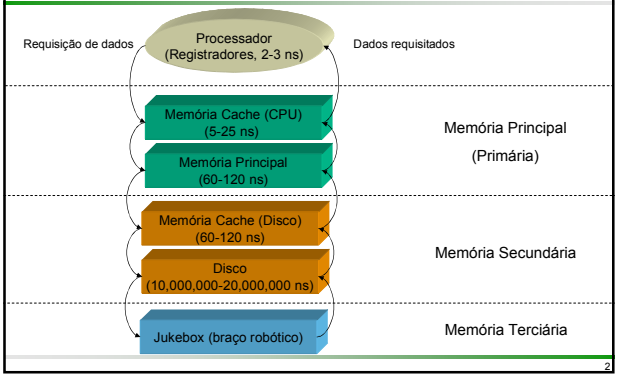
Organização de Arquivos

Algoritmos e
Estruturas de Dados II

- Nesta aula são introduzidos conceitos sobre discos e organização de arquivos e o modelo conseqüencial de processamento de arquivos

José Augusto Baranauskas
Departamento de Física e Matemática – FFCLRP-USP
augusto@ffclrp.usp.br
http://dfm.ffclrp.usp.br/~augusto

Níveis de Armazenamento



Memória Cache

- A memória **cache** corresponde a uma memória de acesso mais rápido do que a memória usual
- Por ser mais rápida seu custo é mais elevado
  - O tamanho da memória **cache** geralmente é bem menor que a memória usual
- Na memória **cache** é colocada uma (pequena) cópia dos dados originais (armazenados na memória usual ou computados anteriormente)
  - Somente os dados mais frequentemente utilizados permanecem no cache
- **Caches** mostram-se extremamente eficientes em muitas áreas da computação devido ao fato que os padrões de acesso em aplicações típicas possuem uma certa localização das referências

Memória Cache

- O funcionamento básico de memórias **cache** pode ser resumido da seguinte forma:
  - Se o dado solicitado encontra-se no **cache**, utilize-o (*cache hit*)
  - Se o dado solicitado não se encontra no **cache**, traga-o para o **cache** e utilize-o (*cache miss*)

Memória Cache

Endereço	Dado	Tag
0	300	2
1	100	0

Memória Principal

Endereço	Dado
0	100
1	200
2	300
3	400

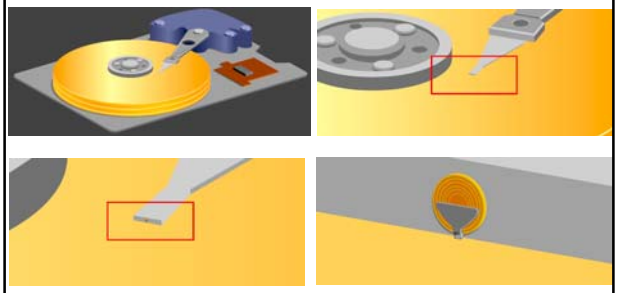
Prefixos Binários

Prefixo	Símbolo	Valor	Base 10
kilo	k/K	2^{10} = 1,024	> 10^3
mega	M	2^{20} = 1,048,576	> 10^6
giga	G	2^{30} = 1,073,741,824	> 10^9
tera	T	2^{40} = 1,099,511,627,776	> 10^{12}
peta	P	2^{50} = 1,125,899,906,842,624	> 10^{15}
exa	E	2^{60} = 1,152,921,504,606,846,976	> 10^{18}
zeta	Z	2^{70} = 1,180,591,620,717,411,303,424	> 10^{21}
yota	Y	2^{80} = 1,208,925,819,614,629,174,706,176	> 10^{24}

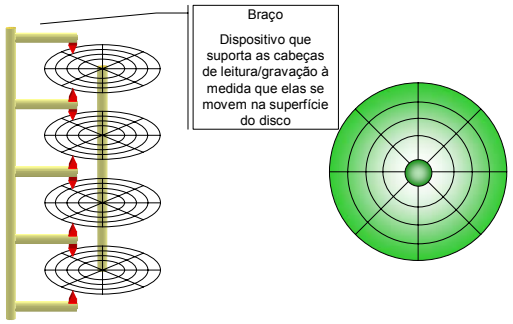
Discos

- Vamos nos concentrar principalmente nos dispositivos de memória secundária de acesso direto, exemplificados pelos discos

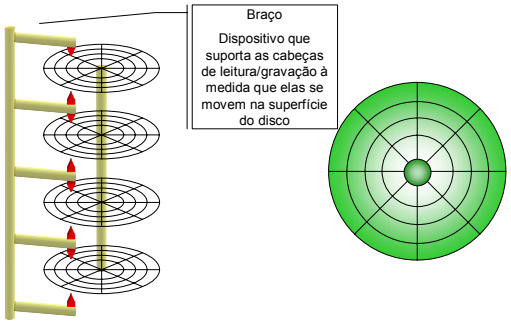
Discos



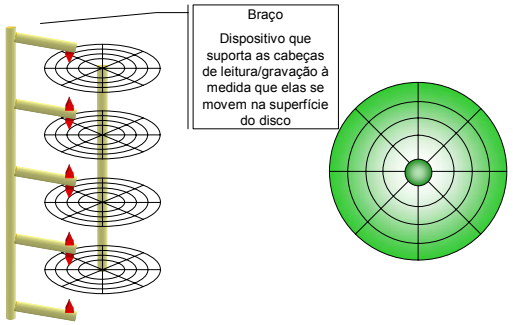
Terminologia sobre Discos



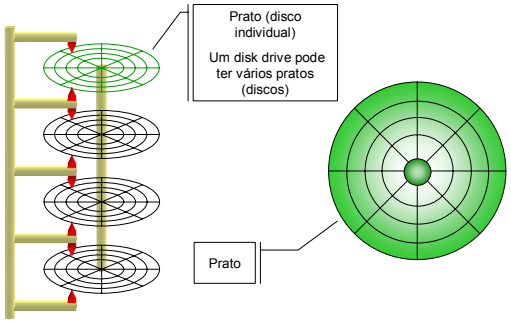
Terminologia sobre Discos



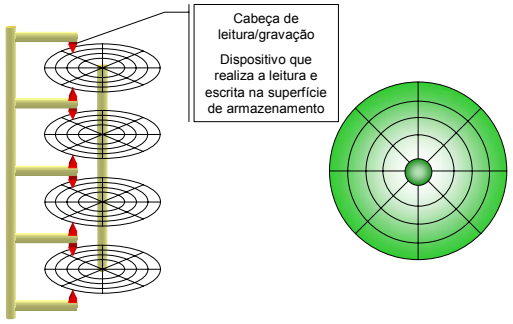
Terminologia sobre Discos



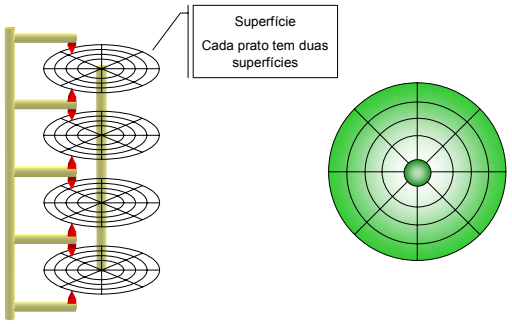
Terminologia sobre Discos



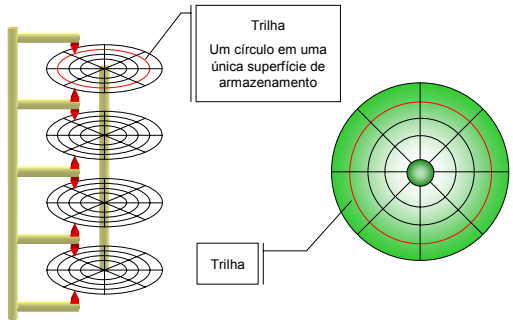
Terminologia sobre Discos



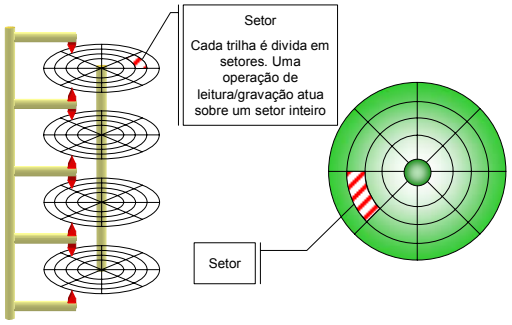
Terminologia sobre Discos



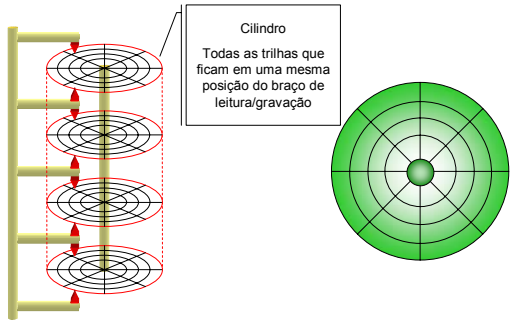
Terminologia sobre Discos



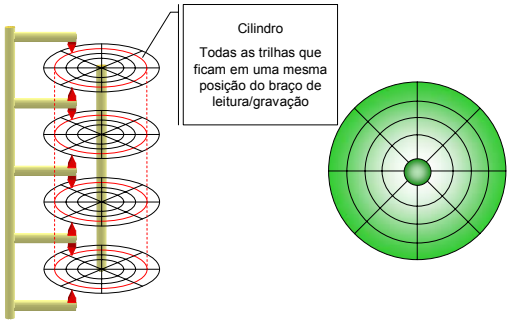
Terminologia sobre Discos



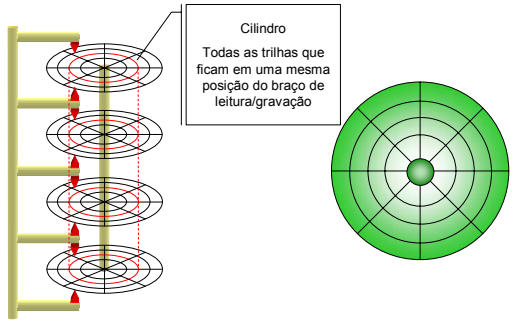
Terminologia sobre Discos



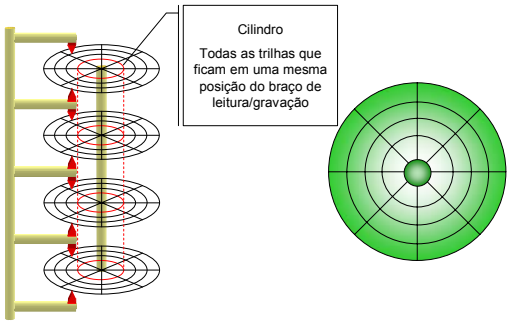
Terminologia sobre Discos



Terminologia sobre Discos



Terminologia sobre Discos



Terminologia sobre Discos

- Um mecanismo impulsor de discos tem um eixo rotor onde são montados os discos e um conjunto de cabeçotes (ou cabeças) de leitura/gravação
- Existe uma cabeça de leitura/gravação para cada superfície
- Durante a leitura ou gravação, os cabeçotes estão estacionários sobre os discos em posição onde a leitura ou gravação deve ser efetuada, enquanto os discos giram em alta velocidade
- Assim, o dispositivo vai ler ou gravar em círculos concêntricos de cada superfície

Terminologia sobre Discos

- A área que pode ser lida ou gravada por uma única cabeça estacionada, é chamada de uma *trilha*
- As trilhas são então círculos concêntricos e cada vez que o disco completa uma rotação, uma trilha completa passa em frente do cabeçote da leitura/gravação
- A coleção de trilhas de todas as superfícies que se encontram simultaneamente sob os cabeçotes de leitura/gravação é chamada de *cilindro*
- As trilhas são divididas em setores
- Um *setor* é o menor segmento endereçável de uma trilha
- As informações são gravadas por blocos, ao longo das trilhas de uma superfície

Terminologia sobre Discos

- Para usar um disco deve-se especificar o número de trilha, ou cilindro, o número do setor no qual começa o bloco, como também a superfície
- Posiciona-se primeiro o conjunto de cabeçotes para o cilindro correto
- Antes de começar a leitura/gravação, espera-se a chegada do setor indicado abaixo do cabeçote de leitura/gravação
- Ao término disso, pode ser feita a transmissão de dados

Tempo de Entrada/Saída

- Assim, existem três fatores que contribuem para o tempo de entrada/saída no caso de discos:
- (i) **Tempo de posicionamento** (*seek time*): tempo para posicionar os cabeçotes de leitura/ gravação para o cilindro (ou trilha) correto; depende da quantidade de cilindros (trilhas) que os cabeçotes precisam se deslocar (movimento mecânico do braço)
- (ii) **Tempo de latência** (*latency time*): tempo decorrido até o que setor correto da trilha chegue abaixo do cabeçote de leitura/gravação (espera rotacional)
- (iii) **Tempo de transmissão***:* tempo para transmitir o bloco de dados de/para o disco

Tempo de Entrada/Saída

- Os tempos máximos de posicionamento em um disco são da ordem de 10 ms
- A rotação típica para discos é 7200 rpm, portanto, o tempo máximo de latência é no máximo 8.3 ms (tempo de uma rotação completa)
- As velocidades de transmissão ficam entre 10^5 bytes/s e 10^6 bytes/s
- A quantidade de bytes que podem ser gravados numa unidade de discos depende de números de superfícies e trilhas por superfície
- Esse valor varia de entre 10^9 bytes para discos pequenos até 10^{12} bytes para grandes unidades de discos

Tempo de Latência (Latency Time)

- Tempo de latência
  - Pior caso, deve-se aguardar uma rotação completa = 60000/RPM ms
  - Caso médio, deve-se aguardar meia rotação = 30000/RPM ms
- Estas fórmulas são derivadas diretamente por regra de três, assumindo A rpm (Rotações por Minuto)
  - A rpm → 1 min → 60 s → 60000 ms
  - 1 rpm → x ms
  - x = 60000/A ms

Tempo de Latência (Latency Time)

Velocidade (rpm)	Pior Caso (ms)	Caso Médio (ms)
5400	11.1	5.6
7200	8.3	4.2
10000	6.0	3.0
12000	5.0	2.5
15000	4.0	2.0

Taxa de Transferência

- Não é possível calcular o tempo de transmissão a partir dos outros fatores do disco
- Entretanto, uma maneira de aproximar esse valor consiste em saber quantos bytes passam sob as cabeças de leitura/gravação em um segundo, ou seja, a taxa de transferência de dados
- Isso depende da densidade dos dados (distância entre os bits em cada setor) como também da velocidade do disco
- A densidade pode ser calculada se o número de setores por trilha for conhecido (SPT), uma vez que há 512 bytes de dados em um setor
- A aproximação consiste em (em Megabits/s):
  - Taxa de Transferência de Dados = (RPM/ 60 * SPT * 512 * 8) / 1000000 Mbits/s
  - Para obter o resultado em megabytes/s basta dividir por 8

Exemplo

- Por exemplo, um disco 7200 rpm
  - Tempo de posicionamento = 10 ms
  - Tempo de latência = 8.3 ms
- Se o disco possui 407 setores por trilha, então sua taxa de transferência é de 200 Mbits/s = 25 Mbytes/s = 25000 bytes/ms
- Assim, o tempo de leitura de um único setor é de 512/25000 = 0.02048 ms
- Para um arquivo com 100 registros, cada registro ocupando 300 bytes, alocados em setores distintos o tempo de leitura é
  - 100 * (10 + 8.3 + 0.02048) = 1832.048 ms ~ 2 s

Sistema de Arquivos

- O Sistema de Arquivos (*file system*) é um software responsável por organizar setores do disco em arquivos e diretórios
- O Sistema de Arquivos gerencia quais setores pertencem a um determinado arquivo bem como quais setores não estão sendo utilizados
- Assim, o Sistema de Arquivos permite trabalhar com arquivos de forma lógica, sem se preocupar com os detalhes do armazenamento físico

Arquivos

- Em nosso contexto, um arquivo é definido como um conjunto de **registros**
- Os registros são formados por unidades de informação denominadas **campos**
- Em geral, há um campo especial denominado **chave**, que identifica univocamente cada registro
- Cada arquivo possui associado um **ponteiro de arquivo** que indica a posição (registro) do arquivo onde será efetuada a próxima leitura/gravação
- Os registros são numerados seqüencialmente a partir da unidade (1, 2, 3, ...)
  - Portanto, um arquivo vazio possui zero registros

Arquivos

- Por exemplo, os registros de um arquivo de funcionários poderiam incluir os seguintes campos:
  - Número de matrícula (chave)
  - Nome
  - Cargo
  - Grau de escolaridade
  - Sexo (M,F)
  - Local
  - Estado civil (S, C)
  - Salário

Arquivos

- No Exemplo
  - Cada linha é um registro (5 registros)
  - Cada coluna é um campo (8 campos)
  - O ponteiro de arquivo (►) encontra-se no 3o. registro

	Registro	E#	Nome	Cargo	Esc.	Sexo	Local	E.C.	Salário
	1	800	Austin	programador	2	F	Campinas	S	10.000
	2	510	William	analista	3	M	Campinhas	C	15.000
►	3	950	Melissa	analista	3	F	Vinhedo	S	12.000
	4	750	Hawkins	programador	2	M	Campinas	S	12.000
	5	620	Newton	programador	2	M	Vinhedo	C	9.000

Especificação

- Operações elementares em arquivos
  - abrir (*open*)
  - fechar (*close*)
  - ler (*read, input*)
  - escrever (*write, output*)
  - testar pelo final-de-arquivo (eof = *end-of-file*)
  - posicionar (*seek*) o arquivo em um determinado registro
  - encontrar a posição atual do ponteiro de arquivo (registro atual ou corrente)

Criação

File::File(string filename, Mode modo)

- *pré-condição:* o arquivo não esteja aberto
- *pós-condição:* o ADT é associado ao arquivo em disco de nome *filename* e o arquivo é aberto no modo especificado
  - enum Mode {read, write, readwrite};
- Em geral, arquivos abertos nos modos **read** ou **readwrite** devem existir em disco; no modo **write** todo conteúdo do arquivo é descartado (se existir) e o arquivo é aberto completamente sem registros (vazio)

Destruição

File::~File();

- *pré-condição:* O arquivo tenha sido criado
- *pós-condição:* O arquivo é fechado, garantindo que toda informação seja efetivamente escrita em disco; após fechado, o arquivo correspondente do sistema operacional permanece residente em disco

Status

bool File::Eof();

- *pré-condição:* Arquivo tenha sido criado
- *pós-condição:* função retorna **true** se a próxima leitura/gravação será efetuada após o final do arquivo, ou seja, se o ponteiro de arquivo encontra-se no final do mesmo; **false** caso contrário

Operações Básicas

void File::Write(FileEntry x);

- *pré-condição:* Arquivo já tenha sido criado nos modos **write** ou **readwrite**
- *pós-condição:* O item **x** é armazenado na posição (registro) onde se encontra o ponteiro de arquivo, sobrescrevendo qualquer informação anterior; o ponteiro de arquivo avança para o próximo registro

Operações Básicas

void File::Write(FileEntry x);

- *pré-condição:* Arquivo já tenha sido criado nos modos **write** ou **readwrite**
- *pós-condição:* O item **x** é armazenado na posição (registro) onde se encontra o ponteiro de arquivo, s
  qualquer informação a
  arquivo avança para o

O tipo **FileEntry** depende da aplicação e pode variar desde um simples caracter ou número até uma **struct** ou **class** com muitos campos

Operações Básicas

void File::Read(FileEntry &x);

- *pré-condição:* Arquivo já tenha sido criado nos modos **read** ou **readwrite**
- *pós-condição:* O registro onde se encontra o ponteiro de arquivo é lido e copiado para a variável **x**; o ponteiro de arquivo avança para o próximo registro

Operações Básicas

void File::Seek(int n);

- *pré-condição:* Arquivo já tenha sido criado
- *pós-condição:* O ponteiro de arquivo é posicionado no registro de número **n**, sendo que qualquer operação de leitura/escrita subseqüente será efetuada no registro **n**

Outras Operações

int File::Size();

- *pré-condição:* Arquivo já tenha sido criado
- *pós-condição:* a função retorna o número de registros no arquivo

Outras Operações

int File::Pos();

- *pré-condição:* Arquivo já tenha sido criado
- *pós-condição:* a função retorna o registro atual (corrente) em que o arquivo se encontra

Pontos Importantes

- Um arquivo pode aberto em três modos elementares:
  - Apenas para leitura (*read* ou *read only*)
  - Apenas para escrita (*write* ou *write only*)
  - Para leitura e escrita (*read-write*)
- O acesso aos registros em um arquivo pode ocorrer de forma
  - Seqüencial (um registro é lido/escrito após o anterior)
  - Aleatória (posiciona-se em um registro para posterior leitura/gravação) ou
  - uma combinação de ambas
- Como regra geral nas linguagens de programação, todo arquivo antes de ser utilizado deve ser **aberto**; ao término das operações com o arquivo ele deve ser **fechado**
- Em C++ o fechamento de arquivo é opcional, já que arquivos são objetos em C++ e seu *finalizador* se encarrega de fechar o arquivo quando o objeto deixa de existir; entretanto é uma boa prática de programação fechar um arquivo quando ele não é mais necessário para economizar recursos do sistema operacional

Organização de Arquivos

- O objetivo da organização de arquivos é proporcionar meios para a recuperação e atualização de registros
- A atualização de um registro pode incluir sua remoção, alteração de alguns dos seus campos ou a inserção de um registro completamente novo
- A recuperação de registros ocorre por meio de **consultas**

Tipos de Consultas

- Existem quatro tipos de consultas em arquivos:
  - Q1: Consulta simples: especifica-se o valor de um único campo
  - Q2: Consulta em intervalo: especifica-se um intervalo de valores para um único campo
  - Q3: Consulta funcional: especifica-se alguns valores determinados dentro do arquivo (exemplo: soma, média, desvio-padrão)
  - Q4: Consulta booleana: uma combinação booleana de Q1-Q3 utilizando os operadores lógicos **and**, **or** e **not**

Tipos de Consultas

- No exemplo, recupere os registros de todos os funcionários com:
  - Q1: Sexo = 'M'
  - Q2: Salário > 9000
  - Q3: Salário > média dos salários de todos os funcionários
  - Q4: (Sexo = 'F' **and** Cargo = 'Programador') **or** (Matricula > 700 **and** Sexo = 'M')

Registro	E#	Nome	Cargo	Esc.	Sexo	Local	E.C.	Salário
1	800	Austin	programador	2	F	Campinas	S	10.000
2	510	William	analista	3	M	Campinhas	C	15.000
3	950	Melissa	analista	3	F	Vinhedo	S	12.000
4	750	Hawkins	programador	2	M	Campinas	S	12.000
5	620	Newton	programador	2	M	Vinhedo	C	9.000

Tipos de Consultas

- No exemplo, recupere os registros de todos os funcionários com:
  - **Q1: Sexo = 'M'**
  - Q2: Salário > 9000
  - Q3: Salário > média dos salários de todos os funcionários
  - Q4: (Sexo = 'F' **and** Cargo = 'Programador') **or** (Matricula > 700 **and** Sexo = 'M')

Registro	E#	Nome	Cargo	Esc.	Sexo	Local	E.C.	Salário
1	800	Austin	programador	2	F	Campinas	S	10.000
2	**510**	**William**	**analista**	**3**	**M**	**Campinhas**	**C**	**15.000**
3	950	Melissa	analista	3	F	Vinhedo	S	12.000
4	**750**	**Hawkins**	**programador**	**2**	**M**	**Campinas**	**S**	**12.000**
5	**620**	**Newton**	**programador**	**2**	**M**	**Vinhedo**	**C**	**9.000**

Tipos de Consultas

- No exemplo, recupere os registros de todos os funcionários com:
  - Q1: Sexo = 'M'
  - **Q2: Salário > 9000**
  - Q3: Salário > média dos salários de todos os funcionários
  - Q4: (Sexo = 'F' **and** Cargo = 'Programador') **or** (Matricula > 700 **and** Sexo = 'M')

Registro	E#	Nome	Cargo	Esc.	Sexo	Local	E.C.	Salário
1	**800**	**Austin**	**programador**	**2**	**F**	**Campinas**	**S**	**10.000**
2	**510**	**William**	**analista**	**3**	**M**	**Campinhas**	**C**	**15.000**
3	**950**	**Melissa**	**analista**	**3**	**F**	**Vinhedo**	**S**	**12.000**
4	**750**	**Hawkins**	**programador**	**2**	**M**	**Campinas**	**S**	**12.000**
5	620	Newton	programador	2	M	Vinhedo	C	9.000

Tipos de Consultas

- No exemplo, recupere os registros de todos os funcionários com:
  - Q1: Sexo = 'M'
  - Q2: Salário > 9000
  - **Q3: Salário > média dos salários de todos os funcionários**
  - Q4: (Sexo = 'F' **and** Cargo = 'Programador') **or** (Matricula > 700 **and** Sexo = 'M')

Registro	E#	Nome	Cargo	Esc.	Sexo	Local	E.C.	Salário
1	800	Austin	programador	2	F	Campinas	S	10.000
2	**510**	**William**	**analista**	**3**	**M**	**Campinhas**	**C**	**15.000**
3	**950**	**Melissa**	**analista**	**3**	**F**	**Vinhedo**	**S**	**12.000**
4	**750**	**Hawkins**	**programador**	**2**	**M**	**Campinas**	**S**	**12.000**
5	620	Newton	programador	2	M	Vinhedo	C	9.000

Tipos de Consultas

- No exemplo, recupere os registros de todos os funcionários com:
  - Q1: Sexo = 'M'
  - Q2: Salário > 9000
  - Q3: Salário > média dos salários de todos os funcionários
  - **Q4: (Sexo = 'F' and Cargo = 'Programador') or (Matricula > 700 and Sexo = 'M')**

Registro	E#	Nome	Cargo	Esc.	Sexo	Local	E.C.	Salário
1	**800**	**Austin**	**programador**	**2**	**F**	**Campinas**	**S**	**10.000**
2	510	William	analista	3	M	Campinhas	C	15.000
3	950	Melissa	analista	3	F	Vinhedo	S	12.000
4	**750**	**Hawkins**	**programador**	**2**	**M**	**Campinas**	**S**	**12.000**
5	620	Newton	programador	2	M	Vinhedo	C	9.000

Organização de Arquivos

- É claro que, para cada tipo de consulta, se o arquivo não estiver organizado adequadamente, é necessário uma ou mais varreduras completas (do início ao final) do arquivo
- Uma primeira idéia seria organizar um arquivo ordenando-o segundo algum campo
- Um método comum de ordenação consiste na ordenação (ou classificação) por intercalação
- A intercalação pressupõe dois arquivos ordenados por algum campo, juntando-os em um único arquivo final, também ordenado
- Em termos gerais a intercalação (*merge*) é o processo de formar uma lista de saída contendo **todos** os itens de duas (ou mais) listas de entrada

Algoritmo de Intercalação

`procedure merge(X,Y,Z)`

- *pré-condição:* assume dois arquivos de entrada X e Y ordenados pelo campo chave
- *pós-condição:* gera como saída o arquivo Z também ordenado por chave composto pelos registros de X e Y
- O procedimento precisa de 3 variáveis do tipo arquivo: duas variáveis de entrada para os arquivos X e Y e uma variável de saída para o arquivo Z
- No algoritmo, assume-se que os registros possuem um campo chave denominado **key**

Algoritmo de Intercalação

```
procedure merge(X,Y,Z)

 Leia registro x de X
 Leia registro y de Y
 while not X.Eof() and
       not Y.Eof() do
   if x.key < y.key then
     Escreva x em Z
     Leia x do arquivo X
   else
     Escreva y em Z
     Leia y do arquivo Y
   endif
 endwhile
```

```
   while not X.Eof() do
     Escreva x em Z
     Leia x do arquivo X
   endwhile

   while not Y.Eof() do
     Escreva y em Z
     Leia y do arquivo Y
   endwhile

end merge
```


Algoritmo de Intercalação

```
procedure merge(X,Y,Z)

 X.read(x);
 Y.read(y);
 while not X.Eof() and
       not Y.Eof() do
   if x.key < y.key then
     Z.write(x);
     X.read(x);
   else
     Z.write(y);
     Y.read(y);
   endif
 endwhile
```

```
   while not X.Eof() do
     Z.write(x);
     X.read(x);
   endwhile

   while not Y.Eof() do
     Z.write(y);
     Y.read(y);
   endwhile

end merge
```

Algoritmo de Intercalação (Usando Modelo Cosseqüencial)

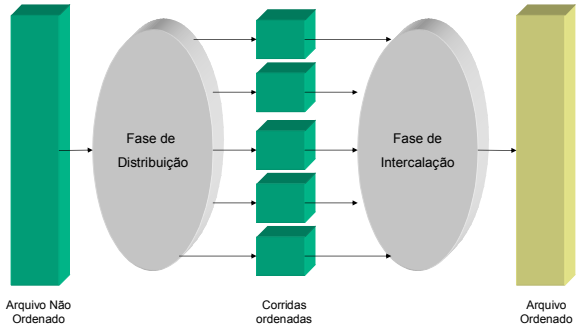
```
procedure merge(X,Y,Z)

 X.read(x);
 Y.read(y);
 acabou ← X.Eof() and Y.Eof();
 while not acabou do
   if x.key < y.key then
     Z.write(x);
     X.read(x);
   else
     Z.write(y);
     Y.read(y);
   endif
   acabou ← X.Eof() and Y.Eof();
 endwhile
```


Classificação por Intercalação

- A Classificação por Intercalação é o método mais comum de ordenação nos dispositivos de memória secundária
- Esse método consiste essencialmente de duas fases distintas
  - Na primeira (distribuição), os segmentos do arquivo de entrada são classificados usando um bom método de ordenação em memória principal
    - Esses segmentos classificados, conhecidos como corridas (*runs)* estão sendo gravados na memória secundária na medida em que são gerados
  - Na segunda (intercalação), as corridas geradas na fase anterior são intercaladas até esgotar as corridas

Classificação por Intercalação

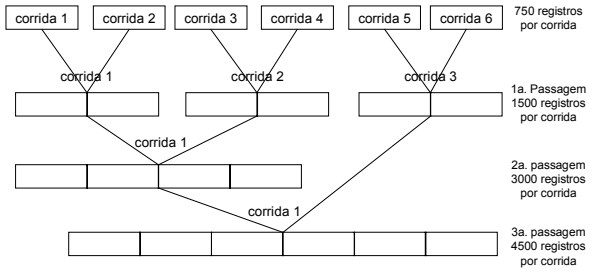


Classificação por Intercalação

- Por exemplo, deve ser classificado um arquivo contendo 4.500 registros A_1,..., A_{4500} em um computador com a memória interna com capacidade de classificar no máximo 750 registros
- Ordenando de cada vez 750 registros, são necessárias seis corridas R_1, R_2, …, R_6

corrida 1	corrida 2	corrida 3	corrida 4	corrida 5	corrida 6
1-750	751-1500	1501-2250	2251-3000	3001-3750	3751-4500

Classificação por Intercalação



Exemplo

- Considere um arquivo com as seguintes chaves

F	45	56	12	43	95	19	8	67

- que é particionado em duas corridas

R_1	45	56	12	43

R_2	95	19	8	67

- Que são ordenadas em memória principal

R_1	12	43	45	56

R_2	8	19	67	95

Exemplo

- As corridas ordenadas

R_1	12	43	45	56
R_2	8	19	67	95

- são intercaladas...

F'								

Exemplo

- As corridas ordenadas

R_1	12	43	45	56
R_2	8	19	67	95

- são intercaladas...

F'	8							

Exemplo

- As corridas ordenadas

R_1	12	43	45	56
R_2	8	19	67	95

- são intercaladas...

F'	8	12						

Exemplo

- As corridas ordenadas

R_1	12	43	45	56
R_2	8	19	67	95

- são intercaladas...

F'	8	12	19					

Exemplo

- As corridas ordenadas

R_1	12	43	45	56
R_2	8	19	67	95

- são intercaladas...

F'	8	12	19	43				

Exemplo

- As corridas ordenadas

R_1	12	43	45	56
R_2	8	19	67	95

- são intercaladas...

F'	8	12	19	43	45			

Exemplo

- As corridas ordenadas

R_1	12	43	45	56
R_2	8	19	67	95

- são intercaladas...

F'	8	12	19	43	45	56		

Exemplo

- As corridas ordenadas

R_1	12	43	45	56
R_2	8	19	67	95

- são intercaladas...

F'	8	12	19	43	45	56	67	

Exemplo

- As corridas ordenadas

R_1	12	43	45	56
R_2	8	19	67	95

- são intercaladas formando uma nova corrida

F'	8	12	19	43	45	56	67	95

- Neste caso, essa corrida é a corrida final, resultando no arquivo F ordenado (F')

Modelo Cosseqüencial

- A intercalação é um exemplo de uma operação **cosseqüencial** de processamento de arquivos
- As operações cosseqüenciais envolvem o processamento coordenado de duas ou mais entradas seqüenciais, de modo a produzir uma única saída, operação muito comum em arquivos
- Exemplos de operações cosseqüenciais são a união (*merge*) e a intersecção (*match*) de duas ou mais entradas
  - Por exemplo, a operação de união é a base do processo de ordenação de arquivos muito grandes, cujo índice não cabe em memória principal

Algoritmo Cosseqüencial de Intercalação (Merge)

- Iniciar (abrir arquivos de entrada e de saída)
- Leia o primeiro item de cada lista
- if *todas as listas atingiram fim-de-arquivo* then acabou←true else acabou←false endif
- while not acabou do
  - Compare os itens atuais de cada uma das listas
  - if os itens são iguais then
    - Processe o item
    - Leia o próximo item de cada lista
  - else
  - if o item da lista A é menor do que o item da lista B then
    - Processe o item da lista A
    - Leia o próximo item da lista A
  - else
  - if o item da lista A é maior do que o item da lista B then
    - Processe o item da lista B
    - Leia o próximo item da lista B
  - endif
  - if *todas as listas atingiram fim-de-arquivo* then acabou ← true endif
- endwhile
- Finalizar (fechar arquivos)

Algoritmo Cosseqüencial de Intersecção (Match)

- Iniciar (abrir arquivos de entrada e de saída)
- Leia o primeiro item de cada lista
- if *uma das listas atingiu fim-de-arquivo* then acabou←true else acabou←false endif
- while not acabou do
  - Compare os itens atuais de cada uma das listas
  - if os itens são iguais then
    - Processe o item
    - Leia o próximo item de cada lista
    - if *uma das listas atingiu fim-de-arquivo* then acabou ← true endif
  - else
  - if o item da lista A é menor do que o item da lista B then
    - Leia o próximo item da lista A
    - if *Lista A atingiu fim-de-arquivo* then acabou ← true endif
  - else
  - if o item da lista A é maior do que o item da lista B then
    - Leia o próximo item da lista B
    - if *Lista B atingiu fim-de-arquivo* then acabou ← true endif
  - endif
- endwhile
- Finalizar (fechar arquivos)

Modelo Cosseqüencial

- Dois ou mais arquivos de entrada devem ser processados simultaneamente para produzir um (ou mais) arquivo de saída
- Cada arquivo é ordenado sobre um ou mais campos chave, e todos os arquivos são ordenados do mesmo modo sobre esses campos
- Para ajudar no gerenciamento condições especiais de início e final de arquivo é interessante, embora não obrigatório, estabelecer valores altos (*high-values*, +∞) e baixos (*low-values*, -∞) que são maiores e menores do que qualquer chave, respectivamente
- 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 pois a ordenação física pode ter grande impacto na eficiência do processamento
- 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 são manipulados somente em memória principal já que não se pode alterar um registro diretamente no arquivo em disco

Intercalação em Múltiplas Vias (*Multiway Merging*)

- Uma variação comum do modelo para processos cosseqüenciais é a **intercalação em K-vias** (*K-way Merge)* na qual **K** arquivos de entrada ordenados são intercalados, gerando um único arquivo de saída
- O valor **K** é a ordem da intercalação

Algoritmo Cosseqüencial de Intercalação K-vias

- Iniciar (abrir arquivos de entrada e de saída)
- Leia o primeiro item de cada lista
- if *todas as listas atingiram fim-de-arquivo* then acabou←true else acabou←false endif
- while not acabou do
  - minItem ← encontre menor item dentre as K listas
  - Processe minItem
  - for i ←1 to K do (em incrementos de +1)
    - if *item da lista i é igual a minItem* then
      - Leia o próximo item da lista i
    - endif
  - next i
  - if *todas as listas atingiram fim-de-arquivo* then acabou ← true endif
- endwhile
- Finalizar (fechar arquivos)

Intercalação K-vias

- Para encontrar o menor item
  - Busca linear: encontre o menor dos K itens procurando sequencialmente desde 1 até o K-ésimo item
  - Busca usando *heap*: K itens (cada um de uma das listas) são mantidos em um *heap* (sob a disciplina: raiz = menor valor); mova o menor item para minItem, substituindo o item movido pelo seu vizinho (próximo item na lista da qual o item movido veio) e então refaça o *heap*

Resumo

- Discos correspondem à dispositivos de armazenamento secundário
- O tempo de acesso a disco é muito grande quando comparado ao tempo de acesso a registradores na CPU
- Para alterar um registro em disco é necessário primeiro que ele seja trazido para memória principal; o registro é alterado em memória principal e novamente escrito em disco
- O modelo cosseqüencial de processamento de arquivos define como operar em dois ou mais arquivos de entrada para produzir arquivos de saída