Capítulo 6

Árvores Binárias

Este capítulo introduz uma das estruturas mais fundamentais na Ciência da Computação: árvores binárias. O uso da palavra *árvore* vem do fato que, quando as desenhamos, o resultado frequentemente se assemelha às árvores de uma floresta. Existem muitas maneiras de definir uma árvore binária. Matematicamente, uma *árvore binária* é um grafo conectado, não direcionado, finito, sem ciclos e sem nenhum vértice com grau maior que três.

Para muitas aplicações na ciência da computação, árvores binárias possuem *raízes:* um nó especial, r, com grau de no máximo dois é chamado de *raiz* da árvore. Para cada nó $u \neq r$, o segundo nó no caminho de u para r é chamado de *pai* de u. Cada um dos outros nós adjacentes a u é chamado de *filho* de u. Muitas da árvores binárias em que estamos interessados são *ordenadas*, assim distinguimos entre o *filho esquerdo* e o *filho direito* de u.

Nas ilustrações, árvores binárias são comumente desenhadas da raiz para baixo, com a raiz no topo do desenho e os filhos esquerdo e direito dados respectivamente pelas posições esquerda e direita no desenho. (Figura 6.1). Por exemplo, a Figura 6.2.a mostra uma árvore binária com nove nós.

As árvores binárias são tão importantes que foi criada uma terminologia para elas: a *profundidade* de um nó, u, em uma árvore binária é o comprimento do caminho de u até a raiz da árvore. Se um nó, w, está no caminho de u até r, então w é chamado de *ancestral* de u e u um *descendente* de w. A *subárvore* de um nó, u, é uma árvore binária com raiz em

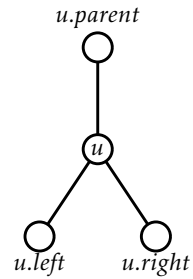

Figura 6.1: O pai, o filho esquerdo e o filho direito do nó u em uma BinaryTree.

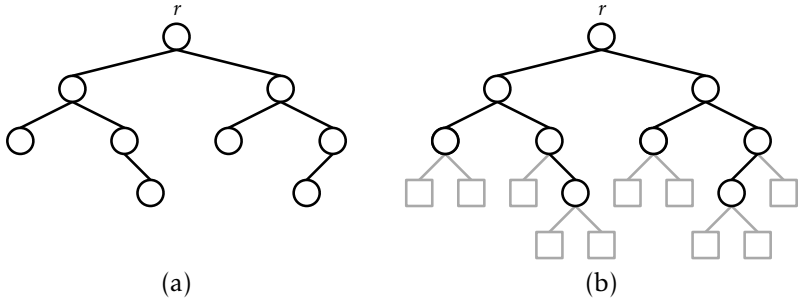

Figura 6.2: Uma árvore binária com (a) nove nós reais e (b) dez nós externos.

u e contém todos os descendentes de u. A *altura* de um nó, u, é o comprimento do percurso mais longo entre u e um dos seus descendentes. A *altura* de uma árvore é a altura de sua raiz. Um nó, u, é uma *folha* se ele não possui filhos.

Algumas vezes pensamos a árvore como se ela fosse expandida com *nós externos*. Qualquer nó que não possua um filho esquerdo possui um nó externo como seu filho esquerdo e, da mesma maneira, qualquer nó que não tenha um filho direito possui um nó externo como seu filho direito (veja Figura 6.2.b). É fácil verificar, por indução, que uma árvore binária com $n \geq 1$ nós reais possui $n+1$ nós externos.

6.1 BinaryTree: Uma Árvore Binária Básica

Um modo simples de representar um nó, u, em uma árvore binária é armazenar explicitamente, e no máximo, três vizinhos de u. Quando um dos três vizinhos não está presente, atribuímos a ele o valor *nil*. Deste modo, ambos os nós externos da árvore e o pai da raiz correspondem ao valor *nil*.

A própria árvore binária pode ser representada por uma referência ao seu nó raiz, r:

```
initialize()
    r ← nil
```

Podemos calcular a profundidade de um nó, u, em uma árvore binária contando o número de passos no caminho de u até a raiz:

```
depth(u)
    d ← 0
    while (u ≠ r) do
        u ← u.parent
        d ← d + 1
    return d
```

6.1.1 Algoritmos Recursivos

Usar algoritmos recursivos torna muito fácil o cálculo envolvendo árvores binárias. Por exemplo, para calcular o tamanho (número de nós) de uma árvore binária com raízes no nó u, recursivamente calculamos o tamanho das duas subárvores com raiz nos filhos de u, somamos os tamanhos e adicionamos um:

```
size(u)
    if u = nil then return 0
    return 1 + size(u.left) + size(u.right)
```

Para calcular a altura de um nó u, podemos calcular a altura das duas subárvores de u, pegar o valor máximo e adicionar 1:

```
height(u)
    if u = nil then return −1
    return 1 + max(height(u.left), height(u.right))
```

6.1.2 Percurso em Árvores Binárias

Ambos os algoritmos da seção anterior usam a recursão para visitar todos os nós de uma árvore binária. Cada um deles visita os nós da árvore binária na mesma ordem que o seguinte código:

```
traverse(u)
    if u = nil then return
    traverse(u.left)
    traverse(u.right)
```

O uso da recursão neste caso produz um código bem sucinto e simples, porém ele pode ser também problemático. A profundidade máxima da recursão é dada pela profundidade máxima do nó na árvore binária, i.e., a

altura da árvore. Se a altura da árvore é muito grande, então essa recursão pode muito bem utilizar mais espaço da pilha do que esteja disponível, causando um fechamento do programa.

Para percorrer uma árvore binária sem utilizar a recursão, você pode usar um algoritmo que se baseie de onde ele vem para saber para onde vai. Veja a Figura 6.3. Se chegamos ao nó u a partir de $u.pai$, então a próxima coisa a ser feita é visitar $u.esquerdo$. Se chegamos a u por $u.esquerdo$, então a próxima coisa a ser feita é visitar $u.direito$. Se chegamos em u por $u.direito$, então terminamos de visitar a subárvore de u, e retornamos para $u.pai$. O código seguinte implementa esta ideia, com o código incluído para lidar com os casos em que qualquer um de $u.esquerdo$, $u.direito$, ou $u.pai$ seja nil:

```
traverse2()
    u ← r
    prv ← nil
    while u ≠ nil do
        if prv = u.parent then
            if u.left ≠ nil then nxt ← u.left
            else if u.right ≠ nil nxt ← u.right
            else nxt ← u.parent
        else if prv = u.left
            if u.right ≠ nil then nxt ← u.right
            else nxt ← u.parent
        else
            nxt ← u.parent
        prv ← u
        u ← nxt
```

Os mesmos resultados que podem ser obtidos usando a recursão também podem ser obtidos desta maneira, sem recursão. Por exemplo, para calcular o tamanho da árvore mantemos um contador, n, e incrementamos n sempre que visitamos um nó pela primeira vez:

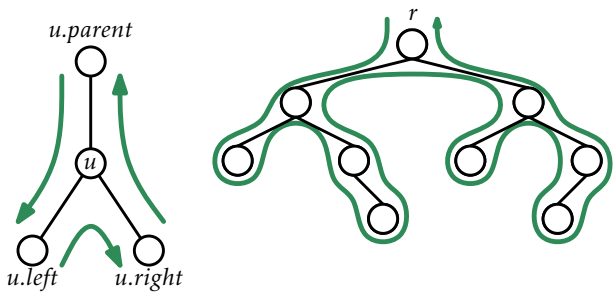

Figura 6.3: Os três casos que ocorrem no nó u quando percorremos uma árvore binária não recursivamente, e o resultado da travessia pela árvore.

```
size2()
    u ← r
    prv ← nil
    n ← 0
    while u ≠ nil do
        if prv = u.parent then
            n ← n + 1
            if u.left ≠ nil then nxt ← u.left
            else if u.right ≠ nil nxt ← u.right
            else nxt ← u.parent
        else if prv = u.left
            if u.right ≠ nil then nxt ← u.right
            else nxt ← u.parent
        else
            nxt ← u.parent
        prv ← u
        u ← nxt
    return n
```

Em algumas implementações de árvore binárias, o campo *pai* não é usado. Quando é este o caso, um implementação não recursiva ainda é possível, porém a implementação deve usar uma Lista (ou Pilha) para

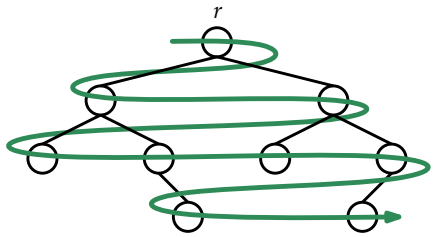

Figura 6.4: Durante um percurso em profundidade, os nós de uma árvore binária são visitados nível por nível e da esquerda para a direita dentro de cada nível.

acompanhar o caminho do nó atual até a raiz.

Um tipo especial de percurso que não cabe no padrão das funções acima é o *percurso em profundidade*. Em um percurso em profundidade, os nós são visitados nível por nível, começando pela raiz e indo para baixo, visitando os nós de cada nível, da esquerda para a direita (veja Figura 6.4). Isto é similar ao modo pelo qual lemos um texto em português. O percurso em profundidade é implementado usando uma fila, q, que inicialmente contém apenas a raiz, r. Em cada passo, extraímos o próximo nó, u, de q, processamos u e adicionamos $u.esquerdo$ e $u.direito$ (se eles não são nil) a q:

```
bf_traverse()
    q ← ArrayQueue()
    if r ≠ nil then q.add(r)
    while q.size() > 0 do
        u ← q.remove()
        if u.left ≠ nil then q.add(u.left)
        if u.right ≠ nil then q.add(u.right)
```

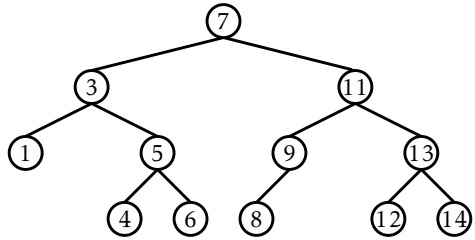

Figura 6.5: Uma árvore binária de busca.

6.2 BinarySearchTree: Uma Árvore Binária de Busca não Balanceada

Uma BinarySearchTree é um tipo especial de árvore binária na qual cada nó, u, também armazena um valor, $u.x$, de uma ordem total. Os valores em uma árvore binária de busca obedece à *propriedade da árvore binária de busca*: para um nó, u, cada valor armazenado na subárvore com raiz em $u.esquerdo$ é menor que $u.x$ e cada valor armazenado na subárvore com raiz em $u.direito$ é maior que $u.x$. Um exemplo de uma BinarySearchTree é mostrado na Figura 6.5.

6.2.1 Busca

A propriedade da árvore binária de busca é extremamente útil porque ela permite localizar rapidamente um valor, x, dentro da árvore binária de busca. Para isto, começamos procurando por x na raiz, r. Quando examinamos um nó, u, podem ocorrer três casos:

1. Se $x < u.x$, então a procura continua em $u.esquerdo$;
2. Se $x > u.x$, então a procura continua em $u.direito$;
3. Se $x = u.x$, então encontramos o nó u que contém x.

A busca termina quando o Caso 3 ocorre ou quando $u \leftarrow nil$. No primeiro

caso, encontramos x. No último caso, concluímos que x não está na árvore binária de busca.

```
find_eq(x)
    w ← r
    while w ≠ nil do
        if x < w.x then
            w ← w.left
        else if x > w.x
            w ← w.right
        else
            return w.x
    return nil
```

Dois exemplos de busca em uma árvore binária de busca são mostrados na Figura 6.6. Como é mostrado no segundo exemplo, mesmo se não encontramos x na árvore, ainda obtemos alguma informação valiosa. Se olhamos para o último nó, u, no qual o Caso 1 ocorre, percebemos que $u.x$ é o menor valor na árvore que é maior que x. De modo análogo, o último nó no qual o Caso 2 ocorre contém o maior valor na árvore que é menor x. Deste modo, guardando a informação do último nó, z, no qual o Caso 1 ocorre, uma BinarySearchTree pode implementar a operação encontra(x) que retorna o menor valor armazenado que é maior que ou igual a x:

```
find(x)
    w ← r
    z ← nil
    while w ≠ nil do
        if x < w.x then
            z ← w
            w ← w.left
        else if x > w.x
            w ← w.right
        else
            return w.x
```

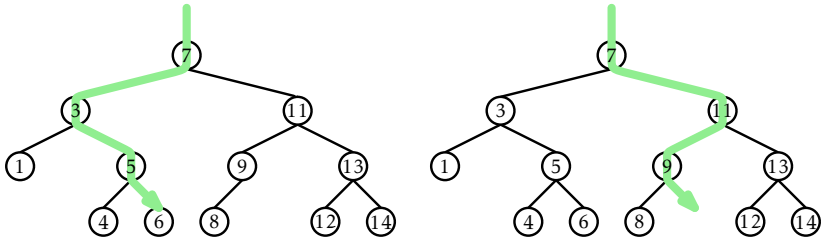

Figura 6.6: Um exemplo de (a) uma busca com sucesso (por 6) e (b) uma busca frustrada (por 10) em uma árvore binária de busca.

```
    if z = nil then return nil
    return z.x
```

6.2.2 Inserção

Para inserir um novo valor, x, a uma BinarySearchTree, procuramos primeiro por x. Se o encontramos, então não precisamos inseri-lo. Caso contrário, armazenamos x em um filho do último nó, p, encontrado durante a busca por x. Se o novo nó é o filho esquerdo ou direito de p depende do resultado da comparação de x e $p.x$.

```
add(x)
    p ← find_last(x)
    return add_child(p, new_node(x))
```

```
find_last(x)
    w ← r
    prev ← nil
    while w ≠ nil do
        prev ← w
```

```
        if (x < w.x) then
            w ← w.left
        else if (x > w.x)
            w ← w.right
        else
            return w
    return prev
```

```
add_child(p, u)
    if p = nil then
        r ← u # inserting into empty tree
    else
        if u.x < p.x then
            p.left ← u
        else if u.x > p.x
            p.right ← u
        else
            return false # u.x is already in the tree
        u.parent ← p
    n ← n + 1
    return true
```

Um exemplo é mostrado na Figura 6.7. A parte que consome mais tempo neste processo é a busca inicial por x, que demanda um tempo proporcional à altura do nó recém adicionado u. No pior caso, isto é igual à altura da BinarySearchTree.

6.2.3 Remoção

Apagar um valor armazenado em um nó, u, de uma BinarySearchTree é um pouco mais difícil. Se u é uma folha, então podemos simplesmente desligar u do seu pai. Melhor ainda: se u possui somente um filho, nós podemos separar u da árvore fazendo $u.pai$ adotar o filho de u (veja Figura 6.8):

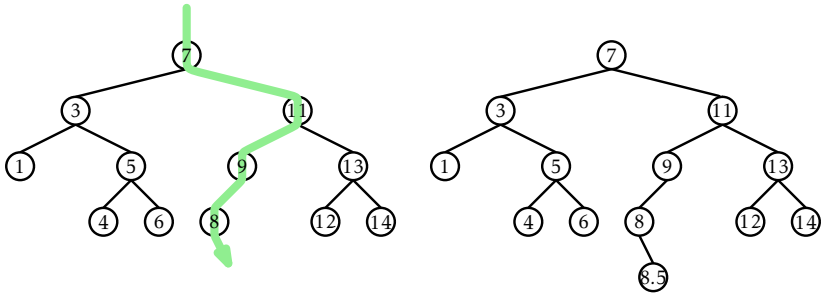

Figura 6.7: Inserindo o valor 8.5 na árvore binária de busca.

```
splice(u)
    if u.left ≠ nil then
        s ← u.left
    else
        s ← u.right
    if u = r then
        r ← s
        p ← nil
    else
        p ← u.parent
        if p.left = u then
            p.left ← s
        else
            p.right ← s
    if s ≠ nil then
        s.parent ← p
    n ← n − 1
```

As coisas ficam complicadas, contudo, quando u possui dois filhos. Neste caso, a coisa mais simples a fazer é encontrar um nó, w, que possua menos que dois filhos de modo que $w.x$ possa substituir $u.x$. Para manter a propriedade da árvore binária de busca, o valor $w.x$ deveria ser próximo ao valor de $u.x$. Por exemplo, escolher w tal que $w.x$ seja o menor valor

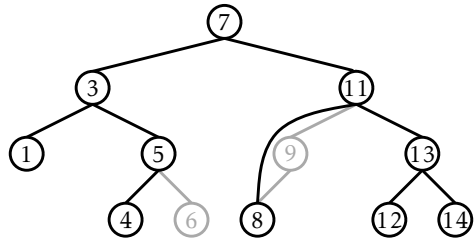

Figura 6.8: Removendo uma folha (6) ou um nó com apenas um filho (9) é fácil.

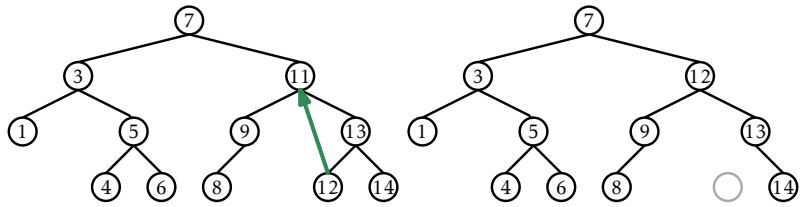

Figura 6.9: Apagar um valor (11) de um nó, u, com dois filhos é realizado substituindo o valor de u pelo menor valor na subárvore direita de u.

maior que $u.x$ irá funcionar perfeitamente. Encontrar o nó w é fácil; ele é o menor valor na subárvore com raiz em $u.direito$. Este nó pode ser removido facilmente porque ele não possui filho esquerdo (veja Figura 6.9).

```
remove_node(u)
    if u.left = nil or u.right = nil then
        splice(u)
    else
        w ← u.right
        while w.left ≠ nil do
                w ← w.left
        u.x ← w.x
        splice(w)
```

6.2.4 Resumo

Cada uma das operações encontrar(x), inserir(x), e remover(x) em uma BinarySearchTree envolve seguir um caminho da raiz da árvore até algum nó árvore. Sem conhecer mais sobre o formato da árvore é difícil dizer muito mais sobre o comprimento deste caminho, exceto que ele é menor que n, o número de nós na árvore. O teorema seguinte (não impressionante) resume o desempenho de uma estrutura de dados BinarySearchTree:

Teorema 6.1. *BinarySearchTree implementa a interface ConjuntoOrdenado e suporta as operações* inserir(x), remover(x), *e* encontrar(x) *em* $O(n)$ *tempos por operação.*

Teorema 6.1 se compara de modo inferior com o Teorema 4.1, que mostra que a estrutura SkiplistConjuntoOrdenado pode implementar a interface ConjuntoOrdenado com tempo esperado de $O(\log n)$ por operação. O problema com a estrutura da BinarySearchTree é que ela pode ficar *desbalanceada*. Em vez de parecer como a árvore na Figura 6.5 ela pode parecer uma longa sequência de n nós, com todos, exceto o último nó possuindo exatamente um único filho.

Existem numerosas formas de evitar uma árvore binária de busca desbalanceada, todos os quais levam a estruturas de dados que têm $O(\log n)$ tempos das operações. No Capítulo 7 mostraremos como tempos de operações *esperados* de $O(\log n)$ podem ser atingidos com a aleatoriedade. No Capítulo 8 mostramos como tempos de operações *amortizados* de $O(\log n)$ podem ser alcançados com operações de reconstruções parciais. No Capítulo 9 mostramos como tempos de operações de *pior caso* de $O(\log n)$ podem ser alcançados com uma árvore que não seja binária: uma nos quais os nós podem ter até quatro filhos.

6.3 Discussão e Exercícios

Árvores Binárias vêm sendo usadas para modelar relacionamentos por milhares de anos. Uma razão para isso é que árvores binárias modelam naturalmente árvores de famílias (pedigree). Essas são as árvores nas

quais a raiz é uma pessoa, os filhos esquerdo e direito são os pais da pessoa, e assim sucessivamente, recursivamente. Nos séculos mais recentes árvore binárias têm também sido usadas para modelar árvores de espécie na biologia, nas quais as folhas da árvore representam espécies existentes e os nós internos representam *eventos de especiação* nos quais duas populações de uma única espécie evoluem em duas espécies separadas.

Árvores Binárias de Busca parecem ter sido descobertas independentemente por vários grupos nos anos 1950 [48, Section 6.2.2]. Referências adicionais para tipos específicos de árvores binárias de busca são fornecidas nos capítulos subsequentes.

Quando implementamos uma árvore binária a partir do zero, várias decisões de projeto devem ser tomadas. Uma delas é a questão se cada nó guarda um ponteiro para seu pai ou não. Se a maioria das operações envolvem simplesmente um caminho seguindo da raiz para uma folha, então o ponteiro para o pai é desnecessário, uma perda de espaço e uma fonte potencial de erros de codificação. Por outro lado, a falta do ponteiro para o pai significa que o percurso da árvore deve ser feito recursivamente ou com o uso de uma pilha explícita. Alguns outros métodos (como inserir ou apagar em alguns tipos de árvores binárias de busca desbalanceadas) são também mais complicados pela ausência do ponteiro para o pai.

Outra decisão de projeto está relacionada em como armazenar o pai, os filhos esquerdo e direito em um nó. Na implementação fornecida aqui, esses ponteiros são armazenados como variáveis separadas. Outra opção é armazená-los em um vetor, p, de tamanho 3, de modo que $u.p[0]$ é o filho esquerdo de u, $u.p[1]$ é o filho direito de u, e $u.p[2]$ é o pai de u. O uso do vetor assim implica que algumas sequências do comando **if** podem ser simplificadas em expressões algébricas.

Um exemplo de tal simplificação ocorre durante o percurso na árvore. Se um percurso chega a um nó u por $u.p[i]$, então o próximo nó neste percurso é $u.p[(i+1) \bmod 3]$. Exemplos similares ocorrem quando existe uma simetria esquerda-direita. Por exemplo, o irmão de $u.p[i]$ é $u.p[(i+1) \bmod 2]$. Este truque funciona melhor se $u.p[i]$ é um filho esquerdo ($i = 0$) ou um filho direito ($i = 1$) de u. Em diversos casos isso significa que algum código complicado que de outra maneira precisaria ter ambas versões para os lados esquerdo e direito podem ser escritos

apenas uma vez. Veja os métodos gira_esquerda(u) and gira_direita(u) na página 161 para um exemplo.

Exercício 6.1. Prove que uma árvore binária com $n \geq 1$ nós possui $n-1$ arestas.

Exercício 6.2. Prove que uma árvore binária com $n \geq 1$ nós reais (internos) possui $n+1$ nós externos.

Exercício 6.3. Prove que, se uma árvore binária, T, possui ao menos uma folha, então ou (a) a raiz de T possui no máximo um filho ou (b) T possui mais de uma folha.

Exercício 6.4. Implemente um método não recursivo, tamanho2(u), que calcule o tamanho da subárvore com raiz no nó u.

Exercício 6.5. Escreva um método não recursivo, altura2(u), que calcule a altura do nó u em uma BinaryTree.

Exercício 6.6. Uma árvore binária é *balanceada em tamanho* se, para cada nó u, os tamanhos das subárvores com raiz em u.*esquerdo* e u.*direito* diferem de no máximo um. Escreva um método recursivo, eh_balanceada(), que testa se uma árvore binária é balanceada. Seu método deve executar num tempo $O(n)$. (Certifique-se de testar seu código em algumas árvores grandes com diferentes formas; é fácil escrever um método que leve muito mais que $O(n)$ em tempo de execução.)

Um percurso em *pré-ordem* de uma árvore binária é um percurso que visita cada nó, u, antes de qualquer de seus filhos. Um percurso *em-ordem* visita u após ter visitado todos os nós na subárvore esquerda de u porém antes de visitar qualquer um dos nós da subárvore direita de u. Um percurso em *pós-ordem* visita u somente após ter visitado todos os outros nós nas subárvores de u. A numeração em pré/em/pós-ordem rotula os nós da árvore com inteiros $0,\ldots,n-1$ na ordem na qual eles são encontrados por um percurso pré/em/pós-ordem. Veja Figura 6.10 para um exemplo.

Exercício 6.7. Crie uma subclasse de BinaryTree cujos nós tenham campos para armazenar números de pré-ordem, pós-ordem, e em-ordem. Escreva os métodos recursivos numero_preOrdem(), numero_emOrdem(), e numero_posOrdem() que atribua esses números corretamente. Esses métodos devem executar num tempo $O(n)$.

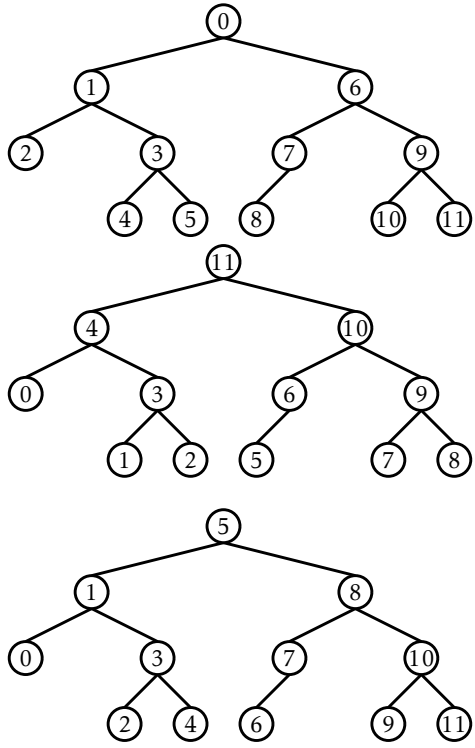

Figura 6.10: numeração em pré-ordem, pós-ordem, e em-ordem de uma árvore binária.

Exercício 6.8. Implemente as funções não recursivas prox_preOrdem(u), prox_emOrdem(u), e prox_posOrdem(u) que retornam o nó seguinte a u em um percurso em pré-ordem, em-ordem, ou pós-ordem, respectivamente. Essas funções devem ter um tempo de execução amortizado constante; se começamos em qualquer nó u e repetidamente chamarmos uma dessas funções e atribuirmos o valor retornado a u até que $u = nil$, então o custo de todas essas chamadas deveria ser de $O(n)$.

Exercício 6.9. Suponha que temos uma árvore binária com números de pré-, pós-, e em-ordem atribuídos aos nós. Mostre como esses números podem ser usados para responder cada uma das seguintes questões em tempo constante:

1. Dado um nó u, determine o tamanho da subárvore com raiz em u.
2. Dado um nó u, determine a profundidade de u.
3. Dados dois nós u e w, determine se u é um ancestral de w

Exercício 6.10. Suponha que você receba uma lista de nós com números de pré-ordem e em-ordem atribuídos a eles. Prove que existe no máximo uma possível árvore com esses números de pré-ordem/em-ordem e mostre como construi-la.

Exercício 6.11. Mostre que o formato de qualquer árvore binária com n nós pode ser representada usando no máximo $2(n-1)$ bits. (Dica: pense sobre gravar o que acontece durante o percurso e então recuperar este registro para reconstruir a árvore.)

Exercício 6.12. Ilustre o que acontece quando adicionamos o valor 3.5 e depois 4.5 na árvore binária de busca na Figura 6.5.

Exercício 6.13. Ilustre o que acontece quando removemos o valor 3 e depois 5 da árvore binária de busca na Figura 6.5.

Exercício 6.14. Implemente um método **obtemLE(x)**, para a BinarySearchTree, que retorne uma lista de todos os itens em uma árvore que sejam menores que ou iguais a x. O tempo de execução do seu método deve ser $O(n' + h)$ no qual n' é o número de itens menores que ou iguais a x e h é a altura da árvore.

Exercício 6.15. Descreva como inserir os elementos $\{1,\ldots,n\}$ para uma BinarySearchTree inicialmente vazia de modo que a árvore resultante tenha altura $n-1$. De quantos modos podemos fazer isso?

Exercício 6.16. Se temos uma BinarySearchTree e executamos a operação inserir(x) seguida por remover(x) (com o mesmo valor de x) necessariamente retornamos à árvore original?

Exercício 6.17. Uma operação remover(x) pode aumentar a altura de algum nó em uma BinarySearchTree? Se sim, de quanto?

Exercício 6.18. Uma operação inserir(x) pode aumentar a altura de algum nó em uma BinarySearchTree? Se sim, de quanto?

Exercício 6.19. Projete e implemente uma versão da BinarySearchTree na qual cada nó, u, mantém os valores $u.tamanho$ (o tamanho da subárvore com raiz em u), $u.profundidade$ (a profundidade de u), e $u.altura$ (a altura da subárvore com raiz em u).

Estes valores devem ser mantidos mesmo durante chamadas a operações de inserir(x) e remover(x), porém isto não deve aumentar o custo dessas operações de mais de um valor constante.