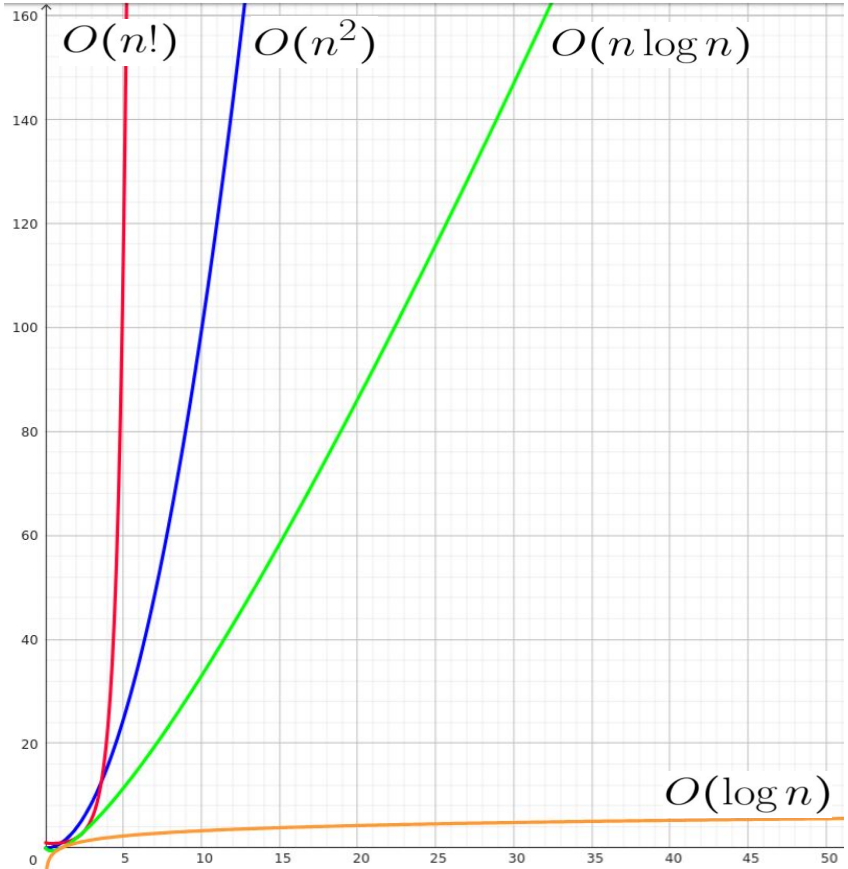
Estrutura de Dados I

Capítulo 9: Eficiência e TADs

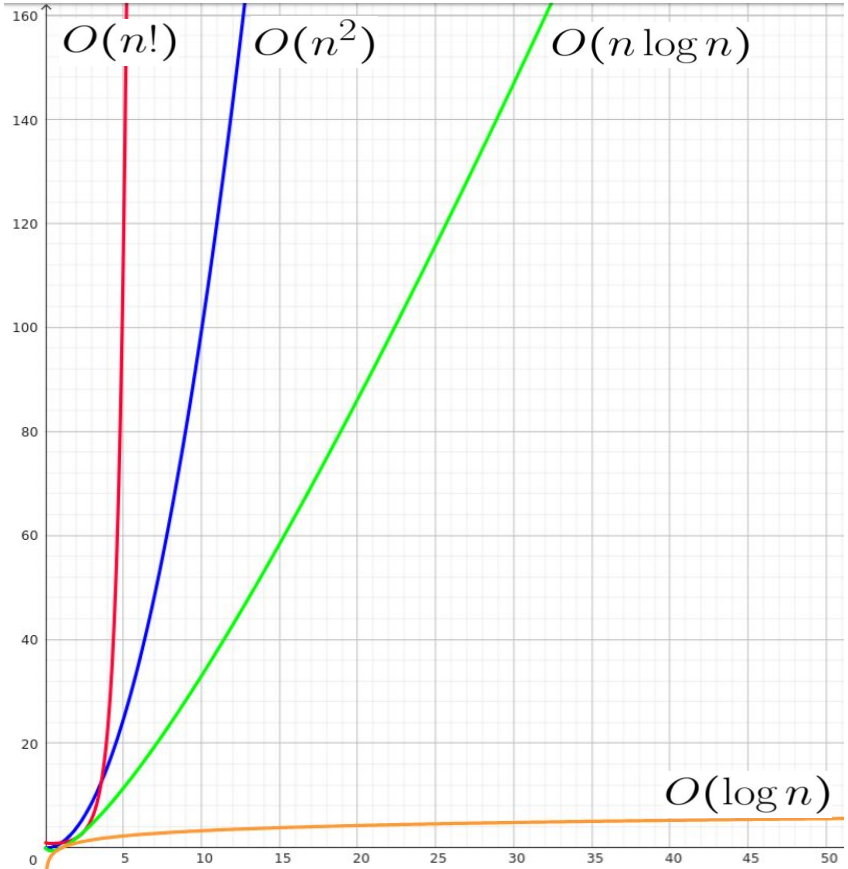
Prof. Abrantes Araújo Silva Filho

Eficiência x Algoritmos



- Até agora nosso entendimento sobre eficiência computacional estava voltada para a eficiência de diversos algoritmos:
    - O tempo de execução pode ser muito diferente dependendo da classe de complexidade do algoritmo.

Eficiência x TADs

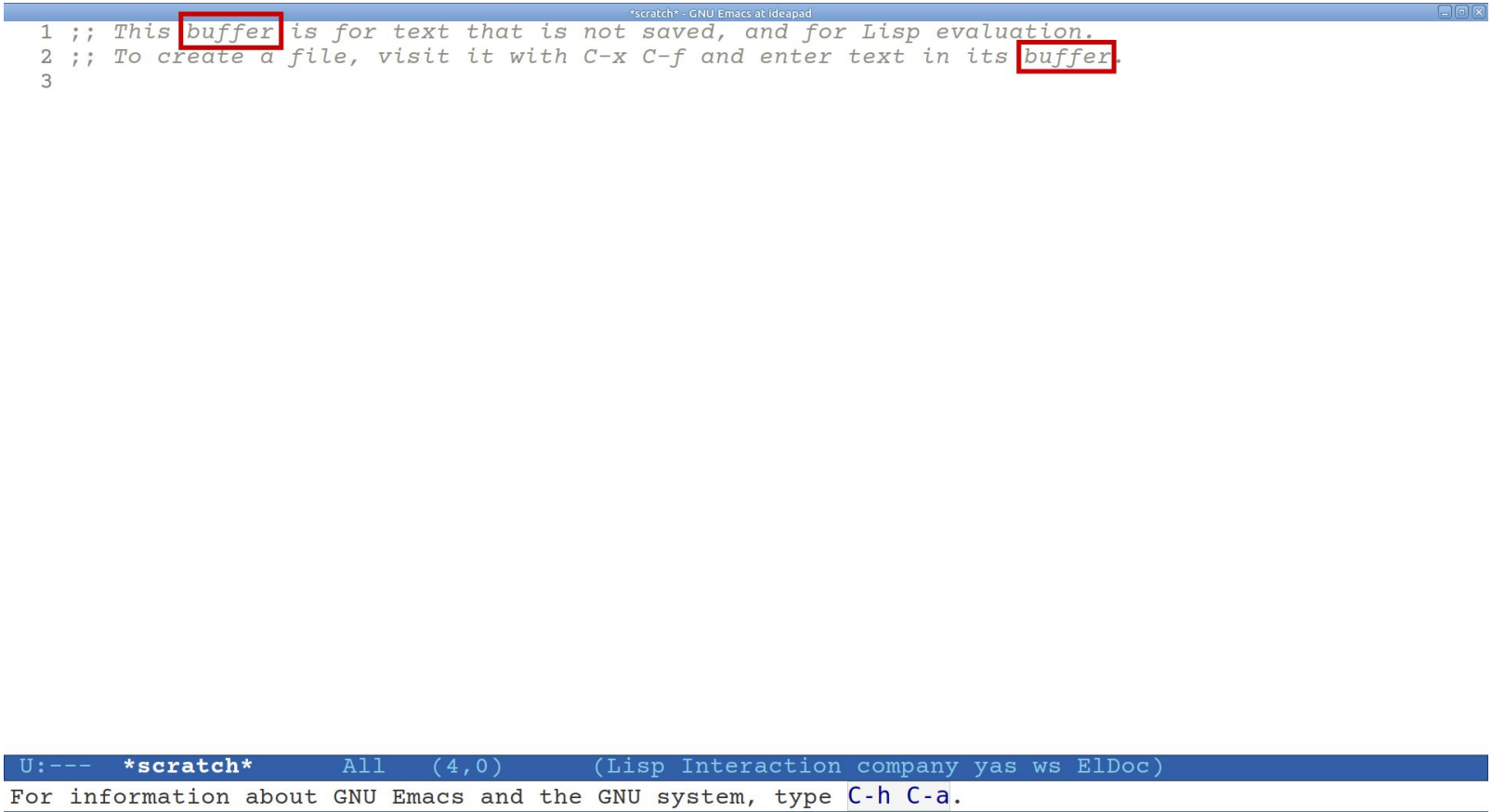


- Será que a eficiência também pode variar de acordo com o TAD escolhido?
    - Sim! Em alguns casos, escolher **uma representação interna melhor pode levar a um grande aumento na eficiência do tempo de execução** de um programa!

Eficiência x TADs

- Para verificar como a escolha da representação interna de um TAD pode influenciar a eficiência, vamos estudar um **editor de buffer**, que utiliza um TAD para representar um buffer e que pode ser **implementado internamente através de diversas estruturas de dados**:
  - array
  - pilha
  - lista encadeada
    - simples
    - dupla

Editor de buffer: Emacs (e outros!)



Editor de buffer

- Um **editor** de texto permite que você faça alterações em um arquivo de texto.
- A seqüência de caracteres que o editor mantém é chamada de **buffer**.
- O editor permite que você faça **alterações no buffer**, através de operações tais como as seguinte:
  - Mover o cursor até o ponto da edição
  - Inserir novo texto na posição do cursor
  - Remover caracteres com o backspace ou delete
- Mostram o conteúdo do buffer em tempo real com as edições: WYSIWYG

Editor de buffer

- Vamos criar um editor de buffer simples:

Command	Operation
`F`	Moves the editing cursor forward one character position
`B`	Moves the editing cursor backward one character position
`J`	Jumps to the beginning of the buffer (before the first character)
`E`	Moves the cursor to the end of the buffer (after the last character)
`Ixxx`	Inserts the characters *xxx* at the current cursor position
`D`	Deletes the character just after the current cursor position

Editor de buffer

```
*Iaxc↵
 a x c
       ^
```
This command inserts the characters 'a', 'x', and 'c', leaving the cursor at the end of the buffer.

```
*J↵
 a x c
^
```
This command moves the cursor to the beginning of the buffer.

```
*F↵
 a x c
  ^
```
This command moves the cursor forward one character.

```
*D↵
 a c
  ^
```
This command deletes the character after the cursor.

```
*Ib↵
 a b c
    ^
*
```
This command inserts the character 'b'.

TAD buffer

- Claramente um editor de buffer precisa de uma estrutura de dados para **armazenar o estado do buffer, a seqüência de caracteres que forma o buffer atual**. Qual seria a melhor estrutura de dados, dentre as várias possíveis?

- Ainda não sabemos qual seria a melhor! Mas já sabemos que, **independentemente da representação interna do buffer, o comportamento deve ser o mesmo**. Isso sugere claramente que o buffer deve ser um TAD, separando o comportamento da representação interna.

- Nunca seja apressado para definir a melhor estrutura de dados!

TAD buffer: a interface buffer.h

```
22 /*** Inicia Boilerplate da Interface ***/
23
24 #ifndef _BUFFER_H
25 #define _BUFFER_H
26
27 /*** Includes ***/
28
29 #include "genlib.h"
30
31 /*** Tipos de Dados ***/
32
33 /**
34  * TIPO: bufferTAD
35  * ---------------
36  * Este tipo abstrato de dado é utilizado para representar um editor de buffer.
37  */
38
39 typedef struct bufferTCD *bufferTAD;
40
```

TAD buffer: a interface buffer.h

```
43 /**
44  * FUNÇÃO: criar_buffer
45  * Uso: buffer = criar_buffer( );
46  * ------------------------------
47  * Esta função aloca memória de modo dinâmico, em quantidade suficiente para a
48  * representação interna do bufferTAD, e inicializa o buffer para representar
49  * um buffer vazio.
50  */
51
52 bufferTAD criar_buffer (void);
53
54 /**
55  * PROCEDIMENTO: liberar_buffer
56  * Uso: liberar_buffer(buffer);
57  * ----------------------------
58  * Este procedimento libera o espaço de armazenamento alocado para o buffer. O
59  * argumento deve ser um PONTEIRO para o buffer (um ponteiro para ponteiro para
60  * struct bufferTCD).
61  */
62
63 void liberar_buffer (bufferTAD *buffer);
64
```

TAD buffer: a interface buffer.h

```
65 /**
66  * PROCEDIMENTOS: mover_cursor_para_frente
67  *                mover_cursor_para_tras
68  * Uso: mover_cursor_para_frente(buffer);
69  *      mover_cursor_para_tras(buffer);
70  * ---------------------------------------
71  * Estes procedimentos movem o cursor para frente e para trás, no buffer, um
72  * caractere por vez. Se "mover_cursor_para_frete" for chamada no final do
73  * buffer, ou se "mover_cursor_para_tras" for chamada no início do buffer, os
74  * procedimentos não têm efeito nenhum.
75  */
76
77 void mover_cursor_para_frente (bufferTAD buffer);
78 void mover_cursor_para_tras (bufferTAD buffer);
79
80 /**
81  * PROCEDIMENTOS: mover_cursor_para_final
82  *                mover_cursor_para_inicio
83  * Uso: mover_cursor_para_final(buffer);
84  *      mover_cursor_para_inicio(buffer);
85  * ---------------------------------------
86  * Estes procedimentos movem o cursor apra o final ou para o início do buffer,
87  * respectivamente.
88  */
89
90 void mover_cursor_para_final (bufferTAD buffer);
91 void mover_cursor_para_inicio (bufferTAD buffer);
```

TAD buffer: a interface buffer.h

```
 93 /**
 94  * PROCEDIMENTO: inserir_caractere
 95  * Uso: inserir_caractere(buffer, c);
 96  * ----------------------------------
 97  * Insere o caractere "c" no buffer "buffer", na posição atual do cursor. Após
 98  * a inserção o cursos é posicionado após o caractere inserido, para permitir
 99  * inserções consecutivas.
100  */
101
102 void inserir_caractere (bufferTAD buffer, char c);
103
104 /**
105  * PROCEDIMENTO: apagar_caractere
106  * Uso: apagar_caractere(buffer);
107  * ------------------------------
108  * Apaga o caractere imediatamente posterior ao cursor. Se o cursor já está no
109  * final do buffer, não causa nenhum efeito.
110  */
111
112 void apagar_caractere (bufferTAD buffer);
113
114 /**
115  * PROCEDIMENTO: exibir_buffer
116  * Uso: exibir_buffer(buffer);
117  * ---------------------------
118  * Exibe o conteúdo atual do buffer no terminal.
119  */
120
121 void exibir_buffer (bufferTAD buffer);
```

TAD buffer: o programa cliente meu_editor.c

- Agora que temos a interface,

já é possível criar o programa cliente?

TAD buffer: o programa cliente meu_editor.c

```
23 /*** Includes: ***/
24
25 #include "buffer.h"
26 #include <ctype.h>
27 #include "genlib.h"
28 #include <stdio.h>
29 #include <stdlib.h>
30 #include "simpio.h"
31
32 /*** Declarações de Subprogramas: ***/
33
34 static void executar_comando (bufferTAD buffer, string linha);
35 static void ajuda (void);
36
```

TAD buffer: o programa cliente meu_editor.c

```
37 /*** Função Main: ***/
38
39 int main (void)
40 {
41     bufferTAD buffer = criar_buffer();
42
43     while (TRUE)
44     {
45         printf("*");
46         executar_comando(buffer, GetLine());
47         exibir_buffer(buffer);
48     }
49
50     liberar_buffer(&buffer);
51 }
```

TAD buffer: o programa cliente meu_editor.c

```
55 /**
56  * Procedimento: executar_comando
57  * Uso: executar_comando(buffer, linha);
58  * ------------------------------------
59  * Faz o parser do comando informado pelo usuário, na linha, e executa esse
60  * comando no buffer.
61  */
62
63 static void executar_comando (bufferTAD buffer, string linha)
64 {
65     switch (toupper(linha[0]))
66     {
67     case 'I':
68         for (int i = 1; linha[i] != '\0'; i++)
69             inserir_caractere(buffer, linha[i]);
70         break;
71     case 'D': apagar_caractere(buffer); break;
72     case 'F': mover_cursor_para_frente(buffer); break;
73     case 'B': mover_cursor_para_tras(buffer); break;
74     case 'J': mover_cursor_para_inicio(buffer); break;
75     case 'E': mover_cursor_para_final(buffer); break;
76     case 'H': ajuda(); break;
77     case 'Q': exit(0); break;
78     default: printf("Comando inválido.\n"); break;
79     }
80 }
```

TAD buffer: o programa cliente meu_editor.c

```
 82 /**
 83  * Procedimento: ajuda
 84  * Uso: ajuda( );
 85  * ------------------;
 86  * Lista os comandos disponíveis no editor.
 87  */
 88
 89 static void ajuda (void)
 90 {
 91     printf("Use os seguintes comandos para editar o buffer:\n");
 92     printf("  I...    Insere o texto informado após a letra \'I\'.\n");
 93     printf("  F       Move o cursor 1 caractere para frente.\n");
 94     printf("  B       Move o cursor 1 caractere para trás.\n");
 95     printf("  J       Move o cursor para o início do buffer.\n");
 96     printf("  E       Move o cursor para o final do buffer.\n");
 97     printf("  D       Apaga o próximo caractere.\n");
 98     printf("  H       Exibe esta ajuda.\n");
 99     printf("  Q       Sai do programa.\n");
100 }
```

TAD buffer: o que falta fazer agora?

- Já temos a interface e o cliente,

o que falta fazer agora?

TAD buffer: o que falta fazer agora?

- Já temos a interface e o cliente,

o que falta fazer agora?

- Escolher uma representação interna para o TAD e fazer a implementação!
  - Array
  - Pilha
  - Lista encadeada (simples e dupla)

TAD buffer: implementação com ARRAY

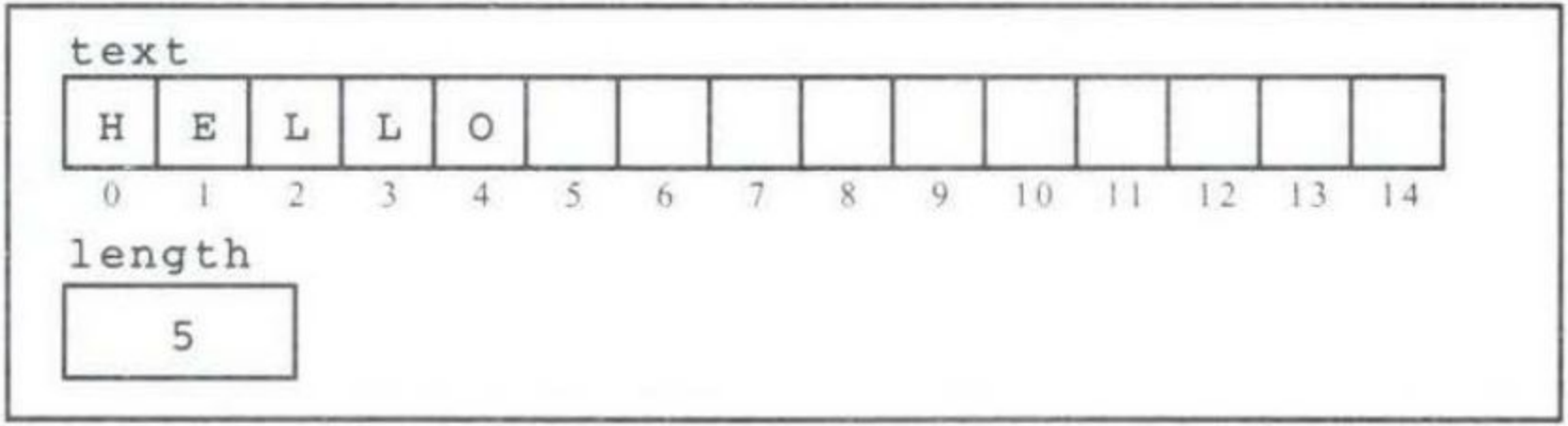
- Parece solução simples, já que strings são arrays de char
- Decisões mais difíceis:
  - Qual o tamanho do array?
  - O caractere ‘\0’ que marca o final da string será armazenado?
  - Como representar a posição atual do cursor?

TAD buffer: implementação com ARRAY

- Parece solução simples, já que strings são arrays de char
- Decisões mais difíceis:
  - Qual o tamanho do array?
    - Vamos limitar para simplificar
  - O caractere ‘\0’ que marca o final da string será armazenado?
    - Não, vamos controlar o tamanho da string por uma variável. Por quê?
  - Como representar a posição atual do cursor?
    - Uma variável

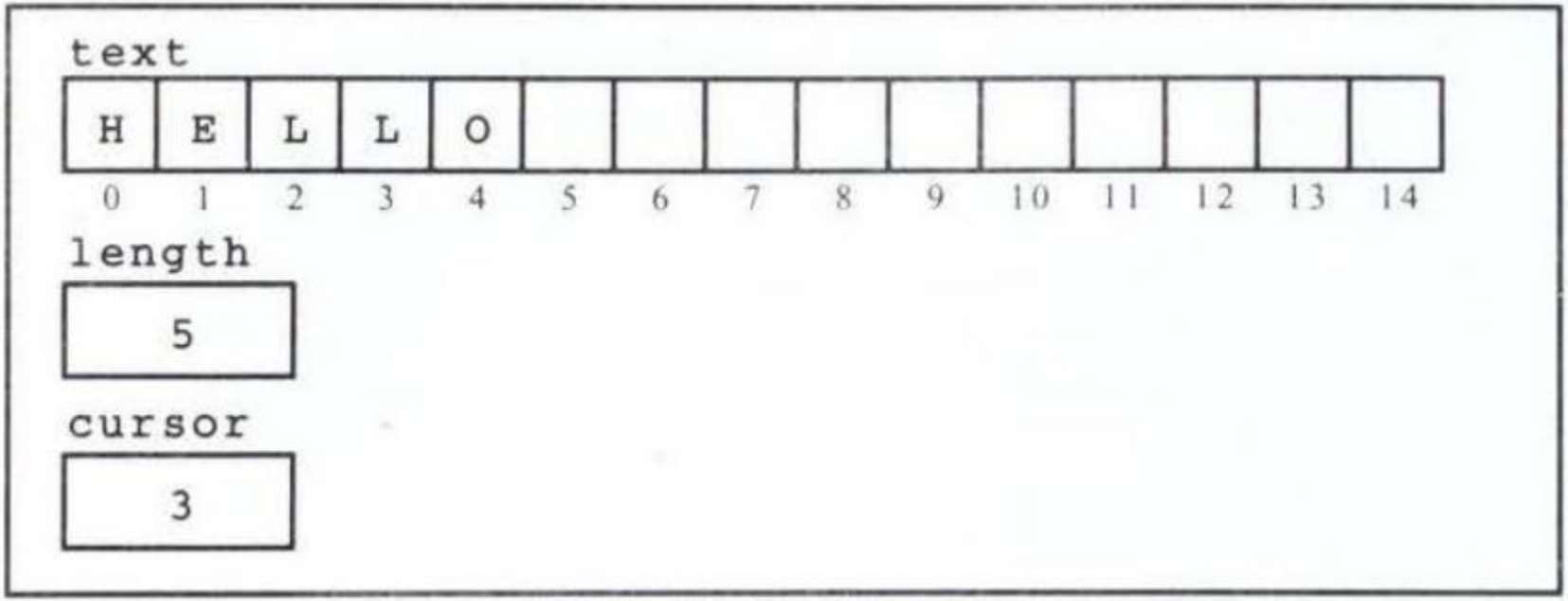
TAD buffer: implementação com ARRAY

- Tamanho fixo
- Não armazenaremos o ‘\0’
  - Por quê?



TAD buffer: implementação com ARRAY

- Cursor indica a posição atual
  - se 0 = início do buffer
  - se igual ao length = final do buffer



TAD buffer: implementação com ARRAY

```
32 /*** Constantes Simbólicas ***/
33
34 #define TAMBUFFER 100
35
36 /*** Tipos de Dados ***/
37
38 /**
39  * Tipo: struct bufferTCD
40  * ----------------------
41  * Nesta representação interna do buffer os caracteres estão armazenados em um
42  * array. Também existirão variáveis para armazenar o tamanho da string do
43  * buffer (o que nos permite ignorar o '\0' como marcador de final) e a posição
44  * do cursor no buffer. A posição do cursor indicará o índice da posição no
45  * array onde o próximo caractere será inserido.
46  *
47  *     texto      array com tamanho TAMBUFFER para os caracteres
48  *     tamanho    quantidade de caracteres no buffer
49  *     cursor     posição atual do cursor de edição
50  */
51
52 struct bufferTCD
53 {
54     char texto[TAMBUFFER];
55     int tamanho;
56     int cursor;
57 };
```

TAD buffer: implementação com ARRAY

```
61 /**
62  * Função: criar_buffer
63  * Uso: buffer = criar_buffer( );
64  * ------------------------------
65  * Cria e retorna um novo bufferTAD. Se a memória não puder ser alocada para o
66  * buffer, retorna NULL.
67  */
68
69 bufferTAD criar_buffer (void)
70 {
71     bufferTAD B = malloc(sizeof(struct bufferTCD));
72     if (B == NULL)
73     {
74         fprintf(stderr, "Erro: impossível alocar buffer.\n");
75         return NULL;
76     }
77
78     B->tamanho = 0;
79     B->cursor = 0;
80
81     return B;
82 }
```

TAD buffer: implementação com ARRAY

```
84 /**
85  * Procedimento: liberar_buffer
86  * Uso: liberar_buffer(buffer);
87  * ----------------------------
88  * Libera a memória alocada para um buffer. Recebe um PONTEIRO para um buffer,
89  * ou seja, um ponteiro para um ponteiro para struct bufferTCD.
90  */
91
92 void liberar_buffer (bufferTAD *buffer)
93 {
94     if (*buffer != NULL)
95     {
96         free(*buffer);
97         *buffer = NULL;
98     }
99 }
```

TAD buffer: implementação com ARRAY

```
110 void mover_cursor_para_frente (bufferTAD buffer)
111 {
112     if (buffer == NULL)
113     {
114         fprintf(stderr, "Erro: movimentação em buffer null.\n");
115         exit(1);
116     }
117
118     if (buffer->cursor < buffer->tamanho)
119         buffer->cursor++;
120 }
121
122 void mover_cursor_para_tras (bufferTAD buffer)
123 {
124     if (buffer == NULL)
125     {
126         fprintf(stderr, "Erro: movimentação em buffer null.\n");
127         exit(1);
128     }
129
130     if (buffer->cursor > 0)
131         buffer->cursor--;
132 }
```

TAD buffer: implementação com ARRAY

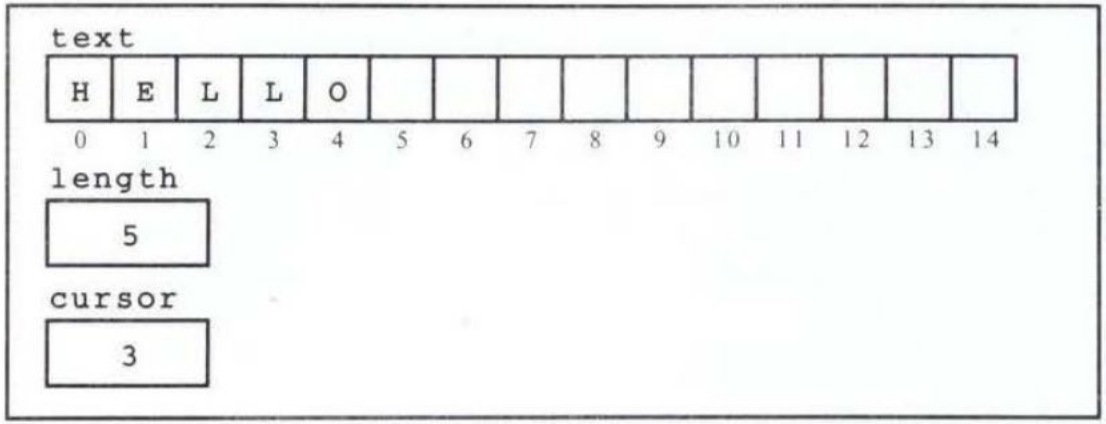
```
143 void mover_cursor_para_final (bufferTAD buffer)
144 {
145     if (buffer == NULL)
146     {
147         fprintf(stderr, "Erro: movimentação em buffer null.\n");
148         exit(1);
149     }
150
151     buffer->cursor = buffer->tamanho;
152 }
153
154 void mover_cursor_para_inicio (bufferTAD buffer)
155 {
156     if (buffer == NULL)
157     {
158         fprintf(stderr, "Erro: movimentação em buffer null.\n");
159         exit(1);
160     }
161
162     buffer->cursor = 0;
163 }
```

TAD buffer: implementação com ARRAY

- Até aqui a implementação foi bem simples, nenhum subprograma causou nenhuma dificuldade.

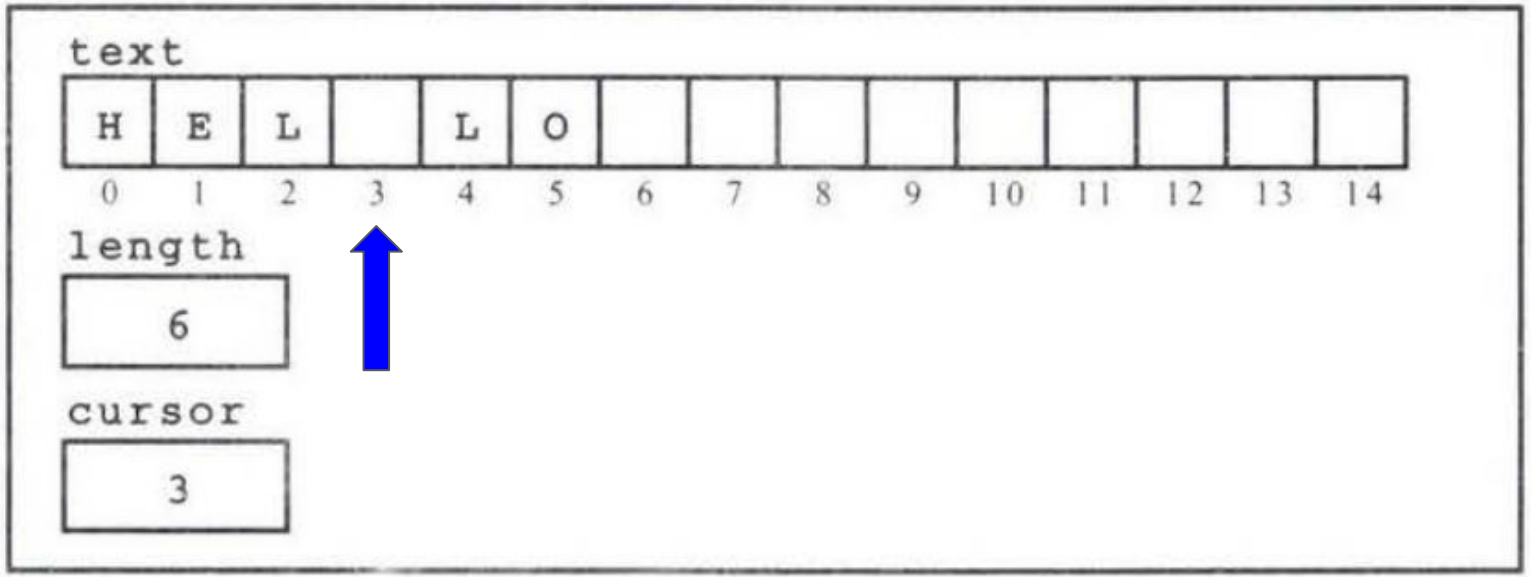
- Mas as operações de inserção e remoção de caracteres precisam de maior cuidado! Considere o seguinte buffer: o que fazer se quisermos inserir um caractere na seguinte posição:

H E L | L O



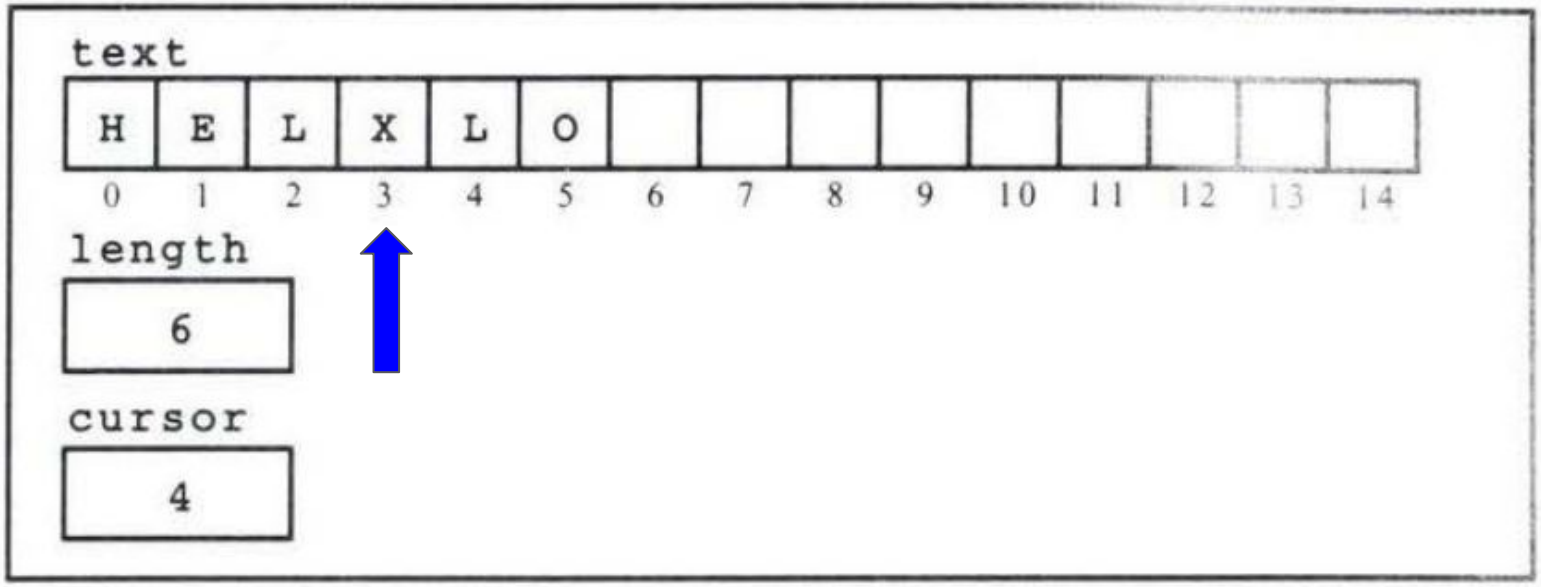
TAD buffer: implementação com ARRAY

- A única solução é abrir um espaço no array, deslocando todos os caracteres após o cursor 1 posição para a direita. E temos que garantir que isso não causará um buffer overflow.



TAD buffer: implementação com ARRAY

- A única solução é abrir um espaço no array, deslocando todos os caracteres após o cursor 1 posição para a direita. E temos que garantir que isso não causará um buffer overflow.



TAD buffer: implementação com ARRAY

```
165 /**
166  * Procedimento: inserir_caractere
167  * Uso: inserir_caractere(buffer, c);
168  * ----------------------------------
169  * Insere o caractere "c" no buffer "buffer", na posição atual do cursor,
170  * deslocando os caracteres após a posição de inserção 1 posição para a
171  * direita. Se ocorrer buffer overflow, ocorrerá um erro.
172  */
173
174 void inserir_caractere (bufferTAD buffer, char c)
175 {
176     if (buffer == NULL)
177     {
178         fprintf(stderr, "Erro: inserção em buffer null.\n");
179         exit(1);
180     }
181     else if (buffer->cursor == TAMBUFFER)
182     {
183         fprintf(stderr, "Erro: buffer overflow.\n");
184         exit(1);
185     }
186
187     for (int i = buffer->tamanho; i > buffer->cursor; i--)
188         buffer->texto[i] = buffer->texto[i - 1];
189
190     buffer->texto[buffer->cursor] = c;
191     buffer->tamanho++;
192     buffer->cursor++;
193 }
```

TAD buffer: implementação com ARRAY

```
195 /**
196  * Procedimento: apagar_caractere
197  * Uso: apagar_caractere(buffer);
198  * ------------------------------
199  * Apaga o caractere imediatamente posterior ao cursor.
200  */
201
202 void apagar_caractere (bufferTAD buffer)
203 {
204     if (buffer == NULL)
205     {
206         fprintf(stderr, "Erro: remoção em buffer null.\n");
207         exit(1);
208     }
209
210     if (buffer->cursor < buffer->tamanho)
211     {
212         for (int i = buffer->cursor + 1; i < buffer->tamanho; i++)
213             buffer->texto[i - 1] = buffer->texto[i];
214         buffer->tamanho--;
215     }
216 }
```

TAD buffer: implementação com ARRAY

```
218 /**
219  * Procedimento: exibir_buffer
220  * Uso: exibir_buffer(buffer);
221  * --------------------------
222  * Exibe o conteúdo atual do buffer no terminal.
223  */
224
225 void exibir_buffer (bufferTAD buffer)
226 {
227     for (int i = 0; i < buffer->tamanho; i++)
228         printf(" %c", buffer->texto[i]);
229
230     printf("\n");
231
232     for (int i = 0; i < buffer->cursor; i++)
233         printf("  ");
234     printf("^\n");
235 }
```

TAD buffer: implementação com ARRAY

```
[abrantesasf@ideapad ~/ed1/cap09]$ ./meu_editor_arraybuff
*Iabrantes araujo silva filho
 a b r a n t e s   a r a u j o   s i l v a   f i l h o
                                                     ^

*b
 a b r a n t e s   a r a u j o   s i l v a   f i l h o
                                                   ^

*b
 a b r a n t e s   a r a u j o   s i l v a   f i l h o
                                                 ^

*b
 a b r a n t e s   a r a u j o   s i l v a   f i l h o
                                                   ^

*b
 a b r a n t e s   a r a u j o   s i l v a   f i l h o
                                                  ^

*b
 a b r a n t e s   a r a u j o   s i l v a   f i l h o
                                                ^

*d
 a b r a n t e s   a r a u j o   s i l v a   i l h o
                                                ^

*IF
 a b r a n t e s   a r a u j o   s i l v a   F i l h o
                                                  ^

*q
```

TAD buffer: eficiência da implementação com ARRAY

- As operações que movem o cursor executam em tempo constante.

- Já as operações de inserção e remoção, como podem deslocar TODOS os caracteres para a direita ou para a esquerda, são proporcionais ao número de caracteres no buffer:

Function	Complexity
MoveCursorForward	$O(1)$
MoveCursorBackward	$O(1)$
MoveCursorToStart	$O(1)$
MoveCursorToEnd	$O(1)$
InsertCharacter	$O(N)$
DeleteCharacter	$O(N)$

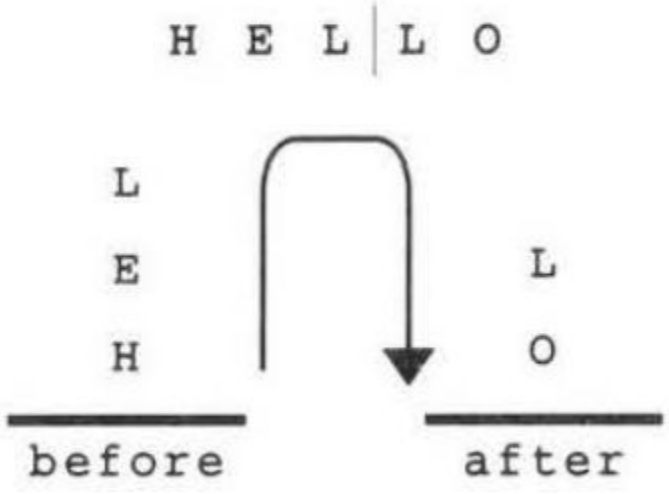
TAD buffer: eficiência da implementação com ARRAY

- Características importantes do bufferTAD com arrays:
  - **É LENTA quando as inserções/remoções são feitas no início do buffer**
  - **É RÁPIDA quando as inserções/remoções são feitas no final do buffer**
- Será que, como é rápida nas alteraçẽos no final, uma **pilha** então seria uma implementação melhor?
  - Ao usar uma pilha, forçaríamos todas as inserções/remoções a serem feitas na extremidade do buffer (topo da pilha), que é uma operação O(1) para uma pilha.
  - A idéia é “meio maluca” mas pode ser implementada com 2 pilhas!

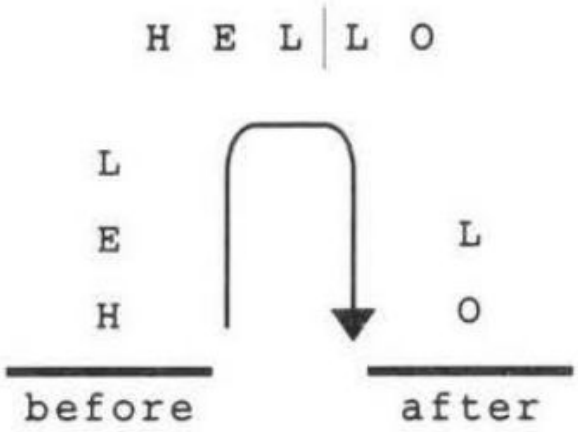
TAD buffer: implementação com PILHA

- A idéia principal aqui é utilizar 2 pilhas ao mesmo tempo:
  - Uma **pilha para caracteres ANTES do cursor**
  - Outra **pilha para caracteres DEPOIS do cursor**

- Como todas as mudanças ocorrem na posição do cursor, isso funciona!



TAD buffer: implementação com PILHA



- Faremos push dos caracteres anteriores ao cursor na pilha “antes”
- Faremos push dos caracteres posteriores ao cursos na pilha “depois”, em ordem reversa
- Para ler o buffer: de baixo para cima na pilha “antes” e de cima para baixo na pilha “depois”
- É confuso, preste atenção!

TAD buffer: implementação com PILHA

```
22 /*** Includes: ***/
23
24 #include "buffer.h"
25 #include "genlib.h"
26 #include <stdio.h>
27 #include <stdlib.h>
28 #include "simpio.h"
29 #include "stackTAD.h"
30 #include "strlib.h"
```

bufferTAD

Pilha com array criada no capítulo 8, modificada para trabalhar com **`char`**.

TAD buffer: implementação com PILHA

```
34 /**
35  * Tipo: bufferTCD
36  * ---------------
37  * Nesta representação de buffer os caracteres são armazenados em uma de duas
38  * pilhas. Os caracteres que estão ANTES do cursor são armazenados na pilha
39  * "antes"; os caracteres que estão DEPOIS do cursor são armazenados na pilha
40  * "depois". O cursor não é representado explicitamente: ele é mantido de
41  * forma implícita, como o limite entre as duas pilhas. Um buffer com as letras
42  * ABCDE, com o cursor entre as letras C e D, seria armazenado como:
43  *
44  *       C
45  *       B         D
46  *       A         E
47  *     -----    ------
48  *     antes    depois
49  */
50
51 struct bufferTCD
52 {
53     stackTAD antes;
54     stackTAD depois;
55 };
```

TAD buffer: implementação com PILHA

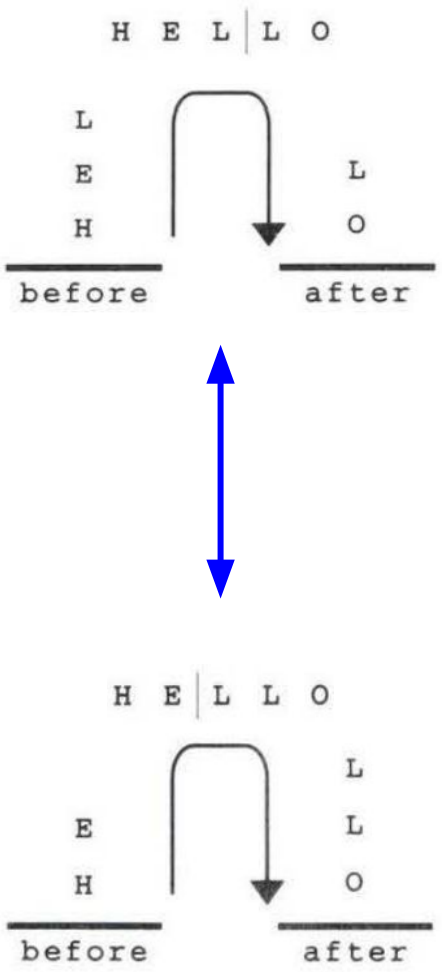
```
59 /**
60  * FUNÇÃO: criar_buffer
61  * Uso: buffer = criar_buffer( );
62  * ------------------------------
63  * Esta função aloca memória de modo dinâmico, em quantidade suficiente para a
64  * representação interna do bufferTAD, e inicializa o buffer para representar
65  * um buffer vazio.
66  */
67
68 bufferTAD criar_buffer (void)
69 {
70     bufferTAD buffer = malloc(sizeof(struct bufferTCD));
71     if (buffer == NULL)
72     {
73         fprintf(stderr, "Erro: impossível alocar buffer.\n");
74         return NULL;
75     }
76
77     buffer->antes = criar_stackTAD();
78     buffer->depois = criar_stackTAD();
79     if (buffer->antes == NULL || buffer->depois == NULL)
80     {
81         fprintf(stderr, "Erro: impossível alocar pilhas do buffer.\n");
82         return NULL;
83     }
84
85     return buffer;
86 }
```

TAD buffer: implementação com PILHA

```
 83 /**
 84  * PROCEDIMENTO: liberar_buffer
 85  * Uso: liberar_buffer(buffer);
 86  * ----------------------------
 87  * Este procedimento libera o espaço de armazenamento alocado para o buffer. O
 88  * argumento deve ser um PONTEIRO para o buffer (um ponteiro para ponteiro para
 89  * struct bufferTCD).
 90  */
 91
 92 void liberar_buffer (bufferTAD *buffer)
 93 {
 94     if (*buffer != NULL)
 95     {
 96         remover_stackTAD(&((*buffer)->antes));
 97         remover_stackTAD(&((*buffer)->depois));
 98         free(*buffer);
 99         buffer = NULL;
100     }
101 }
```

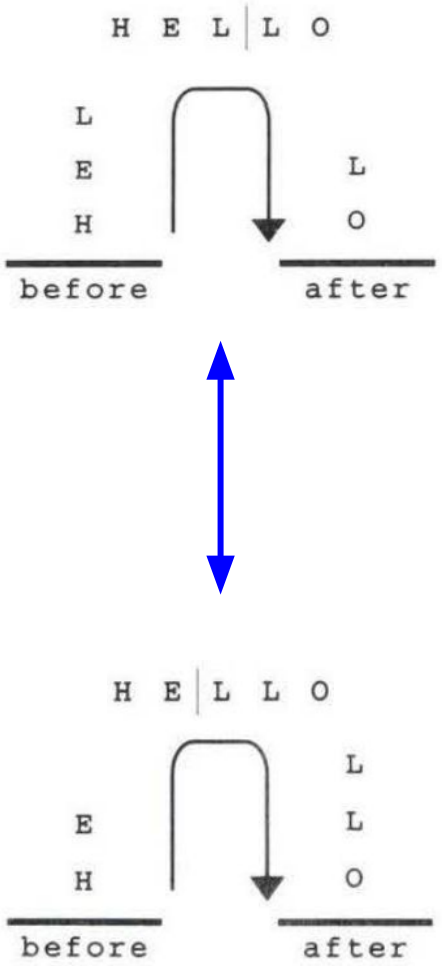
TAD buffer: implementação com PILHA

```
114 void mover_cursor_para_frente (bufferTAD buffer)
115 {
116     if (buffer == NULL)
117     {
118         fprintf(stderr, "Erro: movimentação em buffer null.\n");
119         exit(1);
120     }
121
122     if (!vazia(buffer->depois))
123         push(buffer->antes, pop(buffer->depois));
124 }
125
126 void mover_cursor_para_tras (bufferTAD buffer)
127 {
128     if (buffer == NULL)
129     {
130         fprintf(stderr, "Erro: movimentação em buffer null.\n");
131         exit(1);
132     }
133
134     if (!vazia(buffer->antes))
135         push(buffer->depois, pop(buffer->antes));
136 }
```



TAD buffer: implementação com PILHA

```
148 void mover_cursor_para_final (bufferTAD buffer)
149 {
150     if (buffer == NULL)
151     {
152         fprintf(stderr, "Erro: movimentação em buffer null.\n");
153         exit(1);
154     }
155
156     while (!vazia(buffer->depois))
157         push(buffer->antes, pop(buffer->depois));
158 }
159
160 void mover_cursor_para_inicio (bufferTAD buffer)
161 {
162     if (buffer == NULL)
163     {
164         fprintf(stderr, "Erro: movimentação em buffer null.\n");
165         exit(1);
166     }
167
168     while (!vazia(buffer->antes))
169         push(buffer->depois, pop(buffer->antes));
170 }
```



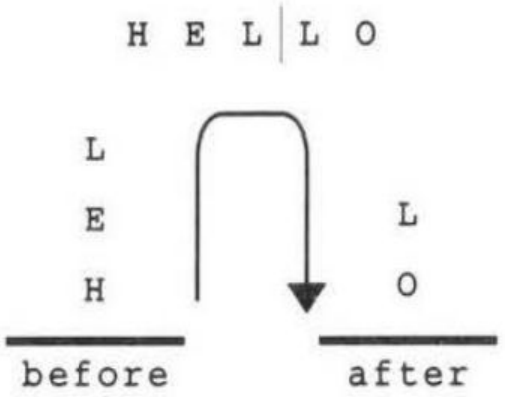
TAD buffer: implementação com PILHA

```
172 /**
173  * PROCEDIMENTO: inserir_caractere
174  * Uso: inserir_caractere(buffer, c);
175  * ----------------------------------
176  * Insere o caractere "c" no buffer "buffer", na posição atual do cursor. Após
177  * a inserção o cursos é posicionado após o caractere inserido, para permitir
178  * inserções consecutivas.
179  */
180
181 void inserir_caractere (bufferTAD buffer, char c)
182 {
183     if (buffer == NULL)
184     {
185         fprintf(stderr, "Erro: inserção em buffer null.\n");
186         exit(1);
187     }
188
189     push(buffer->antes, c);
190 }
```

TAD buffer: implementação com PILHA

```
192 /**
193  * PROCEDIMENTO: apagar_caractere
194  * Uso: apagar_caractere(buffer);
195  * ------------------------------
196  * Apaga o caractere imediatamente posterior ao cursor. Se o cursor já está no
197  * final do buffer, não causa nenhum efeito.
198  */
199
200 void apagar_caractere (bufferTAD buffer)
201 {
202     if (buffer == NULL)
203     {
204         fprintf(stderr, "Erro: remoção em buffer null.\n");
205         exit(1);
206     }
207
208     if (!vazia(buffer->depois))
209         (void) pop(buffer->depois);
210 }
```

TAD buffer: implementação com PILHA




```
212 /**
213  * PROCEDIMENTO: exibir_buffer
214  * Uso: exibir_buffer(buffer);
215  * ---------------------------
216  * Exibe o conteúdo atual do buffer no terminal.
217  */
218
219 void exibir_buffer (bufferTAD buffer)
220 {
221     for (int i = 0; i < qtd_elementos(buffer->antes); i++)
222         printf(" %c", ver_elemento(buffer->antes, i));
223
224     for (int i = qtd_elementos(buffer->depois) - 1; i >= 0; i--)
225         printf(" %c", ver_elemento(buffer->depois, i));
226
227     printf("\n");
228
229     for (int i = 0; i < qtd_elementos(buffer->antes); i++)
230         printf("  ");
231
232     printf("^\n");
233 }
```

TAD buffer: implementação com PILHA

```
[abrantesasf@ideapad ~/ed1/cap09]$ ./meu_editor_stackbuff
*IAbrantes Araujo Silva filho
 A b r a n t e s   A r a u j o   S i l v a   f i l h o
                                                     ^

*b
 A b r a n t e s   A r a u j o   S i l v a   f i l h o
                                                   ^

*b
 A b r a n t e s   A r a u j o   S i l v a   f i l h o
                                                  ^

*b
 A b r a n t e s   A r a u j o   S i l v a   f i l h o
                                                ^

*b
 A b r a n t e s   A r a u j o   S i l v a   f i l h o
                                               ^

*b
 A b r a n t e s   A r a u j o   S i l v a   f i l h o
                                              ^

*d
 A b r a n t e s   A r a u j o   S i l v a    i l h o
                                              ^

*IF
 A b r a n t e s   A r a u j o   S i l v a   F i l h o
                                               ^

*e
 A b r a n t e s   A r a u j o   S i l v a   F i l h o
                                                     ^

*q
```

TAD buffer: eficiência da implementação com PILHA

- As operações que movem o cursor 1 caractere por vez executam em tempo constante: O(1)

- As operações de inserção/remoção também são O(1)

- Já as operações de mover para o início/final, como fazem pop/push de todos os elementos, são proporcionais ao número de caracteres no buffer: O(N)

TAD buffer: ARRAY x PILHA

Function	Array	Stack
MoveCursorForward	$O(1)$	$O(1)$
MoveCursorBackward	$O(1)$	$O(1)$
MoveCursorToStart	$O(1)$	$O(N)$
MoveCursorToEnd	$O(1)$	$O(N)$
InsertCharacter	$O(N)$	$O(1)$
DeleteCharacter	$O(N)$	$O(1)$

Qual é melhor?

TAD buffer: ARRAY x PILHA

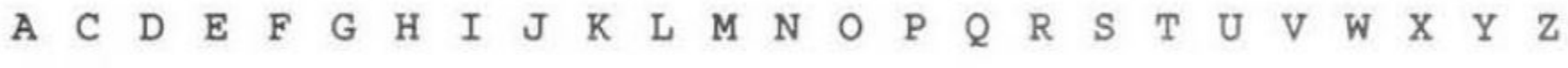
Function	Array	Stack
MoveCursorForward	$O(1)$	$O(1)$
MoveCursorBackward	$O(1)$	$O(1)$
MoveCursorToStart	$O(1)$	$O(N)$
MoveCursorToEnd	$O(1)$	$O(N)$
InsertCharacter	$O(N)$	$O(1)$
DeleteCharacter	$O(N)$	$O(1)$

- **Depende do uso esperado**! Em um editor de buffer, as **operações de inserção e remoção de caracteres são muito mais usadas do que as operações de mover para o início e mover para o final do buffer**, então a implementação através de PILHAS é melhor!

- Existe uma implementação onde todas as operações sejam rápidas? Sim! Mas temos que aprender 2 estruturas de dados novas:
  - **Lista encadeada simples** (lista simplesmente encadeada - LSE)
  - **Lista encadeada dupla** (lista duplamente encadeada - LDE)

TAD buffer: implementação com LISTA ENCADEADA

- Por que a implementação com array era O(N) na inserção e remoção? Imagine, por exemplo, que você esqueceu a letra “B”, no array abaixo. A única alternativa é deslocar todas as letras (exceto a “A”) e depois inserir a letra “B”.



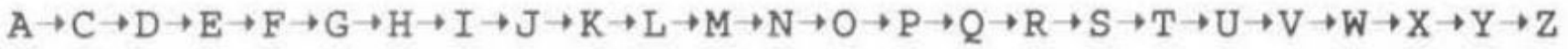
- E se você estivesse escrevendo em PAPEL, usando CANETA. Como você faria para consertar a falta da letra “B”? Uma solução é criar um “desvio” no fluxo de leitura:

TAD buffer: implementação com LISTA ENCADEADA

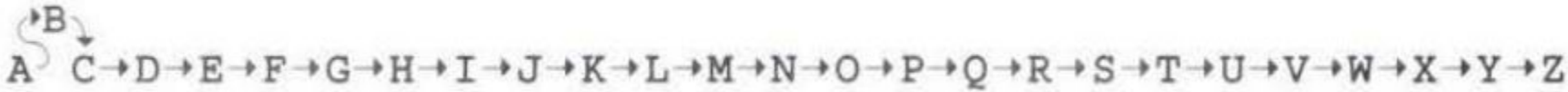
- A vantagem dessa notação “humana” é que nos permite suspender a regra de que todas as letras estão organizadas em seqüência exatamente do jeito que estão impressas na página. O símbolo abaixo da linha cria um “desvio” no fluxo. E não importa o tamanho da linha: escrever a marca e a letra é uma inserção que é executada em tempo O(1).

TAD buffer: implementação com LISTA ENCADEADA

- Uma lista encadeada adota uma estratégia semelhante: ao invés de usar um array com desvios ocasionais do fluxo, a lista encadeada considera que o fluxo é dado por uma seqüência de ponteiros entre as letras. Assim, na situação original teríamos:



- E, após inserir a letra “B”, teríamos:



- Note que agora, a letra “B” não é um desvio, faz parte do fluxo normal!

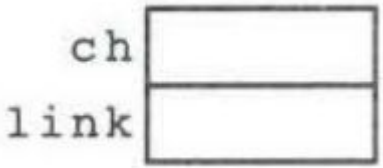
TAD buffer: implementação com LISTA ENCADEADA

- Uma lista encadeada é uma estrutura de dados linear composta por uma sequência de elementos, chamados de **nós** ou **células**, onde cada nó/célula contém um **valor** (ou dado) e um **ponteiro** para o próximo nó/célula na sequência.
    - ATENÇÃO: o nó não é só o valor: é a combinação do valor e do ponteiro para o próximo nó.

- Não utilizam índices e não precisam de um bloco contínuo de memória (os nós/células podem estar espalhados em qualquer local da memória, basta seguir os ponteiros para acessá-los).

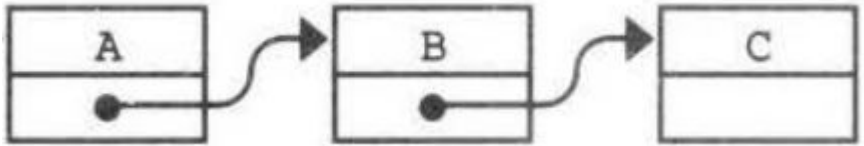
TAD buffer: implementação com LISTA ENCADEADA

- A combinação do valor armazenado (char, int, double, struct, TAD, etc…) e do ponteiro para o próximo elemento é o bloco básico de construção da lista encadeada. Esse bloco é a célula ou nó. No caso de um editor de buffer o valor de um nó armazena 1 caractere e o ponteiro para o próximo nó:



TAD buffer: implementação com LISTA ENCADEADA

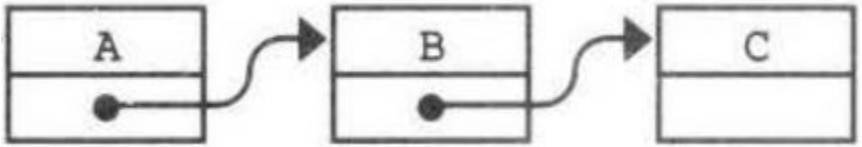
- Três letras encadeadas seriam representadas da seguinte maneira:



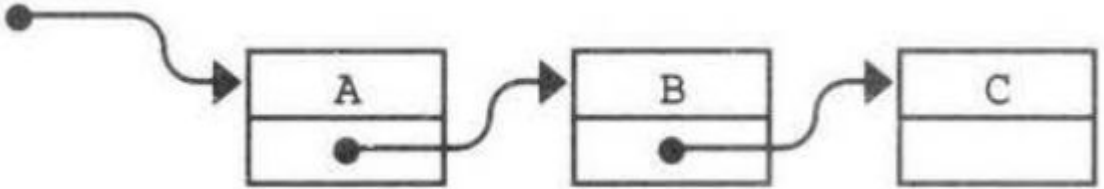
- Há dois grandes problemas na lista acima: quais?

TAD buffer: implementação com LISTA ENCADEADA

- O primeiro problema é que não existe um ponteiro para o 1º elemento da lista e, sem esse ponteiro, não conseguimos nem acessar a lista:

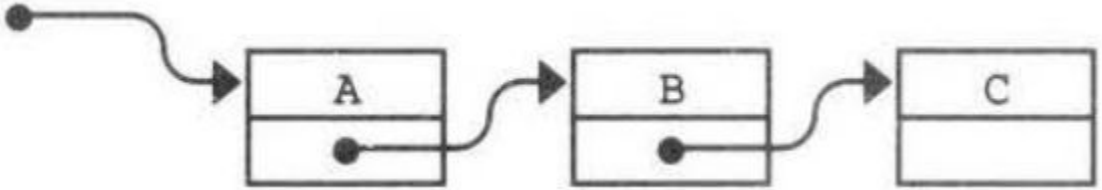


- O correto é ter um ponteiro para algum elemento inicial da lista:

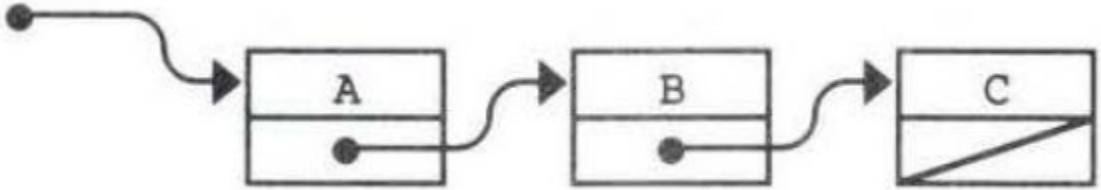


TAD buffer: implementação com LISTA ENCADEADA

- O segundo problema é que não sabemos se a lista termina ou continua após a letra “C”, pois não temos nenhuma informação sobre o ponteiro:



- O correto é ter um ponteiro para NULL no último elemento, para indicar que a lista terminou:



TAD buffer: implementação com LISTA ENCADEADA

- Como criar o nó/célula? A princípio, fazendo algo assim:

```
typedef struct
{
    char letra;
    celulaT *proximo;
} celulaT;
```

- O problema é que **o código acima está ERRADO**. O conceito está certo: a célula contém o valor e um ponteiro para a próxima célula. O problema está no momento da criação do ponteiro: para o compilador, o ponteiro próximo aponta para uma coisa que ainda não existe (celulaT).

TAD buffer: implementação com LISTA ENCADEADA

- O problema ocorre pois **`celulaT`** é um **tipo de dado recursivo**: um tipo de dado que é definido em termos de si mesmo.

- Em C, ao criarmos tipos de dados recursivos, temos que incluir uma structure tag (que vimos no capítulo 8):

```
typedef struct celulaTCD
{
    char letra;
    struct celulaTCD *proximo;
} celulaTCD;

typedef struct celulaTCD *celulaTAD;
```

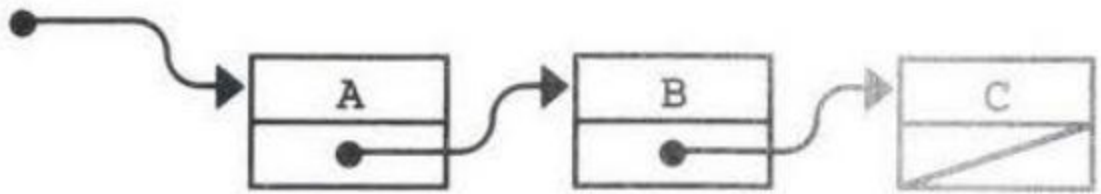
TAD buffer: implementação com LISTA ENCADEADA

- Como representar o bufferTAD então? Precisamos de, no mínimo, duas coisas:
  - Onde o buffer começa
  - A posição do cursor

```
struct bufferTCD
{
    celulaTAD inicio;
    celulaTAD cursor;
};
```

TAD buffer: implementação com LISTA ENCADEADA

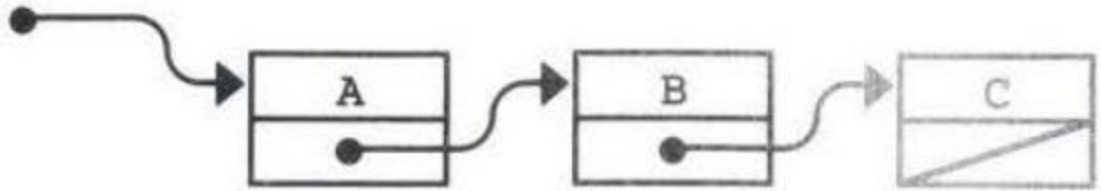
- Armazenar onde o buffer começa é relativamente fácil: bastaria pegar o ponteiro para a primeira letra e armazenar no ponteiro **`inicio`**:




```
struct bufferTCD
{
    celulaTAD inicio;
    celulaTAD cursor;
};
```

TAD buffer: implementação com LISTA ENCADEADA

- O maior problema é com o cursor: no exemplo temos 3 células, mas temos 4 posições para o cursor:

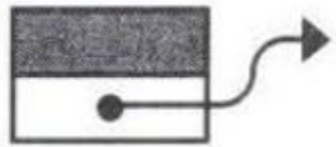


- Se temos 4 posições do cursor, mas só temos 3 células para apontar, como resolver?

```
struct bufferTCD
{
    celulaTAD inicio;
    celulaTAD cursor;
};
```

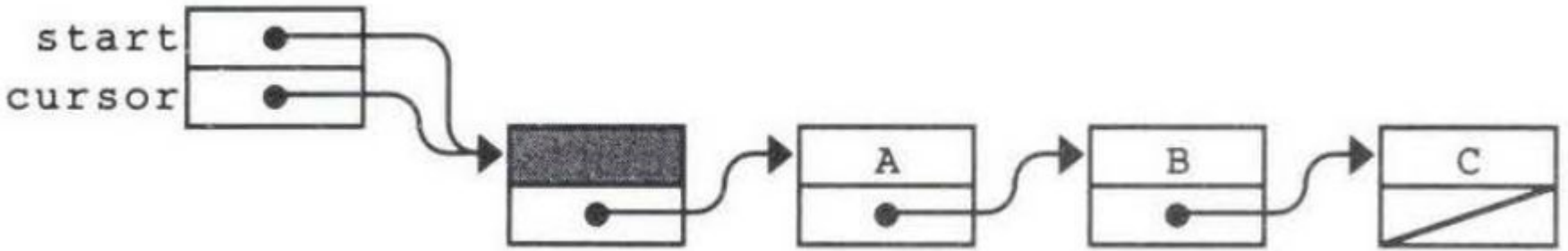
TAD buffer: implementação com LISTA ENCADEADA

- Uma das boas soluções possíveis é utilizar uma “dummy cell”, uma “célula boba” extra no início da lista, para fazer com que a lista contenha uma célula para cada possível ponto de inserção do cursor. O valor armazenado na célula dummy é irrelevante, o que importa é apenas o ponteiro:



TAD buffer: implementação com LISTA ENCADEADA

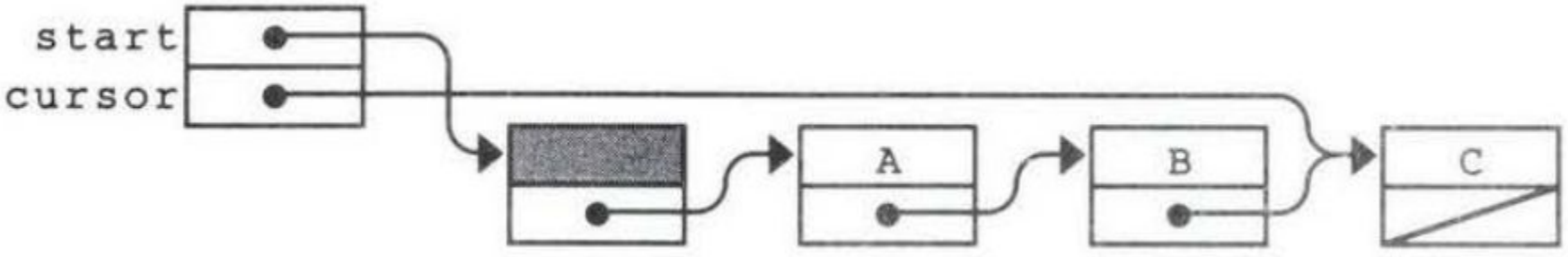
- Quando usamos a abordagem de célula boba, **o cursor aponta para a célula imediatamente anterior ao ponto de inserção**:



No exemplo: **O CURSOR APONTA PARA A CÉLULA IMEDIATAMENTE ANTES DO PONTO DE INSERÇÃO**, então o ponto de inserção é antes da letra “A”

TAD buffer: implementação com LISTA ENCADEADA

- Quando usamos a abordagem de célula boba, **o cursor aponta para a célula imediatamente anterior ao ponto de inserção**:

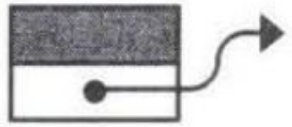


No exemplo: **O CURSOR APONTA PARA A CÉLULA IMEDIATAMENTE ANTES DO PONTO DE INSERÇÃO**, então o ponto de inserção é depois da letra “C”

TAD buffer: implementação com LISTA ENCADEADA

- O uso de uma célula boba é extremamente útil, mas não é imprescindível e também não é usado em todas as listas encadeadas. Isso dependerá da aplicação da lista que está sendo desenvolvida.

- A maior vantagem de usar uma célula boba é **reduzir o número de casos especiais que você precisa considerar quando fizer inserções e remoções em pontos arbitrários na lista** pois, ao usar uma célula boba, o único caso presente é que o cursor aponta para a célula imediatamente anterior ao ponto de inserção. Se você não usar uma célula boba, terá que tratar a inserção na primeira posição de modo diferente das demais, complicando o código.



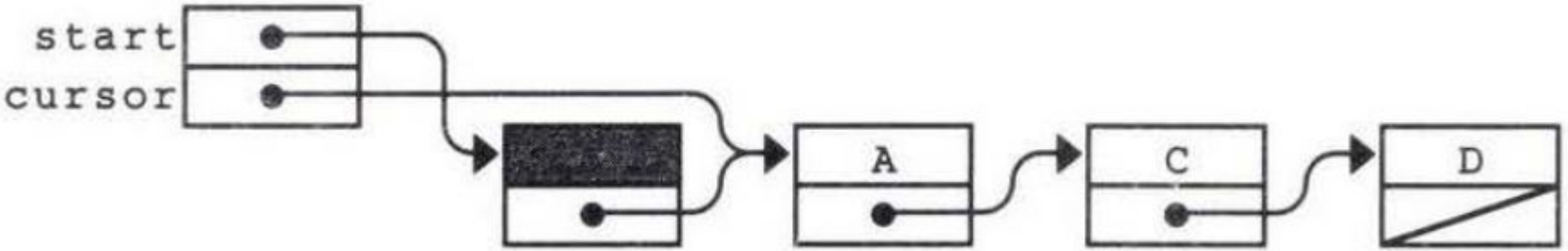
TAD buffer: implementação com LISTA ENCADEADA

- A **inserção** em uma lista encadeada é uma operação que segue uma seqüência exata de passos:
  a. Aloque uma nova célula e armazene o ponteiro para essa nova célula em um variável temporária;
  b. Copie o caractere a ser inserido no membro “letra” da nova célula;
  c. Vá para a célula indicada pelo cursor e copie o ponteiro próximo para o ponteiro próximo da nova célula
  d. Altere o ponteiro próximo da célula apontada pelo cursor e faça com que ele aponte para a nova célula
  e. Altere o cursor para que ele aponte para a nova célula
  f. Eliminação da variável temporária (feita diretamente pela remoção do stack frame)

- Vamos ver a inserção da letra “B” aqui:

TAD buffer: implementação com LISTA ENCADEADA

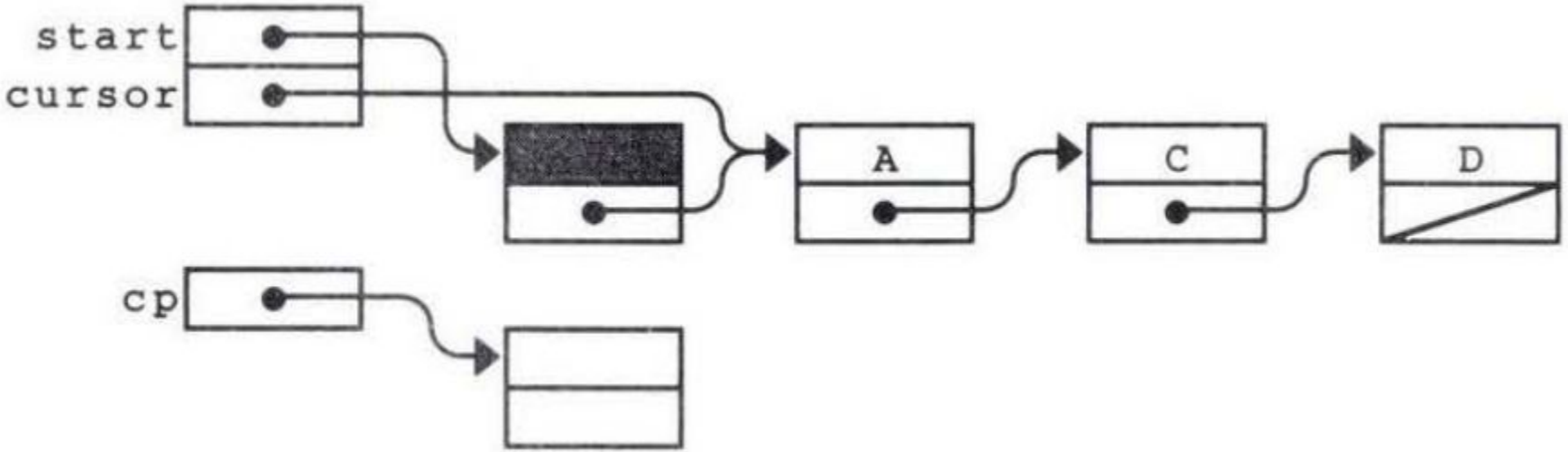
- Supondo que o cursor já esteja entre “A” e “C” (veremos a movimentação do cursor depois), a situação inicial é a seguinte:



Por que o cursor está entre “A” e “C”?

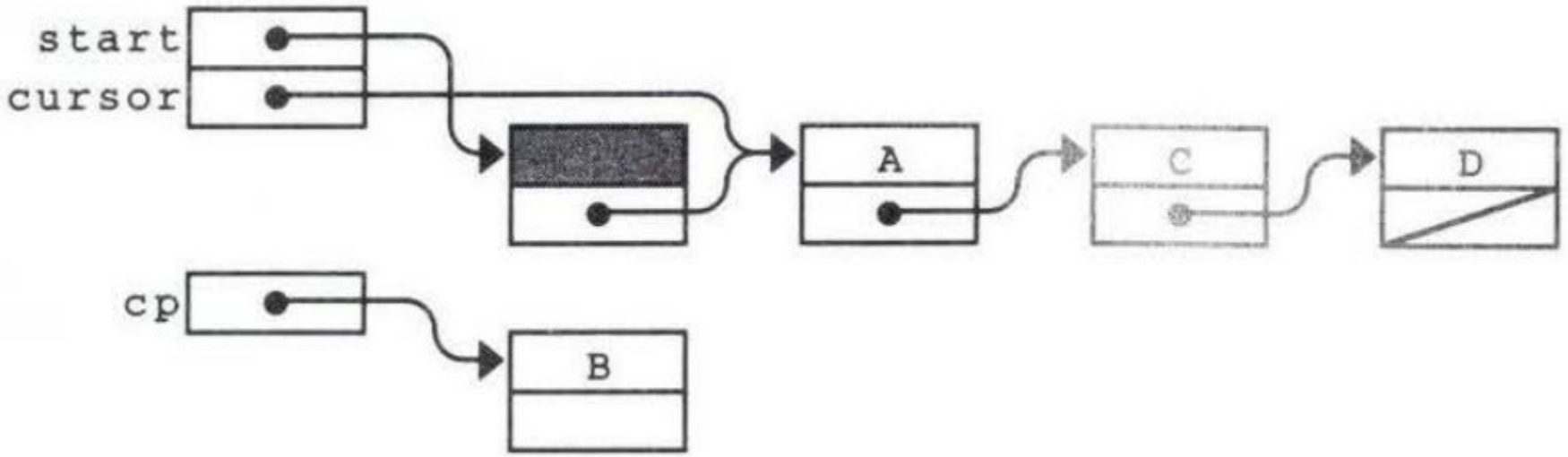
TAD buffer: implementação com LISTA ENCADEADA

- 1º passo: aloque uma nova célula e armazene o ponteiro para essa nova célula em um variável temporária.



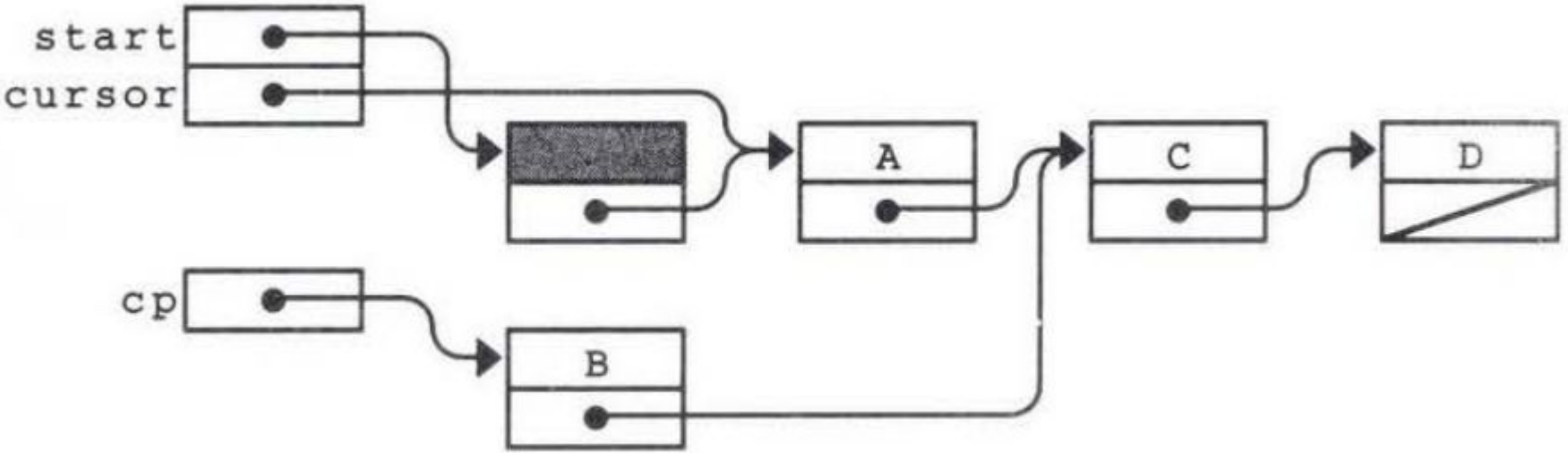
TAD buffer: implementação com LISTA ENCADEADA

- 2º passo: copie o caractere a ser inserido no membro “letra” da nova célula.



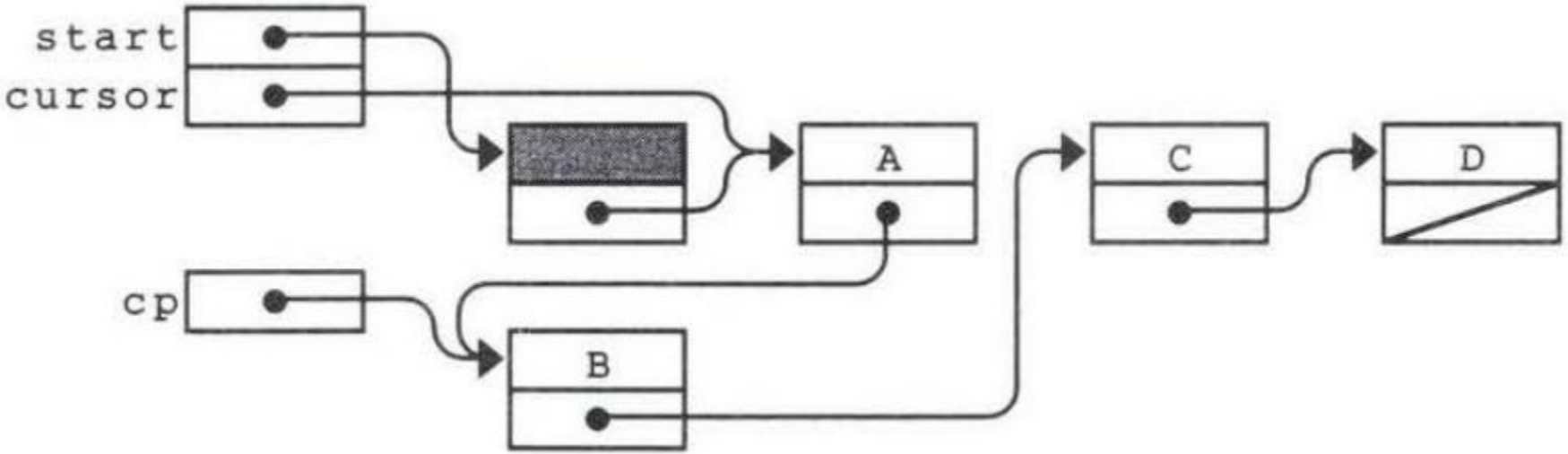
TAD buffer: implementação com LISTA ENCADEADA

- 3º passo: vá para a célula indicada pelo cursor e copie o ponteiro próximo para o ponteiro próximo da nova célula.



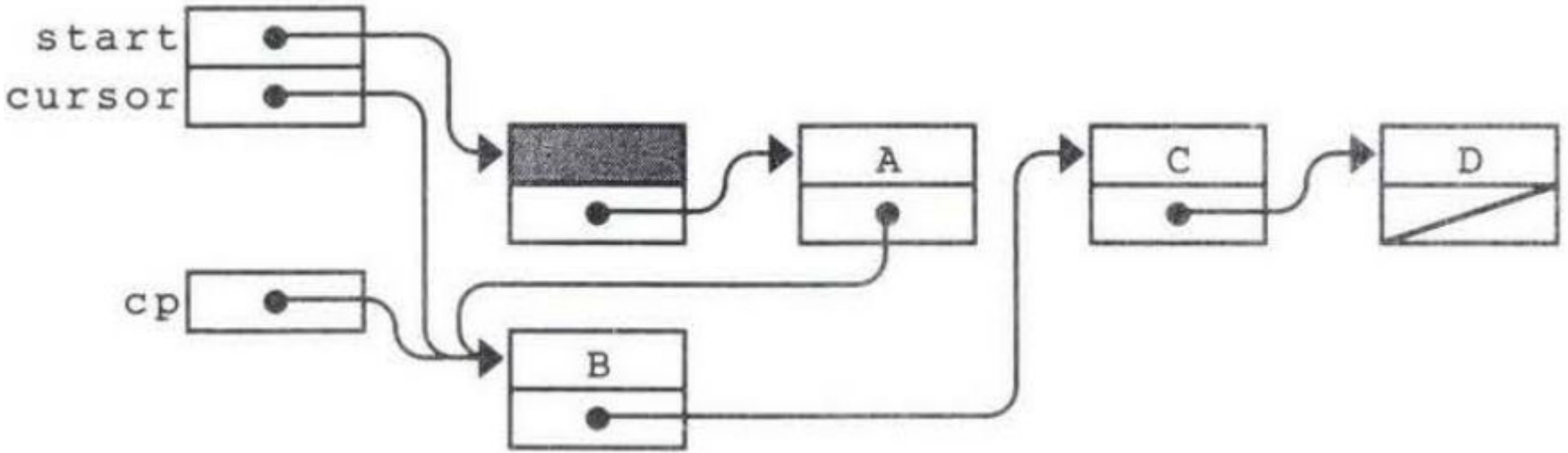
TAD buffer: implementação com LISTA ENCADEADA

- 4º passo: altere o ponteiro próximo da célula apontada pelo cursor e faça com que ele aponte para a nova célula.



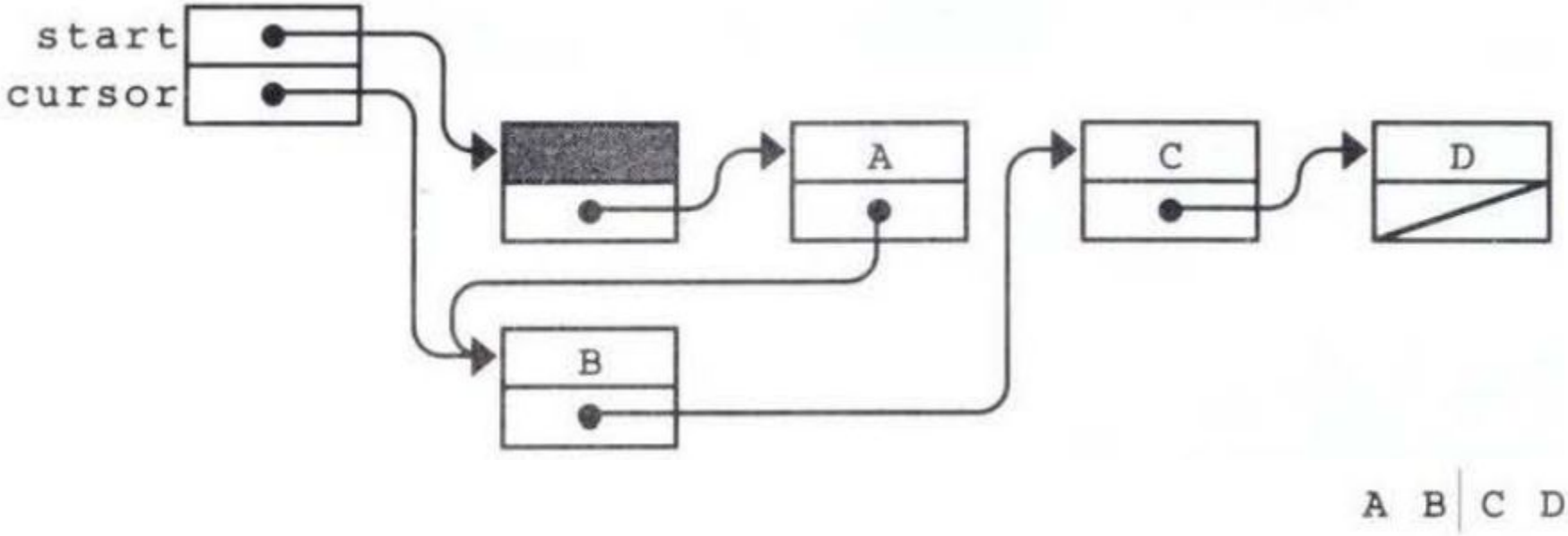
TAD buffer: implementação com LISTA ENCADEADA

- 5º passo: altere o cursor para que ele aponte para a nova célula.



TAD buffer: implementação com LISTA ENCADEADA

- 6º passo: eliminação da variável temporária (feita diretamente pela remoção do stack frame).



TAD buffer: implementação com LISTA ENCADEADA

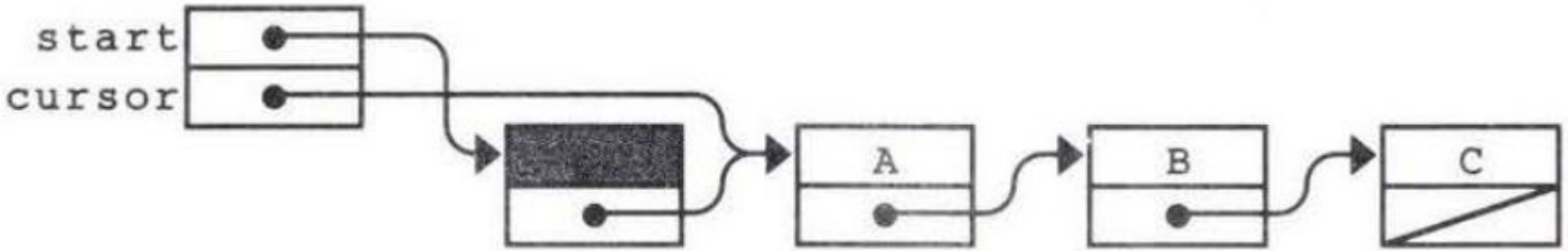
```
void
inserir_caractere (bufferTAD buffer, char c)
{
    if (buffer == NULL)
    {
        fprintf(stderr, "Erro: inserção em buffer null.");
        exit(1);
    }

    // 1: cria nova célula na memória e retorna um ponteiro para a essa célula:
    celulaTAD pc = criar_celula();
    if (pc == NULL)
    {
        fprintf(stderr, "Erro: impossível alocar célula.\n");
        exit(1);
    }

    // 2: copia o caractere para a nova célula:
    pc->letra = c;

    // 3: copia o "próximo" do cursor para o ponteiro "próximo" da nova célula:
    pc->proximo = buffer->cursor->proximo;

    // 4: altera o "próximo" do cursor para que apontar para o nova célula:
    buffer->cursor->proximo = pc;

    // 5: faz o cursor para apontar para a nova célula:
    buffer->cursor = pc;

    // 6: remove o ponteiro temporário (não estritamente necessário):
    pc = NULL;
}
```

TAD buffer: implementação com LISTA ENCADEADA

- A **remoção** em uma lista encadeada é uma operação mais simples: basta remover a célula da cadeia de ponteiros e liberar memória.

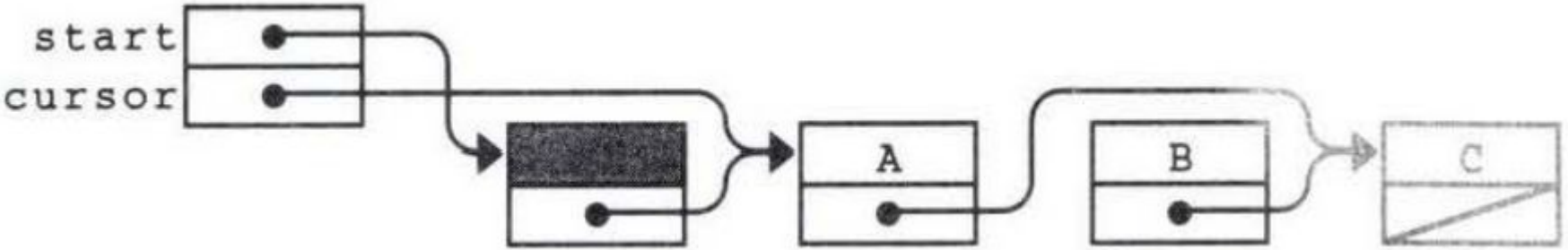
- Vamos assumir que queremos remover a letra “B” de um buffer nesta situação:

A | B C



TAD buffer: implementação com LISTA ENCADEADA

- Tudo que precisamos fazer é alterar os ponteiros e desalocar a célula que foi retirada da lista:



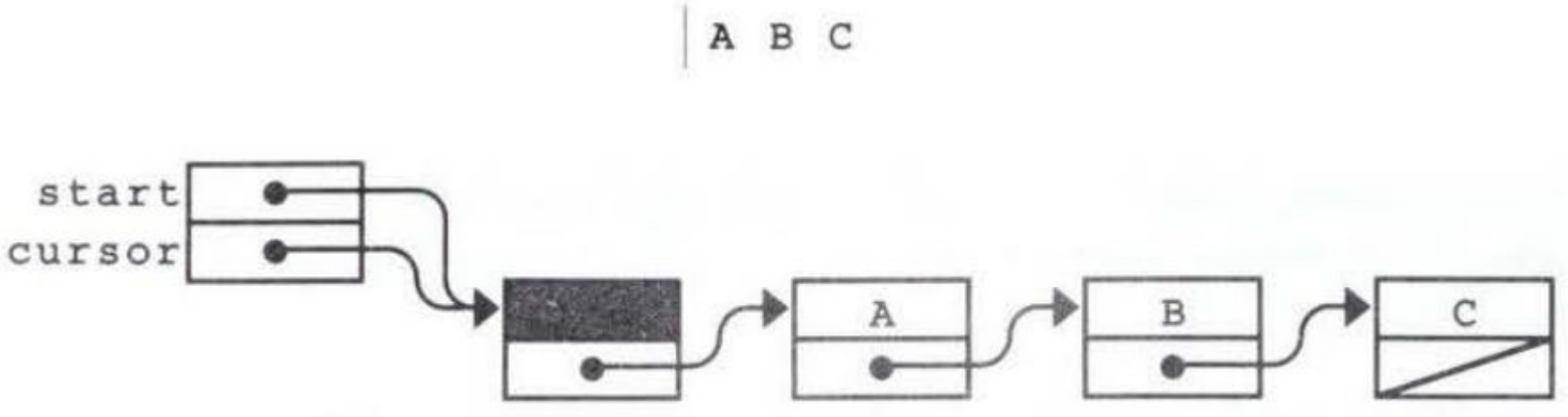
- ATENÇÃO: antes de alterar o ponteiro “próximo” da letra “A”, você deve salvar o endereço da letra “B” em um ponteiro temporário, para poder liberar a memória dessa célula! Se não fizer isso estará causando memory leak.

TAD buffer: implementação com LISTA ENCADEADA

```
void
apagar_caractere (bufferTAD buffer)
{
    celulaTAD temp;

    if (buffer->cursor->proximo != NULL)
    {
        temp = buffer->cursor->proximo;
        buffer->cursor->proximo = temp->proximo;
        remover_celula(&temp);
    }

    temp = NULL;
}
```

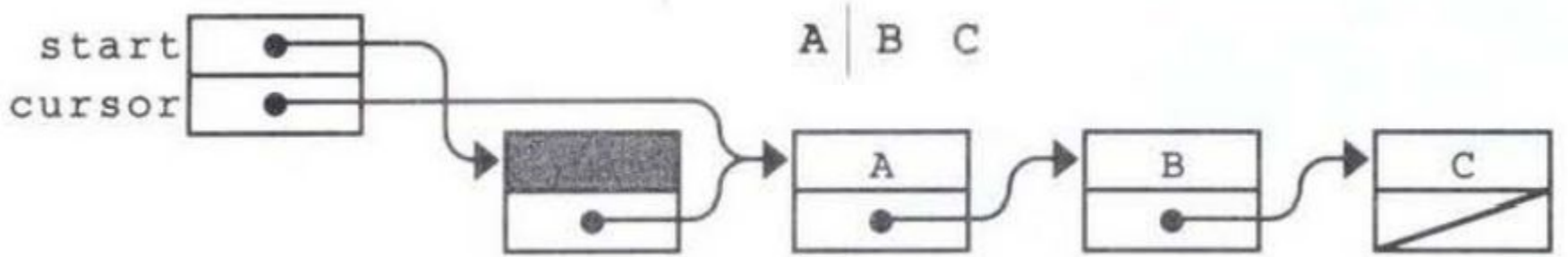
TAD buffer: implementação com LISTA ENCADEADA

- A movimentação do cursor não é difícil de implementar, dependendo do movimento. Para **mover o cursor para frente**, basta verificar se já não estamos no final da lista e ajustar o ponteiro do cursor:



TAD buffer: implementação com LISTA ENCADEADA

```
void
mover_cursor_para_frente (bufferTAD buffer)
{
    if (buffer == NULL)
    {
        fprintf(stderr, "Erro: movimentação em buffer null.\n");
        exit(1);
    }

    if (buffer->cursor->proximo != NULL)
    {
        buffer->cursor = buffer->cursor->proximo;
    }
}
```



TAD buffer: implementação com LISTA ENCADEADA

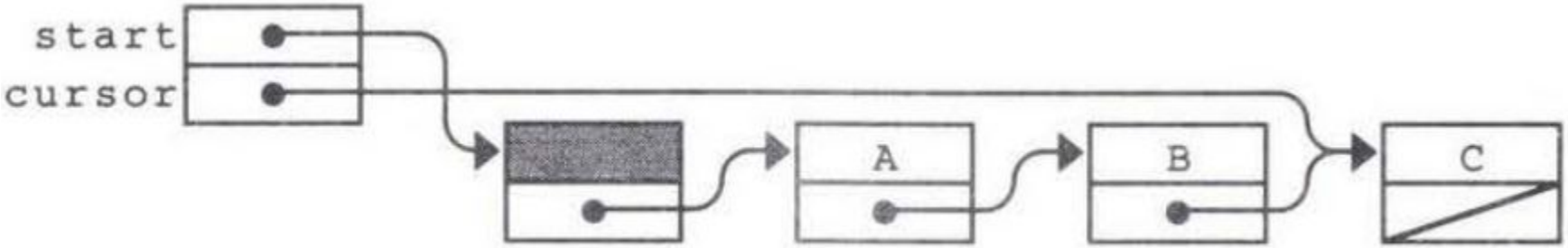
- A movimentação do cursor não é difícil de implementar, dependendo do movimento. Para **mover o cursor para o início** do buffer, basta ajustar o cursor para a célula boba:

```
void
mover_cursor_para_inicio (bufferTAD buffer)
{
    if (buffer == NULL)
    {
        fprintf(stderr, "Erro: movimentação em buffer null.\n");
        exit(1);
    }

    buffer->cursor = buffer->inicio;
}
```

TAD buffer: implementação com LISTA ENCADEADA

- As operações de **mover para trás** e **mover para o final** são mais difíceis de implementar.

- Suponha que o cursor esteja no final do buffer e você queira voltar 1 posição. Visualmente você está na seguinte situação:

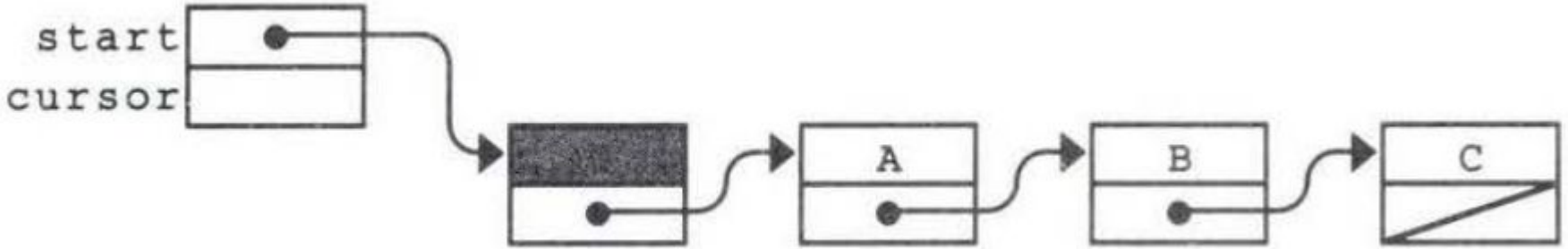


- Não há estratégia O(1) para voltar 1 posição! O problema é que os ponteiros só “vão para a frente”, você não consegue voltar um ponteiro para trás.

TAD buffer: implementação com LISTA ENCADEADA

- Você tem que implementar todas as operações do ponto de vista do buffer.
- Do ponto de vista do buffer, se olharmos apenas o cursor veremos isso:

- Do ponto de vista do buffer, se olharmos o início, veremos isso:



TAD buffer: implementação com LISTA ENCADEADA

- As duas visões, separadamente, não nos ajudam:
  - Na visão do cursor não sabermos quais as células anteriores;
  - Na visão do início conseguimos ver as células, mas não sabemos onde está o cursor

- Para **mover o cursor para trás**, temos então que começar no início e ir percorrendo a lista até encontrar uma célula cujo ponteiro próximo aponte para o mesmo local do cursor. Essa é a célula imediatamente anterior ao cursor.

TAD buffer: implementação com LISTA ENCADEADA

```
void
mover_cursor_para_tras (bufferTAD buffer)
{
    if (buffer == NULL)
    {
        fprintf(stderr, "Erro: movimentação em buffer null.\n");
        exit(1);
    }

    celulaTAD temp;

    if (buffer->cursor != buffer->inicio)
    {
        temp = buffer->inicio;
        while (temp->proximo != buffer->cursor)
        {
            temp = temp->proximo;
        }
        buffer->cursor = temp;
    }
}
```

TAD buffer: implementação com LISTA ENCADEADA

- Para **mover o cursor para o final**, também temos que percorrer a lista, andando para frente até encontrarmos a célula cujo ponteiro próximo aponta para NULL

```
void
mover_cursor_para_final (bufferTAD buffer)
{
    if (buffer == NULL)
    {
        fprintf(stderr, "Erro: movimentação em buffer null.\n");
        exit(1);
    }

    while (buffer->cursor->proximo != NULL)
    {
        mover_cursor_para_frente(buffer);
    }
}
```

TAD buffer: implementação com LISTA ENCADEADA

- Existem vários idiomas comuns em listas encadeadas. Por exemplo, para mover o cursor para trás:

```
temp = buffer->inicio;
while (temp->proximo != buffer->cursor)
{
    temp = temp->proximo;
}
buffer->cursor = temp;
```

```
celulaTAD cp;
for (cp = buffer->inicio; cp->proximo != buffer->cursor; cp = cp->proximo)
{
    ;
}
```

TAD buffer: implementação com LISTA ENCADEADA

- Existem vários idiomas comuns em listas encadeadas. Para executar uma operação em todos os elementos da lista:

```
for (cp = list; cp != NULL; cp = cp->link) {
    code using cp
}
```

- Com o tempo você aprenderá diversos idiomas para listas encadeadas.

TAD buffer: implementação com LISTA ENCADEADA

- Terminando a implementação:

```
bufferTAD
criar_buffer (void)
{
    bufferTAD B = calloc(1, sizeof(struct bufferTCD));
    if (B == NULL)
    {
        fprintf(stderr, "Erro: impossível alocar buffer.\n");
        return NULL;
    }

    celulaTAD temp = criar_celula();
    if (temp == NULL)
    {
        fprintf(stderr, "Erro: a célula não foi criada.\n");
        free(B);
        B = NULL;
        return NULL;
    }

    B->inicio = B->cursor = temp;
    B->inicio->proximo = NULL;

    temp = NULL;

    return B;
}
```

TAD buffer: implementação com LISTA ENCADEADA

- Terminando a implementação:

```
void
liberar_buffer (bufferTAD *buffer)
{
    if (buffer == NULL || *buffer == NULL)
    {
        fprintf(stderr, "Erro: buffer inválido.\n");
    }
    else
    {
        celulaTAD atual, proxima;
        atual = (*buffer)->inicio;
        while (atual != NULL)
        {
            proxima = atual->proximo;
            remover_celula(&atual);
            atual = proxima;
        }
        free(*buffer);
        *buffer = NULL;
    }
}
```

TAD buffer: implementação com LISTA ENCADEADA

- Ao analisar a complexidade computacional da implementação com lista encadeada, temos o seguinte:

Function	Array	Stack	List
MoveCursorForward	$O(1)$	$O(1)$	$O(1)$
MoveCursorBackward	$O(1)$	$O(1)$	$O(N)$
MoveCursorToStart	$O(1)$	$O(N)$	$O(1)$
MoveCursorToEnd	$O(1)$	$O(N)$	$O(N)$
InsertCharacter	$O(N)$	$O(1)$	$O(1)$
DeleteCharacter	$O(N)$	$O(1)$	$O(1)$

- Ué, mas não tínhamos falado que a implementação com listas encadeadas seria melhor? Mas ainda tem O(N)...

TAD buffer: implementação com LISTA ENCADEADA

- Ainda não conseguimos nos livrar do O(N) pois na lista encadeada que vimos os ponteiros só apontam para frente! Isso faz com que as operações de andar para trás e de andar para o final da lista sejam O(N).

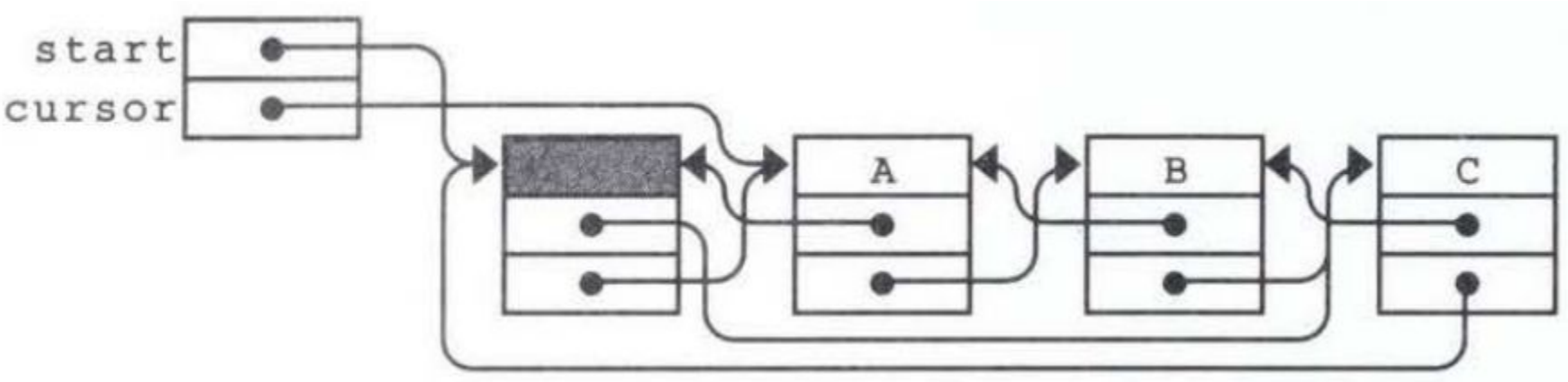
- Para resolver mesmo o problema temos que conhecer outros tipos de lista encadeada. Basicamente temos os seguintes tipos de listas encadeadas:

    - Lista Simplesmente Encadeada (LSE)
    - Lista Duplamente Encadeada (LDE)
    - Lista Circular Simplesmente Encadeada (LCSE)
    - Lista Circular Duplamente Encadeada (LCDE)

TAD buffer: implementação com LISTA ENCADEADA

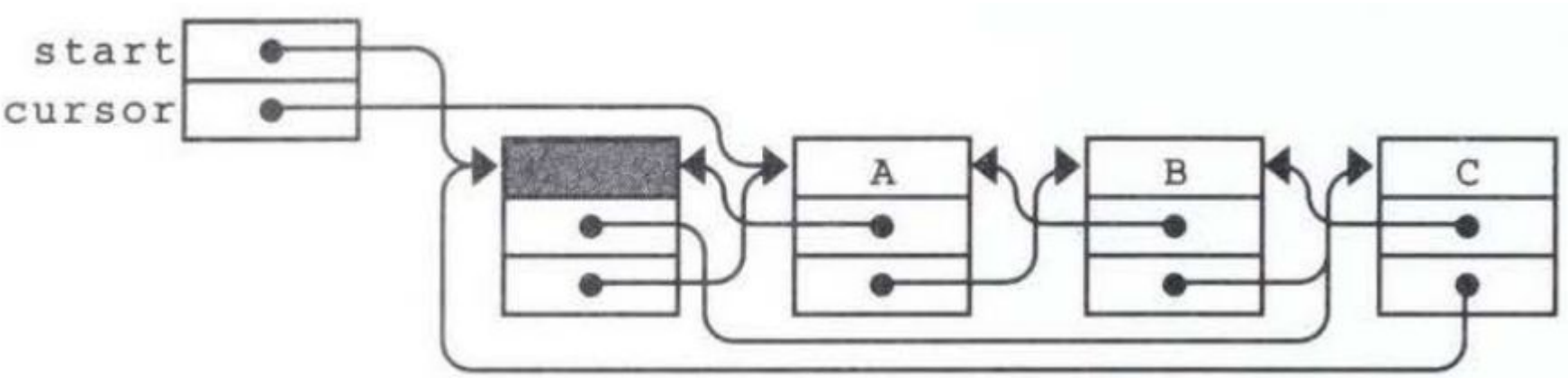
- A lista que implementamos foi uma LSE, uma lista simplesmente encadeada, onde cada célula só tem um ponteiro para a célula da frente.

- Em uma lista duplamente encadeada (LDE), cada célula tem 2 ponteiros: o ponteiro “próximo” e o ponteiro “anterior”:



TAD buffer: implementação com LISTA ENCADEADA

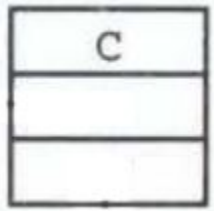
- Para facilitar a implementação, fazemos o seguinte:
  - O ponteiro “anterior” da dummy cell aponta para a última célula do buffer;
  - O ponteiro “próximo” da última célula do buffer aponta para a dummy cell.



- Note que estamos criando aqui uma lista circular duplamente encadeada. Essa lista pode ter tamanho fixo ou não, dependendo da implementação.

TAD buffer: balanço entre tempo e espaço

- Se usarmos uma LDE (ou uma LCDE) conseguiremos que todas as operações do buffer sejam O(1). Mas, em compensação, usaremos 17 bytes para armazenar cada caractere:



- As operações serão todas rápidas, mas gastaremos muito mais espaço do que, por exemplo, a implementação com arrays (1 byte por caractere);

- Isso é o **compromisso tempo-espaço** que o programador deve decidir.

TAD buffer: balanço entre tempo e espaço

- Quando o programador está diante de um **dilema tempo-espaço**, pode ser possível desenvolver uma estratégia híbrida, que não gaste muito espaço mas que também não contenha operações lentas:
    - Combinar um array e lista encadeada, por exemplo:
        - representar o buffer como uma LDE de linhas; e
        - cada linha é um array
- Estratégias híbridas existem e, algumas vezes em situações especiais, são utilizadas. São difíceis de programar corretamente.

Em resumo

- A eficiência dos programas depende de nossos algoritmos e, também, das estruturas de dados que utilizamos em nossos programas.
- Um mesmo TAD pode ser implementado por diferentes estruturas de dados, sem alterações na interface e/ou nos programas clientes.
- Listas encadeadas são estruturas de dados lineares não contíguas na memória, onde o encadeamento é dado por ponteiros.
- LE podem ser simples ou duplas, circulares ou não.
- Existe um compromisso entre tempo e espaço nas estruturas de dados.