Проблема индексации NFT айтемов (Логическая ошибка)

Тут, в файле stbl_staking.fc, происходит минт айтема, если стейка ещё не было

```
;; if hasn't staked before
if (staker::staked_a == 0) {
    cell item_message = begin_cell()
        .store_slice(staker_address)            ;; item owner
        .store_ref(self::nft_content)   ;; item content
        .store_slice(self::initializer) ;; editor
    .end_cell();
    cell nft_body = pack_nft_mint_message(msg::query_id, CONF::NFT_ITEM_AMOUNT, item_message);
    cell nft_mint_msg = delib::int_msg(self::nft_collection_address, false, CONF::NFT_MINT_AMOUNT, nft_body, null());
```

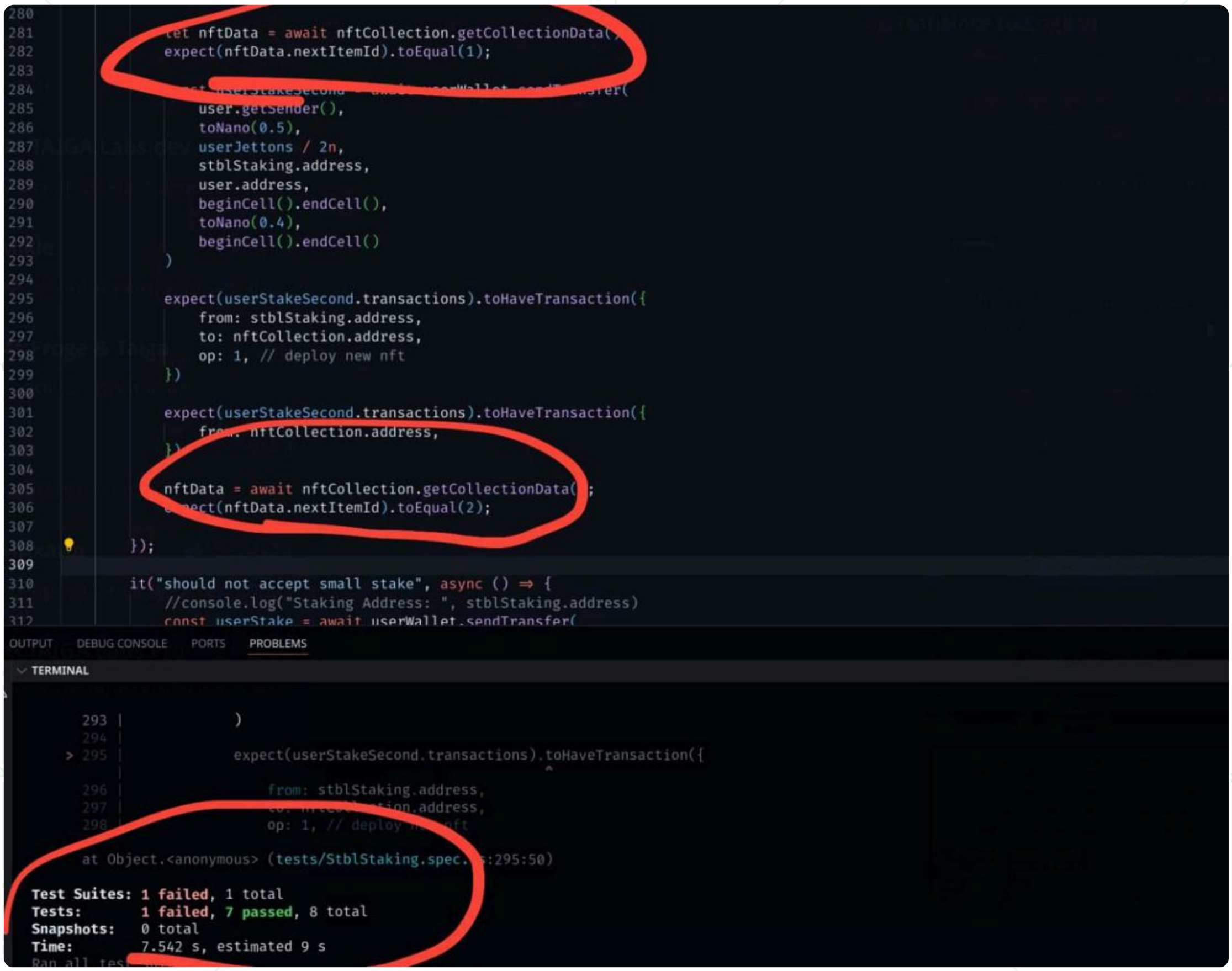
Проблема в том, что в файле helpers.fc, в функции, которая формирует сообщение, отправляемое на контракт коллекции при минте айтема, индекс представлен константой - 0 (95 строка)

```
cell pack_nft_mint_message(int query_id, int item_value, cell nft_message) inline {
    return begin_cell()
        .store_uint(op::nft::mint, 32)
        .store_uint(query_id, 64)
        .store_int(0, 64) ;; item_index
        .store_coins(item_value)  ;; ton to send to the item
        .store_ref(nft_message)
        .end_cell();
}
```

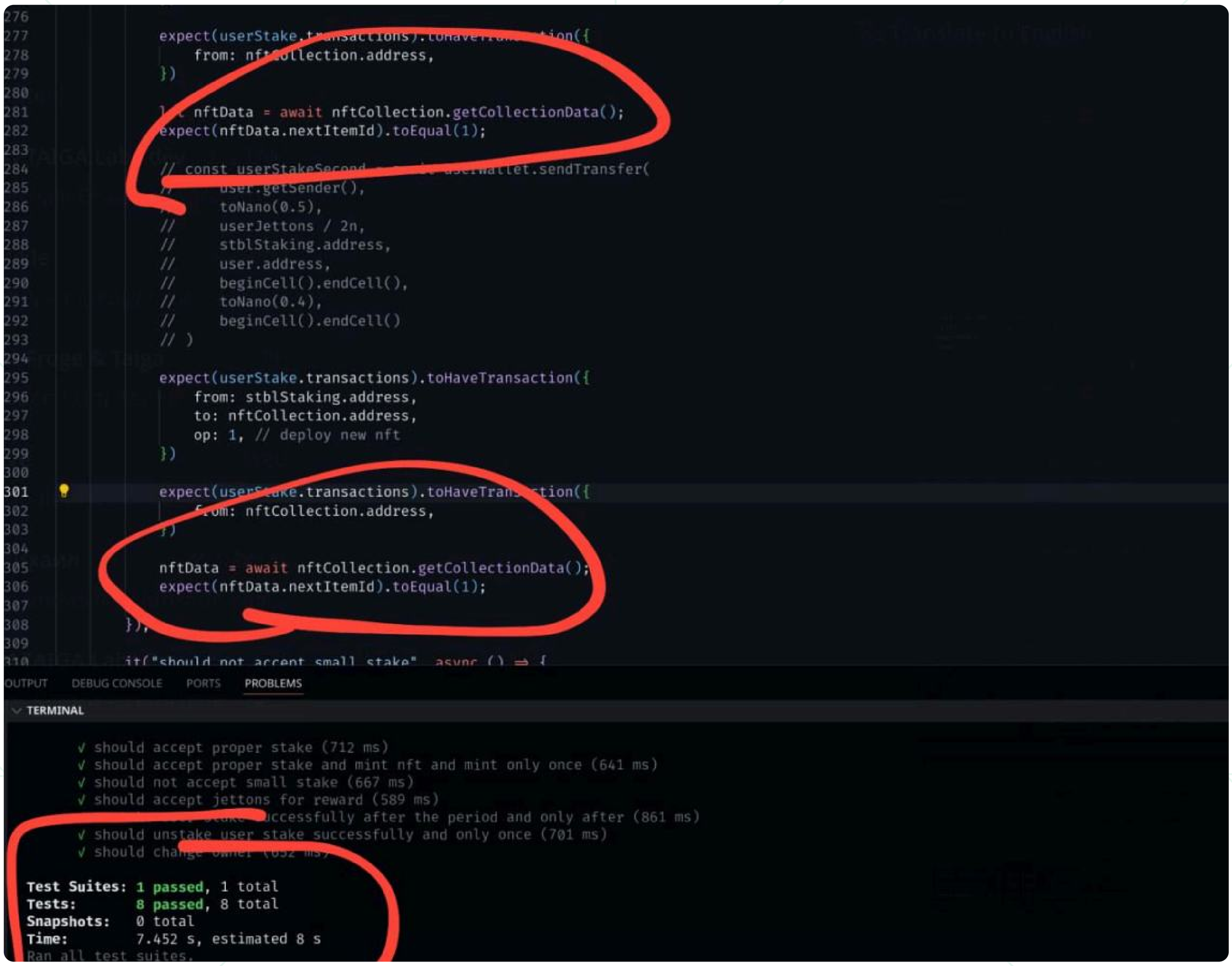
Поэтому айтем с помощью этой функции можно заминтить только один, под индексом 0, при попытке заминтить повторно - будет ошибка 402, ниже код из файла nft_collection.fc

```
cell pack_nft_mint_message(int query_id, int item_value, cell nft_message) inline {
    return begin_cell()
        .store_uint(op::nft::mint, 32)
        .store_uint(query_id, 64)
        .store_int(0, 64) ;; item_index
        .store_coins(item_value)  ;; ton to send to the item
        .store_ref(nft_message)
        .end_cell();
}
```

Поэтому нужно запоминать индекс последнего айтема, который мы заминтили и инкрементировать этот индекс для следующих минтов.



При втором минте - индекс токена должен быть 2, но, как мы видим, вылетает ошибка



Ошибок нет, если после второй транзакции ожидать значение 1, так как nextItemIndex не изменился, а следовательно, айтем не задеплоился

Предпочтительнее использовать константы вместо ассемблерных примитивов (Использование лишнего газа)

Пример:

```
int op::transfer_ownership() asm "0x2da38aaf PUSHINT";
const int op::transfer_ownership = 0x2da38aaf;
```

Стоит определять ошибки/оп-коды и прочие “константы” другим способом, не с помощью ассемблерных примитивов, а с помощью констант, так как второе помогает компилятору оптимизировать код самостоятельно, не стоит это делать за него лишний раз.

Компилятор FunC внутрь ассемблерной функции не смотрит. Например так, “256 PUSHINT” можно заменять на “8 PUSHPOW2”, компилятор может воспользоваться другим кодом, создать значение в другой момент (если так будет удобнее по стеку).

Более того, нинешний подход расходует больше газа (https://docs.ton.org/learn/tvm-instructions/instructions).

Или же вот:

```
cell    delib::stc(slice s)   asm "NEWC SWAP STSLICER ENDC";    ;; converts a slice into cell
```

```
begin_cell()
        .store_slice(s)
.end_cell()
```

вместо delib::stc(s) почти наверняка будет не хуже.

Особенно интересно, зачем SWAP STSLICER(26 ед.газа),
когда есть вполне нормальная операция STSLICE(18 ед. газа).

CE	STSLICE	s b - b'	Stores *Slice* s into *Builder* b.	18
CF16	STSLICER	b s - b'	Equivalent to SWAP STSLICE.	26

Это не все примеры, в коде полно функций, оформленных
в виде ассемблерных примитивов, которые бездумно расходуют газ.

inline/inline_ref
(Совет по оптимизации)

```
104  cell jetton_transfer(int amount, slice to_address, slice response_address,
105                       slice jetton_wall, int ton_amount) inline {
106
107      cell body = pack_simple_jetton_transfer(msg::query_id, amount, to_address,
108                                              response_address, CONF::JETTON_FWD_AMOUNT);
109
110      return delib::int_msg(jetton_wall, false, ton_amount, body, null());
111  }
112
113  () throw_jettons_128(slice to, slice jetton_wall, int amount) impure inline {
114      cell msg = jetton_transfer(amount, to, to, jetton_wall, 0);
115      send_raw_message(msg, MSG::ALL_NOT_RESERVED + MSG::IGNORE_ERRORS);
116
117      commit();
118      throw(ERR::NOT_ENOUGH);
119  }
120
121  (slice, cell) calc_escrow_contract(slice staker_address) inline {
122      cell contract_data = pack_escrow_data(my_address(), staker_address);
123      cell contract_init = delib::basic_state_init(self::escrow_contract_code, contract_data);
124      return (delib::addrstd_by_state(CONF::WC, contract_init), contract_init);
125  }
126
127  () auth_by_escrow_contract(slice staker_address) impure inline {
128      (slice escrow_address, _) = calc_escrow_contract(staker_address);
129      throw_unless(ERR::ACCESS, equal_slice_bits(msg::sender, escrow_address));
130  }
131
132  ;; contract constructor (can be called only once)
133  () constructor(slice msg_body) impure inline_ref {
```

Inline_ref specifier

The code of a function with the inline_ref specifier is put into a separate cell, and every time when the function is called, a CALLREF command is executed by TVM.

So it's similar to inline, but because a cell can be reused in several places without duplicating it, it is almost always more efficient in terms of code size to use inline_ref specifier instead of inline unless the function is called exactly once.

Recursive calls of inline_ref'ed functions are still impossible because there are no cyclic references in the TVM cells.

Источник: (https://docs.ton.org/develop/func/functions)

Как сказано в документации: “с точки зрения размера кода почти всегда более эффективно использовать inline_ref спецификатор вместо inline, если функция не вызывается ровно один раз.”, поэтому непонятно почему именно тут спецификатор inline, хоть функции вызываются много раз (>> 1).

Источник: (https://docs.ton.org/develop/func/functions)

Как сказано в документации: “с точки зрения размера кода почти всегда более эффективно использовать inline_ref спецификатор вместо inline, если функция не вызывается ровно один раз.”, поэтому непонятно почему именно тут спецификатор inline, хоть функции вызываются много раз (>> 1).