19. Definition of a heap, inserting/removing element from heap, constructing heap from vector, complexity analysis, std::priority_queue.

Definition of a heap, max-heap, min-heap

Определение: Бинарное дерево называется **полным**, если все уровни бинарного дерева заполнены, кроме, возможно, последнего уровня, на котором элементы расположены слева направо.

Замечание: Полное бинарное дерево сбалансировано.

Бинарное дерево называется **Максимальной пирамидой**, если оно полное и значение каждого узла больше или равно значениям его дочерних узлов.

Бинарное дерево называется **Минимальной пирамидой**, если оно полное и значение каждого узла меньше или равно значениям его дочерних узлов.

Heap (куча) в памяти хранится с использованием вектора.

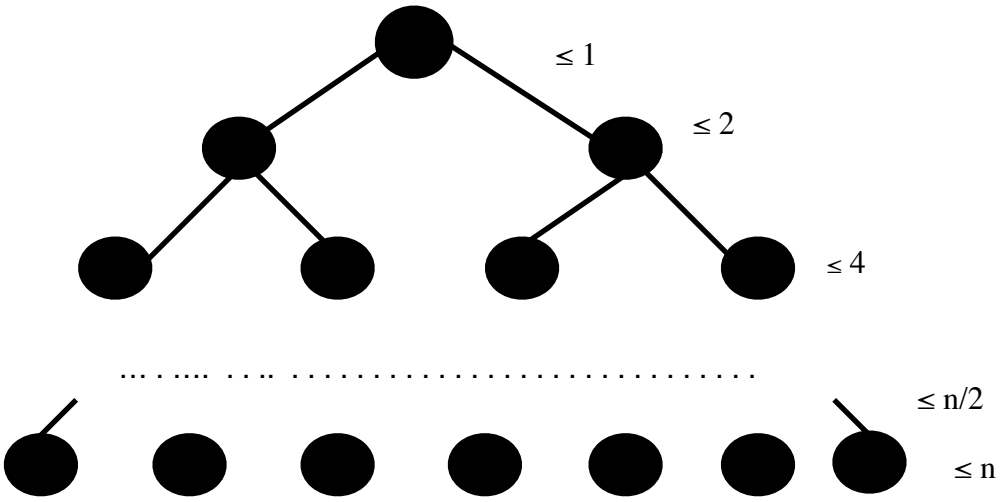
- Корень дерева — элемент с индексом 0.
- Левый и правый дочерние узлы для родителя с индексом i имеют индексы 2i + 1 и 2i + 2 соответственно.
- Родитель элемента с индексом i имеет индекс $[(i-1)/2]$

Constructing heap from vector. (make_heap)

Алгоритм преобразования дерева(или вектора) в кучу: Проходя по вектору справа налево, сравниваем очередной элемент с максимальным из его дочерних узлов. Если один из дочерних узлов больше значения самого элемента, меняем их местами (возможно, несколько раз).

Анализ сложности кол-ва действий make_heap.

Пусть задан вектор с n элементами. Его можно представить в виде дерева, на последнем уровне количество элементов ≤ n, на предпоследнем ≤ n/2 и так далее.



Обозначим кол-во действий в худшем случае T(n). Каждый элемент на уровне k может потребовать до k операций для того, чтобы быть правильно размещённым в куче=>

$$\Rightarrow \quad T(n) \le 0 \cdot n + 1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + \cdots + k \cdot \frac{n}{2^k} + \cdots = \sum_{k=0}^{\infty} k \cdot \frac{n}{2^k} = n \sum_{k=0}^{\infty} \frac{k}{2^k} = c \cdot n$$

По признаку Коши ряд $\sum_{k=0}^{\infty} \frac{k}{2^k}$ сходится => T(n)= O(n)

Inserting:

Процесс вставки элемента в кучу начинается с добавления нового элемента в конец вектора, представляющего кучу. Это нарушает свойства кучи, поэтому необходимо восстановить их с помощью операции "всплытия" (sifting up).

1. Добавляем новый элемент в конец вектора.

2. Сравниваем элемент с его родительским узлом, Если новый элемент нарушает свойства кучи (например, в случае максимальной кучи — он больше родительского элемента), меняем их местами.

3. Повторяем процесс до тех пор, пока элемент не окажется на правильной позиции или не станет корнем.

Сложность:

- В худшем случае операция вставки требует O(logn) времени, так как элемент может "всплыть" от конца вектора до корня.

Removing Max:

1. Заменяем корень (максимальный элемент) на последний элемент в векторе.

2. Удаляем последний элемент

3. Сравниваем новый корень с его дочерними узлами. Если один из дочерних узлов больше (или меньше, в зависимости от типа кучи) текущего элемента, меняем их местами и повторяем процесс, пока не восстановим свойства кучи.

Сложность:

- Операция удаления (удаление корня и восстановление кучи) также требует O(logn) времени, так как элемент может "просочиться" от корня до листа.

Removing

1. **Поиск элемента**: Сначала нужно найти элемент, который мы хотим удалить. Это может быть выполнено через линейный поиск, так как в куче нет прямого способа найти произвольный элемент, как в отсортированном массиве.

2. **Заменить элемент последним элементом**: После того как элемент найден, мы заменяем его последним элементом в куче. Этот шаг аналогичен тому, как мы заменяем корень при удалении корня, но теперь мы удаляем произвольный элемент.

3. **Уменьшение размера кучи**: Удаляем последний элемент

4. **Восстановление свойств кучи**: После замены элемента последним элементом необходимо восстановить свойства кучи. В зависимости от положения элемента (находится ли он выше или ниже в дереве) мы выполняем одну из двух операций:

    - **Если элемент больше своего родителя (в случае максимальной кучи)** или **меньше своего родителя (в случае минимальной кучи)**, то выполняем операцию "всплытия" (sifting up).
    - **Если элемент меньше одного из своих детей (в случае максимальной кучи)** или **больше одного из своих детей (в случае минимальной кучи)**, то выполняем операцию "просачивания вниз" (sifting down).

5. **Повторяем процесс**: Если операция "просачивания вниз" или "всплытия" не восстановила свойства кучи, повторяем её до тех пор, пока элемент не окажется на правильной позиции.

Сложность:

- **Поиск элемента**: Линейная сложность O(n), так как для поиска произвольного элемента нужно пройти по куче.
- **Удаление элемента**: В худшем случае, операция "просачивания вниз" или "всплытия" имеет сложность O(logn), так как высота кучи — это logn.

Функция	Описание	Max-Heap (Сложность)	Min-Heap (Сложность)
Find	Поиск элемента в куче	O(n)	O(n)
Insert	Вставка элемента в кучу	O(logn)	O(logn)
Erase	Удаление элемента из кучи	O(logn)	O(logn)
FindMax	Поиск максимального элемента (для Max-Heap)	O(1)	O(n)
FindMin	Поиск минимального элемента (для Min-Heap)	O(n)	O(1)
heap to Vector	Преобразование кучи в вектор	O(n)	O(n)

std::priority_queue — это стандартная структура данных в C++, основанная на куче. Она предоставляет эффективный способ работы с приоритетными элементами. Использует Max-Heap по умолчанию, но можно изменить поведение для Min-Heap с помощью компаратора.

Основные методы:

1. push(element)

Добавляет элемент в очередь и восстанавливает структуру кучи.
Сложность: O(log n).

2. pop()

Удаляет элемент с наивысшим приоритетом (в Max-Heap — максимальный элемент).
Сложность: O(log n).

3. top()

Возвращает элемент с наивысшим приоритетом (не удаляя его).
Сложность: O(1).

1) std::**make_heap**(start.It, finish.It) — превращает контейнер в heap за O(n) шагов. (итераторы random access)
2) std::**pop_heap**(start.It, finish.It) получает итераторы на кучу, предполагая, что итератор уже указывает на созданную кучу. Он перемещает максимальный элемент к концу диапазона (finish.it - 1), а элементы в диапазоне от [start до finish.it - 1) превращает обратно в кучу.

3) std::**push_heap** принимает итераторы на диапазон [start.it, finish.it - 1), который представляет собой кучу, и добавляет новый элемент в кучу, поддерживая её свойства

4) std::**sort_heap**(start.It, finish.It)). Может получать функцию сравнения.

Как работает Heap Sort

1. **Создание кучи (heapify):**
    - Весь массив преобразуется в кучу (обычно max-heap для сортировки по возрастанию). Это делается с помощью функции `std::make_heap`.
2. **Извлечение максимального элемента:**
    - На каждой итерации максимальный элемент (корень) перемещается в конец массива. Затем последний элемент массива становится новым корнем, и выполняется перестройка кучи для сохранения её свойств.
3. **Повторение:**
    - Шаг 2 повторяется для оставшейся части массива (исключая уже отсортированные элементы).
4. **Результат:**
    - В результате массив становится отсортированным.

Функция	Что делает	Время выполнения
`std::make_heap`	Преобразует контейнер в кучу	O(n)
`std::push_heap`	Добавляет элемент в кучу и восстанавливает её	O(logn)
`std::pop_heap`	Извлекает максимальный элемент и восстанавливает кучу	O(logn)
`heap_sort`	Сортирует элементы с использованием кучи	O(nlogn)

1. **`std::make_heap` O(n):**
    - Эта операция преобразует произвольный контейнер в кучу (например, двоичную). Хотя может показаться, что требуется O(nlogn) операций (каждую вставку в кучу считать отдельно), на практике алгоритм выполняет меньшее количество операций благодаря тому, что элементы на нижних уровнях дерева требуют меньше перестановок.

2. **`std::push_heap` O(logn):**
    - При добавлении нового элемента в кучу он сначала добавляется в конец контейнера. Затем выполняется "всплытие" этого элемента вверх по дереву, чтобы восстановить свойства кучи. Максимальная высота двоичной кучи — logn, поэтому сложность логарифмическая.
3. **`std::pop_heap` O(logn):**
    - Максимальный элемент (в случае max-heap это корень) перемещается в конец контейнера. Затем последний элемент переносится в корень, и выполняется "просеивание" вниз для восстановления свойств кучи. Это также требует logn операций.
4. **`heap_sort` O(nlogn):**
    - Алгоритм сортировки использует свойства кучи:
        - Сначала создаётся куча из всех элементов (O(n)).
        - Затем максимальный элемент (корень) многократно извлекается (O(logn) за каждую итерацию), и свойства кучи восстанавливаются.
        - Всего n извлечений, что даёт общую сложность O(nlogn).