Trabalho Prático 1 em Estruturas de Dados

Escalonador de URLs

Gustavo Freitas Cunha

Matrícula: 2020054498

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brazil

gustavocunha@dcc.ufmg.br

1. Introdução

A busca por informações na web atualmente integra aspectos fundamentais e de suma importância no cotidiano de corporações e pessoas. Tal busca, ao longo dos últimos anos, tem sido massivamente feita por meio de frases ou palavras-chave digitadas em ferramentas de buscas, como o Google. Essas ferramentas utilizam-se de algoritmos sofisticados e que envolvem muita ciência da computação, os quais começaram a surgir a partir da década de 1970. Neste trabalho, será desenvolvido um escalonador, um dos principais elementos de uma máquina de busca, junto com o coletor, que faz o download de conteúdos apontados por URLs. O escalonador auxilia a coleta de URLs na web, bem como define a ordem em que estes serão apontados. Por meio dos ferramentais de que dispõe a linguagem utilizada, bem como dos princípios de Programação Orientada a Objetos e utilização de estruturas de dados, será implementado um programa que coloca à disposição do usuário funções que o permitem controlar o funcionamento do escalonador mediante entradas providas por um coletor, através de um arquivo recebido como entrada.

2. Método

O programa foi desenvolvido e testado em sistema operacional Subsistema Windows para Linux (Windows Subsystem for Linux - WSL), na linguagem C++, compilado pelo compilador G++ da GNU Compiler Collection, em processador Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz, com 8GB de RAM instalada.

2.1 Organização do Código

Este projeto foi estruturado da seguinte forma:

```
- TP
  |- src
  |- bin
  |- obj
```

```
|- include
Makefile
```

O diretório `src` armazena os arquivos de código (extensão `.cpp`), o diretório `include` os arquivos de cabeçalho (extensão `.h`), o diretório `obj` armazena os arquivos objects (extensão `.o`) e o diretório `bin` armazena os executáveis (`.exe`).

Toda essa estruturação está configurada no Makefile disponível no diretório raiz.

Os diretórios `obj` e `bin` foram entregues vazios, conforme instrução, mas após compilação e linkagem pelo Makefile, os arquivos gerados são alocados em suas respectivas pastas.

2.2 Estrutura de Dados

Dentre as estruturas de dados modeladas para este trabalho, estão:

- **class Celula**: esta é a classe em nível hierárquico mais baixo. Classes como ListaEncadeada e FilaEncadeada irão ter uma relação de associação ("contém um") com Celula. Esta classe possui um item private do tipo T, definido por um **template**, e um apontador (também private) prox para um tipo Celula. Quanto aos métodos, possui apenas um construtor que inicializa prox com nullptr. Como foi dito, esta classe será usada em outras estruturas de dados com uma relação agregadora e, por isso, tais classes foram definidas como friend.

- **class ListaEncadeada**: esta classe possui atributos privados: tamanho, host, primeiro (ponteiro para Celula<string>, primeira célula da lista), e ultimo (ponteiro para Celula<string>, última célula da lista). Além disso, possui o método privado Posiciona, que retorna um ponteiro para a célula da lista na posição desejada. Dentre os métodos, estão: o construtor (ListaEncadeada) e destrutor (~ListaEncadeada), um getter para tamanho (GetTamanho), Vazia, que retorna valor booleano conforme estado da lista, GetItem, que retorna o item da célula na posição recebida como parâmetro, InsereInicio e InsereFinal, que dispensam explicações, InsereApos, que insere um item com parâmetro e o insere na posição imediatamente após uma posição recebida como parâmetro, RemovePosicao, que remove a célula na posição recebida como parâmetro, Pesquisa, que retorna valor booleano conforme resultado da pesquisa por item na lista, Imprime, que imprime os itens de todas as células da lista, ImprimeArquivo, que imprime os itens de todas as células da lista em um arquivo recebido como parâmetro e, finalmente, Limpa, que remove todas as células da lista, com exceção da **célula cabeça**. Por ser uma lista, inserções e remoções podem ocorrer em qualquer posição.

- **class FilaEncadeada:** Esta é uma **fila de listas**. Por ser uma fila, a remoção de itens apenas se dá na primeira posição válida (e válida é dito aqui porque há **célula cabeça**) e a inserção se dá apenas na última posição. A definição da

classe possui atributos privados: tamanho, frente (ponteiro para Celula<ListaEncadeada>, primeira célula da fila) e tras (ponteiro para Celula<ListaEncadeada>, última célula da fila). Dentre os métodos, estão: o construtor (FilaEncadeada), o destrutor (~FilaEncadeada), um getter para tamanho (GetTamanho), Vazia, que retorna valor booleano conforme estado da lista, Enfileira, que insere uma Célula<ListaEncadeada> recebida por parâmetro na última posição da célula, Desenfileira, que remove a primeira célula da lista e retorna seu item (uma ListaEncadeada), Limpa, que remove todas as células da fila, com exceção da **célula cabeça**, PesquisaHost, que retorna valor booleano conforme resultado da pesquisa por host recebido como parâmetro no item (ListaEncadeada) das células da fila e GetListaHost, que que retorna um ponteiro para a célula cujo item (ListaEncadeada) possui host igual ao host recebido como parâmetro.

- **class Escalonador:** esta classe possui uma FilaEncadeada, cujo item das células são do tipo ListaEncadeada. Conforme dito na descrição da classe ListaEncadeada, há um atributo host e, claro, a lista de URLs. Dentre os métodos, estão: o construtor (Escalonador); AdicionaUrls, que insere URLs do arquivo de entrada à fila do escalonador conforme critérios estabelecidos; EscalonaTudo, que escalona todas as URLs da fila, conforme critérios estabelecidos pela estratégia **depth-first**; Escalona, que escalona quantidade URLs (quantidade recebida como parâmetro); EscalonaHost, que escalona quantidade URLs de um host (host e quantidade recebidos como parâmetros); VerHost, que imprime todas as URLs de um host recebido como parâmetro; ListaHosts, que imprime todos os hosts que estão na fila do escalonador; LimpaHost, que remove todas as URLs da lista do host recebido como parâmetro; LimpaTudo, que remove todas as células da fila do escalonador, logo, remove todos os hosts e URLs; InsereUrl, que insere URL recebida como parâmetro em uma ListaEncadeada recebida como parâmetro conforme sua profundidade e, por fim, ImprimeTudo, que imprime todas as URLs na fila do escalonador.

2.3 Funcionamento do Programa

O programa principal é principal.cpp. É nele que está a função main e é feita a leitura e processamento dos arquivos passados na linha de comando, bem como a chamada das funções correspondentes. Para que o código pudesse ser mais reutilizável e modularizado, as funções operacionais sobre as estruturas de dados Celula, ListaEncadeada, FilaEncadeada e Escalonador, têm sua assinatura e especificação no respectivo programa .h e .cpp, sendo que a primeira delas apenas possui arquivo .h. Você pode fazer a execução do programa passando, na linha de comando, um arquivo[1] como parâmetro, que deverá conter os comandos, conforme a seguir:

[1] Se o arquivo não estiver no diretório em que o terminal estiver, é necessário indicar seu caminho.

ADD_URLS <quantidade>: adiciona ao escalonador as URLs informadas nas <quantidade> linhas seguintes.

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.

Após a execução e processamento, será gerado um arquivo de saída de mesmo nome e formato que o arquivo de entrada com o sufixo "-out".

No APÊNDICE A, serão dadas mais instruções sobre a compilação e execução do programa.

O programa principal, na função main, lê as linhas do arquivo e processa os comandos por meio da função processaComando, que possui um mecanismo de identificação e leitura dos comandos, bem como a chamada do método correspondente e apuração e processamento dos parâmetros por funções como processaHostQuantidade, processaHost, processaQuantidade, splitComando e validaComando.

2.3 Observações importantes

- Caso o arquivo de entrada, possua, no argumento quantidade, número maior do que o de URLs disponíveis na estrutura de dados correspondente, todas as URLs serão processadas.

- Podem parecer semelhantes, mas as funções de tratamento de host na especificação da classe do escalonador e no programa principal têm papéis diferentes: as primeiras, tratam e identificam os hosts **nas URLs** propriamente ditas, já as outras (no programa principal) identificam os hosts **em comandos** no arquivo de entrada, cujo argumento seja condizente.

3. Análise de Complexidade[2]

Conforme solicitado, a seguir serão analisadas as complexidades dos métodos que executam as principais funções do escalonador (classe Escalonador) e, claro, aqueles que são chamados por estes. Além disso, os principais métodos das estruturas de dados

[2] Complexidade de funções de bibliotecas importadas foram consideradas constantes, conforme orientação.

ListaEncadeada e FilaEncadeada também serão analisados. Ainda, no programa principal, serão analisadas as funções que fazem processamento de strings.

As análises que seguem levam em consideração três dimensões: o número de hosts na fila (denotado **m**), o número de URLs referentes ao host na respectiva lista (denotado **n**) e o tamanho da URL (denotado **p**). Em alguns casos, outras variáveis recebidas como parâmetro foram incluídas na notação assintótica. Em cada caso pertinente, estas variáveis estão descritas na coluna de comentários.

3.1 Complexidade de Tempo

class ListaEncadeada		
Método	**Complexidade**	**Comentário**
Construtor	O(1)	Apenas inicializa ponteiros.
Posiciona	Melhor Caso: O(1) Pior caso: O(n)	Caso a lista esteja vazia, tenha apenas um elemento ou a posição solicitada seja a primeira, a complexidade é constante, caso contrário, a complexidade é linear, conforme tamanho da lista.
GetItem	Melhor Caso: O(1) Pior caso: O(n)	Chama Posiciona. As outras operações têm complexidade constante
InsereInício	O(1)	Apenas manipulação de ponteiros.
InsereFinal	O(1)	Apenas manipulação de ponteiros.
InsereApos	Melhor Caso: O(1) Pior caso: O(n)	Chama Posiciona. De resto, apenas manipulação de ponteiros.
RemovePosicao	Melhor Caso: O(1) Pior caso: O(n)	Chama Posiciona. De resto, apenas manipulação de ponteiros.
Pesquisa	Melhor Caso: O(1) Pior caso: O(n)	Caso a lista esteja vazia, tenha apenas um elemento ou caso a URL pesquisada esteja na primeira posição, a complexidade é constante, caso contrário, a complexidade é linear, conforme tamanho da lista.
Imprime	O(n)	Imprime todos os elementos da lista em um laço que navega por todas as suas posições. Logo, a complexidade varia linearmente conforme o tamanho da lista.
ImprimeArquivo	O(n)	Imprime todos os elementos da lista em um arquivo, utilizando laço que navega por todas as suas posições. Logo, a complexidade varia linearmente conforme o tamanho da lista.
Limpa	O(n)	Navega pelas células da lista, excluindo-as. Por isso, a complexidade varia linearmente conforme o tamanho da lista.
Destrutor	O(n)	Chama Limpa.

Tabela 1 - Complexidade de tempo dos métodos da classe ListaEncadeada

class FilaEncadeada		
Método	**Complexidade**	**Comentário**
Construtor	O(1)	Apenas inicializa ponteiros.
Enfileira	O(1)	Manipulação de ponteiros para encadeamento na última posição da fila (O(1)) e atribuição à nova ListaEncadeada (a ser enfileirada) de todas as URLs devidas. Como em toda chamada desta função a lista a ser enfileirada está vazia, sua complexidade é também constante. Logo, a complexidade desta função como um todo é constante.
Desenfileira	O(1)	Apenas manipulação de ponteiros.
PesquisaHost	Melhor Caso: O(1) Pior caso: O(m)	Navega pelas células da fila verificando correspondência com o host procurado. Caso a fila esteja vazia, tenha apenas um elemento ou caso o host procurado esteja na primeira posição, a complexidade é constante (O(1)). Caso contrário, varia linearmente com o tamanho da fila (O(m)).
GetListaHost	Melhor Caso: O(1) Pior caso: O(m)	Navega pelas células da fila verificando correspondência com o host procurado, caso haja, um ponteiro para a respectiva célula é retornado. Caso a fila esteja vazia, tenha apenas um elemento ou caso o host procurado esteja na primeira posição, a complexidade é constante (O(1)). Caso contrário, varia linearmente com o tamanho da fila (O(m)).
Limpa	O(m)	Navega pelas células da fila, excluindo-as. Por isso, a complexidade varia linearmente conforme o tamanho da fila (m).
Destrutor	O(m)	Chama Limpa.

Tabela 2 - Complexidade de tempo dos métodos da classe FilaEncadeada

class Escalonador		
Método	**Complexidade**	**Comentário**
Construtor	O(1)	Apenas inicializa ponteiros.
validaUrl	O(1)	Condicionais encadeados para verificação de validade da URL, os blocos de código dos condicionais apenas retornam valor booleano - O(1) - (exceto um deles, que convoca função de biblioteca específica, que será considerada O(1)).

identificaHost	O(1)	Utiliza funções da biblioteca string para manipulações da URL recebida para reconhecimento do host. Conforme orientação, essas funções terão complexidade considerada constante - O(1). Sendo assim, a função como um todo também tem complexidade constante.
refinaUrl	O(1)	Utiliza funções da biblioteca string para manipulações da URL recebida para descartar partes indesejadas, conforme especificação. Conforme orientação, essas funções terão complexidade considerada constante - O(1). Sendo assim, a função como um todo também tem complexidade constante.
calculaProfundidade	O(p)	Navega pela URL contando o número de caracteres "/". Por isso, a complexidade varia linearmente conforme o tamanho da URL, aqui denotado p.
InsereUrl	Melhor Caso: O(1) Pior caso: O(n)	Essa inserção pode ter complexidade constante caso a lista esteja vazia (inserção feita pelo método InsereInicio da classe ListaEncadeada - O(1)) ou pode, no pior caso, ter complexidade O(n), quando a URL for inserida ao final da lista (método InsereApos - Melhor Caso: O(1); Pior caso: O(n) -, da classe ListaEncadeada).
AdicionaUrls	Melhor Caso: O(q) Pior caso: O(q*max (m,n))	Esta função possui um laço para a leitura de q linhas (quantidade q recebida como parâmetro) do arquivo de entrada. Dentro deste laço, há alguns condicionais que chamam funções O(1) para verificação da validade da URL e tratamento de caracteres como '\r' e '\n', uma chamada para identificaHost (O(1)) e uma chamada para refinaUrl (O(1)). Depois desses pré-processamentos, a complexidade é condicional, conforme a situação da fila: caso já haja um host referente a URL a ser adicionada - e esta condição é verificada pela função PesquisaHost da classe FilaEncadeada (Melhor Caso: O(1); Pior caso: O(m)) -, a função GetListaHost da classe FilaEncadeada é chamada (Melhor Caso: O(1); Pior caso: O(m)) como também InsereUrl ((Melhor Caso: O(1); Pior caso: O(n)), ficando, assim, a complexidade no melhor caso O(1) e, no pior caso, max(O(m), O(n)). Agora, caso a lista não tenha um host referente a URL a ser adicionada, os métodos InsereUrl ((Melhor Caso: O(1); Pior caso: O(n)) e Enfileira (O(1)), da classe FilaEncadeada são chamados. Ficando, assim, a complexidade no melhor caso O(1) e, no pior caso, O(n).

EscalonaTudo	O(mn)	Este método possui uma medida de robustez no início - O(1), e depois, um laço que navega por todas as posições da fila - O(m) e, para cada posição, os métodos da classe ListaEncadeada ImprimeArquivo (O(n)) e Limpa(O(n)) são chamados. Logo, a complexidade deste método é O(m)*O(n) = O(mn).
Escalona	O(q)	Este método possui uma medida de robustez no início - O(1) e depois um laço while o qual navega pelas células da fila e que, internamente, conforme o tamanho da lista, ou escalona todas as URLs da lista, ou escalona uma quantidade q recebida como parâmetro. Logo, a complexidade deste método varia linearmente conforme q.
EscalonaHost	Melhor caso: O(q) Pior caso: O(max(m, qn))	Este método possui uma medida de robustez no início - O(1), uma chamada para GetListaHost - Melhor Caso: O(1); Pior caso: O(m), e, em seguida condicionais que verificam a relação entre o tamanho da lista e quantidade q de URLS a serem escalonadas recebida como parâmetro. Seja t o tamanho da lista. Caso t>q, um laço while é executado q vezes, chamando a função RemovePosicao - Melhor Caso: O(1); Pior caso: O(n). Caso contrário, todas as URLs da lista são escalonadas, com os métodos ImprimeArquivo - O(n) e Limpa - O(n).
VerHost	Melhor caso: O(n) Pior caso: O(max(m, n))	Este método possui uma medida de robustez no início - O(1), uma chamada para GetListaHost - Melhor Caso: O(1); Pior caso: O(m), e, em seguida, um laço que navega por todas as posições da lista - O(n).
ListaHosts	O(m)	Este método possui uma medida de robustez no início - O(1) e, em seguida, um laço que navega por todas as posições da fila, imprimindo os hosts - O(m).
LimpaHost	Melhor caso: O(n) Pior caso: O(max(m, n))	Este método possui uma medida de robustez no início - O(1), uma chamada para GetListaHost - Melhor Caso: O(1); Pior caso: O(m), e, em seguida, uma chamada para Limpa - O(n).
LimpaTudo	O(m)	Chama Limpa da classe FilaEncadeada - O(m).
ImprimeTudo	O(mn)	Este método possui uma medida de robustez no início - O(1), e, em seguida, uma laço que navega por todas as posições da fila - O(m), que, internamente, chama o método Imprime, da classe ListaEncadeada, que é O(n). Então ficamos com O(mn).

Tabela 3 - Complexidade de tempo dos métodos da classe Escalonador

Programa principal		
Método	**Complexidade**	**Comentário**
processaHostQuantidade	O(1)	Essa função utiliza métodos da biblioteca string para reconhecer host e quantidade em comandos do arquivo de entrada cujos argumentos sejam compatíveis. Conforme orientação, esses métodos serão considerados O(1) e, então, a função, no geral, também será O(1).
processaHost	O(1)	Essa função utiliza métodos da biblioteca string para reconhecer o host em comandos do arquivo de entrada cujos argumentos sejam compatíveis. Conforme orientação, esses métodos serão considerados O(1) e, então, a função, no geral, também será O(1).
processaQuantidade	O(1)	Essa função utiliza métodos da biblioteca string para reconhecer a quantidade em comandos do arquivo de entrada cujos argumentos sejam compatíveis. Conforme orientação, esses métodos serão considerados O(1) e, então, a função, no geral, também será O(1).
splitComando	O(1)	Essa função utiliza métodos da biblioteca string para desconsiderar possíveis argumentos em um comando do arquivo de entrada. Conforme orientação, esses métodos serão considerados O(1) e, então, a função, no geral, também será O(1).
validaComando	O(1)	Condicionais aninhados para verificação de padrão dos comandos utilizando comparações booleanas. Caso um dos condicionais seja satisfeito, é retornado true. Caso contrário, false é retornado - O(1).

Tabela 4 - Complexidade de tempo das funções de processamento de strings do programa principal

3.2 Complexidade de Espaço

class ListaEncadeada		
Método	**Complexidade**	**Comentário**
Construtor	O(1)	Alocação dinâmica de uma variável do tipo Celula, apenas.
Posiciona	O(1)	Apenas declaração de ponteiros e variáveis auxiliares.
GetItem	O(1)	Apenas declaração de ponteiros - O(1) - e chamada para Posiciona, que é O(1).
InsereInício	O(1)	Declaração e alocação de 1 ponteiro, apenas.

InsereFinal	O(1)	Declaração e alocação de 1 ponteiro, apenas.
InsereApos	O(1)	Apenas declaração de ponteiros - O(1) , alocação dinâmica de uma variável do tipo Celula - O(1), e chamada para Posiciona, que é O(1).
RemovePosicao	O(1)	Declaração de ponteiros e variáveis auxiliares - O(1) - e chamada para Posiciona, que é O(1).
Pesquisa	O(1)	Apenas declaração de um ponteiro.
Imprime	O(1)	Apenas declaração de um ponteiro.
ImprimeArquivo	O(1)	Apenas declaração de um ponteiro.
Limpa	O(1)	Apenas declaração de um ponteiro.
Destrutor	O(1)	Chama Limpa, que é O(1).

Tabela 5 - Complexidade de espaço dos métodos da classe ListaEncadeada

class FilaEncadeada		
Método	**Complexidade**	**Comentário**
Construtor	O(1)	Alocação dinâmica de uma variável do tipo Celula, apenas.
Enfileira	O(1)	Declaração de ponteiros - O(1). Alocação dinâmica de uma variável do tipo Celula - O(1). Uma chamada (pois, nesta função, a lista recebida como parâmetro sempre tem um e apenas um elemento) para ListaEncadeada::InsereFinal - O(1).
Desenfileira	O(1)	Declaração de ponteiros e variáveis auxiliares.
PesquisaHost	O(1)	Apenas declaração de ponteiros.
GetListaHost	O(1)	Apenas declaração de ponteiros.
Limpa	O(1)	Apenas declaração de ponteiros.
Destrutor	O(1)	Chama Limpa, que é O(1).

Tabela 6 - Complexidade de espaço dos métodos da classe FilaEncadeada

class Escalonador		
Método	**Complexidade**	**Comentário**
Construtor	O(1)	Alocação dinâmica de uma variável do tipo FilaEncadeada, apenas.

validaUrl	O(1)	Não há declarações ou alocações, apenas chamadas para funções da biblioteca string. Conforme orientação, essas funções terão complexidade considerada constante - O(1). Sendo assim, a função como um todo também tem complexidade constante.
identificaHost	O(1)	Declaração de variáveis auxiliares - O(1) e chamadas para funções da biblioteca string. Conforme orientação, essas funções terão complexidade considerada constante - O(1). Sendo assim, a função como um todo também tem complexidade constante.
refinaUrl	O(1)	Declaração de variáveis auxiliares - O(1) e chamadas para funções da biblioteca string. Conforme orientação, essas funções terão complexidade considerada constante - O(1). Sendo assim, a função como um todo também tem complexidade constante.
calculaProfundidade	O(1)	Declaração de variáveis auxiliares - O(1) e chamadas para funções da biblioteca string. Conforme orientação, essas funções terão complexidade considerada constante - O(1). Sendo assim, a função como um todo também tem complexidade constante.
InsereUrl	O(1)	Declaração de ponteiros e variáveis auxiliares - O(1). Chamadas para calculaProfundidade - O(1), ListaEncadeada::InsereApos - O(1) e ListaEncadeada::InsereInicio - O(1). Logo, temos O(1).
AdicionaUrls	O(q)	Declaração de variáveis auxiliares e ponteiros - O(1). Dentro de um laço, executado **q** vezes (q recebido como parâmetro), há , condicionalmente, ou declaração de uma ListaEncadeada (alocação estática) - O(1) ou declaração de um ponteiro - O(1). Além disso, ainda no laço, há chamadas para algumas funções, todas O(1). Logo, O laço é O(q). Assim, ficamos com O(1) + O(q) = O(q).
EscalonaTudo	O(1)	Declaração de ponteiros - O(1). Chamadas para funções, todas O(1).
Escalona	O(1)	Declaração de ponteiros e variáveis auxiliares - O(1). Chamadas para funções, todas O(1).
EscalonaHost	O(1)	Declaração de ponteiros e variáveis auxiliares - O(1). Chamadas para funções, todas O(1).
VerHost	O(1)	Apenas declaração de ponteiros.
ListaHosts	O(1)	Apenas declaração de ponteiros.

LimpaHost	O(1)	Declaração de ponteiros - O(1). Chamada para ListaEncadeada::Limpa - O(1).
LimpaTudo	O(1)	Chamada para FilaEncadeada::Limpa - O(1)
ImprimeTudo	O(1)	Declaração de ponteiros e variáveis auxiliares - O(1). Chamadas para ListaEncadeada::Imprime - O(1).

Tabela 7 - Complexidade de espaço dos métodos da classe Escalonador

Programa principal		
Método	**Complexidade**	**Comentário**
processaHostQuantidade	O(1)	Declaração de variáveis auxiliares - O(1). Chamada de métodos da biblioteca string, considerados O(1), conforme orientação.
processaHost	O(1)	Declaração de variáveis auxiliares - O(1). Chamada de métodos da biblioteca string, considerados O(1), conforme orientação.
processaQuantidade	O(1)	Declaração de variáveis auxiliares - O(1). Chamada de métodos da biblioteca string, considerados O(1), conforme orientação.
splitComando	O(1)	Apenas chamada de métodos da biblioteca string, considerados O(1), conforme orientação.
validaComando	O(1)	Apenas chamada de métodos da biblioteca string, considerados O(1), conforme orientação.

Tabela 8 - Complexidade de espaço das funções de processamento de strings do programa principal

4. Estratégias de Robustez

Como estratégias de robustez foram utilizadas as macros definidas em `mgassert.h`. Dentre as macro definidas, temos:

- `avisoAssert`: recebe uma expressão lógica e uma mensagem. Quando a expressão lógica falha, é chamado o método `__avisoAssert` e enviadas informações sobre a linha e o arquivo em que foi detectada a falha.
- `__avisoAssert`: recebe uma expressão lógica, uma mensagem e o arquivo e linha em que a falha foi detectada. Exibe a mensagem e aborta o programa. Essa macro foi utilizada em possíveis erros com os quais a execução do programa ficaria comprometida.
- `erroAssert`: recebe uma expressão lógica e uma mensagem. Quando a expressão lógica falha, é chamado o método `__erroAssert` e enviadas informações sobre a linha e o arquivo em que foi detectada a falha.
- `__erroAssert`: recebe uma expressão lógica, uma mensagem e o arquivo e

linha em que a falha foi detectada. Exibe a mensagem, mas não aborta o programa. Essa macro foi utilizada em possíveis erros com os quais a execução do programa segue normalmente.

Em algumas funções também foram implementados condicionais que imprimem um aviso de erro personalizado e abortam o programa ou advertem sobre a exceção, para que o usuário faça o devido tratamento.

Tais mecanismos foram implementados com o fito de tornar o programa mais interativo e dinâmico, além de facilitar o reconhecimento de falhas das mais diversas naturezas: alocação de memória, abertura de arquivo, verificação de entradas válidas, entre outros.

5. Análise Experimental

Utilizando a biblioteca `memlog`, foram distribuídas pelo código chamadas de suas funções `leMemLog` e `escreveMemLog`, as quais possibilitam registro do endereço, identificação e tamanho dos dados lidos ou escritos, respectivamente, em uma dada porção de memória. No programa principal, há as chamadas para inicialização e finalização desse registro (se solicitado pelo usuário, conforme especificado no APÊNDICE A). Essas funcionalidades permitiram as análises e experimentos de desempenho computacional e eficiência de acesso à memória, cujos resultados serão apresentados nas duas próximas subseções.

5.1 Desempenho Computacional

Foram realizados alguns experimentos com a implementação deste projeto, com o objetivo de avaliar seu desempenho e funcionalidade mediante variação considerável do número de hosts, número de URLs por host e tamanho das URLs. A variância de URLs por host e da profundidade das URLs foi fixada em média 5. Os arquivos usados para testes foram gerados pelo gerador de carga disponibilizado. Os resultados são apresentados na tabela a seguir e plotados no gráfico da Figura 1.

Número de hosts	Número de URLs por hosts	Tempo de execução em segundos
100	100	3,6811098
100	200	11,0641172
100	300	25,093938
200	100	9,2012011
200	200	26,6663863
200	300	48,53016
300	100	12,7243655
300	200	42,9560451
300	300	87,3053665

Tabela 9 - Desempenho computacional do programa mediante escalagem das dimensões avaliadas

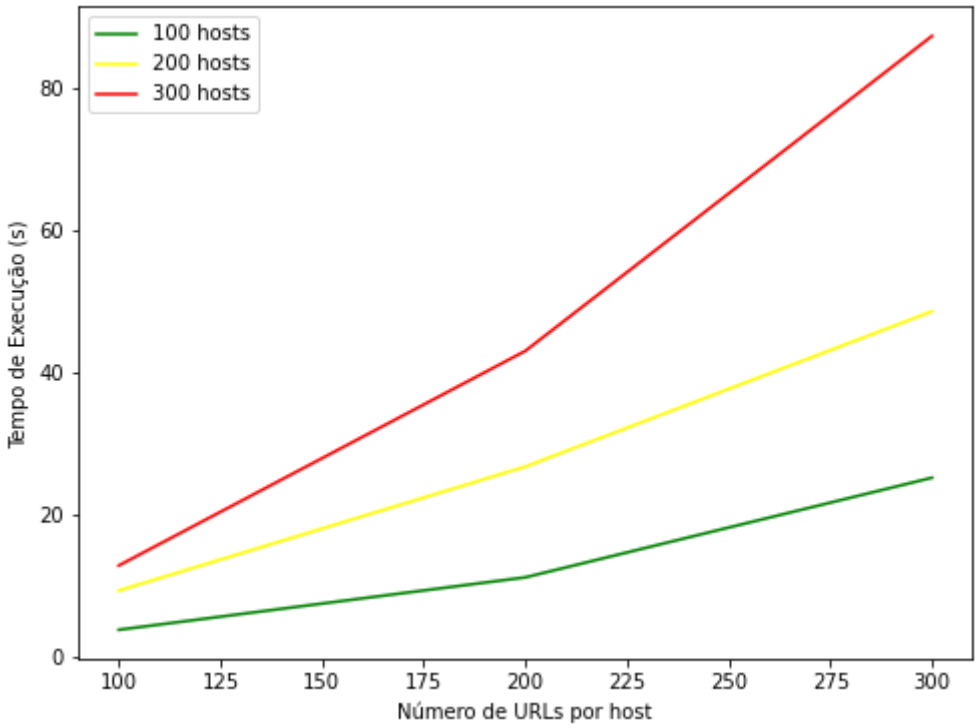

Figura 1 - Gráfico: Desempenho computacional do programa mediante escala das dimensões avaliadas

Os dados gerados estão sujeitos à margem de erro devido à execução inevitável de outros processos do computador e suas características de processamento, mas dão uma boa noção de como ocorre a variação do tempo de execução conforme se aumentam as dimensões avaliadas (número de hosts, número de URLs por host e tamanho das URLs).

5.2 Eficiência de Acesso à Memória

Nesta subseção será analisado como a execução do programa se comporta ao acessar as porções de memória demandadas e o quão eficiente é esse acesso. As análises foram feitas com arquivos de entradas gerados pelo gerador de carga com número de hosts fixado em 5 e número de URLs por host também fixado em 5. As dimensões menores se devem à busca por maior precisão e visualização dos mapas de acesso à memória apresentados, mas refletem, também, a configuração para dimensões maiores.

5.2.1 Padrão de Acesso e Localidade de Referência

Os experimentos realizados nesta subseção foram em fase única, com as especificações descritas anteriormente. Os resultados são apresentados em forma gráfica a seguir.

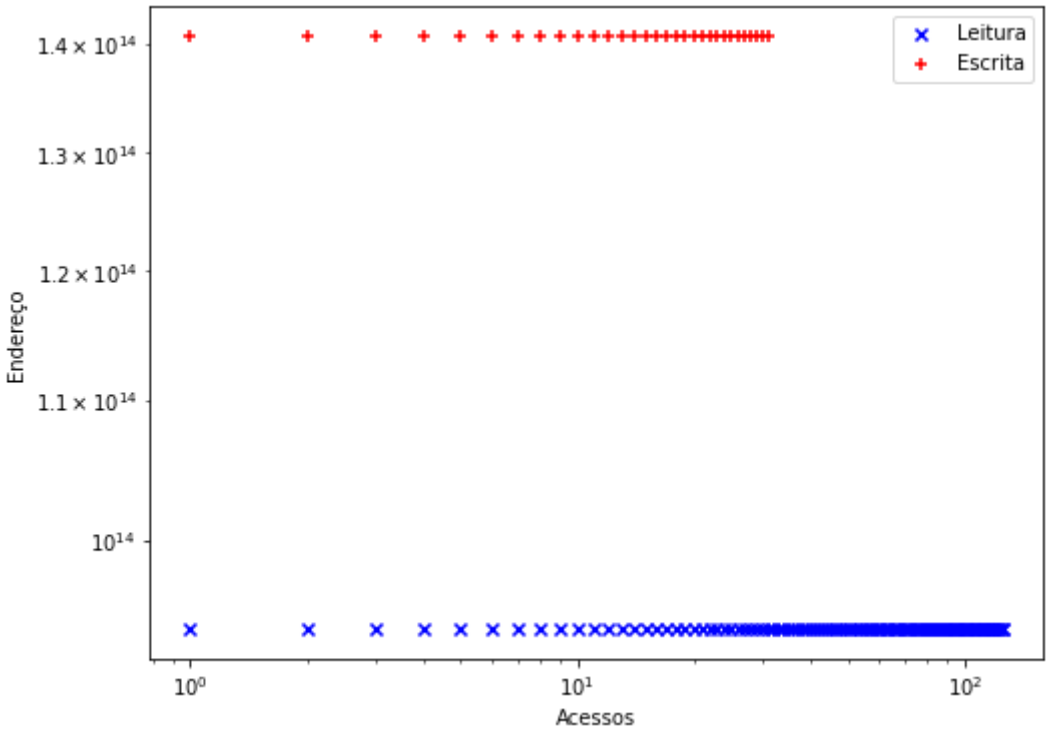

Figura 2 - Gráfico: Acessos x Endereço à memória durante a execução do programa

Note que são feitos muito mais acessos de leitura do que de escrita. Isso se deve às diversas consultas e caminhamentos feitos na fila do escalonador para adicionar novas URLs, de forma que, a cada inserção, enquanto uma escrita na memória é feita, n leituras são feitas, pois as leituras são necessárias para verificação de onde será feita a inserção da nova URL, conforme os critérios estabelecidos.

Ainda, pode-se perceber que, no início da execução, as posições de memória acessadas estão mais distantes, I.E., como maior **distância de pilha**, mas, conforme novas inserções vão sendo realizadas, mais próximas ficam as porções de memória acessadas.

Esse comportamento se deve à maneira como são armazenadas as URLs, que vão se aglomerando ao longo do tempo.

6. Conclusão

Este trabalho teve como objetivo a implementação de escalonador de uma máquina de busca, utilizando os diversos recursos inerentes e oriundos da ciência da computação e lógica de programação. A aprendizagem no processo de desenvolvimento me permitiu vislumbrar aspectos que desconhecia, com relação às máquinas de busca e conhecer um pouco mais de sua história e linha do tempo, apesar de ter contato com esse tipo de ferramenta diariamente. Além do colocado, a utilização de estruturas de dados como Celula, ListaEncadeada, FilaEncadeada e Escalonador foi rica e certamente foi de bom proveito neste trabalho e será em outros, já que é amplamente diversa.

7. Bibliografia

CORMEN , T., LEISERSON, C, RIVEST R., STEIN, C. **Introduction to Algorithms**, Third Edition, MIT Press, 2009.

CHAIMOWICZ, L. and PRATES, R. **Slides virtuais da disciplina de estruturas de dados**, 2020.

Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Class std::string. **C Plus Plus**, 2000.

Disponível em: <https://www.cplusplus.com/reference/string/string/>.

Acesso em: 11 de dezembro de 2021.

Class std::fstream. **C Plus Plus**, 2000.

Disponível em: <https://www.cplusplus.com/reference/fstream/fstream/>.

Acesso em: 11 de dezembro de 2021.

Como converter string em int em C++. **DelftStack**, 2020.

Disponível em: <https://www.delftstack.com/pt/howto/cpp/how-to-convert-string-to-int-in-cpp>.

Acesso em: 11 de dezembro de 2021.

APÊNDICE A
Instruções de Compilação e Execução

Com a utilização do Makefile, a compilação e linkagem do programa se dão de maneira muito simples:

Colocando o seu terminal no diretório raiz do projeto, basta que você digite `make` e o programa será compilado e linkado.

A execução se dá colocando, com seu terminal no diretório raiz do projeto e com o programa compilado, digitando:

```
bin/principal <caminho e nome do arquivo de entrada[3]>
```

Caso o usuário queira fazer uso da análise de log, deve seguir **estritamente** as instruções a seguir:

- Apenas contar o tempo de execução:

  Com o terminal no diretório raiz do projeto e com o programa compilado, digitar[4]:

  ```
  bin/principal <caminho e nome do arquivo de entrada>
   <endereco de destino desejado e nome do arquivo de registro>
  ```

- Contar o tempo de execução, padrão de acesso e localidade:

  ```
  bin/principal <caminho e nome do arquivo de entrada>
   <endereco de destino desejado e nome do arquivo de registro> -l
  ```

Por fim, é possível excluir, caso se queira, os arquivos objects (`.o`) gerados com a compilação e, ainda, o arquivo executável, entrando com o comando `make clean` no terminal, quando ele estiver no diretório raiz do projeto.

Você poderá acompanhar a execução por meio das impressões feitas no terminal.

---

[3] Arquivo com os respectivos comandos e URLs advindas do coletor.
Caso o arquivo de entrada esteja no mesmo diretório atual do terminal, não é preciso indicar seu caminho.

[4] A quebra de linha não é necessária no terminal. Foi feita, nesse caso, apenas para melhor visualização.