

# Buscas em estruturas lineares

## Definições e busca binária

Prof. Marcelo de Souza

45RPE – Resolução de Problemas com Estruturas de Dados

Universidade do Estado de Santa Catarina



## Leitura principal:

- ▶ Capítulo 5 de [Goodrich et al. \(2014\)](#)<sup>1</sup> – Recursão.

## Leitura complementar:

- ▶ Capítulo 5 de [Ziviani \(2010\)](#)<sup>2</sup> – Pesquisa em memória binária.
- ▶ Capítulo 13 de [Pereira \(2008\)](#)<sup>3</sup> – Ordenação e busca.

---

<sup>1</sup>Michael T Goodrich et al. (2014). *Data structures and algorithms in Java*. 6ª ed. John Wiley & Sons.

<sup>2</sup>Nivio Ziviani (2010). *Projeto de Algoritmos com Implementações em Java e C++*. Cengage Learning.

<sup>3</sup>Silvio do Lago Pereira (2008). *Estruturas de Dados Fundamentais: Conceitos e Aplicações*.

# Buscas em estruturas lineares

## Conceitos básicos



Buscar um elemento consiste em verificar se ele está armazenado em uma estrutura de dados.

# Buscas em estruturas lineares

## Conceitos básicos



Buscar um elemento consiste em verificar se ele está armazenado em uma estrutura de dados.

O retorno pode ser:

1. o próprio elemento;
2. a posição onde ele se encontra (-1, caso não seja encontrado); ou
3. um valor lógico indicando o sucesso ou a falha da busca.

# Buscas em estruturas lineares

## Conceitos básicos

Buscar um elemento consiste em verificar se ele está armazenado em uma estrutura de dados.

O retorno pode ser:

1. o próprio elemento;
2. a posição onde ele se encontra (-1, caso não seja encontrado); ou
3. um valor lógico indicando o sucesso ou a falha da busca.

Estratégias:

- ▶ Busca sequencial (ou linear) –  $\mathcal{O}(n)$ .
- ▶ Busca binária –  $\mathcal{O}(\log n)$ .





É a forma mais simples de busca: percorre a estrutura até encontrar o elemento.

- ▶ Logo, sua complexidade assintótica é  $\mathcal{O}(n)$  no pior caso.



É a forma mais simples de busca: percorre a estrutura até encontrar o elemento.

► Logo, sua complexidade assintótica é  $\mathcal{O}(n)$  no pior caso.

Uma busca sequencial **simples** em um *array* de inteiros:

```
1 public int indexOf(int[] array, int value) {  
2     for(int i = 0; i < array.length; i++)  
3         if(array[i] == value)  
4             return i;  
5     return -1;  
6 }
```



Uma busca sequencial **genérica** em um *array*:

```
1 public <E> int indexOf(E[] array, E value) {  
2     for(int i = 0; i < array.length; i++)  
3         if(array[i].equals(value))  
4             return i;  
5     return -1;  
6 }
```





Uma busca sequencial **genérica** em um *array*:

```
1 public <E> int indexOf(E[] array, E value) {  
2     for(int i = 0; i < array.length; i++)  
3         if(array[i].equals(value))  
4             return i;  
5     return -1;  
6 }
```

- ▶ O *array* armazena elementos do tipo genérico E.
- ▶ A assinatura do método define o uso desse tipo genérico: `public <E> int` (linha 1).



Uma busca sequencial **genérica** em um *array*:

```
1 public <E> int indexOf(E[] array, E value) {  
2     for(int i = 0; i < array.length; i++)  
3         if(array[i].equals(value))  
4             return i;  
5     return -1;  
6 }
```

- ▶ O *array* armazena elementos do tipo genérico E.
- ▶ A assinatura do método define o uso desse tipo genérico: `public <E> int` (linha 1).
- ▶ O método `equals` é usado para comparação (linha 3). Por padrão, ele verifica se os elementos são o mesmo objeto, i.e. se apontam para o mesmo endereço de memória.
- ▶ Mas, e **se quisermos comparar objetos de outra maneira?**
  - ▶ e.g., checar a igualdade dos valores dos seus atributos.



# Comparação de igualdade de objetos

Para definir outra estratégia de comparação, a classe deve sobrescrever o método `equals`:

```
1  public class Point {
2      private double x;
3      private double y;
4      //...
5
6      @Override
7      public boolean equals(Object obj) {
8          if (this == obj) return true;
9          if (obj == null) return false;
10         if (getClass() != obj.getClass()) return false;
11         Point o = (Point) obj;
12         return this.x == o.getX() && this.y == o.getY();
13     }
14 }
```

# Busca sequencial



Uma busca sequencial **genérica** em uma lista encadeada, implementada na DoublyLinkedList:

```
1  public int indexOf(E e) {  
2      if(isEmpty()) return -1;  
3      int count = -1;  
4      Node<E> walk = header.getNext();  
5      while(walk != trailer) {  
6          count++;  
7          if(walk.getElement().equals(e))  
8              return count;  
9          walk = walk.getNext();  
10     }  
11     return -1;  
12 }
```

Uma busca sequencial **genérica** em uma lista encadeada, implementada na DoublyLinkedList:

```
1  public int indexOf(E e) {  
2      if(isEmpty()) return -1;  
3      int count = -1;  
4      Node<E> walk = header.getNext();  
5      while(walk != trailer) {  
6          count++;  
7          if(walk.getElement().equals(e))  
8              return count;  
9          walk = walk.getNext();  
10     }  
11     return -1;  
12 }
```

- ▶ A lista encadeada precisa ser percorrida (linhas 4 a 10), verificando se o elemento do nodo atual é igual ao buscado (linha 7).
- ▶ A variável count computa a posição do nodo atual; seu valor é retornado quando o elemento é encontrado (linha 8).



A **busca binária** é uma técnica mais eficiente de busca em um **array ordenado**.

- ▶ Sua complexidade assintótica é  $\mathcal{O}(\log n)$  no pior caso.



A **busca binária** é uma técnica mais eficiente de busca em um **array ordenado**.

- ▶ Sua complexidade assintótica é  $\mathcal{O}(\log n)$  no pior caso.

## Pré-condições:

- ▶ Os elementos devem estar **ordenados para o algoritmo funcionar**.
- ▶ A estrutura deve permitir **acesso aleatório** (arranjos) **para garantir a eficiência**.



A **busca binária** é uma técnica mais eficiente de busca em um **array ordenado**.

- ▶ Sua complexidade assintótica é  $\mathcal{O}(\log n)$  no pior caso.

## Pré-condições:

- ▶ Os elementos devem estar **ordenados para o algoritmo funcionar**.
- ▶ A estrutura deve permitir **acesso aleatório** (arranjos) **para garantir a eficiência**.

## Funcionamento:

1. Avalia o elemento central da lista.
2. Caso seja o elemento buscado, sucesso.
3. Caso contrário, avalia em qual sub-lista o elemento pode estar.
4. Repete a busca com a sub-lista correspondente.



# Busca binária

## Exemplo de funcionamento

Busca binária do elemento 22 no array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

## Busca binária

## Exemplo de funcionamento

Busca binária do elemento 22 no array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

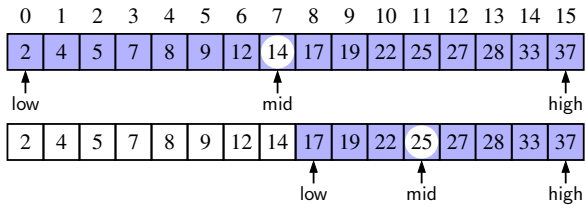
Diagram illustrating a sorted array with indices 0 to 15. The array contains values [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]. The element 14 at index 7 is highlighted. Arrows point to index 0 labeled 'low', index 7 labeled 'mid', and index 15 labeled 'high'.

# Busca binária

## Exemplo de funcionamento

Busca binária do elemento 22 no array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

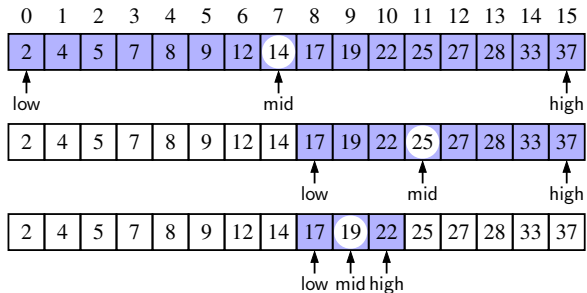


# Busca binária

## Exemplo de funcionamento

Busca binária do elemento 22 no array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

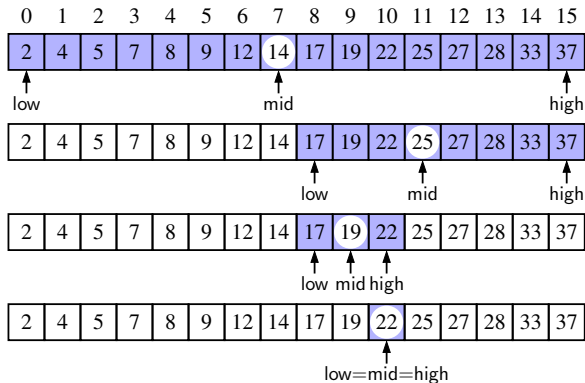


## Busca binária

### Exemplo de funcionamento

Busca binária do elemento 22 no array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

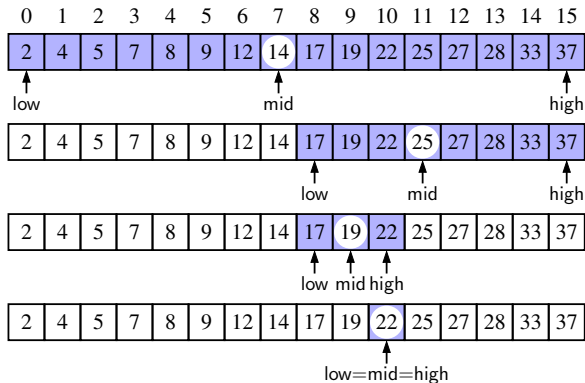


## Busca binária

## Exemplo de funcionamento

Busca binária do elemento 22 no array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37



### Conclusões:

- ▶ Encontra o elemento em 4 avaliações: (14, 25, 19, 22).
- ▶ Note que  $4 = \log_2 16$  (pior caso).
- ▶ Ao buscar o elemento 23, o algoritmo identifica sua inexistência quando `high < low`.

Uma busca binária **simples** em um *array* de inteiros:

```
1  public static int indexOf(int[] array, int value) {
2      int low = 0;
3      int high = array.length - 1;
4      int mid;
5
6      do {
7          mid = (low + high) / 2;
8          if(array[mid] < value)
9              low = mid + 1;
10         else
11             high = mid - 1;
12     } while(array[mid] != value && low <= high);
13
14     if(array[mid] == value)
15         return mid;
16     else
17         return -1;
18 }
```



# Busca binária

## Comparação (detalhada) de objetos

Que tal uma busca binária **genérica**? Neste caso:

- ▶ Não basta verificar se dois objetos são iguais;
- ▶ Precisamos **saber qual deles é menor/maior**.





# Busca binária

## Comparação (detalhada) de objetos

Que tal uma busca binária **genérica**? Neste caso:

- ▶ Não basta verificar se dois objetos são iguais;
- ▶ Precisamos **saber qual deles é menor/maior**.

Para isso, devemos:

1. Implementar a interface Comparable;
2. Implementar o método compareTo.

```
1 public class Point implements Comparable<Point> {
2     //...
3     @Override
4     public int compareTo(Point o) {
5         if(this.x == o.getX()) return ((int) this.y - (int) o.getY());
6         return ((int) this.x - (int) o.getX());
7     }
8 }
```

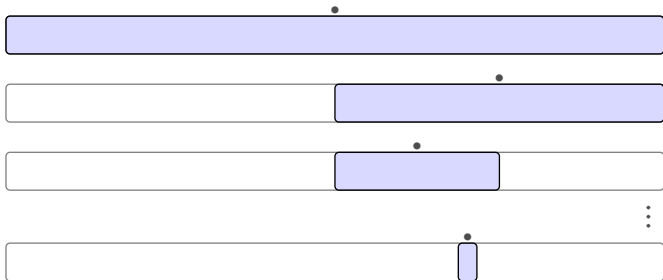


Uma busca binária **genérica** em um *array*:

```
1  public static <E extends Comparable<? super E>> int indexOf(E[] array, E value) {
2      int low = 0;
3      int high = array.length - 1;
4      int mid;
5
6      do {
7          mid = (low + high) / 2;
8          if(array[mid].compareTo(value) < 0)
9              low = mid + 1;
10         else
11             high = mid - 1;
12     } while(array[mid].compareTo(value) != 0 && low <= high);
13
14     if(array[mid].compareTo(value) == 0)
15         return mid;
16     else
17         return -1;
18 }
```

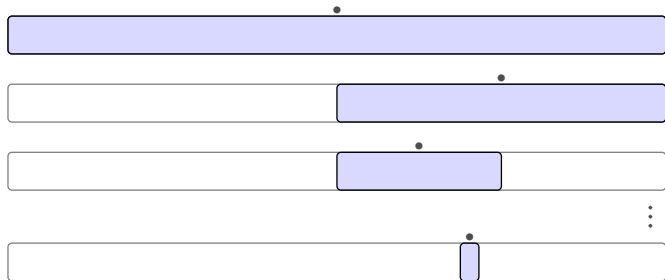
# Busca binária

## Complexidade



# Busca binária

Complexidade



**Tamanho da entrada**

$$n \quad \therefore \quad n/2^0$$

$$n/2 \quad \therefore \quad n/2^1$$

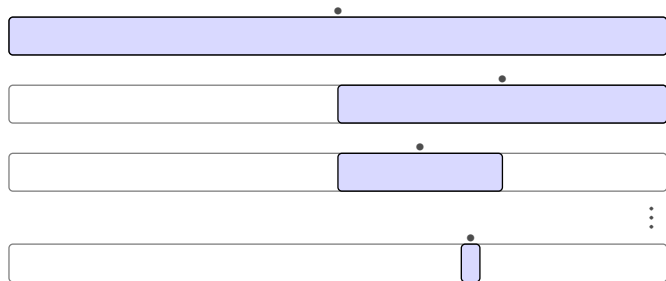
$$n/4 \quad \therefore \quad n/2^2$$

$$\vdots \quad \quad \quad \vdots$$

$$n/n \quad \therefore \quad n/2^i$$

# Busca binária

## Complexidade



## Tamanho da entrada

$$n \quad \therefore \quad n/2^0$$

$$n/2 \quad \therefore \quad n/2^1$$

$$n/4 \quad \therefore \quad n/2^2$$

$$\vdots \quad \quad \quad \vdots$$

$$n/n \quad \therefore \quad n/2^i$$

- ▶ O algoritmo executa  $i + 1$  iterações no pior caso.
- ▶ O valor de  $i$  é tal que  $2^i = n$ .
- ▶ Logo, temos  $i = (\log_2 n) + 1$  iterações no pior caso.
- ▶ A complexidade da busca binária no pior caso é logarítmica, i.e.  $\mathcal{O}(\log n)$ .



Veja o funcionamento das buscas binária usando recursos de visualização:

► <https://www.cs.usfca.edu/~galles/visualization/Search.html>.

Veja a documentação do Java sobre os métodos `binarySearch` da classe `Arrays`:

► <https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/Arrays.html>.

Veja a documentação do Java sobre os métodos `indexOf` da classe `LinkedList`:

► <https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/LinkedList.html>.

45RPE – Resolução de Problemas com Estruturas de Dados  
Prof. Marcelo de Souza