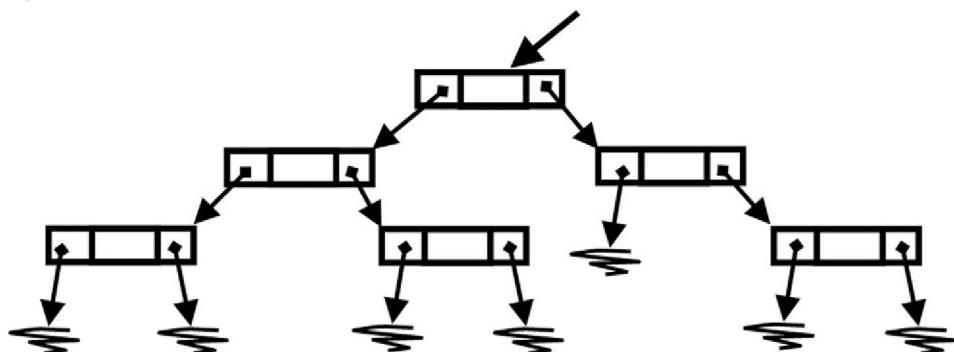


Capítulo 8

Árvores



Seus objetivos neste capítulo

- Entender o conceito, a nomenclatura e a representação usual da estrutura de armazenamento denominada Árvore, e de um tipo especial de Árvore denominada Árvore Binária de Busca — ABB.
- Desenvolver habilidade para elaborar algoritmos sobre Árvores Binárias de Busca e sobre Árvores em geral.
- Conhecer aplicações e a motivação para o uso de Árvores, entender as situações em que o seu uso é pertinente.
- Iniciar o desenvolvimento do seu jogo referente ao Desafio 4.

8.1 Árvores: conceito e representação

A [Figura 8.1](#) mostra a representação usual de uma estrutura de armazenamento denominada Árvore, com as seguintes características:

- Cada Árvore possui um **Nó Raiz** e possivelmente várias **Subárvores**. Na [Figura 8.1](#), um ponteiro denominado Raiz está apontando para o Nó Raiz, que contém o valor '1'. As Subárvores do Nó Raiz estão destacadas com círculos pontilhados. A Subárvore C, mais à direita, é composta pelos Nós de valores '4' e '11'.
- Cada Subárvore também pode ser considerada uma Árvore (e, nesse ponto, a definição passa a ser **recursiva**).
- Os Nós destacados com fundo cinza ('5', '7', '8', '10', '11', '12', '13' e '14') são chamados de **Folhas**, ou **Nós Terminais**, pois não possuem Subárvores.
- Podemos dizer que os Nós com valores '2', '3' e '4' são **Filhos** do Nó com valor '1'; podemos também dizer que o Nó de valor '1' é **Pai** dos Nós de valor '2', '3' e '4'.

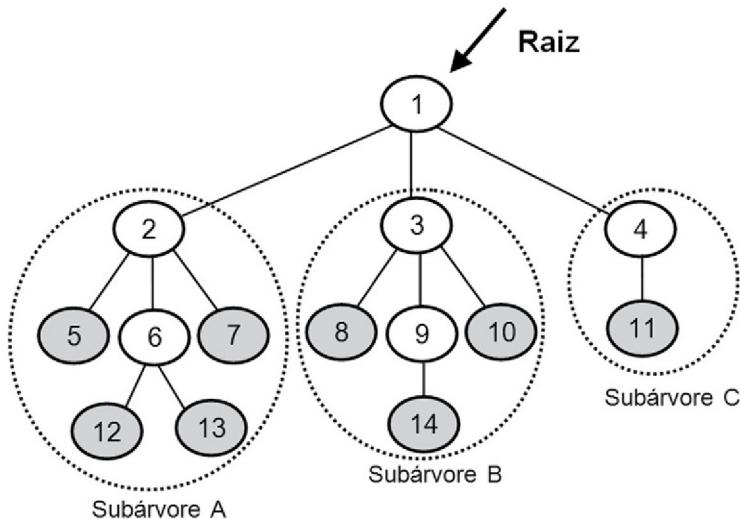


Figura 8.1 Representação usual de uma Árvore.

- A **Altura** de uma Árvore é definida pelo número de níveis em que os seus Nós estão distribuídos. No exemplo da [Figura 8.1](#), a Altura da Árvore como um todo é 4. No Nível 1 temos o Nó de valor '1'; no Nível 2 temos '2', '3' e '4'; no Nível 3 temos os Nós de valores '5' a '11'; no Nível 4 da Árvore, temos os Nós de valores '12', '13' e '14'.

Note que a Raiz da Árvore é representada na parte de cima do diagrama, e as Folhas da Árvore são representadas na parte de baixo. Ou seja, nessa representação, a Árvore está de cabeça para baixo! Note também que a Árvore é uma **estrutura hierárquica**: um Nós Pai é hierarquicamente “superior” aos seus Nós Filhos.

8.2 Árvores Binárias e Árvores Binárias de Busca

Uma propriedade importante de uma Árvore é o **número máximo de Filhos**, considerados todos os Nós. No exemplo da [Figura 8.1](#), na Árvore como um todo, o número máximo de Filhos é três. O Nós de valor '1', por exemplo, possui três Filhos. Nenhum nó da Árvore possui quatro ou mais filhos.

Nas **Árvores Binárias**, cada Nós da Árvore possui, no máximo, dois Filhos ou, ainda, duas Subárvore.

Definição: Árvore Binária

Nas Árvores Binárias, cada Nós da Árvore possui, no máximo, dois Filhos.

Figura 8.2 Definição de Árvore Binária.

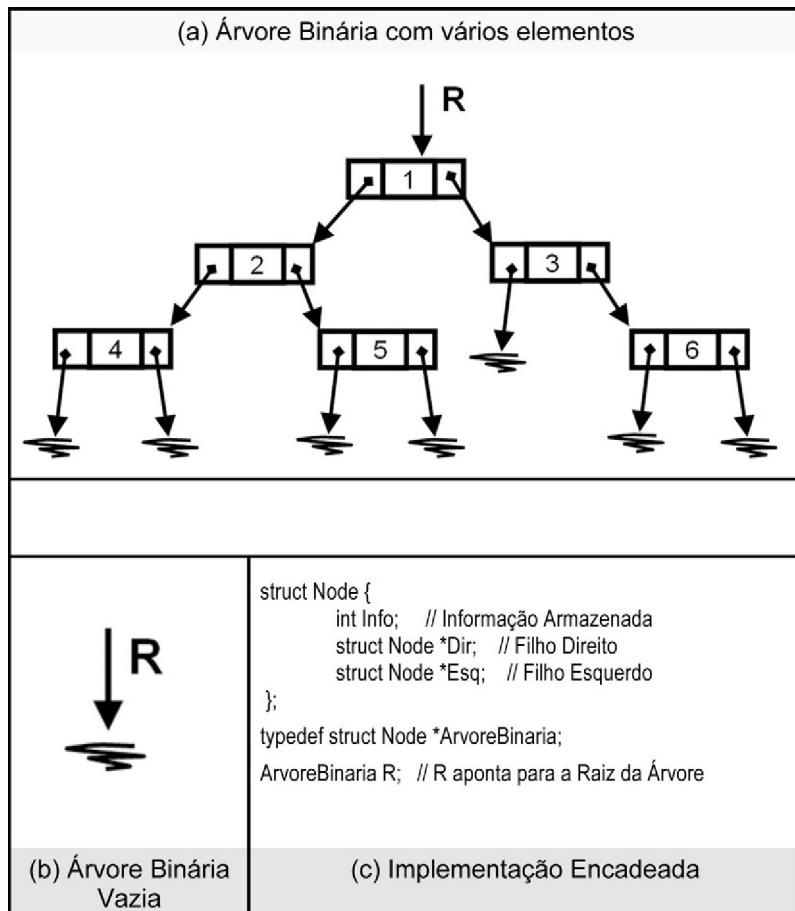


Figura 8.3 Implementando uma Árvore Binária com Alocação Encadeada e Dinâmica.

Nas Árvores Binárias, cada Nó da Árvore possui, no máximo, dois Filhos.

Podemos utilizar uma Árvore Binária para armazenamento temporário de conjuntos de elementos. A [Figura 8.3](#) mostra a representação de uma Árvore Binária, de Raiz R, implementada com Alocação Encadeada. Em cada Nó da Árvore Binária temos a **Informação** a ser armazenada (com valores '1' a '6'), um ponteiro para o **Filho Direito** e um ponteiro para o **Filho Esquerdo**. Nos Nós Terminais, os campos que indicam os Filhos Esquerdo e Direito estão apontando para Null.

Conforme a definição de [Drozdek \(2002; p. 198\)](#), em uma **Árvore Binária de Busca**, para cada Nó as informações armazenadas na Subárvore Esquerda são menores do que a informação armazenada no Nó Raiz, e as informações armazenadas na Subárvore Direita são maiores do que a informação armazenada no Nó Raiz. Para [Pereira \(1996, p. 176\)](#), uma Árvore Binária de Raiz R pode ser considerada uma Árvore Binária de Busca se atender a três critérios: (1) a Informação de cada Nó da Subárvore Esquerda do Nó apontado por R é menor do que a Informação armazenada no Nó apontado por R; (2) nenhum Nó da

Definição: Árvore Binária de Busca — ABB

Uma Árvore Binária com Raiz R pode ser considerada uma Árvore Binária de Busca se atender aos três seguintes critérios:

- (1) A Informação de cada Nó da Subárvore Esquerda de R é menor do que a Informação armazenada no Nó apontado por R.
- (2) A Informação de cada Nó da Subárvore Direita de R é maior do que a Informação armazenada no Nó apontado por R.
- (3) As Subárvores Esquerda e Direita do Nó apontado por R também são ABBs.

Figura 8.4 Definição de Árvore Binária de Busca – adaptada de Pereira (1996, p. 176), de modo a não permitir elementos repetidos.

Subárvore Direita do Nó apontado por R possui informação menor do que a Informação armazenada no Nó apontado por R; (3) as Subárvores Esquerda e Direita do Nó apontado por R também são ABBs. Note que, pelo critério 3, a definição de Pereira é recursiva.

Na definição de Pereira, uma ABB pode ter elementos repetidos (veja o critério 2). Isso não ocorre na definição de Drozdek. Adaptando a definição de Pereira de modo a não permitir repetição de informações em uma Árvore, chegamos à definição da [Figura 8.4](#).

Exercício 8.1 São ABBs?

Verifique se as Árvores R1 e R2, da [Figura 8.5](#), são Árvores Binárias de Busca segundo a definição da [Figura 8.4](#).

A Árvore R1 é uma Árvore Binária de Busca, pois atende aos três critérios da definição da [Figura 8.4](#). As Informações armazenadas em todos os Nós da Subárvore Esquerda de R1 (19, 41 e 47) são menores do que a Informação armazenada no Nó Raiz (50); (2) as Informações armazenadas em todos os Nós da Subárvore Direita de R1 (51, 63 e 89) são maiores do que a Informação armazenada no Nó Raiz (50); (3) se aplicarmos essa mesma análise, recursivamente, para as Subárvores Esquerda e Direita de R1, os critérios continuam sendo respeitados.

A Árvore R2 não é uma ABB. Você saberia dizer a razão? Preste atenção ao critério 3 da definição de Árvore Binária de Busca e compare as Árvores R1 e R2. Por que R1 é uma ABB e R2 não é?

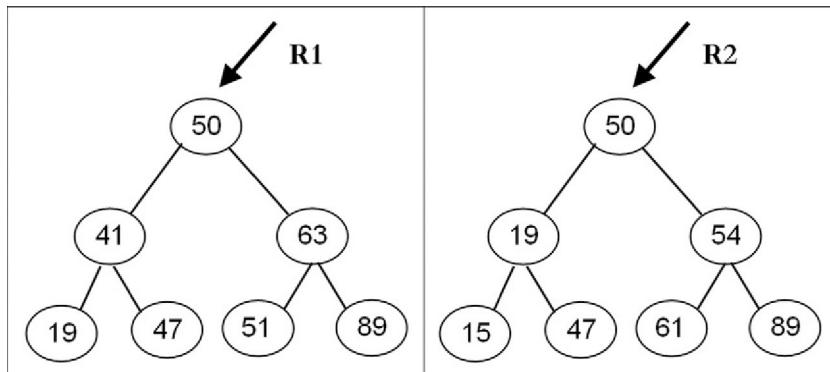


Figura 8.5 R1 e R2 são Árvores Binárias de Busca?

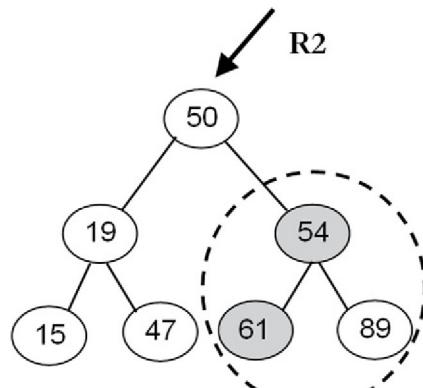


Figura 8.6 R2 não é uma Árvore Binária de Busca.



Vamos considerar isoladamente a Subárvore de R2 composta pelos Nós 54, 61 e 89 (destacada com o círculo pontilhado na [Figura 8.6](#)). Note que o Nô que armazena o valor 61 está na Subárvore Esquerda do Nô que armazena o valor 54. Como 61 é maior do que 54, essa situação quebra o critério 1 da definição de Árvores Binárias de Busca. Se a Subárvore composta pelos Nós 54, 61 e 89 não é uma Árvore Binária de Busca, pelo critério 3 a Árvore R2 como um todo também não será.

8.3 Algoritmos recursivos para Árvores Binárias de Busca

Devido à natureza hierárquica e recursiva das Árvores, os algoritmos para manipulação de Árvores podem ser implementados de modo mais fácil e natural se utilizarmos a recursividade.

Revisão sobre algoritmos recursivos

Um subprograma é recursivo quando faz chamadas a ele mesmo. Um exemplo clássico de algoritmos recursivos é o cálculo do fatorial. A [Figura 8.7](#) apresenta a definição de fatorial. Note, na cláusula ii, que se trata de uma definição recursiva.

Fatorial - Definição: $\left\{ \begin{array}{l} \text{(i) Fatorial de 0 é 1} \\ \text{(ii) Fatorial de } N \text{ é } N * \text{Fatorial}(N - 1) \end{array} \right.$	Dica: É possível resolver de imediato Deixamos para resolver depois <div style="text-align: right; margin-top: 20px;"> </div>
Fatorial – implementação recursiva: <pre style="font-family: monospace;">Inteiro factorial (parâmetro N do tipo Inteiro) { Se (N == 0) Então Retorne 1; Senão Retorne (N * Factorial(N-1)); } // fim Factorial</pre>	

Figura 8.7 Fatorial: recursividade na definição e na implementação.

Para resolver um problema recursivamente, precisamos identificar os casos em que conseguimos resolver o problema de imediato e os casos em que não conseguimos fazê-lo. Nos casos em que conseguimos resolver de imediato, simplesmente apresentamos a solução. Nos casos em que não conseguimos resolver de imediato, precisamos decompor o problema em problemas menores, nos aproximando da solução. No fundo, deixamos para resolver o problema em outro momento, fazendo uma chamada recursiva. Na [Figura 8.7](#), a linha tracejada delimita as situações em que conseguimos resolver de imediato e as situações em que precisamos deixar para resolver em outro momento, decompondo o problema em problemas menores através de uma ou mais chamadas recursivas.

A [Figura 8.7](#) apresenta também uma implementação recursiva para o cálculo do fatorial de N. A situação em que conseguimos resolver o problema de imediato ocorre quando N é igual a zero. Nesse caso, resolvemos o problema simplesmente retornando o valor 1. Nos demais casos, não conseguimos resolver de imediato, decompomos o problema em um problema menor e retornamos o resultado da expressão $N * \text{Fatorial}(N - 1)$. Ou seja, o resultado de fatorial de N será obtido pela multiplicação do valor N pelo resultado do fatorial de N – 1. Calculamos então o fatorial de N – 1 através de uma chamada recursiva, ou seja, uma chamada ao próprio procedimento fatorial.

Vamos exemplificar a execução do algoritmo recursivo da [Figura 8.7](#) calculando o fatorial de N quando N é igual a 3. Na primeira chamada ao procedimento fatorial, N é igual a 3 e o algoritmo retornará o resultado $3 * \text{Fatorial}(2)$. Para calcular o fatorial de 2, fazemos a segunda chamada ao procedimento Fatorial, agora com N igual a 2. Nessa segunda chamada, o resultado será $2 * \text{Fatorial}(1)$. Calculamos o fatorial de 1 na terceira chamada, e o resultado será $1 * \text{Fatorial}(0)$. Na quarta chamada, N é zero, e só então conseguimos resolver o problema em definitivo, sem necessidade de novas chamadas recursivas. Quando N é igual a zero, retornamos o valor 1 como resultado. Esse resultado do fatorial (0) é então repassado à terceira chamada, que o multiplica por 1 e resulta no valor 1. Esse resultado do fatorial (1) é então repassado à segunda chamada, que o multiplica por 2 e resulta no valor 2. Esse resultado do fatorial (2) é então repassado à primeira chamada, que o multiplica por 3 e resulta no valor 6, que é o resultado do fatorial de 3. A [Figura 8.8](#) ilustra a sequência de obtenção de resultados nas quatro chamadas do procedimento Fatorial, para o cálculo do fatorial de 3.

O valor X está na Árvore?

Queremos saber se um valor X está em uma Árvore Binária de Busca de Raiz R. Sendo uma Árvore Binária de Busca, os valores armazenados em cada Nó da Subárvore Esquerda de R precisam ser menores do que o valor armazenado no Nó apontado por R; e os valores armazenados em cada Nó da Subárvore Direita de R precisam ser maiores do que o valor armazenado no Nó apontado por R.

Considere, por exemplo, que o valor de X é 39. Podemos ter quatro situações para X e para o Nó apontado por R, conforme mostra a [Figura 8.9](#). No caso 1, estamos querendo saber se o valor X está em uma Árvore vazia. Essa é uma situação que conseguimos resolver de imediato: X não está na Árvore; apresentamos esse resultado e encerramos o algoritmo.

Chamada	N	Resultado
Primeira	3	$\text{Fatorial}(3) = 3 * \text{Fatorial}(2)$ 2 resultado do Fatorial(2) = 2
Segunda	2	$\text{Fatorial}(2) = 2 * \text{Fatorial}(1)$ 1 resultado do Fatorial(1) = 1
Terceira	1	$\text{Fatorial}(1) = 1 * \text{Fatorial}(0)$
Quarta	0	resultado do Fatorial(0) = 1 1 Fatorial(0) = 1

Figura 8.8 Chamadas recursivas para cálculo do fatorial de 3.

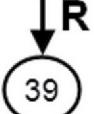
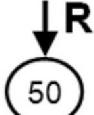
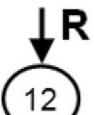
Caso	X	R	Conclusão
Caso 1: Árvore Vazia	39		X não está na árvore. Encerramos o algoritmo.
Caso 2: $R \rightarrow \text{Info} = X$	39		X está na árvore. Encerramos o algoritmo.
			É possível resolver de imediato e encerrar o algoritmo
			Deixamos para resolver depois, com recursividade
Caso 3: $X < R \rightarrow \text{Info}$	39		Se X estiver na Árvore, estará na Subárvore Esquerda de R. O algoritmo não acaba ainda. Fazemos uma chamada recursiva.
Caso 4: $X > R \rightarrow \text{Info}$	39		Se X estiver na Árvore, estará na Subárvore Direita de R. O algoritmo não acaba ainda. Fazemos uma chamada recursiva.

Figura 8.9 Casos do algoritmo que verifica se um valor X está em uma Árvore Binária de Busca R.

No caso 2, estamos procurando X na Árvore R e sabemos que $R \rightarrow \text{Info} = X$. Ou seja, no caso 2, o valor X está armazenado no Nó apontado por R. Também conseguimos resolver essa situação de imediato: temos certeza de que X está na Árvore; apresentamos esse resultado e encerramos o algoritmo.

No caso 3, o Nó apontado por R armazena uma Informação que é maior do que X. O que fazemos nesse caso? Nesse momento da execução ainda não temos como saber com certeza se X está na Árvore. Mas uma coisa nós sabemos: como se trata de uma Árvore Binária de Busca, e como X é menor do que a Informação armazenada no Nó apontado por R, se X estiver na Árvore estará na Subárvore Esquerda de R. Então deixamos para dar uma resposta definitiva posteriormente e fazemos uma chamada recursiva. Analogamente, no caso 4, se X estiver na Árvore estará na Subárvore Direita de R, e também resolvemos com uma chamada recursiva.

Exercício 8.2 Algoritmo recursivo: X está na Árvore?

Desenvolva um algoritmo recursivo para verificar se um valor X faz parte de uma Árvore R. X e R são passados como parâmetros.



Boolean `EstáNaÁrvore` (parâmetro por referência R do tipo ABB, parâmetro X do tipo Inteiro);

/* Verifica se o valor X está na Árvore de Raiz R, retornando Verdadeiro para o caso de X estar na Árvore e Falso para o caso de não estar */

Tratando separadamente cada um dos quatro casos especificados na [Figura 8.9](#), chegamos ao algoritmo da [Figura 8.10](#). Nos casos 1 e 2, conseguimos dar uma resposta definitiva, e encerramos o algoritmo retornando os valores Falso e Verdadeiro, respectivamente.

No caso 3, sabemos que $R \rightarrow \text{Info} > X$. Ou seja, o valor armazenado no Nó apontado por R é maior do que o valor que procuramos. Se X estiver na Árvore, estará na Subárvore Esquerda de R. Assim, chamamos recursivamente o procedimento `EstáNaÁrvore` e passamos como primeiro parâmetro $R \rightarrow \text{Esq}$. Ou seja, verificamos se o valor X está

```
Boolean EstáNaÁrvore (parâmetro por referência R do tipo ABB, parâmetro X do tipo
Inteiro) {

    /* Verifica se o valor X está ou não está na Árvore de Raiz R, retornando Verdadeiro para
    o caso de X estar na Árvore e Falso para o caso de não estar */

    Se (R == Null)
        Então Retorne Falso; // Caso 1: Árvore vazia; X não está na Árvore;
    Senão Se (X == R → Info)
        Então Retorne Verdadeiro; // Caso 2: X está na árvore; acabou o algoritmo
    Senão Se (R → Info > X)
        Então Retorne (Está_Na_Arvore (R → Esq, X));
        // Caso 3: se estiver na Árvore, estará na Subárvore Esquerda

        Senão Retorne (Está_Na_Arvore (R → Dir, X));
        // Caso 4: se estiver na Árvore, estará na Subárvore Direita
    } // fim EstáNaÁrvore
```

Figura 8.10 Algoritmo conceitual — `EstáNaÁrvore`.

na Árvore apontada por $R \rightarrow \text{Esq}$. Analogamente, no caso 4, verificamos se o valor X está na Árvore apontada por $R \rightarrow \text{Dir}$.

Execução do algoritmo EstáNaÁrvore para $X = 39$

Para exemplificar a execução do algoritmo recursivo da [Figura 8.10](#), considere que estamos procurando X com valor igual a 39 na Árvore R esquematizada na [Figura 8.11](#). A execução implicará três chamadas ao procedimento `EstáNaÁrvore`. Nas três chamadas, o valor de X será o mesmo, mas o valor de R será alterado. Os valores de R para a primeira, segunda e terceira chamadas estão identificados na [Figura 8.11](#) por R1, R2 e R3, respectivamente.

Na primeira chamada ao procedimento `EstáNaÁrvore`, a Árvore R (identificada por R1, na primeira chamada) não é vazia, e o valor armazenado no Nó apontado por R1 é 50, maior que 39, que é o valor de X. Identificamos, nessa primeira chamada, o caso 3: situação em que $R \rightarrow \text{Info} > X$. De acordo com o algoritmo, fazemos então uma chamada recursiva através do comando `Retorne(EstáNaÁrvore($R \rightarrow \text{Esq}$, X))`. Com esse comando, o resultado da primeira chamada ao procedimento `EstáNaÁrvore` retornará exatamente o resultado da segunda chamada, que buscará o valor de X na Árvore apontada por $R1 \rightarrow \text{Esq}$.

Entramos, então, na segunda chamada. R, na segunda chamada identificado por R2, agora aponta para o Nó que armazena o valor 28. Ou seja, nessa segunda chamada, identificamos o caso 4, pois $R2 \rightarrow \text{Info} < X$. De acordo com o algoritmo, fazemos uma chamada recursiva através do comando `Retorne(EstáNaÁrvore ($R \rightarrow \text{Dir}$, X))`. Com esse comando, o resultado da segunda chamada ao procedimento `EstáNaÁrvore` retornará precisamente o resultado da terceira chamada, que buscará o valor de X na Árvore apontada por $R2 \rightarrow \text{Dir}$.

Entramos então na terceira chamada. R (identificado por R3) agora aponta para o Nó que armazena o valor 39. Identificamos o caso 2 ($X = R \rightarrow \text{Info}$). De acordo com o algoritmo, retornamos o resultado Verdadeiro nessa terceira chamada.

Ao encerrar a terceira chamada, voltamos para a segunda chamada, no ponto onde foi feita a chamada recursiva. O resultado do comando `Retorne (EstáNaÁrvore($R2 \rightarrow \text{Dir}$, X))` será `Retorne(Verdadeiro)`, haja visto que o resultado da terceira chamada foi Verdadeiro.

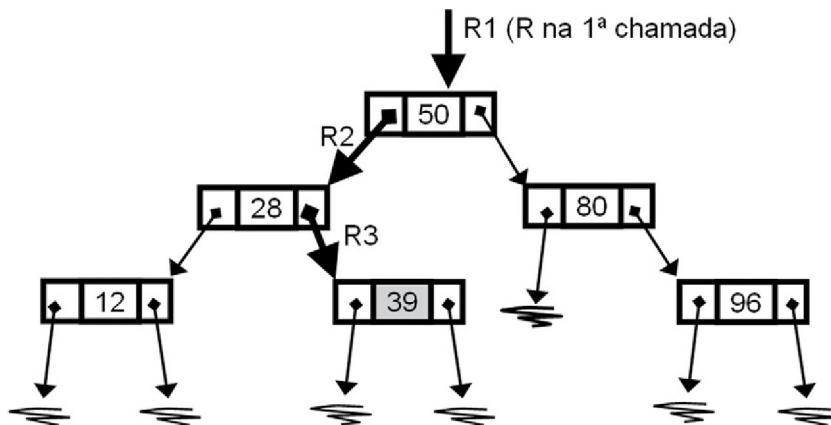


Figura 8.11 Execução de EstáNaÁrvore para $X = 39$.

Chamada	Caso	Resultado
Primeira	3	Retorne (EstáNaÁrvore(R1→Esq, X))
Segunda	4	Verdadeiro → Retorne (EstáNaÁrvore(R2→Dir, X))
Terceira	2	Verdadeiro → Retorne Verdadeiro

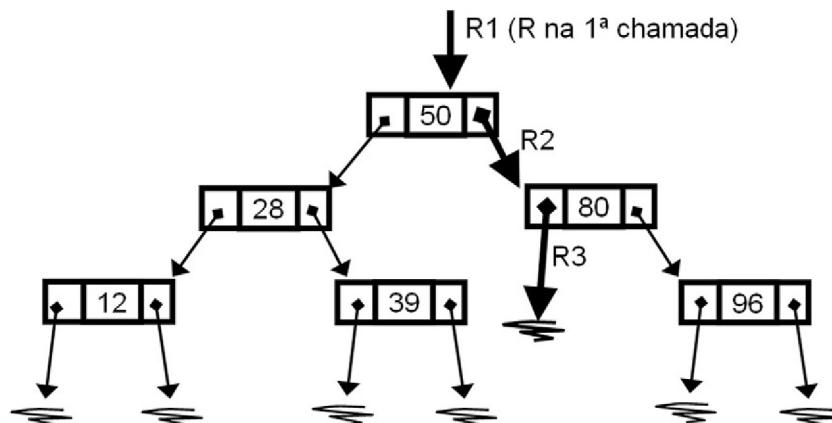
Figura 8.12 Resultados de EstáNaÁrvore para X = 39.

Analogamente, voltamos para a primeira chamada, no ponto onde foi feita a chamada recursiva. O resultado do comando Retorne (EstáNaÁrvore(R1→Esq, X)) será Retorne(Verdadeiro), haja visto que o resultado da segunda chamada foi Verdadeiro. O resultado Verdadeiro na primeira chamada encerra a execução. A [Figura 8.12](#) ilustra a sequência em que os resultados são obtidos e transferidos entre as três chamadas do procedimento EstáNaÁrvore, para a Árvore da [Figura 8.11](#) e X com valor 39.

Execução do algoritmo EstáNaÁrvore para X=70

Vamos fazer uma segunda execução do algoritmo recursivo da [Figura 8.10](#), agora para verificar se X com valor 70 está na Árvore da [Figura 8.13](#). Essa execução implicará três chamadas ao procedimento EstáNaÁrvore. Nas três chamadas, o valor de X será o mesmo; o valor de R será alterado e identificado na [Figura 8.13](#) por R1, R2 e R3, respectivamente.

Na primeira chamada ao procedimento EstáNaÁrvore, a Árvore R (R1) não é vazia, e o valor armazenado no Nό apontado por R1 é 50, menor que 70, que é o valor de X. Identificamos, nessa primeira chamada, o caso 4: situação em que $R \rightarrow \text{Info} < X$. De acordo com o algoritmo, fazemos uma chamada recursiva através do comando

**Figura 8.13** Execução de EstáNaÁrvore para X = 70.

Chamada	Caso	Resultado
Primeira	3	Retorne (EstáNaÁrvore(R1→Dir, X))
Segunda	4	Falso Retorne (EstáNaÁrvore(R2→Esq, X))
Terceira	2	Falso Retorne Falso

Figura 8.14 Resultado de EstáNaÁrvore para X = 70.

Retorne(EstáNaÁrvore($R \rightarrow$ Dir, X)). Com esse comando, o resultado da primeira chamada ao procedimento EstáNaÁrvore retornará exatamente o resultado da segunda chamada, que buscará o valor de X na Árvore apontada por R1→Dir.

Entramos, então, na segunda chamada. R (R2) agora aponta para o Nó que armazena o valor 80. Ou seja, nessa segunda chamada, identificamos o caso 3, pois $R2 \rightarrow$ Info > X. De acordo com o algoritmo, fazemos uma chamada recursiva através do comando Retorne(EstáNaÁrvore ($R \rightarrow$ Esq, X)). Com esse comando, o resultado da segunda chamada ao procedimento EstáNaÁrvore retornará precisamente o resultado da terceira chamada, que buscará o valor de X na Árvore apontada por R2→Esq.

Entramos então na terceira chamada. R (R3) agora aponta para Null. Identificamos o caso 1 (Árvore vazia). De acordo com o algoritmo, retornamos o resultado Falso nessa terceira chamada para indicar que X não está na Árvore.

Ao encerrar a terceira chamada, voltamos para a segunda chamada, no ponto onde foi feita a chamada recursiva. O resultado do comando Retorne (EstáNaÁrvore($R2 \rightarrow$ Esq, X)) será Retorne(Falso), haja visto que o resultado da terceira chamada foi Falso. Analogamente, voltamos para a primeira chamada, no ponto onde foi feita a chamada recursiva. O resultado do comando Retorne (EstáNaÁrvore($R1 \rightarrow$ Dir, X)) será Retorne(Falso), haja visto que o resultado da segunda chamada foi Falso. O resultado Falso na primeira chamada encerra a execução. A [Figura 8.14](#) ilustra a sequência em que os resultados são obtidos e transferidos entre as três chamadas do procedimento EstáNaÁrvore, para a Árvore da [Figura 8.13](#) e X com valor 70. Ao elaborar algoritmos recursivos, procure seguir as sugestões da [Figura 8.15](#).

Ao elaborar algoritmos recursivos:

- Liste todos os casos, identificando-os como caso 1, caso 2, e assim por diante.
- Identifique os casos em que é possível dar uma resposta de imediato, e proponha a resposta.
- Identifique os casos em que não é possível resolver de imediato, e procure resolver com uma ou mais chamadas recursivas.

Figura 8.15 Sugestões para elaborar algoritmos recursivos.

Exercício 8.3 Algoritmo recursivo: imprimir uma Árvore

Desenvolva um algoritmo recursivo para imprimir uma Árvore Binária de Busca de Raiz R, com elementos do tipo Inteiro. Faça uma versão do algoritmo que imprima as informações em ordem crescente e outra versão que imprima em ordem decrescente.

ImprimeTodos (parâmetro por referência R do tipo ABB);

/* Imprime todos os elementos da Árvore de Raiz R */

A [Figura 8.16](#) apresenta duas versões do algoritmo que imprime todos os elementos de uma Árvore Binária de Busca. Para imprimir recursivamente todos os elementos de uma Árvore Binária de Busca, temos dois casos a tratar: caso 1 — a Árvore é vazia; caso 2 — a Árvore não é vazia. No caso 1, não é preciso fazer nada, pois não existem elementos a imprimir. No caso 2, imprimimos o elemento que está na raiz e então, recursivamente, imprimimos todos os elementos das Subárvores Esquerda e Direita ([Figura 8.16a](#)). Quando utilizamos essa sequência — primeiro a Raiz, depois todos da Subárvore Esquerda, depois todos da Subárvore Direita, estamos percorrendo a Árvore em “pré-ordem”. O prefixo “pré” indica que a Raiz está sendo tratada previamente. Se executarmos essa versão do algoritmo com a Árvore R da [Figura 8.13](#), os elementos serão impressos na seguinte ordem: 50, 28, 12, 39, 80, 96.

Na versão da [Figura 8.16b](#), após verificar que estamos tratando o caso 2 (Árvore não vazia), imprimimos recursivamente todos os elementos da Subárvore Esquerda,

(a) Percurso em "pré-ordem": Raiz, Esquerda, Direita

```
ImprimeTodos (parâmetro por referência R do tipo ABB) {
{
/* Imprime todos os elementos da Árvore R na seguinte sequência: primeiro imprime a
Raiz, depois as Subárvores Esquerda e Direita. Percurso em "pré-Ordem": Raiz,
Esquerda, Direita */

Se (R != Null)
Então {   Escreva(R→Info);    // imprime a informação da raiz
          ImprimeTodos(R→Esq);    // imprime todos da Subárvore Esquerda
          ImprimeTodos(R→Dir); }    // imprime todos da Subárvore Direita
} // fim ImprimeTodos – Pré-Ordem
```

```
ImprimeTodos (parâmetro por referência R do tipo ABB)
/* Imprime todos os elementos da Árvore R na seguinte sequência: primeiro imprime
todos da Subárvore Esquerda, depois imprime a Raiz, depois imprime todos da
Subárvore Direita. Os elementos serão impressos em ordem crescente. Percurso em "In
Ordem": Raiz impressa entre os elementos das Subárvores Esquerda e Direita */

Se R != Null
Então {   ImprimeTodos(R→Esq);    // imprime todos da Subárvore Esquerda
          Escreva(R→Info);    // imprime a informação da raiz
          ImprimeTodos(R→Dir); }    // imprime todos da Subárvore Direita
} // fim ImprimeTodos - In Ordem
```

(b) Percurso em "In Ordem": Esquerda, Raiz, Direita

Figura 8.16 Algoritmo para imprimir os elementos da Árvore.



ELSEVIER

depois imprimimos o elemento da Raiz, e só então imprimimos todos os elementos da Subárvore Direita. Quando utilizamos essa sequência — primeiro todos da Subárvore Esquerda, depois o elemento da Raiz, depois todos os elementos da Subárvore Direita — estamos percorrendo a Árvore em “in ordem”. O prefixo “in” indica que a Raiz está sendo tratada entre as Subárvores. Se executarmos essa versão do algoritmo com a Árvore R da [Figura 8.13](#), os elementos serão impressos em ordem crescente: 12, 28, 39, 50, 80, 96.

Se desejarmos imprimir os elementos em ordem decrescente, basta imprimir primeiramente os elementos da Subárvore Direita, depois o elemento da Raiz e então os elementos da Subárvore Esquerda. Também é possível percorrer uma Árvore em “pós-ordem”: elementos da Subárvore Esquerda, depois os elementos da Subárvore Direita e então o elemento da Raiz.

Exercício 8.4 Soma dos elementos de uma Árvore

Implemente uma função recursiva que retorne a soma do valor de cada um dos Nós de uma Árvore Binária R. Considere elementos do tipo Inteiro.

Inteiro Soma (parâmetro por referência R do tipo ÁrvoreBinária);

/* Retorna a soma do valor dos elementos de R. Elementos do tipo Inteiro */

Exercício 8.5 Número de Nós com um único Filho

Implemente uma função recursiva que retorne o número de Nós de uma Árvore R que contém um único Filho. Em um Nó com um único Filho, um dos filhos deve ser nulo e o outro filho deve ser não nulo.

Inteiro NósComUmÚnicoFilho (parâmetro por referência R do tipo ÁrvoreBinária);

/* Retorna o número de Nós de R com um único Filho */

Exercício 8.6 Árvores são iguais?

Desenvolva um algoritmo recursivo para verificar se duas Árvores Binárias são iguais. Considere que duas Árvores vazias são iguais. Duas Árvores não vazias são iguais se apresentarem exatamente a mesma disposição dos elementos.

Boolean Iguais (parâmetros por referência R1, R2 do tipo ÁrvoreBinária);

/* Verifica se R1 e R2 são iguais */

Exercício 8.7 É Árvore Binária de Busca?

Implemente uma função recursiva que verifica se uma Árvore Binária R, passada como parâmetro, é uma Árvore Binária de Busca, conforme a definição da [Figura 8.4](#). Sugestões: leia cuidadosamente a definição antes de iniciar o desenvolvimento. Use subprogramas auxiliares, se necessário.

Boolean É_ABB (parâmetro por referência R do tipo ÁrvoreBinária);

/* verifica se a Árvore Binária R é uma Árvore Binária de Busca, segundo a definição da Figura 8.4. Elementos do tipo Inteiro */

8.4 Árvores Binárias de Busca: inserir e eliminar elementos

Considere a Árvore Binária de Busca da [Figura 8.17](#). Se quisermos inserir o valor 37, em que lugar da Árvore esse novo elemento poderá ser colocado? Lembre-se de que temos um critério a respeitar: valores menores que a Informação armazenada na Raiz devem ficar na Subárvore Esquerda; valores maiores que a Informação armazenada na Raiz devem ficar na Subárvore Direita; esse critério deve ser respeitado para cada Nô da Árvore.

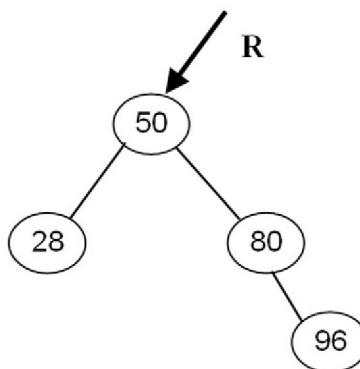


Figura 8.17 ABB: onde inserir o valor 37?

Uma possível estratégia é inserir todos os novos valores como Nós Terminais (sem Filhos). Inserindo novos valores em Nós Terminais, evitamos alterações na estrutura da Árvore, pois não precisaremos mudar de lugar nenhum dos valores já existentes.

Para não quebrar o critério de ordenação que define uma Árvore Binária de busca, existe um único lugar na Árvore onde podemos inserir um valor como Nô Terminal. Na ABB da [Figura 8.17](#), podemos inserir o valor 37 como um Nô Terminal à direita do Nô que armazena o valor 28. Se colocado em um Nô Terminal em qualquer outro local, o valor 37 quebraria o critério que define uma ABB. Analogamente, se quisermos inserir o valor 55 como um Nô Terminal, ele teria que ser posicionado à esquerda do Nô que armazena o valor 80; se quisermos inserir o valor 12, ele teria que ser colocado à esquerda de 28, e assim por diante.

Inserindo novos valores em uma ABB

- Inserir novos elementos como Nós Terminais (sem Filhos).
- Procurar o lugar certo, considerando o critério que define uma ABB e então inserir.

Figura 8.18 ABB: estratégia geral para inserir novos valores.

Algoritmo Insere

Se quisermos inserir um novo valor X em uma ABB de Raiz R, quatro possíveis situações podem acontecer — [Figura 8.19](#). No caso 1, estamos querendo inserir X em uma Árvore vazia. Se a Árvore estiver vazia, podemos inserir X na própria Raiz da Árvore.

Caso	X	R	Conclusão
Caso 1: Árvore Vazia	37		Encontramos o local onde X deve ser inserido. Inserimos e encerramos o algoritmo.
Caso 2: $R \rightarrow \text{Info} = X$	37		X já está na árvore. Não inserimos (para não permitir elementos repetidos) e encerramos o algoritmo.
Caso 3: $X < R \rightarrow \text{Info}$	37		X deve ser inserido na Subárvore Esquerda de R. O algoritmo não acaba ainda. Fazemos uma chamada recursiva.
Caso 4: $X > R \rightarrow \text{Info}$	37		X deve ser inserido na Subárvore Direita de R. O algoritmo não acaba ainda. Fazemos uma chamada recursiva.

Figura 8.19 ABB: casos do algoritmo Insere.

A ABB passará a ter um único Nó, que armazenará o valor X. Esse novo Nó passará a ser a Raiz da Árvore.

No caso 2, queremos inserir X na ABB R e logo na Raiz achamos um valor igual a X. Como não podemos ter dois valores iguais na ABB, simplesmente não inserimos X e encerramos o algoritmo.

No caso 3, sabemos que $R \rightarrow \text{Info} > X$, ou seja, X é menor que a Informação que está armazenada na Raiz. Não temos certeza quanto ao local onde poderemos inserir X. Aliás, nem temos certeza ainda se poderemos inserir X. Mas sabemos que, se X não estiver na Árvore, ele deverá ser inserido na Subárvore Esquerda de R. Analogamente, no caso 4, em que $R \rightarrow \text{Info} < X$, se X não estiver na Árvore, deverá ser inserido na Subárvore Direita de R.

Exercício 8.8 Algoritmo recursivo: Insere em uma ABB

Desenvolva um algoritmo recursivo para inserir um novo valor X em uma Árvore Binária de Busca de Raiz R. X e R são passados como parâmetros. X deve ser inserido como um Nó Terminal.



Insere (parâmetro por referência R do tipo ABB, parâmetro X do tipo Inteiro, parâmetro por referência Ok do tipo Boolean);
 /* Insere o valor X na ABB de Raiz R, como um Nó terminal, sem Filhos. Ok retorna Verdadeiro para o caso de X ter sido inserido e Falso caso contrário. */

Tratando separadamente cada um dos quatro casos especificados na [Figura 8.19](#), chegamos ao algoritmo da [Figura 8.20](#). Nos casos 1 e 2, conseguimos dar uma resposta definitiva, e encerramos o algoritmo retornando os valores Falso e Verdadeiro, respectivamente. Nos casos 3 e 4, não é possível encerrar o algoritmo, e fazemos uma chamada recursiva.

```

Insere (parâmetro por referência R do tipo ABB, parâmetro X do tipo Inteiro, parâmetro por referência Ok do tipo Boolean) {
  /* Insere o valor X na ABB de Raiz R, como um Nó terminal, sem Filhos. Ok retorna Verdadeiro para o caso de X ter sido inserido e Falso caso contrário. */

  Variável P do tipo NodePtr;
  Se (R == Null)
    Então {   P = NewNode;    // Caso 1: Achou o lugar; insere e encerra o algoritmo
              P→Info = X;
              P→Dir = Null;
              P→Esq = Null;
              R = P;
              P = Null;
              Ok = Verdadeiro; }
  Senão {   Se (X == R→Info)
              Então Ok = Falso; // Caso 2: X já está na árvore; não insere; acaba o algoritmo
              Senão {   Se (R→Info > X)
                          Então    /* Caso 3: tenta inserir X na Subárvore Esquerda de R */
                          Insere (R→Esq, X, Ok)
                          Senão    /* Caso 4: tenta inserir X na Subárvore Direita de R */
                          Insere(R→Dir, X, Ok);
                      } // fim senão
                  } // fim senão
  } // fim Insere ABB
}

```

Figura 8.20 Algoritmo conceitual — Insere em ABB.

Execução do algoritmo Insere

Considere a inserção de $X = 37$ na Árvore da [Figura 8.21a](#). R_1 , R_2 e R_3 fazem referência ao parâmetro R na primeira, segunda e terceira chamadas ao procedimento `Insere`, respectivamente. Na primeira chamada identificamos o caso 3 ([Figura 8.19](#)), pois $R_1\rightarrow\text{Info}$ é 50, maior que X , que tem valor 37. De acordo com o algoritmo, fazemos uma chamada recursiva para inserir X na Subárvore Esquerda de R_1 .

Na segunda chamada, R_2 aponta para o Nó cujo valor é 28. Nessa segunda chamada, identificamos o caso 4, pois $R_2\rightarrow\text{Info} < X$. De acordo com o algoritmo, fazemos uma chamada recursiva para inserir X na Subárvore Direita de R_2 .

Na terceira chamada, R_3 está apontando para Null e, assim, identificamos o caso 1. Nesse momento inserimos X . De acordo com o algoritmo para tratar o caso 1, aplicamos os comandos $P = \text{NewNode}$; $P\rightarrow\text{Info} = X$; $P\rightarrow\text{Dir} = \text{Null}$; $P\rightarrow\text{Esq} = \text{Null}$ e chegamos à situação da [Figura 8.21b](#).

A partir da situação da [Figura 8.21b](#), aplicamos o comando $R = P$. Como estamos na terceira chamada, R aqui faz referência a R_3 . Logo, R_3 passa a apontar para onde aponta P . Note, no algoritmo da [Figura 8.20](#), que o parâmetro R é passado por referência. Logo, se atualizamos R_3 , será atualizado também o campo $R_2\rightarrow\text{Dir}$, pois desencadeamos a terceira chamada com o comando `Insere(R2→Dir, X, Ok)`. No fundo, o nosso R_3 é o $R_2\rightarrow\text{Dir}$.

O parâmetro Ok receberá o valor Verdadeiro na terceira chamada. Como também é um parâmetro passado por referência, Ok ficará verdadeiro também na segunda e na

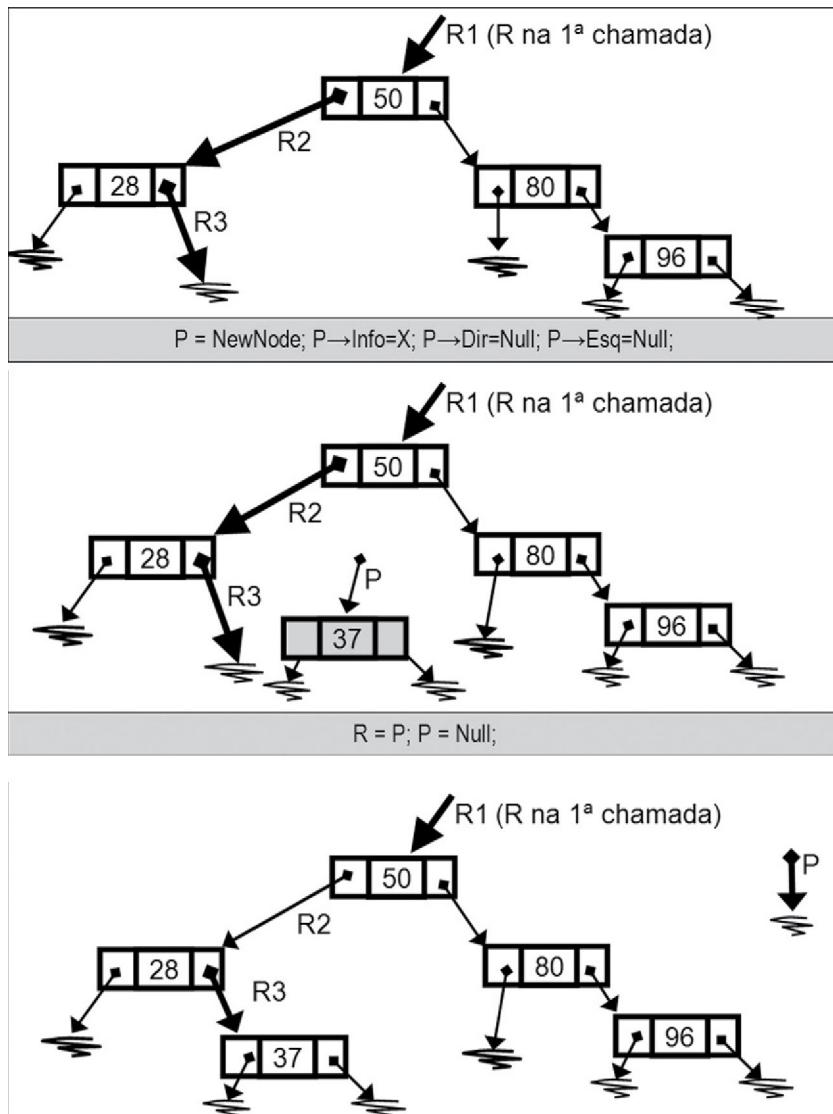


Figura 8.21 Execução de Insere para $X = 37$.

primeira chamadas. P é uma variável temporária, que simplesmente apontamos para Null, e encerramos o algoritmo.

Algoritmo Remove

Na tentativa de remover um valor X de uma ABB de Raiz R, podemos nos deparar com quatro casos, detalhados na [Figura 8.22](#). Note que são as mesmas situações identificadas para o algoritmo Insere ([Figura 8.19](#)) e para o algoritmo que verifica se um valor X está na Árvore ([Figura 8.9](#)). As situações são as mesmas; as ações que devem ser desencadeadas em cada situação é que mudam.

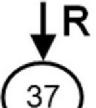
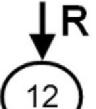
Caso	X	R	Conclusão
Caso 1: Árvore Vazia	37		Se a Árvore está vazia, X não está na Árvore. Não removemos nenhum Nô e encerramos o algoritmo.
Caso 2: $R \rightarrow \text{Info} = X$	37		Encontramos X. Removemos X e fazemos os ajustes necessários na Árvore.
Caso 3: $X < R \rightarrow \text{Info}$	37		A procura por X deve continuar na Subárvore Esquerda de R. O algoritmo não acaba e fazemos uma chamada recursiva.
Caso 4: $X > R \rightarrow \text{Info}$	37		A procura por X deve continuar na Subárvore Direita de R. O algoritmo não acaba e fazemos uma chamada recursiva.

Figura 8.22 ABB: casos do algoritmo Remove.

No caso 1, estamos querendo remover X de uma Árvore vazia. Se a árvore está vazia, X não está na Árvore, não removemos nenhum Nô e encerramos o algoritmo.

No caso 3 encontramos em $R \rightarrow \text{Info}$ um valor maior que X, e, no caso 4, um valor menor que X. Nesses dois casos, ainda não temos certeza se encontraremos X na Árvore. Mas, pelos critérios que definem uma ABB, sabemos que no caso 3 a busca por X deve continuar na Subárvore Esquerda, e, no caso 4, na Subárvore Direita. Portanto, nos casos 3 e 4, continuamos o algoritmo fazendo uma chamada recursiva.

No caso 2, encontramos o valor que queremos remover. Precisamos agora remover efetivamente o Nô que contém o valor X e fazer os ajustes necessários na Árvore. Para exemplificar as possíveis situações de remoção e ajustes na Árvore, considere as três situações da [Figura 8.23](#). Na [Figura 8.23a](#), o Nô que contém o valor a ser removido, 37, não possui Filhos. Na [Figura 8.23b](#), o Nô que contém o valor a ser removido possui um único Filho. E, na [Figura 8.23c](#), o Nô que contém o valor a ser removido possui dois Filhos. Os ajustes na Árvore serão diferentes nesses três casos.

Quando o Nô a ser removido não possui Filhos, eliminamos o Nô, e o ponteiro R passa a apontar para Null, como ilustrado na [Figura 8.23a](#). Quando o Nô a ser removido possui um único Filho, eliminamos o Nô, e o ponteiro R passa a apontar para o Filho não nulo. Veja, na [Figura 8.23b](#), que R passa a apontar para o Nô que contém o elemento de valor 80.

E se o Nô a ser removido da ABB tivesse dois Filhos, como na [Figura 8.23c](#)? Como podemos ajustar a Árvore nesse caso? Lembre-se de que é preciso respeitar o critério de ordenação que define uma Árvore Binária de Busca.

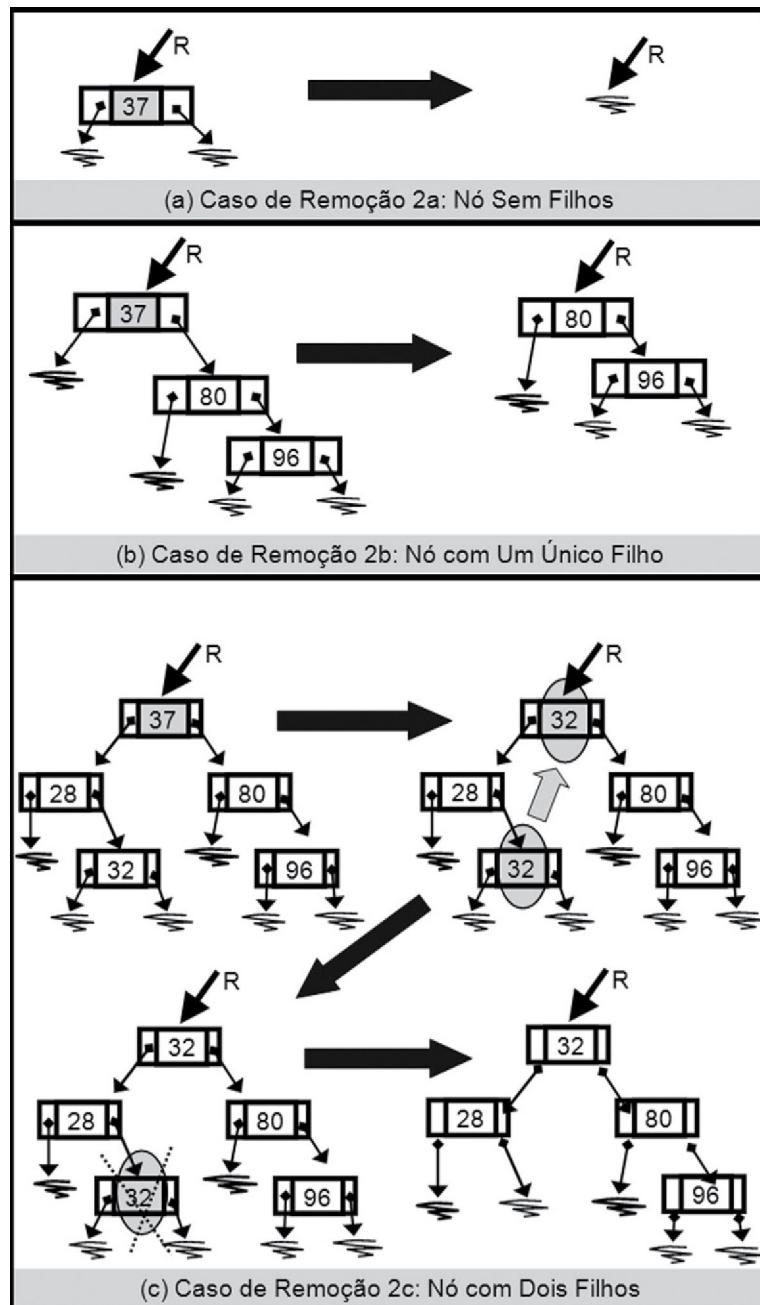


Figura 8.23 Removendo Nós com zero, um ou dois Filhos.

Com o objetivo de minimizar a movimentação dos demais Nós da Árvore, uma possível estratégia para ajustar a Árvore seria substituir o valor que queremos remover (37) por um outro valor da Árvore. Na Árvore inicial da [Figura 8.23c](#), temos dois valores que

(a) Caso de remoção 2a: Nós sem Filhos	Remover o Nós; apontar R para Null.
(b) Caso de remoção 2b: Nós com um único Filho	Remover o Nós e apontar R para o Filho não nulo.
(c) Caso de remoção 2c: Nós com dois Filhos	Encontrar o maior valor da Subárvore Esquerda de R; substituir R→Info por esse maior valor; remover o maior valor da Subárvore Esquerda de R.

Figura 8.24 Ajuste da Árvore para remoção de Nós com zero, um ou dois Filhos.

podemos colocar no lugar do valor 37 sem que o critério de ordenação de uma ABB seja quebrado. Você saberia dizer quais são esses dois valores?

Podemos substituir o valor 37 pelo valor 32, que é o maior valor da Subárvore Esquerda do Nós que armazena o valor 37. Sendo o maior valor da Subárvore Esquerda, se colocarmos 32 no lugar de 37 não haverá nenhum valor na Subárvore Esquerda maior que 32. E também não haverá nenhum valor na Subárvore Direita que seja menor que 32. Ou seja, o critério que define uma ABB não será quebrado. Também poderíamos substituir 37 pelo valor 80, que é o menor valor da Subárvore Direita do Nós que armazena o valor 37. Nesse caso, o critério de ordenação de uma ABB também não seria quebrado.

A [Figura 8.23c](#) mostra uma sequência de operações para ajuste da Árvore na remoção de um Nós com dois Filhos. Primeiramente substituímos o valor 37 pelo valor 32, que é o maior valor da Subárvore Esquerda de R. Em seguida, para a Árvore não ficar com dois valores 32, removemos o valor 32 que está na Subárvore Esquerda de R.

Note que o maior valor da Subárvore Esquerda de R nunca terá dois Filhos. Na verdade, o maior valor nunca terá o Filho Direito porque, se tivesse um Filho Direito, esse Filho Direito seria maior que ele.

A [Figura 8.24](#) resume as ações para ajuste de uma ABB para remoção de um Nós com zero, um ou dois Filhos.

Exercício 8.9 Algoritmo Remove em uma ABB

Desenvolva um algoritmo recursivo para remover um valor X de uma Árvore Binária de Busca de Raiz R. X e R são passados como parâmetros. Como sugestão, trate os casos previstos nas [Figuras 8.22 a 8.24](#).

Remove (parâmetro por referência R do tipo ABB, parâmetro X do tipo Inteiro, parâmetro por referência Ok do tipo Boolean);

```
/* Remove o valor X da ABB de Raiz R. Ok retorna Verdadeiro para o caso de X ter sido encontrado e removido, e Falso caso contrário. */
```

Exercício 8.10 ABB: Cria, Vazia, Destroi

Para criar o Tipo Abstrato de Dados ABB, reúna as operações EstáNaÁrvore, Insere e Remove, implementadas nos Exercícios 8.2, 8.8 e 8.9, e implemente as operações Cria, Vazia e Destroi. A operação Cria inicializa a Árvore como vazia; a operação Vazia verifica se uma ABB está ou não vazia; a operação Destroi remove todos os Nós da Árvore e a deixa vazia.

8.5 Por que uma Árvore Binária de Busca é boa?

Considere, por exemplo, que queremos desenvolver um sistema de votação por telefone, como em programas de televisão em que o telespectador vota e decide quem ganha o prêmio. Queremos que o sistema de votação tenha as seguintes características:

- Cada número de telefone pode votar uma única vez.
- Um sistema de informação deve armazenar todos os números que já ligaram.
- A cada nova ligação recebida, o sistema verifica se aquele número já votou; o voto é computado apenas se o número ainda não votou.
- O resultado parcial da votação deve estar sempre disponível, sendo atualizado automaticamente a cada voto.

Imagine que estejamos utilizando uma Árvore Binária de Busca para armazenar os números de telefone que já votaram. Cada número de telefone que já votou é armazenado em um Nô de ABB. Suponha que em determinado momento a ABB tenha um milhão de Nôs uniformemente distribuídos. Ou seja, um milhão de números de telefone armazenados. Surge uma nova ligação e é preciso saber se aquele número já votou, ou seja, se aquele número está na Árvore. Quantos nôs da ABB teriam que ser visitados, no máximo, para saber se o telefone que está ligando no momento está na Árvore?

Exercício 8.11 Quantos Nôs é preciso visitar?

Analice o algoritmo do Exercício 8.2 e calcule quantos Nôs é preciso visitar, no máximo, para verificar se determinado valor está armazenado em uma Árvore Binária de Busca com um milhão de Nôs uniformemente distribuídos.

Para responder à questão do Exercício 8.11, primeiramente observe o diagrama da [Figura 8.25](#) e tente responder: quantos níveis teria uma Árvore com um milhão de Nôs uniformemente distribuídos?

No nível 1, é possível armazenar um único elemento, e teríamos que visitar, no máximo, um Nô (esse Nô do nível 1) para saber se um elemento X está na Árvore. Do nível 1 ao nível 2 é possível armazenar três elementos, e pelo algoritmo do Exercício 8.2 precisaríamos visitar, no máximo, dois Nôs para saber se um valor X está na Árvore. Isso porque, ao chegar na Raiz do nível 1, visitamos o Nô e, se o valor X que procuramos



Figura 8.25 Níveis de uma ABB.

for menor que a informação armazenada na Raiz, continuaremos procurando no nível 2 apenas na Subárvore Esquerda. Se X for maior que a informação armazenada na Raiz, continuaremos procurando no nível 2 apenas na Subárvore Direita. Ou seja, no nível 2 procuramos em apenas uma das Subárvores; nunca em ambas.

Do nível 1 ao nível 3 cabem sete elementos. Quantos Nós temos que visitar, no máximo, para saber se X está na ABB? Três visitas, no máximo: uma visita para cada nível. Do nível 1 ao nível 4, quatro visitas, no máximo, e cabem 15 elementos. Do nível 1 ao nível 5, são cinco visitas, no máximo, e cabem 31 elementos.

Em uma ABB de N níveis, com Nós uniformemente distribuídos, precisamos visitar, no máximo, N Nós para saber se um valor X está ou não na Árvore. Quantos elementos cabem em uma ABB de N níveis?

Em uma ABB de N níveis cabem $2^N - 1$ Nós. Observe, na Figura 2.26, que uma ABB com 20 níveis pode abrigar, aproximadamente, um milhão de elementos. Em 30 níveis, a ABB pode abrigar aproximadamente um bilhão; em 40 níveis, aproximadamente um trilhão, e assim por diante. Uma análise semelhante pode ser consultada em [Drozdek \(2002, p. 227\)](#). Veja também [Celes \(2004, p. 194\)](#).

Se uma ABB uniformemente distribuída tiver 20 níveis, ela terá cerca de um milhão de elementos, e poderemos saber se um elemento X está nessa Árvore visitando, no máximo, 20 Nós.

Encontrar um valor X entre um milhão de elementos visitando, no máximo, 20 Nós é um bom desempenho, não é? Esse desempenho de uma Árvore Binária de Busca é especialmente significativo quando a quantidade de elementos armazenados é grande. Para encontrar um valor X em uma Lista Encadeada com um milhão de elementos poderia ser necessário visitar, no pior caso, um milhão de Nós.

Esse excelente desempenho só é possível se a Árvore Binária de Busca tiver seus Nós uniformemente distribuídos, como na [Figura 8.25](#). Os algoritmos de inserção e eliminação

Níveis na Árvore	Quantos Nós cabem na Árvore
1	1
2	3
3	7
4	15
5	31
N	$2^N - 1$
10	1.023
13	8.191
16	65.535
18	262.143
20	1 milhão (aproximadamente)
30	1 bilhão (aproximadamente)
40	1 trilhão (aproximadamente)

Figura 8.26 Níveis e quantidade de elementos em uma ABB uniformemente distribuída.

de Nós que elaboramos não garantem que a Árvore permaneça uniformemente distribuída. No próximo capítulo, ajustaremos os algoritmos para que o equilíbrio da Árvore e seu desempenho sejam garantidos.

ABB proporciona agilidade em consultas

Uma ABB permite consultas rápidas, mesmo quando a quantidade de elementos é grande.

8.6 Aplicações de Árvores

Nos exemplos e algoritmos que estudamos até aqui, cada Nô da Árvore Binária de Busca abrigava apenas a informação utilizada como chave de pesquisa. Mas podemos inserir outras informações no Nô, além da chave de pesquisa.

Por exemplo, suponha que temos uma ABB cuja chave de pesquisa seja o nome da pessoa. Isso significa que cada Nô daquela ABB armazena o nome de uma pessoa e também que a ABB está ordenada em função dos nomes. Suponha ainda que os Nôs armazenem o número de telefone daquelas pessoas, como no diagrama da Figura 2.27.

Um exemplo de consulta a essa estrutura seria: “Qual é o telefone da Ana Cláudia?”. Buscamos, então, o Nô da ABB que contém o valor ‘Ana Cláudia’ (algoritmo da [Figura 8.10](#)) e retornamos o telefone armazenado no Nô encontrado.

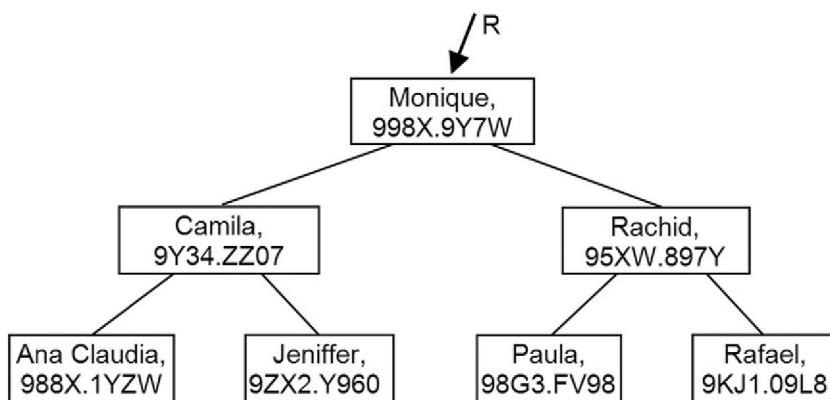
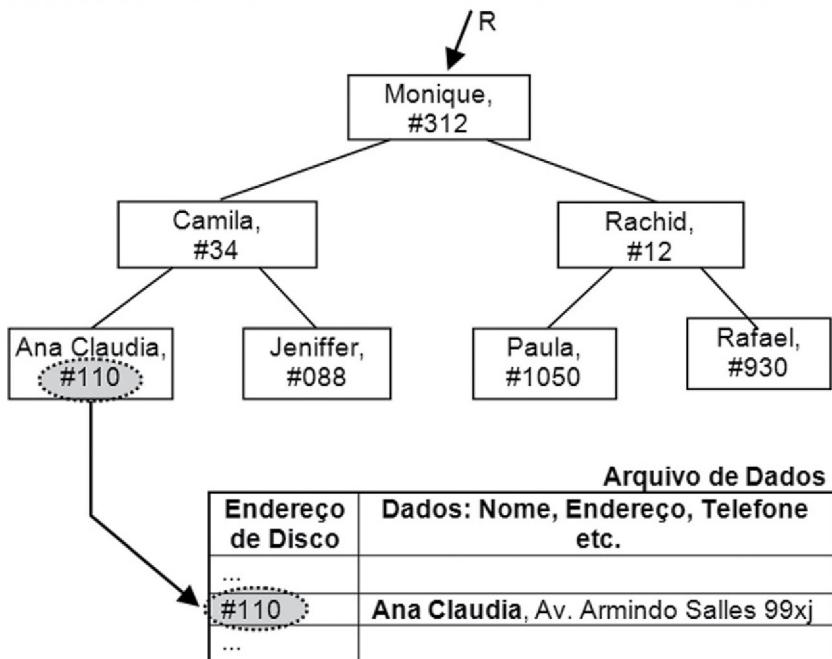


Figura 8.27 ABB com chave de busca e outras informações armazenadas no Nô.

Índices para arquivos

Em vez de armazenar dentro do próprio Nô um conjunto de informações associadas a uma chave de pesquisa, podemos armazenar no Nô a chave de pesquisa e o endereço de disco onde estão armazenadas as informações relativas àquela chave. Por exemplo, no diagrama da [Figura 8.28](#) buscamos o Nô que contém o nome *Ana Cláudia*. No mesmo Nô, encontramos o endereço de disco onde estão armazenados os dados da *Ana Cláudia*. Então vamos ao endereço de disco indicado e encontramos todas as informações disponíveis sobre a *Ana Cláudia*.

**Figura 8.28** ABB como índice para um arquivo.

Esse é o conceito de Índice para Arquivos. A maioria dos sistemas de gerenciamento de banco de dados utiliza alguma variação de Árvore para implementar seus índices. Se, ao programar ou configurar um sistema de banco de dados, você indicar que determinado campo do arquivo será indexado, o sistema montará um índice semelhante ao da **Figura 8.28**, tendo como chave de pesquisa as informações do campo em questão, por exemplo, o campo Nome da Pessoa.

B-Trees

Uma generalização interessante das Árvores Binárias de Busca, direcionada ao armazenamento em disco, é a B-Tree e suas variações. Nas Árvores Binárias de Busca, temos uma única chave de pesquisa armazenada em cada Nô. Em uma B-Tree podemos ter várias chaves de pesquisa em um mesmo Nô.

Considere a B-Tree da **Figura 8.29** e tente imaginar como seria o algoritmo para localizar uma chave X em uma Árvore como essa. Suponha, por exemplo, que queremos encontrar a chave de valor 92. O Nô que está na Raiz não contém a chave 92. Então, em que direção continuamos a busca?

Continuamos a busca pelo valor 92 na Subárvore apontada pelo ponteiro P2, pois 92 está entre os valores 70 e 121. Se estivéssemos buscando um valor menor que 70, continuariammos a busca na Subárvore apontada pelo ponteiro P1; se o valor procurado fosse maior que 154, continuariammos a busca na Subárvore apontada por P4; se o valor procurado estivesse entre 121 e 154, continuariammos a busca na Subárvore apontada por P3.

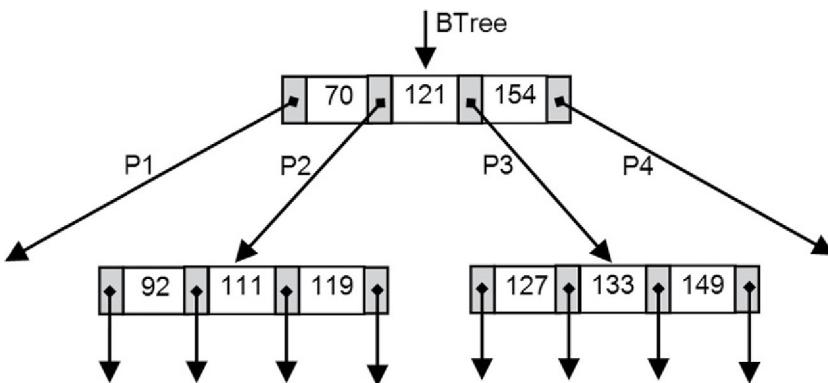


Figura 8.29 B-Tree: várias chaves de pesquisa em um mesmo N o.

QuadTree

Quadtree é uma estrutura utilizada para armazenar informações espaciais. A [Figura 8.30](#) ilustra um tipo de QuadTree — *Region QuadTree* — utilizado para armazenar regiões.

No exemplo da [Figura 8.30](#), subdividimos uma região em quatro quadrantes: Q1, Q2, Q3 e Q4. O quadrante Q1 é homogêneo na cor preta. Armazenamos essa informação em um N o, e não precisamos armazenar mais nada com rela o ao quadrante Q1. Analogamente, o quadrante Q4 é homogêneo na cor branca. Armazenamos essa informa o em um N o, e não precisamos armazenar mais nada com rela o a Q4. Mas os

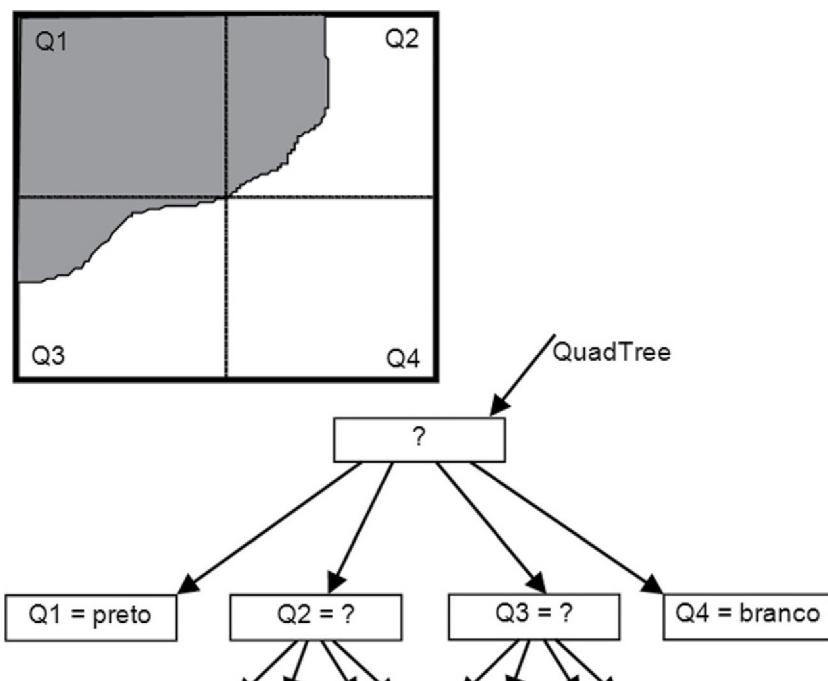


Figura 8.30 Region QuadTree.

quadrantes Q2 e Q3 são heterogêneos. Então, subdividimos recursivamente Q2 e Q3, gerando para cada um deles mais quatro Nós, e assim sucessivamente.

Além da *Region QuadTree*, temos a *Point QuadTree* ou *Point-Region QuadTree*, para armazenamento e busca de informações unidimensionais (ou seja, um *Ponto*, com coordenadas X, Y) em uma região.

Existem diversas variações desse tipo de estrutura para armazenamento de informações espaciais em Árvores. Possíveis benefícios do uso de QuadTrees são a compressão de informações e, principalmente, a agilidade no processamento de operações sobre informações espaciais.

8.7 Avanço de projeto: o Desafio 4

Suponha que queremos desenvolver um jogo em que o computador joga contra um jogador humano. Quando for sua vez de jogar, o computador precisa *escolher* a melhor jogada. Árvores de Decisão para jogos — adaptações de estruturas do tipo Árvore — podem ser utilizadas para essa finalidade.

Árvores de Decisão para jogos servem primeiramente para simular as jogadas futuras e depois servem para ajudar o computador a escolher a melhor jogada. Simplificadamente, o computador procura nas Subárvores de determinada situação de jogo as jogadas em que o computador vence, as jogadas que levam a situações em que ainda é possível vencer, jogadas que levam a situações em que não há risco de derrota, e assim por diante.

[Langsam, Augenstein e Tenembaum \(1996, p. 321-327\)](#) propuseram um algoritmo que utiliza uma Árvore de Decisão como a da [Figura 8.31](#) e avaliam a chance de vitória em uma situação do *jogo da velha*. O algoritmo considera, entre outros fatores, a quantidade de diagonais, horizontais ou verticais ainda em aberto para uma jogada vencedora.

No exemplo da [Figura 8.31](#), na Raiz da Árvore temos uma situação de jogo em que 'X' (o jogador humano) acabou de jogar. A partir dessa situação inicial temos três possíveis jogadas para 'O' (o computador). Uma dessas três opções é uma jogada vencedora. Assim, com base no primeiro nível da Árvore abaixo da Raiz, o computador já consegue tomar uma decisão.

Mas podem existir situações em que uma jogada vencedora não é encontrada no primeiro nível da simulação. Então, a simulação pode avançar mais níveis e aplicar um algoritmo que avalia, para cada Subárvore, a chance de vitória; com base nessa avaliação, o computador escolhe a melhor jogada. Suponha que para cada situação de jogo temos um valor numérico — uma “nota” de 0 a 10 — indicando o quanto aquela situação é boa. Jogadas vencedoras têm nota 10. A nota da Subárvore será o valor máximo dentre as notas de todas as situações de jogo presentes naquela Subárvore.

Essa mesma estratégia — simulação das jogadas futuras para escolha da melhor opção — pode ser aplicada a outros jogos mais complexos como, por exemplo, o xadrez.

Árvores de Decisão dando “inteligência” a um jogo

A simulação de jogadas futuras é um exemplo de Árvore de Decisão utilizada para dar “inteligência” a um jogo. Árvores de Decisão também podem ser utilizadas

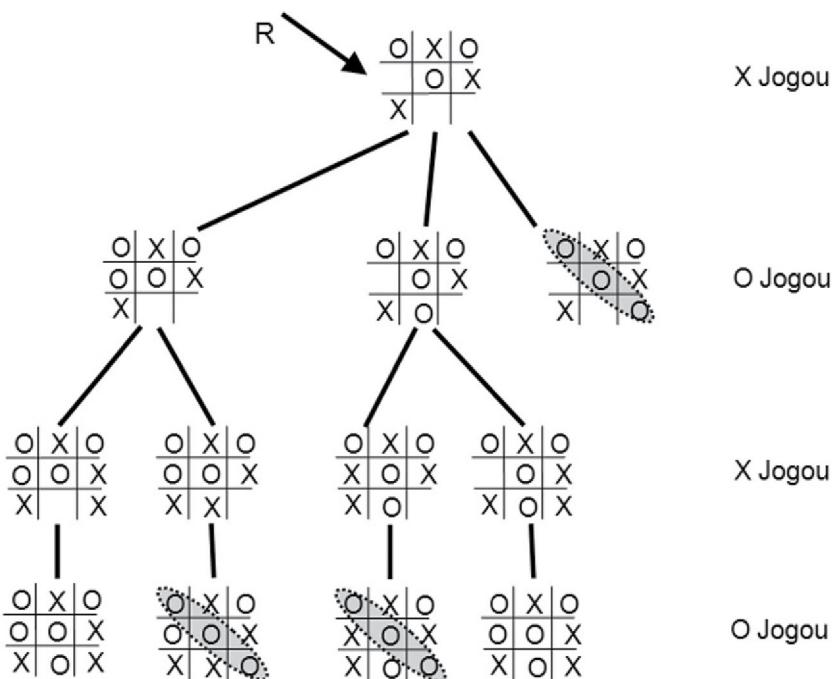


Figura 8.31 Exemplo de Árvore de Decisão para jogos.

para direcionar a lógica do jogo para a situação à esquerda da Raiz caso o usuário humano responda “sim” e para a situação à direita da Raiz caso o usuário humano responda “não”. Árvores de Decisão podem indicar ao jogo o que fazer caso o usuário humano escolha levar a princesa pela estrada estreita ou se escolher levá-la pela estrada larga, e assim por diante.

Exercício 8.12 Discutir aplicações de Árvores em jogos

Identifique alguns jogos que podem ser implementados com o uso de uma estrutura do tipo Árvore. Identifique também outras aplicações, fora do mundo dos games. Sugestão de uso acadêmico: faça uma discussão em grupo. Ao final, cada grupo apresenta a todos os estudantes um novo projeto de Jogo, que ilustre bem a estrutura Árvore.

Exercício 8.13 Avançar o projeto do Desafio 4: defina regras, escolha um nome e inicie o desenvolvimento do seu jogo

No Desafio 4 você deverá desenvolver uma adaptação do *jogo da velha* ou outra aplicação que utilize uma estrutura do tipo Árvore. Se for desenvolver um *jogo da velha*, dê personalidade própria ao seu jogo e escolha um nome que enfatize suas características marcantes. Se for criar um jogo totalmente novo, para cumprir os propósitos acadêmicos pretendidos no Desafio 4, mantenha a característica fundamental: um jogo que utilize uma ou mais Árvores. Sugestão para uso acadêmico: desenvolva o projeto em grupo. Tome as principais decisões em conjunto e divida o trabalho entre os componentes do grupo, cada qual ficando responsável por parte das atividades.

Inicie agora o desenvolvimento do seu jogo referente ao Desafio 4!

Agilidade e suporte a decisões

Árvores são estruturas hierárquicas que proporcionam agilidade na busca e processamento de informações, mesmo quando a quantidade de elementos armazenados é grande. Árvores de Decisão podem ser utilizadas para dar inteligência a um jogo, auxiliando na escolha da melhor opção de jogada ou em outras decisões. É possível adaptar estruturas bem conhecidas, como as Árvores Binárias de Busca, e propor Árvores diferenciadas que atendam a necessidades específicas de sua aplicação.

Consulte nos Materiais Complementares

Vídeos sobre Árvores



Animações sobre Árvores



<http://www.elsevier.com.br/edcomjogos>



Exercícios de fixação

Exercício 8.14 Considere a Árvore da [Figura 8.31](#). Indique:

- o número máximo de Subárvores;
- a Altura da Árvore;
- os Nós Terminais.

Exercício 8.15 Qual seria a sequência de processamento dos elementos ao percorrermos a Árvore Binária de Busca da [Figura 8.17](#) em:

- Pré-ordem (sequência: Raiz, Subárvore Esquerda, Subárvore Direita)?
- In ordem (sequência: Subárvore Esquerda, Raiz, Subárvore Direita)?
- Pós-ordem (sequência: Subárvore Esquerda, Subárvore Direita, Raiz)?

Exercício 8.16 Compare o uso de uma Árvore Binária de Busca e uma Lista, para armazenamento de um conjunto de elementos. Em quais situações o uso da ABB seria vantajoso? Quais seriam as desvantagens?

Exercício 8.17 Desenvolva um algoritmo para calcular a Altura (número de níveis) de uma Árvore Binária R.

Exercício 8.18 Execute algum simulador de operações em uma Árvore Binária de Busca, como, por exemplo, o Tree Explorer ([link 1](#)). Execute operações para inserir e eliminar elementos.



Soluções para alguns dos exercícios

Exercício 8.4 Algoritmo recursivo: soma dos elementos de uma Árvore

```
Inteiro Soma (parâmetro por referência R do tipo ÁrvoreBinária) {  
    /* Retorna a soma do valor de cada um dos elementos de R. Elementos do tipo Inteiro */  
  
    Se (R == Null)                                // Caso 1: árvore vazia  
        Então Retorne 0 ;                         // a soma dos elementos é zero  
    Senão Retorne R→Info + Soma (R→Esq) + Soma (R→Dir);  
        // Caso 2: R é não nulo. Adiciona R→Info à soma dos resultados das chamadas recursivas para as subárvore  
    } // fim Soma
```

Exercício 8.5 Algoritmo recursivo: número de Nós com um único Filho

```
Inteiro NósCom1ÚnicoFilho (parâmetro por referência R do tipo ÁrvoreBinária) {  
    /* Retorna o número de Nós de R com um único Filho */  
    Se (R == Null)                                // Caso 1: árvore vazia  
        Então Retorne 0;                          // número de nós com 1 único filho é zero  
    Senão Se ((R→Dir==Null) E (R→Esq!=Null)) OU ((R→Dir!=Null) E (R→Esq==Null))  
        Então Retorne (1 + NósCom1ÚnicoFilho (R→Esq) + NósCom1ÚnicoFilho (R→Dir));  
            // Caso 2: 1 dos filhos de R é não nulo e o outro é nulo. Pelo menos 1 dos Nós da Árvore possui 1 único Filho.  
            Adiciona 1 à soma do resultados das chamadas recursivas para as subárvore esquerda e direita */  
        Senão Retorne (0+NósCom1ÚnicoFilho (R→Esq)+NósCom1ÚnicoFilho (R→Dir));  
            // Caso 3: o nó da raiz não tem 1 único Filho. Logo, adiciona 0 (zero) à soma dos resultados das chamadas  
            recursivas para as subárvore esquerda e direita */  
    } // fim NósCom1ÚnicoFilho
```

Exercício 8.6 Algoritmo recursivo: Árvores são iguais?

```
Boolean Iguals (parâmetros por referência R1, R2 do tipo ÁrvoreBinária) {  
    /* Verifica se R1 e R2 são iguais. Duas Árvores vazias são iguais. Duas Árvores não vazias são  
    iguais se armazenam valores iguais em suas raízes, se suas Subárvore Esquerdas são iguais e  
    suas Subárvore Direitas também são iguais */  
    Se ((R1 == Null) E (R2 == Null)) // Caso 1: Ambas vazias  
    Então Retorne Verdadeiro;  
    Senão // Caso 2: uma vazia e a outra não vazia  
        Se ((R1==Null) E (R2!=Null)) OU ((R1!=Null) E (R2==Null))  
        Então Retorne Falso;  
        Senão // nenhuma das árvores é vazia  
            Se ((R1→Info == R2→Info) E (Iguals(R1→Esq, R2→Esq)) E (Iguals(R1→Dir, R2→Dir)))  
            Então Retorne Verdadeiro; // são iguais!  
            Senão Retorne Falso; // não são iguais.  
    } // fim Iguals
```

Exercício 8.7 Algoritmo recursivo: é Árvore Binária de Busca?

```

Boolean É_ABB (parâmetro por referência R do tipo ÁrvoreBinária) {
// verifica se R é ou não é uma Árvore Binária de Busca, segundo a definição da Figura 8.4.
Elementos do tipo Inteiro.
Se (R == Null) // Caso 1: árvore vazia
Então Retorne Verdadeiro;
Senão /* Caso 2: árvore não nula. Tenta falsificar verificando se "TemAlguémMaior", ou seja, se
há algum Nô de valor
menor que R→Info na subárvore esquerda de R, ou se "TemAlguémMenor", ou seja, se
há algum Nô de valor
maior que R→Info na subárvore direita */
Se (TemAlguémMaior(R→Esq, R→Info) OU TemAlguémMenor(R→Dir, R→Info))
Então Retorne Falso;
Senão /* pelo Nô atual está tudo ok, mas é preciso verificar se as subárvore esquerda
e direita também são
ABBs. Este é o critério 3 da definição da Figura 8.4 */
Se (É_ABB(R→Esq) E (É_ABB(R→Dir)) // subárvore esq é ABB E subárvore dir
Então Retorne Verdadeiro;
Senão Retorne Falso;
} fim É_ABB

```

```

Boolean TemAlguémMaior (parâmetro por referência R do tipo ÁrvoreBinária, parâmetro X do tipo
Inteiro) {
Se (R == Null)
Então Retorne Falso; // não tem nenhum valor maior que X na árvore R
Senão Se (R→Info > X)
Então Retorne Verdadeiro;
Senão Retorne (TemAlguémMaior( R→Esq, X) OU TemAlguémMaior( R→Dir, X));
} fim TemAlguémMaior

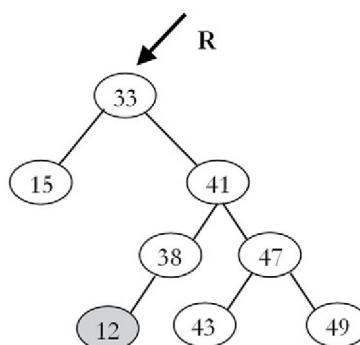
```

```

Boolean TemAlguémMenor (parâmetro por referência R do tipo ÁrvoreBinária, parâmetro X do tipo
Inteiro) {
Se (R == Null)
Então Retorne Falso; // não tem nenhum valor menor que X na árvore R
Senão Se (R→Info < X)
Então Retorne Verdadeiro;
Senão Retorne (TemAlguémMenor( R→Esq, ) OU TemAlguémMenor( R→Dir, X));
} // fim TemAlguémMenor

```

Comentário: Não basta verificar se a informação do filho direito de R é maior que a informação de R e verificar se a informação do filho esquerdo de R é menor que a





informação de R. É preciso verificar a informação de *cada Nó* das subárvores esquerda e direita. Veja na Figura uma situação em que é preciso verificar *cada Nó* da subárvore. O Nó que contém o valor 12 faz com que a Árvore não seja uma Árvore Binária de Busca */

Exercício 8.9 Algoritmo Remove em uma ABB

Remove (parâmetro por referência R tipo ABB, parâmetro X do tipo Inteiro, parâmetro por referência Ok tipo Booleano) {

// Remove X da ABB R. Ok retorna Verdadeiro para o caso de X ter sido encontrado e removido, e // Falso caso contrário

Variável Aux do tipo NodePtr;

Se (R == Null)

Então Ok = Falso; // Caso 1: Árvore vazia: não remove e encerra o algoritmo.

Senão Se (R→Info > X)

Então Remove (R→Esq, X , Ok); // Caso 3: remove X da Subárvore Esq de R

Senão Se (R→Info < X)

Então Remove(R→Dir, X, Erro); // Caso 4: remove X da Subárv Dir de R

Senão

/* Caso 2: Encontrou X - Remove e Ajusta a Árvore. Existem três casos: Nó com 0, 1 ou 2 Filhos – Figuras 8.23 e 8.24 */

{

Aux = R;

Ok = Verdadeiro;

Se (R→Esq = Null E R→Dir = Null) // Caso 2a: Zero Filhos

Então { DeleteNode(Aux); R = Null; } // fim Caso 2a

Senão Se (R→Dir != Null E Esq(R) != Null) // Caso 2c: 2 Filhos

Então {

/* Acha o Nó que contém o Maior Elemento da Subárvore Esquerda de R. O maior é o elemento mais à direita da Subárvore. Ele nunca terá o Filho Direito. */

Aux = R→Esq;

Enquanto (Aux→Dir != Null) Faça Aux = Aux→Dir;

/* Substitui o valor de R→Info - que é o elemento que estarmos querendo eliminar - pelo valor do Maior da Subárvore Esquerda de R. A Árvore ficará com 2 elementos com o mesmo valor. */

R→Info = Aux→Info; // Aux aponta para o Nó que contém o Maior

/* Remove o valor repetido da Subárvore Esquerda de R, através de uma chamada recursiva. Atenção aos parâmetros. Não estamos mais removendo X, mas R→Info, que está repetido. Não estamos mais removendo de R e sim de R→Esq */

Remove(R→Esq, R→Info, Ok);

} // fim Caso 2c

Senão {

// Caso 2b: 1 Único Filho

Se (R→Esq == Null)

Então R = R→Dir;

// "puxa" o Filho Direito; Filho esquerdo é nulo

Senão R = R→Esq;

// "puxa" o Filho Esquerdo; Filho direito é nulo

DeleteNode (Aux);

// desaloca o Nó

} // fim Caso 2b

} // fim remove

Exercício 8.10 ABB Cria, Vazia, Destroi

Sugestão: na operação Destroi, destrua (recursivamente) a Subárvore Esquerda, destrua (recursivamente) a Subárvore Direita e então destrua o Nó da Raiz. Ao final, aponte a Raiz para Null.

Exercício 8.14

Altura: 4; Número Máximo de Subárvores: 3; Nós Terminais (ou Nós Folha): todos os Nós do nível 4, e a Jogada Vencedora do nível 2.

Exercício 8.15

- Pré-ordem: 50, 28, 80, 96;
- In ordem: 28, 50, 80, 96;
- Pós-ordem: 28, 96, 80, 50.

Exercício 8.17 Algoritmo recursivo: altura de uma Árvore

```
Inteiro Altura (parâmetro por referência R do tipo ÁrvoreBinária) {  
    /* Retorna a altura (número de níveis) de uma Árvore R. Elementos do tipo Inteiro. */  
    Variáveis AlturaEsq, AlturaDir do tipo Inteiro;  
    Se (R == Null)           // Caso 1: árvore vazia  
        Então Retorne 0,          // a altura da Árvore é zero  
    Senão { /* Caso 2: R é não nulo. Logo, a altura da árvore é pelo menos 1. Então, a altura da árvore  
    será 1 + a altura da  
        AlturaEsq = Altura (R→Esq);           // altura da subárvore esquerda de R  
        AlturaDir = Altura(R→Dir);           // altura da subárvore direita de R  
        Se AlturaEsq > AlturaDir  
            Então Retorne 1 + AlturaEsq;  
        Senão Retorne 1 + AlturaDir ; }  
    } // fim Altura subárvore esquerda ou 1 + a altura da subárvore direita - dependendo de qual for maior */
```

Links

1. Rocha, V.; Vervloet, M.; Franco, A.; Andrade, C. A. Tree Explorer. EDGames, 2013.
Disponível em: <http://edgames.dc.ufscar.br> (acesso em: setembro de 2013).

Referências e leitura adicional

- Celes, W.; Cerqueira, R.; Rangel, J. L. *Introdução à estrutura de dados*. Rio de Janeiro: Elsevier, 2004.
Drozdek, A. *Estruturas de dados e algoritmos em C++*. São Paulo: Thomson, 2002.
Langsam, Y.; Augenstein, M. J.; Tenenbaum, A. M. *Data Structures Using C and C++*. Upper Saddle River NJ USA: Prentice Hall, 1996.
Pereira, S. L. *Estruturas de dados fundamentais: conceitos e aplicações*. São Paulo: Érica, 1996.