LinkedQueue 类是使用链式描述实现的队列。

```
1    public class LinkedQueue<T> implements IQueue<T>, Iterable<T>, Cloneable {
2        Node<T> front;
3        Node<T> rear;
4        int size;
5        static class Node<T> {
6            T data;
7            Node<T> next;
8            Node(Te, Node<T> p) {
9                data = e;
10               next = p;
11           }
12       }
13       …
14   }
```

字段 front 引用存储队头的结点，字段 rear 引用存储队尾的结点，字段 size 记录队列的数据个数，用于保证方法 size()的时间复杂度为 $O(1)$。空队的判断条件是 size==0、front==null 或 rear==null。

嵌套类 Node 的字段 data 存储入队的数据，字段 next 引用另一个 Node 对象，引用者先于被引用者入队。

2. 构造器

构造器使字段取默认值，构造了空队。

```
1    public LinkedQueue() {
2    }
```

LinkedQueue 类的其他方法的代码留作习题。

5.3 双端队列

双端队列(Double End Queue，简写为 Deque，发音与单词 Deck 相同)是左、右两端都允许入队和出队的队列，如图 5.8 所示。

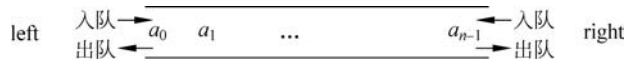

图 5.8 双端队列

左端入队的数据有先入队和后入队之分，同样，右端入队的数据也有先后之分，但左端入队的数据和右端入队的数据无先后关系。为了统一起见，规定 a_0 为双端队列的第 1 个数据，a_1 为第 2 个数据，a_{n-1} 为最后一个数据。

双端队列有两种特殊形式。一端允许入队和出队，另一端只允许出队的双端队列叫作入队受限的双端队列。一端允许入队和出队，另一端只允许入队的双端队列叫作出队受限的双端队列。

双端队列兼具栈和队列的功能，但双端队列比栈和队列更灵活。在同一端入队和出队的双端队列就是栈，一端入队，另一端出队的双端队列就是队列。有些问题需要双端队列，请见本章习题的生成滑动窗口最大值数组问题。

IDeque 接口定义了双端队列：

```
public interface IDeque<T> {
    void clear();
    boolean isEmpty();
    int size();
    boolean offerFirst(T x);
    T pollFirst();
    T peekFirst();
    boolean offerLast(T x);
    T pollLast();
    T peekLast();
}
```

- clear：清空双端队列的全部数据，使之成为空队。
- isEmpty：若双端队列为空队，则返回 true，否则返回 false。
- size：返回双端队列的数据个数。
- offerFirst：数据在左端入队。若成功入队，则返回 true，否则返回 false。
- pollFirst：左端的数据出队，并返回出队的数据。若双端队列为空队，则返回 null。
- peekFirst：返回左端的数据。若双端队列为空队，则返回 null。
- offerLast：数据在右端入队。若成功入队，则返回 true，否则返回 false。

- pollLast：右端的数据出队，并返回出队的数据。若双端队列为空队，则返回 null。
- peekLast：返回右端的数据。若双端队列为空队，则返回 null。

5.3.1 双端队列的数组描述

双端队列的数组描述利用数组元素下标的次序关系表示数据在队列中的顺序，如图 5.9 所示。变量 left 表示左端，right 表示右端，数据的顺序是从 left 到 right。左端入队，执行 left＝left－1 后，数据存入数组元素 left，左端出队，执行 left＝left＋1。右端入队，数据存入数组元素 right，然后执行 right＝right＋1，右端出队，执行 right＝right－1。

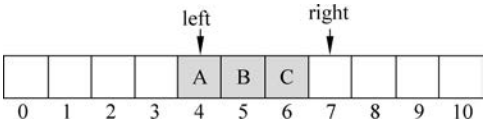

图 5.9 双端队列的数组描述

双端队列的数组描述与队列的数组描述同样面临假溢出问题，解决方法也是将双端队列视为循环队列，即 left＝left＋1，right＝right＋1 后，如果 left、right 大于数组的右边界，则令 left＝0，right＝0。left＝left－1，right＝right－1 后，如果 left、right 小于 0，则令 left、right 等于数组的右边界。图 5.10 是图 5.9 的双端队列在右端入队 5 个数据后的情形。

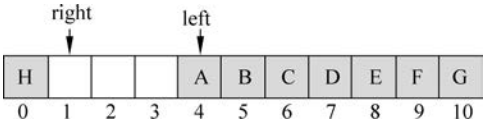

图 5.10 循环双端队列

同队列的数组描述相似，双端队列的数组描述使用 left 和 right 计算队列的数据个数。

若双端队列的数据个数等于 0，则为空队。若双端队列的数据个数等于数组长度，则为满队列。由于变量 left 和变量 right 的设置方式，也可以这样判断空队和满队列：左端或右端出队时，若 left＝＝right 成立，则双端队列为空队；左端或右端入队后，若 left＝＝right，则双端队列为满队列。

若双端队列用完了存储空间成为满队列，则立刻扩大存储空间，即双端队列处于满队列是一个临时状态。

1. CDeque 类

CDeque 类是使用数组描述实现的双端队列。数组 elements 用于存放队列的数据，字段 left 记录了左端出队和入队的位置，字段 right 记录了右端出队和入队的位置。

```
1   public class CDeque<T> implements IDeque<T>, Iterable<T> {
2       private Object[] elements;
3       private int left;
4       private int right;
5       …
6   }
```

双端队列的数组描述维护了不变式：空队时各数组元素为 null。

2. 构造器

构造器构造空队。变量 left 和变量 right 等于 0，双端队列的数据个数等于 0。各数组元素为 null，满足了不变式。

```
1   public CDeque(int maxSize) {
2       elements = new Object[maxSize];
3   }
```

3. 辅助方法 dec 和 inc

dec 方法用于完成 left－1 和 right－1，若结果小于 0，则 left、right 等于数组的右边界。

```
1   private int dec(int i) {
2       if ( -- i < 0)
3           i = elements.length - 1;
4       return i;
5   }
```

inc 方法用于完成 left＋1 和 right＋1，若结果超过数组的右边界，则 left、right 等于 0。

```
1   private int inc(int i) {
2       if (++i >= elements.length)
3           i = 0;
4       return i;
5   }
```

4. size 方法

若 right≥left，size＝right－left，如图 5.9 所示，代码第 2 行用于计算这种情况的数据个数。若 right < left，如图 5.10 所示，数据分布在两部分，第一部分的数据个数为 right，第二部分的数据个数为 element.length－left，代码第 4 行针对这种情况。

```
1   public int size() {
2       int i = right - left;
3       if (i < 0)
4           i += elements.length;
5       return i;
6   }
```

5. offerFirst 方法

offerFirst 方法使数据在左端入队。若成功入队，则返回 true，否则返回 false。

```
1   public boolean offerFirst(T x) {
2       Objects.requireNonNull(x);
3       elements[left = dec(left)] = x;
4       if (left == right)
5           doubleCapacity();
6       return true;
7   }
```

入队的位置是 left－1，第 3 行将数据存入这个位置。第 4 行测试条件 left＝＝right 是否成立，若成立，则处于满队列状态，已耗尽了所有存储空间，第 5 行扩大存储空间，如果无法从系统获得更大的空间，则抛出异常，否则，由于数据已经入队，因此第 6 行返回 true。

6. offerLast 方法

offerLast 方法使数据在右端入队。若成功入队，则返回 true，否则返回 false。

right 就是入队的位置，入队后要将 right 设置为新入队数据的右邻，参见图 5.9。同样，若用尽了存储空间，则扩大存储空间。

```
1   public boolean offerLast(T x) {
2       Objects.requireNonNull(x);
3       elements[right] = x;
4       if ((right = inc(right)) == left)
5           doubleCapacity();
6       return true;
7   }
```

7. pollFirst 方法

pollFirst 方法使左端的数据出队，并返回出队的数据。若队列为空队，则返回 null。

```
1   public T pollFirst() {
2       T result = (T) elements[left];
3       if (result != null) {
4           elements[left] = null;
5           left = inc(left);
6       }
7       return result;
8   }
```

代码第 2 行获取出队的数据，若出队的数据不为 null，为了维护不变式，第 4 行将左端的数据设置为 null，第 5 行更新 left，完成出队。第 7 行，按照接口的约定，返回左端的数据。

8. pollLast 方法

pollLast 方法使右端的数据出队，并返回出队的数据。若队列为空队，则返回 null。代码与 pollFirst 方法相似。

```
1   public T pollLast() {
2       int t;
3       T result = (T) elements[t = dec(right)];
4       if (result != null) {
5           elements[right = t] = null;
6       }
7       return result;
8   }
```

9. peekFirst 方法和 peekLast 方法

peekFirst 方法和 peekLast 方法分别返回双端队列的队头和队尾，若双端队列为空队，则返回 null。

```
1   public T peekFirst() {
2       return (T) elements[left];
3   }
4   public T peekLast() {
5       return (T) elements[dec(right)];
6   }
```

队头和队尾在数组的下标分别是 left 和 right－1，第 2 行和第 5 行分别取出队头和队尾并返回。若双端队列是空队，由于构造器、出队、清空等方法维护了不变式，因此返回 null。

10. clear 方法

数据在数组的存放有两种情形：一个连续区间，如图 5.9 所示；两个连续区间，如图 5.10 所示。clear 方法的一种实现方法是让 left 追赶 right，一边追赶，一边设置数据为 null，但是，需要使用%运算。clear 方法的另一个实现方法是分别清除两个连续区间，从而避免使用%运算。

```
1   public void clear() {
2       for (int i = left, to = (i <= right) ? right : elements.length;; i = 0, to = right) {
3           for (; i < to; left++)
4               elements[i] = null;
5           if (to == right)
6               break;
7       }
8       left = right = 0;
9   }
```

代码第 2 行的 for 语句会循环一次或两次。若初始设置 i＝left，to＝right，即 left≤right，如图 5.9 所示，第 3、4 行清空这个区间，第 6 行结束循环。若初始设置 i＝left，to＝elements.length，即 left＞right，如图 5.10 所示，则第一次循环，第 3、4 行清除图 5.9 右边的区间，第二次循环，i＝0，to＝right，第 3、4 行清除图 5.9 左边的区间。

11. doubleCapacity 方法

左端入队或右端入队后，队列可能会处于满队列的状态，条件 left＝＝right 成立，如图 5.11 所示。此时，需要调用 doubleCapacity 方法扩大空间。

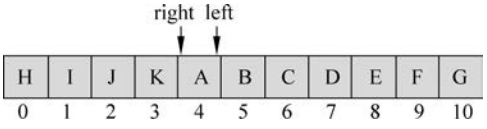

图 5.11 满队列示意图

```
1   private void doubleCapacity() {
2       assert left == right;
3       int n = elements.length;
4       int newCapacity = n << 1;
5       if (newCapacity < 0)
6           throw new IllegalStateException("Sorry, deque too big");
7       Object[] a = new Object[newCapacity];
8       int p = left;
9       int r = n - p;
10      System.arraycopy(elements, p, a, 0, r);
11      System.arraycopy(elements, 0, a, r, p);
12      elements = a;
13      left = 0;
14      right = n;
15  }
```

代码第 4 行设置新空间的大小是原空间的 2 倍。如果原空间很大，再扩大 2 倍可能超

出 int 类型的表示范围，发生溢出，即 newCapacity 为负数，则第 6 行抛出异常。第 9 行的变量 r 是从 left 到数组右边界的数据个数，第 10 行将原空间的 left 到数组右边界的数据复制到新空间，占用的位置是 0，1，…，r－1，第 11 行将原空间从 0 到 left－1 的数据复制到新空间，占用的位置是 r，r＋1，…，r＋p－1。队列在新空间的第一个数据的位置是 0，最后一个数据的位置是 n－1。

12. 性能分析

基于数组描述的双端队列的代码很简单，clear 方法和 doubleCapacity 方法的时间复杂度为 $O(n)$，其他方法的时间复杂度为 $O(1)$。所有方法的空间复杂度为 $O(1)$。

5.3.2 双端队列的链式描述

双端队列的链式描述利用结点之间的引用关系表示数据在队列中的顺序，这些结点构成了单向链表，变量 left 引用队头结点，变量 right 引用队尾结点，如图 5.12 所示。

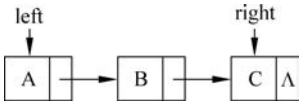

图 5.12 双端队列的链式描述

双端队列的链式描述留作练习。

小结

栈和队列是经典的数据结构，用途广泛。双端队列兼具栈和队列的功能，并且更加灵活。

Java 类库无 Stack 接口，但有 Queue 接口和 Deque 接口，Deque 接口扩展了 Queue 接口。Deque 接口包含栈的 push、pop 操作，这两个操作发生在双端队列的左端，等同于 offerFirst 和 pollFirst。Stack 类是栈的一个早期实现，已经过时。ArrayDeque 类和 LinkedList 类实现了 Deque 接口，前者基于数组描述，后者基于链式描述。

习题

1. 选择题

(1) 一个队列的入队序列是 1，2，3，4，则出队序列是(　　)。

A. 4，3，2，1　　B. 1，2，3，4

C. 1，4，3，2　　D. 3，2，4，1

(2) 队列的操作有(　　)。

A. 对队列中的数据排序

B. 取出最近入队的数据

C. 在队列的数据之间插入新的数据

D. 删除队头的数据

(3) 为了解决计算机与打印机之间的速度不匹配问题，通常设置一个打印数据缓冲区，主机将要输出的数据一次写入该缓冲区，而打印机则依次从该缓冲区中取出数据。该缓冲区的逻辑结构应该是(　　)。

A. 栈　　B. 队列　　C. 树　　D. 图

(4) 如果队列允许在其两端进行入队，但仅允许在一端进行出队。若数据元素 a，b，c，d，e 依次入队后再出队，则不可能得到的出队序列是(　　)。

A. b，a，c，d，e　　B. d，b，a，c，e

C. d，b，c，a，e　　D. e，c，b，a，d

(5) 最适合用作链式描述的队列的链表是(　　)。

A. 带队头指针和队尾指针的循环单向链表

B. 带队头指针和队尾指针的非循环单向链表

C. 只带队头指针的非循环单向链表

D. 只带队头指针的循环单向链表

(6) 最不适合用作链式描述的队列的链表是(　　)。

A. 只带队头指针的非循环双链表　　B. 只带队头指针的循环双链表

C. 只带队尾指针的循环双链表　　D. 只带队尾指针的循环单向链表

(7) 若用数组 A[6]实现循环队列，且当前 rear 和 front 分别为 1 和 5，当从队列中删除一个数据元素，再加入两个数据元素时，rear 和 front 分别为(　　)。

A. 3 和 4　　B. 3 和 0　　C. 5 和 0　　D. 5 和 1

(8) 最大容量为 n 的循环队列，队尾指针为 rear，队头指针为 front，则队空的条件为(　　)。

A. (rear+1) % n==front　　B. rear==front

C. rear+1==front　　D. (rear−1) % n==front

(9) 现有队列 Q 和栈 S，初始时 Q 中的数据依次为 1，2，3，4，5，6(1 在队头)，S 为空。若仅允许下列 3 种操作：出队并输出队头；出队并将队头压栈；出栈并输出栈顶，则不能得到的输出序列是(　　)。

A. 1，2，5，6，4，3　　B. 2，3，4，5，6，1

C. 3，4，5，6，1，2　　D. 6，5，4，3，2，1

(10) 设栈 S 和队列 Q 的初始状态为空，数据 e_1，e_2，e_3，e_4，e_5，e_6 依次通过栈 S，一个数据出栈后即进队列 Q，若 6 个数据出队的序列为 e_2，e_4，e_3，e_6，e_5，e_1，则栈的容量至少为(　　)。

A. 6　　B. 4　　C. 3　　D. 2

2. 填空题

(1) 队列是一种受限的线性表，允许插入的一端称为_____，允许删除的一端称为_____，对队列的访问是按照_____的原则进行的。

(2) 使用链式描述实现队列时，需要两个引用变量分别引用_____和_____。

(3) 在具有 n 个数据的非空队列插入或删除一个数据的时间复杂度为_____。

(4) 引入循环队列是为了克服_____。

(5) 区分循环队列的满与空，有_____、_____和_____方法。

3. 应用题

(1) 实现链式描述的栈 LinkStack,需要实现 IStack 接口。

(2) 实现链式描述的队列 LinkQueue,需要实现 IQueue 接口。

(3) 为类 LinkQueue 增加方法 LinkQueue split(),将奇数位置的数据保留在原队列,偶数位置的数据放到新队列返回。

(4) 为类 CQueue 增加方法 T[] toArray(),按照队头到队尾的次序将队列中的数据存放到数组 T[],即队头放在 T[0],以此类推,队尾放在 T[$n-1$],n 是队列中的数据的个数。

(5) 先执行 1×10^6 次 offer 操作,然后执行 1×10^6 次 poll 操作。测试链式描述队列和数组描述实现的队列的性能。

(6) 实现链式描述的双端队列 LinkedDeque,需要实现 IDeque 接口。

(7) 生成滑动窗口最大值数组问题。有一个整型数组和一个大小为 w 的窗口从数组的最左边滑动到最右边,窗口每次向右滑动一个位置,编写代码输出各窗口的最大值。

例如,数组为 5,4,6,5,3,1,6,2,窗口大小为 3,用符号[]表示窗口,则各窗口的数据及最大值如下:

[5,4,**6**],5,3,1,6,2

5,[4,**6**,5],3,1,6,2

5,4,[**6**,5,3],1,6,2

5,4,6,[**5**,3,1],6,2

5,4,6,5,[3,1,**6**],2

5,4,6,5,3,[1,**6**,2]

输出结果为 6,6,6,5,6,6。提示,使用输入受限的双端队列。

(8) 编写一个类,用两个栈实现队列,需要实现 IQueue 接口。