



Estrutura de Dados



```
var assets = dataModel.assets;
// Loop through all the display elements
assets.forEach(function(asset) {
    asset.html_.innerHTML += 'Selected assets' + ' > ' + collectAssetsData.collectAssets.assets.length;
});
// Add assets to a collection. These assets then appear in the right box on the right side of the builder. Use this
// function always when adding assets to the collection. It adds all elements in one
// array to a collection. This function also updates the collection id to match the assets
function addCollectionOfAssets(asset) {
    var collection = collectAssetsData.collectAssets;
    collection.push(asset);
    collection.forEach(function(asset) {
        asset.html_.innerHTML += 'Selected assets' + ' > ' + collectAssetsData.collectAssets.assets.length;
    });
}
// Load the assets from the lists and render them again
collectAssetsData.collectAssets.length = 0;
// Load the assets from the lists and render them again
collectAssetsData.collectAssets.length = 0;
```

LISTAS LINEARES

APRESENTAÇÃO

O lá aluno(a), você já conhece os vetores, sabe que seu conceito é de uma junção de duas ou mais variáveis do mesmo tipo, que possuem o mesmo nome e endereços de memórias sequenciais. Mas já se perguntou se seria possível ter dados similares com endereços não sequenciais pertencentes a mesma estrutura?

Pois este, é exatamente a forma de funcionamento das listas lineares. E neste módulo explicarei tudo sobre esse assunto.

OBJETIVOS DE APRENDIZAGEM

Ao final desse módulo você deverá ser capaz de:

- Diferenciar vetor de listas lineares;
- Compreender o funcionamento das listas lineares;
- Conhecer listas lineares simples e duplas.

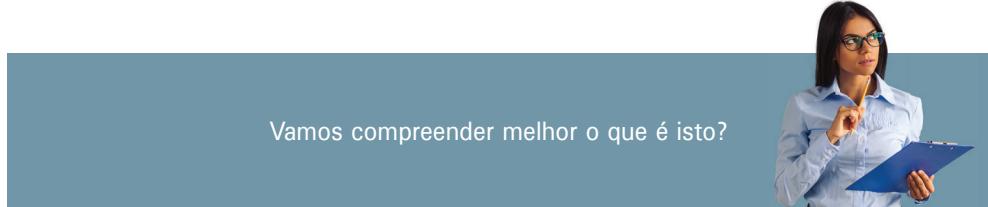
LISTAS LINEARES

Introdução

Primeiro vamos conceituar uma lista, para assim podermos compreender sua aplicação: Uma lista corresponde a uma estrutura que efetua armazenamento linear, ou seja, seus elementos estão ordenados de uma forma sequencial, por isso, para acessar qualquer item temos que seguir um a um (linearmente).

Utilizamos listas para facilitar a manipulação das informações, inserindo elementos que seguem ordenação e prioridade. Cada item de uma lista é conhecido como nó.

Um item bastante relevante é que você saiba que as listas podem ter ligação lógica ou física, entre seus nós.



Vamos compreender melhor o que é isto?

As propriedades de uma lista consideram que seus elementos estruturais seguem a seguinte ordem:

Primeiro Elemento	Segundo Elemento	Terceiro Elemento	...	Enésimo Elemento
x_1	x_2	x_3	...	x_n

RECAPITULANDO

E, neste caso, n é o tamanho da lista; x_1 é o primeiro elemento x_n o último elemento. Lembre-se que as posições são sequenciais, ou seja, x_i é precedido de x_{i-1} e sucedido por x_{i+1} . Sei que parece óbvio mas não podemos esquecer!!!



Ao aplicarmos estas estruturas em códigos, elas poderão ser criadas de duas formas distintas.

TIPO DE LISTAS	
Estática (Vetores)	Dinâmica (Ponteiros)

A forma como será implementada depende das necessidades de seu código, por isso é importante que você conheça bem as características, vantagens e desvantagens de cada tipo. Além disto, algumas operações específicas nos permitem manipular estas listas, e essas ações existirão em ambos tipos:

1. **Criação da lista:** Criar uma lista vazia;
2. **Busca:** Efetuar busca de um elemento na lista;
3. **Inclusão:** Incluir elementos na lista (também conhecido como PUSH);
4. **Remoção:** retirar elementos na lista (também conhecido como POP);
5. **Verificar vazio:** Verificar se a lista está vazia.

Lista linear estática

Listas estáticas são os já conhecidos vetores (ou *arrays*), e para este tipo de lista, temos que nos lembrar que os endereços de memória de cada elemento do vetor são sequenciais, ou seja, estes elementos são fisicamente ligados na memória.

Ao inserir em um vetor, é importante se lembrar que será necessário atender às exigências de inserir em cada índice do vetor, seguindo a ordem e, principalmente, em caso de exclusão, realocar as posições para que não haja espaços vazios dentro do vetor.

Exemplo de inserção em uma lista estática (vetor), para este exemplo utilizaremos uma *struct* com os seguintes dados:

```
1 typedef struct info informacoes;
2 struct info{
3     int codigo;
4     char nome[40];
5     int idade;
6     float salario;
7 }
```



Neste caso, como é um vetor, a criação pode ser feita de forma estática ou dinâmica, mas sempre considerando a quantidade definida, ou seja, uma vez definido o tamanho do vetor, não teria como redimensionar, veja como seria o exemplo de um dimensionamento de 4 posições!

```
1 //Declaração estática
2 informacoes vetorEstatico[4];
3
4 //OU
5
6 //Declaração dinâmica
7 informacoes *vetorDinamico;
8 vetorDinamico = (informacoes*) malloc (4 * sizeof(informacoes));
```

E para cada item do vetor faremos uma inserção destas informações, veja o exemplo desta inserção:

índice 0	índice 1	índice 2	índice 3
187	254	149	925
Sandra Oliveira	Roberto Flores	Paula Silva	Bernardo Lima
19	27	34	54
5290.87	2589.98	2123.34	4300.23
Endereço 0x17d6010	Endereço 0x17d6044	Endereço 0x17d6078	Endereço 0x17d60ac



O código da inserção seria assim, sempre “obedecendo” o índice do vetor, começando do zero, mas saiba que o exemplo foi criado apenas para que seja possível perceber como aconteceria a alteração do índice do vetor, portanto, não há validação de entrada de dados:

```
1 void inserirDadosFuncionario(informacoes *funcionario, int quantidade){  
2     if(quantidade > 0) {  
3         //Inserir código  
4         printf("\nCódigo do funcionário:");  
5         //Gerar código randômico de 0 até 100  
6         funcionario->codigo = rand() % 100;  
7  
8         //Nome  
9         printf("\nNome do funcionário: ");  
10        fgets(funcionario->nome, 40, stdin);  
11  
12         //Idade - Não haverá validação  
13         printf("\nIdade do funcionário: ");  
14         scanf("%i", &funcionario->idade);  
15  
16         //Salário - Não haverá validação  
17         printf("\nSalário R$");  
18         scanf("%f", &funcionario->salario);  
19  
20         //Chamada recursiva para uma nova inserção  
21         getchar();  
22         inserirDadosFuncionario(funcionario+1, quantidade-1);  
23 }
```

BUSCA EM UMA LISTA LINEAR ESTÁTICA

Para efetuar uma busca em uma lista linear estática (vetor), é necessário procurar, índice por índice, até encontrar o elemento procurado. O código a seguir demonstra uma busca pelo código do funcionário:

```
1 informacoes*buscar(informacoes *funcionario, int quantidade, int cod){  
2     if (funcionario->codigo == cod) {  
3         //Registro encontrado, retorna o endereço de memória  
4         return funcionario;  
5     } else if (quantidade > 0) {  
6         //Registro não encontrado, continua a busca  
7         buscar(funcionario+1, quantidade-1, cod);  
8     } else {  
9         //Registro não encontrado, vetor finalizado  
10        return NULL;  
11    }  
12 }
```

Se utilizando este código solicitássemos a busca do código 149, este código nos retornaria o endereço de memória 0x17d6078.

REMOÇÃO EM UMA LISTA LINEAR ESTÁTICA

Ao remover um registro de uma lista estática é importante se lembrar que, não podemos deixar espaços vazios, portanto, temos que reorganizar todo o vetor, isto, inclusive, é uma desvantagem de utilizarmos vetores para este tipo de inserção, afinal, o custo da realocação dos registros será alto.

Veja um exemplo de remoção, para a busca do registro, utilizarei o código anterior!



```

1 void removerRegistro (informacoes *funcionario, int quantidade, int cod){
2     //Ponteiro temporário para armazenar o primeiro registro
3     informacoes *tmp = funcionario;
4     //Busca o endereço do registro que será excluído
5     funcionario = buscarRegistro(funcionario, quantidade, cod);
6
7     if (funcionario == NULL) { //Registro não existe no vetor.
8         printf("\nRegistro não localizado.");
9     } else { //Registro localizado
10        //Descobrir o índice que será excluído
11        int i = funcionario - tmp;
12        //Apontar para o próximo endereço de memória
13        tmp = funcionario + 1;
14        while (i < (quantidade-1)) { //Realocar as posições
15            funcionario->codigo = tmp->codigo;
16            strcpy(funcionario->nome, tmp->nome);
17            funcionario->idade = tmp->idade;
18            funcionario->salario = tmp->salario;
19
20            //Alterar elementos para preencher o próximo registro
21            funcionario++;
22            tmp++;
23            i++;
24        }
25
26        //Apagar informações
27        funcionario->codigo = -1;
28        strcpy(funcionario->nome, "");
29        funcionario->idade = 0;
30        funcionario->salario = 0;
31    }
32 }

```

Se solicitarmos a remoção do código 149, o resultado final seria:

índice 0	índice 1	índice 2	índice 3
187 Sandra Oliveira 19 5290.87	254 Roberto Flores 27 2589.98	925 Bernardo Lima 54 4300.23	-1 0 0

Endereço 0x17d6010 Endereço 0x17d6044 Endereço 0x17d6078 Endereço 0x17d60ac



Essa foi fácil né? Afinal, você já conhece o funcionamento dos vetores, e sabe que os registros possuem uma ligação física na memória, pois seus endereços são sequenciais. Agora vamos compreender o que é, e como funciona, uma lista linear dinâmica!



Lista linear dinâmica

Também conhecida como lista linear encadeada, essa lista mantém uma ligação por apontamento, por isso, não são sequenciais na memória, neste caso, possuem ligações lógicas entre seus registros. Cada nó armazenará o endereço de memória do próximo, e, eventualmente, do anterior.

Uma vantagem de se optar por essa estratégia é que suas ações de inserção e remoção serão mais fáceis, e, além disto, uma lista dinâmica não precisa ter um tamanho previamente definido.

Mas, uma desvantagem, é que seus registros sempre precisarão de um espaço de memória para os apontamentos.

As listas dinâmicas podem ser de dois tipos:

TIPO DE LISTAS DINÂMICAS	
Simples (Aponta para o próximo registro)	Dupla (Aponta para o registro anterior e para o próximo)

As operações, citadas anteriormente, podem ser aplicadas em ambas. Vamos compreender como essas operações ocorrem em cada tipo, começando pela lista simplesmente encadeada.



LISTA LINEAR SIMPLESMENTE ENCADEADA

Dizemos que uma lista é simplesmente encadeada quando ela possui um apontamento para o próximo registro, ou seja, cada registro sempre armazenará o endereço de memória do registro que virá depois dele, afinal, eles não são sequenciais na memória, por isso a necessidade de armazenar este endereço.

Para este tipo de lista, sempre existirá uma estrutura com todos os campos necessários para registro, e, além destes campos, haverá um ponteiro para o próximo elemento, ou seja, para o próximo espaço de memória criado na lista.

Observe o exemplo de criação de estrutura:

```
1 typedef struct info listaSimples;
2 struct info{
3     //Dados básicos do registro
4     int codigo;
5     char nome[40];
6     int idade;
7     float salario;
8
9     //Ponteiro para o próximo registro
10    listaSimples *proxima;
11 }
```

Para iniciar nosso código devemos criar uma lista com este modelo, neste caso, o correto seria iniciá-la com o valor nulo, garantindo que ainda não existe nada dentro dela, o código seria assim:

```
1 listaSimples *lista = NULL;
```



INSERÇÃO EM UMA LISTA LINEAR SIMPLESMENTE ENCADEADA

Uma ação muito importante é a de inserção, saiba que em uma lista encadeada, simples ou dupla, pode-se ter dois tipos de inserção:

FORMAS DE INSERÇÃO EM UMA LISTA ENCADEADA	
Pelo início (Inserção pela cabeça)	Pelo fim (Inserção pela cauda)

Inserção pelo início em uma lista simplesmente encadeada

Nosso primeiro exemplo será uma inserção pelo início da lista, ou seja, pela cabeça. Neste conceito, os registros novos sempre serão inseridos no início da lista.

Vamos começar pelo código utilizado neste tipo de inserção:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 #define tamanho 100
5
6 struct Funcionario {
7     int codigo;
8     char nome[40];
9     int idade;
10    float salario;
11 }
12
13 struct listaSimples {
14     struct Funcionario *ultimo;
15     struct listaSimples *proxima;
16 };
17
18 listaSimples* inserirInicio(listaSimples *lista) {
19     //Alocar novo espaço
20     listaSimples *novo = (listaSimples*) malloc (sizeof(listaSimples));
21     //Preencher o campos básicos
22     //Código - gerado aleatoriamente de 0 até 100
23     novo->codigo = rand() % 100;
24
25     //Nome
26     printf("\nNome do funcionário: ");
27     fgets(novo->nome, 40, stdin);
28
29     //Idade - Não haverá validação
30     printf("\nIdade do funcionário: ");
31     scanf ("%i", &novo->idade);
32
33     //Salário - Não haverá validação
34     printf("\nSalário R$");
35     scanf ("%f", &novo->salario);
36
37     //Ajustar ponteiro para que este registro seja o primeiro da lista
38     novo->proxima = lista;
39
40     //Retornar o endereço de memória do novo
41     return novo;
42 }
```

OBSERVAÇÃO

A linha 21 do código faz com que o registro novo aponte para o primeiro registro da lista, tornando-o o primeiro.



Veja o que aconteceria após uma inclusão, lembrando que a lista estava vazia, ou seja, a lista foi criada como NULL.

codigo 41
Nome Gabrielle
Idade 20
Salário 2500.93
proxímo NULL

endereço do item
0x14ec010



Neste caso o registro incluído é único da lista, visto que ela estava vazia. Então, após a inclusão, a lista apontaria para o endereço 0x14ec010, indicando que este é o primeiro. O campo próximo do registro incluído está apontando para NULL indicando que não existem registros após este.

Agora veremos como seria após a segunda inserção, obviamente executando as linhas de código anterior novamente.

codigo 215	codigo 41
Nome José	Nome Gabrielle
Idade 70	Idade 20
Salário 1299.43	Salário 2500.93
proxímo 0x14ec010	proxímo NULL
endereço do item	endereço do item
0x14ec030	0x14ec010

No nosso exemplo, a inserção é pelo início da lista, portanto, o segundo registro incluído passa a ser o primeiro. Observe que o primeiro registro incluído possui endereço de memória 0x14ec010, e o campo próximo, do segundo registro incluído, passou a apontar para este endereço, guardando-o. Portanto, mesmo que os endereços destes registros não sejam sequenciais, é possível chegar no local onde cada item está alocado, através do apontamento. Neste ponto do código, a lista estaria com o endereço de memória 0x14ec030, indicando que ele é o primeiro registro.

Vamos ver como seria a terceira inserção!

codigo 523	codigo 215	codigo 41
Nome Eder	Nome José	Nome Gabrielle
Idade 34	Idade 70	Idade 20
Salário 815.32	Salário 1299.43	Salário 2500.93
proxímo 0x14ec030	proxímo 0x14ec010	proxímo NULL
endereço do item	endereço do item	endereço do item
0x14ec050	0x14ec030	0x14ec010

Veja que segue sempre a mesma lógica, o novo registro incluído passa a ser a primeira da lista, e o campo próximo, de cada registro, aponta para o endereço de memória do registro seguinte, permitindo assim localizá-lo.

A lista sempre terá apenas o endereço de memória do primeiro registro, neste caso ela estaria com o endereço 0x14ec050, para indicar onde ela se inicia.



INSERÇÃO PELO FIM EM UMA LISTA SIMPLESMENTE ENCADEADA

Outra forma de inserção é pelo fim da lista, ou seja, pela cauda. Neste conceito, os registros novos ficarão após todos os que já foram inseridos anteriormente.

Observe o código referente a este tipo de inserção!

```
1 listaSimples* inserirFim(listaSimples *lista) {
2     //Alocar novo espaço
3     listaSimples *novo = (listaSimples*) malloc (sizeof(listaSimples));
4     //Preencher os campos básicos
5     //Código - gerado aleatoriamente de 0 até 100
6     novo->codigo = rand() % 100;
7
8     //Nome
9     printf("\nNome do funcionário: ");
10    fgets(novo->nome, 40, stdin);
11
12    //Idade - Não haverá validação
13    printf("\nIdade do funcionário: ");
14    scanf ("%i", &novo->idade);
15
16    //Salário - Não haverá validação
17    printf("\nSalário R$");
18    scanf ("%f", &novo->salario);
19
20    //Ajustar ponteiro para que este registro seja o último da lista
21    novo->proximo = NULL;
22
23    //Conectar o novo registro à lista
24    if (lista == NULL){
25        //Se a lista for nula ele será o primeiro
26        return novo;
27    } else {
28        //Se a lista não for nula precisa encontrar o último
29        listaSimples *tmp = lista;
30        //Encontrar o registro que aponta para null
31        while (tmp->proximo != NULL) {
32            tmp = tmp->proximo;
33        }
34        //Apontar o último para o novo, tornando-o o penúltimo
35        tmp->proximo = novo;
36    }
37
38 }
```

IMPORTANTE

Um conceito muito importante é que o último registro inserido sempre terá seu campo próximo apontando para NULL, indicando que não existem registros após ele.



Um código alternativo a esse seria sempre armazenar o endereço de memória do último registro inserido, assim, não seria necessário efetuar um loop para procurar. A ideia é ter uma variável para armazenar o endereço de memória do último registro inserido, assim, sempre que este último for alterado, o código apenas manteria a variável atualizada, com o endereço de memória dele.



Veja o que aconteceria após uma inclusão, considerando que a lista estava vazia, ou seja, que a lista foi criada como NULL:

codigo 41
Nome Gabrielle
Idade 20
Salário 2500.93
proxímo NULL
endereço do item
0x14ec010

Agora veremos como seria após a segunda inserção, no nosso exemplo, a inserção é pelo fim da lista, portanto, o novo registro será o último:

codigo 41	codigo 215
Nome Gabrielle	Nome José
Idade 20	Idade 70
Salário 2500.93	Salário 1299.43
proxímo 0x14ec030	proxímo NULL
endereço do item	endereço do item
0x14ec010	0x14ec030

OBSERVAÇÃO

Observe que o segundo registro incluído possui endereço de memória 0x14ec030, e o campo próximo, do primeiro registro incluído, passou a apontar para este endereço, guardando-o. Portanto, mesmo que os endereços destes registros não seja sequenciais, é possível chegar ao local onde cada item está alocado, através do apontamento.



Vamos ver como seria a terceira inserção:

codigo 41	codigo 215	codigo 523
Nome Gabrielle	Nome José	Nome Eder
Idade 20	Idade 70	Idade 34
Salário 2500.93	Salário 1299.43	Salário 815.32
proxímo 0x14ec030	proxímo 0x14ec050	proxímo NULL
endereço do item	endereço do item	endereço do item
0x14ec010	0x14ec030	0x14ec050

OBSERVAÇÃO

As próximas inserções seguirão a mesma lógica, você pode fazer quantas precisar, uma lista não possui limite de inserção.



BUSCA EM UMA LISTA SIMPLESMENTE ENCADEADA

Para efetuarmos uma busca (pesquisa) em uma lista simplesmente encadeada, temos que ter o identificador do registro, então, iremos de registro em registro procurando, neste caso, de espaço de memória em espaço de memória, sempre utilizando o campo próximo para nos guiar.

O campo código será o identificador deste registro. A função receberá, além da lista, o código a ser localizado passados por parâmetro.

```
1 listaSimples* buscarRegistro(listaSimples *lista, int cod, listaSimples *  
2     listaSimples *atual = lista;  
3     *anterior = NULL;  
4  
5     //Procurar registro  
6     while (atual != NULL) {  
7         if (atual->codigo == cod) {  
8             //Se encontrar retorna o endereço de memória  
9             return atual;  
10        }  
11  
12        //Se não encontrar procura no próximo  
13        *anterior = atual;  
14        atual = atual->proximo;  
15    }  
16    //Se não encontrar retorna nulo  
17    *anterior = NULL;  
18    return NULL;  
19 }
```

OBSERVAÇÃO

No código, o ponteiro anterior recebe o endereço de memória do registro que aponta para aquele que será excluído, e a função retorna o endereço de memória do registro que será excluído, tendo assim acesso aos dois endereços.

Por isso, se efetuarmos a busca do código 215, a função nos retornaria o endereço de memória 0x14ec030 e preencheria o ponteiro anterior com o endereço 0x14ec050.



EXCLUSÃO EM UMA LISTA SIMPLESMENTE ENCADEADA

Para excluirmos um registro em uma lista simplesmente encadeada, é necessário repensar os apontamentos efetuados nela, ou seja, é necessário realocar os ponteiros pois algum espaço de memória será apagado.

Sabemos que o comando para apagar um espaço de memória é o *free*, porém, antes de aplicá-lo, devemos realocar os apontamentos necessários, para que a lista não fique “desconectada”. Além disto, também é importante se lembrar que, para excluir um registro, teremos que procurá-lo primeiro, por isso, pode-se utilizar a função anterior para que ele seja localizado.



```

1 listaSimples* excluirRegistro (listaSimples *lista, int cod) {
2     listaSimples *atual = lista,
3     *anterior = NULL;
4
5     //Procurar registro que será excluído
6     atual = buscarRegistro(lista, cod, &anterior);
7     if (atual != NULL) { //Encontrei o registro
8         if (anterior == NULL) { //É o primeiro registro da lista
9             //O segundo registro da lista passará a ser o primeiro
10            lista = lista->proximo;
11        } else { //É o registro do meio ou o último
12            anterior->proximo = atual->proximo;
13        }
14        //Apagar o registro da memória
15        free(atual);
16    } else {
17        printf("\nRegistro não encontrado");
18    }
19    //Retornar a lista, atualizada ou não.
20    return lista;
21 }
```

Como o endereço de memória que está no ponteiro atual será excluído da lista, então, na linha 12, o registro anterior ao atual passa a apontar para onde o atual aponta, mantendo a ligação da lista. E, para os casos onde o primeiro registro será o excluído, a linha 10 nos mostra que a lista, que possui o primeiro registro, passará a apontar para o segundo, assim, quando ela for retornada (linha 20) terá o endereço correto.



Agora vamos simular como seria a exclusão do registro de código 215, da lista. Como referência utilizaremos a simulação da lista encadeada pelo fim, que possui a seguinte configuração:

codigo 41	codigo 215	codigo 523
Nome Gabrielle	Nome José	Nome Eder
Idade 20	Idade 70	Idade 34
Salário 2500.93	Salário 1299.43	Salário 815.32
proxímo 0x14ec030	proxímo 0x14ec050	proxímo NULL
endereço do item	endereço do item	endereço do item
0x14ec010	0x14ec030	0x14ec050

Após a exclusão, e o reajuste dos ponteiros, a configuração da lista seria assim:

codigo 41	codigo 523
Nome Gabrielle	Nome Eder
Idade 20	Idade 34
Salário 2500.93	Salário 815.32
proxímo 0x14ec050	proxímo NULL
endereço do item	endereço do item
0x14ec010	0x14ec050



IMPORTANTE

Saiba que é sempre importante armazenar onde a lista começa, ou seja, qual o endereço de memória do início da lista, e a partir daí, uma vez localizado o registro a ser excluído, reajustar os apontamentos.



Se por acaso, o primeiro registro for excluído, a lista passará a armazenar o endereço do segundo registro, indicando seu “novo” início. A exclusão poderá ocorrer para qualquer registro, inclusive, se necessário, a lista poderá voltar a ser vazia (nula).

Bem, assim, consideramos todas as ações necessárias, e possíveis, em uma lista simplesmente encadeada. A partir de agora veremos o outro tipo de lista, e aplicaremos as mesmas ações, porém, utilizando as características dela.



LISTA LINEAR DUPLAMENTE ENCADEADA

Dizemos que uma lista é duplamente encadeada quando ela possui um apontamento para o próximo registro e também um apontamento para o registro anterior.

Para este tipo de lista, sempre existirá uma estrutura com todos os campos necessários para registro, e, além destes campos, cada registro terá uma ligação no elemento anterior e posterior a ele.

Observe o exemplo de criação de estrutura:

```
1 typedef struct info listaDupla;
2 struct info{
3     //Dados básicos do registro
4     int codigo;
5     char nome[40];
6     int idade;
7     float salario;
8
9     //Ponteiros
10    listaDupla *anterior;
11    listaDupla *proxima;
12 }
```

Para criar um lista com este modelo utilize, sempre indicando-a como vazia (nula) para garantir que não haverão erros:

```
1 listaDupla *lista = NULL;
```



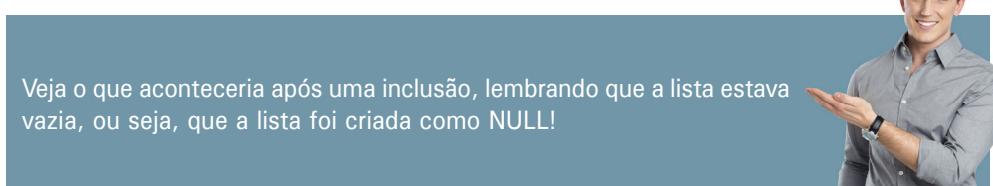
INSERÇÃO PELO INÍCIO EM UMA LISTA DUPLAMENTE ENCADEADA

Como você já sabe, neste conceito, os registros novos sempre serão inseridos no início da lista.

Vamos começar pelo código utilizado neste tipo de inserção:

```
1 listaDupla* inserirInicio(listaDupla *lista) {
2     //Alocar novo espaço
3     listaDupla *novo = (listaDupla*) malloc (sizeof(listaDupla));
4     //Preencher os campos básicos
5     //Código - gerado aleatoriamente de 0 até 100
6     novo->codigo = rand() % 100;
7
8     //Nome
9     printf("\nNome do funcionário: ");
10    fgets(novo->nome, 40, stdin);
11
12    //Idade - Não haverá validação
13    printf("\nIdade do funcionário: ");
14    scanf ("%i", &novo->idade);
15
16    //Salário - Não haverá validação
17    printf("\nSalário R$");
18    scanf ("%f", &novo->salario);
19
20    //Ajustar ponteiro para que este registro seja o primeiro da lista
21    novo->anterior = NULL; //Antes dele não tem nenhum
22    novo->proximo = lista;
23    if (lista != NULL) { //Se já existir algum registro na lista
24        lista->anterior = novo;
25    }
26
27    //Retornar o endereço de memória do novo
28    return novo;
29}
```

A linha 21 do código “limpa” o ponteiro anterior do novo registro, afinal, antes dele não haverá outro, ele será o primeiro. A linha 22 do código faz com que o registro novo aponte para o primeiro registro da lista, tornando-o o primeiro. E, se já existir algum registro na lista, o anterior do primeiro registro apontará para o novo (linha 24), indicando quem vem antes dele, a partir de agora.



Veja o que aconteceria após uma inclusão, lembrando que a lista estava vazia, ou seja, que a lista foi criada como NULL!

codigo 41

Nome Gabrielle

Idade 20

Salário 2500.93

anterior NULL

proxímo NULL

endereço do item

0x14ec010

Neste caso o registro incluído é único da lista, visto que ela estava vazia, então, após a inclusão, a lista apontaria para o endereço 0x14ec010, indicando que este é o primeiro. O campo próximo, e o campo anterior, do registro incluído estão apontando para NULL indicando que não existem registros antes ou depois deste.

Agora veremos como seria a segunda inserção, obviamente executando as linhas de código anterior novamente.

codigo 215	codigo 41
Nome José	Nome Gabrielle
Idade 70	Idade 20
Salário 1299.43	Salário 2500.93
anterior NULL	anterior 0x14ec030
proxímo 0x14ec010	proxímo NULL
endereço do item	endereço do item
0x14ec030	0x14ec010

Veja que os ponteiros, anterior e próximo, foram devidamente preenchidos, garantindo a ligação lógica da lista, e permitindo que os registros sejam localizados na memória. A lista fica com o endereço de memória 0x14ec030, indicando que ele é o primeiro registro.

Vamos ver como seria a terceira inserção:

codigo 523	codigo 215	codigo 41
Nome Eder	Nome José	Nome Gabrielle
Idade 34	Idade 70	Idade 20
Salário 815.32	Salário 1299.43	Salário 2500.93
anterior NULL	anterior 0x14ec050	anterior 0x14ec030
proxímo 0x14ec030	proxímo 0x14ec010	proxímo NULL
endereço do item	endereço do item	endereço do item
0x14ec050	0x14ec030	0x14ec010



Então, você poderá inserir quantos registros precisar, sem dimensionar previamente, pois o novo registro incluído passa a ser o primeiro da lista, e o campo próximo, e anterior, de cada registro, garantem que será possível localizá-los.

IMPORTANTE

Observe que os endereços de memória de cada registro são independentes, não são sequenciais, por isso é importante manter o apontamento, assim é possível saber quem virá antes e depois dele. Neste ponto do código a lista estará com o endereço 0x14ec050, indicando seu início.



INSERÇÃO PELO FIM EM UMA LISTA SIMPLESMENTE ENCADEADA

Aqui, os novos registros sempre ficarão apóis todos os que já foram inseridos anteriormente.

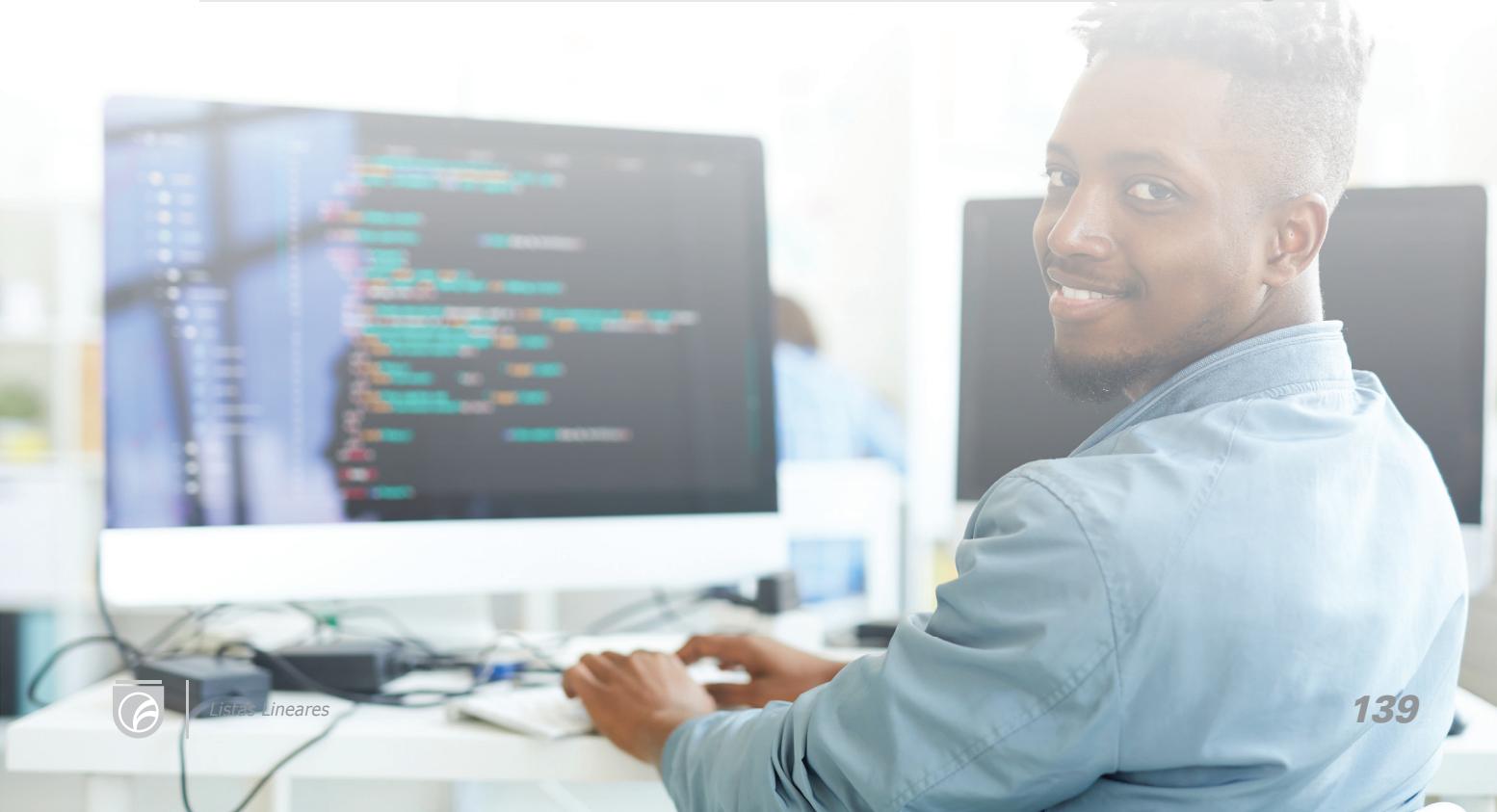
Observe o código referente a este tipo de inserção:

```
1 listaDupla* inserirFim(listaDupla *lista) {
2     //Alocar novo espaço
3     listaDupla *novo = (listaDupla*) malloc (sizeof(listaDupla));
4     //Preencher os campos básicos
5     //Código - gerado aleatoriamente de 0 até 100
6     novo->codigo = rand() % 100;
7
8     //Nome
9     printf("\nNome do funcionário: ");
10    fgets(novo->nome, 40, stdin);
11
12    //Idade - Não haverá validação
13    printf("\nIdade do funcionário: ");
14    scanf ("%i", &novo->idade);
15
16    //Salário - Não haverá validação
17    printf("\nSalário R$");
18    scanf ("%f", &novo->salario);
19
20    //Ajustar ponteiro para que este registro seja o último da lista
21    novo->proximo = NULL;
22
23    //Conectar o novo registro à lista
24    if (lista == NULL) {
25        //Se a lista for nula ele será o primeiro
26        novo->anterior = NULL;
27        return novo;
28    } else {
29        //Se a lista não for nula precisa encontrar o último
30        listaDupla *tmp = lista;
31        //Encontrar o registro que aponta para null
32        while (tmp->proximo != NULL) {
33            tmp = tmp->proximo;
34        }
35        //Apontar o último para o novo, tornando-o o penúltimo
36        tmp->proximo = novo;
37        //O novo deve apontar para o penúltimo
38        novo->anterior = tmp;
39        return lista;
40    }
41 }
```

Não existem registros apóis o novo inserido, por isso ele sempre terá seu campo próximo apontando para NULL (linha 21), e se ele for o único registro da lista, seu anterior também deverá ser NULL (linha 26).

ATENÇÃO

E lembre-se que um código alternativo a esse seria sempre armazenar o endereço de memória último registro, não necessitando de um loop para a busca, facilitando o processamento.



Veja o que aconteceria após a primeira inclusão:

codigo 41
Nome Gabrielle
Idade 20
Salário 2500.93
anterior NULL
proxímo NULL
endereço do item
0x14ec010

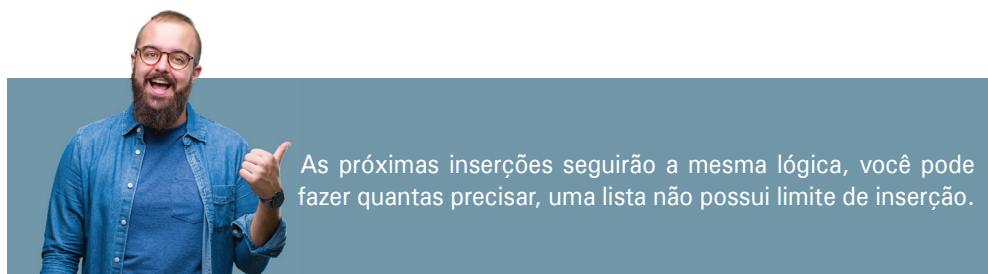
Agora veremos como seria a segunda inserção, no nosso exemplo, a inserção é pelo fim da lista, portanto, o novo registro será o último:

codigo 41	codigo 215
Nome Gabrielle	Nome José
Idade 20	Idade 70
Salário 2500.93	Salário 1299.43
anterior NULL	anterior 0x14ec010
proxímo 0x14ec030	proxímo NULL
endereço do item	endereço do item
0x14ec010	0x14ec030

Veja que os ponteiros, anterior e próximo, apontam para os registros corretos, garantindo a ligação da lista.

Vamos ver como seria a terceira inserção:

codigo 41	codigo 215	codigo 523
Nome Gabrielle	Nome José	Nome Eder
Idade 20	Idade 70	Idade 34
Salário 2500.93	Salário 1299.43	Salário 815.32
anterior NULL	anterior 0x14ec010	anterior 0x14ec030
proxímo 0x14ec030	proxímo 0x14ec050	proxímo NULL
endereço do item	endereço do item	endereço do item
0x14ec010	0x14ec030	0x14ec050



BUSCA EM UMA LISTA DUPLAMENTE ENCADEADA

A partir de um identificador do registro será possível efetuar uma busca entre elementos da lista, para este caso, utilizaremos o código como identificador, e, iremos de espaço de memória em espaço de memória, sempre utilizando o campo próximo para nos “levar”.

A função receberá, além da lista, o código a ser localizado, ambos passados por parâmetro.

```
1 listaDupla* buscarRegistro(listaDupla *lista, int cod) {
2     listaDupla *atual = lista;
3     //Procurar registro
4     while (atual != NULL) {
5         if (atual->codigo == cod) {
6             //Se encontrar retorna o endereço de memória
7             return atual;
8         }
9         //Se não encontrar procura no próximo
10        atual = atual->prox;
11    }
12    //Se não encontrar retorna nulo
13    return NULL;
14 }
```

No código, a função retorna o endereço de memória do registro que será excluído, e como o próprio registro já possui a informação de quem virá antes e depois dele, será fácil manipular os ponteiros.

Por isso, se efetuarmos a busca do código 215, a função nos retornaria o endereço de memória 0x14ec030, e este registro já me informa que o registro de endereço 0x14ec010 vem antes dele e o de endereço vem depois dele 0x14ec050.

EXCLUSÃO EM UMA LISTA DUPLAMENTE ENCADEADA

Para excluirmos um registro em uma lista duplamente encadeada, é necessário realocar os ponteiros, garantindo que a lista não será desconectada.

Vamos ver o código:

```
1 listaDupla* excluirRegistro (listaDupla *lista, int cod) {
2     listaDupla *atual = lista;
3
4     //Procurar registro que será excluído
5     atual = buscarRegistro(lista, cod);
6     if (atual != NULL) { //Encontrei o registro
7         if (atual->anterior == NULL) { //É o primeiro registro da lista
8             //O segundo registro da lista passará a ser o primeiro
9             lista = lista->prox;
10            if (lista != NULL) {
11                lista->anterior = NULL;
12            }
13        } else { //É o registro do meio ou o último
14            atual->anterior->prox = atual->prox;
15            if (atual->prox != NULL) {
16                atual->prox->anterior = atual->anterior;
17            }
18        }
19        //Apagar o registro da memória
20        free(atual);
21    } else {
22        printf("\nRegistro não encontrado");
23    }
24    //Retornar a lista, atualizada ou não.
25    return lista;
26 }
```



Vamos “traduzir” alguns itens do código!

- **atual**: registro que será excluído;
- **atual → anterior**: registro que vem antes do que será excluído;
- **atual → anterior → proximo**: ponteiro próximo, do registro que vem antes do que será excluído;
- **atual → proximo**: registro que vem depois do que será excluído;
- **atual → proximo → anterior**: ponteiro anterior, do registro que vem depois do que será excluído.



O apontamento “atual → anterior → próximo”, da linha 14, indica que estou me referindo ao ponteiro próximo, do registro anterior ao que será excluído, e ele receberá o conteúdo do ponteiro próximo daquele que será excluído, ou seja para onde ele aponta. Afinal, o ponteiro próximo do registro que vem antes do atual (que será excluído), aponta para o atual, sendo assim, já que o atual será excluído, ele “informará” para onde ele aponta, mantendo a ligação da lista.

O apontamento “atual → proximo → anterior”, da linha 16, indica que estou me referindo ao ponteiro anterior, do registro que vem depois do que será excluído, e ele receberá o conteúdo do ponteiro anterior daquele que será excluído, ou seja para onde ele aponta. Afinal, o ponteiro anterior do registro que vem depois do atual (que será excluído), aponta para o atual, sendo assim, já que o atual será excluído, ele “informará” para onde ele aponta, mantendo a ligação da lista. Mas essa ação só pode acontecer se houver algum registro depois do que será excluído.

Agora vamos simular como seria a exclusão do registro de código 215, da lista. Como referência utilizaremos a simulação da lista encadeada pelo início, que possui a seguinte configuração:

codigo 523	codigo 215	codigo 41
Nome Eder	Nome José	Nome Gabrielle
Idade 34	Idade 70	Idade 20
Salário 815.32	Salário 1299.43	Salário 2500.93
anterior NULL	anterior 0x14ec050	anterior 0x14ec030
proxximo 0x14ec030	proxximo 0x14ec010	proxximo NULL
endereço do item	endereço do item	endereço do item
0x14ec050	0x14ec030	0x14ec010



Após a exclusão, e o reajuste dos ponteiros, a configuração da lista seria assim:

codigo 523	codigo 41
Nome Eder	Nome Gabrielle
anterior NULL	anterior 0x14ec050
proximo 0x14ec010	proximo NULL
endereço do item	endereço do item
0x14ec050	0x14ec010

IMPORTANTE

É importante armazenar o endereço inicial da lista, permitindo assim o acesso a ela. Se por acaso, o primeiro registro for excluído, a lista passará a armazenar o endereço do segundo registro, indicando seu “novo” início. A exclusão poderá ocorrer para qualquer registro, inclusive, se necessário, a lista poderá voltar a ser vazia (NULL).



Assim finalizamos nosso conteúdo, saiba que não existe um tipo melhor ou pior, tudo dependerá das suas necessidades computacionais. Não se esqueça que, cada estrutura atende a uma situação específica e possui vantagens e desvantagens para sua utilização, por isso, avalie a solução proposta considerando seu gasto de memória e processamentos necessários, para assim optar para uma estrutura ou outra.



Síntese

Aqui terminamos o conteúdo de listas lineares.

Os conceitos mais importantes, falados neste módulo foram: o de lista linear, que denomina o capítulo, você descobriu que as listas são lineares pois precisam ser acessadas do início ao fim, de forma sequencial.

Para a implementação deste tipo de estrutura em nossos códigos, podemos utilizar listas estáticas ou dinâmicas.

As listas estáticas são os já conhecidos vetores, uma vantagem é que possui endereço sequencial, portanto, não precisamos gastar espaços de memória armazenando esta informação, porém, a desvantagem é que ela precisa ser dimensionada em tempo de compilação, ou seja, antes do programa iniciar já devemos definir seu tamanho, e mesmo que não aproveitemos todos os espaços ele estará reservado para sua utilização, além disto, as operações de inserção e remoção gastam mais processamento, visto que teremos que alocar ou realocar todos as posições, evitando espaços vazios.

As listas dinâmicas possuem apontamento para garantir que cada registro possua a informação de localização de seu próximo registro, ou do registro anterior. E essa caracteriza uma vantagem, afinal, na estrutura teremos que separar um espaço de memória que armazenará esta informação. Mas, uma grande vantagem é que seu dimensionamento é feito em tempo de execução, ou seja, durante o programa seu tamanho pode crescer e/ou diminuir sem gastos desnecessários de memória, e aproveitando todo espaço declarado.

Ainda sobre listas dinâmicas, você também viu que elas podem ter suas inserções pelo início ou pelo fim da lista, e cada tipo possui suas informações que atendem a cada tipo definido.

Por fim, lembre-se que, para qualquer tipo de lista, você poderá aplicar as ações de: criar, inserir elementos, excluir elementos e pesquisar. Bastando apenas atentar-se para as exigências de cada ação, e para o tipo de lista escolhida por você.

Até mais,

Referências

BACKES, André. **Linguagem C:** completa e descomplicada. 2. ed. Rio de Janeiro: Elsevier, 2019.

DAMAS, Luis. **Linguagem C.** 10. ed. Rio de Janeiro: Ltc, 2016.

MIZRAHI, Victorine Viviane. **Treinamento em linguagem C.** São Paulo: Pearson Prentice Hall, 2008. 407 p.

REESE, Richard. **Understanding and using C pointers.** Sebastopol: O'Reilly, 2008. 407 p.