

Memória dinâmica

Por enquanto, a capacidade de armazenamento de um programa estava limitada às variáveis declaradas no código. O espaço destinado à estas variáveis é denominado **memória estática**, pois seus tamanhos foram definidos durante a compilação e sua posição é fixada antes do programa começar a executar. Portanto, tamanho e posição não podem mais ser modificados durante a execução, mesmo que a demanda de memória do programa venha a crescer. O gerenciamento da memória estática é realizado automaticamente pelo compilador e pelo sistema operacional.

Com memória estática apenas, o programa não pode modificar o tamanho de um vetor já declarado. Neste capítulo estudaremos os mecanismos da linguagem C para escrever código que gerencia a própria memória que usa. Ele poderá aumentar ou diminuir a memória (e, portanto, o tamanho) de estruturas e vetores.

O ganho desta flexibilidade exige uma compreensão melhor do funcionamento da memória, de algoritmos mais elaborados e de novas estruturas de dados. Os apontadores são uma opção para contornar esta limitação. Eles permitem endereçar outras posições de memória além da memória estática. Desta forma, eles podem ampliar as fronteiras do programa para armazenar novos dados na memória. Este novo estilo de gerenciamento de memória será chamado de **memória dinâmica**.

Gerenciar memória dinâmica

O programa precisa chamar funções especiais para solicitar mais espaço na memória. O programa torna-se proprietário da memória obtida como resultado da execução destas funções especiais. Dentro de um espaço que foi alocado como memória dinâmica, o programa poderá armazenar qualquer dado (números inteiros ou fracionários, vetores, estruturas, etc), como melhor convier ao programador. Um certo planejamento é necessário para que o programa implemente sua própria lógica de controle para a memória dinâmica.

No final do processamento, o programa deve devolver (liberar) a memória dinâmica que não é mais necessária. Se o programa requisitar memória dinâmica e não foi liberando à medida que não necessitar mais dela, então, aos poucos e repetindo essa situação, ele poderá se apropriar de toda memória disponível na máquina, e até travar o sistema.

Obter memória dinâmica

Para solicitar um trecho contíguo de memória durante a execução do programa, é necessário chamar a função `malloc` (*memory allocation*). O tamanho do espaço solicitado, em bytes, deve ser passado como parâmetro para a função `malloc`. Esse espaço deve ser suficiente para acomodar as informações que desejamos armazenar neste trecho de memória dinâmica que será alocado.

O resultado da solicitação será o endereço do, i.e. um apontador para o, primeiro byte do trecho de memória obtido, ou a constante `NULL` se a solicitação não puder ser atendida. É necessário guardar este endereço em uma variável tipo apontador, pois ele será a única forma através da qual poderemos acessar este trecho de memória dinâmica que acaba de ser alocado.

Por exemplo, para solicitar 1000 bytes e guardar o endereço inicial do trecho de memória obtido em um apontador `ap`:

```
void * ap;  
ap = malloc(1000);
```

O endereço retornado por `malloc` é totalmente genérico e não possui um tipo especificado. Por este motivo, o apontador que guarda este endereço é declarado como `void *`, ou seja, um apontador para um dado cujo tipo não conhecemos.

Ler e atribuir na memória dinâmica

Devido a falta do tipo do apontador, o programa não sabe como deve tratar os dados armazenados no trecho de memória dinâmica. Um apontador de tipo `void *` não aceita o operador de dereferência `*` para atribuição ou leitura.

O programa precisa converter explicitamente o endereço obtido por `malloc` para um apontador para o tipo de dados que desejamos armazenar nesta área de memória dinâmica.

Por exemplo, se desejamos armazenar um número inteiro na memória dinâmica, devemos fazê-lo através de um apontador para um número inteiro. Para isso, convertamos apontador genérico obtido com `malloc` para um apontador para número inteiro:

```
void * ap;  
int *numero;  
  
ap = malloc(1000);  
numero = (int *)ap;
```

Agora, `numero` aponta para o mesmo endereço de memória que o apontador `ap`, ou seja, `numero` aponta para o início do trecho de memória dinâmica obtida por `malloc`. Através do apontador `numero`, o programa trata o início deste trecho de memória como um número inteiro. Podemos atribuir ao endereço apontado por `numero` da mesma forma como já estamos familiarizados:

```
*numero = 10;
```

Convertendo um apontador para um apontador para outro tipo, podemos forçar o programa a interpretar o conteúdo naquele endereço de memória como um tipo de dados diferente.

Podemos armazenar o endereço obtido por `malloc` diretamente no apontador `numero`, sem necessidade do apontador `ap`:

```
int *numero;  
numero = (int *)malloc(1000);  
*numero = 10;
```

Observação 1: O armazenamento do número inteiro requer apenas 4 bytes. Como solicitamos 1000 bytes, o número será armazenado nos primeiros 4 bytes do trecho de memória dinâmica e os demais 996 bytes estão em excesso. É importante solicitar o número correto de bytes com `malloc`, para evitar desperdício.

Observação 2: O apontador retornado pela função `malloc` é a única forma de acessar este novo espaço de memória dinâmica. Através dele, o programa pode escrever e ler dados nesse espaço. É necessário guardar este apontador em uma variável convencional. Se durante a execução do programa este apontador se perder em decorrência de um erro de programação, este espaço de memória dinâmica permanecerá eternamente alocado na memória, mas será impossível recuperar sua posição.

Exemplo

Desejamos criar uma variável de tipo inteiro na memória dinâmica. Utilizaremos a função `malloc` para solicitar espaço. Uma variável de tipo `int` ocupa 4 bytes de memória, que é o valor que precisa ser passado para a função. O endereço da memória obtida deve ser armazenado na variável apontadora

`p`, de tipo `int *`. Como `malloc` retorna um apontador genérico, antes de atribuí-lo à variável `p`, é necessário realizar uma conversão explícita para `int *`:

```
int *p;
p = (int*)malloc(4);
```

Consulte: MemóriaDinâmica\malloc\malloc.vcproj

O conteúdo da nova variável poderá ser acessado através do apontador `p`:

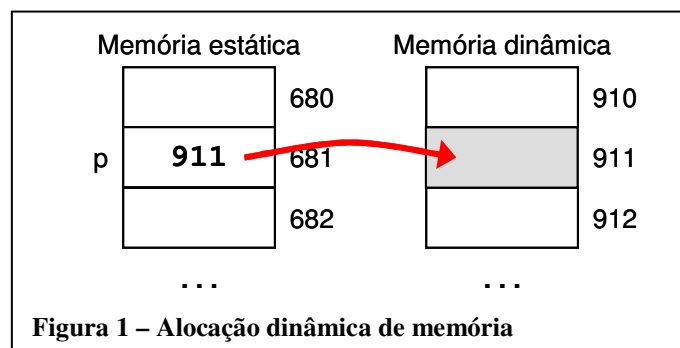
```
*p = 5;
printf("%d", *p);
```

Liberar memória dinâmica

Um espaço de memória solicitado por `malloc` permanece sob propriedade do programa por tempo indeterminado. É preciso devolver este espaço explicitamente ao sistema, quando não precisarmos mais dele. Para isso, devemos usar a função `free`. Ela recebe como parâmetro o apontador para o espaço de memória que deve ser liberado.

Uma vez liberado, é impossível acessar novamente este espaço de memória dinâmica. Todos os apontadores referentes a este espaço tornam-se inválidos e será um erro tentar ler ou artibuir através deles.

Até o final da execução, é conveniente que o programa execute uma chamada para `free` para cada apontador obtido por `malloc`, assim liberando toda a memória requisitada.



Para liberar a memória dinâmica criada no exemplo anterior:

```
free(p);
```

Criar vetor em memória dinâmica

No exemplo anterior, utilizamos `malloc` para obter um apontador para a um pequeno espaço na memória, suficiente para armazenar um valor de tipo inteiro. Um uso mais interessante de memória dinâmica é criar vetores com tamanho exatamente igual ao desejado pelo programa. Basta chamar `malloc` e assim obter o endereço para um trecho de memória suficientemente grande para armazenar o vetor completo.

É muito conveniente a forma como a linguagem C representa vetores. Na prática, o compilador não distingue entre um vetor declarado no início do programa ou um apontador obtido com `malloc`. Ambos são tratados como apontadores para o primeiro elemento do vetor. Para acessar os demais elementos do vetor, utilizaremos o habitual operador de índice `[]`, ou usaremos aritmética de apontadores para percorrer o vetor.

Antes de chamar a função `malloc`, o programa precisa calcular o tamanho total do vetor para garantir que o espaço solicitado será suficiente para armazenar todos os elementos.

Para um vetor de 10 inteiros, cada qual ocupando 4 bytes, precisamos chamar `malloc` para pedir 40 bytes de memória dinâmica.

Uso de `sizeof`

Na solicitação de memória para vetores é necessário saber o tamanho de cada elemento, em bytes, para calcular o tamanho total do vetor. Isto pode ser uma tarefa não trivial, pois os tipos de dados podem apresentar tamanhos diferentes em diferentes sistemas. Além disso, tipos formados por estruturas complexas não necessariamente tem o tamanho igual a soma do tamanho de seus atributos.

Para auxiliar nesta tarefa, a linguagem C possui a pseudo-função `sizeof`. Ela recebe como parâmetro um nome de tipo de dados (inteiro, ponto flutuante, caractere, estrutura, enumeração, etc.) ou uma variável. A função `sizeof` calcula o tamanho necessário para armazenar este tipo de dado ou esta variável.

Por exemplo, para saber o tamanho necessário para armazenar um número inteiro, utiliza-se a expressão `sizeof(int)`. Para conhecer o tamanho necessário para número fracionário, utilizar-se-ia `sizeof(float)`, ou `sizeof(double)`.

Criar espaço para o vetor

O primeiro passo é calcular o tamanho total em bytes do vetor, com auxílio da pseudo-função `sizeof`. Em seguida, chama-se a função `malloc` que retorna um apontador genérico para o espaço obtido na memória dinâmica. Este apontador precisa ser convertido explicitamente para um apontador para o tipo dos elementos do vetor.

É conveniente verificar se o apontador retornado por `malloc` é válido ou não. Se ele for NULL, significa que `malloc` falhou e a memória não pôde ser alocada, por exemplo, por que toda memória disponível já foi esgotada. No caso de falha, o programa precisa ser interrompido, pois não há onde armazenar o vetor.

Exemplo

O programa pergunta ao usuário o tamanho desejado para o vetor e obtém um trecho de memória dinâmica com tamanho exato para armazenar o vetor:

```
int *vetor;
int tamanho;
scanf("%d", &tamanho);
...
vetor = (int*)malloc(sizeof(int) * tamanho);
if (vetor == NULL) {
    printf("Nao ha memoria dinamica suficiente para esta operacao\n");
    return;
}
```

Consulte: MemoriaDinamica\Vetor01\Vetor01.vcproj

Acesso aos elementos do vetor

Os elementos podem ser acessados de duas formas diferentes. A primeira opção é utilizar o operador `[]`. Lembre que o compilador entende o apontador como contendo o endereço do primeiro elemento do vetor, e calcula automaticamente o endereço do elemento desejado. Continuando o exemplo anterior, vamos somar o terceiro e quarto elemento e guardar o resultado no segundo:

```
vetor[1] = vetor[2] + vetor[3];
```

A segunda forma de acesso é utilizar aritmética de apontadores:

```
*(vetor+1) = *(vetor+2) + *(vetor+3);
```

Liberar espaço do vetor

No fim do programa, não se deve esquecer de liberar o vetor com a função `free`.

```
free(vetor);
```

Alterar tamanho de um espaço de memória dinâmica

Com auxílio de memória dinâmica, foi possível criar um vetor exatamente no tamanho necessário. No entanto, uma vez chamada a função `malloc`, o programa não pode mais alterar o tamanho do vetor.

A função `realloc` (*reallocate memory*) solicita ao sistema redimensionar um espaço de memória adquirido previamente com `malloc`. Esta função recebe como parâmetro o apontador para o espaço de memória que desejamos redimensionar e um número inteiro que é o novo tamanho, em bytes. O resultado da solicitação será um apontador para um novo espaço de memória dinâmica com as dimensões desejadas, ou NULL se a solicitação não puder ser atendida por falta de memória disponível.

Se o novo tamanho for maior, `realloc` copia os dados já existentes do espaço atual para o novo espaço alocado. Se o novo tamanho solicitado for menor que o tamanho do espaço

original, então `realloc` copia tantos bytes quanto possíveis do espaço original para o novo espaço.

A função retorna o apontador para o novo espaço de memória, que pode estar em posição diferente daquela ocupada pelo espaço alocado antes da chamada a `realloc`. Portanto, o programa precisa garantir que todos os apontadores que operavam sobre o espaço original sejam atualizados.

Exemplo

O programa é parecido com o exemplo anterior. Em uma segunda etapa, ele permite redimensionar o vetor:

```
int *vetor, *elemento, i;
int tamanho, novo_tamanho;
```

Lê o tamanho do vetor e cria memória dinâmica para armazenar o mesmo

```
scanf("%d", &tamanho);
vetor = (int*)malloc(sizeof(int) * tamanho);
if (vetor == NULL) {
    printf("Nao ha memoria dinamica suficiente para esta operacao\n");
    return;
}
```

Lê os elementos do vetor

```
for (elemento = vetor; elemento < vetor + tamanho; elemento++) {
    scanf("%d", elemento);
}
```

Lê o novo tamanho para o vetor. Note que a variável `vetor` precisa receber o novo apontador, pois o endereço de memória que contém o vetor pode ter mudado.

```
scanf("%d", &novo_tamanho);
vetor = realloc(vetor, novo_tamanho);
```

Consulte: MemoriaDinamica\Vetor02\Vetor02.vcproj

Criar uma estrutura na memória dinâmica

Outro uso interessante da combinação apontadores e memória dinâmica é alocar espaço para armazenar uma estrutura (`struct`). Este procedimento é análogo à criação de vetores em memória dinâmica. Primeiro, determina-se o tamanho da estrutura com a pseudo-função `sizeof`. Em seguida, executa-se `malloc` para criar um espaço de memória dinâmica para armazenar esta estrutura.

Exemplo

O programa cria um número complexo em memória dinâmica:

```

struct complexo {
    float real;
    float imaginario;
}

```

Declara um apontador para um número complexo:

```

struct complexo *numero_complexo;

```

Solicita um trecho na memória dinâmica para armazenar o número. O resultado será um apontador para o espaço que será alocado. O apontador precisa ser convertido para um apontador para o tipo estrutura de número complexo:

```

numero_complexo = (struct complexo *)malloc(sizeof(struct complexo));
if (numero_complexo == NULL) {
    printf("Nao ha memoria dinamica suficiente para esta operacao\n");
    return;
}

```

Acesso aos atributos da estrutura

Os atributos da estrutura são acessados através do operador de-referência (*), tal como é feito para todos os apontadores. Por exemplo, para acessar a estrutura endereço é dado pelo apontador `numero_complexo`, escrevemos `*numero_complexo`.

Para acessar atributos da estrutura, utilizamos o operador de seleção de atributo (.). Note o uso correto dos parênteses:

```

(*numero_complexo).real = 10.0;
(*numero_complexo).imaginario = 10.0;

```

Os parênteses são necessários pois sem eles, o compilador não saberia se deve aplicar o operador de-referência (*) sobre `numero_complexo` ou `numero_complexo.real` (ou `numero_complexo.imaginario`). Neste caso, indicamos que o operador de-referência (*) deve ser aplicado sobre `numero_complexo` para obter o endereço da estrutura para qual ele aponta, para depois aplicar o operador de seleção de atributo (.).

Existe um atalho para acessar atributos de uma estrutura dada por um apontador: o operador `->`. As linhas abaixo são equivalentes às anteriores:

```

numero->real = 10.0;
numero->imaginario = 5.0;

```

Como de costume, no final o programa precisa liberar o espaço de memória dinâmica:

```

free(numero);

```

Estruturas ligadas em memória dinâmica

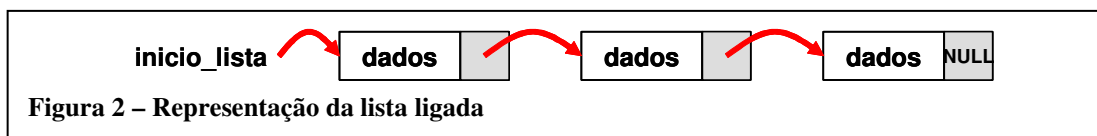
O vetor em memória dinâmica contorna a limitação do tamanho fixo. O programa solicita uma quantidade exata de memória assim que souber o tamanho desejado do vetor. Como estas operações precisam ser programadas manualmente, isto requer um esforço de programação adicional e muita atenção do programador.

Agora desejamos tratar uma forma alternativa para armazenar na memória uma coleção com elementos diferentes e de mesmo tipo. Um vetor armazenaria todos os elementos em posições consecutivas na memória, formando um único bloco.

O vetor possui algumas desvantagens:

- Seu tamanho é fixo, e se o estamos armazenando em memória dinâmica, isto exige algum esforço extra de programação para ir redefinindo seu tamanho.
- É difícil adicionar elementos no começo ou no meio do vetor. Por exemplo, para adicionar um elemento no início, é necessário mover todos os elementos uma posição para frente, abrindo espaço para o novo elemento.
- Pelo mesmo motivo é difícil retirar elementos ou trocá-los de posição.
- O vetor define uma ordem para percorrer os elementos.

O modelo de **lista ligada** propõe reservar memória de forma independente para cada elemento, em posições que não são necessariamente consecutivas. A coleção é estruturada com apontadores conectando um elemento ao outro como em uma corrente. Cada elemento possui um apontador que indica o endereço onde se encontra o próximo elemento. A Figura 2 ilustra como seria uma lista ligada na memória.



O elemento base da lista será declarado como uma estrutura (*struct*) formada por um ou mais atributos de dados, além de um apontador para o próximo elemento da sequência. O último elemento aponta para **NULL**, indicando o fim da sequência. Cada elemento é criado com **malloc** e permanece na memória até ser liberado através de **free**. O programa mantém um apontador para o primeiro elemento da coleção. Para acessar um determinado elemento, realizamos uma busca que parte do primeiro elemento, continua seguindo os apontadores para o próximo elemento, até encontrar aquele que se deseja acessar.

A combinação de estruturas ligadas e memória dinâmica é um dos mecanismos de armazenamento mais versáteis em programação. As estruturas ligadas permitem que novos elementos sejam agregados ou removidos da coleção de forma rápida e fácil. Além disso, as necessidades de uso de memória podem ser controladas pelo programador, que requisita ou libera memória apenas quando vai agregar mais elementos à lista, ou vai retirar elementos da lista. Não é preciso reservar logo de início o máximo de memória que o programa eventualmente usará.

A versatilidade das estruturas ligadas implica alguns pontos de ineficiência no mecanismo de armazenamento. Para cada elemento, comparando com vetores, há um acréscimo no número de bytes necessário para armazená-lo, pois além dos dados precisamos guardar o apontador para o próximo elemento. Além disso, a estrutura ligada não permite acessar diretamente um elemento através de um índice, como era o caso de vetores. A lista ligada sempre exige uma busca a partir do primeiro elemento.

Desta forma, dizemos que a lista ligada é indicada para programas que inserem, alteram e removem freqüentemente elementos da lista, mas poucas vezes buscam elementos pelo índice. Por exemplo, listas ligadas não são recomendadas para vetores e matrizes, que são casos típicos de dados acessados através de índices.

Já a lista ligada é útil quando muitas vezes precisamos inserir, alterar e remover elementos da lista, ou percorrer a lista desde que não possa ser feito eficientemente sem o uso de índices.

A lista vazia

A Figura 2 mostra uma lista ligada com três elementos. Precisamos de uma representação para a lista com zero elementos. A solução mais simples será fazer a variável que aponta para o primeiro elemento da lista armazenar o valor **NULL**, e portanto apontando para nenhum elemento.

Declaração de elementos de uma lista ligada

Um elemento da lista ligada será sempre uma estrutura, cujos atributos são os dados do elemento e o apontador para o próximo elemento. Isto é válido mesmo quando cada elemento armazena apenas um dado.

Por exemplo, uma lista ligada para armazenar um lista de compras poderia ser declarada na forma:

```
struct elemento {
    char descricao[100];
    int quantidade;
    float valor_unitario;

    struct elemento *proximo;
}
```

A estrutura é composta pelos atributos `descricao`, `quantidade`, `valor_unitario` e `proximo`. Os três primeiros são informações comuns que serão armazenadas em cada elemento da lista ligada. O atributo `proximo` armazenará um apontador para o próximo elemento da lista ligada. Note que ele é do tipo `struct elemento *`, ou seja, um apontador para uma estrutura que armazena um (outro) elemento do mesmo tipo. A linguagem C permite declarar apontadores nestas situações, mesmo que a declaração do elemento atual ainda não esteja completa.

O elemento apontado pelo atributo `proximo`, por sua vez, conterà um outro atributo `proximo` que, por sua vez, apontará para um terceiro elemento, e assim por diante, formando uma sequência encadeada de estruturas do mesmo tipo, todas interligados pelo campo `proximo`.

Além da declaração de tipo para a estrutura, o programa precisa conhecer o apontador para o primeiro elemento da lista ligada. Sem ele, o programa não poderia encontrar o primeiro elemento, e por consequência, não poderia encontrar os demais elementos.

Portanto declaramos também uma variável apontadora para o primeiro elemento:

```
struct elemento * inicio_lista;
```

Em situações mais complexas, um elemento poderá ter mais de um apontador declarados como atributo da estrutura. Isso permite com que os elementos sejam encadeados segundo estratégias diferentes, o que pode facilitar o acesso a cada um deles. Em contrapartida, será preciso alocar mais memória para acomodar cada um dos atributos de tipo apontadores. Como cada apontador usa (tipicamente) 4 bytes de memória, em muitos casos este é um preço leve, em face dos benefícios que mais de um apontador pode trazer.

Percorrer a lista ligada

Desejamos percorrer a lista, com objetivo de encontrar um elemento ou imprimir todos eles individualmente. Para isso, mantemos um apontador (chamado `atual`) para o elemento que estamos visitando no momento. Este apontador começa apontando para o primeiro elemento da lista, cujo endereço encontramos em `inicio_lista`.

Para cada elemento, verificamos os dados armazenados na estrutura. Em seguida, atualizamos o apontador `atual` para o endereço do próximo elemento (`atual->proximo`). Repetimos este processo até que o apontador `atual` receba `NULL`, sinalizando que o fim da sequência foi alcançado.

Note que não precisamos conhecer o número de elementos que estão encadeados na lista. Não há necessidade de uma variável contadora ou índice, como era o caso quando lidávamos com vetores.

Exemplo:

```
struct elemento *atual;
atual = inicio_lista;
while (atual != NULL) {
    // Verifica, processa, etc o dado contido neste elemento

    . . .

    // Segue para o proximo elemento da sequencia
    atual = atual->proximo;
}
```

Inserir um novo elemento

Desejamos adicionar um novo elemento à lista. Nesses casos, começamos reservando memória para este elemento (através de `malloc`). O apontador para este novo elemento é guardado em uma variável, `novo`. Em seguida, o programa preenche este elemento com seus dados:

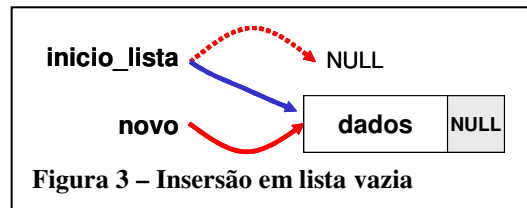
```

struct elemento *novo;
novo = (struct elemento *)malloc(sizeof(struct elemento));
...
novo->quantidade = 10;
novo->valor_unitario = 1.99;

```

Inserção em lista vazia

A inserção possui uma condição importante que precisa ser verificada: se atualmente a lista está vazia.



Se for caso, a inserção é trivial. O apontador `inicio_lista` que é agora NULL passa a ser atualizado para o endereço do novo elemento (Figura 3). Não devemos esquecer que o atributo `proximo` do novo elemento deve receber o valor NULL para sinalizar que agora ele é também o último elemento da sequência:

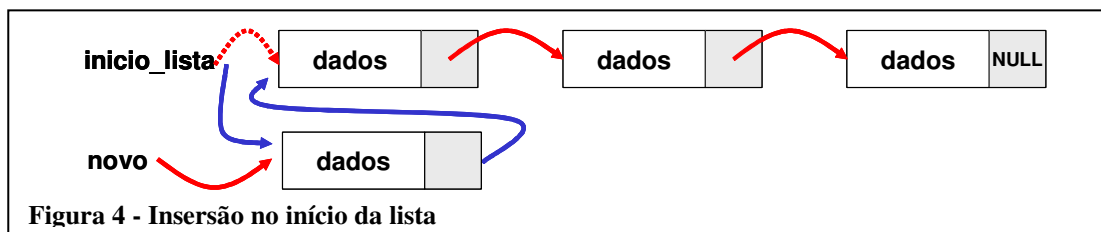
```

novo->proximo = NULL;
inicio_lista = novo;

```

Se a lista não for vazia, então temos três opções: inserir no início, no fim ou no meio da lista (segundo algum critério de ordenação).

Inserção no início da lista



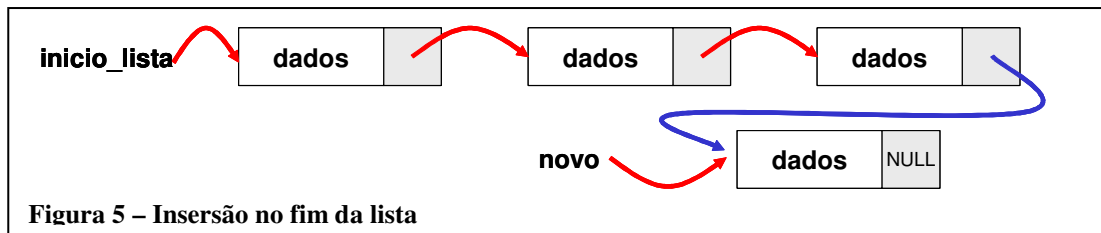
A opção mais ágil é inserir um elemento no início da lista. O novo elemento deve apontar para o início da sequência já existente. Atualizamos o apontador `inicio_lista` para apontar para o endereço do novo elemento:

```

novo->proximo = inicio_lista;
inicio_lista = novo;

```

Inserção no fim da lista



Inserir elementos no final requer mais passos. Precisamos percorrer toda a lista para descobrir qual é o último elemento (Figura 5). Começamos assumindo que o último elemento é o primeiro da lista. Se o atributo `proximo` deste elemento não for NULL, então significa que precisamos seguir para o próximo elemento, até encontrar o último:

```
struct elemento *ultimo;
ultimo = inicio_lista;
while (ultimo->proximo != null) {
    ultimo = ultimo->proximo;
}
```

Nesse ponto, o atributo `proximo` do último elemento conterá NULL. Atualizamos este atributo para o endereço do novo elemento. Não podemos esquecer que o atributo `proximo` do novo elemento também deve conter o valor NULL, sinalizando que ele agora é o último elemento da sequência:

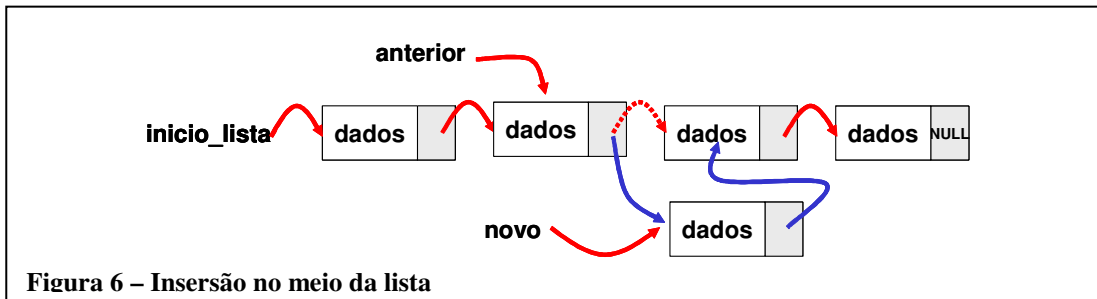
```
novo->proximo = NULL;
ultimo->proximo = novo;
```

Porém, cuidado para o caso quando a lista é vazia. Você consegue imaginar como tratar esse caso particular? Na verdade é igual à inserção na primeira posição:

```
struct elemento *ultimo, *novo;
novo = (struct elemento *)malloc(sizeof(struct elemento));
...
novo->quantidade = 10;
novo->valor_unitario = 1.99;

if (inicio_lista == NULL) {
    // Insere o primeiro elemento
    novo->proximo = NULL;
    inicio_lista = novo;
} else {
    // Insere no final da lista
    ultimo = inicio_lista;
    while (ultimo->proximo != null) {
        ultimo = ultimo->proximo;
    }
    novo->proximo = NULL;
    ultimo->proximo = novo;
}
```

Inserção no meio da lista



Para inserir o elemento no meio da sequência, precisamos localizar o elemento (apontado pela variável `anterior`) que antecederá o novo elemento segundo o critério de ordenação adotado. A inserção é realizada fazendo o novo elemento apontar para o elemento que segue o anterior, e o elemento anterior apontar para o novo elemento.

Antes, é necessário verificar se o critério de ordenação implicaria em adicionar o elemento no início da lista. Se este for o caso, executa-se os passos discutidos anteriormente.

Suponha que os elementos possuem um atributo `valor`. Devemos adicionar o elemento de tal forma que lista permaneça ordenada de forma crescente com base nesse atributo:

```
struct elemento *anterior;
if (novo->valor < inicio_lista->valor) {
    // Inserir no início
    novo->proximo = inicio_lista;
    inicio_lista = novo;
} else {
    anterior = inicio_lista;
    while ((anterior-> proximo != NULL)
        && (anterior->valor < novo->valor)) {
        anterior = anterior->proximo;
    }
    novo->proximo = anterior->proximo;
    anterior->proximo = novo;
}
```

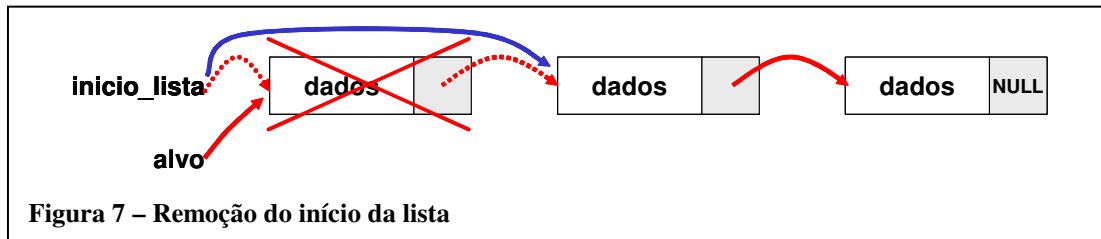
Cuidado quando a variável `anterior` aponta para o último elemento da lista. Tente encontrar uma solução para este caso particular.

Remoção de um elemento da lista

Desejamos remover um elemento da lista. Sempre precisamos identificar o elemento que desejamos remover através de uma busca que percorre a lista. Supondo que esta busca já foi realizada, assumimos que a variável `alvo` contém o endereço do elemento que deve ser removido da lista.

Verificamos se o elemento é o primeiro da lista. Se for, é necessário apenas atualizar a variável `inicio_lista` (Figura 7). Caso contrário, também aproveitamos a busca para localizar o elemento que antecede aquele que desejamos excluir, armazenando seu endereço no apontador `anterior` (Figura 8). Após remover o elemento da lista, costuma-se liberar a memória ocupada pelo elemento, usando a função `free`.

Remoção do primeiro elemento da lista



Atualizamos a variável `inicio_lista` para ela apontar para o segundo elemento (que é o sucessor do primeiro):

```
inicio_lista = inicio_lista->proximo;
```

ou:

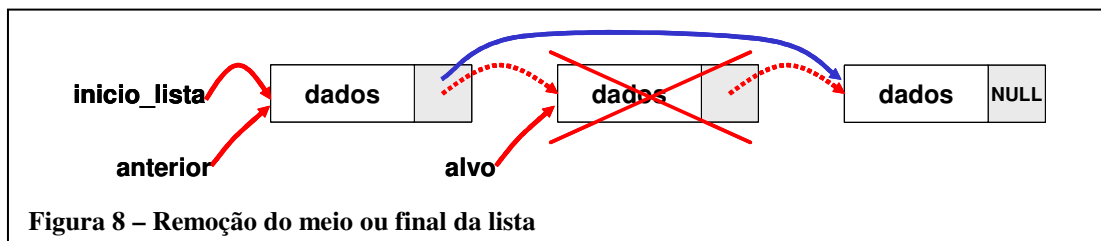
```
inicio_lista = alvo->proximo;
```

Agora liberamos a memória:

```
free(alvo);  
alvo = NULL;
```

Observação: Cuidado para o caso particular quando a lista é vazia!

Remoção do meio ou fim da lista



O elemento `alvo` é removido da lista fazendo o elemento `anterior` apontar para o sucessor do elemento `alvo`:

```
anterior->proximo = alvo->proximo;
```

ou:

```
anterior->proximo = anterior->proximo->proximo;
```

Agora liberamos a memória:

```
free(alvo);  
alvo = NULL;
```

De novo, verifique antes os casos particulares de remoção no início e fim da lista.