# Listas Encadeadas

Algoritmos e Programação 2
Prof. Dr. Anderson Bessa da Costa
Universidade Federal de Mato Grosso do Sul

# Problema: Inserir ordenado

- Escreva uma função que receba um vetor de inteiros ordenado (crescente) $v$, o seu tamanho $n$ e um inteiro $x$. Insira o elemento $x$ no local correto do vetor $v$, de forma que o mesmo se mantenha ordenado após a inserção.

# Limitações de Vetores

- Vetores possuem algumas limitações:
  - Inserir novo elemento entre dois elementos é $O(n)$ no pior caso
  - Quando alocado estaticamente, tamanho é definido em tempo de compilação
  - Mesmo usando alocação dinâmica, redimensionar exige criar um novo alocar novo vetor e copiar elementos

# Listas Encadeadas

- **Lista encadeada** (ou **lista ligada**) é uma representação de uma sequência de objetos na memória do computador
- Cada elemento é armazenado em uma "**célula**" da lista
- Células que armazenam elementos consecutivos da sequência não ficam necessariamente em posições consecutivas da memória

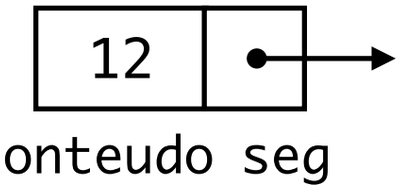
# Definição

- **Lista encadeada** é uma sequência de **células**
- Cada célula contém um objeto de determinado tipo e o endereço da célula seguinte (no caso do último célula, esse endereço é **NULO**)

# Definição (cont.)

- Suponha que os objetos armazenados nas **células** são números inteiros:

```
struct celula {
    int conteudo;
    struct celula *seg;
    // podem existir outros dados
};
```




- *seg* deve ser um ponteiro. Caso contrário, será uma declaração recursiva e irá gerar erro.

# Tipo Célula

- É conveniente tratar as **células** como um novo tipo de dados, que chamaremos **celula**:

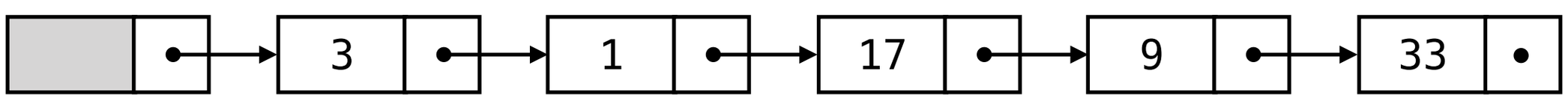
```
typedef struct celula celula;
```

- Uma **celula** $c$ e um ponteiro $p$ para uma **celula** podem ser agora declarados assim:

```
celula c;
celula *p;
```

# Tipo Célula (cont.)

- Se $c$ é uma célula então `c.conteudo` é o conteúdo da célula e `c.seg` é o endereço da célula seguinte
- Se $p$ é o endereço de uma célula, então `p->conteudo` é o conteúdo da célula e `p->seg` é o endereço da célula seguinte
- Se $p$ é o endereço da última célula da lista então `p->seg` contém `NULL`



# Endereço Lista Encadeada

- Endereço de uma **lista encadeada** é o endereço de sua primeira célula
- Se $p$ é o endereço de uma lista, podemos dizer, simplesmente, "$p$ é uma lista" e "considere a lista $p$"
- Reciprocamente, a expressão "$p$ é uma lista" deve ser interpretada como "$p$ é o endereço do primeiro nó de uma lista"

# Listas “com cabeça” e “sem cabeça”

- **Lista encadeada** pode ser vista de duas maneiras:
  - Com cabeça
    - **Primeira célula** sempre existe, mesmo se lista vazia
    - Não armazena conteúdo na cabeça
  - Sem cabeça
    - **Primeira célula** é uma célula ordinária
    - Se lista vazia, não possui nenhuma célula

# Listas Com Cabeça

- Primeira **célula:**
  - Apenas para marcar o início da lista
  - Conteúdo é irrelevante
  - Cabeça da lista
- Para criar uma lista vazia deste tipo, basta dizer

```
celula c, *lst;

c.seg = NULL;
lst = &c;
```

ou

```
celula *lst;

lst = malloc(sizeof(celula));
lst->seg = NULL;
```

# Listas Sem Cabeça

- Conteúdo da primeira **célula** é tão relevante quanto as demais
- Lista está vazia se não tem **célula** alguma
- Para criar uma lista vazia `lst` basta dizer

```
celula *lst;

lst = NULL;
```

**Trataremos a listas com cabeça nessa aula**

# Busca em Lista Encadeada

- É fácil verificar se um objeto $x$ pertence a uma lista encadeada, ou seja, se $x$ é igual ao conteúdo de alguma célula da lista

# Algoritmo: Busca

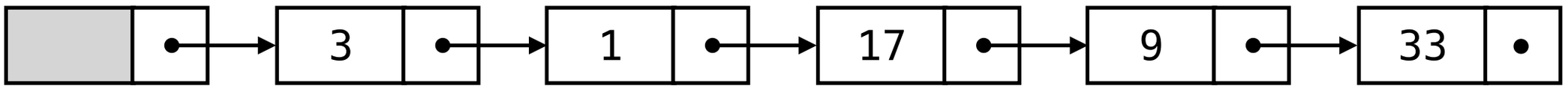
```c
/* Esta função recebe um inteiro x e uma lista encadeada lst
 com cabeca. Devolve o endereco de uma celula que contem x ou
 devolve NULL se tal celula não existe */
celula* busque (int x, celula * lst) {
    celula *p;

    // p aponte para o inicio da lista
    p = lst->seg;

    // enquanto não cheguei ao fim da lista e não encontrei o x
    while (p != NULL && p->conteudo != x)
        // p aponta para o elemento seguinte
        p = p->seg;

    return p;
}
```

# Algoritmo: Busca recursiva

```c
/* Esta função recebe um inteiro x e uma lista encadeada lst
 com cabeca. Devolve o endereco de uma celula que contem x ou
 devolve NULL se tal celula não existe */
celula* busqueR (int x, celula * lst) {
    if (lst->seg == NULL)
        return NULL;
    else if (lst->seg->conteudo == x)
        return lst->seg;
    else
        return buscaR(x, lst->seg);
}
```

# Exemplo: Busca

- O que a função retorna se eu buscar pelo elemento 17?
- E pelo 4?



# Remoção em Lista Encadeada

- Como devemos especificar a célula a ser removido?
- Parece natural apontar para a célula em questão, mas é fácil perceber o defeito da ideia
- É melhor apontar para a célula anterior ao que queremos remover
- É bem verdade que esta convenção não permite remover a primeira célula da lista, mas esta operação não é necessária no caso de listas com cabeça

# Algoritmo: Remoção

```
/* Esta função recebe o endereço p de uma celula em uma
 lista encadeada e remove da lista a celula p->seg.
 A função supõe que p != NULO e p->seg != NULO */
void remova (celula *p) {
    celula *lixo;

    lixo = p->seg;
    p->seg = lixo->seg;
    free(lixo);
}
```

# Inserção de uma Nova Célula

- Suponha que queremos inserir uma nova célula com conteúdo $y$ entre a célula apontado por $p$ e o seguinte
  - É claro que isso só faz sentido se $p$ for diferente de NULO

# Algoritmo: Inserção

```
/* A função insere uma nova célula em uma lista encadeada
 entre a célula p e o seguinte (supõe-se que p != NULO).
 A nova célula terá conteúdo y */
void insira (int y, celula *p) {
    celula *nova;

    nova = (celula *) malloc(sizeof(celula));
    nova->conteudo = y;
    nova->seg = p->seg;
    p->seg = nova;
}
```

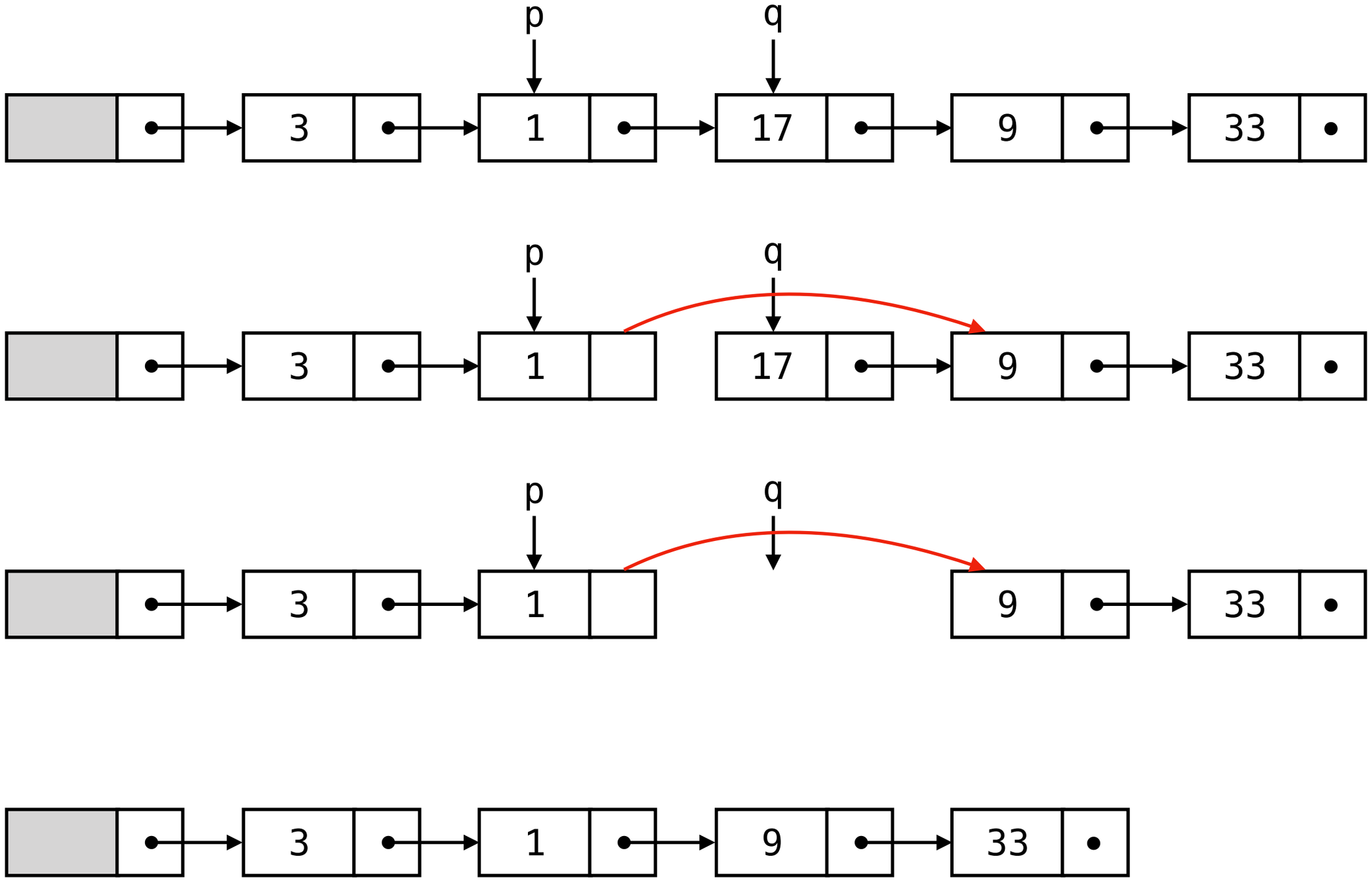
# Busca Seguida de Remoção

- Dado um inteiro $x$, queremos remover da lista a primeira célula que contiver $x$
- Se tal célula não existe, não é preciso fazer nada

# Algoritmo: Busca Seguida de Remoção

```
/* Esta função recebe uma lista encadeada lst com cabeça e remove da
lista
 a primeira célula que contiver x, se tal célula existir */
void busque_e_remova (int x, celula *lst) {
    celula *q, *p;

    p = lst;
    q = lst->seg;

    while (q != NULL && q->conteudo != x) {
        p = q;
        q = q->seg;
    }

    // se encontrou elemento
    if (q != NULL) {
        p->seg = q->seg;
        free(q);
    }
}
```

# Exemplo: Busca 17 e remove



# Busca Seguida de Inserção

- Queremos inserir uma nova **célula** com conteúdo
- Imediatamente antes da primeira **célula** que tiver conteúdo $x$
- Se tal **célula** não existe, devemos inserir $y$ no fim da lista

# Algoritmo: Busca seguida de inserção

```c
/* Recebe uma lista encadeada lst com cabeça e insere uma nova célula
na lista imediatamente antes da primeira que contiver x. Se nenhuma
célula contiver x, a nova célula será inserida ao final da lista. A
nova célula terá conteúdo y */
void busque_e_insira (int y, int x, celula *lst) {
    celula *q, *p, *nova;

    nova = (celula *) malloc((sizeof(celula)));
    nova->conteudo = y;
    p = lst;
    q = lst->seg;

    while (q != NULL && q->conteudo != x) {
        p = q;
        q = q->seg;
    }

    nova->seg = q;
    p->seg = nova;
}
```

# Outros Tipos Listas Ligadas

- Poderíamos definir vários tipos de listas encadeadas além do tipo básico
  - Numa **lista encadeada circular**, o último nó aponta para o primeiro
  - Numa **lista duplamente encadeada**, cada nó contém o endereço do nó anterior e o do nó seguinte

# Outros Tipos Listas Ligadas (Cont.)

As seguintes questões são apropriadas para qualquer tipo de lista ligada

- Em que condições a lista está vazia? Como remover o nó apontado por $p$? Como remover o nó seguinte ao apontado por $p$? Como remover o nó anterior ao apontada por $p$? Como inserir um novo nó entre a apontada por $p$ e o anterior? Como inserir um novo nó entre o apontado por $p$ e o seguinte?

# Referências


- FEOFILOFF, P. Algoritmos em Linguagem C, 1. ed. Rio de Janeiro: Elsevier, 2008.
- SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. Estruturas de Dados e seus Algoritmos. Edição: 3a. Editora: LTC. 2010
- CORMEN, T. H.[et al]. Algoritmos: teoria e prática. 3ª ed. Rio de Janeiro: Elsevier, 2012.