

SIN 211 - Algoritmos e Estruturas de Dados

(Listas Dinâmicas)

Prof^o: Joelson Antônio dos Santos

Universidade Federal de Viçosa
Instituto de Ciências Exatas e Tecnológicas
Campus de Rio Paranaíba - MG

joelsonn.santos@gmail.com
Sala: BBT 233

3 de abril de 2018

Aula de Hoje

- 1 Lista Linear
 - Implementação dinâmica

Créditos

O material desta aula é composto por adaptações e extensões dos originais gentilmente cedidos pelos professores **Moacir Pereira Ponti** e **Rachel Reis**.

Alocação

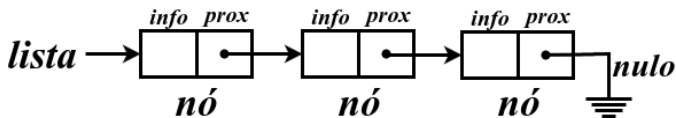
- Estática
 - Sequencial
- Dinâmica
 - Blocos de memórias são denominados **nós** ou **células**.
 - Itens ocupam nós espalhados por toda a memória.

Limitações - Lista Estática

- Quais as principais limitações de se utilizar listas lineares estáticas?
- Listas estáticas podem ser mais eficientes em quais situações?

Listas Encadeadas (simples)

- Listas Encadeadas ou (listas ligadas).
- **Definição:** Estrutura de dados que mantém uma coleção de itens em ordem linear sem exigir que eles ocupem posições consecutivas de memória.
- **Itens** são armazenados em **nós**.
- Cada nó possui dois campos:
 - Informação
 - Endereço para o próximo nó



Listas Encadeadas

- Vantagens:
 - É relativamente fácil inserir e remover elementos em posições aleatórias.
 - Essas atividades geram pouca movimentação de dados na memória.
 - Útil em aplicações cujas quais não necessitam previamente de definição de tamanho máximo da lista.

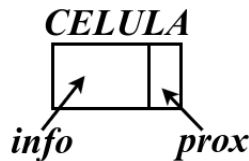
Listas Encadeadas

- Desvantagens:
 - Pode consumir mais tempo de preparação do sistema para alocação e liberação de memória.
 - O acesso a um elemento no meio da lista é mais custoso se comparado ao acesso direto (pelo índice) oferecido pela implementação estática.

Listas Encadeadas (Como implementar?)

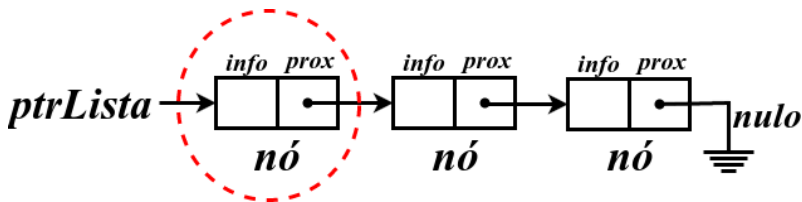
- A primeira estrutura a ser implementada é o **nó** ou **célula**.
- Considerando que a informação armazenada para a lista seja do tipo **inteiro**, tem-se:

```
typedef struct sCell{  
    int info;  
    struct sCell *prox;  
}CELULA;
```



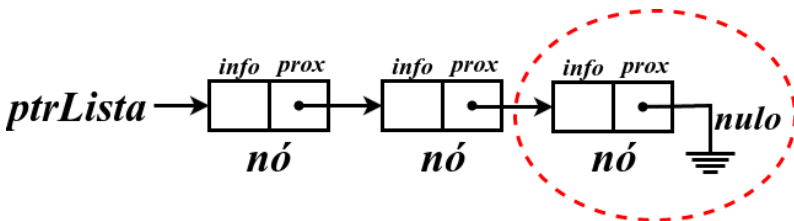
Listas Encadeadas (Como implementar?)

- Uma célula pode ser declarada como:
`CELULA c;`
- Assim tem-se:
 - **c.info:** conteúdo de uma célula.
 - **c.prox:** endereço para a próxima célula.



Listas Encadeadas (Como implementar?)

- O campo do próximo endereço (prox) do último elemento da lista contém o valor do tipo **NULL**, indicando que não é um endereço válido.
- Esse ponteiro nulo é usado para identificar o final de uma lista.

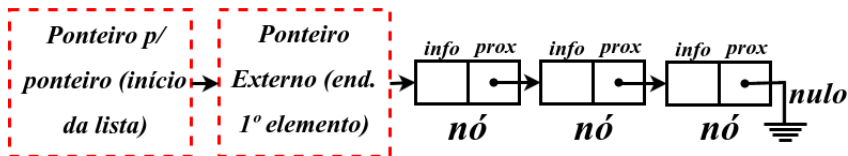


Lista Encadeadas (Como implementar?)

- Dentro de funções (Operações comuns em listas).
 - Cada elemento (nó) é tratado como um ponteiro alocado dinamicamente de acordo com a necessidade.
 - O **ponteiro externo** que aponta para o 1º elemento da lista é apontado por um **ponteiro para ponteiro** para facilitar a manipulação do endereço do 1º elemento da lista.

Listas Encadeadas (Como implementar?)

- Estrutura final da lista encadeada é:



- **Início da lista:** Ponteiro para ponteiro, pois armazena o endereço de um ponteiro.
- **Cada nó** da lista é um ponteiro.

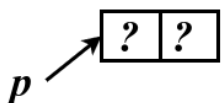
Algumas Operações

- Criar célula/nó;
- Inicializar lista;
- Inserir no início;
- Inserir no fim;
- Remover no início;
- Remover no fim;
- Pesquisar um elemento;
- Imprimir elementos da lista;
- Destruir lista;

Criar célula/nó

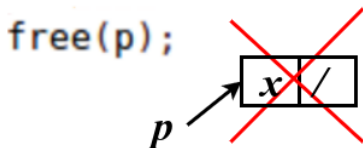
- Alocar dinamicamente um nó vazio para uma lista encadeada.

```
CELULA* criarCelula(){  
    CELULA *nova = (CELULA*)malloc(sizeof(CELULA));  
    return nova;  
}  
  
// dentro de alguma função  
CELULA *p = criarCelula();
```



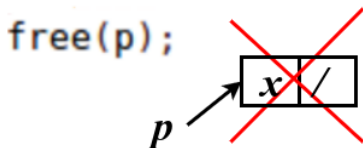
Criar célula/nó

- **p** é o ponteiro que armazena o endereço do nó a ser liberado.



Criar célula/nó

- **p** é o ponteiro que armazena o endereço do nó a ser liberado.




- Por medidas de segurança dos dados, atribuir **NULL** a qualquer ponteiro depois de liberá-lo (usando `free()`) é considerado uma boa prática de programação.

Inicialização da lista

- Inicializar o ponteiro externo à lista com o valor **NULL**.
- O endereço onde o nosso ponteiro para ponteiro aponta receberá o valor **NULL**.
- **lista** é o ponteiro para o ponteiro externo da lista, ou seja, **lista** armazena o endereço do ponteiro que armazena o endereço da primeira célula da lista.

```
void inicializarLista(CELULA **lista){  
    (*lista) = NULL;  
}
```

Verificação de Lista Vazia



```
int listaVazia(CELULA **lista){  
    if((*lista) == NULL){  
        return 1;  
    }  
    return 0;  
}
```


Inserir Elemento no Início (Parte 1)

```
int inserirInicio(CELULA** lista, int elemento){
    CELULA* p = criarCelula(); // cria um novo nó.

    if(p == NULL){
        printf("\n Erro de alocação de memória!");
        return 0; // falha na inserção
    }

    // coloca a informação em p->info
    p->info = elemento;
```

Inserir Elemento no Início (Parte 2)



```
// verifica se lista está vazia
if(listaVazia(lista)){
    (*lista) = p;
    return 1; // inseriu com sucesso
}
p->prox = (*lista);
(*lista) = p;
return 1; // inseriu com sucesso
}
```


Inserir Elemento no Final (Parte 1)

```
int inserirFim(CELULA** lista, int elemento){
    CELULA* p = criarCelula(); // cria um novo nó.
    if(p == NULL){
        printf("\n Erro de alocação de memória!");
        return 0; // falha na inserção
    }
    // coloca a informação em p->info
    p->info = elemento;
```

Inserir Elemento no Final (Parte 2)

```
// verifica se lista está vazia
if(listaVazia(lista)){
    (*lista) = p;
    return 1; // inseriu com sucesso
}
// Cria ponteiro auxiliar para a lista
CELULA* aux = (*lista);
// Percorre a lista usando ponteiro aux.
while(aux->prox != NULL){
    aux = aux->prox;
}
aux->prox = p;
return 1; // inseriu com sucesso
```


Imprimir Lista



```
void imprimirLista(CELULA** lista){
    CELULA *aux = (*lista);
    if(listaVazia(lista)){
        printf("\n Lista vazia!");
    }else{
        printf("\n Lista: ");
        while(aux != NULL){
            // imprime informação
            printf(" %d ", aux->info);
            aux = aux->prox;
        }
        printf("\n");
    }
}
```

Função Principal - Testes

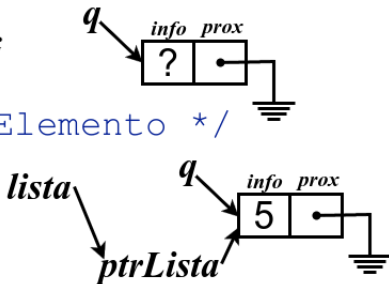
- O que falta neste programa?

```
int main(){  
  
    CELULA *ptrLista;  
    inicializarLista(&ptrLista);  
    inserirInicio(&ptrLista, 1);  
    imprimirLista(&ptrLista);  
    inserirInicio(&ptrLista, 2);  
    imprimirLista(&ptrLista);  
    inserirInicio(&ptrLista, 3);  
    imprimirLista(&ptrLista);  
    inserirFim(&ptrLista, 4);  
    imprimirLista(&ptrLista);  
    inserirFim(&ptrLista, 5);  
    imprimirLista(&ptrLista);  
  
    return 0;  
}
```

Exemplo

- O que acontece quando criamos o primeiro nó?

```
CELULA *q;  
q = criarCelula();  
if (q != NULL) {  
    q->info = 5; /* Elemento */  
}  
(*lista) = q;
```

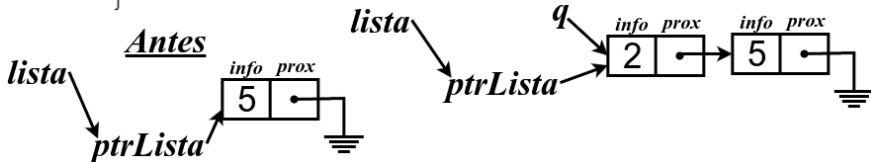


→ Lembrando que *q* armazena o endereço do novo nó e *lista* armazena o endereço do ponteiro externo que aponta para o primeiro nó da lista encadeada.

Exemplo

- Inserindo no início.

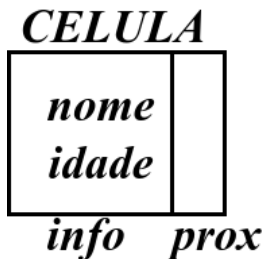
```
q = criarCelula();  
if (q != NULL)  
{  
    q->info = 2; /* Elemento */  
    q->prox = (*lista);  
    (*lista) = q;  
}
```



Lista Encadeada

- A partir de agora, assume-se que o tipo de dado manipulado (campo **info**) pela lista não seja apenas um **int**.
- Assumindo que exista uma estrutura de dados do tipo **Pessoa** com os campos **nome** e **idade**. Como ficaria o tipo abstrato de dados *lista encadeada* para esse tipo de dado?

Lista Encadeada



```
typedef struct sPessoa{
    char nome[50];
    int idade;
}PESSOA;

typedef struct sCell{
    PESSOA info;
    struct sCell *prox;
}CELULA;
```

Lista Encadeada

- O que mudaria nas operações (funções) vistas anteriormente?

Lista Encadeada

- **Exemplo:**

```
void inserirInicio(CELULA** lista, PESSOA elemento){  
    // código aqui...  
}
```


Lista Encadeada - Exemplo

- Imprimir o campo **info** da lista.

```
void imprimirLista(CELULA** lista){
    CELULA *aux = (*lista);
    if(listaVazia(lista)){
        printf("\n Lista vazia!");
    }else{
        printf("\n Lista: ");
        while(aux != NULL){
            imprimirInfo(aux->info);
            aux = aux->prox;
        }
        printf("\n");
    }
}
```

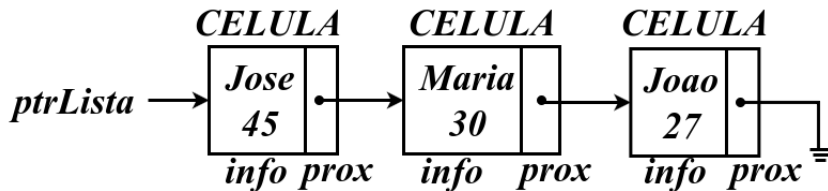
```
void imprimirInfo(PESSOA elemento){
    printf("\n nome: %s ", elemento.nome);
    printf("\n idade: %d \n", elemento.idade);
}
```

Lista Encadeada - Exemplo

- 1 Assumindo que a operação de *inserirInicio(...)* esteja implementada para o tipo de dado **PESSOA** de uma determinada lista.
- 2 Assumindo ainda que a lista esteja vazia.
- 3 Neste contexto, qual será a disposição dos elementos $\{("Joao", 27), ("Maria", 30) \text{ e } ("Jose", 45)\}$ inseridos nesta ordem pela função do primeiro item?

Lista Encadeada - Exemplo

- Solução:



Remover célula no início - (Parte 1)

```
PESSOA removerInicio(CELULA **lista){
    /* ponteiro para armazenar o endereço
       da célula a ser removida. */
    CELULA* removida;
    // variável para armazenar elemento removido.
    PESSOA elementoRemovido;

    if(listaVazia(lista)){
        printf("\n Erro: Lista vazia!");
        // elemento vazio a ser retornado
        strcpy(elementoRemovido.nome, " ");
        elementoRemovido.idade = -1;
        return elementoRemovido;
    }
}
```

Remover célula no início - (Parte 2)

```
// armazena primeiro elemento
removida = (*lista);
elementoRemovido = removida->info;
// lista aponta para o 2º elemento
(*lista) = (*lista)->prox;

// libera memória em relação ao 1º elemento
free(removida);
return elementoRemovido;
}
```

Remover célula no fim - (Parte 1)

```
PESSOA removerFim(CELULA** lista){
    /* ponteiro para armazenar o endereço
    da célula a ser removida. */
    CELULA* removida;
    // marca a célula que será a nova última
    CELULA* anterior;
    // variável para armazenar elemento removido.
    PESSOA elementoRemovido;

    if(listaVazia(lista)){
        printf("\n Erro: Lista vazia!");
        // elemento vazio a ser retornado
        strcpy(elementoRemovido.nome, " ");
        elementoRemovido.idade = -1;
        return elementoRemovido;
    }
}
```

Remover célula no fim - (Parte 2)



```
// verifica se há apenas 1 elemento
if((*lista)->prox == NULL){
    return removerInicio(lista);
}

removida = (*lista);
while(removida->prox != NULL){
    anterior = removida;
    removida = removida->prox;
}

elementoRemovido = removida->info;
anterior->prox = NULL;
free(removida);
return elementoRemovido;
```

Pesquisar Elemento pelo Nome

- Como seria uma pesquisa na lista pelo nome da pessoa?

Pesquisar Elemento pelo Nome

```
CELULA* pesquisarPessoa(CELULA** lista, char* nome){
    CELULA* aux;

    if(listaVazia(lista)){
        printf("\n Lista vazia!");
        return NULL;
    }
    // recebe endereço do 1º elemento da lista
    aux = (*lista);
    while(aux != NULL){
        // se strings iguais, retorna aux
        if(strcmp(aux->info.nome, nome) == 0){
            return aux;
        }
        /* se elemento não encontrado,
        continua percorrendo...*/
        aux = aux->prox;
    }
    return NULL;
}
```

Remoção pelo Nome

- Como seria uma remoção na lista pelo nome da pessoa?


Remoção pelo Nome - (Parte 1)

```
PESSOA removerNome(CELULA** lista, char* nome){
    CELULA* removida;
    CELULA* anterior;

    PESSOA elementoRemovido;
    strcpy(elementoRemovido.nome, " ");
    elementoRemovido.idade = -1;

    if(listaVazia(lista)){
        printf("\n Lista Vazia!");
        return elementoRemovido;
    }
    // pesquisa pelo nome
    removida = pesquisarPessoa(lista, nome);
    // nome não encontrado
    if(removida == NULL){
        printf("\nErro: Nome não encontrado!");
        return elementoRemovido;
    }
}
```

Remoção pelo Nome - (Parte 2)



```
// caso o elemento seja o primeiro
if(removida == (*lista)){
    return removerInicio(lista);
}
elementoRemovido = removida->info;
anterior = (*lista);
// encontra célula anterior a que será removida
while(anterior->prox != removida){
    anterior = anterior->prox;
}
// movimenta o ponteiro. O anterior deve apontar
// para onde o removida aponta.
anterior->prox = removida->prox;
free(removida);
return elementoRemovido;
```

Bibliografia Básica

- TENENBAUM A., LANGSAM Y. e AUGENSTEIN M. J. Estrutura de Dados usando C. Editora Makron, 1995.
 - Páginas 223 à 229
 - Pág 231 (“Operações getnode e freenode”) – até pág. 233.
 - Pág 256 (“Listas ligadas usando variáveis dinâmicas”) – até pág. 258
 - Pág 260 (“Exemplos de operações de listas em C”) – até pág. 262