Trabalho Prático 1 - Escalonador de URLs

Raissa Miranda Maciel
20 de Dezembro de 2021
Matrícula: 2020006965

1.0 Introdução

Esta documentação lida com o desafio de desenvolver um escalonador para definir a ordem que as páginas apontadas pelas URLs serão coletadas. Esse algoritmo é muito utilizado em máquinas de busca que necessitam de uma estratégia para a coleta dos dados. Nesse contexto, temos os *hosts*, que de maneira geral, possuem várias URLs com caminhos diferentes, mas que compartilham do mesmo *host* principal. Sendo assim, o escalonador desenvolvido adota a estratégia *depth-first*, que coleta todas as URLs de um host antes de passar para o próximo, isto é, uma busca em profundidade.

O escalonador deve funcionar apenas para sítios com protocolo HTTP e , sua implementação conta com uma fila de hosts, os quais possuem cada um uma lista de URLs. Os *hosts* são caracterizados pelo nome do sítio sem o prefixo *http://* e *www.* e as URLs são formatadas sem o *www.*, sem */* e fragmentos ao final. A inserção de hosts na fila prioriza a ordem de conhecimento deles e a inserção de URLs na lista prioriza a profundidade delas, ou seja, aquelas com maior quantidade de diretórios são consideradas mais profundas.

Por fim, a coleta de dados para a execução do programa é feita por um arquivo de entrada cujo nome é dado como argumento na linha de comando. Ele deve possuir determinadas instruções que serão realizadas com as URLs, como a visualização da fila de *hosts*, da lista de URLs ou a remoção de todas as URLs de um *host*), que serão detalhadas nas próximas seções deste documento.

As seções estão divididas em conteúdos especificados a seguir. A seção 2.0 explica a estrutura de dados, os tipos abstratos de dados e as funções utilizadas, além de justificar a escolha delas. A seção 3.0 possui a análise de complexidade de tempo e espaço dos principais procedimentos implementados. A seção 4.0 mostra as estratégias de robustez usadas para prover a tolerância a falhas. A seção 5.0 conclui e sumariza o aprendizado com o trabalho. A seção 6.0 contém a bibliografia utilizada. E as instruções para compilação e execução estão em um apêndice ao fim do documento.

2.0 Implementação

Nesta seção serão descritas todas as estruturas de dados utilizadas na composição do programa, assim como a configuração utilizada para os testes.

2.1 Tipos abstratos e funções

A primeira classe é a URL e seu *header file* e implementação estão nos arquivos *URL.hpp* e *URL.cpp*. Uma URL é caracterizada por um nome, um ponteiro para a próxima URL, uma vez que ela pertencerá a uma lista encadeada e uma profundidade, que estão declaradas em sua parte *private*. Os métodos relacionados a uma URL são, primeiramente, o construtor, que verifica se a URL possui o protocolo HTTP (por meio dos métodos URL_certa e URL_valida) antes de criar o objeto, e o destrutor padrão. O método URL_certa também verifica se há sufixos inadequados (.jpg, .gif, .mp3, .avi. .doc, .pdf) e não cria URLs desse formato. Além disso, foram implementados os

métodos *get* e *set* para acessar os membros *privates* e atribuir novos valores a eles. Os métodos de formatação recebem a string da URL completa e retira as partes *www.* , / e fragmento ao final da URL. A formatação para o nome de um host é mais específica e retira também o protocolo *http://* e todos os diretórios presentes, implementado no método *FormataPraHost*. O método CalculaProfundidade retorna o número de diretórios presentes na URL, imprescindível para descobrir o local adequado em que a URL será adicionada na lista. O operador == compara o nome de duas URLs, enquanto o operador < compara a profundidade delas. Por fim, o método ImprimeURL exibe o nome dela. Seu diagrama é apresentado a seguir.

URL
string nome_url URL* next_url int profundidade bool URL_certa
URL(string nome) ~URL() string GetNome() int Get Profundidade void SextNext(URL* u) bool URL_valida() FormataURL() string FormataPraHost() int CalculaProfundidade() bool operator==(URL& u) bool operator<(URL& u) void ImprimeURL()

A próxima classe criada é o Host e seu *header file* e implementação estão nos arquivos *Host.hpp* e *HostL.cpp.* Um host é caracterizado por um nome, um ponteiro para o próximo host, uma vez que ele pertencerá a uma fila, e um ponteiro para uma lista de URLs, as quais são as URLs relacionadas àquele host, que estão declarados em sua parte *private*. Os métodos relacionados a um Host são, primeiramente, o construtor que usa da formatação de strings para atribuir seu nome e um destrutor que, para desalocar a memória utilizada em sua lista de URLs, chama também o destrutor de ListaURL. Além disso, foram implementados os métodos *get* e *set* para acessar os membros *privates* e atribuir novos valores a eles. O operador == compara o nome dos hosts. O método InsereURL adiciona uma nova URL à lista de URLs desse host na posição adequada de acordo com sua profundidade. O método RemoveTodasURLS esvazia a lista de URLs desse host e será melhor descrita em outra classe. Por fim, o método ImprimeHost exibe o nome de todos os hosts na ordem em que foram conhecidos.Ao final do arquivo, mas não membro da classe Host, é implementada uma função que verifica se dois ponteiros apontam para o mesmo host. Seu diagrama de classes é apresentado a seguir.

Host
string nome_host Host* next_host ListaURL* lista_de_urls
Host(string nome)

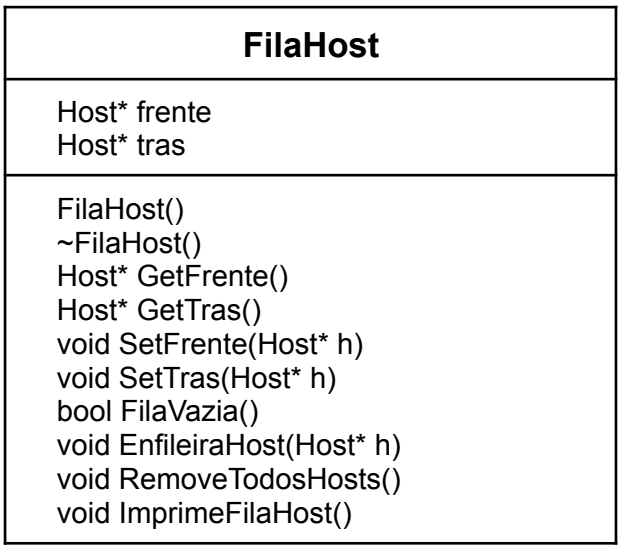
~Host() string GetNome() Host* GetNext() Lista URL* GetPonteiroPraLista(ListaURL *u) void SetPonteiroPraLista(ListaURL *u) void SextNext(Host* h) bool operator ==(Host& h) void InsereURL(URL* u) void RemoveTodasURLS() void ImprimeHost()

A próxima classe criada é ListaURL e seu *header file* e implementação estão nos arquivos *ListaURL.hpp* e *ListaURL.cpp.* Uma lista de URLs é caracterizada por um ponteiro para a primeira posição e um para a última. Os métodos relacionados a uma ListaURL são, primeiramente, seu construtor que, verifica se uma URL é válida(possui protocolo http) antes de a inserir na lista e atualiza os ponteiros primeiro e último para as respectivas posições, o primeiro agora aponta para a URL inseria e o ultimo é *nullptr*. O destrutor dessa classe chama o método EsvaziaLista, responsável por desalocar todas as URLs presentes na lista, enquanto atualiza seus ponteiros primeiro e ultimo, que, no fim, serão ambos *nullptr,* o que caracteriza uma lista vazia (o método EstaVazia verifica essas condições). São implementados também os métodos *get* e *set* para acessar os membros *privates* e atribuir novos valores a eles. O método ImprimeListaURL exibe o nome de todas as URLs presentes na lista. O método Escalona imprime e em seguida desaloca todas as URLs presentes na lista, uma de cada vez. E, por fim, o método TamanhoLista retorna o numero de URLs que essa lista possui. É interessante analisar também, que uma lista encadeada de URLs é caracterizada pela sua flexibilidade em adicionar elementos em qualquer posição, além de alocar memória mediante a adição de novas URLS em espaços não contíguos da memória, o que impossibilita o acesso direto aos seus elementos. Essa estrutura é poderosa quando se trata de quantidade de elementos variável, uma vez que evita o alocamento desnecessário da memória. Seu diagrama é apresentado a seguir.

ListaURL
URL* primeiro URL* ultimo
ListaURL(URL *u) ListaURL() ~ListaURL() URL* GetPrimeiro() URL* GetUltimo() void SetPrimeiro(URL* u) void SetUltimo(URL* u) bool EstaVazia() void EsvaziaLista() void ImprimeListaURL() void Escalona() int TamanhoLista()

A próxima classe criada é FilaHost e seu *header file* e implementação estão nos arquivos *FilaHost.hpp* e *FilaHost.cpp.* Uma fila de hosts é caracterizada por um ponteiro para a posição da

frente e um para a de trás. Os métodos relacionados a uma fila de host são, primeiramente, um construtor que atribui *nullptr* para seus dois ponteiros membros(esta condição é verificada pelo método FilaVazia), e um destrutor que desaloca a memória de todos os hosts presentes na fila. Além disso, foram implementados os métodos *get* e *set* para acessar os membros *privates* e atribuir novos valores a eles. O método enfileira host segue a propriedade da estrutura de dados fila, que só permite a inserção de novos elementos no final e a remoção no início (método FIFO - first in, first out) e sua implementação encadeada evita a alocação desnecessária de memória. Por fim, o método ImprimeFilaHost exibe o nome dos hosts na ordem em que foram conhecidos. Seu diagrama é apresentado a seguir.



A próxima classe criada é Escalonador e seu *header file* e implementação estão nos arquivos *FilaHost.hpp* e *FilaHost.cpp.* O que caracteriza um escalonador é uma fila de hosts, a qual serão aplicadas todas as operações. Os métodos relacionados a um escalonador são, primeiramente, o construtores padrão e o destrutor que, por essa classe possuir uma fila de hosts e não um ponteiro para a fila, será chamado, nesse momento, antes de destruir o Escalonador, o destrutor de FilaHost, pelo princípio de Aquisição de Recurso e Inicialização. O método ADD_URL é responsável por adicionar ao escalonador, isto é, à sua fila, a quantidade de URLs informadas. Para isso, esse método deve passar por toda a formatação adequada de URLs e Hosts e inserir tanto o host quanto a URL nos locais adequados. O método ESCALONA_TUDO deve exibir e remover as URLs de cada host, um de cada vez. O método ESCALONA recebe um número como parâmetro o qual se refere ao números de URLs que devem ser escalonadas adotando a ordem de profundidade, se houverem. O método ESCALONA_HOST escalona todas as URLs de um host, cujo nome é passado como parâmetro. O método VER_HOST também recebe o nome de um host como parâmetro e exibe todas as URLs dele. O método LISTA_HOSTS exibe o nome dos hosts na ordem em que foram conhecidos. O método LIMPA_HOST desaloca a lista de URLs desse host específico enquanto LIMPA_TUDO desaloca todas as listas e também a fila de hosts.
É importar notar que os métodos ESCALONA_TUDO e LIMPA_TUDO devem ser chamados no arquivo de entrada para que toda a memória alocada dinamicamente seja devidamente desalocada. Isso acontece porque na *main* não há a destruição desses objetos. Caso essa desalocação fosse realizada nesses métodos e também na *main,* teríamos o problema de *double free memory.* O diagrama dessa classe é apresentado a seguir:

Escalonador
FilaHost fila_de_hosts
Escalonador() ~Escalonador void ADD_URL(URL* u) void ESCALONA_TUDO() void ESCALONA(int n) void ESCALONA_HOST(string s, int n) void VER_HOST(string host) void LISTA_HOSTS() void LIMPA_HOST(string host) void LIMPA_TUDO()

Por fim, o arquivo *main.cpp* possui a função principal do programa, que receberá o nome do arquivo de entrada como primeiro argumento da linha de comando e criará um arquivo com o mesmo nome de entrada sufixado de *-out*. Esta função conta com a manipulação de arquivos, desde sua abertura até a coleta de dados com sua leitura. Nela também é criado um escalonador, onde serão realizadas todas as operações necessárias. Dessa forma, a seguir estão listadas todos os comandos que o escalonador é capaz de realizar:

- `ADD_URLS <quantidade>`
- `ESCALONA_TUDO`
- `ESCALONA <quantidade>`
- `ESCALONA_HOST <host> <quantidade>`
- `VER_HOST <host>`
- `LISTA_HOSTS`
- `LIMPA_HOST <host>`
- `LIMPA_TUDO`

2.2 Configuração utilizada

O programa foi desenvolvido na linguagem C++, usando o WSL, com Ubuntu-20.04 e o compilador G++ (Ubuntu 9.3.0-17ubuntu1~20.04) da GNU Compiler Collection. O processador da máquina utilizada é Intel(R) Core(TM) i5-4440 CPU @ 3.10Ghz 3.10 GHz com 16GB de RAM instalada.

3.0 Análise de Complexidade

Esta seção apresenta a análise de complexidade dos métodos presentes na classe Escalonador e os métodos subsequentes presentes nele.

Primeiramente, o método **ADD_URL** é composto pelos seguintes métodos e suas respectivas análises:

- URL::URL_certa: esse método realiza apenas operações constantes de atribuição e retorno em tempo O(1) e utiliza estruturas auxiliares O(1). Por isso, tanto sua complexidade assintótica de tempo, quanto de espaço são O(1).
- URL::FormataURL: esse método possui operações constantes em tempo O(1) e um loop com o objetivo de procurar por determinado caractere. Dessa forma, teremos o melhor caso quando o caractere for encontrado na primeira iteração e pior caso quando for encontrado na última iteração ou não for encontrado. Por isso, a complexidade assintótica de tempo é O(1) no

melhor caso e Θ(n) no pior caso. Em relação ao espaço, esse método realiza apenas operações com estruturas auxiliares O(1) e possui complexidade assintótica de espaço O(1).

- Os métodos Host::GetNome, Host::GetPrimeiro, Host::SetPrimeiro, Host::SetUltimo, Host::GetNext, Host::SetNext, FilaHost::GetFrente e Host::host_points_to_equal_object apenas realizam operações constantes e possuem complexidade assintótica de tempo e espaço O(1).
- Host::InsereURL: esse método possui operações constantes em tempo O(1) e um loop que com o objetivo de procurar o lugar adequado para inserir a URL na lista. Se a URL já existir na primeira posição ou sua profundidade for tal qual sua inserção deve ser realizada na primeira posição, o loop terá apenas uma iteração. Entretanto, caso a URL deva ser adicionada no meio da lista, ou no final dela, o loop percorrerá um total de o número de elementos na lista vezes. Por isso sua complexidade assintótica de tempo no melhor caso é O(1) e no pior caso Θ(n). Esse método não realiza alocação de memória dentro de nenhum loop e opera com estruturas auxiliares com complexidade de espaço O(1).
- FilaHost::EnfileiraHost: esse método é composto pelo método FilaVazia que possui apenas uma operação constante de retorno em tempo O(1) e estruturas auxiliares de espaço O(1). Além disso, o método não possui outras operações, senão estruturas condicionais. Por isso, sua complexidade de tempo e espaço são ambas O(1).

O método ADD_URLS também possui um loop que busca pelo host na fila de hosts e terá complexidade assintótica de tempo O(1) caso ele seja encontrado na primeira posição e Θ(n) se for encontrado na última, ou não estiver na fila e precisar percorrer ela até o final.

Analisando todas as complexidades dos métodos presentes no método ADD_URL e as complexidades de suas próprias estruturas e usando a propriedade $\Theta(f(n) + g(n)) = \max(f(n), g(n))$, ao somarmos todas as complexidades assintóticas teremos como O(1) a complexidade assintótica de tempo no melhor caso e Θ(n) no pior caso. Em relação a complexidade assintótica de espaço, teremos complexidade assintótica O(1).

O próximo método é o **ESCALONA_TUDO** e é composto por alguns outros métodos:

- Métodos de Get e Set já analisados anteriormente com complexidade de tempo e espaço O(1)
- ListaURL::Escalona: esse método possui o método URL::ImprimeURL que apenas possui operações constantes em tempo de complexidade O(1) e estruturas auxiliares de espaço com complexidade O(1). Além disso, esse método possui um loop que imprimirá todas as URLs presentes na lista. Dessa forma, no caso de a lista estar vazia, o loop terá nenhuma iteração e se não, terá o número de elementos da lista de iterações. Por isso, sua complexidade assintótica de tempo no melhor caso será O(1) e no pior caso Θ(n). Em relação ao espaço, será alocado na memória apenas 2 ponteiros, o segundo apesar de estar dentro do loop, não ocupará novo espaço na memória e por isso, sua complexidade assintótica de espaço será O(1).

Ao analisarmos todas as complexidades dos métodos presentes no método ESCALONA_TUDO, percebemos a presença de um laço que, se a fila de hosts estiver vazia não iterará nenhuma vez e, se não, percorrerá até achar o host, sendo assim Θ(n). Por isso, e pela propriedade $\Theta(f(n) + g(n)) = \max(f(n), g(n))$, ao somarmos todas as complexidades assintóticas teremos como O(1) a complexidade assintótica de tempo no melhor caso e $\Theta(n^2)$ no pior caso, quando chamar o método Escalona dentro do laço while. Da mesma forma, teremos como O(1) a complexidade assintótica de espaço para qualquer caso.

O próximo método é o **ESCALONA** e é composto pelos seguintes métodos:

- Métodos de Get e Set já analisados anteriormente com complexidade de tempo e espaço O(1).
- Método URL::ImprimeURL com melhor caso de tempo O(1) e pior caso Θ(n) e espaço sempre O(1), como explicado no método anterior.

Após analisarmos as complexidades dos métodos presentes, precisamos verificar a estrutura de ESCALONA. Ela é composta por um laço que escalona n URLs, sendo n passando como parâmetro. Entretanto, se a fila de hosts estiver vazia, não será possível escalonar nenhuma URL. Por isso, a complexidade de tempo no melhor caso será O(1) e no pior caso Θ(n). Em termos de espaço, o método utiliza de estruturas auxiliares O(1) e não possui nenhuma alocação aninhada, tendo assim, complexidade assintótica de espaço O(1).

O próximo método é o **ESCALONA_HOST** e possui a seguinte estrutura:

- Possui métodos de Get e Set e URL::ImprimeURL já analisados anteriormente com complexidade de tempo e espaço O(1)
- Possui o método Escalona já analisado anteriormente com melhor caso O(1) e pior caso Θ(n) para tempo e sempre será O(1) para espaço.

Sabendo disso, o método ESCALONA_HOST possui várias opções a serem seguidas. Primeiramente, pode ter sido passado como parâmetro um host cuja lista de URLs possui menos elementos que a quantidade indicada para escalonar no parâmetro. Nesse caso, o método Escalona, que escalonará todas as URLs desse host e ESCALONA_HOST terá melhor caso em O(1) e pior caso O(n) para tempo e sempre será O(1) para espaço. Por outro lado,o host pode ser achado, respeitando as iterações do primeiro laço, e possuir uma quantidade de elementos maior do que a que se deseja escalonar. Nesse caso teremos outro laço que terá n iterações de acordo com essa quantidade. Ese caso possui dois laços aninhados e, consequentemente, complexidade de tempo $\Theta(n^2)$ e espaço O(1) pois não há apenas alocações simples de operações, que não acompanham em nenhum momento os laços. Por fim, há o caso em que não se encontra o host passado. Assim, será percorrida a fila de hosts inteira e terá complexidade de tempo Θ(n) e espaço O(1), pois não entrará em nenhuma condição, nem em qualquer operação que necessite de alocação de memória.

Analisando todas as complexidades dos métodos presentes no método ADD_URL e as complexidades de suas próprias estruturas e, levando em consideração a propriedade $\Theta(f(n) + g(n)) = \max(f(n), g(n))$, teremos como análise assintótica de tempo no melhor caso O(1) e pior caso $\Theta(n^2)$, com situações explicadas anteriormente. Já para a análise assintótica de espaço teremos O(1) para qualquer ocasião, já que aloca apenas ponteiros dinamicamente e estruturas auxiliares também O(1).

O próximo método é o **VER_HOST** e é composto pelos seguintes métodos:

- Métodos Get que possuem complexidade assintótica de tempo e espaço O(1), como analisados anteriormente.
- ListaURL::ImprimeListaURL: esse método imprime toda a lista de um host. Para isso, ele conta com outros métodos de Get e impressão de uma única URL que são O(1) para tempo e espaço. Além disso, ao percorrer a lista, esse método terá seu melhor caso de tempo quando a lista estiver vazia, não percorrer nada, e terá complexidade O(1) e pior caso quando não está vazia e imprimir todos os elementos da lista, tendo assim, que percorrer por toda ela, o que trará

complexidade de tempo Θ(n). Para a análise de espaço, verificamos que não há alocação de estruturas auxiliares aninhadas e portanto, sempre terá complexidade de tempo O(1).

Analisando todas as complexidades dos métodos presentes em VER_HOST e analisando a presença de um laço que busca pelo host especificado, e possui ImprimeListaURL, podemos concluir que VER_HOST terá melhor caso quando quando a fila de hosts estiver vazia, sendo assim, O(1) e pior caso quando procura pelo host e chama o método ImprimeListaURL que tem pior caso Θ(n). Assim, esse pior caso está caracterizado por dois laços aninhados e possui complexidade assintótica de tempo $\Theta(n^2)$. Para a análise assintótica de espaço teremos O(1) para qualquer situação.

O próximo método é o **LISTA_HOSTS** e possui a chamada do seguinte método:

- FilaHost::ImprimeFilaHost:: esse método percorre a fila de host exibindo pelo método ImprimeHost (que possui complexidade de tempo e espaço O(1)) todos os nomes do host presentes. Para isso, ele possui um laço que percorrerá nenhuma vez, caso a fila estiver vazia e todos os hosts caso contrário. Dessa forma, esse método possui complexidade assintótica de tempo O(1) em seu melhor caso e Θ(n) no pior caso. A análise de espaço será O(1) para qualquer situação, uma vez que esse método somente exibe dados e não armazena nenhuma estrutura auxiliar com complexidade maior que O(1).

Como o método LISTA_HOSTS só possui a chamada do método explicado anteriormente, sua complexidade de tempo e espaço setão semelhantes, ou seja, O(1) e Θ(n) para o pior e melhor caso, respectivamente e sempre O(1) para espaço.

O próximo método é o **LIMPA_HOST** e possui os seguintes métodos:

- Métodos Get com complexidade assintótica de tempo e espaço O(1), como analisadas anteriormente.
- Host::RemoveTodasURLS: esse método apenas possui a chamada do método ListaURL::EsvaziaLista. Este método, então, precisa percorrer a lista de URLs presente e terá melhor caso quando a lista estiver vazia e pior caso quando precisar percorrer todos os n elementos e removê-los. Dessa forma, a análise assintótica do melhor caso é O(1) e do pior caso Θ(n). Não há alocação de memória em laços e portanto apenas estruturas auxiliares O(1) são utilizadas, o que faz sua complexidade de espaço ser O(1) para qualquer caso.

Dada a análise dos métodos presentes em LIMPA_HOST, precisamos considerar, também o laço presente em sua própria estrutura, que não terá nenhuma iteração quando a fila de host estiver vazia e n iterações até achar o host indicado no parâmetro. Quando achar esse host, será chamado o método RemoveTodasURLS que possui pior caso Θ(n), como explicado anteriormente. Dessa forma, este é o pior caso, por se tratar de dois laços aninhados e possui complexidade assintótica de tempo $\Theta(n^2)$. E sem nenhuma iteração teremos o melhor caso em O(1). Ao se tratar de espaço, esse método não necessita de nenhuma alocação de memória aninhada e, portanto, possui complexidade assintótica O(1) para qualquer situação.

O próximo método é o **LIMPA_TUDO** e possui os seguintes métodos:

- Métodos Get com complexidade assintótica de tempo e espaço O(1), como mencionadas nos métodos anteriores.
- Host::RemoveTodasURLS: esse método apenas possui a chamada do método EsvaziaLista, que possui pior e melhor caso para análise de tempo como O(1) e Θ(n), respectivamente e é O(1)

para análise de espaço em qualquer situação. A descrição das suas operações foram mencionadas no método anterior.

Dessa forma, a estrutura de LIMPA_TUDO percorre a fila de hosts enquanto remove a url da lista deles. O melhor caso de tempo será no momento em que a fila está vazia e não precisará percorrer nenhuma elemento, por isso O(1), o pior caso de tempo será quando a fila não estiver vazia e percorrer todos os elementos, chamando RemoveTodasURLS para cada um deles. Como o pior caso desse último método citado é Θ(n), o pior caso para LIMPA_TUDO será Θ(n^2), uma vez que há dois laços aninhados. Em relação ao espaço, esse método trabalha apenas com a desalocação de memória e utiliza estruturas auxiliares O(1) e, assim, possui complexidade assintótica de espaço O(1) para qualquer situação.

4.0 Estratégias de Robustez

A robustez de um código está relacionada aos mecanismos de prevenção a falhas e mau uso do programa. Dessa forma, foram implementados vários testes com o uso de estruturas condicionais que procuram evitar o máximo de erros associados aos arquivos de entrada, a estrutura de dados e o tipo abstrato de dados usados. Isso permite que o programa seja mais seguro e ajuda na compreensão de falhas durante a implementação. Serão descritos os casos gerais de asserção que foram usados.

- Listas e filas vazias: todos os métodos que possuem operações sobre as listas de URLs e a fila de Hosts possui uma estrutura condicional que verifica anteriormente se esta estrutura está vazia. Essa verificação é feita com os métodos FilaHost::FilaVazia e EstaVazia.
- Criação de uma URL: no construtor de URL, há um método que verifica se a URL pode ser criada, ou seja, se possui o protocolo http. Isso evita que sejam criadas URLs desncessárias que não obedecem o padrão especificado.
- Manipulação de ponteiros: nos métodos em que é necessário realizar operações com ponteiros, é verificado antes, se estes ponteiros são nullptr.
- Manipulação de arquivos: os arquivos de entrada e saída que necessitam ser abertos durante a execução do programa devem possuir uma asserção que verifica se foram abertos corretamente.
- Arquivo de entrada: na função main há a verificação se o arquivo de entrada é do formato adequado, ou seja, .txt.

5.0 Conclusões

A construção do programa permitiu a melhor fixação da manipulação de estruturas alocadas dinamicamente. Esse aprendizado foi muito interessante para amadurecer o manuseio e entendimento da memória heap e suas grandes vantagens em relação a alocação estática. Esse estudo deixou ainda mais evidente o uso consciente e otimizado da memória dinâmica, uma vez que sua alocação é flexível ao desejo do usuário. Com a resolução desse trabalho, foi possível praticar, também, a leitura de argumentos na linha de comando e o contato com arquivos, aprofundando assim, nas operações de fluxo de leitura e escrita relacionadas a eles. A maior dificuldade encontrada foi a exibição gráfica da análise do padrão de acesso à memória e da distância de pilha. Esta parte não foi concluída com sucesso, apesar de ser imprescindível para melhor entendimento do desempenho do programa. Em contrapartida, o planejamento e construção do diagrama de classes e a implementação geral do código foram realizadas de maneira muito mais autônoma em relação a trabalhos realizados anteriormente, e esse foi o maior ganho com a resolução deste trabalho.

6.0 Bibliografia

DAMAS, Luis (2007). **Linguagem C.** Capítulo 9: Passagem de Parâmetros na Linha de Comando. Editora LTC.

KRZYZANOWSKI, Paul. **Processing the Command Line.** 2019. Disponível em: https://people.cs.rutgers.edu/~pxk/416/notes/c-tutorials/getopt.html. Acesso em: 14 Nov. 2021.

HAMMEM, David. **Retorno de ifstream.eof().** 2016. Disponível em: https://www.ti-enxame.com/pt/c%2B%2B/por-que-ifstream.eof-nao-retorna-true-depois-de-ler-ultima-linha-de-um-arquivo/l966955117/. Acesso em 10 Dez. 2021.

Apêndice: Instruções para compilação e execução

Para realizar a instalação do programa, é necessário descompactar o arquivo Raissa_Miranda_2020006965.zip. Uma vez descompactado, basta acessar o diretório em que o programa foi armazenado:

> *cd <diretoriodestino>*

Agora, é necessário apenas realizar o comando *make* e executar o programa com o nome do arquivo .txt de teste com os seguintes comandos:

> *make*

> *cd bin*

> *./main <caminho-para-o-teste>*

Será criado um arquivo com o mesmo nome do arquivo de entrada, porém com o sufixo -out no mesmo diretório com os dados de saída.