1. Show that there exists a Huffman tree with \( n \) nodes whose height is \( n-1 \). Explain.

**Solution:** This will happen if every pair of lowest-frequency characters will remain low-frequency even after combining them. In other words, for a set of characters \( \{c_1, c_2, \ldots, c_n\} \) for every \( c_i \) s.t. \( i = 1, 2, \ldots, n-2 \), \( f_i + f_{i+1} < f_{i+2} \). This way the characters are never re-ordered after merging. An example includes the following:

<table>
<thead>
<tr>
<th>Character</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>12</td>
</tr>
</tbody>
</table>

2. Exercise 2.1 in the Huffman coding notes (p. 4).

**Solution:** In this case we simply remove the child and make the parent a leaf. There are no other nodes with the same prefix and we save one bit.

3. (a) Read Section 10.1. This should be very easy, since I assume you already know about stacks and queues. The text has an implementation of a queue in terms of an array of fixed size, in which the operations Enqueue and Dequeue each have cost \( O(1) \). (Remember that this is just a fancy way of saying that the cost is uniformly bounded by some constant.) You don’t have to write anything – just read this and understand it. Let me know if you find anything confusing.

(b) The only problem with that implementation of queues is that the size of a queue is bounded by the size of the array. It would be nicer to have a queue which, like a stack, had no upper bound on its size. (Now of course in practice, a stack does have an upper bound on its size – that’s why we talk about "stack overflow". But also in practice we have ways of dealing with this so that except in very unusual circumstances, stacks act as if their size was not constrained.)

So assuming that we had an implementation for stacks in which their size was not constrained, and in which the operations Push and Pop each have cost 1, I first want you to do Exercise 10.1-6 (page 236). The idea of this exercise is to provide an implementation of a queue in which the size of the queue is not constrained. In particular, you should derive the worst-case costs of Enqueue and Dequeue, and it should be clear that the cost of Dequeue is not \( O(1) \). Presumably, that’s the price we pay for having a queue of unlimited size.

(c) Then finally, do Exercise 17.3-6 (page 463). The idea of this exercise is show that in the implementation of the queue in Exercise 10.1-6 (which you just did), the amortized cost of both Enqueue and Dequeue for this data structure is \( O(1) \). So things really are not so bad – in fact, they’re pretty good. **Solution (for both b and c):** We can have two stacks, one "in" and one "out". That is – when push an item into the stack "in" (it takes \( O(1) \)). When popping, pop from stack "out" if not empty. If "out" is empty, pop every item in stack "in" and push it into stack "out". This process takes \( O(n) \) (\( n \) is the size of stack "in"). However, it is done once per item. Since we push out of stack "in" and into stack "out", the items reverse their order so that they are in a "first in first out" order. The amortized run time is \( O(1) \) and it’s easy to see that every item goes through at most one push (into stack "in") one pop (out of stack "in"), one push (into stack "out") and one pop (out of stack "out"). Hence, a total of 4 operations per item.
4. A stack supports two operations: Push and Pop. Implementing the two operations using a linked
list takes \(O(1)\) time per operation. Suppose we are given two stacks, A and B with \(n\) and \(m\)
elements, respectively. We want to implement the following operations:

- Push(A,x) – push \(x\) elements into A.
- Push(B,x) – push \(x\) elements into B.
- MultipopA(k) – pop \(\min\{k, n\}\) elements from A (that is, try to pop \(k\) elements from A. If
  \(n < k\) stop when the stack is empty).
- MultipopB(k) – pop \(\min\{k, m\}\) elements from B.
- Transfer(k) – repeatedly pop an element from A and push it to B until either \(k\) elements
  have been transferred or A is empty.

a. What is the worst case running time of MultipopA, MultipopB and Transfer in terms of \(m\)
and/or \(n\) and/or \(k\)? Explain briefly. Solution: MultipopA is \(O(\max\{k, n\})\), MultipopB is
\(O(\max\{k, m\})\) and transfer is \(O(k)\)

b. Show that the amortized running time per operation is \(O(1)\) for a sequence of Push(A,n)
and Transfer(n).

Solution: We can use operation counting – For Push(A,n) we do \(O(n)\) operations and for
transfer we do \(n\) pops and \(n\) pushes. Overall we have \(3n\) operations for \(n\) items, so the
amortized runtime is \(3 = O(1)\). We can also use the account method, allocate \(3\$\) for a push,
pay one and save \(2\). Then for the transfer, pay the \(2\$\).