

UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
CURSO DE GRADUAÇÃO EM MATEMÁTICA APLICADA E COMPUTACIONAL

Lucas Amaral Taylor

Relatório
MAC0122 — Princípios de Desenvolvimento de Algoritmos
Projeto Final — Seu primeiro editor de texto

São Paulo-SP
2022

LISTA DE FIGURAS

Figura 1: laço de comando.....	3
Figura 2: exemplo do processo de extração de string-parâmetro.....	5
Figura 3: funções posicao_absoluta() e posicao_relativa().....	8
Figura 4: função descobre_linha().....	9
Figura 5: funções contagem_linhas() e contagem_caracteres().....	9
Figura 6: função de contagem_de_caracteres_linha().....	10
Figura 7: comando de exibição.....	10
Figura 8: funções de referência.....	11
Figura 9: exibição de texto com a função exhibe_linha().....	12
Figura 10: exibição de texto sem a função exhibe_linha().....	12
Figura 11: função exhibe_linha().....	13
Figura 12: função insere().....	15
Figura 13: função desempilha().....	16
Figura 14: função apaga().....	17
Figura 15: função busca().....	18
Figura 16: comando [N].....	22

SUMÁRIO

1. INTRODUÇÃO	1
2 VARIÁVEIS FUNDAMENTAIS E LÓGICAS QUE PERMEIAM TODO PROGRAMA	2
2.1 VARIÁVEIS FUNDAMENTAIS E LAÇO DE COMANDO	2
2.1.1 Variáveis fundamentais	2
2.1.2 Laço de comando	2
2.2. OBSERVAÇÕES RELEVANTES	4
2.2.1 A exclusividade de execução dos comandos que recebem um parâmetro	4
2.2.2 Alocação dinâmica dos comandos com parâmetros	4
2.2.3. Atualização de linhas e número de caracteres	5
2.2.4. Posição absoluta e posição relativa	5
3. ANÁLISE DE FUNÇÕES	7
3.1 CLASSIFICAÇÃO	7
3.2 FUNÇÕES DE POSICIONAMENTO	7
3.3 FUNÇÕES DE CONTAGEM	9
3.4 FUNÇÕES DE EXIBIÇÃO E REFERÊNCIA	10
3.5 FUNÇÕES DE MANIPULAÇÃO	14
4. ANÁLISE DE COMANDOS	19
4.1 O COMANDO A	19
4.2. COMANDOS DE MOVIMENTAÇÃO	20
4.3. EXCLUSÃO DE CARACTERES	21
.4.4. INSERÇÃO DE CARACTERES	21
4.6. COMANDOS DE PILHA	22
4.6. SALVANDO O TEXTO	24
4.7 COMANDOS DE BUSCA	24
5.DISSCUSSÃO SOBRE UM EDITOR DE TEXTO INTERATIVO	26
6 CONCLUSÃO	27

1. INTRODUÇÃO

O relatório em questão visa documentar o desenvolvimento e elaboração do editor de texto criado, descrevendo e justificando todas as decisões tomadas, relatando a interface de programação, analisando a complexidade de tempo e espaço de códigos e listando alguns casos patológicos apresentados na aplicação. Além disso, uma breve discussão sobre a interatividade do editor de texto.

O presente relatório está dividido em seis partes: a introdução, que visa definir os objetivos e apresentar os tópicos abordados, a análise de funções e a análise de comandos, que discutem o desenvolvimento do código, um breve texto sobre a interatividade do editor e, por fim, a conclusão do trabalho realizado.

2 VARIÁVEIS FUNDAMENTAIS E LÓGICAS QUE PERMEIAM TODO PROGRAMA

2.1 VARIÁVEIS FUNDAMENTAIS E LAÇO DE COMANDO

2.1.1 Variáveis fundamentais

Para o funcionamento do programa, há treze variáveis que são constantemente utilizadas durante a execução da aplicação. Elas podem ser divididas em cinco categorias: *string* de comando, *string* de manipulação, inteiros de referência e posição, indicadores e variáveis de pilha.

1. **String comando:** é responsável pela leitura de comando solicitados pelo usuário;
2. **String de manipulação:** é o **char* texto**, a string que guarda o texto extraído do arquivo .txt;
3. **Inteiros de referência:** são inteiros que administram a posição do usuário no programa
 - a. **pos:** referente e posição relativa do usuário;
 - b. **linha_atual:** referente ao número de linha que o usuário está;
 - c. **numero_de_caracteres:** número de caracteres totais no texto;
 - d. **numero_de_linhas:** número de linhas totais do arquivo de texto;
 - e. **marca_pos** e o **marca_linha**, responsáveis pela marcação do comando [M];
4. **Indicadores:** são inteiros que funcionam como booleanos.
 - a. **sem_texto:** indica se o texto foi ou não extraído pelo comando [A];
 - b. **ind:** indica que o programa deve ser finalizado ou não.
5. **Variáveis de pilha:**
 - a. **int t** refere-se ao tamanho da pilha;
 - b. **char pilha:** pilha.
 - c. **int desempilha:** tem a função de indicar se o conteúdo da pilha foi ou não desempilhado;

2.1.2 Laço de comando

A extração dos caracteres do loop de comando sofreu duas modificações durante o desenvolvimento do editor de texto. A primeira versão do editor de texto utilizou a função **gets()** na extração dos comandos inseridos pelo usuário. Apesar de realizar bem e não

apresentar problemas na extração em toda execução, o terminal avisava ser uma função perigosa e necessitava ser evitada. Pesquisei a respeito cheguei a resposta de que a função **gets()** pode causar estouro na memória. Logo, a fim de evitar este problema e as mensagens irritantes do terminal, resolvi não utilizá-la. O uso da função **scanf()** também foi descartado, já que teria problemas com o ' ' (espaço). Por fim, a função escolhida ao final foi a **fgets()** que realiza a tarefa sem causar problemas oriundos das outras duas apresentadas.

Decorrendo a respeito do funcionamento da do comando, ele é responsável por captar os comandos inseridos pelo usuário e definir qual comando será executado. Vale dizer que se trata de um loop infinito controlado pelo **int ind** quando **ind = 0** o editor de texto funciona e quando **ind = 1** o editor é finalizado. Por último, cada comando é executado a partir de uma série de *if/else if*, cada um relacionado com um, desta maneira:

Figura 1: laço de comando

```
if (comando[0] == 'A') ...
if (comando[i] == '!') ...
if (sem_texto == 0) ...
else
{
    if (comando[0] == 'E') ...
    else if (comando[0] == 'I') ...
    else if (comando[i] == 'T' || comando[i] == 'F' || comando[i] == 'O' ||
    else if (comando[i] == 'P' || comando[i] == 'Q') ...
    else if (comando[0] == ':') ...
    else if (comando[i] == 'D') ...
    else if (comando[i] == 'M') ...
    else if (comando[i] == 'V') ...
    else if (comando[i] == 'C' || comando[i] == 'X') ...
    else if (comando[i] == 'U') ...
    else if (comando[i] == 'J' || comando[i] == 'H') ...
```

Fonte: captura de tela do código escrito pelo autor

Apesar de ocupar muitas linhas, acredito que seja a organização mais eficiente, já que facilita futuras manutenções de código. Isso porque, sabendo de um problema em algum comando, o profissional que realizará a manutenção terá a noção de onde analisá-lo.

Dentre todos os comandos, achei conveniente analisar dois separadamente por terem tarefas muito simples e relacionadas mais ao laço de comando do que a manipulação do texto em si. O comando **[!]**, responsável por liberar a memória do texto (caso, este foi inserido) e trocar o valor do **ind** de zero para um, finalizando o programa.

Há um comando, que não recebe apenas uma variável, mas várias: todas as letras minúsculas e as maiúsculas G, K, L, W e Y, todas essas letras não possuem comando definido,

logo, caso o usuário digitar alguma delas esperando algum comando, o programa exibirá uma mensagem indicando que o comando é inválido.

2.2. OBSERVAÇÕES RELEVANTES

Antes de comentar as funções e os comandos do programa, é importante realizar algumas observações que são características que permeiam mais de um comando.

2.2.1 A exclusividade de execução dos comandos que recebem um parâmetro

Há comandos que recebem palavras ou nome de arquivo como parâmetros, são eles: os comandos [I], [A], [E], [B] e [S]. A fim de evitar conflito de chamada de outros comandos acabei optando que as chamadas desses comandos deverias ser feita de maneira única, isto é, quando um desses comandos for chamado a letra que o chama deve estar na primeira posição e a *string* em seguida até o final do texto.

Para exemplificar o conflito a ser evitado, imagine abrir um arquivo com o nome F.txt, utilizando o comando [A]. Para o programa não executar o comando F, a limitação acima foi realizada. O programa entende que o usuário, quando solicita o comando A, utiliza **apenas** o comando A, isto é, ele imagina que ao digitar o comando, o primeiro caractere seja o 'A' e em seguida, venha o nome do arquivo até a tecla ENTER. Exemplo:

* * * EDITOR DE TEXTO * * *

>Atexto.txt <ENTER>

O programa entende que o usuário quer um abrir o arquivo, pelo primeiro caractere ser 'A', e que o nome do arquivo é: texto.txt. Mema lógica aos demais comandos, claro, cada qual com sua particularidade. Por fim, um comando que não recebe apenas strings como parâmetro, mas sim inteiros é o comando ':' que com exceção do F, segue um inteiro como parâmetro, logo a mesma exclusividade foi aplicada a ele. Caso analisado posteriormente na sessão "Comandos de movimentação".

2.2.2 Alocação dinâmica dos comandos com parâmetros

Os comandos que recebem um parâmetro, com exceção do comando 'S' (justificativa apresentada na explicação do comando em questão), utilizam-se da alocação dinâmica para armazenar a *string-parâmetro*. De forma geral, isso ocorre com o seguinte processo: primeiramente, o programa extrai o parâmetro contido no comando, explicado anteriormente

em “A exclusividade de execução dos comandos que recebem um parâmetro”. Posteriormente, é criado um laço a fim de contar a quantidade de caracteres que o parâmetro possui e a função **malloc()** é empregada, para gerenciar o espaço de memória necessário, em um **char***. Por fim, cria-se um laço para inserir os respectivos caracteres neste **char**. Segue um exemplo deste processo:

Figura 2: exemplo do processo de extração e manipulação do comando com string-parâmetro

```
// Leitura do nome do arquivo
char *nome_arq;
int tam_nome = 0;
for (int j = 1; j < strlen(comando); j++)
    tam_nome++;

nome_arq = (char *)malloc(tam_nome * sizeof(char));
for (int j = 1; j < strlen(comando) - 1; j++)
    nome_arq[j - 1] = comando[j];
```

Fonte: captura de tela do código escrito pelo autor

Apesar de executar dois laços independentes, ambos relativos ao tamanho n do comando, conferindo uma complexidade de tempo e espaço $O(n)$ ao trecho do código, como aprendido em aula, o uso da alocação dinâmica confere maior eficiência no uso do espaço. Isso porque a alocação dinâmica separa o espaço necessário na memória e evita os problemas causados pelo uso de uma string de tamanho estático, como: ter um tamanho muito grande e gastar espaço de memória desnecessário ou ter um tamanho pequeno e não ter memória suficiente.

2.2.3. Atualização de linhas e número de caracteres

Os comandos [I], [D], [V], [X], [S], [N] e [U] alteram diretamente o texto, seja excluindo ou adicionando caracteres e linhas. Logo, para manter a posição do cursor atualizada e o pleno funcionamento da aplicação, achei válido a criação de contadores que atualizam a quantidade de caracteres e linhas presentes no texto, ou seja, as variáveis **numero_de_caracteres** e **numero_de_linhas**. As funções **contagem_caracteres()** e **contagem_linhas()**, realizam esta tarefa, a primeira conta quantos caracteres há no texto, enquanto a segunda conta quantos caracteres ‘\n’ há no texto, ou seja, quantas linhas há. Ambas são analisadas detalhadamente na sessão “funções de contagem”.

2.2.4. Posição absoluta e posição relativa

O texto trabalha com uma string contendo todo o texto — detalhamento na explicação do comando A — diante dessa forma de organização, há dois tipos de posição para um caractere: a posição absoluta e a posição relativa. A primeira refere-se a posição que o caractere está no texto todo, enquanto a segunda refere-se a posição que o caractere está na linha. Exemplo: imagine a string “apenas um\nexemplo\0”. A posição absoluta de ‘x’ é 11, enquanto a relativa de ‘x’ é 1. Alguns comandos utilizam-se desse conceito e utilizam funções de conversão de absoluta para relativa e de relativa para absoluta. As funções **posicao_absoluta()**, **posicao_relativa()** são responsáveis por essa tarefa. Explicadas na sessão “funções de posicionamento”.

3. ANÁLISE DE FUNÇÕES

3.1 CLASSIFICAÇÃO

O editor de texto possui 16 funções. Para gerar elegância, organização e maior compreensão do código, elas estão separadas em um arquivo de extensão .h e estão classificadas em 5 grupos. São eles:

1. **Funções de posicionamento:** responsáveis por administrar a posição do cursor, relativa e absoluta;
2. **Funções de contagem:** incumbidas de contar o número de caracteres e linhas no texto e de caracteres em uma linha;
3. **Funções de exibição e referência:** encarregadas de criar uma interface de interação entre o usuário e o programa e exibe informação relevantes de posição;
4. **Funções de mensagem:** são textos aparecidos frequentemente ao usuário
5. **Funções de manipulação:** Funções que alteram diretamente o texto ou trabalham diretamente com ele.

Todas as funções serão analisadas, salvo as funções de mensagem, já que elas se restringem apenas a um **printf()**.

3.2 FUNÇÕES DE POSICIONAMENTO

Assim como explicado anteriormente, um caractere possui duas posições, a posição relativa e a absoluta, a primeira referente a posição do caractere na linha e a segunda a posição no texto. Para isso foram criadas três funções, sendo duas principais: **posicao_absoluta()** e **posicao_relativa()**.

A função **posicao_absoluta()** é responsável pela conversão da posição relativa para a posição absoluta. Ela recebe como parâmetros o texto, a linha que o caractere selecionado está (geralmente, a variável **linha_atual**) e a posição relativa do caractere. Para seu funcionamento é necessário a existência de um texto, isto é, o comando [A] a precede e protótipo da função é expresso da seguinte maneira: **int posicao_absoluta(char *texto, int linha, int posicao_relativa)**

Já a função **posicao_relativa()** faz a tarefa inversa, ela converte a posição absoluta em posição relativa. Recebe como parâmetros o texto, a linha destino (geralmente, também é a **linha_atual**) e a posição absoluta do caractere. Assim como a função anterior, o comando [A]

precede seu funcionamento. O protótipo da função é: **int posicao_relativa(char *texto, int linha, int posicao_absoluta)**

O funcionamento das duas funções é bem semelhante, ambas trabalham com um laço e partem do início do texto até a posição que o cursor está posicionado. Código abaixo:

Figura 3: funções posicao_absoluta() e posicao_relativa()

```
int posicao_absoluta(char *texto, int linha, int posicao_relativa)
{
    int j = 0, k = 0;
    for (int i = 0; j <= linha; i++)
    {
        if (j == linha)
        {
            if (k == posicao_relativa)
                return i;
            k++;
        }
        if (texto[i] == '\n')
            j++;
    }
}

int posicao_relativa(char *texto, int linha, int posicao_absoluta)
{
    int j = 0, k = 0;
    for (int i = 0; j <= linha; i++)
    {
        if (j == linha)
        {
            k++;
            if (i == posicao_absoluta)
                return k;
        }
        if (texto[i] == '\n')
            j++;
    }
}
```

Fonte: captura de tela do código escrito pelo autor

Promovendo uma análise teórica de complexidade de tempo das funções acima, pode-se afirmar que se trata de uma função que depende do tamanho n do texto, o que a torna uma função de complexidade $O(n)$. Durante o desenvolvimento do programa as funções não apresentaram nenhuma patologia.

Por último, outra função de posição, menos importante que as demais, é a função **descobre_linha()** cuja tarefa é descobrir a linha que linha pertence o caractere com posição absoluta x . Ela foi utilizada para o desenvolvimento do comando [B] que será explicado na sessão “comandos de busca”. Ela recebe como parâmetro o texto e a posição absoluta x e trabalha com um laço para descobrir a linha. Sua complexidade de tempo é também relativa ao tamanho do texto, portanto, é $O(n)$ e não apresentou patologias em sua aplicação.

Figura 4: função `descobre_linha()`

```
int descobre_linha(char *texto, int x)
{
    int linha = 0;
    for (int i = 0; i < strlen(texto); i++)
    {
        if (i == x)
            return linha;
        if (texto[i] == '\n')
            linha++;
    }
};
```

Fonte: captura de tela do código escrito pelo autor

3.3 FUNÇÕES DE CONTAGEM

Existem três funções de contagem, duas delas são relacionadas ao texto por completo e uma é restrita apenas a uma linha. Como dito anteriormente, as funções de **`contagem_linhas()`** e **`contagem_caracteres()`** são responsáveis por atualizar as variáveis **`numero_de_caracteres`** e **`numero_de_linhas`**, respectivamente, depois de uma manipulação direta no texto, seja uma exclusão ou uma inserção.

As duas apenas necessitam do texto como parâmetro, ou seja, são dependentes da extração do comando [A] e possui o funcionamento bem-parecido: por um laço que começa no primeiro caractere do texto até o último, a **`contagem_caracteres()`** quanta quantos caracteres existem no texto, enquanto a **`contagem_linhas()`** conta quantos ‘\n’, ou seja, quantas linhas existem no texto. Como ambas dependem exclusivamente do tamanho n do texto, sua complexidade de tempo é $O(n)$. Código abaixo:

Figura 5: funções `contagem_linhas()` e `contagem_caracteres()`

```
int contagem_linhas(char *texto)
{
    int numero_de_linhas = 0;
    for (int i = 0; i < strlen(texto); i++)
    {
        if (texto[i] == '\n')
            numero_de_linhas++;
    }
    return numero_de_linhas;
}

int contagem_caracteres(char *texto)
{
    int numero_caracteres = 0;
    for (int i = 0; i < strlen(texto); i++)
        numero_caracteres++;
    return numero_caracteres;
}
```

Fonte: captura de tela do código escrito pelo autor

Além das duas, há outra chamada: **contagem_de_caracteres_linha()**, esta é responsável por contar quantos caracteres há em uma linha. No desenvolvimento do código do programa, ela foi utilizada para identificar o final da linha, como, por exemplo, a função foi utilizada como restrição da movimentação do caractere dos comandos [T] e diretamente na construção do comando [\$]. Assim como as funções apresentadas antes, trata-se de um laço que percorre o texto e uma variável de contagem para quando estiver na linha e ao final subtrai 1. Código abaixo:

Figura 6: função de `contagem_de_caracteres_linha()`

```
int contagem_caracteres_linha(char *texto, int linha_atual)
{
    int j = 0, i = 0, contador = 0;
    while (1)
    {
        if (j == linha_atual)
            contador++;
        if (texto[i] == '\n')
            j++;
        if (j == linha_atual + 1)
            break;
        i++;
    }
    return contador - 1;
}
```

Fonte: captura de tela do código escrito pelo autor

3.4 FUNÇÕES DE EXIBIÇÃO E REFERÊNCIA

No desenvolvimento do programa, as funções de exibição e referência, na minha opinião, foi fundamental em transformar um editor de texto em linha de comando em algo mais agradável e próximo de uma utilização de um usuário “comum”. Ao final de cada interação, o usuário recebe dados sobre sua localização do texto e a exibição da linha em que ele se encontra. Tais informações são concedidas na chamada das funções de posição que ocorre ou quando o último caractere da *string* comando é lido, isto é, quando `comando[i] == '\0'`, ou quando a interação de um comando com parâmetro é finalizada. Para não ficar tão abstrato, segue exemplo do primeiro caso:

Figura 7: comando de exibição

```
else if (comando[i] == '\0')
{
    localizacao(texto, pos, linha_atual, numero_de_caracteres, numero_de_linhas);
    exibe_linha(texto, linha_atual, pos);
    referencia(marca_linha, marca_pos);
    break;
}
```

Fonte: captura de tela do código escrito pelo autor

Há quatro funções nesta categoria, sendo duas responsáveis pela referência posicional do cursor e as outras duas de exibição. Primeiro, temos a função **referencia()**, que possui o seguinte protótipo: **void referencia(int linha_marcada, int posicao_marcada)**. Trata-se de uma função que apresenta ao usuário a linha e colunas marcadas pelo comando [M] e indica o momento que o programa solicita a interação direta com o usuário. É uma função que recebe as variáveis **marca_linha** e **marca_pos**, já apresentadas na sessão “variáveis fundamentais”, e tem complexidade $O(1)$ já que é responsável somente por imprimi-las acompanhadas pelo “>”.

A função **localizacao()** recebe como parâmetros as variáveis **texto**, **pos** (posição relativa), **linha_atual**, **numero_de_caracteres** e **numero_de_linhas** que possui o seguinte protótipo: **void localizacao(char *texto, int pos, int linha, int numero_caracteres, int numero_linhas)**

Grosso modo, é uma função de impressão. Ela exibe ao usuário a posição relativa atual e o caractere atual. Indiretamente, tem uma complexidade de $O(n)$, isso porque ela chama a função **posicao_absoluta**, já mencionada, para exibir o caractere atual. Essa função que não foi pedida nas instruções do programa, mas foi bastante útil e frequentemente utilizada no desenvolvimento dos comandos, verificação dos funcionamentos das funções e concedida a mim, como usuário, informações relevantes que me davam uma noção da minha localização no texto. Segue o código abaixo das duas:

Figura 8: funções de referência

```
void referencia(int linha_marcada, int posicao_marcada)
{
    printf("\n%d,%d>", linha_marcada, posicao_marcada);
}

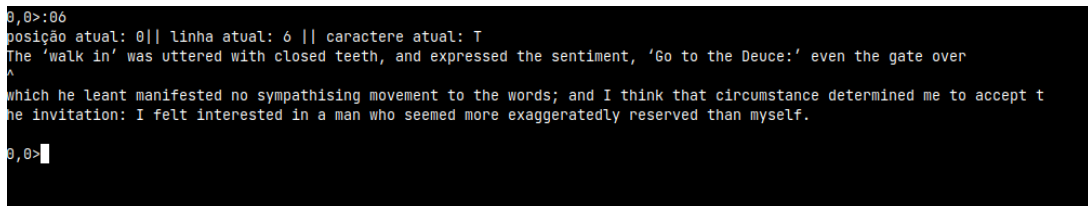
void localizacao(char *texto, int pos, int linha, int numero_caracteres, int numero_linhas)
{
    printf("posição atual: %d || linha atual: %d || caractere atual: %c\n", pos, linha, texto[posicao_absoluta(texto, linha, pos)]);
}
```

Fonte: captura de tela do código escrito pelo autor

Por fim, as outras duas são diretamente relacionadas, por uma chamar a outra, e juntas possuem uma complexidade maior, já que não são funções que só exibem informação. Inicialmente, é importante frisar a motivação da criação da função **exibe_linha()**. Durante a elaboração do projeto nos foi concedido três livros para testar nossos editores, em um deles,

as linhas possuíam mais que 300 caracteres. Isso foi um problema para mim, pois eu estava testando o meu editor com linhas no máximo 100. Em linhas com este limite, a função **posicao_cursor()** que apenas imprime ' ' (espaço) e ^ na linha abaixo dependendo do valor da variável **pos** não cumpria plenamente sua tarefa. Logo, foi necessário a criação de uma função que permitisse uma seleção correta do caractere selecionado pelo cursor, desta necessidade surgiu a função **exibe_linha()**. Para mostrar sua importância, temos duas imagens abaixo, uma com a função **exibe_linha()** e outra sem ela:

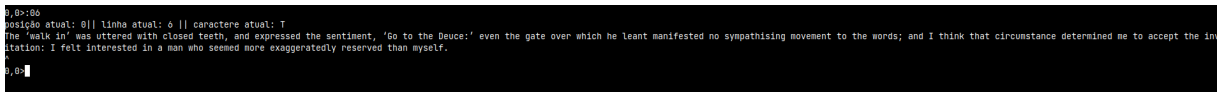
Figura 9: exibição de texto com a função **exibe_linha()**



```
0,0>:06
posição atual: 0|| linha atual: 6 || caractere atual: T
The 'walk in' was uttered with closed teeth, and expressed the sentiment, 'Go to the Deuce:' even the gate over
^
which he leant manifested no sympathising movement to the words; and I think that circumstance determined me to accept t
he invitation: I felt interested in a man who seemed more exaggeratedly reserved than myself.
0,0>
```

Fonte: captura de tela do terminal

Figura 10: exibição de texto sem a função **exibe_linha()**



```
0,0>:06
posição atual: 0|| linha atual: 6 || caractere atual: T
The 'walk in' was uttered with closed teeth, and expressed the sentiment, 'Go to the Deuce:' even the gate over which he leant manifested no sympathising movement to the words; and I think that circumstance determined me to accept the in
vitation: I felt interested in a man who seemed more exaggeratedly reserved than myself.
0,0>
```

Fonte: captura de tela do terminal

A função em questão possui duas tarefas principais: a primeira é limitar o número de caracteres exibidos agrupados em linha e a segunda é posicionar o cursor na posição certa com o auxílio da função **posicao_cursor()**. A fim de facilitar o entendimento, apresenta-se o código abaixo:

Figura 11: função `exibe_linha()`

```

void exibe_linha(char *texto, int linha_atual, int pos)
{
    int j = 0, quebra_linha = 0, aux_pos = 0, sinal = 0;
    for (int i = 0; j <= linha_atual; i++)
    {
        if (j == linha_atual)
        {
            if (pos == aux_pos)
                sinal = 1;

            aux_pos++;
            printf("%c", texto[i]);
            quebra_linha++;
            if (quebra_linha == 120)
            {
                if (sinal == 1)
                {
                    printf("\n");
                    if (pos > 119)
                        pos = pos % 120;
                    posicao_cursor(pos);
                    sinal = 0;
                }
                printf("\n");
                quebra_linha = 0;
            }
        }
        if (texto[i] == '\n')
            j++;
    }

    if (aux_pos - pos < 120)
        posicao_cursor(pos % 120);
}

```

Fonte: captura de tela do código escrito pelo autor

Inicialmente, podemos observar que a função recebe como parâmetros o **char texto**, **int linha_atual** e **int pos**, tais parâmetros são fundamentais para a localização que o usuário está no texto. O lugar do texto é encontrado através de um laço que percorre do começo do texto até o início da linha atual, como feito nas funções anteriores.

Após a linha ser encontrada, temos o uso de três variáveis auxiliares, **int sinal**, **int aux_pos** e **int quebra_linha**, elas são responsáveis por realizar a “quebra de linha”. Durante o desenvolvimento do programa foi definido que deveria ter 120 caracteres. A variável **quebra_linha** é responsável por administrar quantos caracteres foram passados para a linha, enquanto a **aux_pos** identifica se o caractere selecionado pelo usuário está neste intervalo ou não. Se caso tiver, ele altera o valor da variável **sinal** de 0 para 1. Quando o valor é igual a 1, o programa altera o valor da variável **pos** para o resto da divisão do valor **pos** e 120 (limite de caracteres) e, finalmente, chama a função **posicao_cursor**. Há o caso particular de o valor de **aux_pos - pos** ser menor que 120, o que não chamaria o sinal, para isso, apenas a chamada de **posicao_cursosr()** com o resto da divisão já basta, ver o código. Sobre a função **exibe_linha**

podemos falar que sua complexidade de tempo é $O(n)$ já que seu desempenho está incriticamente ligado ao tamanho do texto.

Como diz o nome, a função **posicao_cursor** devolve a posição marcada pelo cursor e possui apenas a variável **pos** como parâmetro. Basicamente trata-se de um laço de impressão: quando a variável do laço está com valores menores que **pos**, ela imprime ' ' (espaço) e quando está no valor de **pos** imprime ^ e finaliza o laço. A complexidade de tempo, em tese, está diretamente ligada ao valor n assumido por **pos**, portanto, $O(n)$. Uma observação sobre sua complexidade é que por depender do valor de **pos**, ela tende a ter um valor baixo, já que salvo algumas exceções, linhas ou parágrafos não possuem tantos caracteres, comparada a um texto.

3.5 FUNÇÕES DE MANIPULAÇÃO

As funções de manipulação são as funções responsáveis por manipular diretamente o texto e estão diretamente ligadas com alguns comandos. A função responsável por inserir determinada palavra do texto, possui o seguinte protótipo:

char *insere (char *texto, char *palavra, int novo_tam, int tam_palavra, int x)

Pelos parâmetros apresentados, nota-se que ela é precedida pelo comando [A], já que depende de um texto e pelo comando [I] já que depende da palavra. Além disso, depende de variáveis do tipo **int** relacionadas ao tamanho do novo texto criado (soma dos caracteres da palavra e do texto), o tamanho do texto, da palavra e as inteiro x representando a posição absoluta do caractere reselcionado.

De maneira geral, a função funciona da seguinte forma: A função aloca dinamicamente a *string* criada **texto_aux**, em seguida realiza a transcrição através de um laço de **texto** para **texto_aux** em três diferentes momentos, listados a seguir.

- I. Até o caractere no índice da posição absoluta de **pos**, a função copia o que exatamente o que está em texto.
- II. Ao chegar em na posição absoluta em questão, a posição começa a copiar os caracteres que formam a palavra.
- III. Ao finalizar o caractere volta a copiar a string texto, porém com o índice atual subtraído pelo tamanho da palavra.

Segue o código abaixo:

Figura 12: função `insere()`

```

char *insere(char *texto, char *palavra, int novo_tam, int tam_palavra, int x)
{
    // Criação de variável auxiliar que será devolvida
    char *texto_aux = (char *)malloc(novo_tam * sizeof(char));

    // Transcrição
    int l = 0;
    for (int i = 0; i - tam_palavra < novo_tam; i++)
    {
        if (i < x)
            texto_aux[i] = texto[i];
        if (i >= x)
        {
            if (l < tam_palavra - 1)
            {
                texto_aux[i] = palavra[l];
                l++;
            }
            else
                texto_aux[i] = texto[i - tam_palavra + 1];
        }
    }
    return texto_aux;
}

```

Fonte: captura de tela do código escrito pelo autor

Ao final, a função devolve a *string* **texto_aux** que será manipulada pelo comando [I] que será analisado posteriormente no relatório. É uma função que está diretamente ligada com o tamanho do texto e da palavra, para generalizar, chamemos de n , logo, tem uma complexidade de tempo e espaço $O(n)$.

Por fim, vale ressaltar que a função `insere` foi a função que mais apresentou casos patológicos na elaboração do editor de texto, creio que seja diretamente associado ao uso alocação dinâmica, mas até o fim do relatório não consegui identificar exatamente. No mais, um erro frequente que a função apresenta com a inserção de uma *string* palavra muito grande é o seguinte: **malloc(): corrupted top size**

A função **desempilha()** é outra função de manipulação e possui o seguinte protótipo:

char *desempilha(char *texto, char *pilha, int novo_tam, int t, int x)

Ela é precedida pelos comandos [A], já que necessita do texto para seu funcionamento, e também dos comandos [C] ou [X] já que depende de uma pilha cheia, –detalhes dos comandos e de suas operações na sessão “Comandos de pilha”–. Ela recebe como parâmetro, o texto, a pilha, o novo tamanho que a variável **texto_aux** receberá e a posição absoluta x atual do texto.

Além disso, ela tem um funcionamento muito semelhante à função **insere()**: cria um texto auxiliar, identifica a posição absoluta que é solicitado a inserção da *string* e por fim,

diferenciando-se da função **insere()** não apenas insere, mas também o desempilha, já que a *string* em questão está em uma estrutura de pilha. A função é dependente do tamanho n do conteúdo empilhado e o texto em questão, portanto, sua complexidade é $O(n)$. Durante a elaboração do programa, constatou-se que grandes trechos de texto, o programa apresenta falhar em desempilhar. Código abaixo:

Figura 13: função desempilha()

```
char *desempilha(char *texto, char *pilha, int novo_tam, int t, int x)
{
    // Criação de variável auxiliar que será devolvida
    char *texto_aux = (char *)malloc(novo_tam * sizeof(char));

    // Transcrição
    int l = t;
    for (int i = 0; i - t < novo_tam; i++)
    {
        if (i <= x)
            texto_aux[i] = texto[i];
        if (i > x)
        {
            if (t > 0)
            {
                texto_aux[i] = pilha[--t];
            }
            else
                texto_aux[i] = texto[i - 1];
        }
    }
    return texto_aux;
}
```

Fonte: captura de tela do código escrito pelo autor

A função **apaga()** foi retirada da sessão 3.3 do livro “Algoritmos em C” do Paulo Feofiloff, livro referência da disciplina. Ela possui o seguinte protótipo:

char *apaga(char *texto, int numero_de_caracteres, int x)

Ela depende de um texto e do número de caracteres, ou seja, é precedida pelo comando [A] e também recebe um **int x** que representa a posição absoluta do caractere, obtida pela função **posicao_absoluta()** — mencionada e analisada anteriormente—. Ela funciona da seguinte forma: o valor contido no índice x é trocado pelo seguinte, ou seja, pelo valor contido no próximo índice até o final do texto. Como é dependente do tamanho n do texto, há uma complexidade $O(n)$. Código abaixo:

Figura 14: função `apaga()`

```
char *apaga(char *texto, int numero_de_caracteres, int x)
{
    for (int j = x; j < numero_de_caracteres; j++)
    {
        texto[j] = texto[j + 1];
    }
    return texto;
}
```

Fonte: captura de tela do código escrito pelo autor

Por último, há o comando **busca()**, responsável por buscar a próxima ocorrência da palavra a partir de determinada posição. Ele devolve a posição do primeiro caractere da próxima ocorrência da palavra. Muito útil para os comandos [B] e [V].

Antes de falar sobre a função, vale justificar a maneira que ela foi implementada: antes de fazê-la, a ideia inicial foi implantar o algoritmo de Boyer-Moore, apresentado em aula e no livro-referência da matéria. Contudo, após diversas tentativas, não consegui implementá-lo, talvez eu estava com uma dificuldade na maneira que o algoritmo identifica a palavra e conta como ocorrência. Logo após essas tentativas frustradas, optei de usar o algoritmo trivial, apresentado no livro-referência, no entanto, tive que fazer uma série de modificações para o seu funcionamento e também para devolver o que eu necessitava, isto é, a posição absoluta do primeiro caractere da próxima ocorrência da palavra.

Feita essas observações sobre a função, cabe a apresentação dela. Primeiramente, temos o seu protótipo:

int busca(char *texto, int numero_de_caracteres, char *palavra, int tam_palavra, int x)

Ela é precedida o texto do comando [A] e as informações sobre seu número de caracteres, ela necessita também da palavra a ser buscada, o tamanho dela, bem como a posição absoluta atual `x`. Ela funciona através de dois laços: o primeiro é o que navega no texto e o segundo realiza a contagem de caracteres da passagem do texto para ver se formam a palavra e, conseqüentemente, a indicação se houve ocorrência ou não dela. Observações importantes sobre o laço é que ele inicia no caractere seguinte da posição atual (`x+1`) e se não

for encontrando devolve a posição absoluta 0. O resultado obtido na função **busca()** é analisado no comando que ela foi chamada. Realizada a justificativa do método empregado, observações e descrição do funcionamento, segue o código abaixo:

Figura 15: função busca()

```
int busca(char *texto, int numero_de_caracteres, char *palavra, int tam_palavra, int x)
{
    int inicio, r;
    // Busca no texto
    for (int k = x + 1; k <= numero_de_caracteres; k++)
    {
        r = 0;
        // Conferindo se há aparição da palavra
        while (r < tam_palavra && palavra[r] == texto[k + r])
        {
            inicio = k;
            r++;
        }

        // Devolução de resultados da busca
        if (r >= tam_palavra - 1)
        {
            return inicio;
        }
        if (k == numero_de_caracteres)
        {
            return 0;
        }
    }
};
```

Fonte: captura de tela do código escrito pelo autor

Sua complexidade de tempo é $O(n^2)$, por possuir dos laços em funcionamento simultâneo, explicados anteriormente.

4. ANÁLISE DE COMANDOS

4.1 O COMANDO A

O comando mais importante do editor de texto é o comando A. Com ele o programa extrai o texto de um arquivo .txt e o organiza na memória para ele ser trabalhado com os demais comandos que interagem com ele.

Para extrair o conteúdo do texto, foi utilizada a função **fopen()** no modo “r”, a minha ideia inicial de método para a extração foi realizá-la caractere por caractere com a função **getc()**. Contudo, além de ser a opção menos viável, ela estava apresentando problemas na transcrição: caracteres fora da tabela ASCII estavam contaminando o texto. Consultei o monitor da disciplina e ele me recomendou o uso da função **fgetc()**. Dessa forma, o texto é extraído linha por linha e não tive problemas na transcrição. A solução foi a mais viável, visto que extinguiu o problema de caracteres indesejados e conferiu elegância ao código por deixá-lo mais legível e eficiente.

Após a extração, como dito anteriormente, o comando A confere organização ao código. Minha primeira ideia foi organizá-lo em uma matriz, sendo as linhas da matriz as linhas do texto e as colunas da matriz os caracteres pertencentes a linha atual. Para isso, eu criei um loop (imagem abaixo) que além de ser deselegante, não funciona. Novamente, tive problemas com caracteres contaminando o texto.

A ideia de organização de matriz foi descartada, porque além de eu ter problemas na organização do texto, estava pensando que com esse ordenamento eu teria dificuldade na implementação das outras funcionalidades. Resolvi então mudar a estratégia, ao invés de trabalhar com o texto em matriz, decidi mexer com ele diretamente. *Grosso modo*, a *string* usada para a extração do texto, seria a mesma utilizada para manipulação com os outros comandos.

Uma observação importante: comando [A] abre a possibilidade do uso de novos comandos. Com exceção do comando de fechamento do programa [!], todos os comandos dependem do texto extraído pelo comando [A], logo foi criada a variável **int sem_texto** para verificar se o texto foi inserido ou não. Se o texto ainda não foi extraído, a variável **sem_texto** tem valor igual a 1, após ser extraído, a variável assume valor 0. Quando possui valor 1, o programa impede que o usuário solicite comandos dependentes de texto e imprime uma mensagem contida no comando **texto_nulo()** que avisa a obrigatoriedade de inserção de um texto pelo comando [A].

4.2. COMANDOS DE MOVIMENTAÇÃO

Na tabela de comandos, há seis que promovem a movimentação do cursor na mesma linha. Dentre eles, temos: o comando [T], [F], [O] e [\$]. Todos estão restritos a trabalhar na mesma linha, ou seja, estão restritos ao intervalo do primeiro caractere até o último caractere de uma mesma linha. Além disso, por trabalharem neste intervalo, a única variável que será alterada na execução de qualquer um desses comandos é a posição relativa, representada pela variável **pos**. Por terem tais características e a mesma natureza, todos os quatro comandos foram agrupados no código.

Além dos quatro, há mais dois: os comandos [P] e [Q]. Na apresentação deste relatório, achei conveniente apresentá-los separadamente dos demais por terem duas características semelhantes que distinguem dos outros quatro. Ambos usam a função **posicao_absoluta()** e a **posicao_relativa()** mencionadas anteriormente e utilizam o ‘ ’ (espaço) para buscar o início da palavra — da mesma palavra com o comando Q e da próxima com o comando P-.

Analisando a complexidade de tempo dos comandos [T], [F] e [O], concluímos que eles têm complexidade $O(1)$ já que cada um, incrementa, decrementa e iguala a zero, respectivamente. Já o comando ‘\$’ tem complexidade $O(n)$ já que está diretamente associado à função **contagem_caracteres_linha()** e os comandos [P] e [Q] também já que estão diretamente relacionadas com as funções **posicao_absoluta()** e a **posicao_relativa()**, ambas possuem uma complexidade $O(n)$. As três funções já foram analisadas na sessão “Funções de posicionamento” e “funções de contagem”.

Além de ter comando de movimentação na linha, existem comandos de movimentação entre linhas, são eles: o comando [J], [H], [:F] e [:x] (sendo x um *int*). Os comandos [J], [H] e [:F] são comandos bem simples e análogos aos comandos [F], [T] e [O] comentados acima. Todos alteram a variável **linha_atual**, J soma um, F subtrai um e [:F] altera para **numero_de_linhas**-1. Por possuírem tal funcionamento, sua complexidade de tempo são $O(1)$.

Já o comando :x é diferente. No primeiro momento ele extrai a informação de qual é a linha destino, utilizando a lógica de extração de informação utilizando os princípios apresentados na sessão “A exclusividade de execução dos comandos que recebem um parâmetro” e “Alocação dinâmica dos comandos com parâmetros”. Logo em seguida, o comando faz uma conversão de *char* para *int* utilizando a função **atoi()** — antes de usá-la fiz algumas pesquisas para realizar a conversão, muitos sites a recomendaram e outros não, acabei optando pelo uso dela, porque ela cumpriu o que eu queria sem problemas -.

Por último, é conduzida uma avaliação do valor convertido: é verificado se é um valor numérico e, se sim, se o valor está no intervalo que existem linhas, isto é, sendo x o valor extraído pelo usuário: $0 \leq x < \text{numero_de_linhas}$.

4.3. EXCLUSÃO DE CARACTERES

Há dois comandos que realizam a exclusão de caracteres: o comando [D] e, indiretamente, o comando [U]. O comando [D] apaga o caractere contido na posição atual, já o comando [U], primeiramente, verifica se há duas ou mais linhas, em seguida, localiza o próximo ‘\n’ presente na linha para, finalmente, apagá-lo.

Ambos utilizam a função **apaga()**, já analisada na sessão “funções de manipulação”. Particularmente, considero que a ideia de usar a mesma função para estes dois comandos foi uma ótima ideia, já que conferiu elegância ao programa por evitar repetição desnecessária de código.

Após a exclusão do caractere, as variáveis **numero_de_linhas** e **numero_de_caracteres** são atualizadas para o pleno funcionamento das demais funções que as utilizam, tópico discutido em “Atualização de linhas e número de caracteres” e as funções responsáveis pelo processo, **contagem_linha()** e **contagem_caracteres**, também já foram analisadas na sessão “funções de contagem”.

4.4. INSERÇÃO DE CARACTERES

O comando [I] é responsável por inserir uma palavra no texto. Como dito anteriormente, ele segue os princípios apresentados em “A exclusividade de execução dos comandos que recebem um parâmetro” e “Alocação dinâmica dos comandos com parâmetros”.

Logo após o processo de extração da palavra, manipulamos para ela ser inserida no texto com a função **insere()**, já analisada na sessão “funções de manipulação”. Após a função **insere()**, a memória alocada na variável **texto** é liberada pela função **free()** e ocorre um novo processo de alocação dinâmica na variável em questão, adicionando espaço de memória suficiente para alocar a palavra e o texto. Por fim, o conteúdo da *string* da variável auxiliar, **texto_aux**, é transcrito para a variável **texto**. Todos os processos de transcrição dependem do tamanho n do texto e da palavra que são transcritos, portanto, a complexidade de tempo e espaço envolvidas no processo é na ordem de $O(n)$.

O comando [N] é responsável por separar as linhas e ele, assim como o comando [I], é um comando de inserção, porém, ao invés de uma palavra, somente o caractere ‘\n’. No momento de desenvolvimento do editor de texto, cogitei em utilizar a mesma função, **insere()**, mas conferindo as notas de aula e o livro referência da disciplina, achei uma opção mais elegante e, consequentemente, melhor.

Na sessão 3.4 do livro “Algoritmos em C”, a função de inserção de um caractere em um vetor foi utilizada para a quebra de linha. O algoritmo é muito simples e consiste na seguinte lógica: primeiramente, é definido duas variáveis de posição absoluta, obtidas pela função **posicao_absoluta()**, a variável **x**, referente a **pos** (posição atual) e a variável **y**, referente ao final da linha. Em seguida, com o uso de um laço de início **y** e final **x** com decremento em um, o caractere atual é substituído pelo seguinte e, no fim, o caractere **texto[x]** é substituído por ‘\n’. No final do processo, ocorre a atualização da variável **pos** para ela ficar no final da quebra de linha e das variáveis **numero_de_linhas** e **numero_de_caracteres**, da premissa já explicada em: “Atualização de linhas e número de caracteres”. Código abaixo:

Figura 16: comando [N]

```
else if (comando[i] == 'N')
{
    // Definição de posições absolutas
    int x = posicao_absoluta(texto, linha_atual, pos);

    // Laço para inserção do caractere \n
    for (int j = numero_de_caracteres; j > x; j--)
    {
        texto[j] = texto[j - 1];
    }
    texto[x] = '\n';

    // Atualização de dados
    numero_de_caracteres = contagem_caracteres(texto);
    numero_de_linhas = contagem_linhas(texto);
    pos = contagem_caracteres_linha(texto, linha_atual) - 1;
}
```

Fonte: captura de tela do código escrito pelo autor

4.6. COMANDOS DE PILHA

No código, os comandos ‘M’, ‘C’, ‘V’, ‘X’ e ‘Z’ são relacionados a pilha. Em linhas gerais, esses comandos simulam o CTRL+C (copiar), CTRL+X (recortar) e CTRL+V (colar) dos editores de texto usuais, porém usando a estrutura de pilha. No programa relatado, optei por usar a estrutura de pilha vetorizada.

A princípio, é necessário estabelecer as características do comando ‘M’. É um comando simples de complexidade $O(1)$ de tempo e espaço, por se tratar de uma simples atribuição de valor. Por padrão, o valor inicial de M é (0,0), sendo o primeiro número representando a posição relativa do caractere marcado pelo cursor (variável **marca_pos**) e o segundo número a linha que o caractere em questão se encontra (variável **marca_linha**). A função **referencia()** mencionada anteriormente é responsável pela apresentação ao usuário dos números de marcação.

Os comandos de empilhamento, [C] e [X] possuem o mesmo funcionamento com a única diferença que o segundo apaga com a função **apaga()** o caractere que foi empilhado. O empilhamento é feito de trás para frente, para respeitar a característica da pilha denominada como LIFO, em inglês, *Last-In-First-Out*, ou seja, último a entrar, primeiro a sair.

Para o processo em questão há duas posições a serem consideradas, a **posicao_pilha** e a **posicao_marcada**. A primeira é a posição absoluta do caractere em que foi executado o comando ‘X’ ou o comando ‘C’, ou seja, **pos**, e a segunda é a posição absoluta da posição marcada, ambas obtidas com a função **posicao_absoluta()**.

Logo em seguida, é analisado qual delas é maior e o processado de empilhamento começa. Caso, o comando ‘X’ for executado, com o empilhamento é realizado o processo de apagar o caractere atual, para isso é utilizado a função **apaga()** — mencionada e analisada da sessão de “funções de manipulação” do presente relatório. Além disso, por motivos de eficiência no uso de espaço, a variável **pilha** é alocada dinamicamente.

O comando ‘Z’ é trivial, não exige que o conteúdo seja desempilhado, mas apenas apresentado. Para isso, foi criado um laço que inicia do fim da pilha até seu começo, devido à maneira que ela foi empilhada. Vale destacara que a complexidade de tempo e espaço é proporcional ao tamanho n da pilha, portanto, $O(n)$.

Finalmente, o comando [V] utiliza o algoritmo de desempilhar, da função **desempilha()**, mencionada na sessão “funções de manipulação”. Há algumas observações importantes:

1. Achei importante colocar uma variável que indica se o texto já foi desempilhado ou não, porque tive problemas de desempilhar o mesmo conteúdo várias vezes. A variável em questão segue o nome de **desempilhado**;
2. Apesar de o texto ser desempilhado, a variável t depois do comando **desempilha()** não devolve $t = 0$, por este motivo, achei conveniente forçar um $t = 0$ no comando em questão.

3. Todos os comandos de manipulação direta com texto necessitam atualizar o número de caracteres e linhas a fim de “atualizar” a quantidade de caracteres para a movimentação do texto.
4. Assim como dito anteriormente, o comando apresenta patologias, caso o texto empilhado/desempilhada possua abundância de caracteres.

4.6. SALVANDO O TEXTO

O comando E, descrito nas instruções como um comando de sobrescrita para um arquivo de nome n, cumpre a tarefa de salvar as modificações realizadas no texto em um arquivo com o nome inserido pelo usuário. A extração do nome do arquivo segue os preceitos em apresentados em: “A exclusividade de execução dos comandos que recebem um parâmetro” e “Alocação dinâmica dos comandos com parâmetros”.

Após testar colocar extensões diferentes de arquivo, como .pdf, cheguei a conclusão de que idealmente a função E deveria aceitar apenas arquivos .txt. Diante disso, coloquei a restrição de obrigatoriamente o nome do arquivo terminar com .txt.

Na função **fopen()**, foi utilizada o modo de escrita “w” e transcrição de conteúdo é realizada com a função **fputs()**, os possíveis erros de abertura e gravação foram definidos e, de maneira geral, não tive problemas na criação deste comando.

4.7 COMANDOS DE BUSCA

No editor de texto, há dois comandos de busca: o comando [B] e comando [S] — sendo rigoroso na classificação, o comando [S] pertence a três classes de comandos, já que este utiliza as funções **busca()**, **insere()** e **apaga()**—.

O comando [B] funciona da seguinte forma: ele segue a premissa dos conceitos expostos em: “A exclusividade de execução dos comandos que recebem um parâmetro” e “Alocação dinâmica dos comandos com parâmetros”. Após extrair a *string-parâmetro* do comando, ele a aloca dinamicamente e esta é utilizada como parâmetro no comando **busca()** — explicado detalhadamente na sessão “funções de manipulação”

Esta função, se caso encontrar a palavra, ela devolve a posição absoluta dela, o programa, depois o programa converte para a posição relativa, com as funções **descobre_linha()** e **posicao_relativa()**, ambas explicadas na sessão “funções de posicionamento”. Finalmente, se caso a posição absoluta for igual a zero, o programa exibe

uma mensagem que a palavra não foi encontrada, caso contrário, o comando chama as funções de exibição e referência com o cursor apontando para o início da palavra buscada.

Particularmente, a decisão de retornar zero foi uma ótima decisão tomada, já que a função busca inicia do caractere seguinte da posição absoluta do cursor chamado. Portanto, é impossível que a posição absoluta zero seja encontrada com a função, sendo assim, fornecendo um valor que pertence ao texto, porém indicando uma invalidade do comando.

Passando para o comando [S], infelizmente, foi o único comando que eu não consegui aplicar de maneira correta. Durante seu desenvolvimento, ele apresentou diversas falhas, todas elas mencionadas a seguir.

O comando [S] segue o preceito definido em “A exclusividade de execução dos comandos que recebem um parâmetro”, porém o conceito de “Alocação dinâmica dos comandos com parâmetros” não foi aplicado e tive que usar *strings* de tamanho estático. Isso foi uma decisão proveniente de diversas tentativas frustradas de aplicar a alocação dinâmica, o programa apresentava problemas na alocação e o problema foi contornado apenas com o uso de *strings* de tamanho estático. Com essa decisão, elegância foi perdida, já que o programa pode apresentar uso excessivo de memória, caso as duas *strings* (**subs** e **buscada**) sejam menores que o tamanho definido ou podem suspender o funcionamento, caso forem maiores.

Além deste problema, para conferir elegância ao código e evitar linhas desnecessárias, eu utilizei as funções: **busca()**, **apaga()** e **insere()**. Em suma, o comando funciona na seguinte forma: primeiro, ele extrai as duas palavras e utiliza ‘/’ como divisor entre elas, logo em seguida, ele realiza dois laços, o primeiro conta a quantidade de caracteres de cada palavra e o outro, preenche o conteúdo das duas *strings*. Em seguida, ele chama a função **busca()** e ela devolve a posição onde o palavra está, a função **apaga()** apaga os caracteres ocupados pela palavra que será substituída e, ao final, a função **insere()** insere a palavra que substituirá.

Infelizmente, no código apenas consegui realizar a substituição somente da próxima ocorrência da palavra e não de todas as palavras no texto. Tentei criar um loop para isso, mas tive problemas. Provavelmente, a razão para tal ocorrido esteja relacionada com o problema mencionada na função **insere()** discutida na sessão “funções de comando”.

Finalmente, o comando possui uma complexidade de tempo $O(n^2)$ e $O(n)$ de espaço por depender de uma série de laços que dependem do tamanho do texto, da palavra buscada e da palavra que substituirá esta.

5.DISSCUSSÃO SOBRE UM EDITOR DE TEXTO INTERATIVO

Na formulação desta aplicação, uma das principais pergunta era: como tornar um programa interativo? Julgo que limitar o uso indevido do código por parte do usuário final e estabelecer uma comunicação com ele em alguns casos para indicar que o programa está funcionando, são os dois pilares para tornar o editor de texto mais interativo.

Para evitar tal “fuga” foram realizadas uma série de limitações para não ocorrer o uso indevido do editor por parte do usuário. Por exemplo, sem o texto extraído do comando [A] todas as funções, exceto o comando [!], não funcionam, para evitar possíveis foi criado uma variável que indica se o texto foi extraído ou não. Se sim, os demais comandos estão liberados para o uso, caso contrário, o usuário fica restrito e o programa o avisa da obrigatoriedade da precedência do uso do comando [A].

Além da limitação do próprio usuário, para estabelecer interatividade com ele, é necessário comunicação. Em alguns comandos, eu julguei necessário o comando avisar que funcionou, como os comandos de empilhamento, [C] e [X], visto que após sua execução não há nada indicando ao usuário final se o comando funcionou ou não

Com esses dois pilares é possível realizar uma melhor interação e relacionamento entre a aplicação e o usuário. Dessa forma, erros são evitados, o uso esperado da função tem mais chances de ocorrer e, finalizando, a comunicação entre usuário e aplicação é melhorada, criando-se um melhor uso desta.

6 CONCLUSÃO

Para a realização deste trabalho foi necessário conhecimento da linguagem de programação em questão, com o conhecimento de estrutura de dados, principalmente, vetores de *char*, *strings*, e pilhas. Saber tratá-los de maneira correta e, na medida do possível, da forma mais eficiente, é essencial para a construção de um programa que tem como principal tarefa a manipulação e interação com um texto.

Durante o desenvolvimento, como relatado, muitas ideias foram descartadas para dar lugar a outras que promoviam o pleno funcionamento da aplicação e uma elegância do código para melhor compreensão de quem precisa interpretá-lo ou manuseá-lo. Apesar de ter algumas dificuldades em sua elaboração, creio que foi um ótimo projeto para trabalhar e principalmente testar meus conhecimentos aprendidos na disciplina.

REFERÊNCIAS

BACKES, André. **Linguagem C - Completa e Descomplicada**. 2. ed. Rio de Janeiro: Gen Ltc, 2018. 448 p.

FEOFILOFF, Paulo. **Algoritmos em C**. Rio de Janeiro: Elsevier, 2009. 232 p.

MAUÁ, Denis Deratani. Alocação dinâmica. Apresentação de slides em .pdf. Disponível em: <https://edisciplinas.usp.br/pluginfile.php/7348292/mod_resource/content/1/malloc.pdf>. Acesso em: 08 de dez. de 2022.

MAUÁ, Denis Deratani. Manipulação de texto. Apresentação de slides em .pdf. Disponível em: <https://edisciplinas.usp.br/pluginfile.php/7325586/mod_resource/content/1/aula4.pdf>. Acesso em: 08 de dez. de 2022.