Binary Search Tree

Santiago Toll Leyva
Universidad de Artes Digitales

Guadalajara, Jalisco

Email: idv16a.stoll@uartesdigitales.edu.mx

Profesor: Efraín Padilla

Julio 11, 2019

Arboles de Busqueda Binaria

Teoria

Los arboles de busqueda binaria son una estructura dinamica que ordena los datos ingresados de tal manera que en cada iteracion de busqueda se elimina la mitad de los valores con una simple condicion.
En los arboles de busqueda se generan nodos; estos nodos contienen su propia informacion y conocen a sus ”hijos”.

Planteamiento del problema

Realizar una busqueda en una gran tupla de elementos puede llegar a ser muy costoso por lo que se propone utilizar un BST para reducir el tiempo de busqueda de un elemento.

Solucion del problema

Cuando un elemento se repite no se crea otro nodo, sino que se aumenta un contador interno que tiene la informacion del nodo. Esto se realiza de esta manera para que no se cree ninguna especie de conflicto cuando se quiere generar un elemento con un valor ya existente en el arbol.

El funcionamiento de este algoritmo se prueba relaizando inserciones, busqueda y eliminacion de nados. En este caso se insertaran, buscaran y eliminaran valores para probar distintos casos como la insercion de un mismo valor dos veces, busqueda de un valor que no se encuentra en el arbol y eliminacion de un valor repetido.

Codigo

Funciones utilizadas para generar vectores.

Listing 1. Vector Generation Functions C++

```cpp
struct Node {
  //Value this node contains.
  int key;
  //Amount of times this value repeats.
  int count;
  //Child Nodes
  Node * leftChild;
  Node * rightChild;

  Node *  Init(int _key) {
    this->key = _key;
    this->count = 1;
    leftChild = nullptr;
    rightChild = nullptr;

    return this;
  }
};

struct BinaryTree {

private:
  void Inorder(Node * node) {
    if (root == nullptr) {
      std::cout << "Root is NULL" << std::endl;
    }
    if (node != nullptr) {
      Inorder(node->leftChild);
      for (int i = 0; i < node->count; i++) {
        std::cout << node->key << std::endl;
      }
      Inorder(node->rightChild);
    }

  }
  Node * AddNode(int key) {
    Node * temp = new Node();
    temp->Init(key);
    if (root == nullptr) {
      root = temp;
    }
    return temp;
  }
  Node * Search(Node* node, int key) {
    if (node == nullptr) {
      std::cout << "Key with value " << key << " was not found" << std::endl;
      return node;
    }
    if (key == node->key) {
      std::cout << "The key with value " << key << " was found " << node->count << " times in
      return node;
    }
    if (node->key < key) {
```

```
54        return Search(node->leftChild, key);
55      }
56      else {
57        return Search(node->rightChild, key);
58      }
59    }
60    Node * Insert(Node* node, int key) {
61      if (node == nullptr) {
62        return AddNode(key);
63      }
64      //If key is lower.
65      if (key < node->key) {
66        node->leftChild = Insert(node->leftChild, key);
67      }
68      //If key is higher.
69      else if (key > node->key) {
70        node->rightChild = Insert(node->rightChild, key);
71      }
72      //If key is the same.
73      else {
74        node->count++;
75      }
76      return node;
77    }
78    Node * Delete(struct Node* node, int key)
79    {
80      if (node == NULL) {
81        return node;
82      }
83
84      if (key < node->key) {
85        node->leftChild = Delete(node->leftChild, key);
86      }
87
88      else if (key > node->key) {
89        root->rightChild = Delete(node->rightChild, key);
90      }
91
92      else
93      {
94        if (root->count > 1)
95        {
96          node->count--;
97          return node;
98        }
99
100       if (node->leftChild == NULL)
101       {
102         Node * temp = node->rightChild;
103         free(node);
104         return temp;
105       }
106       else if (node->rightChild == NULL)
107       {
108         Node * temp = node->leftChild;
109         free(node);
110         return temp;
111       }
```

```
112
113         Node * current = node->rightChild;
114         while (current->leftChild != NULL) {
115           current = current->leftChild;
116         }
117         Node * temp = current;
118         node->key = temp->key;
119         node->rightChild = Delete(node->rightChild, temp->key);
120
121       }
122       return node;
123     }
124
125   public:
126     void Inorder() {
127       Inorder(root);
128     }
129     Node * Search(int key) {
130       return Search(root, key);
131     }
132     Node * Insert(int key) {
133       return Insert(root, key);
134     }
135     Node * Delete(int key) {
136       return Delete(root, key);
137     }
138
139     Node * root = nullptr;
140   };
141
142   int main()
143   {
144     //Create BinaryTree.
145     BinaryTree binaryTree;
146     //Insert a bunch of numbers in the tree.
147     binaryTree.Insert(10);
148     binaryTree.Insert(15);
149     binaryTree.Insert(5);
150     binaryTree.Insert(0);
151     binaryTree.Insert(10);
152
153     //Print elements in Tree.
154     binaryTree.Inorder();
155
156     //Search for existing value.
157     binaryTree.Search(10);
158     //Search for unexisting value.
159     binaryTree.Search(20);
160     std::cout << std::endl;
161
162     binaryTree.Delete(10);
163     binaryTree.Inorder();
164     std::cout << std::endl;
165     binaryTree.Delete(10);
166     binaryTree.Inorder();
167
168   }
```

Results

En la siguiente imagen se puede observar que al eliminar valores repetidos en el arbol solamente se elimina uno a la vez y cuando solo queda uno y es eliminado se lleva a cabo el proceso de modificacion del arbol. En el caso de la busqueda esta regresa la cantidad de veces que se encuentra el nodo dentro del arbol.