	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Jesús Cruz Navarro
Asignatura:	Estructuras de Datos y Algoritmos 2
Grupo:	1
No de Práctica(s):	8
Integrante(s):	Ugalde Velasco Armando
No. de Equipo de cómputo empleado:	
No. de Lista o Brigada:	32
Semestre:	2021-1
Fecha de entrega:	24 de noviembre de 2020
Observaciones:	

CALIFICACIÓN: _____

PRÁCTICA 8: ÁRBOLES 1

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

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.

```python
def insert(self, key: int):
    if self.root is None:
        self.root = TreeNode(key)
        return

    parent = self.root
    if parent.key == key:
        raise KeyError("Repeated key")
    child = parent.left if key < parent.key else parent.right

    while child is not None:
        parent = child
        if parent.key == key:
            raise KeyError("Repeated key")
        child = child.left if key < child.key else child.right

    newNode = TreeNode(key)
    newNode.parent = parent

    if newNode.key < parent.key:
        parent.left = newNode
    else:
        parent.right = newNode
```

Método insert

Para insertar un nodo, se recorre el árbol tomando decisiones respecto al valor del nodo, es decir, si el valor del nodo a insertar es menor al valor del nodo actual, se avanza al hijo izquierdo de éste. Finalmente, cuando se llega a una hoja, se inserta el nodo en la posición adecuada. Si se descubre algún nodo con una llave igual a la del nodo a insertar, se tira una excepción.

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

```python
def search(self, key: int):
    return self.__searchAux(self.root, key)

def __searchAux(self, root: TreeNode, key: int):
    if root is None or root.key == key:
        return root
    elif key < root.key:
        return self.__searchAux(root.left, key)
    else:
        return self.__searchAux(root.right, key)
```

Método search

Para buscar un nodo, se recorre el árbol de la misma forma que en el método insert. Sin embargo, en caso de encontrarse el nodo deseado, éste se retorna. Si se llega al final del árbol sin encontrarlo, se retorna None. Nótese que, para realizar una implementación más robusta, se decidió retornar el **nodo** con la llave deseada o **None** para los casos donde se encuentra el nodo y donde no existe, respectivamente. De esta forma, es posible acceder a la información satélite del nodo encontrado además de otros datos pertinentes.

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

```python
def maximum(self, root: TreeNode = None):
    if root is None:
        if self.root is None:
            return None
        root = self.root

    while root.right is not None:
        root = root.right
    return root
```

Método maximum

En caso de estar vacío, el método retorna **None**, de lo contrario, retorna la hoja en el extremo derecho del árbol.

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

```
def minimum(self, root: TreeNode = None):
    if root is None:
        if self.root is None:
            return None
        root = self.root

    while root.left is not None:
        root = root.left
    return root
```

Método minimum

En caso de estar vacío, el método retorna **None**, de lo contrario, retorna la hoja en el extremo izquierdo del árbol.

- **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.

    Si el árbol está vacío, imprimir un mensaje indicándolo.

```python
def print(self):
    if self.root is None:
        return print("El árbol está vacío\n")
    preorder = self.__formatListStr(self.__preorder)
    inorder = self.__formatListStr(self.__inorder)
    postorder = self.__formatListStr(self.__postorder)
    breadth_first = self.__formatListStr(self.__breadth_first_traversal)
    print("Preorder:", preorder)
    print("Inorder:", inorder)
    print("Postorder:", postorder)
    print("Breadth first:", breadth_first, "\n")
    self.__printPretty()
    print()

def __formatListStr(self, traversalAlg):
    return traversalAlg(self.root)[4:]

def __inorder(self, root: TreeNode):
    if root is None:
        return ""
    base = self.__inorder(root.left)
    base += " → " + str(root.key)
    return base + self.__inorder(root.right)
```

Método print

Se obtienen las representaciones de cada recorrido utilizando los métodos respectivos. El método **__inorder**, **__preorder** y **__postorder** obtienen una cadena con las representaciones respectivas de forma recursiva. Para obtener la representación del recorrido en anchura, se implementó el algoritmo **BFS**, formando la cadena respectiva cada que se visitan los nodos. Finalmente, debido a que las cadenas retornadas por los métodos anteriores contienen una flecha extra al inicio, se implementó el método **__formatListStr** para retornar una nueva cadena sin éstas.

- **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.

```python
def __deleteAux(self, node: TreeNode):
    if node.left is None:
        self.__transplant(node, node.right)
    elif node.right is None:
        self.__transplant(node, node.left)
    else:
        successor = self.minimum(node.right)
        if successor.parent ≠ node:
            self.__transplant(successor, successor.right)
            successor.right = node.right
            successor.right.parent = successor
        self.__transplant(node, successor)
        successor.left = node.left
        successor.left.parent = successor

def delete(self, key: int):
    toDelete = self.search(key)
    if toDelete is not None:
        self.__deleteAux(toDelete)
    else:
        raise KeyError("Key does not exist")
```

Método delete

Se implementó el algoritmo presente en el libro *Introduction to Algorithms*. En éste, se consideran tres casos: el nodo tiene cero, uno o dos hijos. Nótese que se utiliza el método auxiliar **transplant**, que reemplaza un nodo por otro. En el primer caso, solamente es necesario reemplazar el nodo a eliminar por el valor **None**, lo cual se realiza implícitamente mediante el método **transplant** en la primera condición. En el segundo caso, se reemplaza el nodo a eliminar por su único hijo, de nuevo, mediante el método **transplant.** Finalmente, en el tercer caso, se reemplaza el nodo a eliminar por su sucesor, y se realizan los ajustes necesarios para unir el hijo izquierdo del nodo a eliminar con el sucesor. Si el sucesor no es un hijo del nodo a eliminar, primero se “extrae” y se conecta con el hijo derecho del nodo a eliminar, de tal modo que este último sea su hijo.

- **(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()**.

```
def __printPretty(self):
    if self.root is None:
        return
    maxDepth = self.__getDepth()
    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 __computeEdge(depth: int, position: int, idxMutator, edge: str):

        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 __computeEdgeLeft(depth: int, position: int):
        def indexMutator(idx: int):
            return idx - 1
        __computeEdge(depth, position, indexMutator, "/")

    def __computeEdgeRight(depth: int, position: int):
        def indexMutator(idx: int):
            return idx + 1
        __computeEdge(depth, position, indexMutator, "\\")
```

Método printPretty

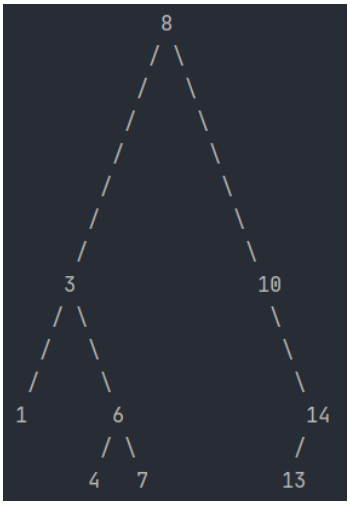
Primero, se obtiene la profundidad máxima del árbol. Posteriormente, se determina el total de líneas de nodos a imprimir y el total de líneas correspondientes a las aristas a imprimir. Lo anterior se dedujo utilizando fórmulas para sumas geométricas. Además, se inicializan las listas que contendrían los caracteres correspondientes a éstas. Después, se definen algunos métodos auxiliares: **__computeEdge**, que coloca los caracteres necesarios para formar una arista dependiendo de la posición y profundidad del nodo; las funciones específicas para cada tipo de arista *(izquierda o derecha);* y finalmente la función **__computePrettySubtree**, que coloca los valores

de las llaves de los nodos y sus aristas en las posiciones correctas de las listas respectivas, de forma recursiva.

```
def __computePrettySubtree(root: TreeNode, depth: int, position: int):
    nodesLines[depth][position] = str(root.key)

    if root.left is not None:
        __computeEdgeLeft(depth + 1, position)
        __computePrettySubtree(root.left, depth + 1, position - 2**(maxDepth - depth))

    if root.right is not None:
        __computeEdgeRight(depth + 1, position)
        __computePrettySubtree(root.right, depth + 1, position + 2**(maxDepth - depth))

__computePrettySubtree(self.root, 0, largestLineLength // 2)
print("".join(nodesLines[0]))
i = 1
for currentDepth in range(1, maxDepth+1):
    for _ in range(2**(maxDepth - currentDepth + 1) - 1):
        print("".join(edgesLines[i]))
        i += 1

    print("".join(nodesLines[currentDepth]))
```

Continuación de método printPretty

Finalmente, se llama a la función anterior con los valores iniciales para colocar la raíz del árbol en la posición correcta. Con ello, se colocan todos los nodos del árbol. Por último, se imprimen las listas obtenidas. A continuación, se muestra un ejemplo de la representación generada:



Representación de árbol

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

```
def test():
    tree = BinaryTree()
    tree.print()
    tree.insert(8)
    tree.insert(3)
    tree.insert(10)
    tree.insert(1)
    tree.insert(6)
    tree.insert(14)
    tree.insert(4)
    tree.insert(7)
    tree.insert(13)
    tree.print()
    try:
        tree.insert(14)  # Repetido, debe mostrar mensaje de error
    except KeyError:
        print("El nodo con llave 14 ya se encuentra en el árbol")
    try:
        tree.insert(1)  # Repetido, debe mostrar mensaje de error
    except KeyError:
        print("El nodo con llave 1 ya se encuentra en el árbol")
    print("Mínimo:", tree.minimum().key)
    print("Máximo:", tree.maximum().key)
    tree.search(4)
    tree.search(8)
    tree.search(13)
    tree.search(2)
    tree.search(15)
    tree.delete(7)  # Borrando el 7 (sin hijos)
    tree.print()
    tree.delete(10)  # Borrando el 10 (solo hijo der)
    tree.print()
```

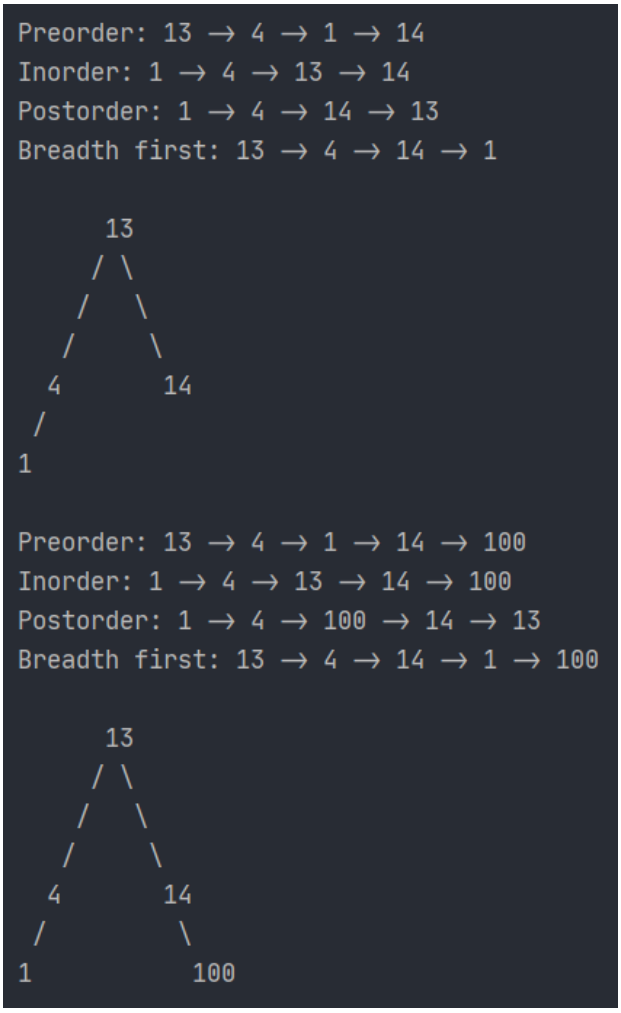
Método test con las operaciones solicitadas

A continuación, se muestran algunas capturas de pantalla de la salida obtenida del programa anterior:

```
El árbol está vacío

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
Breadth first: 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
Breadth first: 8 → 3 → 10 → 1 → 6 → 14 → 4 → 13
```

Salida del programa



Salida del programa

CONCLUSIONES

Los árboles son una de las estructuras de datos más importantes en las Ciencias de la Computación; nos permiten resolver una gran cantidad de problemas de diversa índole, y, en consecuencia, sus aplicaciones son innumerables. Es posible establecer ciertas invariantes en ellos para deducir alguna característica o circunstancia que nos permita resolver algún problema: en particular, la invariante analizada en la práctica nos permite realizar búsquedas en tiempo **O(h),** donde **h** es la altura del árbol.

Sin embargo, es importante mencionar que, para asegurar un rendimiento óptimo en las búsquedas para cualquier caso, es necesario que el árbol esté balanceado o medianamente balanceado. En otras palabras, su altura debe ser **O(log n).** El árbol Red-Black es un ejemplo de una estructura que cumple con la condición anterior, ya que se *“auto-balancea”* para conservar una altura adecuada.