

Processamento Cosequencial e Ordenação de Arquivos Grandes



Estrutura de Dados II

Slides cedidos pelo Prof. Gustavo Batista (ICMC-USP)
[<http://www.icmc.usp.br/~gbatista/>]



Operações Cosequenciais

- Consistem em operações que envolvem o processamento coordenado (simultâneo) de duas ou mais listas de entrada sequenciais, de modo a produzir uma única lista como saída
- Exemplo: merging (intercalação) ou matching (intersecção) de duas ou mais listas

Modelo para implementação de processos cosequenciais

- Considere as duas listas ao lado

Considerar que não existem chaves duplicadas e que as listas estão em ordem ascendente (ordenadas)

<i>List 1</i>	<i>List 2</i>
ADAMS	ADAMS
CARTER	ANDERSON
CHIN	ANDREWS
DAVIS	BECH
FOSTER	BURNS
GARWICK	CARTER
JAMES	DAVIS
JOHNSON	DEMPSEY
KARNS	GRAY
LAMBERT	JAMES
MILLER	JOHNSON
PETERS	KATZ
RESTON	PETERS
ROSEWALD	ROSEWALD
TURNER	SCHMIDT
	THAYER
	WALKER
	WILLIS

FIGURE 7.1 Sample input lists for cosequential operations.



Modelo para implementação de processos cosequenciais

- O algoritmo (matching):
 - lê um nome de cada lista e compara
 - se ambos são iguais, copia o nome para a saída e avança para o próximo nome da lista em cada arquivo
 - Se o nome da Lista1 é menor então avança na Lista1
 - Se o nome da Lista2 é menor então avança na Lista2



Pontos importantes a serem considerados pelo algoritmo

- Inicialização:
 - Como abrir os arquivos e inicializar as informações para o processo funcionar corretamente
- Sincronização:
 - Como avançar adequadamente em cada arquivo
- Gerenciamento de condição de fim-de-arquivo:
 - O processo deve parar ao atingir o fim de uma das listas
- Reconhecimento de erros:
 - Nomes duplicados ou fora de ordem



Matching

FIGURE 7.2 Cosequential match procedure based on a single loop.

PROGRAM: match

call initialize() procedure to:

- open input files LIST_1 and LIST_2
- create output file OUT_FILE
- set MORE_NAMES_EXIST to TRUE
- initialize sequence checking variables

call input() to get NAME_1 from LIST_1

call input() to get NAME_2 from LIST_2

while (MORE_NAMES_EXIST)

if (NAME_1 < NAME_2)

call input() to get NAME_1 from LIST_1

else if (NAME_1 > NAME_2)

call input() to get NAME_2 from LIST_2

else /* match -- names are the same */

write NAME_1 to OUT_FILE

call input() to get NAME_1 from LIST_1

call input() to get NAME_2 from LIST_2

endif

endwhile

finish_up()

end PROGRAM



Matching (Intersecção)

```
PROCEDURE: initialize()

arguments used to return values:
PREV_1, PREV_2    : previous name variables for the 2 lists
LIST_1, LIST_2    : file descriptors for input files to be used
MORE_NAMES_EXIST : flag used by main loop to halt processing

/* set both the previous_name variables (one for each list) to
   a value that is guaranteed to be less than any input value */
PREV_1 := LOW_VALUE
PREV_2 := LOW_VALUE

open file for List 1 as LIST_1
open file for List 2 as LIST_2
create file for output as OUT_FILE
if (both open statements succeed)
    MORE_NAMES_EXIST := TRUE

end PROCEDURE
```

FIGURE 7.4 Initialization procedure for cosequential processing.

Matching (Interpcc30)

FIGURE 7.3 Input routine for match procedure.

```
PROCEDURE: input() /* input routine for MATCH procedure */

input arguments:
  INP_FILE          : file descriptor for input file to be used
                     (could be LIST_1 OR LIST_2)
  PREVIOUS_NAME     : last name read from this list

arguments used to return values:
  NAME              : name to be returned from input procedure
  MORE_NAMES_EXIST  : flag used by main loop to halt processing

  read next NAME from INP_FILE

  /* check for end of file, duplicate names, names out of order */
  if (EOF)
    MORE_NAMES_EXIST := FALSE /* set flag to end processing */

  else if (NAME <= PREVIOUS_NAME)
    issue sequence check error
    abort processing
  endif

  PREVIOUS_NAME := NAME

end PROCEDURE
```




Matching (Intersecção)

- O procedimento cosequencial para intersecção é baseado num único loop, que controla a continuação do processo através de um flag (solução simples e eficiente)
- Dentro do loop tem-se um comando condicional triplo para testar cada uma das condições, e o controle volta ao loop principal a cada passo da operação



Matching (Intersecção)

- O **procedimento principal** não se preocupa com EOF ou com erros na sequência de nomes
- Para garantir uma lógica clara ao procedimento principal, esta tarefa foi deixada para a função **input()**, que verifica a ocorrência de fim de arquivo, bem como a ocorrência de nomes duplicados ou fora de ordem



Matching (Intersecção)

- Observe que a função **initialize()**:
 - abre arquivos de entrada e de saída;
 - seta a flag MORE_NAMES_EXIST como TRUE;
 - inicializa as variáveis que armazenam o nome anterior (uma para cada lista, denominadas PREV_1 e PREV_2) para um valor que garantidamente é menor que qualquer valor de entrada (LOW_VALUE).
 - Isso garante que a função **input()** não precisa tratar a leitura dos 2 primeiros registros como um caso especial



Merging (Intercalação)

- Modelo anterior pode ser facilmente estendido para a intercalação (união) de 2 listas, gerando como saída uma única lista ordenada (sem repetições de nomes):
 - Dois nomes, um de cada lista são comparados, e um nome é gerado na saída a CADA passo do comando condicional;
 - O algoritmo compara os nomes e manda para a saída o dado menor, avançando no arquivo de onde saiu esse dado;
 - O processamento continua enquanto houver nomes em uma das listas. Ambos os arquivos de entrada devem ser lidos até o fim (OTHER_LIST_NAME).



Merging

```
PROGRAM: merge

  call initialize() procedure to:
    - open input files LIST_1 and LIST_2
    - create output file OUT_FILE
    - set MORE_NAMES_EXIST to TRUE
    - initialize sequence checking variables

  call input() to get NAME_1 from LIST_1
  call input() to get NAME_2 from LIST_2

  while (MORE_NAMES_EXIST)

    if (NAME_1 < NAME_2)
      write NAME_1 to OUT_FILE
      call input() to get NAME_1 from LIST_1

    else if (NAME_1 > NAME_2)
      write NAME_2 to OUT_FILE
      call input() to get NAME_2 from LIST_2

    else /* match -- names are the same */
      write NAME_1 to OUT_FILE
      call input() to get NAME_1 from LIST_1
      call input() to get NAME_2 from LIST_2
    endif
  endwhile
  finish_up()

end PROGRAM
```

FIGURE 7.5 Cosequential merge procedure based on a single loop.

FIGURE 7.6 Input routine for merge procedure.

```
PROCEDURE: input() /* input routine for MERGE procedure */

input arguments
  INP_FILE          : file descriptor for input file to be used
                    : (could be LIST_1 OR LIST_2)
  PREVIOUS_NAME     : last name read from this list
  OTHER_LIST_NAME    : most recent name read from the other list

arguments used to return values:
  NAME              : name to be returned from input procedure
  MORE_NAMES_EXIST  : flag used by main loop to halt processing

  read next NAME from INP_FILE

  if (EOF) and (OTHER_LIST_NAME == HIGH_VALUE)
    MORE_NAMES_EXIST := FALSE /* end of both lists */

  else if (EOF)
    NAME := HIGH_VALUE /* just this list ended */

  else if (NAME <= PREVIOUS_NAME) /* sequence check */
    issue sequence check error
    abort processing
  endif

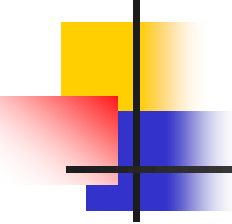
  PREVIOUS_NAME := NAME

end PROCEDURE
```



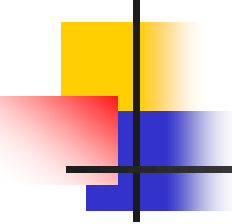
Merging (Intercalação)

- Mudanças na função **input()**:
 - Mantém a flag MORE_NAMES_EXIST em TRUE enquanto existirem entradas em qualquer um dos arquivos
 - Entretanto, não fica lendo do arquivo que já terminou: solução: seta NAME (1 ou 2, dependendo do arquivo que terminou antes) para um valor que não pode ocorrer como entrada válida, e que seja maior em valor que qualquer entrada válida (definida na constante HIGH_VALUE)
 - Usa também OTHER_LIST_NAME para a função saber se a outra lista já terminou



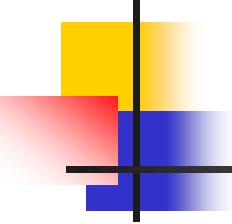
Resumo do Modelo Cosequencial

- Suposições feitas e comentários
 - Dois ou mais arquivos de entrada devem ser processados simultaneamente para produzir um ou mais arquivos de saída
 - Cada arquivo é ordenado sobre um ou mais campos chave, e todos os arquivos são ordenados do mesmo modo sobre esses campos. Não é necessário que as estruturas dos registros sejam iguais em todos os arquivos



Resumo do Modelo Cosequencial

- Suposições feitas e comentários
 - Deve existir um "valor alto" para a chave, que seja maior do que qualquer chave válida, e um "valor baixo" que seja menor do que qualquer chave válida
 - Registros são processados de acordo com a ordem lógica de ordenação. A ordem física não é relevante ao modelo, mas na prática pode ser importante para a implementação; a ordenação física pode ter grande impacto na eficiência do processamento



Resumo do Modelo Cosequencial

- Suposições feitas e comentários
 - Para cada arquivo existe um único registro corrente, cuja chave é aquela disponível no loop de sincronização principal. O modelo não proíbe que se olhe para a frente ou para trás no arquivo, desde que feito fora do loop e sem afetar a estrutura de sincronização
 - Registros podem ser manipulados somente em memória principal. Um programa não pode alterar um registro diretamente no arquivo em disco



Intercalação em K-vias

- A maior parte das aplicações de processamento cosequencial requer mais do que 2 arquivos de entrada
- Pode-se realizar a intercalação de K arquivos de entrada para gerar um único arquivo de saída sequencialmente ordenado
- K é chamado de ordem de uma intercalação em K-vias (**Multway Merging**)



Intercalação em K-vias

- No algoritmo que faz a intercalação de duas listas, a operação de intercalação pode ser vista como contendo os seguintes passos:
 - Decidir qual dos dois nomes disponíveis na entrada tem o menor valor;
 - Enviar este nome para o arquivo de saída;
 - Avançar no arquivo que possua o menor nome;
 - Caso as entradas em ambas as listas sejam iguais, avançar em ambos os arquivos.



Intercalação em K-vias

- A principal modificação deve ser realizada no laço de sincronização
- O laço de sincronização precisa encontrar o menor valor entre um conjunto de valores, copiar para o arquivo de saída e ir para o próximo item
- No caso de itens duplicados em mais de uma lista, deve-se ir para o próximo item em cada uma delas



Intercalação em K-vias

- Dada uma função **min()**, que retorna o menor valor de um conjunto de nomes, não existe razão para restringir o número de arquivos de entrada a 2
- O procedimento utilizado pode ser estendido para manipular 3 ou mais listas, como mostrado a seguir



Intercalação em K-vias

```
while (MORE_NAMES_EXIST)

    OUT_NAME = min(NAME_1, NAME_2, NAME_3, ... NAME_K)
    write OUT_NAME to OUT_FILE

    if (NAME_1 == OUT_NAME)
        call input() to get NAME_1 from LIST_1

    if (NAME_2 == OUT_NAME)
        call input() to get NAME_2 from LIST_2

    if (NAME_3 == OUT_NAME)
        call input() to get NAME_3 from LIST_3
    .
    .
    if (NAME_K == OUT_NAME)
        call input() to get NAME_K from LIST_K

endwhile
```

FIGURE 7.16 K-way merge loop, accounting for duplicate names.



Intercalação em K-vias

- A parte cara desse processamento é a série de testes para verificar em quais listas o nome ocorreu, de forma a determinar de quais listas os próximos novos valores devem ser lidos
- Como o OUT_NAME pode ter ocorrido em várias listas, cada um dos testes deve ser executado em cada ciclo do loop



Intercalação em K-vias

- Vamos supor que existe um vetor de itens (chaves) e um vetor de listas a serem processadas:

```
lista[1], lista[2], ... lista[k]  
nome[1], nome[2], ... nome[k]
```



Intercalação em K-vias

Considerando
que os
nomes
ocorrem em
apenas uma
lista

FIGURE 7.17 K-way merge loop, assuming no duplicate names.

```
/* initialize the process by reading in a name from each list */
for i := 1 to K
    call input() to get name[i] from list[i]
next i

/* now start the K-way merge */
while (MORE_NAMES_EXIST)

    /* find subscript of name that has the lowest collating
       sequence value among the names available on the K lists */
    LOWEST := 1
    for i := 2 to K
        if (name[i] < name[LOWEST])
            LOWEST := i
    next i

    write name[LOWEST] to OUT_FILE

    /* now replace the name that was written out */
    call input() to get name[LOWEST] from list[LOWEST]
endwhile
```



Intercalação em K-vias

FIGURE 7.17 K-way merge loop, assuming no duplicate names.

```
/* initialize the process by reading in a name from each list */
for i := 1 to K
    call input() to get name[i] from list[i]
next i

/* now start the K-way merge */
while (MORE_NAMES_EXIST)

    /* find subscript of name that has the lowest collating
       sequence value among the names available on the K lists */
    LOWEST := 1
    for i := 2 to K
        if (name[i] < name[LOWEST])
            LOWEST := i
    next i

    write name[LOWEST] to OUT_FILE
```

```
// considerando que os nomes podem ocorrer em mais de uma lista
for i := 1 to k
    if (name[i] = name[LOWEST])
        call input() to get name[i] from list[i]
```

*/



Ordenação de um Arquivo em Memória Principal

Foram discutidas em aulas passadas duas abordagens para ordenar arquivos em memória principal:

- Carregar um arquivo em memória principal, ordená-lo e gravá-lo em disco;
- Carregar somente as chaves dos registros, ordená-las e gravar o arquivo em disco conforme as chaves ordenadas (keysorting).
- Vamos voltar a esse tema e supor que um arquivo pode ser totalmente armazenado em memória principal para a sua ordenação



Ordenação de um Arquivo em Memória Principal

- Os principais passos são:
 - Ler arquivo para memória;
 - Ordená-lo usando um bom algoritmo de ordenação em memória;
 - Escrever o arquivo de volta no disco.
- O tempo total de ordenação é igual a soma dos tempos de execução de cada um dos passos
- O desempenho é bom, pois usa E/S sequencial, mas pode ser melhorado realizando-se alguns passos em paralelo!



Sobreposição de Processamento e E/S: Heapsort

- Para ordenação do arquivo em memória: necessário ter todo o arquivo em memória antes de começar a trabalhar. Pode-se mudar isso?
- O Heapsort pode começar a fazer a ordenação a medida em que os valores a serem ordenados são lidos



Sobreposição de Processamento e E/S: Heapsort

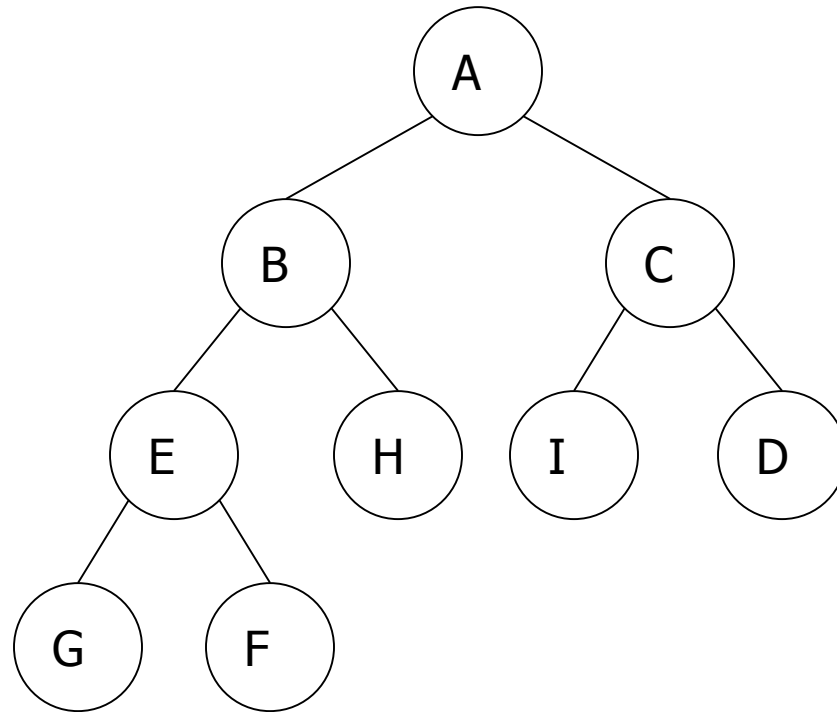
- Abordagem:
 - Realizar a construção da heap enquanto os dados são lidos para a memória principal;
 - Remover os dados da heap em ordem crescente/decrescente de chave e atualizar a heap enquanto os dados são gravados em disco.



Sobreposição de Processamento e E/S: Heapsort

- Heapsort: mantém as chaves em uma heap, uma árvore binária com 3 propriedades:
 - Cada nó possui uma única chave, que é maior ou igual à chave do nó pai (heap-min);
 - É uma árvore binária completa, i.e., todas as folhas estão em, no máximo, dois níveis, e todas as folhas do nível mais baixo estão nas posições mais à esquerda;
 - A heap pode ser mantida em um vetor tal que os índices dos filhos esquerdo e direito de um nó estão nas posições $2i$ e $2i+1$, respectivamente; e o nó pai de um nó j é o menor inteiro igual a $j/2$.

Sobreposição de Processamento e E/S: Heapsort



A	B	C	E	H	I	D	G	F
1	2	3	4	5	6	7	8	9



Construção da Heap

- Heapsort (algoritmo em duas partes):
 - Construção da heap (a partir da leitura das chaves);
 - Escrita das chaves ordenadamente (ordenação).
- A primeira parte pode ocorrer (virtualmente) simultaneamente à leitura das chaves, o que torna o seu custo computacional igual a zero: sai de graça em termos de tempo de execução



Construção da Heap

- Para inserir na heap são necessários os seguintes passos:
 1. Colocar o valor a ser inserido na última posição da heap;
 2. Compara o valor inserido com valor da chave no nó pai. Os valores trocam de posição se o valor inserido for menor que o valor no nó pai;
 3. Repete o passo 2 até que o valor inserido seja maior ou igual que o valor no nó pai ou se atingir o nó raiz.

Constru

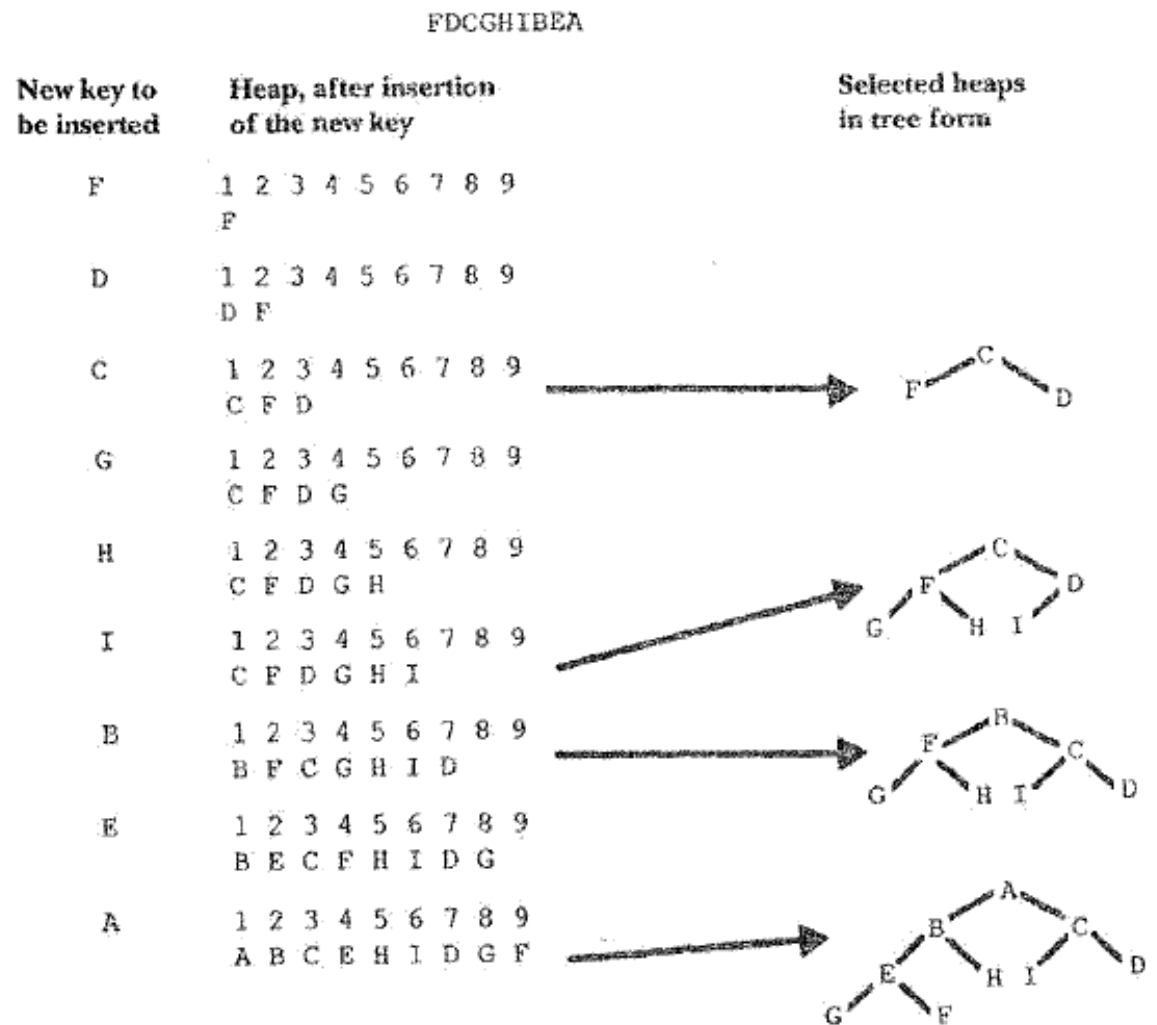


FIGURE 7.21 Sample application of the heap-building algorithm. The keys F, D, C, G, H, I, B, E, and A are added to the heap in the order shown.

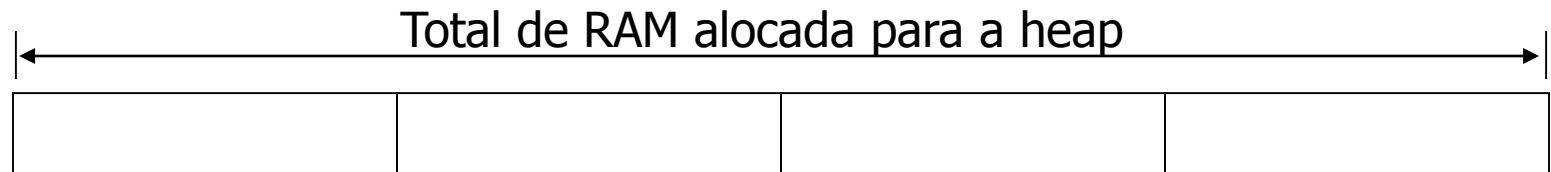


Construção da Heap

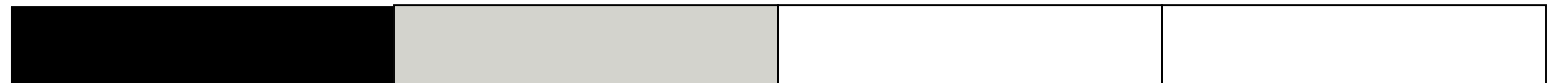
- Na operação de leitura:
 - Lê um bloco de registros para um buffer de entrada e,
 - opera sobre todos os registros desse bloco antes de passar para o próximo bloco.
- Esse buffer pode ser parte da memória utilizada para a heap:
 - Cada vez que um novo bloco é lido, ele é anexado ao final da heap;
 - Cada vez que um registro é absorvido na heap (a heap cresce), o próximo registro está no final da heap.



Construção da Heap



↙ Primeiro buffer de entrada é lido e inicia a primeira parte da heap



↳ Enquanto isso o segundo buffer de entrada é lido





Construção da Heap

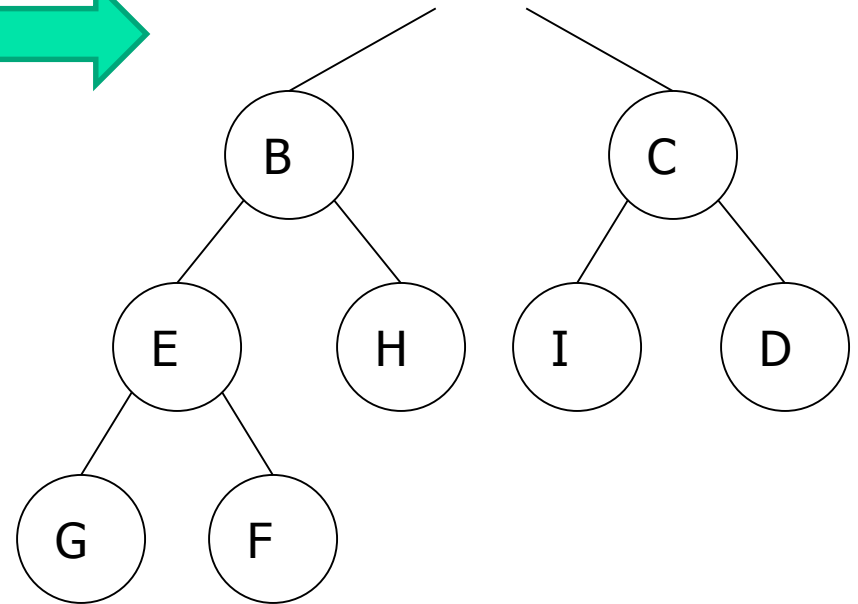
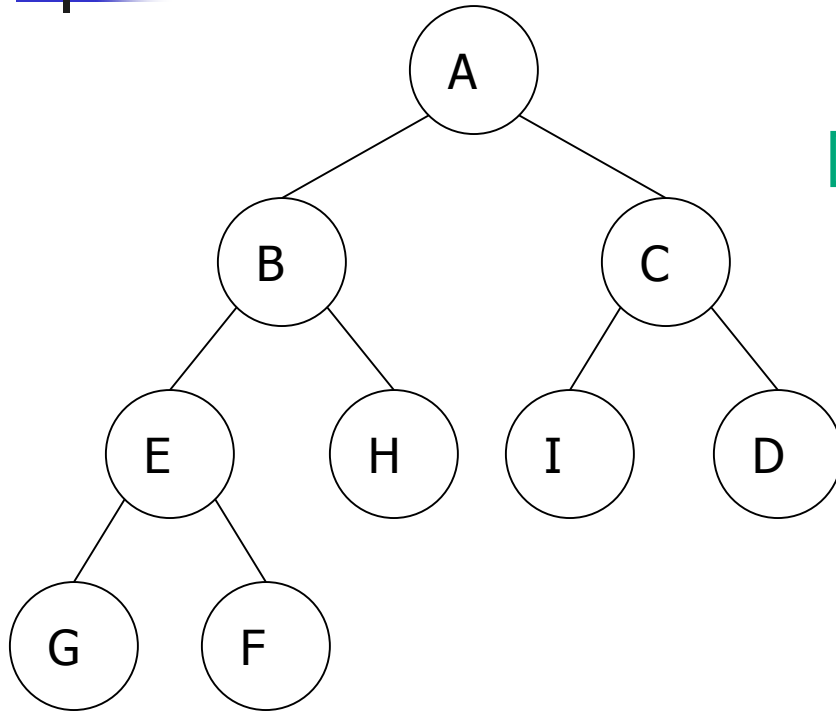
- Quantos buffers utilizar? Onde colocá-los?
 - O número de buffers é igual ao número de blocos do arquivo, e os buffers são colocados em sequência no próprio vetor
- A segunda parte consiste em escrever a heap ordenadamente



Ordenação Simultânea com Escrita

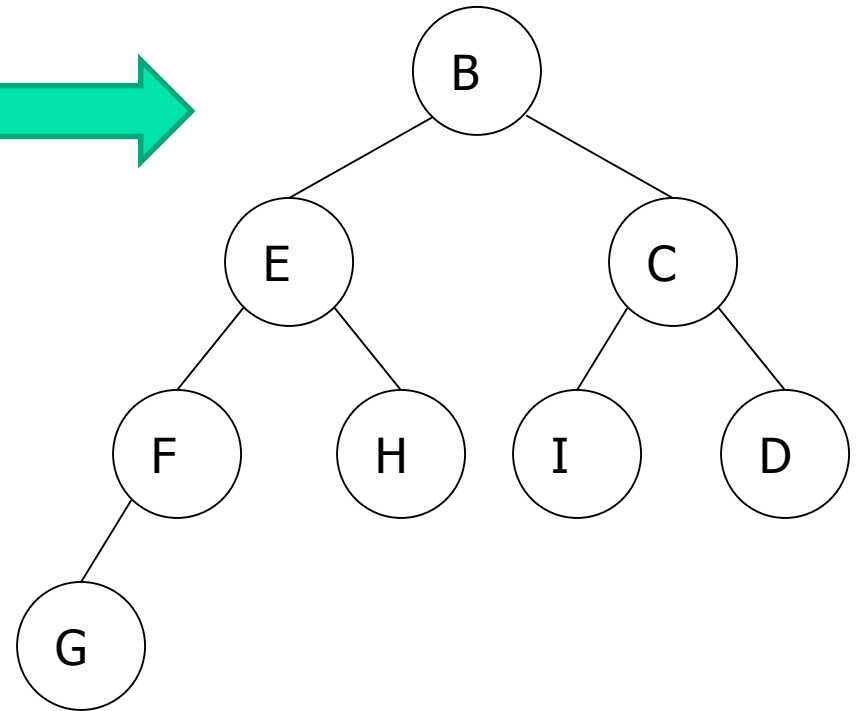
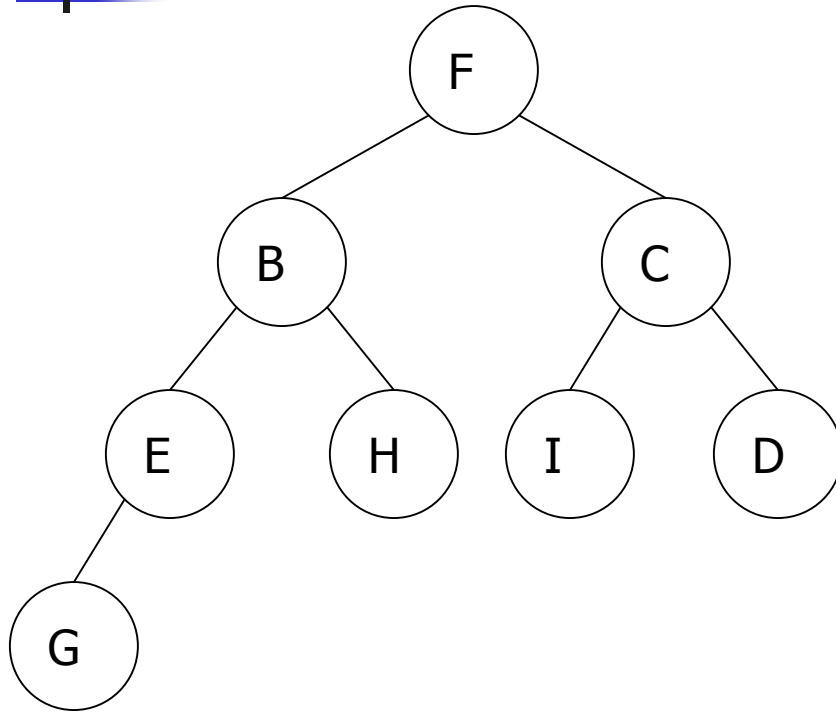
- Para remover da heap são necessários os seguintes passos:
 1. Remover o valor que está na raiz da árvore (a menor chave!);
 2. Mover o elemento na última posição da heap (K) para a raiz. Ajustar a heap com tamanho menor em um elemento;
 3. Enquanto K for maior que o menor dos seus filhos (NovoK), troque K por NovoK (trocar pelo menor dos seus filhos).

Ordenação Simultânea com Escrita



A	B	C	E	H	I	D	G	F
1	2	3	4	5	6	7	8	9

Ordenação Simultânea com Escrita



F	B	C	E	H	I	D	G	
1	2	3	4	5	6	7	8	

Ordenação Simultânea com Escrita



- Não é imediata a possibilidade de realizar processamento e escrita simultaneamente. Mas sabemos de imediato qual será o primeiro registro do arquivo ordenado; e, depois de algum processamento, sabemos quem será o segundo, o terceiro, etc.
- Então, logo que um bloco de registros se encontra ordenado, ele pode ser escrito para o arquivo de saída, enquanto o próximo bloco é processado, e assim por diante



Ordenação Simultânea com Escrita

- Além disso, cada bloco liberado para escrita pode ser considerado um novo buffer de escrita. É necessário uma coordenação cuidadosa entre processamento e escrita, mas, teoricamente pelo menos, as operações podem ser quase que completamente simultâneas
- Toda operação de E/S é essencialmente sequencial (todos os registros são lidos segundo a ordem de ocorrência no arquivo, e escritos ordenadamente)
- Além disso, sabemos que o número de operações de seek realizadas é o menor possível, pois toda E/S é sequencial



Ordenação de Arquivos em Disco

- Os métodos de ordenação vistos até o momento:
 - Memory Sort: realiza a ordenação carregando todos os registros em memória principal;
 - Keysorting: realiza somente a ordenação das chaves em memória principal.
- Ambos os métodos não podem ser utilizados se o arquivo for muito grande

Ordenação de Arquivos em Disco



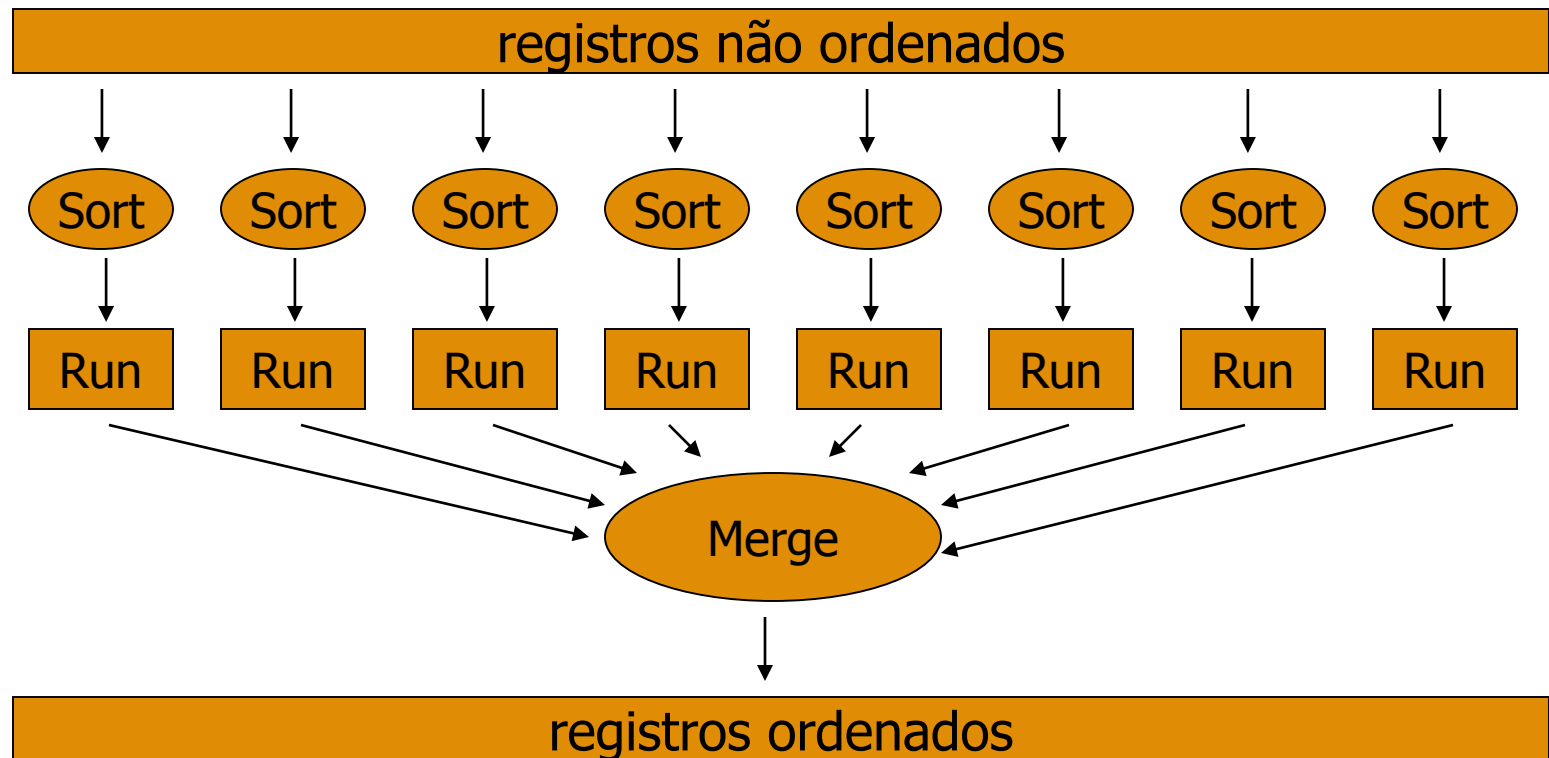
- Por exemplo, suponha um arquivo com 8.000.000 de registros, cada registro com 100 bytes e com um campo chave de 10 bytes. Seriam 800 MB de dados e 80MB de chaves.
- Se supormos um computador com 10MB de memória RAM livre (descontando S.O., programas, buffers de E/S, etc.), o arquivo é muito grande para qualquer um dos métodos anteriores.

Ordenação de Arquivos em Disco



- Uma outra abordagem seria utilizar a idéia do heapsort, mas organizando o arquivo em corridas (runs).
- Uma corrida é um “subarquivo” ordenado.
- No exemplo anterior as corridas podem ter tamanho 10MB/100bytes por registro = 100.000 registros.
- As corridas ordenadas podem ser compostas em um único arquivo utilizando uma intercalação (merge).

Merge Sort





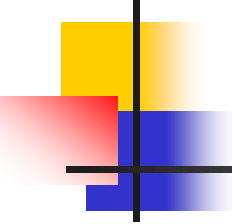
Merge Sort

- Método envolve 2 fases: geração das corridas e intercalação
- Fase 1: Geração das Corridas
 - segmentos do arquivo (corridas) são ordenados em memória RAM, usando algum método eficiente de ordenação interna (p.ex., Heapsort), e gravados em disco
 - corridas vão sendo gravadas a medida em que são geradas



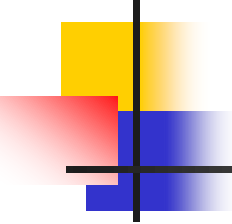
Merge Sort

- Fase 2: Intercalação
 - As corridas geradas na fase anterior são intercaladas, formando o arquivo ordenado
- Esta solução:
 - Pode ordenar arquivos realmente grandes



Qual o custo (tempo) do Merge Sort ?

- Supondo:
 - Arquivo com 80 MB (800.000 registros; cada registro com 100 bytes), e cada corrida com 1 MB (10.000 registros)
 - Características do disco:
 - tempo médio para seek: 18 ms
 - atraso rotacional: 8.3 ms
 - taxa de transferência: 1229 bytes/ms



Qual o custo (tempo) do Merge Sort ?

- Existem quatro pontos a serem considerados:
 - leitura dos registros para a memória para a criação de corridas
 - escrita das corridas ordenadas para o disco
 - leitura das corridas para intercalação
 - escrita do arquivo final em disco



Leitura dos registros e criação das corridas

- Lê-se 1MB de cada vez, para produzir corridas de 1 MB
- Serão 80 leituras, para formar as 80 corridas iniciais
- O tempo de leitura de cada corrida inclui o tempo de acesso a cada bloco (seek + rotational delay) somado ao tempo que leva para transferir cada bloco



Leitura dos registros e criação das corridas

seek = 18ms, rot. delay = 8.3ms, total 26.3ms

Tempo total para a fase de ordenação:

$80 * (\text{tempo de acesso a uma corrida}) + \text{tempo de transferência de 80MB}$

Acesso: $80 * (\text{seek} + \text{rot. delay} = 26.3\text{ms}) = 2\text{s}$

Transferência: 80 MB a 1229 bytes/ms = 65s

Total: 67s



Escrita das corridas ordenadas no disco

- Idem à leitura!

Serão necessários outros 67s



Leitura das corridas do disco para a memória (para intercalação)

- 1MB de MEMÓRIA para armazenar 80 buffers de entrada
 - portanto, cada buffer armazena $1/80$ de uma corrida (12.500 bytes). Logo, cada corrida deve ser acessada 80 vezes para ser lida por completo
- 80 acessos para cada corrida X 80 corridas
 - 6.400 seeks
- Considerando acesso = seek + rot. delay
 - $26.3\text{ms} \times 6.400 = 168\text{s}$
- Tempo para transferir 80 MB = 65s

Escrita do arquivo final em disco



- Precisamos saber o tamanho dos buffers de saída. Nos passos 1 e 2, a MEMÓRIA funcionou como buffer, mas agora a MEMÓRIA está armazenando os dados a serem intercalados
- Para simplificar, assumimos que é possível alocar 2 buffers adicionais de 20.000 bytes para escrita
 - dois para permitir double buffering, 20.000 porque é o tamanho da trilha no nosso disco hipotético

Escrita do arquivo final em disco



- Com buffers de 20.000 bytes, precisaremos de $80.000.000 \text{ bytes} / 20.000 \text{ bytes} = 4.000$ seeks
- Como tempo de seek+rot.delay = 26.3ms por seek, 4.000 seeks usam 4.000×26.3 , e o total de 105s
- Tempo de transferência é 65s



Tempo total

- Leitura dos registros para a memória para a criação de corridas: 67s
- Escrita das corridas ordenadas para o disco: 67s
- Leitura das corridas para intercalação: $168 + 65 = 233$ s
- Escrita do arquivo final em disco: $105 + 65 = 170$ s
- Tempo total do Merge Sort = 537s (8m57s)



Ordenação de um arquivo com 8.000.000 de registros

- Análise - arquivo de 800 MB
- O arquivo aumenta, mas a memória não!
 - Em vez de 80 corridas iniciais, teremos 800
 - Portanto, seria necessário uma intercalação em 800-vias no mesmo 1 MB de memória, o que implica em que a memória seja dividida em 800 buffers na fase de intercalação



Ordenação de um arquivo com 8.000.000 de registros

- Cada buffer comporta $1/800$ de uma corrida, e cada corrida é acessada 800 vezes
- $800 \text{ corridas} \times 800 \text{ seeks/corrída} = 640.000 \text{ seeks no total}$
- O tempo total agora é superior a 5 horas e 19 minutos, aproximadamente 36 vezes maior do que o arquivo de 80 MB (que é apenas 10 vezes menor)



Ordenação de um arquivo com 8.000.000 de registros

- Como melhorar esse tempo de execução?
 - Ordenação: A criação das corridas (leitura+escrita) é feita com E/S sequencial (heapsort);
 - **Intercalação: leitura das corridas: acesso aleatório. A intercalação dos registros envolve ler diversos arquivos (corridas) diferentes, em um comportamento (quase) aleatório;**
 - Intercalação: escrita do arquivo final: também sequencial.

O custo de aumentar o tamanho do arquivo

- A grande diferença de tempo na intercalação dos dois arquivos (de 80 e 800 MB) é devida à diferença nos tempos de acesso às corridas (seek e rotational delay) realizada no merge.
- Em geral, para uma intercalação em K-vias de K corridas, em que cada corrida é do tamanho da memória disponível, o tamanho dos buffers para cada uma das corridas é de:

$$\begin{aligned} (1/K) \times \text{tamanho da memória disponível} = \\ (1/K) \times \text{tamanho de cada corrida} \end{aligned}$$

$$1/80 = 0,0125 * 1.000.000 = 12.500 \text{ bytes}$$



Complexidade do Merge Sort

- Como temos K corridas, a operação de intercalação requer K^2 seeks.
- Medido em termos de seeks, o Merge Sort é $O(K^2)$
- Como K é diretamente proporcional à N (número de registros), o Merge Sort é $O(N^2)$ em termos de seeks
- Assim, conforme o arquivo cresce, o tempo requerido para realizar a ordenação cresce rapidamente



Maneiras de reduzir esse tempo

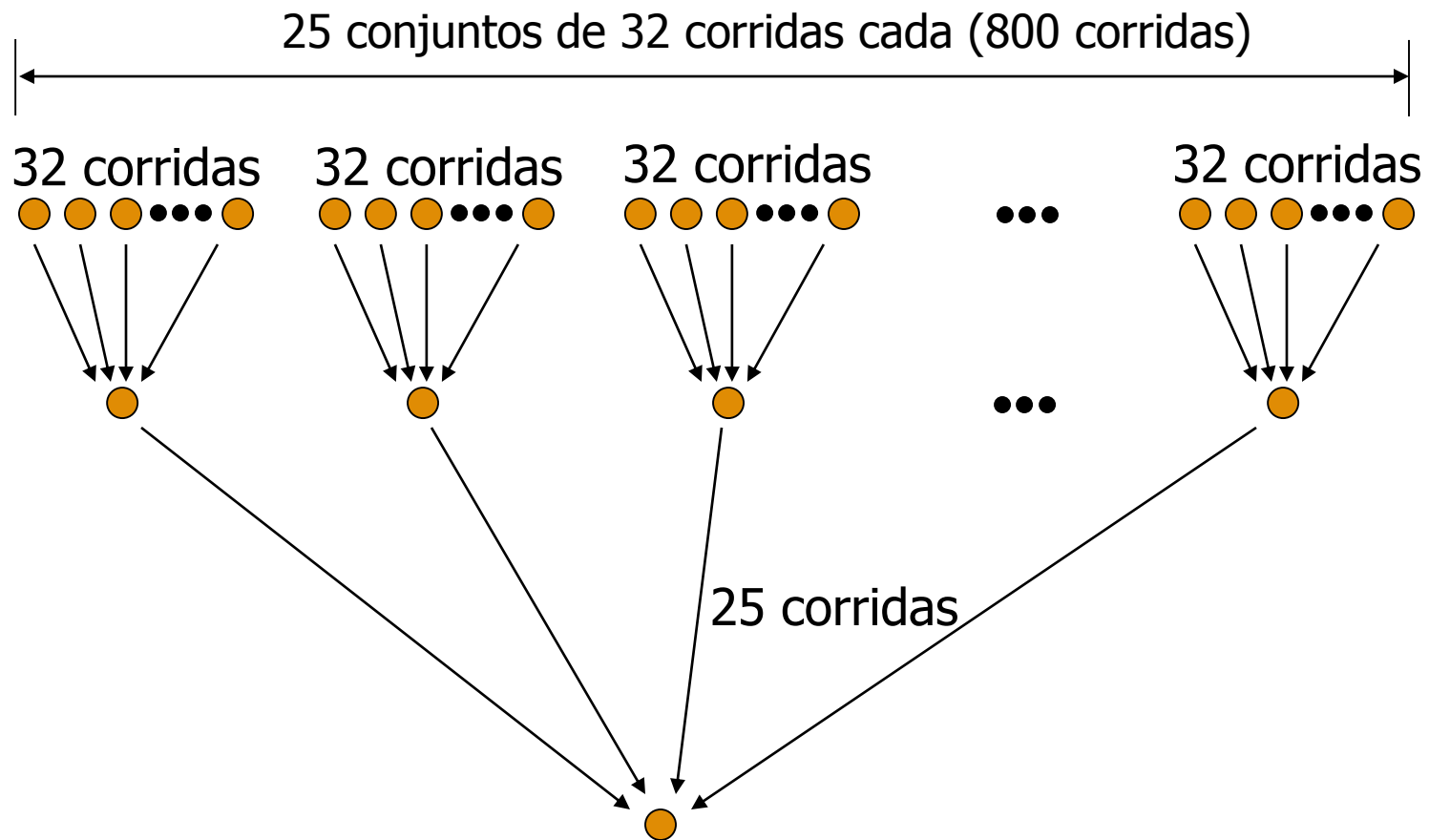
- Usar mais hardware (disk drives, MEMÓRIA, canais de I/O)
- Realizar a intercalação em mais de um passo, o que reduz a ordem de cada intercalação e aumenta o tamanho do buffer para cada corrida
- Aumentar o tamanho das corridas iniciais ordenadas
- Achar meios de realizar I/O simultâneo



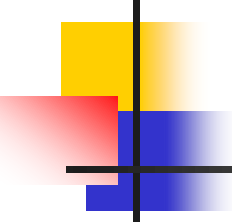
Diminuição do Número de Seeks com Intercalação em Múltiplos Passos (Multistep Merging)

- Ao invés de intercalar todas as corridas ao mesmo tempo, o grupo original é dividido em grupos menores
- A intercalação é feita para cada sub-grupo
- Para cada um desses sub-grupos, um espaço maior é alocado para cada corrida, portanto um número menor de seeks é necessário
- Uma vez completadas todas as intercalações pequenas, o segundo passo completa a intercalação de todas as corridas

Intercalação em Múltiplos Passos



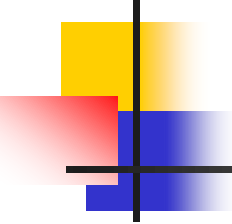
Intercalação em Múltiplos Passos



- É claro que um número menor de seeks será feito no primeiro passo. E no segundo?
- O segundo passo exige não apenas seeking, mas também transferências nas leituras/escritas. Será que as vantagens superam os custos?
- No exemplo do arquivo com 800 MB tínhamos 800 corridas com 10.000 registros cada. Para esse arquivo, a intercalação múltipla poderia ser realizada em dois passos:
 - primeiro, a intercalação de 25 conjuntos de 32 corridas cada
 - depois, uma intercalação em 25-vias

Intercalação em Múltiplos Passos

- Nesse caso, cada registro é lido duas vezes, mas o uso de buffers maiores diminui o seeking
- Passo único visto anteriormente exige 640.000 seeks. Para a intercalação em 2 passos, temos:
 - Primeiro passo
 - Cada intercalação em 32-vias aloca buffers que podem conter $1/32$ de uma corrida, então serão realizados $32 \times 32 = 1024$ seeks
 - Então, 25 vezes a intercalação em 32-vias exige $25 \times 1024 = 25.600$ seeks
 - Cada corrida resultante tem $32 \times 10.000 = 320.000$ registros = 32 MB



Intercalação em Múltiplos Passos

- Segundo passo
 - Cada uma das 25 corridas pode alocar $1/25$ do buffer
 - Portanto, cada buffer pode alocar $1\text{MB}/25 = 40\text{KB}$ (400 registros). Cada corrida tem 32MB, então o buffer pode armazenar $32\text{MB}/40\text{KB} = 1/800$ da corrida. Então, esse passo exige 800 seeks por corrida, num total de $25 \times 800 = 20.000$ seeks
- Total de seeks nos dois passos: $25.600 + 20.000 = 45.600$

E o tempo total de intercalação?

- Nesse caso, cada registro deve ser transmitido 4 vezes, em vez de duas, portanto gastamos mais 651s em cada tempo de transmissão
- Ainda, cada registro é escrito duas vezes: mais 40.000 seeks (assumindo 2 buffers com 20.000 posições cada)
- Somando tudo isso, o tempo total de intercalação = **5.907s* ~ 1hora 38 min.**
 - A intercalação em 800 vias consumia ~5 horas...



E o tempo total de intercalação?

- Aumentando o tamanho do buffer disponível para cada corrida, a um custo de passos extras, diminuiu-se dramaticamente o número de acessos aleatórios
- Seria possível melhorar o tempo total se utilizarmos 3 passos? Talvez, mas os ganhos provavelmente serão menores, pois no caso de 2 passos, o tempo total de seek + rotational delay já foi aproximadamente reduzido ao mesmo que o tempo de transmissão
- E se as corridas fossem distribuídas diferentemente? Por exemplo, como seria se fossem realizadas primeiro 400 intercalações em 2-vias, seguida de uma intercalação em 400-vias? [Discussão detalhada em Knuth 1973, The Art of Computer Programming, vol. 3, Searching and Sorting]



Aumento do tamanho das corridas utilizando Replacement Selection

- O que aconteceria se fosse possível, de algum modo, aumentar o tamanho das corridas iniciais?
- Se, no caso da intercalação em passo único, pudéssemos alocar corridas com 20.000 registros, em vez de 10.000 (limite imposto pelo tamanho da MEMÓRIA), teríamos uma intercalação em 400-vias, em vez de 800-vias, reduzindo o número de seeks pela metade (320.000 seeks)
- Em geral, corridas iniciais maiores implicam em um número menor de corridas, o que implica por sua vez em uma intercalação de ordem menor, com buffers maiores, resultando em um número menor de seeks



Aumento do tamanho das corridas utilizando Replacement Selection

- Idéia básica: selecionar na memória a menor chave, escrever essa chave no arquivo de saída, e usar seu lugar (replace it) para uma nova chave (da lista de entrada)



Aumento do tamanho das corridas utilizando Replacement Selection

- 1. Leia um conjunto de registros e ordene-os utilizando heapsort, criando uma heap; a essa heap dê o nome de primary heap.
- 2. Ao invés de escrever, neste momento, a primary heap inteira ordenadamente e gerar uma corrida (como seria feito no heapsort normal), escreva apenas o registro com menor chave.
- 3. Busque um novo registro no arquivo de entrada e compare sua chave com a chave que acabou de ser escrita.
 - Se a chave lida for maior que a chave já escrita, insira o registro normalmente na heap.
 - Se a chave lida for menor que a chave já escrita, insira o registro numa secondary heap, a qual contém registros menores dos que os que já foram escritos (note, usando a mesma memória!)
- 4. Repita o passo 3 enquanto existirem registros a serem lidos. Ficando a primary heap vazia, transforme a secondary heap em primary heap, e repita passos 2 e 3.

Aumento do tamanho das corridas utilizando Replacement Selection

Input:

21, 67, 12, 5, 47, 16

↑
Front of input string

Remaining input

Memory ($P = 3$)

Output run

21, 67, 12

5 47 16

–

21, 67

12 47 16

5

21

67 47 16

12, 5

–

67 47 21

16, 12, 5

–

67 47 –

21, 16, 12, 5

–

67 – –

47, 21, 16, 12, 5

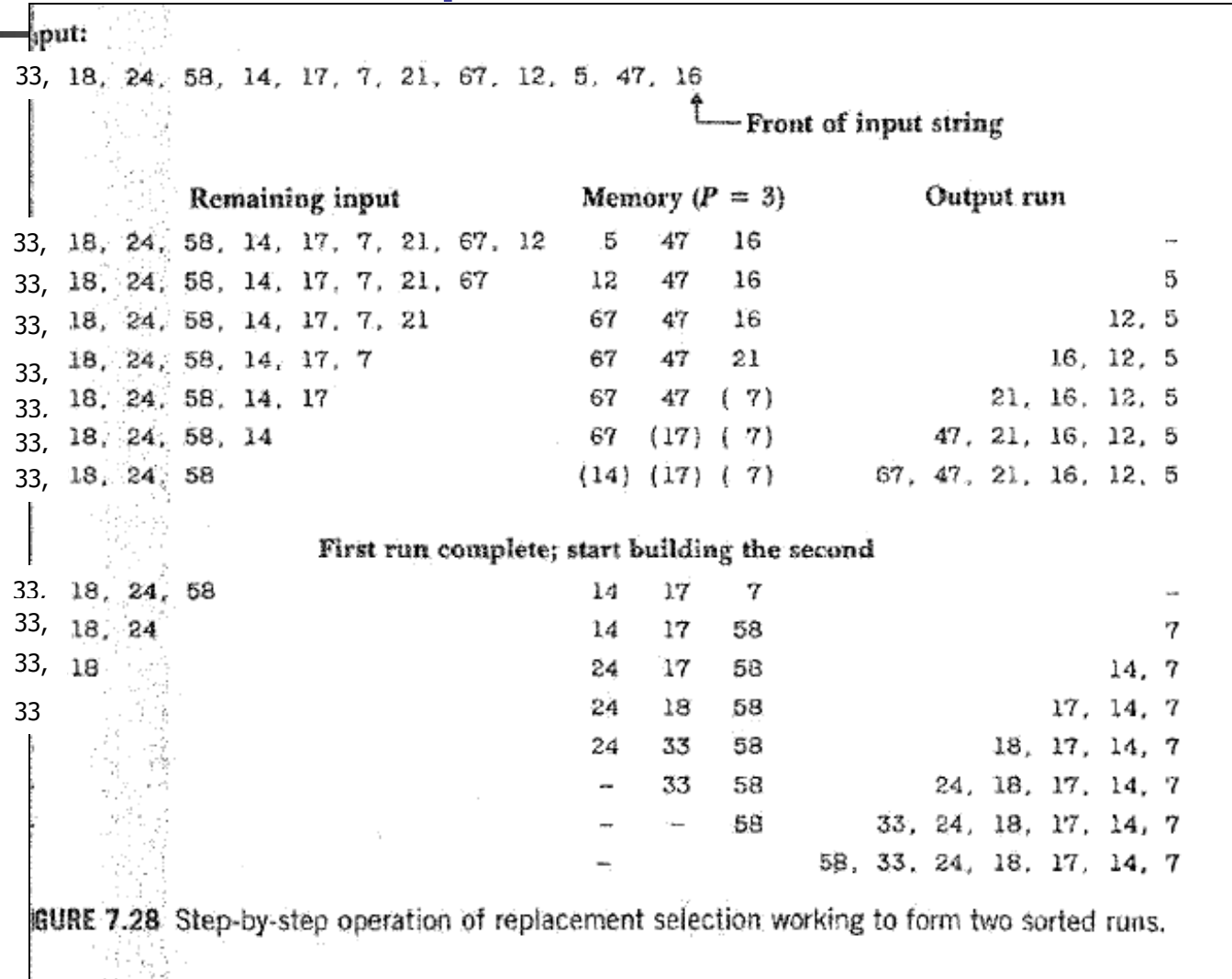
–

– – –

67, 47, 21, 16, 12, 5

FIGURE 7.27 Example of the principle underlying replacement selection.

Aumento do tamanho das corridas utilizando Replacement Selection



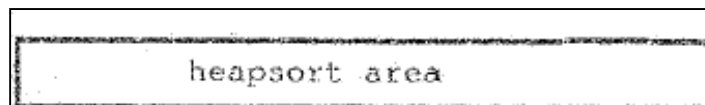


Aumento do tamanho das corridas utilizando Replacement Selection

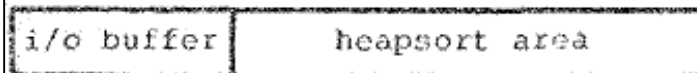
- Dadas P posições de memória, qual o tamanho, em média, da corrida que o algoritmo de replacement selection vai produzir?
 - A corrida terá, em média, tamanho $2P$.

Aumento do tamanho das corridas utilizando Replacement Selection

- Quais são os custos de se usar replacement selection?
 - Precisaremos de buffers p/ I/O e, portanto, não poderemos utilizar toda a MEMÓRIA disponível para ordenação.



(a) In-RAM sort: all available space used for the sort.



(b) Replacement selection: some of available space is used for i/o.

FIGURE 7.29 In-RAM sort versus replacement selection, in terms of their use of available RAM for sorting operation.



Aumento do tamanho das corridas utilizando Replacement Selection

- Ordenação de um arquivo com 8 milhões de registros usando uma MEMÓRIA onde cabem 10.000 registros.
- No Merge Sort, onde registros são lidos na memória até que ela esteja cheia, podemos realizar leituras sequências de 10.000 registros por vez, até que 800 corridas sejam criadas. Portanto, o primeiro passo do processo de ordenação requer 800 seeks para leitura, 800 para escrita, 1.600 no total.



Aumento do tamanho das corridas utilizando Replacement Selection

- Para utilizar replacement selection podemos, por exemplo, reservar um buffer de I/O com capacidade de 2.500 registros, deixando 7.500 vagas para o processo de replacement selection.
- Se o buffer suporta 2.500 registros, podemos executar leituras sequências de 2.500 registros de cada vez, e portanto precisamos de $8.000.000 / 2.500 = 3.200$ seeks para acessar todos os registros do arquivo. Portanto, o primeiro passo do processo de ordenação requer 3.200 seeks para leitura, 3.200 para escrita, 6.400 total.



Aumento do tamanho das corridas utilizando Replacement Selection

- Se os registros ocorrem em sequência aleatória de chaves, o tamanho médio da corrida será $2 \times 7.500 = 15.000$ registros, e teremos $\sim 8.000.000/15.000 = 534$ dessas corridas sendo produzidas.
- Para o **passo de intercalação**, dividimos a memória total (1 MB) em 534 buffers que podem conter cerca de $1\text{MB}/534 = 18.73$ registros.
- Portanto, realizaremos cerca de $15.000/18.73 = 801$ seeks por corrida e, no total: $801 \text{ seeks por corrida} \times 534 \text{ corridas} = 427.734 \text{ seeks}$.



Aumento do tamanho das corridas utilizando Replacement Selection

- Resumindo:

- 800 ordenações em memória: 800 corridas, 681.600 seeks, 4h 58min (seek + rotational delay)
- Replacement selection: 534 corridas: 434.134 seeks, 3h 48 min (seek + rotational delay)
- Replacement selection: 200 corridas: 206.400 seeks, 1h 30 min (seek + rotational delay)
 - O último caso ilustra ganhos quando os registros estão parcialmente ordenados.



Pontos Básicos para Ordenação Externa

- Uma lista de ferramentas conceituais para melhorar ordenação externa inclui:
 - Para ordenação interna, em-MEMÓRIA, use heapsort para formar corridas. Com heapsort e double buffering é possível sobrepor processamento com input e output.
 - Use o máximo de MEMÓRIA possível. Isso permite corridas maiores e buffers maiores na fase de intercalação.



Pontos Básicos para Ordenação Externa

- Uma lista de ferramentas conceituais para melhorar ordenação externa inclui:
 - Se o número de corridas é muito grande, de modo que o seek e rotation times são maiores que o tempo de transmissão, use intercalação em múltiplos passos: aumenta o tempo de transmissão, mas diminui em muito o número de seeks.
 - Considere utilizar Replacement Selection para formação da corrida inicial, especialmente se existe a possibilidade das corridas estarem parcialmente ordenadas.



Ordenação de Arquivos em Fita

- De modo geral, a ordenação em fita requer:
 - a distribuição do arquivo original em corridas ordenadas. Em geral, as corridas são distribuídas em diferentes drives de fita;
 - a intercalação das corridas em um único arquivo ordenado.



Ordenação de Arquivos em Fita

- Criar corridas é fácil
- Em geral, é recomendado a utilização do replacement selection (usando fita, não existe mais o problema do seeking!!!)
- O difícil e complexo é fazer a intercalação



Intercalação Balanceada

- Suponha que o arquivo original foi dividido em 10 corridas, e que o sistema possui 4 tape drives
- Como o arquivo original ocupa um dos tape drives, temos a escolha de distribuir as corridas em 2 ou 3 dos outros drives



Intercalação Balanceada

- **Método da intercalação balanceada em 2-vias**
 - distribuição inicial das corridas em dois drives
 - a cada passo da intercalação, a distribuição das corridas é feita em dois drives, e a saída também é distribuída em dois drives (com exceção da última intercalação)
 - é o algoritmo mais simples de intercalação em fita... e também o mais lento

Intercala

	Tape	Contains runs				
Step 1	T1	R1	R3	R5	R7	R9
	T2	R2	R4	R6	R8	R10
	T3	—				
	T4	—				
Step 2	T1	—				
	T2	—				
	T3	R1-R2	R5-R6	R9-R10		
	T4	R3-R4	R7-R8			
Step 3	T1	R1-R4	R9-R10			
	T2	R5-R8				
	T3	—				
	T4	—				
Step 4	T1	—				
	T2	—				
	T3	R1-R8				
	T4	R9-R10				
Step 5	T1	R1-R10				
	T2	—				
	T3	—				
	T4	—				

FIGURE 7.31 Balanced four-tape merge of 10 runs.

Intercalação Balanceada

	T1	T2	T3	T4	
Step 1	1 1 1 1 1	1 1 1 1 1	—	—	
Step 2	—	—	2 2 2	2 2	Merge ten runs
Step 3	4 2	4	—	—	Merge ten runs
Step 4	—	—	8	2	Merge ten runs
Step 5	10	—	—	—	

FIGURE 7.32 Balanced four-tape merge of 10 runs expressed in a more compact table notation.



Intercalação Balanceada

- Custo associado à intercalação balanceada:
 - Como não existe seeking, vamos medi-lo em termos do tempo gasto transferindo dados. No exemplo, cada dado foi "passado" (transferido) 4 vezes só na fase de intercalação, excluindo o tempo para gerar as corridas iniciais.



Intercalação Balanceada

- De modo geral, para um certo número de corridas iniciais, quantas "passadas" pelos dados serão necessárias? Isto é, começando com N corridas, quantos passos são necessários para reduzir o número de corridas para 1?

$$p = \lceil \log_2 N \rceil$$

No exemplo, $N = 10$ resultou em 4 passos



Intercalação Balanceada

- No exemplo do arquivo com 800 MB (parcialmente ordenado) utilizando replacement selection têm-se 200 corridas, logo $\lceil \log_2 200 \rceil = 8$ passos
- Se conseguirmos fazer leitura e escrita simultaneamente, sobrepondo por completo as operações, cada passo leva aproximadamente 11 minutos, num total de 1h28m
 - Assumindo o tape drive visto anteriormente: 6.250 bpi, taxa de transmissão de 1.250 KB/s. A esta taxa, um arquivo de 800 MB leva 640 segundos = 10.67 minutos para ser lido
- Tempo não competitivo com o merge em disco, uma vez que o tempo de transmissão supera a economia dos tempos gastos em seeks



Intercalação Balanceada

- **Método da intercalação balanceada em K-vias (K-way Balanced Merge)**
 - Para melhorar o desempenho precisamos reduzir o número de passos sobre os dados
 - Olhando a fórmula, podemos observar que é possível reduzir o número de passos aumentando a ordem de cada intercalação



Intercalação Balanceada

- Suponha que temos 20 tape-drives, 10 para input, 10 para output
- Nesse caso, a cada passo podemos combinar 10 corridas, e a cada passo o número de corridas é reduzido para um décimo do número no passo anterior



Intercalação Balanceada

- De modo geral, uma intercalação em K-vias, na qual a ordem de cada intercalação é K, tem como número de passos necessários para realizar a intercalação balanceada de N corridas:

$$p = \lceil \log_K N \rceil$$



Intercalação Balanceada

- No caso do exemplo, $p = \lceil \log_{10} 200 \rceil = 3$; tempo de 33 minutos, o qual é um tempo ótimo. O melhor para disco tinha sido 1h30m!
- **Custo:** 20 tape-drives trabalhando exclusivamente na intercalação.

Intercalações em Múltiplas Fases



- O método de intercalação balanceada é muito simples: tão simples que não aproveita oportunidades de economizar trabalho
- Grande economia pode ser obtida se omitirmos a intercalação quando uma das corridas está “sozinha”. Desse modo, ao fim de um dado passo, realiza-se o rewind em todas as fitas, menos a T3 (exemplo seguinte)

Intercalações em Múltiplas Fases

	<u>T1</u>	<u>T2</u>	<u>T3</u>	<u>T4</u>	
Step 1	1 1 1 1 1	1 1 1 1 1	—	—	
Step 2	—	—	2 2 2	2 2	Merge ten runs
Step 3	4	4	. . 2	—	Merge eight runs
Step 4	—	—	—	10	Merge ten runs

FIGURE 7.33 Modification of balanced four-tape merge that does not rewind between steps 2 and 3 to avoid copying runs.

Intercalações em Múltiplas Fases



- Deste modo, teríamos corridas R1-R4 da fita T1, corridas R5-R8 na fita T2, e as corridas R9-R10 na fita T3. Podemos, então, fazer uma intercalação em 3-vias das 3 fitas para a fita T4!

Intercalações em Múltiplas Fases



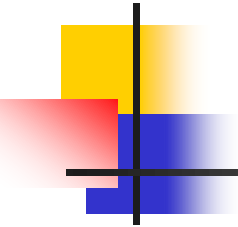
- Ao introduzirmos esse controle, o número de corridas a serem lidas/escritas é reduzido, uma vez que:
 - fizemos intercalação de ordem mais alta (3-way em vez de 2-way)
 - realizamos a intercalação das corridas de uma mesma fita (T3) em mais de uma fase (passos 2 e 4)

Intercalações em Múltiplas Fases



- Essas ideias formam a base para dois métodos famosos de intercalação: intercalação polifásica e intercalação em cascata
- De modo geral, nestas intercalações:
 - A distribuição inicial das corridas é tal que a intercalação inicial é uma intercalação J-1, em que J é o número de drives de fita disponíveis
 - A distribuição das corridas é tal que as fitas frequentemente contêm um número diferente de corridas

Intercalações em Múltiplas Fases

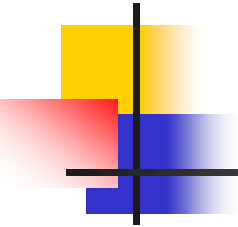


	<u>T1</u>	<u>T2</u>	<u>T3</u>	<u>T4</u>	
Step 1	1 1 1 1 1	1 1 1	1 1	—	
Step 2	. . 1 1 1	. . 1	—	3 3	Merge six runs
Step 3	. . . 1 1	—	5	. 3	Merge five runs
Step 4 1	4	5	—	Merge four runs
Step 5	—	—	—	10	Merge ten runs

FIGURE 7.34 Polyphase four-tape merge of 10 runs.

Intercalação polifásica: a intercalação é executada até que uma das fitas esvazie. Nesse ponto, uma das fitas de saída troca de papel com a fita de entrada

Intercalações em Múltiplas Fases



	T1	T2	T3	T4	
Step 1	1 1 1 1 1	1 1 1	1 1	—	
Step 2	. . 1 1 1	. . 1	—	3 3	Merge six runs
Step 3	. . . 1 1	—	5	—	Merge five runs
Step 4 1	4	5	—	
Step 5	—	—	—	—	

FIGURE 7.34 Polyphase four-tape merge of 10

Evita-se

- (a) Usar um grande número de fitas ($2f$ ou $f+1$ para a "IBMC" de f caminhos contra f)
- (b) Realizar várias leituras e escritas entre as fitas envolvidas (múltiplas fases)

Complexidade

Distribuição inicial das corridas entre as fitas
[ver Knuth 1973, The Art of Programming, vol. 3, Searching and Sorting]

Intercalação polifásica: a intercalação uma das fitas esvazie. Nesse ponto, u troca de papel com a fita de entrada



Pacotes para Mergesort

- Vários pacotes existem
- Normalmente são programas inteligentes que escolhem entre possíveis alternativas
- Porém, eles também permitem aos usuários controlar as estratégias e os dados
- É útil familiarizar-se com as técnicas, mesmo se for só para usar, uma vez que no campo de algoritmos e técnicas, várias lições são aprendidas!!!

<http://sortbenchmark.org/>

http://en.wikipedia.org/wiki/External_sorting

Métodos de Ordenação Externa



- O desenvolvimento nessa área é dependente do estado atual da tecnologia
 - A variedade de tipos de unidades tornam os métodos dependentes de vários parâmetros que afetam seus desempenhos
 - Por esta razão, apenas métodos gerais são, na maioria das vezes, apresentados



Referências

- Além do livro texto...
 - N. Ziviani. Projeto de Algoritmos com Implementações em Pascal e C. Segunda Edição. Thomson. 2004.