César Alán Silva Ramos - A01252916

Act 3.1 - Operaciones avanzadas en un BST

Código en Github

Árbol utilizado dentro de los casos prueba

```
// Ejemplo de BST de 3 niveles:
//          420
//        /    \
//       69     6
//      / \    /  \
//     1   3  5   7
```

Casos Prueba:

Visit

- Preorder

```
Elije un tipo de recorrido (1-4): 1
Recorrido Preorder del BST: 420 69 1 3 6 5 7
```

- Inorder

```
Elije un tipo de recorrido (1-4): 2
Recorrido Inorder del BST: 1 69 3 420 5 6 7
```

- Postorder

```
Elije un tipo de recorrido (1-4): 3
Recorrido Postorder del BST: 1 3 69 5 7 6 420
```

- Level by level

```
Elije un tipo de recorrido (1-4): 4
Recorrido por Niveles del BST: 420 69 6 1 3 5 7
```

Height

```
La altura del BST es: 3
```

Ancestors

```
Datos de los nodos ancestros de 7: 6 420
```

What Level Am I

```
El nivel de 7 es: 3
```

Complejidades Computacionales

Funciones de Visit

```cpp
void preorderTraversal(Node* root) { // En el primer llamdo root representa a la raiz, en los demás representa el nodo actual
if (root == nullptr) {
return;
}
std::cout << root->data << " "; // Procesar el nodo actual
preorderTraversal(root->left); // Recorrer el subárbol izquierdo
preorderTraversal(root->right); // Recorrer el subárbol derecho una vez que el
}

// Función para realizar un recorrido inorder en el BST
void inorderTraversal(Node* root) {
if (root == nullptr) {
return;
}
inorderTraversal(root->left); // Recorrer el subárbol izquierdo
std::cout << root->data << " "; // Procesar el nodo actual
inorderTraversal(root->right); // Recorrer el subárbol derecho
}
```

```
// Función para realizar un recorrido postorder en el BST
void postorderTraversal(Node* root) {
if (root == nullptr) {
return;
}
postorderTraversal(root->left); // Recorrer el subárbol izquierdo
postorderTraversal(root->right); // Recorrer el subárbol derecho
std::cout << root->data << " "; // Procesar el nodo actual
}

// Función para realizar un recorrido por niveles en el BST
void levelOrderTraversal(Node* root) {
if (root == nullptr) {
return;
}
std::queue<Node*> q;
q.push(root);
while (!q.empty()) {
Node* current = q.front();
std::cout << current->data << " ";
q.pop();
if (current->left) {
q.push(current->left);
}
if (current->right) {
q.push(current->right);
}
}
}
```

En cuanto a las complejidades temporales de las funciones de recorridos en los árboles binarios de búsqueda, todas presentan una complejidad lineal de O(n), donde "n" representa el número total de nodos en el árbol. Esto se debe a que en cada una de estas funciones, cada nodo se visita una sola vez, y las operaciones realizadas en cada visita tienen una complejidad constante. Las

diferencias entre estos recorridos se encuentran en el orden en que se visitan los nodos y sus aplicaciones específicas. Sin embargo no difieren en cuanto a complejidades temporales.

El recorrido en preorden comienza visitando la raíz antes de explorar los subárboles de la izquierda y derecha. El recorrido en inorden visita los nodos en orden ascendente empezando de la izquierda. El recorrido en postorden visita la raíz después de los subárboles izquierdos y derechos, es útil para liberar la memoria de un árbol. Por último, el recorrido por niveles visita el árbol por niveles, de arriba hacia abajo y de izquierda a derecha, siendo eficaz para buscar los nodos más cercanos a la raíz.

Height

```
// Obtener altura BST - Función recursiva
int getHeight(Node* root) {
if (root == nullptr) {
return 0; // Si el root apunta a null la altura es 0
} else {
int leftHeight = getHeight(root->left);// Recursión hasta alcanzar un nodo hoja por la parte izquierda
int rightHeight = getHeight(root->right); // Recursión hasta alcanzar nodo hoja parte deerecha
return 1 + std::max(leftHeight, rightHeight); // el +1 represenat cada nivel alcanzado de la parte derecha e izquierda
}
}
```

En esta función, se observa una complejidad temporal de O(n), donde "n" representa el número total de nodos presentes en el árbol. La función recorre el árbol desde la raíz hasta las hojas de forma recursiva. En cada llamada recursiva, se invoca la misma función para el subárbol izquierdo y el subárbol derecho. Esto permite visitar todo el árbol de manera lineal, en función de la cantidad de nodos que existen en el árbol. En otras palabras, cada nodo se visita una vez, lo que conduce a un tiempo de ejecución proporcional al número de nodos en el árbol.

Ancestors

```
// Función para obetener ancestros de un nodo
bool findAncestors(Node* root, int target, std::vector<int>& ancestors) {
if (root == nullptr) {
return false;
}
if (root->data == target) {
return true;
}

if (findAncestors(root->left, target, ancestors) || findAncestors(root->right, target,
ancestors)) {
ancestors.push_back(root->data);
return true;
}

return false;
}
```

En esta función, se busca de manera recursiva un nodo que contenga el valor "target" para determinar sus ancestros. Por lo tanto, la complejidad temporal de esta función es O(n), donde "n" representa la cantidad de nodos en el árbol. En el peor escenario, la función debe recorrer todo el árbol para encontrar el nodo que contiene el valor "target". Durante cada visita a un nodo, se realizan operaciones constantes, como comparaciones entre el valor del nodo y el "target". Esto significa que el tiempo de ejecución de la función es proporcional al número de nodos en el árbol.

What Level Am I

```
int findLevel(Node* root, int target, int level) {
if (root == nullptr) {
return -1; // El dato no se encuentra en el árbol
}
```

```
if (root->data == target) { // Se checa si el dato se encuentra en root
return level; // Se encontró el dato en este nivel
}

// Realiza una búsqueda en los subárboles izquierdo y derecho
int leftLevel = findLevel(root->left, target, level + 1); // busqueda recursiva
if (leftLevel != -1) {
return leftLevel;
}

int rightLevel = findLevel(root->right, target, level + 1);
return rightLevel;
}
```

Esta función tiene como objetivo determinar el nivel en el que se encuentra un dato en un árbol. Similar a la función previamente mencionada que buscaba los ancestros de un dato, en este caso, la diferencia está en que estamos interesados en conocer el nivel específico del dato. La función realiza una búsqueda recursiva en todo el árbol hasta localizar el nodo que contiene el dato "target". Esto da como resultado una complejidad temporal de O(n), donde "n" representa la cantidad de nodos en el árbol. En el peor de los casos, la función tendría que visitar todos los nodos del árbol para encontrar el objetivo deseado y determinar su nivel por medio de operaciones constantes. La complejidad es, por lo tanto, lineal en relación al número de nodos en el árbol.