GBC053 - Gerenciamento de Bancos de Dados

Aula 10
Processamento co-sequencial

Humberto Razente
humberto.razente@ufu.br

Introdução

- Operações co-sequenciais envolvem o processamento coordenado de duas ou mais listas sequenciais para produzir uma única lista como saída
  - intersecção (matching)
    - quais os termos comuns em duas listas?
  - união (merging)
    - qual a lista ordenada de todos os termos de duas listas?

Intersecção de nomes em 2 listas

- Supondo que as listas estejam ordenadas e livres de nomes repetidos
  - lê-se um elemento de cada lista
    - ADAMS é inserido na resposta
  - lê-se o próximo elemento de cada lista
    - CARTER (lista 1) e lê-se sequencialmente a lista 2 até encontrar o elemento ou outro elemento ”maior” que CARTER

List 1	List 2
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

Modelo de implementação de um processo co-sequencial

- Passos:
  - Inicialização
    - é preciso organizar os dados de modo que o procedimento possa ser executado
  - Acesso ao próximo item da lista
    - é preciso dispor de um método simples que acesse os elementos da lista
  - Sincronização
    - deve-se ter certeza que o item corrente de uma lista não está à frente do item corrente da outra lista de modo que um casamento seja perdido

Modelo de implementação de um processo co-sequencial

- Passos (continuação):
  - Manipulação de final de arquivo
    - ao atingir o final de arquivo na lista 1 ou na lista 2, é preciso terminar o algoritmo
  - Reconhecimento de erros
    - ao detectar erros como chaves duplicadas ou elemento fora de ordem, uma ação deve ser tomada

- Objetivo é alcançar um algoritmo razoavelmente eficiente, simples e fácil de ser alterarado para acomodar tipos diferentes de dados

Modelo de implementação de um processo co-sequencial

- Sincronização
  - a cada passo da sincronização, considera-se que há dois elementos: item(1) e item(2)
    - item(1) pode ser menor, igual, ou maior que item(2)

1) se item(1) é menor que item(2), pega-se o próximo elemento da lista 1

2) se item(1) é maior que item(2), pega-se o próximo elemento da lista 2

3) se os termos são iguais, inclui-se na saída e pegam-se os próximos elementos das duas listas

Modelo de implementação de um processo co-sequencial

- União (merging) de duas listas
  - o item vai para saída nos três casos
    1) se item(1) é menor que item(2), pega-se o próximo elemento da lista 1
    2) se item(1) é maior que item(2), pega-se o próximo elemento da lista 2
    3) se os termos são os iguais, inclui-se na saída e pegam-se os próximos elementos das duas listas
  - além disso, lê-se as duas listas até o fim

Extensão do modelo para incluir intercalação multivias

- A aplicação mais comum de processamento co-sequencial que requer mais que 2 arquivos de entrada é a intercalação de k-vias (*k-way merge*)
  - concatenar k listas para criar uma lista ordenada como saída

Extensão do modelo para incluir intercalação multivias

- Laço para determinar quais listas contém o item
- Função minIndex
  - encontrar o índice do item com o menor valor
- Algoritmo

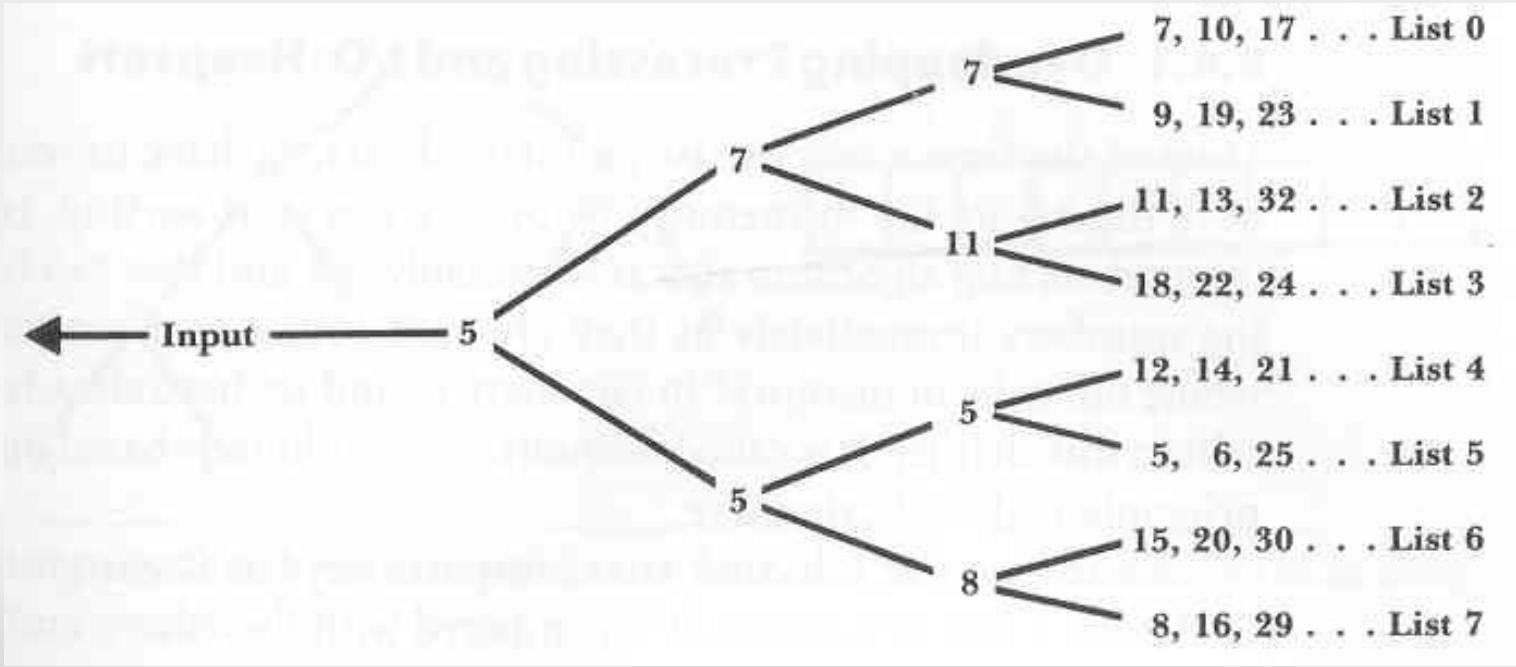
```
int minItem = MinIndex(Item,k);
ProcessItem(minItem);
for (i=0; i<k; i++)
    if (Item(minItem) == Item(i))
        MoreItems[i] = NextItemInList(i);
```

Extensão do modelo para incluir intercalação multivias

- Resumo:
  - Determina-se qual lista tem a chave de menor valor
  - Copia a chave para a saída
  - Passa-se para o próximo item da lista
  - Esse procedimento é executado até que todas as listas sejam varridas
- O algoritmo funciona bem para número de listas (k) pequeno, até k=8
  - a partir de k=8, o número de comparações em sequencia para achar a chave com menor valor torna-se cara → deve-se usar uma árvore de seleção (*selection tree*)

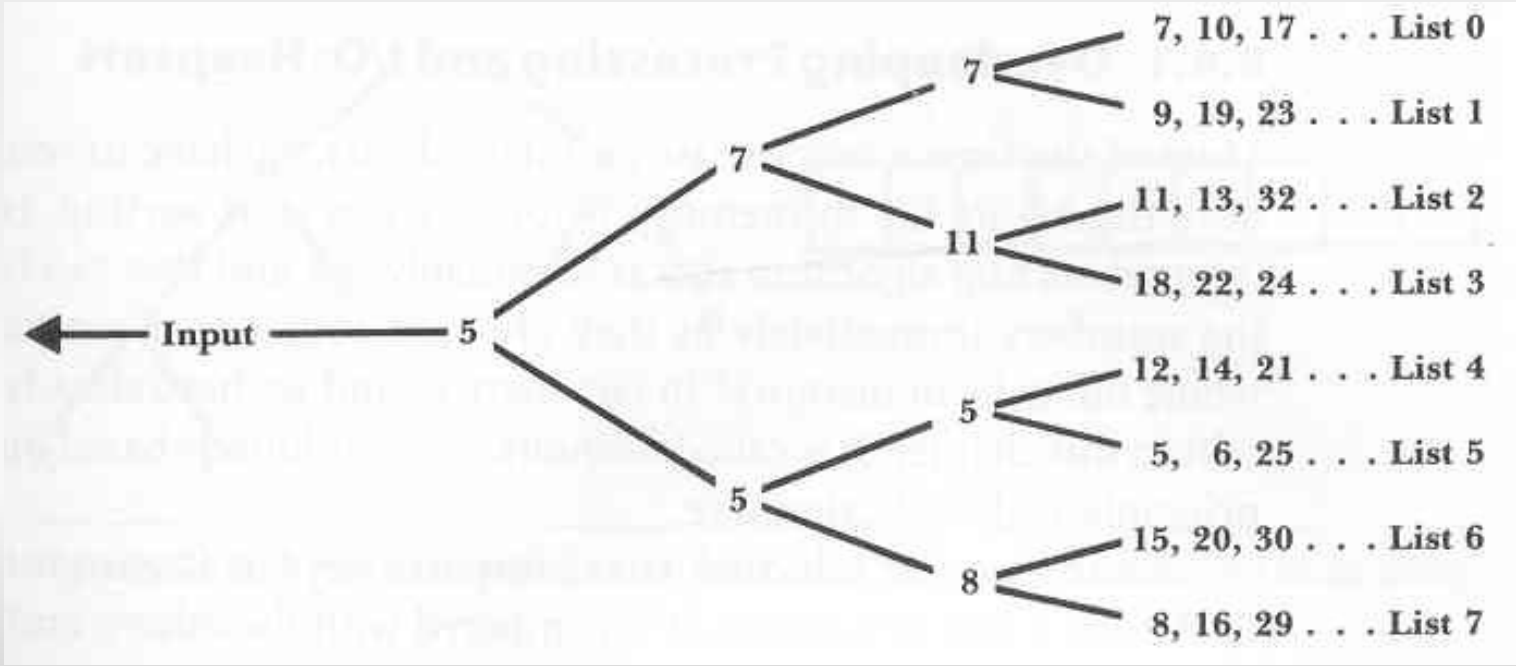
Extensão do modelo para incluir intercalação multivias

- Árvore de seleção
  - reduz-se o tempo requerido para encontrar uma chave com o menor valor por meio do uso de uma estrutura de dados que guarda informações sobre as chaves conforme os ciclos do laço do procedimento principal são executados



Extensão do modelo para incluir intercalação multivias

- Árvore de seleção
  - O valor mínimo sempre está na raiz da árvore
  - Cada chave tem uma referência para a lista de origem
  - A cada passo, pega-se a chave da raiz, lê-se o próximo elemento da lista associada, e re-organiza-se a estrutura



Ordenação de arquivos

Soluções vistas até aqui:

- Para a ordenação de um arquivo que cabe em memória
  - lêem-se os registros para memória
  - ordena-se em memória, seguida da re-escrita em disco
- Uso do keysort, permitindo ser ordenado um arquivo de registros maior que a memória
  - de modo que apenas as chaves são ordenadas, e um índice é escrito em disco

Solução ordenação em memória

- Quando se usa um algoritmo de ordenação interna, em geral precisa-se ler o arquivo todo para memória antes de iniciar o processamento
  - lê-se o arquivo todo para memória
  - ordena-se os registros com um algoritmo como o shellsort
  - escreve-se o arquivo no disco
- Solução é muito melhor que ordenar no próprio arquivo, gerando muitos seeks
  - Na ordenação interna, leituras e escritas são realizadas sequencialmente, e cada registro é lido uma vez e escrito uma vez

Ordenação em memória: intercalando E/S e processamento

- Se a solução é boa, há espaço para uma melhoria no tempo de execução?

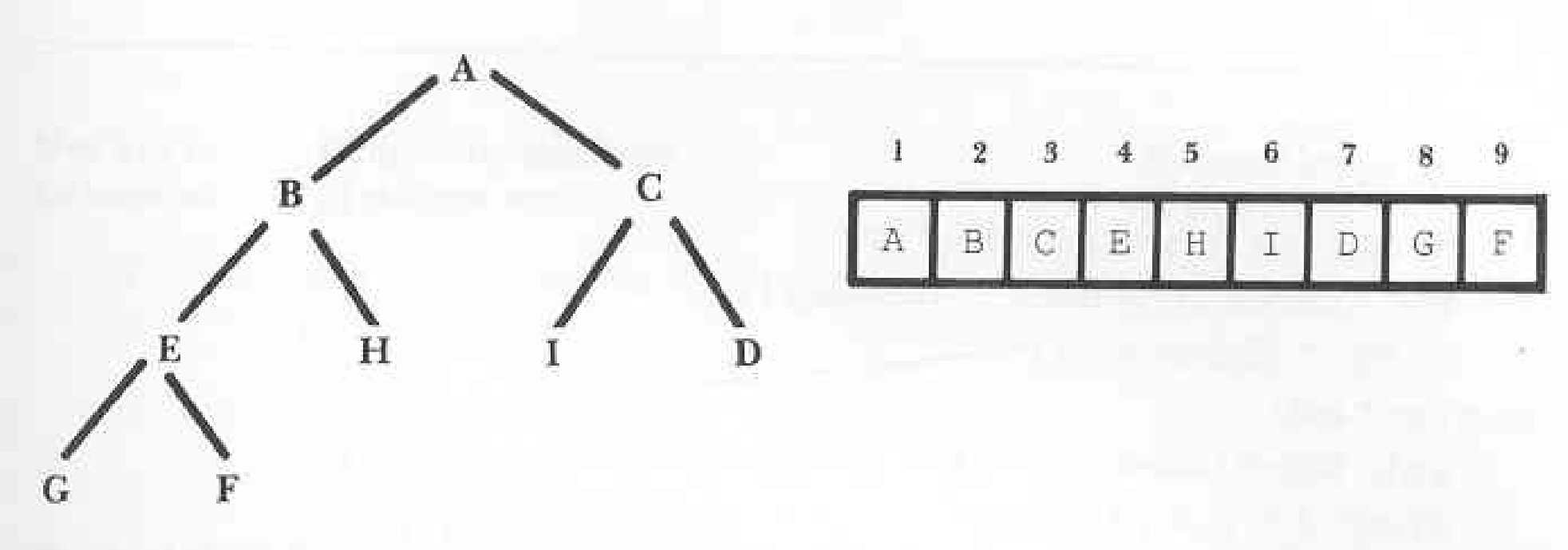
- Pergunta: há um algoritmo de ordenação interna que possa começar a ordenação conforme recebe os registros do arquivo?

  - Uma vez que ler um bloco do disco é demorado, e que ordenação tem custo O(n log n), há um algoritmo de ordenação que possa ser executado em paralelo com as leituras?

Ordenação em memória: intercalando E/S e processamento

- O heapsort resolve este problema mantendo os registros na estrutura heap, que é uma árvore binária com as seguintes propriedades:
  - cada nó tem 1 chave que é maior ou igual à chave do seu nó pai
  - é uma árvore binária completa
    - suas folhas estão em não mais que dois níveis
    - todas as chaves no nível mais baixo estão na posição mais esquerda
  - pode ser alocada sequencialmente em um vetor
    - raiz tem índice 1
    - filhos do nó *i* são *2i* (esquerdo) e *2i+1* (direito)

Ordenação em memória: intercalando E/S e processamento

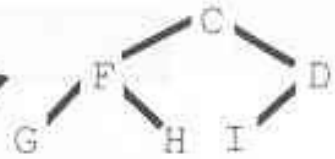
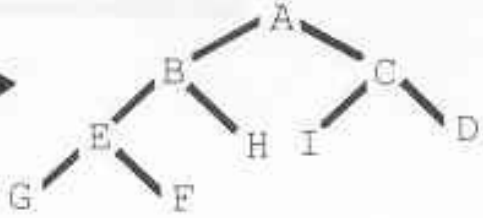


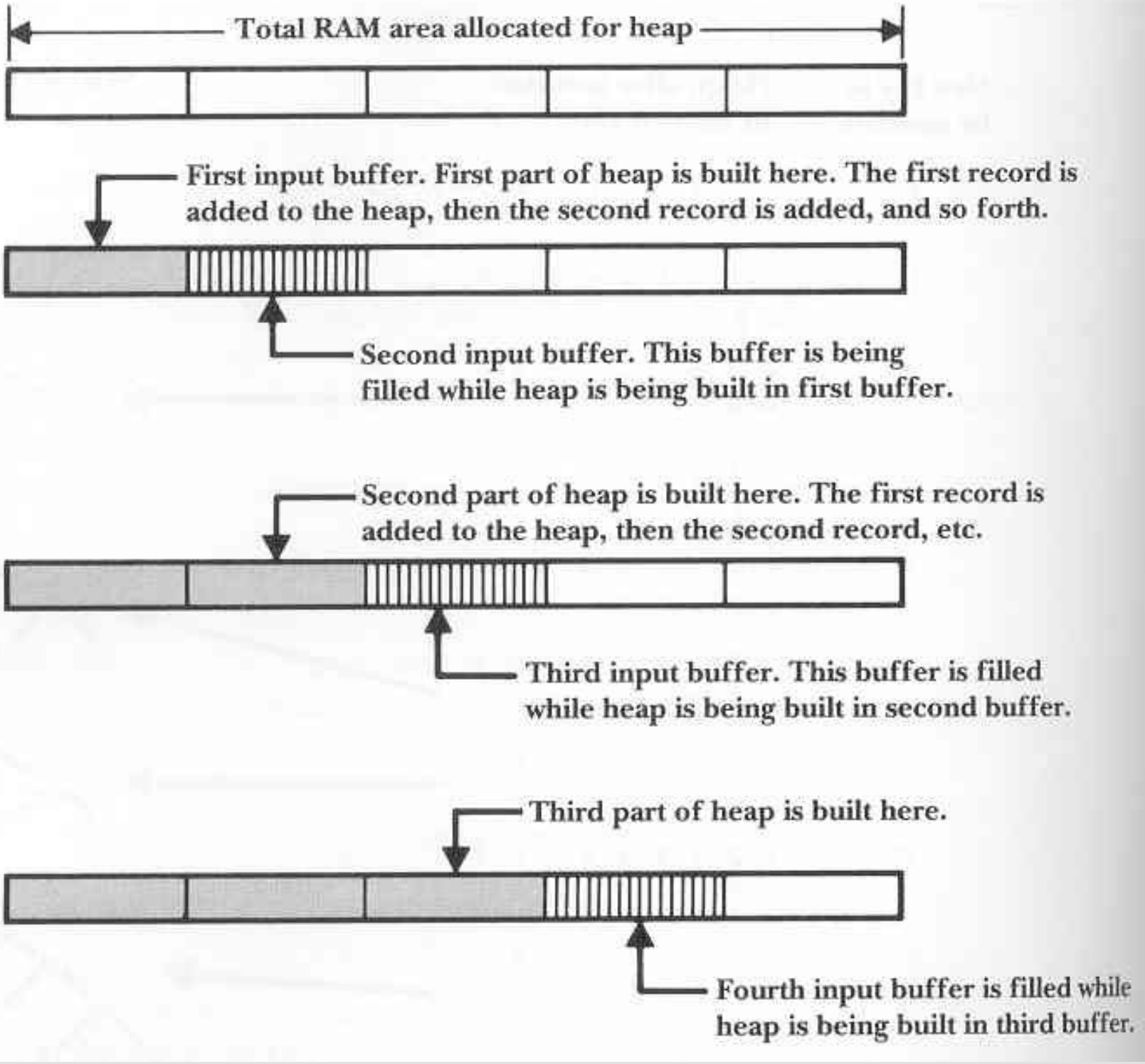
- Heap em forma de árvore e representação em vetor
  - garante que as posições das chaves no vetor são suficientes para impor uma ordem nas chaves

Ordenação em memória: intercalando E/S e processamento

- Funcionamento do Heapsort
  - primeiro passo: constrói-se o heap
  - segundo passo: apresenta-se a saída ordenada
- Primeiro passo pode ser executado enquanto lê-se os dados
  - Lê-se um bloco de registros em um buffer
  - Opera-se sobre os registros
  - Em termos de memória, o buffer de leitura pode ser apontado pela estrutura do heap no processamento

FDCGHIBEA

New key to be inserted	Heap, after insertion of the new key	Selected heaps in tree form
F	1 2 3 4 5 6 7 8 9 F	
D	1 2 3 4 5 6 7 8 9 D F	
C	1 2 3 4 5 6 7 8 9 C F D	
G	1 2 3 4 5 6 7 8 9 C F D G	
H	1 2 3 4 5 6 7 8 9 C F D G H	
I	1 2 3 4 5 6 7 8 9 C F D G H I	
B	1 2 3 4 5 6 7 8 9 B F C G H I D	
E	1 2 3 4 5 6 7 8 9 B E C F H I D G	
A	1 2 3 4 5 6 7 8 9 A B C E H I D G F	


Total RAM area allocated for heap
First input buffer. First part of heap is built here. The first record is added to the heap, then the second record is added, and so forth.
Second input buffer. This buffer is being filled while heap is being built in first buffer.
Second part of heap is built here. The first record is added to the heap, then the second record, etc.
Third input buffer. This buffer is filled while heap is being built in second buffer.
Third part of heap is built here.
Fourth input buffer is filled while heap is being built in third buffer.

Ordenação em memória: intercalando E/S e processamento

Ordenando enquanto se escreve no arquivo

- Escrevendo o heap no arquivo
  - o elemento na raiz do heap é o elemento de menor valor. Remove-se o elemento do heap, e número de elementos é decrementado
  - Heap é reordenado em memória com a troca do maior elemento com o menor de seus filhos até que o heap recupere a ordem
  - Ao se ter um número de registros que formem um bloco, esse bloco pode ser escrito em disco
    - pode-se liberar imediatamente essa memória
  - Toda entrada e saída é feita sequencialmente

Ordenação em memória: intercalando E/S e processamento

- Em um ambiente monotarefa, mesmo com o uso do heapsort, não é possível criar um algoritmo que permita sobrepor a ordenação com E/S
  - vantagem: toda E/S é feita sequencialmente
    - mesmo funcionamento para disco ou fita
    - a melhor solução que tinhamos era o keysort, que funciona bem para uma busca mas não para se obter o arquivo todo ordenado, pois resulta em um seek para cada registro

Ordenação de grandes arquivos com intercalação

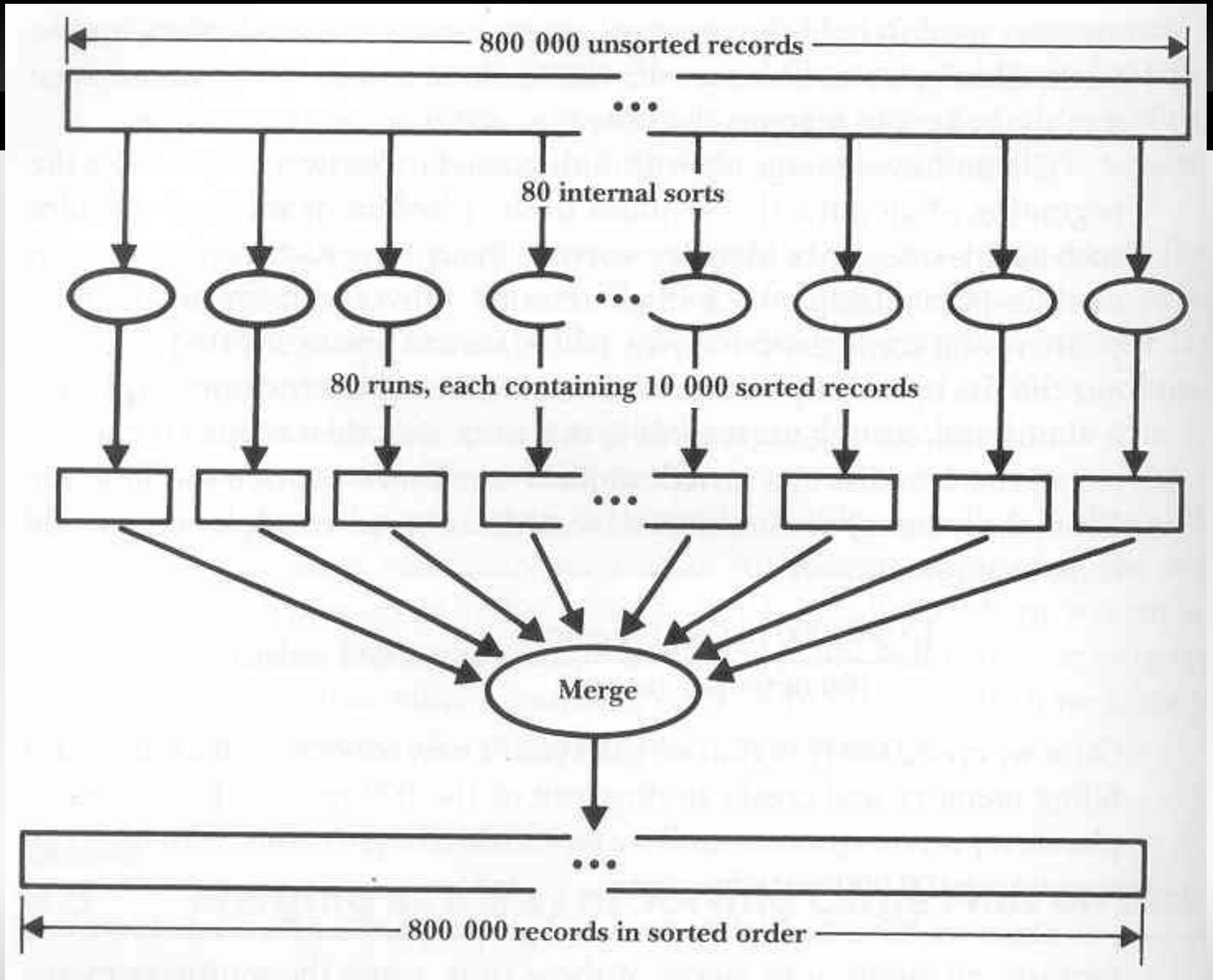
- Keysort permite ordenar um arquivo grande caso suas chaves caibam em memória
  - entretanto apresenta alto custo em número de acessos (seeks) para escrever o arquivo ordenado
- Porém, não permite a ordenação de arquivos realmente grandes
  - um arquivo com 8.000.000 de registros de 100 bytes cada
  - chave com 10 bytes
  - arquivo todo: 800 MB
  - se tivessemos disponível apenas 10 MB para o programa rodar, as chaves (80 MB) não caberiam em memória para o processamento do keysort

Ordenação de grandes arquivos com intercalação

- O algoritmo de intercalação de k-vias (*k-way merge*) provê uma solução atrativa
  - se houver 10 MB de memória disponível para processamento:
    - 10 MB / 100 bytes por registro = 100.000 registros
  - pode-se usar o heapsort para ordenar o subarquivo
  - cria-se k sub-arquivos até consumir o arquivo todo

Ordenação de grandes arquivos com intercalação

- Intercalação de k-vias tem as seguintes características:
  - pode de fato ordenar grandes arquivos
    - e pode até ser extendido para arquivos de quaisquer tamanhos
  - a leitura da entrada de um sub-arquivo é sequencial
    - bem melhor que 1 seek para cada registro (keysort)
  - a leitura entre cada sub-arquivo durante a intercalação e escrita é sequencial
    - acessos (seeks) são necessários ao se alternar entre os sub-arquivos durante a intercalação


800 000 unsorted records
80 internal sorts
80 runs, each containing 10 000 sorted records
Merge
800 000 records in sorted order

Exercício

- 1) O desempenho na ordenação normalmente é medido em termos do número de comparações. Explique porque o número de comparações não é adequado para medir o desempenho na ordenação de grandes arquivos.

Tempo estimado da ordenação por intercalação em disco

- Ambiente hipotético (suposições/simplificações):
  - Arquivos inteiros são armazenados em áreas contínuas do disco
    - acesso (seek) cilindro a cilindro leva tempo igual a zero
    - com isso, apenas um acesso (seek) é necessário para o acesso sequencial de um arquivo
  - Sistema é monotarefa

Tempo estimado da ordenação por intercalação em disco

- Quatro tipos de acesso de E/S na ordenação por intercalação:
  - (1) Leitura de todos os registros na memória para ordenação e criação dos sub-arquivos
  - (2) Escrita dos sub-arquivos ordenados no disco
  - (3) Leitura dos sub-arquivos ordenados na memória para intercalação
  - (4) Escrita do arquivo ordenado no disco

Tempo estimado da ordenação por intercalação em disco

- (1) Leitura de todos os registros na memória para ordenação e criação dos sub-arquivos
  - Leituras de pedaços de 10 MB por vez
    - Custo de acesso = 8 ms (busca) + 3 ms (atraso rotacional) = 11 ms
    - Taxa de transmissão: 14.500 bytes / ms
  - 80 pedaços de 10 MB
    - Custo total
      - Acesso: 80 x 11 ms = 1 segundo
      - Transmissão: 800 MB / 14.500 bytes/ms = 60 segundos
      - Total = 61 segundos

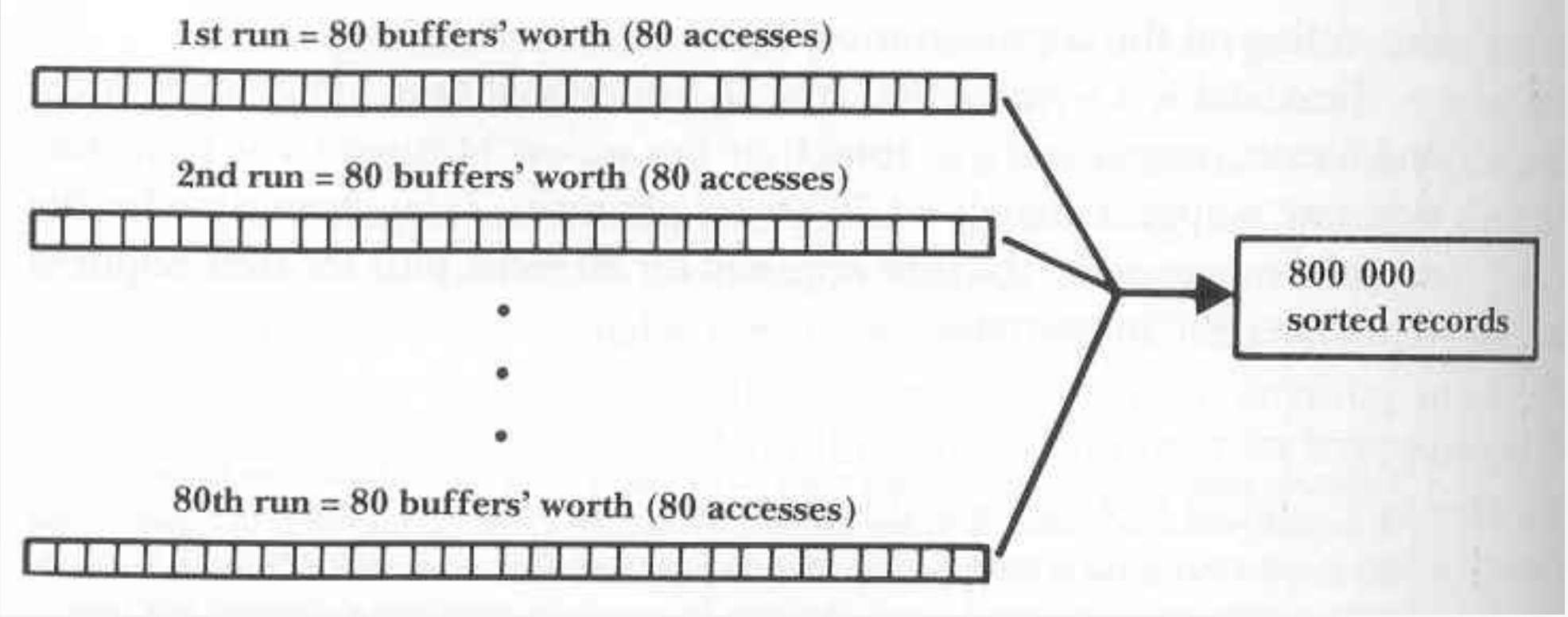
Tempo estimado da ordenação por intercalação em disco

- (2) Escrita dos sub-arquivos ordenados no disco
  - Nesse caso, tempo de escrita é o mesmo da leitura
    - Mesmo número de seeks
    - Mesma quantidade de dados para transferir
    - Total = 61 segundos

Tempo estimado da ordenação por intercalação em disco

- (3) Leitura dos sub-arquivos ordenados na memória para intercalação
  - 10 MB de memória repartidos em 80 sub-arquivos
    - Um buffer contém 1/80 da memória = 125.000 bytes
    - Cada sub-arquivo deve ser acessado 80 vezes para ser lido para a memória
    - São necessários 80 sub-arquivos x 80 acessos = 6.400 acessos ao total
    - Custo acesso = 6.400 x (8 + 3) ms = 70 segundos
    - Custo transferência = 800 MB / 14.500 bytes/ms = 60 s
    - Custo total = 130 segundos

Tempo estimado da ordenação por intercalação em disco



Tempo estimado da ordenação por intercalação em disco

- (4) Escrita do arquivo ordenado no disco
  - Necessário conhecer os buffers de saída.
    - supondo que tem-se 2 buffers de saída com 200.000 bytes cada um
  - Com buffers de 200.000 bytes, tem-se:
    - 800 MB / 200.000 bytes por acesso = 4.000 acessos
    - Custo acesso = 4.000 x (8 + 3) ms = 44 segundos
    - Custo transferência = 800 MB / 14.500 bytes/ms = 60 s
    - Custo total = 104 segundos

Tempo estimado da ordenação por intercalação em disco

- Tempo total da intercalação: 356 segundos
  - 122 segundos na ordenação
  - 234 segundos na intercalação

	Number of seeks	Amount transferred (megabytes)	Seek + rotation time (seconds)	Transfer time (seconds)	Total time (seconds)
Sort: reading	800	800	1	60	61
Sort: writing	800	800	1	60	61
Merge: reading	6400	800	70	60	130
Merge: writing	4000	800	44	60	104
Totals	10 560	3200	116	240	356

Tempo estimado da ordenação por intercalação em disco

- Comparação com o keysort
  - Keysort realiza uma busca para cada registro lido no arquivo
  - Com 800.000 registros no arquivo → 800.000 seeks
  - Tendo 11 ms por busca, o custo de acesso é:
    - de 800.000 x 11 ms = 8.800 segundos = 2 horas, 26 minutos e 40 segundos

Ordenação de um arquivo 10 vezes maior

- Fases: ordenação e intercalação
  - Heapsort durante a ordenação garante que E/S é sequencial → não há mais espaço para melhorias
  - Na leitura da intercalação, há um buffer de memória para cada sub-arquivo → são carregados aleatoriamente
  - O número e o tamanho dos buffers de memória de leitura dos dados determinam o número de vezes que se tem os acessos aleatórios
  - A escrita da intercalação não é influenciada pela organização, como na fase de ordenação

Ordenação de um arquivo 10 vezes maior

- Aumento de 10 vezes do arquivo sem o aumento da memória
  - 800 sub-arquivos, com 800 buffers na intercalação
  - Como cada buffer comporta 1/800 do sub-arquivo, vão ser necessários 800 acessos por sub-arquivo
  - São necessários 800 sub-arquivos x 800 acessos = 640.000 acessos ao total
  - Tempo total = 2 horas e 24 minutos
    - quase 40 vezes mais que a intercalação para o arquivo de 800 MB

Ordenação de um arquivo 10 vezes maior

	Number of seeks	Amount transferred (megabytes)	Seek + rotation time (seconds)	Transfer time (seconds)	Total time (seconds)
Merge: reading	640 000	8000	7040	600	7640
Merge: writing	40 000	8000	440	600	1040
Totals	680 000	16 000	7480	1200	8680

Ordenação de um arquivo 10 vezes maior

- A operação de intercalação requer k^2 acessos
  - arquivo x 10 → número de acessos x 100
  - complexidade da ordenação é $O(k^2)$ acessos (seeks)
- Redução do tempo: ordenação por intercalação:
  - Alocar mais hardware: discos, memória e canais de E/S
  - Fazer a intercalação aumentando o tamanho do buffer para cada sub-arquivo
  - Aumentar o tamanho dos sub-arquivos ordenados
  - Paralelizar as operações de E/S

Ordenação de um arquivo 10 vezes maior

- Aumento da quantidade de memória
  - Uma memória maior reduziria o número sub-arquivos → redução no número total de acessos
    - Com 40 MB de memória, o tempo total da ordenação por intercalação do arquivo de 80.000.000 de registros passaria de 2 horas, 6 minutos e 40 segundos para 16 minutos e 40 segundos → melhoria de 7 vezes

Ordenação de um arquivo 10 vezes maior

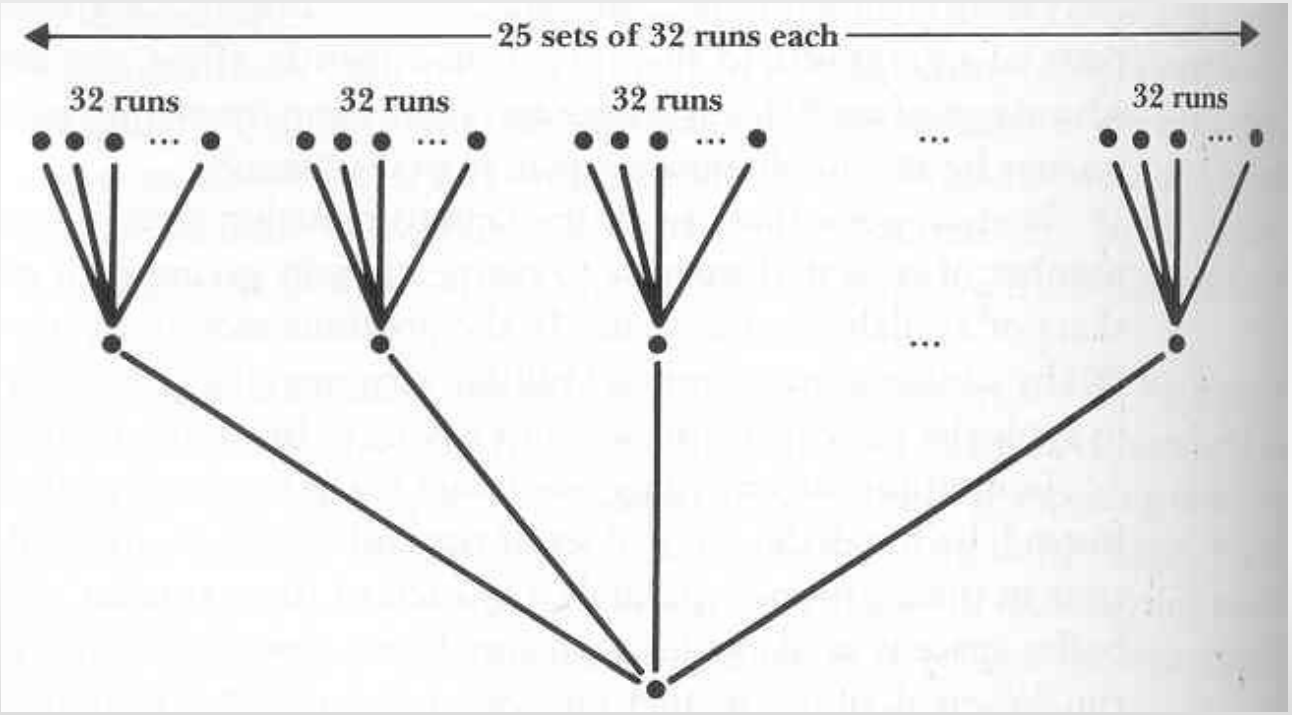
- Aumento do número de discos dedicados
  - Se não há contenção pelo uso do disco, a fonte primária de atraso seria o atraso rotacional e o atraso de transferência
    - Se há um disco para leitura e outra para a escrita, as operações de leitura e escrita poderiam ocorrer simultaneamente
- Aumento do número de canais de E/S
  - Se existe um canal para cada disco, as operações de E/S podem ocorrer de forma simultânea

Diminuição do número de acessos com intercalação de vários passos

- Objetivo é não intercalar todos os sub-arquivos de uma vez
  - Os conjuntos de sub-arquivos originais são quebrados em grupos menores e a intercalação é feita para esses grupos
  - Como as intercalações tem menor ordem, sobra mais espaço de buffer para cada sub-arquivo
    - menos acessos são necessários
  - Ao final das intercalações menores, um segundo passo de intercalação é feito

Diminuição do número de acessos com intercalação de vários passos

- Será que as vantagens da estratégia em dois passos compensam os custos extras de reler e rescrever todos os registros?
  - 25 grupos de 32 sub-arquivos = 800 sub-arquivos de 10 mil registros



Diminuição do número de acessos com intercalação de vários passos

- Primeira passagem da intercalação
  - Cada buffer comporta 1/32 de um sub-arquivo
    - resulta em 32 x 32 = 1.024 acessos
  - Para os 25 grupos, temos 25 x 1.024 = 25.600 acessos
    - cada sub-arquivo resultante tem 320.000 registros ou 320 MB

Diminuição do número de acessos com intercalação de vários passos

- Segunda passagem da intercalação
  - Cada buffer comporta 1/25 da memória ou 1/800 de cada sub-arquivo
    - 25 x 800 = 20.000 acessos
- Total de acessos
  - 25.600 + 20.000 = 45.600 acessos
  - Há também o tempo de transferência, que é dobrado
    - os registros são lidos e escritos 2 vezes na intercalação
  - Tempo total na intercalação = 3.782 segundos (1 hora e 3 minutos)

Diminuição do número de acessos com intercalação de vários passos

	Number of seeks	Amount transferred (megabytes)	Seek + rotation time (seconds)	Transfer time (seconds)	Total time (seconds)
1st Merge: reading	25 600	8000	282	600	882
1st Merge: writing	40 000	8000	440	600	1040
2nd Merge: reading	20 000	8000	220	600	820
2nd Merge: writing	40 000	8000	440	600	1040
Totals	125 600	32 000	1382	2400	3782

- Note que transferência resultante é mais cara que o tempo de acessos (seek+rotation)
  - nesse exemplo, 2 passos é o ponto limite de diminuição do tempo total
    - aumentar um passo não vai decrescer o tempo total

E agora: mais discos? mais processadores?

- Diversas arquiteturas
  - Sistema operacional: monotarefa ou multitarefa?
  - Vários discos: leitura em um disco e escrita em outro? e a velocidade dos barramentos?
  - Múltiplos processadores:
    - multiprocessamento simétrico
    - clusters (papel do barramento desempenhado pela rede)
  - Computadores paralelos massivos
    - Supercomputador com 136 núcleos (68 processadores Itanium), 272 GB de RAM, 30 TB de armazenamento
  - Processadores vetoriais
    - clusters de Playstation 3

Conclusões

- Use o heapsort para formar a lista original de elementos ordenados de um sub-arquivo
  - buffering duplo → pode-se sobrepor E/S e processamento interno
- Use o máximo de memória possível
  - produz sub-arquivos maiores e provê buffers maiores para a fase de intercalação

Conclusões

- Se o número de sub-arquivos é tão grande que os tempos de atraso rotacional e busca são muito maiores que os tempos de transferência, use intercalação em vários passos
- Use mais de um disco e de canal de E/S de forma que entrada e saída possam se sobrepor
  - principalmente se não existem outros usuários no sistema
- Conheça os fundamentos de ordenação externa e seus custos relativos e procure aproveitar novas arquiteturas e sistemas

Exercício teórico

- Leia o capítulo e faça um resumo incluindo:
  - operações co-sequenciais
  - modelo de implementação
  - intercalação multivias
  - intercalação de E/S e processamento ordenação
  - ordenação de grandes arquivos por intercalação
  - tempo estimado para intercalação em disco
  - otimização com intercalação de vários passos

Leitura complementar

- Capítulo **”Cosequencial Processing and the Sorting of Large Files”** do livro
  - Folk et al. ”File Structures: An Object-Oriented Approach with C++”, Editora Pearson, 3ª edição, 1998