**Вопросы для самоконтроля**

**Задание 1**

Амортизационный анализ применяют для алгоритмов, которые запускаются несколько раз. Время алгоритма может сильно колебаться, но мы оцениваем среднее или суммарное время по всем запускам. Чаще всего алгоритм реализует какую-нибудь операцию над структурой данных.

Основной идеей амортизационного анализа является то, что любая трудоёмкая операция меняет состояние программы таким образом, что до следующей трудоёмкой операции обязательно пройдёт достаточно много мелких, тем самым «амортизируя» вклад трудоёмкой операции.

**Задание 2**

Есть три основных метода ведения амортизированного анализа: агрегирующий анализ, метод предоплаты и метод потенциалов. Все три дают правильный ответ и в конкретном случае обычно выбирается наиболее удобный:

1. При агрегирующем анализе вычисляется оценка сверху T(n)
2. В методе предоплаты каждой операции заранее приписывается амортизированная стоимость, которая может отличаться от её реальной стоимости. При этом более «дешёвые» операции обычно имеют амортизированную стоимость выше реальной, а более «дорогие» имеют амортизированную стоимость ниже реальной. За счёт этого при исполнении дешёвых операций накапливается некоторая сумма, которую можно «потратить» на то, чтобы выполнить операцию, чья амортизированная стоимость ниже реальной. Считается, что изначальная сумма равна нулю и если в течение алгоритма она не становится отрицательной, то суммарное время работы алгоритма будет равно разности суммарной амортизированной стоимости операций и накопленной суммы. Таким образом, амортизированная стоимость операций является оценкой сверху для реальной стоимости при условии, что накопленная сумма не становится отрицательной.
3. В методе потенциалов накопленная сумма вычисляется как функция («потенциал») от состояния структуры данных. Амортизированная стоимость при этом равна сумме реальной стоимости и изменения потенциала.

**Задание 3**

Динамические таблицы — это таблицы, которые автоматически обновляются на основе определенного запроса и целевой свежести, упрощая преобразование данных и управление конвейером без необходимости ручных обновлений или индивидуального планирования.

Динамические таблицы обновляются посредством автоматизированного процесса обновления, который регулярно выполняет запрос на преобразование. Этот процесс вычисляет изменения, внесенные в базовые объекты, и объединяет их в динамическую таблицу, используя вычислительные ресурсы, связанные с таблицей.

Частота обновления и актуальность данных определяются целевым запаздыванием, указанным при создании динамической таблицы, которое определяет, насколько актуальными должны быть данные. Процесс обновления использует указанное целевое запаздывание для планирования обновлений. Например, целевое запаздывание в пять минут гарантирует, что данные в динамической таблице не будут отставать от обновлений базовой таблицы более чем на пять минут. Более длительные целевые запаздывания можно использовать для снижения затрат, когда обновления данных в режиме, близком к реальному времени, не требуются. Например, если данные вашей динамической таблицы должны отставать от обновлений базовых таблиц максимум на один час, вы можете указать целевую актуальность в один час (вместо пяти минут) для снижения затрат.

**Задание 4**

**Добавление в конец**: в среднем выполняется за константу, кроме случаев, когда таблица начинает автоматически расширяться, тогда на копирование элементов уходит за полиномиальное время. Если проводить амортизированный анализ, то в среднем случае сложность константная.

**Добавление по индексу**: происходит сдвиг вправо на одну единицу от входной позиции. В худшем случае это будет добавление в начало, что приведет к временной сложности – полиномиальной. Копирование также будет необходимо, но он выполняется особняком, поэтому на сложность не влияет.

**Удаление с конца**: всегда выполняется за константное время, т.к. процесс похож на добавление, но только копирование не происходит.

**Удаление по индексу**: выполняется за полиномиальное время, т.к. требует сдвига влево от конкретной позиции. Худший случай, все тот же удаление первого элемента.

**Задание 5**

Очередь с приоритетами – это тип данных, являющееся наследником простой очереди, то бишь коллекции (линейным типом данных). Также, как и обычная очередь, приоритетная очередь выдает элемент с одного конца, и добавляет элемент с другого. Главное отличие от обычной очереди заключается в том, что в узлах приоритетной очереди содержится атрибут приоритет, который и распределяет внутри самой очереди последовательность входа и выхода. Т.е. у кого приоритет из всех элементов очереди выше, тот первым из очереди и выйдет.

**Задание 6**

Очередь с приоритетами может быть основан на линейных типах данных (массив и списки), на двоичных деревьях (бинарных) или на кучах.

Вот сложность по времени для каждой структуры:

|  |  |  |  |
| --- | --- | --- | --- |
| Операции | peek() | insert() | delete() |
| List | O(1) | O(n) | O(1) |
| Binary Heap | O(1) | O(log n) | O(log n) |
| Binary Search Tree | O(1) | O(log n) | O(log n) |

**Задание 7**

Binary Heap – это структура данных, элементы которой представляют собой узлы с содержимыми данными (ключом) и каким-то весом узла (значение). И для бинарной кучи должны выполняться следующие три условия:

1. Значение в любой вершине не больше (если куча для минимума), чем значения её потомков.
2. На **i-ом** слое **2i** вершин, кроме последнего. Слои нумеруются с нуля.
3. Последний слой заполнен слева направо.

Если значение измененного элемента увеличивается, то свойства кучи восстанавливаются функцией **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).

Если значение измененного элемента уменьшается, то свойства кучи восстанавливаются функцией **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)

**Задание 8**

Дан массив a[0...n−1]. Требуется построить d-кучу с минимумом в корне. Наиболее очевидный способ построить такую кучу из неупорядоченного массива — сделать нулевой элемент массива корнем, а дальше по очереди добавить все его элементы в конец кучи и запускать от каждого добавленного элемента siftUp. Временная оценка такого алгоритма O(nlogn). Однако можно построить кучу еще быстрее — за O(n).

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

**Задание 9**

**Enqueue (вставка элемента):**

* Добавляем элемент в конец массива
* Просеиваем вверх до восстановления свойств кучи – siftUp ()
* Сложность: O(logn)

**Dequeue (извлечение максимума):**

* Берем элемент из корня (максимум)
* Перемещаем последний элемент в корень
* Просеиваем вниз до восстановления свойств кучи – siftDown ()
* Сложность: O(logn)

**IncreaseKey (увеличение приоритета):**

* Увеличиваем значение элемента
* Просеиваем вверх (так как новый ключ может быть больше родителя) – siftUp ()
* Сложность: O(logn)

.

.