



Ordenação Externa

SCC0607 – Estrutura de Dados III

Anderson Canale Garcia

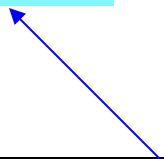
Material adaptado de:

Cristina D. Aguiar



Ordenação

A N A		R U A	<i>b</i>	A U G U S T O	<i>b</i>	P A I V A		I B A T E		<i>b b</i>		
A N T O N I A		R U A	<i>b</i>	X V	<i>b</i>	D E	<i>b</i>	M A I O		I B A T E		<i>b</i>
J O A O		R U A	<i>b</i>	A		R I O	<i>b</i>	C L A R O		<i>b b b b b b b b</i>		
M A R I A		R U A	<i>b</i>	1		S A O	<i>b</i>	C A R L O S		<i>b b b b b b b</i>		
P E D R O		R U A	<i>b</i>	X V		S A O	<i>b</i>	C A R L O S		<i>b b b b b b b</i>		



ordenação baseada em um determinado campo,
usando suas chaves



Ordenação em Memória Interna

* Arquivo cabe em RAM *



Ordenação em Memória Interna

- Etapas

- leitura de todos os registros do arquivo
- ordenação dos registros em RAM
- escrita dos registros ordenados no arquivo

- Custo

- soma dos custos das 3 etapas
- leitura e escrita sequenciais minimizam *seeks*
- uso de métodos eficientes de ordenação interna



Heapsort

- Melhora o desempenho da ordenação interna
 - Paraleliza a ordenação com o processamento de entrada e saída (ou seja, leitura e escrita de registros)

parte 1 = leitura; parte 2 = ordenação

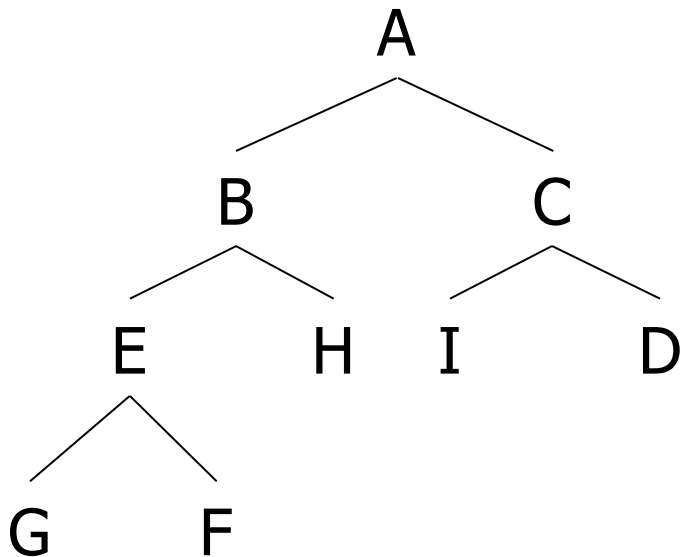
parte 1 = ordenação; parte 2 = escrita

possíveis
paralelismos



Heap

- Estrutura que mantém as chaves
- Árvore binária, implementada como vetor



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

Filhos de i : $2i$ e $2i+1$

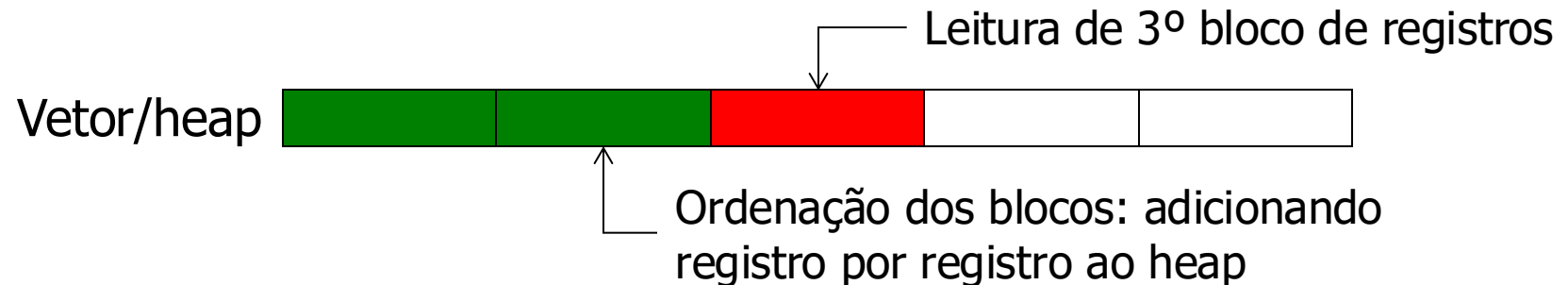
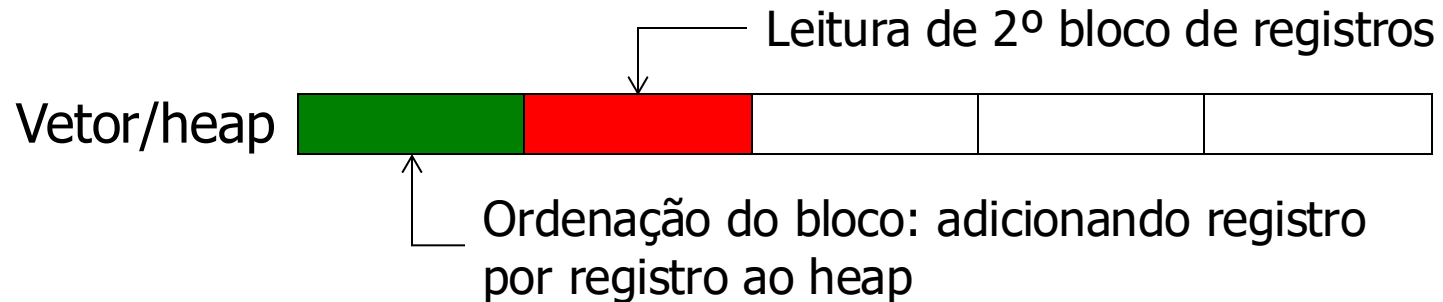
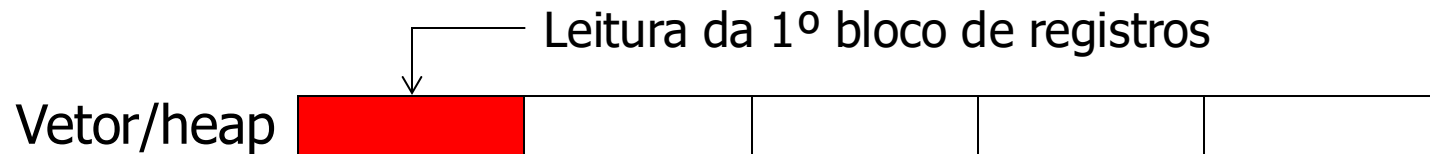
Pai de j : $\lfloor j/2 \rfloor$



Ordenação Interna Heapsort

* Paralelizando leitura com ordenação *

Paralelismo leitura/ordenação



...



Construção do heap

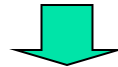
- Algoritmo
 - insere o elemento no fim do vetor
 - enquanto o elemento for menor do que o seu pai, troca-o de lugar com o pai
- Exemplo
 - *heap* com 9 posições
 - chaves: F, D, C, G, H, I, B, E, A



Construção do *heap*

- Elemento: F

1	2	3	4	5	6	7	8	9



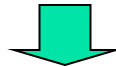
1	2	3	4	5	6	7	8	9
F								



Construção do *heap*

- Elemento: D

1	2	3	4	5	6	7	8	9
F	D							



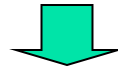
1	2	3	4	5	6	7	8	9
D	F							



Construção do *heap*

- Elemento: C

1	2	3	4	5	6	7	8	9
D	F	C						



1	2	3	4	5	6	7	8	9
C	F	D						



Construção do *heap*

- Elemento: G

1	2	3	4	5	6	7	8	9
C	F	D	G					



Construção do *heap*

- Elemento: H

1	2	3	4	5	6	7	8	9
C	F	D	G	H				



Construção do *heap*

- Elemento: I

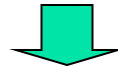
1	2	3	4	5	6	7	8	9
C	F	D	G	H	I			



Construção do *heap*

- Elemento: B

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



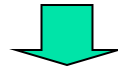
1	2	3	4	5	6	7	8	9
B	F	C	G	H	I	D		



Construção do *heap*

- Elemento: E

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



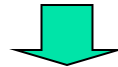
1	2	3	4	5	6	7	8	9
B	E	C	F	H	I	D	G	



Construção do *heap*

- Elemento: A

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



Ordenação Interna Heapsort

* Paralelizando ordenação com escrita *



Algoritmo

- Passos

- recupera o registro da raiz do *heap*
- enquanto rearranja o *heap*, grava esse registro no arquivo de saída

- Rearranjo do *heap*

- retira o elemento da raiz
- coloca o último elemento k do *heap* como raiz
- enquanto k for maior do que seus filhos, troca-o de lugar com seu menor filho



Exemplo

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

Recupera-se raiz A, colocando em seu lugar F

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

Enquanto grava A no arquivo ordenado, rearranja heap

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



Exemplo

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

Recupera-se raiz B, colocando em seu lugar G

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

Enquanto grava B no arquivo ordenado, rearranja *heap*

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

E assim por diante, até *heap* esvaziar



Ordenação Externa

* Arquivo não cabe em RAM *



Sort-Merge Externo

- Fase 1
 - cria subarquivos ordenados (i.e., *runs*) a partir do arquivo original
- Fase 2
 - combina os subarquivos ordenados em subarquivos ordenados maiores até que o arquivo completo esteja ordenado



Ordenação por Intercalação de Arquivos em Disco

- **k-Way Mergesort**, ou intercalação de k listas, em disco
- Ao invés de considerar os registros individualmente, podemos considerar **blocos de registros ordenados** (corridas, ou *runs*)
 - Para minimizar os seeks
- Método envolve 2 fases: geração das corridas (*runs*, conjuntos de dados), e intercalação



Intercalação em k-vias

- Esta solução
 - Pode **ordenar arquivos realmente grandes**
 - Geração das corridas envolve apenas acesso seqüencial aos arquivos
 - A leitura das corridas e a escrita final também só envolve acesso seqüencial
 - Aplicável também a arquivos mantidos em fita, já que E/S é seqüencial

g	24
a	19
d	31
c	33
b	14
e	16
r	16
d	21
m	3
p	2
d	7
a	14

arquivo
original

fase 1

g	24
a	19
d	31
c	33
b	14
e	16
r	16
d	21
m	3
p	2
d	7
a	14

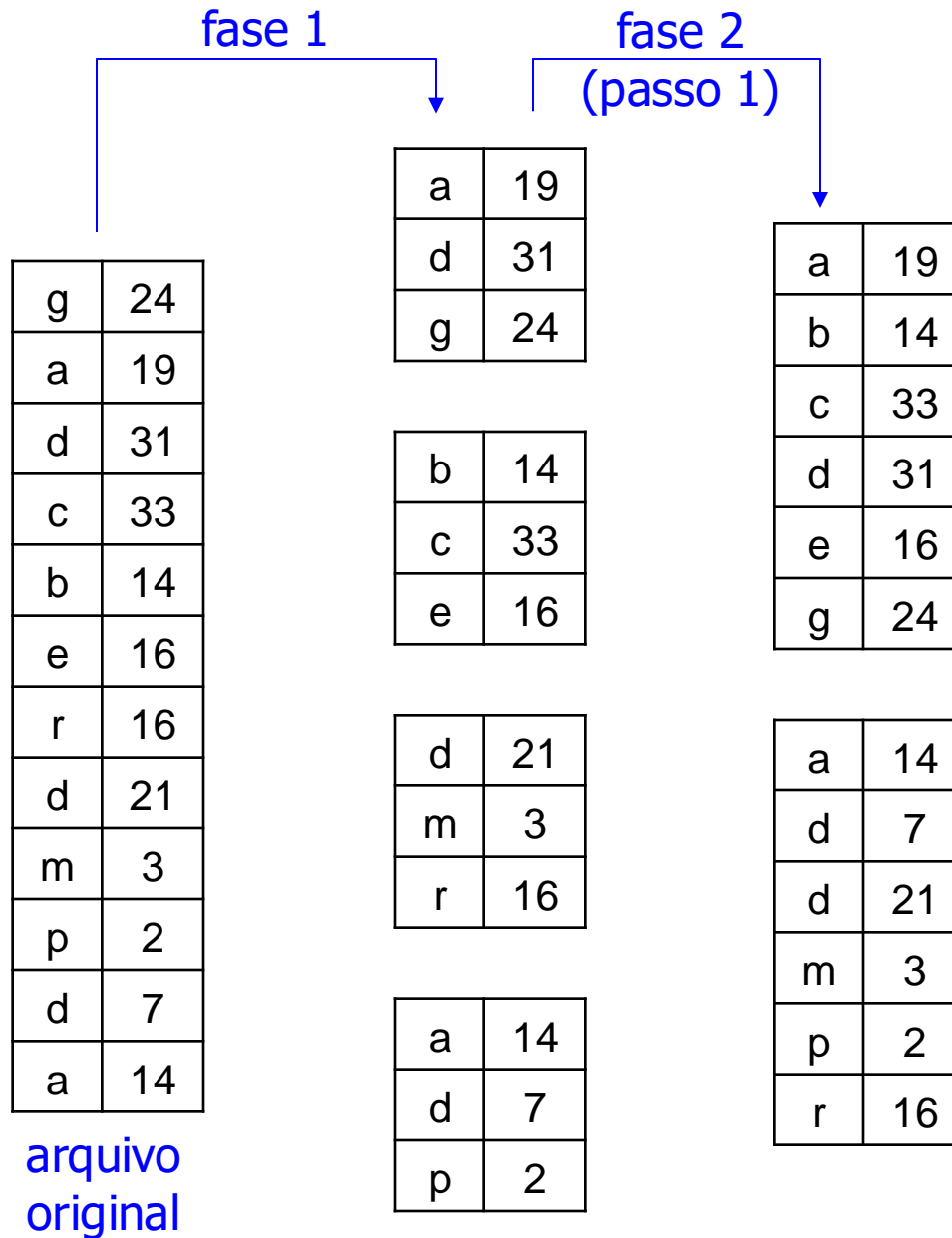
arquivo
original

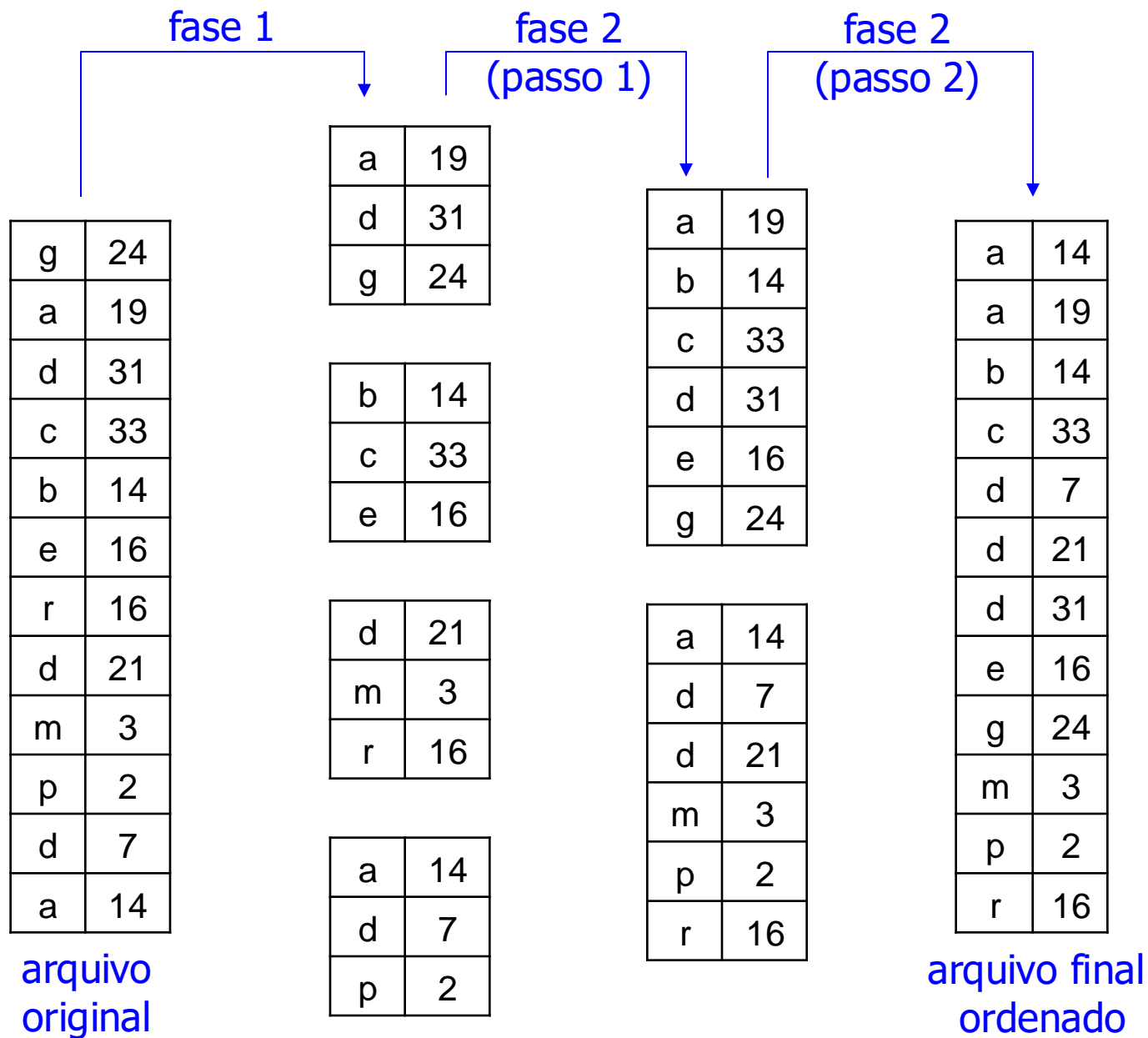
a	19
d	31
g	24

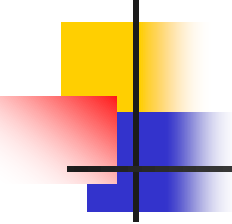
b	14
c	33
e	16

d	21
m	3
r	16

a	14
d	7
p	2

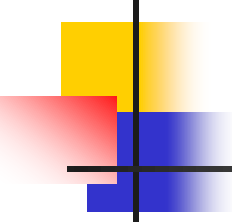






Qual o custo (tempo) do MergeSort ?

- Supondo
 - Arquivo com 80 MB, com registros de 100 bytes, e cada corrida com 1 MB
 - 1MB = 10.000 registros
 - Arquivo armazenado em áreas contíguas do disco (*extents*), *extents* alocados em mais de uma trilha, de tal modo que um único *rotational delay* é necessário para cada acesso
 - Características do disco
 - tempo médio para seek: 18 ms
 - atraso rotacional: 8.3 ms
 - taxa de transferência: 1229 bytes/ms
 - tamanho da trilha: 20.000 bytes



Qual o custo (tempo) do MergeSort ?

- Quatro passos a serem considerados
 - Leitura dos registros, do disco para a memória, para criar as 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 necessário 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 1.229 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) → 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 Mergesort = 537 s



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



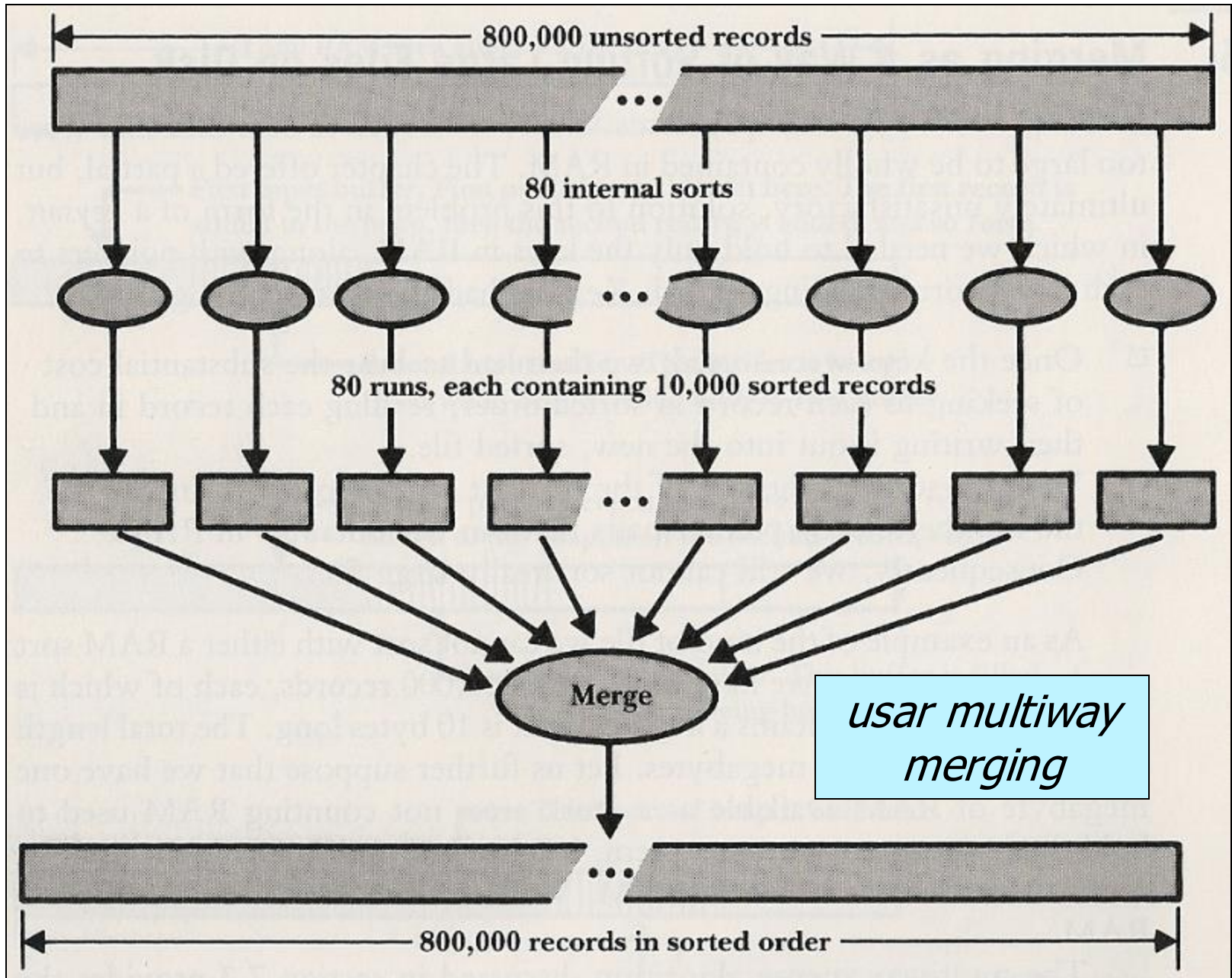
Ordenação de um arquivo com 8.000.000 de registros

- 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 10 apenas vezes menor do que este)



Ordenação de um arquivo com 8.000.000 de registros

Definitivamente: necessário **diminuir o tempo gasto** obtendo dados na fase de intercalação





Multiway Merging

- Árvore de seleção
 - Tipo de uma árvore de torneio
 - Guarda a menor das chaves
- Menor chave
 - sempre está na raiz da árvore
 - garante fácil recuperação



Multiway Merging

- Algoritmo
 - Indica de qual arquivo foi obtida a menor chave
 - Lê a próxima chave desse arquivo
 - Reestrutura a árvore de seleção
- Número de níveis da árvore
 - $\approx \log_2 K$
 - onde K é o número de arquivos de dados

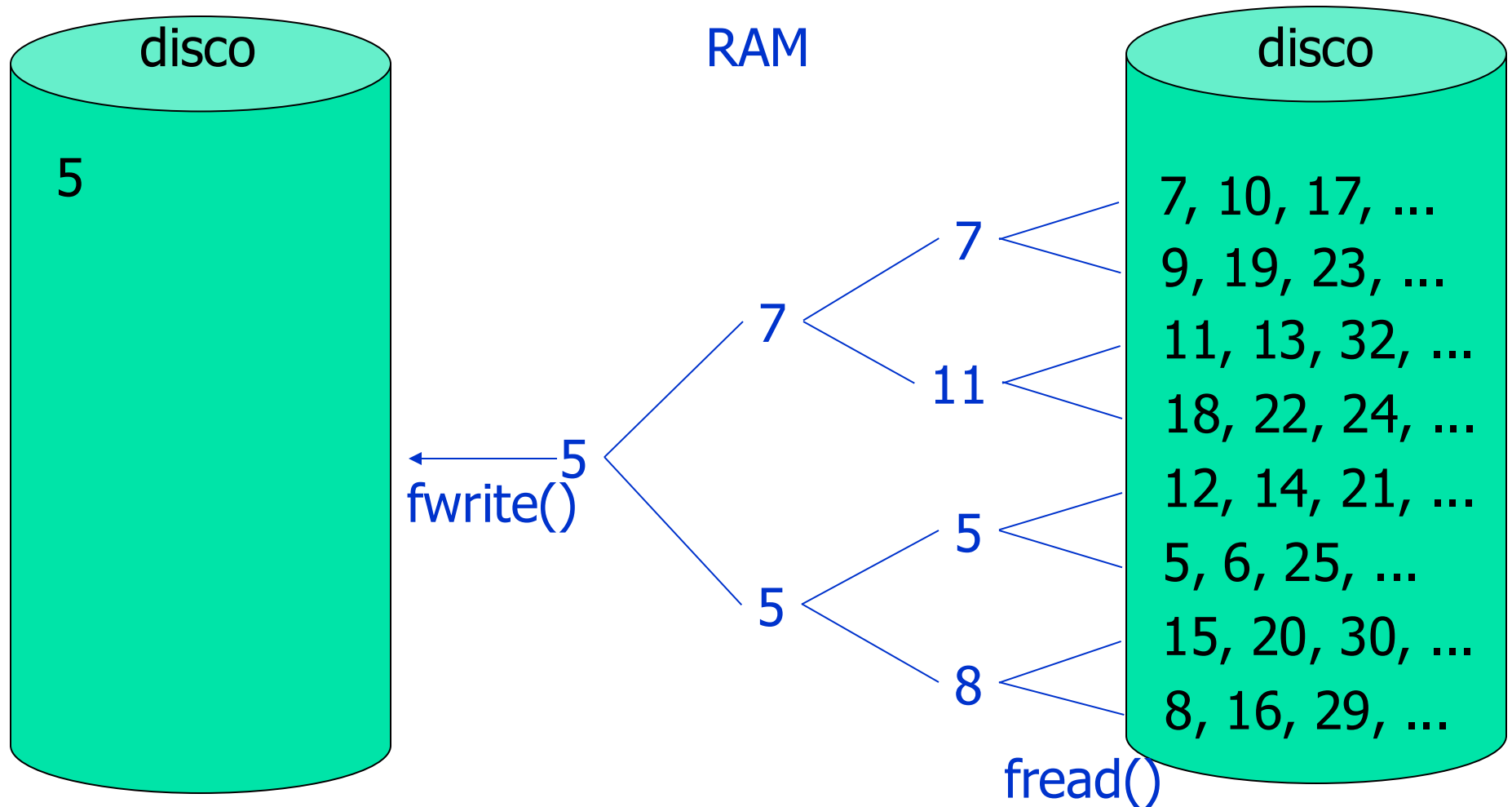


Multiway Merging ($K = 8$)

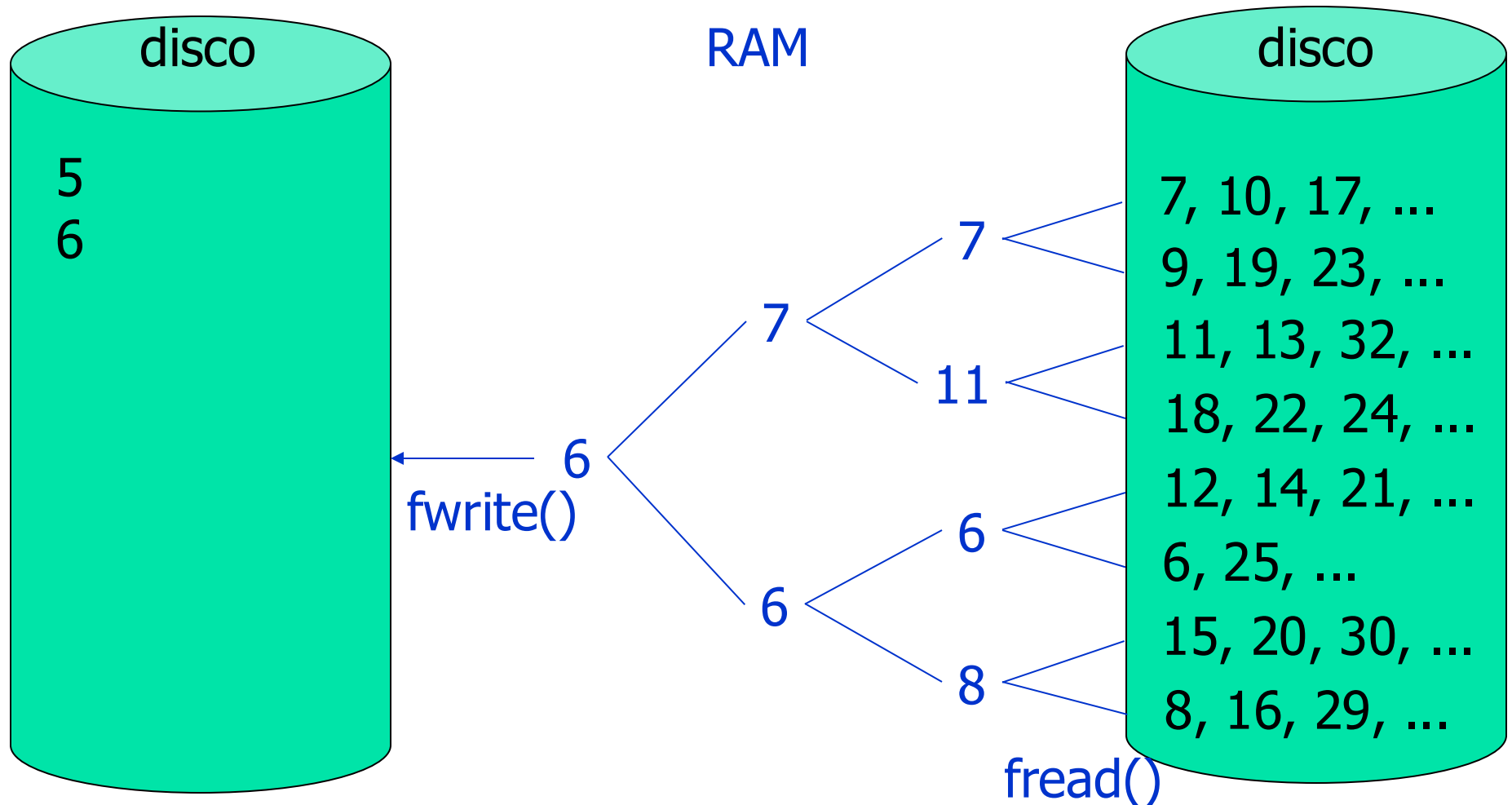
disco

7, 10, 17, ...
9, 19, 23, ...
11, 13, 32, ...
18, 22, 24, ...
12, 14, 21, ...
5, 6, 25, ...
15, 20, 30, ...
8, 16, 29, ...

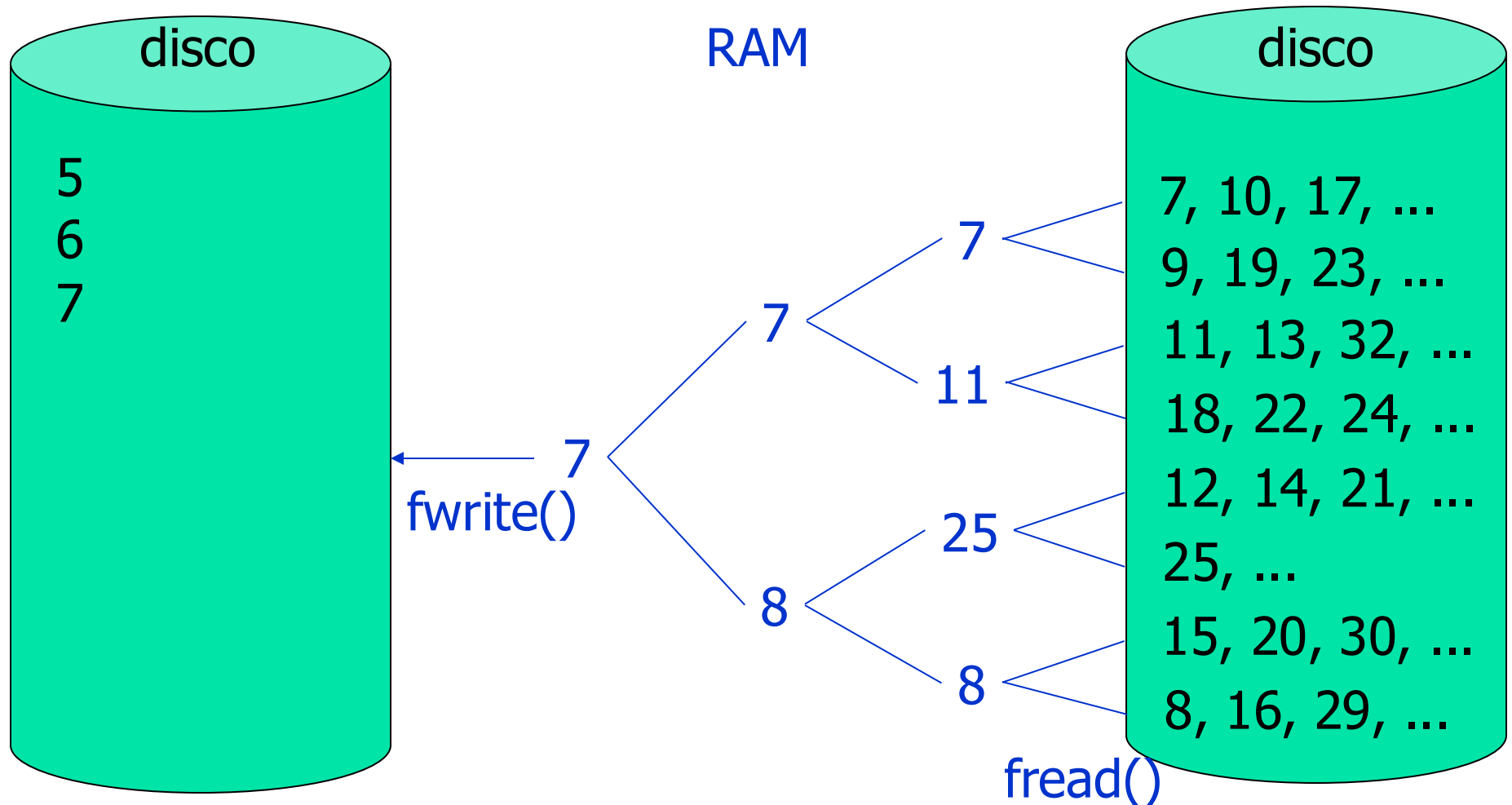
Multiway Merging (K = 8)



Multiway Merging (K = 8)



Multiway Merging (K = 8)



Multiway Merging (K = 8)

