

# Estrutura de Dados

Claudio Cesar de Sá, Alessandro Ferreira Leite, Lucas Hermman  
Negri, Gilmário Barbosa

Departamento de Ciência da Computação  
Centro de Ciências e Tecnologias  
Universidade do Estado de Santa Catarina

8 de outubro de 2017

- 1 O Curso
  - Ferramentas
  - Metodologia e avaliação
  - Dinâmica
  - Referências
  
- 2 Listas
  - Listas – Tamanho Limitado
  - Listas – Tamanho Ilimitado

# Agradecimentos

Vários autores e colaboradores ...

- Ao Google Images ...

# Onde estamos ...

## 1 O Curso

- Ferramentas
- Metodologia e avaliação
- Dinâmica
- Referências

## 2 Listas

- Listas – Tamanho Limitado
- Listas – Tamanho Ilimitado

## Estrutura de Dados – EDA001

- **Turma:**
- **Professor:** Claudio Cesar de Sá
  - claudio.sa@udesc.br
  - Sala 13 Bloco F
- **Carga horária:** 72 horas-aula • Teóricas: 36 • Práticas: 36
- **Curso:** BCC
- **Requisitos:** LPG, Linux, sólidos conhecimentos da linguagem C – há um documento específico sobre isto
- **Período:** 2º semestre de 2017
- **Horários:**
  - 3ª 15h20 (2 aulas) - F-205 – aula expositiva
  - 5ª 15h20 (2 aulas) - F-205 – lab

## Ementa

Representação e manipulação de tipos abstratos de dados. Estruturas lineares. Introdução a estruturas hierárquicas. Métodos de classificação. Análise de eficiência. Aplicações.

# Objetivos I

- *Geral:*

Há um documento específico sobre isto = Plano de Ensino

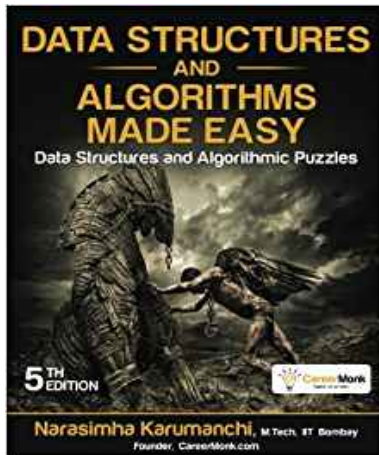
# Objetivos II

- *Específicos:*

Há um documento específico sobre isto = Plano de Ensino



# Livros que estarei usando ...



# Conteúdo programático

Há um documento específico sobre isto = Plano de Ensino

Há um documento específico sobre isto = Plano de Ensino

# Conteúdo programático

Há um documento específico sobre isto = Plano de Ensino



## Ferramentas ... nesta ordem

- Linux
- Linguagem C (ora o compilador g++)
- Codeblock, Geany, Sublime, Atom, etc

## **Metodologia:**

*As aulas serão expositivas e práticas. A cada novo assunto tratado, exemplos são demonstrados utilizando ferramentas computacionais adequadas para consolidar os conceitos tratados.*

## Avaliação

- Três provas –  $\approx 90\%$ 
  - $P_1$ : xx/ago
  - $P_2$ : xx/set
  - $P_3$ : xx/set
  - $P_4$ : 2x/out
  - $P_5$ : 1x/nov
  - $P_F$ : 2x/nov(provão: todo conteúdo)
- Exercícios de laboratório –  $\approx \%$
- Presença e participação: 75% é o mínimo obrigatório para a UDESC. Quem quiser faltar por razões diversas, ou assuntos específicos, trate pessoalmente com o professor.
- Tarefas extras que geram pontos por excelência



- Média para aprovação: 6,0 (seis)  
Nota maior ou igual a 6,0, repito a mesma no Exame Final. Caso contrário, regras da UDESC se aplicam.
- Sitio das avaliações: <https://run.codes/Users/login> código da disciplina: **GEPZ**

# Dinâmica de Aula I

- Há um monitor na disciplina – Lucas – ver no site de monitoria da UDESC os horários
- Há uma lista de discussão (para avisos e dúvidas gerais):  
**`eda-lista@googlegroups.com`**
- $\approx$  Teoria na 3a. feira
- $\approx$  Prática na 5a. feira
- E/ou 50% do tempo em teoria, 50% implementações
- Onde tudo vai estar atualizado?

# Dinâmica de Aula II

- [https://github.com/claudiosa/CCS/tree/master/estrutura\\_dados\\_EDA](https://github.com/claudiosa/CCS/tree/master/estrutura_dados_EDA)
- Ou seja, tudo vai estar *rolando* no GitHub do professor
- No Google: github + claudiosa
- Finalmente ...

## Dinâmica de Aula III

- Questões específicas (leia-se: notas, dor-de-dente, etc) venha falar pessoalmente com o professor!

## Básica:

- Há um documento específico sobre isto = Plano de Ensino ... veja em detalhes tudo que foi escrito aqui
- Mais uma vez: [https://github.com/claudiosa/CCS/tree/master/estrutura\\_dados\\_EDA](https://github.com/claudiosa/CCS/tree/master/estrutura_dados_EDA)

# Antes de Começarmos .... I

- Todos os cursos de Estrutura de Dados começam com uma motivação em torno da área para Ciência
- Vou omitir ... mas reflita se ela é ou não onipresente no nosso cotidiano?
- Exemplos: bancos eletrônicos, web, smartphones, etc

# Onde estamos ...

## 1 O Curso

- Ferramentas
- Metodologia e avaliação
- Dinâmica
- Referências

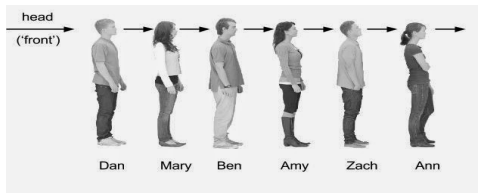
## 2 Listas

- Listas – Tamanho Limitado
- Listas – Tamanho Ilimitado

## Capítulo 05 – Listas

Pontos fundamentais a serem cobertos:

- 1 Contexto e motivação
- 2 Definição
- 3 Implementações
- 4 Exercícios





## Atenção:

- Duas partes este capítulo
- Listas com um vetor de tamanho fixo internamente
- Listas estruturadas em nós alocados dinamicamente
- Conceitualmente, equivalentes!
- Implementações discutidas: alocação dinâmica!
- Aqui justifica-se o estudo extensivo de ponteiros no início do curso
- A terminologia aqui adotada é proveniente de vários livros e do professor!

- Uma seqüência de nós ou elementos dispostos em uma ordem estritamente linear.
- Cada elemento da lista é acessível um após o outro, em ordem.
- Pode ser implementada de várias maneiras
  - ❶ Em um vetor
  - ❷ Em uma estrutura que tem um vetor de tamanho fixo e uma variável para armazenar o tamanho da lista
  - ❸ Conjunto de nós criados e ligados dinamicamente (abordagem aqui adotada nos códigos apresentados)

- Uma seqüência de nós ou elementos dispostos em uma ordem estritamente linear.
- Cada elemento da lista é acessível um após o outro, em ordem.
- Pode ser implementada de várias maneiras
  - ① Em um vetor
  - ② Em uma estrutura que tem um vetor de tamanho fixo e uma variável para armazenar o tamanho da lista
  - ③ Conjunto de nós criados e ligados dinamicamente (abordagem aqui adotada nos códigos apresentados)
  - ④ As duas implementações iniciais são exercícios de disciplinas anteriores.

- ① Talvez a estrutura de dados mais importante
- ② Generaliza Pilhas e Filas
- ③ Utilizada em várias outras estruturas como grafos e árvores

- ① Talvez a estrutura de dados mais importante
- ② Generaliza Pilhas e Filas
- ③ Utilizada em várias outras estruturas como grafos e árvores
- ④ Os exemplos em código apresentados, utilizam extensivamente endereçamentos de memória (ponteiros e ponteiros para ponteiros) e alocações dinâmicas de memória
- ⑤ Contudo, para fins conceitual usaremos uma lista *rígida* nestes slides



1

2

# Exemplo de Uso

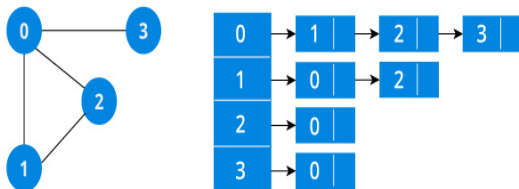


Figura 1: Uso de lista para representar matriz de adjacência



## Definição

- Um conjunto de nós,  $x_1, x_2, x_3, \dots, x_n$ , organizados estruturalmente de forma a refletir as posições relativas dos mesmos.
- Se  $n > 0$ , então  $x_1$  é o primeiro nó.
- Seja  $L$  uma lista de  $n$  nós, e  $x_k$  um nó  $\in L$  e  $k$  a posição do nó em  $L$ .
- Então,  $x_k$  é precedido pelo nó  $x_{k-1}$  e seguido pelo nó  $x_{k+1}$ .
- O último nó de  $L$  é  $x_{n-1}$ . Quando  $n = 0$ , dizemos que a lista está vazia.

# Representação

- Os nós de uma lista são armazenados em *endereços contínuos* (apenas os endereços)
- A relação de ordem é representada pelo fato de que se o endereço do nó  $x_i$  é conhecido, então o endereço do nó  $x_{i+1}$  também pode ser determinado.
- A Figura 2 apresenta a representação de uma lista linear de  $n$  nós, com endereços representados por  $k$

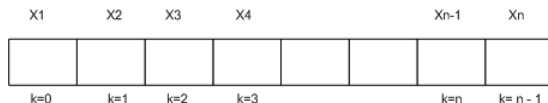


Figura 2: Exemplo de representação de lista – usando um vetor

# Representação com um Vetor de Inteiros

- Para exemplificar a implementações em C, vamos considerar que o conteúdo armazenado na lista é do tipo inteiro (nos códigos são *strings*).
- A estrutura da lista possui a seguinte representação:

```
1 struct lista{  
2     int cursor;  
3     int elemento[N];  
4 }  
5 typedef struct lista Lista;
```

- Trata-se de uma estrutura heterogênea constituída de membros distintos entre si.
- Os membros são as variáveis *cursor*, que serve para armazenar a quantidade de elementos da lista e o vetor *elemento* de inteiros que armazena os nós da lista.
- Até o momento uma alocação estática

- Para atribuírmos um valor a algum membro da lista devemos utilizar a seguinte notação:

```
1 Lista->elemento[0] = 1 //atribui o valor 1 ao primeiro elemento da
2 Lista->elemento[n-1] = 4 //atribui o valor 4 ao ultimo elemento da
```

# Operações Primitivas

- As operações básicas que devem ser implementadas em uma estrutura do tipo Lista são:

Operação	Descrição
criar()	cria uma lista vazia.
inserir( $l,e$ )	insere o elemento $e$ no final da lista $l$ .
remover( $l,e$ )	remove o elemento $e$ da lista $l$ .
imprimir( $l$ )	imprime os elementos da lista $l$ .
pesquisar( $l,e$ )	pesquisa o elemento $e$ na lista $l$ .

Tabela 1: Operações básicas da estrutura de dados lista.

# Operações auxiliares

- Além das operações básicas, temos as operações “auxiliares”. São elas:

Operação	Descrição
empty( $l$ )	determina se a lista $l$ está ou não vazia.
destroy( $l$ )	libera o espaço ocupado na memória pela lista $l$ .

Tabela 2: Operações auxiliares da estrutura de dados lista.

# Interface do Tipo Lista

```
1  /* Aloca dinamicamente a estrutura lista, inicializando
2   * seus campos e retorna seu ponteiro. A lista depois
3   * de criada terah tamanho igual a zero. Sem malloc ... ainda */
4  Lista* criar(void);
5
6  /* Insere o elemento e no final da lista l, desde que,
7   * a lista nao esteja cheia ... dada a limitacao inicial */
8  void inserir(Lista* l, int e);
9
10 /* Remove o elemento e da lista l,
11  * desde que a lista nao esteja vazia e o elemento
12  * e esteja na lista. A funcao retorna 0 se o elemento
13  * nao for encontrado na lista ou 1 caso contrario. */
14 void remover(Lista* l, int e);
15
16 /* Pesquisa na lista l o elemento e. A funcao retorna
17  * o endereco(indice) do elemento se ele pertencer a lista
18  * ou -1 caso contrario.*/
19 int pesquisar(Lista* l, int e);
20
21 /* Lista os elementos da lista l. */
22 void imprimir(Lista* l);
```

# Implementação das Listas de Tamanho Fixo

A utilização de vetores para implementar a lista traz algumas vantagens como:

- ① Os elementos são armazenados em posições contíguas da memória
- ② Basta ver a estrutura, internamente é um vetor
- ③ Economia de memória, pois os ponteiros para o próximo elemento da lista são explícitos
- ④ Há um índice de acesso direto e o *cursor* para indicar o último elemento



# Implementação das Listas de Tamanho Fixo

No entanto, as desvantagens são:

- ❶ Custo de inserir/remover elementos da lista

# Implementação das Listas de Tamanho Fixo

No entanto, as desvantagens são:

- ① Custo de inserir/remover elementos da lista
- ② Neste caso se refere ao deslocamento células a frente no caso de inserção

# Implementação das Listas de Tamanho Fixo

No entanto, as desvantagens são:

- ❶ Custo de inserir/remover elementos da lista
- ❷ Neste caso se refere ao deslocamento células a frente no caso de inserção
- ❸ ou deslocamento células para trás no caso de remoção

# Implementação das Listas de Tamanho Fixo

No entanto, as desvantagens são:

- ❶ Custo de inserir/remover elementos da lista
- ❷ Neste caso se refere ao deslocamento células a frente no caso de inserção
- ❸ ou deslocamento células para trás no caso de remoção
- ❹ Finalmente: limitação da quantidade de elementos da lista
- ❺ Este é ponto ... tamanho fixo!
- ❻ Aqui, usamos alocação dinâmica, mas todos conceitos aqui são complementares

# Função de Criação

- A função que cria uma lista, deve criar e retornar uma lista vazia;
- A função deve atribuir o valor zero ao tamanho da lista, ou seja, fazer  $l \rightarrow cursor = 0$ , como podemos ver no código abaixo.
- A complexidade de tempo para criar a lista é constante, ou seja,  $O(1)$ .

```
1  /*
2   * Aloca dinamicamente a estrutura lista, inicializando seus
3   * campos e retorna seu ponteiro. A lista depois de criada
4   * terá tamanho igual a zero.
5   */
6  Lista* criar(void){
7      Lista* l = (Lista*) malloc(sizeof(Lista));
8      l->cursor = 0;
9      return l;
10 }
```

# Função de Inserção

- A inserção de qualquer elemento ocorre no final da lista, desde que a lista não esteja cheia.
- Com isso, para inserir um elemento basta atribuírmos o valor ao elemento cujo índice é o valor referenciado pelo campo *cursor*, e incrementar o valor do cursor, ou seja fazer `l->elemento[l->cursor++] = valor`, como podemos verificar no código abaixo, a uma complexidade de tempo constante,  $O(1)$ .

```
1  /*
2   * Insere o elemento e no final da lista l, desde que,
3   * a lista nao esteja cheia.
4   */
5  void inserir(Lista* l, int e){
6      if (l == NULL || l->cursor == N){
7          printf("Error. A lista esta cheia\n");
8      }else{
9          l->elemento[l->cursor++] = e;
10     }
11 }
```

# Função de Remoção

- Para remover um elemento da lista, primeiro precisamos verificar se ele está na lista, para assim removê-lo, e deslocar os seus sucessores, quando o elemento removido não for o último.
- A complexidade de tempo da função de remoção é  $O(n)$ , pois é necessário movimentar os  $n$  elementos para remover um elemento e ajustar a lista.

```
1 /* remove um elemento da lista */
2 void remover(Lista* l, int e){
3     int i, d = pesquisar(l,e);
4     if (d != -1){
5         for(i = d; i < l->cursor; i++)
6         {
7             l->elemento[i] = l->elemento[i + 1];
8         }
9         l->cursor--;
10    }
11 }
```

# Função de Pesquisa

- Para pesquisar um elemento qualquer na lista é necessário compará-lo com os elementos existentes, utilizando alguns dos algoritmos de busca conhecidos;
- A complexidade de tempo dessa função depende do algoritmo de busca implementado. Se utilizarmos a busca seqüencial, a complexidade da função será  $O(n)$ . No entanto, é possível baixá-lo para  $O(n \log n)$ .

```
1 int pesquisar(Lista* l, int e){
2     if (l == NULL)
3         return;
4
5     int i = 0;
6     while (i <= l->cursor && l->elemento[i] != e)
7         i++;
8
9     return i > l->cursor ? -1 : i;
10 }
```



# Função de Impressão

- A impressão da lista ocorre através da apresentação de todos os elementos compreendidos entre o intervalo:  $[0..l \rightarrow cursor]$ .
- A complexidade de tempo da função de impressão é  $O(n)$ , pois no pior caso, quando lista estiver cheia, é necessário percorrer os  $n$  elementos da lista.

```
1 /* Apresenta os elementos da lista l. */
2 void imprimir(Lista* l){
3     int i;
4     for(i = 0; i < l->cursor; i++)
5         printf("%d ", l->elemento[i]);
6     printf("\n");
7 }
```

# Exemplo de Uso da Lista

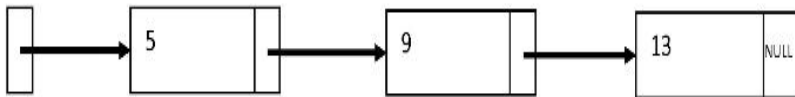
```
1 \#include <stdio.h>
2 \#include "list.h"
3 int main(void)
4 {
5     Lista* l = criar();
6     int i, j = 4;
7
8     /* Inserir 5 elementos na lista */
9     for (i = 0; i < 5; i++)
10         inserir(l, j * i);
11
12     /* Apresenta os elementos inseridos na lista*/
13     imprimir(l);
14     /* Remove o segundo elemento da lista*/
15     remover(l, j);
16     /* Apresenta os elementos da lista */
17     imprimir(l);
18 }
```

# Listas *Ilimitadas* (ou *Listas Encadeadas*)

## Complemento

- Nesta parte vamos discutir as listas *ilimitadas*
- Crescem ou não dinamicamente de acordo com a disponibilidade de memória
- O usuário controla a memória etc
- Todos conceitos vistos anteriormente, são aqui preservados
- ....

# Estrutura clássica da implementação dessas listas



```
struct Node {  
    int data;  
    Node *nextPtr;  
};
```

```
Node *head;
```

Figura 3: Lista Simplesmente Encadeada – LSE

Conceitualmente, nada foi modificada em relação ao que se tinha anteriormente!

## *Espalhadas pela memória principal ... não contíguas!*

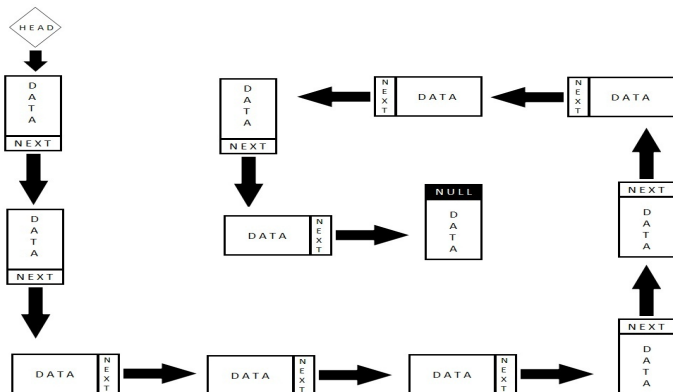


Figura 4: Lista Simplesmente Encadeada – LSE

# Truques destas Implementações de Listas

## Observações:

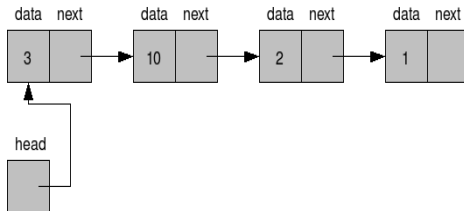
- Ao contrário do que foi visto, estas estruturas são alocadas dinamicamente na MP;
- Idem quanto as remoções e a sua liberação de área;
- O ponto é que facilmente pode se perder o ponteiro de início de lista se o mesmo entrar no cálculo ou numa operação de lista;
- Logo, vamos passar apenas o seu endereço afim de preservá-lo na função principal
- Uma proteção (*blindagem*) deste ponteiro cabeça de lista
- Ou seja, passando apenas o seu endereço para as funções, estas terão que receber como um ponteiro de ponteiro!
- Sim, pois virá o endereço de um ponteiro que tem o início de lista, logo, uma estrutura de nó com um ponteiro de ponteiro é agora interessante.
- Veja as implementações e as estude com cuidado!

# Complexidade de Implementações de Listas

Parameter	Linked List	Array	Dynamic Array
Indexing	$O(n)$	$O(1)$	$O(1)$
Insertion/deletion at beginning	$O(1)$	$O(n)$ , if array is not full (for shifting the elements)	$O(n)$
Insertion at ending	$O(n)$	$O(1)$ , if array is not full	$O(1)$ , if array is not full $O(n)$ , if array is full
Deletion at ending	$O(n)$	$O(1)$	$O(n)$
Insertion in middle	$O(n)$	$O(n)$ , if array is not full (for shifting the elements)	$O(n)$
Deletion in middle	$O(n)$	$O(n)$ , if array is not full (for shifting the elements)	$O(n)$
Wasted space	$O(n)$ (for pointers)	0	$O(n)$

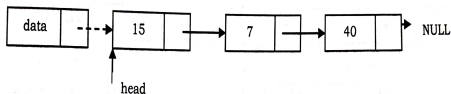
Figura 5: Comparativo da complexidade quanto as implementações

# Incluindo um nó no início da lista



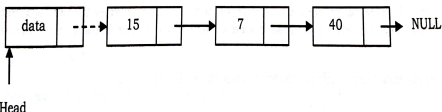
- Update the next pointer of new node, to point to the current head.

New node



- Update head pointer to point to the new node.

New node





# Incluindo um nó numa posição da lista

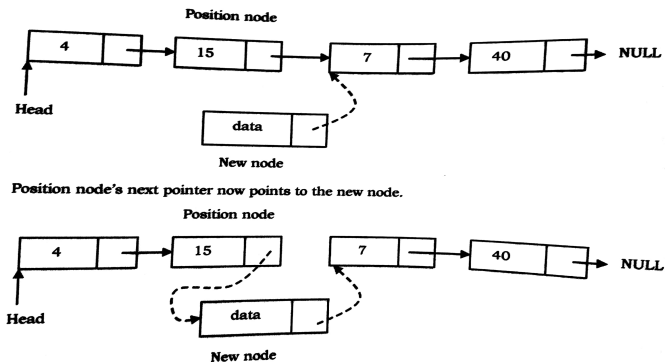
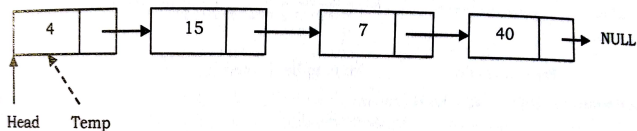


Figura 7: Incluir um nó numa posição da lista

# Excluindo o nó no início da lista – *cabeça* da lista

Create a temporary node which will point to the same node as that of head.



Now, move the head nodes pointer to the next node and dispose of the temporary node.

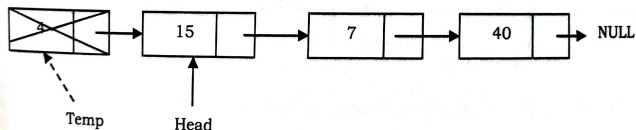


Figura 8: Exclui o nó no início da lista

# Excluindo um nó no meio da lista

Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.

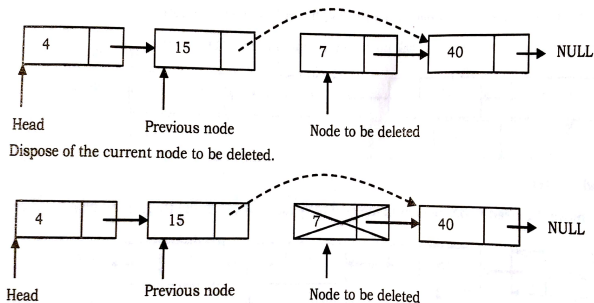
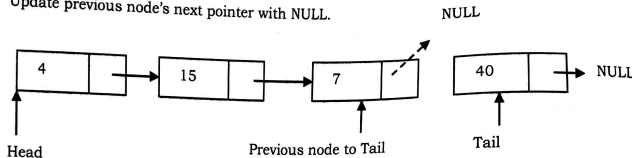


Figura 9: Exclui nó no meio da lista – k-ésima posição ou por conteúdo

# Excluindo o último nó da lista

- Update previous node's next pointer with NULL.



- Dispose of the tail node.

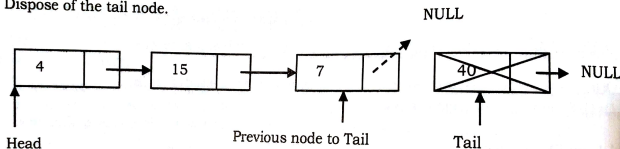


Figura 10: Exclui o último nó da lista

# Listas Duplamente Encadeadas – LDE

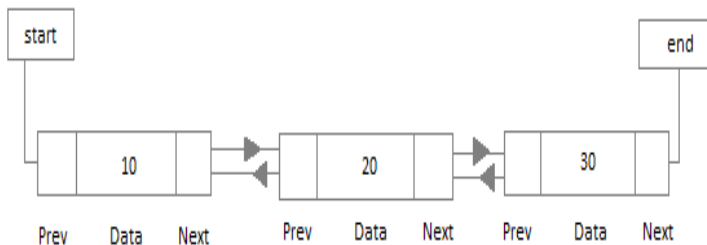


Figura 11: Listas Duplamente Encadeadas – LDE

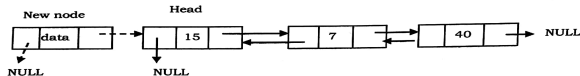
- Para onde estão apontando o primeiro e o último ponteiro ?
- Podem ser circulares?

# Inserindo um nó em listas duplamente encadeadas – LDE

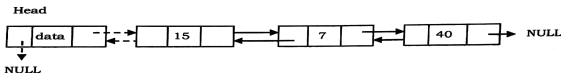
## Inserting a Node in Doubly Linked List at the Beginning

In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps:

- Update the right pointer of the new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.



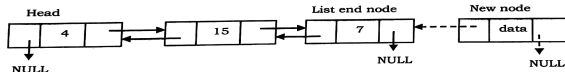
- Update head node's left pointer to point to the new node and make new node as head.



## Inserting a Node in Doubly Linked List at the Ending

In this case, traverse the list till the end and insert the new node.

- New node right pointer points to NULL and left pointer points to the end of the list.



- Update right pointer of last node to point to new node.

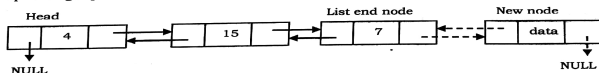


Figura 12: Incluindo um nó numa LDE

# Listas Circulares

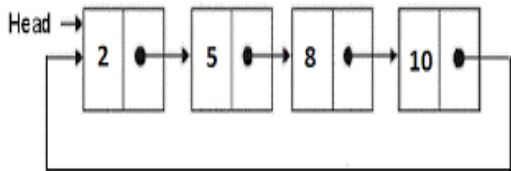
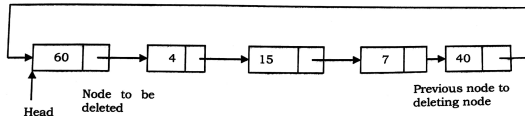


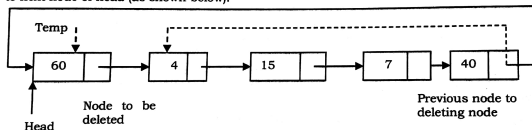
Figura 13: Listas Circulares

# Excluindo o nó *cabeça* de lista

- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



- Create a temporary node which will point to the head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



- Now, move the head pointer to next node. Create a temporary node which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).

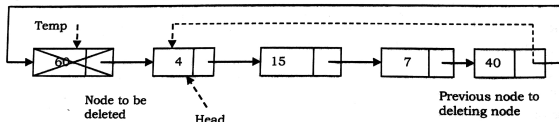


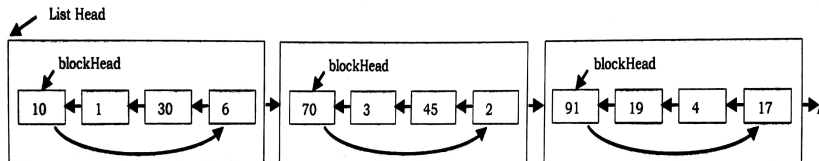
Figura 14: Exclui o nó do início da lista



# Listas Blocadas ou Encapsuladas – *enrolled*

One of the biggest advantages of linked lists over arrays is that inserting an element at any location takes only  $O(1)$  time. However, it takes  $O(n)$  to search for an element in a linked list. There is a simple variation of the singly linked list called *unrolled linked lists*.

An unrolled linked list stores multiple elements in each node (let us call it a block for our convenience). In each block, a circular linked list is used to connect all nodes.

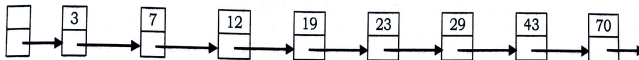


Assume that there will be no more than  $n$  elements in the unrolled linked list at any time. To simplify this problem, all blocks, except the last one, should contain exactly  $\lfloor \sqrt{n} \rfloor$  elements. Thus, there will be no more than  $\lfloor \sqrt{n} \rfloor$  blocks at any time.

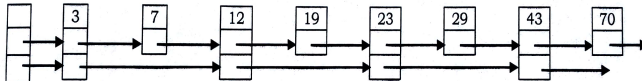
Figura 15: Listas Encapsuladas

# Listas Niveladas – *skipped*

Skip Lists with One Level



Skip Lists with Two Levels



Skip Lists with Three Levels

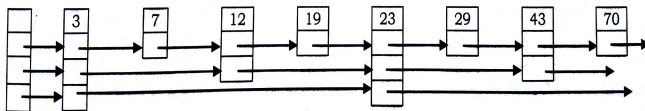


Figura 16: Listas Niveladas

Aqui esta *lista* é grande ...

- Inverter a lista
- *Merge* de duas listas
- Concatenar duas listas simples
- 
- 
-

- ① Karumanchi, Narashimha (2017). *Data Structures and Algorithms Made Easy – Data Structures and Algorithms and Puzzles*. CareerMonk.com
- ② Tenenbaum, A. M., Langsam, Y., and Augenstein, M. J. (1995). Estruturas de Dados Usando C. MAKRON Books, pp. 207-250.
- ③ Wirth, N. (1989). Algoritmos e Estrutura de dados. LTC, pp. 151-165.