Pesquisa em Memória Secundária*

Última alteração: 31 de Agosto de 2010

*Transparências elaboradas por Wagner Meira Jr, Flávia Peligrinelli Ribeiro, Israel Guerra, Nívio Ziviani e Charles Ornelas Almeida

Conteúdo do Capítulo

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).
- 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 $\rightarrow$ transformar o endereço usado pelo programador na localização física de memória correspondente.
- *Espaço de Endereçamento* $\rightarrow$ endereços usados pelo programador.
- *Espaço de Memória* $\rightarrow$ 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: $f : N \rightarrow M$.
- O mapeamento permite ao programador usar um espaço de endereçamento que pode ser maior que o espaço de memória primária disponível.

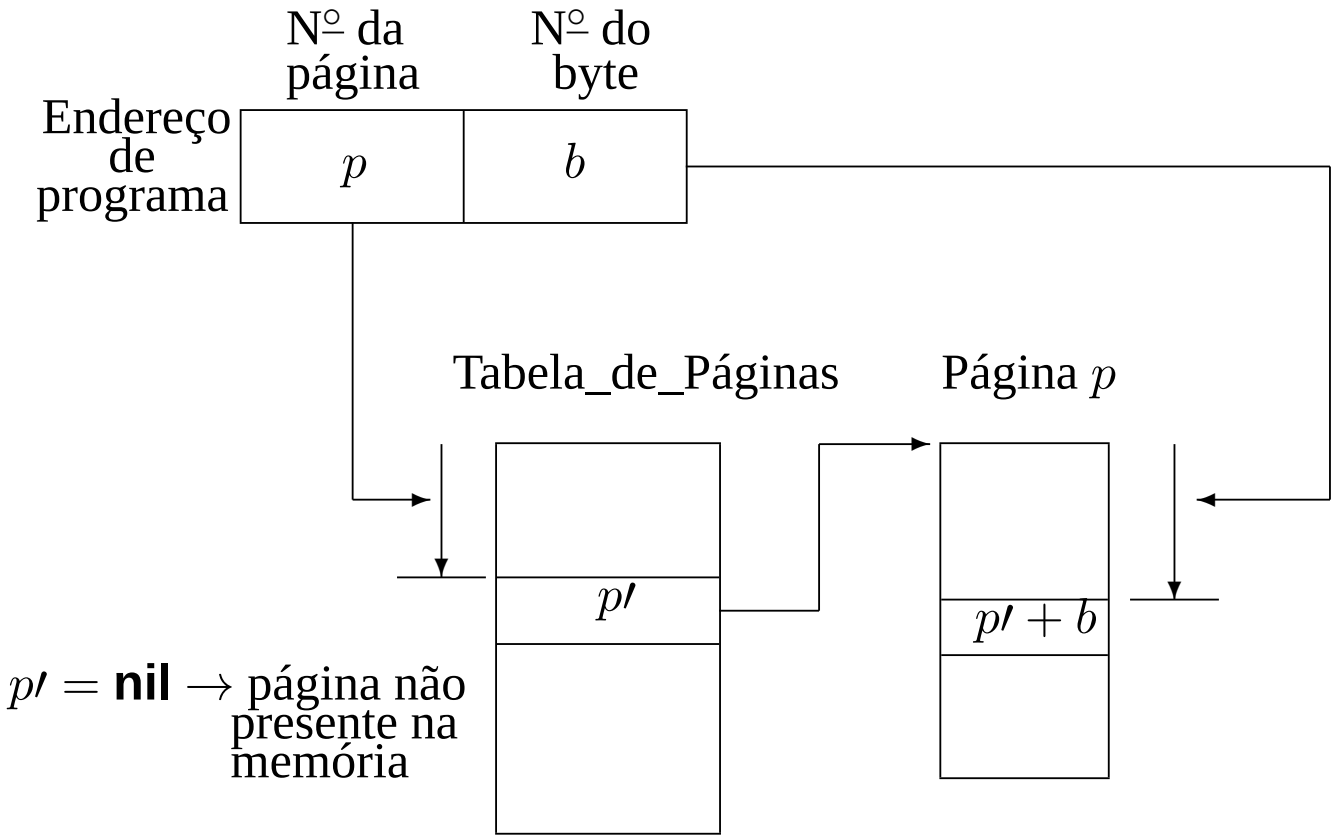
Memória Virtual: Sistema de Paginação

- O espaço de endereçamento é dividido em páginas de tamanho igual, em geral, múltiplos de 512 Kbytes.
- 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 $\rightarrow$ determinar qual página um programa está endereçando, encontrar a moldura, se existir, que contenha a página.
  2. Transferência de páginas $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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



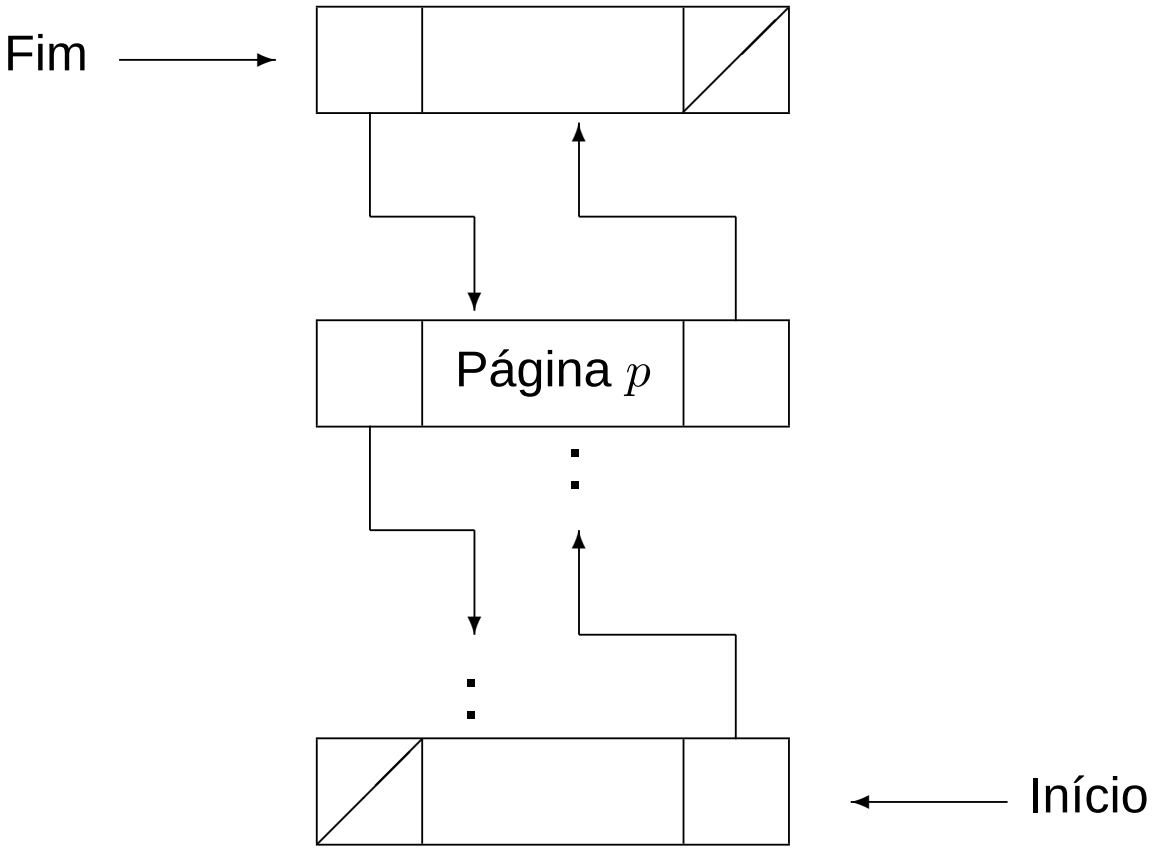
Memória Virtual: Reposição de Páginas

- Se não houver uma moldura de página vazia $\rightarrow$ uma página deverá ser removida da memória principal.
- Ideal $\rightarrow$ 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 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

```
#define TAMANHODAPAGINA   512
#define ITENSPORPAGINA    64   /* TamanhodaPagina / TamanhodoItem */

typedef struct TipoRegisto {
   TipoChave Chave;
   /* outros componentes */
} TipoRegistro;
typedef struct TipoEndereco {
   long p;
   char b;
} TipoEndereco;
typedef struct TipoItem {
   TipoRegistro Reg;
   TipoEndereco Esq, Dir;
} TipoItem;
typedef TipoItem TipoPagina[ItensPorPagina];
```

Memória Virtual

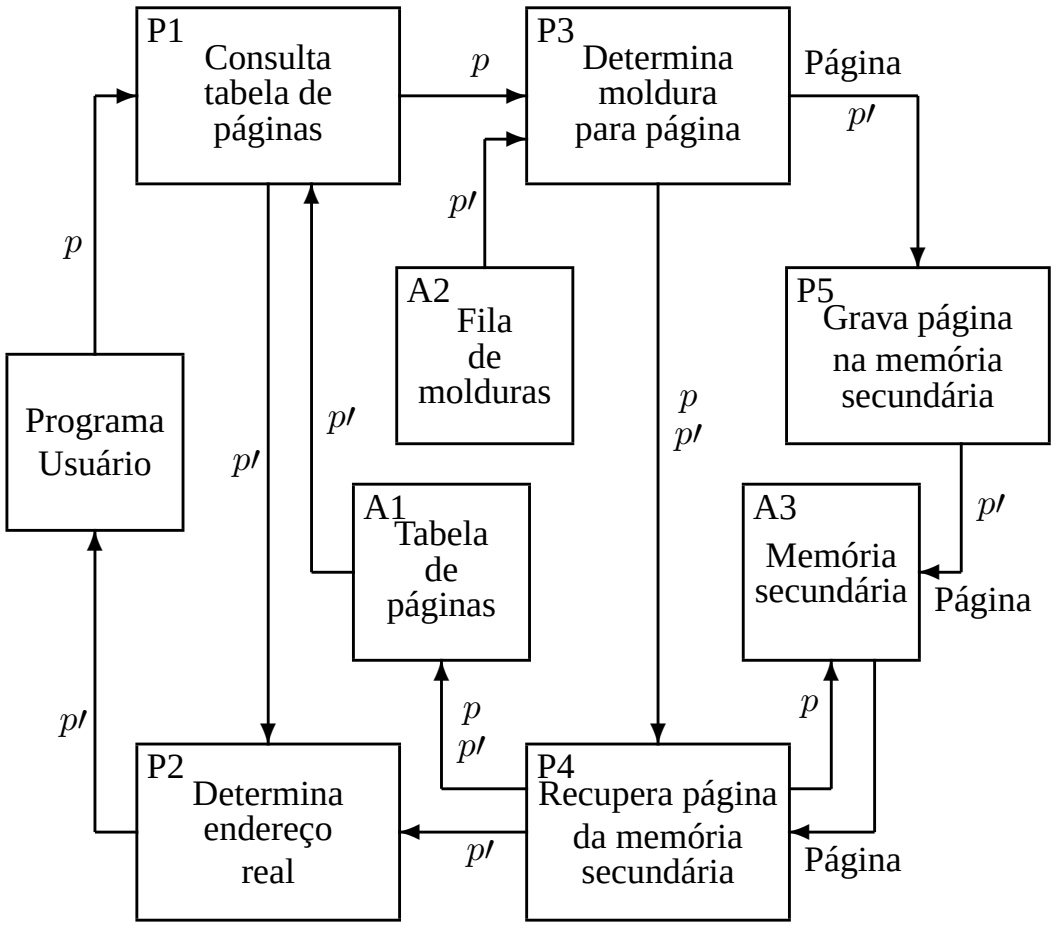
- Em casos em que precisamos manipular mais de um arquivo ao mesmo tempo:
  - A tabela de páginas para cada arquivo pode ser declarada separadamente.
  - A fila de molduras é única $\rightarrow$ cada moldura deve ter indicado o arquivo a que se refere aquela página.

```
typedef struct TipoPagina {
  char tipo; /* armazena o codigo do tipo:0,1,2 */
  union {
    TipoPaginaA Pa;
    TipoPaginaB Pb;
    TipoPaginaC Pc;
  }P;
} TipoPagina;
```

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

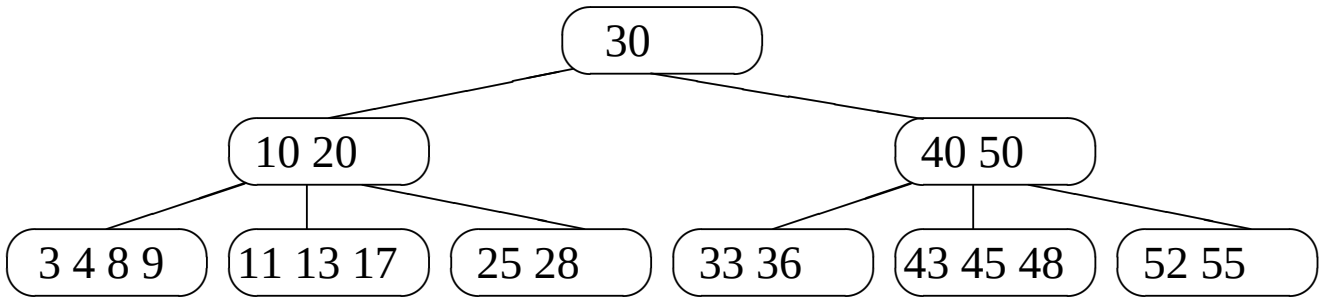
- Dividido em círculos concêntricos (trilhas).
- Cilindro $\rightarrow$ todas as trilhas verticalmente alinhadas e que possuem o mesmo diâmetro.
- Latência rotacional $\rightarrow$ tempo necessário para que o início do bloco contendo o registro a ser lido passe pela cabeça de leitura/gravação.
- Tempo 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: Discos Óticos de Apenas-Leitura (CD-ROM)

- Grande capacidade de armazenamento (600 MB) e baixo custo.
- 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.
- Registros 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 - TAD Dicionário

- Estrutura de Dados:

```
typedef long TipoChave;
typedef struct TipoRegistro {
   TipoChave Chave;
   /*outros componentes*/
} TipoRegistro;
typedef struct TipoPagina* TipoApontador;
typedef struct TipoPagina {
   short n;
   TipoRegistro r[MM];
   TipoApontador p[MM + 1];
} TipoPagina;
```

Árvores B - TAD Dicionário

- Operações:
  - Inicializa

```
void Inicializa(TipoApontador *Dicionario)
{  *Dicionario = NULL; }
```

  - Pesquisa
  - Insere
  - Remove

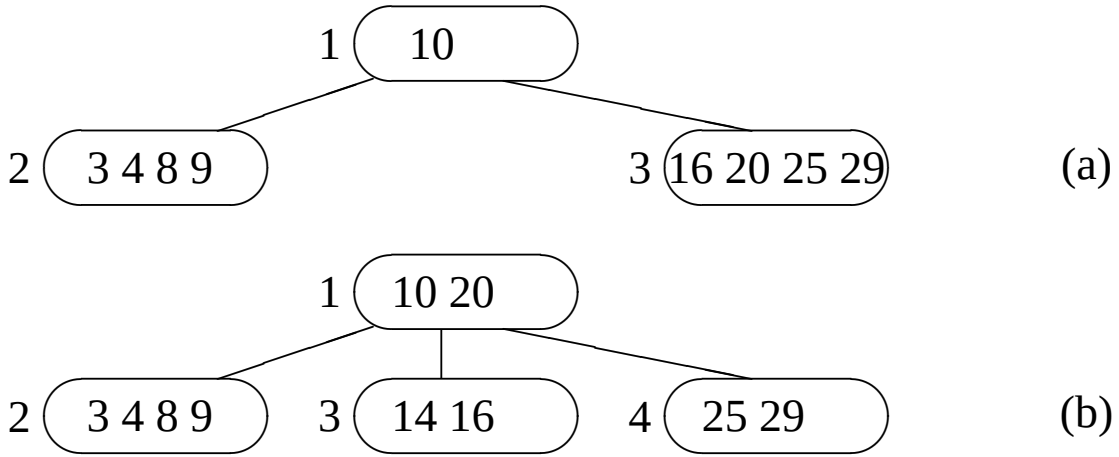
Árvores B - Pesquisa

```
void Pesquisa(TipoRegistro *x, TipoApontador Ap)
{ long i = 1;
  if (Ap == NULL)
  { printf("TipoRegistro nao esta presente na arvore\n");
    return;
  }
  while (i < Ap->n && x->Chave > Ap->r[i-1].Chave) i++;
  if (x->Chave == Ap->r[i-1].Chave)
  { *x = Ap->r[i-1];
    return;
  }
  if (x->Chave < Ap->r[i-1].Chave)
  Pesquisa(x, Ap->p[i-1]);
  else Pesquisa(x, Ap->p[i]);
}
```

Árvores B - Inserção

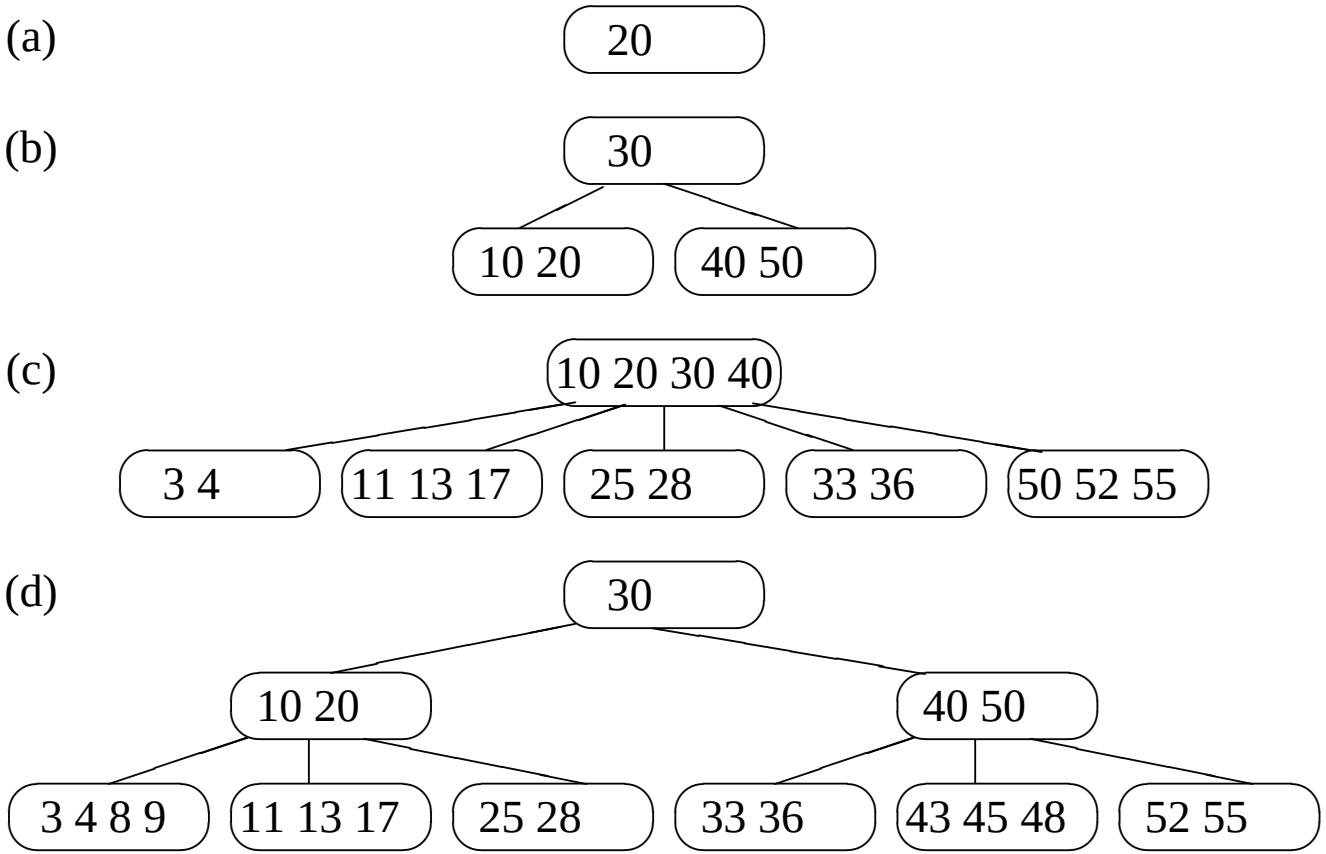
1. Localizar a página apropriada aonde o regisro deve ser inserido.
2. Se o registro a ser inserido encontra uma página com menos de $2m$ registros, o processo de inserção fica limitado à página.
3. Se o registro a ser inserido encontra uma página cheia, é criada uma nova página, no caso da página pai 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 - Primeiro refinamento do algoritmo Insere

```
void Ins(TipoRegistro Reg, TipoApontador Ap, short *Cresceu,
         TipoRegistro *RegRetorno, TipoApontador *ApRetorno)
{ long i = 1; long j; TipoApontador ApTemp;
  if (Ap == NULL)
  { *Cresceu = TRUE; Atribui Reg a RegRetorno;
    Atribui NULL a ApRetorno; return;
  }
  while (i < Ap -> n && Reg.Chave > Ap -> r[i-1].Chave) i++;
  if (Reg.Chave == Ap -> r[i-1].Chave) { printf(" Erro: Registro ja esta presente\n"); return; }
  if (Reg.Chave < Ap -> r[i-1].Chave) Ins(Reg, Ap -> p[i--], Cresceu, RegRetorno, ApRetorno);
  if (!*Cresceu) return;
  if (Numero de registros em Ap < mm)
  { Insere na pagina Ap e *Cresceu = FALSE; return; }
  /* Overflow: Pagina tem que ser dividida */
  Cria nova pagina ApTemp;
  Transfere metade dos registros de Ap para ApTemp;
  Atribui registro do meio a RegRetorno;
  Atribui ApTemp a ApRetorno;
}

void Insere(TipoRegistro Reg, TipoApontador *Ap)
{ Ins(Reg, *Ap, &Cresceu, &RegRetorno, &ApRetorno);
  if (Cresceu) { Cria nova pagina raiz  para RegRetorno e ApRetorno; }
}
```

Árvores B - Procedimento InsereNaPágina

```
void InsereNaPagina(TipoApontador Ap,
                    TipoRegistro Reg, TipoApontador ApDir)
{ short NaoAchouPosicao;
  int k;
  k = Ap->n;  NaoAchouPosicao = (k > 0);
  while (NaoAchouPosicao)
    { if (Reg.Chave >= Ap->r[k-1].Chave)
      { NaoAchouPosicao = FALSE;
        break;
      }
      Ap->r[k] = Ap->r[k-1];
      Ap->p[k+1] = Ap->p[k];
      k--;
      if (k < 1) NaoAchouPosicao = FALSE;
    }
  Ap->r[k] = Reg;
  Ap->p[k+1] = ApDir;
  Ap->n++;
}
```

Árvores B - Refinamento final do algoritmo Insere

```
void Ins(TipoRegistro Reg, TipoApontador Ap, short *Cresceu,
         TipoRegistro *RegRetorno, TipoApontador *ApRetorno)
{ long i = 1; long j;
  TipoApontador ApTemp;
  if (Ap == NULL)
  { *Cresceu = TRUE; (*RegRetorno) = Reg; (*ApRetorno) = NULL;
    return;
  }
  while (i < Ap->n && Reg.Chave > Ap->r[i-1].Chave) i++;
  if (Reg.Chave == Ap->r[i-1].Chave)
  { printf(" Erro: Registro ja esta presente\n"); *Cresceu = FALSE;
    return;
  }
  if (Reg.Chave < Ap->r[i-1].Chave) i--;
  Ins(Reg, Ap->p[i], Cresceu, RegRetorno, ApRetorno);
  if (!*Cresceu) return;
  if (Ap->n < MM)   /* Pagina tem espaco */
    { InsereNaPagina(Ap, *RegRetorno, *ApRetorno);
      *Cresceu = FALSE;
      return;
    }
  /* Overflow: Pagina tem que ser dividida */
  ApTemp = (TipoApontador)malloc(sizeof(TipoPagina));
  ApTemp->n = 0; ApTemp->p[0] = NULL;
```

{— *Continua na próxima transparência* —}

Árvores B - Refinamento final do algoritmo Insere

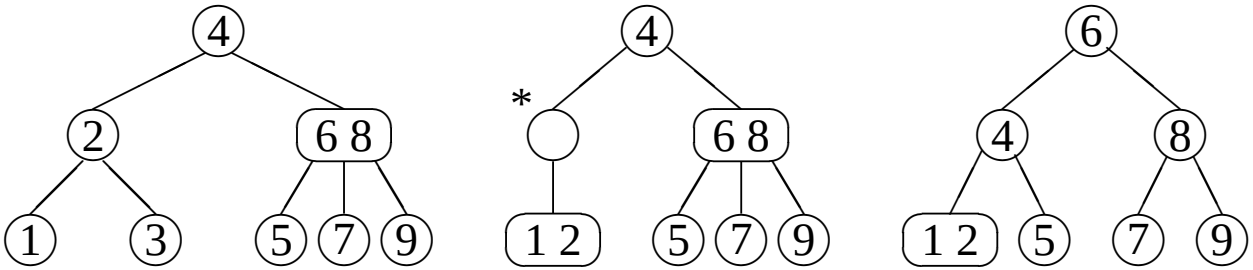
```
  if (i < M + 1)
  { InsereNaPagina(ApTemp, Ap->r[MM-1], Ap->p[MM]);
    Ap->n--;
    InsereNaPagina(Ap, *RegRetorno, *ApRetorno);
  }
  else InsereNaPagina(ApTemp, *RegRetorno, *ApRetorno);
  for (j = M + 2; j <= MM; j++)
    InsereNaPagina(ApTemp, Ap->r[j-1], Ap->p[j]);
  Ap->n = M;  ApTemp->p[0] = Ap->p[M+1];
  *RegRetorno = Ap->r[M];  *ApRetorno = ApTemp;
}

void Insere(TipoRegistro Reg, TipoApontador *Ap)
{ short Cresceu;
  TipoRegistro RegRetorno;
  TipoPagina *ApRetorno, *ApTemp;
  Ins(Reg, *Ap, &Cresceu, &RegRetorno, &ApRetorno);
  if (Cresceu)  /* Arvore cresce na altura pela raiz */
  { ApTemp = (TipoPagina *)malloc(sizeof(TipoPagina));
    ApTemp->n = 1;
    ApTemp->r[0] = RegRetorno;
    ApTemp->p[1] = ApRetorno;
    ApTemp->p[0] = *Ap;  *Ap = ApTemp;
  }
}
```

Árvores B - Remoção

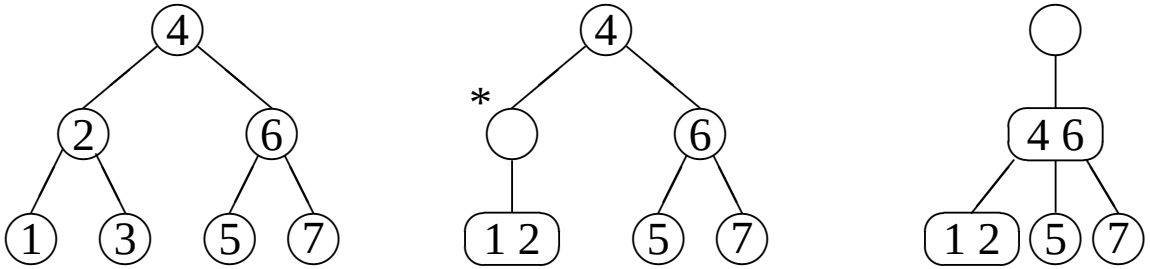
- Página com o registro a ser retirado é folha:
  1. retira-se o registro,
  2. se a página não possui pelo menos de m registros, a propriedade da árvore B é violada. Pega-se um registro emprestado da página vizinha. Se não existir registros suficientes na página vizinha, as duas páginas devem ser fundidas 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.



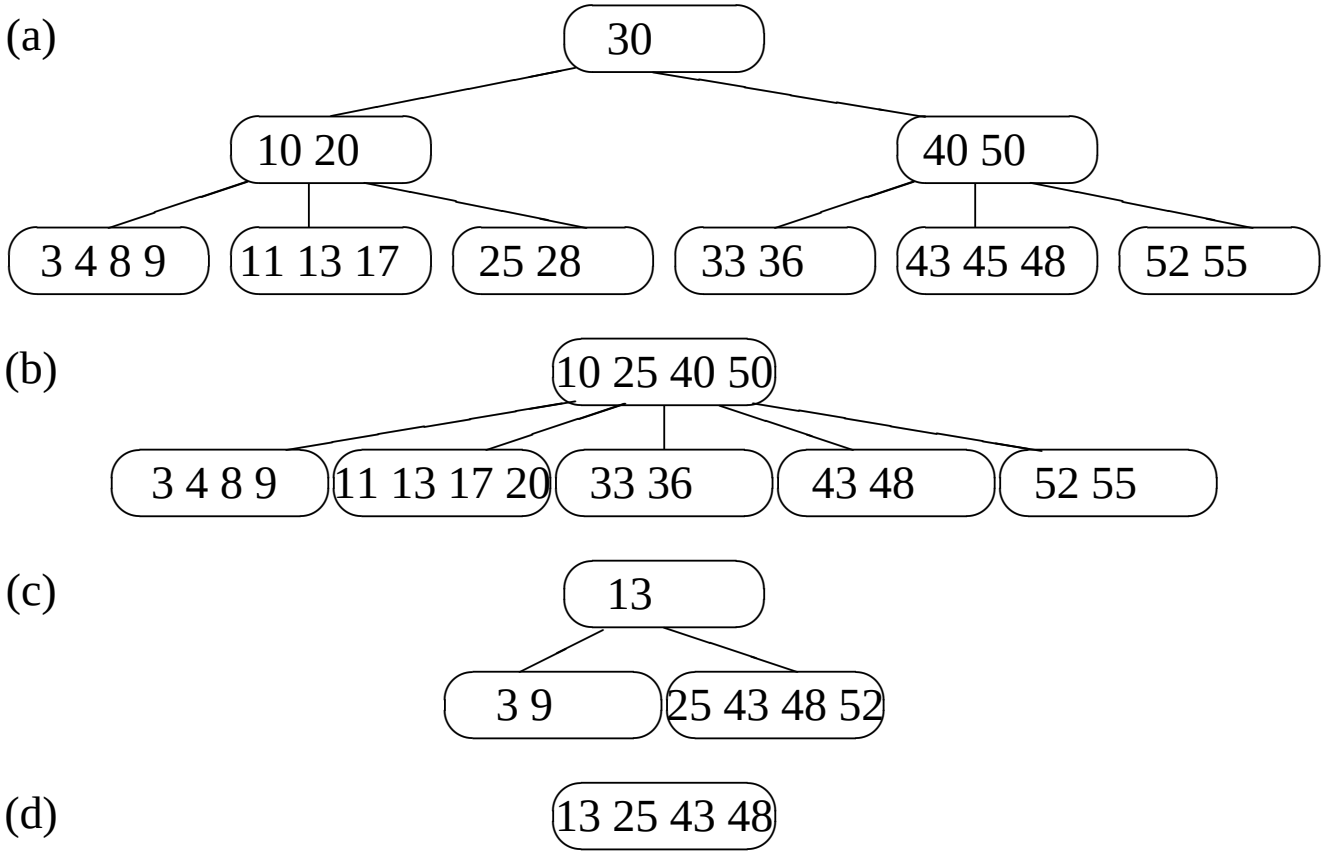
(a) Página vizinha possui mais do que m registros



(b) Página vizinha possui exatamente m registros

Árvores B - Remoção

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



Árvores B - Procedimento Retira

```
void Reconstitui(TipoApontador ApPag, TipoApontador ApPai,
                 int PosPai, short *Diminuiu)
{ TipoPagina *Aux; long DispAux, j;
  if (PosPai < ApPai->n)  /* Aux = TipoPagina a direita de ApPag */
  { Aux = ApPai->p[PosPai+1]; DispAux = (Aux->n - M + 1) / 2;
    ApPag->r[ApPag->n] = ApPai->r[PosPai];
    ApPag->p[ApPag->n + 1] = Aux->p[0]; ApPag->n++;
    if (DispAux > 0)  /* Existe folga: transfere de Aux para ApPag */
    { for (j = 1; j < DispAux; j++)
        InsereNaPagina(ApPag, Aux->r[j-1], Aux->p[j]);
      ApPai->r[PosPai] = Aux->r[DispAux-1]; Aux->n -= DispAux;
      for (j = 0; j < Aux->n; j++) Aux->r[j] = Aux->r[j + DispAux];
      for (j = 0; j <= Aux->n; j++) Aux->p[j] = Aux->p[j + DispAux];
      *Diminuiu = FALSE;
    }
    else /* Fusao: intercala Aux em ApPag e libera Aux */
      { for (j = 1; j <= M; j++) InsereNaPagina(ApPag, Aux->r[j-1], Aux->p[j]);
        free(Aux);
        for (j = PosPai + 1; j < ApPai->n; j++)
          { ApPai->r[j-1] = ApPai->r[j]; ApPai->p[j] = ApPai->p[j+1]; }
        ApPai->n--;
        if (ApPai->n >= M) *Diminuiu = FALSE;
      }
  }
{— Continua na próxima transparência —}
```

Árvores B - Procedimento Retira

```
else /* Aux = TipoPagina a esquerda de ApPag */
  { Aux = ApPai->p[PosPai-1]; DispAux = (Aux->n - M + 1) / 2;
    for (j = ApPag->n; j >= 1; j--) ApPag->r[j] = ApPag->r[j-1];
    ApPag->r[0] = ApPai->r[PosPai-1];
    for (j = ApPag->n; j >= 0; j--) ApPag->p[j+1] = ApPag->p[j];
    ApPag->n++;
    if (DispAux > 0) /* Existe folga: transf. de Aux para ApPag */
    { for (j = 1; j < DispAux; j++)
        InsereNaPagina(ApPag, Aux->r[Aux->n - j],
                              Aux->p[Aux->n - j + 1]);
      ApPag->p[0] = Aux->p[Aux->n - DispAux + 1];
      ApPai->r[PosPai-1] = Aux->r[Aux->n - DispAux];
      Aux->n -= DispAux; *Diminuiu = FALSE;
    }
    else /* Fusao: intercala ApPag em Aux e libera ApPag */
      { for (j = 1; j <= M; j++)
          InsereNaPagina(Aux, ApPag->r[j-1], ApPag->p[j]);
        free(ApPag); ApPai->n--;
        if (ApPai->n >= M) *Diminuiu = FALSE;
      }
  }
```

{— *Continua na próxima transparência* —}

Árvores B - Procedimento Retira

```
void Ret(TipoChave Ch, TipoApontador *Ap, short *Diminuiu)
{ long j, Ind = 1;
  TipoApontador Pag;
  if (*Ap == NULL)
  { printf("Erro: registro nao esta na arvore\n"); *Diminuiu = FALSE;
    return;
  }
  Pag = *Ap;
  while (Ind < Pag->n && Ch > Pag->r[Ind-1].Chave) Ind++;
  if (Ch == Pag->r[Ind-1].Chave)
  { if (Pag->p[Ind-1] == NULL)    /* TipoPagina folha */
    { Pag->n--; *Diminuiu = (Pag->n < M);
      for (j = Ind; j <= Pag->n; j++) { Pag->r[j-1] = Pag->r[j]; Pag->p[j] = Pag->p[j+1]; }
      return;
    }
    /* TipoPagina nao e folha: trocar com antecessor */
    Antecessor(*Ap, Ind, Pag->p[Ind-1], Diminuiu);
    if (*Diminuiu) Reconstitui(Pag->p[Ind-1], *Ap, Ind - 1, Diminuiu);
    return;
  }
  if (Ch > Pag->r[Ind-1].Chave) Ind++;
  Ret(Ch, &Pag->p[Ind-1], Diminuiu);
  if (*Diminuiu) Reconstitui(Pag->p[Ind-1], *Ap, Ind - 1, Diminuiu);
}
```

{— *Continua na próxima transparência* —}

Árvores B - Procedimento Retira

```
void Antecessor(TipoApontador Ap, int Ind,
                TipoApontador ApPai, short *Diminuiu)
{ if (ApPai->p[ApPai->n] != NULL)
  { Antecessor(Ap, Ind, ApPai->p[ApPai->n], Diminuiu);
    if (*Diminuiu)
    Reconstitui(ApPai->p[ApPai->n], ApPai, (long)ApPai->n, Diminuiu);
    return;
  }
  Ap->r[Ind-1] = ApPai->r[ApPai->n - 1];
  ApPai->n--; *Diminuiu = (ApPai->n < M);
}
```

{— *Continua na próxima transparência* —}

Árvores B - Procedimento Retira

```
void Ret(TipoChave Ch, TipoApontador *Ap, short *Diminuiu)
{ long j, Ind = 1;
  TipoApontador Pag;
  if (*Ap == NULL)
  { printf("Erro: registro nao esta na arvore\n"); *Diminuiu = FALSE;
    return;
  }
  Pag = *Ap;
  while (Ind < Pag->n && Ch > Pag->r[Ind-1].Chave) Ind++;
  if (Ch == Pag->r[Ind-1].Chave)
  { if (Pag->p[Ind-1] == NULL)    /* TipoPagina folha */
    { Pag->n--;
      *Diminuiu = (Pag->n < M);
      for (j = Ind; j <= Pag->n; j++)
        { Pag->r[j-1] = Pag->r[j]; Pag->p[j] = Pag->p[j+1]; }
      return;
    }
    /* TipoPagina nao e folha: trocar com antecessor */
    Antecessor(*Ap, Ind, Pag->p[Ind-1], Diminuiu);
    if (*Diminuiu)
    Reconstitui(Pag->p[Ind-1], *Ap, Ind - 1, Diminuiu);
    return;
  }
```

{— *Continua na próxima transparência* —}

Árvores B - Procedimento Retira

```
   if (Ch > Pag->r[Ind-1].Chave) Ind++;
   Ret(Ch, &Pag->p[Ind-1], Diminuiu);
   if (*Diminuiu) Reconstitui(Pag->p[Ind-1], *Ap, Ind - 1, Diminuiu);
}

void Retira(TipoChave Ch, TipoApontador *Ap)
{ short Diminuiu;
   TipoApontador Aux;
   Ret(Ch, Ap, &Diminuiu);
   if (Diminuiu && (*Ap)->n == 0)  /* Arvore diminui na altura */
   { Aux = *Ap;    *Ap = Aux->p[0];
     free(Aux);
   }
}
```

Árvores B* - TAD Dicionário

- Estrutura de Dados:

```
typedef int TipoChave;
typedef struct TipoRegistro {
  TipoChave Chave;
  /* outros componentes */
} TipoRegistro;
typedef enum {
  Interna, Externa
} TipoIntExt;
typedef struct TipoPagina *TipoApontador;
typedef struct TipoPagina {
  TipoIntExt Pt;
  union {
    struct {
      int ni;
      TipoChave ri[MM];
      TipoApontador pi[MM + 1];
    } U0;
    struct {
      int ne;
      TipoRegistro re[MM2];
    } U1;
  } UU;
} TipoPagina;
```

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* - Procedimento para pesquisar na árvore B*

```
void Pesquisa(TipoRegistro *x, TipoApontador *Ap)
{ int i;
  TipoApontador Pag;
  Pag = *Ap;
  if ((*Ap)->Pt == Interna)
  { i = 1;
    while (i < Pag->UU.U0.ni && x->Chave > Pag->UU.U0.ri[i - 1]) i++;
    if (x->Chave < Pag->UU.U0.ri[i - 1])
    Pesquisa(x, &Pag->UU.U0.pi[i - 1]);
    else Pesquisa(x, &Pag->UU.U0.pi[i]);
    return;
  }
  i = 1;
  while (i < Pag->UU.U1.ne && x->Chave > Pag->UU.U1.re[i - 1].Chave)
    i++;
  if (x->Chave == Pag->UU.U1.re[i - 1].Chave)
  *x = Pag->UU.U1.re[i - 1];
  else printf("TipoRegistro nao esta presente na arvore\n");
}
```

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 $\rightarrow$ página segura se o número de chaves é igual a $2m$,
  - remoção $\rightarrow$ 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 $\rightarrow$ executa uma operação de recuperação
- Processo modificador $\rightarrow$ executa uma operação de inserção ou retirada.
- Dois tipos de travamento:
  - Travamento para leitura $\rightarrow$ permite um ou mais leitores acessarem os dados, mas não permite inserção ou retirada.
  - Travamento exclusivo $\rightarrow$ 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 - 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}$.
  - 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: $Pr\{\text{Psmp esteja no } 1^{\underline{o}} \text{ nível}\} = \frac{4}{7}$,
  $Pr\{\text{Psmp esteja acima do } 1^{\underline{o}} \text{ nível}\} = \frac{3}{7}$.
- Árvore B de ordem m:
  $Pr\{\text{Psmp esteja no } 1^{\underline{o}} \text{ nível}\} = 1 - \frac{1}{(2\ln 2)m} + O(m^{-2})$,
  $Pr\{\text{Psmp esteja acima do } 1^{\underline{o}} \text{ nível}\} = \frac{3}{7} = \frac{1}{(2\ln 2)m} + O(m^{-2})$.

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 com que as páginas a serem particionadas em uma inserção estejam 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*.