Estrutura de Dados I

Capítulo 10: Estruturas Lineares

Prof. Abrantes Araújo Silva Filho

Introdução

- O TAD Pilha que estudamos é um exemplo que uma classe geral de tipos abstratos chamada de **TADs lineares**, na qual os elementos formam uma "**linha reta conceitual**".

- A implementação interna de TADs lineares pode ser feita, por exemplo, com:
  - Arrays
  - Listas encadeadas
    - Simples e/ou duplas
    - Circulares e/ou não circulares

- Nosso objetivo agora é aprender outro TAD linear, o **TAD Fila (queue)**.

- A fila também é da categoria dos containers.

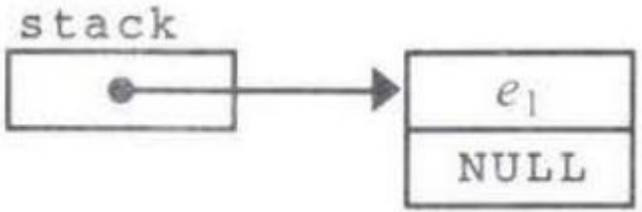
10.1 Mais sobre Pilhas

- A implementação interna mais comum para uma pilha é através do uso de arrays, mas também é possível utilizar **listas encadeadas** para criar um TAD pilha.

- Ao usar uma lista encadeada para implementar uma pilha, a pilha vazia é:

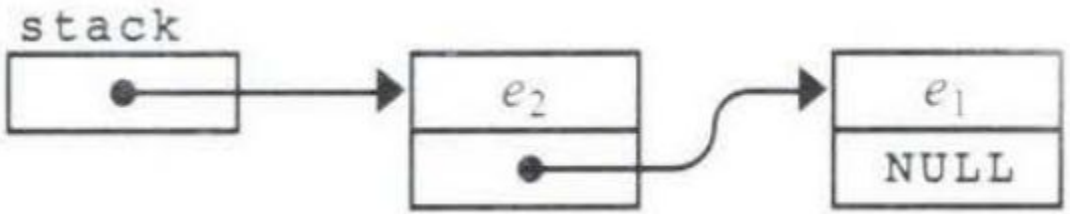
10.1 Mais sobre Pilhas

- Ao fazer o push de um elemento, ele é adicionado no início da lista (o início da lista passa a ser o topo da fila):



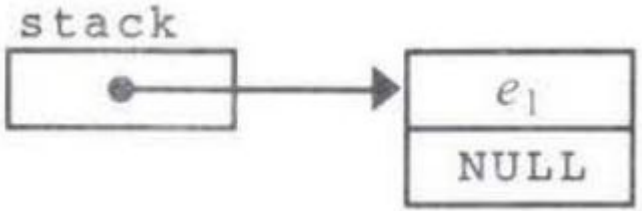
10.1 Mais sobre Pilhas

- Ao fazer o push de mais um elemento, ele também é adicionado no início da lista (o início da lista é o topo da fila):



10.1 Mais sobre Pilhas

- Ao fazer o pop de um elemento, ele é removido do início da lista (o início da lista é o topo da fila):



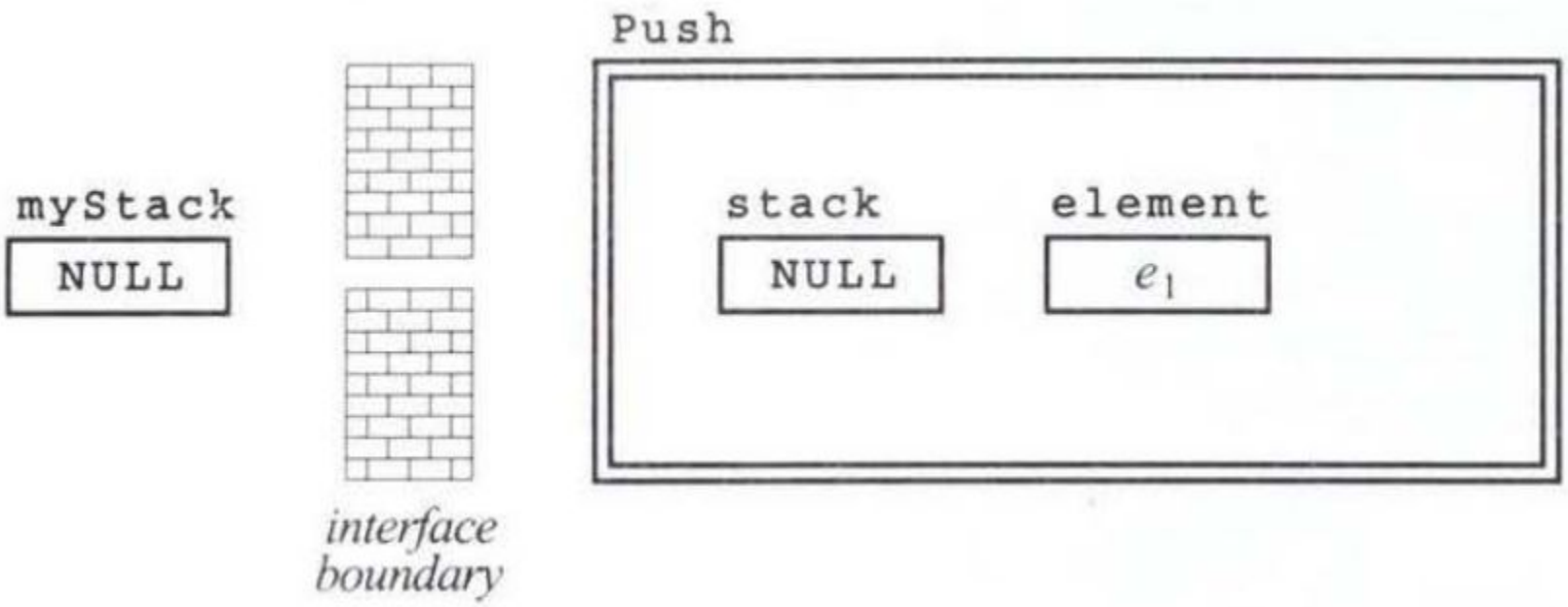
10.1 Mais sobre Pilhas

- Como implementar um TAD Pilha usando uma lista encadeada?

- Um **erro comum** é fazer algo parecido com isso:

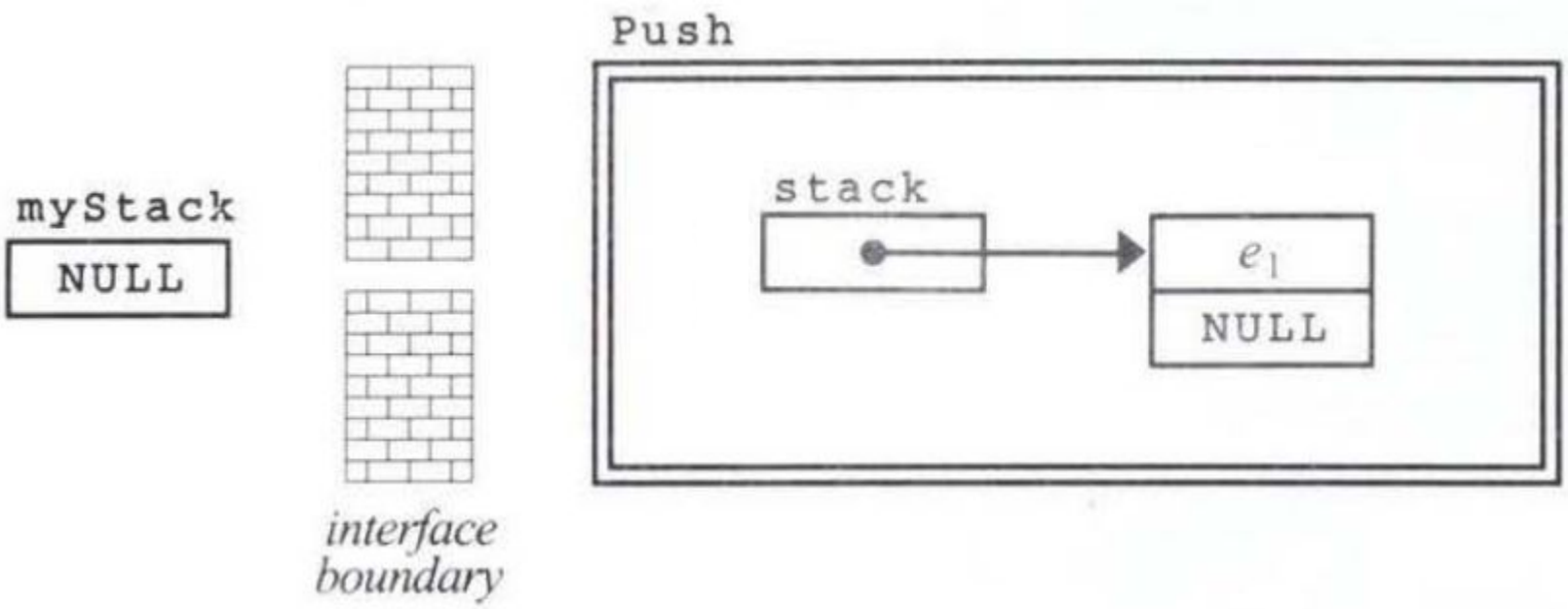
```
typedef struct cellT *stackADT;

stackADT NewStack(void)
{
    return (NULL);
}

myStack = NewStack();
```

10.1 Mais sobre Pilhas



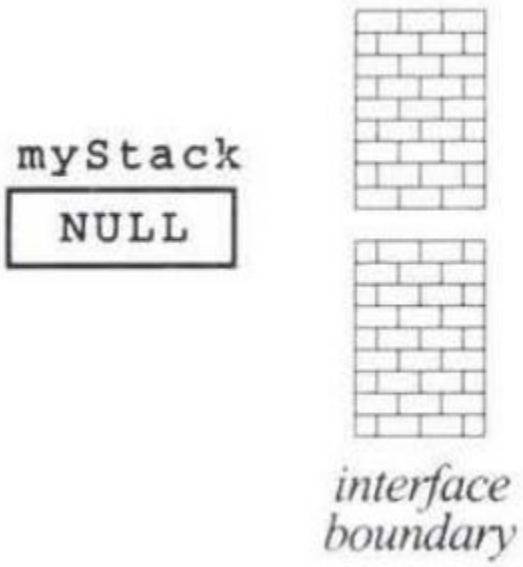
O valor do stack dentro da função push é um **CÓPIA** do valor do myStack na aplicação cliente! Esse é o erro!

10.1 Mais sobre Pilhas



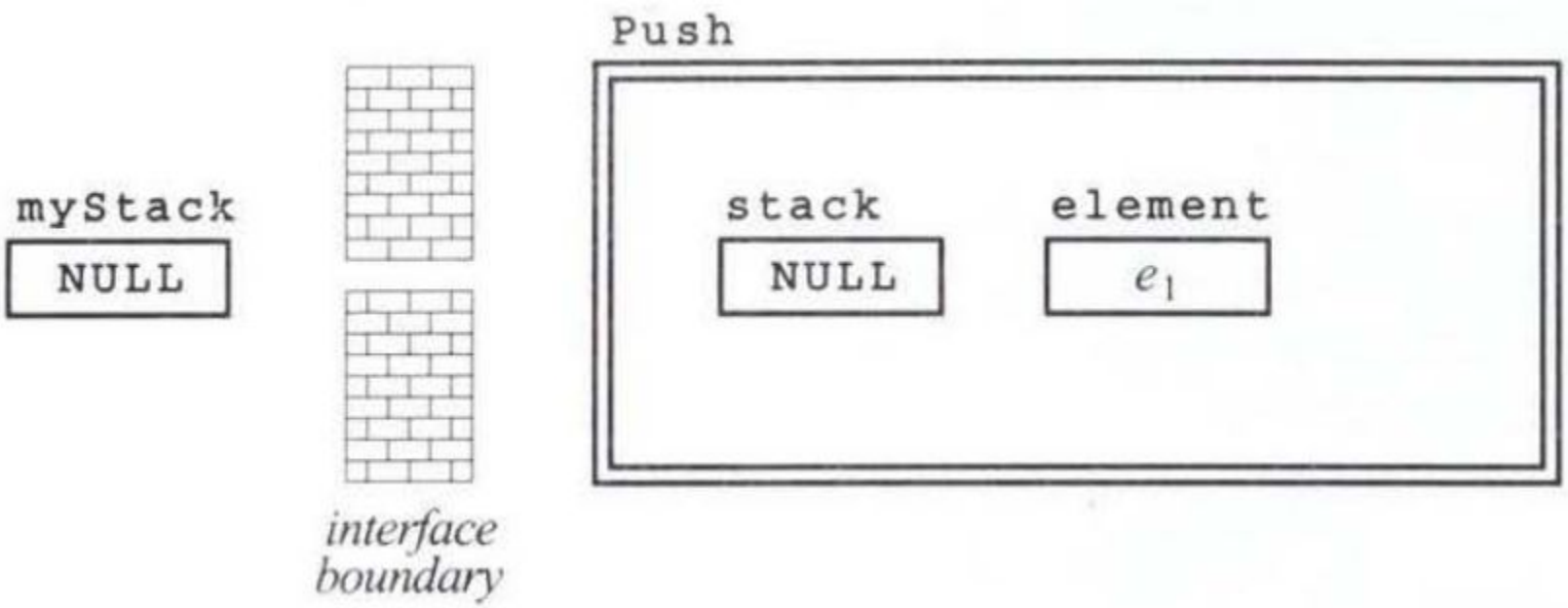
A função push até coloca o elemento no topo da pilha, mas o que ocorre depois?

10.1 Mais sobre Pilhas



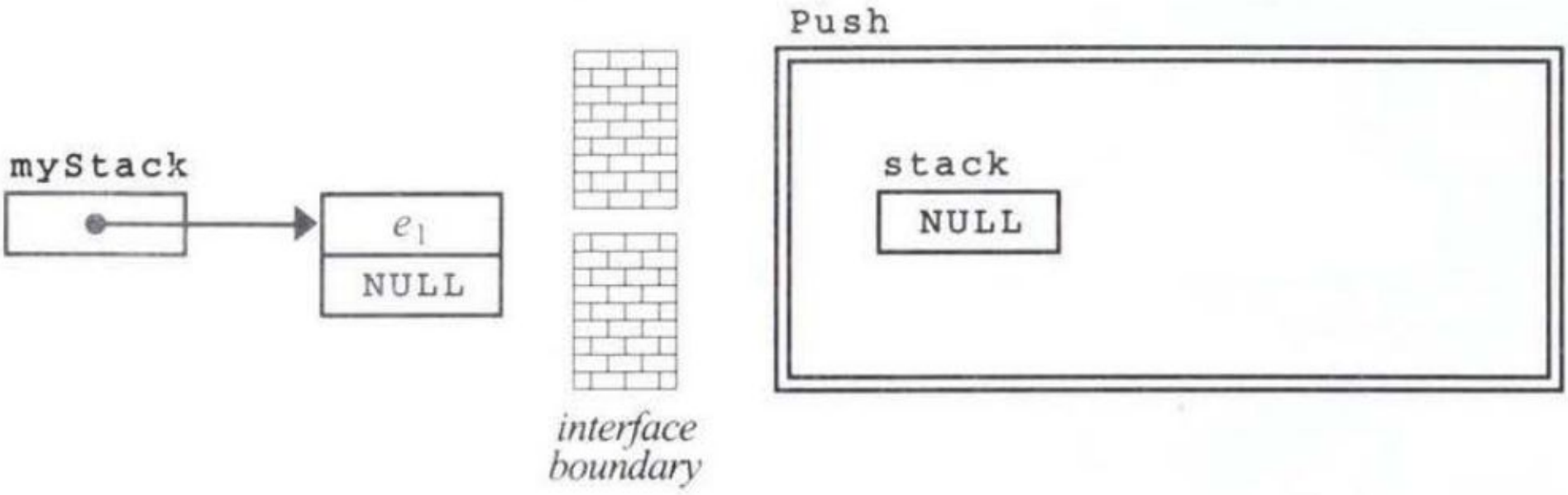
Quando a função retornar e o stack frame for removido, tudo será perdido!

10.1 Mais sobre Pilhas



O que **gostaríamos que ocorresse** é que, quando a função push recebesse uma cópia do stack e o elemento a ser inserido, ela inserisse o elemento no myStack do cliente. Se isso pudesse ser feito, seria algo que ficaria assim:

10.1 Mais sobre Pilhas



Mas isso é **impossível pela semântica da linguagem C**! A função push não pode alterar o valor do myStack, se ele foi passado como uma cópia!

10.1 Mais sobre Pilhas

- Como regra geral, se uma função exportada por uma interface precisa alterar o valor de qualquer dado associado com um TAD, **o dado deve ser acessível dentro do lado da implementação** na barreira da abstração.

- Uma solução geral para isso é garantir que o tipo concreto de dado tenha um endereço de memória que não se altera. **Qualquer localização na memória cujos valores sejam alterados em resposta a operações realizadas pela implementação devem estar embutidas dentro do tipo concreto de dado**.

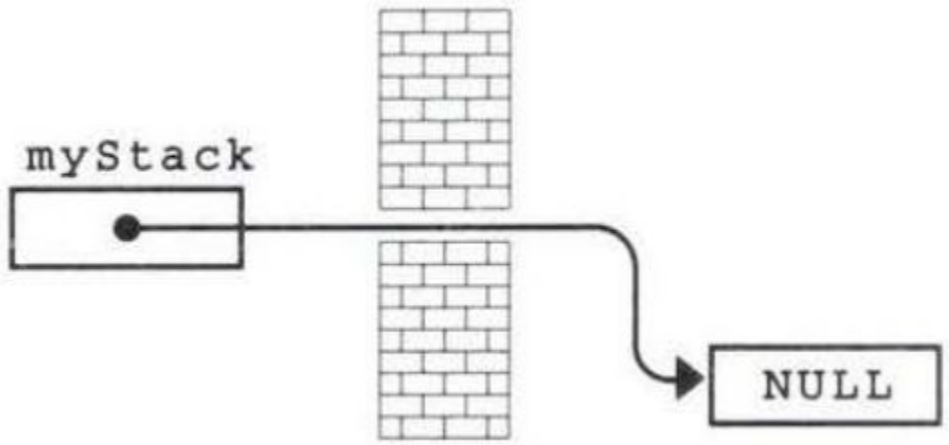
10.1 Mais sobre Pilhas

- Em nosso exemplo, temos que fazer isso:

```
typedef struct stackCDT *stackADT;

struct stackCDT {
    cellT *start;
};

stackADT NewStack(void)
{
    stackADT stack;

    stack = New(stackADT);
    stack->start = NULL;
    return (stack);
}
```

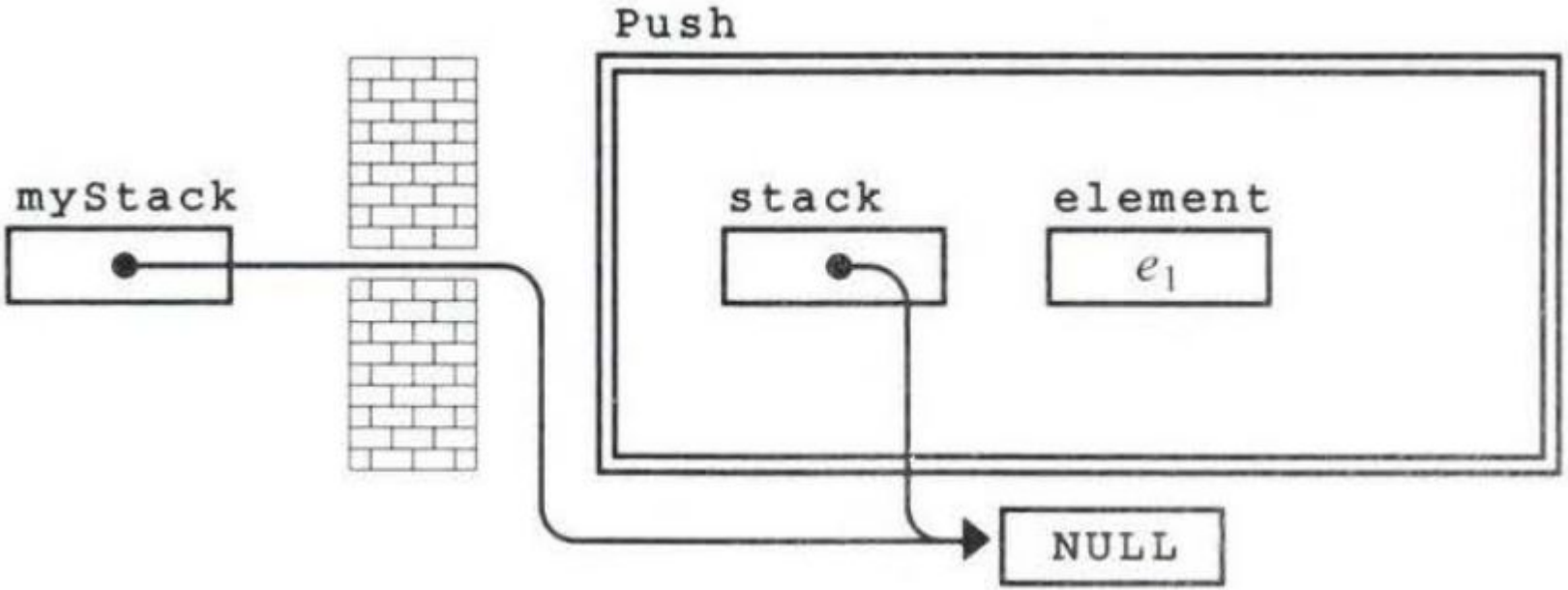
10.1 Mais sobre Pilhas

- Ao criar um stack, teríamos então a variável “myStack” apontando para o tipo concreto que está do lado da implementação na barreira da interface, o que significa que o cliente não tem acesso direto aos dados por si mesmo.



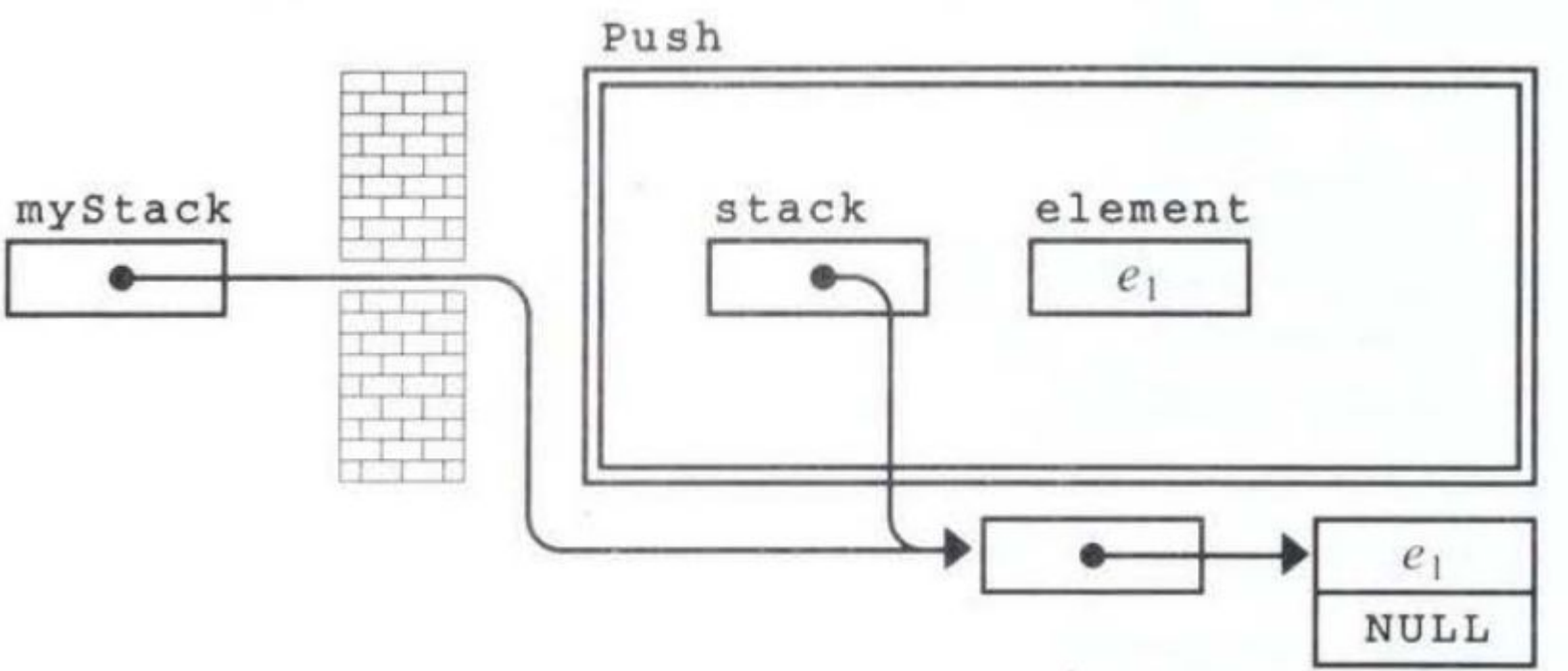
10.1 Mais sobre Pilhas

- Agora a função push, mesmo recebendo uma CÓPIA do ponteiro, tem acesso aos dados:



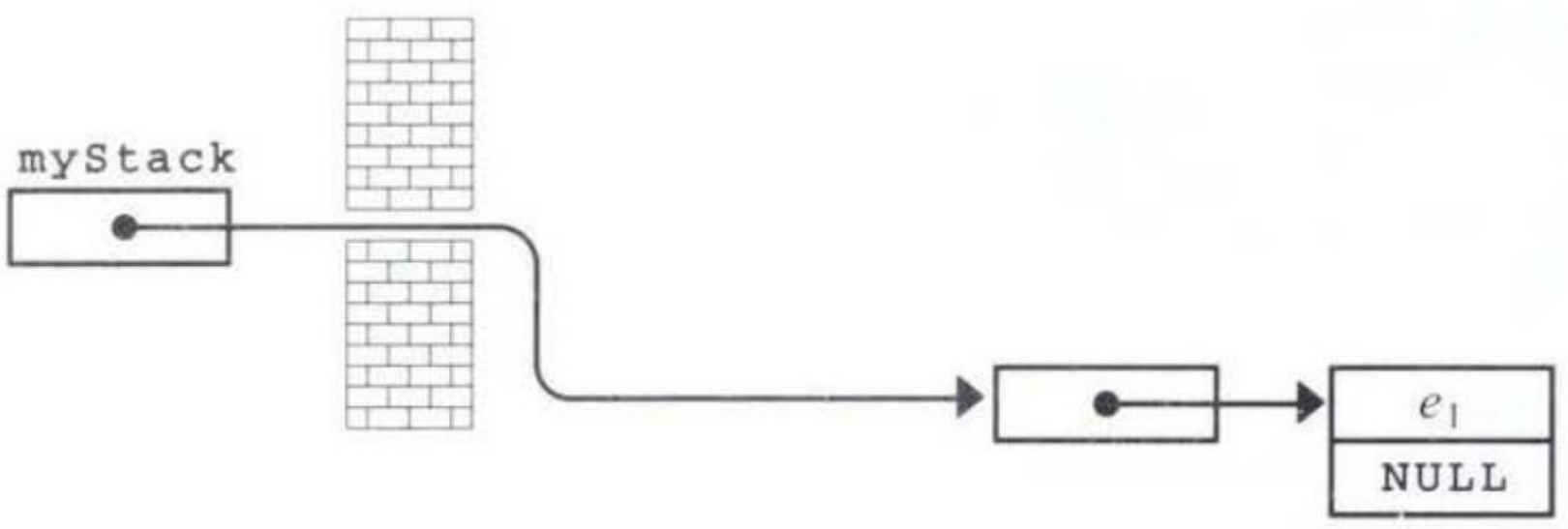
10.1 Mais sobre Pilhas

- A função push insere o elemento do topo da pilha, …



10.1 Mais sobre Pilhas

- … e quando o stack frame for eliminado a lista conterá o elemento no topo da pilha!



10.1 Mais sobre Pilhas

```
26 /*** Inicialização do Boilerplate da Interface ***/
27
28 #ifndef _STACKTAD_H
29 #define _STACKTAD_H
30
31 /*** Includes ***/
32
33 #include <stdbool.h>
34 #include <stdio.h>
35 #include <stdlib.h>
36
```

10.1 Mais sobre Pilhas

```
41 /**
42  * Tipo: elementoT
43  * ---------------
44  * O tipo "elementoT" é utilizado nesta interface para indicar o tipo de dado
45  * dos valores que serão armazenados no stack, ou seja, representa o tipo de
46  * dado dos elementos. Nesta interface, por padrão, o tipo de dado armazenado é
47  * "double", mas isso pode ser alterado ao se editar a linha da definição a
48  * seguir (é necessário recompilar a implementação da interface se esse tipo
49  * padrão for alterado).
50  */
51
52 typedef double elementoT;
53
```

10.1 Mais sobre Pilhas

```
54 /**
55  * Tipo abstrato: stackTAD
56  * -----------------------
57  * O tipo stackTAD repersenta um tipo abstrato que é uma pilha, para armazenar
58  * os elementos do tipo "elementoT". Como o stackTAD é definido apenas como um
59  * ponteiro para uma estrutura concreta que não está definida nesta interface,
60  * os clientes não têm acesso à implementação interna.
61  */
62
63 typedef struct stackTCD *stackTAD;
64
```

10.1 Mais sobre Pilhas

```
65 /**
66  * TIPO: stack_status
67  * ------------------
68  * Este tipo representa os possíveis status de ERRO ou SUCESSO nos retornos dos
69  * subprogramas que determinam o comportamento do TAD Pilha. Atualmente os
70  * seguintes valores são definidos:
71  *
72  *     STACK_OK             : operação realizada com sucesso
73  *     STACK_ERRO_ALOCACAO  : erro na alocação de memória par o stack
74  *     STACK_ERRO_STACK     : stack passado como argumento é inválido
75  *     STACK_ERRO_ARGUMENTO : argumento (não especificado) passado é inválido
76  *     STACK_ERRO_POSICAO   : posição especificada inválida
77  *     STACK_ERRO_CHEIO     : erro que indica pilha cheia
78  *     STACK_ERRO_VAZIO     : erro que indica pilha vazia
79  *     STACK_ERRO_NAOIMPL   : erro por subprograma ainda não implementado
80  */
81
82 typedef enum
83 {
84     STACK_OK,
85     STACK_ERRO_ALOCACAO,
86     STACK_ERRO_STACK,
87     STACK_ERRO_ARGUMENTO,
88     STACK_ERRO_POSICAO,
89     STACK_ERRO_CHEIO,
90     STACK_ERRO_VAZIO,
91     STACK_ERRO_NAOIMPL
92 } stack_status;
```

10.1 Mais sobre Pilhas

```
 96 /**
 97  * Função: criar_stackTAD
 98  * Uso: stackTAD = criar_stackTAD( );
 99  * ----------------------------------
100  * Esta função aloca e retorna um novo stack, que está inicialmente vazio. Se
101  * ocorrer algum erro na criação do stack, será retornado o valor NULL.
102  */
103
104 stackTAD
105 criar_stackTAD (void);
106
```

10.1 Mais sobre Pilhas

```
107 /**
108  * Função: remover_stackTAD
109  * Uso: status = remover_stackTAD(&stackTAD);
110  * ------------------------------------------
111  * Esta função recebe um PONTEIRO para uma pilha e libera toda memória alocada
112  * para o stackTAD. O retorno é o stack_status correspondente. Caso a remoção
113  * seja realizada com sucesso, o retorno será STACK_OK. Se ocorrer algum erro
114  * stack_status será o erro correspondente.
115  */
116
117 stack_status
118 remover_stackTAD (stackTAD *stack);
119
```

10.1 Mais sobre Pilhas

```
120 /**
121  * Função: push
122  * Uso: status = push(stack, elemento);
123  * ------------------------------------
124  * Esta função faz o push do "elemento" no "stack", retornando o status_stack
125  * correspondente. Caso a operação de push seja realizada com sucesso, a função
126  * retorna STACK_OK.
127  */
128
129 stack_status
130 push (stackTAD stack, const elementoT elemento);
```

10.1 Mais sobre Pilhas

```
132 /**
133  * Função: pop
134  * Uso: status = pop(stack, *elemento);
135  * ------------------------------------
136  * Esta função retira o elemento que está no topo da pilha "stack" e coloca
137  * o elemento no endereço apontado pelo PONTEIRO "elemento". O valor a ser
138  * retirado é sempre o último que entrou na pilha. A função retorna STACK_OK em
139  * caso de sucesso, ou retorna o stack_status de erro correspondente.
140  *
141  * Atenção: em caso de erro o valor em "elemento" não é alterado e não deve ser
142  * considerado válido ou confiável. O valor em "elemento" só pode ser
143  * considerado válido e confiável se o retorno for STACK_OK.
144  */
145
146 stack_status
147 pop (stackTAD stack, elementoT *elemento);
148
```

10.1 Mais sobre Pilhas

```
149 /**
150  * Função: vazio
151  * Uso: if(vazio(stack, &esta_vazio) == STACK_OK && esta_vazio = true) . . .
152  * ------------------------------------------------------------------------
153  * Esta função verifica se o stack está vazio, e recebe como argumentos a pilha
154  * "stack" e um PONTEIRO para bool, onde será armazenado o valor TRUE ou FALSE
155  * que indicará se a pilha está vazia (TRUE) ou não está vazia (FALSE). A função
156  * retorna o código STACK_OK se foi bem sucedida, ou um código de erro caso
157  * contrário.
158  *
159  * ATENÇÃO: caso o ponteiro "esta_vazio" seja NULL, será retornado o status
160  * STACK_ERRO_ARGUMENTO e nenhum valor será atribuído ao ponteiro. Se o ponteiro
161  * for válido mas ocorrer algum outro erro (por exemplo, stack inválido), a
162  * função retornará o stack_status STACK_ERRO_STACK e atribuirá o valor FALSE
163  * ao ponteiro "esta_vazio", ou seja: o valor armazenado em "esta_vazio" só é
164  * confiável se o stack_status retornado foi STACK_OK. Caso o stack_status
165  * retornado seja diferente disso, o valor em "esta_vazio" NÃO É CONFIÁVEL como
166  * indicador de pilha vazia ou não.
167  */
168
169 stack_status
170 vazio (const stackTAD stack, bool *esta_vazia);
```

10.1 Mais sobre Pilhas

```
172 /**
173  * Função: cheio
174  * Uso: if(cheio(stack, &esta_cheio) == STACK_OK && esta_cheio == true) . . .
175  * ------------------------------------------------------------------------
176  * Esta função verifica se o stack está cheio, e recebe como argumentos a pilha
177  * "stack" e um PONTEIRO para bool, onde será armazenado o valor TRUE ou FALSE
178  * que indicará se a pilha está cheio (TRUE) ou não está cheia (FALSE). A função
179  * retorna o código STACK_OK se foi bem sucedida, ou um código de erro caso
180  * contrário.
181  *
182  * ATENÇÃO: caso o ponteiro "esta_cheio" seja NULL, será retornado o status
183  * STACK_ERRO_ARGUMENTO e nenhum valor será atribuído ao ponteiro. Se o ponteiro
184  * for válido mas ocorrer algum outro erro (por exemplo, stack inválido), a
185  * função retornará o stack_status STACK_ERRO_STACK e atribuirá o valor FALSE
186  * ao ponteiro "esta_cheio", ou seja: o valor armazenado em "esta_cheio" só é
187  * confiável se o stack_status retornado foi STACK_OK. Caso o stack_status
188  * retornado seja diferente disso, o valor em "esta_cheio" NÃO É CONFIÁVEL como
189  * indicador de pilha cheia ou não.
190  */
191
192 stack_status
193 cheio (const stackTAD stack, bool *esta_cheio);
```

10.1 Mais sobre Pilhas

```
195 /**
196  * Função: tamanho
197  * Uso: status = tamanho(stack, &tam);
198  * ----------------------------------
199  * Esta função recebe como argumento um "stack" e o endereço apontado por um
200  * PONTEIRO para um long int, e armazena no local apontado o tamanho alocado
201  * do stack. As seguintes situações especiais podem ocorrer:
202  *
203  *    a) Se o ponteiro "tam" for NULL a função não faz nada e simplesmente
204  *       retorna STACK_ERRO_ARGUMENTO;
205  *    b) Se o ponteiro "tam" for válido mas houver algum erro com o stack (por
206  *       exemplo, stack inválido), será armazenado em "tam" o valor 0 (zero), e
207  *       a função retornará STACK_ERRO_STACK. Note que, nesse caso, o valor
208  *       armazenado em "tam" não tem significado algum.
209  *    c) Se o stack tiver tamanho fixo, será armazenado em "tam" o tamanho
210  *       alocado da pilha e a função retornará STACK_OK;
211  *    d) Se o stack tiver tamanho dinâmico será armazenado em "tam" o valor -1,
212  *       indicando que não há tamanho máximo fixo definido, e a função também
213  *       retornará STACK_OK.
214  *
215  * Note que o valor armazenado em "tam" somente terá validade se o retorno da
216  * função tenha sido STACK_OK. Em qualquer outra situação o valor armazenado em
217  * "tam" não é confiável.
218  */
219
220 stack_status
221 tamanho (const stackTAD stack, long int *tam);
```

10.1 Mais sobre Pilhas

```
223 /**
224  * Função: qtd_elementos
225  * Uso: status = qtd_elementos(stack, &nelem);
226  * ------------------------------------------
227  * Esta função recebe como argumento um "stack" e o endereço apontado por um
228  * PONTEIRO para um long int, e armazena no local apontado a quantidade atual de
229  * elementos armazenados no stack, ou seja, retorna o tamanho efetivo do stack.
230  * Duas situações especiais podem ocorrer:
231  *
232  *    a) Se o ponteiro "nelem" for NULL a função não faz nada e simplesmente
233  *       retorna STACK_ERRO_ARGUMENTO;
234  *    b) Se o ponteiro "nelem" for válido mas houver algum erro com o stack (por
235  *       exemplo, stack inválido), será armazenado em "nelem" o valor 0 (zero),
236  *       e a função retornará STACK_ERRO_STACK. Note que, nesse caso, o valor
237  *       armazenado em "nelem" não tem significado algum.
238  *    c) Se o "stack" e o "nelem" forem válidos, atribui ao endereço apontado
239  *       a quantidade atual de elementos na pilha, retornando STACK_OK.
240  *
241  * Note que o valor armazenado em "nelem" será confiável desde que o retorno da
242  * função tenha sido STACK_OK. Em qualquer outra situação o valor armazenado em
243  * "nelem" não é confiável.
244  */
245
246 stack_status
247 qtd_elementos (const stackTAD stack, long int *nelem);
```

10.1 Mais sobre Pilhas

```
249 /**
250  * FUNÇÃO: espaco_restante
251  * Uso: status = espaco_restante(stack, &espaco);
252  * ----------------------------------------------
253  * Esta função é utilizada geralmente para DEBUG, não é um comportamento padrão
254  * de um TAD pilha. Ela recebe um "stack" e o endereço apontado por um PONTEIRO
255  * para um long int, "espaco", e armazena nesse enderço a quantidade de "vagas"
256  * ainda disponíveis na pilha. As seguintes situações especiais podem ocorrer:
257  *
258  *    a) Se o ponteiro "espaco" for NULL, a função não faz nada e simplesmente
259  *       retorna STACK_ERRO_ARGUMENTO;
260  *    b) Se o ponteiro "espaco" for válido mas o stack não (por exemplo, stack
261  *       nulo ou inválido), a função armazena em "espaco" o valor 0 e retornará
262  *       o STACK_ERRO_STACK. Nessa situação o valor armazenado em "espaco" não
263  *       é válido e não tem significado nenhum, devendo ser desconsiderado.
264  *    c) Se "espaco" e "stack" foram válidos e a pilha tiver tamanho fixo, será
265  *       será armazenado em "espaco" o espaço restante na pilha, e a função
266  *       retornará STACK_OK;
267  *    d) Se o stack tiver tamanho dinâmico será armazenado em "espaco" o valor
268  *       -1, indicando que não há tamanho máximo fixo definido, e a função
269  *       retornará STACK_OK.
270  *
271  * Note que o valor armazenado em "espaco" será confiável desde que o retorno da
272  * função tenha sido STACK_OK. Em qualquer outra situação o valor armazenado em
273  * "espaco" não é confiável.
274  */
275
276 #ifdef debug
277 stack_status
278 espaco_restante (const stackTAD stack, long int *espaco);
279 #endif
```

10.1 Mais sobre Pilhas

```
281 /**
282  * FUNÇÃO: ver_elemento
283  * Uso: status = ver_elemento(stack, posicao, elementoT *elemento);
284  * ------------------------------------------------------------------
285  * Esta função é apenas para DEBUG, não é comportamento padrão de um TAD pilha.
286  * Ela recebe um "stack", uma "posição" (iniciada em 0) e um endereço apontado
287  * por um PONTEIRO para um elementoT, "elemento", e aramzena nesse endereço o
288  * elemento que está na posição especificada da pilha (não faz o pop de nenhum
289  * elemento). Se a função for concluída com sucesso retorna STACK_OK. As
290  * seguinte situações de erro são possíveis:
291  *
292  *     a) Se o ponteiro "elemento" for NULL, a função não faz nada e retorna
293  *        o erro STACK_ERRO_ARGUMENTO;
294  *     b) Se o "stack" for inválido, a função não faz nada e retorna o erro
295  *        STACK_ERRO_STACK;
296  *     c) Se o "stack" estiver vazio, a função não faz nada e retorna o erro
297  *        STACK_ERRO_VAZIO;
298  *     d) Se a "posicao" for inválida, a função não faz nada e retorna o erro
299  *        STACK_ERRO_POSICAO.
300  *
301  * Note que em caso de erro o conteúdo apontado por "elemento" é inválido, não
302  * tem significado algum.
303  */
304
305 #ifdef debug
306 stack_status
307 ver_elemento (const stackTAD stack, const size_t posicao, elementoT *elemento);
308 #endif
```

10.1 Mais sobre Pilhas

```
310 /**
311  * Função: imprimir_stack
312  * Uso: (void) imprimir_stack(stack, limite);
313  * ------------------------------------------
314  * TODO: esta função está definida mas ainda não implementado. Quando for
315  * implementado será utilizada para DEBUG pois não é comportamento padrão de um
316  * TAD pilha. Ela imprime os elementos atuais da pilha, até uma quantidade
317  * limite (para evitar que o terminal do usuário recebe várias e várias linhas).
318  * Atualmente retorna o erro STACK_ERRO_NAOIMPL.
319  */
320
321 #ifdef debug
322 stack_status
323 imprimir_stack (const stackTAD stack, const size_t limite);
324 #endif
```

10.1 Mais sobre Pilhas

```
46 struct celulaTCD
47 {
48     elementoT elemento;
49     struct celulaTCD *proximo;
50 };
51
52 typedef struct celulaTCD *celulaTAD;
```

10.1 Mais sobre Pilhas

```
64 struct stackTCD
65 {
66     celulaTAD inicio;
67     size_t nelem;
68 };
```

10.1 Mais sobre Pilhas

```
87 stackTAD
88 criar_stackTAD (void)
89 {
90     stackTAD S = calloc(1, sizeof(struct stackTCD));
91     if (S == NULL)
92         return NULL;
93
94     S->inicio = NULL;
95     S->nelem = 0;
96     return S;
97 }
98
```

10.1 Mais sobre Pilhas

```
109 stack_status
110 remover_stackTAD (stackTAD *stack)
111 {
112     if (stack == NULL || *stack == NULL)
113         return STACK_ERRO_STACK;
114
115     celulaTAD atual, proxima;
116
117     atual = (*stack)->inicio;
118     while (atual != NULL)
119     {
120         proxima = atual->proximo;
121         remover_celula(&atual);
122         atual = proxima;
123     }
124     free(*stack);
125     *stack = NULL;
126
127     return STACK_OK;
128 }
```

10.1 Mais sobre Pilhas

```
139 stack_status
140 push (stackTAD stack, const elementoT elemento)
141 {
142     if (stack == NULL)
143         return STACK_ERRO_STACK;
144
145     celulaTAD temp = criar_celula();
146     if (temp == NULL)
147         return STACK_ERRO_ALOCACAO;
148
149     temp->elemento = elemento;
150     temp->proximo = stack->inicio;
151     stack->inicio = temp;
152     stack->nelem += 1;
153
154     return STACK_OK;
155 }
```

10.1 Mais sobre Pilhas

```
172 stack_status
173 pop (stackTAD stack, elementoT *elemento)
174 {
175
176     if (stack == NULL)
177         return STACK_ERRO_STACK;
178     else if (elemento == NULL)
179         return STACK_ERRO_ARGUMENTO;
180
181     bool esta_vazio;
182     if (vazio(stack, &esta_vazio) == STACK_OK && esta_vazio == true)
183         return STACK_ERRO_VAZIO;
184
185     *elemento = stack->inicio->elemento;
186
187     celulaTAD temp = stack->inicio;
188     stack->inicio = stack->inicio->proximo;
189     stack->nelem -= 1;
190     remover_celula(&temp);
191
192     return STACK_OK;
193 }
```

10.1 Mais sobre Pilhas

```
207 stack_status
208 vazio (const stackTAD stack, bool *esta_vazia)
209 {
210     if (esta_vazia == NULL)
211         return STACK_ERRO_ARGUMENTO;
212     else if (stack == NULL)
213     {
214         *esta_vazia = false;
215         return STACK_ERRO_STACK;
216     }
217
218     *esta_vazia = stack->nelem == 0;
219     return STACK_OK;
220 }
```

10.1 Mais sobre Pilhas

```
235 stack_status
236 cheio (const stackTAD stack, bool *esta_cheio)
237 {
238     if (esta_cheio == NULL)
239         return STACK_ERRO_ARGUMENTO;
240     else if (stack == NULL)
241     {
242         *esta_cheio = false;
243         return STACK_ERRO_STACK;
244     }
245
246     *esta_cheio = false;
247     return STACK_OK;
248 }
```

10.1 Mais sobre Pilhas

```
274 stack_status
275 tamanho (const stackTAD stack, long int *tam)
276 {
277     if (tam == NULL)
278         return STACK_ERRO_ARGUMENTO;
279     else if (stack == NULL)
280     {
281         *tam = 0;
282         return STACK_ERRO_STACK;
283     }
284
285     *tam = -1;
286     return STACK_OK;
287 }
288
```

10.1 Mais sobre Pilhas

```
312 stack_status
313 qtd_elementos (const stackTAD stack, long int *nelem)
314 {
315     if (nelem == NULL)
316         return STACK_ERRO_ARGUMENTO;
317     else if (stack == NULL)
318     {
319         *nelem = 0;
320         return STACK_ERRO_STACK;
321     }
322
323     *nelem = stack->nelem;
324     return STACK_OK;
325 }
```

10.1 Mais sobre Pilhas

```
354 #ifdef debug
355 stack_status
356 espaco_restante (const stackTAD stack, long int *espaco)
357 {
358     if (espaco == NULL)
359         return STACK_ERRO_ARGUMENTO;
360     else if (stack == NULL)
361     {
362         *espaco = 0;
363         return STACK_ERRO_STACK;
364     }
365
366     *espaco = -1;
367     return STACK_OK;
368 }
369 #endif
```

10.1 Mais sobre Pilhas

```
395 #ifdef debug
396 stack_status
397 ver_elemento (const stackTAD stack, const size_t posicao, elementoT *elemento)
398 {
399     bool esta_vazio;
400 
401     if (elemento == NULL)
402         return STACK_ERRO_ARGUMENTO;
403     else if (stack == NULL)
404         return STACK_ERRO_STACK;
405     else if (vazio(stack, &esta_vazio) == STACK_OK && esta_vazio == true)
406         return STACK_ERRO_VAZIO;
407     else if (posicao >= stack->nelem)
408         return STACK_ERRO_POSICAO;
409 
410     celulaTAD temp = stack->inicio;
411     for (size_t i = 0; i < posicao; i++)
412     {
413         temp = temp->proximo;
414     }
415     *elemento = temp->elemento;
416 
417     return STACK_OK;
418 }
419 #endif
```

10.1 Mais sobre Pilhas

```
452 static celulaTAD
453 criar_celula (void)
454 {
455     celulaTAD temp = calloc(1, sizeof(struct celulaTCD));
456     if (temp == NULL)
457         return NULL;
458
459     return temp;
460 }
```

10.1 Mais sobre Pilhas

```
470 static void
471 remover_celula (celulaTAD *celula)
472 {
473     if (celula && *celula)
474     {
475         free(*celula);
476         *celula = NULL;
477     }
478 }
```

10.2 Queue (fila)

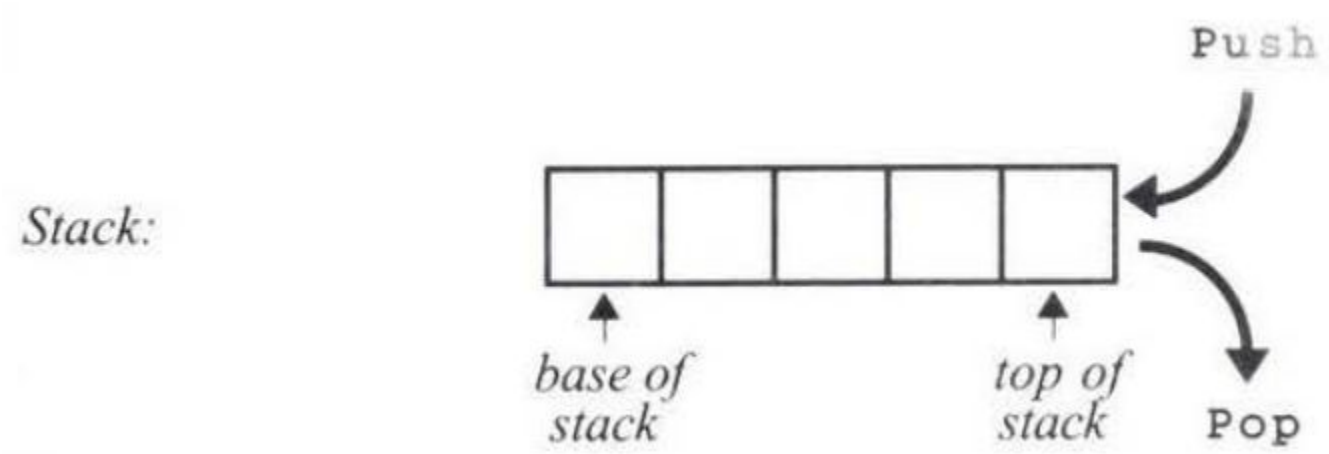
- É um TAD linear da categoria dos **containers**, ou seja, é um TAD que **permite armazenar e recuperar dados independentemente de seu valor/conteúdo**.

- A principal característica do comportamento de uma fila é que **os dados só podem ser retirados da fila na mesma ordem em que foram adicionados**, ou seja, de modo **FIFO** (first in, first out).

- O comportamento da fila é dado por 2 subprogramas principais e outros acessórios:
  - **enqueue(fila, item)** — insere o item no fim da fila
  - **dequeue(fila)** — remove o item do começo da fila
  - vazia(fila) — verifica se a fila está vazia
  - cheia(fila) — verifica se a fila está cheia
  - quantidade(fila) — verifica quantos elementos estão na fila

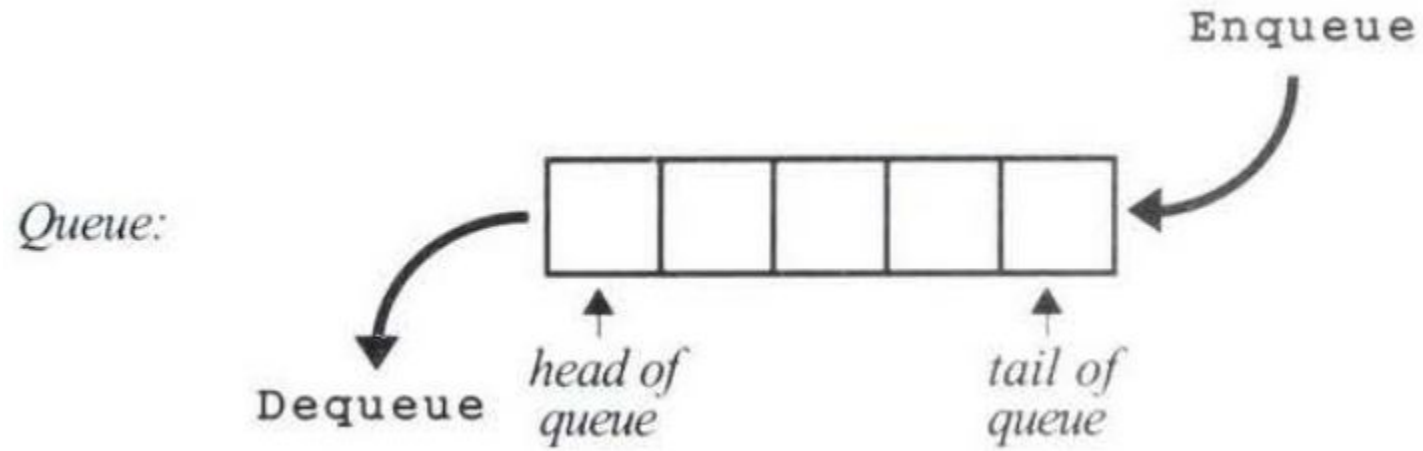
10.2 Queue (fila)

- Comparação entre uma pilha e uma fila:



10.2 Queue (fila)

- Comparação entre uma pilha e uma fila:



10.2.1 queueTAD.h (com tipo de dado genérico!)

```
38 /**
39  * Tipo: elementoT
40  * ---------------
41  * O tipo elementoT é utilizado nesta inferface para indicar o tipo de valor que
42  * será armazenado na fila. Por padrão a fila é utilizada para armazenar valores
43  * do tipo "void *" (tornando a fila genérica), mas isso pode ser modificado
44  * alterando-se a defição na linha abaixo.
45  */
46
47 typedef void *elementoT;
48
49 /**
50  * Tipo abstrato: queueTAD
51  * -----------------------
52  * O tipo "queueTAD" é um tipo abstrado de dado para representar uma fila. É
53  * definido como um ponteiro para queueTCD (o tipo concreto que implementa a
54  * fila), que está disponível apenas para a implementação, não para os clientes.
55  */
56
57 typedef struct queueTCD *queueTAD;
58
```

10.2.1 queueTAD.h (com tipo de dado genérico!)

```
59 /**
60  * Tipo: queue_status
61  * ------------------
62  * Define uma enumeração com os possíveis status de retorno das funções desta
63  * interface, incluindo o status de sucesso e os diversos status de erro que
64  * podem ser retornados. Os seguintes membros estão definidos:
65  *
66  *     QUEUE_OK                : operação realizada com sucesso
67  *     QUEUE_ERRO_QUEUE        : queue inválida
68  *     QUEUE_ERRO_ALOCACAO     : erro na alocação de memória
69  *     QUEUE_ERRO_ARGUMENTO    : argumento inválido
70  *     QUEUE_ERRO_POSICAO      : posição inválida
71  *     QUEUE_ERRO_CHEIA        : fila cheia
72  *     QUEUE_ERRO_VAZIA        : fila vazia
73  */
74
75 typedef enum
76 {
77     QUEUE_OK,
78     QUEUE_ERRO_QUEUE,
79     QUEUE_ERRO_ALOCACAO,
80     QUEUE_ERRO_ARGUMENTO,
81     QUEUE_ERRO_POSICAO,
82     QUEUE_ERRO_CHEIA,
83     QUEUE_ERRO_VAZIA
84 } queue_status;
```

10.2.1 queueTAD.h (com tipo de dado genérico!)

```
 88 /**
 89  * Função: CRIAR_QUEUE
 90  * Uso: queue = criar_queue( );
 91  * ----------------------------
 92  * Aloca e retorna uma fila vazia. Se não for possível criar a fila, retorna o
 93  * valor NULL.
 94  */
 95
 96 queueTAD
 97 criar_queue (void);
 98
 99 /**
100  * Função: REMOVER_QUEUE
101  * Uso: status = remover_queue(&queue);
102  * ------------------------------------
103  * Recebe um PONTEIRO para um queueTAD e faz a liberação de todas as estruturas
104  * de memória utilizadas pela fila (remove a fila e todos os seus elementos). Os
105  * possíveis retornos são:
106  *
107  *     a) QUEUE_OK: operação realizada com sucesso (o ponteiro "queue" informado
108  *        será direcionado para NULL);
109  *     b) QUEUE_ERRO_ARGUMENTO: ponteiro passado como argumento não é válido; e
110  *     c) QUEUE_ERRO_QUEUE: queue inválida.
111  *
112  * A remoção da queue e liberação correta da memória só ocorrem se a função
113  * retornar QUEUE_OK. É responsabilidade do cliente passar um ponteiro válido, e
114  * que aponte para uma queue válida.
115  */
116
117 queue_status
118 remover_queue (queueTAD *queue);
```

10.2.1 queueTAD.h (com tipo de dado genérico!)

```
120 /**
121  * Função: ENQUEUE
122  * Uso: status = enqueue(queue, elemento);
123  * ---------------------------------------
124  * Recebe uma "queue" e um "elemento", e enfileira o elemento no final da fila.
125  * Os possíveis retornos são:
126  *
127  *     a) QUEUE_OK: operação realizada com sucesso;
128  *     b) QUEUE_ERRO_QUEUE: queue inválida; e
129  *     c) QUEUE_ERRO_ARGUMENTO: elemento inválido.
130  */
131
132 queue_status
133 enqueue (queueTAD queue, const elementoT elemento);
134
```

10.2.1 queueTAD.h (com tipo de dado genérico!)

```
135 /**
136  * Função: DEQUEUE
137  * Uso: status = dequeue(queue, &elemento);
138  * ----------------------------------------
139  * Recebe uma "queue" e um PONTEIRO para um "elemento". Se a operação for
140  * realizada com sucesso, desenfileira o elemento no início da fila e coloca
141  * esse elemento no endereço apontado por "elemento".
142  * Os possíveis retornos são:
143  *
144  *     a) QUEUE_OK: operação realizada com sucesso;
145  *     b) QUEUE_ERRO_QUEUE: queue inválida;
146  *     c) QUEUE_ERRO_ARGUMENTO: ponteiro elemento inválido; e
147  *     d) QUEUE_ERRO_VAZIA: queue vazia.
148  *
149  * O valor armazenado no local apontado por "elemento" só é válido e confiável
150  * se a função tiver retornado QUEUE_OK.
151  */
152
153 queue_status
154 dequeue (queueTAD queue, elementoT *elemento);
```

10.2.1 queueTAD.h (com tipo de dado genérico!)

```
156 /**
157  * Função: VAZIA
158  * Uso: if (vazia(queue, &esta_vazia) == QUEUE_OK && esta_vazia == true) . . .
159  * ------------------------------------------------------------------------
160  * Recebe uma "queue" e um PONTEIRO para um booleano "esta_vazia", e retorna
161  * valores que nos permitem identificar se a fila está vazia ou não (ou, se
162  * ocorrer algum erro, permitem identificar esse erro). Os seguintes retornos
163  * são possíveis:
164  *
165  *     a) QUEUE_OK: operação realizada com sucesso; "esta_vazia" contém o valor
166  *        booleano indicativo de fila vazia (true) ou não vazia (false);
167  *     b) QUEUE_ERRO_ARGUMENTO: ponteiro "esta_vazia" inválido; e
168  *     c) QUEUE_ERRO_QUEUE: queue inválida.
169  *
170  * O valor true ou false armazenado em "esta_vazia" só é válido se o retorno da
171  * função tiver sido QUEUE_OK.
172  */
173
174 queue_status
175 vazia (const queueTAD queue, bool *esta_vazia);
176
```

10.2.1 queueTAD.h (com tipo de dado genérico!)

```
177 /**
178  * Função: CHEIA
179  * Uso: if (cheia(queue, &esta_cheia) == QUEUE_OK && esta_cheia == true) . . .
180  * ------------------------------------------------------------------------
181  * Recebe uma "queue" e um PONTEIRO para um booleano "esta_cheia", e retorna
182  * valores que nos permitem identificar se a fila está cheia ou não (ou, se
183  * ocorrer algum erro, permitem identificar esse erro). Os seguintes retornos
184  * são possíveis, nas seguintes situações:
185  *
186  *     a) QUEUE_OK: operação realizada com sucesso; "esta_cheia" contém o valor
187  *        booleano indicativo de fila cheia (true) ou não cheia (false);
188  *     b) QUEUE_ERRO_ARGUMENTO: ponteiro "esta_cheia" inválido;
189  *     c) QUEUE_ERRO_QUEUE: queue inválida.
190  *
191  * A implementação definirá se a fila é fixa ou dinâmica e, caso a fila seja
192  * dinâmica, esta função sempre retornará false se as operações de aumento de
193  * tamanho forem realizadas com sucesso. Se a fila não puder ser aumentada de
194  * tamanho por algum problema ou limitação de recursos, a função colocará true
195  * em "esta_cheia" e retornará o erro: QUEUE_ERRO_ALOCACAO.
196  *
197  * O valor true ou false armazenado em "esta_cheia" só é válido se o retorno da
198  * função tiver sido QUEUE_OK.
199  */
200
201 queue_status
202 cheia (const queueTAD queue, bool *esta_cheia);
```

10.2.1 queueTAD.h (com tipo de dado genérico!)

```
204 /**
205  * Função: NUM_ELEMENTOS
206  * Uso: status = num_elementos(queue, &nelem);
207  * ----------------------------------------
208  * Recebe uma "queue" e armazena no local apontado pelo ponteiro "nelem" o
209  * tamanho efetivo da fila ou seja, a quantidade atual de elementos armazenados
210  * na fila. Os seguintes retornos são possíveis nas seguintes situaçoes:
211  *
212  *     a) QUEUE_OK: operação realizada com sucesso; "nelem" contém a quantidade
213  *        atual de elementos na fila;
214  *     b) QUEUE_ERRO_ARGUMENTO: ponteiro "nelem" inválido; e
215  *     c) QUEUE_ERRO_QUEUE: queue inválida.
216  */
217
218 queue_status
219 num_elementos (const queueTAD queue, size_t *nelem);
```

10.2.1 queueTAD.h (com tipo de dado genérico!)

```
221 /**
222  * Função: INFO
223  * Uso: status = info(queue, &din, &tamax);
224  * ----------------------------------------
225  * Esta função não faz parte dos comportamentos normais esperados para uma fila
226  * mas é definida nesta interface para que o cliente possa obter diversas
227  * informações sobre a fila e sua implementação interna como, por exemplo, se a
228  * impementação se dá através de uma fila fixa (o tamanho máximo alocado não
229  * se altera) ou de uma fila dinâmica (o tamanho alocado aumenta automaticamente
230  * em tempo de execução e, assim, não há um tamanho máximo). Os seguintes
231  * retornos são possíveis:
232  *
233  *     a) QUEUE_OK: operação realizada com sucesso;
234  *             - "din" conterá true se a fila for dinâmica, ou false caso não; e
235  *             - "tamax" conterá -1 se a fila for dinâmica, ou o tamanho máximo
236  *               definido para a fila (de tamanho fixo).
237  *     b) QUEUE_ERRO_ARGUMENTO: ponteiro "din" ou "tammax" inválido (nesse caso
238  *        o próprio cliente deve verificar a validade dos ponteiros antes de
239  *        passá-los para esta função);
240  *     c) QUEUE_ERRO_QUEUE: queue inválida.
241  *
242  * Os valores armazenados em "din" e "tamax" só são válidos e confiáveis se o
243  * retorno da função tiver sido QUEUE_OK.
244  */
245
246 queue_status
247 info (const queueTAD queue, bool *din, int *tamax);
248
```

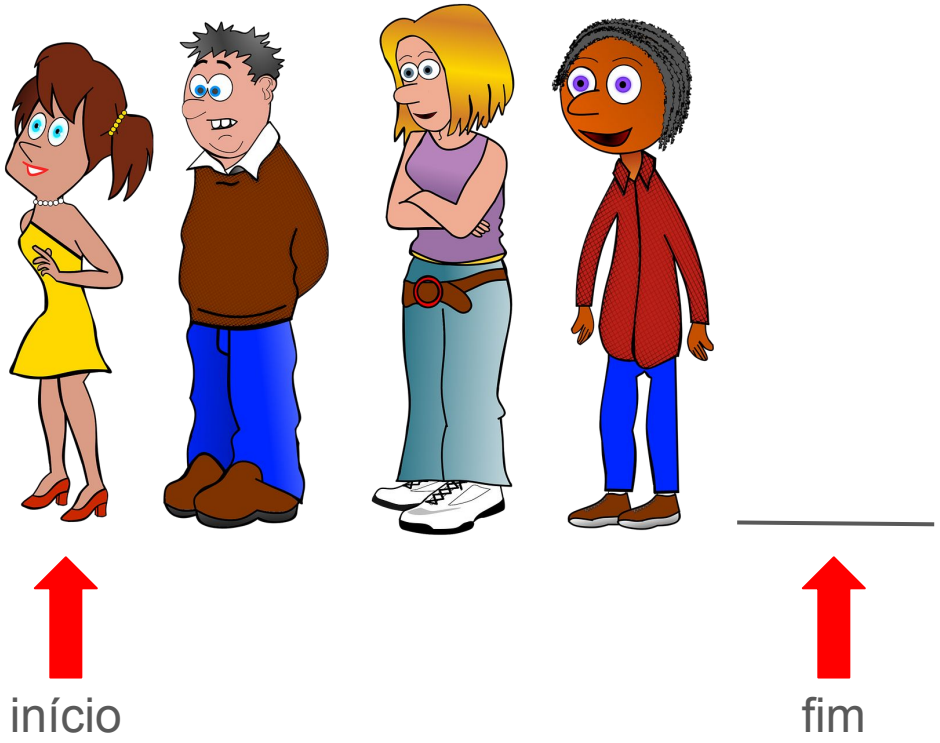
10.2.1 queueTAD.h (com tipo de dado genérico!)

```
249 /**
250  * Função: VER_ELEMENTO
251  * Uso: status = ver_elemento(queue, posicao, &elemento);
252  * ------------------------------------------------------
253  * Esta função NÃO ESTÁ PRESENTE em situações normais de uso da fila, mas é
254  * definida aqui apenas para ser utilizada em situações de DEBUG, pois não é um
255  * comportamento padrão de uma fila. A função retorna o elemento especificado na
256  * "posicao" no endereço apontado por "elemento", SEM DESENFILEIRAR o elmento. O
257  * início da fila é definido como posição 0 (zero). Os seguintes retornos são
258  * possíveis:
259  *
260  *     a) QUEUE_OK: operação realizada com sucesso; "elemento" contém o elemento
261  *        na posição indicada, sem desenfileirar;
262  *     b) QUEUE_ERRO_ARGUMENTO: ponteiro "elemento" inválido;
263  *     c) QUEUE_ERRO_QUEUE: queue inválida; e
264  *     d) QUEUE_ERRO_POSICAO: posição inválida.
265  *
266  * O cliente deve informar uma posição não negativa válida, ou seja, de 0 até
267  * (nelem - 1).
268  */
269
270 #ifdef debug
271 queue_status
272 ver_elemento (const queueTAD queue, const size_t posicao, elementoT *elemento);
273 #endif
```

10.2.2 Implementação de uma fila: possibilidades

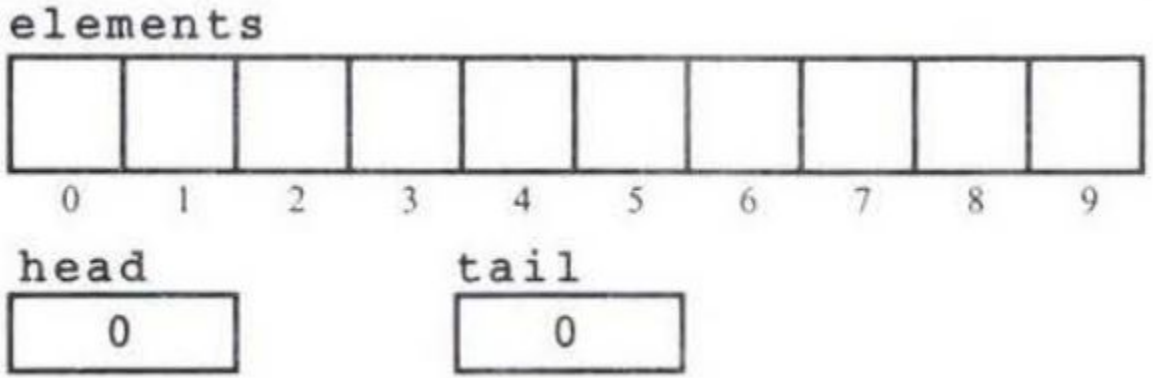
- Como a fila é um TAD linear, duas implementações óbvias são possíveis:
    - Usar um array
        - Bom para filas de tamanho fixo
        - Implementação ligeiramente mais complexa
    - Usar uma lista simplesmente encadeada (LSE)
        - Bom para filas de tamanho dinâmico (sem tamanho máximo)
        - Implementação ligeiramente mais simples

10.2.2 queueTAD_array.c (implementação com array)



10.2.2 queueTAD_array.c (implementação com array)

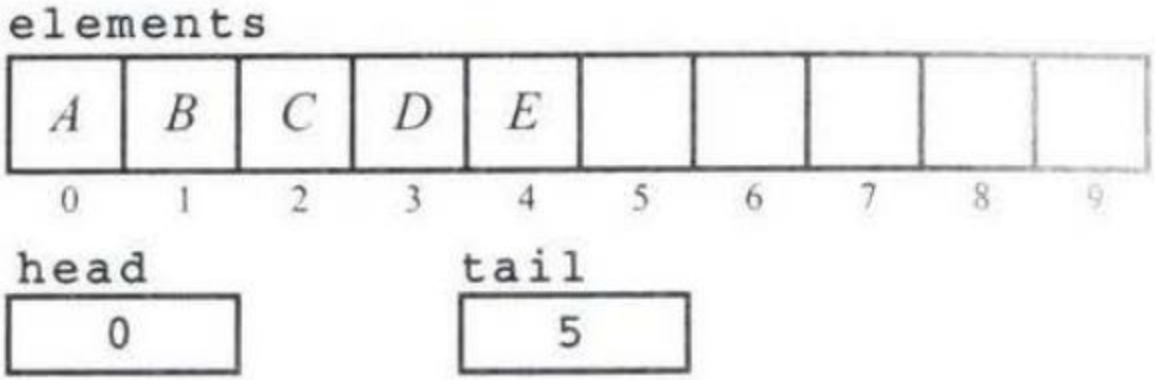
- A implementação com array tem uma dificuldade sutil. Vamos considerar que temos uma fila para 10 elementos. A fila vazia está assim:



Note que quando a fila está vazia, “início” = “fim”
(cuidado com essa interpretação!)

10.2.2 queueTAD_array.c (implementação com array)

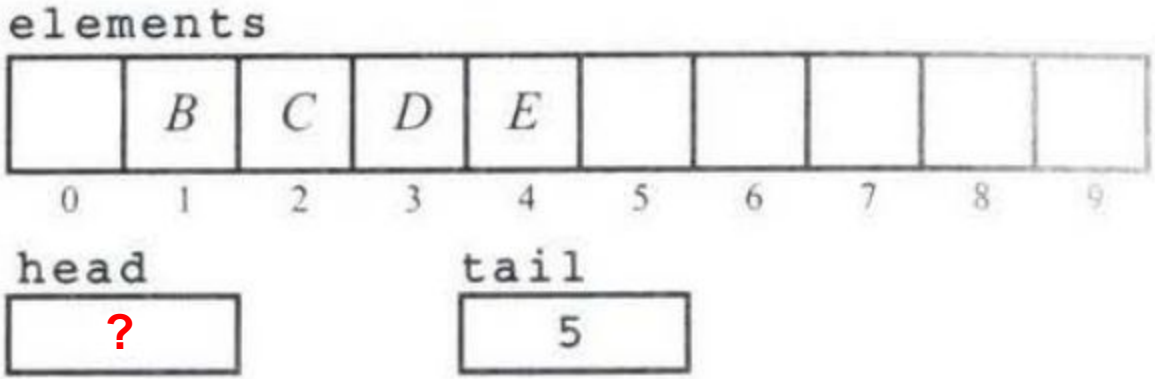
- Depois do enfileiramento de 5 elementos, nossa fila está assim:



Note que “início” permaneceu indicando o início da fila, e que “fim” aponta para o próximo local vago na fila - o final da fila.

10.2.2 queueTAD_array.c (implementação com array)

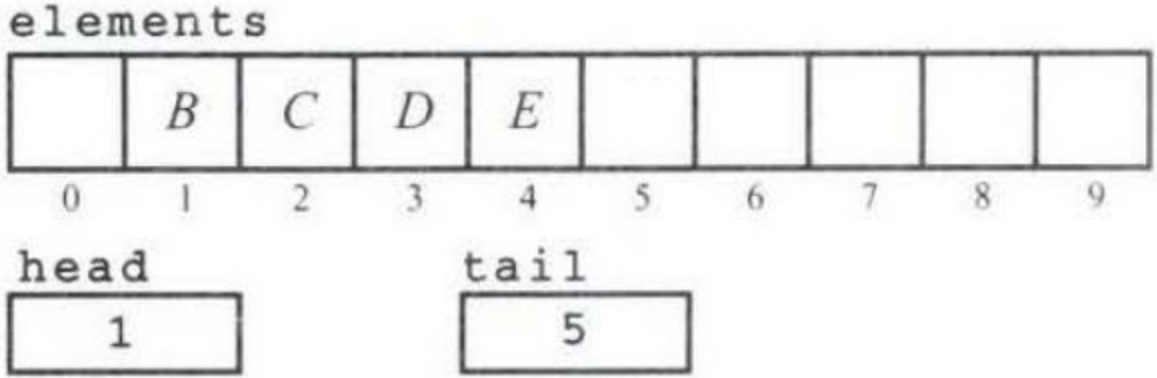
- Se o primeiro elemento sair da fila, qual deve ser o novo começo?



Devemos deslocar todos os elementos para a esquerda, ou devemos deslocar o índice “início”?

10.2.2 queueTAD_array.c (implementação com array)

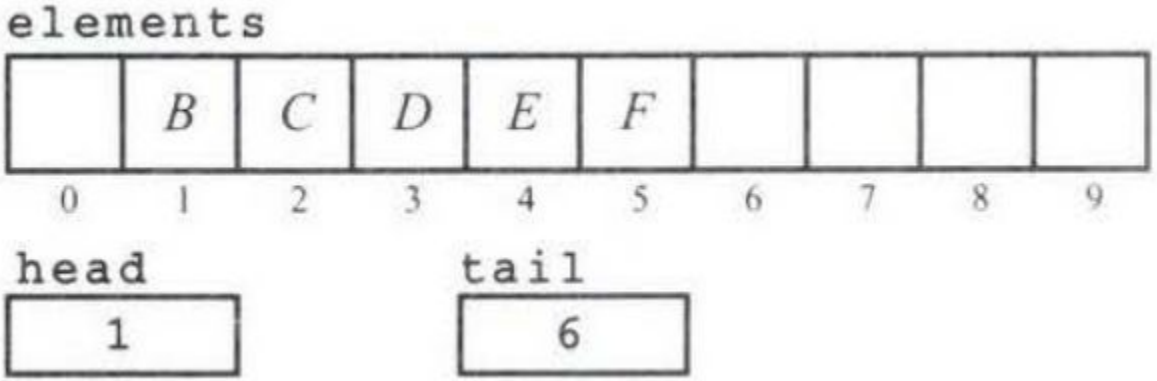
- Se o primeiro elemento sair da fila, qual deve ser o novo começo?



Devemos deslocar o índice, pois isso é O(1); deslocar todos os elementos para a esquerda seria O(N).

10.2.2 queueTAD_array.c (implementação com array)

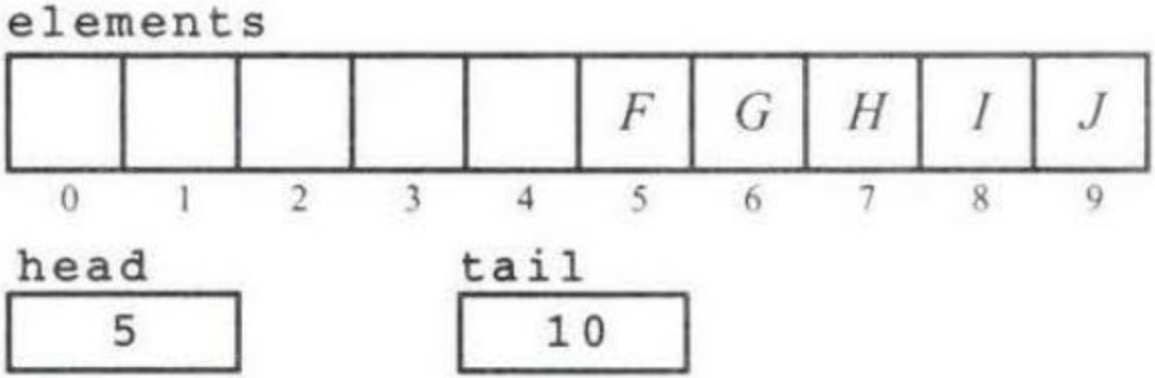
- Se um novo elemento for enfileirado, temos:



Esse é o processo normal: se um elemento SAIR da fila, aumentamos o índice “início”; se um elemento ENTRAR na fila, aumentamos o índice “fim” para o próximo lugar livre.

10.2.2 queueTAD_array.c (implementação com array)

- E quando chegarmos nessa situação?

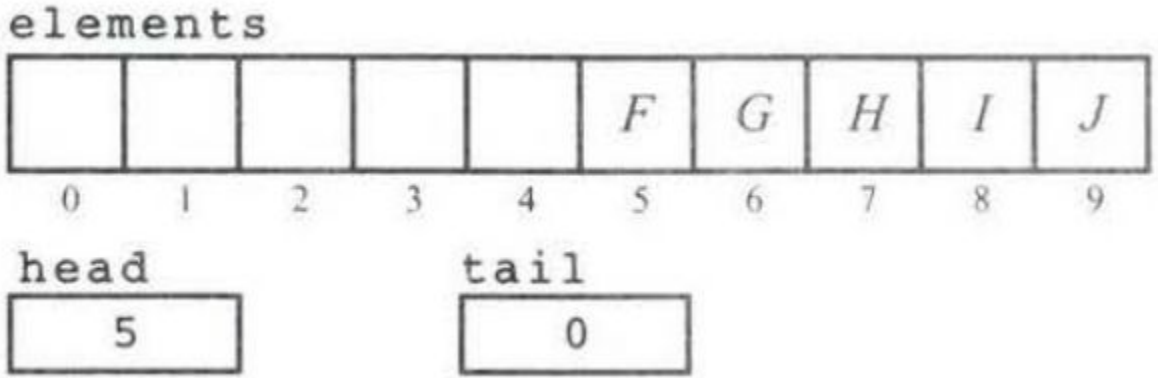


A posição apontada por “fim” não existe no array. Nesse caso:

- a fila está cheia?
- temos que aumentar o array?

10.2.2 queueTAD_array.c (implementação com array)

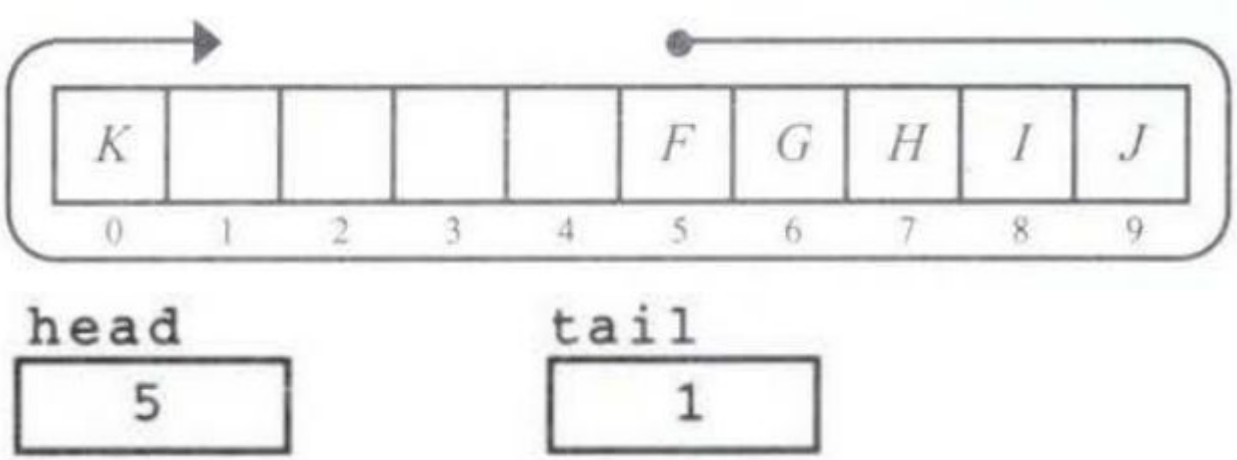
- **O “fim” dá a volta no array e apontará para a primeira posição do array!**



A fila não está cheia, ainda temos 5 lugares! A dificuldade com a implementação com arrays é visualizar que, agora, o final está em uma posição menor do que o início!

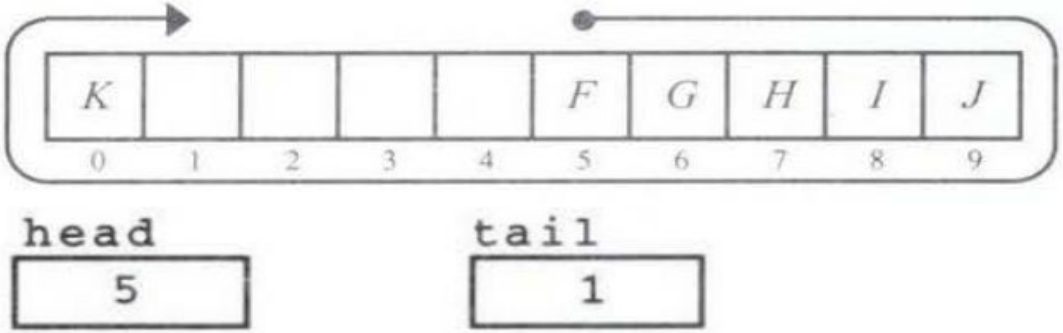
10.2.2 queueTAD_array.c (implementação com array)

- Se enfileirarmos mais 1 elemento, teremos o seguinte:

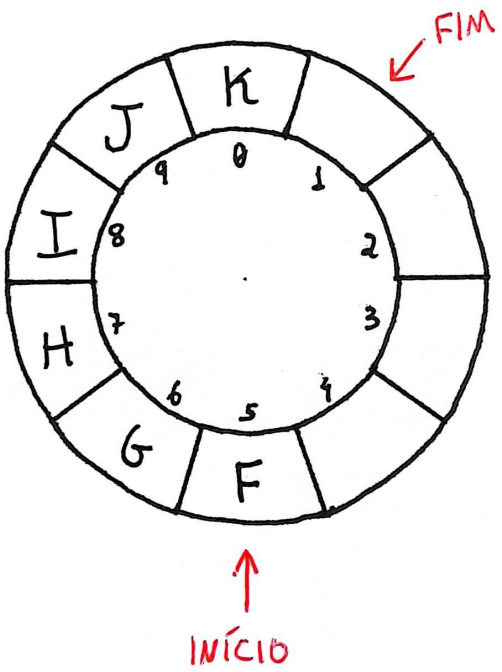


Note que, apesar de parecer descontínua, a fila É CONTÍNUA: começa no índice “início” e termina no índice “fim”! isso é um **ring buffer**, também chamado de **array circular**.

10.2.2 queueTAD_array.c (implementação com array)

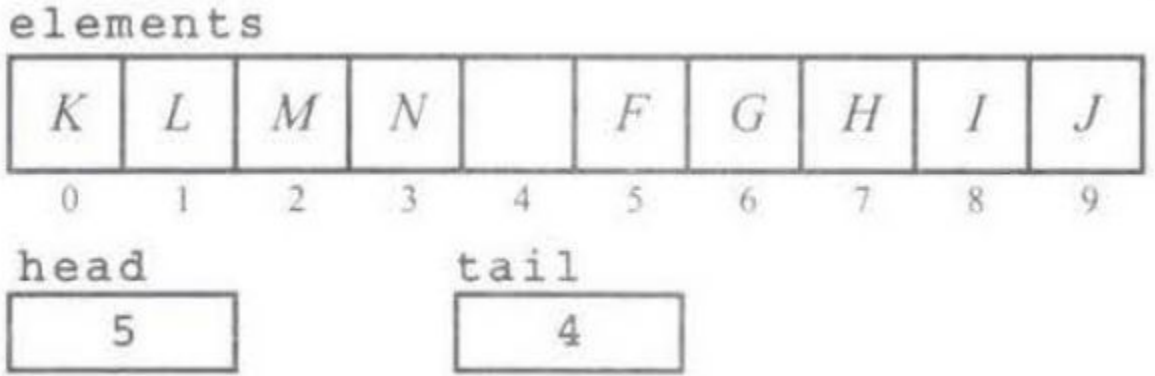


RING BUFFER
(ARRAY CIRCULAR)



10.2.2 queueTAD_array.c (implementação com array)

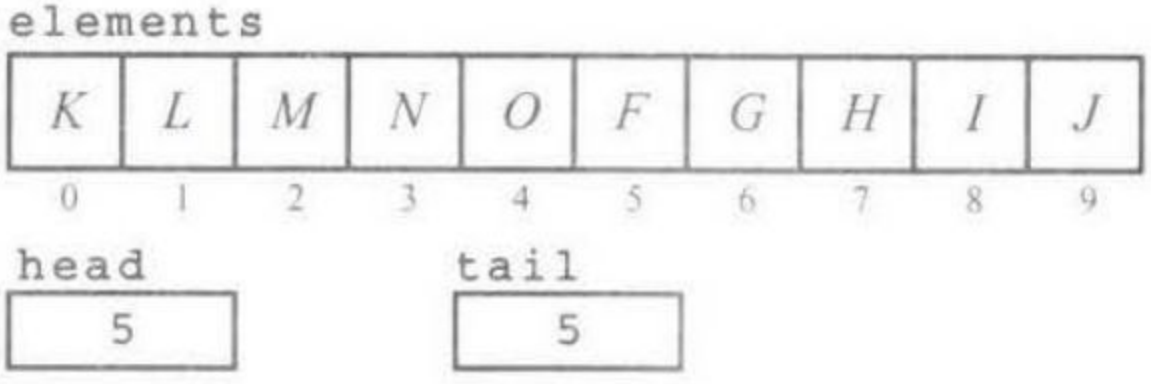
- A operação da fila continua normalmente até, por exemplo, só restar 1 vaga:



A posição apontada por “fim” continua apontando para o próximo lugar livre na fila, e a posição apontado por “início” continua marcando o próximo elemento a sair da fila (a frente da fila)

10.2.2 queueTAD_array.c (implementação com array)

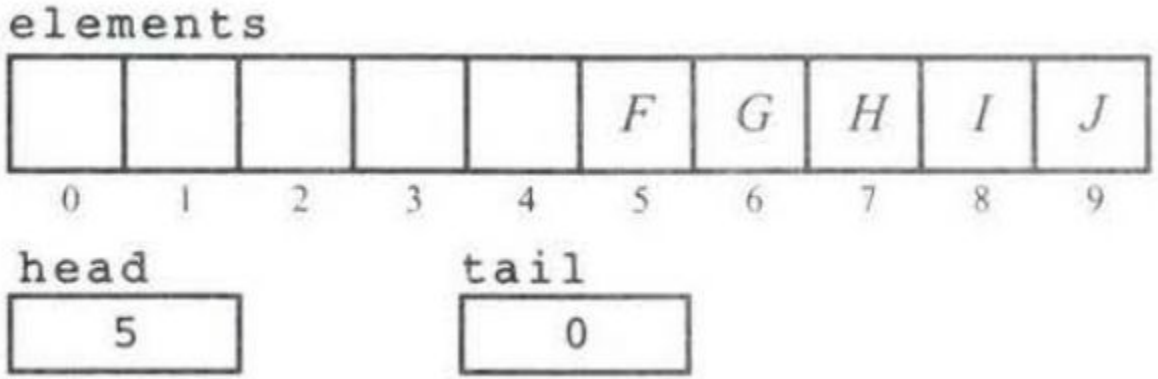
- Se enfileirarmos mais 1 elemento, aí sim a fila ficará cheia:



Note que, caso a fila esteja cheia, “início” = “fim”. Isso pode ser confundido com uma fila vazia, então tome cuidado com essa interpretação! Se os índices forem iguais, a fila pode estar vazia OU cheia!

10.2.2 queueTAD_array.c (implementação com array)

- Para implementar o comportamento circular, usamos **aritmética modular**:




```
fim = (fim + 1) % tam
    = (9 + 1) % 10
    = 10 % 10
    = 0
```

```
inicio = (inicio + 1) % tam
       = (4 + 1) % 10
       = 5 % 10
       = 5
```

10.2.2 queueTAD_array.c (implementação com array)

```
#define TAMMAXQUEUE 100

struct queueTCD
{
    elementoT elementos[TAMMAXQUEUE];
    size_t nelem;
    size_t inicio;
    size_t fim;
}
```

posição do próximo elemento a SAIR da fila (começo da fila)

posição do próximo elemento a ENTRAR na fila (final da fila, o próximo lugar VAGO)

10.2.2 queueTAD_array.c (implementação com array)

```
queueTAD
criar_queue (void)
{
    queueTAD Q = calloc(1, sizeof(struct queueTCD));
    if (Q == NULL)
        return NULL;

    Q->inicio = Q->fim = Q->nelem = 0;
    return Q;
}
```

10.2.2 queueTAD_array.c (implementação com array)

```
 99 queue_status
100 remover_queue (queueTAD *queue)
101 {
102     if (queue == NULL)
103         return QUEUE_ERRO_ARGUMENTO;
104     else if (*queue == NULL)
105         return QUEUE_ERRO_QUEUE;
106
107     free(*queue);
108     *queue = NULL;
109
110     return QUEUE_OK;
111 }
```

10.2.2 queueTAD_array.c (implementação com array)

```
121 queue_status
122 enqueue (queueTAD queue, const elementoT elemento)
123 {
124     if (queue == NULL)
125         return QUEUE_ERRO_QUEUE;
126     else if (elemento == NULL)
127         return QUEUE_ERRO_ARGUMENTO;
128     else if (queue->nelem == TAMMAXQUEUE)
129         return QUEUE_ERRO_CHEIA;
130
131     queue->elementos[queue->fim] = elemento;
132     queue->fim = (queue->fim + 1) % TAMMAXQUEUE;
133     queue->nelem += 1;
134     return QUEUE_OK;
135 }
```

10.2.2 queueTAD_array.c (implementação com array)

```
137 /**
138  * Função: DEQUEUE
139  * Uso: status = dequeue(queue, &elemento);
140  * ----------------------------------------
141  * Verifica se a queue é válida e desenvileira o elemento no início da fila. O
142  * elemento é colocado no endereço apontado pelo ponteiro "elemento". Retorna o
143  * queue_status apropriado.
144  */
145
146 queue_status
147 dequeue (queueTAD queue, elementoT *elemento)
148 {
149     if (queue == NULL)
150         return QUEUE_ERRO_QUEUE;
151     else if (elemento == NULL)
152         return QUEUE_ERRO_ARGUMENTO;
153     else if (queue->nelem == 0)
154         return QUEUE_ERRO_VAZIA;
155
156     *elemento = queue->elementos[queue->inicio];
157     queue->inicio = (queue->inicio + 1) % TAMMAXQUEUE;
158     queue->nelem -= 1;
159     return QUEUE_OK;
160 }
```

10.2.2 queueTAD_array.c (implementação com array)

```
171 queue_status
172 vazia (const queueTAD queue, bool *esta_vazia)
173 {
174     if (queue == NULL)
175         return QUEUE_ERRO_QUEUE;
176     else if (esta_vazia == NULL)
177         return QUEUE_ERRO_ARGUMENTO;
178
179     *esta_vazia = queue->nelem == 0;
180     return QUEUE_OK;
181 }
```

10.2.2 queueTAD_array.c (implementação com array)

```
192 queue_status
193 cheia (const queueTAD queue, bool *esta_cheia)
194 {
195     if (queue == NULL)
196         return QUEUE_ERRO_QUEUE;
197     else if (esta_cheia == NULL)
198         return QUEUE_ERRO_ARGUMENTO;
199
200     *esta_cheia = queue->nelem == TAMMAXQUEUE;
201     return QUEUE_OK;
202 }
```

10.2.2 queueTAD_array.c (implementação com array)

```
212 queue_status
213 num_elementos (const queueTAD queue, size_t *nelem)
214 {
215     if (queue == NULL)
216         return QUEUE_ERRO_QUEUE;
217     else if (nelem == NULL)
218         return QUEUE_ERRO_ARGUMENTO;
219
220     *nelem = queue->nelem;
221     return QUEUE_OK;
222 }
```

10.2.2 queueTAD_array.c (implementação com array)

```
233 queue_status
234 info (const queueTAD queue, bool *din, int *tamax)
235 {
236     if (queue == NULL)
237         return QUEUE_ERRO_QUEUE;
238     else if (din == NULL)
239         return QUEUE_ERRO_ARGUMENTO;
240     else if (tamax == NULL)
241         return QUEUE_ERRO_ARGUMENTO;
242
243     *din = false;
244     *tamax = TAMMAXQUEUE;
245     return QUEUE_OK;
246 }
```

10.2.2 queueTAD_array.c (implementação com array)

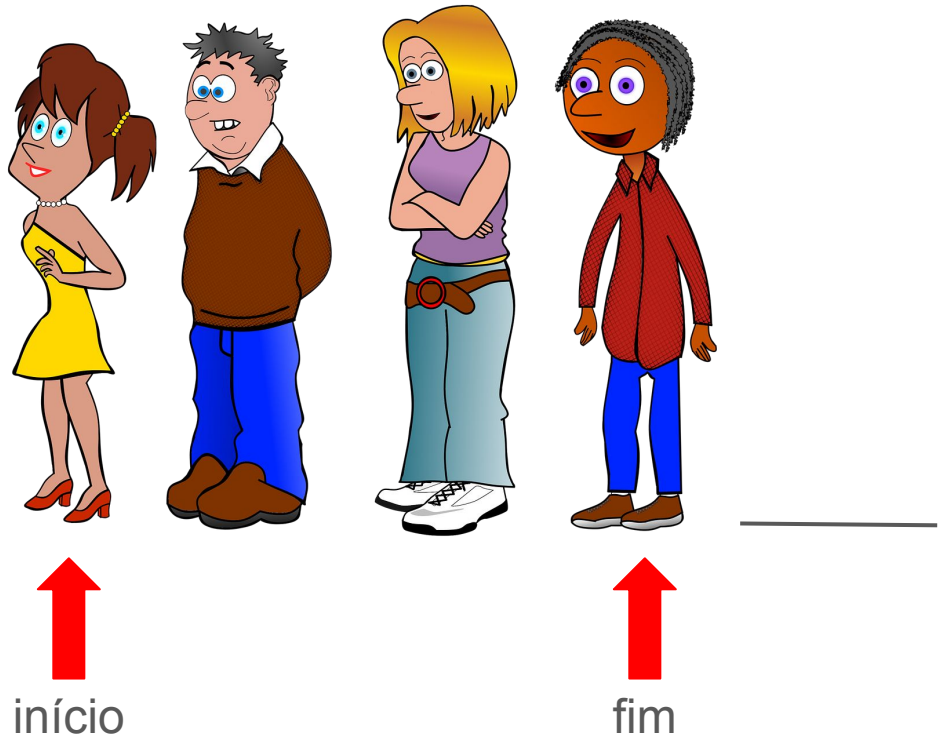
```
259 #ifdef debug
260 queue_status
261 ver_elemento (const queueTAD queue, const size_t posicao, elementoT *elemento)
262 {
263     if (queue == NULL)
264         return QUEUE_ERRO_QUEUE;
265     else if (elemento == NULL)
266         return QUEUE_ERRO_ARGUMENTO;
267     else if (posicao >= queue->nelem)
268         return QUEUE_ERRO_POSICAO;
269
270     *elemento = queue->elementos[(queue->inicio + posicao) % TAMMAXQUEUE];
271     return QUEUE_OK;
272 }
273 #endif
```

10.2.2 queueTAD_array.c (implementação com array)

- Um programa cliente para a interface queueTAD.h está disponível no arquivo de códigos para esta aula.

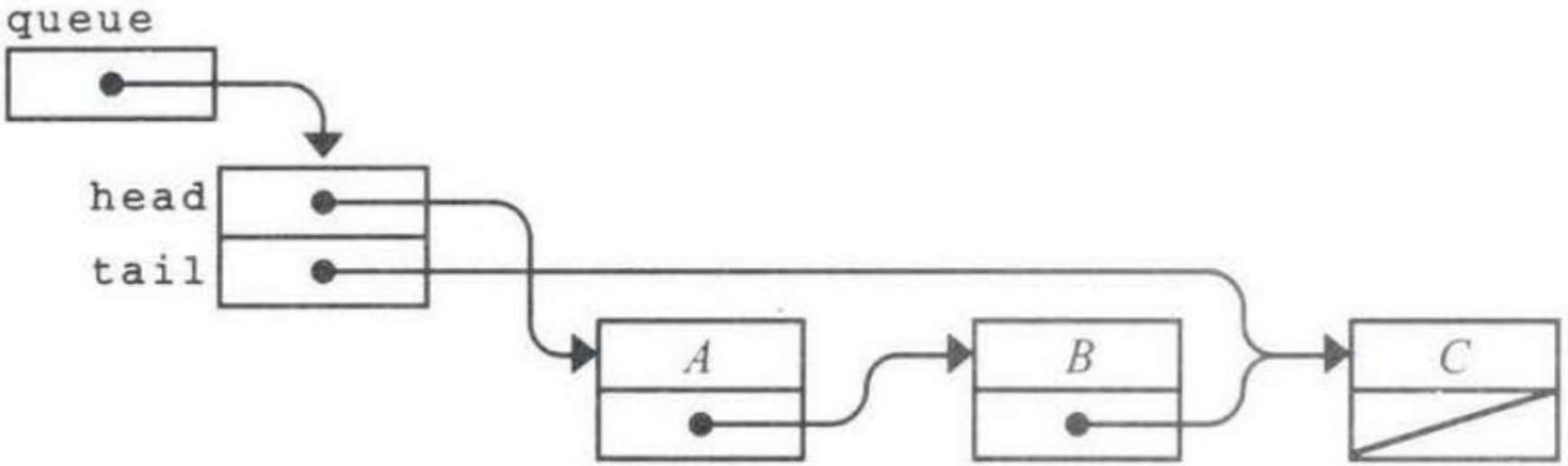
- Estude os códigos da interface (queueTAD.h), da implementação (queueTAD_array.c) e do cliente (queueTAD_cliente.c) para entender como tudo funciona!

10.2.3 queueTAD_lse.c (implementação com LSE)



10.2.3 queueTAD_lse.c (implementação com LSE)

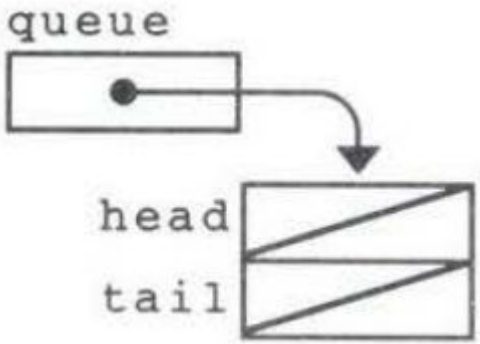
- Uma fila também pode ser implementada com uma LSE:



O ponteiro “início” sempre apontará para o início da fila, e o ponteiro “fim” sempre apontará para o final da fila, que **não tem tamanho limitado**.

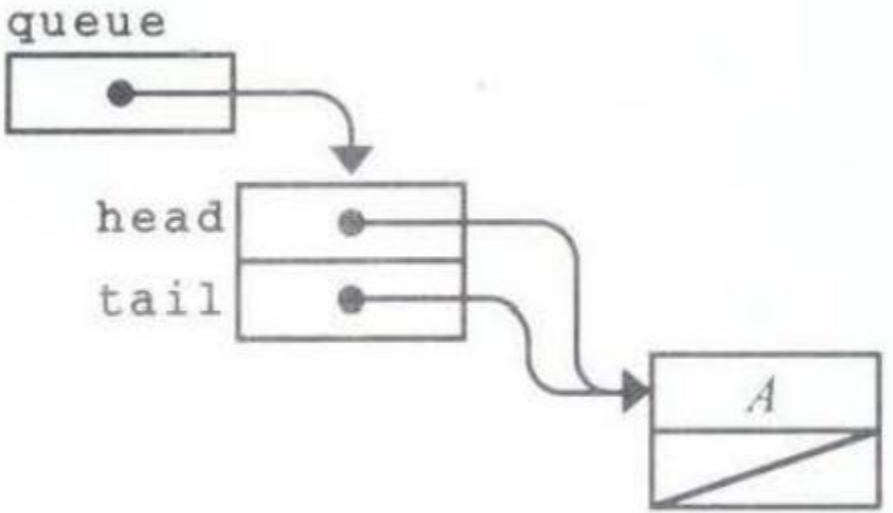
10.2.3 queueTAD_lse.c (implementação com LSE)

- Uma fila vazia é representada por ponteiros NULL:



10.2.3 queueTAD_lse.c (implementação com LSE)

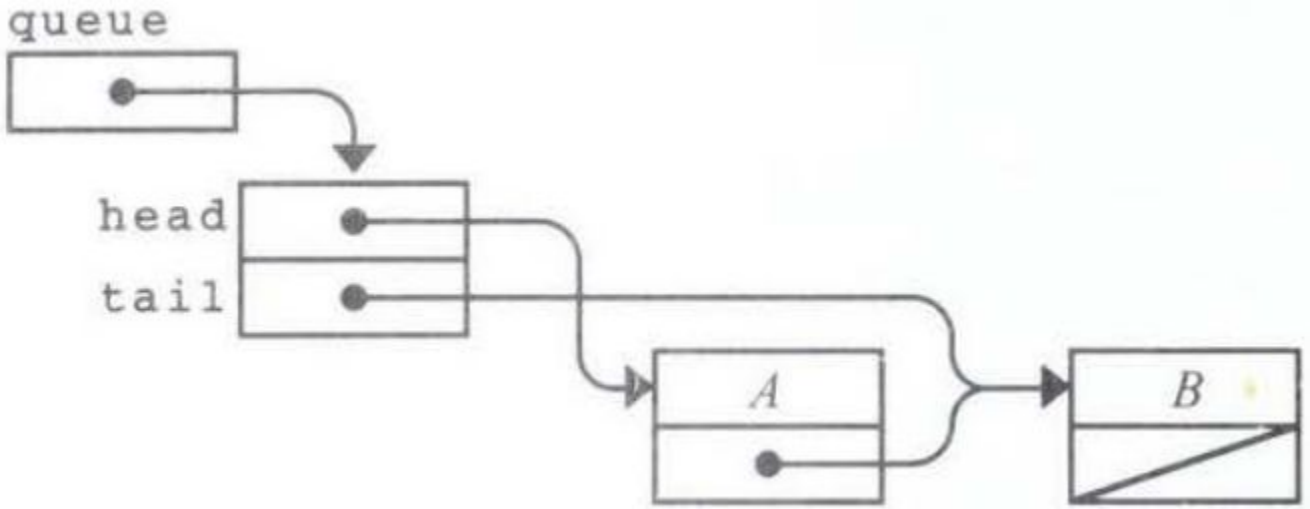
- Ao enfileirar elementos:



Se a fila está vazia, enqueue deve ajustar os 2 ponteiros (início e fim) para fazer com que ambos apontem para a célula inserida.

10.2.3 queueTAD_lse.c (implementação com LSE)

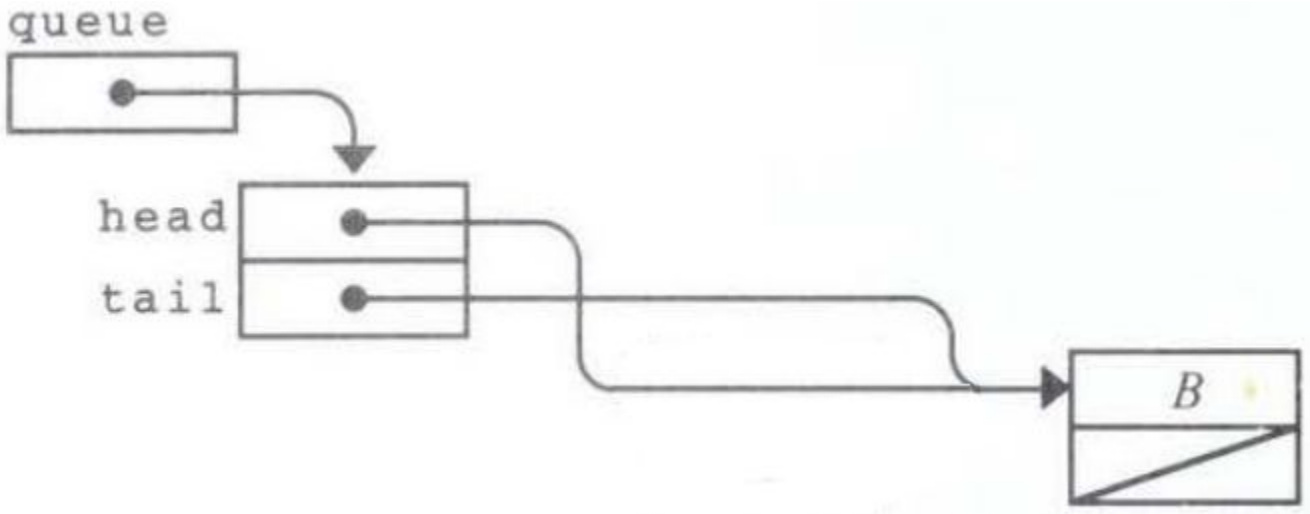
- Ao enfileirar elementos:



Se a fila está não é vazia, enqueue deve ajustar somente o ponteiro “fim” para fazer com que ele aponte para a última célula.

10.2.3 queueTAD_lse.c (implementação com LSE)

- Ao desenfileirar elementos:



Apontar o ponteiro “início” para a próxima célula e remover a célula do início da fila (no exemplo estamos apontando para a última célula pois a fila só tinha 2 elementos). Se a fila esvaziar, apontar ambos para NULL.

10.2.3 queueTAD_lse.c (implementação com LSE)

```
/**
 * Tipo: struct celulaTCD
 * ----------------------
 * Define uma célula (nó) da lista encadeada simples (LSE) que implementará a
 * fila. O tipo concreto é o "celulaTCD"; também é criado um tipo "abstrato" com
 * o nome de "celulaTAD" (na verdade não é um tipo abstrato real, pois a
 * implementação concreta está visível, mas isso simplificará a implementação).
 */

struct celulaTCD
{
    elementoT elemento;
    struct celulaTCD *proximo;
};

typedef struct celulaTCD *celulaTAD;
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
53 /**
54  * Tipo: struct queueTCD
55  * ---------------------
56  * Este tipo define a representação concreta da fila. Esta implementação utiliza
57  * uma lista simplesmente encadeada (LSE) para armazenar so dados, e contém
58  * apenas os ponteiros "início" e "fim" da lista. Como a implementaçao é através
59  * de uma LSE, a lista é considerada dinâmica e não tem tamanho máximo definido.
60  * O número de elementos atualmente na fila também é armazenado para facilitar
61  * a consulta dessa informação. Nesta implementação:
62  *
63  *     a) O próximo elemento a ser enfileirado será colocado após a última
64  *        célula da lista, apontada pelo ponteiro "fim"; e
65  *     b) O próximo elemento a ser desenfileirado será a primeira célula da
66  *        lista, apontada pelo ponteiro "inicio".
67  */
68
69 struct queueTCD
70 {
71     celulaTAD inicio;
72     celulaTAD fim;
73     size_t nelem;
74 };
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
76 /**
77  * Tipo: celula_status
78  * -------------------
79  * Define uma enumeração com os possíveis status de retorno das funções privadas
80  * desta implementação, para o tratamento das celulaTAD. Inclui o status de
81  * sucesso e os diversos status de erro que podem ser retornados. Os seguintes
82  * membros estão definidos:
83  *
84  *     CELULA_OK                : operação realizada com sucesso
85  *     CELULA_ERRO_CELULA       : célula inválida
86  *     CELULA_ERRO_ALOCACAO     : erro na alocação/liberação de memória
87  *     CELULA_ERRO_ARGUMENTO    : argumento inválido
88  */
89
90 typedef enum
91 {
92     CELULA_OK,
93     CELULA_ERRO_CELULA,
94     CELULA_ERRO_ALOCACAO,
95     CELULA_ERRO_ARGUMENTO
96 } celula_status;
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
 98 /*** Declarações de Suprogramas Privados ***/
 99
100 static celulaTAD criar_celula (void);
101 static celula_status remover_celula (celulaTAD *celula);
102
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
105 /**
106  * Função: CRIAR_QUEUE
107  * Uso: queue = criar_queue( );
108  * ----------------------------
109  * Usa calloc para criar a fila e ajusta os ponteiros e a contagem de elementos.
110  * Retorna NULL em caso de erro, ou o ponteiro para a fila em caso de sucesso.
111  */
112
113 queueTAD
114 criar_queue (void)
115 {
116     queueTAD Q = calloc(1, sizeof(struct queueTCD));
117     if (Q == NULL)
118         return NULL;
119
120     Q->inicio = Q->fim = NULL;
121     Q->nelem = 0;
122     return Q;
123 }
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
133 queue_status
134 remover_queue (queueTAD *queue)
135 {
136     if (queue == NULL)
137         return QUEUE_ERRO_ARGUMENTO;
138     else if (*queue == NULL)
139         return QUEUE_ERRO_QUEUE;
140
141     celulaTAD atual, proximo;
142     celula_status status;
143
144     atual = (*queue)->inicio;
145     while (atual != NULL)
146     {
147         proximo = atual->proximo;
148         status = remover_celula(&atual);
149         if (status != CELULA_OK)
150             return QUEUE_ERRO_ALOCACAO;
151         atual = proximo;
152     }
153
154     free(*queue);
155     *queue = NULL;
156
157     return QUEUE_OK;
158 }
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
168 queue_status
169 enqueue (queueTAD queue, const elementoT elemento)
170 {
171     if (queue == NULL)
172         return QUEUE_ERRO_QUEUE;
173     else if (elemento == NULL)
174         return QUEUE_ERRO_ARGUMENTO;
175
176     celulaTAD nova = criar_celula();
177     if (nova == NULL)
178         return QUEUE_ERRO_ALOCACAO;
179
180     nova->elemento = elemento;
181     nova->proximo = NULL;
182
183     if (queue->inicio == NULL)
184     {
185         queue->inicio = nova;
186     }
187     else
188     {
189         queue->fim->proximo = nova;
190     }
191     queue->fim = nova;
192     queue->nelem += 1;
193
194     return QUEUE_OK;
195 }
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
206 queue_status
207 dequeue (queueTAD queue, elementoT *elemento)
208 {
209     if (queue == NULL)
210         return QUEUE_ERRO_QUEUE;
211     else if (elemento == NULL)
212         return QUEUE_ERRO_ARGUMENTO;
213     else if (queue->nelem == 0)
214         return QUEUE_ERRO_VAZIA;
215 
216     *elemento = queue->inicio->elemento;
217 
218     celulaTAD temp = queue->inicio;
219     celula_status status;
220 
221     queue->inicio = temp->proximo;
222 
223     status = remover_celula(&temp);
224     if (status != CELULA_OK)
225         return QUEUE_ERRO_ALOCACAO;
226 
227     if (queue->inicio == NULL)
228         queue->fim = NULL;
229 
230     queue->nelem -= 1;
231 
232     return QUEUE_OK;
233 }
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
235 /**
236  * Função: VAZIA
237  * Uso: if (vazia(queue, &esta_vazia) == QUEUE_OK && esta_vazia == true) . . .
238  * ------------------------------------------------------------------------
239  * Recebe uma "queue" e um PONTEIRO para um booleano "esta_vazia", e retorna
240  * valores que nos permitem identificar se a fila está vazia ou não (ou, se
241  * ocorrer algum erro, permitem identificar esse erro).
242  */
243
244 queue_status
245 vazia (const queueTAD queue, bool *esta_vazia)
246 {
247     if (queue == NULL)
248         return QUEUE_ERRO_QUEUE;
249     else if (esta_vazia == NULL)
250         return QUEUE_ERRO_ARGUMENTO;
251
252     *esta_vazia = queue->nelem == 0;
253     return QUEUE_OK;
254 }
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
256 /**
257  * Função: CHEIA
258  * Uso: if (cheia(queue, &esta_cheia) == QUEUE_OK && esta_cheia == true) . . .
259  * ------------------------------------------------------------------------
260  * Recebe uma "queue" e um PONTEIRO para um booleano "esta_vazia", e retorna
261  * valores que nos permitem identificar se a fila está vazia ou não (ou, se
262  * ocorrer algum erro, permitem identificar esse erro). Como a implementação é
263  * com uma LSE, a fila nunca estará cheia, ou seja, terá tamanho ilimitado.
264  */
265
266 queue_status
267 cheia (const queueTAD queue, bool *esta_cheia)
268 {
269     if (queue == NULL)
270         return QUEUE_ERRO_QUEUE;
271     else if (esta_cheia == NULL)
272         return QUEUE_ERRO_ARGUMENTO;
273
274     *esta_cheia = false;
275     return QUEUE_OK;
276 }
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
278 /**
279  * Função: NUM_ELEMENTOS
280  * Uso: status = num_elementos(queue, &nelem);
281  * ----------------------------------------
282  * Recebe uma "queue" e armazena no local apontado pelo ponteiro "nelem" o
283  * tamanho efetivo da fila ou seja, a quantidade atual de elementos.
284  */
285
286 queue_status
287 num_elementos (const queueTAD queue, size_t *nelem)
288 {
289     if (queue == NULL)
290         return QUEUE_ERRO_QUEUE;
291     else if (nelem == NULL)
292         return QUEUE_ERRO_ARGUMENTO;
293
294     *nelem = queue->nelem;
295     return QUEUE_OK;
296 }
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
298 /**
299  * Função: INFO
300  * Uso: status = info(queue, &din, &tamax);
301  * ----------------------------------------;
302  * Esta função não faz parte dos comportamentos normais esperados para uma fila
303  * mas é definida nesta interface para que o cliente possa obter diversas
304  * informações sobre a fila e sua implementação interna.
305  */
306
307 queue_status
308 info (const queueTAD queue, bool *din, int *tamax)
309 {
310     if (queue == NULL)
311         return QUEUE_ERRO_QUEUE;
312     else if (din == NULL)
313         return QUEUE_ERRO_ARGUMENTO;
314     else if (tamax == NULL)
315         return QUEUE_ERRO_ARGUMENTO;
316
317     *din = true;
318     *tamax = -1;
319     return QUEUE_OK;
320 }
321
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
322 /**
323  * Função: VER_ELEMENTO
324  * Uso: status = ver_elemento(queue, posicao, &elemento);
325  * ------------------------------------------------------
326  * Retorna o elemento armazenado em "posicao", sem desenfileirar o elemento.
327  * Como a posição de início não é fixa, a posição retornada depende de onde
328  * está o começo da fila, ou seja: o cliente informará sempre uma posição
329  * absoluta, mas a função utilizará a posição relativa a partir do índie de
330  * início da fila.
331  */
332
333 #ifdef debug
334 queue_status
335 ver_elemento (const queueTAD queue, const size_t posicao, elementoT *elemento)
336 {
337     if (queue == NULL)
338         return QUEUE_ERRO_QUEUE;
339     else if (elemento == NULL)
340         return QUEUE_ERRO_ARGUMENTO;
341     else if (posicao >= queue->nelem)
342         return QUEUE_ERRO_POSICAO;
343
344     celulaTAD temp = queue->inicio;
345     for (size_t i = 0; i < posicao; i++)
346         temp = temp->proximo;
347     *elemento = temp->elemento;
348
349     return QUEUE_OK;
350 }
351 #endif
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
355 /**
356  * Função: CRIAR_CELULA
357  * Uso: celula = criar_celula( );
358  * ------------------------------
359  * Aloca a memória e cria uma nova célula para a LSE. Retorna o ponteiro para a
360  * célula alocada, ou NULL em caso de erro.
361  */
362
363 static celulaTAD
364 criar_celula (void)
365 {
366     celulaTAD C = calloc(1, sizeof(struct celulaTCD));
367     if (C == NULL)
368         return NULL;
369
370     C->proximo = NULL;
371     return C;
372 }
```

10.2.3 queueTAD_lse.c (implementação com LSE)

```
374 /**
375  * Função: REMOVER_CELULA
376  * Uso: status = remover_celula(&celula);
377  * ---------------------------------------
378  * Recebe um ponteiro para uma celulaTAD e faz a liberação de memória dessa
379  * célula, retornando CELULA_OK. Em caso de erro, retorna o celula_status
380  * correspondente.
381  */
382
383 static celula_status
384 remover_celula (celulaTAD *celula)
385 {
386     if (celula && *celula)
387     {
388         free(*celula);
389         *celula = NULL;
390         return CELULA_OK;
391     }
392
393     return CELULA_ERRO_ALOCACAO;
394 }
```

10.2.3 queueTAD_lse.c (implementação com LSE)

- Um programa cliente para a interface queueTAD.h está disponível no arquivo de códigos para esta aula.

- Estude os códigos da interface (queueTAD.h), da implementação (queueTAD_lse.c) e do cliente (queueTAD_cliente.c) para entender como tudo funciona!

Em resumo

- TADs usados para representar uma lista ordenada por alguma critério são chamados de TADs lineares
- Pilhas e Filas são TADs lineares e diferenciam-se pelo comportamento de entrada e saída
- Pilhas podem ser implementadas com arrays ou listas encadeadas
- Como regra geral, se uma função de uma interface precisar alterar o valor de qualquer dado associado com um TAD, o dado deve ser acessível dentro do lado da implementação na barreira da interface

Em resumo

- Filas podem ser implementadas com arrays (ring buffers, arrays circulares) ou com listas encadeadas.
  - Implementação com array é um pouco mais complexa
  - Aritmética modular facilita implementação com array

  - Implementação com lista encadeada é um pouco mais simples
  - Devemos ter cuidado na alocação/liberação das células das listas, além da alocação/liberação da fila em si