



# Tecnológico de Monterrey

---

Documentación de código Avl trees

---

## **Programming of data structures**

Inés Alejandro Garcia Mosqueda A00834571

Profesores: Luis Ricardo Peña Llamas  
Jorge Gonzalez

23/10/2022

Como unidad básica de la estructura de dato se implemento una clase llamada nodo, con la misma configuración que el árbol binario, el cual contiene punteros a otros nodos dependiendo si son mayores o menores los nodos en la estructura

```
class Node{
public:
    int data;
    int factor;
    Node* left;
    Node* right;
    Node(){
        this->data = 0;
        this->factor = 0;
        this->left = NULL;
        this->right = NULL;
    };
    Node(int data){
        this->data = data;
        this->factor = 0;
        this->left = NULL;
        this->right = NULL;
    };
    Node(int data, Node* Left, Node* right){
        this->data = data;
        this->left = Left;
        this->right = right;
        this->factor = 0;
        if (Left != NULL)
            this->factor -= 1;
        if(right != NULL)
            this->factor += 1;
    };
    ~Node(){
        delete left;
        delete right;
    }; //Destructor de Nodo
    void setData(int data){
        this->data = data;
    };
    int getData(){
        return this->data;
    };
};
```

Para la implementación del árbol avl se debe tener en cuenta el funcionamiento de autobalance, ya que siempre queremos tener sub arboles completos y a la vez que el árbol principal sea completo y balanceado. Para esto utilizamos un factor por nodo, el cual será resultado de la resta entre la altura máxima del subárbol izquierdo por la altura máxima del subárbol derecho, por lo que el procedimiento para implementar el árbol avl será ir registrando los factores de cada nodo dependiendo de la altura máxima para poder decidir que combinación de rotaciones ejecturar, ya que existen cuatro posibles rotaciones (left rotation, right rotation, left right rotation y right left rotation), para esto se implementaron los métodos de cada rotación dentro de la clase AvlTree

```

46 class AVLTree{
47     Node* head;
48     int len;
49     private:
50         void insertNode(Node*&, int);
51         void deleteNode(Node*&, int&);
52         void PreOrder(Node*);
53         void InOrder(Node*);
54         void PostOrder(Node*);
55         void SubstituteToMin(Node*&, Node*&);
56         int updateHeight(Node*);
57         void updateHeight(){ updateHeight(head); };
58         void rightRotation(Node*&);
59         void leftRotation(Node*&);
60         void leftRightRotation(Node*&);
61         void rightLeftRotation(Node*&);
62         void balance(Node*&);
63         void find(Node*, int, bool&);
64
65     public:
66         AVLTree(){
67             head = NULL;
68             len = 0;
69         };
70         void insertNode(int data){ insertNode(head, data); };
71         void deleteNode(int data){ deleteNode(head, data); };
72         void PreOrder(){ PreOrder(head); };
73         void InOrder(){ InOrder(head); };
74         void PostOrder(){ PostOrder(head); };
75         void BFT();
76         void fillAVL(string);
77         bool find(int);
78         void print();
79         int size(){ return len; };
80     };
81
82

```

## Insert

El método de inserción de nodo (insertNode(int);) tiene una complejidad de  $O(\log n)$  ya que para la inserción se hace una búsqueda binaria del espacio correspondiente a el dato por insertar y debido a que el árbol siempre estará balanceado el peor de los casos es  $O(\log n)$

```

void AVLTree::insertNode(Node* &node, int data){
    if (node == NULL){
        Node* newNode = new Node(data);
        node = newNode;
        len++;
        updateHeight();
    }
    else{
        if (data > node->getData())
            insertNode(node->right, data);
        else if (data < node->getData())
            insertNode(node->left, data);
        else
            cout<< "\n-->" << data << " Dato repetido";
            balance(node);
    }
}

```

Dentro de la recursividad de la inserción del nodo se ejecuta una función llamada balance, la cual es fundamental para la consistencia del árbol, ya que este define las rotaciones necesarias para cada subárbol recorrido durante la inserción del dato, siendo esta función complejidad  $O(\log n)$  ya que para esta implementación se tuvo que implementar internamente la función que actualiza los factores de cada nodo en la recursividad

```

269 void AvlTree::balance(Node* &node){
270     if (node->factor > 1) {
271         if (node->left->factor > 0)
272             rightRotation(node);
273         //Right rotation
274     else
275         leftRightRotation(node);
276         //left right rotation
277         updateHeight(node);
278     }
279     else if (node->factor < -1) {
280         if (node->right->factor > 0)
281             rightLeftRotation(node);
282         //Right left rotation
283     else
284         leftRotation(node);
285         //left rotation
286         updateHeight(node);
287     }
288 }

```

Las cuatro funciones de rotaciones existentes son las siguientes y son consideradas complejidad  $O(1)$  ya que no contienen ninguna iteración interna

<pre> 226 void AvlTree::rightRotation(Node* &amp;node) { 227     Node* auxLeft; 228     Node* auxRightAux; 229     auxLeft = node-&gt;left; 230     auxRightAux = auxLeft-&gt;right; 231     auxLeft-&gt;right = node; 232     auxLeft-&gt;right-&gt;left = auxRightAux; 233     node = auxLeft; 234     //cout&lt;&lt;"RightRotation\n"; 235 } 236 void AvlTree::LeftRotation(Node* &amp;node) { 237     Node* auxRight; 238     Node* auxLeftAux; 239     auxRight = node-&gt;right; 240     auxLeftAux = auxRight-&gt;left; 241     auxRight-&gt;left = node; 242     auxRight-&gt;left-&gt;right = auxLeftAux; 243     node = auxRight; 244     //cout&lt;&lt;"LeftRotation\n"; 245 } </pre>	<pre> 246 void AvlTree::LeftRightRotation(Node* &amp;node) { 247     Node* auxLeft; 248     Node* auxRightAux; 249     auxLeft = node-&gt;left; 250     auxRightAux = auxLeft-&gt;right; 251     auxLeft-&gt;right = auxRightAux-&gt;left; 252     auxRightAux-&gt;left = auxLeft; 253     node-&gt;left = auxRightAux; 254     //cout&lt;&lt;"LeftRightRotation\n"; 255     rightRotation(node); 256 } 257 void AvlTree::rightLeftRotation(Node* &amp;node) { 258     Node* auxRight; 259     Node* auxLeftAux; 260     auxRight = node-&gt;right; 261     auxLeftAux = auxRight-&gt;left; 262     auxRight-&gt;left = auxLeftAux-&gt;right; 263     auxLeftAux-&gt;right = auxRight; 264     node-&gt;right = auxLeftAux; 265     //cout&lt;&lt;"RightLeftRotation\n"; 266     leftRotation(node); 267 } </pre>
--	---

## Delete

Respecto a la función delete (deleteNode(int)); se hace una búsqueda binaria del dato por borrar y al terminar de borrarlos asegurarse que se este balanceado. En el peor caso se obtiene una complejidad  $O(\log n)$  ya que la integridad del árbol no se modifica puesto a que se vuelve a balancear

```
101 void AvlTree::deleteNode(Node* &node, int& data){
102     if (node != NULL)
103     {
104         if (data < node->data){
105             deleteNode(node->left, data);
106         }
107         else if(data > node->data){
108             deleteNode(node->right, data);
109         }
110         else
111         {
112             Node* auxNodo = node;
113
114             if (auxNodo->right == NULL){
115                 node = auxNodo->left;
116             }
117             if (auxNodo->left == NULL){
118                 node = auxNodo->right;
119             }
120             if (auxNodo->left != NULL && auxNodo->right != NULL )
121                 SubstituteToMin(node->right, auxNodo);
122             std::cout<<"Se borro correctamente";
123             delete auxNodo;
124             len--;
125             updateHeight(head);
126         }
127         if (node!=NULL)
128             balance(node);
129     }
130 }
131
```

## Size

La función size (size()) se ejecuta en complejidad constante ya que internamente en la estructura se va registrando la cantidad de datos que se van insertando o eliminando de la estructura, por lo que solo se retorna el valor guardado en la variable "len" de la clase AvlTree

```
79     void print();
80     int size(){ return len; };
81 };
82
```

## Print

La función print se considera complejidad  $O(n)$  ya que se requiere el recorrido por todo el árbol para poder imprimir todos los datos del árbol

Para esta implementación del print se utilizo el recorrido InOrder y BFT para poder visualizar los datos en forma ascendente y además visualizar el árbol por niveles y estar seguros de la integridad del árbol

```

307 void AvlTree::print(){
308     cout<<"\n\nDatos del arbol en orden ascendente"<<endl;
309     InOrder();
310     cout<<"\n\nRepresentacion por niveles"<<endl;
311     BFT();
312     cout<<endl;
313 }

```

## Find

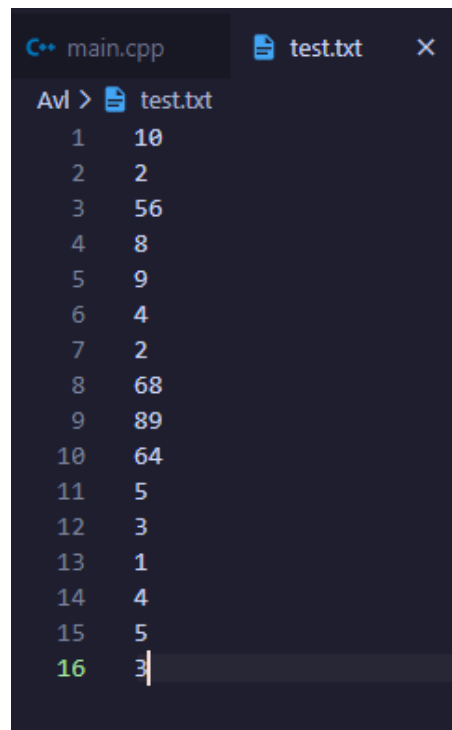
Finalmente se tiene la función encargada de la búsqueda de un dato dentro de la estructura, obteniendo complejidad  $O(\log n)$  ya que nuevamente se implementa una búsqueda binaria y aprovechar las propiedades de esta estructura de datos

```

290 void AvlTree::find(Node* node, int data, bool &search){
291     if(node != NULL){
292         if (data > node->data)
293             find(node->right, data, search);
294         else if (data < node->data)
295             find(node->left, data, search);
296         else
297             search = true;
298     }
299 }
300
301 bool AvlTree::find(int data){
302     bool resSearch = false;
303     find(head, data, resSearch);
304     return resSearch;
305 }
306

```

## Caso de prueba



```
main.cpp test.txt X
Avl > 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
15 5
16 3
```

Se incluye la lectura de un archivo txt llamado test.txt, para dar entrada a 16 datos aleatorios a la estructura Avl

Para la prueba se incluye la siguiente función main, la cual pretende probar las funciones implementadas

```
315 int main(){
316     AvlTree avl;
317     avl.fillAvl("test.txt");
318     cout<<"\n\nEn el arbol hay: "<< avl.size()<< " Elementos" <<endl;
319     avl.print();
320     cout <<"Busqueda numero 10: "<<avl.find(10)<<endl;
321     cout<<"\nBorrando 4"<<endl;
322     avl.deleteNode(4);
323     cout<<"\nBorrando 10"<<endl;
324     avl.deleteNode(10);
325     cout<<"\n\nEn el arbol hay: "<< avl.size()<< " Elementos" <<endl;
326     avl.print();
327     cout <<"Busqueda numero 10: "<<avl.find(10)<<endl;
328     return 0;
329 }
```

Obteniendo el siguiente resultado

```
-->2 Dato repetido
-->4 Dato repetido
-->5 Dato repetido
-->3 Dato repetido

En el arbol hay: 12 Elementos

Datos del arbol en orden ascendente
1, 2, 3, 4, 5, 8, 9, 10, 56, 64, 68, 89,

Representacion por niveles
9
4 56
2 8 10 68
1 3 5 64 89

Busqueda numero 10: 1

Borrando 4
Se borro correctamente
Borrando 10
Se borro correctamente

En el arbol hay: 10 Elementos

Datos del arbol en orden ascendente
1, 2, 3, 5, 8, 9, 56, 64, 68, 89,

Representacion por niveles
9
5 68
2 8 56 89
1 3 64

Busqueda numero 10: 0
PS C:\Users\nessy\Desktop\entregaExpress\Av1>
```

Siendo este el resultado esperado para nuestros casos de prueba