Hashing Extensível:
Implementação

Organização e Recuperação de Dados

Profa. Valéria

UEM – CTC – DIN

Slides preparados com base no Cap. 11 do livro FOLK, M.J. & ZOELLICK, B. *File Structures*. 2nd Edition, Addison-Wesley Publishing Company, 1992

Aspectos da implementação

- A função *hash* para o *hashing extensível* pode ser a mesma do *hashing* estático
  - Qualquer função **hash** que recebe uma **chave** e retorna um **inteiro**
  - Não precisa mais modular pelo tamanho máximo do espaço de endereços
- O número de bits que será utilizado como endereço varia conforme a profundidade global, por isso precisamos de uma função que mapeie o retorno da função *hash* para um endereço adequado ao diretório
  - Função **gerar_endereco(chave, profundidade)**
    - Chama a função *hash* para obter um inteiro **val_hash**, inverte a ordem dos bits de **val_hash** e retorna um endereço com profundidade bits

Aspectos da implementação

- Por que inverter a ordem dos bits?
  - Os bits menos significativos (mais à direita) tendem a variar mais do que os bits mais significativos
    - Muitos endereços nem chegam a usar os bits “mais altos”, de modo que esses bits serão iguais a zero para grande parte das chaves
  - Exemplo

Chave	**Saída binária da função *hash***				***invertido***
bill	**0000**	0011	0110	1100	0011...
lee	**0000**	0100	0010	1000	0001...
pauline	**0000**	1111	0110	0101	1010...
alan	0100	1100	1010	0010	0100...
julie	0010	1110	0000	1001	1001...
mike	**0000**	0111	0100	1101	1011...
elizabeth	0010	1100	0110	1010	0101...
mark	**0000**	1010	0000	0111	1110...

Aspectos da implementação

- Função **gerar_endereco(chave, profundidade)**

 # Inverte os bits do endereço e extrai PROFUNDIDADE bits

faça **val_ret** = 0 *# Armazenará a sequência de bits*

faça **mascara** = 1 *# Máscara 0...001 para extrair o bit menos significativo*

val_hash = **hash(chave)**

Para **j** = 1 até **profundidade** faça

val_ret = desloque um bit de **val_ret** uma posição à esquerda *# val_ret << 1*

 # Extraia o bit de mais baixa ordem de val_hash

bit_baixa_ordem = **val_hash** *e_bitwise* **mascara** *# val_hash & mascara*

 # Insira bit_baixa_ordem no final de val_ret

val_ret = **val_ret** *ou_bitwise* **bit_baixa_ordem** *# val_ret | bit_baixa_ordem*

val_hash = desloque um bit de **val_hash** uma posição à direita *# val_hash >> 1*

retorne **val_ret**

Bucket e diretório

- A estrutura do ***bucket***
  - Classe **bucket**
    - **prof**: inteiro que armazena a profundidade do *bucket*, i.e., quantos bits estão sendo usados para endereçar as chaves deste *bucket* particular
    - **cont**: inteiro que armazena o número de chaves contidas no *bucket*
    - **chaves**: lista que armazena até TAM_MAX_BUCKET chaves
      - Essa estrutura pode ser modificada para armazenar todo o registro em vez de somente as chaves
- A estrutura do **diretório**
  - Classe **diretorio**
    - **refs**: lista de referências (RRN) para buckets
    - **prof_dir**: inteiro que armazena a profundidade do diretório

Hashing extensível

- Os objetos **bucket** são armazenados em um **arquivo binário de *buckets***
- O objeto **diretório** é inicialmente criado na RAM e posteriormente armazenado em um **arquivo binário de diretório**
- A estrutura do **hashing extensível**
  - Classe **hashing_extensível**
    - **arq_bk**: descritor do arquivo de *buckets*
    - **dir**: referência para um objeto diretório

Utilize **pack/unpack** do módulo **struct** para formatar os dados escritos/lidos nos arquivos de *buckets* e de diretório

Inicialização do *hashing*

- Inicialização do **hashing extensível**
  - Se o *hashing* existir
    - Abra os arquivos de diretório e de buckets
    - Leia a profundidade (**prof_dir**)
    - Calcule o tamanho do diretório → tamanho = $2^{\textbf{prof\_dir}}$
    - Leia os registros do arquivo de diretório para um objeto **diretorio**
  - Se o *hashing* não existir
    - Crie o arquivo de *buckets*
    - Crie um objeto **diretorio** e atribua a **dir**
    - Inicialize **prof_dir** com 0
    - Crie um **bucket** vazio no arquivo de *buckets* e atribua seu RRN ao **dir.refs**

Finalização do *hashing*

- Finalização do **hashing extensível**
  - Abra o arquivo de diretório
  - Escreva o objeto **diretorio** no arquivo de diretório
    - Você pode gravar **prof_dir** no cabeçalho do arquivo
  - Feche os arquivos de diretório e de *buckets*

Busca e inserção

- Função de busca: ***op_buscar(chave)***

  endereco = **gerar_endereco(chave, prof_dir)**

  ref_bk = **dir.refs[endereco]**

  Leia o *bucket* da **ref_bk** para **bk_encontrado** e busque pela **chave**

  Se a **chave** foi encontrada, então retorne *True*, **ref_bk** e **bk_encontrado**

  Retorne ***False*, ref_bk** e **bk_encontrado**

- Função de inserção: ***op_inserir(chave)***

  achou, **ref_bk**, **bk_encontrado = op_buscar(chave)**

  Se **achou**, então retorne ***False*** *# Erro: chave duplicada*

  inserir_chave_bk(chave, ref_bk, bk_encontrado)

  retorne ***True***

Inserção e auxiliares

- Função ***inserir_chave_bk(chave, ref_bk, bucket)***

  Se **bucket.cont** < TAM_MAX_BUCKET, então insira a **chave** no **bucket** e escreva-o em **ref_bk** no arquivo de *buckets*

  Senão

  dividir_bk (ref_bk, bucket)

  op_inserir(chave) *# recursão indireta*

- Função ***dividir_bk (ref_bk, bucket)***

  Se **bucket.prof** é igual a **prof_dir**, então **dobrar_dir()**

  Crie um **novo_bucket** e atribua o seu RRN a **ref_novo_bucket**

  novo_inicio, **novo_fim** = **encontrar_novo_intervalo(bucket)**

  Insira **novo_bucket** no **dir** de acordo com **novo_inicio** e **novo_fim**

  Incremente **bucket.prof**

  Faça **novo_bucket.prof** receber **bucket.prof**

  Redistribua as chaves entre **bucket** e **novo_bucket**

  Escreva **bucket** e **novo_bucket** nos respectivos RRNs do arquivo de *buckets*

Inserção e auxiliares

- Função ***dobrar_dir()***

  Crie uma lista **novas_refs**

  Insira cada referência em **dir.refs** duas vezes em **novas_refs**

  Atribua **novas_refs** a **dir.refs**

  Incremente **prof_dir**

Inserção e auxiliares

- Função ***encontrar_novo_intervalo(bucket)***

  Encontre o "endereço comum" das chaves contidas em **bucket**

  - **end_comum** = **gerar_endereco(**qualquer chave de **bucket**, **bucket.prof)**

  Desloque **end_comum** um bit para esquerda e então coloque 1 no lugar do bit de menos significativo

  - Esse será o novo **end_comum** das chaves que ficarão no novo *bucket*

  novo_inicio receberá **end_comum** preenchido com 0's à direita até que o endereço tenha o tamanho correto

  novo_fim receberá **end_comum** preenchido com 1's à direita até que o endereço tenha o tamanho correto

  - Para saber quantos bits precisam ser preenchidos em **end_comum** (**bits_a_preencher**)
    - **bits_a_preencher** = **prof_dir** – (**bucket.prof** + 1)

Inserção e auxiliares

- Função ***encontrar_novo_intervalo(bucket)***

Código python

```
mascara = 1
chave = bucket.chaves[0]
end_comum = gerar_endereco(chave, bucket.prof)
end_comum = end_comum << 1
end_comum = end_comum | mascara
bits_a_preencher = dir_prof - (bucket.prof + 1)
novo_inicio, novo_fim = end_comum, end_comum
for i in range(bits_a_preencher):
        novo_inicio = novo_inicio << 1
        novo_fim = novo_fim << 1
        novo_fim = novo_fim | mascara
return novo_inicio, novo_fim
```

Resumindo a inserção

- A função ***op_inserir*** gerencia a inserção
  - Se a chave é encontrada, *op_inserir* retorna *False* e termina
  - Senão, *op_inserir* chama *inserir_chave_bk* passando como parâmetro o *bucket* no qual será feita a inserção
  - Se *inserir_chave_bk* encontra espaço no *bucket* para a inserção, a chave é inserida e a operação termina
  - Se o *bucket* está cheio, *inserir_chave_bk* chama *dividir_bk* para fazer a divisão do *bucket*
  - *dividir_bk* primeiramente determina se o diretório é grande o suficiente para acomodar o novo *bucket*
  - Se o diretório precisa ser expandido, *dividir_bk* chama *dobrar_dir* para dobrar o tamanho do diretório
  - *dividir_bk* aloca um novo *bucket*, o liga as referências apropriadas do diretório e redistribui as chaves entre os dois *buckets*
  - Quando o controle retorna para *inserir_chave_bk*, *op_inserir* é chamada para uma nova tentativa de inserção, agora usando a nova estrutura de diretório e *buckets*
  - *op_inserir* chama *inserir_chave_bk* novamente (recursão indireta)
  - O ciclo continua até que exista um *bucket* disponível para acomodar a nova chave

Remoção e auxiliares

- Função de remoção: ***op_remover(chave)***

  achou, ref_bk, bk_encontrado = **op_buscar(chave)**

  Se não **achou**, então retorne *False*

  Retorne **remover_chave_bk (chave, ref_bk, bk_encontrado)**

- Função ***remover_chave_bk(chave, ref_bk, bucket)***

  Faça **removeu** receber *False*

  Busque por **chave** em **bucket**

  Remova **chave** do **bucket** e decremente **bucket.cont**

  Reescreva **bucket** em **ref_bk** no arquivo de *buckets*

  Faça **removeu** receber *True*

  Se **removeu**, então

  tentar_combinar_bk (ref_bk, bucket)

  retorne *True*

  Senão retorne *False*

Remoção e auxiliares

- Função ***tentar_combinar_bk (ref_bk, bucket)***

  # verifique se bucket tem um amigo

  tem_amigo, endereco_amigo = **encontrar_bk_amigo(bucket)**

  Se não **tem_amigo**, então retorne *# fim*

  Leia o *bucket* do **endereco_amigo** e armazene-o em **bk_amigo**

  # verifique se bucket e bk_amigo podem ser concatenados

  Se (**bk_amigo.cont** + **bucket.cont**) ≤ TAM_MAX_BUCKET, então

  ref_amigo = **dir.refs[endereço_amigo]**

  bucket = **combinar_bk (ref_bk, bucket, ref_amigo, bk_amigo)**

  Faça a entrada do **dir** que apontava para **bk_amigo** apontar para **bucket**

  # após a concatenação, verifique se o diretório pode diminuir de tamanho

  Se **tentar_diminuir_dir()**

  # se o diretório diminuir, um novo amigo pode ter surgido

  tentar_combinar_bk(ref_bk, bucket) *# recursão*

Remoção e auxiliares

- Função ***encontrar_bk_amigo(bucket)***

  Se **prof_dir** é igual a 0, então retorne *False, None*

  Se **bucket.prof** < **prof_dir**, então retorne *False, None*

  Encontre o **end_comum** das chaves contidas em *bucket*

  Encontre o endereço do *bucket* amigo (**end_amigo**)

  - **end_amigo** = **end_comum** com o bit menos significativo invertido

    end_amigo = **end_comum** XOR_ *bitwise* 1 *# end_bk ^ 1*

  Retorne *True*, **end_amigo**

- Função ***combinar_bk(ref_bk, bucket, ref_amigo, bk_amigo)***

  Copie as chaves do **bk_amigo** para **bucket**

  Atualize **bucket.cont** e decremente **bucket.prof**

  Reescreva **bucket** em **ref_bk** no arquivo de *buckets*

  Remova **bk_amigo** do **ref_amigo** no arquivo de *buckets* *# remoção lógica*

  Retorne **bucket**

Remoção e auxiliares

- Função ***tentar_diminuir_dir()***

  Se **prof_dir** é igual a 0, então retorne *False*

  Faça **tam_dir** receber $2^{\textbf{prof\_dir}}$

  Faça **diminuir** receber *True* *# assuma que é possível e tente provar o contrário*

  Para **i** = 0 até **tam_dir** – 1 com passo 2

  Se **dir.refs[i]** != **dir.refs[i+1]** então

  Faça **diminuir** receber *False*

  Interrompa o laço *# break*

  Se **diminuir** então

  Crie uma lista **novas_refs**

  Para cada duas referências das **refs** atual, insira uma em **novas_refs**

  Faça **dir.refs** receber **novas_refs**

  Decremente **prof_dir**

  Retorne **diminuir**

Resumo da remoção

- A função ***op_remover*** recebe a chave e inicia o processo de remoção
  - Se a chave não for encontrada, retorna *False* e termina
  - Senão, o *bucket* contendo a chave é passado para a função *remover_chave_bk*
  - *remover_chave_bk* remove a chave do *bucket* e o repassa para a função *tentar_combinar_bk*
  - *tentar_combinar_bk* verifica se o *bucket* que agora está menor pode ser combinado com seu *bk_amigo*
  - Se não existir um *bucket* amigo, retorna e termina
  - Se existir um amigo e o número de chaves nos dois *buckets* amigos for menor ou igual ao tamanho do *bucket*, então eles são concatenados
  - Após a concatenação, *tentar_combinar_bk* chama *tentar_diminuir_dir*, que verifica se o diretório pode ser reduzido
  - Quando o controle volta para *tentar_combinar_bk*, verifica se o diretório diminuiu
  - Se sim, um novo amigo pode ter surgido, então *tentar_combinar_bk* é chamada recursivamente
  - A remoção termina quando não houverem mais *buckets* amigos a serem concatenados