



**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
(IFMG - CAMPUS SABARÁ)**

Tiago Mol Fonseca e Samuel Vital Santos Silva

TP Escalonador

MINAS GERAIS

2025

Introdução

Neste trabalho foi desenvolvido um escalonador de links, o sistema foi desenvolvido em Java com uso do GITHUB para compartilhamento do desenvolvimento, como referenciado nas instruções de desenvolvimento disponibilizadas pelo professor o escalonador deverá ser utilizado para definir a ordem em que as páginas apontadas pelas URLs serão coletadas, esta ordem variando de acordo com a estratégia de escalonamento abordada, para este trabalho foram desenvolvidas as abordagens breadth-first e best-first.

Método

As classes desenvolvidas neste trabalho foram: *Escalonador.java*, *Leitor.java*, *ListaURLS.java*, *Main.java*, *URL.java*. Abaixo encontra-se a descrição de cada uma e seus métodos explicados.

Escalonador.java

Esta classe é responsável por gerenciar, organizar e escalar URLs de acordo com diferentes critérios de prioridade e estratégias de ordenação. Seu principal objetivo é controlar a inserção, agrupamento, filtragem e remoção de URLs, utilizando como base uma estrutura de dados do tipo lista duplamente encadeada representada pela classe *listaURLS*.

O método **ADD_URLS()** é responsável por adicionar novas URLs ao escalonador. Ele recebe como parâmetros a quantidade de URLs a serem inseridas e um vetor contendo essas URLs. Durante o processo, o método verifica a existência de URLs duplicadas, garantindo que apenas URLs únicas sejam inseridas ao final da lista.

O método **QTD_URLS_HOST()** contabiliza quantas URLs pertencem a um determinado host. Para isso, percorre toda a lista e compara o host de cada URL com o host informado, retornando o total encontrado.

Os métodos de escalonamento implementam diferentes estratégias de ordenação e seleção das URLs:

- **ESCALONA_TUDO_PROF()** escala todas as URLs disponíveis, agrupando-as por host e respeitando a ordem de prioridade baseada na profundidade do caminho.

- ***ESCALONA_PROF()*** realiza o mesmo escalonamento por profundidade, porém limita a quantidade total de URLs retornadas.
- ***ESCALONA_TUDO_LARG()*** escalona todas as URLs seguindo uma estratégia em largura, alternando URLs entre diferentes hosts.
- ***ESCALONA_LARG()*** aplica o escalonamento em largura com limitação da quantidade de URLs escalonadas.
- ***ESCALONA_TUDO_BEST()*** escalona todas as URLs priorizando os hosts com maior quantidade de URLs disponíveis.
- ***ESCALONA_BEST()*** segue a estratégia *best*, mas limita a quantidade total de URLs retornadas.

O método ***ESCALONA_HOST()*** realiza o escalonamento exclusivo das URLs pertencentes a um único host, respeitando a quantidade informada. Após o escalonamento, as URLs selecionadas são removidas da lista principal.

O método ***VER_HOST()*** retorna todas as URLs associadas a um host específico, ordenadas por prioridade com base na profundidade do caminho, sendo URLs mais superficiais priorizadas. Esse método utiliza um algoritmo de ordenação simples para organizar as URLs conforme a profundidade.

O método ***LISTA_HOSTS()*** gera e retorna um vetor contendo todos os hosts conhecidos pelo escalonador, respeitando a ordem em que foram inseridos e evitando duplicações.

Por fim, os métodos de limpeza permitem o gerenciamento do estado interno do escalonador:

- ***LIMPA_HOST()*** remove todas as URLs associadas a um host específico.
- ***LIMPA_TUDO()*** remove todas as URLs armazenadas, reinicializando completamente o escalonador.

Leitor.java

Esta classe é responsável por realizar a leitura dos arquivos de teste, interpretar os comandos neles contidos e gerar os arquivos de saída correspondentes. Além disso, cabe a ela invocar os métodos da classe **Escalonador** de acordo com os comandos extraídos dos arquivos de teste.

A classe possui dois métodos principais: ***getLinhas()*** e ***interpretador()***.

O método ***getLinhas()*** recebe como parâmetro uma String que representa o caminho, no sistema do usuário, do arquivo de teste a ser lido. Durante sua execução, o método utiliza as classes ***BufferedReader*** e ***FileReader*** para realizar a leitura do arquivo .txt, extraíndo todas as linhas presentes.

Cada linha lida é armazenada em um vetor de String, que ao final da execução do método é retornado, contendo todo o conteúdo do arquivo de teste.

O método ***interpretador()*** recebe como parâmetros um vetor de String, contendo as linhas extraídas do arquivo de teste, e uma String que indica o caminho de saída onde os arquivos de resultado deverão ser salvos após a execução do sistema.

Esse método processa cada linha individualmente, realizando a separação dos comandos a partir dos espaços em branco presentes em cada linha. Dessa forma, é possível identificar corretamente cada instrução e seus respectivos parâmetros, permitindo que os métodos apropriados da classe ***Escalonador*** sejam chamados e executados conforme definido nos comandos do arquivo de teste.

ListaURLS.java

Esta classe é utilizada para a criação e gerenciamento das listas de URLs associadas a cada host. A estrutura adotada é uma lista duplamente encadeada clássica. A classe implementa os métodos tradicionais dessa estrutura de dados, como ***inserirInicio***, ***inserirFim***, ***removerInicio*** e ***removerFim***, responsáveis pelas operações básicas de inserção e remoção de elementos na lista.

Além desses, a classe possui dois métodos específicos:

- ***removeURL()***: recebe como parâmetro uma URL específica e realiza a remoção desse elemento da lista, caso esteja presente.
- ***qtdURLS()***: retorna a quantidade total de URLs atualmente armazenadas na lista.

Main.java

A classe ***Main*** é responsável pela execução do programa. Trata-se de uma classe simples, cuja função principal é inicializar a classe ***Leitor*** e coordenar o processamento dos arquivos de teste.

Por meio de um laço de repetição, a classe percorre todos os arquivos de teste, utilizando o método ***getLinhas()*** da classe ***Leitor*** para obter os comandos

presentes em cada arquivo. Em seguida, esses comandos são processados e executados por meio do método ***interpretador()***, também pertencente à classe ***Leitor***.

URL.java

Esta classe é responsável por representar e manipular uma URL, realizando a extração e o armazenamento de suas principais partes estruturais. Além disso, a é utilizada como um elemento da lista duplamente encadeada implementada na classe ***ListaURLS***, por meio das referências para o próximo e o elemento anterior.

A classe possui atributos que armazenam a URL completa, o protocolo, o host, o caminho e a profundidade do caminho. No momento da instanciação, o construtor recebe uma *String* contendo a URL completa e, a partir dela, inicializa automaticamente todos os demais atributos, chamando métodos específicos de extração para cada componente.

O método ***extrairProtocolo()*** é responsável por identificar e retornar o protocolo da URL, considerando o trecho anterior à sequência "://". Caso o protocolo não seja encontrado, o método trata a exceção e retorna null.

O método ***extrairHost()*** realiza a extração do host da URL. Para isso, remove inicialmente o protocolo, identifica o trecho correspondente ao domínio até a primeira barra e, se presente, remove o prefixo www.. Em caso de erro durante o processamento, o método retorna null.

O método ***extrairCaminho()*** é responsável por extrair o caminho da URL. Ele remove o protocolo e o host, identifica o início do caminho a partir da primeira barra e desconsidera parâmetros de consulta e fragmentos. Caso a URL não possua caminho explícito, o método retorna a raiz.

O método ***extrairProfundidade()*** calcula a profundidade do caminho da URL, ou seja, a quantidade de níveis existentes no path. Para isso, utiliza o caminho extraído, remove barras excedentes e contabiliza os segmentos separados por /. URLs cujo caminho seja apenas a raiz possuem profundidade zero.

Análise de Complexidade

A classe Escalonador concentra a maior parte da lógica do sistema e, consequentemente, os maiores custos computacionais.

ADD_URLS

Verifica duplicidade antes de inserir URLs.

Tempo: $O(n^2)$

Dois laços aninhados:

Externo: quantidade

Interno: até quantidade

Espaço: $O(n)$

Vetor auxiliar urls_unique com tamanho n

QTD_URLS_HOST

Percorre toda a lista de URLs contando as que pertencem ao host informado.

Tempo: $O(n)$

Percorre a lista uma única vez

Espaço: $O(1)$

Apenas variáveis auxiliares

LISTA_HOSTS

Gera a lista de hosts distintos.

Tempo: $O(n^2)$

Para cada URL, verifica duplicidade no vetor de hosts (n)

Espaço: $O(n)$

Vetor de hosts com tamanho máximo n

VER_HOST

Retorna URLs de um host específico ordenadas por profundidade.

Tempo: $O(n^2)$

Percorre toda a lista: $O(n)$

Ordenação por Bubble Sort:

No pior caso, $n \times n$

Espaço: $O(n)$

Vetores auxiliares de URLs e profundidades

ESCALONA_HOST

Escalona URLs de um host e remove da lista principal.

Tempo: $O(n^2)$

Chamada a **VER_HOST**: $O(n^2)$

Percorre lista novamente para remoção: $O(n \times \text{quantidade})$

No pior caso, quantidade $\approx n$:

Espaço: $O(n)$

Vetor de saída com tamanho quantidade

Métodos de Escalonamento Global

ESCALONA_TUDO_PROF, ESCALONA_PROF

Chamam:

LISTA_HOSTS → $O(n^2)$

QTD_URLS_HOST para cada host → $O(n \times h)$

ESCALONA_HOST para cada host → $O(h \times n^2)$

Como $h \leq n$:

$O(n^3)$

Espaço: $O(n^2)$

Vetores auxiliares e matrizes temporárias

ESCALONA_TUDO_LARG e ESCALONA_LARG

Além das operações acima:

Uso de matriz hosts × urls

Tempo: $O(n^3)$

Espaço: $O(n^2)$

ESCALONA_TUDO_BEST e ESCALONA_BEST

Seleção repetida do host com maior número de URLs

Para cada URL escalonada, percorre o vetor de hosts

Tempo: $O(n^3)$

Espaço: $O(n^2)$

Métodos de Limpeza

LIMPA_HOST

Percorre toda a lista

Tempo: $O(n)$

LIMPA_TUDO

Remove cada elemento individualmente

Tempo: $O(n)$

Espaço: $O(1)$

Conclusão

Neste trabalho foi desenvolvido e documentado um escalonador de URLs em Java, com o objetivo de organizar e definir a ordem de coleta de páginas a partir de diferentes estratégias de escalonamento, conforme especificado nas instruções do projeto.

Ao longo do desenvolvimento, foi possível consolidar conhecimentos sobre estruturas de dados, especialmente listas duplamente encadeadas, bem como sobre o processamento e manipulação de URLs, leitura e interpretação de arquivos de entrada e organização modular de um sistema orientado a objetos. Além disso, a implementação das estratégias de escalonamento *breadth-first*, *depth-first* e *best-first* permitiu compreender, na prática, como diferentes critérios de priorização impactam o desempenho e o comportamento do sistema. Por fim, a análise de complexidade contribuiu para o entendimento dos custos computacionais envolvidos, reforçando a importância de avaliar tempo e espaço na escolha de algoritmos e estruturas adequadas.

Apêndice

Compilação e Execução do Sistema

A compilação do sistema deve ser realizada de forma convencional, mantendo todos os arquivos-fonte na mesma pasta. Esse procedimento pode ser feito normalmente por meio do **Visual Studio Code (VS Code)** ou de qualquer outro ambiente de desenvolvimento compatível com Java.

Para a execução do sistema, deve-se executar a classe **Main.java**. Durante a execução, o programa solicitará ao usuário:

- o caminho do arquivo de teste (arquivo de entrada);

Os arquivos devem estar localizados na mesma pasta do sistema ou em uma de suas subpastas, conforme ilustrado nos exemplos apresentados.

É importante que o caminho informado seja relativo à pasta raiz do sistema e que sejam especificados corretamente tanto o nome do arquivo de entrada quanto o nome do arquivo de saída, incluindo obrigatoriamente a extensão **.txt**.

Exemplo: Caminho escrito durante execução

```
IsInExceptionMessages' '-cp' 'C:\Users\elian\AppData\Roaming\Code\User\w
Digite o caminho do arquivo de entrada seguido de seu nome
testes_TP2/teste5.txt
```

Exemplo: Estrutura de pastas

Nome	Status	Data de modificação	Tipo	Tamanho
teste1	✓	05/01/2026 16:42	Documento de Te...	1 KB
teste2	✓	05/01/2026 16:42	Documento de Te...	1 KB
teste3	✓	05/01/2026 16:42	Documento de Te...	1 KB
teste4	✓	05/01/2026 16:42	Documento de Te...	1 KB
teste5.out	✓	12/01/2026 21:47	Documento de Te...	1 KB
teste5	✓	05/01/2026 16:42	Documento de Te...	1 KB