Universidad Nacional de San Agustín de Arequipa

Escuela Profesional de Ciencia de la Computación

Estructura de Datos Avanzados

Laboratorio 2: Arboles - Red-Black Tree

Presentado por:
Rushell Vanessa Zavalaga Orozco

Docente :
Rolando Jesús Cárdenas Talavera

1. Actividades

Implemente el arbol 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.

1.1. Algoritmo Red Black Tree

Un **árbol rojo-negro (Red-Black Tree, RBT)** es una estructura de datos auto-balanceada que es un tipo particular de árbol binario de búsqueda. La característica clave de los árboles rojo-negro es que mantienen un equilibrio en su altura, lo que garantiza que las operaciones comunes, como la inserción, eliminación y búsqueda, se realicen en un tiempo logarítmico $O(\log n)$, donde n es el número de nodos en el árbol.

1.1.1. Propiedades de un árbol rojo-negro

Un árbol rojo-negro es un árbol binario de búsqueda que cumple con las siguientes cinco propiedades:

1. **Cada nodo es rojo o negro**: Esto es lo que le da el nombre al árbol. Los nodos tienen un atributo de color, que puede ser rojo o negro.
2. **La raíz siempre es negra**: La raíz del árbol siempre tiene el color negro. Esto es crucial para mantener el equilibrio del árbol.
3. **Todas las hojas (nodos nulos) son negras**: En los árboles rojo-negro, las hojas se representan por nodos nulos (también llamados nodos NIL), y estos nodos NIL se consideran negros. No se suelen mostrar explícitamente, pero se tienen en cuenta para las propiedades.
4. **Un nodo rojo no puede tener hijos rojos**: Esto se llama la "propiedad de no rojos consecutivos". Si un nodo es rojo, ambos hijos deben ser negros, lo que ayuda a mantener el equilibrio del árbol.
5. **Cualquier camino desde un nodo hasta sus hojas descendientes contiene el mismo número de nodos negros**: Esta propiedad garantiza que el árbol no se desequilibre demasiado, ya que obliga a que todos los caminos desde la raíz hasta una hoja tengan la misma cantidad de nodos negros. A esto se le llama la propiedad de la .ªltura negra".

1.1.2. Operaciones en un árbol rojo-negro

- **Búsqueda**: El árbol rojo-negro es un árbol binario de búsqueda, por lo que la búsqueda de un elemento se realiza de manera similar a la de cualquier otro árbol binario de búsqueda, en un tiempo $O(\log n)$, donde n es el número de nodos.

- **Inserción**: La inserción en un árbol rojo-negro sigue las reglas de inserción de un árbol binario de búsqueda, pero luego es necesario reequilibrar el árbol para asegurarse de que las propiedades de los nodos rojos y negros no se violen. Este proceso puede implicar rotaciones (rotaciones a la izquierda o derecha) y cambios de color.
- **Eliminación**: La eliminación en un árbol rojo-negro es más compleja que en otros árboles binarios de búsqueda, ya que también puede violar las propiedades del árbol rojo-negro. Después de eliminar un nodo, es necesario reequilibrar el árbol, lo que implica rotaciones y recoloreos.

2. Procedimiento:

Las pruebas se realizaron utilizando el siguiente enfoque:

- Se generaron n llaves aleatorias donde n varió desde 100 hasta 10,000 en incrementos de 100.
- Para cada valor de n, se insertaron las llaves en un árbol rojo-negro.
- Se realizaron 100 búsquedas aleatorias en el árbol para medir el tiempo promedio de búsqueda, repitiendo cada búsqueda 1,000 veces para obtener un promedio más preciso.
- Los tiempos de búsqueda se almacenaron en un archivo de texto para su posterior análisis.

3. Resultados

Los resultados obtenidos de las pruebas se registraron en el archivo `Times.txt`. A continuación, se presenta una tabla con algunos de los tiempos promedio de búsqueda para diferentes tamaños de n:

Número de llaves n	**Tiempo promedio de búsqueda (ns)**
100	26
150	35
200	28
250	34
10000	62

Cuadro 1: Tiempos promedio de búsqueda en función del número de llaves.

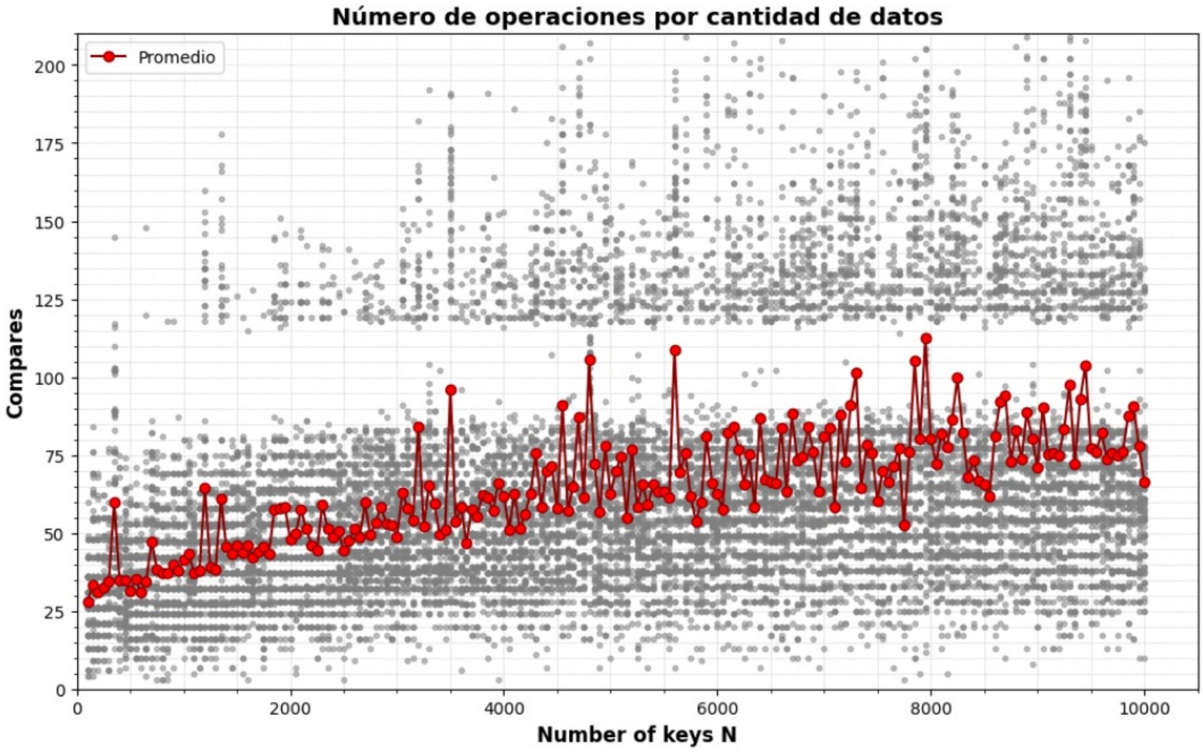

Figura 1: Número de operaciones por cantidad de datos.

4. Conclusiones

Los resultados indican que el tiempo promedio de búsqueda en el árbol rojo-negro aumenta con el número de llaves n, lo que es consistente con la teoría que establece que el tiempo de búsqueda es $O(\log n)$. Este comportamiento se debe a la naturaleza equilibrada de la estructura de datos, que permite un acceso eficiente incluso con un gran número de elementos.

5. Codigo

5.1. Main

```
#include <iostream>
#include <vector>
#include <fstream>
#include <chrono>
#include <random>
#include <cstdlib>
#include "R_RBT.cpp"

using namespace std::chrono;
using namespace std;
```

```
void guardar(int n, vector<double> &v){
    ofstream archivo("Times.txt",  ios::app);
    archivo << n << " ";
    for (const double &time : v) {
        archivo << time << " ";
    }
    archivo << endl;
    archivo.close();
}

int main(){
    srand(time(NULL));
    ofstream archivo;
    archivo.open("Times.txt",  ios::out);
    archivo.close();

    for(int n = 100 ; n <= 10000 ; n+=50){
        RedBlack_Tree<int> RBT;
        vector<int> numbersRAN;

        cout<<"\nAgregando>>\n";
        for(int i=0; i<n; i++){
            int random = rand() % (n*10);
            numbersRAN.push_back(random);
            RBT.Add(random);
        }

        RBT.dibujar();

        vector<double> times;
        int repetitions = 1000;
        for (int i = 0; i < 100; i++) {
            int search_value = numbersRAN[rand() % n];
            auto start_time = high_resolution_clock::now();
            for (int j = 0; j < repetitions; j++) {
                RBT.Find(search_value);
            }
            auto end_time = high_resolution_clock::now();
            auto time_spent_ns = duration_cast<nanoseconds>(end_time - start_time);

            times.push_back(time_spent_ns.count() / repetitions);
            cout<<" / T / "<<time_spent_ns.count() / repetitions;
        }


        cout<<"\n Promedio > "<<accumulate(times.begin(), times.end(), 0.0) / times.size() << " ns\n'
        guardar(n, times);
    }
}
```

5.2. Red Black Tree

```
#include<iostream>
#include <fstream>
using namespace std;
```

```
template<class T>
class Nodo{
    public:
       T m_Dato;
       Nodo<T> * m_pSon[3];
       int m_Color;

       Nodo(T d){
           m_Dato = d;
           // Hijo Izquierdo
           m_pSon[0] = 0;
           // Hijo Derecho
           m_pSon[1] = 0;
           // Padre
           m_pSon[2] = 0;
           // Color 0 = N ; 1 = R
           m_Color = 1;
       }
};

template<class T>
class RedBlack_Tree{
    private:
       Nodo<T> * m_pRoot;

    public:
        RedBlack_Tree(){
            m_pRoot = 0;
        }

        void RR(Nodo<T> *p){
            Nodo<T> * NodoTMP = new Nodo<T>(p->m_Dato);
            if(p->m_pSon[0]->m_pSon[1]) NodoTMP->m_pSon[0] = p->m_pSon[0]->m_pSon[1];
            NodoTMP->m_pSon[1] = p->m_pSon[1];
            NodoTMP->m_Color = p->m_Color;

            p->m_Dato = p->m_pSon[0]->m_Dato;
            p->m_Color = p->m_pSon[0]->m_Color;
            p->m_pSon[1] = NodoTMP;

            if(NodoTMP->m_pSon[0]) NodoTMP->m_pSon[0]->m_pSon[2] = NodoTMP;
            if(NodoTMP->m_pSon[1]) NodoTMP->m_pSon[1]->m_pSon[2] = NodoTMP;

            NodoTMP->m_pSon[2] = p;

            if(p->m_pSon[0]->m_pSon[0]) p->m_pSon[0] = p->m_pSon[0]->m_pSon[0];
            else p->m_pSon[0] = 0;

            if(p->m_pSon[0]) p->m_pSon[0]->m_pSon[2] = p;
        }

        void LR(Nodo<T> *p){
            Nodo<T> * NodoTMP = new Nodo<T>(p->m_Dato);
            if(p->m_pSon[1]->m_pSon[0]) NodoTMP->m_pSon[1] = p->m_pSon[1]->m_pSon[0];
            NodoTMP->m_pSon[0] = p->m_pSon[0];
            NodoTMP->m_Color = p->m_Color;
```

```
        p->m_Dato = p->m_pSon[1]->m_Dato;
        p->m_pSon[0] = NodoTMP;

        if(NodoTMP->m_pSon[0]) NodoTMP->m_pSon[0]->m_pSon[2] = NodoTMP;
        if(NodoTMP->m_pSon[1]) NodoTMP->m_pSon[1]->m_pSon[2] = NodoTMP;

        NodoTMP->m_pSon[2] = p;

        if(p->m_pSon[1]->m_pSon[1]) p->m_pSon[1] = p->m_pSon[1]->m_pSon[1];
        else p->m_pSon[1] = 0;

        if(p->m_pSon[1]) p->m_pSon[1]->m_pSon[2] = p;
    }

    bool Add(T D){
        return Add(D, m_pRoot, NULL);
    }

    bool Add(T D, Nodo<T> *&p, Nodo<T> *padre){
        if (!p){ p = new Nodo<T>(D); p->m_pSon[2] = padre; FixInsert(p); return true; }
        if (D == p->m_Dato) return false;
        return Add(D, p->m_pSon[p->m_Dato < D], p);
    }

    void FixInsert(Nodo<T> *p){
        if(!p->m_pSon[2]) {m_pRoot->m_Color = 0; return; }
        while (p->m_pSon[2]->m_Color == 1){
            Nodo<T> *abuelo = p->m_pSon[2]->m_pSon[2];
            Nodo<T> *tio = m_pRoot;
            if (p->m_pSon[2] == abuelo->m_pSon[0]){
                if (abuelo->m_pSon[1]) tio = abuelo->m_pSon[1];
                if (tio->m_Color == 1){ // CASO 1: PAPA Y TIO SON ROJOS
                    // CAMBIA DE COLOR
                    p->m_pSon[2]->m_Color = 0; // PAPA
                    tio->m_Color = 0; // TIO
                    abuelo->m_Color = 1; // ABUELO
                    if (abuelo->m_Dato != m_pRoot->m_Dato) p = abuelo;
                    else break;
                } // CASO 2: PAPA ES ROJO, TIO NEGRO Y ADEMAS P ES HIJO DERECHO Y PADRE DE P ES E
                else if (p == abuelo->m_pSon[0]->m_pSon[1]) LR(p->m_pSon[2]);
                else { // CASO 3: PAPA ES ROJO, TIO NEGRO Y P ES HIJO IZQUIERDO
                    p->m_pSon[2]->m_Color = 0;
                    abuelo->m_Color = 1;
                    RR(abuelo);
                    if (abuelo->m_Dato != m_pRoot->m_Dato) p = abuelo;
                    else break;
                }
            } else {
                if (abuelo->m_pSon[0]) tio = abuelo->m_pSon[0];
                if (tio->m_Color == 1) {
                    p->m_pSon[2]->m_Color = 0;
                    tio->m_Color = 0;
                    abuelo->m_Color = 1;
                    if (abuelo->m_Dato != m_pRoot->m_Dato) p = abuelo;
                    else break;
                } else if (p == abuelo->m_pSon[1]->m_pSon[0]) RR(p->m_pSon[2]);
```

```
                else {
                    p->m_pSon[2]->m_Color = 0;
                    abuelo->m_Color = 1;
                    LR(abuelo);
                    if (abuelo->m_Dato != m_pRoot->m_Dato) p = abuelo;
                    else break;
                }
            }
        }
        m_pRoot->m_Color = 0;
    }

    bool Remove(T D){
        return Remove(D, &(m_pRoot));
    }

    bool Remove(T D, Nodo<T> **p_tmp){
        if (!(*p_tmp)) return false;
        if ((*p_tmp)->m_Dato == D){
            if ((*p_tmp)->m_pSon[0] && !(*p_tmp)->m_pSon[1]){
                Nodo<T> *p_tmp1 = *p_tmp;
                (*p_tmp) = (*p_tmp)->m_pSon[0];
                FixRemove(*p_tmp);
                delete p_tmp1;
                p_tmp1 = 0;
                return true;
            }else if (!(*p_tmp)->m_pSon[0] && (*p_tmp)->m_pSon[1]){
                Nodo<T> *p_tmp1 = *p_tmp;
                (*p_tmp) = (*p_tmp)->m_pSon[1];
                FixRemove(*p_tmp);
                delete p_tmp1;
                p_tmp1 = 0;
                return true;
            }else if ((*p_tmp)->m_pSon[0] && (*p_tmp)->m_pSon[1]){
                Nodo<T> *p_tmp1 = *p_tmp;
                Nodo<T> *p_tmp2 = (*p_tmp)->m_pSon[1], *p_tmp3 = (*p_tmp)->m_pSon[1], *p;
                while (p_tmp2->m_pSon[0]){
                    p = p_tmp2;
                    p_tmp2 = p_tmp2->m_pSon[0];
                }
                Nodo<T> *p_tmpSon = p_tmp2->m_pSon[1];
                p_tmp2->m_pSon[0] = (*p_tmp)->m_pSon[0];
                if (p_tmp2 != p_tmp3){
                    p_tmp2->m_pSon[1] = p_tmp3;
                    p->m_pSon[0] = p_tmpSon;
                }

                *p_tmp = p_tmp2;
                FixRemove(*p_tmp);
                delete p_tmp1;
                p_tmp1 = 0;
                return true;
            }else{
                FixRemove(*p_tmp);
                delete *p_tmp;
                *p_tmp = 0;
                return true;
```

```
            }
        }
        return Remove(D, &((*p_tmp)->m_pSon[(*p_tmp)->m_Dato < D]));
    }

    void FixRemove(Nodo<T> *p){
        while(p->m_Dato != m_pRoot->m_Dato && p->m_Color == 0){
            Nodo<T> * p_tmp = m_pRoot;
            if(p->m_pSon[2]->m_pSon[0] == p){
                if(p->m_pSon[2]->m_pSon[1]) p_tmp = p->m_pSon[2]->m_pSon[1];
                if(p_tmp){
                    if(p_tmp->m_Color == 1){
                        p_tmp->m_Color = 0;
                        p->m_pSon[2]->m_Color = 1;
                        LR(p->m_pSon[2]);
                        p_tmp = p->m_pSon[2]->m_pSon[1];
                    }
                    if(!p_tmp->m_pSon[0] && !p_tmp->m_pSon[1]){
                        p_tmp->m_Color = 1;
                        p = p->m_pSon[2];
                    } else if (p_tmp->m_pSon[0]->m_Color == 0 && p_tmp->m_pSon[1]->m_Color == 0)
                        p_tmp->m_Color = 1;
                        p = p->m_pSon[2];
                    } else if(p_tmp->m_pSon[1]->m_Color == 0){
                        p_tmp->m_pSon[0]->m_Color = 0;
                        p_tmp->m_Color = 1;
                        RR(p_tmp);
                        p_tmp = p->m_pSon[2]->m_pSon[1];
                    } else {
                        p_tmp->m_Color = p->m_pSon[2]->m_Color;
                        p->m_pSon[2]->m_Color = 0;
                        if(p_tmp->m_pSon[1]) p_tmp->m_pSon[1]->m_Color = 0;
                        LR(p->m_pSon[2]);
                        p = m_pRoot;
                    }
                }
            } else if(p->m_pSon[2]->m_pSon[1] == p) {
                if(p->m_pSon[2]->m_pSon[0]) p_tmp = p->m_pSon[2]->m_pSon[0];
                if(p_tmp){
                    if(p_tmp->m_Color == 1){
                        p_tmp->m_Color = 0;
                        p->m_pSon[2]->m_Color = 1;
                        RR(p->m_pSon[2]);
                        p_tmp = p->m_pSon[2]->m_pSon[0];
                    }
                    if(!p_tmp->m_pSon[0] && !p_tmp->m_pSon[1]){
                        p_tmp->m_Color = 1;
                        p = p->m_pSon[2];
                    } else if (p_tmp->m_pSon[0]->m_Color == 0 && p_tmp->m_pSon[1]->m_Color == 0)
                        p_tmp->m_Color = 1;
                        p = p->m_pSon[2];
                    } else if(p_tmp->m_pSon[0]->m_Color == 0){
                        p_tmp->m_pSon[1]->m_Color = 0;
                        p_tmp->m_Color = 1;
                        RR(p_tmp);
                        p_tmp = p->m_pSon[2]->m_pSon[0];
                    } else {
```

```
                    p_tmp->m_Color = p->m_pSon[2]->m_Color;
                    p->m_pSon[2]->m_Color = 0;
                    if(p_tmp->m_pSon[0]) p_tmp->m_pSon[0]->m_Color = 0;
                    LR(p->m_pSon[2]);
                    p = m_pRoot;
                }
            }
        }
    }
    p->m_Color = 0;
}

void print(){
    print(m_pRoot);
}

void print(Nodo<T> *r){
    if (!r) return;
    cout<<r->m_Dato<<" ";
    //cout << " COLOR> " << r->m_Color << " dir> " << r << " _ data> " << r->m_Dato << " _ di
    print(r->m_pSon[0]);
    print(r->m_pSon[1]);
}
void dibujar(){
    ofstream archivo;
    archivo.open("RBT.dot");
    archivo << "graph B {\n";
    dibujar(m_pRoot, archivo);
    archivo << "\n}";
    archivo.close();
    //system("E: & cd Projects & cd First_Year & cd Laboratorio_ED_II & cd Graphviz & cd bin
}

void dibujar(Nodo<T> *r, ofstream &archivo){
    if (!r)
        return;
    archivo << r->m_Dato << "[label = \"" << r->m_Dato << " | " << r->m_Color << "\" ]; \n";
    if (r->m_pSon[0])
        archivo << r->m_Dato << " -- " << r->m_pSon[0]->m_Dato << ";\n";
    if (r->m_pSon[1])
        archivo << r->m_Dato << " -- " << r->m_pSon[1]->m_Dato << ";\n";
    dibujar(r->m_pSon[0], archivo);
    dibujar(r->m_pSon[1], archivo);
}

Nodo<T> *Find(T D){
    if (!m_pRoot)
        return nullptr;
    Nodo<T> *p_tmp = m_pRoot;
    while (p_tmp){
        if (p_tmp->m_Dato == D)
            return p_tmp;
        p_tmp = p_tmp->m_pSon[p_tmp->m_Dato < D];
    }
    return nullptr;
}
```

```
        Nodo<T> *Minimun(RedBlack_Tree<T> y){
            if (!y.m_pRoot)
                return NULL;
            Nodo<T> *p_tmp = y.m_pRoot;
            while (p_tmp->m_pSon[0]){
                p_tmp = p_tmp->m_pSon[0];
            }
            return p_tmp;
        }

};
```