Asignatura	Datos del alumno	Fecha
Programación científica y HPC	**Apellidos:** Martínez García	29/04/2022
	Nombre: Kevin	

Laboratorio: Árboles Binarios

1. Introducción

El objetivo de esta actividad consistió en trabajar con algunos de los tipos abstractos de datos vistos en clases de teoría como los árboles binarios, las pilas y las colas. La estructura de esta memoria va a ser la siguiente. En primer lugar, propondremos una reestructuración de las clases proporcionadas como material del trabajo de cara a crear una estructura con mayor potencial de ampliación en un futuro. Seguidamente, plantearemos una forma de recorrer un árbol binario en profundidad de forma iterativa. En tercer lugar, desarrollaremos un algoritmo iterativo para recorrer un árbol binario en amplitud y en último lugar, propondremos una posible implementación de un árbol de Huffman.

2. Jerarquía de clases propuesta

Como material de la práctica se proporcionaba la clase `ArbolBinarioOrdenado` que no es más que una implementación de ese mismo tipo abstracto de datos. Sin embargo, y como hemos visto en la introducción, al final de esta memoria trabajaremos los árboles de Huffman. Teniendo en cuenta que ambos tipos de árboles son casos particulares de un árbol binario consideramos adecuado crear una *súper clase* `ArbolBinario` de la cual heredarían `ArbolBinarioOrdenado` y `ArbolHuffman`.

De esta forma, los métodos presentes en la clase `ArbolBinarioOrdenado` se movieron a la clase `ArbolBinario` quedando accesibles a sus subclases mediante el sistema de herencia en Python. El método `insertarElemento` en la clase `ArbolBinario` simplemente inserta un nuevo elemento buscando el primer hueco libre por amplitud. El método `tieneElemento` en esta misma clase simplemente hace una búsqueda en todos los subárboles hasta encontrar el elemento deseado. Estos dos métodos se reescriben en la clase `ArbolBinarioOrdenado` pues su implementación difiere de la de su *súper clase*.

Por último, todos los métodos de recorridos del árbol (*in orden*, *pre orden*, etc.) se colocaron en la *súper clase* `ArbolBinario` pues son independientes de la implementación concreta del tipo de árbol.

3. Recorrido de un árbol en profundidad de forma iterativa

La implementación proporcionada de un recorrido en profundidad de un árbol binario se hacía de forma recursiva. A continuación vamos a proponer la implementación de algunas de estas forma de recorrido utilizando métodos iterativos. Además, en cada caso examinaremos la complejidad temporal y espacial del algoritmo diseñado, así como un ejemplo para verificar su correcto funcionamiento.

3.1. Recorrido *pre orden*

El recorrido en *pre orden* en profundidad de un árbol binario sigue el siguiente orden de operaciones:

1. Visitamos raíz del árbol actual.
2. Recorremos su sub-árbol izquierdo.
3. Recorremos su sub-árbol derecho.

Nuestra implementación devolverá como resultado del recorrido una lista con las raíces de los nodos visitados de acuerdo a este orden.

3.1.1. Implementación del algoritmo iterativo *pre orden*

La implementación propuesta para este tipo de recorrido es la que aparece en la Figura 1 a continuación. De este código lo primero que se hace notar es el uso de una estructura "auxiliar" `p` que no es más que una pila. La implementación de `Pila` es la misma que la propuesta en sesiones de teoría y, para poder hacer uso de la clase se añadió la sentencia `from pila import Pila` al principio del script.

```
        def preOrdenIterativo(self):
            p = Pila(type(self))
            l = []
            if not self.estaVacio():
                p.apilar(self)
                while not p.estaVacia():
                    actual = p.desapilar()
                    l.append(actual._raiz) # Visitamos la raíz del árbol actual
                    if not actual._arbolDcho.estaVacio():
                        p.apilar(actual._arbolDcho) # Recorremos su sub-árbol dcho.
                    if not actual._arbolIzdo.estaVacio():
                        p.apilar(actual._arbolIzdo) # Recorremos su sub-árbol izdo.
            return l
```

Figura 1: Implementación del recorrido iterativo en profundidad

La idea de este método consiste en utilizar la pila como una estructura que nos permite acceder de forma inmediata al nodo que hemos insertado de forma más reciente (por la política *Last In First Out*). De esta forma, para cierto nodo que estemos visitando (`actual`) procedemos añadiéndolo a la lista de visitados (`l`). Seguidamente si el nodo tiene un sub-árbol derecho lo apilamos y hacemos lo mismo para su sub-árbol izquierdo (también en caso de que no esté vacío). En la siguiente iteración, al desapilar obtendremos en principio el subárbol izquierdo (y guardaremos su raíz en `l`). El proceso se repetirá hasta que eventualmente no exista ese sub-árbol izquierdo y, por el funcionamiento de la pila, comencemos a procesar el sub-árbol derecho.

3.1.2. Análisis de la complejidad y propuesta de ejemplo

Para hablar de la complejidad de un algoritmo siempre debemos tener en cuenta tanto la *complejidad temporal* como la *complejidad espacial*. Comenzando por la *complejidad temporal* si tomamos la sentencia `while not p.estaVacia():` como instrucción crítica, entonces es fácil ver que la complejidad temporal es $\mathcal{O}(n)$. Para verlo simplemente hay que darse cuenta de que, para que un nodo se pueda incluir en `l` debe haberse almacenado previamente en `p` y `l` debe contener todos los nodos del árbol. Por tanto el bucle iterará *n* veces o lo que es lo mismo, tantas veces como nodos existan en el árbol.

En cuanto al coste espacial, en el peor de los casos también tendremos un coste lineal con el número de elementos del árbol, es decir, $\mathcal{O}(n)$. Este es el caso en el que cada nodo que visitemos (excepto los hoja) tengan un hijo derecho además de un izquierdo. En esa situación, la pila almacenará $n/2 - 1$ elementos y, para n's suficientemente grandes podremos afirmar que la cota superior será $\mathcal{O}(n)$.

Una situación más favorable ocurriría en el caso de que el árbol se mantuviese *balanceado*. Un *árbol binario balanceado* es un tipo de árbol que verifica que la diferencia de altura entre los dos sub-árboles de todo nodo es 1 a lo sumo (en la sección 4 definiremos el concepto de *altura*). En un árbol binario balanceado podemos afirmar que su altura es $h = \lfloor log_2(n) \rfloor + 1$ y por tanto, el máximo número de elementos que contendrá la pila será del orden de h. En ese caso, la cota espacial vendrá dada por $\mathcal{O}(log_2(n))$.

Una forma de comprobar el funcionamiento del método sería crear un árbol binario ordenado, introducir una serie de elementos y finalmente, comprobar que el método programado da el mismo resultado que el que se proporcionaba ya implementado en la práctica (`preOrden`). El árbol binario ordenado con el que vamos a trabajar es el que aparece en la Figura 2 a continuación.

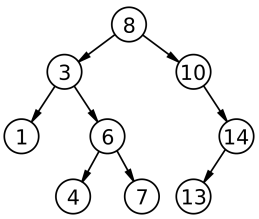


Figura 2: Árbol binario ordenado de ejemplo

Para hacerlo, podemos abrir una sesión interactiva de Python por terminal e introducir las lineas que aparecen en la Figura 3. Podemos ver como efectivamente ambos métodos devuelven el mismo resultado.

3.2. Recorrido *in orden*

El recorrido en *in orden* en profundidad de un árbol binario sigue el siguiente orden de operaciones:

```
>>> from arbol import ArbolBinarioOrdenado
>>> abo = ArbolBinarioOrdenado()
>>> elems = [8, 3, 1, 6, 4, 7, 10, 14, 13]
>>> for elem in elems:
...     abo.insertarElem(elem)
...
>>> print(f"Pre Orden Recursivo -> {abo.preOrden()}")
Pre Orden Recursivo -> [8, 3, 1, 6, 4, 7, 10, 14, 13]
>>> print(f"Pre Orden Iterativo -> {abo.preOrdenIterativo()}")
Pre Orden Iterativo -> [8, 3, 1, 6, 4, 7, 10, 14, 13]
```

Figura 3: Prueba de *pre orden* iterativo

1. Recorremos el sub-árbol izquierdo.
2. Visitamos raíz del árbol actual.
3. Recorremos el sub-árbol derecho.

Nuestra implementación devolverá como resultado del recorrido una lista con las raíces de los nodos visitados de acuerdo a este orden.

3.2.1. Implementación del algoritmo iterativo *in orden*

De nuevo, para implementar este algoritmo se utilizó una estructura auxiliar de tipo `Pila`. El código implementado es el que aparece en la Figura 4. Para cierto nodo que estemos visitando (`actual`) procedemos apilando su sub-árbol izquierdo en caso de que exista. Eventualmente, llegará un momento en que no podamos continuar por la izquierda, en ese caso, desapilaremos el último elemento introduido en `p`, lo marcaremos como visitado (guardándolo en `l`) y seguiremos por su rama derecha. En caso de que tampoco podamos continuar por su rama derecha, volveremos a su nodo padre consultando la cima de la pila. La variable *booleana* `fin_rama` se utiliza para indicar que cierto sub-árbol se ha recorrido completamente (se han visitado todos sus nodos) y no se debe explorar de nuevo.

3.2.2. Análisis de la complejidad y propuesta de ejemplo

Por los mismos motivos que vimos en la subsección 3.1.2, la complejidad temporal de este algoritmo es $\mathcal{O}(n)$. Si analizamos la complejidad espacial, podemos encontrarnos con el peor caso de un árbol *degenerado* o *sesgado*. Un *árbol binario sesgado*, es un tipo de árbol en el que cada nodo tiene a lo sumo un nodo hijo, tal y como aparece en la Figura 5 a continuación. Además este árbol está *degenerado a izquierdas*, es decir, sus nodos únicamente tienen hijos en su rama izquierda. Podemos ver como en este caso la pila `p` almacenará todos los nodos hasta el nodo hoja y a partir de este punto comenzará a desapilarlos. Esto nos lleva a la conclusión de que la cota espacial vendrá dada por $\mathcal{O}(n)$. Por otra parte, en el caso de un *árbol binario balanceado* seguiremos teniendo $\mathcal{O}(log_2(n))$.

```
    def inOrdenIterativo(self):
        p = Pila(type(self))
        l = []
        if not self.estaVacio():
            p.apilar(self)
            actual = self
            fin_rama = False
            while not p.estaVacia():
                if not actual._arbolIzdo.estaVacio() and not fin_rama:
                    p.apilar(actual._arbolIzdo)
                    actual = actual._arbolIzdo
                else:
                    actual = p.desapilar()
                    l.append(actual._raiz)
                    if not actual._arbolDcho.estaVacio():
                        p.apilar(actual._arbolDcho)
                        actual = actual._arbolDcho
                        fin_rama = False
                    else:
                        actual = p.cima()
                        fin_rama = True
            return l
```

Figura 4: Implementación recorrido *in orden* en profundidad

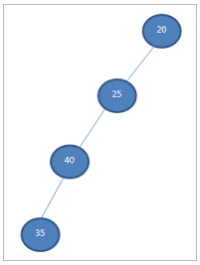


Figura 5: Árbol binario sesgado

De nuevo, para comprobar el funcionamiento del método podríamos crear un árbol binario ordenado, insertar una serie de elementos y comprobar que el método implementado devuelve el mismo resultado que el proporcionado como material de la práctica (`inOrden`). Al igual que en la sección 3.1, mediante una sesión interactiva de Python, introduciremos el árbol de la Figura 3 y comprobaremos el funcionamiento tal y como aparece en la Figura 6. Podemos ver que efectivamente los resultados son idénticos.

```
        >>> print(f"In Orden Recursivo -> {abo.inOrden()}")
        In Orden Recursivo -> [1, 3, 4, 6, 7, 8, 10, 13, 14]
        >>> print(f"In Orden Iterativo -> {abo.inOrdenIterativo()}")
        In Orden Iterativo -> [1, 3, 4, 6, 7, 8, 10, 13, 14]
```

Figura 6: Prueba de *pre orden* iterativo

4. Recorrido iterativo del árbol en amplitud

Para hablar del recorrido en amplitud de un árbol binario, debemos definir previamente una serie de conceptos.

- **Profundidad**: la profundidad de un nodo es el número de aristas desde dicho nodo a la raíz del árbol.
- **Altura**: la altura de un nodo es el número de aristas que hay en el camino más largo desde el nodo hasta una hoja. La altura del árbol se define como la altura de su raíz más uno.
- **Nivel**: el nivel de un nodo es la diferencia entre la altura de la raíz y la profundidad de dicho nodo.

De estas definiciones se desprende que si el árbol tiene altura h, el nodo raíz ocupará el nivel $h-1$, sus nodos hijos el nivel $h-2$, y así sucesivamente hasta alcanzar el nivel 0. En este sentido, un recorrido de un árbol binario por amplitud simplemente es un recorrido del árbol por niveles, es decir, mostraremos los nodos del nivel $h-1$, seguidos de los del nivel $h-2$ y así hasta el máximo nivel de profundidad.

4.1. Implementación del recorrido iterativo en amplitud

Una implementación de este tipo de recorrido es la que aparece en el código de la Figura 7 a continuación. Esta implementación difiere de las que hemos visto anteriormente en el hecho de utilizar una `Cola` como estructura auxiliar en lugar de una `Pila`. Para entender el algoritmo, podemos plantear una situación en la que para cierto nodo `padre` hemos encolado sus dos nodos hijo (primero el izquierdo y luego el derecho). Desencolamos el primer elemento (el hijo izquierdo) y procedemos a encolar sus nodos hijo (en caso de que los tenga). En la siguiente iteración, por la política *First In First Out* de la cola, desencolaremos el hijo derecho del nodo `padre`. De esta forma, la cola nos permite hacer un recorrido en amplitud porque, antes de descender un nivel en el árbol habremos recorrido completamente el nivel anterior (ya que dichos nodos estarían encolados antes).

```
    def amplitudIterativo(self):
        c = Cola(type(self))
        l = []
        if not self.estaVacio():
            c.encolar(self)
            while not c.estaVacia():
                actual = c.desencolar()
                l.append(actual._raiz)
                if not actual._arbolIzdo.estaVacio():
                    c.encolar(actual._arbolIzdo)
                if not actual._arbolDcho.estaVacio():
                    c.encolar(actual._arbolDcho)
        return l
```

Figura 7: Prueba de *pre orden* iterativo

4.2. Análisis de la complejidad y propuesta de ejemplo

De nuevo, vamos a tratar de analizar tanto la *complejidad temporal* como la *complejidad espacial* del algoritmo implementado. En cuanto a la *complejidad temporal*, es evidente que el árbol debe recorrerse en su totalidad o lo que es lo mismo, el bucle `while` deberá iterar tantas veces como nodos existan en el árbol. Por ese motivo, podemos concluir que la complejidad es $\mathcal{O}(n)$. En cuanto a la complejidad espacial, en este caso ocurre una situación opuesta a los recorridos que hemos visto previamente. Por una parte, si el árbol está *degenerado o sesgado* la complejidad será de orden $\mathcal{O}(1)$, pues como máximo tendremos un elemento en la cola. Por otra parte, si el árbol está *balanceado* entonces el coste dependerá de la *amplitud* del árbol es decir del nivel con el máximo número de nodos. Sabemos que en un árbol de altura *h* podremos tener hasta 2^{h-1} nodos en el último nivel. Luego si $h = \lfloor log_2(n) \rfloor + 1$ entonces la cola contendrá $\lfloor \frac{n}{2} \rfloor + 1$ elementos (esto es así porque en el peor caso tendremos el último nivel completo y contendrá exactamente $\lfloor \frac{n}{2} \rfloor + 1$ elementos). En otras palabras el peor caso fija una cota de $\mathcal{O}(n)$.

Para comprobar el correcto funcionamiento de nuestra implementación, vamos a partir del árbol que ya vimos en la Figura 3 y vamos a ver como se dividiría en diferentes niveles. La división de dicho árbol en niveles es la que aparece en la Figura 8 a continuación. De esta imagen se deduce que, recorriendo cada nivel de izquierda a derecha, nuestro método debería devolver la lista `[8, 3, 10, 1, 6, 14, 4, 7, 13]`.

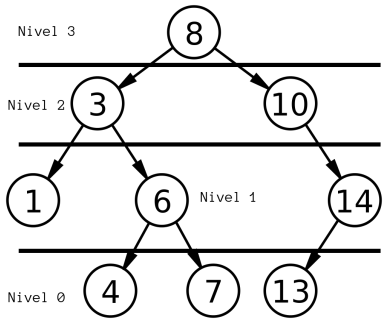


Figura 8: Árbol binario ordenado de ejemplo dividido por niveles

Para comprobarlo, podemos abrir una sesión interactiva en Python desde terminal, generar el árbol y comprobar que efectivamente el método devuelve la lista que ya hemos mencionado. El código para comprobarlo es el que aparece en la Figura 9 a continuación y, como se puede ver, ambas listas son idénticas.

```
>>> print(f"Recorrido en amplitud -> {abo.amplitudIterativo()}")
Recorrido en amplitud -> [8, 3, 10, 1, 6, 14, 4, 7, 13]
```

Figura 9: Prueba de recorrido en amplitud

5. Algoritmo de Huffman

El algoritmo de Huffman se usa para construir un código de Huffman asociado a cierta cadena de caracteres. Para construirlo, necesitamos un *diccionario* con la frecuencia de cada uno de los símbolos de dicha cadena. El algoritmo comienza generando un árbol binario de acuerdo al siguiente procedimiento.

1. Generamos un nodo hoja para cada uno de los símbolos del diccionario y almacenamos en cada uno la frecuencia del carácter que representa.
2. Introducimos estos nodos hoja en una lista y la ordenamos ascendentemente en función del valor de la frecuencia.
3. Mientras la lista contenga más de un nodo hacer
   a) Tomar los dos nodos con los menores valores de frecuencia.
   b) Unir dichos nodos mediante un nodo padre que almacenará la suma de las frecuencias de sus dos hijos
   c) Etiquetar el hijo izquierdo con un 0 y el hijo derecho con un 1
   d) Insertar el nodo padre en la lista de forma ordenada (manteniendo el orden ascendente de frecuencia)
4. El elemento que queda en la lista será el nodo raíz del árbol generado.

En esta sección de la memoria no mostraremos explícitamente todo el código implementado pues su extensión es considerable. Si que comentaremos algunos de los aspectos que consideramos más importantes respecto al funcionamiento del algoritmo. En primer lugar, como vimos en la sección de introducción, se creó una clase `ArbolHuffman` que hereda de `ArbolBinario` pues no es más que un caso particular de árbol binario. Un `ArbolHuffman` hereda los atributos de instancia de su *súper clase* y añade los siguientes.

- `frec`: Cada sub-árbol que se genere tendrá asociado un elemento numérico que representa la frecuencia (si es un nodo hijo, la frecuencia del símbolo que representa, si es un nodo padre la suma de la frecuencia de sus hijos).
- `padre`: Cada nodo tendrá una referencia a su nodo padre (que es una referencia a otro objeto `ArbolHuffman`). Este atributo se añadió con el objetivo de facilitar el recorrido desde las hojas del árbol hasta el nodo raíz. En el nodo raíz esta referencia quedará a `None`.
- `codigo`: Este atributo representa la etiqueta 0 o 1 de cada nodo del árbol.
- `simbolo`: Este atributo almacena el símbolo que representa cada nodo. En principio, el símbolo sólo resulta relevante para los nodos hoja.
- `hojas`: En el nodo raíz almacenaremos una lista con todos los nodos hoja del árbol, esto facilitará ciertas tareas de codificación como veremos más adelante.

Además, por conveniencia a la hora de representar visualmente los árboles generados, en la raíz de cada

nodo almacenamos una tupla con la forma `(simbolo, frec, codigo)`. De esta forma, si recorremos el árbol mediante alguno de los métodos implementados en secciones anteriores podemos reconocer con mayor facilidad cada uno de los nodos.

5.1. Implementación de la construcción, codificación y decodificación

Por su extensión, no entraremos en el detalle de la implementación del método constructor del árbol. Simplemente comentar que el procedimiento es igual al explicado al principio de esta misma sección y que, en el código adjunto a esta memoria aparece comentado con detalle para poder entender con facilidad su funcionamiento.

Una vez implementada la construcción del árbol, quedaban pendientes los métodos de codificación y decodificación. Por una parte, el método `decodificar` recibe como argumento un código y devuelve el símbolo correspondiente al mismo (en caso de que exista). Como puede verse en la Figura 10(a) el método recorre el código recibido dígito a dígito y se mueve al sub-árbol izquierdo en caso de que se lea un 0, al sub-árbol derecho en caso de que se lea un 1 y arroja una excepción si se lee cualquier otro dígito (en este caso consideraremos que el código recibido está mal formado). Si mientras recorremos el código llegamos a un nodo vacío devolvemos `None` pues no codifica ninún símbolo conocido. Al terminar de leer el código pueden ocurrir dos cosas, que lleguemos a un nodo hoja que representa un símbolo (y devolveremos dicho símbolo), o que el nodo al que hemos llegado no represente ningún símbolo (por lo que devolveremos `None`).

```
def decodificar(self, codigo):
    aux = self
    for digito in codigo:
        if digito == '0':
            aux = aux._arbolIzdo
        elif digito == '1':
            aux = aux._arbolDcho
        else:
            raise ValueError
        if aux.estaVacio():
            return None
    if (aux._simbolo != ""):
        return aux._simbolo
    else:
        return None
```

(a) Método decodificación

```
def codificar(self, simbolo):
    nodo_hoja = [nodo for nodo in \
            self._hojas if \
            nodo._simbolo == simbolo]
    if not nodo_hoja:
        return None
    else:
        nodo = nodo_hoja[0]
        codigo = nodo._codigo
        while nodo.__tienePadre():
            nodo = nodo._padre
            codigo = nodo._codigo+codigo
    return codigo
```

(b) Método codificación

Figura 10: Operaciones sobre un árbol de Huffman

Por otra parte el método `codificar` recibe un símbolo como argumento y devuelve el código correspondiente al mismo. Como puede verse en la Figura 10(b) el método comienza haciendo una búsqueda en `hojas` del nodo que contiene al símbolo pasado como argumento. En caso de que no se encuentre dicho nodo, devolvemos `None`. En caso de que si exista almacenamos el código de dicho nodo y comenzamos el recorrido hacia la raíz del árbol. Esto se hace mediante el método privado `__tienePadre` que simplemente

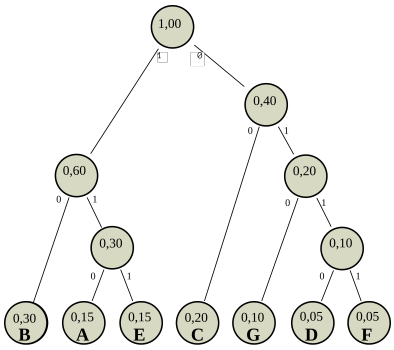
nos indica si el padre de cierto nodo no es `None`. Mientras el nodo actual tenga padre iremos ascendiendo y almacenando en `codigo` el código que vamos generando. Al llegar al nodo raíz saldremos del bucle y devolveremos el código.

5.2. Codificación de una cadena de caracteres

Para obtener la codificación de cierta cadena de caracteres, vamos a comenzar creando un árbol de Huffman mediante una sesión interactiva en Python. Para ello escribiremos el código que aprece en la Figura 11a) a continuación. Si se hace un recorrido por amplitud del árbol generado, es fácil ver la división por niveles que se sugiere en la propia Figura. Si representamos gráficamente este árbol, se obtendría la construcción que aparece en la Figura 11b). A partir del árbol, es fácil ver que para la cadena `"DEBA"` el código a devolver sería `011011110110` (`D: 0110`, `E: 111`, `B: 10`, `A: 110`). Si lanzamos una sesión interactiva en Python podemos comprobar si efectivamente el método devuelve esa codificación. Las líneas para comprobarlo son las que aparecen en la Figura 12. Mostramos como además al decodificar la cadena de 0s y 1s generada volvemos a la cadena original `"DEBA"`.

```
>>> from arbol import ArbolHuffman
>>> dic = {"A":0.15, "B":0.3, "C":0.2, "D":0.05, "E":0.15, "F":0.05, "G":0.10}
>>> ah = ArbolHuffman(dic)
>>> print(ah.amplitudIterativo())
[('', 1.0), <<--- NIVEL 4
('', 0.4, '0'), ('', 0.6, '1'), <<--- NIVEL 3
('C', 0.2, '0'), ('', 0.2, '1'), ('B', 0.3, '0'), ('', 0.3, '1'), <<--- NIVEL 2
('G', 0.1, '0'), ('', 0.1, '1'), ('A', 0.15, '0'), ('E', 0.15, '1'), <<--- NIVEL 1
('D', 0.05, '0'), ('F', 0.05, '1') <<--- NIVEL 0]
```

(a) Creación Python Árbol Huffman



(b) Representación gráfica árbol Huffman

Figura 11: Árbol de Huffman de ejemplo

```
>>> cadena = "DEBA"; res = ""; aux = {}
>>> for sim in cadena:
...     codif = ah.codificar(sim)
...     res = res + codif
...     aux[sim] = codif
>>> print(res)
011011110110
>>> aux
{'D': '0110', 'E': '111', 'B': '10', 'A': '110'}
>>> cadena_original = ""
>>> for cod in aux.values():
...     cadena_original = cadena_original + ah.decodificar(cod)
>>> print(cadena_original)
DEBA
```

Figura 12: Prueba codificación y decodificación con árbol de Huffman