Ayudantía I1

Ordenamiento
Heaps
Árboles

Ordenamiento
Ejercicio 1

1) En clase vimos que la base de *Quicksort* es la buena elección de un pivote para la función *Partition* y usamos un pivote elegido de forma aleatoria. Sin embargo, existen otras maneras de elegirlo.

Una de ellas es reemplazar la función *Partition* por *MedianPartition*. En la función *MedianPartition* el pivote se elige con la función *Median*(A, i, f), que entrega en tiempo O(n) el índice en donde se ubica la mediana del arreglo $A[i..f]$.

Pregunta 1 - I1-2025-2

a) [1 pt.] Modifique los pseudocódigos de *Quicksort* y *Partition* de modo de adaptarlos a este nuevo escenario (*MedianPartition*) y explique porqué *MedianPartition* es O(n).

Pregunta 1.a) - I1-2025-2

Pregunta 1.a) - Solución

```
Quicksort(A, i, f)
    if i < f then
        p ← MedianPartition(A, i, f) ------ Se cambia la elección del pivote
        Quicksort(A, i, p-1)
        Quicksort(A, p+1, f)

MedianPartition( (A, i , f )
        p ← Median A[i, f ]
        m, M ← secuencias vac´ıas
        Insertar p en M
        for x ∈ A[i, f ] :
                if x < p : Insertar x en m
                elif x > p : Insertar x en M
        A[i, f ] ← Concat(m, M)
        return i + |m|
```

p ← Median A[i, f] → O(n) (por enunciado)

El resto es O(n) ya que funciona igual a Partition!

O(n)+O(n)=O(2n)=O(n)

b) [2 pts.] Argumente en qué casos el uso de *MedianPartition* podría mejorar el tiempo de ejecución del peor caso de *Quicksort*.

Pregunta 1.b) - I1-2025-2

Pregunta 1.b) - Solución

b) En Quicksort ocurre cuando la partición produce divide el arreglo en dos partes de tamaños muy diferentes, en particilar cuando el pivote es el máximo o mínimo. Al seleccionar el Pivote como la mediana se evita seleccionar los valores que dan el peor caso.

Def. la mediana representa el valor de la variable de posición central en un conjunto de datos ordenados.

Si yo escojo siempre el valor que me separa los datos en **mitades perfectas**, voy a caer siempre en un **dividir para conquistar** con complejidad ideal! Por ejemplo:

1, 3, 3, **6**, 7, 8, 9

Dado que la mediana es 6, aplicaré luego MedianPartition de los primeros y últimos 3 datos (separación con máximo 1 de diferencia en arrays pares).

c) [3 pts.] Otra posible mejora de *QuickSort* es cambiar a *InsertionSort* para sub-secuencias pequeñas (≤ 20 elementos). Modifique el pseudocódigo de *Quicksort* para implementar esta mejora.

Pregunta 1.c) - I1-2025-2

Pregunta 1.c) - Solución

b) En Quicksort ocurre cuando la partición produce divide el arreglo en dos partes de tamaños muy diferentes, en particilar cuando el pivote es el máximo o mínimo. Al seleccionar el Pivote como la mediana se evita seleccionar los valores que dan el peor caso.

c) Se debe agregar el control

```
Quicksort(A, i, f)
   n ← f - i + 1                    ------ Se debe agregar el cambio de quicksort a
   if n ≤ 20 then                   ------ Insertion sort para sub arreglos <= 20
      InsertionSort(A, i, f)        ------
   else if i < f then               ------
      p ← Partition(A, i, f)     o  p ← MedianPartition(A, i, f)
      Quicksort(A, i, p-1)
      Quicksort(A, p+1, f)
```

Orden lineal
Ejercicio 1

Verdadero o falso:
RadixSort es un algoritmo de ordenación que puede ordenar *n* números enteros y cuyo tiempo de ejecución es siempre O(n), justifique su respuesta.

Pregunta 1 - EX-2017-1



Pregunta 1 - EX-2017-1

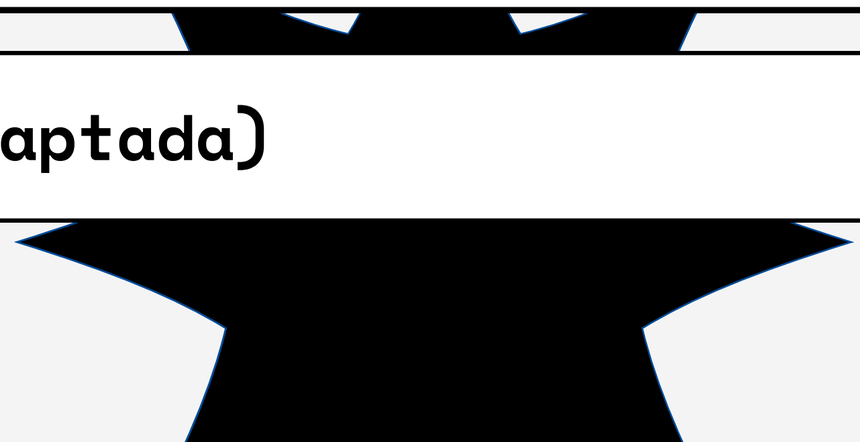
RadixSort funciona haciendo CountingSort por cada dígito desde el menos significativo hasta el más significativo, es decir hace *digitos* veces una operación que es $n+k$, donde n es la cantidad de números a ordenar y k es el número más grande.

El problema está si $k>>n$, por ejemplo $k=2^n$, porque ahora por cada dígito tendríamos que hacer 2^n+n operaciones, lo que es $O(d*2^n)$

Orden lineal
Ejercicio 2

Verdadero o falso:
Radix Sort es un algoritmo de orden estable.

Pregunta 4 - I2-2016-2 (Adaptada)



Pregunta 4 - I2-2016-2 (Adaptada)

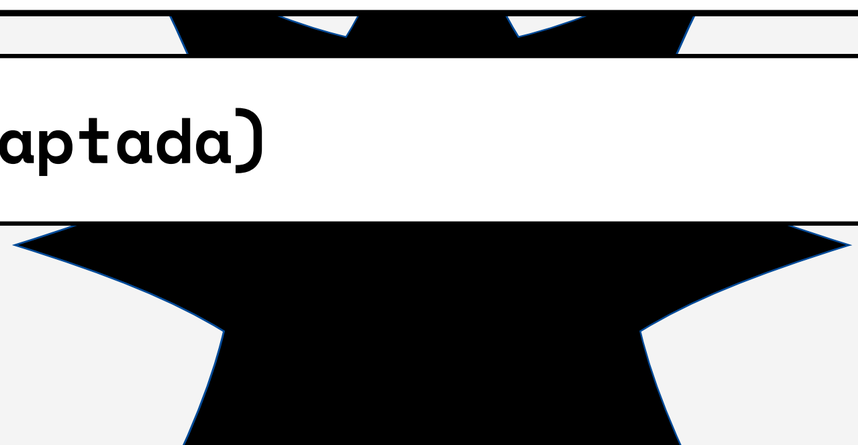
Si, RadixSort es un algoritmo de orden estable porque ocupa CountingSort como herramienta principal.

CountingSort es estable porque cuando asigna los elementos al arreglo de salida, se recorre desde el último al primer elemento de A.
Por cada vez que se ‘visita’ un índice en C, la cantidad de elementos menores a ese índice disminuye en 1, por lo que si tenemos dos elementos iguales, el que estaba más a la derecha en A va a tener i elementos menores, disminuye en 1, y el que estaba a su izquierda va a tener i-1 elemetos menores, por lo que quedará a la izquierda del primero.

Heaps
Ejercicio 1

Sean {a_0, a_1,...a_n} una colección de elementos que se van a insertar a un *min-heap* con la operación de inserción estándar.
Explique, que propiedad deben cumplir las llaves de estos elementos para obtener el peor y mejor escenario en complejidad computacional. Además, explicite el total de intercambios generados.

Pregunta 1 - I1-2017-1 (Adaptada)



Pregunta 1 - I1-2017-1 (Adaptada)

Dado que es un *min-heap*, sabemos que al insertar un nodo la condición de ser intercambiado es que el nodo insertado sea menor que su padre.

Mejor escenario:
Para que la condición anterior **nunca** se cumpla, es decir, no tenga que hacer ningún intercambio a raíz de una inserción, debe pasar que los elementos {a_0, a_1,...a_n} estén en orden ascendiente, es decir:

$$\forall a_i \; i \in [1, n] \; a_{i-1} < a_i$$

Claramente, por construcción, esto genera **0 intercambios.**

Pregunta 1 - I1-2017-1 (Adaptada)

Peor escenario:
Para que la condición anterior **siempre** se cumpla, debe pasar que los elementos {a_0, a_1,...a_n} estén en orden descendente es decir: $\forall a_i \; i \in [1, n] \; a_{i-1} > a_i$

Sobre la cantidad de intercambios, sabemos que lo peor que puede pasar que es tenga que hacer log(i) intercambios, desde la base hasta la punta cuando hay i nodos. Pero dado que esto ocurre en todos las inserciones lo podemos expresar como:

$$\sum_{i=0}^{n} (\lfloor log_2(i) \rfloor)$$

Heaps
Ejercicio 2

Sea N el número de corredores en una carrera de atletismo particular, en la que cada corredor hace su tiempo sin que nadie más corra, por lo que se registra un tiempo a la vez. Estos tiempos se registran de la forma (nombre, tiempo), donde el tiempo es un número entero. Este es un evento masivo, por lo que hay más de 100 corredores.

Pregunta 2 Heaps



1. Se le pide proponer una estructura para siempre conocer al corredor que va ganando la carrera, esto en O(1). Y sea capaz de seguir funcionando, en máximo O(log(N)) si es que descalifican a un corredor.
2. Se quieren también manejar dos estructuras más, del mismo tipo que en 1., pero que permitan conocer el podio (3 primeros) y el TOP15 en O(1), no necesariamente en orden.

Pregunta 2 Heaps



Pregunta 2 - Heaps

1.

Se puede pensar en un *min-heap*, en el cual se guarden todos los corredores. La prioridad bajo la cual se ordenaría sería que para un registro r, se tome sólo el tiempo ($r.tiempo$ o $r[1]$).

Esto permite conocer el ganador en $O(1)$, ya que es la cabeza. Y es resistente a descalificaciones en $O(\log(N))$ ya que si descalifican a cualquier corredor, por ejemplo, el actual ganador, el *min-heap* se va a reordenar como máximo en $O(\log(N))$.

Pregunta 2 - Heaps

2.
Para el podio se puede pensar en un max-*heap* de tamaño 3. Este permite ver el podio en O(1), ya que son los primeros 3 elementos del arreglo.
Esto funciona si y sólo si se tienen dos reglas sobre la inserción: primero, los primeros 3 corredores entran al heap sin mayor consideración. Segundo, si el *max-heap* guarda a los tres más rápidos, podemos pensar en la cabeza como el más lento de los más rápidos. Entonces querremos cambiar ese elemento si el nuevo registro tiene un tiempo menor a la cabeza del heap.

Pregunta 2 - Heaps

2.
Para el TOP15 se puede pensar en un max-*heap* de tamaño 15. Con la misma condición sobre la inserción que el caso del podio.

Árboles
Ejercicio 1

¿Es todo árbol AVL un árbol rojo-negro? Justifica.

Pregunta 1 - I2-2015-2



Pregunta 1 - I2-2015-2

Sí. En un árbol AVL, en cada nodo la diferencia de alturas entre sus subárboles es a lo más 1. Esto implica que la ruta más larga desde la raíz a una hoja no es más del doble que la ruta más corta.

Esto permite colorear todos los nodos de la ruta más corta como negros, y los nodos adicionales de las rutas más largas como rojos, intercalándolos sin que haya dos rojos consecutivos.

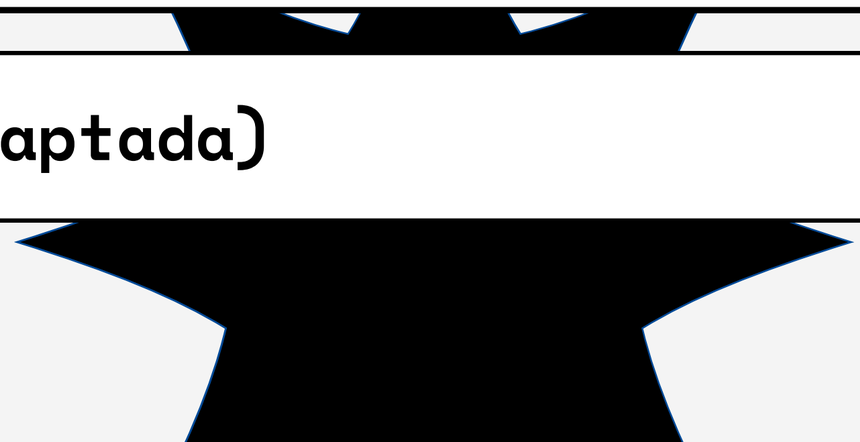
De esta forma, se pueden cumplir todas las propiedades de un árbol rojo-negro.

Árboles
Ejercicio 2

Inserta las claves A, C, E, H, L, M, P, R y S (en ese orden) en un árbol inicialmente vacío, para los siguientes tipos: árbol de búsqueda binario sin propiedades de balance, AVL y 2-3.

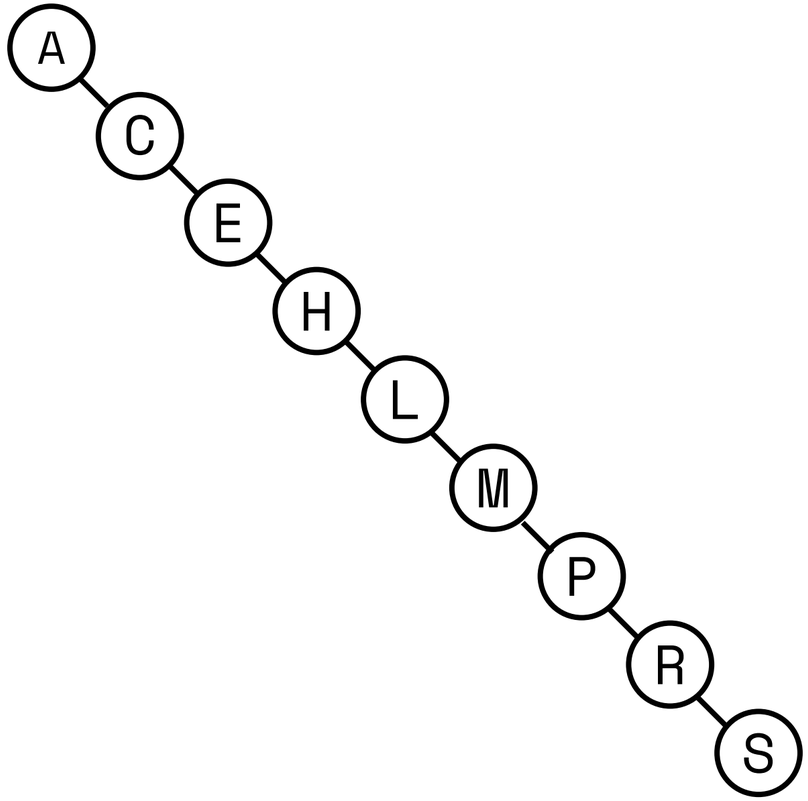
¿Por qué el árbol binario simple puede volverse ineficiente con ciertas secuencias de inserción? ¿Cómo evitan esto los árboles AVL y 2-3?

Pregunta 3 - I1-2016-2 (Adaptada)



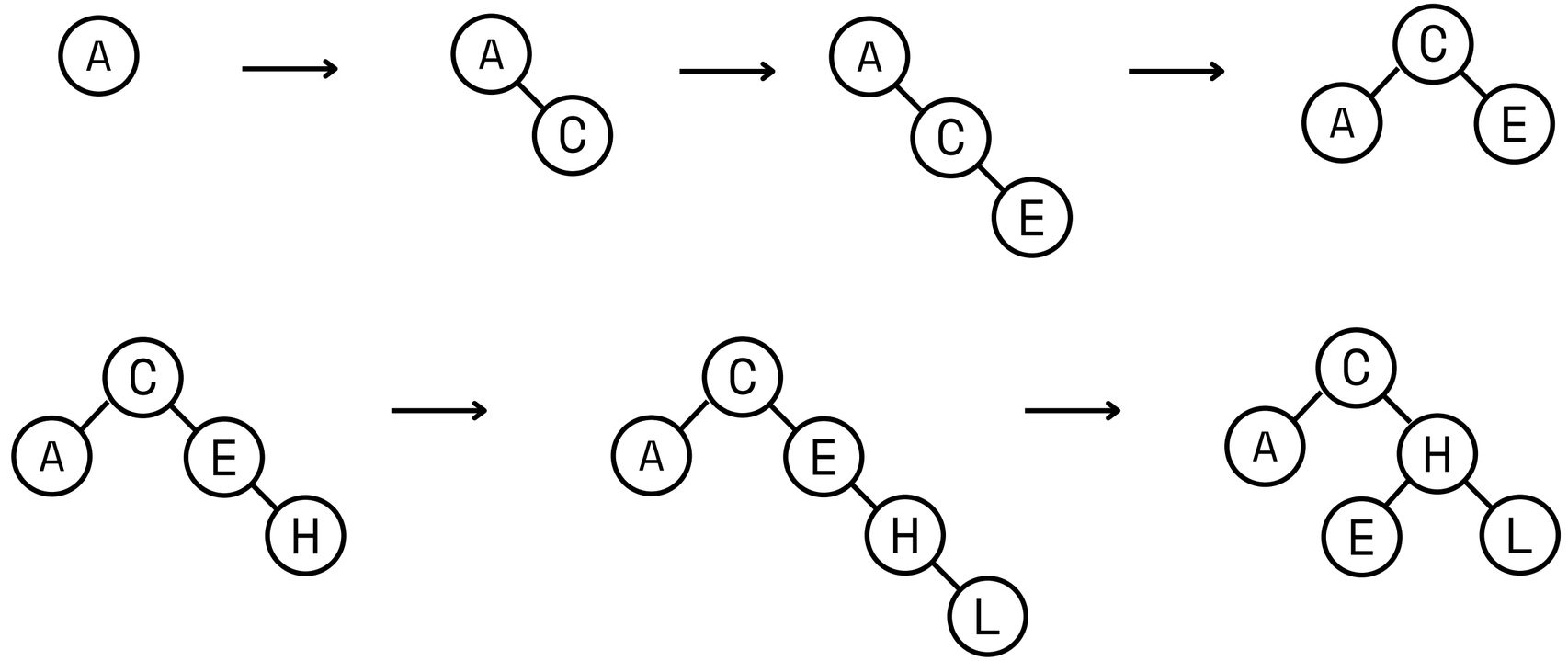
Pregunta 3 - I1-2016-2 (Adaptada)

Árbol binario sin propiedades de balance



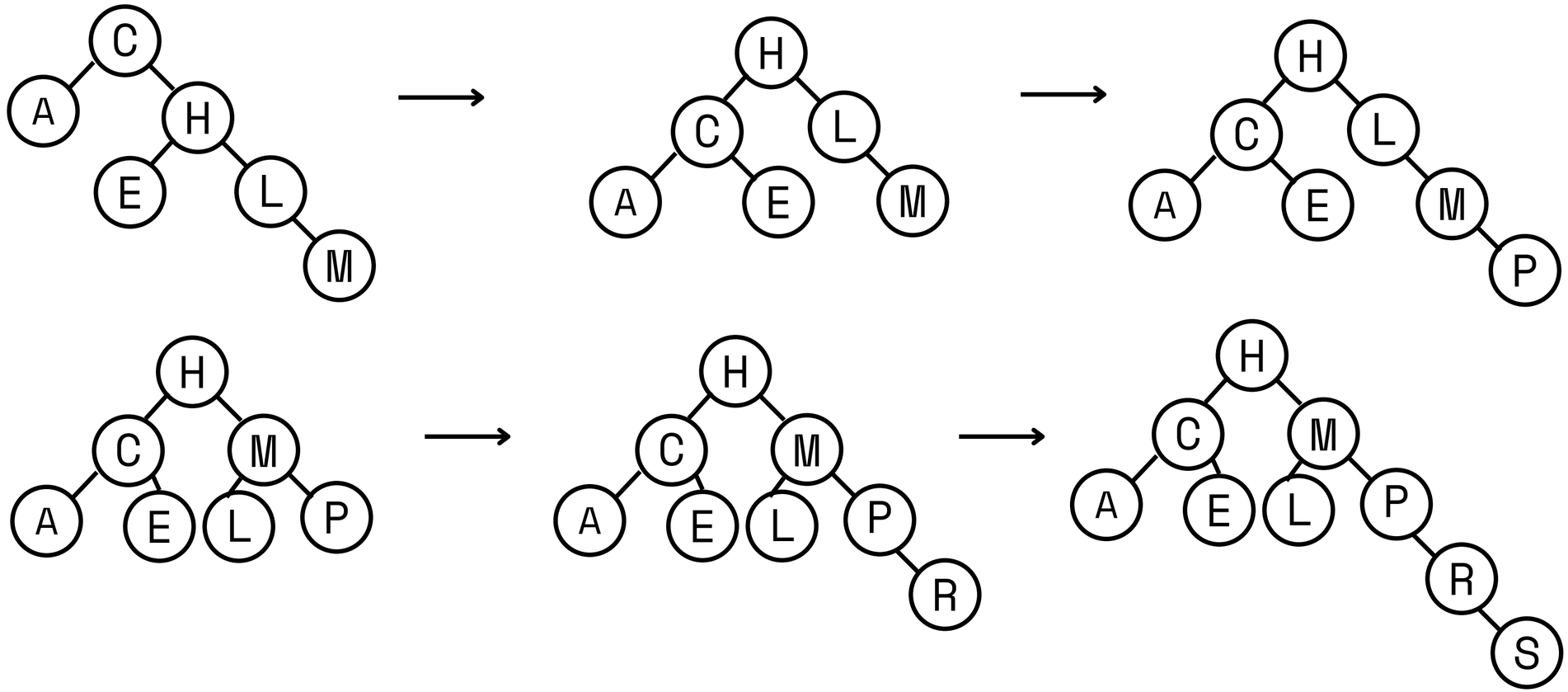
Pregunta 3 - I1-2016-2 (Adaptada)

Árbol AVL



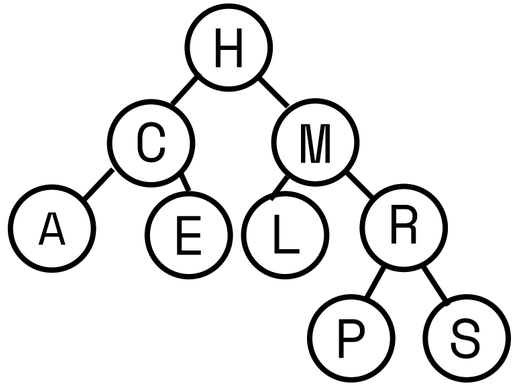
Pregunta 3 - I1-2016-2 (Adaptada)

Árbol AVL



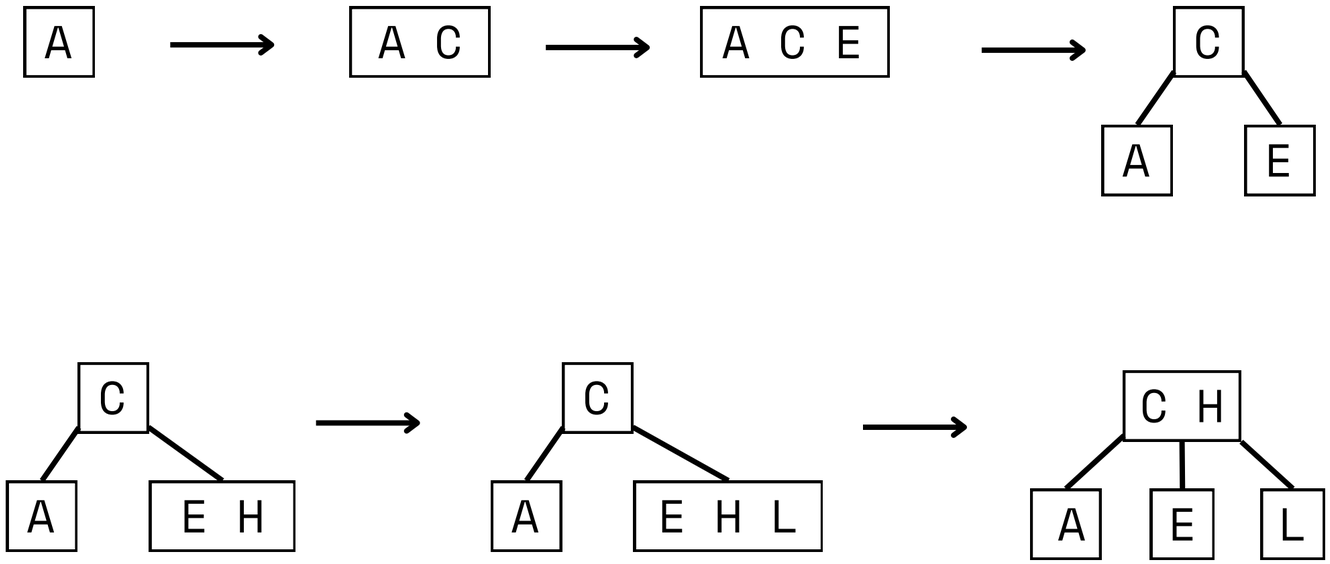
Pregunta 3 - I1-2016-2 (Adaptada)

Árbol AVL



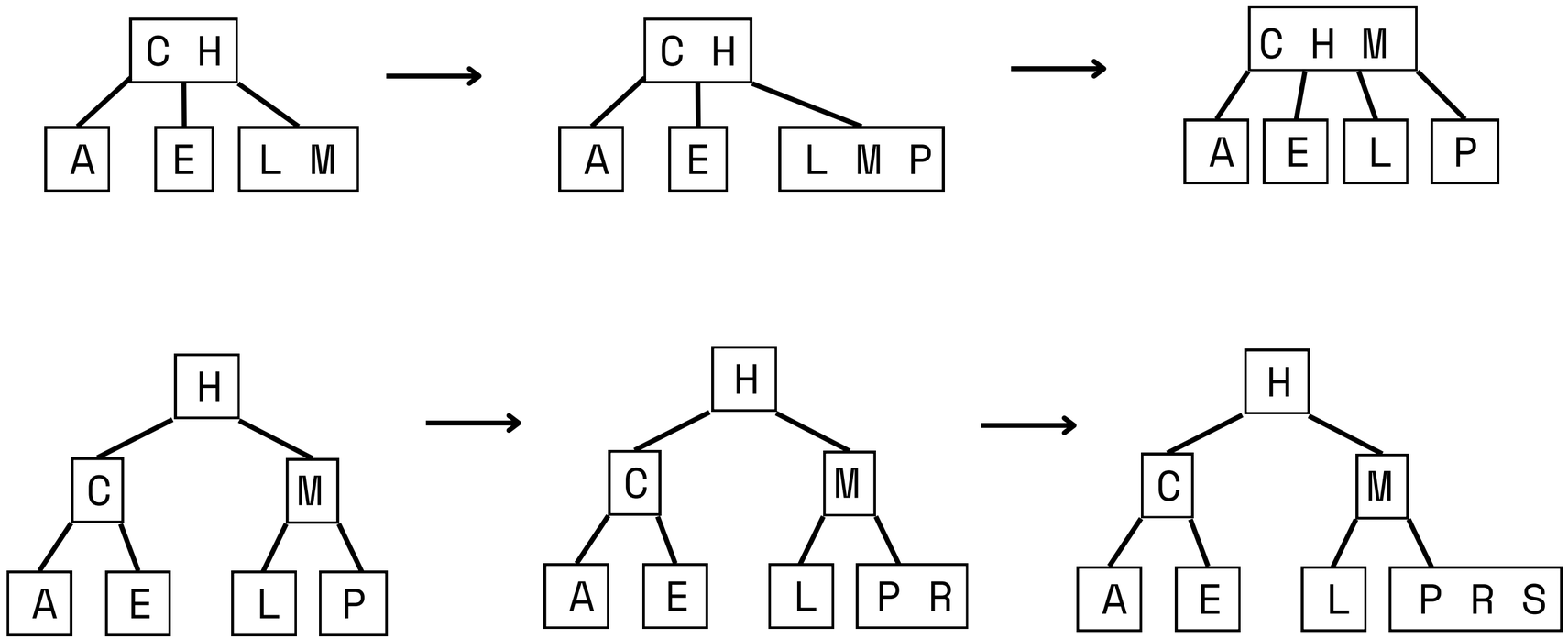
Pregunta 3 - I1-2016-2 (Adaptada)

Árbol 2-3



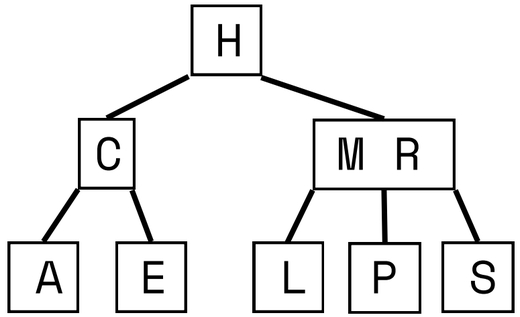
Pregunta 3 - I1-2016-2 (Adaptada)

Árbol 2-3



Pregunta 3 - I1-2016-2 (Adaptada)

Árbol 2-3



Pregunta 3 - I1-2016-2 (Adaptada)

El árbol binario simple puede volverse muy alto si se insertan claves ordenadas, formando una estructura similar a una lista.
Los árboles AVL y 2-3 evitan esto mediante rotaciones o divisiones que mantienen el árbol balanceado.

Feedback