MANUAL DEL USUARIO

Una guía para la utilización del árbol rojo-negro implementado

Autores: Jeaustin Sirias, Jose P. Apú, Juan F. Cortés & Marlon Lazo

1. Lo básico

Este manual muestra una estructura de datos basada en el arbol de busquedabinaria Rojo-Negro implementado en lenguaje de programación **C++**. Se introduce la documentación de sus alcances, capacidades, descarga, ejecución y uso de sus funciones a saber: inicializacion del arbol, busqueda, insercion y eliminacion de nodos.

Para implementar la estructura red black tree, se ha utilizado programación orientada a objetos, por lo que ha sido necesario aplicar el principio de encapsulamiento, esto se logra agrupando los métodos (funciones) y atributos (variables) dentro de la clase.

Para implementar una clase se deben de definir dos secciones, una pública y otra privada, por convención los atributos se definen en la parte privada y los métodos dependiendo si se requiere que se llamen únicamente desde la función principal main, se definen en la parte pública y si se desea que que estos permanezcan disponibles solo para el desarrollador, esto es que solo se puedan llamar dentro de la clase, se declaran en la parte privada. Es por este motivo que solo se mencionan las funciones públicas.

2. Descarga y acondicionamiento

Para acceder a los archivos ejecutables y documentación de este proyecto se debe de clonar el repositorio de github que que se encuentra en la dirección `https://github.com/josepabloapu/IE-0724-Proyecto-1.git` mediante el siguiente comando:

```
$ git clone https://github.com/josepabloapu/IE-0724-Proyecto-1.git
```

Una prueba para determinar si la operación ha resultado exitosa acompaña por defecto al directorio de la estructura; para verificarlo, cree una carpeta nominada 'build' dentro del directorio del proyecto usando los siguientes comandos en su terminal de comandos:

```
$ mkdir build
$ cd build/
$ cmake
$ make
$ ./exe
```

Esta carpeta debe de contener los archivos:

```
CMakeCache.txt  CMakeFiles  cmake_install.cmake  exe  librbt.so  Makefile
```

Los cuales son necesarios para para compilar y ejecutar el programa, además de esto, elimina archivos innecesarios. Una vez creada esta carpeta se puede comenzar a realizar pruebas a las funciones disponibles de la estructura.

3. Utilizando el árbol rojo-negro

La estructura tiene disponible los siguiente métodos:

```
rbt();
~rbt();
int rbt_create(std::vector<float> in_number_list);
int rbt_node_add(std::shared_ptr<rbt_node> new_node);
int rbt_node_remove(rbt_node *node_to_remove);
int rbt_search(float num, rbt_node *found_node);
int rbt_max_get(rbt_node *max_node);
int rbt_min_get(rbt_node *min_node);
int rbt_print();
```

rbt(): es ser el constructor de la clase, es una subrutina que se encarga de inicializar un objeto de la clase. Se le pasa por parámetro los valores iniciales que contendrá el objeto en la estructura.

~rbt(): es el destructor de la clase, método que se utiliza para destruir objetos del tipo de la clase. Este no recibe parámetros ni retorna algo. Se ejecuta de forma automática justo cuando el objeto alcanza el límite de su tiempo de vida.

int rbt_create(): crea la estructura según las reglas establecidas para arboles rojo negros, recibe como parámetro un vector de números flotantes, no admite otro tipo de caracteres, devuelve un código de error con valor 0 indicando que la operación ha sido exitosa.

int rbt_node_add(): su función es ingresar al árbol un nuevo nodo que contiene un número flotante, de esta forma se obtiene un sub-árbol. Recibe en su argumento un puntero al nodo a ingresar, devuelve el código de error 0 indicando que se ha insertado el nodo correctamente, o -1 en caso que se intente ingresar un dato no válido en el árbol.

int rbt_node_remove(): realiza la función inversa de "rbt_node_add()", se encarga de buscar un nodo para posteriormente eliminarlo, a su vez repara las posibles violaciones que se puedan dar en las reglas del árbol rojo negro, dejando el árbol balanceado. Recibe en su argumento el puntero al nodo a eliminar con el flotante como atributo, devuelve un código de error 0 en caso de ser exitosa la operación, -1 en caso de intentar ingresar un carácter no válido y un -4 si el valor no se encuentra en el árbol.

int rbt_search(): mediante esta función se busca un dato en la estructura, recibe como argumento el flotante a buscar y un puntero en el cual se guarda la dirección y el dato del nodo que contiene dicho número. Devuelve como código de error -4 en caso que el nodo no se encuentre en el árbol.

int rbt_max_get(): se encarga de buscar el nodo con el valor máximo en el árbol o sub-árbol. Recibe como parámetro el puntero a la raíz del árbol. Devuelve un entero de 0 como

código de error en caso de encontrarse algún valor o un -5 si el árbol está vacío.

int rbt_min_get(): se encarga de buscar el nodo con el valor mínimo a partir del nodo raíz que se le indique. Recibe como parámetro el puntero a la raíz del árbol. Devuelve el valor 0 en caso de encontrarse algún valor o -5 en caso que el árbol este vacío. Esta función devuelve la información del nodo, no hace modificaciones en el árbol.

int rbt_print(): imprime la información contenida en todos los nodos del árbol, su color, valor y el valor de sus hijos, esto lo hace para cada nodo del árbol, por lo que es posible visualizar la distribución de los datos en el árbol. Retorna 0 en caso de imprimirse los datos correctamente o -5 en caso que el árbol esté vacío.

4. Alcances

A continuación se muestra un ejemplo de cómo utilizar la estructura de datos implementada mediante las funcionalidades de sus métodos.

```
#include "rbt.h"
#include "rbt_node.h"

int main()
{
    //se inicializa un arbol vacio
    std::shared_prt<rbt> mi_arbol(new rbt());

    //se introduce un vector de numeros inicial
    std::vector<float> elementos{18, 9, 13, 6, 1, 3, 16, 50, 5, 4};

    //se inserta el vector de elementos al arbol:
    mi_arbol->rbt_create(elementos);

    //Se imprime el árbol para visualizar el resultado

    my_tree->rbt_print();

    //Se crea un nodo con a insertar
    std::shared_ptr<rbt_node> new_node->value = 10;

    //Se inserta un nodo
    int new_value = rbt_node_add(new_node)

    //Se elimina un nodo

    rbt_node *node_to_remove = new rbt_node();
    node_to_remove->value = 5;

    my_tree->rbt_node_remove(node_to_remove);

    //Se busca un valor

    //Se obtiene el valor máximo del árbol
```

```
    rbt_node *my_max_node = new rbt_node();
    int max = tree->rbt_max_get(my_max_node);

    //Se obtiene el valor mínimo del árbol

    rbt_node *my_min_node = new rbt_node();
    int min = tree->rbt_min_get(my_min_node);
}
```

5. Rendimiento: análisis temporal de la función de inserción

Se puede notar en las figuras 1 y 2 que la complejidad difiere mucho de la linealidad, que corresponde al peor caso de un árbol no autobalanceable, esto es al ingresarle un vector de números ordenados, ya que de esta forma el árbol se comporta como una lista y en consecuencia tiene una complejidad de $O(n)$.

La complejidad temporal que se ha obtenido para el método insertar es de $Olog(n)$, para esto se ha ingresando una serie de números aleatorios comprendidos entre 0 y 10 mil, esto con la finalidad de simular el caso promedio.

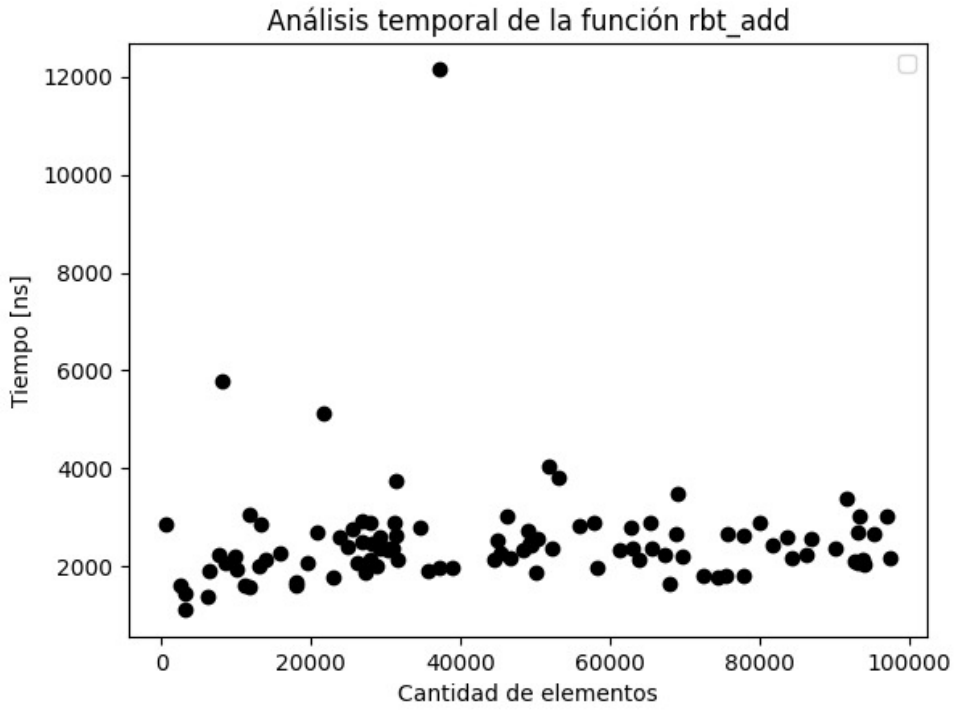

Figura 1: Inserción de 100 valores aleatorios con valores en el intervalo de [0,10.000] y sus respectivos tiempos de procesamiento

Ahora se procede a insertar una mayor cantidad de datos para obtener una mejor resolución del comportamiento de gráfica, y tener una mayor muestra para el análisis.

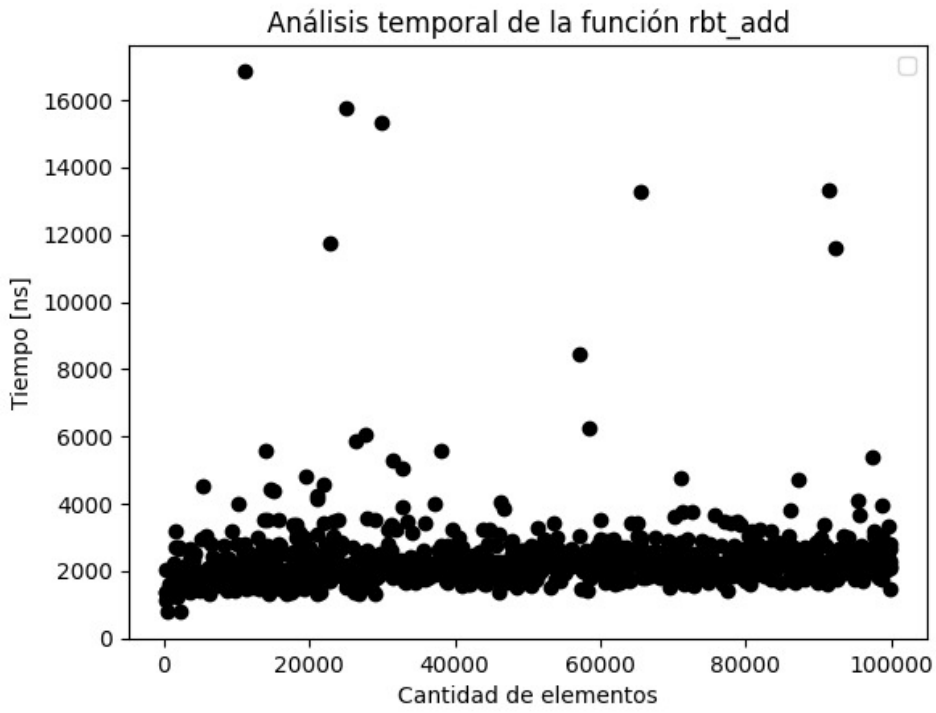

Figura 2: Inserción de 1000 valores aleatorios con valores en el intervalo de [0,10.000] y sus respectivos tiempos de procesamiento

Se obtiene el resultado esperado para un árbol autobalanceable del tipo rojo negro, esto implica que otros métodos de la clase tales como rotar o cambio de color también están bien implementadas. Se espera que la función remover que presenta una complejidad mayor entre las funciones de la estructura tenga un comportamiento similar al que se presentar en las figuras 1 y 2.