

Buscas em estruturas lineares

Buscas sequencial e binária

Prof. Marcelo de Souza

45RPE – Resolução de Problemas com Estruturas de Dados
Universidade do Estado de Santa Catarina

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 em um *array*:

```
1 def index(array, value):
2     for i in range(len(array)):
3         if array[i] == value:
4             return value
5     return -1
```



Uma busca sequencial em uma lista encadeada (classe LinkedList):

```
1  def index(self, e):
2      if self.is_empty(): return -1
3      count = -1
4      walk = self.header.next
5      while walk != self.trailer:
6          count += 1
7          if walk.element == e:
8              return count
9          walk = walk.next
10     return -1
```

Uma busca sequencial em uma lista encadeada (classe LinkedList):

```
1  def index(self, e):  
2      if self.is_empty(): return -1  
3      count = -1  
4      walk = self.header.next  
5      while walk != self.trailer:  
6          count += 1  
7          if walk.element == e:  
8              return count  
9          walk = walk.next  
10     return -1
```

- ▶ A lista encadeada precisa ser percorrida (linhas 5 a 9), 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

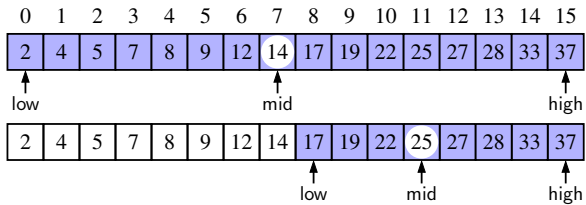
Diagram illustrating a sorted array with indices 0 to 15. The array contains values [2, 4, 5, 7, 8, 9, 12, 14, 17, 19, 22, 25, 27, 28, 33, 37]. The element 14 at index 7 is highlighted. Arrows point to index 0 labeled 'low', index 7 labeled 'mid', and index 15 labeled 'high'.

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

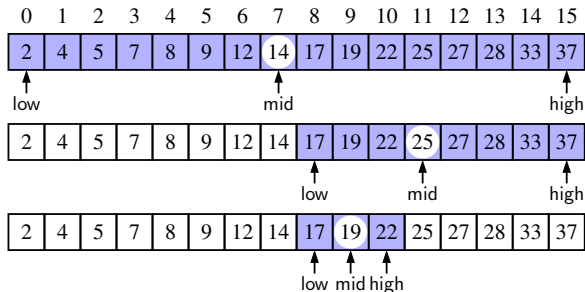


Busca binária

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

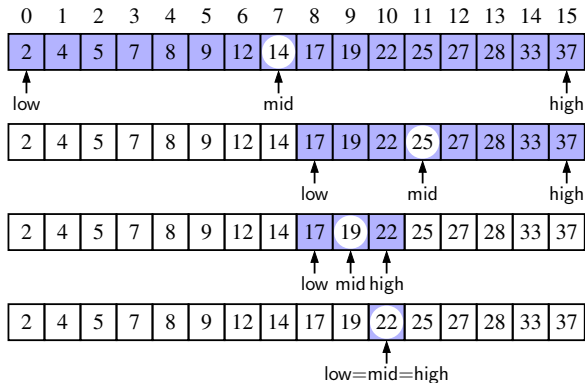


Busca binária

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

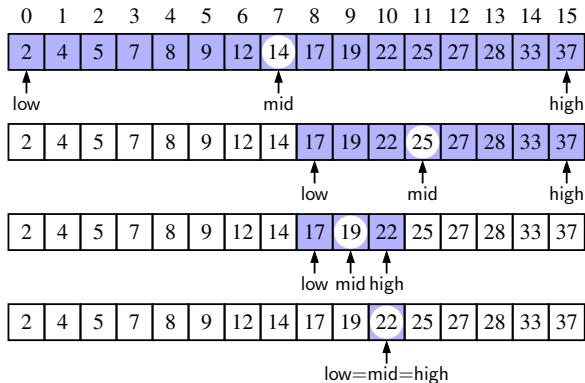


Busca binária

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:

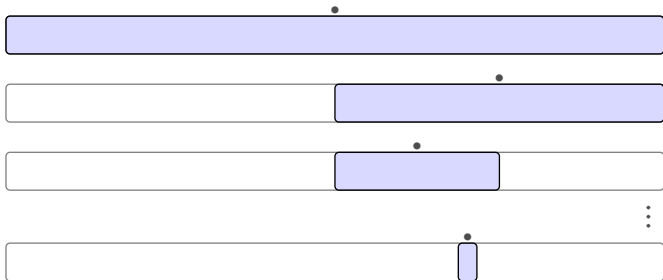
- ▶ Encontra o elemento em 4 avaliações: (14, 25, 19, 22).
- ▶ Note que $4 = \log_2 16$ (pior caso).
- ▶ Ao buscar o elemento 23, o algoritmo identifica sua inexistência quando `high < low`.

Uma busca binária em um *array*:

```
1 def binary_search(array, value):
2     low = 0
3     high = len(arr) - 1
4
5     while low <= high:
6         mid = (low + high) // 2
7         if array[mid] == value:
8             return mid
9         if array[mid] < value:
10            low = mid + 1
11        else:
12            high = mid - 1
13
14    return -1
```

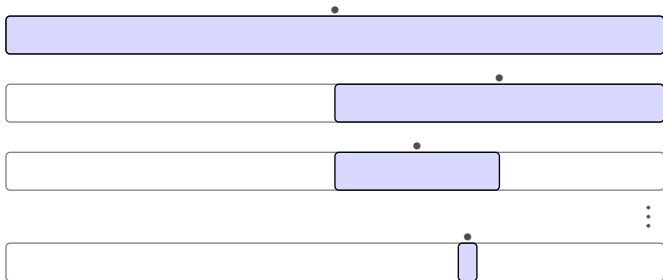
Busca binária

Análise de complexidade



Busca binária

Análise de complexidade



Tamanho da entrada

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

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

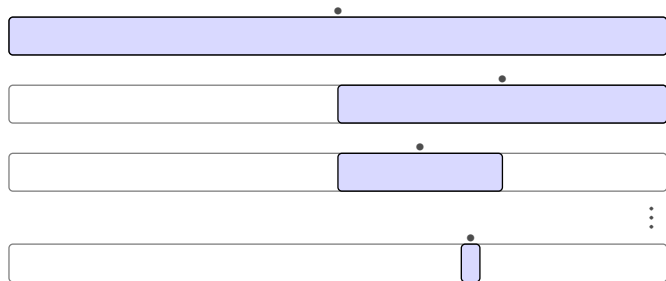
$$n/4 \quad \therefore \quad n/2^2$$

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

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

Busca binária

Análise de 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 sequencial e binária usando recursos de visualização:

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

Veja detalhes da busca binária em Python:

- ▶ <https://www.geeksforgeeks.org/python/python-program-for-binary-search>.

- ▶ https://www.w3schools.com/python/python_dsa_binarysearch.asp.

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