Repaso I2

Arboles - BackanTrack - Hash - Heaps disjuntos

Carlos Paredes - cparedesr@uc.cl

Sergio (Huasito) A - matamalaappels@uc.cl

Árbol ABB

Root

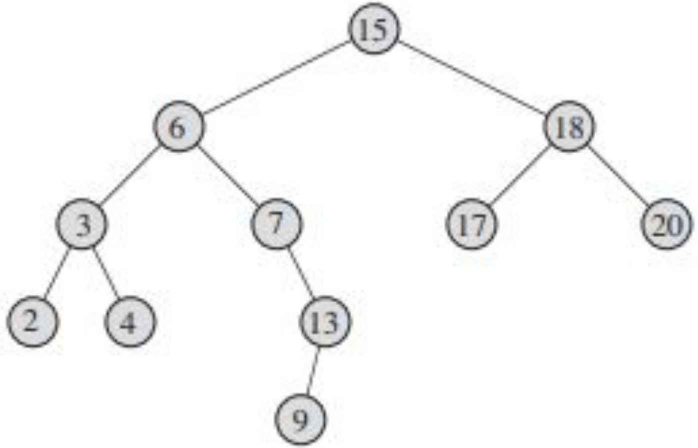
claves menores ← → claves mayores

Recorridos :

In - order : L-N-R

Pre - order : N-L-R

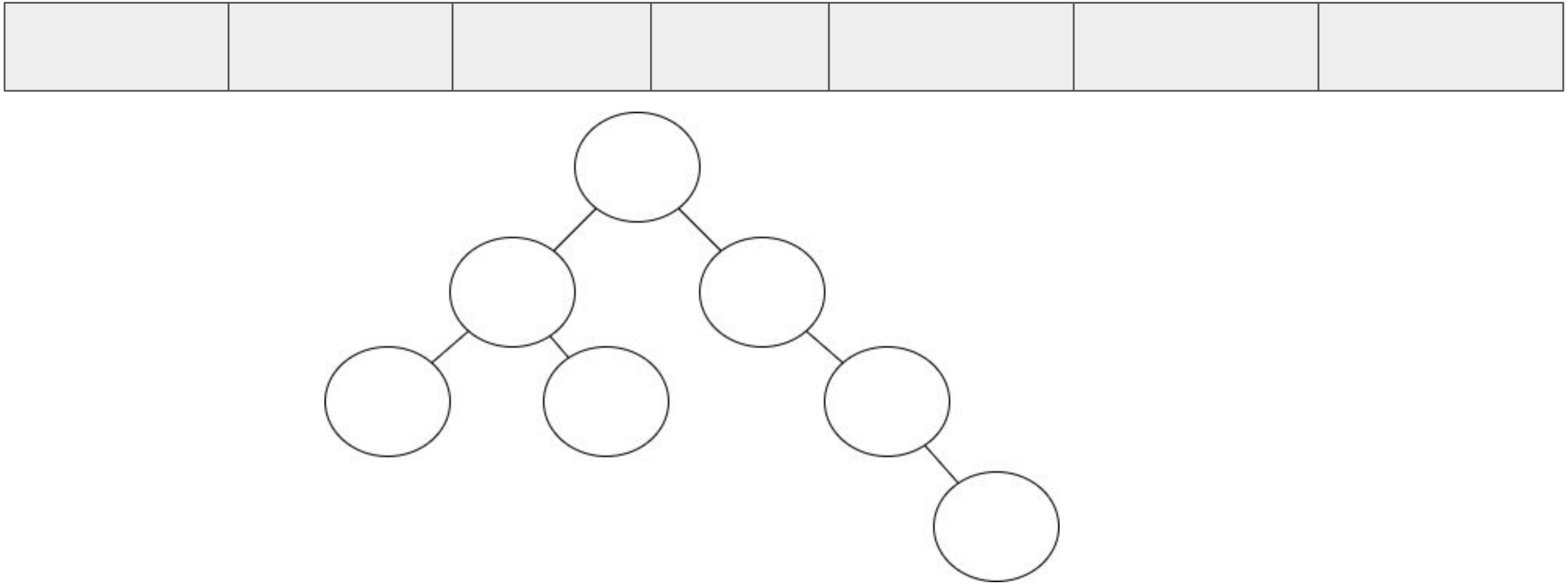
Post-order : L-R-N

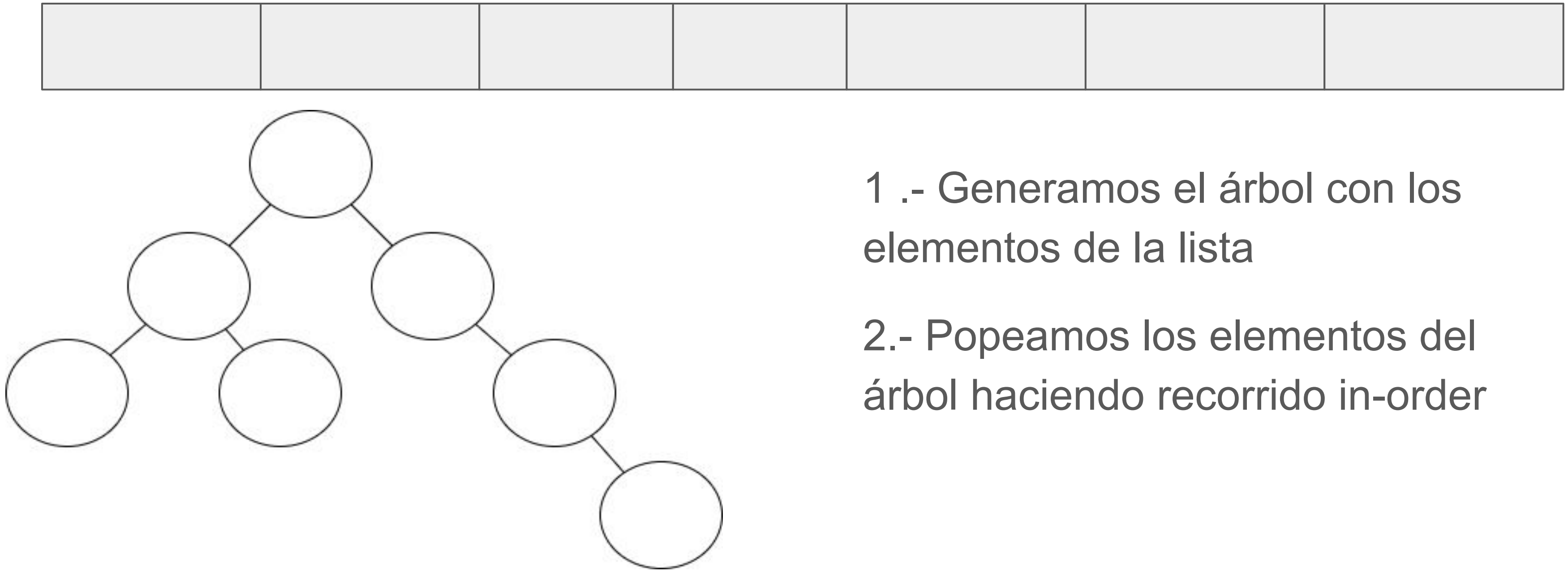


1. La propiedad fundamental de un ABB es que las claves almacenadas en el subárbol izquierdo son todas menores que la clave almacenada en la raíz, la que a su vez es menor que cualquiera de las claves almacenadas en el subárbol derecho. Teniendo presente esta propiedad, responde:

    a. Considera que tienes un ABB vacío T sin autobalance y una lista desordenada L de n números. ¿Cómo puedes utilizar T para ordenar L? ¿Cuál sería la complejidad de este algoritmo en notación Ω? ¿Qué características tiene L cuando se da este caso? Da un ejemplo con $n = 11$.

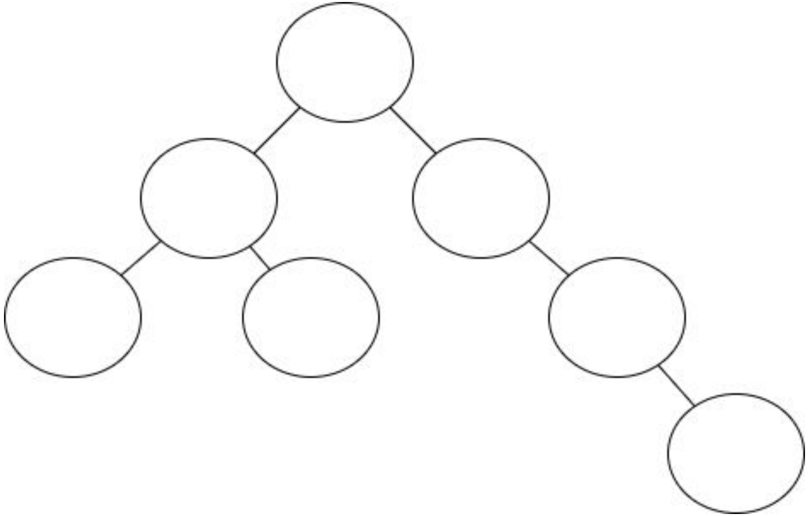
    b. Definimos B_x como los nodos en la ruta de búsqueda de una hoja x en un árbol T. Definimos A_x como todos los nodos a la izquierda de B_x, y C_x como todos los nodos a la derecha de B_x. ¿Es posible que haya un nodo en C_x de clave menor a la clave de un nodo en A_x?
    Si la respuesta es sí, da un ejemplo. En caso contrario, demuestra que no es posible.





1 .- Generamos el árbol con los elementos de la lista

2.- Popeamos los elementos del árbol haciendo recorrido in-order



Complejidad ? $\Omega(n \cdot log(n))$

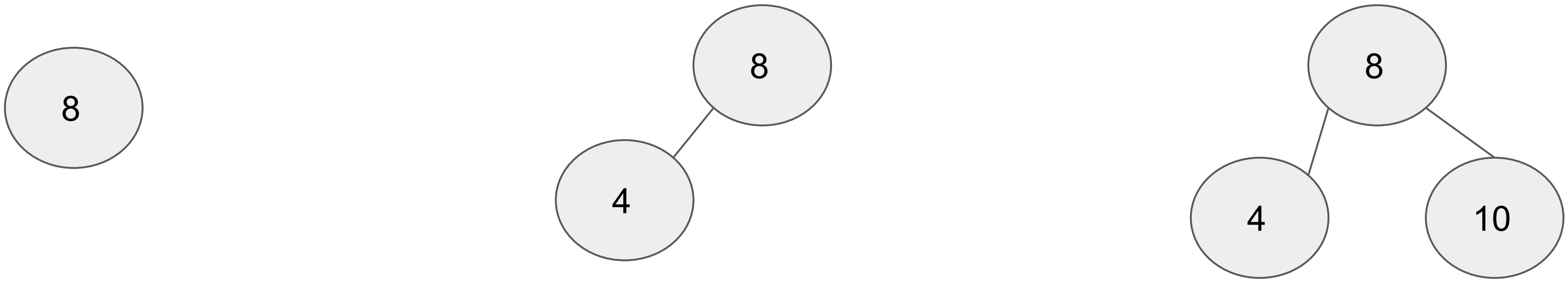
Pq ?

n inserciones

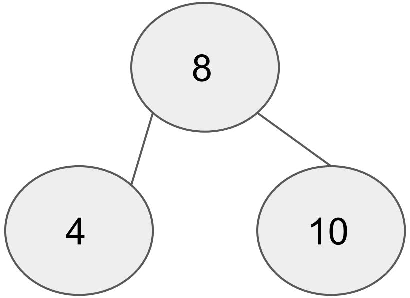
cada una toma $\mathcal{O}(log(n$

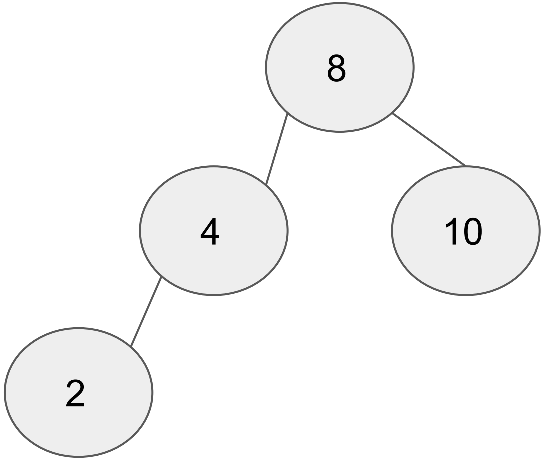
8	4	10	2	6	9	11	1	3	5	7

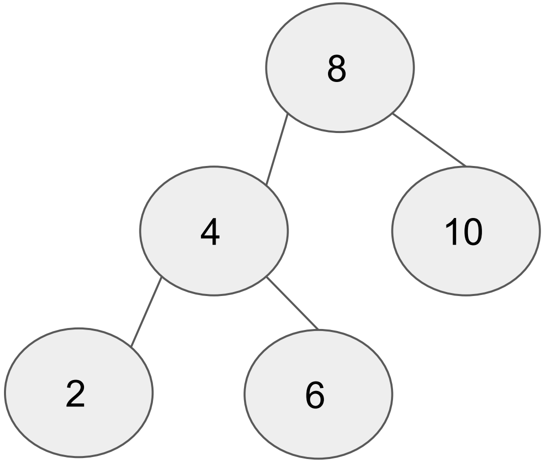
8	4	10	2	6	9	11	1	3	5	7



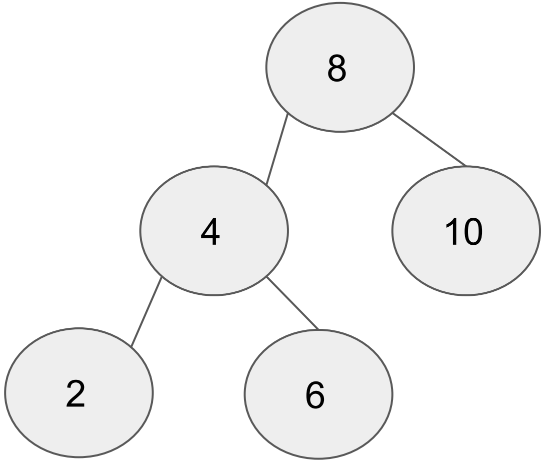
8	4	10	2	6	9	11	1	3	5	7

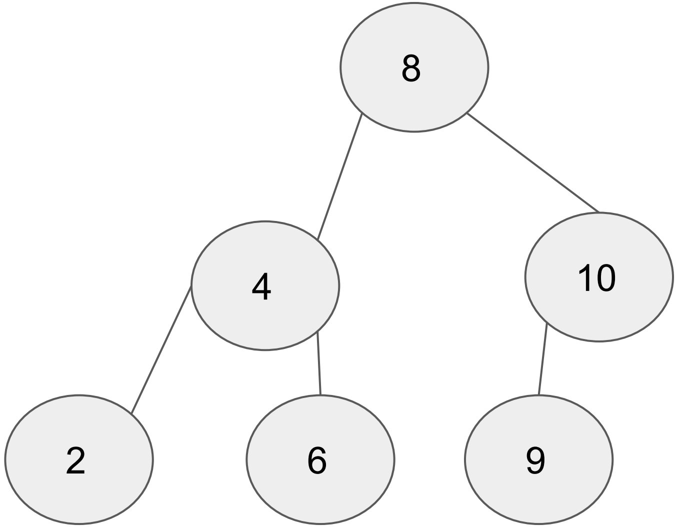




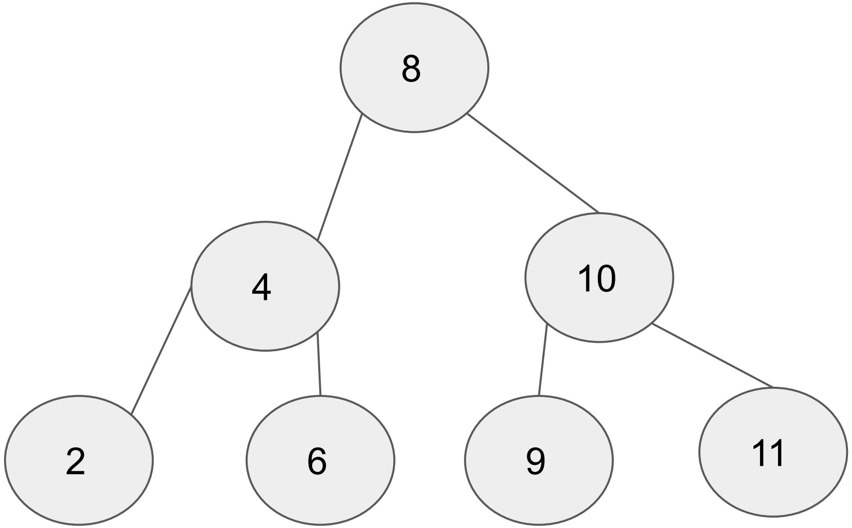


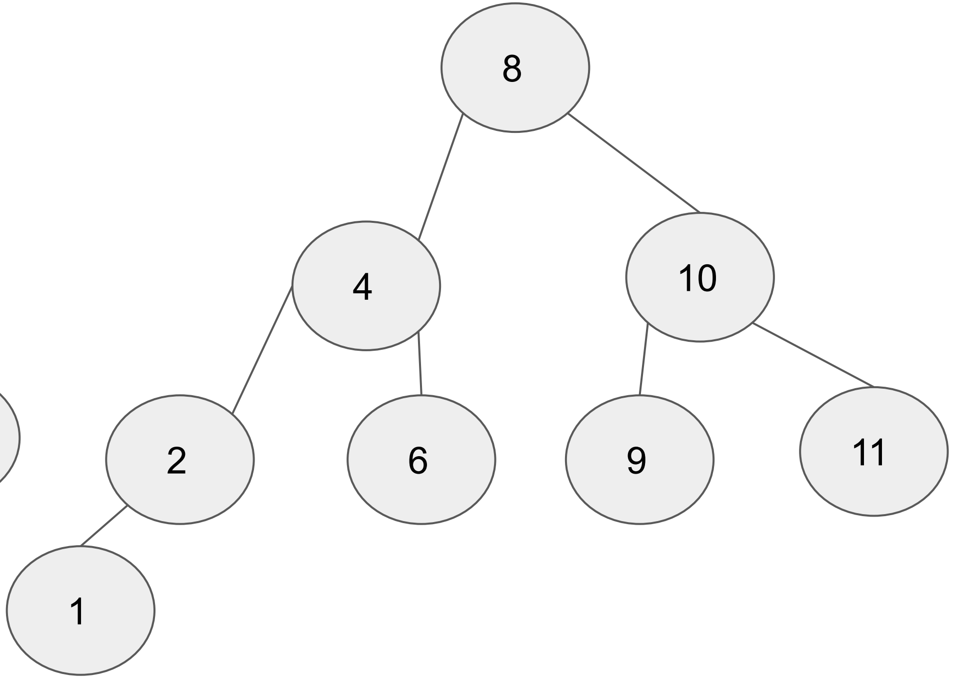
8	4	10	2	6	9	11	1	3	5	7



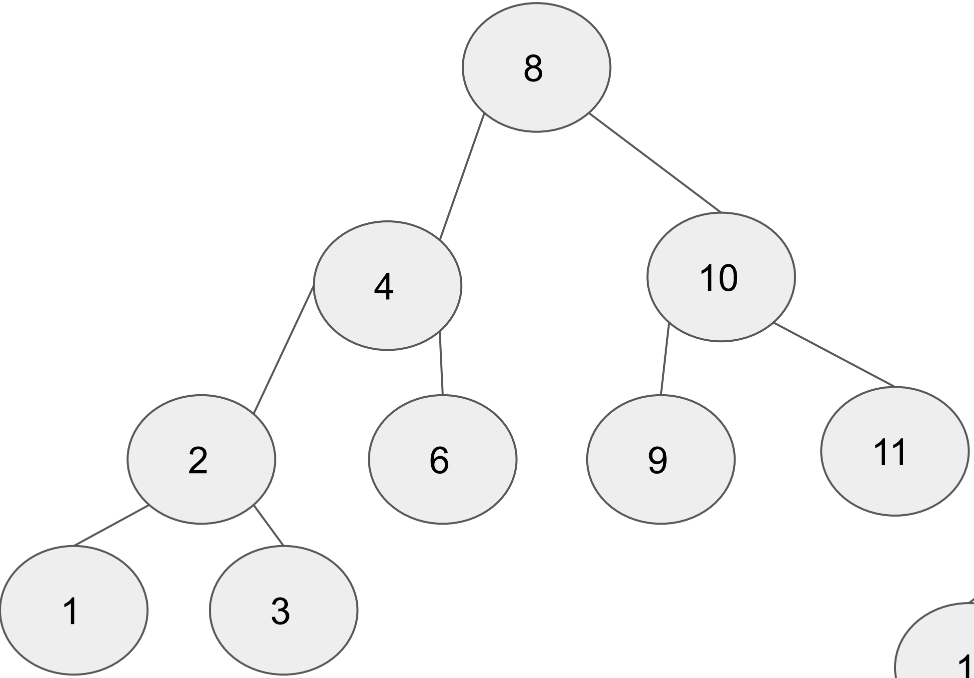


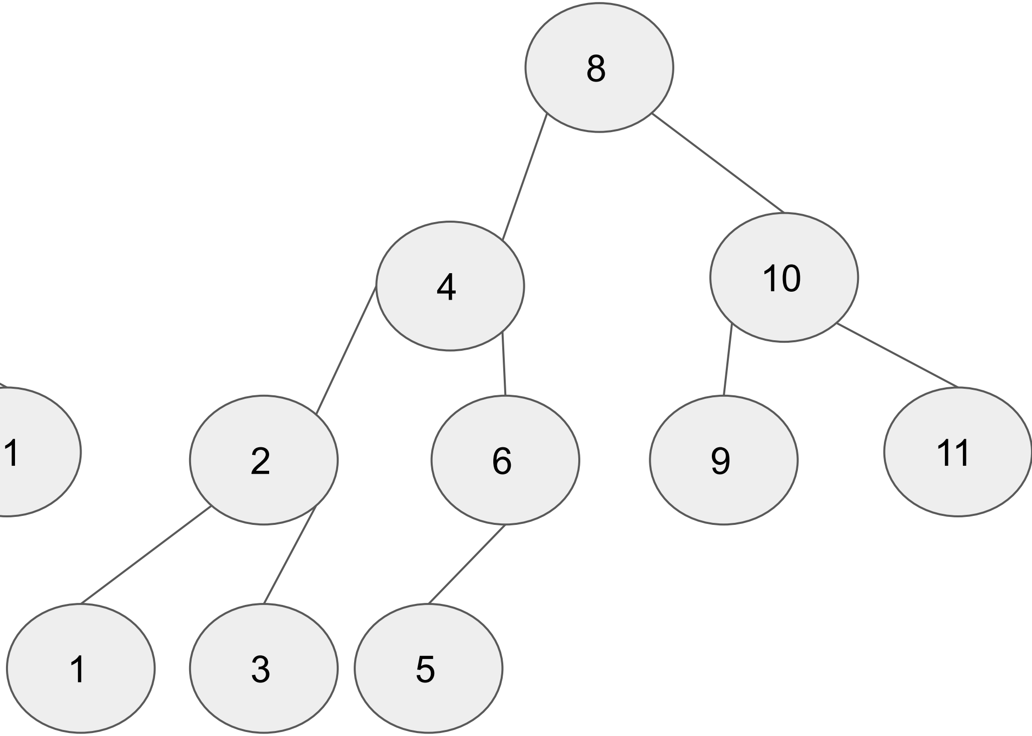
8	4	10	2	6	9	11	1	3	5	7



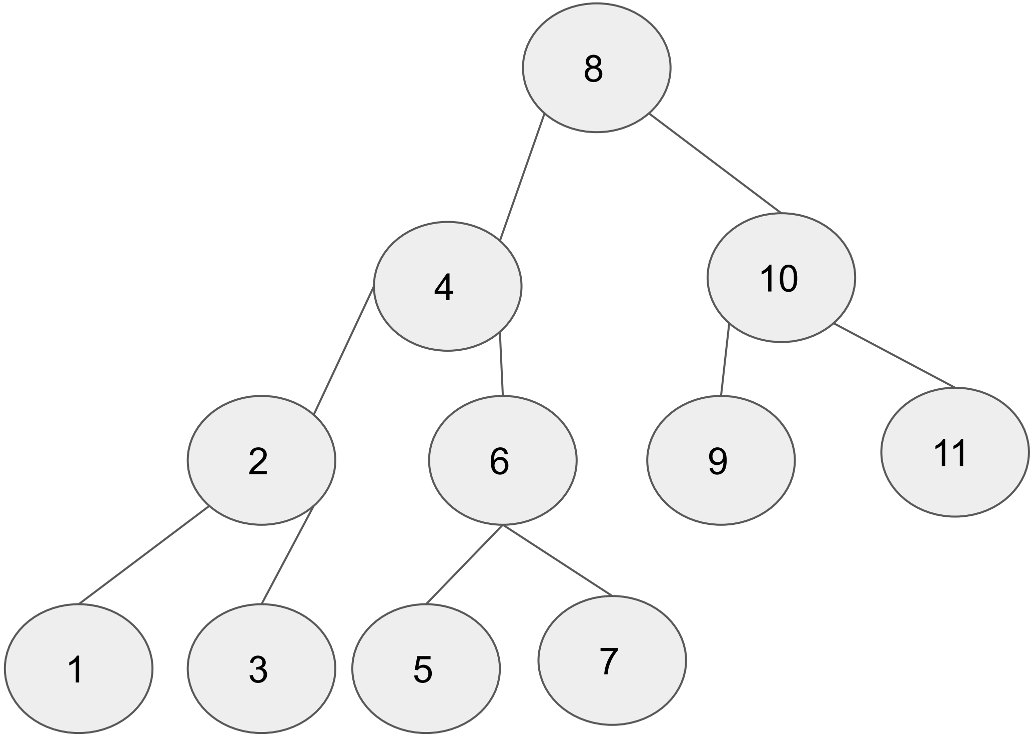


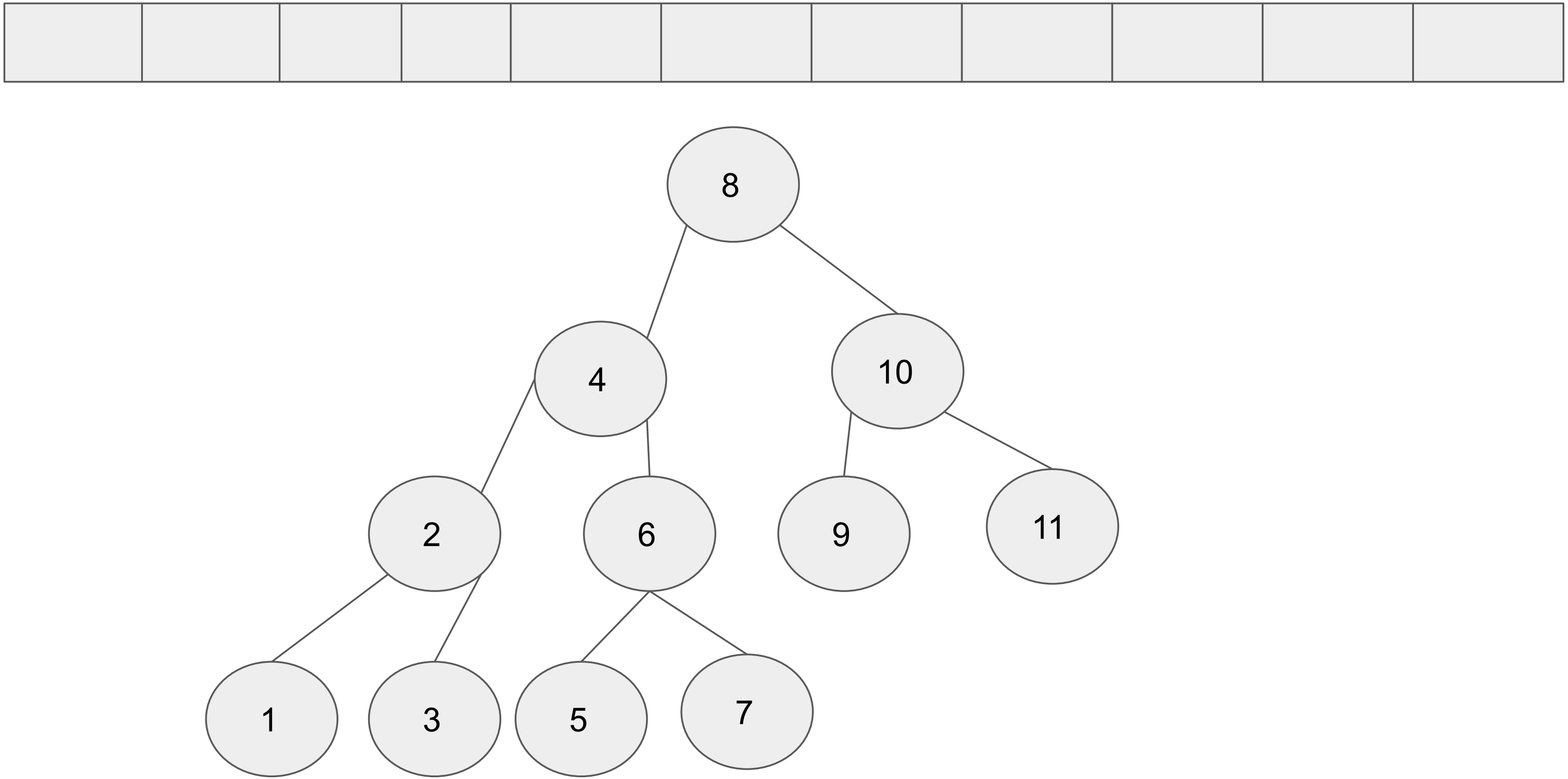
8	4	10	2	6	9	11	1	3	5	7



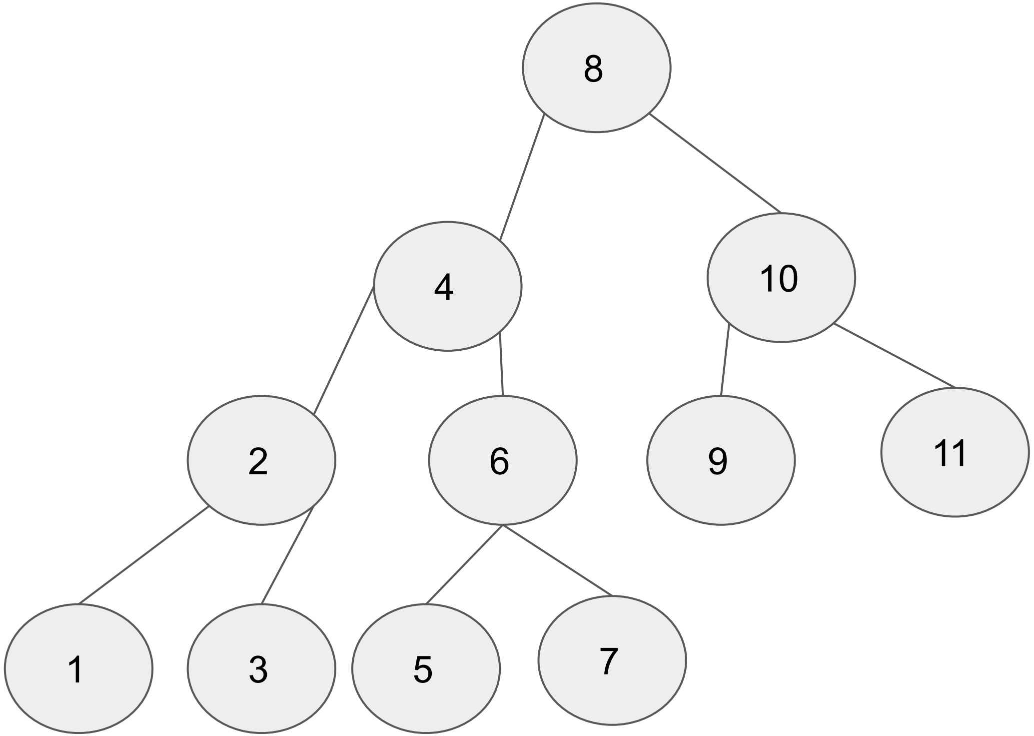


8	4	10	2	6	9	11	1	3	5	7



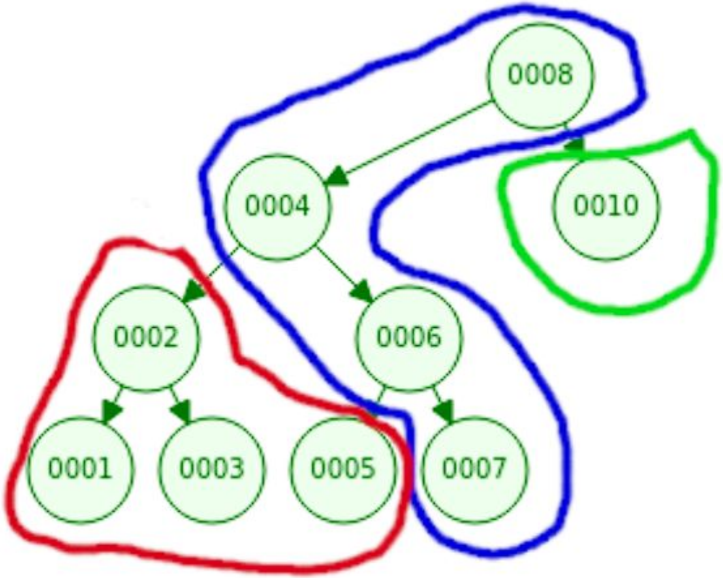

8
4
10
2
6
9
11
1
3
5
7

1	2	3	4	5	6	7	8	9	10	11



b. Definimos B_x como los nodos en la ruta de búsqueda de una hoja x en un árbol T. Definimos A_x como todos los nodos a la izquierda de B_x, y C_x como todos los nodos a la derecha de B_x. ¿Es posible que haya un nodo en C_x de clave menor a la clave de un nodo en A_x?
Si la respuesta es sí, da un ejemplo. En caso contrario, demuestra que no es posible.

b. Definimos B_x como los nodos en la ruta de búsqueda de una hoja x en un árbol T. Definimos A_x como todos los nodos a la izquierda de B_x, y C_x como todos los nodos a la derecha de B_x. ¿Es posible que haya un nodo en C_x de clave menor a la clave de un nodo en A_x? Si la respuesta es sí, da un ejemplo. En caso contrario, demuestra que no es posible.



b. Definimos B_x como los nodos en la ruta de búsqueda de una hoja x en un árbol T. Definimos A_x como todos los nodos a la izquierda de B_x, y C_x como todos los nodos a la derecha de B_x. ¿Es posible que haya un nodo en C_x de clave menor a la clave de un nodo en A_x?
Si la respuesta es sí, da un ejemplo. En caso contrario, demuestra que no es posible.

b. Definimos B_x como los nodos en la ruta de búsqueda de una hoja x en un árbol T. Definimos A_x como todos los nodos a la izquierda de B_x, y C_x como todos los nodos a la derecha de B_x. ¿Es posible que haya un nodo en C_x de clave menor a la clave de un nodo en A_x?
Si la respuesta es sí, da un ejemplo. En caso contrario, demuestra que no es posible.

$$B_x = \{b \mid b \text{ está en la ruta de búsqueda hacia } x\}$$

$$A_x = \{a \mid a \notin B_x \land (a \text{ es hijo izquierdo de un nodo en } B_x \text{ o es descendiente de otro elemento en } A_x)\}$$

$$C_x = \{c \mid c \notin B_x \land (c \text{ es hijo derecho de un nodo en } B_x \text{ o es descendiente de otro elemento en } C_x)\}$$

$$B_x = \{b \mid b \text{ está en la ruta de búsqueda hacia } x\}$$

$$A_x = \{a \mid a \notin B_x \wedge (a \text{ es hijo izquierdo de un nodo en } B_x \text{ o es descendiente de otro elemento en } A_x)\}$$

$$C_x = \{c \mid c \notin B_x \wedge (c \text{ es hijo derecho de un nodo en } B_x \text{ o es descendiente de otro elemento en } C_x)\}$$

Para $a \in A_x$, tenemos 2 casos:

- a es hijo izquierdo de un nodo en B_x: Llamemos a dicho nodo "b". Por definición de ABB, a es menor a todo el subárbol derecho de b. Pero por definición de B_x, b forma parte de la ruta hacia x, y como $a \notin B_x$, x tiene que estar en el subárbol derecho de b. Por lo tanto, $a < x$.
- a es descendiente de otro elemento en A_x: Por definición de ABB, si un nodo dado es menor a un número, todos sus descendientes también cumplen con esa desigualdad, sin importar la profundidad. Todos los a que surgen de este caso son descendientes de los a del primer caso, por lo que también se cumple que $a < x$

Es decir,

$$\forall a \in A_x,\ a < x$$

Análogamente,

$$\forall c \in C_x,\ x < c$$

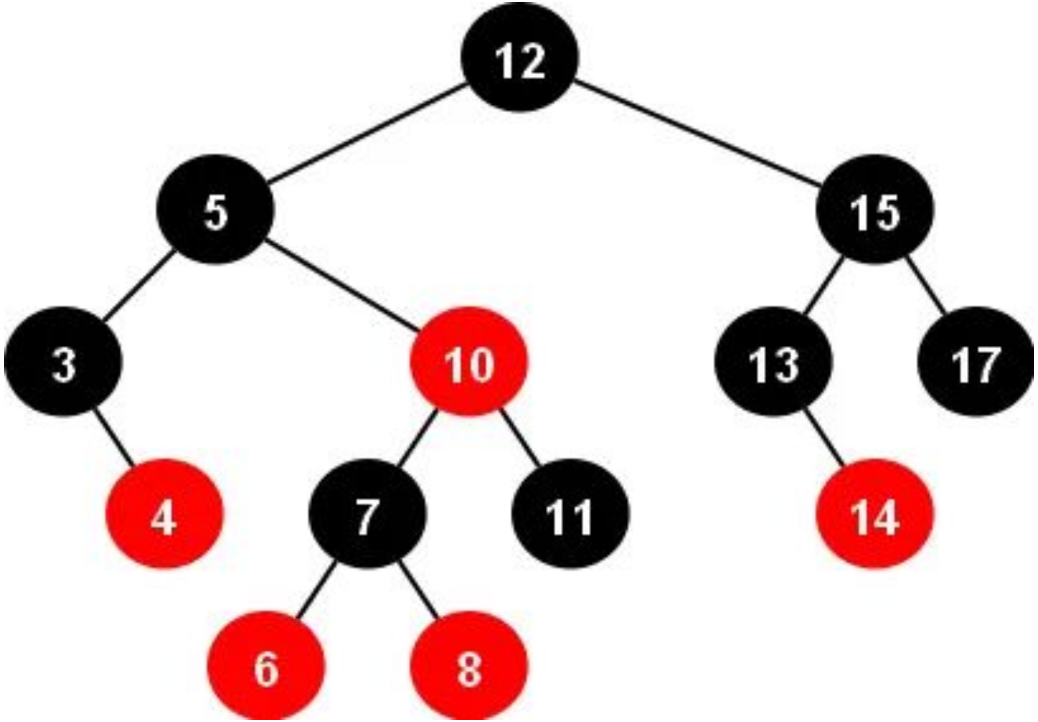
Por lo tanto,

$$\forall a \in A_x,\ \forall c \in C_x,\ a < c$$

Rojo Negro

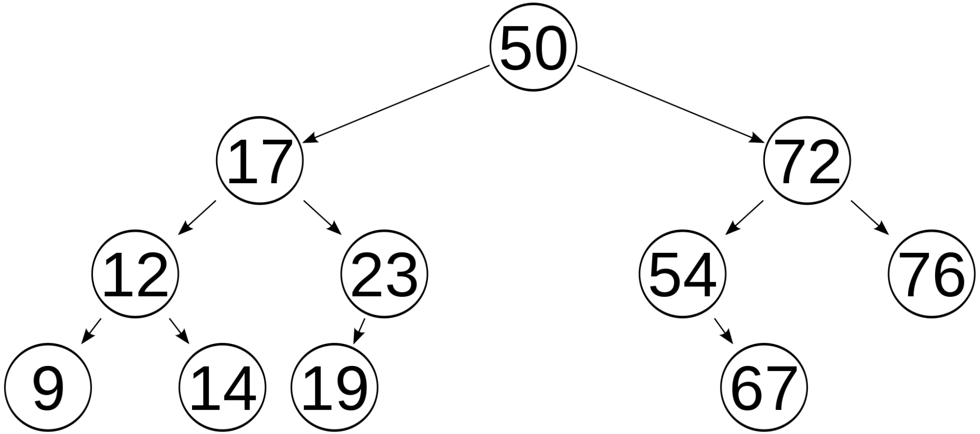
- o es rojo o es negro
- root -> negro
- hojas -> negras
- si un nodo es rojo, sus hijos son negros
- $\forall$ camino simple de un nodo a sus hojas, deben haber la misma cantidad de nodos negros.

Operations : https://www.youtube.com/watch?v=UaLIHuR1t8Q&t=605s


12
5
15
3
10
13
17
4
7
11
14
6
8

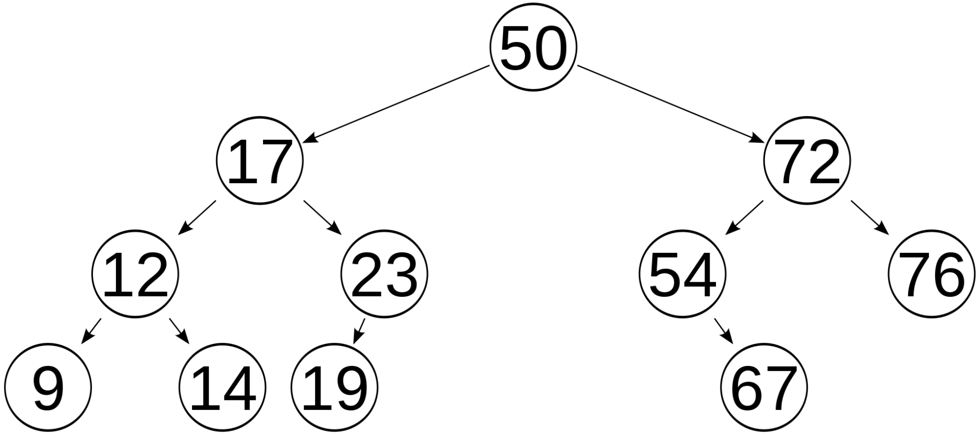
- Es todo Árbol AVL un rojo negro ?

- Es todo Árbol AVL un rojo negro ?



- Es todo Árbol AVL un rojo negro ?

Si !

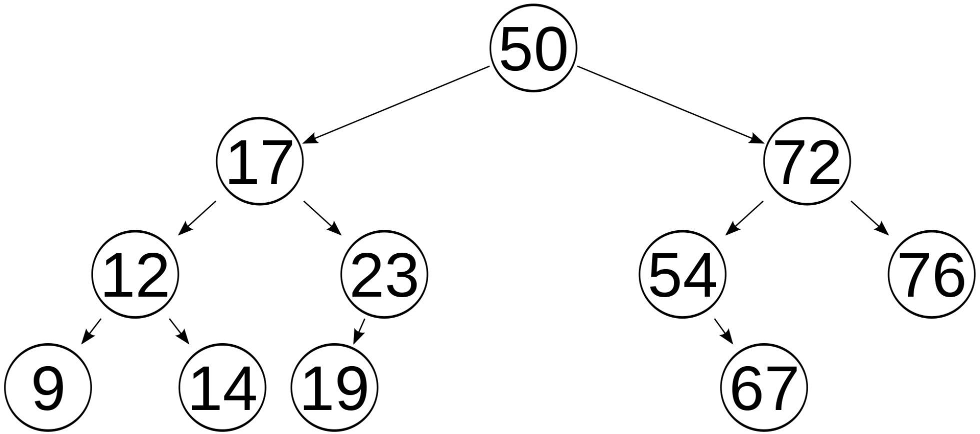


- Es todo Árbol AVL un rojo negro ?

Si !

Notamos que el árbol AVL mas “desvalanceado” es aquel que el sub-arbol derecho tiene 1 mas de altura que su correspondiente sub-arbol izquierdo

Esto significa que la cantidad de nodos en la ruta más larga crece de forma 2H y la de la ruta más corta h+1, h altura del árbol



BackanTrack

- Dominio ?
- Mejoras :

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

BackanTrack

- Dominio ? {1, 2, 3, 4, 5, 6, 7, 8, 9}
- Mejoras :

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

BackanTrack

- Dominio ? {1, 2, 3, 4, 5, 6, 7, 8, 9}
- Mejoras :

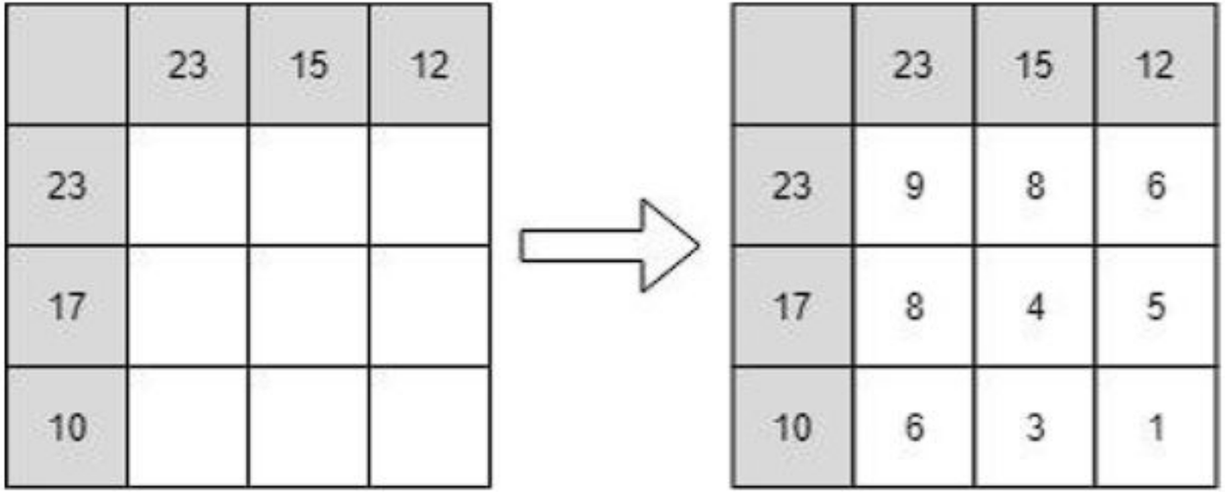
  comenzar resolviendo el bloque

  mas “rellenado” (Heurística)

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

BackanTrack

Kakuro :



BackanTrack

Variables ?

Dominio ?

	23	15	12
23			
17			
10			

	23	15	12
23	9	8	6
17	8	4	5
10	6	3	1

BackanTrack

Variables : celdas

Dominio : { 1, ,,, MAX }

	23	15	12
23			
17			
10			

	23	15	12
23	9	8	6
17	8	4	5
10	6	3	1

Hashing

Una tienda quiere premiar a sus ***k*** clientes más rentables. Para ello cuenta con la lista de las ***n*** compras de los últimos años, en que cada compra es una tupla de la forma *(ID Cliente, Monto)*. La rentabilidad de un cliente es simplemente la suma de los montos de todas sus compras.
Explica cómo usar tablas de hash y heaps para resolver este problema en tiempo esperado —o promedio- $O(n + n\log(k))$.

El problema se separa en dos partes.

- Calcular la rentabilidad de cada cliente
- Buscar los *k* clientes más rentables

Hashing

Calcular la rentabilidad de cada cliente

Queremos obtener el set de tuplas *(ID Cliente, Rentabilidad)*, donde la rentabilidad para el ID Cliente ***i*** es la suma de todos los montos de las tuplas de la forma *(i, Monto)*.

Para esto creamos una tabla de hash ***T*** donde se almacenan tuplas *(ID Cliente, Rentabilidad)*. Cada vez que se inserte un ID Cliente *:*

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

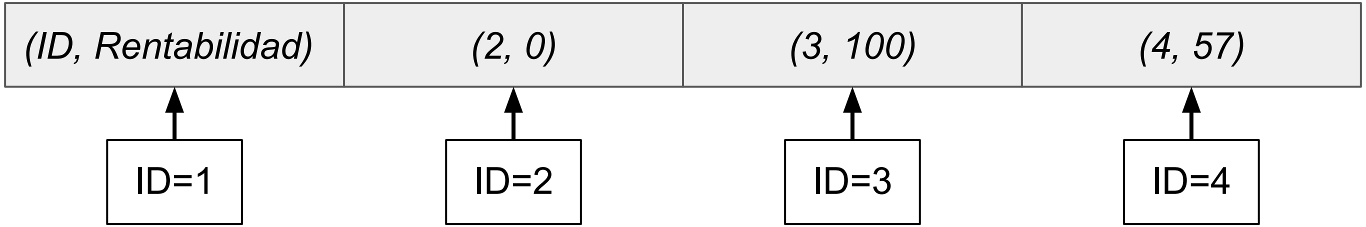
Hashing

Calcular la rentabilidad de cada cliente

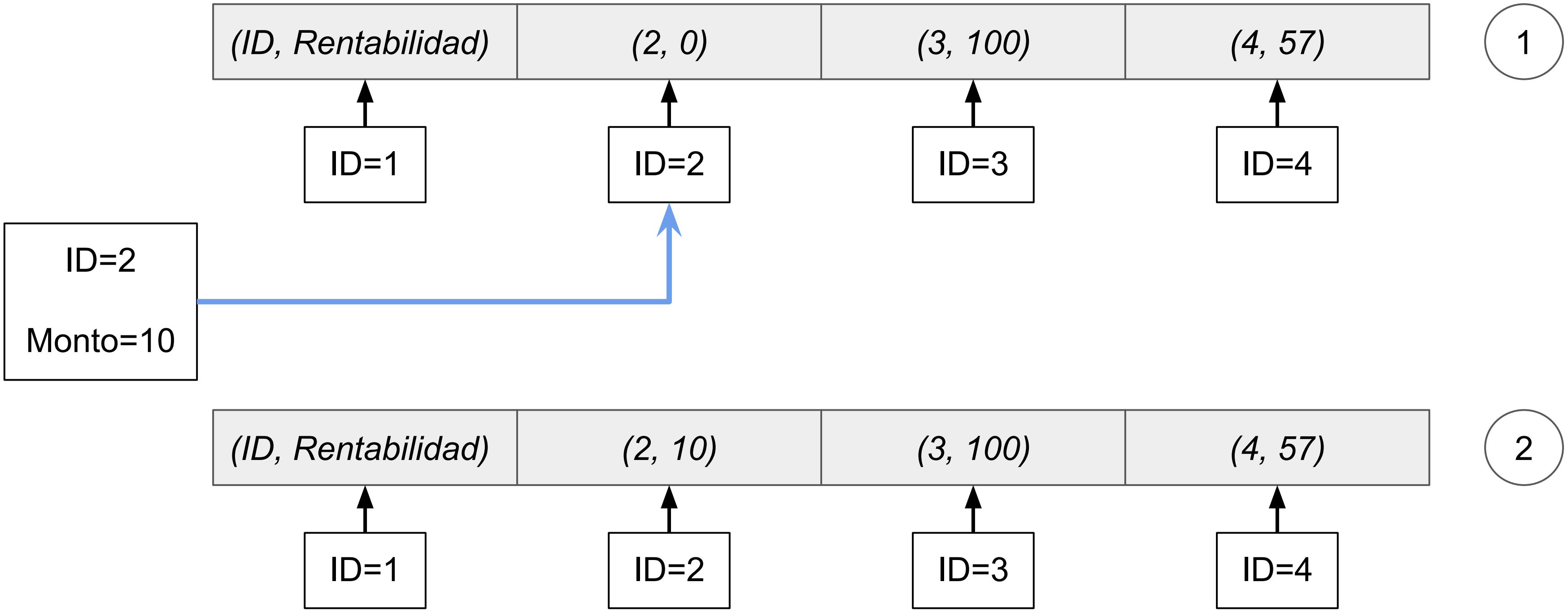
Queremos obtener el set de tuplas *(ID Cliente, Rentabilidad)*, donde la rentabilidad para el ID Cliente ***i*** es la suma de todos los montos de las tuplas de la forma *(i, Monto)*.

Para esto creamos una tabla de hash ***T*** donde se almacenan tuplas *(ID Cliente, Rentabilidad)*. Cada vez que se inserte un ID Cliente *i*:

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.



Hashing


(ID, Rentabilidad)
(2, 0)
(3, 100)
(4, 57)
1
ID=1
ID=2
ID=3
ID=4
ID=2
Monto=10
(ID, Rentabilidad)
(2, 10)
(3, 100)
(4, 57)
2
ID=1
ID=2
ID=3
ID=4

Hashing

Calcular la rentabilidad de cada cliente

Queremos obtener el set de tuplas *(ID Cliente, Rentabilidad)*, donde la rentabilidad para el ID Cliente ***i*** es la suma de todos los montos de las tuplas de la forma *(i, Monto)*.

Para esto creamos una tabla de hash ***T*** donde se almacenan tuplas *(ID Cliente, Rentabilidad)*. Cada vez que se inserte un ID Cliente *i*:

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

Al hacer esto con todas las *n* tuplas hemos efectivamente encontrado la rentabilidad para cada cliente.

Hashing

Calcular la rentabilidad de cada cliente

Queremos obtener el set de tuplas *(ID Cliente, Rentabilidad)*, donde la rentabilidad para el ID Cliente ***i*** es la suma de todos los montos de las tuplas de la forma *(i, Monto)*.

Para esto creamos una tabla de hash ***T*** donde se almacenan tuplas *(ID Cliente, Rentabilidad)*. Cada vez que se inserte un ID Cliente :

- Si ya está en la tabla, se suma el monto recién insertado al monto guardado.
- Si no, se guarda en la tabla con el monto recién insertado como monto inicial.

Al hacer esto con todas las *n* tuplas hemos efectivamente encontrado la rentabilidad para cada cliente.

La inserción en esta tabla tiene tiempo esperado ***O(1)*** como se ha visto en clases. Como se realizan ***n*** inserciones, esta parte tiene tiempo esperado ***O(n)***.

Hashing

Buscar los k clientes más rentables

Sea m el total de clientes distintos. Creamos un min-heap de tamaño k que contendrá los k clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los ***k* más rentables** encontrados hasta el momento.

Iteramos sobre las tuplas en ***T***, insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre T .

Hashing

Buscar los *k* clientes más rentables

Sea *m* el total de clientes distintos. Creamos un min-heap de tamaño *k* que contendrá los *k* clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los ***k*** **más rentables** encontrados hasta el momento.

Iteramos sobre las tuplas en ***T***, insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre *T* .

Para cada elemento que veamos que sea mayor a la raíz, lo intercambiamos con esta. Luego hacemos *shift-down* para restaurar la propiedad del heap. Esto mantiene la propiedad descrita en el párrafo anterior.

Hashing

Buscar los *k* clientes más rentables

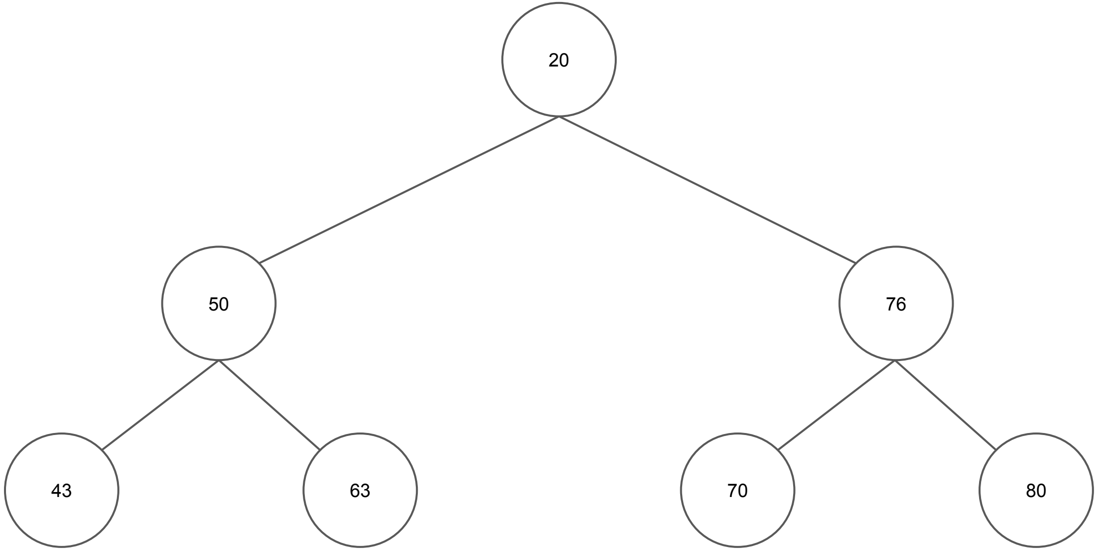
Sea *m* el total de clientes distintos. Creamos un min-heap de tamaño *k* que contendrá los *k* clientes más rentables que hemos visto hasta el momento. De este modo, la raíz del heap es el cliente **menos rentable** de los ***k* más rentables** encontrados hasta el momento.

Iteramos sobre las tuplas en ***T***, insertando en el heap hasta llenarlo, usando la rentabilidad como prioridad. Luego de haber llenado el heap, seguimos iterando sobre *T* .

Para cada elemento que veamos que sea mayor a la raíz, lo intercambiamos con esta. Luego hacemos *shift-down* para restaurar la propiedad del heap. Esto mantiene la propiedad descrita en el párrafo anterior.

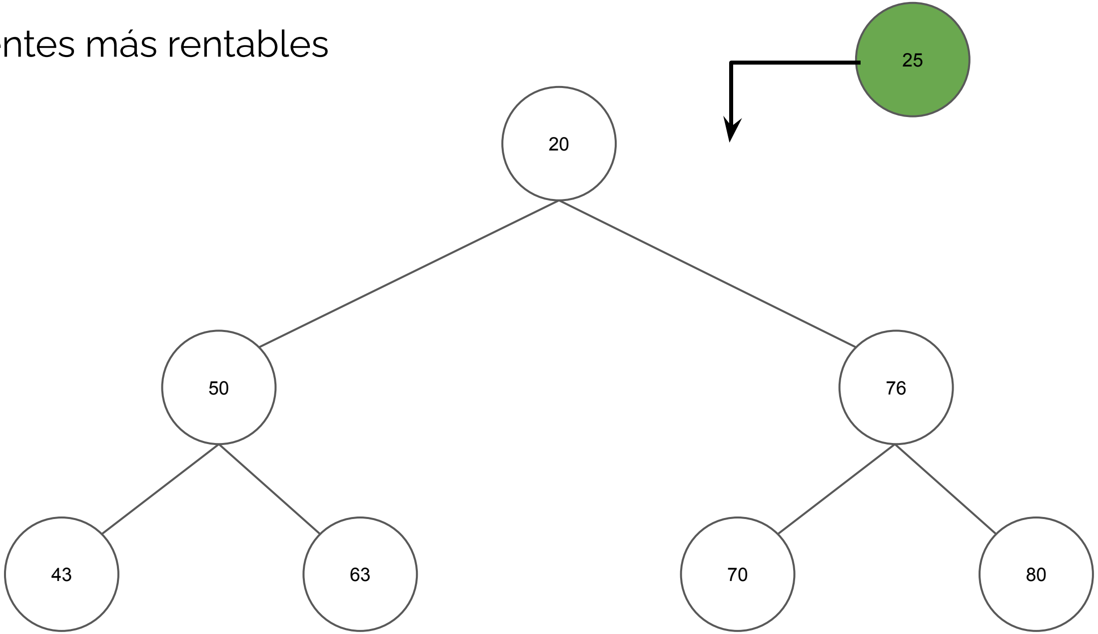
Hashing

Buscar los *k* clientes más rentables



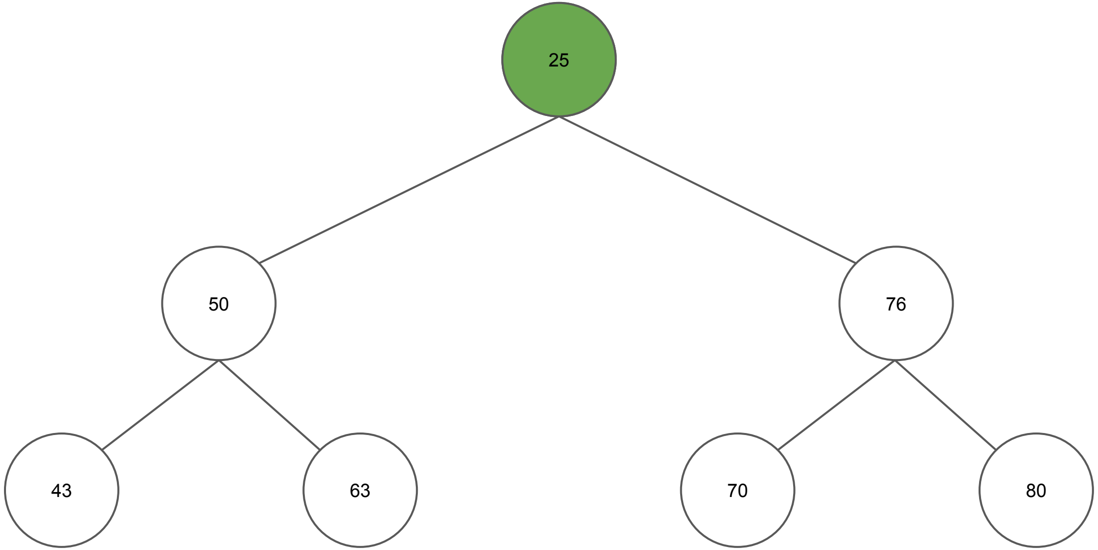
Hashing

Buscar los *k* clientes más rentables



Hashing

Buscar los *k* clientes más rentables



Hashing

Buscar los k clientes más rentables

Cada inserción en el heap toma $O(\log k)$. En el peor caso insertamos los m elementos en el heap, por lo que esta parte es $O(m \log(k))$. Pero como en el peor caso $m=n$, esta parte es $O(n \log k)$.