

ACTIVIDAD 3.4

BINARY SEARCH TREE

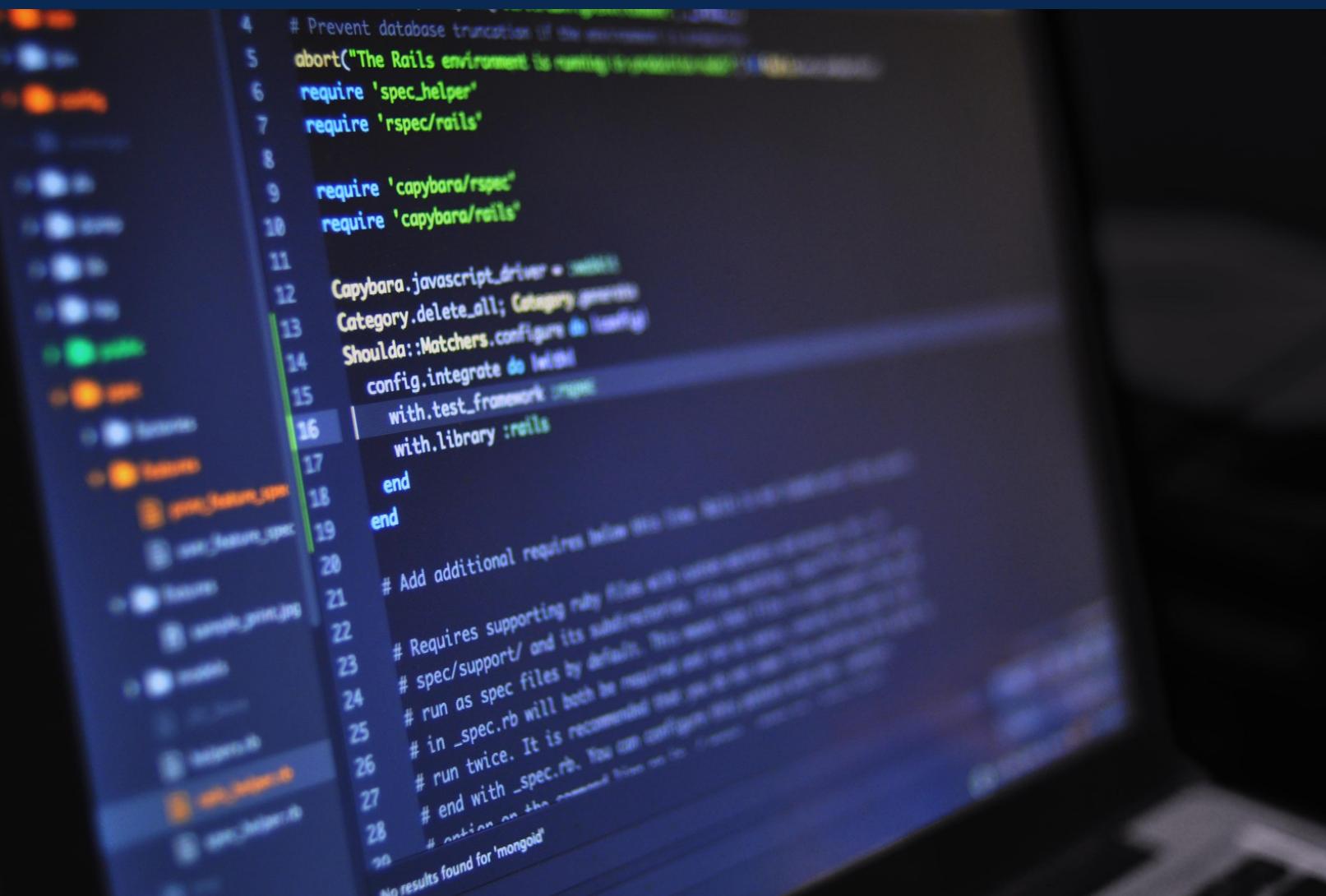
INTEGRANTES:

ANDRÉS EMILIANO DE LA GARZA ROSALES A01384096

RODRIGO DE JESÚS MELÉNDEZ MOLINA A00831646

FIDEL MORALES BRIONES A01198630

MÉTODOS, CLASES Y FUNCIONES UTILIZADAS



EVIDENCIA COMPETENCIA

```

long long ipToLong(string ip1, string ip2, string ip3, string i
string current, ip = "";
vector<string> ips = { ip1, ip2, ip3, ip4 };

if (ips[0][0] == ' ')
    ips[0].erase(0, 1);

for (int i = 0; i < 4; i++)
{
    current = ips[i];

    if (current.length() == 1) {
        ip += "00" + current;
    }
    else if (current.length() == 2) {
        ip += "0" + current;
    }
    else {
        ip += current;
    }
}

return stoll(ip);
}

```

ipToLong()

Este método se encarga de convertir las IP's del archivo de string a long long. Primero se crea un vector de strings con cada uno de los cuatro valores de una IP. Se remueve el espacio en blanco que existe en el primer elemento. Y después se agregan los ceros necesarios a los números que sean de uno o dos dígitos para juntar los cuatro valores de la IP. Finalmente se convierte de string a long long.

Salidas: una IP en tipo de variable long long
Complejidad en tiempo: O(1)
Complejidad en espacio: O(1)

TimeTest

```
template <class T>
void timeTest(T tree, string fileName, string treeName) {
    clock_t time;
    fstream bitacora;
    string line, ip1, ip2, ip3, ip4;
    long long ipGenerada;

    bitacora.open(fileName, ios::in);

    time = clock();
    while (getline(bitacora, line, 'P')) {
        getline(bitacora, line, ' ');
        getline(bitacora, ip1, '.');
        getline(bitacora, ip2, '.');
        getline(bitacora, ip3, '.');
        getline(bitacora, ip4, ' ');
        ipGenerada = ipToLong(ip1, ip2, ip3, ip4);
        getline(bitacora, line, '\n');
        tree.search(ipGenerada);
    }
    time = clock() - time;
    cout << treeName << (float)time / CLOCKS_PER_SEC << " segundos" << endl;

    bitacora.close();
}
```

Esta función la utilizamos para tomar el tiempo que el programa necesita para buscar cada una de las direcciones IP de un archivo .txt dentro de un árbol

Salidas: Un valor entero con la cantidad de tiempo que se tardo
Complejidad en tiempo: $O(n \log n)$
Complejidad en espacio: $O(1)$

class Node

Clase nodo la utilizamos para crear el constructor y tener 3 punteros diferentes, uno que va hacia los hijos derecha e izquierda respectivamente y un puntero hacia el padre.

Dentro de la clase tenemos diferentes funciones como:

- Funcion GetData, obtener el valor de los datos
Complejidad O(1)
- getAccess, la cantidad de accesos al nodo
Complejidad O(1)
- Funciones que establecen punteros a los hijos con los valores proporcionados
Complejidades O(1)
- getParent, funcion que devuelve el puntero al padre nodo con el valor proporcionado
Complejidad O(1)
- addAccess, incrementar el contador por cada vez que se accesa a un nodo.
Complejidad O(1)

```
template <class T>
class Node {
private:
    T data;
    Node<T>* left;
    Node<T>* right;
    Node<T>* parent;
    int access = 0;

public:
    Node(T data) {
        this->data = data;
        this->left = nullptr;
        this->right = nullptr;
        this->parent = nullptr;
    }

    T getData() {
        return data;
    }

    int getAccess() {
        return access;
    }

    Node<T>* getLeft() {
        return left;
    }

    void setLeft(Node<T>* left) {
        this->left = left;
    }

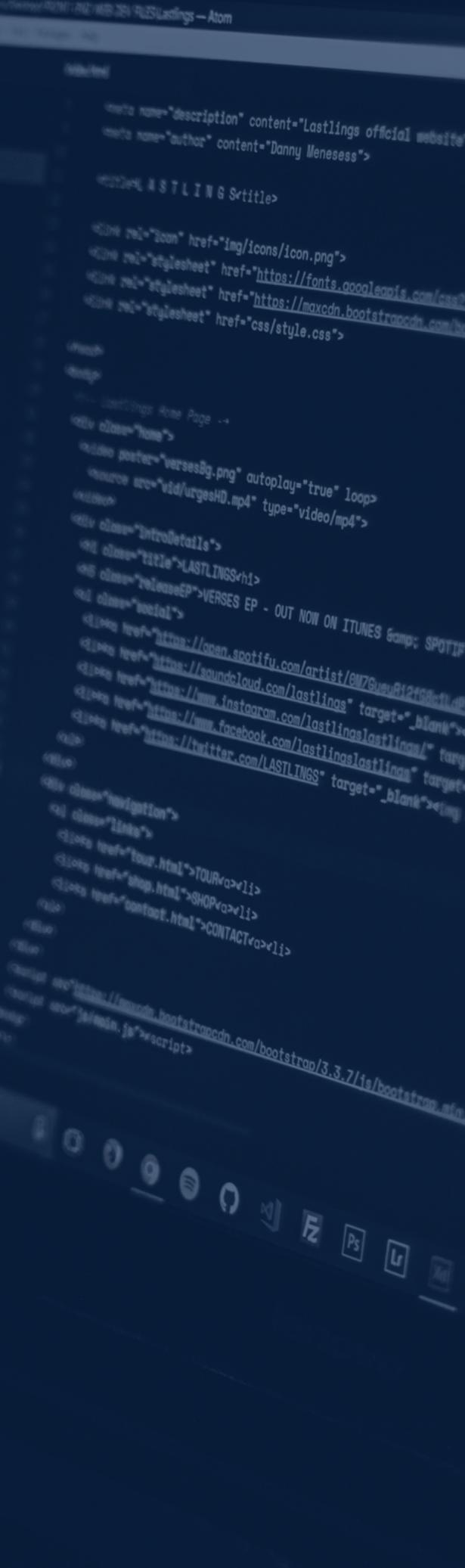
    Node<T>* getRight() {
        return right;
    }

    void setRight(Node<T>* right) {
        this->right = right;
    }

    Node<T>* getParent() {
        return parent;
    }

    void setParent(Node<T>* parent) {
        this->parent = parent;
    }

    void addAccess() {
        access++;
    }
};
```



```
template <class T>
class ArbolBiselado {
private:
    Node<T>* root;
    void rotarIzquierda(Node<T>* node) {
        Node<T>* y = node->getRight();
        node->setRight(y->getLeft());
        if (y->getLeft() != nullptr) {
            y->getLeft()->setParent(node);
        }
        y->setParent(node->getParent());
        if (node->getParent() == nullptr) {
            this->root = y;
        }
        else if (node == node->getParent()->getLeft()) {
            node->getParent()->setLeft(y);
        }
        else {
            node->getParent()->setRight(y);
        }
        y->setLeft(node);
        node->setParent(y);
    }
    void rotarDerecha(Node<T>* node) {
        Node<T>* y = node->getLeft();
        // y es igual al hijo izquierdo
        node->setLeft(y->getRight());
        // el izquierdo del nodo rotado apunta al derecho del hijo izquierdo, balanceo
        if (y->getRight() != nullptr) {
            y->getRight()->setParent(node);
        }
        // si se hizo el balanceo el progenitor es el nuevo padre
        y->setParent(node->getParent());
        // si es raiz actualizar raiz
        if (node->getParent() == nullptr) {
            this->root = y;
        }
        else if (node == node->getParent()->getRight()) {
            // el hijo derecho del padre del nodo es y
            node->getParent()->setRight(y);
        }
        else {
            // el hijo izquierdo del padre del nodo es y
            node->getParent()->setLeft(y);
        }
        // establece lado derecho
        y->setRight(node);
        // establece parent
        node->setParent(y);
    }
}
```

class ArbolBiselado

Esta clase cuenta con diferentes funciones como:

- rotarIzquierda (zig), el hijo izq del nodo se convierte en padre y el hijo der del nuevo padre se convierte en el hijo izq del padre anterior
Complejidad Tiempo O(1)
Complejidad Espacio O(1)
- rotarDerecha (zag), el hijo der del nodo se convierte en padre y el hijo izq del nuevo padre se convierte en el hijo der del padre anterior
Complejidad Tiempo O(1)
Complejidad Espacio O(1)

Estas funciones se utilizan para biselar nodos y tratar de mantener un equilibrio.

class ArbolBiselado

```
void biselar(Node<T>* node) {
    while (node->getParent()) {
        if (!node->getParent()->getParent()) {
            if (node == node->getParent()->getLeft()) {
                // zig
                rotarDerecha(node->getParent());
            }
            else {
                // zag
                rotarIzquierda(node->getParent());
            }
        }
        else if (node == node->getParent()->getLeft() && node->getParent() == node->getParent()->getLeft()) {
            // zig-zig
            rotarDerecha(node->getParent()->getParent());
            rotarDerecha(node->getParent());
        }
        else if (node == node->getParent()->getRight() && node->getParent() == node->getParent()->getRight()) {
            // zag-zag
            rotarIzquierda(node->getParent()->getParent());
            rotarIzquierda(node->getParent());
        }
        else if (node == node->getParent()->getRight() && node->getParent() == node->getParent()->getLeft()) {
            // zig-zag
            rotarIzquierda(node->getParent());
            rotarDerecha(node->getParent());
        }
        else {
            // zag-zig
            rotarDerecha(node->getParent());
            rotarIzquierda(node->getParent());
        }
    }
}
```

La función biselar que involucra las operaciones de rotacionIzquierda y rotacionDerecha, esta función la utilizamos para mover un nodo que se accedio recientemente hacia la raiz.

La función puede biselarlo de distintas maneras como: zig, zag, zig-zag, zag-zig, zig-zig- zag-zag.

Complejidad tiempo $O(\log n)$

Complejidad espacio $O(1)$

Métodos public ArbolBiselado

- Función search, la utilizamos para buscar un elemento en el árbol biselado y saber si ese elemento existe dentro de el.

Complejidad tiempo $O(\log n)$

Complejidad espacio $O(1)$

- Función Insert, la utilizamos para insertar un nuevo nodo en el arbol Biselado, lo añade de la misma forma que un BST pero cuando llega a su posición correcta, entonces el nodo se bisela hasta llegar a la raíz.

Complejidad tiempo $O(\log n)$

Complejidad Espacio $O(1)$

```
public:  
    ArbolBiselado() {  
        root = nullptr;  
    }  
    Node<T>* getRoot() {  
        return root;  
    }  
    //search if an element exists inside of the tree  
    bool search(T data) {  
  
        if (root->getData() == data) {  
            root->addAccess();  
            return true;  
        }  
        Node<T>* current = root;  
        while (current != nullptr) {  
            if (current->getData() == data) {  
                biselar(current);  
                current->addAccess();  
                return true;  
            }  
            else if (data < current->getData()) {  
                current = current->getLeft();  
            }  
            else {  
                current = current->getRight();  
            }  
        }  
        return false;  
    }  
    void insert(T key) {  
        Node<T>* node = new Node<T>(key);  
        Node<T>* y = nullptr;  
        Node<T>* x = this->root;  
        while (x != nullptr) {  
            y = x;  
            if (node->getData() < x->getData()) {  
                x = x->getLeft();  
            }  
            else if (node->getData() > x->getData()) {  
                x = x->getRight();  
            }  
            else {  
                x->addAccess();  
                biselar(x);  
                return;  
            }  
        }  
        // si nunca se asignó y AB vacío  
        node->setParent(y);  
        if (y == nullptr) {  
            root = node;  
        }  
        // Si es menor el valor a insertar es hijo izquierdo  
        else if (node->getData() < y->getData()) {  
            y->setLeft(node);  
        }  
        // Si es mayor el valor a insertar es hijo derecho  
        else {  
            y->setRight(node);  
        }  
        //Biselar el nodo hasta la raiz  
        biselar(node);  
        // tamanio++;  
    }  
}
```

Métodos public ArbolBiselado

Esta función se utiliza para leer las diferentes direcciones IP desde el archivo txt. y las inserta en el árbol Biselado utilizando la función insert que vimos anteriormente.

Complejidad Tiempo $O(n \log n)$

Complejidad Espacio $O(1)$

```
void fileToSplayTree(string fileName) {
    fstream bitacora;
    string line, ip1, ip2, ip3, ip4;
    long long ipGenerada;

    bitacora.open(fileName, ios::in);

    while (getline(bitacora, line, 'P')) {
        getline(bitacora, line, ' ');
        getline(bitacora, ip1, '.');
        getline(bitacora, ip2, '.');
        getline(bitacora, ip3, '.');
        getline(bitacora, ip4, '.');
        ipGenerada = ipToLong(ip1, ip2, ip3, ip4);
        getline(bitacora, line, '\n');
        insert(ipGenerada);
    }
    bitacora.close();
};
```

C L A S S B S T

Clase BinarySearchTree que cuenta con su constructor respectivo, un nodo que apunta hacia la raíz del árbol y diferentes funciones para poder ordenar los datos:

- search, función que verifica y busca si un elemento con datos tipo T se encuentra en el arbol.

Complejidad Tiempo $O(\log n)$

Complejidad Espacio $O(1)$

```
template <class T>
class BST {
private:
    Node<T>* root;
public:
    BST() {
        root = nullptr;
    }

    Node<T>* getRoot() {
        return root;
    }

    bool search(T data) {
        Node<T>* current = root;

        while (current != nullptr) {
            if (current->getData() == data) {
                return true;
            }
            else if (data < current->getData()) {
                current = current->getLeft();
            }
            else {
                current = current->getRight();
            }
        }
        return false;
    }
}
```

Class BST

```
void insert(T value) {
    Node<T>* current = root;
    if (current == nullptr) {
        root = new Node<T>(value);
        return;
    }
    while (current != nullptr) {
        if (value < current->getData()) {
            if (current->getLeft() == nullptr) {
                current->setLeft(new Node<T>(value));
                return;
            }
            else {
                current = current->getLeft();
            }
        }
        else if (value > current->getData()) {
            if (current->getRight() == nullptr) {
                current->setRight(new Node<T>(value));
                return;
            }
            else {
                current = current->getRight();
            }
        }
        else {
            return;
        }
    }
}

// metodo para cargar las ips al BST
void fileToBst(string fileName) {
    fstream bitacora;
    string line, ip1, ip2, ip3, ip4;
    long long ipGenerada;

    bitacora.open(fileName, ios::in);

    while (getline(bitacora, line, '\n')) {
        getline(bitacora, line, ' ');
        getline(bitacora, ip1, '.');
        getline(bitacora, ip2, '.');
        getline(bitacora, ip3, '.');
        getline(bitacora, ip4, ' ');
        ipGenerada = ipToInt(ip1, ip2, ip3, ip4);
        getline(bitacora, line, '\n');
        insert(ipGenerada);
    }

    bitacora.close();
}
```

- Función insert, se utiliza para ingresar un nuevo elemento de dato tipo T en el BST.
Complejidad Tiempo $O(\log n)$
Complejidad Espacio $O(n)$
- Función FileToBst, función que lee un archivo txt. en donde se encuentra el registro de las direcciones IP y construye el BST a partir de este archivo.
Complejidad Tiempo $O(n \log n)$
Complejidad Espacio $O(1)$

MAIN

- Abrir bitácora
- Crear un BST y un SplayTree
- Agregar datos en nodos del BST y SplayTree
- Importar información a un archivo .txt
- Función para verificar el tiempo que se tarda en encontrar las ips en cada árbol

```
int main() {
    clock_t time;
    BST<long long> bst;
    ArbolBiselado<long long> splayTree;
    ArbolBiselado<long long> splayTreeUnchanged;
    // Lectura de archivo
    bst.fileToBst("bitacoraelb.txt");
    splayTreeUnchanged.fileToSplayTree("bitacoraelb.txt");
    // Caso de prueba 1, buscar 100000000 veces la misma ip
    cout << "Caso prueba 1: 100000000 busquedas de la misma ip" << endl;
    //BST
    time = clock();
    for (int i = 0; i <= 100000000; i++) {
        bst.search(41243152151);
    }
    time = clock() - time;
    cout << "BST: " << (float)time/CLOCKS_PER_SEC << " segundos" << endl;
    splayTree = splayTreeUnchanged;
    //Splay Tree
    time = clock();
    for (int i = 0; i <= 100000000; i++) {
        splayTree.search(41243152151);
    }
    time = clock() - time;
    cout << "Splay Tree: " << (float)time / CLOCKS_PER_SEC << " segundos" << endl;
    // Caso de prueba 2, busqueda de 12006 ips ascendentes
    cout << "\nCaso prueba 2: busqueda de ips ascendentes" << endl;
    // BST
    timeTest(bst, "ipsAsc.txt", "BST: ");
    splayTree = splayTreeUnchanged;
    // Splay Tree
    timeTest(splayTree, "ipsAsc.txt", "Splay Tree: ");
    // Caso de prueba 3, busqueda de 12006 ips descendentes
    cout << "\nCaso prueba 3: busqueda de ips descendentes" << endl;
    // BST
    timeTest(bst, "ipsDesc.txt", "BST: ");
    splayTree = splayTreeUnchanged;
    // Splay Tree
    timeTest(splayTree, "ipsDesc.txt", "Splay Tree: ");
    // Caso de prueba 4, busqueda de ips, dos ips se repiten consecutivamente muchas veces, y las demás son aleatorias
    cout << "\nCaso prueba 4: busqueda de dos ips repetidas con ips aleatorias" << endl;
    // BST
    timeTest(bst, "ipsDosRepetidas.txt", "BST: ");
    splayTree = splayTreeUnchanged;
    // Splay Tree
    timeTest(splayTree, "ipsDosRepetidas.txt", "Splay Tree: ");
    return 0;
}
```

CASOS DE PRUEBA

Caso prueba 1: 100000000 busquedas de la misma ip

BST: 14.705 segundos

Splay Tree: 0.479 segundos

Caso prueba 2: busqueda de ips ascendentes

BST: 0.014 segundos

Splay Tree: 0.018 segundos

Caso prueba 3: busqueda de ips descendentes

BST: 0.015 segundos

Splay Tree: 0.016 segundos

Caso prueba 4: busqueda de dos ips repetidas con aleatorias

BST: 0.936 segundos

Splay Tree: 0.807 segundos

CASOS DE PRUEBA

Caso prueba 1: 100000000 busquedas de la misma ip

BST: 14.497 segundos

Splay Tree: 0.48 segundos

Caso prueba 2: busqueda de ips ascendentes

BST: 0.014 segundos

Splay Tree: 0.014 segundos

Caso prueba 3: busqueda de ips descendentes

BST: 0.018 segundos

Splay Tree: 0.014 segundos

Caso prueba 4: busqueda de dos ips repetidas con ips aleatorias

BST: 0.923 segundos

Splay Tree: 0.805 segundos

CASOS DE PRUEBA

Caso prueba 1: 100000000 busquedas de la misma ip

BST: 14.716 segundos

Splay Tree: 0.48 segundos

Caso prueba 2: busqueda de ips ascendentes

BST: 0.014 segundos

Splay Tree: 0.014 segundos

Caso prueba 3: busqueda de ips descendentes

BST: 0.017 segundos

Splay Tree: 0.015 segundos

Caso prueba 4: busqueda de dos ips repetidas con ips aleatorias

BST: 0.941 segundos

Splay Tree: 0.818 segundos

MUCHAS GRACIAS

ANDRÉS EMILIANO DE LA GARZA ROSALES A01384096
RODRIGO DE JESÚS MELÉNDEZ MOLINA A00831646
FIDEL MORALES BRIONES A01198630