1 Чтобы реализовать стек, используя две очереди, проделаем следующие шаги. Для реализации нам необходимо "выразить"все команды стека через команды очереди. У стека есть 2 команды: PUSH() и POP(). Действуем так: если пользователь хочешь запушить элемент, мы смотрим на первую очередь: если она пустая - просто кладем в нее элемент, а если нет - все содержимое перекладываем во вторую очередь, кладем наш элемент в первую, а затем кладем из второй в первую все элементы из второй. Таким способом мы формируем из входной последовательности полностью реверснутую, а значит для операции POP() нам нужно будет просто взять первый элемент перво очереди. Итак, мы реализовали команды PUSH() и POP(), теперь выясним, с какой сложностью это проделано. Очевидно, POP() - за $O(1)$, потому что мы просто достаем первый элемент очереди. А вот PUSH() - уже за $O(n)$, так как в худшем случае нам придется 2 раза перегонять $(n-1)$ элемент сначала из 1 во 2 очередь, а затем обратно.

2 Хранить почти-полное троичное дерево в массиве можно следующим образом. Нумеруем вершины дерева сверху-вниз слева-направо <u>от 1</u>, то есть корень дерева это 1, его левый сын - 2, центральный - 3, правый - 4, и так далее. В таком случае номер левого ребенка это (номер родителя * 3 - 1), номер центрального это (номер родителя * 3), а номер правого - это (номер родителя * 3 + 1). В обратную сторону так: если (номер ребенка + 1) делится на 3, то номер родителя это частное от этого деления, если делится нацело, то это число и есть номер родителя, иначе если (номер ребенка - 1) делится на 3, то номер родителя это частное от этого деления.

3 Сначала формируем по исходному массиву кучу по убыванию за $O(n)$. Затем последовательно k раз извлекам минимальный элемент из кучи. Итого, получаем сложноть $O(n + k * \log_2(n)$

4 Нам известно, что $y > x$, причем между ними нет ни одного числа, принадлежащего дереву, а также правое поддерево x пустое. Это значит, что x нах-ся где-то в левом поддереве эл-та y. Таким образом, мы пока что доказали, что y это предок x. Далее возможны 2 варианта: либо само число x нах-ся непосредственно сразу за y без промежуточных эл-тов, тогда y - самый нижний предок, чей левый дочерний узел явл-ся самим x; либо в левом узле от y находится число, которое меньше, чем x, а тогда путь к x будет проходить через y и этот элемент. Таких элементов может быть несколько, но все они будут строго меньше x. Например, $5 - 1 - 2 - 3 - 4$, где $y = 5, x = 4$. Тогда при последовательном посроении дерева между y и x будут лежать аж 3 эл-та (аналогично, между ними на пути может лежать произвольное число элементов, удовлетворяющих данному неравенству: $elem < x < y$), но при этом y будет самым нижним предком, чей левый дочерний узел явл-ся предком x.

5. Рассмотрим левый узел. Пойдем от противного. Пусть у левой дочерней вершины <u>есть правое поддерево</u>. Тогда, все элементы этого поддерева удовлетворяют неравенству $a.key < elem < b.key$, так как лежат в правом поддереве a (а значит больше самого a), но при этом в левом поддереве b (а значит меньше b). Это противоречит условию задачи о том, что между $a.key$ и $b.key$ не лежит ни одно число. Это значит, что у левой дочерней вершины нет правого поддерева. Абсолютно аналогично с правым поддеревом вершины b.

6 . Двоичное дерево поиска - дерево, в левом поддереве которого лежат элементы, меньшие его, а в правом - большие. Построить такое дерево по произвольной входной последовательности = отсортировать массив входных данных за линейное время, что невозможно.

7. Время ожидания каждого клиента будет минимальным, если обслуживать клиентов в порядке возрастания времени на их обслуживание, потому что время ожидания каждого клиента это сумма всего времени на обслуживание всех предыдущих, а эту сумму мы можем минимизировать за счет минимизации каждого слагаемого, причем начиная уже с самого короткого по обслуживанию человека. То есть, чтобы второй по времени ожидания человек ждал минимально, первым надо обслужить самого быстрого. Также это можно объяснить так: для самого последнего обслуживаемого клиента не важно, в каком порядке обслуживали всех предыдущих, потому что его личное время ожидания при этом не изменится. Но уже для минимизации времени ожидания предпоследнего клиента очень важно, чтобы клиент с самым долгим временем обслуживания был после него, и так далее. Теперь нужно предложить сам алгоритм, который наиболее эффективно отсортирует по возрастанию время обслуживания каждого клиента, и это и будет ответом на то, в каком порядке нужно обслуживать клиентов. Для этого воспользуемся сортировкой "HeapSort сложность которой равна $O(n * \log_2(n)$.