AEDII Arquivos

•

1 Pré-Árvore B

Estrutura interna de arquivos de dados

→ Armazenamento não volátil também conhecidos como memória secundária, são aqueles que em que o arquivo gravado não se perde quando "desligado". Alguns exemplos de memória secundária são: HD's externo, DVD, Pen-drives entre outros. São de capacidade maior, porém lentas e baratas.

→ Armazenamento volátil são aqueles que em que o que nele está se perde quando o computador é desligado. São os registradores, memória cache, e memória primária. Possuem capacidade menor, são rápidas e caras.

→ Seeking: posiciona a cabeça de L/E no local desejado; o conteúdo de todo um cilindro poder ser lido com apenas 1 seeking; movimento mais lendo da operação de leitura/escrita; deve ser reduzido ao mínimo.

→ O acesso do seek é direto (aleatório ou randômico), pois não há a necessidade de ler todos os dados em sequência. - Por outro lado, fitas precisam de um acesso sequencial.

→ Paginação: capacidade de processar parte dos dados que estavam em memória secundária na memória principal (moldura de memória). Subconjunto do disco em memória principal.

→ Memória virtual

- paginação
- mapeamento de endereços
- reposição de páginas
  - LRU - Menos Recentemente Utilizada
  - LFU - Menos Frequentemente Utilizada
  - FIFO - Primeiro a Entrar Primeiro a Sair

→ Organização de arquivos na memória secundária

- Bloco: unidade de transferência de dados entre memória principal e a memória secundária
  - um bloco é formado por uma ou
  - É comum considerar que em cada bloco pode haver vários registros
    Se R (tamanho fixo do registro, para simplificar) e B (tamanho do bloco) e $R \leq B$:
    fator de blocagem fb = floor(B/R) = número de registros inteiros que cabem em um bloco mais páginas
- Organização espalhada: os blocos são totalmente preenchidos; se um registro não cabe inteiramente na parte vazia do bloco, coloca o que couber e um ponteiro para o próximo bloco
- Organização não espalhada: registros não podem ser divididos. Cada bloco pode conter até fb registros.

- Alocação de Blocos:

  Blocos alocados sequencialmente

  - Leitura fácil (leitura sequencial é ótima, e na leitura aleatória depende da facilidade de localização do deslocamento do registro dentro do arquivo)
  - Expansão complicada: se não houver espaço disponível até o próximo arquivo tem que ser removido para outro local
  - Fragmentação externa (buracos entre os arquivos): maior ou menor dependendo da política de alocação - o disco pode ficar fragmentado, isto é, com vários trechos disponíveis intercortados por trechos utilizados
  - Métodos de ajuste sequencial
    * Primeiro ajuste: seleciona o primeiro trecho encontrado (a partir do início da lista) grande o suficiente - é o mais eficiente: balanço entre tempo de achar um bloco de tamanho suficiente (retorna assim que achar o primeiro) e fragmentação (não deixa sobrar sistematicamente o menor ou maior trecho)
    * Próximo ajuste: seleciona o próximo trecho grande o suficiente (a partir do índice "corrente", ajustado após a última alocação) - similar ao primeiro ajuste, mas chega mais rápido ao fim do heap
    * Melhor ajuste: seleciona o menor trecho dentre os trechos grandes o suficiente - pode ser o pior: gasta tempo analisando tudo e, a menos que o ajuste seja perfeito, deixa sobrar normalmente trechos pequenos que não podem ser reutilizados
    * Pior ajuste: seleciona o maior trecho de todos - tenta evitar esse desperdício, deixando sobrar trechos maiores que podem ser ainda utilizados, e assim posterga a criação de blocos pequenos

  Blocos alocados sequencialmente ordenado

  - Leitura ordenada eficiente (sequencial) – O(b): O próximo registro pode estar no mesmo bloco
  - Mínimo / Máximo estão no cabeçalho do arquivo – O(1): Isto podemos fazer para todos os tipos de alocação
  - Busca: dá para usar busca binária (baseada nos blocos!)

- Inserção: cara! O(b)
    - * Tem que achar a posição certa : O(lg b)
    - * Tem que abrir espaço para o registro (deslocar todos os registros com chave maior para frente) : O(b)
- Exclusão: cara pelos mesmos motivos! - O(b)
- Modificação: busca + atualização

Blocos alocados por lista ligada

Blocos com alocação indexada

- Índices primários
- Índice de clustering
- Índices secundários
- Organização indexada multiníveis

2 Árvore B

→ Com inspiração das árvores binárias de busca e com o dinamismo dos índices multiníveis cria-se a Árvore B.

→ Definição:

- 1. Cada nó x contém os seguintes campos
    - $n[x]$, o número de chaves atualmente armazenadas no nó x;
    - as $n[x]$ chaves, armazenadas em ordem não decrescente, de modo que $key_1[x] \leq key_2[x] \leq ... \leq key_n[x]$;
    - $leaf[x]$, um valor booleano indicando se x é um folha ($true$) ou um nó interno ($false$);
    - se x é um nó interno, x contém $n[x]+1$ ponteiros $c_1[x], c_2[x], ...c_{n[x]+1}[x]$ para seus filhos.

- $\boxed{\text{2.}}$ As chaves $key_i[x]$ separam as faixas de valores armazenado em cada subárvore: denotando por key_i uma chave qualquer armazenada na subárvore com nó $c_i[x]$, tem-se:

$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq ... \leq key_{n[x]}[x] \leq k_{n[x]+1}$$

- $\boxed{\text{3.}}$ Todas as folhas aparecem no mesmo nível, que é a altura da árvore, h.

- $\boxed{\text{4.}}$ Há um limite inferior e superior no número de chaves que um nó pode conter, expressos em termos de um inteiro fixo $t \geq 2$ chamado o *grau mínimo* (ou *ordem*) da árvore.

  - Todo nó que não seja a raiz deve conter pelo menos $t - 1$ chaves.
  - Todo nó interno que não seja a raiz deve conter pelo menos t filhos.
  - Todo nó deve conter no máximo $2t - 1$ chaves (e portanto todo nó interno deve ter no máximo $2t$ filhos). Dizemos que um nó está *cheio* se ele contiver exatamente $2t - 1$ chaves.

- $\boxed{\text{Altura.}}$ Teorema: Para toda árvore B de grau mínimo $t \geq 2$ contendo n chaves, sua altura h máxima será:

$$h \leq \log_t \frac{n+1}{2}$$

$\rightarrow$ Criação de uma árvore vazia

BTreeCreate(T)
$i \leftarrow AllocateNode()$
$leaf[x] \leftarrow true$
$n[x] \leftarrow 0$
$DiskWrite(x)$
$root[T] \leftarrow x$

$\rightarrow$ *B-Tree-Search* toma como entrada um ponteiro para o nó de raiz x de uma subárvore e uma chave k que deve ser procurada nessa subárvore.

Se k está na B-árvore, *B-Tree-Search* retorna o par ordenado (y, i), que consiste em um nó y e um índice i tal que $key_i[y] = k$. Caso contrário, o procedimento retorna NIL. Assim, a chamada de nível superior é da forma *B-Tree-Search*(root[T], k).

BTreeSearch(x, k)
$i = 1$
while $i \leq n[x]$ e $k > key_i[x]$ **do**
    i = i + 1
end while
if $i \leq n[x]$ e $k = key_i[x]$ **then**
    return (x, i)
else if $leaf[x]$ **then**
    return NIL
else
    $DiskRead(c_i[x])$
end if
return $BTreeSearch(x.c_i, k)$

→ Inserção ocorrem sempre nas folhas

- Caso o nó em que a folha será inserida estiver cheia , alguma das chaves deverá ser promovida e esse nó se subdividirá.
- Durante a busca da localização de inserção do nó, no caminho da raiz até uma folha, se achar um nó filho (a ser seguido) cheio, já o subdivide.
- Vamos assumir que o nó atual (pai do nó cheio) não é cheio, a menos da raiz que deve ser tratada separadamente

---

→ Divisão de um nó na árvore: *BTreeSplitChild(x, i, y):* tem como entrada um nó interno x *não cheio*, um índice i e um nó y tal que $y = c_i[x]$ é um filho *cheio* de x. O procedimento divide y em 2 e ajusta x de forma que este terá um filho adicional.

BTreeSplitChild(x, i, y)
$z \leftarrow AllocateNode()$
$leaf[z] \leftarrow leaf[y]$

```
4: n[z] ← t − 1
5: for j ← 1 to t − 1 do
6:     key_j[z] ← key_{j+t}[y]
7: end for
8: if not leaf[y] then
9:     for j ← 1 to t do
10:        c_j[z] ← c_{j+t}[y]
11:    end for
12: end if
13: n[y] ← t − 1
14: for j ← n[x] downto i do
15:     key_{j+1}[x] ← key_j[x]
16: end for
17: key_i[x] ← key_t[y]
18: n[x] ← n[x] + 1
19: DiskWrite(y)
20: DiskWrite(z)
21: DiskWrite(x)
```

→→→ Linhas 2-10 Aloca e inicializa um nó para ser filho da chave que será promovida.
→→→ Linha 13 Ajusta o nó que foi dividido.
→→→ Linhas 14-18 Ajusta o nó em que foi parar o nó que saiu do nó dividido.

→ Iserção de uma chave na árvore com raiz T:

```
BTreeInsert(T, k)
r ← root[T]
if n[r] = 2t − 1 then
    s ← AllocateNode()
    root[T] ← s
    leaf[s] ← false
    n[s] ← 0
    c_1 ← r
    BTreeSplitChild(s, 1, r)
    BTreeInsertNonFull(s, k)
else
    BTreeInsertNonFull(r, k)
```

```
end if
```

→ Iserção de uma chave em uma subárvore cuja raiz x não está cheia:

```
BTreeInsertNonFull(x, k)
i ← n[x]
if leaf[x] then
    while i ≥ 1 and k<key_i[x] do
        key_{i+1}[x] ← key_i[x]
        i ← i − 1
    end while
    key_{i+1}[x] ← k
    n[x] ← n[x] + 1
    DiskWrite(x)
else
    while i ≥ 1 and k<key_i[x] do
        i ← i − 1
    end while
    i ← i + 1
    DiskRead(c + i[x])
    if n[c_i[x]] = 2t − 1 then
        BTreeSplitChild(x, i, c_i[x]
        if k>key_i[x] then
            i ← i + 1
        end if
    end if
    BTreeInsertNonFull(c_i[x], k)
end if
```

→ Remoção em árvores B - BTreeDelete(x, k): remoção da chave k da subárvore com raiz x.

- $\boxed{2.}$ - Se a chave k está no nó x e x é um nó interno, faça:
  - a) Se o filho y que pecede k no nó x tem pelo menos t chaves, então encontre o predecessor de k' de k na subárvore com raiz y. Delete recursivamente k', e substitua k por k' em x.
  - b) Simetricamente, se o filho z imediatamente após k no nó x tem pelo menos t chaves, então encontre o sucessor k' de k na

subárvore com raiz z. delete recursivamente k', e substitua k por k' em x.

  - c) Caso contrário, se ambos y e z possuem apenas $t-1$ chaves, faça a junção de k e todas as chaves de z em y, de forma que x perde tanto a chave k como o ponteiro para z, e y agora contém $2t-1$ chaves. Então, libere z e delete recursivamente k e y.

- $\boxed{3.}$ - Se a chave k não está presente no nó interno x, determine a raiz $c_i[x]$ da subárvore apropriada que deve conter k (se k estiver presente na árvore). Se $c_i[x]$ tem apenas $t-1$ chaves, execute o passo 3a ou 3b conforme necessário para garantir que o algoritmo desça para um nó contendo pelo menos t chaves. Então, continue no filho apropriado de x.

  - a) Se $c_i[x]$ contém apenas $t-1$ chaves mas tem um irmão imediato com pelo menos t chaves, dê para $c_i[x]$ uma chave extra movendo uma chave de x para $c_i[x]$, movendo uma chave do irmão imediato de $c_i[x]$ à esquerda ou à direita, e movendo o ponteiro do filho apropriado do irmão para o nó $c_i[x]$.
  - b) Se $c_i[x]$ e ambos os irmãos imediatos de $c_i[x]$ contêm $t-1$ chaves, faça a junção de $c_i[x]$ com um de seus irmãos. Isso implicará em mover uma chave de x para o novo nó fundido (que se tornará a chave mediana para aquele nó).

- $\boxed{1.}$ - Se a chave k está no nó x e x é uma folha, exclua a chave k de x. - (pelos procedimentos anteriores, já sabemos que o nó x tem pelo menos t chaves).

- $\boxed{***}$ Quando um dos nós for a raiz

  - Ela pode ter menos do que $t-1$ chaves
  - Se ficar com zero chaves precisa desalocar o bloco e atualizar quem é a nova raiz.
  - Precisa de uma camada extra sobre a chamada da deleção:

    BTreeDeleteFromRoot(T, k)
    $r \leftarrow raiz[T]$
    if $n[r] = 0$ **then**
        return
    else
        $BTreeDelete(r, k)$

end if
if $n[r] = 0$ **and not** $leaf[r]$ **then**
    $raiz[T] \leftarrow c1[r]$
    $desaloca(r)$
end if

2.1 Hashing (Espalhamento): Tenenbaum cap 7.4

- Considere uma conjunto de chaves k e seja I conjunto de índices.
  $\rightarrow I = \{0, 1, 2, ..., I - 1\}$

Table 1: Tabela com índices

0
1
2
.
.
.
$I - 1$

- Uma função hash, denotada por $h(k)$
  $h : k \rightarrow I$ para cada chave associa-se um único índice. A idéia é que o custo de uma busca seja $O(1)$.
  - Ex: $h(k) = k \ \% \ T$, onde T é o tamanho do conjunto de índices.

$$Colisao = \begin{cases} h(53) = 3 \\ h(13) = 3 \end{cases} \tag{1}$$

    *colisão é o fato de duas chaves distintas k_1 e k_2 serem associadas ao mesmo índice.

- Tratamento de Colisões
  $\rightarrow$ Endereçamento aberto: consiste em uma função chamada de $rehash-rh(k)$

  $\rightarrow rh : I \rightarrow I$
  $\rightarrow$ Essa função é responsável por "realocar" a chave na qual houve uma colisão.
  *há um limite no tamanho da lista e o tempo de busca.

$\rightarrow$ Dadas funções $h(k)$ e $rh(i)$

bool search (int k)
int $p = h(k)$
while $(T[p]$ **is not** $k)$ **and** $(T[p]$ **is not** $-1))$ **do**
    $p = r//h(k)$
end while
if $T[p] == -1$ **then**
    return false
else
    return true
end if

o laço entrará num loop infinito caso a tabela esteja totalmente cheia. Uma alternativa é definir como cheia a $tabela - 1$ para garantir que a busca termine.

bool insert (int k)
int $p = h(k)$
while $(T[p]$ **is not** $k)$ **and** $(T[p]$ **is not** $-1))$ **do**
    $p = rh(p)$
end while
if $T[p] == -1$ **then**
    $T[p] = k$
    return true
else
    return false
end if

bool remove (int k)
int $p = h(k)$
while $(T[p]$ **is not** $k)$ **and** $(T[p]$ **is not** $-1))$ **do**
    $p = rh(p)$
end while
if $T[p] == k$ **then**
    $T[p] = ?$ //o que fazer?
end if

Atualizar as funções de busca e inserção considerando o marcador de removido.

$\rightarrow$ a função de busca não muda

→ a função de inserção precisa considerar o marcador de removido.

↓ insert modificado.

```
bool insert (int k)
int p = h(k)
int temp = −1
while (T[p] is not k) and (T[p] is not −1)) do
    if T[p] == −2 then
        temp = p
    end if
    p = rh(p)
end while
if T[p] == k then
    return false
else
    if temp is not −1 then
        T[temp] = k
    else
        T[p] = k
        return true
    end if
end if
```

- Eficiência dos métodos de *rehash*

→ Uma medida de eficiencia é contar o número de posições examinadas(probabilidade) antes de encontrar uma chave.
→ O número médio (valores altos de T) de consultas para $\frac{2T-n+1}{2T-2n+2}$ onde n é o número de chaves presentes na tabela.
→ Define-se por *lead factor* $(lf)\frac{n}{T}$ (fração da tabela que está preenchida)
→ Inicialmente todas as posições da tabela tem a mesma probabilidade de serem preenchidas. A medida que a tabela vai enchendo, algumas posições tem um aumento nessa probabilidade.
→ Uma forma de eliminar a aglomeração primária é fazer com que a função $rh()$ dependa de dois argumentos $rh(i, j)$ onde i é o número de vezes que a função $rh()$ for aplicada.
$rh(i, j) = (i + j)\ \%\ T$.

$rh(i,1) = (i+1)\ \%\ T$
$rh(i,2) = (i+2)\ \%\ T$
$rh(i,3) = (i+3)\ \%\ T$

→ Uma variação dessa técnica é usar uma permutação aleatória de 0 até $T-1$.

$$(p_0, p_1, p_2, ..., p_{t+1}) \text{ e define o } rh(j) \text{ como } (h(r)+pj)\ \%\ T$$

→ Uma terceira abordagem é $rh(j) = (h(r)+j^2)\ \%\ T$.
→ Uma quarta opção é $rh(i,k) = (i+h_{key})\ \%\ T$.
↪ $h_{key} = (1+h(k))\ \%\ T$.

– Aglomeração Secundária
→ chaves com o mesmo valor de *hash*, isto é, $k_1 \neq^* k_2$ e $h(k_1) = h(k_2)$ tendem a seguir o mesmo *rehash* até serem inseridas.
→ uma forma de eliminar a aglomeração secundária é chamada de *duplo hash* e usa duas funções *hash* h_1 e h_2 usada na fase inicial, isto é, para determinar a posição da chave na tabela. Se a posição estiver ocupada o *rehash* fica dessa forma:

$$rh(i,k) = (1+h_2(k))\ \%\ T$$

2.2 Tabela hash ordenada

A idéia geral é manter as chaves que colidirem na mesma posição em ordem decrescente.

```
1: int insert (int key)
2: int i, j, tempKey, nKey
3: bool first
4: i = h(key)
5: nKey = key
6: first = true
7: while Table[i] > nKey do
8:     i = rh(i)
9: end while
10: int tk = Table[i]
11: while ((tk is not −1) and (tk is not nKey) do
12:     tempKey = Table[i]
13:     if tempKey < nKey then
14:         Table[i] = nKey
15:         nKey = tempKey
16:         if first then
17:             j = i
18:             first = false
19:         end if
20:     end if
21:     i = rh(i)
22:     tk = Table[i]
23: end while
24: if tk == −1 then
25:     Table[i] = nKey
26: end if
27: if first then
28:     return i
29: else
30:     return j
31: end if
```

O que permite que as chaves fiquem ordenadas e que a busca seja bem-sucedida é o if $(13 - 20)$. Com ele as chaves só serão movidas se o elemento a ser inserido for maior que a chave em que está localizado o seu hash.

3 Notas

→ O que está de vermelho com asterisco* precisa ver se está correto.
→ O que está somente de vermelho é para incluir no arquivo.