

# SIN 211 - Algoritmos e Estruturas de Dados

(Lista Linear Estática)

Prof<sup>o</sup>: 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*

27 de março 2018

# Aula de Hoje

## 1 Tipos Abstratos de Dados (TAD)

## 2 Estruturas de Dados

- Lista Linear

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

# Tipo de Dado

- Conjunto de valores/domínio que uma variável pode assumir.
- **Exemplo:** conjunto dos números inteiros
  - ..., -2, -1, 0, +1, +2, ...
- Estruturas de dados
  - Tipos de dados que possuem alguma relação entre si.

```
typedef struct sAluno{  
    char nome[30];  
    float notaFinal;  
}Aluno;
```

# Tipo Abstrato de Dados (TAD)

- Um conjunto de valores e uma sequência de operações sobre estes valores.
- Uma abstração matemática de um modelo básico.
- Além da estrutura, também devemos atribuir operações que são convenientes para o nosso *TAD*.

```
typedef struct sAluno{  
    char nome[30];  
    float notaFinal;  
}Aluno;
```

# Tipo Abstrato de Dados (TAD)

- Quais as vantagens de utilizarmos *TAD*?
  - Encapsulamento
  - Segurança
  - Flexibilidade
  - Reutilização

# Tipo Abstrato de Dados (TAD)

- A principal ideia é a criação de novos tipos.
- O usuário não tem detalhes das implementações das operações, apenas da utilidade de cada função.
- **Exemplo:**
  - *TAD*: Arquivo em C.
  - Estrutura: `FILE *f`;
  - Operações: `fopen()`, `fclose()`, etc;

# Tipo Abstrato de Dados (TAD)

- Como implementar um *TAD*?
  - Dividir em módulos, criar um arquivo de cabeçalho e chamá-lo no código.
- Criação de dois arquivos
  - **Cabeçalho:** “arquivo.h”: protótipo de funções, tipos dos ponteiros, dados globais.
  - **Código de implementação:** “arquivo.c”: declaração do tipo de dados e implementação das funções.



# Exemplo *TAD* - Arquivo Header .h

```
1 // aluno.h
2 typedef struct sAluno ALUNO;
3
4 // protótipos de funções
5 ALUNO* cria_aluno(char nome[], float n);
6
7 void deteta_aluno(ALUNO* a);
8
9 void acessa_aluno(ALUNO* a, char nome[], float* n);
```

# Exemplo *TAD* - Arquivo .c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include "aluno.h"
5
6  typedef struct sAluno{
7      char nome[30];
8      float notaFinal;
9  }ALUNO;
10
11  // aloca e retorna um aluno
12  ALUNO* cria_aluno(char nome[], float n){
13      ALUNO* a = (ALUNO*) malloc(sizeof(ALUNO));
14      if(a != NULL){
15          strcpy(a->nome, nome);
16          a->notaFinal = n;
17      }
18      return a;
19  }
20
21  // libera a memória para um aluno
22  void deleta_aluno(ALUNO* a){
23      free(a);
24  }
25
26  void acessa_aluno(ALUNO* a, char nome[], float *n){
27      *n = a->notaFinal;
28      strcpy(nome, a->nome);
29  }
```

# Exemplo *TAD* - (Cont.)

```
31  int main(){
32      float notaTemp;
33      ALUNO *a1, *a2;
34      char nomeTemp[30];
35
36      a1 = cria_aluno("Jose", 6.0);
37      a2 = cria_aluno("Maria", 8.5);
38
39      acessa_aluno(a1, nomeTemp, &notaTemp);
40      printf("\nA nota de %s e %f", nomeTemp, notaTemp);
41      acessa_aluno(a2, nomeTemp, &notaTemp);
42      printf("\nA nota de %s e %f", nomeTemp, notaTemp);
43
44      return 0;
45  }
```

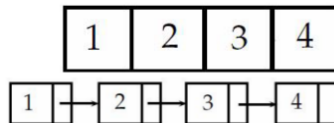
# *TAD* - Implementação

- Tudo que está implementado no arquivo “.h” é “transferido” para o arquivo “.c” durante a chamada do “include”.
- Os arquivos devem estar na mesma pasta.
- Outra forma de implementar é deixar apenas a função principal e funções que não manipulam a *TAD* no arquivo “.c”.

# TAD - Clássicos

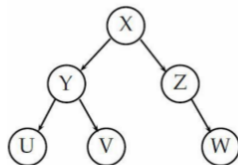
- Listas

- Sequenciais, encadeadas.



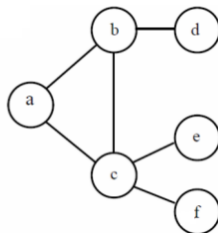
- Árvores

- Problemas de hierarquia.



- Grafos

- Redes, percursos, relacionamentos.



# Lista Linear

- Estruturas flexíveis, que podem aumentar ou diminuir seu tamanho e número de elementos de acordo com a demanda.
- É possível:
  - Inserir, remover e acessar elementos.
- Ideal quando não é possível prever a demanda por memória (caso seja implementada dinamicamente).

# Lista Linear

- Definição:
  - Uma lista é uma sequência de zero ou mais elementos  $a_1, a_2, \dots, a_n$  onde:
    - Todos os  $a_i$  elementos são de um mesmo tipo.
    - $n$  é o tamanho da lista linear.
- Propriedade fundamental:
  - $a_i$  precede  $a_{i+1}$  e sucede  $a_{i-1}$ .
  - $a_1$  é o primeiro elemento da lista.
  - $a_n$  é o último elemento da lista.

# Lista Linear

- Tipos de implementação:
  - **Lista estática:** *Arrays* (Vetores) são utilizados.
  - **Lista dinâmica:** Ponteiros são utilizados.
- Qual das implementações é mais eficiente? Sempre?



# Lista Linear

- Como visto, para implementarmos uma lista *TAD* devemos:
  - Definir a estrutura de dados.
  - Definir suas operações.
- Existe um único conjunto de operações que seja adequado para todas as aplicações?

# Lista Linear

- Operações comuns:
  - Inicialização da lista
  - Acesso a um elemento
  - Buscar um elemento
  - Remover um elemento
  - Inverter a lista
  - Ordenar a lista
  - Concatenar duas listas
  - Inserir novo elemento
  - Verificar se a lista está cheia
  - Verificar se a lista está vazia
  - Modificar um elemento existente
  - Determinar o tamanho da lista

# Lista Linear Estática

- Itens estão em posições consecutivas na memória.
- Lista pode ser percorrida em qualquer direção.
- Inserção de elementos:
  - **Desordenada:** insere o elemento na última posição.
  - **Ordenada:** inserir um elemento na posição  $a_i$ , causa deslocamento dos itens localizados após o ponto de inserção. O inverso ocorre ao se remover o elemento da posição  $a_i$ .

# Lista Linear Estática - Implementação

- Definição da lista linear estática:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define MAX_TAM 10
4
5  typedef struct sLista{
6      char elem[MAX_TAM];
7      int ultimo;
8  }Lista;
```

- ultimo** é utilizado para representar o índice do último elemento inserido na lista (vetor nesta representação gráfica).



ultimo = 3

# Lista Linear Estática - Implementação

- Inicializar a lista:

```
void inicializarLista(Lista* lista){  
    lista->ultimo = -1;  
}
```

- A variável **ultimo** é inicializada com valor -1 para indicar que a lista está vazia.
- Podemos usar o valor 0 na variável “**ultimo**”?

# Lista Linear Estática - Implementação

- Formas para criar a variável do tipo **Lista**:

```
Lista* lista = (Lista*)malloc(sizeof(Lista));  
inicializarLista(lista);
```

```
// Ou
```

```
Lista lista;  
inicializarLista(&lista);
```




# Lista Linear Estática - Implementação

- Verificar se a lista está cheia:

```
int listaCheia(Lista *lista){  
    if(lista->ultimo == (MAX_TAM - 1)){  
        return 1;  
    }  
    return 0;  
}
```

# Lista Linear Estática - Implementação

- Verificar se a lista está vazia:



```
int listaVazia(Lista *lista){  
    if(lista->ultimo == -1){  
        return 1;  
    }  
    return 0;  
}
```



# Lista Linear Estática - Implementação

- Inserir um elemento no fim da lista:

```
int inserirFim(Lista* lista, char elemento){  
    // verifica se é possível inserir  
    if(listaCheia(lista)){  
        printf("\nErro: Lista cheia!");  
        return 0;  
    }  
    // insere o elemento na última posição  
    lista->elem[lista->ultimo+1] = elemento;  
    // atualiza o indicador da última posição  
    lista->ultimo = (lista->ultimo + 1);  
  
    return 1;  
}
```

# Lista Linear Estática - Implementação

- Modificar um elemento existente:

```
int modificarElemento(Lista* lista, int posicao, char elem){  
    // verifica se posição é válida  
    if(posicao < 0 || posicao > lista->ultimo){  
        printf("\nErro: Posicao invalida!");  
        return 0;  
    }  
    // substituir o elemento  
    lista->elem[posicao] = elem;  
    return 1;  
}
```

# Lista Linear Estática - Implementação

- Remover um elemento da posição  $k$ :

```
int removerElemento(Lista* lista, int posicao){
    int i;
    if(listaVazia(lista)){
        printf("\nErro: Lista vazia!");
        return 0;
    }
    if(posicao < 0 || posicao > lista->ultimo){
        printf("\nErro> Posicao invalida!");
        return 0;
    }

    // mover os elementos posteriores ao removido
    for(i = posicao; i < lista->ultimo; i++){
        lista->elem[i] = lista->elem[i + 1];
    }
    // atualiza o indicador do último elemento
    lista->ultimo--;
    return 1; // removeu com sucesso
}
```

# Lista Linear Estática - Implementação

- Acesso ao elemento da posição  $k$ :

```
char acessarElemento(Lista* lista, int posicao){  
    if(listaVazia(lista)){  
        printf("\nErro: Lista vazia!");  
        return ' ';  
    }  
    if(posicao < 0 || posicao > lista->ultimo){  
        printf("\nErro: Posicao invalida!");  
        return ' ';  
    }  
    return lista->elem[posicao];  
}
```

# Lista Linear Estática - Implementação

- Pesquisa um elemento na lista:

```
int pesquisarElemento(Lista* lista, char elem){
    int i;
    if(listaVazia(lista)){
        printf("\nErro: Lista vazia!");
        return 0;
    }

    // percorre todos os elementos da lista
    for(i = 0; i <= lista->ultimo; i++){
        if(lista->elem[i] == elem){
            return i; // retorna a posição do elemento
        }
    }
    // Se percorreu a lista e não encontrou o elemento
    return -1;
}
```

# Lista Linear Estática - Implementação

- Retornar o tamanho lista:

```
int tamanhoLista(Lista* lista){  
    return lista->ultimo + 1;  
}
```

# Lista Linear Estática

- **Vantagens:**

- Fácil inserção e remoção de elementos no final da lista.
- Tempos constantes para acessos a elementos pelo seu índice.

- **Desvantagens:**

- Difícil inserção e remoção no meio da lista.
- Necessidade de movimentação dos elementos durante algumas operações.

# Lista Linear Estática

- Quando usar esse tipo de implementação?
  - Tamanho definido.
  - Busca é a operação mais frequente.
  - Inserção e remoção no final da lista.



# Exercícios

- 1 Crie uma função para imprimir todos os elementos da lista.
- 2 Crie uma função para inserir um elemento em uma posição  $k$  informada pelo usuário.

# Bibliografia Básica

- ZIVIANI N. Projeto de Algoritmos: com implementações em Pascal e C, Ed. Pioneira *Thomson Learning*, 2002. Capítulos 1 e 3.
- TENENBAUM A. M. Estruturas de dados usando C. São Paulo: Makron Books do Brasil. Cap. 1,4.