

Pesquisa Externa

Estrutura de Dados II (BCC203)
Prof. Guilherme Tavares de Assis

Universidade Federal de Ouro Preto – UFOP
Instituto de Ciências Exatas e Biológicas – ICEB
Departamento de Computação – DECOM

Pesquisa Externa

- Métodos de pesquisa em memória secundária devem ser empregados quando o espaço ocupado pelos dados, a serem tratados, é maior que a memória interna disponível.
 - O custo para acessar um dado em um arquivo é bem maior do que o custo de processamento na memória interna.
- Medida de complexidade: n° de transferências de dados entre as memórias principal e secundária (minimização).
 - Memória secundária: apenas um dado pode ser acessado em um determinado momento (acesso sequencial).
 - Memória principal: acesso a qualquer dado a um custo uniforme (acesso direto).
- Características da arquitetura e do SO tornam os métodos dependentes de parâmetros que afetam seus desempenhos.

Sistema de Paginação

- Sistema de paginação consiste em uma estratégia que pode promover a implementação eficiente de métodos de pesquisa externa.
 - O espaço de endereçamento da memória secundária é dividido em páginas de tamanho igual que, em geral, é múltiplo de 512 *bytes* (cada página contém vários itens).
 - A memória principal é dividida em molduras de páginas de igual tamanho das páginas da memória secundária.
 - Durante a execução de um método de pesquisa externa, molduras de páginas da memória principal conterão algumas páginas da memória secundária (páginas ativas), enquanto as páginas restantes estarão residentes em memória secundária (páginas inativas).

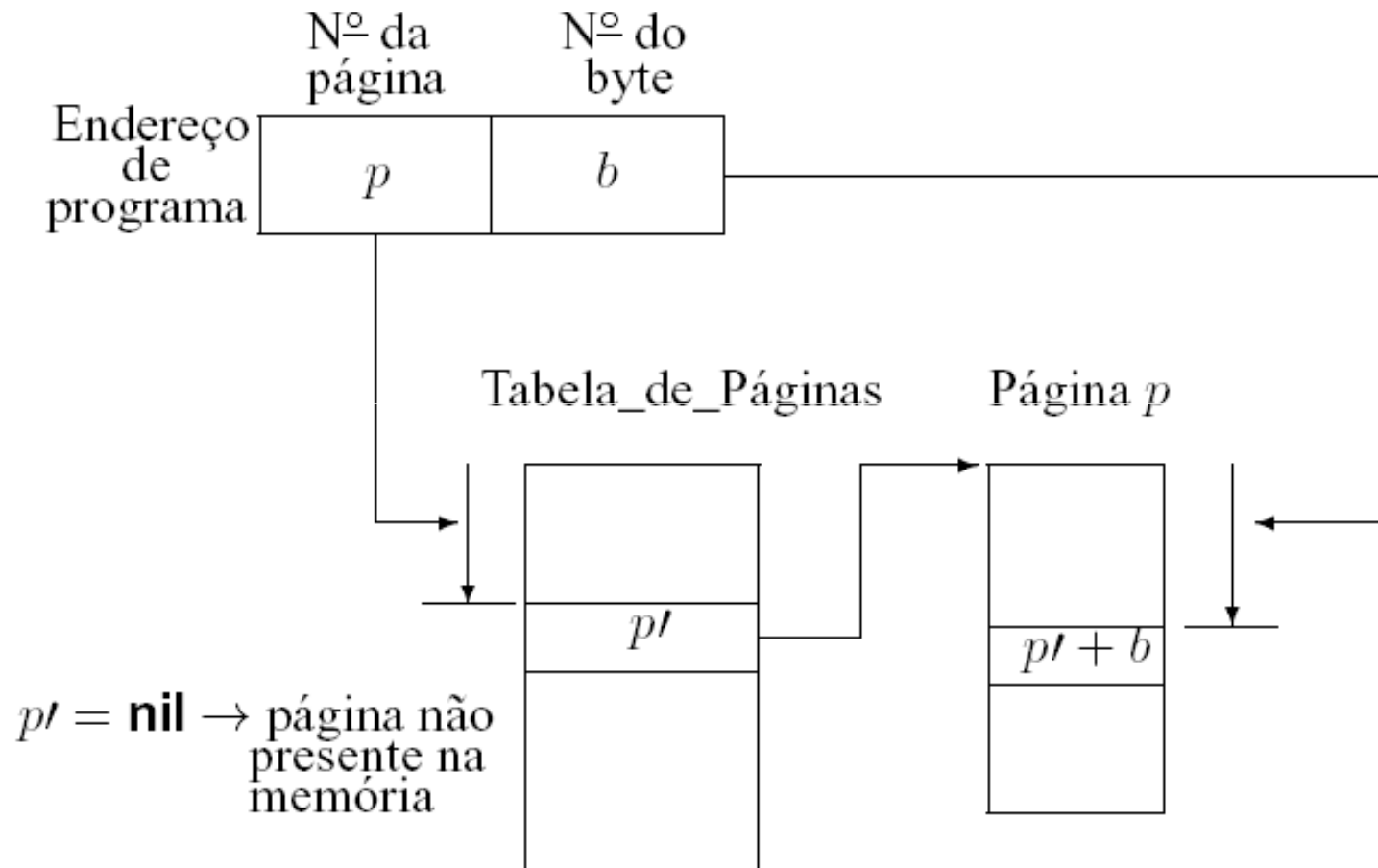
Sistema de Paginação

- O mecanismo do sistema de paginação possui duas funções:
 - Mapeamento de endereços: determinar qual página da memória secundária um programa está endereçando, e encontrar a moldura de tal página na memória principal, caso exista.
 - Transferência de páginas: transferir páginas da memória secundária para a memória primária, quando necessário, e transferi-las de volta para a memória secundária quando não estiverem sendo mais utilizadas.

Sistema de Paginação

- Para endereçar um item de uma página, uma parte dos bits do endereço é utilizada para representar o n° da página (**p**) e a outra parte o n° do *byte* do item dentro da página (**b**).
- O mapeamento de endereços das memórias principal e secundária é realizado por meio de uma Tabela de Páginas.
 - A *p*-ésima entrada da tabela contém a localização **p'** da moldura de página que contém a página número **p**, caso a mesma esteja na memória principal.
 - A tabela de páginas pode ser representada como um vetor do tamanho de número possível de páginas.
 - Caso a página número **p** não esteja em um moldura na memória principal, **p'** não terá valor (**p' = null**).

Sistema de Paginação



Sistema de Paginação

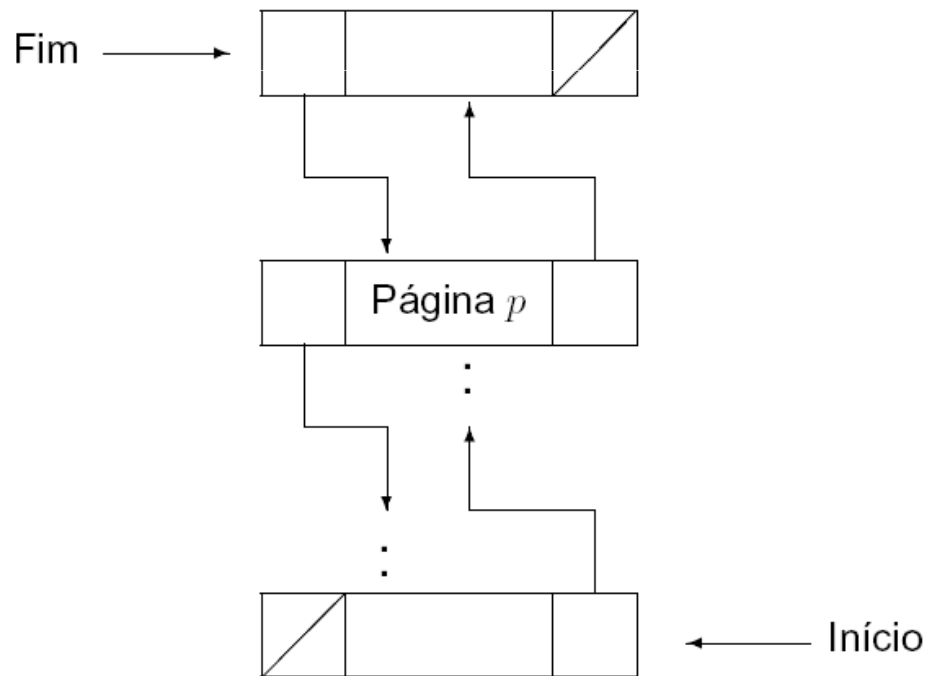
- Quando acontecer de um programa precisar de uma página **p** que não esteja na memória principal (**p'** = null), a página **p** deve ser trazida da memória secundária para a memória principal e a tabela de páginas deve ser atualizada.
- Se não houver uma moldura de página vazia, uma página deverá ser removida da memória principal.
 - O ideal é remover a página que não será referenciada pelo período de tempo mais longo no futuro.
 - Não há como prever o futuro: deve-se tentar inferir o futuro a partir do comportamento passado.

Sistema de Paginação

■ Políticas de remoção de páginas da memória principal:

■ Menos Recentemente Utilizada (LRU):

- Remove a página menos recentemente utilizada.
- Princípio: comportamento futuro deve seguir o passado recente.



Funcionamento:

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

Sistema de Paginação

- Menos Frequentemente Utilizada (LFU):
 - Remove a página menos frequentemente utilizada.
 - Desvantagem: uma página recentemente trazida da memória secundária tem um baixo número de acessos e pode ser brevemente removida.
- Ordem de Chegada (FIFO):
 - Remove a página que se encontra na memória principal há mais tempo (fila tradicional).
 - Algoritmo mais simples e barato de se manter.
 - Desvantagem: ignora o fato de que a página mais antiga pode ser a mais referenciada.

Acesso Sequencial Indexado

- Utiliza o princípio da pesquisa sequencial.
 - Cada item é lido sequencialmente até encontrar uma chave maior ou igual a chave de pesquisa.
- Requisitos necessários para aumentar a eficiência:
 - O arquivo deve estar ordenado pelo campo chave do item.
 - Um arquivo de índice de páginas, contendo pares de valores $\langle \mathbf{x}, \mathbf{p} \rangle$, deve ser criado, onde \mathbf{x} representa a chave de um item e \mathbf{p} representa o endereço da página na qual o primeiro item contém a chave \mathbf{x} .

Acesso Sequencial Indexado

- Considere o arquivo seqüencial indexado (15 itens).
 - Cada página tem capacidade para armazenar 4 itens do disco.
 - Cada entrada do índice de páginas armazena a chave do 1º item de cada página e o endereço de tal página no disco.

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
---	----------	---	-------------	---	-------------	---	----------

- Para se pesquisar por um item, deve-se:
 - localizar, no índice de páginas, a página que pode conter o item desejado de acordo com sua chave de pesquisa;
 - realizar uma pesquisa sequencial na página localizada.

Acesso Sequencial Indexado

```
#include <iostream>
#include <stdio.h>
using namespace std;

#define ITENSPAGINA 4
#define MAXTABELA 100

// definição de uma entrada da tabela de índice das páginas
typedef struct {
    int posicao;
    int chave;
} tipoindice;

// definição de um item do arquivo de dados
typedef struct {
    char titulo[31];  int chave;  float preco;
} tipoitem;

// continuando ...
```

Acesso Sequencial Indexado

```
int pesquisa (tipoindice tab[], int tam,
             tipoitem* item, FILE *arq) {

    tipoitem pagina[ITENSPAGINA];
    int i, quantitens;
    long desloc;

    // procura pela página onde o item pode se encontrar
    i = 0;
    while (i < tam && tab[i].chave <= item->chave) i++;

    // caso a chave desejada seja menor que a 1a chave, o item
    // não existe no arquivo
    if (i == 0) return 0;
    else {
        (*)
    }
    // continuando ...
}
```

Acesso Sequencial Indexado

```
// a ultima página pode não estar completa
```

```
if (i < tam) quantitens = ITENSPAGINA;
```

```
else {
```

```
    fseek (arq, 0, SEEK_END);
```

```
    quantitens = (ftell(arq)/sizeof(tipoitem))%ITENSPAGINA;
```

```
}
```

```
// lê a página desejada do arquivo
```

```
desloc = (tab[i-1].posicao-1)*ITENSPAGINA*sizeof(tipoitem);
```

```
fseek (arq, desloc, SEEK_SET);
```

```
fread (&pagina, sizeof(tipoitem), quantitens, arq);
```

```
// pesquisa sequencial na página lida
```


```
for (i=0; i < quantitens; i++)
```

```
    if (pagina[i].chave == item->chave) {
```

```
        *item = pagina[i];    return 1;
```

```
    }
```

```
return 0;
```

```
} 
```

```
// continuando ...
```

Acesso Sequencial Indexado

```
int main () {

    tipoindice tabela[MAXTABELA];
    FILE *arq;    tipoitem x;    int pos, cont;

    // abre o arquivo de dados
    if ((arq = fopen("livros.bin","rb")) == NULL) {
        cout << "Erro na abertura do arquivo\n"; return 0;
    }

    // gera a tabela de índice das páginas
    cont = 0; pos = 0;
    while (fread(&x, sizeof(x), 1, arq) == 1) {
        cont++;
        if (cont%ITENSPAGINA == 1) {
            tabela[pos].chave = x.chave;
            tabela[pos].posicao = pos+1;
            pos++;
        }
    }

    // continuando ...
```

Acesso Sequencial Indexado

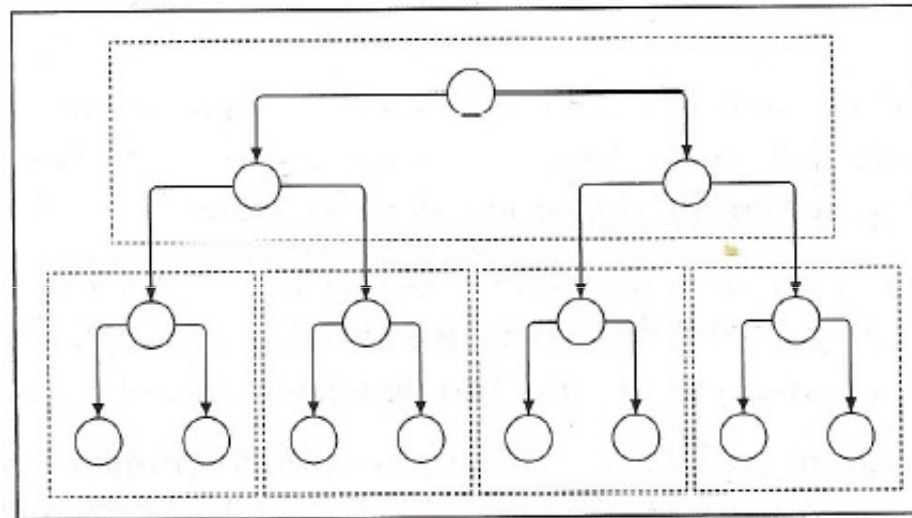
```
fflush (stdout);  
cout << "Código do livro desejado:";  cin >> x.chave;  
  
// ativa a função de pesquisa  
if (pesquisa (tabela, pos, &x, arq))  
    printf ("Livro %s (codigo %d) foi localizado",  
            x.titulo, x.chave);  
else  
    printf ("Livro de código %d nao foi localizado",x.chave);  
  
fclose (arq);  
return 0;  
  
}
```


Árvores de Pesquisa

- As árvores binárias de pesquisa são estruturas eficientes quando a memória principal armazena todos os itens.
 - Proporcionam acesso direto e sequencial, facilidade de inserção e retirada de itens, e boa utilização de memória.
- Para se pesquisar um item em grandes arquivos de dados armazenados em memória secundária, árvores binárias de pesquisa podem ser usadas de forma simplista.
 - Os nodos da árvore são armazenados em disco e os seus apontadores à esquerda e à direita armazenam endereços de disco ao invés de endereços de memória principal.
 - São necessários cerca de $\log_2 n$ acessos a disco para se encontrar um item (por exemplo, para $n = 1024$, aproximadamente, 10 acessos a disco devem ocorrer).

Árvores de Pesquisa

- Para diminuir o nº de acessos a disco, os nodos de uma árvore podem ser agrupados em páginas.



- Neste caso ótimo, em termos de acessos a disco, o formato da árvore muda de binário para quartenário, o que reduz pela metade o nº de acessos a disco no pior caso.
 - Para $n=1024$, recupera-se um item com 5 acessos no pior caso.

Árvores de Pesquisa

- A forma de organizar os nodos dentro de páginas é muito importante em relação ao n° esperado de páginas lidas quando se realiza uma pesquisa.
 - A organização ótima é difícil de ser obtida durante a construção da árvore (problema complexo de otimização).
- O algoritmo simples "alocação sequencial" armazena os nodos em posições consecutivas na página à medida que vão surgindo, sem considerar o formato físico da árvore.
 - Vantagem: utiliza todo o espaço disponível na página.
 - Desvantagem: os nodos de uma página estão relacionados pela ordem de entrada dos itens e não pela localidade na árvore, piorando bastante o tempo de pesquisa.

Árvores de Pesquisa

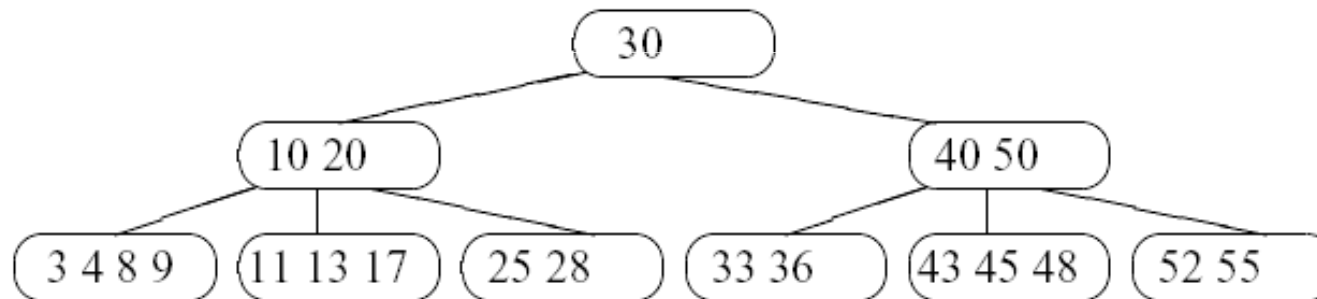
- O método de alocação de nodos em páginas, proposto por Muntz e Uzgalis (1970), considera a proximidade dos nodos dentro da árvore.
 - O novo nodo é sempre colocado na mesma página do nodo pai.
 - Se a página do nodo pai estiver cheia, o novo nodo é colocado no início de uma nova página criada.
 - Vantagem: nº de acessos na pesquisa é próximo do ótimo.
 - Desvantagem: ocupação média das páginas é baixa (ordem de 10%).
- **Árvore B:** solução para o problema, sendo uma proposta para manter o crescimento da árvore equilibrado e permitir inserções e retiradas.

Árvore B

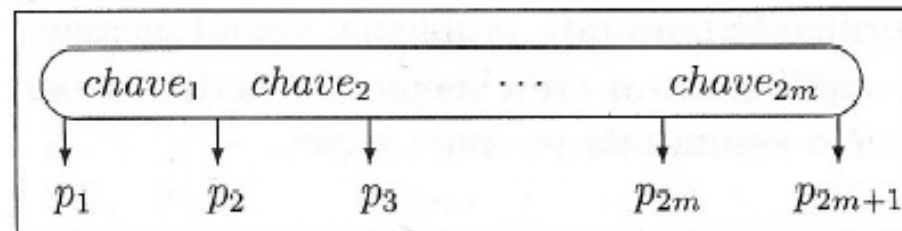
- Árvore B é uma técnica de organização e manutenção de arquivos, proposta por Bayer e McCreight (1972).
- É uma árvore n-ária: mais de dois descendentes por nodo.
 - Os nodos são mais comumente chamados de páginas.
- Em uma árvore B de ordem **m**, tem-se:
 - página raiz: contém entre **1** e **2m** itens;
 - demais páginas: contém, no mínimo, **m** itens e **m+1** descendentes e, no máximo, **2m** itens e **2m+1** descendentes;
 - páginas folhas: aparecem todas no mesmo nível;
 - itens: aparecem dentro de uma página em ordem crescente, de acordo com suas chaves, da esquerda para a direita.

Árvore B

- Ex.: árvore B de ordem $m = 2$ com três níveis.
 - Todas as páginas contêm de 2 a 4 itens, exceto a raiz que pode conter de 1 a 4 itens.
 - O esquema representa uma extensão natural da árvore binária de pesquisa.



- Forma geral de uma página de uma árvore B de ordem m :



Árvore B: 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;
```

Árvore B: Inicialização

```
void Inicializa (TipoApontador Arvore)
{
    Arvore = NULL;
}
```


Árvore B: Pesquisa

- A operação de pesquisa em uma árvore B é semelhante à pesquisa em uma árvore binária de pesquisa.
- Para encontrar o item x , deve-se:
 - comparar a chave do item x com as chaves que estão na página raiz até encontrar a chave desejada ou o intervalo no qual ela se encaixa;
 - caso não tenha localizado a chave desejada, seguir o apontador para a subárvore do intervalo encontrado;
 - repetir o processo recursivamente até encontrar a chave desejada ou atingir uma página folha (apontador nulo).
- O intervalo desejado pode ser encontrado por meio de uma pesquisa sequencial ou algum outro método mais eficiente.

Árvore 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]);
}

```

Pesquisa sequencial para se encontrar o intervalo desejado

Verifica se a chave
desejada foi localizada

Ativação recursiva da Pesquisa
em uma das subárvores
(esquerda ou direita)

Árvore B: Caminhamento

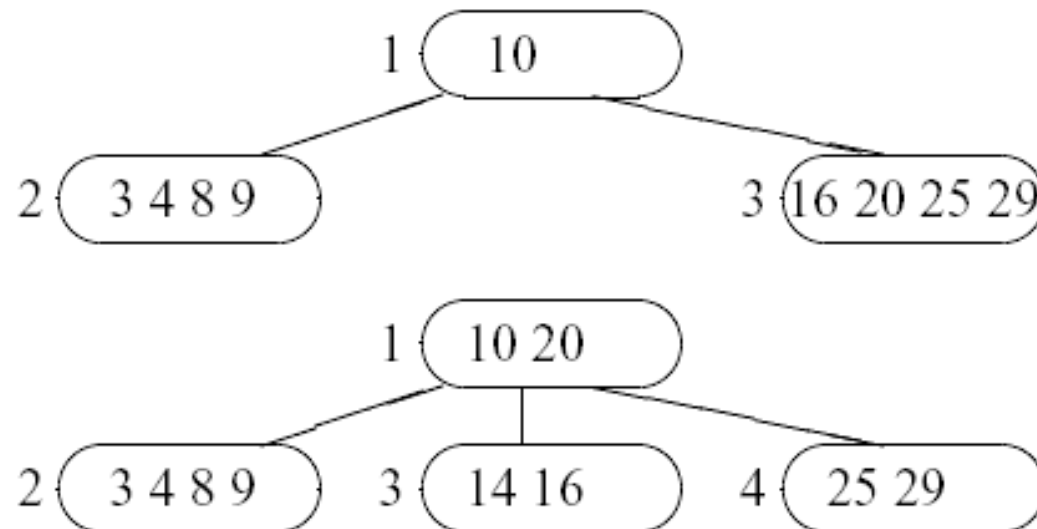
```
void Imprime(TipoApontador arvore){  
    int i = 0;  
  
    if (arvore == NULL) return;  
  
    while (i <= arvore->n) {  
        Imprime(arvore->p[i]);  
        if (i != arvore->n)  
            cout << arvore->r[i].Chave << " ";  
        i++;  
    }  
  
}
```

Árvore B: Inserção

- Para inserir o item **x** em uma árvore B, deve-se:
 - Localizar a página apropriada aonde o item deve ser inserido ("pesquisa sem sucesso").
 - Se o item a ser inserido encontra uma página com menos de **$2m$** itens, o processo de inserção fica limitado a tal página.
 - Se o item a ser inserido encontra uma página cheia, é criada uma nova página para divisão dos itens, envolvendo a nova página, a página onde o item seria inserido, e a página pai de ambas.
 - Se a página pai estiver cheia, o processo de divisão se propaga.
 - No pior caso, o processo de divisão pode propagar-se até a raiz da árvore B e, neste caso, sua altura aumenta.
 - Única forma de aumentar a altura de uma árvore B: divisão da raiz.

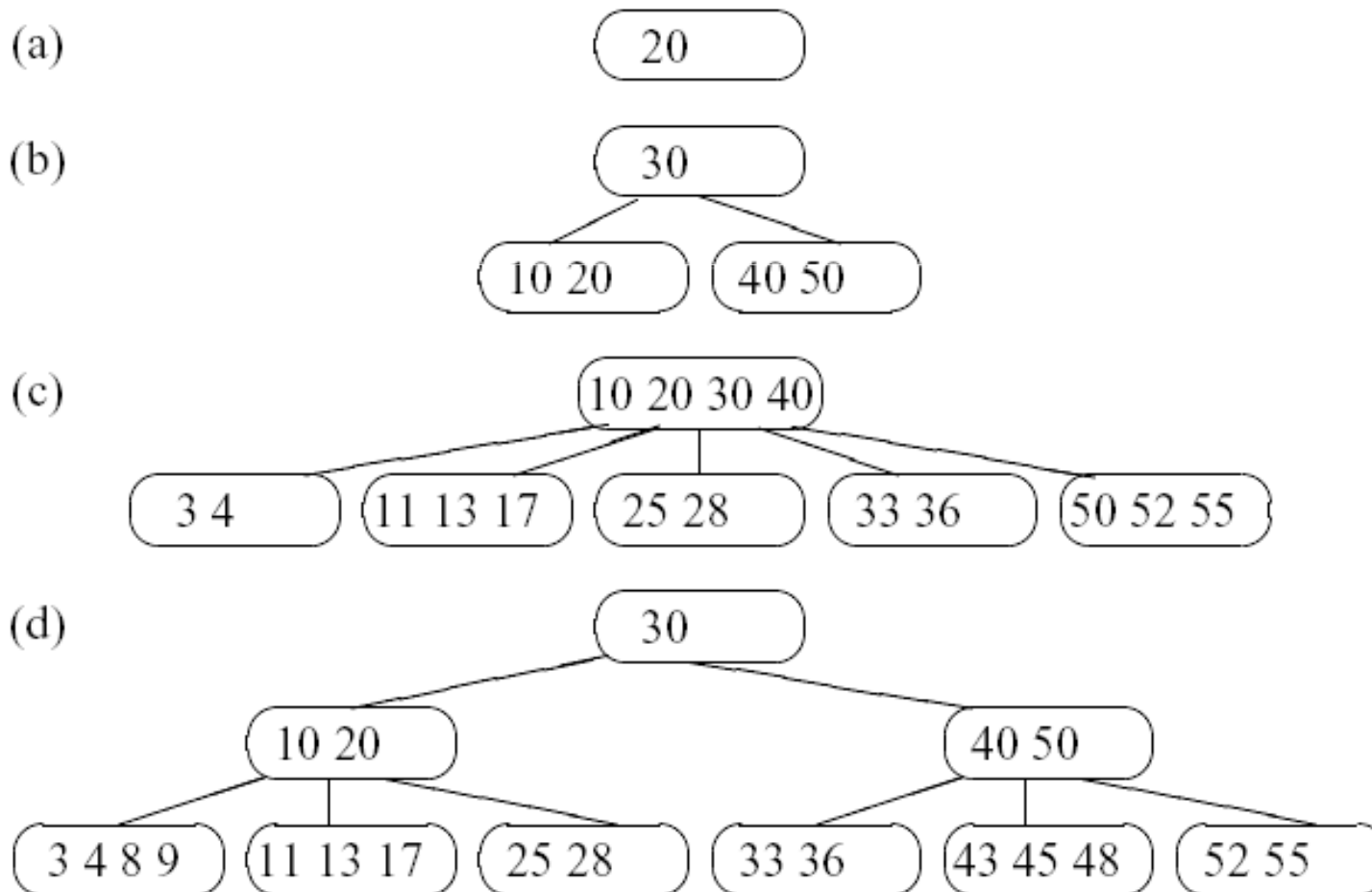
Árvore B: Inserção

- Ex.: inserção do item com chave 14 em uma árvore B.
 - Item contendo a chave 14 não se encontra na árvore.
 - Página 3, onde o item deveria ser inserido, está cheia.
 - Página 4 é criada, por meio da divisão da página 3.
 - Os $2m+1$ itens são distribuídos igualmente entre as páginas 3 e 4, e o item do meio (chave 20) é movido para a página pai.



Árvore B: Inserção

- Ex.: inserção das chaves (a) 20; (b) 10, 40, 50, 30; (c) 55, 3, 11, 4, 28, 36, 33, 52, 17, 25, 13; (d) 45, 9, 43, 8, 48.



Árvore B: Inserção

```

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++;
}

```

Árvore B: Inserção

```
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;  
  }
```

/ continua ... */*

Árvore B: Inserção

```

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 ... */*

Árvore B: Inserção

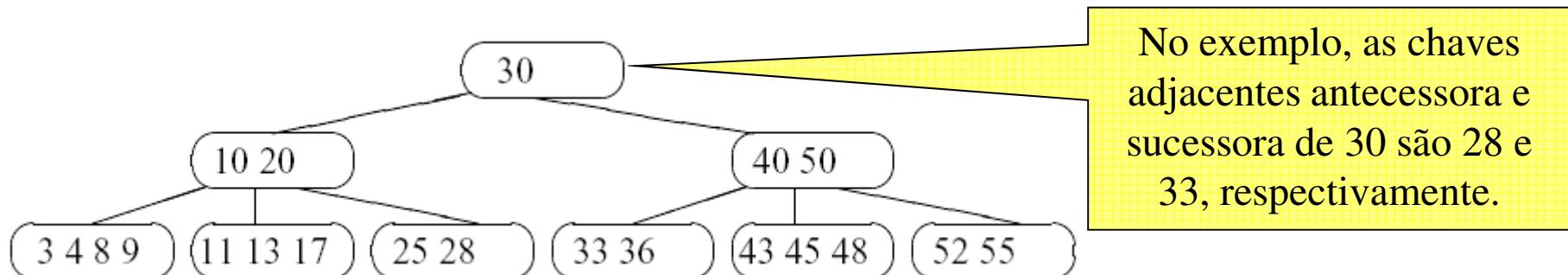
```
if ( i < M + 1)
{ InsereNaPagina(ApTemp, Ap->r [MM - 1], Ap->p [MM] );
  Ap->n--;
  InsereNaPagina(Ap, *RegRetorno, *ApRetorno);
}
else InsereNaPagina(ApTemp, *RegRetorno, *ApRetorno);
for ( j = M + 2; j <= MM; j++)
  InsereNaPagina(ApTemp, Ap->r [j - 1], Ap->p [j] );
Ap->n = M;  ApTemp->p [0] =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

Árvore B: Inserção

```
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;
  }
}
```

Árvore B: Remoção

- A remoção de um item em uma árvore B sempre implica na remoção de um item presente em uma página folha.
 - Caso o item desejado esteja em uma página folha, a remoção é direta.
 - Caso contrário, deve-se:
 - Substituir o item desejado pelo item que contenha a chave adjacente antecessora (item mais a direita da subárvore à esquerda) ou sucessora (item mais a esquerda da subárvore à direita);
 - Remover o item desejado.

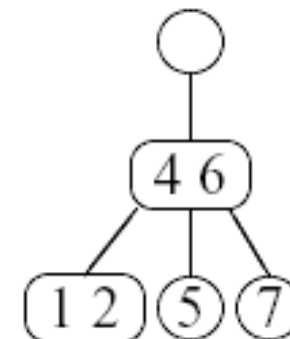
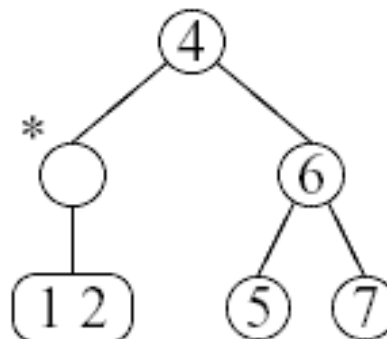
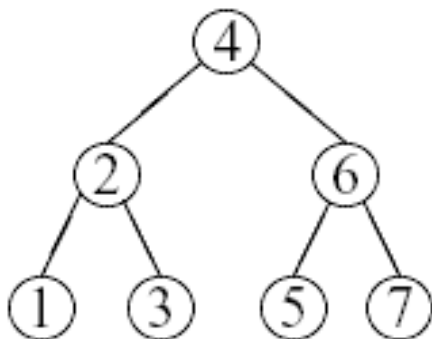


Árvore B: Remoção

- Assim que um item é removido de uma página folha, deve-se verificar se tal página contém, pelo menos, **m** itens, ou seja, se a propriedade da árvore B não foi violada.
 - Caso tenha sido violada, é necessário reconstituir a propriedade da árvore B, tomando emprestado um item da página vizinha.
 - Existem duas possibilidades para efetivar tal operação de empréstimo.

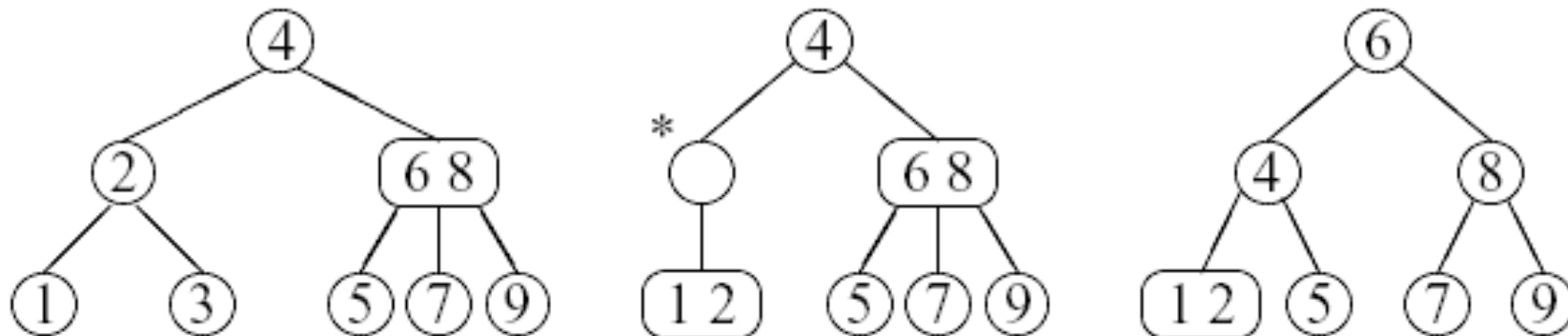
Árvore B: Remoção

- 1ª possibilidade: a página vizinha possui **m** itens.
 - Já que o n° total de itens nas duas páginas é **2m-1**, as mesmas devem ser fundidas em uma só, tomando emprestado da página pai o item do meio e permitindo liberar uma das páginas.
 - Este processo pode propagar até a página raiz e, ficando a mesma vazia, a página raiz será eliminada, causando redução na altura da árvore.
- Ex.: remoção da chave 3 em uma árvore B de ordem m=1.



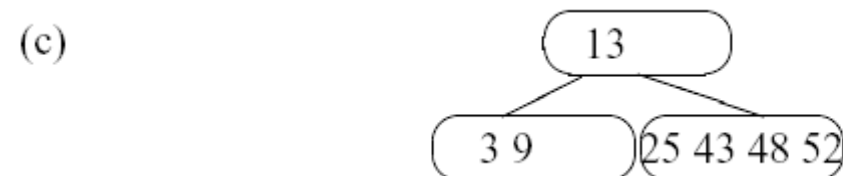
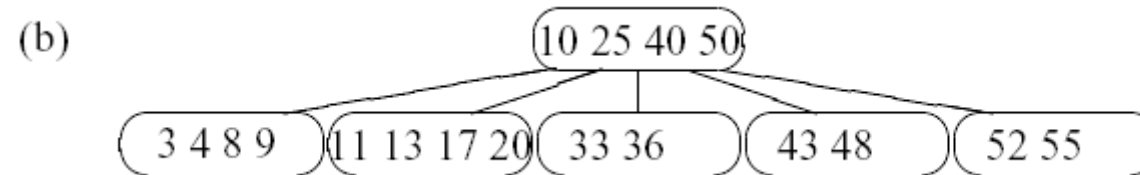
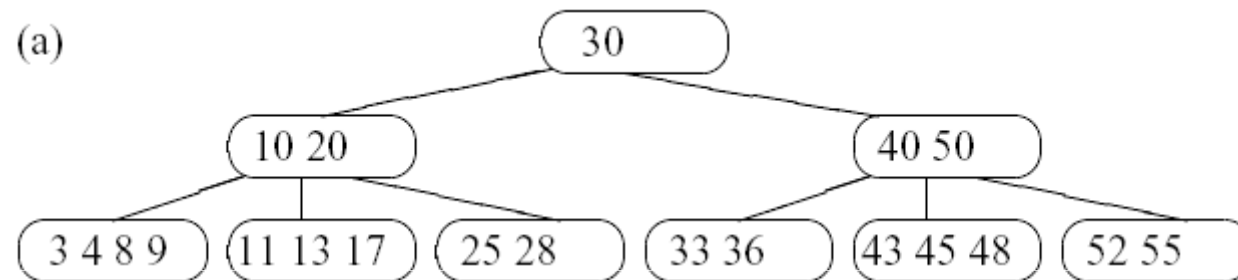
Árvore B: Remoção

- 2ª possibilidade: o nº de itens na página vizinha é maior que **m**.
 - Deve-se tomar emprestado um item da página vizinha e trazê-lo para a página em questão via página pai.
- Ex.: remoção da chave 3 em uma árvore B de ordem $m=1$.



Árvore B: Remoção

- Ex.: remoção das chaves (a) 45, 30, 28; (b) 50, 8, 10, 4, 20, 40, 55, 17, 33, 11, 36; (c) 3, 9, 52.



Árvore B: Remoção – Procedimento Reconstitui

```
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++)

```
    InseNaPagina(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;
```

(**) }

/ continua ... */*

Árvore B: Remoção – Procedimento Reconstitui

```

else /* Fusao: intercala Aux em ApPag e libera Aux */
{
  for (j = 1; j <= M; j++) InereNaPagina(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 ... */

Árvore B: Remoção – Procedimento Reconstitui

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++)
  InserirNaPagina(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;
```

(***)

}

/ continua ... */*

Árvore B: Remoção – Procedimento Reconstitui

```

else /* Fusao: intercala ApPag em Aux e libera ApPag */
{
    (***)
    for (j = 1; j <= M; j++)
        InereNaPagina(Aux, ApPag->r[j-1], ApPag->p[j]);
    free(ApPag);  ApPai->n--;
    if (ApPai->n >= M)  *Diminuiu = FALSE;
}
}
(*)

```

Árvore B: Remoção – Procedimento Antecessor

```

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);
}

```

Árvore B: Remoção – Procedimento Ret

```
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;
  }
```

/ continua ... */*

Árvore B: Remoção – Procedimento Ret

```

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 ... */

```

Árvore B: Remoção – Procedimento Ret

```
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);  
}
```

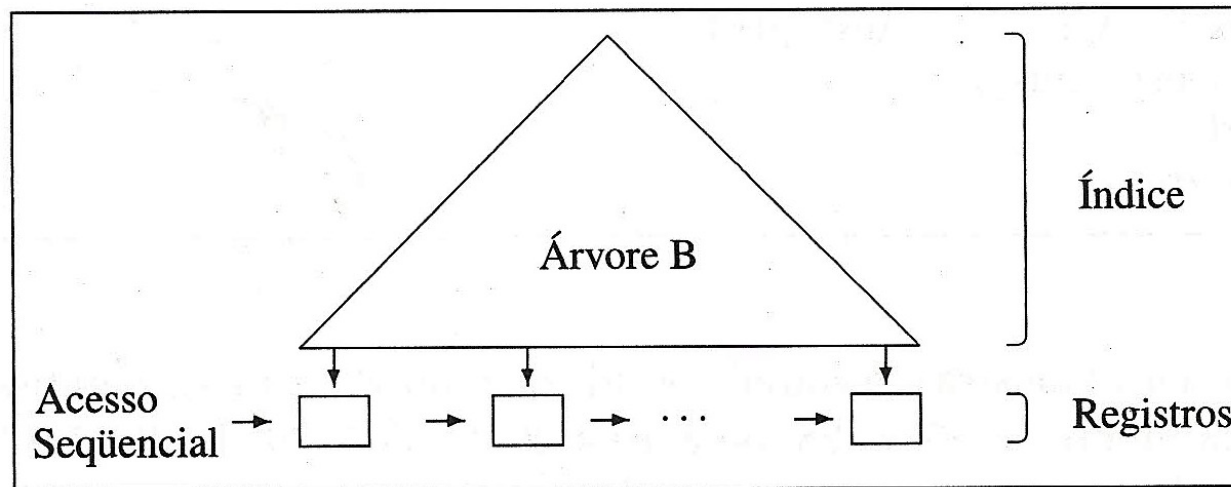

Árvore B: Remoção – Procedimento Retira

```
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);
    }
}
```

"Diminuiu" indica se uma quantidade menor do que **m** itens passa a ocupar a página.

Árvore B*

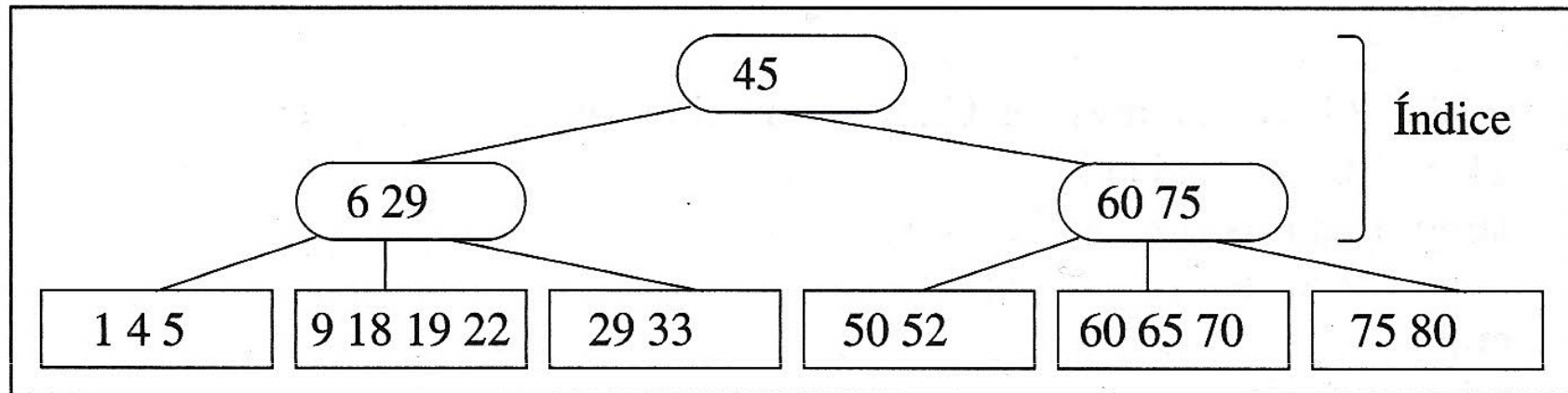
- Uma das alternativas para implementação da árvore B é a árvore B*, onde:
 - todos os itens estão armazenados no último nível, ou seja, o nível das páginas folha;
 - os níveis acima do último nível constituem um índice cuja organização é a de uma árvore B.



Árvore B*

- Assim, em uma árvore B*, há uma separação lógica entre o índice (árvore B) e os itens que constituem o arquivo propriamente dito.
 - No índice, só aparecem as chaves dos itens.
 - Os itens completos do arquivo encontram-se nas páginas folha.
- Ademais, as páginas folha podem ou não estar conectadas da esquerda para a direita, permitindo um acesso sequencial mais eficiente do que o acesso via índice.

Árvore B*



Árvore B*: Estrutura de Dados

- Pelo fato da estrutura das páginas do índice de uma árvore B* ser diferente da estrutura de suas páginas folha, há necessidade de se categorizar as páginas da árvore como sendo internas ou externas.
- Como não há necessidade do uso de apontadores nas páginas folha, é possível armazenar uma quantidade maior de itens nas mesmas (ordem m é maior).
 - Isto não cria problema para o algoritmo de inserção, pois as metades da página particionada permanecem no mesmo nível da página original antes da partição.

Árvore B*: Estrutura de Dados

```
typedef long TipoChave;
```

```
typedef struct TipoRegistro {  
    TipoChave Chave;  
    /* outros componentes */  
} TipoRegistro;
```

```
typedef enum {Interna, Externa} TipoIntExt;
```

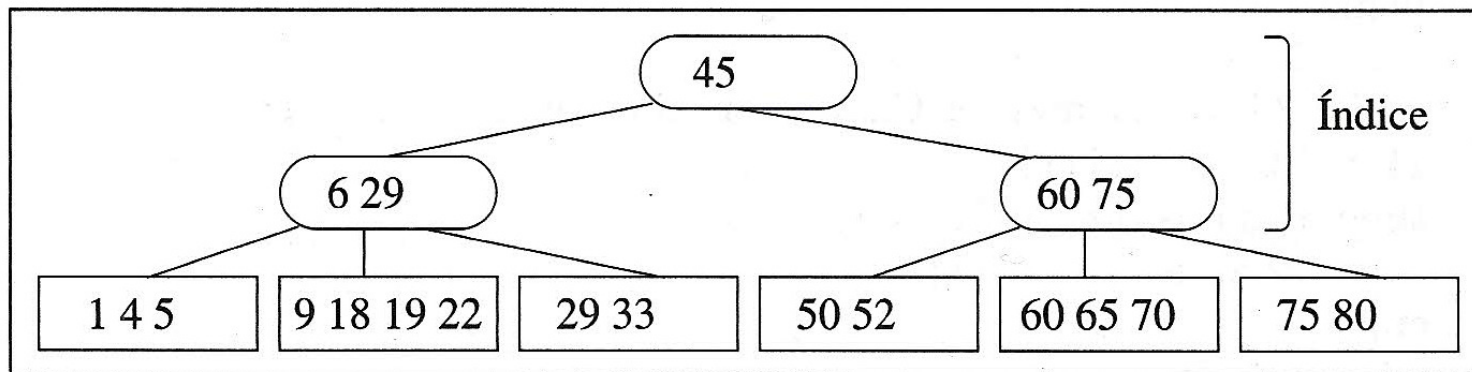
```
typedef struct TipoPagina* TipoApontador;
```

Árvore B*: Estrutura de Dados

```
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;
```

Árvore B*: Pesquisa

- A operação de pesquisa em uma árvore B* é semelhante à pesquisa em uma árvore B, sendo que:
 - a pesquisa sempre leva a uma página folha;
 - os valores encontrados ao longo do caminho são irrelevantes desde que conduzam à página folha correta;
 - ao encontrar a chave desejada em uma página do índice, a pesquisa não pára; no caso, o apontador da direita é seguido e a pesquisa continua até que se encontre uma página folha.



Árvore B*: Pesquisa

```

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");
}

```

Pesquisa sequencial na página interna

Ativação recursiva em uma das subárvores: a Pesquisa só pára ao encontrar uma página folha.

Pesquisa sequencial na página folha

Verifica se a chave desejada foi localizada

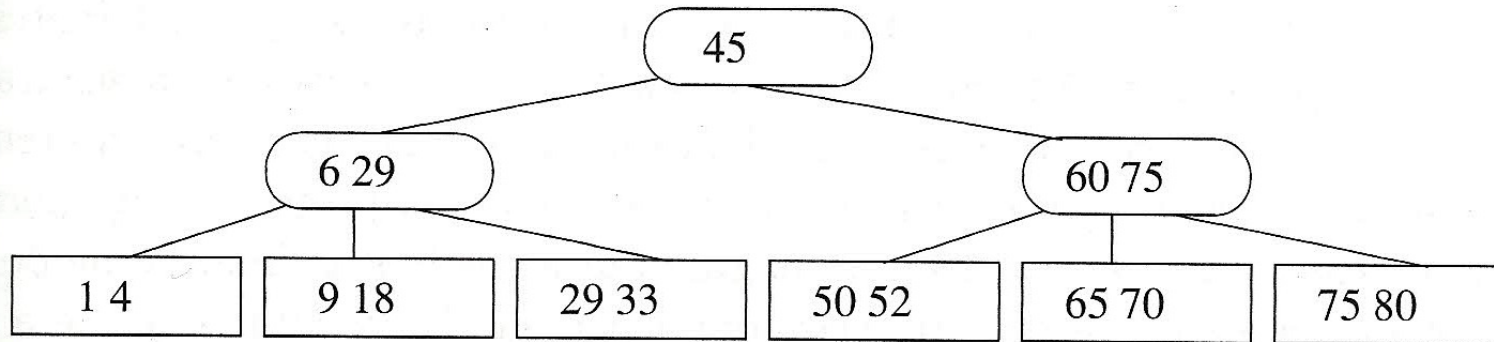
Árvore B*: Inserção

- A operação de inserção de um item em uma árvore B* é essencialmente igual à inserção de um item na árvore B.
- Diferença:
 - Quando uma folha é dividida em duas, o algoritmo promove uma cópia da chave que pertence ao item do meio para a página pai no nível anterior, retraindo o próprio item do meio na página folha da direita.

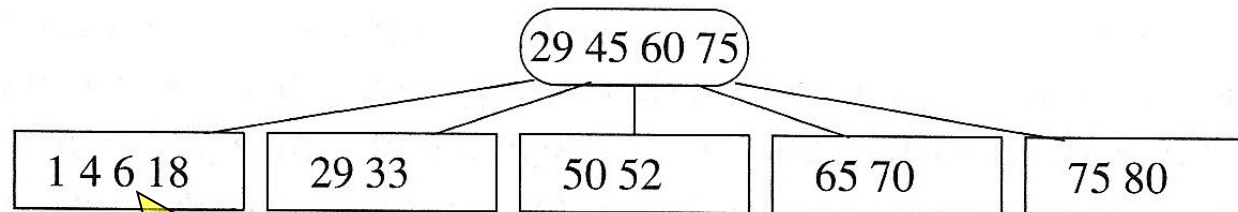
Árvore B*: Remoção

- A operação de remoção em uma árvore B* é relativamente mais simples do que em uma árvore B.
 - O item a ser removido reside sempre em uma página folha, não havendo necessidade de se localizar o item com a chave antecessora.
 - Caso a folha fique com pelo menos **m** itens, as páginas do índice não precisam ser modificadas, mesmo se uma cópia da chave que pertence ao item a ser removido esteja no índice.
 - As páginas do índice precisarão ser modificadas apenas se a folha ficar com uma quantidade de itens menor que **m**.

Árvore B*: Remoção



(a) Retirada dos registros 5, 19, 22, 60 da árvore da Figura 6.15



(b) Retirada do registro 9 em (a): a estrutura do índice é modificada

O registro de chave 6 pode ser criado na página folha?