Задача 3-1 (75 баллов).

Реализуйте следующий класс для хранения множества целых чисел:

```
class FixedSet {
 public:
  FixedSet();
  void Initialize(const vector<int>& numbers);
  bool Contains(int number) const;
};
```

`FixedSet` получает при вызове `Initialize` набор целых чисел, который впоследствии и будет хранить. Набор чисел не будет изменяться с течением времени (до следующего вызова `Initialize`). Операция `Contains` возвращает `true`, если число `number` содержится в наборе. Мат. ожидание времени работы `Initialize` должно составлять $O(n)$, где n — количество чисел в `numbers`. Затраты памяти должны быть порядка $O(n)$ в худшем случае. Операция `Contains` должна выполняться за $O(1)$ в худшем случае.

С помощью этого класса решите модельную задачу: во входе будет дано множество различных чисел, а затем множество запросов — целых чисел. Необходимо для каждого запроса определить, лежит ли число из запроса в множестве.

В первой строке входа — число n — размер множества. $1 \le n \le 100\,000$. В следующей строке n различных целых чисел, по модулю не превосходящих 10^9. В следующей строке — число q — количество запросов. $1 \le q \le 1\,000\,000$. В следующей строке q целых чисел, по модулю не превосходящих 10^9. Для каждого запроса нужно вывести в отдельную строку “Yes” (без кавычек), если число из запроса есть в множестве и “No” (без кавычек), если числа из запроса нет в множестве. См. примеры.

Пример входа	Пример выхода
3 1 2 3 4 1 2 3 4	Yes Yes Yes No
3 3 1 2 4 10 1 5 2	No Yes No Yes

Решение.

Нужно реализовать либо вариант `FixedSet` с помощью двухуровнего хеширования, либо вариант с использованием случайного графа.

Задача 3-2 (85 баллов).

Нынешним утром Боб хвастался перед Джо тем, что на днях побывал в совершенно удивительном королевстве X. Слушая его рассказ, Джо вдруг подумал, что королевство, о котором говорит Боб очень похоже на то, в котором побывал сам Джо лет эдак 50 назад. С этим путешествием у Джо связаны приятные воспоминания, и даже сохранилась старинная карта этого королевства Y, на которой отмечены все города.

Итак, на этой карте отмечено N городов, каждый из которых имеет натуральный номер от 1 до N. Между некоторыми городами построены дороги, по которым можно двигаться в обе стороны. Королевство устроено так, что от каждого города до любого другого можно добраться единственным способом, возможно, заходя на пути в некоторые другие города.

Боб, как настоящий путешественник, посетил все города королевства X, а также составил полную его карту. Удивительное дело, но в этом королевстве также N городов, пронумерованных числами от 1 до N. По карте Боба видно, что королевство X тоже устроено таким образом, что от каждого города можно добраться до любого другого единственным способом, возможно, заходя на пути в некоторые другие города.

За долгие 50 лет нумерация городов совершенно изменилась. Но Джо будет бесконечно вам благодарен, если вы определите, действительно ли Боб побывал в том же королевстве, что и Джо когда-то? И если это так, то он хочет знать для каждого города его новый номер.

В первой строке записано натуральное число $N(1 \leq N \leq 10^5)$ — количество городов в королевстве. В следующих $N-1$ строках дано описание королевства Y, а именно пары чисел: номера городов, соединенных дорогой. Следующие $N-1$ строк описывают королевство X в том же формате.

Если Джо ошибся, и королевства разные, то выведите «-1» (без кавычек). Иначе выведите N чисел, каждое в отдельной строке. i-ое число должно быть равно новому номеру города i. Если решений несколько, выведите любое.

Пример входа	Пример выхода
4 3 1 2 3 4 3 2 4 4 1 3 4	1 2 4 3

Решение.

В задаче требуется определить, изоморфны ли два дерева, и если да, то построить изоморфизм. Сведем задачу к случаю корневых деревьев: для деревьев с фиксированными корнями и ориентацией вниз от этих корней проверим, изоморфны ли они. Сначала выполним такое сведение. Нетрудно показать, что при любом изоморфизме центр дерева остается центром, а значит, можно подвесить деревья за их центры. Стоит отметить, что в дереве может быть 2 вершины в центре. В этом случае необходимо сделать проверку на изоморфизм корневых деревьев для обоих центров.

Найти центр дерева можно двумя способами:

1. Отрезать от дерева листья, пока не останется 1 или 2 вершины.

2. Взять середину самой длинной цепи в дереве. Для поиска самой длинной цепи возьмем произвольную стартовую вершину и найдем самую удаленную от нее с помощью обхода в глубину/ширину. Пусть это вершина u. Далее для u найдем

самую удаленную, пусть это вершина v. Тогда путь (u, v) образует самую длинную цепь.

Первый способ достаточно очевиден, но его эффективная реализация может быть сложнее. Доказательство корректности второго способа оставляется читателю в качестве упражнения. В обоих случаях поиск центра выполняется за $O(n)$.

Теперь решим задачу для корневых деревьев. Пусть у вершины v_1 сыновья $u_1, \ldots, u_k$, а у v_2 сыновья $u'_1, \ldots, u'_k$. Тогда если v_1 отображается в v_2 (например, в случае корней), то должна существовать некоторая перестановка, отображающая $u_1, \ldots, u_k$ в $u'_1, \ldots, u'_k$.

Сопоставим каждой вершине обоих деревьев некоторую метку m такую, что поддеревья, растущие из вершин v_1 и v_2, изоморфны тогда и только тогда, когда $m(v_1) = m(v_2)$. Как проставлять данные метки? Будем делать это рекурсивно в процессе обхода дерева в глубину. Для всякой листовой вершины u поставим $m(u) = 1$. Теперь рассмотрим вершину v, пусть у её сыновей метки $m_1, m_2, \ldots, m_k$. Из предыдущего абзаца следует, что метка вершины полностью определяется метками её сыновей. Если у двух вершин одни и те же метки сыновей (без учета порядка), то мы нашли искомый изоморфизм для этих поддеревьев. Поэтому, чтобы поставить метку $m(v)$, будем использовать следующий алгоритм:

- Заведем глобальный счетчик l и хеш-таблицу, в которой будем для мультимножеств из меток сыновей хранить соответсвующее значение счетчика.
- Если $\{m_1, \ldots, m_k\}$ есть в таблице, то возьмем соответствующую метку для v из таблицы.
- Иначе увеличим l и добавим новое мультимножество в таблицу.

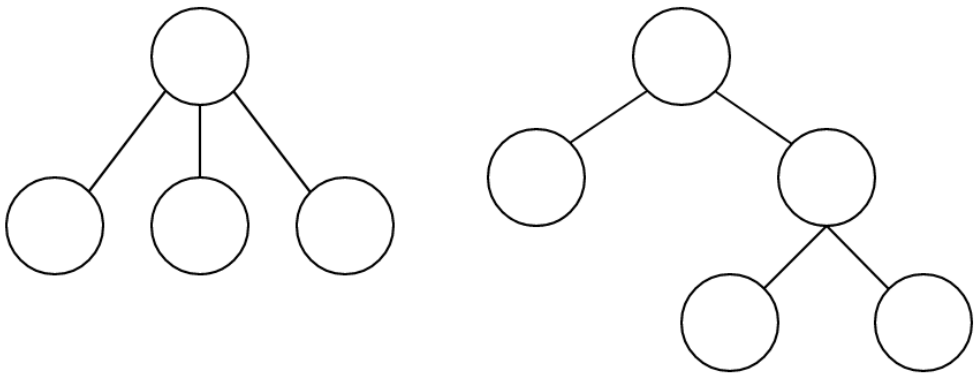
Оперировать мультимножествами сложно, поэтому зафиксируем порядок: будем класть в таблицу отсортированные вектора $(m_1, \ldots, m_k)$.

Теперь, зная метки всех вершин, построить искомый изоморфизм не составляет труда. Проблема в том, что хранить описанную таблицу достаточно накладно по памяти и, скорее всего, не пройдет в ограничения задачи. Фактически в задаче требуется уметь быстро сравнивать на равенство сложные объекты (поддеревья), а такую задачу можно решать с помощью хеширования. Осталось лишь определить соответствующую хеш-функцию. Вместо явного хранения векторов $(m_1, \ldots, m_k)$ будем хранить значения хеш-функции от них, иными словами, нужно определить $h(v) = h((m_1, \ldots, m_k))$.

Может возникнуть желание использовать классическое полиномиальное хеширование:

$$h(v) = \sum_{i=1}^{k} m_i \cdot x^{i-1} \mod M.$$

Но нетрудно предъявить пример, на котором данное хеширование не работает:



В обоих случаях имеем одинаковый хеш $1 + x + x^2 \mod M$.

Сведем задачу хеширования поддеревьев к хешированию строк. Рассмотрим обход в глубину поддерева. Когда идем вниз по ребру, будем выписывать '(', а когда в обратную сторону — ')'. Тогда обход поддерева порождает некоторую правильную скобочную последовательность. Дополнительно допишем в начало '(', а в конец ')' (тогда листу будет соответствовать последовательность '()'). Пусть теперь у сыновей вершины v получаются последовательности $s_1, \ldots, s_k$ и, без потери общности, $h(s_1) \le h(s_2) \le \cdots \le h(s_k)$ (переупорядочим их если это не так). Тогда обходу вершины v соответствует последовательность

$$(s_1 + s_2 + \cdots + s_k).$$

Теперь осталось посчитать полиномиальный хеш данной строки:

$$h(v) = 1 + \sum_{i=1}^{k} h_i \cdot x^{1+\sum_{j=1}^{i-1} |s_j|} \mod M + 2x^{1+|s_1|+\cdots+|s_k|}.$$

Здесь первая 1 соответствует открывающей '(', а последняя 2 соответствует закрывающей ')'. При подсчете здесь используется свойство полиномиального хеша

$$h(s + t) = h(s) + h(t) \cdot x^{|s|}.$$

Разумеется, не нужно явно строить строки s_i, поскольку нам достаточно знать лишь их длины, а эту величину можно посчитать через размеры поддеревьев.

С использованием хешей получаем $O(n)$ памяти. Нахождение центров требует $O(n)$ операций, подсчет хеш-функции описанным способом занимает $O(n \log n)$ (из-за сортировки).

Можно использовать идею с метками вместе с хешированием и получить $O(n)$ на подсчет хешей (в случае меток сортировка также нужна, но значения меток лежат в диапазоне $[1, 2n]$, поэтому можно использовать сортировку подсчетом и получить линейное время).

Задача 3-3 (75 баллов). Пете поручили написать менеджер памяти для новой стандартной библиотеки языка `H++`. В распоряжении у менеджера находится массив из N последовательных ячеек памяти, пронумерованных от 1 до N. Задача менеджера —

обрабатывать запросы приложений на выделение и освобождение памяти. Запрос на выделение памяти имеет один параметр K. Такой запрос означает, что приложение просит выделить ему K последовательных ячеек памяти. Если в распоряжении менеджера есть хотя бы один свободный блок из K последовательных ячеек, то он обязан в ответ на запрос выделить такой блок. При этом наш менеджер выделяет память из самого длинного свободного блока, а если таких несколько, то из них он выбирает тот, у которого номер первой ячейки — наименьший. После этого выделенные ячейки становятся занятыми и не могут быть использованы для выделения памяти, пока не будут освобождены. Если блока из K последовательных свободных ячеек нет, то запрос отклоняется. Запрос на освобождение памяти имеет один параметр T. Такой запрос означает, что менеджер должен освободить память, выделенную ранее при обработке запроса с порядковым номером T. Запросы нумеруются, начиная с единицы. Гарантируется, что запрос с номером T — запрос на выделение, причем к нему еще не применялось освобождение памяти. Освобожденные ячейки могут снова быть использованы для выделения памяти. Если запрос с номером T был отклонен, то текущий запрос на освобождение памяти игнорируется. Требуется написать симуляцию менеджера памяти, удовлетворяющую приведенным критериям.

В первой строке входа два числа N и M — количество ячеек памяти и запросов соответственно ($1 \le N \le 2^{31} - 1$, $1 \le M \le 10^5$). Каждая из следующих M строк содержит по одному числу. $(i + 1)$-я строка содержит положительное число K, если i-й запрос — запрос на выделение K ячеек памяти ($1 \le K \le N$), и отрицательное число $-T$, если i-й запрос — запрос на освобождение памяти, выделенной по запросу номер T ($1 \le T < i$).

Для каждого запроса на выделение памяти выведите в выход одно число на отдельной строке с результатом выполнения этого запроса. Если память была выделена, выведите номер первой ячейки памяти в выделенном блоке, иначе выведите число -1.

Пример входа	Пример выхода
6 8 2 3 -1 3 3 -5 2 2	1 3 -1 -1 1 -1

Решение.

Для решения этой задачи нам понадобится две структуры данных: двоичная куча и двусвязный список. Будем хранить отрезки свободной памяти в куче таким образом, чтобы самый длинный свободный отрезок всегда оказывался в голове кучи. Также будем хранить все свободные и занятые отрезки памяти по порядку в двусвязном списке. Изначально есть единственный свободный отрезок от 1 до N, он лежит и в куче, и в списке. По мере проведения операций по выделению и освобождению памяти, части этого отрезка станут занятыми, а затем освободятся, поэтому может получиться несколько

свободных и несколько занятых отрезков. В двусвязном списке все свободные и занятые отрезки идут по порядку, при этом нет двух подряд идущих свободных отрезков: если такая ситуация может произойти, мы просто объединяем соседние свободные отрезки. Все отрезки также должны помнить, на какой позиции в куче они лежат, чтобы их можно было удалять из кучи. Кроме того, заведем массив с результатами запросов, в котором в случае успешного запроса на выделение будет храниться указатель на выделенный отрезок памяти, иначе будет храниться какое-нибудь специальное значение, например `NULL`.

Рассмотрим реализации операций выделения и освобождения памяти.

Выделение памяти:

Находим самый длинный свободный отрезок в голове кучи. Сравниваем его длину с длиной запрашиваемого отрезка памяти. Если длина недостаточна, отклоняем запрос. Если длина свободного отрезка равна K — количеству запрашиваемой памяти — выделяем этот отрезок, удаляем из кучи, помечаем, как занятый. Если длина свободного отрезка более K, выделяем его начальный кусок длины K в отдельный отрезок, а оставшийся конец — в другой отрезок. Первый из этих отрезков — создается как новый занятый отрезок, а второй — получается сдвигом начала изначального наибольшего свободного отрезка вправо и уменьшением его длины. После этого уменьшенный начальный отрезок просеивается вниз по куче, а новый занятый — добавляется в двусвязный список сразу перед ним. Если удалось выделить память, сохраняем в массиве с результатами запросов указатель на выделенный отрезок.

Освобождение памяти:

Находим по указателю в массиве с результатами запросов занятый отрезок. Во-первых, он должен стать свободным. Во-вторых, посмотрим на его соседей в двусвязном списке. Их максимум двое: соседи слева и справа. Тогда если какие-то из них свободные, то их нужно "приклеить" к нашему отрезку и создать таким образом более длинный свободный отрезок. Их дальнейшие соседи уже не могут быть свободными, т.к. мы поддерживаем свойство, что в двусвязном списке со свободными и занятыми отрезками у свободного отрезка соседи — занятые отрезки. Итак, объединим наш отрезок со всеми свободными соседями (т.е. с 0, 1 или 2 соседями), затем пометим как свободный и добавим в кучу.

Таким образом, мы научились обрабатывать оба запроса. Какая же сложность у наших операций? Операции с двусвязным списком выполняются за константу времени, пометки на отрезках и изменения длин — тоже. Единственная структура данных, с которой происходят операции не за константное время,— куча. В ней операции делаются за $O(size)$, где $size$ — текущий размер кучи. Заметим, что если всего было сделано m запросов, то изначальный свободный отрезок не мог разбиться занятыми на более чем m свободных отрезков. Поэтому размер кучи всегда не более m, а значит все операции с кучей работают за $O(\log(m))$. Итого сложность задачи $O(m \log(m))$. Затраты памяти $O(m)$.

У этой задачи также есть и чуть более простое решение с sqrt-декомпозицией с семинара. Будем хранить свободные отрезки в блоках размера примерно $\sqrt{m}$. Порядок отрезков в блоках и порядок блоков при этом естественный. Для каждого блока также будем помнить максимальный отрезок в нем (самый левый из них). Тогда для выделения памяти нужно пройти по всем блокам и выбрать максимум. Освобождение памяти

устроено примерно так же, как в предыдущем решении (нужно объединить свободных соседей), для поиска нужного блока пользуемся тем, что отрезки в блоках у нас упорядочены естественным образом. Сложность такого решения $O(m\sqrt{m})$, но на практике оно не будет уступать предыдущему при заданных ограничениях.

Задача 3-4 (75 баллов). (Задача о скользящей k-й статистике.) В первой строке входа — три целых числа n, m, k. Во второй строке n целых чисел, задающих массив чисел, по которому будут двигаться два указателя, l и r. Изначально оба указателя направлены на самый первый элемент массива. Есть две операции: L — сдвинуть указатель l на один элемент вправо и R — сдвинуть указатель r на один элемент вправо. В третьей строке входного файла — m символов R или L, без пробелов, в одну строку. Это порядок выполняемых операций. Гарантируется, что указатель l никогда не "обгоняет" указатель r. Гарантируется, что указатель r никогда не выйдет за пределы массива. При этом $1 \le n, k \le 100000$, $0 \le m \le 2n - 2$. Все числа в массиве неотрицательные и не превосходят 10^9.

Выведите ровно m строк, в каждой — ровно по одному целому числу. После выполнения каждой из операций нужно вывести k-е в порядке возрастания число среди всех чисел от l до r включительно, либо -1, если всего чисел от l до r меньше, чем k.

Пример входа	Пример выхода
7 4 2 4 2 1 3 6 5 7 RRLL	4 2 2 -1
4 6 1 1 2 3 4 RLRLRL	1 2 2 3 3 4

Решение.

Заведем две кучи: `smallestKNumbers` и `otherNumbers`. В первой будем хранить, очевидно, k наименьших чисел из текущего отрезка от l до r (или меньше, если длина текущего отрезка меньше k), а во второй — все остальные числа из данного отрезка. При этом `smallestKNumbers` будет хранить в голове максимальный элемент, а `otherNumbers` — минимальный. Таким образом, в каждый момент времени, если на отрезке от l до r есть хотя бы k элементов, то k-я порядковая статистика находится в голове `smallestKNumbers`, иначе k-я порядковая статистика не существует, и тогда размер `smallestKNumbers` меньше k.

Нам потребуются следующие операции от обеих куч:

- Добавить элемент в кучу,
- Узнать минимум в куче,
- Удалить данный элемент массива из той кучи, в которой он находится.

Две первые операции реализуются как в обычной куче, и они работают за логарифм от размера кучи. Для третьей операции потребуется дополнительно хранить для каждого элемента массива в какой куче он лежит и на какой позиции. Имея эту информацию, мы умеем удалять элементы из кучи также за логарифм от размера, это было рассказано на лекциях.

Как тогда выглядит наш алгоритм.

Изначально, когда $l = r = 1$, мы добавляем в `smallestKNumbers` первый элемент массива, а `otherNumbers` оставляем пустой. Далее обрабатываем операции сдвига.

Сдвиг r означает, что нужно добавить $r + 1$-й элемент массива в наш отрезок:

1. Добавляем $r + 1$-й элемент в `smallestKNumbers`.
2. Если размер `smallestKNumbers` становится больше k, то удаляем голову `smallestKNumbers` и переносим ее в `otherNumbers`.
3. Выводим k-ю порядковую статистику на основе `smallestKNumbers`.

Сдвиг l означает, что нужно удалить l-й элемент массива из нашего отрезка:

1. Мы знаем, в какой куче и на какой позиции находится l-й элемент массива — удаляем его из этой кучи по этой позиции.
2. Если после этой операции размер `smallestKNumbers` стал меньше k, а куча `otherNumbers` непуста, то удаляем голову `otherNumbers` (в ней лежит наименьший из остальных элементов, т.е. тот, который теперь должен оказаться в `smallestKNumbers`) и добавляем в `smallestKNumbers`.
3. Выводим k-ю порядковую статистику на основе `smallestKNumbers`.

Такое решение работает за $O(m \log n)$ времени, т.к. каждый запрос обрабатывает за $O(\log n)$. Памяти тратим линейное количество, т.к. нам нужно хранить не более n элементов отрезка в кучах в каждый момент времени, помимо основного массива, и про каждый из них нам нужна константа дополнительной информации.

Замечания по реализации.

Чтобы в каждый момент времени иметь корректную информацию о том, в какой куче и на какой позиции находится каждый элемент массива, необходимо для каждого элемента хранить дополнительную информацию, например, в такой структуре:

```
struct ArrayElementInHeap {
  int value;
  enum {NONE, SMALLEST_K_NUMBERS, OTHER_NUMBERS} whichHeapIn;
  int heapPosition;
};
```

Необходимо иметь параллельный исходному массив `arrayElements` из `ArrayElementInHeap` и хранить в кучах, например, указатели на элементы на элементы этого массива. При операциях с кучами может происходить одно из двух:

1. Два элемента одной кучи обмениваются позициями — нужно поменять и их `heapPosition` друг с другом. Эту операцию необходимо инкапсулировать в методе `smartSwap` кучи, т.к. она точно используется как минимум 3 раза в операциях `siftUp` и `siftDown`, а на самом деле ее можно использовать вместо "ручных" решений почти во всех методах обычной кучи.

2. Элемент добавляется в кучу — в этот момент необходимо правильно инициализировать `whichHeapIn` и `heapPosition`.

3. Элемент удаляется из кучи — в этот момент необходимо присвоить `whichHeapIn = NONE;`, а также присвоить `heapPosition` какое-нибудь специальное значение.

Как мы видим, здесь в некоторой степени нарушается инкапсуляция данных: две разных кучи имеют доступ к элементам одного и того же массива `arrayElements`, что чревато рассинхронизацией данных. Почему так произошло? Потому что на самом деле здесь необходим более умный дизайн. Вместо того, чтобы работать с массивом чисел и двумя кучами отдельно, нужно понять, что на самом деле от нас хотят. А от нас хотят всего лишь реализовать интерфейс очереди, у которой кроме стандартных операций есть еще операция "узнать k-ю порядковую статистику среди элементов в очереди".

Ну давайте реализуем такой интерфейс в виде класса `QueueWithKthOrderStatistics`, в который и добавим в качестве членов как обычную очередь `arrayElements` (в которой для каждого элемента, находящегося в очереди, хранится `ArrayElementInHeap`, указывающий, в какой куче и на какой позиции он находится), так и обе кучи `smallestKNumbers` и `otherNumbers`.

При этом сдвиг r вправо означает выполнение операции `Push` для такой очереди, при этом нужно добавить $r+1$-й элемент исходного (внешнего) массива с числами в `arrayElements`, проинициализировать его по умолчанию, а затем произвести указанные выше действия с кучами. Сдвиг l влево означает операцию Pop для нашей кучи, которая представляет собой удаление первого элемента из `arrayElements` сначала указанным выше образом из соответствующей кучи, а потом вызовом `arrayElements.Pop();`. Операция `GetKthStatistics` нашей очереди будет смотреть на кучу `smallestKNumbers`, и если в ней есть k элементов, то возвращать ее голову, иначе ответит, что k-й порядковой статистики нет.

Т.к. теперь все данные инкапсулированы внутрь нашей собственной очереди, и все методы, имеющие к ним доступ, пишем мы сами, можно не беспокоиться за целостность данных, нужно только аккуратно реализовать операции $Push$ и Pop как указано.

Отметим, что нам не нужно делать очередь `arrayElements` на основе вектора размера n, мы можем всегда в ней хранить только те элементы, которые сейчас в очереди. Таким образом, мы поддерживаем операции с очередью "онлайн", т.е. если бы вместо исходного массива у нас был бы просто поток запросов извне, вида "добавить элемент", "удалить элемент", "узнать k-ю порядковую статистику", то мы могли бы обрабатывать эти запросы с помощью нашего класса за гарантированную логарифмическую сложность от текущего количества сохраненных элементов.

Далее, есть еще одна маленькая проблема: нам нужны две кучи, одна из которых в голове хранит минимум, а другая — максимум, и мы, конечно же, не хотим из-за этого писать две разные кучи. На самом деле, если задуматься, эти две кучи ничем не отличаются, кроме функции сравнения. Поэтому код нужно написать для всей кучи один раз,

при этом пользуясь для сравнения элементов некоторой специальной функцией сравнения, которой можно параметризовать кучу. Сделать это можно как минимум тремя способами (в порядке предпочтения):

1. Параметризовать шаблонный класс кучи функтором сравнения элементов, как принято в STL.
2. Передавать указатель на функцию сравнения элементов в конструктор кучи и запоминать.
3. Реализовать базовый класс кучи и породить от него два класса — кучу с минимумом в голове и кучу с максимумом в голове — в которых переопределен только виртуальный метод сравнения элементов кучи.

.

Первый способ наиболее предпочтителен, поскольку привычен и много раз опробован в STL. Третий способ наименее предпочтителен, поскольку наиболее затратен как в смысле написания кода и количества классов в системе классов, так и с точки зрения производительности.

Главное, что ни в коем случае нельзя писать две эти кучи с помощью copy-paste, т.к. дублирование кода — самое главное зло, с которым необходимо бороться.

Задача 3-5 (85 баллов). В этой задаче вам нужно реализовать структуру данных для приближенного поиска ближайшего вектора в заданной коллекции. В качестве расстояния между векторами мы будем использовать угол между ними $\alpha(\bar{x}, \bar{y})$, т.е.

$$\alpha(\bar{x}, \bar{y}) = acos(\langle \bar{x}, \bar{y} \rangle).$$

Все вектора в задаче уже нормализованы.

Пусть для заданного вектора $\bar{x}$ ближайшим является $\bar{q}$. Тогда ваша структура должна вернуть такой $\bar{y}$ в качестве ответа, что

$$\alpha(\bar{x}, \bar{y}) \leq 3 \max(0.1, \alpha(\bar{x}, \bar{q})).$$

Вам не нужно писать ввод-вывод, а также функцию main. Все это уже реализовано за вас здесь: `https://gist.github.com/astiunov/bdc3cdb03269b15881e979863afa6c54`. Тот же код запускается на сервере.

Пришлите заголовочный файл с реализацией класса ApproximateNearestNeighbor. Пример класса вы можете посмотреть здесь: `https://gist.github.com/astiunov/a3672be5bb1bdc7939624c15ee6783e8`.

Конструктор принимает вектора, среди которых нужно искать ближайшего, а метод GetNearest возвращает индекс вектора (нумерация с 1), достаточно близкого к заданному (исходя из условия).

Количество векторов в коллекции не привышает 50000, количество запросов не превышает 50000. Все вектора имеют одинаковую размерность d, причем $3 \leq d \leq 50$.

Несколько советов:

- Использование acos в коде сведет на нет все ваши усилия по написанию быстрого решения, потому что вычисление арккосинуса очень дорогое.

- В качестве хеш-функции используйте проекцию на случайную плоскость, как описано здесь: `https://en.wikipedia.org/wiki/Locality-sensitive_hashing#Random_projection`. Значением одной хеш-функции будет 1 бит. Чтобы сэкономить память группируйте биты в машинные слова и используйте эти слова как ключи (а не список битов).
- Для тестирования можно взять готовые коллекции векторов (которые к тому же непосредственно применяются на практике), например отсюда: `https://nlp.stanford.edu/projects/glove/`.

Решение.
Нужно реализовать в точности алгоритм LSH с лекции.