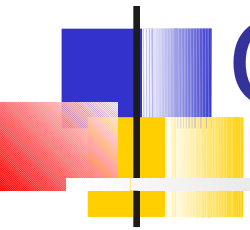


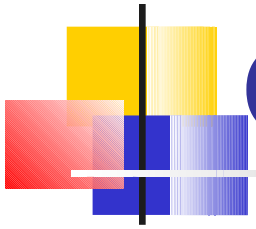
Processamento Coseqüencial

SCC-503 – Algoritmos e Estruturas de Dados II

Thiago A. S. Pardo

Leandro C. Cintra

M.C.F. de Oliveira



Operações Coseqüenciais

- Envolvem o processamento coordenado (simultâneo) de duas ou mais listas de entrada seqüenciais, de modo a produzir uma única lista como saída

  - Exemplo: *merging* (união/intercalação) ou *matching* (intersecção) de duas ou mais listas ordenadas mantidas em arquivo

Exemplo: como seria o merging?

Lista1	**Lista2**
Adams	Adams
Carter	Anderson
Chin	Andrews
Davis	Bech
Foster	Rosewald
Garwich	Schmidt
Rosewald	Thayer
Turner	Walker
	Willis

Exemplo: como seria o matching?

Lista1	**Lista2**
Adams	Adams
Carter	Anderson
Chin	Andrews
Davis	Bech
Foster	Rosewald
Garwich	Schmidt
Rosewald	Thayer
Turner	Walker
	Willis

Modelo para implementação de processos coseqüenciais

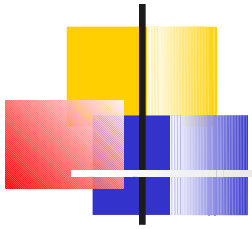
- Algoritmo de *merging* (união)
  - Entrada: 2 listas organizadas alfabeticamente
  - Lê um nome de cada lista e compara-os
    - Se ambos são iguais, copia o nome para a saída e avança para o próximo nome da lista em cada arquivo
    - Se o nome da Lista1 é menor, ele é copiado para a saída e avança-se na Lista1
    - Se o nome da Lista1 é maior, copia o nome da Lista2 para a saída e avança-se na Lista2

Modelo para implementação de processos coseqüenciais

- Pontos importantes
  - Inicialização
  - Acesso ao próximo item
  - Sincronização
  - Condições de fim de arquivo
  - Reconhecimento de erros: duplicidade, ordenação

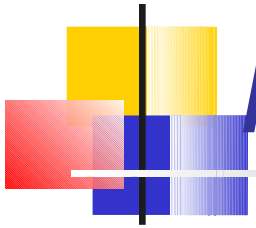
Algoritmo: merging

```
//abre arquivos lista1 e lista2, cria arquivo saída
inicializa()

//verifica se foi possivel ler itens de uma das listas
mais_itens = proxItem(lista1) || proxItem(lista2)

enquanto (mais_itens) faca {
  se (item(1) < item(2)) //funcoes retornam itens atuais
     escreva item(1) em saída
     mais_itens = proxItem(lista1) || currItem(lista2)
  senão se (item(2) < item(1))
     escreva item(2) em saída
     mais_itens = proxItem(lista2) || currItem(lista1)
  senão
     escreve item(1) em saída
     mais_itens = proxItem(lista1) || proxItem(lista2)
}

finaliza() //fecha arquivos
```



Cuidados no merging

- Garantir que os itens estão ordenados
  - Comparação do atual com o anterior
- Quando uma lista acabar, a outra lista deve continuar a ser processada, podendo ser copiada diretamente na saída
- Cuidado com repetição de itens

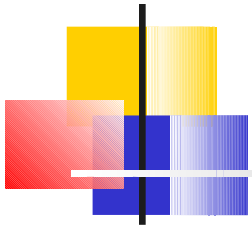


Matching

- Como funcionaria o *matching*?
  - Esboce o algoritmo

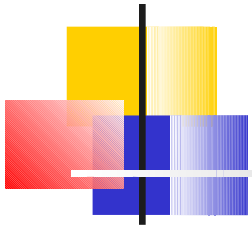
Algoritmo: matching

```
//abre arquivos lista1 e lista2, cria arquivo saída
inicializa()
//verifica se foi possivel ler itens de ambas as listas
mais_itens = proxItem(lista1) && proxItem(lista2)

enquanto (mais_itens) faca {
   se (item(1) < item(2))
      mais_itens = proxItem(lista1)
   senão se (item(2) < item(1))
       mais_itens = proxItem(lista2)
   senão
       escreve item(1) em saída
       mais_itens = proxItem(lista1) && proxItem(lista2)
}

finaliza() //fecha arquivos
```



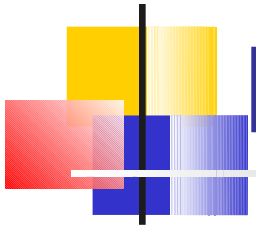
Cuidados no matching

- Garantir que os itens estão ordenados nas listas
  - Comparação do nome atual com o nome anterior
- Quando uma das listas terminar, encerra-se o processo
- Cuidado com repetição



Observações

- Em alguns casos o arquivo de saída pode ser um dos arquivos de entrada
- Cada arquivo está ordenado por um ou mais campos, e todos com a mesma ordenação
  - Mas não é necessário que todos tenham a mesma organização ou estrutura
- Pode-se usar valores máximos e mínimos conhecidos para a chave por segurança e facilidade de verificação de fim de arquivo
- Registros serão processador em ordem lógica de organização



Multiway merging

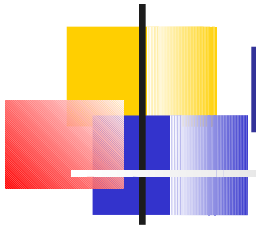
- Merging de mais de 2 listas
  - Processamento coseqüencial mais comum

- Processo de procurar o menor item das listas para escrever e a seguir ler novo item da lista correspondente

Multiway merging

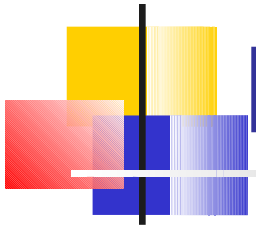
- Algoritmo

```
...
enquanto (mais_itens) {
   menor_item = minItem(item(1) , item(2), ..., item(k))
   escreva menor_item em saída
   mais_itens = falso
   para (i = 0; i < k; i++) {
      se (item(i) == menor_item)
         mais_itens = proxItem(lista(i)) || mais_itens
      senao
         mais_itens = currItem(lista(i)) || mais_itens
   }
}
...
```



Multiway merging

- Encontrar o mínimo e a qual lista pertence é um processo caro.
- Se um dado item ocorre em apenas uma das listas, um procedimento mais eficiente e simples pode ser usado
  - Usar como base os vetores **lista** e **item**
    - Referência às listas: lista[1], ..., lista[K]
    - Itens atuais das listas: item[1], ..., item[K]



Multiway merging

- Algoritmo 2

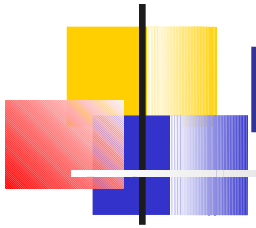
```
para (i = 0; i < k; i++)
      mais_itens = mais_itens || proxItem(lista[i])

enquanto (mais_itens) {
   ind_min = minItemIndex(item, k) // pega indice
   escreva item[ind_min] em saída
   mais_itens[ind_min] = proxItem(lista[ind_min])
}
...
```

Multiway merging

- Algoritmo 2

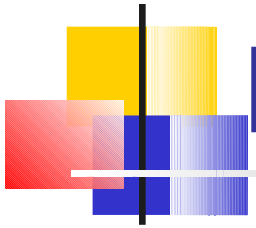
```
para (i = 0; i < k; i++)
      mais_itens = mais_itens || proxItem(lista[i])

enquanto (mais_itens) {
   ind_min = minItemIndex(item, k)
   escreva item[ind_min] em saída
   mais_itens[ind_min] = proxItem(lista[ind_min])
}
...
```

Não tem que se procurar pela lista



Multiway merging

- Com muitas listas, vale a pena usar uma árvore de seleção
  - Como uma “árvore de torneio”, que guarda a menor das chaves
  - A menor chave sempre está na raiz da árvore, sendo fácil recuperá-la, portanto
    - Se ela indicar a lista de onde veio, basta se ler novamente da lista correspondente e reestruturar a árvore



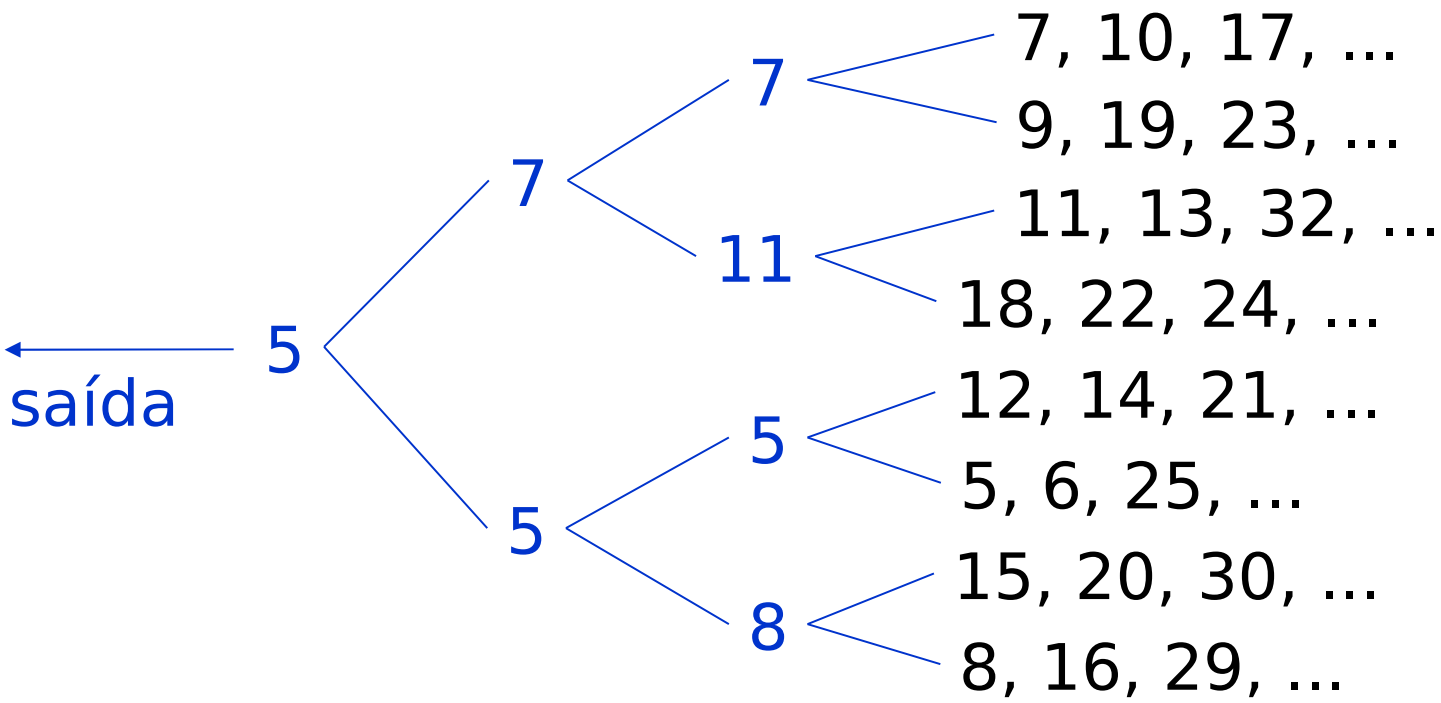
Multiway merging

- Exemplo com K=8 listas de números

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

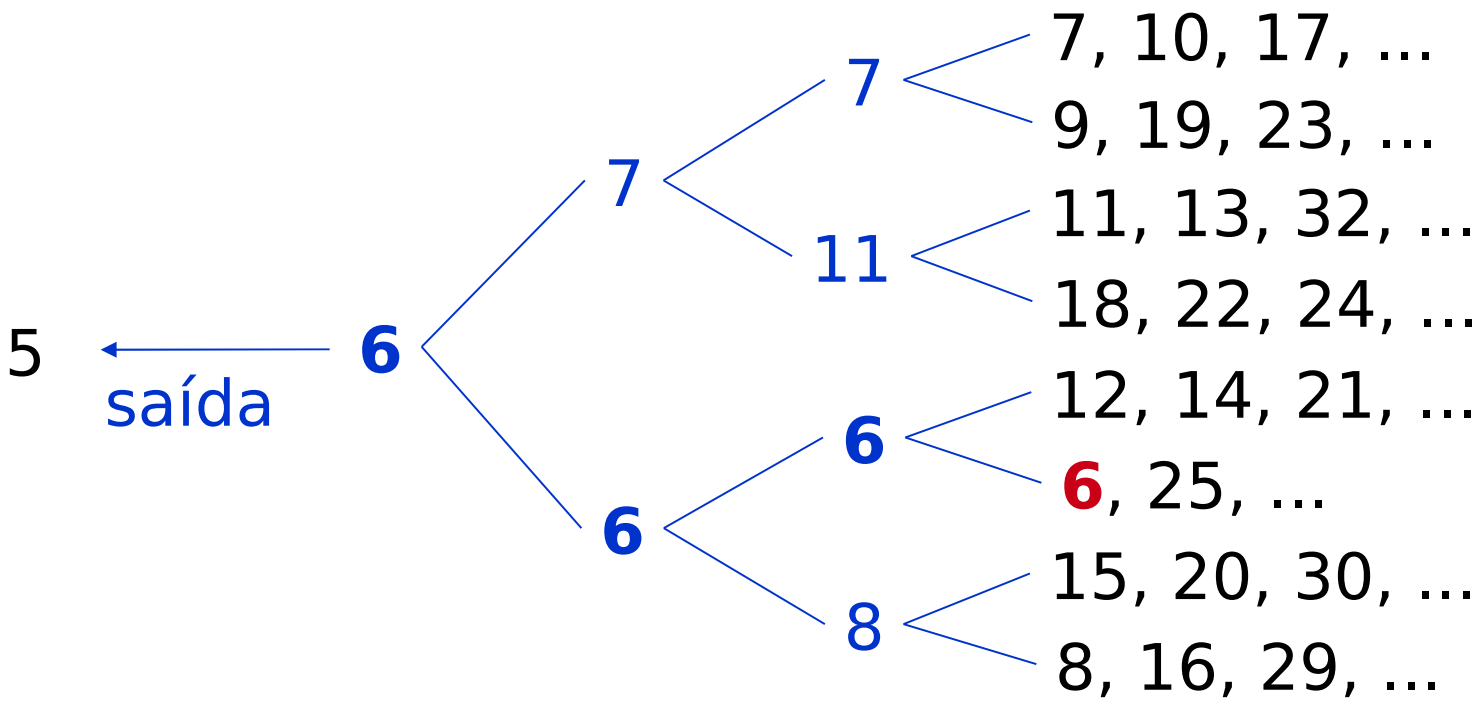
Multiway merging

- Exemplo com K=8 listas de números
  - Número de níveis da árvore $\left(\log_2(k)\right)$



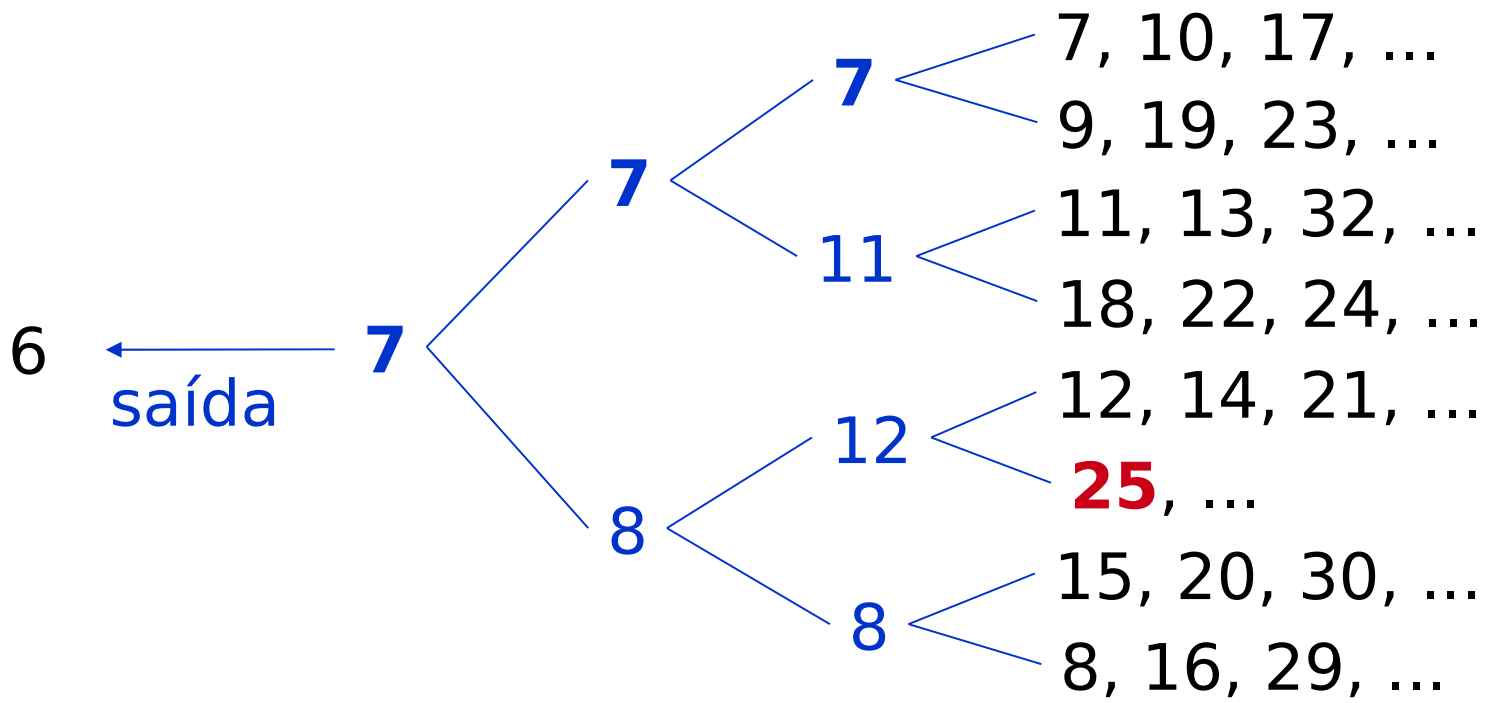
Multiway merging

- Exemplo com K=8 listas de números
  - Após processar o 5



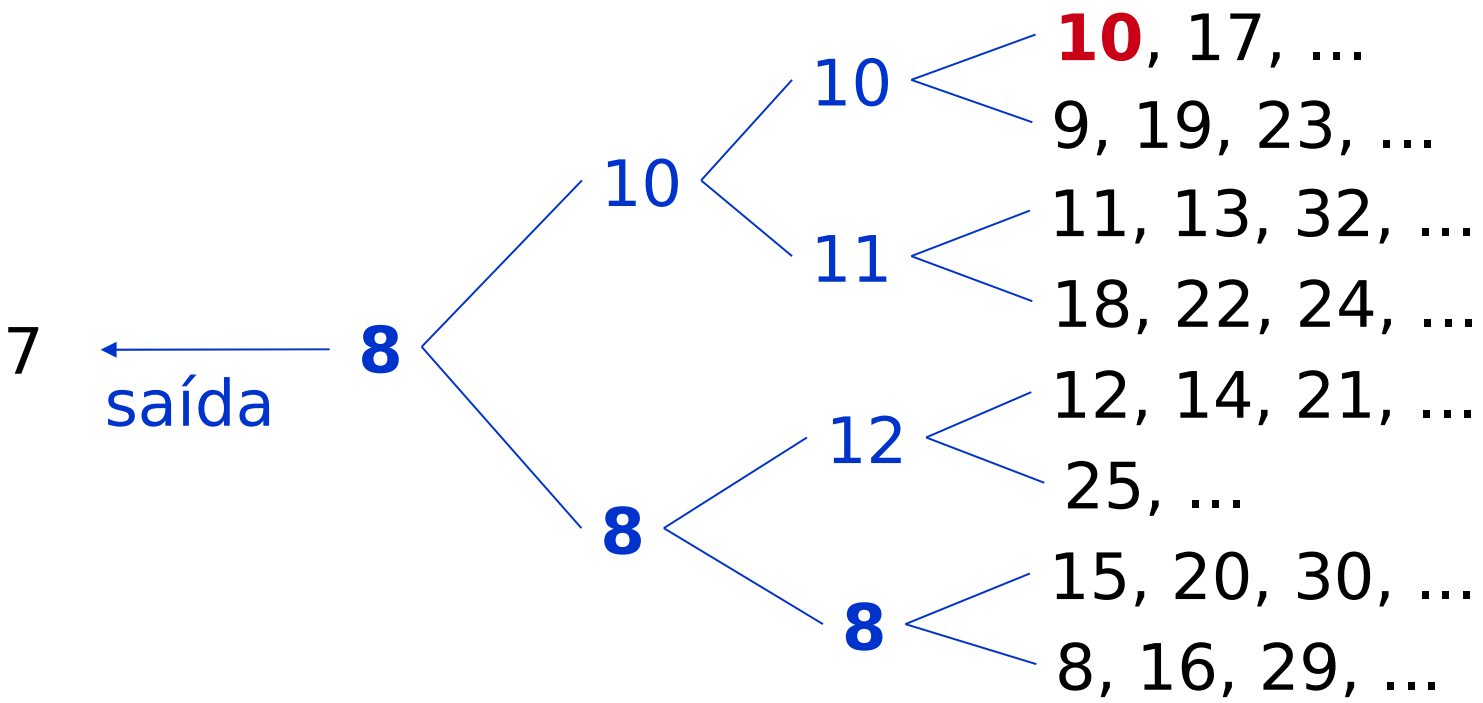
Multiway merging

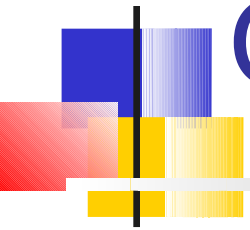
- Exemplo com K=8 listas de números
  - Após processar o 6



Multiway merging

- Exemplo com K=8 listas de números
  - Após processar o 7

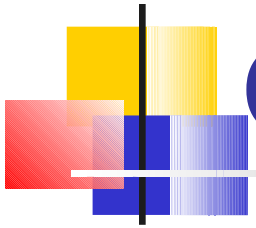




Ordenação em RAM

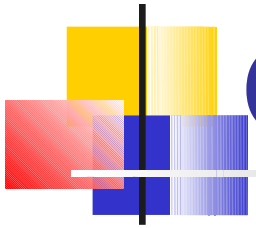
Ordenação em RAM

- Situação: arquivo cabe em RAM
  - 3 processos
    - Leitura de todos os registros do arquivo
    - Ordenação dos registros em memória
    - Escrita dos registros ordenados no arquivo
      - Custo = soma dos custos dos 3 processos
        - Muito melhor do que ordenação do arquivo no próprio arquivo



Ordenação em RAM

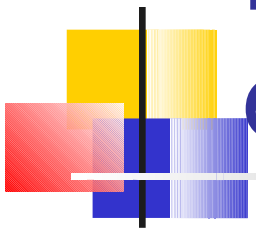
- Situação: arquivo cabe em RAM
  - 3 processos
    - Leitura de todos os registros do arquivo
    - Ordenação dos registros em memória
    - Escrita dos registros ordenados no arquivo
      - Leitura e escrita seqüenciais → ótimo, pois minimiza número de seeks
      - Escolha de um bom método de ordenação interna: heapsort, por exemplo
      - Dá para melhorar ainda mais? Como?



Ordenação em RAM

- Situação: arquivo cabe em RAM
  - 3 processos
    - Leitura de todos os registros do arquivo
    - Ordenação dos registros em memória
    - Escrita dos registros ordenados no arquivo
      - Leitura e escrita seqüenciais → ótimo, pois minimiza número de seeks
      - Escolha de um bom método de ordenação interna: heapsort, por exemplo
      - Paralelizando ordenação com leitura/escrita de registros

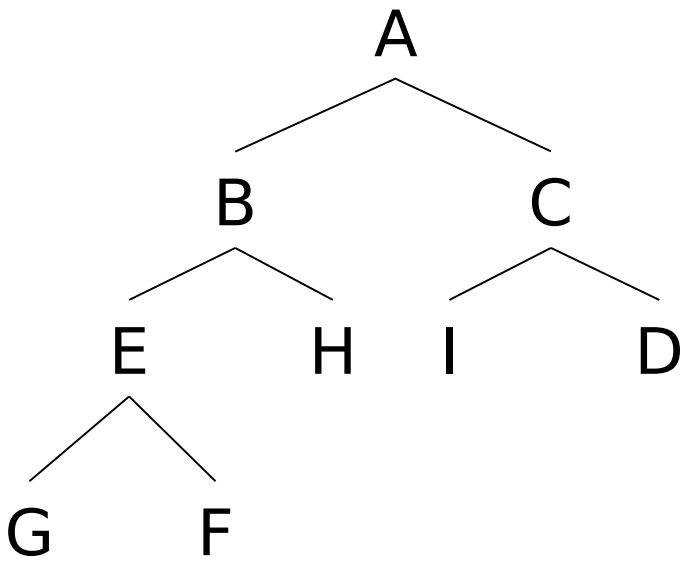
Processamento e entrada/saída: paralelizando



- Seria ótimo poder ordenar os dados disponíveis enquanto outros estão sendo lidos
  - Heapsort permite isso!
    - Começa ordenando um conjunto inicial de registros, construindo o heap
    - Conforme mais dados vão chegando, eles vão sendo incorporados no heap

Processamento e entrada/saída: paralelizando

- Exemplo de heap mínimo
  - Em árvore (binária completa) e em 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: j/2

Processamento e entrada/saída: paralelizando

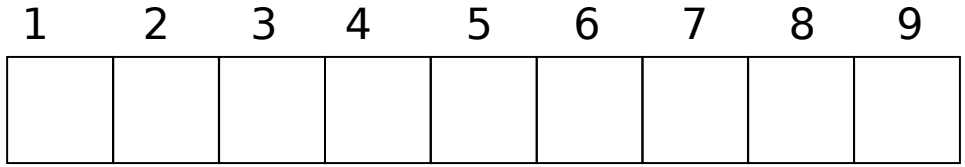
- Construção do heap

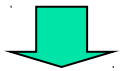
1. Insere-se um novo elemento no fim do vetor

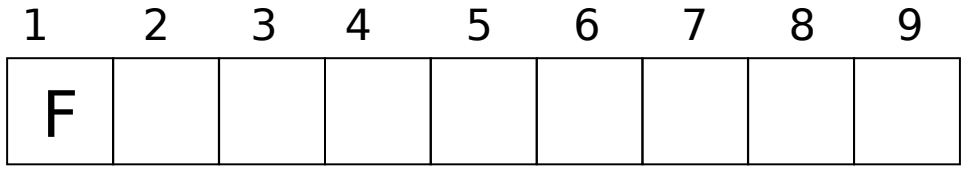
1. Enquanto menor do que seu pai, troca-o de lugar com o pai

Processamento e entrada/saída: paralelizando

- Elemento: F

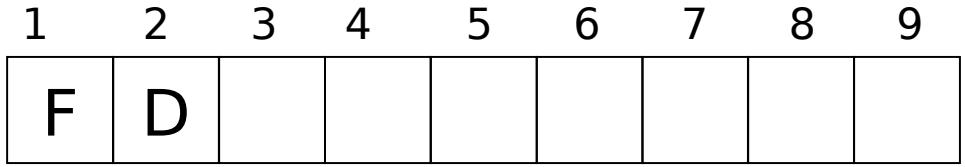


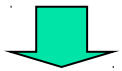


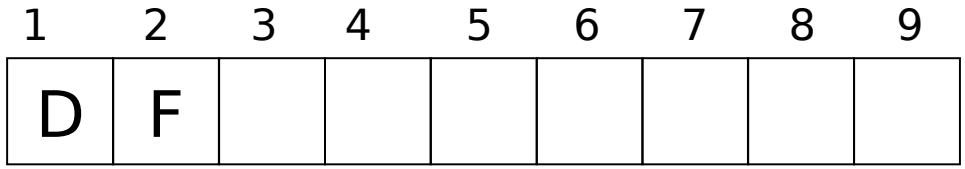


Processamento e entrada/saída: paralelizando

- Elemento: D

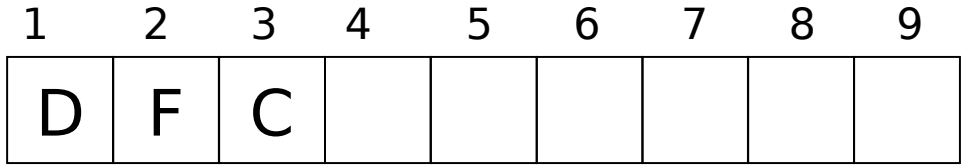


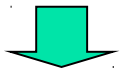


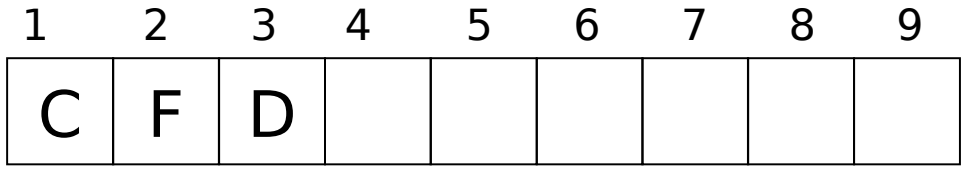


Processamento e entrada/saída: paralelizando

- Elemento: C

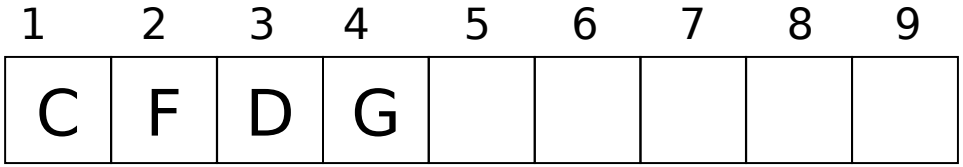






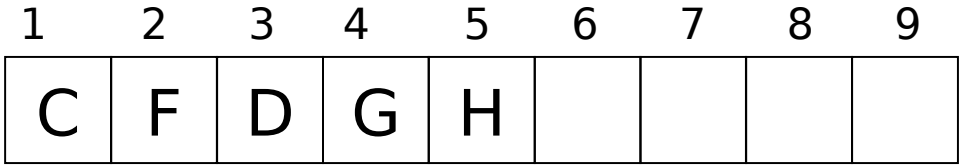
Processamento e entrada/saída: paralelizando

- Elemento: G



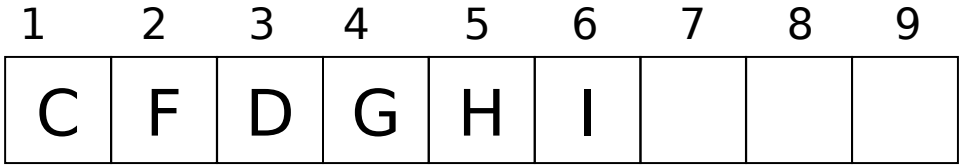
Processamento e entrada/saída: paralelizando

- Elemento: H



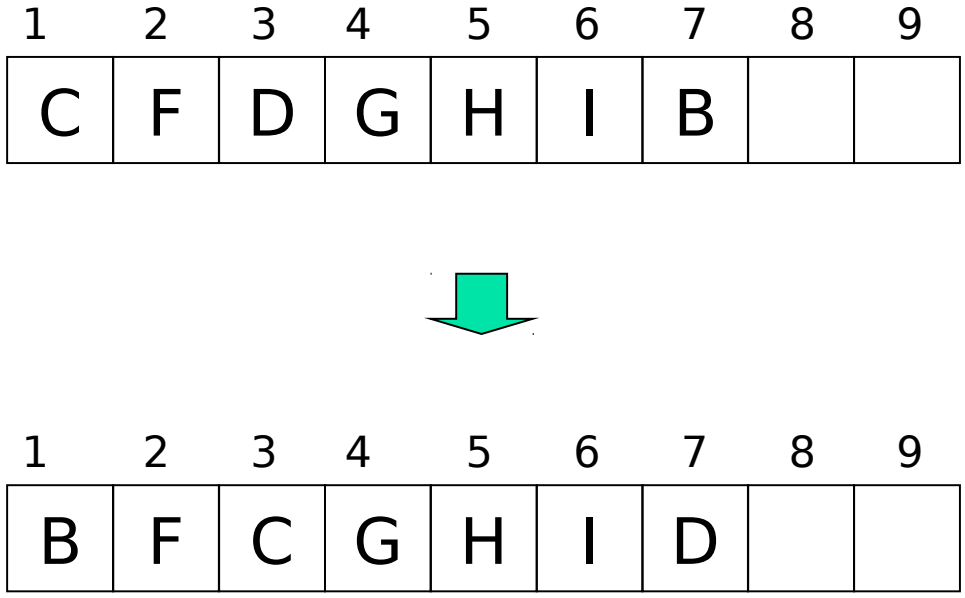
Processamento e entrada/saída: paralelizando

- Elemento: I



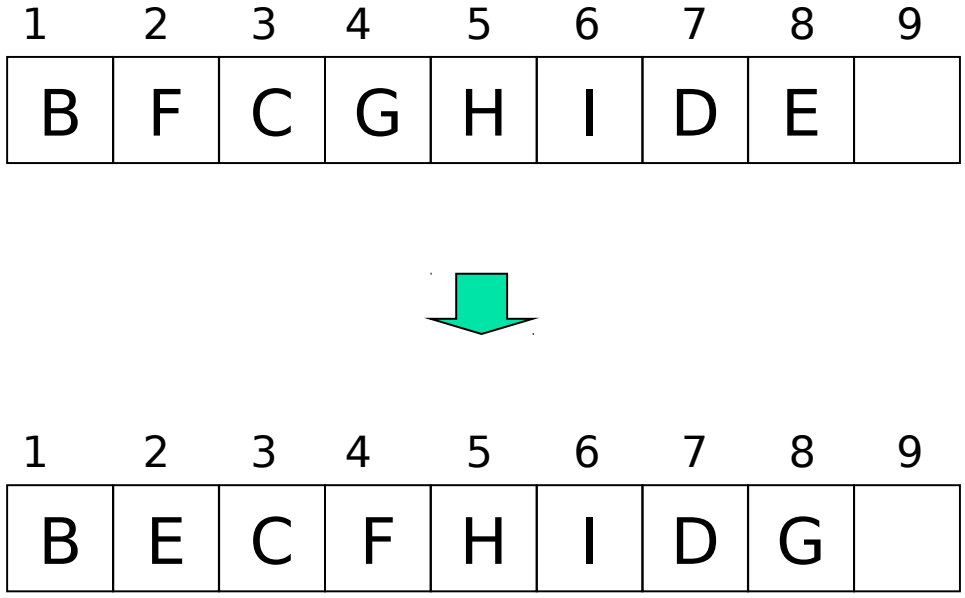
Processamento e entrada/saída: paralelizando

- Elemento: B



Processamento e entrada/saída: paralelizando

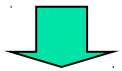
- Elemento: E



Processamento e entrada/saída: paralelizando

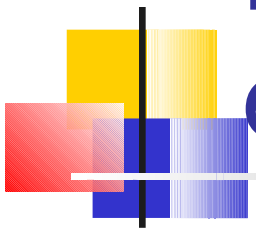
- Elemento: A

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



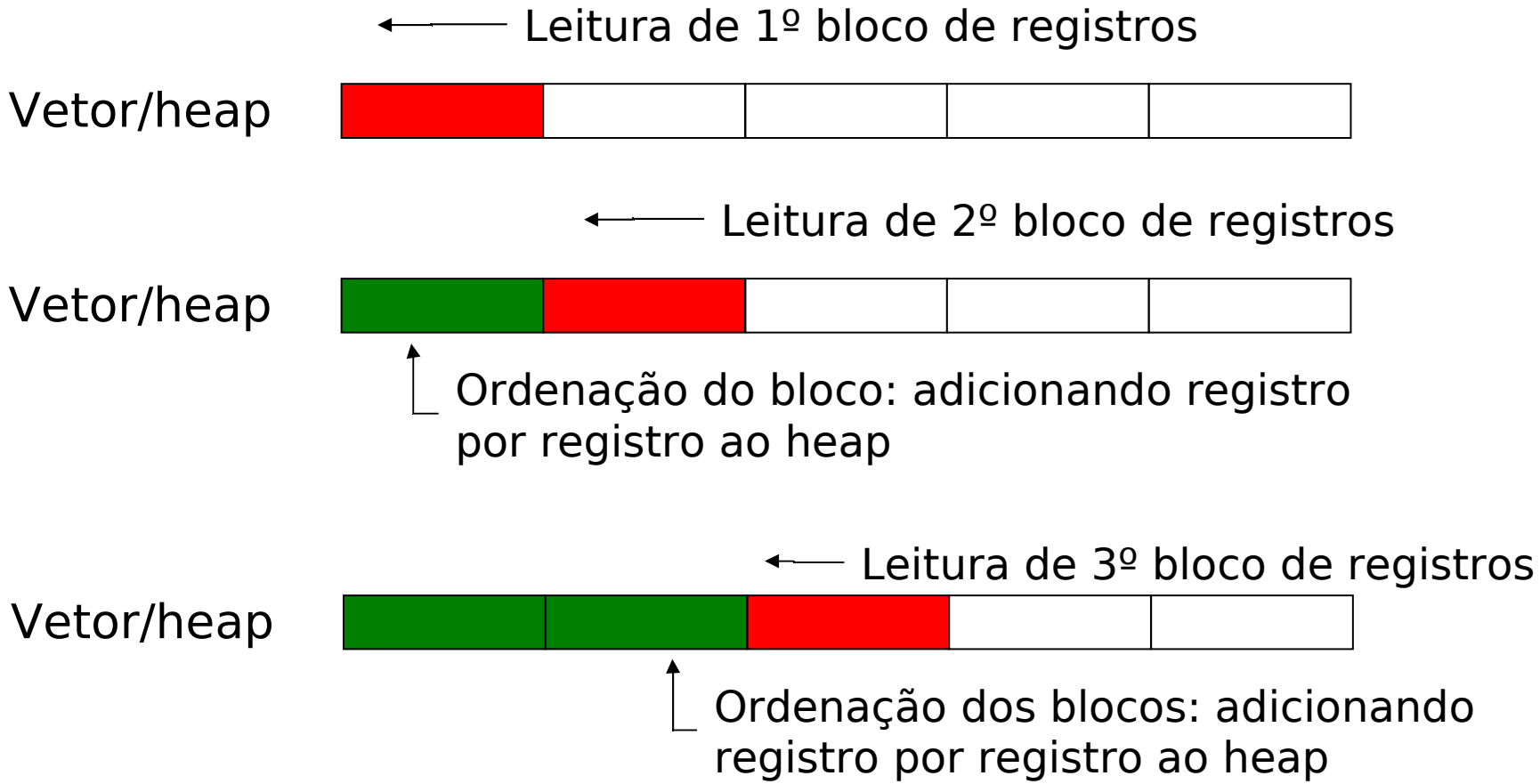
1	2	3	4	5	6	7	8	9
A	B	C	E	H	I	D	G	F

Processamento e entrada/saída: paralelizando



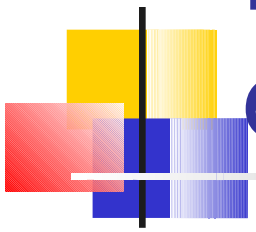
- Percebe-se que
  - O heap se rearranja conforme novos elementos são inseridos
  - Portanto, não é preciso ter todos os registros para se iniciar a ordenação
    - Enquanto ordenação é feita, novos blocos de registros podem ser lidos e adicionados ao final do vetor, para serem absorvidos na seqüência

Processamento e entrada/saída: paralelizando



...

Processamento e entrada/saída: paralelizando



- Após ordenação, seria ótimo poder gravar registros enquanto heap se rearranja
  - Heapsort permite isso!
    - Recupera registro da raiz do heap
    - Enquanto rearranja heap, grava esse registro no arquivo de saída

Processamento e entrada/saída: paralelizando

- Rearranjo do heap

1. Retira-se o elemento da raiz

1. Coloca-se último elemento K do vetor na raiz, “decrementando” tamanho do vetor

1. Enquanto K maior do que seus filhos, troca-o de lugar com seu menor filho

Processamento e entrada/saída: paralelizando

- Exemplo

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

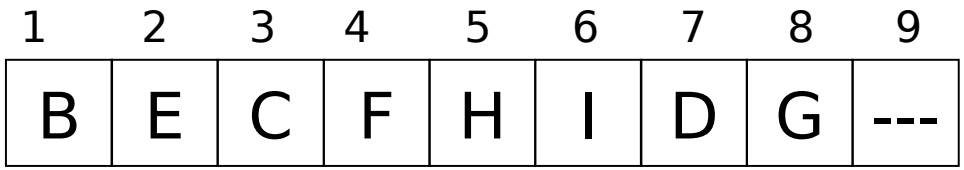
Recupera-se raiz A, colocando em seu lugar F

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

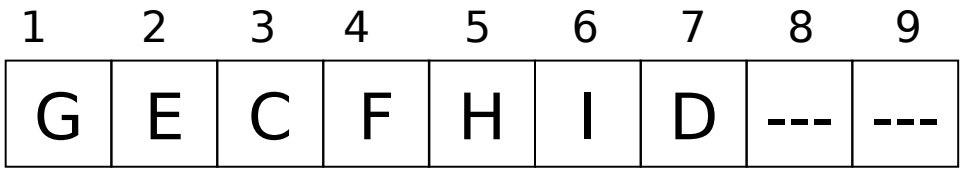
Enquanto grava A no arquivo ordenado, rearranja heap

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

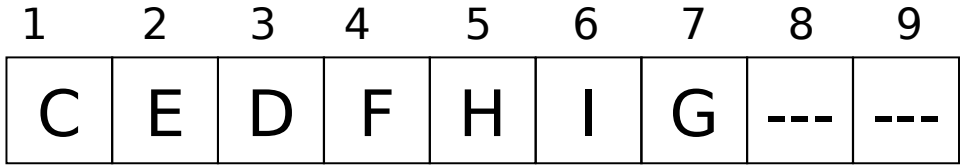
Processamento e entrada/saída: paralelizando



Recupera-se raiz B, colocando em seu lugar G

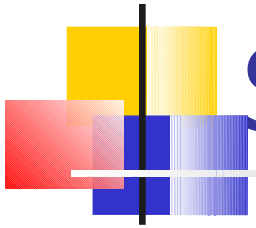


Enquanto grava B no arquivo ordenado, rearranja heap



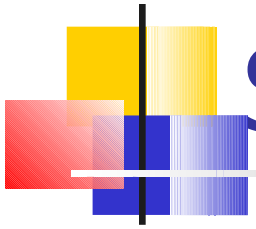
E assim por diante, até heap esvaziar

Ordenação de grandes arquivos



Solução até então

- Se arquivo não cabe na memória
  - Keysorting
    - Muitos seeks
    - Limitado ao número de chaves que poderíamos colocar na memória
      - Não serve para arquivos realmente grandes, de fato
        - Exemplo: arquivo com 8.000.000 registros, cada registro com 100 bytes, cada chave com 10 bytes, precisaríamos de 80 megabytes só para as chaves.



Solução até então

- Se arquivo não cabe na memória

ou se é inviável carregá-lo completamente para a memória principal

  - Solução melhor?

Ordenação com multiway merging

- Algoritmo

1. Lê-se o máximo de registros que cabem em nossa memória disponível
   Supondo 1MB disponível:
   1.000.000 bytes de RAM/100 bytes por registro = 10.000 registros em RAM

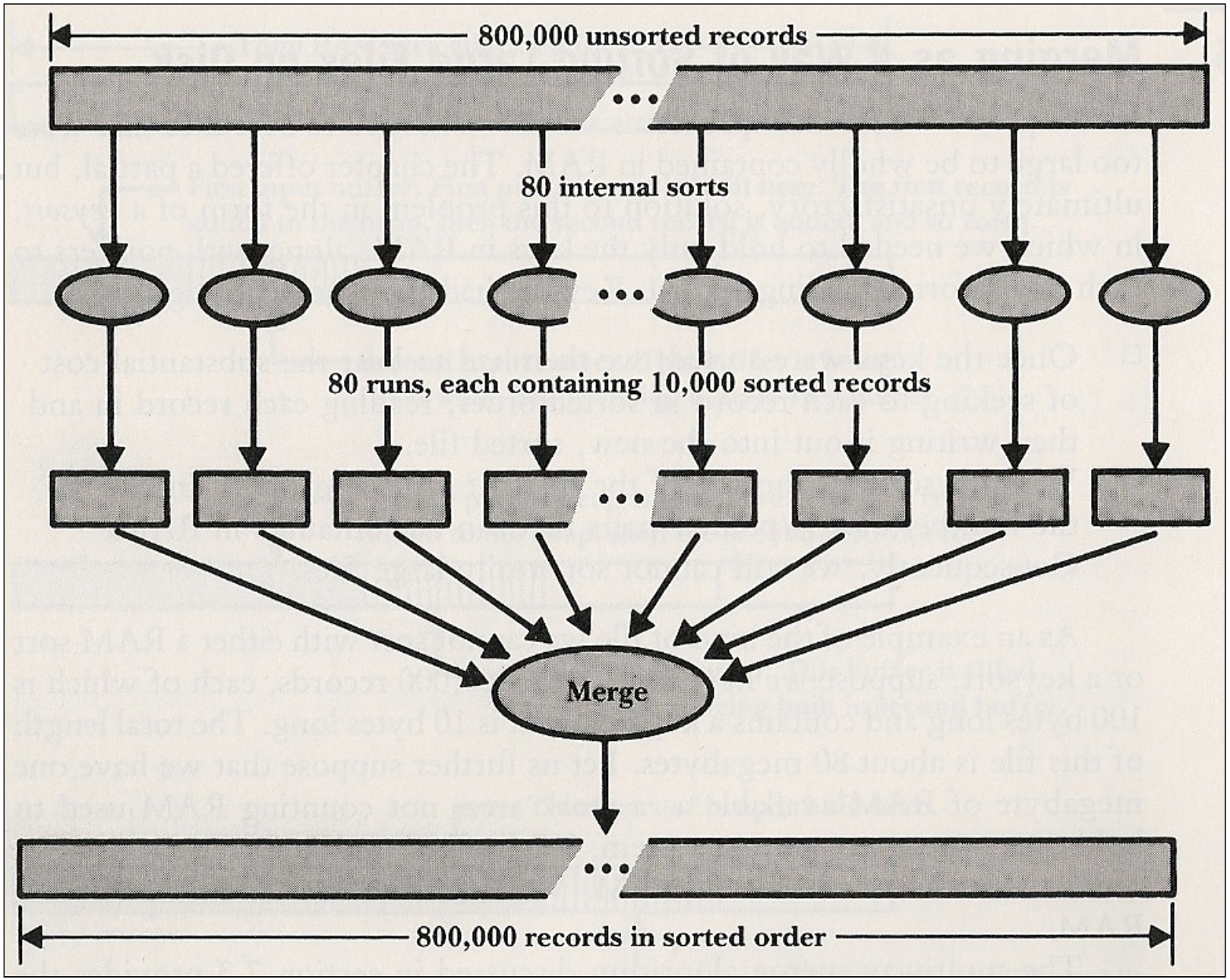
2. Ordena-se os registros na RAM
   Usando nosso heapsort aprimorado

3. Gravam-se os registros ordenados em um novo arquivo, chamado *run*

4. Fazem-se passos 1-3 até que não haja mais registros a serem ordenados
   800.000 registros no arquivo original/10.000 registros em cada rodada = 80 rodadas (novos arquivos ordenados)

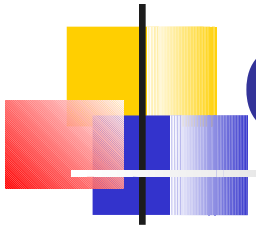
5. Faz-se multiway merging sobre as rodadas criadas

merging


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

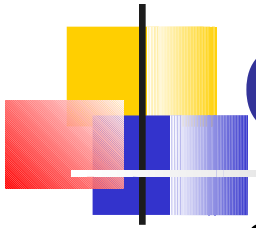
Ordenação com multiway merging

- Esta solução
  - Pode ordenar arquivos realmente grandes
  - Leitura dos registros envolve apenas acesso seqüencial ao arquivo original
  - A escrita final também só envolve acesso seqüencial
  - Aplicável também a arquivos mantidos em fita, já que E/S é seqüencial



Custo

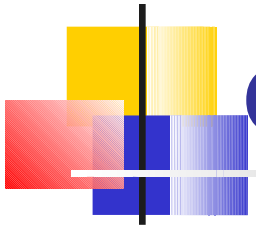
- Quatro passos a serem considerados
  - Leitura dos registros, do disco para a memória, para ordenação
  - Escrita dos registros ordenados para o disco
  - Leitura dos *run files* para merging
  - Escrita do arquivo final em disco



Custo

- Supondo:
  - Arquivo com 8.000.000 registros de 100 bytes (80MB, portanto), e cada rodada com 10 MB
  - 10MB = 100.000 registros
  - Arquivo armazenado em áreas contíguas do disco (*extents*), *extents* alocados em mais de uma trilha, de tal modo que apenas um único *rotational delay* é necessário para cada acesso
  - Características hipotéticas do disco
    - tempo médio para seek: 8 ms
    - rotational delay: 3 ms
    - taxa de transferência: 14.500 bytes/ms
    - tamanho da trilha: 200.000 bytes

Leitura dos registros e criação das rodadas



- Lê-se 10MB de cada vez, para produzir rodadas de 10 MB
- Serão 80 leituras, para formar as 80 rodadas iniciais
- O tempo de leitura de cada rodada 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 rodadas

seek de 8ms + *rot. delay* de 3ms = 11ms por seek

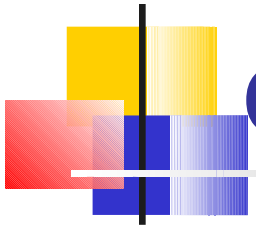
Tempo total: 80*(tempo de **acesso** a uma rodada) + tempo de **transferência** de 800MB

Acesso: 80*(seek + rot.delay = 11ms) = 1s

Transferência: 800 MB a 14.500 bytes/ms = 60s

Total: **61s**

Escrita das rodadas ordenadas no disco

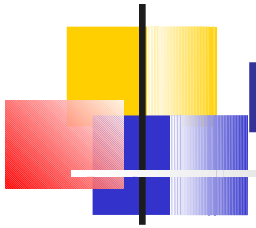


- Idem à leitura!

Serão necessários outros 61s

Leitura das rodadas do disco para a memória (para merging)

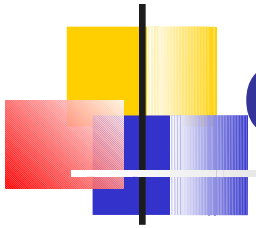
- 10MB de MEMÓRIA para armazenar 80 buffers de entrada
  - portanto, cada buffer armazena 10/80 de uma rodada (125.000 bytes) → cada rodada deve ser acessada 80 vezes para ser lida por completo
- 80 acessos para cada rodada X 80 rodadas
  - 6.400 seeks
- Considerando acesso = seek + rot. delay
  - 11ms X 6.400 = 70s
- Tempo para transferir 800 MB = 60s
- Total = **130s**



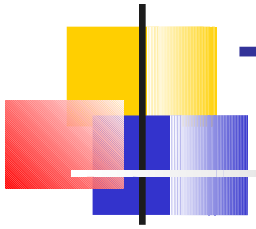
Escrita do arquivo final em disco

- 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 do merging
- Para simplificar, assumimos que é possível alocar 2 *buffers* adicionais de 200.000 bytes para escrita
  - dois para permitir *double buffering*, 200.000 porque é o tamanho da trilha no nosso disco hipotético

Escrita do arquivo final em disco

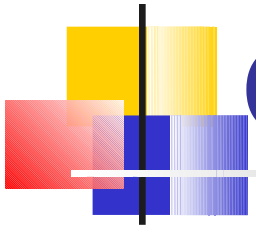


- Com *buffers* de 200.000 bytes, precisaremos de 800.000.000 bytes / 200.000 bytes = 4.000 seeks
- Com tempo de seek+rot.delay = 11ms por seek, 4.000 seeks usam 4.000 X 11, totalizando 44s
- Tempo de transferência é ainda 60s
- Total de **104s**



Tempo total

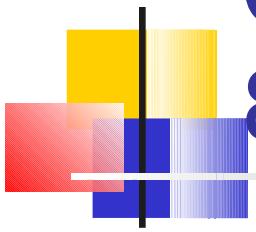
- leitura dos registros para a memória para a criação de rodadas: 61s
- escrita das rodadas ordenadas para o disco: 61s
- leitura das rodadas para merging: 130s
- escrita do arquivo final em disco: 104s
- tempo total da ordenação por multiway merging= 356s, ou **5min56s**



Comparação

- Quanto tempo levaria um método que não usa merging?
  - Por exemplo, *keysorting*
    - Se for necessário **um** *seek* separado para cada registro, i.e, 8.000.000 seeks a 11ms cada, o resultado seria um tempo total (só para *seek*) = 88.000s = **24h26m**

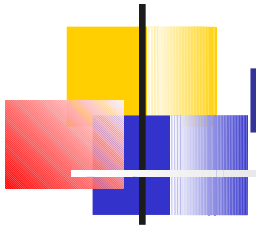
Ordenação de um arquivo com 8.000.000 de registros



- Análise - arquivo de 8.000 MB
  - Qual o efeito de aumentar o arquivo?

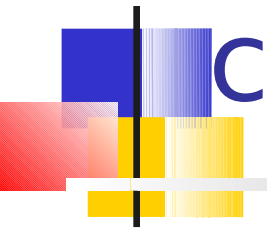
- O arquivo aumenta, mas a memória não!
  - Em vez de 80 rodadas iniciais, teremos 800
  - Portanto, seria necessário merging com 800 rodadas nos mesmos 10 MB de memória, o que implica em que a memória seja dividida em 800 buffers

Ordenação de um arquivo com 8.000.000 de registros



- Cada buffer comporta 10/800 de uma rodada, e cada rodada é acessada 800 vezes
- 800 rodadas X 800 seeks/rodada= 640.000 seeks no total
- O tempo total agora é superior a 2h24m, aproximadamente 25 vezes maior do que o arquivo de 800 MB (que é apenas 10 vezes menor do que este)

Ordenação de um arquivo com 80.000.000 de registros



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

O custo de aumentar o tamanho do arquivo

- A grande diferença de tempo no merging dos dois arquivos (de 800 e 8000 MB) é conseqüência da diferença nos tempos de acesso às rodadas (seek e rotational delay)

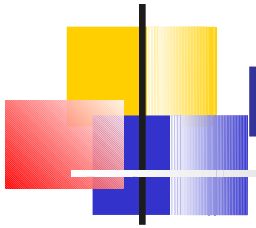
- Em geral, para merging de K rodadas, em que cada rodada é do tamanho da memória disponível, o tamanho dos buffers para cada uma das rodadas é de:

  (1/K) x tamanho da memória = (1/K) x tamanho de cada rodada

  sendo que são necessários K seeks para as rodadas

Complexidade do método

- Como temos K rodadas, a operação de merging requer K^2 seeks
- Medido em termos de seeks, temos $O(K^2)$
- Como K é diretamente proporcional à N, pode-se dizer que o método é $O(N^2)$, em termos de seeks



Maneiras de reduzir esse tempo

1. usar mais (em quantidade) hardware (disk drives, MEMÓRIA, canais de E/S)

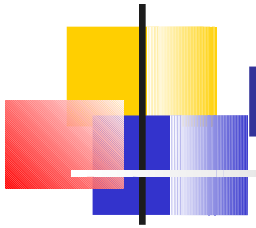
1. realizar o merging em mais de uma etapa, o que reduz a ordem de cada merging e aumenta o tamanho do buffer para cada rodada

1. aumentar o tamanho das rodadas iniciais

1. realizar E/S simultâneo ao merging

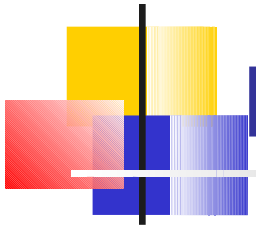
Redução do número de seeks: Merging em Múltiplos Passos

- Em vez de fazer merging de todas as rodadas simultaneamente, o grupo original é dividido em sub-grupos menores
- Merging é feito para cada sub-grupo
- Para cada sub-grupo, um espaço maior é alocado para cada rodada, portanto um número menor de seeks é necessário
- Uma vez completadas todos os merging menores, o segundo passo completa os merging de todas as rodadas



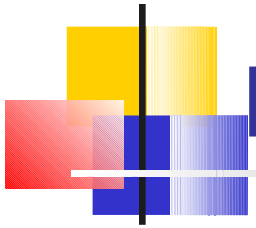
Merging em Múltiplos Passos

- 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 8000 MB tinhamos 800 rodadas com 100.000 registros cada. Para esse arquivo, o merging múltiplo poderia ser realizada em dois passos:
  - primeiro, merging de 25 conjuntos de 32 rodadas cada
  - depois, merging dos 25 conjuntos



Merging em Múltiplos Passos

- Passo único visto anteriormente exige 640.000 seeks
- Merging em 2 passos, temos no **passo 1**:
  - Cada merging de 32 rodadas aloca buffers que podem conter **1/32** de uma rodada. Então, serão realizados 32 X 32 = **1024** seeks
  - Então, 25 vezes o merging exige 25 X 1024 = **25.600** seeks
  - Cada rodada resultante tem 32 X 100.000 = 320.000 registros = 320 MB



Merging em Múltiplos Passos

- No **passo 2**, cada uma das 25 rodadas de 32 MB pode alocar 1/25 do buffer
  - Portanto, cada buffer aloca **4000** registros, ou seja, 1/800 da rodada. Então, esse passo exige 800 seeks por rodada, num total de 25 X 800 = 20.000 seeks

- Total de seeks nos dois passos: 25.600 + 20.000 = **45.600 seeks**

E o tempo **total** de merging?

- Nesse caso, cada registro é transmitido 4 vezes, em vez de duas
  - Assim gastamos mais 1200s em tempo de transmissão
- Ainda, cada registro é escrito duas vezes: mais 40.000 seeks (assumindo 2 buffers de 200.000 bytes cada)
- Somando tudo isso, o tempo total de merging = 3.782s ~ 1h03min
  - O merging tradicional consumia ~2h25m

Aumento do tamanho das rodadas

- Se pudéssemos alocar rodadas com 200.000 registros, ao invés de 100.000 (limite da memória), teríamos merging de 400 rodadas, ao invés de 800
- Neste caso, seriam necessários 800 seeks por rodada, e o número total de seeks seria: 800 seeks/rodada X 400 rodada = 320.000 seeks
  - Isso é metade do número requerido pelo merge de 800 vias com corridas de 100.000 bytes.
- Como aumentar o tamanho das rodadas iniciais sem usar mais memória?
  - sacrificar eficiência em memória primária para diminuir o esforço em disco: ver *replacement selection*.