Exercise 17.3-6. We will name each of the stacks as the tail stack or the head stack. (which will be clear shortly that they’ll correspond to the head and tail of the queue).
Cost of Enqueue: Push the element to the tail stack. $O(1)$ time
Cost of Dequeue:
Case 1- Head queue is not empty. Pop one element from the head queue: $O(1)$ time
Case 2- Head queue is empty. Then we have to pop every element in tail queue and push it to the head queue one by one. If the number of elements in the tail queue is $s$, this takes $2s+1$ time (1 pop and 1 push for every element and pop the new head from head stack)
As the potential function, we’ll use twice the number of elements in the tail queue (ie say before the operation they were $s$)
\[
\dot{c}_{\text{ENQUEUE}} = c_{\text{ENQUEUE}} + \Delta \Phi = 1 + 2 \cdot (s + 1) - 2s = 1 + 2s + 2 - 2s = 3
\]
\[
\dot{c}_{\text{DEQUEUE}} = c_{\text{DEQUEUE}} + \Delta \Phi
\]
Case 1: We pop from head stack. so the number of elements in tail stack doesn’t change.
\[
\dot{c}_{\text{DEQUEUE,case1}} = c_{\text{DEQUEUE,case1}} + \Delta \Phi
\]
\[
\dot{c}_{\text{DEQUEUE,case1}} = 1 + 2 \cdot s - 2 \cdot s = 1
\]
Case 2: We empty the tail stack and pop again from the head stack.
\[
\dot{c}_{\text{DEQUEUE,case2}} = c_{\text{DEQUEUE,case2}} + \Delta \Phi
\]
\[
\dot{c}_{\text{DEQUEUE,case2}} = 2 \cdot s + 1 + 2 \cdot 0 - 2 \cdot s = 2s + 1 - 2s = 1
\]
So the amortised cost for enqueue is 3 and amortised cost for dequeue is 1 where both are $O(1)$.

Exercise 17.3-7. The data structure I’ll be using is a regular unsorted array.
Calculating the actual costs:
Insert: Just add the new element to the end of the array. Cost is $O(1)$
Delete_Larger_Half: Find the lower median in $O(|S|)$ time using SELECTION algorithm. Then partition the array around this element in $O(|S|)$ time with a single sweep. Then delete the elements that are larger than (ie on the right side of) the median in constant time. So the actual cost for deleting the larger half is $O(|S|)$. Let’s assume actual cost is $d_1 |S| + d_2$
Say our potential function is a constant $c$ times the number of elements we have in the set. In other words $\Phi(S) = c|S|

Let’s calculate the amortised costs:
\[
\dot{c}_{\text{amortised}} = c_{\text{actual}} + \Delta \Phi
\]
Insert: Increases the number of elements by 1.
\[
\dot{c}_{\text{insert}} = 1 + (c(|S| + 1) - c|S|) = 1 + c|S| + c - c|S| = c + 1
\]
Delete_Larger_Half: Reduces the number of elements in the set by half.
\[
\dot{c}_{\text{delete_larger_half}} = d_1 |S| + d_2 + \left(\frac{c|S|}{2} - c|S|\right)
\]
\[
\dot{c}_{\text{delete_larger_half}} = d_1 |S| - \frac{c}{2}|S| + d_2
\]
If we set the constant in potential function $c$, to be twice the constant $d_1$ that is coming from the actual cost of delete_larger_half, the amortised costs become:
\[
\dot{c}_{\text{insert}} = 2d_1 + 1 = O(1)
\]
\[
\dot{c}_{\text{delete_larger_half}} = d_2 = O(1)
\]
Since every operation needs constant time, $m$ operations would require $O(m)$ time.

Exercise 17.4-1. When implementing a dynamic hash table, the insertions are carried in this manner: First the element is hashed with using it’s key and the returning hash value is used as the index to access the hash table. Then there are two possibilities: either the location addressed by the index is empty so we just put our element to that location or that location is already occupied with another element that hashes to the same location. If we are implementing an open-address hash table, we can just insert one element to one index (as opposed to chaining) so we must find a new location to the new element either with linear or quadratic probing or double hashing etc.
So when we have a hash collision, insertions may take longer - since the second location we are looking might also be occupied. Especially when load factor gets close to 1, almost all the locations in the table is occupied so in the worst case we may have to look all the locations in the table before finding a suitable empty location for the item we are trying to insert (which takes O(n) time where n is the size of the hash table). Since we want to avoid that kind of inefficiency - main purpose of the hash tables was to insert and search in constant time - we might consider the table to be full before it's load factor actually reaches 1.

Assume we consider table to be full when load factor reaches 0.75.

There are two cases:
i. \( a < 0.75 \) before insertion. As shown in page 242 Cormen, the expected number of lookups before a successful insertion is \( \frac{1}{1-a} \). Since \( a < 0.75 \), \( \frac{1}{1-a} < 4 \). So we'll have an expected number of 4 lookups for occupied locations before an insertion, in other words expected value of the actual cost is 4. Using a potential analysis similar to the one we did in class, it can be shown that the amortised cost of each insertion is O(1).

ii. \( a = 0.75 \) before insertion, which means the next insertion will trigger a table expansion and double the hash table size. Also we have to change our hash function, such that it'll give mappings to the new expanded hash table (i.e., for simplicity one can use key%table_size as the hash function). Also all the elements in the smaller hash table needs to be re-hashed to the new table. Thus the actual cost for this insertion becomes O(n) where n was the hash table size before insertion. But a similar amortised analysis as in first part would show that amortised cost for this insertion is also O(1).

Exercise 17.4-2. This corresponds to the cases ii) b and ii) c we covered in class. Since \( a_{i-1} \geq \frac{1}{2} \), we have to consider two different possibilities for the final load factor.

i. \( a_{i-1} > \frac{1}{2} \) then \( a_i \geq \frac{1}{2} \). assume before the deletion the table had exactly n elements and the table size was s.

actual cost of deletion, \( c_i = 1 \), since table size doesn't change.

\[
\Phi(T_i) = 2 \times (n - 1) - s \\
\Phi(T_{i-1}) = 2 \times n - s \\
\Delta \Phi = \Phi(T_i) - \Phi(T_{i-1}) \\
\Delta \Phi = (2 \times (n - 1) - s) - (2 \times n - s) = -2 \\
c_i = c_i + \Delta \Phi = 1 - 2 = -1
\]

ii. \( a_{i-1} = \frac{1}{2} \) then \( \frac{1}{4} < a_i < \frac{1}{2} \). assume before the deletion the table had exactly n elements and the table size was s. (therefore 2n=s)

actual cost of deletion, \( c_i = 1 \), since table size doesn't change.

\[
\Phi(T_i) = \frac{1}{2}s - (n - 1) = \frac{1}{2}2n - n + 1 = 1 \\
\Phi(T_{i-1}) = 2 \times n - s = 0 \\
\Delta \Phi = \Phi(T_i) - \Phi(T_{i-1}) \\
\Delta \Phi = 1 - 0 = 1 \\
c_i = c_i + \Delta \Phi = 1 + 1 = 2
\]

Exercise 17.4-3.

Case I. insertion: Assuming table size is doubled when load factor reaches 1. assume before insertion the table had exactly n elements and the table size was s.

Case Ii. \( \frac{1}{2} \leq a_{i-1} < \frac{1}{2} \) then \( a_{i-1} \leq \frac{1}{2} \). actual cost of deletion, \( c_i = 1 \), since table size doesn’t change.

\[
\Phi(T_i) = |2(n + 1) - s| = s - 2n - 2 \\
\Phi(T_{i-1}) = |2n - s| = s - 2n \\
\Delta \Phi = \Phi(T_i) - \Phi(T_{i-1}) \\
\Delta \Phi = (s - 2n - 2) - (s - 2n) = -2 \\
c_i = c_i + \Delta \Phi = 1 - 2 = -1
\]

Case IIi. \( \frac{1}{2} \leq a_{i-1} < 1 \)
actual cost of deletion, $c_i = 1$, since table size doesn’t change...

$\Phi(T_i) = |2(n + 1) - s| = 2n + 2 - s$

$\Phi(T_{i-1}) = |2n - s| = 2n - s$

$\Delta \Phi = \Phi(T_i) - \Phi(T_{i-1})$

$\Delta \Phi = (2n + 2 - s) - (2n - s) = 2$

$\hat{c}_i = c_i + \Delta \Phi = 1 + 2 = 3$

Case I.iii. $a_{i-1} = 1$, table insert doubles the table size ($s=n$)

actual cost of deletion, $c_i = s + 1$

$\Phi(T_i) = |2(n + 1) - 2s| = 2$

$\Phi(T_{i-1}) = |2n - s| = 2n - s = s$

$\Delta \Phi = \Phi(T_i) - \Phi(T_{i-1})$

$\Delta \Phi = 2 - s$

$\hat{c}_i = c_i + \Delta \Phi = (s + 1) + (2 - s) = 3$

Case II. deletion: When load factor drops below 1/3 we contract it by multiplying the size by 2/3. Assume before insertion the table had exactly $n$ elements and the table size was $s$.

Case I. $a_{i-1} = \frac{1}{3}$, deletion makes the table size $\frac{2}{3}s$, $s = 3n$

actual cost of deletion, $c_i = \frac{1}{3} - 1$, copying all the elements before contraction except the one being deleted...

$\Phi(T_i) = |2(n - 1) - \frac{2}{3}s| = |2n - 2 - 2n| = 2$

$\Phi(T_{i-1}) = |2n - s| = s - 2n = s - 2 \frac{2}{3} = \frac{s}{3}$

$\Delta \Phi = \Phi(T_i) - \Phi(T_{i-1})$

$\Delta \Phi = 2 - \frac{s}{3}$

$\hat{c}_i = c_i + \Delta \Phi = \frac{s}{3} - 1 + 2 - \frac{s}{3} = 1$

Case IIi. $\frac{1}{3} < a_{i-1} \leq \frac{1}{2}$, after deletion $a_i \leq \frac{1}{2}$ ($2n<s$)

actual cost of deletion, $c_i = 1$, since table size doesn’t change...

$\Phi(T_i) = |2(n - 1) - s| = s - (2n - 2) = s - 2n + 2$

$\Phi(T_{i-1}) = |2n - s| = s - 2n$

$\Delta \Phi = \Phi(T_i) - \Phi(T_{i-1})$

$\Delta \Phi = (s - 2n + 2) - (s - 2n) = 2$

$\hat{c}_i = c_i + \Delta \Phi = 1 + 2 = 3$

Case IIIi. $\frac{1}{2} \leq a_{i-1} \leq 1$, after deletion $a_i \geq \frac{1}{2}$ ($2n \geq s$)

actual cost of deletion, $c_i = 1$, since table size doesn’t change...

$\Phi(T_i) = |2(n - 1) - s| = 2n - 2 - s$

$\Phi(T_{i-1}) = |2n - s| = 2n - s$

$\Delta \Phi = \Phi(T_i) - \Phi(T_{i-1})$

$\Delta \Phi = (2n - 2 - s) - (2n - s) = -2$

$\hat{c}_i = c_i + \Delta \Phi = 1 - 2 = -1$

Since in all the cases the amortized cost of insertion or deletion is smaller than a constant (i.e., $\beta = 3$) all table operations (specifically table delete) is bounded above by a constant.

Exercise 17-1.

a. for (i=0; i<n; i++)

```c
{  
j = rev4(i)
  if(j < i)
    swap(A[i],A[j])
}
```

Every loop takes $k+\text{const}=O(k)$ time and we have $n$ iterations so the run time is $O(nk)$. We could use if(i<j) instead of if(j<i) but not if(i!=j) because it’d swap each element twice and we’d end up
with the initial permutation.

b.
c. Yes it is possible. The code given below calculates the number of leading 1’s in every step, replaces it with 000...001 where the number of zeros is equal to the number of leading 1’s.

BIT_REVERSED_INCREMENT(count,k)
{
    //find the leading 1’s
    x = 0;
    while(count AND 2^{k-1}! = 0)
    {
        left_shift(count, 1);
        x ++;
    }
    //here x is equal to the number of leading 1s
    //put the new 1 in 000...001
    count=count OR 2^{k-1}
    //put the new leading zero’s
    while(x>0)
    {
        right_shift(count, 1);
        x--;
    }
    return count;
}

Using a similar analysis to the INCREMENT function in Cormen, page 408, we can show that amortised cost per increment is O(1), so n iterations take O(n) time..