



Mestrado Profissional em Matemática, Estatística e Computação Aplicado à Indústria  
(MECAI)

---

Trabalho de MAI5001 - Introdução a Ciências da Computação

---

## ***Análise e Desempenho dos Algoritmos de Busca***

**Prof.Dr.: Adenilso Simões**

**Aluno: João Carlos Batista**  
**NºUSP: 6792197**

**USP – São Carlos 19/06/2017**

# Sumário

<b>LISTA DE FIGURAS.....</b>	<b>III</b>
<b>LISTA DE TABELAS.....</b>	<b>IV</b>
<b>CAPÍTULO 1: BUSCA SEQUENCIAL COM SENTINELA .....</b>	<b>1</b>
1.1. DESCRIÇÃO DO ALGORITMO .....	1
1.1.1 CARACTERÍSTICAS.....	1
1.1.2 CODIFICAÇÃO EM C# .....	1
1.1.3 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	2
1.2 MELHOR CASO.....	2
1.3 PIOR CASO.....	2
<b>CAPÍTULO 2: BUSCA SEQUENCIAL SEM SENTINELA .....</b>	<b>4</b>
2.1. DESCRIÇÃO DO ALGORITMO .....	4
2.1.1. CARACTERÍSTICAS.....	4
2.1.2. CODIFICAÇÃO EM C# .....	5
2.1.5 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	5
2.2. MELHOR CASO.....	5
2.3. PIOR CASO.....	6
<b>CAPÍTULO 3: BUSCA SEQUENCIAL ORDENADO .....</b>	<b>7</b>
3.1. DESCRIÇÃO DO ALGORITMO .....	7
3.1.1 CARACTERÍSTICAS.....	7
3.1.2 CODIFICAÇÃO EM C# .....	7
3.1.3 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	8
3.2. MELHOR CASO.....	8
3.3. PIOR CASO.....	8
<b>CAPÍTULO 4: BUSCA BINÁRIA .....</b>	<b>10</b>
4.1. DESCRIÇÃO DO ALGORITMO .....	10
4.1.1 CARACTERÍSTICAS.....	10
4.1.2 CODIFICAÇÃO EM C# .....	10
4.1.3 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	11

4.2. MELHOR CASO.....	11
4.3. PIOR CASO.....	12
<b>CAPÍTULO 5: BUSCA INTERPOLADA .....</b>	<b>13</b>
5.1. DESCRIÇÃO DO ALGORITMO.....	13
5.1.1 CARACTERÍSTICAS.....	13
5.1.2 CODIFICAÇÃO EM C#.....	13
5.1.2 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	14
5.2. MELHOR CASO.....	14
5.3. PIOR CASO.....	15
5.4. ANÁLISE DE DESEMPENHO DA BUSCA INTERPOLADA VERSOS BUSCA BINÁRIA.....	16
<b>CAPÍTULO 6: BUSCA, INSERÇÃO E REMOÇÃO EM ÁRVORES AVL.....</b>	<b>18</b>
6.1. DESCRIÇÃO DO ALGORITMO.....	18
6.1.1 CARACTERÍSTICAS.....	19
6.1.2 CODIFICAÇÃO EM C#.....	19
6.1.2 COMPLEXIDADE DE TEMPO E DE ESPAÇO.....	23
6.2. INSERÇÃO EM ÁRVORE AVL.....	23
6.3. BUSCA EM ÁRVORE AVL.....	25
6.4. REMOÇÃO EM ÁRVORE AVL.....	25
6.5. ANÁLISE DAS OPERAÇÕES NA ÁRVORE AVL.....	27
<b>REFERÊNCIAS.....</b>	<b>29</b>
<b>ANEXO: PROJETO EM C# .....</b>	<b>29</b>

# Lista de Figuras

FIGURA 1: DADOS NÃO UNIFORMEMENTE DISTRIBUÍDOS .....	16
FIGURA 2:TESTE 1 COM DADOS NÃO UNIFORMEMENTE DISTRIBUÍDOS .....	16
FIGURA 3:DESEMPENHO NA BUSCA TESTE 1 .....	17
FIGURA 4:TESTE 2 COM DADOS UNIFORMEMENTE DISTRIBUÍDOS .....	17
FIGURA 5: DESEMPENHO NA BUSCA TESTE 2 .....	18
FIGURA 6: DESEMPENHO ÁRVORE AVL .....	27

# Lista de Tabelas

TABELA 1 COMPLEXIDADE DE TEMPO E DE ESPAÇO. BUSCA SEQUENCIAL COM SENTINELA.....	2
TABELA 2 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO BUSCA SEQUENCIAL SEM SENTINELA.....	5
TABELA 3 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO BUSCA SEQUENCIAL ORDENADA.....	8
TABELA 4 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO BUSCA BINARIA .....	11
TABELA 5 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO BUSCA INTERPOLADA.....	14
TABELA 6 COMPLEXIDADE DE TEMPO E DE ESPAÇO. ALGORITMO ARVORE AVL .....	23

# CAPÍTULO 1: Busca sequencial com sentinela

## 1.1. Descrição do algoritmo

A busca sequencial com sentinela é viável quando não sabemos se o elemento procurado está no vetor, por esse motivo é necessário colocar uma sentinela no final do vetor para atuar como condição de parada. O algoritmo procura sequencialmente o elemento num vetor não ordenado, quando acha finaliza a busca. Se tivermos um vetor com  $n$  elementos não ordenado, adiciona-se a sentinela na última posição como critério de parada caso não encontre o elemento.

### 1.1.1 Características

- É fácil de implementar, com uso de sentinela o algoritmo ocorre em tempo linear
- É recomendado o usar em pequena e média quantidade de dados.

### 1.1.2 Codificação em C#

```
static public void busca_com_sentinela(int[] vector, int procurado, int sentinela)
{
    int posicao = 0;
    for (int i = 0; vector[i] != sentinela; i++) // n
    {
        if (vector[i] == procurado)
        {
            WriteLine($"chave encontrada " + vector[i] + " posicao " + i);
            break;
        }
        posicao = i;
    }

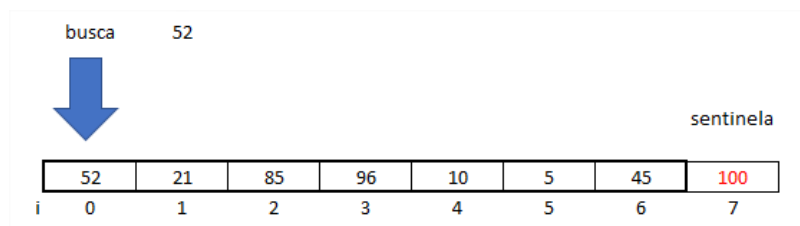
    if (vector[posicao + 1] == sentinela)
    {
        WriteLine($" chave nao encontrada "+ procurado+" chegou no sentinela "+ vector[posicao + 1]);
    }
}
```

### 1.1.3 Complexidade de tempo e de espaço

*Tabela 1 Complexidade de tempo e de espaço. Busca sequencial com sentinela*

Melhor caso	1	→	$O(1)$
Caso médio	$\frac{n+1}{2}$	→	$O(n)$
Pior caso	$n + 1$	→	$O(n)$
Complexidade de espaços		→	$O(1)$

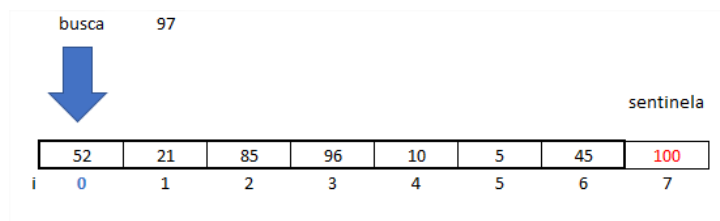
### 1.2 Melhor Caso



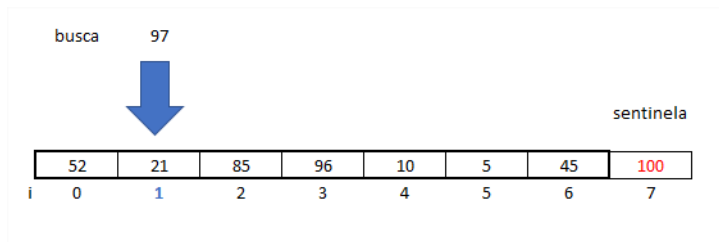
Melhor caso quando o elemento a ser buscado está na primeira posição

### 1.3 Pior Caso

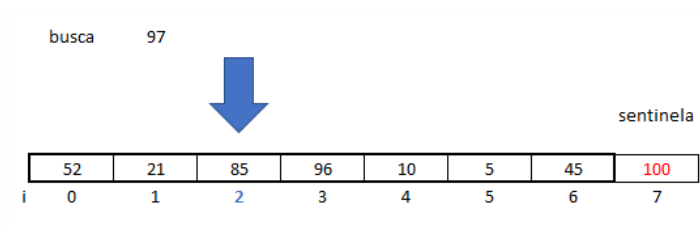
Iteração 1



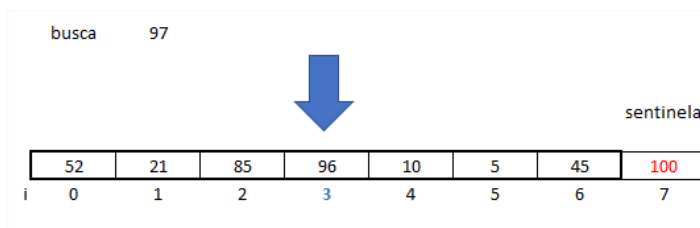
Iteração 2



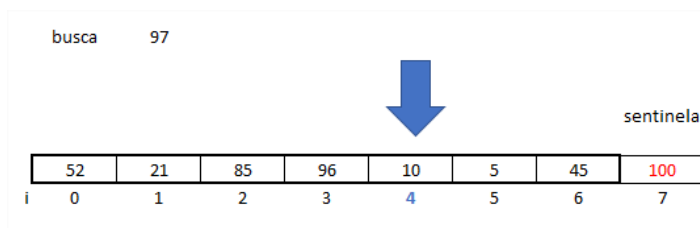
Iteração 3



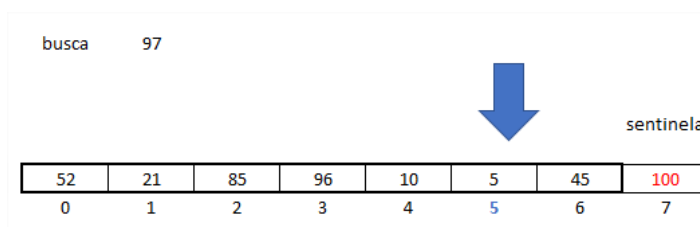
Iteração 4



Iteração 5

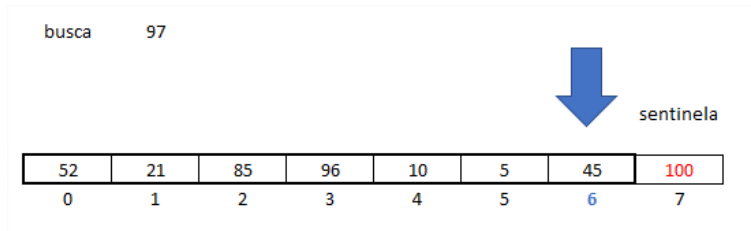


Iteração 6

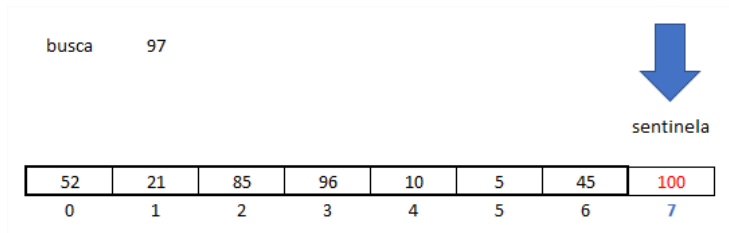


Iteração 7





Iteração 8



O elemento não está na lista.

## CAPÍTULO 2: Busca sequencial sem sentinela

### 2.1. Descrição do algoritmo

O algoritmo realiza busca sequencial nos  $n$  elementos não ordenado, faz comparação a cada registro para verificar se é o elemento procurado.

#### 2.1.1.Características

- É fácil de implementar, sem uso de sentinela o algoritmo ocorre em tempo linear
- É recomendado o usar em pequena e média quantidade de dados, mostra valores repetidos de chaves quando existe.

```
static public void busca_sem_sentinela(int[] vector, int procurado)
{
    int indica_achou = 0;
    int posicao_do_vetor = 0;
    for (int i = 0; i < vector.Length; i++)// n
    {
        if (vector[i] == procurado)
        {
            posicao_do_vetor = i + 1;
            indica_achou = 1;
        }
    }
}
```

```

        WriteLine($" achou " + procurado + " posicao " +
        posicao_do_vetor);
    }

}

if (indica_achou == 0)
{
    WriteLine($" nao encontrado " + procurado);
}
}

```

### 2.1.2. Codificação em C#

### 2.1.5 Complexidade de tempo e de espaço

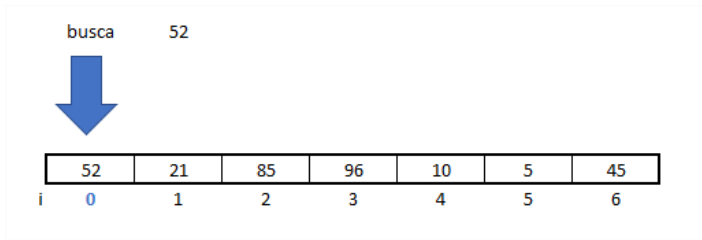
O número de comparações está entre 1 a  $n$  elementos até encontrar o elemento da busca

A complexidade em pior caso, caso médio e melhor caso e:

*Tabela 2 Complexidade de tempo e de espaço. Algoritmo Busca sequencial sem sentinela*

Melhor caso	1	→	O (1)
Caso médio	$\frac{n}{2}$	→	O (n)
Pior caso	$n$	→	O (n)

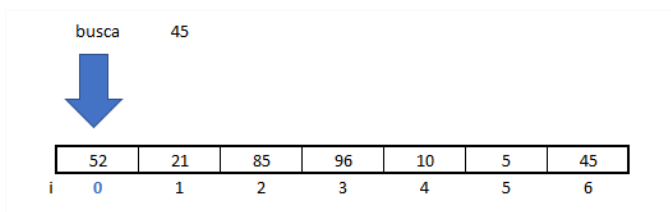
## 2.2. Melhor Caso



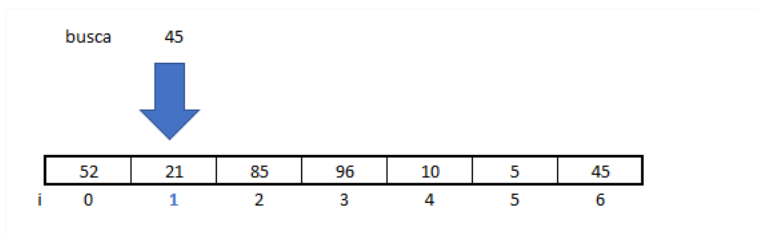
O elemento da busca está na primeira posição

## 2.3. Pior Caso

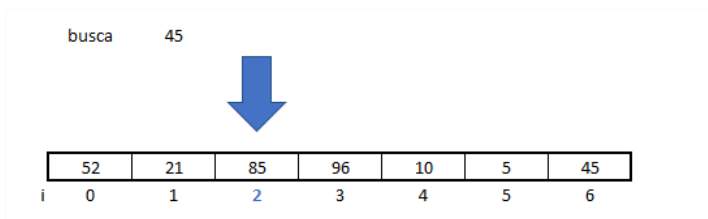
Iteração 1



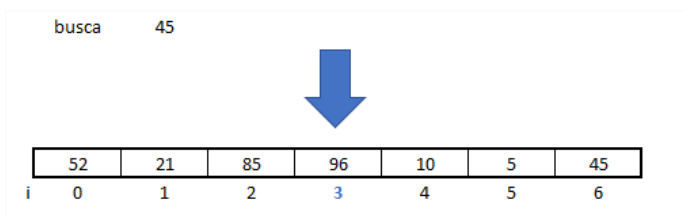
Iteração 2



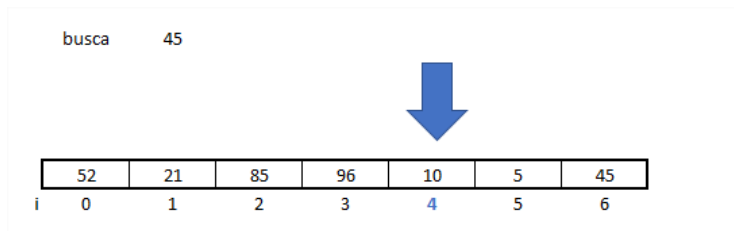
Iteração 3



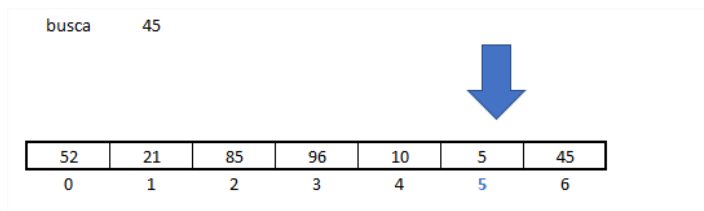
Iteração 4



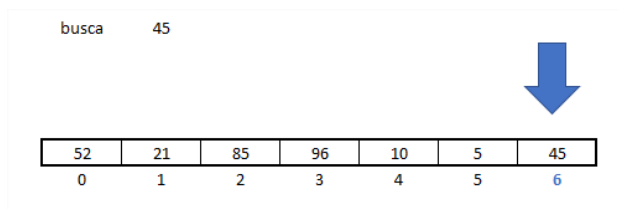
Iteração 5



Iteração 6



Iteração 7



## CAPÍTULO 3: Busca sequencial ordenado

### 3.1. Descrição do algoritmo

A busca sequencial é a forma mais simples de procurar um elemento num vetor, pois o algoritmo percorre o vetor para buscar a chave.

#### 3.1.1 Características

- É fácil de implementar
- Pode ser rápido se o elemento procura estiver próximo do início do vetor, e recomendado para problemas de dimensão média.

#### 3.1.2 Codificação em C#

```
static public void busca_sequencial_ordenado(int[] vector, int procurado)
```

```

{
    for (int i = 0; i < vector.Length; i++)// n-1
    {
        if (vector[i] == procurado)
        {
            WriteLine($" achou  " + vector[i] + " posição  " + i);
            i = vector.Length+1;// depois que achou, sai do for
        }
    }
}

```


### 3.1.3 Complexidade de tempo e de espaço

*Tabela 3 Complexidade de tempo e de espaço. Algoritmo Busca sequencial ordenada*

O número de comparações no pior caso	→	$n$
A complexidade em		
Pior caso	→	$O(n)$
Caso médio	→	$O(n/2)$
Melhor caso	→	$O(1)$

### 3.2. Melhor Caso


procurado = 5




i	0	1	2	3	4	5	6	7	8	9
	5	10	21	45	52	85	96	110	121	152

### 3.3. Pior Caso


procurado = 152




i	0	0	1	2	3	4	5	6	7	8	9
		5	10	21	45	52	85	96	110	121	152




i	1	0	1	2	3	4	5	6	7	8	9
		5	10	21	45	52	85	96	110	121	152




i	2	0	1	2	3	4	5	6	7	8	9
		5	10	21	45	52	85	96	110	121	152




i	3	0	1	2	3	4	5	6	7	8	9
		5	10	21	45	52	85	96	110	121	152




i	4	0	1	2	3	4	5	6	7	8	9
		5	10	21	45	52	85	96	110	121	152



i	5	0	1	2	3	4	5	6	7	8	9
		5	10	21	45	52	85	96	110	121	152



i	6	0	1	2	3	4	5	6	7	8	9
		5	10	21	45	52	85	96	110	121	152



i	7	0	1	2	3	4	5	6	7	8	9
		5	10	21	45	52	85	96	110	121	152

</

## CAPÍTULO 4: Busca binária

### 4.1. Descrição do algoritmo

A busca binária compara elemento com elemento dos meios dos vetores/sub-vetores menores. A pesquisa é direcionada se elemento for menor que a chave do meio de cada sub-vetor, pesquise na primeira metade do vetor caso contrário pesquise na segunda parte do vetor, este procedimento de subdivisão do tamanho do vetor ocorre até encontrarmos o elemento.

#### 4.1.1 Características

- A cada comparação reduz o tamanho do vetor pela metade
- É simples de implementar.
- É eficiente na busca porém o vetor precisa estar ordenado.

#### 4.1.2 Codificação em C#

```
static public void busca_Binaria( int[] vector, int procurado)
{
    int indice_menor, indice_meio, indice_maior;
    indice_menor = 0; indice_maior = vector.Length;
    while (indice_menor <= indice_maior)
    {
        indice_meio = (indice_menor + indice_maior) / 2;
```

```

        WriteLine($"indice do meio " + indice_meio);

        if (vector[indice_meio] == procurado)
        {
            WriteLine($" achou  " + vector[indice_meio] + " posicao  " + indice_meio);
            indice_menor = indice_maior + 1; /// condicao para sair do while
        }
        if (vector[indice_meio] < procurado)
        { indice_menor = indice_meio + 1; }
        else
        { indice_maior = indice_meio - 1; }
    }
}

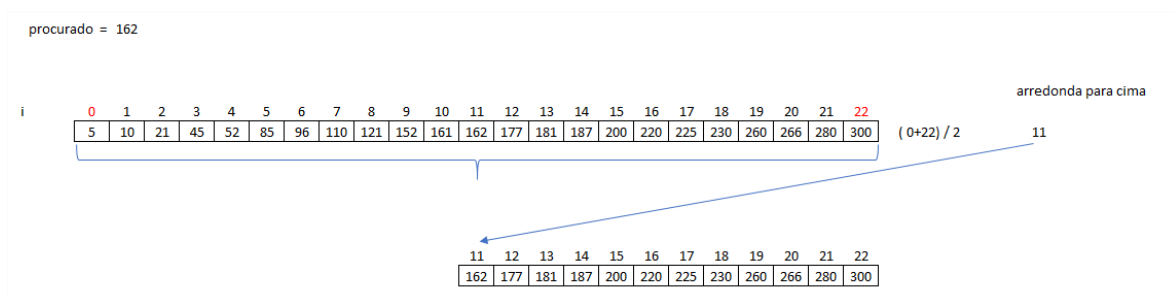
```

### 4.1.3 Complexidade de tempo e de espaço

Tabela 4 Complexidade de tempo e de espaço. Algoritmo busca binaria

Caso médio	→ $O(\log n)$
Melhor caso	→ $O(1)$
Pior caso	→ $O(\log n)$
Complexidade de espaços	→ $O(3)$

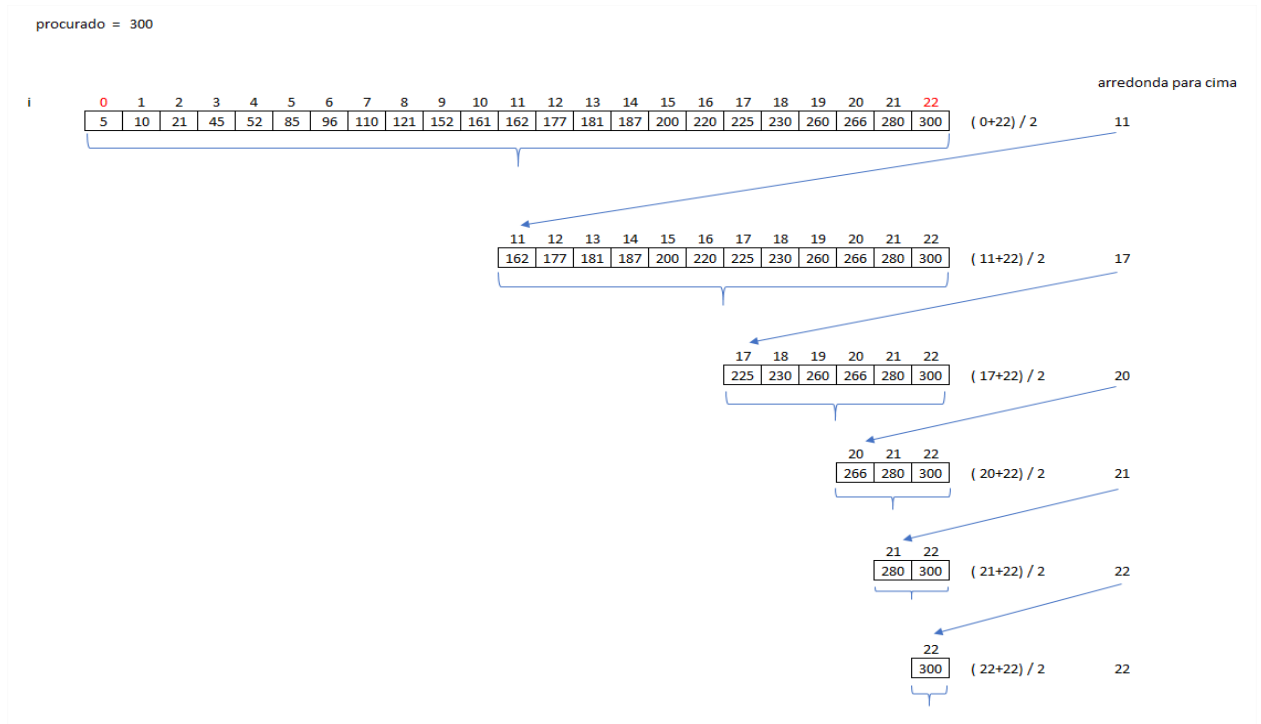
## 4.2. Melhor Caso



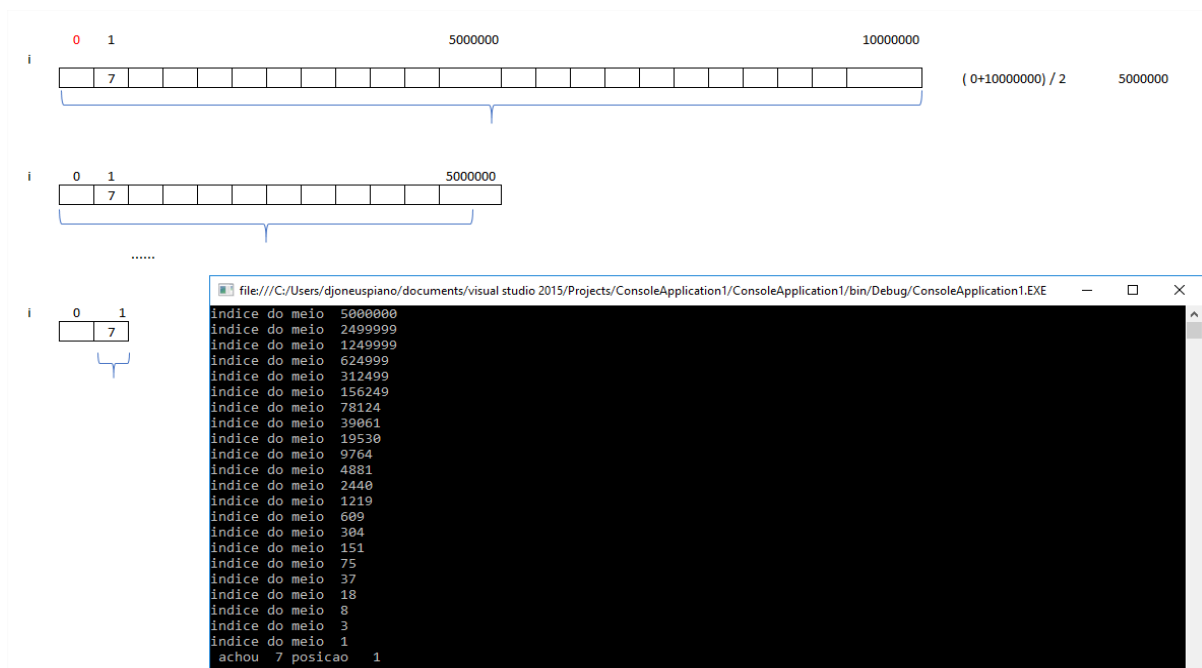
O melhor caso é quando o elemento está no meio.



## 4.3. Pior Caso



Teste realizado quando o elemento está na primeira posição. Nestes exemplos temos 10000000 elementos gerado de forma aleatória, com uso de semente (este mesmo teste pode ser repetido) e depois ordenado com o algoritmo QuickSort, identificamos o elemento na posição 1 que é o 7



# CAPÍTULO 5: Busca interpolada

## 5.1. Descrição do algoritmo

A busca por interpolação pode ser mais eficiente do que a busca binária, o que vai definir o maior desempenho é, se as chaves estiverem uniformemente distribuídas. A diferença é a formula implementada para definir o meio, está formula viabiliza convergir mais rápido para a chave procurada, a formula é:

```
meio=(inicio+((final-inicio)*(elemento_procurado-vetor[inicio]))/(vetor[final]- vetor[inicio]));
```

Esta busca e mais eficaz se as chaves estiverem uniformemente distribuídas, o número de comparações chega a  $\log(\log n)$ . Em algumas situações as chaves tentem a se aglomerar em torno de uns determinados valores, a qual torna o conjunto não uniformemente distribuídas, isso implica que a busca fica tão ruim ao ponto de ser comparável a uma busca sequencial.

### 5.1.1 Características

- A cada comparação reduz o tamanho do vetor para menos que a metade, isso e viável por causa da formula e da chave estar uniformemente distribuídas
- E simples de implementar.
- É eficiente a busca porem o vetor precisa estar ordenado e uniformemente distribuídos.

### 5.1.2 Codificação em C#

```
static public void Busca_Interpolada(int elemento_procurado, int[] vetor)
{
    int inicio = 0;
    int meio;
    int final = vetor.Length - 1;
    int achou = 0; // flag para indicar se encontrou o elemento no final da busca

    while (inicio < final)
    {
        meio=(inicio+((final-inicio)*(elemento_procurado-vetor[inicio]))/(vetor[final]-
vetor[inicio]));
    }
}
```

```

        WriteLine($" meio " + meio);

        if (elemento_procurado < vetor[meio])
        {
            final = meio - 1;
        }
        else if (elemento_procurado > vetor[meio])
        {
            inicio = meio + 1;
        }
        else
        {
            WriteLine($" achou " + vetor[meio] + " posicao " + meio);
            inicio = final + 1;
            achou = 1;
        }
    }

    if(achou==0)
    { WriteLine($" nao encontrado ");    }
}

```

### 5.1.2 Complexidade de tempo e de espaço

*Tabela 5 Complexidade de tempo e de espaço. Algoritmo busca interpolada*

Caso médio	→ $O((\log(\log n))/2)$
Melhor caso	→ $O(1)$
Pior caso	→ $O(\log(\log n))$
Complexidade de espaços	→ $O(3)$

## 5.2. Melhor Caso

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
vector	15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255	270	285	300	315	330

Para este vetor que está com as chaves uniformemente distribuídos a busca é de ordem 1, para buscar qualquer chave contida no vetor.

### 5.3. Pior Caso

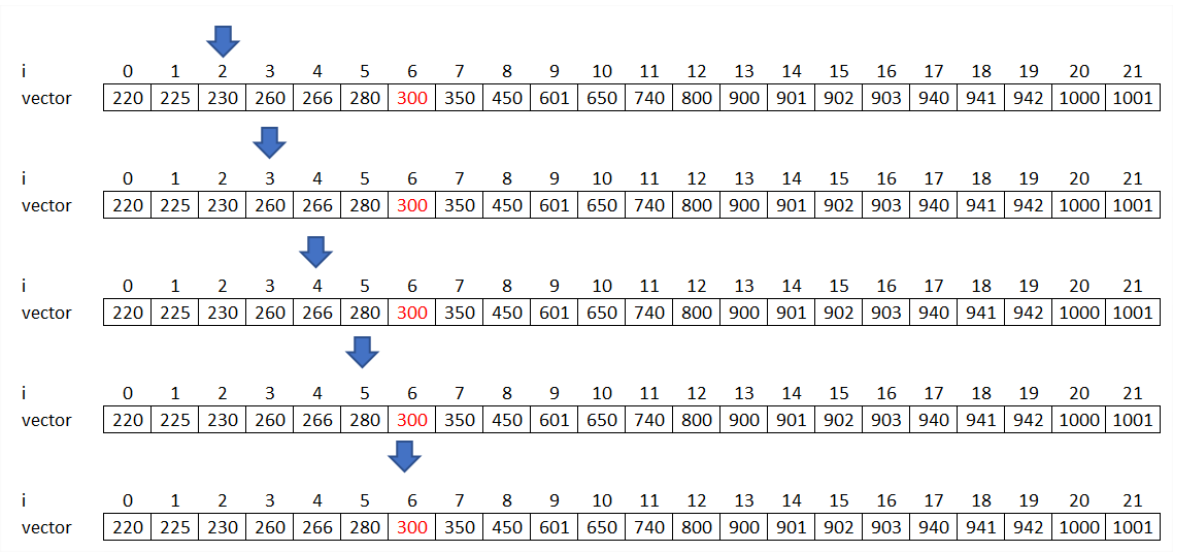
Vetor de busca não uniformemente distribuídos:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
vector	220	225	230	260	266	280	300	350	450	601	650	740	800	900	901	902	903	940	941	942	1000	1001

Procurar a chave 300, análise do algoritmo.

```
----- Busca Interpolada -----
meio  2
meio  3
meio  4
meio  5
meio  6
achou 300 posicao 6 n° de comparacoes 5
```

Seguiu está sequência:



Representação dos dados no vetor:



Figura 1: Dados não uniformemente distribuídos

## 5.4. Análise de desempenho da busca interpolada versus busca binária

Análise de desempenho em busca binária e busca interpolada, os dados não uniformemente distribuído neste primeiro teste, os números de elementos no vetor é 50

```
int[] vector10 = new int[]
```

```
{5,10,21,45,52,85,96,110,121,152,161,162,177,181,187,200,220,225,230,  
260,266,280,300,350,450,601,650,740,800,900,901,902,903,940,941,942,  
1000,1001,1010,1019,1028,1037,1046,1055,1064,1073,1082,1091,1100,1109};
```

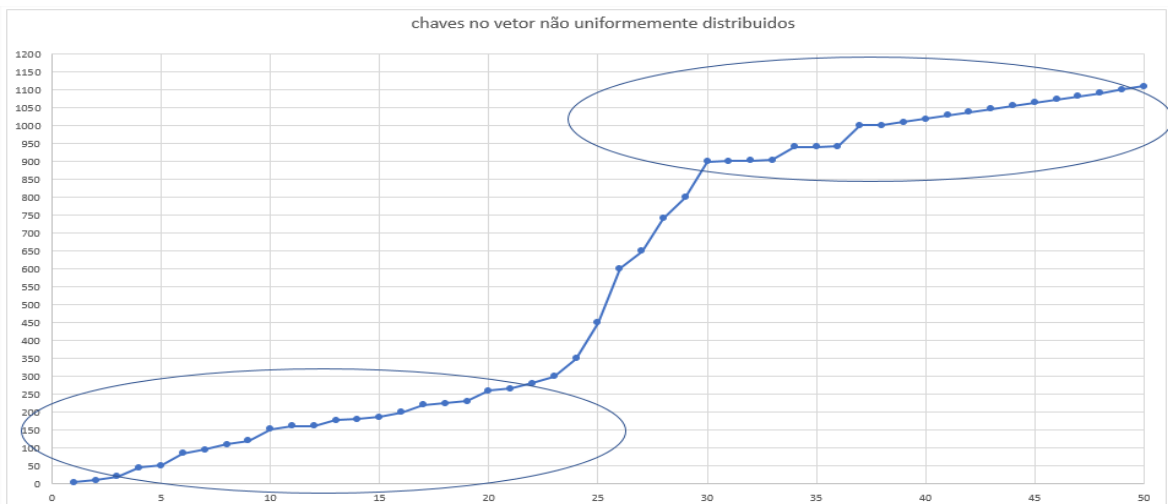
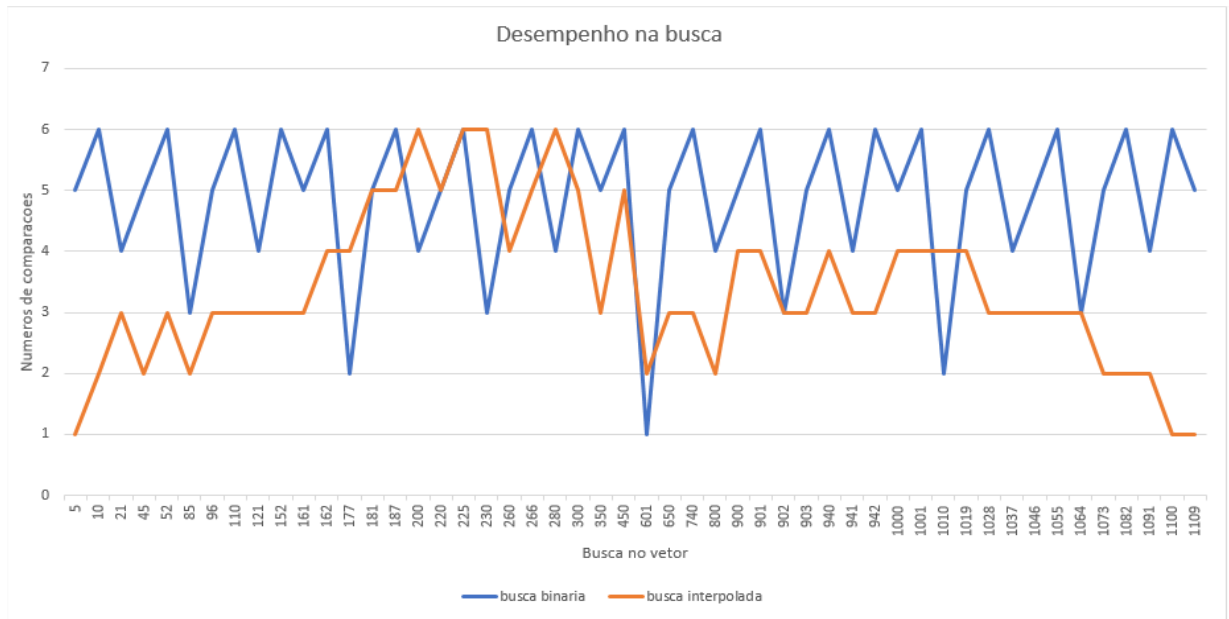


Figura 2: Teste 1 com dados não uniformemente distribuídos

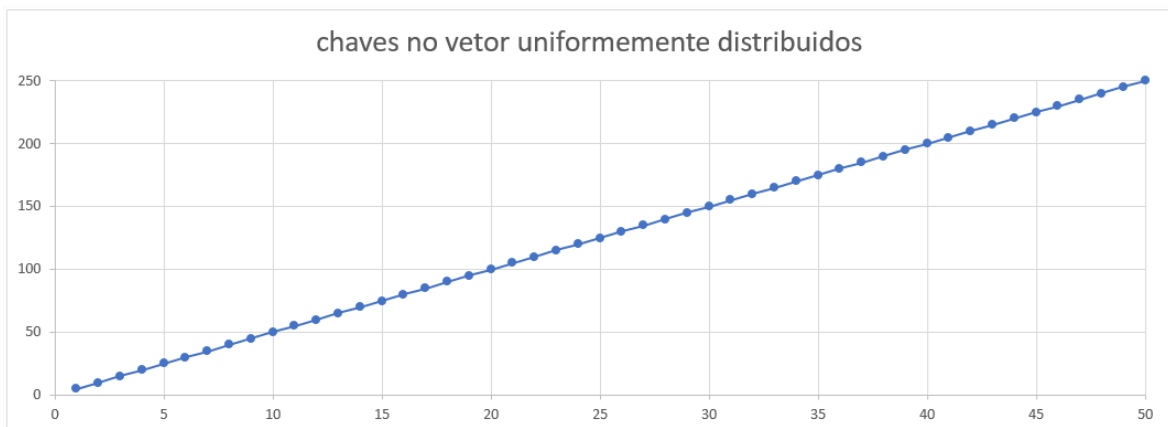
No gráfico acima temos o primeiro conjunto de dados em volta da chave 200 e outro conjunto de dados em volta da chave 1000



*Figura 3:Desempenho na busca teste 1*

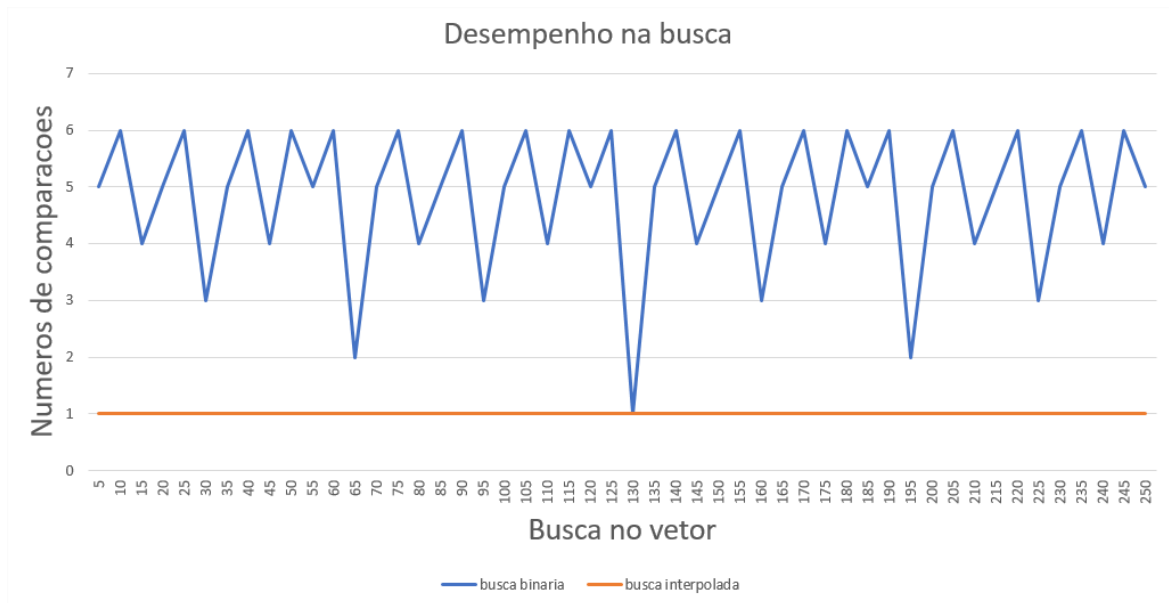
Análise de desempenho em busca binária e busca interpolada, os dados são uniformemente distribuídos neste segundo teste, os números de elementos no vetor é 50

```
int[] vector11 = new int[]
{5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100,105,110,
115,120,125,130,135,140,145,150,155,160,165,170,175,180,185,190,195,
200,205,210,215,220,225,230,235,240,245,250};
```



*Figura 4:Teste 2 com dados uniformemente distribuídos*

O gráfico acima os dados estão uniformemente distribuídos



*Figura 5: Desempenho na busca teste 2*

O teste foi realizado num ambiente controlado, tendo conhecimento da organização dos dados no vetor e foi analisado o número de comparações para que a busca fosse concluída, encontrando as chaves 5,10, 15, 20 ... até 250. Esta mesma busca foi aplicada em ambos os algoritmos.

## CAPÍTULO 6: Busca, inserção e remoção em árvores AVL

### 6.1. Descrição do algoritmo

Uma árvore binária AVL balanceada com  $n$  elementos operam em busca, inserção e remoção nos piores casos sempre em  $O(\log_2 n)$ . Segundo as comprovações de delson-Velskii e Landis que garante que a árvore balanceada nunca será 45% mais alta que a correspondente árvore perfeitamente balanceada, independentemente do número de nós existentes (1).

### 6.1.1 Características

- A ordenação opera quando a árvore fica desbalanceada a cada inserção de dados.
- É eficiente para trabalhar com muitos dados

### 6.1.2 Codificação em C#

```
class AVL
{
    class no    // criação do nó com dois ponteiros
    {
        public int dado;
        public no esquerda; // ponteiros a esquerda
        public no direita;  // ponteiro a direita
        public no(int dado)
        {
            this.dado = dado;
        }
    }
    no raiz;

    public AVL()
    {
    }

    //----- adiciona um nó na árvore -----

    public void adicionar_no(int dado)
    {
        no novo_no = new no(dado); // cria uma estrutura de dados nó
        if (raiz == null)           // adiciona nó na raiz
        {
            raiz = novo_no;
        }
        else
        {
            raiz = inserir(raiz, novo_no);
            // verifica onde adicionar o novo nó
        }
    }

    private no inserir(no no_atual, no recebe_novo_no)
    {
        if (no_atual == null) // primeira inserção
        {
            no_atual = recebe_novo_no;
            return no_atual;
        }
        else if (recebe_novo_no.dado < no_atual.dado) // insere no lado esquerdo
        {
            no_atual.esquerda = inserir(no_atual.esquerda, recebe_novo_no);
            no_atual = balancear_arvore(no_atual); // verifica se tem que balancear
        }
        else if (recebe_novo_no.dado > no_atual.dado) // insere no lado direito
        {
        }
```



```

        no_atual.direita = inserir(no_atual.direita, recebe_novo_no);
        no_atual = balancear_arvore(no_atual); // verifica se tem que balancear
    }
    return no_atual;
}

//----- faz o balanceamento da arvore -----

private no balancear_arvore(no no_atual)
{
    int fator = fator_de_balanciamento(no_atual);
    if (fator > 1)
    {
        if (fator_de_balanciamento(no_atual.esquerda) > 0)
        {
            no_atual = rotacao_LL(no_atual);
        }
        else
        {
            no_atual = rotacao_LR(no_atual);
        }
    }
    else if (fator < -1)
    {
        if (fator_de_balanciamento(no_atual.direita) > 0)
        {
            no_atual = rotacao_RL(no_atual);
        }
        else
        {
            no_atual = rotacao_RR(no_atual);
        }
    }
    return no_atual;
}

//----- deleta no, raiz ou folha -----
public void Delete(int chave_procurado) // deleta um no
{ raiz = Delete(raiz, chave_procurado); }

private no Delete(no no_atual, int chave_procurado)
{
    no no_pai;
    if (no_atual == null)
    { return null; }
    else
    {
        //procura na sub arvore da esquerda
        if (chave_procurado < no_atual.dado)
        {
            no_atual.esquerda = Delete(no_atual.esquerda, chave_procurado);
            if (fator_de_balanciamento(no_atual) == -2)
            {
                if (fator_de_balanciamento(no_atual.direita) <= 0)
                {
                    no_atual = rotacao_RR(no_atual);
                }
                else
                {
                    no_atual = rotacao_RL(no_atual);
                }
            }
        }
    }
}

```

```

    }
    //procura na sub arvore da direita
    else if (chave_procurado > no_atual.dado)
    {
        no_atual.direita = Delete(no_atual.direita, chave_procurado);
        if (fator_de_balanciamento(no_atual) == 2)
        {
            if (fator_de_balanciamento(no_atual.esquerda) >= 0)
            {
                no_atual = rotacao_LL(no_atual);
            }
            else
            {
                no_atual = rotacao_LR(no_atual);
            }
        }
    }
    else // chave encontrada
    {
        if (no_atual.direita != null)
        {
            //deleção com sucesso
            no_pai = no_atual.direita;
            while (no_pai.esquerda != null)
            {
                no_pai = no_pai.esquerda;
            }
            no_atual.dado = no_pai.dado;
            no_atual.direita = Delete(no_atual.direita, no_pai.dado);
            if (fator_de_balanciamento(no_atual) == 2)//balanceamento
            {
                if (fator_de_balanciamento(no_atual.esquerda) >= 0)
                {
                    no_atual = rotacao_LL(no_atual);
                }
                else { no_atual = rotacao_LR(no_atual); }
            }
        }
        else
        {
            return no_atual.esquerda;
        }
    }
    }
    return no_atual;
}
// faz busca na arvore
public void busca(int chave)
{
    if (busca(chave, raiz).dado == chave)
    {
        Console.WriteLine("{0} encontrado ", chave);
    }
    else
    {
        Console.WriteLine(" nao encontrado !");
    }
}

private no busca(int chave_procurado, no no_atual)

```

```

{
    if (chave_procurado < no_atual.dado)
    {
        if (chave_procurado == no_atual.dado)
        {
            return no_atual;
        }
        else
            return busca(chave_procurado, no_atual.esquerda);
    }
    else
    {
        if (chave_procurado == no_atual.dado)
        {
            return no_atual;
        }
        else
            return busca(chave_procurado, no_atual.direita);
    }
}

//----- calcula o fator de balanceamento e altura da arvore -----
private int maximo(int sub_arvore_esquerda, int sub_arvore_direita)
{
    return sub_arvore_esquerda > sub_arvore_direita ? sub_arvore_esquerda :
sub_arvore_direita;
}

private int altura_da_arvore(no no_atual)
{
    int altura = 0;
    if (no_atual != null)
    {
        // l --> sub arvore a esquerda
        // r --> sub arvore a direita
        int l = altura_da_arvore(no_atual.esquerda);
        int r = altura_da_arvore(no_atual.direita);
        int calcular_altura = maximo(l, r);
        altura = calcular_altura + 1;
    }
    return altura;
}

private int fator_de_balanciamento(no no_atual)
{
    // l --> sub arvore a esquerda
    // r --> sub arvore a direita
    int l = altura_da_arvore(no_atual.esquerda);
    int r = altura_da_arvore(no_atual.direita);
    int fator = l - r;
    return fator;
}

// ----- rotações para manter balanceado a arvore AVL -----
private no rotacao_RR(no no_pai)
{
    // RR --> direita a direita
    no pivo = no_pai.direita;
    no_pai.direita = pivo.esquerda;
    pivo.esquerda = no_pai;
}

```

```

    return pivo;
}
private no rotacao_LL(no no_pai)
{ // LL --> esquerda a esquerda
  no pivo = no_pai.esquerda;
  no_pai.esquerda = pivo.direita;
  pivo.direita = no_pai;
  return pivo;
}
private no rotacao_LR(no no_pai)
{ // LR --> esquerda e depois a direita
  no pivo = no_pai.esquerda;
  no_pai.esquerda = rotacao_RR(pivo);
  return rotacao_LL(no_pai);
}
private no rotacao_RL(no no_pai)
{ ///RL --> direita e depois a esquerda
  no pivo = no_pai.direita;
  no_pai.direita = rotacao_LL(pivo);
  return rotacao_RR(no_pai);
}
}

```

## 6.1.2 Complexidade de tempo e de espaço

Tabela 6 Complexidade de tempo e de espaço. Algoritmo árvore AVL

Caso médio	$\rightarrow O(\log_2^n)$
Melhor caso	$\rightarrow O(\log_2^n)$
Pior caso	$\rightarrow O(\log_2^n)$

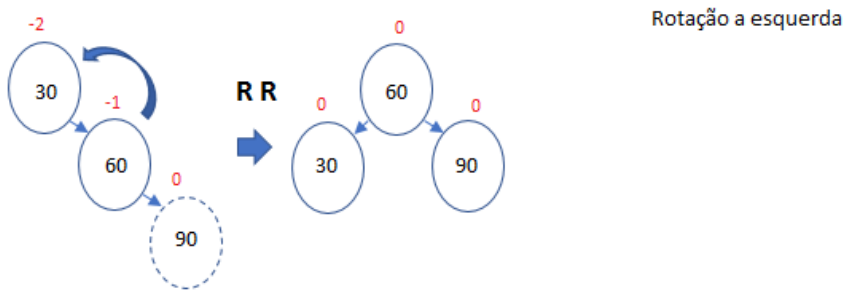
## 6.2. Inserção em árvore AVL

sequencia de inserção	1°	2°	3°	4°	5°	6°	7°	8°	10°
	30	60	90	20	10	130	100	2	8

Primeiro passo inserimos o 30

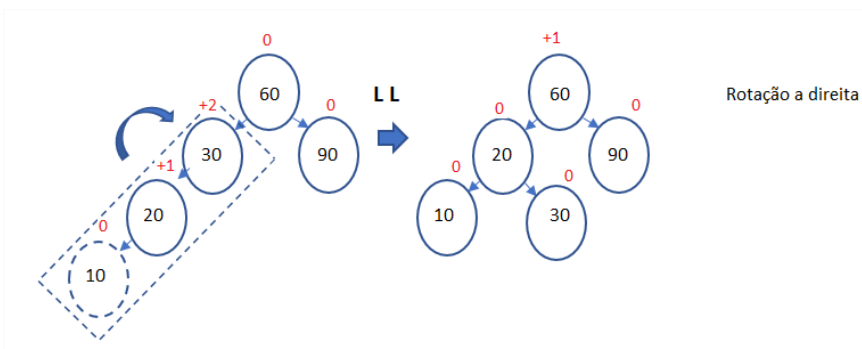


Terceiro passo inserimos o 60 e 90



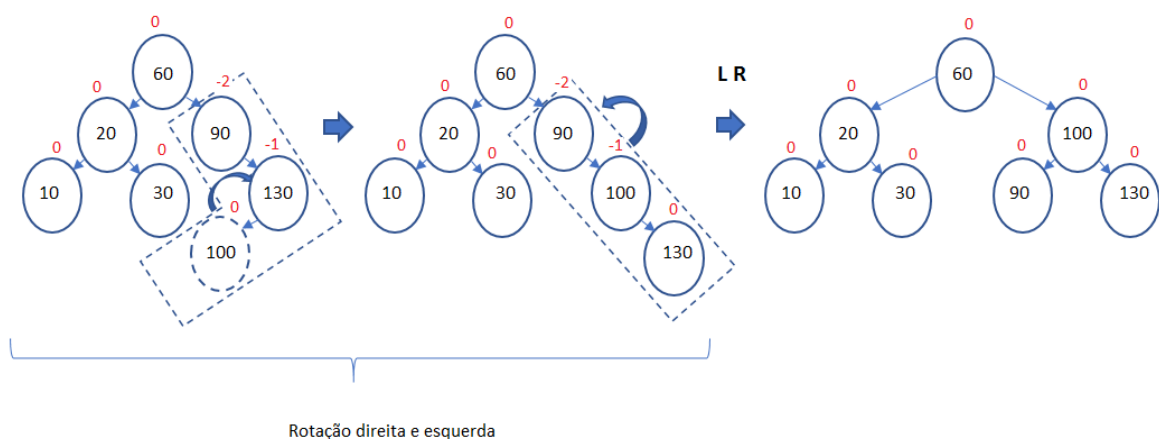
A inserção do 90 gerou um desbalanceamento no nó 30 para corrigir e necessário fazer uma rotação RR

Quarto passo inserimos 20 e 10



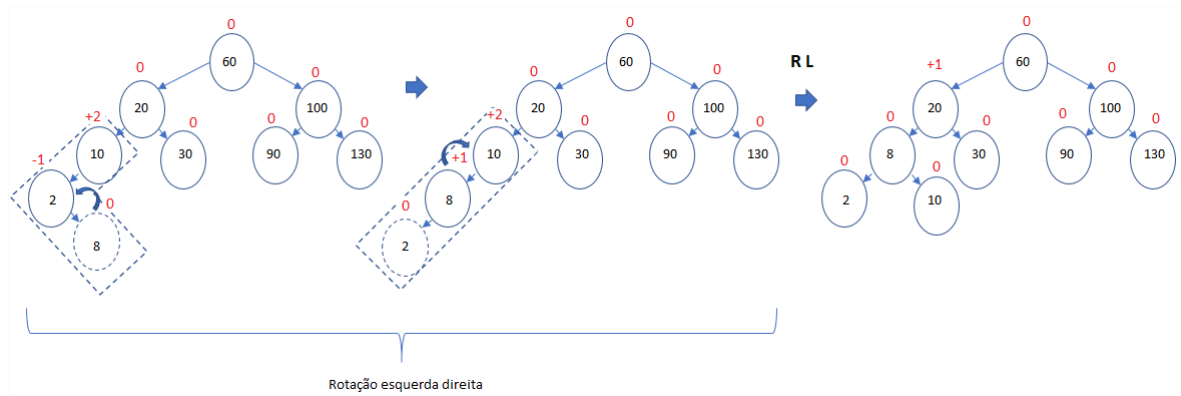
A inserção do elemento 10 gerou um desbalanceamento a subárvore de raiz 30, para corrigir e necessário fazer uma rotação LL

Quinto passo inserimos 130 e 100



A inserção da chave 100 gerou um desbalanceamento na subárvore 90, e necessário fazer uma rotação para a direita e outra rotação para a esquerda, rotação L R

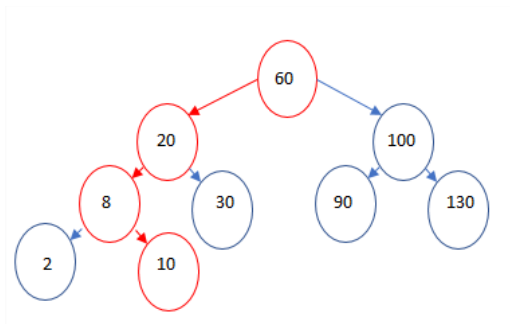
Sexto passo inserimos 2 e 8



A inserção da chave 8 gerou um desbalanceamento no nó 10, para corrigir e necessário fazer uma rotação para a esquerda e outra rotação para direita, rotação R L

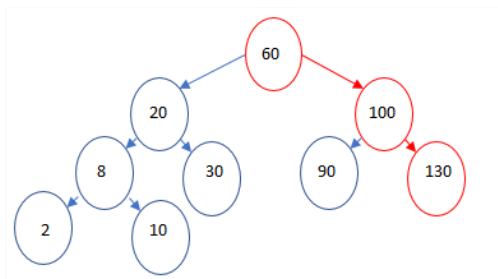
### 6.3. Busca em arvore AVL

Busca da chave 10



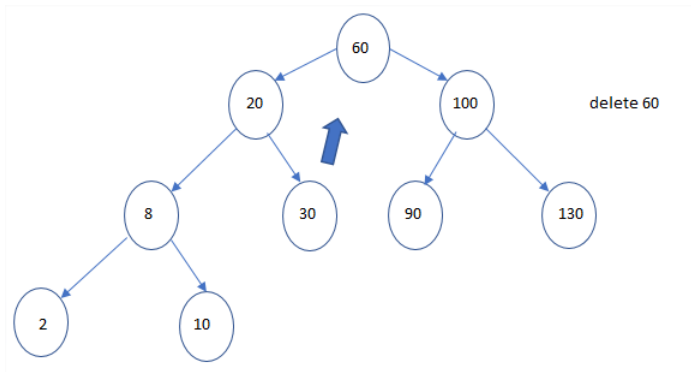
Para buscar a chave 10 foi necessário primeiro acessar a raiz e depois seguir na direção esquerda para acessar o nó 20 e prosseguir a esquerda para acessar o nó 8, neste nó verifica que a chave é maior, muda a direção para direita para achar o 10.

Busca da chave 130



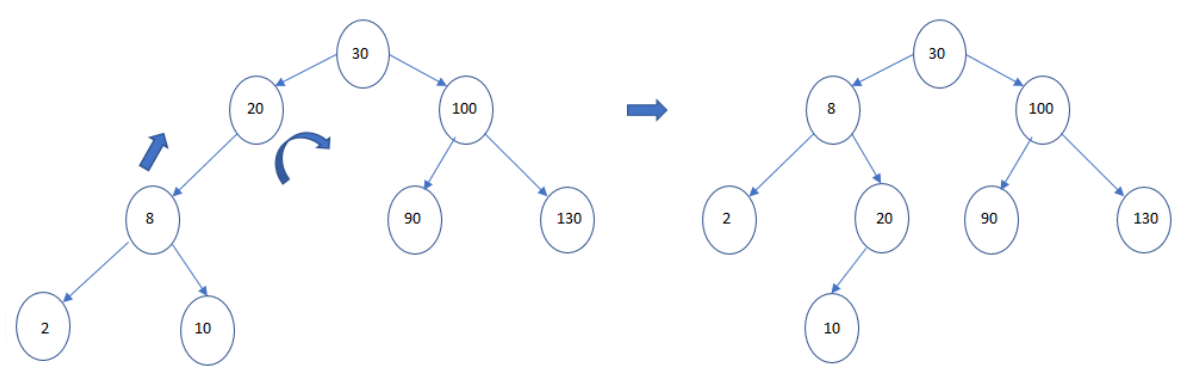
### 6.4. Remoção em arvore AVL

Remover a chave 60



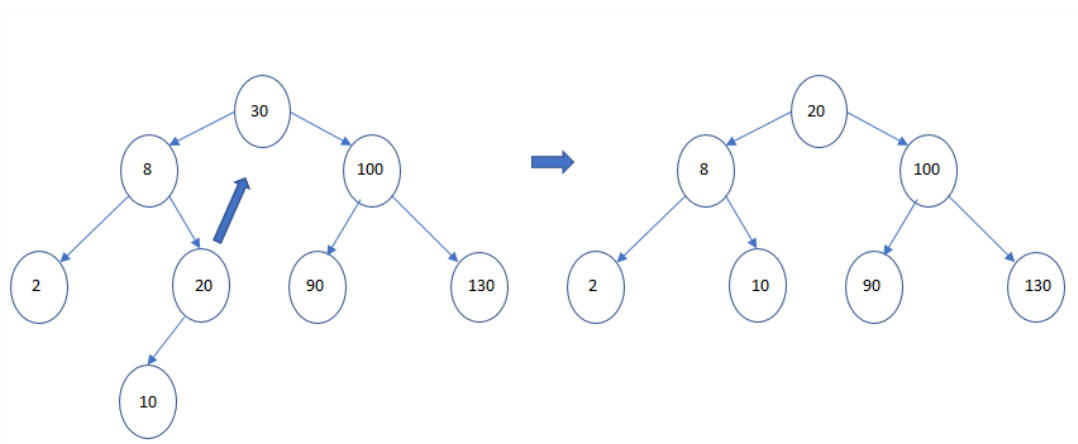
Balanceamento:

Para remover a raiz 60, primeiro ocorre uma busca na subárvore a esquerda para achar a maior chave entre os menores dessa subárvore, neste exemplo a chave 30 irá substituir a raiz.



Ocorreu uma rotação para a direita, onde o nó 8 subiu um nível e teve a sua folha direita o nó 20, a folha 10 ficou à esquerda da sua nova raiz, a chave 20.

Remover a chave 30



A chave 20 assumiu a posição da raiz e a folha 10 assumiu a posição de folha a direita do nó 8.

## 6.5. Analise das operações na árvore AVL

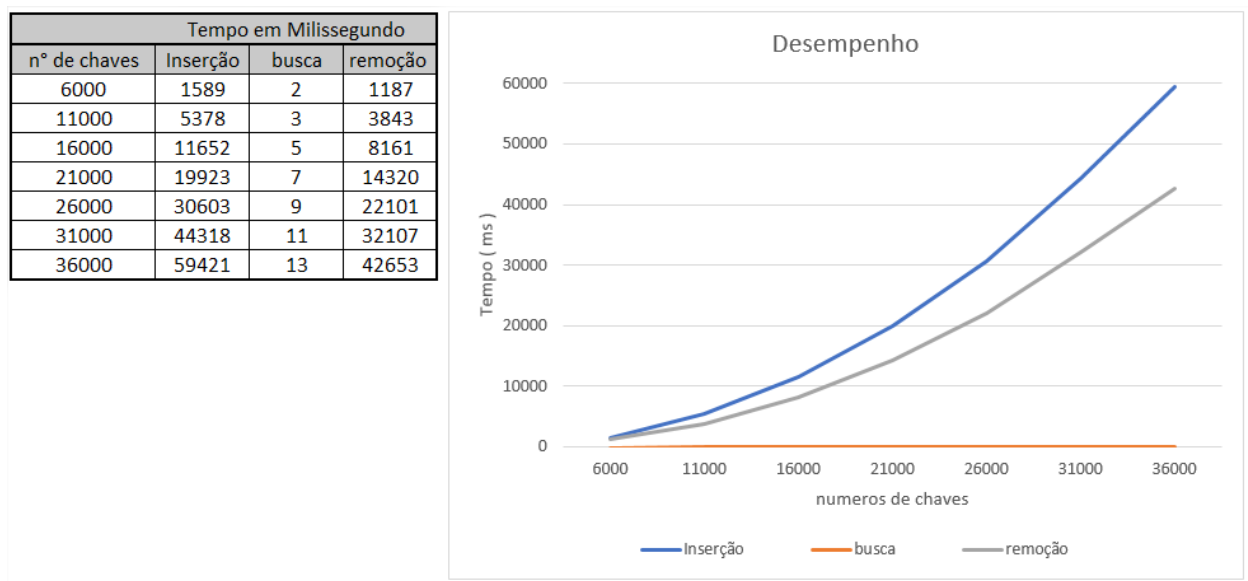


Figura 6: Desempenho árvore AVL

A análise se procedeu para cada conjunto de chave, primeiro inserimos 6000 chaves e medimos o tempo de inserção. Segundo fizemos a procura de cada chave, uma por uma e medimos o tempo para fazer toda essa busca das 6000 chaves. Em terceiro fizemos a remoção de todas as 6000 chaves e também medimos o tempo. A análise de tempo para cada entrada dos números de chaves está na tabela e o comportamento das funções de inserção, busca e remoção está representado no gráfico ao lado.





# REFERÊNCIAS

T. H. Cormen, C. E. Leiserson, R. L. Rivest e C. Stein, Introduction to Algorithms, McGraw-Hill, 2001, second edition.

[1] <http://dcm.ffclrp.usp.br/~augusto/teaching/aedi/AED-I-Arvores-AVL.pdf>

## Anexo: PROJETO EM C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static System.Console;
using System.Diagnostics;

class AVL
{
    class no // criacao do no com dois ponteiros
    {
        public int dado;
        public no esquerda; // ponteiros a esquerda
        public no direita; // ponteiro a direita
        public no(int dado)
        {
            this.dado = dado;
        }
    }
    no raiz;

    public AVL()
    {
    }

    //----- adiciona um nó na arvore -----

    public void adicionar_no(int dado)
    {
        no novo_no = new no(dado); // cria uma estrutura de dados nó
    }
}
```

```

    if (raiz == null)           // adiciona nó na raiz
    {
        raiz = novo_no;
    }
    else
    {
        raiz = inserir(raiz, novo_no);
        // verifica onde adicionar o novo nó
    }
}

private no inserir(no no_atual, no recebe_novo_no)
{
    if (no_atual == null) // primeira insercao
    {
        no_atual = recebe_novo_no;
        return no_atual;
    }
    else if (recebe_novo_no.dado < no_atual.dado) // insere no lado esquerdo
    {
        no_atual.esquerda = inserir(no_atual.esquerda, recebe_novo_no);
        no_atual = balancear_arvore(no_atual); // verifica se tem que balancear
    }
    else if (recebe_novo_no.dado > no_atual.dado) // insere no lado direito
    {
        no_atual.direita = inserir(no_atual.direita, recebe_novo_no);
        no_atual = balancear_arvore(no_atual); // verifica se tem que balancear
    }
    return no_atual;
}

//----- faz o balanceamento da arvore -----

private no balancear_arvore(no no_atual)
{
    int fator = fator_de_balanciamento(no_atual);
    if (fator > 1)
    {
        if (fator_de_balanciamento(no_atual.esquerda) > 0)
        {
            no_atual = rotacao_LL(no_atual);
        }
        else
        {
            no_atual = rotacao_LR(no_atual);
        }
    }
    else if (fator < -1)
    {
        if (fator_de_balanciamento(no_atual.direita) > 0)
        {
            no_atual = rotacao_RL(no_atual);
        }
        else
        {
            no_atual = rotacao_RR(no_atual);
        }
    }
    return no_atual;
}

```

```

//----- deleta no, raiz ou folha -----
public void Delete(int chave_procurado) // deleta um no
{ raiz = Delete(raiz, chave_procurado); }

private no Delete(no no_atual, int chave_procurado)
{
    no no_pai;
    if (no_atual == null)
    { return null; }
    else
    {
        //procura na sub arvore da esquerda
        if (chave_procurado < no_atual.dado)
        {
            no_atual.esquerda = Delete(no_atual.esquerda, chave_procurado);
            if (fator_de_balanciamento(no_atual) == -2)
            {
                if (fator_de_balanciamento(no_atual.direita) <= 0)
                {
                    no_atual = rotacao_RR(no_atual);
                }
                else
                {
                    no_atual = rotacao_RL(no_atual);
                }
            }
        }
        //procura na sub arvore da direita
        else if (chave_procurado > no_atual.dado)
        {
            no_atual.direita = Delete(no_atual.direita, chave_procurado);
            if (fator_de_balanciamento(no_atual) == 2)
            {
                if (fator_de_balanciamento(no_atual.esquerda) >= 0)
                {
                    no_atual = rotacao_LL(no_atual);
                }
                else
                {
                    no_atual = rotacao_LR(no_atual);
                }
            }
        }
    }
    else // chave encontrada
    {
        if (no_atual.direita != null)
        {
            //deleção com sucesso
            no_pai = no_atual.direita;
            while (no_pai.esquerda != null)
            {
                no_pai = no_pai.esquerda;
            }
            no_atual.dado = no_pai.dado;
            no_atual.direita = Delete(no_atual.direita, no_pai.dado);
            if (fator_de_balanciamento(no_atual) == 2) //rebalanciamento
            {
                if (fator_de_balanciamento(no_atual.esquerda) >= 0)
                {
                    no_atual = rotacao_LL(no_atual);
                }
            }
        }
    }
}

```

```

        }
        else { no_atual = rotacao_LR(no_atual); }
    }
}
else
{
    return no_atual.esquerda;
}
}
}
return no_atual;
}
// faz busca na arvore
public void busca(int chave)
{
    if (busca(chave, raiz).dado == chave)
    {
        Console.WriteLine("{0} encontrado ", chave);
    }
    else
    {
        Console.WriteLine(" nao encontrado !");
    }
}

private no busca(int chave_procurado, no no_atual)
{
    if (chave_procurado < no_atual.dado)
    {
        if (chave_procurado == no_atual.dado)
        {
            return no_atual;
        }
        else
            return busca(chave_procurado, no_atual.esquerda);
    }
    else
    {
        if (chave_procurado == no_atual.dado)
        {
            return no_atual;
        }
        else
            return busca(chave_procurado, no_atual.direita);
    }
}

//----- calcula o fator de balanceamento e altura da arvore -----
private int maximo(int sub_arvore_esquerda, int sub_arvore_direita)
{
    return sub_arvore_esquerda > sub_arvore_direita ? sub_arvore_esquerda :
sub_arvore_direita;
}

private int altura_da_arvore(no no_atual)
{
    int altura = 0;

```

```

        if (no_atual != null)
        {
            // l --> sub arvore a esquerda
            // r --> sub arvore a direita
            int l = altura_da_arvore(no_atual.esquerda);
            int r = altura_da_arvore(no_atual.direita);
            int calcular_altura = maximo(l, r);
            altura = calcular_altura + 1;
        }
        return altura;
    }

private int fator_de_balanciamento(no no_atual)
{
    // l --> sub arvore a esquerda
    // r --> sub arvore a direita
    int l = altura_da_arvore(no_atual.esquerda);
    int r = altura_da_arvore(no_atual.direita);
    int fator = l - r;
    return fator;
}

//----- rotações para manter balanceado a arvore AVL -----
private no rotacao_RR(no no_pai)
{
    // RR --> direita a direita
    no pivo = no_pai.direita;
    no_pai.direita = pivo.esquerda;
    pivo.esquerda = no_pai;
    return pivo;
}
private no rotacao_LL(no no_pai)
{
    // LL --> esquerda a esquerda
    no pivo = no_pai.esquerda;
    no_pai.esquerda = pivo.direita;
    pivo.direita = no_pai;
    return pivo;
}
private no rotacao_LR(no no_pai)
{
    // LR --> esquerda e depois a direita
    no pivo = no_pai.esquerda;
    no_pai.esquerda = rotacao_RR(pivo);
    return rotacao_LL(no_pai);
}
private no rotacao_RL(no no_pai)
{
    //RL --> direita e depois a esquerda
    no pivo = no_pai.direita;
    no_pai.direita = rotacao_LL(pivo);
    return rotacao_RR(no_pai);
}
}
//----- arvore avl -----
namespace ConsoleApplication1
{
    class Program
    {

        static public void QuickSort(int[] vetor, int primeiro, int ultimo)
        {

            int baixo, alto, meio, pivo, repositorio;

```

```

baixo = primeiro;
alto = ultimo;
meio = (int)((baixo + alto) / 2);

pivo = vetor[meio];

while (baixo <= alto)
{
    while (vetor[baixo] < pivo)
        baixo++;
    while (vetor[alto] > pivo)
        alto--;
    if (baixo < alto)
    {
        repositorio = vetor[baixo];
        vetor[baixo++] = vetor[alto];
        vetor[alto--] = repositorio;
    }
    else
    {
        if (baixo == alto)
        {
            baixo++;
            alto--;
        }
    }
}

if (alto > primeiro)
    QuickSort(vetor, primeiro, alto);
if (baixo < ultimo)
    QuickSort(vetor, baixo, ultimo);
}

```

```

///-----
static public void RandomNumber(int range, int quantidades_de_numeros, int[]
vector, int sentinela)
{
    Random teste = new Random(591);
    // semente usada para repetirmos o mesmo teste
    for (int i = 0; i < quantidades_de_numeros + 1; i++)
    {

        if (i < quantidades_de_numeros)
        {
            vector[i] = teste.Next(0, range);
            if (vector[i] == sentinela)
            {
                i = i - 1;
            }
        }
        else
        {
            if (sentinela != 0)
            {
                // caso tenha sentinela, vai na última posição
                vector[quantidades_de_numeros] = sentinela;
            }
        }
    }
}

```

```

        else
        {
            vector[i] = teste.Next(0, range);

            if (vector[i] == sentinela)
            {
                i = i - 1;
            }
        }
    }
}

static public void imprimir(int[] vector)
{
    for (int i = 0; i < vector.Length; i++)// n
    {
        Console.WriteLine(vector[i]);
    }
}

//----- busca com sentinela -----
static public void busca_com_sentinela(int[] vector, int procurado, int sentinela)
{
    int posicao = 0;
    for (int i = 0; vector[i] != sentinela; i++)// n
    {
        if (vector[i] == procurado)
        {
            WriteLine($"chave encontrada " + vector[i] + " posicao " + i);
            break;
        }
        posicao = i;
    }

    if (vector[posicao + 1] == sentinela)
    {
        WriteLine($" chave nao encontrada "+ procurado+" chegou no sentinela "+ vector[posicao + 1]);
    }
}

//----- busca sem sentinela -----
static public void busca_sem_sentinela(int[] vector, int procurado)
{
    int indica_achou = 0;
    int posicao_do_vetor = 0;
    for (int i = 0; i < vector.Length; i++)// n
    {
        if (vector[i] == procurado)
        {
            posicao_do_vetor = i + 1;// sai do laço
            indica_achou = 1;
            WriteLine($" achou " + procurado + " posicao " + posicao_do_vetor);
        }
    }
}

```



```

        if (indica_achou == 0)
        {
            WriteLine($" nao encontrado " + procurado);
        }
    }

    static public void vetor_decrescente(int[] vector, int[] vetor_decrescente)
    {
        int k = vector.Length - 1;

        for (int i = 0; i < vector.Length; i++)// n/2 ou (n+1)/2
        {
            if (i <= k)
            {
                vetor_decrescente[i] = vector[k];
                vetor_decrescente[k] = vector[i];
                k--;
            }
            else // quando i e maior que k (cessa o processo)
            {
                i = vector.Length + 1;
            }
        }
    }

//----- busca sequencial ordenado -----

    static public void busca_sequencial_ordenado(int[] vector, int procurado)
    {
        for (int i = 0; i < vector.Length; i++)// n-1
        {
            if (vector[i] == procurado)
            {
                WriteLine($" achou " + vector[i] + " posicao " + i);
                i = vector.Length + 1;// depois que achou, sai do for
            }
        }
    }

//----- busca binaria -----

    static public void busca_Binaria(int[] vector, int procurado)
    {
        int indice_menor, indice_meio, indice_maior;
        indice_menor = 0; indice_maior = vector.Length;
        int comparacoes = 0;
        while (indice_menor <= indice_maior)
        {
            comparacoes = comparacoes + 1;
            indice_meio = (indice_menor + indice_maior) / 2;

            // WriteLine($"indice do meio " + indice_meio);

            if (vector[indice_meio] == procurado)
            {
                // WriteLine($" achou " + vector[indice_meio] + " posicao "
                // + indice_meio+" n° de comparacoes "+ comparacoes);
                WriteLine($" " + comparacoes);
            }
        }
    }

```

```

        indice_menor = indice_maior + 1;
        // condicao para sair do while
    }
    if (vetor[indice_meio] < procurado)
    { indice_menor = indice_meio + 1; }
    else
    { indice_maior = indice_meio - 1; }
}

}

///----- busca interpolada -----

static public void Busca_Interpolada(int elemento_procurado, int[] vetor)
{
    int inicio = 0;
    int meio;
    int final = vetor.Length - 1;
    int achou = 0;
    // flag para indiar se encontrou o elemento no final da busca
    int comparacoes = 0;

    while (inicio < final)
    {
        comparacoes = comparacoes + 1;
        meio = (inicio + ((final - inicio) * (elemento_procurado -
        vetor[inicio])) / (vetor[final] - vetor[inicio]));
        WriteLine($" meio " + meio);

        if (elemento_procurado < vetor[meio])
        {
            final = meio - 1;
        }
        else if (elemento_procurado > vetor[meio])
        {
            inicio = meio + 1;
        }
        else
        {
            WriteLine($" achou " + vetor[meio] + " posicao " + meio + "
            n° de comparacoes " + comparacoes);
            inicio = final + 1;
            achou = 1;
        }
    }

    if (achou == 0)
    { WriteLine($" nao encontrado "); }
}

///----- análise de desempenho -----

static public void Desempenho_busca_binaria_verso_busca_interpolada()
{
    int[] vetor10 = new int[] // Dados não uniformemente distribuídas
    {5,10,21,45,52,85,96,110,121,152,161,162,177,181,187,200,220,225,230,
    260,266,280,300,350,450,601,650,740,800,900,901,902,903,940,941,942,
    1000,1001,1010,1019,1028,1037,1046,1055,1064,1073,1082,1091,1100,1109
    };

```

```

int[] vector11 = new int[] //Dados uniformemente distribuídas
{ 5,10,15,20,25,30,35,40,45,50,55,60,65,70,75,80,85,90,95,100,105,110,
  115,120,125,130,135,140,145,150,155,160,165,170,175,180,185,190,195,
  200,205,210,215,220,225,230,235,240,245,250};
WriteLine($"----- Dados não uniformemente distribuídos ---");
WriteLine("");
WriteLine($"----- Busca Binaria -----");
WriteLine("");
for (int i = 0; i < 50; i++)
{ busca_Binaria(vector10, vector10[i]); }
WriteLine("");
WriteLine($"----- Busca Interpolada -----");
WriteLine("");

for (int i = 0; i < 50; i++)
{ Busca_Interpolada(vector10[i], vector10); }
WriteLine("");
WriteLine($"----- Dados não uniformemente distribuídos ---");
WriteLine("");
WriteLine($"----- Busca Binaria -----");
WriteLine("");
for (int i = 0; i < 50; i++)
{ busca_Binaria(vector11, vector11[i]); }
WriteLine("");
WriteLine($"----- Busca Interpolada -----");
WriteLine("");

for (int i = 0; i < 50; i++)
{ Busca_Interpolada(vector11[i], vector11); }
WriteLine("");
}

//-----
static public void Desempenho_AVL()
{
    var stopwatch = new Stopwatch();
    int quantidade_de_numero = 1000; // valor inicial

    for (int j = 0; j < 7; j++)
    {
        quantidade_de_numero = quantidade_de_numero + 5000; // valor inicial
        int[] vector15 = new int[quantidade_de_numero];
        RandomNumber(vector15.Length, vector15.Length - 1, vector15, 0);

        AVL tree = new AVL();
        WriteLine($"Tempo de insercao " + quantidade_de_numero + " chaves ");
        WriteLine("");
        stopwatch.Start(); // tempo inicio
        for (int i = 0; i < vector15.Length; i++)
        { tree.adicionar_no(vector15[i]); }
        stopwatch.Stop(); // tempo final
        WriteLine($"{stopwatch.ElapsedMilliseconds}");
        stopwatch.Reset();

        WriteLine("");
        WriteLine($"Tempo de busca " + quantidade_de_numero + " chaves ");
        WriteLine("");
        stopwatch.Start(); // tempo inicio
        for (int i = 0; i < vector15.Length; i++)
        { tree.busca(vector15[i]); }
    }
}

```

```

        stopwatch.Stop();    // tempo final
        WriteLine($"{stopwatch.ElapsedMilliseconds}");
        stopwatch.Reset();
        WriteLine("");
        WriteLine($"Tempo de delete " + quantidade_de_numero + "      chaves ");
        WriteLine("");
        stopwatch.Start();    // tempo inicio
        for (int i = 0; i < vector15.Length; i++)
        { tree.Delete(vector15[i]); }
        stopwatch.Stop();    // tempo final
        WriteLine($"{stopwatch.ElapsedMilliseconds}");
        stopwatch.Reset();

    }

}

static void Main(string[] args)
{
    var stopwatch = new Stopwatch();
    int quantidade_de_numero1 = 10000000; // valor inicial

    int numeros_de_casa2 = 10000000;

    // for (int i = 0; i < 1; i++)// teste para analise
    // {
    int posicao1, posic;
    quantidade_de_numero1 = quantidade_de_numero1;
    int[] vector01 = new int[quantidade_de_numero1];
    int[] vector02 = new int[quantidade_de_numero1];
    int[] vector03 = new int[quantidade_de_numero1];
    int[] vector04 = new int[quantidade_de_numero1];
    int[] vector05 = new int[quantidade_de_numero1];

    // RandomNumber(numeros_de_casa2, quantidade_de_numero1, vector01);
    int sentinela = 8;
    int procurado = 47960395;
    int procurado_2 = 74080075;
    int procurado_3 = 244775;
    int procurado_4 = 7;
    int procurado_5 = 450; //7
    // int procurado_4 = 162;
    RandomNumber(vector01.Length, vector01.Length - 1, vector01, sentinela);
    RandomNumber(vector02.Length, vector02.Length - 1, vector02, 0);
    RandomNumber(vector03.Length, vector03.Length - 1, vector03, 0);
    RandomNumber(vector04.Length, vector04.Length - 1, vector04, 0);
    RandomNumber(vector05.Length, vector05.Length - 1, vector05, 0);

    QuickSort(vector03, 0, vector03.Length - 1);
    QuickSort(vector04, 0, vector04.Length - 1);
    QuickSort(vector05, 0, vector05.Length - 1);

    stopwatch.Start();    // tempo inicio
    busca_com_sentinela(vector01, procurado, sentinela);
    stopwatch.Stop();    // tempo final
    WriteLine($"{stopwatch.ElapsedMilliseconds}");
    stopwatch.Reset();
}

```

```

stopwatch.Start();    // tempo inicio
busca_sem_sentinela(vector02, procurado_2);
stopwatch.Stop();     // tempo final
WriteLine($"{stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();
// busca sequencial

stopwatch.Start();    // tempo inicio
busca_sequencial_ordenado(vector03, procurado_3);
stopwatch.Stop();     // tempo final
WriteLine($"{stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();

stopwatch.Start();    // tempo inicio
busca_Binaria(vector04, procurado_4);
stopwatch.Stop();     // tempo final
WriteLine($"{stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();

stopwatch.Start();    // tempo inicio
Busca_Interpolada(procurado_5, vector05);
stopwatch.Stop();     // tempo final
WriteLine($"{stopwatch.ElapsedMilliseconds}");
stopwatch.Reset();

Desempenho_busca_binaria_verso_busca_interpolada();

Desempenho_AVL();

Console.ReadKey();

    }
}

```