Trabalho Prático 1
Escalonador de URLs

Igor Lacerda Faria da Silva[1]

[1]Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte - MG - Brasil

igorlfs@ufmg.br

1 Introdução

O problema proposto foi implementar um escalonador de URLs, como parte de um coletor de uma máquina-de-busca. O propósito de um escalonador é auxiliar na definição de uma ordem para que as páginas sejam coletadas. Existem diversas estratégias para realizar essa tarefa, mas a adotada nesse trabalho foi a *depth-first*, que consiste em coletar todas as URLs de um dado host antes de passar para o próximo. Foi implementada uma leitura de arquivos, que continham uma série de instruções para a execução do escalonamento, cuja saída também foi impressa em um arquivo.

Esta documentação tem como proposta explicar como se deu essa implementação, desde questões mais ligadas ao funcionamento do programa (Seção 2) e estratégias de robustez (Seção 4) como análises de complexidade (Seção 3) e experimentais (Seção 5). Ao final do texto, encontra-se uma conclusão (cujo conteúdo está mais relacionado ao aprendizado pessoal do autor com o trabalho), bibliografias e, por último, as instruções para compilação e execução.

2 Método

O programa foi desenvolvido em C++ e compilado utilizando o g++, do GNU Compiler Collection. A máquina que foi usada durante o desenvolvimento conta com 3.8Gi de memória RAM, e processador Intel(R) Core(TM) i3-2350M CPU @ 2.30GHz, e roda o sistema operacional GNU/Linux (versão do kernel: 5.15.6).

A formatação do código fonte (**incluindo a indentação**): **foi feita usando a ferramenta clang-format**. Foi usado um arquivo customizado para isso, que se encontra na raiz do projeto, com o nome de *.clang-format*. É um arquivo bem curto, baseado em preferências pessoais do autor, mas que **garante a consistência da formatação do projeto**.

2.1 Organização do código

O projeto atende à especificação no que diz respeito à organização do código de forma geral (cabeçalhos em `./include`, etc). Em particular, a única divergência é que os *headers* usam a extensão `.hpp` e não `.h` (idiossincrasia do editor de texto).

Alguns dos arquivos de cabeçalho definem estruturas de dados mais básicas e gerais, como `cell.hpp`, que define uma célula genérica de lista usando templates; `linearlist.hpp`, que define uma lista abstrata; e 2 especializações baseadas em alocação dinâmica de memória, a saber `linkedlist.hpp` (lista encadeada) e `linkedqueue.hpp` (fila encadeada). Outros arquivos já apresentam estruturas mais voltadas à aplicação em questão, como `url.hpp` (um conjunto de strings conforme definido na especificação), `site.hpp` (um Host e uma lista de de URLs, para serem os membros da fila) e `escalonador.hpp`, que implementa de fato as diversas operações solicitadas. Por fim, também tem os ligados à robustez e às análises, como o `msgassert.hpp`.

A estruturas dos arquivos fonte é similar, mas algumas classes possuem métodos muito simples, que não foram implementados num arquivo fonte separado. As listas não abstratas e a classes `URL` e `Escalonador` foram implementados separadamente. O último arquivo dessa categoria é o programa principal, que se limita a fazer somente o básico para atender à especificação, chamando métodos implementados em outras partes do programa.

2.2 Estruturas de Dados, TADs e métodos

Existem duas estruturas de dados principais no programa: lista encadeada e fila (encadeada). Ambas são derivações de uma estrutura *lista linear*, sendo que a fila é muito mais restritiva com relação a algumas operações, como inserção e remoção. Em princípio, a lista linear consiste numa sequência de n elementos (que pode ser vazia), em que há noção de sucessão e antecedência. Nesse sentido, uma propriedade interessante é a existência de um elemento que não tem sucessores (chamado cauda) e um elemento que não tem antecessores (chamado cabeça), que não são necessariamente distintos.

Neste trabalho, uma classe abstrata `LinearList` foi usada para representar uma lista linear. Ela possui apenas um membro, `size` (int), e métodos muito simples: um construtor que inicializa o membro como 0, um método `getSize()` que retorna `size`, um método `empty()` que verifica se `size` é igual a 0, e um método virtual de limpeza: `clear()`.

2.2.1 Lista Encadeada

A primeira especialização de lista linear trabalhada foi a lista (simplesmente) encadeada, implementada na classe `LinkedList` (herdeira de `LinearList`). Na lista simplesmente encadeada, cada elemento é uma célula, que possui seu conteúdo e um apontador para a posição seguinte. A célula foi implementada na classe `Cell`, que possui esses membros e um construtor que inicializa a posição

seguinte para nulo. A `LinkedList` usa alocação dinâmica de memória, com uma célula cabeça cujo conteúdo não importa, tendo como propósito simplificar a implementação de alguns métodos, e células do tipo `URL`.

A implementação da classe `LinkedList` teve como grande inspiração as aulas do professor Chaimowicz. Os principais métodos são os de construção, destruição (e limpeza), inserção e remoção. No construtor somente é criada uma nova célula, que assume o papel de cabeça e cauda. No destrutor, o método auxiliar `clear()` é chamado, e a célula restante é destruída. O método `clear()` deleta as células em sequência, caso existam. Há 2 métodos de inserção, a depender da posição: `insertBeg()`, para o começo e `insertPos()` em uma posição arbitrária. A classe também conta com um método de remoção, `removeBeg()`, que remove elementos do começo (não foi preciso remover elementos de outras posições, pela especificação).

Além disso, há um método de impressão `print()` (que imprime, em sequência, as URLs) e alguns voltados para o uso com URLs: o `searchDepth()`, que compara as profundidades das URLs para definir a posição de inserção (que é em seguida passada para o `setPos()`) e o `containsUrl()` que verifica se uma dada URL está presente na lista. Por fim, existem dois métodos de escalonamento: um que toma um inteiro n como parâmetro, e escalona até n células, e um que escalona toda a lista.

2.2.2 Fila Encadeada

Em filas, que são um tipo específico de listas, a inserção só é permitida em uma extremidade e a remoção só é permitida na outra extremidade. Na classe `LinkedQueue` (herdeira de `LinearList`), que implementa uma fila, a inserção só é permitida nos fundos (*rear*) e a remoção então só é permitida na frente (*front*). De resto, ela é semelhante à `LinkedList`, também inspirada nas aulas do professor Chaimowicz, fazendo uso de células e afins. Em particular, ela é uma fila de Sites (tipo `Site`), que é uma classe que contém um Host e uma lista de URLs. Os principais métodos da `LinkedQueue` são o contrutor, destrutor (e limpar), enfileirar (`line()`), que funcionam de forma semelhante aos seus análogos da `LinkedList` (aqui, `line()` é como `insertBeg()`, já que é a única posição permitida de inserção).

Ademais, também possui métodos relacionados ao uso do escalonador em si, como o método que retorna a lista de URLs dado um Host (se estiver presente) (`getUrlsFromHost()`), e duas funções de impressão distintas (uma imprime os Hosts e a outra imprime até n URLs entre todas da fila, em ordem). Finalmente, o método `escalonaTudo()`, que chama o método de escalonar tudo para cada um dos sites da fila.

2.3 Outras classes

Além das listas (e célula), o programa conta com 3 classes ligadas ao escalonamento: `URL`, `Site` e `Escalonador`. A `URL` é somente um conjunto de strings, cada uma representando uma parte da URL (protocolo, host, path, etc), um

natural (a profundiade), e alguns métodos: diversos *getters*, um método de impressão (`print`) e 2 contrutores, um *default* (necessário para o uso como célula via templates) e um que constrói uma URL dada uma string. Essa classe foi implementada para simplificar alguns requisitos na construção de URLs e facilitar a separação entre as diferentes partes da URL.

A classe `Site`, como mencionado anteriormente, tem como propósito fundamental aliar uma lista de URLs a um Host (string), possuindo alguns *getters*, *setters*, *printers*[1] e 2 construtores: um *default* e um que dada uma URL, inicializa o Host e a cabeça da lista de URLs. O interessante dessa implementação é que ela permite separar os Hosts da sua lista de URLs, assim, é possível escalonar e continuar "lembrando" do Host.

A classe `Escalonador` possui como membros uma fila de sites e um arquivo de saída, e implementa todos os métodos da especificação, além de alguns adicionais, para a leitura do arquivo (`readFile()`) e auxiliar durante a inserção (`addUrls(), isUrlForbidden()`). E claro, um destrutor e um contrutor, que manejam o arquivo de saída.

3 Análise de Complexidade

A seguir, são analisadas as complexidades de tempo e de espaço dos principais métodos (as 8 instruções da especificação) do escalonador. A escolha de analisar somente esses métodos teve como base `essa` pergunta do fórum. Alguns pressupostos foram tomados, como assumir que certas funções padrões de C++ são $\Theta(1)$. Outra presunção é que o construtor de URLs com parâmetro é $\Theta(1)$, o que não é bem verdade.

3.1 Métodos

- ADD_URLS (`addUrls()` e `insertUrl()`)

  Tempo: a inserção de URLs é dividida em 2 métodos. `addUrls()` tem finalidades simples: coletar a linha, verificar se o final do arquivo foi atingido e chamar `insertUrl()` se a linha coletada for válida. `insertUrl()` é o "cérebro" da inserção. Por isso, não vou analisar minuciosamente `addUrls()`. São realizadas algumas verificações para a inserção, e algumas declarações de variáveis do tipo string, todas $O(1)$. Em seguida, é usada a função `getUrlsFromHost()`. Essa função aparece em diversos métodos. Ela busca na fila, sequencialmente, se dado Host está presente, e se estiver, retorna sua lista de URLs (caso contrário um ponteiro nulo é retornado). Para `insertUrl()`, o melhor caso é se o Host está ausente, pois aí a fila é estendida, e o método para enfileirar é $\Theta(1)$. Então, no melhor caso, a inserção é $\Theta(n)$, para n tamanho da fila.

  Se o Host está presente, existe variação conforme a posição na fila (se o Host corresponde ao primeiro elemento, temos o melhor caso, se corresponde ao último, temos o pior), e deve ser avaliado se a URL já foi inserida

[1] Achei essa terminoliga apropriada para métodos de impressão.

anteriormente. Se a URL for repetida, nada é realizado. Mas para se verificar que a URL é repetida, é preciso conferir, o que é $\Theta(m)$, pela busca sequencial de `containsUrl()`. Considerando o pior caso, temos $\Theta(n+m)$ nessa situação. Mais uma possibilidade é *Host presente, URL ausente*: a princípio é recuperada a *profundidade* da URL em tempo constante, é feita uma busca para encontrar a posição certa de inserção (que na pior das hipóteses é a última), e por úlitmo, a inserção na posição correta, com `insertPos()`, que chama `setPos()`. Então, recapitulando o pior cenário possível: Host presente, URL ausente, URL tem profundidade maior do que as presentes. Assim, além das buscas supracitadas, `searchDepth()` e `insertPos()` "adiconam um nível à linearidade" (ambos $\Theta(m)$), cada, concluindo $\Theta(n+3m)$. Aqui com certeza existe uma redundância, entre a busca pela profundidade correta e a inserção, que caminha até a posição correta.

Espaço: com Host ausente ou URL repetida, esse método é $\Theta(1)$ com relação a complexidade de espaço. Nos outros casos também, nenhuma estrutura depende de algum tamanho em particular.

- ESCALONA_TUDO (`escalonaTudo()`)

  Tempo: essa função chama `escalonaTudo()` da classe `LinkedQueue`, que por sua vez faz chamadas sucessivas à função `escalonaTudo()` da classe `LinkedList`, e uma verificação $O(1)$. A fila é percorrida completamente, supondo-se tamanho n. No método da lista, cada uma é percorrida m_i vezes, em que m_i é o tamanho de cada lista. Certamente existe uma lista com M elementos tal que $M \geq m_i \forall i \mid 1 \leq i \leq n$. No pior dos casos é possível então assumir que todas as listas têm tamanho M. Então, cada elemento da fila entra em um laço que é executado M vezes. Assim, temos que a complexidade assintótica de tempo do método `escalonaTudo()` é $\Theta(Mn)$, ou seja, quadrática. Também podem ser consideradas outras situações, como: todos os Sites *estão vazios.* Nesse caso, o `escalonaTudo()` da lista só executa uma comparação, então nesse caso o método como um todo fica $\Omega(n)$.

  Espaço: apesar de existirem múltiplos casos, todos envolvem estruturas auxiliares unitárias, então a complexidade de espaço é $\Theta(1)$.

- ESCALONA (`escalonaN()`)

  Tempo: essa função chama `escanolaNUrls()` da classe `LinkedQueue` e faz uma verificação. Se a fila não está vazia, são executados dois laços *nestados*. No laço de fora, são realizadas operações de custo constante. No laço de dentro também! A remoção do começo e a impressão só envolvem operações unitárias. A regra que quebra ambos os laços está relacionada a n, parâmetro da função. Desse modo, apesar de o laço interior também ser controlado pelo tamanho da lista, a função `escalonaNUrls()` como um todo é $\Theta(n)$. Naturalmente, se n é maior que a quantidade de URLs na fila como um todo, a execução é parada ao se atingir o fim da fila.

  Espaço: são criadas diversas estruturas auxiliares de tamanho constante. Portanto, a complexidade de espaço é $\Theta(1)$.

- ESCALONA_HOST (`escalonaHost()`)

  Tempo: essa função chama `getUrlsFromHost()`, e caso o Host esteja presente, escalona-o. Se o Host estiver ausente, a fila inteira é caminhada, em complexidade $\Theta(n)$, em que n é o tamanho da fila, mas há um retorno logo em seguida. Se o Host estiver na fila, o melhor caso é se ele corresponder ao primeiro elemento da fila, e o pior é se ele corresponder ao último. É feita uma verficação do parâmetro `int` passado: o escalonamento só ocorre para, no máximo C elementos, em que C é o tamanho da lista. Ou seja, se supormos um parâmetro arbitrariamente grande, o laço será executado ainda C vezes. Desse modo, no pior caso, esse método tem complexidade de tempo $\Theta(???)$.

  Espaço: são criadas algumas estruturas auxiliares unitárias. A complexidade de espaço é $\Theta(1)$.

- VER_HOST (`listUrls()`)

  Tempo: essa função chama a função `getUrlsFromHost()`, e caso o Host esteja presente, imprime sua lista em ordem. Há também uma verficiação que é $O(1)$. Se o Host estiver ausente, é preciso caminhar a fila inteira, o que representa uma complexidade $\Theta(n)$, em que n é o tamanho da fila. Se o Host estiver na fila, o melhor caso é se ele corresponder ao primeiro elemento da fila, pois assim é realizada apenas uma comparação (analogamente, o pior caso é se for o último). Nesse caso, o método `print()` da classe `LinkedList` é chamado, percorrendo a lista toda, o que representa uma complexidade $\Theta(m)$, em que m é o tamanho da lista. Assim, no geral, no pior caso, o método `listUrls()` é $\Theta(m + n)$, ou seja, sua complexiade de tempo é linear.

  Espaço: são criadas algumas variáveis unitárias, nos métodos chamados. Assim, a complexidade de espaço é $\Theta(1)$.

- LISTA_HOSTS (`listHosts()`)

  Tempo: essa função simplesmente chama a função `printHosts()` e verifica se a escrita ocorreu normalmente. A função `printHosts()` realiza operações constantes e, por sua vez, chama outra função, `printHost()`, que realiza uma única operação constante. No entanto, `printHosts()` possui um laço que é iterado n vezes, em que n é o tamanho da fila. Desse modo, sua complexidade de tempo é $\Theta(n)$.

  Espaço: no método auxiliar `printHosts()` é criada uma variável unitária. Assim, a complexidade de espaço de `listHosts()` é $\Theta(1)$.

- LIMPA_HOST (`clearHost()`)

  Tempo: essa função chama a função `getUrlsFromHost()`, e caso o Host esteja presente, limpa-o. Aqui há vários cenários a serem considerados. Primeiro que se o Host não estiver presente, ele não é limpo. Mas para se deduzir que o Host está ausente, deve-se caminhar a fila inteira, o que tem complexidade $\Theta(n)$, em que n é o tamanho da fila. Por outro lado, se o Host estiver na fila, o melhor caso é se ele for o primeiro elemento, então só é realizada uma comparação. No entanto, o método `clear()` é chamado, que tem complexidade $\Theta(m)$, em que m é o tamanho

da lista. Portanto, nesse caso, a complexidade é também $\Theta(m)$. O pior caso é a busca pelo último item da fila, que vai ter complexidade $\Theta(m+n)$ (considerando o método `clear()`), que é ainda linear.

Espaço: ao longo da execução são criadas algumas estruturas auxiliares de tamanho unitário, então esse método tem complexidade de espaço $\Theta(1)$.

- LIMPA_TUDO (`clearAll()`)

  Tempo: essa função chama a função `clear()` da `LinkedQueue`. Cada um dos n sites é deletado, mas cada lista de m_i URLs também é. Certamente existe uma lista com M elementos tal que $M \geq m_i \forall i \mid 1 \leq i \leq n$. No pior do casos, assume-se que todas as listas tem M elementos a serem deletados. Assim, para cada `Site` da fila são feitas $M+1$ exclusões, o que resulta em uma complexidade de tempo quadrática $\Theta(n(M+1))$.

  Espaço: como nos casos anteriores, é criada uma variável auxiliar unitária, então esse método tem complexidade de espaço $\Theta(1)$.

4 Estratégias de Robustez

Foram empregadas algumas estratégias de robustez ao longo do programa. Não foi priorizada nenhuma propriedade: tanto robustez como corretude são usadas, a depender do caso. Quando se opta pela corretude é porque houve algum "erro irremediável", desse modo, é acionado o macro `erroAssert(e,m)`, definido no header `msgassert.h`.

4.1 URL

Alguns cuidados precisaram ser tomados ao se trabalhar com a URL: no construtor, assume-se que toda URL contém a substring "://". Essa precaução é então tomada na hora de se inserir a URL na lista: se ela não conter a substring, ela não é inserida. Outro cuidado tomado foi na hora de remover o "www." das URLs que o possuem: deve ser o "www." que é sucessor de "://". No mais, é um importante prestar atenção em onde começam e onde terminam as partes da URL.

4.2 LinkedList

As principais exceções na lista ligada estão relacionadas à memória ou ao acesso de uma posição inválida. Como a alocação é dinâmica, sempre se verifica se ela ocorreu apropriadamente, usando o `std::nothrow`. Essa verificação é feita no construtor e nos métodos de inserção. Quando a lista está vazia, o método de remoção não deve funcionar, então o programa é abortado caso feita essa solicitação. O método `setPos()` possui uma verificação se a posição solicitada é válida, e se não for, o programa é encerrado. Mais um caso deve ser tratado: não é válido escalonar posições excedentes ao tamanho da lista. Quando isso

foi uma preocupação, na hora de escalonar um Host, foi tomado o tamanho da lista como um limite superior para o escalonamento.

4.3 Site

Nessa classe só existe um tratamento possível, que não foi implementado. Em princípio, a função `setHost()` deveria limpar a lista de URLs do Site (afinal, não faz sentido um novo Host armazenar as URLs antigas). Mas como o programa não tem sobreposição de Sites em nenhum momento, essa tática não foi adotada.

4.4 LinkedQueue

Na fila, assim como na lista, é preciso sempre verificar se a alocação dinâmica ocorreu adequadamente (no construtor, no método `line()`). Ademais, o método `escalonaNUrls()` também exige certo tratamento: se a lista está vazia, não há como escalonar n URLs, mas aqui a abordagem foi de tolerância à falhas. Também é preciso checar se o final foi atingido sem atingir as n URLs, seguindo a mesma abordagem.

4.5 Escalonador

Como mencionado anterirormente, o escalonador possui um membro que é um arquivo de saída. Desta maneira, o construtor e o destrutor ficam responsáveis por gerenciar a abertura e o fechamento do tal, respectivamente. Em caso de erro o programa é abortado. É nessa classe em que há uma validação das URLs com respeito ao protocolo e afins. Sempre que é realizada uma operação que envolve a escrita no arquivo, é verificada se a mesma ocorreu conforme o esperado usando o `fail()`. O método `clearAll()`, que *limpa tudo* faz uma checagem adicional para verificar se a limpeza ocorreu como esperado, avaliando se o tamanho é nulo.

O arquivo é parcialmente validado pela função `isLineValid()`, que avalia, linha a linha, se a **instrução** fornecida está correta. Algumas instruções exigem verificação mais robusta ("match" completo), outras são menos rígidas, com possíveis exceções tratadas em classes de nível mais baixo. A numeração dos comandos seguiu a da especificação do TP, e não uma ordem mais coerente com a implementação. Detalhe: é importante que a instrução ESCALONA venha depois de ESCALONA_TUDO e ESCALONA_HOST, para evitar conflitos. Após a execução de cada instrução, verifica-se se a leitura ocorreu como esperado, usando a função `bad()`. Em duas etapas é checado se o EOF foi atingido: na leitura das instruções e na leitura das URLs, que é feita num método separado (`addUrls()`).

4.6 Programa principal

No programa principal existem dois tratamentos: é exigido que o usuário passe pelo menos um parâmetro ao executrar o programa e, como na classe

`Escalonador`, verifica-se a abertura e fechamento do arquivo de entrada usando a função `is_open()`.

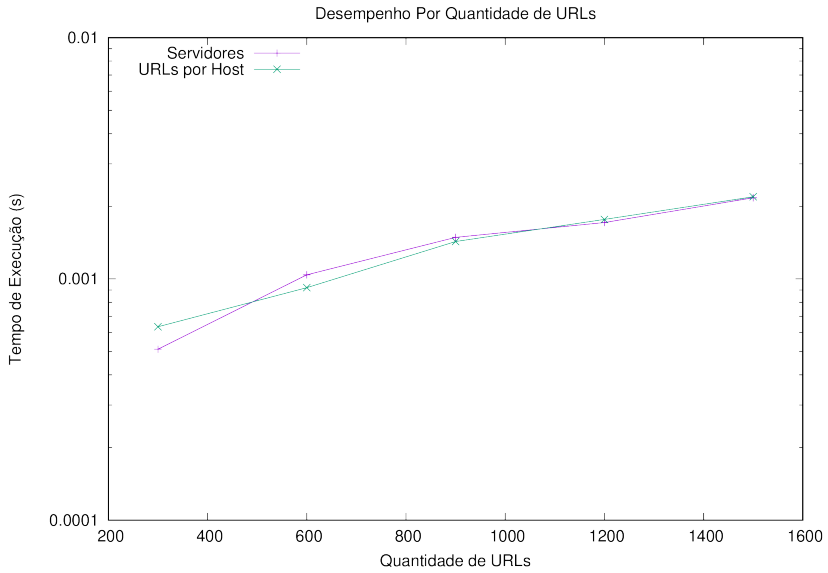
5 Análise Experimental

Nesta seção são apresentados alguns experimentos que avaliam a performance do programa: tanto no que diz respeito ao desempenho computacional (tempo de execução) quanto a eficiência no uso de memória (padrão de acesso e localidade de referência). Grandes agradecimentos ao professor Wagner Meira Júnior, por disponibizar o gerador de carga usado na subseção 5.1.

5.1 Desempenho Computacional

Nesta subseção é avaliado o impacto da variação de parâmetros para o programa, usando o "gerador de carga" e a biblioteca `memlog`, que foi usada para criar um registro com os tempos de início e finalização do programa. Somente as operações padrão do gerador de carga foram analisadas, ou seja, somente a inserção de URLs, a listagem do Hosts, o escalonamento completo e o método de limpeza geral. Além disso, não houve um detalhamento da duração de cada etapa. Essas escolhas foram feitas por simplificação, uma vez que uma análise metódica de cada operação ficaria muito extensa e imprática.

Foram feitas duas análises do impacto na performance para diferentes parâmetros de variação da quantidade de URLs: variando o total de servidores (e mantendo o número de URLs por Host constante, igual a 3) e variando o número de URLs por Host (e mantendo o total de servidores constante):



Mais especificamente, os parâmetros para o gerador de carga para o teste com servidores foram: `-s <quantidade> -u 3 -v 0 -p 5 -q 0.68`, em que a quantidade é a variável. A variança das URLs foi nula para se obter um número exato de URLs, e os parâmetros para a profunidade foram completamente arbitrários. Já no caso dos parâmetros para o teste variando as URLs por Host: `-s 30 -u <quantidade> -v 0 -p 5 -q 0.68`, em que a quantidade é a variável e as mesmas justificativas para os outros parâmetros se aplicam.

Os resultados obtidos não contradizem a análise de complexidade e estão dentro do esperado. Outro parâmetro analisado foi a *profundidade* das URLs, mas não foi obtida diferença significativa para essa mudança (como esperado).

5.2 Eficiência de Acesso à Memória

6 Conclusões

Neste trabalho foi implementado um programa que é um componente de uma máquina de busca: um escalonador de URLs, usando a estratégia depth-first, fazendo uso de filas e listas alocadas dinamicamente e contando com leitura e escrita de arquivos.

6.1 Aprendizado Pessoal

Essa seção está escrita em primeira pessoa intencionalmente.

Implementei, ainda que com grande ajuda, uma fila e uma lista encadeada. Como já tinha implementado uma pilha por causa de um dos exercícios das aulas, fico à mercê de implementar uma àrvore para completar as estruturas de dados básicas. Mas imagino que essas são cenas do próximo capítulo. Ainda sobre a implementação, uma coisa que pensei posteriormente é que talvez fosse mais interessante fazer uma implementação abstrata dessas classes e herdar, especificando as operações. Por exemplo, fazer uma lista encadeada genérica e criar uma classe lista encadeada de URLs, implementando as operações específicas de URLs. Isso seria muito interessante no sentido de que eu teria à minha disposição implementações prontas pra quando precisassse. Inclusive, acho que em algum momento, tentei fazer isso, mas não foi muito pra frente.

Dessa vez, mal tive problemas com o LaTeX, e até o momento em que estou escrevendo isso, não mexi com testes de unidade. Ou seja, meu ganho "tangente" não foi tão alto dessa vez.

Fazer as análises de complexidade foi mais uma vez bem trabalhoso. Fiquei meio chateado de ter *comido uma mosca* no TP 0, mas a expectaviva é que nesse devo ter comido *mais*. Mas acho que estou começando a ganhar familiaridade com o assunto, inclusive pulei umas coisas mais *low-level* de propósito.

7 Bibliografia

1. CHAIMOWICZ, Luiz. **Listas Encadeadas**. [S. l.], 24 ago. 2020. Disponível em: `https://www.youtube.com/watch?v=l4gEM46FznI`. Acesso em 6 dec 2021.
2. CHAIMOWICZ, Luiz. **Filas - Implementação com Apontadores**. [S. l.], 24 ago. 2020. Disponível em: `https://www.youtube.com/watch?v=scNtNVD6HRE`. Acesso em 6 dec 2021.

Instruções

Compilação

Você pode compilar o programa da seguinte maneira:

1. Abra um terminal;
2. Utilize o comando `cd` para mudar de diretório para a localização da raiz do projeto;
3. Utilize o comando `make`.

Pronto! O programa principal foi compilado.

Execução

Você pode rodar o programa da seguinte maneira:

1. Abra um terminal;
2. Utilize o comando `cd` para mudar de diretório para a localização da raiz do projeto;
3. Utilize o comando `./bin/binary <nome-arquivo-entrada>`. Esse parâmetro é obrigatório!