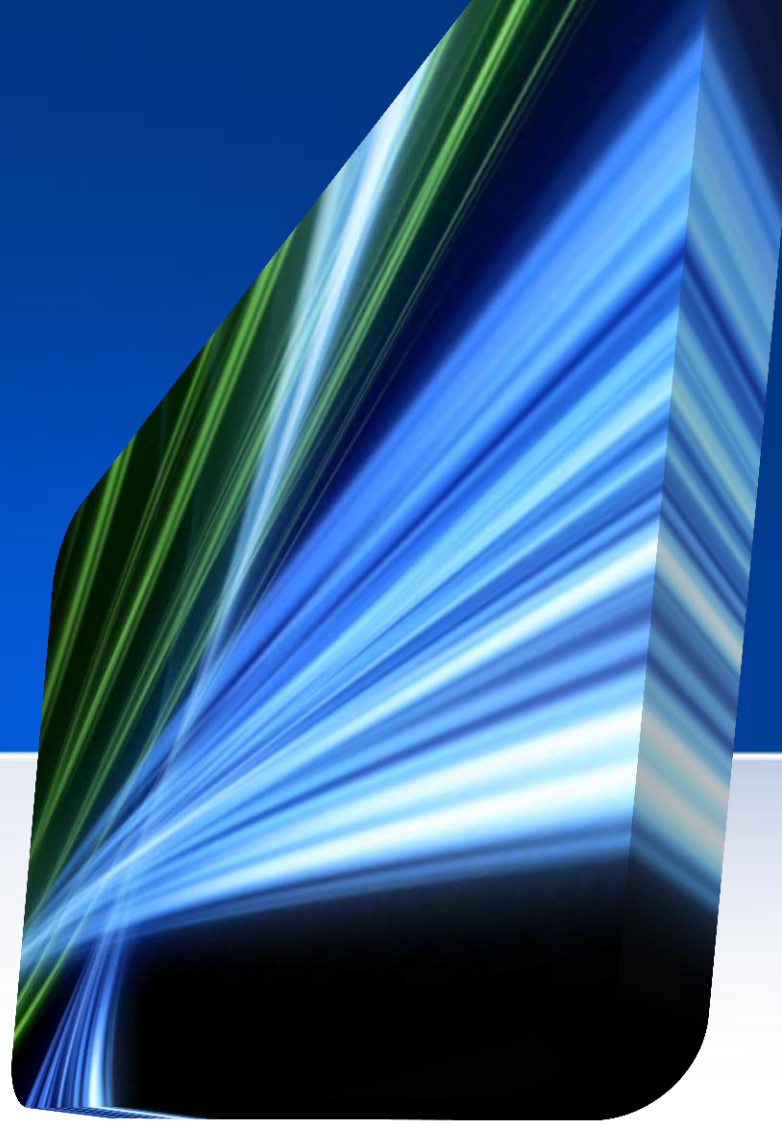
LISTAS ENCADEADAS SIMPLES



ED – 2014

Faculdade São Luís

Professor Emanuel

INTRODUÇÃO

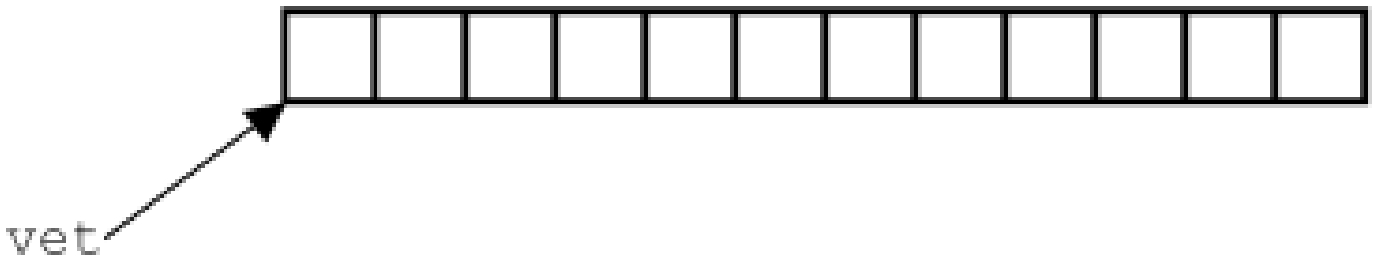
Para representarmos um grupo de dados, já vimos que podemos usar um vetor em C. O vetor é a forma mais primitiva de representar diversos elementos agrupados. Para simplificar a discussão dos conceitos que serão apresentados agora, vamos supor que temos que desenvolver uma aplicação que deve representar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

INTRODUÇÃO

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.



INTRODUÇÃO

O fato de o vetor ocupar um espaço contíguo na memória nos permite acessar qualquer um de seus elementos a partir do ponteiro para o primeiro elemento. De fato, o símbolo vet, após a declaração acima, como já vimos, representa um ponteiro para o primeiro elemento do vetor, isto é, o valor de vet é o endereço da memória onde o primeiro elemento do vetor está armazenado.

INTRODUÇÃO

De posse do ponteiro para o primeiro elemento, podemos acessar qualquer elemento do vetor através do operador de indexação vet[i].

Dizemos que o vetor é uma estrutura que possibilita acesso randômico aos elementos, pois podemos acessar qualquer elemento aleatoriamente.

INTRODUÇÃO

No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos.

INTRODUÇÃO

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 sub-utilizando o espaço de memória reservado.

INTRODUÇÃO

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.

LISTA ENCADEADA

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado.

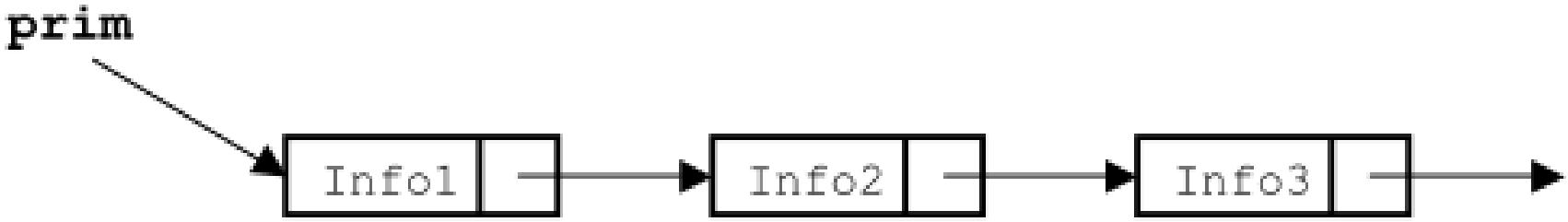
LISTA ENCADEADA

No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista.

LISTA ENCADEADA

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. A figura ilustra o arranjo da memória de uma lista encadeada.

LISTA ENCADEADA



LISTA ENCADEADA

A estrutura consiste numa sequência encadeada de elementos, em geral chamados de nós da lista. A lista é representada por um ponteiro para o primeiro elemento (ou nó).

Do primeiro elemento, podemos alcançar o segundo seguindo o encadeamento, e assim por diante. O último elemento da lista aponta para NULL, sinalizando que não existe um próximo elemento.

LISTA ENCADEADA

Para exemplificar a implementação de listas encadeadas em C, vamos considerar um exemplo simples em que queremos armazenar valores inteiros numa lista encadeada. O nó da lista pode ser representado pela estrutura abaixo:

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

LISTA ENCADEADA

Devemos notar que trata-se de uma estrutura auto-referenciada, pois, além do campo que armazena a informação (no caso, um número inteiro), há um campo que é um ponteiro para uma próxima estrutura do mesmo tipo. Embora não seja essencial, é uma boa estratégia definirmos o tipo Lista como sinônimo de struct lista.

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

```
/* função de inicialização: retorna uma lista vazia */

Lista* inicializa (void)
{
   return NULL;
}
```

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.

FUNÇÃO DE INSERÇÃO

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.

FUNÇÃO DE INSERÇÃO

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.

FUNÇÃO DE INSERÇÃO

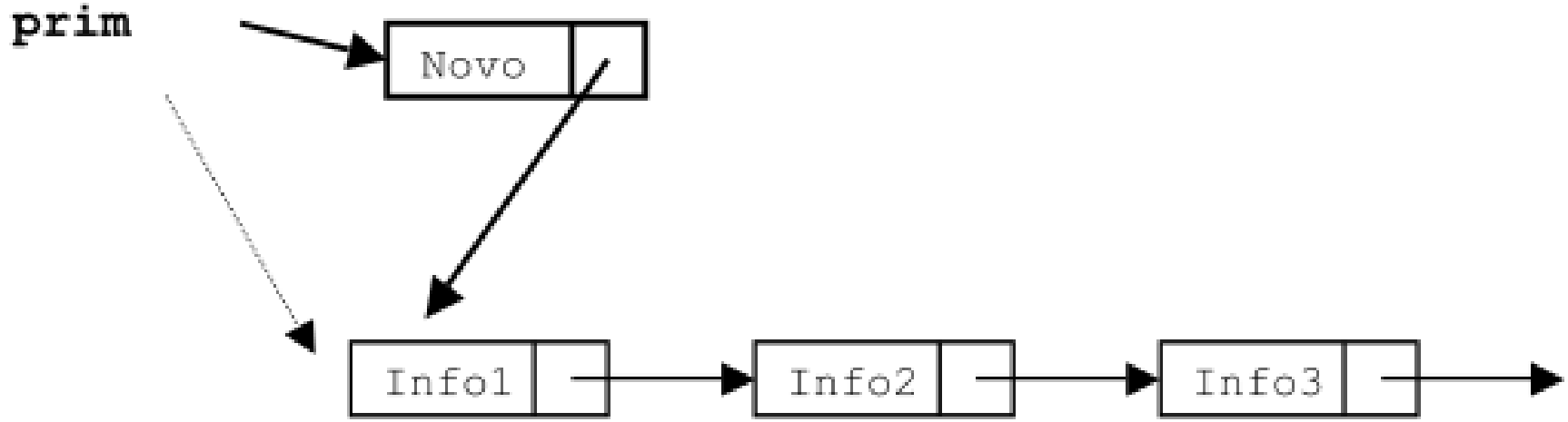
```
/* inserção no início: retorna a lista atualizada */

Lista* insere (Lista* l, int i)
{
  Lista* novo = (Lista*) malloc(sizeof(Lista));
  novo->info = i;
  novo->prox = l;
  return novo;
}
```

FUNÇÃO DE INSERÇÃO

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 ilustra a operação de inserção de um novo elemento no início da lista.

FUNÇÃO DE INSERÇÃO



FUNÇÃO DE INSERÇÃO

A seguir, ilustramos um trecho de código que cria uma lista inicialmente vazia e insere nela novos elementos.

```
int main (void)
{
  Lista* l;        /* declara uma lista não inicializada */
  l = inicializa();  /* inicializa lista como vazia */
  l = insere(l, 23); /* insere na lista o elemento 23 */
  l = insere(l, 45); /* insere na lista o elemento 45 */
  ...
  return 0;
}
```

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 QUE PERCORRE OS ELEMENTOS DA LISTA

```
/* 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);
}
```