

UNIVERSIDADE FEDERAL DE VIÇOSA

Relatório sobre trabalho prático de Algoritmo e Estruturas de  
Dados

VINÍCIUS SIMIM RIBEIRO

MAT : EF02645

Samuel Silva

MAT: EF02662

## Relatório sobre trabalho prático de Algoritmo e Estruturas de Dados

Trabalho apresentado como requisito parcial para a obtenção de créditos para aprovação na disciplina de Algoritmo e Estruturas de Dados do curso de Ciência da Computação, Universidade Federal de Viçosa.

Prof.<sup>a</sup> Dr.<sup>a</sup> Thais R.M. Braga Silva

## INTRODUÇÃO

Os algoritmos fazem parte do dia-a-dia das pessoas, eles podem ser entendidos como quaisquer tipos de instruções que tem como objetivo a obtenção de uma resolução exata ou até mesmo aproximada. Estudantes da computação em geral são os principais responsáveis pelo estudo de algoritmos, os algoritmos os possibilitam resolver inúmeros problemas sendo esses pequenos ou grandes. Como o algoritmo é um conceito muito importante para estudantes da computação, no decorrer de sua formação ele estuda bem a fundo os vários conceitos e técnicas utilizadas para uma melhor implementação de tal. Um desses estudos é nominado como algoritmos de Ordenação, eles são responsáveis por organizar diferentes tipos de elementos em ordens ascendentes ou descendentes de acordo com decisões tomadas pelo programador ou pelo usuário do sistema. Os algoritmos de ordenação estudados em sala de aula ao decorrer do semestre foram, BubbleSort, SelectSort, InsertSort, ShellSort, QuickSort e HeapSort. Cada um desses algoritmos possuem complexidades assintóticas divergentes tanto em número de comparações quanto em número de movimentações, devido a isso é que o estudo desses algoritmos a fundo é importante, pois até o pior dos algoritmos de ordenação pode ser considerado o melhor para se utilizar dependendo das variáveis do problema a ser solucionado. Contudo correspondente a esses fatos supracitados que nossa Prof. Thais Silva, com o intuito de nos proporcionar melhor entendimento da matéria estudada dentro de sala, aplicou um trabalho onde o objetivo principal foi implementar os 6 algoritmos de ordenação compreendidos em sala em cima de um sistema **TAD dicionário (TP 1)** implementado no início do semestre. Para a realização desse trabalho foi utilizado os conhecimentos absorvidos em classe e extraclasse, ferramentas de suporte à compilação de algoritmos (IDE) como **CodeBlocks (16.01)**, conhecimentos práticos e teóricos na linguagem de programação C e duas máquinas, para a execução do trabalho.

## DESENVOLVIMENTO

A leitura das especificações postadas na plataforma virtual de suporte ao aluno da faculdade (PVANET) foi o primeiro passo para dar continuidade ao início do trabalho, após ler algumas vezes chegamos a um entendimento, mas para esclarecer melhor e tirar algumas dúvidas que surgiram em seguida a leitura procuramos o monitor da disciplina. O monitor nos esclareceu bem o funcionamento do trabalho e como ele devia ser implementado, a partir daí iniciamos a implementação do código. O código implementado, possui os seguintes arquivos:

main.c

TAD\_Numerolinhas.h

TAD\_Numerolinhas.c

TAD\_Palavras.h

TAD\_Palavras.c

TAD\_Lista\_Palavras.h

TAD\_Lista\_Palavras.c

TAD\_Letra\_Alfabeto.h

TAD\_Letra\_Alfabeto.c

TAD\_Dicionario.h

TAD\_Dicionario.c

O arquivo TAD\_Numerolinhas.h é composto por um tipo abstrato de dados (TAD) e suas operações como ilustrado abaixo:

```

1  #ifndef TAD_NUMEROLINHAS_H_INCLUDED
2  #define TAD_NUMEROLINHAS_H_INCLUDED
3
4  typedef struct{
5      int Nlinhas;
6  }Numero_Linhas;
7
8  typedef struct Celula *Apontador;
9  typedef struct Celula{
10     Numero_Linhas Item_Nlinhas;
11     struct Celula *pProx;
12 }TAD_Celula;
13
14 typedef struct{
15     Apontador pPrimeiro;
16     Apontador pUltimo;
17     int Total_linhas;
18 }TAD_NumeroLinhas;
19
20 void Cria_Lnumero_Vazia(TAD_NumeroLinhas *pLinhas);
21 int Verifica_Lnumero_Vazia(TAD_NumeroLinhas *pLinhas);
22 int Alimenta_Lnumero(TAD_NumeroLinhas *pLinhas, int x);
23 int Remove_Lnumero(TAD_NumeroLinhas *pLinhas);
24 int Verifica_Lnumero_Existentes(TAD_NumeroLinhas *pLinhas, int x);
25 void Printa_Total_Nlinhas(TAD_NumeroLinhas *pLinhas);
26 void Printa_Lnumero(TAD_NumeroLinhas *pLinhas);
27
28 #endif // TAD_NUMEROLINHAS_H_INCLUDED
29

```

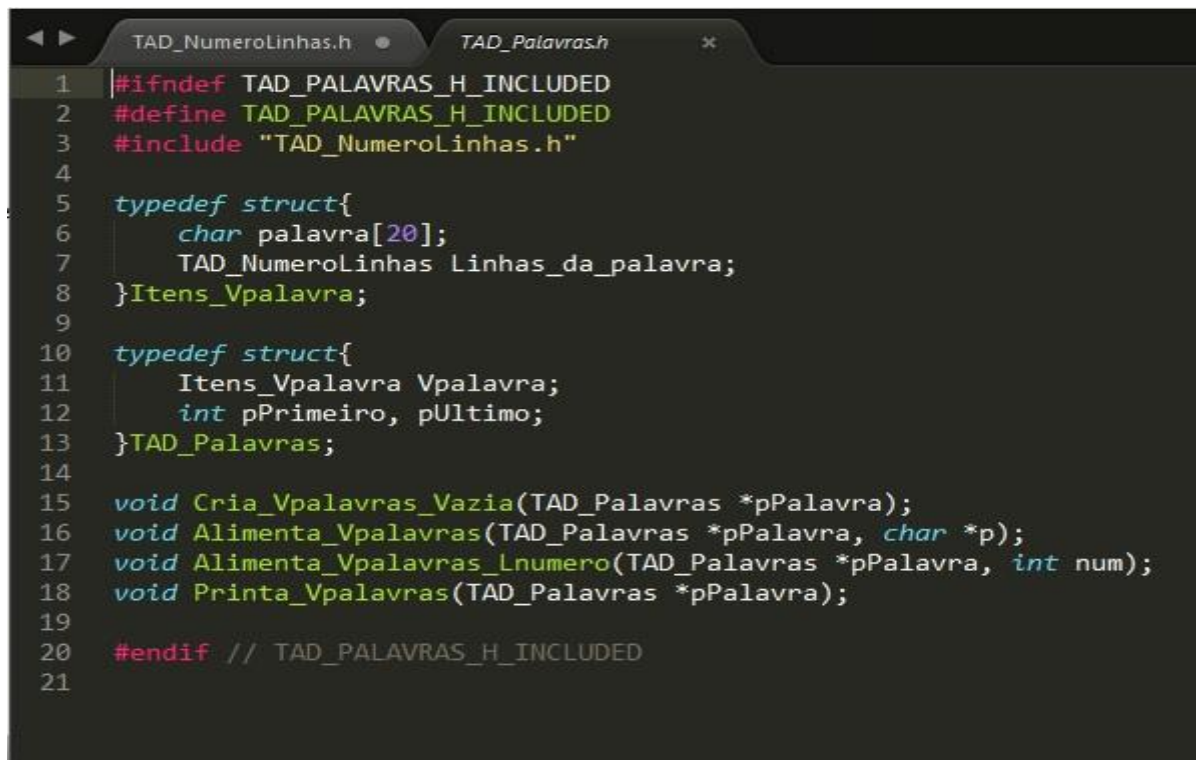
Figura 1- TAD\_Numerolinhas.h

Por decisão de projeto e pelas especificações contidas no documento do trabalho foi reaproveitado o TAD\_Numerolinhas.h utilizado no primeiro trabalho. Esse TAD foi construído em um formato de lista encadeada contendo em seu interior uma variável “Nlinhas” do tipo inteiro, uma variável “Total\_linhas” do tipo inteiro, outras variáveis ilustradas a cima cada uma com suas determinadas explicações e o cabeçalho das operações utilizadas pelo TAD.

Já o arquivo TAD\_Numerolinhas.c é composto pelos subprogramas e suas implementações como ilustrado a baixo:

Figura 2- TAD\_Numerolinhas.c

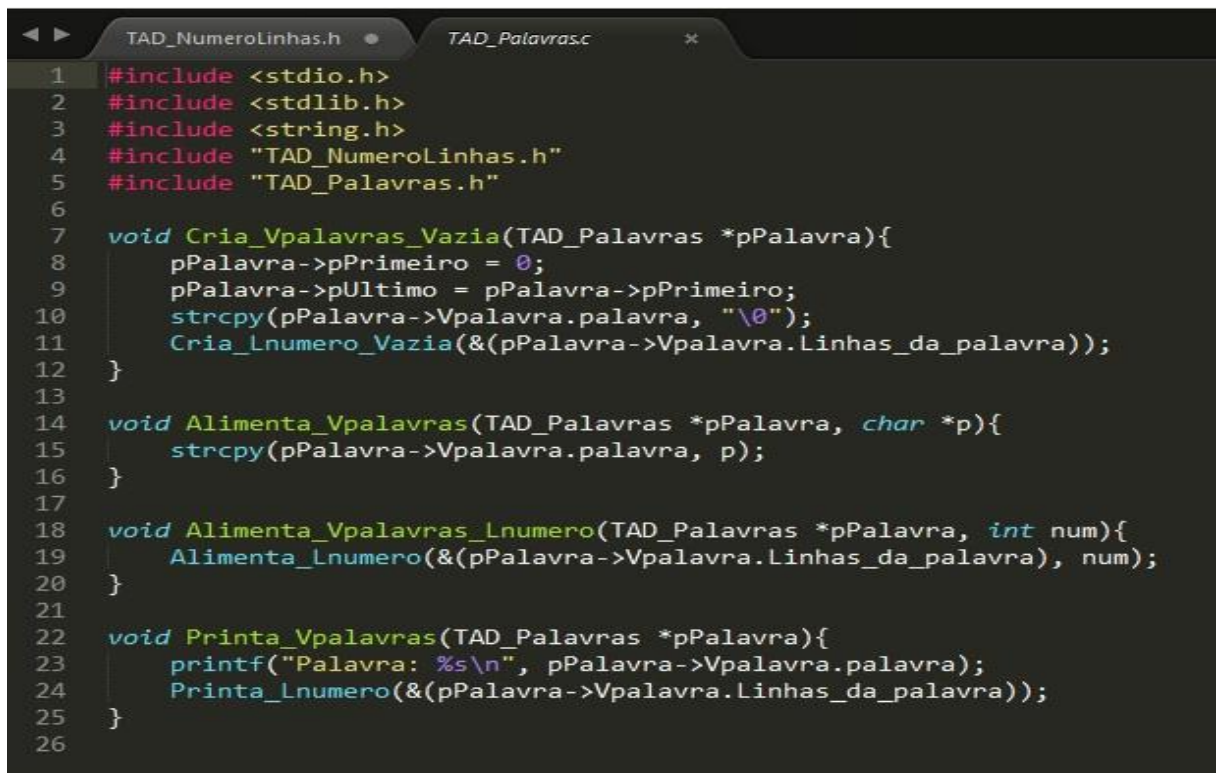
O arquivo TAD\_Palavras também foi reutilizado do primeiro trabalho e de acordo com a ilustração abaixo pode-se observar que em seu interior contém uma variável “palavra” do tipo char, uma variável “Linhas\_da\_palavra” do tipo TAD\_Numerolinhas, demais variáveis e o cabeçalho das operações a serem utilizadas posteriormente.



```
1  #ifndef TAD_PALAVRAS_H_INCLUDED
2  #define TAD_PALAVRAS_H_INCLUDED
3  #include "TAD_NumeroLinhas.h"
4
5  typedef struct{
6      char palavra[20];
7      TAD_NumeroLinhas Linhas_da_palavra;
8  }Itens_Vpalavra;
9
10 typedef struct{
11     Itens_Vpalavra Vpalavra;
12     int pPrimeiro, pUltimo;
13 }TAD_Palavras;
14
15 void Cria_Vpalavras_Vazia(TAD_Palavras *pPalavra);
16 void Alimenta_Vpalavras(TAD_Palavras *pPalavra, char *p);
17 void Alimenta_Vpalavras_Lnumero(TAD_Palavras *pPalavra, int num);
18 void Printa_Vpalavras(TAD_Palavras *pPalavra);
19
20 #endif // TAD_PALAVRAS_H_INCLUDED
21
```

Figura 3-TAD\_Palavras.h

Já o arquivo TAD\_Palavras.c é composto pelos subprogramas e suas implementações como ilustrado a baixo:



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "TAD_NumeroLinhas.h"
5  #include "TAD_Palavras.h"
6
7  void Cria_Vpalavras_Vazia(TAD_Palavras *pPalavra){
8      pPalavra->pPrimeiro = 0;
9      pPalavra->pUltimo = pPalavra->pPrimeiro;
10     strcpy(pPalavra->Vpalavra.palavra, "\\0");
11     Cria_Lnumero_Vazia(&(pPalavra->Vpalavra.Linhas_da_palavra));
12 }
13
14 void Alimenta_Vpalavras(TAD_Palavras *pPalavra, char *p){
15     strcpy(pPalavra->Vpalavra.palavra, p);
16 }
17
18 void Alimenta_Vpalavras_Lnumero(TAD_Palavras *pPalavra, int num){
19     Alimenta_Lnumero(&(pPalavra->Vpalavra.Linhas_da_palavra), num);
20 }
21
22 void Printa_Vpalavras(TAD_Palavras *pPalavra){
23     printf("Palavra: %s\\n", pPalavra->Vpalavra.palavra);
24     Printa_Lnumero(&(pPalavra->Vpalavra.Linhas_da_palavra));
25 }
26
```

Figura 4-TAD\_Palavras.c

O arquivo TAD\_Lista\_Palavras.h é a parte do código a se dar mais ênfase na documentação, pois todas as operações de ordenação pedidas nas especificações do trabalho foram implementadas nesse arquivo por decisão de projeto. Em seu interior como ilustrado a baixo contém na estrutura Itens\_Vlista\_Palavras uma variável “PontePalavra” do tipo TAD\_palavras com o intuito de fazer a ponte com os TAD’s anteriormente implementados. O interior da estrutura TAD\_Lista\_Palavras compõe-se de dois vetores de tipo Itens\_Vlista\_Palavras, um desses vetores é responsável por receber todas as palavras provenientes de um arquivo e armazena-las em um de seus 400 espaços, espaços esses que foram colocados considerando que no arquivo disponibilizado contém um máximo de 400 palavras iniciadas com cada letra do alfabeto, ou seja 400 palavras iniciadas com a letra “A”, 400 palavras iniciadas com a letra “B” e assim consecutivamente. Além dessa consideração a maquina utilizada apresentava erros no programa ao inicializarmos o vetor com um número de espaços cada vez maiores que 800, então por decisão de projeto visando uma lógica de acordo com as palavras dispostas no arquivo de texto disponibilizado para testes, decidimos considerar o vetor com o tamanho de entrada 400. O segundo vetor é responsável por armazenar a copia do primeiro vetor, para que ao chamar os algoritmos de ordenação seja passado para eles a copia do vetor e não o vetor original como pedido na documentação. As outras variáveis são provenientes do tipo de lista a ser utilizada na implementação do TAD, no caso lista por arranjo. E além dessas variáveis supracitadas o arquivo possui o cabeçalho de suas operações onde diferente do primeiro trabalho foram acrescentadas as operações de ordenação e de copia do vetor.



```

1  #ifndef TAD_LISTA_PALAVRAS_H_INCLUDED
2  #define TAD_LISTA_PALAVRAS_H_INCLUDED
3  #include "TAD_NumeroLinhas.h"
4  #include "TAD_Palavras.h"
5
6  typedef struct{
7      TAD_Palavras PontePalavra; //Tad palavras
8      int comparacao;
9      int movimentacao;
10 }Itens_Vlista_Palavras;
11
12 typedef struct{
13     Itens_Vlista_Palavras Vetor_Lpalavras[400]; //Vetor original de palavras
14     Itens_Vlista_Palavras Copia_Vetor[400]; //Cópia do vetor de palavras, para a ordenação
15     int pPrimeiro, pUltimo; //Índices
16     int contapalavra; //Contador de palavras
17 }TAD_Lista_Palavras;
18
19 //Cabeçalho das funções
20
21 void Cria_Vlista_Palavra(TAD_Lista_Palavras *pLista_Palavra);
22 int Verifica_Palavra_Existente(TAD_Lista_Palavras *pLista_Palavra, char *p);
23 int Alimenta_Vlista_Palavra(TAD_Lista_Palavras *pLista_Palavra, char *p, int num);
24 int Remove_Vlista_Palavra_Dada(TAD_Lista_Palavras *pLista_Palavra, char *p);
25 void Printa_Total_Palavras(TAD_Lista_Palavras *pLista_Palavra);
26 void Printa_Vlista_Palavra(TAD_Lista_Palavras *pLista_Palavra);
27 void Ordena_Boobles_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a);
28 void Ordena_Selects_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a);
29 void Ordena_Inserts_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a);
30 void Ordena_Shells_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a);
31 void Quicks_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a);
32 void Ordena(int Esq, int Dir, Itens_Vlista_Palavras *a);
33 void Particao(int Esq, int Dir, int *i, int *j, Itens_Vlista_Palavras *a);
34 void Ordena_Heaps_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a);
35 void Refaz(int Esq, int Dir, Itens_Vlista_Palavras *a);
36 void Constroi(Itens_Vlista_Palavras *a, TAD_Lista_Palavras *pLista_Palavra);
37 void Alimenta_Vetor_Copia(TAD_Lista_Palavras *pLista_Palavra);
38 void Printa_Vetor_Desordenado(TAD_Lista_Palavras *pLista_Palavra);
39 void Inicia_comparacao_movimentacao(Itens_Vlista_Palavras *a);
40
41 #endif // TAD_LISTA_PALAVRAS_H_INCLUDED

```

Figura 5 - TAD\_Lista\_Palavras.h

Já o arquivo TAD\_Palavras.c é composto pelos subprogramas e suas implementações como ilustrado a baixo:

```

4  #include <time.h>
5  #include "TAD_Numerolinhas.h"
6  #include "TAD_Palavras.h"
7  #include "TAD_Lista_Palavras.h"
8
9  void Cria_Vlista_Palavra(TAD_Lista_Palavras *pLista_Palavra){ ...
13 }
14
15 int Verifica_Palavra_Existente(TAD_Lista_Palavras *pLista_Palavra, char *p){ ...
27 }
28
29 int Alimenta_Vlista_Palavra(TAD_Lista_Palavras *pLista_Palavra, char *p, int num){ ...
49 }
50
51 int Remove_Vlista_Palavra_Dada(TAD_Lista_Palavras *pLista_Palavra, char *p){ ...
70 }
71
72 void Printa_Total_Palavras(TAD_Lista_Palavras *pLista_Palavra){
73     printf("Total de palavras: %d\n", pLista_Palavra->contapalavra);
74 }
75
76 void Printa_Vlista_Palavra(TAD_Lista_Palavras *pLista_Palavra){ ...
83 }
84
85 void Ordena_BoobleS_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){ ...
107 }
108
109 void Ordena_SelectS_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){ ...
132 }
133
134 void Ordena_InsertS_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){ ...
157 }
158
159 void Ordena_ShellS_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){ ...
190 }
191
192 void QuickS_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){ ...
199 }
200
201 void Ordena(int Esq, int Dir, Itens_Vlista_Palavras *a){ ...
209 }
210
211 void Particao(int Esq, int Dir, int *i, int *j, Itens_Vlista_Palavras *a){ ...
230 }
231
232 void Ordena_HeapS_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){ ...
249 }
250
251 void Constroi(Itens_Vlista_Palavras *a, TAD_Lista_Palavras *pLista_Palavra){ ...
260 }
261
262 void Refaz(int Esq, int Dir, Itens_Vlista_Palavras *a){ ...
277 }
278
279 void Alimenta_Vetor_Copia(TAD_Lista_Palavras *pLista_Palavra){ ...
285 }
286
287 void Printa_Vetor_Desordenado(TAD_Lista_Palavras *pLista_Palavra){ ...
293 }

```

Figura 6 - TAD\_Lista\_Palavras.c

Dos subprogramas acima ilustrados vamos dar ênfase nessa documentação aos algoritmos de ordenação já que é o intuito principal do trabalho.

O algoritmo de ordenação BubbleSort é considerado o pior de todos os algoritmos simples ao analisar sua função de complexidade assintótica, no pior caso, no melhor e no médio, em relação a comparação sua complexidade assintótica é  $O(n^2)$  (quadrática), já a função de complexidade de sua movimentação varia, pois caso seus elementos já estiverem ordenados o número de movimentações diminui a zero. A sua vantagem é que ele é considerado um algoritmo simples e estável, já sua desvantagem se dá pelo fato de que o arquivo independente de estar ordenado ou não leva a um caso de comparação quadrático. A implementação do método bolha está ilustrada abaixo:

```
void Ordena_BoobleS_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){
    int i, j,
    comparacao = 0,
    movimentacao = 0;
    Itens_Vlista_Palavras aux;
    clock_t Ticks[2]; //Função para cálculo de tempo
    Ticks[0] = clock(); //Função para cálculo de tempo
    //Ordenação do vetor de palavras
    //Controla as passadas
    for(i = 0; i < pLista_Palavra->contapalavra - 1; i++){
        //Percorre o vetor ordenando
        for(j = 1; j < pLista_Palavra->contapalavra - i; j++){
            if(strcmp(a[j].PontePalavra.Vpalavra.palavra, a[j-1].PontePalavra.Vpalavra.palavra) < 0 ){
                strcpy(aux.PontePalavra.Vpalavra.palavra, a[j].PontePalavra.Vpalavra.palavra);
                strcpy(a[j].PontePalavra.Vpalavra.palavra, a[j-1].PontePalavra.Vpalavra.palavra);
                strcpy(a[j-1].PontePalavra.Vpalavra.palavra, aux.PontePalavra.Vpalavra.palavra);
                movimentacao++; //Calcula quantidade de movimentações
            }
            comparacao++; //Calcula quantidade de comparações
        }
    }
    printf("Numero de comparacoes do algoritmo: %d\n", comparacao);
    printf("Numero de movimentacoes do algoritmo: %d\n", movimentacao);
    //Imprime o vetor já ordenado
    for(i = 0; i < pLista_Palavra->contapalavra; i++){
        printf("Palavra: %s\n", a[i].PontePalavra.Vpalavra.palavra);
    }
    Ticks[1] = clock(); //Função de tempo
    double Tempo = (Ticks[1] - Ticks[0]) * 1000.0 / CLOCKS_PER_SEC; //Função de tempo
    printf("Tempo gasto: %g ms.", Tempo); //Mostra o tempo gasto em cada ordenação

    return 0;
}
```

Figura 7 – BubbleSort

O Algoritmo de ordenação SelectSort é considerado uma versão melhorada do método Bolha, assim como o BubbleSort ele é um algoritmo de simples implementação, possui uma função de complexidade assintótica em relação a comparação no pior caso, no melhor e no médio  $O(n^2)$  (quadrática), porém o seu número de movimentações independente se o algoritmo está ordenado ou não é inferior ao do método Bolha possuindo uma função de complexidade assintótica  $O(n)$  (linear). Portanto o método de Seleção é considerado no quesito movimentação melhor que o Bolha e é bem utilizado quando o vetor é de tamanho pequeno e possui um grande número de elementos contido dentro de cada espaço. A implementação do método de Seleção está ilustrada abaixo:

```

//Algoritmo de ordenação SelectSort
void Ordena_SelectS_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){
    int i, j, Min,
        comparacao = 0,
        movimentacao = 0;
    Itens_Vlista_Palavras aux;
    clock_t Ticks[2]; //Função de tempo
    Ticks[0] = clock(); //Função de tempo
    /*Procura pela palavra que vem primeiro
    na ordem alfabética */

    //Percorre o dicionário
    for(i = 0; i < pLista_Palavra->contapalavra - 1; i++){
        Min = i; //Menor recebe i
        /*Percorre o vetor procurando pela palavra que vem antes
        de acordo com o alfabeto */
        for(j = i + 1; j < pLista_Palavra->contapalavra; j++){
            if(strcmp(a[j].PontePalavra.Vpalavra.palavra, a[Min].PontePalavra.Vpalavra.palavra) < 0)
                Min = j;
            comparacao++;
        }
        //Faz a troca do menor com o vetor da posição i
        strcpy(aux.PontePalavra.Vpalavra.palavra, a[Min].PontePalavra.Vpalavra.palavra);
        strcpy(a[Min].PontePalavra.Vpalavra.palavra, a[i].PontePalavra.Vpalavra.palavra);
        strcpy(a[i].PontePalavra.Vpalavra.palavra, aux.PontePalavra.Vpalavra.palavra);
        movimentacao++;
    }
    printf("Numero de comparacoes do algoritmo: %d\n", comparacao);
    printf("Numero de movimentacoes do algoritmo: %d\n", movimentacao);
    for(i = 0; i < pLista_Palavra->contapalavra; i++){
        printf("Palavra: %s\n", a[i].PontePalavra.Vpalavra.palavra);
    }
    Ticks[1] = clock(); //Função de tempo
    double Tempo = (Ticks[1] - Ticks[0]) * 1000.0 / CLOCKS_PER_SEC; //Função de tempo
    printf("Tempo gasto: %g ms.", Tempo); //Imprime o tempo gasto

    return 0;
}

```

Figura 8 – SelectSort

O algoritmo de ordenação InsertSort quando comparado com o BubbleSort e o SelectSort é considerado o melhor algoritmo de ordenação entre eles, assim como o Bolha e o de Seleção ele é considerado também um algoritmo simples, porém a diferença está quando a função de complexidade assintótica em relação a comparação e movimentação é  $O(n^2)$  (quadrática) apenas no pior caso, caso esse que se dá somente quando os elementos se encontrem completamente desordenados. Já o seu melhor caso possui função de complexidade assintótica no quesito comparação e movimentação  $O(n)$  (linear), caso esse que se dá unicamente quando os algoritmos se encontram totalmente ordenados. A implementação do método de Inserção está ilustrada abaixo:

```

//Algoritmo de ordenação Insert Sort

void Ordena_Insert5_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){
    int i, j,
        comparacao = 0,
        movimentacao = 0;
    Itens_Vlista_Palavras aux;
    clock_t Ticks[2]; //Função de tempo
    Ticks[0] = clock(); //Função de tempo

    //Percorre o vetor
    for(i = 1; i < pLista_Palavra->contapalavra; i++){
        strcpy(aux.PontePalavra.Vpalavra.palavra, a[i].PontePalavra.Vpalavra.palavra);
        j = i - 1; // j passa a estar uma posição atrás do i
        //Compara a palavra atual(i) com os anteriores e arrasta os maiores para posição j+1
        while((j >= 0) && ((strcmp(aux.PontePalavra.Vpalavra.palavra, a[j].PontePalavra.Vpalavra.palavra) < 0))){
            //arrasta o j, para a próxima posição
            strcpy(a[j+1].PontePalavra.Vpalavra.palavra, a[j].PontePalavra.Vpalavra.palavra);
            j--;
            movimentacao++;
        }
        //Insere a palavra na posição correta
        strcpy(a[j+1].PontePalavra.Vpalavra.palavra, aux.PontePalavra.Vpalavra.palavra);
        movimentacao++;
        comparacao++;
    }
    printf("Numero de comparacoes do algoritmo: %d\n", comparacao);
    printf("Numero de movimentacoes do algoritmo: %d\n", movimentacao);
    for(i = 0; i < pLista_Palavra->contapalavra; i++){
        printf("Palavra: %s\n", a[i].PontePalavra.Vpalavra.palavra);
    }
    Ticks[1] = clock(); //Função de tempo
    double Tempo = (Ticks[1] - Ticks[0]) * 1000.0 / CLOCKS_PER_SEC; //Função de tempo
    printf("Tempo gasto: %g ms.", Tempo); //Função de tempo

    return 0;
}

```

Figura 9 - InsertSort

O algoritmo de ordenação ShellSort é considerado uma melhoria do algoritmo de Inserção, diferente dos outros três apresentados anteriormente o ShellSort é classificado como um algoritmo de ordenação sofisticado, isso se dá pelo fato de que conjectura-se que sua função de complexidade assintótica em relação a comparação é  $O(n(\ln n))$ . A palavra conjectura-se é utilizada, pois a análise de complexidade do ShellSort é desconhecida, ela possui problemas matemáticos muito complexos, a começar pela sequência de Knuth utilizada para pré-ordenar o vetor antes de ordená-lo oficialmente. O Algoritmo de ordenação ShellSort é uma ótima opção à se utilizar para arquivos de tamanho moderado, além disso o algoritmo é de uma implementação simples. Sua desvantagem se dá pelo fato de que o tempo de execução de tal algoritmo varia de acordo com a ordem inicial do arquivo e também pelo motivo dele não ser estável. A implementação do método Concha está ilustrada abaixo:

```

void Ordena_Shells_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){
    int i, j, h = 1,
        comparacao = 0,
        movimentacao = 0;
    Itens_Vlista_Palavras aux;
    clock_t Ticks[2]; //Função de tempo
    Ticks[0] = clock(); //Função de tempo

    do{ //Calcula a sequência a ser usada
        h = ((h * 3) + 1);
    }while (h < pLista_Palavra->contapalavra);
    do{
        //Calcula o h da próxima passada
        h = h/3;
        //Controla as passadas do algoritmo pela sequência
        for(i = h; i < pLista_Palavra->contapalavra ; i++){
            strcpy(aux.PontePalavra.Vpalavra.palavra, a[i].PontePalavra.Vpalavra.palavra);
            j = i;
            //Compara a palavra atual com a palavra da posição j-h
            while ((strcmp(a[j-h].PontePalavra.Vpalavra.palavra, aux.PontePalavra.Vpalavra.palavra)) > 0){
                strcpy(a[j].PontePalavra.Vpalavra.palavra, a[j-h].PontePalavra.Vpalavra.palavra);
                j = j - h; //Descreece o j
                movimentacao++;
                if(j < h)
                    break;
            }
            //Faz a troca e coloca as palavras nos lugares corretos
            strcpy(a[j].PontePalavra.Vpalavra.palavra, aux.PontePalavra.Vpalavra.palavra);
            movimentacao++;
            comparacao++;
        }
    }while (h != 1);
    printf("Numero de comparacoes do algoritmo: %d\n", comparacao);
    printf("Numero de movimentacoes do algoritmo: %d\n", movimentacao);
    for(i = 0; i < pLista_Palavra->contapalavra; i++){
        printf("Palavra: %s\n", a[i].PontePalavra.Vpalavra.palavra);
    }
    Ticks[1] = clock(); //Função de tempo
    double Tempo = (Ticks[1] - Ticks[0]) * 1000.0 / CLOCKS_PER_SEC; //Função de tempo
    printf("Tempo gasto: %g ms.", Tempo); //Função de tempo

    return 0;
}

```

Figura 10 – ShellSort

O algoritmo de ordenação QuickSort entre todos os 4 algoritmos supracitados é o que possui uma implementação mais complexa, ele constitui-se de três subprogramas onde em seu interior chama cada um desses subprogramas varias vezes até que os elementos do arquivo se encontrem ordenados. Essa característica de um subprograma chamar outro subprograma é denominada de recursividade. Assim como o ShellSort o algoritmo de ordenação QuickSort é classificado como um algoritmo sofisticado, porém diferente do método Concha a função de complexidade assintótica do algoritmo em seu melhor caso em relação a comparação é evidentemente  $O(n \log n)$ , essa situação ocorre quando cada partição divide o arquivo em duas partes. Já em seu pior caso o algoritmo chega a ser de complexidade  $O(n^2)$  (quadrática), esse pior caso ocorre quando o pivô é escolhido sempre como o maior ou menor elemento, porém esse pior caso pode ser revertido empregando algumas modificações no algoritmo. O QuickSort é extremamente eficiente para ordenar arquivos com entradas de tamanhos grandes e necessita apenas de  $n \log n$  comparações em media para ordenar  $n$  itens. Sua desvantagem se dá pelo fato do algoritmo possuir uma implementação tanto quanto complexa, pois um

pequeno descuido na hora de implementar leva o algoritmo ao seu pior caso para algumas entradas de dados, além disso o Quick em seu pior caso de comparações é de complexidade assintótica  $O(n^2)$  (quadrática) e o método não é estável. A implementação do QuickSort está ilustrada abaixo:

```
void QuickS_Char(TAD_Lista_Palavras *pLista_Palavra, Itens_Vlista_Palavras *a){
    int i;
    clock_t Ticks[2]; // Função de tempo
    Ticks[0] = clock(); // Função de tempo

    // Utiliza a função ordena para ir ordenando o vetor
    Ordena(0, pLista_Palavra->contapalavra - 1, a); // Passa parâmetros iniciais
    for(i = 0; i < pLista_Palavra->contapalavra; i++){
        printf("Palavra: %s\n", a[i].PontePalavra.Vpalavra.palavra);
    }
    Ticks[1] = clock(); // Função de tempo
    double Tempo = (Ticks[1] - Ticks[0]) * 1000.0 / CLOCKS_PER_SEC; // Função de tempo
    printf("Tempo gasto: %g ms.\n", Tempo); // Função de tempo
    printf("Comparacao: %d\n", a->comparacao);
    printf("Movimentacao: %d\n", a->movimentacao);
    return 0;
}

void Ordena(int Esq, int Dir, Itens_Vlista_Palavras *a){
    int i, j, k;
    /* Cria as partições necessárias para
    a ordenação do vetor */

    Particao(Esq, Dir, &i, &j, a);
    if (Esq < j)
        Ordena(Esq, j, a);
    if (i < Dir)
        Ordena(i, Dir, a);
}
```

Figura 11 – QuickSort



```

void Particao(int Esq, int Dir, int *i, int *j, Itens_Vlista_Palavras *a){
    Itens_Vlista_Palavras aux, pivo;

    *i = Esq;
    *j = Dir;
    //Calcula o pivô a ser usado
    strcpy(pivo.PontePalavra.Vpalavra.palavra, a[((*i+*j)/2)].PontePalavra.Vpalavra.palavra);
    do{
        /*Compara o pivô com as palavras da esquerda
        até o pivô for menor que a posição i do vetor pela esquerda*/
        a->comparacao++;
        while(strcmp(pivo.PontePalavra.Vpalavra.palavra, a[*i].PontePalavra.Vpalavra.palavra) > 0){
            a->comparacao++;
            (*i)++;
        }
        a->comparacao++;
        /*Compara o pivô com as palavras da direita
        até o pivô ser maior do a posição j do vetor pela direita */
        while(strcmp(pivo.PontePalavra.Vpalavra.palavra, a[*j].PontePalavra.Vpalavra.palavra) < 0){
            a->comparacao++;
            (*j)--;
        }
        a->comparacao++;
        if(*i <= *j){
            //Faz a troca de posições das palavras na posição i e j
            strcpy(aux.PontePalavra.Vpalavra.palavra, a[*i].PontePalavra.Vpalavra.palavra);
            strcpy(a[*i].PontePalavra.Vpalavra.palavra, a[*j].PontePalavra.Vpalavra.palavra);
            strcpy(a[*j].PontePalavra.Vpalavra.palavra, aux.PontePalavra.Vpalavra.palavra);
            (*i)++;
            (*j)--;
            a->movimentacao++;
        }
    }while(*i <= *j);
}

```

Figura 12-Partição- Quicksort

Por ultimo e não menos importante o algoritmo de ordenação HeapSort por sua vez assim como o QuickSort possui uma grande complexidade em sua implementação, ele assim como os dois anteriores é considerado um algoritmo sofisticado. A sua diferença em relação aos outros dois algoritmos sofisticados é que independente de qualquer variável a sua complexidade assintótica em relação à comparação vai ser sempre  $O(n \log n)$ , porém por melhor que seja o HeapSort dependendo de algumas constantes o QuickSort na sua melhor implementação chega a ter um tempo de execução ainda menor do que o Heap. Sua vantagem se da pelo fato de que sua função de complexidade assintótica independente da entrada será  $O(n \log n)$ , já sua desvantagem se da pelo fato do anel interno no algoritmo ser bastante complexo e ele não ser um algoritmo estável. O algoritmo é recomendado para o caso de que suas aplicações não poderem encontrar de fato um pior caso e não é recomendado para arquivos com poucos registros já que ele gasta uma grande quantidade de tempo para fazer a construção do Heap. A implementação do QuickSort está ilustrada abaixo:



```

void Ordena_HeapS_Char(TAD_Lista_Palavras *pLista_Palavras, Itens_Vlista_Palavras *a){
    int Esq, Dir, n, i;
    Itens_Vlista_Palavras aux;
    clock_t Ticks[2]; //Função de tempo
    Ticks[0] = clock(); //Função de tempo
    Constroi(a, pLista_Palavras); //Chama a função de construção
    Esq = 0; //Marcador Esquerda
    Dir = pLista_Palavras->contapalavra - 1; //Marcador Direita
    //Percorre até direita for <= 0
    while(Dir > 0){
        strcpy(aux.PontePalavra.Vpalavra.palavra, a[0].PontePalavra.Vpalavra.palavra);
        strcpy(a[0].PontePalavra.Vpalavra.palavra, a[Dir].PontePalavra.Vpalavra.palavra);
        strcpy(a[Dir].PontePalavra.Vpalavra.palavra, aux.PontePalavra.Vpalavra.palavra);
        Dir--;
        Refaz(Esq, Dir, a);
        a->movimentacao++;
        a->comparacao++;
    }
    for(i = 0; i < pLista_Palavras->contapalavra; i++){
        printf("Palavra: %s\n", a[i].PontePalavra.Vpalavra.palavra);
    }
    Ticks[1] = clock(); //Função de tempo
    double Tempo = (Ticks[1] - Ticks[0]) * 1000.0 / CLOCKS_PER_SEC; //Função de tempo
    printf("Tempo gasto: %g ms.\n", Tempo); //Função de tempo
    printf("Comparacao: %d\n", a->comparacao);
    printf("Movimentacao: %d\n", a->movimentacao);
    return 0;
}

//Função de construção do heap
void Constroi(Itens_Vlista_Palavras *a, TAD_Lista_Palavras *pLista_Palavras){
    int Esq, n;

    n = pLista_Palavras->contapalavra - 1;
    Esq = n/2+1; //Calcula a esquerda(pivô)
    while(Esq > 0){
        Esq--;
        Refaz(Esq, n, a); //Chama a função refaz
    }
}

//Função para refazer o heap

```

Figura 13 – HeapSort

```

void Refaz(int Esq, int Dir, Itens_Vlista_Palavras *a){
    int j = Esq * 2;
    Itens_Vlista_Palavras aux;

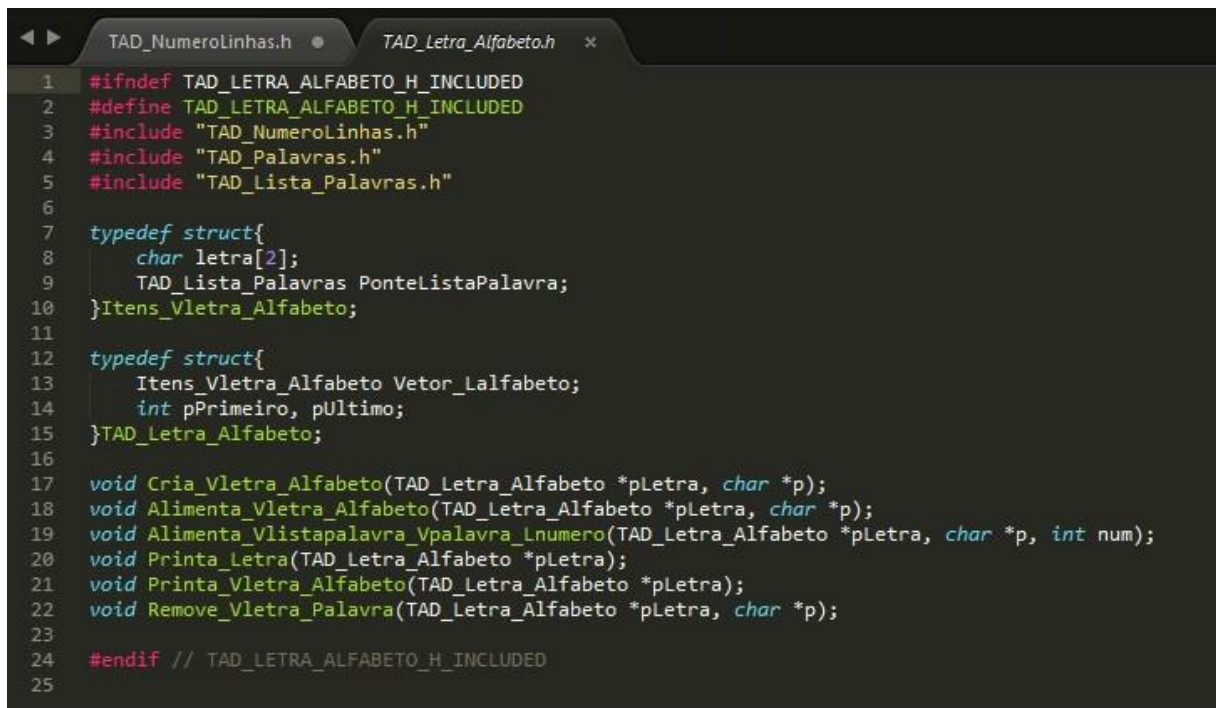
    strcpy(aux.PontePalavra.Vpalavra.palavra, a[Esq].PontePalavra.Vpalavra.palavra);
    while(j <= Dir){ //Compara o j com direita
        if((j < Dir) && (strcmp(a[j].PontePalavra.Vpalavra.palavra, a[j+1].PontePalavra.Vpalavra.palavra) < 0))
            j++;
        a->comparacao++;
        if(strcmp(a[Esq].PontePalavra.Vpalavra.palavra, a[j].PontePalavra.Vpalavra.palavra) >= 0)
            break;
        a->comparacao++;
        strcpy(a[Esq].PontePalavra.Vpalavra.palavra, a[j].PontePalavra.Vpalavra.palavra);
        Esq = j;
        j = Esq * 2;
    }
    strcpy(a[Esq].PontePalavra.Vpalavra.palavra, aux.PontePalavra.Vpalavra.palavra);
}

//Função para refazer o heap

```

Figura 14-Refaz-Heapsort

O arquivo TAD\_Letra\_Alfabeto.h também foi reutilizado do primeiro trabalho e de acordo com a ilustração abaixo pode-se observar que em seu interior contém uma variável “letra” do tipo char, uma variável “PonteListaPalavra” do tipo TAD\_Lista\_Palavras, demais variáveis e o cabeçalho das operações a serem utilizadas posteriormente.



```
1 #ifndef TAD_LETRA_ALFABETO_H_INCLUDED
2 #define TAD_LETRA_ALFABETO_H_INCLUDED
3 #include "TAD_NumeroLinhas.h"
4 #include "TAD_Palavras.h"
5 #include "TAD_Lista_Palavras.h"
6
7 typedef struct{
8     char letra[2];
9     TAD_Lista_Palavras PonteListaPalavra;
10 }Itens_Vletra_Alfabeto;
11
12 typedef struct{
13     Itens_Vletra_Alfabeto Vetor_Lalfabeto;
14     int pPrimeiro, pUltimo;
15 }TAD_Letra_Alfabeto;
16
17 void Cria_Vletra_Alfabeto(TAD_Letra_Alfabeto *pLetra, char *p);
18 void Alimenta_Vletra_Alfabeto(TAD_Letra_Alfabeto *pLetra, char *p);
19 void Alimenta_Vlistapalavra_Vpalavra_Lnumero(TAD_Letra_Alfabeto *pLetra, char *p, int num);
20 void Printa_Letra(TAD_Letra_Alfabeto *pLetra);
21 void Printa_Vletra_Alfabeto(TAD_Letra_Alfabeto *pLetra);
22 void Remove_Vletra_Palavra(TAD_Letra_Alfabeto *pLetra, char *p);
23
24 #endif // TAD_LETRA_ALFABETO_H_INCLUDED
25
```

Figura 15 - TAD\_Letra\_Alfabeto.h

Já o arquivo TAD\_Letra\_Alfabeto.c é composto pelos subprogramas e suas implementações como ilustrado a baixo:

```

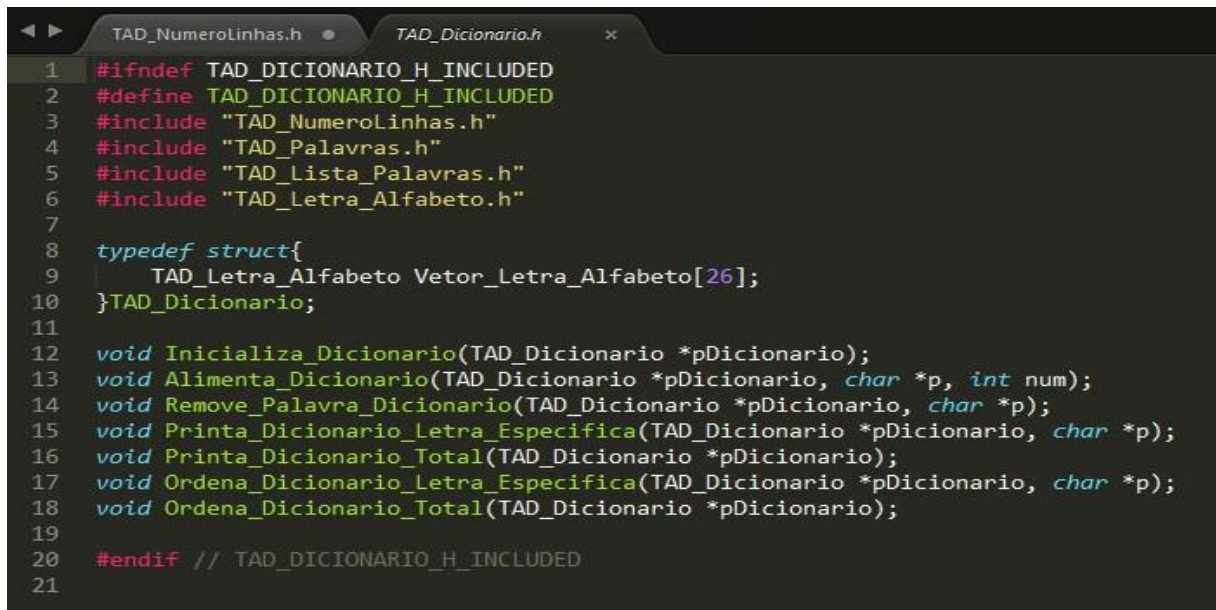
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "TAD_NumeroLinhas.h"
5  #include "TAD_Palavras.h"
6  #include "TAD_Lista_Palavras.h"
7  #include "TAD_Letra_Alfabeto.h"
8
9  /* Cria o vetor com a letra do
10 alfabeto vazia*/
11
12 void Cria_Vletra_Alfabeto(TAD_Letra_Alfabeto *pLetra, char *p){
13     pLetra->pPrimeiro = 0;
14     pLetra->pUltimo = pLetra->pPrimeiro;
15     strcpy(pLetra->Vetor_Lalfabeto.lettra, "");
16     Alimenta_Vletra_Alfabeto(pLetra, p);
17     Cria_Vlista_Palavra(&(pLetra->Vetor_Lalfabeto.PonteListaPalavra));
18 }
19 //Insere uma letra nesse vetor
20
21 void Alimenta_Vletra_Alfabeto(TAD_Letra_Alfabeto *pLetra, char *p){
22     strcpy(pLetra->Vetor_Lalfabeto.lettra, p);
23 }
24 //Insere uma palavra na lista
25
26 void Alimenta_Vlistapalavra_Vpalavra_Lnumero(TAD_Letra_Alfabeto *pLetra, char *p, int num){
27     Alimenta_Vlista_Palavra(&(pLetra->Vetor_Lalfabeto.PonteListaPalavra), p, num); // Utiliza a função para inserir a lista de palavras
28 }
29
30 //Imprime a letra do alfabeto
31
32 void Printa_Letra(TAD_Letra_Alfabeto *pLetra){
33     printf("Letra do alfabeto: %s\n", pLetra->Vetor_Lalfabeto.lettra);
34 }
35
36 //Imprime a letra e a palavra
37 void Printa_Vletra_Alfabeto(TAD_Letra_Alfabeto *pLetra){
38     Printa_Letra(pLetra); //Utiliza essa função para chamar a letra
39     Printa_Vlista_Palavra(&(pLetra->Vetor_Lalfabeto.PonteListaPalavra)); //Utiliza essa função para imprimir a palavra
40 }
41
42 //Remoção da palavra dada
43 void Remove_Vletra_Palavra(TAD_Letra_Alfabeto *pLetra, char *p){
44     Remove_Vlista_Palavra_Dada(&(pLetra->Vetor_Lalfabeto.PonteListaPalavra), p); //Utiliza a função para remover a palavra dada
45 }

```

Figura 16 - TAD\_Letra\_Alfabeto.c

O arquivo TAD\_Dicionario.h também foi reutilizado do algoritmo feito para o primeiro trabalho, porém assim como o arquivo TAD\_Lista\_Palavra.h foram feitos acréscimos de algumas operações no cabeçalho como:

- Ordena\_Dicionario\_Letra\_Especific.
- Ordena\_Dicionario\_Total.



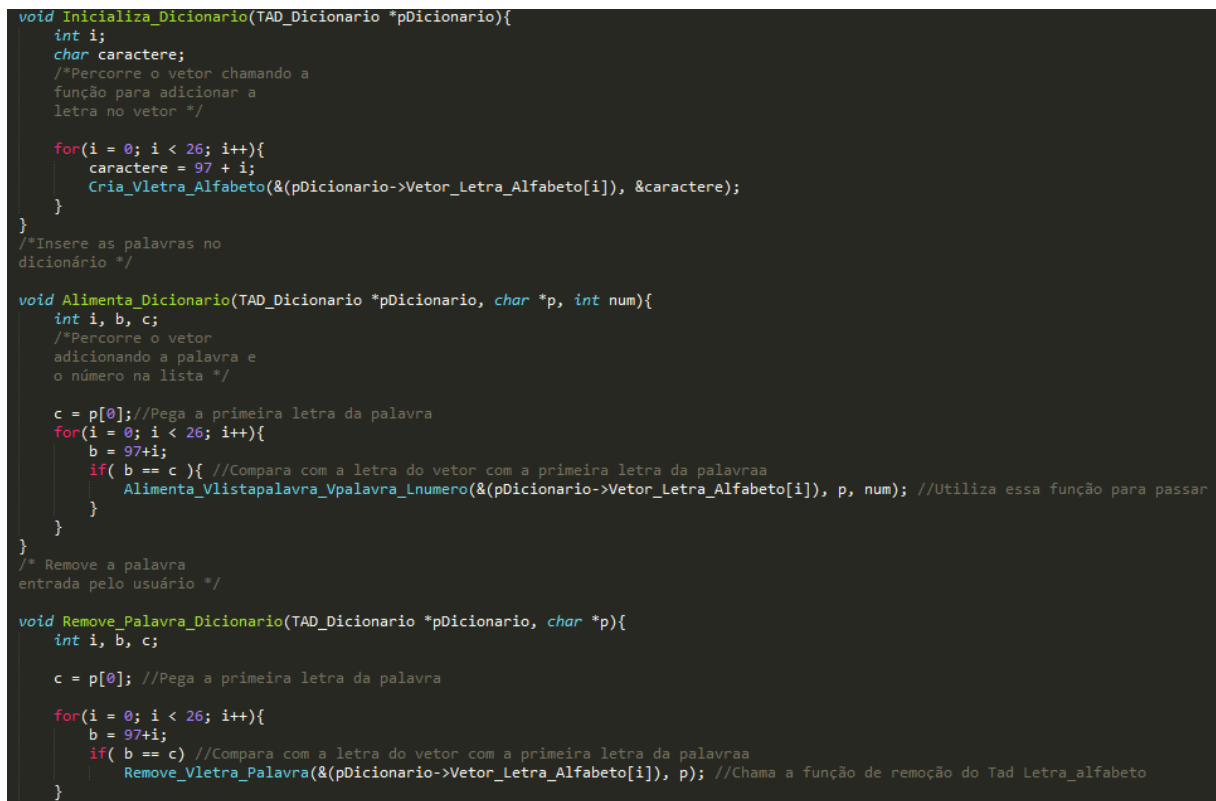
```

1  #ifndef TAD_DICIONARIO_H_INCLUDED
2  #define TAD_DICIONARIO_H_INCLUDED
3  #include "TAD_NumeroLinhas.h"
4  #include "TAD_Palavras.h"
5  #include "TAD_Lista_Palavras.h"
6  #include "TAD_Letra_Alfabeto.h"
7
8  typedef struct{
9      TAD_Letra_Alfabeto Vetor_Letra_Alfabeto[26];
10 }TAD_Dicionario;
11
12 void Inicializa_Dicionario(TAD_Dicionario *pDicionario);
13 void Alimenta_Dicionario(TAD_Dicionario *pDicionario, char *p, int num);
14 void Remove_Palavra_Dicionario(TAD_Dicionario *pDicionario, char *p);
15 void Printa_Dicionario_Letra_Especificas(TAD_Dicionario *pDicionario, char *p);
16 void Printa_Dicionario_Total(TAD_Dicionario *pDicionario);
17 void Ordena_Dicionario_Letra_Especificas(TAD_Dicionario *pDicionario, char *p);
18 void Ordena_Dicionario_Total(TAD_Dicionario *pDicionario);
19
20 #endif // TAD_DICIONARIO_H_INCLUDED
21

```

Figura 17 - TAD\_Dicionario.h

Já o arquivo TAD\_Dicionario.c é composto pelos subprogramas e suas implementações como ilustrado a baixo:



```

void Inicializa_Dicionario(TAD_Dicionario *pDicionario){
    int i;
    char caractere;
    /*Percorre o vetor chamando a
    função para adicionar a
    letra no vetor */

    for(i = 0; i < 26; i++){
        caractere = 97 + i;
        Cria_Vletra_Alfabeto(&(pDicionario->Vetor_Letra_Alfabeto[i]), &caractere);
    }
}

/*Insere as palavras no
dicionário */

void Alimenta_Dicionario(TAD_Dicionario *pDicionario, char *p, int num){
    int i, b, c;
    /*Percorre o vetor
    adicionando a palavra e
    o número na lista */

    c = p[0]; //Pega a primeira letra da palavra
    for(i = 0; i < 26; i++){
        b = 97+i;
        if( b == c ){ //Compara com a letra do vetor com a primeira letra da palavra
            Alimenta_Vlistapalavra_Vpalavra_Lnumero(&(pDicionario->Vetor_Letra_Alfabeto[i]), p, num); //Utiliza essa função para passar
        }
    }
}

/* Remove a palavra
entrada pelo usuário */

void Remove_Palavra_Dicionario(TAD_Dicionario *pDicionario, char *p){
    int i, b, c;

    c = p[0]; //Pega a primeira letra da palavra

    for(i = 0; i < 26; i++){
        b = 97+i;
        if( b == c ){ //Compara com a letra do vetor com a primeira letra da palavra
            Remove_Vletra_Palavra(&(pDicionario->Vetor_Letra_Alfabeto[i]), p); //Chama a função de remoção do TAD Letra_alfabeto
        }
    }
}

```

Figura 18 - TAD\_Dicionario.c

A operação **Ordena\_Dicionario\_Letra\_Especificas** adicionada ao TAD\_Dicionario tem a função de imprimir na tela, de acordo com a letra do dicionário escolhida, a lista de palavras

correspondente a tal letra desordenada, assim após a impressão da lista o subprograma exibe um painel de escolha onde o usuário deve digitar o número correspondente a tal opção desejada para selecionar qual tipo de algoritmo utilizar para ordenar a lista de palavra exibida. Depois de selecionado a opção o algoritmo fará uma copia da lista de palavras a ser ordenada e ordenará tal copia, imprimindo na tela o número de comparações e movimentações feitas pelo algoritmo e a lista de palavra ordenada de forma ascendente.

```
void Ordena_Dicionario_Letra_Especific(TAD_Dicionario *pDicionario, char *p){
    int i, b, c, j, opcao;

    c = p[0]; //Pega a primeira letra da palavra

    for(i = 0; i < 26; i++){
        b = 97+i;
        if(b == c){
            Printa_Vetor_Desordenado(&(pDicionario->Vetor_Letra_Alfabeto[i].Vetor_Lalfabeto.PonteListaPalavra)); //Impressão do dicionário
            printf("\n");
            Alimenta_Vetor_Copia(&(pDicionario->Vetor_Letra_Alfabeto[i].Vetor_Lalfabeto.PonteListaPalavra)); //Alimenta a cópia do vetor
            printf("Selecione o algoritmo de ordenacao:\n");
            printf("Digite 1 para: BubbleSort\n");
            printf("Digite 2 para: SelectSort\n");
            printf("Digite 3 para: InsertSort\n");
            printf("Digite 4 para: ShellSort\n");
            printf("Digite 5 para: QuickSort\n");
            printf("Digite 6 para: HeapSort\n");
            printf("Digite 7 para: Sair\n");
            scanf("%d",&opcao);
            do{
                switch(opcao)
                {
                    case 1:
                        //Seleciona a função de bubble sort
                        printf("\nBubbleSort\n");
                        //Chama a função de ordenação Bubble Sort
                        Ordena_BoobleS_Char(&(pDicionario->Vetor_Letra_Alfabeto[i].Vetor_Lalfabeto.PonteListaPalavra), &(pDicionario->Vetor_Le
                        printf("\nOpcao\n");
                        scanf("%d",&opcao);
                        break;

                    case 2:
                        //Seleciona a função de Select sort
                        printf("\nSelectSort\n");
                        //Chama a função de ordenação Select Sort
                        Ordena_SelectS_Char(&(pDicionario->Vetor_Letra_Alfabeto[i].Vetor_Lalfabeto.PonteListaPalavra), &(pDicionario->Vetor_Le
                        printf("\nOpcao\n");
                        scanf("%d",&opcao);
                        break;
                }
            } while(opcao != 7);
        }
    }
}
```

Figura 19 - Ordena\_Dicionario\_Letra\_Especific

A operação **Ordena\_Dicionario\_Total** adicionada ao TAD\_Dicionario tem a função de imprimir na tela, toda lista de palavras correspondente as letras do alfabeto desordenada, assim após a impressão da lista o subprograma exibe um painel de escolha onde o usuário deve digitar o número correspondente a tal opção desejada para selecionar qual tipo de algoritmo utilizar para ordenar toda lista de palavra exibida. Depois de selecionado a opção o algoritmo fará uma copia de cada lista de palavras a ser ordenada e ordenará tal copia, imprimindo na tela o número de comparações e movimentações feitas pelo algoritmo e toda lista de palavra ordenada de forma ascendente.

```

void Ordena_Dicionario_Total(TAD_Dicionario *pDicionario){
    int i, j, b, c, opcao;
    clock_t Ticks[2]; //função de tempo
    Ticks[0] = clock(); //função de tempop

    for(i = 0; i < 26; i++){
        c = 97 + i;
        printf("Vetor letra: %c\n", c);
        Printa_Vetor_Desordenado(&(pDicionario->Vetor_Letra_Alfabeto[i].Vetor_Lalfabeto.PonteListaPalavra)); //Imprime vetor desordenado
        printf("\n");
    }

    printf("Selecione o algoritmo de ordenação:\n");
    printf("Digite 1 para: BubbleSort\n");
    printf("Digite 2 para: SelectSort\n");
    printf("Digite 3 para: InsertSort\n");
    printf("Digite 4 para: ShellSort\n");
    printf("Digite 5 para: QuickSort\n");
    printf("Digite 6 para: HeapSort\n");
    // printf("Digite 7 para: Sair\n");
    scanf("%d", &opcao);
    switch(opcao)
    {
        case 1:
            //Seleciona a função de bubble sort
            printf("\nBubbleSort\n");
            for(j = 0; j < 26; j++){
                b = 97 + j;
                printf("Vetor letra: %c\n", b);
                //Inicializa comparacao e movimentacao com zero
                Inicia_comparacao_movimentacao(&(pDicionario->Vetor_Letra_Alfabeto[j].Vetor_Lalfabeto.PonteListaPalavra));
                //Alimenta a copia do vetor
                Alimenta_Vetor_Copia(&(pDicionario->Vetor_Letra_Alfabeto[j].Vetor_Lalfabeto.PonteListaPalavra));
                //Chama a função para utilizar o bubble sort
                Ordena_BoobleS_Char(&(pDicionario->Vetor_Letra_Alfabeto[j].Vetor_Lalfabeto.PonteListaPalavra), &(pDicionario->Veto
                printf("\n");
            }
            break;
        case 2:
            //Seleciona a função de Select sort
            printf("\nSelectSort\n");
            for(j = 0; j < 26; j++){
                b = 97 + j;
                printf("Vetor letra: %c\n", b);
                //Inicializa comparacao e movimentacao com zero

```

Figura 20 - Ordena\_Dicionario\_Total

## TESTE

Logo após a implementação e execução do algoritmo, dois arquivos (.txt), foram usados como entrada do algoritmo e o algoritmo testados em duas máquinas diferentes.

As configurações das máquinas se encontram a seguir:

### Máquina1

```

C:\> Prompt de Comando
Microsoft Windows [versão 10.0.14393]
(c) 2016 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Samuel>systeminfo

Nome do host:                DESKTOP-NN8N5B8
Nome do sistema operacional:  Microsoft Windows 10 Pro
Versão do sistema operacional: 10.0.14393 N/A compilação 14393
Fabricante do sistema operacional: Microsoft Corporation
Configuração do SO:          Estação de trabalho autônoma
Tipo de compilação do sistema operacional: Multiprocessor Free
Proprietário registrado:     Samuel
Organização registrada:
Identificação do produto:     00331-10000-00001-AA602
Data da instalação original:   02/09/2016, 04:12:48
Tempo de Inicialização do Sistema: 17/11/2017, 13:27:23
Fabricante do sistema:        ASUS
Modelo do sistema:            All Series
Tipo de sistema:              x64-based PC
Processador(es):              1 processador(es) instalado(s).
                                [01]: Intel64 Family 6 Model 60 Stepping 3 GenuineIntel ~3700 Mhz
Versão do BIOS:               American Megatrends Inc. 2001, 16/06/2014
Pasta do Windows:             C:\WINDOWS
Pasta do sistema:              C:\WINDOWS\system32
Inicializar dispositivo:      \Device\HarddiskVolume2
Localidade do sistema:        pt-br;Português (Brasil)
Localidade de entrada:        pt-br;Português (Brasil)
Fuso horário:                 (UTC-03:00) Brasília
Memória física total:          8.130 MB
Memória física disponível:     3.545 MB
Memória Virtual: Tamanho Máximo: 9.410 MB
Memória Virtual: Disponível:   2.794 MB
Memória Virtual: Em Uso:       6.616 MB
Local(is) de arquivo de paginação: C:\pagefile.sys
Domínio:                       WORKGROUP
Servidor de Logon:             \\DESKTOP-NN8N5B8
Hotfix(es):                    15 hotfix(es) instalado(s).
                                [01]: KB3176935
                                [02]: KB3176936
                                [03]: KB3176937
                                [04]: KB3186568
                                [05]: KB3199209
                                [06]: KB3199986
                                [07]: KB3211320
                                [08]: KB4013418
                                [09]: KB4023834
                                [10]: KB4033637
                                [11]: KB4035631
                                [12]: KB4048951
                                [13]: KB4049065
                                [14]: KB4051613
                                [15]: KB4048953

```

Figura 21-Configurações máquina 1



## Máquina 2

```
Tipo de compilação do sistema operacional: Multiprocessor Free
Proprietário registrado: vinicius
Organização registrada: Microsoft
Identificação do produto: 00327-30377-29544-AAOEM
Data da instalação original: 06/06/2017, 15:16:36
Tempo de Inicialização do Sistema: 05/12/2017, 20:33:44
Fabricante do sistema: Positivo Informatica SA
Modelo do sistema: H14BT58
Tipo de sistema: x64-based PC
Processador(es): 1 processador(es) instalado(s).
[01]: Intel64 Family 6 Model 55 Stepping 8 GenuineIntel ~1578 Mhz
Versão do BIOS: American Megatrends Inc. 1.07.U, 15/05/2015
Pasta do Windows: C:\WINDOWS
Pasta do sistema: C:\WINDOWS\system32
Inicializar dispositivo: \Device\HarddiskVolume1
Localidade do sistema: pt-br;Português (Brasil)
Localidade de entrada: pt-br;Português (Brasil)
Fuso horário: (UTC-03:00) Brasília
Memória física total: 1.937 MB
Memória física disponível: 424 MB
Memória Virtual: Tamanho Máximo: 5.137 MB
Memória Virtual: Disponível: 1.789 MB
Memória Virtual: Em Uso: 3.348 MB
Local(is) de arquivo de paginação: C:\pagefile.sys
```

Figura 22- configurações máquina2

Os testes feitos nas duas máquinas obtiveram diferentes resultados em milissegundos, para cada um dos algoritmos testados.

Os gráficos com os tempos registrados pelas duas máquinas com os algoritmos podem ser analisados abaixo.



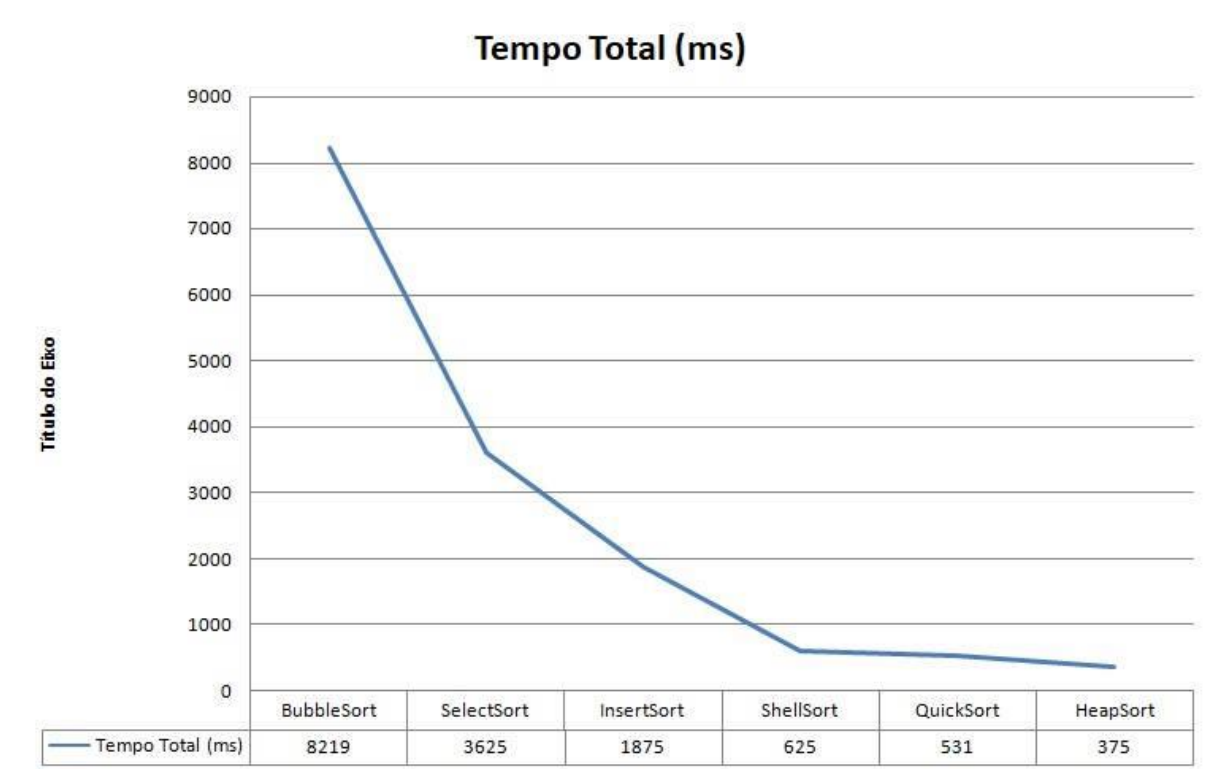


Figura 23-Gráfico Máquina1

Os tempos de execução da máquina 1 foram os seguintes:

| Algoritmos de Ordenação | Tempo Total (ms) |
|-------------------------|------------------|
| BubbleSort              | 8219             |
| SelectSort              | 3625             |
| InsertSort              | 1875             |
| ShellSort               | 625              |
| QuickSort               | 531              |
| HeapSort                | 375              |

Figura 24-Tabela Máquina 1

Gráfico Máquina 2

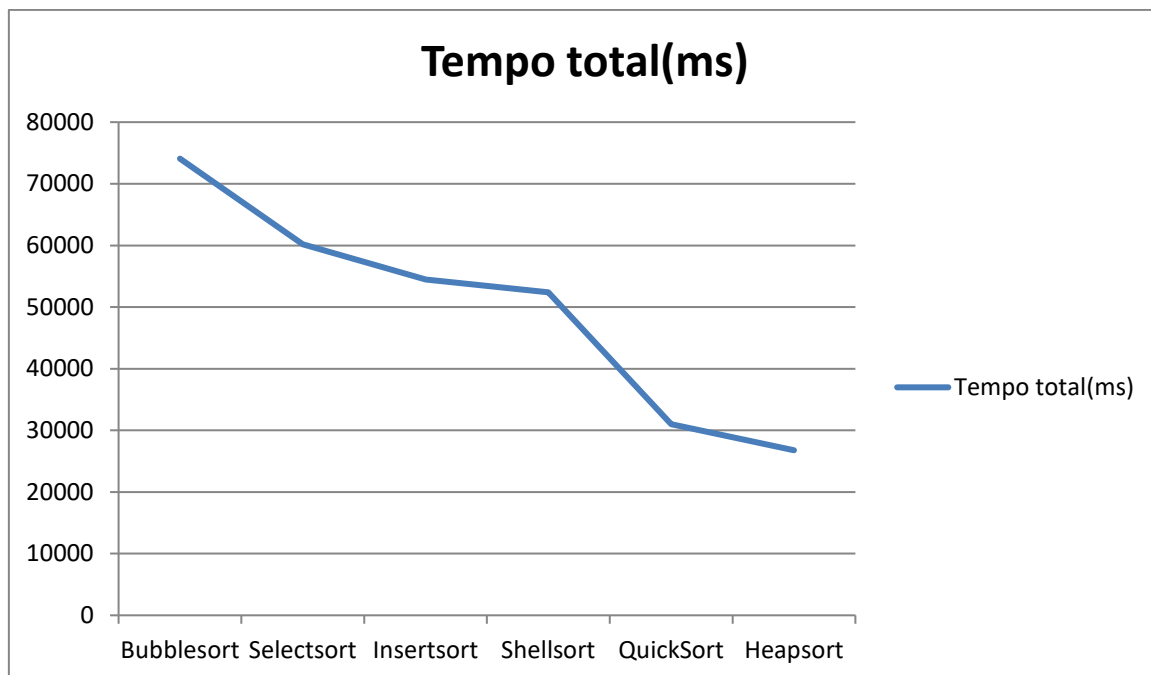


Figura 25-Gráfico Máquina2

O tempo de execução da máquina 2 foram os seguintes:

| Algoritmo  | Tempo |
|------------|-------|
| Bubblesort | 74081 |
| Selectsort | 60205 |
| Insertsort | 54488 |
| Shellsort  | 52391 |
| Quicksort  | 31006 |
| Heapsort   | 26774 |

Figura 26-Tabela Máquina 2

Os resultados mostram a diferença de tempo que cada algoritmo de ordenação leva para cumprir o seu papel de ordenar todas as palavras. Nota-se que os algoritmos quick-sort e

heap-sort conseguem ordenar as palavras em um tempo menor que os outros, mostrando a eficiência que esses algoritmos possuem, enquanto o bubblesort é o algoritmo com maior tempo de execução, mostrando sua ineficiência se comparado aos outros algoritmos. É importante ressaltar que os testes apresentados aqui, se tratam de um arquivo com cerca de palavras para a melhor visualização da diferença de métodos de ordenação.

A grande diferença entre o tempo gasto pelo os algoritmos em cada uma das máquinas se deve a diferenças técnicas entre os computadores.

## **CONCLUSÃO**

Durante todo o período, estudamos os tipos abstratos de dados, maneiras de implementá-los, o custo de um algoritmo e por último e não menos importante, os algoritmos de ordenação.

O trabalho final da disciplina, que teve como objetivo uma atualização do dicionário proposto no primeiro trabalho, uniu todos os conhecimentos adquiridos durante o estudo da matéria, conhecimentos esses que podemos destacar, principalmente, a utilização de tipos abstratos de dados e dos algoritmos de ordenação.

Podemos observar ao implementar esses algoritmos, a diferença de movimentações, comparações e o tempo gasto para executar cada ordenação, tempo esse que em algoritmos em que o tempo é algo de extrema importância, se torna algo decisivo para o sucesso do algoritmo.

Por fim, o trabalho nos mostrou a importância de todos os algoritmos de ordenação, suas vantagens e desvantagens, bem como a influência que estes podem ter sobre a eficiência de um algoritmo.

