Mini Googlebot

Projeto PARTE 2 da disciplina Algoritmos e Estruturas de Dados I - Prof. Rudinei Goularte

Integrantes

Eduardo Vinícius Barbosa Rossi - 10716887

Lucas Ferreira de Almeida - 11262063

Compilador utilizado

Neste trabalho utilizamos o compilador GCC no Linux.

Instruções de uso do programa

- Primeiramente, tenha o compilador GCC instalado em seu computador.
- Execute o comando "make" no terminal de sua preferência para compilar o programa e gerar um executável.
- Execute o comando "make run" para executar o programa, após compilar.
    - Na tela inicial do programa aparecerá um menu com os comandos disponíveis para uso. Há um número que antecede cada um dos comandos; ao digitar este número e pressionar enter, o comando relativo àquele número será executado.
- Execute o comando "make clean" para limpar os arquivos gerados pela compilação do programa e pelo comando "make zip".
- Execute o comando "make zip" para criar um arquivo .zip contendo os diretórios e arquivos necessários para a compilação do programa.

Justificativas

Parte 1 do Projeto

Para organizar os sites do arquivo "googlebot.txt", decidimos utilizar a estrutura de dados **lista**. Era necessário inserir e remover elementos em qualquer posição, por isso não foi conveniente implementar a estrutura **pilha**, visto que a pilha apenas permite acesso a um elemento de uma das extremidades.

Decidimos implementar a lista como **simplesmente encadeada** principalmente por não sabermos a quantidade de elementos que seriam inseridos. Além disso, para inserir (ou remover) um elemento no meio da lista, não há a necessidade de mover os elementos posteriores.

Implementamos a inserção de modo que a lista fique sempre **ordenada** pois, como é dito no enunciado, os sites devem ser ordenados pela chave.

Foi utilizada a **busca sequencial otimizada** pois não é possível realizar a **busca binária** em listas simplesmente encadeadas.

Houveram algumas modificações nos TADs **item** e **lista**.

Mesmo sabendo que estamos trabalhando com sites nesse projeto, decidimos criar um TAD item mais genérico. Pensamos como desenvolvedores de um TAD: esse TAD item pode ser utilizado em diversas situações, então ele precisaria atender adequadamente qualquer tipo de dado, não só sites, mas livros, músicas e filmes, por exemplo. A solução foi adicionar uma propriedade "void *conteudo", que seria o ponteiro para uma estrutura, para um vetor, para uma string ou para o que o usuário desejar.

Isso implicou em alterações nas funções do TAD item e do TAD lista. Foi necessário acrescentar/alterar as seguintes funções ao TAD item:

- item_criar: foi adicionado um parâmetro para o conteúdo: `void *conteudo`. Neste programa em específico, o usuário pode passar um site neste parâmetro e armazená-lo no item.
- item_apagar: foi adicionado um parâmetro para que o conteúdo do item seja apagado adequadamente: `void (*apagar_conteudo) (void **conteudo)`. Caso o usuário tenha armazenado um site, por exemplo, deve escrever `&site_apagar` neste parâmetro.
- item_get_conteudo: ao passar o item, retorna o conteúdo daquele item. No caso de armazenar um site, retorna o endereço de memória do site.
- item_set_conteudo: é possível alterar o conteúdo de um item através do parâmetro `void *conteudo`.

E no TAD lista:

- lista_apagar: acrescentado parâmetro para apagar o conteúdo de cada um dos itens da lista (essa função executa item_apagar e precisa ter acesso a uma função `apagar_conteudo`). Exemplo de uso, com uma lista de sites: `lista_apagar(lista_de_sites, &site_apagar)`
- lista_imprimir_conteudo: a função lista_imprimir apenas imprime na saída padrão todas as chaves em sequência. Já a função lista_imprimir_conteudo, imprime não só a chave mas o conteúdo, e possui parâmetros adicionais: `FILE *output`, para que o usuário decida onde a lista será impressa; `void (*imprimir_conteudo)(void *, FILE *)`, para que o usuário imprima o conteúdo de cada item de maneira correta; `const char *format` para que o usuário decida como a chave de cada item será impressa. É uma função mais genérica que a lista_imprimir e foi necessária para gerar um arquivo de saída e verificar se o programa está sendo executado corretamente. Exemplo de uso, com uma lista de sites: `lista_imprimir_conteudo(lista_de_sites, stdout, &site_imprimir, "%04d\n")`

Além disso, no TAD item foram adicionadas duas funções: item_get_chave_erro e item_get_conteudo_erro, que retorna se houve erro ou não por referência. É útil caso o usuário queira fazer esse tipo de verificação e desalocar a memória alocada dinamicamente de maneira adequada antes de abortar a execução do programa.

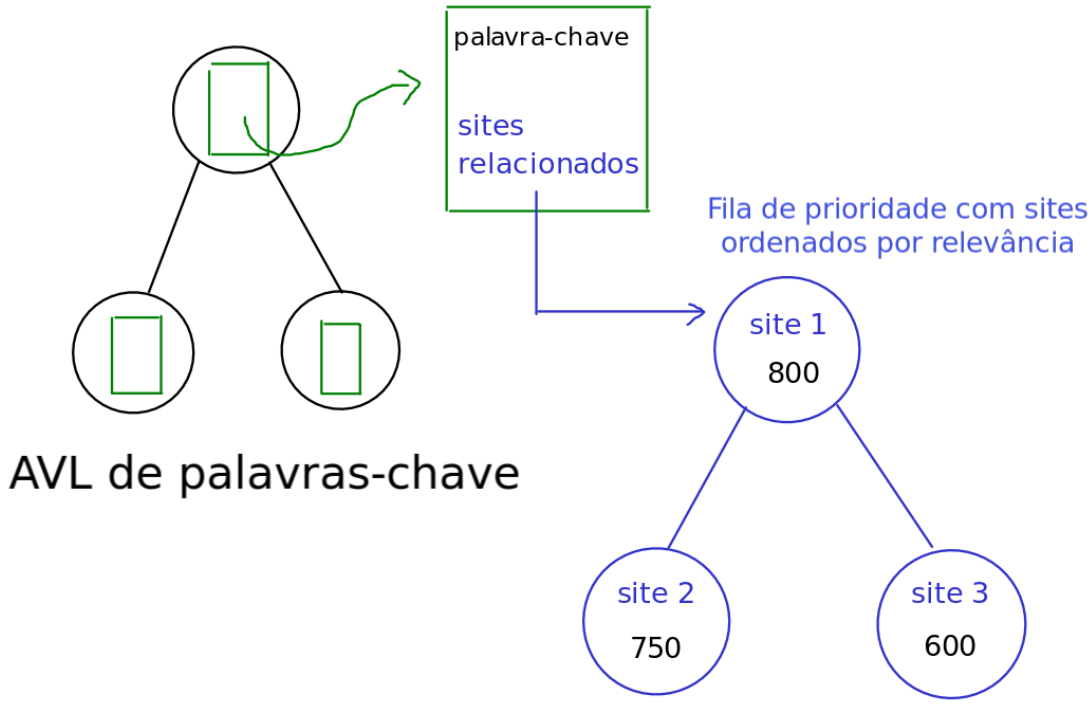
Parte 2 do Projeto

Para a parte 2, decidimos utilizar uma **AVL** de registros (`struct`). Cada registro contém uma palavra-chave e uma **Fila de Prioridade**.

A `avl_de_palavras_chave`, como decidimos chamar essa estrutura, contém todas as palavras-chave de cada um dos sites da lista de sites. Ela é ordenada pela ordem alfabética das palavras-chave.

Além disso, a fila de prioridade `sites_relacionados` de cada um dos registros da `avl_de_palavras_chave` armazena os sites que contém aquela palavra-chave. Como se trata de uma fila de prioridade, o "topo" contém o site mais relevante.

Segue um diagrama da estrutura:



A estrutura em azul é a fila de prioridade, com a relevância de cada site, a estrutura em verde é o registro de cada nó da avl, e a estrutura em preto é a avl de palavras-chave.

Nós escolhemos a **AVL** pois, ao inserir, remover e buscar as palavras-chave, temos uma baixa complexidade assintótica de tempo, já que se trata de uma árvore binária de busca que se mantém balanceada. A escolha da **Fila de prioridade** é para inserir sites em baixa complexidade assintótica de tempo e, principalmente, ter fácil acesso ao site de maior relevância para aquela determinada palavra-chave (O(1)), o que poderia ser mais custoso utilizando outra estrutura de dados.

Tanto a AVL quanto a Fila de prioridade foram implementadas de modo **encadeado**, pois não sabemos de antemão quantos sites e palavras-chave serão adicionados, então seria melhor adicioná-los dinamicamente.

Como o enunciado deixa claro que a **busca** é a operação mais **frequente** e não nos é passada uma restrição quanto à **memória**, optamos por investir em uma busca mais **otimizada**, ocupando mais memória que o necessário.

Segue uma tabela com a complexidade de tempo de cada operação relevante, sendo "n" a quantidade de palavras-chave e "s" a quantidades de sites em determinada palavra-chave.

Operação	**Complexidade**
Busca por palavra-chave	O(log n)
Acesso ao site mais relevante de palavra-chave	O(log n)
Inserção de site em palavra-chave	O(log n + log s)
Impressão dos sites mais relevantes de determinada palavra-chave	O(log n + s)

Comentários adicionais

Cada vez que o programa é executado, um arquivo "googlebot_out.txt" é gerado de acordo com as operações que foram executadas.

A parte 1 do projeto foi adaptada para o uso das novas estruturas.