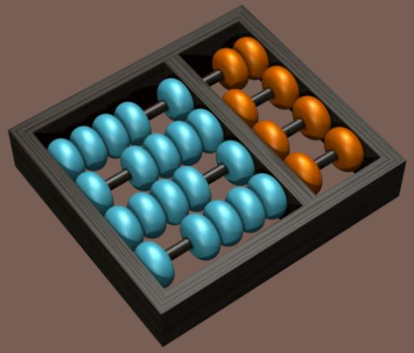


UNICAMP

LISTA LINEAR

Prof.: Julio Cesar dos Reis

Roteiro

- Definição e Estrutura da lista linear
- Tipo abstrato de dados (TAD)
- Operações do TAD lista linear
  - **Arranjos desordenados**
  - **Arranjos ordenados**
- Arranjos dinâmicos

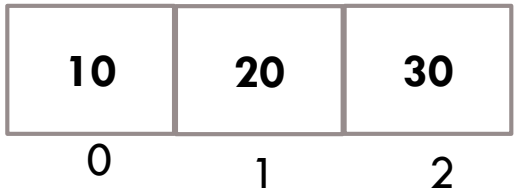
Definição

- Cada elemento da estrutura de dados é precedido por um elemento e sucedido por outro
  - Exceção do primeiro e último

- Os elementos estão em uma ordem
  - Ordenados por uma chave
  - Ordem de inserção

Uso de arranjos

- Lista indexada de itens
  - Indexado por índice



- Um bloco sequencial de memória

- Ordem lógica dos elementos é a mesma ordem física (em memória principal) dos elementos

Estrutura da Lista Linear

elemento

```
typedef struct{
      int chave;
    // outros campos...
} Elemento;
```

lista

```
typedef struct{
     Elemento Arranjo[TAM];
     int qtdElem;
} Lista;
```

#define TAM 100
(alocação estática)

Operações na Lista linear

- Inicialização / Reinicialização
- Obtenção da quantidade de elementos
- Impressão

Operações na Lista linear

- Inicialização / Reinicialização
- Obtenção da quantidade de elementos
- Impressão
- **Inserção**
  - Posição indicada pelo usuário
  - No final do arranjo
  - Ordenado

Operações na Lista linear

- Inicialização / Reinicialização
- Obtenção da quantidade de elementos
- Impressão
- **Inserção**
  - Posição indicada pelo usuário
  - No final do arranjo
  - Ordenado
- **Remoção**
- **Busca**
- Destruição

Tipo abstrato de dados (TAD)

- Especificação de um conjunto de operações permitidas associadas à uma estrutura de dados
  - Separa o uso de uma estrutura de dados dos detalhes de sua implementação

Tipo abstrato de dados (TAD)

- Especificação de um conjunto de operações permitidas associadas à uma estrutura de dados
  - Separa o uso de uma estrutura de dados dos detalhes de sua implementação

- **Interface** (arquivo xxx.h)
  - Conjunto de operações de uma TAD
  - Consiste dos nomes e demais convenções usadas para executar cada operação
    - Assinatura das funções
  - Ex.: void iniciar_lista(Lista *p_l);

Tipo abstrato de dados (TAD)

- **Implementação** (arquivo xxx.c)
  - Algoritmos que realizam as operações
  - Local onde dados são acessados de fato

Tipo abstrato de dados (TAD)

- **Implementação** (arquivo xxx.c)
  - Algoritmos que realizam as operações
  - Local onde dados são acessados de fato

- **Cliente** (#include “xxx.h”)
  - Código externo que utiliza um TAD
  - Invoca as operações definidas no TAD
  - Não conhece os detalhes de implementação

TAD em C

- Declarado como um registro

- Conjunto de protótipos de funções que recebem/usam o registro

Lista linear como um TAD


```
void iniciar_lista(Lista *p_l);
void reiniciar_lista(Lista *p_l);
int consultar_tamanho(Lista *p_l);
void imprimir_lista(Lista *p_l);
```

Lista linear como um TAD


```
void iniciar_lista(Lista *p_l);
void reiniciar_lista(Lista *p_l);
int consultar_tamanho(Lista *p_l);
void imprimir_lista(Lista *p_l);

bool inserirElemLista(Lista *p_l, Elemento ele, int pos_usu);
bool inserirElemFinalLista(Lista *p_l, Elemento ele);
```

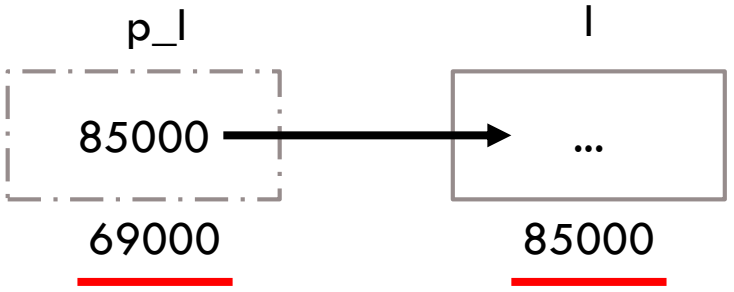
Lista linear como um TAD


```
void iniciar_lista(Lista *p_l);
void reiniciar_lista(Lista *p_l);
int consultar_tamanho(Lista *p_l);
void imprimir_lista(Lista *p_l);

bool inserirElemLista(Lista *p_l, Elemento ele, int pos_usu);
bool inserirElemFinalLista(Lista *p_l, Elemento ele);

bool removerElemListaPeloIndice(Lista *p_l, int pos_usu);
bool removerElemListaPelaChave(Lista *p_l, int chave);

int buscaSequencial(Lista *p_l, int chave_busca);

...
```

Inicialização

- Colocar 0 no campo de quantidade de elementos

```
void iniciar_lista(Lista *p_l){
      p_l->qtdElem=0;
}
```



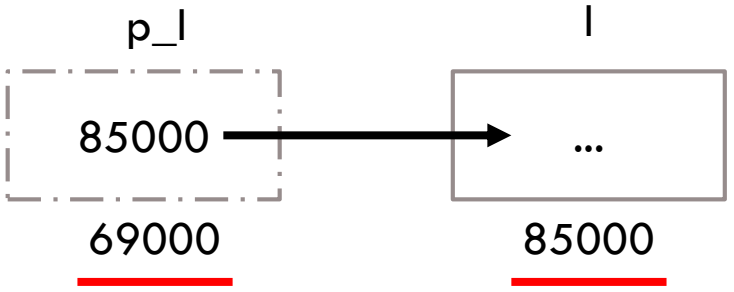
Código cliente

```
Lista l;
iniciar_lista(&l);
```

Inicialização

- Colocar 0 no campo de quantidade de elementos

```
void iniciar_lista(Lista *p_l){
      p_l->qtdElem=0;
}
```



Código cliente

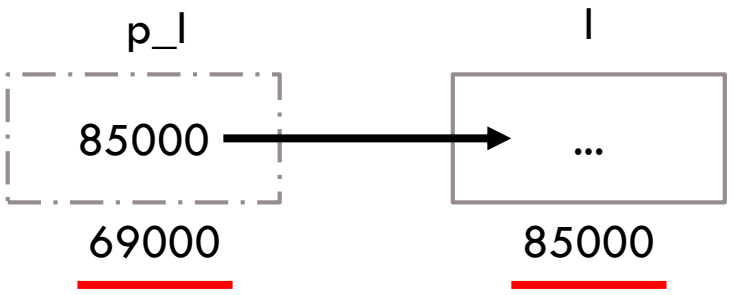
```
Lista l;
iniciar_lista(&l);
```

E se o parâmetro não for um ponteiro?

Inicialização / Reinicialização

- Colocar 0 no campo de quantidade de elementos

```
void iniciar_lista(Lista *p_l){
      p_l->qtdElem=0;
}
```



Código cliente

```
Lista l;
iniciar_lista(&l);
```

E se o parâmetro não for um ponteiro?

Como seria uma operação para reinicializar a lista?

Quantidade de Elementos

- Retorna o valor do campo *qtdElem*

```
int consultar_tamanho(Lista *p_l){
      return p_l->qtdElem;
}
```

Impressão

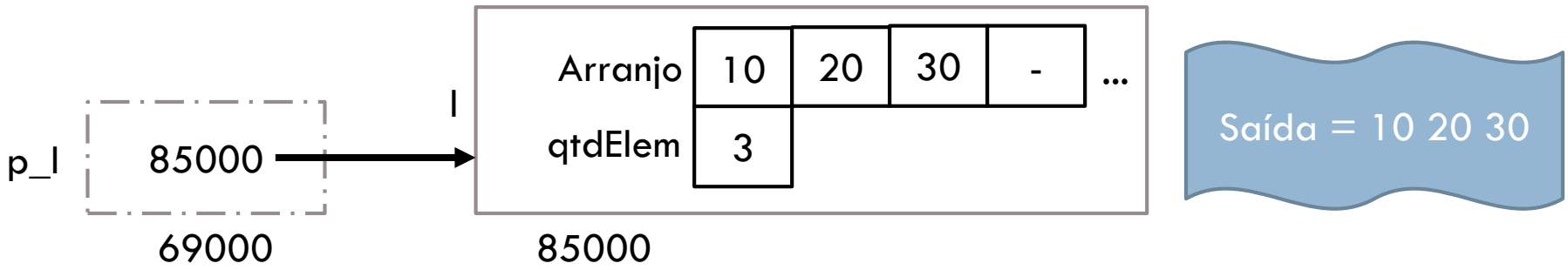
- Itera sobre os elementos válidos
- Apresenta suas informações (e.g., número da chave)

```
void imprimir_lista(Lista *p_l){
        int pos;
        for (pos=0; pos<p_l->qtdElem; pos++){
                printf("%d ", p_l->Arranjo[pos].chave);
        }
}
```

Impressão

- Itera sobre os elementos válidos
- Apresenta suas informações (e.g., número da chave)

```
void imprimir_lista(Lista *p_l){
        int pos;
        for (pos=0; pos<p_l->qtdElem; pos++){
                printf("%d ", p_l->Arranjo[pos].chave);
        }
}
```



Inserção – **Arranjo desordenado**

- **Posição no arranjo indicada pelo usuário**

Inserção – **Arranjo desordenado**

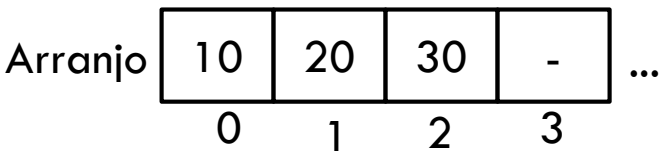
- **Posição no arranjo indicada pelo usuário**
- Procedimento
  1. Verificações
     1. Posição indicada deve ser válida
     2. Arranjo não pode estar cheio

Inserção – Verificações


```
bool inserirElemLista(Lista *p_l, Elemento ele, int pos_usu){
    if ((p_l->qtdElem == TAM) || (pos_usu < 0) || (pos_usu > p_l->qtdElem))
        return false;

    (...)
}
```

Inserção – **Arranjo desordenado**

- **Posição no arranjo indicada pelo usuário**
- Procedimento
  1. Verificações
     1. Posição indicada deve ser válida
     2. Arranjo não pode estar cheio
  2. Encontre a posição correta
  3. Desloque os elementos para a direita

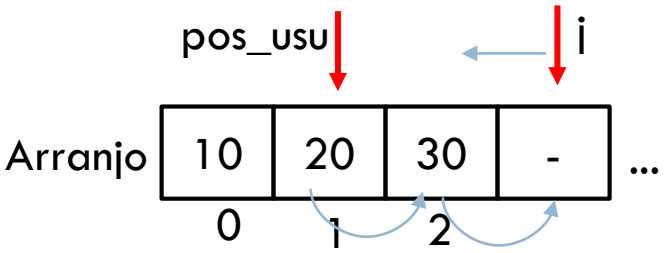


Parâmetros de entrada

ele.chave=55 TAM=100
pos_usu=1

Inserção – Encontra posição e desloca elementos


```
bool inserirElemLista(Lista *p_l, Elemento ele, int pos_usu){
    if ((p_l->qtdElem == TAM) || (pos_usu < 0) || (pos_usu > p_l->qtdElem))
        return false;

    for (int j = p_l->qtdElem; j > pos_usu; j--) //encontra posição
        p_l->Arranjo[j] = p_l->Arranjo[j-1];   //desloca elementos p/ direita

    (...)
}
```



Parâmetros de entrada

ele.chave=55 TAM=100
pos_usu=1

Inserção – Encontra posição e desloca elementos


```
bool inserirElemLista(Lista *p_l, Elemento ele, int pos_usu){
    if ((p_l->qtdElem == TAM) || (pos_usu < 0) || (pos_usu > p_l->qtdElem))
        return false;

    for (int j = p_l->qtdElem; j > pos_usu; j--) //encontra posição
        p_l->Arranjo[j] = p_l->Arranjo[j-1];   //desloca elementos p/ direita

    (...)
}
```

Inserção – **Arranjo desordenado**

- **Posição no arranjo indicada pelo usuário**
- Procedimento
  1. Verificações
     1. Posição indicada deve ser válida
     2. Arranjo não pode estar cheio
  2. Encontre a posição correta
  3. Desloque os elementos para a direita
  4. Insira o novo elemento na posição correta
  5. Some 1 no campo *qtdElem*

Inserção – Adiciona novo elemento


```
bool inserirElemLista(Lista *p_l, Elemento ele, int pos_usu){
    if ((p_l->qtdElem == TAM) || (pos_usu < 0) || (pos_usu > p_l->qtdElem))
        return false;

    for (int j = p_l->qtdElem; j > pos_usu; j--) //encontra posição
        p_l->Arranjo[j] = p_l->Arranjo[j-1];   //desloca elementos

    p_l->Arranjo[j] = ele;
    p_l->qtdElem++;
    return true;
}
```

Inserção – **Arranjo desordenado**

- **No final do arranjo**
- Procedimento
  1. Verificações
     1. Arranjo não pode estar cheio

Inserção – **Arranjo desordenado**

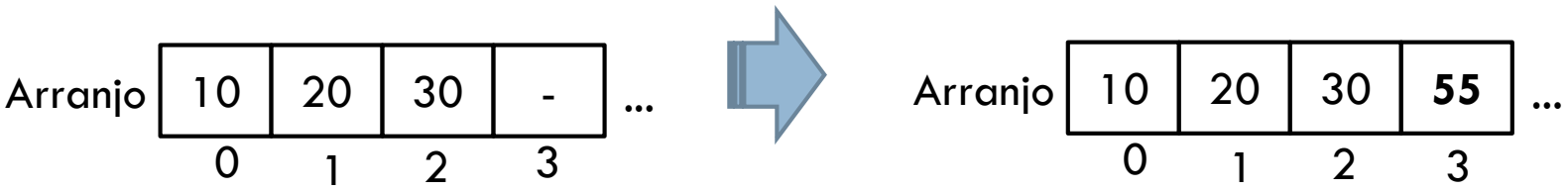

```
bool inserirElemFinalLista(Lista *p_l, Elemento ele){
    if ((p_l->qtdElem == TAM))
        return false;



}
```

Inserção – **Arranjo desordenado**

- **No final do arranjo**
- Procedimento
  1. Verificações
     1. Arranjo não pode estar cheio
  2. Insira o novo elemento na próxima posição válida
  3. Some 1 no campo qtdElem

Inserção – Arranjo **desordenado**


```
bool inserirElemFinalLista(Lista *p_l, Elemento ele){
     if ((p_l->qtdElem == TAM))
          return false;
    p_l->Arranjo[p_l->qtdElem] = ele;
    p_l->qtdElem++;
    return true;
}
```



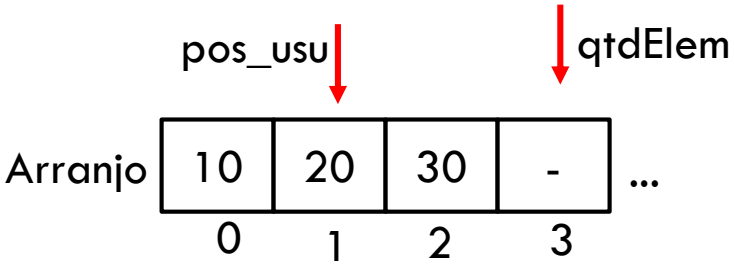
Remoção – **Arranjo desordenado**

- **Posição no arranjo indicada pelo usuário**
- Procedimento
  1. Verifica se a posição indicada é válida

Remoção – Arranjo desordenado


```
bool removerElemListaPeloIndice(Lista *p_l, int pos_usu){
    if ((pos_usu < 0) || (pos_usu > p_l->qtdElem))
        return false;

    (...)

}
```

Verificação



Parâmetros de entrada

qtdElem=3 TAM=100
pos_usu=1

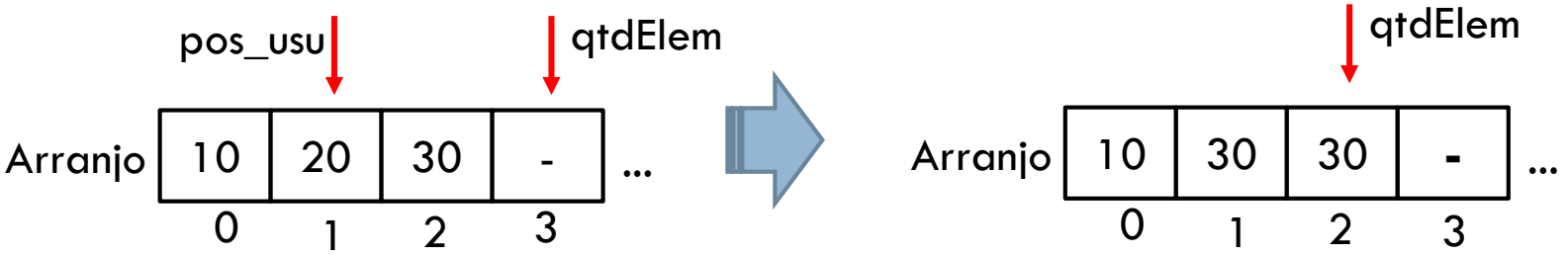
Remoção – **Arranjo desordenado**

- **Posição no arranjo indicada pelo usuário**
- Procedimento
  I. Verifica se a posição indicada é válida
  II. Troca o elemento na posição indicada com o último elemento
  III. Diminui o valor do campo *qtdElem*

Remoção – Arranjo desordenado


```
bool removerElemLista(Lista *p_l, int pos_usu){
    if ((pos_usu < 0) || (pos_usu > p_l->qtdElem))
        return false;

    p_l->Arranjo[pos_usu] = p_l->Arranjo[p_l->qtdElem - 1];
    p_l->qtdElem--;
    return true;
}
```

Troca



Remoção – **Arranjo desordenado**

- **Valor do elemento indicado pelo usuário**
- Procedimento
  1. Encontrar o elemento com a chave na lista

Remoção – **Arranjo desordenado**


```
bool removerElemListaPelaChave(Lista *p_l, int chave){
    int pos = buscaSequencial(p_l, chave);
    if(pos == -1) return false;




}
```

Encontra elemento

Remoção – **Arranjo desordenado**

- **Valor do elemento indicado pelo usuário**
- Procedimento
  I. Encontra o elemento com a chave na lista
  II. Troca o elemento encontrado com o último elemento
  III. Diminui o valor do campo *qtdElem*

Remoção – **Arranjo desordenado**


```
bool removerElemListaPelaChave(Lista *p_l, int chave){
      int pos = buscaSequencial(p_l, chave);
      if(pos == -1) return false;

   p_l->Arranjo[pos] = p_l->Arranjo[p_l->qtdElem - 1];
   p_l->qtdElem--;
   return true;
}
```

Coloca último na posição encontrada

Busca – **Arranjo desordenado**

- Busca Sequencial
  - Recebe um valor de chave do usuário
  - Investiga cada elemento do arranjo
  - Se encontrar o elemento
    - Retorna a posição do índice do arranjo onde ele se encontra
  - Se o elemento não existe na lista
    - Retorna -1

Busca – **Arranjo desordenado**


```
int buscaSequencial(Lista *p_l, int chave_busca){
     int i;
     for (i=0; i<p_l->qtdElem; i++)
          if (p_l->Arranjo[i].chave==chave_busca)
            return i;
    return -1;
}
```

Busca – Arranjo desordenado


```
int buscaSequencial(Lista *p_l, int chave_busca){
     int i;
     for (i=0; i<p_l->qtdElem; i++)
          if (p_l->Arranjo[i].chave==chave_busca)
            return i;
    return -1;
}
```

- Busca binária é mais eficiente?
- Podemos fazer busca binária? Por quê?

Análise sobre operações de buscas

- Se operação de busca for muito frequente, efetuar busca binária pode ser vantajoso

- Busca binária exige que o arranjo esteja ordenado

Análise sobre operações de buscas

- Se operação de busca for muito frequente, efetuar busca binária pode ser vantajoso
- Busca binária exige que o arranjo esteja ordenado
- Duas soluções
  I. Ordenar o arranjo aplicando um algoritmo de ordenação
     - Viável se não tiver que aplicar sempre pois tem um custo
  II. Manter o arranjo ordenado nas operações de inserção e remoção

Inserção – **Arranjo ordenado**

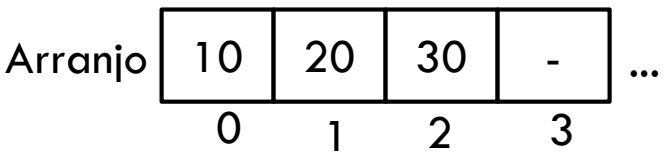
- Verifique se a lista não está cheia

Inserção – **Arranjo ordenado**


```
bool inserirElemListaOrdenada(Lista *p_l, Elemento ele){
    if ((p_l->qtdElem >= TAM))
        return false;
```

Inserção – **Arranjo ordenado**

- Verifique se a lista não está cheia
- Encontre a posição correta do elemento a ser inserido
- Desloque o elementos maiores para a direita
  - Cria espaço no arranjo



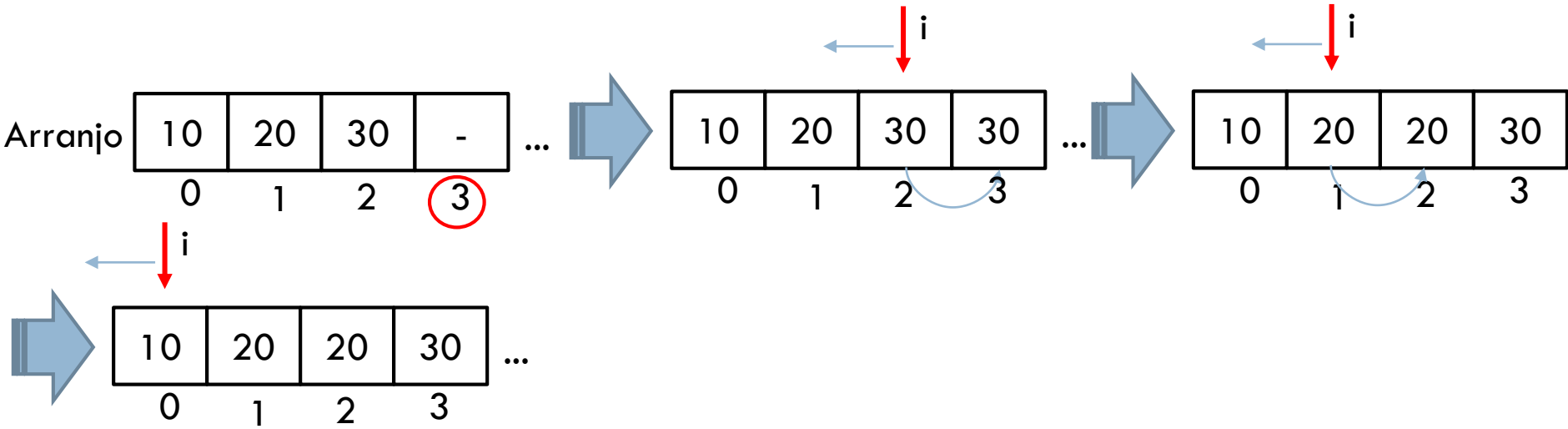
Parâmetros de entrada

qtdElem=3 TAM=100
nova chave=15

Inserção – Arranjo ordenado


```
bool inserirElemListaOrdenada(Lista *p_l, Elemento ele){
    if ((p_l->qtdElem >= TAM))
        return false;
    int i;
    for (int i = p_l->qtdElem-1; i >=0 && p_l->Arranjo[i] > ele.chave; i--)
        p_l->Arranjo[i+1] = p_l->Arranjo[i];   //desloca elementos p/ direita

    (...)
```

qtdElem=3
ele.chave=15

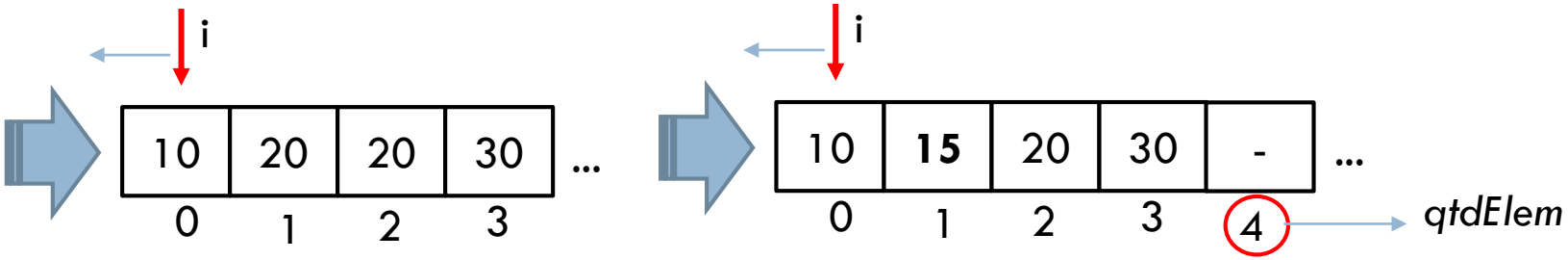


Inserção – **Arranjo ordenado**

- Verifique se a lista não está cheia
- Encontre a posição correta do elemento a ser inserido
- Desloque o elementos maiores para a direita
- Insira o novo elemento na posição correta

Inserção – **Arranjo ordenado**


```
bool inserirElemListaOrdenada(Lista *p_l, Elemento ele){
     if ((p_l->qtdElem >= TAM))
          return false;
    int i;
    for (int i = p_l->qtdElem-1; i >=0 && p_l->Arranjo[i] > ele.chave; i--)
          p_l->Arranjo[i+1] = p_l->Arranjo[i];   //desloca elementos p/ direita
    p_l->Arranjo[i+1] = ele;
    p_l->qtdElem++;
    return true;
}
```



Remoção – **Arranjo ordenado**

- **Valor do elemento indicado pelo usuário**
- Procedimento
  1. Encontre o elemento com a chave na lista
     - Pode-se usar busca binária

Remoção – **Arranjo ordenado**


```
bool removerElemListaPelaChave(Lista *p_l, int chave){
    int pos = buscaBinaria(p_l, chave);
    if(pos == -1) return false;

    (...)
}
```

Encontra elemento

Remoção – **Arranjo ordenado**

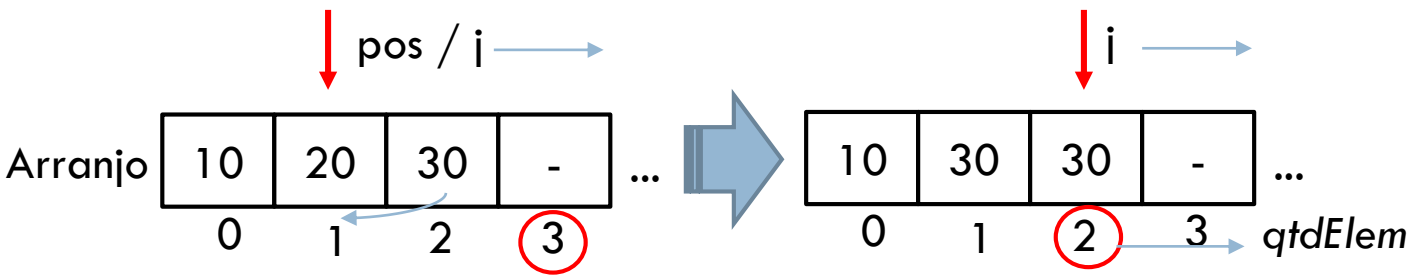
- **Valor do elemento indicado pelo usuário**
- Procedimento
  1. Encontra o elemento com a chave na lista
     - Pode-se usar busca binária
  2. Ao encontrar a posição, desloque os elementos para a esquerda (preenche espaço)
  3. Diminui o valor do campo *qtdElem*

Remoção – Arranjo ordenado


```
bool removerElemListaPelaChave(Lista *p_l, int chave){
      int pos = buscaBinaria(l, chave);
      if(pos == -1) return false;

    int j;
    for(j=pos; j < p_l->qtdElem-1; j++)
          p_l->Arranjo[j] = p_l->Arranjo[j+1];

    p_l->qtdElem--;
    return true;
}
```

qtdElem=3
chave=20

desloca elementos para a esquerda



Busca – **Arranjo ordenado**


```
int buscaBinaria(Lista *p_l, int chave_busca){
    int esq, dir, meio;
    esq = 0;        dir = p_l->qtdElem-1;
    while(esq <= dir) {
        meio = ((esq + dir) / 2);
        if(p_l->Arranjo[meio].chave == chave_busca)
            return meio; //encontrei elemento
        else {
            if(p_l->Arranjo[meio].chave < chave_busca)
                esq = meio + 1; //parte superior do arranjo
            else
                dir = meio - 1; //parte inferior do arranjo
        }
    }
    return -1;}
```

Análise sobre o tipo de arranjo

- Lista com arranjos desordenados
  - Inserção e remoção mais eficiente, mas exige busca sequencial

Análise sobre o tipo de arranjo

- Lista com arranjos desordenados
  - Inserção e remoção mais eficiente, mas exige busca sequencial
- Lista com arranjos ordenados
  - Busca mais eficiente, mas prejudicial a inserção e remoção

Análise sobre o tipo de arranjo

- Lista com arranjos desordenados
  - Inserção e remoção mais eficiente, mas exige busca sequencial
- Lista com arranjos ordenados
  - Busca mais eficiente, mas prejudicial a inserção e remoção
- Muitas inserções/remoções e poucas buscas
  - Usar **arranjos desordenados**

Análise sobre o tipo de arranjo

- Lista com arranjos desordenados
  - Inserção e remoção mais eficiente, mas exige busca sequencial
- Lista com arranjos ordenados
  - Busca mais eficiente, mas prejudicial a inserção e remoção
- Muitas inserções/remoções e poucas buscas
  - Usar **arranjos desordenados**
- Muitas buscas e poucas inserções/remoções
  - Usar **arranjos ordenados**

Arranjos dinâmicos

- Estudamos arranjos com tamanhos fixos (e alocação estática)
  - Inicialização a priori de um espaço delimitado

Arranjos dinâmicos

- Estudamos arranjos com tamanhos fixos (e alocação estática)
  - Inicialização a priori de um espaço delimitado

- Dificuldades
  - Saber o tamanho adequado do arranjo de antemão
  - Desperdício de memória pela alocação de grandes arranjos com poucos elementos usados

Arranjos dinâmicos

- Alocação dinâmica de arranjos
  - Aumentam ou diminuem de tamanho de acordo com a necessidade de dados a serem manipulados

Modificações na Estrutura


```
typedef struct{
      int chave;
    // outros campos...
} Elemento;
```

lista

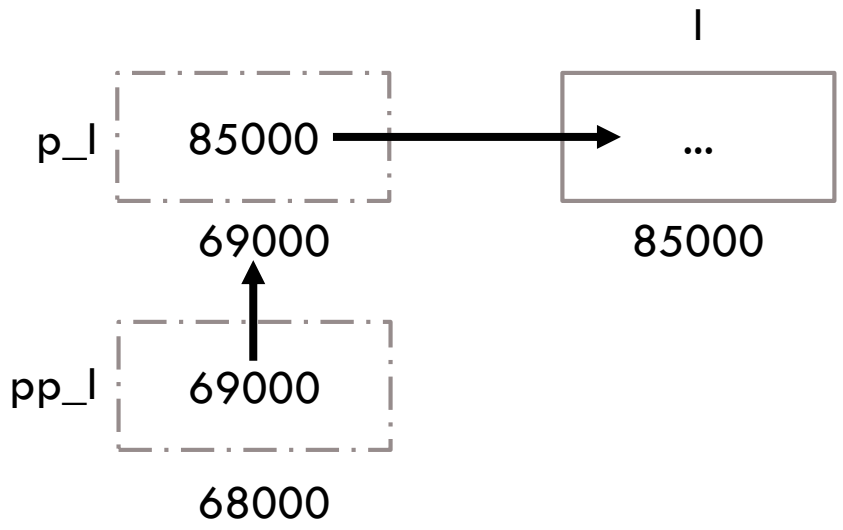
```
typedef struct {
     Elemento *elemts; // ponteiro para arranjo de elementos
     int qtdElem; // quantidade de elementos presentes na lista
     int alocado; // tamanho alocado da lista
} Lista;
```

Inicialização de arranjos dinâmicos


```
void iniciar_lista(Lista **pp_l, int tam){
      (*pp_l)->elemts = malloc(tam * sizeof(Elemento));
      (*pp_l)->qtdElem=0;
      (*pp_l)->alocado = tam;
}
```

Código cliente

```
Lista *p_l, l;
p_l=&l;
iniciar_lista(&p_l);
```



Finalização de arranjos dinâmicos


```
void finalizar_lista(Lista **pp_l){
      free((*pp_l)->elemts); //desaloca memória dos elementos
      free(*pp_l); //desaloca lista
      (*pp_l)=null;
}
```

Código cliente

```
Lista *p_l, l;
p_l=&l;
finalizar_lista(&p_l);
```

Inserção em arranjos dinâmicos

- Verifique se o arranjo alcançou seu tamanho limite
  - Se alcançou, aloca dinamicamente um novo arranjo com o **dobro** do tamanho do atual

Inserção em arranjos dinâmicos


```
bool inserirElemLista(Lista *p_l, Elemento ele){
    if ((p_l->qtdElem == p_l->alocado)){ //alcançou o tamanho limite
        //apontador temporário guarda referência ao arranjo
        Elemento *e_temp = p_l->elemts;
        //dobra o tamanho do campo alocado
        p_l->alocado=p_l->alocado*2;
        //aloca um arranjo com o novo tamanho
        p_l->elemts = malloc(p_l->alocado * sizeof(Elemento));









}
```

Inserção em arranjos dinâmicos

- Verifique se o arranjo alcançou seu tamanho limite
  - Se alcançou, aloca dinamicamente um novo arranjo com o **dobro** do tamanho do atual
- Transporta elementos para o novo arranjo

Inserção em arranjos dinâmicos


```
bool inserirElemLista(Lista *p_l, Elemento ele){
    if ((p_l->qtdElem == p_l->alocado)){ //alcançou o tamanho limite
        //apontador temporário guarda referência ao arranjo
        Elemento *e_temp = p_l->elemts;
        //dobra o tamanho do campo alocado
        p_l->alocado=p_l->alocado*2;
        //aloca um arranjo com o novo tamanho
        p_l->elemts = malloc(p_l->alocado * sizeof(Elemento));
        //transporta elementos válidos de e_temp para elemts
        for (int pos = 0; pos < p_l->qtdElem; pos++)
            p_l->elemts[pos] = e_temp[pos];
        free(e_temp); //desaloca memória do e_temp



}
```

Inserção em arranjos dinâmicos

- Verifique se o arranjo alcançou seu tamanho limite
  - Se alcançou, aloca dinamicamente um novo arranjo com o **dobro** do tamanho do atual
- Transporta elementos para o novo arranjo

- Insira o novo elemento
- Incremente o campo que contabiliza a quantidade de elementos

Inserção em arranjos dinâmicos


```
bool inserirElemLista(Lista *p_l, Elemento ele){
    if ((p_l->qtdElem == p_l->alocado)){ //alcançou o tamanho limite
        //apontador temporário guarda referência ao arranjo
        Elemento *e_temp = p_l->elemts;
        //dobra o tamanho do campo alocado
        p_l->alocado=p_l->alocado*2;
        //aloca um arranjo com o novo tamanho
        p_l->elemts = malloc(p_l->alocado * sizeof(Elemento));
        //transporta elementos válidos de e_temp para elemts
        for (int pos = 0; pos < p_l->qtdElem; pos++)
            p_l->elemts[pos] = e_temp[pos];
        free(e_temp); //desaloca memória do e_temp
    }
    p_l->elemts[p_l->qtdElem] = ele;     p_l->qtdElem++;
    return true;
}
```

Remoção em arranjos dinâmicos

- Ao efetuar operações de remoção, diminua o tamanho do arranjo quando ele ficar **pouco cheio**
  - Evita desperdício de memória

- **Pouco cheio** pode ser um fator (¼ ou ½) do tamanho alocado atual para o arranjo

Síntese

- Modelagem e operações em listas com arranjos
  - Lista linear como um TAD

Síntese

- Modelagem e operações em listas com arranjos
  - Lista linear como um TAD
- Implementação das operações
  - Uso de arranjos ordenados e desordenados

Síntese

- Modelagem e operações em listas com arranjos
  - Lista linear como um TAD
- Implementação das operações
  - Uso de arranjos ordenados e desordenados
- Alocação dinâmica de memória (arranjos dinâmicos)
  - Permite que os arranjos aumentem e diminuam de tamanho conforme a necessidade