Algoritmos e Estruturas de Dados II

2º Período Engenharia da Computação

Prof. Edwaldo Soares Rodrigues
Email: edwaldo.rodrigues@uemg.br

UNIVERSIDADE
DO ESTADO DE MINAS GERAIS | UEMG
UNIDADE DIVINÓPOLIS

Busca em Arranjos

- Como sabemos, arranjos são uma maneira de armazenar muita informação em série

- Esta informação, em situações práticas, é usualmente dividida em registros do C com o recurso **struct**

- Precisamos de estratégias para encontrar dados nestes arranjos

Busca em Arranjos

- Usualmente, cada **struct** do arranjo tem um campo que chamamos de chave

- Por exemplo, suponha um arranjo de dados do tipo **Aluno**:

```
struct Aluno{
    string nome;
    double nota_prova1;
    double nota_prova2;
    int matricula;
}
```

Busca em Arranjos

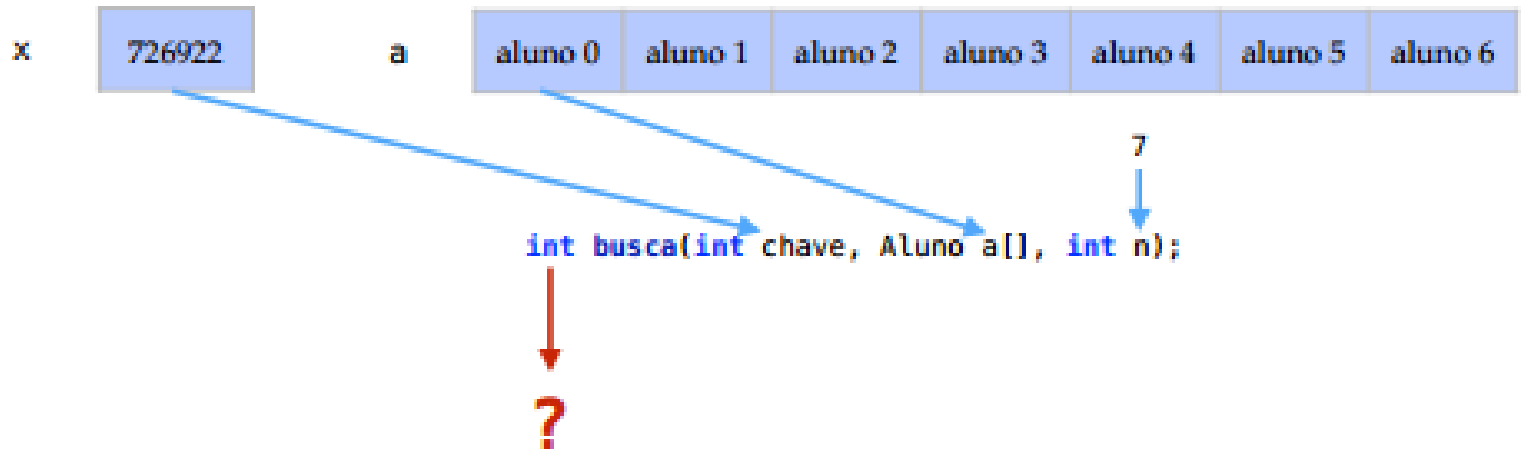
- Nesta estrutura, cada aluno terá um nome, uma nota para cada prova e um número de matrícula

- Como os números de matrícula não se repetem, eles são bons candidatos a serem o **membro chave** dos alunos

```
struct Aluno{
    string nome;
    double nota_prova1;
    double nota_prova2;
    int matricula;
}
```

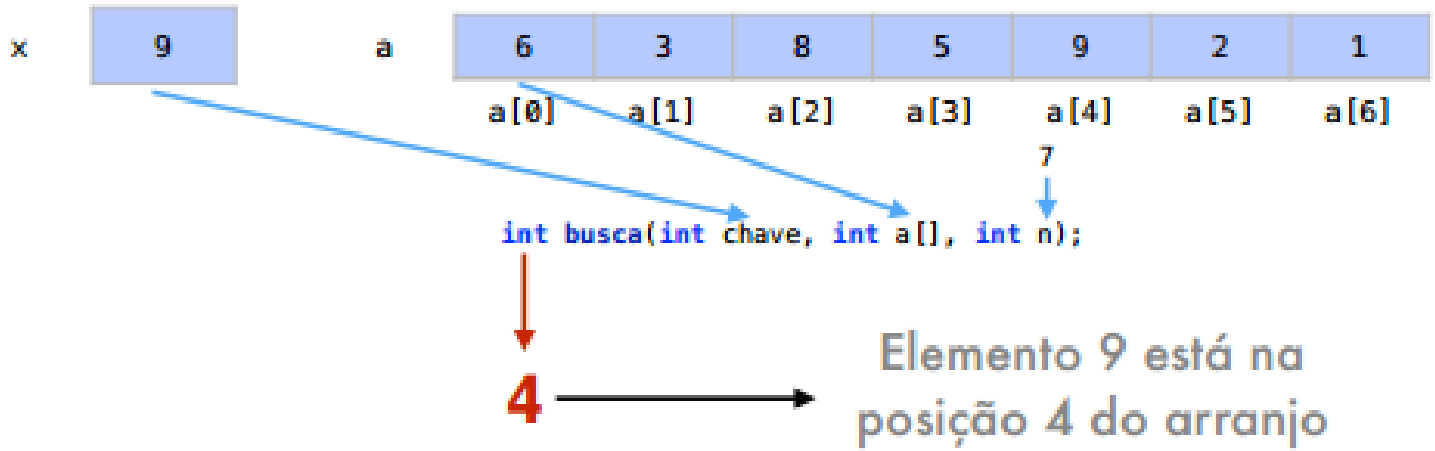
Busca em Arranjos

- Neste caso prático, o problema da busca em arranjo consiste em encontrar em qual posição do arranjo a está o aluno com número de matrícula x



Busca em Arranjos

- Para apresentação didática dos métodos, consideraremos arranjos de int onde o próprio valor do int é sua chave



Busca em Arranjos

- Dois métodos serão apresentados:
  - Busca Sequencial;
  - Busca Binária;

Busca Sequencial

Busca Sequencial

- O método mais simples para busca é, a partir do primeiro elemento, pesquisar sequencialmente um a um até encontrar a chave procurada

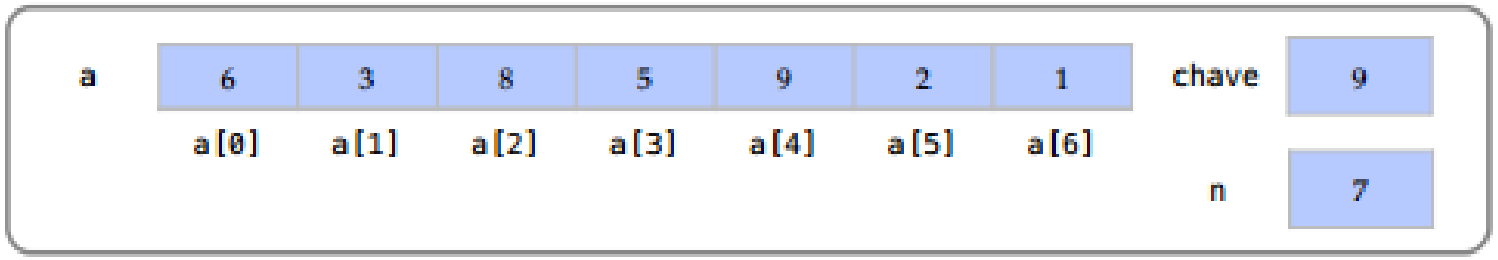
Busca Sequencial

```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```

Busca Sequencial

- A função procura o elemento chave no arranjo a, que tem tamanho n

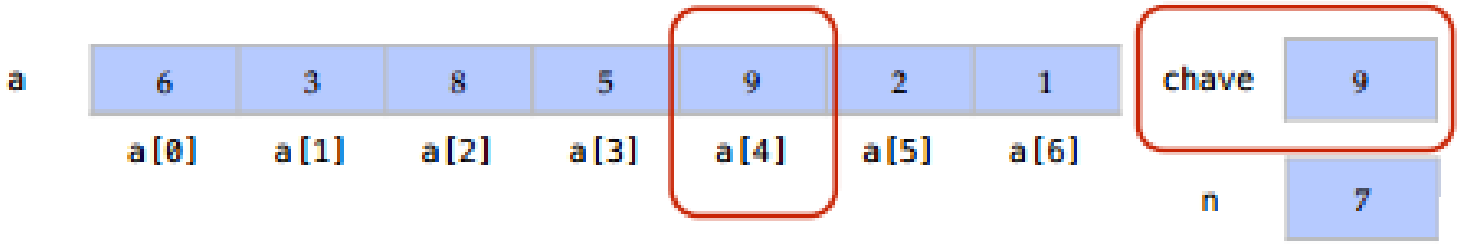
```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```



Busca Sequencial

- Ela retornará um int que dirá em qual posição do arranjo a está o **elemento chave**

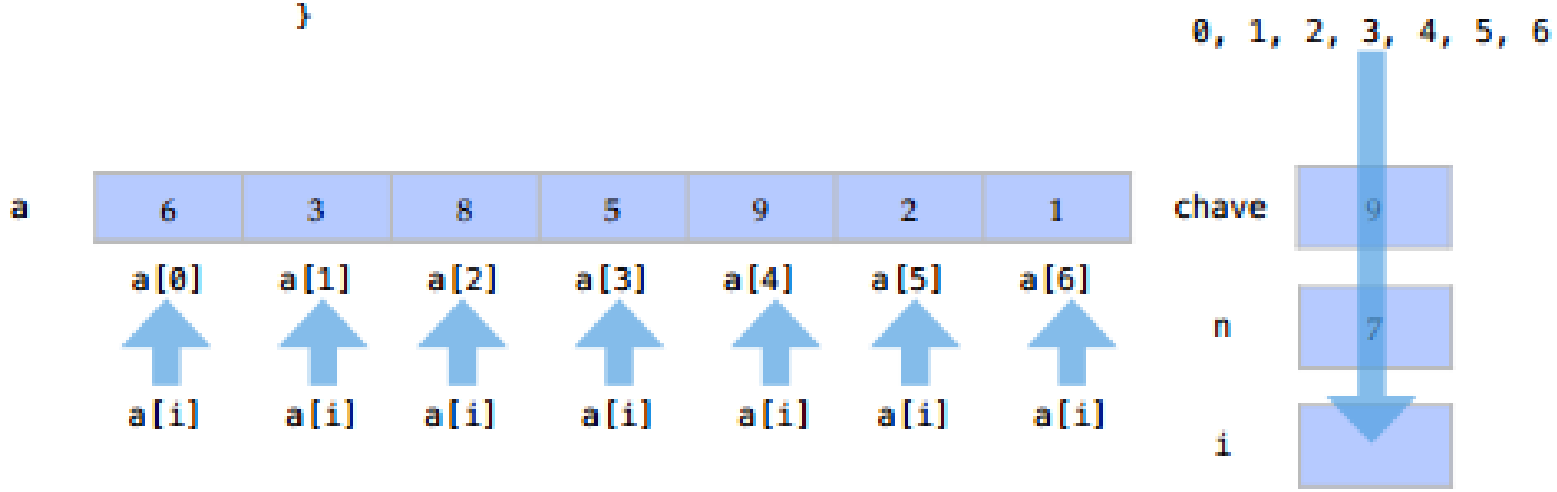
```
4
↑
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```



Busca Sequencial

- A estrutura de repetição ocorre com i de 0 até n-1 para percorrer da primeira até a última posição a[i] do arranjo a

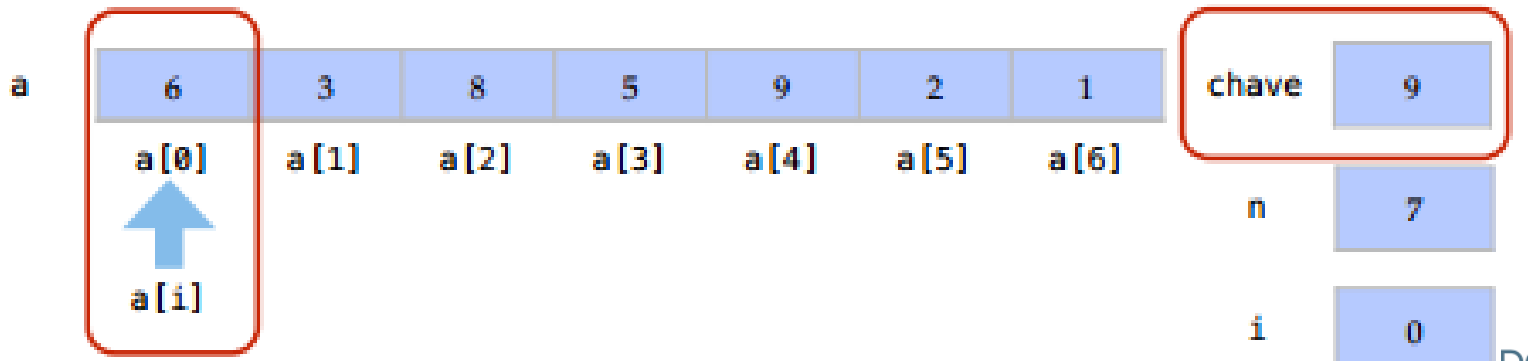
```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```



Busca Sequencial

- Comparamos cada elemento do arranjo com o elemento chave

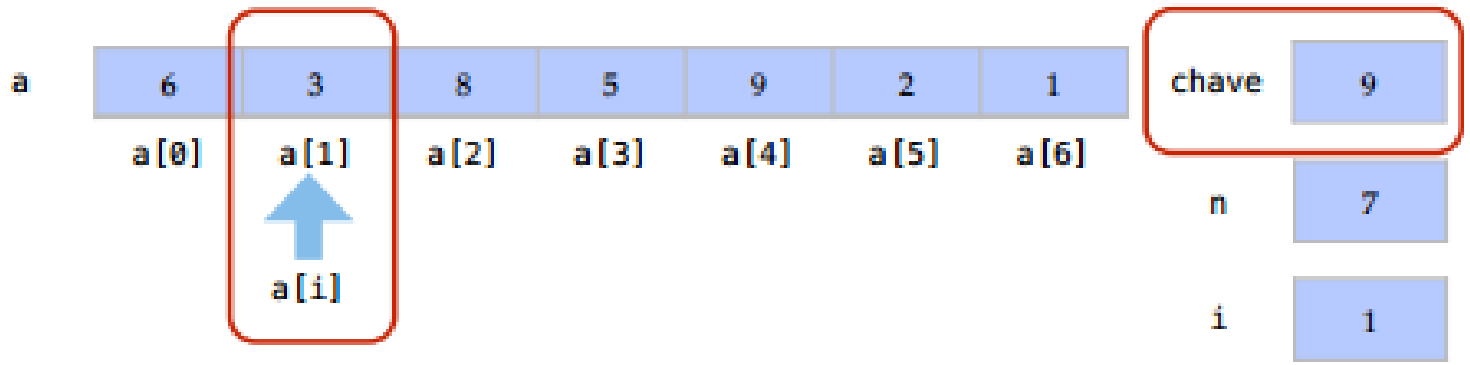
```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```



Busca Sequencial

- Comparamos cada elemento do arranjo com o elemento chave

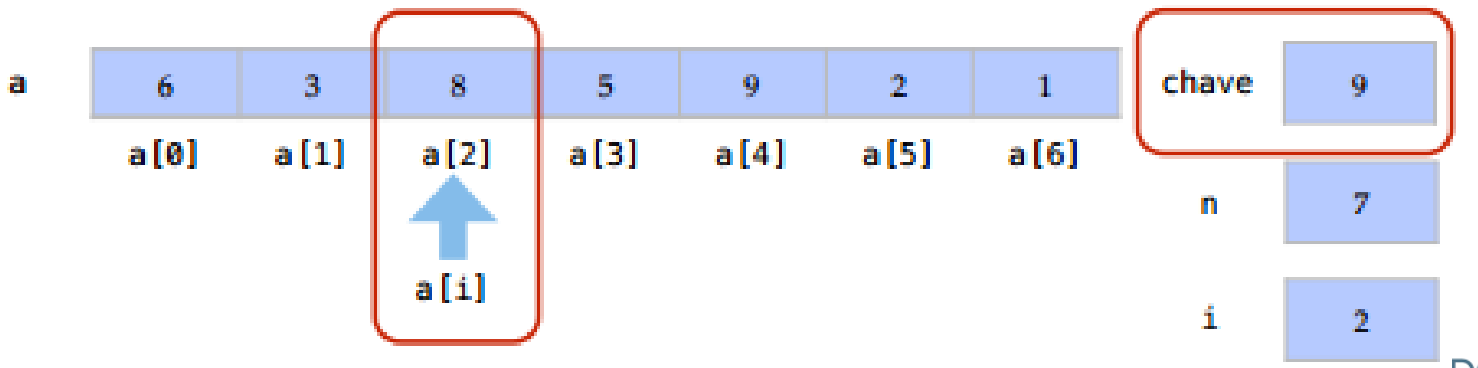
```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```



Busca Sequencial

- Comparamos cada elemento do arranjo com o elemento chave

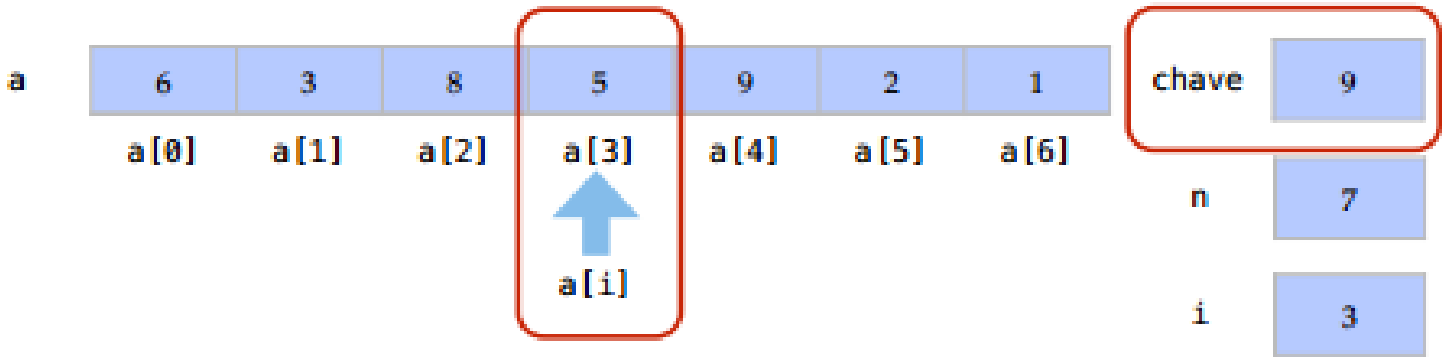
```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```



Busca Sequencial

- Comparamos cada elemento do arranjo com o elemento chave

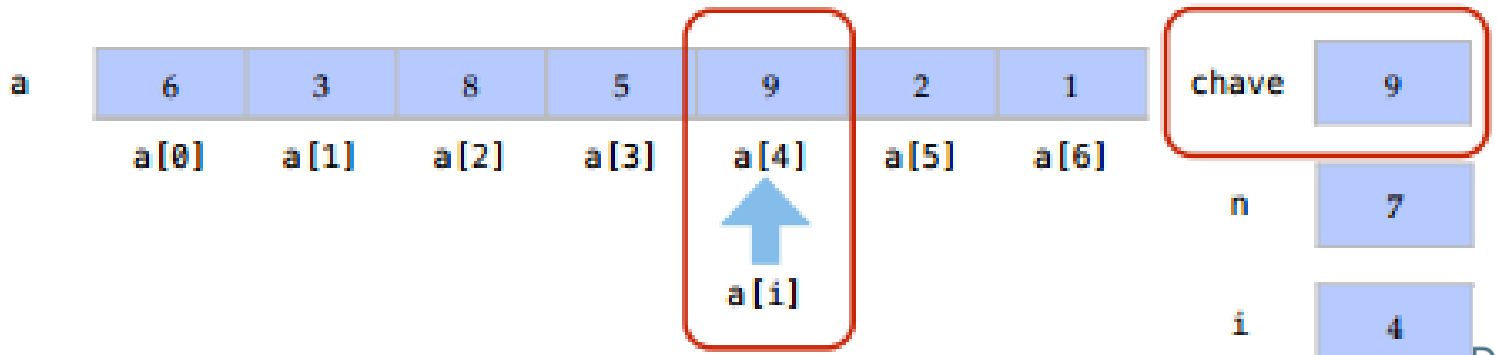
```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```



Busca Sequencial

- Comparamos cada elemento do arranjo com o elemento chave

```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```

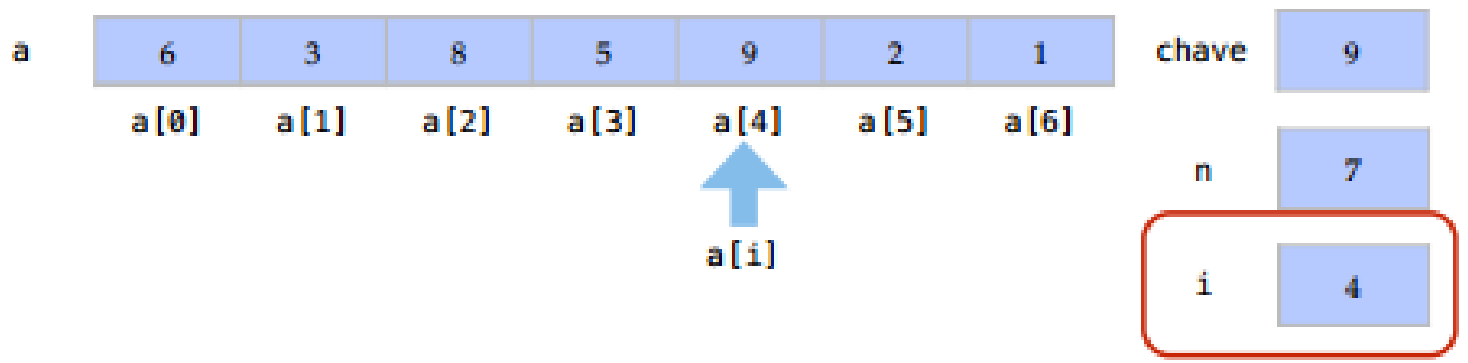


Busca Sequencial

- Se o elemento chave foi encontrado, o valor de i, que tem sua posição nesta iteração da repetição, é retornado

```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```

true

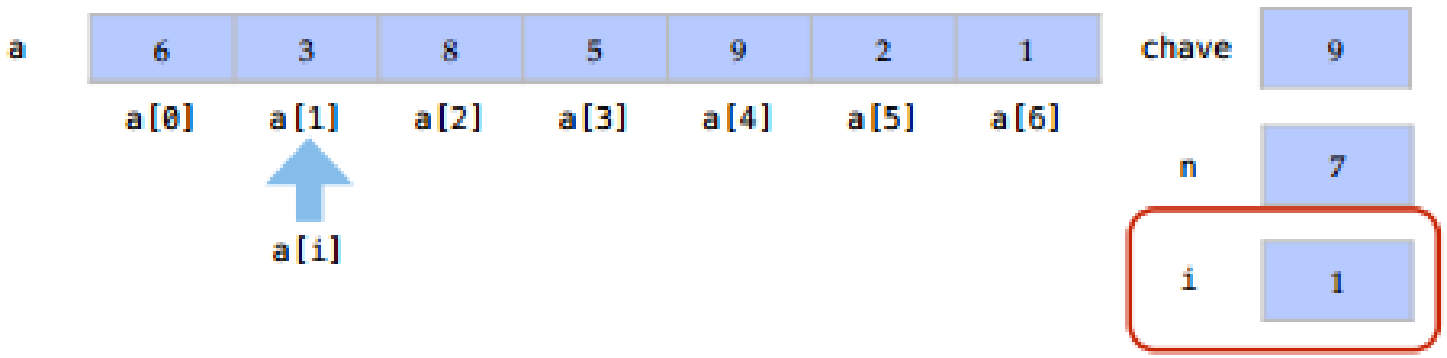


Busca Sequencial

- Se o elemento chave não foi encontrado e a comparação deu false, passamos para a próxima iteração de repetição

```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```

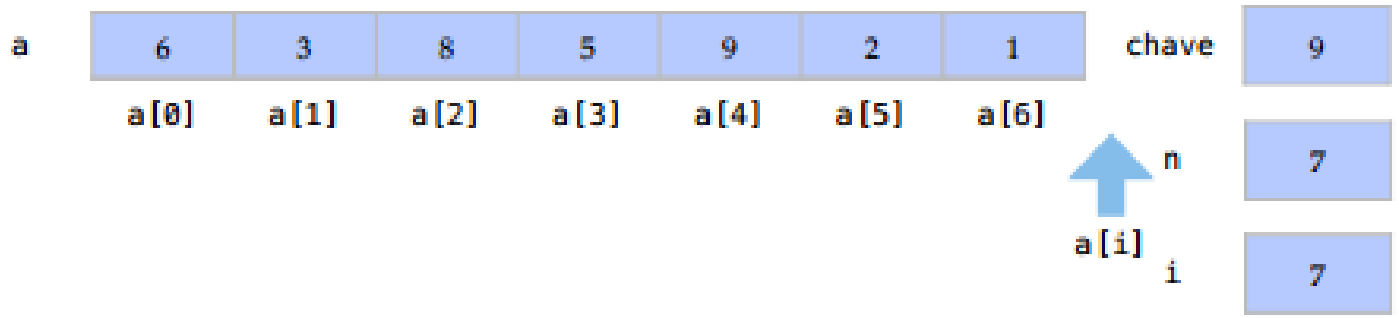
false



Busca Sequencial

- Se em nenhuma das iterações o elemento foi encontrado, saímos do for e retornamos -1. O retorno de -1 avisa para a função chamadora que o elemento não foi encontrado

```
int buscaSequencial(int chave, int a[], int n){
    for (int i=0; i<n; i++)
    {
        if (a[i] == chave)
        {
            return i;
        }
    }
    return -1;
}
```



Busca Sequencial - Análise

- Em relação ao número de comparações:
  - **Melhor caso:** Quando o elemento que procuramos é o primeiro do arranjo
  - **Pior caso:** o elemento que procuramos é o último do arranjo ou não está no arranjo

Busca Sequencial - Análise

- Em relação ao número de comparações:
  - **Melhor caso:** $f(n) = 1$
  - **Pior caso:** $f(n) = n$
  - **Caso médio:** $f(n) = (n+1)/2$
    - Estas análises assumem que todas as buscas encontrarão o elemento
  - **Pesquisa sem sucesso:** $f(n) = n$

Busca Sequencial - Análise

- Em notação O, podemos dizer que em relação ao número de comparações, a busca sequencial é O(n) para todos os casos, com exceção do melhor caso, no qual o algoritmo é O(1);

Busca Binária

Busca Binária

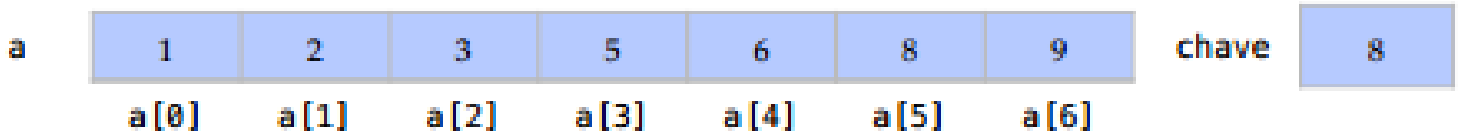
- É um método mais eficiente que a busca sequencial
- Porém, a busca binária requer que os elementos estejam ordenados

Busca Binária

- A cada iteração, pesquisamos o elemento do meio
  - Se a chave é maior, repetimos o processo na primeira metade
  - Senão, repetimos este passo na segunda metade
- A busca binária é um método interessante pois remete a maneira como buscamos palavras em um dicionário

Busca Binária

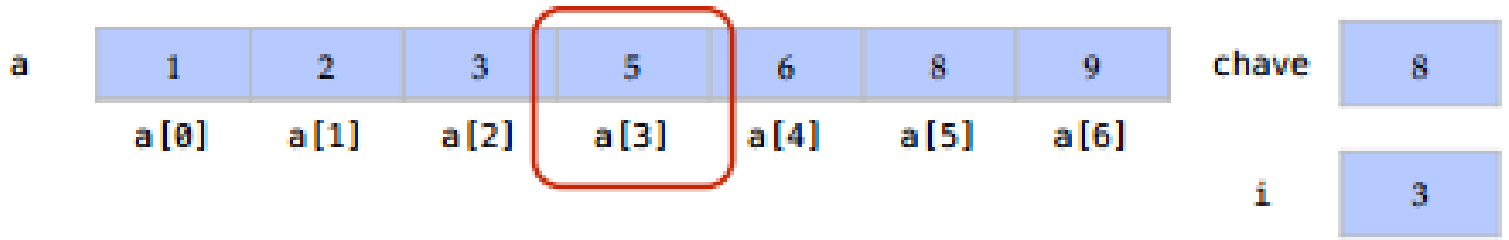
- Procuraremos o elemento 8 no arranjo a



Busca Binária

- Comparamos a chave com o elemento do meio 6/2=3

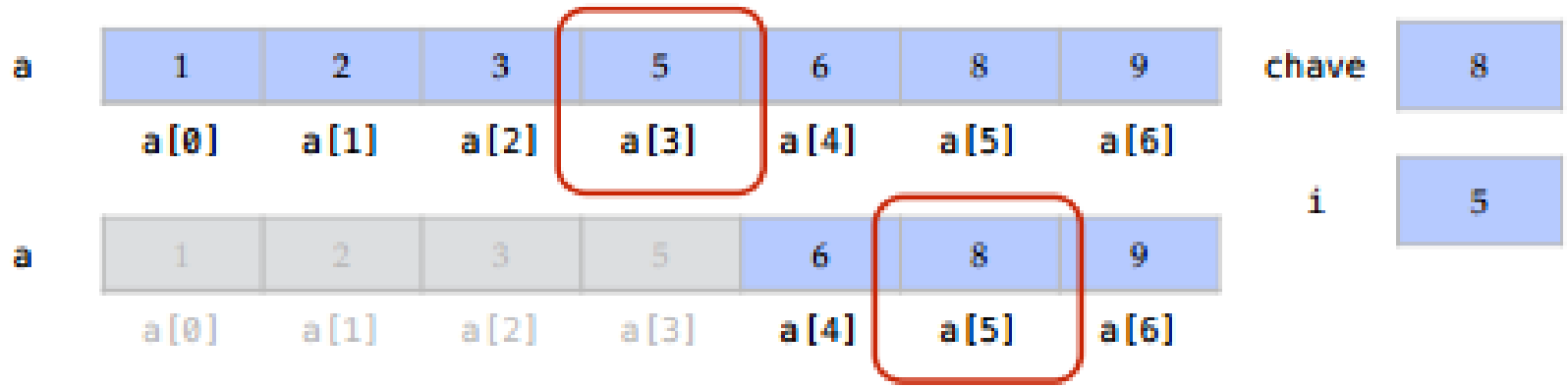
- Como a chave é maior que a[3] ou 5, sabemos agora que só precisamos procurar do lado direito do arranjo



Busca Binária

- Desconsideramos então os elementos de a[0] a a[3] do arranjo e repetimos o procedimento.

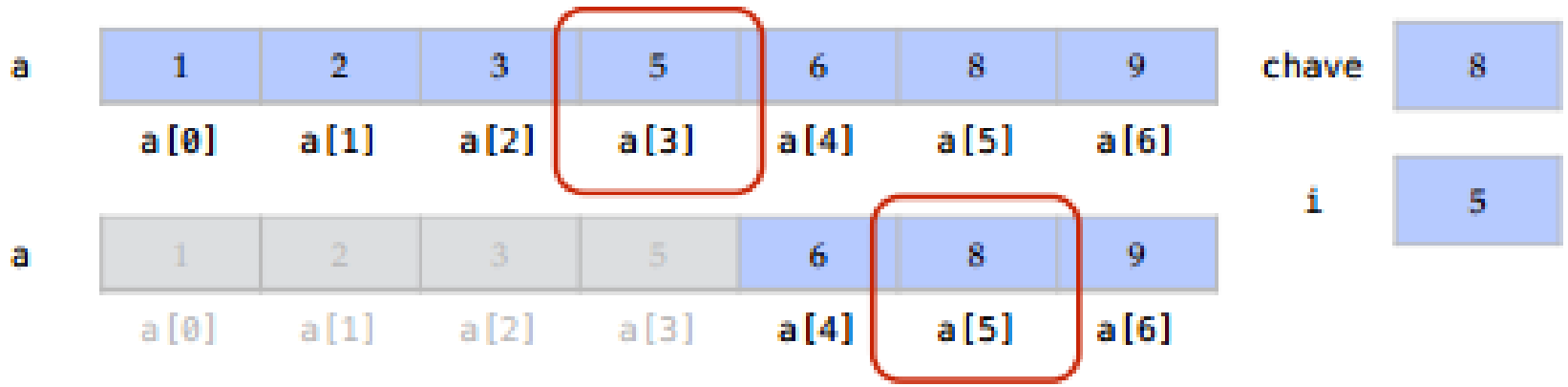
- O elemento do meio entre os que sobraram agora é a[5]



Busca Binária

- Como o elemento do meio agora é igual à chave, o algoritmo retorna o valor de i

- Com a busca binária, já encontramos o elemento na segunda comparação



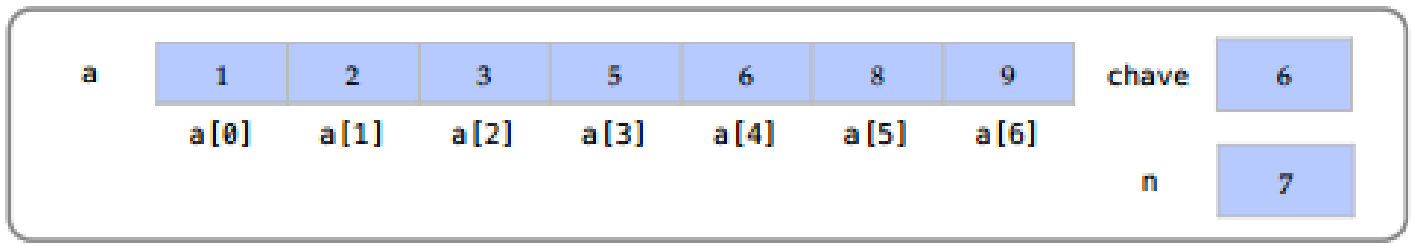
Busca Binária

```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```

Busca Binária

- Similarmente à busca sequencial, a função procura o elemento chave, neste exemplo o valor 6, no arranjo a, que tem tamanho n

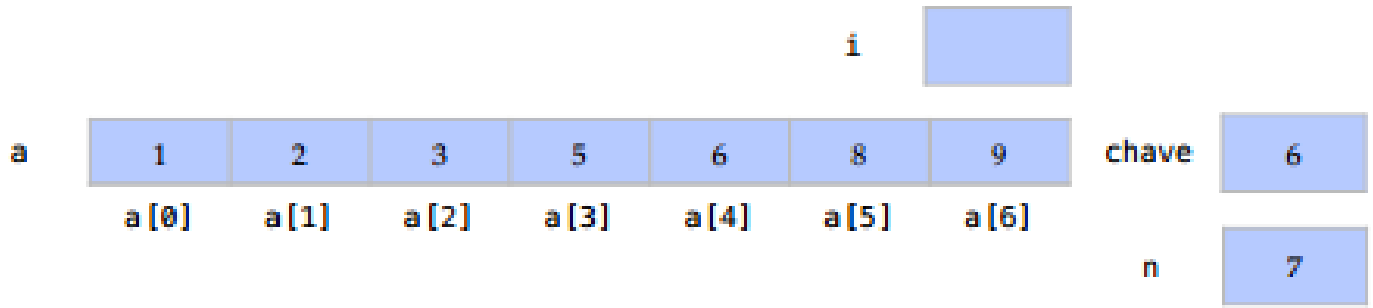
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- O índice i marcará o elemento sendo comparado, assim como no for da busca sequencial

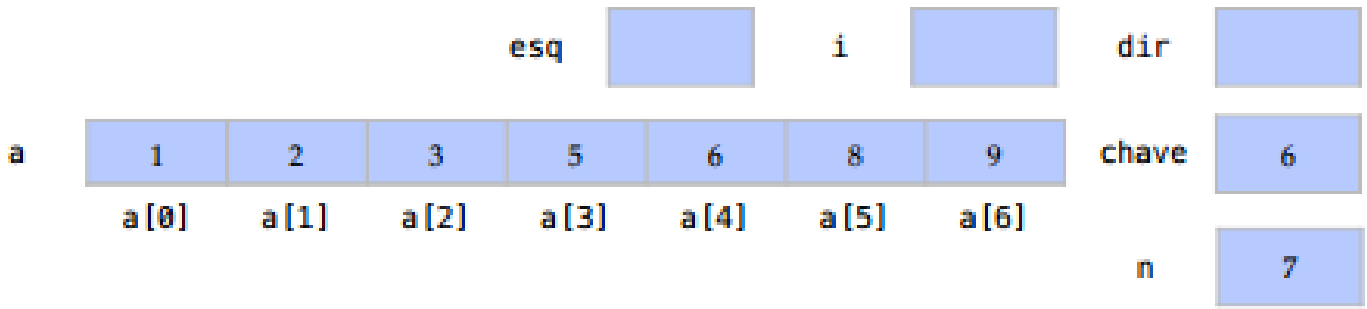
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Os índices esq e dir marcarão os limites de onde a busca está sendo feita

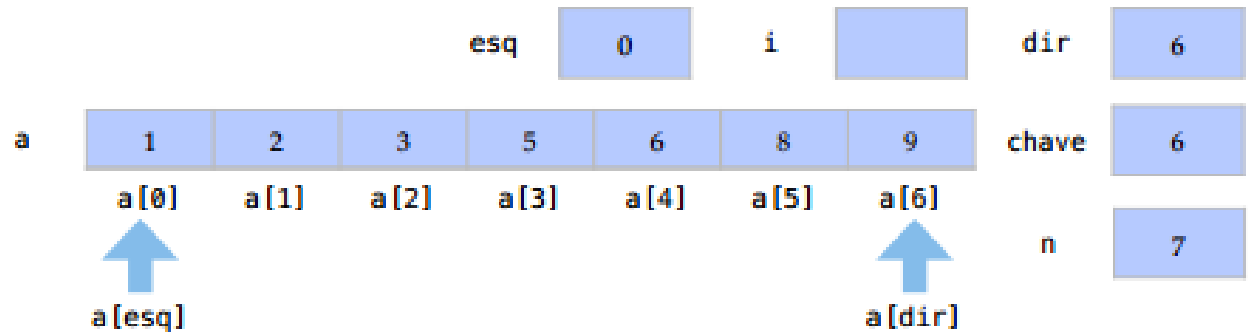
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Inicialmente, a busca será feita entre os elementos a[0] e a[n-1], ou a[6]

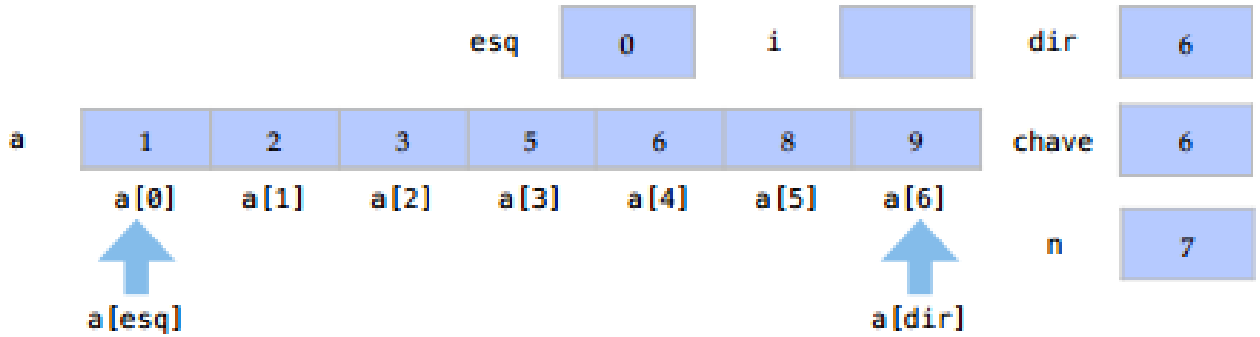
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Estamos em uma estrutura de repetição onde a cada passo um elemento i será comparado

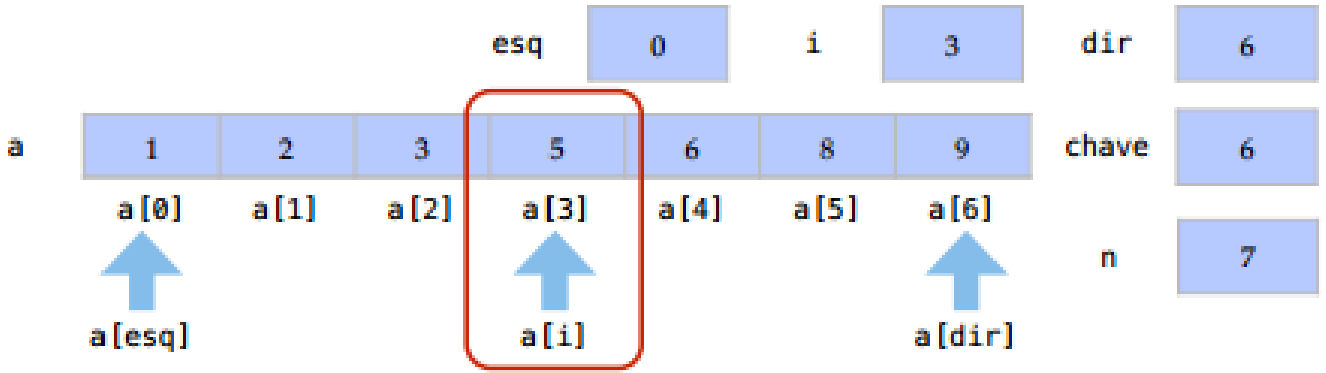
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- O índice i aponta então para o elemento do meio do arranjo

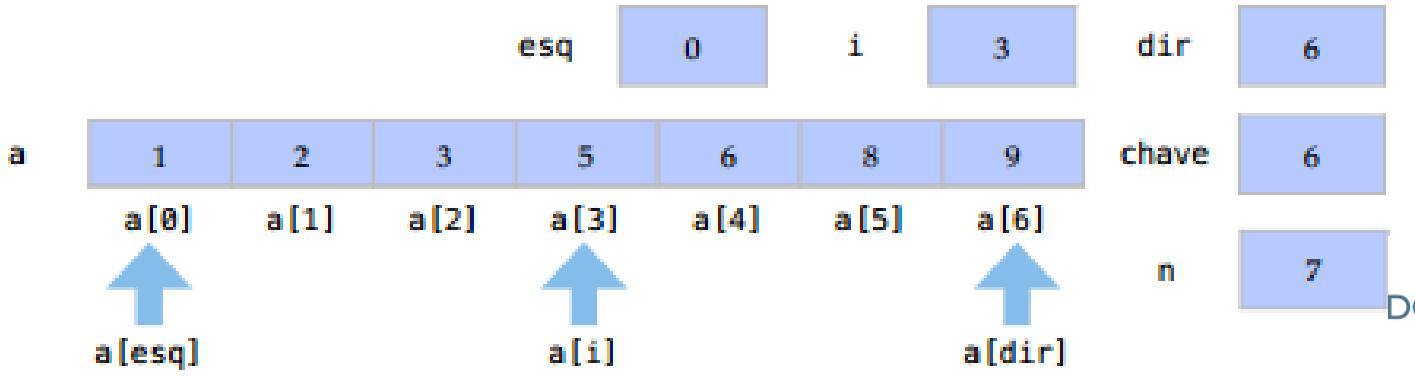
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- A chave 6, é maior que o elemento do meio 5

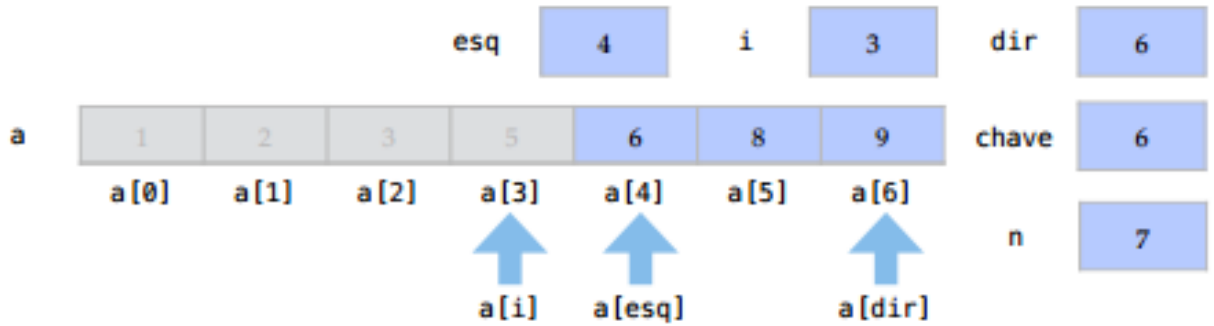
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Por isto, pesquisaremos agora apenas na metade à direita de i e, a partir de agora, a metade à esquerda será desconsiderada pelo algoritmo

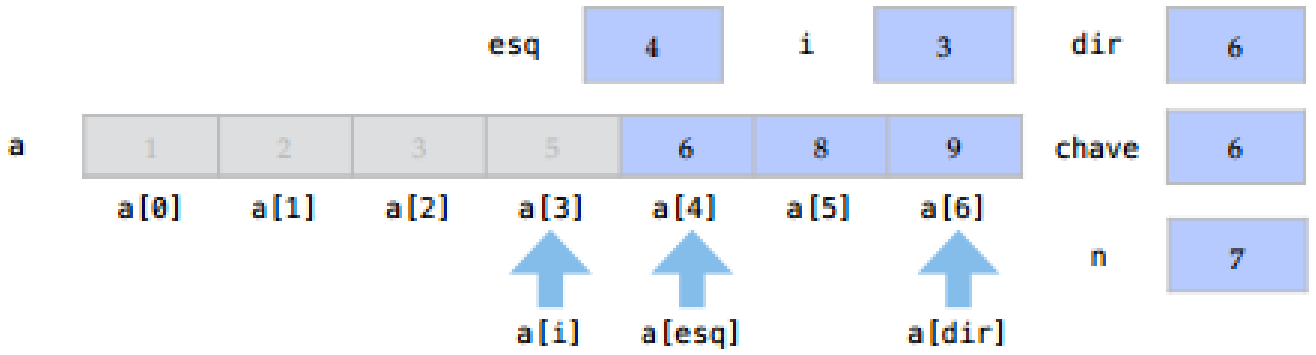
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Se a chave fosse menor que a[i], o contrário ocorreria, e a metade à direita passaria a ser desconsiderada

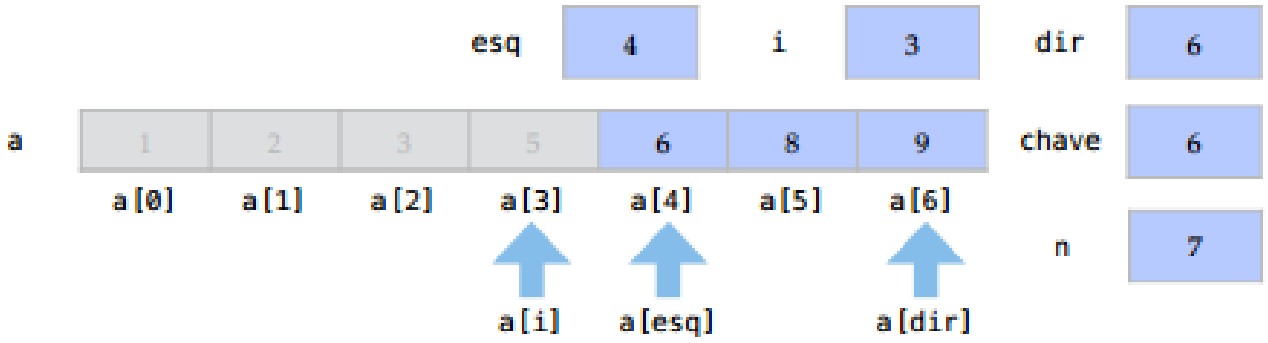
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Se a chave não é o elemento a[i], a repetição deve continuar pois o elemento i não foi encontrado

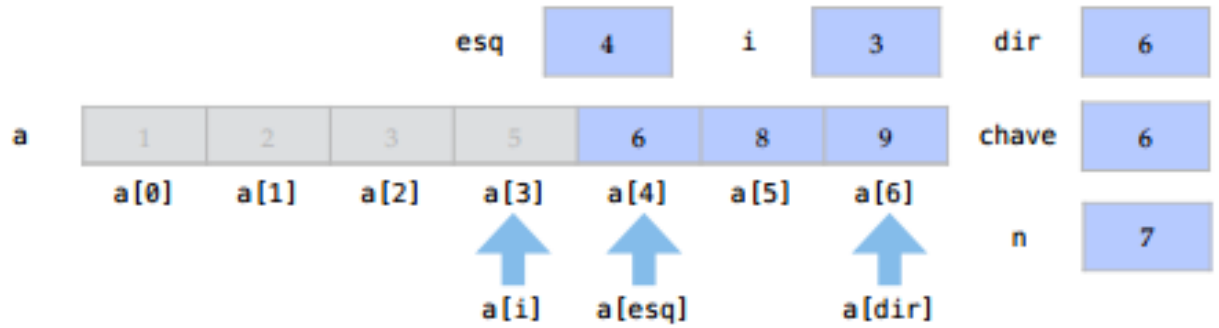
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Se o índice esq é menor ou igual a dir, a repetição deve continuar pois isto indica que o arranjo inteiro não foi pesquisado

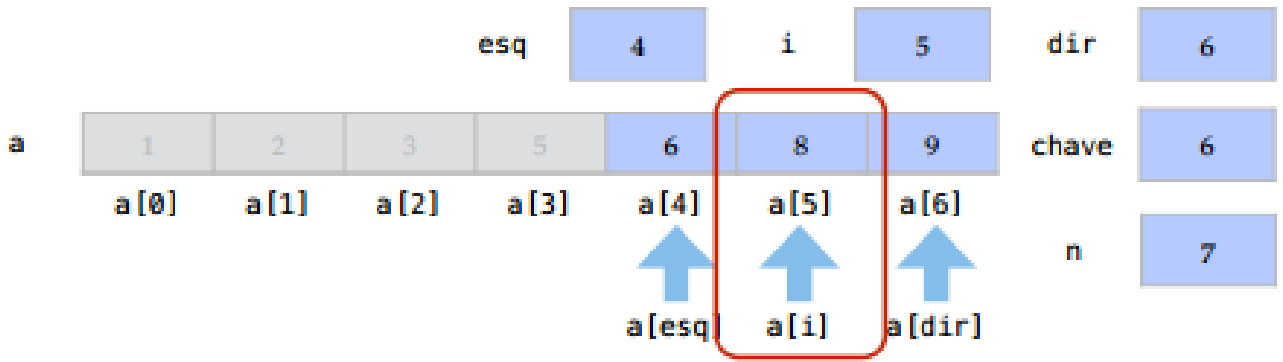
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- O índice i marca o elemento do meio entre os que ainda estão sendo considerados

```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

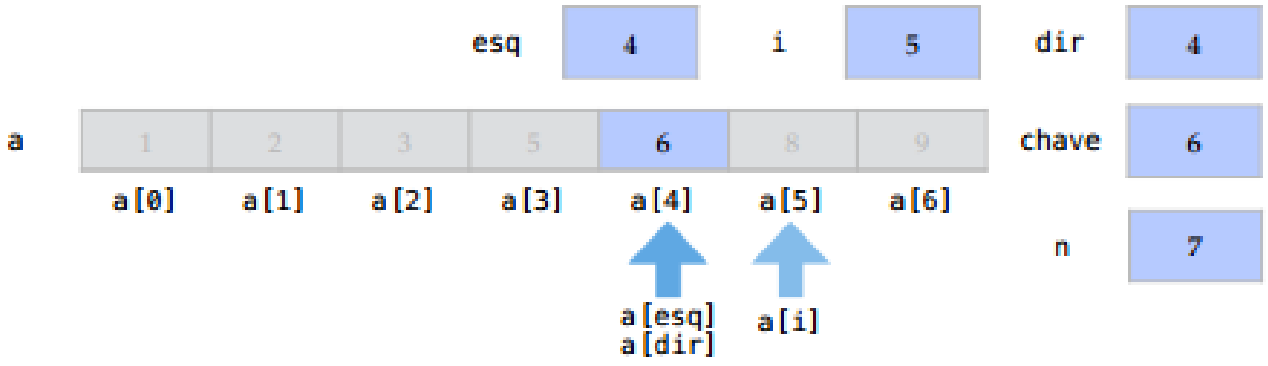
- Como a[5] (ou 8) é maior que a chave 6 que procuramos, alteramos dir para restringir a busca ao lado esquerdo do arranjo

```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```

Busca Binária

- O elemento a[i] ainda não é a chave que procuramos e os índices esq e dir não se cruzaram indicando que todo o arranjo já foi percorrido

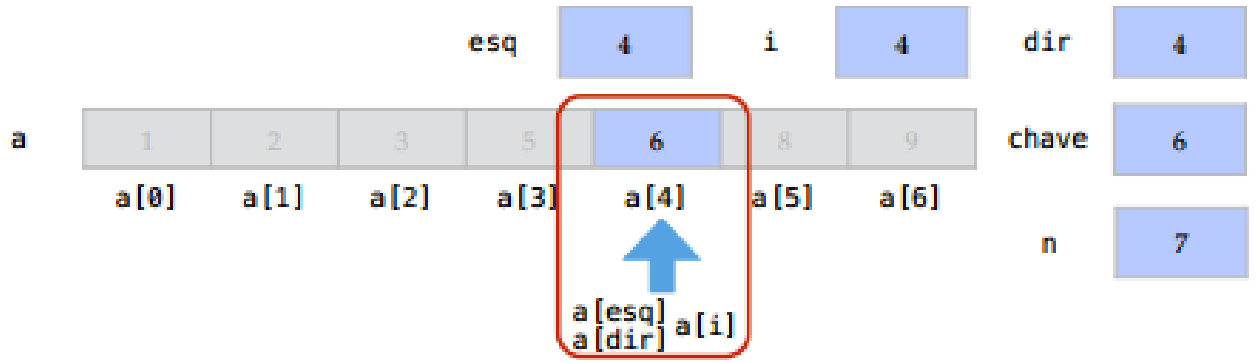
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Na próxima iteração, o elemento do meio marcado por i é o único elemento ainda não considerado

```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Como a chave não é maior que a[i] (ela é igual), deslocamos o índice dir mais uma vez para indicar que não há mais elementos a se pesquisar

```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```

Busca Binária

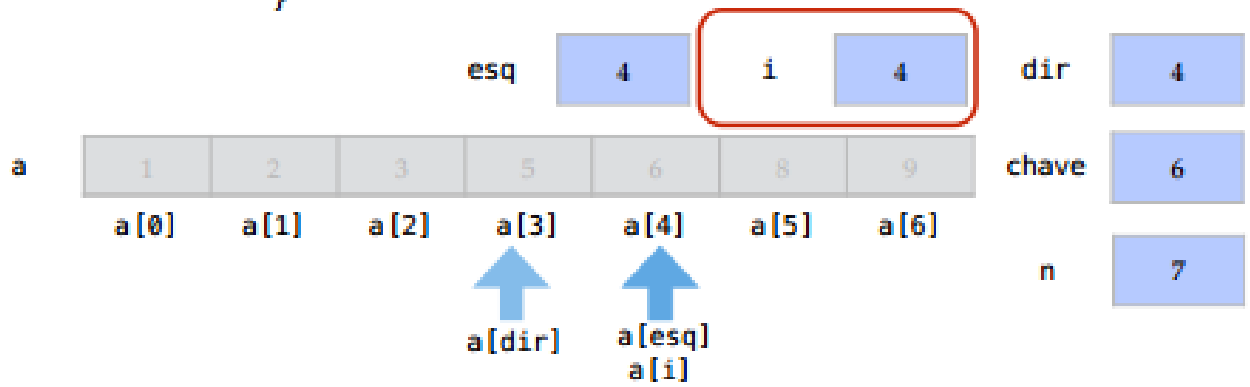
- Como a chave foi encontrada e não há mais elementos para se pesquisar, nenhuma das condições para se continuar o laço é verdadeira

```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```

Busca Binária

- Como o elemento i que pesquisamos é a chave, retornamos este índice

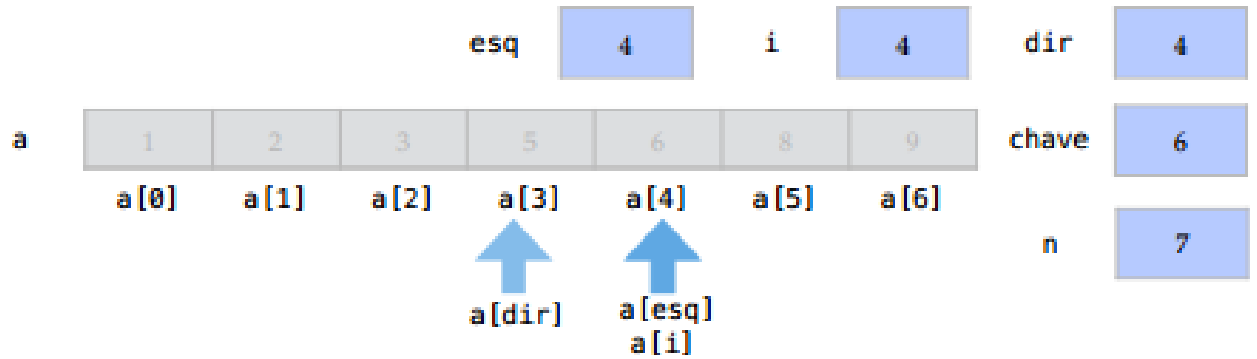
```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária

- Se o elemento i não fosse a chave, seria porque o elemento não estava no arranjo. Deveríamos então retornar -1

```
int buscaBinaria(int chave, int a[], int n){
    int i;
    int esq = 0;
    int dir = n-1;
    do {
        i = (esq + dir)/2;
        if (chave > a[i]){
            esq = i + 1;
        } else {
            dir = i - 1;
        }
    } while (chave != a[i] && esq <= dir);
    if (chave == a[i]){
        return i;
    } else {
        return -1;
    }
}
```



Busca Binária - Análise

- Em relação ao número de comparações temos um cenário similar à busca sequencial:
  - **Melhor caso:** Quando o elemento que procuramos é o primeiro que testamos (meio arranjo)
  - **Pior caso:** o elemento que procuramos é o último que comparamos ou quando o elemento não está no arranjo

Busca Binária - Análise

- Em relação ao número de comparações:
  - **Melhor caso:** $f(n) = O(1)$
  - **Pior caso:** $f(n) = O(\log n)$
  - **Caso médio:** $f(n) = O(\log n)$

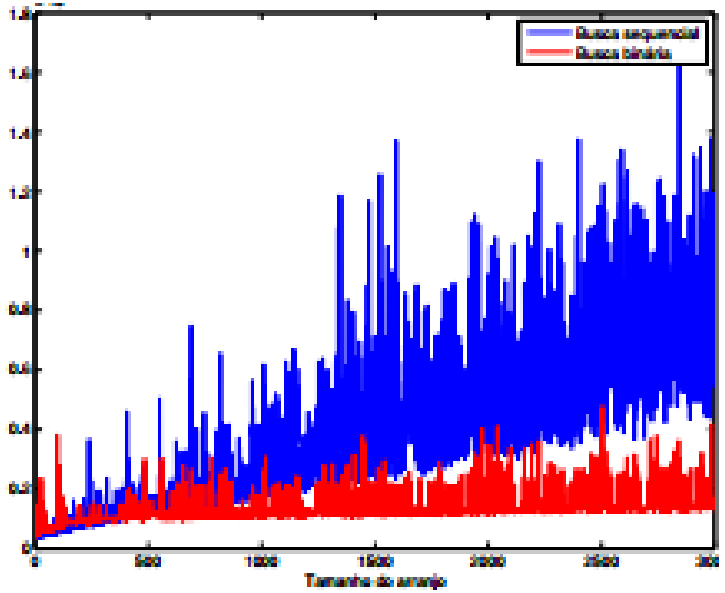
Busca Binária - Análise

- Em notação O, mesmo os piores casos são O(log n) pois a cada passo o algoritmo elimina metade do problema
  - $n \rightarrow n/2 \rightarrow n/2^2 \rightarrow n/2^3 \rightarrow \ldots \rightarrow 1$
    - ($n/2^k = 1 \rightarrow k = \log n$ passos)
  - Isto o deixa muito mais eficiente em relação à busca sequencial O(n)
    - Se n = 1.000.000, log n = 6

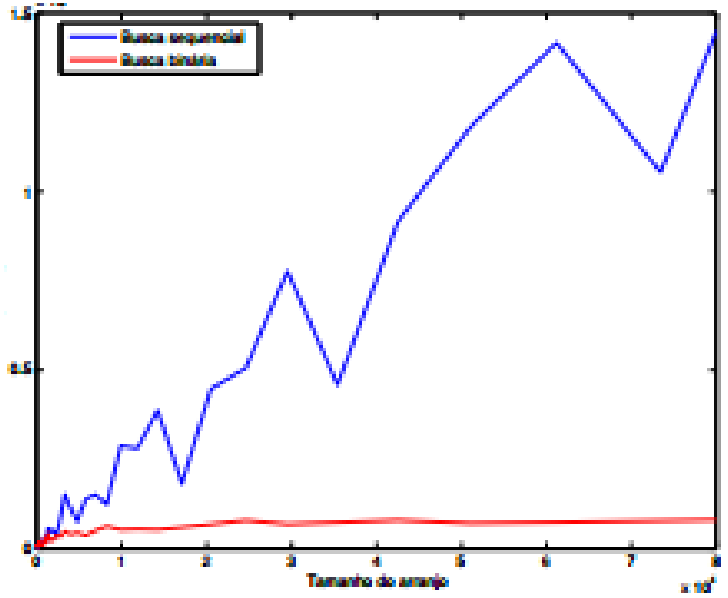
Busca Binária - Análise

- Uma desvantagem é que manter o arranjo ordenado pode ter um custo muito alto
  - Isto torna este método mais vantajoso para aplicações pouco dinâmicas
    - Este é justamente o caso de dicionários

Busca Binária - Experimentos



(a) Arranjos menores



(b) Arranjos maiores

Algoritmos e Estruturas de Dados II

- Bibliografia:
  - Básica:
    - CORMEN, Thomas, RIVEST, Ronald, STEIN, Clifford, LEISERSON, Charles. Algoritmos. Rio de Janeiro: Elsevier, 2002.
    - EDELWEISS, Nina, GALANTE, Renata. Estruturas de dados. Porto Alegre: Bookman. 2009. (Série livros didáticos informática UFRGS,18).
    - ZIVIANI, Nívio. Projeto de algoritmos com implementação em Pascal e C. São Paulo: Cengage Learning, 2010.
  - Complementar:
    - ASCENCIO, Ana C. G. Estrutura de dados. São Paulo: Pearson, 2011. ISBN: 9788576058816.
    - PINTO, W.S. Introdução ao desenvolvimento de algoritmos e estrutura de dados. São Paulo: Érica, 1990.
    - PREISS, Bruno. Estruturas de dados e algoritmos. Rio de Janeiro: Campus, 2000.
    - TENEMBAUM. Aaron M. Estruturas de dados usando C. São Paulo: Makron Books. 1995. 884 p. ISBN: 8534603480.
    - VELOSO, Paulo A. S. Complexidade de algoritmos: análise, projeto e métodos. Porto Alegre, RS: Sagra Luzzatto, 2001

Algoritmos e Estruturas de Dados II