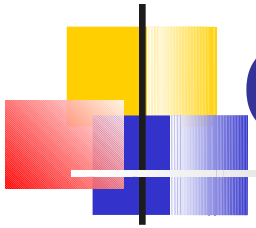


Organização de Arquivos

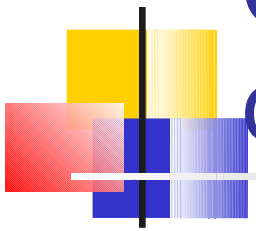
SCC-503 Algoritmos e Estruturas de Dados II

Thiago A. S. Pardo

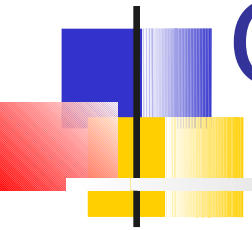
Leandro C. Cintra

M.C.F. de Oliveira

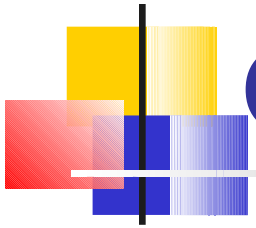
Organização de arquivos para desempenho



- Organização de arquivos visando desempenho
  - Complexidade de espaço
    - Compressão e compactação de dados
    - Reuso de espaço
  - Complexidade de tempo
    - Ordenação e busca de dados



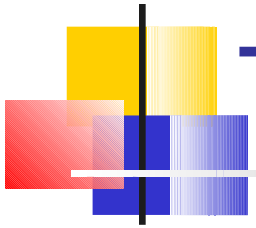
Compressão



Compressão de dados

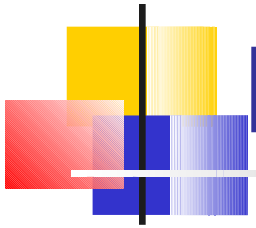
- A *compressão de dados* envolve a codificação da informação de modo que o arquivo ocupe menos espaço
  - Transmissão mais rápida
  - Processamento seqüencial mais rápido
  - Menos espaço para armazenamento

- 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



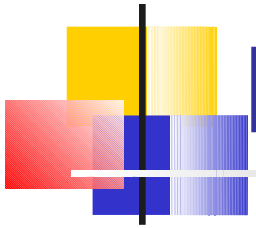
Técnicas

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



Notação diferenciada

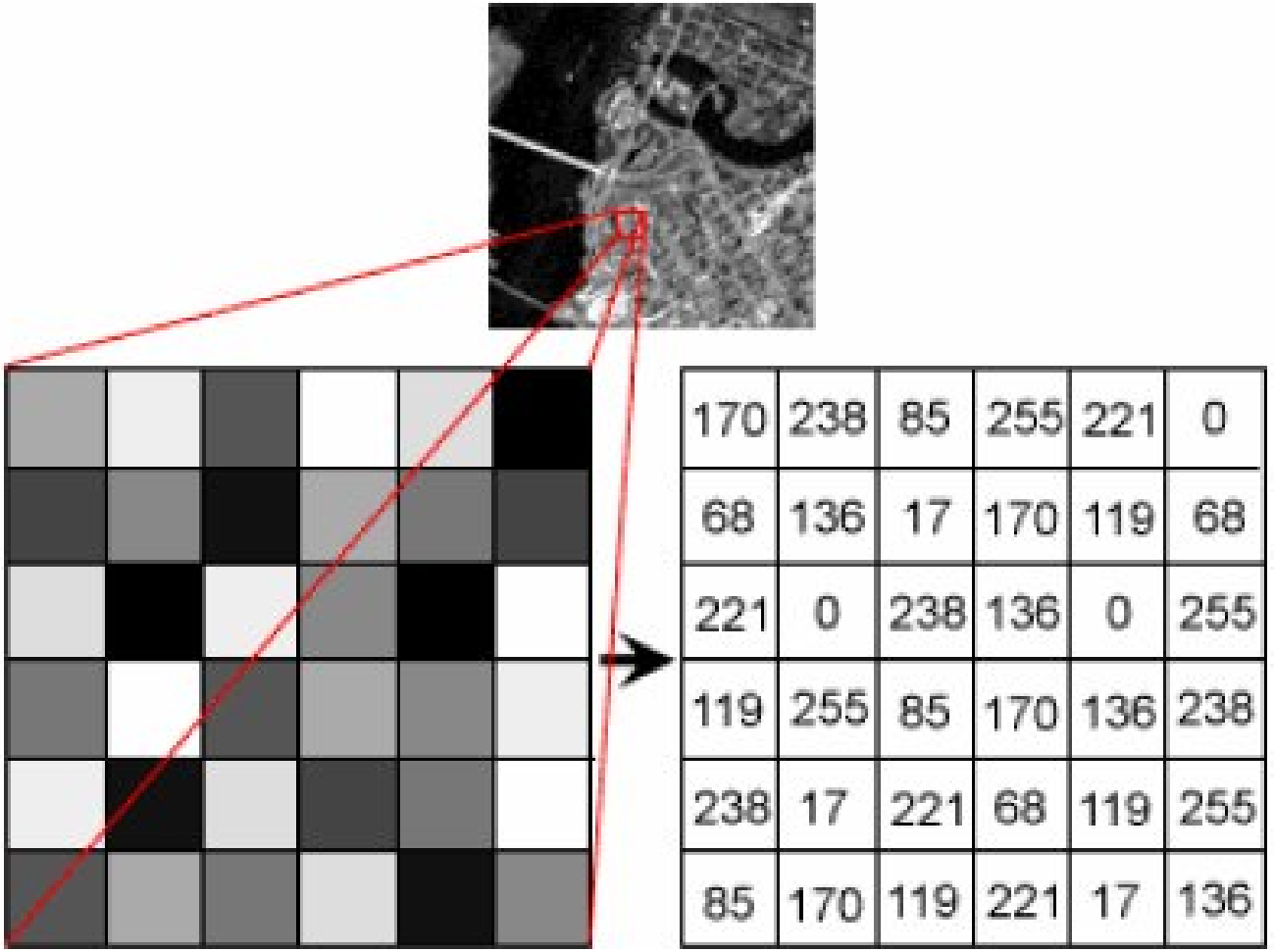
- Exemplo
  - Códigos de estado, armazenados na forma de texto: 2 bytes
    - 50 estados americanos
      - 2 bytes (para representação de 2 caracteres): NY, CA, etc.
    - Alternativa: com 50 opções, pode-se usar 6 bits
      - Por que?
        - É possível guardar a informação em 1 byte e economizar 50% do espaço
  - Desvantagens?



Notação diferenciada

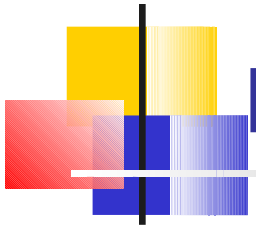
- Exemplo
  - Códigos de estado, armazenados na forma de texto: 2 bytes
    - 50 estados americanos
      - 2 bytes (para representação de 2 caracteres): NY, CA, etc.
    - Alternativa: com 50 opções, pode-se usar 6 bits
      - Por que?
        - É possível guardar a informação em 1 byte e economizar 50% do espaço
  - Desvantagens?
    - Legibilidade, codificação/decodificação

Omissão de seqüências repetidas



170	238	85	255	221	0
68	136	17	170	119	68
221	0	238	136	0	255
119	255	85	170	136	238
238	17	221	68	119	255
85	170	119	221	17	136

Omissão de seqüências repetidas

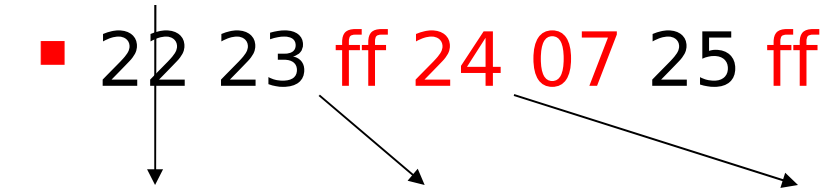


- Para a seqüência hexadecimal
  - 22 23 24 24 24 24 24 24 24 25 26 26 26 26 26 26 26 25 24
- Como melhorar isso?

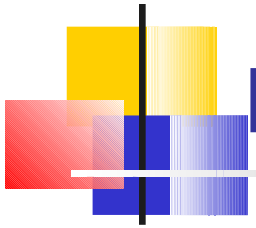
Omissão de seqüências repetidas

- Para a seqüência hexadecimal
  - 22 23 24 24 24 24 24 24 24 25 26 26 26 26 26 26 25 24

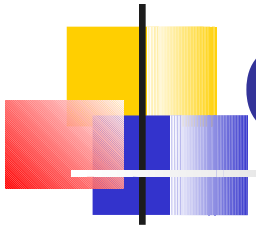
- Usando 0xff como código indicador de repetição (código de *run-length*)
  - 22 23 ff 24 07 25 ff 26 06 25 24



Omissão de seqüências repetidas



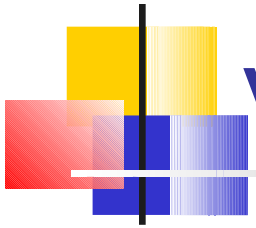
- Bom para dados esparsos ou com muita repetição
  - Imagens de astronomia e microscopia, por exemplo
- Garante redução de espaço sempre?
  - Há exceções?



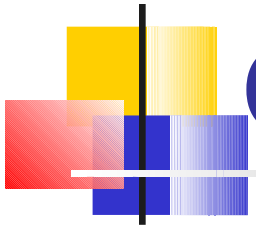
Questão

- No código ASCII: 1 byte por caractere (fixo)
  - ‘A’ = 65 (8 bits)
  - Cadeia ‘ABC’ ocupa 3 bytes (24 bits)
    - É possível melhorar?

Códigos de tamanho variável

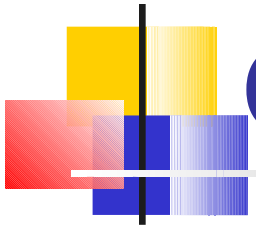


- Código de Huffman
  - Exemplo de código de tamanho variável
    - Idéia: valores mais freqüentes são associados a códigos menores
    - Exemplo de código desse tipo: código Morse



Código de Huffman

- Se letras que ocorrem com freqüência têm códigos menores, as cadeias tendem a ficar mais curtas
- Requer informação sobre a freqüência de ocorrência de cada símbolo a ser codificado
  - Muito usado para codificar texto



Código de Huffman

- Exemplo

Alfabeto: {A, B, C, D}

Freqüência: A > B > C = D

Possível codificação:

A=0, B=110, C=10, D=111

Cadeia: ABACCDA

Código: 0110010101110

Codificação não pode serambígua

Ex. A=0, B=01, C=1

ACBA → 01010

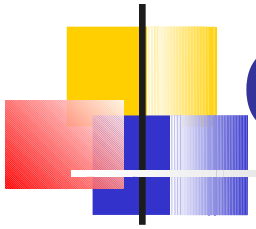
- É possível decodificar?

Código de Huffman

- Cada prefixo de um código identifica as possibilidades de codificação
  - Se primeiro bit é **0**, então A; se é **1**, então será 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

‘ABC’: quantos bits?



Código de Huffman

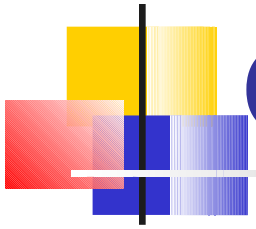
- Cada prefixo de um código identifica as possibilidades de codificação
  - Se primeiro bit é **0**, então A; se é **1**, então será B, C ou D, dependendo do próximo bit
  - Se segundo bit é **0**, então C; se **1**, então B ou D, dependendo do próximo bit
  - Se terceiro bit é **0**, então B; se **1**, 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

‘ABC’: quantos bits?

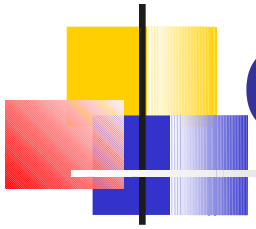
‘ABC’: 011010 (6 bits)

→ menos de 1 byte!



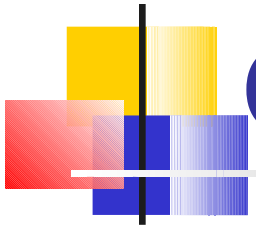
Código de Huffman

- 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”?
    - Quais as unidades mais freqüentes?



Código de Huffman

- Código de Huffman



Código de Huffman

- Código de Huffman
- Calcula-se a freqüência de cada símbolo do alfabeto

ABACCDA ⟶ Freqüência: A/3, B/1, C/2, D/1

Código de Huffman

- Código de Huffman
- Recupere os dois símbolos que têm menor freqüência (B/1 e D/1)

B
D

ABACCDA ⟶ Freqüência: A/3, C/2

Código de Huffman

- Código de Huffman
  1. Recupere os dois símbolos que têm menor freqüência (B/1 e D/1)
  2. Seus códigos devem diferenciá-los (B=0 e D=1)

B: 0
D: 1

ABACCDA ⟶ Freqüência: A/3, C/2

Código de Huffman

- Código de Huffman
  1. Recupere os dois símbolos que têm menor freqüência (B/1 e D/1)
  2. Seus códigos devem diferenciá-los (B=0 e D=1)
  3. Combinam-se esses símbolos e somam-se suas freqüências (BD/2, indicando ocorrência de B ou D)

B: 0
D: 1

ABACCDA ⟶ Freqüência: A/3, C/2, BD/2

Código de Huffman

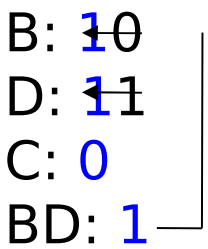
- Código de Huffman
  1. Recupere os dois símbolos que têm menor freqüência (C/2 e BD/2)

B: 0
D: 1
C
BD

ABACCDA ⟶ Freqüência: A/3

Código de Huffman

- Código de Huffman
  1. Recupere os dois símbolos que têm menor freqüência (C/2 e BD/2)
  2. Seus códigos devem diferenciá-los (C=0 e BD=1)



ABACCDA ⟶ Freqüência: A/3

Código de Huffman

- Código de Huffman
  1. Recupere os dois símbolos que têm menor freqüência (C/2 e BD/2)
  2. Seus códigos devem diferenciá-los (C=0 e BD=1)
  3. Combinam-se esses símbolos e somam-se suas freqüências (CBD/4)

B: 10
D: 11
C: 0
BD: 1

ABACCDA ⟶ Freqüência: A/3, CBD/4

Código de Huffman

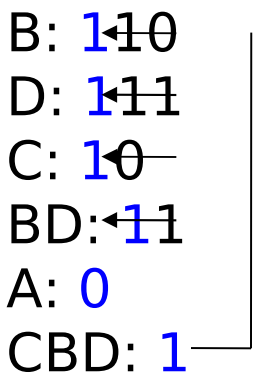
- Código de Huffman
  1. Recupere os dois símbolos que têm menor freqüência (A/3 e CBD/4)

B: 10
D: 11
C: 0
BD: 1
A
CBD

ABACCDA ⟶ Freqüência: ...

Código de Huffman

- Código de Huffman
  1. Recupere os dois símbolos que têm menor freqüência (A/3 e CBD/4)
  2. Seus códigos devem diferenciá-los (A=0 e CBD=1)



ABACCDA ⟶ Freqüência: ...

Código de Huffman

- Código de Huffman
  1. Recupere os dois símbolos que têm menor freqüência (A/3 e CBD/4)
  2. Seus códigos devem diferenciá-los (A=0 e CBD=1)
  3. Combinam-se esses símbolos e somam-se suas freqüências (ACBD/7)

B: 110
D: 111
C: 10
BD: 11
A: 0
CBD: 1

ABACCDA ⟶ Freqüência: ABCD/7

Código de Huffman

- Código de Huffman
  1. Encerra-se o processo, pois há somente um símbolo

B: 110
D: 111
C: 10
BD: 11
A: 0
CBD: 1

ABACCDA ⟶ Freqüência: ABCD/7

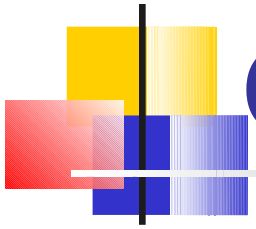
Código de Huffman

- Código de Huffman
  1. Encerra-se o processo, pois há somente um símbolo

B: 110
D: 111
C: 10
BD: 11
A: 0
CBD: 1

Símbolo	Código
A	0
B	110
C	10
D	111

ABACCDA ⟶ Freqüência: ABCD/7

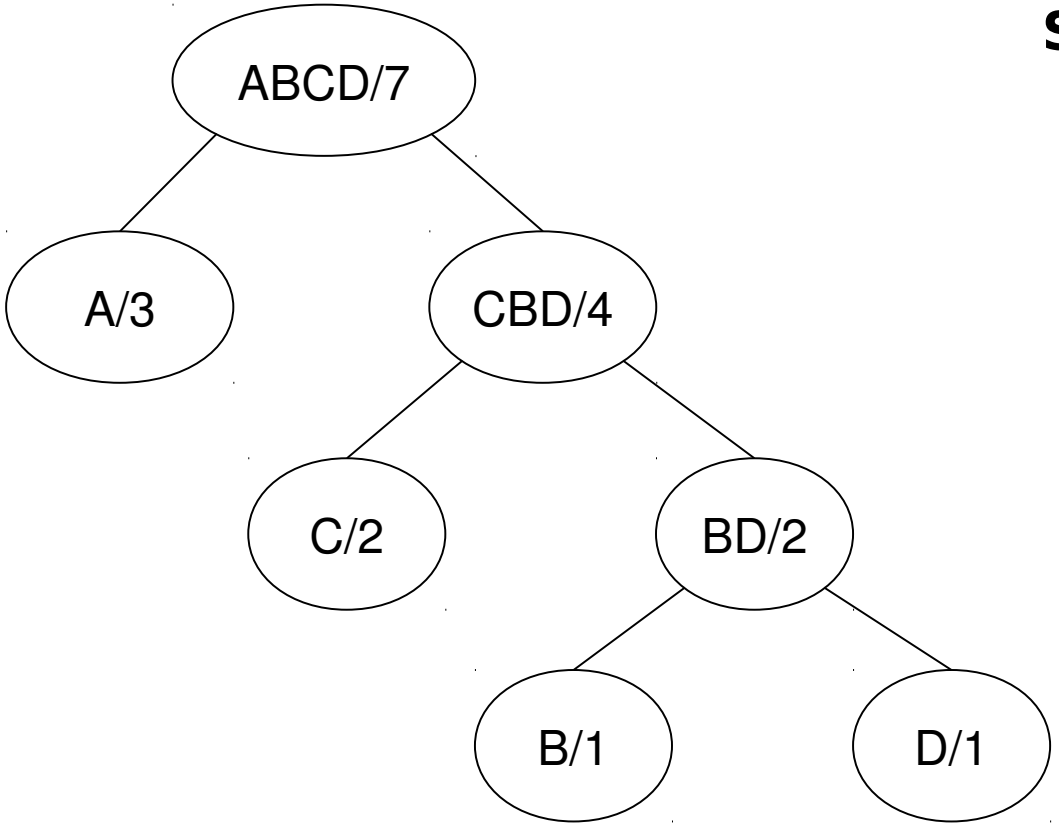


Código de Huffman

- Combinação de dois símbolos em 1
  - Árvore de Huffman é construída passo a passo após cada combinação de símbolos (árvore binária)
    - Cada nó da árvore representa um símbolo (e sua freqüência)
    - Cada nó folha representa um símbolo do alfabeto original
  - Ao se percorrer a árvore de uma folha X qualquer para a raiz, tem-se o código do símbolo X
    - Escalada por ramo esquerdo: 0 no início do código
    - Escalada por ramo direito: 1 no início do código

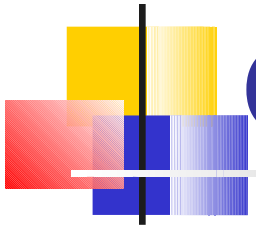
Código de Huffman

- Exemplo



Símbolo	Código
A	0
B	110
C	10
D	111

Símbolos mais freqüentes mais à esquerda e mais próximos da raiz



Código de Huffman

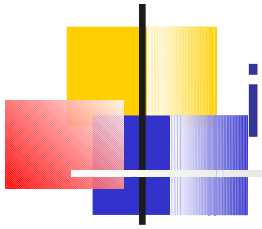
- Questão
  - Onde fica a tabela de códigos?

Código de Huffman

- Exercício em duplas: construir a árvore para os dados abaixo

Símbolo	Freqüência
A	15
B	6
C	7
D	12
E	25

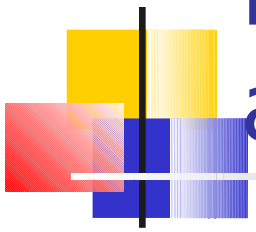
Técnicas de compressão irreversíveis



- Até agora, todas as técnicas eram reversíveis
- Algumas são irreversíveis (também chamadas de compressão com perdas)
  - Por exemplo, salvar uma imagem de 400 por 400 pixels como 100 por 100 pixels
    - Trocam-se 16 pixels por 1
- Onde se usa isso?

Compactação e reuso de espaço

Manipulação de dados em arquivos

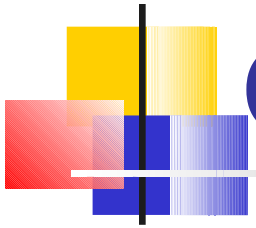


- Operações básicas que podemos fazer com os dados nos arquivos?

Manipulação de dados em arquivos

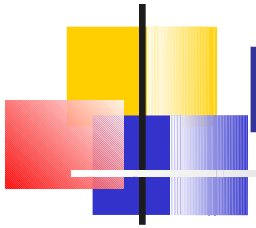
- Operações básicas que podemos fazer com os dados nos arquivos?
  - Adição de registros: relativamente simples
  - Eliminação de registros
  - Atualização de registros: eliminação e adição de um registro

    - O que pode acontecer com o arquivo?



Compactação

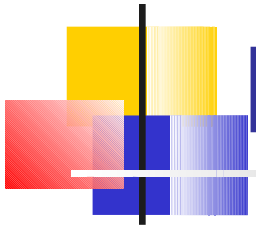
- **Compactação**
  - Busca por regiões do arquivo que não contêm dados
  - Posterior recuperação desses espaços perdidos
    - Os espaços vazios são provocados, por exemplo, pela eliminação de registros



Eliminação de registros

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

- Como fazer?



Eliminação de registros

- Geralmente, áreas apagadas são marcadas com um marcador especial
- Quando o procedimento de compactação é ativado, 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)

Processo de compactação: **exemplo**

Arquivo original

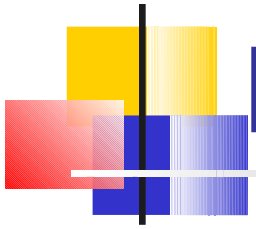
Maria|Rua 1|123|São Carlos|.....
João|Rua A|255|Rio Claro|.....
Pedro|Rua 10|56|Rib. Preto|.....

Após remover segundo registro

Maria|Rua 1|123|São Carlos|.....
*|ão|Rua A|255|Rio Claro|.....
Pedro|Rua 10|56|Rib. Preto|.....

Após compactação do arquivo

Maria|Rua 1|123|São Carlos|.....
Pedro|Rua 10|56|Rib. Preto|.....

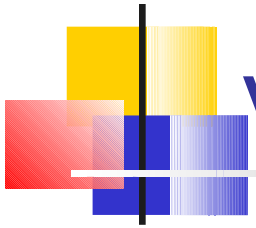


Recuperação dinâmica

- 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 dinâmico de recuperação de espaços vazios
  - Marcar registros apagados
  - Identificar e localizar os espaços antes ocupados por esses registros, sem buscas exaustivas

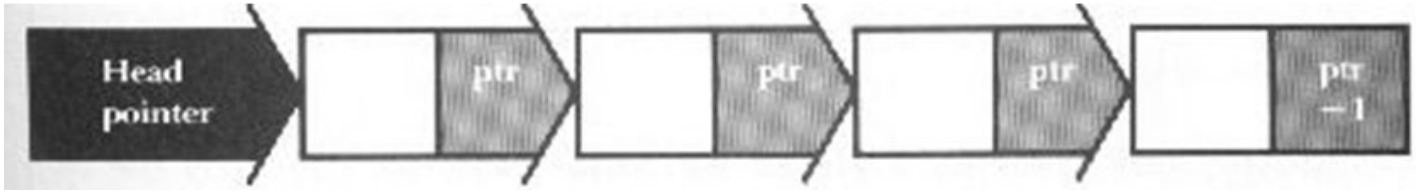
Como localizar os espaços vazios?



- Registros de tamanho fixo
  - Lista encadeada de registros eliminados no próprio arquivo
    - Lista constitui-se de espaços vagos, endereçados por meio de seus RRNs
    - Cabeça da lista está no *header* do arquivo
    - Um registro eliminado contém o RRN do próximo registro eliminado
    - Inserção e remoção ocorrem sempre no início da lista (pilha)

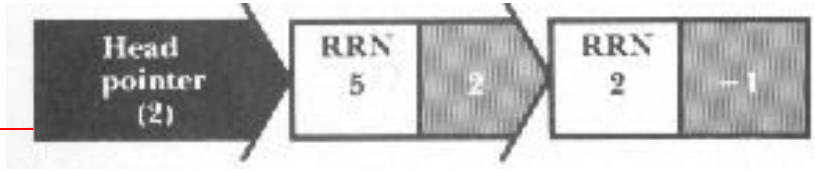
Registros de tamanho fixo

Lista encadeada: formato

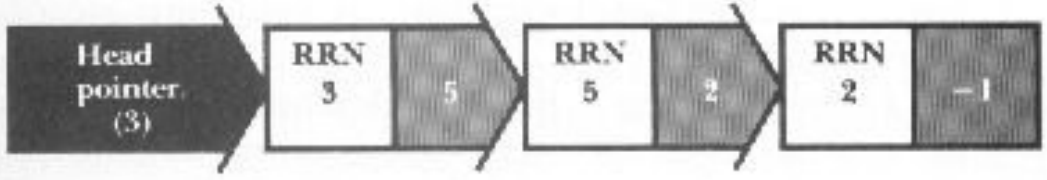


Pilha antes e depois da inserção do nó correspondente ao registro de RRN 3

Número de registros na lista



Remoção do registro de RRN 2
e depois o de RRN 5



Remove depois o de RRN 3

Exemplo

List head (first available record) → 5

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

Remoção de 3 e depois 5

List head (first available record) → 1

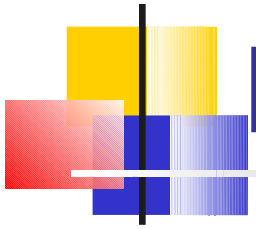
0	1	2	3	4	5	6
Edwards . . .	*5	Wills . . .	*−1	Masters . . .	*3	Chavez . . .

Remoção de 1

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 . . .

Adição de 3 registros

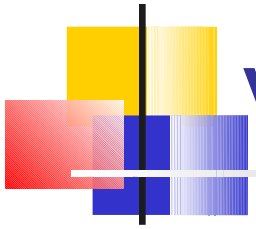


Registros de tamanho fixo

- Por que se usa uma pilha e não uma fila ou outra estrutura de dados?

- A pilha poderia ser mantida na memória principal?
  - Vantagens?
  - Desvantagens?

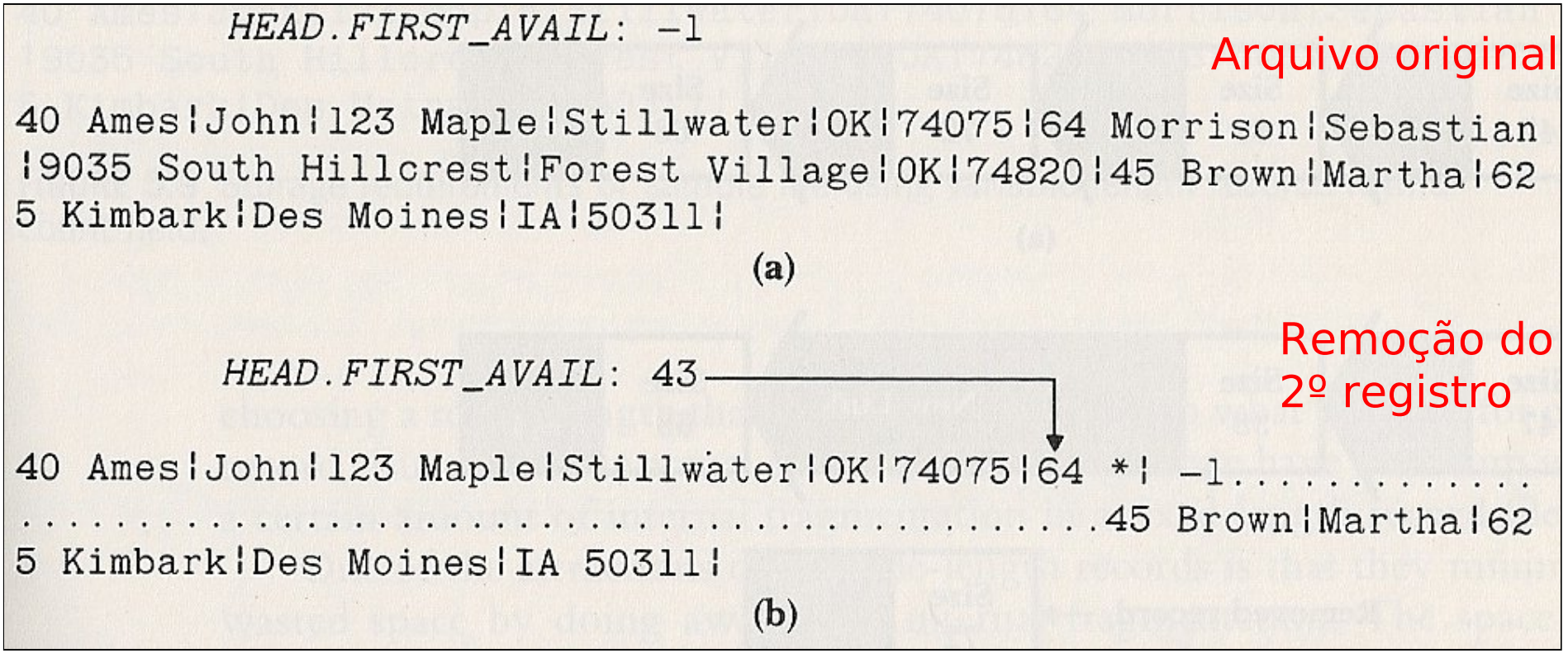
Registros de tamanho variável



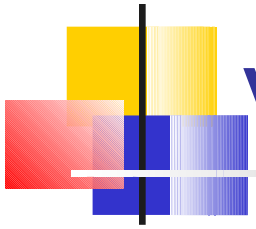
- Supondo arquivos com contagem de bytes antes de cada registro
- Marcação dos registros eliminados via um marcador especial
- Lista de registros eliminados... mas não dá para usar RRNs
  - Tem que se usar a posição de início no arquivo

Eliminação de registros

No registro de cabeçalho

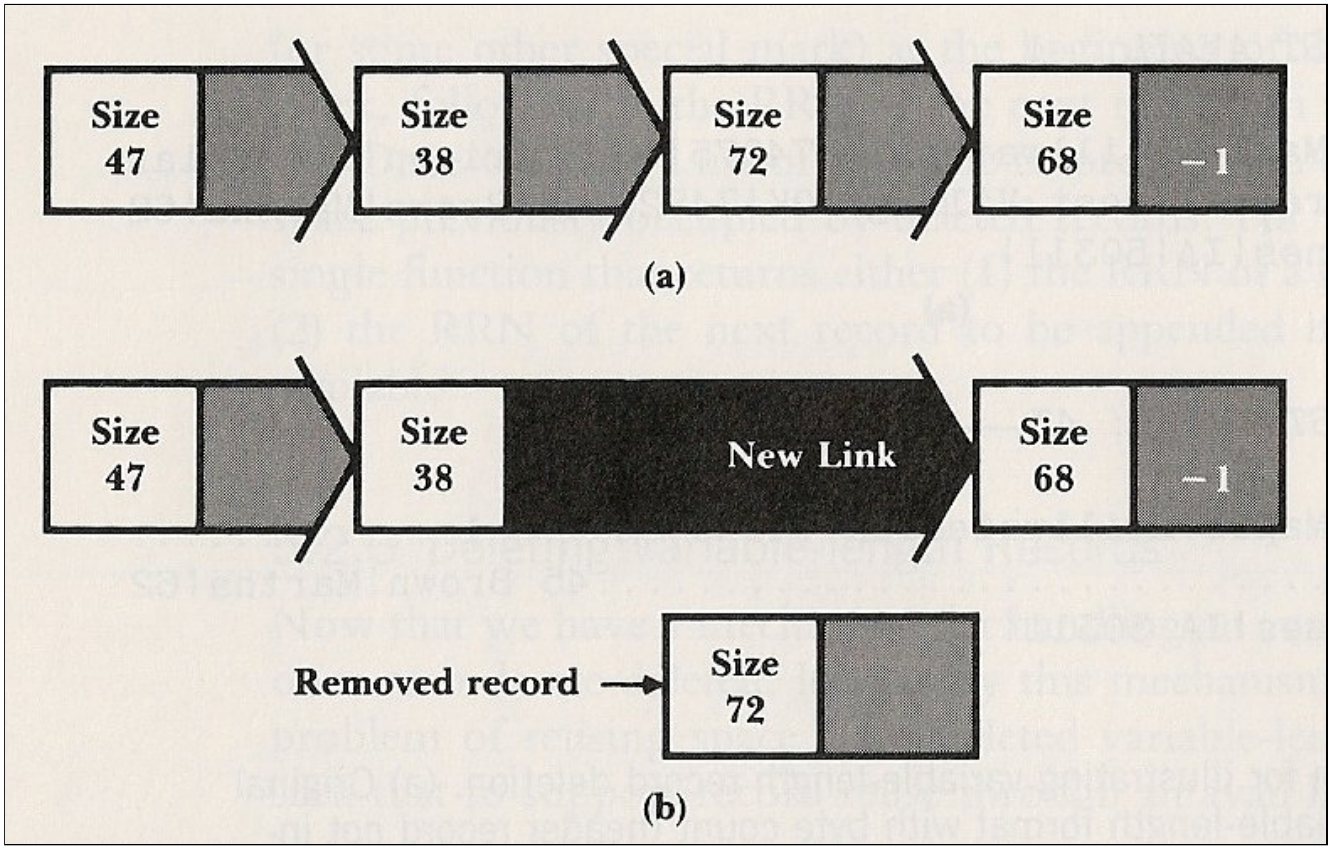


Registros de tamanho variável



- Para recuperar registros, não é possível usar uma pilha
  - É necessário uma busca seqüencial na lista para encontrar uma posição com espaço suficiente

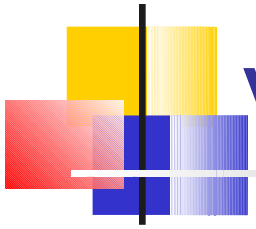
Adição de um registro de 55 bytes: exemplo



Antes da escolha

Depois da escolha

Registros de tamanho variável

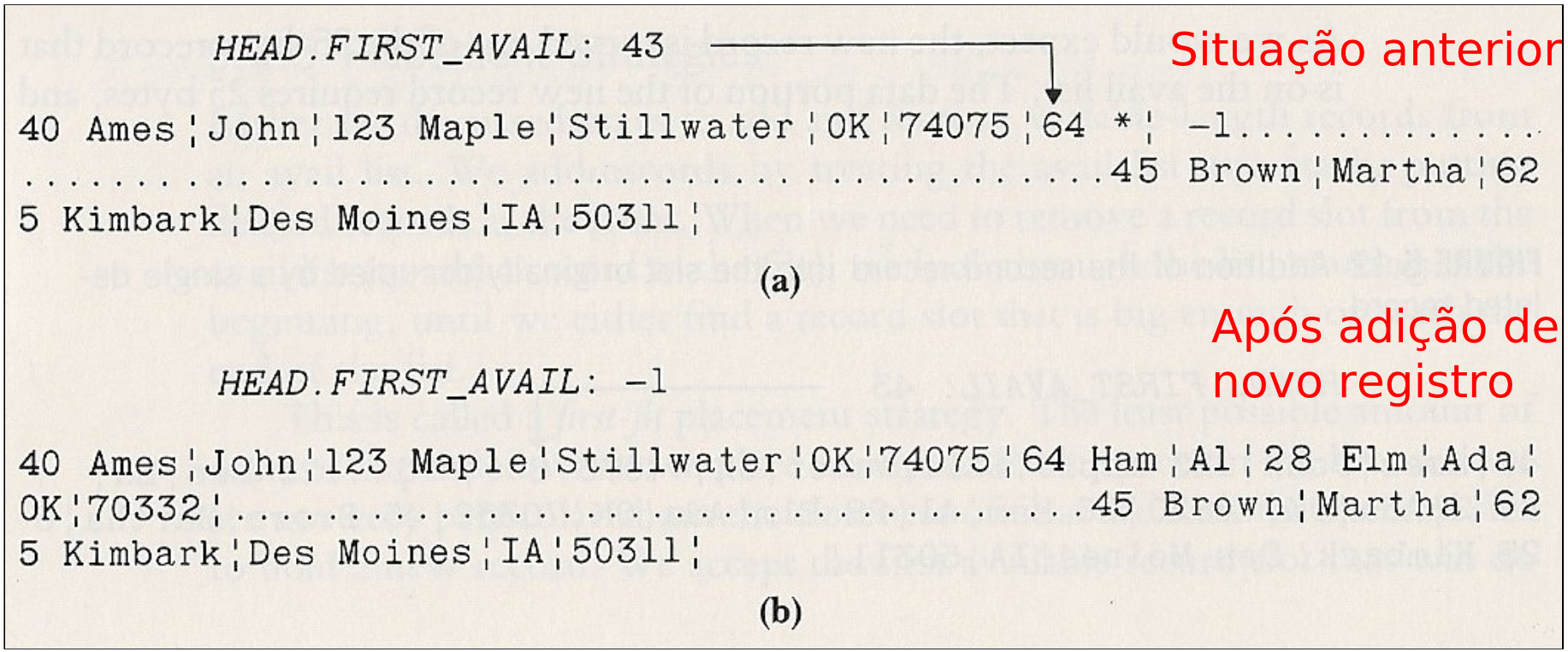


- Estratégias de alocação de espaço
  - *First-fit*: pega-se o primeiro que servir, como feito anteriormente
    - Desvantagem?

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



Registros de tamanho variável

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

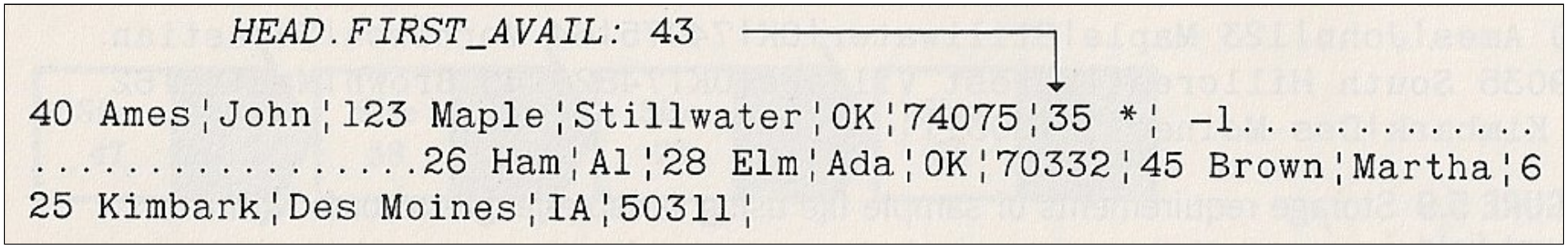
Registros de tamanho variável

- Estratégias de alocação de espaço
  - *First-fit*: pega-se o primeiro que servir, como feito anteriormente
  - Soluções?
    - Colocar o espaço que sobrou na lista de espaços disponíveis
    - Escolher o espaço mais justo possível

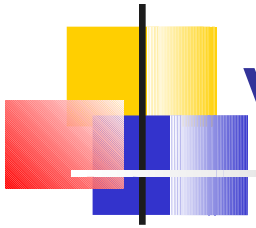
Combatendo a fragmentação

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

Adição do espaço restante à lista



Registros de tamanho variável

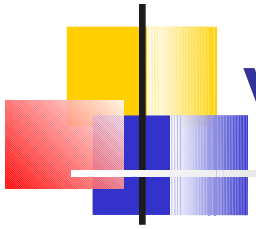


- Solução: escolher o espaço mais justo possível
  - *Best-fit*: pega-se o mais justo
    - Organiza-se a lista de espaços livres de forma ascendente, buscando o primeiro que couber
    - Desvantagem?

Registros de tamanho variável

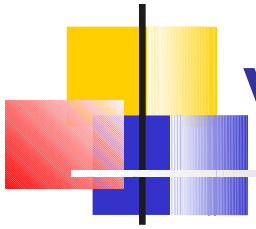
- Solução: escolher o espaço mais justo possível
  - *Best-fit*: pega-se o mais justo
    - Organiza-se a lista de espaços livres de forma ascendente, buscando o primeiro que couber
    - Desvantagem?
      - O espaço que sobra pode ser tão pequeno que não dá para reutilizar
      - Fragmentação externa

Registros de tamanho variável



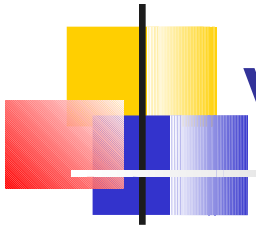
- Solução: escolher o espaço mais justo possível
  - *Best-fit*: pega-se o mais justo
    - Organiza-se a lista de espaços livres de forma ascendente, buscando o primeiro que couber
    - Desvantagem?
      - Vale a pena organizar a lista?

Registros de tamanho variável



- Solução: escolher o maior espaço possível
  - *Worst-fit*: pega-se o maior
    - Diminui a fragmentação externa
    - Lista organizada de forma descendente?
      - O processamento pode ser mais simples, pois se pega o primeiro espaço da lista (o maior)

Registros de tamanho variável



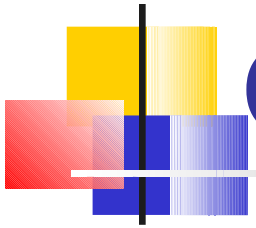
- Outra forma de combater fragmentação externa
  - Junção de espaços vazios adjacentes
    - *Coalescimento*
      - Qual a dificuldade desta abordagem?

Registros de tamanho variável

- Outra forma de combater fragmentação externa
  - Junção de espaços vazios adjacentes
    - *Coalescimento*
      - Qual a dificuldade desta abordagem?
        - A adjacência de registros na lista é lógica, não física, o que forçaria a busca por registros adjacentes
        - Tem solução?

Registros de tamanho variável

- Outra forma de combater fragmentação externa
  - Junção de espaços vazios adjacentes
    - *Coalescimento*
      - Qual a dificuldade desta abordagem?
        - A adjacência de registros na lista é lógica, não física, o que forçaria a busca por registros adjacentes
        - Que tal mais de um encadeamento?



Observações

- Estratégias de alocação só fazem sentido com registros de tamanho variável
  - Por que?

- Recomendações

  - Se espaço está sendo desperdiçado como resultado de fragmentação interna, então a escolha é entre *first-fit* e *best-fit*
    - A estratégia *worst-fit* pode piorar esse problema

  - Se o espaço está sendo desperdiçado devido à fragmentação externa, deve-se considerar a *worst-fit*

Exercício em duplas

- Implemente o esquema de remoção e adição de registros de tamanho fixo, mantendo uma lista encadeada de espaços disponíveis no arquivo
  - Atenção: a lista deve ser mantida no próprio arquivo

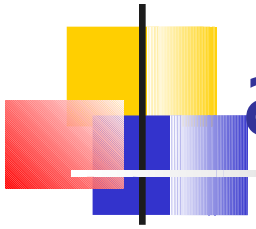
Lembrando...

List head (first available record) → 1

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

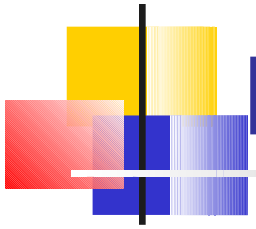
Ordenação e busca em arquivos

Ordenação e busca em arquivos



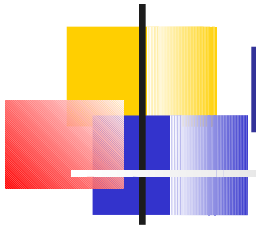
- É relativamente fácil buscar elementos em conjuntos ordenados

- A ordenação pode ajudar a diminuir o número de acessos a disco



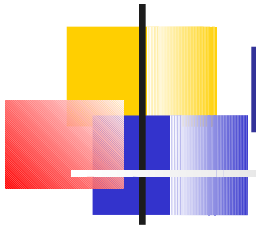
Busca em arquivos

- Já vimos busca seqüencial
  - O(n) → Muito ruim para acesso a disco!
- E a busca binária?
  - Modo de funcionamento?
  - Complexidade de tempo?



Busca binária

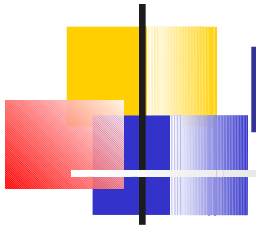
- Dificuldade: ?



Busca binária

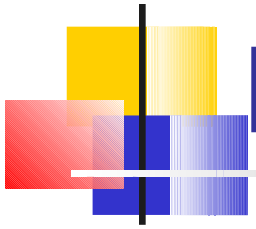
- Dificuldade: ordenar os dados em arquivo para se fazer a busca binária

- Alternativa: ordenar os dados em RAM
  - Ainda é necessário: ler todo o arquivo e ter memória interna disponível



Busca binária

- Limitações
  - Registros de tamanho fixo
  - Manter um arquivo ordenado é muito caro
  - Requer mais do que 1 ou 2 acessos
    - Por exemplo, em um arquivo com 1.000 registros, são necessários aproximadamente 10 acessos em média → ainda é ruim!

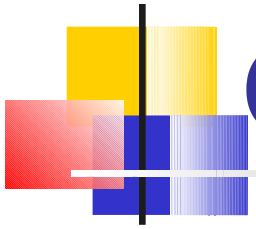


Busca binária

- Exercício em duplas para entregar

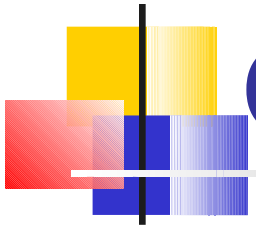
  - Implementar em C uma sub-rotina de busca binária em um arquivo ordenado por número USP

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



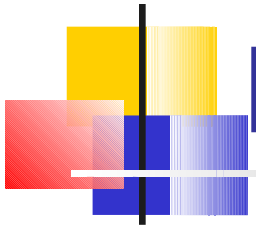
Ordenação

- Alternativa para carregar registros na RAM e ordená-los?
  - Tem como fazer melhor?



Ordenação

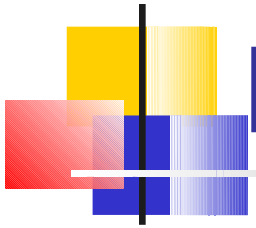
- Alternativa para carregar registros na RAM e ordená-los?
  - Carregar somente as chaves para ordenação
    - Pois elas são essenciais para a ordenação, não o registro todo



Keysorting

- Ordenação por chaves

- Idéia 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

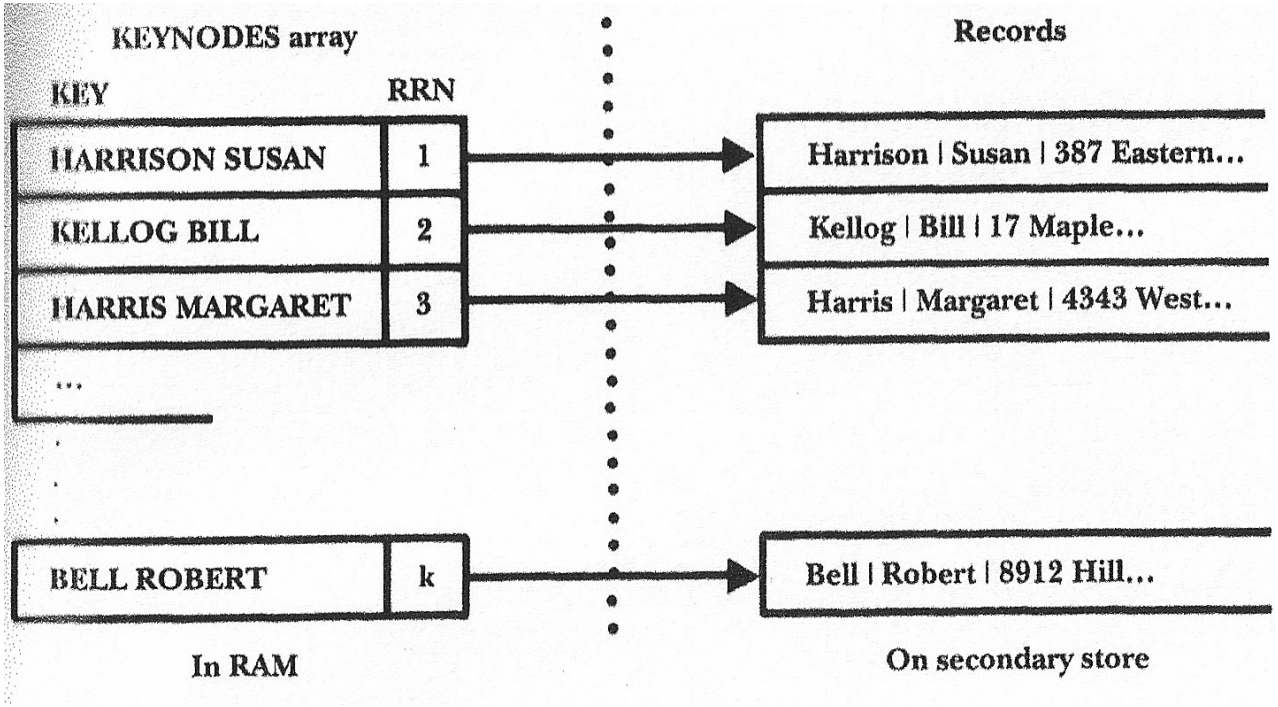
1. Cria-se na memória interna um vetor, em que cada posição tem uma chave do arquivo e um ponteiro para o respectivo registro no arquivo (RRN ou byte inicial)

1. Ordena-se o vetor na memória interna

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

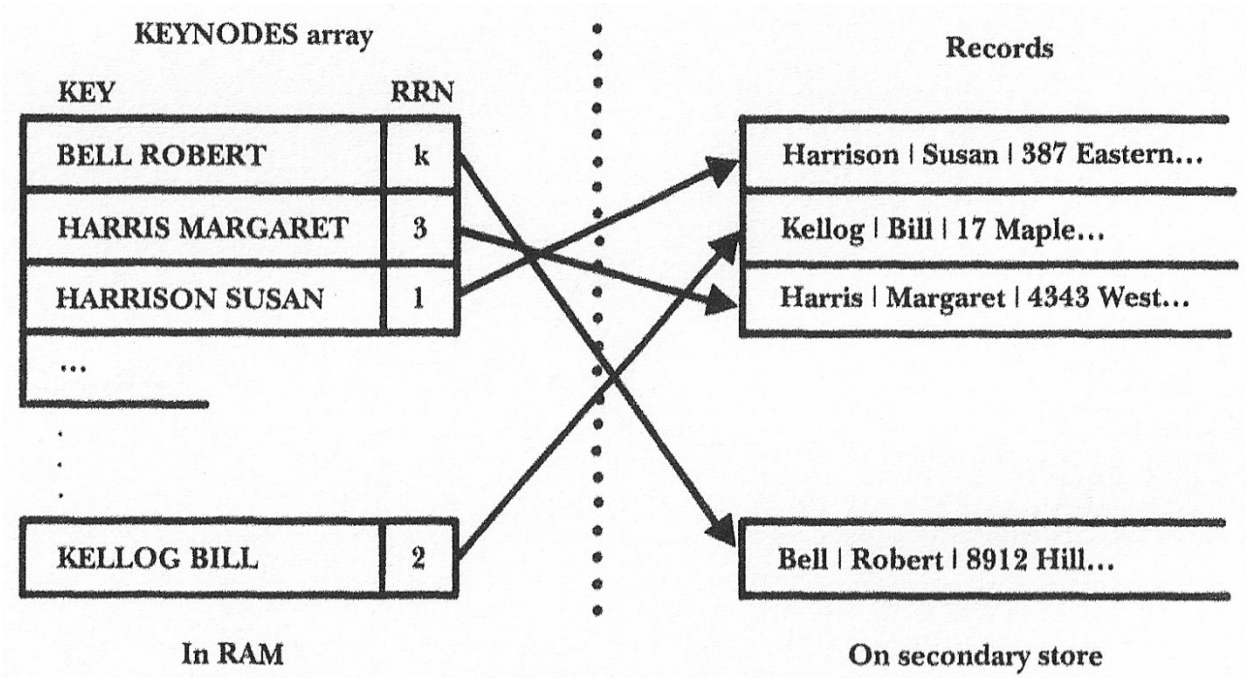
Keysorting

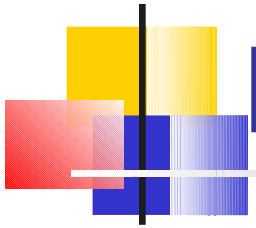
- Exemplo
  - Carregando dados na RAM



Keysorting

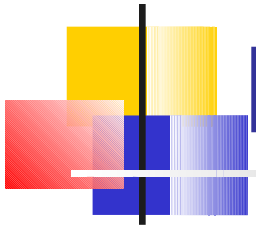
- Exemplo
  - Ordenando dados em RAM





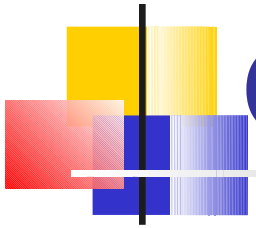
Keysorting

- 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 seqüencial
    - Alterna-se leitura no arquivo antigo e escrita no arquivo novo



Keysorting

- Questões
  - Por que criar um novo arquivo?
  - Não vale a pena usar o vetor ordenado como um índice?
    - Nesse caso, em um outro arquivo



Questão delicada

- Independentemente do método de ordenação
  - O que fazer com os espaços vazios originados de registros eliminados?
    - E a estrutura de dados que os mantêm para que sejam reutilizados?
      - *Pinned records*