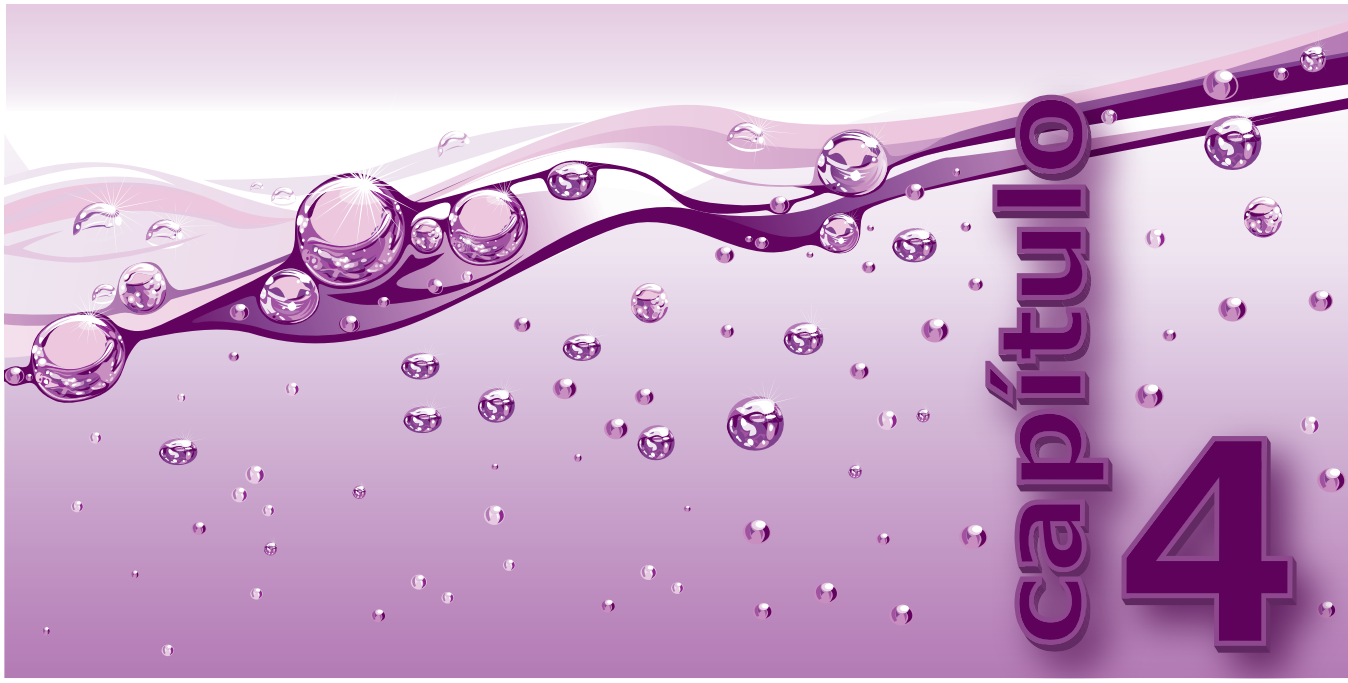


Busca Hierárquica em Memória Principal

Após estudar este capítulo, você deverá ser capaz de:

- Definir e usar os seguintes conceitos:
  - ❒ Árvore binária de busca
  - ❒ Árvore binária degenerada
  - ❒ Rotações direita e esquerda
  - ❒ Árvore binária balanceada
  - ❒ Árvore autoajustável
  - ❒ Árvore AVL
  - ❒ Árvore afunilada
  - ❒ Zig e zag
  - ❒ Zig-zag
  - ❒ Zig-zig
  - ❒ Zag-zag
  - ❒ Zag-zig
- Explicar quando não se deve usar árvore binária ordinária para implementar tabela de busca
- Descrever os algoritmos de busca e inserção em árvore binária ordinária de busca
- Identificar os casos de remoção de chave em árvore binária ordinária de busca
- Mostrar como se encontra o sucessor/antecessor imediato de um nó de árvore binária de busca
- Provar que caminhamento infixo em árvore binária de busca visita suas chaves em ordem crescente
- Descrever as operações de rotação em árvores binárias
- Mostrar que rotações preservam a ordenação de árvores binárias de busca
- Explicar como é efetuado o balanceamento de uma árvore AVL
- Descrever a operação de afunilamento
- Contrastar árvore AVL e árvore afunilada

objetivos

No Capítulo 3, foram apresentadas tabelas de busca implementadas como estruturas lineares que apresentam vantagens e desvantagens resumidamente enumeradas a seguir:

- Tabelas de busca implementadas como listas indexadas sem ordenação apresentam custo temporal $\theta(n)$ para operações de busca, inserção e remoção. Por outro lado, tabelas de busca implementadas como listas encadeadas admitem inserção com custo temporal $\theta(1)$, mas, em compensação, as operações de busca e remoção têm custo temporal $\theta(n)$.
- Tabelas implementadas como listas indexadas ordenadas permitem buscas com custo temporal $\theta(\log n)$ (usando busca binária) ou $\theta(\log \log n)$ (usando busca por interpolação), mas, por outro lado, operações de inserção e remoção apresentam custo temporal $\theta(n)$.
- Listas com saltos são excelentes alternativas para implementação de tabelas de busca que realizam operações em memória principal. De fato, o custo temporal das operações de busca, inserção e remoção numa lista com saltos, no pior caso, é $\theta(n)$, mas esse caso é muito difícil de ocorrer na prática. Na prática, entretanto, essas listas apresentam custo temporal esperado $\theta(\log n)$ para essas operações, o que as torna bem competitivas.

Assim como o capítulo anterior, este capítulo lida com busca em memória principal. Contudo o presente capítulo trata de tabelas implementadas como árvores binárias e não como listas.

A primeira estrutura de dados a ser discutida neste capítulo (v. **Seção 4.1**) é a árvore binária ordinária de busca. Essa estrutura de dados oferece custo temporal $\theta(\log n)$ em operações de busca, inserção e remoção no melhor caso. No pior caso, todavia, quando a árvore é degenerada, o custo de cada uma dessas operações é $\theta(n)$, o que torna árvores binárias ordinárias de busca uma opção pior do que listas encadeadas, por exemplo.

As duas próximas estruturas a serem discutidas neste capítulo são árvores binárias balanceadas de busca (árvores AVL na **Seção 4.4**), que garantem busca, inserção e remoção com custo temporal $\theta(\log n)$ mesmo no pior caso[1]. Entretanto quem paga o preço por esse formidável desempenho é o programador responsável por implementá-las, pois essa tarefa exige um esforço considerável.

Por fim, a última estrutura de dados hierárquica usada em implementações de tabelas de busca em memória principal a ser discutida neste capítulo é a árvore afunilada (v. **Seção 4.5**). Esse tipo de estrutura é novamente uma árvore binária de busca, mas não é nem ordinária nem balanceada. Quer dizer, esse tipo de árvore pode ser classificado como autoajustável, pois ela é reestruturada cada vez que é acessada numa operação de busca, inserção ou remoção. A ideia na qual se baseiam árvores afuniladas é a mesma que norteia a noção de memória cache (v. **Capítulo 1**): facilitar o acesso aos elementos acessados mais recentemente. A análise de custo temporal de árvores afuniladas requer uma nova ferramenta de análise denominada análise amortizada, que será discutida no **Capítulo 5**.

Na **Seção 4.6**, será apresentada uma comparação entre todas as estruturas de dados discutidas neste capítulo.

4.1 Árvores Binárias Ordinárias de Busca

4.1.1 Conceitos

Uma **árvore binária de busca** é um tipo de árvore binária usado para implementar tabelas de busca e que satisfaz as seguintes propriedades[2]:

[1] **Árvore rubro-negra** é outro tipo bem conhecido de árvore balanceada. Esse tipo de árvore é discutido exclusivamente no **Apêndice F**, que se encontra no site dedicado a este livro na internet.

[2] Por conveniência, este capítulo lida apenas com chaves primárias. Mas isso não constitui uma limitação de árvores binárias de busca. Se forem usadas chaves secundárias, basta trocar *menor* por *menor ou igual* ou *maior* por *maior ou igual* nas descrições.

- Para todo nó X, se sua subárvore esquerda não for vazia, todas as chaves que se encontram nessa subárvore são menores do que a chave do nó X (v. **Figura 4–1**).
- Para todo nó X, se sua subárvore direita não for vazia, todas as chaves que se encontram nessa subárvore são maiores do que a chave do nó X (novamente, v. **Figura 4–1**).

Uma propriedade importante decorrente da definição de árvores binárias de busca é o fato de a menor chave armazenada numa tal árvore ser aquela que se encontra no nó mais à esquerda na árvore. Isto é, para encontrar a menor chave que se encontra numa árvore binária de busca desce-se a árvore sempre seguindo o ponteiro para o filho esquerdo de cada nó visitado. Quando não for mais possível seguir um desses ponteiros, o último nó visitado conterá a menor chave da árvore, como ilustra a **Figura 4–2**. Raciocínio semelhante é empregado para encontrar a maior chave armazenada numa árvore binária de busca. Ou seja, nesse último caso, basta seguir sempre o ponteiro para o filho direito de cada nó visitado. O último nó encontrado nesse percurso conterá a maior chave.

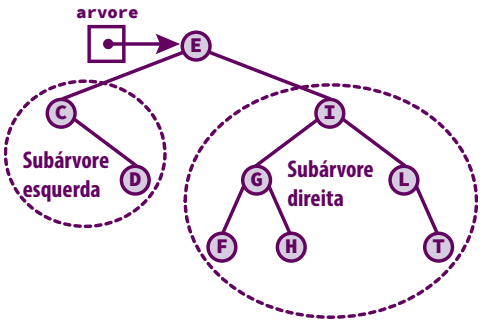


Figura 4–1: Relação entre Chaves numa Árvore Binária de Busca

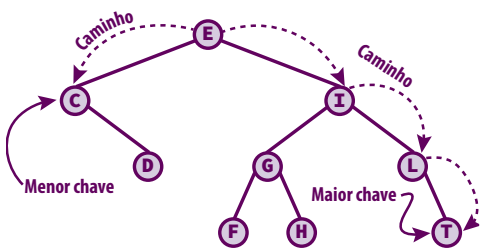


Figura 4–2: Maior e Menor Chaves de uma Árvore Binária de Busca

É importante alertar que o fato de a menor chave de uma árvore binária ser encontrada do modo descrito no último parágrafo, não implica necessariamente que se esteja lidando com uma árvore binária de busca. Raciocínio similar é empregado para concluir que o fato de a maior chave de uma árvore binária encontrar-se em seu nó mais à direita não implica que essa árvore seja uma árvore binária de busca. É fácil mostrar que essas duas afirmações são verdadeiras por meio de contraexemplos.

Teorema 4.1: Uma árvore binária é uma árvore binária de busca se, e somente se, um caminhamento em ordem infixa acessa as chaves armazenadas nessa árvore em ordem crescente.

Prova: (⇒) Deve-se mostrar que, se uma árvore binária é uma árvore binária de busca, um caminhamento visita as chaves dessa árvore em ordem crescente. Essa prova será feita por indução sobre a altura a da árvore.

Base da indução. Se $a = 0$, o teorema é válido, pois a árvore é vazia e, consequentemente, a lista de nós visitados é vazia (e toda lista vazia é considerada ordenada).

Hipótese indutiva. Suponha que o teorema vale para todo k, tal que $0 \leq k < a$.

Passo indutivo. Deve-se mostrar que o teorema é válido para $k = a$. Seja A uma árvore binária de busca de altura a. Então as subárvores esquerda e direita, respectivamente, A_E e A_D têm alturas menores do que a. Portanto, de acordo com a hipótese indutiva, um caminhamento em ordem infixa em cada uma dessas subárvores acessa suas chaves em ordem crescente. Como A é uma árvore de busca, a chave que se encontra na raiz de A é maior do que qualquer chave em A_E e menor do que qualquer chave em A_D. Portanto, um caminhamento em ordem infixa em A acessa as chaves dessa árvore em ordem crescente.

(⇐) Agora suponha que um caminhamento em ordem infixa acessa as chaves de uma árvore binária A em ordem crescente e que a subárvore esquerda de A seja A_E e que a subárvore direita de A seja A_D. Como é efetuado um caminhamento em ordem infixa em A_E antes que a raiz de A seja visitada, ou A_E

é uma subárvore vazia ou todas as suas chaves são menores do que a chave que se encontra na raiz de A. Usando um raciocínio similar, conclui-se que A_D é uma subárvore vazia ou todas as suas chaves são maiores do que a chave que se encontra na raiz de A. Portanto A é uma árvore binária de busca. ■

O **Teorema 4.1** permite decidir se uma árvore binária é uma árvore binária de busca ou não. Quer dizer, de acordo com esse teorema, se você efetuar um caminhamento em ordem infixa numa árvore binária contendo chaves e essas chaves forem visitadas em ordem crescente, você pode concluir que se trata de uma árvore binária de busca (v. exemplo na **Seção 4.7.3**).

Inserção

A inserção de uma chave (ou registro) numa árvore binária de busca segue o algoritmo descrito na **Figura 4–3**.

ALGORITMO INSEREEMÁRVOREBINÁRIADEBUSCA

ENTRADA: O conteúdo de um novo elemento (nó)

ENTRADA/SAÍDA: Uma árvore binária de busca

1. Se a árvore estiver vazia, armazene o elemento na raiz da árvore e retorne
2. Se a chave do novo elemento for menor do que a chave que se encontra na raiz:
   - **2.1** Se a subárvore esquerda estiver vazia, torne o novo elemento filho esquerdo da raiz e retorne
   - **2.2** Caso contrário, repita este procedimento a partir do **Passo 2** usando a raiz dessa subárvore
3. Se a chave do novo elemento for maior do que a chave que se encontra na raiz:
   - **3.1** Se a subárvore direita estiver vazia, torne o novo elemento filho direito da raiz e retorne
   - **3.2** Caso contrário, repita este procedimento a partir do **Passo 2** usando a raiz dessa subárvore

FIGURA 4–3: ALGORITMO DE INSERÇÃO EM ÁRVORE BINÁRIA DE BUSCA

Suponha, por exemplo, que as letras E, I, G, C, H, L, F, D e T representem chaves associadas aos elementos que se desejam inserir numa árvore binária de busca na ordem em que essas chaves se encontram. A **Figura 4–4 (a)** mostra a árvore inicialmente vazia representada por um ponteiro nulo (ilustrado pela barra invertida na referida figura), enquanto **Figura 4–4 (b)** apresenta a árvore resultante da inserção do elemento contendo a chave E. Nesse último caso, segue-se apenas o **Passo 1** do algoritmo de inserção exibido na **Figura 4–4**.

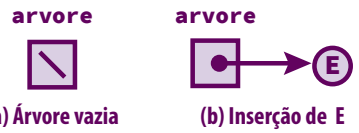


FIGURA 4–4: INSERÇÃO NUMA ÁRVORE BINÁRIA DE BUSCA VAZIA

Lembre-se que, como foi visto no **Capítulo 3**, elementos de uma tabela de busca são constituídos por pares chave/valor, embora na maioria das ilustrações de tabelas de busca só apareçam as chaves. Isso ocorre por duas razões: (1) nas operações de busca, inserção e remoção do componente de tal par que recebe mais atenção é a chave e (2) além de não contribuir para absorção do conhecimento que se deseja transmitir, o uso desses pares completos nessas ilustrações as tornariam bem mais complicadas de desenhar. Portanto para simplificar as figuras e facilitar o entendimento, apenas as chaves aparecem como conteúdos dos nós na **Figura 4–4** e na maioria das figuras deste livro que ilustram tabelas de busca.

A **Figura 4–5 (c)** ilustra a inserção do elemento contendo a chave I na árvore do mesmo exemplo. Nessa inserção, seguem-se os **Passos 1**, **3** e **3.1** do algoritmo de inserção. Por outro lado, a inserção do elemento que contém a chave G mostrada na **Figura 4–5 (d)** segue os **Passos 1**, **3**, **3.2**, **2** e **2.1**, nessa ordem. Nesse último caso, o **Passo 1** é executado para verificar se a árvore está vazia; o **Passo 3** é executado porque, além de a árvore

não estar vazia, a chave do novo elemento é maior do que a chave que se encontra na raiz da árvore. Então o **Passo 3.2** é executado porque essa raiz já possui um filho direito. O **Passo 2** é levado a efeito porque a chave do novo nó é menor do que a chave desse filho direito e, finalmente, o **Passo 2.1** é aplicado, de forma que o nó contendo a chave *G* é inserido como filho esquerdo do nó que contém a chave *G*.

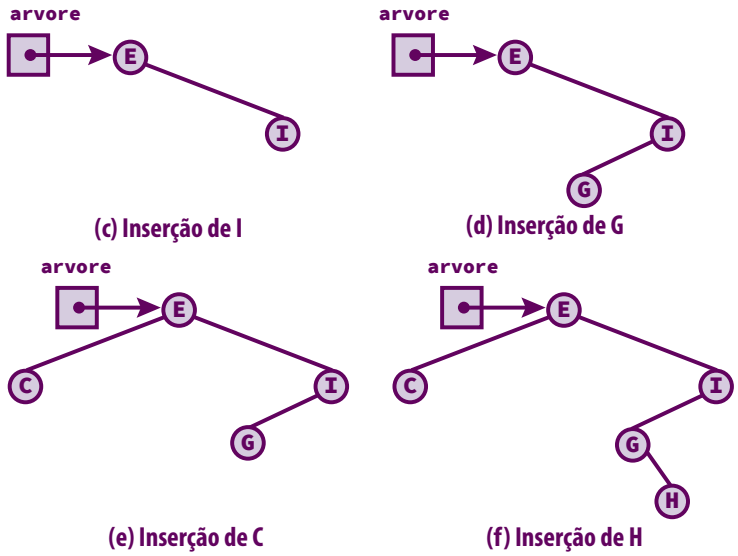


Figura 4–5: Inserção numa Árvore Binária de Busca 1

A inserção do elemento contendo a chave *C*, mostrada na **Figura 4–5 (e)**, é simples: basta seguir os **Passos 1**, **2** e **2.1** do algoritmo de inserção (v. **Figura 4–3**). Já a inserção do elemento que contém a chave *H*, mostrada na **Figura 4–5 (f)**, é um pouco mais complicada, pois requer a execução dos **Passos 1**, **2**, **3**, **3.2**, **2**, **2.1**, **2**, **3** e **3.1**, nessa ordem.

O leitor é convidado a justificar, com base no algoritmo apresentado na **Figura 4–3**, as inserções das chaves *L*, *F*, *D* e *T*. Os resultados dessas inserções são ilustrados na **Figura 4–6**.

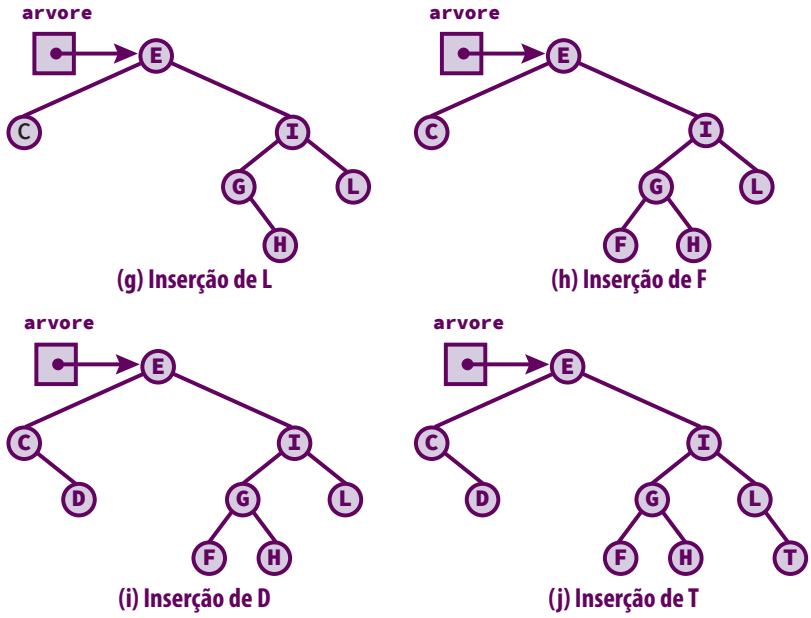


Figura 4–6: Inserção numa Árvore Binária de Busca 2

A **Figura 4–6 (j)** mostra que a árvore resultante das inserções de chaves do último exemplo apresentado realmente satisfaz as propriedades enumeradas no início desta seção. Quer dizer, todas as chaves na subárvore esquerda de *E* são menores do que *E* e todas as chaves na subárvore direita de *E* são maiores do que *E*. É importante notar ainda que toda subárvore de uma árvore binária de busca também é uma árvore binária de busca. Por exemplo, na **Figura 4–6 (j)**, a subárvore direita de *E* também é uma árvore binária de busca, pois todas as chaves na subárvore esquerda de *I* são menores do que *I* e todas as chaves na subárvore direita de *I* são maiores do que *I*. O mesmo raciocínio pode ser usado para qualquer outra subárvore mostrada na **Figura 4–6 (j)**.

Um determinado conjunto de chaves pode originar diversas árvores binárias dependendo da ordem na qual elas são inseridas. Suponha, por exemplo, que se tenha à disposição o seguinte conjunto de chaves: *1*, *3*, *7*, *8*, *12*, *13* e *16*. Quando as chaves que compõem esse conjunto são inseridas numa árvore binária de busca nessa ordem, o resultado é uma árvore que muito se assemelha a uma lista encadeada, como mostra a **Figura 4–7**. De fato, do ponto de vista prático de operações básicas em tabelas de busca, a árvore que aparece nessa figura é equivalente a uma lista simplesmente encadeada ordenada em ordem crescente de chaves e com o agravante de ocupar mais espaço[3].

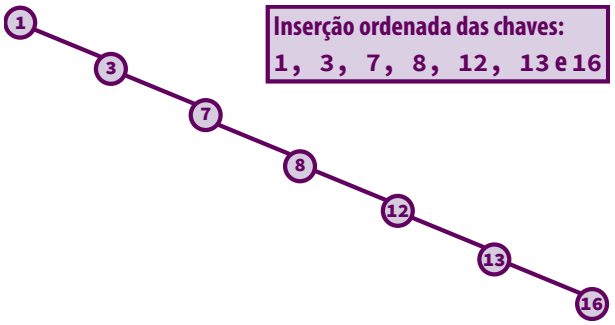


Figura 4–7: Chaves Ordenadas Inseridas numa Árvore Binária de Busca

A árvore binária de busca da **Figura 4–7** representa um dos piores casos para operações elementares de uma tabela de busca[4]. Ou seja, nesse caso, operações de busca, inserção e remoção têm todas custo temporal $\theta(n)$, o que se poderia obter com a implementação de tabela de busca com o pior custo temporal apresentada no capítulo anterior.

Se, neste ponto da discussão, o leitor achar que não faz sentido estudar árvores de busca binárias ordinárias (i.e., tal qual elas foram descritas nesta seção), ele tem razão, pois, de fato, árvores binárias de busca ordinárias raramente são usadas na prática. Quer dizer, o programador só deve utilizar tabelas de busca implementadas como árvores binárias de busca ordinárias quando tiver absoluta certeza que o pior caso jamais ocorrerá. Mas como tipicamente o programador desconhece a configuração das chaves, árvores binárias de busca ordinárias serão exploradas mais por razões didáticas do que práticas, pois elas são mais fáceis de entender e implementar do que aquelas que são usadas na prática e que serão exploradas em seções posteriores.

Busca

O algoritmo de busca para árvores binárias de busca, apresentado na **Figura 4–8**, é bem parecido com o algoritmo usado em inserção. Esse algoritmo de busca é usado por qualquer árvore binária de busca discutida neste capítulo, com exceção das árvores afuniladas, que serão discutidas na **Seção 4.5**.

[3] Quer dizer, nesse caso, uma lista simplesmente encadeada utilizaria apenas um ponteiro para o próximo elemento da lista, enquanto uma árvore binária de busca utiliza dois ponteiros, sendo que um deles nunca é usado nesse caso.

[4] O outro pior caso extremo ocorre quando as chaves estão ordenadas em ordem inversa. Outros casos tão ruins quanto esses ocorrem quando as chaves estão quase ordenadas em ordem direta ou inversa.

Algoritmo BuscaEmÁrvoreBináriaDeBusca

Entrada: Uma árvore binária de busca e uma chave de busca

Saída: O valor associado à chave que casa com a chave de busca ou um valor informando que a chave não foi encontrada

1. Faça um ponteiro *p* apontar para a raiz da árvore
2. Enquanto *p* não for um ponteiro nulo, faça:
   - **2.1** Se a chave de busca for igual àquela do nó para o qual *p* aponta, retorne o valor associado a essa chave
   - **2.2** Se a chave de busca for menor do que a chave do nó para o qual *p* aponta, faça *p* apontar para o filho esquerdo desse nó
   - **2.3** Se a chave de busca for maior do que a chave do nó para o qual *p* aponta, faça *p* apontar para o filho direito desse nó
3. Retorne um valor informando que a chave não foi encontrada

Figura 4–8: Algoritmo de Busca em Árvore Binária de Busca

A **Figura 4–9** mostra uma busca bem-sucedida, ao passo que a **Figura 4–10** ilustra uma busca malsucedida numa árvore binária de busca. Novamente, o leitor é convidado a justificar como essas buscas são efetuadas à luz do algoritmo da **Figura 4–8**.

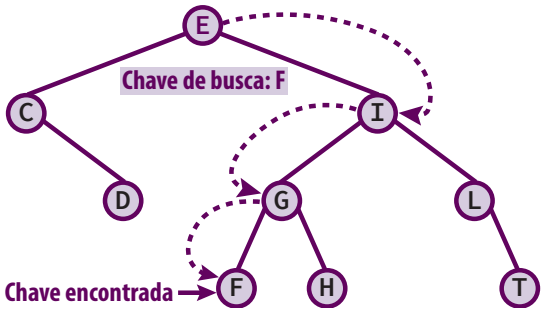


Figura 4–9: Busca Bem-sucedida numa Árvore Binária de Busca

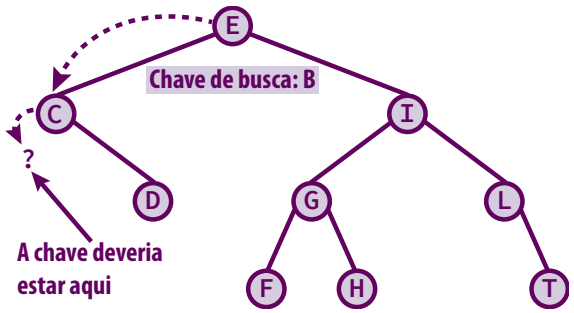


Figura 4–10: Busca Malsucedida numa Árvore Binária de Busca

Remoção

Para remover um nó de uma árvore binária de busca, três casos devem ser levados em consideração de acordo com o número de filhos do nó a ser removido:

- **Caso 1: O nó a ser removido não possui filhos**. Nesse caso, ele pode ser eliminado sem outros ajustes na árvore, conforme mostra a **Figura 4–11**.

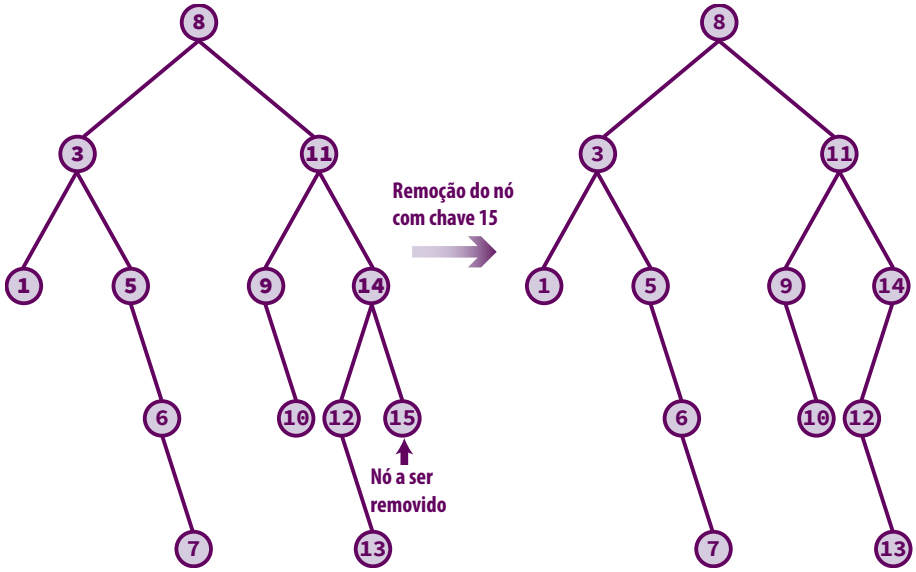


Figura 4–11: Remoção de Nó numa Árvore Binária de Busca: Caso 1

- **Caso 2: O nó a ser removido possui apenas um filho**. Nesse caso, o único filho é movido para cima para ocupar o lugar do nó removido, como mostra a **Figura 4–12**.

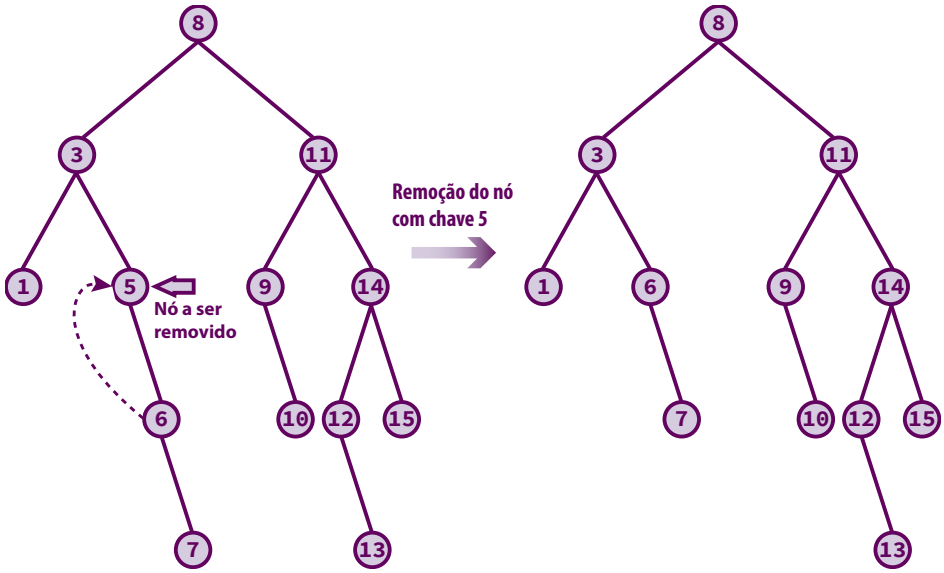


Figura 4–12: Remoção de Nó numa Árvore Binária de Busca: Caso 2

- **Caso 3: O nó a ser removido possui dois filhos**. Neste caso, o sucessor imediato em ordem infixa deve ocupar o lugar do nó removido. O **sucessor** (**imediato**) de um nó numa árvore binária de busca é o nó que contém a menor chave que é maior do que a chave do nó em questão. Portanto para encontrar o sucessor de um nó que possui filho direito, começa-se a busca por esse filho e prossegue-se descendo na árvore sempre seguindo o filho esquerdo de cada nó visitado até que se encontre um ponteiro nulo (i.e., um nó sem filho esquerdo). O último nó visitado seguindo esse procedimento é o sucessor procurado. Esse sucessor não pode ter filho à esquerda (por que será?). Assim o filho direito desse sucessor pode ser movido para cima para ocupar o lugar do próprio sucessor. Este caso é ilustrado na **Figura 4–13**.

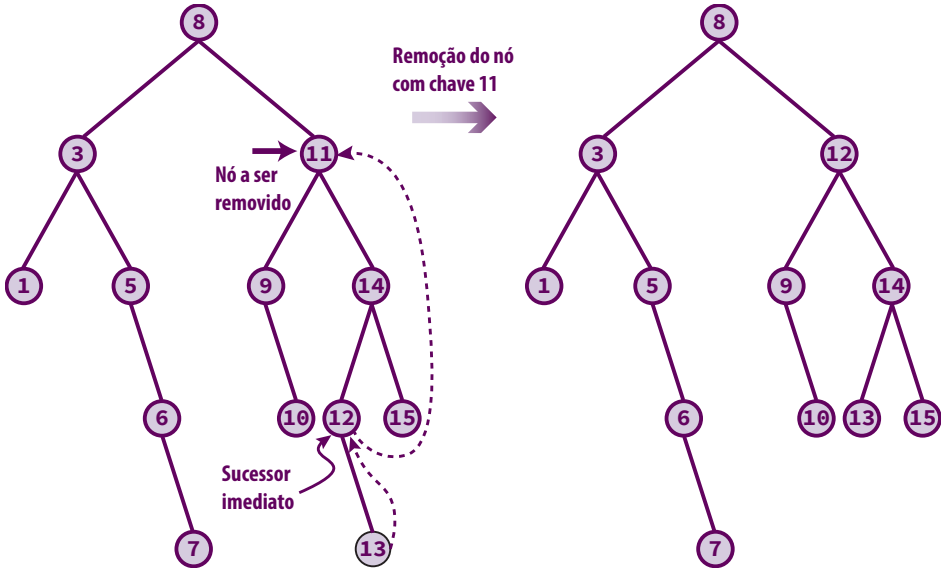


Figura 4–13: Remoção de Nó numa Árvore Binária de Busca: Caso 3

Alternativamente, no **Caso 3** de remoção, o nó a ser removido pode ser substituído por seu **antecessor** (em vez de sucessor) **imediato**, que é o nó que possui a maior chave menor do que a chave do nó em questão. Não há nenhuma vantagem ou desvantagem de uma alternativa sobre a outra. Quer dizer, a substituição do nó removido por esse antecessor produz exatamente o mesmo resultado que a substituição pelo referido sucessor. Aqui, se escolheu usar o referido sucessor como substituto do nó removido.

O algoritmo de remoção de nós de árvores binárias de busca, apresentado na **Figura 4–14**, resume o arrazoado apresentado acima.

Algoritmo RemoveEmÁrvoreBináriaDeBusca

Entrada: A chave do nó a ser removido

Entrada/Saída: Uma árvore binária de busca

Saída: Um valor informando se a operação foi bem-sucedida

1. Efetue uma busca pelo nó que contém a chave de busca usando o algoritmo de busca da **Figura 4–8**
2. Se a chave não for encontrada, encerre informando o fracasso da operação
3. Se o nó a ser removido for uma folha, remova-a e retorne informando o sucesso da operação
4. Se o nó a ser removido possuir apenas um filho, substitua o referido nó por seu filho e retorne informando o sucesso da operação
5. Se o nó a ser removido possuir dois filhos faça o seguinte:
   - **5.1** Encontre o nó que é sucessor imediato do nó a ser removido
   - **5.2** Guarde o conteúdo do nó sucessor
   - **5.3** Remova o nó sucessor usando o **Passo 3** ou o **Passo 4**
   - **5.4** Substitua o conteúdo do nó que seria removido pelo conteúdo do nó sucessor

Figura 4–14: Algoritmo de Remoção de Nó em Árvore Binária de Busca

4.1.2 Implementação

O arquivo de dados a ser utilizado na implementação de árvore binária de busca será `CEPs.bin` (v. **Apêndice A**).

Definições de Tipos

As seguintes definições de tipos serão usadas nesta implementação de árvore binária de busca:

```
typedef struct rotNoABB {
           struct rotNoABB *esquerda;
           tCEP_Ind         conteudo;
           struct rotNoABB *direita;
        } tNoArvoreBB, *tArvoreBB;
```

Os tipos `tCEP` e `tCEP_Ind` usados nesta implementação foram definidos na **Seção 3.3.2** e também são exibidos no **Apêndice A**.

Iniciação

A função `IniciaArvoreBB()`, inicia uma árvore binária de busca fazendo com que o ponteiro que a representa seja nulo.

```
void IniciaArvoreBB(tArvoreBB *arvore)
{
   *arvore = NULL;
}
```

Busca

A função `BuscaArvoreBB()` implementa a busca em árvore binária delineada no algoritmo apresentado na **Figura 4–8**. Essa função retorna o endereço do conteúdo do nó que possui uma determinada chave numa árvore binária de busca se ela for encontrada; ou **NULL**, se a busca não for bem-sucedida. Os parâmetros dessa função são:

- `arvore` (entrada) — ponteiro para a raiz da árvore na qual será efetuada a busca
- `chave` (entrada) — a chave de busca

```
tCEP_Ind *BuscaArvoreBB(tArvoreBB arvore, tCEP chave)
{
   int compara; /* Resultado da comparação de duas chaves */

      /* Enquanto 'arvore' não assume NULL ou a chave de */
      /* um nó não casa com 'chave', a busca prossegue   */
   while (arvore) {
         /* Compara a chave de busca com a chave do nó corrente */
      compara = strcmp(chave, arvore->conteudo.chave);

         /* Verifica se a chave foi encontrada */
      if (!compara)
         return &arvore->conteudo; /* Chave encontrada */

         /* Se a chave de busca for menor do que a chave do nó */
         /* corrente, desce-se pela subárvore esquerda. Caso   */
         /* contrário, desce-se pela subárvore direita.        */
      arvore = compara < 0 ? arvore->esquerda : arvore->direita;
   }

   return NULL; /* A chave não foi encontrada */
}
```

Inserção

A função `InsereArvoreBB()`, apresentada a seguir, insere uma nova chave numa árvore binária de busca e seus parâmetros são:

- `*arvore` (entrada/saída) — ponteiro para a raiz da árvore na qual será feita a inserção
- `*conteudo` (entrada) — conteúdo (chave/valor) do nó que será inserido

A função `InsereArvoreBB()` retorna `1`, se não houver inserção porque a chave já existe, ou `0`, se a inserção for exitosa.

```
int InsereArvoreBB( tArvoreBB *arvore, const tCEP_Ind *conteudo )
{
   tArvoreBB p, /* Ponteiro usado para descer na árvore */
             q, /* Ponteiro que apontará para o pai de p */
             pNovoNo; /* Ponteiro para nó que será inserido */
   int       compara; /* Resultado da comparação de duas chaves */

   p = *arvore; /* Começa a busca na raiz da árvore */
   q = NULL; /* A raiz não tem pai */

      /* Desce na árvore até encontrar um ponteiro nulo */
   while (p) {
         /* Compara a chave do novo nó com a chave do nó corrente */
      compara = strcmp(conteudo->chave, p->conteudo.chave);

         /* A chave é primária. Portanto, se ela */
         /* for encontrada, não haverá inserção. */
      if (!compara) /* A chave foi encontrada */
         return 1; /* Não há mais nada a fazer */

         /* Faz q apontar para o nó para o qual */
         /* p aponta antes que p mude de valor  */
      q = p;

         /* Se a chave de busca for menor do que a chave do nó corrente, */
         /* desce-se à esquerda. Caso contrário, desce-se à direita.     */
      if (compara < 0)
         p = p->esquerda;
      else
         p = p->direita;
   }

      /*** Aqui, sabe-se que a chave não foi encontrada ***/

   pNovoNo = ConstroiNoArvoreBB(*conteudo);  /* Constrói um novo nó */

      /* Se a árvore não estiver vazia, q está apontando */
      /* para o nó que será o pai do novo nó             */
   if (!q) /* A árvore estava vazia */
      *arvore = pNovoNo; /* Novo nó será a raiz */

      /* Se a chave do novo nó for menor do que a chave */
      /* de seu pai, ele será o filho da esquerda. Caso */
      /* contrário, ele será o filho da direita.        */
   else if (strcmp(conteudo->chave, q->conteudo.chave) < 0)
         /* q aponta para o nó que terá um filho à esquerda */
      q->esquerda = pNovoNo;
   else
         /* q aponta para o nó que terá um filho à direita */
      q->direita = pNovoNo;

   return 0; /* Inserção foi OK */
}
```

A **Figura 4–15** e a **Figura 4–16** ilustram a atuação da função `InsereArvoreBB()` em duas inserções de nós.

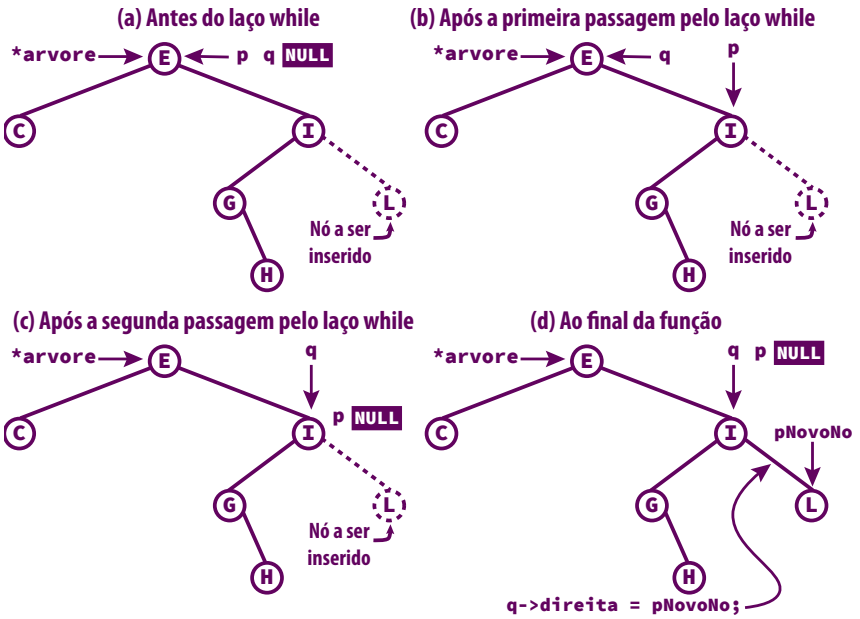


FIGURA 4–15: INSERÇÃO DE UM NÓ USANDO INSEREARVOREBB() 1

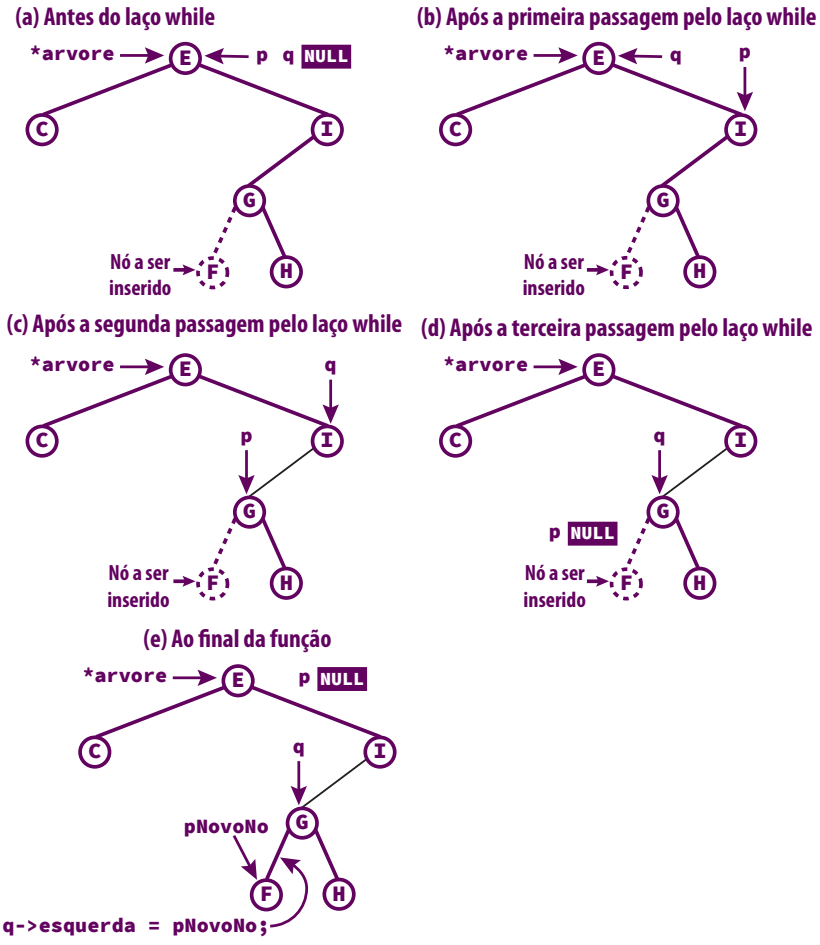


FIGURA 4–16: INSERÇÃO DE UM NÓ USANDO INSEREARVOREBB() 2

A função `InsereArvoreBB()` chama a função `ConstroiNoArvoreBB()`, definida a seguir, para criar um nó de uma árvore binária. A função `ConstroiNoArvoreBB()` possui um único parâmetro — `item` — que representa o conteúdo efetivo do nó que será criado e cujo endereço será retornado. Essa função é definida com `static` porque ela não será de interesse de nenhum programa-cliente.

```
static tArvoreBB ConstroiNoArvoreBB(tCEP_Ind item)
{
   tArvoreBB no;

   no = malloc(sizeof(tNoArvoreBB)); /* Tenta alocar o novo nó */

      /* Se não houve alocação, aborta o programa */
   ASSEGURA(no, "Erro: Nao foi possivel alocar no'");

   no->conteudo = item; /* Preenche o conteúdo do nó */
   no->esquerda = NULL; /* Este nó ainda não tem filho */
   no->direita = NULL;

   return no;
}
```

Remoção

A função `RemoveArvoreBB()`, apresentada adiante, remove um nó de uma árvore binária de busca levando em consideração os três casos discutidos acima. Essa função usa os seguintes parâmetros:

- `arvore` (entrada/saída) — endereço do ponteiro que representa a árvore na qual será feita a remoção
- `chave` (entrada) — chave do nó a ser removido

A função `RemoveArvoreBB()` retorna `0`, se a remoção for bem-sucedida, ou `1`, se o nó a ser removido não for encontrado.

```
int RemoveArvoreBB(tArvoreBB *arvore, tCEP chave)
{
   tArvoreBB p, /* Apontará para o nó que será removido */
             q, /* Apontará para o pai de p */
             subs, /* Apontará para o nó que substituirá p */
             filho, /* O filho esquerdo de 'subs' */
             pai; /* O pai de 'subs' */
   int       compara; /* Resultado da comparação de duas chaves */

   p = *arvore; /* Começa a busca na raiz da árvore */
   q = NULL; /* A raiz não tem pai */

      /* Procura o nó a ser removido. O laço encerra quando */
      /* esse nó ou um ponteiro nulo for encontrado.        */
   while (p) {
         /* Compara a chave que será removida com a chave do nó corrente */
      compara = strcmp(chave, p->conteudo.chave);

         /* Verifica se a chave foi encontrada */
      if (!compara)
         break; /* Chave encontrada */

        /* Faz q apontar para o nó para o qual p aponta antes de alterar o valor de p */
      q = p;

         /* Se a chave procurada for menor do que a chave do nó corrente, */
         /* desce-se pela subárvore esquerda. Caso contrário, desce-se    */
         /* pela subárvore direita.                                       */
      p = compara < 0 ? p->esquerda : p->direita;
   }
```

```
   /* Se a chave não foi encontrada, a tarefa está encerrada */
if (!p)
   return 1; /* Não ocorreu remoção */

/*********************************************************************/
/* Neste ponto, p aponta para o nó que será removido e q aponta para seu */
/* pai. O próximo passo é encontrar o nó que substituirá o nó apontado   */
/* por p. O ponteiro 'subs' apontará para esse nó substituto.            */
/*********************************************************************/

   /* Faz 'subs' apontar para o nó que substituirá p */
if (!p->esquerda)
      /* O nó a ser eliminado não possui nenhum filho ou só tem  */
      /* filho à direita. Esse filho ocupará o lugar do seu pai. */
    subs = p->direita;
else if (!p->direita)
      /* O nó a ser eliminado só tem filho à esquerda. */
      /* Esse filho ocupará o lugar do seu pai.        */
    subs = p->esquerda;
else {
      /* p tem dois filhos. Neste caso, faz-se 'subs' apontar para o sucessor */
      /* em ordem infixa de p e 'pai' apontar para o pai de 'subs'.           */
   pai = p;
   subs = p->direita;

   filho = subs->esquerda; /* 'filho' é sempre o filho à esquerda de 'subs' */

      /* Faz 'subs' apontar para o sucessor em ordem infixa de p e 'pai' */
      /* apontar para o pai de  'subs'. Usa-se o ponteiro 'filho' para   */
      /* descer na árvore sempre à esquerda até que 'filho' seja NULL.   */
   while (filho) {
      pai = subs;
      subs = filho;
      filho = subs->esquerda;
   }

      /* Neste ponto, sabe-se que 'subs' é o sucessor em ordem infixa */
      /* de p. Se 'pai' e 'p' estiverem apontando para o mesmo nó,    */
      /* 'subs' não tem filho à esquerda.                             */
   if (pai != p) {
         /* Faz com que o filho direito de 'subs' */
         /* torne-se o filho esquerdo de 'pai'    */
      pai->esquerda = subs->direita;

         /* Substitui o filho direito de 'subs' pelo filho direito */
         /* de p, que aponta para o nó a ser removido              */
      subs->direita = p->direita;
   }

      /* Substitui o filho esquerdo de 'subs' pelo filho esquerdo de p */
   subs->esquerda = p->esquerda;
} /* else */

/*** Faz o nó apontado por 'subs' ocupar o lugar do nó apontado por p ***/

   /* Verifica se o nó a ser removido é a raiz pois a remoção da raiz é especial */
if (!q) /* O nó removido era a raiz */
   *arvore = subs;
else if (p == q->esquerda)
   q->esquerda = subs; /* O filho esquerdo de q será removido */
```

```
   else
      q->direita = subs; /* O filho direito de q será removido */

   free(p); /* Libera o nó removido */

   return 0; /* A remoção foi bem sucedida */
}
```

A função `RemoveArvoreBB()` é relativamente complexa, mas, por outro lado, seu entendimento é crucial para assimilar operações sobre as árvores de busca mais sofisticadas que serão discutidas mais adiante. Para facilitar o entendimento dessa função, é útil dividir suas instruções em duas partes:

1. A primeira parte consiste em procurar o nó que deve ser removido, que é aquele que contém a chave especificada como parâmetro da referida função. Essa parte encerra com a instrução **if**:

   ```
   if (!p)
      return 1;
   ```

   Essa instrução encerra a função `RemoveArvoreBB()` quando o nó procurado não é encontrado, pois, obviamente, o nó só poderá ser removido se ele for encontrado. Agora, considerando o **Caso 3** de remoção descrito acima e ilustrado na **Figura 4–13**, a **Figura 4–17** mostra a situação ao final da execução dessa primeira parte da função.

```
p = *arvore;
q = NULL;

while (p) {
   compara = strcmp(chave, p->conteudo.chave);

   if (!compara)
      break;

   q = p;

   p = compara < 0 ? p->esquerda : p->direita;
}

if (!p)
   return 1;
```

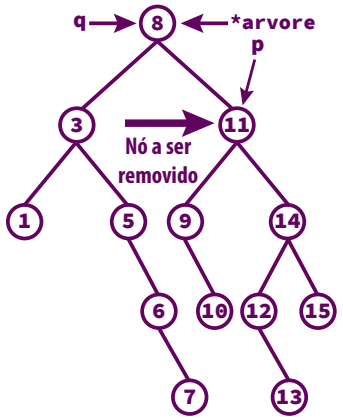


Figura 4–17: Ação da Função de Remoção de Nós numa Árvore Binária de Busca 1

2. A segunda parte da função começa na instrução **if** que segue a instrução **if** discutida acima. É essa parte que, efetivamente, lida com a remoção do nó após ele ter sido encontrado. Para facilitar a discussão que seguirá, essa parte é reproduzida abaixo sem os comentários que a acompanham na função `RemoveArvoreBB()`.

```
1.  if (!p->esquerda)
2.     subs = p->direita;
3.  else if (!p->direita)
4.     subs = p->esquerda;
5.  else {
6.     pai = p;
7.     subs = p->direita;
8.     filho = subs->esquerda;
9.
10.    while (filho) {
11.       pai = subs;
12.       subs = filho;
13.       filho = subs->esquerda;
14.    }
```

```
15.
16.    if (pai != p) {
17.       pai->esquerda = subs->direita;
18.       subs->direita = p->direita;
19.    }
20.
21.    subs->esquerda = p->esquerda;
22. }
23.
24. if (!q)
25.    *arvore = subs;
26. else if (p == q->esquerda)
27.    q->esquerda = subs;
28. else
29.    q->direita = subs;
30.
31. free(p);
```

Quando o nó a ser removido não possui nenhum filho (**Caso 1**) ou possui apenas filho direito (**Caso 2**), a condição da instrução **if** na **Linha 1** é satisfeita, de modo que a instrução na **Linha 2**:

```
subs = p->direita;
```

é executada. Essa última instrução faz com que o ponteiro `subs` aponte para o filho direito do nó a ser removido. Se esse nó não possuir filho direito `subs` assumirá **NULL**. Adiante, o nó apontado pelo ponteiro `subs` irá substituir o nó ora sendo removido. O fato de esse ponteiro ser **NULL** não afeta o resultado que será obtido, como será visto.

Quando o nó a ser removido possui apenas filho esquerdo (**Caso 2**, novamente), a seguinte instrução (**Linha 4**) será executada:

```
subs = p->esquerda;
```

Essa instrução faz com que `subs` aponte para o filho esquerdo do nó a ser removido. Novamente, o nó apontado por `subs` substituirá o nó que está sendo removido.

O caso que se pretende escrutinar aqui é o **Caso 3**, que é o mais complexo. Esse caso ocorre quando o nó a ser removido possui dois filhos e, como foi descrito acima, o nó removido deverá ser substituído pelo nó que é seu sucessor imediato. Quando esse caso ocorre, as instruções nas linhas de **6** a **23** no trecho de programa acima são executadas. Em resumo, o que essas instruções fazem é exatamente encontrar o nó que é sucessor imediato do nó a ser removido.

Voltando ao processo de remoção do nó cuja chave é *11* e cujo último status foi apresentado na **Figura 4–17**, as instruções nas **Linhas 6**, **7** e **8**:

```
6. pai = p;
7. subs = p->direita;
8. filho = subs->esquerda;
```

do trecho de programa em discussão são responsáveis pela configuração apresentada na **Figura 4–18**.

O próximo trecho da função a ser executado é o laço **while**:

```
10. while (filho) {
11.    pai = subs;
12.    subs = filho;
13.    filho = subs->esquerda;
14. }
```

A execução desse laço **while** faz com que os ponteiros `subs`, `pai` e `filho` assumam os valores ilustrados na **Figura 4–19**.

```
pai = p;
subs = p->direita;
filho = subs->esquerda;
```

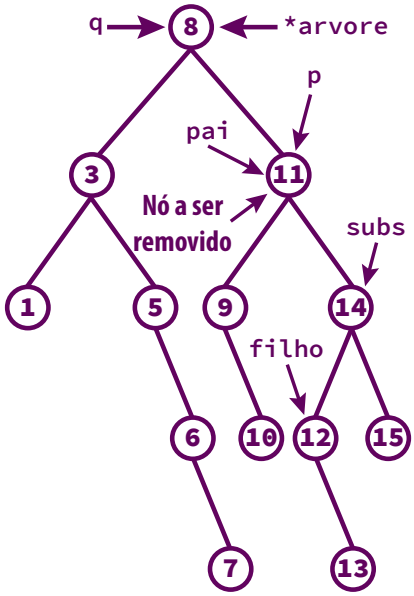
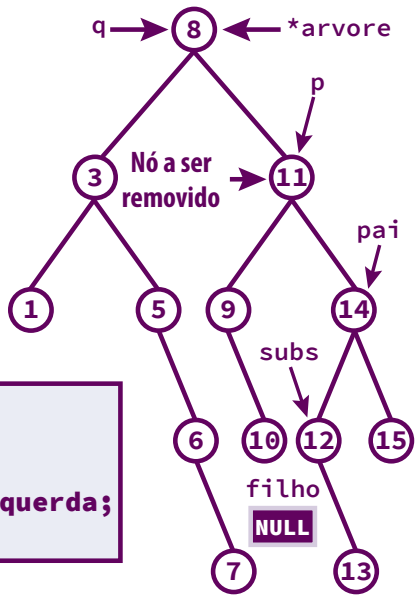


Figura 4–18: Ação da Função de Remoção de Nós numa Árvore Binária de Busca 2




```
while (filho) {
   pai = subs;
   subs = filho;
   filho = subs->esquerda;
}
```

Figura 4–19: Ação da Função de Remoção de Nós numa Árvore Binária de Busca 3

O próximo conjunto de instruções da função sob escrutínio a ser executado é:

```
16. if (pai != p) {
17.    pai->esquerda = subs->direita;
18.    subs->direita = p->direita;
19. }
20.
21. subs->esquerda = p->esquerda;
```

A execução dessas instruções é ilustrada na **Figura 4–20**.

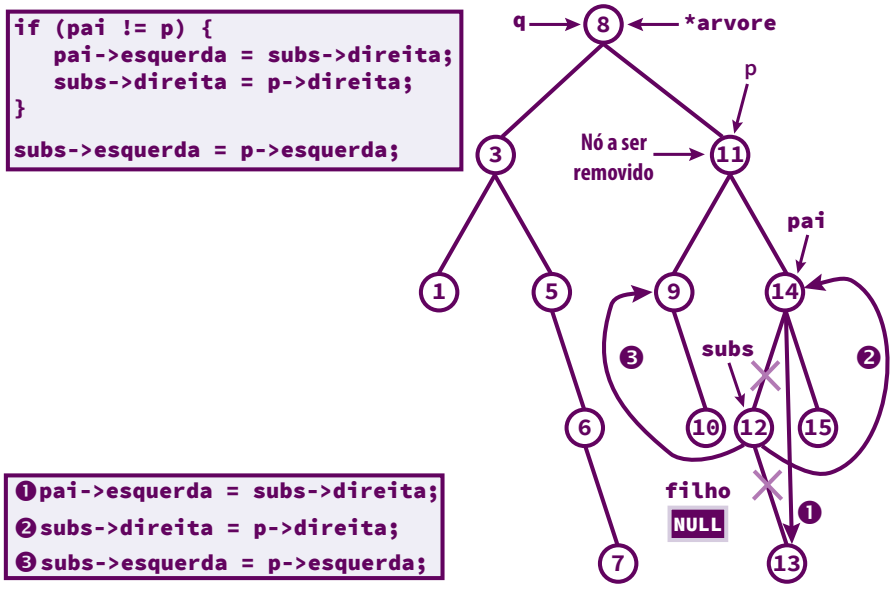


Figura 4–20: Ação da Função de Remoção de Nós numa Árvore Binária de Busca 4

As instruções que concluem a remoção são as seguintes:

```
24. if (!q)
25.    *arvore = subs;
26. else if (p == q->esquerda)
27.    q->esquerda = subs;
28. else
29.    q->direita = subs;
30.
31. free(p);
```

A **Figura 4–21** ilustra a execução dessas instruções.

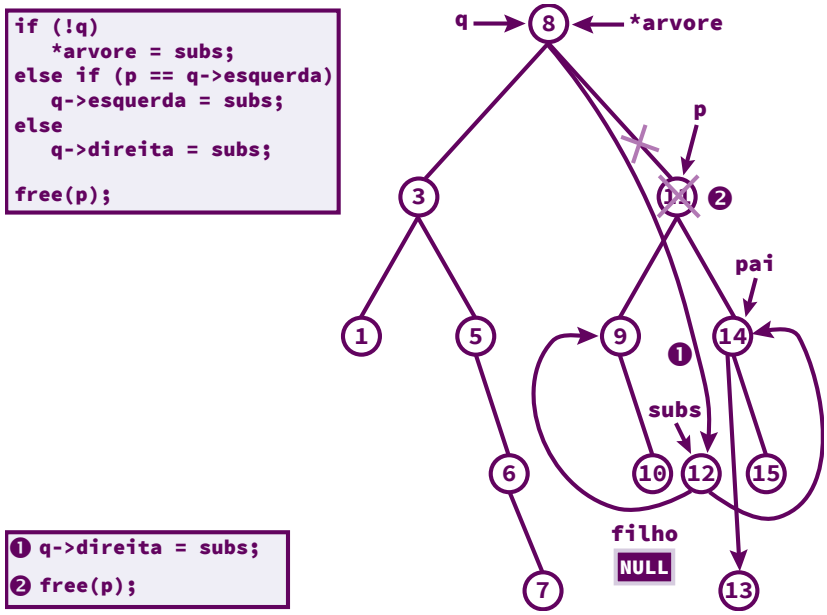


Figura 4–21: Ação da Função de Remoção de Nós numa Árvore Binária de Busca 5

4.1.3 Análise

Teorema 4.2: No pior caso, a altura de uma árvore binária ordinária de busca que armazena n chaves é n.

Prova: O pior caso de uma árvore binária ordinária de busca ocorre quando as chaves são inseridas em ordem crescente ou decrescente. Nesse caso, usando o algoritmo padrão de inserção, haverá um nó em cada nível, de modo que a altura da árvore será n. ■

Corolário 4.1: No pior caso, uma operação de busca, inserção ou remoção numa árvore binária ordinária de busca tem custo temporal $\theta(n)$, em que n é o número de chaves armazenadas na árvore.

Prova: No pior caso, qualquer uma dessas operações requer uma descida até a única folha da árvore; ou seja, essa operação requer que n nós sejam visitados. Logo o custo temporal de qualquer uma dessas operações é $\theta(n)$. ■

De acordo com o **Corolário 4.1**, os custos das operações de busca, inserção ou remoção em árvores binárias ordinárias de busca não são melhores do que os custos correspondentes para listas simplesmente encadeadas. O mais grave é que esse pior caso não é improvável na prática.

4.2 Rotações em Árvores Binárias de Busca

Rotação é uma operação que envolve um nó de uma árvore binária de busca e um dos seus filhos, que pode ser o filho esquerdo ou o filho direito de acordo com o tipo de rotação. Rotações são extremamente importantes no estudo das árvores binárias de busca que serão discutidas nas próximas seções.

Rotações alteram as posições relativas de alguns nós de uma árvore binária, de modo que sua configuração é alterada. O objetivo prático do uso de rotações é reduzir a altura de uma árvore binária de busca ao mesmo tempo que mantém suas propriedades como árvore de busca. Em geral, uma operação de rotação eleva um nó (e seus filhos) para o próximo nível superior da árvore e abaixa outro nó para o próximo nível inferior da árvore.

Existem dois tipos de rotação: (1) **rotação direita** e (2) **rotação esquerda**. Numa rotação direita, o filho esquerdo de um nó gira em torno do seu pai no sentido horário, como mostra a **Figura 4–22**[5]. O pré-requisito para que uma rotação direita possa ocorrer é que o referido filho esquerdo exista (i.e., ele não pode ser nulo).

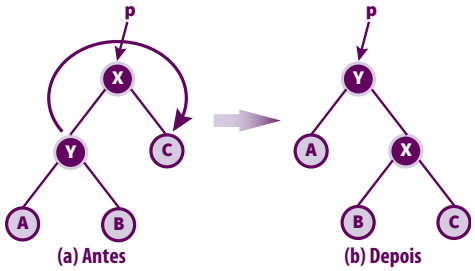


Figura 4–22: Rotação Direita em Árvore Binária de Busca

Na rotação ilustrada na **Figura 4–22**, diz-se, informalmente, que *ocorre uma rotação (à direita) do nó Y em torno do nó X* ou, simplesmente, *uma rotação sobre X*. Nesse contexto, o nó que sofre a rotação (nó Y na figura) é denominado **pivô** e seu pai (nó X na figura) é denominado **raiz**. Em outras palavras, de modo mais simples, pivô é o nó que sobe um nível e raiz é o nó que desce um nível.

Numa rotação direita, ocorre o seguinte:

1. A raiz passa a ocupar o lugar ora ocupado por seu filho direito. Na **Figura 4–22**, o nó rotulado com X ocupa o lugar do nó ocupado pelo nó rotulado com C.

[5] As letras no interior dos nós são rótulos usados para facilitar as referência aos nós aos quais essas letras estão associadas. Ou seja, essas letras *não são chaves*.

2. O filho direito do antigo filho esquerdo da raiz (na **Figura 4–22**, é o nó rotulado com *B*) torna-se o filho esquerdo da raiz.
3. O filho esquerdo da raiz ocupa o lugar dantes ocupado pela raiz. Na **Figura 4–22**, o nó rotulado com *Y* ocupa o lugar do nó ocupado pelo nó rotulado com *X*.
4. O ponteiro *p* que apontava para a antiga raiz (i.e., *X*) passa a apontar para a nova raiz (i.e., *Y*).

Numa rotação esquerda, o filho direito de um nó gira em torno do seu pai no sentido anti-horário, como mostra a **Figura 4–23**. O pré-requisito para que uma rotação esquerda possa ocorrer é que o referido filho direito exista (i.e., ele não pode ser nulo).

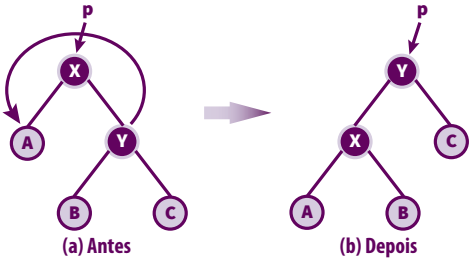


Figura 4–23: Rotação Esquerda em Árvore Binária de Busca

Na rotação ilustrada na **Figura 4–23**, diz-se, informalmente, que *ocorre uma rotação (à esquerda) do nó Y em torno do nó X* ou, simplesmente, *uma rotação sobre X*.

Numa rotação esquerda, acontece o seguinte:

1. A raiz ocupa o lugar do filho esquerdo. Na **Figura 4–23**, o nó *X* ocupa o lugar do nó *A*.
2. O filho esquerdo do filho direito da raiz (na **Figura 4–23**, é o nó *B*) torna-se o filho direito da raiz.
3. O filho direito da raiz ocupa o antigo lugar da raiz. Na **Figura 4–23**, nó rotulado com *Y* ocupa o lugar do nó outrora ocupado pelo nó rotulado com *X*.
4. O ponteiro *p* que apontava para a antiga raiz (*X*) passa a apontar para a nova raiz (*Y*).

Descritas em palavras, as rotações acima parecem ser complicadas. Mas, se você utilizar ilustrações gráficas como as últimas duas figuras, verá que elas são bastante simples de entender. Numa rotação direita, por exemplo, o filho esquerdo (i.e., *Y*) da raiz (i.e., *X*) gira de modo que seu pai se torna seu filho direito. Como um nó não pode ter dois filhos direitos, o antigo filho direito de *Y* (i.e., *B*) se torna filho esquerdo de *X*. E não poderia ser diferente, pois *X* já possui filho direito. Um raciocínio similar pode ser empregado para melhor entender a rotação esquerda.

Em representações gráficas de árvores que ilustram rotações, tipicamente, em vez de nós, frequentemente, utilizam-se triângulos para representar os filhos dos nós protagonistas das operações de rotação (esses nós estão representados com uma cor mais escura na **Figura 4–22** e na **Figura 4–23**). Esses triângulos representam subárvores das quais os nós protagonistas das rotações são raízes. Essas subárvores podem consistir de um único nó ou até ser vazias. Quando uma tal subárvore não é vazia, os níveis de seus nós podem aumentar ou diminuir. A **Figura 4–24** ilustra as rotações direita e esquerda utilizando essa notação.

Supondo que $chaves(A_i)$ seja o conjunto de chaves da subárvore A_i e $chave(n)$ seja a chave do nó n, tem-se que, na **Figura 4–24**, a seguinte relação é válida antes e depois de uma rotação esquerda ou direita:

$$chaves(A_1) < chave(y) < chaves(A_2) < chave(x) < chaves(A_3)$$

A **Figura 4–25** ilustra a preservação dessas propriedades durante uma rotação.

É importante observar que a ordem de visitação de nós num caminhamento infixo é preservada quando se faz uma rotação esquerda ou direita em torno de qualquer nó de uma árvore binária. Portanto uma árvore binária

de busca permanece sendo uma árvore de busca após qualquer operação de rotação esquerda ou direita em torno de um de seus nós (v. **Seção 4.1.1**).

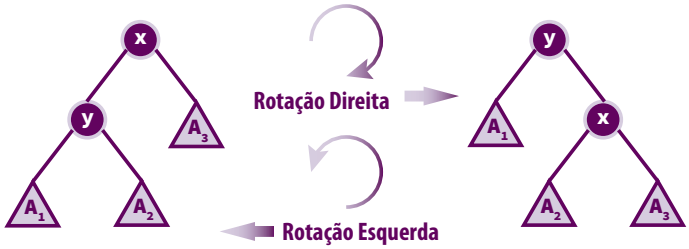


Figura 4–24: Rotações em Árvore Binária de Busca

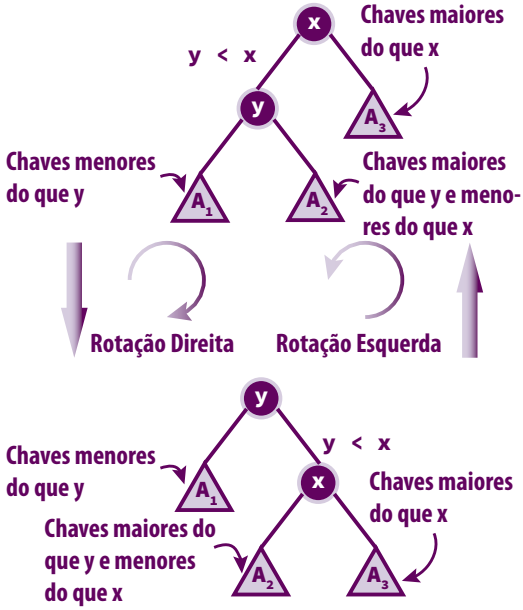


Figura 4–25: Preservação de Ordem em Rotações em Árvores Binárias de Busca

A função `RotacaoDireitaArvoreBB()`, apresentada a seguir, implementa a operação de rotação direita sobre a raiz de uma árvore binária. Nessa função, o parâmetro e a variável local são rotulados de acordo com a **Figura 4–22**, mas, diferentemente do que ocorre nessa figura, `x` e `y` são ponteiros para nós e não nós em si. Essa função retorna o endereço da raiz da árvore após a rotação. O tipo `tArvoreBB` é o mesmo definido na **Seção 4.1.2**.

```
static tArvoreBB RotacaoDireitaArvoreBB(tArvoreBB x)
{
   tArvoreBB y = x->esquerda; /* Guarda o endereço da nova raiz após a rotação*/

      /* Garante que é possível efetuar a rotação */
   ASSEGURA(y, "Rotacao direita impossivel: filho esquerdo e' nulo");

      /*************************************************/
      /* A ordem das instruções a seguir é essencial */
      /*************************************************/

      /* O filho direito de y torna-se filho esquerdo de x */
   x->esquerda = y->direita;

   y->direita = x; /* x torna-se filho direito de y */

   return y; /* Retorna a nova raiz */
}
```

A função `RotacaoEsquerdaArvoreBB()` implementa a operação de rotação esquerda sobre a raiz de uma árvore binária. Nessa função, `x` e `y` são ponteiros para os nós rotulados com `X` e `Y`, respectivamente, na **Figura 4–23**. Essa função retorna o endereço da nova raiz da árvore após a rotação.

```
static tArvoreBB RotacaoEsquerdaArvoreBB(tArvoreBB x)
{
   tArvoreBB y = x->direita; /* Guarda o endereço da nova raiz após a rotação */

      /* Garante que é possível efetuar a rotação */
   ASSEGURA(y, "Rotacao esquerda impossivel: filho direito e' nulo");

      /*********************************************/
      /* A ordem das instruções a seguir é essencial */
      /*********************************************/

      /* O filho esquerdo de y torna-se filho direito de x */
   x->direita = y->esquerda;

      /* x torna-se filho esquerdo de y */
   y->esquerda = x;

   return y; /* Retorna a nova raiz */
}
```

Note que, qualquer que seja o número de nós ou altura de uma árvore binária, uma rotação esquerda ou direita de um nó dessa árvore tem custo temporal $\theta(1)$, pois ela só envolve a alteração de alguns poucos ponteiros. Em árvores binárias que serão discutidas nas próximas seções, são comuns rotações duplas que são combinações das rotações básicas esquerda e direita discutidas nesta seção. Essas rotações duplas também possuem custos temporais $\theta(1)$ pelas mesmas razões expostas aqui.

Em geral, rotações alteram o formato de uma árvore binária de busca preservando suas propriedades. Elas diminuem a altura de uma árvore movendo subárvores de alturas menores para baixo e subárvores de alturas maiores para cima.

4.3 Balanceamento de Árvores Binárias de Busca

O **balanceamento** de uma árvore binária de busca é medido pela diferença entre as alturas de quaisquer duas de suas subárvores. Uma árvore binária **perfeitamente balanceada** é aquela na qual cada nó possui subárvores de mesma altura. Tal árvore possui número de nós igual a $2^a - 1$, em que a é a altura da árvore, conforme foi mostrado no **Apêndice B** do **Volume 1**. Mas esse critério de balanceamento é rígido demais para ser utilizado na prática e precisa ser relaxado. Quer dizer, nem sempre é viável manter uma árvore binária de busca com sua altura mínima, pois o custo computacional pode ser muito elevado.

O fato de uma árvore binária de busca ser mais eficiente do que uma lista encadeada depende do formato da árvore. Além disso, uma árvore inicialmente balanceada pode se tornar muito desbalanceada após experimentar muitas inserções e remoções. A **Figura 4–26** mostra três árvores binárias contendo a mesma informação, mas com configurações bem distintas. Como já foi discutido, uma árvore inclinada como aquela da **Figura 4–26 (a)** é praticamente uma lista encadeada. A árvore da **Figura 4–26 (b)** não é perfeitamente balanceada, mas apresenta um balanceamento aceitável. Por outro lado, a árvore da **Figura 4–26 (c)** não representa o pior caso de balanceamento, mas seu balanceamento não pode ser considerado aceitável.

Existem várias técnicas de balanceamento de árvores binárias de busca. Árvores com autobalanceamento reduzem suas alturas para valores próximos a $\log_2 n$ executando rotações quando chaves são inseridas ou removidas. O ônus associado a essas transformações é justificado pela aceleração das operações posteriores.

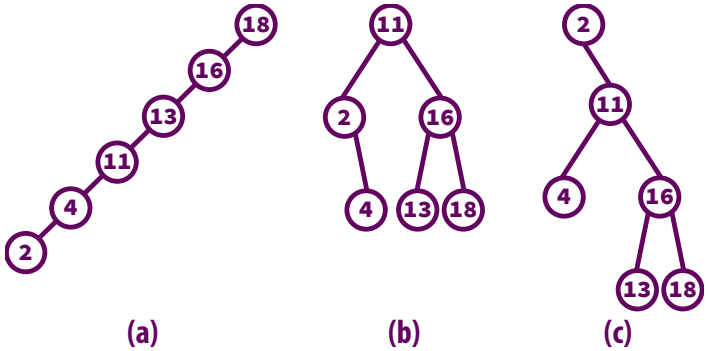


Figura 4–26: Árvores de Busca com Diferentes Balanceamentos

4.4 Árvores AVL

4.4.1 Motivação e Conceito

Uma **árvore AVL** é uma árvore binária de busca na qual as alturas de quaisquer duas subárvores nunca diferem em mais de *1*. A denominação *AVL* é uma homenagem a **A**delson-**V**elskii e **L**andis que demonstraram várias propriedades interessantes dessas árvores. Árvore AVL é também a árvore binária balanceada de busca mais antiga usada na prática, tendo sido inventada em 1962.

O custo temporal da maioria das operações com árvores binárias de busca é $\theta(a)$, em que a é a altura da árvore, que pode se tornar $\theta(n)$, em que n é o número de nós, quando a árvore é inclinada (v. **Figura 4–7**). Nesse último caso, uma árvore binária de busca se comporta como uma lista encadeada. Árvores AVL garantem que suas alturas são sempre $\theta(\log n)$ após operações de inserção ou remoção. Portanto operações sobre árvore AVL têm custo temporal $\theta(\log n)$.

Note que toda árvore perfeitamente balanceada também é uma árvore AVL, mas a recíproca não é verdadeira, pois árvores perfeitamente balanceadas têm critério de balanceamento mais rígido do que árvores AVL. Por exemplo, a árvore da **Figura 4–27 (a)** é AVL, mas não é perfeitamente balanceada, enquanto a árvore da **Figura 4–27 (b)** é AVL e também é perfeitamente balanceada.

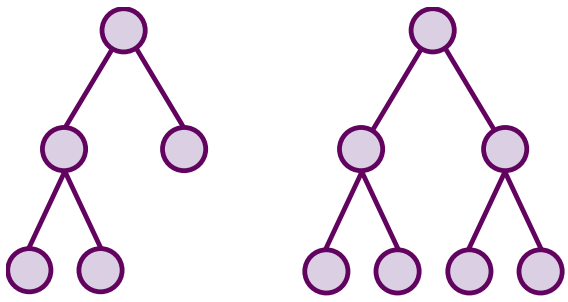


Figura 4–27: Árvore AVL e Árvore Perfeitamente Balanceada

O **balanceamento de um nó** é definido como a altura de sua subárvore esquerda menos a altura de sua subárvore direita. Portanto de acordo com a definição apresentada acima, cada nó de uma árvore AVL tem balanceamento -1, 0 ou 1, conforme sua subárvore esquerda tem altura menor, igual ou maior do que a altura de sua subárvore direita, respectivamente.

A **Figura 4–28** ilustra uma árvore AVL. Nessa figura, o número no interior de cada nó é o balanceamento do nó.

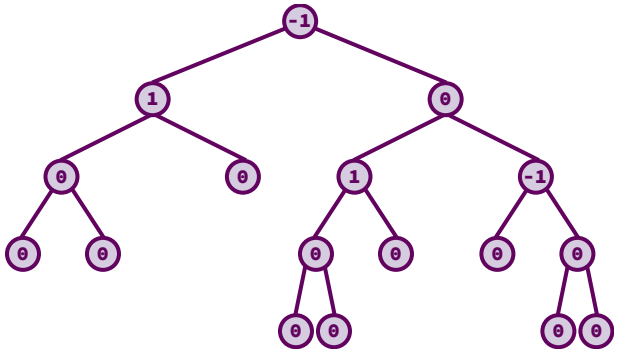


Figura 4–28: Balanceamento de Nós numa Árvore AVL

Uma consequência imediata da propriedade de balanceamento de árvores AVL é que qualquer subárvore de uma árvore AVL é em si uma árvore AVL. Essa propriedade de balanceamento também tem como importante consequência o fato de manter pequena a altura de qualquer árvore AVL, como mostra o **Teorema 4.3**.
Restabelecer o balanceamento de uma árvore perfeitamente balanceada após inserção ou remoção de um nó não é trivial, mas, no caso de árvores AVL, essa tarefa é relativamente fácil. Para facilitar as discussões sobre esse rebalanceamento as seguintes definições serão adotadas:

- **Caminho de inserção** numa árvore binária de busca são os nós visitados desde a raiz da árvore até a folha na qual será efetuada a inserção de um nó [v. **Figura 4–29 (a)**]. Por outro lado, **caminho inverso de inserção** é um caminho de inserção considerado no sentido oposto (i.e., do último nó do caminho de inserção até a raiz).
- **Caminho de remoção** numa árvore binária de busca são os nós visitados desde a raiz da árvore até um nó dessa árvore que será removido, se esse nó for encontrado, ou são os nós visitados desde a raiz da árvore até um nó nulo, quando o nó a ser removido não for encontrado [v. **Figura 4–29 (b)**]. **Caminho inverso de remoção** é um caminho de remoção considerado no sentido oposto.

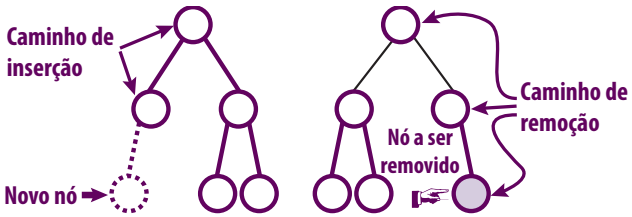


Figura 4–29: Caminho de Inserção e Caminho de Remoção

4.4.2 Desbalanceamento Devido a Inserção

Após um nó ser inserido, apenas os nós que estão no caminho de inserção têm suas subárvores alteradas. Além disso, apenas o primeiro nó de baixo para cima que se tornou desbalanceado no caminho inverso de inserção precisa ser rebalanceado (v. adiante).

A **Figura 4–30** a seguir ilustra todas as inserções possíveis que podem ser feitas na árvore da **Figura 4–28**. Nessa figura, cada inserção que resulta numa árvore balanceada é rotulada com B e cada inserção que resulta numa árvore desbalanceada é rotulada com D_i (sendo $1 \leq i \leq 12$). Note que a árvore torna-se desbalanceada apenas quando um nó é inserido na subárvore esquerda de um nó que tinha balanceamento igual a 1 ou quando um nó é inserido na subárvore direita de um nó que tinha balanceamento igual a -1. Na **Figura 4–30**, os nós com fundos coloridos são os nós mais profundos que se tornam desbalanceados em consequência de alguma inserção abaixo deles.

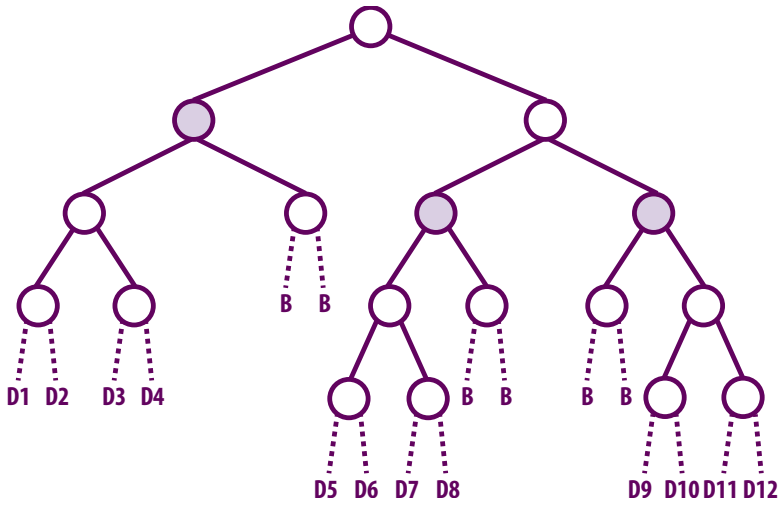


Figura 4–30: Possíveis Inserções numa Árvore AVL

Para manter uma árvore AVL balanceada após uma inserção é necessário transformar a árvore, de tal modo que:

- O caminhamento em ordem infixa na árvore resultante seja igual ao da árvore original (i.e., a árvore resultante deve continuar sendo uma árvore de busca)
- A árvore resultante permaneça balanceada segundo o critério AVL

Os possíveis desbalanceamentos que podem ocorrer em operações de inserção e remoção são resumidos a quatro casos. Suponha que *z* seja o primeiro ancestral de um nó *w* recém inserido que se torna desbalanceado em virtude dessa inserção. Suponha ainda que *y* seja o filho de *z* que se encontra no caminho entre *w* e *z* e que *x* é o primeiro neto de *z* encontrado nesse mesmo caminho. Esses quatro casos de desbalanceamento mencionados serão descritos a seguir.

Caso 1: Esquerda-esquerda

Esse tipo de desbalanceamento é decorrente de uma inserção na subárvore esquerda (enraizada em *x*) de um nó (*y*) que é filho esquerdo de outro nó (*z*) cujo balanceamento é igual a *1* (antes da inserção). Nesse caso, *y* é filho esquerdo de *z* e *x* é filho esquerdo de *y*. O rebalanceamento requerido nesse caso é uma simples rotação à direita como mostra a **Figura 4–31**. Nessa figura, os triângulos representam subárvores e o desbalanceamento nesse caso ocorre quando há uma inserção na subárvore A_1 ou na subárvore A_2. A seta para a esquerda nessa figura indica o nó sobre o qual incide a rotação (i.e., o nó *z*)[6].

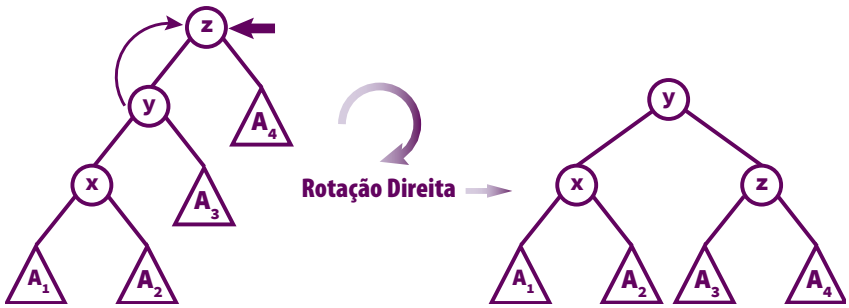


Figura 4–31: Correção de Desbalanceamento Esquerda-esquerda em Árvore AVL

Para tornar a discussão mais palpável, a **Figura 4–32** mostra de modo mais concreto esse caso de desbalanceamento, enquanto a **Figura 4–33** apresenta a correção efetuada para corrigir esse desbalanceamento.

[6] Lembre-se que *x*, *y* e *z* são rótulos associados aos nós para facilitar a discussão (i.e., não representam chaves).

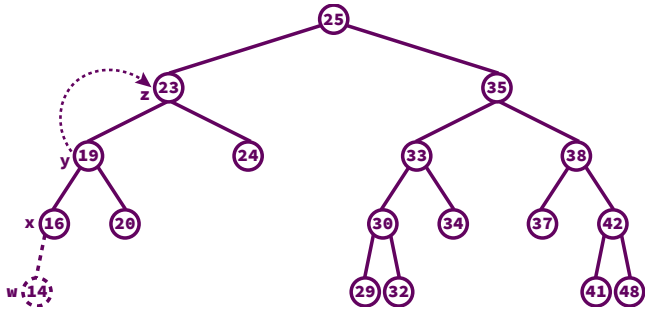


Figura 4–32: Exemplo de Desbalanceamento Esquerda-esquerda em Árvore AVL

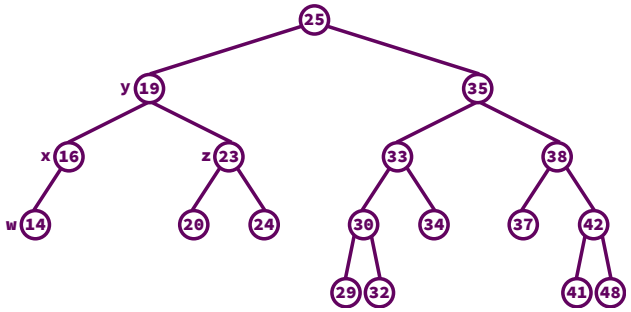


Figura 4–33: Exemplo de Correção de Desbalanceamento Esquerda-esquerda

Caso 2: Esquerda-direita

Esse tipo de desbalanceamento é ocasionado por uma inserção na subárvore direita (enraizada em x) de um nó (y) que é filho esquerdo de outro nó (z) cujo balanceamento é igual a *1* (antes da inserção). Aqui, y é filho esquerdo de z e x é filho direito de y. A **Figura 4–34** mostra que o rebalanceamento neste caso requer duas rotações: uma rotação esquerda seguida de uma rotação direita. Nessa figura, a inserção ocorre na subárvore A_2 ou na subárvore A_3 e as setas mais escuras indicam os nós sobre os quais incidem as rotações.

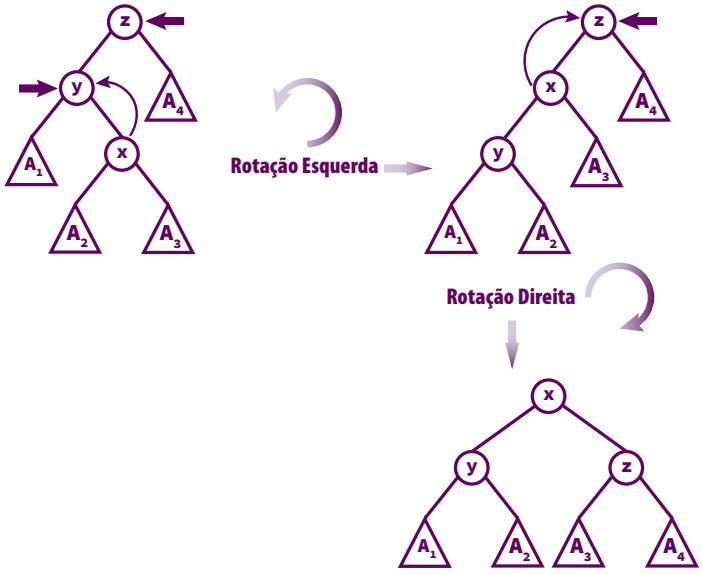


Figura 4–34: Correção de Desbalanceamento Esquerda-direita em Árvore AVL

A **Figura 4–35** mostra de modo mais concreto esse caso de desbalanceamento, ao passo que a **Figura 4–36** e a **Figura 4–37** apresentam as correções efetuadas para retificar esse desbalanceamento.

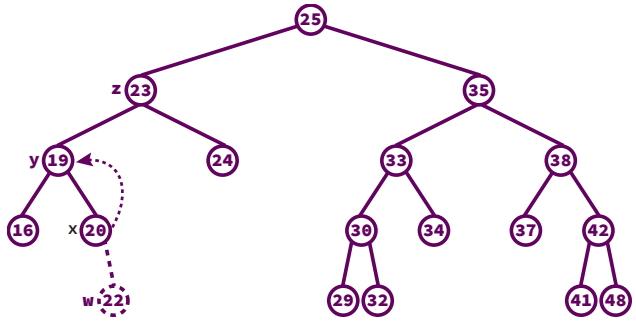


Figura 4–35: Exemplo de Desbalanceamento Esquerda-direita em Árvore AVL

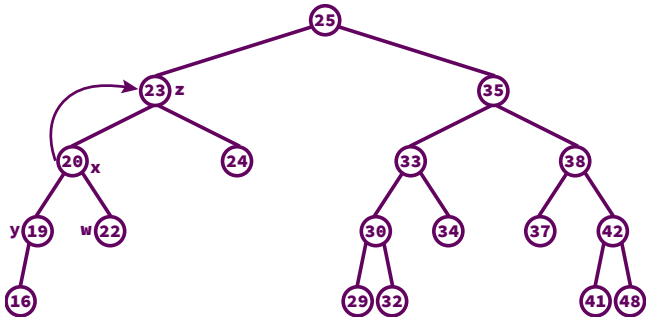


Figura 4–36: Exemplo de Correção de Desbalanceamento Esquerda-direita 1

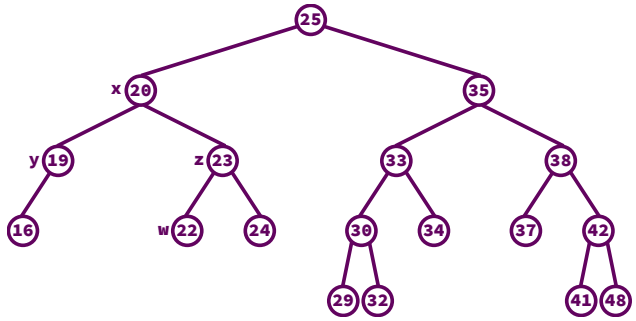


Figura 4–37: Exemplo de Correção de Desbalanceamento Esquerda-direita 2

Caso 3: Direita-direita

Esse tipo de desbalanceamento é ocasionado por uma inserção na subárvore direita (enraizada em x) de um nó (y) que é filho direito de outro nó (z) cujo balanceamento é igual a -1 (antes da inserção). Nesse caso, y é filho direito de z e x é filho direito de y. Aqui, o rebalanceamento requerido é uma simples rotação à esquerda, como mostra a **Figura 4–38**. Nessa figura, a inserção ocorre na subárvore A_3 ou A_4.

A **Figura 4–39** mostra de modo mais concreto esse caso de desbalanceamento, enquanto a **Figura 4–40** apresenta a correção efetuada para corrigir esse desbalanceamento.

Note que o **Caso 3** é simétrico ao **Caso 1**. Quer dizer, a descrição do **Caso 3** pode ser obtida da descrição do **Caso 1** trocando-se entre si as palavras *direita* e *esquerda*. E o mesmo é o caso das respectivas correções desses casos.

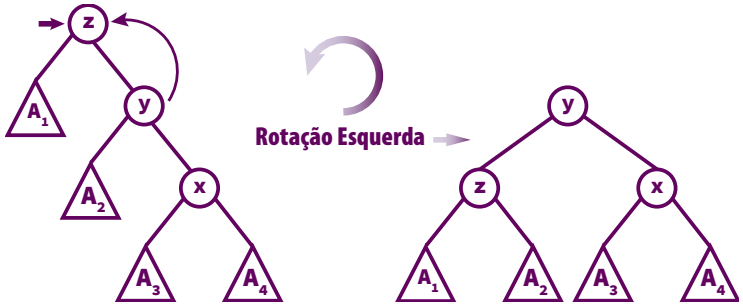


Figura 4–38: Correção de Desbalanceamento Direita-direita em Árvore AVL

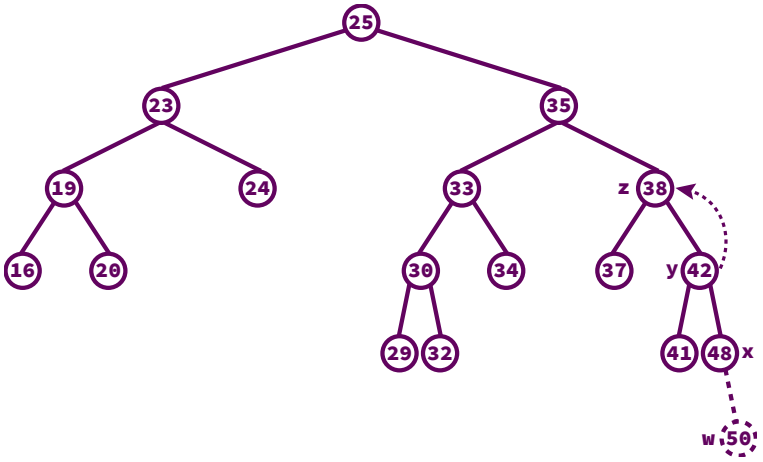


Figura 4–39: Exemplo de Desbalanceamento Direita-direita em Árvore AVL

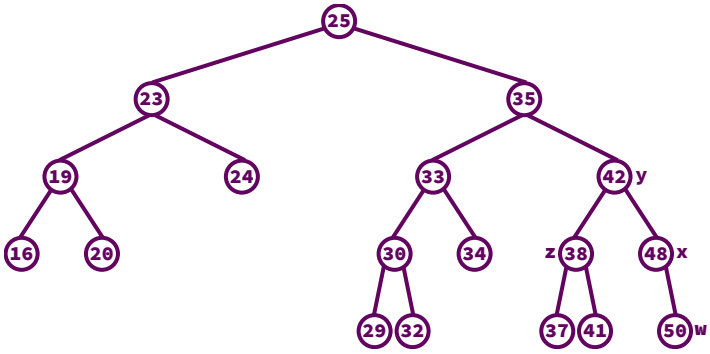


Figura 4–40: Exemplo de Correção de Desbalanceamento Direita-direita

Caso 4: Direita-esquerda

Este tipo de desbalanceamento decorre de uma inserção na subárvore esquerda (enraizada em x) de um nó (y) que é filho direito de outro nó (z) cujo balanceamento é igual a -1 (antes da inserção). Aqui, y é filho direito de z e x é filho esquerdo. Nesse caso, a correção necessária consiste numa rotação direita seguida de uma rotação esquerda. Ambas as rotações são efetuadas sobre o nó z. A **Figura 4–41** ilustra o rebalanceamento necessário nesse caso. As setas pretas nessa figura indicam os nós sobre os quais incidem as rotações. Nessa mesma figura, a inserção ocorre na subárvore A_2 ou A_3.

A **Figura 4–42** mostra de modo mais concreto esse caso de desbalanceamento, enquanto a **Figura 4–43** e a **Figura 4–44** apresentam as correções efetuadas para retificar esse desbalanceamento. Note que o **Caso 4** é simétrico ao **Caso 2**.

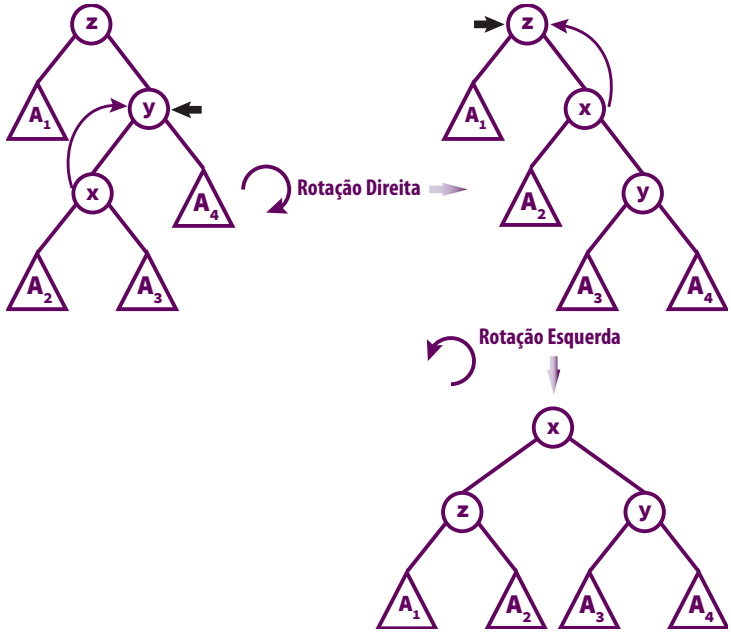


Figura 4–41: Correção de Desbalanceamento Direita-esquerda em Árvore AVL

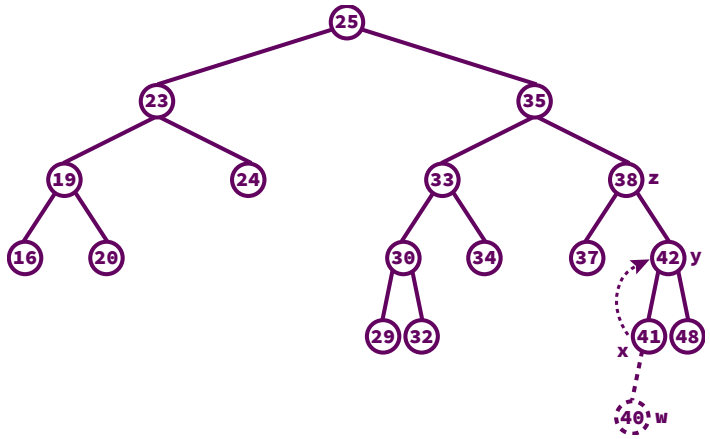


Figura 4–42: Exemplo de Desbalanceamento Direita-esquerda em Árvore AVL

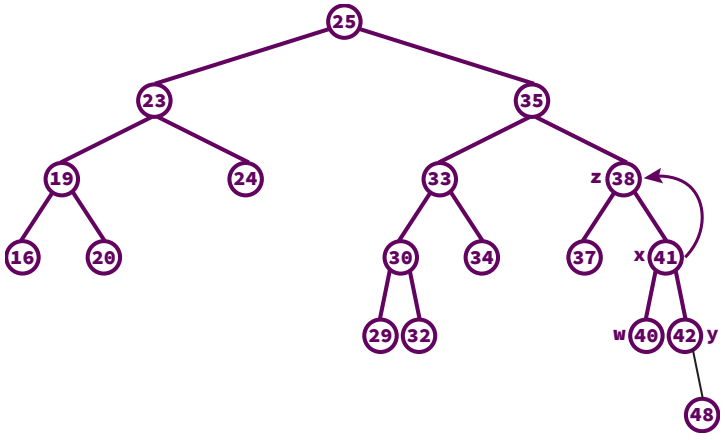


Figura 4–43: Exemplo de Correção de Desbalanceamento Direita-esquerda 1

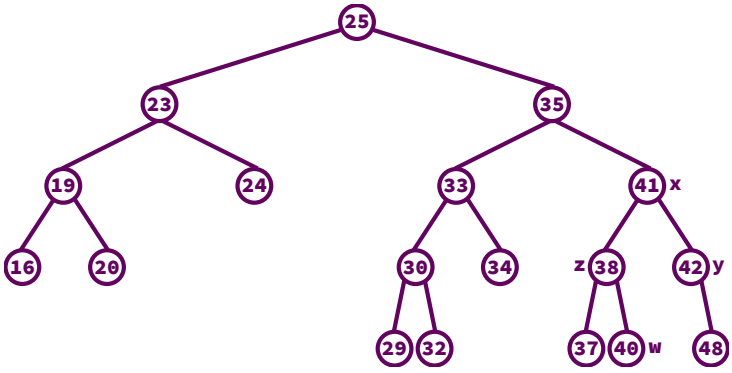


Figura 4–44: Exemplo de Correção de Desbalanceamento Direita-esquerda 2

Em resumo, o procedimento a ser adotado para corrigir um desbalanceamento em virtude de inserção é o seguinte:

1. Se o balanceamento[7] de um ancestral for maior do que *1*, então tem-se um desbalanceamento esquerda-esquerda (**Caso 1**) ou esquerda-direita (**Caso 2**). Para verificar qual deles se aplica, compara-se a chave do nó recém inserido com a chave do filho esquerdo do ancestral ora escrutinado. Se a chave recém-inserida for menor do que a chave do referido filho, trata-se do **Caso 1** (i.e., esquerda-esquerda) de desbalanceamento; caso contrário, trata-se do **Caso 2** (i.e., esquerda-direita).
2. Se o balanceamento de um ancestral for menor do que *–1*, então tem-se um desbalanceamento direita-direita (**Caso 3**) ou direita-esquerda (**Caso 4**). Para decidir qual é o caso correto, compara-se a chave do nó recém inserido com a chave do filho direito do ancestral sob escrutínio. Se a chave recém-inserida for menor do que a chave do referido filho, trata-se do **Caso 4** (i.e., direita-esquerda) de desbalanceamento; caso contrário, trata-se do **Caso 3** (i.e., direita-direita).

O algoritmo de inserção em árvore AVL é apresentado na **Figura 4–45**.

Algoritmo InsereEmÁrvoreAVL

Entrada: O conteúdo de um novo elemento (nó)

Entrada/Saída: Uma árvore AVL

Saída: Um valor informando se a operação foi bem-sucedida

1. Se a árvore estiver vazia, faça com que o novo nó seja a raiz da árvore e retorne informando o sucesso da operação
2. Se a chave do novo nó for igual à chave da raiz, retorne informando o fracasso da operação
3. Se na chave do novo nó for menor do que a chave da raiz, insira-o na subárvore esquerda usando **InsereEmÁrvoreAVL**
4. Se na chave do novo nó for maior do que a chave da raiz, insira-o na subárvore direita usando **InsereEmÁrvoreAVL**
5. Atualize a altura da árvore
6. Obtenha o balanceamento (*bal*) da árvore
7. Se *bal* > *1*, faça:
   - **7.1** Se a chave recém-inserida for menor do que a chave da raiz da subárvore esquerda, efetue uma rotação direita na árvore e retorne informando o sucesso da operação
   - **7.2** Se a chave recém-inserida for maior do que a chave da raiz da subárvore esquerda:

Continua

Figura 4–45: Algoritmo de Inserção em Árvore AVL

[7] Lembrando: o balanceamento de um nó é a altura de sua subárvore esquerda menos a altura de sua subárvore direita.

Algoritmo InsereEmÁrvoreAVL (Continuação) — Continuação

7.2.1 Efetue uma rotação esquerda na subárvore esquerda
7.2.2 Efetue uma rotação direita na árvore
7.2.3 Retorne informando o sucesso da operação

8. Se *bal* < *–1*, faça:
8.1 Se a chave recém-inserida for maior do que a chave da raiz da subárvore direita, efetue uma rotação esquerda na árvore e retorne informando o sucesso da operação
8.2 Se a chave recém-inserida for menor do que a chave da raiz da subárvore direita:
8.2.1 Efetue uma rotação direita na subárvore direita
8.2.2 Efetue uma rotação esquerda na árvore
8.2.3 Retorne informando o sucesso da operação

Figura 4–45 (Cont.): Algoritmo de Inserção em Árvore AVL

4.4.3 Desbalanceamento Causado por Remoção

A análise dos casos de desbalanceamentos de árvores AVL em virtude de remoção é semelhante à análise correspondente para inserção, mas as correções desses desbalanceamentos são bem mais complicadas porque um dado desbalanceamento pode se propagar no caminho inverso de remoção até a raiz da árvore. Do mesmo modo que inserção, há quatro casos possíveis de desbalanceamento que podem ocorrer após a remoção de um nó. Entretanto, diferentemente do que ocorre com inserção, não é suficiente corrigir o balanceamento do primeiro nó que era ancestral do nó removido que se torna desbalanceado em virtude dessa remoção. Quer dizer, após corrigir esse balanceamento pode ser necessário ainda corrigir nós que sejam ancestrais do nó recém-corrigido. Os casos de desbalanceamento devido a remoção são semelhantes àqueles causados por inserção, mas as causas são (obviamente) diferentes. Esses casos serão resumidos a seguir.

Esquerda-esquerda

Esse desbalanceamento é igual ao desbalanceamento de idêntica denominação que ocorre durante inserção, mas a causa aqui é uma remoção numa subárvore direita (em vez de uma inserção numa subárvore esquerda). Uma rotação direita em torno do nó que se torna desbalanceado é suficiente para corrigir esse desbalanceamento, como ilustra a **Figura 4–46**. Nessa figura, a remoção do nó *w*, que se encontra na subárvore direita de *z*, faz com que esse último nó se torne desbalanceado. Essa figura mostra ainda a devida correção.

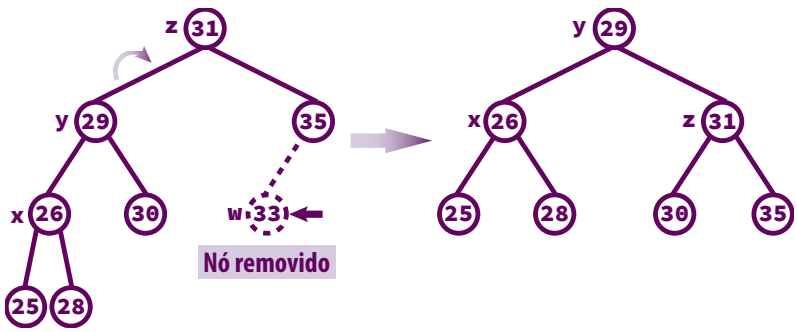


Figura 4–46: Exemplo de Desbalanceamento Esquerda-esquerda após Remoção

Esquerda-direita

Esse desbalanceamento é idêntico ao desbalanceamento de mesma denominação que ocorre durante inserção, mas a causa aqui é (novamente) devido a uma remoção numa subárvore direita. Assim como ocorre no

desbalanceamento idêntico que ocorre durante uma inserção, uma rotação dupla corrige o problema, como mostra a **Figura 4–47**.

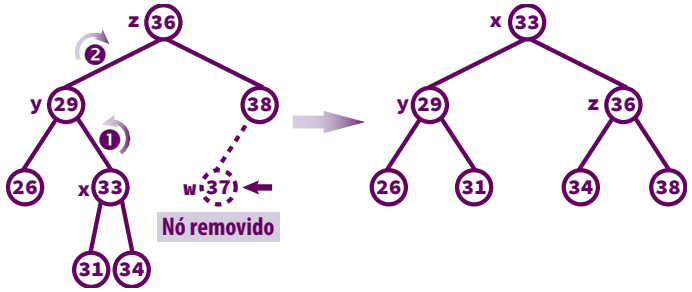


Figura 4–47: Exemplo de Desbalanceamento Esquerda-direita após Remoção

Direita-direita

Esse desbalanceamento é idêntico ao desbalanceamento de mesma denominação que ocorre durante inserção, porém a causa aqui é uma remoção numa subárvore esquerda. A **Figura 4–48** mostra um exemplo desse caso de desbalanceamento e sua devida correção.

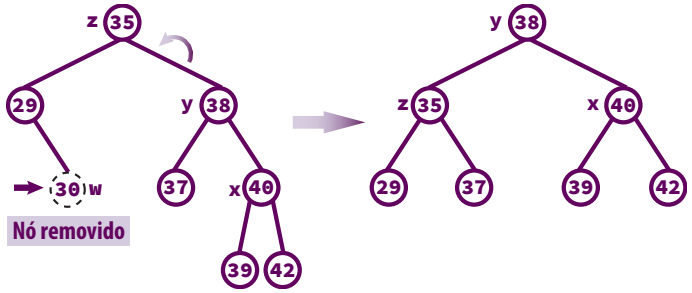


Figura 4–48: Exemplo de Desbalanceamento Direita-direita após Remoção

Direita-esquerda

Esse desbalanceamento é idêntico ao desbalanceamento de mesma denominação que ocorre durante inserção, contudo a causa aqui é uma remoção numa subárvore esquerda. A **Figura 4–49** apresenta um exemplo desse caso de desbalanceamento e sua respectiva correção.

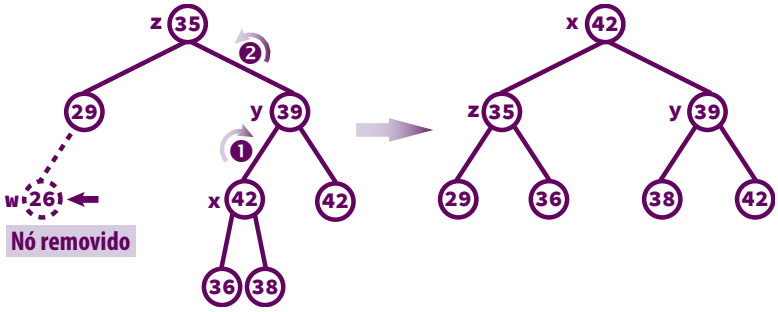


Figura 4–49: Exemplo de Desbalanceamento Direita-esquerda após Remoção

Propagação de Desbalanceamento

O rebalanceamento de uma árvore AVL após a remoção de um nó é um tanto mais complicado do que após inserção, mas continua envolvendo combinações de rotações (às vezes, múltiplas) esquerdas e direitas de porções da árvore. O algoritmo básico de remoção de nós é aquele apresentado na **Seção 4.1.2**, mas esse algoritmo é

acrescido com porções de código que realizam o rebalanceamento da árvore. Contudo, assim como ocorre com inserção, nem sempre o rebalanceamento de uma árvore AVL se faz necessário após uma operação de remoção.

Para entender a razão pela qual reajustes múltiplos podem ser necessários, considere o exemplo de remoção apresentado na **Figura 4–50**.

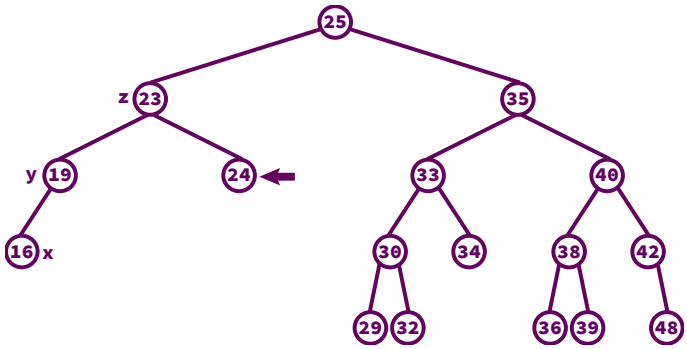


FIGURA 4–50: REMOÇÃO DE NÓ EM ÁRVORE AVL COM DESBALANCEAMENTO

Quando o nó indicado pela seta na **Figura 4–50** é removido, o nó rotulado como *z* torna-se desbalanceado e, de fato, esse é o único nó que se torna desbalanceado em virtude da referida remoção. Conforme se pode constatar nessa figura, esse é um desbalanceamento do tipo esquerda-esquerda (**Caso 1**) que pode ser corrigido com uma simples rotação direita sobre o nó *z*. A remoção do nó em questão e a subsequente correção do desbalanceamento mencionado são mostrados na **Figura 4–51**.

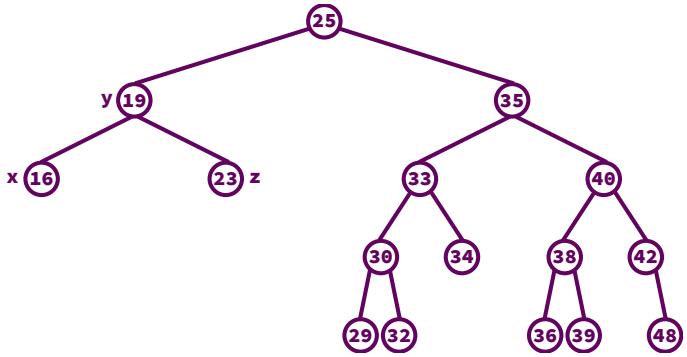


FIGURA 4–51: CORREÇÃO PARCIAL APÓS REMOÇÃO DE NÓ EM ÁRVORE AVL

O problema é que o rebalanceamento mostrado na **Figura 4–51** faz com que o nó contendo a chave *25* (i.e., a raiz da árvore) se torne desbalanceado. Esse novo desbalanceamento é do tipo direita-direita e é corrigido com uma única rotação esquerda em torno do nó contendo a chave *25*, o que resulta na **Figura 4–52**.

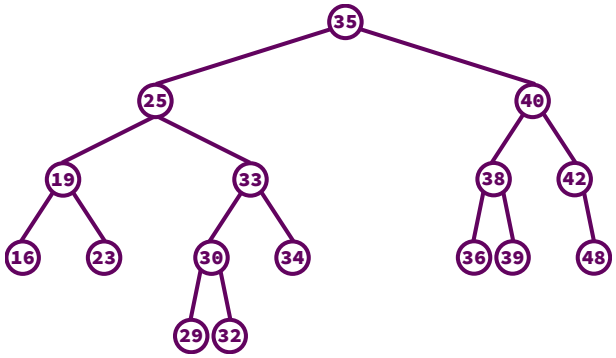


FIGURA 4–52: CORREÇÃO COMPLETA APÓS REMOÇÃO DE NÓ EM ÁRVORE AVL

O que esse último exemplo mostra é que após o balanceamento de um nó, pode ser necessário rebalancear alguns de seus ancestrais. Assim é necessário seguir o caminho desde o último nó rebalanceado até a raiz da árvore para checar se esses ancestrais se tornaram desbalanceados e, se for o caso, proceder com a devida correção.

O algoritmo de remoção em árvore AVL é apresentado na **Figura 4–53**.

ALGORITMO REMOVEEMÁRVOREAVL

ENTRADA: A chave do nó a ser removido
ENTRADA/SAÍDA: Uma árvore AVL
SAÍDA: Um valor informando o sucesso da operação

1. Se a árvore estiver vazia, retorne informando que não houve remoção
2. Se a chave a ser removida for menor do que a chave da raiz, remova-a da subárvore esquerda usando **REMOVEEMÁRVOREAVL**
3. Caso contrário, se a chave a ser removida for maior do que a chave da raiz, remova-a da subárvore direita usando **REMOVEEMÁRVOREAVL**
4. Caso contrário, faça:
   - **4.1** Se o nó a ser removido for uma folha, remova-a
   - **4.2** Se o nó a ser removido possuir apenas um filho, substitua o referido nó por seu filho
   - **4.3** Se o nó a ser removido possuir dois filhos:
     - **4.3.1** Substitua o conteúdo do nó que seria removido pelo conteúdo de seu sucessor
     - **4.3.2** Remova o nó sucessor usando **REMOVEEMÁRVOREAVL**
5. Se a árvore ficou vazia, retorne informando o sucesso da remoção
6. Atualize a altura da árvore
7. Obtenha o balanceamento (*bal*) da árvore
8. Se $-1 \leq bal \leq 1$, retorne informando o sucesso da operação
9. Se $bal > 1$ e o balanceamento da subárvore esquerda for maior do que ou igual a 0, efetue uma rotação direita na árvore e retorne informando o sucesso da operação
10. Se $bal > 1$ e o balanceamento da subárvore esquerda for menor do que 0:
    - **10.1** Efetue uma rotação esquerda na subárvore esquerda
    - **10.2** Efetue uma rotação direita na árvore
    - **10.3** Retorne informando o sucesso da operação
11. Se $bal < -1$ e o balanceamento da subárvore direita for menor do que ou igual a 0, efetue uma rotação esquerda na árvore e retorne informando o sucesso da operação
12. Se $bal < -1$ e o balanceamento da subárvore direita for maior do que 0:
    - **12.1** Efetue uma rotação direita na subárvore direita
    - **12.2** Efetue uma rotação esquerda na árvore
    - **12.3** Retorne informando o sucesso da operação

FIGURA 4–53: ALGORITMO DE REMOÇÃO EM ÁRVORE AVL

4.4.4 Implementação

Definições de Tipos

Para facilitar inserções e remoções em árvores AVL, um novo campo é adicionado na definição de tipo do nó da árvore. Esse novo campo armazena o valor do balanceamento do nó (`-1`, `0` ou `1`). A definição do tipo de nó de uma árvore AVL é apresentada a seguir:

```
typedef struct rotNoAVL {
          struct rotNoAVL *esquerda, *direita; /* Filhos deste nó */
          tCEP_Ind         conteudo; /* Par chave/índice */
          int              altura; /* Altura da árvore que tem este nó como raiz */
       } tNoAVL, *tArvoreAVL;
```

Busca

Busca em árvore AVL é exatamente igual a busca em árvore binária ordinária de busca (v. **Seção 4.1.2**).

Inserção

A função `InsereAVL()`, apresentada a seguir, insere um novo nó numa árvore AVL. Essa função retorna o endereço da raiz da árvore se a inserção ocorrer, ou **NULL** se não houver inserção porque a chave já se encontra na árvore. Os parâmetros dessa função são:

- `arvore` (entrada) — ponteiro para a raiz da árvore na qual será feita a inserção
- `*conteudo` (entrada) — conteúdo do nó que será inserido

```
tArvoreAVL InsereAVL(tArvoreAVL arvore, const tCEP_Ind *conteudo)
{
   int bal,     /* Balanceamento de um nó */
       compara; /* Comparação de duas chaves */

      /* Se a árvore estiver vazia, o novo nó será a raiz da árvore */
   if (!arvore)
      return NovoNoAVL(conteudo);

      /* Compara a chave do nó a ser inserido com a chave do nó corrente */
   compara = strcmp(conteudo->chave, arvore->conteudo.chave);

   if (compara < 0)
      arvore->esquerda = InsereAVL(arvore->esquerda, conteudo);
   else if (compara > 0)
      arvore->direita = InsereAVL(arvore->direita, conteudo);
   else
      return NULL; /* Chave já existe */

      /* Atualiza a altura desta arvore */
   arvore->altura = MAIOR(AlturaAVL(arvore->esquerda), AlturaAVL(arvore->direita))+1;

   bal = BalanceamentoAVL(arvore); /* Obtém o balanceamento da árvore */

      /* Examina os 4 casos possíveis de desbalanceamento */

   if (bal > 1) {
      /* Desbalanceamento esquerda-esquerda ou esquerda-direita */

      compara = strcmp(conteudo->chave, arvore->esquerda->conteudo.chave);

      if (compara < 0) /* Caso esquerda-esquerda */
         return RotacaoDireitaAVL(arvore);

      if (compara > 0) { /* Caso esquerda-direita */
         arvore->esquerda = RotacaoEsquerdaAVL(arvore->esquerda);
         return RotacaoDireitaAVL(arvore);
      }
   }

   if (bal < -1) {
      /* Desbalanceamento direita-direita ou direita-esquerda */

      compara = strcmp(conteudo->chave, arvore->direita->conteudo.chave);
```

```
      if (compara > 0) /* Caso direita-direita */
         return RotacaoEsquerdaAVL(arvore);

      if (compara < 0) { /* Caso direita-esquerda */
         arvore->direita = RotacaoDireitaAVL(arvore->direita);
         return RotacaoEsquerdaAVL(arvore);
      }
   }
   return arvore; /* Não ocorreu desbalanceamento */
}
```

A macro `MAIOR` invocada por `InsereAVL()` resulta no maior de dois números e sua implementação é trivial. Por sua vez, a função `BalanceamentoAVL()`, que é chamada por `InsereAVL()` para calcular o balanceamento de um nó, é implementada como:

```
static int BalanceamentoAVL(tArvoreAVL pNo)
{
   return pNo ? AlturaAVL(pNo->esquerda) - AlturaAVL(pNo->direita) : 0;
}
```

A função `AlturaAVL()` chamada por `InsereAVL()` e `BalanceamentoAVL()` para calcular a altura de uma árvore AVL é trivial, pois simplesmente não há o que calcular visto que a altura de cada nó é armazenada no próprio nó. Assim a função `AlturaAVL()` é implementada como:

```
int AlturaAVL(tArvoreAVL arvore)
{
   return arvore ? arvore->altura : 0;
}
```

Remoção

A implementação de uma função iterativa para remoção em árvores AVL que leve em consideração o que foi exposto é bastante complicada. Portanto novamente, será dada preferência a uma implementação recursiva que é relativamente mais fácil de entender, implementar e, ao mesmo tempo, é menos sujeita a erros de programação. Usando uma implementação recursiva da função que efetua a remoção de um nó de uma árvore AVL, todos os endereços dos nós ancestrais do nó removido são armazenados na pilha de execução e podem ser acessados durante a fase de decréscimo da função. Em caso de dúvidas na implementação dessa função, sugere-se que o leitor consulte o **Capítulo 4** do **Volume 1** desta obra.

A função `RemoveAVL()`, apresentada adiante, remove um nó de uma árvore AVL. Essa função retorna o endereço da raiz da árvore após a conclusão da operação, se a remoção for bem-sucedida ou **NULL**, se a árvore estiver vazia, se ela se tornar vazia após a remoção ou se a chave não for encontrada. Os parâmetros dessa função são:

- `arvore` (entrada) — ponteiro que representa a árvore na qual será feita a remoção
- `chave` (entrada) — a chave do nó a ser removido

```
tArvoreAVL RemoveAVL(tArvoreAVL arvore, tCEP chave)
{
   int        bal,     /* BalanceamentoAVL de um nó    */
              compara; /* Comparação de duas chaves */
   tArvoreAVL p;       /* Ponteiro auxiliar */

   if (!arvore)  /* Se a árvore estiver vazia, não há remoção */
      return NULL;

      /* Compara a chave do nó a ser removido com a chave do nó corrente */
   compara = strcmp(chave, arvore->conteudo.chave);
```

```
if ( compara < 0 )
      /* O nó a ser removido se encontra na subárvore esquerda */
   arvore->esquerda = RemoveAVL(arvore->esquerda, chave);
else if( compara > 0 )
      /* O nó a ser removido se encontra na subárvore direita */
   arvore->direita = RemoveAVL(arvore->direita, chave);
else {
   /* Chave encontrada. Este é o nó a ser removido */

      /* Se o nó possui apenas um filho ou não possui nenhum */
      /* filho, p apontará para o nó a ser removido          */
   if( (!arvore->esquerda) || (!arvore->direita) ) {
      p = arvore->esquerda ? arvore->esquerda : arvore->direita;

      if(!p) { /* Nó não possui filhos */
         p = arvore;
         arvore = NULL;
      } else /* Nó possui um único filho */
            /* Copia o conteúdo do filho único para o pai */
         *arvore = *p;

      free(p); /* Libera o devido nó */
   } else { /* Nó possui dois filhos */
         /* Obtém o sucessor em ordem infixa do nó a ser removido */
      p = MenorNoAVL(arvore->direita);

         /* Copia o conteúdo do sucessor para o nó que seria removido */
      arvore->conteudo = p->conteudo;

         /* Remove o referido sucessor */
      arvore->direita = RemoveAVL(arvore->direita, p->conteudo.chave);
   }
}

   /* Se a árvore só tinha um nó e ele foi removido, o serviço está completo */
if (!arvore)
  return arvore;

   /* Atualiza a altura do nó corrente */
arvore->altura = MAIOR(AlturaAVL(arvore->esquerda),
                       AlturaAVL(arvore->direita)) + 1;

   /* Obtém o balanceamento desta árvore para  */
   /* verificar se ela se tornou desbalanceada */
bal = BalanceamentoAVL(arvore);

   /* Examina os 4 casos possíveis de desbalanceamento */

   /* Caso esquerda-esquerda */
if (bal > 1 && BalanceamentoAVL(arvore->esquerda) >= 0)
   return RotacaoDireitaAVL(arvore);

   /* Caso esquerda-direita */
if (bal > 1 && BalanceamentoAVL(arvore->esquerda) < 0) {
   arvore->esquerda =  RotacaoEsquerdaAVL(arvore->esquerda);
   return RotacaoDireitaAVL(arvore);
}

   /* Caso direita-direita */
if (bal < -1 && BalanceamentoAVL(arvore->direita) <= 0)
   return RotacaoEsquerdaAVL(arvore);
```

```
      /* Caso direita-esquerda */
   if (bal < -1 && BalanceamentoAVL(arvore->direita) > 0) {
      arvore->direita = RotacaoDireitaAVL(arvore->direita);
      return RotacaoEsquerdaAVL(arvore);
   }
   return arvore; /* Não ocorreu desbalanceamento */
}
```

4.4.5 Análise

Teorema 4.3: A altura de uma árvore AVL contendo n elementos é $\theta(\log n)$.

Prova: Suponha que n_a seja o número mínimo de nós de uma árvore AVL com altura a. Então, de acordo com a definição de árvore AVL, o valor de n_a é determinado pela seguinte relação de recorrência:

$$n_a = n_{a-1} + n_{a-2} + 1$$

sendo que $n_0 = 0$ e $n_1 = 1$ são as condições iniciais dessa relação de recorrência.

Resolvendo-se essa relação de recorrência[8], obtêm-se os seguintes limites para a altura a de uma árvore AVL em função de seu número de nós n:

$$\log(n+1) \leq a < 1{,}44 \cdot \log(n+2) - 0{,}328$$

Esse resultado significa que a altura a de uma árvore AVL contendo n nós é $\theta(\log n)$. ■

Teorema 4.4: No pior caso, uma operação de busca, inserção ou remoção em árvore AVL tem custo temporal $\theta(\log n)$.

Prova: Qualquer operação básica de tabela de busca implementada como árvore AVL (i.e., busca, inserção e remoção) tem, no pior caso, custo temporal $\theta(a)$, em que a é a altura da árvore. Mas, como foi provado na **Seção 4.4.1** que a é $\theta(\log n)$, tem-se que cada uma dessas operações básicas tem custo temporal $\theta(\log n)$. No caso de inserção e remoção, esse custo, por enquanto, é parcial, pois faltou levar em consideração o rebalanceamento da árvore.

Conforme foi visto na **Seção 4.2**, operações de rotação apresentam custo temporal $\theta(1)$, pois uma rotação altera apenas alguns ponteiros. Os cálculos de balanceamento após operações de inserção e remoção também têm custo temporal $\theta(1)$, pois eles só envolvem operações aritméticas elementares. No pior caso, os ajustes de balanceamento numa árvore AVL devido a operações de inserção e de remoção acrescentam um fator $\theta(\log n)$ aos custos temporais dessas operações. Portanto pela regra da soma da análise assintótica, tem-se que o custo temporal de cada operação básica é $\theta(\log n)$. ■

Conforme foi visto no **Capítulo 12** do **Volume 1**, a altura mínima de uma árvore binária é $\lfloor \log_2 n + 1 \rfloor$. Agora, como mostra a **Figura 4–7**, quando chaves ordenadas são inseridas numa árvore ordinária de busca, a árvore obtém sua altura máxima que corresponde exatamente ao número de nós da árvore. A diferença entre uma árvore com altura máxima e uma árvore com altura mínima pode ser gigantesca. Por exemplo, uma árvore binária com altura máxima *1.000.000* tem altura mínima igual a *20* (que é igual a $\lfloor \log_2 1000000 + 1 \rfloor$).

Devido ao uso de um campo adicional que armazena a altura em cada nó, árvores AVL consomem mais espaço do que árvores binárias ordinárias de busca. Mesmo assim, os dois tipos de árvores possuem custo espacial $\theta(n)$, visto que o espaço adicional não altera o custo espacial, pois toda implementação encadeada de tabela de busca vista até aqui tem custo espacial $\theta(n)$.

As operações de inserção e remoção exibem um custo adicional $\theta(\log n)$ devido ao uso de recursão. Esse custo espacial é desprezível na prática. Por exemplo, uma remoção numa árvore AVL contendo *1.000.000* requer o empilhamento de apenas cerca de *20* registros de ativação na pilha de execução.

---

[8] No **Apêndice B** do **Volume 1**, é apresentada uma breve introdução a relações de recorrência homogêneas, o que não é o caso aqui, pois essa relação de recorrência é heterogênea e sua resolução está além do escopo deste livro.

A **Tabela 4–1** resume os custos temporais das operações básicas em tabelas de busca implementadas como árvores AVL.

Operação	Custo da Operação	Custo do Balanceamento	Custo Total
Busca	$\theta(\log n)$	*Não há*	$\theta(\log n)$
Inserção	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$
Remoção	$\theta(\log n)$	$\theta(\log n)$	$\theta(\log n)$

Tabela 4–1: Custos Temporais de Operações com Árvores AVL

A **Tabela 4–2** resume as razões favoráveis e contrárias ao uso de árvores AVL

Prós	Contras
Todas as operações têm, no pior caso, custo temporal $\theta(\log n)$, pois árvores AVL estão sempre balanceadas	**Algoritmos de inserção e remoção são difíceis de implementar e depurar. Notadamente, se eles não forem recursivos**
As operações de rebalanceamento acrescentam apenas um fator constante aos custos das operações de inserção e remoção de nós	**O custo espacial é $\theta(n)$**

Tabela 4–2: Prós e Contras de Árvores AVL

4.5 Árvores Binárias Afuniladas

4.5.1 Conceitos

As seções anteriores mostraram que os custos temporais das operações básicas em árvores binárias de busca podem ser substancialmente melhorados por meio de balanceamento. Mas como será visto na presente seção, a eficiência dessas operações pode ser melhorada não apenas com balanceamento. A técnica a ser desenvolvida nesta seção baseia-se no pressuposto de que nem todos os elementos armazenados numa árvore de busca são acessados com a mesma frequência. Assim de acordo com essa técnica, elementos acessados com maior frequência são colocados próximos à raiz da árvore.

Uma **árvore afunilada** (*splay tree*, em inglês) é uma árvore binária de busca **autoajustável** no sentido de que cada operação de busca, inserção ou remoção provoca uma alteração em seu formato. A denominação *afunilada* é derivada do fato de, após uma dessas operações, um elemento ser movido para níveis mais altos e mais estreitos (i.e., afunilados) da árvore. Mais precisamente, por meio de rotações, o último elemento acessado na árvore ou um de seus parentes é movido para cima e se torna a nova raiz. Árvores afuniladas foram propostas originalmente por Sleator e Tarjan em 1985 (v. **Bibliografia**) e boa parte do texto e figuras apresentadas nesta seção são baseadas no artigo original desses autores.

Usando essa técnica, denominada **afunilamento**, pode ser que, eventualmente, elementos acessados ocasionalmente sejam promovidos para posições próximas à raiz da árvore de busca, mas a tendência é que, na maioria das vezes, isso ocorra com elementos com grande frequência de acesso.

O resultado que se pretende alcançar com árvores afuniladas é o mesmo pretendido com árvores balanceadas (p. ex., árvores AVL): obter melhor desempenho em operações de busca, inserção e remoção. Contudo esses dois tipos de árvores de busca são bem diferentes em diversos aspectos. Em primeiro lugar, como foi visto na **Seção 4.3**, árvores balanceadas procuram atingir esse objetivo por meio de balanceamento, mas esse não é o caso de árvores afuniladas. Quer dizer, uma árvore afunilada não tem a pretensão de ser balanceada e pode até

mesmo apresentar custo temporal semelhante ao custo temporal de uma tabela encadeada no pior caso [i.e., $\theta(n)$]. Acontece que o princípio que norteia árvores afuniladas não está associado aos seus formatos, pois ele é baseado no conceito heurístico de localidade de referência (v. **Seção 1.5**) e não no conceito de balanceamento. Assim do mesmo modo que um sistema de gerenciamento de memória cache (v. **Seção 1.4**) é susceptível a lapsos de cache, uma árvore afunilada também permite que ocorram operações com custo temporal $\theta(n)$, que corresponde ao pior caso. Novamente, assim como ocorre com memória cache, uma árvore afunilada só apresentará bom desempenho se o número de operações efetuadas no pior caso não superar a expectativa. Mais precisamente, após uma sequência de operações, os elementos próximos ao topo da árvore devem ser acessados com mais frequência do que aqueles que se encontram em níveis inferiores.

Cada vez que se acessa um nó de uma árvore afunilada, executa-se uma alteração radical na árvore, movendo-se o nó recentemente acessado para cima, de modo que ele se torna a raiz da árvore modificada. Desse modo, nós que são frequentemente acessados são elevados e permanecem próximos da raiz. Nós inativos, por outro lado, serão gradativamente colocados cada vez mais longe da raiz.

Árvores afuniladas funcionam bem quando alguns de seus nós são acessados com mais frequência do que outros. Por exemplo, considere um banco de dados de registros de alunos de uma universidade. Agora, suponha que um aluno tranque sua matrícula por um determinado período, de modo que, durante esse período, o registro dele será acessado com pouquíssima frequência. Então é natural que ele seja mantido distante dos registros dos alunos regularmente matriculados e que, portanto, são acessados frequentemente. Quando o aluno mencionado retornar às atividades normais, seu registro passará a ser acessado com mais frequência e, à medida que é acessado, ele vai se aproximando dos registros mais usados.

Outro exemplo no qual o uso de árvore afunilada é adequado seria um banco de dados de informações sobre filmes. Nesse caso, os registros dos filmes mais populares (i.e., aqueles mais acessados) seriam armazenados em nós próximos da raiz da árvore, enquanto filmes menos populares seriam armazenados em nós mais profundos.

4.5.2 Afunilamento

Quando se lida com árvores afuniladas, cada operação de busca, inserção ou remoção é combinada com uma operação básica: o afunilamento de um nó. **Afunilar um nó** significa promovê-lo a raiz e, nesse processo, outros nós podem também ocupar níveis mais elevados na árvore. No contexto corrente, um nó a ser afunilado é denominado **nó-alvo** ou simplesmente **alvo**. É importante notar que, quando a chave de um nó-alvo não faz parte de uma árvore, ele passa a ser aquele que foi acessado por último durante uma operação de busca, inserção ou remoção.

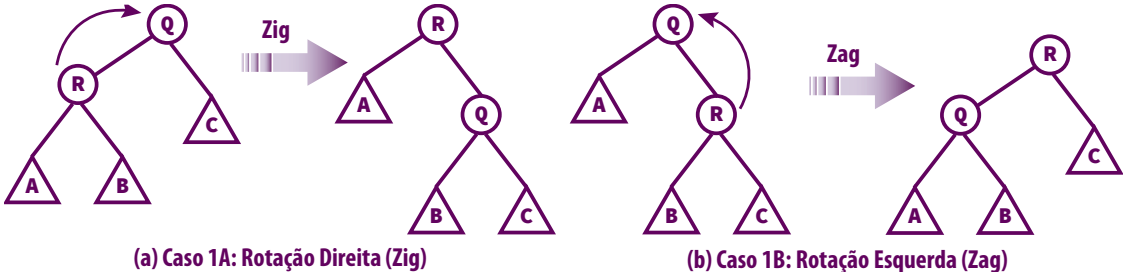
Uma operação de afunilamento de um nó consiste numa sequência de rotações, de modo que, a cada rotação, o nó-alvo passa para um nível mais elevado na árvore (i.e., ele fica cada vez mais próximo da raiz da árvore a cada rotação).

As rotações necessárias para afunilamento de um nó dependem das posições relativas entre ele, seu pai e seu avô. Nos casos que serão descritos a seguir, suponha que R é o nó-alvo, Q é seu pai e P é seu avô. Se o nó R for a própria raiz da árvore, não haverá nenhum reajuste. Caso contrário, existem três casos de reajuste que serão discutidos abaixo.

Antes de prosseguir, lembre-se que o afunilamento de um nó não tem como objetivo balancear a árvore da qual ele faz parte, embora isso eventualmente possa ocorrer. Quer dizer, apesar de afunilamento utilizar rotações, como ocorre com árvores AVL, essas operações são usadas aqui para elevar o nível de um nó e não para rebalancear uma árvore.

Caso 1: O Alvo É Filho da Raiz (Zig e Zag)

Nesse caso, o pai Q de R (o alvo do afunilamento) é a raiz da árvore e uma rotação simples à direita ou à esquerda é suficiente. O tipo de rotação depende do fato de R ser filho esquerdo (que requer rotação direita) ou direito (que requer rotação esquerda) de Q. A **Figura 4–54** ilustra essas situações e os respectivos afunilamentos.



(a) Caso 1A: Rotação Direita (Zig) (b) Caso 1B: Rotação Esquerda (Zag)

Figura 4–54: Casos de Afunilamento Zig e Zag

No jargão de árvores afuniladas, a rotação para a direita é denominada **zig**, enquanto a rotação para a esquerda é denominada de **zag**.

Caso 2: Configuração Homogênea (Zig-zig e Zag-zag)

Existem duas situações nesse caso:

- **Caso 2A:** R é filho esquerdo de Q, que é filho esquerdo de P. Nessa situação, são efetuadas duas rotações à direita, denominadas em conjunto como **zig-zig**. A **Figura 4–55** mostra essa situação.

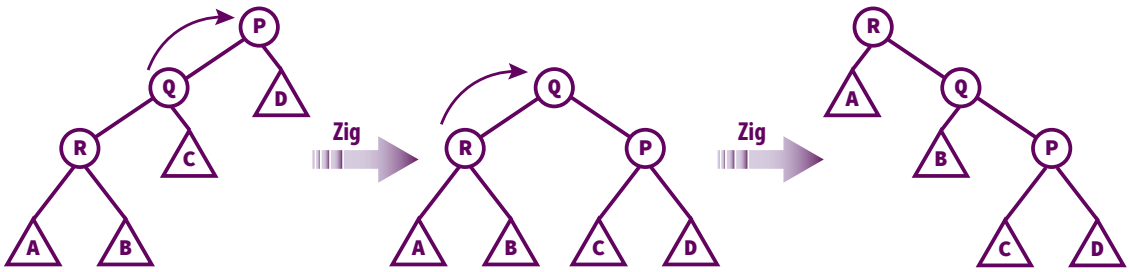


Figura 4–55: Caso de Afunilamento 2A (Zig-zig)

- **Caso 2B:** R é filho direito de Q, que é filho direito de P. Nesse caso, são efetuadas duas rotações à esquerda, denominadas **zag-zag**, como mostra a **Figura 4–56**.

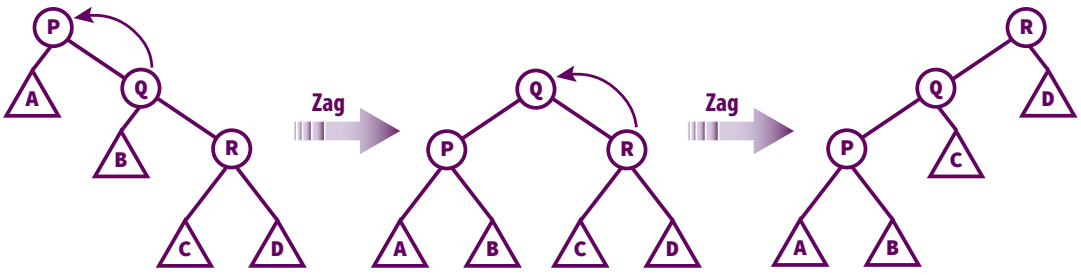


Figura 4–56: Caso de Afunilamento 2B (Zag-zag)

Observe que, nesses dois últimos casos, a primeira rotação gira Q (o pai) em torno de P (o avô) e a segunda rotação, gira R (o filho) em torno de Q (o pai), de modo que, ao final, o nó R é movido dois níveis acima. Note ainda na **Figura 4–55** e na **Figura 4–56** que a primeira rotação tende a balancear a árvore, mas a segunda rotação tende a desbalanceá-la. Portanto esses dois tipos de rotação não possuem correspondência para árvores balanceadas, como árvores AVL (v. **Seção 4.5**). Ou seja, uma operação zig-zig é diferente de uma rotação direita

dupla de um mesmo nó frequentemente usada com árvores AVL, como mostra a **Figura 4–57**. De modo semelhante, zag-zag é diferente de uma rotação esquerda dupla de um mesmo nó.

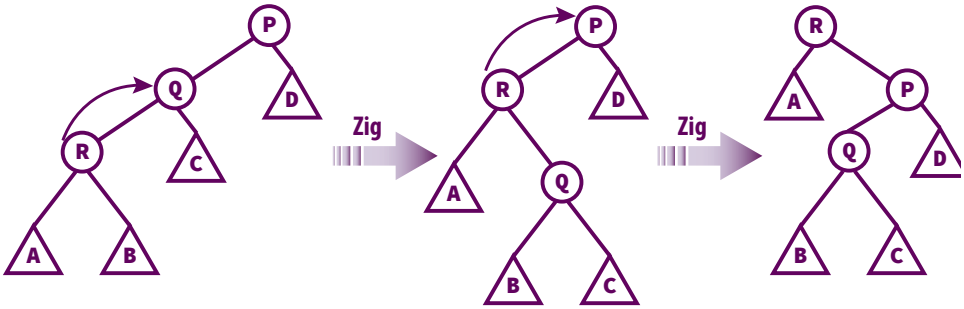


Figura 4–57: Nem Toda Rotação Direita Dupla É Zig-zig

Para evitar o erro mostrado na **Figura 4–57**, sempre pense em elevar o nó-alvo dois níveis em cada caso (exceto quando resta apenas uma única rotação zig ou zag). Note ainda que são apenas os nós no caminho do nó-alvo até a raiz que têm suas posições relativas alteradas. Nenhuma das subárvores dos nós envolvidos diretamente no afunilamento (mostradas como *A* , *B*, *C* e *D* na **Figura 4–56**) altera sua forma.

Caso 3: Configuração Heterogênea (Zag-zig e Zig-zag)

Nesse caso, também existem duas situações:

- **Caso 3A:** *R* é filho direito de *Q*, que é filho esquerdo de *P*. Nessa situação, o afunilamento de *R* requer uma rotação esquerda (zag) de *R* em torno de *Q* e, em seguida, uma rotação direita (zig) de *R* em torno de *P*, como ilustra a **Figura 4–58**. Esse tipo de rotação é denominado **zag-zig**.

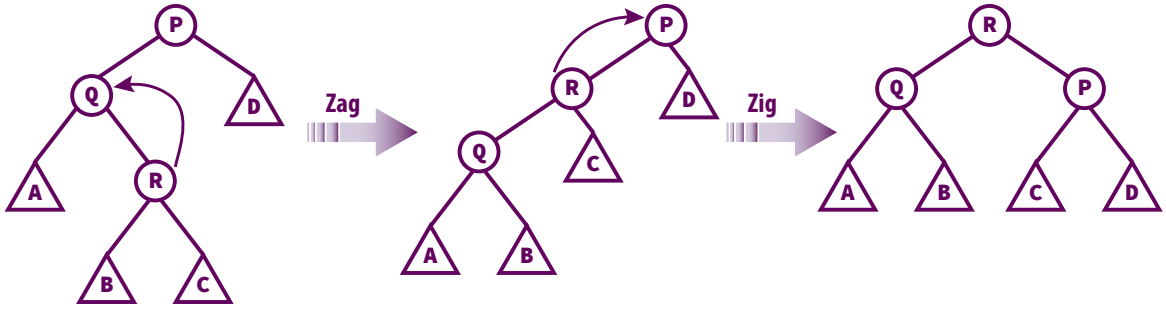


Figura 4–58: Caso de Afunilamento 3A (Zag-zig)

- **Caso 3B:** *R* é filho esquerdo de *Q*, que é filho direito de *P*. O afunilamento nesse caso requer uma rotação direita (zig) de *R* em torno de *Q* e, em seguida, uma rotação esquerda (zag) de *R* sobre *P*. Esse tipo de rotação é denominado **zig-zag** e é ilustrado na **Figura 4–59**.

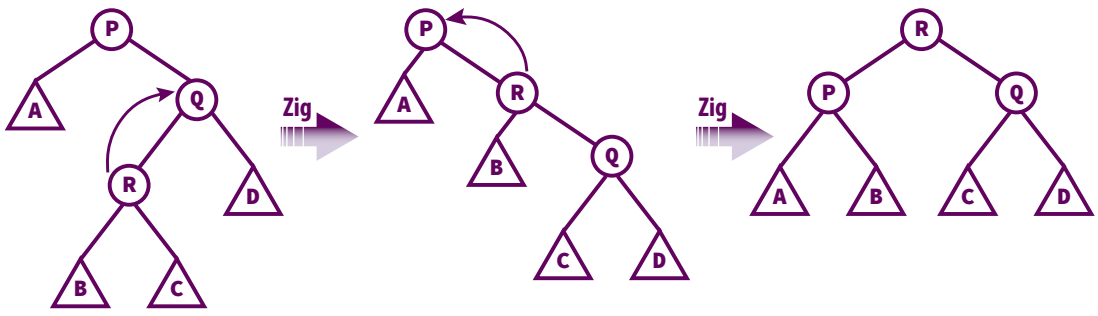


Figura 4–59: Caso de Afunilamento 3B (Zig-zag)

Observe que, nessas duas últimas situações, a primeira rotação gira *R* em torno de *Q* e a segunda rotação gira *R* em torno de *P*, de modo que, ao final, o nó *R* é elevado dois níveis acima. Essas duas rotações tendem a balancear a árvore e possuem rotações correspondentes para árvores AVL (v. **Seção 4.4**).

Exemplos de Afunilamento

A **Figura 4–60** mostra o afunilamento do nó cujo conteúdo é *10* (com fundo colorido na figura). Por sua vez, a **Figura 4–61** mostra o afunilamento do nó cujo conteúdo é *21* (com fundo colorido na figura).

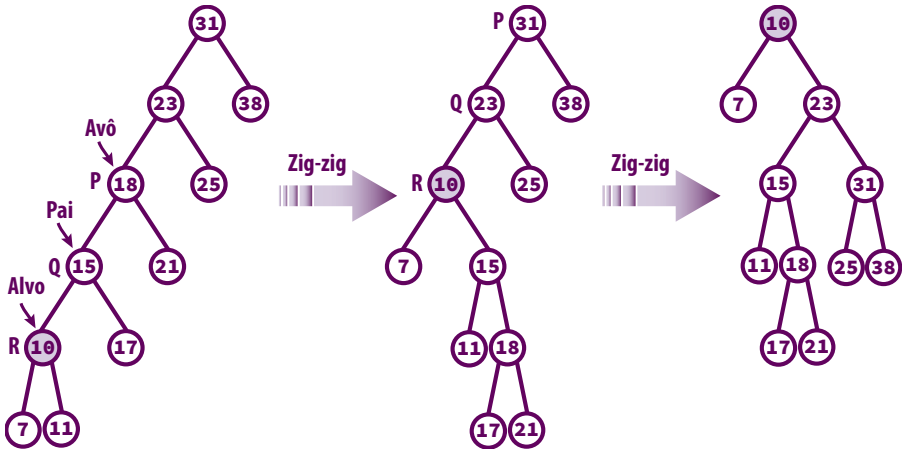


Figura 4–60: Exemplo de Afunilamento 1

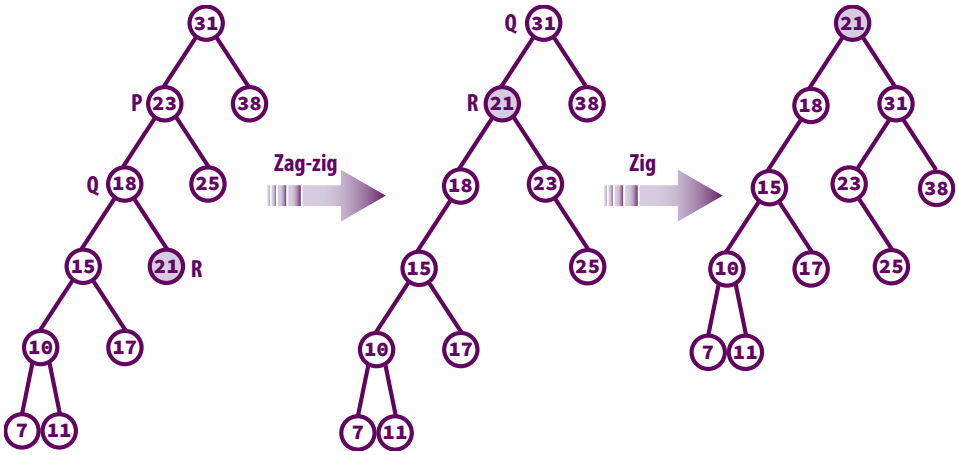


Figura 4–61: Exemplo de Afunilamento 2

Sempre que um nó de uma árvore afunilada é acessado, ele passa a ser a raiz da árvore e esse efeito é obtido por meio das rotações descritas acima. Por outro lado, essas rotações têm como efeito colateral a elevação de níveis de alguns nós encontrados no caminho que leva até o nó em questão. Quanto mais profundo for o nó acessado, maior será o número de nós que têm seus níveis elevados, de modo que futuros acessos a esses nós apresentarão menor custo temporal. Além disso, o efeito líquido de um grande número de afunilamentos numa árvore dessa natureza é um balanceamento *razoável* da árvore (v. **Seção 4.6**).

Agora, se você já entende bem sobre rotações em árvores binárias de busca (o que é provável após tantas rotações), talvez esteja intrigado com a seguinte pergunta: *Não é possível elevar um nó até a raiz simplesmente efetuando rotações desse nó em torno de seu pai sem ter que se preocupar com posições relativas de nós?* A resposta a essa questão é obviamente *sim*, visto que, a cada rotação, o pivô da rotação é elevado um nível acima. Assim de

rotação em rotação em torno de seu pai ele acabará chegando até a raiz. Ocorre, porém, que o efeito colateral desejado não é obtido. Ou seja, utilizando esse raciocínio os nós próximos ao nó afunilado não ficam cada vez mais próximos da raiz da árvore como se deseja.

Para entender melhor essa última afirmação, suponha, por exemplo, que se pretenda afunilar o nó contendo a chave c_1 na árvore binária de busca ilustrada na **Figura 4–62 (a)**. Então quando esse afunilamento é conduzido corretamente, a árvore obtida é aquela mostrada na **Figura 4–62 (b)**. Por outro lado, quando a configuração da árvore é alterada por meio das rotações descritas no parágrafo anterior, a árvore obtida é aquela mostrada na **Figura 4–63 (b)**.

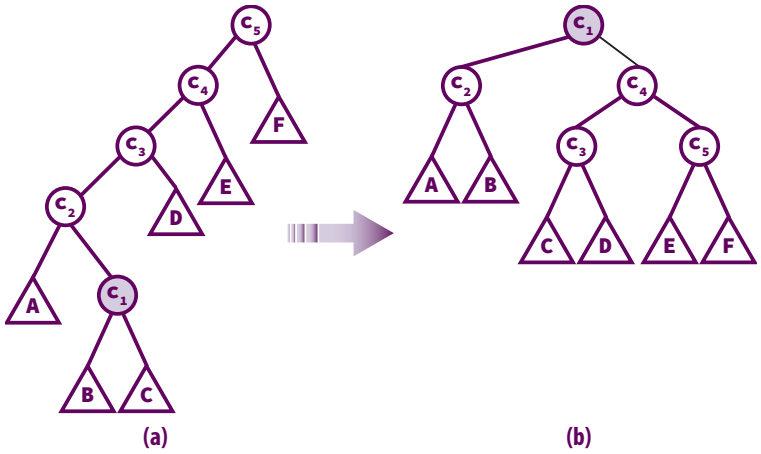


Figura 4–62: Afunilamento Correto de um Nó

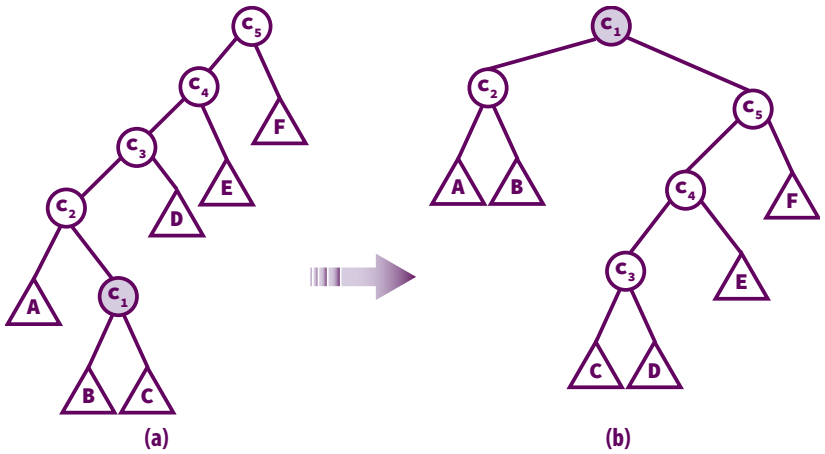


Figura 4–63: Afunilamento Incorreto de um Nó

O que se nota nessas duas últimas figuras é que a árvore obtida por meio das rotações descritas antes apresenta o efeito principal e o efeito colateral desejados. Por outro lado, as rotações intuitivas descritas acima produziram apenas o efeito principal desejado que era elevar o nó alvo até a raiz. Quer dizer, nós que inicialmente estavam próximos do nó-alvo não foram elevados junto com ele. Por exemplo, como mostra a **Figura 4–63**, o nó contendo a chave c_3, que, inicialmente, estava a dois níveis de distância do nó afunilado, passou a distar três níveis do nó afunilado após as rotações equivocadas. É importante notar ainda que a árvore da **Figura 4–62 (b)** é mais bem balanceada do que a árvore da **Figura 4–63 (b)** e esse é outro efeito colateral importante ocasionalmente obtido com as rotações corretas.

Embora seja difícil visualizar esses efeitos colaterais estudando-se exemplos pequenos como aqueles apresentados neste livro, afunilamento tem como efeito secundário diminuir aproximadamente pela metade as alturas dos nós encontrados no caminho que leva a um nó-alvo.

A nomenclatura *zig*, *zag*, *zig-zag* etc. utilizada para rotular as operações de rotação descritas acima não contribui com nada para o melhor entendimento dessas operações. Mas em compensação, também não atrapalha. Essa terminologia foi incluída neste texto porque ela é comumente utilizada em discussões sobre árvores afuniladas. No fundo, ela é irrelevante e você pode, se preferir, continuar usando *esquerda*, *direita*, *esquerda-direita* etc.

Afunilamentos Ascendente e Descendente

Existem duas maneiras básicas de afunilar um nó. Em ambas, se o nó for encontrado, ele é colocado na raiz, enquanto, se o nó não for encontrado, o último nó acessado durante a busca pelo nó-alvo se torna raiz.

O tipo de afunilamento descrito informalmente até aqui é denominado **afunilamento ascendente**, pois ele parte do nó alvo que pode se encontrar num nível inferior da árvore e sobe até a raiz que está no nível mais elevado da árvore. Mas existe um outro tipo de afunilamento, denominado **afunilamento descendente**, que afunila a árvore à medida que seus nós são visitados na busca pelo nó a ser afunilado.

A implementação de afunilamento ascendente não é tão trivial quanto parece sugerir a discussão apresentada no início desta seção, mas não é tão difícil de implementar se ela for recursiva. Uma alternativa para implementação de afunilamento ascendente é o uso de um ponteiro para o pai de cada nó. Essa opção acrescenta um custo extra em termos de espaço (porque um campo a mais é usado em cada nó) e tempo de execução (porque esse campo adicional precisa ser frequentemente atualizado).

O algoritmo de afunilamento descendente baseia-se no fato de qualquer operação de busca, inserção ou remoção requerer afunilamento. Então por que não afunilar todos os nós à medida que eles são visitados durante uma busca pelo nó-alvo? Assim procedendo, torna-se desnecessário guardar todos endereços dos nós visitados, como implicitamente ocorre com o algoritmo de afunilamento ascendente. Portanto o algoritmo de afunilamento descendente é relativamente mais eficiente e mais fácil de implementar do que o algoritmo de afunilamento ascendente.

A **Figura 4–64** apresenta o algoritmo de afunilamento descendente. Esse algoritmo é baseado no artigo original de Sleator e Tarjan (1985) (v. **Bibliografia**).

ALGORITMO AFUNILAMENTODESCENDENTE

ENTRADA: A chave (*c*) do nó a ser afunilado

ENTRADA/SAÍDA: Uma árvore binária de busca

1. Se a árvore estiver vazia, retorne

2. Faça dois ponteiros, *esq* e *dir*, apontarem para um nó auxiliar (*noAux*) com subárvores vazias

3. Enquanto o nó contendo a chave a ser afunilada não for a raiz ou filho da raiz, faça:

  3.1 Se *c* for menor do que a chave da raiz, faça:

    3.1.1 Se a subárvore esquerda da raiz estiver vazia, encerre o laço

    3.1.2 Se *c* for menor do que a chave da raiz da subárvore esquerda

      3.1.2.1 Faça uma rotação direita na árvore

      3.1.2.2 Se a subárvore esquerda ficou vazia, encerre o laço

CONTINUA

FIGURA 4–64: ALGORITMO DE AFUNILAMENTO DESCENDENTE

Algoritmo AfunilamentoDescendente (Continuação)

3.1.3 Faça a subárvore esquerda do ponteiro *dir* apontar para a árvore
3.1.4 Faça o ponteiro *dir* apontar para a árvore
3.1.5 Faça o ponteiro que representa a árvore apontar para seu filho esquerdo
3.2 Caso contrário, se *c* for maior do que a chave da raiz da subárvore esquerda
3.2.1 Se a subárvore direita da raiz estiver vazia, encerre o laço
3.2.2 Se *c* for maior do que a chave da raiz da subárvore direita
3.2.2.1 Faça uma rotação esquerda na árvore
3.2.2.2 Se a subárvore direita ficou vazia, encerre o laço
3.2.3 Faça a subárvore direita do ponteiro *esq* apontar para a árvore
3.2.4 Faça o ponteiro *esq* apontar para a árvore
3.2.5 Faça o ponteiro que representa a árvore apontar para seu filho direito
3.3 Caso contrário (a chave da raiz é igual a *c*), encerre o laço
4. Faça a subárvore direita do ponteiro *esq* apontar para o filho esquerdo da árvore
5. Faça a subárvore esquerda do ponteiro *dir* apontar para o filho direito da árvore
6. Faça o filho esquerdo da árvore apontar para o filho direito de *noAux*
7. Faça o filho direito da árvore apontar para o filho esquerdo de *noAux*

Figura 4–64 (Cont.): Algoritmo de Afunilamento Descendente

A **Figura 4–65** ilustra o funcionamento do algoritmo da **Figura 4–64** no afunilamento do nó cujo conteúdo é *21* (com fundo colorido na figura). Note que esse é o mesmo exemplo de afunilamento ascendente apresentado na **Figura 4–61** e que as árvores resultantes nos dois exemplos *não são iguais* (**Figura 4–66**).

A **Figura 4–66** mostra, de maneira mais clara, a árvore resultante do exemplo de afunilamento visto na **Figura 4–65**. Compare essa última árvore com aquela obtida no exemplo da **Figura 4–61** e note que elas são diferentes, apesar de a diferença entre elas ser ínfima.

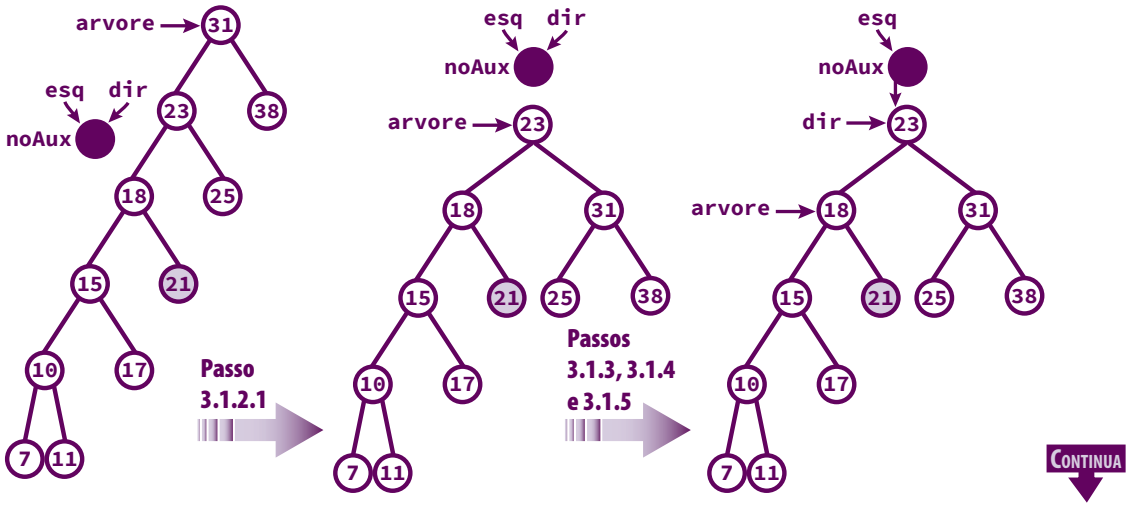


Figura 4–65: Exemplo de Afunilamento Descendente de Nó 1

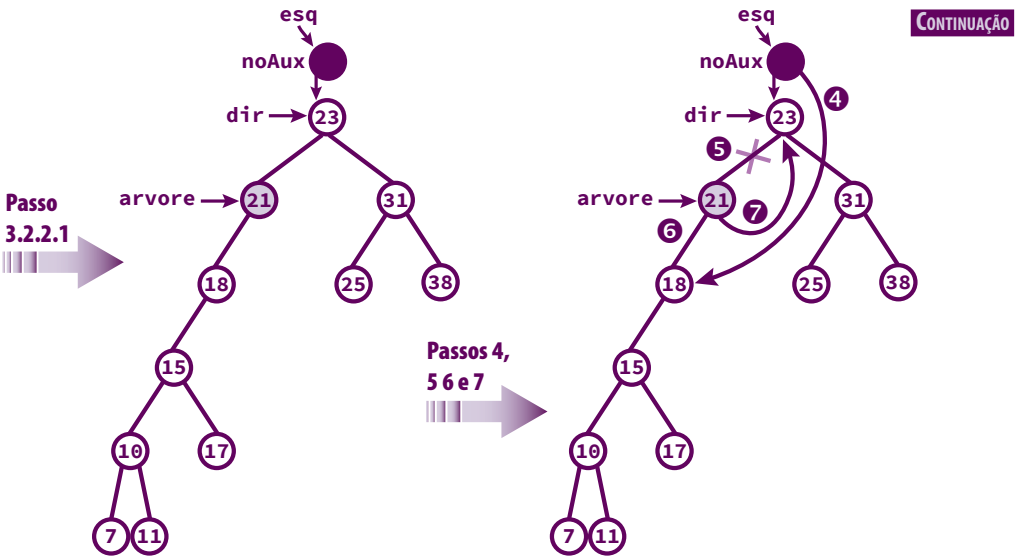


FIGURA 4–65 (CONT.): ALGORITMO DE AFUNILAMENTO DESCENDENTE DE NÓ 1

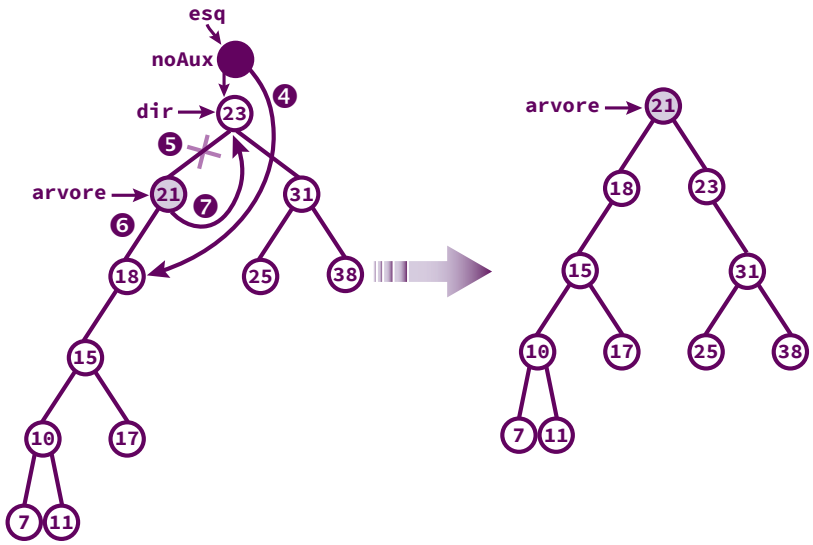


FIGURA 4–66: EXEMPLO DE AFUNILAMENTO DESCENDENTE DE NÓ 2

4.5.3 Operações Básicas

Busca

O algoritmo de busca de um nó contendo uma chave *c* é descrito na **Figura 4–67**.

ALGORITMO BUSCAEMÁRVOREAFUNILADA

ENTRADA: Uma chave de busca

ENTRADA/SAÍDA: Uma árvore afunilada

SAÍDA: O valor associado à chave do nó cuja chave casa com a chave de busca ou um valor informando que a chave não foi encontrada

CONTINUA

FIGURA 4–67: ALGORITMO DE BUSCA EM ÁRVORE AFUNILADA

Algoritmo BuscaEmÁrvoreAfunilada (Continuação)

1. Se a árvore estiver vazia, retorne um valor informando que a chave não se encontra na árvore
2. Afunile a chave *c* usando o algoritmo **AfunilamentoDescendente** (v. **Figura 4–64**)
3. Compare a chave de busca com aquela que se encontra na raiz:
   - **3.1** Se as chaves forem iguais, retorne o valor associado à chave da raiz
   - **3.2** Se as chaves forem diferentes, retorne um valor informando que a chave não se encontra na árvore

Figura 4–67 (Cont.): Algoritmo de Busca em Árvore Afunilada

Inserção

O algoritmo de inserção de um nó contendo uma determinada chave *c* numa árvore afunilada é descrito na **Figura 4–68**.

Algoritmo InsereEmÁrvoreAfunilada

Entrada: O conteúdo de um novo nó
Entrada/Saída: Uma árvore afunilada
Saída: Um valor informando se a operação foi bem-sucedida

1. Se a árvore estiver vazia, crie um novo nó com o conteúdo de entrada, torne-o raiz da árvore e retorne informando o sucesso da operação
2. Afunile a chave *c* do novo nó usando o algoritmo da **Figura 4–64**
3. Se a chave que se encontra na raiz for igual a *c*, encerre informando que não ocorreu inserção (pois a chave é considerada primária)
4. Crie um novo nó contendo a chave *c*
5. Se *c* for menor do que a chave que se encontra na raiz:
   - **5.1** Faça o filho direito do novo nó apontar para a raiz
   - **5.2** Faça o filho esquerdo do novo nó apontar para o filho esquerdo da raiz
   - **5.3** Torne nulo o filho esquerdo da raiz
6. Se *c* for maior do que a chave que se encontra na raiz:
   - **6.1** Faça o filho esquerdo do novo nó apontar para a raiz
   - **6.2** Faça o filho direito do novo nó apontar para o filho direito da raiz
   - **6.3** Torne nulo o filho direito da raiz
7. Faça o ponteiro que representa a árvore apontar para o novo nó

Figura 4–68: Algoritmo de Inserção em Árvore Afunilada

É importante notar que o **Passo 5** do algoritmo de inserção apresentado na **Figura 4–68** assume que a chave do nó a ser inserido é maior do que a chave do filho esquerdo do nó afunilado (que ora se encontra na raiz), pois, se esse não for o caso, não se terá uma árvore binária de busca ao final do processo de inserção. Ocorre, porém, que, nesse caso, existe uma relação entre o nó a ser inserido e o nó afunilado que impede que essa suposição seja contrariada. Para tornar a discussão mais palpável, o que se pretende mostrar é que uma situação como aquela ilustrada na **Figura 4–69** não pode ocorrer se o **Passo 5** do algoritmo sob discussão for seguido.

O que coíbe o surgimento de uma situação como aquela da **Figura 4–69** é o fato de existir uma relação entre o nó a ser inserido e o nó que foi afunilado. Ou seja, quando se tenta afunilar um nó cuja chave não se encontra numa árvore afunilada, o nó que é realmente afunilado é o último nó acessado. No caso de inserção, esse último

nó acessado é aquele que seria pai do nó a ser inserido se a inserção ocorresse como numa árvore binária ordinária de busca. Portanto existem duas relações alternativas importantes entre o nó ser inserido e o nó afunilado:

[1] O nó a ser inserido é antecessor imediato do nó afunilado (caso ele fosse inserido como filho esquerdo numa inserção ordinária) ou

[2] O nó a ser inserido é sucessor imediato do nó afunilado (caso ele fosse inserido como filho direito numa inserção ordinária).

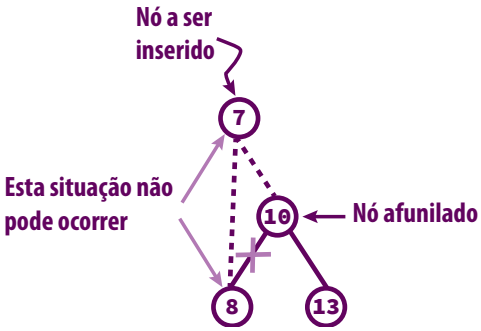


FIGURA 4–69: CONFIGURAÇÃO IMPOSSÍVEL EM INSERÇÃO DE NÓ APÓS AFUNILAMENTO

Ora, mas se o nó a ser inserido for antecessor imediato do nó afunilado, todas as chaves que são menores do que a chave do nó afunilado também são menores do que a chave do nó a ser inserido. Isso explica por que a situação mostrada na **Figura 4–69** não pode acontecer.

Um raciocínio semelhante é usado para mostrar que, quando a chave do nó a ser inserido é maior do que a chave do nó afunilado, o filho direito do nó afunilado não pode ter chave menor do que aquela do nó a ser inserido. Ou seja, isso mostra que o **Passo 6** do algoritmo de inserção exibido na **Figura 4–68** também é correto.

Remoção

O algoritmo de remoção de um nó contendo uma chave *c* de uma árvore afunilada é apresentado na **Figura 4–70**.

ALGORITMO REMOVEEMÁRVOREAFUNILADA

ENTRADA: A chave (*c*) do nó que será removido

ENTRADA/SAÍDA: Uma árvore afunilada

SAÍDA: Um valor informando se a operação foi bem-sucedida

1. Se a árvore estiver vazia, retorne um valor informando que a chave não se encontra na árvore
2. Afunile a chave *c* usando o algoritmo da **Figura 4–64**
3. Se a chave que se encontra na raiz for diferente de *c*, retorne um valor informando que a chave não se encontra na árvore.
4. Se a chave que se encontra na raiz for igual a *c*, faça o seguinte:

   4.1 Se a raiz não possui filhos:

   4.1.1 Remova a raiz

   4.1.2 Torne nulo o ponteiro que representa a árvore

   4.1.3 Retorne um valor informando o sucesso da operação

CONTINUA

FIGURA 4–70: ALGORITMO DE REMOÇÃO EM ÁRVORE AFUNILADA

Algoritmo RemoveEmÁrvoreAfunilada (Continuação)

4.2 Se a raiz possui apenas um filho:

4.2.1 Faça o ponteiro que representa a árvore apontar para o filho da raiz

4.2.2 Remova a raiz

4.2.3 Retorne um valor informando o sucesso da operação

4.3 Faça dois ponteiros *E* e *D* apontarem para as os filhos esquerdo e direito da raiz, respectivamente

4.4 Libere a raiz. (Neste ponto, obtêm-se duas árvores: o ponteiro *E* aponta para uma delas e o ponteiro *D* aponta para a outra. Todas as chaves da primeira árvore são menores do que as chaves da segunda árvore)

4.5 Afunile o nó que contém a menor chave na árvore *D* (ao final, *D* estará apontando para esse nó)

4.6 Como o último nó afunilado não tem filho esquerdo, faça com que o filho esquerdo desse nó aponte para a árvore apontada por *E*.

4.7 Faça o ponteiro que representa a árvore apontar para o nó apontado por *D*.

4.8 Retorne um valor informando o sucesso da operação

Figura 4–70 (Cont.): Algoritmo de Remoção em Árvore Afunilada

4.5.4 Implementação Descendente

Definições de Tipos

Afunilamento descendente não requer o acréscimo de nenhum campo ao nó de uma árvore binária ordinária de busca, de modo que aqui serão utilizadas as mesmas definições de tipos apresentadas na **Seção 4.1.2**.

Afunilamento

A operação de afunilamento descendente é implementada pela função `Afunila()` apresentada abaixo. Essa função retorna a raiz da árvore após o afunilamento e usa os seguintes parâmetros:

- `arvore` (entrada/saída) — raiz da árvore que será afunilada
- `chave` (entrada) — chave do nó que será afunilado

Note que, se a chave não for encontrada, o último nó acessado se tornará a raiz da árvore. Essa função é chamada pelas demais funções que implementam as operações básicas de busca, inserção e remoção.

```
tArvoreBB Afunila(tArvoreBB arvore, tCEP chave)
{
   int         compara;/* Resultado de comparação de duas chaves */
   tNoArvoreBB noAux; /* Um nó auxiliar */
   tArvoreBB   esq,   /* Ponteiro auxiliar */
               dir;   /* Outro ponteiro auxiliar */

      /* Passo 1: Se a árvore estiver vazia, retorne */
   if (!arvore)
      return arvore;

      /* Passo 2: Faça 'esq' e 'dir' apontarem para 'noAux' */
   esq = dir = &noAux;
   noAux.esquerda = noAux.direita = NULL;
```

```
   /* Passo 3: Enquanto o nó contendo a chave a ser   */
   /* afunilada não for a raiz ou filho da raiz, faça */
while(1) {
      /* Compara a chave do nó a ser afunilado com a chave da raiz */
   compara = strcmp(chave, arvore->conteudo.chave);

      /* Passo 3.1: Verifique se a chave encontra-se na subárvore esquerda */
   if (compara < 0) {
         /* Passo 3.1.1: Se a chave não se encontra */
         /* na subárvore esquerda, encerre o laço   */
      if (!arvore->esquerda)
         break;

         /* Passo 3.1.2: Verifique se a chave é menor do */
         /* que a chave da raiz da subárvore esquerda    */

      compara = strcmp(chave, arvore->esquerda->conteudo.chave);

      if (compara < 0) {
            /* Passo 3.1.2.1: Faça uma rotação zig na árvore */
         arvore = RotacaoDireitaArvoreBB(arvore);

            /* Passo 3.1.2.2: Encerre o laço se a subárvore esquerda ficou vazia */
         if (!arvore->esquerda)
            break;
      }

         /* Passo 3.1.3: Faça a subárvore esquerda  */
         /* do ponteiro 'dir' apontar para a árvore */
      dir->esquerda = arvore;

         /* Passo 3.1.4: Faça o ponteiro 'dir' apontar para a árvore */
      dir = arvore;

         /* Passo 3.1.5: Faça o ponteiro que representa */
         /* a árvore apontar para seu filho esquerdo    */
      arvore = arvore->esquerda;

      /* Passo 3.2: Verifique se a chave se encontra na subárvore direita */
   } else if (compara > 0) {
         /* Passo 3.2.1: Se a chave não se encontra */
         /* na subárvore direita, encerre o laço    */
      if (!arvore->direita)
         break;

         /* Passo 3.2.2: Verifique se a chave é maior do */
         /* que a chave da raiz da subárvore direita     */

      compara = strcmp(chave, arvore->direita->conteudo.chave);

      if (compara > 0) {
            /* Passo 3.2.2.1: Faça uma rotação zag na árvore */
         arvore = RotacaoEsquerdaArvoreBB(arvore);

            /* Passo 3.2.2.2: Encerra o laço se a subárvore direita ficou vazia */
         if (!arvore->direita)
            break;
      }
         /*  Passo 3.2.3: Faça a subárvore direita do */
         /* ponteiro 'esq' apontar para a árvore      */
      esq->direita = arvore;

         /* Passo 3.2.4: Faça o ponteiro 'esq' apontar para a árvore */
```

```
         esq = arvore;

            /*  Passo 3.2.5: Faça o ponteiro que representa */
            /* a árvore apontar para seu filho direito      */
         arvore = arvore->direita;
      } else /* A chave já está na raiz */
         break;
   }

      /* Passo 4: Faça a subárvore direita do ponteiro */
      /* 'esq' apontar para o filho esquerdo da árvore */
   esq->direita = arvore->esquerda;

      /* Passo 5: Faça a subárvore esquerda do ponteiro */
      /* 'dir' apontar para o filho direito da árvore   */
   dir->esquerda = arvore->direita;

      /* Passo 6: Faça o filho esquerdo da árvore */
      /* apontar para o filho direito de 'noAux'  */
   arvore->esquerda = noAux.direita;

      /* Passo 7: Faça o filho direito da árvore  */
      /* apontar para o filho esquerdo de 'noAux' */
   arvore->direita = noAux.esquerda;

   return arvore;
}
```

Busca

A função `BuscaArvoreFunil()` apresentada abaixo implementa a operação de busca numa árvore afunilada. Essa função usa os seguintes parâmetros:

- `arvore` (entrada) — árvore que será pesquisada
- `chave` (entrada) — chave de busca

A função `BuscaArvoreFunil()` retorna o endereço do conteúdo que contém a referida chave, se ela for encontrada. Caso contrário, ela retorna **NULL**.

```
tCEP_Ind *BuscaArvoreFunil(tArvoreBB *arvore, tCEP chave)
{
      /* Verifica se a árvore está vazia */
   if (!*arvore)
      return NULL; /* A árvore está vazia */

      /* Afunila o nó que contém a chave de busca */
   *arvore = Afunila(*arvore, chave);

      /* Se a chave foi encontrada, ela agora está na raiz */
   if (!strcmp(chave, (*arvore)->conteudo.chave))
      return &(*arvore)->conteudo;

   return NULL; /* A chave não foi encontrada */
}
```

Inserção

A função `InsereArvoreFunil()` apresentada abaixo implementa a operação de inserção de nós numa árvore afunilada. Essa função usa os seguintes parâmetros:

- `*arvore` (entrada/saída) — ponteiro para a raiz da árvore na qual será feita a inserção
- `*conteudo` (entrada) — conteúdo do nó que será inserido

A função `InsereArvoreFunil()` retorna `1`, se não houver inserção porque a chave já existe, ou `0`, se a inserção ocorrer.

```
int InsereArvoreFunil(tArvoreBB *arvore, tCEP_Ind *conteudo)
{
   tArvoreBB pNovoNo; /* Ponteiro para nó que será inserido */
   int          compara; /* Resultado da comparação de duas chaves */

      /* Se a árvore estiver vazia torna o novo nó sua raiz e retorna */
   if (!*arvore) {
      *arvore = ConstroiNoArvoreBB(*conteudo);
      return 0;
   }

      /* Afunila o nó que contém a chave do nó a ser inserido */
   *arvore = Afunila(*arvore, conteudo->chave);

      /* Compara a chave do nó a ser inserido com a chave que */
      /* se encontra na raiz da árvore após o afunilamento    */
   compara = strcmp(conteudo->chave, (*arvore)->conteudo.chave);

      /* Verifica se a chave já existe */
   if (!compara)
      return 1; /* A chave já existe e é primária */

   pNovoNo = ConstroiNoArvoreBB(*conteudo); /* Constrói o novo nó */

      /* Faz com que o novo nó se torne a raiz da árvore */
   if (compara < 0) {
      /*                                                      */
      /* A chave do novo nó é menor do que a chave da raiz atual */

         /* Faz o filho direito do novo nó apontar para a raiz */
      pNovoNo->direita = *arvore;

         /* Faz o filho esquerdo do novo nó apontar */
         /* para o filho esquerdo da raiz           */
      pNovoNo->esquerda = (*arvore)->esquerda;

      (*arvore)->esquerda = NULL; /* Torna nulo o filho esquerdo da raiz */
   } else {
      /*                                                      */
      /* A chave do novo nó é maior do que a chave da raiz atual */

         /* Faz o filho esquerdo do novo nó apontar para a raiz */
      pNovoNo->esquerda = *arvore;

         /* Faz o filho direito do novo nó apontar para o filho direito da raiz */
      pNovoNo->direita = (*arvore)->direita;
      (*arvore)->direita = NULL;
   }

   *arvore = pNovoNo; /* O novo nó passa a ser a nova raiz */

   return 0; /* Inserção foi OK */
}
```

Remoção

A função `RemoveNoArvoreFunil()` apresentada abaixo implementa a operação de inserção de nós numa árvore afunilada. Essa função retorna `0`, se a remoção for bem-sucedida, ou `1`, se o nó a ser removido não for encontrado. Ela usa como parâmetros:

- `arvore` (entrada/saída) — endereço do ponteiro que representa a árvore na qual será feita a remoção
- `chave` (entrada) — chave do nó a ser removido

```
int RemoveArvoreFunil(tArvoreBB *arvore, tCEP chave)
{
   tArvoreBB pE, /* Apontará para o filho esquerdo da raiz */
               pD, /* Apontará para o filho direito da raiz  */
               pS, /* Apontará para o sucessor da raiz */
               p;  /* Um ponteiro auxiliar */

      /* Verifica se a árvore está vazia */
   if (!*arvore)
      return 1; /* Não há o que remover */

      /* Afunila o nó que será removido */
   *arvore = Afunila(*arvore, chave);

      /* Se a chave foi encontrada, ela agora está na raiz */
   if (strcmp(chave, (*arvore)->conteudo.chave))
      return 1; /* Chave não foi encontrada */

   /*                                                    */
   /* A chave encontra-se na raiz e será removida a seguir */
   /*                                                    */

      /* Verifica quantos filhos a raiz possui */
   if (!(*arvore)->esquerda && !(*arvore)->direita) {
         /* A raiz não tem filhos logo a árvore ficará vazia após a remoção */
      free(*arvore);
      *arvore = NULL;
   } else if ((*arvore)->esquerda && (*arvore)->direita) {
      /*                          */
      /* A raiz tem dois filhos */

      pE = (*arvore)->esquerda; /* Faz pE apontar para o filho esquerdo da raiz */

      pD = (*arvore)->direita; /* Faz pD apontar para o filho direito da raiz */

         /* A raiz já pode ser liberada */
      free(*arvore);

         /* Obtém o sucessor imediato da antiga raiz */
      pS = MenorNoArvoreBB(pD);

         /* Afunila o sucessor imediato da antiga raiz */
      pD = Afunila(pD, pS->conteudo.chave);

         /* Apesar de o sucessor ter sido movido para a raiz, */
         /* ele ainda deve ocupar o mesmo endereço em memória */
      ASSEGURA(pS == pD, "Sucessor nao esta' na raiz");

         /* O nó apontado por pD não pode ter filho esquerdo */
      ASSEGURA(!pD->esquerda, "Sucessor tem filho esquerdo");

         /* Faz com que o filho esquerdo do sucessor */
         /* seja a árvore apontada por pE            */
      pD->esquerda = pE;

         /* Faz o ponteiro que representa a árvore apontar para a nova raiz */
      *arvore = pD;
   } else {
      /* A raiz só tem um filho. Esse filho será a */
      /* nova raiz e a raiz antiga será removida.  */
```

```
      p = *arvore; /* Guarda o endereço da raiz para removê-la depois */

         /* Faz a raiz apontar para seu único filho */
      *arvore = (*arvore)->esquerda ? (*arvore)->esquerda : (*arvore)->direita;

      free(p); /* Remove a raiz antiga */
   }

   return 0; /* A remoção foi bem sucedida */
}
```

A função `RemoveNoArvoreFunil()` chama `MenorNoArvoreBB()` para encontrar o nó que contém a menor chave de uma árvore binária de busca. Essa última função é similar à função `MenorChaveArvoreBB()` que será apresentada na **Seção 4.7.1**.

4.5.5 Análise

Árvores afuniladas apresentam algumas boas propriedades em comparação com árvores balanceadas, tais como árvores AVL. Implementação de árvores afuniladas, por exemplo, é bem mais simples do que implementações de árvores balanceadas.

Árvores afuniladas apresentam ótima localidade de referência (v. **Seção 1.5.1**) porque o nó mais recentemente acessado passa a ser raiz da árvore, de modo que um acesso subsequente desse nó terá custo $\theta(1)$, o que mostra que árvores afuniladas apresentam ótima localidade temporal. Além disso, nós próximos ao nó mais recentemente acessado têm seus níveis elevados, o que mostra que árvores afuniladas apresentam ótima localidade espacial. Por causa dessas características, árvores afuniladas são frequentemente usadas na implementação de algoritmos de gerenciamento de cache (v. **Seção 1.4**).

A eficiência de uma árvore afunilada não é derivada de seu formato, como ocorre com árvores balanceadas. Essa eficiência é derivada da mesma heurística que norteia o conceito de cache. Ou seja o objetivo de um afunilamento é minimizar o número de operações necessárias para acessar um nó durante certo período. Assim o custo temporal $\theta(n)$ que árvores afuniladas apresentam no pior caso não é considerado ruim, desde que esse pior caso não ocorra com muita frequência (v. **Capítulo 5**).

A **Tabela 4–3** apresenta um resumo de pontos positivos e negativos exibidos por árvores afuniladas.

Prós	Contras
Operações de busca, inserção e remoção têm, em média, custo temporal $\theta(\log n)$, considerando um número de operações suficientemente grande e não uniformes	Individualmente, cada operação tem, no pior caso, custo temporal $\theta(n)$. Em contraste, árvores AVL possuem custo temporal $\theta(\log n)$ no pior caso
Usam menos espaço do que árvores AVL, pois não precisam armazenar informações adicionais para auxiliar o autoajuste	—
São muito mais fáceis de implementar do que árvores AVL	—

Tabela 4–3: Prós e Contras de Árvores Afuniladas

4.6 Comparando Árvores Binárias de Busca

Este capítulo discutiu três tipos de árvores binárias de busca:

[1] Árvore binária ordinária de busca (v. **Seção 4.1**)

[2] Árvore AVL (v. **Seção 4.4**)

[3] Árvore afunilada (v. **Seção 4.5**)

Cada abordagem de implementação de árvore binária de busca possui vantagens e desvantagens dependendo do contexto no qual elas são utilizadas. Esta seção indica em que situação cada uma delas é a mais apropriada.

Atualmente, árvores binárias de busca ordinárias são usadas apenas com o propósito didático, pois, além de serem mais fáceis de implementar, elas servem como base para implementações mais sofisticadas e complicadas.

Em qualquer implementação de tabela de busca, o que importa é o custo com que as operações básicas de busca, inserção e remoção são executadas. Mas deve-se ainda levar em consideração a dificuldade com que um programador se depara durante a própria implementação.

O foco de árvores afuniladas é em cada nó individualmente, enquanto, em árvores balanceadas, o foco é na altura da árvore constituída pelo conjunto de nós. Quer dizer, o enfoque da técnica de afunilamento é nos elementos em si e não no formato da árvore de busca. Essa técnica funciona bem quando alguns elementos são acessados com muito mais frequência do que outros. Se elementos próximos da raiz forem acessados com a mesma frequência com que elementos em níveis bem mais baixos são acessados, então árvore afunilada não é uma boa escolha para tabela de busca. Nesse caso, é melhor utilizar uma árvore com autobalanceamento (i.e., árvore AVL).

Para facilidade de referência, a **Tabela 4–4** compara árvores afuniladas e árvores AVL.

	Árvore Afunilada	Árvore AVL
Princípio básico	Localidade de referência	Balanceamento
Complexidade temporal	$\theta(n)$ no pior caso	$\theta(\log n)$ no pior caso
Complexidade espacial	$\theta(n)$	$\theta(n)$
Custo temporal amortizado	$\theta(\log n)$	$\theta(\log n)$
Balanceada?	Não	Sim
Paradigma	Heurístico	Determinístico
Implementação	Relativamente fácil	Complicada se recursão não for usada

Tabela 4–4: Diferenças entre Árvores Afuniladas e Árvores AVL

A título de comparação, a **Tabela 4–5** mostra as alturas das árvores binárias de busca criadas para o arquivo `CEPs.bin` descrito no **Apêndice A**. Como esse arquivo possui *673.580* registros, a altura (ideal) de uma árvore perfeitamente balanceada seria igual a *20* (i. e, $log_2\ 673580$), que, conforme foi discutido na **Seção 4.4**, não faz sentido tentar obter na prática.

Abordagem	Altura da árvore
Árvore Binária Ordinária	*3.376*
Árvore AVL	*24*
Árvore Afunilada	*445*

Tabela 4–5: Desempenho de Árvores Binárias de Busca com CEPs.bin

A **Tabela 4–5** mostra que os resultados relativos obtidos condizem com a expectativa. Ou seja, era esperado que uma árvore binária ordinária fosse bastante desbalanceada, visto que muitas chaves (CEPs) no arquivo `CEPs.bin` estão ordenadas. Por outro lado, o fato de árvores AVL possuírem alturas próximas da altura ideal, com pequena vantagem para a árvore AVL, também era esperado. Enfim a implementação de tabela de busca com árvore afunilada parece ruim, visto que a altura dessa árvore é cerca de 20 vezes maior do que a altura

ideal. Mas novamente, lembre-se que árvore afunilada não é árvore balanceada, de forma que uma altura ainda maior do que essa ainda seria aceitável.

A análise amortizada de árvores afuniladas, que será apresentada no **Capítulo 5**, garante que m operações consecutivas são executadas com custo temporal máximo $\theta(m \cdot \log n)$, mas essa garantia não exclui a possibilidade de uma única operação ter custo temporal $\theta(n)$. Portanto embora essa garantia de custo temporal não seja tão enfática quanto nos casos de árvores AVL, que apresentam custos temporais $\theta(\log n)$ no pior caso, o desempenho líquido de árvores afuniladas quando se considera uma sequência de operações é o mesmo exibido por árvores balanceadas.

4.7 Exemplos de Programação

4.7.1 Menor e Maior Chaves numa Árvore Binária de Busca

Problema: (a) Escreva uma função que retorne o elemento contendo a menor chave de uma tabela de busca implementada como árvore binária de busca. (b) Escreva uma função que retorne o elemento contendo a maior chave de uma tabela de busca implementada como árvore binária de busca.

Solução: Uma propriedade fundamental de árvores de busca é o fato de um caminhamento de ordem infixa resultar em acesso às chaves armazenadas em ordem crescente. Com base neste fato, pode-se concluir que a menor chave de uma árvore binária de busca está armazenada no nó mais à esquerda da árvore. A função `MenorChaveArvoreBB()` apresentada a seguir mostra como a menor chave de uma árvore binária de busca pode ser obtida. Essa função recebe como único parâmetro um ponteiro para a raiz da árvore e retorna o endereço do conteúdo que contém a menor chave, se a árvore não estiver vazia. Caso contrário, ela retorna **NULL**.

```
tCEP_Ind *MenorChaveArvoreBB(tArvoreBB p)
{
   tArvoreBB q; /* Apontará para o nó que armazena a menor chave */

      /* Se a árvore estiver vazia não há menor chave */
   if (!p)
      return NULL;

      /* A menor chave encontra-se no nó mais à esquerda da árvore de busca */
   while (p) {
      q = p; /* Faz q apontar para o nó para o qual p aponta */
      p = p->esquerda; /* Faz p apontar para o nó à esquerda */
   }
      /* Neste ponto, q aponta para o nó que armazena a menor chave */
   return &q->conteudo;
}
```

Utilizando um raciocínio análogo, pode-se obter o índice do registro que contém a maior chave armazenada numa árvore binária de busca, como faz a função `MaiorChaveArvoreBB()` a seguir.

```
tCEP_Ind *MaiorChaveArvoreBB(tArvoreBB p)
{
   tArvoreBB q; /* Apontará para o nó que armazena a maior chave */

      /* Se a árvore estiver vazia não há maior chave */
   if (!p)
      return NULL;

      /* A maior chave encontra-se no nó mais à direita da árvore de busca */
   while (p) {
      q = p; /* Faz q apontar para o nó para o qual p aponta */
```

```
      p = p->direita; /* Faz p apontar para o nó à direita */
   }
      /* Neste ponto, q aponta para o nó que armazena a maior chave */
   return &q->conteudo;
}
```

4.7.2 Exibindo em Ordem as Chaves de uma Árvore Binária de Busca

Problema: (a) Defina uma função que acessa em ordem crescente, as chaves contidas numa árvore binária de busca. (b) Defina uma função usada por um programa-cliente que chama a função definida no item (a) para escrever em arquivo os conteúdos dos nós de uma árvore binária de busca.

Solução de (a): Para acessar as chaves armazenadas numa árvore de busca binária em ordem crescente basta efetuar um caminhamento em ordem infixa na árvore, como faz a função `CaminhamentoInfixoBB()` apresentada adiante. Os parâmetros dessa função são:

- `arvore` (entrada) — ponteiro para a raiz da árvore sobre a qual será efetuado o caminhamento
- `op` (entrada) — ponteiro para a função que representa a operação de saída a ser efetuada sobre a informação contida em cada nó
- `stream` (entrada) — stream no qual o resultado da operação será escrito

```
void CaminhamentoInfixoBB( tArvoreBB arvore, tOperacao op, FILE *stream )
{
   if (arvore) {
         /* Caminha na subárvore esquerda */
      CaminhamentoInfixoBB(arvore->esquerda, op, stream);

      op(arvore->conteudo, stream); /* Visita a raiz */

         /* Caminha na subárvore direita */
      CaminhamentoInfixoBB(arvore->direita, op, stream);
   }
}
```

O tipo `tOperacao` do segundo parâmetro da função `CaminhamentoInfixoBB()` é definido como:

```
typedef void (*tOperacao) (tCEP_Ind, FILE*);
```

Solução de (b): A função `EscreveEmArquivoArvoreBB()` escreve em arquivo os conteúdos dos nós de uma árvore binária de busca e seu único parâmetro é um ponteiro para a raiz da árvore.

```
void EscreveEmArquivoArvoreBB(tArvoreBB arvore)
{
   FILE *stream;

      /* Tenta abrir o arquivo para escrita em modo texto */
   stream = fopen(NOME_ARQUIVO_CHAVES, "w");

      /* Se arquivo não foi aberto, aborta o programa */
   ASSEGURA(stream, "Impossivel criar arquivo para escrita de chaves");

   printf("\nEscrevendo tabela em arquivo...");

      /* Cabeçalho de apresentação */
   fprintf( stream, "   Chave   \tIndice\n"
                    "   =====   \t======\n");

      /* Apresenta cada estrutura numa linha separada */
   CaminhamentoInfixoBB( arvore, ExibeConteudoNoArvoreBB, stream );

   fclose(stream);
```

```
   printf("\n*** Resultado escrito no arquivo %s *** \n", NOME_ARQUIVO_CHAVES);
}
```

A função `ExibeConteudoNoArvoreBB()` usada como parâmetro por `EscreveEmArquivoArvoreBB()` na chamada de `CaminhamentoInfixoBB()` poderia ser definida como:

```
void ExibeConteudoNoArvoreBB(tCEP_Ind conteudo, FILE *stream)
{
   fprintf( stream, "%s\t%d\n", conteudo.chave, conteudo.valor );
}
```

4.7.3 Checando Árvores Binárias de Busca

Problema: (a) Escreva uma função que verifica se uma árvore binária é uma árvore binária *de busca*. O tipo de nó e o tipo de ponteiro para nó da árvore binária são aqueles usados na **Seção 4.1.2**. (b) Suponha que o tipo do campo `conteudo` seja **int** (em vez de `tCEP_Ind`, como nos exemplos anteriores). Escreva um programa que cria as árvores binárias de busca da **Figura 4–71** e verifica se essas árvores são legítimas árvores binárias de busca. [Note que a árvore da **Figura 4–71 (a)** deve ser aprovada no teste, mas isso não deve ocorrer com a árvore da **Figura 4–71 (b)**.]

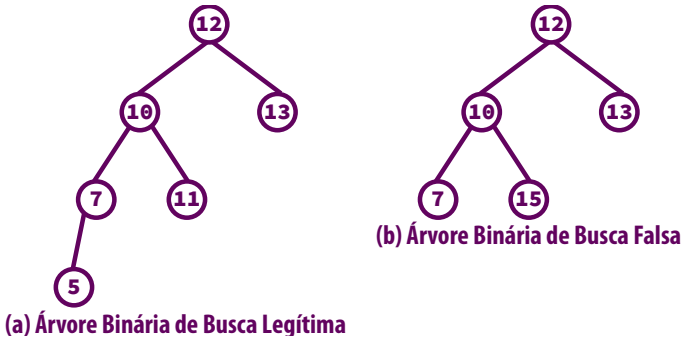


Figura 4–71: Checando Árvores Binárias de Busca

Solução de (a): Para tornar a função solicitada genérica (i.e., para torná-la independente do conteúdo efetivo de cada nó), é uma boa ideia incluir um parâmetro na função solicitada que compara os conteúdos dos nós da árvore. O tipo `tCompara` definido abaixo é aquele que programadores de C estão acostumados a usar em funções como **bsearch()** e **qsort()** da biblioteca padrão dessa linguagem.

```
typedef int (*tCompara) (const void *, const void *);
```

Existe um algoritmo tentador para resolver o problema proposto, mas que, de fato, não funciona (v. questão **32** na **Seção 4.8**). Um algoritmo que realmente funciona nesse caso é aquele que segue o **Teorema 4.1**. Ou seja, de acordo com esse teorema, se for efetuado um caminhamento em ordem infixa numa árvore binária e as chaves presentes nessa árvore forem visitadas em ordem crescente, então essa árvore é uma árvore binária de busca.

A função `EhArvoreBinDeBusca()` a seguir verifica se uma árvore binária é realmente uma árvore binária de busca efetuando um caminhamento em ordem infixa nessa árvore e verificando se as chaves são visitadas em ordem crescente. O primeiro parâmetro dessa função representa o endereço da raiz da árvore que se deseja testar e o segundo parâmetro é um ponteiro para uma função que compara os conteúdos efetivos de dois nós da árvore. A especificação de retorno dessa função de comparação deve ser compatível com aquela de **strcmp()** da biblioteca padrão de C.

```
int EhArvoreBinDeBusca(tNoArvoreBB *p, tCompara compara)
{
   static tNoArvoreBB *antecessor = NULL; /* Armazenará o antecessor de cada */
                                          /* nó visitado em ordem infixa     */
      /* Efetua um caminhamento em ordem infixa guardando */
      /* o valor do nó antecessor imediato do nó corrente */
   if (p) {
         /* Verifica se a subárvore esquerda é de busca */
      if (!EhArvoreBinDeBusca(p->esquerda, compara))
         return 0;

         /* Se o conteúdo do nó corrente for menor do que o conteúdo */
         /* do nó antecessor, não se  trata de árvore de busca       */
      if ( antecessor && compara(&p->conteudo, &antecessor->conteudo) < 0 )
         return 0;

      antecessor = p; /* O antecessor passa a ser o nó corrente */

         /* Verifica se a subárvore direita é de busca */
      return EhArvoreBinDeBusca(p->direita, compara);
   }

   return 1; /* Se ainda não houve retorno, a árvore foi aprovada */
}
```

Solução de (b): Primeiro, é necessário escrever uma função que compara o conteúdo de dois nós. Como foi dito acima, a especificação de retorno dessa função deve ser semelhante aquele de **strcmp**(), que faz parte da biblioteca padrão de C. No caso em que o tipo do campo `conteudo` é **int**, essa função é muito fácil de implementar, como faz a função `ComparaInts()`:

```
int ComparaInts(const void *e1, const void *e2)
{
   ASSEGURA(e1 && e2, "Elemento nulo recebido por ComparaInts()");

   return *(int *)e1 - *(int *)e2;
}
```

Agora já se pode escrever a função **main**() que irá compor o programa solicitado:

```
int main(void)
{
   tNoArvoreBB *raiz = NoNovo(12); /* Raiz da árvore da Figura (a) */

      /* Insere os demais nós na árvore da Figura (a) */
   raiz->esquerda = NoNovo(10);
   raiz->direita = NoNovo(13);
   raiz->esquerda->esquerda = NoNovo(7);
   raiz->esquerda->direita = NoNovo(11);
   raiz->esquerda->esquerda->esquerda = NoNovo(5);

      /* Testa se a árvore binária da Figura (a) é de busca */
   if (EhArvoreBinDeBusca(raiz, ComparaInts))
      printf("\n\t>>> A arvore da Figura (a) e' de busca");
   else
      printf("\n\t>>> A arvore da Figura (a) NAO e' de busca");

   DestroiArvoreBB(raiz); /* Destrói a árvore da Figura (a) */

      /* Cria a árvore da Figura (b) */
   raiz = NoNovo(12);
   raiz->esquerda = NoNovo(10);
```

```
   raiz->direita = NoNovo(13);
   raiz->esquerda->esquerda = NoNovo(7);
   raiz->esquerda->direita = NoNovo(15);

      /* Testa se a árvore binária da Figura (b) é de busca */
   if (EhArvoreBinDeBusca(raiz, ComparaInts))
      printf("\n\t>>> A arvore da Figura (b) e' de busca");
   else
      printf("\n\t>>> A arvore da Figura (b) NAO e' de busca");

   return 0;
}
```

A função **main()** definida acima, de fato, constrói as árvores solicitadas e verifica que a árvore da **Figura 4–71 (a)** é uma árvore binária de busca e que a árvore da **Figura 4–71 (b)** não o é. Essa função chama a função `NoNovo()` para criar cada nó da árvore em questão. Essa última função é semelhante à função `ConstroiNoArvoreBB()` apresentada na **Seção 4.1.2**. A referida função **main()** também chama `DestroiArvoreBB()` para liberar o espaço ocupado pela primeira árvore construída pelo programa. A função `DestroiArvoreBin()`, discutida no **Capítulo 12** do **Volume 1**, é semelhante à função `DestroiArvoreBB()`.

4.7.4 Conferindo Balanceamento AVL

Problema: Escreva uma função em C que recebe como parâmetro um ponteiro para a raiz de uma árvore binária e retorna `1`, se a referida árvore satisfizer o balanceamento AVL, ou `0`, em caso contrário.

Solução: A função `EhArvoreAVL()`, apresentada a seguir, atende aquilo que foi solicitado. É importante notar que essa função não testa se a árvore binária cuja raiz é recebida como parâmetro é uma árvore *de busca*. Ela apenas testa se essa árvore apresenta um balanceamento que satisfaz o critério AVL de balanceamento.

```
int EhArvoreAVL(tNoArvoreBB *raiz)
{
   int altEsquerda, /* Altura de uma subárvore esquerda */
       altDireita;  /* Altura de uma subárvore direita  */

      /* Se a árvore estiver vazia, ela é AVL */
   if (!raiz)
      return 1;

      /* Obtém as alturas das subárvores esquerda e direita */
   altEsquerda = AlturaArvoreBB2(raiz->esquerda);
   altDireita = AlturaArvoreBB2(raiz->direita);

      /* A árvore será AVL se sua raiz tiver balanceamento  */
      /* AVL e suas subárvores esquerda e direita forem AVL */
   if ( ABS(altEsquerda - altDireita) <= 1 && EhArvoreAVL(raiz->esquerda) &&
        EhArvoreAVL(raiz->direita) )
    return 1;

   return 0; /* Se ainda não houve retorno, a árvore não é AVL */
}
```

A função `EhArvoreAVL()` chama a função `AlturaArvoreBB2()` que calcula a altura de uma árvore (ou subárvore) binária e é definida como:

```
int AlturaArvoreBB2(tNoArvoreBB *raiz)
{
      /* Se a árvore estiver vazia, sua altura é zero */
   if (!raiz)
```

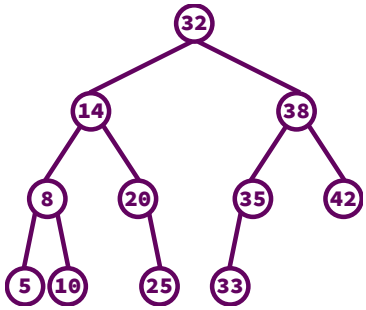
```
      return 0;

      /* Se a árvore não estiver vazia, calcula-se a */
      /* maior altura de suas subárvore e soma-se um */
   return 1 + MAIOR_VALOR( AlturaArvoreBB2(raiz->esquerda),
                           AlturaArvoreBB2(raiz->direita) );
}
```

As macros `ABS` e `MAIOR_VALOR` usadas pelas funções acima calculam o valor absoluto e o maior valor dentre dois números, respectivamente, e são fáceis de serem implementadas.

4.8 Exercícios de Revisão

Árvores Binárias Ordinárias de Busca (Seção 4.1)

1. Como uma chave é inserida numa árvore binária ordinária de busca?
2. Descreva o algoritmo de busca numa árvore binária de busca.
3. (a) Mostre que o fato de a menor chave de uma árvore binária ser aquela que se encontra mais à esquerda na árvore não implica que ela seja uma árvore binária de busca. (b) Mostre que o fato de a maior chave de uma árvore binária ser aquela que se encontra mais à direita na árvore não implica que ela seja uma árvore binária de busca.
4. (a) Como se encontra o sucessor imediato de um nó de uma árvore binária de busca que possui dois filhos? (b) Como se encontra o antecessor imediato de um nó de uma árvore binária de busca que possui dois filhos?
5. (a) Qual é o único nó de uma árvore binária de busca que não possui antecessor? (b) Qual é o único nó de uma árvore binária de busca que não possui sucessor?
6. (a) Mostre que, numa árvore binária de busca, se um nó possui filho direito, seu sucessor imediato não pode ter filho esquerdo. (b) Mostre que, numa árvore binária de busca, se um nó possui filho esquerdo, seu antecessor imediato não pode ter filho direito.
7. (a) Se um nó de uma árvore binária de busca não possui filho direito, como se encontra seu sucessor imediato? (b) Se um nó de uma árvore binária de busca não possui filho esquerdo, como se encontra seu antecessor imediato?
8. Suponha que os nós de uma árvore binária de busca contenham apenas chaves inteiras cujos valores são: *28*, *37*, *23*, *55*, *46*, *24*, *10* e *13*. Desenhe essa árvore quando os nós são inseridos nessa mesma ordem.
9. Descreva os casos de remoção de um nó de uma árvore binária ordinária de busca.
10. De acordo com o que foi visto na **Seção 4.1**, quando o nó a ser removido de uma árvore binária de busca possui dois filhos, ele é substituído por seu sucessor imediato em ordem infixa. Existe alguma alternativa para esse procedimento que seja igualmente eficiente?
11. Se um nó de uma árvore binária de busca não possuir filhos ele tem sucessor? Se for o caso, onde ele se encontra?
12. (a) O que é uma árvore binária inclinada à esquerda? (b) Sob que condição é criada uma árvore binária de busca inclinada à esquerda?
13. (a) O que é uma árvore binária inclinada à direita? (b) Sob que condição é criada uma árvore binária de busca inclinada à direita?
14. Desenhe a árvore binária de busca resultante da inserção de um nó contendo a chave *11* na árvore da figura abaixo.



15. Desenhe a árvore binária de busca resultante da remoção do nó contendo a chave *14* na árvore ilustrada na figura do exercício **14**.

16. Apresente a árvore resultante da remoção do nó com conteúdo *35* na árvore ilustrada na figura do exercício **14**.

17. Apresente a árvore resultante da remoção da raiz da árvore ilustrada na figura do exercício **14**.

18. Desenhe todas as árvores binárias de busca possíveis que contenham as chaves *1*, *2* e *3*.

19. Desenhe a árvore binária de busca criada pelo seguinte programa:

```
#include <stdlib.h>

typedef struct no {
           char       conteudo;
           struct no *filho[2];
        } tNo, *tArvore;

int InsereNoListaSE( tArvore *arvore, char conteudo )
{
   tArvore p = *arvore, q = NULL, pNovoNo;

   while (p) {
      if (conteudo == p->conteudo)
         return 1;

      q = p;

      p = p->filho[conteudo < p->conteudo];
   }

   pNovoNo = malloc(sizeof(*pNovoNo));

   pNovoNo->conteudo = conteudo;
   pNovoNo->filho[0] = pNovoNo->filho[1] = NULL;

   if (!q)
      *arvore = pNovoNo;
   else
      q->filho[conteudo < q->conteudo] = pNovoNo;

   return 0;
}
void CriaArvore(tArvore *arvore, char ar[], int inf, int sup)
{
  for (int i = inf; i < sup; ++i)
      InsereNoListaSE(arvore, ar[i]);
}
```

CONTINUA

CONTINUAÇÃO

```
int main(void)
{
   char alfabeto[] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G',
                       'H', 'I', 'J', 'K', 'L', 'M', 'N',
                       'O', 'P', 'Q', 'R', 'S', 'T', 'U',
                       'V', 'W', 'X', 'Y', 'Z' };

   tArvore raiz = NULL;

   CriaArvore( &raiz, alfabeto, 0,
               sizeof(alfabeto)/sizeof(alfabeto[0]) );

   return 0;
}
```

20. Se a função `CriaArvore()` do programa do exercício **19** for substituída pela função `CriaArvore2()` a seguir, qual será a árvore de busca criada pelo programa?

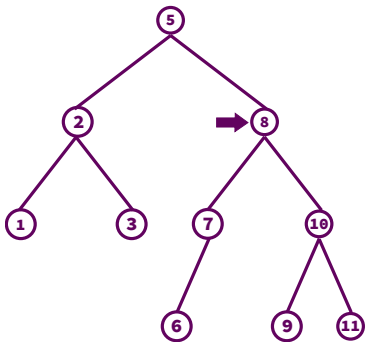
```
void CriaArvore2(tArvore *arvore, char ar[], int inf, int sup)
{
   int meio;

   if (inf <= sup) {
      meio = (inf + sup)/2;

      InsereNoListaSE(arvore, ar[meio]);

      CriaArvore(arvore, ar, inf, meio - 1);
      CriaArvore(arvore, ar, meio + 1, sup);
   }
}
```

21. Após estudar listas com saltos e árvores binárias de busca, um candidato a cientista maluco decidiu adaptar o método de inserção dessas listas para essas árvores. Seu algoritmo de inserção segue os seguintes passos:

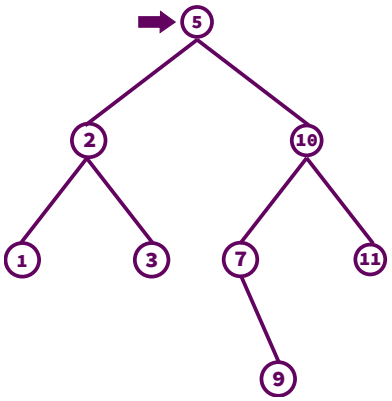
1. Se a árvore estiver vazia, crie um novo nó contendo a chave, torne esse nó raiz da árvore e encerre.
2. Caso contrário, lance uma moeda.
3. Se o resultado do lançamento for cara, insira a chave na subárvore esquerda.
4. Se o resultado do lançamento for coroa, insira a chave na subárvore direita.

O cientista batizou sua criação de *CrazyTree*. (a) Como é possível efetuar uma operação de busca numa *CrazyTree*? (b) Qual é o custo temporal dessa operação de busca? (c) Sugira um meio de efetuar remoções numa *CrazyTree*. (d) Faz sentido balancear uma *CrazyTree*? Explique seu raciocínio.

22. Quando um nó a ser removido possui dois filhos, ele pode ser substituído por seu sucessor imediato (como foi visto na **Seção 4.1.2**) ou por seu antecessor imediato. Mostre como o nó contendo a chave *8* na figura a seguir é substituído dessa última maneira.



23. Mostre como o nó contendo a chave *5* na árvore binária de busca a seguir é removida de modo que ele seja substituído por seu (a) antecessor imediato e (b) sucessor imediato.



24. (a) Mostre que, quando se substitui um nó de uma árvore binária de busca que possui dois filhos por seu sucessor imediato em ordem infixa, o resultado é uma árvore binária de busca. (b) Mostre que, quando se substitui um nó de uma árvore binária de busca que possui dois filhos por seu antecessor imediato em ordem infixa, o resultado é uma árvore binária de busca.

25. Apresente cinco permutações das chaves *1*, *2*, *3*, *5*, *7*, *9*, *10* e *11* tais que quando essas chaves forem inseridas numa árvore binária de busca inicialmente vazia em qualquer dessas ordenações o resultado seja a árvore da questão **23**.

26. Um algoritmo bastante comum, porém equivocado, utilizado para verificar se uma árvore binária é uma árvore (binária) de busca é implementado pela função `EhArvoreDeBuscaErrada()` a seguir. (a) Qual é a abordagem usada por essa função? (b) Mostre que essa função não funciona apresentando uma árvore binária que não é uma árvore binária de busca, mas que, mesmo assim, passa no teste efetuado pela função `EhArvoreDeBuscaErrada()`.

```
int EhArvoreDeBuscaErrada(tNo *p)
{
   int compEsquerda,
       compDireita;

      /* Se a árvore estiver vazia, ela é de busca */
   if (!p)
      return 1;

      /* Compara o conteúdo do nó corrente com os   */
      /* conteúdos de seus filhos esquerdo e direito */
   compEsquerda = !p->esquerda || p->esquerda->conteudo < p->conteudo;

   compDireita = !p->direita || p->direita->conteudo > p->conteudo;

      /* Se o nó corrente e seus filhos satisfazem o    */
      /* critério, a árvore será de busca se suas sub-  */
      /* árvores esquerda e direita também satisfizerem */
   if (compEsquerda && compDireita)
     return EhArvoreDeBuscaErrada(p->esquerda) && EhArvoreDeBuscaErrada(p->direita);
   else
      return 0;

   return 0;
}
```

27. Numa **remoção negligente**, os nós a serem removidos são mantidos na árvore e apenas marcados como removidos. Quais são as vantagens e desvantagens dessa abordagem?

28. Suponha que se tenha uma estimativa antecipada de quão frequentemente chaves de busca são acessadas numa árvore binária. As chaves devem ser inseridas na árvore em ordem crescente ou decrescente de provável frequência de acesso? Explique sua resposta.

29. Desenhe todas as árvores binárias de busca estruturalmente diferentes que resultem quando n chaves são inseridas numa árvore inicialmente vazia, quando $2 \leq n \leq 5$.

30. Para que servem árvores binárias de busca ordinárias?

31. Apresente o custo temporal para as operações de busca, inserção e remoção em árvores binárias ordinárias de busca no melhor caso, no pior caso e no caso médio.

32. Mostre que o fato de o menor valor armazenado numa árvore binária encontrar-se na folha mais à esquerda da árvore não implica necessariamente que essa árvore seja uma árvore binária de busca.

33. Como as chaves de uma árvore binária de busca podem ser armazenadas em ordem crescente num array?

34. Se um nó de uma árvore binária não possui filho esquerdo, onde se encontra o antecessor imediato desse nó?

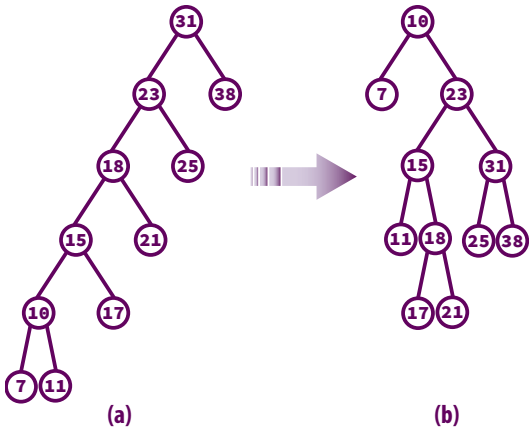
Rotações em Árvores Binárias de Busca (Seção 4.2)

35. (a) Descreva a operação de rotação esquerda simples de um nó. (b) Descreva a operação de rotação direita simples de um nó.

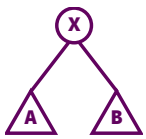
36. (a) Que condição deve satisfazer o filho esquerdo do nó sobre o qual incide uma rotação direita? (b) Que condição deve satisfazer o filho direito do nó sobre o qual incide uma rotação esquerda?

37. Que propriedades de uma árvore binária de busca são preservadas após uma operação de rotação?

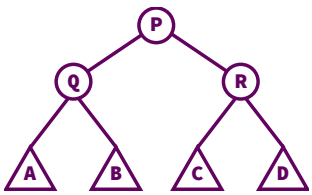
38. Quais são as rotações necessárias para transformar a árvore da figura (a) na árvore da figura (b) abaixo?



39. O que significam os triângulos rotulados A e B na figura abaixo?



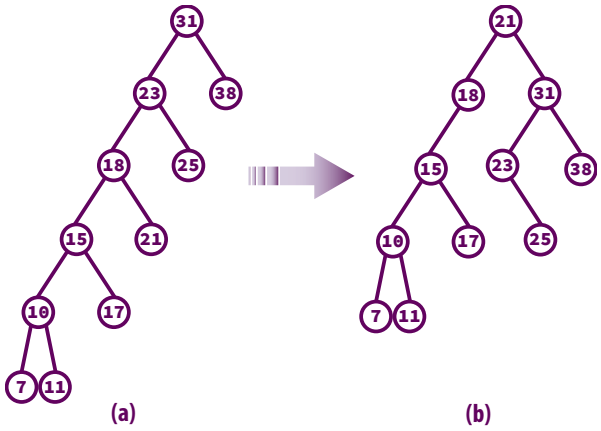
40. Qual é a relação entre as chaves da árvore na figura a seguir?



41. (a) Desenhe a árvore resultante da rotação esquerda do nó *R* na árvore da figura do exercício **40**. (b) Desenhe a árvore resultante da rotação direita do nó *Q* na árvore da figura do exercício **40**.

42. Qual é o custo temporal de uma rotação de nó numa árvore binária de busca?

43. Quais são as rotações necessárias para transformar a árvore da figura (a) na árvore da figura (b) abaixo?



Balanceamento de Árvores Binárias de Busca (Seção 4.3)

44. O que é uma árvore binária perfeitamente balanceada?

45. (a) Qual é o número mínimo de nós de uma árvore perfeitamente balanceada com profundidade (altura) *p*? (b) Qual é o número máximo de nós de uma árvore perfeitamente balanceada com profundidade *p*?

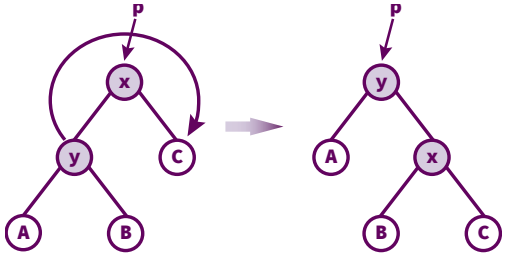
46. (a) Qual é o número mínimo de folhas de uma árvore perfeitamente balanceada de altura *p*? (b) o número máximo de folhas de uma árvore perfeitamente balanceada com altura *p*?

47. Por que árvores binárias de busca balanceadas são desejáveis?

48. Apresente um exemplo de árvore binária de busca perfeitamente balanceada que se torna degenerada após uma sequência de remoções de nós.

49. (a) Qual é a vantagem das árvores perfeitamente balanceadas? (b) Qual é a desvantagem das árvores perfeitamente balanceadas?

50. Sejam *a*, *b* e *c* nós arbitrários nas subárvores *A*, *B* e *C*, respectivamente, na árvore à esquerda da figura a seguir. Como as alturas de *a*, *b* e *c* mudam quando uma rotação direita é executada no nó *y* dessa figura?



Árvores AVL (Seção 4.4)

51. O que é uma árvore AVL?

52. (a) Apresente um exemplo de árvore AVL que também seja perfeitamente balanceada. (b) Dê exemplo de uma árvore AVL que não seja perfeitamente balanceada.

53. Que vantagem árvores AVL oferecem com relação a árvores binárias perfeitamente balanceadas?

54. O que é balanceamento de um nó de uma árvore AVL?

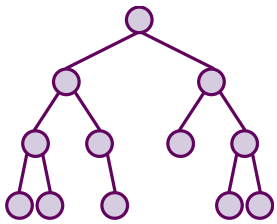
55. Mostre que cada nó de uma árvore AVL tem balanceamento *–1*, *0* ou *1*.

56. Apresente exemplos das possíveis situações nas quais uma árvore AVL pode se tornar desbalanceada após uma operação de inserção.

57. Para que servem operações de rotação em árvores AVL?

58. (a) Descreva a rotação dupla esquerda-direita. (b) Descreva a rotação dupla direita-esquerda. (c) Em que situações cada uma dessas rotações é utilizada?

59. Suponha que a figura abaixo represente o arcabouço de uma árvore binária de busca. (a) Qual é o balanceamento de cada nó desta árvore considerando a definição de balanceamento para árvores AVL apresentada na **Seção 4.4**? (b) Essa árvore de busca é uma árvore AVL?

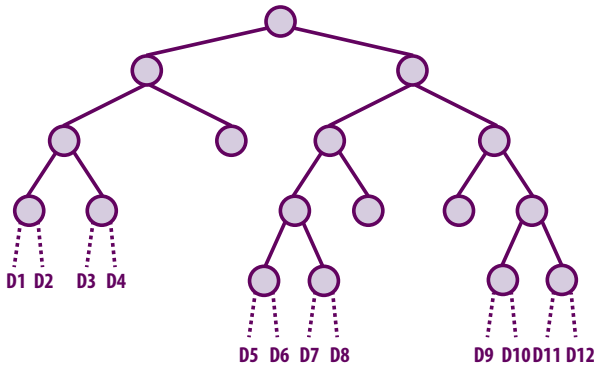


60. Considerando a árvore da questão **59**, em que posições podem ser inseridos nós nessa árvore de modo que ela continue balanceada segundo o critério AVL?

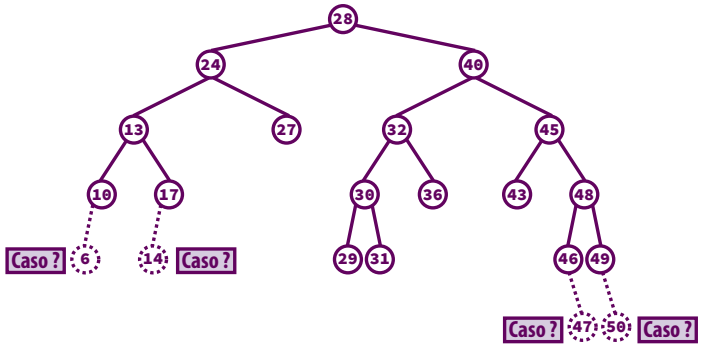
61. Quais propriedades de uma árvore AVL são preservadas em qualquer das operações de rotação apresentadas?

62. Descreva as operações de rebalanceamento que podem ocorrer após a inserção de um nó numa árvore AVL.

63. Suponha que a figura a seguir represente o arcabouço de uma árvore AVL. Quando algum dos nós rotulados como D_1, D_2, ..., D_{12} é inserido nessa árvore ocorre um desbalanceamento. Identifique o caso de desbalanceamento (esquerda-esquerda, esquerda-direita etc.) que a inserção de cada um desses nó provoca nessa árvore e informe qual é a respectiva correção (rotação ou rotações) que cada um desses desbalanceamentos requer.



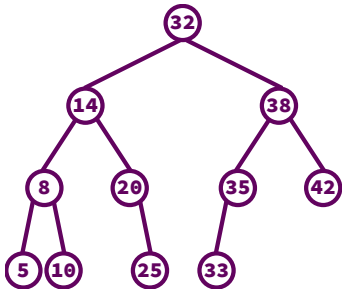
64. Considerando a árvore AVL da figura a seguir, em que caso de desbalanceamento devido a inserção se encaixam as inserções dos nós contendo as chaves: (a) *6* (b) *14* (c) *47* e (d) *50*?



65. Numa árvore binária de Fibonacci de ordem n, se $n = 0$ ou $n = 1$, a árvore consiste num único nó e, se $n > 1$, a árvore consiste numa raiz, tendo a árvore de Fibonacci de ordem $n - 1$ como sua subárvore esquerda e a árvore de Fibonacci de ordem $n - 2$ como sua subárvore direita. Uma árvore binária de Fibonacci pode ser considerada o pior caso de árvore AVL, pois ela possui o menor número de nós dentre todas as árvores AVL de altura a. Desenhe árvores de Fibonacci para alturas iguais a *1*, *2*, *3* e *4*.

66. Qual é o custo de temporal de uma (a) busca, (b) inserção e (c) remoção numa árvore AVL?

67. (a) Apresente o balanceamento de cada nó da árvore ilustrada na figura abaixo. (b) Essa árvore é AVL?



68. Apresente a árvore AVL resultante da inserção de um nó com chave igual a *11* na árvore ilustrada na figura do exercício **67**.

69. Apresente a árvore AVL resultante da remoção do nó contendo a chave *35* na árvore ilustrada na figura do exercício **67**.

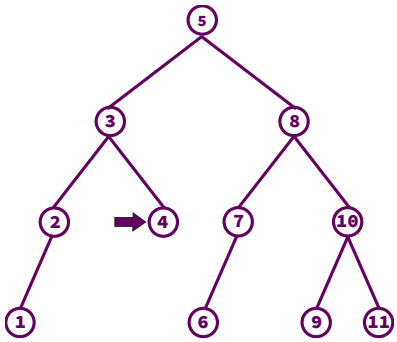
70. Apresente a árvore AVL resultante da remoção da raiz da árvore ilustrada na figura do exercício **67**.

71. A ordem com que um conjunto de chaves é inserido numa árvore AVL influencia a eficiência das operações básicas sobre essa árvore?

72. Suponha que os nós de uma árvore AVL contenham apenas chaves inteiras cujos valores são: *10*, *20*, *30*, *40*, *50*, *60*, *70* e *80*. Desenhe essa árvore quando os nós são inseridos nessa mesma ordem.

73. (a) Uma árvore AVL contendo três nós pode ser inclinada à esquerda (ou à direita)? (b) Uma árvore AVL contendo quatro nós pode ser inclinada à esquerda (ou à direita)?

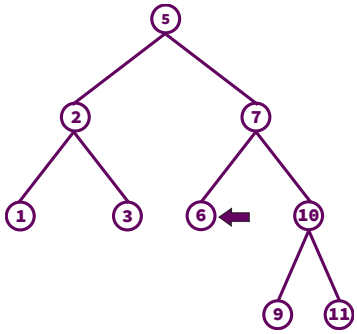
74. A árvore AVL da figura abaixo torna-se desbalanceada após a remoção do nó que contém a chave *4*. Mostre como a árvore resultante dessa remoção é rebalanceada.



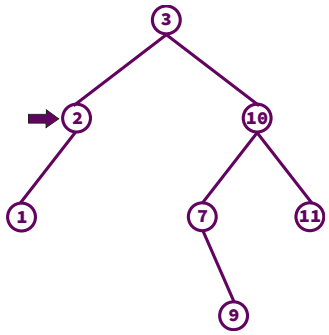
75. Desenhe a árvore AVL resultante da inserção em ordem alfabética das letras do alfabeto latino.

76. (a) Apresente representações gráficas de árvores AVL contendo o número mínimo de nós quando a altura da árvore é *1*, *2*, *3* e *4*. (b) Qual é o número mínimo de nós de uma árvore AVL com altura igual a *5*?

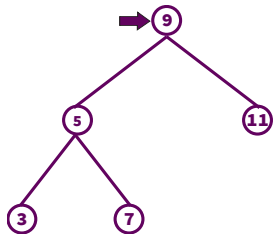
77. A árvore AVL da figura abaixo torna-se desbalanceada após a remoção do nó que contém a chave *6*. Mostre como a árvore resultante dessa remoção é rebalanceada.



78. A remoção do nó com a chave *2* na figura abaixo é relativamente fácil, mas requer rebalanceamento um tanto complexo. Apresente a árvore resultante dessa remoção com o subsequente rebalanceamento.



79. A remoção do nó contendo a chave *9* na árvore AVL da figura abaixo requer rebalanceamento. Apresente a árvore resultante dessa remoção com o subsequente rebalanceamento.



Árvores Binárias Afuniladas (Seção 4.5)

80. O que é uma árvore afunilada?

81. Em que diferem árvores afuniladas e árvores AVL?

82. Suponha que os nós de uma árvore afunilada contenham apenas chaves inteiras cujos valores são: *10*, *20*, *30*, *40*, *50*, *60*, *70* e *80*. Desenhe essa árvore quando os nós são inseridos nessa mesma ordem.

83. Apresente a nova configuração da árvore afunilada obtida no exercício **82** quando se efetua uma busca pela chave *40*.

84. Apresente a nova configuração da árvore afunilada obtida no exercício **82** quando se efetua uma busca pela chave *32*.

85. Apresente a nova configuração da árvore afunilada obtida no exercício **82** quando o nó contendo a chave *50* é removido.

86. A reconfiguração de uma árvore AVL nunca ocorre durante uma operação de busca. Por que, então, uma árvore afunilada pode ser reconfigurada durante tal operação?

87. Qual é a configuração de nós (avô, pai e filho) de uma árvore afunilada que requer cada um dos seguintes afunilamentos?
(a) Zig-zag
(b) Zig-zig
(c) Zag-zag
(d) Zag-zig

88. Em que diferem rotações duplas em árvores AVL e rotações duplas em árvores afuniladas?

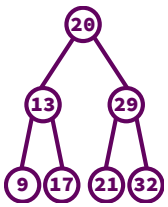
89. (a) Se uma chave não for encontrada durante uma operação de busca numa árvore afunilada, ocorre afunilamento? (b) Em caso afirmativo, como é esse afunilamento?

90. (a) Descreva o afunilamento que ocorre durante a inserção de um nó numa árvore afunilada. (b) Descreva o afunilamento que ocorre durante a remoção de um nó de uma árvore afunilada.

91. (a) O que significa afunilar uma chave que faz parte de uma árvore afunilada? (b) O que significa afunilar uma chave que não faz parte de uma árvore afunilada?

92. (a) Construa graficamente a árvore afunilada resultante da inserção das seguintes chaves: *1*, *2*, *3*, *4* e *5*, inseridas nessa ordem. (b) Desenhe a árvore afunilada resultante da remoção da chave *1* da árvore obtida no item (a).

93. Considere a árvore binária de busca da figura abaixo. Se todos os nós dessa árvore forem acessados em ordem crescente de suas chaves e cada acesso afunilar o nó acessado, qual será o formato da árvore resultante?



94. Se todos os nós da árvore do exercício **93** forem acessados em ordem decrescente de suas chaves e cada acesso afunilar o nó acessado, qual será o formato da árvore resultante?

95. Qual é a relação entre árvore afunilada e localidade de referência?

96. Apresente exemplos práticos de uso de árvores afuniladas.

97. (a) Descreva todas as rotações necessárias para a obtenção da árvore da **Figura 4–62 (b)** a partir da árvore da **Figura 4–62 (a)**. (b) Descreva todas as rotações necessárias para a obtenção da árvore da **Figura 4–63 (b)** a partir da árvore da **Figura 4–63 (a)**.

98. (a) O que é afunilamento ascendente? (b) O que é afunilamento descendente? (c) Por que afunilamento descendente é mais fácil de implementar?

99. Quais são os custos temporais no pior caso das operações de busca, inserção e remoção em árvores afuniladas?

100. (a) Efetue o afunilamento ascendente apresentado na **Figura 4–60** usando afunilamento descendente. (b) O resultado obtido é o mesmo?

Comparando Árvores Binárias de Busca (Seção 4.6)

101. Por que o resultado de um caminhamento em ordem infixa de uma árvore binária de busca é o mesmo quer ela seja ordinária, AVL ou afunilada?

102. Apresente uma comparação entre árvores AVL e árvores afuniladas.

103. Em que situação você escolheria árvore afunilada para implementar uma tabela de busca em detrimento a árvore AVL?

104. Em que situação você escolheria árvore AVL para implementar uma tabela de busca em detrimento a árvore afunilada?

Exemplos de Programação (Seção 4.7)

105. Qual é o papel desempenhado pela função `ExibeConteudoNoArvoreBB()` no exemplo da **Seção 4.7.2**?

106. Como se verifica se uma árvore binária pode ser classificada como uma árvore binária de busca?

107. (a) Qual é o papel desempenhado pela variável `antecessor` na implementação da função `EhArvoreDeBusca()` apresentada na **Seção 4.7.3**? (b) Por que essa variável precisa ter duração fixa?

108. Como se verifica se uma árvore binária pode ser classificada como uma árvore AVL?

109. No pior caso, qual é o custo temporal da operação de checagem de árvores binárias de busca apresentada da na **Seção 4.7.3**?

110. No pior caso, qual é o custo temporal da operação que encontra a menor (ou maior) chave de uma árvore binária de busca?

111. Qual é o custo temporal da operação que escreve as chaves de uma árvore binária ordenadas em ordem crescente?

112. Qual é o custo temporal da operação que verifica se uma árvore binária tem balanceamento AVL apresentada da na **Seção 4.7.4**?

4.9 Exercícios de Programação

EP4.1 Escreva uma função em C que determine se uma árvore binária é perfeitamente balanceada.

EP4.2 Suponha que as chaves armazenadas nos nós de uma árvore binária sejam do tipo **int**. Escreva uma função recursiva que exibe o conteúdo de cada ancestral de um nó cujo endereço essa função recebe como parâmetro.

EP4.3 Suponha que o conteúdo armazenado em cada nó de uma árvore binária seja do tipo **int**. Escreva uma função que retorna o número de nós que apresentam um valor menor do que o valor que ela recebe como parâmetro.

EP4.4 Escreva uma função recursiva que armazena em ordem crescente numa lista simplesmente encadeada os conteúdos efetivos dos nós de uma árvore binária de busca. Suponha que o conteúdo efetivo de cada nó da árvore seja do tipo **int**.

EP4.5 Escreva uma função em C que retorne o endereço do nó que contém a i-ésima menor chave armazenada numa árvore binária de busca, supondo que as chaves são do tipo **int**.

EP4.6 Suponha que as chaves armazenadas numa árvore binária de busca sejam do tipo **int**. Escreva uma função que encontra o piso de uma chave (possivelmente) armazenada numa árvore desse tipo.

EP4.7 Suponha que as chaves armazenadas numa árvore binária de busca sejam do tipo **int**. Escreva uma função que encontra o teto de uma chave (possivelmente) armazenada numa árvore desse tipo.

EP4.8 Suponha que as chaves armazenadas numa árvore binária de busca sejam do tipo **int**. Escreva uma função que retorna uma lista encadeada contendo todas as chaves que se encontram entre os valores inteiros `c1` e `c2`.

EP4.9 Escreva uma função que cria uma árvore AVL com o número mínimo de nós para uma determinada altura recebida como parâmetro. O conteúdo de cada deve ser um número inteiro gerado aleatoriamente.

EP4.10 Conforme foi visto no texto, se um array de chaves ordenadas for usado para criar uma árvore binária de busca inserindo-as na ordem em que se encontram, a árvore binária resultante será inclinada à esquerda (se as chaves estiverem em ordem decrescente) ou à direita (se as chaves estiverem em ordem

crescente). Implemente uma função que recebe como parâmetro de entrada um array de chaves ordenadas e utiliza uma abordagem semelhante a uma de busca binária de tal modo que a árvore binária de busca criada por essa função seja razoavelmente balanceada.

EP4.11 Reimplemente a função `RemoveArvoreBB()` apresentada na **Seção 4.1.2** de modo que quando um nó a ser removido possui dois filhos ele seja substituído por seu antecessor imediato em ordem infixa (em vez de ser substituído pelo sucessor imediato em ordem infixa, como ocorre na **Seção 4.1.2**).

EP4.12 Dadas duas árvores binárias de busca A e B tais que as chaves armazenadas em A são menores do que qualquer chave armazenada em B, uma maneira de unir essas duas árvores de modo a obter uma única árvore binária de busca é seguindo o seguinte procedimento:

1. Afunile o nó que contém a maior chave da árvore A. Como esse nó contém a maior chave da árvore A, ele não possui filho direito.
2. Torne raiz da árvore B o filho direito da raiz de A.

Crie uma função em C que implementa o procedimento descrito acima.

EP4.13 Dada uma árvore binária de busca A e um nó contendo uma chave c (**pivô**), é possível dividir essa árvore em duas outras binárias de busca B e C seguindo o seguinte procedimento:

1. Afunile o nó contendo a chave c. Desse modo, todas as chaves na subárvore esquerda da nova raiz da árvore A serão menores do que c e todas as chaves na subárvore direita da nova raiz dessa árvore serão maiores do que c.
2. Faça o ponteiro que representará a árvore B apontar para a nova raiz mencionada.
3. Faça o ponteiro que representa a árvore C apontar para o filho direito da árvore B.
4. Torne nulo o filho direito de B.

EP4.14 Supondo que c é a chave de um novo nó a ser inserido numa árvore afunilada, uma alternativa para o algoritmo de inserção implementado pela função `InsereArvoreFunil()` é a seguinte:

1. Encontre o nó contendo a maior chave que é menor do que ou igual à chave c (i.e., o nó contendo o piso de c). Se o piso de c for igual a c, encerre, visto que a chave é considerada primária.
2. Divida a árvore original conforme foi descrito no exercício **EP4.13** tendo como pivô o nó encontrado no passo anterior. A árvore A obtida nessa divisão será aquela contendo as menores chaves enquanto a árvore B será aquela contendo as maiores chaves.
3. Crie um novo nó e use a operação descrita no exercício **EP4.12** para tornar A o filho esquerdo e B o filho direito do novo nó.

Crie uma função em C que implementa o procedimento descrito acima.

EP4.15 (a) Escreva uma função em C que efetua buscas numa árvore binária usando caminhamento infixo. (b) Qual é o custo temporal dessa função? (c) Qual é o custo espacial dessa função? (d) Em que situação tal função se faz necessária?

EP4.16 Suponha que uma árvore binária de busca armazena chaves secundárias. Escreva uma função que retorna o número de nós nessa árvore com chaves iguais a uma chave recebida como parâmetro.

EP4.17 Suponha que uma árvore binária de busca armazena chaves secundárias. Implemente uma função de remoção para essa árvore que remove todos os nós na árvore que tenha chaves iguais a uma chave recebida como parâmetro.

EP4.18 Escreva uma função semelhante àquela apresentada na **Seção 4.7.4** que, além de verificar seus nós possuem balanceamento AVL, também verifica que ela é uma árvore de busca.

EP4.19 Implemente uma tabela de busca como um TAD que represente árvores binárias de busca usando as seguintes definições de tipos:

```
typedef enum {ESQUERDA = 0, DIREITA = 1} tDirecao;

typedef struct noBB2 {
           tConteudo     conteudo; /* Conteúdo do nó */
           struct noBB2 *filho[2]; /* Os filho do nó  */
        } tNoBB2;

typedef struct {
           tNoBB2 *raiz;  /* Raiz da árvore */
           int     nNos; /* Número de nós da árvore */
        } tArvoreBB2;
```