MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PIAUÍ
Campus Picos
Sistemas de Informação

Universidade Federal do Piauí - Campus CSHNB
Sistemas de Informação
Algoritmos e Programação II
João Marcos Sousa Rufino Leal
Raildom da Rocha Sobrinho
João Arthur Carvalho Oliveira

Trabalho Prático

1 Introdução

O desenvolvimento de um editor de texto básico que suporta os comandos "type c"e "undo t"representa uma implementação fundamental e eficaz para a manipulação de texto. O comando "type c"adiciona o caractere c ao final do texto atual, possibilitando a construção do texto de maneira incremental e intuitiva. Por sua vez, o comando "undo t"permite desfazer todas as operações realizadas nos últimos t segundos, retornando o texto ao seu estado anterior e refletindo a funcionalidade essencial encontrada em editores de texto modernos.

Para ilustrar o funcionamento do sistema, considere um exemplo prático: ao adicionar os caracteres 'a', 'b', e 'c' na sequência e, em seguida, emitir um comando "undo 3", o editor reverte o texto para 'a', removendo os caracteres 'b' e 'c' que foram inseridos nos últimos 3 segundos. Este exemplo demonstra como o editor de texto é capaz de gerenciar e reverter operações de maneira precisa, mantendo a integridade do texto de acordo com as ações realizadas.

A abordagem adotada para o desenvolvimento deste editor oferece uma solução clara e prática para as operações básicas de manipulação de texto. O sistema é projetado para ser eficiente e funcional, garantindo que os comandos "type c"e "undo t"sejam executados de maneira confiável e eficaz. Com uma estrutura de código bem definida, o editor não apenas realiza as funções básicas, mas também estabelece um alicerce robusto para futuras expansões e melhorias.

Este editor de texto serve como uma base sólida para o desenvolvimento de ferramentas de edição mais avançadas. Ele fornece uma fundação para a implementação de funcionalidades adicionais, como suporte a múltiplos formatos de texto, opções de formatação e uma gestão mais sofisticada do histórico de operações. A clareza e a funcionalidade do sistema permitem uma integração mais fácil de novos recursos e ajustes, contribuindo para a construção de uma ferramenta de edição de texto mais abrangente e poderosa.

Além disso, a implementação deste editor pode fornecer insights valiosos sobre o comportamento dos usuários e sobre como eles interagem com o texto. Analisar o uso dos comandos e a frequência de desfazer ações pode revelar padrões úteis para otimizar a interface e melhorar a usabilidade geral.

Ainda, a possibilidade de adicionar funcionalidades como a busca e substituição de texto oferece um valor adicional significativo. Implementar uma ferramenta de busca e substituição permite aos usuários encontrar rapidamente partes específicas do texto e realizar alterações em massa, aumentando a eficiência e a flexibilidade na edição. Essa característica é especialmente útil em documentos extensos, onde alterações frequentes e localizadas são necessárias. Ao incorporar esses recursos adicionais, o editor pode se tornar uma ferramenta ainda mais completa e adaptada às diversas necessidades dos usuários.

2 Solução Proposta

A solução proposta para o editor de texto simula a funcionalidade de type e undo através de um processamento eficiente de comandos lidos a partir de um arquivo. A implementação é organizada de forma modular para facilitar a compreensão e manutenção do código. O editor lê comandos de um arquivo de entrada, aplica as operações especificadas, e escreve o resultado em um arquivo de saída. A seguir, detalhamos como cada componente da solução contribui para o funcionamento geral do programa.

2.1 Inclusão de Bibliotecas e Definições

Para realizar as operações básicas, o código faz uso de algumas bibliotecas padrões:

- **stdio.h:** Utilizada para abrir, ler e escrever arquivos.
- **stdlib.h:** Usada para conversões de tipos de dados, como a transformação de strings em números.
- **string.h:** Facilita a manipulação de strings, como dividir e comparar textos.
- **ctype.h:** A biblioteca ctype.h fornece funções para classificar e converter caracteres.

2.2 Estrutura de Dados

Para armazenar os comandos, o programa usa uma estrutura que mantém informações sobre cada operação, como o tipo de comando, o caractere a ser adicionado (no caso de type), a quantidade de números (no caso do undo), e o tempo de reversão (no caso de undo). Essa estrutura facilita o processamento e execução eficiente dos comandos. Ela contém quatro partes principais:

- **Comando:** Uma string que armazena o comando extraído de cada linha do arquivo.
- **caractere:** Uma string que armazena os caracteres dos types.
- **numero_undo:** Um array de inteiros para armazenar os números dos undos.
- **Segundos:** Um array de inteiros para armazenar os segundos extraídos.

2.3 Movimentação dos Ponteiros no Arquivo

Quando um editor de texto processa comandos que incluem uma lógica de undo, como no caso deste projeto, é essencial manipular os ponteiros de arquivo corretamente para garantir que os dados sejam lidos e processados na ordem correta.

2.3.1 Leitura dos Comandos

- **Rebobinamento Inicial:** Antes de processar os comandos, o arquivo de entrada é rebobinado com rewind(arquivo_entrada). Isso garante que a leitura comece a partir do início do arquivo, permitindo que cada linha de comandos seja processada do início ao fim. No entanto, o processamento das operações em si ocorre em ordem inversa.
- **Leitura e Processamento de Comandos:** O código não lê diretamente de trás para frente no arquivo; em vez disso, lê os comandos sequencialmente e os armazena na estrutura COMANDO. Uma vez que todos os comandos em uma linha são lidos, eles são processados em ordem inversa.

- **Leitura de Strings e Comandos:** O código usa fgets para ler linhas de comandos, identificando comandos type e undo. Para cada comando type, é lido um caractere associado. Para undo, é lido um número que especifica o tempo durante o qual as operações devem ser desfeitas.
- **Ajustes do Ponteiro:** O uso de fseek ajusta o ponteiro do arquivo conforme necessário para ler os caracteres e números associados corretamente. Isso inclui movimentar o ponteiro para avançar sobre caracteres que não são necessários para a interpretação direta dos comandos.

2.3.2 Interpretação dos Comandos

Após a leitura dos comandos, o ponteiro do arquivo é ajustado para garantir que a leitura dos dados continue corretamente:

- **Ajuste do Ponteiro para undo:** Quando um comando undo é encontrado, o ponteiro precisa ser ajustado para capturar os segundos associados a ele. Isso é feito movendo o ponteiro para ler o próximo número de forma sequencial após a palavra undo.
- **Pular Caracteres e Delimitações:** Depois de ler um comando, o ponteiro é movido para o início do próximo comando ou bloco de dados, conforme necessário. Isso envolve pular caracteres de separação (como espaços ou vírgulas) para garantir que a próxima leitura comece corretamente.

2.3.3 Processamento de Comandos em Ordem Inversa

Após ler todos os comandos de uma linha:

- **Execução em Ordem Inversa:** Os comandos são processados de trás para frente para aplicar a lógica de undo corretamente. Isso significa que os efeitos dos comandos type podem ser revertidos se um undo subsequente assim o exigir.
- **Ajuste do Índice:** Durante a aplicação dos comandos, o índice da estrutura COMANDO é ajustado. A função Undo manipula o índice i para retroceder o processo sobre comandos type que precisam ser desfeitos, assegurando que o texto final reflita todas as operações corretamente.

2.4 Função Principal (main)

A função principal é o ponto de partida do nosso programa. Ela tem algumas responsabilidades importantes:

- **Gerenciamento de Arquivos:** A função principal abre o arquivo de entrada, de onde vamos ler os comandos, e o arquivo de saída, onde o texto final será gravado. Ela verifica se os arquivos foram abertos corretamente para evitar erros durante a execução.
- **Coordenação do Processo:** Ela também coordena o fluxo do programa. Isso significa que conta quantas linhas de comandos existem no arquivo de entrada, pois cada linha representa um conjunto de comandos que precisa ser processado separadamente.
- **Chamadas de Função:** Para cada linha de comandos, a função principal chama as outras funções que processam e executam os comandos. Quando tudo estiver feito, ela garante que os arquivos sejam fechados corretamente.

2.5 Função processa_dados

Essa função é responsável por ler e interpretar os comandos de uma linha do arquivo de entrada:

- **Leitura de Comandos:** Ela lê uma linha do arquivo e divide essa linha em comandos individuais (type e undo). Para cada comando type, ela armazena o caractere que deve ser adicionado ao texto. Para cada comando undo, ela armazena o tempo durante o qual as ações devem ser desfeitas.
- **Armazenamento Estruturado:** Os comandos lidos são organizados em uma estrutura de dados (como um array de estruturas), o que facilita o acesso e manipulação das informações durante a execução.

2.6 Função processa_linha

Depois de organizar os comandos, essa função executa as operações conforme os comandos armazenados:

- **Execução de Comandos:** Ela percorre os comandos em ordem inversa, começando do último até o primeiro, aplicando a lógica de desfazer operações (undo) antes de adicionar operações (type).
- **Construção do Texto:** Durante essa execução, o texto é montado e atualizado na memória. Cada caractere adicionado por um comando type é incorporado ao texto atual.
- **Gravação do Texto Final:** Quando todos os comandos de uma linha foram processados, o texto resultante é escrito no arquivo de saída.

2.7 Função Undo

A função Undo é responsável por desfazer os comandos type e undo realizados dentro de um período de tempo especificado por um comando undo. Metodologia de como a função Undo trabalha:

- **Identificação do Comando Undo:** Quando um comando undo é encontrado, a função Undo é chamada com o índice do comando undo atual.
- **Iteração sobre Comandos Anteriores:** A função então verifica os comandos anteriores a partir do comando undo atual. Ela faz isso através de um loop que itera para trás na lista de comandos, começando pelo comando anterior ao undo.
- **Verificação do Tempo:** Para cada comando anterior, a função verifica se ele foi executado dentro do tempo especificado no comando undo. Isso é feito comparando o campo segundos de cada comando type anterior com o tempo de undo.
- **Desfazendo Comandos Type:**

  Se um comando type anterior foi realizado dentro do período de tempo do comando undo, a função decrementa o índice i, que controla a execução do loop principal. Isso efetivamente ignora ou remove o caractere adicionado por esse comando type.

  A imagem 1 demostra a lógica implementada através de um fluxograma.

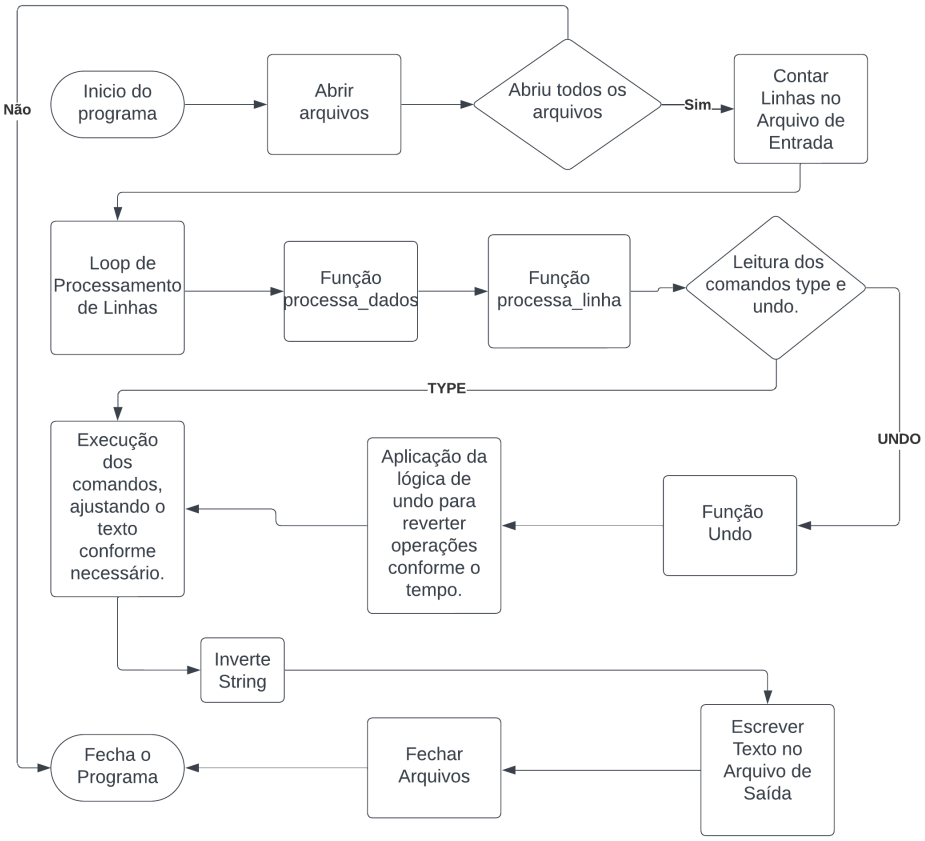

Figura 1: Fluxo de Execução

3 Análise de Desempenho

Para a análise de desempenho, utilizaremos a notação de complexidade de algoritmos Big O. Dividiremos a análise em seções que correspondem às funções principais do código, elencando suas complexidades individuais e, por fim, a complexidade total do programa. Isso nos fornecerá parâmetros para o desempenho do programa como um todo.

3.1 Função '`main`'

1. função 'main' é responsável por abrir os arquivos de entrada e saída, contar o número de linhas no arquivo e processar cada linha de comandos chamando as funções `processa_dados` e `processa_linha`.

   A contagem de linhas é feita por meio de um loop '`while`' que lê caractere por caractere, incrementando o contador de linhas a cada '\n'. Como esse loop '`while`' lê todos os caracteres, sua complexidade é $O(N)$, onde N é o número de caracteres no arquivo de entrada.

3.1.1 Processamento de linha

2. O processamento de cada linha é feito por um loop 'for' que chama as funções '`processa_dados`' e '`processa_linha`'. Assim, a complexidade é $O(L * (C1 + C2))$, onde L é o número de linhas, $C1$ é a complexidade de '`processa_dados`' e $C2$ é a complexidade de '`processa_linha`'.

3.2 Funçao 'processa_dados'

1. A função '`processa_dados`' é responsável por ler e processar corretamente os comandos de uma linha do arquivo de entrada. Isso é feito por meio de um loop '`do-while`' que lê os comandos até encontrar um comando que não seja '`type`' ou 'undo'. Portanto, sua complexidade é $O(M)$, onde M é o número de comandos por linha.

2. Há ainda um loop '`do-while`' responsável por processar os segundos dos comandos, também com complexidade $O(M)$, onde M é o número de comandos.

   No pior caso, ambos os loops processam todos os comandos em uma linha, resultando em uma complexidade combinada de $O(M)$.

3.3 Função `processa_linha`

A função `processa_linha` constrói o texto a partir dos comandos e escreve o resultado no arquivo de saída.

1. O loop `for`, de trás para frente, itera sobre os comandos procurando os `undo` e, caso os encontre, processa a quantidade de segundos a serem desfeitos. A complexidade é $O(M \cdot U)$, onde M é o número de comandos e U é o número de comandos a serem desfeitos.

2. A reversão da string: o loop `for` reverte a string para que seja impressa na ordem correta no arquivo de saída, com complexidade $O(T)$, onde T é o comprimento da string.

3. Por fim, o `fprintf` imprime a string, já na ordem correta, no arquivo de saída, com complexidade $O(T)$.

3.4 Função 'undo'

1. A função '`undo`' desfaz um determinado número de comandos baseando-se nos segundos de cada comando. Nesta função, há um loop '`for`' que executa '`numero_undo`' vezes, onde '`numero_undo`' é o número de comandos a serem desfeitos. Portanto, a complexidade é $O(U)$, onde U é o número de comandos a serem desfeitos.

3.5 Conclusão

Para uma linha de comandos:

- `processa_dados`: $O(M)$
- `processa_linha`: $O(M \cdot U + T)$

Para todas as linhas:

- `main`: $O(L \cdot (O(M) + O(M \cdot U + T))) = O(L \cdot M \cdot (1 + U + \frac{T}{M}))$

Simplificando, a complexidade geral é $O(L \cdot M \cdot U)$, assumindo que T e M são comparáveis.

Portanto, a complexidade final do código é $O(L \cdot M \cdot U)$, onde L é o número de linhas, M é o número de comandos por linha e U é o número de comandos a serem desfeitos. Isso nos dá uma complexidade de crescimento linear, como mostrado no gráfico 2.

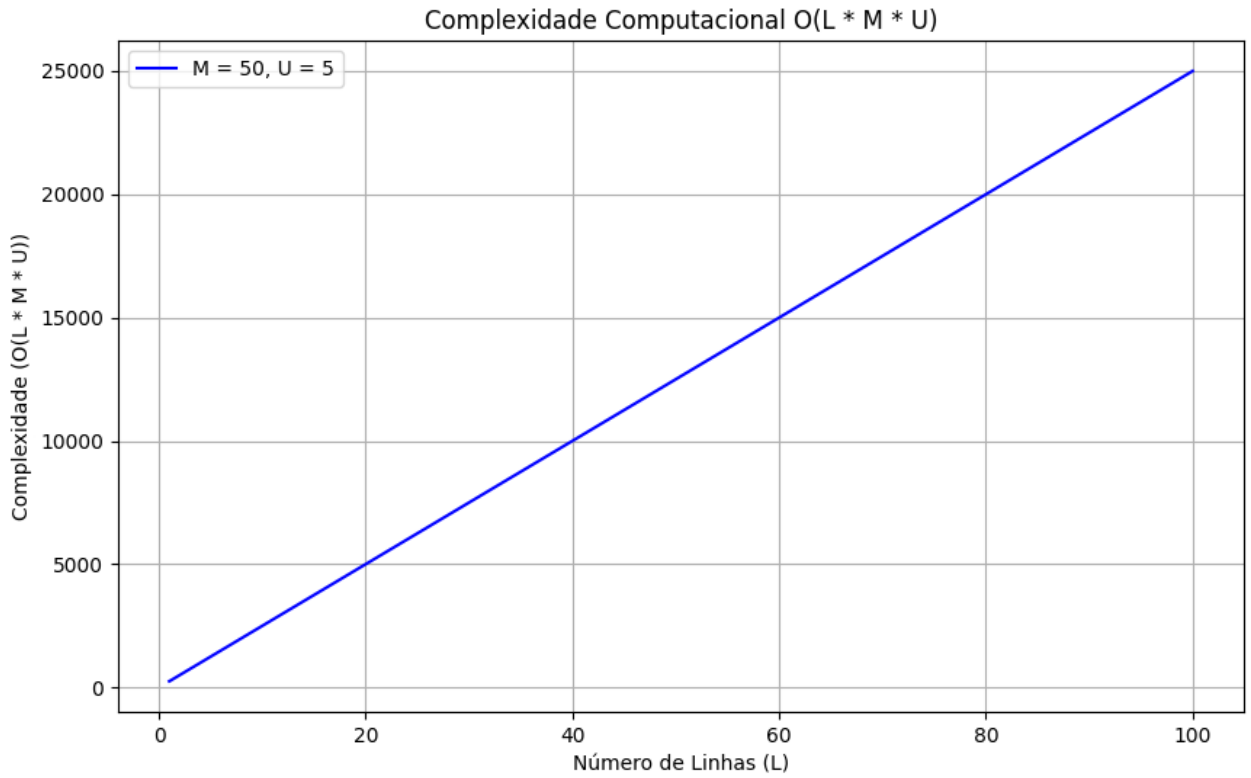

Figura 2: Gráfico de Complexidade

4 Conclusão

O editor de texto desenvolvido é uma solução prática e eficiente para a manipulação básica de texto, oferecendo os comandos essenciais de "type"e "undo". A implementação desses comandos resulta em um sistema que lida bem com cenários de uso típicos, proporcionando uma experiência de usuário satisfatória.

A modularidade do código é um dos principais pontos fortes do sistema, garantindo clareza e facilitando a manutenção e expansão do software. Esta estrutura modular permite que o sistema seja facilmente modificado ou aprimorado sem comprometer sua integridade geral.

No entanto, o sistema enfrenta limitações quando submetido a grandes volumes de dados ou a um elevado número de operações de "undo". Esses pontos críticos indicam a necessidade de melhorias para otimizar o desempenho e aumentar a robustez do editor. Considerações futuras podem incluir a implementação de algoritmos mais eficientes para o gerenciamento do histórico e a utilização de técnicas de armazenamento aprimoradas.

A base sólida estabelecida por este projeto proporciona um ponto de partida promissor para futuras evoluções. A implementação de novos recursos, como funcionalidades avançadas de edição e suporte a diferentes tipos de arquivos, pode ser explorada para aumentar a versatilidade e a funcionalidade do sistema. Além disso, melhorias na arquitetura para lidar de maneira mais eficiente com múltiplas operações de "undo"poderão aprimorar ainda mais a experiência do usuário.

Em suma, o editor de texto não apenas resolve o problema proposto de maneira eficaz, mas também estabelece uma fundação robusta para a expansão e aprimoramento contínuos, garantindo que o sistema possa evoluir para atender a necessidades mais complexas e exigentes no futuro.