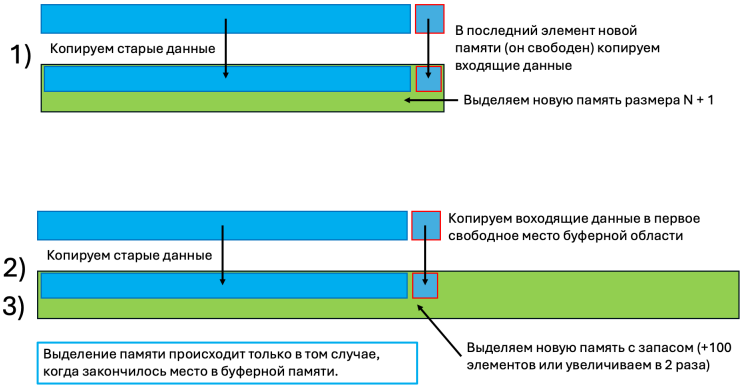
Лабораторная II. Динамическая память.

Увеличивающийся буфер (3 - 5)

Представим, что у нас возникла задача: добавлять элементы в конец массива по необходимости. Мы не хотим заранее занимать некоторый «максимально возможный» размер памяти, так как это может оказаться чрезвычайно накладным, если в итоге использоваться будет лишь малая часть выделенной памяти. Хороший выход - использовать динамическую память. Есть несколько простых стратегий, которые можно использовать:

1. добавлять строго по одному элементу;
2. создавать запас пустых элементов фиксированного размера (например всегда занимать по 100 элементов);
3. использовать мультипликативное увеличение памяти, например увеличивать размер памяти вдвое.

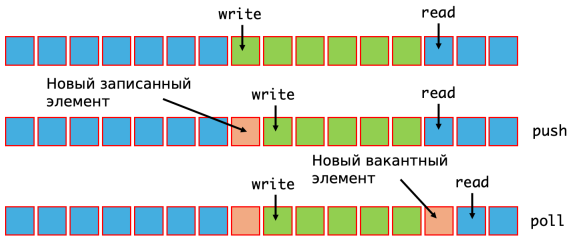


Первая стратегия хороша тем, что все элементы в памяти всегда инициализированы и их количество точно равно количеству элементов в выделенной памяти. Однако мы получаем асимптотическую сложность добавления элемента $O(N)$, что достаточно медленно. Второй и третий способы подразумевает, что количество хранящихся в памяти элементов и количество элементов, которое можно разместить в памяти, -

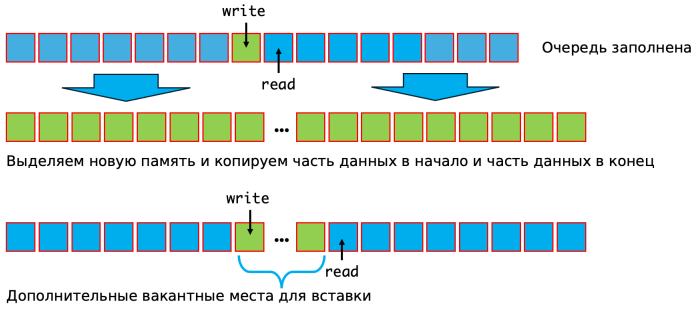
это два разных числа, их нужно где-то отдельно хранить. Тем не менее мы ожидаем выигрыш в асимптотической сложности: часть ситуаций, когда нам нужно добавление элементов пройдёт без выделения памяти и копирования. Мы можем ожидать улучшения асимптотической сложности при частом добавлении. Такая сложность называется «амортизированной», т.е. для одного элемента это может быть или $O(N)$, или $O(1)$ операций, а для большого количества элементов сложность может отличаться и от первой, и от второй.
Напишите программу, которая оценит асимптотическую сложность во всех трёх случаях прямым измерением времени на примере массива целых чисел. Представьте результаты в виде графиков в логарифмическом масштабе, оцените степень полиномиальной сложности «методом наименьших квадратов».

Очередь (6 - 8)

Очередь - это стратегия получения и добавления данных, которая гарантирует, что данные при обращении будут получены в том же порядке, в котором добавлены, говорят «первый вошёл, первый вышел» (First In First Out). Мы можем организовать подобную стратегию внутри массива. Запомним два индекса: индекс `poll_idx`, по которому мы извлекаем данные и индекс `push_idx`, по которому мы добавляем данные. Добавим правило цикличности индексов: если следующий индекс превышает максимальный допустимый индекс для массива, то мы подразумеваем вставку или чтение по нулевому индексу.



Очевидно, что если мы ничего не будем читать из очереди, а будем только добавлять, то возникнет конфликт индексов: мы будем должны добавить элемент в нулевой индекс, но он уже занят элементом, который необходимо прочитать. В этом случае мы будем увеличивать количество элементов в массиве **между** (как это правильно сделать?) индексами вставки и чтения, причём сама стратегия увеличения размера массива - это оптимальная стратегия, полученная в первой части лабораторной работы.



Методы для работы с очередью:

```
1 void queue_new(std::size_t &read, std::size_t &write, int* &data);
2 bool queue_poll(std::size_t &read, std::size_t &write, int* &data, int &element);
3 bool queue_push(std::size_t &read, std::size_t &write, int* &data, int element);
4 void queue_del(int* &data);
```

Обратим внимание на функции `queue_poll` и `queue_push`. Они могут завершиться провалом: `queue_poll` можно выполнить на пустой очереди, тогда нужно вернуть `false`, `queue_push` может потребовать увеличения памяти, но вызов `new` может провалиться, тогда нужно вернуть `false`.
Реализуйте описанную очередь и вычислите амортизированную асимптотическую сложность операций `queue_push` и `queue_poll`. Результаты представьте в виде графиков.

Двунаправленная очередь элементов некоторого произвольного типа (9 - 10)

Представим, что нам нужно добавлять и удалять элементы с обоих концов нашей структуры данных. Т.е. наши операции

`push` и `poll` можно будет выполнять с разных сторон очереди:

- `push_front` – добавить в начало очереди;
- `poll_front` – взять из начала очереди;
- `push_back` – добавить в конец очереди;
- `poll_back` – взять из конца очереди.

Помимо этого мы хотим работать не только с целыми числами, а с объектами произвольного типа. Единственное ограничение - все объекты должны быть одного размера в байтах. В этом случае мы уже не можем оперировать указателями на заданный тип `int`, мы можем использовать только адрес `void *` и размер данных в байтах. Например сигнатура функции добавления элемента в начало очереди может выглядеть так:

```
bool push_front(
    std::size_t const element_size,
    std::size_t &read_element_offset,
    std::size_t &write_element_offset,
    void * &data,
    void const *element_address // address of the data to copy to the queue
);
```

Реализуйте и протестируйте описанную структуру данных.