Reflexión Act 3.4: Actividad Integral de BST

Desde mi punto de vista personal, creo que el uso de un árbol binario de búsqueda no queda para esta situación, puesto que tenemos varios valores repetidos y se rompe la propiedad de un BST. Además, hace que el acceso a estos valores repetidos sea difícil de lograr dentro del mismo BST, pues una vez ya insertados esos valores hay que tomar la decisión de ponerlos a la izquierda o la derecha. Esta decisión se tomó para nuestro caso con base a cuales datos serán insertados más adelante, si mayores o menores en promedio.

Un BST es muy eficiente para determinar organizar valores de manera rápida y eficiente. Pues la misma estructura de un BST, a la hora de hacer una inserción, organiza los elementos de una manera en la que compara qué valor es mayor o menor que otro.

Para esta situación donde se tienen valores repetidos, y había que accesar a los valores más grandes, repetidamente, se recomienda utilizar un max heap. Pues acceder a los valores más grandes del conjunto siempre tendrá una complejidad de $O(1)$ pues el elemento más grande siempre estará en la raíz del árbol, mientras que en un BST normal tiene una complejidad de $O(n)$ y su ubicación solo se determina por el valor de su número y por cómo está organizado el árbol. A pesar de que la propiedad de un BST se rompe en esta situación, la inserción sigue siendo $O(n)$, y la verificación de la altura del árbol es $O(h)$.

Para nuestro programa, decidimos optar por hacer un preprocesamiento de los datos y capturar cuantas veces se están repitiendo las IPs para almacenarlas en un vector de objetos de tipo entry. Estos objetos entry almacenan el número de accesos a la IP (int) y la IP misma en formato string. Seguido de esto, se insertan en el BST de manera secuencial como se encuentran en el vector siguiendo como mejor se puede las propiedades del mismo. Una vez el árbol está completo, se traversa el árbol en orden, comenzando con los números más grandes y guardándolos todos en otro vector. Una vez se encuentren todos los elementos del árbol en el vector, se imprimen los primeros 5 elementos de este para obtener así las primeras 5 IPs con el mayor número de accesos.

Esto se hizo de esta manera porque nos resultaba sencillo y al mismo tiempo seguíamos utilizando el BST en su totalidad para resolver el problema.

¿Cómo podemos determinar si una red está infectada o no?

En este caso se nos brinda ya una bitácora con los accesos que se hicieron a una serie de IPs, que técnicamente a esto se le conoce como logs. En este caso los logs arrojan un mensaje donde se especifica qué sucedió exactamente cuando se intentó hacerle un

ping a esa IP. Podemos determinar con la información que nos dan que, cuando el mensaje del log dice “Illegal user”, entonces ese ping realizado a esa hora en esa fecha se consideraría un intento de acceso malicioso.

Según mi experiencia, los pings a una IP son repetitivos, pues cada que se envía un paquete con datos a una dirección, se interactúa con la IP. Por lo que por la cantidad de pings que se realizan no se puede determinar si una red está infectada o no. Sin embargo, si se está intentando acceder a los distintos dispositivos dentro de la red y no se logra entrar porque se tienen las credenciales incorrectas por una cantidad de pings anormal, ahí podría considerarse que dentro de la red ya hay alguien, la red está ahora infectada, que intenta sin permiso entrar a algún dispositivo.

```
void BST::destroy(Node* node){

    // lo primero que se elimina son las hojas.
    // va por niveles
    if (node != nullptr){ C1
        destroy(node->getLeft()); C2
        destroy(node->getRight()); C3
        delete node; C4
    }

}
```

C_1	1	$T(n) = C_1(1) + C_2(1) + C_3(1) + C_4(1)$
C_2	1	$T(n) = 1[C_1 + C_2 + C_3 + C_4]$
C_3	1	$T(n) = O(1)$
C_4	1	

```
BST::BST(){
    root = nullptr; C1
}

BST::~BST(){
    destroy(root); C1
}
```

C_1	1	$T(n) = C_1(1)$

$T(n) = O(1)$

C_1	1	$T(n) = C_1(1)$

$T(n) = O(1)$

```
void fiveGreatest(Node* root){

    if( root != nullptr){C1

        fiveGreatest(root->getRight()); C2
        greatestIP.push_back(root->getData());C3
        greatestAccess.push_back(root->getNumAccesos());C4
        fiveGreatest(root->getLeft());C5
    }
}

void BST::print5greatest(){

    if( root != nullptr){ C1

        fiveGreatest(root->getRight()); C2
        greatestIP.push_back(root->getData()); C3
        greatestAccess.push_back(root->getNumAccesos()); C4

        fiveGreatest(root->getLeft());C5
    }

    for (int i = 0; i < 5; i++){ C6
        cout << i+1 << ". " <<"IP: " << greatestIP[i] << " " << "Numero Accesos:
        " << greatestAccess[i] << endl;  C7
    }

}
```

C_1	1	$T(n) = C_1(1) + C_2(1) + C_3(1) + C_4(1) + C_5(1)$
C_2	1	$T(n) = 1[C_1 + C_2 + C_3 + C_4 + C_5]$
C_3	1	$T(n) = O(1)$
C_4	1	
C_5	1	

C_1	1	$T(n) = C_1(1) + C_2(1) + C_3(1) + C_4(1) + C_5(1) +$
C_2	1	$C_6(n) + C_7(1)$
C_3	1	$T(n) = n\,C_6 + 1[C_1 + C_2 + C_3 + C_4 + C_5 + C_7]$
C_4	1	$T(n) = O(n)$
C_5	1	
C_6	n	
C_7	1	

```
int BST::height(){

    // devolvemos el valor de
    return bstHeight(root);

}
```

C_1	1	$T(n) = C_1(1)$
		$T(n) = 1\ C_1$
		$T(n) = O(1)$

```
int bstHeight(Node* root){

    if (root == nullptr){
        return 0;
    }

    int leftHeight = bstHeight(root->getLeft());
    int rightHeight = bstHeight(root->getRight());

    // sumar 1 al lado que tenga más altura
    return  leftHeight > rightHeight ? leftHeight + 1 : rightHeight + 1;
}
```

C_1	1	$T(n) = 2T(\frac{h}{2}) + 2$
C_2	1	$2^2[T(\frac{h}{2^2} + 2(2)]$
C_3	$\frac{1}{2}$	$2^k T(\frac{n}{2^k} + 2k)$
C_4	$\frac{1}{2}$	$\log n = k$
C_5	1	$2\log h + (\frac{h}{2\log h} + 2\log h)$
		$T(n) = O(h)$

```
void BST::insert(string data,int numAccesos){

    Node* current = root;
    // father es el papá del nodo hoja que queremos insertar.
    Node* father = nullptr;
    while(current != nullptr){

        // conforme vamos bajando, el padre va a ser actual
        father = current;
        // condicional para ver si se va a la derecha o izquierda
        if(current->getNumAccesos() > numAccesos){
          current = current->getLeft();
        }
        else{
          current = current->getRight();
        }

    }

    // si el arbol está vacío
    if (father == nullptr){
        root = new Node(data,numAccesos);
    } else{

        if (father->getNumAccesos() > numAccesos){

          father->setLeft(new Node(data,numAccesos));
        } else if (father->getNumAccesos() <= numAccesos){
          father->setRight(new Node(data,numAccesos));
        }
    }

}
```

C_1	1	$T(n) = C_1(1) + C_2(1) + C_3(\log n) + C_4(1) + C_5(1) + C_6(1) +$
C_2	1	$C_7(1) + C_8(1) + C_9(1) + C_{10}(1) + C_{11}(1) + C_{12}(1) + C_{13}(1)$
C_3	$\log n$	$T(n) = \log n\,(C_3 + 1)[C_1 + C_2 + C_4 + C_5 + C_6 + C_7 + C_8 + C_9 + C_{10}$
C_4	1	$+ C_{11} + C_{12} + C_{13}]$
C_5	1	$T(n) = O(\log n)$
C_6	1	
C_7	1	
C_8	1	
C_9	1	
C_{10}	1	
C_{11}	1	
C_{12}	1	
C_{13}	1	