Centro Paula Souza

Fatec Jahu

ÁRVORES BINÁRIAS EM JAVASCRIPT

ESTRUTURA DE DADOS

CST em Desenvolvimento de Software Multiplataforma

PROF. Me. TIAGO A. SILVA

PARA SOBREVIVER AO JAVASCRIPT

Non-zero value

0

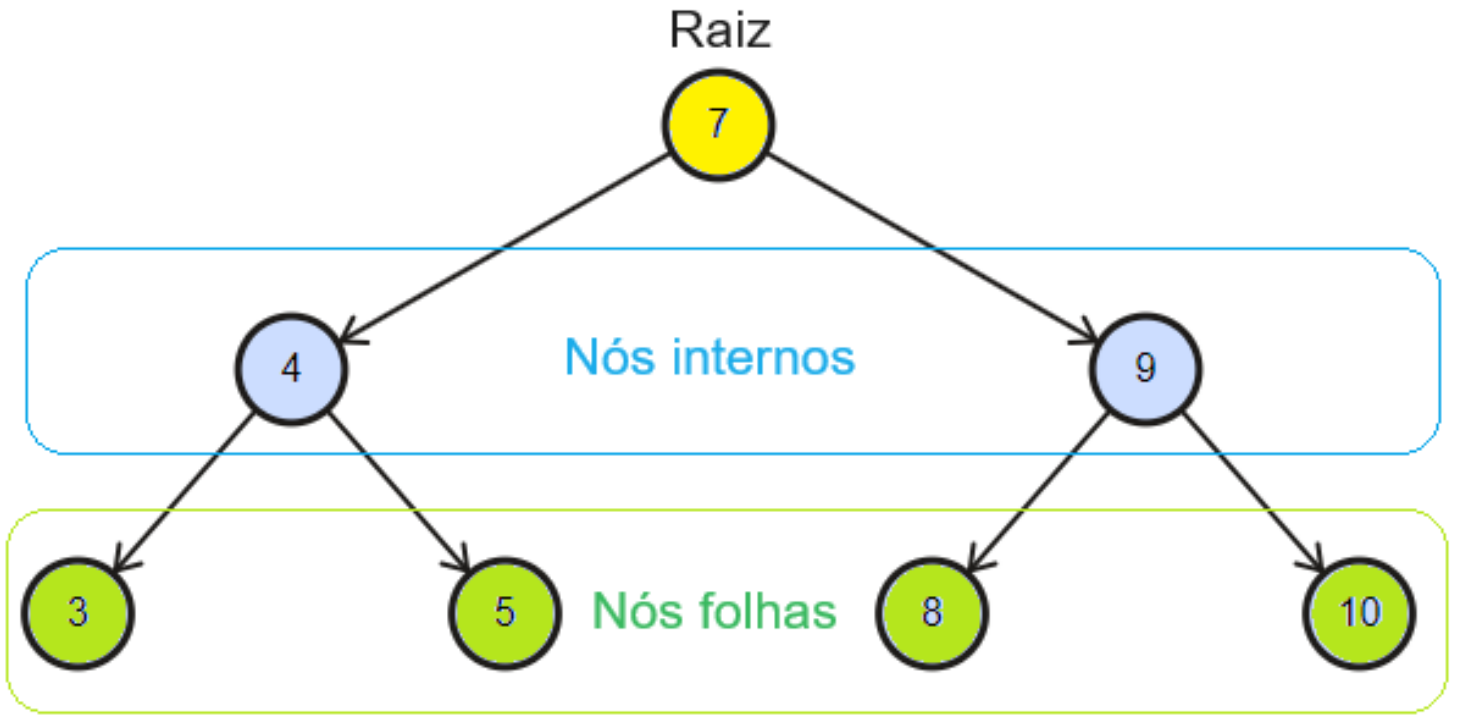
null

undefined

O QUE SÃO ÁRVORES BINÁRIAS?

- São uma estrutura de dados fundamental na ciência da computação e são amplamente usadas para representar dados de forma hierárquica e eficiente.
- Elas consistem em nós organizados de maneira que cada nó possui, no máximo, dois filhos: um à esquerda e outro à direita.
- Isso permite organizar e manipular dados de forma eficiente para diversas operações, como busca, inserção e remoção.
- No entanto, como qualquer estrutura de dados, as árvores binárias têm vantagens e desvantagens específicas que afetam seu desempenho e aplicabilidade em diferentes cenários.

O QUE SÃO ÁRVORES BINÁRIAS?



NÓS DA ÁRVORE BINÁRIA

- Definição da estrutura de um nó, contendo o valor do nó, ponteiros para o filho esquerdo e direito.

```
1  // Classe Node para representar um nó na árvore binária
2  class Node {
3      constructor(value) {
4          this.value = value; // Valor armazenado no nó
5          this.left = null;    // Referência para o nó filho à esquerda
6          this.right = null;   // Referência para o nó filho à direita
7      }
8  }
```

O QUE VAMOS IMPLEMENTAR?

```
10  // Classe BinaryTree para representar a árvore binária
11  class BinaryTree {
12
13      constructor() {
14          this.root = null; // Inicialmente, a árvore está vazia, então a raiz é null
15      }
16
17      // Método para inserir um valor na árvore
18      insert(value) {···
27      }
28
29      // Método auxiliar para encontrar a posição correta e inserir o nó na árvore
30      _insertNode(node, newNode) {···
50      }
51
52      // Percurso em-ordem: visita a subárvore esquerda, o nó atual e a subárvore direita
53      inOrder(node = this.root) {···
59      }
60
61      // Percurso pré-ordem: visita o nó atual, a subárvore esquerda e a subárvore direita
62      preOrder(node = this.root) {···
68      }
69
70      // Percurso pós-ordem: visita a subárvore esquerda, a subárvore direita e o nó atual
71      postOrder(node = this.root) {···
77      }
78
79      // Método para buscar um valor na árvore
80      search(value) {···
82      }
83
84      // Método auxiliar para realizar a busca recursivamente
85      _searchNode(node, value) {···
100     }
101
102     // Método para remover um nó com o valor especificado
103     remove(value) {···
105     }
```

CONSTRUTOR DA CLASSE BYNARYTREE

```
10  // Classe BinaryTree para representar a árvore binária
11  class BinaryTree {
12
13      constructor() {
14          this.root = null; // Inicialmente, a árvore está vazia, então a raiz é null
15      }
```

- É responsável por adicionar um novo nó à árvore de forma que a propriedade de ordenação seja mantida: todos os valores menores que um nó devem estar na sua subárvore esquerda, e todos os valores maiores, na sua subárvore direita.

```
17      // Método para inserir um valor na árvore
18      insert(value) {
19          const newNode = new Node(value); // Cria um novo nó com o valor dado
20          if (this.root === null) {
21              // Se a árvore estiver vazia, o novo nó se torna a raiz
22              this.root = newNode;
23          } else {
24              // Caso contrário, insere o nó na posição correta
25              this._insertNode(this.root, newNode);
26          }
27      }
```

INSERINDO NÓS NA ÁRVORE BINÁRIA

```
29      // Método auxiliar para encontrar a posição correta e inserir o nó na árvore
30      _insertNode(node, newNode) {
31          if (newNode.value < node.value) {
32              // Se o valor do novo nó for menor que o valor do nó atual, vá para a subárvore esquerda
33              if (node.left === null) {
34                  // Se não houver nó à esquerda, insere o novo nó aqui
35                  node.left = newNode;
36              } else {
37                  // Caso contrário, chama o método recursivamente na subárvore esquerda
38                  this._insertNode(node.left, newNode);
39              }
40          } else {
41              // Se o valor do novo nó for maior ou igual ao valor do nó atual, vá para a subárvore direita
42              if (node.right === null) {
43                  // Se não houver nó à direita, insere o novo nó aqui
44                  node.right = newNode;
45              } else {
46                  // Caso contrário, chama o método recursivamente na subárvore direita
47                  this._insertNode(node.right, newNode);
48              }
49          }
50      }
```

PERCURSO DA ÁRVORE EM ORDEM

- O objetivo é visitar cada nó da árvore em uma sequência ordenada (do menor para o maior valor, em uma árvore binária de busca). O percurso in-order é especialmente útil para obter os elementos da árvore em ordem crescente.

```
52      // Percurso em-ordem: visita a subárvore esquerda, o nó atual e a subárvore direita
53      inOrder(node = this.root) {
54          if (node !== null) {
55              this.inOrder(node.left); // Visita a subárvore esquerda
56              console.log(node.value); // Visita o nó atual
57              this.inOrder(node.right); // Visita a subárvore direita
58          }
59      }
```

PERCURSO DA ÁRVORE EM PRÉ ORDEM

- Em um percurso pre-order, os nós da árvore são visitados na seguinte ordem: raiz, subárvore esquerda e subárvore direita. Este método é útil em cenários onde é necessário processar ou manipular cada nó antes de seus filhos, como na clonagem de árvores ou avaliação de expressões em árvores de sintaxe.

```
61      // Percurso pré-ordem: visita o nó atual, a subárvore esquerda e a subárvore direita
62      preOrder(node = this.root) {
63          if (node !== null) {
64              console.log(node.value); // Visita o nó atual
65              this.preOrder(node.left); // Visita a subárvore esquerda
66              this.preOrder(node.right); // Visita a subárvore direita
67          }
68      }
```

PERCURSO DA ÁRVORE EM PÓS ORDEM

- Os nós são visitados na seguinte ordem: subárvore esquerda, subárvore direita e nó atual (raiz). Esse percurso é útil em casos onde precisamos processar primeiro todos os nós filhos antes de visitar o nó pai, como em algoritmos de remoção de uma árvore ou avaliação de expressões matemáticas.

```
70      // Percurso pós-ordem: visita a subárvore esquerda, a subárvore direita e o nó atual
71      postOrder(node = this.root) {
72          if (node !== null) {
73              this.postOrder(node.left); // Visita a subárvore esquerda
74              this.postOrder(node.right); // Visita a subárvore direita
75              console.log(node.value); // Visita o nó atual
76          }
77      }
```

BUSCA DA ÁRVORE BINÁRIA

- Este método explora a estrutura da árvore binária para encontrar o valor desejado de forma eficiente, utilizando a propriedade da árvore binária de busca:
  - Para cada nó, todos os valores menores estão à sua esquerda, e todos os valores maiores estão à sua direita.
  - Essa propriedade permite que o método de busca seja rápido, especialmente em árvores balanceadas.

```
    // Método para buscar um valor na árvore
    search(value) {
        return this._searchNode(this.root, value); // Inicia a busca a partir da raiz
    }

    // Método auxiliar para realizar a busca recursivamente
    _searchNode(node, value) {
        if (node === null) {
            // Se o nó atual é null, o valor não está na árvore
            return false;
        }
        if (value === node.value) {
            // Se o valor é encontrado, retorna true
            return true;
        } else if (value < node.value) {
            // Se o valor procurado é menor, continua a busca na subárvore esquerda
            return this._searchNode(node.left, value);
        } else {
            // Se o valor procurado é maior, continua a busca na subárvore direita
            return this._searchNode(node.right, value);
        }
    }
```

REMOVENDO ITENS DA ÁRVORE BINÁRIA

- O método remove em uma árvore binária de busca é responsável por remover um nó com um valor específico da árvore, mantendo a estrutura e propriedades da árvore binária de busca.
- Este processo é mais complexo do que a inserção ou a busca, pois envolve diversos casos, dependendo da quantidade de filhos que o nó a ser removido possui.

```
102     // Método para remover um nó com o valor especificado
103     remove(value) {
104         this.root = this._removeNode(this.root, value); // Inicia a remoção a partir da raiz
105     }
```

```
107     // Método auxiliar para remover o nó recursivamente
108     _removeNode(node, value) {
109         if (node === null) {
110             return null; // Se o nó é null, não há nada para remover
111         }
112 
113         if (value < node.value) {
114             // Se o valor a ser removido é menor, continua na subárvore esquerda
115             node.left = this._removeNode(node.left, value);
116             return node;
117         } else if (value > node.value) {
118             // Se o valor a ser removido é maior, continua na subárvore direita
119             node.right = this._removeNode(node.right, value);
120             return node;
121         } else {
122             // Se o valor é igual ao nó atual, este é o nó a ser removido
123 
124             // Caso 1: Nó sem filhos (nó folha)
125             if (node.left === null && node.right === null) {
126                 node = null; // Remove o nó ao definir como null
127                 return node;
128             }
```

```
129
130             // Caso 2: Nó com um filho
131             if (node.left === null) {
132                 node = node.right;
133                 return node;
134             } else if (node.right === null) {
135                 node = node.left;
136                 return node;
137             }
138
139             // Caso 3: Nó com dois filhos
140             // Encontra o nó com o menor valor na subárvore direita
141             const aux = this._findMinNode(node.right);
142             node.value = aux.value; // Substitui o valor do nó atual pelo valor mínimo encontrado
143             node.right = this._removeNode(node.right, aux.value); // Remove o nó duplicado na subárvore direita
144             return node;
145         }
146     }
```

EXEMPLO DE USO

```
157  // Exemplo de uso da árvore binária
158  const tree = new BinaryTree();
159  tree.insert(15);
160  tree.insert(10);
161  tree.insert(20);
162  tree.insert(8);
163  tree.insert(12);
164  tree.insert(18);
165  tree.insert(25);
166
167  console.log("Percurso em-ordem:");
168  tree.inOrder();
169
170  console.log("Buscar valor 18:");
171  console.log(tree.search(18) ? "Encontrado" : "Não encontrado");
172
173  console.log("Remover valor 10:");
174  tree.remove(10);
175  tree.inOrder();
```

CONCLUSÃO

- As árvores binárias são uma estrutura de dados poderosa e versátil, amplamente aplicável em diversos problemas de computação, **especialmente onde o tempo de acesso e busca eficiente são essenciais**.
- No entanto, seu **desempenho depende do balanceamento** e da natureza dos dados, além de exigir maior complexidade de implementação.
- Para superar essas desvantagens, **o uso de árvores binárias balanceadas**, fornece o equilíbrio necessário, embora com um custo adicional em termos de complexidade de código e manutenção.

EXERCÍCIO

1) Considere todas as estruturas de dados estudadas até hoje e faça um bechmark para avaliar o desempenho delas. Abaixo estão as instruções de metodologia:
   a) Cada estrutura deverá ter 10 mil elementos
   b) Teste a velocidade de inserção
   c) Teste a velocidade de listagem
   d) Teste a velocidade de remoção
   e) Organize os dados em uma planilha e apresente suas conclusões de qual melhor caso de uso para cada estrutura de dados

OBRIGADO!

- Encontre este **material on-line** em:
  - www.tiago.blog.br
  - Plataforma Teams
- Em caso de **dúvidas**, entre em contato:
  - **Prof. Tiago**: tiago.silva238@fatec.sp.gov.br