Projeto Final – Editor de Texto

Antonio Gabriel Freitas da Silva - 13687290

Strutcs

Ao todo, foram usadas 3 structs, uma delas é “linha”, que será as linhas que serão lidas da lista encadeada. Além disso, há também “Pilha” e “Cursor”, para utilizarmos a pilha de memória e indicar as posições do cursor ao decorrer da leitura do arquivo que pode ser aberto no editor de texto implementado.

Comparação

Ao início do programa, serão vistas essas duas declarações que cumprem uma tarefa de “pseudo-função” para visualizar máximo e mínimo entre dois elementos inteiros:

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))

#define MIN(a,b) ((a) < (b) ? (a) : (b))
```

Funções preliminares

As funções linha “inicializa_lista”, “Cursor_” e “mostrarCursor” foram criadas apenas para fazer uma ilustração do cursor na linha de comando e inicializar a lista encadeada que é criada como principal estrutura de dados, a terceira, em especial, analisa se o índice do cursor (denotado como indice na struct Cursor) está menor que o tamanho do texto e assim prossegue mostrando a sua posição em coordenadas, além disso, Cursor_ também mostra o marcador sendo mostrado na linha 0 do programa.

Funções principais

Main: A função principal começa inicializando algumas variáveis, incluindo dois ponteiros para o tipo char que armazenam o antigo e novo texto a ser substituído, uma estrutura de dados chamada linha, que representa cada linha do arquivo, uma estrutura de dados chamada Cursor que armazena a posição atual do cursor, uma estrutura de dados chamada Pilha que será usada para armazenar o estado anterior do cursor.

Em seguida, o programa solicita ao usuário o nome do arquivo a ser carregado. Depois de ler a entrada do usuário, o programa começa um loop infinito que mostra o cursor na tela e aguarda uma ação do usuário. A ação do usuário é lida como uma string de caracteres.

O programa usa um switch case para interpretar a entrada do usuário. Cada case representa uma diferente ação que o usuário pode fazer, como carregar um arquivo, inserir texto, deletar texto, mover o cursor, entre outras. Quando a ação é identificada, a função correspondente é chamada. Algumas dessas funções são: carregarArquivo, inserirTexto, deletarTexto, moveCursor,

juntarLinhas, separaLinha, procurarPalavra, substituirPalavra, C, X, V, Z, além de um M que vai ser denotado como o marcador do editor de texto, que ao ser acionado, irá marcar a posição do cursor.

Ao final, a função main retorna zero, indicando que o programa foi executado com sucesso.

void carregarArquivo(char *arquivo, linha *cabeca): A função "carregarArquivo" abre o arquivo de texto especificado pelo nome "arquivo" e carrega o conteúdo desse arquivo em uma lista encadeada com nós do tipo "linha". Cada nó da lista contém o texto de uma linha do arquivo. A função começa verificando se o arquivo pode ser aberto com a função "fopen", e se não for possível abri-lo, exibe uma mensagem de erro e retorna. Caso contrário, a função inicializa a lista encadeada usando a função "inicializa_lista" e cria uma variável "atual" que irá percorrer a lista. Em seguida, a função usa a função "fgets" para ler as linhas do arquivo de texto e armazená-las em uma variável temporária "buffer". Se a linha lida termina com um caractere de nova linha, ele é removido do buffer. Em seguida, a função aloca memória suficiente para armazenar o texto da linha e usa a função "strcpy" para copiar o conteúdo do buffer para o nó da lista encadeada. Em seguida, uma nova linha é criada para a próxima iteração do loop. O processo é repetido até que todas as linhas do arquivo sejam lidas. Por fim, a função fecha o arquivo usando a função "fclose" e retorna.

void sobrescreverArquivo(linha *cabeca, char *arquivo): A função sobrescreverArquivo é usada para salvar as alterações feitas no arquivo aberto na memória. Ela recebe dois argumentos: uma estrutura linha que representa a lista encadeada que contém o texto do arquivo e o nome do arquivo (arquivo). O primeiro passo da função é abrir o arquivo para escrita com a função fopen. Se o arquivo não puder ser aberto, a função exibe uma mensagem de erro e retorna. Em seguida, a função usa um ponteiro de linha (atual) para percorrer a lista encadeada linha. Enquanto atual não for igual a NULL, a função usa a função fprintf para escrever cada linha (representada pela variável atual) no arquivo, seguido de uma nova linha (\n). Depois de escrever todas as linhas no arquivo, a função fecha o arquivo com a função fclose e termina a execução.

void mostrarLinha(linha *cabeca, Cursor *cursor): A função mostrarLinha recebe como argumentos uma lista ligada de linhas (linha *cabeca) e um ponteiro para um objeto Cursor que representa o cursor atual na lista. A função procura a linha na lista cujo número seja igual ao número da linha armazenado no objeto Cursor e a imprime na saída padrão. Em termos de complexidade de tempo, a função percorre a lista ligada de linhas, o que resulta em uma complexidade de tempo de O(n), onde n é o número de linhas na lista. Em termos de complexidade de memória, a função não requer alocação adicional de memória, portanto sua complexidade de memória é de O(1).

inserirTexto(linha *cabeca, char *texto, Cursor *cursor): Esta função insere um texto dado em uma determinada linha na lista encadeada de linhas.

Aqui, a lista encadeada de linhas é representada pelo ponteiro "cabeca". A posição onde o texto será inserido é determinada pelo ponteiro "cursor", que contém informações sobre a linha e o índice na linha onde o texto será inserido. A função começa percorrendo a lista encadeada a partir do ponteiro "cabeca" até chegar na linha desejada, determinada pelo campo "numeroLinha" do cursor. Em seguida, é calculado o tamanho do texto atual na linha e do texto a ser inserido. Aloca-se memória para armazenar o novo texto resultante da concatenação dos dois. Em seguida, o texto atual é copiado para o novo texto, seguido pelo texto a ser inserido na posição indicada pelo cursor. O restante do texto atual é copiado para o final do novo texto. Por fim, a função atualiza a linha atual com o novo texto e libera a memória alocada para o texto antigo. Também é atualizada a posição do cursor, acrescentando o tamanho do texto inserido. A complexidade de tempo desta função é O(n), onde n é o tamanho da linha onde o texto está sendo inserido. Isso se deve ao fato de que a função precisa percorrer a lista encadeada até chegar na linha desejada e também realizar cópias de texto de tamanho n.

void deletarTexto(linha *cabeca, Cursor *cursor): A função deletarTexto é usada para excluir um caractere na posição do cursor atual. A função começa buscando a linha atual na lista de linhas, usando um loop que percorre até a posição cursor->numeroLinha. Em seguida, a função determina o tamanho atual do texto na linha selecionada. Depois, é alocado um novo espaço de memória para o novo texto, usando o tamanho atual do texto. O próximo passo é copiar o texto atual para o novo espaço de memória, excluindo o caractere na posição do cursor. Isso é feito usando duas chamadas a strncpy, que copiam os caracteres antes e depois da posição do cursor, respectivamente. Por fim, o caractere nulo é adicionado ao final do novo texto. Por fim, a linha atual é atualizada com o novo texto e o espaço de memória alocado para o texto antigo é liberado. A complexidade de tempo desta função é de O(n), onde n é o tamanho do texto da linha atual. Isso ocorre porque a função precisa percorrer o texto inteiro uma vez para copiá-lo para o novo espaço de memória.

void moveCursor(linha *atual, Cursor *cursor, char *direction): A função moveCursor é utilizada para mover o cursor em um editor de texto, dado uma direção especificada. Ela recebe uma estrutura linha como a linha atual em que o cursor está posicionado, uma estrutura Cursor para representar a posição atual do cursor, e uma string direction que indica a direção desejada.

A complexidade de tempo teórica é O(n) na pior das hipóteses, onde n é o tamanho da linha atual. Isto é devido ao uso do loop na verificação da direção de movimento do cursor. A complexidade de espaço é O(1), pois apenas um cursor é alocado dinamicamente.

void linhaAnterior(linha **cabeca, Cursor *cursor): A função linhaAnterior move o cursor para a linha anterior no arquivo. A função recebe uma referência para a cabeça da lista encadeada de linhas (linha **cabeca) e uma referência para a estrutura de cursor (Cursor *cursor). O primeiro passo da

função é verificar se o número da linha atual é zero, o que significa que o cursor já está na primeira linha e não pode mover-se para trás. Se isso for verdade, a função retorna imediatamente sem fazer nada. Se o cursor não estiver na primeira linha, a função decrementa o número da linha atual armazenado na estrutura de cursor. Em seguida, o código usa um loop para iterar através das linhas anteriores na lista encadeada até chegar à linha anterior. Finalmente, o índice do cursor é definido como o comprimento da linha anterior, colocando o cursor no final da linha. Quanto à complexidade de tempo, a função tem complexidade linear O(n) com relação ao número de linhas anteriores a serem percorridas. Embora a função possa iterar através de todas as linhas anteriores na lista, o tempo total gasto será proporcional ao número de linhas. Além disso, a complexidade de espaço é O(1), pois a função só precisa armazenar algumas variáveis locais e não há alocação dinâmica de memória.

void proximaLinha(linha *atual, Cursor *cursor): Esta função "proximaLinha" é usada para mover o cursor para a próxima linha no arquivo. Recebe como entrada a referência para a linha atual "atual" e o cursor "cursor". Antes de mover o cursor para a próxima linha, a função verifica se a linha atual tem uma próxima linha (atual->prox != NULL). Se não houver uma próxima linha, a função retorna sem fazer nada. Caso contrário, a função atualiza o cursor para apontar para a próxima linha. Para isso, ela atualiza o número da linha (cursor->numeroLinha++) e ajusta o índice do cursor para ser o menor entre o índice atual e o comprimento da linha (cursor->indice = MIN(cursor->indice, strlen(atual->texto))). A complexidade de tempo desta função é O(1), pois apenas uma atribuição e uma comparação são realizadas.

void juntarLinhas(linha *atual): Esta função junta duas linhas consecutivas em uma só. Ela começa verificando se existe uma linha subsequente à linha atual. Se sim, a função calcula o tamanho dos textos das duas linhas, aloca memória para o novo texto, que será a concatenação dos textos das duas linhas. Em seguida, copia o texto da linha atual e da linha subsequente para o novo texto, e atualiza a linha atual com este novo texto. A função também atualiza a lista ligada, removendo a linha subsequente e liberando sua memória. A complexidade de tempo desta função é O(n), onde n é o tamanho da concatenação dos textos das duas linhas. Isso é devido ao uso da função strncpy, que tem uma complexidade de tempo linear. A alocação de memória com malloc e a liberação de memória com free têm uma complexidade de tempo constantante O(1).

void separaLinha(linha *cabeca, Cursor *cursor): A função separaLinha é usada para separar uma linha em duas, no ponto indicado pelo cursor. A variável atual é inicializada com o valor de cabeca, e é usada para iterar através da lista encadeada de linhas até que a linha correta seja encontrada (usando um loop for). Aloca-se memória para a nova linha, usando malloc. Inicializa-se a nova linha, copiando a parte do texto da linha atual após o cursor para sua nova posição (atual->texto + cursor->indice). Atualiza-se os ponteiros da nova linha para apontar para o próximo elemento na lista (atual-

>prox) e para o elemento anterior (atual). A linha atual é atualizada, terminando-se seu texto na posição do cursor (atual->texto[cursor->indice] = '\0') e atualizando o ponteiro para a próxima linha (atual->prox = novaLinha). Finalmente, a posição do cursor é atualizada, incrementando-se seu número de linha e redefinindo-se seu índice para 0.

Em termos de complexidade de tempo, o loop for tem complexidade linear (O(n)), já que o número de iterações é igual ao número de linhas antes da linha correta. Alocar memória com malloc tem complexidade constante (O(1)). As operações de cópia de strings (strcpy e strlen) também têm complexidade linear (O(m)), onde m é o tamanho do texto da linha atual após o cursor. Portanto, a complexidade total desta função é O(n + m).

procurarPalavra(linha *cabeca, char *texto, Cursor *cursor): A função procurarPalavra procura por uma string específica (recebida como argumento como texto) na lista encadeada de linhas (recebida como argumento como cabeca). O cursor (recebido como argumento como cursor) é usado para armazenar a posição da palavra encontrada na lista encadeada de linhas. A variável numeroLinha é usada para armazenar o número da linha atual enquanto a lista encadeada de linhas é percorrida. A variável atual aponta para a linha atual na lista encadeada. O loop percorre a lista encadeada de linhas, começando da linha apontada por cabeca até chegar ao final da lista (atual é igual a NULL). Em cada iteração, a função strstr é chamada para procurar por texto a partir da posição do cursor (cursor->indice) na linha atual (atual->texto). Se a string é encontrada, os valores de cursor->numeroLinha e cursor->indice são atualizados com o número da linha e a posição da palavra encontrada, respectivamente, e a função retorna 1. Se a string não é encontrada, a variável atual é atualizada para apontar para a próxima linha na lista encadeada e o valor de cursor->indice é redefinido como 0. Se a string não é encontrada em nenhuma linha, a função retorna 0.

A complexidade de tempo da função é O(n * m), onde n é o número de linhas e m é o tamanho médio da string em cada linha. Isso ocorre porque o loop percorre todas as linhas da lista encadeada e a função strstr pode ter uma complexidade de tempo linear com o tamanho da string.

void substituirPalavra(linha *cabeca, char *antigo, char *novo, Cursor *cursor): A função substituirPalavra é uma função que procura uma palavra antiga em todo o texto e a substitui por uma nova palavra. A complexidade de tempo da função é O(n * m), onde n é o número de linhas e m é o número de caracteres em cada linha. Isso ocorre porque a função usa a função procurarPalavra para procurar a palavra antiga, e a função procurarPalavra tem uma complexidade de tempo O(n * m) porque percorre cada linha e compara cada caractere da linha com a palavra antiga. Além disso, a função substituirPalavra usa as funções deletarTexto e inserirTexto

para substituir a palavra antiga. Essas funções podem ter uma complexidade de tempo linear ou quadrática, dependendo de como a memória foi alocada.

A complexidade de espaço da função é O(m), onde m é o número de caracteres na nova palavra. Isso ocorre porque a função aloca memória para armazenar a nova palavra na memória.

void C(linha *cabeca, Cursor *cursor, Pilha **memoPilha): Ela começa pegando a linha atual a partir do cursor e armazenando a posição marcada (M) e final (cursor->indice) da seleção de texto. Em seguida, ela aloca memória suficiente para armazenar o texto selecionado e o copia para esse novo espaço de memória. Por fim, ela cria um novo elemento da pilha "novaPilha", armazenando nesse elemento o texto copiado e apontando para o elemento anterior da pilha (memoPilha).

A complexidade de tempo dessa função é O(n), onde n é o comprimento do texto selecionado. Isso é porque o loop principal que copia o texto selecionado é executado uma vez para cada caractere do texto selecionado. Além disso, a alocação de memória para o novo elemento da pilha e o texto copiado também consome tempo.

A complexidade de espaço é O(n), pois a função aloca memória para armazenar o texto selecionado, sendo que o tamanho desse espaço é diretamente proporcional ao comprimento do texto selecionado. Além disso, também há a alocação de memória para o novo elemento da pilha, que é independente do tamanho do texto selecionado.

void X(linha *cabeca, Cursor *cursor, Pilha **memoPilha): A função X armazena uma cópia da parte do texto da linha atual do cursor, compreendida entre os índices inicio (que é uma constante) e fim (que é a posição atual do cursor na linha), em uma estrutura de pilha, como um elemento. A estrutura de pilha é implementada como uma lista encadeada, com cada elemento da pilha representando um elemento da lista, armazenando o texto copiado como uma string. A complexidade de espaço desta função é O(n), onde n é o comprimento do texto armazenado na pilha. Isso ocorre porque cada vez que é adicionado um novo elemento na pilha, é necessário alocar espaço de memória para armazenar o texto.

A complexidade de tempo desta função é O(n^2), onde n é o comprimento do texto da linha atual. Isso ocorre porque é necessário realizar uma iteração ao longo do texto da linha para copiar o texto selecionado, e outra iteração para remover o texto da linha.

void V(linha *atual, Pilha *memoPilha, Cursor *cursor): A função V é uma função de colar (ou paste) na qual o texto que foi previamente cortado ou copiado é inserido na posição atual do cursor. A função começa verificando se há texto na pilha de memo (uma estrutura de dados usada para armazenar o texto que foi cortado ou copiado). Se houver, a função retira o elemento no topo da pilha e o insere na posição atual do cursor usando a função inserirTexto.

Depois disso, a memória alocada para armazenar esse texto na pilha é liberada.

A complexidade de tempo desta função é de O(n), onde n é o comprimento do texto colado. A complexidade de espaço depende da implementação da função inserirTexto e da pilha de memória, mas é geralmente linear com o tamanho do texto colado.

A complexidade de espaço é O(1), pois apenas uma estrutura de pilha é alocada dinamicamente. Além disso, é liberado o espaço ocupado pela estrutura de pilha após a operação de colar, mantendo a complexidade de espaço constante.

void mostrarPilha(Pilha *pilha): A função "mostrarPilha" imprime os elementos de uma pilha. Ela tem como entrada uma variável "pilha" do tipo Pilha que representa o topo da pilha.

O algoritmo percorre a pilha elemento por elemento utilizando um laço "while" que para quando "atual" for igual a "NULL", ou seja, quando chegar ao fim da pilha. Para cada elemento da pilha, a função imprime seu índice e o seu texto.

A complexidade temporal desta função é linear, ou seja, O(n), onde "n" é o número de elementos na pilha. Isto ocorre porque a função percorre toda a pilha uma única vez, imprimindo cada elemento. Portanto, a quantidade de tempo gasto pela função é diretamente proporcional ao número de elementos na pilha.

Conclusão

Ao término deste relatório, percebemos que as estruturas de dados são importantes para organizar e armazenar informações de maneira eficiente, permitindo acesso, busca, inserção e remoção de dados de forma rápida e eficiente. A lista encadeada é uma estrutura de dados importante que é amplamente utilizada em diversos aplicativos, incluindo editores de texto. Na implementação de uma lista encadeada em um editor de texto, é possível armazenar e manipular informações sobre o conteúdo do documento, incluindo caracteres, palavras e linhas. Além disso, a lista encadeada pode ser usada para implementar recursos como histórico de edição, desfazer e refazer ações e movimentação rápida pelo documento. Portanto, a implementação de uma lista encadeada é fundamental para garantir a eficiência e a funcionalidade de um editor de texto.