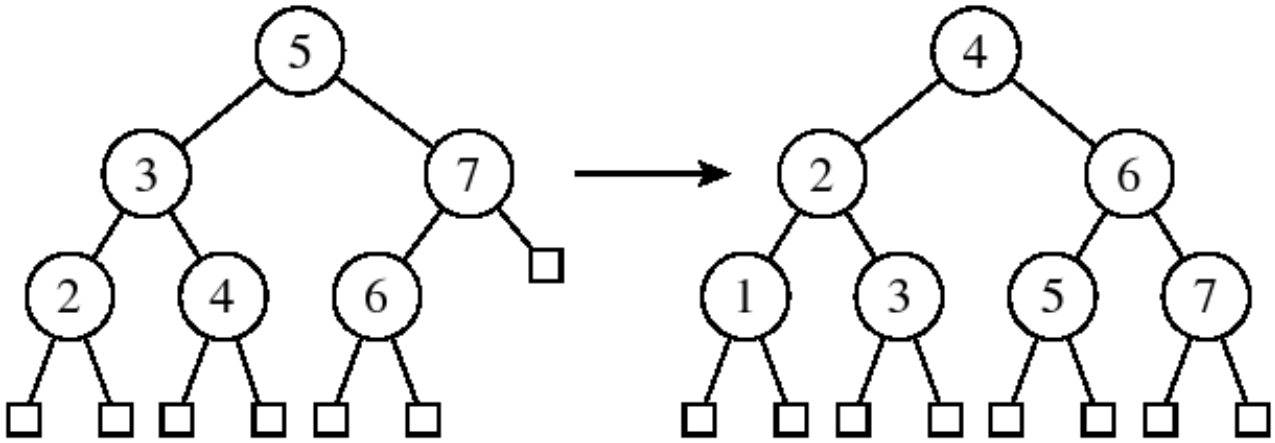
Árvores Binárias de Pesquisa Com Balanceamento

- Árvore completamente balanceada ⇒ 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.

Árvores Binárias de Pesquisa Com Balanceamento

- Para inserir a chave 1 na árvore do exemplo à esquerda e obter a árvore à direita do mesmo exemplo 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.

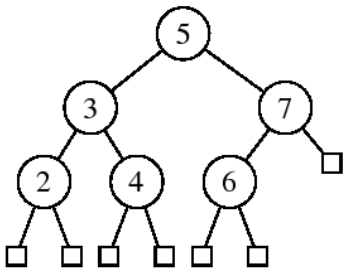
- Objetivo: Procurar obter bons tempos de pesquisa, próximos do tempo ótimo da árvore completamente balanceada, mas sem pagar muito para inserir ou retirar da árvore.

- Heurísticas: existem várias heurísticas baseadas no princípio acima.

Uma Forma de Contornar este Problema

- Gonnet e Baeza-Yates (1991) apresentam algoritmos que utilizam vários critérios de balanceamento para árvores de pesquisa, tais como restrições impostas:
  - todos os nós externos apareçam no mesmo nível.
  - na diferença das alturas de subárvores de cada nó
  - na redução do comprimento do caminho interno

    corresponde à soma dos comprimentos dos caminhos entre a raiz e cada um dos nós internos da árvore.



8 = 0 + 1 + 1 + 2 + 2 + 2

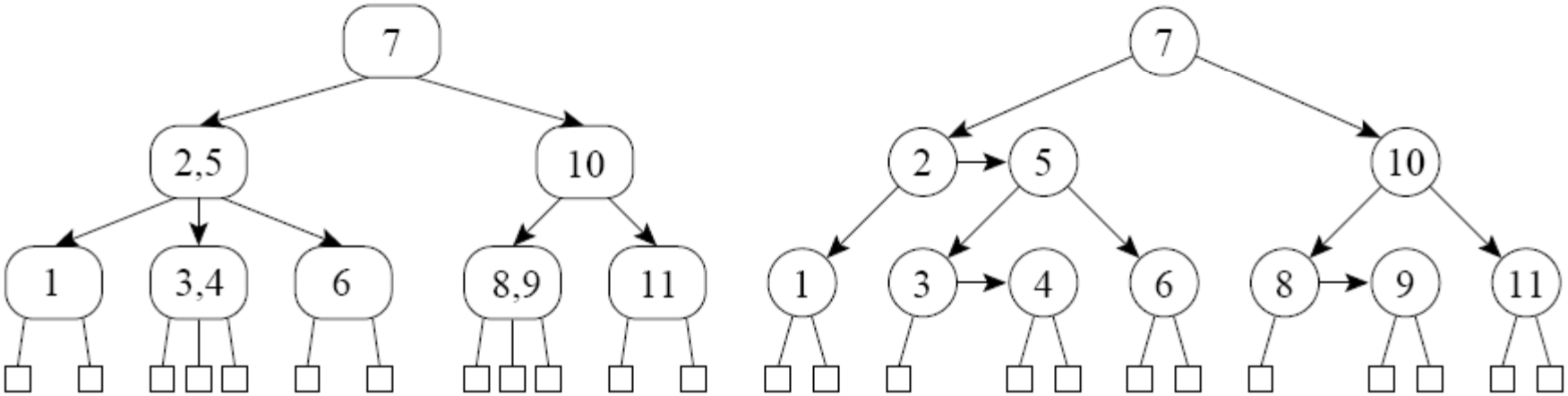
Árvores SBB

- Árvores B : estrutura para memória secundária (Bayer R. e McCreight E. M., 1972)

- Árvore 2-3 : caso especial de árvore B mais apropriada para memória primária
  - Cada nó tem duas ou três subárvores
  - Pode ser representada como uma árvore binária

Árvores SBB

- Exemplo: Uma árvore 2-3 e a árvore B binária correspondente (Bayer, R. 1971)



Árvores SBB

- Árvore 2-3: 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

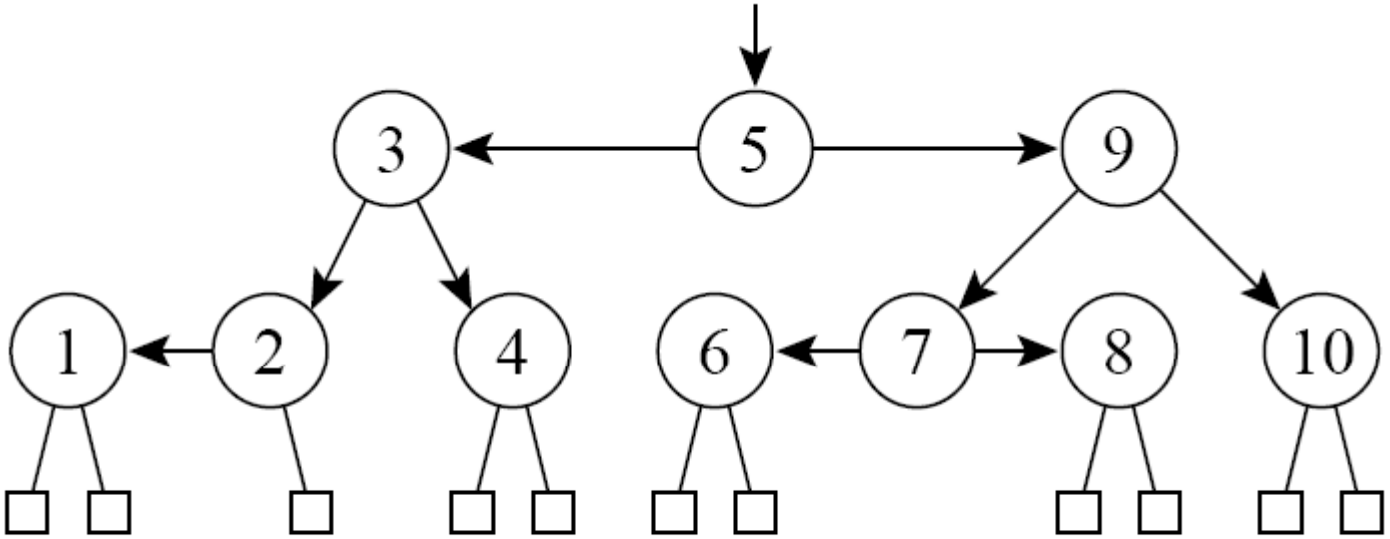
- Eliminação da assimetria nas árvores B binárias: árvores B binárias simétricas (*Symmetric Binary B-trees – SBB*)

- Árvore SBB é uma árvore binária com 2 tipos de 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.

Árvores SBB



A pesquisa em uma árvore SBB é idêntica a uma árvore sem balanceamento. A pesquisa ignora se os apontadores são horizontais ou verticais

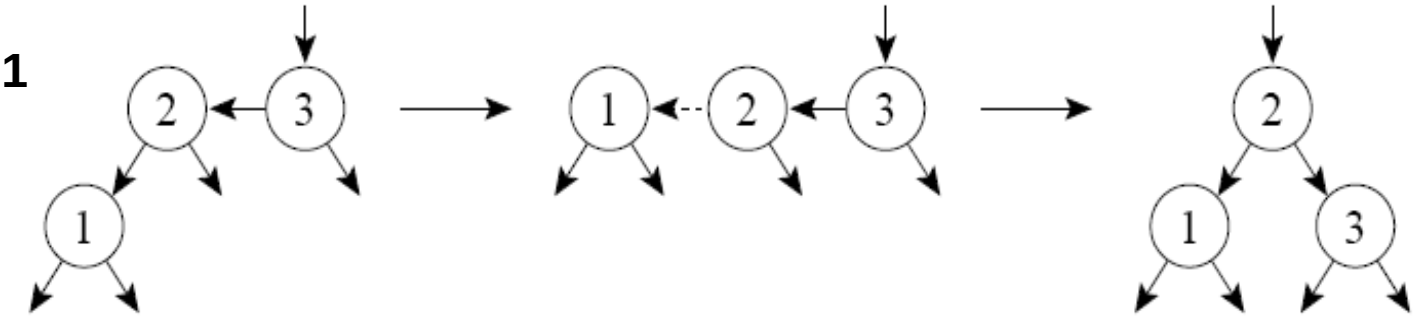
Transformações para Manutenção da Propriedade SBB

- 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
- Dependendo da situação anterior à inserção ou retirada, podem, aparecer dois apontadores horizontais sucessivos
- Neste caso: é necessário realizar uma transformação

Transformações para Manutenção da Propriedade SBB

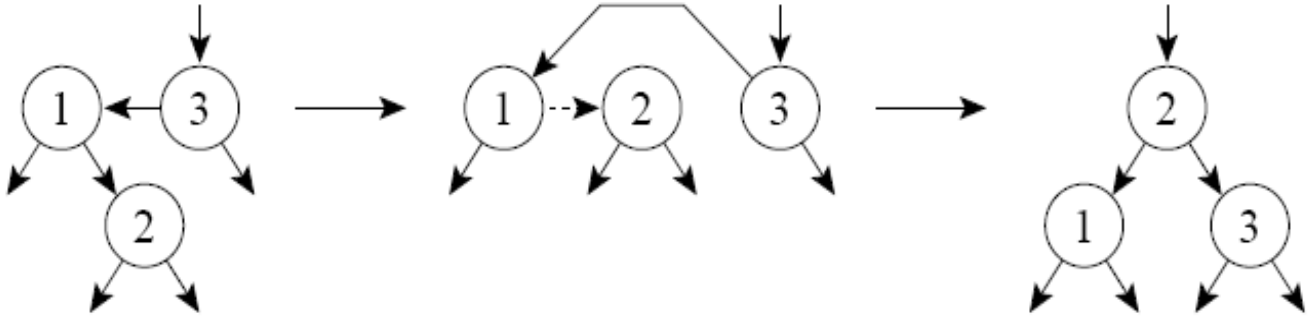
- Transformações propostas por Bayer R. 1972

Inserção do 1



(a) Esquerda−esquerda (EE)

Inserção do 2



(b) Esquerda−direita (ED)

Exemplo

- Inserção das Chaves 7, 10, 5, 2, 4, 9, 3, 6

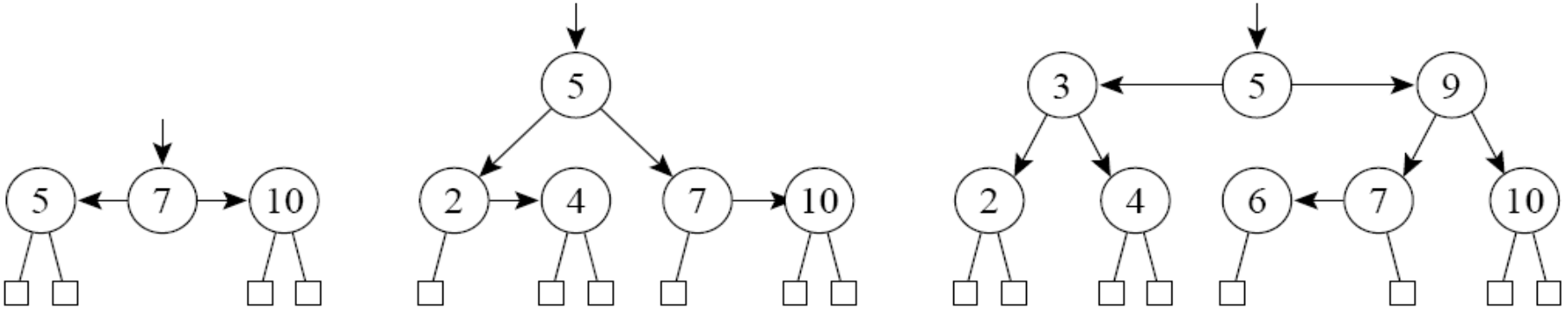
Exemplo

- Inserção de uma sequência de chaves em uma árvore SBB inicialmente vazia.

  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.



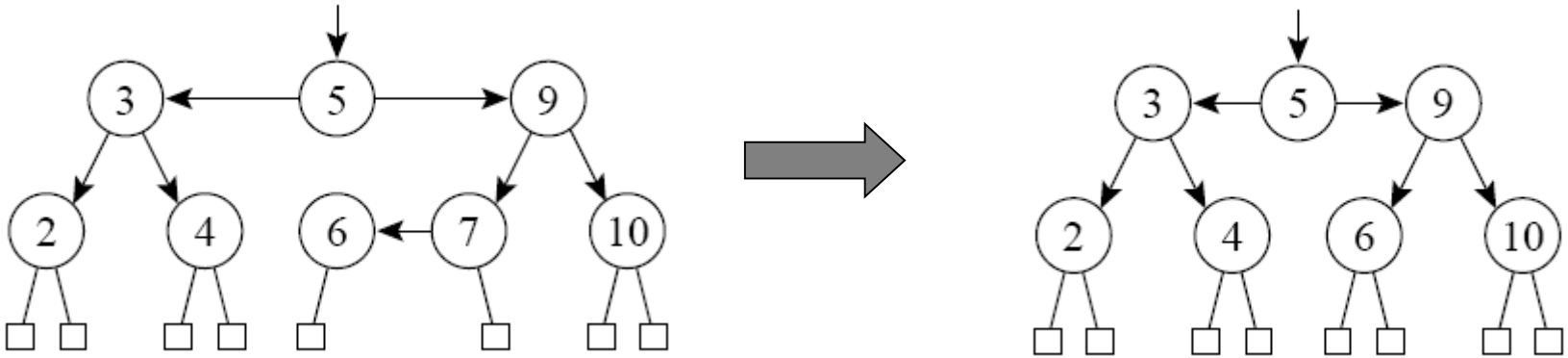
Retirada

- Mais complexa...
- Procura sempre fazer modificações locais
- EsqCurto (DirCurto): chamado quando um nó folha é retirado da sub-árvore da esquerda (direita) tornando menor a altura
- Antecessor: chamado quando o nó possui dois descendentes
- Em alguns casos, o procedimento não é muito intuitivo

Exemplo

- Caso “simples” 1

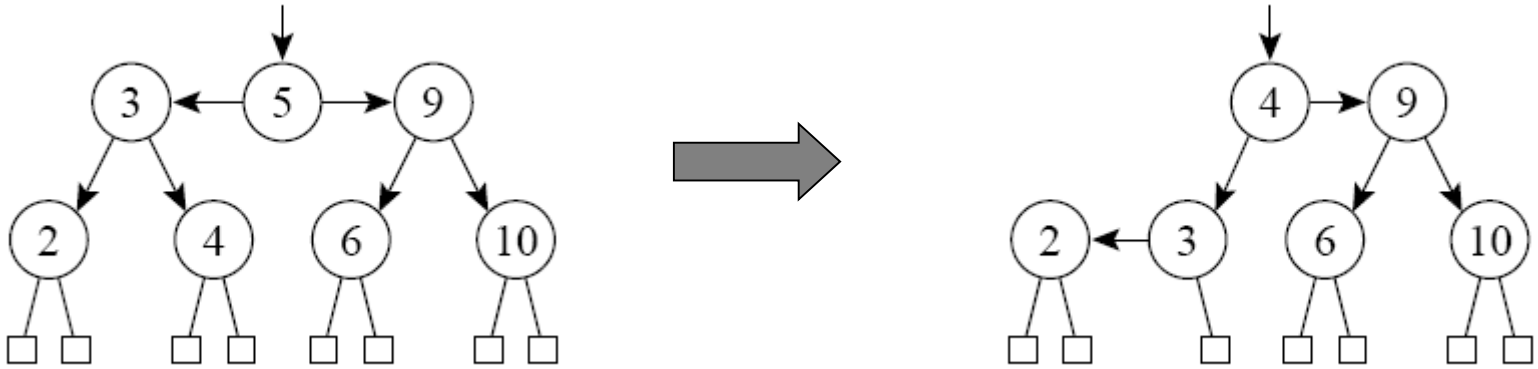
  Retirada da chave 7



Exemplo

- Caso “simples” 2

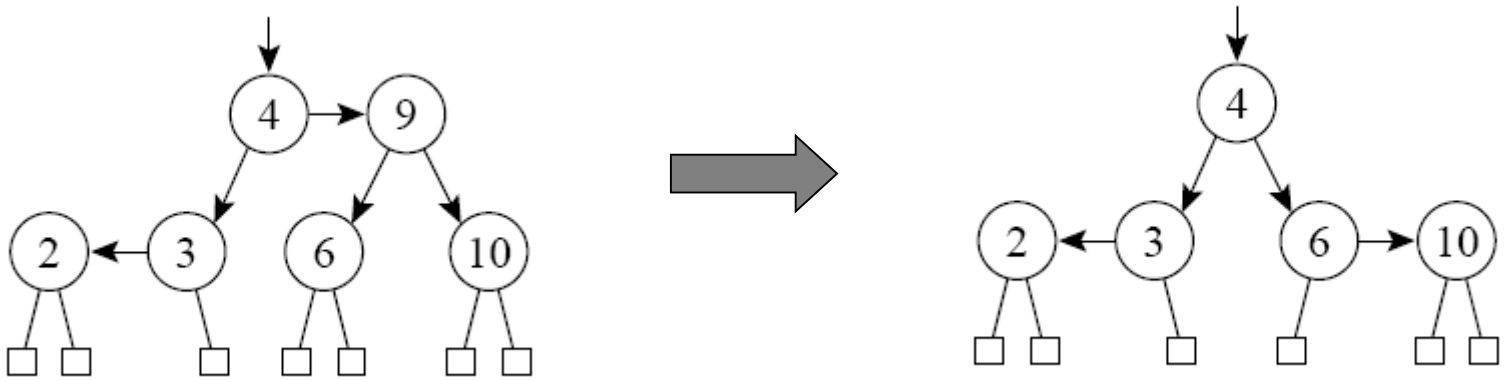
  Retirada da chave 5



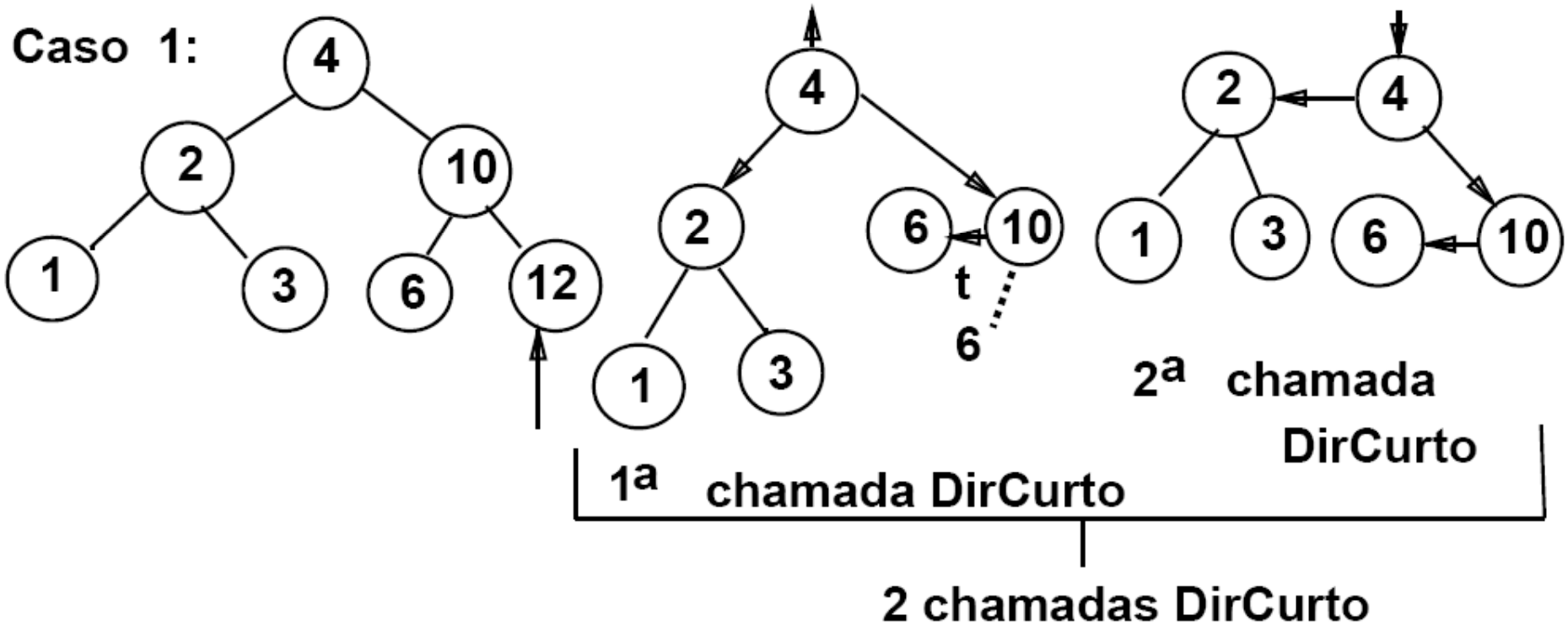
Exemplo

- Caso “simples 3”

  Retirada da chave 9

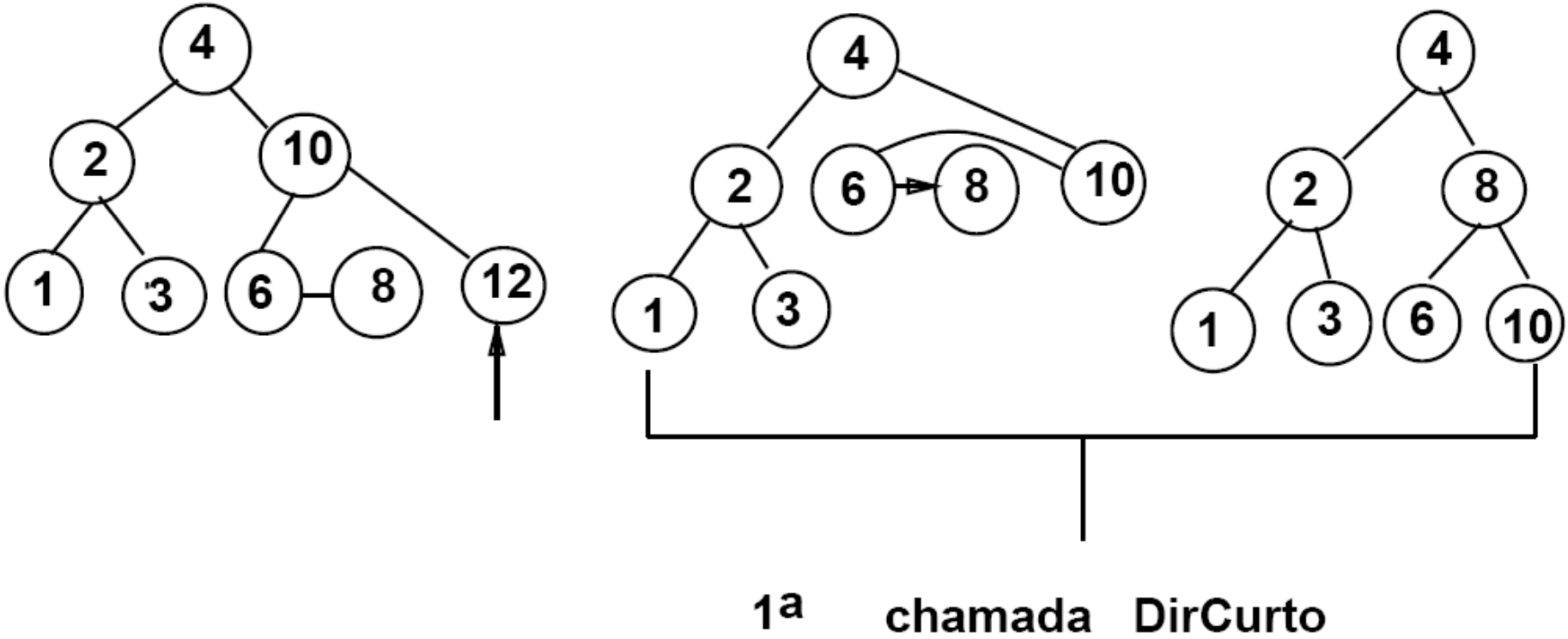


Casos complexos
Retirada do 12



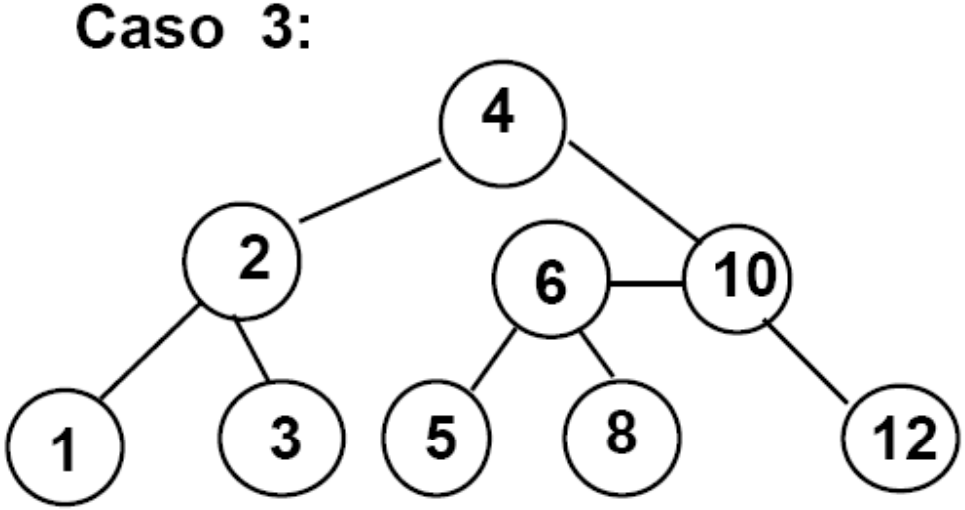
Casos complexos
Retirada do 12

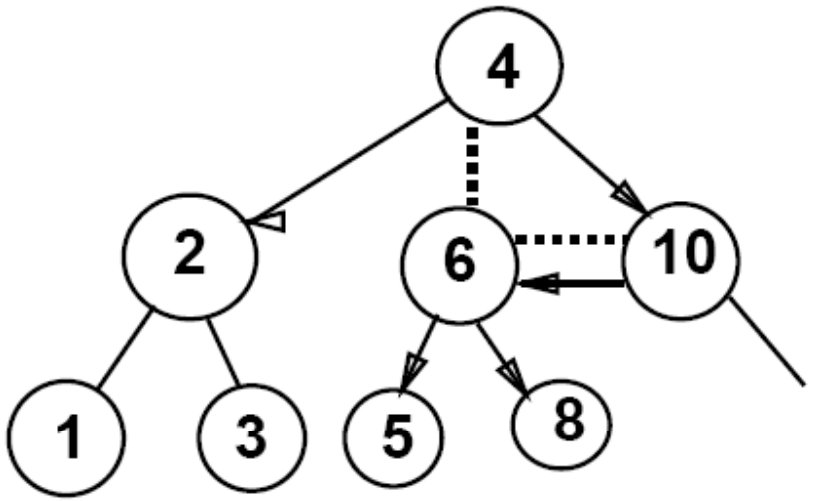
Caso 2:

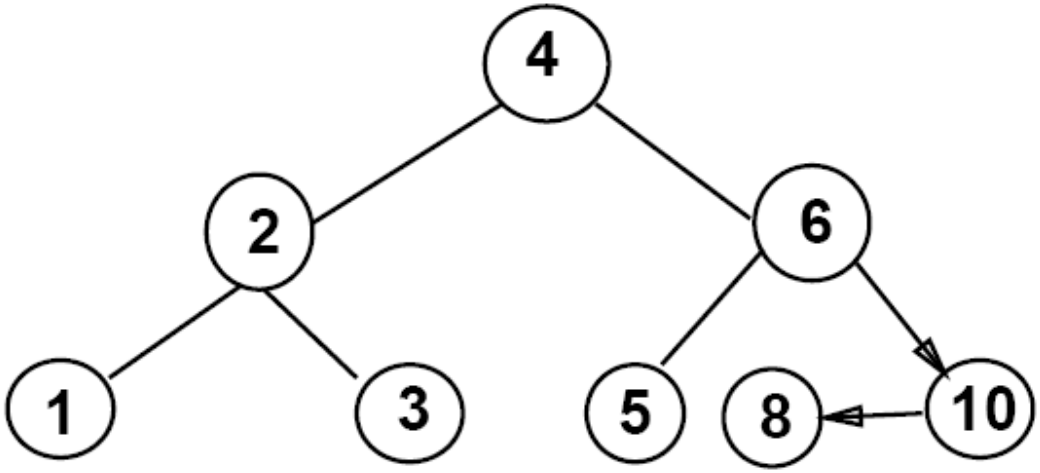


Casos complexos
Retirada do 12

Caso 3:

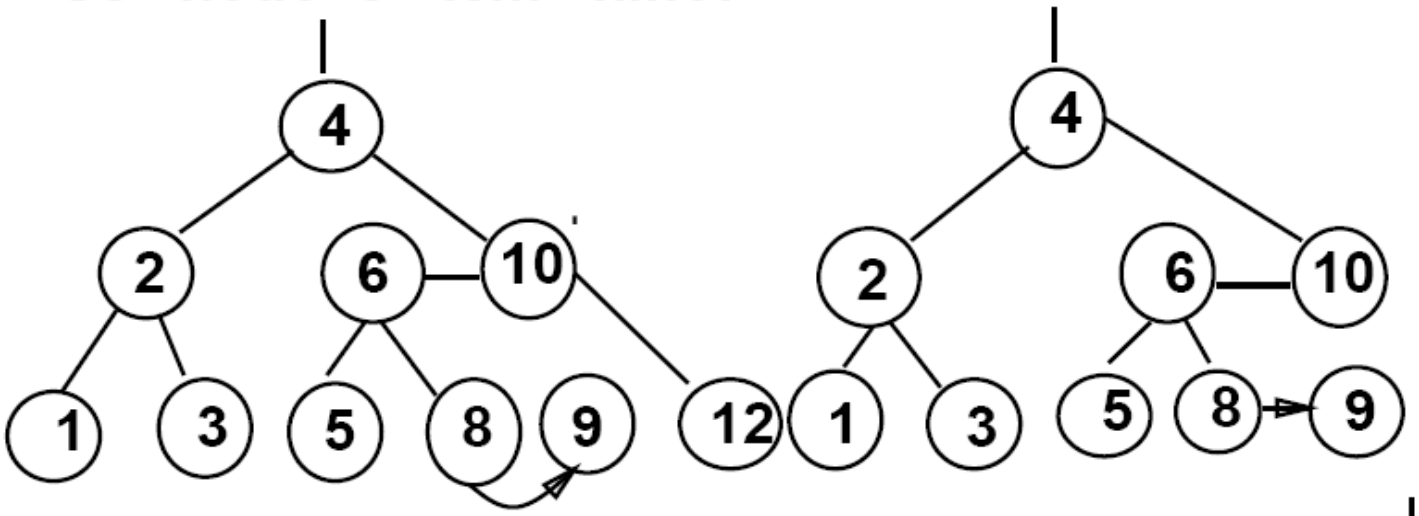


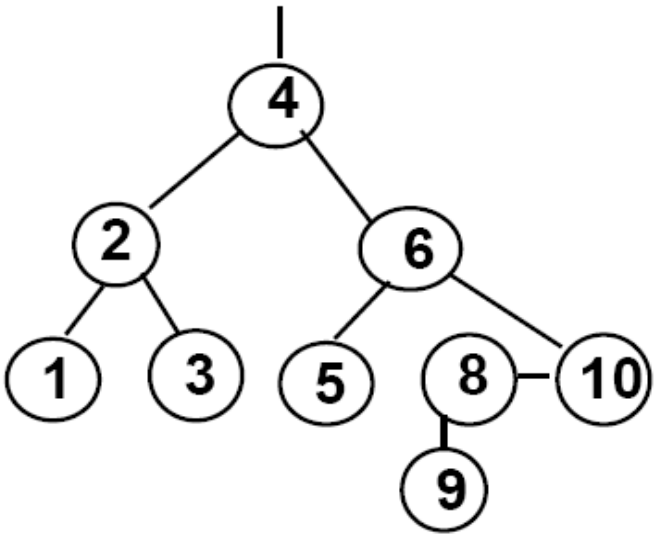


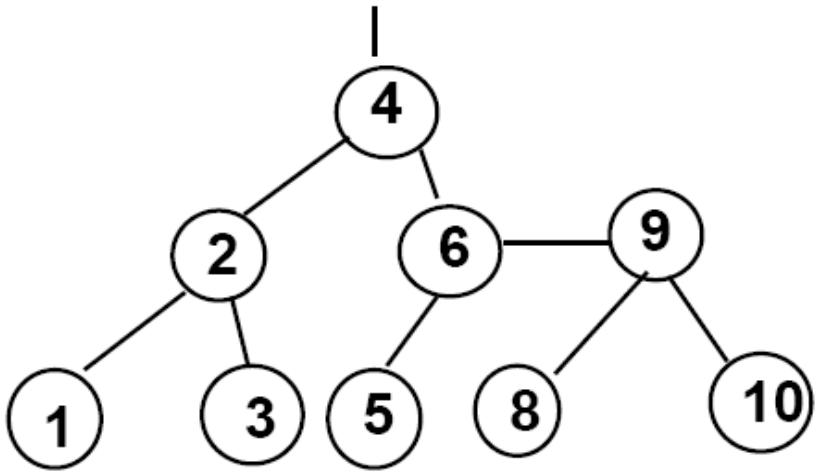


Casos complexos
Retirada do 12

Se nodo 8 tem filho:







Análise

- Nas árvores SBB é necessário distinguir dois tipos de **alturas**

  1- altura vertical *h* : 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* : representa o número máximo de comparações de chaves obtido 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:

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

Análise

- De fato, Bayer (1972) mostrou que:

$$\log(n+1) \leq k \leq 2\log(n+2) - 2$$

- Custo para manter a propriedade SBB: custo para percorrer o caminho de pesquisa para encontrar a chave, seja para inserí-la ou para retirá-la
  - Logo: o custo e *O(log n)*
- Número de comparações em uma pesquisa com sucesso na árvore SBB:
  - Melhor caso: C(n) = O(1)
  - Pior caso: C(n) = O(log n)
  - Caso médio: C(n) = O(log n)
- **Observe:** Na prática, o caso médio para C(n) é apenas cerca de 2% pior que o C(n) para uma árvore completamente balanceada, conforme mostrado em Ziviani e Tompa (1982)

Estrutura de Dados Árvore SBB para Implementar o TAD Dicionário

```
#include <sys/time.h>
#include<stdlib.h>
#include<stdio.h>

#define TRUE 1
#define FALSE 0
#define max 10

typedef int TipoChave;

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

Estrutura de Dados Árvore SBB para Implementar o TAD Dicionário

```
typedef enum {
                    Vertical, Horizontal
              } Inclinacao;

typedef struct No* Apontador;

typedef struct No {
                    Registro Reg;
                    Apontador Esq, Dir;
                    Inclinacao BitE, BitD;
            } No;
```

Procedimentos Auxiliares para Árvores SBB

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

void ED(Apontador *Ap)
{
   Apontador 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(Apontador *Ap)
{
   Apontador Ap1;
   Ap1 = (*Ap)->Dir;          (*Ap)->Dir = Ap1->Esq;    Ap1->Esq = *Ap;
    Ap1->BitD = Vertical;  (*Ap)->BitD = Vertical;
   *Ap = Ap1;
 }

void DE(Apontador *Ap)
{
    Apontador 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;
}
```

Procedimentos para Inserir na Árvores SBB

```
void Insere(Registro x, Apontador *Ap)
 {
      short Fim;
      Inclinacao Iap;
      IInsere(x, Ap, &IAp, &Fim);
 }
```

Procedimentos para Inserir na Árvores SBB

```
void IInsere(Registro x, Apontador *Ap, Inclinacao *IAp, short *Fim)
{
        if (*Ap == NULL) {
                *Ap = (Apontador)malloc(sizeof(No));
                *IAp = Horizontal;
                (*Ap)->Reg = x;
                (*Ap)->BitE = Vertical;
                (*Ap)->BitD = Vertical;
                (*Ap)->Esq = NULL;
                (*Ap)->Dir = NULL;
               *Fim = FALSE;
                return;
         }
```

Procedimentos para Inserir na Árvores SBB

```
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;
}
```

Procedimentos para Inserir na Árvores 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;
 }
}
```

Procedimento de Inicialização da Árvore SBB

```
void Inicializa(Apontador *Dicionario)
{

        *Dicionario = NULL;
}
```

Procedimento de Retirada da Árvore SBB

- Retira contém um outro procedimento interno de nome IRetira.
- IRetira usa 3 procedimentos internos:
  - EsqCurto, DirCurto e Antecessor
- EsqCurto (DirCurto):
  - 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;
- Antecessor:
  - 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 de Retirada da Árvore SBB

```
void Retira(Registro x, Apontador *Ap)
{
     short Fim;
     IRetira(x, Ap, &Fim);
 }
```

Procedimento de Retirada da Árvore SBB

```
void IRetira(Registro x, Apontador *Ap, short *Fim)
{
      No *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;
        }
```

Procedimento de Retirada da Árvore SBB

```
        *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 */
}
```

Procedimento de Retirada da Árvore SBB

```
void DirCurto(Apontador *Ap, short *Fim)
 {     /* Folha direita retirada => arvore curta na altura direita */
    Apontador 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;
      }
```

Procedimento de Retirada da Árvore SBB

```
    (*Ap)->BitE = Horizontal;
     if ((*Ap)->Esq->BitD == Horizontal) {
                ED(Ap);       *Fim = TRUE;
                return;
     }
     if ((*Ap)->Esq->BitE == Horizontal) {
                 EE(Ap); *Fim = TRUE;
     }
}
```

Procedimento de Retirada da Árvore SBB

```
void EsqCurto(Apontador *Ap, short *Fim)
{    /* Folha esquerda retirada => arvore curta na altura esquerda */
     Apontador 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;
       }
```

Procedimento de Retirada da Árvore SBB

```
    (*Ap)->BitD = Horizontal;
     if ((*Ap)->Dir->BitE == Horizontal) {
           DE(Ap);                  *Fim = TRUE;
           return;
     }
      if ((*Ap)->Dir->BitD == Horizontal) {
           DD(Ap);                  *Fim = TRUE;
      }
}
```

Procedimento de Retirada da Árvore SBB

```
void Antecessor(Apontador q, Apontador *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;
}
```