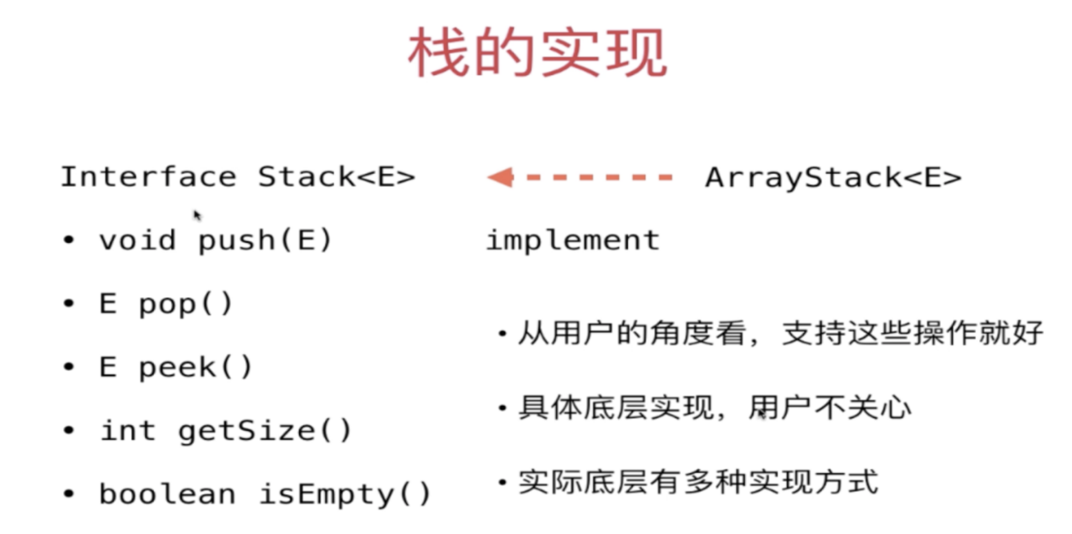
数据结构与算法5-栈和队列的实现

笔记本：	我的笔记		
创建时间：	2020/10/3 8:09	**更新时间：**	2020/10/3 10:05
作者：	liuhouer		
标签：	算法		



栈的实现

Interface Stack<E> ◀- - - - - - - ArrayStack<E>

implement

- void push(E)
- E pop()
- E peek()
- int getSize()
- boolean isEmpty()

- 从用户的角度看，支持这些操作就好
- 具体底层实现，用户不关心
- 实际底层有多种实现方式

1.用数组实现栈

1>数组类：

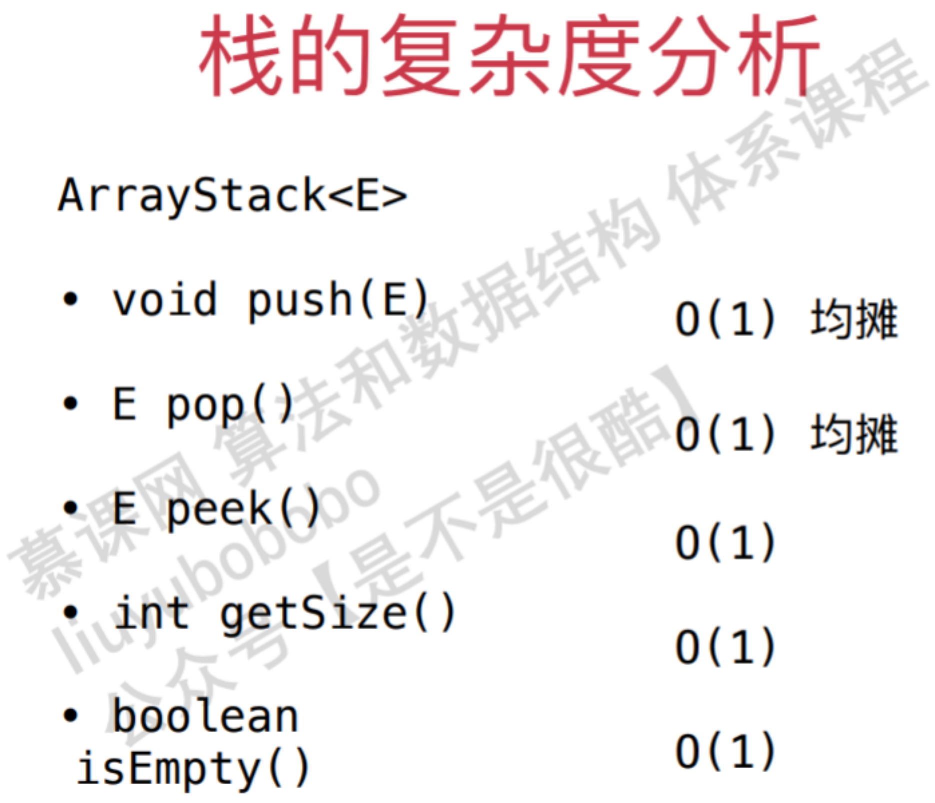
```
public class Array<E> {


    private E[] data;
    private int size;


    // 构造函数，传入数组的容量capacity构造Array
    public Array(int capacity){
        data = (E[])new Object[capacity];
        size = 0;
    }


    // 无参数的构造函数，默认数组的容量capacity=10
    public Array(){
        this(10);
    }


    // 获取数组的容量
    public int getCapacity(){
        return data.length;
    }


    // 获取数组中的元素个数
    public int getSize(){
        return size;
    }


    // 返回数组是否为空
    public boolean isEmpty(){
        return size == 0;
    }


    // 在index索引的位置插入一个新元素e
    public void add(int index, E e){


        if(index < 0 || index > size)
            throw new IllegalArgumentException("Add failed. Require index >= 0
and index <= size.");


        if(size == data.length)
            resize(2 * data.length);


        for(int i = size - 1; i >= index ; i --)
            data[i + 1] = data[i];


        data[index] = e;


        size ++;
    }


    // 向所有元素后添加一个新元素
    public void addLast(E e){
        add(size, e);
    }


    // 在所有元素前添加一个新元素
    public void addFirst(E e){
        add(0, e);
    }


    // 获取index索引位置的元素
    public E get(int index){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Get failed. Index is
illegal.");
        return data[index];
    }


    public E getLast(){
        return get(size - 1);
    }


    public E getFirst(){
        return get(0);
    }


    // 修改index索引位置的元素为e
    public void set(int index, E e){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Set failed. Index is
illegal.");
        data[index] = e;
    }


    // 查找数组中是否有元素e
    public boolean contains(E e){
        for(int i = 0 ; i < size ; i ++){
            if(data[i].equals(e))
                return true;
        }
        return false;
    }


    // 查找数组中元素e所在的索引，如果不存在元素e，则返回-1
    public int find(E e){
        for(int i = 0 ; i < size ; i ++){
            if(data[i].equals(e))
                return i;
        }
        return -1;
    }


    // 从数组中删除index位置的元素, 返回删除的元素
    public E remove(int index){
        if(index < 0 || index >= size)
            throw new IllegalArgumentException("Remove failed. Index is
illegal.");


        E ret = data[index];
        for(int i = index + 1 ; i < size ; i ++)
            data[i - 1] = data[i];
        size --;
        data[size] = null; // loitering objects != memory leak


        if(size == data.length / 4 && data.length / 2 != 0)
            resize(data.length / 2);
        return ret;
    }


    // 从数组中删除第一个元素, 返回删除的元素
    public E removeFirst(){
        return remove(0);
    }


    // 从数组中删除最后一个元素, 返回删除的元素
    public E removeLast(){
        return remove(size - 1);
    }


    // 从数组中删除元素e
    public void removeElement(E e){
        int index = find(e);
        if(index != -1)
            remove(index);
    }


    @Override
    public String toString(){


        StringBuilder res = new StringBuilder();
        res.append(String.format("Array: size = %d , capacity = %d\n", size,
data.length));
        res.append('[');
        for(int i = 0 ; i < size ; i ++){
            res.append(data[i]);
            if(i != size - 1)
                res.append(", ");
        }
        res.append(']');
        return res.toString();
    }
    // 将数组空间的容量变成newCapacity大小
    private void resize(int newCapacity){

        E[] newData = (E[])new Object[newCapacity];
        for(int i = 0 ; i < size ; i ++)
            newData[i] = data[i];
        data = newData;
    }
}
```

2>栈接口：

```
public interface Stack<E> {

    int getSize();
    boolean isEmpty();
    void push(E e);
    E pop();
    E peek();
}
```

3>数组实现栈：

```
public class ArrayStack<E> implements Stack<E> {

    private Array<E> array;

    public ArrayStack(int capacity){
        array = new Array<>(capacity);
    }

    public ArrayStack(){
        array = new Array<>();
    }

    @Override
    public int getSize(){
        return array.getSize();
    }

    @Override
    public boolean isEmpty(){
        return array.isEmpty();
    }

    public int getCapacity(){
        return array.getCapacity();
    }

    @Override
    public void push(E e){
        array.addLast(e);
    }

    @Override
    public E pop(){
        return array.removeLast();
    }
    @Override
    public E peek(){
        return array.getLast();
    }


    @Override
    public String toString(){
        StringBuilder res = new StringBuilder();
        res.append("Stack: ");
        res.append('[');
        for(int i = 0 ; i < array.getSize() ; i ++){
            res.append(array.get(i));
            if(i != array.getSize() - 1)
                res.append(", ");
        }
        res.append("] top");
        return res.toString();
    }
}
```



栈的复杂度分析

ArrayStack<E>

• void push(E)	O(1) 均摊
• E pop()	O(1) 均摊
• E peek()	O(1)
• int getSize()	O(1)
• boolean isEmpty()	O(1)

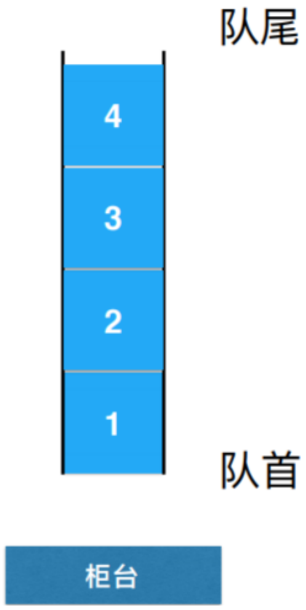
栈的应用

- undo操作 -编辑器
- 系统调用栈-操作系统
- 括号匹配-编辑器
- 计算器-表达式解析

2.用数组实现队列

队列queue

- 队列是一种先进先出的数据结构（先到先得）
- First In First Out (FIFO)

- 

队列的实现

Interface Queue <------------ ArrayQueue implement

• void enqueue(E)
• E dequeue()
• E getFront()
• int getSize()
• boolean isEmpty()

1>队列接口

```
public interface Queue<E> {


    int getSize();
    boolean isEmpty();
    void enqueue(E e);
    E dequeue();
    E getFront();
}
```

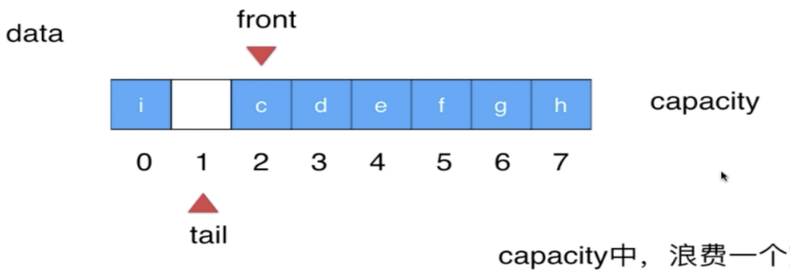
2>数组实现队列

```
public class ArrayQueue<E> implements Queue<E> {


    private Array<E> array;


    public ArrayQueue(int capacity){
        array = new Array<>(capacity);
    }


    public ArrayQueue(){
        array = new Array<>();
    }
    @Override
    public int getSize(){
        return array.getSize();
    }


    @Override
    public boolean isEmpty(){
        return array.isEmpty();
    }


    public int getCapacity(){
        return array.getCapacity();
    }


    @Override
    public void enqueue(E e){
        array.addLast(e);
    }


    @Override
    public E dequeue(){
        return array.removeFirst();
    }


    @Override
    public E getFront(){
        return array.getFirst();
    }


    @Override
    public String toString(){
        StringBuilder res = new StringBuilder();
        res.append("Queue: ");
        res.append("front [");
        for(int i = 0 ; i < array.getSize() ; i ++){
            res.append(array.get(i));
            if(i != array.getSize() - 1)
                res.append(", ");
        }
        res.append("] tail");
        return res.toString();
    }


    public static void main(String[] args) {


        ArrayQueue<Integer> queue = new ArrayQueue<>();
        for(int i = 0 ; i < 10 ; i ++){
            queue.enqueue(i);
            System.out.println(queue);
            if(i % 3 == 2){
                queue.dequeue();
                System.out.println(queue);
            }
        }
    }
}
```

循环队列：对数组队列进行优化>减小时间复杂度，减小对象的位移

循环队列

front == tail 队列为空　　　(tail + 1) % c == front 队列满



capacity中，浪费一个空间

```
public class LoopQueue<E> implements Queue<E> {


    private E[] data;
    private int front, tail;
    private int size;  // 有兴趣的同学，在完成这一章后，可以思考一下：
                       // LoopQueue中不声明size，如何完成所有的逻辑？
                       // 这个问题可能会比大家想象的要难一点点：）


    public LoopQueue(int capacity){
        data = (E[])new Object[capacity + 1];
        front = 0;
        tail = 0;
        size = 0;
    }


    public LoopQueue(){
        this(10);
    }


    public int getCapacity(){
        return data.length - 1;
    }


    @Override
    public boolean isEmpty(){
        return front == tail;
    }


    @Override
    public int getSize(){
        return size;
    }


    @Override
    public void enqueue(E e){


        if((tail + 1) % data.length == front)
            resize(getCapacity() * 2);


        data[tail] = e;
        tail = (tail + 1) % data.length;
        size ++;
    }


    @Override
    public E dequeue(){
        if(isEmpty())
            throw new IllegalArgumentException("Cannot dequeue from an empty
queue.");


        E ret = data[front];
        data[front] = null;
        front = (front + 1) % data.length;
        size --;
        if(size == getCapacity() / 4 && getCapacity() / 2 != 0)
            resize(getCapacity() / 2);
        return ret;
    }


    @Override
    public E getFront(){
        if(isEmpty())
            throw new IllegalArgumentException("Queue is empty.");
        return data[front];
    }


    private void resize(int newCapacity){


        E[] newData = (E[])new Object[newCapacity + 1];
        for(int i = 0 ; i < size ; i ++)
            newData[i] = data[(i + front) % data.length];


        data = newData;
        front = 0;
        tail = size;
    }


    @Override
    public String toString(){


        StringBuilder res = new StringBuilder();
        res.append(String.format("Queue: size = %d , capacity = %d\n", size,
getCapacity()));
        res.append("front [");
        for(int i = front ; i != tail ; i = (i + 1) % data.length){
            res.append(data[i]);
            if((i + 1) % data.length != tail)
                res.append(", ");
        }
        res.append("] tail");
        return res.toString();
    }


    public static void main(String[] args){


        LoopQueue<Integer> queue = new LoopQueue<>();
        for(int i = 0 ; i < 10 ; i ++){
            queue.enqueue(i);
            System.out.println(queue);


            if(i % 3 == 2){
                queue.dequeue();
                System.out.println(queue);
            }
        }
    }
}
```