Ficha 29

Estructuras No Lineales: Árboles Binarios de Búsqueda

1.] Árboles binarios: necesidades de uso.

Para ciertos problemas en los que el conjunto de datos y/o resultados crece de forma impredecible, o directamente se desconoce cuántos son los datos que deben procesarse, los *arreglos adaptativos* (que son los que están disponibles en Python a través del tipo o clase *list*) pueden ayudar a plantear soluciones muy eficientes en cuanto al tiempo de ejecución y también en cuanto al uso de memoria. Si se quiere, por ejemplo, cargar una secuencia de datos con el objetivo de ir almacenándolos en un arreglo adaptativo (una variable de tipo *list* en Python), pero de forma que el arreglo/lista se vaya manteniendo ordenado a medida que los datos llegan, se puede aplicar nuestro conocido *algoritmo de inserción ordenada con búsqueda binaria* y la tarea se cumplirá con mucha rapidez.

Sin embargo, debemos recordar que cada cuando se agrega un valor al arreglo/lista (y sobre todo si ese casillero se agrega entre dos casillas previamente existentes) en ese momento en (el peor caso) internamente se crea un segundo arreglo temporal, se copian en ese arreglo los valores a la izquierda del que se va a agregar, se agrega el nuevo valor, y se copian los que restan a la derecha del que se acaba de insertar. Sin entrar en demasiados detalles, esa operación de inserción mantiene un uso preciso de la cantidad de memoria (exactamente la memoria que se necesita, aunque en un momento dado existirán dos arreglos en lugar de solo uno), pero insume un tiempo hasta finalizar la inserción que es *O(n)* en el *peor caso* (todos los *n* elementos del viejo arreglo deben copiarse al nuevo arreglo)[1].

Como el arreglo quedará finalmente ordenado (estamos haciendo inserción ordenada), las búsqueda que sean necesarias después de crearlo pueden hacerse con búsqueda binaria, y obtener un muy buen tiempo de búsqueda de O(log(n)) en el peor caso. Pero nos queda el tiempo implícito de la forma O(n) para una inserción, en el peor de los casos.

Nos preguntamos si es posible mejorar ese rendimiento en el tiempo de inserción, pero manteniendo (o incluso mejorando) la ventaja del uso relativamente eficiente de la memoria. Concretamente: ¿sería posible recurrir a otra forma de organizar los datos, de forma que sigamos teniendo un consumo aceptable de memoria pero tal que se reduzca el tiempo de inserción (por ejemplo, llevando ese tiempo a O(log))? La respuesta es que esa nueva estructura puede ser la que se conoce como *árbol binario*, y en particular, la variante conocida como *árbol binario de búsqueda* acerca de la cuál trata esta Ficha.

[1] El análisis es un poco más complejo, y requiere elementos de análisis amortizado además de análisis de peor caso, pero en general podemos tomar la visión general expuesta en el texto: el proceso completo tendrá un *peor caso* en O(n) unidades de tiempo.

2.] Árboles binarios: definiciones y conceptos.

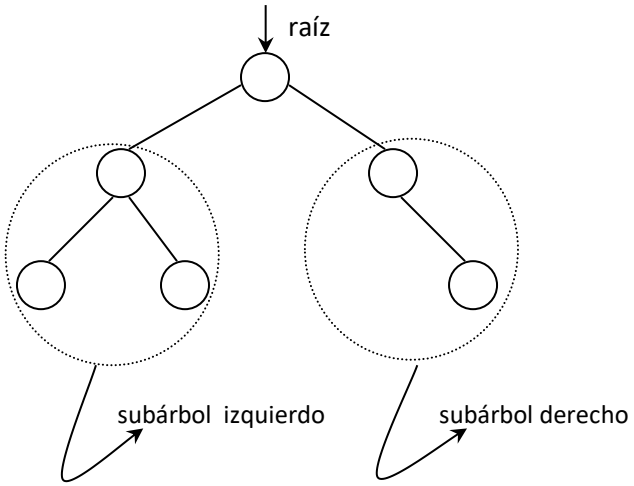
Un *árbol binario* es una estructura de datos no lineal, que puede estar vacía, o que puede contener un elemento (o *nodo*) designado como la *raíz* del árbol. Si el nodo raíz existe, de él se desprenden dos subconjuntos de elementos, con las siguientes características [1]:

a.) Ambos subconjuntos tienen intersección vacía: es decir, ningún nodo contenido en uno de los subconjuntos puede estar también en el otro.

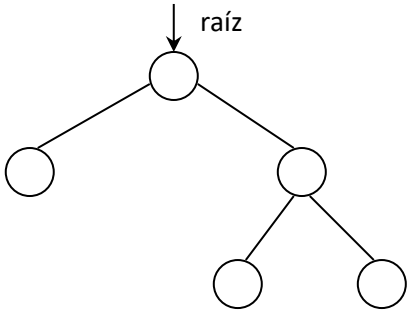
b.) Ambos subconjuntos son a su vez árboles binarios, y se designan respectivamente como *subárbol izquierdo* y *subárbol derecho*.

El siguiente gráfico muestra algunos ejemplos de figuras que representan árboles binarios, y figuras que no los representan pues no respetan en algún punto la definición anterior:

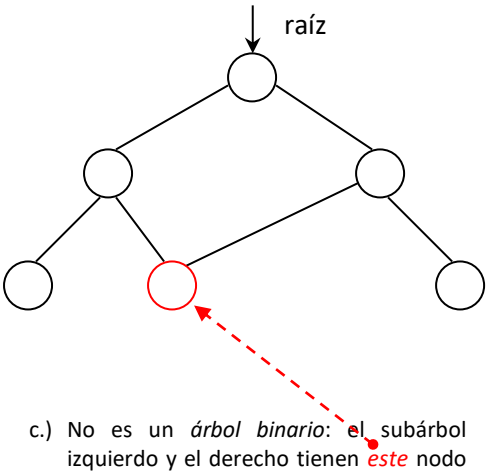
Figura 1: Ejemplos válidos y no válidos de árboles binarios.



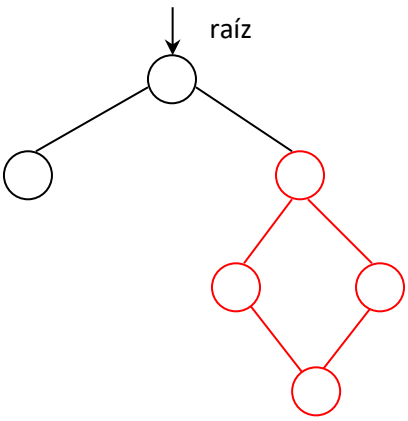
a.) Ejemplo correcto de *árbol binario*, marcando sus dos subárboles



b.) Otro ejemplo correcto



c.) No es un *árbol binario*: el subárbol izquierdo y el derecho tienen *este* nodo en común (no respeta la regla a).



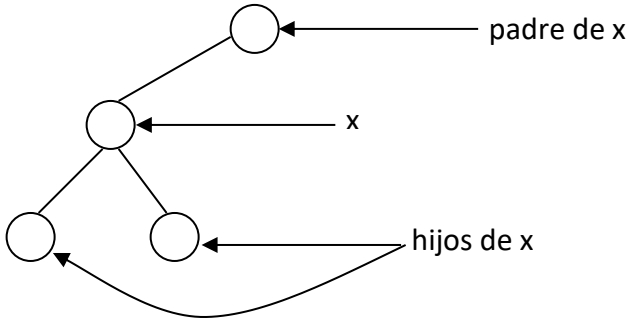
d.) No es *árbol binario*: el subárbol de la derecha no es un árbol binario, y por lo tanto no se cumple la regla b).

Observemos que la definición de *árbol binario* es recursiva, pues según la regla b), un árbol binario está compuesto *de otros dos árboles binarios*... Como vemos, el objeto que se quiere definir forma parte de la definición, y esta es la esencia de una definición recursiva. Esto debe ser tenido en cuenta a la hora de intentar procesar el árbol binario en forma completa, pues los algoritmos para hacerlo se aplican sobre el árbol principal primero, y luego recursivamente sobre cada uno de los subárboles.

En el trabajo con árboles binarios se maneja una terminología muy específica, aunque sencilla de definir y comprender. Mostramos una breve referencia a los principales términos y conceptos usados:

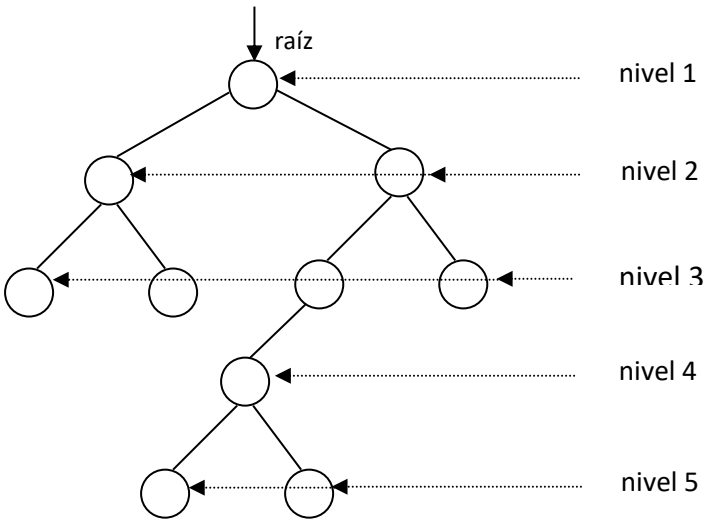
- ✓ Los enlaces o uniones entre los distintos nodos, se llaman *arcos*. Dependiendo de la forma en que se implemente el árbol esos *arcos* podrán ser referencias o punteros (o sea, variables que contienen una dirección de memoria) o podrían ser índices de un arreglo.
- ✓ Dado un nodo *x*, los dos nodos inmediatamente debajo de *x* (y unidos a él con sendos arcos) se llaman *hijos de x*. El nodo de cual depende *x* hacia arriba, se dice el *padre de x*. De las reglas dadas en la definición, se deduce que en un árbol binario ningún nodo puede tener más de un padre, y todo nodo puede tener dos hijos, un hijo, o ningún hijo. Tómese un tiempo para comprender el porqué de estas afirmaciones.

Figura 2: Padre, hijo izquierdo e hijo derecho de un nodo x.



- ✓ Se llama *nivel de un nodo*, a la cantidad de arcos que deben seguirse para llegar hasta ese nodo comenzando desde el arco para entrar a la *raíz*. El concepto de *nivel* da una idea formal de la división del árbol en "franjas horizontales" o "pisos". Así, el nodo *raíz* es alcanzable con un solo arco, y por lo tanto se dice que está en el *nivel 1 del árbol*. Los dos hijos del nodo *raíz* (si existen) son alcanzables con dos arcos cada uno, y por lo tanto se dice que ambos están en el *nivel 2 del árbol*, y así con el resto (ver figura siguiente).

Figura 3: Un árbol binario y sus niveles.



- ✓ Se llama *altura de un árbol* al número de nivel máximo del mismo. En el árbol de la figura anterior la altura es *h = 5*, pues el número máximo de nivel de ese árbol es *5*. Veremos más adelante, en ocasión del tema *árboles binarios de búsqueda*, que el control de la altura es un elemento muy importante a tener en cuenta en el rendimiento del tiempo de búsqueda en un árbol.
- ✓ La *implementación* de un *árbol binario* puede hacerse de manera que los nodos sean ellos mismos objetos de una clase *TreeNode*, que tendrá *tres atributos*: en uno de ellos (el atributo *info*) se guardará el valor (o la referencia al objeto) que se quiera almacenar en el nodo, y los otros dos serán referencias con las direcciones de los nodos hijos. Esas referencias a los hijos pueden valer *None* si el hijo correspondiente no existe.
- ✓ Luego, una clase *Tree* (o *SearchTree* si estamos ya pensando en árboles de búsqueda, como veremos) se puede usar para representar al árbol mismo, conteniendo una referencia *root* que se usará para contener la dirección del primer nodo del árbol. De hecho, lo único que se necesita declarar de antemano para trabajar con el árbol es la referencia *root*: los nodos serán creados con la función constructora de la clase *TreeNode* cada vez que sea requerido, dentro de algún método de inserción de la clase que representa al árbol. Veremos la forma de llevar a la práctica esta implementación en esta misma ficha.

3.] Árboles binarios de búsqueda.

Los *árboles binarios* son estructuras de datos aplicables en numerosas situaciones y problemas. Quizás la aplicación más obvia se da en el campo de la *búsqueda de una clave o valor en un conjunto*. Como vimos, si un conjunto de *n* claves está almacenado en un arreglo/lista, una búsqueda llevará podría llevar en el peor caso a *n* comparaciones (*O(n)*), pues en ese peor caso se deben comparar todas las claves[2]. Sin embargo, usando un *árbol binario* (y un poco de ingenio), se pueden lograr tiempos de búsqueda sustancialmente mejores: si el árbol se organiza de la manera adecuada, el tiempo de búsqueda puede bajarse a *O(log(n))*.

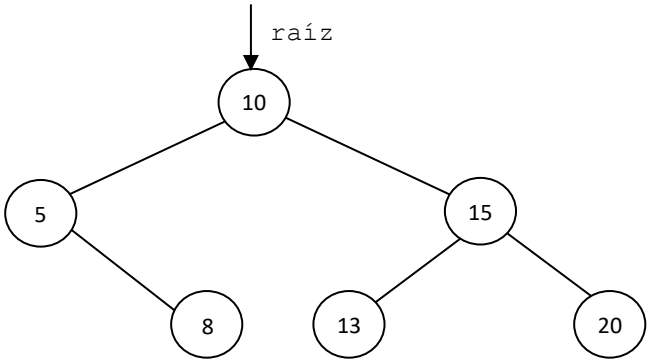
El secreto para lograr esto es tratar de organizar las claves de forma que al hacer una comparación, puedan desecharse otras claves sin tener que comparar contra ellas (esto es lo que hace, por caso, la *búsqueda binaria* en un arreglo ordenado: si al comparar contra un elemento del arreglo no se encuentra la clave, se sigue buscando a la derecha o a la izquierda según la clave sea menor o mayor, pero no se busca en todo el arreglo).

Con un árbol binario puede lograrse una organización de claves que favorezca una búsqueda rápida, siguiendo una regla sencilla: para insertar una clave *x*, se compara primero contra la clave del nodo raíz. Si *x* es mayor, se intenta insertar la clave en el *subárbol derecho*, pero si *x* es menor, se intenta la inserción en el *subárbol izquierdo*. Notemos que en este contexto, es común que si entrase una clave repetida (una clave que ya esté en el árbol), simplemente se ignore y se rechace la inserción (un árbol de búsqueda suele usarse para soporte de claves que se supone que no vendrán repetidas, aunque existen variantes simples para aceptar claves repetidas: una de ellas es modificar la regla para que los menores vayan a la izquierda y los mayores ***o iguales*** vayan a la derecha). En cada subárbol se repite la misma estrategia contra la raíz del subárbol, hasta que en algún momento se llegue a un camino nulo, y en ese

[2] Si el arreglo/lista estuviese ordenado la búsqueda llevaría un tiempo O(log(n)) en el peor caso, pues puede aplicarse búsqueda binaria, pero el arreglo debe estar ordenado. Y no siempre es práctico ordenar el arreglo para hacer una búsqueda, pues el ordenamiento lleva un tiempo O(n*log(n)) si se usa un algoritmo rápido.

lugar se inserta la clave. Un árbol binario creado con esta regla de *menores a la izquierda, mayores a la derecha*, se suele llamar *árbol binario de búsqueda*, o también *árbol ordenado*. El gráfico de la página siguiente muestra un árbol binario de búsqueda, en el cual las claves se insertaron en esta secuencia: {10, 15, 20, 5, 13, 8}.

Figura 4: Un árbol binario de búsqueda.



Como puede verse, para cada nodo del árbol se cumple que todos sus descendientes por la izquierda son menores que él, y todos sus descendientes por la derecha son mayores. De esta forma, buscar una clave en el árbol resulta una tarea sencilla y rápida: por ejemplo, si nos dieran a buscar la clave *x = 13*, deberíamos comenzar por la *raíz* y comparar contra el *info* de ella. Como el valor almacenado en la raíz es *10*, y *13* es mayor a *10*, la búsqueda sigue por el *hijo derecho* del nodo *raíz*, que es el *15*. Dado que ese nodo tampoco contiene al *13*, y *13* es menor que *15*, la búsqueda sigue por la izquierda del *15*, donde encontramos al *13*, y allí termina la búsqueda con éxito.

Si observamos, para buscar la clave no fue necesario mirar todos los nodos, sino solo uno en cada nivel del árbol. Esto reduce de manera dramática la cantidad de comparaciones a realizar, comparado contra una búsqueda secuencial en la que se revisan todos los nodos en el peor caso. Puede probarse que si el árbol cumple con ciertas condiciones de *balance* o *equilibrio*, el tiempo para buscar una clave en el peor caso es *O(log(n))*.

La secuencia de nodos que efectivamente fueron *vistos* al buscar la clave *x*, se llama *camino de búsqueda de la clave x*. En nuestro ejemplo, el camino de búsqueda de *x = 13* es el conjunto de nodos *{10, 15, 13}*.

4.] Implementación de árboles binarios de búsqueda.

En el proyecto *[F29] Árboles Binarios de Búsqueda* que acompaña a esta ficha, se implementa de forma básica la idea del *árbol binario de búsqueda*. En ese proyecto, el módulo *tree.py* contiene dos clases: una (llamada *TreeNode*) para representar a cada nodo del árbol, y la otra (llamada *SearchTree*) representa al árbol completo como colección. La clase *TreeNode* como dijimos, representa a cada nodo, y no ofrece mayor dificultad:

```
class TreeNode:
    def __init__(self, x, ls=None, rs=None):
        self.info = x
        self.left = ls
        self.right = rs
```

```
    def __str__(self):
        return str(self.info)
```

La clase dispone de tres atributos: El designado como *info* se usa para almacenar el valor que se quiere guardar en ese nodo (y por el cual el árbol será ordenado). Y los atributos *left* y *right* se usan para almacenar las direcciones o referencias a los nodos que serán, respectivamente, el hijo izquierdo y el hijo derecho de ese nodo en el árbol.

La función constructora *__init__()* de la clase (o simplemente, *el constructor de la clase*) crea un nodo con esos atributos y los inicializa adecuadamente. Observe que por default, los valores que llegan en los parámetros *ls* y *rs* son *None*: esto implica que un nodo podría tener sus atributos *left* y *right* valiendo *None*, lo cual muestra que un nodo podría no tener alguno de sus dos hijos. De hecho, un nodo puede tener dos hijos no nulos, o tener un solo hijo, o no tener ninguno (y en este caso, se lo designa como un *nodo hoja* o *nodo terminal*). Por su parte, el método *__str__()* solo retorna la conversión a cadena del valor contenido en el atributo *info*.

La clase *SearchTree* representa al árbol de búsqueda completo como colección de nodos. Como cada nodo del árbol es un objeto que contiene en sus atributos *left* y *right* las direcciones de sus hijos, resulta entonces que la clase *SearchTree* solo necesita recordar la dirección del primero de esos nodos, que es el nodo raíz. Por lo tanto, la clase mantiene un atributo *root* para almacenar justamente una referencia a ese nodo. Y para facilitar la tarea de saber cuántos nodos tiene el árbol en un momento dado, la clase mantiene un segundo atributo *count* para llevar la cuenta: cada vez que se agregue o se elimine un nodo, se sumará o se restará 1 a ese atributo. El constructor *__init__()* de la clase asigna en *root* el valor *None* (indicando con ello que un árbol arranca vacío al ser creado) y asigna 0 en *count* (lo cuál implica, obviamente, que el árbol arranca sin nodos).

La definición completa de la clase se muestra a continuación. Analizaremos más abajo los detalles de implementación de cada método:

```
class TreeNode:
    def __init__(self, x, ls=None, rs=None):
        self.info = x
        self.left = ls
        self.right = rs

    def __str__(self):
        return str(self.info)


class SearchTree:
    def __init__(self):
        self.root = None
        self.count = 0

    def __str__(self):
        return "| " + self.make_inorder(self.root) + "|"

    def preorder_travel(self):
        return "| " + self.make_preorder(self.root) + "|"

    def postorder_travel(self):
        return "| " + self.make_postorder(self.root) + "|"

    def is_empty(self):
```

```
        return self.root is None

    def add(self, x):
        if x is None:
            return False

        p = self.root
        q = None
        while p is not None:
            y = p.info
            if x == y:
                break
            q = p
            if x < y:
                p = p.left
            else:
                p = p.right

        # si x ya existía, no insertar y salir con False...
        if p is not None:
            return False

        # no existía... hay que insertar x...
        new = TreeNode(x)
        if q is None:
            self.root = new
        elif x < q.info:
            q.left = new
        else:
            q.right = new

        # contar el nuevo nodo...
        self.count += 1
        return True

    def remove(self, x):
        if x is None:
            return False

        ca = self.size()
        self.root = self.delete_node(self.root, x)
        return self.size() != ca

    def delete_node(self, p, x):
        if p is not None:
            y = p.info
            if x < y:
                menor = self.delete_node(p.left, x)
                p.left = menor
            else:
                if x > y:
                    mayor = self.delete_node(p.right, x)
                    p.right = mayor
                else:
                    # x encontrado... proceder a eliminarlo...
                    if p.left is None:
                        p = p.right
                    elif p.right is None:
                        p = p.left
                    else:
                        two = self.delete_with_two_children(p.left, p)
```

```
                    p.left = two

                self.count -= 1
        return p

    def delete_with_two_children(self, d, p):
        if d.right is not None:
            rs = self.delete_with_two_children(d.right, p)
            d.right = rs
        else:
            p.info = d.info
            d = d.left
        return d

    def contains(self, x):
        if x is None:
            return False

        p = self.root
        while p is not None:
            if x == p.info:
                return True

            elif x < p.info:
                p = p.left
            else:
                p = p.right

        return False

    def size(self):
        return self.count

    def make_inorder(self, p):
        r = ""
        if p is not None:
            r = self.make_inorder(p.left)
            r += str(p) + " "
            r += self.make_inorder(p.right)
        return r

    def make_preorder(self, p):
        r = ""
        if p is not None:
            r = str(p) + " "
            r += self.make_preorder(p.left)
            r += self.make_preorder(p.right)
        return r

    def make_postorder(self, p):
        r = ""
        if p is not None:
            r = self.make_postorder(p.left)
            r += self.make_postorder(p.right)
            r += str(p) + " "
        return r
```

El método *add()* comienza realizando una búsqueda del valor *x* en el árbol, pues en caso de estar ya en él, debe rechazar la inserción. La búsqueda se hace avanzando en el camino de

búsqueda de *x* usando una referencia *p*, que se ajusta a la izquierda o la derecha del nodo visitado según la clave sea menor o mayor que *x*. Notar que además de *p* se usa una segunda referencia *q* a modo de persecutor, para no perder la dirección del último nodo visitado por *p* cuando este se anule si no existe la clave. Al salir del ciclo, se verifica si *p* salió valiendo *None*, lo cual sólo puede ocurrir si *x* no existía en el árbol. En ese caso, se crea un nuevo nodo, asignando la clave *x* en él, y se "engancha" ese nodo en el árbol de la forma siguiente: si el árbol estaba vacío (lo cual se deduce del valor de la referencia persecutora *q* al terminar la búsqueda), el nuevo nodo se inserta como raíz. Y si el árbol no estaba vacío, se inserta a la izquierda o a la derecha de *q*, según sea el valor de la clave *x*. En ambos casos se suma 1 al atributo *count* (para llevar la cuenta de la cantidad de nodos que tiene el árbol en ese momento) y se finaliza retornando True (la inserción tuvo éxito). Notar que si la clave *x* estaba repetida, el algoritmo presentado simplemente ignora el hecho, y termina retornando *False* sin realizar otra acción adicional.

El método *contains()* toma una clave *x* que entra como parámetro y simplemente recorre su camino de búsqueda para determinar si está o no en el árbol. El método de inserción también hacía una búsqueda antes de insertar el nodo, pero en esa función la búsqueda se hacía con un puntero/referencia de persecución para no perder la dirección del nodo padre del que será insertado. En el método *contains()*, solo se pretende verificar si la clave existe, y por ello el persecutor no es necesario. El método devuelve *True* si es que *x* existe en el árbol, o devuelve *False* si *x* no existe.

El método *size()* simplemente informa cuántos nodos tiene el árbol, y para ello solo tiene que retornar el valor del atributo *count*. El método *is_empty()* retorna informa si el árbol está vacío o no, para lo cual solo tiene que chequear el valor del atributo *root*: si este es *None*, entonces no está apuntando a nodo alguno y por lo tanto el árbol está vacío y el método retorna *True*. Obviamente, retornará *False* si *root* es diferente de *None*.

El resto de los métodos provistos serán explicados en los apartados que siguen en esta misma ficha.

5.] Eliminación (o borrado) de un valor en un árbol binario de búsqueda.

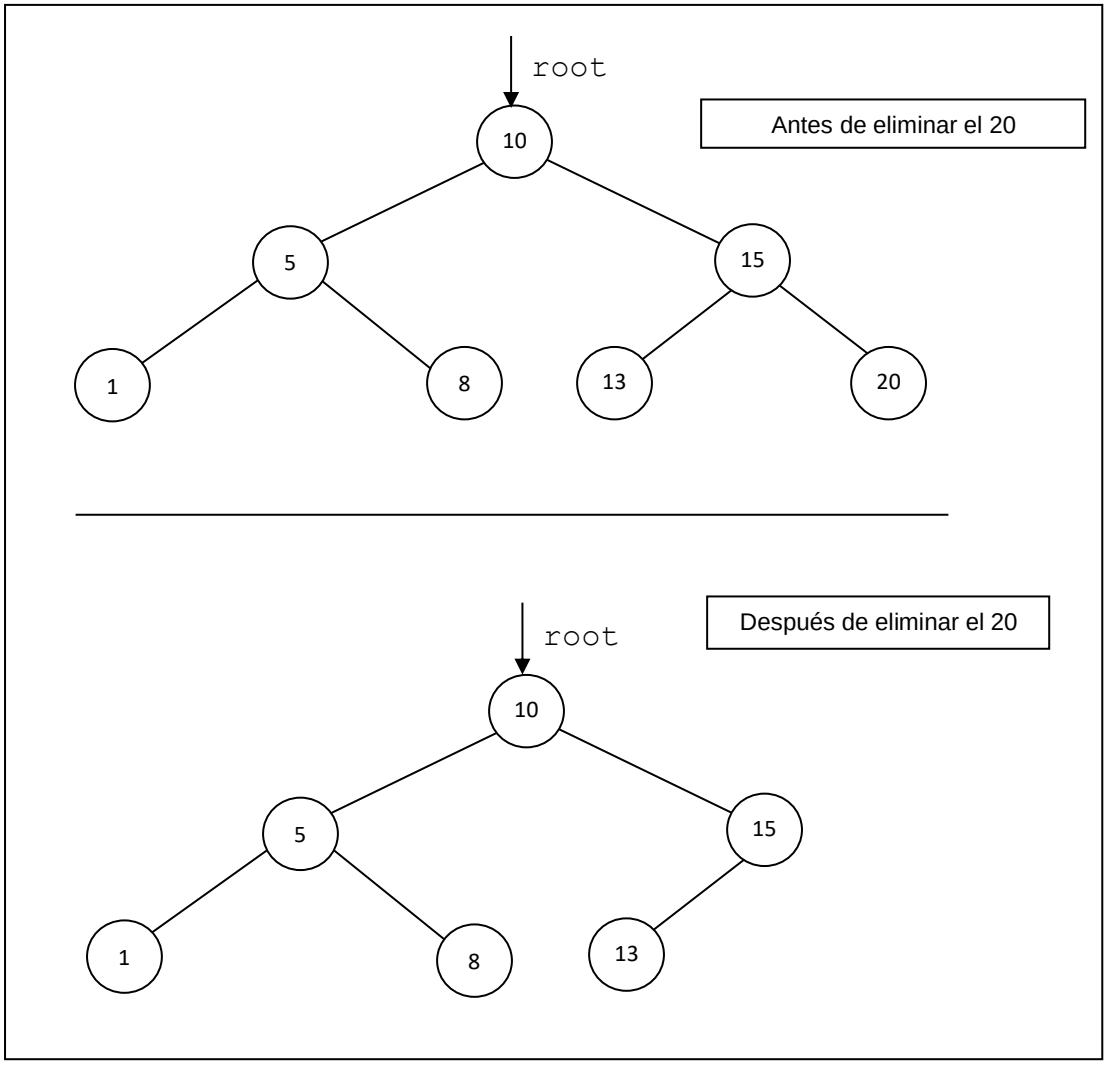
La tarea de gestionar un árbol de búsqueda (o cualquier otra estructura de datos) no estaría completa si no se incluyen métodos para permitir la eliminación de elementos del árbol. Nos proponemos desarrollar un método que tome como parámetro una clave *x*, la busque en el árbol, y en caso de encontrarla elimine al nodo que la contiene, haciendo que el árbol (obviamente) siga siendo un árbol y siga siendo de búsqueda. Un rápido análisis del problema sugiere que hay que prestar atención a varios casos, y que no todos ellos son triviales [1]:

a. El nodo a borrar podría ser una hoja (es decir, un nodo sin hijos).
b. El nodo a borrar podría tener un (y solo un) hijo, sin importar de qué lado.
c. El nodo a borrar podría tener exactamente dos hijos.

El primer caso es simple: solo debemos ubicar el nodo a borrar y poner en *None* el enlace con su padre. Como el nodo borrado no tiene hijos, no debemos preocuparnos por reenganchar sus sucesores. La ***Figura 5*** de la página siguiente muestra un árbol de búsqueda antes y

después de eliminar el nodo que contiene el valor 20, que es una hoja. El nodo con valor 15 queda con su hijo derecho nulo.

Figura 5: Eliminación de un nodo sin hijos.

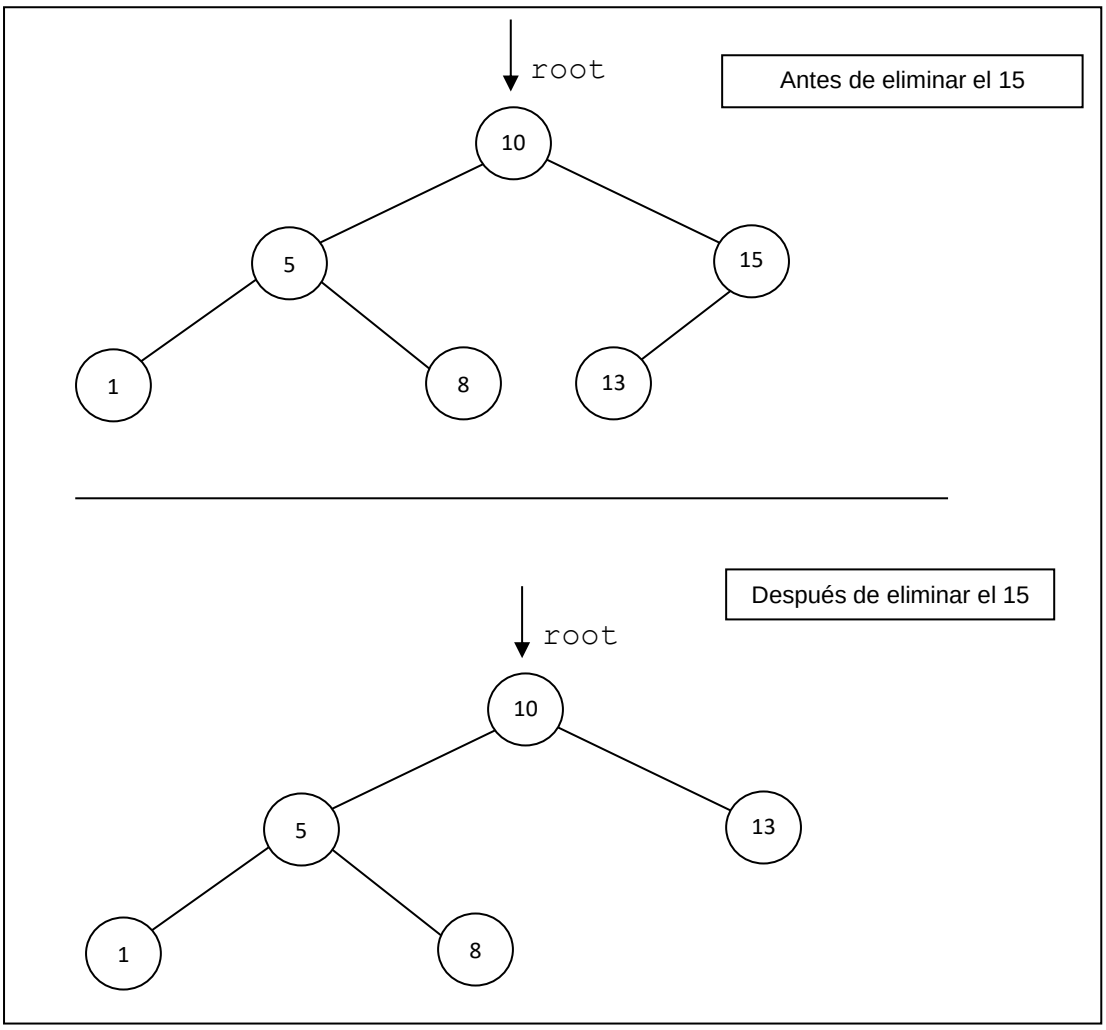


El segundo caso es también simple (en los hechos equivale a eliminar un nodo en una lista): se ubica el nodo a borrar y se ajusta el enlace del padre de forma que ahora apunte al único hijo[3] del nodo que se quiere borrar. La idea es directa: si borramos un nodo con un único hijo, hacemos que ese hijo ocupe el lugar del padre eliminado (ver ***Figura 6*** *en página siguiente*).

[3] Si se trata de relaciones padre-hijo, la película *Frequency* (del año 2000, dirigida por *Gregory Hoblit* e interpretada por *Dennis Quaid*) nos muestra la historia del detective John Sullivan que no puede superar la muerte repentina de su padre (que era bombero) en un incendio 30 años antes. Un día Sullivan encuentra un equipo de radioaficionado que era de su padre, y al comenzar a usarlo descubre que por algún extraño efecto causado por una aurora boreal, puede increíblemente hacer contacto con su padre a través de ese equipo, en tiempos diferentes con 30 años de separación. Sullivan salva la vida de su padre avisándole sobre el incendio… Y esto, por supuesto, cambia la línea temporal… provocando una sucesión de hechos inesperados de todo tipo y color… Tendrán que verla para saber… ☺.

El tercer caso ya no es tan simple: el nodo a borrar tiene dos hijos y no es posible que ambos sean apuntados desde el padre del que queremos borrar. En este caso, la idea es reemplazar al nodo que se quiere borrar por otro, tal que este otro sea fácilmente suprimible de su lugar original, pero que obligue a la menor cantidad posible de operaciones en el árbol.

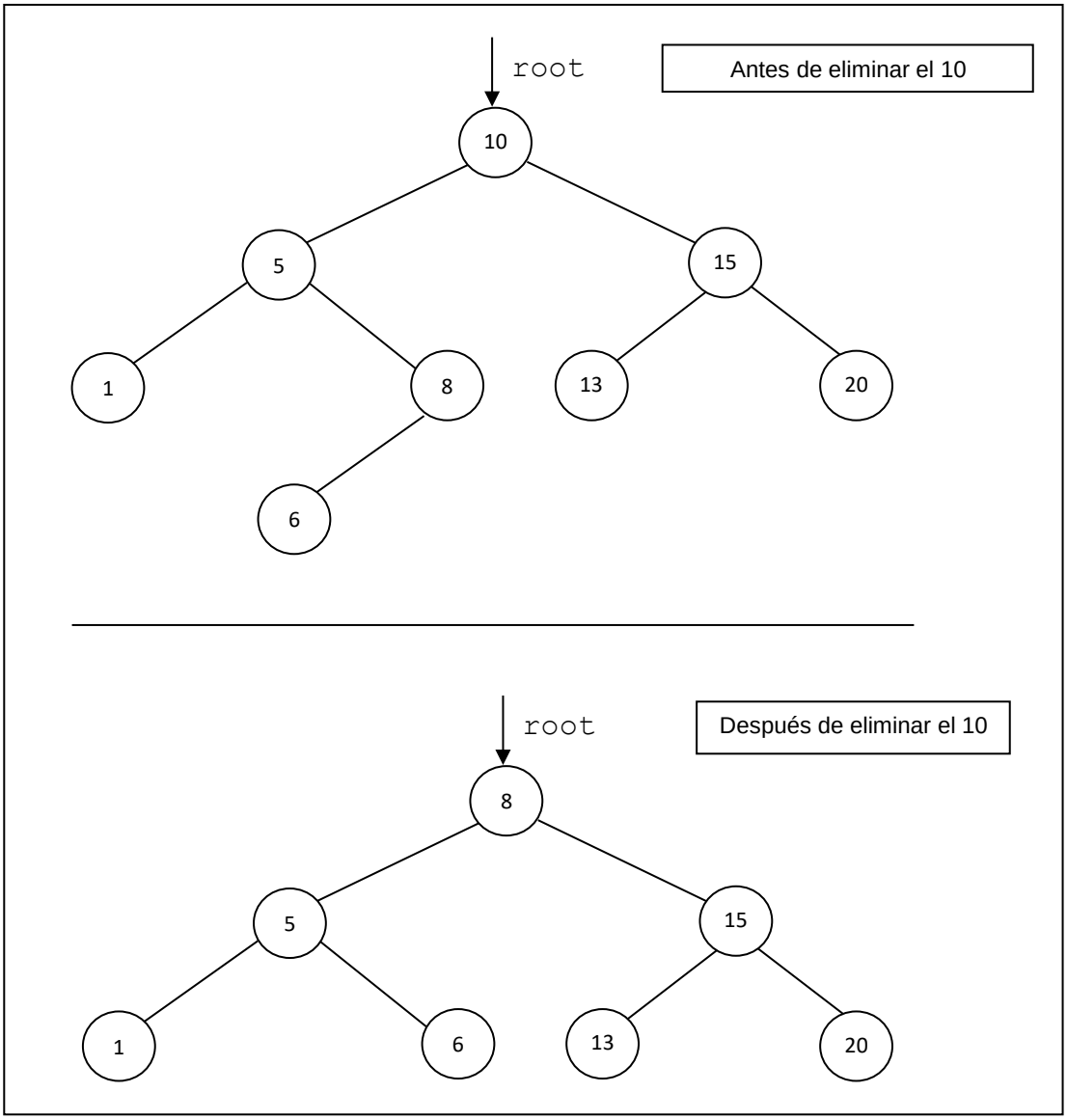
Figura 6: Eliminación de un nodo con un único hijo.



Si observamos bien, ese nodo puede ser el *mayor descendiente izquierdo* del que queremos borrar (o el *menor descendiente derecho*). En el árbol de la ***Figura 7***, si queremos eliminar el nodo que contiene al 10 podemos reemplazarlo por el que contiene al 8 (que es el mayor de todos los que son menores que el 10).

Eliminar del árbol a ese mayor descendiente izquierdo es simple, pues o bien tendrá un solo hijo a la izquierda, o no tendrá hijos (no puede tener un hijo a la derecha, pues si lo tuviera ese hijo sería mayor que él y no podría entonces ser elegido como el *mayor* descendiente izquierdo). En el caso de la ***Figura 6***, el nodo que contiene al 8 no tiene hijos, y por lo tanto bastaría con poner en *None* el enlace derecho del que contiene al 5. Si el 8 tuviera un hijo a la izquierda, ese hijo quedaría a la derecha del 5, reemplazando al 8. La ***Figura 7*** de la página siguiente ilustra lo dicho:

Figura 7: Eliminación de un nodo con dos hijos.



En el modelo *[F29] Árboles Binarios de Búsqueda* que acompaña a esta ficha, la clase *SearchTree* del módulo *tree.py* incluye un método *remove()* para eliminar un nodo del árbol aplicando las ideas vistas, restando uno al atributo *count* y retornando *True* si la eliminación efectivamente tuvo éxito. El método elimina el nodo con valor *x* del árbol, y usa dos métodos auxiliares. El primero (*delete_node()*) busca recursivamente el nodo que contiene a *x*, y determina si tiene uno o ningún hijo. En cualquiera de los dos casos, procede a eliminarlo. Pero si el nodo tiene dos hijos, invoca a otro método (*delete_with_two_children()*) que es el que finalmente aplica el proceso de reemplazar a ese nodo por su mayor descendiente izquierdo, recursivamente. Se deja su análisis para el estudiante.

6.] Recorrido completo de árboles binarios.

Una vez que un árbol binario ha sido creado (sea o no de búsqueda), tarde o temprano surge el problema de tener que recorrerlo en forma completa. Por ejemplo, si se pide obtener la

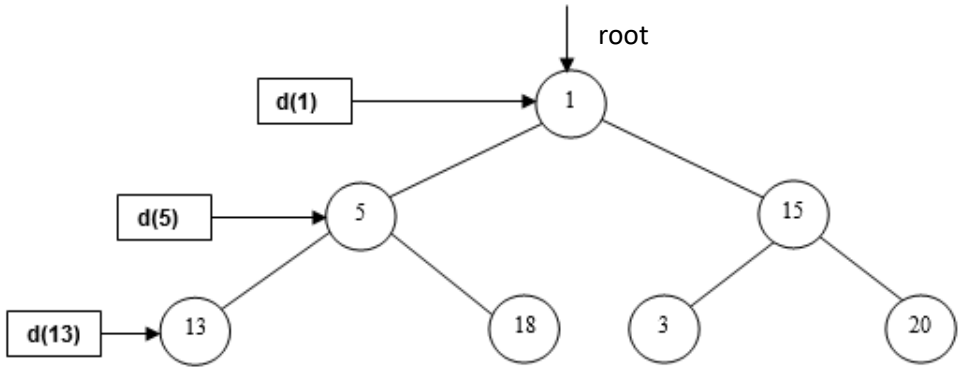
conversión a cadena del árbol completo, el método *__str__()* debería hacer un recorrido nodo a nodo de todo el árbol (y no sólo el camino de búsqueda de una clave).

El proceso de recorrido de un árbol binario no es tan sencillo como el de recorrer una lista o un arreglo unidimensional, simplemente porque un árbol *no es una estructura lineal*: cada nodo puede tener más de un sucesor, y por lo tanto a partir de un nodo hay varios caminos posibles por los que puede seguir el recorrido. En general, los problemas a los que se enfrenta un programador a la hora de desarrollar un método para recorrer un árbol binario pueden resumirse en los siguientes [1]:

- ✓ En cada nodo del árbol pueden abrirse dos caminos diferentes, y se debe decidir por cuál seguir.
- ✓ Sin embargo, una vez decidido el camino a seguir (a la izquierda o a la derecha), en alguna parte debe guardarse la dirección del nodo que se abandona, para luego poder volver a él y recorrer el otro camino. Por ejemplo, si estando en un nodo ***x*** se decide seguir por el subárbol izquierdo de ***x***, debemos guardar la dirección de ***x*** en algún lugar, pues de otro modo al terminar de recorrer el subárbol izquierdo no podríamos volver a ***x*** para seguir con el derecho. Este problema se repite con cada nodo del árbol, y no solo con la raíz.
- ✓ Finalmente, queda por determinar en qué momento se detiene el recorrido. En un árbol binario cada rama termina en *None* y por ello encontrar en un árbol una dirección None no es señal de haber terminado el proceso.

Analizando con cuidado los tres problemas recién planteados, vemos que en realidad es el segundo de ellos el más complicado: ¿dónde (y cómo) guardar las direcciones de los nodos que se abandonan al decidir un recorrido por la izquierda o por la derecha? Como son muchos los nodos en los cuales ese problema se presentará, es obvio que debemos usar una estructura de datos para ir almacenando esas direcciones. En principio cualquiera podría ser útil: una lista, un arreglo, una cola, una pila... Pero la estructura de datos elegida debe facilitar al máximo el proceso de guardado y recuperación de una dirección, para no complicar inútilmente el algoritmo de recorrido del árbol.

Figura 8: Esquema de recorrido de un árbol binario.

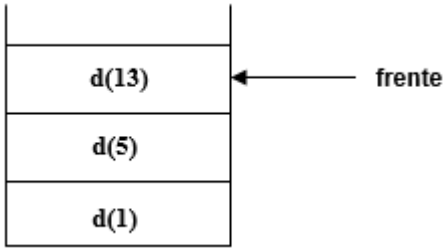


Mostramos en la ***Figura 8*** anterior un proceso simplificado de un recorrido de árbol binario (en este caso, el árbol dibujado no es de búsqueda, pero eso no es importante), para tratar de deducir en qué estructura de datos debemos apoyarnos para guardar las direcciones. Supongamos que el proceso de recorrido comienza con el nodo raíz, cuyo valor es 1. Llamemos *d(1)* a la dirección del nodo que contiene al valor 1 (en la práctica, esa dirección será

almacenada en una referencia/puntero usada para recorrer el árbol). Luego de procesar el nodo *d(1)*, supongamos que se decide seguir el recorrido por la izquierda. Por lo tanto, antes de saltar al nodo *d(5)*, la dirección *d(1)* debe guardarse en alguna parte. Del mismo modo, al terminar con *d(5)* debe guardarse esa misma dirección para poder saltar a *d(13)*.

Una buena idea es que las direcciones de los nodos "abandonados" se vayan almacenando en una *pila*, pues de esta forma la última dirección guardada en la pila queda encima de la misma, y es la primera que será recuperada, permitiendo volver al último nodo que se procesó antes de bajar por esa rama. Para nuestro ejemplo, en esa pila se almacenaría primero la dirección *d(1)*, luego la dirección *d(5)* y finalmente *d(13)*, quedando como se muestra en la ***Figura 9***:

Figura 9: Esquema de una pila conteniendo las direcciones de los nodos visitados en el recorrido.



¿Por qué es útil una pila? Pues bien: al terminar de procesar el nodo *d(13)* ya no es posible seguir "bajando" por el árbol, pues *d(13)* no tiene descendientes. Por lo tanto, se debe volver al nodo padre de *d(13)*, que es *d(5)*, y desde allí seguir el recorrido hacia la derecha del padre. El acceso a la dirección del padre de *d(13)* debería ser inmediato y simple: no deberíamos tener que "buscar" esa dirección, sino que la misma tendría que estar disponible en orden constante [*O(1)*]. Y eso es justamente lo que garantiza una pila al invertir la secuencia de entrada: antes de bajar a *d(13)*, la última dirección guardada en pila será *d(5)*. Por lo tanto, al terminar con *d(13)* nos quedará disponible *d(5)* en forma inmediata.

En este punto, quizás el lector estará comenzando a sentirse incómodo al pensar que deberá programar una pila para guardar direcciones, y luego usar esa pila en un algoritmo que sea capaz de recorrer un árbol binario. Si bien es cierto que semejante algoritmo es una muy buena práctica de programación, veremos que no será necesario tomarse tanto trabajo para gestionar la pila de direcciones, pues conocemos una técnica que puede manejar automáticamente una pila para facilitar la tarea del programador, sin siquiera tener que declarar esa pila: esa técnica es la *recursividad* [2]:.

Si un problema puede plantearse de forma *recursiva*, entonces también puede escribirse una función o un método recursivo que lo resuelva. En el caso que nos ocupa, el problema es el *recorrido de un árbol binario*, y existen al menos tres formas clásicas de plantear ese problema en forma recursiva. Las tres estrategias son:

- Recorrido en Preorden (u *orden previo*)
- Recorrido en Entreorden (u *orden central*)
- Recorrido en Postorden (u *orden posterior*)

En general, cuando se recorre una estructura de datos cualquiera, el proceso que se aplica a cada nodo o componente de esta suele designarse como *visitar* el nodo. Con esta aclaración, el recorrido en *Preorden* se define de la siguiente forma recursiva:

Recorrido en Preorden

1.) Visitar el nodo raíz.
2.) Recorrer en ***Preorden*** el subárbol izquierdo.
3.) Recorrer en ***Preorden*** el subárbol derecho.

Puede verse claramente que la definición anterior es *recursiva*: el concepto que se está definiendo (*recorrido en preorden*) vuelve a nombrarse en la propia definición (en los pasos 2 y 3). Si bien parece un poco concisa, la definición dice simplemente que si se quiere recorrer en *preorden* un árbol binario, comience procesando el nodo raíz, y luego aplique el *mismo* método de recorrido para el árbol de la izquierda y para el árbol de la derecha. Esto es: en el árbol de la izquierda, aplique los tres pasos desde el principio: procesar la raíz, recorrer en *preorden* el árbol izquierdo y luego el derecho, y así con cada nodo del árbol.

Veamos cómo sería recorrido el árbol de la ***Figura 8*** aplicando esta técnica, suponiendo que la visita del nodo consiste en mostrar en pantalla su valor contenido:

- El recorrido comienza con el nodo raíz del árbol, y muestra el valor de este: muestra el valor 1.
- Luego, se aplica el paso 2 que indica "*recorrer en preorden el subárbol izquierdo*" del 1. Por lo tanto, el proceso se "traslada" al subárbol cuya raíz es 5, y en ese nodo se aplica un *preorden*. Es decir, se vuelven a aplicar sobre ese nodo los tres pasos de la definición:
  - Visitar raíz: muestra el valor 5.
  - *Preorden en el subárbol izquierdo* del 5: trasladar el proceso al nodo 13, y volver a empezar todo:
    - Visitar raíz: muestra el valor 13.
    - *Preorden en el subárbol izquierdo* del 13: aquí volvería a empezar todo, pero este subárbol está vacío, por lo tanto, termina sin efectuar ninguna operación, y vuelve al nodo 13.
    - *Preorden en el subárbol derecho* del 13: también está vacío, y el proceso termina sin efectuar acción alguna.
  - *Preorden en el subárbol derecho* del 5: trasladar el proceso al nodo 18 y volver a empezar todo:
    - Visitar raíz: muestra el valor 18.
    - *Preorden en el subárbol izquierdo* del 18: aquí volvería a empezar todo, pero este subárbol está vacío, por lo tanto, termina sin efectuar ninguna operación, y vuelve al nodo 18.
    - *Preorden en el subárbol derecho* del 18: también está vacío, y el proceso termina sin efectuar acción alguna.
- Ahora estamos de nuevo en el nodo 1, habiendo recorrido y mostrado el contenido de todo el subárbol izquierdo. El mismo proceso se repite ahora, haciendo un *preorden en el subárbol derecho* (dejamos el seguimiento de este para el lector).

Ahora bien: un análisis detallado de la secuencia de recorrido en *preorden*, revela que cada nodo del árbol será *tocado* tres veces durante el recorrido: la primera vez, cuando se llega al nodo desde *arriba*, la segunda vez cuando se vuelve *desde el subárbol izquierdo*, y la tercera

cuando se vuelve *desde el subárbol derecho*. Pero será procesado (*visitado*) en solo una de esas ocasiones. Y allí está la diferencia entre los tres tipos de recorrido:

- ✓ En el *preorden*, cada nodo se visita cuando se llega a él por *primera vez*.
- ✓ En el *entreorden*, cada nodo se visita cuando se vuelve a él *desde la izquierda*.
- ✓ En el *postorden*, cada nodo se visita cuando se vuelve a él *desde la derecha*.

Por lo tanto, las definiciones de los recorridos que faltan pueden ser las siguientes:

Recorrido en Entreorden

1.) Recorrer en ***Entreorden*** el subárbol izquierdo.
2.) Visitar el nodo raíz.
3.) Recorrer en ***Entreorden*** el subárbol derecho.

Recorrido en Postorden

1.) Recorrer en ***Postorden*** el subárbol izquierdo.
2.) Recorrer en ***Postorden*** el subárbol derecho.
3.) Visitar el nodo raíz.

Podemos ver que la secuencia que saldría en pantalla con cada una de las tres estrategias, para el árbol de la ***Figura 8***, sería la siguiente (tenga cuidado el lector de hacer un seguimiento estricto de cada algoritmo para asegurarse que logra los resultados que se muestran):

Recorrido en Preorden: { 1, 5, 13, 18, 15, 3, 20 }
Recorrido en Entreorden: { 13, 5, 18, 1, 3, 15, 20 }
Recorrido en Postorden: { 13, 18, 5, 3, 20, 15, 1 }

Finalmente, mostramos (a nivel de esquema) un método *display()* que solo invoca a otro método *mostrar_preorden()* (ambos podrían estar contenidos en la clase *SearchTree,* aunque al mostrarlos solo como un esquema, no los hemos incluido en ella) tal que este último recorre en forma recursiva un árbol binario y muestra el contenido de este en *preorden* (prescindiendo de detalles de interfaz). Cuando *display()* se activa, solo invoca a *mostrar_preorden()* y le envía el valor de *self.root* como parámetro. Dejamos para los estudiantes la tarea de incluir estos métodos en la clase *SearchTree* y probar su funcionamiento.

```
def display(self):
    self.mostrar_preorden(self.root)

def mostrar_preorden (self, p):
    if p is not None:
        print(p.info)
        mostrar_preorden(p.left)
        mostrar_preorden(p.right)
```

El método *mostrar_preorden()* anterior toma como parámetro una referencia a la raíz del árbol que se quiere recorrer, verifica si esa referencia está apuntando a algún nodo (pues en caso contrario el árbol está vacío y no hay recorrido posible) y en caso afirmativo lanza los tres pasos del recorrido en preorden, tal como los indica la definición. Los dos últimos pasos, son llamadas recursivas al propio método *mostrar_preorden()*.

Uno de los temas que más dificultad suele presentar cuando se escribe un algoritmo recursivo es encontrar la condición de corte de ese algoritmo. Una buena estrategia es identificar en qué casos el problema que se quiere resolver es trivial, y preguntar por ese caso. Pues bien: ¿en qué caso es trivial el recorrido de un árbol binario? Respuesta: cuando el árbol está vacío, pues en ese caso no hay acción alguna que realizar. Por ello, el método comienza preguntando si la referencia de entrada es distinta de *None* y sólo en ese caso procede al recorrido.

En este caso, se supone que la visita que debe realizarse a cada nodo consiste en mostrar su *info* por pantalla. En el recorrido en *preorden*, la visita se hace *antes* de los recorridos de los subárboles. Esos recorridos son realizados por el mismo método *mostrar_preorden()*, que se invoca a sí mismo dos veces luego de la visita al nodo *p*: en la primera invocación toma como parámetro el puntero/referencia al hijo izquierdo de *p* (con lo cual recorrerá ese árbol aplicando de nuevo todo el algoritmo), y en la segunda toma como parámetro la referencia al hijo derecho de *p*, con el mismo efecto.

Introduciendo pequeños cambios se pueden obtener los recorridos en *entreorden* y *postorden*, simplemente moviendo la instrucción que hace la visita del nodo. Aquí van los planteos de cada método recursivo (dejamos la modificación del método *display()* de cada uno, para el estudiante):

```
def mostrar_entreorden(self, p):
    if p is not None:
        mostrar_entreorden(p.left)
        print(p.info)
        mostrar_entreorden(p.right)

def mostrar_postorden(self, p):
    if p is not None:
        mostrar_postorden(p.left)
        mostrar_postorden(p.right)
        print(p.info)
```

Como ya dijimos, la *recursividad* se encarga de gestionar la pila necesaria para almacenar las direcciones de los nodos que se abandonan durante el recorrido. La idea es que al llamar a uno de estos método, cada vez que se invoca a sí mismo se almacena en el *stack segment* el valor del parámetro *p*. Como el *stack segment* es en los hechos un segmento de memoria que se accede en modo *LIFO*, se comporta como una *pila*. Cada vez que el método termina su ejecución actual, se retiran del *stack segment* los valores colocados por ese método, y quedan disponibles los que se encontraban inmediatamente debajo, que resultan ser los valores correspondientes al nodo padre del que se está procesando. Todos los pasos que se mostraron para explicar el recorrido del árbol de la ***Figura 8***, son realizados también ahora pero de forma implícita por la *recursividad*.

Cabe una observación más, por demás interesante: si el árbol binario que se está recorriendo *es de búsqueda* (la función de inserción graba con la regla *menores a la izquierda y mayores a la derecha*), entonces el recorrido en *entreorden* propuesto para el método *mostrar_entreorden()* provocará que la secuencia de valores que se muestra en pantalla salga *ordenada*. Esto es un resultado que no depende de la suerte de quien cargue los datos: siempre que el árbol sea de búsqueda, el recorrido en *entreorden* visitará los nodos en secuencia ordenada. ¿Por qué? Es lógico si se piensa en qué hace el recorrido en *entreorden*: primero procesa todos los nodos del subárbol izquierdo (que son menores que el valor de la raíz), luego procesa la raíz, y finalmente procesa los nodos del subárbol derecho (que son

mayores que el valor de la raíz). Como en cada subárbol procede recursivamente igual, entonces cada subárbol es también recorrido *de menor a mayor*. El resultado general es una secuencia ordenada de visitas.

Todos los métodos de recorrido aquí planteados tienen *precedencia izquierda*: primero se recorre el subárbol izquierdo, y luego se recorre el derecho. Sin embargo, esto no es obligatorio. Los mismos algoritmos pueden plantearse con *precedencia derecha* con mucha facilidad, simplemente invirtiendo el orden de las llamadas recursivas:

```
# Preorden con precedencia derecha...
def mostrar_preorden (self, p):
    if p is not None:
        print(p.info)
        mostrar_preorden(p.right)
        mostrar_preorden(p.left)

# Entreorden con precedencia derecha...
def mostrar_entreorden(self, p):
    if p is not None:
        mostrar_entreorden(p.right)
        print(p.info)
        mostrar_entreorden(p.left)

# Postorden con precedencia derecha...
def mostrar_postorden(self, p):
    if p is not None:
        mostrar_postorden(p.right)
        mostrar_postorden(p.left)
        print(p.info)
```

Como es obvio, si el árbol es de búsqueda con regla *menores a la izquierda, mayores a la derecha*, entonces si se invierte la precedencia y se recorre el árbol en entreorden el resultado será una secuencia también ordenada, pero ahora de mayor a menor... Por supuesto, el mismo resultado podría lograrse cambiando la regla de inserción, haciendo: *mayores a la izquierda, menores a la derecha* y luego recorriendo el árbol en entreorden con precedencia izquierda.

En nuestra clase *SearchTree*, el método *__str__()* retorna una cadena con el contenido del árbol recorrido en entreorden (con ***precedencia izquierda***) por lo que los elementos aparecerán en orden de menor a mayor en esa cadena. El recorrido recursivo es realizado por el método *make_inorder()*:

```
def __str__(self):
    return "| " + self.make_inorder(self.root) + "|"

def make_inorder(self, p):
    r = ""
    if p is not None:
        r = self.make_inorder(p.left)
        r += str(p) + " "
        r += self.make_inorder(p.right)
    return r
```

La clase incluye otros dos métodos públicos *preorder_travel()* y *postorder_travel()* (no estándar ni de uso automático como *__str__()*) que permiten obtener cadenas en base a recorridos en preorden y en postorden, y también se usan dos métodos auxiliares *make_preorder()* y *make_postorder()* para hacer

los recorridos en forma recursiva (y de nuevo, con ***precedencia izquierda***). Dejamos su análisis para el estudiante:

```
    def preorder_travel(self):
        return "| " + self.make_preorder(self.root) + "|"

    def postorder_travel(self):
        return "| " + self.make_postorder(self.root) + "|"

    def make_preorder(self, p):
        r = ""
        if p is not None:
            r = str(p) + " "
            r += self.make_preorder(p.left)
            r += self.make_preorder(p.right)
        return r

    def make_postorder(self, p):
        r = ""
        if p is not None:
            r = self.make_postorder(p.left)
            r += self.make_postorder(p.right)
            r += str(p) + " "
        return r
```

El proyecto *[F29] Árboles Binarios de Búsqueda* incluye un pequeño programa principal para testear en forma rápida y simple la funcionalidad de la clase SearchTree:

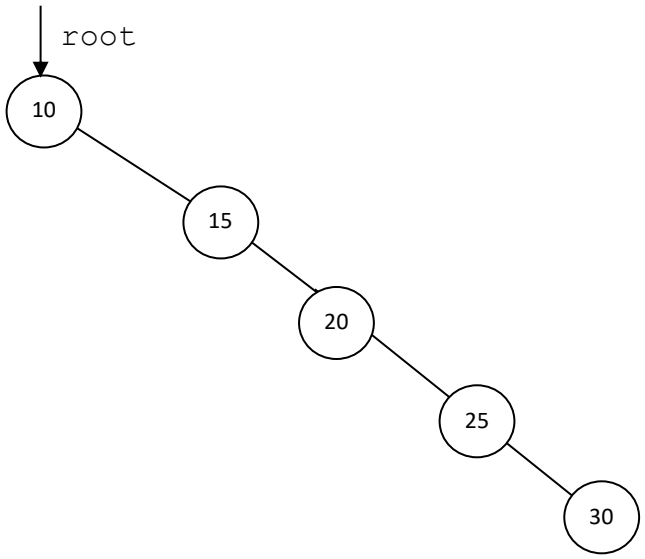
```
from tree import *


def main():
    a = SearchTree()
    a.add(6)
    a.add(26)
    a.add(3)
    a.add(16)
    a.add(11)
    a.add(1)

    x = 16
    print("Contenido del árbol (Entreorden):", a)
    print("Contenido del árbol (Preorden):", a.preorder_travel())
    print("Contenido del árbol (Postorden):", a.postorder_travel())

    print("Cantidad de nodos:", a.size())
    print("¿Está vacío el árbol?:", a.is_empty())
    print("¿Está contenido el valor", x, "en el árbol?:", a.contains(x))

    a.remove(6)
    print("Contenido del árbol (Entreorden) luego de una eliminación:", a)


if __name__ == '__main__':
    main()
```

7.] Consideraciones finales.

El algoritmo de inserción visto para árboles de búsqueda es fuertemente dependiente de la secuencia en la que vienen las claves a ser insertadas. El mismo algoritmo puede producir

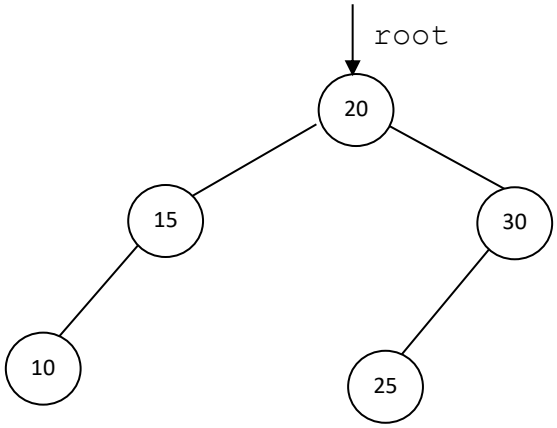
árboles cuyas formas sean completamente diferentes, de acuerdo a cómo vengan las claves. Por ejemplo, si entregamos las claves en esta secuencia: {10, 15, 20, 25, 30} el algoritmo de inserción generará el siguiente árbol, cuya altura es h = 5, igual al número de nodos del árbol (ver la ***Figura 10*** siguiente):

Figura 10: Un árbol de búsqueda desbalanceado.



El árbol es de búsqueda, pero está organizado de forma tan desbalanceada que ha degradado en una estructura lineal simple... Pero el caso es que si pretendemos buscar una clave en este árbol, nuestro tiempo de búsqueda en el peor caso (si queremos buscar el valor 30 por ejemplo) será O(n) (tendremos que hacer n = 5 comparaciones) y ya no O(log(n)). ¿Cuál es el problema con este árbol? ¿Qué entendemos por *desbalance*? Intuitivamente, podemos ver que el árbol anterior tiene su altura mucho mayor a la que se podría tener si organizáramos los nodos de otra forma. Las mismas claves podrían producir este otro árbol (de altura h = 3) si fueran ingresadas en esta otra secuencia: {20, 15, 10, 30, 25} (ver la ***Figura 11*** siguiente).

Figura 11: Un árbol de búsqueda balanceado.



En este árbol, ubicar una clave llevará como máximo tres comparaciones, mientras que en el anterior tendremos cinco. Como puede verse, la altura del árbol determina la cantidad máxima de comparaciones que tendremos que realizar. Se puede probar que si esa altura fuera la ***mínima posible***, entonces el tiempo de búsqueda en el peor caso es O(log(n)), pues

las claves estarían distribuidas aproximadamente por mitades entre cada subárbol, sin dejar niveles desocupados. Pero si permitimos que la altura del árbol crezca de manera indiscriminada, ya no tendremos un tiempo de orden logarítmico para el peor caso en una búsqueda y nos aproximaremos al tiempo de orden lineal típico de las listas o los arreglos unidimensionales.

Sin entrar en mayores detalles, intuitivamente podemos asumir que mientras menor sea la altura de un árbol de búsqueda, menor será el tiempo de respuesta en una búsqueda, incluso para el peor caso. Es decir: *mientras más se aproxime la altura del árbol a la mínima altura posible para el número de nodos que tiene, más balanceado estará ese árbol, y el tiempo se búsqueda en el peor caso tenderá a O(log(n))*. El concepto de balance (o equilibrio) de un árbol binario es más complejo y más formal para definir, pero aquí nos daremos por cumplidos con esa idea intuitiva, pues el desarrollo completo de estos temas escapa al alcance de este curso.

Obviamente, una implementación correcta de la estructura de árbol binario de búsqueda no puede depender del azar para lograr un árbol bien balanceado, que favorezca búsquedas rápidas. La solución para garantizar que el tiempo de búsqueda en el peor caso sea de tiempo O(log(n)) consiste (como era de esperar) en modificar el algoritmo de inserción para que el mismo algoritmo no solo inserte un nuevo nodo, sino que también compruebe si esa inserción provocó un crecimiento desmedido en la altura del árbol, y en ese caso, aplicar las operaciones de rotación de nodos que sean necesarias para que el árbol siga siendo de búsqueda, pero con una altura menor. Todos estos procesos y estrategias de mantenimiento del balance o equilibrio de un árbol de búsqueda son necesarios, pero también son complejos y extensos… y exceden el alcance introductorio de esta ficha de clase (y de la asignatura), por lo cual nos limitamos solo a mencionar el tema, dejando para asignaturas posteriores la cobertura detallada del mismo.

Bibliografía

[1] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[2] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: https://docs.python.org/3/.