1. В ходе амортизационного анализа мы усредняем время по всем выполняемым операциям. Это можно использовать, чтобы показать, что если операция является дорогостоящей, то при усреднении по всей последовательности, средняя стоимость операции будет небольшой.

2. Если 3 метода для такого анализа:

Групповой анализ – мы выясняем суммарное время выполнения последовательности операций над структурой, пусть T(n), тогда амортизированная стоимость, приходящаяся на одну операцию, будет определяться соотношением T(n)/n.

Метод бух. учета -здесь у нас у каждой операции могут быть разные учетные стоимости, мы присваиваем операции какую-то амортизированную стоимость. Если она превышает реальное время работы. Мы присваем эту разницу как кредит в структуру. Другие операции у которых амортизированная стоимость превышае реальное время работы, могут скомпенсировать кредит.

Метод потенциалов – проще говоря здесь вместо кредита используется потенциал, это величина уже связана со структурой данных в целом, а не с отдельными ее объектами.

3. Динамическая таблица — это структура данных, реализованная как массив, размер которого может изменяться во время выполнения программы. В отличие от статических массивов, динамическая таблица может расширяться или сжиматься, чтобы вместить добавляемые или удаляемые элементы. Обычно она реализуется с помощью массива с выделенной памятью, размер которой удваивается при переполнении, а при значительном удалении элементов — уменьшается

Добавление элемента

Если в таблице есть свободное место, новый элемент просто добавляется в первый свободный индекс (операция за O(1)).

Если таблица заполнена, создаётся новый массив в два раза большего размера, все элементы копируются в него, и новый элемент добавляется (операция за O(n)O(n) в момент расширения).

Удаление элемента

Элемент удаляется по индексу, после чего все последующие элементы сдвигаются на одну позицию влево, чтобы сохранить непрерывность (операция за O(n)).

Если количество элементов становится значительно меньше размера массива (например, меньше четверти), таблица может быть сжата, создаётся массив меньшего размера и элементы копируются в него

4. Амортизированная сложность добавления

Большинство вставок выполняются за O(1), когда есть свободное место.

Расширение таблицы происходит редко, но требует копирования всех элементов, что стоит O(n).

Амортизированная стоимость вставки остаётся O(1), так как затраты на расширение распределяются на множество операций вставки.

Амортизированная сложность удаления

Удаление элемента требует сдвига элементов, что в худшем случае — O(n).

При значительном уменьшении числа элементов возможно сжатие таблицы, что также требует копирования элементов.

Однако, сжатие выполняется редко, поэтому амортизированная стоимость удаления также считается O(1) при правильной стратегии сжатия

5. Приоритетная очередь — это абстрактная структура данных наподобие стека или очереди, где у каждого элемента есть приоритет. Элемент с более высоким приоритетом находится перед элементом с более низким приоритетом. Если у элементов одинаковые приоритеты, они располагаются в зависимости от своей позиции в очереди.

6. Обычно приоритетные очереди реализуются с помощью куч: бинарной, биномиальной, фибонначиевой, толстая, тонкая и других их разновидностей.

7. Двоичная пирамида – представляет собой корневое бинарное дерево, которое хранит значения в вершинах. Существует требование кучи: значение хранящееся в вершине, больше или равно значению в его предке.

siftUp и siftDown рекурсивно чинит кучу, если требование кучи где-то не соблюдается. Оно меняет значения.

Алгоритм:

SiftDown

Работа процедуры: если i-й элемент меньше, чем его сыновья, всё поддерево уже является кучей, и делать ничего не надо. В противном случае меняем местами i-й элемент с наименьшим из его сыновей, после чего выполняем siftDown для этого сына. Процедура выполняется за время O(logn).

SiftUp

Работа процедуры: если элемент больше своего отца, условие 1 соблюдено для всего дерева, и больше ничего делать не нужно. Иначе, мы меняем местами его с отцом. После чего выполняем siftUp для этого отца. Иными словами, слишком маленький элемент всплывает наверх. Процедура выполняется за время O(logn).

8. Представим, что в массиве хранится дерево (a[0]− корень, а потомками элемента a[i] являются a[di+1]...a[di+d]). Сделаем siftDown для вершин, имеющих хотя бы одного потомка: от nd до 0,— так как поддеревья, состоящие из одной вершины без потомков, уже упорядочены.

9.

**int** dequeue():

**int** min = a[0]

a[0] = a[a.heapSize - 1]

a.heapSize = a.heapSize - 1

siftDown(0)

**return** min

**void** enqueue(key : **int**):

a.heapSize = a.heapSize + 1

a[a.heapSize - 1] = key

siftUp(a.heapSize - 1)

void decreasekey(A, change):

A.value += change

siftUp(A)