1

利用accounting方法分析；

每一位被置1的时候，代价为3；

> 1用于本次置1的开销；
>
> 1用于存储在该位置上，用于支付此位置被置0的开销；
>
> 1用于初始的b个1变0的开销；

因为初始共有b个1，若这b个1都变为0，共需要b开销。但是我们已知$n = \omega(b) \geq b$；因此n次操作中，“1用于初始的b个1变0的开销”的加和一定大于b；

因此，$\sum a_i \leq \sum c_i$对于任意n个操作序列都成立；

因此总平摊代价为O(n);

2

伪代码设计：

```
ENQUEUE(e,stack-in,stack-out):
    stack-in.push(e)
    IF stack-out.isEmpty()
    THEN:   #把stack-in的元素倒入stack-out中
        stack-out.push(stack-in.pop())

DEQUEUE(e,stack-in,stack-out):
    return stack-out.pop()
```

假设共进行n次ENQUEUE和DEQUEUE操作。

一个对象被压入栈后，最多被弹出2次；因此对于ENQUEUE和DEQUEUE操作而言，调用stack.pop的次数不超过2n，且调用stack.push的次数不超过n；

最坏情况下操作序列的代价为$T(n) \leq 3n = O(n)$

平摊代价为$O(n)/n = O(1)$

3

用无序数组即可实现该数据结构。

在无序数组中，对于INSERT(S,x)的时间为O(1)；对于DELETE-LARGER-HALF(S)，先用划分的方法找到S的中位数，然后删除最大的|S|/2个元素，假设找中位数和删除的总时间为$c_1 n$。

用Accounting方法分析这个过程，插入一个元素收取$1 + 2c_1$费用，其中1付给插入操作。c_1用于预支删除操作；每次DELETE-LARGER-HALF操作，全体集合元素都需要消耗c_1，然后将删除部分元素的余款(c_1)分配到剩余的元素上。这样数组中每个元素始终有$2c_1$存款。

所以任意m个操作序列可以在O(m)时间内运行。