```cpp
template<typename T>
    class sequential_queue5 {

        struct node
        {
            std::shared_ptr<T> data;
            std::unique_ptr<node> next;

            node()
            {}

            node(T _data) : data(std::move(_data))
            {
            }
        };

        std::unique_ptr<node> head;
        node* tail;

        std::mutex head_mutex;
        std::mutex tail_mutex;

        std::condition_variable cv;

        node* get_tail()
        {
            std::lock_guard<std::mutex> lg(tail_mutex);
            return tail;
        }

        std::unique_ptr<node> wait_pop_head()
        {
            //? protect head node with mutex and unique_lock
            std::unique_lock<std::mutex> lock(head_mutex);

            //? Need to wait for the condion variable
            //* (maybe someone pushing to the queue at the moment)
            //? and also check if there is something in the queue
            //* (head.get() == get_tail()) - when head and tail points to the dummy node
            //! do we stick in the loop until one element will appear in the queue?
            //! do we need this?
            head_condition.wait(lock, [&] { return head.get() != get_tail(); });

            //! 'const' cast issue here, remvoe const (copy ellision won't work on const)
            // std::unique_ptr<node> const old_head = std::move(head);
            std::unique_ptr<node> old_head = std::move(head);

            head = std::move(old_head->next);
            return old_head;    //! be carefull, non const variable needed to allow copy ellision
        }

    public:
        sequential_queue5() :head(new node), tail(head.get())
        {}

        void push(T value)
        {
            std::shared_ptr<T> new_data(std::make_shared<T>(std::move(value)));
            std::unique_ptr<node> p(new node);
            node* const new_tail = p.get();
            {
                std::lock_guard<std::mutex> lgt(tail_mutex);
                tail->data = new_data;
                tail->next = std::move(p);
                tail = new_tail;
            }

            cv.notify_one();
        }

        std::shared_ptr<T> pop()
        {
            std::lock_guard<std::mutex> lg(head_mutex);
            if (head.get() == get_tail())
            {
                return std::shared_ptr<T>();
            }
            std::shared_ptr<T> const res(head->data);
            std::unique_ptr<node> const old_head = std::move(head);
            head = std::move(old_head->next);
            return res;
        }

        std::shared_ptr<T> wait_pop()
        {
            //! std::unique_ptr and not std::unique_lock
            std::unique_ptr<node> old_head = wait_pop_head();    //! no need std::move() because of copy ellision
            return old_head ? old_head->data : std::shared_ptr<T>();
        }

        // Printer method: prints cells from top to bottom
        void printData();
    };

    template <typename T>
    inline void sequential_queue5<T>::printData()
    {
        if (head.get() == get_tail())
        {
            std::cout << "Queue is empty...\n";
            return;
        }

        std::lock_guard<std::mutex> hlg(head_mutex);

        node* current = head.get();
        std::cout << "Queue from top to bottom...\n";
        int index{};
        while (current->data != nullptr)
        {
            std::cout << "current: " << current << ", value [" << index++ << "]: " << *(current->data) << std::endl;
            current = (current->next).get();
        }
        std::cout << "End of the queue...\n";
    }
```