Busca Binária e Ordenação

Organização e Recuperação de Dados

Profa. Valéria

UEM – CTC – DIN

Slides preparados com base no Cap. 5 do livro FOLK, M.J. & ZOELLICK, B. *File Structures*. 2nd Edition, Addison-Wesley Publishing Company, 1992, e nos slides disponibilizados pelo Prof. Pedro de Azevedo Berger (DCC/UnB)

Retomando...

- Até agora, recuperação rápida (acesso direto) é feita via **RRN** ou ***byte-offset***, quando conhecido
  - Não dá nenhuma informação sobre o conteúdo do registro
- Mesmo conhecendo o RRN/*byte-offset* do registro que estamos buscando:
  - Qual o registro armazenado no RRN 1520? 👎 Improvável
  - Qual o registro armazena os dados do João da Silva? 👍 Provável

Ou seja, estamos interessados em busca por chave!

Busca por chave

- A busca por chave em um arquivo desordenado implica em uma **busca sequencial**
  - Se nenhum registro contém a chave buscada?
    - Busca sequencial no arquivo todo (Pior caso)
  - Se existir mais de um registro contendo a chave buscada e nós queremos encontrar todos eles?
    - Busca sequencial no arquivo todo (Pior caso)

- Podemos substituir a busca sequencial pela **busca binária**
  - Qual é o custo?

Busca binária em memória

- Função que recebe um valor **x** a ser buscado e uma lista ordenada **v**
- Retorna um índice **m** tal que **v[m] == x** ou **-1** se não encontra **x**

```
def buscaBinaria(x: int, v: list) -> int:
    i = 0
    f = len(v)-1
    while i <= f:
        m = (i + f)//2
        if v[m] == x: return m
        if v[m] < x: i = m + 1
        else: f = m - 1
    return -1
```

Busca binária em arquivo

- Pré-requisitos para BB em arquivo
  - Requer registros de tamanho fixo
  - O arquivo deve estar ordenado em ordem crescente da chave de busca

- Complexidade
  - Em um arquivo de *n* registros
    - Busca binária: ***O(log$_2$n)***, pior caso: $\lfloor log_2 n \rfloor$
      - Em cada "chute", elimina-se metade das possibilidades
    - Busca sequencial: ***O(n)***, pior caso: *n*
  - Se o arquivo pesquisado dobrar de tamanho
    - A busca **binária** faz uma leitura a mais
    - A busca **sequencial** faz o dobro de leituras

Busca binária em arquivo

- Pré-requisitos para BB em arquivo
  - Requer registros de tamanho fixo
  - O arquivo deve estar ordenado em ordem crescente da chave de busca

- Complexidade
  - Em um arquivo de *n* registros
    - Busca binária: ***O(log$_2$n)***, pior caso: $\lfloor log_2 n \rfloor$
      - Em cada "chute", elimina-se metade das possibilidades
    - Busca sequencial: ***O(n)***, pior caso: *n*
  - Se o arquivo pesquisado dobrar de t
    - A busca **binária** faz uma leitura a ma
    - A busca **sequencial** faz o dobro de leituras

Quanto maior o *n*, maior é a diferença de desempenho entre a busca binária e a busca sequencial

Busca binária em arquivo

- A busca binária é mais eficiente do que a sequencial, **porém o arquivo deve estar ordenado por chave**
  - Custo de ordenar e manter o arquivo ordenado após novas inserções
    - A cada inserção deve-se reordenar o arquivo ou fazer a inserção no arquivo em memória e regravar em disco
- Uma primeira solução → ***Ordenação Interna***
  - Só é factível se o arquivo for pequeno e couber na memória
    1. Ler todo o arquivo do disco para a RAM (Acesso sequencial)
    2. Ordenar os registros em RAM usando um algoritmo de ordenação
    3. Gravar o arquivo de volta para o disco (Acesso sequencial)
  - Minimiza o número de acessos ao disco, mas nem sempre é possível de ser usada
  - Deve ser utilizada sempre que possível

Busca binária em arquivo

- A busca binária é mais eficiente do que a sequencial, **porém o arquivo deve estar ordenado por chave**
  - Custo de ordenar e manter o arquivo ordenado após novas inserções
    - A cada inserção deve-se reordenar o arquivo ou fazer a inserção no arquivo em memória e regravar em disco
- Uma primeira solução → ***Ordenação Interna***
  - Só é factível se o arquivo for pequeno e couber na memória
    1. Ler todo o arquivo do disco para a RAM (Acesso sequencial)
    2. Ordenar os registros em RAM usando um algoritmo de ordenação
    3. Gravar o arquivo de volta para o disco (Acesso sequencial)
  - Minimiza o número d
    de ser usada
  - Deve ser utilizada sem

A ***ordenação interna para arquivos*** exemplifica uma classe de soluções para o problema de minimizar ao disco → faça com que os acessos ao arquivo sejam sequenciais e execute a parte complexa, que envolve acessos aleatórios, em RAM

Problemas da busca binária e da ordenação interna

Estratégia "*sort, then binary search*"

- **Problema 1**: A busca binária requer menos acessos que a busca sequencial, mas ainda assim são muitos
  - Para um arquivo com 100.000 registros, teremos em média 16,5 acessos
    - Supondo que para cada leitura será necessário um *seek* em disco, o custo é alto para um arquivo pesquisado frequentemente
  - **Situação IDEAL** → obter o desempenho do acesso direto com a vantagem do acesso por chave
    - Veremos mais adiante que isso pode ser feito usando-se um índice

Problemas da busca binária e da ordenação interna

- **Problema 2**: Manter um arquivo ordenado é custoso
  - Se as inserções são muito frequentes
    - Situação 1: arquivo desordenado + busca sequencial
      - Busca lenta, mas inserção rápida
    - Situação 2: arquivo ordenado + busca binária
      - Busca mais rápida, mas a inserção é lenta
      - A cada inserção deve-se pesquisar a posição de inserção e deslocar os registros posteriores em uma posição
      - Mais lento que a Situação 1!
  - Se as inserções são pouco frequentes
    - Busca binária + atualizações do arquivo ordenado em lote
      - Merge entre o arquivo de dados e o arquivo de atualizações
  - **Possível solução...**
    - Com o uso de índices, o arquivo de dados não precisa estar ordenado

Problemas da busca binária e da ordenação interna

- **Problema 3**: A ordenação interna só é viável para arquivos pequenos
  - A nossa habilidade de fazer busca binária é limitada pela nossa habilidade de ordenar o arquivo
  - Se o arquivo for grande, teremos que usar um algoritmo de ordenação externa (i.e. ordenação em disco)

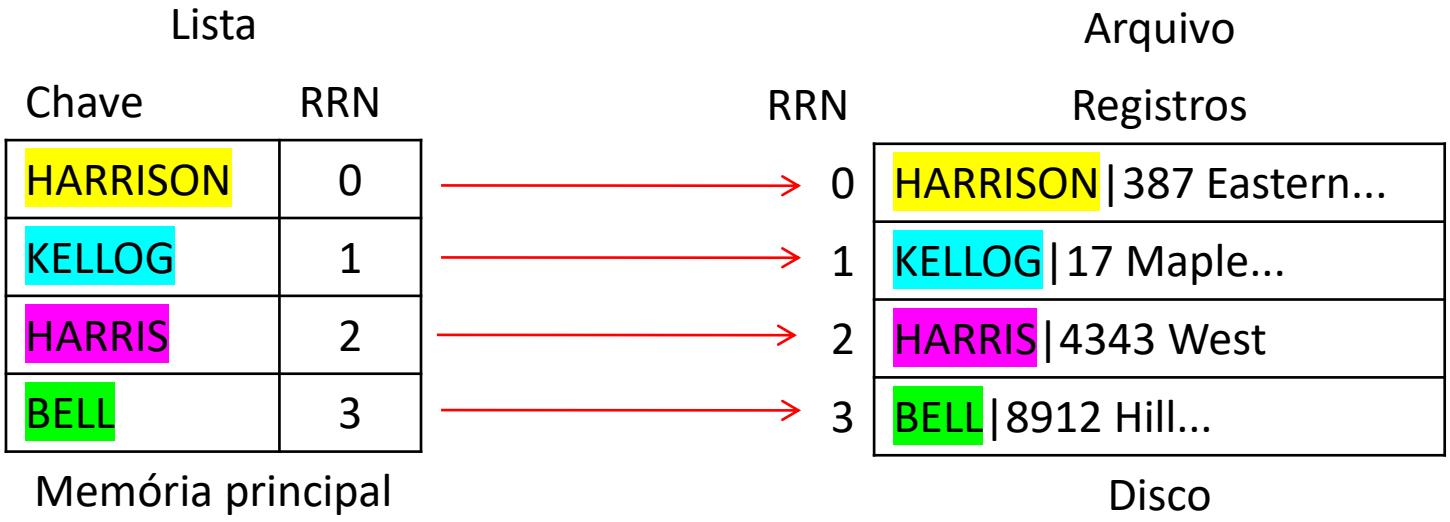
  - **Variação da *ordenação interna* para arquivos grandes → *Keysort***
    - O tamanho do arquivo que o *Keysort* pode ordenar **ainda é limitado pela quantidade de memória RAM disponível**, mas seu limite é maior que o da ordenação interna vista anteriormente

Keysort

- **Para ordenar um arquivo, precisamos apenas das chaves**
- Pode ser utilizado quando a memória RAM é limitada para armazenar o arquivo completo, mas suficiente para armazenar uma lista de chaves e referências para os registros do arquivo
- Passos do ***Keysort***:
  1. Leia o arquivo e coloque em uma lista as **chaves** e os respectivos **RRNs/*byte-offset*** de cada registro
     - Cada registro será lido integralmente do arquivo, mas apenas a chave e o respectivo RRN/*byte-offset* serão inseridos na lista
  2. Ordene a lista usando qualquer algoritmo de ordenação interna
  3. Reescreva o arquivo de dados segundo a ordem dada pela lista ordenada
     - Será necessário ler novamente cada registro do arquivo fonte e reescrevê-lo no arquivo destino

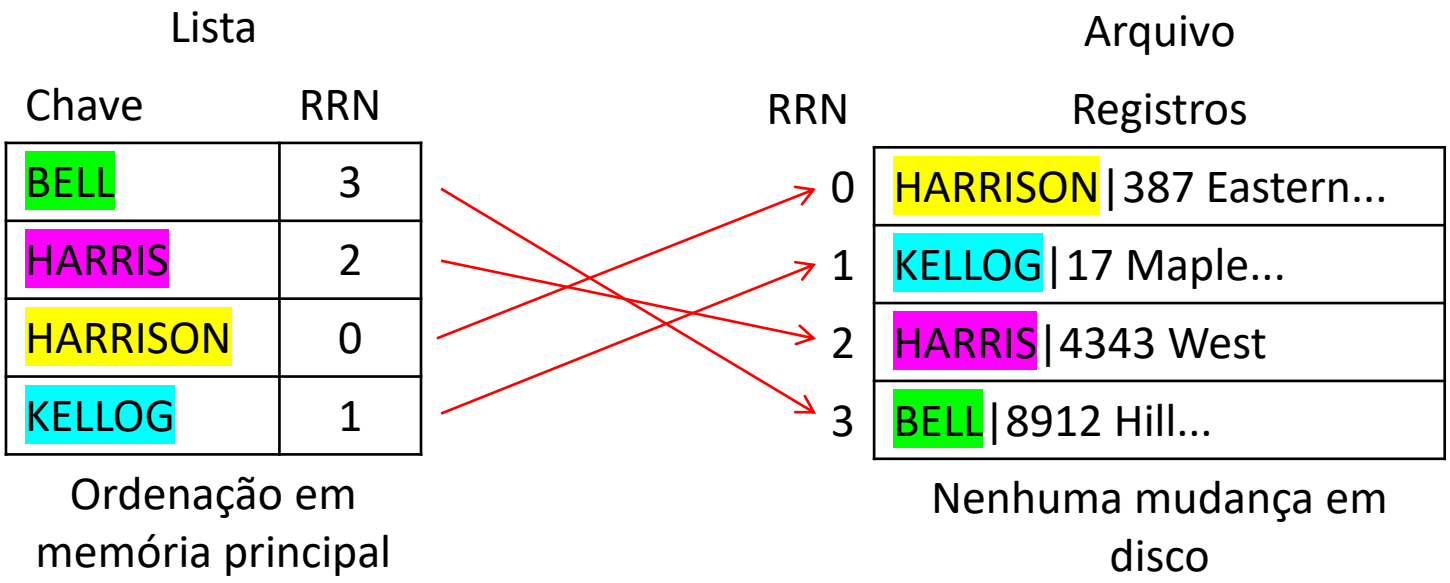
Keysort

- **Passo 1**: Leia o arquivo sequencialmente e coloque em uma lista as **chaves** e os respectivos **RRNs/*byte-offset*** de cada registro



Keysort

- **Passo 2**: Ordene a lista



Perceba que as chaves foram ordenadas na memória, mas não houve alteração no arquivo

Keysort

- **Passo 3**: Reescreva o arquivo de registros segundo a ordem dada pela lista ordenada

Lista

Chave	RRN
BELL	3
HARRIS	2
HARRISON	0
KELLOG	1

A memória principal pode ser liberada

Arquivo

RRN	Registros
0	BELL|8912 Hill...
1	HARRIS|4343 West...
2	HARRISON|387 Eastern...
3	KELLOG|17 Maple...

Criação de um **novo arquivo ordenado** que substitui o antigo

Perceba que os registros mudaram de RRN no arquivo

Keysort: pseudocódigo para registros de tamanho fixo

```
abra o arquivo de entrada como ENTRADA
crie o arquivo de saída como SAIDA
leia o cabeçalho de ENTRADA e grave uma cópia em SAIDA
CONT_ REG ← número de registros do arquivo ENTRADA (lido do cabeçalho)
/* leia os registros e inicialize LISTA */
para i ← 0 até CONT_ REG-1
    leia um registro de ENTRADA para BUFFER
    extraia a chave e armazene-a em LISTA[i].CHAVE
    LISTA[i].RRN ← i
ordene LISTA pelo campo CHAVE
/* leia os registros de acordo com sua ordenação e
    os escreva de forma ordenada no arquivo de saída */
para i ← 0 até CONT_ REG-1
    busque em ENTRADA o reg com RRN = LISTA[i].RRN
    leia reg de ENTRADA para BUFFER
    escreva BUFFER para SAIDA
feche ENTRADA e SAIDA
```

Limitações do Keysort

- Cada registro será lido duas vezes para a escrita do arquivo ordenado
- Ler/escrever os registros na ordem da lista ordenada custa caro
  - São lidas porções dispersas em disco, gerando tempos elevados de *seek*, latência e transferência
    - A leitura/escrita não é sequencial → nº de *seeks* = nº de registros
- **Solução**
  - Usar a lista de chaves para criar um índice em vez de mover os registros em disco
  - Passamos a ter dois arquivos:
    - O arquivo de índice que é ordenado e acessado na realização das buscas
    - O arquivo de dados que permanece desordenado, mas é referenciado pelo arquivo de índice

Índices

- Use a lista de chaves como um **índice** para o arquivo de registros
  - Posteriormente, grave o índice em um arquivo de índice
- **Faça busca binária no índice em memória**
  - Use o RRN/*byte-offset* associado à chave para o acesso no arquivo
    - O arquivo de dados passa a ser acessível por chave com um único acesso (acesso direto)
  - Na inserção de novos registros:
    - Reutilize o espaço de registros removidos ou insira o novo registro no final do arquivo
    - Atualize o índice que está na memória e deve permanecer ordenado
  - Como a busca binária agora é feita no índice, a restrição de tamanho fixo para os registros de dados deixa de existir
    - Os registros do arquivo de índice ainda precisam ter tamanho fixo

Registros "fixos" (pinned records)

- Os registros ligados na lista de espaços disponíveis (PED ou LED) são "fixos" (*pinned*)
  - Não podem ser movidos de sua localização física
- Ordenar arquivos com registros fixos é problemático
  - Registros logicamente removidos e ligados em uma LED/PED não podem mudar de endereço físico
    - Cada registro removido possui um ponteiro (endereço físico) para o próximo na LED/PED
- O uso de um índice também soluciona esse problema
  - Apenas o índice é ordenado
  - O arquivo de dados nunca é ordenado, evitando o problema dos registros fixos