# Estructuras de Datos

Elizabeth Huang Wu C23913

***Resumen*—En este trabajo se realiza un análisis de varias estructuras de datos para hacer una comparación de complejidades temporales teóricas con los resultados prácticos. Con el desarrollo de varios experimentos se recolectan los datos necesarios y se llega a la conclusión que las complejidades teóricas son respaldadas por los tiempos de ejecución obtenidos en la mayor parte. Sin embargo se tiene que considerar como los datos se ingresan.**

***Palabras clave*—estructura, datos, lista, árbol, tiempos, rojinegro, tabla, hash, búsqueda.**

## I. Introducción

En este trabajo se refiere al tema de las estructuras de datos, que se define como una manera de guardar y organizar datos para facilitar las modificaciones y su acceso. A la hora de implementar el diseño de un algoritmo, es importante utilizar una estructura de datos apropiada, ya que no se puede utilizar solo una para todos los casos existentes. Dado a eso, es significativo conocer las limitaciones y ventajas de cada una.

Para poder realizar este análisis se realizara un experimento donde se puedan ver los tiempos de ejecución del algoritmo de búsqueda en diferentes estructuras de datos con diferentes inserciones: lista enlazada, árboles de búsqueda binaria, árbol rojinegro, tabla hash, inserción ordenada e inserción aleatoria. Más específicamente, se quiere comparar la complejidad temporal teórica con lo practico de las estructuras con los datos obtenidos.

1. Lista Enlazada: Una lista enlazada es una estructura donde los objetos se acomodan de una forma lineal. Es diferente a un arreglo ya que utiliza punteros en los objetos para navegar en vez de índices. En este caso se utilizara una lista simplemente enlazada, entonces solo tiene de atributo un puntero al siguiente.
2. Árbol de Búsqueda Binaria: Se organiza de en un árbol binario. Cada nodo contiene atributos como un hijo izquierdo, un hijo derecho y un puntero al padre. Si alguno de los atributos no existe, se le pone un valor NIL.
3. Árbol Rojinegro: Un árbol rojinegro es un árbol de búsqueda binaria que contiene un dato extra por nodo, su color. Los únicos dos colores posibles son rojo o negro. Con las restricciones dadas a base de los colores, se garantiza que el árbol no tenga un camino más grande que el doble de otro. En otras palabras, siempre es un árbol balanceado.
4. Tabla Hash: Las tablas hash son una estructura de datos utilizada para implementar un arreglo asociativo de claves a valores. Funcionan utilizando una función hash para calcular un índice desde donde se puede encontrar el valor deseado. Si dos claves diferentes generan el mismo índice, se utilizan técnicas como el encadenamiento o la direccionamiento abierto para resolver las colisiones.

## II. Metodología

Para lograr lo propuesto se piensa crear un programa en el lenguaje programador c++ que nos permitirá hacer las pruebas necesarias para el análisis y la comparación de estas estructuras. El experimento se realizara en la IDE 'Visual Studio Code', en una computadora con sistema operativo macOS con un procesador 6-Core Intel Core i5, GPU AMD Radeon Pro 5300 4 GB y RAM 8 GB 2667 MHz DDR4. Es importante tener en cuenta el hardware en el cual se realizan estas pruebas ya que los resultados pueden variar del dispositivo.

Tanto el código, se creara un archivo .h para cada estructura de datos y se implementaran los operadores básicos, entre estos: crear, destruir, insertar, eliminar y buscar. El código a utilizar para cada estructura se muestra en los apéndices, son basados en el pseudocódigo del libro de Cormen y colaboradores [1]. En esta investigación, se utilizaran las estructuras de lista enlazada y árbol de búsqueda binaria para desarrollar los experimentos prácticos. Para cada estructura, se insertaran 1 000 000 elementos y se buscaran 10 000 al menos 4 veces y se guardaran los tiempos que dura en buscar dicha claves. Habrán dos tipos de inserciones, aleatoria y ordenada, esto es para poder diferenciar la duración del operador en ambos casos. El rango de los datos aleatorio sera de [0, 2n), n siendo el número de elementos por ser insertados. La búsqueda también sera aleatoria con números entre [0, 2n).

Para poder capturar los tiempos de ejecución de las búsquedas, se utilizara la biblioteca <chrono>. Antes de la operación y después de la operación se guarda el tiempo actual, luego se resta el final con el principio para obtener el tiempo de duración. En este trabajo se estará utilizando milisegundos para poder hacer un análisis con datos más precisos y detallados.

## III. Resultados

Los tiempos de ejecución de las 4 corridas de los del algoritmo de búsqueda en una lista enlazada se muestra en el cuadro I. Los tiempos del árbol de búsqueda binaria, se encuentran en el cuadro II. Los tiempos del árbol rojinegro, se encuentran en el cuadro III. Por ultimo, los tiempos de la tabla hash, se encuentran en el cuadro IV.

**Lista Enlazada**

1. Lista Aleatoria: Entre los datos recolectados, se ve una desviación estándar de aproximadamente 348.70. De esto se puede deducir que los datos son relativamente consistentes, no muy dispersos pero sí tiene variabilidad. Esta consistencia se puede ver ya que los valores se acercan a la media 34459.9.

Cuadro I
TIEMPO DE EJECUCIÓN DEL ALGORITMO DE BÚSQUEDA EN UNA LISTA ENLAZADA.

| | Tiempo (ms) | | | | |
|---|---|---|---|---|---|
| | Corrida | | | | |
| Inserción | 1 | 2 | 3 | 4 | Prom. |
| Aleatorio | 34946.8 | 34217.9 | 34620.8 | 34053.9 | 34459.9 |
| Ordenado | 32037.7 | 32510.0 | 32601.9 | 32525.4 | 33168.8 |

Cuadro II
TIEMPO DE EJECUCIÓN DEL ALGORITMO DE BÚSQUEDA EN UN ÁRBOL DE BÚSQUEDA BINARIA.

| | Tiempo (ms) | | | | |
|---|---|---|---|---|---|
| | Corrida | | | | |
| Inserción | 1 | 2 | 3 | 4 | Prom. |
| Aleatorio | 8.8 | 7.8 | 7.7 | 7.4 | 7.9 |
| Ordenado | 30194.5 | 29913.1 | 30029.7 | 29885.7 | 30005.8 |

2. Lista Ordenada: De los datos obtenidos, hay una desviación estándar alrededor de 222.73. Similar a la lista aleatoria, los tiempos de búsqueda son relativamente consistentes con una media de 33168.8, pero sí hay variabilidad.

**Árbol de Búsqueda Binaria**

1. Árbol de Búsqueda Binaria Aleatorio: Los tiempos recolectados en las corridas son parecidas con una desviación de 0.5262. Con esto se puede decir que hay una consistencia alta y poca variabilidad. También se puede asumir dado a que todos los datos se acercan a la media 7.9.
2. Árbol de Búsqueda Binaria Ordenado: La desviación estándar de los datos recolectados es de 121.65, son relativamente consistentes pero sí tienen variabilidad. Se refleja la consistencia dado a que los datos se acercan a la media 30005.8.

Cuadro III
TIEMPO DE EJECUCIÓN DEL ALGORITMO DE BÚSQUEDA EN UN ÁRBOL ROJINEGRO.

| | Tiempo (ms) | | | | |
|---|---|---|---|---|---|
| | Corrida | | | | |
| Inserción | 1 | 2 | 3 | 4 | Prom. |
| Aleatorio | 3.8 | 4.2 | 3.9 | 4.0 | 4.0 |
| Ordenado | 4.1 | 4.3 | 4.0 | 3.7 | 4.0 |

Cuadro IV
TIEMPO DE EJECUCIÓN DEL ALGORITMO DE BÚSQUEDA EN UNA TABLA HASH.

| | Tiempo (ms) | | | | |
|---|---|---|---|---|---|
| | Corrida | | | | |
| Inserción | 1 | 2 | 3 | 4 | Prom. |
| Aleatorio | 1.9 | 1.6 | 1.2 | 1.4 | 1.5 |
| Ordenado | 2.4 | 1.8 | 3.2 | 2.4 | 2.5 |

**Árbol Rojinegro**

1. Árbol Rojinegro Aleatorio: Los datos extraídos de las pruebas muestran que hay una variabilidad baja, que se puede ver con la desviación estándar de 0.1479. También se ve una consistencia alta, ya que los datos obtenidos se acercan a la media de 4.
2. Árbol Rojinegro Ordenado: La desviación estándar de los datos obtenidos fue de 0.2165 y todo se acercan a la media 4. De esto se puede asumir que los datos tienen poca variabilidad y una consistencia alta.

**Tabla Hash**

1. Tabla Hash Aleatorio: Se considera que los datos tienen poca variabilidad dado a la desviación estándar de 0.2586. Los datos se acercan a su media de 1.5 y dan a saber que tienen una consistencia significante.
2. Tabla Hash Ordenado: Los datos tienen una desviación estándar de 0.4974, además se acercan a la media de 2.5. Se sabe que tiene una variabilidad baja y una consistencia alta.

**Tipos de Inserción**

Los resultados se pueden ver en la figura 1 y 2.

1. Inserción Aleatoria: El promedio de tiempo para la lista simplemente enlazada fue más lenta que la del árbol de búsqueda binaria, la tabla hash y el árbol rojinegro. La diferencia es extraordinaria ya que excede el doble del tiempo.
2. Inserción Ordenada: A diferencia de la inserción aleatoria, en la inserción ordenada ambas estructuras, la lista y el árbol binario, tienen tiempos de búsqueda similares y se encuentran del lado más lento. Por el otro lado, la tabla hash y el árbol rojinegro tienen tiempos bajos y parecidos a la inserción aleatoria.

## IV. CONCLUSIONES

A partir de los resultados se puede concluir que para la lista enlazada entre los dos casos de inserción, los promedios no tienen una gran diferencia, sin embargo sí se nota que los tiempos de la lista ordenada son más bajos. Los resultados son los esperados considerando la complejidad temporal teórica. La lista aleatoria con cada búsqueda requiere un escaneo completo de la lista en el peor caso. Mientras que en la lista ordenada, tiene el beneficio de saber si un elemento no existe dependiendo de los valores que se van recorriendo que potencialmente reduzca el tiempo de búsqueda. En ambas listas, la complejidad en el peor caso es de $O(n)$.

Mientras que para el árbol de búsqueda binaria, entre los dos tipos de inserciones, se ve una diferencia significativa de tiempos de ejecución ya que el promedio del ordenado es más que la mitad del tiempo promedio del aleatorio. Estos resultados son esperados dado a que el árbol aleatorio puede ser más equilibrado lo cual significa que la altura del árbol no es tan larga y reduce el tiempo de búsqueda. Mientras que un árbol con inserción ordenada tiene una altura significativamente larga ya que la entrada es de manera descendiente

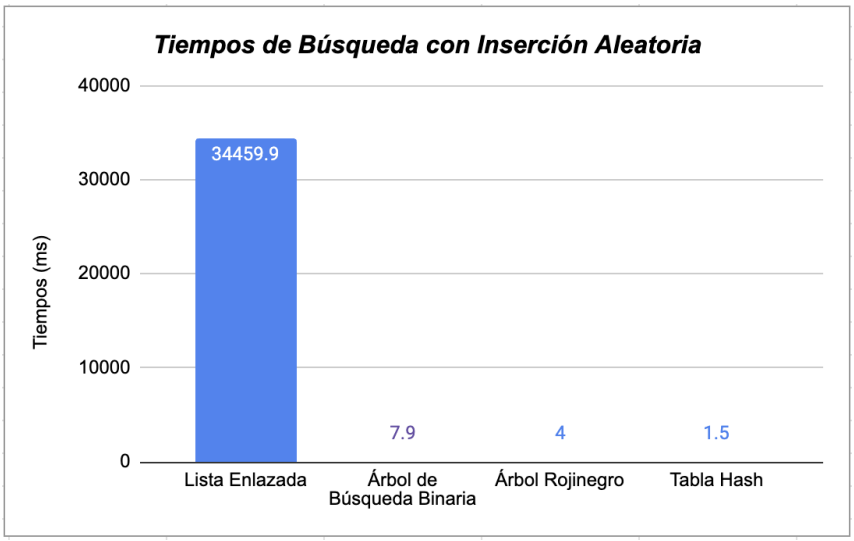



Figura 1. Gráfico comparativo de los tiempos promedio de búsqueda aleatoria en las estructuras de datos con inserción aleatoria.

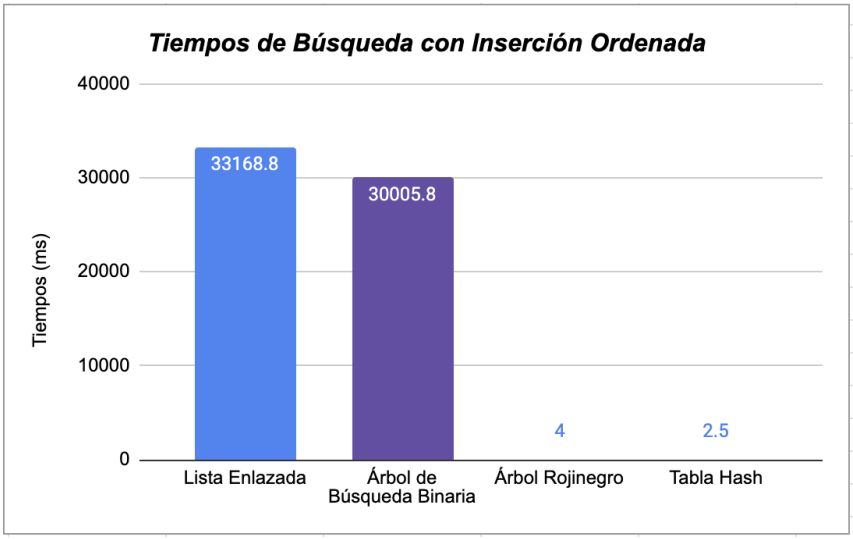



Figura 2. Gráfico comparativo de los tiempos promedio de búsqueda aleatoria en las estructuras de datos con inserción ordenada.

o ascendiente. Es decir, o solo hay hijos izquierdos o solo hay hijos derechos. Este tipo de inserción hace que el árbol se parezca a una lista enlazada . Dado a su similitud de la estructura, su tiempo de búsqueda se acerca a la de la lista simplemente enlazada. El peor caso de una búsqueda en un árbol de inserción aleatoria es de *O(logn)*. La inserción ordenada tiene una complejidad de *O(n)*, dado a que es posible que se tenga que recorrer el árbol completo para encontrar la llave deseada.

El árbol rojinegro no tiene diferencias significativas entre los tiempos de búsqueda en los dos tipos de inserciones, ya que ambos tienen un promedio de 4.0 milisegundos. Los resultados son los esperados ya que el árbol rojinegro se balancea de manera automática tras cada inserción y eliminación, a diferencia de un árbol de búsqueda binaria. Esto significa que la altura del árbol garantiza ser de *O(logn)*, dado a las propiedades de la estructura. En ambos casos, el tiempo de la búsqueda es proporcional a *O(logn)*.

Por último, la tabla hash también se encuentra con promedios similares en ambos casos, con una diferencia de un milisegundo. Los resultados son los esperados, bajos y parecidos en ambos casos. Con la inserción ordenada, la tabla hash cuenta con una distribución más balanceada ya que tiene un tamaño m cuyo factor de carga sea = 1, es decir que cada llave tiene un dato. La duración de esta búsqueda es de O(1) ya que siempre es exitosa. Mientras que con la inserción aleatoria, es posible que los datos se encadenen si se genera un número duplicado. En este caso, la búsqueda podría durar un poco menos si en la llave en la tabla se encuentra vacía, ya que se retorna de forma inmediata. En lo teórico, la complejidad de la búsqueda no exitosa seria de O(1 + ), siendo el tamaño de la lista con la clave por recorrer.

Entre las estructuras se ve una diferencia significativa entre los promedios de búsqueda con inserción aleatoria. La lista enlazada tarda más que el doble buscando un dato comparado a al árbol de búsqueda binaria. Esta diferencia significativa se

da ya que la lista requiere recorrer todos los nodos uno por uno y no tiene manera de acelerar este proceso. La capacidad del árbol binario de búsqueda de dividir y conquistar hace que el tiempo se mantenga parecido a O(logn). El árbol rojinegro en todo momento mantiene un altura y tiempo de O(logn). Por ultimo, la tabla hash con inserciones aleatorias y una función hash distribuyen los elementos de manera uniforme en la tabla, minimizando las colisiones y permitiendo búsquedas en tiempo de O(1 + ).

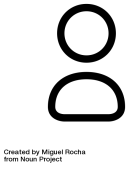
La diferencia entre tiempos también es significativa en la inserción ordenada. En el caso de la lista, la única diferencia es el orden en el cual se ingresan los datos. Es posible reducir el tiempo ya que el dato si existe en la lista en todo momento, mientras que en la inserción aleatoria hay una posibilidad que no exista el dato y se recorra toda la lista. En el caso del árbol de búsqueda binaria, la inserción ordenada causa que el árbol tome la forma de una lista enlazada y tienda a tiempo O(n) en el caso que se tenga que recorrer todo el árbol. El árbol rojinegro queda igual por sus propiedades y el balanceo automático por inserción. En el caso de la tabla hash, la pequeña diferencia es posiblemente dado al caso que no exista el número y la lista con la clave este vacía, en ese caso el tiempo de duración es mínimo.

El experimento y análisis que se llevó a cabo nos ayuda a decidir cual es la mejor estructura de datos dependiendo del tipo de problema, algoritmo e inserción de datos en situaciones reales. Nos deja un conocimiento más profundo y valioso que nos ayuda reconocer el rendimiento de cada uno. Los resultados nos deja concluir que las complejidades temporales teóricas son respaldadas por los tiempos prácticos.

## REFERENCIAS

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

**Elizabeth Huang** Estudiante de la Universidad de Costa Rica.



Created by Miguel Rocha
from Noun Project

## APÉNDICE A
## CÓDIGO DE LOS ALGORITMOS

El código se muestra en los algoritmos 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 y 12.

**Algoritmo 1** Algoritmo del nodo de una lista simplemente enlazada.

```
// Nodos de la lista:
template <typename T>
class llnode
{
private:
    T key;
    llnode<T> *next;

public:
    llnode() {
        this->next = nullptr;
    };

    llnode (const T& k, llnode<T> *y = nullptr):key(k), next(y)  {};

    ~llnode() {};

    llnode<T>* GetNext() {
        return next;
    }

    void SetNext(llnode<T>* next){
        this->next = next;
    }

    int GetKey(){
        return key;
    }

};
```

**Algoritmo 2** Algoritmo de la lista simplemente enlazada.

```
// Lista enlazada con nodo centinela:
template <typename T>
class llist
{
private:
    llnode<T> *nil;

public:
    llist() {
        nil = new llnode<T>();
        nil->SetNext(nil);
    };

    ~llist() {
        while (nil->GetNext() != nil){
            Delete(nil->GetNext());
        }
        delete nil;
    };

    void Insert(llnode<T>* x) {
        x->SetNext(nil->GetNext());
        nil->SetNext(x);
    };

    llnode<T>* Search(const T& k) {
        llnode<T>* x = nil->GetNext();
        while (x != nil) {
            if (x->GetKey() == k) {
                return x;
            }
            x = x->GetNext();
        }
        return this->nil;
    };

    void Delete(llnode<T>* x) {
        llnode<T>* prev = nil;
        llnode<T>* curr = nil->GetNext();

        while (curr != nil && curr != x) {
            prev = curr;
            curr = curr->GetNext();
        }

        if (curr == x) {
            prev->SetNext(curr->GetNext());
            delete curr;
        }
    };
};
```

**Algoritmo 3** Algoritmo del nodo de un árbol de búsqueda binaria.

```
template <typename T>
class bstnode
{
private:
    T key;
    bstnode<T> *p, *left, *right;
public:
    bstnode() {};

    bstnode(const T& k, bstnode<T> *w = nullptr, bstnode<T> *y = nullptr, bstnode<T> *z = null
        key(k), p(w), left(y), right(z) {};

    ~bstnode() {
    };

    int GetKey(){
        return key;
    }

    bstnode<T>* GetParent(){
        return p;
    }

    bstnode<T>* GetLeft(){
        return left;
    }

    bstnode<T>* GetRight(){
        return right;
    }

    void SetParent(bstnode<T>* p){
        this->p = p;
    }

    void SetLeft(bstnode<T>* left){
        this->left = left;
    }

    void SetRight(bstnode<T>* right){
        this->right = right;
    }
};
```

**Algoritmo 4** Algoritmo de un árbol de búsqueda binaria.

```
template <typename T>
class bstree {
private:
    bstnode<T> *root;
public:
    bstree() {
        this->root = nullptr;
    };

    ~bstree() {
        clearTree(root);
    };

    void clearTree(bstnode<T>* x){
        if (x) {
            clearTree(x->GetLeft());
            clearTree(x->GetRight());
            delete x;
        }
    }

    bstnode<T>* GetRoot(){
        return root;
    }

    void OrderedTree(int n) {
        root = new bstnode<int>(0);
        bstnode<int>* aux = root;

        for (int i = 1; i < n; i++){
            bstnode<int>* newNode = new bstnode<int>(i);
            aux->SetRight(newNode);
            newNode->SetParent(aux);
            aux = newNode;
            std::cout << i << std::endl;
        }
    }
```

**Algoritmo 5** Algoritmo de un árbol de búsqueda binaria.

```
void Insert(bstnode<T>* z) {
    bstnode<T>* x = root;
    bstnode<T>* y = nullptr;
    while (x != nullptr) {
        y = x;
        if (z->GetKey() < x->GetKey()){
            x = x->GetLeft();
        } else {
            x = x->GetRight();
        }
    }
    z->SetParent(y);
    if (y == nullptr) {
        root = z;
    } else if (z->GetKey() < y->GetKey()) {
        y->SetLeft(z);
    } else {
        y->SetRight(z);
    }
};
void InorderWalk(bstnode<T> *x) {
    if (x != nullptr) {
        InorderWalk(x->GetLeft());
        std::cout << x->GetKey() << std::endl;
        InorderWalk(x->GetRight());
    }
};

bstnode<T>* Search(bstnode<T> *x, const T& k) {
    if (x == nullptr || k == x->GetKey()) {
        return x;
    }
    if (k < x->GetKey()) {
        return Search(x->GetLeft(), k);
    } else {
        return Search(x->GetRight(), k);
    }
};

bstnode<T>* IterativeSearch(bstnode<T> *x, const T& k) {
    // Mientras x no sea puntero nulo y la clave de x no sea igual a k
    while (x != nullptr && k != x->GetKey()) {
        // Si la clave es menor a k
        if (k < x->GetKey()) {
            // Se mueve al hijo izquierdo de x
            x = x->GetLeft();
        } else {
            // Si no, se mueve al hijo derecho de x
            x = x->GetRight();
        }
    }
    // Retornar el nodo con la clave buscada
    return x;
};
```

**Algoritmo 6** Algoritmo de un árbol de búsqueda binaria.

```
    bstnode<T>* Minimum(bstnode<T> *x) {
        while (x->GetLeft() != nullptr) {
            x = x->GetLeft();
        }
        return x;
    };

    bstnode<T>* Maximum(bstnode<T> *x) {
        while (x->GetRight() != nullptr) {
            x = x->GetRight();
        }
        return x;
    };

    bstnode<T>* Successor(bstnode<T> *x) {
        if (x->GetRight() != nullptr) {
            return Minimum(x->GetRight());
        } else {
            bstnode<T>* y = x->GetParent();
            while (y != nullptr && x == y->GetRight()) {
                x = y;
                y = y->GetParent();
            }
            return y;
        }
    };

    void Delete(bstnode<T>* z) {
        if (z->left == nullptr) {
            Transplant(z, z->GetRight());
        } else if (z->GetRight() == nullptr) {
            Transplant(z, z->GetLeft());
        } else {
            bstnode<T>* y = Minimum(z->GetRight());
            if (y->GetParent() != z) {
                Transplant(y, y->GetRight());
                y->SetRight(z->GetRight());
                y->GetRight()->SetParent(y);
            }
            Transplant(z, y);
            y->SetLeft(z->GetLeft());
            y->GetLeft()->SetParent(y);
        }
        delete z;
    };
    void Transplant(bstnode<T>* u, bstnode<T>* v) {
        if (u->GetParent() == nullptr) {
            root = v;
        } else if (u == u->GetParent()->GetLeft()) {
            u->GetParent()->SetLeft(v);
        // Si x es hijo derecho
        } else {
            u->GetParent()->SetRight(v);
        }
        if (v != nullptr) {
            v->SetParent(u->GetParent());
        }
    }
};
```

**Algoritmo 7** Algoritmo de un árbol rojinegro.

```
enum colors {RED, BLACK};

class rbtnode
{
private:
    T key;
    rbtnode<T> *p, *left, *right;
    enum colors color;

public:
    rbtnode() {
    };

    rbtnode (const T& k, rbtnode<T> *w = nullptr, rbtnode<T> *y = nullptr, rbtnode<T> *z = nul

    ~rbtnode() {
    };

    int GetKey() {
        return this->key;
    }

    void SetPadre(rbtnode<T>* w) {
        this->p = w;
    };

    rbtnode<T>* GetPadre(){
        return this->p;
    }

    void SetLeft(rbtnode<T>* y) {
        this->left = y;
    };

    rbtnode<T>* GetLeft() {
        return this->left;
    }

    void SetRight(rbtnode<T>* z) {
        this->right = z;
    }

    rbtnode<T>* GetRight() {
        return this->right;
    }

    void SetColor(colors c){
        this->color = c;
    }

    colors GetColor(){
        return this->color;
    }
};
```

**Algoritmo 8** Algoritmo de un árbol rojinegro.

```
template <typename T>
class rbtree {
public:
    rbtnode<T> *root;
    rbtnode<T> *nil;

    rbtree() {
        this->nil = new rbtnode<T>();
        nil->SetColor(BLACK);
        this->root = this->nil;
    };

    ~rbtree() {
        destroyTree(root);
        delete nil;
    };

    void destroyTree(rbtnode<T>* n){
        if (n != nil) {
            destroyTree(n->GetLeft());
            destroyTree(n->GetRight());
            delete n;
        }
    }

    void Insert(rbtnode<T>* z) {
        rbtnode<T>* x = root;
        rbtnode<T>* y = nil;
        while (x != nil) {
            y = x;
            if (z->GetKey() < x->GetKey()){
                x = x->GetLeft();
            } else {
                x = x->GetRight();
            }
        }

        z->SetPadre(y);

        if (y == nil) {
            root = z;
        } else if (z->GetKey() < y->GetKey()) {
            y->SetLeft(z);
        } else {
            y->SetRight(z);
        }

        z->SetLeft(nil);
        z->SetRight(nil);
        z->SetColor(RED);

        InsertFixUp(z);
    };
```

**Algoritmo 9** Algoritmo de un árbol rojinegro.

```
void InsertFixUp(rbtnode<T>* z) {
    while(z->GetPadre()->GetColor() == RED){
        if (z->GetPadre() == z->GetPadre()->GetPadre()->GetLeft()){
            rbtnode<T>* y = z->GetPadre()->GetPadre()->GetRight();
            if (y->GetColor() == RED) {
                z->GetPadre()->SetColor(BLACK);
                y->SetColor(BLACK);
                z->GetPadre()->GetPadre()->SetColor(RED);
                z = z->GetPadre()->GetPadre();
            } else {
                if (z == z->GetPadre()->GetRight()){
                    z = z->GetPadre();
                    RotateLeft(z);
                }
                z->GetPadre()->SetColor(BLACK);
                z->GetPadre()->GetPadre()->SetColor(RED);
                RotateRight(z->GetPadre()->GetPadre());
            }
        } else {
            rbtnode<T>* y = z->GetPadre()->GetPadre()->GetLeft();
            if (y->GetColor() == RED) {
                z->GetPadre()->SetColor(BLACK);
                y->SetColor(BLACK);
                z->GetPadre()->GetPadre()->SetColor(RED);
                z = z->GetPadre()->GetPadre();
            } else {
                if (z == z->GetPadre()->GetLeft()){
                    z = z->GetPadre();
                    RotateRight(z);
                }
                z->GetPadre()->SetColor(BLACK);
                z->GetPadre()->GetPadre()->SetColor(RED);
                RotateLeft(z->GetPadre()->GetPadre());
            }

        }
    }
    root->SetColor(BLACK);
};

void RotateLeft(rbtnode<T>* z) {
    rbtnode<T>* y = z->GetRight();
    z->SetRight(y->GetLeft());
    if (y->GetLeft() != nil){
        y->GetLeft()->SetPadre(z);
    }
    y->SetPadre(z->GetPadre());

    if (z->GetPadre() == nil) {
        this->root = y;
    } else if (z == z->GetPadre()->GetLeft()) {
        z->GetPadre()->SetLeft(y);
    } else {
        z->GetPadre()->SetRight(y);
    }
    y->SetLeft(z);
    z->SetPadre(y);
};
```

**Algoritmo 10** Algoritmo de un árbol rojinegro.

```
void RotateRight(rbtnode<T>* z) {
        rbtnode<T>* y = z->GetLeft();
        z->SetLeft(y->GetRight());
        if (y->GetRight() != nil) {
            y->GetRight()->SetPadre(z);
        }
        y->SetPadre(z->GetPadre());

        if (z->GetPadre() == nil) {
            this->root = y;
        } else if (z == z->GetPadre()->GetRight()) {
            z->GetPadre()->SetRight(y);
        } else {
            z->GetPadre()->SetLeft(y);
        }
        y->SetRight(z);
        z->SetPadre(y);
    };

    void InorderWalk(rbtnode<T> *x) {
        if (x != nil){
            InorderWalk(x->GetLeft());
            std::cout << x->GetKey() << std::endl;
            InorderWalk(x->GetRight());
        }
    };

    rbtnode<T>* Search(rbtnode<T> *x, const T& k) {
        if (x == nil || k == x->GetKey()) {
            return x;
        }
        if (k < x->GetKey()) {
            return Search(x->GetLeft(), k);
        } else {
            return Search(x->GetRight(), k);
        }
    };

    rbtnode<T>* IterativeSearch(rbtnode<T> *x, const T& k) {
        while (x != nil && k != x->GetKey()) {
            if (k <  x->GetKey()) {
                x = x->GetLeft();
            } else {
                x = x->GetRight();
            }
        }
        return x;
    };
```

**Algoritmo 11** Algoritmo de un árbol rojinegro.

```
rbtnode<T>* Minimum(rbtnode<T> *x) {
    while (x->GetLeft() != nil) {
        x = x->GetLeft();
    }
    return x;
};

rbtnode<T>* Maximum(rbtnode<T> *x) {
    while (x->GetRight() != nil) {
        x = x->GetRight();
    }
    return x;
};

rbtnode<T>* Successor(rbtnode<T> *x) {
    if (x->GetRight() != nil) {
        return Minimum(x->GetRight());
    } else {
        rbtnode<T>* y = x->GetPadre();
        while (y != nil && x == y->GetRight()) {
            x = y;
            y = y->GetPadre();
        }
        return y;
    }

};
```

**Algoritmo 12** Algoritmo de una tabla hash.

```
template <typename T>
class chtable {
public:
    chtable(int sz) : size(sz) , table(sz) {
    };

    ~chtable() {};

    void Insert(const T& k) {
        int q = hash(k);
        table[q].Insert(new llnode<T>(k));
    };

    T* Search(const T& k) {
        int q = hash(k);
        T* result = new int(table[q].Search(k)->GetKey());
        return result;
    };

private:
    int size;

    int hash(const T& key) const {
        return (key) % size;
    }

    std::vector< llist<T> > table;
};
```