Universidade Federal de Ouro Preto
Departamento de Computação – DECOM
Disciplina: BCC203 - Estrutura de Dados II
Professor: Guilherme Tavares de Assis
Resumo II – Ordenação Externa

Turma: 11
Nome: Pedro Lucas Damasceno Silva
Matrícula: 20.1.4003

- **INTRODUÇÃO**

Além das métricas vistas na análise de complexidade de algoritmos de ordenação em memória interna (número de comparações e operações de permutação), para a ordenação externa, assim como para a pesquisa externa, há também o fator de transferência entre as memórias, sendo este o principal responsável pela eficiência de um algoritmo. Na ordenação, todavia, ao contrário da pesquisa na qual se davam apenas da memória externa para a interna, as transferências ocorrem em ambas as direções e devem ser sempre contabilizadas. É importante ressaltar que dado o alto custo de acesso a memória secundária e as possíveis restrições de acesso inerentes a alguns dispositivos, a eficiência dos métodos de ordenação externa são diretamente dependentes da tecnologia atual.

Semelhante à partição de vetores utilizada nos algoritmos de ordenação de memória interna, a intercalação (combinar dois ou mais blocos ordenados em um) é o método mais importante de ordenação externa. É uma operação custosa, pois requer a transferência de dados, e os algoritmos de ordenação sempre visam reduzir a frequência com que é efetuada. Quanto menor o número de vezes que um mesmo item é lido ou escrito, melhor a eficiência do algoritmo.

Portanto, a estratégia geral dos métodos de ordenação externa é:

1. Dividir o arquivo em blocos que caibam em memória interna
2. Ordenar cada bloco
3. Intercalar os blocos ordenados, criando blocos cada vez maiores até que todo o arquivo esteja ordenado

- **INTERCALAÇÃO BALANCEADA DE VÁRIOS CAMINHOS**

1. Requerimentos
   a. Memória interna suficiente para comportar a divisão dos blocos
   b. Unidades de fita magnética suficientes para a implementação, devendo ser a maior quantidade possível

2. Criação dos blocos ordenados – nessa etapa, toda a extensão do arquivo é lida apenas uma vez.
   a. Ler o arquivo de acordo com a divisão de blocos estabelecida
   b. Transferir cada bloco à memória principal
   c. Ordenar os elementos
   d. Retornar os elementos às fitas de entrada

3. Intercalação – nessa etapa, blocos maiores (já ordenados) são criados a partir dos menores originados da etapa anterior.
   a. Criação de um vetor contendo os primeiros registros de cada fita
   b. Iteração no vetor para determinar o menor elemento
   c. Transferência desse menor elemento para uma fita de saída
   d. Inserção do próximo elemento da fita à qual pertencia o elemento transferido. Uma vez que uma fita esteja vazia, a parte do vetor correspondente é inutilizada
   e. Reinício das instruções (a – d) até que a intercalação esteja completa e todos os primeiros blocos ordenados sejam completamente trilhados
   f. Repetição até que todos os blocos sejam intercalados e seus elementos estejam devidamente inseridos nas fitas de saída
   g. Utilização da mesma estratégia para ordenar as fitas obtidas anteriormente para ordená-las e criar um único bloco completamente ordenado

4. Observações
   a. A etapa de criação dos blocos e cada intercalação implicam em uma leitura completa sobre o arquivo.
   b. A ordem de complexidade pode ser obtida por:

$P(n) = \log_f (n/m)$
P(n) – número de leituras durante a intercalação f – número de fitas utilizadas em cada leitura n – número de registros m – número de registros cabíveis em memória interna

   c. É possível implementar de forma a utilizar apenas uma fita de saída. Isso implica em uma etapa adicional de leitura a cada intercalação, mas em alguns casos pode ser mais eficiente pois uma quantidade maior de fitas de entrada implica em menos blocos para cada, diminuindo a quantidade necessária de intercalações. É importante ressaltar que a melhor implementação é subjetiva e depende diretamente das circunstâncias nas quais será utilizada.
   d. Quando há memória principal disponível, é possível realizar a leitura de um bloco inteiro durante a intercalação. Desta forma, ao invés de ler item por item, ganhamos eficiência ao economizar leituras em dispositivos de memória secundária.

5. Implementação por meio de Substituição por Seleção

   A determinação do menor elemento do vetor durante a intercalação pode ser feita substituindo a iteração mencionada anteriormente pela implementação de um *heap*. Dessa forma, ao construir um *heap* mínimo, o menor elemento estaria sempre na raiz, e, uma vez removido e outro elemento do bloco inserido, bastaria reconstituir a propriedade do *heap*. Haja vista a complexidade logarítmica de reconstituição do *heap*, sua eficiência é comprovadamente maior, principalmente em vetores grandes, se comparada à iteração simples de complexidade linear. Todavia, é preciso armazenar também em memória primária a referência de qual fita o elemento pertence.

Ademais, o *heap* também pode ser utilizado para gerar blocos iniciais ordenados de forma mais eficiente. Nesse caso, quando um elemento é retirado do vetor utilizado para preencher as fitas de entrada, deve-se observar se o seu substituto é maior ou menor do que ele. Caso seja menor, esse elemento é marcado e considerado maior que os outros presentes na estrutura; se for maior, nada acontece e o *heap* é reconstituído normalmente. Uma vez que todos os elementos presentes no vetor estejam marcados, todos são desmarcados. Ao final do processo, é possível obter blocos ordenados maiores e de tamanhos diferentes, o que é muito positivo para o processo de intercalação.

Nessa implementação, caso um arquivo já ordenado crescentemente seja fornecido, apenas uma fita será preenchida e a fase de intercalação é dispensada. Já no caso de um arquivo decrescentemente (pior caso), obteremos blocos de tamanhos regulares em um resultado idêntico ao método de ordenação interna visto anteriormente. Também é possível utilizar o sistema de paginação para diminuir a necessidade de acessos à memória secundária.

A substituição por seleção é vantajosa a partir de 8 ou mais fitas. Para poucas fitas o método não é ideal já que o menor item é obtido por f-1 comparações. De forma geral, é desejável que o número de fitas seja o maior possível para completar a ordenação em poucos passos.

- **INTERCALAÇÃO POLIFÁSICA**

Elaborada com a intenção de contornar problemas da intercalação balanceada de vários caminhos (necessidade de muitas fitas e custo de cópia adicional do arquivo), a intercalação polifásica parte dos blocos iniciais ordenados. Esses blocos são distribuídos de forma desigual entre as fitas disponíveis com exceção de uma que é deixada livre. Em seguida, a intercalação é executada até que uma das fitas de entrada se esvazie e a fita vazia torna-se a próxima fita de saída.

Ao final do processo de intercalação, todos os elementos estão presentes em apenas um bloco de uma fita. Também é possível que uma fita contenha todos os elementos, mas mais de um bloco (pior caso, pode ser em decorrência de blocos distribuídos de maneira igualitária). Nesse caso, apenas uma fita será obtida após as intercalações iniciais, mas essa fita possuirá mais de um bloco ordenado, e sendo a única, impossibilita que o processo de intercalação continue. Para tratar esse caso, um bloco é copiado para outra fita para que possa ser intercalado com os outros.

Para valores menores de f, a intercalação polifásica se mostra ligeiramente melhor do que a intercalação balanceada. Para valores de f > 8, a intercalação balanceada de vários caminhos pode ser mais rápida.

- ***QUICKSORT* EXTERNO**

Assim como o *quicksort* interno, o externo também utiliza o método de divisão e conquista, mas difere do interno quanto ao pivô (nesse caso é um conjunto de elementos em memória interna que ocupam $\log_n$ posições). Ao contrário dos outros métodos vistos anteriormente, é *in situ*, ou seja, despensa a utilização de dispositivos secundários.

O algoritmo particiona o arquivo a partir dos elementos pivô e gera dois subarquivos, sobre os quais o algoritmo será chamado recursivamente até que o subarquivo diminua o suficiente para caber em memória principal. Ao final do processo, o arquivo estará completamente ordenado.

- ‘subarquivo’ é uma partição dentro do próprio arquivo. Como o método é *in situ*, ele não necessita da criação de arquivos auxiliares.

- ❖ Para a partição do arquivo é necessária memória interna suficiente para j – i – 1 posições, sendo necessariamente ≥ 3. Caso o arquivo de entrada possua no máximo j – i – 1 registros, ele é ordenado em um único passo.
- ❖ Nas chamadas recursivas, o subarquivo de menor tamanho deve ser ordenado inicialmente e subarquivo vazios ou com apenas um registro são ignorados.

São utilizados dois ponteiros para leitura, um que parte do início e outro do final do arquivo. Ambos vão caminhando e, se identificada a necessidade de trocar elementos de lugar, outros dois ponteiros são utilizados para realizar essas gravações.

- ❖ O ponteiro de escrita nunca pode estar à frente do ponteiro de leitura.

O posicionamento dos ponteiros (Li, Ls, Ei, Es), as referências dos elementos do pivô (i*, j*) e os limites do pivô (Linf e Lsup, iniciados em -inf e inf) são determinados com o auxílio de variáveis em memória interna.

Partição

a. Os primeiros Área – 1 registros são lidos alternadamente dos extremos do pivô e armazenados na área de memória interna.
b. Ao ler Área-ésimo registro, cuja chave é C, comparamos a chave com Lsup e, sendo maior, j recebe Es e o registro é escrito no subarquivo 2. Caso contrário, C é comparada com Linf e, sendo menor, i recebe Ei e o registro é escrito no subarquivo 1. Caso Linf ≤ C ≤ Lsup, o registro é inserido na memória interna.
c. Para garantir que os apontadores de leitura e escrita estejam posicionados corretamente, a ordem alternada de leitura é interrompida se Li = Ei ou Ls = Es. Dessa forma, nenhum registro é sobrescrito e perdido.
d. Quando o pivô está completamente preenchido, deve-se remover um registro considerando os tamanhos atuais dos subarquivos. Nesse caso, o registro escolhido pode ser o menor (e escrito no subarquivo à esquerda) ou o maior (e escrito no subarquivo à direita) e os limites são atualizados mediante ao elemento removido. Se o subarquivo à esquerda for menor, então o menor elemento será removido e vice-versa.
e. Ao final do processo (quando Li e Ls se cruzam), i e j indicam os limites dos subarquivos (o primeiro com elementos menores que o pivô e o segundo com elementos maiores) e o próprio pivô, já ordenado, entre eles.
f. O processo é reinicializado em uma chamada recursiva sobre os subarquivos.

Análise

O melhor caso ocorre quando o arquivo já está ordenado, e a ordem de complexidade é dada por:

O (número de registros / tamanho do bloco de leitura)

O pior caso ocorre quando as partições geradas possuem tamanhos inadequados (maior tamanho possível e vazio). Nesse caso, à medida que o número de registros cresce, a probabilidade de ocorrência do pior caso tende a zero. A ordem de complexidade, nesse caso, é dada por:

$$O\ (\text{número de registros}^2 / \text{tamanho da área})$$

Por fim, o caso médio, e mais provável, possui a ordem de complexidade definida por:

$$O\ (n / b \times \log (n / \text{área}))$$

Sendo ‘n’ o número de registros e ‘b’ o tamanho do bloco de leitura ou gravação do sistema operacional.