BUSCA EM LISTAS

LISTAS SEQÜENCIAIS, LISTAS SIMPLESMENTE E DUPLAMENTE ENCADEADAS E LISTAS CIRCULARES

ALGORITMOS DE BUSCA EM LISTAS COM ALOCAÇÃO SEQÜENCIAL

Busca em Listas Lineares

- **A operação de** busca **é uma das três mais freqüentes numa lista, junto com a** inserção **e a** remoção **de um elemento**
- **Cada nó da lista é formado por registros que contêm um campo *chave***
  - **Considera-se que todas as chaves da lista** são distintas**: por exemplo, numa lista de alunos, a chave é o no. de matrícula**
  - **Esta chave será a** *chave de busca;* **é este campo que será testado, e nenhum outro**
- **Os registros da lista podem estar ordenados ou não de acordo com o valor desta chave:**
  - **Isto dá origem a** listas ordenadas **e** não-ordenadas**, sendo que essa organização é fundamental para a definição do algoritmo de busca**

Algoritmos de Busca

- **O algoritmo de busca retorna a** posição **do elemento cuja chave é igual à fornecida como parâmetro ou -1, se o elemento não for encontrado na lista**
- **O algoritmo trivial de busca tem complexidade O(*n*), onde *n* é o número de elementos da lista:**
  - **Partindo-se do 1o. elemento da lista, a lista é varrida até que o elemento seja encontrado ou que o último item da lista seja encontrado**
- **O algoritmo de busca trivial pode ser melhorado com a inserção da chave procurada no fim da lista, evitando que dois testes sejam feitos durante a busca**

Solução Trivial: Função Busca_Lista0

```
Ponteiro Busca_Lista0(TipoLista *L, TipoChave x) {
   Apontador i;
   i = 0;
   while(i != L->Ultimo) {
      if (L->Item[i].Chave == x)
        return i;
      else
        i++;
   }
   return -1; //não achou Item
}
```

Dois testes que podem ser substituídos

```
typedef int Apontador;
```

Solução 1: Função Busca_Lista

```
Ponteiro Busca_Lista(TipoLista *L, TipoChave x) {
    Apontador pos;
    if (L->Ultimo > MaxTam) {
           printf ("Erro na busca");
           return -1;
    } else {
           pos = 0;
           L->Item[L.Ultimo].Chave = x;
           while (L->Item[pos].Chave != x) {
                 pos++;
            }
           if (pos == L->Ultimo) return -1;
           else return pos;
     }
}
```

```
typedef int Apontador;
```

Busca em Lista Ordenada

- **Pode-se usar o fato de que as chaves estão ordenadas segundo algum critério para evitar que toda a lista seja varrida (em boa parte dos casos):**
  - **Seja uma lista L={1,3,4,5,7,8,10} e chave_buscada = 6; se até ser feita a comparação com um valor de chave maior que o buscado (no caso, 7), a chave não for encontrada, então, com certeza o elemento não existe na lista**
- **A complexidade "de pior caso" permanece inalterada (é O(n)) em relação à busca simples, mas a complexidade média resultante é** menor

Função Busca_Lista_Ordenada

```
Ponteiro Busca_Lista_Ordenada(TipoLista *L, TipoChave x) {
   Apontador pos;
        pos = 0;
   if (L->Ultimo > MaxTam) {
        printf ("Erro na busca");
        return -2;
   } else {
         printf("Procuro x=%d\n", x);
         L->Item[L->Ultimo].chave = x;
         while (L->Item[pos].chave < x) { pos++; }
   }
   printf("Achei em: %d", pos);
   return pos; //o que fazer se a chave não for achada?
}
```

Busca Binária

- **Se a lista seqüencial é ordenada, existe ainda uma forma mais eficiente: a** busca binária
- Idéia: **percorrer a lista com se folheia uma lista telefônica, quando busca-se um nome**
  - **A 1a. posição verificada é ~ a metade da lista; se a chave procurada estiver acima/abaixo do valor do meio, metade dos valores é dispensada no próximo passo**
  - **No passo seguinte, o mesmo procedimento é adotado para a parte restante**
  - **Isto é repetido até o esgotamento completo da lista ou até encontrar o valor buscado**

Função Busca_Binaria

```
Ponteiro Busca_Binaria(TipoLista *L, TipoChave x) {
  Apontador inf, sup, meio;
  if (L->Ultimo > MaxTam) {
     printf("Erro na busca");
     return -1;
  } else {
     inf = L->Primeiro;
     sup = L->Ultimo-1;
     while (inf <= sup) {
              meio = (inf + sup) / 2; //divisão inteira
              if (L->Item[meio].Chave == x) inf = sup +1;
              else if (L->Item[meio].Chave < x) inf = meio + 1;
                    else sup = meio - 1;
     }
  }
  return meio; //alguma alteração aqui?
}
```

```
typedef int Apontador;
```

Analisando a Busca Binária

- **A complexidade da busca é reduzida: O(*log*n) passos são realizados no pior caso (lembre-se que a cada passo o número máximo de elementos cai pela metade)**
- **O número máximo de iterações é $1+\lfloor \log_2 n \rfloor$**
- **Tanto a inserção quanto a remoção usam a busca; por isso, deve-se pensar na eficiência da implementação:**
  - **Inserção: busca para evitar chaves repetidas**
  - **Remoção: encontrar a chave a ser removida**

ALGORITMOS DE BUSCA EM LISTAS LINEARES COM ALOCAÇÃO ENCADEADA

Busca em Listas Encadeadas

- **Similar ao caso de listas seqüenciais:**
  - **Cada nó da lista é formado por registros que contêm um campo *chave*; todas as chaves da lista** são distintas
- **Os registros da lista podem estar ordenados ou não de acordo com o valor desta chave:** listas ordenadas **e** não-ordenadas
- **Pensando-se na utilização do resultado da busca na inserção e remoção de itens da lista, o algoritmo de busca em lista encadeada é ligeiramente diferente do algoritmo visto para alocação seqüencial**

Busca em Listas Encadeadas (cont.)

- **O resultado da busca fornece dois ponteiros *pont* e *ant* que "apontam", respectivamente, para:**
  - **O elemento procurado (se ele existir) ou para NULL (caso contrário);**
  - **O último nó pesquisado antes do retorno da função**
- **Mas, por que o ponteiro *ant*?**
  - **Ele é de extrema importância para a remoção e inserção numa lista encadeada**
- **Após a realização da busca, os algoritmos de inserção e remoção tornam-se triviais!**

Procedimento Busca_Lista

```
void Busca_Lista(TipoLista *L, TipoChave x, Ponteiro *ant,
   Ponteiro *pont){
   Ponteiro pos;
   *ant = L->Primeiro; *pont = NULL; pos = *(ant)->Prox;
      while (pos != NULL) {
            if (pos->Item.Chave != x) {
                  *ant = pos;
                  pos = pos->Prox;
            } else {
                  *pont = pos;
                  pos = NULL;
            }
      }
}
```

Ponteiros de ponteiros

Novamente, estes dois testes podem ser evitados (**exercício**)

```
typedef struct celula *Ponteiro;

[Ponteiro *ant é igual a struct celula **ant]
```

Busca em Listas Ordenadas

- **Novamente, pode-se tirar proveito da ordenação das chaves para agilizar o processo de busca.**
- **Suponha uma lista encadeada com as chaves ordenadas em ordem crescente:**
  - **Se a chave de valor *x1* é buscada, o algoritmo deve pesquisar na lista até que uma chave *x2* de valor maior ou igual a *x1* seja encontrada**
    - Se *x1 == x2,* então o elemento está na lista, *pont* aponta para ele e *ant,* para seu antecessor
    - Se *x1 < x2,* então o elemento não está na lista, *pont* aponta para *NULL* e *ant,* para seu antecessor (exatamente o ponteiro necessário para sua inserção de maneira ordenada na lista!!!)

Procedimento Busca_Lista_Ordenada

```
void Busca_Lista_Ordenada(TipoLista *L, TipoChave x, Ponteiro
   *ant, Ponteiro *pont){
   Ponteiro pos;

   *ant = L->Primeiro; *pont = NULL; pos = *(ant)->Prox;
   while (pos != NULL) {
        if (pos->Item.Chave < x) {
                *(ant) = pos;
                pos = pos->Prox;
        } else {
                if (pos->Item.Chave == x) *pont = pos;
                pos = NULL;
        }
    }
}
```

Estes dois testes podem ser evitados

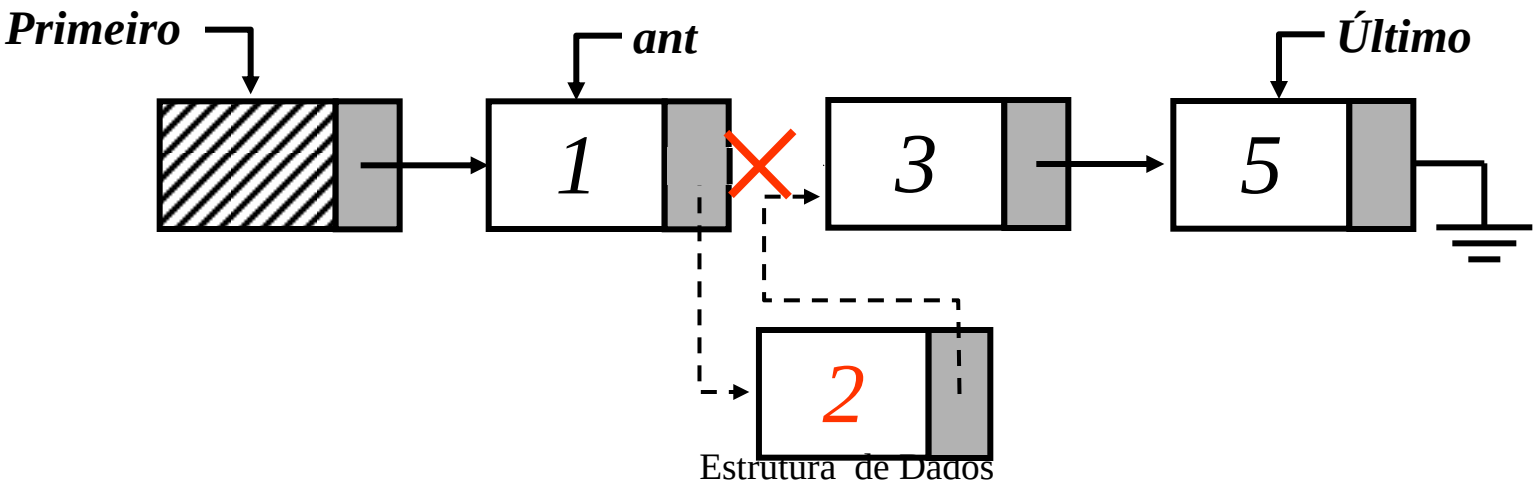
```
typedef struct celula *Ponteiro;

[Ponteiro *ant é igual a struct celula **ant]
```

Procedimento Busca_Lista_Ordenada com Nó-Sentinela

```
void Busca_Lista_Ordenada(TipoLista *L, TipoChave x, Ponteiro
   *ant, Ponteiro *pont){
   Ponteiro pos, sentinela;
   *ant= L->Primeiro; *pont= NULL; pos = *(ant)->Prox;
   sentinela = malloc(sizeof *sentinela);
   L->Ultimo->Prox = sentinela; sentinela->Prox = NULL;
   sentinela->Item.Chave = x;
   while (pos->Item.Chave < x) {
      *ant = pos;
      pos = pos->Prox;
   }
   if (pos->Item.Chave == x && *ant != L->Ultimo)
       *pont = pos
   else *pont = NULL;
   free(sentinela);
   L->Ultimo->Prox = NULL;
}
```

Analisando a Busca

- **A complexidade “de pior caso” permanece inalterada (é O(n)) em relação à busca em listas seqüenciais**
- **O algoritmo de busca binária não pode ser aplicado diretamente em listas encadeadas (por que?)**
- **Como dito anteriormente, o resultado da busca facilita a inserção e a remoção de itens na lista:**



Busca com Inserção em Lista Ordenada

```
int Insere_Lista_Ordenada (TipoItem x, TipoLista *L) {
     Ponteiro pNovo, ant, pont;
     Busca_Lista_Ordenada(L, x.Chave, &ant, &pont);
     if (pont == NULL) { //item não existe:  pode inserir
        pNovo = malloc(sizeof *pNovo);
        if (pNovo == NULL) {
          printf("Sem memoria para alocar");  return 0;
        } else {
        pNovo->Item = x; //Copia Item para a Lista
        pNovo->Prox = ant->Prox;
        ant->Prox = pNovo;
        //atualiza ponteiro para o último elemento
        if (L->Ultimo == ant) L->Ultimo=pNovo;
        L->Ultimo->Prox = NULL; //certifica que a lista termina em NULL
        return 1;
      } else return 0;
}
```

```
typedef struct celula *Ponteiro;
```

Analisando o algoritmo...

- **Note que na primeira implementação do algoritmo de remoção em listas, havia um parâmetro que era o ponteiro para o nó anterior aquela a ser removido:**

```
void Retira(Apontador p, TipoLista *Lista,
TipoItem *Item)
```

- **Este primeiro parâmetro é exatamente o resultado do procedimento de busca:**
  - **Se após a busca, *pont* == NULL, então o elemento não existe e não há remoção alguma a ser feita**
  - **Se *pont* aponta para o elemento a ser retirado, *ant* apontará para seu predecessor e pode ser passado como o primeiro argumento de `Retira_Lista`**

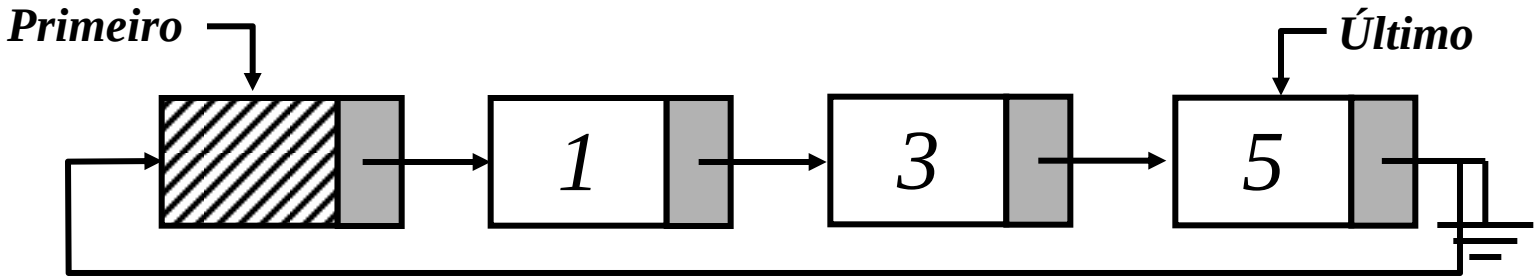
Analisando os algoritmos

- **Note que as interfaces das funções de inserção e remoção** não foram alteradas!
- **Mais ainda, não há necessidade de redefinir um procedimento de remoção**
- **Para a inserção, vale a pena apenas definir um novo procedimento para a inserção em lista ordenada**
  - **A inserção simples no fim da lista só utiliza a busca para evitar a duplicidade de chaves e não para definir o local correto de inserção**
- **Manter listas ordenadas pode ser bastante útil em várias situações: listas de alunos em ordem alfabética, lista de candidatos de um concurso pelo número de pontos, etc.**

LISTAS ENCADEADAS CIRCULARES

Listas Circulares

- **Alguns algoritmos das listas encadeadas podem ter seus desempenhos "melhorados" com uma simples modificação na estrutura da lista:**
  - **Ao invés do campo *Prox* do último nó da lista apontar para *NULL*, ele passa a apontar o nó-cabeça, transformando a lista em** circular**… Vantagens?**



Nova solução para a busca

- **A chave buscada é colocada no nó-cabeça da lista, de maneira que a busca sempre encontre a chave procurada**
- **Novamente, o ponteiro *ant* aponta para o último nó pesquisado antes do retorno, e *pont* aponta para o elemento cuja chave é igual à buscada, ou para *NULL* se a chave não existir na lista.**
- **O mesmo princípio pode ser aplicado para o caso de listas não-ordenadas, com pequenas modificações...**
- **Obviamente, as operações de criação, inserção e remoção das listas devem ser revistas...**

Melhoria na Busca

```
void Busca_Lista_Ordenada(TipoLista *L, TipoChave x,
   Ponteiro *ant, Ponteiro *pont){
   Ponteiro pos, sentinela;
    *ant= L->Primeiro; *pont= NULL; pos = *(ant)->Prox;
   sentinela = malloc(sizeof *sentinela);
   L->Ultimo->Prox = sentinela; sentinela->Prox = NULL;
   sentinela->Item.Chave = x;
    while (pos->Item.Chave < x) {
       *ant = pos;
       pos = pos->Prox;
    }
    if (pos->Item.Chave == x && *ant != L->Ultimo)
        *pont = pos
   else *pont = NULL;
   free(sentinela);
   L->Ultimo->Prox = NULL;
}
```

```
typedef struct celula *Ponteiro;

[Ponteiro *ant é igual a struct celula **ant]
```

Procedimento Busca_Lista_Circular Ordenada

```
void Busca_Lista_Ordenada(TipoLista *L, TipoChave x,
   Ponteiro *ant, Ponteiro *pont){
   Ponteiro pos;
    *ant= L->Primeiro; *pont= NULL; pos = *(ant)->Prox;
   L->Primeiro->Item.Chave = x;
   while (pos->Item.Chave < x) {
      *ant = pos;
      pos = pos->Prox;
    }
    if (pos->Item.Chave == x && pos != L->Primeiro)
        *pont = pos
   else *pont = NULL; //chave não existe
}
```

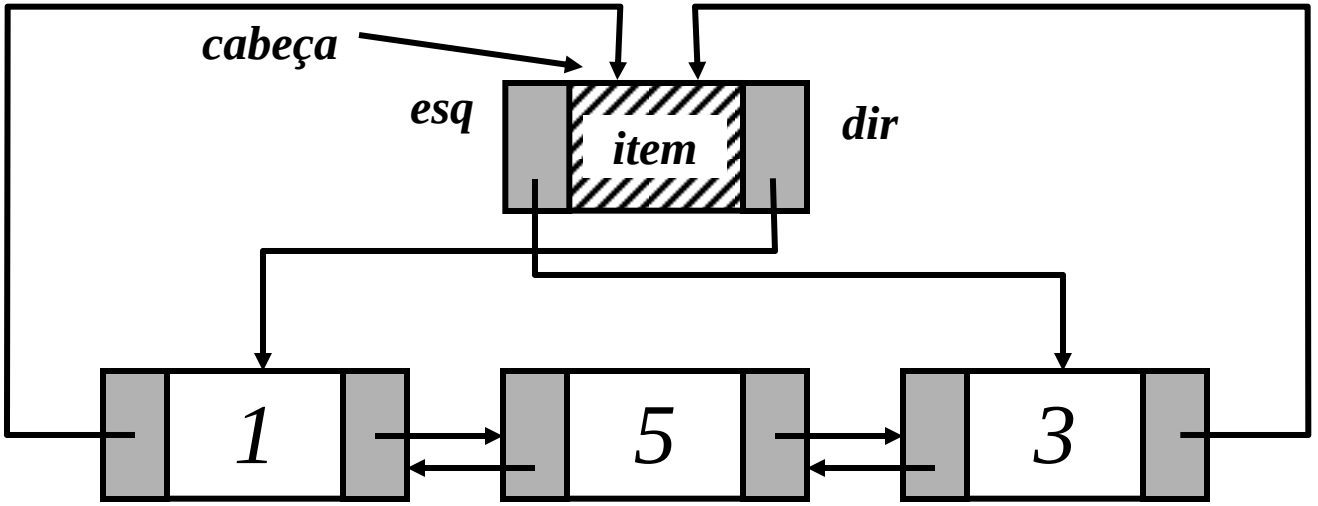
LISTAS DUPLAMENTE ENCADEADAS

Listas Duplamente Encadeadas

- **Listas Lineares Simplesmente encadeadas: movimentação somente na direção dos ponteiros, do início para o fim da lista**
- **Assim, a única forma de encontrar o elemento que precede *P* é recomeçar a percorrer a lista a partir de seu início**
  - **Por isso, os algoritmos de busca sempre retornam o ponteiro *ant* para facilitar a inserção e remoção**
- **Uma solução diferente é usar uma lista duplamente encadeada**

Listas Duplamente Encadeadas

- **Listas Lineares Duplamente Encadeadas:**
  - **Cada nó da lista é composto por no mínimo 3 campos: o Item, um ponteiro para a esquerda e outro para a direita.**
- **Exemplo de lista circular duplamente encadeada**

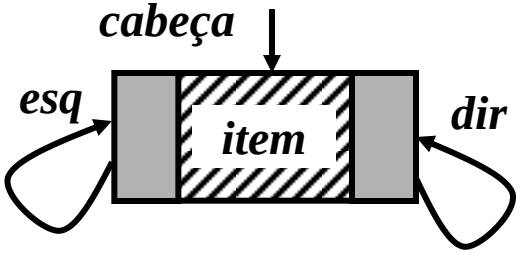


Listas Duplamente Encadeadas

- **Se *P* aponta para uma célula ou nó numa lista duplamente encadeada:**

  $P = \text{Dir}(\text{Esq}(P)) = \text{Esq}(\text{Dir}(P))$

- **Isto quer dizer que podemos caminhar em listas deste tipo para frente e para trás, usando os ponteiros apropriados**
- **Lista duplamente encadeada circular vazia?**



Busca em Lista Duplamente Encadeada Circular Ordenada

- **A busca far-se-á colocando-se a chave de busca no nó-cabeça.**
- **Percorre-se a lista até que o elemento apontado seja maior ou igual à chave de busca.**
- **A função de busca retorna o ponteiro para o item buscado, ou, se ele não existir, o ponteiro para a posição na lista imediatamente à direita.**
- **A inserção em ordem, pois, é feita na posição imediatamente à esquerda do elemento apontado pela busca.**
  - **Se a lista estiver vazia, ou a inserção deve ser feita ao final, a função retorna Lista->Primeiro.**
- **É preciso testar o retorno da função busca a fim de verificar se o ponteiro aponta para o item buscado, ou não, pois a função busca só retorna o ponteiro.**

Estrutura da Lista Duplamente Encadeada Circular

```
typedef int TipoChave;

typedef struct {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;

typedef struct Celula_str
   *Ponteiro;
typedef struct Celula_str
   {
  TipoItem Item;
  Ponteiro esq;
  Ponteiro dir;
} Celula;
```

```
typedef struct {
  Ponteiro Primeiro,
   Ultimo;
  int Tamanho;
} TipoListaDupEnc;
```

Busca em Lista Duplamente Encadeada Ordenada

```
Ponteiro BuscaLDE_Ordenada(TipoChave x, TipoListaDupEnc
   *L) {
   Ponteiro pos;
   if (Vazia(L)) return NULL; //NULL para lista vazia
   else { /* existe pelo menos 1 nó */
        L->Primeiro->Item.Chave = x;
        if (x <= L->Ultimo->Item.Chave) { //teste genérico
               pos = L->Primeiro->dir;
               while (pos->Item.Chave < x) {
                      pos = pos->dir; }
               return pos;
        } else //item não existe
        return L->Primeiro;
   }
}
```

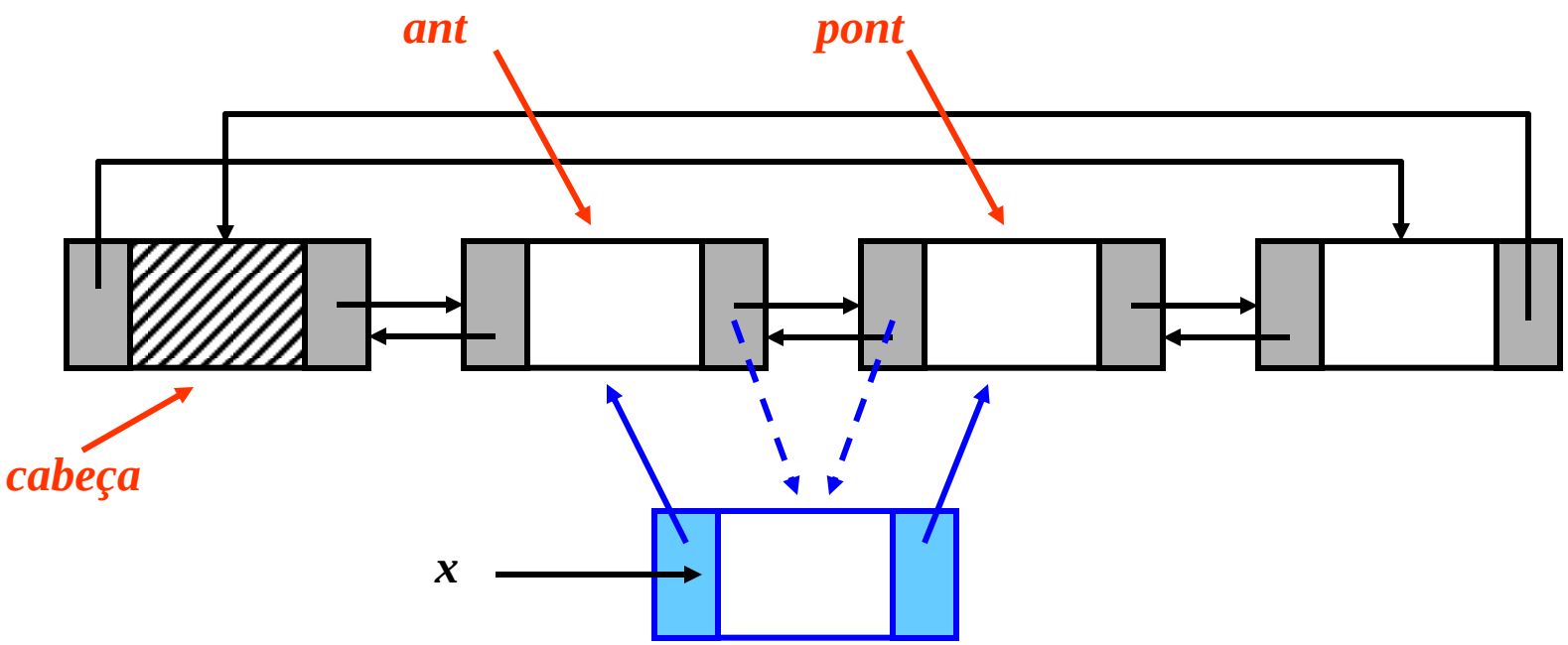
```
typedef struct celula *Ponteiro;
```

Busca em Lista Duplamente Encadeada Ordenada - Alternativa

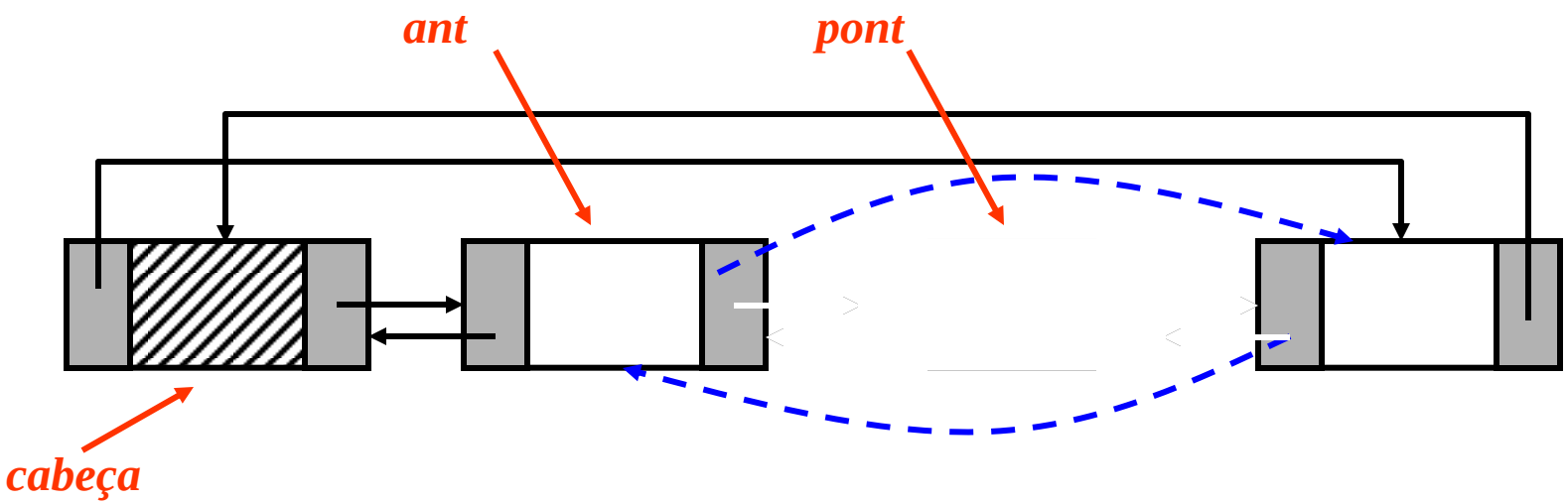
```
void BuscaLDEOrdenadaAlt(TipoListaDupEnc *lista, TipoChave
   x) {
  Ponteiro pos;
  pos=lista->Primeiro->dir;
  lista->Primeiro->Item.Chave = x; //coloca item buscado
   no HEAD NODE
  while (pos->Item.Chave < x)
  {
    pos=pos->dir; //se item não existir, busca termina no
   HEAD NODE
  }
}
```

```
typedef struct celula *Ponteiro;
```

Inserção em Lista Duplamente Encadeada



Remoção em Lista Duplamente Encadeada



Algoritmos de Remoção em Lista Duplamente Encadeada

```
int RemoverLDE(TipoItem *x, TipoListaDupEnc *lista,
   Ponteiro pos) {
   if (Vazia(lista) || pos==NULL) return 0;  //lista
   vazia, nada removido
   *x=pos->Item;  //devolve valores que serão
   removidos
   pos->dir->esq=pos->esq;
   pos->esq->dir=pos->dir;
   if (pos->dir == lista->Primeiro)
       lista->Ultimo=pos->esq;
   free(pos); //libera espaço ocupado por nó em pos
   return 1; //remoção bem sucedida
}
```

```
typedef struct celula *Ponteiro;
```

Algoritmos de Inserção em Lista Duplamente Encadeada

```
void InserirLDEOrdem(TipoItem x, TipoListaDupEnc *lista) {
  Ponteiro pnew;
  Ponteiro pos; //pos é o elemento na lista ou posição posterior
  if (Vazia(lista)==0) pos=BuscaLDE_Ordenada(x.Chave, lista);
  else pos=lista->Primeiro; //lista está vazia; aponta para nó cabeça
  if (pos->Item.Chave != x.Chave || pos==lista->Primeiro) { //item não
   existe:  pode inserir; pos aponta para nó posterior à posição de
   inserção
    pnew=malloc(sizeof *pnew); //cria espaço e ponteiro para ele
    pnew->Item=x;
    pnew->esq=pos->esq;
    pnew->dir=pos->esq->dir;
    pnew->esq->dir=pnew;
    pnew->dir->esq=pnew;
    if (pos==lista->Primeiro) lista->Ultimo=pnew; //atualiza ponteiro para
   último elemento
    lista->Ultimo->dir=lista->Primeiro; //certifica que a lista é circular
    lista->Primeiro->esq=lista->Ultimo;
  } else
  puts("Elemento com a Chave ja existe.  Nada foi inserido.");
}
```

```
typedef struct celula *Ponteiro;
```