Алгоритм упорядоченного циклического массива без блокировок

При отображении графиков по котировкам возникает задача одновременной записи (из одного потока) и чтения (из нескольких) циклического массива данных, представляющих собой key-value значения - по сути, "развернутая" хэш-таблица. Помимо редких операций вставки в начало и частых в конец, периодически возникает необходимость вставки в середину и итерации по массиву в обратном порядке, что приводит к необходимости держать массив отсортированным.

В общем случае подобная MWMR структура данных реализуется на основе сбалансированного дерева, очереди с приоритетами или скип-листа, что в данном случае излишне, поскольку писатель всегда один => проблем с расширением не возникает. Кроме того, хочется использовать массив из соображений эффективной работы с памятью.

Задача: разработать линеаризуемую версию алгоритма SWMR циклического отсортированного массива без блокировок.

Подходы

В основе лежит идея того, что если массив всегда сдвигать только в одну сторону, то итерация в эту сторону будет предоставлять гарантию, что все элементы, которые были в массиве на момент создания итератора, будут обойдены. Изначально она описана тут:

https://schani.wordpress.com/2007/08/18/a-reader-lock-free-sorted-array/

Поскольку элементы сложные (key - struct value), нужно обеспечить версионирование ячеек с ключами (или всего массива целиком) - нечетные версии сигнализируют о записи в процессе, четные - об окончании записи. Также видимо перед перемещением в ячейку нужно писать специальный маркерный ключ.

Алгоритм должен реализовывать следующий интерфейс:

```
public interface TimeSortedArray<V> {
    /**
     * Returns N -- current size. Must work in O(1).
     */
    int size();
    /**
     * Returns value for a given time. Must work in O(log(N)) using binary search.
     */
    V get(long time);
    /**
     * Updates value for a given time. Must work in O(1) amortized when adding item with lowest of highest time,
     * use exponential resizing to ensure amortize cost guarantee. Can be O(N) worst-case when resizing or
     * when putting into the middle.
     */
    void put(long time, V value);
    /**
     * Removes value for a given time. Logical removal is appropriate. Ideally, should be O(1) worst-case
     * when removing item with the lowest time.
     */
    void remove(long time);
    /**
     * Iterates items backwards in time. Must work in O(1) to start and for each step and guarantee
     * that no time is skipped despite concurrent updates with put or remove.
     */
    TimeIterator<V> iterateBackwards();
    interface TimeIterator<V> {
        /**
         * Moves iterator to next time and returns it. Result is {@link Long#MIN_VALUE} when there are not more
items
         * to iterate. Must work in O(1).
         */
        long nextTime();
        /**
         * Returns current value that corresponds. Updates after {@link #nextTime()} invocation.
         */
        V value();
    }
}
```