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 busca se dado Host está na fila (`isHostInQueue()`), o que retorna a lista de URLs dado um Host (`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, temos um 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

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

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

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()`). Além disso, a função `getUrlsFromHost()` pressupõe que o Host está na fila: sempre que esse método é chamado, o método `isHostInQueue()` é chamado antes, para verificar essa condição. Caso sejam feitas alterações a posteriori que por ventura quebrem essa condição, o `geteUrlsFromHost()` *joga* uma exceção. Por fim, 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

6 Conclusões

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!