Estrutura de Dados

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

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

7 de outubro de 2017

Sumário I

Agradecimentos

Vários autores e colaboradores ...

- Ao Google Images ...

Onde estamos ...

Disciplina

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

Ementa

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

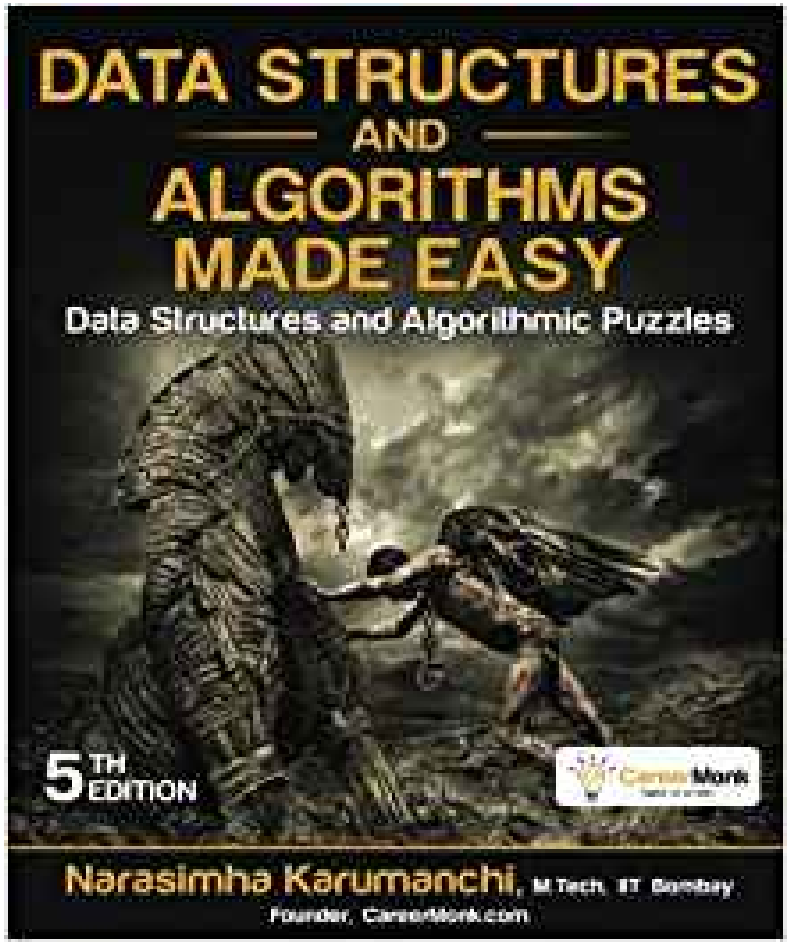
Objetivos I

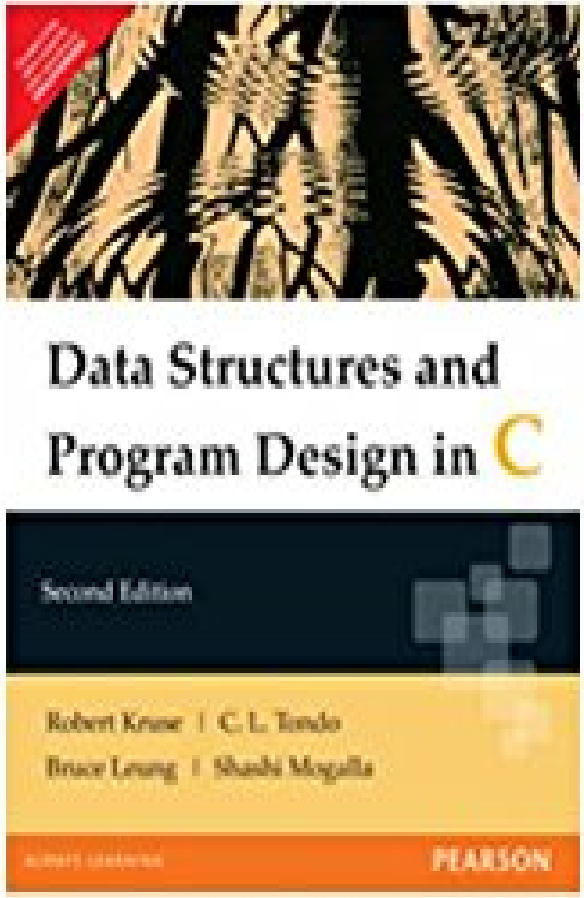
- ***Geral:***
  Há um documento específico sobre isto = Plano de Ensino

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

Bibliografia UDESC

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 e avaliação I

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

Metodologia e avaliação III

- 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`
- 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!

Bibliografia I

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`

Antes de Começarmos I

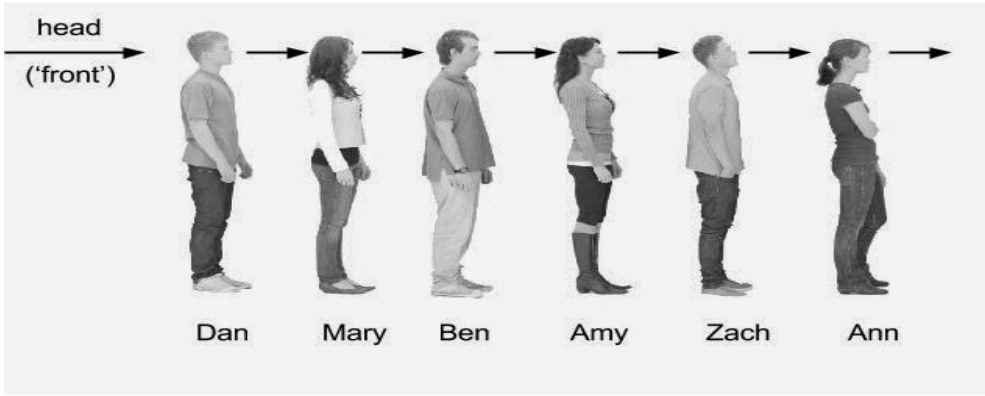
- Todos os cursos de Estrutura de Dados começam com uma motivação em torno da área para Ciência
- Vou omitir ... mas reflita se ela é ou não onipresente no nosso cotidiano?
- Exemplos: bancos eletronicos, web, smartphones, etc

Onde estamos ...

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

Introdução

- 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 dinâmicamente (abordagem aqui adotada nos códigos apresentados)

Introdução

- 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 dinâmicamente (abordagem aqui adotada nos códigos apresentados)
  4. As duas implementações iniciais são exercícios de disciplinas anteriores.

Motivação

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

Motivação

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 dinãmicas de memória
5. Contudo, para fins conceitual usaremos uma lista *rígida* nestes slides

Aplicações

Aplicações

1.
2.

Exemplo de Uso

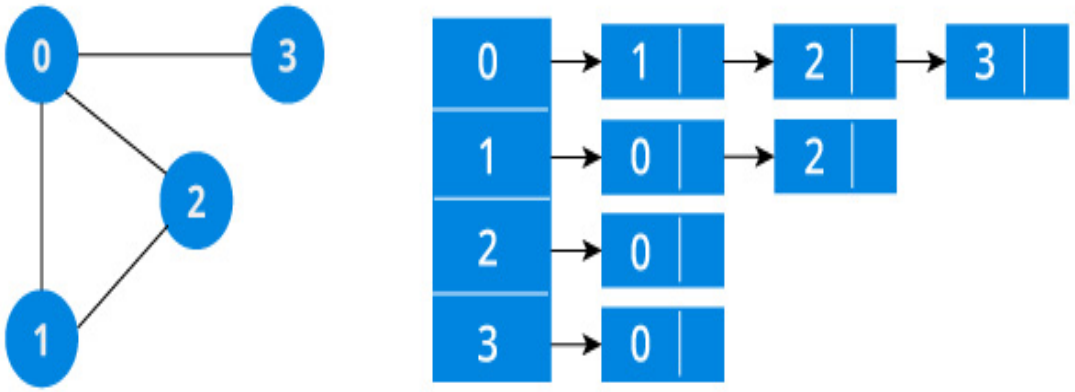


Figura 1: Uso de lista para representar matriz de adjacência

Definição

Definição

- Um conjunto de nós, $x_1, x_2, x_3, \cdots, x_n$, 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

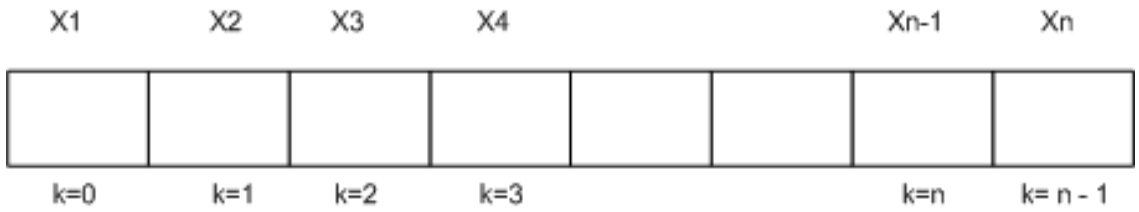


Figura 2: Exemplo de representação de lista – usando um vetor

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

Representação Estática – Tamanho Fixo

- Para atribuirmos 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

```c
/* Aloca dinamicamente a estrutura lista, inicializando
 * seus campos e retorna seu ponteiro. A lista depois
 * de criada terah tamanho igual a zero. Sem malloc ... ainda */
Lista* criar(void);

/* Insere o elemento e no final da lista l, desde que,
 * a lista nao esteja cheia ... dada a limitacao inicial */
void inserir(Lista* l, int e);

/* Remove o elemento e da lista l,
 * desde que a lista nao esteja vazia e o elemento
 * e esteja na lista. A funcao retorna 0 se o elemento
 * nao for encontrado na lista ou 1 caso contrario. */
void remover(Lista* l, int e);

/* Pesquisa na lista l o elemento e. A funcao retorna
 * o endereco(indice) do elemento se ele pertencer a lista
 * ou -1 caso contrario.*/
int pesquisar(Lista* l, int e);

/* Lista os elementos da lista l. */
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:

1. 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:

1. Custo de inserir/remover elementos da lista
2. Neste caso se refere ao deslocamento células a frente no caso de inserção
3. ou deslocamento células para trás no caso de remoção

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
3. ou deslocamento células para trás no caso de remoção
4. Finalmente: limitação da quantidade de elementos da lista
5. Este é ponto ... tamanho fixo!
6. 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 $l->cursor = 0$, 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   * terah 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 atribuirmos 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: $[0..l->cursor]$.
- 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

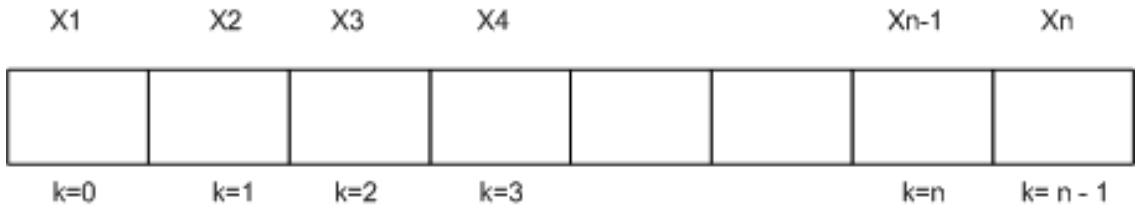
```
1  \#include <stdio.h>
2  \#include "list.h"
3  int main(void)
4  {
5      Lista* l = criar();
6      int i, j = 4;
7
8      /* Inserir 5 elementos na lista */
9      for (i = 0; i < 5; i++)
10       inserir(l,j * i);
11
12     /* Apresenta os elementos inseridos na lista*/
13     imprimir(l);
14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

Listas *Ilimitadas*

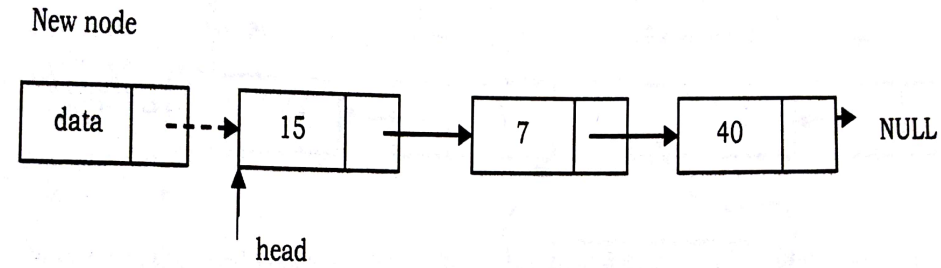
Complemento

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

Incluindo um nó no início da lista



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



- Update head pointer to point to the new node.

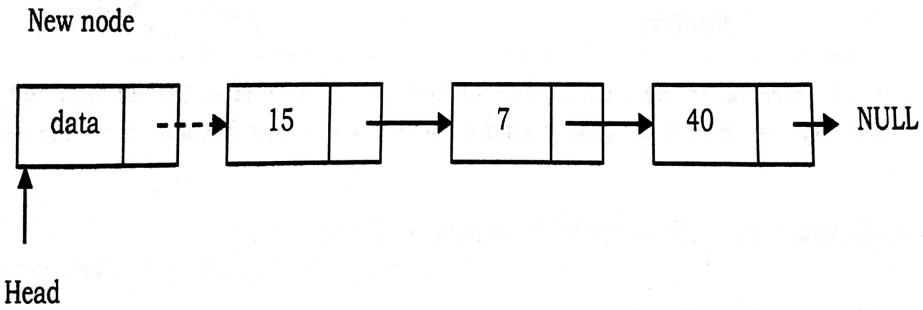


Figura 3: Incluir um nó no início da lista

Incluindo um nó numa posição da lista

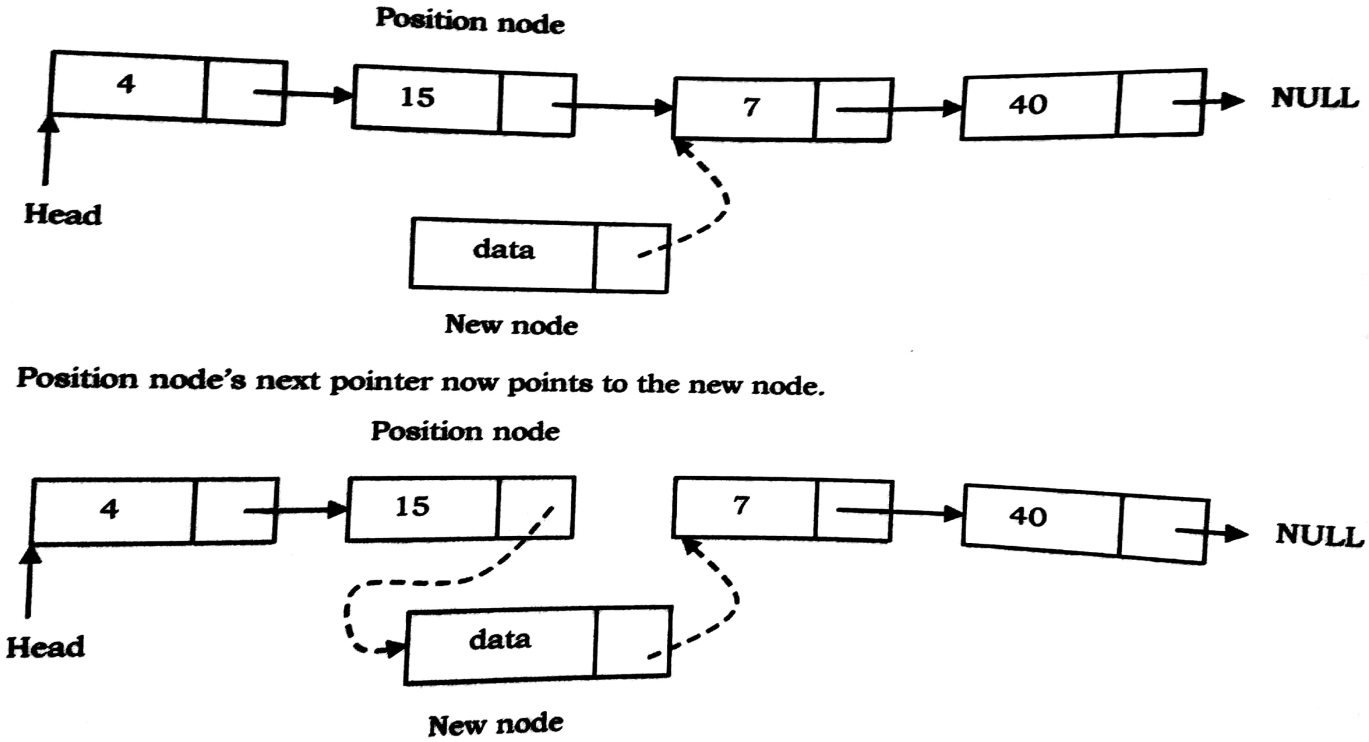


Figura 4: Incluir um nó numa posição da lista

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

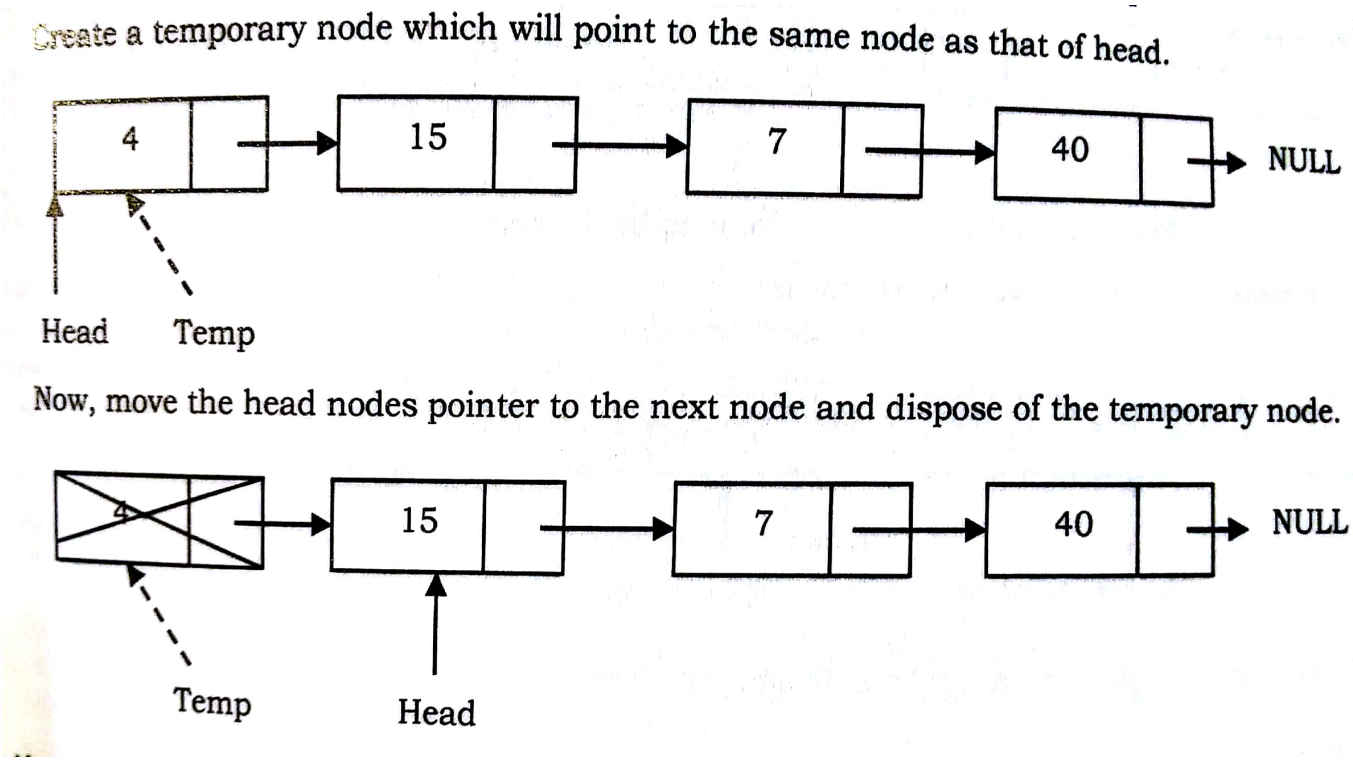


Figura 5: Exclui o nó no início da lista

Excluindo um nó no meio da lista

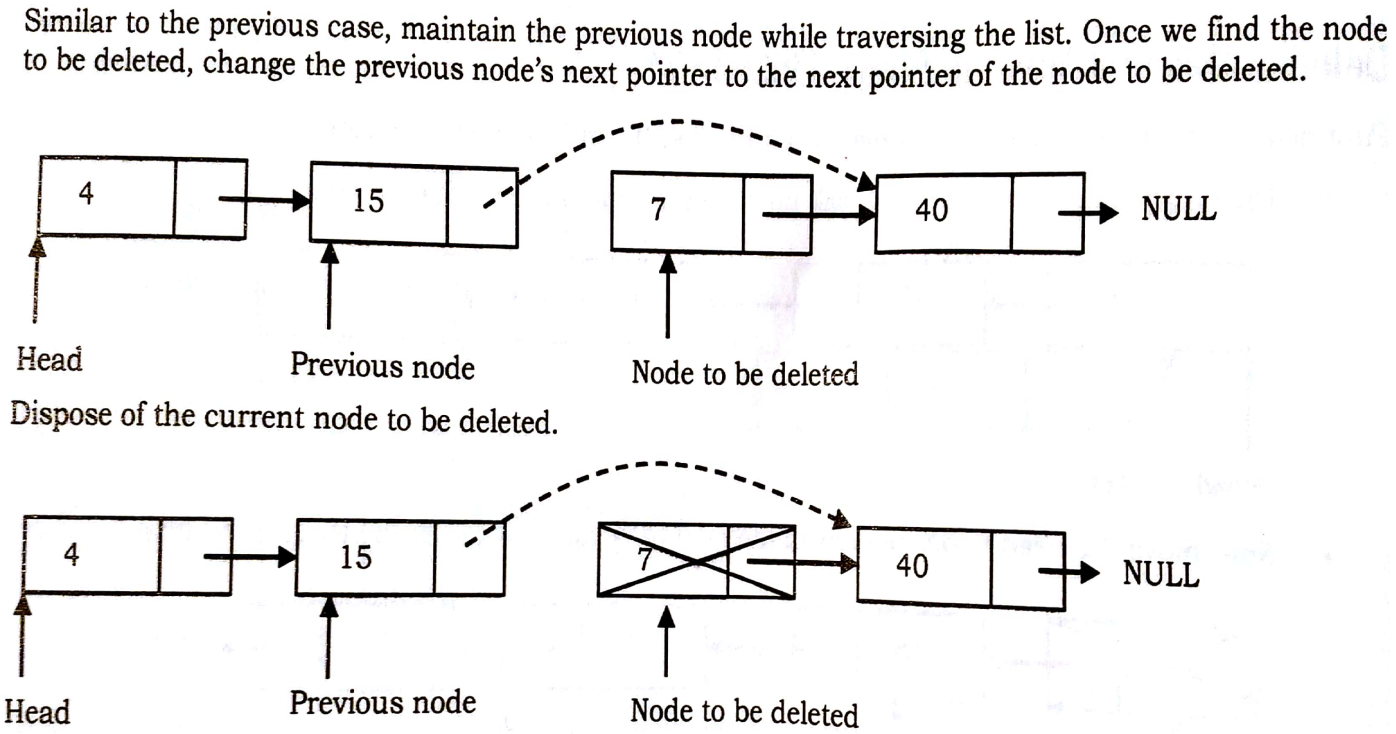


Figura 6: Exclui nó no meio da lista – k-ésima posição ou por conteúdo

Excluindo o último nó da lista

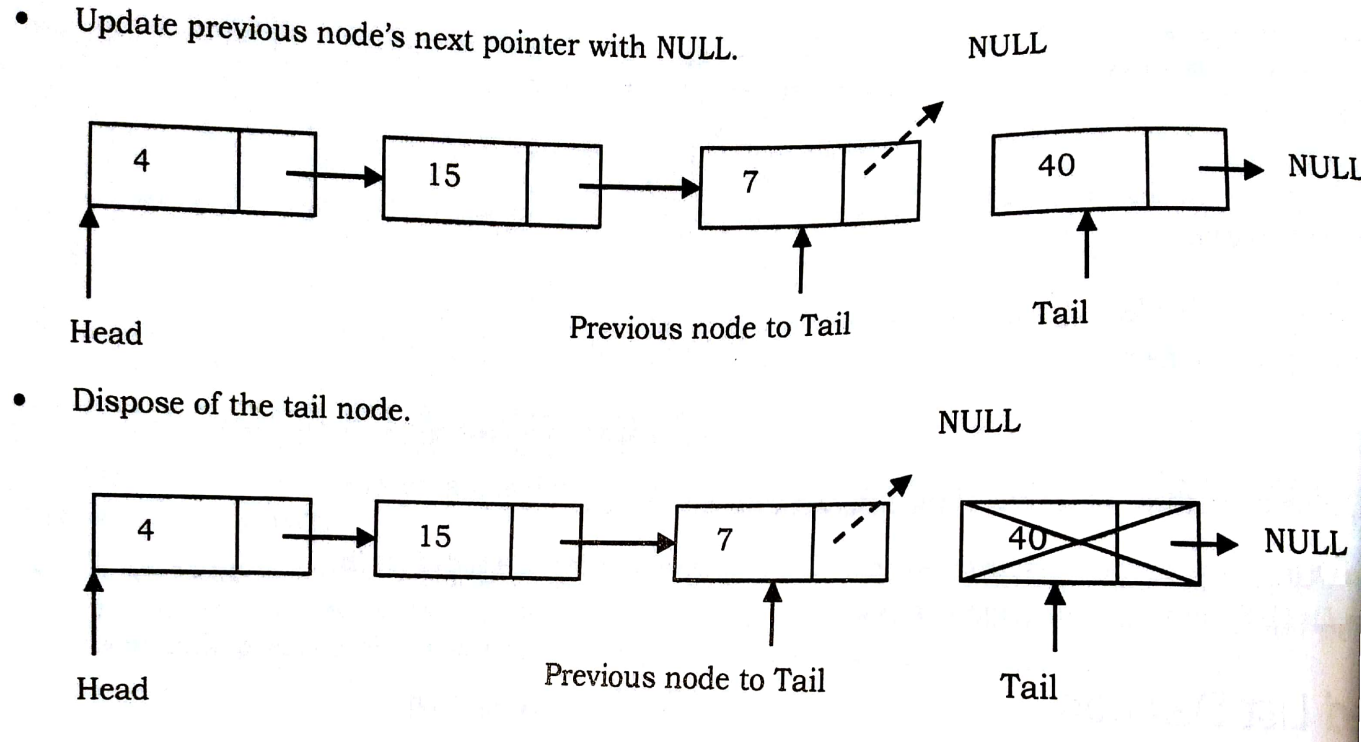


Figura 7: Exclui o último nó da lista

Listas Duplamente Encadeadas

Figura 8: Listas Duplamente Encadeadas

Figura 9: Listas Circulares

Exercícios

Aqui esta *lista* é grande ...

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

Referências

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 Augestein, 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.