

# Estruturas de Dados

## Pesquisa em Memória Secundária: Árvores B

---

Professores: Anisio Lacerda  
Wagner Meira Jr.

# Pesquisa em Memória Secundária

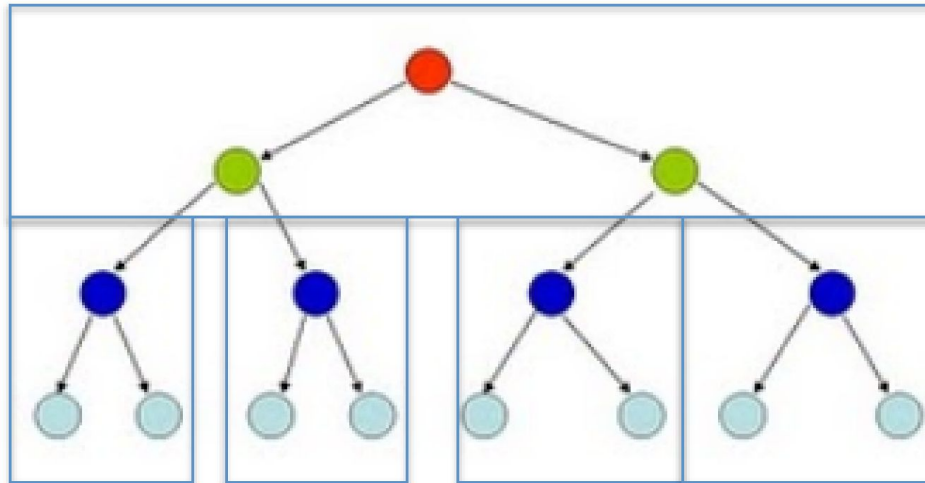
- Arquivos contém mais registros do que a memória interna (primária) pode armazenar
- Custo para acessar um registro é ordens de grandeza maior que o custo de acesso à memória primária
- **Medida de complexidade:** custo de transferir dados entre a memória principal e secundária (minimizar o número de transferências)

# Árvores B: Introdução

- **Problema:** acessar dados em arquivos grandes armazenados em memória secundária
- **Solução 1:** Usar uma árvore binária
  - Armazenar nós em disco
  - Ponteiros *esq* e *dir* apontam para endereços em disco
  - Custo de leitura:  $O(\log n)$  acessos em disco
    - $n = 10^6$ ,  $\log(n) \approx 20$  acessos em disco!

## Solução 2

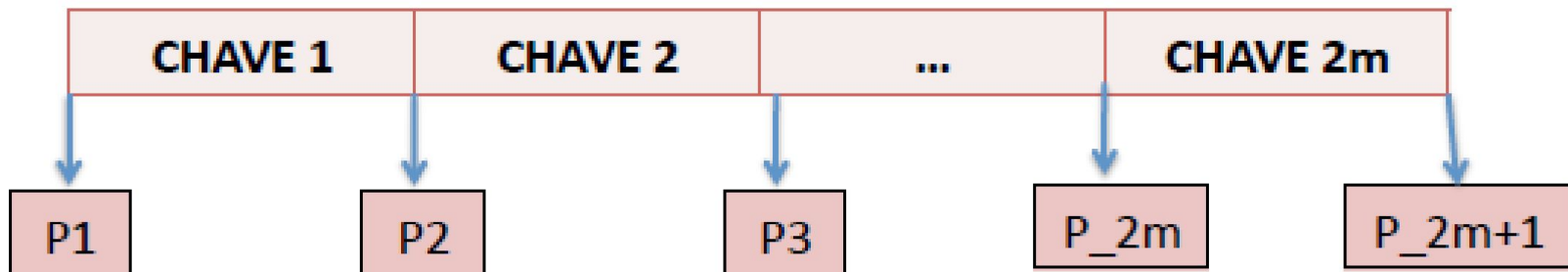
- Agrupar nós da árvore binária em páginas



- Árvore binária  $\square$  árvore quaternária (4 filhos por página)
- Problema: qual a melhor forma de distribuir os registros entre as páginas? (problema de otimização complexo)

# Árvores B

- Solução para o problema de distribuir os registros entre páginas
- Proposto por Bayer & McCreight em 1972 (*Boeing Research Lab*)
- Página de uma árvore B de ordem  $m$ :

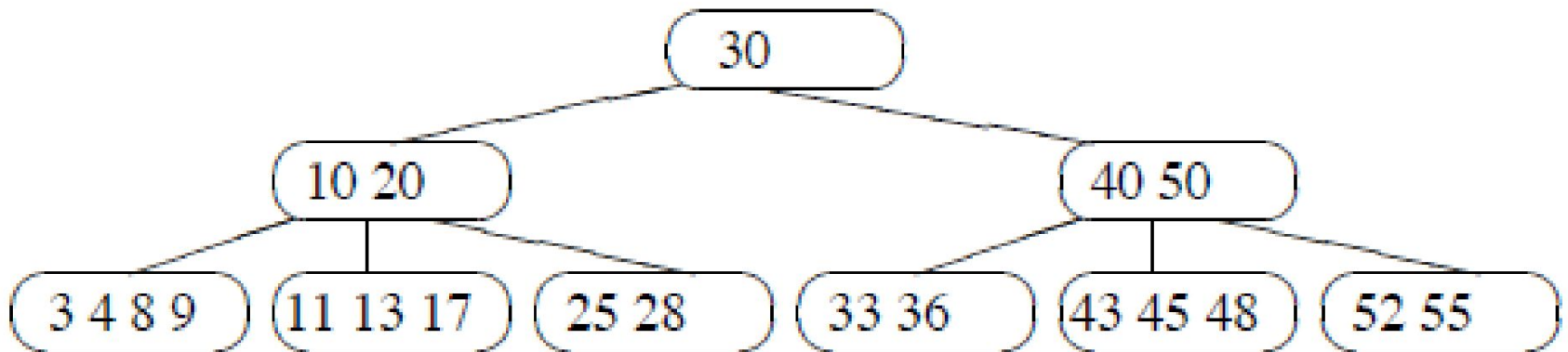


# Árvores B - Propriedades

- São árvores  $n$ -árias: mais de um registro por nó
- Em uma árvore B de ordem  $m$ :
  - Página raiz: entre 1 a  $2m$  registros
  - Demais páginas:
    - No mínimo  $m$  registros e  $m+1$  descendentes
    - No máximo  $2m$  registros e  $2m+1$  descendentes
  - Páginas folhas: aparecem todas no mesmo nível
  - Registros em ordem crescente da esquerda para a direita

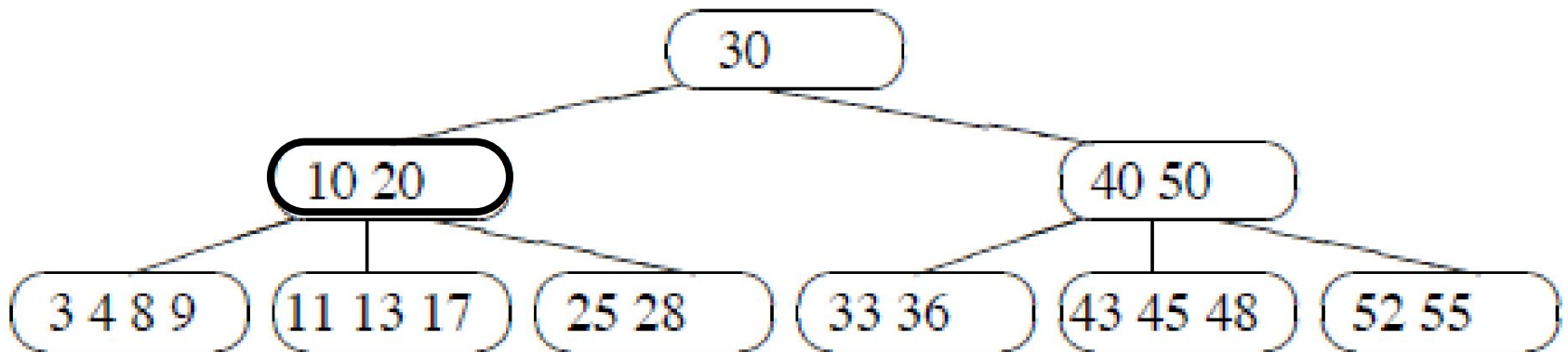
# Exemplo

- Árvore B de ordem  $m=2$  com três níveis:



# Exemplo

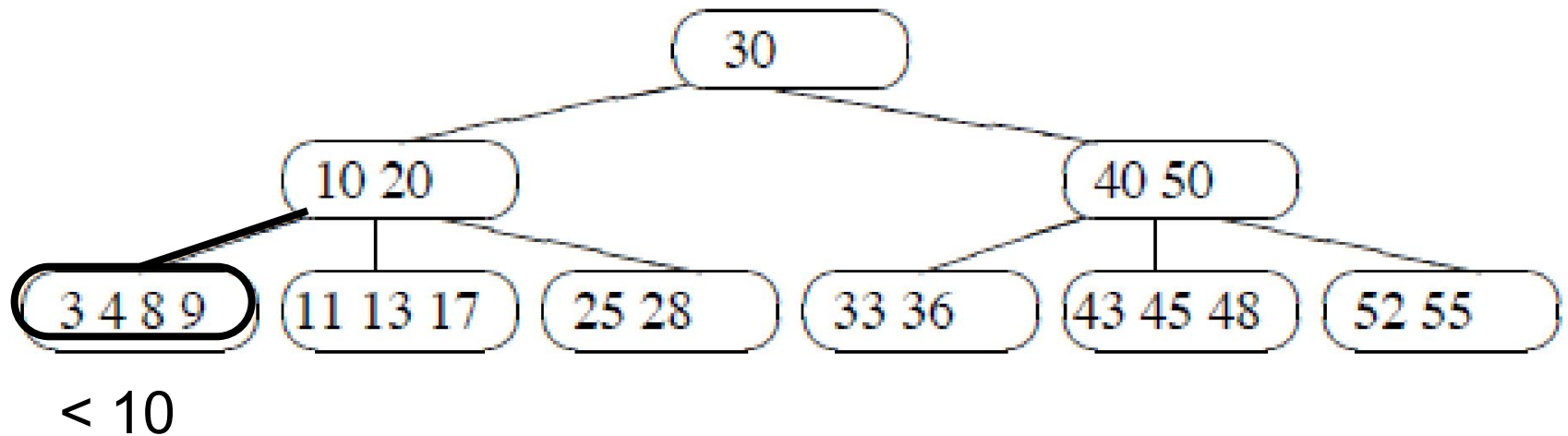
- Árvore B de ordem  $m=2$  com três níveis:





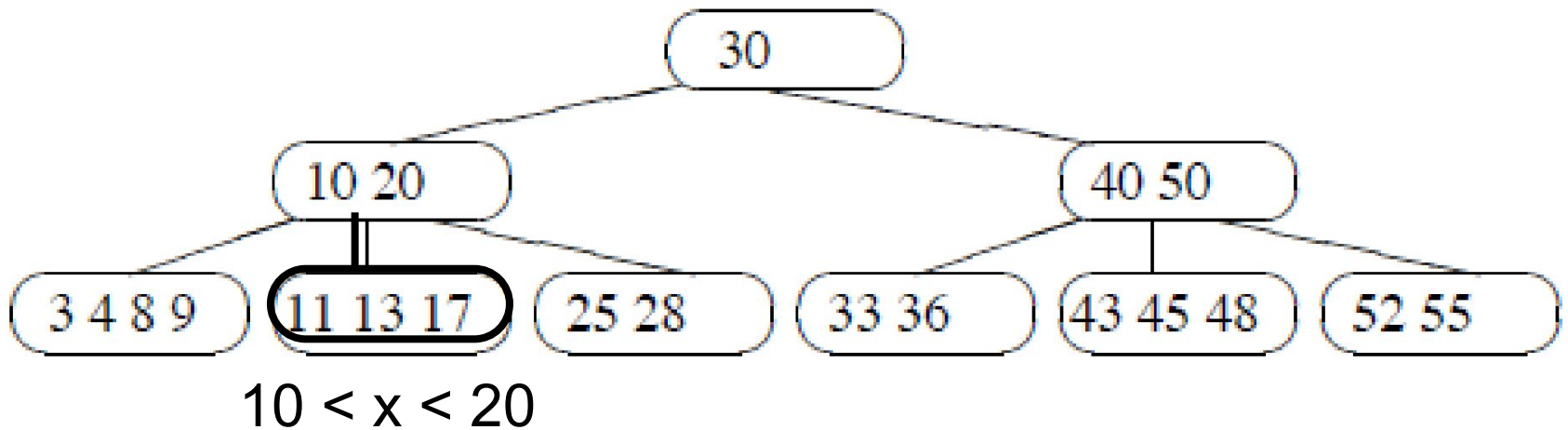
# Exemplo

- Árvore B de ordem  $m=2$  com três níveis:



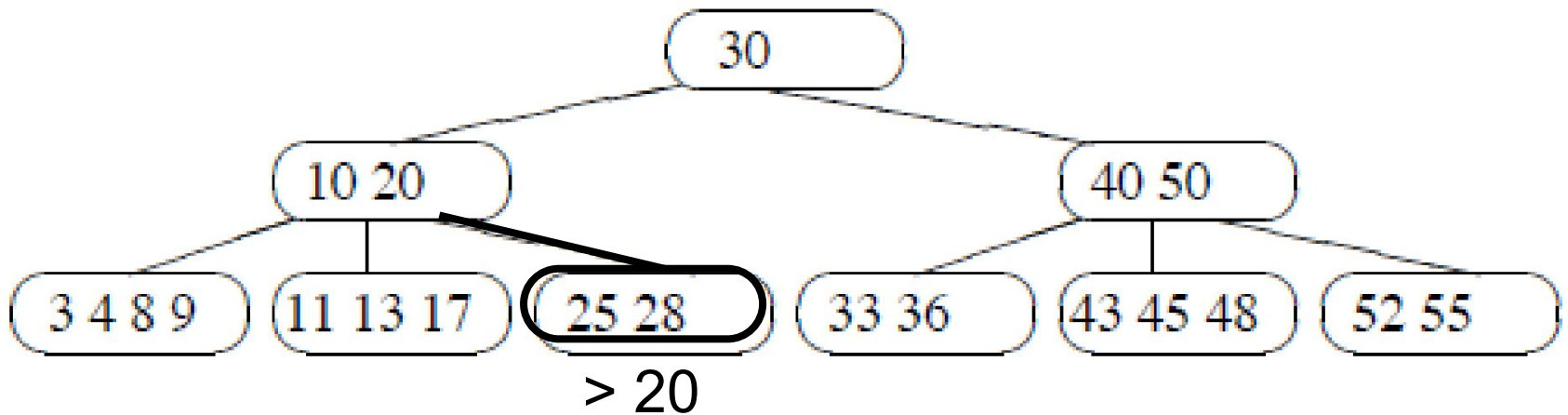
# Exemplo

- Árvore B de ordem  $m=2$  com três níveis:



# Exemplo

- Árvore B de ordem  $m=2$  com três níveis:

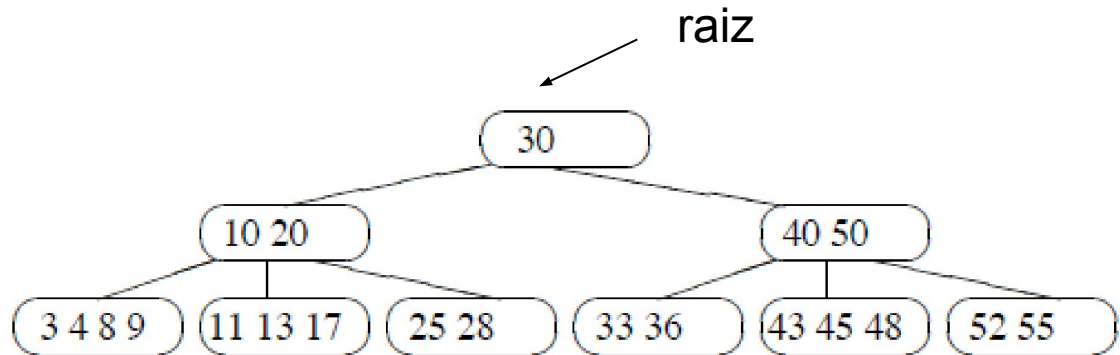


# TAD Árvore B

```
const int M = 2;  
class Registro {  
    int chave;  
    // Outros dados:  
    // ...  
};
```

```
class Pagina {  
    int n;  
    Registro registros[2*M];  
    Pagina * filhos[2*M+1];  
};
```

```
class ArvoreB {  
    Pagina * raiz;  
    // Operações:  
    // ...  
};
```



## ■ Operações:

- ❑ Inicializar

- ❑ Pesquisar

- ❑ Inserir

- ❑ Remover

---

## Árvore B – Inicializar

```
void ArvoreB :: inicializa() {  
    raiz = NULL;  
}
```

# Árvore B - Pesquisa

```
Registro * Pagina :: pesquisa(Registro x) {  
    if (this == NULL) {  
        printf("Registro não encontrado.");  
        return NULL;  
    }
```

```
    int i = 1;  
    while (i < n && x.chave > registros[i-1].chave) i++;
```

} busca dentro da página

```
    if (x.chave == registros[i-1].chave) //Encontrou x  
        return &registros[i-1];
```

```
    if (x.chave < registros[i-1].chave)  
        return filhos[i-1] -> pesquisa(x);
```

} busca na sub-árvore à esquerda do registro *i*

```
    else  
        return filhos[i] -> pesquisa(x);
```

} busca na sub-árvore à direita do registro *i*

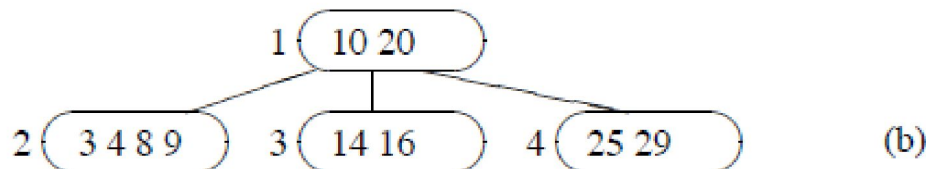
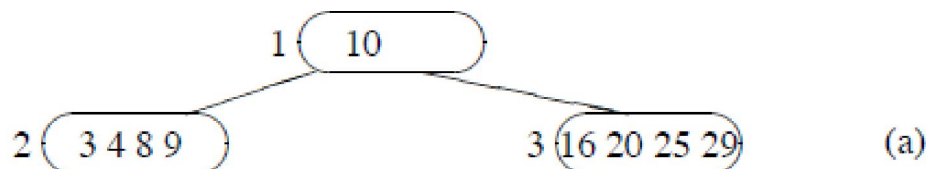
```
}
```

# Árvore B - Inserção

1. Localizar a página onde o registro deve ser inserido
2. Se a página encontrada tem menos de  $2m$  registros, o processo de inserção fica limitado à página
3. Se a página encontrada está cheia, é criada uma nova página. No caso da página “pai” estar cheia, o processo de divisão se propaga

# Árvore B - Inserção

1. Localizar a página onde o registro deve ser inserido
  2. Se a página encontrada tem menos de  $2m$  registros, o processo de inserção fica limitado à página
  3. Se a página encontrada está cheia, é criada uma nova página. No caso da página “pai” estar cheia, o processo de divisão se propaga
- Ex.:  $m=2$ . Inserir o registro com chave 14:

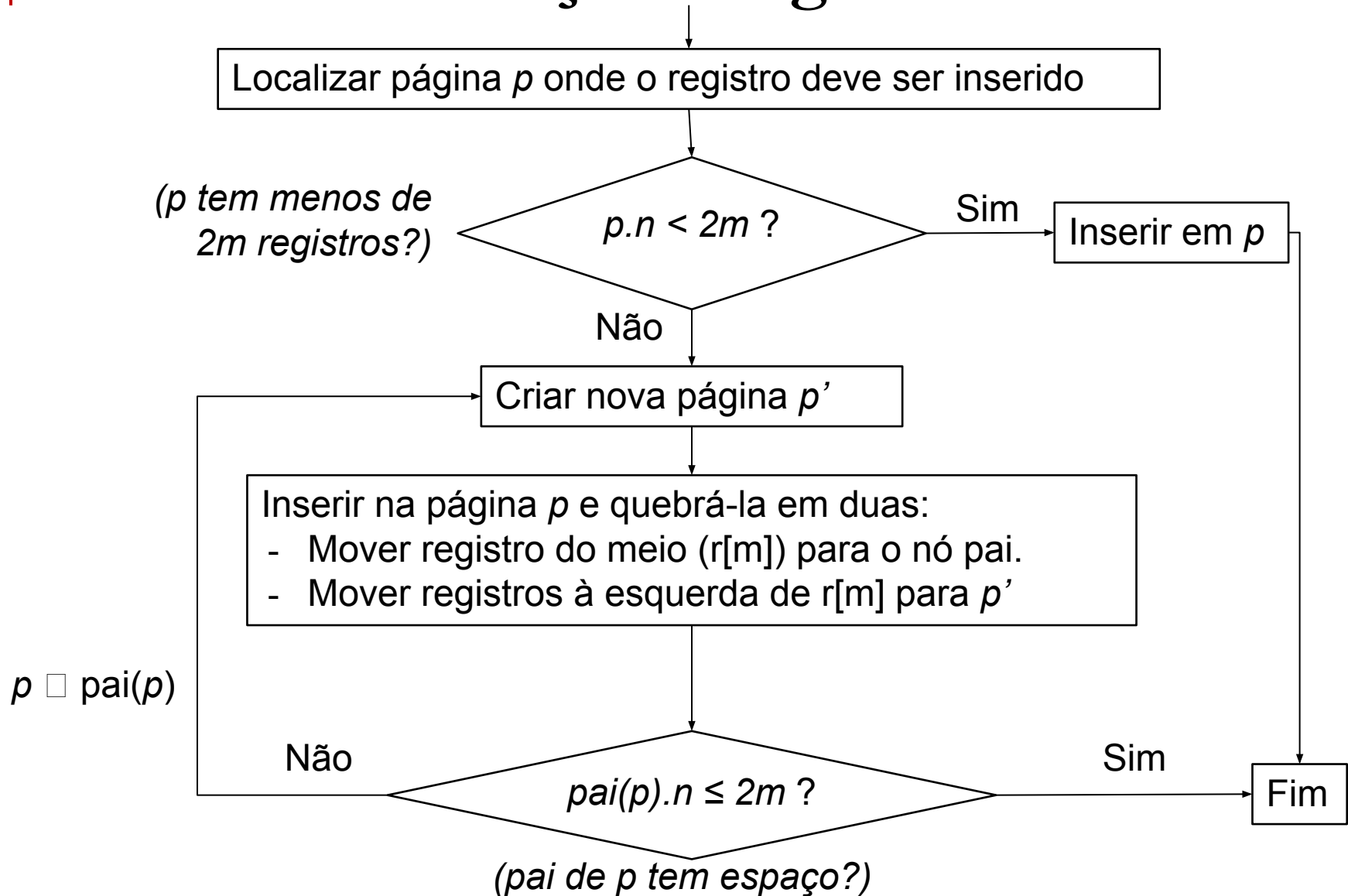




# Árvore B - Inserção

- Se a página a receber o novo registro contém  $2m$  registros, a mesma é quebrada em duas, cada uma com  $m$  registros. O  $(m+1)$ -ésimo registro (o do meio) é movido para o nó pai
- Se a página-pai estiver cheia, o mesmo procedimento de divisão é repetido recursivamente
- No pior caso, uma nova raiz é criada, aumentando a altura da árvore
- Obs.: Uma árvore B somente aumenta a altura com a divisão da raiz

# Árvore B – Inserção - Algoritmo



# Inserção - Código

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

# Inserção - Código

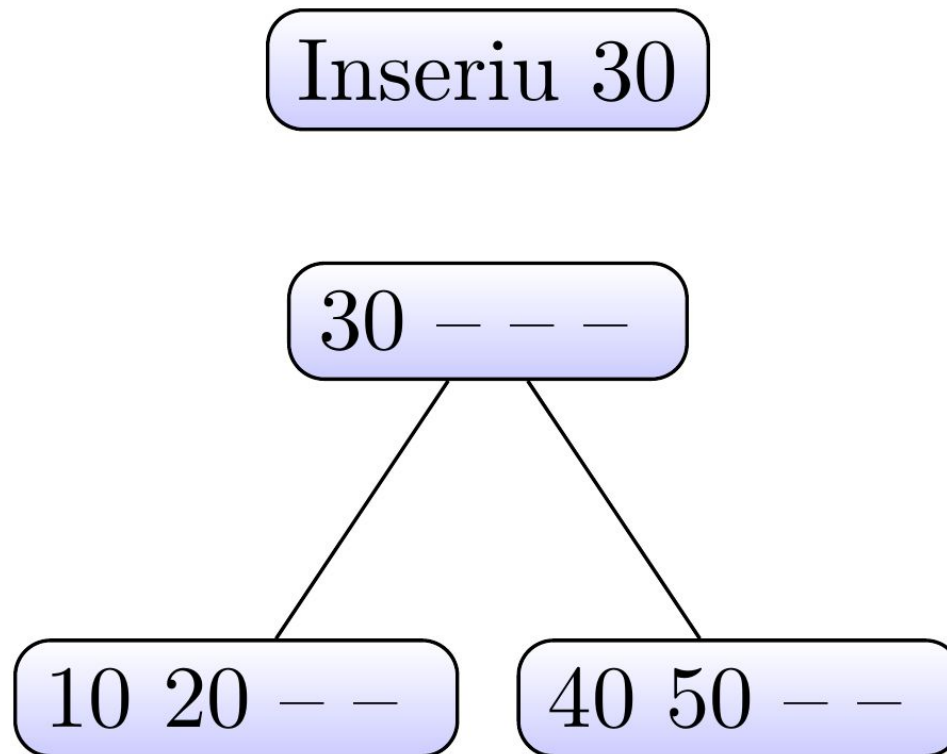
```
void Ins(Registro Reg, Apontador Ap, short *Cresceu, Registro *RegRetorno, Apontador *ApRetorno)
{ long i = 1,j; Apontador ApTemp;
  if (Ap == NULL) { *Cresceu = TRUE; (*RegRetorno) = Reg; (*ApRetorno) = NULL; return; }
  while (i < Ap->n && Reg.Chave > Ap->r[i-1].Chave) i++;
  if (Reg.Chave == Ap->r[i-1].Chave)
    { printf(" Erro: Registro ja esta presente %ld\n",Reg.Chave); *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;
    }
  ApTemp = (Apontador)malloc(sizeof(Pagina));
  ApTemp->n = 0; ApTemp->p[0] = NULL;
  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 - Exemplo

Inseriu 20, 10, 40 e 50

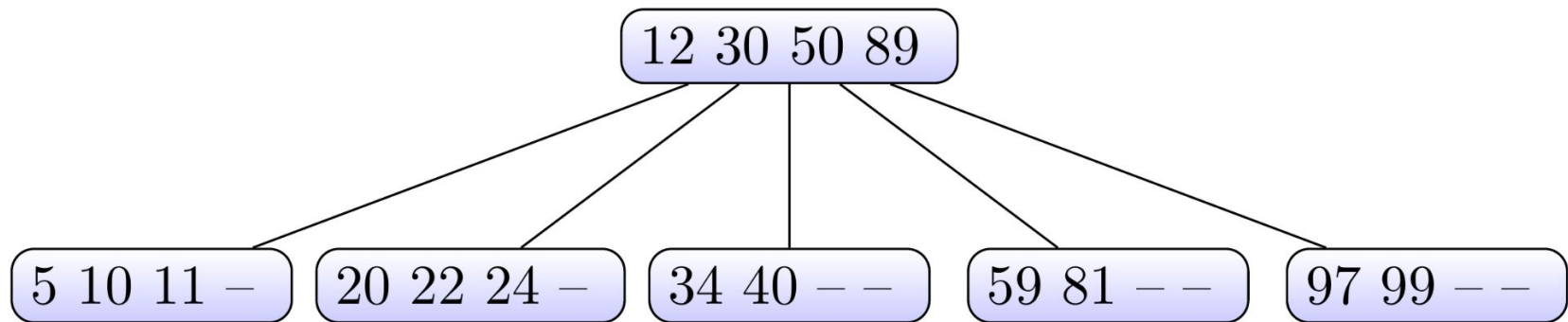
10 20 40 50

# Árvore B - Exemplo



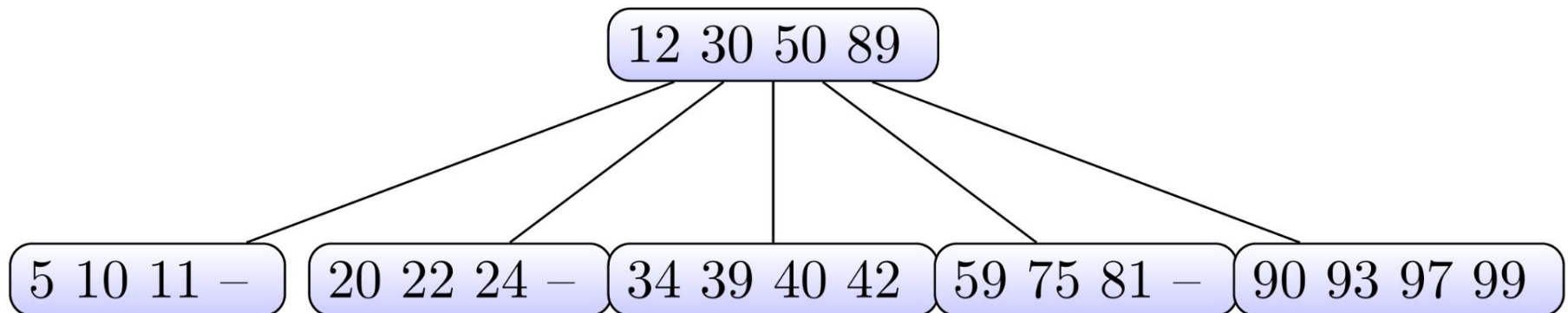
# Árvore B - Exemplo

Inseriu 59,81,34,11,24,99,12,89,5,97,22



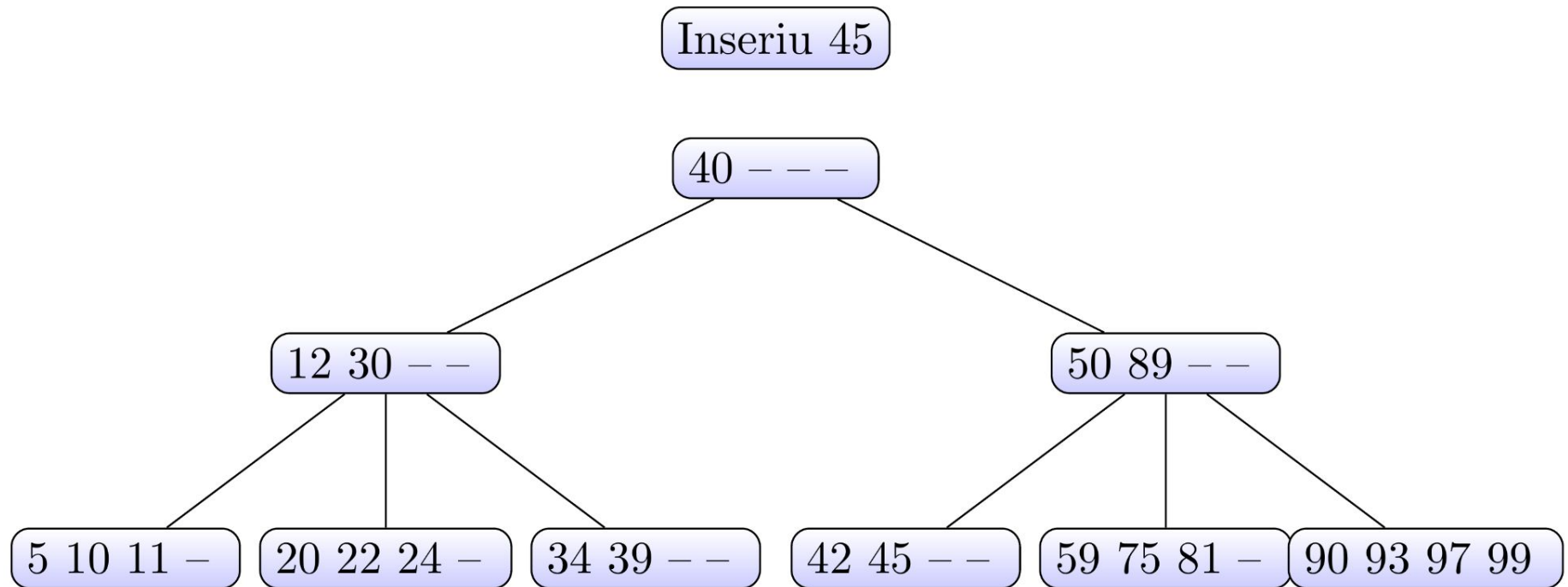
# Árvore B - Exemplo

Inseriu 39,42,90,75,93

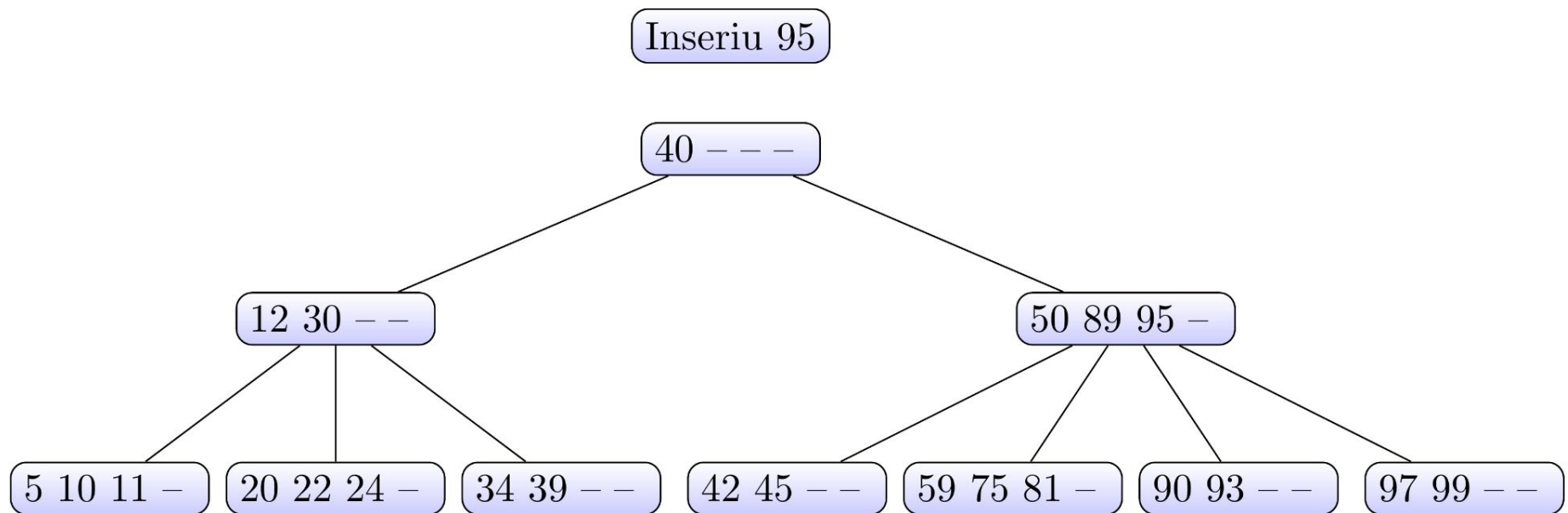




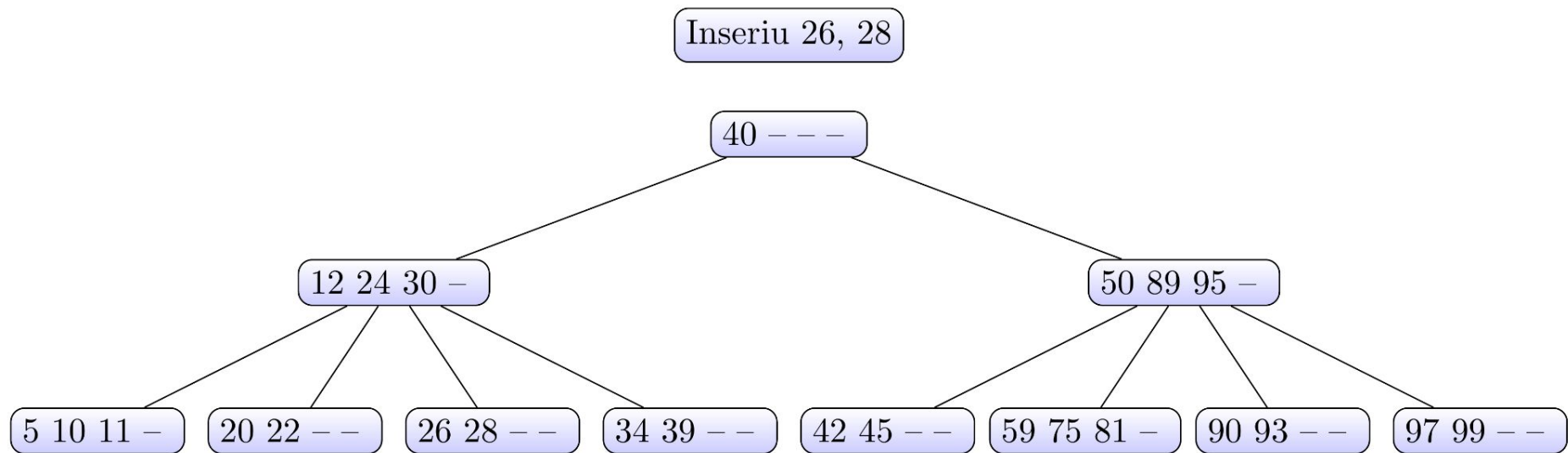
# Árvore B - Exemplo



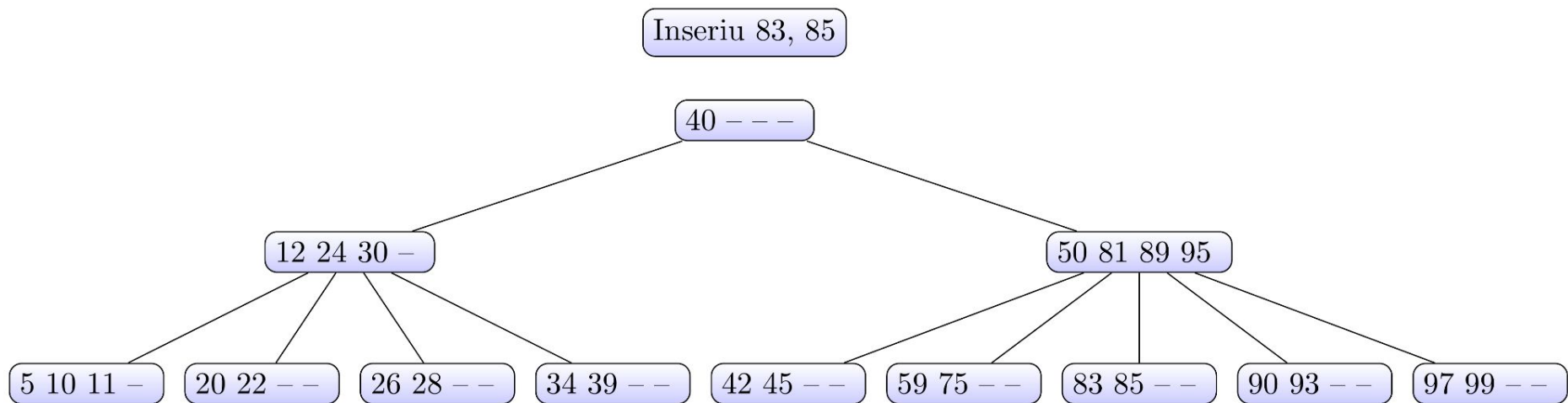
# Árvore B - Exemplo



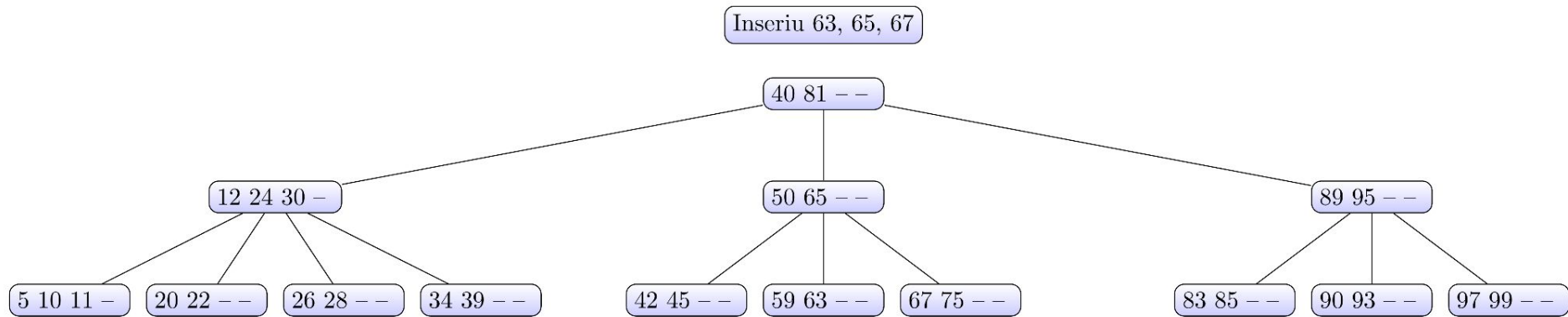
# Árvore B - Exemplo



# Árvore B - Exemplo



# Árvore B - Exemplo



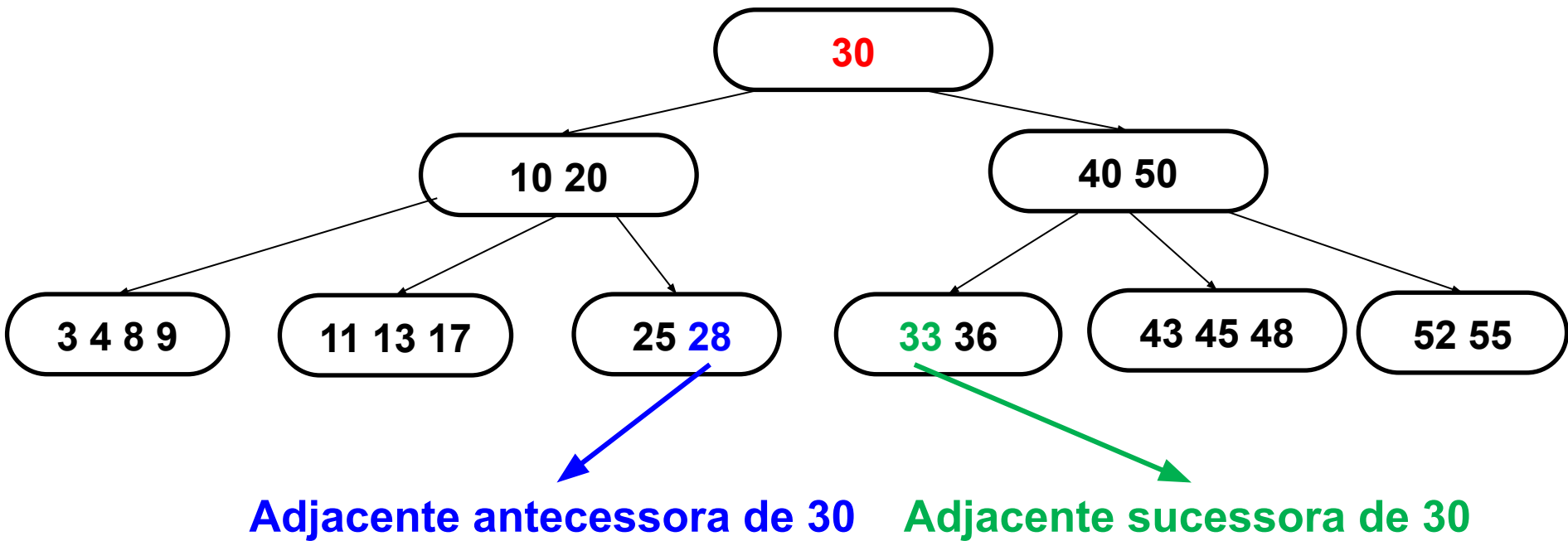
# Árvore B - Remoção

- Página com registro a ser removido é folha:
  - Retira-se o registro
  - Se a página não possui pelo menos  $m$  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.
- Página com o registro não é folha:
  - O registro a ser retirado deve ser primeiramente substituído por um registro contendo uma chave adjacente.

# Árvore B - Remoção

- Quando o registro a ser removido não pertence a uma página-folha, o mesmo deve ser substituído pelo registro de chave adjacente
- Chave adjacente *antecessora*: está na página-folha mais à direita na sub-árvore à esquerda
- Chave adjacente *sucessora*: está na página-folha mais à esquerda na sub-árvore à direita

# Chaves adjacentes - Exemplo

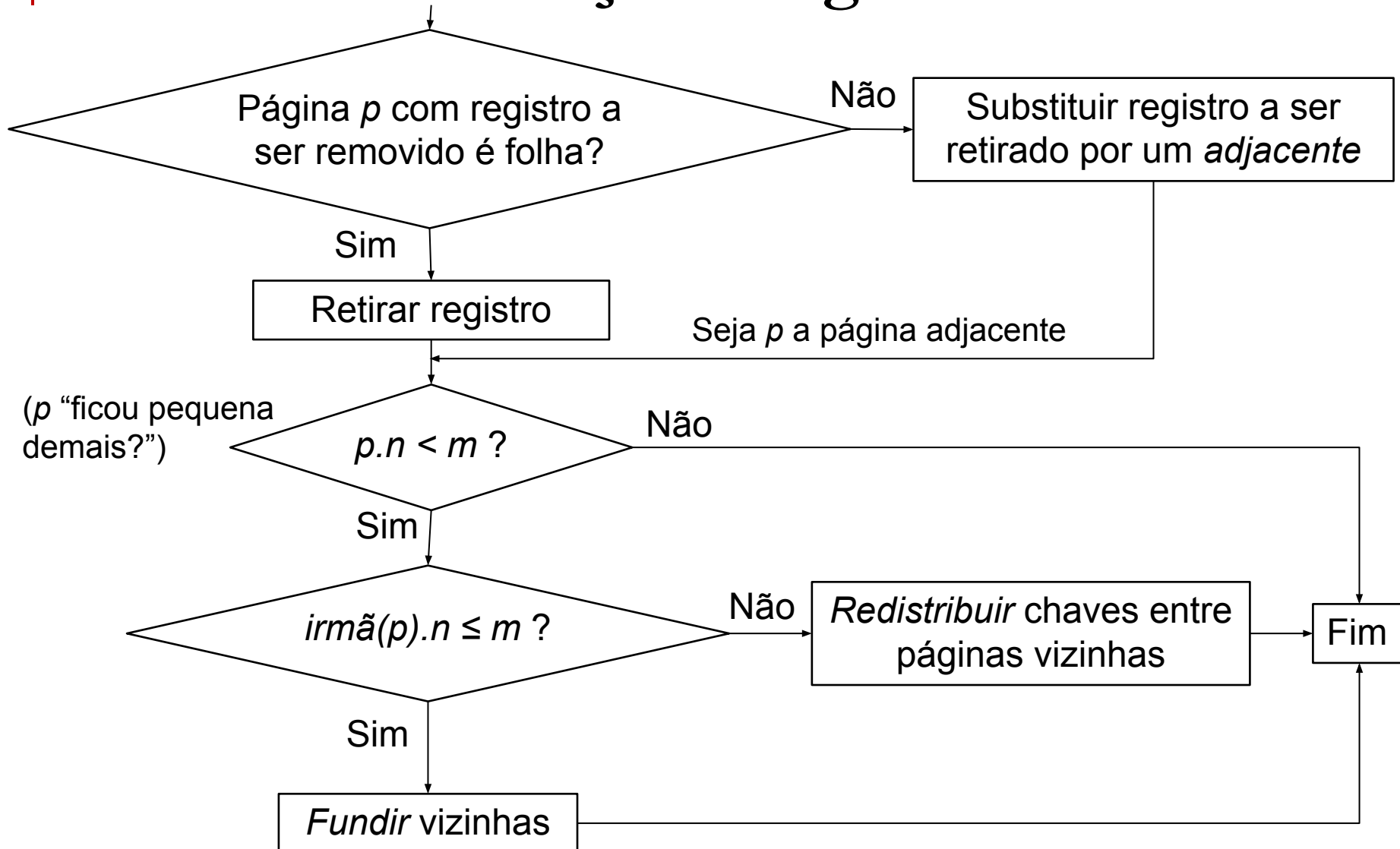




# Árvore B - Remoção

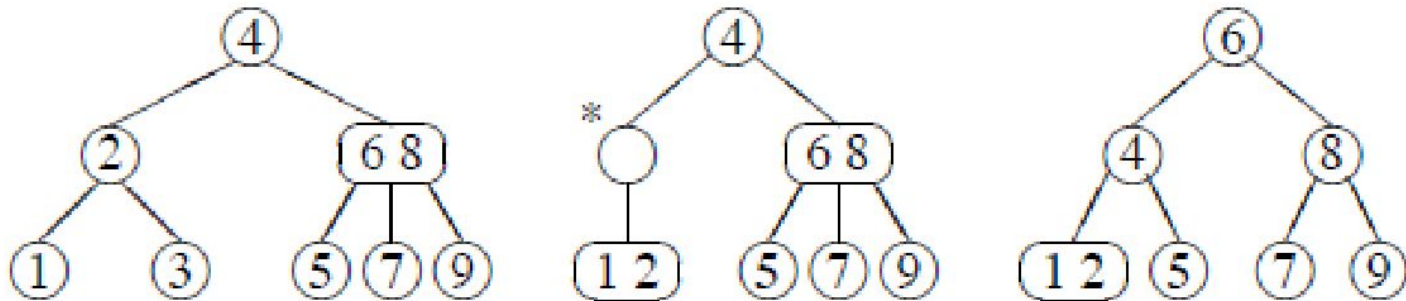
- Se o número de registros restantes na página-folha for  $< m$ , um registro da página vizinha deve ser emprestado
- Se a página vizinha tiver  $m$  registros apenas, as duas páginas devem ser fundidas, pois possuem  $2m-1$  registros
- Fundindo duas páginas:
  - O registro do meio deve ser emprestado do nó-pai
  - O procedimento é propagado até a raiz
  - Se o número de registros na raiz zerar, reduzir altura da árvore

# Árvore B – Remoção - Algoritmo

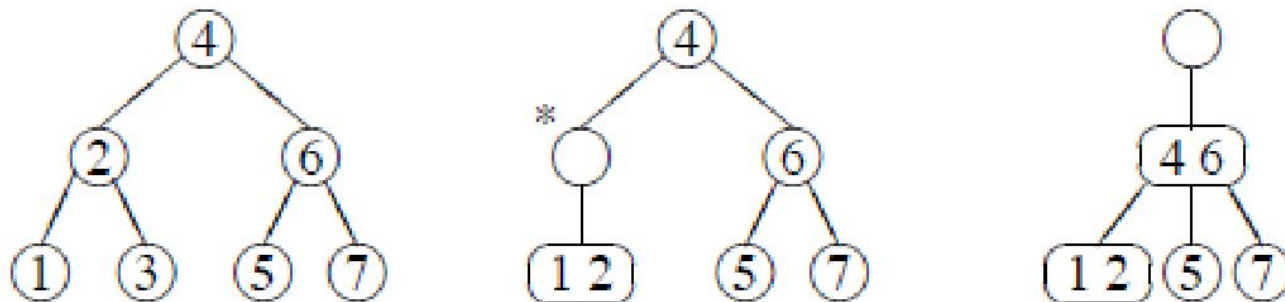


# Remoção - Exemplo

- Ordem  $m=1$ . Retirando a chave 3



(a) Página vizinha possui mais do que  $m$  registros



(b) Página vizinha possui exatamente  $m$  registros

# Remoção - Pseudo Código

Ret(Ch, Ap, Diminuiu)

Se Ap == NULL, não está na árvore

Se Ch no registro

Se página folha,

Remove

Diminuiu =  $n < m$

Senão

Antecessor

Se Diminuiu, Reconstitui

Ret(Ch, Ap->[i], Diminuiu)

Se Diminuiu, Reconstitui

# Remoção - Pseudo Código

```
Reconstitui(ApPag, ApPai, PosPai, Diminuiu)
  Se PosPai < ApPai -> n // Verifica irmão à direita
    Se Irmão à direita tem folga
      Transfere Chave passando pelo pai
      Diminuiu = False
    Senão
      Fusão com o irmão à direita
      Se Pai >= m, Diminuiu = False
  Senão
    Se Irmão à esquerda tem folga
      Transfere Chave passando pelo pai
      Diminuiu = False
    Senão
      Fusão com irmão à esquerda
      Se Pai >= m, Diminuiu = False
```

# Remoção - Algoritmo

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

void Antecessor(Apontador Ap, int Ind, Apontador 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);
}
```

# Remoção - Algoritmo

```
void Ret(TipoChave Ch, Apontador *Ap, short *Diminuiu)
{ long j, Ind = 1;
  Apontador 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++; // Procura chave
  if (Ch == Pag->r[Ind-1].Chave) // Achou Ch em Pag
  { if (Pag->p[Ind-1] == NULL) /* Pagina folha */
    { Pag->n--;
      *Diminuiu = (Pag->n < m); // Folha ainda tem m registros?
      for (j = Ind; j <= Pag->n; j++) { Pag->r[j-1] = Pag->r[j]; Pag->p[j] = Pag->p[j+1]; }
      return;
    }
    Antecessor(*Ap, Ind, Pag->p[Ind-1], Diminuiu); /* Pagina nao e folha: trocar com antecessor */
    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);
}
```

# Remoção - Algoritmo

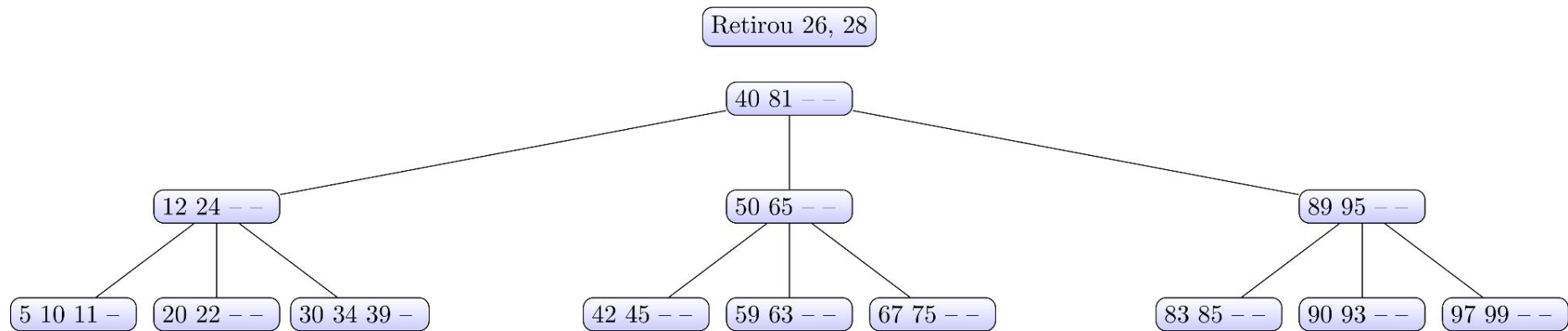
```
void Reconstitui(Apontador ApPag, Apontador ApPai, int PosPai, short *Diminuiu)
{
    Pagina *Aux;
    long DispAux, j;
    if (PosPai < ApPai->n) /* Aux = Pagina a direita de ApPag */
    {
        Aux = ApPai->p[PosPai+1]; DispAux = (Aux->n - m + 1) / 2;
        ApPag->r[ApPag->n] = ApPai->r[PosPai];
        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++) InereNaPagina(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++) 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;
    }
}
else .... (próximo slide)
```



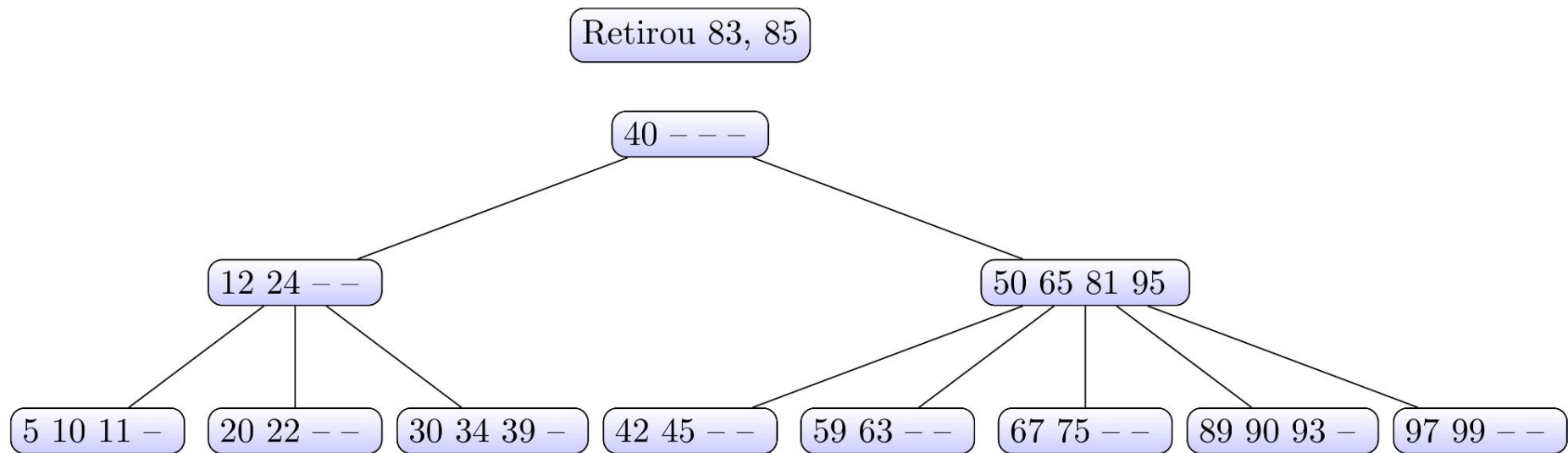
# Remoção - Algoritmo

```
void Reconstitui(Apontador ApPag, Apontador ApPai, int PosPai, short *Diminuiu)
{
    Pagina *Aux;
    long DispAux, j;
    if (PosPai < ApPai->n) /* Aux = Pagina a direita de ApPag */
    {
        .... (slide anterior)
    }
    else { /* Aux = Pagina 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: transfere de Aux para ApPag */
        {
            for (j = 1; j < DispAux; j++) InseNaPagina(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++) InseNaPagina(Aux, ApPag->r[j-1], ApPag->p[j]);
            free(ApPag); ApPai->n--;
            if (ApPai->n >= m) *Diminuiu = FALSE;
        }
    }
}
```

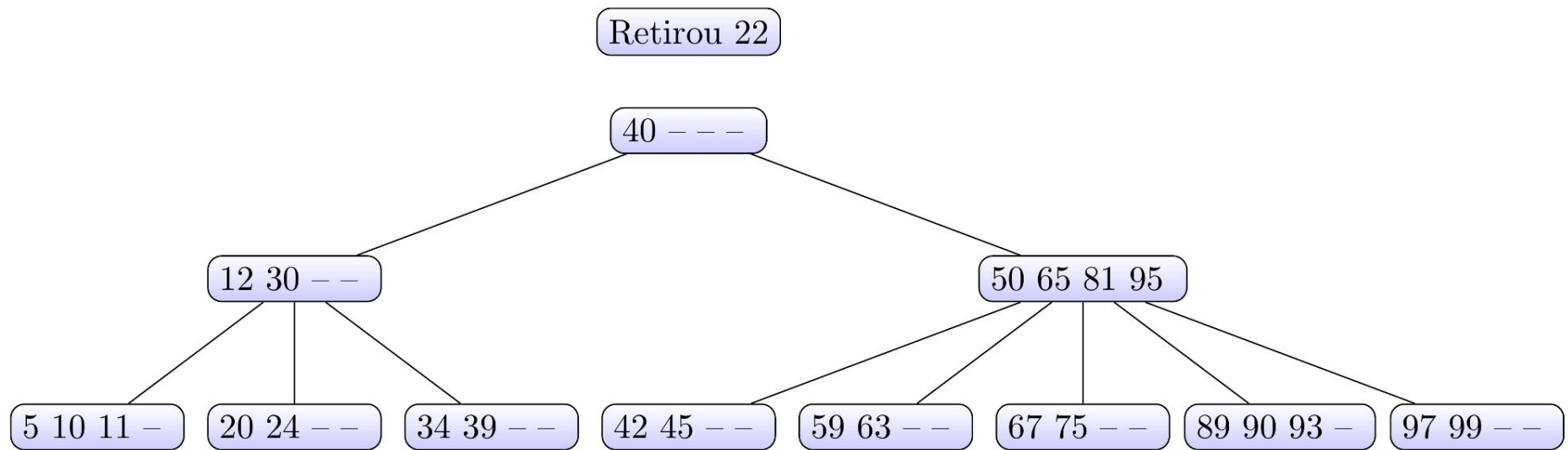
# Árvore B - Exemplo



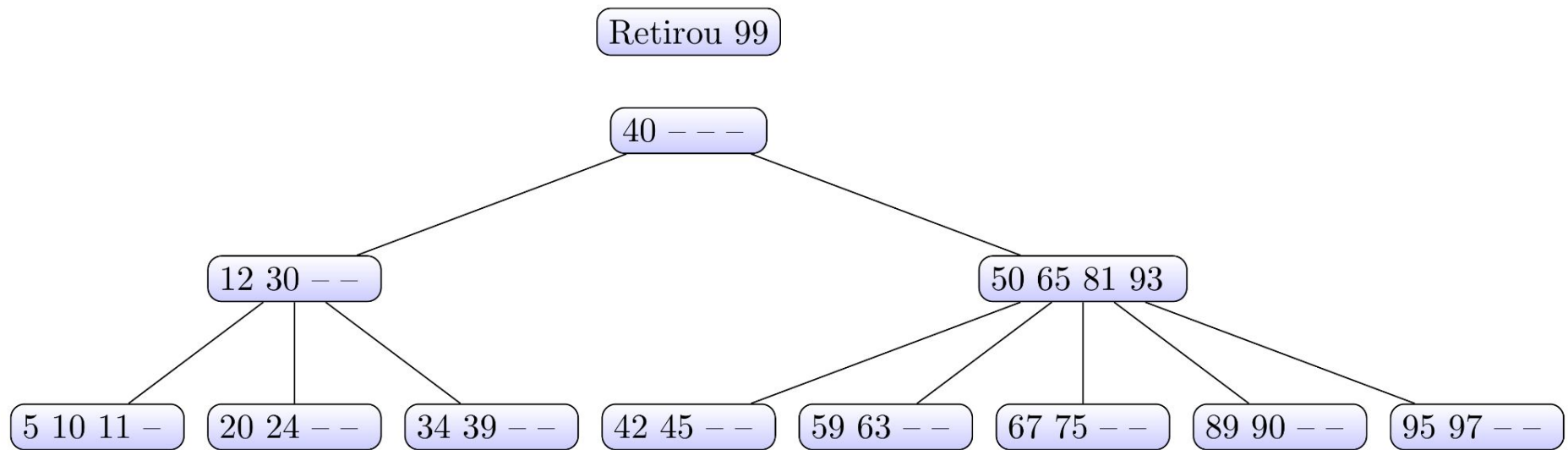
# Árvore B - Exemplo



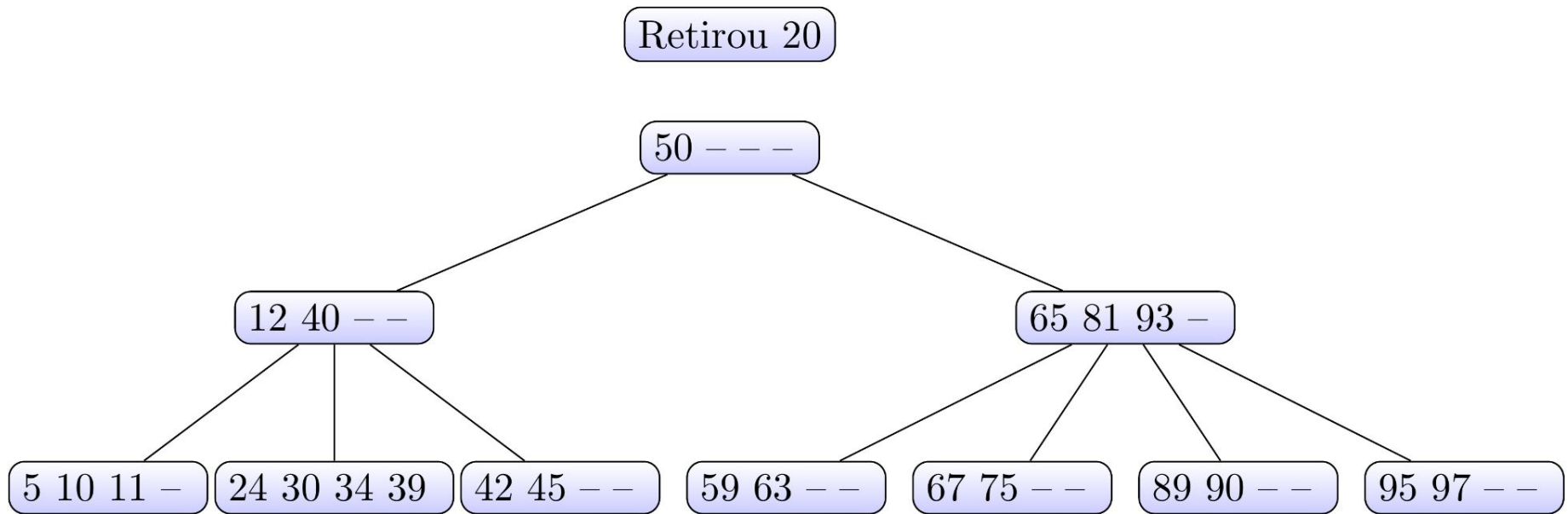
# Árvore B - Exemplo



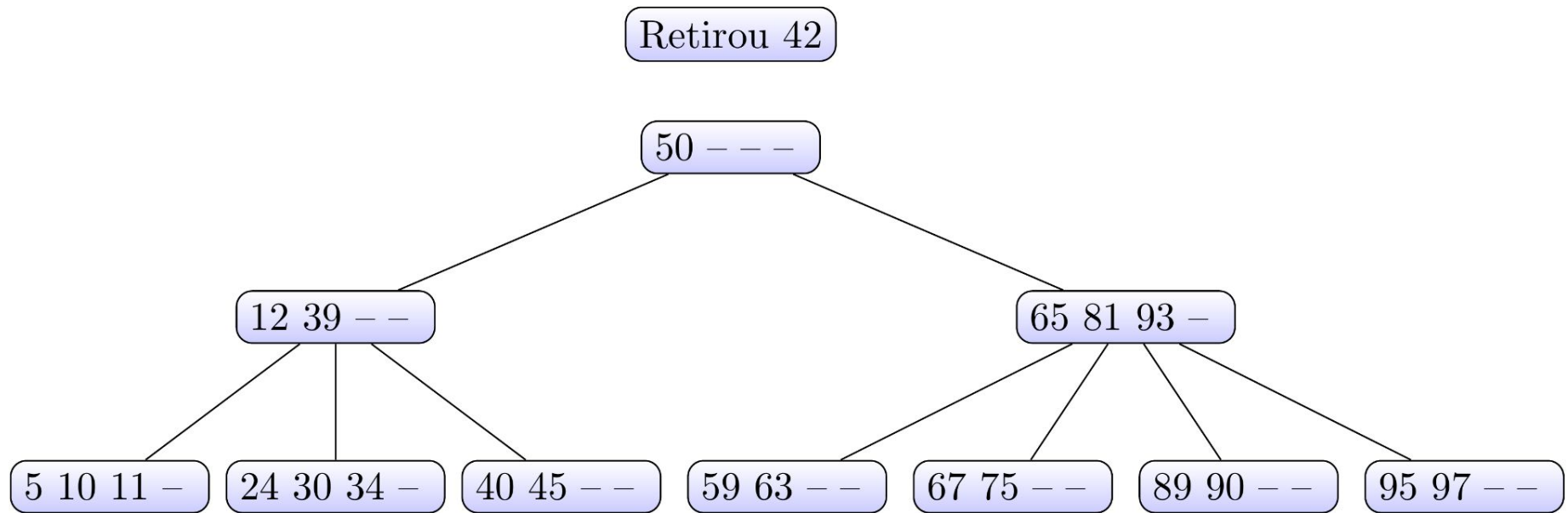
# Árvore B - Exemplo



# Árvore B - Exemplo

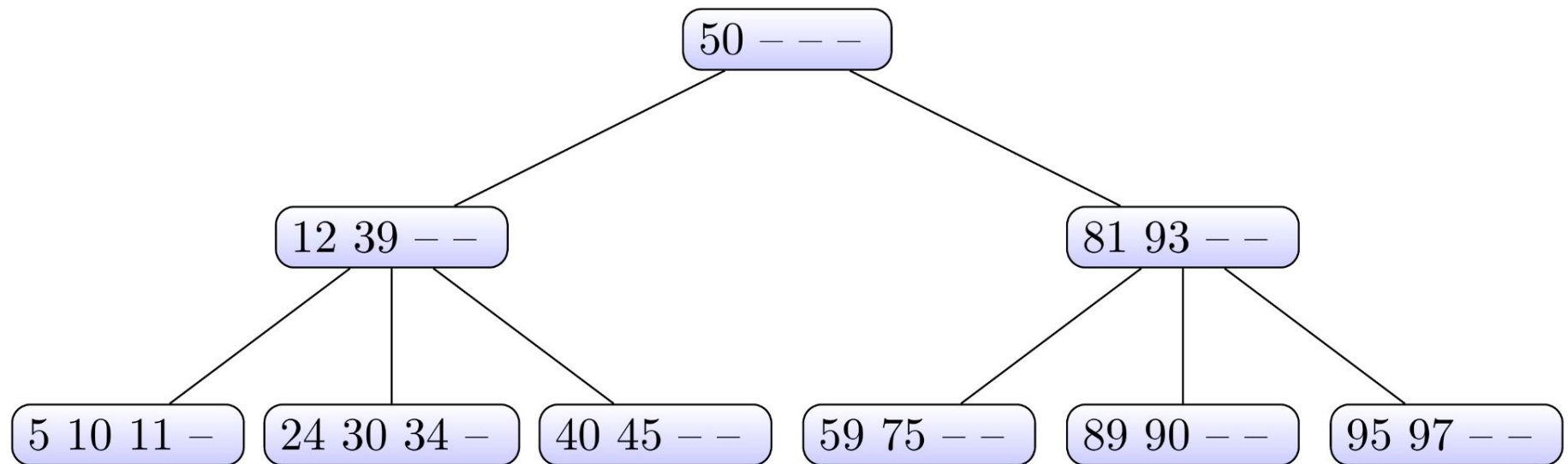


# Árvore B - Exemplo



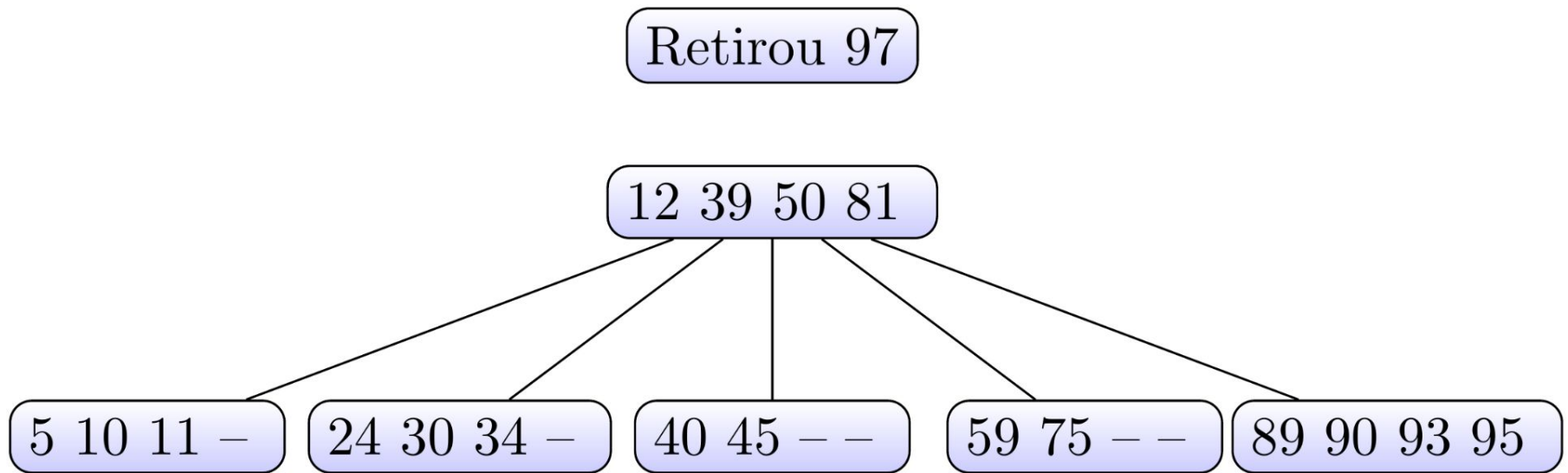
# Árvore B - Exemplo

Retirou 63, 65, 67

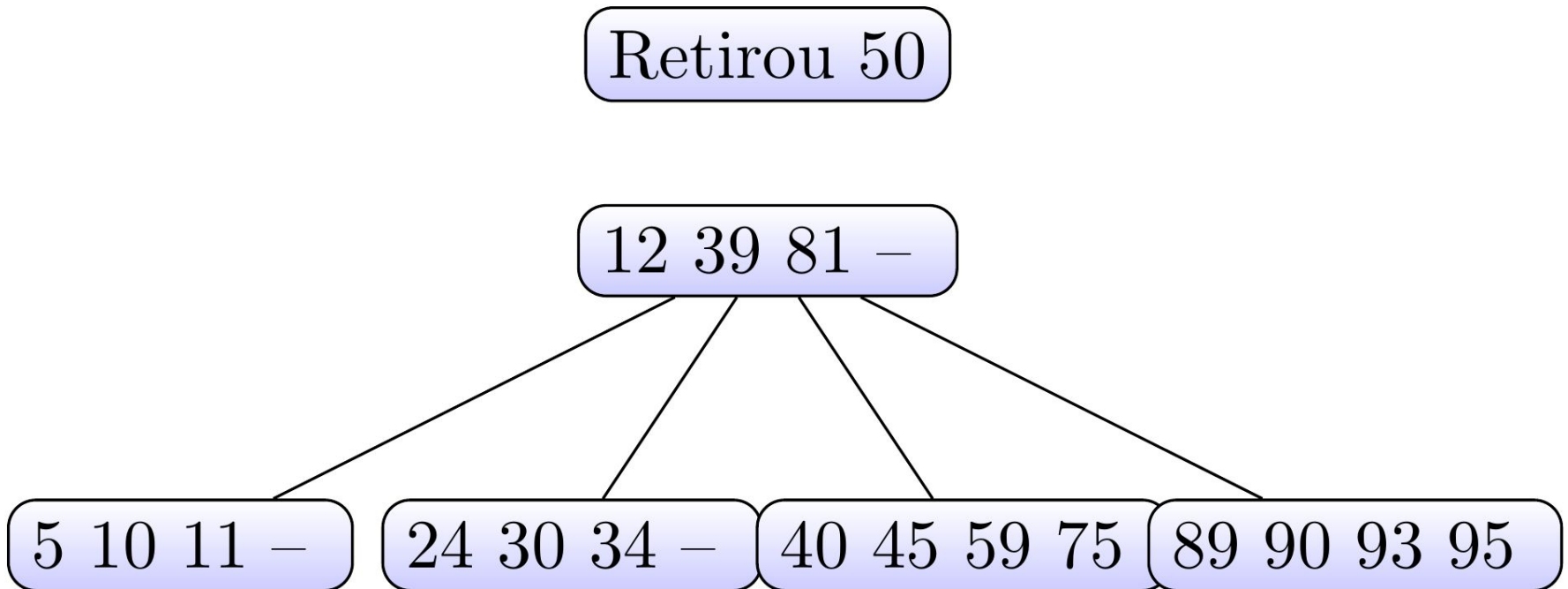




# Árvore B - Exemplo



# Árvore B - Exemplo



# Árvore B – Considerações práticas

- Simples, fácil manutenção, eficiente e versátil
- Permite acesso sequencial eficiente
- Custo para pesquisar, inserir e retirar registros do arquivo é logarítmico
- Emprego onde o acesso concorrente ao banco de dados é necessário é viável e relativamente simples de ser implementado
- Inserção e remoção de registros sempre deixam a árvore balanceada
- Uma árvore B de ordem  $m$  com  $N$  registros contém no máximo  $\log_{m+1} N$  páginas

# Árvore B – Considerações práticas

- Dada uma árvore B de ordem  $m$  com  $N$  registros:

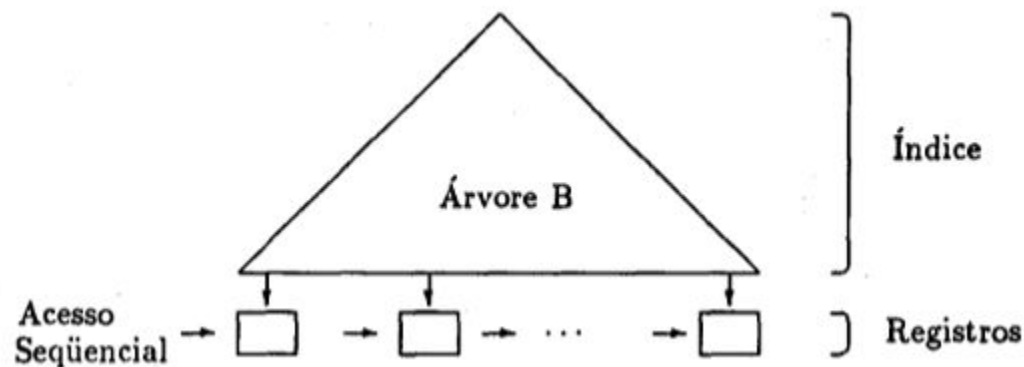
$$\log_{2m+1}(N + 1) \leq altura \leq 1 + \log_{m+1}\left(\frac{N + 1}{2}\right)$$

- Custo para processar uma operação de pesquisa 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 folhas em direção à raiz) de uma árvore 2-3 (árvore B de ordem  $m=1$ ):
- Conjectura: a altura esperada de uma árvore 2-3 aleatória com  $N$  chaves é  $h(N) \cong \log_{7/3}(N + 1)$

# Árvores B\*

# Árvore B\*

- Similar à Árvore B.
- Os registros ficam gravados apenas em nós folha.
- Nos nós internos ficam gravados apenas as chaves dos registros, para auxiliar na localização dos mesmos.
- As folhas também são encadeadas da esquerda para a direita, permitindo acesso sequencial das páginas.
- Essas mudanças fazem com que a inserção e remoção se tornem procedimentos mais simples.



# Árvore B\* - Pesquisa

- Similar à Árvore B.
  - Diferença: Uma vez que os registros estão gravados apenas nas folhas, não podemos encerrar a busca caso a chave seja encontrada em um nó interno.
- **Convenção:** Se uma chave  $k$  está presente em um nó interno, então seu registro estará no seu filho da direita.
  - Dessa forma conseguimos lidar com a presença das duplicatas das chaves e encontrar o registro desejado.
- A pesquisa segue até encontrar o registro com a chave desejada em uma folha.

# Árvore B\* - Inserção

- Também similar à Árvore B.
  - ❑ Diferença: O registro só pode ser inserido em um nó folha. Caso essa nó exceda sua capacidade máxima, durante o processo de divisão, apenas uma cópia da chave do registro do meio será propagada para os nós internos.
- **Convenção:** Se uma chave  $k$  está presente em um nó interno, então seu registro estará no seu filho da direita.
  - ❑ Sendo assim, caso o nó tenha que ser dividido após a inserção, o registro do meio deve ficar no filho à direita.



# Árvore B\* - Remoção

- Relativamente mais simples que a Árvore B.
  - ❑ O registro a ser removido sempre estará em uma folha, logo não precisamos computar o elemento adjacente.
  - ❑ Desde que a página folha fique pelo menos com metade dos registros, as páginas do índice não precisam ser modificadas, mesmo se uma cópia da chave que pertence ao registro a ser retirado esteja no índice.