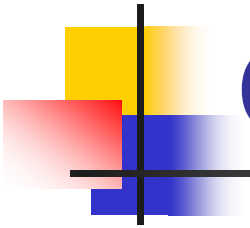
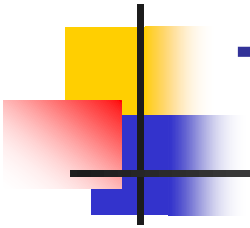


Compressão de dados

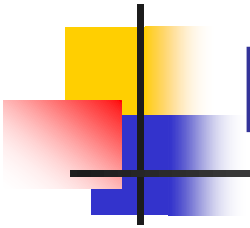
- A *compressão de dados* envolve a codificação da informação de modo que o arquivo ocupe menos espaço
  - Transmissão mais rápida
  - Processamento sequencial mais rápido
  - Menos espaço para armazenamento

- Algumas técnicas são gerais, e outras específicas para certos tipos de dados, como voz, imagem ou texto
  - Técnicas reversíveis vs. irreversíveis
  - A variedade de técnicas é enorme



Técnicas

- Notação diferenciada
  - Redução de redundância
- Omissão de sequências repetidas
  - Redução de redundância
- Códigos de tamanho variável
  - Código de Huffman



Notação diferenciada

- Exemplo
  - Códigos de estado (SP, MG, RJ, ...), armazenados na forma de texto: 2 bytes
    - Por exemplo, como existem 27 estados no Brasil, pode-se armazenar os estados em 5 bits
    - É possível guardar a informação em 1 byte e economizar 50% do espaço
  - Desvantagens?
    - Legibilidade, codificação/decodificação

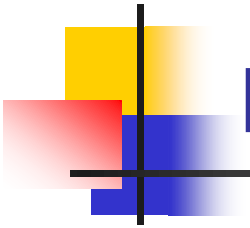
Omissão de sequências repetidas

Ex.: representação de imagens
- O fundo a imagem resulta em códigos repetidos

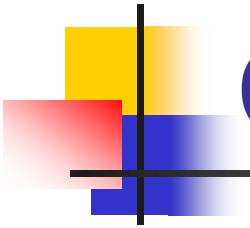
Omissão de sequências repetidas

- Para a sequência
  - 22 23 24 24 24 24 24 24 24 25 26 26 26 26 26 26 26 25 24
- Usando um código indicador de repetição (código de *run-length*)
  - 22 23 ff 24 07 25 ff 26 06 25 24

Omissão de sequências repetidas

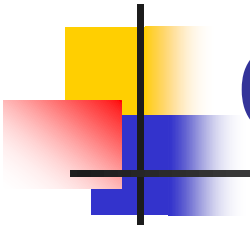


- Bom para dados esparsos ou com muita repetição
  - Imagens do céu, por exemplo
- Garante redução de espaço sempre?
  - Toda sequencia de repetições é substituída por 3 valores: código, valor repetido, número de repetições



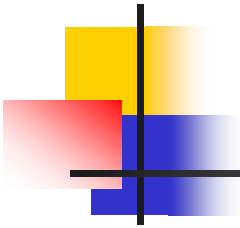
Códigos de tamanho fixo

- Código ASCII: 1 byte por caracter (fixo)
  - ‘A’ = 65 (8 bits)
  - Cadeia ‘ABC’ ocupa 3 bytes
  - Ignora frequência dos caracteres



Códigos de tamanho variável

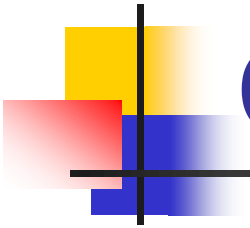
- Princípio
  - Alguns valores ocorrem mais frequentemente que outros, assim, seus códigos deveriam ocupar o menor espaço possível
- Código Morse
  - Letras mais frequentes – códigos menores
  - Distribuição de frequência conhecida
  - Tabela fixa de códigos de tamanho variável



Códigos de tamanho variável

- Código de Huffman
  - Código de tamanho variável
  - Distribuição de frequência desconhecida
  - Tabela de códigos construída dinamicamente

- Se letras que ocorrem com maior frequência têm códigos menores, as cadeias tendem a ficar mais curtas, e o arquivo menor



Código de Huffman

- Muito usado para codificar texto
- Como estabelecer a relação entre frequência e código?

Código de Huffman

- Exemplo

  Alfabeto: {A, B, C, D}
  Frequência: A > B > C = D
  Possível codificação:
  A=0, B=110, C=10, D=111
  Cadeia: ABACCDA
  Código: 0110010101110

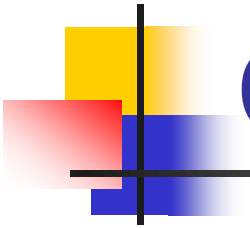
  É possível decodificar?

  Codificação deve ser não ambígua
  Ex. A=0, B=01, C=1
  ACBA → 01010
  É possível decodificar?

Código de Huffman

- Cada "prefixo" de um código identifica as possibilidades de codificação
  - Se primeiro bit é 0, então A; se 1, então B, C ou D, dependendo do próximo bit
  - Se segundo bit é 0, então C; se 1, então B ou D, dependendo do próximo bit
  - Se terceiro bit é 0, então B; se 1, então D
  - Quando o símbolo é determinado, começa-se novamente a partir do próximo bit

Símbolo	Código
A	0
B	110
C	10
D	111



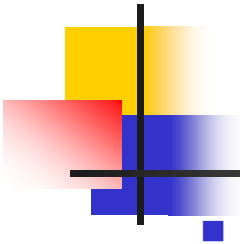
Código de Huffman

- Como representar todas as mensagens possíveis de serem expressas em língua portuguesa
  - Como determinar os códigos de cada letra/sílaba/palavra?
    - Qual a unidade do “alfabeto”?

Código de Huffman

- Código de Huffman
  - Calcula-se a frequência de cada símbolo do alfabeto
    1. Encontre os dois símbolos que têm menor frequência (B/1 e D/1)
    2. O último símbolo de seus códigos deve diferenciá-los (B=0 e D=1)
    3. Combinam-se esses símbolos e somam-se suas frequências (BD/2, indicando ocorrência de B ou D)
    4. Repete-se o processo até restar um símbolo
       1. C/2 e BD/2 ➔ 0 para C e 1 para BD ➔ CBD/4
       2. A/3 e CBD/4 ➔ 0 para A e 1 para CBD ➔ ACBD/7

ABACCDA ⟶ Freq.: A/3, B/1, C/2 e D1

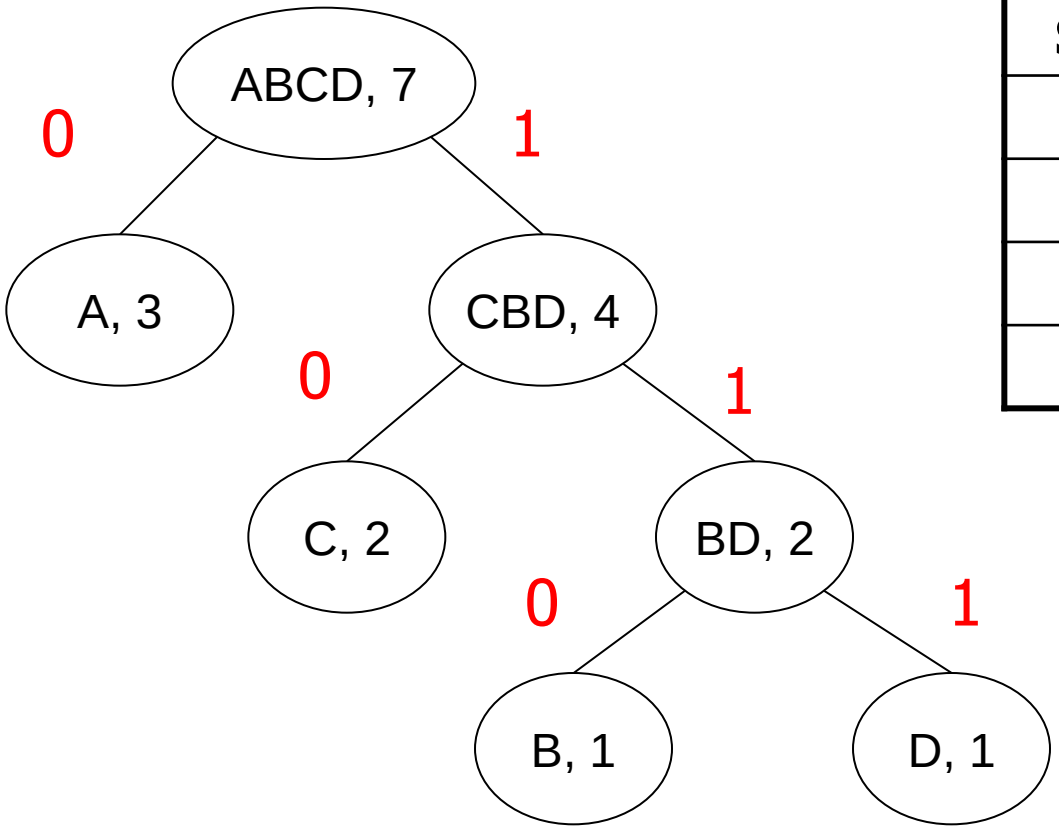


Código de Huffman

- Combinação de dois símbolos em 1
- Árvore de Huffman
  - Construída passo a passo após cada combinação de símbolos
  - Árvore binária
    - Cada nó da árvore representa um símbolo (e sua frequência)
    - Cada nó folha representa um símbolo do alfabeto original
  - Ao se percorrer a árvore da raiz até um símbolo (folha), o caminho fornece o código deste símbolo
    - Escalada por ramo esquerdo: 0 no início do código
    - Escalada por ramo direito: 1 no início do código

Código de Huffman

- Exemplo



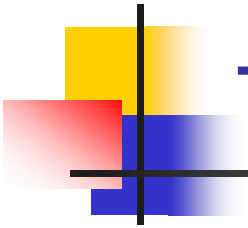
Símbolo	Código
A	0
B	110
C	10
D	111

Símbolos mais frequentes mais à esquerda e mais próximos da raiz

Código de Huffman

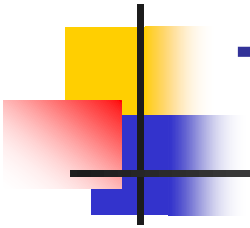
- Exercício: construir a árvore para os dados abaixo

Símbolo	Freqüência
A	15
B	6
C	7
D	12
E	25
F	4
G	6
H	1
I	15



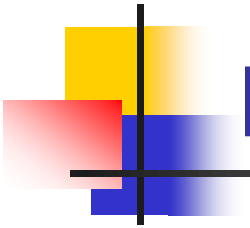
Técnicas de compressão irreversíveis

- Até agora, todas as técnicas eram reversíveis
- Algumas são irreversíveis
  - Por exemplo, salvar uma imagem de 400 por 400 pixels como 100 por 100 pixels
- Compressão de Fala – codificação de voz



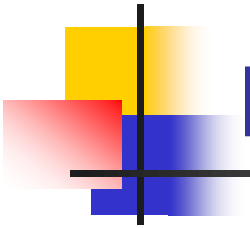
Técnicas de Compactação

- Compactação:
  - Diminuição do tamanho do arquivo eliminando espaços não utilizados, gerados após operações de eliminação de registros



Manipulação de dados em arquivos

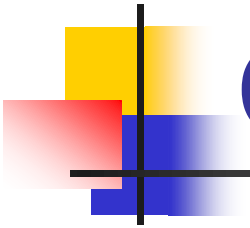
- Que operações básicas podemos fazer com os dados nos arquivos?



Manipulação de dados em arquivos

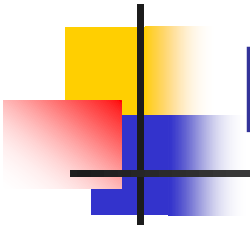
- Que operações básicas podemos fazer com os dados nos arquivos?
  - Adição de registros: relativamente simples
  - Eliminação de registros
  - Atualização de registros: eliminação e adição de um registro

    - O que pode acontecer com o arquivo?



Compactação

- **Compactação**
  - Busca por regiões do arquivo que não contêm dados
  - Posterior recuperação desses espaços perdidos
    - Os espaços vazios são provocados, por exemplo, pela eliminação de registros



Eliminação de registros

- Devem existir mecanismos que
  1. Permitam reconhecer áreas que foram apagadas
  2. Permitam recuperar e utilizar os espaços vazios

- Possibilidades?

Eliminação de registros

- Uso de Marcadores Especiais e Recuperação Esporádica
- Geralmente, áreas apagadas são marcadas com um marcador especial
- Eventualmente o procedimento de compactação é ativado, e o espaço de todos os registros marcados é recuperado de uma só vez
  - Maneira mais simples de compactar: executar um programa de cópia de arquivos que "pule" os registros apagados (se existe espaço suficiente para outro arquivo)
  - Ou compactação "in loco": mais complicada e demorada

Processo de compactação: **exemplo**

FIGURE 5.3 Storage requirements of sample file using 64-byte fixed-length records. (a) Before deleting the second record. (b) After deleting the second record. (c) After compaction—the second record is gone.

```
Ames|John|123 Maple|Stillwater|OK|74075|.....................
Morrison|Sebastian|9035 South Hillcrest|Forest Village|OK|74820|
Brown|Martha|625 Kimbark|Des Moines|IA|50311|.................
```

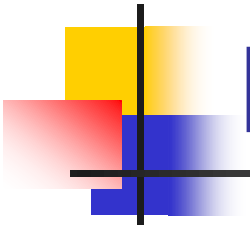
(a)

```
Ames|John|123 Maple|Stillwater|OK|74075|.....................
*|rrison|Sebastian|9035 South Hillcrest|Forest Village|OK|74820|
Brown|Martha|625 Kimbark|Des Moines|IA|50311|.................
```

(b)

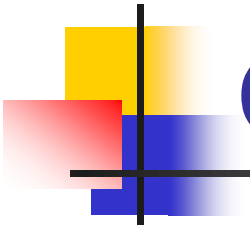
```
Ames|John|123 Maple|Stillwater|OK|74075|.....................
Brown|Martha|625 Kimbark|Des Moines|IA|50311|.................
```

(c)



Recuperação dinâmica

- Muitas vezes, o procedimento de compactação é esporádico
  - Um registro apagado não fica disponível para uso imediatamente
- Em aplicações interativas que acessam arquivos altamente voláteis, pode ser necessário um processo de recuperação de espaço, tão logo ele seja criado (dinâmico)
  - Marcar registros apagados
  - Na inserção de um novo registro, identificar espaço deixado por eliminações anteriores, sem buscas exaustivas
- Requisitos para reaproveitar espaço em novas inserções:
  - Reconhecer espaço disponível no arquivo
  - Pular diretamente para esse espaço, se existir



Como localizar os espaços vazios?

- Registros de tamanho fixo (possuem RRN)
  - Encadear os registros eliminados no próprio arquivo, formando uma Lista de Espaços Disponíveis (LD)
    - LD constitui-se de espaços vagos, endereçados por meio de seus RRNs
    - Cabeça da lista está no *header* do arquivo
    - Um registro eliminado contém a marca de eliminado e o RRN do próximo registro eliminado (que serve como ponteiro para o próximo elemento desta lista)
    - Inserção e remoção ocorrem sempre no início da LD (pilha) Por que?

Registros de tamanho fixo

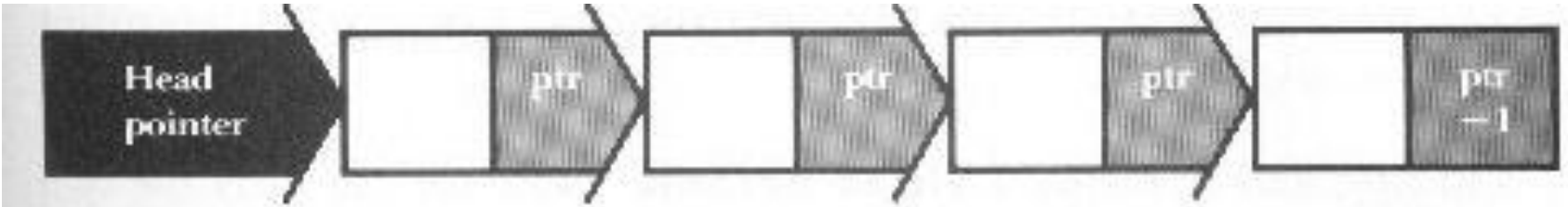
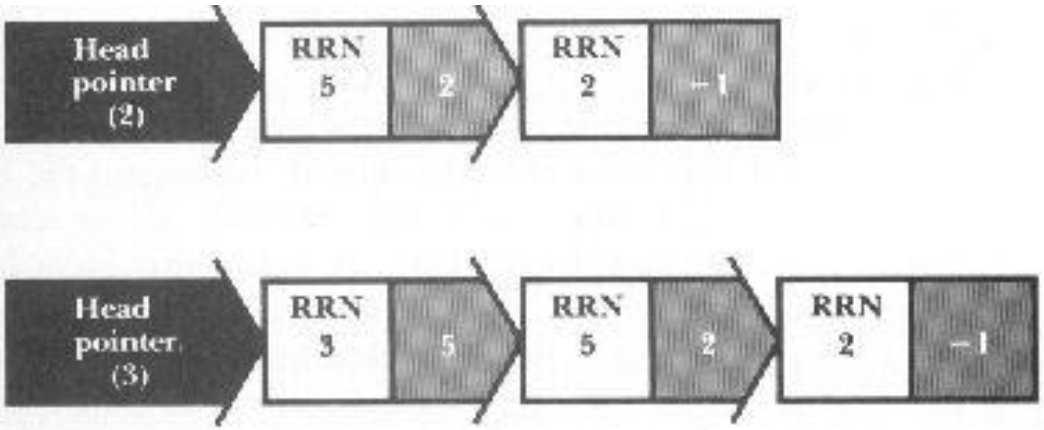


FIGURE 5.4 A linked list.



Pilha LD antes e depois da inserção do nó correspondente ao registro de RRN 3

Exemplo

List head (first available record) → 5

0	1	2	3	4	5	6
Edwards . . .	Bates . . .	Wills . . .	*−1	Masters . . .	*3	Chavez . . .

(a)

List head (first available record) → 1

0	1	2	3	4	5	6
Edwards . . .	*5	Wills . . .	*−1	Masters . . .	*3	Chavez . . .

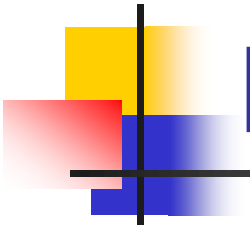
(b)

List head (first available record) → −1

0	1	2	3	4	5	6
Edwards . . .	*1st new rec*	Wills . . .	*3rd new rec*	Masters . . .	*2nd new rec*	Chavez . . .

(c)

FIGURE 5.5 Sample file showing linked lists of deleted records. (a) After deletion of records 3 and 5, in that order. (b) After deletion of records 3, 5, and 1, in that order. (c) After insertion of three new records.



Registros de tamanho variável

- Supondo arquivos com indicação do número de bytes antes de cada registro
- Marcação dos registros eliminados via um marcador especial
- Lista LD... mas não dá para usar RRNs
  - Tem que se usar a posição de início no arquivo

Eliminação de registros

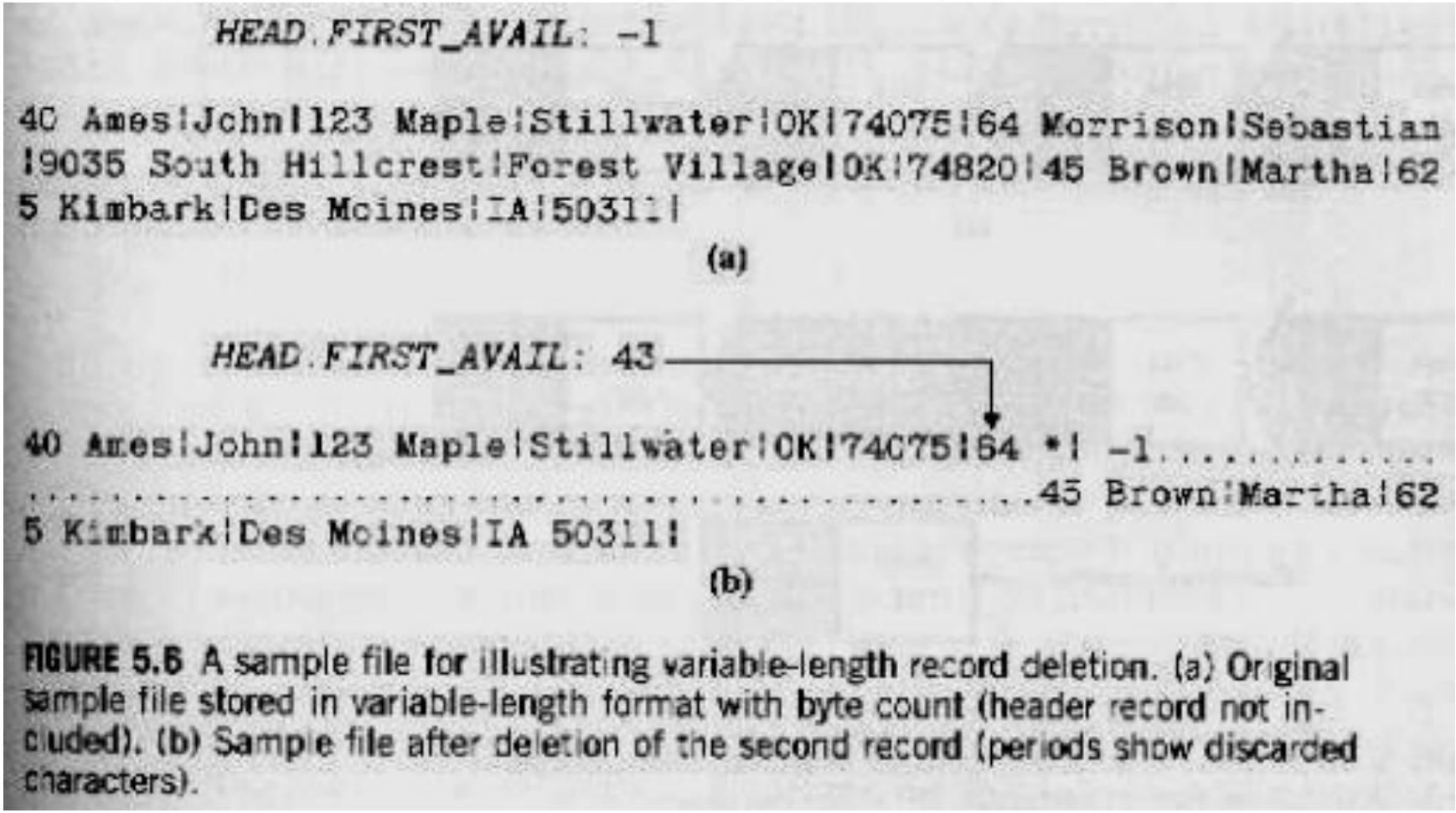
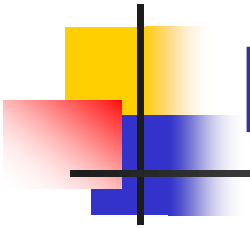


FIGURE 5.6 A sample file for illustrating variable-length record deletion. (a) Original sample file stored in variable-length format with byte count (header record not included). (b) Sample file after deletion of the second record (periods show discarded characters).



Registros de tamanho variável

- Para recuperar registros, não é possível usar uma pilha
  - É necessária uma busca sequencial na LD para encontrar uma posição com espaço suficiente

Adição de um registro de 55 bytes: exemplo

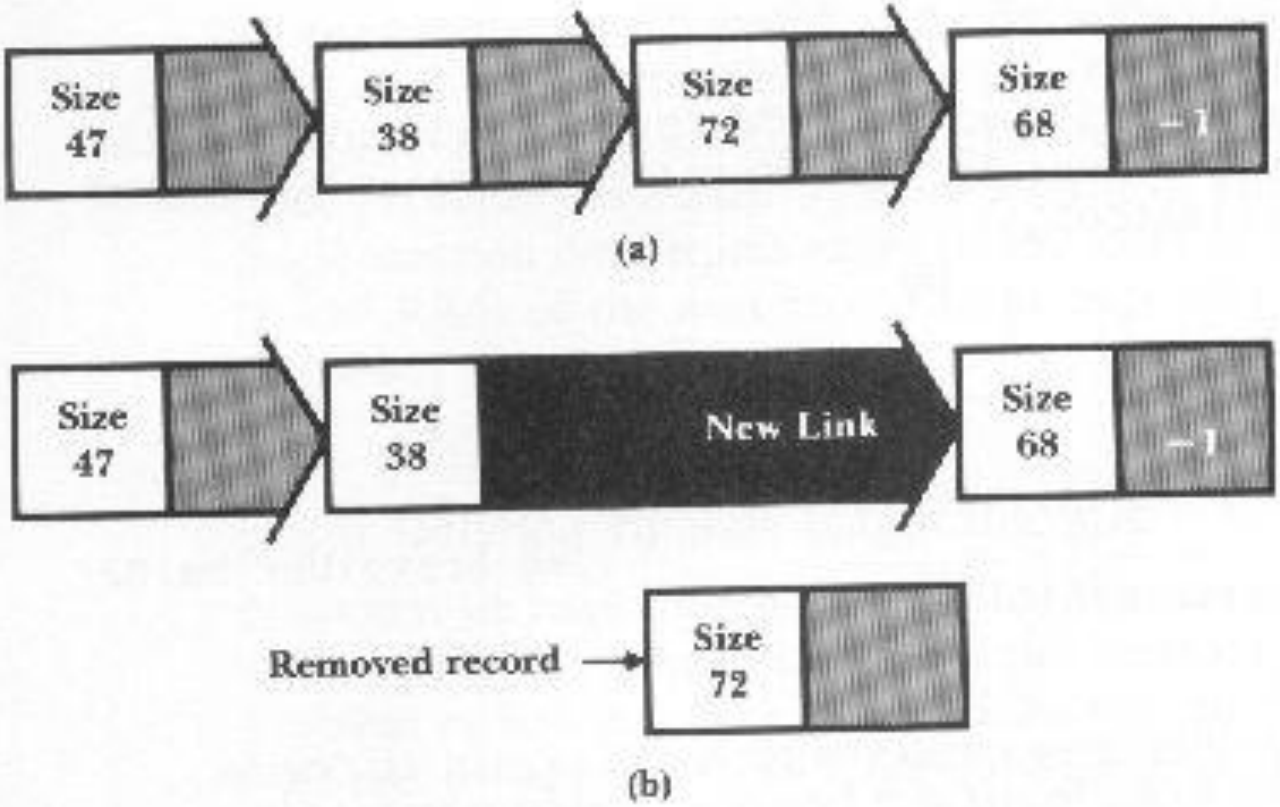
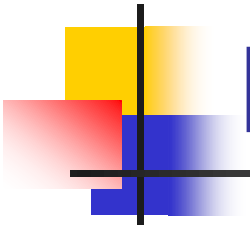
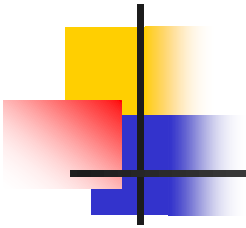


FIGURE 5.7 Removal of a record from an avail list with variable-length records. (a) Before removal. (b) After removal.



Registros de tamanho variável

- Estratégias de alocação de espaço
  - *First-fit*: pega-se o primeiro que servir, como feito anteriormente
    - Desvantagem?

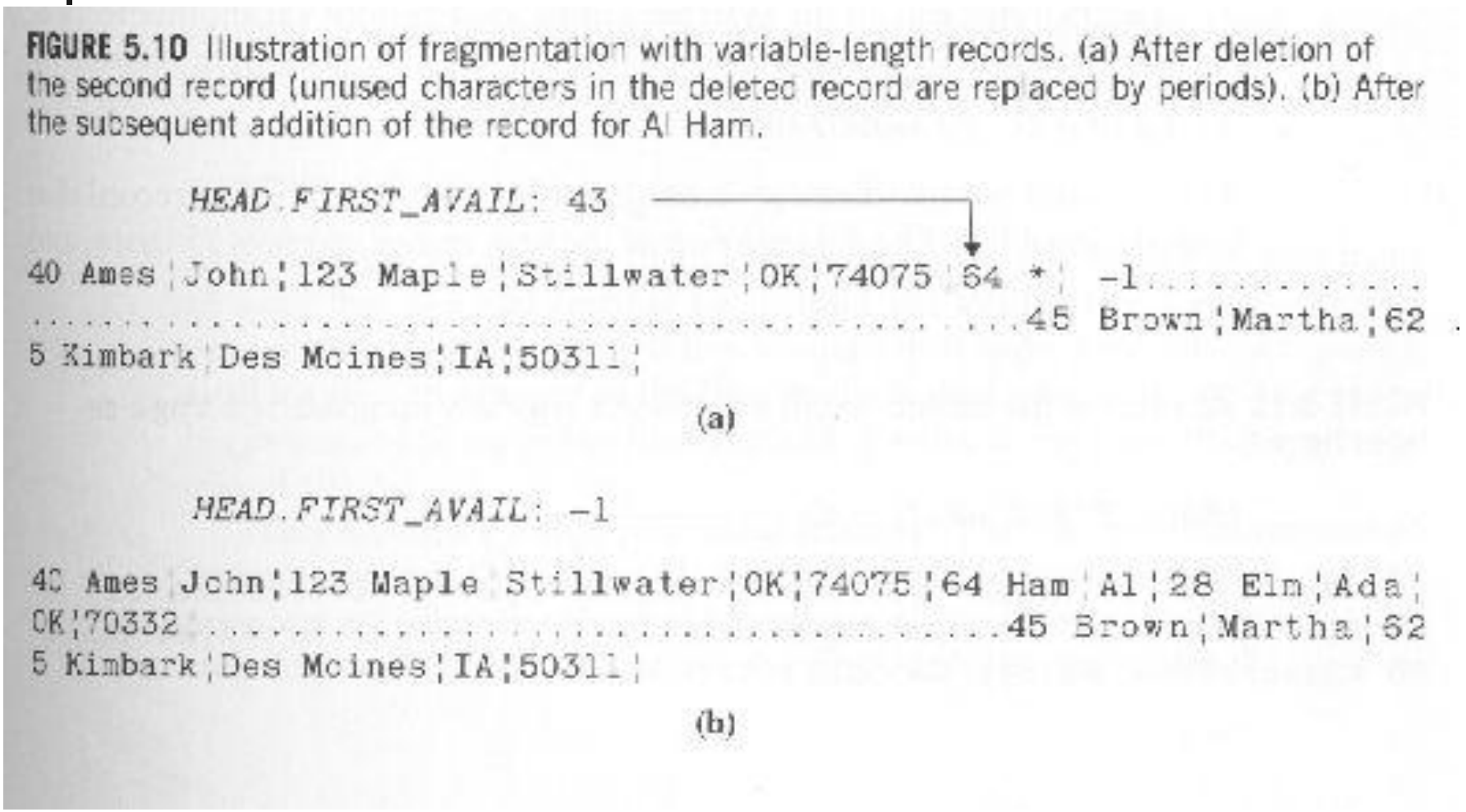


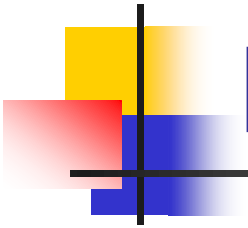
Registros de tamanho variável

- Estratégias de alocação de espaço
  - *First-fit*: pega-se o primeiro que servir, como feito anteriormente
    - Desvantagem?
      - Fragmentação interna

Fragmentação interna

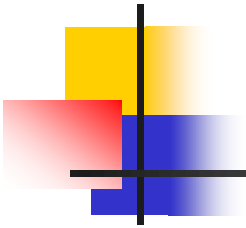
FIGURE 5.10 Illustration of fragmentation with variable-length records. (a) After deletion of the second record (unused characters in the deleted record are replaced by periods). (b) After the subsequent addition of the record for Al Ham.





Registros de tamanho variável

- Estratégias de alocação de espaço
  - *First-fit*: pega-se o primeiro que servir, como feito anteriormente
    - Solução para a Fragmentação Interna?



Registros de tamanho variável

- Estratégias de alocação de espaço
  - *First-fit*: pega-se o primeiro que servir, como feito anteriormente
    - Solução para a Fragmentação Interna?
      - Colocar o espaço que sobrou na lista de espaços disponíveis

Combatendo a fragmentação

- Solução: colocar o espaço que sobrou na lista LD
  - Parece uma boa estratégia, independentemente da forma que se escolhe o espaço

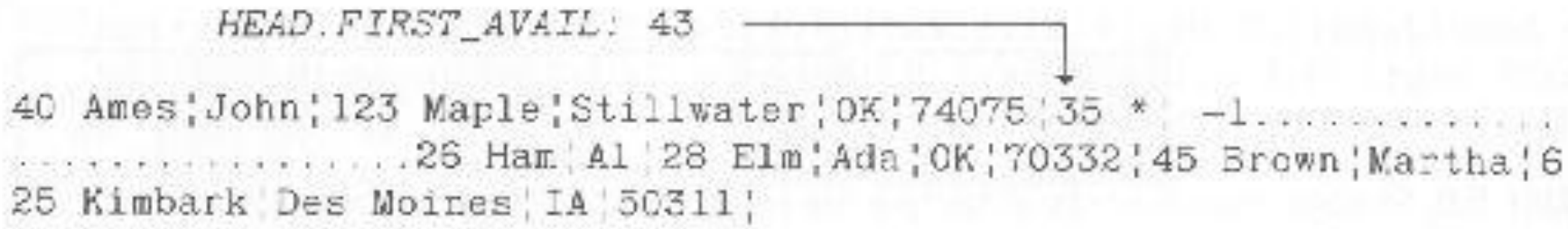
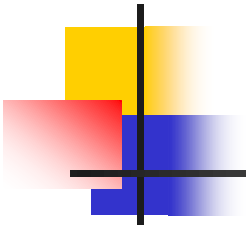
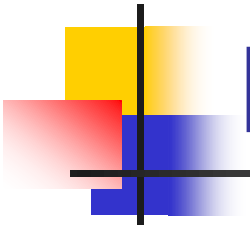


FIGURE 5.11 Combatting internal fragmentation by putting the unused part of the deleted slot back on the avail list.



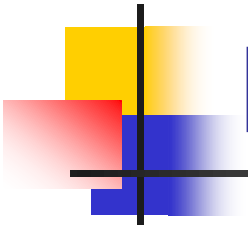
Registros de tamanho variável

- Alternativa: escolher o espaço mais justo possível
  - *Best-fit*: pega-se o mais justo
    - Desvantagem?



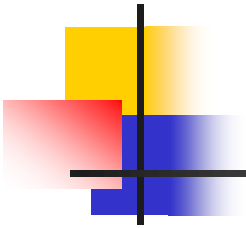
Registros de tamanho variável

- Alternativa: escolher o espaço mais justo possível
  - *Best-fit*: pega-se o mais justo
    - Desvantagem?
      - O espaço que sobra é tão pequeno que não dá para reutilizar
        - Fragmentação externa



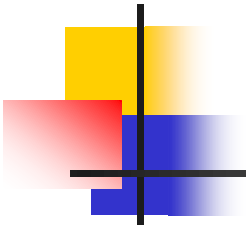
Registros de tamanho variável

- Alternativa: escolher o espaço mais justo possível
  - *Best-fit*: pega-se o mais justo
    - É conveniente organizar a lista LD de forma ascendente segundo o tamanho dos registros
    - O espaço mais "justo" é encontrado primeiro



Registros de tamanho variável

- Alternativa: escolher o maior espaço possível
  - *Worst-fit*: pega-se o maior
    - Diminui a fragmentação externa
    - Lista LD organizada de forma descendente
      - O processamento pode ser mais simples

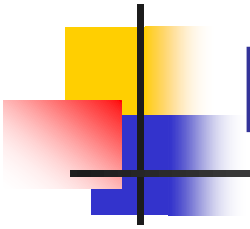


Registros de tamanho variável

- Outra forma de combater fragmentação externa
  - *"Coalescing the holes"* - Aglutinação, Fusão, Junção de espaços vazios adjacentes
    - Qual a dificuldade desta abordagem?

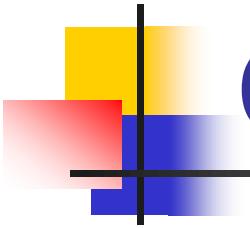
Registros de tamanho variável

- Outra forma de combater fragmentação externa
  - *"Coalescing the holes"* - Aglutinação, Fusão, Junção de espaços vazios adjacentes
    - Qual a dificuldade desta abordagem?
      - Registros eliminados que são adjacentes no arquivo não são necessariamente adjacentes na lista LD



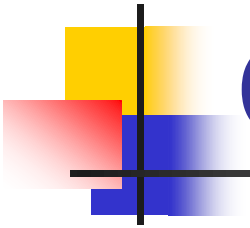
Registros de tamanho variável

- Outra forma de combater fragmentação externa
  - Gerar um novo arquivo, sem os registros "vazios"



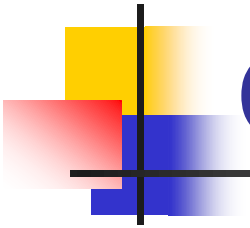
Observações

- Estratégias de alocação só fazem sentido com registros de tamanho variável
- Se espaço está sendo desperdiçado como resultado de fragmentação interna, então a escolha é entre *first-fit* e *best-fit*
  - A estratégia *worst-fit* piora esse problema
- Se o espaço está sendo desperdiçado devido à fragmentação externa, deve-se considerar a *worst-fit*



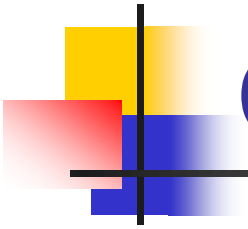
Ordenar para otimizar tempo

- Lembre-se: acessar memória externa custa muito!
  - Se um acesso acesso a RAM demorasse 20 segundos, o correspondente ao disco demoraria 58 dias!
- Ordenação torna a busca mais eficiente
  - Mas ordenar também envolve um custo
  - Se cada comparação envolver um seek, então a ordenação tb será proibitiva



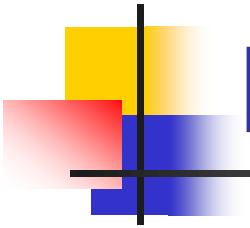
Ordenação e busca em arquivos

- É relativamente fácil buscar elementos em conjuntos ordenados
- A ordenação pode ajudar a diminuir o número de acessos a disco



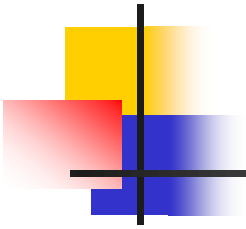
Ordenação e busca em arquivos

- Já vimos busca sequencial
  - O(n) → Muito ruim para acesso a disco!
  - É beneficiada por *buffering*
- E a busca binária?
  - Modo de funcionamento?
  - Complexidade de tempo?
    - Dominada pelo tempo (número) de *seeking*



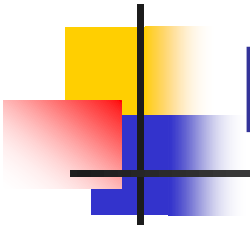
Busca binária

- Requisito: arquivo ordenado
- Dificuldade: aplicar um método de ordenação conhecido nos dados em arquivo
- Alternativa: ordenar os dados em RAM
  - Leitura sequencial, por setores (bom!)
  - Ainda é necessário: ler todo o arquivo e ter memória interna disponível



Busca binária

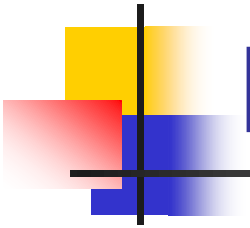
- Limitações
  - Registros de tamanho fixo para encontrar "o meio"
  - Manter um arquivo ordenado é muito caro
    - Custo pode superar os benefícios da busca binária
    - Inserções em *"batch mode"*: ordena e merge
  - Requer mais do que 1 ou 2 acessos
    - Por exemplo, em um arquivo com 1.000 registros, são necessários aproximadamente 10 acessos ($\sim \log_2 1000$)em média → ainda é ruim!



Busca binária

- Exercício
  - Implementar em C uma sub-rotina de busca binária em um arquivo ordenado por número USP

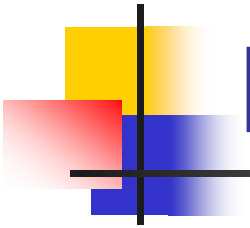
```
struct aluno {
        char nome[50];
        int nro_USP;
}
```



Problemas a solucionar

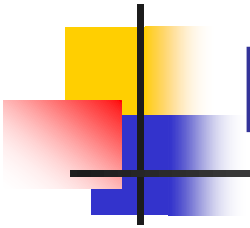
- Reordenação dos registros do arquivo a cada inserção

- Quando o arquivo não cabe na RAM



Problemas a solucionar

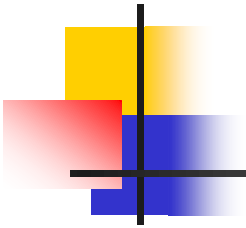
- Reordenação dos registros do arquivo a cada inserção
  - Uso de índices e/ou hashing
  - Outras estruturas de dados (árvores)
- Quando o arquivo não cabe na RAM
  - *Keysort* – variação de ordenação interna
  - Alternativa para não reordenar o arquivo



Keysorting

- Ordenação por chaves

- Ideia básica
  - Não é necessário que se armazenem todos os dados na memória principal para se conseguir a ordenação
  - Basta que se armazenem as chaves



Keysorting

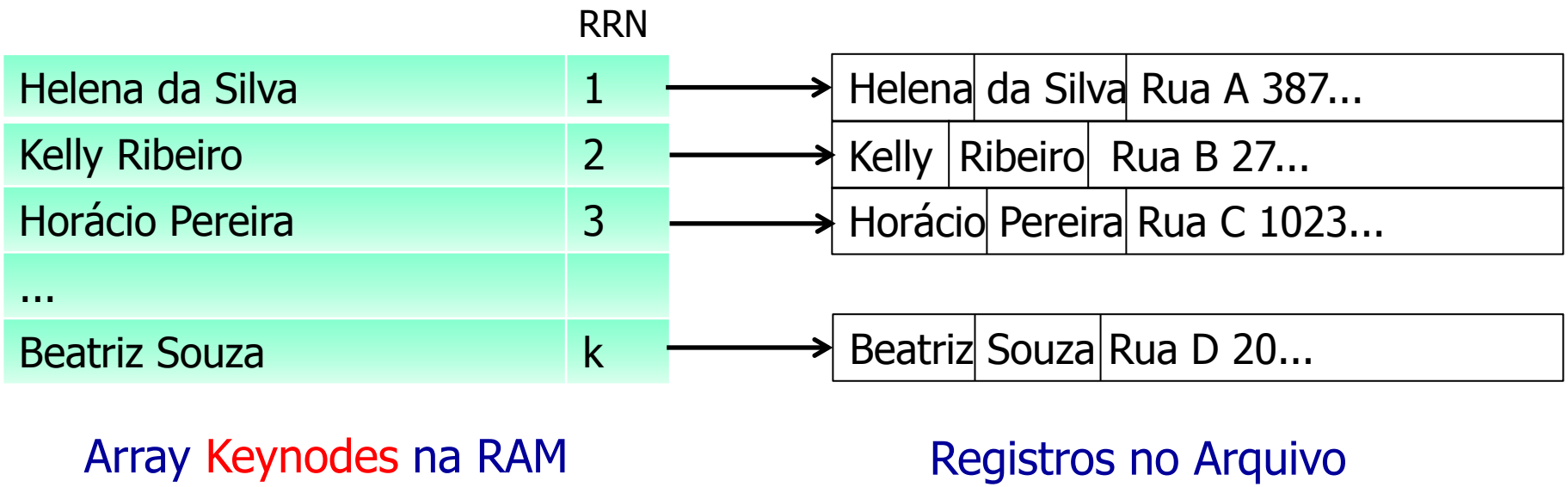
- Método

1. Cria-se na memória interna um vetor, em que cada posição tem uma chave do arquivo e um ponteiro para o respectivo registro no arquivo (RRN, se reg. tamanho fixo, ou byte inicial, se tam. variável)

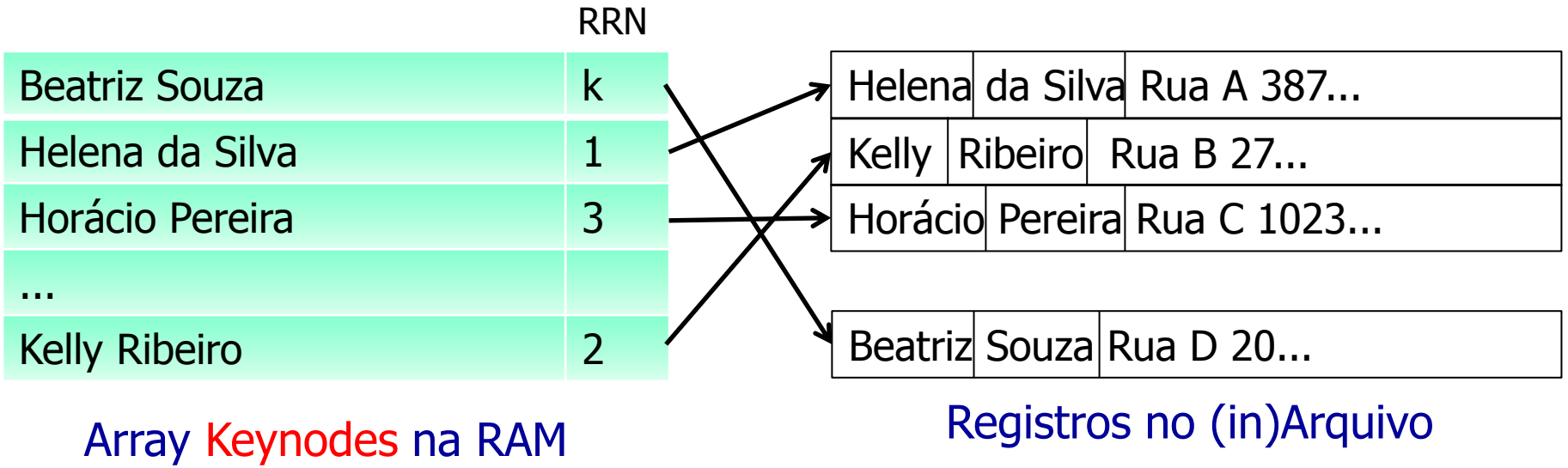
2. Ordena-se o vetor na memória interna

3. Cria-se um novo arquivo com os registros na ordem em que aparecem no vetor ordenado na memória principal

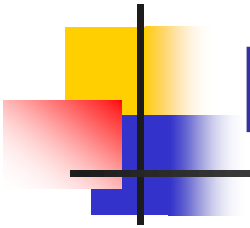
Keysorting – antes de ordenar



Keysorting – após ordenar Keynodes

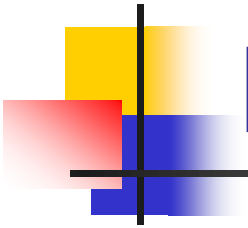


para i = 1 até N (número de registros)
    seek o arquivo até o registro cujo RRN é Keynodes[i].RRN
    leia esse registro (in-buffer) na RAM
    escreva esse registro (out-buffer) no arquivo de saída



Keysorting

- Limitações
  - Inicialmente, é necessário ler as chaves de todos os registros no arquivo
  - Depois, para se criar o novo arquivo, devem-se fazer vários *seeks* no arquivo para cada posição indicada no vetor ordenado
    - Mais uma leitura completa do arquivo
    - Não é uma leitura sequencial
    - Alterna-se leitura no arquivo antigo e escrita no arquivo novo



Keysorting

- Questões
  - Por que criar um novo arquivo?
    - Temos que ler todos os registros (e não sequencialmente!) para reescrevê-los no novo arquivo!!
  - Não vale a pena usar o vetor ordenado como um índice?

Usar Keynodes como Arquivo de Índices