Опашки

Пламен Танов
Ненко Табаков
Технологично училище „Електронни системи“
Технически университет – София

версия 0.1

Въведение

- Опашките са контейнери за данни
- Служат за съхранението на множество от данни
- В зависимост от това къде се вмъкват новите данни и от къде се изтриват различаваме различни видове контейнери: стекове, FIFO опашки, случайни опашки, приоритетни опашки и т.н.

FIFO опашки

- FIFO опашката е абстрактна структура от данни, която поддържа изтриване и добавяне на обекти
- First In, First Out (FIFO)
- Пръв влязъл, пръв излязъл (обратно на стек!)
- Вместо да премахваме последно вмъкнатия елемент (като в стек), премахваме елемента, който е бил в опашката най-дълго време (първи е влязъл там!)
- Обикновено така се подреждат задачите, които трябва да свършим – правим това, което ни кажат първо. Останалите задачи, които са ни казали след това, ги помним и изпълняваме в реда, в който са ни ги възложили...

Дефиниция

Опашка FIFO е АТД, който съдържа две основни операции: вмъкване (put) на нов елемент и изваждане (get) на елемента, който е бил вмъкнат най-рано

Интерфейс за работа с FIFO опашка

```
//инициализация на опашката с sizeOfQueue елемента
void QUEUEinit(int sizeOfQueue);
//проверка дали опашката е празна
int QUEUEempty();
//добавяне в края на опашката
void QUEUEput(Item itemToPut);
//взимане на първия елемент от опашката
Item QUEUEget();
//освобождава паметта, заделена при инициализацията и др.
void QUEUEfree();
```

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.)

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
```

Опашка след операцията: 1

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
                    Опашка след операцията: 1
put(2);
                    Опашка след операцията: 1, 2
```

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
                    Опашка след операцията: 1
put(2);
                    Опашка след операцията: 1, 2
put(3);
                    Опашка след операцията: 1, 2, 3
```

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
                    Опашка след операцията: 1
put(2);
                    Опашка след операцията: 1, 2
put(3);
                    Опашка след операцията: 1, 2, 3
put(4);
                    Опашка след операцията: 1, 2, 3, 4
```

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
                    Опашка след операцията: 1
put(2);
                    Опашка след операцията: 1, 2
put(3);
                    Опашка след операцията: 1, 2, 3
put(4);
                    Опашка след операцията: 1, 2, 3, 4
get();//връща 1
                    Опашка след операцията: 2, 3, 4
```

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
                    Опашка след операцията: 1
put(2);
                    Опашка след операцията: 1, 2
put(3);
                    Опашка след операцията: 1, 2, 3
put(4);
                    Опашка след операцията: 1, 2, 3, 4
get();//връща 1
                    Опашка след операцията: 2, 3, 4
get();//връща 2
                    Опашка след операцията: 3, 4
```

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
                    Опашка след операцията: 1
put(2);
                    Опашка след операцията: 1, 2
put(3);
                    Опашка след операцията: 1, 2, 3
put(4);
                    Опашка след операцията: 1, 2, 3, 4
get();//връща 1
                    Опашка след операцията: 2, 3, 4
get();//връща 2
                    Опашка след операцията: 3, 4
put(5);
                    Опашка след операцията: 3, 4, 5
```

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
                    Опашка след операцията: 1
put(2);
                    Опашка след операцията: 1, 2
put(3);
                    Опашка след операцията: 1, 2, 3
put(4);
                    Опашка след операцията: 1, 2, 3, 4
get();//връща 1
                    Опашка след операцията: 2, 3, 4
get();//връща 2
                    Опашка след операцията: 3, 4
put(5);
                    Опашка след операцията: 3, 4, 5
get();//връща 3
                    Опашка след операцията: 4, 5
```

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
                    Опашка след операцията: 1
put(2);
                    Опашка след операцията: 1, 2
put(3);
                    Опашка след операцията: 1, 2, 3
put(4);
                    Опашка след операцията: 1, 2, 3, 4
get();//връща 1
                    Опашка след операцията: 2, 3, 4
get();//връща 2
                    Опашка след операцията: 3, 4
put(5);
                    Опашка след операцията: 3, 4, 5
get();//връща 3
                    Опашка след операцията: 4, 5
get();//връща 4
                    Опашка след операцията: 5
```

Пример

Вмъкваме последователно числата от 1 до 5 и ги изваждаме в същия ред (първо сме вмъкнали 1 и първо изваждаме 1, след това 2, след него 3 и т.н.):

```
put(1);
                    Опашка след операцията: 1
put(2);
                    Опашка след операцията: 1, 2
put(3);
                    Опашка след операцията: 1, 2, 3
put(4);
                    Опашка след операцията: 1, 2, 3, 4
get();//връща 1
                    Опашка след операцията: 2, 3, 4
get();//връща 2
                    Опашка след операцията: 3, 4
put(5);
                    Опашка след операцията: 3, 4, 5
get();//връща 3
                    Опашка след операцията: 4, 5
get();//връща 4
                    Опашка след операцията: 5
```

Винаги взимаме **първия** елемент, а слагаме на **края**!

Реализация на FIFO опашка$_1$

Дефинирайки интерфейс за работа с FIFO опашка ние вече имаме начина за работа с него. Поради това можем да пишем програми без да се интересуваме от точната реализация на опашката и също така да правим различни реализации (на опашка) без да е необходимо да променяме кода на нашите програми

Реализация на FIFO опашка$_2$

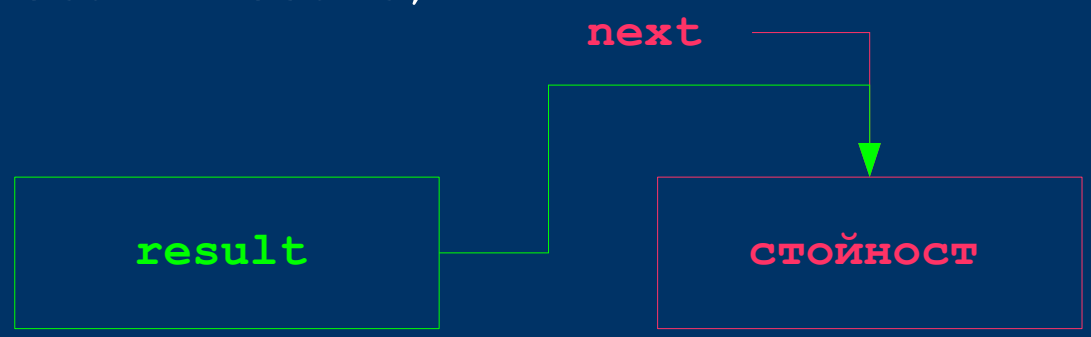
- Възможни са различни реализации на опашки
- Свободни сме да организираме съхранението на елементите по какъвто начин искаме, доколкото поддържаме илюзията, че елементите са организирани по подходящия начин (отговарят на разгледания интерфейс)
- Най-често използваните са следните две:
  - Със свързани списъци
  - С масив

Реализация със списъци

- Съхраняваме елементите в списъка в ред от най-рано вмъкнат към най-късно вмъкнат (този ред е обратен на този при стек)
- За всяко **`put()`**, **`get()`** извикване се заделя/освобождава памет – бавни операции, но ползваме памет точно толкова колкото ни е нужно
- Поддържаме два указателя към списъка:
  - Към началото (**`head`**) - за да можем да изваждаме първия елемент
  - Към края (**`tail`**) - за да можем да вмъкваме нов елемент в края
- Вмъкваме в края (**`tail`**), а четем елементите от началото (**`head`**)

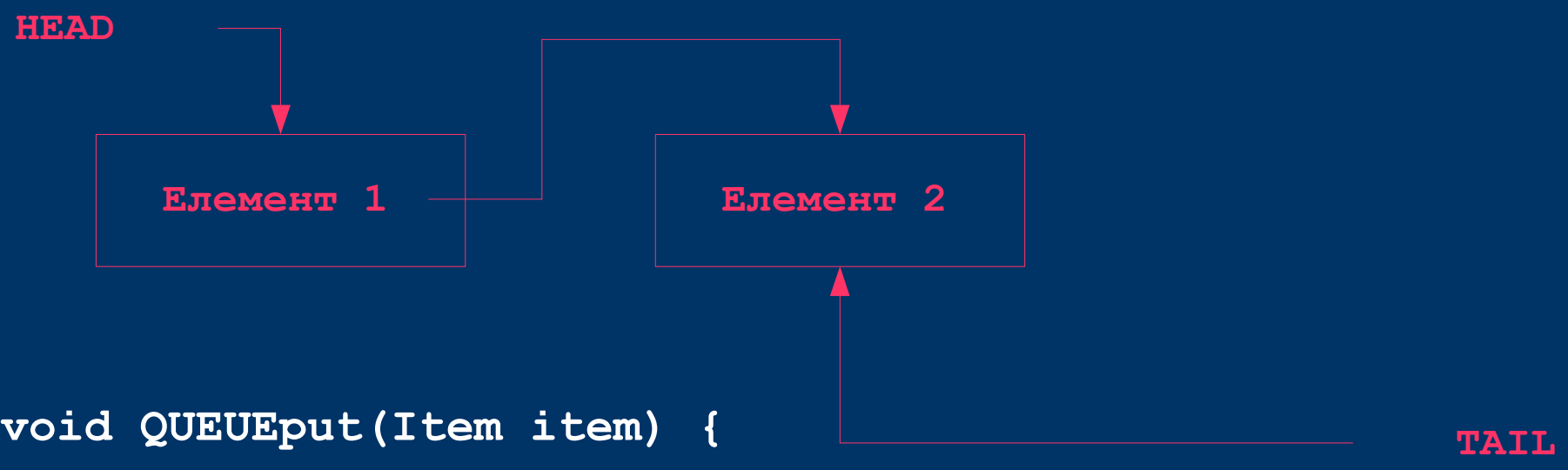
Заделяне на памет за възел

```
//заделя памет за нов възел, чиято стойност става newItem
//и започва да сочи към next
Link NEW(Item newItem, Link next) {

   //заделяме памет за новия възел:
   Link result = malloc(sizeof (*result));

   //Копираме стойността на newItem в/у стойността на result
   result->item = newItem;

   result->next = next; //насочваме го към следващия елемент

   return result;
}
```



Вмъкване на елемент$_1$

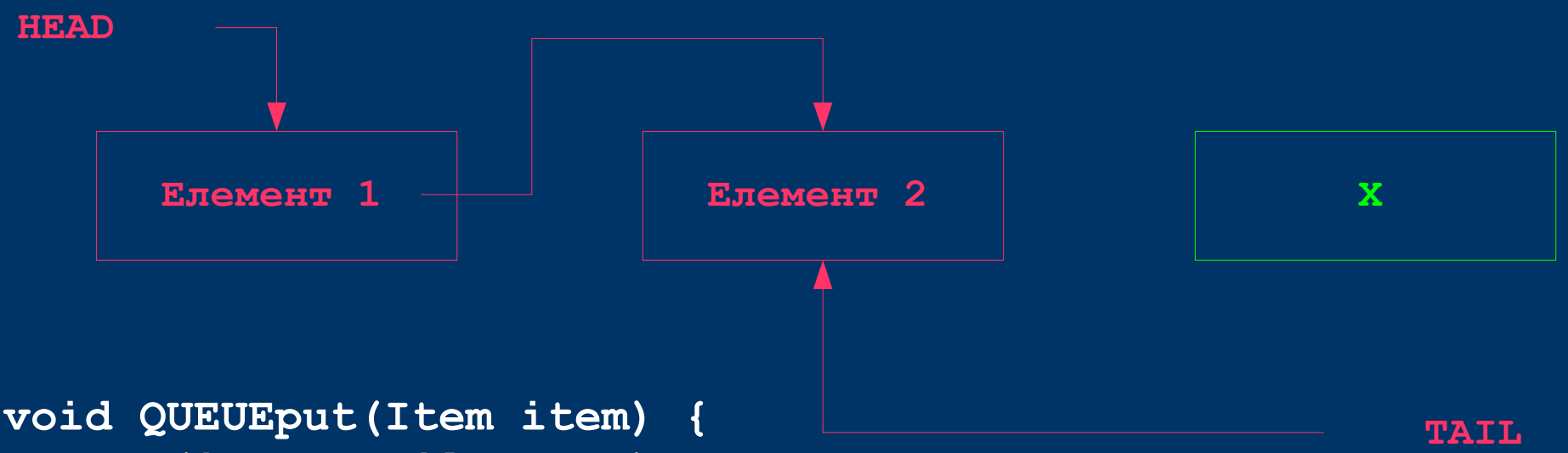
- За да вмъкнем елемент към опашка трябва да го добавим в края й, т.е. след елемента, към който сочи **`tail`**




```
void QUEUEput(Item item) {

}
```

Вмъкване на елемент$_2$

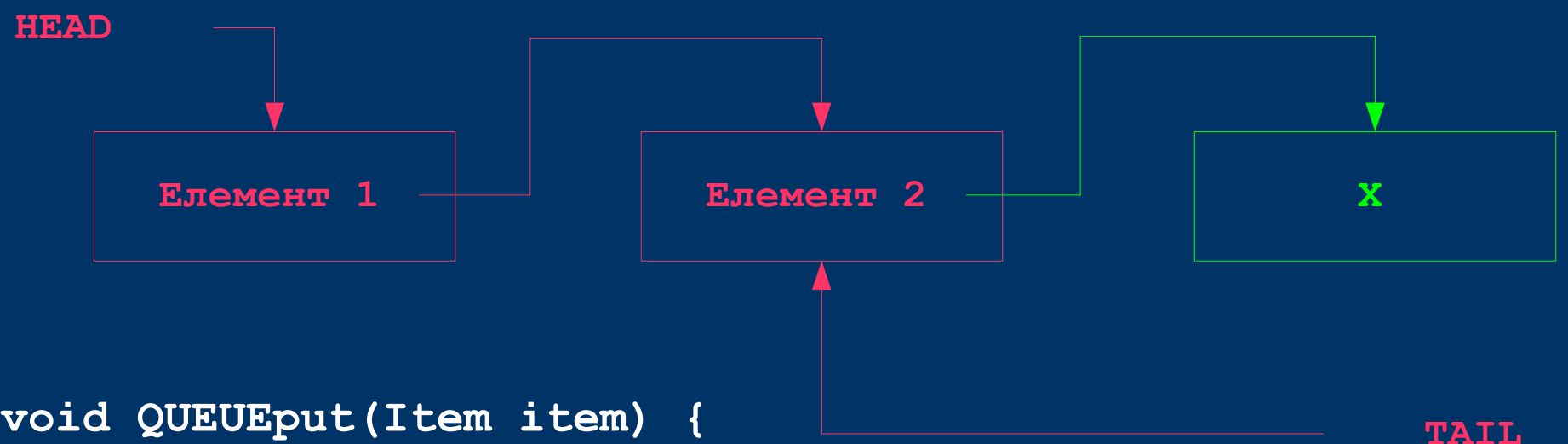
- Заделяме памет за новия елемент
- Инициализираме стойността му




```
void QUEUEput(Item item) {
   NEW(item, tail->next);

}
```

Вмъкване на елемент$_3$

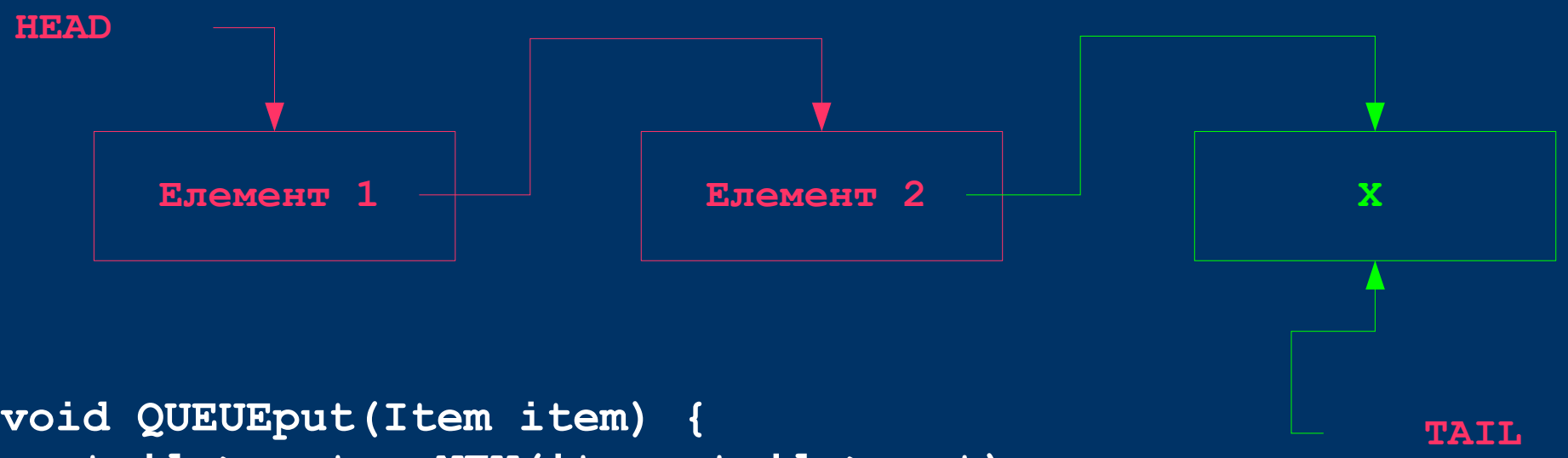
- Насочваме **`tail->next`** към новият елемент




```
void QUEUEput(Item item) {
   tail->next = NEW(item, tail->next);

}
```

Вмъкване на елемент$_3$

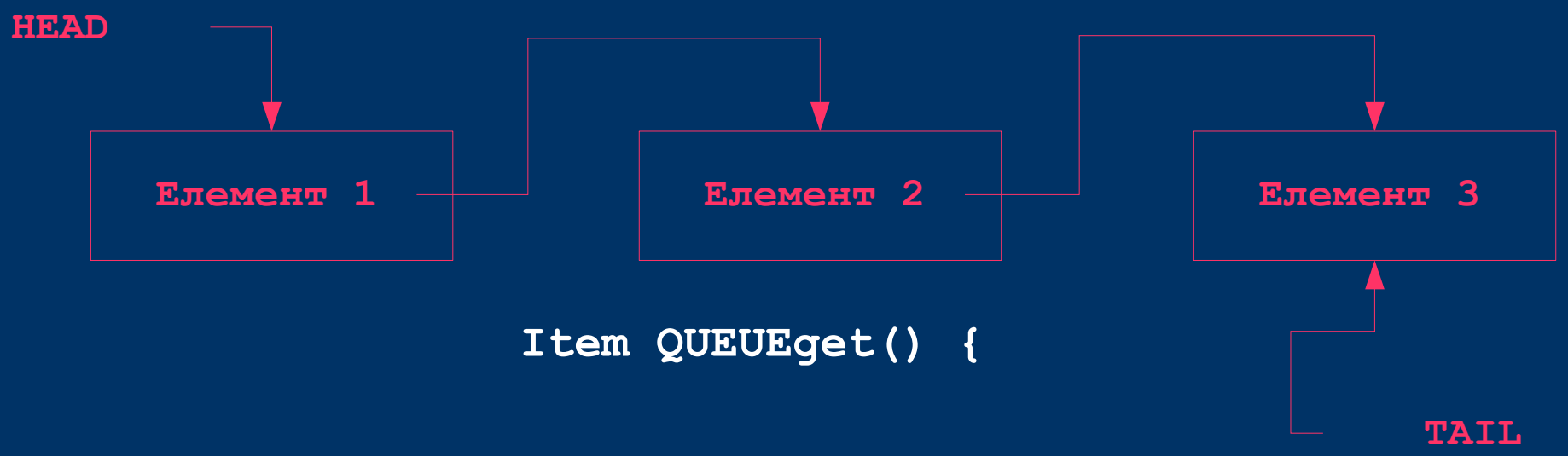
- Насочваме **tail** към **tail->next**




```
void QUEUEput(Item item) {
   tail->next = NEW(item, tail->next);
   tail = tail->next;
}
```

Четене на елемент$_1$

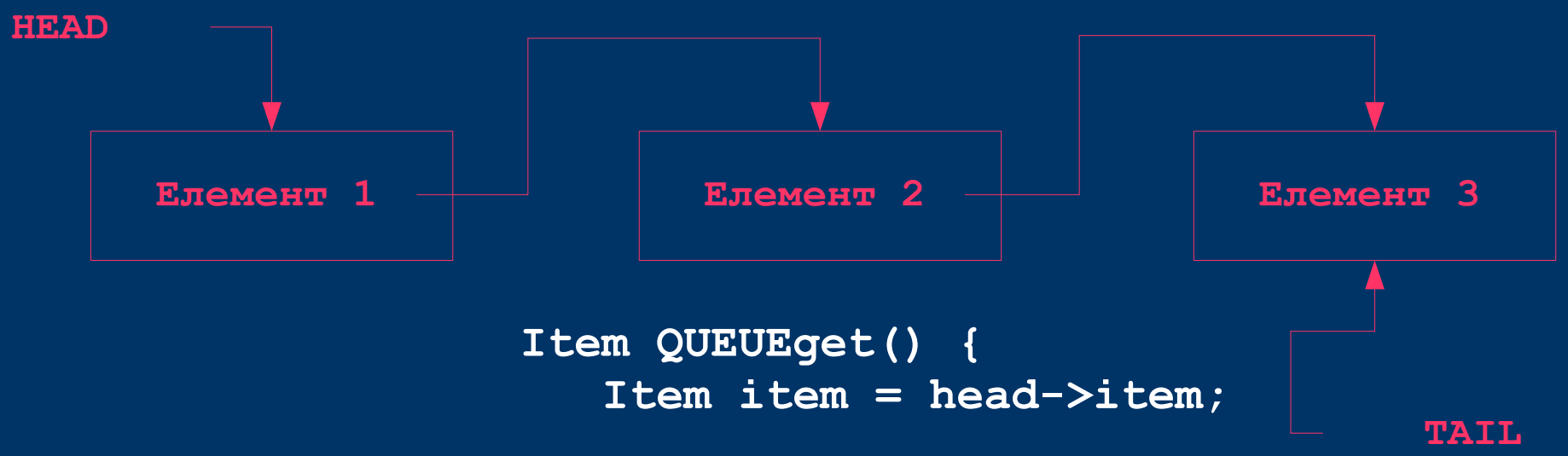
- Взимаме елемент от началото на опашката (**head**)




```
Item QUEUEget() {

}
```

Четене на елемент$_2$

- Взимаме стойността на елемента, към който сочи **head** и я записваме в дадена променлива



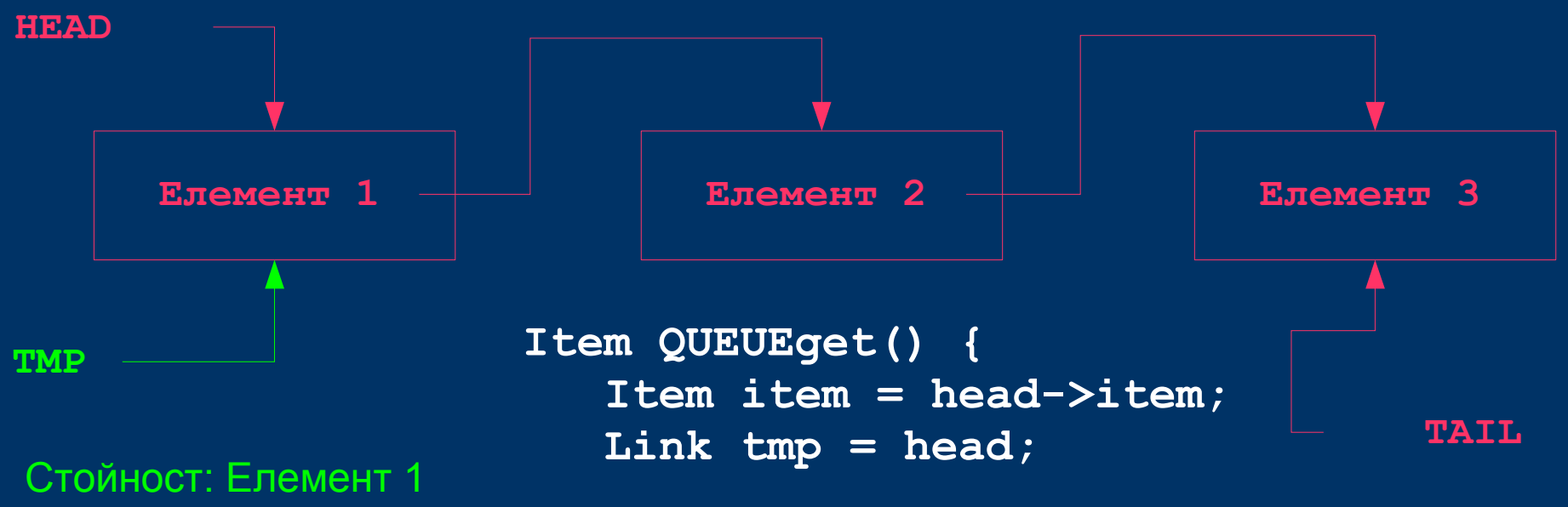

```
Item QUEUEget() {
    Item item = head->item;
```

Стойност: Елемент 1

```
}
```

Четене на елемент$_3$

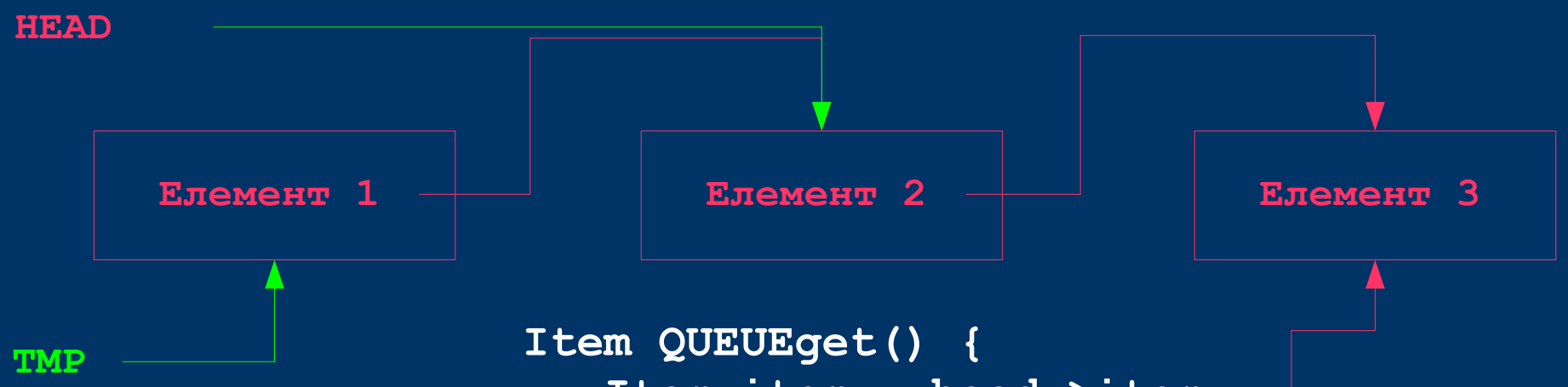
- Запазваме указател към елемента, към който сочи **`head`** (за да можем по-късно да освободим заетата памет от него)




```
Item QUEUEget() {
    Item item = head->item;
    Link tmp = head;

}
```

Четене на елемент$_4$

- Насочваме **head** към **head->next**

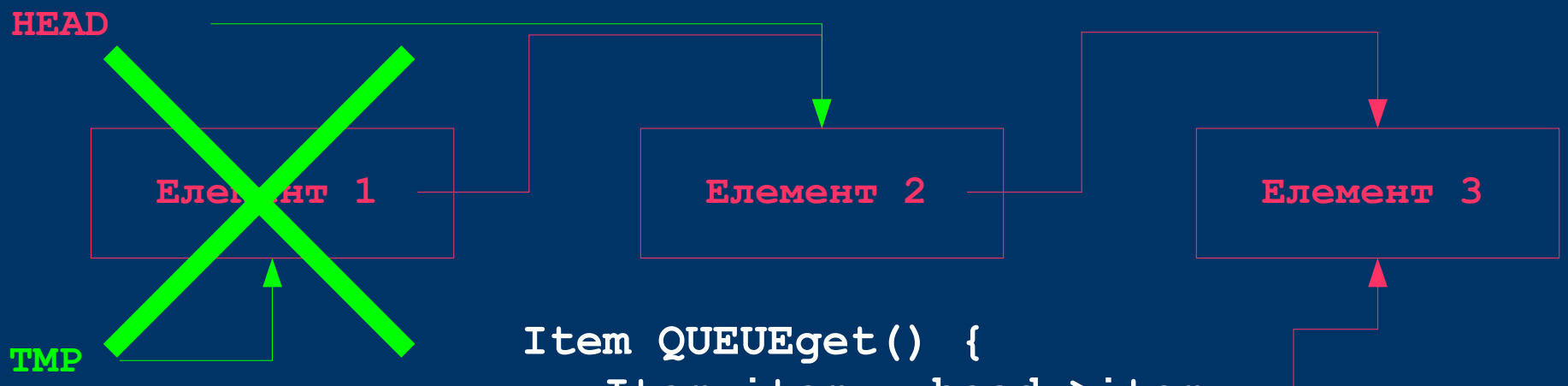


Стойност: Елемент 1

```
Item QUEUEget() {
    Item item = head->item;
    Link tmp = head;
    head = head->next;

}
```

Четене на елемент$_5$

- Освобождаваме мястото, заделено от променливата, към която сочи **TMP**. Това е елементът, който искаме да прочетем и чиято стойност се пази в „Стойност“

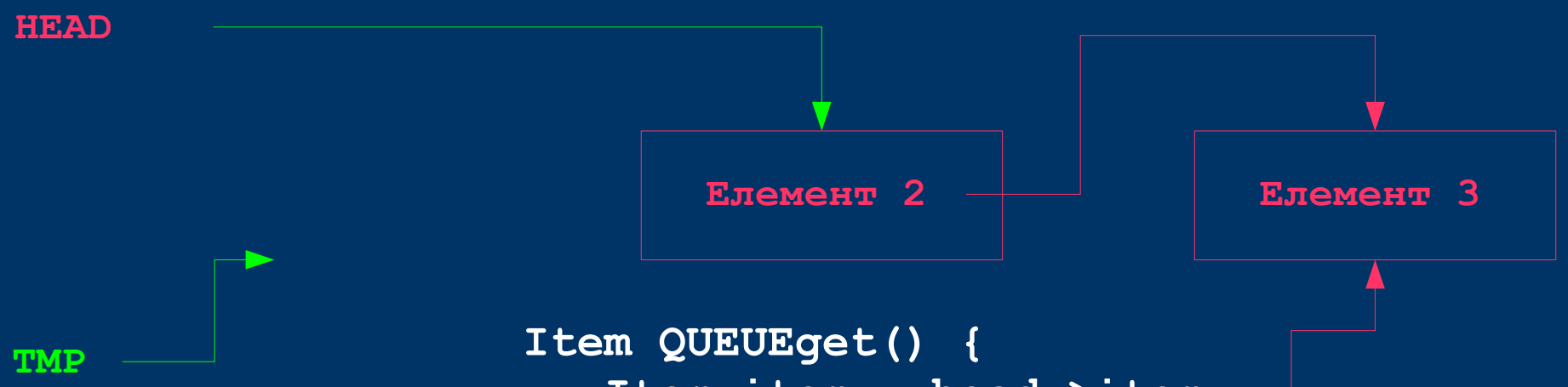


Стойност: Елемент 1

```
Item QUEUEget() {
    Item item = head->item;
    Link tmp = head;
    head = head->next;
    free(tmp);

}
```

Четене на елемент$_6$

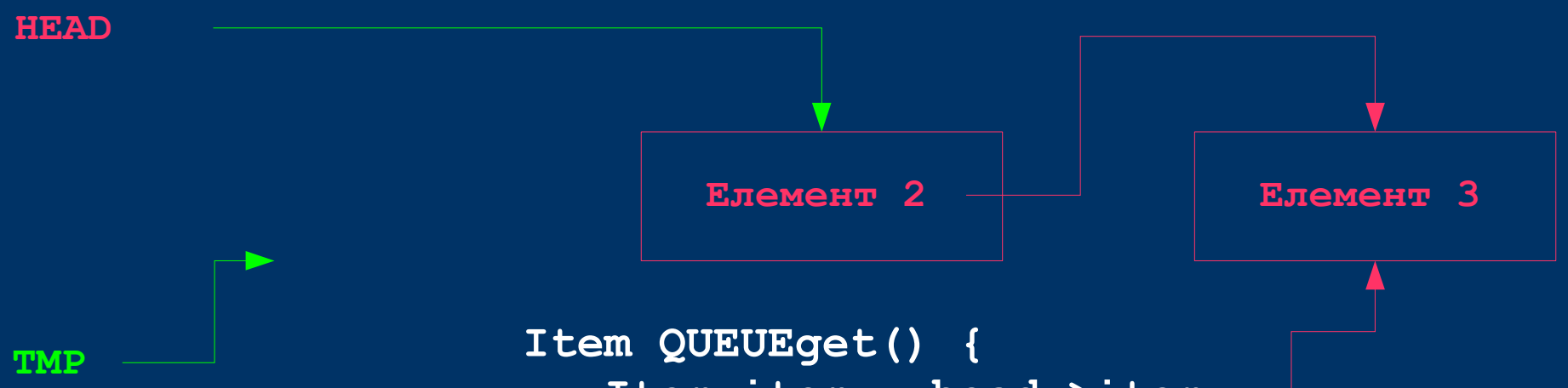
- Освобождаваме мястото, заделено от променливата, към която сочи **TMP**. Това е елементът, който искаме да прочетем и чиято стойност се пази в „Стойност“




```
Item QUEUEget() {
    Item item = head->item;
    Link tmp = head;
    head = head->next;
    free(tmp);

}
```

Четене на елемент$_7$

- Ползваме променливата, записана в „Стойност“



Стойност: Елемент 1

```
Item QUEUEget() {
    Item item = head->item;
    Link tmp = head;
    head = head->next;
    free(tmp);
    return item;
}
```

Пример$_1$

Различието между FIFO опашка и стек е, че новите елементи се вмъкват в края, вместо в началото.

Съответно тази програма пази указател **`tail`** към последния елемент на списъка, така че функцията **`QUEUEput()`** може да добави нов възел чрез свързването му към възела, сочен от **`tail`**, след което обновява **`tail`** да сочи към новия възел.

Функциите **`QUEUEget()`**, **`QUEUEinit()`** и **`QUEUEempty()`** са еднакви със техните съответствия в стековата реализация със свързан списък

Пример$_2$

```
#include <stdlib.h>
#include "Item.h" //тук е информацията за типа на елементите
#include "QUEUE.h"//прототипите на функциите по-долу
typedef struct QUEUEnode * Link;
struct QUEUEnode {Item item; Link next;};

static Link head, tail;

//заделя памет за нов възел
Link NEW(Item newItem, Link next) {
   Link result = malloc(sizeof (*result));
   result->item = newItem;//инициализира елемента
   result->next = next; //насочваме го към следващия елемент
   return result;
}
void QUEUEinit(int sizeOfQueue) {
   head = NULL;
}
int QUEUEempty(void) {
   return head == NULL;
}
```

Пример$_3$

```
void QUEUEput(Item item) {
   if(head == NULL){//ако до сега няма елементи и вмъкваме 1вия
                    //инициализираме head и tail:
      head = (tail = (Link) NEW(item, head));
      return;       //излизаме от функцията
   }
   tail->next = NEW(item, tail->next);
   tail = tail->next;
}
Item QUEUEget() {
   Item item = head->item;
   Link tmp = head;
   head = head->next;
   free(tmp);
   return item;
   //не проверяваме дали изобщо има елементи!
}
```

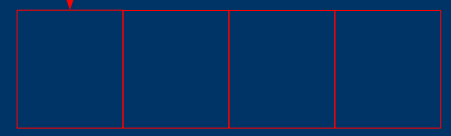
Реализация с масив

- Трябва предварително да знаем максималния брой на елементите!!! (това е голям недостатък – повече памет)
- По-бърз е, защото не заделяме/трием памет при всяко **`put()`**, **`get()`** извикване
- Когато четем от началото (т.е. изпълняваме функцията **`QUEUEget()`**) не местим всички елементи с една позиция на ляво, а пазим индекса на първия (**`head`**) и последния елемент (**`tail`**) на опашката. Елементите на опашката са елементите между тези два индекса
- Когато вмъкваме елемент – увеличаваме **`tail`**
- Когато четем (трием) елемент – увеличаваме **`head`**

Оператор %

```
0 % 4 = 0
1 % 4 = 1
2 % 4 = 2
3 % 4 = 3
4 % 4 = 0
5 % 4 = 1
6 % 4 = 2
....
```

```
i = 0
i % 4 = 0
```



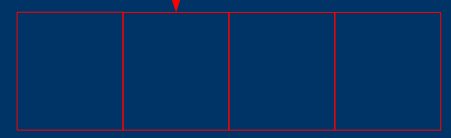
Ако броят на елементите в масив е 4, а **`i=0,1,2,...`**, то **`i % 4`** винаги ще дава поредния елемент от масива.

Когато **`i`** надвиши броя на елементите, то **`i % 4`** ще започне от началото!

Оператор %

```
0 % 4 = 0
1 % 4 = 1
2 % 4 = 2
3 % 4 = 3
4 % 4 = 0
5 % 4 = 1
6 % 4 = 2
....
```

```
i = 1
i % 4 = 1
```



Ако броят на елементите в масив е 4, а **`i=0,1,2,...`**, то **`i % 4`** винаги ще дава поредния елемент от масива.

Когато **`i`** надвиши броя на елементите, то **`i % 4`** ще започне от началото!

Оператор %

```
0 % 4 = 0
1 % 4 = 1
2 % 4 = 2
3 % 4 = 3
4 % 4 = 0
5 % 4 = 1
6 % 4 = 2
....
```

```
i = 2
i % 4 = 2
```

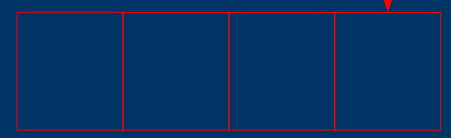
Ако броят на елементите в масив е 4, а **`i=0,1,2,...`**, то **`i % 4`** винаги ще дава поредния елемент от масива.

Когато **`i`** надвиши броя на елементите, то **`i % 4`** ще започне от началото!

Оператор %

```
0 % 4 = 0
1 % 4 = 1
2 % 4 = 2
3 % 4 = 3
4 % 4 = 0
5 % 4 = 1
6 % 4 = 2
....
```

```
i = 3
i % 4 = 3
```



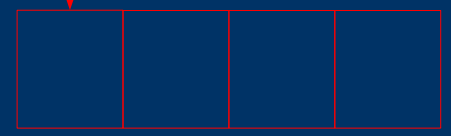
Ако броят на елементите в масив е 4, а **`i=0,1,2,...`**, то **`i % 4`** винаги ще дава поредния елемент от масива.

Когато **`i`** надвиши броя на елементите, то **`i % 4`** ще започне от началото!

Оператор %

```
0 % 4 = 0
1 % 4 = 1
2 % 4 = 2
3 % 4 = 3
4 % 4 = 0
5 % 4 = 1
6 % 4 = 2
....
```

```
i = 4
i % 4 = 0
```



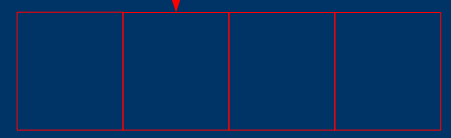
Ако броят на елементите в масив е 4, а **`i=0,1,2,...`**, то **`i % 4`** винаги ще дава поредния елемент от масива.

Когато **`i`** надвиши броя на елементите, то **`i % 4`** ще започне от началото!

Оператор %

```
0 % 4 = 0
1 % 4 = 1
2 % 4 = 2
3 % 4 = 3
4 % 4 = 0
5 % 4 = 1
6 % 4 = 2
....
```

```
i = 5
i % 4 = 1
```



Ако броят на елементите в масив е 4, а **`i=0,1,2,...`**, то **`i % 4`** винаги ще дава поредния елемент от масива.

Когато **`i`** надвиши броя на елементите, то **`i % 4`** ще започне от началото!

Оператор %

```
0 % 4 = 0
1 % 4 = 1
2 % 4 = 2
3 % 4 = 3
4 % 4 = 0
5 % 4 = 1
6 % 4 = 2
....
```

```
i = 6
i % 4 = 2
```

Ако броят на елементите в масив е 4, а **`i=0,1,2,...`**, то **`i % 4`** винаги ще дава поредния елемент от масива.

Когато **`i`** надвиши броя на елементите, то **`i % 4`** ще започне от началото!

Пример$_1$

Съдържанието на опашката представлява елементите от масива между **`head`** и **`tail`**, вземайки под внимание увиването обратно към 0, когато се стигне до края на масива. Ако **`head`** и **`tail`** са равни, тогава приемаме, че опашката е празна. Ако след **`put()`** те са станали равни – тогава тя е пълна! Както обикновено, не разглеждаме някои условия за грешки, но правим размера на масива с 1 по-голям от максималния брой елементи, които клиентът очаква да види в опашката. Бихме могли обаче да усложним програмата, така че да прави такива проверки

Пример 2

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно.

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно. В началото: - празна!

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно. В началото: - празна!

put(1);

Опашка след операцията: 1

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно.

В началото: - празна!

put(1);

Опашка след операцията: 1

put(2);

Опашка след операцията: 1 2

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно.

В началото: - празна!

put(1);

Опашка след операцията: 1

put(2);

Опашка след операцията: 1 2

put(3);

Опашка след операцията: 1 2 3

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно.

В началото: | | | | - празна!

put(1);

Опашка след операцията: | 1 | | | |

put(2);

Опашка след операцията: | 1 | 2 | | |

put(3);

Опашка след операцията: | 1 | 2 | 3 | |

put(4);

Опашка след операцията: | 1 | 2 | 3 | 4 | - пълна!

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно.

В началото: | | | | - празна!

put(1);

Опашка след операцията: | 1 | | | |

put(2);

Опашка след операцията: | 1 | 2 | | |

put(3);

Опашка след операцията: | 1 | 2 | 3 | |

put(4);

Опашка след операцията: | 1 | 2 | 3 | 4 | - пълна!

get();//връща 1

Опашка след операцията: | | 2 | 3 | 4 |

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно.

В началото: | | | | - празна!

put(1);

Опашка след операцията: | 1 | | | |

put(2);

Опашка след операцията: | 1 | 2 | | |

put(3);

Опашка след операцията: | 1 | 2 | 3 | |

put(4);

Опашка след операцията: | 1 | 2 | 3 | 4 | - пълна!

get();//връща 1

Опашка след операцията: | | 2 | 3 | 4 |

get();//връща 2

Опашка след операцията: | | | 3 | 4 |

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно.

В началото: | | | | - празна!

put(1);

Опашка след операцията: | 1 | | | |

put(2);

Опашка след операцията: | 1 | 2 | | |

put(3);

Опашка след операцията: | 1 | 2 | 3 | |

put(4);

Опашка след операцията: | 1 | 2 | 3 | 4 | - пълна!

get();//връща 1

Опашка след операцията: | | 2 | 3 | 4 |

get();//връща 2

Опашка след операцията: | | | 3 | 4 |

put(5);

Опашка след операцията: | 5 | | 3 | 4 |

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно.

В началото: | | | | - празна!

```
put(1);
```

Опашка след операцията: | 1 | | | |

```
put(2);
```

Опашка след операцията: | 1 | 2 | | |

```
put(3);
```

Опашка след операцията: | 1 | 2 | 3 | |

```
put(4);
```

Опашка след операцията: | 1 | 2 | 3 | 4 | - пълна!

```
get();//връща 1
```

Опашка след операцията: | | 2 | 3 | 4 |

```
get();//връща 2
```

Опашка след операцията: | | | 3 | 4 |

```
put(5);
```

Опашка след операцията: | 5 | | 3 | 4 |

```
get();//връща 3
```

Опашка след операцията: | 5 | | | 4 |

Пример$_2$

head е заграден с , tail - с

При put() стойността се слага в , а то се премества с една позиция в дясно. При get() стойността се взима от и пак се мести в дясно.

В началото: | | | | - празна!

```
put(1);
```

Опашка след операцията: | 1 | | | |

```
put(2);
```

Опашка след операцията: | 1 | 2 | | |

```
put(3);
```

Опашка след операцията: | 1 | 2 | 3 | |

```
put(4);
```

Опашка след операцията: | 1 | 2 | 3 | 4 | - пълна!

```
get();//връща 1
```

Опашка след операцията: | | 2 | 3 | 4 |

```
get();//връща 2
```

Опашка след операцията: | | | 3 | 4 |

```
put(5);
```

Опашка след операцията: | 5 | | 3 | 4 |

```
get();//връща 3
```

Опашка след операцията: | 5 | | | 4 |

```
get();//връща 4
```

Опашка след операцията: | 5 | | | |

Пример$_3$

```
#include <stdlib.h>
#include "Item.h"
static Item * items;        //указател към динамично заделен масив
static int N,               //максимален брой елементи
           head, tail;      //началото и краят на опашката
//инициализира опашката, заделя място и променя head и tail
void QUEUEinit(int sizeOfQueue) {
    N = sizeOfQueue + 1;  //ще заделим място за един елемент повече
    items = malloc(N * sizeof(Item));//трябва да се освободи някога!
    head = tail = 0;
}
int QUEUEempty() {                    //дали опашката е празна
    return head % N == tail;
}
void QUEUEput(Item newItem) {         //добавя елемент в опашката
//не проверяваме дали не добавяме повече от възможния брой елементи!
    items[tail++] = newItem;
    tail = tail % N;
}
Item QUEUEget() {                     //взима елемент от опашката
    head = head % N;
    return items[head++]; //не проверяваме дали изобщо има елементи!
}
```

Други видове опашки. Обобщени опашки

Освен стек и FIFO

Различават се по това кой елемент четем (трием):

- Случайна опашка
  - Премахва се случаен елемент
- Дек
  - Вмъкване и изтриване и в двата края (от който поискаме)
- Опашки с приоритети
  - Елементите имат ключове и правилото за изтриване е „премахни елемента с най-малък ключ“
- Символни таблици
  - Елементите имат ключове и правилото за изтриване е „премахни елемента, чийто ключ е равен на даден ключ“
- Други, в зависимост какво точно ни трябва...