Trabalho Prático 1
Escalonador de URLs

Luís Felipe Ramos Ferreira
2019022553

Universidade Federal de Minas Gerais
(UFMG) Belo Horizonte – MG – Brasil

lframos_ferreira@outlook.com

1. Introdução

A proposta do Trabalho Prático 1, da disciplina de Estruturas de Dados, foi implementar um programa para realizar o escalonamento de URLs. Esse programa de escalonamento tem como objetivo auxiliar um **coletor** de uma máquina de busca. O escalonador foi implementado de forma a organizar as URLs e seus hosts(sítios), de modo que o **coletor** possa coletar as URLs na ordem desejada. No caso deste trabalho, foi adotada a estratégia de coleta *depth-first* (busca em profundidade), que coleta todas as URLs de um host antes de passar para o próximo.

Especificamente para este Trabalho Prático, o programa foi feito para responder aos comandos presentes em um arquivo de texto que será passado na linha de comando. Os comandos que podem ser executados são os seguintes:

● ADD_URLS <quantidade> : adiciona ao escalonador as URLs informadas nas linhas seguintes. O parâmetro <quantidade> indica quantas linhas serão lidas antes do próximo comando.

● ESCALONA_TUDO: escalona todas as URLs seguindo as regras estabelecidas previamente. Quando escalonadas, as URLs são exibidas e removidas da lista.

● ESCALONA <quantidade> : limita a quantidade de URLs escalonadas.

● ESCALONA_HOST <host> <quantidade> : são escalonadas apenas URLs deste host.

● VER_HOST <host> : exibe todas as URLs do host, na ordem de prioridade.

● LISTA_HOSTS: exibe todos os hosts, seguindo a ordem em que foram conhecidos.

● LIMPA_HOST <host> : limpa a lista de URLs do host.

● LIMPA_TUDO: limpa todas as URLs, inclusive os hosts.

2. Implementação

O programa foi desenvolvido em um ambiente operacional Linux, na linguagem C++ compilada pelo compilador G++ da GNU Compiler Collection.

2.1. Estrutura de dados

O programa teve como base em sua implementação duas estruturas de dados, sendo elas Fila e Lista, ambas duplamente encadeadas. A escolha do encadeamento duplo foi feita devido à uma maior facilidade na remoção de elementos, procedimento que ocorre diversas vezes na execução do escalonamento de URLs armazenadas nas listas. A Fila, por sua vez, foi escolhida para ser utilizada no armazenamento de hosts, como especificado pelo enunciado do trabalho. As duas estruturas possuem muitas funcionalidades em comum. Por esse motivo, uma classe base chamada EstruturaEncadeada foi criada, e suas características herdadas pelas duas estruturas supracitadas.

As estruturas descritas acima foram implementadas como *template,* funcionalidade da linguagem de programação utilizada, a qual permite que os tipos aceitos pelas estruturas no armazenamento sejam dinâmicos.

Essas estruturas foram implementadas em classes, as quais possuem os atributos *head* (início da estrutura) e *tail* (final da estrutura), que são ponteiros para o tipo de dados que elas armazenam. Estes atributos servem para facilitar a inserção e remoção dos elementos das estruturas, além de simplificar a navegação por elas. Elas também possuem o número de elementos que possuem armazenado, uma vez que esta é uma informação importante para algumas operações.

Para facilitar a criação da Lista e da Fila, citadas acima, foi criado também uma s*truct* denominada Celula. Essa estrutura foi colocada como um *template,* funcionalidade comentada acima, a qual possibilitou dinamismo para os tipos que seriam armazenados em cada estrutura.

Especificamente para o cenário desse trabalho, URLs e os sítios (hosts) dessas URLs serão valores importantes para serem armazenados nas estruturas supracitadas. Para obter maior controle sobre como esses dados são armazenados e representados, duas *structs* auxiliares foram criadas, com os nomes URL e Sitio.

Por fim, tem-se também uma classe para o escalonador, que irá realizar todas as operações desejadas sobre as URLs e sítios (hosts). Essa classe possui métodos que representam cada um dos comandos já especificados.

2.2. Classes

Como já explicitado, para organizar e modularizar a implementação do trabalho, foram criadas 4 (três) classes e 3 (três) *structs*.

A classe EstruturaEncadeada<T> possui células do tipo dinâmico T, uma vez que foi declarada como um *template*, as quais representam a cabeça e a calda da estrutura. Ela também possui como atributo o número de elementos armazenados em si. Ambas as estruturas que herdam desta classe utilizam um duplo encadeamento, o que facilita a inserção, remoção e acesso de elementos. Os métodos da classe são aqueles que toda estrutura encadeada derivada deve possuir, como um método para

acesso do número de elementos, para checagem da existência de um elemento, para limpar a estrutura e para checar se está vazia.

A classe Lista<T> herda a classe EstruturaEncadeada e, assim, herda seus atributos e métodos. A classe Escalonador, que posteriormente será destacada, foi declarada como *friend* na classe Lista, uma vez que deverá ter acesso aos seus membros. A lista possui diversos métodos, dentre eles se destacam, para essa implementação, os métodos de inserção e remoção de itens.

A classe Fila<T>, assim como a Lista, herdou a classe EstruturaEncadeada e, assim, herdou seus atributos e métodos. Essa estrutura seguem um padrão LIFO (*Last In, First Out)*. A fila possui muitos métodos, dentre os quais se destacam os de enfileiramento e desenfileiramento de elementos.

A última classe, denominada Escalonador, é a classe que representa o objeto escalonador que realiza os comandos e operações desejados sobre as URLs. O seu atributo mais importante é a fila de sítios (hosts). Ela possui alguns métodos para leitura e identificação do comando que será realizado, a partir do arquivo. Cada comando, por sua vez, tem um método específico implementado na classe.

2.3. Structs

A primeira *Struct* implementada na solução do problema foi a estrutura Célula<T>, declarada como um *template*. Ela serve como nó para as estruturas encadeadas. Sua declaração como *template* teve como intuito tornar dinâmico os tipos que cada estrutura encadeada irá armazenar. Cada Celula<T> possui um valor do tipo T, assim como ponteiros para as células anterior e posterior à ela na estrutura. Seus métodos são apenas os construtores e o destrutor.

As duas *Structs* restantes são representativas para os tipos de dados que são armazenados nas estruturas encadeadas. A estrutura URL representa, como o nome indica, uma URL(*Uniform Resource Locator*). Ela possui como atributo uma string para representar a URL em si, assim como a quantidade de barras que a URL possui, informação relevante para a ordenação na hora de inserir as URLs nas listas. Além dos construtores e do destrutor, ela também possui um método “contaBarras,” para realizar contagem do número de barras.

Por fim, a *Struct* Sitio representa o sítio(host) de uma URL. Como atributo, ela possui uma string que representa o sítio(host) em si, assim como um ponteiro para a lista de URLs deste sítio. Seus métodos são apenas os construtores e o destrutor.

2.4. Escalonamento e execução dos comandos

O programa tem como objetivo principal, como supracitado, o escalonamento das URLs seguindo uma lógica de *depth first.* O armazenamento correto dos dados, principalmente o das URLs na ordem desejada, é o mais importante para o funcionamento do programa.

Para garantir esse funcionamento, uma instância da classe Escalonador é criada para executar o escalonamento das URLs. Esse escalonador possui como atributo a fila de sítios(hosts) conhecidos pelo escalonador. Dessa forma, se um sítio ainda não estiver presente na fila, ele é enfileirado.

Cada sítio enfileirado possui uma lista de URLs que contêm todas as URLs

conhecidas e válidas adicionadas pelo arquivo de entrada. As URLs são adicionadas já obedecendo a ordem de prioridade dada pela proximidade à raiz do sítio, evitando assim problemas com ordenamento. Como dito, apenas as URLs válidas são inseridas nas estruturas de dados.

Por fim, o escalonador possui um método destacado para a realização dos comandos contidos no arquivo de entrada. Cada comando descrito possui seu método específico implementado no escalonador e, quando tais comandos são lidos pelo escalonador, eles são executados. A comunicação entre o escalonador e o arquivo de entrada é feita por meio de um método de leitura de arquivo também implementado dentro da classe Escalonador.

3. Análise de Complexidade

3.1. Tempo

- **Método 'contaBarras' (struct URL):** esse método itera sobre o atributo mURL, uma string de n caracteres, em busca do caractere barra. Dessa forma, sua complexidade de tempo é O(n).

- **Função 'validaURL':** essa função irá utilizar a função '*regex_match*' da biblioteca <*regex*>, que possui uma complexidade O(n). Portanto, sua complexidade de tempo é O(n).

- **Função 'reduzURL':** essa função utiliza as funções '*find*' e '*erase*' da biblioteca <*string*>. Ambas funções possuem complexidade linear. Portanto, a complexidade de tempo é O(n).

- **Função URLparaSitio:** essa função utiliza a função '*getline*' da biblioteca <*string*>, cuja complexidade de tempo é linear. Portanto, a complexidade de tempo é O(n).

- **Método 'getElemento' (structs URL e Sitio):** esse método irá retornar o valor do elemento que define a estrutura que, nestes casos, é uma string. Sua complexidade é constante, isto é, O(1).

- **Método 'getNumElementos' (classe EstruturaEncadeada):** esse método retorna o valor do número de elementos da estrutura de dados. Como esse valor é incrementado e decrementado durante os métodos de inserção e remoção de elementos, sua complexidade é constante. Portanto, sua complexidade de tempo é O(1).

- **Método 'estaVazia' (classe EstruturaEncadeada:** esse método retorna a comparação de igualdade do atributo do número de alimentos com o número 0. Portanto, sua complexidade de tempo é constante, isto é, O(1).

- **Método 'limpa' (classe EstruturaEncadeada):** esse método itera por toda a estrutura encadeada, deletando cada um de seus itens. Supondo que a estrutura possui n elementos, a complexidade de tempo da função será linear, ou seja, O(n).

- **Método ‘contem’ (classe EstruturaEncadeada):** esse método itera pela estrutura encadeada em busca do elemento passado como argumento. Supondo que a estrutura possui n elementos, a complexidade de tempo da função será linear, isto é, O(n).

- **Método ‘enfileira’ (classe Fila):** esse método enfileira um novo elemento na fila, isto é, ao final dela. Como são realizadas apenas operações constantes, o método possui complexidade de tempo O(1).

- **Método ‘desenfileira’ (classe Fila):** esse método desenfileira um elemento da fila, isto é, retira um elemento do início da fila. Assim como o método ‘enfileira’, realiza apenas operações constantes, resultando em uma complexidade de tempo O(1).

- **Método ‘getSitio’ (classe Fila):** esse método itera pela fila em busca do elemento desejado. Supondo que a fila possui n elementos, sua complexidade de tempo será linear,ou seja, O(n).

- **Método ‘procuraPosInsercao’ (classe Lista):** esse método itera pela lista realizando operações em busca da posição de inserção desejada. Supondo que a lista possui n elementos, sua complexidade de tempo é linear, ou seja, O(n).

- **Método ‘insere’ (classe Lista):** esse método insere o elemento na posição desejada. Para tal, utiliza o método ‘procuraPosInsercao’, que possui complexidade linear. Logo, a complexidade de tempo é O(n).

- **Método ‘removeInicio’ (classe Lista):** esse método remove o primeiro elemento na lista. Como ela é encadeada, essa remoção é feita com operações constantes. Portanto, a complexidade de tempo é O(1).

- **Método ‘remove’ (classe Lista):** esse método itera pela lista em busca do elemento que deve ser removido. Supondo que a lista possui n elementos, a complexidade de tempo será linear, ou seja, O(n).

- **Método ‘leArquivo’ (classe Escalonador):** esse método utiliza a função ‘*getline*’ da biblioteca <*string*>, cuja complexidade de tempo é linear. Portanto, a complexidade de tempo é O(n).

- **Método ‘addURLs’ (classe Escalonador):** esse método percorre um laço de 0 até n. No melhor caso, nada é realizado dentro do laço, e sua complexidade é O(n). No pior caso, funções de complexidade linear O(m) são chamadas dentro do laço principal, resultando em uma complexidade de tempo quadrática, isto é, O(nm).

- **Método ‘escalonaTudo’ (classe Escalonador):** esse método itera pela fila de sítios, de tamanho n, e, para cada sítio, itera por sua lista de URLs, de tamanho m. Portanto, sua complexidade de tempo é quadrática, ou seja, O(nm).

- **Método ‘escalona’ (classe Escalonador):** esse método itera pela fila de sítios escalonando URLs até escalonar, no máximo, a quantidade n de URLs que a fila possui. Portanto, sua complexidade de tempo é linear, isto é, O(n).

- **Método ‘escalonaHost’ (classe Escalonador):** esse método itera pela lista de URLs de um sítio, até, no máximo, a quantidade n de elementos na lista. Logo, sua complexidade de tempo será linear, ou seja, O(n).

- **Método ‘verHost’ (classe Escalonador):** esse método itera pela lista de URLs do sítio, exibindo as URLs armazenadas. Supondo que a lista possui n elementos, sua complexidade de tempo é linear, ou seja, O(n).

- **Método ‘listaHosts’ (classe Escalonador):** esse método itera pela fila e exibe os sítios armazenados nela. Supondo que a fila possui n elementos, sua complexidade de tempo é linear, isto é, O(n).

- **Método ‘limpaHost’ (classe Escalonador):** esse método utiliza as funções ‘getSitio’, da classe Fila, e ‘limpa’, da classe EstruturaEncadeada. Como a função ‘limpa’ é chamada para uma estrutura encadeada de tipos URL, ambas possuem complexidade linear. Logo, a complexidade de tempo será O(n).

- **Método ‘limpaTudo’ (classe Escalonador):** esse método chama a função ‘limpa’, da classe EstruturaEncadeada. Como ela é chamada para uma estrutura que possui n elementos que armazena o tipo Sitio, o qual possui uma outra estrutura encadeada com m elementos como atributo, a complexidade de tempo será quadrática, isto é, O(nm).

- **Método ‘realizaComando’ (classe Escalonador):** esse método realiza um dos comandos descritos pelo trabalho, de acordo com o arquivo de entrada. No pior caso, um comando de complexidade quadrática é chamado, e a complexidade final será O(nm). No melhor caso, um comando de complexidade linear O(n) é chamado.

- **Função ‘getComando’:** essa função realiza operações constantes para retornar o valor inteiro que representa o comando passado como argumento. Portanto, sua complexidade de tempo é constante, ou seja, O(1).

3.2. Espaço

- **Método ‘contaBarras’ (struct URL):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função ‘validaURL’:** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função ‘reduzURL’:** essa função realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função URLparaSitio:** essa função realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘getElemento’ (structs URL e Sitio):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método 'getNumElementos' (classe EstruturaEncadeada):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘estaVazia’ (classe EstruturaEncadeada:** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘limpa’ (classe EstruturaEncadeada):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘contem’ (classe EstruturaEncadeada):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘enfileira’ (classe Fila):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘desenfileira’ (classe Fila):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘getSitio’ (classe Fila):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘procuraPosInsercao’ (classe Lista):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘insere’ (classe Lista):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘removeInicio’ (classe Lista):** esse método realiza todas as suas

operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘remove’ (classe Lista):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘leArquivo’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘addURLs’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘escalonaTudo’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘escalona’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘escalonaHost’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘verHost’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘listaHosts’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘limpaHost’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘limpaTudo’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Método ‘realizaComando’ (classe Escalonador):** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

- **Função ‘getComando’:** esse método realiza todas as suas operações sobre estruturas auxiliares constantes. Logo, sua complexidade de espaço é O(1).

4. Estratégias de robustez

Para tratar falhas e deixar o código mais robusto, foi utilizada a biblioteca ‘msgassert.h’, disponibilizada pelos professores. Essa biblioteca define duas ‘macros’ para tratar os possíveis erros encontrados no uso das funções do programa.

Ela foi utilizada em funções e métodos onde foi julgado necessário garantir que parâmetros de entrada fossem dados da forma esperada. Ela também foi utilizada para garantir que os arquivos utilizados na implementação fossem abertos corretamente.

É importante destacar também que nos métodos da classe Escalonador, que representam comandos a serem realizados, foi utilizado o recurso de condição *if*. Isso foi feito para evitar erros de *segmentation fault (core dumped)*, dentre outros erros comuns na linguagem C++, os quais poderiam ocorrer em casos específicos de entradas na realização dos comandos.

5. Conclusão

Esse trabalho prático teve como propósito criar um escalonador de URLs, isto é, um recurso que define a ordem de coleta das URLs, que iria auxiliar um coletor de uma máquina de busca. Uma abordagem orientada a objetos foi adotada e, por isso, a linguagem escolhida para o desenvolvimento da matriz e das operações sobre ela foi C++.

As estruturas encadeadas e os tipos de valores que são utilizados no desenvolvimento do programa foram implementados como classes e *structs,* com o intuito de obter controle acerca das operações realizadas sobre eles, assim como sobre os atributos que os definem. O escalonador, ponto principal do trabalho, também foi feito em uma classe, cujos métodos são, em sua maioria, os comandos especificados pelo enunciado.

Pode-se verificar que a solução adotada foi importante para que o trabalho funcionasse bem. A abordagem orientada a objetos permite, como dito, uma melhor manipulação do sistema. A utilização de *templates,* recurso disponibilizado pela linguagem C++, foi essencial para garantir um funcionamento dinâmico das estruturas implementadas.

Por meio da implementação do trabalho, foi possível trabalhar e enraizar conceitos importantes de programação orientada a objetos, como abstração e herança. Esses conceitos, assim como uma visão geral da programação orientada a objetos, foram revisados diversas vezes para que o trabalho pudesse ser concluído. Destaca-se aqui o uso de herança em classes que utilizam *templates.* Esse tipo de mecanismo é extremamente útil para diversas aplicações, mas exigiu muita refatoração para sua implementação correta e eficaz. Certamente, foi um dos maiores desafios encontrados durante a execução do trabalho.

6. Referências bibliográficas

GEEKSFORGEEKS. Core Dump (Segmentation fault) in C/C++.geeksforgeeks. Disponível em: <https://www.geeksforgeeks.org/core-dump-segmentation-fault-c-cpp/>. Acesso em: 24/11/2021.

CPLUSPLUS. String. Disponível em: <http://www.cplusplus.com/forum/general/266792/>. Acesso em: 25/11/2021.

MIZRAHI, Victorine Viviane. Treinamento em linguagem C. 2ª Edição. Editora Pearson.

DROZDEK, Adam. Estrutura de dados e algoritmos em c++. 4ªEdição. Editora Cengage Learning.

MODERNESCPP. Surprise Included: Inheritance and Member Functions of Class Templates. Disponível em: <https://www.modernescpp.com/index.php/surprise-included-inheritance-and-member-functions-of-class-templates>. Acesso em: 11/12/2021.

GEEKSFORGEEKS. The C++ Standard Template Library (STL).geeksforgeeks. Disponível em: <https://www.geeksforgeeks.org/the-c-standard-template-library-stl/>. Acesso em: 26/11/2021.

7. Compilação e execução

- Acesse o diretório TP
- No terminal, utilize comando *make* para compilar os arquivos e gerar o 'target' do programa.
- Para utilizar o programa, digite no terminal ‘bin/programa’ seguido do nome do arquivo de texto que contém os comandos que serão realizados.
- Para apagar os arquivos ‘object’ e o ‘target’, utilize no terminal o comando make clean.