



Instituto Noroeste Fluminense de Educação Superior  
Departamento de Ciências Exatas, Biológicas e das Terras(PEB)

Sarah Victória Cardoso Machado

Trabalho de Estrutura de Dados II

## **Implementações**

Santo Antônio de Pádua

2021

# Índice

## 1 – Bubble Sort (Ordenação por Bolha)

1.1 – Resumo do Método.

1.2 – Complexidade.

1.3 – Resultados.

1.4 – Implementação usando lista encadeada.

## 2 – Insertion Sort (Ordenação por Inserção)

2.1 – Resumo do Método.

2.2 – Complexidade.

2.3 – Resultados.

2.4 – Implementação.

## 3 – Selection Sort (Ordenação por Seleção)

3.1 – Resumo do Método.

3.2 – Complexidade.

3.3 – Resultados.

3.4 – Implementação.

## 4 – Shell Sort (Ordenação por Incrementos Diminutos)

4.1 – Resumo do Método.

4.2 – Complexidade.

4.3 – Resultados.

4.4 – Implementação.

## 5 – Heap

5.1 – Resumo do Método.

5.2 – Complexidade.

5.3 – Implementação usando vetor.

## **6 - HeapSort**

**6.1** – Resumo do Método.

**6.2** – Complexidade.

**6.3** - Resultado

**6.4** – Implementação usando vetor.

## **7 - MergeSort**

**7.1** – Resumo do Método.

**7.2** – Complexidade.

**7.3** - Resultado

**7.4** – Implementação usando vetor.

## **8 - QuickSort**

**8.1** – Resumo do Método.

**8.2** – Complexidade.

**8.3** - Resultado

**8.4** – Implementação usando vetor

## **9 - Árvore binária de busca**

**9.1** – Resumo do Método.

**9.2** – Complexidade.

**9.3** – Implementação.

## **10 - AVL**

**10.1** – Resumo do Método e comparação com ABB.

**10.2** – Rotações Simples

**10.3** – Rotações Dupla

**10.4** – Implementação.

## 1 – Bubble Sort (Ordenação por Bolha)

### 1.1 - Resumo do Método

Bubble sort, ou ordenação por bolha, consiste em um método que percorre uma lista, podendo ser sequencial ou encadeada, verificando os valores, caso o valor seguinte seja maior ele troca os valores. Sabendo que a última posição terá o maior valor, realizamos o mesmo processo novamente indo da primeira até a penúltima posição, a partir disso, repetimos o processo com todas as posições exceto a primeira que estará ordenada.

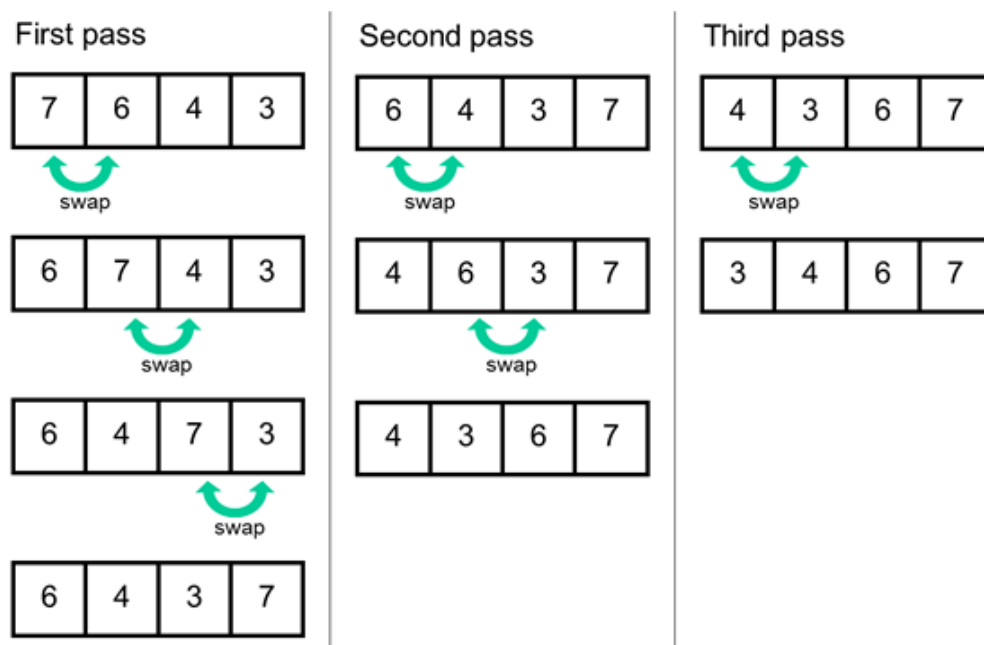


Figura 1 - Exemplo de passos executados pelo bubble sort.

Fonte: <http://www.computersciencebytes.com>

### 1.2 - Complexidade

Para realizar o cálculo da complexidade é interessante considerar o for dentro de outro for, indicando que teremos um termo quadrático. Sendo assim, teremos uma lista  $n$ , com  $n-1$  na primeira parte do processo, seguindo-se para  $n-2$ ,  $n-3$ ,  $n-4 \dots 1$ .

$$\sum_{x=1}^{x=n-1} n - x$$

Somando a P.A, temos:

$$(n - 1) \frac{(1)+(n-1)}{2} = \frac{(n-1)+(n-1)^2}{2} = \frac{n^2-n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Podendo ser,  $O(n^2)$  ou  $\Theta(n^2)$ .

Dessa forma, vemos que o algoritmo executa o mesmo número de operações independente da entrada. Dito isso, sua complexidade será  $\Theta(n^2)$  para todos os casos.

### 1.3 – Resultados

Tabela - Lista com 100 elementos

Construção da Lista	Comparação	Troca
Em ordem crescente	9801	0
Em ordem decrescente	9801	4950
Aleatório	Em média 9801	Em média 2 274,33

Obs: Diante de valores diversos, sorteados pelo aleatório, foram realizadas três execuções, a fim de elaborar uma média aritmética com os valores de troca.

Executável do decrescente:

```
"C:\Users\Del\Documents\Técnicas e práticas de programação\bubblesort_sarah\bin\Debug\bub...
99 - 98 - 97 - 96 - 95 - 94 - 93 - 92 - 91 - 90 - 89 - 88 - 87 - 86 - 85 - 84 -
83 - 82 - 81 - 80 - 79 - 78 - 77 - 76 - 75 - 74 - 73 - 72 - 71 - 70 - 69 - 68 -
67 - 66 - 65 - 64 - 63 - 62 - 61 - 60 - 59 - 58 - 57 - 56 - 55 - 54 - 53 - 52 -
51 - 50 - 49 - 48 - 47 - 46 - 45 - 44 - 43 - 42 - 41 - 40 - 39 - 38 - 37 - 36 -
35 - 34 - 33 - 32 - 31 - 30 - 29 - 28 - 27 - 26 - 25 - 24 - 23 - 22 - 21 - 20 -
19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 -
1 - 0 -

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

O número de trocas é: 4950
O número de comparações foi: 9801
Process returned 0 (0x0)   execution time : 0.096 s
Press any key to continue.
-
```

Figura 2 - Execução do Bubble Sort com vetor decrescente ordenado.

Fonte: Elaborado pela autora

Executáveis do aleatório:

```
"C:\Users\Del\Documents\Técnicas e práticas de programação\bubblesort_sarah\bin\Debug\bub...
43 - 61 - 10 - 59 - 14 - 15 - 15 - 1 - 31 - 10 - 7 - 64 - 50 - 59 - 70 - 7 - 7 -
35 - 7 - 30 - 69 - 71 - 28 - 47 - 41 - 68 - 58 - 3 - 28 - 74 - 61 - 92 - 6 - 63 -
- 75 - 90 - 64 - 82 - 96 - 18 - 54 - 28 - 54 - 62 - 22 - 21 - 46 - 21 - 39 - 11 -
- 78 - 13 - 80 - 28 - 28 - 79 - 71 - 53 - 14 - 19 - 81 - 74 - 11 - 21 - 80 - 27 -
- 90 - 22 - 76 - 48 - 74 - 78 - 57 - 61 - 45 - 54 - 5 - 36 - 64 - 54 - 5 - 86 -
27 - 33 - 81 - 76 - 21 - 23 - 9 - 63 - 38 - 53 - 21 - 46 - 92 - 27 - 76 - 87 -
91 - 79 -

1 - 3 - 5 - 5 - 6 - 7 - 7 - 7 - 7 - 9 - 10 - 10 - 11 - 11 - 13 - 14 - 14 - 15 -
15 - 18 - 19 - 21 - 21 - 21 - 21 - 21 - 22 - 22 - 23 - 27 - 27 - 28 - 28 -
28 - 28 - 28 - 30 - 31 - 33 - 35 - 36 - 38 - 39 - 41 - 43 - 45 - 46 - 46 - 47 -
48 - 50 - 53 - 53 - 54 - 54 - 54 - 54 - 57 - 58 - 59 - 59 - 61 - 61 - 62 -
63 - 63 - 64 - 64 - 64 - 68 - 69 - 70 - 71 - 71 - 74 - 74 - 74 - 75 - 76 - 76 -
76 - 78 - 78 - 79 - 79 - 80 - 80 - 81 - 81 - 82 - 86 - 87 - 90 - 90 - 91 - 92 -
92 - 96 -

O número de trocas é: 2069
O número de comparações foi: 9801
Process returned 0 (0x0)   execution time : 0.088 s
Press any key to continue.
-
```

Figura 3 - Execução do Bubble Sort com vetor aleatório.

Fonte: Elaborado pela autora

```
"C:\Users\Del\Documents\Técnicas e práticas de programação\bubblesort_sarah\bin\Debug\bub...
16 - 83 - 51 - 84 - 32 - 76 - 32 - 23 - 3 - 40 - 51 - 37 - 60 - 3 - 26 - 9 - 80 -
9 - 45 - 56 - 19 - 13 - 8 - 76 - 7 - 49 - 81 - 28 - 65 - 29 - 97 - 47 - 89 - 8 -
6 - 29 - 26 - 9 - 76 - 69 - 94 - 2 - 26 - 39 - 75 - 70 - 77 - 20 - 73 - 16 - 92 -
- 95 - 28 - 81 - 35 - 66 - 14 - 83 - 95 - 14 - 55 - 48 - 52 - 94 - 62 - 53 - 63 -
- 11 - 74 - 67 - 42 - 58 - 87 - 18 - 70 - 54 - 65 - 12 - 68 - 55 - 48 - 51 - 18 -
- 94 - 66 - 18 - 20 - 11 - 70 - 12 - 38 - 85 - 39 - 51 - 4 - 85 - 56 - 84 - 12 -
21 - 24 -

2 - 3 - 3 - 4 - 7 - 8 - 9 - 9 - 9 - 11 - 11 - 12 - 12 - 12 - 13 - 14 - 14 - 16 -
16 - 18 - 18 - 18 - 19 - 20 - 20 - 21 - 23 - 24 - 26 - 26 - 26 - 28 - 28 - 29 -
29 - 32 - 32 - 35 - 37 - 38 - 39 - 39 - 40 - 42 - 45 - 47 - 48 - 48 - 49 - 51 -
51 - 51 - 51 - 52 - 53 - 54 - 55 - 55 - 56 - 56 - 58 - 60 - 62 - 63 - 65 - 65 -
66 - 66 - 67 - 68 - 69 - 70 - 70 - 70 - 73 - 74 - 75 - 76 - 76 - 76 - 77 - 80 -
81 - 81 - 83 - 83 - 84 - 84 - 85 - 85 - 86 - 87 - 89 - 92 - 94 - 94 - 94 - 95 -
95 - 97 -

O número de trocas é: 2371
O número de comparações foi: 9801
Process returned 0 (0x0)   execution time : 0.080 s
Press any key to continue.
-
```

Figura 4 - Execução do Bubble Sort com vetor aleatório.

Fonte: Elaborado pela autora

```
"C:\Users\Del...\Documents\Técnicas e práticas de programapão\bubblesort_sarah\bin\Debug\bub..."
52 - 25 - 62 - 1 - 66 - 80 - 11 - 76 - 27 - 29 - 19 - 10 - 75 - 35 - 97 - 26 - 6
1 - 10 - 13 - 16 - 6 - 8 - 12 - 45 - 27 - 75 - 4 - 44 - 28 - 89 - 40 - 99 - 53 -
27 - 60 - 61 - 68 - 77 - 88 - 1 - 84 - 88 - 84 - 50 - 34 - 64 - 59 - 19 - 2 - 1
1 - 74 - 58 - 76 - 87 - 71 - 85 - 43 - 85 - 95 - 53 - 81 - 22 - 51 - 54 - 44 - 7
5 - 6 - 24 - 91 - 51 - 45 - 85 - 78 - 19 - 65 - 15 - 92 - 98 - 87 - 37 - 75 -
22 - 33 - 19 - 63 - 37 - 21 - 79 - 31 - 23 - 1 - 16 - 63 - 39 - 4 - 44 - 28 - 3
8 - 81 -

1 - 1 - 1 - 2 - 4 - 4 - 5 - 6 - 6 - 7 - 8 - 10 - 10 - 11 - 11 - 12 - 13 - 15 - 1
6 - 16 - 19 - 19 - 19 - 19 - 21 - 22 - 22 - 23 - 24 - 25 - 26 - 27 - 27 - 27 - 2
8 - 28 - 29 - 31 - 33 - 34 - 35 - 37 - 37 - 38 - 39 - 40 - 43 - 44 - 44 - 44 - 4
5 - 45 - 50 - 51 - 51 - 52 - 53 - 53 - 54 - 58 - 59 - 60 - 61 - 61 - 62 - 63 - 6
3 - 64 - 65 - 66 - 68 - 71 - 74 - 75 - 75 - 76 - 76 - 77 - 78 - 79 - 80 - 8
1 - 81 - 84 - 84 - 85 - 85 - 85 - 87 - 87 - 88 - 88 - 89 - 91 - 92 - 95 - 97 - 9
8 - 99 -

O número de trocas é: 2383
O número de comparações foi: 9801
Process returned 0 (0x0)    execution time : 0.092 s
Press any key to continue.
-
```

Figura 5 - Execução do Bubble Sort com vetor aleatório.

Fonte: Elaborado pela autora

Executável do crescente:

```
"C:\Users\Del...\Documents\Técnicas e práticas de programapão\bubblesort_sarah\bin\Debug\bub..."
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 -
19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 -
35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 -
51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 -
67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 -
83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 -
99 - 100 -

1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 -
19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 -
35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 -
51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 -
67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 -
83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 -
99 - 100 -

O número de trocas é: 0
O número de comparações foi: 9801
Process returned 0 (0x0)    execution time : 0.084 s
Press any key to continue.
-
```

Figura 6 - Execução do Bubble Sort com vetor aleatório.

Fonte: Elaborado pela autora

## 1.4 – Implementação utilizando lista encadeada

```
#include <iostream>

#include <stdlib.h>                // Srand, rand

#include <time.h>

// Srand, rand


    using namespace std;


    int const TAM = 100;

    // Tamanho do vetor

    int comparacao;

    int troca;


    typedef struct bubble Tlista;


struct bubble
{
    int valor;

    Tlista *proximo;

};
```



```
// COMENTE AS FUNÇÕES PARA RODAR
```

```
/*
```

```
void aleatorio(Tlista *&listap)
```

```
{
```

```
    Tlista *auxiliar;
```

```
    for(int c=0; c<TAM; c++)
```

```
    {
```

```
        if(listap == NULL)
```

```
        {
```

```
            auxiliar = new(Tlista);
```

```
            auxiliar->valor = rand()%TAM;
```

```
            auxiliar->proximo = NULL;
```

```
            listap = auxiliar;
```

```
        }
```

```
    else
```

```
    {
```

```
        auxiliar = new(Tlista);
```

```
        auxiliar->valor = rand()%TAM;
```

```
        auxiliar->proximo = listap;
```

```
        listap = auxiliar;
```

```
    }
```

```
}
```

```
*/
```

```
void decrescente(Tlista *&listap)
```

```

{
    Tlista *auxiliar;

    for(int i=0; i<TAM; i++)
    {
        if(listap == NULL)
        {
            auxiliar = new(Tlista);

            auxiliar->valor = i;

            auxiliar->proximo = NULL;

            listap = auxiliar;
        }
        else
        {
            auxiliar = new(Tlista);

            auxiliar->valor = i;

            auxiliar->proximo = listap;

            listap = auxiliar;
        }
    }
}*/

void crescente(Tlista *&listap)
{
    Tlista *auxiliar;

    for(int i=TAM; i>0; i--)

```

```

    {
        if(listap == NULL)
        {
            auxiliar = new(Tlista);

            auxiliar->valor = i;

            auxiliar->proximo = NULL;

            listap = auxiliar;
        }
        else
        {
            auxiliar = new(Tlista);

            auxiliar->valor = i;

            auxiliar->proximo = listap;

            listap = auxiliar;
        }
    }
}

void exhibir(Tlista *listap)
{
    Tlista*auxiliar;

    auxiliar = listap;

    while(auxiliar != NULL)
    {

        cout << auxiliar->valor << " - ";
    }
}

```

```

        auxiliar = auxiliar->proximo;

    }

    cout << endl;
}

void bubblesort(Tlista*listap)
{
    Tlista *auxiliar, *segundo_auxiliar;

    int j;

    for(int i=0; i<TAM-1; i++)
    {

        auxiliar = listap;

        // auxiliazr recebe a lista
        segundo_auxiliar = listap->proximo;

        // recebe o próximo da lista

        for(int t=0; t<TAM-1; t++)

            // verifica a troca
            {

                comparacao = comparacao + 1;

                if(auxiliar->valor > segundo_auxiliar->valor)

```

```

        {

            j = segundo_auxiliar->valor;

            segundo_auxiliar->valor = auxiliar->valor;
            auxiliar->valor = j;
            troca = troca + 1;

        }

        auxiliar = auxiliar->proximo;
        segundo_auxiliar = segundo_auxiliar->proximo;
    }
}

int main()
{

    setlocale(LC_ALL, "portuguese");

    srand(time(NULL));

    Tlista *lista = NULL;

    crescente(lista);
    exhibir(lista);

    cout << endl;

```

```
bubblesort(lista);

exibir(lista);


cout << endl << "O número de trocas é: " << troca;

cout << endl << "O número de comparações foi: " << comparacao;


return 0;

}
```

## 2 – Insertion Sort (Ordenação por Inserção)

### 2.1 - Resumo do Método

Insertion sort, ou ordenação por inserção, consiste em um método que percorre um vetor, verificando os valores, ordenando os valores em suas posições corretas, um a um pegando o maior valor e realizando a troca dos valores até que ordene.

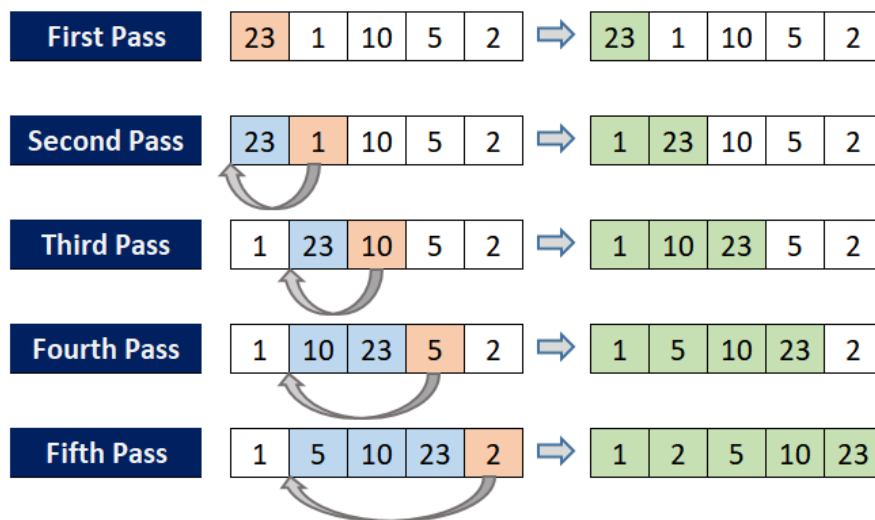


Figura 7 - Demonstração dos passos executados pelo Insertion Sort.

Fonte: <https://www.alphacodingskills.com>

### 2.2 - Complexidade

Para realizar o cálculo da complexidade é interessante ressaltar que o pior caso, seria em uma ordem contrária a que você quer, sendo assim, se eu for ordenar crescente um vetor de números decrescentes esse será o pior caso, pois cada vez que eu for olhar o valor seguinte será menor que os anteriores. Logo, teremos que fazer essa comparação dependendo de  $n$ , portanto temos uma complexidade de  $O(n^2)$ .

Dessa forma, obtemos que sua complexidade será:

Melhor caso -  $O(n)$  quando todos os elementos já estiverem nas respectivas posições.

Caso médio -  $O(n^2)$ .

Pior Caso -  $O(n^2)$  quando todos os elementos estiverem fora de suas respectivas posições

### 2.3 – Resultados

Tabela 1 - Lista com 10 elementos

Construção da Lista	Comparação	Troca
Em ordem crescente	9	0
Em ordem decrescente	9	45
Aleatório	9	Em média 18

Tabela 2 - Lista com 100 elementos

Construção da Lista	Comparação	Troca
Em ordem crescente	99	0
Em ordem decrescente	99	4950
Aleatório	99	Em média 2 541.67

Obs: Diante de valores diversos, sorteados pelo aleatório, foram realizadas três execuções, a fim de elaborar uma média aritmética com os valores de troca.

Executável do decrescente (10 elementos):

```

Os valores decrescentes são:

10 - 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 -
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -
Número de trocas: 45
Número de comparações: 9

Process returned 0 (0x0)   execution time : 0.052 s
Press any key to continue.

```

*Figura 8 - Execução do Insertion Sort em vetor decrescente de 10 elementos.*

*Fonte: Elaborado pela Autora*

Executável do decrescente (100 elementos):



```

Os valores decrescentes são:
100 - 99 - 98 - 97 - 96 - 95 - 94 - 93 - 92 - 91 - 90 - 89 - 88 - 87 - 86 - 85 -
84 - 83 - 82 - 81 - 80 - 79 - 78 - 77 - 76 - 75 - 74 - 73 - 72 - 71 - 70 - 69 -
68 - 67 - 66 - 65 - 64 - 63 - 62 - 61 - 60 - 59 - 58 - 57 - 56 - 55 - 54 - 53 -
52 - 51 - 50 - 49 - 48 - 47 - 46 - 45 - 44 - 43 - 42 - 41 - 40 - 39 - 38 - 37 -
36 - 35 - 34 - 33 - 32 - 31 - 30 - 29 - 28 - 27 - 26 - 25 - 24 - 23 - 22 - 21 -
20 - 19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 4 -
3 - 2 - 1 -

1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 -
19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 -
35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 -
51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 -
67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 -
83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 -
99 - 100 -

Número de trocas: 4950
Número de comparações: 99

```

*Figura 9 - Execução do Insertion Sort em vetor decrescente de 100 elementos.  
Fonte: Elaborado pela Autora*

Executáveis do aleatório (10 elementos):

```

Os valores aleatório são:
1 - 4 - 8 - 9 - 7 - 3 - 2 - 2 - 5 - 1 -

1 - 1 - 2 - 2 - 3 - 4 - 5 - 7 - 8 - 9 -

Número de trocas: 27
Número de comparações: 9

Process returned 0 (0x0)   execution time : 0.373 s
Press any key to continue.

```

*Figura 10 - Execução do Insertion Sort em vetor aleatório de 10 elementos.  
Fonte: Elaborado pela Autora*

```

Os valores aleatório são:
8 - 1 - 6 - 5 - 1 - 6 - 10 - 7 - 2 - 9 -

1 - 1 - 2 - 5 - 6 - 6 - 7 - 8 - 9 - 10 -

Número de trocas: 17
Número de comparações: 9

Process returned 0 (0x0)   execution time : 0.028 s
Press any key to continue.

```

*Figura 11 - Execução do Insertion Sort em vetor aleatório de 10 elementos.  
Fonte: Elaborado pela Autora*

```

Os valores aleatório são:
3 - 7 - 2 - 9 - 6 - 7 - 10 - 4 - 10 - 10 -

2 - 3 - 4 - 6 - 7 - 7 - 9 - 10 - 10 - 10 -
Número de trocas: 10
Número de comparações: 9
Process returned 0 (0x0)   execution time : 0.050 s
Press any key to continue.

```

Figura 12 - Execução do Insertion Sort em vetor aleatório de 10 elementos.  
Fonte: Elaborado pela Autora

Executáveis do aleatório (100 elementos):

```

Os valores aleatório são:
64 - 99 - 15 - 80 - 28 - 44 - 52 - 98 - 17 - 48 - 71 - 58 - 23 - 91 - 78 - 61 -
59 - 67 - 60 - 20 - 3 - 77 - 24 - 72 - 29 - 53 - 59 - 93 - 59 - 88 - 99 - 10 - 7
9 - 92 - 32 - 67 - 67 - 80 - 11 - 16 - 92 - 29 - 73 - 81 - 79 - 88 - 36 - 48 - 3
7 - 24 - 64 - 47 - 59 - 26 - 25 - 90 - 72 - 28 - 56 - 65 - 97 - 31 - 19 - 54 - 2
5 - 16 - 81 - 4 - 15 - 41 - 58 - 20 - 56 - 59 - 43 - 38 - 66 - 30 - 81 - 64 - 25
- 85 - 12 - 31 - 96 - 60 - 29 - 87 - 23 - 39 - 91 - 79 - 34 - 91 - 38 - 35 - 22
- 7 - 73 - 1 -

1 - 3 - 4 - 7 - 10 - 11 - 12 - 15 - 15 - 16 - 16 - 17 - 19 - 20 - 20 - 22 - 23 -
23 - 24 - 24 - 25 - 25 - 25 - 26 - 28 - 28 - 29 - 29 - 29 - 30 - 31 - 31 - 32 -
34 - 35 - 36 - 37 - 38 - 38 - 39 - 41 - 43 - 44 - 47 - 48 - 48 - 52 - 53 - 54 -
56 - 56 - 58 - 58 - 59 - 59 - 59 - 60 - 60 - 61 - 64 - 64 - 64 - 65 -
66 - 67 - 67 - 67 - 71 - 72 - 72 - 73 - 73 - 77 - 78 - 79 - 79 - 79 - 80 - 80 -
81 - 81 - 81 - 85 - 87 - 88 - 88 - 90 - 91 - 91 - 91 - 92 - 92 - 93 - 96 - 97 -
98 - 99 - 99 -

Número de trocas: 2675
Número de comparações: 99

```

Figura 13 - Execução do Insertion Sort em vetor aleatório de 100 elementos.  
Fonte: Elaborado pela Autora

```

Os valores aleatório são:
98 - 45 - 54 - 37 - 11 - 11 - 24 - 19 - 3 - 55 - 68 - 16 - 73 - 28 - 47 - 99 - 8
- 22 - 70 - 73 - 37 - 58 - 48 - 42 - 13 - 79 - 48 - 3 - 35 - 26 - 8 - 49 - 30 -
61 - 37 - 29 - 38 - 73 - 96 - 95 - 95 - 85 - 64 - 16 - 11 - 62 - 49 - 3 - 18 -
52 - 28 - 70 - 95 - 77 - 68 - 84 - 84 - 81 - 40 - 80 - 60 - 22 - 50 - 36 - 36 -
5 - 28 - 93 - 76 - 52 - 13 - 77 - 59 - 97 - 62 - 3 - 94 - 23 - 21 - 71 - 61 - 59
- 62 - 96 - 70 - 86 - 92 - 52 - 47 - 1 - 6 - 19 - 10 - 29 - 91 - 34 - 83 - 29 -
98 - 30 -

1 - 3 - 3 - 3 - 3 - 5 - 6 - 8 - 8 - 10 - 11 - 11 - 11 - 13 - 13 - 16 - 16 - 18 -
19 - 19 - 20 - 21 - 22 - 22 - 23 - 24 - 26 - 28 - 28 - 28 - 29 - 29 - 29 - 30 -
30 - 34 - 35 - 36 - 36 - 37 - 37 - 37 - 38 - 40 - 42 - 45 - 47 - 47 - 48 - 48 -
49 - 49 - 50 - 52 - 52 - 52 - 54 - 55 - 58 - 59 - 59 - 60 - 61 - 61 - 62 - 62 -
62 - 64 - 68 - 68 - 70 - 70 - 71 - 73 - 73 - 76 - 77 - 77 - 79 - 80 - 81 -
83 - 84 - 84 - 85 - 86 - 91 - 92 - 93 - 94 - 95 - 95 - 95 - 96 - 96 - 97 - 98 -
98 - 99 -

Número de trocas: 2232
Número de comparações: 99

```

Figura 14 - Execução do Insertion Sort em vetor aleatório de 100 elementos.  
Fonte: Elaborado pela Autora

```

Os valores aleatório são:
78 - 83 - 43 - 99 - 56 - 73 - 69 - 59 - 36 - 91 - 33 - 39 - 95 - 40 - 100 - 22 -
38 - 1 - 56 - 78 - 30 - 41 - 64 - 97 - 44 - 20 - 70 - 62 - 60 - 93 - 20 - 21 -
73 - 88 - 19 - 22 - 100 - 75 - 98 - 62 - 50 - 79 - 14 - 67 - 75 - 91 - 33 - 53 -
29 - 21 - 92 - 60 - 64 - 36 - 40 - 100 - 99 - 53 - 68 - 38 - 7 - 44 - 1 - 53 -
67 - 70 - 6 - 37 - 51 - 33 - 2 - 91 - 52 - 67 - 35 - 9 - 16 - 26 - 16 - 95 - 3 -
83 - 11 - 38 - 62 - 94 - 55 - 84 - 75 - 69 - 91 - 17 - 12 - 73 - 52 - 84 - 2 -
21 - 55 - 53 -

1 - 1 - 2 - 2 - 3 - 6 - 7 - 9 - 11 - 12 - 14 - 16 - 16 - 17 - 19 - 20 - 20 - 21 -
21 - 21 - 22 - 22 - 26 - 29 - 30 - 33 - 33 - 33 - 35 - 36 - 36 - 37 - 38 - 38 -
38 - 39 - 40 - 40 - 41 - 43 - 44 - 44 - 50 - 51 - 52 - 52 - 53 - 53 - 53 -
55 - 55 - 56 - 56 - 59 - 60 - 60 - 62 - 62 - 62 - 64 - 64 - 67 - 67 - 67 -
69 - 69 - 70 - 70 - 73 - 73 - 73 - 75 - 75 - 75 - 78 - 78 - 79 - 83 - 83 -
84 - 88 - 91 - 91 - 91 - 91 - 92 - 93 - 94 - 95 - 95 - 97 - 98 - 99 - 99 -
- 100 - 100 -

Número de trocas: 2718
Número de comparações: 99

```

Figura 15 - Execução do Insertion Sort em vetor aleatório de 100 elementos.  
Fonte: Elaborado pela Autora

Executável do crescente (10 elementos):

```

Os valores crescentes são:
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -

Número de trocas: 0
Número de comparações: 9

Process returned 0 (0x0)   execution time : 0.033 s
Press any key to continue.

```

Figura 16 - Execução do Insertion Sort em vetor crescente de 10 elementos.  
Fonte: Elaborado pela Autora

Executável do crescente (100 elementos):

```

Os valores crescentes são:
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

Número de trocas: 0
Número de comparações: 99

Process returned 0 (0x0)   execution time : 0.364 s

```

Figura 17 - Execução do Insertion Sort em vetor crescente de 100 elementos.  
Fonte: Elaborado pela Autora

## 2.4 – Implementação utilizando lista sequencial

```
#include <iostream>

#include <time.h>

#include <conio.h>

#include <stdlib.h>

// Aleatório


using namespace std;


int const TAM = 100;

// para alterar facilmente o tamanho do vetor

int comparacao = 0;

int troca = 0;

int vetor[TAM];


void insertion_sort(int vetor_[TAM])
{
    int anterior;

    int agora;

    int auxiliar;

    for(agora=1; agora<TAM; agora++)
    {
```

```

        comparacao++;

//Conta as comparações

        auxiliar = vetor_[agora];
        anterior = agora - 1;

        while((anterior>=0)&&(auxiliar<vetor_[anterior]))

            // compara e vê se é menor

                {

                    troca++;

                    vetor_[anterior+1] = vetor_[anterior];

                    anterior--;

                    //decremento

                }

            vetor_[anterior+1] = auxiliar;

            // Termina a troca se houver

        }

    }

void exibir(int vetor_[TAM])

{

    cout << endl;

```

```

        for(int i=0; i<TAM; i++){

            cout << vetor_[i] << " - ";

        }

    }

void valores_aleatorios (int vetor_[TAM])
{
    for(int a=0; a<TAM; a++){
        vetor_[a] = rand() % TAM + 1;

    }
}

void crescente(int vetor_[TAM])
{
    for(int i = 0; i < TAM; i++)
    {
        vetor_[i] = i;

        cout << vetor[i]<< " - ";

    }

}

void decrescente ( int vetor_[TAM] )

```

```

{

    for ( int i = 0; i < TAM; i++ ) {

        vetor_ [i] = TAM - i;

        cout << vetor[i] << " - " ;

    }

}

int main()

{

    setlocale(LC_ALL, "portuguese");

    srand (time(NULL));

    // comando de valores aleatórios


    //Comente para rodar cada vetor


    /**valores_aleatorios(vetor);

    cout << endl << "Os valores aleatório são: " << endl;

    exibir(vetor);

    insertion_sort(vetor);

    cout << endl;

    cout << endl;

    exibir(vetor);

    cout << endl;

```

```
cout << endl << "Número de trocas: " << troca;

cout << endl;

cout << endl << "Número de comparações: " << comparacao;

cout << endl;

cout << endl;/**/
```

```
cout << endl << " Os valores crescentes são:" << endl;

crescente(vetor);

insertion_sort(vetor);

cout << endl;

cout << endl;

exibir(vetor);

cout << endl;

cout << endl << "Número de trocas: " << troca;

cout << endl;

cout << endl << "Número de comparações: " << comparacao;

cout << endl;

cout << endl;
```

```
/*cout << endl << "Os valores decrescentes são: " << endl;

cout << endl;

cout << endl;

decrescente(vetor);

insertion_sort(vetor);

cout << endl;
```



```
    cout << endl;

    exibir(vetor);

    cout << endl;

    cout << endl << "Número de trocas: " << troca;

    cout << endl;

    cout << endl << "Número de comparações: " << comparacao;

    cout << endl;

    cout << endl;*/

    return 0;

}
```

### 3 – Selection Sort (Ordenação por Seleção)

#### 3.1 - Resumo do Método

Selection sort, ou ordenação por seleção, consiste em um método que percorre um vetor, verificando os valores, e enviando o menor valor (mínimo) para a primeira posição da lista, repetindo este processo ignorando o valor já ordenado, até que todos os valores estejam em suas respectivas posições corretas. Pela maneira como o método se comporta, é possível percebermos duas coisas, primeiramente que o mesmo separa a entrada em duas sub-listas, sendo a da esquerda já ordenada, enquanto a da direita ainda a ordenar. Ademais, que por verificar todos os elementos da entrada à cada varredura, até o último ser ordenado, este método pode se tornar ineficiente dependendo do tamanho da entrada, devido ao alto tempo de execução.

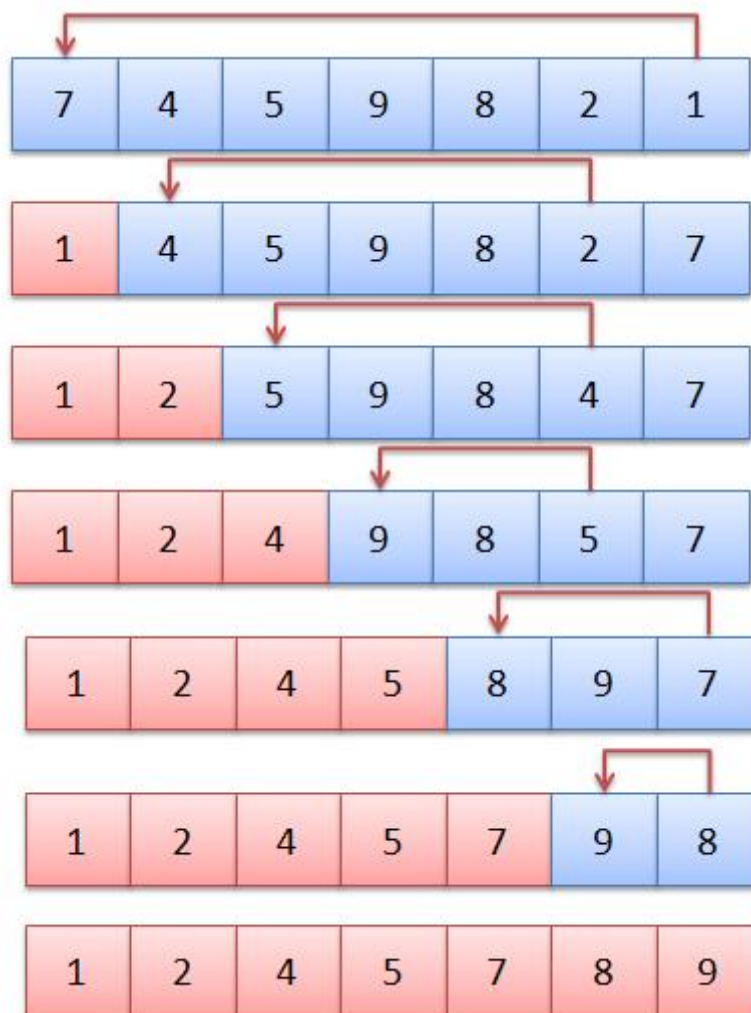


Figura 18 - Demonstração dos passos executados pelo Selection Sort.

Fonte: <http://nerds-attack.blogspot.com>

### 3.2 - Complexidade

Para realizar o cálculo da complexidade é interessante considerar quantas vezes será executado o for mais interno, pois ele que irá dominar o algoritmo.

$$1 + (n - 1) * [3 + X] = 1 + 3 * (n - 1) + X * (n - 1)$$
$$n + (n - 1) + (n - 2) \dots + 2$$

Obs:  $n - 1$  consiste na quantidade de vezes que vai rodar o for, vezes a quantidade de passos do for mais externo, porém como não sabemos utilizaremos  $x$ .

Sendo assim, a partir dessa P.A de razão menos um, temos:

$$(n + 2) * (n - 1) / 2 = n^2 + n - 2 / 2$$

Agora, juntamos com a anterior e teremos:

$$1 + 3(n - 1) + n^2 + n - 2 / 2 = (1 - 1 - 3) + (3n + n/2) + n^2/2$$

- Depende de  $n$  elevado a 0 =  $(1 - 1 - 3)$ ;
- Depende de  $n$  elevado a 1 =  $(3n + n/2)$ ;
- Depende de  $n$  ao quadrado =  $n^2/2$ ;

Dessa forma, entendemos que sua complexidade será  $O(n^2)$  para todos os casos.

### 3.3 – Resultados

Tabela 1 - Lista com 10 elementos

Construção da Lista	Comparação	Troca
Em ordem crescente	55	0
Em ordem decrescente	55	35
Aleatório	55	Em média 24.67

Tabela 2 - Lista com 100 elementos

Construção da Lista	Comparação	Troca
Em ordem crescente	5050	0

Em ordem decrescente	5050	2600
Aleatório	5050	Em média 446,67

Executável do decrescente (10 elementos):

```

10 - 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 -
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -

0 número de trocas foi: 35
0 número de comparações foi: 55
Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.

```

*Figura 19 - Execução do Selection Sort em vetor decrescente de 10 elementos.  
Fonte: Elaborado pela Autora*

Executável do decrescente (100 elementos):

```

100 - 99 - 98 - 97 - 96 - 95 - 94 - 93 - 92 - 91 - 90 - 89 - 88 - 87 - 86 - 85 -
84 - 83 - 82 - 81 - 80 - 79 - 78 - 77 - 76 - 75 - 74 - 73 - 72 - 71 - 70 - 69 -
68 - 67 - 66 - 65 - 64 - 63 - 62 - 61 - 60 - 59 - 58 - 57 - 56 - 55 - 54 - 53 -
52 - 51 - 50 - 49 - 48 - 47 - 46 - 45 - 44 - 43 - 42 - 41 - 40 - 39 - 38 - 37 -
36 - 35 - 34 - 33 - 32 - 31 - 30 - 29 - 28 - 27 - 26 - 25 - 24 - 23 - 22 - 21 -
20 - 19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 4 -
3 - 2 - 1 -

1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 -
19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 -
35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 -
51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 -
67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 -
83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 -
99 - 100 -

0 número de trocas foi: 2600
0 número de comparações foi: 5050
Process returned 0 (0x0)   execution time : 0.105 s

```

*Figura 20 - Execução do Selection Sort em vetor decrescente de 100 elementos.  
Fonte: Elaborado pela Autora*

Executáveis do aleatório (10 elementos):

```

2 - 2 - 1 - 7 - 4 - 6 - 6 - 6 - 2 - 8 -
1 - 2 - 2 - 2 - 4 - 6 - 6 - 6 - 7 - 8 -

o número de trocas foi: 21
o número de comparações foi: 55
Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.

```

*Figura 21 - Execução do Selection Sort em vetor aleatório de 10 elementos.  
Fonte: Elaborado pela Autora*

```

5 - 1 - 5 - 8 - 10 - 8 - 6 - 10 - 4 - 4 -
1 - 4 - 4 - 5 - 5 - 6 - 8 - 8 - 10 - 10 -

o número de trocas foi: 24
o número de comparações foi: 55
Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.

```

*Figura 22 - Execução do Selection Sort em vetor aleatório de 10 elementos.  
Fonte: Elaborado pela Autora*

```

9 - 5 - 3 - 9 - 1 - 3 - 8 - 9 - 9 - 9 -
1 - 3 - 3 - 5 - 8 - 9 - 9 - 9 - 9 - 9 -

o número de trocas foi: 29
o número de comparações foi: 55
Process returned 0 (0x0)   execution time : 0.024 s
Press any key to continue.

```

*Figura 23 - Execução do Selection Sort em vetor aleatório de 10 elementos.  
Fonte: Elaborado pela Autora*

Executáveis do Aleatório (100 elementos):

```

27 - 28 - 20 - 47 - 11 - 14 - 29 - 38 - 4 - 3 - 69 - 66 - 39 - 22 - 34 - 11 - 71
- 49 - 34 - 72 - 37 - 37 - 16 - 52 - 41 - 82 - 48 - 58 - 52 - 80 - 70 - 17 - 72
- 12 - 93 - 24 - 20 - 3 - 47 - 38 - 9 - 48 - 51 - 93 - 46 - 60 - 73 - 60 - 66 -
91 - 49 - 50 - 3 - 48 - 91 - 69 - 89 - 1 - 70 - 48 - 40 - 73 - 67 - 52 - 99 - 5
4 - 88 - 61 - 81 - 10 - 56 - 75 - 45 - 24 - 30 - 59 - 50 - 71 - 74 - 48 - 75 - 3
- 79 - 85 - 57 - 50 - 41 - 88 - 76 - 73 - 56 - 83 - 94 - 83 - 22 - 47 - 4 - 90
- 18 - 38 -

1 - 3 - 3 - 3 - 3 - 4 - 4 - 9 - 10 - 11 - 11 - 12 - 14 - 16 - 17 - 18 - 20 - 20
- 22 - 22 - 24 - 24 - 27 - 28 - 29 - 30 - 34 - 34 - 37 - 37 - 38 - 38 - 38 - 39
- 40 - 41 - 41 - 45 - 46 - 47 - 47 - 47 - 48 - 48 - 48 - 48 - 49 - 49 - 50
- 50 - 50 - 51 - 52 - 52 - 52 - 54 - 56 - 56 - 57 - 58 - 59 - 60 - 60 - 61 - 66
- 66 - 67 - 69 - 69 - 70 - 70 - 71 - 71 - 72 - 72 - 73 - 73 - 73 - 74 - 75 - 75
- 76 - 79 - 80 - 81 - 82 - 83 - 83 - 85 - 88 - 88 - 89 - 90 - 91 - 91 - 93 - 93
- 94 - 99 -

o número de trocas foi: 428
o número de comparações foi: 5050
Process returned 0 (0x0)   execution time : 0.088 s
Press any key to continue.

```

Figura 24 - Execução do Selection Sort em vetor aleatório de 100 elementos.  
Fonte: Elaborado pela Autora

```

35 - 78 - 14 - 4 - 43 - 24 - 35 - 42 - 31 - 25 - 88 - 19 - 26 - 95 - 79 - 12 - 2
9 - 89 - 12 - 54 - 100 - 20 - 85 - 57 - 87 - 74 - 2 - 68 - 40 - 59 - 98 - 34 - 4
4 - 75 - 62 - 86 - 24 - 31 - 14 - 50 - 9 - 70 - 41 - 56 - 17 - 84 - 44 - 90 - 93
- 38 - 27 - 9 - 92 - 57 - 80 - 32 - 85 - 64 - 100 - 22 - 56 - 79 - 84 - 96 - 16
- 26 - 15 - 95 - 81 - 52 - 84 - 49 - 28 - 52 - 74 - 90 - 84 - 45 - 78 - 54 - 20
- 70 - 41 - 97 - 4 - 1 - 98 - 95 - 80 - 87 - 74 - 49 - 94 - 88 - 39 - 43 - 28 -
99 - 58 - 94 -

1 - 2 - 4 - 4 - 9 - 9 - 12 - 12 - 14 - 14 - 15 - 16 - 17 - 19 - 20 - 20 - 22 - 2
4 - 24 - 25 - 26 - 26 - 27 - 28 - 28 - 29 - 31 - 31 - 32 - 34 - 35 - 35 - 38 - 3
9 - 40 - 41 - 41 - 42 - 43 - 43 - 44 - 44 - 45 - 49 - 49 - 50 - 52 - 52 - 54 - 5
4 - 56 - 56 - 57 - 57 - 58 - 59 - 62 - 64 - 68 - 70 - 70 - 74 - 74 - 74 - 75 - 7
8 - 78 - 79 - 79 - 80 - 80 - 81 - 84 - 84 - 84 - 84 - 85 - 85 - 86 - 87 - 87 - 8
8 - 88 - 89 - 90 - 90 - 92 - 93 - 94 - 94 - 95 - 95 - 95 - 96 - 97 - 98 - 98 - 9
9 - 100 - 100 -

o número de trocas foi: 444
o número de comparações foi: 5050
Process returned 0 (0x0)   execution time : 0.076 s
Press any key to continue.

```

Figura 25 - Execução do Selection Sort em vetor aleatório de 100 elementos.  
Fonte: Elaborado pela Autora

```

100 - 70 - 49 - 51 - 27 - 55 - 48 - 32 - 82 - 35 - 30 - 100 - 19 - 42 - 71 - 26
- 1 - 46 - 58 - 96 - 34 - 27 - 70 - 77 - 35 - 73 - 89 - 1 - 23 - 44 - 80 - 52 - 9
20 - 37 - 83 - 66 - 71 - 11 - 91 - 50 - 90 - 44 - 1 - 18 - 57 - 61 - 99 - 42 - 9
6 - 55 - 45 - 78 - 74 - 72 - 65 - 65 - 15 - 10 - 70 - 11 - 85 - 29 - 27 - 22 - 2
1 - 83 - 92 - 65 - 93 - 39 - 16 - 5 - 92 - 74 - 82 - 49 - 22 - 22 - 25 - 48 - 72
- 84 - 65 - 28 - 83 - 17 - 79 - 73 - 57 - 63 - 20 - 58 - 83 - 61 - 39 - 43 - 2
- 16 - 55 - 8 -

1 - 1 - 1 - 2 - 5 - 8 - 10 - 11 - 11 - 15 - 16 - 16 - 17 - 18 - 19 - 20 - 20 - 2
1 - 22 - 22 - 22 - 23 - 25 - 26 - 27 - 27 - 27 - 28 - 29 - 30 - 32 - 34 - 35 - 3
5 - 37 - 39 - 39 - 42 - 42 - 43 - 44 - 44 - 45 - 46 - 48 - 48 - 49 - 49 - 50 - 5
1 - 52 - 55 - 55 - 55 - 57 - 57 - 58 - 58 - 61 - 61 - 63 - 65 - 65 - 65 - 6
6 - 70 - 70 - 70 - 71 - 71 - 72 - 72 - 73 - 73 - 74 - 74 - 77 - 78 - 79 - 80 - 8
2 - 82 - 83 - 83 - 83 - 84 - 85 - 89 - 90 - 91 - 92 - 92 - 93 - 96 - 96 - 9
9 - 100 - 100 -

o número de trocas foi: 468
o número de comparações foi: 5050
Process returned 0 (0x0)   execution time : 0.081 s
Press any key to continue.

```

Figura 26 - Execução do Selection Sort em vetor aleatório de 100 elementos.  
Fonte: Elaborado pela Autora

Executável do crescente (10 elementos):

```

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -
o número de trocas foi: 10
o número de comparações foi: 55
Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.

```

*Figura 27 - Execução do Selection Sort em vetor crescente de 10 elementos.  
Fonte: Elaborado pela Autora*

Executável do crescente (100 elementos):

```

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

o número de trocas foi: 100
o número de comparações foi: 5050
Process returned 0 (0x0)   execution time : 0.356 s
Press any key to continue.

```

*Figura 28 - Execução do Selection Sort em vetor crescente de 100 elementos.  
Fonte: Elaborado pela Autora*

### 3.4 – Implementação utilizando lista sequencial

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
// Aleatório

using namespace std;

const int TAM = 100;
// Para mudar o tamanho do vetor facilmente, para rodar com 100.
int vetor[TAM];
int trocap = 0;
int comparacao = 0;

void iniciar(int vetor_[TAM])
{
    for(int v = 0; v < TAM; v++)
    {
        vetor_[v] = TAM - v;
    }
}

void valores_aleatorios (int vetor_[TAM])
{
    for(int a=0; a<TAM; a++){
        vetor_[a] = rand() % TAM + 1;
    }
}

void exibir(int vetor_[TAM])
// exibe e também é o vetor decrescente
{
    for(int v = 0; v < TAM; v++)
    {
        cout << vetor_[v] << " - ";
    }
    cout << endl;
}

void crescente(int vetor_[TAM])
{
    for(int v = 0; v < TAM; v++)
    {
        vetor_[v] = v;
        cout << vetor[v]<< " - ";
    }
}
```



```

void selectionsort(int vetor_[TAM])
{
    int posicao_m;
    int troca;
    //auxiliar de troca
    int v;
    int p;
    // Posições

    for(p = TAM-1; p >= 0 ; p--)
    {
        posicao_m = p;
        for(v = p ; v >= 0 ; v--)
        {
            comparacao = comparacao + 1;
            // Contagem de comparação em cima do if, pois ele compara
            mesmo que não entre no if
            if(vetor_[posicao_m] <= vetor_[v])
            {

                posicao_m = v;
                trocap = trocap + 1;
            }
        }
        troca = vetor_[p];
        vetor_[p] = vetor_[posicao_m];
        vetor_[posicao_m] = troca;
    }
}

int main()
{

    setlocale(LC_ALL, "portuguese");

    srand (time(NULL));

    // Comente para utilizar cada vetor, ou seja, se for usar um comente os
    demais!

    //aleatório

    /**iniciar(vetor);
    valores_aleatorios(vetor);
    exibir(vetor);
    cout << endl;
    cout << endl;
    selectionsort(vetor);
    cout << endl;

```

```

        exibir(vetor);
        cout << endl;
    cout << endl << " o número de trocas foi: " << trocap << endl;
    cout << endl << " o número de comparações foi: " << comparacao << endl;

    //decrecente

    iniciar(vetor);
    exibir(vetor);
    cout << endl;
    cout << endl;
    selectionsort(vetor);
    cout << endl;
    exibir(vetor);
    cout << endl;
    cout << endl;
    cout << endl << " 0 número de trocas foi: " << trocap;
    cout << endl;
    cout << endl;
    cout << " 0 número de comparações foi: " << comparacao << endl;*/

    // crescente

    iniciar(vetor);
    crescente(vetor);
    cout << endl;
    cout << endl;
    selectionsort(vetor);
    cout << endl;
    exibir(vetor);
    cout << endl << " o número de trocas foi: " << trocap << endl;
    cout << endl << " o número de comparações foi: " << comparacao <<
endl;

    return 0;
}

```

## 4 – Shell Sort (Ordenação por Incrementos Diminutos)

### 4.1 - Resumo do Método

Shell sort, ou ordenação por incrementos diminutos, consiste em um método derivado do Insertion Sort, que percorre um vetor, verificando os valores, através do gap ou “h”. O algoritmo verifica os elementos em cada ponta do gap, trocando-os caso o elemento da direita seja maior que seu comparativo, e então segue para o próximo elemento até que chegue ao final da entrada, o algoritmo então diminui o gap e re-executa o código, até que o gap seja 1, indicando a ordenação completa da entrada quando o mesmo for concluído. Diferentemente de seu antecessor (Insertion Sort). O Shell Sort, por sua vez, trabalha de maneira incrivelmente efetiva para entradas de larga escala.

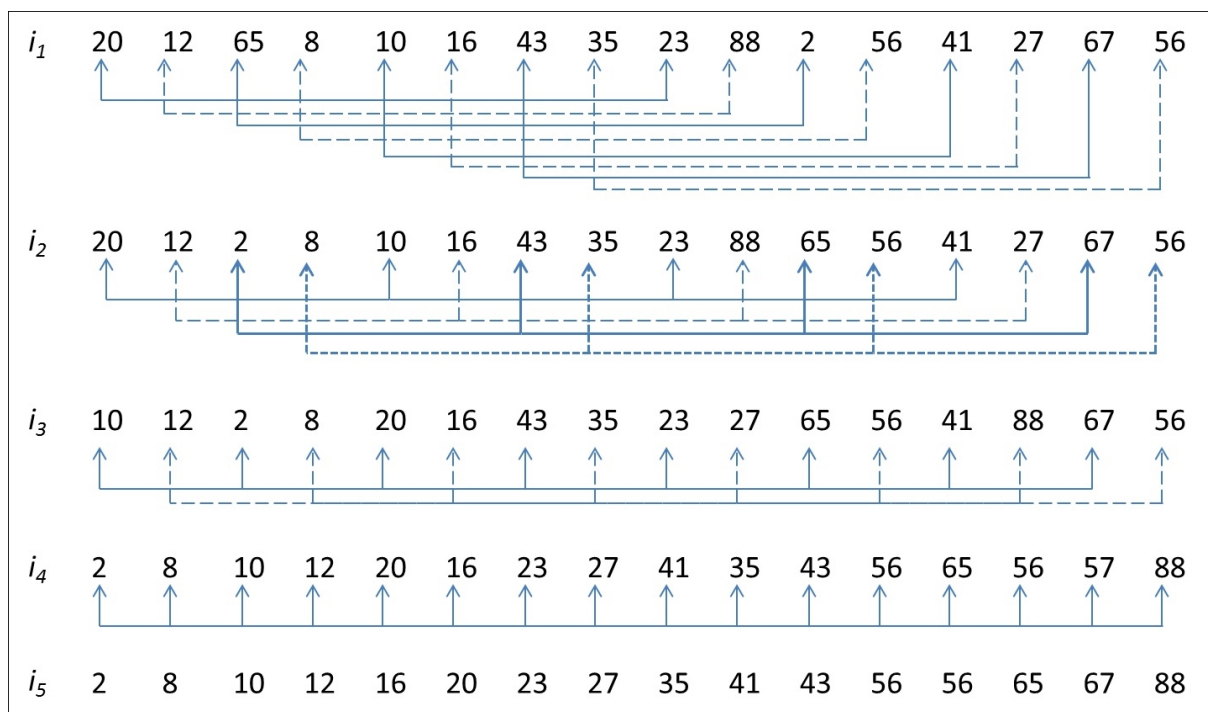


Figura 29 - Demonstração dos passos executados pelo Shell Sort. Gap  $N/2$ ,  $N/4$ ,  $N/8$ , 1.

Fonte: <https://subscription.packtpub.com>

### 4.2 - Complexidade

A complexidade do Shell sort, depende da sequência de passos visto a entrada. Sendo assim, depende do gap e terá uma complexidade de  $O(n \log^2 n)$  para todos os casos.

### 4.3 – Resultados

Tabela 1 - Lista com 100 elementos. Gap = 1

Construção da Lista	Comparação	Troca
Em ordem crescente	187	0
Em ordem decrescente	187	646
Aleatório	187	Em média 894

Tabela 2 - Lista com 100 elementos. Gap = TAM/2

Construção da Lista	Comparação	Troca
Em ordem crescente	189	0
Em ordem decrescente	189	900
Aleatório	189	Em média 927

Tabela 3 - Lista com 100 elementos. Gap = TAM/2

Construção da Lista	Comparação	Troca
Em ordem crescente	189	0
Em ordem decrescente	185	778
Aleatório	189	Em média 994.33

Executável do decrescente (Gap = 1)

```

100 - 99 - 98 - 97 - 96 - 95 - 94 - 93 - 92 - 91 - 90 - 89 - 88 - 87 - 86 - 85 -
84 - 83 - 82 - 81 - 80 - 79 - 78 - 77 - 76 - 75 - 74 - 73 - 72 - 71 - 70 - 69 -
68 - 67 - 66 - 65 - 64 - 63 - 62 - 61 - 60 - 59 - 58 - 57 - 56 - 55 - 54 - 53 -
52 - 51 - 50 - 49 - 48 - 47 - 46 - 45 - 44 - 43 - 42 - 41 - 40 - 39 - 38 - 37 -
36 - 35 - 34 - 33 - 32 - 31 - 30 - 29 - 28 - 27 - 26 - 25 - 24 - 23 - 22 - 21 -
20 - 19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 4 -
3 - 2 - 1 -
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 -
19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 -
35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 -
51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 -
67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 -
83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 -
99 - 100 -

0 número de comparações foi: 187
0 número de trocas foi: 646
Process returned 0 (0x0)   execution time : 0.084 s
Press any key to continue.

```

*Figura 30 - Execução do Shell Sort em vetor decrescente de 100 elementos com Gap = 1.*

*Fonte: Elaborado pela Autora*

Executáveis do aleatório (Gap = 1):

```

17 - 13 - 50 - 20 - 40 - 82 - 56 - 88 - 79 - 36 - 67 - 55 - 45 - 77 - 52 - 30 -
59 - 56 - 49 - 83 - 16 - 39 - 54 - 41 - 83 - 28 - 2 - 92 - 88 - 81 - 9 - 66 - 2 -
45 - 2 - 26 - 9 - 59 - 76 - 54 - 76 - 21 - 36 - 34 - 20 - 50 - 58 - 75 - 37 -
31 - 30 - 37 - 20 - 8 - 5 - 64 - 67 - 17 - 44 - 76 - 41 - 74 - 34 - 8 - 49 - 48 -
5 - 33 - 32 - 77 - 63 - 36 - 44 - 28 - 90 - 97 - 92 - 73 - 17 - 64 - 61 - 37 -
8 - 88 - 20 - 41 - 54 - 86 - 29 - 7 - 86 - 76 - 30 - 42 - 37 - 75 - 57 - 15 - 6
9 - 62 -

2 - 2 - 2 - 5 - 5 - 7 - 8 - 8 - 8 - 9 - 9 - 13 - 15 - 16 - 17 - 17 - 17 - 20 - 2
0 - 20 - 20 - 21 - 26 - 28 - 28 - 29 - 30 - 30 - 30 - 31 - 32 - 33 - 34 - 34 - 3
6 - 36 - 36 - 37 - 37 - 37 - 37 - 39 - 40 - 41 - 41 - 41 - 42 - 44 - 44 - 45 - 4
5 - 48 - 49 - 49 - 50 - 50 - 52 - 54 - 54 - 54 - 55 - 56 - 56 - 57 - 58 - 59 - 5
9 - 61 - 62 - 63 - 64 - 64 - 66 - 67 - 67 - 69 - 73 - 74 - 75 - 75 - 76 - 76 - 7
6 - 76 - 77 - 77 - 79 - 81 - 82 - 83 - 83 - 86 - 86 - 88 - 88 - 88 - 90 - 92 - 9
2 - 97 -

0 número de comparações foi: 187
0 número de trocas foi: 904
Process returned 0 (0x0)   execution time : 0.084 s
Press any key to continue.

```

Figura 31 - Execução do Shell Sort em vetor aleatório de 100 elementos com Gap = 1.  
Fonte: Elaborado pela Autora

```

85 - 46 - 52 - 64 - 68 - 62 - 33 - 27 - 42 - 15 - 39 - 67 - 25 - 87 - 70 - 62 -
78 - 97 - 28 - 94 - 83 - 92 - 54 - 14 - 11 - 12 - 31 - 11 - 11 - 89 - 10 - 43 -
67 - 59 - 77 - 96 - 40 - 39 - 39 - 7 - 7 - 46 - 77 - 3 - 62 - 51 - 37 - 83 - 1 -
67 - 66 - 62 - 83 - 70 - 72 - 83 - 42 - 93 - 87 - 24 - 61 - 74 - 30 - 31 - 9 -
44 - 54 - 90 - 46 - 12 - 23 - 10 - 46 - 35 - 10 - 87 - 30 - 65 - 83 - 27 - 80 -
5 - 1 - 77 - 64 - 41 - 68 - 15 - 19 - 57 - 58 - 33 - 62 - 99 - 49 - 96 - 39 - 13
- 8 - 43 -

1 - 1 - 3 - 5 - 7 - 7 - 8 - 9 - 10 - 10 - 10 - 11 - 11 - 11 - 12 - 12 - 13 - 14
- 15 - 15 - 19 - 23 - 24 - 25 - 27 - 27 - 28 - 30 - 30 - 31 - 31 - 33 - 33 - 35
- 37 - 39 - 39 - 39 - 39 - 40 - 41 - 42 - 42 - 43 - 43 - 44 - 46 - 46 - 46 -
49 - 51 - 52 - 54 - 54 - 57 - 58 - 59 - 61 - 62 - 62 - 62 - 62 - 64 - 64
- 65 - 66 - 67 - 67 - 67 - 68 - 68 - 70 - 72 - 74 - 77 - 77 - 77 - 78 - 80
- 83 - 83 - 83 - 83 - 83 - 85 - 87 - 87 - 87 - 89 - 90 - 92 - 93 - 94 - 96 - 96
- 97 - 99 -

0 número de comparações foi: 187
0 número de trocas foi: 901
Process returned 0 (0x0)   execution time : 0.081 s
Press any key to continue.

```

Figura 32 - Execução do Shell Sort em vetor aleatório de 100 elementos com Gap = 1.  
Fonte: Elaborado pela Autora

```

32 - 4 - 40 - 69 - 71 - 57 - 21 - 72 - 99 - 64 - 51 - 49 - 34 - 8 - 9 - 44 - 72
- 64 - 7 - 28 - 93 - 38 - 93 - 91 - 85 - 37 - 43 - 52 - 43 - 13 - 81 - 25 - 71 -
39 - 30 - 66 - 45 - 69 - 87 - 8 - 55 - 53 - 97 - 56 - 90 - 41 - 94 - 58 - 44 -
11 - 88 - 2 - 28 - 22 - 86 - 66 - 43 - 76 - 12 - 6 - 61 - 37 - 93 - 51 - 3 - 97
- 28 - 33 - 32 - 65 - 96 - 80 - 41 - 76 - 79 - 13 - 33 - 11 - 89 - 70 - 29 - 21
- 72 - 54 - 100 - 9 - 42 - 42 - 43 - 37 - 70 - 77 - 3 - 88 - 92 - 80 - 55 - 65 -
68 - 93 -

2 - 3 - 3 - 4 - 6 - 7 - 8 - 8 - 9 - 9 - 11 - 11 - 12 - 13 - 13 - 21 - 21 - 22 -
25 - 28 - 28 - 28 - 29 - 30 - 32 - 32 - 33 - 33 - 34 - 37 - 37 - 37 - 38 - 39 -
40 - 41 - 41 - 42 - 42 - 43 - 43 - 43 - 44 - 44 - 45 - 49 - 51 - 51 - 52 -
53 - 54 - 55 - 55 - 56 - 57 - 58 - 61 - 64 - 64 - 65 - 65 - 66 - 66 - 68 - 69 -
69 - 70 - 70 - 71 - 71 - 72 - 72 - 72 - 76 - 76 - 77 - 79 - 80 - 80 - 81 - 85 -
86 - 87 - 88 - 88 - 89 - 90 - 91 - 92 - 93 - 93 - 93 - 93 - 94 - 96 - 97 - 97 -
99 - 100 -

0 número de comparações foi: 187
0 número de trocas foi: 877
Process returned 0 (0x0)   execution time : 0.076 s
Press any key to continue.

```

Figura 33 - Execução do Shell Sort em vetor aleatório de 100 elementos com Gap = 1.  
Fonte: Elaborado pela Autora

Executável do crescente (Gap = 1)

```

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

0 número de comparações foi: 187
0 número de trocas foi: 0
Process returned 0 (0x0)   execution time : 0.084 s
Press any key to continue.

```

Figura 34 - Execução do Shell Sort em vetor crescente de 100 elementos com Gap = 1.  
Fonte: Elaborado pela Autora

Executável do decrescente (Gap = TAM/2):

```

100 - 99 - 98 - 97 - 96 - 95 - 94 - 93 - 92 - 91 - 90 - 89 - 88 - 87 - 86 - 85 -
84 - 83 - 82 - 81 - 80 - 79 - 78 - 77 - 76 - 75 - 74 - 73 - 72 - 71 - 70 - 69 -
68 - 67 - 66 - 65 - 64 - 63 - 62 - 61 - 60 - 59 - 58 - 57 - 56 - 55 - 54 - 53 -
52 - 51 - 50 - 49 - 48 - 47 - 46 - 45 - 44 - 43 - 42 - 41 - 40 - 39 - 38 - 37 -
36 - 35 - 34 - 33 - 32 - 31 - 30 - 29 - 28 - 27 - 26 - 25 - 24 - 23 - 22 - 21 -
20 - 19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 4 -
3 - 2 - 1 -

1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 -
19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 -
35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 -
51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 -
67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 -
83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 -
99 - 100 -

0 número de comparações foi: 189
0 número de trocas foi: 900
Process returned 0 (0x0)   execution time : 0.085 s
Press any key to continue.

```

Figura 35 - Execução do Shell Sort em vetor decrescente de 100 elementos, Gap = TAM/2.  
Fonte: Elaborado pela Autora

Executáveis do aleatório (Gap = TAM/2)

```

51 - 36 - 15 - 92 - 58 - 100 - 86 - 14 - 86 - 64 - 64 - 67 - 12 - 70 - 70 - 41 -
41 - 14 - 96 - 51 - 88 - 59 - 20 - 13 - 63 - 26 - 95 - 4 - 11 - 16 - 14 - 80 -
44 - 57 - 84 - 59 - 5 - 45 - 80 - 79 - 76 - 22 - 28 - 56 - 49 - 62 - 21 - 16 - 4 -
0 - 75 - 94 - 87 - 20 - 64 - 45 - 43 - 18 - 25 - 13 - 51 - 50 - 74 - 89 - 21 - 3 -
2 - 88 - 33 - 22 - 49 - 72 - 67 - 41 - 88 - 20 - 71 - 36 - 64 - 99 - 26 - 26 - 7 -
9 - 93 - 56 - 83 - 57 - 75 - 52 - 50 - 9 - 96 - 6 - 48 - 52 - 100 - 7 - 42 - 92 -
- 24 - 35 - 28 -

4 - 5 - 6 - 7 - 9 - 11 - 12 - 13 - 13 - 14 - 14 - 14 - 15 - 16 - 16 - 18 - 20 -
20 - 20 - 21 - 21 - 22 - 22 - 24 - 25 - 26 - 26 - 26 - 28 - 28 - 32 - 33 - 35 -
36 - 36 - 40 - 41 - 41 - 41 - 42 - 43 - 44 - 45 - 45 - 48 - 49 - 49 - 50 - 50 -
51 - 51 - 51 - 52 - 52 - 56 - 56 - 57 - 57 - 58 - 59 - 62 - 63 - 64 - 64 -
64 - 64 - 67 - 67 - 70 - 70 - 71 - 72 - 74 - 75 - 75 - 76 - 79 - 79 - 80 - 80 -
83 - 84 - 86 - 86 - 87 - 88 - 88 - 88 - 89 - 92 - 92 - 93 - 94 - 95 - 96 - 96 -
99 - 100 - 100 -

0 número de comparações foi: 189
0 número de trocas foi: 940
Process returned 0 (0x0)   execution time : 0.075 s
Press any key to continue.

```

Figura 36 - Execução do Shell Sort em vetor aleatório de 100 elementos, Gap = TAM/2.  
Fonte: Elaborado pela Autora

```

79 - 8 - 90 - 89 - 11 - 71 - 67 - 55 - 78 - 65 - 96 - 85 - 60 - 16 - 29 - 98 - 6
7 - 2 - 58 - 9 - 40 - 24 - 74 - 38 - 55 - 34 - 28 - 95 - 21 - 34 - 47 - 13 - 98
- 62 - 29 - 75 - 76 - 56 - 21 - 85 - 100 - 37 - 49 - 65 - 83 - 69 - 69 - 18 - 42
- 71 - 78 - 86 - 37 - 10 - 97 - 98 - 82 - 82 - 91 - 79 - 40 - 8 - 45 - 71 - 38
- 100 - 52 - 76 - 42 - 70 - 83 - 29 - 77 - 39 - 12 - 14 - 17 - 60 - 31 - 1 - 60
- 19 - 72 - 94 - 99 - 85 - 84 - 66 - 23 - 43 - 87 - 52 - 23 - 65 - 53 - 32 - 78
- 4 - 1 - 81 -

1 - 1 - 2 - 4 - 8 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 16 - 17 - 18 - 19 - 21 - 21
- 23 - 23 - 24 - 28 - 29 - 29 - 31 - 32 - 34 - 34 - 37 - 37 - 38 - 38 - 39
- 40 - 40 - 42 - 42 - 43 - 45 - 47 - 49 - 52 - 52 - 53 - 55 - 55 - 56 - 58 - 60
- 60 - 60 - 62 - 65 - 65 - 65 - 66 - 67 - 67 - 69 - 69 - 70 - 71 - 71 - 71 - 72
- 74 - 75 - 76 - 76 - 77 - 78 - 78 - 78 - 79 - 79 - 81 - 82 - 82 - 83 - 84
- 85 - 85 - 85 - 86 - 87 - 89 - 90 - 91 - 94 - 95 - 96 - 97 - 98 - 98 - 98 - 99
- 100 - 100 -

0 número de comparações foi: 189
0 número de trocas foi: 1009
Process returned 0 (0x0)   execution time : 0.079 s
Press any key to continue.

```

Figura 37 - Execução do Shell Sort em vetor aleatório de 100 elementos, Gap = TAM/2.  
Fonte: Elaborado pela Autora

```

80 - 61 - 23 - 88 - 82 - 27 - 49 - 13 - 17 - 83 - 11 - 50 - 38 - 33 - 58 - 91 -
57 - 28 - 74 - 10 - 41 - 81 - 82 - 15 - 63 - 77 - 33 - 21 - 91 - 79 - 72 - 92 -
76 - 55 - 6 - 86 - 68 - 90 - 99 - 79 - 59 - 63 - 39 - 2 - 34 - 65 - 81 - 13 - 55
- 14 - 87 - 64 - 27 - 16 - 89 - 41 - 89 - 82 - 94 - 58 - 53 - 49 - 5 - 35 - 15
- 3 - 4 - 70 - 9 - 60 - 81 - 88 - 46 - 47 - 34 - 30 - 33 - 51 - 58 - 19 - 93 - 1
1 - 40 - 8 - 50 - 95 - 84 - 57 - 80 - 80 - 84 - 26 - 78 - 73 - 93 - 42 - 50 - 79
- 90 - 47 -

2 - 3 - 4 - 5 - 6 - 8 - 9 - 10 - 11 - 11 - 13 - 13 - 14 - 15 - 15 - 16 - 17 - 19
- 21 - 23 - 26 - 27 - 27 - 28 - 30 - 33 - 33 - 33 - 34 - 34 - 35 - 38 - 39 - 40
- 41 - 41 - 42 - 46 - 47 - 47 - 49 - 49 - 50 - 50 - 50 - 51 - 53 - 55 - 55 - 57
- 57 - 58 - 58 - 58 - 59 - 60 - 61 - 63 - 63 - 64 - 65 - 68 - 70 - 72 - 73 - 74
- 76 - 77 - 78 - 79 - 79 - 79 - 80 - 80 - 80 - 81 - 81 - 81 - 82 - 82 - 83
- 84 - 84 - 86 - 87 - 88 - 88 - 89 - 89 - 90 - 90 - 91 - 91 - 92 - 93 - 93 - 94
- 95 - 99 -

0 número de comparações foi: 189
0 número de trocas foi: 832
Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.

```

Figura 38 - Execução do Shell Sort em vetor aleatório de 100 elementos, Gap = TAM/2.  
Fonte: Elaborado pela Autora

Executável do crescente (Gap = TAM/2)

```

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

0 número de comparações foi: 189
0 número de trocas foi: 0
Process returned 0 (0x0)   execution time : 0.100 s
Press any key to continue.

```

Figura 39 - Execução do Shell Sort em vetor crescente de 100 elementos, Gap = TAM/2.  
Fonte: Elaborado pela Autora

Executável do decrescente (Gap = TAM/3):

```

100 - 99 - 98 - 97 - 96 - 95 - 94 - 93 - 92 - 91 - 90 - 89 - 88 - 87 - 86 - 85 -
84 - 83 - 82 - 81 - 80 - 79 - 78 - 77 - 76 - 75 - 74 - 73 - 72 - 71 - 70 - 69 -
68 - 67 - 66 - 65 - 64 - 63 - 62 - 61 - 60 - 59 - 58 - 57 - 56 - 55 - 54 - 53 -
52 - 51 - 50 - 49 - 48 - 47 - 46 - 45 - 44 - 43 - 42 - 41 - 40 - 39 - 38 - 37 -
36 - 35 - 34 - 33 - 32 - 31 - 30 - 29 - 28 - 27 - 26 - 25 - 24 - 23 - 22 - 21 -
20 - 19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 4 -
3 - 2 - 1 -

1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 -
19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 -
35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 -
51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 -
67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 -
83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 -
99 - 100 -

0 número de comparações foi: 185
0 número de trocas foi: 778
Process returned 0 (0x0)   execution time : 0.084 s
Press any key to continue.

```

Figura 40 - Execução do Shell Sort em vetor decrescente de 100 elementos, Gap = TAM/3.  
Fonte: Elaborado pela Autora

Executáveis do aleatório (Gap=TAM/3):

```

42 - 92 - 72 - 11 - 86 - 8 - 36 - 45 - 33 - 25 - 31 - 79 - 97 - 19 - 85 - 27 - 1
9 - 23 - 32 - 71 - 78 - 48 - 68 - 60 - 24 - 43 - 94 - 58 - 25 - 84 - 89 - 64 - 5
3 - 22 - 91 - 90 - 97 - 49 - 43 - 52 - 24 - 45 - 90 - 80 - 10 - 54 - 47 - 54 - 3
7 - 30 - 14 - 95 - 100 - 83 - 9 - 8 - 56 - 90 - 34 - 93 - 9 - 34 - 43 - 18 - 29
- 45 - 23 - 8 - 99 - 63 - 28 - 87 - 74 - 41 - 17 - 6 - 3 - 85 - 58 - 44 - 53 - 1
8 - 78 - 38 - 67 - 39 - 16 - 73 - 11 - 27 - 25 - 81 - 17 - 52 - 53 - 18 - 50 - 5
2 - 52 - 82 -

3 - 6 - 8 - 8 - 8 - 9 - 9 - 10 - 11 - 11 - 14 - 16 - 17 - 17 - 18 - 18 - 18 - 19
- 19 - 22 - 23 - 23 - 24 - 24 - 25 - 25 - 25 - 27 - 27 - 28 - 29 - 30 - 31 - 32
- 33 - 34 - 34 - 36 - 37 - 38 - 39 - 41 - 42 - 43 - 43 - 44 - 45 - 45 - 45
- 47 - 48 - 49 - 50 - 52 - 52 - 52 - 52 - 53 - 53 - 53 - 54 - 54 - 56 - 58 - 58
- 60 - 63 - 64 - 67 - 68 - 71 - 72 - 73 - 74 - 78 - 78 - 79 - 80 - 81 - 82 - 83
- 84 - 85 - 85 - 86 - 87 - 89 - 90 - 90 - 90 - 91 - 92 - 93 - 94 - 95 - 97 - 97
- 99 - 100 -

0 número de comparações foi: 185
0 número de trocas foi: 931
Process returned 0 (0x0)   execution time : 0.088 s
Press any key to continue.

```

Figura 41 - Execução do Shell Sort em vetor aleatório de 100 elementos, Gap = TAM/3.  
Fonte: Elaborado pela Autora

```

96 - 79 - 88 - 74 - 51 - 24 - 81 - 35 - 45 - 78 - 80 - 49 - 48 - 64 - 40 - 17 -
81 - 72 - 72 - 19 - 51 - 56 - 69 - 66 - 36 - 18 - 88 - 83 - 42 - 10 - 94 - 51 -
52 - 4 - 36 - 78 - 13 - 74 - 80 - 38 - 82 - 50 - 10 - 59 - 59 - 72 - 63 - 96 - 6
1 - 78 - 4 - 82 - 43 - 69 - 21 - 12 - 46 - 72 - 87 - 70 - 11 - 63 - 30 - 18 - 31
- 33 - 71 - 91 - 49 - 36 - 31 - 72 - 15 - 1 - 40 - 49 - 24 - 83 - 8 - 44 - 14 -
9 - 11 - 15 - 68 - 48 - 48 - 47 - 82 - 85 - 58 - 16 - 3 - 39 - 41 - 88 - 87 - 3
9 - 32 - 54 -

1 - 3 - 4 - 4 - 8 - 9 - 10 - 10 - 11 - 11 - 12 - 13 - 14 - 15 - 15 - 16 - 17 - 1
8 - 18 - 19 - 21 - 24 - 24 - 30 - 31 - 31 - 32 - 33 - 35 - 36 - 36 - 36 - 38 - 3
9 - 39 - 40 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 48 - 48 - 49 - 49 - 4
9 - 50 - 51 - 51 - 51 - 52 - 54 - 56 - 58 - 59 - 59 - 61 - 63 - 63 - 64 - 66 - 6
8 - 69 - 69 - 70 - 71 - 72 - 72 - 72 - 72 - 72 - 74 - 74 - 78 - 78 - 78 - 79 - 8
0 - 80 - 81 - 81 - 82 - 82 - 82 - 83 - 83 - 85 - 87 - 87 - 88 - 88 - 88 - 91 - 9
4 - 96 - 96 -

0 número de comparações foi: 185
0 número de trocas foi: 1005
Process returned 0 (0x0)   execution time : 0.075 s
Press any key to continue.

```

Figura 42 - Execução do Shell Sort em vetor aleatório de 100 elementos, Gap = TAM/3.  
Fonte: Elaborado pela Autora



```

71 - 16 - 80 - 75 - 10 - 35 - 32 - 43 - 96 - 44 - 78 - 94 - 22 - 13 - 72 - 77 -
2 - 69 - 10 - 98 - 64 - 70 - 62 - 62 - 25 - 25 - 64 - 76 - 4 - 46 - 79 - 76 - 85
- 19 - 12 - 85 - 60 - 61 - 94 - 52 - 7 - 86 - 2 - 26 - 95 - 57 - 7 - 52 - 52 -
69 - 40 - 72 - 7 - 3 - 52 - 74 - 28 - 46 - 14 - 68 - 12 - 77 - 59 - 64 - 80 - 60
- 93 - 56 - 59 - 19 - 42 - 34 - 32 - 68 - 12 - 100 - 72 - 37 - 22 - 38 - 67 - 6
9 - 62 - 62 - 60 - 59 - 48 - 81 - 14 - 11 - 36 - 27 - 43 - 58 - 39 - 51 - 8 - 8
- 8 - 100 -

2 - 2 - 3 - 4 - 7 - 7 - 7 - 8 - 8 - 8 - 10 - 10 - 11 - 12 - 12 - 12 - 13 - 14 -
14 - 16 - 19 - 19 - 22 - 22 - 25 - 25 - 26 - 27 - 28 - 32 - 32 - 34 - 35 - 36 -
37 - 38 - 39 - 40 - 42 - 43 - 43 - 44 - 46 - 46 - 48 - 51 - 52 - 52 - 52 -
56 - 57 - 58 - 59 - 59 - 59 - 60 - 60 - 60 - 61 - 62 - 62 - 62 - 62 - 64 - 64 -
64 - 67 - 68 - 68 - 69 - 69 - 69 - 70 - 71 - 72 - 72 - 72 - 74 - 75 - 76 - 76 -
77 - 77 - 78 - 79 - 80 - 80 - 81 - 85 - 85 - 86 - 93 - 94 - 94 - 95 - 96 - 98 -
100 - 100 -

0 número de comparações foi: 185
0 número de trocas foi: 1047

Process returned 0 (0x0)   execution time : 0.083 s
Press any key to continue.

```

Figura 43 - Execução do Shell Sort em vetor aleatório de 100 elementos,  $Gap = TAM/3$ .  
Fonte: Elaborado pela Autora

Executável do crescente ( $Gap = TAM/3$ ):

```

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

0 número de comparações foi: 185
0 número de trocas foi: 0

Process returned 0 (0x0)   execution time : 0.086 s
Press any key to continue.

```

Figura 44 - Execução do Shell Sort em vetor crescente de 100 elementos,  $Gap = TAM/3$ .  
Fonte: Elaborado pela Autora

#### 4.4 – Implementação utilizando lista sequencial

```
#include <iostream>
```

```
#include <time.h>
```

```
#include <conio.h>
```

```
#include <stdlib.h>
```

```
// Aleatório
```

```
using namespace std;
```

```
int const TAM = 100;

int vetor[TAM];

int comparacao;

int troca;


void valores_aleatorios (int vetor_[TAM])
{
    for(int a=0; a<TAM; a++)
    {
        vetor_[a] = rand() % TAM + 1;

    }
}


void decrescente ( int vetor_[TAM] )
{
    for ( int d = 0; d < TAM; d++ )
    {

        vetor_[d] = TAM - d;

    }

}
```

```
void crescente(int vetor_[TAM])
{

    for (int c = 0; c < TAM; c++)
    {

        vetor_[c] = c;

    }

}
```

```
void exhibe(int vetor_[TAM])
{

    for (int e=0; e<TAM; e++)
    {

        cout << vetor_[e] << " - ";

    }

    cout << endl;

}
```

```
void shellsort(int vetor_[TAM]){
```

```

    int gap, c;

    //pode chamar de h ou gap, por isso a conta do erthal  $h = h * 3 + 1$ ;
    // c == controle


    int valor;

    //guarda o valor para fazer a troca


    // GAPS UTILIZADOS:

    // comente para trocar de gap


    //gap = 1;

    //gap = TAM/2;

    gap = TAM/3;

while(gap < TAM)
{

    gap = gap * 2 + 3; //gap inicial


    //TESTES ALEATÓRIOS PARA EU VER SE ORDENA:

    //gap = gap * 3 + 1;

    // Padrão

    //gap = gap * 2 + 1;

```

```

        // também ordena
    }    //gap = gap * 2 + 3;

        //também ordena

while(gap > 1){

    gap = gap/10;

    for(int i = gap; i < TAM; i++){

        valor = vetor_[i];

        c = i - gap;

        //comparação

        comparacao = comparacao + 1;

        while((c>=0) && (valor <=vetor_[c] ))

        {// verificação

            vetor_[c+gap] = vetor_[c];

            // troca

            c = c - gap;

            troca = troca + 1;

        }

        vetor_[c+gap] = valor;

        // troca

```

```
    }  
}  
  
}
```

```
int main()  
{  
  
    setlocale(LC_ALL, "portuguese");  
    srand (time(NULL));  
  
  
    crescente(vetor);  
    exhibe(vetor);  
    cout << endl;  
    shellsort(vetor);  
    exhibe(vetor);  
    cout << endl;  
  
    cout << endl << "O número de comparações foi: " << comparacao << endl;  
    cout << endl << "O número de trocas foi: " << troca << endl;
```

```
return 0;
```

```
}
```

## 5 – HEAP (Lista de Prioridade)

### 5.1 - Resumo do Método

Lista de prioridade, também denominada de Heap, consiste em uma estrutura, na qual o interesse é pegar o que tem maior ou menor prioridade. O caso em questão consiste em um maxheap, ou heap máximo, onde o maior valor é localizado na raiz, não podendo ter filhos maiores. Heap é uma estrutura que apesar de não ordenada, é capaz de com baixa complexidade, ou seja, com muita eficiência, entregar o elemento de maior prioridade, onde inserindo e removendo ele se reajusta.

Funções que compõem o heap:

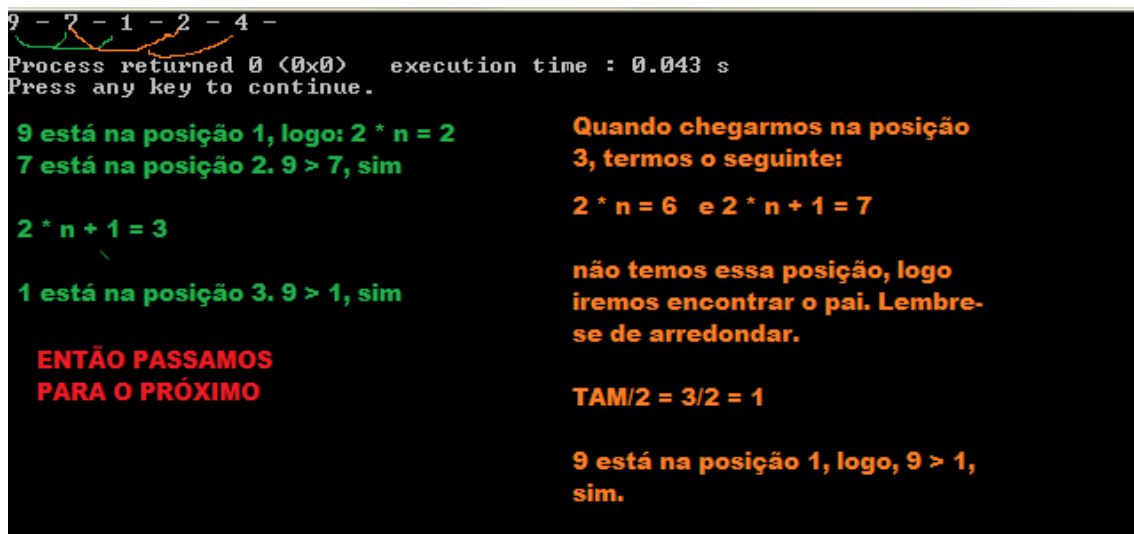
**Inserir** - Colocando o elemento na última posição e chamando a função subir, que verificada as posições para que o heap se mantenha.

**Remover** - Trocando a primeira com a última posição, para aí sim remover.

**Alterar prioridade** - Nela é necessário as funções de sobe e desce, pois ao darmos prioridade a um elemento, ou seja, subir ele, devemos reorganizar o heap, de modo que ele continue respeitando as prioridades.

O heap é identificado partir da seguinte expressão  $2 * n$ ,  $2 * n + 1$  e  $TAM/2$ .

Sendo assim, temos o seguinte exemplo:



```
9 - 7 - 1 - 2 - 4 -
Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.

9 está na posição 1, logo:  $2 * n = 2$ 
7 está na posição 2.  $9 > 7$ , sim

 $2 * n + 1 = 3$ 
1 está na posição 3.  $9 > 1$ , sim

ENTÃO PASSAMOS
PARA O PRÓXIMO

Quando chegarmos na posição
3, temos o seguinte:

 $2 * n = 6$  e  $2 * n + 1 = 7$ 

não temos essa posição, logo
iremos encontrar o pai. Lembre-
se de arredondar.

 $TAM/2 = 3/2 = 1$ 

9 está na posição 1, logo,  $9 > 1$ ,
sim.
```

Figura 45 - Execução do Heap com análise  
Fonte: Elaborado pela Autora

Obs: Isso para prioridade de maior valor, para menor verifica-se se é menor.



## 5.2 - Complexidade

Visto “n”, como quantidade de elementos de um heap, temos que, tanto a inserção quanto a remoção terá complexidade  $O(\log(n))$ .

## 5.3 - Implementação utilizando lista de prioridade

```
#include <iostream>
#include <stdlib.h>
// Srand, rand
#include <time.h>
// Tempo
#include <conio.h>

using namespace std;

const int TAM = 10;
int vetor[TAM];
int p = 1;
int tam;

void subir(int vetor[TAM], int tam)//controle de tamanho
{
    int auxiliar;
    // Auxilia na troca
    int pai=tam/2;
```

```

    if(pai>=1)
    {

        if(vetor[tam] > vetor[pai])
        {
            // se tiver maior ptioridade faz a troca

            auxiliar=vetor[tam];
            vetor[tam]=vetor[pai];
            vetor[pai]=auxiliar;
            subir(vetor,pai);
            // Atualiza o tam

        }

    }
}

void descer(int vetor[TAM], int tam, int posicao)
{
    int auxiliar;
    //troca
    int esquerda = 2 * posicao;
    // filho esquerdo
    int direita = 2 * posicao + 1;
    // filho direito
    int maior = posicao;

```

```

        if((esquerda <= tam) && (vetor[esquerda] >
vetor[posicao] ))
        {

            maior = esquerda;

        }
        if((direita <= tam) && (vetor[direita] >
vetor[maior]))
        {

            maior = direita;

        }
        if(maior != posicao)
        {
            auxiliar = vetor[posicao];
            vetor[posicao] = vetor[maior];
            vetor[maior] = auxiliar;
            descer(vetor, tam, maior);
        }
    }
}

```

```

void inserir(int *vetor,int elemento){

    tam = tam + 1;
    vetor[tam]=elemento;
    subir(vetor,tam);

}

```

```
void excluir(int *vetor){

    vetor[p]= vetor[tam];
    tam = tam - 1;
    descer(vetor, tam, p);

}

void exibir(int vetor[TAM])
{
    for(int e=1;e<=tam;e++){
        cout << vetor[e] << " - ";

    }

    cout << endl;

}

int main(){

    inserir(vetor,15);
    inserir(vetor,6);
    inserir(vetor,8);
    inserir(vetor,2);
    inserir(vetor,3);
    exibir(vetor);
    excluir(vetor);
```

```
    exibir(vetor);  
    excluir(vetor);  
    exibir(vetor);  
    return 0;  
}
```

## **6– HeapSort (Ordenação de Lista de Prioridade)**

## 6.1 - Resumo do método

Heapsort, consiste em um método de ordenação que deriva do heap, de modo a ordenar os elementos à medida que insere. Possui um tempo de execução muito bom em relação a conjuntos ordenados aleatoriamente, além de um nível comportado de uso de memória. Com elementos inseridos por vetores, nesse caso, crescente, decrescente e aleatório.

6 5 3 1 8 7 2 4

## 6.2 - Complexidade

O HeapSort possui complexidade  $O(n \log n)$  para todos os casos.

## 6.3 - Resultados

Tabela 1 - Lista com 10 elementos

Construção da Lista	Comparação	Troca
Crescente	19	5
Decrescente	9	0
Aleatório	Em média 13 comparações	Em média 4,6 trocas

Tabela 2 - Lista com 100 elementos

Construção da Lista	Comparação	Troca
Crescente	480	16
Decrescente	99	0
Aleatório	Em média 203,6 comparações	Em média 15,6 trocas

Executáveis do Aleatório:

```
9 - 9 - 6 - 7 - 9 - 6 - 5 - 1 - 5 - 3 -
0 número de trocas é: 3
0 número de comparações é: 14
Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.
```

*Figura 46 - Execução do HeapSort em vetor aleatório de 10 elementos  
Fonte: Elaborado pela Autora*

```
9 - 8 - 7 - 8 - 7 - 2 - 2 - 4 - 7 - 6 -
0 número de trocas é: 6
0 número de comparações é: 14
Process returned 0 (0x0)   execution time : 0.046 s
Press any key to continue.
```

*Figura 47 - Execução do HeapSort em vetor aleatório de 10 elementos  
Fonte: Elaborado pela Autora*

```

10 - 8 - 8 - 4 - 7 - 7 - 2 - 1 - 3 - 5 -
0 número de trocas é: 5
0 número de comparações é: 11
Process returned 0 (0x0)   execution time : 0.039 s
Press any key to continue.

```

*Figura 48 - Execução do HeapSort em vetor aleatório de 10 elementos*  
*Fonte: Elaborado pela Autora*

```

100 - 100 - 95 - 98 - 97 - 91 - 93 - 97 - 94 - 91 - 95 - 77 - 73 - 88 - 84 - 88
- 91 - 93 - 75 - 69 - 89 - 74 - 68 - 68 - 69 - 72 - 62 - 68 - 85 - 55 - 60 - 58
- 82 - 79 - 47 - 48 - 84 - 57 - 62 - 40 - 67 - 28 - 63 - 55 - 70 - 66 - 55 - 48
- 49 - 25 - 65 - 40 - 39 - 52 - 6 - 13 - 32 - 53 - 47 - 7 - 30 - 56 - 12 - 6 - 5
- 44 - 76 - 36 - 74 - 2 - 36 - 30 - 35 - 4 - 63 - 14 - 50 - 42 - 3 - 24 - 11 -
44 - 65 - 13 - 6 - 8 - 56 - 29 - 52 - 57 - 63 - 27 - 42 - 48 - 1 - 2 - 40 - 46 -
20 - 9 -
0 número de trocas é: 9
0 número de comparações é: 207
Process returned 0 (0x0)   execution time : 0.069 s
Press any key to continue.

```

*Figura 49 - Execução do HeapSort em vetor aleatório de 100 elementos*  
*Fonte: Elaborado pela Autora*



```

100 - 100 - 97 - 98 - 99 - 94 - 90 - 96 - 98 - 96 - 97 - 90 - 88 - 82 - 77 - 66
- 78 - 97 - 90 - 90 - 90 - 80 - 84 - 86 - 55 - 80 - 82 - 29 - 59 - 68 - 63 - 38
- 46 - 54 - 59 - 85 - 87 - 82 - 66 - 72 - 68 - 82 - 74 - 51 - 70 - 58 - 62 - 80
- 77 - 33 - 41 - 40 - 8 - 11 - 78 - 13 - 8 - 42 - 7 - 14 - 26 - 6 - 60 - 18 - 30
- 1 - 37 - 10 - 43 - 16 - 58 - 14 - 61 - 36 - 64 - 57 - 40 - 60 - 65 - 4 - 52 -
51 - 8 - 63 - 71 - 71 - 52 - 7 - 27 - 19 - 38 - 41 - 52 - 24 - 56 - 17 - 59 - 9
- 62 - 15 -

0 número de trocas é: 15
0 número de comparações é: 199
Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.

```

*Figura 50 - Execução do HeapSort em vetor aleatório de 100 elementos  
Fonte: Elaborado pela Autora*

```

100 - 100 - 98 - 93 - 98 - 90 - 96 - 90 - 91 - 97 - 96 - 88 - 87 - 94 - 77 - 82
- 86 - 64 - 77 - 93 - 83 - 64 - 88 - 68 - 53 - 28 - 45 - 62 - 83 - 53 - 71 - 60
- 75 - 71 - 84 - 19 - 39 - 37 - 49 - 59 - 82 - 80 - 67 - 38 - 30 - 30 - 87 - 59
- 57 - 25 - 34 - 19 - 24 - 25 - 33 - 39 - 44 - 22 - 48 - 8 - 51 - 30 - 62 - 7 -
41 - 23 - 39 - 9 - 18 - 67 - 15 - 13 - 16 - 12 - 13 - 29 - 25 - 16 - 14 - 56 - 9
- 60 - 77 - 2 - 10 - 30 - 2 - 18 - 2 - 20 - 20 - 1 - 13 - 7 - 26 - 3 - 4 - 53 -
4 - 23 -

0 número de trocas é: 23
0 número de comparações é: 205
Process returned 0 (0x0)   execution time : 0.057 s
Press any key to continue.

```

*Figura 51 - Execução do HeapSort em vetor aleatório de 100 elementos  
Fonte: Elaborado pela Autora*

## 6.4 - Implementação utilizando lista sequencial

```
#include <iostream>
#include <stdlib.h> //
Srand, rand
#include <time.h> //
Tempo
#include <conio.h>
```

```
using namespace std;
```

```
const int TAM = 10;
// muda o tamanho pela constante
int vetor[TAM];
int troca = 0;
int comparacao = 0;
```

```
void subir(int vetor[TAM], int tam)
```

```
// controle de tamanho
```

```
{
```

```
    int auxiliar;
```

```
    // Auxilia na troca
```

```
    int pai=tam/2;
```

```
    if(pai>=1)
```

```
    {
```

```
        if(vetor[tam] > vetor[pai])
```

```
        {
```

```
            // se tiver maior ptioridade faz a troca
```

```
            troca = troca + 1;
```

```
            auxiliar=vetor[tam];
```

```
            vetor[tam]=vetor[pai];
```

```
            vetor[pai]=auxiliar;
```

```
            subir(vetor,pai);
```

```
            // Atualiza o tam
```

```

    }

    comparacao = comparacao + 1;
    // Contagem da comparação
}
}

void descer(int vetor[TAM], int tam, int posicao)
{
    int auxiliar;
    //troca
    int esquerda = 2 * posicao;
    // filho esquerdo
    int direita = 2 * posicao + 1;
    // filho direito
    int maior = posicao;

    if((esquerda <= tam) && (vetor[esquerda] > vetor[posicao]
))
    {

        maior = esquerda;

    }
    if((direita <= tam) && (vetor[direita] > vetor[maior]))
    {

        maior = direita;

    }
    if(maior != posicao)
    {
        auxiliar = vetor[posicao];
        vetor[posicao] = vetor[maior];
        vetor[maior] = auxiliar;
        descer(vetor, tam, maior);
    }
}

```

```

    }
}

void aleatorios(int vetor[TAM])
{
    for(int a=1; a<=TAM; a++)
    {

        vetor[a] = rand() % TAM + 1;
        subir(vetor,a);

    }

}

void exhibe (int vetor[TAM])
{

    for(int c=1;c<=TAM;c++){
        cout << vetor[c] << " - ";
    }

    cout << endl;
}

void crescente(int vetor[TAM])
{
    for(int c=1; c<=TAM; c++)
    {

        vetor[c]=c;
        subir(vetor,c);

    }
}

void decrescente(int vetor[TAM])
{
    for(int d=1; d<=TAM; d++)

```

```

    {

vetor[d]= TAM - d;
subir(vetor,d);

    }
}

int main()
{

    setlocale(LC_ALL, "portuguese");
    srand(time(NULL));

    //Para alterar o vetor só mudar a chamada da função!

    aleatorios(vetor);
    exhibe(vetor);

    cout << endl << " O número de trocas é: " << troca;
    cout << endl << " O número de comparações é: " <<
comparacao;
    cout << endl;

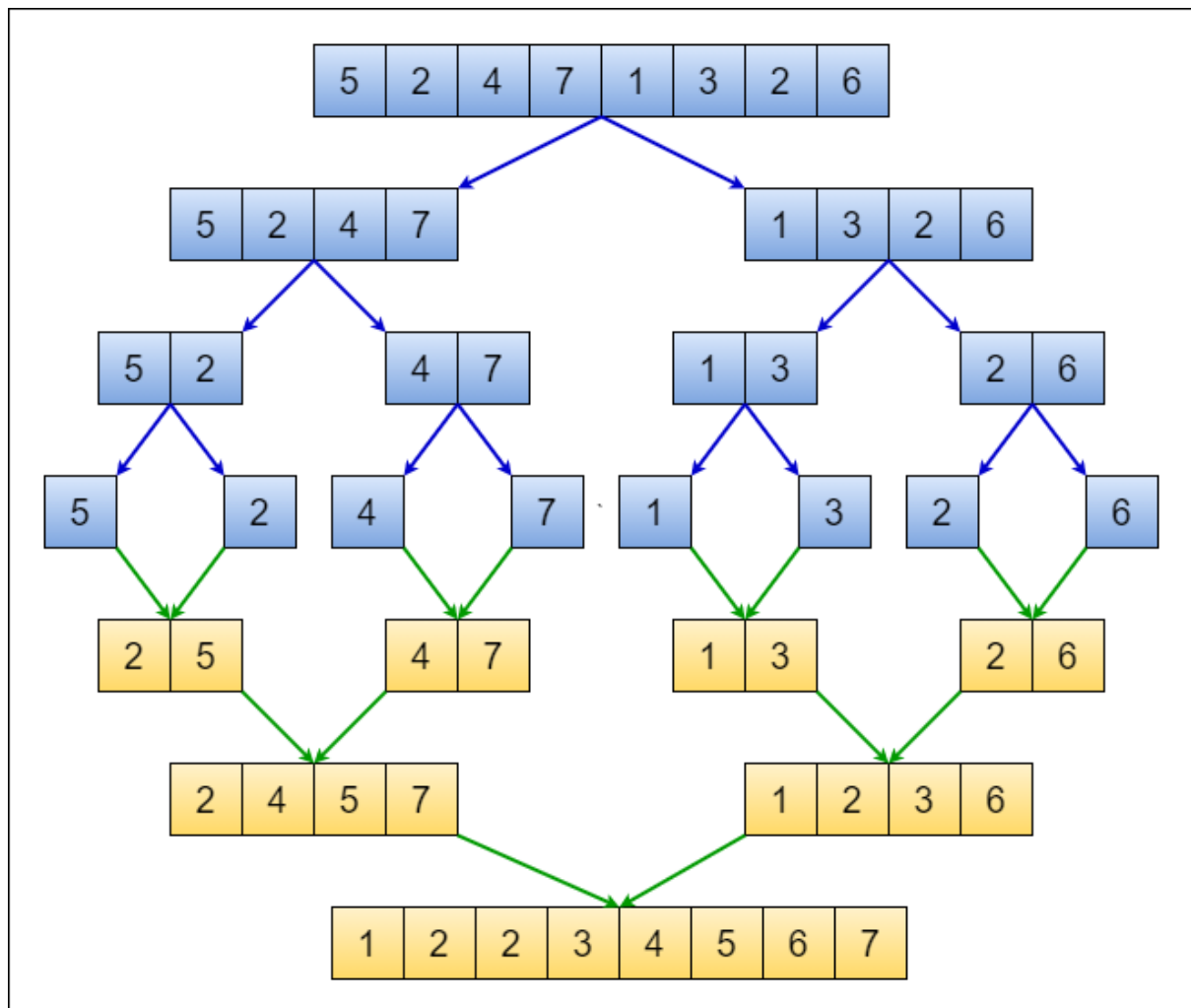
    return 0;
}

```

## 7 - MergeSort

### 7.1 - Resumo do Método

MergeSort ou ordenação por mistura, é um algoritmo de ordenação, que consiste em dividir uma lista em sublistas. Adiante, verifica-se se um é menor que outro, se sim, troca, começando pela lista da direita, ou seja, do início ao meio e depois a da esquerda, sendo do meio ao fim. Por fim, com as sublistas ordenadas, comparamos o topo de uma sublista com a outra até que todos sejam verificados, após isso as sublistas se agrupam novamente. Ademais, esse método utiliza-se da recursividade.



## 7.2 - Complexidade

Para apontar a complexidade é interessante considerar três passos importantes:

1. Divisão: Calcula o meio;
2. Recursividade: resolvendo as sublistas;
3. Agrupar: Reagrupando os elementos;

Com isso, temos o seguinte:

$$\sum_{i=1}^{\log_2 n} n - 2^{i-1} \xrightarrow[\text{Fator de subtração de } n]{\text{Desconsidera}} \log_2 n \cdot n \Rightarrow O(n \log n)$$

Fonte: Elaborado pela Autora

Nessa perspectiva, teremos complexidade  $O(n \log n)$  para todos os casos.

## 7.3 - Resultados

Tabela 1 - Lista com 10 elementos

Construção da Lista	Comparações	Trocas
Crescente	34	9
Decrescente	34	9
Aleatório	34	9

Tabela 2 - Lista com 100 elementos

Construção da Lista	Comparação	Troca
Crescente	672	99
Decrescente	672	99
Aleatório	672	99

## Executáveis do aleatório

```
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -  
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -  
  
Foram realizadas 9 trocas  
Foram realizadas 34 comparações  
Process returned 0 (0x0)   execution time : 0.041 s  
Press any key to continue.  
-
```

*Figura 61 - Execução do MergeSort em vetor crescente de 10 elementos*

*Fonte: Elaborado pela Autora*

```
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -  
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -  
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -  
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -  
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -  
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -  
98 - 99 -  
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -  
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -  
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -  
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -  
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -  
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -  
98 - 99 -  
  
Foram realizadas 99 trocas  
Foram realizadas 672 comparações  
Process returned 0 (0x0)   execution time : 0.067 s  
Press any key to continue.  
-
```

*Figura 62 - Execução do MergeSort em vetor crescente de 100 elementos*

*Fonte: Elaborado pela Autora*



Executáveis do decrescente:

```
10 - 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 -  
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -  
  
Foram realizadas 9 trocas  
Foram realizadas 34 comparações  
Process returned 0 (0x0)   execution time : 0.066 s  
Press any key to continue.
```

*Figura 63 - Execução do MergeSort em vetor decrescente de 10 elementos*  
*Fonte: Elaborado pela Autora*

```
100 - 99 - 98 - 97 - 96 - 95 - 94 - 93 - 92 - 91 - 90 - 89 - 88 - 87 - 86 - 85 -  
84 - 83 - 82 - 81 - 80 - 79 - 78 - 77 - 76 - 75 - 74 - 73 - 72 - 71 - 70 - 69 -  
68 - 67 - 66 - 65 - 64 - 63 - 62 - 61 - 60 - 59 - 58 - 57 - 56 - 55 - 54 - 53 -  
52 - 51 - 50 - 49 - 48 - 47 - 46 - 45 - 44 - 43 - 42 - 41 - 40 - 39 - 38 - 37 -  
36 - 35 - 34 - 33 - 32 - 31 - 30 - 29 - 28 - 27 - 26 - 25 - 24 - 23 - 22 - 21 -  
20 - 19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 4 -  
3 - 2 - 1 -  
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 -  
19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 -  
35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 -  
51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 -  
67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 -  
83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 -  
99 - 100 -  
  
Foram realizadas 99 trocas  
Foram realizadas 672 comparações  
Process returned 0 (0x0)   execution time : 0.080 s  
Press any key to continue.
```

*Figura 64 - Execução do MergeSort em vetor decrescente de 100 elementos*  
*Fonte: Elaborado pela Autora*

Executáveis do aleatório:

```
2 - 8 - 5 - 1 - 10 - 5 - 9 - 9 - 3 - 5 -  
1 - 2 - 3 - 5 - 5 - 5 - 8 - 9 - 9 - 10 -  
  
Foram realizadas 9 trocas  
Foram realizadas 34 comparações  
Process returned 0 (0x0)   execution time : 0.041 s  
Press any key to continue.  
-
```

*Figura 65 - Execução do MergeSort em vetor decrescente de 10 elementos  
Fonte: Elaborado pela Autora*

```
42 - 68 - 35 - 1 - 70 - 25 - 79 - 59 - 63 - 65 - 6 - 46 - 82 - 28 - 62 - 92 - 96  
- 43 - 28 - 37 - 92 - 5 - 3 - 54 - 93 - 83 - 22 - 17 - 19 - 96 - 48 - 27 - 72 -  
39 - 70 - 13 - 68 - 100 - 36 - 95 - 4 - 12 - 23 - 34 - 74 - 65 - 42 - 12 - 54 -  
69 - 48 - 45 - 63 - 58 - 38 - 60 - 24 - 42 - 30 - 79 - 17 - 36 - 91 - 43 - 89 -  
7 - 41 - 43 - 65 - 49 - 47 - 6 - 91 - 30 - 71 - 51 - 7 - 2 - 94 - 49 - 30 - 24  
- 85 - 55 - 57 - 41 - 67 - 77 - 32 - 9 - 45 - 40 - 27 - 24 - 38 - 39 - 19 - 83 -  
30 - 42 -  
1 - 2 - 3 - 4 - 5 - 6 - 6 - 7 - 7 - 9 - 12 - 12 - 13 - 17 - 17 - 19 - 19 - 22 -  
23 - 24 - 24 - 24 - 25 - 27 - 27 - 28 - 28 - 30 - 30 - 30 - 30 - 32 - 34 - 35 -  
36 - 36 - 37 - 38 - 38 - 39 - 39 - 40 - 41 - 41 - 42 - 42 - 42 - 42 - 43 - 43 -  
43 - 45 - 45 - 46 - 47 - 48 - 48 - 49 - 49 - 51 - 54 - 54 - 55 - 57 - 58 - 59 -  
60 - 62 - 63 - 63 - 65 - 65 - 65 - 67 - 68 - 68 - 69 - 70 - 70 - 71 - 72 - 74 -  
77 - 79 - 79 - 82 - 83 - 83 - 85 - 89 - 91 - 91 - 92 - 92 - 93 - 94 - 95 - 96 -  
96 - 100 -  
  
Foram realizadas 99 trocas  
Foram realizadas 672 comparações  
Process returned 0 (0x0)   execution time : 0.068 s  
Press any key to continue.  
-
```

*Figura 66 - Execução do MergeSort em vetor decrescente de 100 elementos  
Fonte: Elaborado pela Autora*

## 7.4 - Implementação

```
#include <iostream>
#include <time.h>
#include <stdlib.h>

using namespace std;

int const TAM = 10;
int vetor[TAM];
int troca = 0;
int comparacao = 0;

void aleatorio (int vetor_[TAM])
{
    for(int a=0; a<TAM; a++){
        vetor_[a] = rand() % TAM + 1;
    }
}

void crescente(int vetor_[TAM])
{
    for(int i = 0; i < TAM; i++)
    {
        vetor_[i]= i;
    }
}

void decrescente(int vetor_[TAM])
{

```

```

        for(int d=0; d<=TAM; d++)
        {
            vetor[d]= TAM - d;
        }
    }
    int agrupa( int *vetor_,int inicio, int meio, int
fim)
    {
        int auxiliar[TAM];
        // auxiliar é uma lista vazia onde serão colocados
os elementos
        int esquerda = inicio;
        //começo da primeira lista
        int direita = meio;
        // começo da segunda lista

for( int a = inicio; a < fim; a++)
{

    if((esquerda < meio) && ((direita >= fim) ||
(vetor_[esquerda] < vetor_[direita]))){

        auxiliar[a] = vetor_[esquerda];
        esquerda++;

    }
    else{

        auxiliar[a] = vetor_[direita];
        direita++;

    }
}

```

```

        comparacao = comparacao + 1;
    }
    for(int c = inicio; c < fim; c++){

        vetor_[c] = auxiliar[c];
        // recebe o auxiliar pois ele que está recebendo
os valores ordenados, então eu passo para o vetor
principal

    }

}
void mergesort(int *vetor_, int inicio, int fim)

{

    int meio;
    meio = ((inicio + fim) / 2);
    // para pegar o meio

    if(inicio < meio)
        // se for menor a lista da esquerda vai ter
terminado
    {
        mergesort( vetor, inicio, meio);
        // lista da direita
        mergesort( vetor, meio, fim);
        // lista da esquerda
        troca = troca + 1;
        agrupa(vetor_, inicio, meio, fim);
        // agrupa os valores novamente
    }
}

```

```

    }

}

void exhibe (int vetor_[TAM])
{

    for(int e = 0; e < TAM; e++)
    {
        cout << vetor_[e] << " - ";

    }
    cout << endl;
}

int main()
{

    aleatorio(vetor);
    exhibe(vetor);
    mergesort(vetor, 0, TAM);
    exhibe(vetor);

    cout << endl << " Foram realizadas " << troca <<
" trocas " << endl;
    cout << endl << " Foram realizadas " <<
comparacao << " comparações " << endl;

    return 0;
}

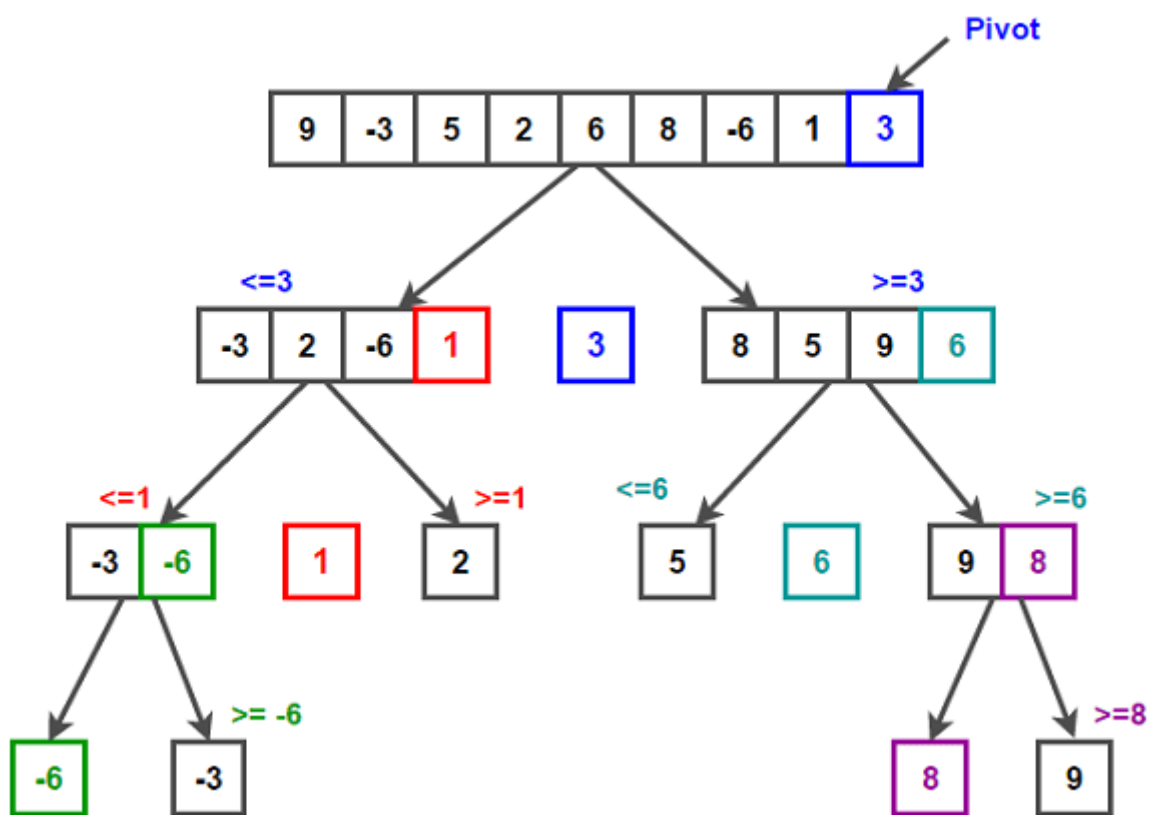
```

## 6.1 - Resumo do método

## 8 - QuickSort

### 8.1 - Resumo do método

QuickSort é um algoritmo de ordenação bastante eficiente. Consiste basicamente em usar um pivot e dividir a lista em dois, onde os maiores ficaram na direita e os menores na esquerda, colocando o pivot no lugar certo de maneira sucessiva. Dessa forma, os pivôs vão para o lugar certo organizando a lista.



### 8.2 - Complexidade

Para o cálculo da complexidade iremos considerar o pior caso, que ocorre quando o pivot divide a lista de forma desbalanceada, ou seja, uma com tamanho 0 em uma lista e  $n - 1$  em outra, sendo  $n$  o tamanho original. Isso pode acontecer quando o pivot é o maior ou menor elemento da lista. Nessa perspectiva, seria uma lista ordenada ou inversamente ordenada.

Se isso ocorre teremos chamadas cujo tamanho é igual a da lista anterior. Desse modo teremos:

$$T(n) = T(n - 1) + T(0) + O(n) \\ T(n - 1) + O(n)$$

Logo, na série aritmética teremos uma complexidade de pior caso de  $O(n^2)$

Já o melhor caso, ocorre quando não tem tamanho maior que  $n/2$ , já que cada lista será executada em  $n/2$  dando maior rapidez a execução

Nesse caso teremos o seguinte:

$$T(n) \leq 2T(n/2) + O(n)$$

Obtendo a complexidade de  $O(n \log n)$

Por fim, temos  $O(n^2)$  para o pior caso,  $O(n \log n)$  para o melhor caso e  $O(n \log n)$  para o caso médio.

## 8.2 - Resultados

Tabela 1 - Lista com 10 elementos

Construção da lista	Comparação	Troca
Crescente	45	9
Decrescente	45	9
Aleatório	22,3 em média	6,3 em média



Tabela 2 - Lista com 100 elementos

Construção da Lista	Comparação	Troca
Crescente	4950	99
Decrescente	4950	99
Aleatório	736,6 em média	67 em média

#### Executáveis do Crescente

```
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -

foram realizadas 9 trocas
foram realizadas 45 comparações
Process returned 0 (0x0)   execution time : 0.054 s
Press any key to continue.
```

*Figura 67 - Execução do QuickSort em vetor decrescente de 10 elementos*  
*Fonte: Elaborado pela Autora*

```

0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 -
18 - 19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 -
34 - 35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 -
50 - 51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 -
66 - 67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 -
82 - 83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 -
98 - 99 -

foram realizadas 99 trocas
foram realizadas 4950 comparações
Process returned 0 (0x0)   execution time : 0.079 s
Press any key to continue.

```

*Figura 68 - Execução do QuickSort em vetor decrescente de 100 elementos*  
*Fonte: Elaborado pela Autora*

#### Executáveis do decrescente

```

10 - 9 - 8 - 7 - 6 - 5 - 4 - 3 - 2 - 1 -
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 -

foram realizadas 9 trocas
foram realizadas 45 comparações
Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.

```

*Figura 69 - Execução do QuickSort em vetor decrescente de 10 elementos*  
*Fonte: Elaborado pela Autora*

```

100 - 99 - 98 - 97 - 96 - 95 - 94 - 93 - 92 - 91 - 90 - 89 - 88 - 87 - 86 - 85 -
84 - 83 - 82 - 81 - 80 - 79 - 78 - 77 - 76 - 75 - 74 - 73 - 72 - 71 - 70 - 69 -
68 - 67 - 66 - 65 - 64 - 63 - 62 - 61 - 60 - 59 - 58 - 57 - 56 - 55 - 54 - 53 -
52 - 51 - 50 - 49 - 48 - 47 - 46 - 45 - 44 - 43 - 42 - 41 - 40 - 39 - 38 - 37 -
36 - 35 - 34 - 33 - 32 - 31 - 30 - 29 - 28 - 27 - 26 - 25 - 24 - 23 - 22 - 21 -
20 - 19 - 18 - 17 - 16 - 15 - 14 - 13 - 12 - 11 - 10 - 9 - 8 - 7 - 6 - 5 - 4 -
3 - 2 - 1 -
1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - 11 - 12 - 13 - 14 - 15 - 16 - 17 - 18 -
19 - 20 - 21 - 22 - 23 - 24 - 25 - 26 - 27 - 28 - 29 - 30 - 31 - 32 - 33 - 34 -
35 - 36 - 37 - 38 - 39 - 40 - 41 - 42 - 43 - 44 - 45 - 46 - 47 - 48 - 49 - 50 -
51 - 52 - 53 - 54 - 55 - 56 - 57 - 58 - 59 - 60 - 61 - 62 - 63 - 64 - 65 - 66 -
67 - 68 - 69 - 70 - 71 - 72 - 73 - 74 - 75 - 76 - 77 - 78 - 79 - 80 - 81 - 82 -
83 - 84 - 85 - 86 - 87 - 88 - 89 - 90 - 91 - 92 - 93 - 94 - 95 - 96 - 97 - 98 -
99 - 100 -

foram realizadas 99 trocas
foram realizadas 4950 comparações
Process returned 0 (0x0)   execution time : 0.070 s
Press any key to continue.

```

*Figura 69 - Execução do QuickSort em vetor decrescente de 100 elementos  
Fonte: Elaborado pela Autora*

Executáveis do aleatório:

```

6 - 9 - 9 - 5 - 8 - 6 - 2 - 10 - 10 - 1 -
1 - 2 - 5 - 6 - 6 - 8 - 9 - 9 - 10 - 10 -

foram realizadas 7 trocas
foram realizadas 27 comparações
Process returned 0 (0x0)   execution time : 0.040 s
Press any key to continue.
-

```

*Figura 70 - Execução do QuickSort em vetor aleatório de 10 elementos  
Fonte: Elaborado pela Autora*

```
2 - 2 - 10 - 5 - 5 - 9 - 10 - 3 - 7 - 6 -  
2 - 2 - 3 - 5 - 5 - 6 - 7 - 9 - 10 - 10 -  
  
foram realizadas 6 trocas  
foram realizadas 19 comparações  
Process returned 0 (0x0)   execution time : 0.051 s  
Press any key to continue.  
-
```

*Figura 70.1 - Execução do QuickSort em vetor aleatório de 10 elementos*  
*Fonte: Elaborado pela Autora*

```
5 - 10 - 1 - 4 - 4 - 8 - 3 - 4 - 3 - 5 -  
1 - 3 - 3 - 4 - 4 - 4 - 5 - 5 - 8 - 10 -  
  
foram realizadas 6 trocas  
foram realizadas 21 comparações  
Process returned 0 (0x0)   execution time : 0.094 s  
Press any key to continue.  
-
```

*Figura 70.2 - Execução do QuickSort em vetor aleatório de 10 elementos*  
*Fonte: Elaborado pela Autora*

```

75 - 43 - 7 - 31 - 75 - 29 - 19 - 53 - 7 - 93 - 43 - 19 - 65 - 50 - 19 - 42 - 80
- 71 - 22 - 36 - 13 - 63 - 20 - 6 - 8 - 38 - 79 - 11 - 85 - 99 - 4 - 70 - 27 -
74 - 57 - 35 - 65 - 15 - 22 - 11 - 31 - 52 - 20 - 40 - 4 - 44 - 55 - 79 - 5 - 69
- 83 - 49 - 48 - 7 - 78 - 40 - 14 - 9 - 29 - 47 - 13 - 63 - 45 - 11 - 22 - 45 -
47 - 87 - 43 - 83 - 14 - 2 - 56 - 49 - 12 - 28 - 95 - 85 - 88 - 10 - 69 - 19 -
90 - 26 - 67 - 68 - 31 - 51 - 92 - 54 - 34 - 60 - 39 - 6 - 3 - 11 - 9 - 3 - 51 -
55 -
2 - 3 - 3 - 4 - 4 - 5 - 6 - 6 - 7 - 7 - 7 - 8 - 9 - 9 - 10 - 11 - 11 - 11 - 11 -
12 - 13 - 13 - 14 - 14 - 15 - 19 - 19 - 19 - 19 - 19 - 20 - 20 - 22 - 22 - 22 - 26 -
27 - 28 - 29 - 29 - 31 - 31 - 31 - 34 - 35 - 36 - 38 - 39 - 40 - 40 - 42 - 43 -
43 - 43 - 44 - 45 - 45 - 47 - 47 - 48 - 49 - 49 - 50 - 51 - 51 - 52 - 53 - 54 -
55 - 55 - 56 - 57 - 60 - 63 - 63 - 65 - 65 - 67 - 68 - 69 - 69 - 70 - 71 - 74 -
75 - 75 - 78 - 79 - 79 - 80 - 83 - 83 - 85 - 85 - 87 - 88 - 90 - 92 - 93 - 95 -
99 -

foram realizadas 68 trocas
foram realizadas 792 comparações
Process returned 0 (0x0)   execution time : 0.105 s
Press any key to continue.

```

*Figura 71 - Execução do QuickSort em vetor aleatório de 100 elementos  
Fonte: Elaborado pela Autora*

```

16 - 86 - 6 - 46 - 87 - 49 - 16 - 87 - 47 - 72 - 43 - 65 - 58 - 80 - 5 - 56 - 61
- 9 - 93 - 70 - 10 - 38 - 5 - 73 - 29 - 14 - 17 - 5 - 47 - 70 - 16 - 14 - 73 -
45 - 53 - 47 - 47 - 71 - 65 - 44 - 44 - 6 - 11 - 2 - 88 - 46 - 27 - 69 - 67 - 1
- 83 - 35 - 82 - 25 - 19 - 54 - 15 - 94 - 73 - 57 - 46 - 20 - 100 - 71 - 73 - 70
- 68 - 17 - 68 - 46 - 66 - 75 - 74 - 7 - 84 - 8 - 74 - 29 - 72 - 41 - 49 - 68 -
69 - 91 - 6 - 3 - 50 - 64 - 31 - 94 - 70 - 23 - 28 - 42 - 100 - 95 - 58 - 60 -
96 - 6 -
1 - 2 - 3 - 5 - 5 - 5 - 6 - 6 - 6 - 6 - 7 - 8 - 9 - 10 - 11 - 14 - 14 - 15 - 16
- 16 - 16 - 17 - 17 - 19 - 20 - 23 - 25 - 27 - 28 - 29 - 29 - 31 - 35 - 38 - 41
- 42 - 43 - 44 - 44 - 45 - 46 - 46 - 46 - 46 - 47 - 47 - 47 - 47 - 49 - 49 - 50
- 53 - 54 - 56 - 57 - 58 - 58 - 60 - 61 - 64 - 65 - 65 - 66 - 67 - 68 - 68 - 68
- 69 - 69 - 70 - 70 - 70 - 70 - 71 - 71 - 72 - 72 - 73 - 73 - 73 - 73 - 74 - 74
- 75 - 80 - 82 - 83 - 84 - 86 - 87 - 87 - 88 - 91 - 93 - 94 - 94 - 95 - 96 - 100
- 100 -

foram realizadas 71 trocas
foram realizadas 742 comparações
Process returned 0 (0x0)   execution time : 0.164 s
Press any key to continue.

```

*Figura 71.1 - Execução do QuickSort em vetor aleatório de 100 elementos  
Fonte: Elaborado pela Autora*

```

80 - 84 - 48 - 63 - 68 - 13 - 82 - 56 - 37 - 32 - 24 - 85 - 27 - 78 - 93 - 90 -
86 - 76 - 28 - 6 - 13 - 64 - 98 - 82 - 8 - 71 - 12 - 82 - 94 - 43 - 91 - 23 - 77
- 70 - 33 - 100 - 43 - 56 - 34 - 57 - 42 - 10 - 23 - 25 - 31 - 43 - 17 - 26 - 2
4 - 59 - 81 - 47 - 49 - 74 - 63 - 13 - 87 - 2 - 64 - 82 - 68 - 44 - 17 - 74 - 94
- 5 - 37 - 14 - 20 - 31 - 86 - 11 - 5 - 89 - 86 - 91 - 78 - 7 - 71 - 35 - 21 -
71 - 1 - 84 - 71 - 25 - 7 - 85 - 8 - 76 - 77 - 13 - 81 - 25 - 20 - 93 - 99 - 12
- 14 - 30 -
1 - 2 - 5 - 5 - 6 - 7 - 7 - 8 - 8 - 10 - 11 - 12 - 12 - 13 - 13 - 13 - 13 - 14 -
14 - 17 - 17 - 20 - 20 - 21 - 23 - 23 - 24 - 24 - 25 - 25 - 25 - 26 - 27 - 28 -
30 - 31 - 31 - 32 - 33 - 34 - 35 - 37 - 37 - 42 - 43 - 43 - 43 - 44 - 47 - 48 -
49 - 56 - 56 - 57 - 59 - 63 - 63 - 64 - 64 - 68 - 68 - 70 - 71 - 71 - 71 - 71 -
74 - 74 - 76 - 76 - 77 - 77 - 78 - 78 - 80 - 81 - 81 - 82 - 82 - 82 - 82 - 84 -
84 - 85 - 85 - 86 - 86 - 86 - 87 - 89 - 90 - 91 - 91 - 93 - 93 - 94 - 94 - 98 -
99 - 100 -

foram realizadas 62 trocas

foram realizadas 676 comparações

Process returned 0 (0x0)    execution time : 0.139 s
Press any key to continue.

```

*Figura 71.2 - Execução do QuickSort em vetor aleatório de 100 elementos*

*Fonte: Elaborado pela Autora*

### 8.3 - Implementação

```
#include <iostream>
#include <time.h>

using namespace std;

int const TAM = 100;
int vetor[TAM];
int troca = 0;
int comparacao = 0;

void aleatorio (int vetor_[TAM])
{
    for(int a=0; a<TAM; a++){
        vetor_[a] = rand() % TAM + 1;
    }
}

void crescente(int vetor_[TAM])
{
    for(int i = 1; i < TAM; i++)
    {
        vetor_[i]= i;
    }
}

void decrescente(int vetor_[TAM])
{
    for(int d=0; d<=TAM; d++)
    {
```

```

        vetor[d]= TAM - d;
    }
}
void exhibe (int vetor_[TAM])
{

    for(int e = 0; e < TAM; e++)
    {
        cout << vetor_[e] << " - ";

    }
    cout << endl;
}

int posicao(int *vetor_, int inicio, int fim){

    int controle_de_inicio = inicio;
    int pivot = vetor_[fim];
    int auxiliar; // auxiliar de troca

    for(int posi = inicio; posi < fim; posi++){
        comparacao++;
        if(vetor_[posi] < pivot){

            auxiliar = vetor_[posi];
            vetor_[posi] = vetor_[controle_de_inicio];
            vetor_[controle_de_inicio] = auxiliar;
            controle_de_inicio = controle_de_inicio + 1;

        }
    }
}

```



```

        auxiliar = vetor_[fim];
        vetor_[fim] = vetor_[controle_de_inicio];
        vetor_[controle_de_inicio] = auxiliar; //
controle de pivot
        return controle_de_inicio;
}

void quicksort ( int *vetor_, int inicio, int fim){

    int armazena; // armazena a posição do pivô

    if(inicio < fim){

        armazena = posicao(vetor_, inicio, fim);
        quicksort(vetor_, inicio, armazena - 1);
        quicksort(vetor_, armazena+1, fim);
        // sistema recursivo
        troca++;

    }

}

int main()
{
    srand(time(NULL));

    aleatorio(vetor);
    exhibe(vetor);
    quicksort(vetor, 0, TAM-1);

```

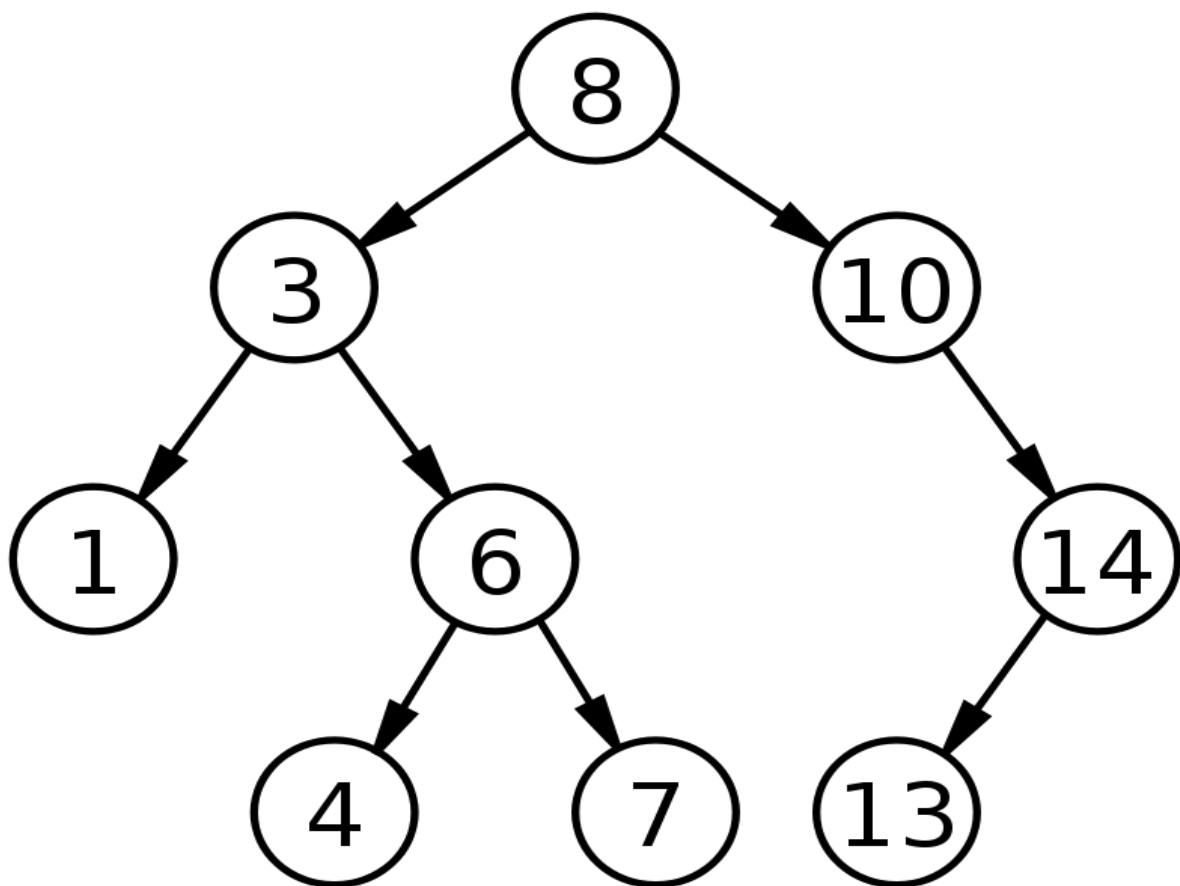
```
// tam menos 1 pq eu começo com 0
exibe(vetor);

    cout << endl << " foram realizadas " << troca << "
trocas " << endl;
    cout << endl << " foram realizadas " << comparacao
<< " comparações " << endl;
    return 0;
}
```

## 9 - Árvore Binária de Busca

### 9.1 - Resumo do método

Uma árvore binária de busca ou Árvore binária de pesquisa, consiste em uma estrutura de dados de árvore binária, sintetizadas em nós, onde os nós da subárvore esquerda são compostos pelos elementos menores que a raiz e os nós da subárvore direita, formados pelos valores maiores que a raiz, sendo essa a formação padrão. Contudo, existem as árvores invertidas, zig-zag e afins.



A árvore binária de busca é formada por alguns elementos, sendo:

**Nós:** Itens guardados na árvore;

**Raiz:** É o nó do topo da árvore;

**Filhos:** Nós que vem depois de outros nós;

**Pais:** Nós que vem antes de outros nós;

**Folhas:** São os últimos nós da árvore, nós que não tem filhos;

## 9.2 - Complexidade

A complexidade de uma Árvore binária de busca depende da altura da árvore. A complexidade do pior caso, será equivalente à altura da árvore. Portanto, o pior caso pode ser  $O(n)$ , onde  $n$  é o número de nós da árvore. No entanto, no melhor caso, com uma árvore perfeitamente balanceada, teremos uma busca eficiente, para isso precisaríamos ter elementos inseridos de maneira recursiva e em ordem simétrica, ou seja, em ordem crescente, como é o caso da implementação adiante. Ademais, o pior caso da inserção acontece quando deve-se percorrer toda árvore para inserir um novo elemento, já para busca temos o pior caso quando o elemento pesquisado não está na árvore.

## 9.2 - Implementação

```
#include <iostream>
#include <locale.h>
using namespace std;
```

```
typedef struct arvore Tarvore;
```

```
struct arvore
{
    int dado;
    Tarvore * direita;
    Tarvore * esquerda;
};
```

```
void insere(Tarvore *&arvore, int valor)
// função de inserir que recebe como parâmetro a
árvore e referência, já que
```

```
// vamos alterar, verificamos se a arvore está vazia,
pois se estiver alocamos
// um espaço de memória, então arvore no campo dado
vai receber o valor que
// estou passando e os ponteiros da direita e
esquerda vão receber null
// ao inserir mais um valor vamos verificar se ele é
maior ou menor que a raiz.
// ex: se for maior, vamos fazer uma chamada
recursiva passando o ponteiro
// a direita e o valor.
// o mesmo será feito para a esquerda, mas sendo
menor.
```

```
// é interessante ressaltar que: sempre vai estar
caindo no primeiro if, pois
//estimulamos que os filhos serão nulos
{
    if(arvore == NULL)
    {
        arvore = new(Tarvore);
        arvore->dado = valor;
        arvore->esquerda = NULL;
        arvore->direita = NULL;
    }
    else
    {
        if (valor > arvore->dado)
        {
            insere(arvore->direita, valor);
        }
    }
}
```

```

        else if (valor < arvore->dado) //para evitar
repetições
        {
            insere(arvore->esquerda, valor);
        }
    }
}

```

```

void ordem_simetrica(Tarvore *arvore)

```

```

// função de ordem simétrica que  exhibe os valores da
esquerda, o visita e
//depois a direita, colocando os números de maneira
crescente

```

```

{
    if(arvore != NULL)
    {
        ordem_simetrica(arvore->esquerda);
        cout << endl << arvore->dado << "    ";
        ordem_simetrica(arvore->direita);
    }
}

```

```

Tarvore* busca(Tarvore *arvore,int valor )

```

```

// função de busca que retorna um ponteiro do tipo
Tárvore, passando
// a arvore por parâmetro e o valor que iremos
procurar, nada vai ser alterado.

```

```

// Adiante, verificamos se a árvore está vazia, pq se
// não tiver ninguém ela vai
// retornar que o ponteiro é nulo. Depois verificamos
// se o valor que buscamos
// está na raiz, se sim retornamos ele, para ai
// procurar na esquerda se for
// menor ou na direita se for maior

{

    if(arvore == NULL){

        return NULL;
    }
    else{

        if(valor == arvore -> dado){
            return arvore;
        }
        else if (valor < arvore->dado) {
            busca(arvore ->direita, valor);
        }
        else if (valor > arvore -> dado){
            busca(arvore ->esquerda, valor);
        }
    }
}

int main()
{
    setlocale(LC_ALL, "portuguese");

    Tarvore* arv = NULL;

```

```

Tarvore *valor;

    insere(arv, 9);
    insere(arv, 12);
    insere(arv, 10);
    cout << endl << " Valores em ordem simétrica: "
<< endl;
    cout << endl;
    ordem_simetrica(arv);
    cout << endl;
    cout << endl;
    cout << endl <<
"-----
-----" << endl;
    cout << endl << "                               Mais
Informações                               " << endl;
    valor = busca(arv,9);
    cout << endl;
    cout << endl << " O valor encontrado na busca é:
" << valor -> dado << endl;
    cout << endl;
    cout << endl <<
"-----
-----" << endl;

    return 0;
}

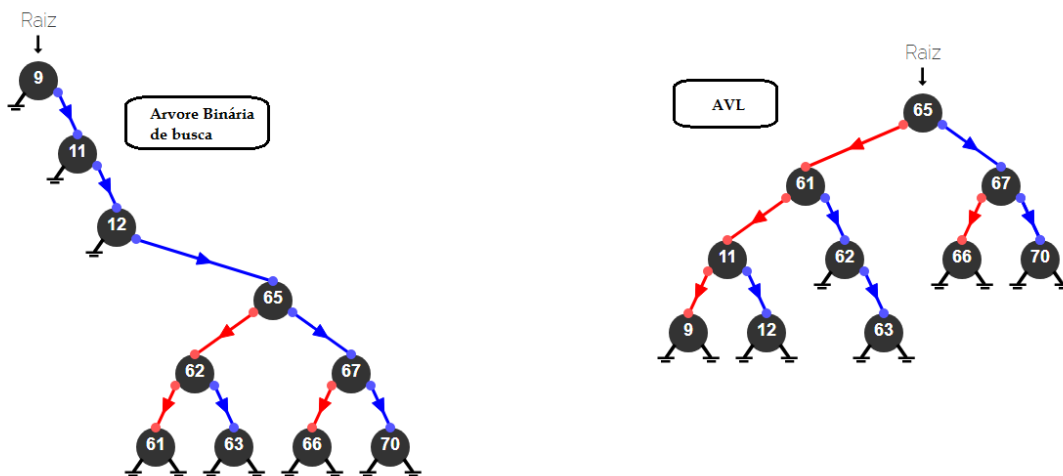
```



## 10 - AVL

### 10.1 - Resumo do método e comparação com a ABB

Uma árvore AVL é uma árvore binária de busca, ou seja, nos preocupamos com que os elementos de uma subárvore a esquerda contenha elementos menores que a raiz e a direita maiores que a raiz. Contudo, ela é uma árvore binária de busca mais eficiente, pois tem a seguinte propriedade: para todo nó dentro de uma AVL, a diferença de altura entre a subárvore esquerda e direita daquele nó vai ser no máximo um. Portanto, a mesma altura ou a altura+1. Sendo assim, ela garante esse balanceamento através de rotações, que podem ser as simples (a direita ou a esquerda) ou duplas (dupla a direita ou dupla a esquerda). Para mais, relacionando-as no que desrespeito a complexidade, temos a imagem a baixo como auxílio explicativo.



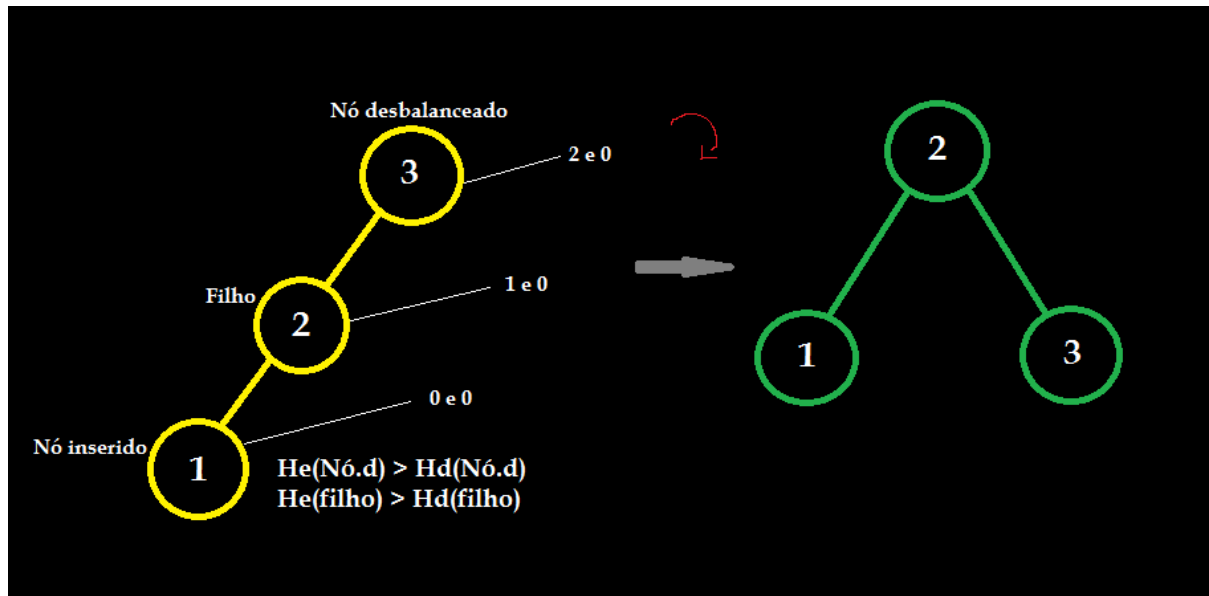
*Fonte: Elaborado pela Autora*

Sendo assim, percebemos que a árvore binária de busca não segue a propriedade de nós regulados e tem uma altura maior, já na AVL, embora tenha o mesmo número de elementos, os nós estão dispostos de maneira que a altura da árvore diminui. Dito isso, sabemos que a complexidade de uma árvore binária de busca é baseada na sua altura, onde o pior caso é  $O(n)$ , e o melhor caso  $O(\log n)$  e é exatamente aí que a AVL a supera, pela sua propriedade de regulação de nós a AVL realiza rotações que minimizam o número de comparações efetuadas na pior caso, atingindo a complexidade de  $O(\log n)$  tanto para o melhor caso, quanto para o pior.

## 10.2 - Rotações

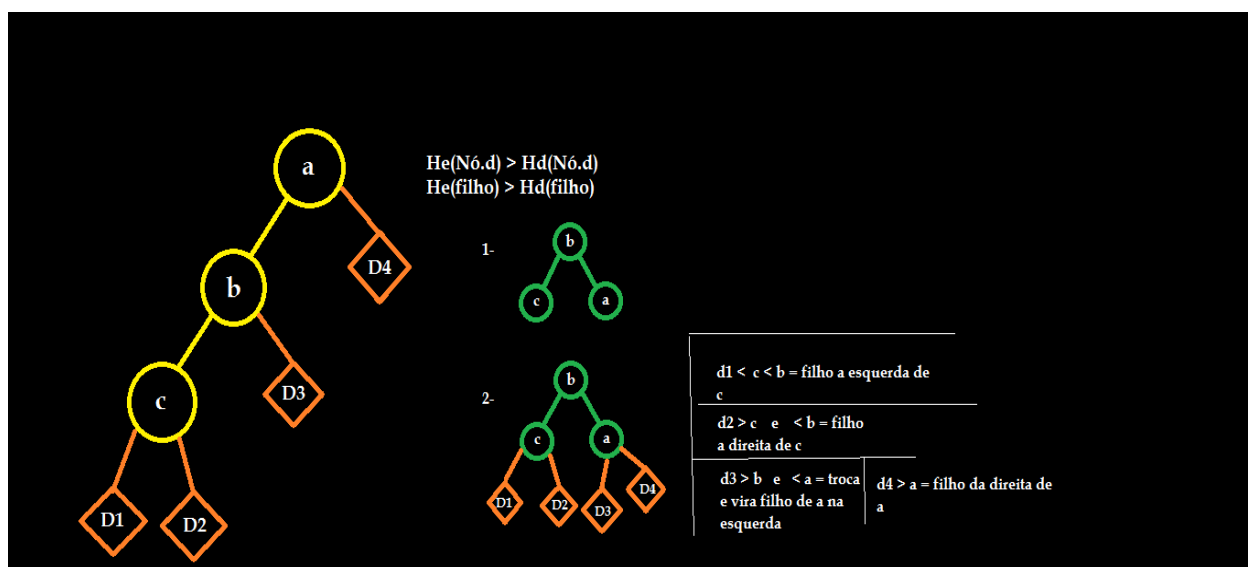
**Rotação a direita:** Ela acontece quando a altura do nó desregulado a esquerda é maior que a altura do nó desregulado a direita e a altura do filho a esquerda é maior que a altura do filho a direita.

exemplo1:



Fonte: Elaborado pela Autora

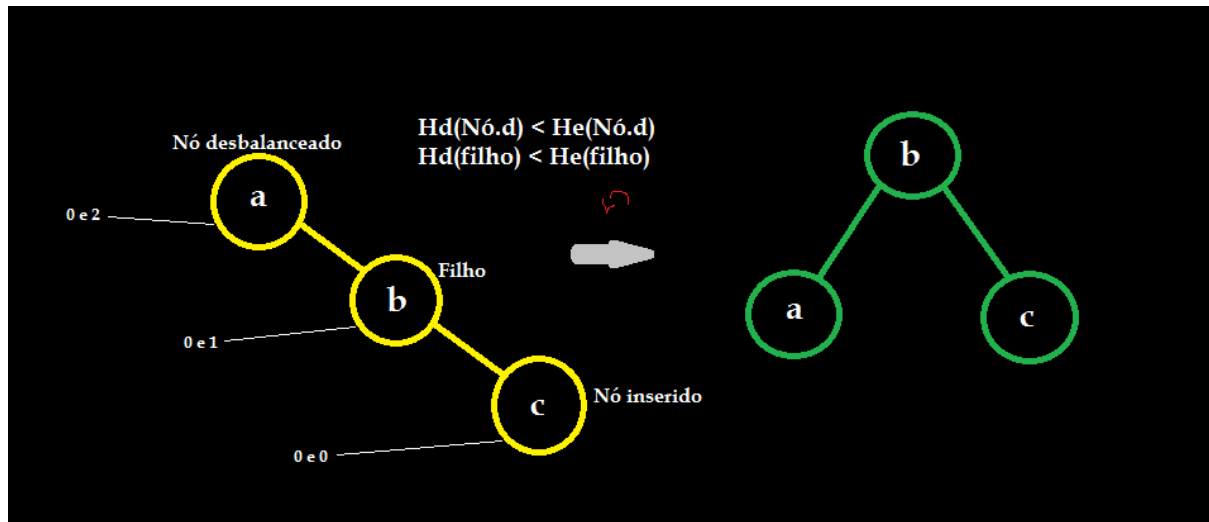
Contudo, suponhamos que existam mais elementos nessa árvore. Por exemplo:



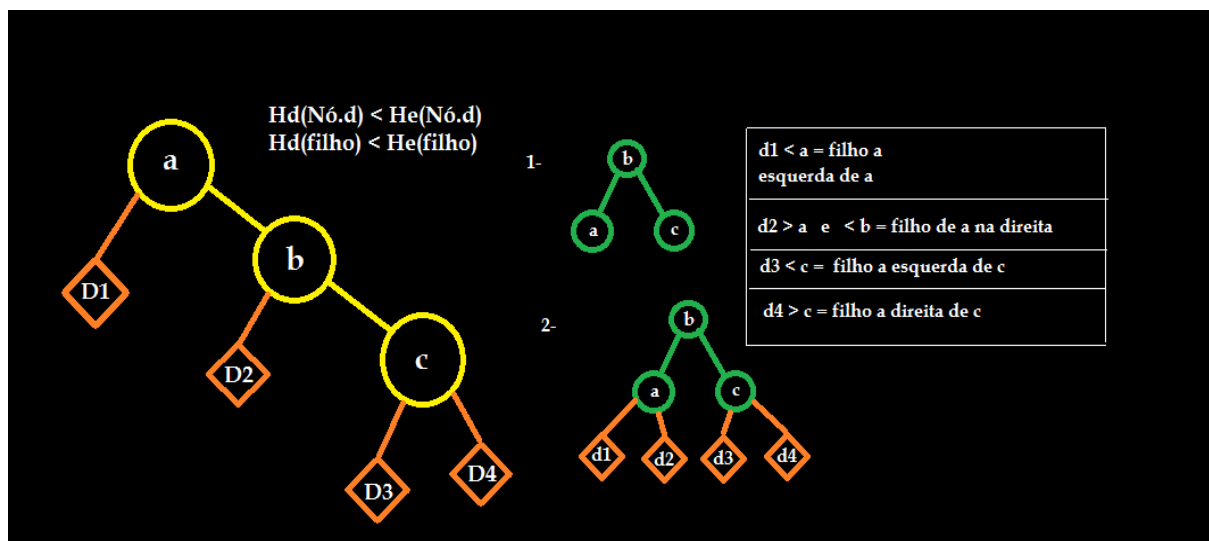
Fonte: Elaborado pela Autora

Ou seja, antes d3 era filho a direita de b, depois precisou ser alterado, virando filho a esquerda de a.

**Rotação a esquerda:** A rotação a esquerda segue a mesma linha da direita, mas de maneira contrária. Ela ocorre quando a altura do nó desregulado a esquerda é menor que a altura do nó desregulado a direita e a altura do filho a esquerda é menor que a altura do filho da direita. Por exemplo:



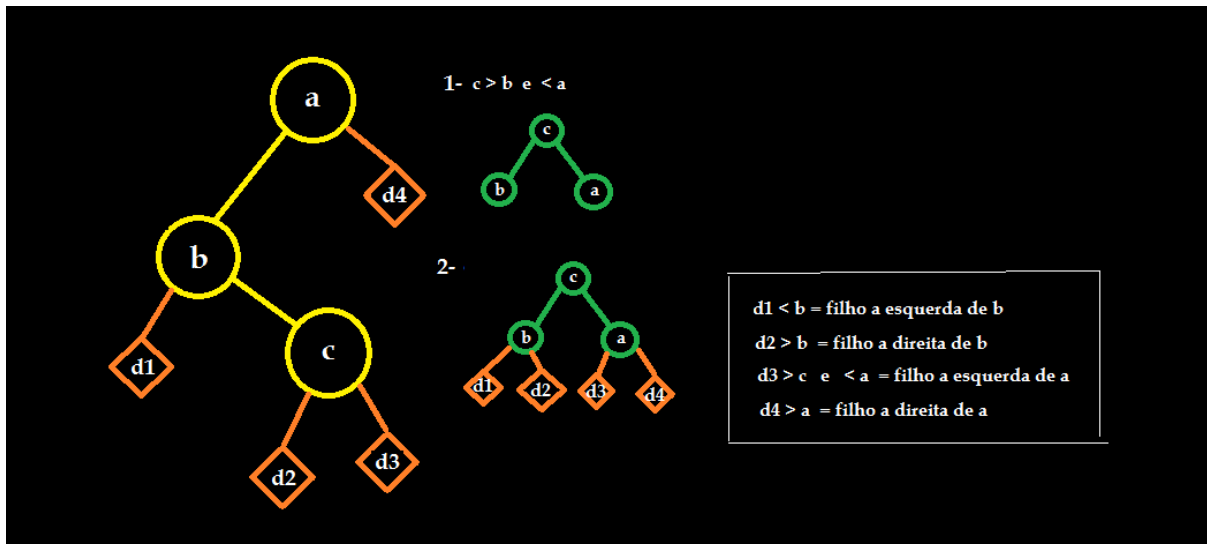
Para mais, assim como na R.D é interessante demonstrar com filhos. Sendo assim, temos:



Ou seja, d2 era filho da esquerda do filho da direita(b) e virou filho a direita do filho da esquerda(a).

Entendendo isso fica fácil compreender as duplas, tanto a direita, quando a esquerda. Ademais, ao contrário do que se pode pensar uma dupla rotação não significa fazer duas vezes a mesma rotação.

Por exemplo:



Ou seja, de forma análoga as anteriores é realizada uma troca de ponteiros, como o ponteiro que apontava para “a” e agora aponta para “c” dentre outras alterações. Sendo assim, entendemos que a **rotação dupla a direita** consiste na rotação a esquerda com filho e a direita com o pai. Sendo assim, é evidente que a **rotação dupla a esquerda** combina uma rotação a direita no filho e uma rotação a esquerda no pai.

### 10.3 - Implementação

```
#include <iostream>
```

```
#include <locale.h>
```

```
using namespace std;
```

```
typedef struct elemento Tarvore;
```

```
struct elemento{
    int dado;
    Tarvore *esquerda;
    Tarvore *direita;
};
```

```
//Função do tipo inteiro que calcula a altura da
//árvore, onde é passado por [
//referência a árvore, com variáveis de altura a
//direita e esquerda que vão
//servir para estipular qual a maior altura.
Primeiro, verifico se a árvore é
//nula, pois se for retornará 0, caso contrário será
//feita uma recurssão para
//receber a altura passando o ponteiro da esquerda e
//da direita.
//Por fim, verifica se a direita é maior que a
//esquerda, se for retorna
//a altura da direita + 1, se não retorna a altura da
//esquerda +1
```

```
int altura(Tarvore *arvore)
{

    int altura_direita;
    int altura_esquerda;

    if( arvore == NULL){
        return 0;
    } else
    {
        altura_esquerda = altura(arvore->esquerda);
        altura_direita = altura(arvore->direita);
        if (altura_direita > altura_esquerda) return
altura_direita+1;
```

```

        else return altura_esquerda+1;

    }
}

//Função que rotaciona a direita, nela é passado por
//parâmetro e referência o nó
//desregulado e dois ponteiros do tipo Tarvore filho
//e nó_inserido, onde o filho
//vai receber o ponteiro do nó_desregulado a
//esquerda, o nó_inserido vai rece-
//ber o ponteiro do filho a direita, o ponteiro do
//filho a direita vai receber
//o nó_desregulado, o ponteiro do nó_desregulado a
//esquerda recebe o nó_inserido
//e o nó_desregulado recebe o filho, tornando o filho
//a nova raiz.

```

//explicação mais lixo kk que ue já dei na minha vida, juro que eu tentei kkkk

```

void rotacao_direita(Tarvore *&no_desregulado)
{

    Tarvore *filho, *no_inserido;
    filho = no_desregulado->esquerda;
    no_inserido = filho->direita;
    filho->direita = no_desregulado;
    no_desregulado->esquerda = no_inserido;

    no_desregulado = filho;
}

```

```
}
```

```
//Função que rotaciona a esquerda, funciona com a  
mesma lógica da direita só que  
//ao contrário
```

```
void rotacao_esquerda(Tarvore *&no_desregulado)  
{
```

```
    Tarvore *filho, *no_inserido;
```

```
    filho = no_desregulado->direita;  
    no_inserido = filho->esquerda;  
    filho->esquerda = no_desregulado;  
    no_desregulado->direita = no_inserido;
```

```
    no_desregulado = filho;
```

```
}
```

```
// Função de rotação dupla a direita, nela é chamada  
a rotação a esquerda  
// para o filho, depois a direita para o pai
```

```
void rotacao_dupla_direita(Tarvore *&arvore)  
{
```

```
    rotacao_esquerda(arvore->esquerda);  
    rotacao_direita(arvore);
```

```
}
```

```
// A mesma coisa da função anterior, só que ao
contrário, ou seja, vou chamar
//a rotação a direita para o filho e a esquerda para
o pai
```

```
void rotacao_dupla_esquerda(Tarvore *&arvore)
{

    rotacao_direita(arvore->direita);
    rotacao_esquerda(arvore);
}
```

```
void balancear(Tarvore*&arvore, int valor){

    int altura_direita, altura_esquerda;
    altura_esquerda = altura(arvore->esquerda);
    altura_direita = altura(arvore->direita);

    // será verificado qual nó está desbalanceado,
se está desbalanceado
    // para esquerda ou pra direita

    if((altura_direita - altura_esquerda) > 1){
        // esse primeiro if é auto explicativo
        if
(altura(arvore->direita->direita)>altura(arvore->dire
ita->esquerda)){
            //se a altura da direita a direita for
maior
            // ele vai chamar a rotação a esquerda,
passando a árvore e se for
```



```

        // menor vai chamar a rotação dupla a
esquerda
        rotacao_esquerda(arvore);
    }
    else{

        // caso contrário faz uma rde

        rotacao_dupla_esquerda(arvore);
    }
}
// a mesma coisa ideia
else if((altura_esquerda - altura_direita) > 1){
    if
(altura(arvore->esquerda->esquerda)>altura(arvore->es
querda->direita)){

        rotacao_direita(arvore);
    }
    else{

        rotacao_dupla_direita(arvore);
    }
}
}

int insere(Tarvore *&arvore, int valor)
{
    if(arvore == NULL)
    {
        arvore = new(Tarvore);
        arvore->dado = valor;
    }
}

```

```

    arvore->direita = NULL;
    arvore->esquerda = NULL;

    // se a árvore for nula vamos alocar um espaço de
    memória, além de fazer
    //receber o valor e colocar os ponteiros para null

    return 0;

}
else
{ // se for menor ou igual insere a esquerda, se
não a direita
    if(valor >= arvore->dado)
    {
        insere(arvore->direita,valor);
    }
    else
    {
        insere(arvore->esquerda,valor);
    }
    balancear(arvore,valor);
}
}

```

```

void ordem_simetrica(Tarvore *arvore){

    if(arvore != NULL){

        ordem_simetrica(arvore->esquerda);
        cout << arvore->dado << "  ";
    }
}

```

```

        ordem_simetrica(arvore->direita);
    }
}

int main()
{
    setlocale(LC_ALL, "portuguese");

    Tarvore *arv = NULL;

    insere(arv,9);
    insere(arv,13);
    insere(arv,19);
    insere(arv,2);
    insere(arv,1);

    cout << endl << "                        Informações da AVL"
    << endl;
    cout << endl << endl;
    cout << "Percursso simétrico: ";
    cout << endl;
    cout << endl;
    ordem_simetrica(arv);

    cout << endl << endl;

    cout << "Valor a esquerda: " << arv->esquerda->dado
    << endl;
    cout << "Valor da Raiz : " << arv->dado << endl;
    cout << "Valor a direita : " << arv->direita->dado
    << endl;

```

```
    cout << "Altura da árvore : " << altura(arv) <<  
endl;
```

```
    cout << endl << endl;
```

```
    return 0;  
}
```