Geeks México

BLOG DE PROGRAMACIÓN EN ESPAÑOL SOBRE JAVA, FRAMEWORKS, BASES DE DATOS, CÓMPUTO EN LA NUBE, ETC. EN ESPAÑOL Y EN INGLÉS.

Borrar elementos de un

Arbol binario

HACE 20 MINS DEJA UN COMENTARIO

En el post Árboles binarios en Java (https://geeks-mexico.com/2017/11/15/arboles-binarios-en-java/) se habló sobre como crear un árbol binario en Java y se analizaron las siguientes operaciones:

- Agregar un nuevo nodo
- Buscar un nodo
- Imprimir in order
- Imprimir in pos order
- Imprimir in pre order

En este post se explicará como implementar el borrado de un nodo en el árbol.

Posibles escenarios

Cuando queremos borrar un elemento de un árbol binario, nos daremos cuenta que hay 3 posibles escenarios:

- El nodo no tiene nodos hijos
- El nodo tiene 1 hijo
- El nodo tiene 2 hijos

A continuación se presentará como borrar el nodo con Java para cada uno de ellos.

El nodo no tiene nodos hijos

Si el nodo que se desea borrar no tiene ningún nodo hijo, el borrado será muy simple ya que lo único que se debe hacer es asignar el valor de null a la referencia padre que apunta a el.

El borrado para este escenario tendrá una complejidad de O(Log N) por la búsqueda + O(Log 1) del borrado lo cual da una complejidad total de **O(Log N)**.

El nodo tiene un hijo

Si el nodo que se desea borrar tiene un nodo hijo el procedimiento será el siguiente:

- Buscar el elemento que deseamos borrar
- Apuntar el puntero padre que apunta a el a su nodo hijo

El borrado para este escenario tendrá una complejidad de O(Log N) por la búsqueda + O(1) de la actualización de referencias = **O(Log N)**.

El nodo tiene 2 hijos

Este es el escenario más complejo y debemos seguir lo siguiente:

1. Buscar el elemento
2. Para el siguiente paso tenemos dos opciones
    - Buscar el elemento más grande en el sub árbol de la izquierda (Que será conocido como **predecesor**).
    - Buscar el elemento más pequeño en el sub árbol de la derecha (Que será conocido como **sucesor**)
3. Intercambiar el predecesor o sucesor con el nodo que se desea borrar, antes de hacerlo se deben

validar los siguientes dos escenarios

1. Si el predecesor o sucesor no tienen nodos hijos: El elemento se intercambiará con el predecesor o sucesor y será borrado
2. Si el predecesor o sucesor tiene un nodo hijo: El elemento se intercambiará con el predecesor o sucesor, será borrado y su hijo ahora será hijo del predecesor o sucesor.

El borrado para este escenario tendrá una complejidad de O(Log N).

Analizando el código

Una vez que entendemos lo que se debe realizar, el siguiente paso será programarlo y analizar paso a paso el código:

- Programando la búsqueda del predecessor:

```
1  public Node findPredecessor() {
2      if (this.getRight() == null) {
3          return this;
4      } else {
5          return this.getRight().findPredeces
6      }
7  }
```

- Programando la búsqueda del sucessor:

```
1  public Node findSuccessor() {
2      if (this.getLeft() == null) {
3          return this;
4      } else {
5          return this.getLeft().findSuccessor
6      }
7  }
```

- Programando el borrado:

```
1   public Node delete(Integer value) {
2       Node response = this;
3       if (value < this.value) {        this.
4           this.right = this.right.delete(val
```

```
5       } else {
6           if (this.left != null && this.righ
7               Node temp = this;
8               Node maxOfTheLeft = this.left.
9               this.value = maxOfTheLeft.getV
10              temp.left=temp.left.delete(max
11          } else if (this.left != null) {
12              response = this.left;
13          } else if (this.right != null) {
14              response = this.right;
15          } else {
16              response = null;
17          }
18      }
19      return response;
20  }
```

Analizando método por método:

- *findPredecessor : Busca el nodo más grande de la rama*
- *findSuccessor : Busca el nodo más pequeño de la rama*
- *delete: Busca en el árbol y cuando encuentra el elemento intercambia el predecesor de la izquierda por el elemento a borrar*

Todo el código

A continuación se muestra como se ve la clase Node completa:

```
1   import java.util.Optional;
2
3   public class Node {
4       private Integer value;
5       private Node left;
6       private Node right;
7
8       public Node(Integer value) {
9           this.value = value;
10      }
11
12      public Integer getValue() {
13          return value;
14      }
15
16      public void setValue(Integer value) {
```

```
17          this.value = value;
18      }
19
20      public Node getLeft() {
21          return left;
22      }
23
24      public void setLeft(Node left) {
25          this.left = left;
26      }
27
28      public Node getRight() {
29          return right;
30      }
31
32      public void setRight(Node right) {
33          this.right = right;
34      }
35
36      public void add(Integer value) {
37          if (value < this.value) {
38              if (left != null) {
39                  left.add(value);
40              } else {
41                  left = new Node(value);
42              }
43          } else {
44              if (right != null) {
45                  right.add(value);
46              } else {
47                  right = new Node(value);
48              }
49          }
50      }
51
52      public Optional<Node> find(Integer va
53          if (value == this.value) {
54              return Optional.of(this);
55          } else if (value < this.value) {
56              if (this.left != null) {
57                  return this.left.find(va
58              } else {
59                  return Optional.empty();
60              }
61          } else {
62              if (this.right != null) {
63                  return this.right.find(va
64              } else {
65                  return Optional.empty();
66              }
67          }
68      }
69
70      public void printInOrder() {
71          if (left != null) {
72              left.printInOrder();
73          }
74          System.out.println(value);
75          if (right != null) {
76              right.printInOrder();
77          }
78      }
```

```
79
80      public void printPreOrder() {
81          System.out.println(value);
82          if (left != null) {
83              left.printPreOrder();
84          }
85          if (right != null) {
86              right.printPreOrder();
87          }
88      }
89
90      public void printPosOrder() {
91          if (left != null) {
92              left.printPreOrder();
93          }
94          if (right != null) {
95              right.printPreOrder();
96          }
97          System.out.println(value);
98      }
99
100     public Node findPredecessor() {
101         if (this.getRight() == null) {
102             return this;
103         } else {
104             return this.getRight().findPr
105         }
106     }
107
108     public Node findSuccessor() {
109         if (this.getLeft() == null) {
110             return this;
111         } else {
112             return this.getLeft().findSuc
113         }
114     }
115
116     public Node delete(Integer value) {
117         Node response = this;
118         if (value < this.value) {
119             this.right = this.right.delet
120         } else {
121             if (this.left != null && this
122                 Node temp = this;
123                 Node maxOfTheLeft = this.
124                 this.value = maxOfTheLeft
125                 temp.left = temp.left.del
126             } else if (this.left != null)
127                 response = this.left;
128             } else if (this.right != null
129                 response = this.right;
130             } else {
131                 response = null;
132             }
133         }
134         return response;
135     }
136
137     @Override
138     public String toString() {
139         return "Node [value=" + value + "
140     }
```

```
141
142  }
```

Probando el código

El último paso es probar todo el código y ver que el borrado funciona, para esto ejecutaremos el siguiente código:

```
1  public class TreeTest {
2      public static void main(String[] args)
3          Node root = new Node(32);
4          root.add(10);
5          root.add(55);
6          root.add(1);
7          root.add(19);
8          root.add(16);
9          root.add(23);
10         root.add(79);
11         System.out.println("Tree before de
12         root.printPreOrder();
13         root.delete(55);
14         root.delete(16);
15         root.delete(32);
16         System.out.println("Tree after del
17         root.printPreOrder();
18
19     }
20 }<span                data-mce-type="bookma
```

Como se puede ver es posible borrar nodos en los 3 escenarios mencionados anteriormente.

Si te gusta el contenido y quieres enterarte cuando realicemos un post nuevo síguenos en nuestras redes sociales https://twitter.com/geeks_mx (https://twitter.com/geeks_mx) y https://www.facebook.com/geeksJavaMexico/ (https://www.facebook.com/geeksJavaMexico/).

Autor: Alejandro Agapito Bautista

Twitter: @raidentrance

Contacto:raidentrance@gmail.com