

Listas Estáticas

Estruturas de Dados I

Prof. Tiago A. Almeida

Departamento de Computação
Universidade Federal de São Carlos (UFSCar)

Sumário

- 1 Conceitos Básicos
- 2 Implementação Estática de Listas
- 3 Listas Estáticas Ordenadas
- 4 Busca em Listas Estáticas
- 5 Conclusão

Sumário

- 1 Conceitos Básicos
- 2 Implementação Estática de Listas
- 3 Listas Estáticas Ordenadas
- 4 Busca em Listas Estáticas
- 5 Conclusão

Listas

Definição

- Listas são estruturas flexíveis que admitem as operações de **inserção, remoção e recuperação** de itens
- É uma sequencia de zero ou mais itens x_1, x_2, \dots, x_n na qual x_i é de um determinado tipo e n representa o tamanho da lista
- Sua principal propriedade estrutural diz respeito as **posições relativas** dos itens
 - Se $n \geq 1$, x_1 é o primeiro item e x_n é o último
 - Em geral, x_i precede x_{i+1} para $i = 1, 2, \dots, n - 1$ e x_i sucede x_{i-1} para $i = 2, 3, \dots, n$

Listas

Aplicação

- Diversos tipos de aplicações requerem uma lista
 - Lista telefônica
 - Lista de tarefas
 - Gerência de memória
 - Simulação
 - Compiladores, etc.

TAD Listas

Tipo Abstrato de Dados

- Vamos definir um TAD com as principais operações sobre uma lista
 - Para simplificar vamos definir apenas as operações principais, posteriormente, outras operações podem ser definidas

TAD Listas

Principais operações

- Criar lista
- Limpar lista
- Inserir item (última posição)
- Inserir item (por posição)
- Remover item (última posição)
- Recuperar item (dado uma chave)
- Recuperar item (por posição)
- Contar número de itens
- Verificar se a lista está vazia
- Verificar se a lista está cheia
- Imprime lista

TAD Listas I

Criar lista

- Pré-condição: nenhuma
- Pós-condição: inicia a estrutura de dados

Limpar lista

- Pré-condição: nenhuma
- Pós-condição: coloca a estrutura de dados no estado inicial

TAD Listas II

Inserir item (última posição)

- Pré-condição: a lista não está cheia
- Pós-condição: insere um item na última posição, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

Inserir item (por posição)

- Pré-condição: a lista não está cheia
- Pós-condição: insere um item na posição informada, que deve estar dentro da lista, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

TAD Listas III

Remover item (por posição)

- Pré-condição: uma posição válida da lista é informada
- Pós-condição: o item na posição fornecida é removido da lista, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

Recuperar item (dado uma chave)

- Pré-condição: nenhuma
- Pós-condição: recupera o item dado uma chave, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

TAD Listas IV

Recuperar item (por posição)

- Pré-condição: uma posição válida da lista é informada
- Pós-condição: recupera o item na posição fornecida, retorna **true** se a operação foi executada com sucesso, **false** caso contrário

Contar número de itens

- Pré-condição: nenhuma
- Pós-condição: retorna o número de itens na lista

TAD Listas V

Verificar se a lista está vazia

- Pré-condição: nenhuma
- Pós-condição: retorna **true** se a lista estiver vazia e **false** caso-contrário

Verificar se a lista está cheia

- Pré-condição: nenhuma
- Pós-condição: retorna **true** se a lista estiver cheia e **false** caso-contrário

TAD Listas VI

Imprime lista

- Pré-condição: nenhuma
- Pós-condição: imprime na tela os itens da lista

TAD Listas

Como implementar o TAD Listas?

Basicamente existem duas formas

- Estática (utilizando vetores)
- Dinâmica (utilizando listas ligadas)

Sumário

- 1 Conceitos Básicos
- 2 Implementação Estática de Listas
- 3 Listas Estáticas Ordenadas
- 4 Busca em Listas Estáticas
- 5 Conclusão

Listas - Implementação Estática

Características

- Os itens da lista são armazenados em posições contíguas de memória
- A lista pode ser percorrida em qualquer direção
- A inserção de um novo item pode ser realizada após o último item com custo constante

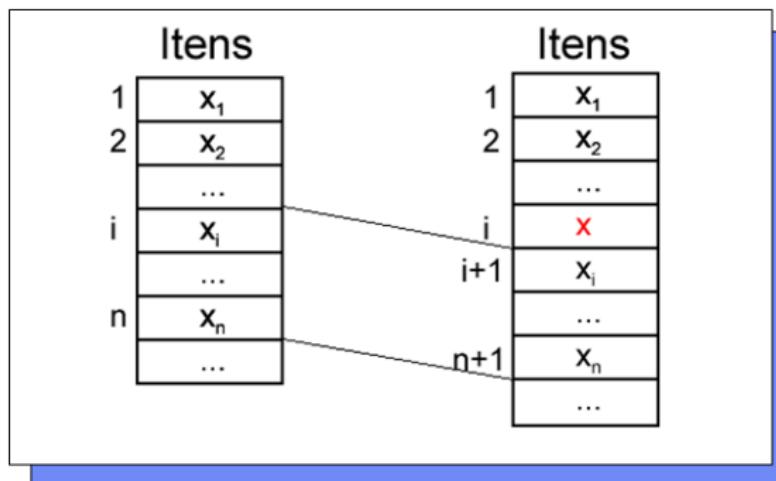
Listas - Implementação Estática

```
1 #ifndef LISTAESTATICA_H_INCLUDED
2 #define LISTAESTATICA_H_INCLUDED
3
4 #define TAM 100
5
6 //represents o item armazenado
7 typedef struct {
8     int chave;
9     int valor;
10 } ITEM;
11
12 //represents a lista de itens
13 typedef struct {
14     ITEM itens[TAM];
15     int fim;
16 } LISTA;
17
18 void criar(LISTA *lista);
19 void imprimir(LISTA *lista);
20 int vazia(LISTA *lista);
21 int cheia(LISTA *lista);
22 int inserir_fim(LISTA *lista, ITEM *item);
23 int inserir_posicao(LISTA *lista, int pos, ITEM *item);
24 int remover(LISTA *lista, int pos);
25 int tamanho(LISTA *lista);
26
27 #endif // LISTAESTATICA_H_INCLUDED
```

Listas - Implementação Estática

Exercícios

- Inserir um item na posição i
- Problema do descolamento de itens



Implementação da Inserção no Meio da Lista

```
1 int inserir_posicao(LISTA *lista, int pos, ITEM *item) {
2     int i;
3
4     //verifica se existe espaço e se a posição está na lista
5     if (!cheia(lista) && pos <= lista->fim + 1) {
6         for (i = lista->fim; i >= pos; i--) { //movo os itens
7             lista->itens[i + 1] = lista->itens[i];
8         }
9
10        lista->itens[pos] = *item; //insiro novo item
11        lista->fim++; //incremento tamanho
12
13        return 1;
14    } else {
15        return 0;
16    }
17 }
```

Listas - Implementação Estática

Exercícios

- Implemente uma função para retirar itens da lista dado uma posição

```
1 int remover(LISTA *lista, int pos){  
2     ...  
3 }
```

Listas - Implementação Estática

Exercícios

Crie funções que implementem as seguintes operações:

- Verificar se a lista **L** está ordenada (crescente ou decrescente)
- Fazer uma cópia da Lista **L1** em outra **L2**
- Fazer uma cópia da Lista **L1** em **L2**, eliminando repetidos
- Inverter **L1**, colocando o resultado em **L2**
- Inverter a própria **L1**
- Intercalar **L1** com **L2**, gerando **L3** ordenada (considere **L1** e **L2** ordenadas)
- Eliminar de **L1** todas as ocorrências de um dado item (**L1** está ordenada)

Sumário

- 1 Conceitos Básicos
- 2 Implementação Estática de Listas
- 3 Listas Estáticas Ordenadas
- 4 Busca em Listas Estáticas
- 5 Conclusão

Listas Ordenadas

Características

- Uma lista pode ser mantida em **ordem crescente/decrescente** segundo o valor de alguma **chave**
- Essa ordem **facilita a pesquisa** de itens
- Por outro lado, a **inserção e remoção são mais complexas** pois deve manter os itens ordenados

TAD Listas Ordenadas

Tipo Abstrato de Dados

- O TAD listas ordenadas é o mesmo do TAD listas, apenas difere na implementação
- As operações diferentes serão a inclusão e inserção de itens

Inserir item

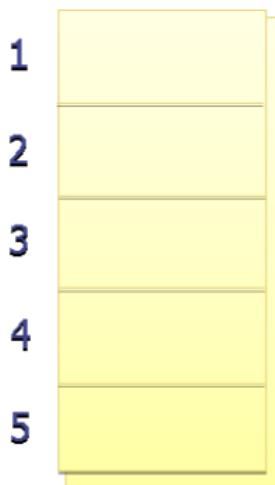
- Pré-condição: a lista não está cheia
- Pós-condição: insere um item em uma posição tal que a lista é mantida ordenada

Remover item (por posição)

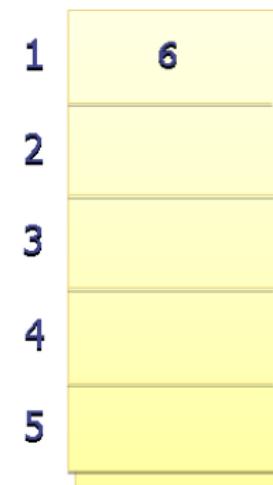
- Pré-condição: uma posição válida da lista é informada
- Pós-condição: o item na posição fornecida é removido da lista, a lista é mantida ordenada

Inserção Ordenada

- Exemplo de inserção ordenada

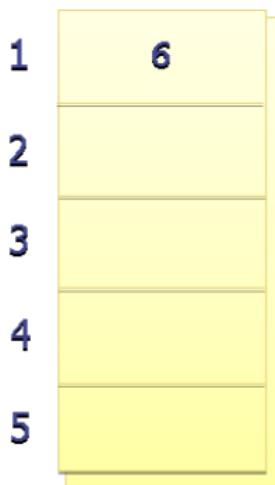


Insere valor 6 →

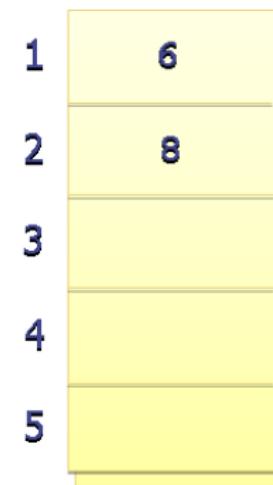


Inserção Ordenada

- Exemplo de inserção ordenada

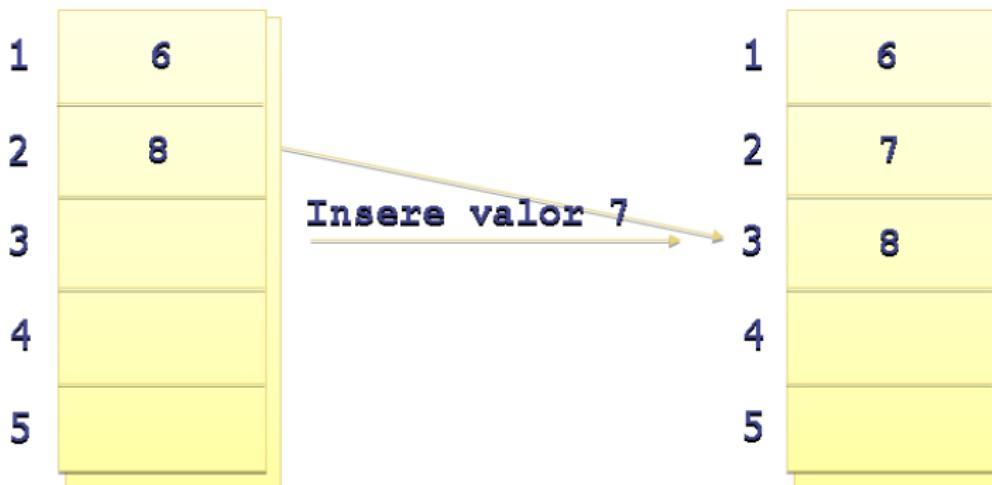


Inserção valor 8



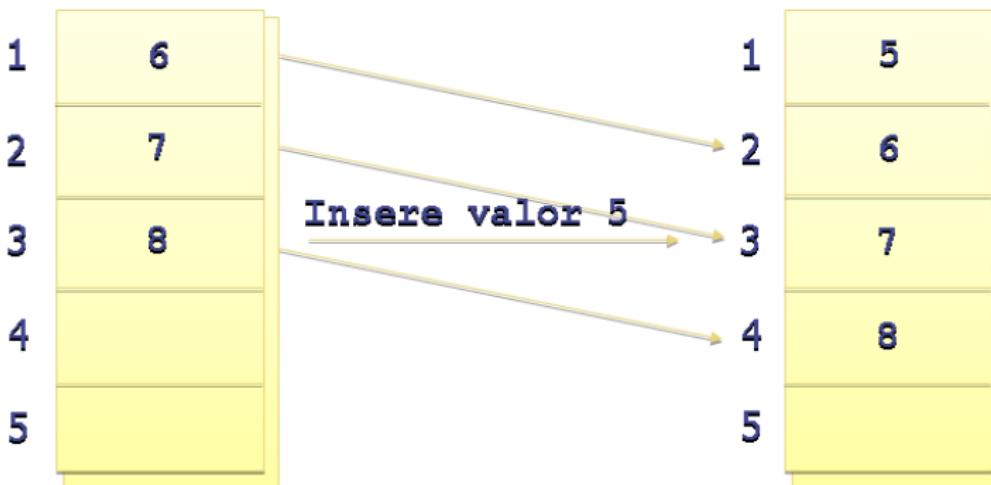
Inserção Ordenada

- Exemplo de inserção ordenada



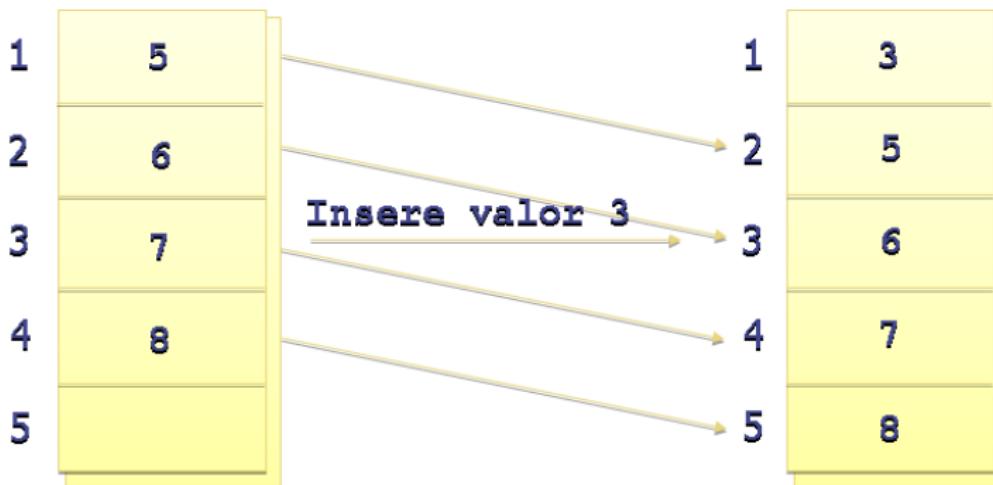
Inserção Ordenada

- Exemplo de inserção ordenada



Inserção Ordenada

- Exemplo de inserção ordenada



Implementação Inserção Ordenada

```
1 int inserir_ordenado(LISTA *lista, ITEM *item) {
2     if (!cheia(lista)) { //verifica se existe espaço
3         int pos = lista->fim+1;
4
5         //move os itens até encontrar a posição de inserção
6         while (pos > 0 && lista->itens[pos-1].chave > item->chave) {
7             lista->itens[pos] = lista->itens[pos-1];
8             pos--;
9         }
10
11         lista->itens[pos] = *item; //insere novo item
12         lista->fim++; //incrementa tamanho da lista
13         return 1;
14     }
15
16     return 0;
17 }
```

Implementação Remoção Ordenada

Exercícios

Implemente a remoção de itens de uma lista ordenada - mantendo a ordenação após a eliminação.

```
1 int remover(LISTA *lista, int pos){  
2     ...  
3 }
```

Sumário

- 1 Conceitos Básicos
- 2 Implementação Estática de Listas
- 3 Listas Estáticas Ordenadas
- 4 Busca em Listas Estáticas
- 5 Conclusão

Busca em Listas Estáticas

Introdução

- Uma tarefa comum a ser executada sobre listas é a **busca de itens** dado uma chave
- No caso da lista **não ordenada**, a busca será **sequencial** (consultar todos os elementos)
- Porém, com uma **lista ordenada**, diferentes estratégias podem ser aplicadas que aceleram essa busca
 - Busca sequencial “otimizada”
 - Busca binária

Busca Sequencial

Busca sequencial

- Na busca sequencial, a ideia é procurar um elemento que tenha uma determinada chave, começando do **início** da lista, e parar quando a **lista** terminar ou quando o elemento for encontrado

Implementação Busca Sequencial

```
1 int busca_sequencial(LISTA *lista, int chave, ITEM *item) {
2     int i;
3
4     for (i = 0; i <= lista->fim; i++) { //percorre a lista
5         if (lista->itens[i].chave == chave) { //se encontrar a chave
6             *item = lista->itens[i]; //armazeno o elemento
7             return 1; //indica que encontrou
8         } else if (lista->itens[i].chave > chave) { //se a chave é maior
9             return 0; //indica que não encontrou
10        }
11    }
12
13    return 0; //indica que não encontrou
14 }
```

Busca Sequencial (Ordenada)

- Quando os elementos estão ordenados, a busca sequencial pode ser “otimizada” (acelerada)
- Vejamos o seguinte exemplo

Busca Sequencial (Ordenada)

Exemplo 1

- procurar pelo valor 4

1	3	3 é diferente de 4
2	5	
3	6	
4	7	
5	8	

Busca Sequencial (Ordenada)

Exemplo 1

- procurar pelo valor 4

1	3
2	5
3	6
4	7
5	8



5 é diferente de 4

Busca Sequencial (Ordenada)

Exemplo 1

- procurar pelo valor 4

1	3
2	5
3	6
4	7
5	8



**5 é diferente de 4
E maior que 4!**

Busca Sequencial (Ordenada)

Exemplo 2

- procurar pelo valor 8

1	3	3 é diferente de 8
2	5	
3	6	
4	7	
5	8	

Busca Sequencial (Ordenada)

Exemplo 2

- procurar pelo valor 8

1	3
2	5
3	6
4	7
5	8

5 é diferente de 8

Busca Sequencial (Ordenada)

Exemplo 2

- procurar pelo valor 8

1	3
2	5
3	6
4	7
5	8

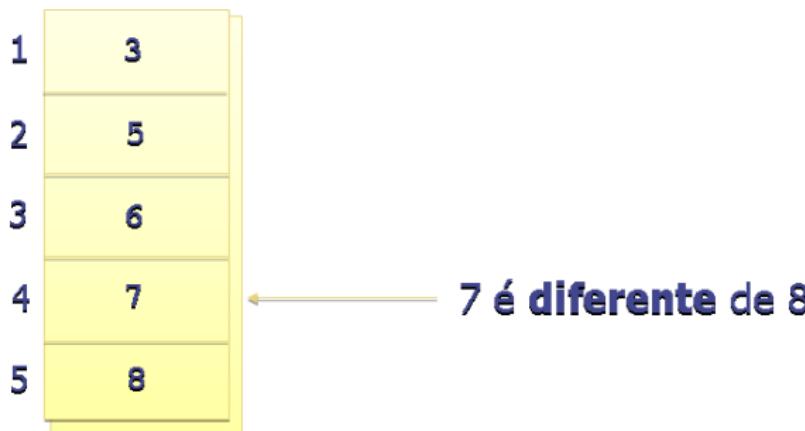


6 é diferente de 8

Busca Sequencial (Ordenada)

Exemplo 2

- procurar pelo valor 8



Busca Sequencial (Ordenada)

Exemplo 2

- procurar pelo valor 8



Busca Sequencial (Ordenada)

- A **lista ordenada** permite **realizar buscas sequenciais mais rápidas** uma vez que, caso a chave procurada não exista, pode-se parar a busca tão logo se encontre um elemento com chave maior que a procurada
- Entretanto, essa melhoria **não altera a complexidade da busca sequencial**, que ainda é $O(n)$. Porque?

Busca Binária

- A **busca binária** é um algoritmo de busca mais sofisticado e **bem mais eficiente** que a busca sequencial
- Entretanto, a busca binária **somente pode ser aplicada em estruturas que permitem acessar cada elemento em tempo constante**, tais como os vetores
- A ideia é, a cada iteração, dividir o vetor ao meio e descartar metade do vetor
- Essa ideia é melhor ilustrada utilizando uma figura...

Busca Binária

Exemplo 1

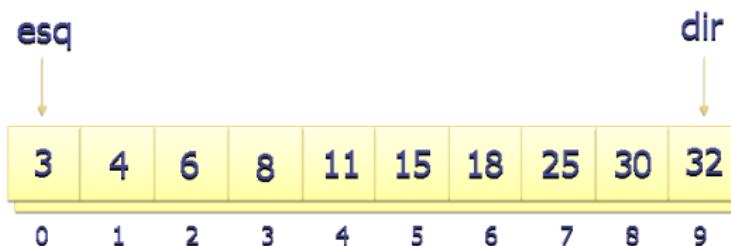
- Para procurar pelo valor 30

3	4	6	8	11	15	18	25	30	32
0	1	2	3	4	5	6	7	8	9

Busca Binária

Exemplo 1

- Para procurar pelo valor 30

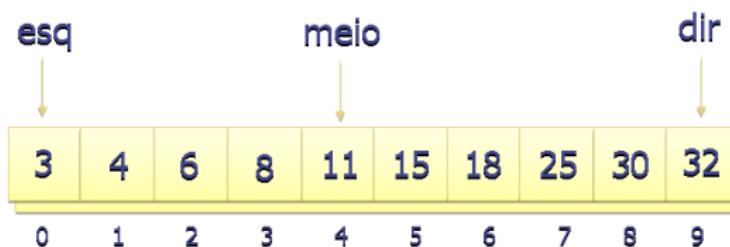


$$\text{meio} = (\text{esq} + \text{dir}) / 2$$
$$9 / 2 = 4$$

Busca Binária

Exemplo 1

- Para procurar pelo valor 30

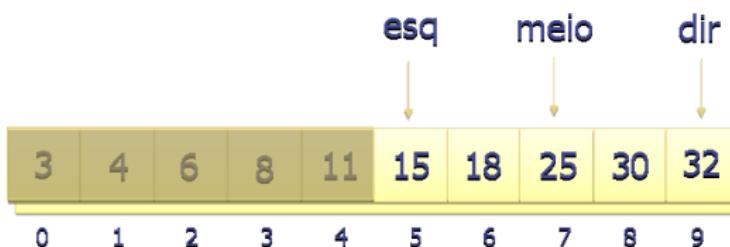


itens[meio].chave **menor** do que 30
esq = meio + 1

Busca Binária

Exemplo 1

- Para procurar pelo valor 30

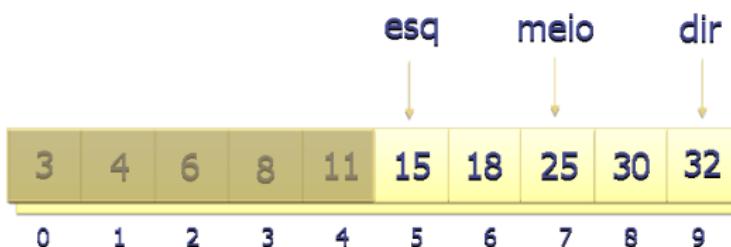


$$\text{meio} = (\text{esq} + \text{dir}) / 2$$
$$14 / 2 = 7$$

Busca Binária

Exemplo 1

- Para procurar pelo valor 30

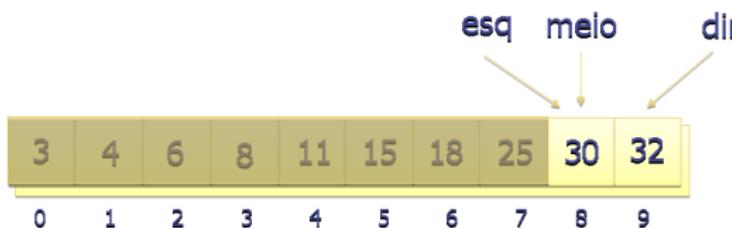


itens[meio].chave **menor** do que 30
esq = meio + 1

Busca Binária

Exemplo 1

- Para procurar pelo valor 30

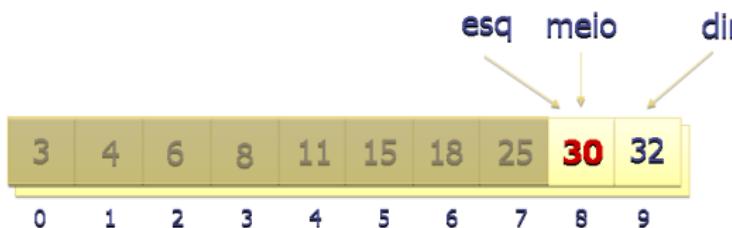


itens[meio].chave **igual** a 30

Busca Binária

Exemplo 1

- Para procurar pelo valor 30



Chave Encontrada.

Busca Binária

Exemplo 2

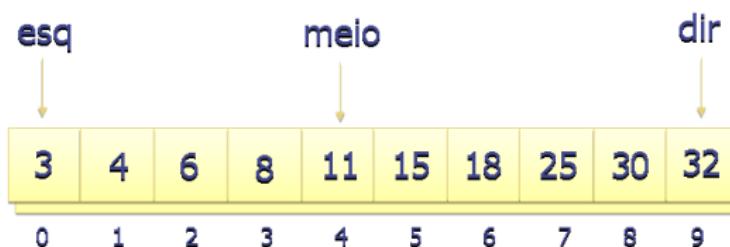
- Para procurar pelo valor 7 (inexistente!)

3	4	6	8	11	15	18	25	30	32
0	1	2	3	4	5	6	7	8	9

Busca Binária

Exemplo 2

- Para procurar pelo valor 7 (inexistente!)

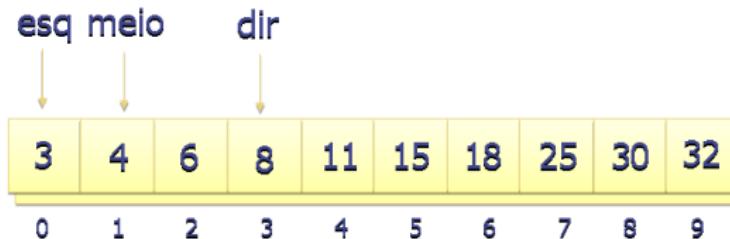


itens[meio].chave **maior** do que 7
dir = meio - 1

Busca Binária

Exemplo 2

- Para procurar pelo valor 7 (inexistente!)

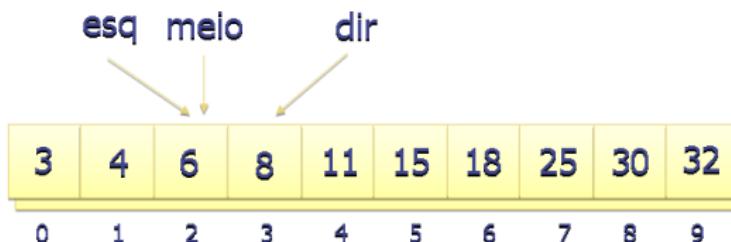


itens[meio].chave **menor** do que 7
esq = meio + 1

Busca Binária

Exemplo 2

- Para procurar pelo valor 7 (inexistente!)

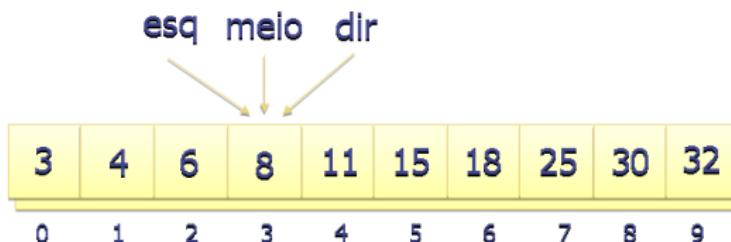


itens[meio].chave **menor** do que 7
esq = meio + 1

Busca Binária

Exemplo 2

- Para procurar pelo valor 7 (inexistente!)

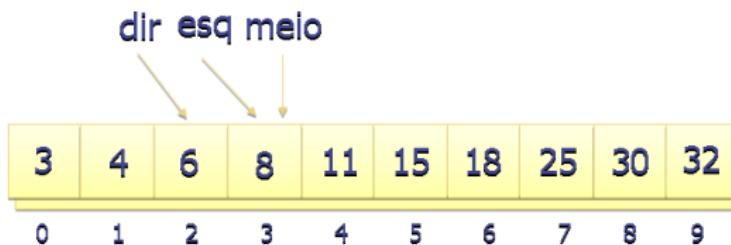


itens[meio].Chave **maior** do que 7
dir = meio - 1

Busca Binária

Exemplo 2

- Para procurar pelo valor 7 (inexistente!)



dir < esq => chave não encontrada!

Busca Binária

- É importante lembrar que
 - Busca binária somente funciona em vetores ordenados
 - Busca sequencial funciona com vetores ordenados ou não
- A busca binária é muito eficiente. Ela é $O(\log_2 n)$. Para $n = 1.000.000$, aproximadamente 20 comparações são necessárias

Implementação Busca Binária

```
1 int busca_binaria(LISTA *lista, int chave, ITEM *item) {
2     int esq = 0;
3     int dir = lista->fim;
4
5     while (esq <= dir) {
6         int meio = (esq + dir) / 2; //calcula meio do vetor
7
8         if (lista->itens[meio].chave == chave) { //encontrou a chave
9             *item = lista->itens[meio];
10            return 1;
11        }
12
13        if (lista->itens[meio].chave > chave) { //o meio é maior que a ←
14            chave
15            dir = meio - 1; //desconsidero os itens maiores que o meio
16        } else {
17            esq = meio + 1; //desconsidero os itens menores que o meio
18        }
19
20    return 0;
21 }
```

Exercício

Exercício

- Implemente a busca binária utilizando um procedimento recursivo

Sumário

- 1 Conceitos Básicos
- 2 Implementação Estática de Listas
- 3 Listas Estáticas Ordenadas
- 4 Busca em Listas Estáticas
- 5 Conclusão

Pontos fortes e fracos

Pontos Fortes

- Tempo constante de acesso aos dados

Pontos Fracos

- Custo para inserir e retirar elementos da lista, dada uma posição fornecida pelo usuário
- Tamanho máximo da lista é (dependendo da linguagem) definido em tempo de compilação

Quando utilizar...

Quando utilizar...

Essa implementação simples é mais comumente utilizadas em certas situações:

- Listas pequenas
- Tamanho máximo da lista é conhecido
- Poucas ocorrências de utilização dos métodos de inserção e remoção, dada uma posição definida pelo usuário