Trabalho Prático 2 em Estruturas de Dados

Ordenação em Memória Extra

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 partir de 2015, entramos na chamada era do *zetabyte*, com geração de mais de 2,5 quintilhões de bytes de dados por dia. A robustez do volume de dados da população mundial on-line impõe desafios no que tange ao processamento, pesquisa e inferência estatística com essa grande base de dados, sendo um deles a busca por sites populares dentre os outros bilhões existentes na Web. Destarte, este trabalho implementará, utilizando algoritmos de ordenação como QuickSort, estruturas de dados como Heap (fila de prioridade) e, ainda, uma memória externa, aqui denominada fita, um programa que faz a busca em um arquivo (que simularia a grande base de dados da web) dos links que direcionam para as páginas mais acessadas, retornando-os.

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`). 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 Entidade**: esta classe tem como atributos uma string url, que representa a URL da página e numAcessos, que representa o número de acessos da página linkada em url. Ainda, há um atributo indiceFitaOrigem que é utilizado na etapa de intercalação de rodadas, para identificar de onde veio a entidade e, para isso, de onde será retirada a próxima (mais detalhes a seguir). Como métodos, temos getters e setters para os atributos pertinentes e métodos de impressão da entidade em arquivo, ou na saída padrão.

- **class Fita**: esta classe tem como atributos um vetor Entidade entidades, um inteiro numEntidades, que é o tamanho do vetor entidades e um inteiro incioVet, que sinaliza o índice da primeira entidade válida do vetor entidades. O método alocaVetorEntidades() é responsável por fazer alocação dinâmica do vetor entidades, de acordo com tamanho recebido como parâmetro. Também temos métodos getters e setters para os atributos pertinentes, além de métodos de impressão de todas as entidades da fita em arquivo ou na saída padrão. Temos, também, o método compara(), que visa mitigar possíveis empates no atributo numAcessos entre as entidades do vetor entidades, desempatando com a ordem alfabética do atributo url, o método PopFront(), que remove e retorna o primeira elemento do vetor entidades e Vazio(), que retorna valor Booleano de acordo com a situação do vetor Entidades (vazio ou não).

- **class Heap:** Esta é uma fila de prioridade, utilizada, conforme especificação, na etapa de intercalação de rodadas (detalhes das etapas na próxima seção). De acordo com a definição da estrutura de dados heap, esta classe implementa métodos que adicionam e removem elementos respeitando a especificação (Add(), Pop() e Refaz()). Além disso, há um método para impressão da situação atual do heap na saída padrão e um método Vazio() que retorna valor booleano de acordo com a situação do heap (vazio ou não). Detalhe é que o heap é composto de instâncias de Entidade e ordenado de acordo com o atributo numAcessos de cada uma. Como atributos, temos o vetor de Entidade vet, que é a representação simplificada do heap e tamVet, o tamanho de vet. Ainda, há o método compara(), que visa mitigar possíveis empates no atributo numAcessos entre as entidades do vetor vet, desempatando com a ordem alfabética do atributo url.

2.3 Funcionamento do Programa

A função main deve receber, **obrigatoriamente e nessa ordem**, os 3 argumentos a seguir:

1) Nome do arquivo de entrada;
2) Nome do arquivo de saída;
3) Número de entidades (variável denominada **numEntidades**) por fita (arquivos gerados na ETAPA 1 - mais detalhes a seguir);

Há, ainda, mais dois **argumentos opcionais**:

4) caminho/nome do arquivo de log (caso seja passado, o tempo de execução do programa é contado e registrado no arquivo);
5) `-l`, que ativa o registro de acesso (registra acessos aos endereços de memória no arquivo passado no argumento 5, além do tempo de execução do programa).

O programa principal é main.cpp. É nele que está a função main e é feita a leitura e processamento do arquivo de entrada passado na linha de comando, bem como a chamada das funções de acordo com cada etapa, definidas a seguir:

ETAPA 1: Geração de rodadas. Nesta etapa, são gerados

numFitas = M[1]/numEntidades + 1 (divisão inteira) arquivos não vazios,

denominados **fitas**, armazenados em formato .txt no diretório raiz deste projeto. As fitas são ordenadas de forma decrescente segundo o algoritmo QuickSort, cuja função foi implementada no programa principal. O **pivô do QuickSort** foi escolhido como a mediana do primeiro, do elemento central e do último elemento do vetor, objetivando mitigar escolhas que possam fazer com que haja o pior caso de execução, quando o pivô é o maior ou menor elemento do vetor.

ETAPA 2: Intercalação de rodadas. **Esta etapa somente ocorre caso sejam geradas mais de uma fita.** Caso ocorra, as fitas geradas na etapa anterior são intercaladas, utilizando a estrutura de dados Heap, implementada como uma classe. As entidades mais populares (com maior atributo numAcessos) são lidas do início de cada fita (já que foram ordenadas em ordem decrescente na etapa anterior) e colocadas no heap. A entidade com maior numAcessos do heap é escrita no arquivo de saída e uma outra entidade, tirada do início do mesmo arquivo da qual a última entidade veio (caso este não esteja vazio), é inserida no heap. O processo é repetido até que o heap fique vazio. Detalhe que as inserções e retiradas do heap, feitas através do respectivos métodos implementados na classe Heap, visam a manter a especificação desta estrutura. O resultado no arquivo de saída é que as entidades são ordenadas de forma decrescente de acordo com numAcessos, conforme desejado.

[1] Denotemos por **M** o número de entidades passadas no arquivo de entrada.

2.4 Funções do programa principal

Dentre as principais funções no programa principal, temos:

- contaLinhasArquivo: conta e retorna o número de linhas do arquivo recebido como parâmetro.
- geraEntidade: extrai e reconhece, de uma linha do arquivo de entrada, url e número de acessos de uma entidade, alocando-a.
- mediana: calcula e retorna a mediana de três entidades, com base em seu atributo numAcessos.
- Particao: método auxiliar ao QuickSort, que gera partições.
- Ordena: método recursivo auxiliar ao QuickSort, que chama Particao enquanto possível para o vetor original recebido como parâmetro.
- QuickSort: algoritmo de ordenação. Chama Ordena para o vetor recebido como parâmetro.
- ordenaFita: chama QuickSort com o vetor de entidades da fita recebida como parâmetro.
- transcreveFita: transcreve para o arquivo recebido como parâmetro todas as entidades da fita recebida como parâmetro.
- geraRodadas: gera as fitas ordenadas cada uma com um número de entidades recebido como parâmetro, com entidades lidas do arquivo de entrada recebido como parâmetro. A quantidade de fitas também é recebida como parâmetro.
- intercalaRodadas: intercala as entidades nas fitas geradas em geraRodadas através de um heap

2.5 Observações importantes

- **Este programa apenas funciona caso todos os parâmetros de entrada OBRIGATÓRIOS tenham sido passados na linha de comando e NA ORDEM CORRETA ESPECIFICADA.** São obrigatórios, conforme estabelecido na seção 2.3:

  caminho/nome do arquivo de entrada; caminho/nome do arquivo de saída e número de entidades por fita.

- Em caso de entidades com mesmo número de acessos, a impressão no arquivo de saída se dará de forma a seguir a ordem alfabética, analisando-se o atributo **url** (string) **TODO, inclusive o protocolo e extensões**. Exemplo de impressão no arquivo de saída:

  http://bcvkzvgbo.hgo.dna.com/bitor/ibednez/pdawlo/svt.html 144

  http://cocp.com/bpb/vsuluq/uvk.html 144

  http://mndq.kwag.com/uukwcibx/umenmeya/rmy/ajxlo.html 144

  Note que o desempate, nesse caso, se deu pela primeira letra após o protocolo

‘http://’, conforme destaque em vermelho.

Esse desempate é feito analisando-se entidades que possuem atributo numAcessos iguais tanto nas fitas, quanto no heap, desempatando pela ordem alfabética do atributo url.

3. Análise de Complexidade[2]

As análises que seguem levam em consideração três dimensões: o número de entidades em cada fita **(denotado n)**, o número de fitas geradas **(denotado m)** e o número de entidades no arquivo de entrada **(denotado j)**. Todas essas variáveis são recebidas como parâmetros de execução do programa. Em alguns casos, outras variáveis foram incluídas na notação assintótica. Em cada caso pertinente, estas variáveis estão descritas na coluna de comentários.

class Entidade				
Método(s)	**Tempo**	**Comentário**	**Espaço**	**Comentário**
Construtor	O(1)	Apenas inicialização de variáveis	O(1)	Não há alocações ou declarações.
Getters e Setters	O(1)	Apenas retornam ou alteram valores de variáveis já existentes.	O(1)	Não há alocações ou declarações.
ImprimeEntidade	O(1)	Impressão dos atributos url e numEntidades na saída padrão.	O(1)	Não há alocações ou declarações.
ImprimeEntidadeArquivo	O(1)	Impressão dos atributos url e numEntidades em arquivo.	O(1)	Não há alocações ou declarações.

Tabela 1 - Complexidade de tempo e espaço dos métodos da classe Entidade

class Fita				
Método(s)	**Tempo**	**Comentário**	**Espaço**	**Comentário**
Construtor	O(1)	Apenas inicialização de variáveis.	O(1)	Não há alocações ou declarações.
AlocaVetorEntidades	O(1)	Considerando a ordem de complexidade da função new constante, temos que a função também apresenta complexidade de tempo constante.	O(n)	Alocação de vetor de tamanho n.

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

Getters e Setters	O(1)	Apenas retornam ou alteram valores de variáveis já existentes.	O(1)	Não há alocações ou declarações.
ImprimeFita	O(n)	Laço com impressão das entidades da fita na saída padrão, executado n vezes.	O(1)	Apenas declarações de variáveis auxiliares.
ImprimeFitaArquivo	O(n)	Laço com impressão das entidades da fita em arquivo recebido como parâmetro, executado n vezes.	O(1)	Apenas declarações de variáveis auxiliares.
PopFront	O(1)	Apenas atribuições(O(1)) e decrementos(O(1)). Logo, complexidade O(1).	O(1)	Apenas declarações de variáveis auxiliares.
Compara	$O(n^2)$	Temos dois laços aninhados com comandos internos constantes (troca de elementos de um vetor). O primeiro laço tem variável de controle ‘i’ e o segundo, ‘j’. Note que, quando i=0, o segundo laço é executado n-1 vezes. Quando i=1, o segundo laço é executado n-2 vezes, até que quando i=n-1 o segundo laço não é executado. Sendo assim, essa função tem complexidade equivalente à soma dos n-1 primeiros inteiros, que é (n(n-1))/2. Logo, sua ordem de complexidade é quadrática.	O(1)	Apenas declarações de variáveis auxiliares.
IncrementaNumEntidades	O(1)	Apenas incremento de uma variável.	O(1)	Não há alocações ou declarações.
Vazia	O(1)	Cadeia de três condicionais com comandos constantes. Logo, complexidade constante.	O(1)	Não há alocações ou declarações.

Tabela 2 - Complexidade de tempo e espaço dos métodos da classe Fita

class Heap				
Método(s)	**Tempo**	**Comentário**	**Espaço**	**Comentário**
Construtor	O(1)	Alocação dinâmica de memória (O(1)) e atribuição de variáveis (O(1)). Logo, O(1).	O(k)	Seja k o parâmetro maxTam. Temos a alocação de vetor de tamanho k. Assim, a complexidade de espaço varia linearmente conforme k.
Refaz	O(log (k))	Considerando **k** o tamanho atual do heap. No pior caso, a função percorre todo um galho da árvore binária (heap), ou seja, executa log (k) operações.	O(1)	Apenas declarações de variáveis auxiliares.
Add	O(k log (k))	Considerando **k** o tamanho atual do heap. Para os nós internos (k/2 elementos), chama refaz, logo executa: k/2 log (k).	O(1)	Apenas declarações de variáveis auxiliares.
Pop	O(log (k))	Manipulação de variáveis (índices) - O(1); Chama Refaz - (O(log (k))). Logo, O(log (k)). Com k sendo o tamanho atual do heap.	O(1)	Apenas declarações de variáveis auxiliares..
Vazio	O(1)	Apenas verificação da variável tamVet e retorno de acordo.	O(1)	Não há alocações ou declarações.
ImprimeHeap	O(k)	Chama ImprimeEntidade (O(1)) para cada um dos k elementos do heap.	O(1)	Não há alocações ou declarações.
Compara	$O(k^2)$	Considerando **k** o tamanho atual do heap. Temos dois laços aninhados com comandos internos constantes (troca de elementos de um vetor). O primeiro laço tem variável de controle ‘i’ e o segundo, ‘j’. Note que, quando i=0, o segundo laço é executado k-1 vezes. Quando i=1, o segundo laço é executado k-2 vezes, até que quando i=k-1 o segundo laço não	O(1)	Apenas declarações de variáveis auxiliares.

		é executado. Sendo assim, essa função tem complexidade equivalente à soma dos k-1 primeiros inteiros, que é (k(k-1))/2. Logo, sua ordem de complexidade é quadrática.		

Tabela 3 - Complexidade de tempo e espaço dos métodos da classe Heap

main.cpp				
Método(s)	**Tempo**	**Comentário**	**Espaço**	**Comentário**
contaLinhasArquivo	O(k)	Considerando **k** o número de linhas do arquivo de entrada. Essa função possui um laço que, para cada linha do arquivo, faz um incremento em um contador inteiro. Logo, sua ordem de complexidade é linear com relação à k.	O(1)	Apenas declarações de variáveis auxiliares.
geraEntidade	O(1)	Considerando a ordem de complexidade das funções da biblioteca string O(1), temos, ademais, apenas atribuições e settagem de variáveis com métodos setters da classe Entidade, que são todos O(1). Logo, a complexidade é constante.	O(1)	Apenas declarações de variáveis auxiliares.
mediana	O(1)	Faz a ordenação de um vetor de 3 elementos apenas por meio de 2, e sempre 2, comparações. Após isso, retorna o elemento central (a mediana). Logo, complexidade constante.	O(1)	Declarações de variáveis auxiliares (O(1)) e declaração de um vetor de 3, e sempre 3, posições (O(1)). Logo, complexidade de espaço constante.

QuickSort	O(n log(n))	Essa análise diz respeito ao algoritmo QuickSort como um todo, isto é, já estão sendo levados em conta os custos das funções Particao e Ordena.O pior caso deste algoritmo seria se, sistematicamente, o pivô escolhido fosse o primeiro ou último elemento do vetor. Como, nesse caso, ele foi implementado escolhendo o pivô como "**mediana de três**", o pior caso nunca ocorre. Assim, ficamos com o caso médio que, de acordo com Sedgewick e Flajolet (1996, p. 17), é O(n log(n)). Sendo n conforme definido no início desta seção.	O(1)	Necessita de apenas uma pequena pilha como memória auxiliar.
ordenaFita	O(mn log(n))	Faz a chamada do QuickSort para todas as m fitas geradas na ETAPA 1. Assim, temos O(mn log(n)).	O(1)	Não há alocações ou declarações para este método propriamente dito.
transcreveFita	O(n)	Impressão das n entidades de uma fita em uma arquivo, utilizando os getters da classe Entidade para recuperação da url e numAcessos. Assim, a complexidade varia linearmente conforme n.	O(1)	Apenas declarações de variáveis auxiliares.
geraRodadas	$O(m^2 nlog(n))$	Gera n entidades para cada uma das m fitas usando dois laços encadeados, chamando a função geraEntidade, que é O(1), isto é, temos um custo até aqui da ordem de mn. Depois, há um laço que, para cada uma das m fitas, são chamados os métodos ordenaFita (O(mn log(n))), transcreveFita (O(m)) e Fita::Compara ($O(n^2)$). Assim, temos complexidade O{O(mn) + [O(m*(O(mn log(n)) + O(m) + $O(n^2)$))]} = $O(m * max(\ O(mnlog(n)),\ O(n^2)\ )$	O(1)	Apenas declarações de variáveis auxiliares.

intercalaRodadas	O(mnk^2)	Primeiramente há um laço executado segundo o número m de fitas que chama as funções Fita::PopFront (O(1)) e Heap::Add (O(k log (k)), sendo k o número de entidades atual no heap). Logo, até aqui temos O(mk log(k)). Em seguida, há um laço executado da ordem de mn vezes com chamadas para Heap::compara ($O(k^2)$), Entidade::ImprimeArquivo (O(n)), Heap::Add (O(k log (k))) e algumas outras funções cuja ordem de complexidade é constante. Logo, temos O { [O(mk log(k))] + [O(O(mn)*(O(k^2)+O(n)+O(k log (k)))]} = O(mnk^2).	O(1)	Apenas declarações de variáveis auxiliares.

Tabela 4 - Complexidade de tempo e espaço dos métodos 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.

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

5. Conclusão

Este trabalho teve como objetivo a implementação de um ordenador de URLs utilizando memória externa (fitas). Além do reforço, sempre importante, aos conhecimentos da construção, projeção e compromisso das estruturas de dados necessárias, a implementação deste trabalho me permitiu vislumbrar aspectos relacionais no que diz respeito ao algoritmo de ordenação QuickSort (na escolha do pivô) e à utilização do heap. Pude concluir que há uma linha tênue entre desempenho e complexidade de implementação (e muitas outras variáveis como estabilidade, adaptabilidade e outras) no que diz respeito aos inúmeros algoritmos de ordenação disponíveis. O que está em evidência é a escolha mais adequada.

6. 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: 18 de janeiro de 2022.

Big Data. **Wikipedia**, 2021.

Disponível em: <https://pt.wikipedia.org/wiki/Big_data>.

Acesso em: 18 de janeiro de 2022.

APÊNDICE A
Instruções de Compilação e Execução

Com a utilização de um arquivo 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/main <argumentos obrigatórios> <possíveis argumentos opcionais>[3]
```

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.

[3] Mais detalhes sobre os argumentos obrigatórios e opcionais devem ser vistos na seção 2.3.