Data Structures and Algorithms

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Waiting Lines in a System

Context: we need to store incoming tasks in a system/program until they can be processed.

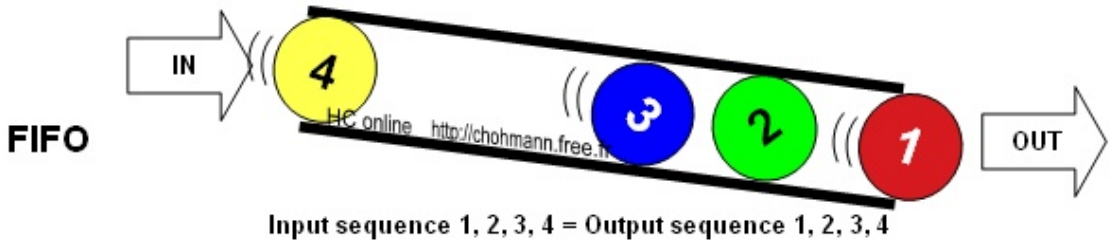
Question: what should be the first pending task to be processed?

- Several possibilities: LIFO/FIFO/priorities/...
- Implementation can be static or dynamic
- Time/Space complexity?

Who's next?

Choosing the next task: **FIFO**

- **F**irst **I**n **F**irst **O**ut



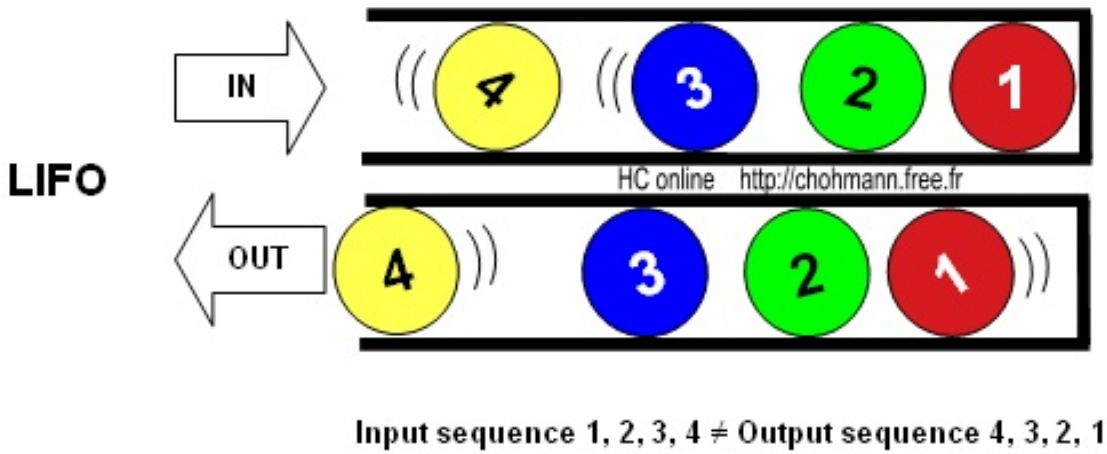
Examples:

- Task Scheduler
- Connections to a Server
- Waiting for a movie, at the bank, etc.

Who's next?

Choosing the next task: **LIFO**

- **L**ast **I**n **F**irst **O**ut

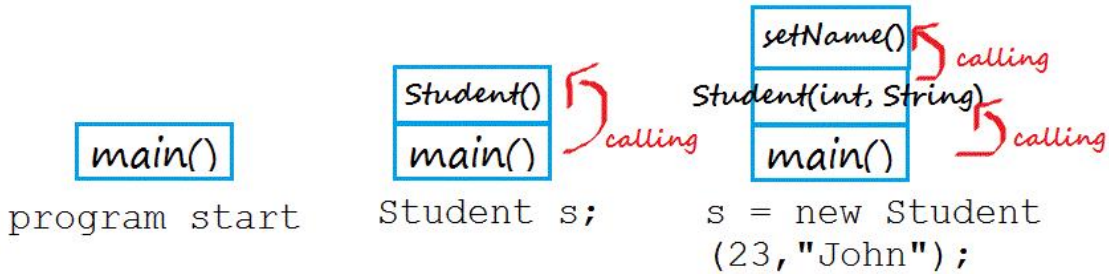


Examples:

- Washing the plates =)
- Your Web browser history, etc.

More on LIFO

Executing a program



- Program starts with `main()`
- First instruction detected: the next line of `main()` is pending
- First instruction calls a subprogram: the next line is pending
- ...

Application: **Recursivity**

Data Structures

- **Stack**: implements strategy LIFO
  - **void** `push(int)`: insertion of a new element – $\mathcal{O}(1)$
  - **int** `pop()`: returns and remove the latest element added to the structure – $\mathcal{O}(1)$
- **Queue**: implements strategy FIFO
  - **void** `enqueue(int)`: insertion of a new element – $\mathcal{O}(1)$
  - **int** `dequeue()`: returns and remove the earliest element added to the structure – $\mathcal{O}(1)$
- Deque: supports both strategies
  - Most queue implementations keep track of both the earliest and latest elements in the structure, and thus they can be easily adapted to that case.

Extended list of operations

Available in the implementations of most programming languages.

- **int** `size()`. Keep track of the number of elements in a separate field. Increment this field after each `push`/`enqueue` operation and decrement its value after each `pop`/`dequeue` operation. – $\mathcal{O}(1)$.

- **bool** `empty()`. Simply check whether the size equals 0. – $\mathcal{O}(1)$.

- **int** `peek()`. Returns, but does not remove the latest element added in a stack (resp., the earliest element added in a queue) – $\mathcal{O}(1)$.

  $\longrightarrow$ Store the peek element in a separate variable.

  $\longrightarrow$ Before a `push`/`enqueue` operation, the former peek element (if any) is inserted into the stack/queue.

  $\longrightarrow$ After a `pop`/`dequeue` operation, the new peek element (if any) is extracted from the stack/queue.

Dynamic implementation: Stack

with a Singly Linked List

We keep adding/removing elements at the **head** of the list.

```
typedef list<int> Stack;

//Complexity: O(1)
int push(Stack& s, int e) { s.push_front(e); }

//Complexity: O(1)
int pop(Stack& s) {
   int e = s.front();
   s.pop_front();
   return e;
}
```

Dynamic Implementation: Queue

With a Singly Linked List + pointer to the tail

- We add elements at the **tail** of the list.
- We remove elements at the **head** of the list.

Remark: updating the tail takes $\mathcal{O}(n)$ time *after removal* but only $\mathcal{O}(1)$ after insertion.

```
typedef list<int> Queue;
//Complexity: O(1)
int enqueue(Queue& q, int e) { q.push_back(e); }
//Complexity: O(1)
int dequeue(Queue& q) {
   int e = q.front();
   q.pop_front();
   return e;
}
```

Dynamic implementation: Deque

with a Doubly-Linked List

```
typedef list<int> Deque; //The C++ list is doubly-linked

void push_front(Dequeue& d, int e) { d.push_front(e); } //O(1)

void push_back(Dequeue& d, int e) { d.push_back(e); } //O(1)

int pop_front(Dequeue& d) { //O(1)
    int e = d.front();
    d.pop_front();
    return e;
}

int pop_back(Dequeue& d) { //O(1)
    int e = d.back();
    d.pop_back();
    return e;
}
```

Remark: If we were using a singly linked list, then pop_back() would run in $\mathcal{O}(n)$ time.

Static implementation

Some motivations

- Constant-access to any element is not very useful for Stacks/Queues...

- Nevertheless, arrays require less memory storage than lists (*i.e.*, we avoid storing the information zone).

$\Longrightarrow$ If a good upper bound on the max. capacity of the data structures is known, then it might be more advantageous to implement it using an array rather than a list.

- Objective: avoid `push_back` and `push_front` operations (worst case complexity in $\mathcal{O}(n)$).

$\longrightarrow$ We should resize only if the structure has reached its max. capacity.

Static implementation: Stack

We keep track of the position of the peek (last element) in a separate variable.

```
struct Stack {
    vector<int> container;
    int last = -1;
}

//O(1)
void push(Stack& s, int e) { s.container[++s.last] = e; }

//O(1)
int pop(Stack& s) {
    int e = s.container[s.last--];
    return e;
}
```

Static implementation: Queue

First try: keep track of the position of the earliest and latest elements in separate variables

```
struct Queue {
   vector<int> container;
   int first = 0, last = -1;
}

//O(1)
void enqueue(Queue& q, int e){ q.container[++q.last] = e; }

//O(1)
int dequeue(Queue& q) { return q.container[q.first++]; }
```

Drawbacks...

Let $n =$ `container.size()` be the max. capacity of the queue.

- We add n elements in the queue

  `first = 0, last = n-1`

- We remove all elements from the queue

  `first = n, last = n-1`

- If we try to insert a new element, then an error occurs (out of range).

Whereas the queue is empty!

Circular Queue

We use modular arithmetic: whenever we reach the back of the vector, we reset to 0 in order to access to the emptied space at the front.

```
//Complexity: O(1)
void enqueue(Queue& q, int e) {
   if(++q.last == q.container.size()) { q.last = 0; }
   q.container[q.last] = e;
}

//Complexity: O(1)
int peek(Queue& q) {
   int e = q.container[q.first++];
   if(q.first == q.container.size()) { q.first = 0; }
   return e;
}
```

Circular Queue

We use modular arithmetic: whenever we reach the back of the vector, we reset to 0 in order to access to the emptied space at the front.

```
//Complexity: O(1)
void enqueue(Queue& q, int e) {
   if(++q.last == q.container.size()) { q.last = 0; }
   q.container[q.last] = e;
}

//Complexity: O(1)
int peek(Queue& q) {
   int e = q.container[q.first++];
   if(q.first == q.container.size()) { q.first = 0; }
   return e;
}
```

Remark: still border effects (if full capacity)...

FIFO vs. LIFO: Simulations

Our goal: simulating a stack with several queues, and vice-versa.

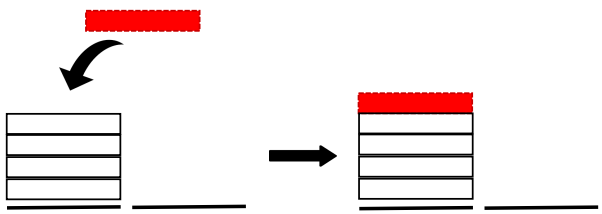
. . . But why??

- Because its funny!
- Purely functional implementation: variables in a functional language are immutable (data persistence). Stacks (as singly-linked lists) have a purely functional implementation, but Queues NOT.
- Expressive power (*i.e.*, which one is the strongest?)

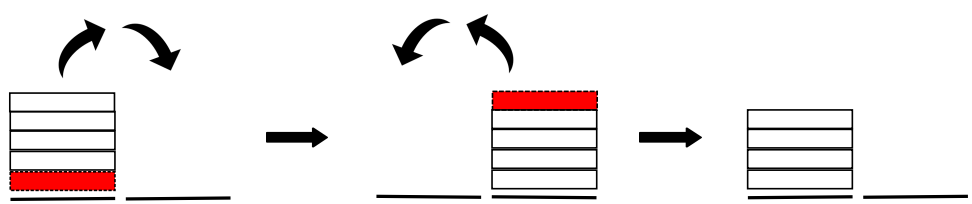
Simulation of a Queue with two Stacks

- Insert all elements into a Stack `s1`
- For simulating a `dequeue` operation, move all elements from `s1` to an auxiliary stack `s2`. Then, push all remaining elements back to `s1`

Add an element:



Get (and remove) the **first** element:



Observation: `dequeue` in $\mathcal{O}(n)\ldots$

Simulation in **Constant Amortized Time**

Intuition: we needn't push back all elements to `s1`!

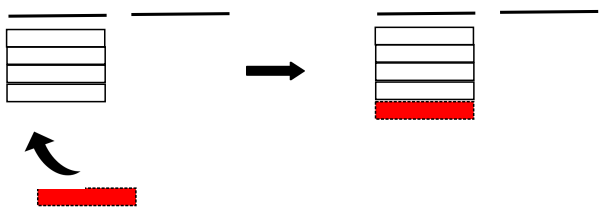
Potential function: # elements in `s1`

```
struct Queue { stack<int> s1, s2; };

//Complexity: O(1); Change of potential: +1
//Amortized complexity: O(1)
void enqueue(Queue& q, int e) { q.s1.push(e); }

//Complexity: O(s1.size()) if s2 is empty (otherwise, in O(1)
//Change of potential: -s1.size() if s2 is empty (otherwise, 0)
//Amortized complexity: O(1)
int dequeue(Queue& q){
    if(s2.empty())
        while(!s1.empty()) { s2.push(s1.top()); s1.pop(); }
    int e = s2.top(); s2.pop();
    return e;
}
```

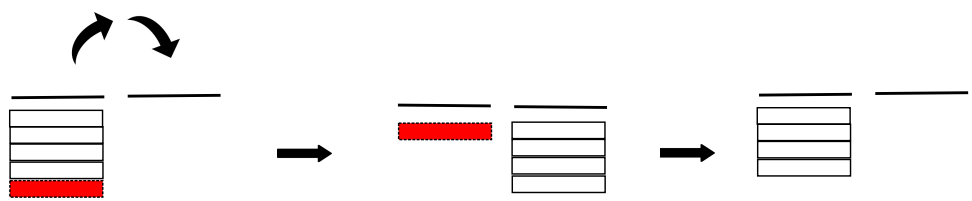
Simulation of a Stack with two Queues

- Insert all elements into a Queue `q1`
- For simulating a `pop` operation, move all *but one* elements from `q1` to an auxiliary queue `q2`. Then, enqueue all remaining elements back to `q1`

Add an element:



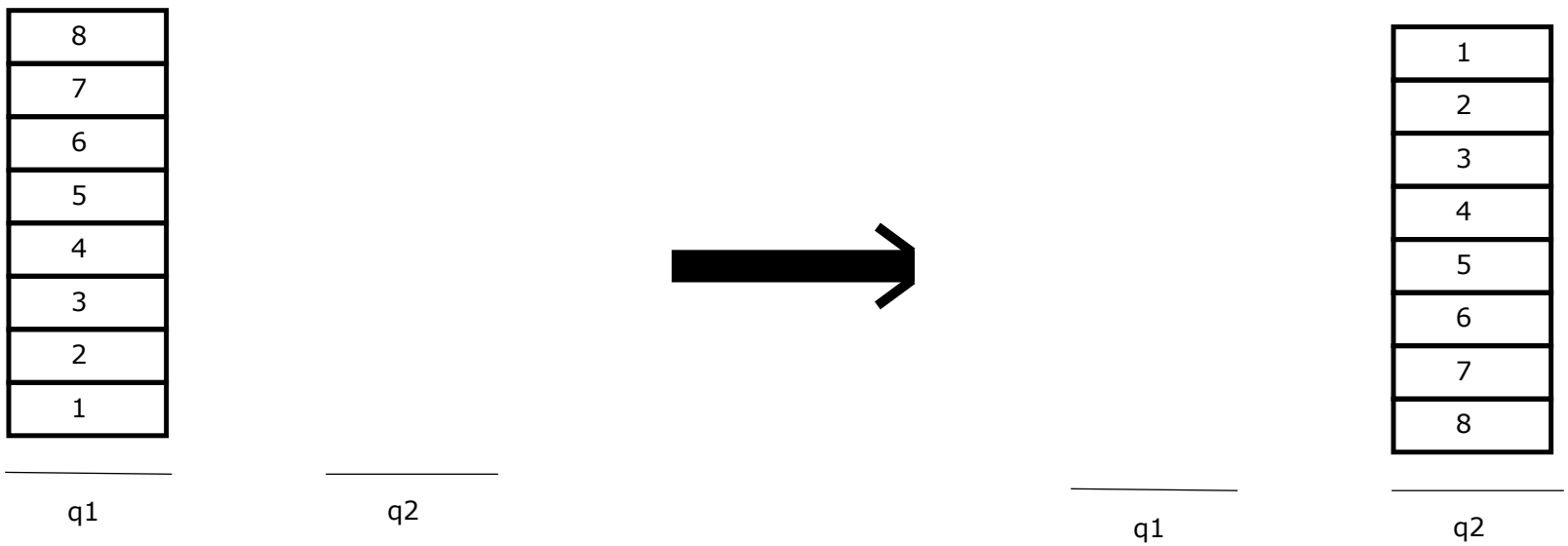
Get (and remove) the **last** element:



Alternatively, we may switch the respective roles of `q1`,`q2` after each `pop` operation. **However, the same trick as before does not work**: (amortized) complexity is in $\mathcal{O}(n)$.

Simulation in **Logarithmic Amortized Time**

Intuition: we use q2 for the pop operations. If it is empty, then we move all elements from q1 to q2 **in reverse order** (w.r.t. insertion time).



`reverse(q1,q2)` in $\mathcal{O}(n \log n)$ time $\Longrightarrow$ Amortized complexity in $\mathcal{O}(\log n)$

Queue reversal

Assumption: n is a power of two (else, add sufficiently many dummy element in q1)

- For $i = 1, 2, \ldots, \log n$ we switch consecutive blocks of size $n/2^i$.

For $j = 1, 2, \ldots, 2^{i-1}$ **do**

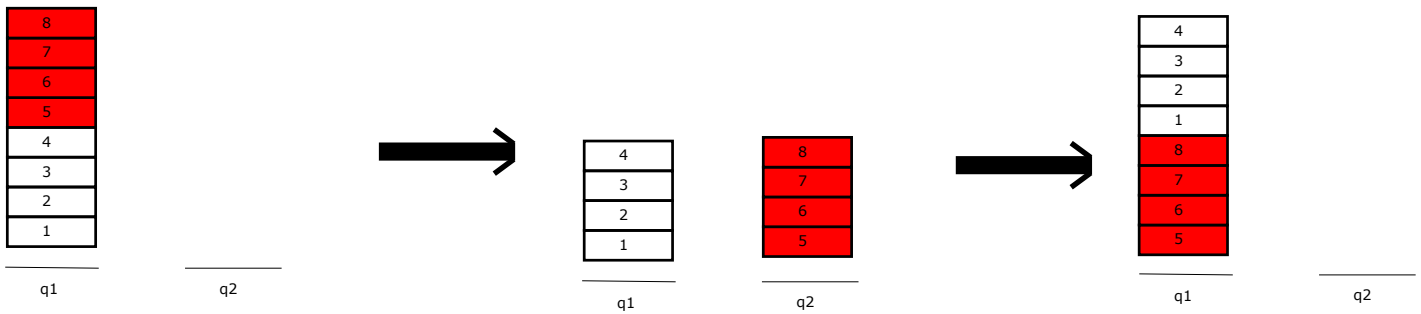
  - Move $n/2^i$ elements from q1 to q2
  - Move $n/2^i$ elements from (the front of) q1 to (the back of) q1
  - Move all elements from q2 to q1

Complexity: $\mathcal{O}(n)$ per loop. Hence, it is in $\mathcal{O}(n \log n)$.

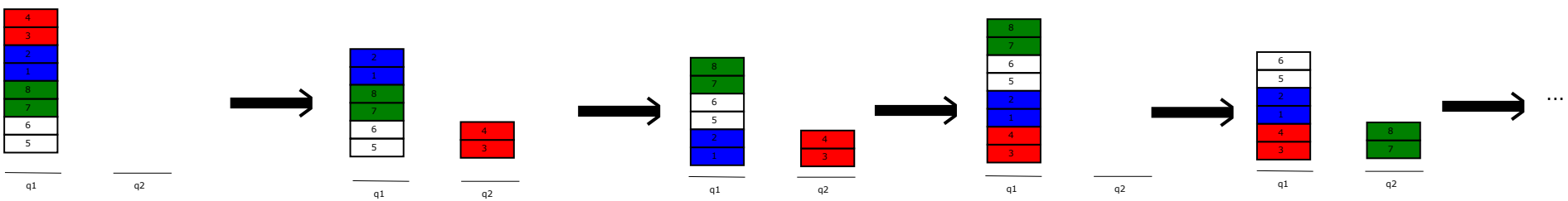
Observation: Inspired from radix sort...

Example

- $i = 1$



- $i = 2$



Simulation of a Queue with **six** Stacks

Sorry for that...

Motivation: Worst-case $\mathcal{O}(1)$ per operation ("real-time").

Reminder: Our previous two-stack implementation consists of a "head stack" `s2` (for `dequeue`) and a "tail stack" `s1` (for `enqueue`).

Improvement strategy: do not wait the very last moment (*i.e.*, when `s2` is empty) to move all elements from `s1` to `s2`.

The queue now has two possible modes:

- Normal mode: we proceed as before (`enqueue` $\rightarrow$ `s1.push` and `dequeue` $\rightarrow$ `s2.pop`)
- Recopy mode: we move all elements from `s1` to `s2`.

  1) We distribute the copy of `s1` (reversed) to `s2` throughout several operations.

  2) *Upcoming operations (until we completed the copy) are performed using additional stacks*

Recopy mode

- We enter in recopy mode as soon as `s1.size()` $>$ `s2.size()`.

— Remark: we can set a boolean flag to 1 in order to indicate whether we are in recopy mode. —

Intuition:

- When we enter in recopy mode we have `s1.size()` == `s2.size()+1`
- In this situation, the reverse copy of `s1` can be done in $\mathcal{O}$(`s2.size()`) time.
- We can distribute the copy throughout `s2.size()` operations (this ensures that less than `s2.size()` operations `dequeue` have occurred, and so that we needn't terminate the recopy process until that point)
- At the end of the recopy process, the new head stack contains at least `s1.size()` elements and at most `s2.size()` operations `enqueue` have occurred $\Longrightarrow$ we exit the recopy mode.

Four more stacks

• We cannot reverse copy `s1` directly into `s2` because `s2` may not be empty. $\Longrightarrow$ **intermediate stack** `s3`.

- Move all elements from `s2` (reversed) to `s3`
- Move all elements from `s1` (reversed) to `s2`
- Move back all elements from `s3` to `s2`.

• While we are in recopy mode, we cannot use `s1` for the operations `enqueue` $\Longrightarrow$ **new tail stack** `s4` (after we exit recopy mode, `s1 = s4`).

• While we are in recopy mode, we cannot use `s2` for the operations `dequeue` $\Longrightarrow$ **copy head stack** `s5`.

• The copy of the new head stack (resulting from the reverse copy from `s1` to `s2`) must be also created during the recopy mode $\Longrightarrow$ **new copy head stack** `s6`

Sketch of implementation

```
struct Queue {

    bool recopy_mode = 0;

    //Normal mode
    stack<int> s1; //tail stack
    stack<int> s2; //head stack

    //Recopy mode
    stack<int> s3; //intermediate stack for the reverse copy
    stack<int> s4; //new tail stack
    stack<int> s5; //copy of former head stack
    stack<int> s6; //copy of new head stack
    int nr_dequeue = 0; //Number of dequeue operations
};
```

Sketch of implementation cont'd

```
void enqueue(Queue& q, int e){
    if(!q.recopy_mode) {
        q.s1.push(e);
        if(q.s1.size() > q.s2.size()) { q.recopy_mode = 1; }
    } else {
        q.s4.push(e);
        //+ continue recopy process throughout O(1) operations
}}

int dequeue(Queue& q) {
    if(!q.recopy_mode) {
        int e = q.s2.top(); q.s2.pop();
        if(q.s1.size() > q.s2.size()) { q.recopy_mode = 1; }
        return e;
    } else {
        q.nr_dequeue++; //one less element to recopy from s2
        int e = q.s5.top(); q.s5.pop();
        //+ continue recopy process throughout O(1) operations
        return e;
}}
```

Sketch of implementation cont"d

- The full recopy process

```
//Move all elements from s2 to s3
while(!q.s2.empty()){ q.s3.push(q.s2.top()); q.s2.pop(); }

//Move all elements from s1 to both s2 and s6
while(!q.s1.empty()) {
    q.s2.push(q.s1.top());
    q.s6.push(q.s1.top());
    q.s1.pop();
}

//Move back all elements from s3 (except those dequeued meanwhile)
while(q.s3.size() > q.nr_dequeue) {
    q.s2.push(q.s3.top());
    q.s6.push(q.s3.top());
    q.s3.pop();
}

//Exit recopy mode
q.s1 = q.s4; q.s4 = stack<int>();
q.s5 = q.s6; q.s6 = stack<int>(); //The destruction of former s5 can also be distributed
q.recopy_mode = 0; q.nr_dequeue = 0;
```

Questions