1.

Для создания стека необходимо реализовать две операции: push(x) и pop().

Заведем две очереди: q_1 и q_2.

В q_1 будут храниться данные, а q_2 будет использоваться как вспомогательная, вначале она пустая.

push(x):

При добавлении элемента в стек, он добавляется в очередь q_1.

pop():

При извлечении элемента из стека, нам нужно достать его из хвоста очереди. Для этого перегоняем все элементы, кроме последнего, в очередь q_2. Извлекаем нужный (последний) элемент. Меняем местами ссылки на очереди (q_1 становится пустой запасной, а q_2 – теперь основная с данными).

Две основные операции операции реализованы, таким образом, получилась структура данных – стек.

2.

Почти-полное дерево – это дерево, для которого существует такое целое число $h \geq 0$, что:

1.каждый лист в дереве имеет уровень h или $h+1$

2. если узел дерева имеет правого потомка уровня $h+1$, тогда все его левые потомки, являющиеся листами, также имеют уровень $h+1$.

Если i – номер клетки родителя, то потомки имеют номера: $i \cdot 3+1$, $i \cdot 3+2$, $i \cdot 3+3$.

Если i – номер ребенка, то родителя – $\lfloor \frac{i}{3} \rfloor$.

Таким образом, в массиве можно хранить троичное дерево следующим образом: корневой узел имеет номер $i = 0$, затем для каждого ребенка узла i формулы имеют вид: $i \cdot 3+1$, $i \cdot 3+2$, $i \cdot 3+3$.

Так как дерево почти-полное, то пропуски расположены на последнем уровне непрерывно справа, следовательно, оставшиеся места можно заполнить $null$ такого как -1, если такого значения заведомо нет или $Integer.MIN\_VALUE$, то есть, сделать флажок об отсутствии потомков. Число таких пропусков: $3^{h+1} - 1 - len(a)$, где $3^{h+1} - 1$ – число элементов в полном троичном дереве, $len(a)$ – длина данного массива.

3.

1. Пусть элемент x расположен в самой крайней левой ветви. У элемента x нет правого ребенка. Следовательно, y не может стоять ниже, чем x. То есть, он находится выше. Находится в правом поддреве относительно корня он не может, так как иначе его левый дочерний узел не сможет быть предком x. Так как элемент $y > x$, то он расположен выше x в левой ветке. Он следующий по возрастанию, следовательно, он расположен непосредственно перед x, ведь дальше вверх по левой ветви следуют большие элементы.

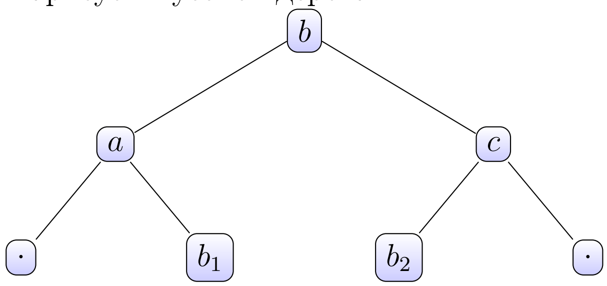
2. Пусть элемент x не расположен самой в крайней левой ветви, но расположен в левом поддереве. Тогда перед ним есть хотя бы один элемент, меньший x. И следующий по возрастанию за x y, у которого левый дочерний узел является предком x, будет расположен в первом, если отсчитывать от низа, узле, где $y > x$. Таким образом, y не будет расположен непосредственно перед x, но будет являться самым нижним предком x, у которого левый дочерний узел также является предком x.

Аналогично рассматривается случай, если x расположен в правом поддереве.

4.

Так как $a.key < b.key < c.key$ и у a нет правого потомка, то a не может быть расположена выше, чем b.

Нарисуем кусочек дерева:



Докажем, что узлов b_1 и b_2 не может быть.

От противного:

Пусть узел b_1 есть, тогда $b_1.key > a.key$ и $b_1.key < b.key$ (по свойству узлов бинарного дерева). Но это приводит к противоречию, так как по условию в промежутках $(a.key, b.key)$ узлов нет.

Пусть узел b_2 есть, тогда $b_2.key < c.key$ и $b_2.key > b.key$ (по свойству узлов бинарного дерева). Но это приводит к противоречию, так как по условию в промежутках $(b.key, c.key)$ узлов нет.

Следовательно, этих узлов быть не может.

5.

1. Пусть строка хранится как массив символов. Устроим строку w следующим образом: если элемент i входит в подмножество A, то $w[i] = 1$, иначе $w[i] = 0$. Длина такой строки будет n. Нам разрешено посмотреть значение всего одного элемента, так что мы сразу должны смотреть на ячейку с информаицей об x, поэтому меньшей длины строки не может быть. Если $w[x] = 1$, то элемент x входит в A, иначе нет. Следовательно, $s(n, k, 1) = n$

6.

Заведем структуру данных – очередь с приоритетом, которая поддерживает операции вставки и извлечения элемента с максимальным приоритетом.

Оптимальным будет порядок, упорядоченный согласно возрастанию t_i.

Таким образом, вставим все элементы в очередь с приоритетом: номер i – это ключ, $\frac{1}{t_i}$ – приоритет. Чем меньше t_i, тем больше приоритет этого клиента.

Таким образом, последовательно извлекая элементы из очереди мы получим необходимую последовательность номеров клиентов.

Корректность:

Если t_i – последовательность времен обслуживания, то оптимальным будет такая расстановка, когда максимальное значение этой последовательности встречается минимальное количество раз в итоговой сумме. То есть, если мы запишем итоговую сумму:

$a_1 + (a_1 + a_2) + (a_1 + a_2 + a_3) + ... + (a_1 + ... + a_n)$, где a_i – значение элементов t_i после их упорядочивания по возрастанию. Заметим, что при таком алгоритме как раз максимальный элемент встречается единожды, а следующие за ним в порядке увеличения повторений.

Таким образом, данный алгоритм позволяет найти последовательность номеров, соответствующую минимальной итоговой сумме.

Сложность:

Сортировка напрямую заняла бы $O(n \log n)$. Введение структуры данных – очередь с приоритетом, позволяет снизить сложность до $\mathbf{O}(\log \mathbf{n})$. Например, при реализации очереди с приоритетом на основе двоичной кучи.