
# Prática Profissional II – Linguagem de Programação Estruturada

Curso: Análise e Desenvolvimento de Sistemas
Modalidade: Presencial
Professor Esp. Wesley Tschiedel
Email: wesley.tschiedel@ucb.br

# 7. Representação dos dados na memória: Alocação estática e Dinâmica.
# 8. Gerenciamento de memória em tempo de execução.
# Exercícios de fixação.

Podemos definir **lista encadeada** como um recurso através do qual um conjunto de blocos de memória é mantido numa ordem lógica, mas não necessariamente física.

Para manter a **ordem linear** em um conjunto de blocos espalhados por todo o espaço de memória, cada bloco deve **armazenar** o **endereço** do **bloco seguinte**.

Cada item na lista é chamada **nó** e contém dois campos, um campo de **informação** e um campo do **endereço seguinte**.

O **campo** de **informação** armazena o real **elemento** da lista.

O **campo** do **endereço** seguinte **contém** o **endereço** do **próximo nó** na lista.

O endereço, que é usado para acessar determinado nó, é conhecido como **ponteiro**.

A lista encadeada inteira é acessada a partir de um **ponteiro externo** lista que aponta para (contém o endereço de) o primeiro nó na lista.

Por ponteiro **“externo”**, entendemos aquele que não está incluído dentro de um nó.

Seu valor pode ser acessado diretamente, por referência a uma variável.

O campo do próximo endereço do último nó na lista contém um valor especial, conhecido como **NULL**, que não é um endereço válido.

Esse **ponteiro nulo** (ou **NULL**) é usado para indicar o final de uma lista.

A lista sem nós é chamada **lista vazia** ou **lista nula**.

O valor do ponteiro externo lista para esta lista é o ponteiro nulo.

Uma lista pode ser inicializada com uma lista vazia pela operação **list = null**.

Se **p** é um ponteiro para um nó, **node(p)** refere-se ao nó apontado por **p**, **info(p)** refere-se à parte da informação desse nó e **next(p)** refere-se à parte do endereço seguinte e é, portanto, um ponteiro.

Sendo assim, se **next(p)** não for **NULL**, **info(next(p))** se referirá à parte da informação do nó posterior a **node(p)** na lista.

# INSERINDO E REMOVENDO NÓS DE UMA LISTA

Uma **lista** é uma **estrutura** de dados **dinâmica**.

O **número de nós** de uma lista pode **variar** consideravelmente à medida que são **inseridos** e **removidos** elementos.

A **natureza dinâmica** de uma lista pode ser **comparada** à **natureza estática** de um vetor, cujo tamanho permanece constante.

Vamos supor que tenhamos uma lista de inteiros, e queiramos incluir o **inteiro 6** no **início da lista**.

O primeiro passo é obter um nó para armazenar o inteiro adicional.

Se uma lista precisa crescer e diminuir, é necessário um mecanismo para obter nós vazios a ser incluídos na lista.

Ao contrário de um vetor, uma lista não vem com um conjunto pré-fornecido de locais de armazenamento nos quais podem ser colocados elementos.

Para isso é necessário alocar memória dinamicamente.

O uso de **alocação dinâmica** torna-se vantajoso quando temos a necessidade de armazenar uma quantidade indeterminada de elementos.

Uma coleção cujo tamanho pode variar durante a sua existência.

Em C, uma variável ponteiro para um inteiro pode ser criada pela declaração:

```
int *p;
```

Assim que uma variável **p** for declarada como um ponteiro para um tipo de objeto específico, será possível criar dinamicamente um objeto desse tipo específico e atribuir seu endereço a **p**.

Podemos realizar essa alocação chamando a função **malloc(size).**

A função **malloc()** aloca dinamicamente uma parte da memória, de tamanho **size**, e retorna um ponteiro para um item do tipo **char**.

Veja as declarações:

```
extern char *malloc();
int *pi;
float *pr;
```

comandos:

```
pi = (int *) malloc (sizeof (int));
pr = (float *) malloc (sizeof (float));
```

Criam dinamicamente a variável inteira ***pi** e a variável flutuante ***pr**.

Essas variáveis são chamadas variáveis dinâmicas.

Ao executar esses comandos, o operador **sizeof** retorna o tamanho, em bytes, de seu operando. **malloc** poderá, então, criar um objeto desse tamanho.

Assim, **malloc(sizeof(int))** aloca armazenamento para um inteiro.

**malloc(sizeof(float))** aloca armazenamento para um número de ponto flutuante.

**malloc** retorna também um ponteiro para armazenamento que ela aloca.

Esse ponteiro serve para o primeiro byte (por exemplo, caractere) desse armazenamento e é do tipo **char *.**

Para forçar esse ponteiro a apontar para um inteiro ou real, usamos o operador de conversão (**int ***) ou (**float ***).

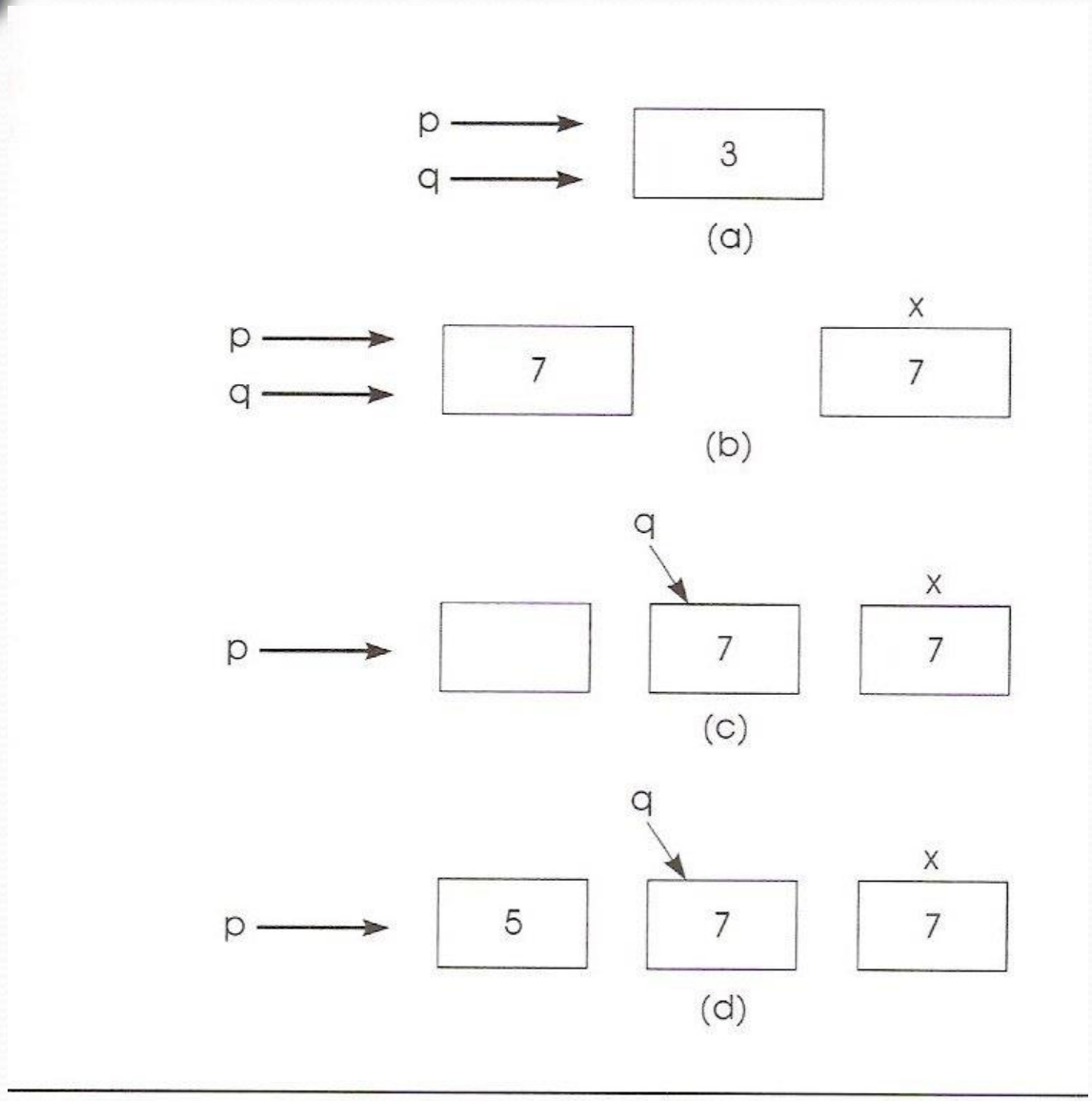
O operador **sizeof** retorna um valor do tipo **int**, enquanto a função **malloc** espera um parâmetro de tipo **unsigned**. Podemos escrever da seguinte forma:

```
pi = (int *) malloc ((unsigned) (sizeof(int)));
```

Entretanto, a conversão sobre o operador **sizeof** é frequentemente omitida.

# EXEMPLO

```
1.  main()
2.  {
3.      int *p, *q;
4.      int x;
5.      p = (int *) malloc(sizeof(int));
6.      *p = 3;
7.      q = p;
8.      printf(" %d %d \n", *p, *q);
9.      x = 7;
10.     *q = x;
11.     printf(" %d %d \n", *p, *q);
12.     p = (int *) malloc(sizeof(int));
13.     *p = 5;
14.     printf(" %d %d \n", *p, *q);
15.     }
```


p
q
3
(a)
p
q
7
x
7
(b)
q
p
7
x
7
(c)
q
p
5
7
x
7
(d)

A função **free** é usada em C para liberar o armazenamento de uma variável alocada dinamicamente.

**free(p);**

invalida quaisquer referências futuras a ***p** (a menos, evidentemente, que um novo valor seja atribuído a **p** por um comando de atribuição ou por uma chamada a **malloc**).

Chamar **free(p)** torna o armazenamento ocupado por ***p** disponível para reutilização, se necessário.

A função **free** espera um parâmetro ponteiro do tipo **char ***.

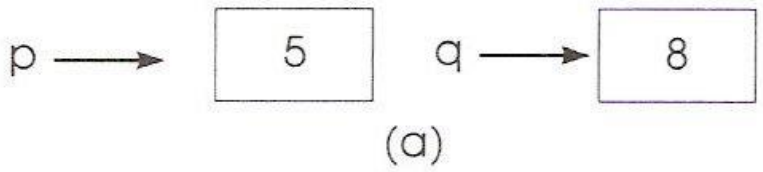
Para que o comando fique “**limpo**”, devemos escrever:

```
free((char *) p);
```

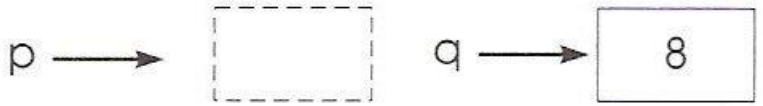
Entretanto, na prática, a conversão do parâmetro é frequentemente omitida.

# EXEMPLO

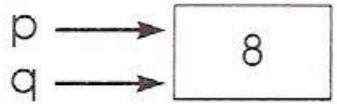
```
1.  main()
2.  {
3.      int *p, *q;
4.      p = (int *) malloc(sizeof(int));
5.      *p = 5;
6.      q = (int *) malloc(sizeof(int));
7.      *q = 8;
8.      free(p);
9.      p = q;
10.     q = (int *) malloc(sizeof(int));
11.     *q = 6;
12.     printf(" %d %d \n", *p, *q);
13.
14.}
```



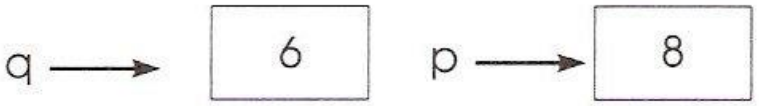

(a)




(b)




(c)




(d)

Para criar elementos para uma lista vamos pressupor a existência de um mecanismo para obter nós vazios. A operação:

**p = lst_cria( );**

Está operação obtém um nó vazio e define o conteúdo de uma variável chamada **p** com o endereço desse nó. Dessa forma, o valor de **p** é um ponteiro para esse nó recém alocado.

# EXEMPLO

```
1.  #include<stdio.h>
2.  #include<stdlib.h>

3.  struct lista{
4.          int info;
5.          struct lista* prox;
6.          };
7.  typedef struct lista Lista;

8.  Lista* lst_cria(void);
9.  Lista* lst_insere(Lista* l, int i);
10. void lst_imprime(Lista* l);
```

```
11. main()
12. {
13.       Lista* l; //declara uma lista não inicializada
14.       l=lst_cria(); //cria e inicializa lista como vazia
15.       l=lst_insere(l, 23); //insere na lista o elemento 23
16.       l=lst_insere(l, 45); //insere na lista o elemento 45
17.       lst_imprime(l);
18.       system("pause");
19.       return 0;
20.       }
21.
22. /*função de criação: retorna lista vazia*/
23. Lista* lst_cria(void)
24. {
25.        return NULL;
26.       }
```

```
27. /*inserção no início: retorna a lista atualizada*/
28. Lista* lst_insere(Lista* l, int i)
29. {
30.       Lista* novo = (Lista*)malloc(sizeof(Lista));
31.       novo->info=i;
32.       novo->prox=l;
33.       return novo;
34.       }

35. /* função imprime: imprime valores dos elementos*/
36. void lst_imprime(Lista* l)
37. {
38.     Lista* p; //variável auxiliar para percorrer a lista
39.     for(p=l; p!=NULL; p=p->prox)
40.         printf("Info = %d, End do prox = %x\n", p->info, p->prox);
41.     }
```

# Atividade Prática

# Referências Bibliográficas

**Básica:**


- EVARISTO, J.,**Aprendendo a programar programando em C**, Book Express, 2001, 205p.
- MIZRAHI, V. V., **Treinamento em Linguagem C**, Módulo 1 e 2, Makron Books do Brasil Editora Ltda, 1990, 273 p.
- SCHILDT, H., **C Completo e Total**, Editora Makron Books doBrasil Editora Ltda, 1997, 827p.

# Referências Bibliográficas

**Complementar:**


- DEITEL, H. M. e Deitel, P. J.,**C++ Como Programar**, 3. ed. Porto Alegre: Artmed Editora S.A, 2001. 1098 p.

- MANZANO, J. A. N. G. **Estudo Dirigido: Linguagem C.** 6. ed. São Paulo: Érica, 2002.

- SOFFNER, Renato. **Algoritmos e programação em linguagem C**. São Paulo Saraiva 2013.

- TENENBAUM, A. M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estruturas de Dados Usando C.** São Paulo: Makron Books, 1995.

- ZIVIANI, Nivio. **Projeto de algoritmos: com implementações em Pascal e C.** 3. ed., rev. e ampl. São Paulo, SP: Cencage Learning, 2015. xx, 639 p.