

CAPÍTULO 10

Árvores

Até agora neste livro, descrevemos algumas estruturas de dados sequenciais. A ~~primeira estrutura de dados não sequencial que abordamos no livro foi a tabela hash.~~ Neste capítulo, conheceremos outra estrutura de dados não sequencial chamada árvore (tree), muito útil para armazenar informações que devam ser encontradas facilmente.

Neste capítulo, veremos:

- a terminologia de árvores;
- como criar uma árvore binária de busca;
- como percorrer uma árvore;
- adição e remoção de nós;
- a árvore AVL.

Estrutura de dados de árvore

Uma árvore é um modelo abstrato de uma estrutura hierárquica. O exemplo mais comum de uma árvore na vida real seria o de uma árvore genealógica ou o organograma de uma empresa, como podemos ver na Figura 10.1.

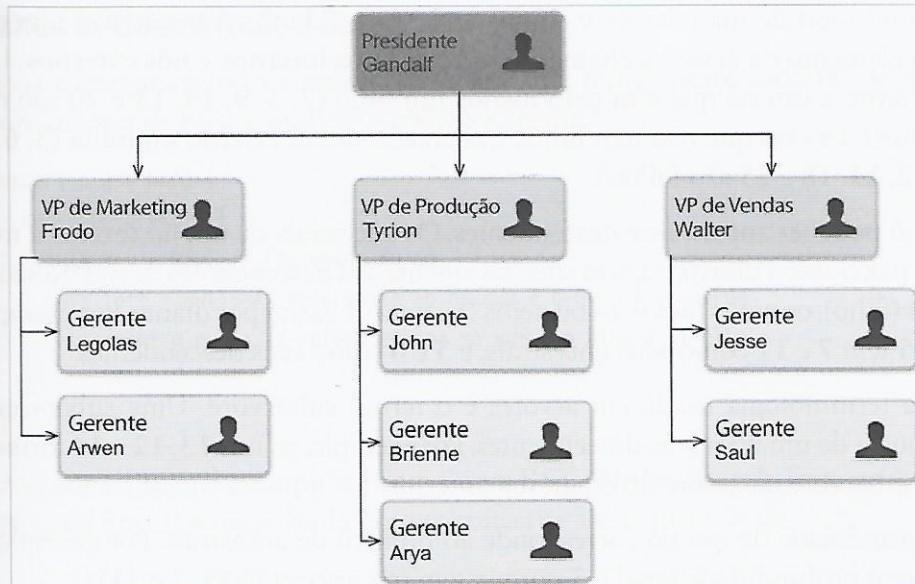


Figura 10.1

Terminologia de árvores

Uma árvore é constituída de **nós** (ou nodos) com um relacionamento pai-filho. Todo nó tem um pai (exceto o primeiro nó no topo) e zero ou mais filhos, como mostra a figura a seguir:

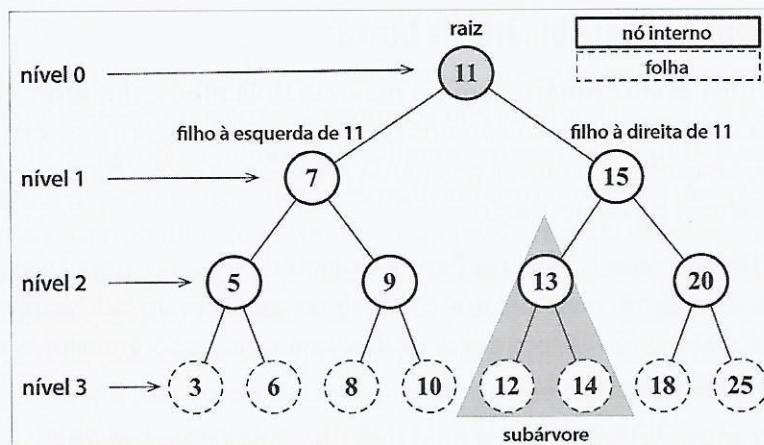


Figura 10.2

O nó no topo de uma árvore é chamado de **raiz** (11). É o nó que não tem pai. Cada elemento da árvore é chamado de nó. Há **nós internos** e **nós externos**. Um nó interno é um nó que tem pelo menos um filho (7, 5, 9, 15, 13 e 20 são nós internos). Um nó que não tem filhos é chamado de nó externo ou **folha** (3, 6, 8, 10, 12, 14, 18 e 25 são folhas).

Um nó pode ter ancestrais e descendentes. Os ancestrais de um nó (exceto a raiz) são o pai, o avô, o bisavô, e assim sucessivamente. Os descendentes de um nó são os filhos (filho), os netos (neto), os bisnetos (bisneto), e assim por diante. Por exemplo, o nó 5 tem 7 e 11 como seus ancestrais, e 3 e 6 como seus descendentes.

Outra terminologia usada em árvores é o termo **subárvore**. Uma subárvore é composta de um nó e seus descendentes. Por exemplo, os nós 13, 12 e 14 formam uma subárvore na árvore do diagrama anterior.

A profundidade de um nó corresponde ao número de ancestrais. Por exemplo, o nó 3 tem profundidade igual a 3 porque tem três ancestrais (5, 7 e 11).

A altura de uma árvore corresponde à profundidade máxima dos nós. Uma árvore também pode ser dividida em níveis. A raiz está no **nível 0**, seus filhos estão no **nível 1**, e assim sucessivamente. A árvore do diagrama anterior tem altura igual a 3 (a profundidade máxima é 3, conforme mostra o **nível 3** na figura anterior).

Agora que vimos os termos mais importantes relacionados às árvores, podemos conhecê-las melhor.

Árvore binária e árvore binária de busca

Um nó em uma **árvore binária** tem no máximo dois filhos: um filho à esquerda e um filho à direita. Essa definição nos permite escrever algoritmos mais eficazes para inserir, pesquisar e remover nós na/da árvore. As árvores binárias são muito usadas em ciência da computação.

Uma **BST** (**Binary Search Tree**, ou Árvore Binária de Busca) é uma árvore binária, mas permite armazenar somente nós com valores menores do lado esquerdo e nós com valores maiores do lado direito. O diagrama da seção anterior exemplifica uma árvore binária de busca.

Essa é a estrutura de dados com a qual trabalharemos neste capítulo.

Cápit...
Criar...
Vamo...
árvo...
exp...
]...
O dia...
Binári...
Assim...
rências...
termin...
cada n...
o nó ar...
balhar...
filho à d...
precisa...
detalhe...
dito de...
de key...
logia de...

Criando as classes Node e BinarySearchTree

Vamos começar criando a nossa classe `Node` que representará cada nó de nossa árvore binária de busca, usando o código a seguir:

```
export class Node {
    constructor(key) {
        this.key = key; // {1} valor do nó
        this.left = null; // referência ao nó que é o filho à esquerda
        this.right = null; // referência ao nó que é o filho à direita
    }
}
```

O diagrama a seguir exemplifica como uma **BST** (Binary Search Tree, ou Árvore Binária de Busca) é organizada no que concerne à estrutura de dados:

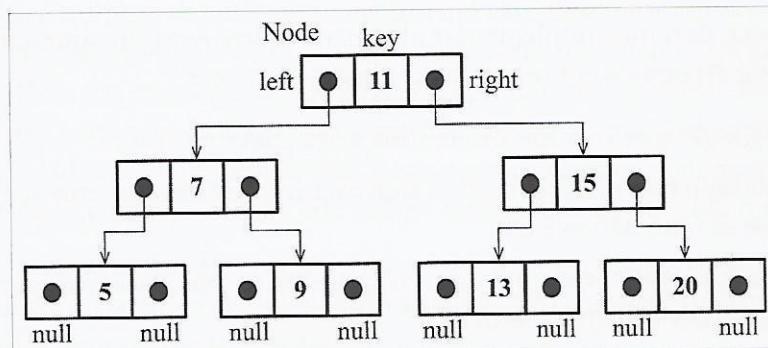


Figura 10.3

Assim como nas listas ligadas, trabalharemos novamente com ponteiros (referências) para representar a conexão entre os nós (chamadas de arestas [edges] na terminologia de árvore). Quando trabalhamos com as listas duplamente ligadas, cada nó tinha dois ponteiros: um para indicar o próximo nó e outro para indicar o nó anterior. Ao trabalhar com árvores, usaremos a mesma abordagem, isto é, trabalharemos também com dois ponteiros. No entanto, um ponteiro referenciará o filho à **esquerda**, enquanto o outro apontará para o filho à **direita**. Por esse motivo, precisaremos de uma classe `Node` que representará cada nó da árvore. Um pequeno detalhe que vale a pena mencionar é que, em vez de chamar o nó propriamente dito de nó ou de item, como fizemos nos capítulos anteriores, nós o chamaremos de `key` ({1}). Uma chave (`key`) é o termo pelo qual um nó é conhecido na terminologia de árvores.

A seguir, declararemos a estrutura básica de nossa classe `BinarySearchTree`:

```
import { Compare, defaultCompare } from '../util';
import { Node } from './models/node';
export default class BinarySearchTree {
    constructor(compareFn = defaultCompare) {
        this.compareFn = compareFn; // usado para comparar os valores dos nós
        this.root = null; // {1} nó raiz do tipo Node
    }
}
```

Seguiremos o mesmo padrão que usamos na classe `LinkedList` (do Capítulo 6, *Listas ligadas*). Isso significa que declararemos também uma variável para que possamos controlar o primeiro nó da estrutura de dados. No caso de uma árvore, em vez de declarar `head`, temos `root` ({1}).

Na sequência, devemos implementar alguns métodos. A seguir, apresentamos os métodos que criaremos em nossa classe `BinarySearchTree`:

- `insert(key)`: esse método insere uma nova chave na árvore.
- `search(key)`: esse método busca a chave na árvore e devolve `true` se ela existir, e `false` se o nó não existir.
- `inOrderTraverse()`: esse método visita todos os nós da árvore usando um percurso em-ordem (in-order).
- `preOrderTraverse()`: esse método visita todos os nós da árvore usando um percurso pré-ordem (pre-order).
- `postOrderTraverse()`: esse método visita todos os nós da árvore usando um percurso pós-ordem (post-order).
- `min()`: esse método devolve a chave/valor mínimo da árvore.
- `max()`: esse método devolve a chave/valor máximo da árvore.
- `remove(key)`: esse método remove a chave da árvore.

Implementaremos cada um desses métodos nas próximas seções.

Inserindo uma chave na BST

Os métodos que criaremos neste capítulo são um pouco mais complexos do que aqueles implementados nos capítulos anteriores. Usaremos muita recursão em

nossos métodos. Se você não tiver familiaridade com recursão, consulte o Capítulo 9, *Recursão*.

O código a seguir é a primeira parte do algoritmo usado para inserir uma nova key em uma árvore:

```
insert(key) {  
    if (this.root == null) { // {1}  
        this.root = new Node(key); // {2}  
    } else {  
        this.insertNode(this.root, key); // {3}  
    }  
}
```

Para inserir um novo nó (ou key) em uma árvore, há dois passos que devemos seguir.

O primeiro é verificar se a inserção constitui um caso especial. O caso especial para a BST é o cenário em que o nó que estamos tentando adicionar é o primeiro nó da árvore ({1}). Se for, tudo que temos a fazer é apontar `root` para esse novo nó ({2}) criando uma instância da classe `Node` e atribuindo-a à propriedade `root`. Por causa das propriedades do construtor de `Node`, basta passar o valor que queremos adicionar na árvore (`key`), e seus ponteiros `left` e `right` terão automaticamente o valor `null`.

O segundo passo consiste na adição do nó em uma posição que não seja o `root`. Nesse caso, precisaremos de um método auxiliar ({3}) para nos ajudar a fazer isso; esse método deve ser declarado assim:

```
insertNode(node, key) {  
    if (this.compareFn(key, node.key) === Compare.LESS_THAN) { // {4}  
        if (node.left == null) { // {5}  
            node.left = new Node(key); // {6}  
        } else {  
            this.insertNode(node.left, key); // {7}  
        }  
    } else {  
        if (node.right == null) { // {8}  
            node.right = new Node(key); // {9}  
        } else {  
            this.insertNode(node.right, key); // {10}  
        }  
    }  
}
```

A função `insertNode` nos ajudará a encontrar o lugar correto para inserir um novo nó. A lista a seguir descreve o que esse método faz:

- Se a árvore não estiver vazia, devemos encontrar um local para adicionar o novo nó. Por esse motivo, chamaremos o método `insertNode` passando o nó raiz e a chave que queremos inserir como parâmetros ({3}).
- Se a chave do nó for menor que a chave do nó atual (nesse caso, é a raiz [{4}]), devemos verificar o filho à esquerda do nó. Observe que estamos usando aqui a função `compareFn`, que pode ser passada no construtor da classe `BST` para comparar os valores, pois `key` pode não ser um número, mas um objeto complexo. Se não houver um nó à esquerda ({5}), a nova key será inserida nesse local ({6}). Caso contrário, devemos descer um nível na árvore chamando `insertNode` recursivamente ({7}). Nesse caso, o nó com o qual faremos a comparação na próxima vez será o filho à esquerda do nó atual (subárvore do nó à esquerda).
- Se a chave do nó for maior que a chave do nó atual e não houver nenhum filho à direita ({8}), a nova chave será inserida nesse local ({9}). Caso contrário, também chamaremos o método `insertNode` recursivamente, porém o novo nó a ser comparado será o filho à direita ({10} – a subárvore do nó à direita).

Vamos aplicar essa lógica em um exemplo para que possamos compreender melhor esse processo. Considere o seguinte cenário: temos uma nova árvore e estamos tentando inserir a sua primeira chave. Nesse caso, executaremos este código:

```
const tree = new BinarySearchTree();
tree.insert(11);
```

Nesse cenário, teremos um único nó em nossa árvore e a propriedade `root` apontará para ele. O código que será executado está nas linhas {1} e {2} de nosso código-fonte.

Vamos supor agora que já temos a árvore a seguir (Figura 10.4):

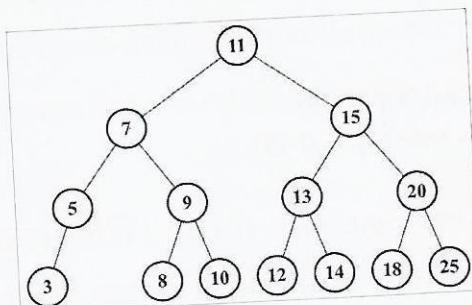


Figura 10.4

O código para criar a árvore do diagrama anterior é uma continuação do código anterior (no qual inserimos a chave 11), conforme vemos a seguir:

```
tree.insert(7);
tree.insert(15);
tree.insert(5);
tree.insert(3);
tree.insert(9);
tree.insert(8);
tree.insert(10);
tree.insert(13);
tree.insert(12);
tree.insert(14);
tree.insert(20);
tree.insert(18);
tree.insert(25);
```

Gostaríamos de inserir uma nova chave cujo valor é 6, portanto executaremos o código a seguir também:

```
tree.insert(6);
```

Eis os passos que serão executados:

1. A árvore não está vazia, portanto o código da linha {3} será executado. O código chamará o método `insertNode(root, key[6])`.
2. O algoritmo verificará a linha {4} (`key[6] < root[11]` é `true`), depois verificará a linha {5} (`node.left[7]` não é `null`) e, por fim, passará para a linha {7} chamando `insertNode(node.left[7], key[6])`.
3. Entraremos no método `insertNode` novamente, porém com parâmetros diferentes. O código verificará a linha {4} de novo (`key[6] < node[7]` é `true`), depois verificará a linha {5} (`node.left[5]` não é `null`) e, por fim, passará para a linha {7} chamando `insertNode(node.left[5], key[6])`.
4. Executaremos o método `insertNode` mais uma vez. O código verificará a linha {4} novamente (`key[6] < node[5]` é `false`), depois passará para a linha {8} (`node.right` é `null` – o nó 5 não tem nenhum filho à direita como descendente) e, por fim, a linha {9} será executada, inserindo a chave 6 como o filho à direita do nó 5.

5. Depois disso, as chamadas dos métodos serão desempilhadas e a execução terminará.

Eis o resultado após a chave 6 ter sido inserida na árvore:

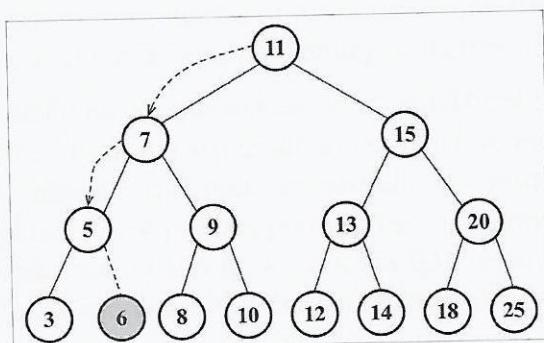


Figura 10.5

Percorrendo uma árvore

Percorrer uma árvore (ou caminhar por ela) é o processo de visitar todos os seus nós e executar uma operação em cada um deles. Entretanto, como devemos fazer isso? Devemos começar do topo da árvore ou da parte inferior? Do lado esquerdo ou do lado direito? Há três abordagens distintas que podem ser usadas para visitar todos os nós de uma árvore: em-ordem (in-order), pré-ordem (pre-order) e pós-ordem (post-order).

Nas próximas seções, exploraremos melhor os usos e as implementações desses três tipos de percurso em árvores.

Percorso em-ordem LVR

Um percurso **em-ordem** (in-order) visita todos os nós de uma BST em ordem crescente, o que significa que todos os nós serão visitados, do menor para o maior. Uma aplicação do percurso em-ordem seria ordenar uma árvore. Vamos observar a sua implementação:

```

inOrderTraverse(callback) {
  this.inOrderTraverseNode(this.root, callback); // {1}
}
  
```

O método `inOrderTraverse` recebe uma função `callback` como parâmetro, a qual pode ser usada para executar a ação desejada quando visitamos o nó (esse padrão é conhecido como visitante [visitor]; para mais informações sobre esse assunto, consulte http://en.wikipedia.org/wiki/Visitor_pattern). Como a maior parte dos algoritmos que estamos implementando para a BST é recursiva, usaremos um método auxiliar que receberá o nó `root` da árvore (ou subárvore) e a função `callback` ({1}). O método auxiliar está listado a seguir:

```
inOrderTraverseNode(node, callback) {  
    if (node != null) { // {2}  
        this.inOrderTraverseNode(node.left, callback); // {3}  
        callback(node.key); // {4}  
        this.inOrderTraverseNode(node.right, callback); // {5}  
    }  
}
```

Para percorrer uma árvore usando o método em-ordem, devemos inicialmente verificar se o `node` da árvore passado como parâmetro é `null` ({2}) – esse é o ponto em que a recursão é interrompida, ou seja, é o caso de base do algoritmo de recursão).

Em seguida, visitamos o nó à esquerda ({3}) chamando a mesma função recursivamente. Então visitamos o nó raiz ({4}) executando uma ação nesse nó (`callback`) e depois visitamos o nó à direita ({5}).

Vamos tentar executar esse método usando a árvore da seção anterior como exemplo, assim:

```
const printNode = (value) => console.log(value); // {6}  
tree.inOrderTraverse(printNode); // {7}
```

Em primeiro lugar, devemos criar uma função de callback ({6}). Tudo que faremos é exibir o valor do nó no console do navegador. Então podemos chamar o método `inOrderTraverse` passando a nossa função de callback como parâmetro ({7}). Quando esse código for executado, a saída a seguir será exibida no console (cada número será mostrado em uma linha diferente):

3 5 6 7 8 9 10 11 12 13 14 15 18 20 25

O diagrama a seguir mostra o caminho que o método `inOrderTraverse` seguiu:

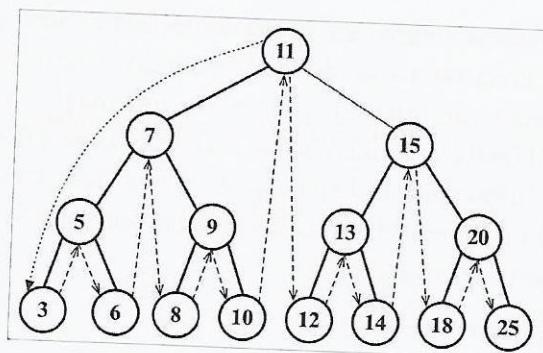


Figura 10.6

Percorso pré-ordem VLR

Um percurso **pré-ordem** (pre-order) visita o nó antes de visitar seus descendentes. Uma aplicação do percurso pré-ordem seria exibir um documento estruturado. Vamos analisar a sua implementação:

```
preOrderTraverse(callback) {
    this.preOrderTraverseNode(this.root, callback);
}
```

Eis a implementação do método `preOrderTraverseNode`:

```
preOrderTraverseNode(node, callback) {
    if (node != null) {
        callback(node.key); // {1}
        this.preOrderTraverseNode(node.left, callback); // {2}
        this.preOrderTraverseNode(node.right, callback); // {3}
    }
}
```

A diferença entre os percursos em-ordem e pré-ordem é que o percurso pré-ordem visita o nó raiz antes ({1}), depois o nó à esquerda ({2}) e, por fim, o nó à direita ({3}), enquanto o percurso em-ordem executa as linhas na seguinte ordem: {2}, {1} e {3}.

A saída a seguir será exibida no console (cada número será mostrado em uma linha diferente):

```
11 7 5 3 6 9 8 10 15 13 12 14 20 18 25
```

O diagrama seguinte mostra o caminho percorrido pelo método `preOrderTraverse`:

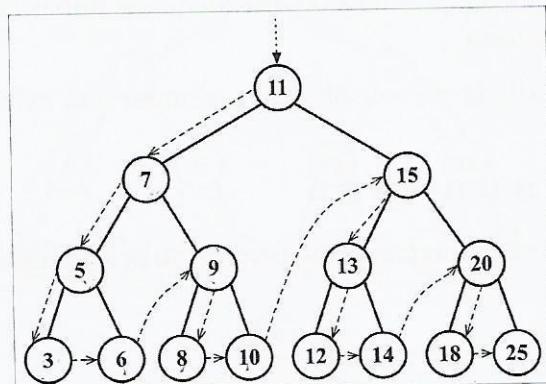


Figura 10.7

Percorso pós-ordem LRV

Um percurso **pós-ordem** (post-order) visita o nó depois de visitar os seus descendentes. Uma aplicação do percurso pós-ordem poderia ser calcular o espaço usado por um arquivo em um diretório e em seus subdiretórios.

Vamos analisar a sua implementação:

```
postOrderTraverse(callback) {  
    this.postOrderTraverseNode(this.root, callback);  
}
```

Eis a implementação de postOrderTraverseNode:

```
postOrderTraverseNode(node, callback) {  
    if (node != null) {  
        this.postOrderTraverseNode(node.left, callback); // {1}  
        this.postOrderTraverseNode(node.right, callback); // {2}  
        callback(node.key); // {3}  
    }  
}
```

Nesse caso, o percurso pós-ordem visitará o nó à esquerda ({1}), depois o nó à direita ({2}) e, por último, o nó raiz ({3}).

Os algoritmos para as abordagens em-ordem, pré-ordem e pós-ordem são muito parecidos; a única diferença está na ordem em que as linhas {1}, {2} e {3} são executadas em cada método.

Esta será a saída exibida no console (cada número será exibido em uma linha diferente):

```
3 6 5 8 10 9 7 12 14 13 18 25 20 15 11
```

O diagrama a seguir mostra o caminho percorrido pelo método `postOrderTraverse`:

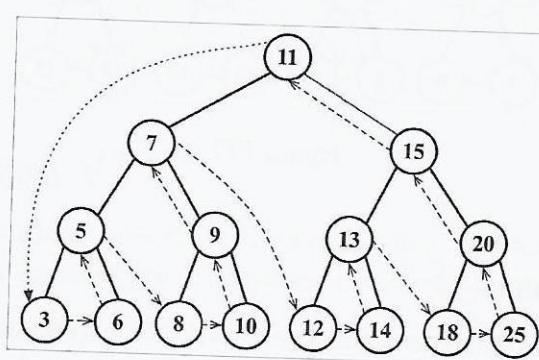


Figura 10.8

Pesquisando valores em uma árvore

Há três tipos de pesquisa geralmente executados em árvores:

- pesquisa de valores mínimos;
- pesquisa de valores máximos;
- pesquisa de um valor específico.

Vamos analisar cada uma dessas pesquisas nas próximas seções.

Pesquisando valores mínimos e máximos

Considere a árvore a seguir (Figura 10.9) em nossos exemplos.

Apenas olhando a Figura 10.9, você poderia encontrar facilmente os valores mínimo e máximo na árvore?

Se obser
é a meno
último r
Essa info
os nós m
Inicialme

```
min() {
  return
}
```

O métod
({1}), decl

```
minNode  
let cu  
while  
curr  
}  
return
```

O método
da árvore.
da própria
root da árv
No método
até encontra

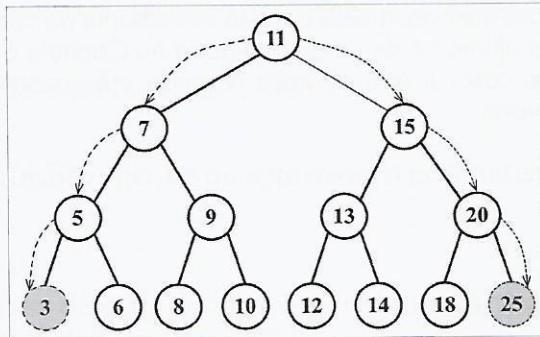


Figura 10.9

Se observar o nó mais à esquerda no último nível da árvore, você verá o valor 3, que é a menor chave nessa árvore; se observar o nó mais distante à direita (também no último nível da árvore), encontrará o valor 25, que é a maior chave nessa árvore. Essa informação nos ajuda muito na implementação de métodos que encontrarão os nós mínimo e máximo da árvore.

Inicialmente vamos analisar o método que encontrará a chave mínima da árvore:

```

min() {
    return this.minNode(this.root); // {1}
}
  
```

O método `min` será o método exposto ao usuário, o qual chama o método `minNode` ({1}), declarado a seguir:

```

minNode(node) {
    let current = node;
    while (current != null && current.left != null) { // {2}
        current = current.left; // {3}
    }
    return current; // {4}
}
  
```

O método `minNode` nos permite encontrar a chave mínima, a partir de qualquer nó da árvore. Podemos usá-lo para encontrar a chave mínima de uma subárvore ou da própria árvore. Por esse motivo, chamaremos o método `minNode` passando o nó `root` da árvore ({1}), pois queremos encontrar a chave mínima de toda a árvore.

No método `minNode`, percorreremos a aresta esquerda da árvore (linhas {2} e {3}) até encontrar o nó no nível mais alto dela (na extremidade esquerda).

i A lógica usada no método `minNode` é muito semelhante ao código que usamos para iterar até o último nó de uma lista ligada no Capítulo 6, *Listas ligadas*. A diferença, nesse caso, é que estamos iterando até encontrar o nó mais à esquerda da árvore.

De modo semelhante, também temos o método `max`, cujo código apresenta o aspecto a seguir:

```
max() {
    return this.maxNode(this.root);
}

maxNode(node) {
    let current = node;
    while (current != null && current.right != null) { // {5}
        current = current.right;
    }
    return current;
}
```

Para encontrar a chave máxima, percorremos a aresta direita da árvore `{5}` até encontrar o último nó na extremidade direita dela.

Assim, para o valor mínimo, sempre percorreremos o lado esquerdo da árvore; para o valor máximo, sempre navegaremos para o lado direito dela.

Pesquisando um valor específico

Nos capítulos anteriores, implementamos também os métodos `find`, `search` e `get` para encontrar um valor específico na estrutura de dados. Implementaremos o método `search` para a BST também. Vamos analisar a sua implementação:

```
search(key) {
    return this.searchNode(this.root, key); // {1}
}

searchNode(node, key) {
    if (node == null) { // {2}
        return false;
    }
    if (this.compareFn(key, node.key) === Compare.LESS_THAN) { // {3}
        return this.searchNode(node.left, key); // {4}
    } else if (
```

edição
os
A
à
ecto
a
até
re;
ra
do

```
        this.compareFn(key, node.key) === Compare.BIGGER_THAN
    ) { // {5}
    return this.searchNode(node.right, key); // {6}
} else {
    return true; // {7}
}
}
```

Nossa primeira tarefa deve ser declarar o método `search`. Seguindo o padrão dos demais métodos declarados para a BST, usaremos um método auxiliar para nos ajudar na lógica de recursão ({1}).

O método `searchNode` pode ser usado para encontrar uma chave específica na árvore ou em qualquer uma de suas subárvores. Esse é o motivo pelo qual chamaremos esse método na linha {1} passando o nó `root` da árvore como parâmetro.

Antes de iniciar o algoritmo, validaremos se o `node` passado como parâmetro é válido (não é `null` ou `undefined`). Se for inválido, é sinal de que a chave não foi encontrada, e devolveremos `false`.

Se o nó não for `null`, devemos continuar a pesquisa. Se a `key` que estamos procurando for menor que o nó atual ({3}), continuaremos a pesquisa usando a subárvore do filho à esquerda ({4}). Se o valor que estamos procurando for maior que o nó atual ({5}), continuaremos a pesquisa a partir do filho à direita do nó atual ({6}). Caso contrário, significa que a chave que estamos procurando é igual à chave do nó atual, e devolveremos `true` para informar que ela foi encontrada ({7}).

Podemos testar esse método usando o código a seguir:

```
console.log(tree.search(1) ? 'Key 1 found.' : 'Key 1 not found.');
console.log(tree.search(8) ? 'Key 8 found.' : 'Key 8 not found.');
```

Eis a saída exibida:

```
Value 1 not found.
Value 8 found.
```

Vamos descrever mais detalhes sobre como o método foi executado para encontrar a chave 1:

1. Chamamos o método `searchNode` passando o `root` da árvore como parâmetro ({1}). `node[root[11]]` não é `null` ({2}), portanto passamos para a linha {3}.
2. `key[1] < node[11]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `node[7]`, `key[1]` como parâmetros.

3. `node[7]` não é `null` ({2}), portanto passamos para a linha {3}.
4. `key[1] < node[7]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `node[5]`, `key[1]` como parâmetros.
5. `node[5]` não é `null` ({2}), portanto passamos para a linha {3}.
6. `key[1] < node[5]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `node[3]`, `key[1]` como parâmetros.
7. `node[3]` não é `null` ({2}), portanto passamos para a linha {3}.
8. `key[1] < node[3]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `null`, `key[1]` como parâmetros. `null` foi passado como parâmetro porque `node[3]` é uma folha (ele não tem filhos, portanto o filho à esquerda será `null`).
9. `node` é `null` (linha {2}, o `node` a ser pesquisado nesse caso é `null`), portanto devolvemos `false`.
10. Depois disso, as chamadas dos métodos serão desempilhadas e a execução terminará.

Vamos fazer o mesmo exercício para pesquisar o valor 8, da seguinte maneira:

1. Chamamos o método `searchNode` passando `root` como parâmetro ({1}). `node[root[11]]` não é `null` ({2}), portanto passamos para a linha {3}.
2. `key[8] < node[11]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `node[7]`, `key[8]` como parâmetros.
3. `node[7]` não é `null` ({2}), portanto passamos para a linha {3}.
4. `key[8] < node[7]` é `false` ({3}), portanto passamos para a linha {5}.
5. `key[8] > node[7]` é `true` ({5}), portanto passamos para a linha {6} e chamamos o método `searchNode` novamente, passando `node[9]`, `key[8]` como parâmetros.
6. `node[9]` não é `null` ({2}), portanto passamos para a linha {3}.
7. `key[8] < node[9]` é `true` ({3}), portanto passamos para a linha {4} e chamamos o método `searchNode` novamente, passando `node[8]`, `key[8]` como parâmetros.
8. `node[8]` não é `null` ({2}), portanto passamos para a linha {3}.
9. `key[8] < node[8]` é `false` ({3}), portanto passamos para a linha {5}.
10. `key[8] > node[8]` é `false` ({5}), portanto passamos para a linha {7} e devolvemos `true` porque `node[8]` é a chave que estamos procurando.

11. Depois disso, as chamadas dos métodos serão desempilhadas e a execução terminará.

Removendo um nó

O próximo e último método que implementaremos para a nossa BST é o método `remove`. Esse é o método mais complexo que implementaremos neste livro. Vamos começar pelo método que estará disponível para ser chamado a partir da instância de uma árvore, assim:

```
remove(key) {
    this.root = this.removeNode(this.root, key); // {1}
}
```

Esse método recebe a `key` que desejamos remover, e chama também `removeNode`, passando `root` e a `key` a ser removida como parâmetros ({1}). Um aspecto muito importante a ser observado é que `root` recebe o valor devolvido pelo método `removeNode`. Entenderemos o porquê em breve.

A complexidade do método `removeNode` se deve aos diferentes cenários com os quais devemos lidar, além do fato de o método ser recursivo.

Vamos observar a implementação de `removeNode`, exibida a seguir:

```
removeNode(node, key) {
    if (node == null) { // {2}
        return null;
    }
    if (this.compareFn(key, node.key) === Compare.LESS_THAN) { // {3}
        node.left = this.removeNode(node.left, key); // {4}
        return node; // {5}
    } else if (
        this.compareFn(key, node.key) === Compare.BIGGER_THAN
    ) { // {6}
        node.right = this.removeNode(node.right, key); // {7}
        return node; // {8}
    } else {
        // key é igual a node.item
        // caso 1
        if (node.left == null && node.right == null) { // {9}
            return null;
        } else if (node.left != null && node.right != null) {
            let successor = this.successor(node);
            node.key = successor.key;
            node.left = this.removeNode(node.left, successor.key);
            return node;
        } else {
            let child = node.left || node.right;
            node.left = node.right = null;
            return child;
        }
    }
}
```

```
node = null; // {10}
return node; // {11}
}
// caso 2
if (node.left == null) { // {12}
    node = node.right; // {13}
    return node; // {14}
} else if (node.right == null) { // {15}
    node = node.left; // {16}
    return node; // {17}
}
// caso 3
const aux = this.minNode(node.right); // {18}
node.key = aux.key; // {19}
node.right = this.removeNode(node.right, aux.key); // {20}
return node; // {21}
}
}
```

Como ponto de parada, temos a linha {2}. Se o nó que estamos analisando for `null`, isso significa que `key` não está presente na árvore e, por esse motivo, devolveremos `null`. Se o nó não for `null`, precisamos encontrar a `key` na árvore. Portanto, se a `key` que estamos procurando tiver um valor menor que o nó atual ({3}), passaremos para o próximo nó da aresta à esquerda da árvore ({4}). Se `key` for maior que o nó atual ({6}), passaremos para o próximo nó na aresta à direita da árvore ({7}), o que significa que analisaremos as subárvore.

Se encontrarmos a chave que estamos procurando (`key` é igual a `node.key`), teremos três cenários distintos para tratar.

Removendo uma folha

O primeiro cenário é aquele em que o nó é uma folha, sem filhos à esquerda nem à direita ({9}). Nesse caso, tudo que temos a fazer é remover o nó atribuindo-lhe o valor `null` ({9}). No entanto, como aprendemos na implementação das listas ligadas, sabemos que atribuir `null` ao nó não é suficiente, e devemos cuidar também das referências (ponteiros). Nesse caso, o nó não tem nenhum filho, mas tem um nó pai. Devemos atribuir `null` em seu nó pai, e isso pode ser feito devolvendo `null` ({11}).

Como o nó já tem o valor `null`, a referência do pai ao nó receberá `null` também, e esse é o motivo pelo qual estamos devolvendo o valor de `node` no método `removeNode`. O nó pai sempre receberá o valor devolvido pelo método. Uma alternativa a essa abordagem seria passar o pai e o `node` como parâmetros do método.

Se observarmos as primeiras linhas de código desse método, veremos que estamos atualizando as referências dos ponteiros da esquerda e da direita dos nós nas linhas {4} e {7}, e estamos também devolvendo o nó atualizado nas linhas {5} e {8}.

O diagrama a seguir exemplifica a remoção de um nó que é uma folha:

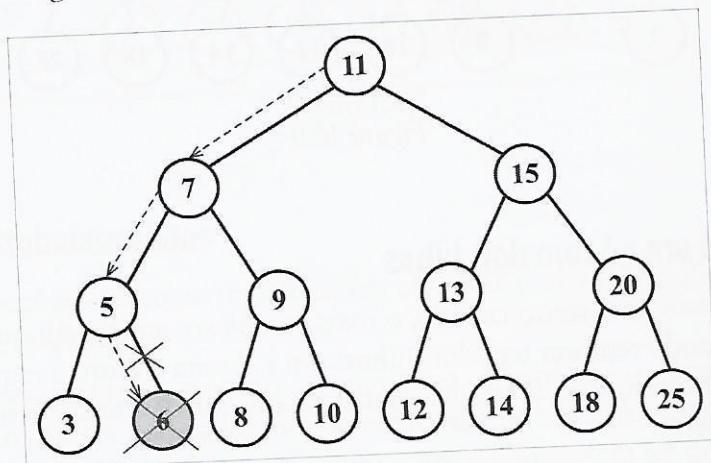


Figura 10.10

Removendo um nó com um filho à esquerda ou à direita

Vamos agora analisar o segundo cenário, aquele em que um nó tem um filho à esquerda ou à direita. Nesse caso, devemos pular esse nó e fazer o ponteiro do pai apontar para o nó filho.

Se o nó não tiver um filho à esquerda ({12}), significa que ele tem um filho à direita, portanto modificaremos a referência do nó para o seu filho à direita ({13}) e devolveremos o nó atualizado ({14}). Faremos o mesmo se o nó não tiver o filho à direita ({15}); atualizaremos a referência do nó para o seu filho à esquerda ({16}) e devolveremos o valor atualizado ({17}).

O diagrama a seguir exemplifica a remoção de um nó com apenas um filho à esquerda ou à direita:

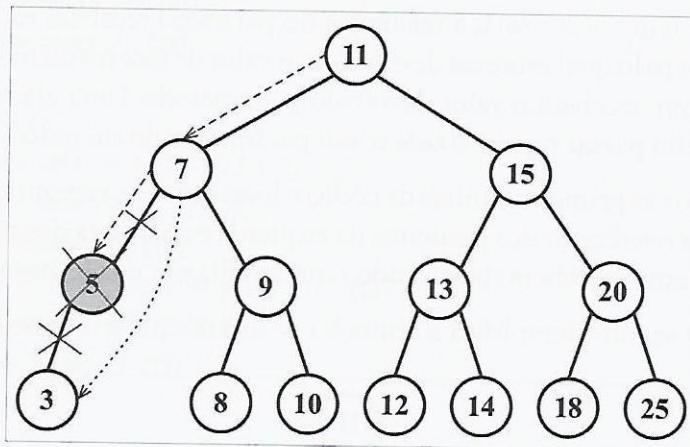


Figura 10.11

Removendo um nó com dois filhos

Agora chegamos ao terceiro cenário, o mais complexo: aquele em que o nó que estamos tentando remover tem dois filhos: um à direita e outro à esquerda. Para remover um nó com dois filhos, há quatro passos que devem ser executados, da seguinte maneira:

1. Depois que o nó que queremos remover for encontrado, precisamos encontrar o nó mínimo da subárvore da aresta à sua direita (o seu sucessor, {18}).
2. Em seguida, atualizamos o valor do nó com a chave do nó mínimo de sua subárvore à direita ({19}). Com essa ação, estamos substituindo a chave do nó, o que significa que ele foi removido.
3. No entanto, agora temos dois nós na árvore com a mesma chave, e isso não pode ocorrer. O que devemos fazer agora é remover o nó mínimo da subárvore à direita, pois ele foi transferido para o local em que estava o nó removido ({20}).
4. Por fim, devolvemos a referência ao nó atualizado para o seu pai ({21}).

A implementação do método `findMinNode` é praticamente igual à implementação do método `min`. A única diferença é que, no método `min`, devolvemos apenas a chave, enquanto, no método `findMinNode`, devolvemos o nó.

O diagrama a seguir exemplifica a remoção de um nó com um filho à esquerda e outro à direita:

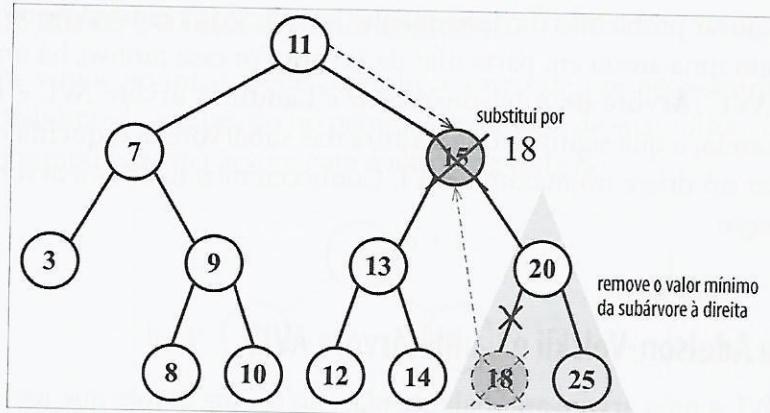


Figura 10.12

Árvores autobalanceadas

Agora que você já sabe como trabalhar com uma BST, poderá mergulhar no estudo das árvores, se quiser.

A BST tem um problema: conforme a quantidade de nós que você adicionar, uma das arestas da árvore poderá ser muito profunda, o que significa que um galho da árvore poderá ter um nível alto, enquanto outro galho poderá ter um nível baixo, como mostra o diagrama a seguir:

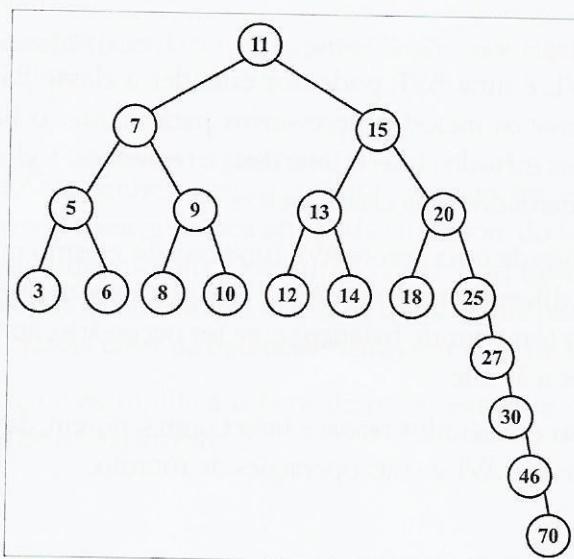


Figura 10.13