Pesquisa em Memória Secundária*

Última alteração: 10 de Outubro de 2006

*Transparências elaboradas por Wagner Meira Jr, Flávia Peligrinelli Ribeiro, Nívio Ziviani e Charles Ornelas, Leonardo Rocha, Leonardo Mata

Introdução

- **Pesquisa em memória secundária**: arquivos contém mais registros do que a memória interna pode armazenar.
- Custo para acessar um registro é algumas ordens de grandeza maior do que o custo de processamento na memória primária.
- Medida de complexidade: custo de trasferir dados entre a memória principal e secundária (minimizar o número de transferências).
- Memórias secundárias: apenas um registro pode ser acessado em um dado momento (acesso seqüencial).
- Memórias primárias: acesso a qualquer registro de um arquivo a um custo uniforme (acesso direto).
- Em um método eficiente de pesquisa, o aspecto sistema de computação é importante.
- As características da arquitetura e do sistema operacional da máquina tornam os métodos de pesquisa dependentes de parâmetros que afetam seus desempenhos.

Modelo de Computação para Memória Secundária - Memória Virtual

- Normalmente implementado como uma função do sistema operacional.
- Modelo de armazenamento em dois níveis, devido à necessidade de grandes quantidades de memória e o alto custo da memória principal.
- Uso de uma pequena quantidade de memória principal e uma grande quantidade de memória secundária.
- Programador pode endereçar grandes quantidades de dados, deixando para o sistema a responsabilidade de trasferir o dado da memória secundária para a principal.
- Boa estratégia para algoritmos com pequena localidade de referência.
- Organização do fluxo entre a memória principal e secundária é extremamente importante.

Memória Virtual

- Organização de fluxo → transformar o endereço usado pelo programador na localização física de memória correspondente.
- *Espaço de Endereçamento* → endereços usados pelo programador.
- *Espaço de Memória* → localizações de memória no computador.
- O espaço de endereçamento N e o espaço de memória M podem ser vistos como um mapeamento de endereços do tipo: $f : N \to M$.
- O mapeamento permite ao programador usar um espaço de endereçamento que pode ser maior que o espaço de memória primária disponível.

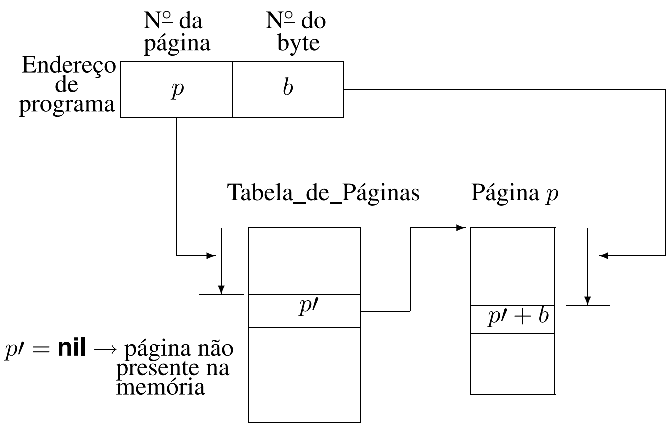
Memória Virtual: Sistema de Paginação

- O espaço de endereçamento é dividido em páginas de tamanho igual, em geral, múltiplos de 512 *bytes*.
- A memória principal é dividida em molduras de páginas de tamanho igual.
- As molduras de páginas contêm algumas páginas ativas enquanto o restante das páginas estão residentes em memória secundária (páginas inativas).
- O mecanismo possui duas funções:
  1. Mapeamento de endereços → determinar qual página um programa está endereçando, encontrar a moldura, se existir, que contenha a página.
  2. Transferência de páginas → transferir páginas da memória secundária para a memória primária e transferí-las de volta para a memória secundária quando não estão mais sendo utilizadas.

Memória Virtual: Sistema de Paginação

- Endereçamento da página → uma parte dos bits é interpretada como um número de página e a outra parte como o número do byte dentro da página (*offset*).
- Mapeamento de endereços → realizado através de uma Tabela de Páginas.
  - a p-ésima entrada contém a localização p' da Moldura de Página contendo a página número p desde que esteja na memória principal.
- O mapeamento de endereços é: $f(e) = f(p, b) = p' + b$, onde e é o endereço do programa, p é o número da página e b o número do byte.

Memória Virtual: Mapeamento de Endereços



Memória Virtual: Reposição de Páginas

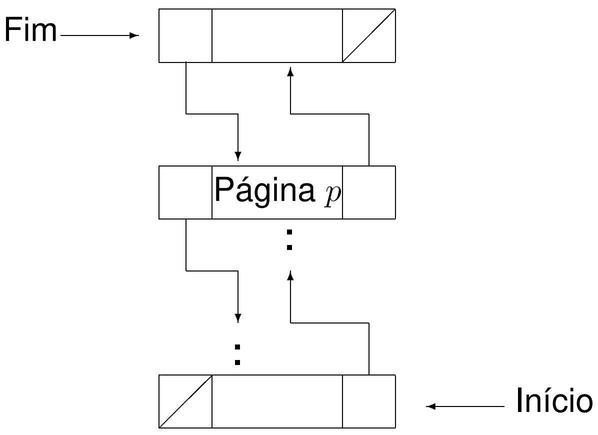
- Se não houver uma moldura de página vazia → uma página deverá ser removida da memória principal.
- Ideal → remover a página que não será referenciada pelo período de tempo mais longo no futuro.
  - tentamos inferir o futuro a partir do comportamento passado.

Memória Virtual: Políticas de Reposição de Páginas

- **Menos Recentemente Utilizada (LRU):**
  - um dos algoritmos mais utilizados,
  - remove a página menos recentemente utilizada,
  - parte do princípio que o comportamento futuro deve seguir o passado recente.
- **Menos Freqüentemente Utilizada (LFU):**
  - remove a página menos feqüentemente utilizada,
  - inconveniente: uma página recentemente trazida da memória secundária tem um baixo número de acessos registrados e pode ser removida.
- **Ordem de Chegada (FIFO):**
  - remove a página que está residente há mais tempo,
  - algoritmo mais simples e barato de manter,
  - desvantagem: ignora o fato de que a página mais antiga pode ser a mais referenciada.

Memória Virtual: Política LRU

- Toda vez que uma página é utilizada ela é removida para o fim da fila.
- A página que está no início da fila é a página LRU.
- Quando uma nova página é trazida da memória secundária ela deve ser colocada na moldura que contém a página LRU.



Memória Virtual: Estrutura de Dados

```
package cap6.umtipo;
class Registro {
  private short chave;
  // Outros componentes e métodos de um registro
}
class Endereco {
  private short p;
  private byte b; // b ∈ [0, itensPorPagina − 1]
  // Métodos para operar com um endereço
}
class Item {
  private Registro reg;
  private Endereco esq, dir;
  // Métodos para operar com um item
}
public class Pagina {
  private Item pagina[];
  public Pagina (byte itensPorPagina) {
    // itensPorPagina = tamanhoDaPagina/tamanhoDoItem
    this.pagina = new Item[itensPorPagina];
  }
  // Métodos para operar com uma página
}
```


Memória Virtual

- Em casos em que precisamos manipular mais de um arquivo ao mesmo tempo:
  - Deve-se utilizar os mecanismos de **H**erança e **P**olimorfismo de Java que permitem que uma página possa ser definida como vários tipos diferentes.
  - A fila de molduras é única → cada moldura deve ter indicado o arquivo a que se refere aquela página.

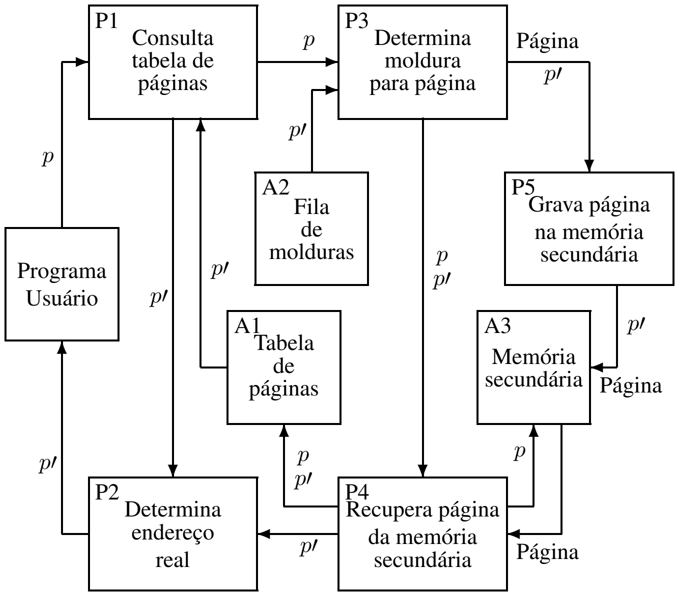
```
package cap6.variostipos;
public abstract class Pagina {
  // Componentes e métodos de uma página
}
class PaginaA extends Pagina {
  // Componentes e métodos de uma página do tipo A
}
class PaginaB extends Pagina {
  // Componentes e métodos de uma página do tipo B
}
class PaginaC extends Pagina {
  // Componentes e métodos de uma página do tipo C
}
```

Memória Virtual

- Procedimentos para comunicação com o sistema de paginação:
  - $obtemRegistro \rightarrow$ torna disponível um registro.
  - $escreveRegistro \rightarrow$ permite criar ou alterar o conteúdo de um registro.
  - $descarregaPaginas \rightarrow$ varre a fila de molduras para atualizar na memória secundária todas as páginas que tenham sido modificadas.

Memória Virtual - Transformação do Endereço Virtual para Real

- Quadrados $\rightarrow$ resultados de processos ou arquivos.
- Retângulos $\rightarrow$ processos transformadores de informação.



Acesso Seqüencial Indexado

- Utiliza o princípio da pesquisa seqüencial $\rightarrow$ cada registro é lido seqüencialmente até encontrar uma chave maior ou igual a chave de pesquisa.
- Providências necessárias para aumentar a eficiência:
  - o arquivo deve ser mantido ordenado pelo campo chave do registro,
  - um arquivo de índices contendo pares de valores $< x, p >$ deve ser criado, onde x representa uma chave e p representa o endereço da página na qual o primeiro registro contém a chave x.
  - Estrutura de um arquivo seqüencial indexado para um conjunto de 15 registros:

3	14	25	41
1	2	3	4

1 | 3 5 7 11 | 2 | 14 17 20 21 | 3 | 25 29 32 36 | 4 | 41 44 48 |

Acesso Seqüencial Indexado: Disco Magnético

- Dividido em círculos concêntricos (trilhas).
- Cilindro $\rightarrow$ todas as trilhas verticalmente alinhadas e que possuem o mesmo diâmetro.
- Latência rotacional $\rightarrow$ tempo necessário para que o início do bloco contendo o registro a ser lido passe pela cabeça de leitura/gravação.
- Tempo de busca (*seek time*) $\rightarrow$ tempo necessário para que o mecanismo de acesso desloque de uma trilha para outra (maior parte do custo para acessar dados).
- Acesso seqüencial indexado = acesso indexado + organização seqüencial,
- Aproveitando características do disco magnético e procurando minimizar o número de deslocamentos do mecanismo de acesso $\rightarrow$ esquema de índices de cilindros e de páginas.

Acesso Seqüencial Indexado: Disco Magnético

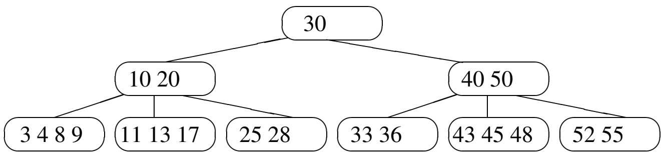
- Para localizar o registro que contenha uma chave de pesquisa são necessários os seguintes passos:
  1. localize o cilindro correspondente à chave de pesquisa no índice de cilindros;
  2. desloque o mecanismo de acesso até o cilindro correspondente;
  3. leia a página que contém o índice de páginas daquele cilindro;
  4. leia a página de dados que contém o registro desejado.

Acesso Seqüencial Indexado: Discos Óticos de Apenas-Leitura (CD-ROM)

- Grande capacidade de armazenamento (600 MB) e baixo custo para o usuário final.
- Informação armazenada é estática.
- A eficiência na recuperação dos dados é afetada pela localização dos dados no disco e pela seqüência com que são acessados.
- Velocidade linear constante $\rightarrow$ trilhas possuem capacidade variável e tempo de latência rotacional varia de trilha para trilha.
- A trilha tem forma de uma espiral contínua.
- Tempo de busca: acesso a trilhas mais distantes demanda mais tempo que no disco magnético. Há necessidade de deslocamento do mecanismo de acesso e mudanças na rotação do disco.
- Varredura estática: acessa conjunto de trilhas vizinhas sem deslocar mecanismo de leitura.
- Estrutura seqüencial implementada mantendo-se um índice de cilindros na memória principal.

Árvores B

- Árvores n-árias: mais de um registro por nodo.
- Em uma árvore B de ordem m:
  - página raiz: 1 e $2m$ registros.
  - demais páginas: no mínimo m registros e $m + 1$ descendentes e no máximo $2m$ registros e $2m + 1$ descendentes.
  - páginas folhas: aparecem todas no mesmo nível.
- Os registros aparecem em ordem crescente da esquerda para a direita.
- Extensão natural da árvore binária de pesquisa.
- Árvore B de ordem $m = 2$ com três níveis:



Árvores B - Estrutura e operações do dicionário para árvore B

- A estrutura de dados árvore B será utilizada para implementar o tipo abstrato de dados Dicionário e suas operações: $inicializa$, $pesquisa$, $insere$ e $retira$.
- A operação $inicializa$ é implementada pelo construtor da classe $ArvoreB$. As demais operações são descritas a seguir.
- A operação $pesquisa$ é implementada por um método privado sobrecarregado. Este método é semelhante ao método $pesquisa$ para a árvore binária de pesquisa.

Árvores B - Estrutura e operações do dicionário para árvore B

```
package cap6;
import cap4.Item; // vide Programa do capítulo 4
public class ArvoreB {
  private static class Pagina {
    int n; Item r[]; Pagina p[];
    public Pagina (int mm) {
      this.n = 0; this.r = new Item[mm];
      this.p = new Pagina[mm+1];
    }
  }
  private Pagina raiz;
  private int m, mm;
  // Entra aqui o método privado da transparência 21
  public ArvoreB (int m) {
    this.raiz = null; this.m = m; this.mm = 2*m;
  }
  public Item pesquisa (Item reg) {
    return this.pesquisa (reg, this.raiz);
  }
  public void insere (Item reg) { vide transparências 24 e 25 }

  public void retira (Item reg) { vide transparências 30, 31 e 32 }
}
```

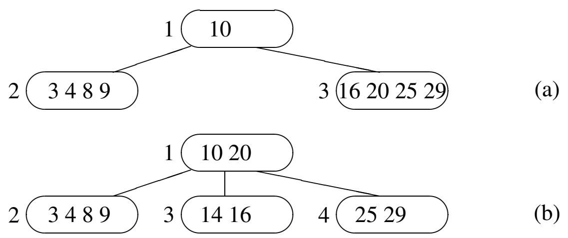

Árvores B - Método para pesquisar na árvore B

```
private Item pesquisa (Item reg, Pagina ap) {
  if (ap == null) return null; // Registro não encontrado
  else {
    int i = 0;
    while ((i < ap.n−1) && (reg.compara (ap.r[i]) > 0)) i++;
    if (reg.compara (ap.r[i]) == 0) return ap.r[i];
    else if (reg.compara (ap.r[i]) < 0)
           return pesquisa (reg, ap.p[i]);
    else return pesquisa (reg, ap.p[i+1]);
  }
}
```


Árvores B - Inserção

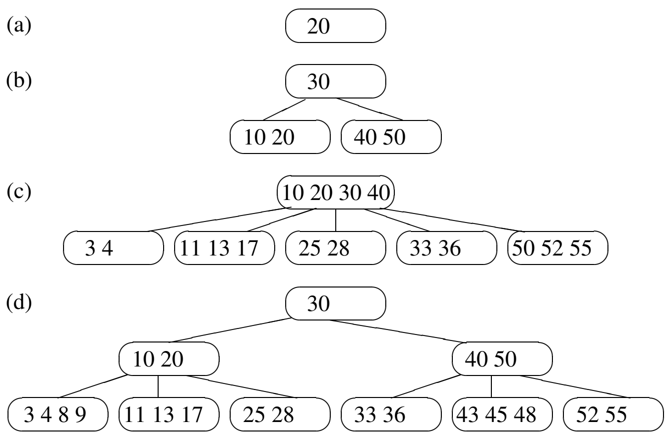
1. Localizar a página apropriada aonde o regisro deve ser inserido.
2. Se o registro a ser inserido encontra uma página com menos de $2m$ registros, o processo de inserção fica limitado à página.
3. Se o registro a ser inserido encontra uma página cheia, é criada uma nova página, no caso da página pai estar cheia o processo de divisão se propaga.

Exemplo: Inserindo o registro com chave 14.



Árvores B - Inserção

Exemplo de inserção das chaves: 20, 10, 40, 50, 30, 55, 3, 11, 4, 28, 36, 33, 52, 17, 25, 13, 45, 9, 43, 8 e 48



Árvores B - Método $insereNaPagina$

```
private void insereNaPagina
                 (Pagina ap, Item reg, Pagina apDir) {
  int k = ap.n − 1;
  while ((k >= 0) && (reg.compara (ap.r[k]) < 0)) {
    ap.r[k+1] = ap.r[k]; ap.p[k+2] = ap.p[k+1]; k−−;
  }
  ap.r[k+1] = reg; ap.p[k+2] = apDir; ap.n++;
}
```


Árvores B - Refinamento final do método $insere$

```
public void insere (Item reg) {
  Item regRetorno[] = new Item[1];
  boolean cresceu[] = new boolean[1];
  Pagina apRetorno = this.insere (reg, this.raiz, regRetorno, cresceu);
  if (cresceu[0]) {
    Pagina apTemp = new Pagina(this.mm);
    apTemp.r[0] = regRetorno[0];
    apTemp.p[0] = this.raiz;
    apTemp.p[1] = apRetorno;
    this.raiz = apTemp; this.raiz.n++;
  } else this.raiz = apRetorno;
}
private Pagina insere (Item reg, Pagina ap, Item[] regRetorno,
                       boolean[] cresceu) {
  Pagina apRetorno = null;
  if (ap == null) { cresceu[0] = true; regRetorno[0] = reg; }
  else {
    int i = 0;
    while ((i < ap.n−1) && (reg.compara (ap.r[i]) > 0)) i++;
    if (reg.compara (ap.r[i]) == 0) {
      System.out.println ("Erro: Registro ja existente");
      cresceu[0] = false;
    }
    else {
      if (reg.compara (ap.r[i]) > 0) i++;
      apRetorno = insere (reg, ap.p[i], regRetorno, cresceu);
      if (cresceu[0])
        if (ap.n < this.mm) { // Página tem espaço
          this.insereNaPagina (ap, regRetorno[0], apRetorno);
          cresceu[0] = false; apRetorno = ap;
        }

        // Continua na próxima transparência
```


Árvores B - Refinamento final do método $insere$

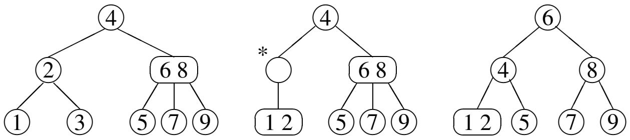
```
        else { // Overflow: Página tem que ser dividida
          Pagina apTemp = new Pagina (this.mm); apTemp.p[0] = null;
          if (i <= this.m) {
            this.insereNaPagina (apTemp,ap.r[this.mm−1],ap.p[this.mm]);
            ap.n−−;
            this.insereNaPagina (ap, regRetorno[0], apRetorno);
          } else this.insereNaPagina (apTemp, regRetorno[0], apRetorno);
          for (int j = this.m+1; j < this.mm; j++) {
            this.insereNaPagina (apTemp, ap.r[j], ap.p[j+1]);
            ap.p[j+1] = null; // transfere a posse da memória
          }
          ap.n = this.m; apTemp.p[0] = ap.p[this.m+1];
          regRetorno[0] = ap.r[this.m]; apRetorno = apTemp;
        }
      }
    }
  return (cresceu[0] ? apRetorno : ap);
}
```


Árvores B - Remoção

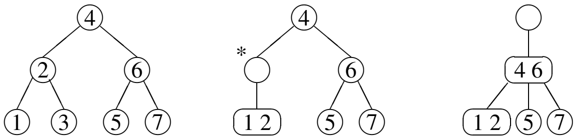
- Página com o registro a ser retirado é folha:
  1. retira-se o registro,
  2. se a página não possui pelo menos de m registros, a propriedade da árvore B é violada. Pega-se um registro emprestado da página vizinha. Se não existir registros suficientes na página vizinha, as duas páginas devem ser fundidas em uma só.
- Pagina com o registro não é folha:
  1. o registro a ser retirado deve ser primeiramente substituído por um registro contendo uma chave adjacente.

Árvores B - Remoção

Exemplo: Retirando a chave 3.



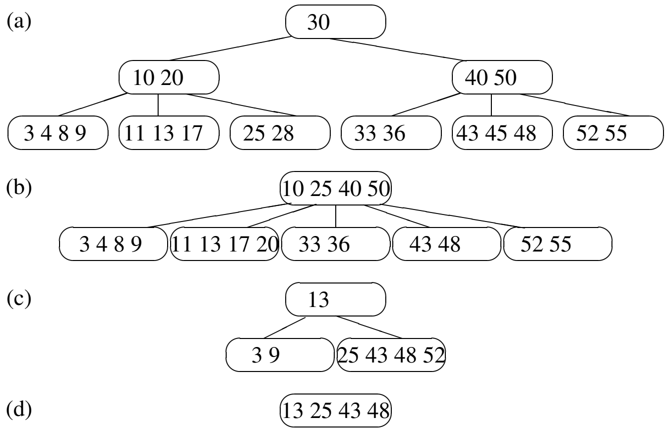
(a) Página vizinha possui mais do que m registros



(b) Página vizinha possui exatamente m registros

Árvores B - Remoção

Exemplo de remoção das chaves 45 30 28; 50 8 10 4 20 40 55 17 33 11 36; 3 9 52.



Árvores B - Operação *retira*

```
public void retira (Item reg) {
  boolean diminuiu[] = new boolean[1];
  this.raiz = this.retira (reg, this.raiz, diminuiu);
  if (diminuiu[0] && (this.raiz.n == 0)) { // Árvore diminui na altura
    this.raiz = this.raiz.p[0];
  }
}
private Pagina retira (Item reg, Pagina ap, boolean[] diminuiu) {
  if (ap == null) {
    System.out.println ("Erro: Registro nao encontrado");
    diminuiu[0] = false;
  }
  else {
    int ind = 0;
    while ((ind < ap.n-1) && (reg.compara (ap.r[ind]) > 0)) ind++;
    if (reg.compara (ap.r[ind]) == 0) { // achou
      if (ap.p[ind] == null) { // Página folha
        ap.n--; diminuiu[0] = ap.n < this.m;
        for (int j = ind; j < ap.n; j++) {
          ap.r[j] = ap.r[j+1]; ap.p[j] = ap.p[j+1];
        }
        ap.p[ap.n] = ap.p[ap.n+1];
        ap.p[ap.n+1] = null; // transfere a posse da memória
      }
      else { // Página não é folha: trocar com antecessor
        diminuiu[0] = antecessor (ap, ind, ap.p[ind]);
        if (diminuiu[0]) diminuiu[0] = reconstitui (ap.p[ind], ap, ind);
      }
    }
    else { // não achou
      if (reg.compara (ap.r[ind]) > 0) ind++;
      ap.p[ind] = retira (reg, ap.p[ind], diminuiu);
      if (diminuiu[0]) diminuiu[0] = reconstitui (ap.p[ind], ap, ind);
    }
  }
  return ap;
}
```


Árvores B - Método *antecessor* utilizado no método *retira*

```
private boolean antecessor(Pagina ap, int ind, Pagina apPai) {
  boolean diminuiu = true;
  if (apPai.p[apPai.n] != null) {
    diminuiu = antecessor (ap, ind, apPai.p[apPai.n]);
    if (diminuiu)
      diminuiu=reconstitui (apPai.p[apPai.n], apPai, apPai.n);
  }
  else {
    ap.r[ind] = apPai.r[--apPai.n];
    diminuiu = apPai.n < this.m;
  }
  return diminuiu;
}
```

Árvores B - Método *reconstitui* utilizado no método *retira*

```
private boolean reconstitui (Pagina apPag, Pagina apPai, int posPai) {
  boolean diminuiu = true;
  if (posPai < apPai.n) { // aux = Página à direita de apPag
    Pagina aux = apPai.p[posPai+1];
    int dispAux = (aux.n - this.m + 1)/2;
    apPag.r[apPag.n++] = apPai.r[posPai]; apPag.p[apPag.n] = aux.p[0];
    aux.p[0] = null; // transfere a posse da memória
    if (dispAux > 0) { // Existe folga: transfere de aux para apPag
      for (int j = 0; j < dispAux - 1; j++) {
        this.insereNaPagina (apPag, aux.r[j], aux.p[j+1]);
        aux.p[j+1] = null; // transfere a posse da memória
      }
      apPai.r[posPai] = aux.r[dispAux - 1];
      aux.n = aux.n - dispAux;
      for (int j = 0;  j < aux.n; j++) aux.r[j] = aux.r[j+dispAux];
      for (int j = 0; j <= aux.n; j++) aux.p[j] = aux.p[j+dispAux];
      aux.p[aux.n+dispAux] = null; // transfere a posse da memória
      diminuiu = false;
    }
    else { // Fusão: intercala aux em apPag e libera aux
      for (int j = 0; j < this.m; j++) {
        this.insereNaPagina (apPag, aux.r[j], aux.p[j+1]);
        aux.p[j+1] = null; // transfere a posse da memória
      }
      aux = apPai.p[posPai+1] = null; // libera aux
      for (int j = posPai; j < apPai.n-1; j++) {
        apPai.r[j] = apPai.r[j+1]; apPai.p[j+1] = apPai.p[j+2];
      }
      apPai.p[apPai.n--] = null; // transfere a posse da memória
      diminuiu = apPai.n < this.m;
    }
  }

      // Continua na próxima transparência
```


Árvores B - Método *reconstitui* utilizado no método *retira*

```
  else { // aux = Página à esquerda de apPag
    Pagina aux = apPai.p[posPai-1];
    int dispAux = (aux.n - this.m + 1)/2;
    for (int j = apPag.n-1; j >= 0; j--) apPag.r[j+1] = apPag.r[j];
    apPag.r[0] = apPai.r[posPai-1];
    for (int j = apPag.n; j >= 0; j--) apPag.p[j+1] = apPag.p[j];
    apPag.n++;
    if (dispAux > 0) { // Existe folga: transfere de aux para apPag
      for (int j = 0; j < dispAux - 1; j++) {
        this.insereNaPagina (apPag, aux.r[aux.n-j-1], aux.p[aux.n-j]);
        aux.p[aux.n-j] = null; // transfere a posse da memória
      }
      apPag.p[0] = aux.p[aux.n - dispAux + 1];
      aux.p[aux.n - dispAux + 1] = null; // transfere a posse da memória
      apPai.r[posPai-1] = aux.r[aux.n - dispAux];
      aux.n = aux.n - dispAux; diminuiu = false;
    }
    else { // Fusão: intercala apPag em aux e libera apPag
      for (int j = 0; j < this.m; j++) {
        this.insereNaPagina (aux, apPag.r[j], apPag.p[j+1]);
        apPag.p[j+1] = null; // transfere a posse da memória
      }
      apPag = null; // libera apPag
      apPai.p[apPai.n--] = null; // transfere a posse da memória
      diminuiu = apPai.n < this.m;
    }
  }
  return diminuiu;
}
```


Árvores B* - Estrutura e operações do dicionário para árvore B*

```
package cap6;
import cap4.Item; // vide Programa do capítulo 4
public class ArvoreBEstrela {
  private static abstract class Pagina {
    int n; Item chaves[];
  }
  private static class PaginaInt extends Pagina {
    Pagina p[];
    public PaginaInt (int mm) {
      this.n = 0; this.chaves = new Item[mm];
      this.p = new Pagina[mm+1];
    }
  }
  private static class PaginaExt extends Pagina {
    Object registros[];
    public PaginaExt (int mm2) {
      this.n = 0; this.chaves = new Item[mm2];
      this.registros = new Object[mm2];
    }
  }
  private Pagina raiz;
  private int mm, mm2;

  // Entram aqui os métodos privados apresentados na transparência 36
  public ArvoreBEstrela (int mm, int mm2) {
    this.raiz = null; this.mm = mm; this.mm2 = mm2;
  }
  public Object pesquisa (Item chave) {
    return this.pesquisa (chave, this.raiz);
  }
}
```


Árvores B* - Pesquisa

- Semelhante à pesquisa em árvore B,
- A pesquisa sempre leva a uma página folha,
- A pesquisa não pára se a chave procurada for encontrada em uma página índice. O apontador da direita é seguido até que se encontre uma página folha.

Árvores B* - Método para pesquisar na árvore B*

```
private Object pesquisa (Item chave, Pagina ap) {
  if (ap == null) return null; // Registro não encontrado
  else {
    if (this.eInterna (ap)) {
      int i = 0; PaginaInt aux = (PaginaInt)ap;
      while ((i < aux.n−1) && (chave.compara (aux.chaves[i]) > 0)) i++;
      if (chave.compara (aux.chaves[i]) < 0)
        return pesquisa (chave, aux.p[i]);
      else return pesquisa (chave, aux.p[i+1]);
    }
    else {
      int i = 0; PaginaExt aux = (PaginaExt)ap;
      while ((i < aux.n−1) && (chave.compara (aux.chaves[i]) > 0)) i++;
      if (chave.compara (aux.chaves[i]) == 0) return aux.registros[i];
      return null; // Registro não encontrado
    }
  }
}

private boolean eInterna (Pagina ap) {
  Class classe = ap.getClass ();
  return classe.getName().equals(PaginaInt.class.getName());
}
```


Árvores B* - Inserção e Remoção

- Inserção na árvore B*
  - Semelhante à inserção na árvore B,
  - Diferença: quando uma folha é dividida em duas, o algoritmo promove uma cópia da chave que pertence ao registro do meio para a página pai no nível anterior, retendo o registro do meio na página folha da direita.
- Remoção na árvore B*
  - Relativamente mais simples que em uma árvore B,
  - Todos os registros são folhas,
  - Desde que a folha fique com pelo menos metade dos registros, as páginas dos índices não precisam ser modificadas, mesmo se uma cópia da chave que pertence ao registro a ser retirado esteja no índice.

Acesso Concorrente em Árvore B*

- Acesso simultâneo a banco de dados por mais de um usuário.
- Concorrência aumenta a utilização e melhora o tempo de resposta do sistema.
- O uso de árvores B* nesses sistemas deve permitir o processamento simultâneo de várias solicitações diferentes.
- Necessidade de criar mecanismos chamados protocolos para garantir a integridade tanto dos dados quanto da estrutura.
- Página segura: não há possibilidade de modificações na estrutura da árvore como conseqüência de inserção ou remoção.
  - inserção → página segura se o número de chaves é igual a $2m$,
  - remoção → página segura se o número de chaves é maior que m.
- Os algoritmos para acesso concorrente fazem uso dessa propriedade para aumentar o nível de concorrência.

Acesso Concorrente em Árvore B* - Protocolos de Travamentos

- Quando uma página é lida, a operação de recuperação a trava, assim, outros processos, não podem interferir com a página.
- A pesquisa continua em direção ao nível seguinte e a trava é liberada para que outros processos possam ler a página .
- Processo leitor → executa uma operação de recuperação
- Processo modificador → executa uma operação de inserção ou retirada.
- Dois tipos de travamento:
  - Travamento para leitura → permite um ou mais leitores acessarem os dados, mas não permite inserção ou retirada.
  - Travamento exclusivo → nenhum outro processo pode operar na página e permite qualquer tipo de operação na página.

Árvore B - Considerações Práticas

- Simples, fácil manutenção, eficiente e versátil.
- Permite acesso seqüencial eficiente.
- Custo para recuperar, inserir e retirar registros do arquivo é logaritmico.
- Espaço utilizado é, no mínimo 50% do espaço reservado para o arquivo,
- Emprego onde o acesso concorrente ao banco de dados é necessário, é viável e relativamente simples de ser implementado.
- Inserção e retirada de registros sempre deixam a árvore balanceada.
- Uma árvore B de ordem m com N registros contém no máximo cerca de $log_{m+1} N$ páginas.

Árvore B - Considerações Práticas

- Limites para a altura máxima e mínima de uma árvore B de ordem m com N registros: $\log_{2m+1}(N+1) \leq altura \leq 1 + \log_{m+1}\left(\frac{N+1}{2}\right)$
- Custo para processar uma operação de recuperação de um registro cresce com o logaritmo base m do tamanho do arquivo.
- Altura esperada: não é conhecida analiticamente.
- Há uma conjectura proposta a partir do cálculo analítico do número esperado de páginas para os quatro primeiros níveis (das folha em direção à raiz) de uma **árvore 2-3** (árvore B de ordem $m = 1$).
- Conjetura: a altura esperada de uma árvore 2-3 **randômica** com N chaves é $\overline{h}(N) \approx \log_{7/3}(N+1)$.

Árvores B Randômicas - Outras Medidas de Complexidade

- A utilização de memória é cerca de $\ln 2$.
  - Páginas ocupam $\approx$ 69% da área reservada após N inserções randômicas em uma árvore B inicialmente vazia.
- No momento da inserção, a operação mais cara é a partição da página quando ela passa a ter mais do que $2m$ chaves. Envolve:
  - Criação de nova página, rearranjo das chaves e inserção da chave do meio na página pai localizada no nível acima.
  - $Pr\{$j partições$\}$: probabilidade de que j partições ocorram durante a N-ésima inserção randômica.
  - Árvore 2-3: $Pr\{$0 partições$\} = \frac{4}{7}$, $Pr\{$1 ou mais partições$\} = \frac{3}{7}\cdot$
  - Árvore B de ordem m: $Pr\{$0 partições$\} = 1 - \frac{1}{(2\ln 2)m} + O(m^{-2})$, $Pr\{$1 ou + partições$\} = \frac{1}{(2\ln 2)m} + O(m^{-2})$.
  - Árvore B de ordem $m = 70$: 99% das vezes nada acontece em termos de partições durante uma inserção.

Árvores B Randômicas - Acesso Concorrente

- Foi proposta uma técnica de aplicar um travamento na *página segura mais profunda* (Psmp) no caminho de inserção.
- Uma página é **segura** se ela contém menos do que $2m$ chaves.
- Uma página segura é a mais profunda se não existir outra página segura abaixo dela.
- Já que o travamento da página impede o acesso de outros processos, é interessante saber qual é a probabilidade de que a página segura mais profunda esteja no primeiro nível.
- Árvore 2-3: $Pr\{$Psmp esteja no $1^{\underline{o}}$ nível$\} = \frac{4}{7}$, $Pr\{$Psmp esteja acima do $1^{\underline{o}}$ nível$\} = \frac{3}{7}\cdot$
- Árvore B de ordem m: $Pr\{$Psmp esteja no $1^{\underline{o}}$ nível$\} = 1 - \frac{1}{(2\ln 2)m} + O(m^{-2})$, $Pr\{$Psmp esteja acima do $1^{\underline{o}}$ nível$\} = \frac{3}{7} = \frac{1}{(2\ln 2)m} + O(m^{-2})$.

Árvores B Randômicas - Acesso Concorrente

- Novamente, em árvores B de ordem $m = 70$: 99% das vezes a Psmp está em uma folha. (Permite alto grau de concorrência para processos modificadores.)
- Soluções muito complicadas para permitir concorrência de operações em árvores B não trazem grandes benefícios.
- Na maioria das vezes, o travamento ocorrerá em páginas folha. (Permite alto grau de concorrência mesmo para os protocolos mais simples.)

Árvore B - Técnica de Transbordamento (ou Overflow)

- Assuma que um registro tenha de ser inserido em uma página cheia, com $2m$ registros.
- Em vez de particioná-la, olhamos primeiro para a página irmã à direita.
- Se a página irmã possui menos do que $2m$ registros, um simples rearranjo de chaves torna a partição desnecessária.
- Se a página à direita também estiver cheia ou não existir, olhamos para a página irmã à esquerda.
- Se ambas estiverem cheias, então a partição terá de ser realizada.
- Efeito da modificação: produzir uma árvore com melhor utilização de memória e uma altura esperada menor.
- Produz uma utilização de memória de cerca de 83% para uma árvore B randômica.

Árvore B - Influência do Sistema de Paginação

- O número de níveis de uma árvore B é muito pequeno (três ou quatro) se comparado com o número de molduras de páginas.
- Assim, o sistema de paginação garante que a página raiz esteja sempre na memória principal (se for adotada a política LRU).
- O esquema LRU faz também com que as páginas a serem particionadas em uma inserção estejam automaticamente disponíveis na memória principal.
- A escolha do tamanho adequado da ordem m da árvore B é geralmente feita levando em conta as características de cada computador.
- O tamanho ideal da página da árvore corresponde ao tamanho da página do sistema, e a transferência de dados entre as memórias secundária e principal é realizada pelo sistema operacional.
- Estes tamanhos variam entre 512 *bytes* e 4.096 *bytes*, em múltiplos de 512 *bytes*.