Práctica de Estudio 8: **Arboles. Parte II**
Estructura de Datos y Algoritmos Grupo 01
Facultad de Ingeniería
Departamento de Computación

	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Jesus Cruz Navarro
Profesor: _____

Estructura de Datos y Algoritmos II
Asianatura: _____

01
Grupo: _____

08
No de Práctica(s): _____

Diego Santiago Gutierrez
Integrante(s): _____

No. de Equipo de cómputo _____

No. de Lista o Brigada: _____

Tercer Semestre
Semestre: _____

24/11/2020
Fecha de entrega: _____

Observaciones: _____

_____

CALIFICACIÓN: _____

REQUISITOS:

1) Diseñar y desarrollar las clases Nodo y árbol, que implementen un Árbol Binario de Búsqueda, con las siguientes operaciones:

***- Insertar(valor)**: Se añade un valor al árbol. No debe aceptar repetidos, si sucede este caso, arrojar un mensaje de error en consola y no añadir el nodo. El método no regresa nada.*

***- Buscar(valor)**: Se busca un valor en el árbol. Si existe el valor regresar True, en caso contrario, regresar False.*

***- Max()**: Regresa el valor máximo del árbol. Si esta vacío, regresa un None.*

***- Min()**: Regresa el valor mínimo del árbol. Si esta vacío, regresa un None.*

*- **Imprimir( )**:- Se imprime el árbol en :*

*- **PreOrden:** EN UNA SOLA LINEA.*

*- **InOrden:** EN UNA SOLA LINEA.*

*- **PostOrden**: EN UNA SOLA LINEA.*

*- **Anchura**: EN UNA SOLA LINEA.*

***- Eliminar(valor)**: Se elimina un valor del árbol, si existe. Si no existe el valor, arrojar un mensaje de error. CUIDADO: al eliminar un nodo, el árbol debe mantener las propiedades de los ABB. El método no regresa nada.*

***- (EXTRA) PrintPretty()**: Dibujar el árbol usando código ASCII: el árbol debe imprimirse desde la raíz, con los hijos en el mismo nivel y usando diagonales para representar las ramas, guardando espacio equidistantes entre nodos.*

Esté método se llama al final del método Imprimir().

2) Generar un programa en Python que implemente, un objeto tipo árbol y ejecute las siguientes operaciones:

Imprimir()

Insertar(8)

Insertar(3)

Insertar(10)

Insertar(1)

Insertar(6)

Insertar(14)

Insertar(4)

Insertar(7)

Insertar(13)

Imprimir( )

Insertar(14) # Repetido, debe mostrar mensaje de error

Insertar(1) # Repetido, debe mostrar mensaje de error

Mínimo(),

Máximo(),

Buscar(4)

Buscar(8)

Buscar(13)

Buscar(2)

Buscar(15)

Borrar(7) # Borrando el 7 (sin hijos)

Imprimir()

Borrar(10) # Borrando el 10 (solo hijo der)

Imprimir()

Borrar(6) # Borrando el 6 (solo hijo izq)

Imprimir()

Borrar(3) # Borrando el 3 (ambos hijos)

Imprimir()

Borrar(3) # Borrando el 3 (nodo no existe)

Imprimir()

Borrar(8) # Borrando el 8 (raíz, ambos hijos)

Imprimir()

Añadir(100)

Imprimir()

Práctica de Estudio 8: **Arboles. Parte II**
Estructura de Datos y Algoritmos Grupo 01
Facultad de Ingeniería
Departamento de Computación

OBJETIVO:

El estudiante conocerá e identificará las características de la estructura no lineal árbol.

INTRODUCCIÓN:

La búsqueda en árboles binarios es un método de búsqueda simple, dinámico y eficiente considerado como uno de los fundamentales en Ciencia de la Computación. De toda la terminología sobre árboles, tan sólo recordar que la propiedad que define un árbol binario es que cada nodo tiene a lo más un hijo a la izquierda y uno a la derecha. Para construir los algoritmos consideraremos que cada nodo contiene un registro con un valor clave a través del cual efectuaremos las búsquedas. En las implementaciones que presentaremos sólo se considerará en cada nodo del árbol un valor del tipo *elemento* aunque en un caso general ese tipo estará compuesto por dos: una clave indicando el campo por el cual se realiza la ordenación y una información asociada a dicha clave o visto de otra forma, una información que puede ser compuesta en la cual existe definido un orden.

Un árbol binario de búsqueda (ABB) es un árbol binario con la propiedad de que todos los elementos almacenados en el subárbol izquierdo de cualquier nodo *x* son menores que el elemento almacenado en *x* ,y todos los elementos almacenados en el subárbol derecho de *x* son mayores que el elemento almacenado en *x*

Formas de búsqueda de un árbol:

PreOrden: RAIZ-PREORDEN(IZQ)-PREORDEN(DER)

inOrden: InOrden(izq) – Raiz – InOrden(Der)

PostOrden: InOrden(izq) – Raiz – InOrden(Der) - Raiz

BFS

DFS

DESARROLLO:

a)

```
class  BinaryTree():
    def __init__(self):                 #Contructor de la clase···
    def addRoot(self, value: int):           #Metodo agregar···
    def _AddNode(self, value , node):                   #Metodo para agregar
    def _Search( self, node: Node , value: int ):       #Busqueda de un valor
    def Min(self, node: Node = None):           #Valor minimo···
    def Max(self, node: Node = None):           #Valor maximo···
    def Delete( self, value: int):                 #Borrar valor ···
    def _SearchHelper(self, value: int):           #Buscador ayuda···
    def NextNode( self, node: Node):         #Metodo para saber el nodo siguie
    def PreviousNode( self, node: Node ):    #Metodo para saber el nodo anteri
    def _reTransplant ( self , original: Node , reemplazo: Optional[ Node ]):
    def _HelperDelete( self, node: Node):                   #Ayuda a elminina
    def Print(self):                                        #metodo imprimir
```

```
def __valuesList(self, traversalAlg):  #List
def __inorder(self, root: Node):        #InOr
def __preorder(self, root: Node):        #RAI
def __postorder(self, root: Node):       #Pos
def __anchura_traversal(self, root: Node): #
def __getDepth(self): #Metodo que obtiene la
def __printPretty(self): #Metodo para imprim
```

Clase BinaryTree:

La clase Tree será aquela que realiza las operaciones de búsqueda, eliminacion y agregación de los nodos que esten en el. A partir de una lógica de correspondencia a partir de sus valores agregados que serán dependientes a si son mayor o menor que el valor padre.

```python
class Node:#Clase Node
    def __init__(self , value: int):#Atributos de la clase

        self.value:int = value
        self.parentNode: Optional[Node] = None
        self.leftChild: Optional[Node] = None
        self.rightChild: Optional[Node] = None
```

Clase Node

La clase nodo se distingue por sus atributos, como los nodos son diferenciados por el lado derecho y el lado izquierdo, es necesario crear esta distinción por medio de sus atributos, al igual que estos dos nodos tienen su valor propio. Se utiliza programacion tipada para recibir valroes que ayudaran con las operaciones a realizar

a.insertar)

```python
def addRoot(self, value: int):
    if self.root is None:
        self.root = Node(value)
        return

    parentNode = self.root
    if parentNode.value == value:
        raise KeyError("Valor repetido")

    if value < parentNode.value:
        childNode = parentNode.leftChild
    else:
        childNode = parentNode.rightChild

    while childNode != None:
        parentNode = childNode

        if parentNode.value == value:
            raise KeyError("Valor repetido")

        if value < childNode.value:
            childNode = childNode.leftChild
        else:
            childNode = childNode.rightChild

    newNode = Node(value)
    newNode.parentNode = parentNode

    if newNode.value < parentNode.value:
        parentNode.leftChild = newNode
    else:
        parentNode.rightChild = newNode
```

Método addRoot

El metodo incertará valores en las raices, en el caso de que no haya una raiz el primer valor a leer será un objeto tipo Node que recibe el valor. A partir de ahí, el nodo padre será la raiz.

Se verifica si el valor a insertar es mayot o menor al valor del padre, dependiendo de esto se diriirá al lado izquierdo o derecho del nodo. Se asiganará como hijo de su respectivo padre y se guarda el registro del nodo actual.

```
def _AddNode(self, value , node):
    if ( value < node.value ):
        if( node.leftChild != None ):
            self._AddNode( value, node.leftChild )
        else:

            node.leftChild = Node(value)

    else:
        if( node.rightChild !=None ):
            self._AddNode(value, node.rightChild )
        else:
            node.rightChild = Node( value )
```

Metodo _addNode

Para agregar el nodo, es necesario saber si es mayor o menor al elemento a comparar. Si el nodo izquierdo o derecho del nodo que queremos agregar tiene valores,, se hace recursivamente la llamada de la función para determinar de que lado irá acomodado. Si este no tiene elementos, se convertirá en un objeto tipo Node.

a.recorridos)

```
def __inorder(self, root: Node):        #InOrden(iz
    if root == None:
        return ""
    base = self.__inorder(root.leftChild)
    base += " -> " + str(root.value)
    return base + self.__inorder(root.rightChild)

def __preorder(self, root: Node):       #RAIZ-PREO
    if root is None:
        return ""
    base = " -> " + str(root.value)
    base += self.__preorder(root.leftChild)
    return base + self.__preorder(root.rightChild)

def __postorder(self, root: Node):      #PostOrden
    if root is None:
        return ""
    base = self.__postorder(root.leftChild)
    base += self.__postorder(root.rightChild)
    return base + " -> " + str(root.value)
```

Recorrido InOrd

En busqueda inOrd, es necesario verificar si el arbol tiene elementos para recorrer; de ser así, se buscará InOrd del lado izquiedo hasta que llegue a un nodo.left Null por medio de recursividad de la funcíón, después imprimirá los nodos en medida en que fueron recorridos pará después buscar por su lado derecho InOrd.

Recorrido preOrd

Para el recorrido, imprimirá el valor del nodo que se encuentre en ese momento, para despues buscar por el lado izquiedo e ir repitiendo el mismo proceso hastá llegar el nodo izquierdo None, después regresivamente irá a buscar del lado derecho llamando recursivamente la función.

Recorrido postOrd

Para realizar postorden es necesario buscar del lado izquierdo hasta que se llegue al none, despues de esto se buscará por el derecho, hasta llegar al none, finalmente imprimirá este valor y la funcion se realizará recursivamente

```
def Print(self):                                                    #metodo
    if self.root == None:
        return print("El arbol actual esta vacio \n")
    else:
        preOrder = self.__valuesList(self.__preorder)
        inOrder = self.__valuesList(self.__inorder)
        postOrder = self.__valuesList(self.__postorder)
        anchura = self.__valuesList(self.__anchura_traversal)
        print("Preorder:", preOrder)
        print("Inorder:", inOrder)
        print("Postorder:", postOrder)
        print("Anchura:", anchura, "\n")
        self.__printPretty()
        print()
```

Metodo que permite la impresión de los recorridos.

Los valroes seran almacenados en un string “Base” para se impresos en forma lineal.

RESULTADOS:

a.Minimo)

```
def Min(self, node: Node = None):
    if node == None:
        if self.root == None:
            return None
        node = self.root

    while node.leftChild != None:
        node = node.leftChild
    return node
```

Metodo Min

Buscará el valor minimo del arbol, que bien sabemos que se encuentra del lado izquierdo, de ahí haremos su busqueda hasta llegar a none del arbol, es decir al valor minimo.

a.Maximo)

```
def Max(self, node: Node = None):
    if node == None:
        if self.root == None:
            return None
        node = self.root

    while node.rightChild != None:
        node = node.rightChild
    return node
```

Buscará el valor máximo del arbol, que bien sabemos que se encuentra del lado derecho, de ahí haremos su busqueda hasta llegar a none del arbol, es decir al valor máximo.

a.Eliminar)

```
def Delete( self, value: int):
    valueDelete = self._SearchHelper( value )
    if valueDelete != None:
        self._HelperDelete(valueDelete)
    else:
        print("No existe valor en el arbol")
```

Metodo eliminar:

Para eliminar es necesario buscar el valor a elimar, es por ello que se utiliza un helper para poder realizar esta tarea, si este valor es None significa que no existe, pero si este existe entonces se utilizará de otro helper para eliminar el valor.

```
def _SearchHelper(self, value: int):
    return self._Search(self.root , value)
```

Metodo searhHeler

Busca el elemento por medio de una función buscar. <Search>

```
def _Search( self, node: Node , value: int ):
    if (node == None or node.value == value):
        return node

    elif value < node.value:
        return self._Search(node.leftChild, value)
    else:
        return self._Search(node.rightChild, value)
```

Metodo Search

Dependiendo de si el valor es mayor o menor que el nodo raiz, determinará si busscar por la derecha o por la izquierda. Hará esto recursivamente hasya encontrar o no con el elemento a buscar.

```
def _HelperDelete( self, node: Node):                    #Ayuda a e

    if node.leftChild == None:                           #si el nod
        self._reTransplant ( node, node.rightChild )     #Se replan
    elif node.rightChild == None:                         #Si el no
        self._reTransplant ( node, node.leftChild )       #Se repla
    else:
        nextNode = self.Min(node.rightChild)              #el nodo
        if nextNode.parentNode != node:                   #Si el pa
            self._reTransplant ( nextNode, nextNode.rightChild )
            nextNode.rightChild = node.rightChild          #El no
            nextNode.rightChild.parentNode = nextNode      #el pa
```

Metodo HelperDelete

Para eliminar el valor es necesario acomodar el arbol, de esta formapodemos asegurar que se mantiene el orden y la propiedad de un abb, es por ello que se utiliza el metodo retransplant para poder realizar esta tarea. Si el nodo hijo no tiene valores, se realiza un retransplante de este lado, al igual que el lado derecho.

Si no son none, entonces sacaremos el valor minimo del nodo hijo izquierdo para poder concer el nodo siguiente, si este valor es diferente de none, se transplanta el valor del hijo derecho y se obtiene el nodo siguiente por el atributo padre de este.

```
def _reTransplant ( self , original: Node , reemplazo: Optional[ Node ]):

    if original.parentNode == None:                #Si el padre del nodo
        self.root = reemplazo                      #La raiz será el valor
    elif original == original.parentNode.leftChild: #O si el valor origina
        original.parentNode.leftChild = reemplazo   #este será el valor a
    else:
        original.parentNode.rightChild = reemplazo  #De lo contrario se re

    if reemplazo != None:                          #Si el remplazo es dif
        reemplazo.parentNode = original.parentNode  #EL padre del demplazo
```

Metodo reTransplant

Para que se remplace, es necesario obtener el nodo que vamos a remplazar, si el padre de este nodo original es none, entonces la raiz es en si el remplazo. Pero si el valor del padre de su hijo es igual al valor original, entonces este será el remplazo. De lo contrario el hijo derecho del padre del nodo orignal, será el remplazo.
Si el remplazo es diferente de un none, entonces es lo mismo que el padre del original.

a.PrettyPrint():

```
def __getDepth(self): #Metodo que obtiene lá
    if self.root == None:
        return 0

    queue = Queue()     #Creamos una cola ti
    queue.put(self.root)   #Insertamos la ra
    depth = 0       #la profundidad de esto

    while not queue.empty(): #Mientras la co
        size = queue.qsize() #Se tomará el v
        depth += 1 #su profndidad será cada

        for _ in range(size):
            value = queue.get()          #O
            if value.leftChild != None:
                queue.put(value.leftChild)
            if value.rightChild != None:
                queue.put(value.rightChild)

    return depth - 1
```

Metodo getDepth

Para obtener la profundida creamos una cola a la cual le iremos metiendo valores de la raiz, entonces estos valroes los iremos comparando y obteniendo uno de cada ellos, hasta que tomamos el tamaño para meterlo dentro de un for que revisará de los lados izquierdo y derecho de los hijos, tomando cada vez un nivel mas de altura que va acumulando.

```
def __printPretty(self): #Metodo para imprimir bonito
    if self.root is None:  #si la raiz no tiene valores
        return                #salimos
    maxDepth = self.__getDepth() #Buscamos la altura maxima
    largestLineLength = 2 ** (maxDepth + 2) - 3
    TOTAL_LINES = maxDepth + 1
    TOTAL_EDGE_LINES = 2**(maxDepth + 1) - 1 - maxDepth
    nodesLines = [[" "] * largestLineLength for _ in range(TOTAL_LINES)]
    edgesLines = [[" "] * largestLineLength for _ in range(TOTAL_EDGE_LINES)]

    def __drawEdge(depth: int, position: int, idxMutator, edge: str): #Dibujar

    def __drawEdgeLeft(depth: int, position: int): #Aristas izquiedas ···

    def __drawEdgeRight(depth: int, position: int): #Aristas Derechas ···

    def __miniTree(root: Node, depth: int, position: int): #Dibujamos las linea

    __miniTree(self.root, 0, largestLineLength // 2)    #Estableemos el patron
    print("".join(nodesLines[0]))
    i = 1
    for currentDepth in range(1, maxDepth+1):          #insertamos lineas
        for _ in range(2**(maxDepth - currentDepth + 1) - 1):
            print("".join(edgesLines[i]))
            i += 1

        print("".join(nodesLines[currentDepth]))       #imprimimos linea
```

Metodo prettytprint

Para el metodo, fue necesario conocer la profundida del arbol, para así, poder dibujar las respectivas lineas que componen al arbol, es por ello que el total de las tineas se verá infleunciado con la profundida de este mismo arbol, recorriendo y contantdo las lineas tanto de los mini arboles creados, asi como de los lados derecho, izquiedo y el dibujo en general de las aristas necesarias para darle el

epacio necesario al rabol. Despues de conocer estos valores, se emmpezara con un dibujo de linea para darle forma al arbol.

```
def __drawEdge(depth: int, position: int, idxMutator, edge: str): #Dibujar aristas

    finalDepth = int(2**(maxDepth + 1) * (-0.5**depth + 1) - depth)
    startDepth = finalDepth - 2**(maxDepth-depth+1) + 2
    currentCharIdx = position
    for currDepth in range(startDepth, finalDepth+1):
        currentCharIdx = idxMutator(currentCharIdx)
        edgesLines[currDepth][currentCharIdx] = edge

def __drawEdgeLeft(depth: int, position: int): #Aristas izquiedas
    def indexMutator(idx: int):
        return idx - 1
    __drawEdge(depth, position, indexMutator, "/")

def __drawEdgeRight(depth: int, position: int): #Aristas Derechas
    def indexMutator(idx: int):
        return idx + 1
    __drawEdge(depth, position, indexMutator, "\\")

def __miniTree(root: Node, depth: int, position: int): #Dibujamos las lineas de los ar
    nodesLines[depth][position] = str(root.value)

    if root.leftChild is not None:                      #Si no esta vacio lo dibuja
        __drawEdgeLeft(depth + 1, position)
        __miniTree(root.leftChild, depth + 1, position - 2**(maxDepth - depth))

    if root.rightChild is not None:                     #si no esta vacio lo dibuja
        __drawEdgeRight(depth + 1, position)
        __miniTree(root.rightChild, depth + 1, position + 2**(maxDepth - depth))
```

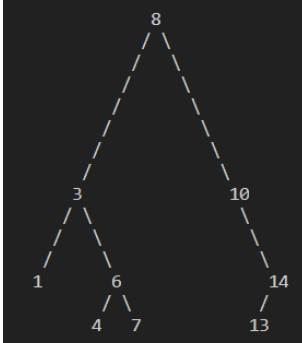
Metodos que sirven para dibujar el arbol

RESULTADOS:

```
from BinaryTree import BinaryTree
def main():
    arbol = BinaryTree()
    arbol.Print()
    arbol.addRoot(8)
    arbol.addRoot(3)
    arbol.addRoot(10)
    arbol.addRoot(1)
    arbol.addRoot(6)
    arbol.addRoot(14)
    arbol.addRoot(4)
    arbol.addRoot(7)
    arbol.addRoot(13)
    arbol.Print()
    try:
        arbol.addRoot(14)  # Repetido, debe mostrar mensaje de err
    except KeyError:
        print("El nodo con llave 14 ya se encuentra en el árbol")
    try:
        arbol.addRoot(1)  # Repetido, debe mostrar mensaje de erro
    except KeyError:
        print("El nodo con llave 1 ya se encuentra en el árbol")
    print("Mínimo:", arbol.Min().value)
    print("Máximo:", arbol.Max().value)
    arbol._SearchHelper(4)
    arbol._SearchHelper(8)
    arbol._SearchHelper(13)
    arbol._SearchHelper(2)
    arbol._SearchHelper(15)
    arbol.Delete(7)  # Borrando el 7 (sin hijos)
    arbol.Print()
    arbol.Delete(10)  # Borrando el 10 (solo hijo der)
    arbol.Print()
    arbol.Delete(6)  # Borrando el 6 (solo hijo izq)
    arbol.Print()
    arbol.Delete(3)  # Borrando el 3 (ambos hijos)
    arbol.Print()
try:
    arbol.Delete(3)  # Borrando el 3 (nodo no existe)
except KeyError:
    print("El nodo con llave 3 no se encuentra en el árbol")
arbol.Print()
arbol.Delete(8)  # Borrando el 8 (raíz, ambos hijos)
arbol.Print()
arbol.addRoot(100)
arbol.Print()
```

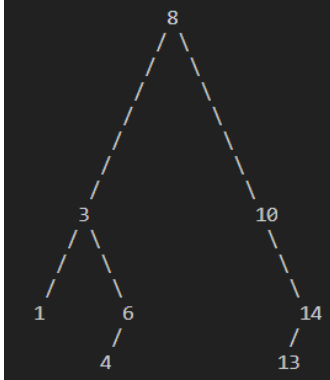
b.Salida de datos:

```
Preorder:  -> 8 -> 3 -> 1 -> 6 -> 4 -> 7 -> 10 -> 14 -> 13
Inorder:  -> 1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 10 -> 13 -> 14
Postorder:  -> 1 -> 4 -> 7 -> 6 -> 3 -> 13 -> 14 -> 10 -> 8
Anchura:  -> 8 -> 3 -> 10 -> 1 -> 6 -> 14 -> 4 -> 7 -> 13
```

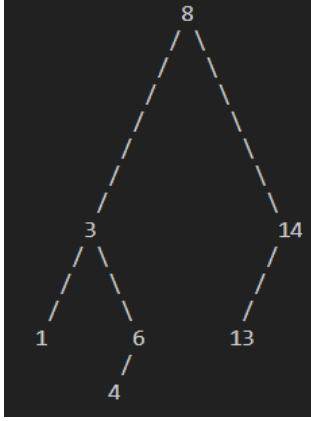


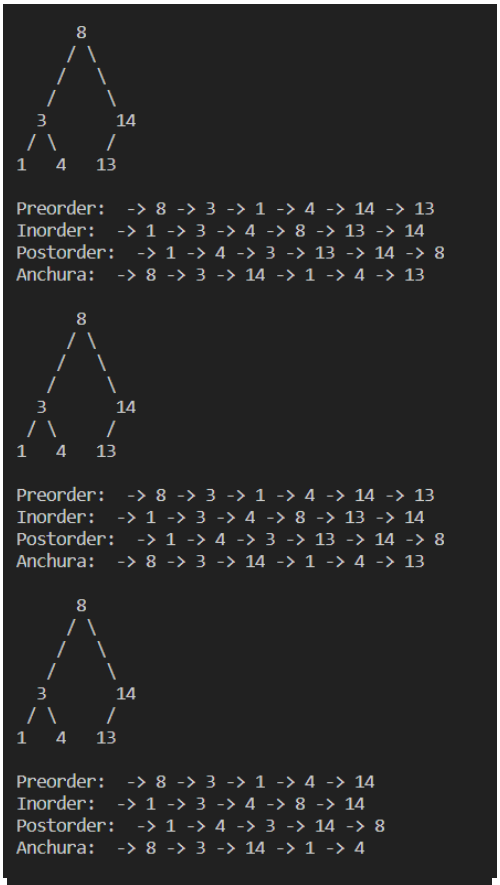

```
El nodo con llave 14 ya se encuentra en el árbol
El nodo con llave 1 ya se encuentra en el árbol
Mínimo: 1
Máximo: 14
```

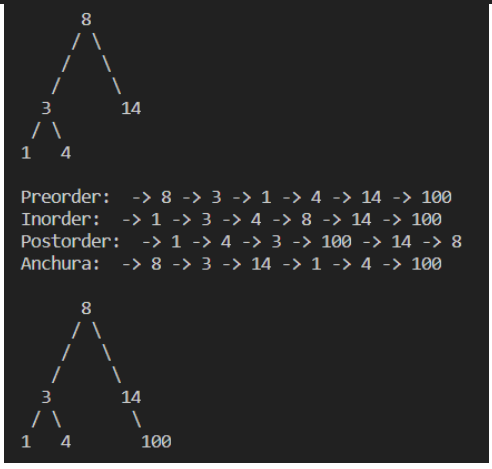
```
Preorder:  -> 8 -> 3 -> 1 -> 6 -> 4 -> 10 -> 14 -> 13
Inorder:  -> 1 -> 3 -> 4 -> 6 -> 8 -> 10 -> 13 -> 14
Postorder:  -> 1 -> 4 -> 6 -> 3 -> 13 -> 14 -> 10 -> 8
Anchura:  -> 8 -> 3 -> 10 -> 1 -> 6 -> 14 -> 4 -> 13
```




```
Preorder:  -> 8 -> 3 -> 1 -> 6 -> 4 -> 14 -> 13
Inorder:  -> 1 -> 3 -> 4 -> 6 -> 8 -> 13 -> 14
Postorder:  -> 1 -> 4 -> 6 -> 3 -> 13 -> 14 -> 8
Anchura:  -> 8 -> 3 -> 14 -> 1 -> 6 -> 13 -> 4
```




8
3
14
1
4
13
Preorder: -> 8 -> 3 -> 1 -> 4 -> 14 -> 13
Inorder: -> 1 -> 3 -> 4 -> 8 -> 13 -> 14
Postorder: -> 1 -> 4 -> 3 -> 13 -> 14 -> 8
Anchura: -> 8 -> 3 -> 14 -> 1 -> 4 -> 13
8
3
14
1
4
13
Preorder: -> 8 -> 3 -> 1 -> 4 -> 14 -> 13
Inorder: -> 1 -> 3 -> 4 -> 8 -> 13 -> 14
Postorder: -> 1 -> 4 -> 3 -> 13 -> 14 -> 8
Anchura: -> 8 -> 3 -> 14 -> 1 -> 4 -> 13
8
3
14
1
4
13
Preorder: -> 8 -> 3 -> 1 -> 4 -> 14
Inorder: -> 1 -> 3 -> 4 -> 8 -> 14
Postorder: -> 1 -> 4 -> 3 -> 14 -> 8
Anchura: -> 8 -> 3 -> 14 -> 1 -> 4


8
3
14
1
4
Preorder: -> 8 -> 3 -> 1 -> 4 -> 14 -> 100
Inorder: -> 1 -> 3 -> 4 -> 8 -> 14 -> 100
Postorder: -> 1 -> 4 -> 3 -> 100 -> 14 -> 8
Anchura: -> 8 -> 3 -> 14 -> 1 -> 4 -> 100
8
3
14
1
4
100

CONCLUCIONES:

Un arbol binario es una herramienta que permite que sus noodos esten asociados a una clave de ordenacion, siempre y cuando esta sea comparable. Es por ello que nos interesa si son arboles ordenados o no, ya que esto permite una solución eficiente en colecciones ordenadas de elementos en tiempo O(n), donde veremos que hay una relación coherente entre todos sus elementos que lo componen. Se ve como la clave de la raiz dependerá en como se ordenan los nuevos nodos, es decir que los valores mayores a este irán del lado izquierdo, mientras que los mayores se acomodan del aldo derecho, ¿Qué significa esto? Esto da una ventaja ya que permite aprovecharse y explotar al máximo esta propiedad para reacomodarlos, ordenarlos, buscar cualqueir valor, inclusive el máximo y el minimo en su correspondiente esquina del arbol.

Dentro de la computación, un arbol juega un papel importante, ya que este permite una colección de datos estrechamente relacionada con sus llaves, sea el caso de un arbol tipo B que permite el mejor rendimiento para un disco duro, al igual que un arbol normal ayuda a ver una comparativa de sus elementos.