Trabalho Prático 2:

Ordenação em Memória Externa

Erick Sunclair Santos Batista - 2020026877

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

ericksunclair@ufmg.br

1. Introdução

Atualmente uma quantidade enorme de dados é produzida diariamente e recorrentemente precisamos armazenar e organizar esses dados. Em alguns casos o volume de dados é tão grande que nem conseguimos armazenar estes dados em nossa memória principal (por exemplo a memória RAM de um computador pessoal). Tendo isso em vista, neste trabalho vamos desenvolver um sistema de ordenação utilizando fitas como memórias secundárias para armazenamento temporário dos dados auxiliando na sua ordenação.

O sistema desenvolvido recebe o nome de um arquivo de entrada que vai possuir n URLs com o número de acessos de cada endereço e retorna um arquivo de saída com essas URLs ordenadas por número de acessos (do menor para o maior número de acessos). No entanto, esse programa também recebe um inteiro que representa o número máximo de URLs que a memória consegue armazenar simultaneamente. O sistema ordena esses dados respeitando esse limite.

A seção 2 aborda as características do código implementado e as configurações usadas para testar o programa. A seção 3 apresenta uma análise de complexidade de tempo e espaço do programa. A sessão 4 possui uma análise experimental dos resultados obtidos nos testes feitos no programa. Por fim, a seção 5 possui a conclusão do trabalho desenvolvido.

2. Método

O programa foi desenvolvido na linguagem C++, compilado pelo compilador G++ da GNU Compiler Collection e os testes foram realizados em um computador com Windows 10, intel core i5 3.4 GHz e 8GB de memória RAM.

O sistema é composto de um tipo base entidade que vai armazenar a URL, o seu número de acessos e um índice representando a fita de qual essa entidade veio (se ela tiver vindo de alguma), um tipo abstrato fita, responsável por armazenar temporariamente um número máximo de entidades (URLs e seus respectivos números de acessos) e um tipo abstrato heap, que vai armazenar também um número máximo de entidades temporariamente, porém, enquanto cada fita armazena até o número máximo de URLs enviado como parâmetro de entrada ao programa, o heap armazena até o número total de fitas que forem criadas, pois ele inicialmente pega um item de cada fita, o que será explicado a seguir.

Tomando o número máximo de entidades que a memória pode armazenar simultaneamente como sendo um valor “n”, a primeira etapa do sistema consiste em salvar n entidades do arquivo de entrada em uma fita, organizar essa fita usando o método de ordenação quicksort e imprimir a fita ordenada em um arquivo temporário de saída chamado “rodada-m.txt”, sendo m o número da fita/rodada. Isso é feito até que o arquivo

de entrada seja totalmente lido. Com isso serão geradas total_urls/n (arredondando o resultado para cima) fitas, sendo total_urls o número total de entidades no arquivo de entrada e n o número máximo de entidades por rodada. Caso total_urls não seja múltiplo de n a última fita não vai ser cheia, já que ela vai conter apenas os itens que sobrarem.

Após a criação das rodadas a segunda etapa do programa é utilizar o heap para organizar os dados de todas as fitas e imprimir para o arquivo de saída. Primeiramente o heap pega e armazena o primeiro elemento de cada fita (o menor dessa fita, já que as fitas já foram organizadas anteriormente). Em seguida o heap vai imprimir o seu menor valor e pegar o próximo valor da fita que continha o item que acabou de imprimir, e vai fazer isso até que todas as fitas sejam completamente lidas e o heap esvazie. Ao final desse processo o arquivo de saída vai conter todas as entidades do arquivo de entrada organizadas em ordem crescente de número de acessos.

3. Análise de Complexidade

3.1. Complexidade Temporal

Com relação à complexidade temporal temos o tipo entidade com as funções: set_url, set_num_acessos, set_ind_rodada, get_url, get_num_acessos, get_ind_rodada e a sobrecarga do operador "=", além do seu próprio construtor, sendo todas essas fuções responsáveis por apenas atribuir valores ou retornar valores, por isso todas têm complexidade O(1).

Já o tipo heap possui seu construtor que é um método O(1), o método retira é O(n), onde n é o número de entidades salvas no momento no heap. Além disso, os métodos ordena_entidades, ordena e particao juntos possuem complexidade no melhor caso de O(nlog(n)) e no pior caso de O(n^2), que é a complexidade do método quicksort, já que esssa funções apenas implementam esse método de ordenação. O caso médio para essas funções (juntas) também é O(nlog(n)).

O tipo fita possui as três mesma funções do tipo heap para ordenação quicksort, com a mesma complexidade, e possui um construtor O(1) e as funções le_entidades e escreve_entidades que são O(n), sendo n o número de entidades que a fita vai ler ou escrever.

3.2. Complexidade Espacial

Sobre a complexidade temporal as funções construtoras do heap e da fita alocam um espaço na memória de num_fitas e num_entidades, respectivamente, portanto o construtor do heap possui complexidade espacial O(n), sendo n o número total de fitas criadas, e o construtor da fita tem complexidade espacial O(m), sendo m o número máximo de entidades por fita.

Todas as funções do tipo entidade e a função adiciona (do heap) não criam variáveis portanto possuem complexidade espacial O(0). Todas as outras funções do código tem complexidade O(1) pois apenas criam variáveis simples.

4. Análise Experimental

Para análise de desempenho foram feitos quatro testes diferentes. No primeiro foram armazenados e organizados 500 URLs sendo permitidos 7 URLs por fita. Os resultados podem ser vistos na Imagem 4.1. Na

Imagem 4.2 temos o resultado para os mesmos 500 URLs, porém com 70 URLs por fita. Percebe-se que no caso de 70 entidades por fita, em que se faz necessária a criação de menos fitas, o tempo de execução caiu pela metade e o tempo gasto em cada função foi tão pequeno que não foi registrado.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 50.00      0.01     0.01                             entidade::operator=(entidade const&)
 50.00      0.02     0.01                             entidade::get_num_acessos() const
```

Imagem 4.1 – Resultado para 500 URLs e 7 URLs por fita

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated
```

Imagem 4.2 – Resultado para 500 URLs e 70 URLs por fita

Na Imagem 4.3 vemos o resultado para 5000 URLs divididos em 22 por fita e na Imagem 4.4 em 250 por fita. Nota-se que o tempo para 250 URLs por fita é consideravelmente menor, já que precisam ser produzidas menos fitas. Inclusive, no caso em que 500 URLs são divididas em fitas de 7 (totalizando 72 fitas) o tempo gasto ainda é maior (0.02 segundos) do que no caso de 5000 URLs serem divididas em fitas de 250 (totalizando 20 fitas), em que o tempo gasto é de 0.01 segundos, mesmo que o número de URLs a serem organizadas seja 10 vezes maior. Com isso percebemos que o número de fitas que criamos influencia mais no aumento do tempo de execução do que o aumento no número total de URLs.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 54.55      0.06     0.06                             entidade::get_num_acessos() const
 18.18      0.08     0.02                             heap::particao(int, int, int*, int*, entidade*)
 18.18      0.10     0.02                             entidade::operator=(entidade const&)
  9.09      0.11     0.01                             entidade::get_url[abi:cxx11]() const
```

Imagem 4.3 – Resultado para 5000 URLs e 22 URLs por fita

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.00      0.01     0.01                             entidade::get_num_acessos() const
```

Imagem 4.4 – Resultado para 5000 URLs e 250 URLs por fita

Outra informação que podemos tirar desses resultados é que a função get_num_acessos é a que mais gasta tempo em quase todos os testes, provavelmente por essa função ser chamada muitas vezes, apesar de ser uma função simples (apenas retorna o valor do número de acessos de uma entidade).

5. Conclusão

Neste trabalho foi criado em C++ um ordenador em memória externa baseado em fitas, um heap e o método de ordenação eficiente quicksort e foram feitos perfis de análise de custo e desempenho do programa e de suas funções separadamente.

Com os teste realizados pudemos perceber como o método quicksort é realmente rápido. Além disso, podemos analisar quais fatores mais influenciam no desempenho do sistema, ficando claro que quanto mais fitas temos que criar mais demorado é o programa, então o aumento do tempo de execução se deve menos ao aumento específico do número de URLs e mais à diminuição do número máximo de URLs por fita, o que faz com que mais rodadas tenham que ser geradas e aumenta mais consideravelmente o tempo de programa.

Com esse trabalho foi possível aprender e se aprofundar em conceitos muito importantes como métodos de ordenação, Quicksort, Heap, ordenações externa e interna e como contornar a falta de memória em um dispositivo quando precisamos armazenar e organizar grande quantidade de dados.

Bibliografia

SMARTI. **ALGORITMOS DE ORDENAÇÃO E ANÁLISE DE COMPLEXIDADE**. Disponível em: https://smarti.blog.br/algoritmos-de-ordenacao/. Acesso em: 26 jan. 2022.

TREINAWEB CURSOS. **Conheça os principais algoritmos de ordenação**. Disponível em: https://www.treinaweb.com.br/blog/conheca-os-principais-algoritmos-de-ordenacao. Acesso em: 26 jan. 2022.

WIKIPEDIA. **Heap**. Disponível em: https://pt.wikipedia.org/wiki/Heap. Acesso em: 26 jan. 2022.

WIKIPEDIA. **Quicksort**. Disponível em: https://pt.wikipedia.org/wiki/Quicksort. Acesso em: 26 jan. 2022.

Apêndice A - Instruções de Compilação e Execução

Para executar o programa recomenda-se usar o Makefile disponibilizado na matriz do diretório do programa pois ele já compila, linka e executa um programa para o arquivo de entrada. Para alterar o nome do programa de entrada, o nome do programa de saída e o número máximo de entidades por fita/rodada basta editar a linha 26 do Makefile, sendo o primeiro argumento o nome do arquivo de entrada, o segundo argumento o nome do arquivo de saída e o terceiro argumento o número máximo de entidades. O arquivo de entrada deve estar na matriz do diretório do programa e o arquivo de saída também será gerado na matriz. Uma observação importante: para o correto funcionamento do programa o arquivo de entrada não deve ter espaços ou quebras de linha (“/n”) ao fim do arquivo. Para executar deve-se seguir os seguintes passo:

- Utilizando um terminal acesse o diretório “/Erick_Sunclair_2020026877/TP”.
- Execute o arquivo Makefile utilizando o seguinte comando: “make all”.
- Será criado na matriz um arquivo de saída com o nome enviado como parâmetro ao programa. Esse arquivo vai conter os mesmos dados do arquivo de entrada, porém organizados em ordem crescente de número de acessos.
- Para apagar os objetos criados na compilação do programa basta executar no terminal outro comando do arquivo Makefile: “make clean”.
- E podemos executar novamente o programa.

Os arquivos das rodadas são criados no diretório “/rodadas” mas eles não atrapalham em nada na execução de futuros programas, e, como são arquivos temporários, eles não possuem importância com relação ao resultado final.