



# Tecnológico de Monterrey

---

Documentación de código BST

---

## **Programming of data structures**

Inés Alejandro Garcia Mosqueda A00834571

Profesores: Luis Ricardo Peña Llamas  
Jorge Gonzalez

17/10/2022

Se hace la implementación de un árbol binario, donde al instanciar la clase BST el cual contiene una clase llamada nodo, el cual contiene los accesos hacia otros nodos (left, right)

```
class Node{
public:
    int data;
    Node *left;
    Node *right;
    Node();
    Node(int&);
    Node(int&, Node*, Node*);
    int getData();
    void setData(int&);
    ~Node();
};
```

Esta clase nodo solo tiene métodos para obtener y sobrescribir la información interna del Nodo

Por otro lado, tenemos la clase BST el cual contiene un atributo de tipo puntero a un Nodo, el cual en la implementación es la cabeza del árbol binario

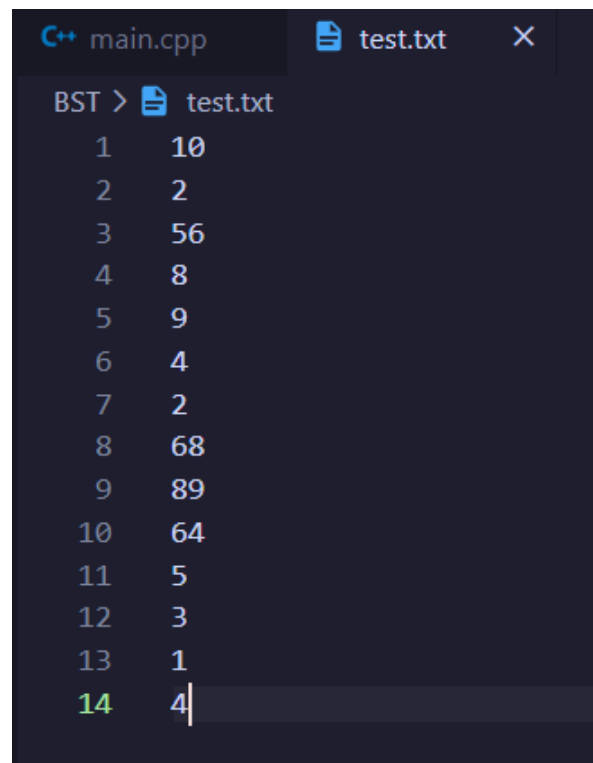
```
class BST{
    Node* head;
private:
    void insertNode(Node*&, int&); //Insercion de nodo en un nodo // jala
    void deleteNode(Node*&, int&); //Algoritmo de eliminacion de un nodo //
    void PreOrder(Node*&); //Ordenamiento PreOrder // jala
    void PostOrder(Node*&); //Ordenamiento PostOrder // jala
    void InOrder(Node*&); //Ordenamiento InOrder // jala
    void SubstituteToMin(Node*&, Node*&);
public:
    BST(){ Node *head = NULL; }; // jala
    BST(int data){ head = new Node(data); }; // jala
    ~BST(){ deleteTree(head); }; // jala
    void fillBST(std::string); // jala
    void searchNode(int&); //Busqueda de un nodo dentro del arbol
    void insert(int&); //Jala
    void InOrder(); // Jala
    void PreOrder(); // Jala
    void PostOrder(); // Jala
    void BFT(); // Jala
    void deleteNode(int&); //Jala
    void deleteTree(Node*&); //Jala
    void visit();
    int height();
    void ancestors();
    int whatLevelAmI();
};
```

Dentro de la estructura se crearon métodos para insertar elemento en el árbol, borrar elemento del árbol, buscar elemento en el árbol, acceder a los elementos del árbol en diferentes secuencias como lo es PreOrder, PostOrder, InOrder, level by level (BFT) estas con el fin de aprovechar sus características para nuestra conveniencia.

Además se agregaron funciones extra para consultar otro tipo de datos, como lo es la función visit para elegir la opción de secuencia a imprimir, height para saber la altura del árbol, ancestors, el cual nos indica los padres del dato que le pongamos y la función whatLevelAmI para saber la altura en la que se encuentra el dato dentro del árbol

Para demostración de esto también se implementó un método de lectura de un archivo txt con los datos del árbol (en este caso estos datos fueron aleatorios para asegurar el funcionamiento de este

### Lista de prueba



```
C++ main.cpp test.txt X
BST > test.txt
1 10
2 2
3 56
4 8
5 9
6 4
7 2
8 68
9 89
10 64
11 5
12 3
13 1
14 4
```

En la prueba se hará la lectura y la implementación de cada método requerido en la asignación

Todas las funciones fueron diseñadas recursivas por cuestiones de simplicidad

Función inOrder, PostOrder y PreOrder son de la misma complejidad debido a que es prácticamente el mismo código a diferencia de la acción requerida en la implementación

Haciéndolas de **complejidad  $O(n)$**  ya que se requiere recorrer toda la estructura

Código respecto a los tres tipos de recorridos en el árbol (PReOrder, InOrder, PostOrder)

```
void BST::PostOrder(Node* &nodo){  
    if (nodo != NULL)  
    {  
        PostOrder(nodo->left);  
        PostOrder(nodo->right);  
        std::cout<< " " << nodo->data;  
    }  
}
```

```
void BST::PreOrder(Node* &nodo){  
    if (nodo != NULL)  
    {  
        std::cout<< " " << nodo->data;  
        PreOrder(nodo->left);  
        PreOrder(nodo->right);  
    }  
}
```

```
void BST::InOrder(Node* &nodo){  
    if(nodo != NULL)  
    {  
        InOrder(nodo->left);  
        std::cout << " " <<nodo->data;  
        InOrder(nodo->right);  
    }  
}
```

El recorrido del árbol respecto a la altura se realiza de igual manera (recursiva), y guardando los datos del recorrido en otra estructura de datos, en este caso una cola (queue) para después imprimir los datos recopilados y mostrarlos al usuario

La complejidad para este código es de  $O(n)$  debido a que se requiere hacer todo el recorrido de la estructura

```
void BST::BFT(){
    if (head != NULL)
    {
        std::queue<Node*> elements;
        elements.push(head);
        Node* auxNode;
        while (!elements.empty())
        {
            elements.push(NULL);
            auxNode = elements.front();
            while (auxNode != NULL)
            {
                std::cout<<auxNode->data<<" ";
                if (auxNode->left != NULL)
                {
                    elements.push(auxNode->left);
                }
                if (auxNode->right != NULL)
                {
                    elements.push(auxNode->right);
                }
                elements.pop();
                auxNode = elements.front();
            }
            elements.pop();
            std::cout<<std::endl;
        }
    }
}
```

Respecto a la función de ancestros y nivel del dato ocurre lo mismo, ya que a partir de un dato se tiene que hacer la búsqueda de dicho dato, pero con diferentes datos de retorno

Al ser una búsqueda binaria la que se realiza, la función es de **complejidad  $O(\log n)$**

Finalmente, la función encargada de encontrar la altura del árbol es de **complejidad  $O(n)$** , ya que en la implementación se tiene que recorrer todo el árbol y regresar el valor máximo entre alturas encontradas

#### Resultado de caso de prueba

```
2 Repetido
4 Repetido

Secuencia InOrder 1 2 3 4 5 8 9 10 56 64 68 89
Secuencia PostOrder 1 3 5 4 9 8 2 64 89 68 56 10
Secuencia PreOrder 10 2 1 8 4 3 5 9 56 68 64 89
Secuencia BFT
10
2 56
1 8 68
4 9 64 89
3 5

Evaluacion del numero 8
Ancestros: 10 2
Nivel de este numero: 3
Altura: 4 niveles
```