# Bases de Dados

# Parte IX
# Organização Física dos Dados



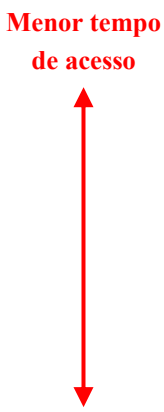


## Unidades de Medida da Informação

- A unidade fundamental é o byte.
- 1 byte corresponde a 8 bits e cada bit permite representar ou um 0 ou um 1.
- 1 Kilobyte (Kbyte ou KB) = $2^{10}$ bytes = 1024 bytes
- 1 Megabyte (Mbyte ou MB) = $2^{20}$ bytes = 1024 Kbytes
- 1 Gigabyte (Gbyte ou GB) = $2^{30}$ bytes = 1024 Mbytes
- 1 Terabyte (Tbyte ou TB) = $2^{40}$ bytes = 1024 Gbytes
- 1 Petabyte (Pbyte ou PB) = $2^{50}$ bytes = 1024 Tbytes

## Dispositivos de Armazenamento da Informação

- **Memória cache**: memória utilizada para acelerar a execução de programas; é o tipo de memória com o menor tempo de acesso.
- **Memória principal ou RAM**: área de trabalho principal do CPU; dispositivo volátil; menor custo que memória cache mas mais lenta.
- **Memória flash**: memória não-volátil de leitura rápida; utilizada em dispositivos de vídeo, fotografia, MP3 e acessórios USB; maior custo e mais lenta na escrita que memória RAM.
- **Disco**: dispositivo de menor custo e de maior capacidade que a memória.
- **DVD**: disco óptico de escrita única e de limitada capacidade (4.5 a 15 Gbytes por disco).
- **CD-ROM**: menos recente que o DVD e de menor capacidade (640 a 870 Mbytes por disco).
- **Fita magnética (*tape*)**: utilizado essencialmente para cópias de segurança (*backups*); é o dispositivo mais barato de armazenamento.





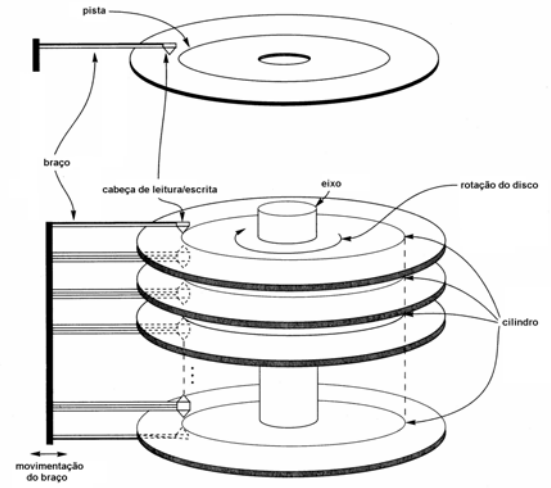



## Armazenamento da Informação em BDs

- Onde guardar os dados que constituem uma BD?
  - Os dados são tipicamente armazenados em disco.
- Usam-se discos para as bases de dados porque as bases de dados possuem grandes volumes de informação que têm de existir durante grandes períodos de tempo:
  - Em geral, uma BD é demasiado grande para caber totalmente em memória.
  - Menor risco de perdas de informação pois os discos são não-voláteis.
  - Menor custo.
- O acesso a uma parte da base de dados envolve:
  - Localizar em disco a parte da base de dados que se pretende aceder.
  - Copiar essa parte para a memória principal para processamento.
  - Em caso de alteração, reescrever essa parte no disco.
- Para obter boa performance, é importante conhecer técnicas de armazenamento da informação em disco.

# Discos

- Os discos são habitualmente constituídos por várias **placas** circulares empilhadas com cabeças de leitura/escrita independentes.
- Cada placa contém um conjunto de **pistas** circulares e concêntricas (capacidade até 150 Kbytes).
- O conjunto das várias pistas com o mesmo diâmetro nas várias placas constitui um **cilindro**.
- A informação guardada num mesmo cilindro pode ser acedida mais rapidamente.

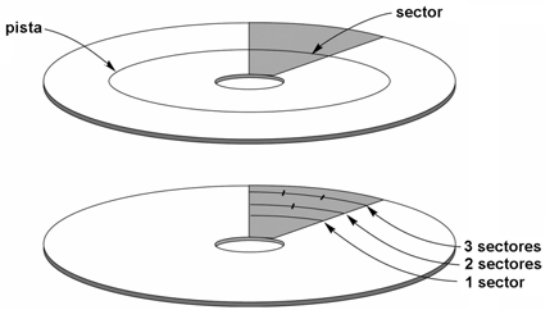







# Discos

- Cada pista é normalmente dividida em **sectores** e **blocos** de tamanho fixo.
- A divisão em sectores é feita pelo fabricante e não pode ser alterada. O tamanho típico dos sectores é de 512 bytes.
- A divisão em blocos (ou páginas) é feita pelo sistema operativo quando o disco é formatado. O tamanho típico dos blocos varia entre 512 bytes e 4 Kbytes.



# Discos

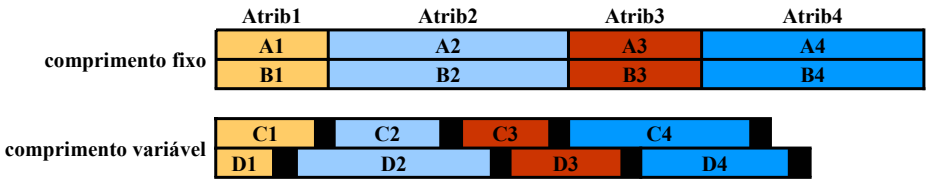
- A transferência de dados entre a memória e o disco é feita em unidades de blocos.
- A leitura de um bloco do disco está condicionada por vários factores:
  - Tempo de posicionamento da cabeça de leitura na pista correcta (*seek time*).
  - Tempo de rotação do disco até que o bloco pretendido fique em posição (*latency time*).
  - Tempo de transferência de um bloco do disco para a memória (*block transfer time*).
- O tempo de posicionamento varia entre 7 a 10 ms para discos comuns.
- O tempo de rotação depende da velocidade de rotação (2 ms para um disco de 15.000 rpm).
  - 15.000 rpm = 250 rps = 4 ms por rotação = 2 ms em média para rodar até o bloco pretendido
- O tempo de transferência depende do tamanho do bloco, do tamanho da pista e da velocidade de rotação (0.4 ms para um disco de 15.000 rpm, com 40 Kbytes por pista e blocos de 4 Kbytes).
  - 40/4 = 10 blocos por pista = 0.4 ms para transferir um bloco





# Registos

- Os dados de uma BD são normalmente guardados em ficheiros de registos.
- Um registo é um conjunto de valores que representam os atributos de um tuplo de uma relação.
- O formato de um registo é definido pelo tipo de dados dos atributos associados à tabela que o registo representa (definido pelo CREATE TABLE).
- Os registos podem ter comprimento fixo ou variável (depende do tipo de dados). Se tiverem comprimento variável é necessário utilizar caracteres especiais para separar os valores do registo.



## Registos

- Dado que a unidade de transferência para o disco é o bloco, para guardamos os dados de uma BD em disco temos de associar registos com blocos em disco.
- Se o tamanho do bloco for B bytes e o tamanho do registo for R bytes (supondo que $B \geq R$) então o número de registos por bloco (*blocking factor* ou *bfr*) é B/R.
- Uma organização que obrigue os registos a pertencerem a um único bloco pode levar a desperdício de espaço nos blocos (pois em geral B não é um múltiplo de R).
- Uma alternativa é permitir que os registos atravessem mais do que um bloco. Para tal, utiliza-se um apontador no final dos blocos para indicar o bloco onde está a restante informação do registo. Esta organização é igualmente utilizada quando o tamanho dos registos excede o tamanho dos blocos ou quando os registos têm comprimento variável.





## Ficheiros de Registos

- Para além da associação entre registos e blocos em disco é também importante conhecer a organização do ficheiro de registos, isto é, como é que os registos são organizados em disco e como são interligados:
  - Ficheiros não-ordenados (*heap files*)
  - Ficheiros ordenados (*sorted files*)
  - Ficheiros de acesso directo (*hash files*)
- O objectivo de uma boa organização é minimizar o número de transferências de blocos do disco para a memória que são necessárias para localizar um determinado registo.

## Operações sobre Ficheiros de Registos

- **Operações de leitura**: não alteram o ficheiro, apenas localizam determinados registos para que os seus valores possam ser lidos.
- **Operações de escrita**: alteram o ficheiro removendo ou adicionando registos ou alterando valores nos registos existentes.
- Estas operações são normalmente implementadas pelos SGBDs por utilização de operações mais básicas que manipulam apenas um registo de cada vez:
  - **FindNext**: procura o registo seguinte que satisfaz a condição de selecção, transfere o bloco que contém esse registo para um *buffer* em memória (se não estiver já lá) e coloca o apontador do registo corrente a apontar para o início do registo.
  - **Delete**: remove o registo corrente e escreve o bloco respectivo para o disco.
  - **Insert**: localiza o bloco onde o novo registo deve ser inserido, transfere esse bloco para memória (*buffer*), escreve o registo no bloco em *buffer* e escreve o bloco para o disco.
  - ...



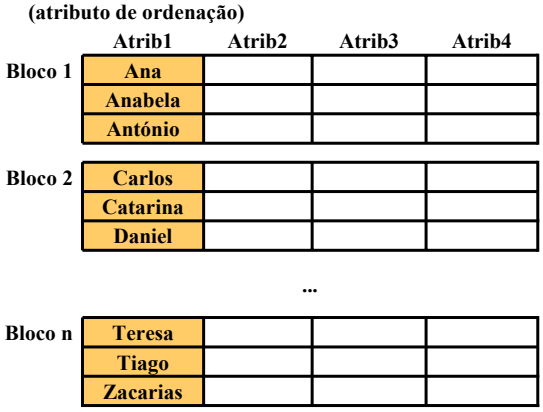


## Ficheiros Não-Ordenados

- Tipo de organização mais simples e básica.
  - Os registos são colocados no ficheiro pela ordem que são inseridos.
  - Novos registos são adicionados no fim do ficheiro.
- Vantagens
  - Inserir um novo registo é muito eficiente: o último bloco do ficheiro é copiado para memória, o novo registo é adicionado e o bloco é reescrito para o disco. O endereço do último bloco do ficheiro é guardado no cabeçalho do ficheiro.
- Desvantagens
  - Procurar um registo é dispendioso: envolve pesquisa linear sobre os blocos em disco.
  - Remover registos leva a desperdício de espaço (blocos com buracos): é necessário reorganizar periodicamente o ficheiro para recuperar espaço.
  - Obter dados ordenados é dispendioso: é necessário criar uma cópia auxiliar do ficheiro com os registos ordenados.

# Ficheiros Ordenados

- Tipo de organização em que os registos estão fisicamente ordenados (nos blocos) com base nos valores de um dos atributos (**atributo de ordenação**).
- Se o atributo de ordenação é um atributo chave então é designado como **chave de ordenação**.





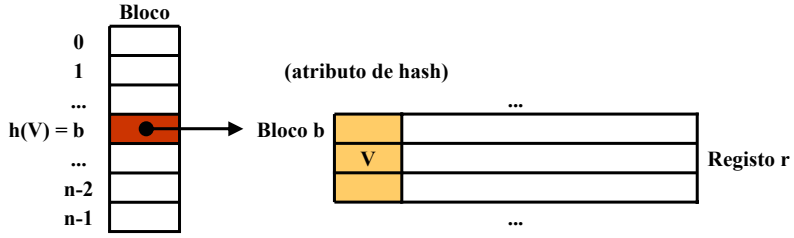



# Ficheiros Ordenados

- Vantagens
  - Aceder aos registos por ordem do atributo de ordenação é muito eficiente: não requer ordenação e o registo seguinte está normalmente no mesmo bloco.
  - Procurar um registo com base num valor do atributo de ordenação é muito eficiente se usarmos pesquisa binária o tempo médio de acesso para B blocos é $\log_2 B$.
- Desvantagens
  - Procurar ou obter dados ordenados com base em outros atributos (diferentes do atributo de ordenação) é igual a ficheiros não-ordenados.
  - Inserir e remover registos são operações dispendiosas: os registos têm de permanecer ordenados após estas operações. A inserção obriga a arranjar espaço para o registo a inserir o que, em média, leva a deslocar metade dos registos para outros blocos para libertar espaço. Podemos melhorar a inserção usando um ficheiro de transacções e fazendo periodicamente a junção desse ficheiro com o ficheiro ordenado (em contrapartida o algoritmo de procura fica mais complexo e menos eficiente).

# Ficheiros de Acesso Directo

- Tipo de organização em que os registos são associados a blocos por utilização de uma **função de *hashing*** sobre os valores de um dos atributos (**atributo de *hash***).
  - O registo r com valor V no atributo de *hash* é armazenado no bloco em disco b, em que b = h(V) e h é a função de *hashing*.
  - O endereço dos blocos em disco é mantido numa tabela de tamanho fixo que é guardada no cabeçalho do ficheiro.



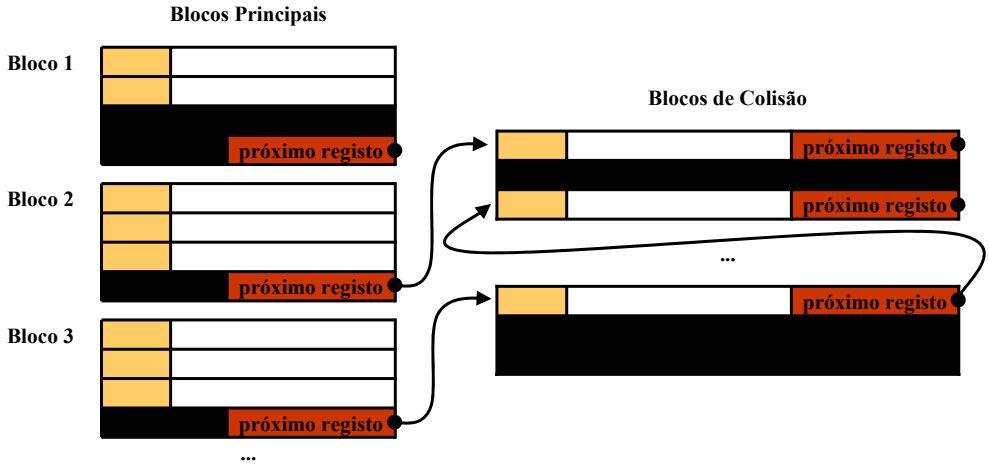





# Ficheiros de Acesso Directo

- Vantagens
  - Procurar um registo com base num valor do atributo de *hash* é muito eficiente: normalmente só precisamos de aceder a um único bloco para obter o registo.
- Desvantagens
  - Procurar registos com base em outros atributos (diferentes do atributo de *hash*) é igual a ficheiros não-ordenados.
  - Problemas de colisão: ocorre uma colisão quando a função de *hashing* associa um novo registo a um bloco em disco que já está cheio. Os registos de colisão são armazenados em blocos (auxiliares) de colisão.
  - O facto do número de blocos ser fixo pode levar a desperdício de espaço (existência de menos registos do que o esperado) ou a um grande número de colisões (existência de mais registos do que o esperado). Quando existe uma grande discrepância entre o número de registos armazenados e esperados podemos alocar um novo número de blocos e utilizar uma nova função de *hashing* para redistribuir os registos existentes. Em alternativa podemos utilizar técnicas de *hashing* com expansão dinâmica.

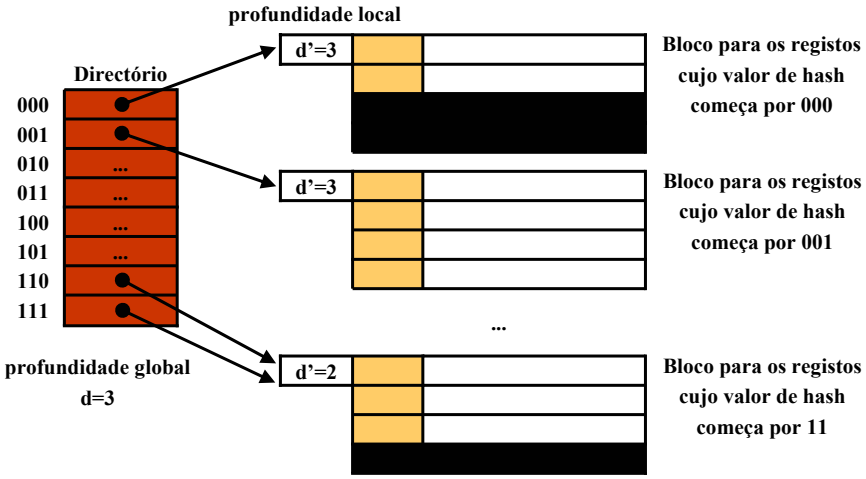
# Ficheiros de Acesso Directo com Blocos de Colisão








# Ficheiros de Acesso Directo com Expansão Dinâmica



# Índices

- **Índices** ou **estruturas de acesso secundárias** são estruturas de dados auxiliares que visam minimizar o tempo de acesso a registos em resposta a operações de procura sobre determinados atributos.
- Os índices são construídos tendo por base os valores de um dos atributos (**atributo de indexação**).
  - Qualquer atributo pode ser utilizado para construir um índice.
  - Diferentes atributos podem ser utilizados para construir múltiplos índices.
- Os índices não alteram a disposição física dos registos nos blocos em disco.





# Índices

- Os índices são guardados em **ficheiros ordenados pelo atributo de indexação**. Cada entrada no ficheiro de índices é composta por:
  - Um valor do atributo de indexação.
  - Um apontador para o bloco que possui o registo com esse valor ou um apontador para o bloco que possui apontadores para os registos com esse valor.
- A ideia é que sendo o ficheiro de índices mais pequeno que o ficheiro de registos, podemos tê-lo em memória e usá-lo para localizar mais depressa os registos nos blocos em disco.
- Existem vários tipos de ficheiros ordenados de índices:
  - Índices primários
  - Índices *cluster*
  - Índices secundários

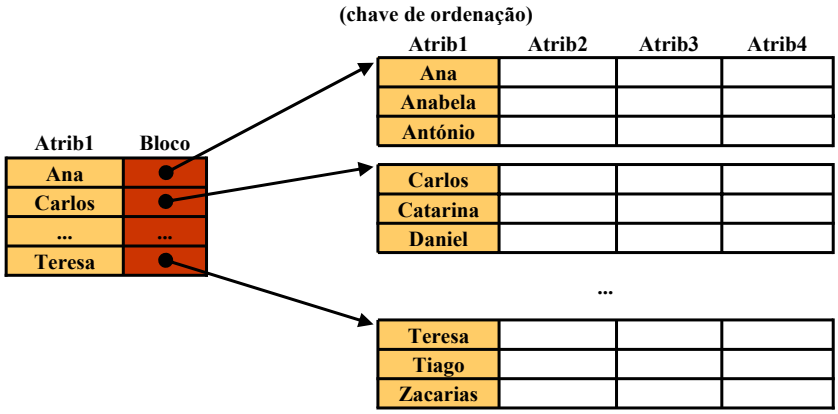
# Índices Primários

- Um índice primário é um índice construído sobre um ficheiro ordenado por um atributo chave (**chave de ordenação**). O atributo de indexação de um índice primário é a **chave de ordenação do ficheiro ordenado**.
- Um índice primário possui uma entrada para cada bloco em disco no ficheiro de registos. Cada entrada num índice primário é composta por:
  - O valor da chave de ordenação do primeiro registo do bloco.
  - O apontador para esse bloco.
- O total de entradas num índice primário é igual ao número de blocos em disco no ficheiro de registos.
- Um índice primário é um **índice esparso**. Um índice diz-se esparso se não tiver uma entrada para cada registo existente. Caso contrário, diz-se denso.
- O maior inconveniente de um índice primário é inserir e remover registos pois obriga a ordenação no ficheiro de registos e no ficheiro de índices.





# Índices Primários



# Índices Primários

- Considere um ficheiro ordenado com **r = 30.000** registos guardado num disco com blocos de **B = 1024** bytes. Os registos têm um tamanho fixo de **R = 100** bytes e não atravessam blocos.
  - O *blocking factor* é **bfr** = $\lfloor B/R \rfloor = \lfloor 1024/100 \rfloor = 10$ registos por bloco.
  - O número de blocos necessários para o ficheiro é **b** = $\lceil r/bfr \rceil = \lceil 30.000/10 \rceil = 3000$ blocos.
  - Pesquisa binária sobre o ficheiro necessita no máximo de $\lceil \log_2 b \rceil = \lceil \log_2 3000 \rceil = 12$ acessos a blocos.
- Considere agora a construção de um índice primário com um atributo de indexação de **V = 9** bytes e um apontador para blocos de **P = 6** bytes.
  - Cada entrada no ficheiro de índices ocupa $R_i = V + P = 15$ bytes.
  - O número de entradas no ficheiro de índices é $r_i = b = 3000$ entradas.
  - O *blocking factor* no ficheiro de índices é $bfr_i = \lfloor B/R_i \rfloor = \lfloor 1024/15 \rfloor = 68$ registos por bloco.
  - O número de blocos necessários para o ficheiro de índices é $b_i = \lceil r_i/bfr_i \rceil = 45$ blocos.
  - Pesquisa binária sobre o ficheiro de índices necessita no máximo de $\lceil \log_2 b_i \rceil = \lceil \log_2 45 \rceil = 6$ acessos a blocos.
- Para encontrar um registo utilizando o ficheiro de índices é necessário no máximo **6 + 1 = 7 acessos a blocos a disco**, ou seja, **menos 5 acessos do que sem o ficheiro de índices**.

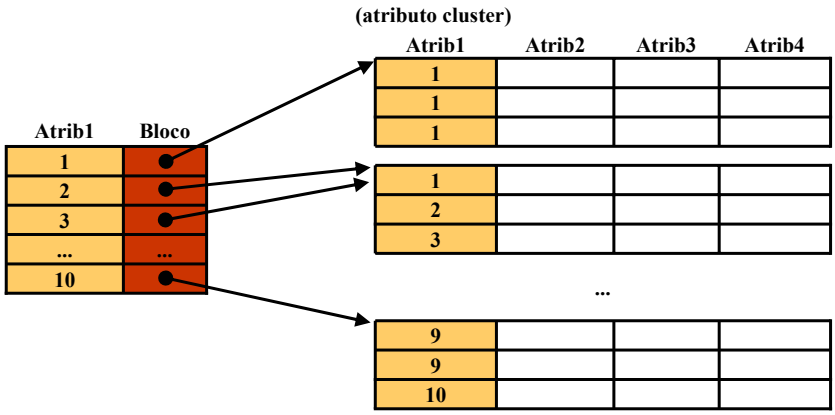




# Índices Cluster

- Um índice *cluster* é um índice construído sobre um ficheiro ordenado por um atributo não chave, ou seja, que pode ter registos com valores iguais para esse atributo (**atributo *cluster***). O atributo de indexação de um índice *cluster* é o **atributo cluster do ficheiro ordenado**.
- Um índice *cluster* possui uma entrada para cada valor distinto do atributo *cluster*. Cada entrada num índice *cluster* é composta por:
  - O valor do atributo *cluster*.
  - O apontador para o primeiro bloco que possui um registo com esse valor.
- Um índice *cluster* é um **índice esparso**.
- O maior inconveniente de um índice *cluster* é igualmente inserir e remover registos pois obriga a ordenação no ficheiro de registos e no ficheiro de índices.

# Índices Cluster





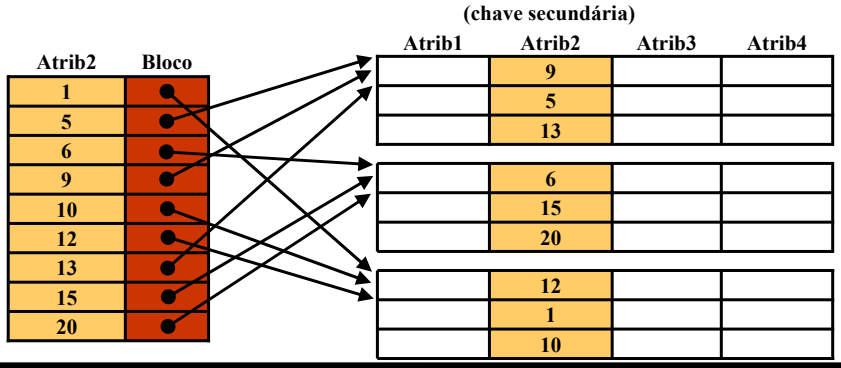



# Índices Secundários

- Um índice secundário é um índice construído sobre um ficheiro para o qual já existe um outro meio de acesso primário. O atributo de indexação de um índice secundário pode ser um **atributo chave candidata (chave secundária)** ou um **atributo não chave do ficheiro**.
- Um índice secundário possui uma entrada para cada valor distinto do atributo de indexação. Cada entrada num índice secundário é composta por:
  - O valor do atributo de indexação.
  - O apontador para o bloco que possui o registo com esse valor ou o apontador para o bloco que possui apontadores para os registos com esse valor.
- É possível ter vários índices secundários para o mesmo ficheiro.
- Normalmente, um índice secundário requer mais espaço e maior tempo de procura do que um índice primário. No entanto, o ganho no tempo de procura conseguido com um índice secundário é incomparavelmente superior ao conseguido com um índice primário.

# Índices Secundários

- Quando o atributo de indexação é uma **chave secundária** então um índice secundário é um **índice denso**. Cada entrada no índice é composta por:
  - O valor da chave secundária.
  - O apontador para o bloco que possui o registo com esse valor.





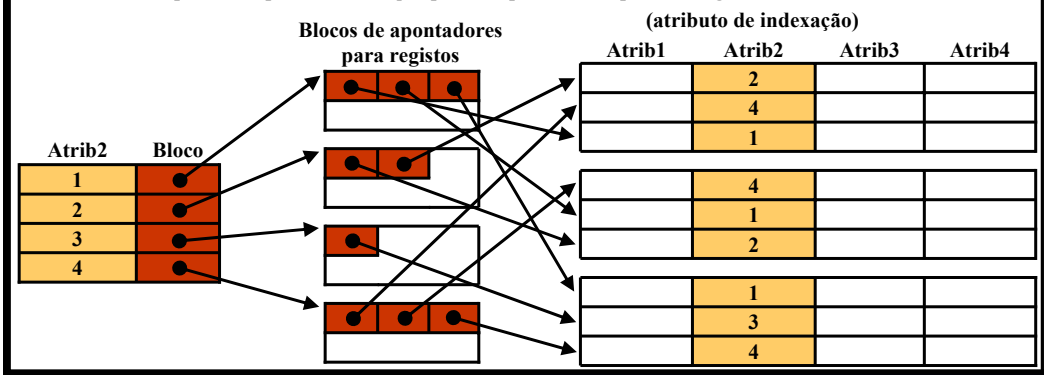



# Índices Secundários

- Considere novamente um ficheiro com **r = 30.000** registos guardado num disco com blocos de **B = 1024** bytes. Os registos têm um tamanho fixo de **R = 100** bytes e não atravessam blocos.
  - O *blocking factor* é **bfr** = $\lfloor B/R \rfloor = \lfloor 1024/100 \rfloor = 10$ registos por bloco.
  - O número de blocos necessários para o ficheiro é **b** = $\lceil r/bfr \rceil = \lceil 30.000/10 \rceil = 3000$ blocos.
  - Pesquisa linear sobre o ficheiro necessita em média de **b/2 = 3000/2 = 1500** acessos a blocos.
- Considere agora a construção de um índice secundário com um atributo de indexação de **V = 9** bytes e um apontador para blocos de **P = 6** bytes.
  - Cada entrada no ficheiro de índices ocupa $R_i = V + P = 15$ bytes.
  - O número de entradas no ficheiro de índices é $r_i = r = 30.000$ entradas.
  - O *blocking factor* no ficheiro de índices é $bfr_i = \lfloor B/R_i \rfloor = \lfloor 1024/15 \rfloor = 68$ registos por bloco.
  - O número de blocos necessários para o ficheiro de índices é $b_i = \lceil r_i/bfr_i \rceil = 442$ blocos.
  - Pesquisa binária sobre o ficheiro de índices necessita no máximo de $\lceil \log_2 b_i \rceil = \lceil \log_2 442 \rceil = 9$ acessos a blocos.
- Para encontrar um registo utilizando o ficheiro de índices é necessário no máximo **9 + 1 = 10 acessos a blocos a disco**, ou seja, **menos 1490 acessos do que sem o ficheiro de índices**.

# Índices Secundários

- Quando o atributo de indexação é um atributo não chave então um índice secundário é um **índice esparso**. Cada entrada no índice é composta por:
  - O valor do atributo de indexação.
  - O apontador para o bloco que possui apontadores para os registos com esse valor.








# Índices

| Tipo de Índice | Atributo de Indexação é | | Número de Entradas | Densidade |
|---|---|---|---|---|
| | Atributo Chave | Atributo de Ordenação | | |
| Primário | Sim | Sim | Blocos do ficheiro | Esparso |
| Cluster | Não | Sim | Valores distintos | Esparso |
| Secundário (chave) | Sim | Não | Registos do ficheiro | Denso |
| Secundário (não chave) | Não | Não | Valores distintos | Esparso |

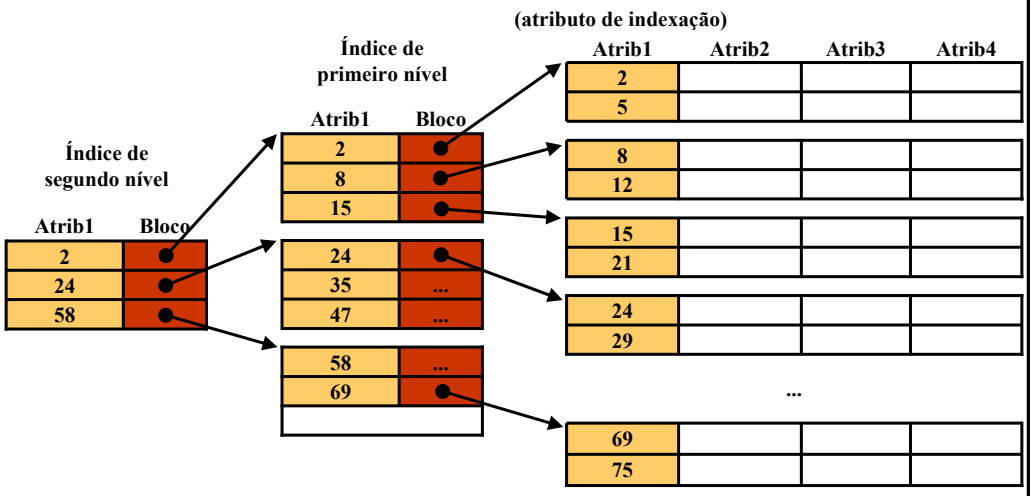
# Índices de Níveis Múltiplos

- Como um ficheiro de índices é um ficheiro ordenado podemos construir um índice primário sobre o próprio ficheiro de índices. O índice original é chamado de **índice de primeiro nível** e o índice sobre o ficheiro de índices é chamado de **índice de segundo nível**.
- Podemos repetir o processo criando índices de terceiro nível, de quarto nível e assim sucessivamente até que as entradas de um nível (**nível de topo**) caibam num só bloco em disco.
  - Se o número de blocos necessários para o primeiro nível é $\mathbf{b_1}$, então o número de blocos necessários para o segundo nível é $\mathbf{b_2} = \lceil \mathbf{b_1}/\mathbf{bfr_i} \rceil$, para o terceiro nível é $\mathbf{b_3} = \lceil \mathbf{b_2}/\mathbf{bfr_i} \rceil$ e assim sucessivamente até o nível de topo t em que $\mathbf{b_t} = \mathbf{1}$.
- Um índice de níveis múltiplos pode ser criado a partir de qualquer tipo de índice de primeiro nível (primário, *cluster* ou secundário).
- Inserir ou remover registos num índice de níveis múltiplos é um problema pois obriga a ordenação em todos os níveis do índice.





# Índices de Níveis Múltiplos



## Índices de Níveis Múltiplos

- Considere novamente o índice secundário anterior com $\mathbf{b_i = 442}$ blocos necessários para o ficheiro de índices e com um *blocking factor* de $\mathbf{bfr_i = 68}$ registos por bloco.
- Considere agora a construção de um índice de níveis múltiplos a partir do índice secundário.
  - O número de blocos do primeiro nível é $\mathbf{b_1 = b_i = 442}$ blocos.
  - O número de blocos necessários para o segundo nível é $\mathbf{b_2 = \lceil b_1/bfr_i \rceil = \lceil 442/68 \rceil = 7}$ blocos.
  - O número de blocos necessários para o terceiro nível é $\mathbf{b_3 = \lceil b_2/bfr_i \rceil = \lceil 7/68 \rceil = 1}$ bloco, ou seja, o terceiro nível é o nível de topo.
- Para encontrar um registo utilizando o índice de níveis múltiplos é necessário aceder a um bloco por cada nível mais o bloco do ficheiro de registos, ou seja, **3 + 1 = 4 acessos a blocos a disco**, o que equivale no máximo a **menos 6 acessos do que com apenas o índice secundário**.



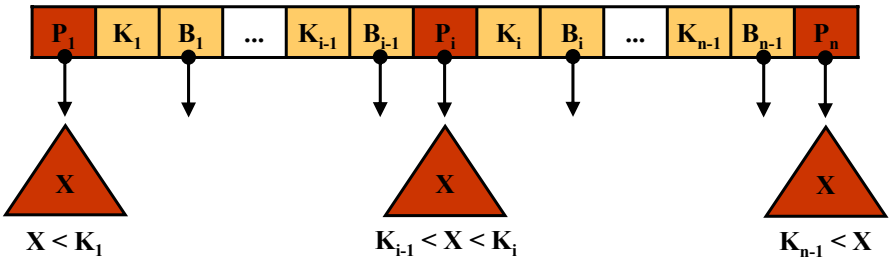


## Árvores B e Árvores B+

- Por forma a simplificar as operações de inserção e remoção, os índices de níveis múltiplos são normalmente implementados como árvores B ou árvores B+.
- A ideia chave das árvores B e B+ é tentar manter algum espaço livre nos nós da árvore de modo a permitir a eficiente inserção e remoção de valores. **Cada nó corresponde a um bloco em disco** e está sempre com a sua capacidade entre 50% a 100% ocupada.
- A inserção de valores num nó não completo é muito eficiente. A inserção num nó completo causa uma separação em dois nós. Esta separação pode propagar-se a outros nós.
- A remoção de valores é muito eficiente se o nó respectivo não ficar com menos do que 50% da sua capacidade. Se isso acontecer, o nó é agrupado a nós vizinhos. Este agrupamento pode propagar-se a outros nós.
- Por experimentação verificou-se que as árvores B e B+ quando estabilizam ficam, **em média, com 69% de cada nó ocupado**.

# Árvores B

- Estrutura dos nós de uma árvore B.





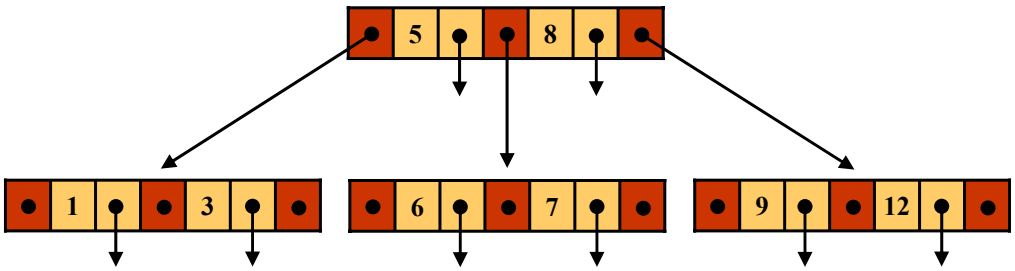



# Árvores B

- Uma árvore B de ordem p é definida como:
  - Cada nó é da forma $<P_1, (K_1, B_1), ..., (K_{i-1}, B_{i-1}), P_i, (K_i, B_i), ..., (K_{n-1}, B_{n-1}), P_n>$, em que $n \leq p$, cada $P_i$ é um apontador para uma sub-árvore B, e cada par $(K_i, B_i)$ é uma entrada em que $B_i$ é o apontador para o bloco em disco que possui o registo com o valor de pesquisa $K_i$.
  - Em cada nó têm-se que $K_1 < ... < K_{i-1} < K_i < ... < K_{n-1}$.
  - Para qualquer valor de pesquisa X na sub-árvore apontada por $P_i$ têm-se que $X < K_i$ para $i = 1$ ou $K_{i-1} < X < K_i$ para $1 < i < n$ ou $K_{i-1} < X$ para $i = n$.
  - Cada nó tem no máximo p apontadores para sub-árvores.
  - Cada nó, excepto o nó raíz e os nós folha, tem pelo menos $\lceil p/2 \rceil$ apontadores para sub-árvores. O nó raíz tem pelo menos 2 apontadores para sub-árvores a menos que seja o único nó da árvore.
  - Um nó com n apontadores para sub-árvores, $n \leq p$, tem $n - 1$ pares de entradas $(K_i, B_i)$.
  - Todos os nós folha ficam no mesmo nível da árvore. Os apontadores para sub-árvores dos nós folha são todos NULL.

# Árvores B

- Árvore B de ordem p = 3.





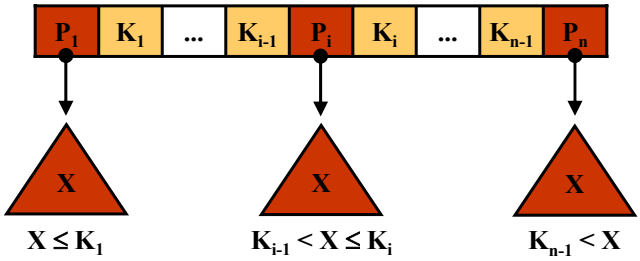



# Árvores B

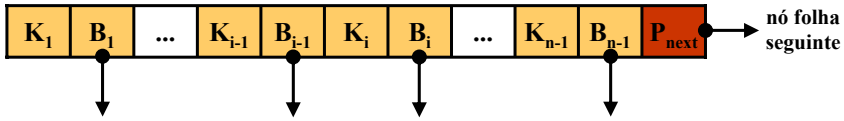
- Considere a construção de uma árvore B com um atributo de indexação de **V = 9** bytes, um apontador para blocos de **P = 6** bytes e um disco com blocos de **B = 512** bytes.
  - Cada nó tem **p** apontadores (para sub-árvores) e **p – 1** pares de entradas (valor de pesquisa/apontador para bloco de registos):
    **(p * P) + (p – 1) * (P + V) ≤ B ⇔ (p * 6) + (p – 1) * (6 + 9) ≤ 512 ⇒ p = 25**
  - A média de ocupação por nó é de **25 * 0.69 ≈ 17 apontadores e 16 entradas**.
- Em média, uma árvore B de ordem p = 25 tem a seguinte distribuição:
  - Raíz: 1 nó 17 apontadores 16 entradas
  - Nível 1: 17 nós 289 apontadores 272 entradas
  - Nível 2: 289 nós 4913 apontadores 4624 entradas
  - Nível 3: 4913 nós 0 apontadores 78.608 entradas
  - **Apontadores para blocos de registos: 16 + 272 + 4624 + 78.608 = 83.520**

# Árvores B+

- Estrutura dos nós internos de uma árvore B+.




- Estrutura dos nós folha de uma árvore B+.








# Árvores B+

- Os nós internos de uma árvore B+ de ordem p são definidos como:
  - Cada nó interno é da forma <$P_1$, $K_1$, ..., $K_{i-1}$, $P_i$, $K_i$, ..., $K_{n-1}$, $P_n$>, em que n ≤ p, cada $P_i$ é um apontador para uma sub-árvore B+, e cada $K_i$ é uma entrada para um valor de pesquisa.
  - Em cada nó interno têm-se que $K_1 < ... < K_{i-1} < K_i < ... < K_{n-1}$.
  - Para qualquer valor de pesquisa X na sub-árvore apontada por $P_i$ têm-se que $X \leq K_i$ para i = 1 ou $K_{i-1} < X \leq K_i$ para 1 < i < n ou $K_{i-1} < X$ para i = n.
  - Cada nó interno tem no máximo p apontadores para sub-árvores.
  - Cada nó interno, excepto o nó raíz, tem pelo menos $\lceil p/2 \rceil$ apontadores para sub-árvores. O nó raíz tem pelo menos 2 apontadores para sub-árvores a menos que seja o único nó da árvore.
  - Um nó interno com n apontadores para sub-árvores, n ≤ p, tem n – 1 valores de pesquisa.

# Árvores B+

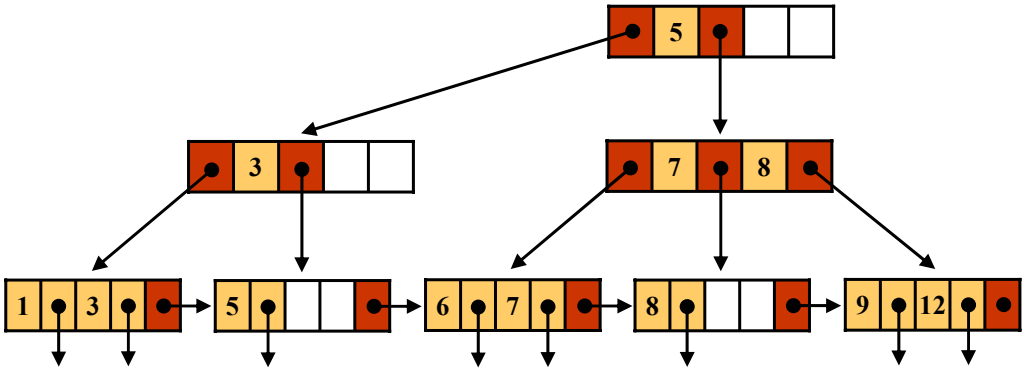
- Os nós folha de uma árvore B+ de ordem p' são definidos como:
  - Cada nó folha é da forma $<(K_1, B_1), ..., (K_{i-1}, B_{i-1}), (K_i, B_i), ..., (K_{n-1}, B_{n-1}), P_{next}>$, em que $n \leq p'$, cada par $(K_i, B_i)$ é uma entrada em que $B_i$ é o apontador para o bloco em disco que possui o registo com o valor de pesquisa $K_i$, e $P_{next}$ é um apontador para o próximo nó folha.
  - Em cada nó folha têm-se que $K_1 < ... < K_{i-1} < K_i < ... < K_{n-1}$.
  - Cada nó folha tem pelo menos $\lceil p'/2 \rceil$ pares de entradas $(K_i, B_i)$.
  - Todos os nós folha ficam no mesmo nível da árvore.






# Árvores B+

- Árvore B+ de ordem p = 3 nos nós internos e ordem p' = 2 nos nós folha.



# Árvores B+

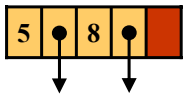
- Considere a construção de uma árvore B+ com um atributo de indexação de **V = 9** bytes, um apontador para blocos de **P = 6** bytes e um disco com blocos de **B = 512** bytes.
  - Cada nó interno tem **p** apontadores (para sub-árvores) e **p – 1** entradas (valores de pesquisa):
    **(p * P) + (p – 1) * V ≤ B ⇔ (p * 6) + (p – 1) * 9 ≤ 512 ⇒ p = 34**
  - A média de ocupação por nó interno é de **34 * 0.69 ≈ 23 apontadores e 22 entradas**.
  - Cada nó folha tem **p'** pares de entradas (valor de pesquisa/apontador para bloco de registos) e um apontador (para o próximo nó folha):
    **p' * (V + P) + P ≤ B ⇔ p' * (9 + 6) + 6 ≤ 512 ⇒ p' = 33**
  - A média de ocupação por nó folha é de **33 * 0.69 ≈ 22 entradas**.
- Em média, uma árvore B+ de ordem p = 34 e p' = 33 tem a seguinte distribuição:
  - Raíz: 1 nó 23 apontadores 22 entradas
  - Nível 1: 23 nós 529 apontadores 506 entradas
  - Nível 2: 529 nós 12.167 apontadores 11.638 entradas
  - Nós folha: 12.167 nós **12.167 * 22 = 267.674 apontadores para blocos de registos**



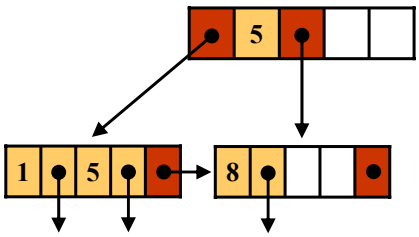


# Inserção em Árvores B+

- Inserindo os valores 8 e 5.

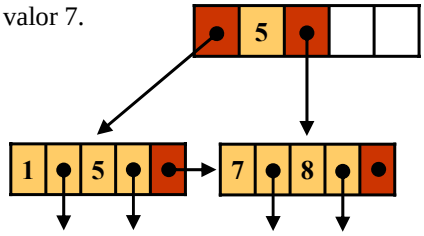



- Inserindo o valor 1 (*overflow*: novo nível).

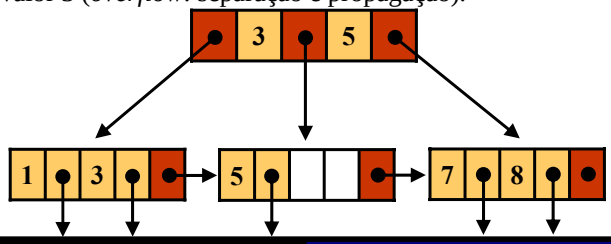


## Inserção em Árvores B+

- Inserindo o valor 7.




- Inserindo o valor 3 (*overflow*: separação e propagação).

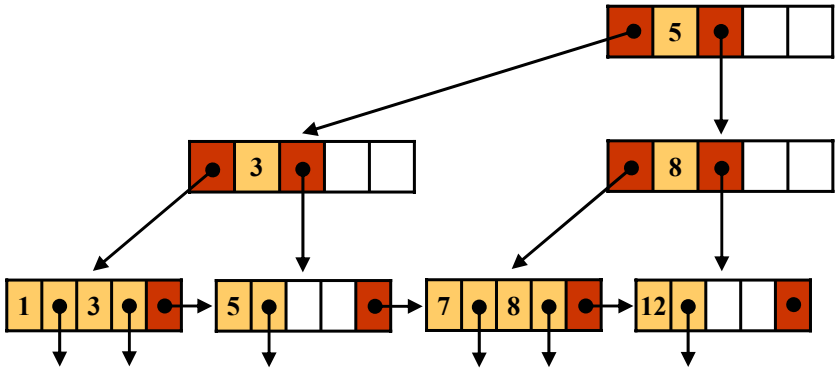






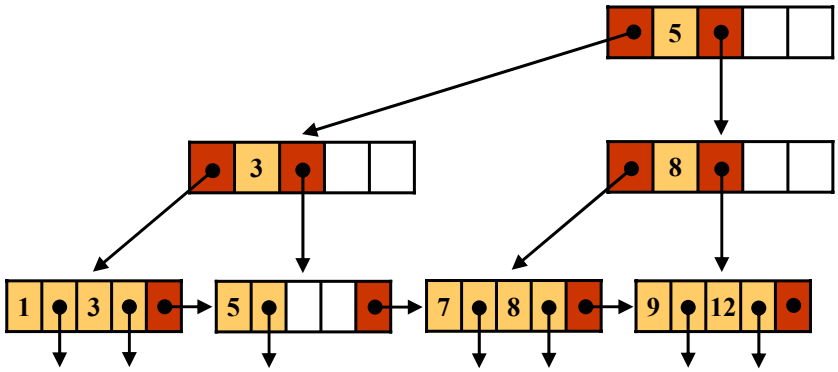

## Inserção em Árvores B+

- Inserindo o valor 12 (*overflow*: separação, propagação e novo nível).



## Inserção em Árvores B+

- Inserindo o valor 9.

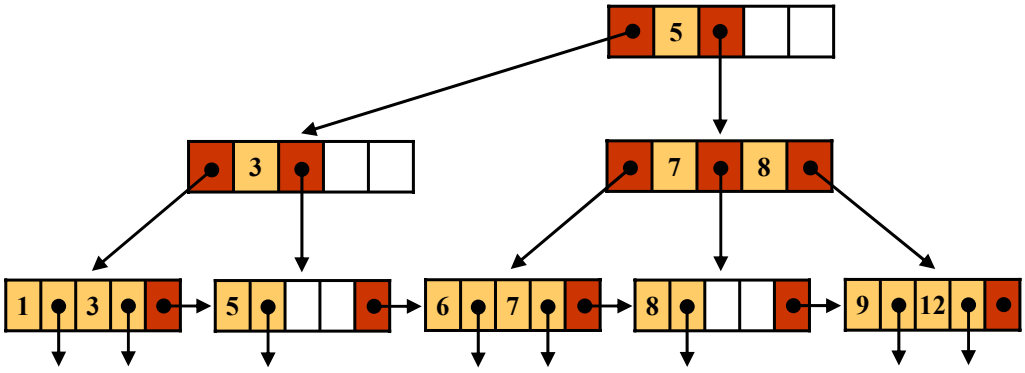







## Inserção em Árvores B+

- Inserindo o valor 6 (*overflow*: separação e propagação).



## Índices sobre Atributos Múltiplos

- Também é possível construir índices sobre mais do que um atributo. A diferença é que em lugar de termos apenas um atributo de indexação, temos um tuplo com n atributos de indexação. Por exemplo, um índice construído sobre dois atributos Atrib1 e Atrib2 tem como tuplo de indexação o par <Atrib1, Atrib2>.
- Os índices sobre atributos múltiplos são guardados em **ficheiros ordenados pelo tuplo de indexação**. Cada entrada no ficheiro de índices é composta por:
  - Um tuplo de valores dos atributos de indexação.
  - Um apontador para o bloco que possui o registo com esses valores ou um apontador para o bloco que possui apontadores para os registos com esses valores.
- Todas as estruturas de dados referidas anteriormente para construir índices sobre atributos únicos podem ser adaptadas para construir índices sobre atributos múltiplos.





## Índices em SQL

- Criar um ficheiro de índices sobre os atributos de uma dada tabela:

  **CREATE** [**UNIQUE**] **INDEX** <NOME>

  **ON** <TABELA>(<ATRIB_1> [**DESC**], ..., <ATRIB_N> [**DESC**]);
  - A opção **UNIQUE** indica que não existem valores iguais para os atributos de indexação.
  - A opção **DESC** indica que a ordenação do atributo correspondente deve ser feita por ordem descendente.
- Remover um índice quando este não é mais necessário.

  **DROP INDEX** <ÍNDICE>;

# Índices em SQL

- Indexar a relação DEPARTAMENTO com base no atributo Nome.

  **CREATE UNIQUE INDEX** DNOME_INDEX

  **ON** DEPARTAMENTO(Nome);

- Indexar a relação EMPREGADO com base nos atributos NomeP (ordem descendente) e NomeF.

  **CREATE INDEX** ENOME_INDEX

  **ON** EMPREGADO(NomeP **DESC**, NomeF);

- Remover os índices definidos anteriormente.

  **DROP INDEX** DNOME_INDEX;

  **DROP INDEX** ENOME_INDEX;