
UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

# Escuela Profesional de Ciencia de la Computación

## Estructura de datos avanzados

---

## Lab 2: Árboles - Red-Black Tree

---

***Alumnos:***
*Alejandro Antonio Villa Herrera*
*Samuel Felipe Chambi Ytusaca*
*Julio Enrique Yauri Ituccayasi*
***Profesor:***
*Rolando Jesús Cárdenas Talavera*

17 de octubre de 2022

# Índice

**1. Actividad** **2**

**2. Resultados** **3**
2.1. Clase Node . . . . . 4
2.2. Clase RB_Tree . . . . . 5
2.3. Main . . . . . 11

# 1. Actividad

- Implemente el árbol Red-Black. Ejecute el algoritmo varias veces con datos desde 10 a 10 000 y mida el tiempo medio de accesos partiendo desde la raíz hasta un nodo aleatorio. Tendrá que obtener un gráfico similar al que se muestra en Fig 1
- Analizar la complejidad computacional.

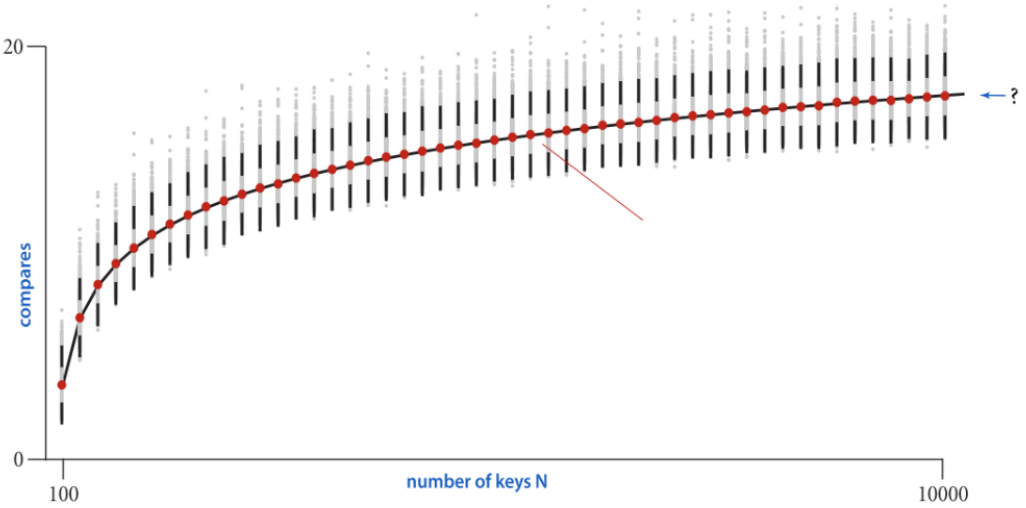


Figure 2: Número de operaciones por cantidad de datos

Figura 1: Número de operaciones por cantidad de datos

## 2. Resultados

Se realizaron dos pruebas, la primera insertando elementos al árbol desde 100 hasta 10000 en un intervalo de 100 en 100 y en la segunda en intervalos de 10 en 10, para luego, en cada iteración buscar un número aleatorio, el resultado de la primera prueba se ve en la gráfica de la figura 2. Para la segunda prueba el resultado en la gráfica en la figura 3. Entonces ambos gráficos muestran un comportamiento similar al que se tiene en la figura 1.

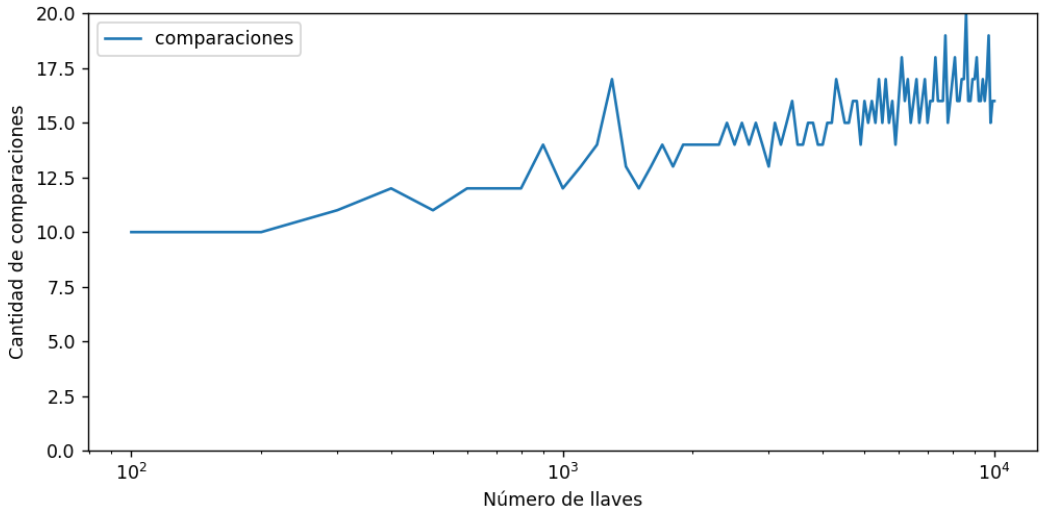


Figura 2: Operaciones realizada con una cantidad de datos ascendente de 100 en 100

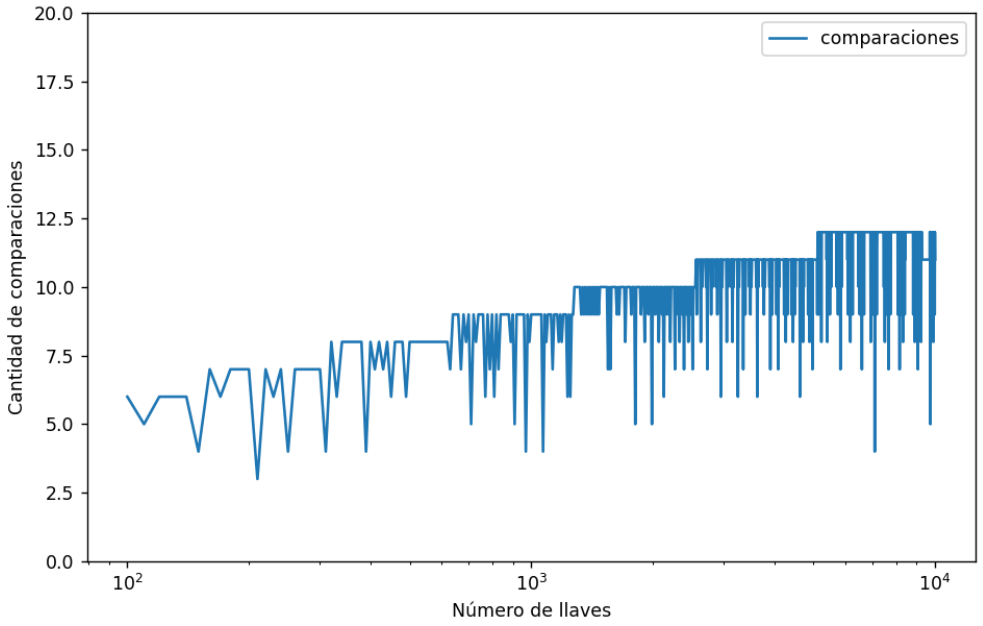


Figura 3: Operaciones realizada con una cantidad de datos ascendente de 10 en 10

Ahora se muestran la parte del main donde se insertan los elementos, y se realizan las búsquedas, para luego guardarlas en un archivo txt para su posterior gráfica, figura 4. Finalmente en la figura 5 se muestra la complejidad del algoritmo, un contador que por cada comparación hecha aumenta en 1, como se usa un algoritmo simple de búsqueda la complejidad es $O(log(n))$

```
RB_Tree<int> RB;
vector<pair<int,int>> res;
int n = pow(10,2);
for( ; n<=10000 ; n+=10)
{
    srand(time(NULL));
    for(int j=1 ; j<=n ; ++j)
        RB.insert(j);
    int num = 1 + rand() % (n/10);
    int comp = RB.search(num);
    res.push_back(make_pair(n,comp));
    RB.clear();
}
guardar(res,"data.txt");
```

Figura 4: Main del programa

```
template<typename T>
int RB_Tree<T>::search(T v)  // log(n)
{
    Node<T>* tmp = root;
    int c=0;
    while(tmp)
    {
        c++;
        if(tmp->value == v)
            return c;
        if(v < tmp->value)
            tmp = tmp->pSon[0];
        else
            tmp = tmp->pSon[1];
    }
    return -1;
}
```

Figura 5: Algoritmo usado para calcular la cantidad de comparaciones hechas al buscar un elemento

A continuación se muestra la implementación del arbol Red-Black, para esto se crearon dos clases, la primera *Node.h* que hace la abstracción de los nodos en el árbol y la clase *RB_Tree.h* que representa al árbol Red-Black y sus operaciones.

## 2.1. Clase Node

La clase Node es una representación de un nodo en un árbol, este guarda un valor (que puede ser cualquier tipo gracias al uso de plantillas), un arreglo de punteros

(hijo izquierdo, hijo derecho y padre) y un booleano que representa al color (0 negro, 1 rojo).

```
#ifndef NODE
#define NODE

template<typename T>
class Node
{
public:
    T value;
    Node<T>* pSon[3];        // izq, der, padre
    bool col;                // color 0 negro, 1 rojo
    Node(T v);
    void autoMatate(Node<T>* N);
};

template<typename T>
Node<T>::Node(T v)
{
    value = v;
    pSon[0] = pSon[1] = pSon[2] = 0;
    col = 0;
}

template<typename T>
void Node<T>::autoMatate(Node<T>* N)
{
    if(!N) return;
    autoMatate(N->pSon[0]);
    autoMatate(N->pSon[1]);
    delete N;
}

#endif
```

## 2.2. Clase RB_Tree

La clase RB_Tree representa al árbol binario Red-Black, con sus respectivas implementaciones (rotaciones, insert y insert fixup)

```
#ifndef RB_TREE
#define RB_TREE

#include <fstream>
```

```
#include <iostream>
#include "node.h"

template<typename T>
class RB_Tree
{
private:
    Node<T>* root;
    int size;

    void inOrder(Node<T>*& N);
    void preOrder(Node<T>*& N);
    void postOrder(Node<T>*& N);
    void graficar(Node<T>*& N, std::ofstream& f);
    void leftRotation(Node<T>* N);
    void rightRotation(Node<T>* N);
public:
    RB_Tree():root(nullptr),size(0){}
    ~RB_Tree();
    void insert(T v);
    void insertFixup(Node<T>*& N);
    void inOrder();
    void postOrder();
    void preOrder();
    int search(T v);                    // devuelve la cantidad de
   comparaciones hasta encontrar
    void clear();
    int getSize(){ return size; }
    void graficar(std::string dir);
};

template<typename T>
RB_Tree<T>::~RB_Tree()
{
    root->autoMatate(root);     // recursivo
}

template<typename T>
void RB_Tree<T>::inOrder()
{
    inOrder( root );
    std::cout<<'\n';
}

template<typename T>
```

```
49 void RB_Tree<T>::inOrder( Node<T>*& N )
50 {
51     if( !N ) return;
52     inOrder( N->pSon[0] );
53     std::cout<<N->value<<' ';
54     inOrder( N->pSon[1] );
55 }
56 
57 template<typename T>
58 void RB_Tree<T>::preOrder()
59 {
60     preOrder( root );
61     std::cout<<'\n';
62 }
63 
64 template<typename T>
65 void RB_Tree<T>::preOrder( Node<T>*& N )
66 {
67     if( !N ) return;
68     std::cout<<N->value<<' ';
69     preOrder( N->pSon[0] );
70     preOrder( N->pSon[1] );
71 }
72 
73 template<typename T>
74 void RB_Tree<T>::postOrder()
75 {
76     postOrder( root );
77     std::cout<<'\n';
78 }
79 
80 template<typename T>
81 void RB_Tree<T>::postOrder( Node<T>*& N )
82 {
83     if( !N ) return;
84     postOrder( N->pSon[0] );
85     postOrder( N->pSon[1] );
86     std::cout<<N->value<<' ';
87 }
88 
89 template<typename T>
90 void RB_Tree<T>::leftRotation(Node<T>* N)
91 {
92     Node<T>* y = N -> pSon[1];
93     N -> pSon[1] = y -> pSon[0];
```

```
94
95      if(y -> pSon[0]) (y -> pSon[0]) -> pSon[2] = N;
96      y -> pSon[2] = N -> pSon[2];
97
98      if(!(N -> pSon[2])) // si N es raiz
99          root = y;
100     else if(N == ((N -> pSon[2]) -> pSon[0])) // si es hijo izquierdo
101         (N -> pSon[2]) -> pSon[0] = y;
102     else                                      // si es hijo derecho
103         (N -> pSon[2]) -> pSon[1] = y;
104
105     y -> pSon[0] = N;
106     N -> pSon[2] = y;
107 }
108
109 template<typename T>
110 void RB_Tree<T>::rightRotation(Node<T>* N)
111 {
112     Node<T>* y = N -> pSon[0];
113     N -> pSon[0] = y -> pSon[1];
114
115     if(y -> pSon[1]) (y -> pSon[1]) -> pSon[2] = N;
116     y -> pSon[2] = N -> pSon[2];
117
118     if(!(N -> pSon[2])) // si N es raiz
119         root = y;
120     else if(N == ((N -> pSon[2]) -> pSon[0])) // si es hijo izquierdo
121         (N -> pSon[2]) -> pSon[0] = y;
122     else                                      // si es hijo derecho
123         (N -> pSon[2]) -> pSon[1] = y;
124
125     y -> pSon[1] = N;
126     N -> pSon[2] = y;
127 }
128
129 template<typename T>
130 void RB_Tree<T>::insert(T v)
131 {
132     Node<T>* z = new Node<T>(v);
133     Node<T>* y = nullptr;
134     Node<T>* x = root;
135     while(x)
136     {
137         y = x;
138         if(z->value < x->value)
```

```
139             x = x->pSon[0];
140         else
141             x = x->pSon[1];
142     }
143     z->pSon[2] = y;
144     if(!y)
145         root = z;
146     else if(z->value < y->value)
147         y->pSon[0] = z;
148     else
149         y->pSon[1] = z;
150     z->pSon[0] = nullptr;
151     z->pSon[1] = nullptr;
152     z->col = 1;
153     insertFixup(z);
154     size++;
155 }
156 
157 template <typename T>
158 void RB_Tree<T>::insertFixup(Node<T>* &a){
159     Node<T> *y; // Tio
160     while(a->pSon[2] && a->pSon[2]->col)
161     {
162         if(a->pSon[2]->pSon[2] && a->pSon[2] == a->pSon[2]->pSon[2]->
    pSon[0])
163         {
164             y = a->pSon[2]->pSon[2]->pSon[1]; //Tio derecho
165             if(y && y->col)          // existe tio y es rojo
166             {
167                 y->col = 0;
168                 a->pSon[2]->col = 0;
169                 a->pSon[2]->pSon[2]->col = 1;
170                 a = a->pSon[2]->pSon[2];
171             }
172             else
173             {
174                 if(a == a->pSon[2]->pSon[1])
175                 {
176                     a = a->pSon[2];
177                     leftRotation(a);
178                 }
179                 a->pSon[2]->col = 0;
180                 a->pSon[2]->pSon[2]->col = 1;
181                 rightRotation(a->pSon[2]->pSon[2]);
182             }
```

```
183         }
184         else if(a->pSon[2]->pSon[2] && a->pSon[2] == a->pSon[2]->pSon
    [2]->pSon[1])
185         {
186             y = a->pSon[2]->pSon[2]->pSon[0]; //Tio iszquierdo
187             if(y && y->col)
188             {
189                 y -> col = 0;
190                 a->pSon[2]->col = 0;
191                 a->pSon[2]->pSon[2]->col = 1;
192                 a = a->pSon[2]->pSon[2];
193             }
194             else
195             {
196                 if(a == a->pSon[2]->pSon[0])
197                 {
198                     a = a->pSon[2];
199                     rightRotation(a);
200                 }
201                 a->pSon[2]->col = 0;
202                 a->pSon[2]->pSon[2]->col = 1;
203                 leftRotation(a->pSon[2]->pSon[2]);
204             }
205         }
206         // if(a==root) break;
207     }
208     root->col = 0;
209 }
210
211 template<typename T>
212 int RB_Tree<T>::search(T v)
213 {
214     Node<T>* tmp = root;
215     int c=0;
216     while(tmp)
217     {
218         c++;
219         if(tmp->value == v)
220             return c;
221         if(v < tmp->value)
222             tmp = tmp->pSon[0];
223         else
224             tmp = tmp->pSon[1];
225     }
226     return -1;
```

```
227 }
228
229 template<typename T>
230 void RB_Tree<T>::clear()
231 {
232     root->autoMatate(root);
233     size = 0;
234     root = nullptr;
235 }
236
237 #endif
```

## 2.3. Main

La función main crea una instancia del árbol Red-Black, inserta elementos en el árbol y luego busca un número aleatorio. Guarda todas las búsquedas y la cantidad de comparaciones para luego ser graficadas.

```
1  #include <bits/stdc++.h>
2  #include "RB_tree.h"
3  using namespace std;
4
5  void guardar(vector<pair<int,int>>& V, string dir)
6  {
7      ofstream f(dir);
8      for(auto a : V)
9          f<<a.first<<' '<<a.second<<'\n';
10     f.close();
11 }
12
13 int main()
14 {
15     RB_Tree<int> RB;
16     vector<pair<int,int>> res;
17     int n = pow(10,2);
18     for( ; n<=10000 ; n+=10)
19     {
20         srand(time(NULL));
21         for(int j=1 ; j<=n ; ++j)
22             RB.insert(j);
23         int num = 1 + rand() % (n/10);
24         int comp = RB.search(num);
25         res.push_back(make_pair(n,comp));
26         RB.clear();
27     }
```

```
28      guardar(res,"data.txt");
29  }
```