ACH2024

Aula 16

Organização de arquivos: Alocação indexada

Profa. Ariane Machado Lima

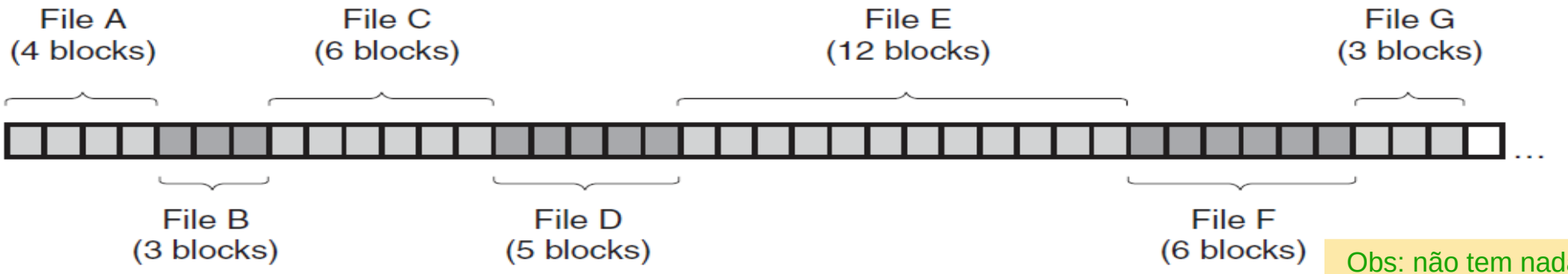
Aulas passadas

Algoritmos e estruturas de dados para lida com memória secundária

- Organização interna de arquivos
- Acesso à memória secundária (por blocos - seeks)
- Tipos de alocação de arquivos na memória secundária:
  - Sequencial (ordenado e não ordenado)
  - Ligada
  - Indexada
  - Árvores-B
  - Hashing (veremos também hashing em memória principal)
- Algoritmos de processamento cossequencial e ordenação em disco

Aula passada - Alocação sequencial

- Blocos alocados sequencialmente no disco (pelos cilindros)



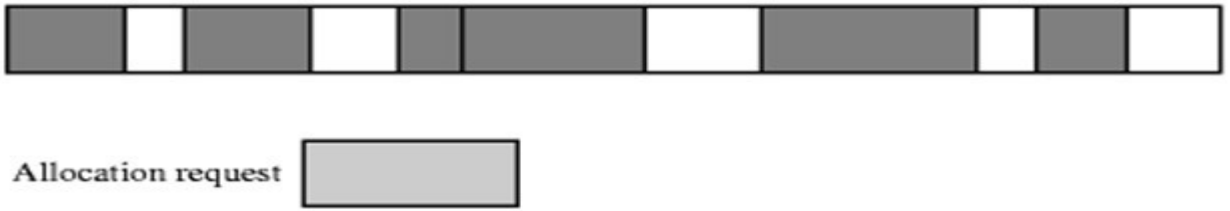
- E como os registros são organizados pelos blocos?
  - Não ordenados (**sequencial não ordenado** – heap files)
  - Ordenados por um campo chave (**sequencial ordenado** - sorted files)

Obs: não tem nada a ver com a estrutura de dados "heap"

Alocação sequencial

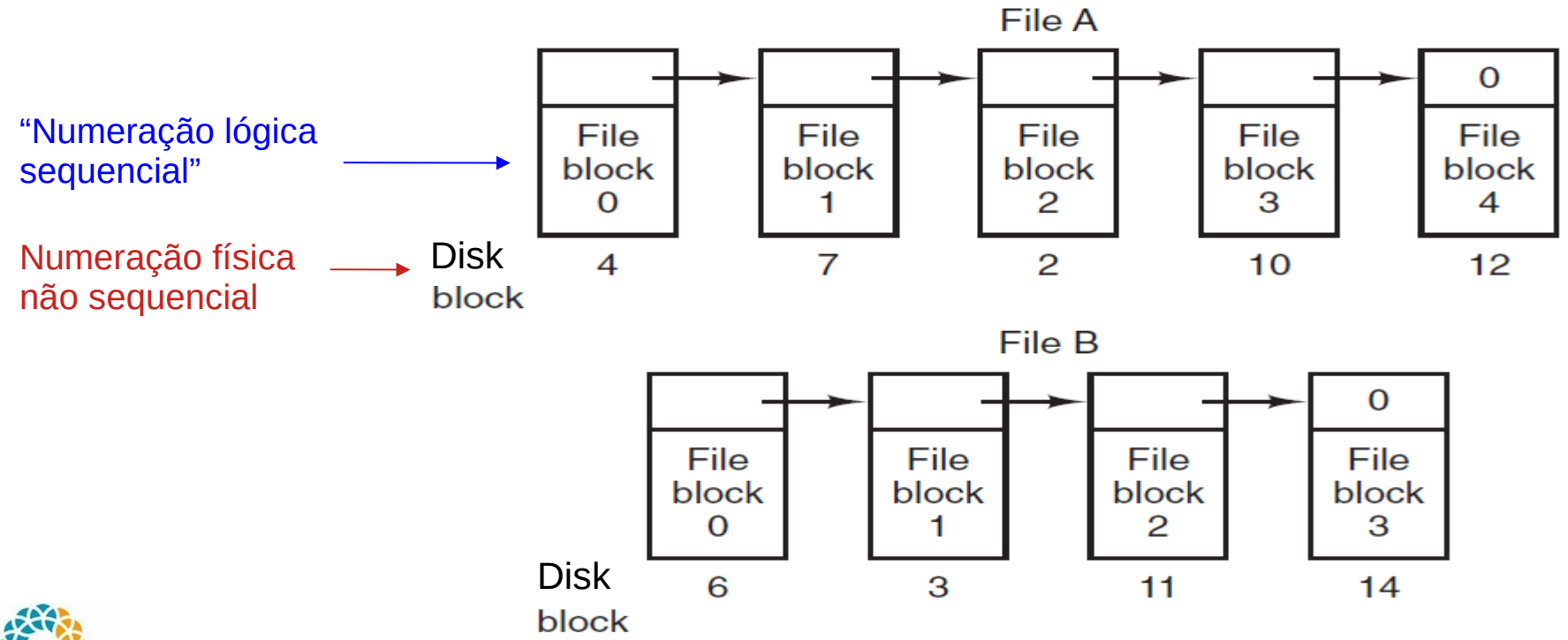
Fragmentação externa:

- Com o tempo (após alocações/desalocações sucessivas), o disco pode ficar fragmentado, isto é, com vários trechos disponíveis intercortados por trechos utilizados



Alocação por listas ligadas

Cada arquivo é uma lista ligada de blocos



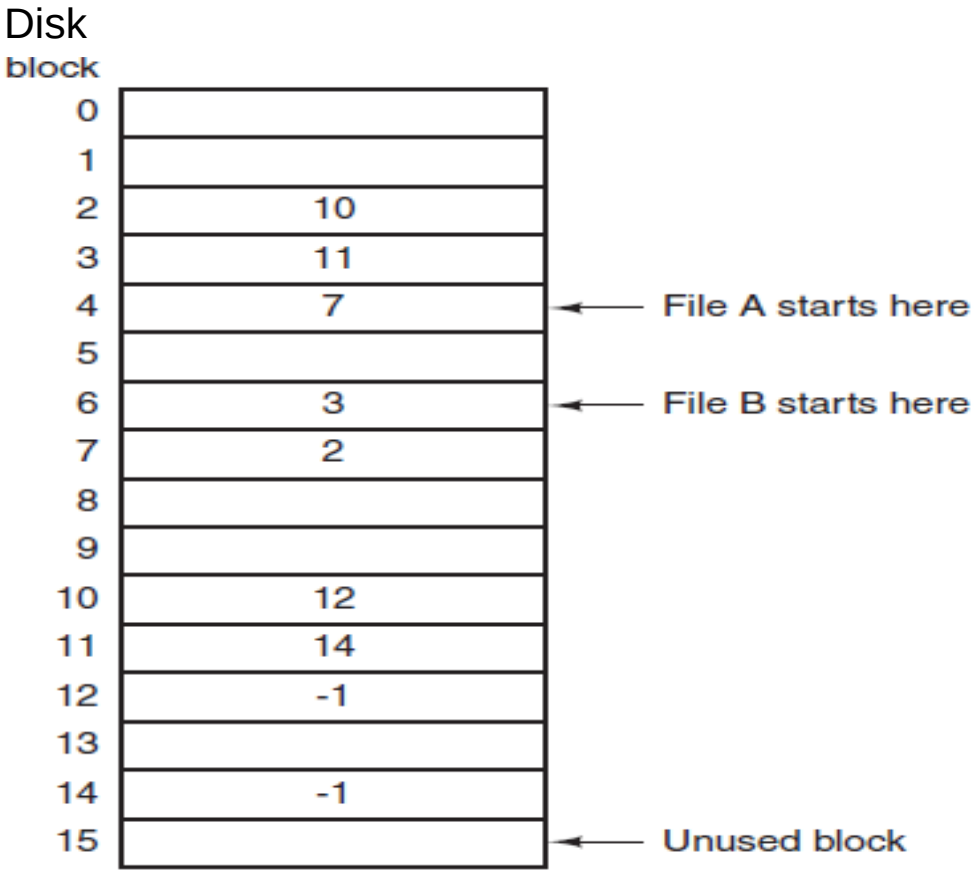
Fonte: (TANEMBAUM, 2015)

Complexidades (sempre em termos de nr de seeks...)

	Sequencial		Ligada
	Não-Ordenado	**Ordenado**	**Ordenado**
Busca	O(b)	O(lg b)	O(b), O(lg b) só se usar FAT (discos pequenos)
Inserção**	O(1) se tiver espaço no final, O(b) c.c.	O(1) se tiver espaço no bloco, O(b) c.c.	O(1)
Remoção* , **	O(1)	O(1)	O(1)
Leitura ordenada	ω(b) (depende do alg de ord. externa)	O(b), O(1) se fizer a leitura toda de uma vez	O(b)
Mínimo/máximo	O(b)	O(1)	O(1)
Modificação**	O(1)	O(b) se no campo chave, O(1) c.c.	O(1)
* considerando uso de bit de validade			
** considerando que já se sabe a localização do registro (busca já realizada)			

Alocação por listas ligadas com uso de uma File Allocation Table (FAT) em memória

- t[i] armazena o próximo bloco do bloco
- Vantagens:
  - Economiza espaço nos blocos de dados (que só conterão dados e não ponteiros) → preciso de menos blocos → b menor impacta nas velocidades dependentes de b
  - Para uma leitura aleatória (dado um deslocamento em relação ao início do arquivo), o encadeamento (para achar o bloco certo) é seguido apenas sobre a tabela (que está toda em memória) → O(1)



Fonte: (TANEMBAUM, 2015)

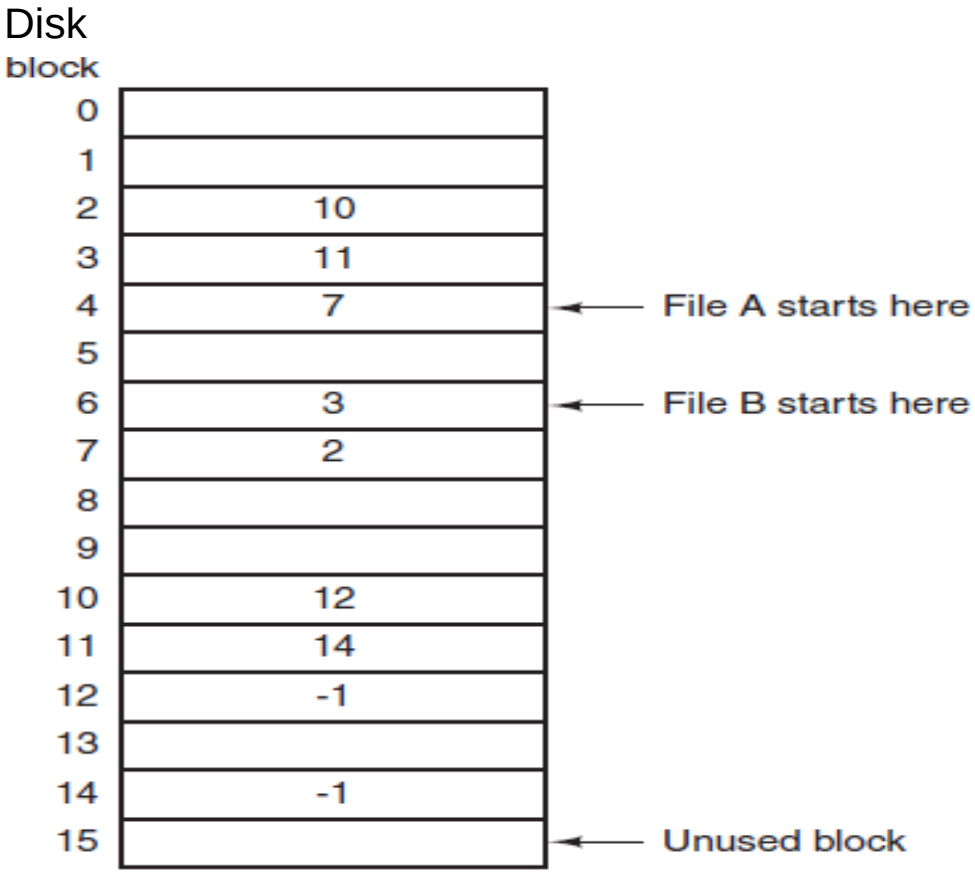
Daria para implementar uma busca binária!!!

Alocação por listas ligadas com uso de uma File Allocation Table (FAT) em memória

- t[i] armazena o próximo bloco do bloco
- Desvantagens:
  - Não escalável para grandes discos

Ex: para um disco de 1TB e blocos de 1KB, a tabela ocuparia 3GB de memória

  - Sistema de arquivos FAT32, por ex, impõe tamanho máximo de disco de 2TB



Fonte: (TANEMBAUM, 2015)

Outra alternativa?

- Lista ligada aproveita espaço (resolve fragmentação externa), mas leitura aleatória fica horrível (sem tabela de alocação)
- Tabela de alocação acelera a leitura aleatória mas gasta muita memória
- Como diminuir esse último problema?
- Por que manter em memória as informações de arquivos que não foram abertos? Que tal “uma tabela” por arquivo?

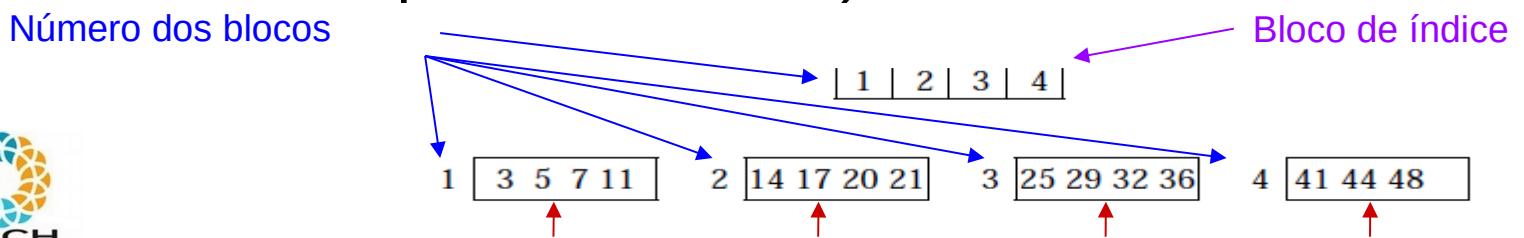
Aula de hoje

Algoritmos e estruturas de dados para lida com memória secundária

- Organização interna de arquivos
- Acesso à memória secundária (por blocos - seeks)
- Tipos de alocação de arquivos na memória secundária:
  - Sequencial (ordenado e não ordenado)
  - Ligada
  - Indexada
  - Árvores-B
  - Hashing (veremos também hashing em memória principal)
- Algoritmos de processamento cossequencial e ordenação em disco

Alocação indexada

- Um ou mais blocos de índices contém ponteiros para os blocos de fato
- Blocos de índices são como uma tabela de alocação específica daquele arquivo
  - i-ésima entrada do primeiro bloco de índice contém o número do i-ésimo bloco de dado do arquivo
- Blocos de índice carregados na memória sob demanda (assim como arquivos de dados)



Alocação indexada

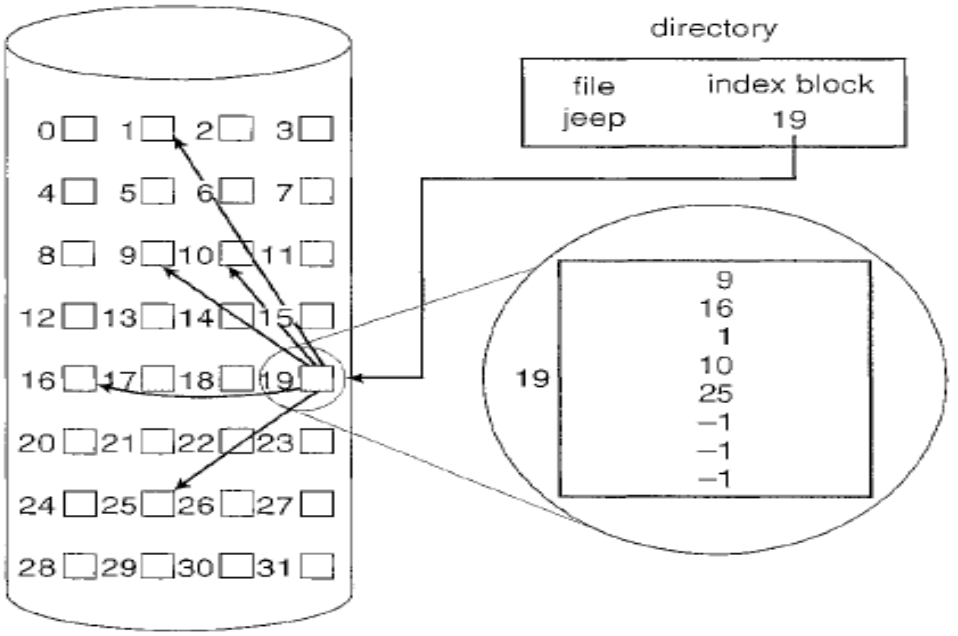


Figure 11.8 Indexed allocation of disk space.

Fonte: (SILBERCHATZ et al, 2009)

Cabeçalhos de arquivo do tipo I-nodes (index-nodes): visão geral

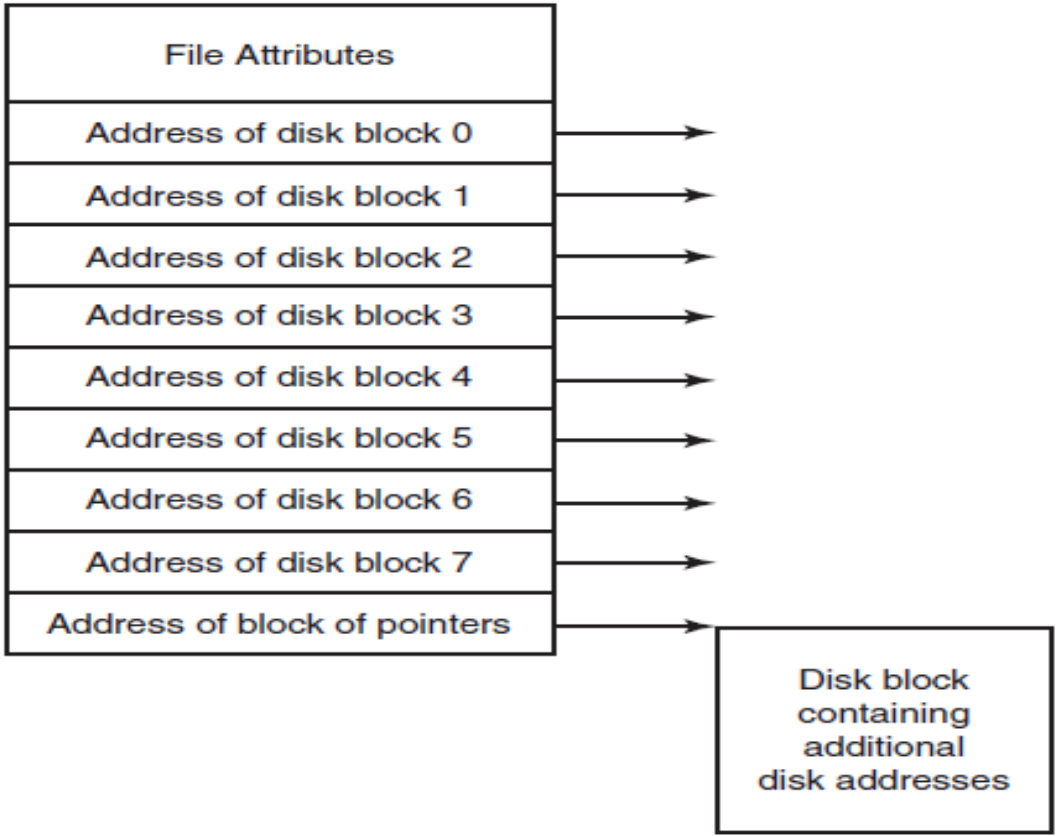
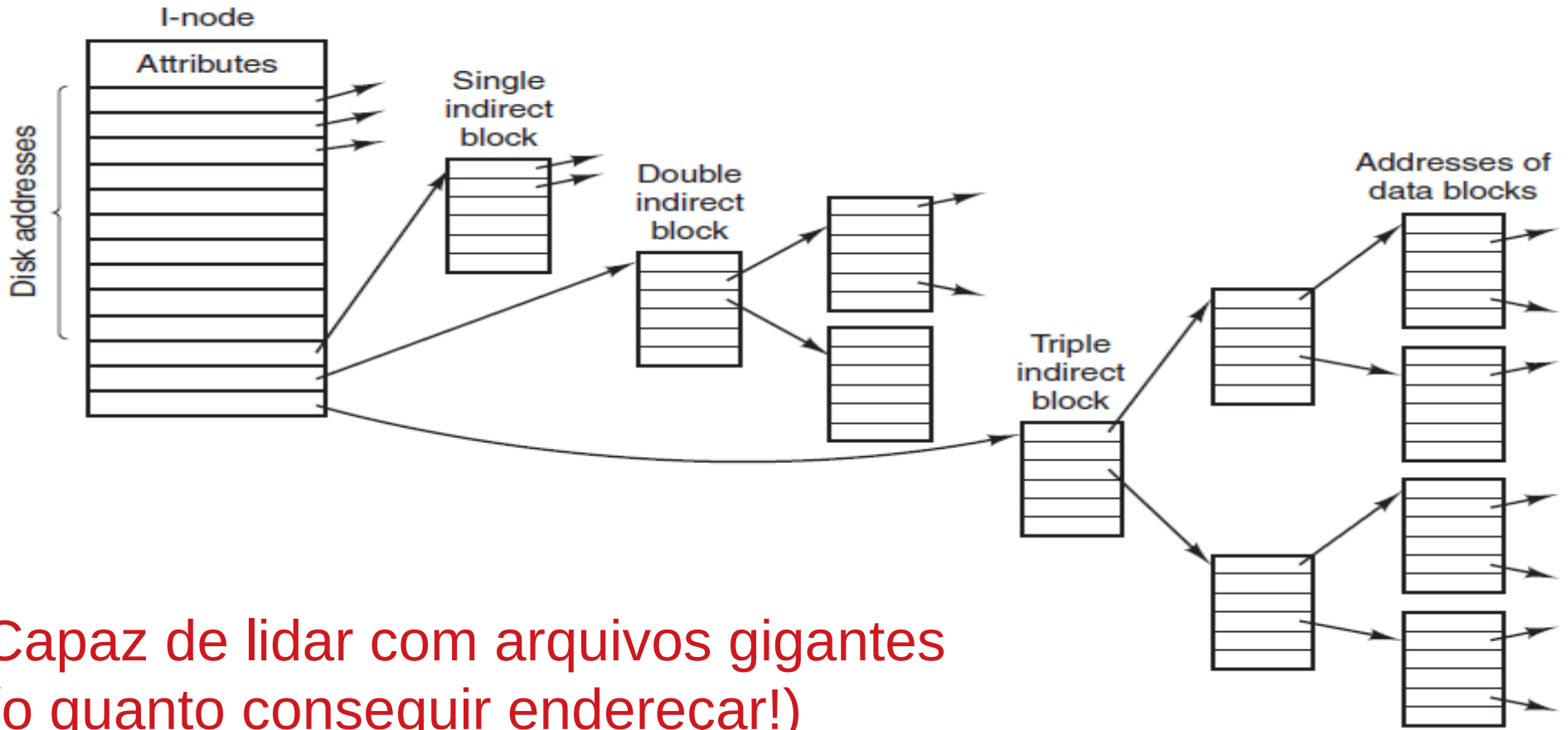


Figure 4-13. An example i-node.

Fonte: (TANEMBAUM, 2015)

I-nodes (index-nodes) do UNIX/LINUX:



Capaz de lidar com arquivos gigantes
(o quanto conseguir endereçar!)

Fonte: (TANEMBAUM, 2015)

Leitura complementar

- Mais detalhes sobre o sistema de arquivos do Linux e Windows: cap 4, 10 e 11 do livro do Tanenbaum (referência no último slide)

Índices

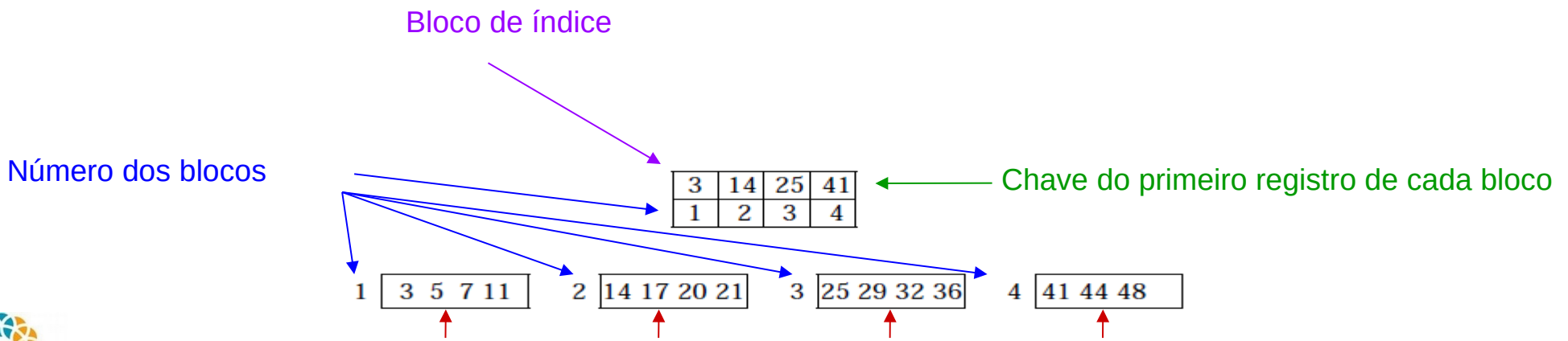
- Podem ser utilizados para:
  - Organização física dos arquivos (como visto aqui)
  - Estruturar caminhos secundários de acesso aos dados, de forma a acelerar buscas

Índices

- Podem ser utilizados para:
  - Organização física dos arquivos (como visto a seguir)
  - Estruturar caminhos secundários de acesso aos dados, de forma a acelerar buscas

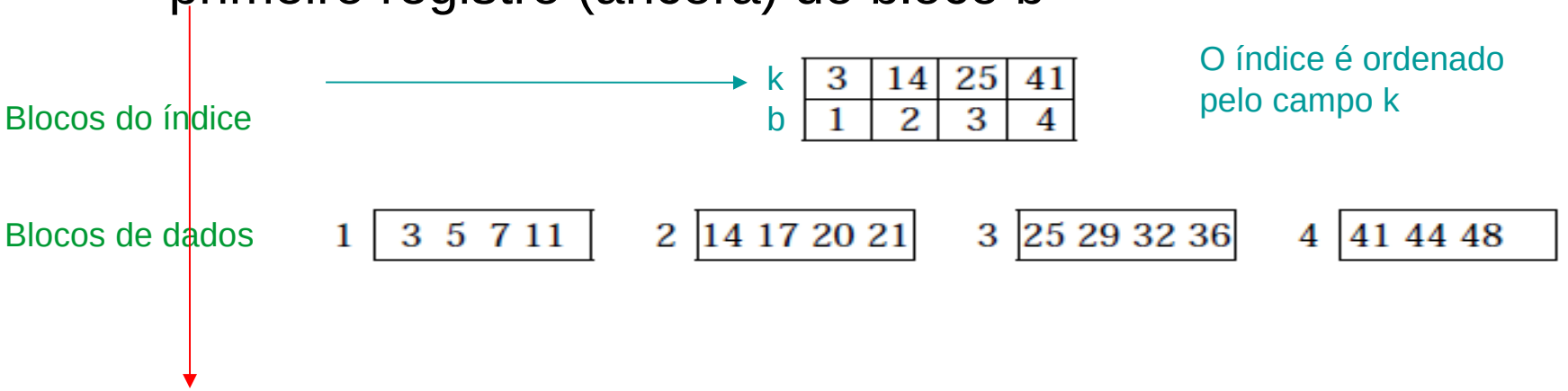
Alocação indexada – arquivos ordenados

- Os blocos de índices possuem, além dos ponteiros para os blocos de dados, a chave do primeiro registro de cada bloco de dado



Alocação indexada

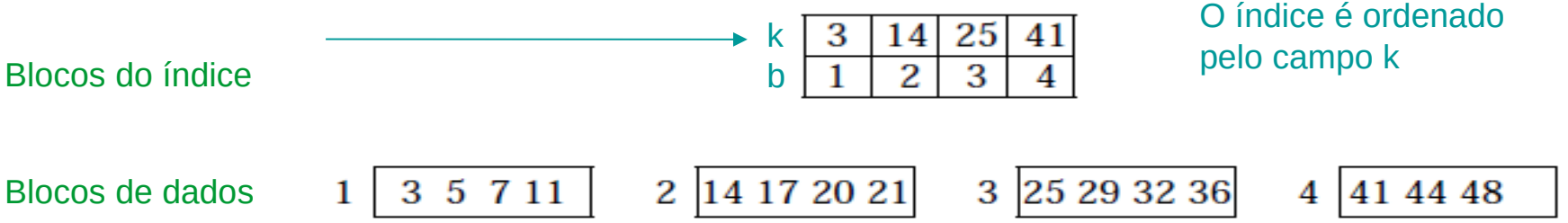
- Índice primário: arquivo ORDENADO de registros de tamanho fixo contendo os campos <k,b>, sendo k a chave (primária) do primeiro registro (âncora) do bloco b



Isto é, k possui valores ÚNICOS e ORDENA FISICAMENTE os registros

Alocação indexada

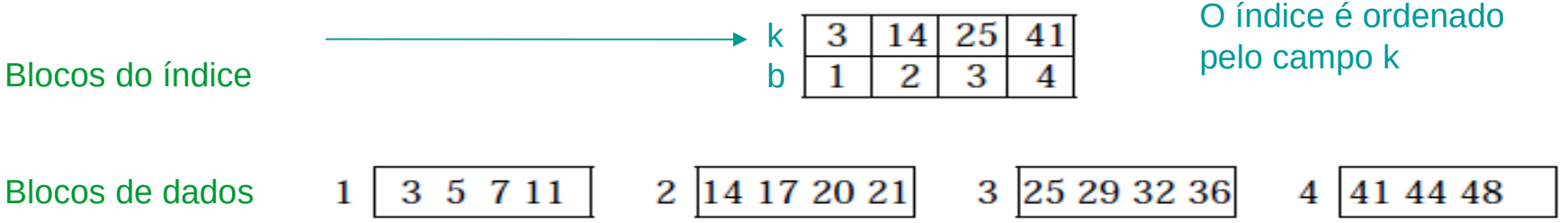
- Índice primário: arquivo ORDENADO de registros de tamanho fixo contendo os campos <k,b>, sendo k a chave (primária) do primeiro registro (âncora) do bloco b



- Qual a vantagem?

Alocação indexada

- Índice primário: arquivo ORDENADO de registros de tamanho fixo contendo os campos <k,b>, sendo k a chave (primária) do primeiro registro (âncora) do bloco b



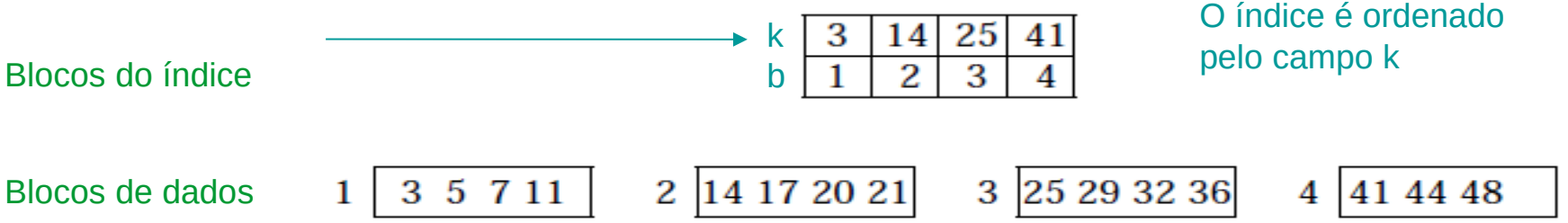
Qual a vantagem?

Não só permite uma busca binária, mas ela é muito rápida!!!

Índice tem bi blocos, sendo bi << B (B = nr de blocos de dados, no exemplo acima, bi = 1 e B = 4), pois cada entrada é menor que um registro, e há só uma entrada por bloco de dado representado (b)→ complexidade da busca binária:

Alocação indexada

- Índice primário: arquivo ORDENADO de registros de tamanho fixo contendo os campos <k,b>, sendo k a chave (primária) do primeiro registro (âncora) do bloco b



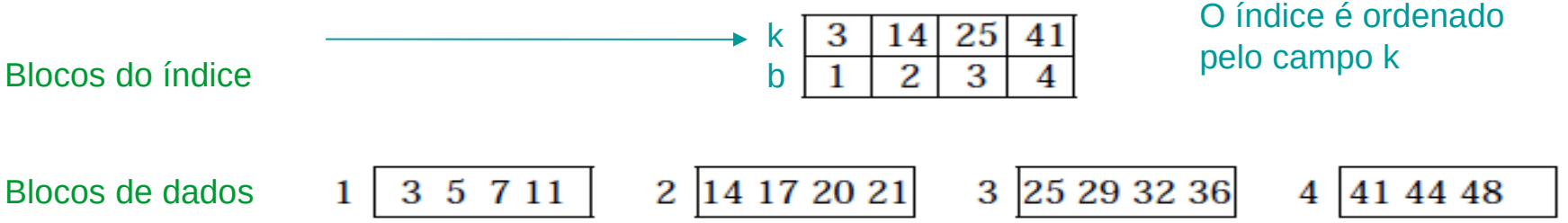
Qual a vantagem?

Não só permite uma busca binária, mas ela é muito rápida!!!

Índice tem bi blocos, sendo bi << B (B = nr de blocos de dados, no exemplo acima, bi = 1 e B = 4), pois cada entrada é menor que um registro, e há só uma entrada por bloco de dado representado (b)→ complexidade da busca binária: O(log bi)

Alocação indexada

- Índice primário: arquivo ORDENADO de registros de tamanho fixo contendo os campos <k,b>, sendo k a chave (primária) do primeiro registro (âncora) do bloco b



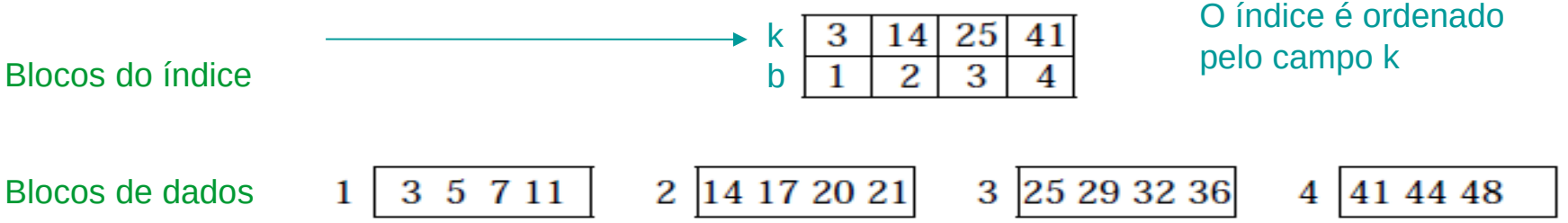
- Observação:

  este é um índice **esparso**

  se fosse **denso**, teria uma entrada por registro de dado

Alocação indexada

- Índice primário: arquivo ORDENADO de registros de tamanho fixo contendo os campos <k,b>, sendo k a chave (primária) do primeiro registro (âncora) do bloco b



- Como fica a inserção e remoção neste tipo de alocação?

Alocação indexada

- Problema: inserção / remoção
  - Altera a posição em disco de vários registros: O(b + bi) = O(b)
    - → altera âncora de vários blocos
    - → altera várias entradas do índice primário
  - Formas de contornar o problema:
    - Remoção por bits de validade (O(1) para remoção neste caso, mas arquivo de índice pode ficar desnecessariamente grande, afetando buscas
    - Inserção: usar um arquivo desordenado de overflow (inserção em O(1))
      - Ou uma lista ligada de overflow (os registros nos blocos e na lista podem ser ordenados para melhorar a busca)

      (LEMBRANDO DAS REORGANIZAÇÕES PERIÓDICAS)

Alocação indexada

- Problema: inserção / remoção
  - Altera a posição em disco de vários registros: O(b + bi) = O(b)
    - → altera âncora de vários blocos
    - → altera várias entradas do índice primário
  - Formas de contornar o problema:
    - Remoção por bits de validade (O(1) para remoção neste caso, mas arquivo de índice pode ficar desnecessariamente grande, afetando buscas
    - Inserção: usar um arquivo desordenado de overflow (inserção em O(1))
      - Ou uma lista ligada de overflow (os registros nos blocos e na lista podem ser ordenados para melhorar a busca)

      (LEMBRANDO DAS REORGANIZAÇÕES PERIÓDICAS)

VELHO DILEMA ENTRE TEMPO DE BUSCA E DINAMISMO!

Alocação indexada

- **Leitura sequencial**:

Alocação indexada

- **Leitura sequencial**: O(b+bi) = O(b)

  Tem que acessar cada bloco de índice para acessar cada bloco de dado na ordem correta

  Mesmo que em uma transação

	Sequencial		Ligada	Indexada
	Não-Ordenado	**Ordenado**	**Ordenado**	**Ordenado**
Busca	O(b)	O(lg b)	O(b), O(lg b) só se usar FAT (discos pequenos)	O(log bi)
Inserção**	O(1) se tiver espaço no final, O(b) c.c.	O(1) se tiver espaço no bloco, O(b) c.c.	O(1)	O(b+bi) = O(b)
Remoção* , **	O(1)	O(1)	O(1)	O(1)
Leitura ordenada	ω(b) (depende do alg de ord. externa)	O(b), O(1) se fizer a leitura toda de uma vez	O(b)	O(b+bi) = O(b)
Mínimo/máximo	O(b)	O(1)	O(1)	O(1)
Modificação**	O(1)	O(b) se no campo chave, O(1) c.c.	O(1)	O(b) se no campo chave, O(1) c.c.
* considerando uso de bit de validade				
** considerando que já se sabe a localização do registro (busca já realizada)				

VELHO DILEMA ENTRE TEMPO DE BUSCA E DINAMISMO!

E se o campo de ordenação física não tiver valores únicos?

E se o campo de ordenação física não tiver valores únicos?

→ Índice de *clustering*

Índice de clustering

- Índice de clustering: arquivo ORDENADO de registros de tamanho fixo contendo os campos <c, b>, sendo c um campo de classificação física que não possui valores distintos
  - Uma entrada <c, b> para cada valor distinto de c, sendo b o primeiro bloco da primeira ocorrência da chave com valor c

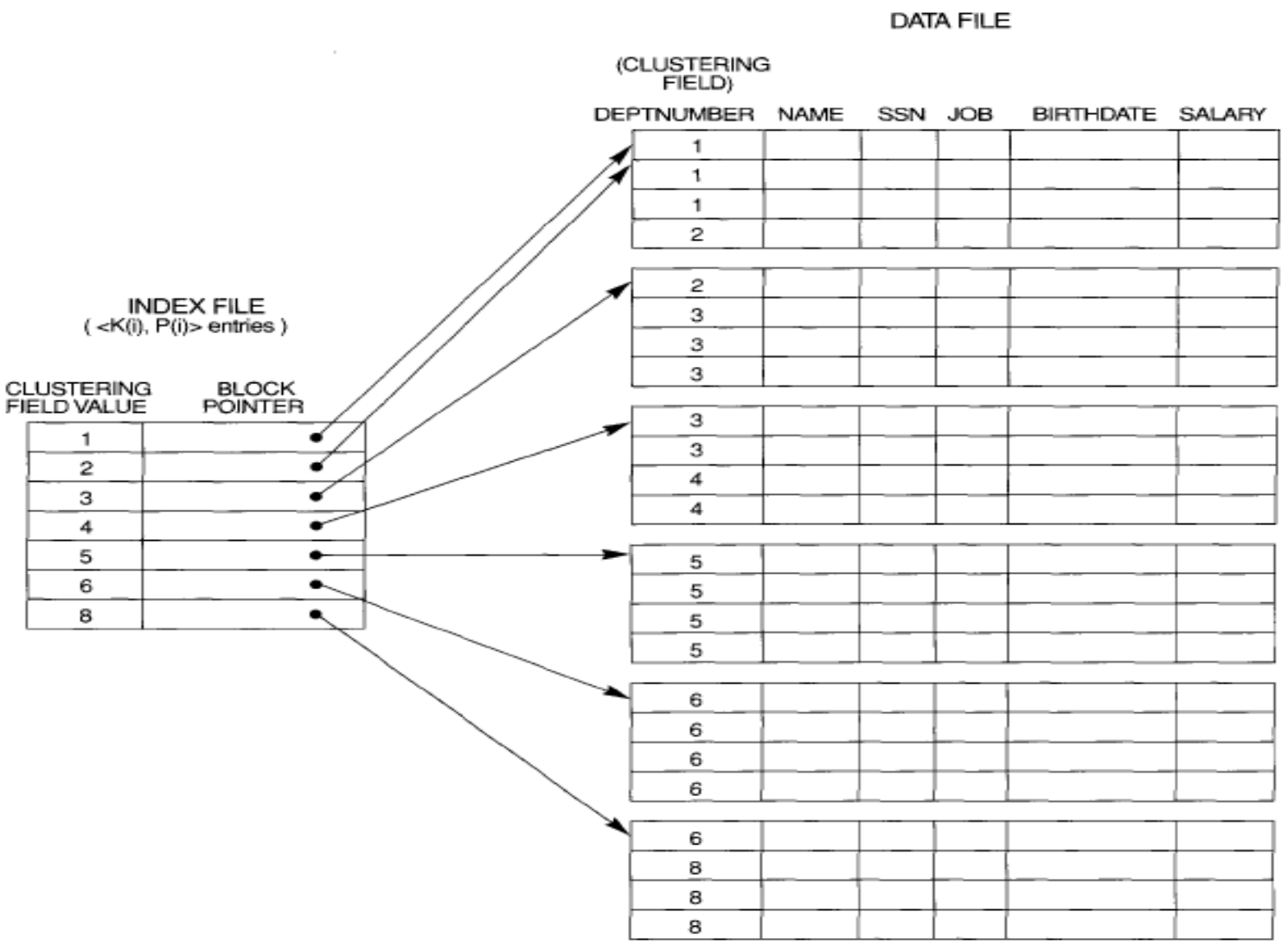


FIGURE 14.2 A clustering index on the DEPTNUMBER ordering nonkey field of an EMPLOYEE file.

Fonte:
Elmasri & Navathe

Índice de clustering

- Inserção / remoção: ainda problemático, pois c ordena fisicamente os registros
  - Alternativas:
    - Reservar um ou mais blocos para cada valor de c (ligados por ponteiros)
    - Remoção por bit de validade

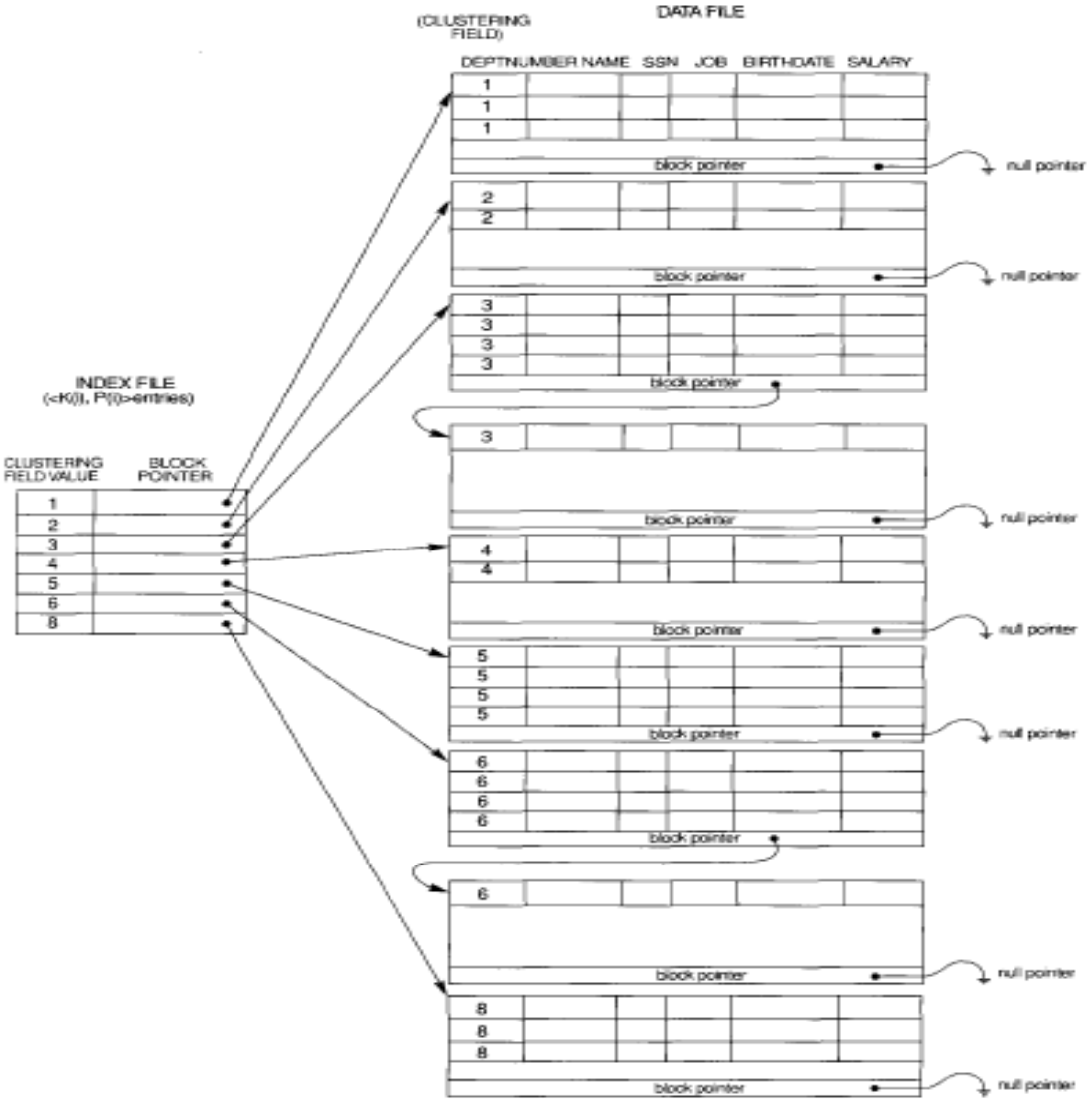


FIGURE 14.3 Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

Fonte:
Elmasri & Navathe

Índice de clustering

- Busca:

Índice de clustering

- Busca:
  - Também binária nos blocos de índice
  - Busca sem sucesso: NÃO acessa bloco de dado
  - Busca com sucesso: quer só o primeiro registro ou todos?
    - Primeiro: acesso a um bloco de dado (informado no índice)
    - Todos: tem que ler q blocos (todos daquele valor de chave)
  - Quanto maior o número de valores distintos, maior o tempo de busca binária nos índices

Índices primários, clustering, e...

- Índices primários e de clustering são baseados no campo de ordenação física de um arquivo

  → cada arquivo pode ter no máximo um índice primário OU um índice de clustering

- E para os campos que não ordenam fisicamente o arquivo? Podemos ter algo semelhante?

  – Quantos **índices secundários** quiser!

Índices secundários

- Índice secundário: arquivo ORDENADO de registros de tamanho fixo contendo os campos <i, w>, sendo i um campo de indexação que NÃO ordena fisicamente os registros, podendo ter valores distintos ou não
  - w aponta para um bloco ou registro
- Podem existir vários índices secundários

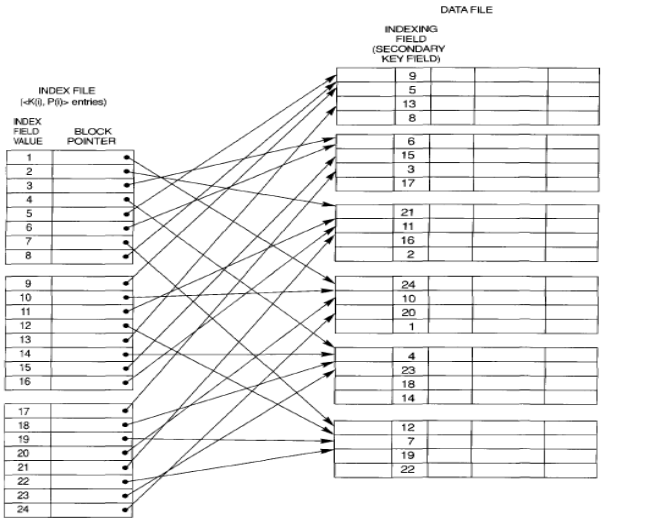


FIGURE 14.4 A dense secondary index (with block pointers) on a nonordering key field of a file.

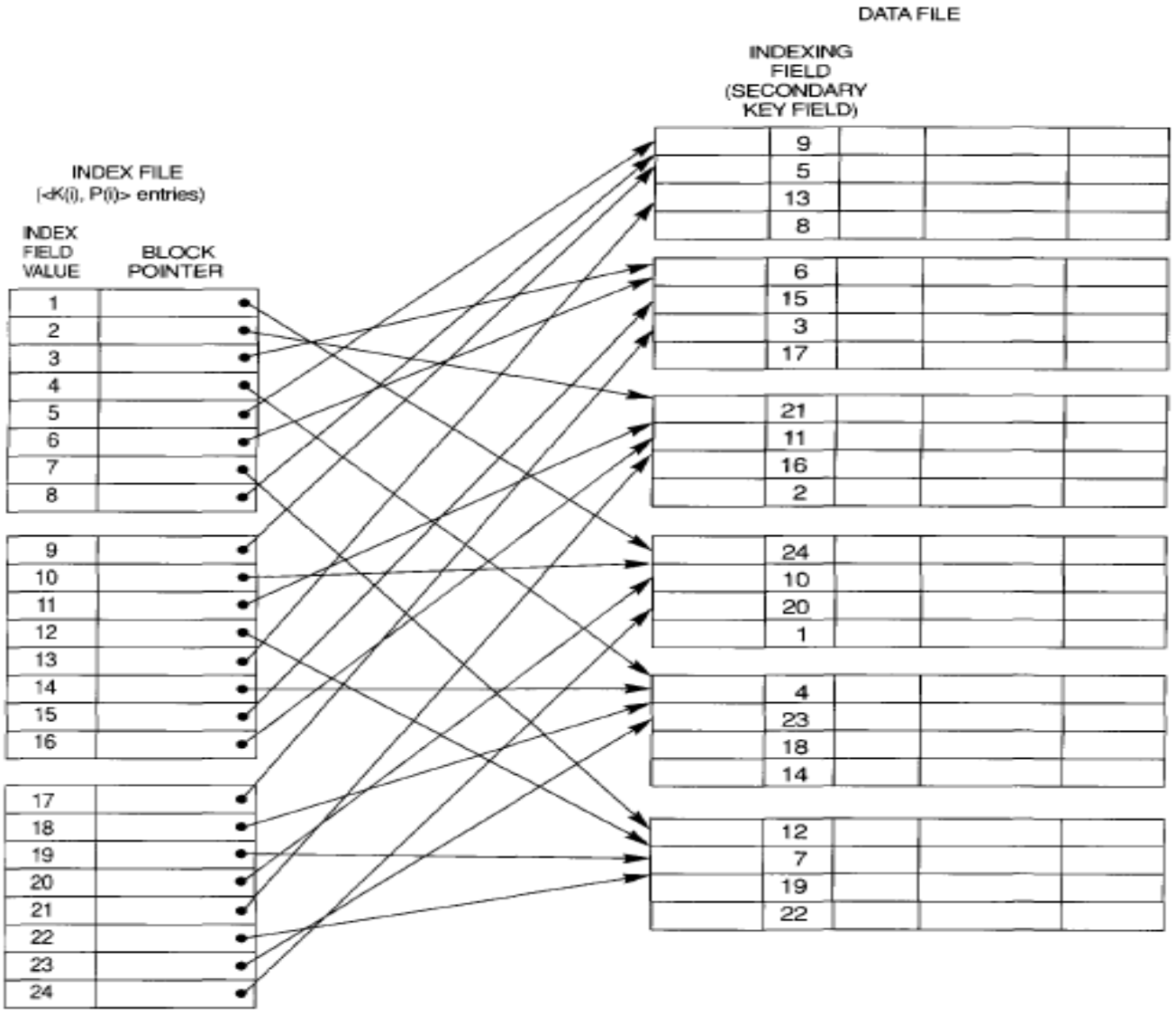


FIGURE **14.4** A dense secondary index (with block pointers) on a nonordering key field of a file.

Fonte:
Elmasri & Navathe

Índices secundários

- Se i tem valores distintos, o índice é denso
- Se i tem valores duplicados:
  - Opção 1: diversas entradas para um mesmo i, cada uma com w apontando para um registro (denso)
  - Opção 2: 1 entrada para cada valor de i, e w multivalorado (campo de tamanho variável) aponta para blocos (esparso)
  - Opção 3: uma entrada para cada valor de i e w (campo de tamanho fixo) aponta para um bloco de ponteiros de registros (esparso) – mais usada

Pensem na busca, inserção e remoção em cada caso...

Para esses 3 tipos de índices:

- Busca binária – O(log bi)
- O que dá para fazer se o arquivo é muito grande e o próprio índice ficou grande (com muitos blocos, ie, grande bi)?

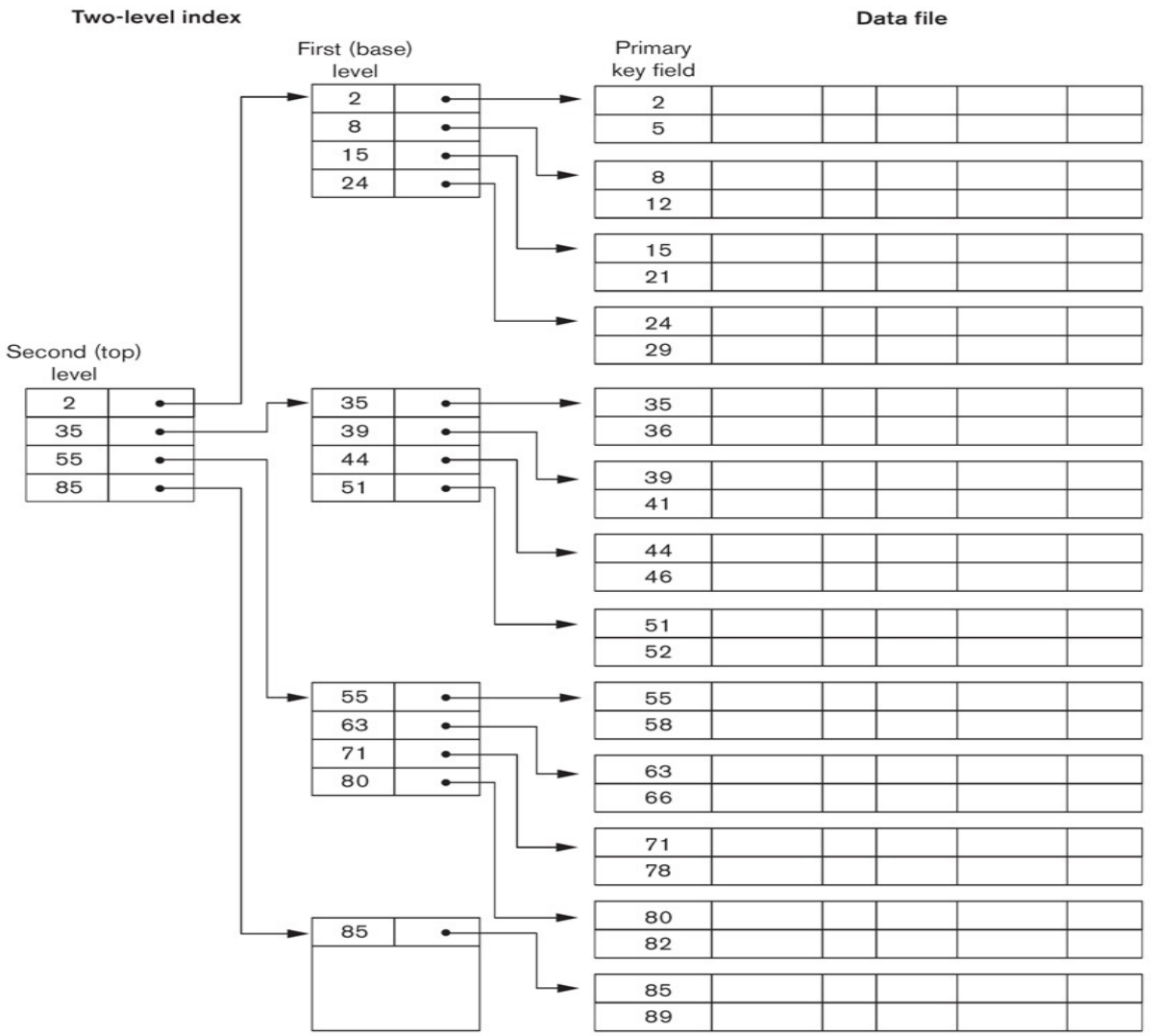
Para esses 3 tipos de índices:

- Busca binária – O(log bi)
- O que dá para fazer se o arquivo é muito grande e o próprio índice ficou grande (com muitos blocos, ie, grande bi)?
  - ÍNDICE DO ÍNDICE!!!

Organização indexada multiníveis

- Nível 1: arquivo de índices para os dados
- Nível 2: arquivo de índices para o arquivo de índices nível 1
- Se no nível 2 precisar de mais de um bloco, criar nível 3!
- ….

Figure 18.6
A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.



Organização indexada multiníveis

- Nível 1: arquivo de índices para os dados
- Nível 2: arquivo de índices para o arquivo de índices nível 1
- Se no nível 2 precisar de mais de um bloco, criar nível 3!
- ….
- **Busca**:

Organização indexada multiníveis

- Nível 1: arquivo de índices para os dados
- Nível 2: arquivo de índices para o arquivo de índices nível 1
- Se no nível 2 precisar de mais de um bloco, criar nível 3!
- ….
- **Busca**: 1 acesso em cada nível (rápida!) - O(t) - precisa acessar t blocos, sendo t o número de níveis

  t =

Organização indexada multiníveis

- Nível 1: arquivo de índices para os dados
- Nível 2: arquivo de índices para o arquivo de índices nível 1
- Se no nível 2 precisar de mais de um bloco, criar nível 3!
- ….
- **Busca**: 1 acesso em cada nível (rápida!) - O(t) - precisa acessar t blocos, sendo t o número de níveis

$t = \text{ceil}(\log_{fbi} r^1)$,

fbi = nr de registros que cabem em um bloco de índice (fator de blocagem dos blocos de índice)

r^1 = nr total de registros de índice no nível 1

Organização indexada multiníveis

- **Inserção / remoção**: ?

Organização indexada multiníveis

- **Inserção / remoção**: cada vez mais complicado!!!
  - Posso ter que alterar tudo!
  - Aplicáveis as mesmas “gambiarras” das mencionadas na alocação indexada de um nível

Organização indexada multiníveis

- **Inserção / remoção**: cada vez mais complicado!!!
  - Posso ter que alterar tudo!
  - Aplicáveis as mesmas “gambiarras” das mencionadas na alocação indexada de um nível

	Sequencial		Ligada	Indexada	Indexada Multinível
	Não-Ordenado	Ordenado	Ordenado	Ordenado	Ordenado
Busca	O(b)	O(lg b)	O(b), O(lg b) só se usar FAT (discos pequenos)	O(log bi)	O(log $_{fbi}$ r[1])
Inserção**	O(1) se tiver espaço no final, O(b) c.c.	O(1) se tiver espaço no bloco, O(b) c.c.	O(1)	O(b+bi) = O(b)	O(b+bi) = O(b)
Remoção* , **	O(1)	O(1)	O(1)	O(1)	O(1)
Leitura ordenada	ω(b) (depende do alg de ord. externa)	O(b), O(1) se fizer a leitura toda de uma vez	O(b)	O(b+bi) = O(b)	O(b+bi) = O(b)
Mínimo/máximo	O(b)	O(1)	O(1)	O(1)	O(1)
Modificação**	O(1)	O(b) se no campo chave, O(1) c.c.	O(1)	O(b) se no campo chave, O(1) c.c.	O(b) se no campo chave, O(1) c.c.
* considerando uso de bit de validade					
** considerando que já se sabe a localização do registro (busca já realizada)					

VELHO DILEMA ENTRE TEMPO DE BUSCA E DINAMISMO!

O que fazer??? Como ter uma busca eficiente mas permitir uma inserção e remoção razoável?

Lembrando de AED 1...

- Busca eficiente em dados ordenados sem gastar memória:

Lembrando de AED 1...

- Busca eficiente em dados ordenados sem gastar memória:
  - Busca binária (em um vetor)
  - Mas o problema era justamente inserção / deleção
  - Qual era a alternativa de **continuar fazendo busca binária** mas permitir dinamismo de inserção / remoção?

Lembrando de AED 1...

- Busca eficiente em dados ordenados sem gastar memória:
  - Busca binária (em um vetor)
  - Mas o problema era justamente inserção / deleção
  - Qual era a alternativa de **continuar fazendo busca binária** mas permitir dinamismo de inserção / remoção?
- Árvores Binárias de Busca!!!!

Lembrando de AED 1...

- Busca binária (em um vetor):

  -4, 2, 3, 5, 19, 21, 25

Lembrando de AED 1...

- Busca binária (em um vetor):

  -4, 2, 3, 5, 19, 21, 25

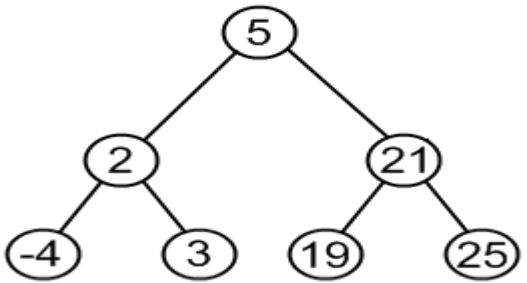
- Árvores Binárias de Busca:

Lembrando de AED 1...

- Busca binária (em um vetor):

  -4, 2, 3, 5, 19, 21, 25

- Árvores Binárias de Busca:

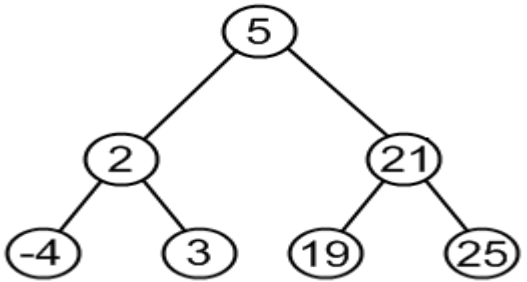


Lembrando de AED 1...

- Busca binária (em um vetor):

  -4, 2, 3, 5, 19, 21, 25

- Árvores Binárias de Busca:



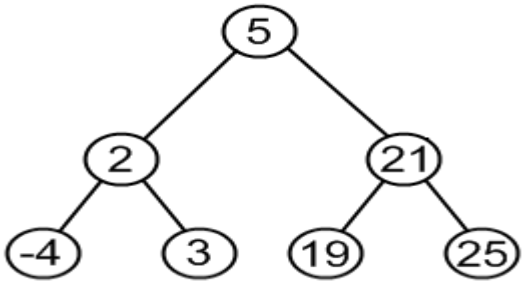
IMPLEMENTAÇÃO ?

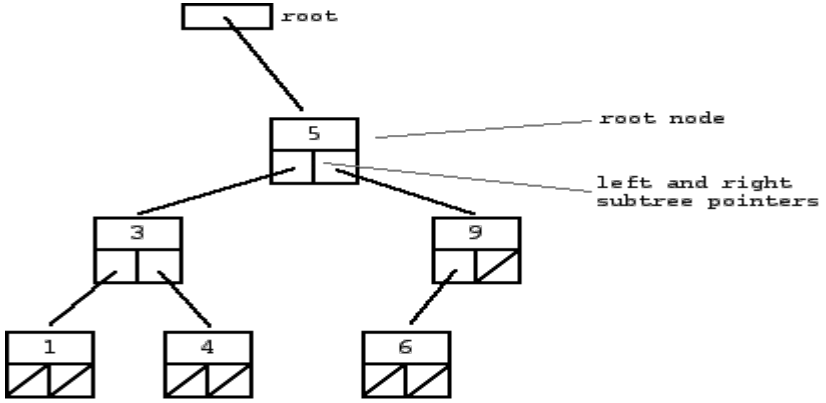
Lembrando de AED 1...

- Busca binária (em um vetor):

  -4, 2, 3, 5, 19, 21, 25

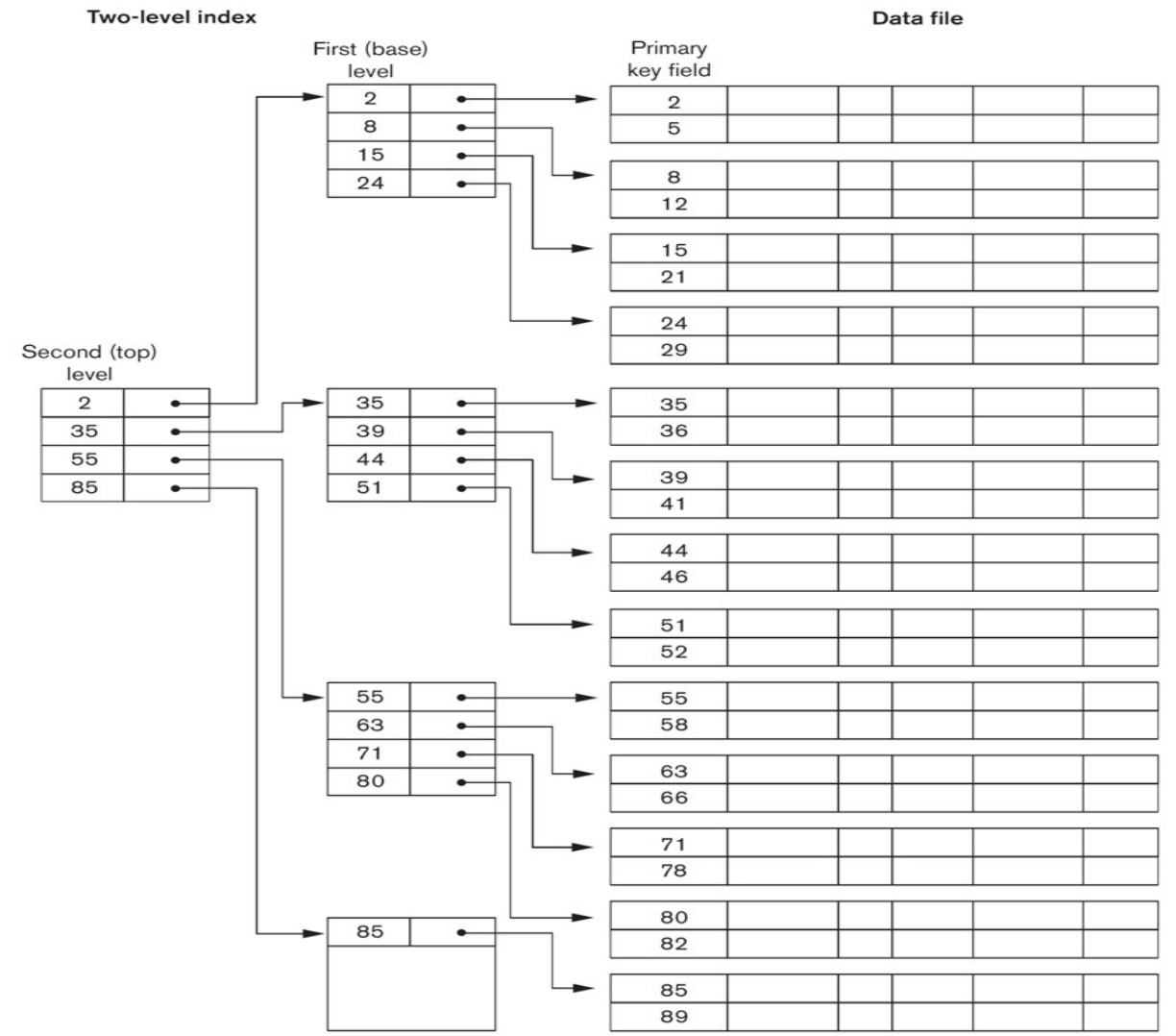
- Árvores Binárias de Busca:





Podemos pensar em algo semelhante para melhorar o dinamismo dos índices múltiníveis?

Figure 18.6
A two-level primary index resembling ISAM (Indexed Sequential Access Method) organization.



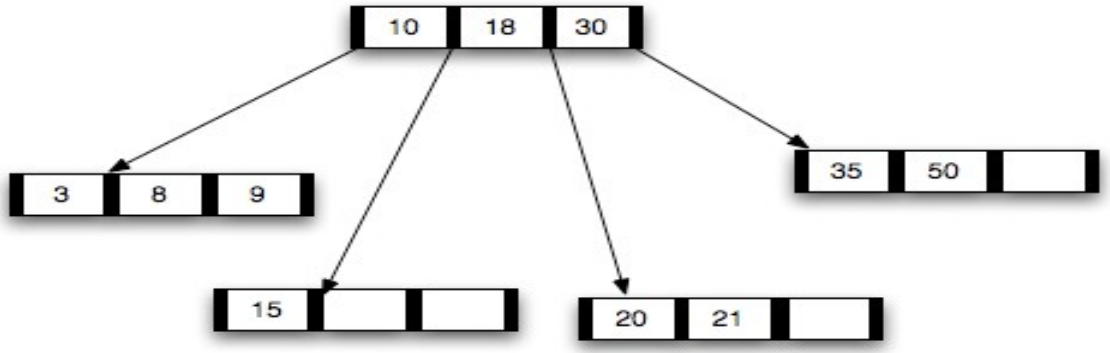
Podemos pensar em algo semelhante para melhorar o dinamismo dos índices múltiníveis?

- Árvores de busca n+1-árias!
- N = nr de registros representados em um nó da árvore (bloco), cada registro com uma chave k_i
- N+1 ponteiros para nós filhos contendo registros com chaves em cada intervalo

  O segredo será mantê-las balanceadas!

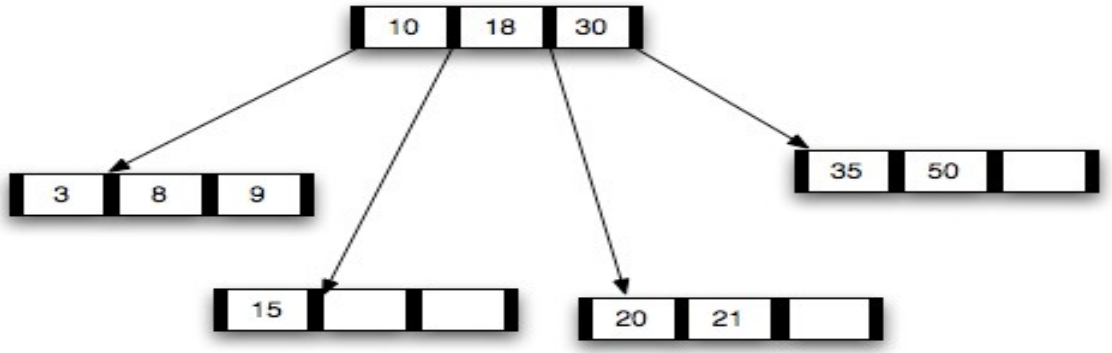
ÁRVORES B !!!

- Registros organizados pela árvore, assim como na árvore binária de busca
- Logo, se os registros possuem uma chave única, não há repetição de valores na árvore
- Abaixo é representada só a chave para simplificar a figura, mas na verdade deve conter, para cada chave k_i, o resto do registro (demais dados daquele item) ou um ponteiro p_i para o registro (k_i, p_i)



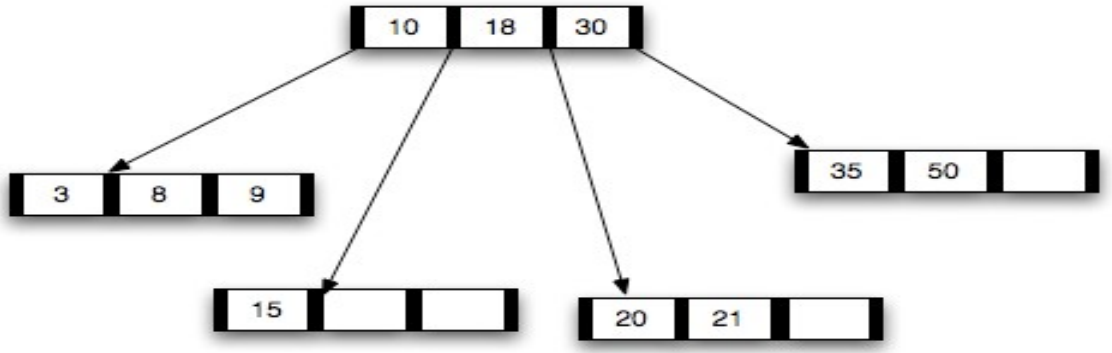
Árvore B Clássica

- Vamos começar estudando a árvore B clássica, depois fica fácil adaptar para a árvore B+



Árvore B Clássica

- Como deve ser a estrutura de dados para essa árvore?

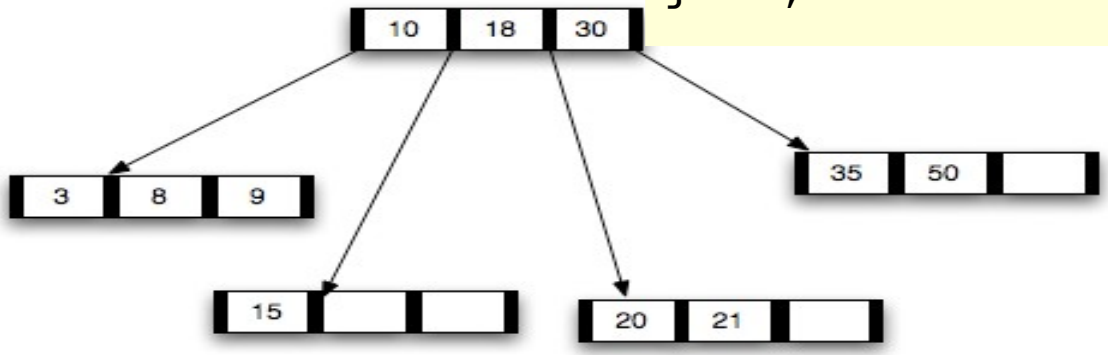


Árvore B Clássica

- Como deve ser a estrutura de dados para essa árvore?

Lembrando que, para simplificar, estamos colocando só a chave do registro

```
typedef  int  TipoChave;
typedef struct str_no {
    TipoChave chave[MAX_CHAVES];
    struct str_no* filho[MAX_CHAVES+1];
    int numChaves;
    bool folha;
} NO;
```

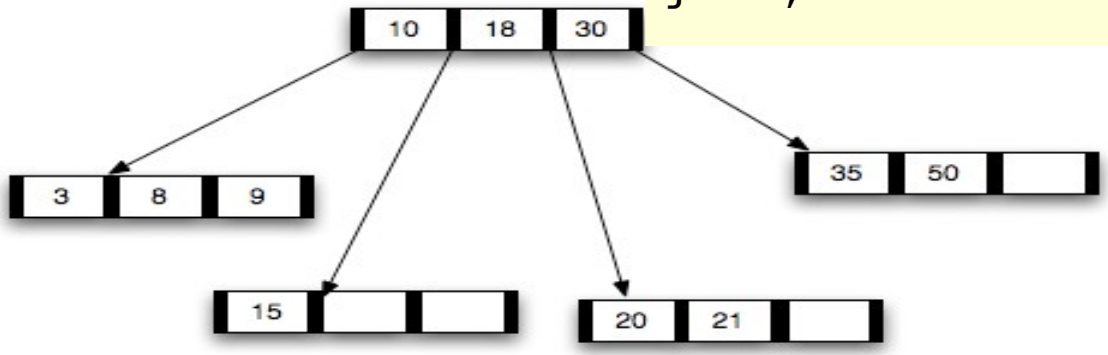


Árvore B Clássica

- Como deve ser a estrutura de dados para essa árvore?

PROVISÓRIO!!! →

```
typedef  int  TipoChave;
typedef struct str_no {
    TipoChave chave[MAX_CHAVES];
    struct str_no* filho[MAX_CHAVES+1];
    int numChaves;
    bool folha;
} NO;
```



Referências

- ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems**. 4 ed. Ed. Pearson-Addison Wesley. Cap 13 (até a seção 13.7).
- GOODRICH et al, **Data Structures and Algorithms in C++.** Ed. John Wiley & Sons, Inc. 2nd ed. 2011. Seção 14.2
- RAMAKRISHNAN, R.; GEHRKE, J. **Data Management Systems** 3ed. Ed McGraw Hill. 2003. cap 8 e 9.
- SILBERSCHATZ, A.; GALVIN, p. B.; GAGNE, G. **Operating Systems Concepts**. 8 ed. Ed. John Wiley $ Sons. 2009. Cap 11
- TANEMBAUM, A. S. & BOS, H. **Modern Operating Systems**. Pearson, 4th ed. 2015. Cap 4