

### Actividad 3.1 – Operaciones avanzadas en un BST

#### Visit $O(1)$

```
void BST::visit(int order){  
  
    switch (order)  
    {  
    case 1:  
  
        preOrder(root);  
        break;  
  
    case 2:  
  
        inOrder(root);  
        break;  
  
    case 3:  
  
        postOrder(root);  
        break;  
  
    case 4:  
  
        byLevel(root);  
        break;  
  
    default:  
        break;  
    }  
}
```

## Preorder

```
void preOrder(Node* root){
    if (root != nullptr){
        std::cout << root->getData() << " ";
        preOrder(root->getLeft());
        preOrder(root->getRight());
    }
}
```

### Tarea 3.1 Analisis de complejidad

preOrder (root)  $T(n)$

if root != null

(1) 1

\* Siempre  
entraron la  
mitad de los  
nodos

print root->getData

(2) 1

preorder (root->getLeft())

(3)  $T(\frac{n}{2})$

preorder (root->getRight())

(4)  $T(\frac{n}{2})$

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 1 + 1$$

$$T(n) = 2 \left[ T(\frac{n}{2}) + 2 \right]$$

$$\frac{n}{2^k} = 1$$

$$2 \left[ 2 \left( \frac{n}{2} \right) + 2(2) \right]$$

$$\log n = k$$

$$2^2 \left[ \frac{n}{2^2} + 2(2) \right]$$

$$2^{\log_2 n} T\left(\frac{n}{2^{\log_2 n}}\right) + 2(\log_2 n)$$

$$2^2 \left[ 2 \left( T\left(\frac{n}{2^3}\right) + 2(3) \right) \right]$$

$$n(1) + 2(\log_2 n)$$

$$\left[ 2^k T\left(\frac{n}{2^k}\right) + 2k \right]$$

$$\boxed{O(n)}$$

## Inorder

```
void inOrder(Node* root){  
    if( root != nullptr){  
        inOrder(root->getLeft());  
        std::cout << root->getData() << " ";  
        inOrder(root->getRight());  
    }  
}
```

inOrder(root)  $T(n)$

if root  $\neq$  null (1)

inorder(root->getLeft) (2)  $T(\frac{n}{2})$

print root->getData (3) 1

inorder(root->getRight) (4)  $T(\frac{n}{2})$

$$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 1 + 1$$

$$2[T(\frac{n}{2}) + 2]$$

$$2^k T(\frac{n}{2^k}) + 2k$$

$$2^{\log n} T(\frac{n}{2^{\log n}}) + 2 \log n$$

$$n \left( \frac{n}{n} \right) + 2 \log n$$

$$T(n) = n + 2 \log n$$

$$T(n) = \boxed{O(n)}$$

## Postorder

```
void postOrder(Node* root){  
    if (root != nullptr){  
        postOrder(root->getLeft());  
        postOrder(root->getRight());  
        std::cout << root->getData() << " ";  
    }  
}
```

postOrder(root)  $c_1 T(n)$

if root  $\neq$  null  $c_2 1$

postOrder(root->left)  $c_3 T(\frac{n}{2})$

postOrder(root->right)  $c_4 T(\frac{n}{2})$

print root->getData  $c_5 1$

$$T(n) = 2T\left(\frac{n}{2}\right) + 2$$

$$n = 2^k$$

$$\log n = k$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + 2k$$

$$2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + 2 \log n$$

$$n \left(\frac{n}{n}\right) + 2 \log n$$

$$T(n) = n + 2 \log n$$

$$T(n) = \boxed{O(n)}$$

## Level by level

```
void byLevel(Node* root){  
  
    // Guardamos la altura del árbol  
    int height = bstHeight(root);  
  
    for (int i = 0; i < height; i++){  
        levelNodes(root, 0, i);  
        std::cout << std::endl;  
    }  
}
```

void byLevel(root)

height = bstHeight(root)  $C_1$

\* para esta función el método vale 1 por la asignación

for (int i=0; i < height; i++)  $C_2 n$

levelNodes(root, 0, i)  $C_3 n(n-1)$

print " "  $C_4 n-1$

\* visitamos todos los nodos  $n$  veces.

$$T(n) = C_1 + C_2 n + C_3 (n^2 - n)$$

Debe ser  $O(n^2)$

$$T(n) = C_1 + C_2 n + C_3 n^2 - C_3 n$$

$$T(n) = C_1 + n(C_2 - C_3) + C_3 n^2$$

$$T(n) = a + nb + cn^2 = \boxed{O(n^2)}$$



## Height

```
/* Función que devuelve la altura de un BST*/
int BST::height(){

    // devolvemos el valor de altura que encontramos con la función.
    return bstHeight(root);

}
```

```
/* Función escondida que encuentra la altura de un arbol recursivamente.*/
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;

}
```

bstHeight(root)      T(n)

if (root == null)    C<sub>1</sub> 1

return 0    C<sub>2</sub> 1

leftHeight = bstHeight(root->getLeft())    C<sub>3</sub> T( $\frac{n}{2}$ )

rightHeight = bstHeight(root->getRight())    C<sub>4</sub> T( $\frac{n}{2}$ )

return leftHeight > rightHeight ? leftHeight + 1 : rightHeight + 1    C<sub>5</sub> 1

$T(n) = 2T(\frac{n}{2}) + 2$

$2^2 [ T(\frac{n}{2^2}) + 2(2) ]$

$2^k T(\frac{n}{2^k}) + 2k$

$\log n = k$

$2^{\log n} T(\frac{n}{2^{\log n}} + 2\log n)$

$O(n)$

\* Visitamos todos los nodos 1 vez.

## Ancestors

```
/* Función que visita los ancestros de un elemento del árbol.*/
void BST::ancestors(int data){

    // si no se encuentra ese elemento en el árbol no tiene ancestros.
    if (search(data) == false){
        std::cout << "Ese elemento no existe en el árbol. No hay ancestros.\n";
        return;
    }

    Node *current = root;

    // si el valor a revisar sus ancestros es la raíz, entonces imprimimos mensaje.
    if (root->getData() == data){
        std::cout << "Este nodo no tiene ancestros. Es la raíz." << std::endl;
    }

    while(current->getData() != data){

        std::cout << current->getData() << " ";

        // operador ternario -> (CONDICIÓN) ? ( TRUE que haces si es cierto) : ( FALSE que haces si es falso)
        // el pivote se mueve a la izquierda o a la derecha
        current = (data < current->getData()) ? current->getLeft() : current->getRight();

    }
}
```

ancestors (data)

1 if (search(data) == false)  $C_1$  1  
return

1 Node current = root  $C_2$  1  
if root->data == data  $C_3$  1  
print es raíz  
return

while (current->getData() != data)  $C_4$   $\sum_{i=1}^n$   
print current->getData()  $C_5$   $\sum_{i=1}^n$   
current = data < current->getData() ? current->getLeft() : current->getRight();  
 $C_6$   $\sum_{i=1}^n$

$T(n) = C_1 + C_2 + C_3 + C_4 \sum_{i=1}^n + C_5 \sum_{i=1}^n + C_6 \sum_{i=1}^n$

$T(n) = C_1 + C_2 + C_3 + \sum_{i=1}^n (C_4 + C_5 + C_6) = C_4 + C_5 + C_6$

$T(n) = a + \sum_{i=1}^n (b) = O(n)$

## WhatlevelAml

```
/* Función para encontrar en qué nivel se encuentra un elemento del árbol */
int BST::whatlevelamI(int data){

    // El nodo raíz se ubica en el nivel 0.
    int nivel = 0;

    // Si el dato no se encuentra en el árbol
    if (search(data) == false){
        return -1;
    }

    Node *current = root;

    while(current->getData() != data){

        // operador ternario -> (CONDICIÓN) ? ( TRUE que haces si es cierto) : ( FALSE que haces si es falso)
        // el pivote se mueve a la izquierda o a la derecha
        current = (data < current->getData()) ? current->getLeft() : current->getRight();

        nivel = nivel + 1;
    }

    return nivel;
}
```

- whatLevelamI (data)

nivel = 0  $C_1$  1

if ( search(data) == false )  $C_2$  1

return -1

Node current = root  $C_3$  1

while current->Data != data  $C_4$

current = data < current->getData ? current->getLeft : current->getRight

$C_6$  nivel++  $i+j-1$   $C_5$   $i+j-1$

$$T(n) = C_1 + C_2 + C_3 + C_4 i+j + C_5 i+j-1 + C_6 i+j-1$$

$$T(n) = (C_1 + C_2 + C_3 - C_5 - C_6) + i+j (C_4 + C_5 + C_6)$$

$$a + i+j(b) = \boxed{O(n)}$$