五毒神掌刷题法

步骤

- 第1遍：

  5分钟：审题+思考，如果5分钟内有思路则直接进行代码实现，如果没有则看各种题解，寻求多种优秀的解法，理解并比较这些解法的优劣，最后进行解法背书。

- 第2遍：

  马上进行一次默写，在 LeetCode 上提交并通过，在其中多体会各种题解对算法的优化过程。

- 第3遍：

  24小时后重做一遍，加深对不同解法的理解和熟练程度。

- 第4遍：

  一周后再重做。

- 第5遍：

  面试前一两周再做最后一遍回顾。

体会

最近一周一直在思考超哥的五遍刷题法为什么能够颠覆我们大多数人原本的刷题套路和理念，后来我想明白了，超哥的这种刷题理念实际上是将我们每个人潜意识中习惯的逃避思维进行了纠正，并提供了一套绝对理性和有效的刷题方法。

我们原先那种只做一遍的固定刷题套路往往将我们折磨的筋疲力竭，写出的又长又丑的代码最后只感动了自己，甚至自己下一次也不愿意再写出这样的代码，对这道题的印象便只剩下我已通关的错觉。可我们往往忘了，经典的算法是经过无数人千锤百炼之后的结果，而优秀的题解便往往采用这样的算法思维，如果我们一直闭门造车而对眼前的宝藏视而不见的话便只能一直处在事倍功半的状态。只有站在巨人的肩膀上，高屋建瓴，不断提升自身的眼界和底蕴才能达到我们预期的理想高度。

ps：很惭愧，这周刚报了班工作上就刚好进入一个紧张的忙碌期，导致这一周都没好好刷题，希望下周调整好状态，紧跟大家的步伐。

Queue 之 LinkedList 源码分析

因为 LinkedList 是 Queue 的实现类，所以 LinkedList 也是一个双端队列结构，那么在分析主要源码方法时只需要取一端进行分析即可。

addFirst(E e)

```
// 向队列头插入元素
public void addFirst(E e) {
    linkFirst(e);
}

private void linkFirst(E e) {
    final Node<E> f = first;
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    if (f == null)
        last = newNode;
    else
        f.prev = newNode;
    size++;
    modCount++;
}
```

主要关注linkFirst()方法，关键步骤在于第二行：Node<E> newNode = new Node<>(null, e, f); 由于 LinkedList 也是一个双向链表，所以相邻节点之间是通过前驱后继指针进行关联的，那么在链表头进行插入时，新的头节点的前驱必然为 null，并且后继指针指向原头节点 f 作为后继节点。

同时原头节点 f 的前驱指向新头节点（如果 f 不为空）

getFirst()

```
// 返回头节点元素
public E getFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return f.item;
}
```

这里直接将头部元素进行了返回，不做太多讲解

get(int index)

```
// 获取指定索引位置的元素
public E get(int index) {
    checkElementIndex(index);
    return node(index).item;
}
```

checkElementIndex()检查索引值的合法性

重点在 node() 方法上

```
// 返回指定元素索引处的(非空)节点
Node<E> node(int index) {
    // assert isElementIndex(index);
    if (index < (size >> 1)) {
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

这里对给定索引位置的节点检索使用了比较巧妙的对半查找，通过判断给定索引的位置，比如索引节点在前半部，那么从队列头往后查找；如果在后半部分则从尾部往前找，这样能保证只检索一半以内的节点就可得到结果，提高了检索效率。

removeFirst()

```
// 删除并返回列表中的第一个元素
public E removeFirst() {
    final Node<E> f = first;
    if (f == null)
        throw new NoSuchElementException();
    return unlinkFirst(f);
}

// 释放非空首节点 f
private E unlinkFirst(Node<E> f) {
    // assert f == first && f != null;
    final E element = f.item;
    final Node<E> next = f.next;
    f.item = null;
    f.next = null; // help GC
```

```
        first = next;
        if (next == null)
            last = null;
        else
            next.prev = null;
        size--;
        modCount++;
        return element;
    }
```

还是重点查看 unlinkFirst() 方法，这里主要做的操作就是将给定首节点 f 移除，并且将 f 的后一个节点置为首节点（如果存在），如果 f 没有下一个节点，那么在移除 f后将首尾部节点都置空。

优先队列 PriorityQueue 源码分析

add(E e)

```
    // 添加元素
    public boolean add(E e) {
        return offer(e);
    }
    public boolean offer(E e) {
        if (e == null)
            throw new NullPointerException();
        modCount++;
        int i = size;
        if (i >= queue.length)
            grow(i + 1);
        size = i + 1;
        if (i == 0)
            queue[0] = e;
        else
            siftUp(i, e);
        return true;
    }
```

重点关注 offer() 方法。首先，判断元素个数是否达到了最大容量，达到了则先进行一次扩容操作 grow()，然后再将 元素插入至 i 索引位置 siftUp(i, e)，这里的 i 是队列的长度，即最后插入队列的最后位置。

element()

```
    // 获取首部元素
    public E element() {
        E x = peek();
        if (x != null)
            return x;
        else
            throw new NoSuchElementException();
    }
    public E peek() {
```

```
        return (size == 0) ? null : (E) queue[0];
    }
```

如果存在首部元素则返回，否则抛出异常。

remove()

```
    // 弹出首部元素并返回
    public E remove() {
        E x = poll();
        if (x != null)
            return x;
        else
            throw new NoSuchElementException();
    }

    @SuppressWarnings("unchecked")
    public E poll() {
        if (size == 0)
            return null;
        int s = --size;
        modCount++;
        E result = (E) queue[0];
        E x = (E) queue[s];
        queue[s] = null;
        if (s != 0)
            siftDown(0, x);
        return result;
    }
```

如果存在首部元素则弹出，不存在则抛出异常。主要看 poll() 方法，弹出首元素后将元素个数减一，如果此时在队列中还存在元素，那么将首部元素替换为原队末元素，并进行自上而下的堆化处理 siftDown(0, x)，这里 x 即指原队末元素。