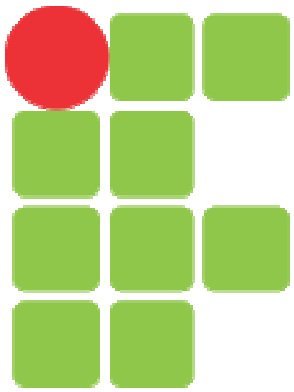


INSTITUTO FEDERAL
CEARÁ

Ernani Andrade Leite

ernani@ifce.edu.br

Abstração de Dados

Um **processo** é qualquer sequência finita ordenada de passos que visa promover transformações definidas sobre uma determinada matéria-prima.

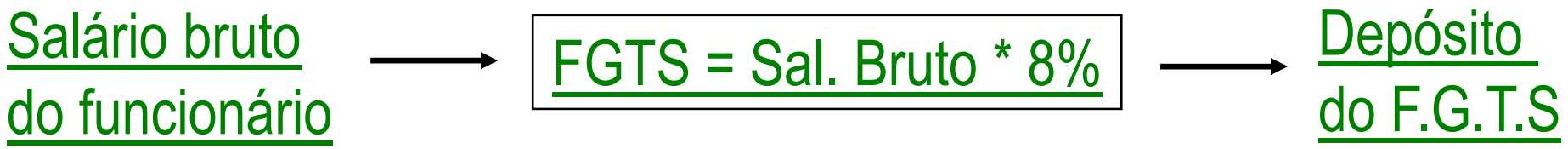
INSTITUTO FEDERAL
CEARÁ

Processamento de Dados

Quando a matéria-prima usada no processo é **abstrata**, isto é, apresenta-se sob a forma de valores, quantidades ou símbolos, então denomina-se **processamento de dados**. Quando o processamento é realizado por um computador, **entrada** refere-se aos dados que são colhidos do mundo real, externo ao computador, e **processo** refere-se a uma série finita de operações que são realizadas a partir destes dados, a fim de transformá-los em alguma informação desejada (**saída**).

DADOS → CONHECIMENTO → INFORMAÇÃO

Processamento de Dados



É importante notar que dado e informação são conceitos relativos: dado é o que entra no processo, enquanto informação é o que sai. A informação gerada pelo processo, pode, em outra situação, servir como dado para um outro processo.

Questões básicas para o programador:

1. Como representar a abstração da realidade dentro do computador ? **Estrutura de Dados**
2. Como representar o conhecimento necessário para manipular esta abstração ? **Algoritmo (ou Programa)**

INSTITUTO FEDERAL
CEARÁ

Algoritmos

Os algoritmos fazem parte do dia-a-dia das pessoas.
As instruções para o uso de medicamentos, as indicações de como montar um aparelho qualquer, uma receita de culinária são alguns exemplos de algoritmos.

Segundo Ziviani, um algoritmo pode ser visto como uma sequência de ações executáveis para a obtenção de uma solução para um determinado tipo de problema.

INSTITUTO FEDERAL
CEARÁ

Estrutura de Dados

Estrutura de dados e algoritmos estão intimamente ligados. Não se pode estudar estruturas de dados sem considerar os algoritmos associados a elas, assim como a escolha dos algoritmos em geral depende da representação e da estrutura de dados.

Para resolver um problema é necessário escolher uma abstração da realidade, em geral através da definição de um conjunto de dados que representa a situação real. A seguir deve ser escolhida a forma de representar estes dados.

A **escolha da representação** dos dados é determinada, entre outras, pelas operações a serem realizadas sobre os dados.

Programas

O processamento de dados é feito pela execução de programas. Um **programa** é uma sequência de instruções codificadas em uma linguagem de programação e para ser executado precisa ser armazenado na memória do computador.

Segundo Ziviani, programar é basicamente estruturar dados e construir algoritmos.

De acordo com Wirth, **programas** são formulações concretas de algoritmos abstratos, baseados em representações e estruturas específicas de dados. Em outras palavras, programas representam uma classe especial de algoritmos capazes de serem seguidos por computadores.

INSTITUTO FEDERAL
CEARÁ

Linguagens de Programação

Um computador só é capaz de seguir programas em linguagem de máquina, que correspondem a uma sequência de instruções obscuras e desconfortáveis.

Para controlar tal problema é necessário construir linguagens mais adequadas para facilitar a tarefa de programar um computador.

Segundo Dijkstra, uma linguagem de programação é uma técnica de notação para programar, com a intenção de servir de veículo tanto para a expressão do raciocínio algorítmico quanto para a execução automática de um algoritmo por um computador.

INSTITUTO FEDERAL
CEARÁ

Tipos de Dados

Em linguagens de programação é importante classificar constantes, variáveis, expressões e funções de acordo com certas características, as quais indicam o seu **tipo de dados**. Este tipo deve caracterizar o conjunto de valores a que uma constante pertence, ou que podem ser assumidos por uma variável ou expressão, ou que podem ser gerados por uma função (Wirth, 1976, pp.4-40). Tipos simples de dados são grupos de valores indivisíveis, como os tipos básicos ***int***, ***char*** e ***float*** do C. Por exemplo, uma variável do tipo ***int*** pode assumir um valor numérico do intervalo dos inteiros, e nenhum outro valor. Os tipos estruturados em geral definem uma coleção de valores simples (vetor), ou um agregado de valores de tipos diferentes (registro, ou ***struct***).

Tipos Abstratos de Dados, Segundo Ziviani

Um **Tipo Abstrato de Dados** (**TAD**) pode ser visto como um modelo matemático, acompanhado das operações definidas sobre o modelo. O conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação forma um exemplo de um TAD.

TAD podem ser considerados generalizações de tipos primitivos de dados, da mesma forma que procedimentos são generalizações de operações primitivas tais como adição, subtração e multiplicação. Assim como um procedimento é usado para encapsular partes de um algoritmo, o tipo abstrato de dados pode ser usado para encapsular tipos de dados. Neste caso a definição do tipo e todas as operações definidas sobre ele podem ser localizadas em uma única seção do programa.

INSTITUTO FEDERAL
CEARÁ

Tipos Abstratos de Dados, Segundo Pereira

Um **TAD** é formado por um conjunto de valores e por uma série de funções que podem ser aplicadas sobre estes valores. Funções e valores, em conjunto, constituem um modelo matemático que pode ser empregado para "modelar"e solucionar problemas do mundo real; servindo para especificar as características relevantes dos objetos envolvidos no problema, de que forma eles se relacionam e como podem ser manipulados.

Entretanto, sendo o TAD apenas um modelo matemático, sua definição não leva em consideração como os valores serão representados na memória do computador (organização dos bits), nem se preocupa com o "tempo" que será gasto para aplicar as funções (rotinas) sobre tais valores.

Tipos Abstratos de Dados, Segundo Pereira

Para que se possa realmente aplicar um modelo matemático na resolução de problemas por computador, é preciso antes transformá-lo em um **tipo de dados concreto** (ou simplesmente, tipo de dados). A transformação de um tipo de dados abstrato em um tipo de dados concreto é chamada **implementação**. É durante o processo de implementação que a estrutura de armazenamento dos valores é especificada, e que os algoritmos que desempenharão o papel das funções são projetados.

INSTITUTO FEDERAL
CEARÁ

Implementação de um TAD

Freqüentemente, nenhuma implementação é capaz de representar um modelo matemático completamente; assim, é necessário reconhecer as limitações, vantagens e desvantagens, de uma implementação particular.

O projetista deve ser capaz de escolher aquela mais adequada para resolver o problema específico proposto, tomando como medidas de eficiência da implementação sobretudo, as suas necessidades de espaço de armazenamento e tempo de execução.

As representações mais utilizadas para os TAD, que serão estudados a seguir, são as implementações através de arranjos (*vetores*) e de apontadores.

INSTITUTO FEDERAL
CEARÁ

Estrutura de Dados Básicas

Uma das formas mais comumente usadas para se manter dados agrupados é a lista.

Afinal, quem nunca organizou uma lista de compras antes de ir ao mercado, ou então uma lista de amigos que irão participar do futebol no final-de-semana.

Lista são estruturas muito flexíveis porque podem crescer ou diminuir de tamanho durante a execução de um programa, de acordo com a demanda. Itens podem ser acessados, inseridos ou retirados de um lista. Duas listas podem ser concatenadas para formar uma lista única, assim como uma lista pode ser partida em duas ou mais listas.

INSTITUTO FEDERAL
CEARÁ

Listas Lineares

Uma **lista linear** é uma seqüência de zero ou mais itens: x_1, x_2, ..., x_n, onde x_i é de um determinado tipo e n representa o tamanho da lista linear.

Sua principal propriedade estrutural fundamenta-se apenas na posição relativa dos elementos, que são dispostos linearmente.

Se $n = 0$, dizemos que a lista está **vazia**; caso contrário, são válidas as seguintes propriedades:

1. x_1 é o primeiro elemento da lista;
2. x_n é o último elemento da lista;
3. x_k, $2 \leq k \leq n-1$, é precedido pelo elemento x_{k-1} e seguido por x_{k+1} na lista.
4. x_i é dito estar na i-ésima posição da lista.

INSTITUTO FEDERAL
CEARÁ

TAD Lista Linear (1/2)

Para criar um **tipo abstrato de dados** Lista, é necessário definir um conjunto de operações sobre os objetos do tipo Lista. O conjunto de operações a ser definido depende de cada aplicação, não existindo um conjunto de operações que seja adequado a todas as aplicações.

1. Criar um lista linear vazia.
2. Inserir um novo item imediatamente após o *i*-ésimo item.
3. Retirar o *i*-ésimo item.
4. Localizar o *i*-ésimo item para examinar e/ou alterar o conteúdo de seus componentes.
5. Combinar duas ou mais listas lineares em uma lista única.
6. Partir uma lista linear em duas ou mais listas.
7. Fazer uma cópia da lista linear.
8. Ordenar os itens da lista em ordem ascendente ou descendente, de acordo com alguns de seus componentes.
9. Pesquisar a ocorrência de um item com um valor particular em algum componente.

INSTITUTO FEDERAL
CEARÁ

TAD Lista Linear (2/2)

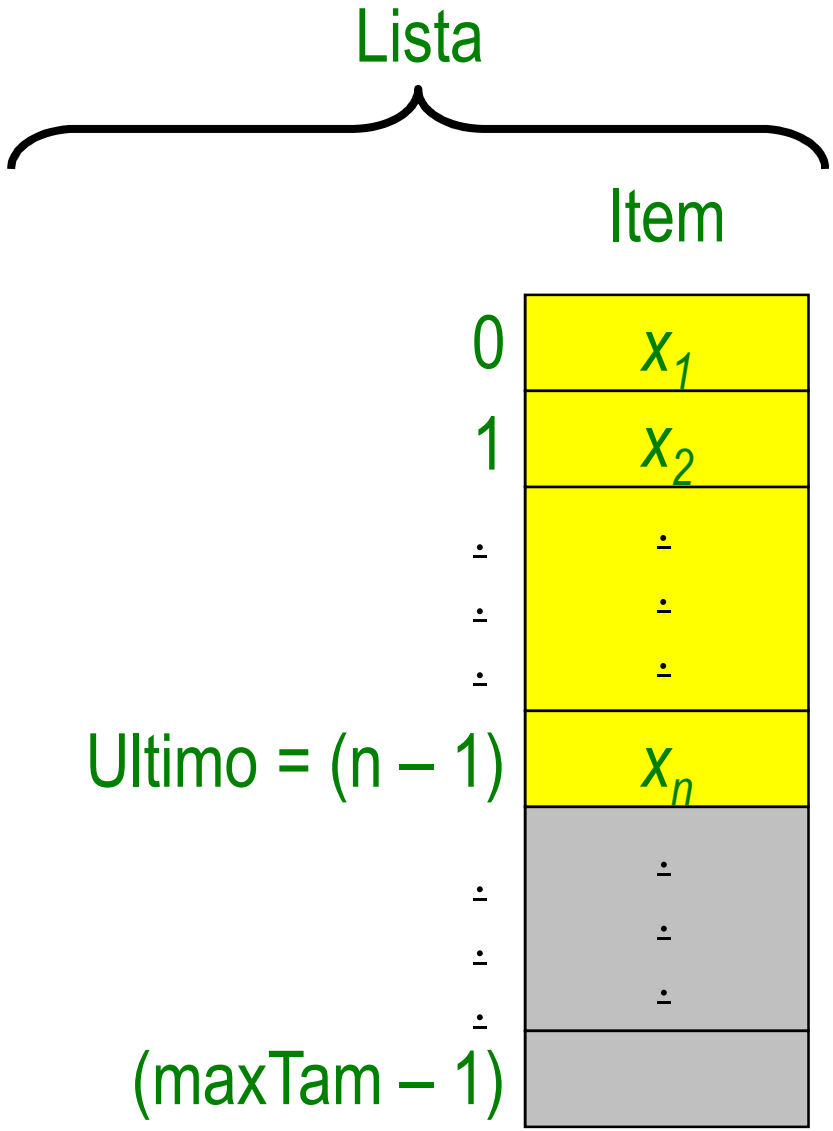
Restringindo o conjunto de operações necessário para uma aplicação que utilize o TAD Lista Linear:

1. **FazListaVazia(Lista)**. Faz a lista ficar vazia.
2. **ListaVazia(Lista)**. Esta função retorna 1 (*true*) se a lista está vazia; senão retorna 0 (*false*).
3. **Imprime(Lista)**. Imprime os itens da lista na ordem de ocorrência.
4. **Insere(x, Lista)**. Insere *x* após o último item da lista. (ou)
4. **Insere(x, Lista, p)**. Insere *x* na posição *p* da lista.
5. **Retira(p, Lista, x)**. Retira o item *x* que está na posição *p* da lista, retirando-o da lista e deslocando os itens a partir da posição *p+1* para as posições anteriores.

Existem várias estruturas de dados que podem ser usadas para representar listas lineares, cada uma com vantagens e desvantagens particulares. As duas representações mais utilizadas são as implementações através de *arranjos* (*vetores*) e de *apontadores*.

Implementação de Listas através de Arranjos

Em um tipo estruturado arranjo, os itens da lista são armazenados em posições *contíguas* de memória, conforme ilustra a Figura ao lado. Neste caso a lista pode ser percorrida em qualquer direção. A inserção de um novo item pode ser realizada após o último item com custo constante. A inserção de um novo item no meio da lista requer um deslocamento de todos os itens localizados após o ponto de inserção. Da mesma forma, retirar um item do início da lista requer um deslocamento de itens para preencher o espaço deixado vazio.



Implementação de Listas através de Arranjos

```
// modelo matemático (estrutura de dados)
struct TipoItem {      // cada item da lista corresponde a um
  char nome[30];       // registro (TipoItem) composto apenas
};                                  // do campo nome

const maxTam = 10;   // tamanho máximo da lista

struct TipoLista {
  TipoItem Item[maxTam];
  int Ultimo;
};
```

O campo *Item* é o principal componente do registro *TipoLista*. Os itens são armazenados em um **vetor** de tamanho suficiente para armazenar a lista. Já o campo *Ultimo* do registro *TipoLista* contém o valor da posição do último elemento da lista. O *i*-ésimo item da lista está armazenado na *i*-ésima posição do **vetor**, $0 \leq i \leq Ultimo$. A constante*maxTam* define o tamanho máximo permitido para a lista.

Implementação de Listas através de Arranjos

A implementação de listas através de arranjos tem como vantagem o acesso indexado e a economia de memória, pois os apontadores, ou índices, são implícitos nesta estrutura.

Como desvantagens pode-se citar:

1. o custo para inserir ou retirar itens da lista, pode causar um deslocamento de todos os itens, no pior caso;

2. em aplicações em que não existe previsão sobre o crescimento da lista, a utilização de arranjos em linguagens como o C pode ser problemática porque neste caso o tamanho máximo da lista tem que ser definido em tempo de compilação.

Implementação de Listas através de Arranjos

```
// Faz a 'Lista' ficar vazia
void FazListaVazia(TipoLista *Lista) {
  Lista->Ultimo = -1;
}
```

```
// Esta função retorna 1 (true) se a 'Lista'
// está vazia; senão retorna 0 (false)
int ListaVazia(TipoLista *Lista) {
  return(Lista->Ultimo == -1);
}
```

```
// Imprime os itens da 'Lista'
void Imprime(TipoLista *Lista) {
  clrscr();
  for (int i=0; i<=Lista->Ultimo; i++)
    printf("%d- %s\n", i, Lista->Item[i].nome);
}
```

Implementação de Listas através de Arranjos

```
// Insere o item 'x' na final da 'Lista'.
int Insere(TipoItem x, TipoLista *Lista) {

  if (Lista->Ultimo == (maxTam – 1))
     return(0);     // Erro: Lista Cheia !

  else {
    Lista->Ultimo = Lista->Ultimo + 1;
// item inserido no final da lista
    Lista->Item[Lista->Ultimo] = x;
    return(1); // Item inserido com sucesso
  }
}
```

Implementação de Listas através de Arranjos

```
// Insere o item 'x' na posição 'p' da 'Lista'.
int Insere(TipoItem x, TipoLista *Lista, int p) {

  if (Lista->Ultimo == (maxTam – 1))
     return(0);          // Erro: Lista Cheia !

  else {
     Lista->Ultimo = Lista->Ultimo + 1;
     for (int i=Lista->Ultimo; i<p; i--)
        Lista->Item[i] = Lista->Item[i-1];

// item inserido na posição 'p' da lista
     Lista->Item[p] = x;
     return(1); // Item inserido com sucesso
  }

}
```

desloca os itens a partir da <u>última posição</u> até **p** para as posições seguintes, liberando a posição 'p' para inserir o item

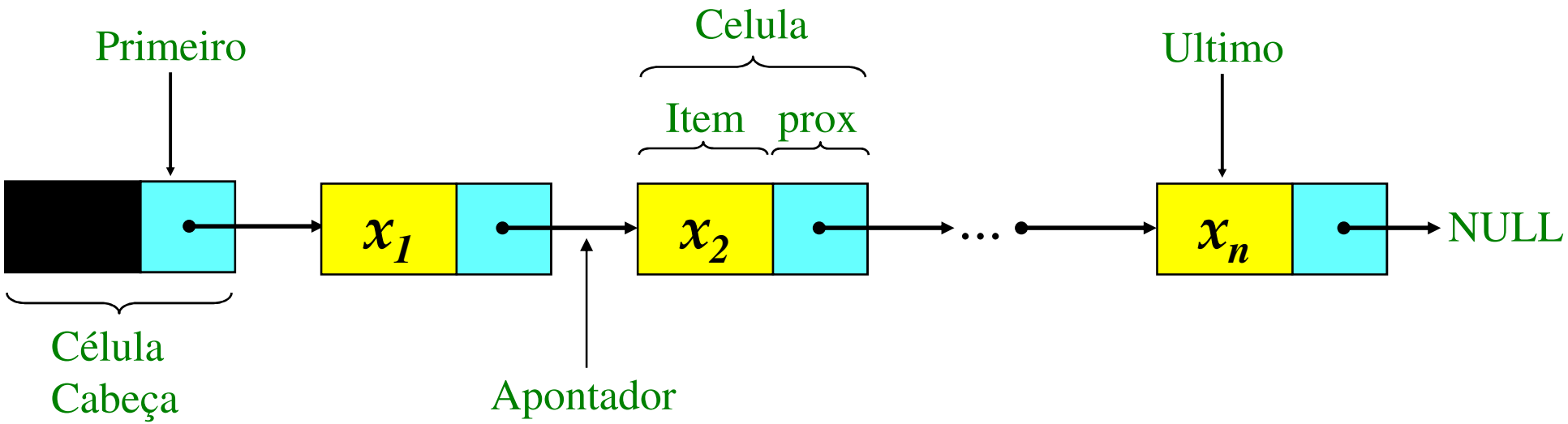
Implementação de Listas através de Arranjos

```
// Retira o item 'x' que está na posição 'p' da 'Lista'
int Retira(int p, TipoLista *Lista, TipoItem *x) {
  if ((ListaVazia(Lista)) || (p < 0) ||
      (p > Lista->Ultimo))
     return(0);          // Erro: Lista vazia ou
                                  // posição não existe.
  else {
    *x = Lista->Item[p]; // item retornado

    for (int i=p; i<=(Lista->Ultimo-1); i++)
      Lista->Item[i] = Lista->Item[i+1];

    Lista->Ultimo = Lista->Ultimo - 1;
    return(1); // Item retirado com sucesso
  }
}
```

desloca os itens a partir da posição **p+1** para as posições anteriores, ocupando a posição do item retirado

Implementação de Listas através de Apontadores

Em uma implementação de listas através de apontadores, cada item da lista é encadeado com o seguinte através de uma variável do tipo *Apontador*. Este tipo de implementação permite utilizar posições não contíguas de memória, sendo possível inserir e retirar elementos sem haver necessidade de deslocar os itens seguintes da lista. A Figura abaixo ilustra uma lista representada desta forma. Observe que existe uma **célula cabeça** que aponta para a célula que contém x_1.



Implementação de Listas através de Apontadores

```
// modelo matemático (estrutura de dados)
struct TipoItem {       // cada item da lista corresponde a um
  char nome[30];        // registro (TipoItem) composto apenas
};                                  // do campo nome

typedef struct Celula *Apontador;
struct Celula {
   TipoItem Item;
   Apontador prox;
};

struct TipoLista {
  Apontador Primeiro;
  Apontador Ultimo;
};
```

A lista é constituída de células, onde cada registro célula contém um *Item* da lista e um "apontador" para a célula seguinte (*prox*). O registro *TipoLista* contém um "apontador" para a célula cabeça (*Primeiro*) e um "apontador" para a última célula da lista (*Ultimo*).

Implementação de Listas através de Apontadores

A implementação através de apontadores permite inserir ou retirar itens do meio da lista a um custo constante, aspecto importante quando a lista tem que ser mantida em ordem.

Em aplicações em que não existe previsão sobre o crescimento da lista é conveniente usar **listas encadeadas** por apontadores, porque neste caso o tamanho máximo da lista não precisa ser definido a priori.

A maior desvantagem deste tipo de implementação é a utilização de memória extra para armazenar os apontadores.

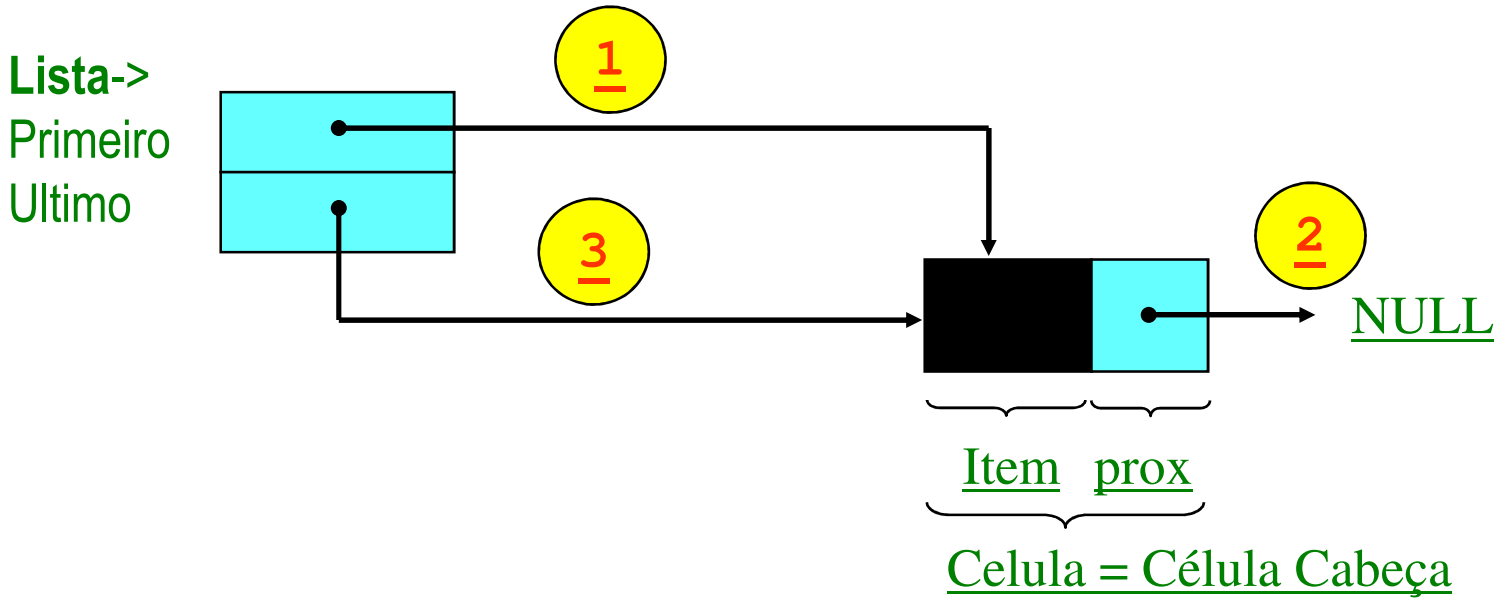
Implementação de Listas através de Apontadores

```
// Faz a 'Lista' ficar vazia criando a célula cabeça

void FazListaVazia(TipoLista *Lista) {
  Lista->Primeiro = (Apontador) malloc(sizeof(Celula));
  Lista->Primeiro->prox = NULL;
  Lista->Ultimo = Lista->Primeiro;
}
```

```
// Esta função retorna 1 (true) se a 'Lista' está vazia;
// senão retorna 0 (false)

int ListaVazia(TipoLista *Lista) {
  return(Lista->Primeiro == Lista->Ultimo);
}
```

Implementação de Listas através de Apontadores

```
void FazListaVazia(TipoLista *Lista) {
  1. Lista->Primeiro = (Apontador) malloc(sizeof(Celula));
  2. Lista->Primeiro->prox = NULL;
  3. Lista->Ultimo = Lista->Primeiro;
}
```



Implementação de Listas através de Apontadores

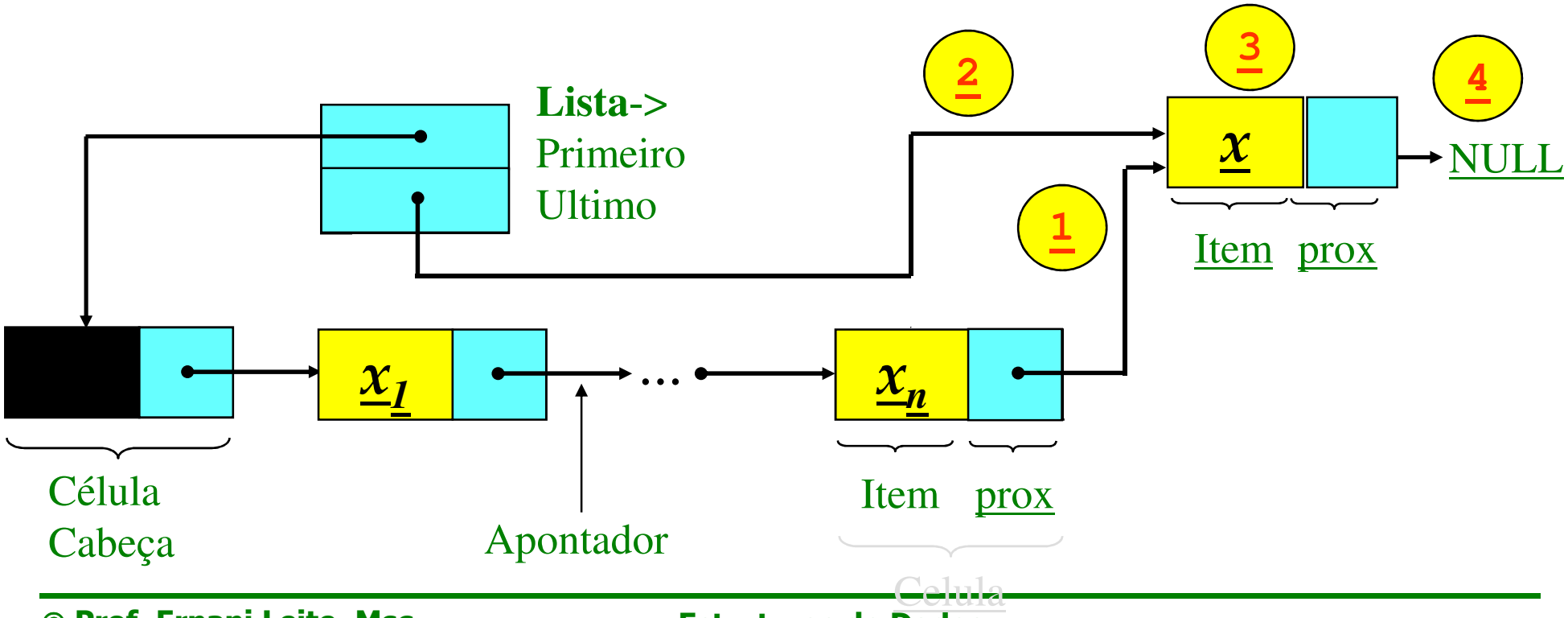
```
// Insere o item 'x' no final da 'Lista'
void Insere(TipoItem x, TipoLista *Lista) {

  Lista->Ultimo->prox = (Apontador) malloc(sizeof(Celula));

  Lista->Ultimo = Lista->Ultimo->prox;
  Lista->Ultimo->Item = x;
  Lista->Ultimo->prox = NULL;

}
```

a instrução **malloc** cria uma nova célula

insere ***x*** após o último item da lista

Implementação de Listas através de Apontadores

```
void Insere(TipoItem x, TipoLista *Lista) {
1. Lista->Ultimo->prox = (Apontador) malloc(sizeof(Celula));
2. Lista->Ultimo = Lista->Ultimo->prox;
3. Lista->Ultimo->Item = x;
4. Lista->Ultimo->prox = NULL;
}
```



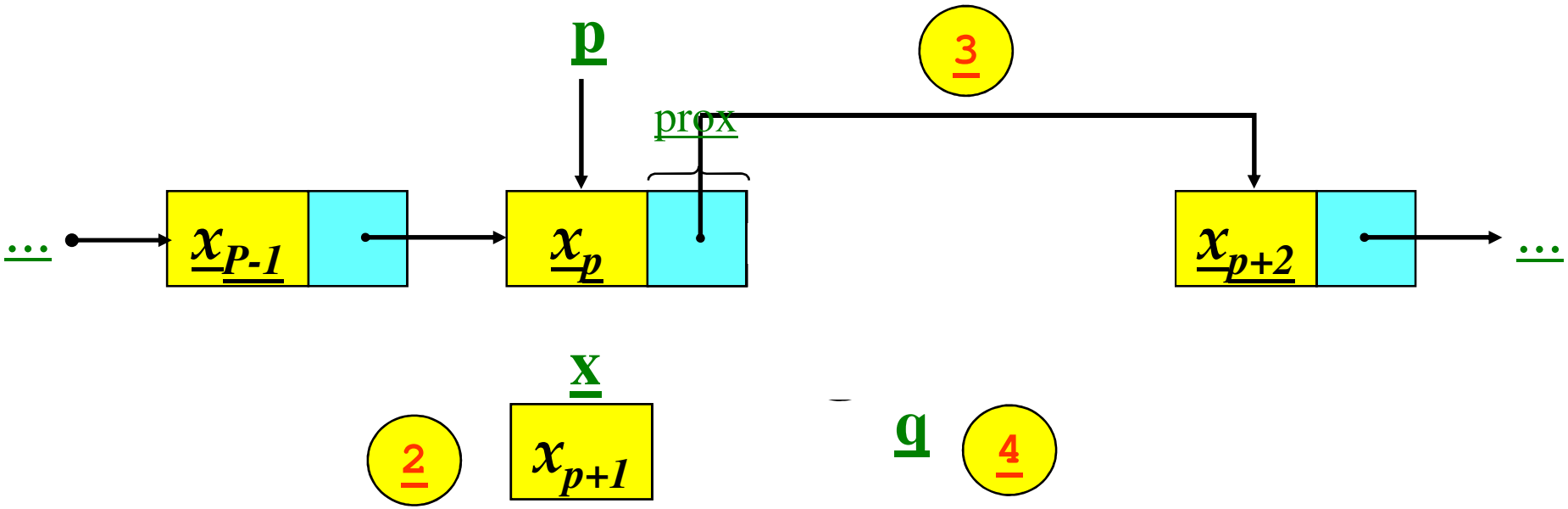
Implementação de Listas através de Apontadores

```
// Retira o item 'x' que o seguinte ao apontador por 'p'
int Retira(Apontador p, TipoLista *Lista, TipoItem *x) {
  if ((ListaVazia(Lista)) || (p == NULL) ||
      (p->prox == NULL))
     return(0); // Erro: Lista vazia ou posição inválida.
  else {
    Apontador q = p->prox;
    *x = q->Item; // item retornado
    p->prox = q->prox;
//  se retirando a última célula da lista
    if (p->prox == NULL)
       Lista->Ultimo = p;

    free(q);

    return(1); // Item retirado com sucesso
  }
}
```

a instrução **free** libera da memória a célula do item retirado da lista

Implementação de Listas através de Apontadores

```
// Retira o item 'x' que é o seguinte ao apontador por 'p'
int Retira(Apontador p, TipoLista *Lista, TipoItem *x) {
1. Apontador q = p->prox;
2. *x = q->Item; // item retornado
3. p->prox = q->prox;
4. free(q);
```



Implementação de Listas através de Apontadores

```
// Imprime os itens da 'Lista'
void Imprime(TipoLista *Lista) {
  clrscr();
  int i = 1;

  Apontador p = Lista->Primeiro->prox;
  while (p != NULL) {
    printf("%d- %s\n", i, p->Item.nome);
    i = i + 1;
    p = p->prox;
  }

}
```

posiciona no início da lista

avança para a próxima célula da lista

TAD Pilha (1/4)

Existem aplicações para listas lineares nas quais inserções, retiradas e acessos a itens ocorrem sempre em um dos extremos da lista. Uma **pilha** é uma lista linear em que todas as inserções, retiradas e geralmente todos os acessos são feitos em apenas **um** extremo da lista.

Os itens em uma pilha estão colocados um sobre o outro, com o item inserido mais recentemente no topo e o item inserido menos recentemente no fundo.

O modelo intuitivo de uma pilha é o de um monte de pratos em uma prateleira, sendo conveniente retirar os pratos ou adicionar novos pratos na parte superior.

INSTITUTO FEDERAL
CEARÁ

TAD Pilha (2/4)

As pilhas possuem a seguinte propriedade: **O ÚLTIMO ITEM INSERIDO É O PRIMEIRO ITEM QUE PODE SER RETIRADO DA LISTA**. Por esta razão as pilhas são chamadas de listas **LIFO**, termo formado a partir de "**Last-In, First-Out**".

Existe uma ordem linear para pilhas, que é a ordem do "**mais recente para o menos recente**".

Esta propriedade torna a pilha uma ferramenta ideal para processamento de estruturas aninhadas de profundidade imprevisível, situação em que é necessário garantir que subestruturas mais internas sejam processadas antes da estrutura que as contenham. A qualquer instante uma pilha contém uma sequência de obrigações adiadas, cuja ordem de remoção da pilha garante que as estruturas mais internas serão processadas antes das estruturas mais externas.

TAD Pilha (3/4)

Estruturas aninhadas ocorrem frequentemente na prática. Um exemplo simples é a situação em que é necessário caminhar em um conjunto de dados e guardar uma lista de coisas a fazer posteriormente.

O controle de chamadas de subprogramas e sintaxe de expressões aritméticas são exemplos de estruturas aninhadas.

As pilhas ocorrem também em conexão com **algoritmos recurssivos** e estrutura de natureza recursiva, tais como as árvores.

Em Síntese: Pilha é um caso especial de Lista Linear.

TAD Pilha (4/4)

Um **tipo abstrato de dados** Pilha, acompanhado de um conjunto de operações, é apresentado a seguir.

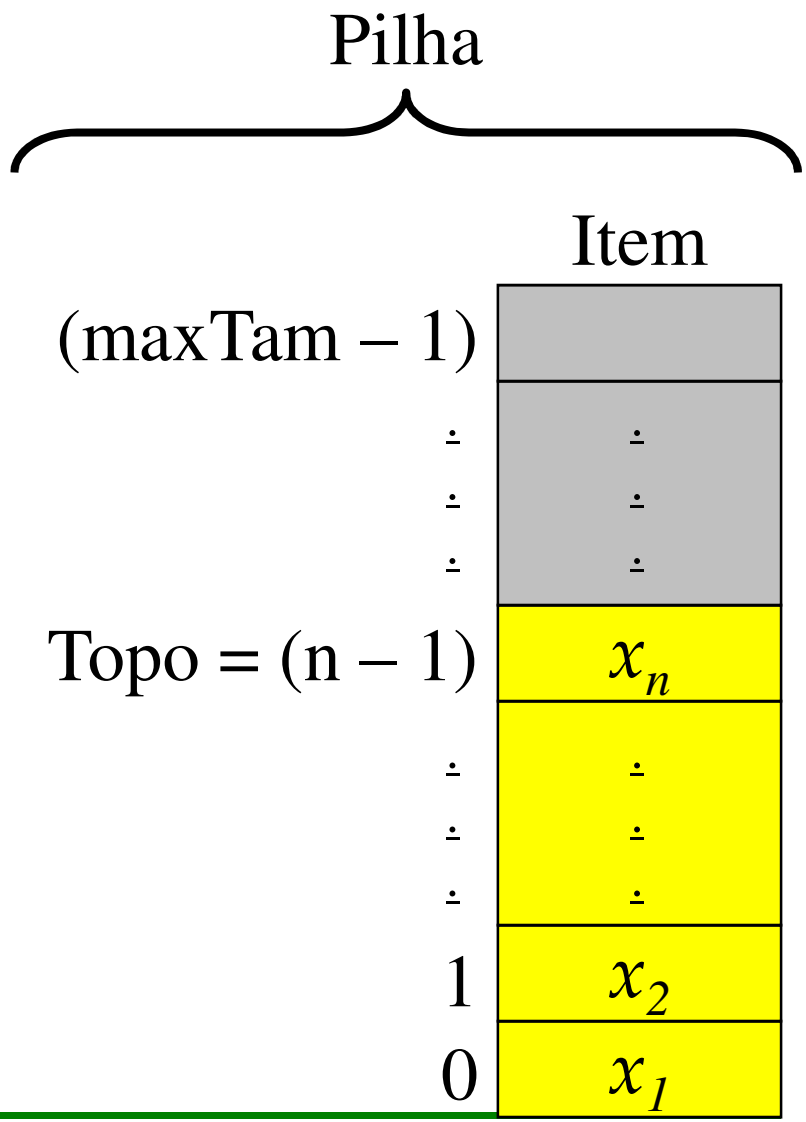
1. **FazPilhaVazia(Pilha)**. Faz a pilha ficar vazia.
2. **PilhaVazia(Pilha)**. Esta função retorna 1 (*true*) se a pilha está vazia; senão retorna 0 (*false*).
3. **Empilha(x, Pilha)**. Insere o item *x* no topo da pilha.
4. **Desempilha(Pilha, x)**. Retira o item *x* que esta no topo da pilha.
5. **ImprimePilha(Pilha)**. Imprime os itens da pilha (do mais recente para o menos recente).

Como no caso do tipo abstrato de dados Lista, existem várias opções de estruturas de dados que podem ser usadas para representar Pilhas. As duas representações mais utilizadas são as implementações através de *arranjos* (*vetores*) e de *apontadores*.

Implementação de Pilhas através de Arranjos

Em uma implementação através de arranjo, os itens da pilha são armazenados em posições contíguas de memória, conforme ilustra a Figura ao lado. Devido as características da pilha as operações de inserção e de retirada de itens devem ser implementadas de forma diferente das implementações apresentadas anteriormente para listas.

Como as inserções e as retiradas ocorrem no topo da pilha, um campo chamado *Topo* é utilizado para controlar a posição do item no topo da pilha.



Implementação de Pilhas através de Arranjos

```
// modelo matemático (estrutura de dados)
struct TipoItem {      // cada item da pilha corresponde a um
  char nome[30];       // registro (TipoItem) composto apenas
};                                  // do campo nome

const maxTam = 10;   // tamanho máximo da pilha

struct TipoPilha {
  TipoItem Item[maxTam];
  int Topo;
};
```

O campo *Item* é o principal componente do registro *TipoPilha*. Os itens são armazenados em um **vetor** de tamanho suficiente para armazenar a pilha. Já o campo *Topo* do registro *TipoPilha* contém o valor da posição do item que está no topo da pilha. A constante *maxTam* define o tamanho máximo permitido para a pilha.

Implementação de Pilhas através de Arranjos

```
// Faz a 'Pilha' ficar vazia

void FazPilhaVazia(TipoPilha *Pilha) {
  Pilha->Topo = -1;
}
```

```
// Esta função retorna 1 (true) se a 'Pilha'
// está vazia; senão retorna 0 (false)

int PilhaVazia(TipoPilha *Pilha) {
  return(Pilha->Topo == -1);
}
```

Implementação de Pilhas através de Arranjos

```
// Insere o item 'x' no 'Topo' da 'Pilha'.
int Empilha(TipoItem x, TipoPilha *Pilha) {

  if (Pilha->Topo == (maxTam – 1))
     return(0);    // Erro: Pilha Cheia !

  else {
    Pilha->Topo = Pilha->Topo + 1;
// item inserido no topo da pilha
    Pilha->Item[Pilha->Topo] = x;
    return(1);    // Item inserido com sucesso
  }
}
```

Implementação de Pilhas através de Arranjos

```
// Retira o item 'x' que está no topo da 'Pilha'
int Desempilha(TipoPilha *Pilha, TipoItem *x) {
  if (PilhaVazia(Pilha))
     return(0); // Erro: Pilha vazia.

  else {
// item retornado
    *x = Pilha->Item[Pilha->Topo];
    Pilha->Topo = Pilha->Topo - 1;
    return(1); // Item retirado com sucesso
  }
}
```

Implementação de Pilhas através de Arranjos

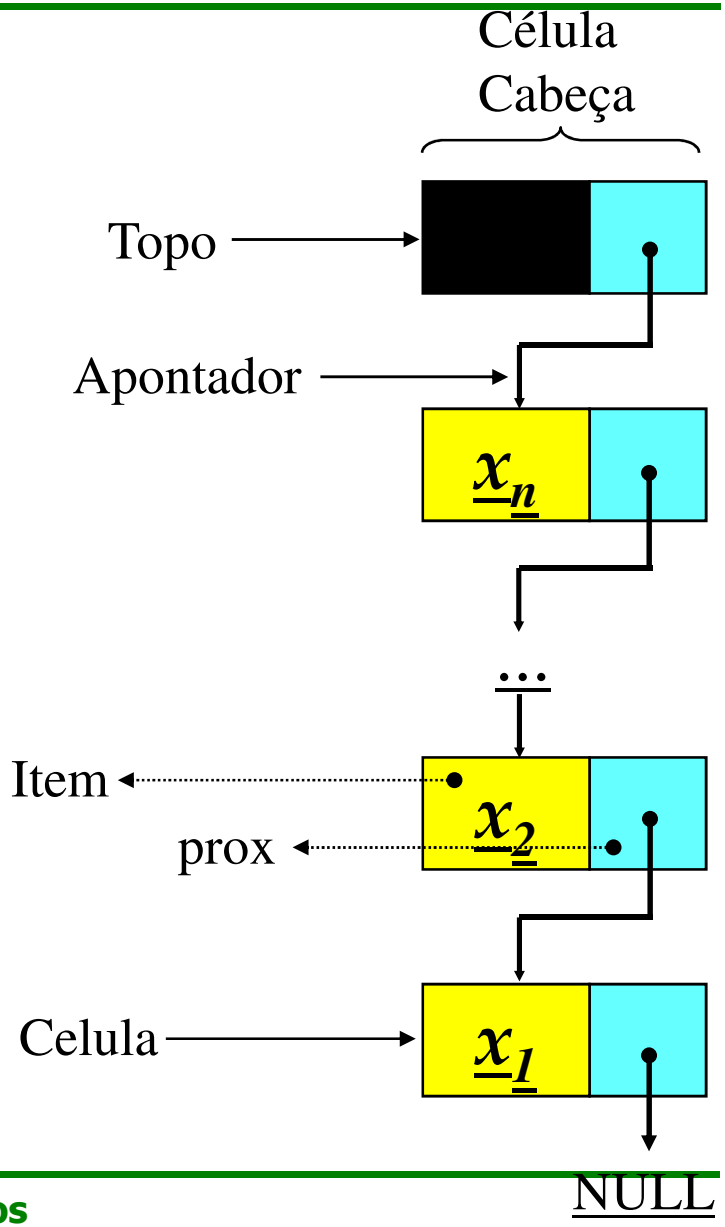
```
// Imprime os itens da pilha (do mais recente para o menos recente).
void ImprimePilha(TipoPilha *Pilha) {
  TipoItem x;
  TipoPilha PilhaAux;
  FazPilhaVazia(&PilhaAux);
  clrscr();
  while (!PilhaVazia(Pilha)) {
    Desempilha(Pilha, &x);
    printf("%s\n", x.nome);
// salva os itens desempilhados da 'Pilha' na 'PilhaAux'
    Empilha(x, &PilhaAux);
  }
// retorna o itens para a pilha original (Pilha)
  while (!PilhaVazia(&PilhaAux)) {
    Desempilha(&PilhaAux, &x);
    Empilha(x, Pilha);
  }
}
```

Implementação de Pilhas através de Apontadores

Uma célula cabeça é mantida no topo da pilha para facilitar a implementação das operações empilha e desempilha quando uma pilha está vazia.

Para desempilhar o item x_n da pilha basta desligar a célula cabeça da lista e a célula que contém x_n passa a ser a célula cabeça.

Para empilhar um novo item basta fazer a operação contrária, criando uma nova célula cabeça e colocando o novo item na antiga célula cabeça.



Implementação de Pilhas através de Apontadores

```
// modelo matemático (estrutura de dados)
struct TipoItem {      // cada item da pilha corresponde a um
  char nome[30];       // registro (TipoItem) composto apenas
};                                  // do campo nome

typedef struct Celula *Apontador;
struct Celula {
   TipoItem Item;
   Apontador prox;
};

struct TipoPilha {
  Apontador Topo;
};
```

Cada célula de uma pilha contém um item da pilha (campo *Item*) e um "apontador" para outra célula (campo *prox*). O registro *TipoPilha* possui o campo *Topo* que é o "apontador" para o topo da pilha (célula cabeça).

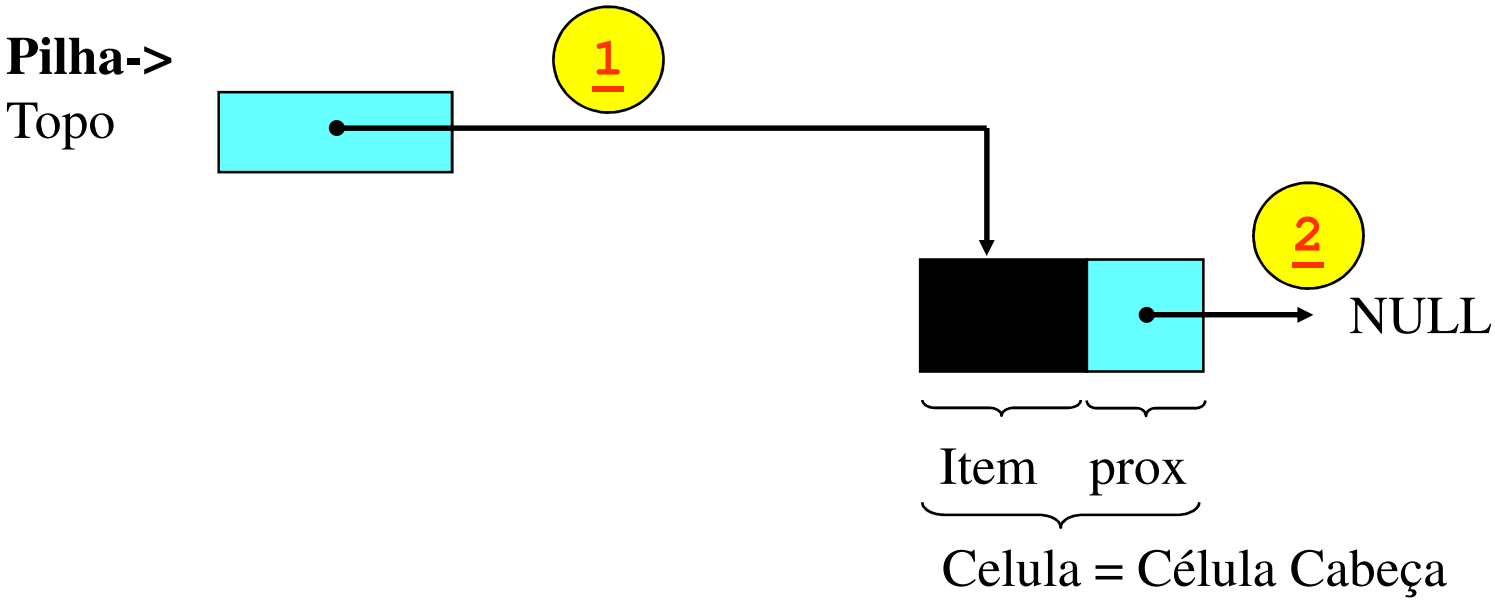
Implementação de Pilhas através de Apontadores

```
// Faz a 'Pilha' ficar vazia criando a célula cabeça
void FazPilhaVazia(TipoPilha *Pilha) {
  Pilha->Topo = (Apontador) malloc(sizeof(Celula));
  Pilha->Topo->prox = NULL;
}
```

```
// Esta função retorna 1 (true) se a 'Pilha' está vazia;
// senão retorna 0 (false)

int PilhaVazia(TipoPilha *Pilha) {
  return(Pilha->Topo->prox == NULL);
}
```

Implementação de Pilhas através de Apontadores

```
void FazPilhaVazia(TipoPilha *Pilha) {
  1. Pilha->Topo = (Apontador) malloc(sizeof(Celula));
  2. Pilha->Topo->prox = NULL;
}
```

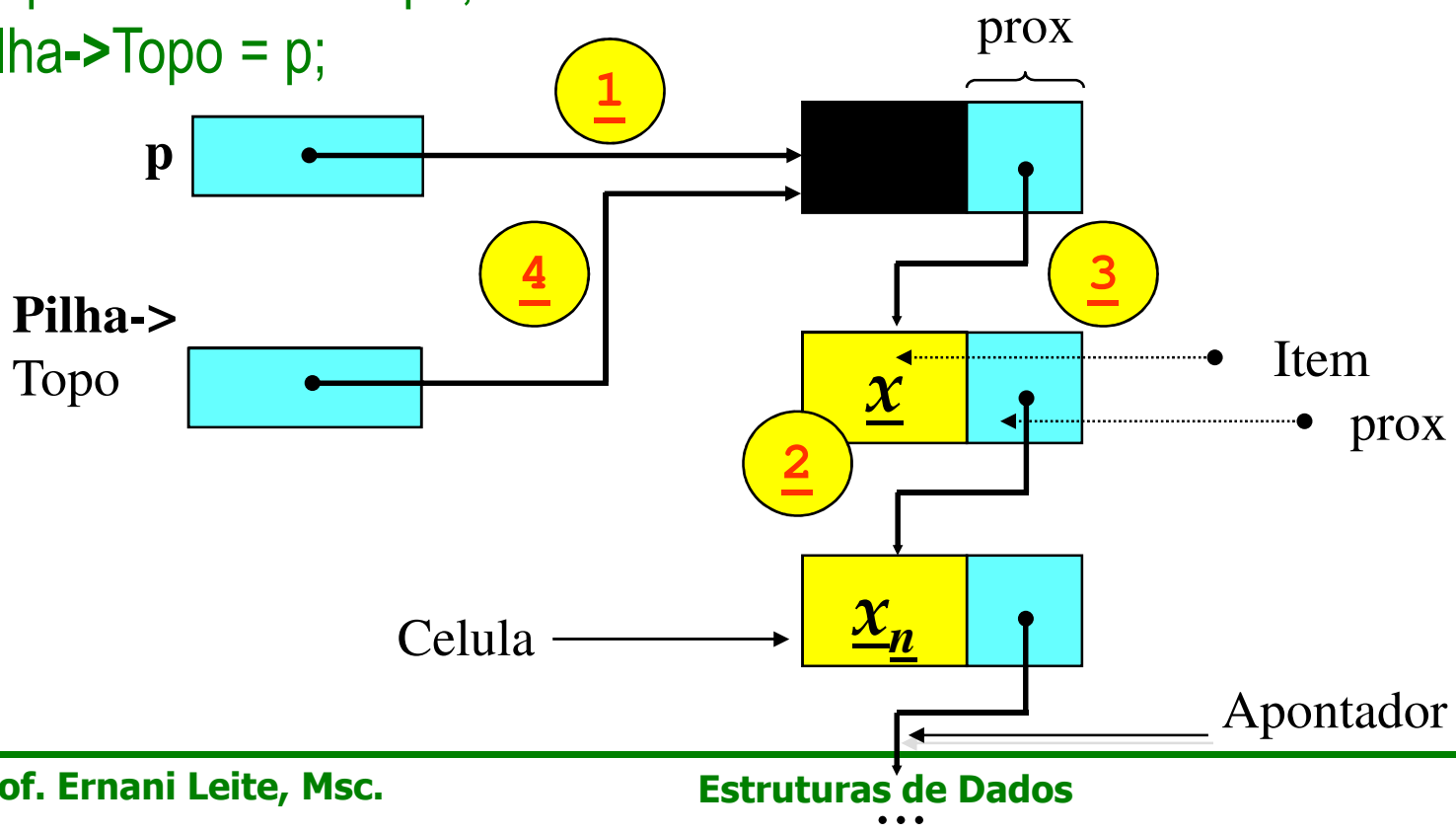


Implementação de Pilhas através de Apontadores

```
// Insere o item 'x' no 'Topo' da 'Pilha'.
void Empilha(TipoItem x, TipoPilha *Pilha) {
  Apontador p;

// cria uma nova célula cabeça
  p = (Apontador) malloc(sizeof(Celula));

// coloca o item "x" no topo da pilha, ou seja,
// na antiga célula cabeça
  Pilha->Topo->Item = x;

// atualiza o topo da pilha
  p->prox = Pilha->Topo;
  Pilha->Topo = p;
}
```

Implementação de Pilhas através de Apontadores

```
void Empilha(TipoItem x, TipoPilha *Pilha) {
Apontador p;
1. p = (Apontador) malloc(sizeof(Celula));
2. Pilha->Topo->Item = x;
3. p->prox = Pilha->Topo;
4. Pilha->Topo = p;
```

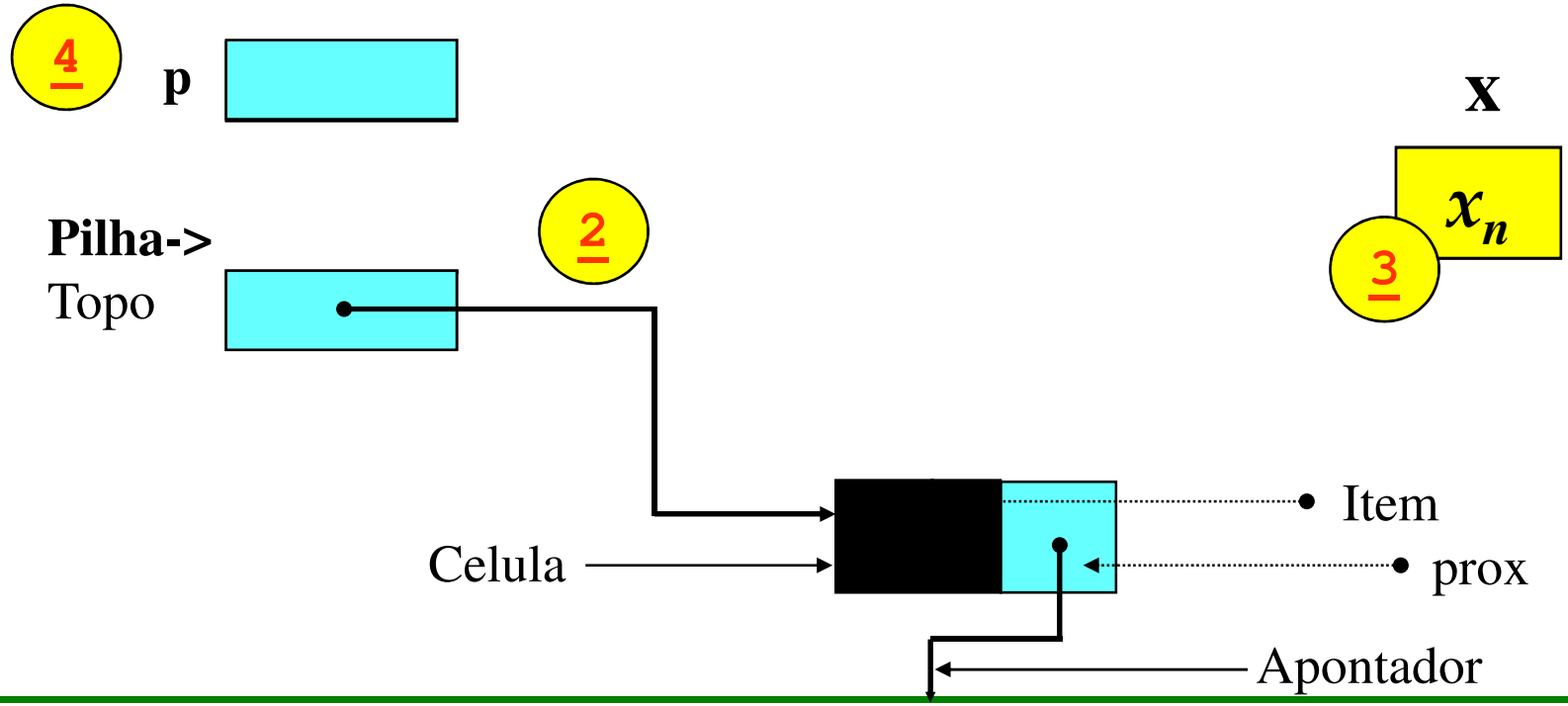


Implementação de Pilhas através de Apontadores

```
// Retira o item 'x' que está no topo da 'Pilha'
int Desempilha(TipoPilha *Pilha, TipoItem *x) {
  if (PilhaVazia(Pilha))
     return(0); // Erro: Pilha vazia.
  else {
// retira o item x_n do topo da pilha e desliga
// a célula cabeça, a célula que contém x_n
// torna-se a nova célula cabeça
     Apontador p;
     p = Pilha->Topo;
     Pilha->Topo = Pilha->Topo->prox;
     *x = Pilha->Topo->Item; // item retornado
     free(p);
     return(1); // Item retirado com sucesso
  }
}
```

Implementação de Pilhas através de Apontadores

```
int Desempilha(TipoPilha *Pilha, TipoItem *x) {
Apontador p;
1. p = Pilha->Topo;
2. Pilha->Topo = Pilha->Topo->prox;
3. *x = Pilha->Topo->Item;
4. free(p);
```



Implementação de Pilhas através de Apontadores

```
// Imprime os itens da pilha (do mais recente // para o menos recente).
void ImprimePilha(TipoPilha *Pilha) {
  TipoItem x;
  TipoPilha PilhaAux;
  FazPilhaVazia(&PilhaAux);
  clrscr();
  while (!PilhaVazia(Pilha)) {
    Desempilha(Pilha, &x);
    printf("%s\n", x.nome);
// salva os itens desempilhados da 'Pilha' na 'PilhaAux'
    Empilha(x, &PilhaAux);
  }
// retorna o itens para a pilha original (Pilha)
  while (!PilhaVazia(&PilhaAux)) {
    Desempilha(&PilhaAux, &x);
    Empilha(x, Pilha);
  }
}
```

INSTITUTO FEDERAL
CEARÁ

Referências

- Estrutura de Dados Fundamentais: conceitos e aplicações.
  - Silvio do Lago Pereira.
  - 2ª ed. - São Paulo: Érica, 1996.
- Instituto de Computação da UNICAMP
  - Flávio K. Miyazawa & Tomasz Kowaltowski
- Material adaptado do Prof.: Omero Francisco Bertol - UTFPR.