

# Índices Lineares

Organização e Recuperação de Dados  
Profa. Valéria

UEM – CTC – DIN

Slides preparados com base no Cap. 6 do livro FOLK, M.J. & ZOELLICK, B. *File Structures*. 2<sup>nd</sup> Edition, Addison-Wesley Publishing Company, 1992, e nos slides disponibilizados pelo Prof. Pedro de Azevedo Berger (DCC/UnB)

# O que é um índice?

- ☞ Um índice é uma estrutura que visa facilitar/agilizar as buscas por chave em arquivos de dados
- ☞ Todos os índices são baseados no mesmo conceito básico → **chaves ordenadas + referências**
- ☞ Por enquanto, trataremos de **índice lineares**
  - **Arranjos** em que cada elemento é do tipo {chave, referência}
- ☞ **Propriedades dos índices:**
  - Ordenam de forma indireta → proporcionam uma visão ordenada do arquivo de registros sem rearranjá-lo fisicamente
    - Facilita a inserção no arquivo de dados
    - Elimina o problema dos registros “fixos” (*pinned records*) na ordenação
  - Proporcionam múltiplas visões ordenadas de um mesmo arquivo
    - Podemos ter diferentes índices referenciando o mesmo arquivo de dados (p.e., um catálogo de biblioteca com índices por autor, título, tema, etc.)
  - Permitem acesso direto por chave a registros de tamanho fixo e variável

# Exemplo de arquivo de dados

## @ Exemplo: Catálogo de músicas

- Registros de tamanho variável precedidos por um campo de tamanho (2 bytes)
  - Gravadora, ID, Título, Artista, Compositor, Ano
- **Chave primária** = Gravadora + ID
  - Ex.: **LON2312**

32 43RCA|2626|Like a Rolling Stone|Rolling Stones|...

77 53LON|2312|Help|Beatles|...

132 33WAR|23699|Creep|Radiohead|...

167 42ANG|3795|Pink|Aerosmith|...

...

*Byte-offset do  
registro*

Como organizar este arquivo para garantir acesso rápido a um registro qualquer, dada a sua chave primária?

- Estamos assumindo que o cabeçalho utiliza os 32 primeiros bytes do arquivo e que o campo de tamanho dos registros ocupa 2 bytes

# Índices lineares

🕒 **Índice primário** → estrutura em que cada registro tem dois campos:

1) **Chave primária** → gravadora+ID

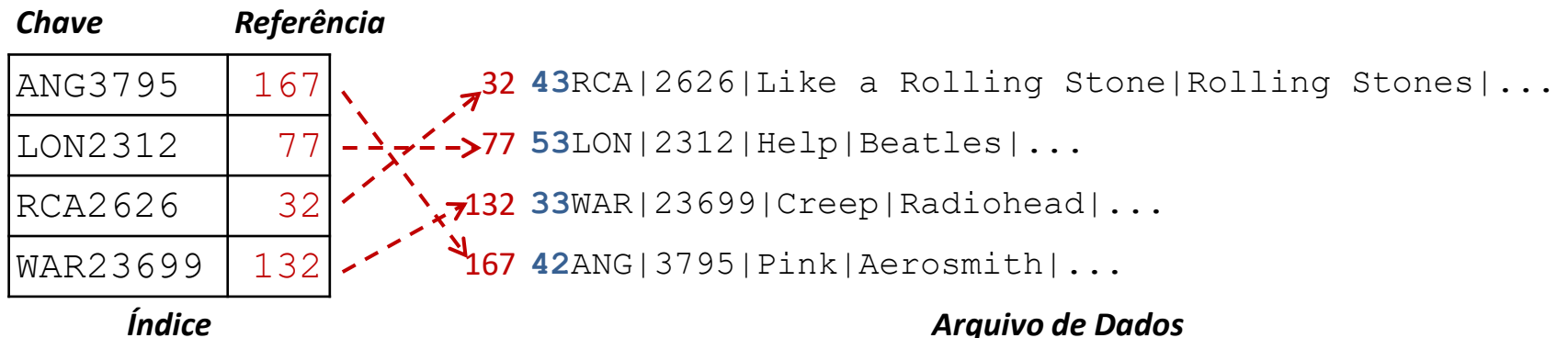
2) **Referência** → *byte-offset* do registro de dados correspondente (RRN se os registros tem tamanho fixo)

– Cada registro do arquivo de dados corresponde a um único registro no índice

- **Relação 1 para 1**

– O índice sempre é **ordenado** pela chave

- O arquivo de dados permanece organizado segundo a ordem de inserção



# Índices lineares

- 🕒 Usar o índice para buscar um registro pela chave é simples
  - O índice permite a **aplicação (indireta) de busca binária** no arquivo de dados, mesmo que ele tenha registros de tamanho variável

```
função busca(CHAVE)
    encontre a posição de CHAVE no índice (usando B.B.)
    recupere o byte-offset correspondente ao registro CHAVE
    posicione o ponteiro de L/E do arquivo de dados sobre o
        registro (usando SEEK e o byte-offset)
    leia o registro do arquivo de dados e retorne-o
fim função
```

# Índices lineares

## @ Considerações

- O índice é mais fácil de trabalhar
  - É muito menor que o arquivo de dados e viabiliza a busca binária se mantido em RAM ou se os registros do índice tiverem tamanho fixo
- A inserção de registros será rápida se o índice for mantido na RAM
  - Índices que não podem ser mantidos na RAM devem ser organizados de outra forma
- Os registros do índice poderiam conter outros campos além do par {CHAVE, BYTEOFFSET}
  - Por ex., o tamanho do registro
  - Mas lembre-se que adicionar informações extras ao índice significa aumentar o seu tamanho físico e, conseqüentemente, a memória necessária para armazená-lo

# Índices lineares

## @ Vantagens

- O arquivo de dados pode ser tratado de forma *sequencial*
  - Novos registros podem ser inseridos no final do arquivo, de acordo com ordem de entrada (ou em espaço disponível para reutilização, se houver uma LED)
  - Facilita a inserção e a manutenção
- Qualquer chave pode ser localizada rapidamente no índice usando busca binária (supondo que o índice está em memória)
- Uma vez localizada a chave no índice, encontra-se o *byte-offset* do registro correspondente e um único acesso é necessário no arquivo de dados
  - Mais rápido do que fazer busca binária em um arquivo ordenado armazenado em disco

# Operações básicas

- ⌚ Vamos assumir que o índice pode ser carregado inteiro do disco para a memória RAM na forma de um arranjo de registros que chamaremos de INDEX

Uma *list* python  
é um arranjo

- ⌚ Operações para manutenção do arquivo de dados e seu índice:

1. Criar os arquivos de índice e de dados
2. Carregar o índice para a memória
3. Regravar o índice em arquivo depois de usá-lo
4. Inserir registros
5. Remover registros
6. Atualizar registros
7. Buscar registro pelo valor de chave



# Operações básicas

## 1. Criar arquivos de índice e de dados

- Os arquivos de dados e de índice são criados como arquivos vazios contendo apenas o cabeçalho
- Esses arquivos serão posteriormente preenchidos com os dados

## 2. Carregar o índice para a memória

- Leia os cabeçalhos dos arquivos de índice e de dados
- Verifique se os cabeçalhos do índice e do arquivo de dados são compatíveis
  - Se o índice não for válido, gere um novo índice
- Leia os registros do arquivo de índice para INDEX
  - O formato do arquivo de índice deve ser definido de modo a facilitar a leitura dos dados para a estrutura INDEX
  - A leitura do arquivo de índice deve ser sequencial

# Operações básicas

## 3. **Regravar** o arquivo de índice depois de usá-lo

- A regravação só é necessária se houver alteração em **INDEX**

## 4. **Inserir** registros

- Novos registros devem ser inseridos no final do arquivo de dados ou em um espaço disponível da **LED**
  - Se um espaço disponível foi reaproveitado, a **LED** deve ser atualizada
- A entrada correspondente {CHAVE, BYTEOFFSET} deve ser inserida em **INDEX** preservando a ordenação
  - A CHAVE deve ser armazenada no índice na forma canônica
  - A inserção da nova entrada pode demandar a reordenação de **INDEX**

# Operações básicas

## 5. Remover registros

- Ao remover (logicamente) um registro do arquivo de dados, a **LED** deve ser atualizada para posterior reutilização do espaço
- A entrada correspondente {CHAVE, BYTEOFFSET} em **INDEX** deve ser removida
  - **INDEX** será reorganizado, mantendo-o ordenado pela CHAVE
    - Também é possível fazer a remoção lógica da entrada do e aguardar uma rotina de manutenção de todo o sistema

# Operações básicas

## 6. Atualizar registros

- O registro será atualizado no arquivo de dados
- A atualização de **INDEX** dependerá de cada caso
  - Se o registro alterado couber no espaço do registro atual, então nenhuma mudança é feita em **INDEX**
  - Se o registro alterado não couber no espaço atual, ele deverá ser gravado em outra posição do arquivo e o seu BYTEOFFSET será atualizado em **INDEX**
  - Se houver alteração da chave primária, **INDEX** deverá ser reordenado

# Índices muito grandes na memória

- ⌚ E se o índice for grande demais para ser mantido em memória?
- ⌚ Nesse caso, é recomendado utilizar estruturas de dados especializadas que requerem um número menor de *seeks*
  - *Árvores-B*
    - Uso de estruturas não lineares para organizar os índices
    - Podem combinar acesso por chave e acesso sequencial de forma eficiente
  - *Hash*
    - Uso de funções de “espalhamento”
    - Para os casos em que a velocidade de acesso é a maior prioridade

# Acesso por múltiplas chaves

Chave	Referência	
ANG3795	167	32 43RCA 2626 Like a Rolling Stone Rolling Stones ...
LON2312	77	77 53LON 2312 Help Beatles ...
RCA2626	32	132 33WAR 23699 Creep Radiohead ...
WAR23699	132	167 42ANG 3795 Pink Aerosmith ...
Índice Primário		Arquivo de Dados

- ⌚ Buscas por chaves primárias são raras
  - “Encontre o registro com a chave WAR23699”
- ⌚ É mais comum que as buscas utilizem **chaves secundárias**
  - “Encontre o registro da música “Creep”
- ⌚ Podemos criar novos índices para o catálogo de músicas, por ex., um índice por título, um por compositor e um por artista
  - Esses índices são chamados de **secundários**
  - Lembre-se que chaves secundárias admitem duplicação

# Índices secundários

- ⌚ Um índice secundário é uma estrutura distinta e, em geral, relaciona uma chave secundária à uma chave primária (e não diretamente ao *offset* do registro de dados)
- ⌚ Podemos criar tantos índices secundários quantas sejam as chaves de busca
- ⌚ Cada índice fornece uma visão diferente do arquivo de dados

# Índices secundários

@ **Índice secundário por artista**: estrutura que relaciona a chave **Artista** com a chave **RÓTULO+ID**

- **RÓTULO+ID** é a chave primária e **Artista** é uma chave secundária (a duplicidade de valores da chave Artista é permitida)

<i>Chave secundária</i>	<i>Chave primária</i>
AEROSMITH	ANG3795
AEROSMITH	DG139201
AEROSMITH	DG18807
AEROSMITH	MER75016
BEATLES	COL38358
BEATLES	LON2312
BOB DYLAN	FF245
RADIOHEAD	WAR23699
ROLLING STONES	ANG1520
ROLLING STONES	COL31809
ROLLING STONES	RCA2629

- ✓ Note que a referência é para a chave primária e não para o *byte-offset* do registro. Isso significa que o índice primário deverá ser consultado para se encontrar o *byte-offset* do registro
- ✓ Existem vantagens em não vincular a chave secundária a um endereço físico
- ✓ Perceba que o índice é ordenado pela chave secundária
- ✓ Perceba que as chaves duplicadas são ordenadas pela chave primária



# Busca em índice secundário

```
função busca_no_secundario(CHAVE)
    encontre a posição de CHAVE no índice secundário
    recupere a CHAVE_PRIMARIA correspondente à CHAVE
    chame busca(CHAVE_PRIMARIA) para obter o registro de dados
    retorne o registro
fim função
```

# Busca por combinação de chaves

- 🕒 O uso de múltiplos índices secundários facilita as buscas por combinação de chaves

- Exemplo

- Encontre todas as gravações com artista “ROLLING STONES” E com título “LIKE A ROLLING STONE”

Artista=“Rolling Stones”	Título=“Like a rolling...”	Resultado (AND)
ANG1520	FF245	RCA2629
COL31809	RCA2629	
RCA2629		

- 🕒 Operações de **interseção** e **união** de listas são facilitadas se as listas estiverem ordenadas → os índices sempre estão ordenados!
- 🕒 Com o uso de índices secundários, esse tipo de busca é simples e rápida, pois as operações lógicas (AND e OR) serão realizadas nos índices que estarão em memória

# Inserir registros (múltiplas chaves)

- ⌚ Na inserção de um novo registro, as respectivas entradas devem ser inseridas em todos os índices
- ⌚ Inserção em índice secundário:
  - O custo é similar ao de inserir um novo registro no índice primário
  - A chave secundária deve ser armazenada na forma canônica
  - Haverá uma duplicação natural de chaves secundárias
    - As chaves duplicadas são agrupadas e devem estar ordenadas pelo campo de referência (i.e., chave primária)
    - A ordenação por chave primária facilitará a implementação das buscas combinadas (interseção e união)

# Remover registros (múltiplas chaves)

- ⌚ Implica na remoção do registro do arquivo de dados e de todos os índices
  - A remoção do registro no arquivo de dados será lógica e o espaço será inserido na LED
- ⌚ A remoção das entradas dos índices pode ser feita de duas formas: (1) ***Excluir-todas-referências*** e (2) ***Excluir-algumas-referências***
- ⌚ (1) ***Excluir-todas-referências***
  - Remove a entrada do índice primário e de todos os índices secundários
  - Implica reorganizar todos os índices
  - **Caso os índices secundários façam referência direta ao *byte-offset* do registro de dados, essa abordagem é obrigatória!**

# Remover registros (Múltiplas chaves)

## ⌚ (2) *Excluir-algumas-referências*

- Faz uso da “indireção” índice secundário → índice primário
  - Remove apenas a referência do índice primário
- A busca por chave secundária continuará funcionando corretamente, pois quando for feita a busca no índice primário, o retorno será “chave inexistente”
- **Mesmo que existam “n” índices secundários, apenas uma remoção em índice será necessária**
- **Vantagem** → economia de tempo (evitando as remoções) quando vários índices secundários estão associados ao índice primário
- **Desvantagem** → desperdício de espaço que é ocupado por registros inválidos, além da queda do desempenho da busca com o passar do tempo
  - Os índices secundários acumularão “falsos candidatos”
  - Uma alternativa para minimizar o problema dos falsos candidatos é fazer “coletas de lixo” periódicas nos índices secundários

# Atualizar registros (Múltiplas chaves)

- ⌚ Se a **chave secundária** é alterada
  - Provavelmente o respectivo índice secundário precisará ser reordenado
- ⌚ Se a **chave primária** é alterada
  - O índice primário deverá ser reordenado e os campos de referência (chave primária) de todos os índices secundários precisam ser atualizados
  - Pode ser necessária uma “reordenação local” nos índices secundários, uma vez que chaves duplicadas devem estar ordenadas pela chave primária
- ⌚ Se um **campo não-chave** é alterado
  - Se o registro for de tamanho variável, pode ser necessário tratar como uma remoção seguida de inserção no arquivo → mudança de *byte-offset*
    - Atualiza o respectivo BYTEOFFSET no índice primário
    - Os índices secundários não precisam ser reordenados!

# Melhorando a estrutura do índice secundário

- ⌚ A estrutura de índice secundário mostrada anteriormente tem dois problemas:
  1. Demanda a reordenação do índice cada vez que um novo registro é inserido no arquivo, mesmo que a chave desse registro já pertença ao índice secundário
  2. A repetição das chaves secundárias resulta em estruturas maiores do que o necessário
    - Faz com que os índices tenham menos chances de caber na memória

Como melhorar a estrutura do índice secundário de forma que ele ocupe menos espaço e exija menos ordenação?

# Listas invertidas

- ⌚ Organizar o índice secundário de forma que cada registro contenha a chave secundária e um ponteiro para uma lista encadeada de chaves primárias
- ⌚ Índices que relacionam uma chave com uma lista de referências são chamados de **listas invertidas**
  - Em vez de buscarmos nos registros pelas ocorrências de uma chave, temos a chave ligada a uma lista de registros nos quais ela ocorre → daí o nome de lista invertida
  - Já que temos um conjunto de chaves primárias associado a cada chave secundária, podemos ligar cada chave secundária a uma **lista de chaves primárias**



# Listas invertidas

- 🌀 As listas invertidas surgiram no contexto da recuperação de informação com o nome de **arquivos invertidos** ou **índices invertidos**
  - São estruturas de dados utilizadas para implementar buscas por palavras-chaves em conjuntos de documentos
  - Em vez de buscarmos todos os documentos pela ocorrência de uma palavra-chave, temos cada palavra-chave ligada a uma lista de documentos nos quais ela ocorre

# Listas invertidas

## Exemplo

### Índice de documentos

d1	●→	fãs	●→	filme	●→	meninas_malvadas	●→	...
d2	●→	fãs	●→	lady_gaga	●→	venon	●→	...
d3	●→	filme	●→	terrorismo	●→	críticas	●→	...
d4	●→	lázaro_ramos	●→	estreia	●→	novo	●→	...
d5	●→	a_revolução_dos_bichos	●→	george_orwell	●→	estreia	●→	...

### Índice Invertido

a_revolução_dos_bichos	●→	d5	
acusados	●→	d2	
críticas	●→	d2	●→ d3
estreia	●→	d2	●→ d3 ●→ d4 ●→ d5
falsas	●→	d2	
fãs	●→	d1	●→ d2
filme	●→	d1	●→ d3 ●→ d4
george_orwell	●→	d5	
globo	●→	d4	
lady_gaga	●→	d2	
lázaro_ramos	●→	d4	
meninas_malvadas	●→	d1	
novo	●→	d4	
outubro	●→	d1	
quadrinhos	●→	d5	
sabotar	●→	d2	
terrorismo	●→	d3	
venom	●→	d2	

### Documentos

**d1.** Entenda por que fãs do filme 'Meninas malvadas' comemoram o 3 de outubro.

**d2.** Fãs de Lady Gaga são acusados de sabotar estreia de 'Venom' com falsas críticas.

**d3.** Filme sobre terrorismo recebe críticas antes mesmo da estreia.

**d4.** Lázaro Ramos estreia novo filme dia 11 na Globo.

**d5.** 'A revolução dos bichos', de George Orwell, estreia em quadrinhos.

# Listas invertidas

## @ Índice secundário com lista invertida: uso de duas estruturas

1. **Índice** → Entradas com dois campos:

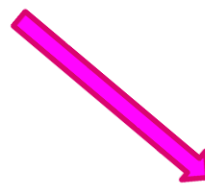
- **Chave**, que armazena a chave secundária (no exemplo, Artista)
- **Posição**, que armazena a posição da 1ª chave primária na lista invertida

2. **Lista invertida** → Entradas com dois campos:

- **Chave**, que armazena a chave primária (no exemplo, ROTULO+ID)
- **Próximo**, que armazena a posição da próxima chave primária na lista invertida

795
9201
807
5016

	<i>Chave secundária</i>	<i>Pos</i>
0	AEROSMITH	3
1	BEATLES	15
2	BOB DYLAN	7
3	...	...



	<i>Chave primária</i>	<i>Prox</i>
0	RCA2629	-1
1	LON2312	12
2	WAR23699	-1
3	ANG3795	5
4	DG18807	6
5	DG139201	4
6	MER75016	-1
7	...	...



# Listas invertidas

@ Diferentes índices secundários podem utilizar uma mesma lista invertida

- Nesse caso, a lista terá um campo *Próximo* para cada índice secundário que a utiliza

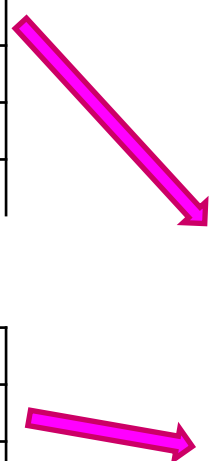
	<i>Artista</i>	<i>Pos</i>
0	AEROSMITH	3
1	BEATLES	15
2	BOB DYLAN	7
3	...	...

	<i>Título</i>	<i>Pos</i>
0	DREAM ON	4
1	LIKE A ROLLING STONE	7
2	YESTERDAY	1
3	...	...

	<i>Chave primária</i>	<i>Prox. Artista</i>	<i>Prox. Título</i>
0	RCA2629	-1	-1
1	LON2312	12	9
2	WAR23699	-1	-1
3	ANG3795	5	-1
4	DG18807	6	16
5	DG139201	4	-1
6	MER75016	-1	-1
7	FF245	18	0
8	...	...	...



# Listas invertidas

## @ Vantagens

- O índice secundário só será reordenado quando uma nova chave for inserida ou quando uma chave existente for alterada
- A remoção ou inserção de registros com chaves existentes implicam apenas em mudanças na lista invertida
  - Ocasionalmente, pode ocorrer alteração do campo *Posição* do índice secundário
  - Para remover todas as ocorrências de uma chave secundária, basta atribuir NULO (-1) ao campo de referência no índice secundário
- O índice secundário será menor e mais fácil de ser manipulado
- A lista invertida não é ordenada, o que facilita a sua manutenção
  - Podemos facilmente reutilizar os espaços dos registros removidos na lista invertida
  - Podemos inserir novas chaves sempre no final da lista invertida, uma vez que ela não é ordenada por chave

# Listas invertidas

## @ Desvantagem

- Perda da localidade:
  - As chaves primárias associadas com uma chave secundária específica normalmente não estão agrupadas fisicamente na lista invertida
    - Recuperar a lista de chaves primárias associadas a uma chave secundária pode envolver várias leituras na lista invertida
    - Se a lista invertida estiver em memória juntamente com os índices, esse custo não será alto

# Binding

- 🕒 Chamamos de ligação ou “*binding*” a associação feita entre uma **chave** e o **endereço físico do registro** ao qual ela se refere
- 🕒 Em que momento essa ligação ocorre?
  - **No índice primário, a ligação sempre se dá no momento da criação do índice**
  - **Nos índices secundários esse momento pode variar**
    - A ligação pode ocorrer no momento da criação do índice
      - Quando o índice secundário “aponta” para o arquivo de dados
    - A ligação pode ocorrer no momento em que a chave for de fato utilizada, ou seja, no momento de uma busca
      - Quando o índice secundário “aponta” para o índice primário

# Binding

@ Os dois momentos da criação das ligações:

- **Early-binding** (ligação precoce) → associa a chave ao endereço físico do registro de dados no momento da criação do índice

Índice Secundário		Arquivo de Dados	
Chave Secundária	Byte-offset	Registro	
...	...	...	
COLDPLAY	167	...	
...	...	167	ANG   3795   SHIVER   COLDPLAY...
		...	

- **Late-binding** (ligação tardia) → associa a chave secundária a uma chave primária, que por sua vez se associa a um endereço físico

Índice Secundário		Índice Primário		Arquivo de Dados	
Chave Secundária	Chave Primária	Chave Primária	Byte-offset	Registro	
...	...	ANG3795	167	...	
COLDPLAY	ANG3795	...	...	...	
...	...			167	ANG   3795   SHIVER   COLD...
				...	



# Ligação tardia (*Late binding*)

@ *Late-binding* → A associação da chave secundária a um endereço físico **ocorre no momento da busca**

## @ **Propriedades**

- O acesso ao registro se dá de forma indireta: chave secundária → chave primária → *byte-offset*
- Busca mais lenta nos índices secundários devido à indireção
- **Baixo custo nas alterações**
  - Uso da abordagem *Excluir-algumas-referências*
  - **Mais seguro:** modificações de *byte-offset* dos registros afeta apenas o índice primário
    - Maior prevenção contra erros de codificação (esquecimento de atualizar um dos  $n$  índices secundários) e falhas do sistema (por ex., queda de energia no momento das atualizações)
  - **Menos problemas de inconsistência:** se o índice primário estiver consistente, os índices secundários também estarão

# Ligação tardia (*Late binding*)

- ⌚ Principal motivação para o uso do *late binding*
  - “É desejável que modificações importantes sejam feitas em um único arquivo do que em vários arquivos”

# Ligação precoce (*Early binding*)

@ **Early-binding** → A associação da chave secundária a um endereço físico **ocorre no momento da criação dos índices**

## @ **Propriedades**

- Busca rápida a partir dos índices secundários → o impacto será maior quando os índices estiverem em disco
- **Alto custo nas alterações**
  - Qualquer modificação relativa a *byte-offset* no registro de dados demanda ajustes em todos os índices (primário e secundários)
  - Uso obrigatório da abordagem **Excluir-todas-referências**
  - Consistência garantida somente após o ajuste de todos os índices
  - **Menos seguro:** modificações referentes à *byte-offset* no arquivo de dados afetam todos os índices
    - Quanto mais índices temos para atualizar, maiores as chances de ocorrer algum problema na atualização (erro de codificação, falha do sistema, etc.)

# Ligação precoce (*Early binding*)

## @ Quando usar *early binding*?

- “O arquivo é estático ou sofre poucas alterações e o desempenho da busca é a prioridade”

# Exercício

- ⌚ O arquivo de dados representado abaixo contém registros de tamanho fixo contendo três campos: **Matrícula**, **Categoria** e **Departamento**
- ⌚ Considerando que o campo **Matrícula** é a chave primária, crie o **índice primário** e dois **índices secundários**: um por **Categoria** e outro por **Departamento**
  - Os dois índices secundários devem usar um **mesmo arquivo de lista invertida**

RRN	Matrícula	Categoria	Departamento
0	2000	Analista	Engenharia
1	1040	Técnico	Computação
2	980	Docente	Arquitetura
3	1900	Secretário	Computação
4	2050	Docente	Computação
5	1010	Analista	Computação
6	1550	Docente	Engenharia
7	430	Secretário	Engenharia
8	2200	Técnico	Computação
9	1990	Secretário	Arquitetura
10	1790	Analista	Arquitetura
11	1100	Analista	Engenharia
12	730	Secretário	Computação
13	1620	Técnico	Computação
14	790	Docente	Engenharia
15	690	Docente	Arquitetura