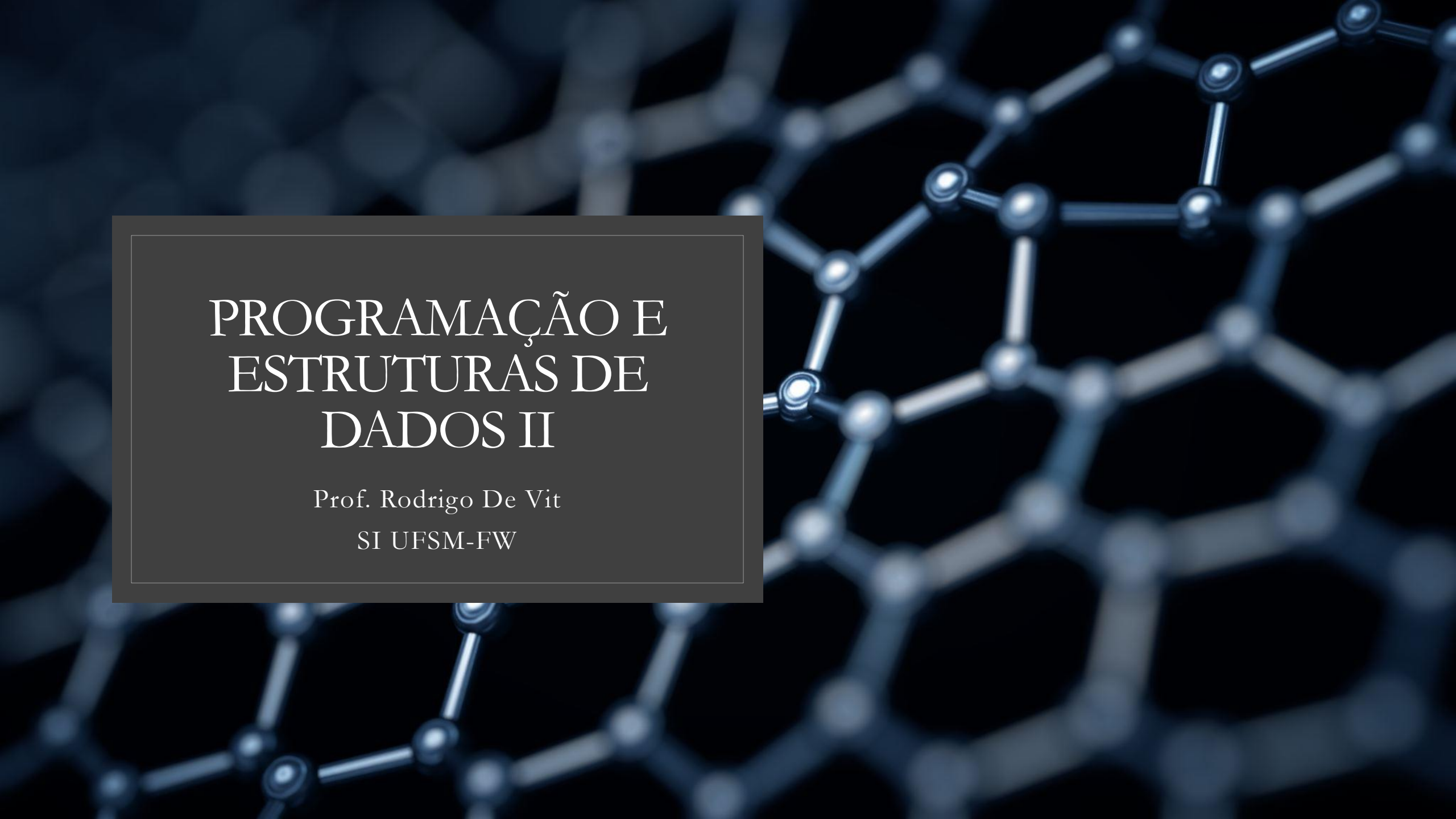


PROGRAMAÇÃO E ESTRUTURAS DE DADOS II

Prof. Rodrigo De Vit

SI UFSM-FW

Revisão

- **Aula (02), 13 e 14 de agosto:**
- UNIDADE 1 - ESTRUTURAS LINEARES E ENCADEADAS
- 1.1 - Abstração de dados.
- Neste tópico, discutiremos uma importante técnica de programação baseada na definição de ***Tipos Abstratos de Dados*** **(TAD)**. Veremos também como a linguagem C pode nos ajudar na implementação de um TAD, através de alguns de seus mecanismos básicos de ***modularização*** (divisão de um programa em vários arquivos fontes).

- Editar arquivo prog1.c e str.h.

- Agora, ao invés de repetir manualmente os cabeçalhos dessas funções, todo módulo que quiser usar as funções de str.c precisa apenas incluir o arquivo str.h.

ESTRUTURAS LINEARES E ENCADEADAS

- **Aula (03), 20 e 21 de agosto e Aula (04), 27 e 28 de agosto:**
- UNIDADE 1 - ESTRUTURAS LINEARES E ENCADEADAS
- 1.2 - Estrutura Dinâmica- Listas Encadeadas.
- 1.2.1 - Conceituação e ponteiros.
- 1.2.2 - Listas lineares.
- 1.2.3 - Manipulação de pilhas e filas.
- 1.2.4 - Listas duplamente encadeadas.

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas

- Para representarmos um grupo de dados, já vimos que podemos usar um vetor em C. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Para simplificar a discussão dos conceitos que serão apresentados agora, vamos supor que temos que desenvolver uma aplicação que deve representar um grupo de valores inteiros. Para tanto, podemos declarar um vetor escolhendo um número máximo de elementos.

```
#define MAX 1000

int vet[MAX];
```

- Ao declararmos um vetor, reservamos um espaço contíguo de memória para armazenar seus elementos, conforme ilustra a figura abaixo.

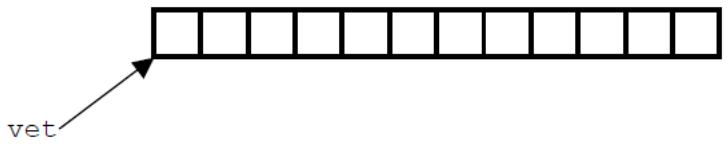
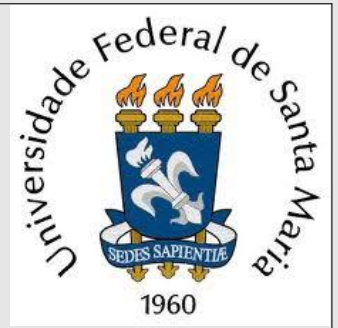


Figura 9.1: Um vetor ocupa um espaço contíguo de memória, permitindo que qualquer elemento seja acessado indexando-se o ponteiro para o primeiro elemento.

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas



- O vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barata (computacionalmente) para alterarmos a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos subutilizando o espaço de memória reservado.
- A solução para esses problemas é utilizar estruturas de dados que cresçam à medida que precisarmos armazenar novos elementos (e diminuam à medida que precisarmos retirar elementos armazenados anteriormente). Tais estruturas são chamadas ***dinâmicas*** e armazenam cada um dos seus elementos usando **alocação dinâmica**.
- A seguir, discutiremos a estrutura de dados conhecida como ***lista encadeada***. As listas encadeadas são amplamente usadas para implementar diversas outras estruturas de dados com semânticas próprias, que serão tratadas nos capítulos seguintes.

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas

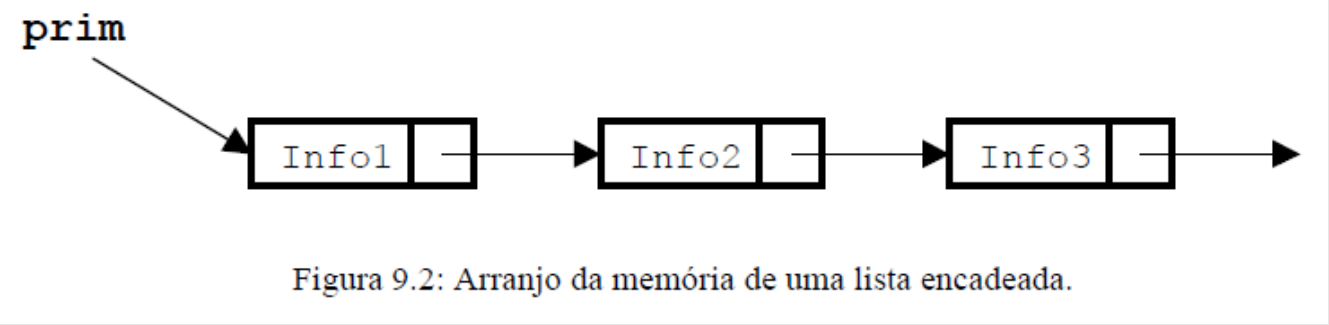


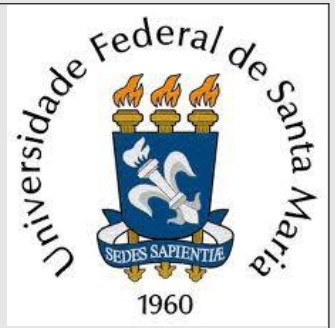
Figura 9.2: Arranjo da memória de uma lista encadeada.

- Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. O espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. Não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista.

```
struct lista {
        int info;
        struct lista* prox;
};
typedef struct lista Lista;
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas



- **Função de inicialização**
- A função que inicializa uma lista deve criar uma lista vazia, sem nenhum elemento.
- Como a lista é representada pelo ponteiro para o primeiro elemento, uma lista vazia é representada pelo ponteiro NULL, pois não existem elementos na lista. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é NULL. Uma possível implementação da função de inicialização é mostrada a seguir:

```
/* função de inicialização: retorna uma lista vazia */
Lista* inicializa (void)
{
        return NULL;
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas

- **Função de inserção**
- Uma vez criada a lista vazia, podemos inserir novos elementos nela. Para cada elemento inserido na lista, devemos alocar dinamicamente a memória necessária para armazenar o elemento e encadeá-lo na lista existente. A função de inserção mais simples insere o novo elemento no início da lista.
- Uma possível implementação dessa função é mostrada a seguir. Devemos notar que o ponteiro que representa a lista deve ter seu valor atualizado, pois a lista deve passar a ser representada pelo ponteiro para o novo primeiro elemento. Por esta razão, a função de inserção recebe como parâmetros de entrada a lista onde será inserido o novo elemento e a informação do novo elemento, e tem como valor de retorno a nova lista, representada pelo ponteiro para o novo elemento.

```
/* inserção no início: retorna a lista atualizada */
Lista* insere (Lista* l, int i)
{
   Lista* novo = (Lista*) malloc(sizeof(Lista));
   novo->info = i;
   novo->prox = l;
   return novo;
}
```

Esta função aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda a informação no novo nó e faz este nó apontar para (isto é, ter como próximo elemento) o elemento que era o primeiro da lista. A função então retorna o novo valor que representa a lista, que é o ponteiro para o novo primeiro elemento. A Figura 9.3 ilustra a operação de inserção de um novo elemento no início da lista.

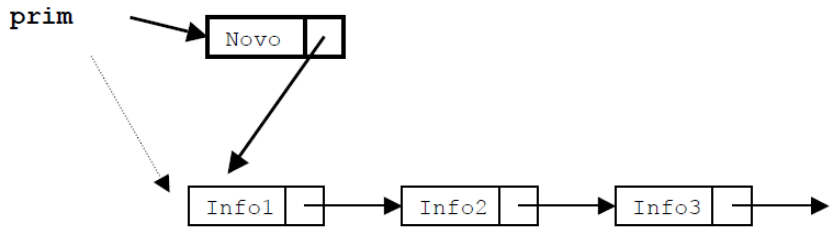
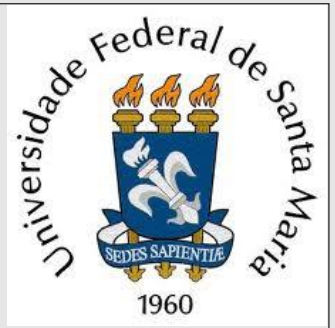


Figura 9. 3: Inserção de um novo elemento no início da lista.

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas



- **Função que percorre os elementos da lista**
- Para ilustrar a implementação de uma função que percorre todos os elementos da lista, vamos considerar a criação de uma função que imprima os valores dos elementos armazenados numa lista. Uma possível implementação dessa função é mostrada a seguir.

```
/* função imprime: imprime valores dos elementos */
void imprime (Lista* l)
{
        Lista* p; /* variável auxiliar para percorrer a lista */
        for (p = l; p != NULL; p = p->prox)
                printf("info = %d\n", p->info);
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas

- **Função que verifica se lista está vazia**
- Pode ser útil implementarmos uma função que verifique se uma lista está vazia ou não. A função recebe a lista e retorna 1 se estiver vazia ou 0 se não estiver vazia. Como sabemos, uma lista está vazia se seu valor é NULL. Uma implementação dessa função é mostrada a seguir:

```
/* função vazia: retorna 1 se vazia ou 0 se não vazia */
int vazia (Lista* l)
{
        if (l == NULL)
                return 1;
        else
                return 0;
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas

- **Função de busca**
- Outra função útil consiste em verificar se um determinado elemento está presente na lista. A função recebe a informação referente ao elemento que queremos buscar e fornece como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é NULL.

```
/* função busca: busca um elemento na lista */
Lista* busca (Lista* l, int v)
{
        Lista* p;
        for (p=l; p!=NULL; p=p->prox)
                if (p->info == v)
                        return p;
        return NULL; /* não achou o elemento */
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas

Função que retira um elemento da lista

Para completar o conjunto de funções que manipulam uma lista, devemos implementar uma função que nos permita retirar um elemento. A função tem como parâmetros de entrada a lista e o valor do elemento que desejamos retirar, e deve retornar o valor atualizado da lista, pois, se o elemento removido for o primeiro da lista, o valor da lista deve ser atualizado.

A função para retirar um elemento da lista é mais complexa. Se descobrirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer com que o novo valor da lista passe a ser o ponteiro para o segundo elemento, e então podemos liberar o espaço alocado para o elemento que queremos retirar. Se o elemento a ser removido estiver no meio da lista, devemos fazer com que o elemento anterior a ele passe a apontar para o elemento seguinte, e então podemos liberar o elemento que queremos retirar. Devemos notar que, no segundo caso, precisamos do ponteiro para o elemento anterior para podermos acertar o encadeamento da lista. As Figuras 9.4 e 9.5 ilustram as operações de remoção.

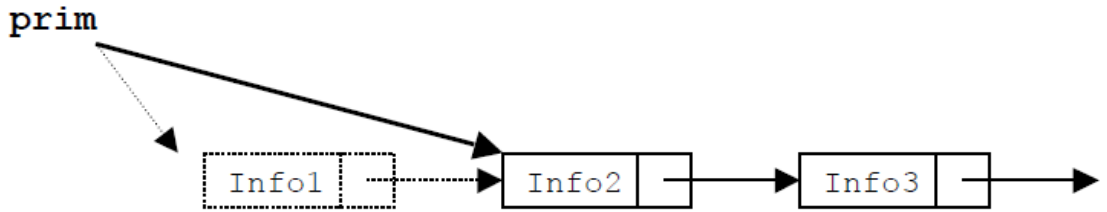


Figura 9.4: Remoção do primeiro elemento da lista.

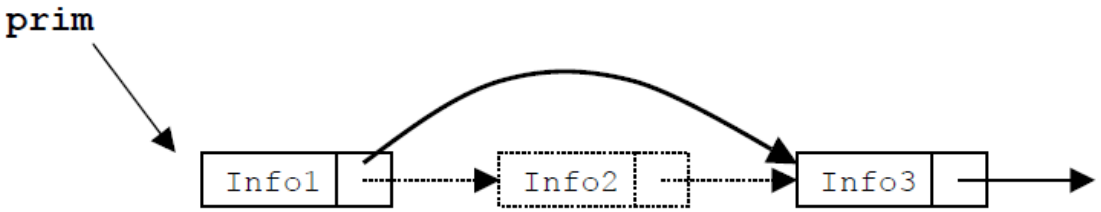


Figura 9.5: Remoção de um elemento no meio da lista.

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas

```
/* função retira: retira elemento da lista */
Lista* retira (Lista* l, int v) {
   Lista* ant = NULL; /* ponteiro para elemento anterior */
   Lista* p = l;      /* ponteiro para percorrer a lista*/

   /* procura elemento na lista, guardando anterior */
   while (p != NULL && p->info != v) {
      ant = p;
      p = p->prox;
   }

   /* verifica se achou elemento */
   if (p == NULL)
      return l;   /* não achou: retorna lista original */

   /* retira elemento */
   if (ant == NULL) {
      /* retira elemento do inicio */
      l = p->prox;
   }
   else {
      /* retira elemento do meio da lista */
      ant->prox = p->prox;
   }
   free(p);
   return l;
}
```

O caso de retirar o último elemento da lista recai no caso de retirar um elemento no meio da lista, conforme pode ser observado na implementação ao lado.

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas

Função para liberar a lista

Uma outra função útil que devemos considerar destrói a lista, liberando todos os elementos alocados. Uma implementação dessa função é mostrada abaixo. A função percorre elemento a elemento, liberando-os. É importante observar que devemos guardar a referência para o próximo elemento antes de liberar o elemento corrente (se liberássemos o elemento e depois tentássemos acessar o encadeamento, estaríamos acessando um espaço de memória que não estaria mais reservado para nosso uso).

```
void libera (Lista* l)
{
   Lista* p = l;
   while (p != NULL) {
       Lista* t = p->prox; /* guarda referência para o próximo elemento
*/
      free(p);             /* libera a memória apontada por p */
      p = t;               /* faz p apontar para o próximo */
   }
}
```

Um programa que ilustra a utilização dessas funções é mostrado a seguir.

```
#include <stdio.h>

int main (void) {
   Lista* l;           /* declara uma lista não iniciada */
   l = inicializa();   /* inicia lista vazia */
   l = insere(l, 23); /* insere na lista o elemento 23 */
   l = insere(l, 45); /* insere na lista o elemento 45 */
   l = insere(l, 56); /* insere na lista o elemento 56 */
   l = insere(l, 78); /* insere na lista o elemento 78 */
   imprime(l);         /* imprimirá: 78 56 45 23 */
   l = retira(l, 78);
   imprime(l);         /* imprimirá: 56 45 23 */
   l = retira(l, 45);
   imprime(l);         /* imprimirá: 56 23 */
   libera(l);
   return 0;
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Encadeadas

- **Manutenção da lista ordenada**

A função de inserção vista antes armazena os elementos na lista na ordem inversa à ordem de inserção, pois um novo elemento é sempre inserido no início da lista. Se desejarmos manter os elementos em ordem, cada novo elemento deve ser inserido na ordem correta. Para exemplificar, vamos considerar que queremos manter nossa lista de números inteiros em ordem crescente. A função de inserção, neste caso, tem a mesma assinatura da função de inserção mostrada, mas percorre os elementos da lista a fim de encontrar a posição correta para a inserção do novo. Com isto, temos que saber inserir um elemento no meio da lista.

```
/* função auxiliar: cria e inicializa um nó */
Lista* cria (int v)
{
   Lista* p = (Lista*) malloc(sizeof(Lista));
   p->info = v;
   return p;
}

/* função insere_ordenado: insere elemento em ordem */
Lista* insere_ordenado (Lista* l, int v)
{
   Lista* novo = cria(v); /* cria novo nó */
   Lista* ant = NULL;     /* ponteiro para elemento anterior */
   Lista* p = l;          /* ponteiro para percorrer a lista*/

   /* procura posição de inserção */
   while (p != NULL && p->info < v) {
      ant = p;
      p = p->prox;
   }

   /* insere elemento */
   if (ant == NULL) {   /* insere elemento no início */
      novo->prox = l;
      l = novo;
   }
   else {   /* insere elemento no meio da lista */
      novo->prox = ant->prox;
      ant->prox = novo;
   }
   return l;
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Circulares

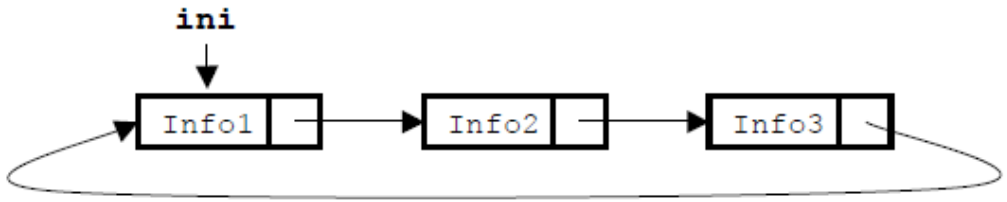


Figura 9.7: Arranjo da memória de uma lista circular.

Para percorrer os elementos de uma lista circular, visitamos todos os elementos a partir do ponteiro do elemento inicial até alcançarmos novamente esse mesmo elemento. O código abaixo exemplifica essa forma de percorrer os elementos. Neste caso, para simplificar, consideramos uma lista que armazena valores inteiros. Devemos salientar que o caso em que a lista é vazia ainda deve ser tratado (se a lista é vazia, o ponteiro para um elemento inicial vale `NULL`).

```
void imprime_circular (Lista* l)
{
   Lista* p = l;        /* faz p apontar para o nó inicial */
   /* testa se lista não é vazia */
   if (p) {
   {
      /* percorre os elementos até alcançar novamente o início */
      do {
         printf("%d\n", p->info);    /* imprime informação do nó */
         p = p->prox;                /* avança para o próximo nó */
      } while (p != l);
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Duplamente Encadeadas

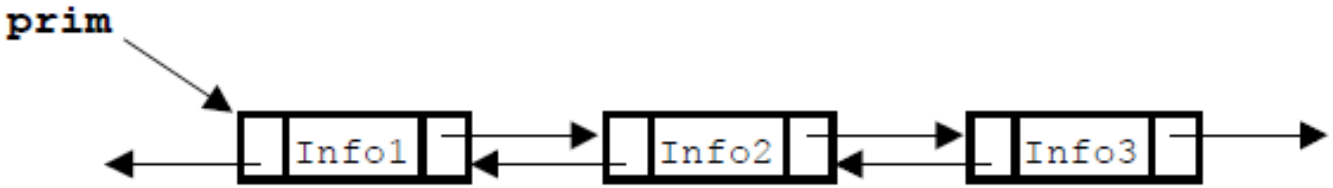


Figura 9.8: Arranjo da memória de uma lista duplamente encadeada.

Para exemplificar a implementação de listas duplamente encadeadas, vamos novamente considerar o exemplo simples no qual queremos armazenar valores inteiros na lista. O nó da lista pode ser representado pela estrutura abaixo e a lista pode ser representada através do ponteiro para o primeiro nó.

```
struct lista2 {
   int info;
   struct lista2* ant;
   struct lista2* prox;
};

typedef struct Lista2 Lista2;
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Duplamente Encadeadas

Função de inserção

O código a seguir mostra uma possível implementação da função que insere novos elementos no início da lista. Após a alocação do novo elemento, a função acertar o duplo encadeamento.

```
/* inserção no início */
Lista2* insere (Lista2* l, int v)
{
   Lista2* novo = (Lista2*) malloc(sizeof(Lista2));
   novo->info = v;
   novo->prox = l;
   novo->ant = NULL;
   /* verifica se lista não está vazia */
   if (l != NULL)
      l->ant = novo;
   return novo;
}
```

Nessa função, o novo elemento é encadeado no início da lista. Assim, ele tem como próximo elemento o antigo primeiro elemento da lista e como anterior o valor NULL. A seguir, a função testa se a lista não era vazia, pois, neste caso, o elemento anterior do então primeiro elemento passa a ser o novo elemento. De qualquer forma, o novo elemento passa a ser o primeiro da lista, e deve ser retornado como valor da lista atualizada. A Figura 9.9 ilustra a operação de inserção de um novo elemento no início da lista.

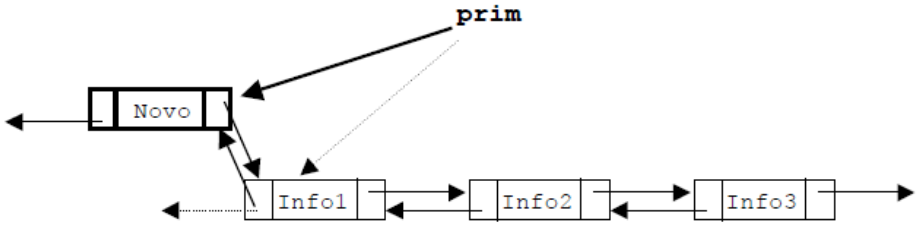


Figura 9.9: Inserção de um novo elemento no início da lista.

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Duplamente Encadeadas

Função de busca

A função de busca recebe a informação referente ao elemento que queremos buscar e tem como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é `NULL`.

```
/* função busca: busca um elemento na lista */
Lista2* busca (Lista2* l, int v)
{
   Lista2* p;
   for (p=l; p!=NULL; p=p->prox)
      if (p->info == v)
         return p;
   return NULL;       /* não achou o elemento */
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Duplamente Encadeadas

Função de busca

A função de busca recebe a informação referente ao elemento que queremos buscar e tem como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é `NULL`.

```
/* função busca: busca um elemento na lista */
Lista2* busca (Lista2* l, int v)
{
   Lista2* p;
   for (p=l; p!=NULL; p=p->prox)
      if (p->info == v)
         return p;
   return NULL;       /* não achou o elemento */
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Duplamente Encadeadas

Função que retira um elemento da lista

A função de remoção fica mais complicada, pois temos que acertar o encadeamento duplo. Em contrapartida, podemos retirar um elemento da lista conhecendo apenas o ponteiro para esse elemento. Desta forma, podemos usar a função de busca acima para localizar o elemento e em seguida acertar o encadeamento, liberando o elemento ao final.

Se `p` representa o ponteiro do elemento que desejamos retirar, para acertar o encadeamento devemos conceitualmente fazer:

```
p->ant->prox = p->prox;
p->prox->ant = p->ant;
```

isto é, o anterior passa a apontar para o próximo e o próximo passa a apontar para o anterior. Quando `p` apontar para um elemento no meio da lista, as duas atribuições acima são suficientes para efetivamente acertar o encadeamento da lista. No entanto, se `p` for um elemento no extremo da lista, devemos considerar as condições de contorno. Se `p` for o primeiro, não podemos escrever `p->ant->prox`, pois `p->ant` é `NULL`; além disso, temos que atualizar o valor da lista, pois o primeiro elemento será removido.

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Duplamente Encadeadas

Uma implementação da função para retirar um elemento é mostrada a seguir:

```
/* função retira: retira elemento da lista */
Lista2* retira (Lista2* l, int v) {
   Lista2* p = busca(l,v);

   if (p == NULL)
      return l;   /* não achou o elemento: retorna lista inalterada */

   /* retira elemento do encadeamento */
   if (l == p)
      l = p->prox;
   else
      p->ant->prox = p->prox;

   if (p->prox != NULL)
      p->prox->ant = p->ant;

   free(p);

   return l;
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Duplamente Encadeadas

Lista circular duplamente encadeada

Uma lista circular também pode ser construída com encadeamento duplo. Neste caso, o que seria o último elemento da lista passa ter como próximo o primeiro elemento, que, por sua vez, passa a ter o último como anterior. Com essa construção podemos percorrer a lista nos dois sentidos, a partir de um ponteiro para um elemento qualquer. Abaixo, ilustramos o código para imprimir a lista no sentido reverso, isto é, percorrendo o encadeamento dos elementos anteriores.

```
void imprime_circular_rev (Lista2* l)
{
   Lista2* p = l;         /* faz p apontar para o nó inicial */
   /* testa se lista não é vazia */
   if (p) {
   {
      /* percorre os elementos até alcançar novamente o início */
      do {
         printf("%d\n", p->info);    /* imprime informação do nó */
         p = p->ant;                 /* "avança" para o nó anterior */
      } while (p != l);
}
```

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Duplamente Encadeadas

10.2. Implementações recursivas

Uma lista pode ser definida de maneira recursiva. Podemos dizer que uma lista encadeada é representada por:

uma lista vazia; ou
um elemento seguido de uma (sub-)lista.

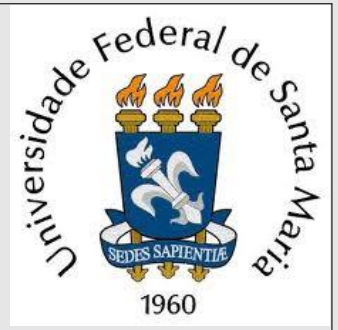
Neste caso, o segundo elemento da lista representa o primeiro elemento da sub-lista. Com base na definição recursiva, podemos implementar as funções de lista recursivamente. Por exemplo, a função para imprimir os elementos da lista pode ser re-escrita da forma ilustrada abaixo:

```
/* Função imprime recursiva */
void imprime_rec (Lista* l)
{
   if (vazia(l))
      return;
   /* imprime primeiro elemento */
   printf("info: %d\n",l->info);
   /* imprime sub-lista */
   imprime_rec(l->prox);
}
```

É fácil alterarmos o código acima para obtermos a impressão dos elementos da lista em ordem inversa: basta invertermos a ordem das chamadas às funções `printf` e `imprime_rec`.

ESTRUTURAS LINEARES E ENCADEADAS

Estrutura Dinâmica- Listas Duplamente Encadeadas



A função para retirar um elemento da lista também pode ser escrita de forma recursiva. Neste caso, só retiramos um elemento se ele for o primeiro da lista (ou da sub-lista). Se o elemento que queremos retirar não for o primeiro, chamamos a função recursivamente para retirar o elemento da sub-lista.

```
/* Função retira recursiva */
Lista* retira_rec (Lista* l, int v)
{
   if (vazia(l))
      return l;   /* lista vazia: retorna valor original */

   /* verifica se elemento a ser retirado é o primeiro */
   if (l->info == v) {
      Lista* t = l;     /* temporário para poder liberar */
      l = l->prox;
      free(t);
   }
   else {
      /* retira de sub-lista */
      l->prox = retira_rec(l->prox,v);
   }
   return l;
}
```

A função para liberar uma lista também pode ser escrita recursivamente, de forma bastante simples. Nessa função, se a lista não for vazia, liberamos primeiro a sub-lista e depois liberamos a lista.

```
void libera_rec (Lista* l)
{
   if (!vazia(l))
   {
      libera_rec(l->prox);
      free(l);
   }
}
```