Pesquisa em Memória Primária*

Última alteração: 7 de Setembro de 2010

*Transparências elaboradas por Fabiano C. Botelho, Israel Guerra e Nivio Ziviani

Pesquisa em Memória Primária

- Introdução - Conceitos Básicos
- Pesquisa Sequencial
- Pesquisa Binária
- Árvores de Pesquisa
  - Árvores Binárias de Pesquisa sem Balanceamento
  - Árvores Binárias de Pesquisa com Balanceamento
    - Árvores SBB
    - Transformações para Manutenção da Propriedade SBB
- Pesquisa Digital
  - Trie
  - Patricia
- Transformação de Chave (*Hashing*)
  - Funções de Transformação
  - Listas Encadeadas
  - Endereçamento Aberto
  - *Hashing* Perfeito com ordem Preservada
  - *Hashing* Perfeito Usando Espaço Quase Ótimo

Introdução - Conceitos Básicos

- Estudo de como recuperar informação a partir de uma grande massa de informação previamente armazenada.
- A informação é dividida em **registros**.
- Cada registro possui uma chave para ser usada na pesquisa.
- **Objetivo da pesquisa:**

  Encontrar uma ou mais ocorrências de registros com chaves iguais à **c**have de pesquisa.

- **Pesquisa com sucesso** X **Pesquisa sem sucesso**.

Introdução - Conceitos Básicos

- Conjunto de registros ou arquivos $\Rightarrow$ tabelas
- **Tabela:**

  associada a entidades de vida curta, criadas na memória interna durante a execução de um programa.
- **Arquivo:**

  geralmente associado a entidades de vida mais longa, armazenadas em memória externa.
- **Distinção não é rígida:**

  tabela: arquivo de índices

  arquivo: tabela de valores de funções.

Escolha do Método de Pesquisa mais Adequado a uma Determinada Aplicação

- **Depende principalmente:**
  1. Quantidade dos dados envolvidos.
  2. Arquivo estar sujeito a inserções e retiradas frequentes.

  Se conteúdo do arquivo é estável é importante minimizar o tempo de pesquisa, sem preocupação com o tempo necessário para estruturar o arquivo

Algoritmos de Pesquisa ⇒ Tipos Abstratos de Dados

- É importante considerar os algoritmos de pesquisa como **tipos abstratos de dados**, com um conjunto de operações associado a uma estrutura de dados, de tal forma que haja uma independência de implementação para as operações.
- **Operações mais comuns:**
  1. Inicializar a estrutura de dados.
  2. Pesquisar um ou mais registros com determinada chave.
  3. Inserir um novo registro.
  4. Retirar um registro específico.
  5. Ordenar um arquivo para obter todos os registros em ordem de acordo com a chave.
  6. Ajuntar dois arquivos para formar um arquivo maior.

Dicionário

- Nome comumente utilizado para descrever uma estrutura de dados para pesquisa.
- **Dicionário** é um **tipo abstrato de dados** com as operações:
  1. Inicializa
  2. Pesquisa
  3. Insere
  4. Retira
- Analogia com um dicionário da língua portuguesa:
  - Chaves $\Longleftrightarrow$ palavras
  - Registros $\Longleftrightarrow$ entradas associadas com cada palavra:
    - pronúncia
    - definição
    - sinônimos
    - outras informações

Pesquisa Sequencial

- **Método de pesquisa mais simples:** a partir do primeiro registro, pesquise sequencialmente até encontrar a chave procurada; então pare.
- Armazenamento de um conjunto de registros por meio do tipo estruturado arranjo:

```
#define MAXN 10
typedef long TipoChave;
typedef struct TipoRegistro {
  TipoChave Chave;
  /* outros componentes */
} TipoRegistro;
typedef int TipoIndice;
typedef struct TipoTabela {
  TipoRegistro Item[MAXN + 1];
  TipoIndice n;
} TipoTabela;
```

Pesquisa Sequencial

```
void Inicializa(TipoTabela *T)
{ T->n = 0; }

TipoIndice Pesquisa(TipoChave x, TipoTabela *T)
{ int i;
  T->Item[0].Chave = x;  i = T->n + 1;
  do {i--;} while (T->Item[i].Chave != x);
  return i;
}

void Insere(TipoRegistro Reg, TipoTabela *T)
{ if (T->n == MAXN)
  printf("Erro : tabela cheia\n");
  else { T->n++; T->Item[T->n] = Reg; }
}
```

Pesquisa Sequencial

- Pesquisa retorna o índice do registro que contém a chave x;
- Caso não esteja presente, o valor retornado é zero.
- A implementação não suporta mais de um registro com uma mesma chave.
- Para aplicações com esta característica é necessário incluir um argumento a mais na função Pesquisa para conter o índice a partir do qual se quer pesquisar.

Pesquisa Sequencial

- Utilização de um registro **sentinela** na posição zero do **array**:
  1. Garante que a pesquisa sempre termina:
     se o índice retornado por Pesquisa for zero, a pesquisa foi sem sucesso.
  2. Não é necessário testar se $i > 0$, devido a isto:
     - o anel interno da função Pesquisa é extremamente simples: o índice i é decrementado e a chave de pesquisa é comparada com a chave que está no registro.
     - isto faz com que esta técnica seja conhecida como **pesquisa sequencial rápida**.

Pesquisa Binária

- **Pesquisa em tabela pode ser mais eficiente $\Rightarrow$ Se registros forem mantidos em ordem**
- Para saber se uma chave está presente na tabela
  1. Compare a chave com o registro que está na posição do meio da tabela.
  2. **Se** a chave é menor **então** o registro procurado está na primeira metade da tabela
  3. **Se** a chave é maior **então** o registro procurado está na segunda metade da tabela.
  4. Repita o processo até que a chave seja encontrada, ou fique apenas um registro cuja chave é diferente da procurada, significando uma pesquisa sem sucesso.

Algoritmo de Pesquisa Binária

```
TipoIndice Binaria(TipoChave x, TipoTabela *T)
{ TipoIndice i, Esq, Dir;
  if (T->n == 0)
  return 0;
  else
  { Esq = 1;
    Dir = T->n;
    do
      { i = (Esq + Dir) / 2;
        if (x > T->Item[i].Chave)
        Esq = i + 1;
        else Dir = i - 1;
      } while (x != T->Item[i].Chave && Esq <= Dir);
    if (x == T->Item[i].Chave) return i; else return 0;
  }
}
```

Pesquisa para a chave G:

1	2	3	4	5	6	7	8
A	*B*	*C*	***D***	*E*	*F*	*G*	*H*
				E	***F***	*G*	*H*
						G	*H*

Pesquisa Binária: Análise

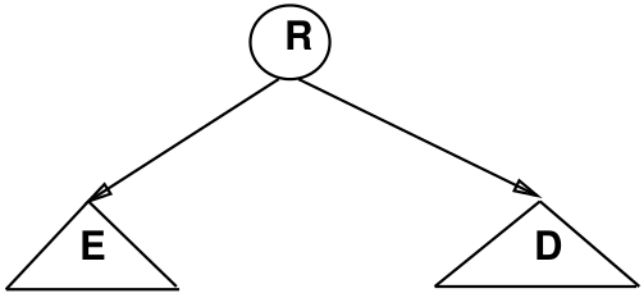
- A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio.
- **Logo:** o número de vezes que o tamanho da tabela é dividido ao meio é cerca de $\log n$.
- **Ressalva:** o custo para manter a tabela ordenada é alto:
  a cada inserção na posição p da tabela implica no deslocamento dos registros a partir da posição p para as posições seguintes.
- Consequentemente, a pesquisa binária não deve ser usada em aplicações muito dinâmicas.

Árvores de Pesquisa

- A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar informação.
- Particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de:
  1. Acesso direto e sequencial eficientes.
  2. Facilidade de inserção e retirada de registros.
  3. Boa taxa de utilização de memória.
  4. Utilização de memória primária e secundária.

Árvores Binárias de Pesquisa sem Balanceamento

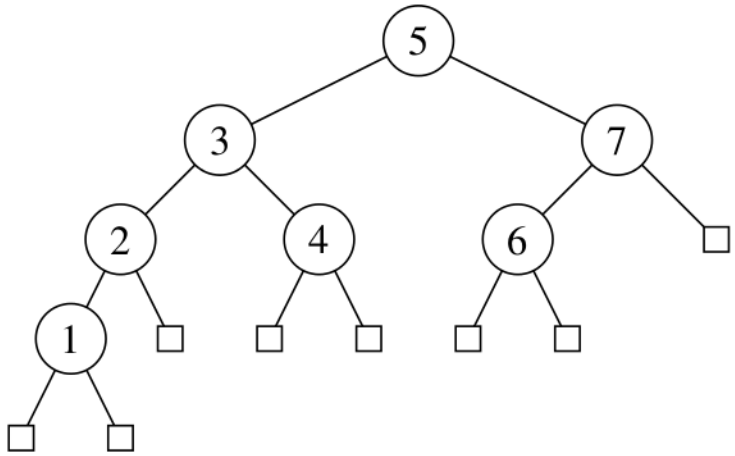
- Para qualquer nó que contenha um registro



Temos a relação invariante

1. Todos os registros com chaves menores estão na subárvore à esquerda.
2. Todos os registros com chaves maiores estão na subárvore à direita.

Árvores Binárias de Pesquisa sem Balanceamento



- O **nível** do nó raiz é 0.
- Se um nó está no nível i então a raiz de suas subárvores estão no nível $i + 1$.
- A **altura** de um nó é o comprimento do caminho mais longo deste nó até um nó folha.
- A altura de uma árvore é a altura do nó raiz.

Implementação do Tipo Abstrato de Dados Dicionário usando a Estrutura de Dados Árvore Binária de Pesquisa

Estrutura de dados:

```
typedef long TipoChave;
typedef struct TipoRegistro {
   TipoChave Chave;
   /* outros componentes */
} TipoRegistro;
typedef struct TipoNo * TipoApontador;
typedef struct TipoNo {
   TipoRegistro Reg;
   TipoApontador Esq, Dir;
} TipoNo;
```

Procedimento para Pesquisar na Árvore Uma Chave x

- Compare-a com a chave que está na raiz.
- Se x é menor, vá para a subárvore esquerda.
- Se x é maior, vá para a subárvore direita.
- Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha é atingido.
- Se a pesquisa tiver sucesso o conteúdo retorna no próprio registro x.

```
void Pesquisa(TipoRegistro *x, TipoApontador *p)
{ if (*p == NULL)
  { printf("Erro: Registro nao esta presente na arvore\n"); return; }
  if (x->Chave < (*p)->Reg.Chave)
  { Pesquisa(x, &(*p)->Esq); return; }
  if (x->Chave > (*p)->Reg.Chave) Pesquisa(x, &(*p)->Dir);
  else *x = (*p)->Reg;
}
```

Procedimento para Inserir na Árvore

- Atingir um apontador nulo em um processo de pesquisa significa uma pesquisa sem sucesso.
- O apontador nulo atingido é o ponto de inserção.

```
void Insere(TipoRegistro x, TipoApontador *p)
{ if (*p == NULL)
  { *p = (TipoApontador)malloc(sizeof(TipoNo));
    (*p)->Reg = x; (*p)->Esq = NULL; (*p)->Dir = NULL;
    return;
  }
  if (x.Chave < (*p)->Reg.Chave)
  { Insere(x, &(*p)->Esq); return; }
  if (x.Chave > (*p)->Reg.Chave)
  Insere(x, &(*p)->Dir);
  else printf("Erro : Registro ja existe na arvore\n");
}
```

Procedimentos para Inicializar e Criar a Árvore

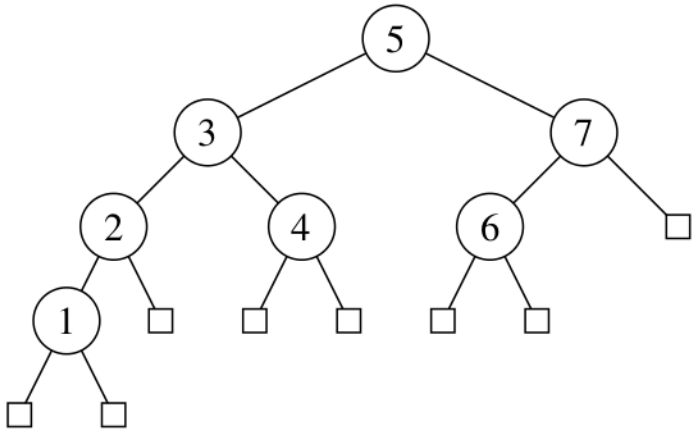
```
void Inicializa(TipoApontador *Dicionario)
{ *Dicionario = NULL; }
end; { Inicializa }

{-- Entra aqui a definição dos tipos mostrados no slide 18 --}
{-- Entram aqui os procedimentos Insere e Inicializa --}
int main(int argc, char *argv[])
{ TipoDicionario Dicionario; TipoRegistro x;
  Inicializa(&Dicionario);
  scanf("%d%*[^\n]", &x.Chave);
  while(x.Chave > 0)
    { Insere(x,&Dicionario);
      scanf("%d%*[^\n]", &x.Chave);
    }
}
```

Procedimento para Retirar x da Árvore

- Alguns comentários:
  1. A retirada de um registro não é tão simples quanto a inserção.
  2. Se o nó que contém o registro a ser retirado possui no máximo um descendente $\Rightarrow$ a operação é simples.
  3. No caso do nó conter dois descendentes o registro a ser retirado deve ser primeiro:
     - substituído pelo registro mais à direita na subárvore esquerda;
     - ou pelo registro mais à esquerda na subárvore direita.

Exemplo da Retirada de um Registro da Árvore



Assim: para retirar o registro com chave 5 na árvore basta trocá-lo pelo registro com chave 4 ou pelo registro com chave 6, e então retirar o nó que recebeu o registro com chave 5.

Procedimento para Retirar x da Árvore

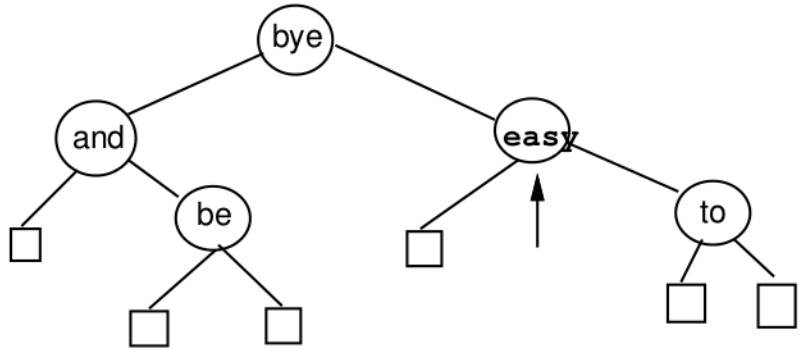
```
void Antecessor(TipoApontador q, TipoApontador *r)
{ if ((*r)->Dir != NULL)
  { Antecessor(q, &(*r)->Dir);
    return;
  }
  q->Reg = (*r)->Reg;
  q = *r;
  *r = (*r)->Esq;
  free(q);
}
```

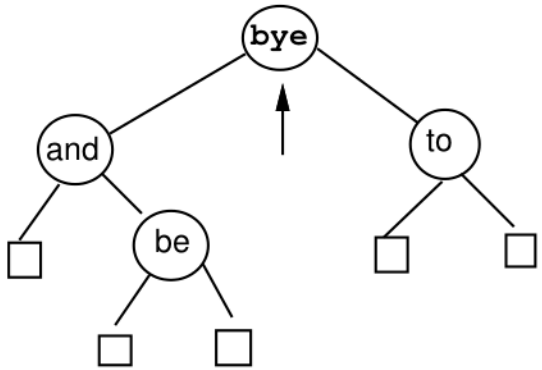
Procedimento para Retirar x da Árvore

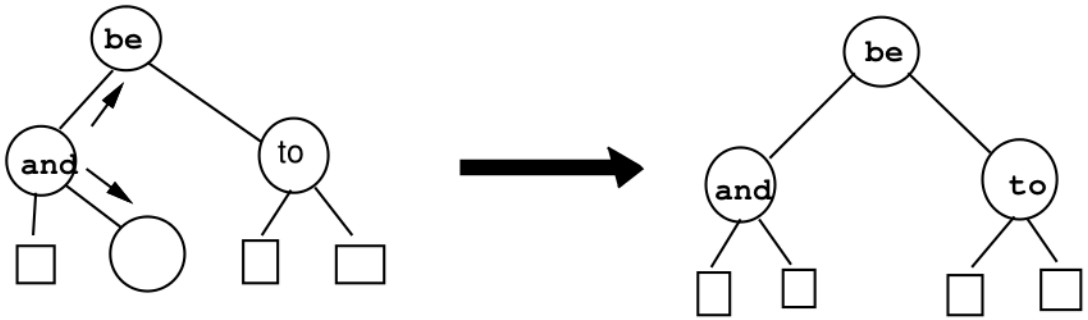
```
void Retira(TipoRegistro x, TipoApontador *p)
{ TipoApontador Aux;
  if (*p == NULL) { printf("Erro : Registro nao esta na arvore\n"); return; }
  if (x.Chave < (*p)->Reg.Chave) { Retira(x, &(*p)->Esq); return; }
  if (x.Chave > (*p)->Reg.Chave) { Retira(x, &(*p)->Dir); return; }
  if ((*p)->Dir == NULL)
  { Aux = *p;  *p = (*p)->Esq;
    free(Aux); return;
  }
  if ((*p)->Esq != NULL) { Antecessor(*p, &(*p)->Esq); return; }
  Aux = *p;  *p = (*p)->Dir;
  free(Aux);
}
```

- **Obs.:** proc. recursivo Antecessor só é ativado quando o nó que contém registro a ser retirado possui 2 descendentes. Solução usada por Wirth, 1976, p.211.

Outro Exemplo de Retirada de Nó







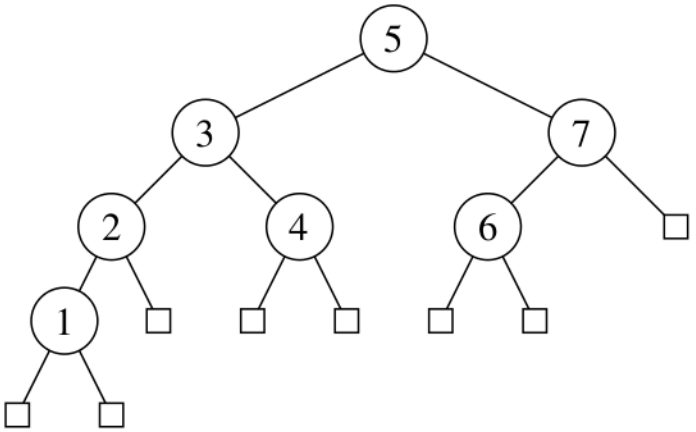
Caminhamento Central

- Após construída a árvore, pode ser necessário percorrer todos os registros que compõem a tabela ou arquivo.
- Existe mais de uma ordem de **caminhamento** em árvores, mas a mais útil é a chamada ordem de **caminhamento central**.
- O caminhamento central é mais bem expresso em termos recursivos:
  1. caminha na subárvore esquerda na ordem central;
  2. visita a raiz;
  3. caminha na subárvore direita na ordem central.
- Uma característica importante do caminhamento central é que os nós são visitados de forma ordenada.

Caminhamento Central

```
void Central(TipoApontador p)
{ if (p == NULL) return;
  Central(p->Esq);
  printf("%ld\n", p->Reg.Chave);
  Central(p->Dir);
}
```

- Percorrer a árvore usando caminhamento central recupera, na ordem: 1, 2, 3, 4, 5, 6, 7.



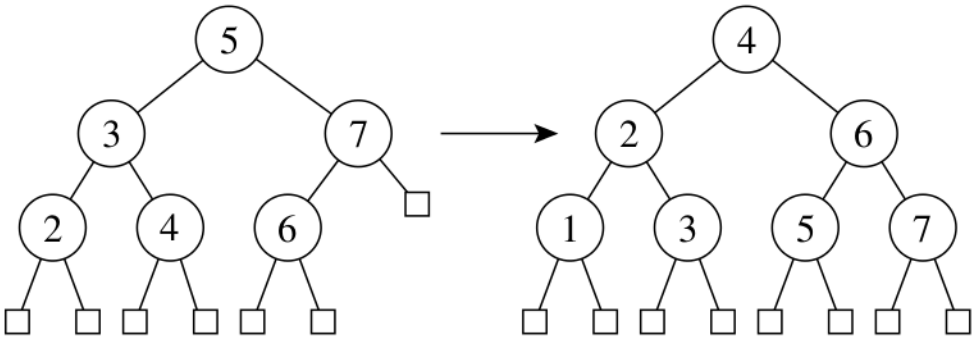
Análise

1. Para obter o pior caso basta que as chaves sejam inseridas em ordem crescente ou decrescente. Neste caso a árvore resultante é uma lista linear, cujo número médio de comparações é $(n+1)/2$.
2. Para uma **árvore de pesquisa randômica** o número esperado de comparações para recuperar um registro qualquer é cerca de $1,39 \log n$, apenas 39% pior que a árvore completamente balanceada.

- Uma árvore A com n chaves possui $n+1$ nós externos e estas n chaves dividem todos os valores possíveis em $n+1$ intervalos. Uma inserção em A é considerada *randômica* se ela tem probabilidade igual de acontecer em qualquer um dos $n+1$ intervalos.
- Uma *árvore de pesquisa randômica* com n chaves é uma árvore construida através de n inserções randômicas sucessivas em uma árvore inicialmente vazia.

Árvores Binárias de Pesquisa com Balanceamento

- Árvore completamente balanceada $\Rightarrow$ nós externos aparecem em no máximo dois níveis adjacentes.
- Minimiza tempo médio de pesquisa para uma distribuição uniforme das chaves, onde cada chave é igualmente provável de ser usada em uma pesquisa.
- Contudo, custo para manter a árvore completamente balanceada após cada inserção é muito alto.
- Para inserir a chave 1 na árvore à esquerda e obter a árvore à direita é necessário movimentar todos os nós da árvore original.



Uma Forma de Contornar este Problema

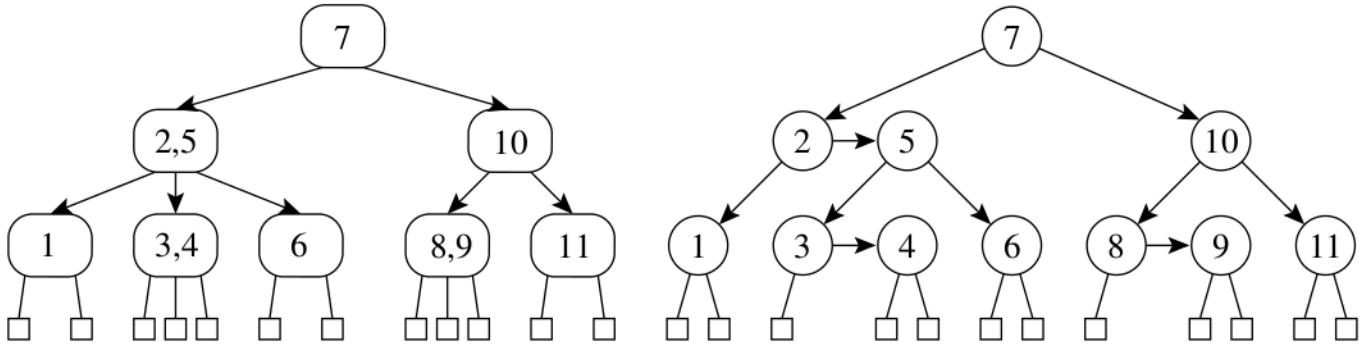
- Procurar solução intermediária que possa manter árvore "quase-balanceada", em vez de tentar manter a árvore completamente balanceada.
- **Objetivo:** Procurar obter bons tempos de pesquisa, próximos do tempo ótimo da árvore completamente balanceada, mas sem pagar muito para inserir ou retirar da árvore.
- **Heurísticas:** existem várias heurísticas baseadas no princípio acima.
- Gonnet e Baeza-Yates (1991) apresentam algoritmos que utilizam vários critérios de balanceamento para árvores de pesquisa, tais como restrições impostas:
  - na diferença das alturas de subárvores de cada nó da árvore,
  - na redução do **comprimento do caminho interno**
  - ou que todos os nós externos apareçam no mesmo nível.

Uma Forma de Contornar este Problema

- **Comprimento do caminho interno:** corresponde à soma dos comprimentos dos caminhos entre a raiz e cada um dos nós internos da árvore.
- Por exemplo, o comprimento do caminho interno da árvore à esquerda na figura do slide 31 é $8 = (0 + 1 + 1 + 2 + 2 + 2)$.

Árvores SBB

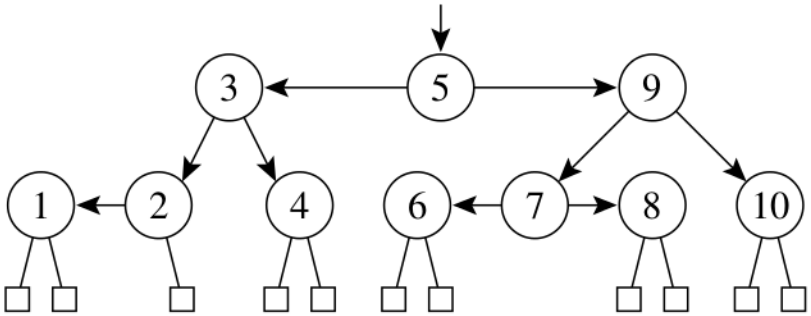
- Árvores B $\Rightarrow$ estrutura para memória secundária. (Bayer R. e McCreight E.M., 1972)
- **Árvore 2-3** $\Rightarrow$ caso especial da árvore B.
- Cada nó tem duas ou três subárvores.
- Mais apropriada para memória primária.
- **Exemplo:** Uma árvore 2-3 e a árvore B binária correspondente



Árvores SBB

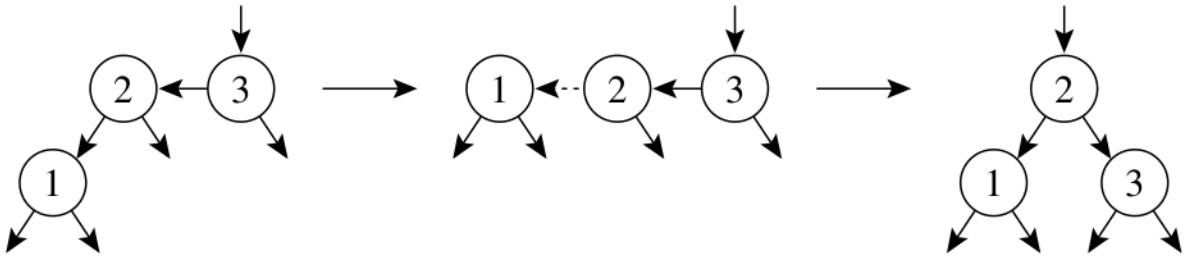
- Árvore 2-3 $\Rightarrow$ **árvore B binária** (assimetria inerente)
  1. Apontadores à esquerda apontam para um nó no nível abaixo.
  2. Apontadores à direita podem ser verticais ou horizontais.

  Eliminação da assimetria nas árvores B binárias $\Rightarrow$ árvores B binárias simétricas (*Symmetric Binary B-trees* – SBB)
- **Árvore SBB** tem apontadores verticais e horizontais, tal que:
  1. todos os caminhos da raiz até cada nó externo possuem o mesmo número de apontadores verticais, e
  2. não podem existir dois apontadores horizontais sucessivos.

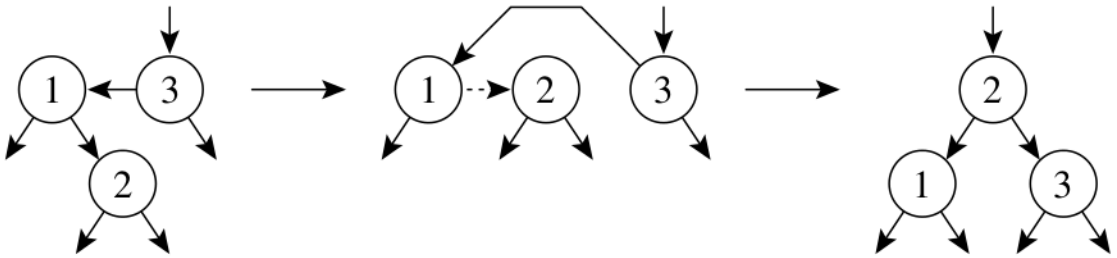


Transformações para Manutenção da Propriedade SBB

- O algoritmo para árvores SBB usa transformações locais no caminho de inserção ou retirada para preservar o balanceamento.
- A chave a ser inserida ou retirada é sempre inserida ou retirada após o apontador vertical mais baixo na árvore.
- Nesse caso podem aparecer dois apontadores horizontais sucessivos, sendo necessário realizar uma transformação:



(a) Esquerda–esquerda (EE)



(b) Esquerda–direita (ED)

Estrutura de Dados Árvore SBB para Implementar o Tipo Abstrato de Dados Dicionário

```
typedef int TipoChave;
typedef struct TipoRegistro {
   /* outros componentes */
   TipoChave Chave;
} TipoRegistro;
typedef enum {
   Vertical, Horizontal
} TipoInclinacao;
typedef struct TipoNo* TipoApontador;
typedef struct TipoNo {
   TipoRegistro Reg;
   TipoApontador Esq, Dir;
   TipoInclinacao BitE, BitD;
} TipoNo;
```

Procedimentos Auxiliares para Árvores SBB

```
void EE(TipoApontador *Ap)
{ TipoApontador Ap1;
  Ap1 = (*Ap)->Esq; (*Ap)->Esq = Ap1->Dir; Ap1->Dir = *Ap;
  Ap1->BitE = Vertical; (*Ap)->BitE = Vertical; *Ap = Ap1;
}

void ED(TipoApontador *Ap)
{ TipoApontador Ap1, Ap2;
  Ap1 = (*Ap)->Esq; Ap2 = Ap1->Dir; Ap1->BitD = Vertical;
  (*Ap)->BitE = Vertical; Ap1->Dir = Ap2->Esq; Ap2->Esq = Ap1;
  (*Ap)->Esq = Ap2->Dir; Ap2->Dir = *Ap; *Ap = Ap2;
}
```

Procedimentos Auxiliares para Árvores SBB

```
void DD(TipoApontador *Ap)
{ TipoApontador Ap1;
  Ap1 = (*Ap)->Dir; (*Ap)->Dir = Ap1->Esq; Ap1->Esq = *Ap;
  Ap1->BitD = Vertical; (*Ap)->BitD = Vertical; *Ap = Ap1;
}

void DE(TipoApontador *Ap)
{ TipoApontador Ap1, Ap2;
  Ap1 = (*Ap)->Dir; Ap2 = Ap1->Esq; Ap1->BitE = Vertical;
  (*Ap)->BitD = Vertical; Ap1->Esq = Ap2->Dir; Ap2->Dir = Ap1;
  (*Ap)->Dir = Ap2->Esq; Ap2->Esq = *Ap; *Ap = Ap2;
}
```

Procedimento para Inserir na Árvore SBB

```
void IInsere(TipoRegistro x, TipoApontador *Ap,
             TipoInclinacao *IAp, short *Fim)
{ if (*Ap == NULL)
  { *Ap = (TipoApontador)malloc(sizeof(TipoNo));
    *IAp = Horizontal;  (*Ap)->Reg = x;
    (*Ap)->BitE = Vertical;  (*Ap)->BitD = Vertical;
    (*Ap)->Esq = NULL; (*Ap)->Dir = NULL; *Fim = FALSE;
    return;
  }
  if (x.Chave < (*Ap)->Reg.Chave)
  { IInsere(x, &(*Ap)->Esq, &(*Ap)->BitE, Fim);
    if (*Fim) return;
    if ((*Ap)->BitE != Horizontal) { *Fim = TRUE; return; }
    if ((*Ap)->Esq->BitE == Horizontal)
    { EE(Ap); *IAp = Horizontal; return; }
    if ((*Ap)->Esq->BitD == Horizontal) { ED(Ap); *IAp = Horizontal; }
    return;
  }
```

Procedimento para Inserir na Árvore SBB

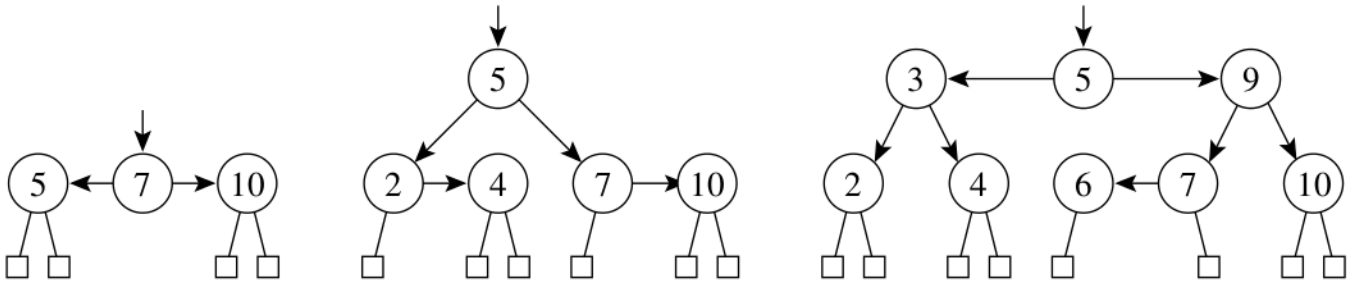
```
  if (x.Chave <= (*Ap)->Reg.Chave)
  { printf("Erro: Chave ja esta na arvore\n");
    *Fim = TRUE;
    return;
  }
  IInsere(x, &(*Ap)->Dir, &(*Ap)->BitD, Fim);
  if (*Fim) return;
  if ((*Ap)->BitD != Horizontal) { *Fim = TRUE;  return; }
  if ((*Ap)->Dir->BitD == Horizontal)
  { DD(Ap); *IAp = Horizontal; return;}
  if ((*Ap)->Dir->BitE == Horizontal) { DE(Ap); *IAp = Horizontal; }
}

void Insere(TipoRegistro x, TipoApontador *Ap)
{ short Fim;  TipoInclinacao IAp;
  IInsere(x, Ap, &IAp, &Fim);
}
```

Exemplo

Inserção de uma sequência de chaves em uma árvore SBB:

1. Árvore à esquerda é obtida após a inserção das chaves 7, 10, 5.
2. Árvore do meio é obtida após a inserção das chaves 2, 4 na árvore anterior.
3. Árvore à direita é obtida após a inserção das chaves 9, 3, 6 na árvore anterior.




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

Procedimento Retira

- Retira contém um outro procedimento interno de nome IRetira.
- IRetira usa 3 procedimentos internos:EsqCurto, DirCurto, Antecessor.
  - EsqCurto (DirCurto) é chamado quando um nó folha que é referenciado por um apontador vertical é retirado da subárvore à esquerda (direita) tornando-a menor na altura após a retirada;
  - Quando o nó a ser retirado possui dois descendentes, o procedimento Antecessor localiza o nó antecessor para ser trocado com o nó a ser retirado.

Procedimento para Retirar da Árvore SBB

```
void EsqCurto(TipoApontador *Ap, short *Fim)
{ /* Folha esquerda retirada => arvore curta na altura esquerda */
  TipoApontador Ap1;
  if ((*Ap)->BitE == Horizontal)
  { (*Ap)->BitE = Vertical; *Fim = TRUE; return; }
  if ((*Ap)->BitD == Horizontal)
  { Ap1 = (*Ap)->Dir; (*Ap)->Dir = Ap1->Esq; Ap1->Esq = *Ap; *Ap = Ap1;
    if ((*Ap)->Esq->Dir->BitE == Horizontal)
    { DE(&(*Ap)->Esq); (*Ap)->BitE = Horizontal;}
    else if ((*Ap)->Esq->Dir->BitD == Horizontal)
          { DD(&(*Ap)->Esq); (*Ap)->BitE = Horizontal; }
    *Fim = TRUE;  return;
  }
  (*Ap)->BitD = Horizontal;
  if ((*Ap)->Dir->BitE == Horizontal) { DE(Ap); *Fim = TRUE; return; }
  if ((*Ap)->Dir->BitD == Horizontal) { DD(Ap); *Fim = TRUE; }
}
```

Procedimento para Retirar da Árvore SBB – DirCurto

```
void DirCurto(TipoApontador *Ap, short *Fim)
{ /* Folha direita retirada => arvore curta na altura direita */
  TipoApontador Ap1;
  if ((*Ap)->BitD == Horizontal)
  { (*Ap)->BitD = Vertical; *Fim = TRUE; return; }
  if ((*Ap)->BitE == Horizontal)
  { Ap1 = (*Ap)->Esq; (*Ap)->Esq = Ap1->Dir; Ap1->Dir = *Ap; *Ap = Ap1;
    if ((*Ap)->Dir->Esq->BitD == Horizontal)
    { ED(&(*Ap)->Dir); (*Ap)->BitD = Horizontal; }
    else if ((*Ap)->Dir->Esq->BitE == Horizontal)
          { EE(&(*Ap)->Dir); (*Ap)->BitD = Horizontal;}
    *Fim = TRUE;  return;
  }
  (*Ap)->BitE = Horizontal;
  if ((*Ap)->Esq->BitD == Horizontal) { ED(Ap); *Fim = TRUE; return; }
  if ((*Ap)->Esq->BitE == Horizontal) { EE(Ap); *Fim = TRUE; }
}
```

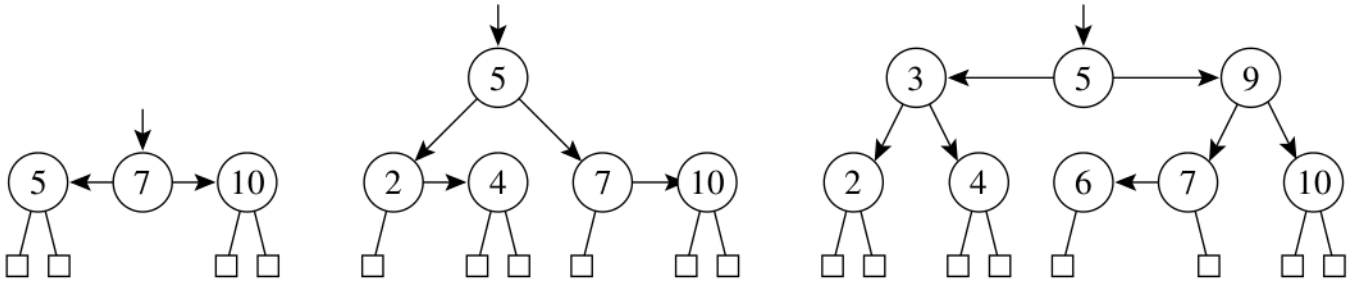
Procedimento para Retirar da Árvore SBB – Antecessor

```
void Antecessor(TipoApontador q, TipoApontador *r, short *Fim)
{ if ((*r)->Dir != NULL)
  { Antecessor(q, &(*r)->Dir, Fim);
    if (!*Fim) DirCurto(r, Fim);  return;
  }
  q->Reg = (*r)->Reg; q = *r; *r = (*r)->Esq; free(q);
  if (*r != NULL) *Fim = TRUE;
}
```

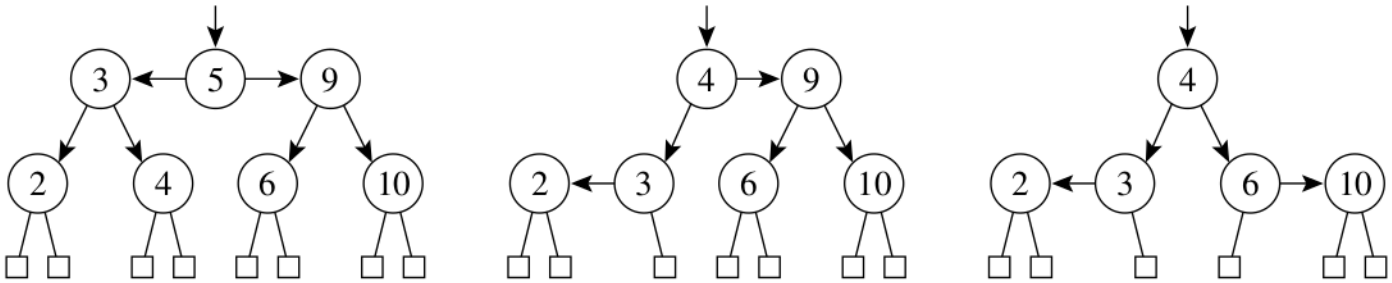
Procedimento para Retirar da Árvore SBB

```
void IRetira(TipoRegistro x, TipoApontador *Ap, short *Fim)
{ TipoNo *Aux;
  if (*Ap == NULL) { printf("Chave nao esta na arvore\n"); *Fim = TRUE; return; }
  if (x.Chave < (*Ap)->Reg.Chave)
  { IRetira(x, &(*Ap)->Esq, Fim); if (!*Fim) EsqCurto(Ap, Fim);  return; }
  if (x.Chave > (*Ap)->Reg.Chave)
  { IRetira(x, &(*Ap)->Dir, Fim);
    if (!*Fim) DirCurto(Ap, Fim);  return;
  }
  *Fim = FALSE; Aux = *Ap;
  if (Aux->Dir == NULL)
  { *Ap = Aux->Esq;  free(Aux);
    if (*Ap != NULL) *Fim = TRUE;  return;
  }
  if (Aux->Esq == NULL)
  { *Ap = Aux->Dir; free(Aux);
    if (*Ap != NULL) *Fim = TRUE;  return;
  }
  Antecessor(Aux, &Aux->Esq, Fim);
  if (!*Fim) EsqCurto(Ap, Fim); /* Encontrou chave */
}
void Retira(TipoRegistro x, TipoApontador *Ap)
{ short Fim; IRetira(x, Ap, &Fim); }
```

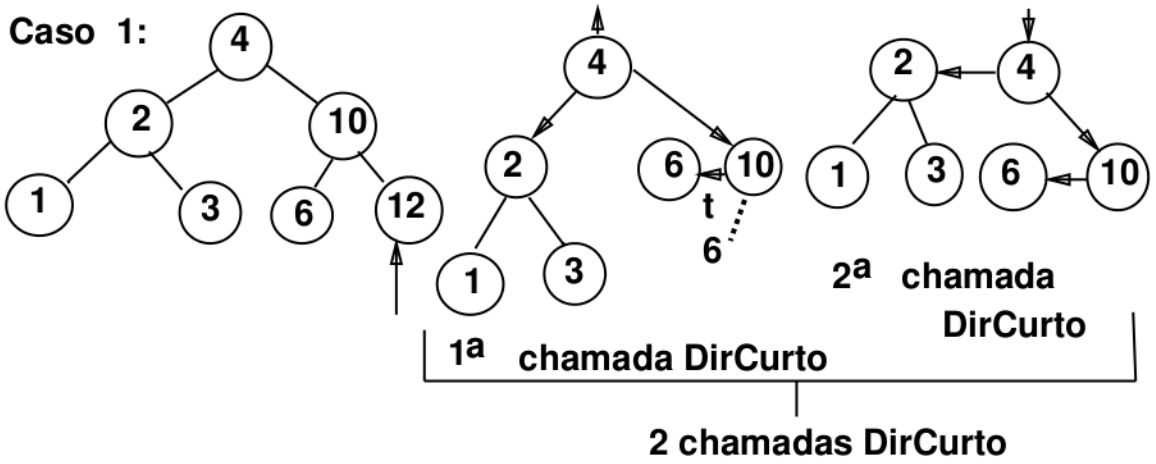
Exemplo

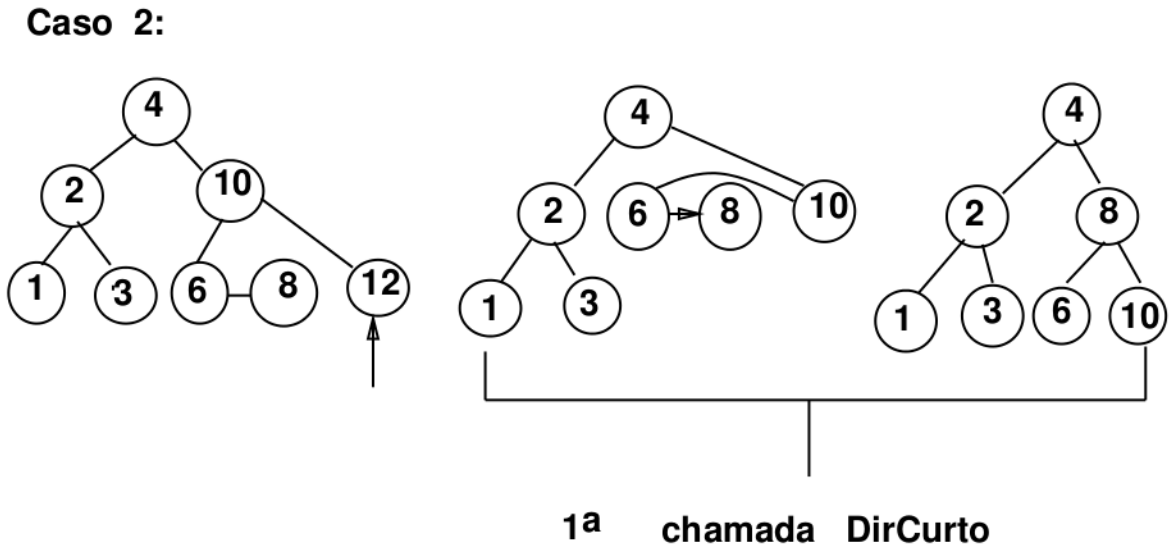


- A árvore à esquerda abaixo é obtida após a retirada da chave 7 da árvore à direita acima.
- A árvore do meio é obtida após a retirada da chave 5 da árvore anterior.
- A árvore à direita é obtida após a retirada da chave 9 da árvore anterior.



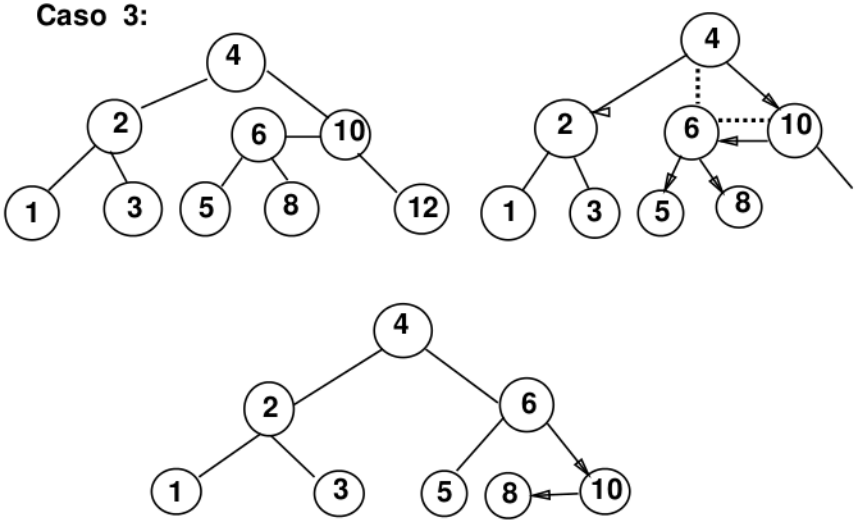
Exemplo: Retirada de Nós da Árvore SBB



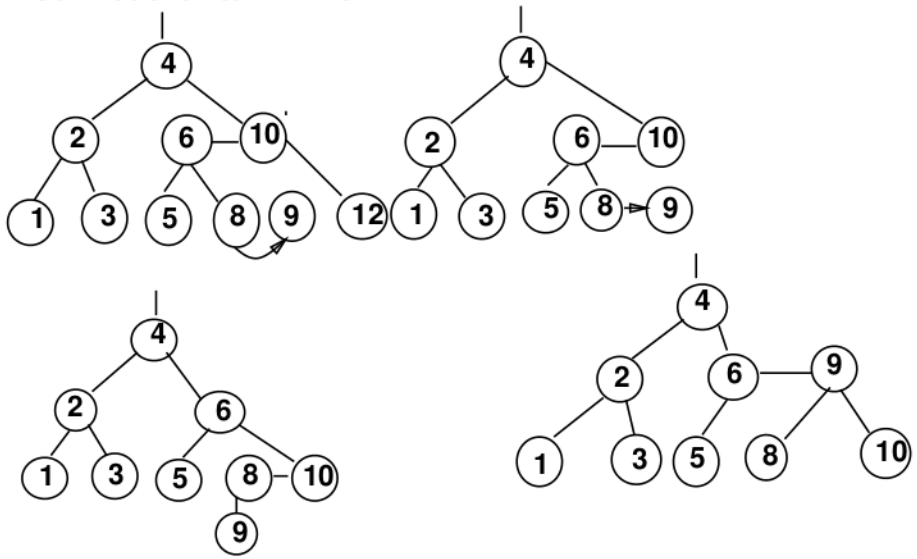


Exemplo: Retirada de Nós da Árvore SBB

Caso 3:



Se nodo 8 tem filho:



Análise

- Nas árvores SBB é necessário distinguir dois tipos de **alturas**:
  1. **Altura vertical** $h \rightarrow$ necessária para manter a altura uniforme e obtida através da contagem do número de apontadores verticais em qualquer caminho entre a raiz e um nó externo.
  2. **Altura** $k \rightarrow$ representa o número máximo de comparações de chaves obtida através da contagem do número total de apontadores no maior caminho entre a raiz e um nó externo.
- A altura k é maior que a altura h sempre que existirem apontadores horizontais na árvore.
- Para uma árvore SBB com n nós internos, temos que

$$h \leq k \leq 2h.$$