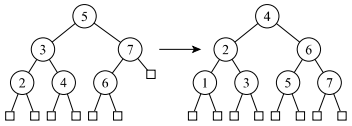
Árvores Binárias de Pesquisa com Balanceamento

- 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.
- 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

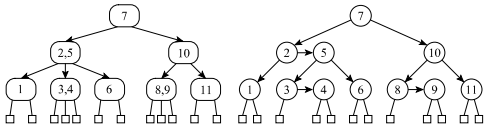
- Procurar solução intermediária que possa manter árvore "quase-balanceada", em vez de tentar manter a árvore completamente balanceada.
- **Objetivo:** Procurar obter bons tempos de pesquisa, próximos do tempo ótimo da árvore completamente balanceada, mas sem pagar muito para inserir ou retirar da árvore.
- **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

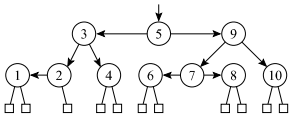
- Árvores B ⇒ estrutura para memória secundária. (Bayer R. e McCreight E.M., 1972)
- **Árvore 2-3** ⇒ caso especial da árvore B.
- Cada nó tem duas ou três subárvores.
- Mais apropriada para memória primária.
- **Exemplo:** Uma árvore 2-3 e a árvore B binária correspondente



Árvores SBB

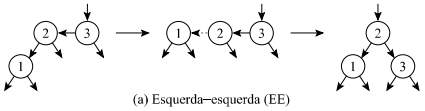
- Árvore 2-3 ⇒ **árvore B binária** (assimetria inerente)
  1. Apontadores à esquerda apontam para um nó no nível abaixo.
  2. Apontadores à direita podem ser verticais ou horizontais.

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

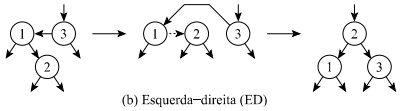


Transformações para Manutenção da Propriedade SBB

- 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

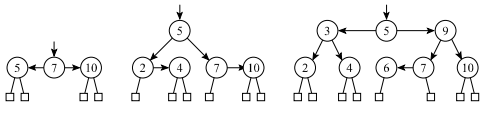
```
  if (x.Chave <= (*Ap)->Reg.Chave)
  { printf("Erro: Chave ja esta na arvore\n");
    *Fim = TRUE;
    return;
  }
  IInsere(x, &(*Ap)->Dir, &(*Ap)->BitD, Fim);
  if (*Fim) return;
  if ((*Ap)->BitD != Horizontal) { *Fim = TRUE;  return; }
  if ((*Ap)->Dir->BitD == Horizontal)
  { DD(Ap); *IAp =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

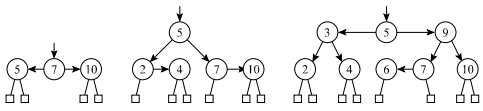

Procedimento para Retirar da Árvore SBB – Antecessor

```
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;
}
```

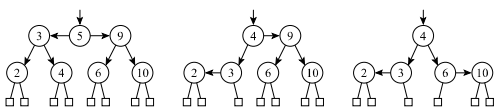

Procedimento para Retirar da Árvore SBB

```
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); }
```

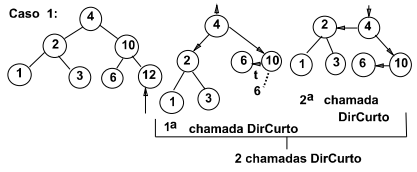

Exemplo

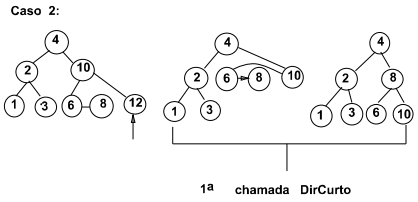


- A árvore à esquerda abaixo é obtida após a retirada da chave 7 da árvore à direita acima.
- A árvore do meio é obtida após a retirada da chave 5 da árvore anterior.
- A árvore à direita é obtida após a retirada da chave 9 da árvore anterior.

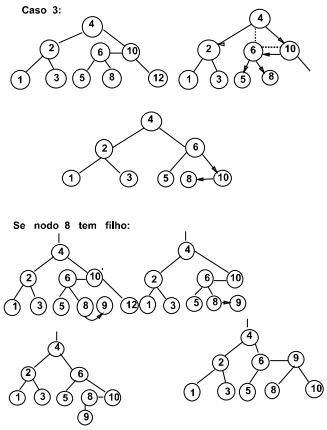


Exemplo: Retirada de Nós da Árvore SBB





Exemplo: Retirada de Nós da Árvore SBB



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

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

Análise

- De fato Bayer (1972) mostrou que

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

- Custo para manter a propriedade SBB $\Rightarrow$ Custo para percorrer o caminho de pesquisa para encontrar a chave, seja para inserí-la ou para retirá-la.
- **Logo:** O custo é $O(\log n)$.
- Número de comparações em uma pesquisa com sucesso é:

  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).