Problem 1. (Based on Book Exercise 6.2) Show the min heap that results from inserting the following sequence of elements: 10, 12, 1, 14, 6, 5, 8, 15, 3, 9, 7, 4. Show the heap as both a tree and an array.

Solution 1. See Figure 1.

```
   1
  / \  /
 3   4
/ \ / \ / \
6 7 5 8
\ /   / \\
15 14 9 10
```

Figure 1: The solution to Problem 1.

Problem 2. (Based on Book Exercise 6.3) Show the step-by-step result of calling pop() three times from the min heap resulting from Problem 1. Show the heap as both a tree and an array.

Solution 2. See Figure 2.

```
   5
  / \  /
 6   8
/ \ / \ / \\
10 7 9 12
\ /   / \\
15 14
```

Figure 2: The heap in Problem 2.

Problem 3. Suppose we wanted to implement a heap via a ternary approach (each node has up to three children).

- For a node at index $p$ in the array, what indices should contain $p$’s children?
- For a node at index $c$ in the array, what index should contain $c$’s parent?

Solution 3.
• $3p + 1$, $3p + 2$, $3p + 3$
• $(c - 1)/3$

Problem 4. (Based on Book Exercise 6.1) Consider implementing a min priority queue (not necessarily using a heap). Ignoring any time spent to allocate memory, is it possible to implement a min priority queue that has the following worst-case running times:

- $O(1)$ for `push()`, `front()`, and `pop()`?
- $O(1)$ for `pop()` and `front()`?
- $O(1)$ for `push()` and `front()`?
- $O(1)$ for `push()` and `pop()`?

For each answer, either describe why not or give a rough idea for an implementation.

Solution 4.

- No: this would allow general sorting in $O(n)$ worst-case time sorting via `push()`ing all elements, then `pop()`ing them.
- Yes: use a sorted array, spend $\Theta(n)$ time for `push()` to insert the new element into the sorted array.
- Yes: use an unsorted array, `pop()` finds the smallest element and removes it in $O(n)$ time.
- Yes:
  - Use an unsorted array to hold the elements.
  - `push()` adds the element to the unsorted array.
  - A variable $k$ tracks how many times `pop()` has been called since the last `front()` call.
  - `pop()` does nothing.
  - When `front()` is called, spend $O(n \log(n))$ time to sort the elements, then remove the $k$ smallest, and return the smallest of the remaining elements.