Estruturas de Dados

Listas Dinâmicas Duplamente Encadeadas & Variantes

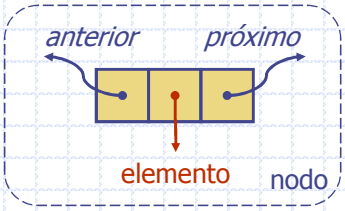
Prof. Ricardo J. G. B. Campello

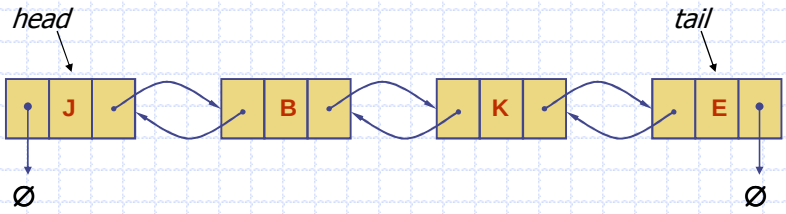
Listas Duplamente Encadeadas

- Listas simplesmente encadeadas são ineficientes para realizar certas operações. Por exemplo:
  - Remover o último elemento ou um elemento intermediário
    - É preciso percorrer toda lista para encontrar o elemento anterior
- Muitas aplicações não demandam tais operações. P. ex:
  - Pilhas e filas podem ser implementadas eficientemente apenas removendo elementos do início de uma lista
- E quando tais operações são necessárias ?
  - Podemos utilizar uma **lista duplamente encadeada**

Listas Duplamente Encadeadas

- No caso de encadeamento duplo, cada nodo armazena:
  - elemento (registro, objeto, ...)
  - *ponteiro* para o próximo nodo
  - *ponteiro* para o nodo anterior

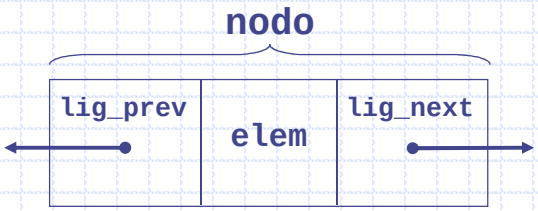




3

Listas Duplamente Encadeadas

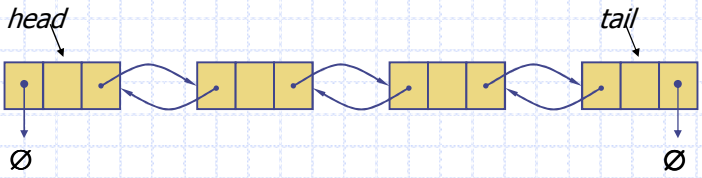
- **Nodo:**




```
typedef struct list_node {
          tipo_elem  elem;
          struct list_node  *lig_next, *lig_prev;
        } nodo;
```

4

Listas Duplamente Encadeadas



- **Lista (Idem ao caso simplesmente encadeado):**

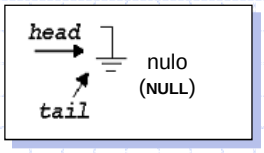
```
typedef struct {
          int nelem;
          nodo *head, *tail;
        } Lista;
```

5

Listas Duplamente Encadeadas

- **Inicialização (Idem ao caso simplesmente encadeado):**

```
Lista *Definir(void){
      Lista *L;
      L = malloc(sizeof(Lista));
      L->nelem = 0;
      L->head = NULL;
      L->tail = NULL;
      return L;
}
```



6

Inserindo no Início

```
nodo *Inserir_frente(tipo_elem x, Lista *L){
  nodo *Pa;
  Pa = malloc(sizeof(nodo));
  Pa->elem = x;
  Pa->lig_next = L->head;
  Pa->lig_prev = NULL;                          /* Mudança */
  L->head = Pa;
  if (L->tail == NULL) L->tail = L->head;
  else ((L->head)->lig_next)->lig_prev = Pa; /* Mudança */
  L->nelem++;
  return Pa;
}                                    /* O(1) */
```

7

Removendo do Início

```
tipo_elem Remover_frente(Lista *L){
   tipo_elem x;
   nodo *Pa;
   Pa = L->head;
   L->head = Pa->lig_next;
   if (L->head == NULL) L->tail = NULL;
   else (L->head)->lig_prev = NULL; /* Única mudança */
   x = Pa->elem;
   free(Pa);
   L->nelem--;
   return x;
}                                        /* O(1) */
```

8

Inserindo no Final

```
nodo *Inserir_final(tipo_elem x, Lista *L){
  nodo *Pa;
  Pa = malloc(sizeof(nodo));
  Pa->elem = x;
  Pa->lig_next = NULL;
  Pa->lig_prev = L->tail;   /* Única mudança */
  if (L->head == NULL) L->head = Pa;
  else (L->tail)->lig_next = Pa;
  L->tail = Pa;
  L->nelem++;
  return Pa;
}                                        /* O(1) */
```

9

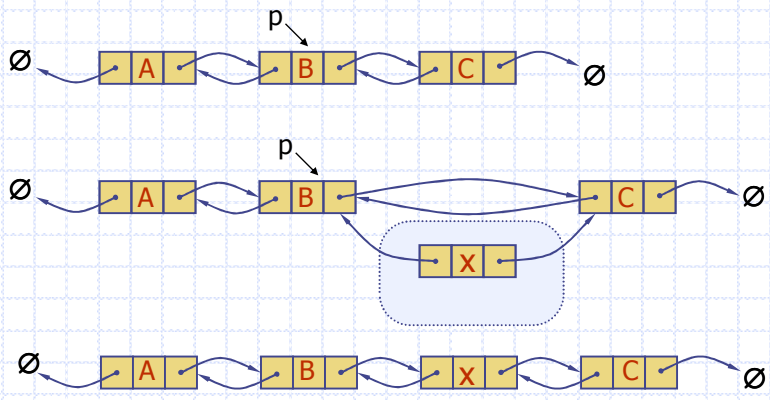
Removendo do Final

```
tipo_elem Remover_final(Lista *L){
      tipo_elem x;
      nodo Pa*;
      Pa = L->tail;
      L->tail = (L->tail)->lig_prev;
      if (L->tail == NULL) L->head = NULL;
      else (L->tail)->lig_next = NULL;
      x = Pa->elem;
      free(Pa);
      L->nelem--;
      return x;
}                                  /* O(1) !!! */
```

10

Inserção Intermediária

```
nodo *Inserir_apos(nodo *p, tipo_elem x, Lista *L);
```



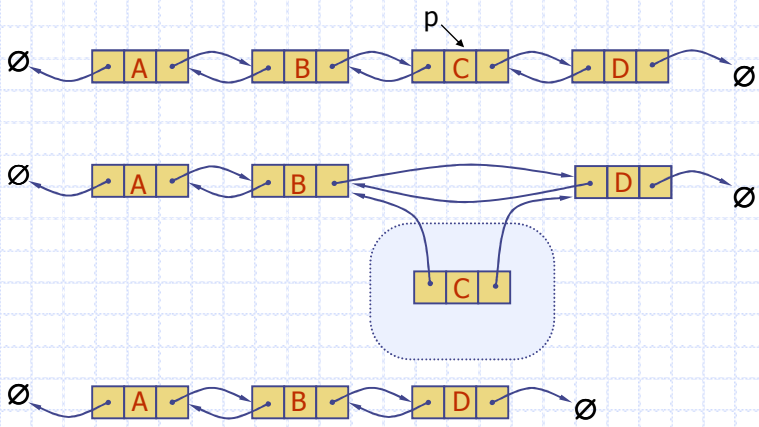
11

Inserção Intermediária

```
nodo *Inserir_apos(nodo *p, tipo_elem x, Lista *L){
  nodo *Pa;
  if (p == L->tail)
       return Inserir_final(x, L);
  else {
       Pa = malloc(sizeof(nodo));
       Pa->elem = x;
       Pa->lig_next = p->lig_next;
       Pa->lig_prev = p;
       p->lig_next = Pa;
       (Pa->lig_next)->lig_prev = Pa;
       L.nelem++;
       return Pa;
  }
}                                        /* O(1) */
```

Remoção Intermediária

```
tipo_elem Remover(nodo *p, Lista *L);
```



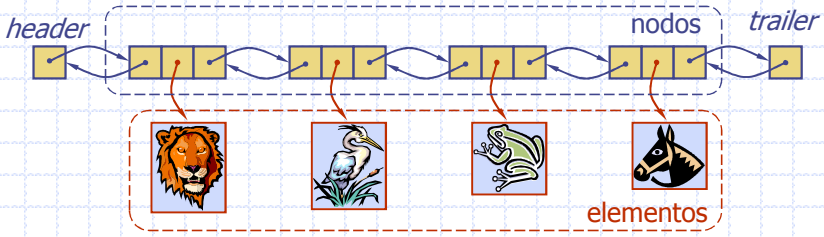
13

Remoção Intermediária

```
tipo_elem Remover(nodo *p, Lista *L){
  tipo_elem x;
  if (p == L->head) return Remover_frente(L);
  else if (p == L->tail) return Remover_final(L);
       else {
              (p->lig_prev)->lig_next = p->lig_next;
              (p->lig_next)->lig_prev = p->lig_prev;
              x = p->elem;
              free(p);
              L->nelem--;
              return x;
       }
}                                          /* O(1) */
```

Variantes

- Lista com **Sentinelas**:
  - Nodos especiais *header* e *trailer*
    - Não armazenam elementos
    - *header* possui campo *lig_prev* nulo (**NULL**)
    - *trailer* possui campo *lig_next* nulo (**NULL**)

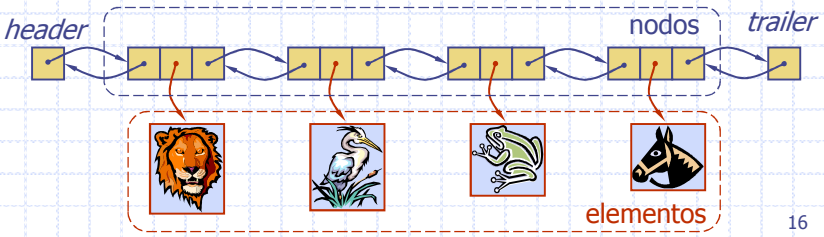


15

Variantes

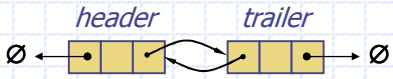
- **Lista com Sentinelas:**

```
typedef struct {
          int nelem;
          nodo header, trailer;
        } Lista;
```



16

Variantes



- Lista com Sentinelas (Inicialização):

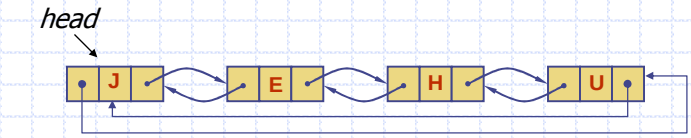
```
Lista *Definir(void){
   Lista *L = malloc(sizeof(Lista));
   L->nelem = 0;
   (L->header).lig_prev = NULL;
   (L->header).lig_next = &(L->trailer);
   (L->trailer).lig_next = NULL;
   (L->trailer).lig_prev = &(L->header);
  return L;
}
```

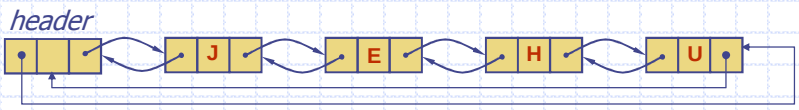
- Inserção e Remoção:
  - Dispensa diferenciar entre nodos intermediários e extremos

17

Variantes

- Lista **Circular**:
  - Último nodo aponta para o primeiro e vice-versa
  - Pode ser com sentinela ou não





18

Resumo (Listas Estáticas vs Dinâmicas)

- **Desvantagens**:
  - **Listas Estáticas (Seqüenciais)**:
    - Inserir/remover elem. intermediários requer deslocamentos ($O(n)$)
    - Exige previsão de espaço
  - **Listas Dinâmicas Simplesmente Encadeadas**:
    - Remover no final ou em posição intermediária é ineficiente ($O(n)$)
    - Acesso por colocação (*rank*) na lista não é direto ($O(n)$)
  - **Listas Dinâmicas Duplamente Encadeadas**:
    - Acesso por colocação (*rank*) na lista não é direto ($O(n)$)

19

Exercícios

1. 1.a. Indique quais das seguintes operações, pedidas como exercício na aula anterior (listas dinâmicas simplesmente encadeadas), não requerem modificações para o caso duplamente encadeado:
   - `Localizar, Remover_elem, Remover_rank, Buscar_elem, Buscar_rank, Modificar, Modify`
   - Nota - Considere apenas a versão sem sentinela e não circular

   1.b. Reimplemente todas as operações do item anterior que requerem modificações, discutindo o tempo de execução assintótico (BIG-O) de cada uma
2. Mostre a definição de `Lista` em C para as variantes circulares com e sem sentinela, bem como o procedimento de inicialização `Definir` para ambos os casos

Exercícios

3. Modifique as funções:
   - `Inserir_frente`
   - `Inserir_final`
   - `Inserir_apos`
   - `Remover_frente`
   - `Remover_final`
   - `Remover`

   para que essas operem apropriadamente para as variantes de lista:
   - Não circular com sentinelas
   - Circular sem sentinela
   - Circular com sentinela

   e discuta os tempos de execução assintóticos em cada um dos casos

Bibliografia

- A. M. Tenembaum et al., *Data Structures Using C*, Prentice-Hall, 1990
- M. T. Goodrich & R. Tamassia, *Data Structures and Algorithms in C++/Java*, John Wiley & Sons, 2002/2005
- N. Ziviani, *Projeto de Algoritmos*, Thomson, 2a. Edição, 2004
- J. L. Szwarcfiter & L. Markenzon, *Estruturas de Dados e seus Algoritmos*, LTC, 1994
- Schildt, H. "C Completo e Total", 3a. Edição, Pearson, 1997