<!--Estudio Shonos-->

# Colas de Prioridad y AVL{

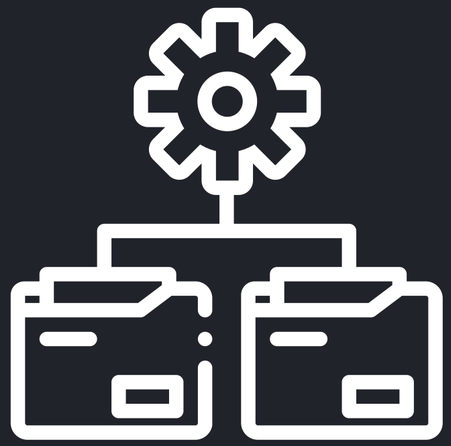
<Por="Carlos Barrientos"/>
<Por="Julio Ruiz"/>
<Por="Eddie Giron"/>
<Por="Samer Aranki"/>
<Por="Herbert Alfaro"/>

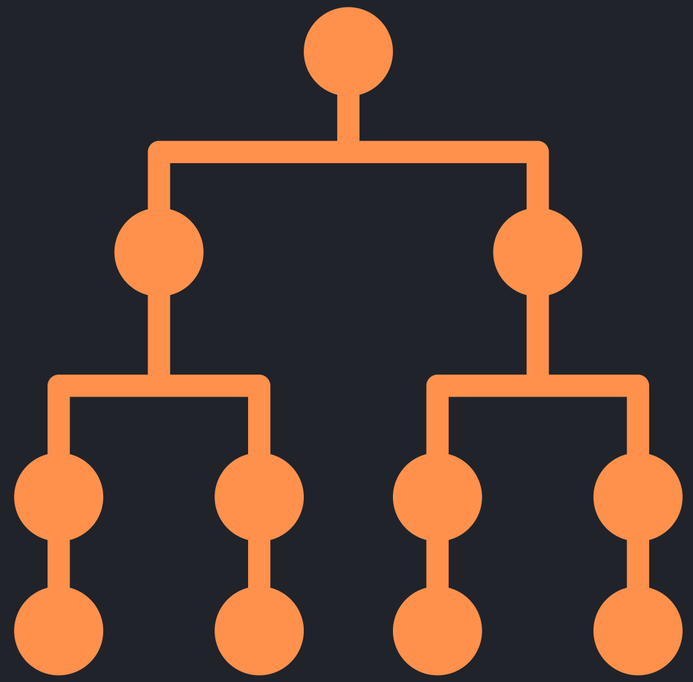
}

# Contenidos

# Introducción {

Los arboles AVL y Colas de Prioridad son estructuras de datos que nos ayudan como Ingenieros en Informática y Sistemas a organizar y sistematizar datos de una manera mas ordenada y trabajar de formas mas eficientes cuando se trata de dejar un sistema de la mejor forma posible.
Por lo cual es importante tener bien en cuenta como funciona cada una de estas estructuras de datos.





}

# Cola de Prioridad Teoría{

## ¿Qué es una Cola de Prioridad?

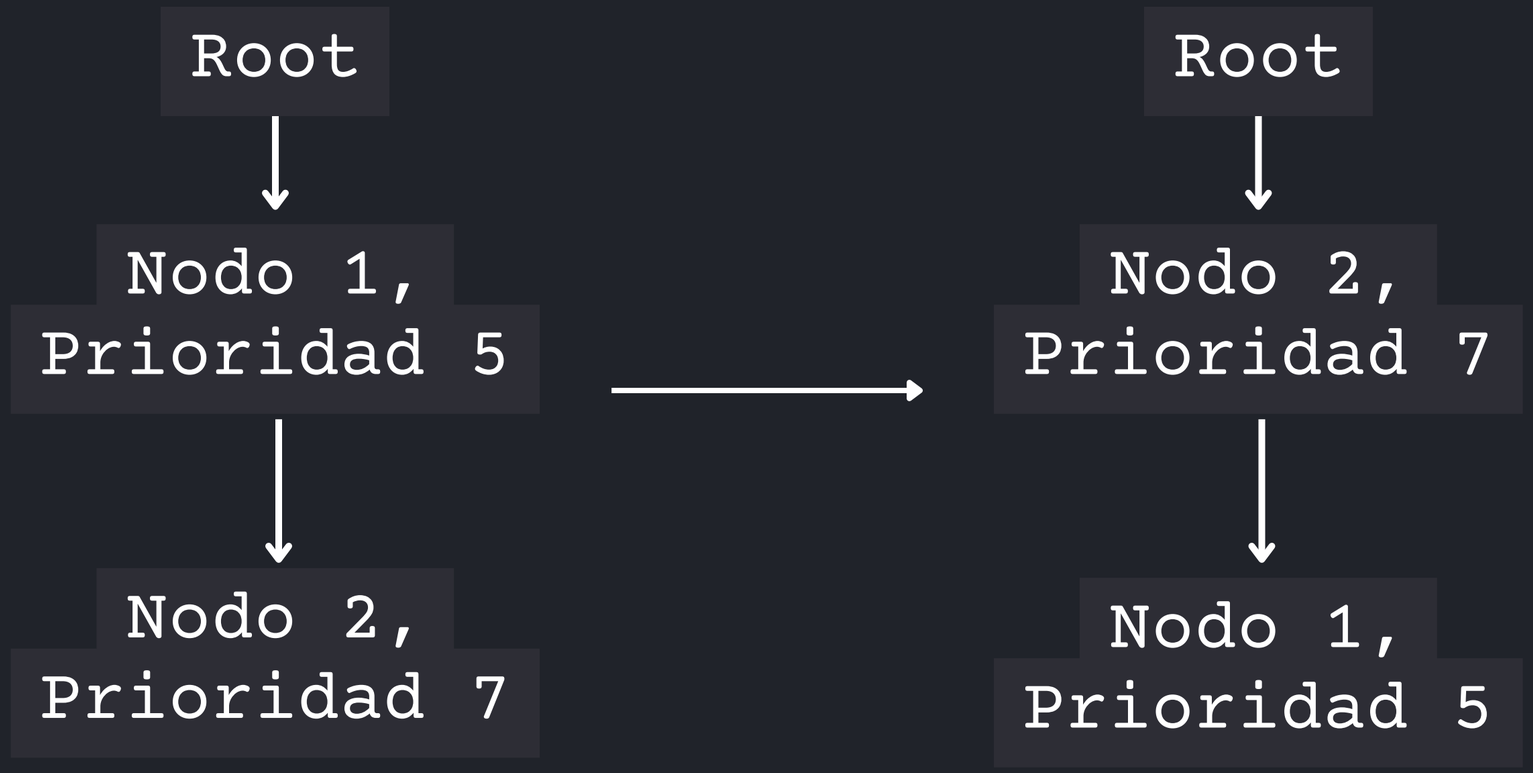
La cola de prioridad es un tipo de estructura de datos de tipo cola que organiza los elementos en función de su prioridad. Hay distintos tipos, como las implementadas con arreglos y las basadas en Heap. En las colas de prioridad, los elementos se manejan en orden de prioridad, con los más importantes manejados primero

## ¿Para que sirve una Cola de Prioridad?

La organización de prioridades tiene múltiples aplicaciones en nuestra vida diaria, como la priorización de la ruta más corta en grafos y su uso en aplicaciones como Waze. También se aplica en las peticiones a servidores y en los procesos de pago en páginas de transacciones.

}

# Estructura de Cola de Prioridad {




## ¿Como funciona una Cola de Prioridad?

Los datos entran como si fuera una cola normal (el primero que entra es el primero en salir), conforme los datos van entrando se ve la prioridad que tiene cada nodo y se acomodan de forma que el dato de mayor prioridad salga de primero de la estructura de datos.

}

<!---->

# Inserción {

<Ejemplo"max-heap"/>

}

<!---->

# El mejor de los casos {

<Ejemplo"max-heap"/>

}

> Este es un ejemplo de un Heap especial llamado máximum Heap o maxheap, el cual cumple con la condición de que todo nodo almacena un valor mayor o igual al de sus hijos. La definición es equivalente para un minheap.

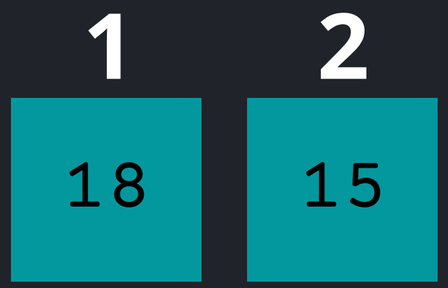
> El tiempo de inserción es logarítmico (se recorren log n nodos para llegar de la raíz a una hoja, la altura del árbol es logarítmica porque es binario y árbol completo) , es decir, si existen n elementos en la estructura la inserción se realiza en a lo mas k log n pasos (cuando decimos orden log n estamos asumiendo que hay una constante que multiplica por log n, generalmente pequeño).

La elección de comenzar en el índice 1 a menudo se debe a la simplicidad de las fórmulas para encontrar los índices de los nodos padre e hijo en el arreglo.
Cuando comienzas con el índice 1, las fórmulas para encontrar los índices del padre y los hijos de un nodo en un arreglo son las siguientes:

- Índice del padre: i // 2 (división entera)
- Índice del hijo izquierdo: 2 * i
- Índice del hijo derecho: 2 * i + 1

}

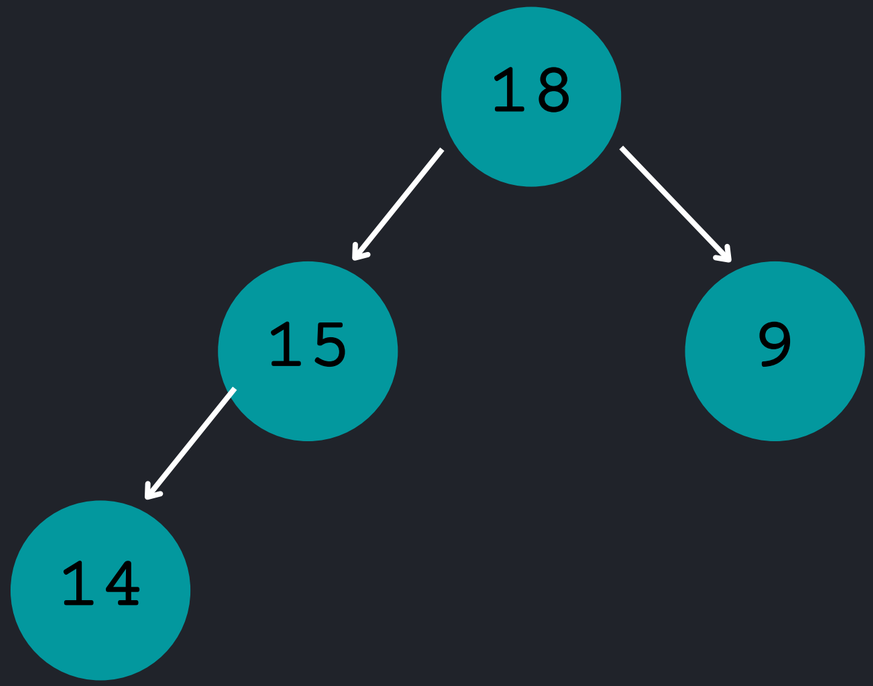
1



1
2
18
15

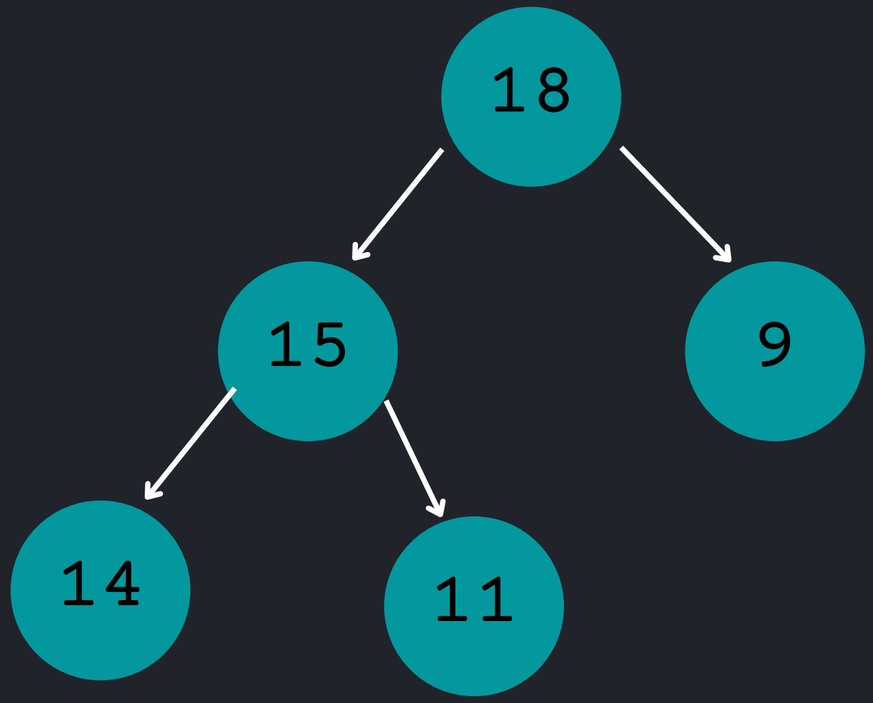

18
15

| 1 | 2 | 3 |
| --- | --- | --- |
| 18 | 15 | 9 |

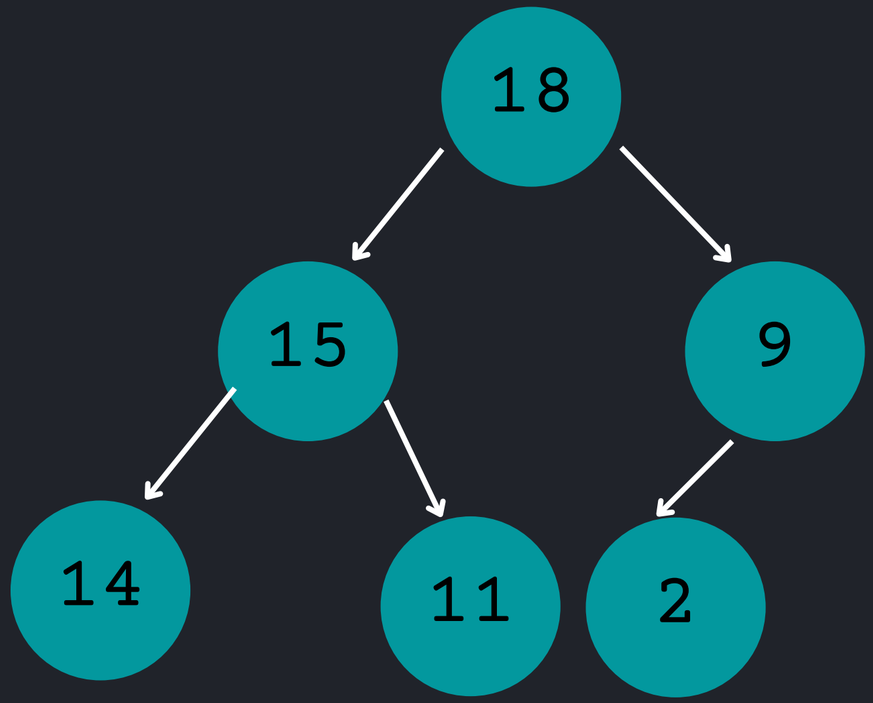
| 1 | 2 | 3 | 4 |
|---|---|---|---|
| 18 | 15 | 9 | 14 |



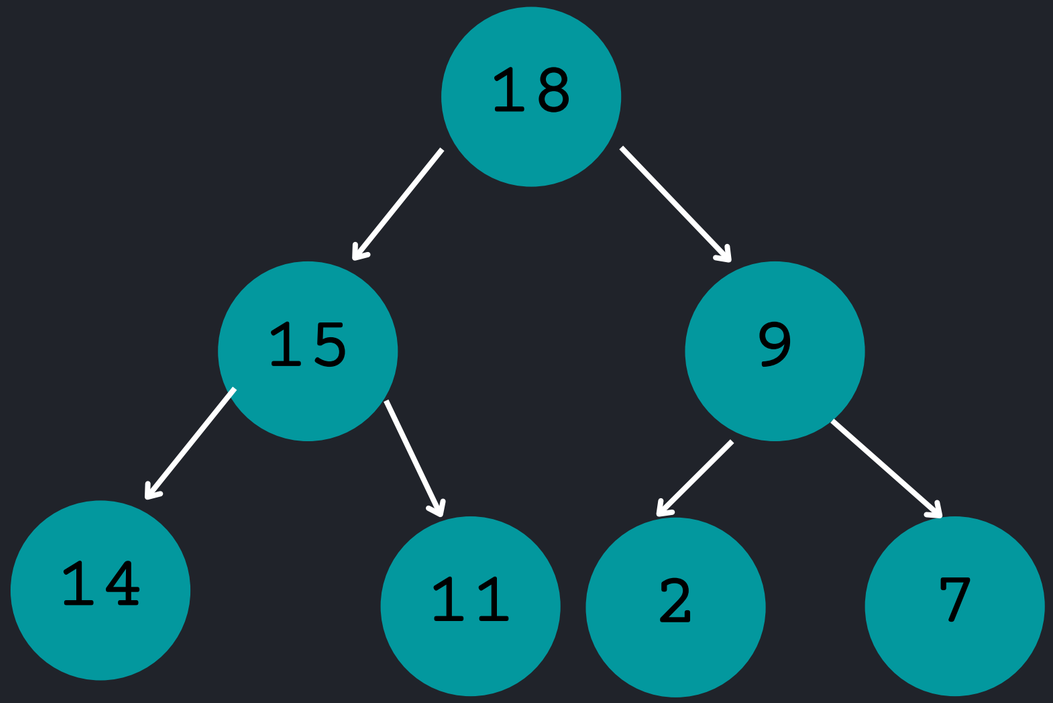
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 18 | 15 | 9 | 14 | 11 |



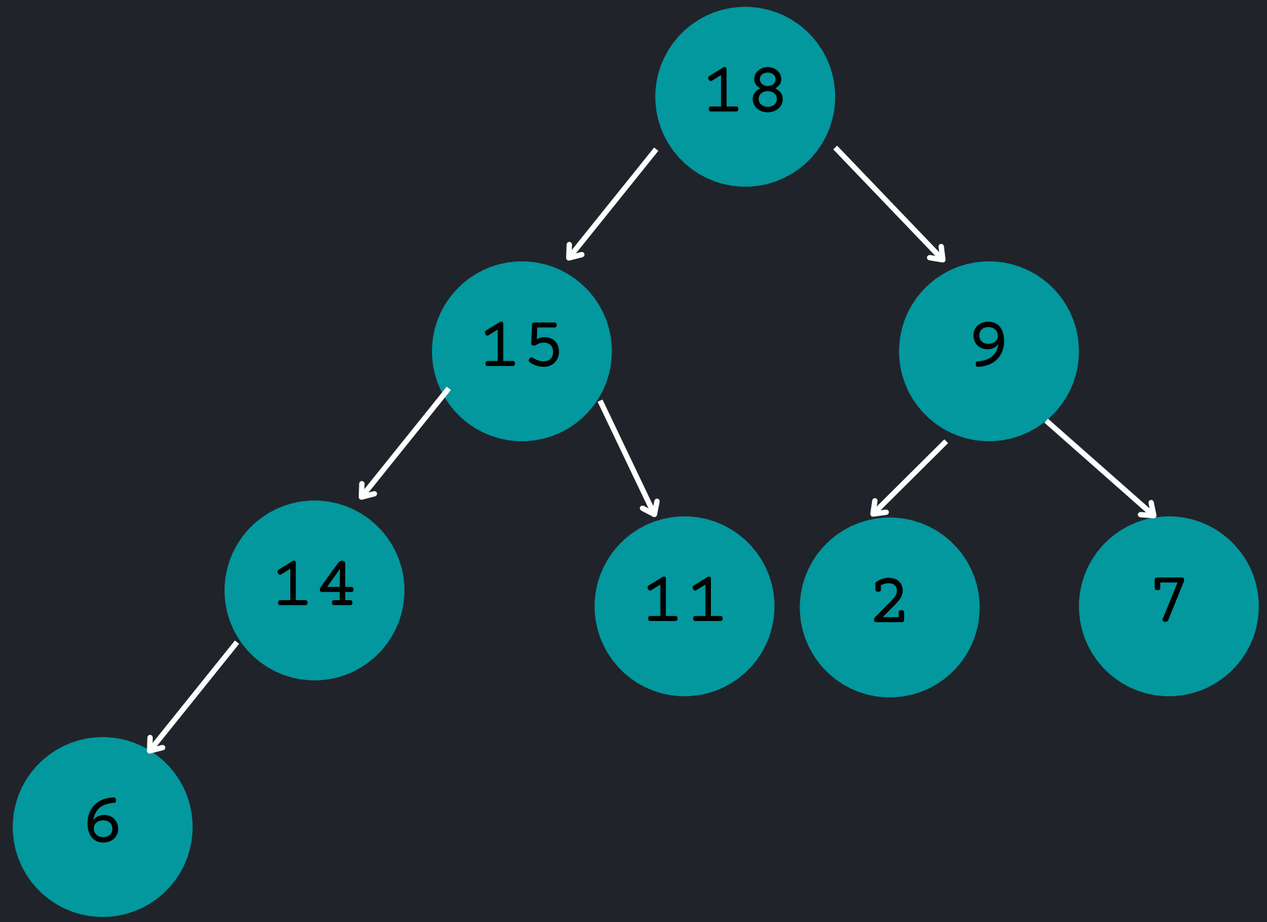
| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 18 | 15 | 9 | 14 | 11 | 2 |



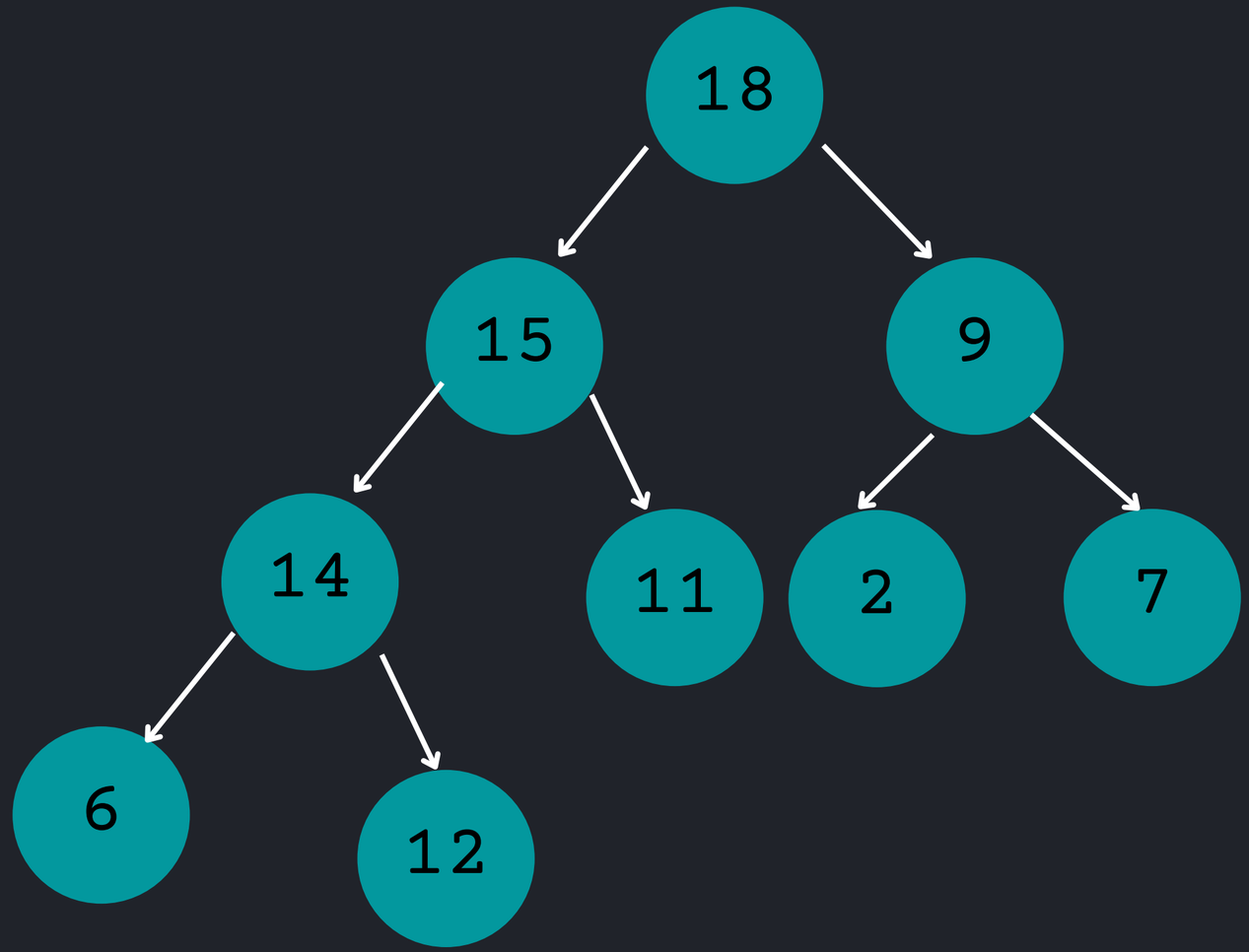
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 18 | 15 | 9 | 14 | 11 | 2 | 7 |



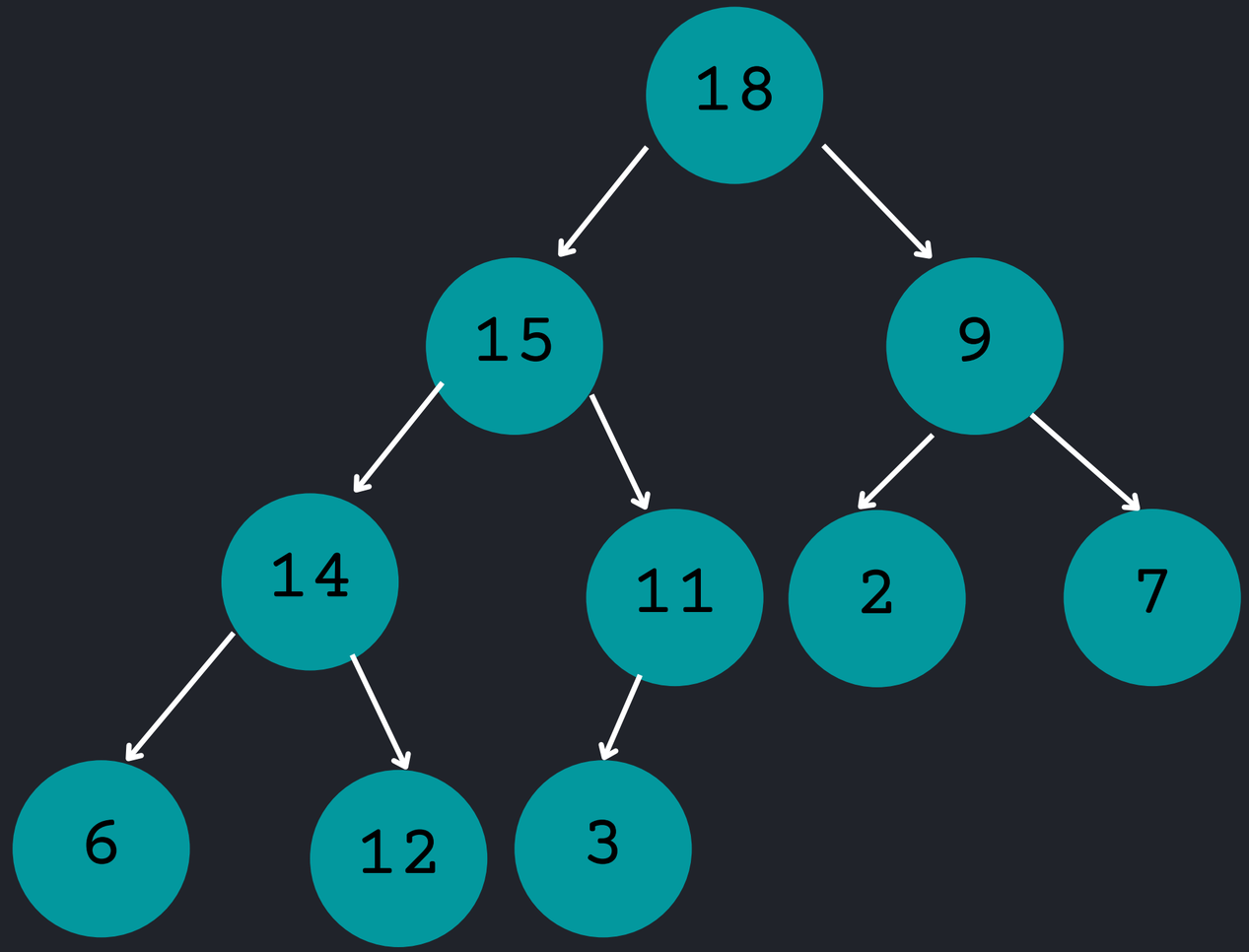
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 18 | 15 | 9 | 14 | 11 | 2 | 7 | 6 |



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 18 | 15 | 9 | 14 | 11 | 2 | 7 | 6 | 12 |



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 18 | 15 | 9 | 14 | 11 | 2 | 7 | 6 | 12 | 3 |



<!---->

# El peor de los casos {

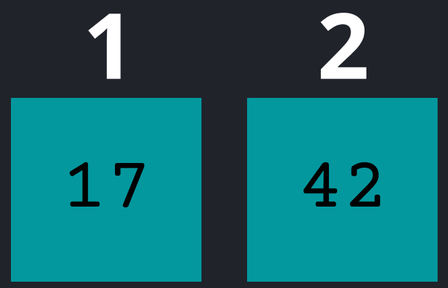
<Ejemplo"max-heap"/>

}

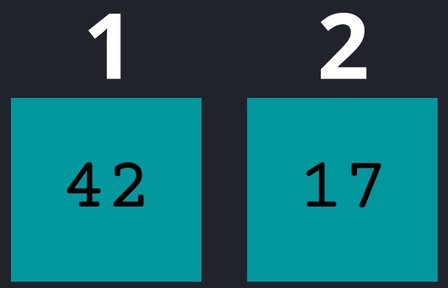
1

17

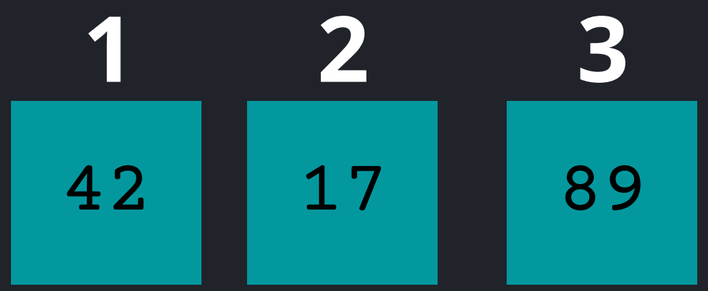
17

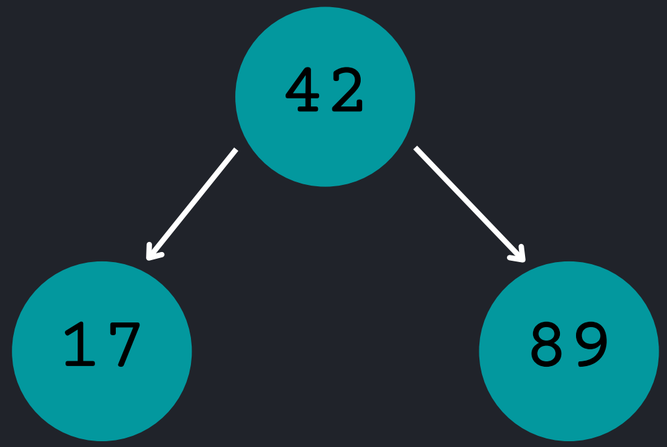

1
2
17
42

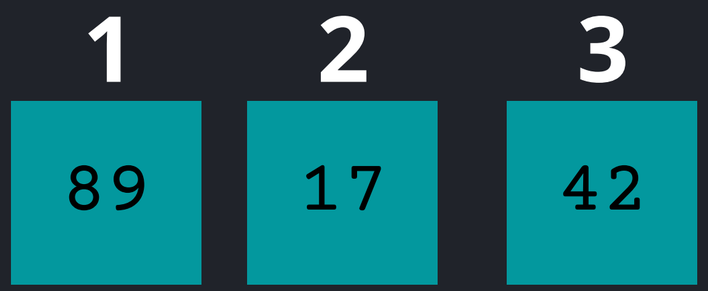

17
42


1
2
42
17

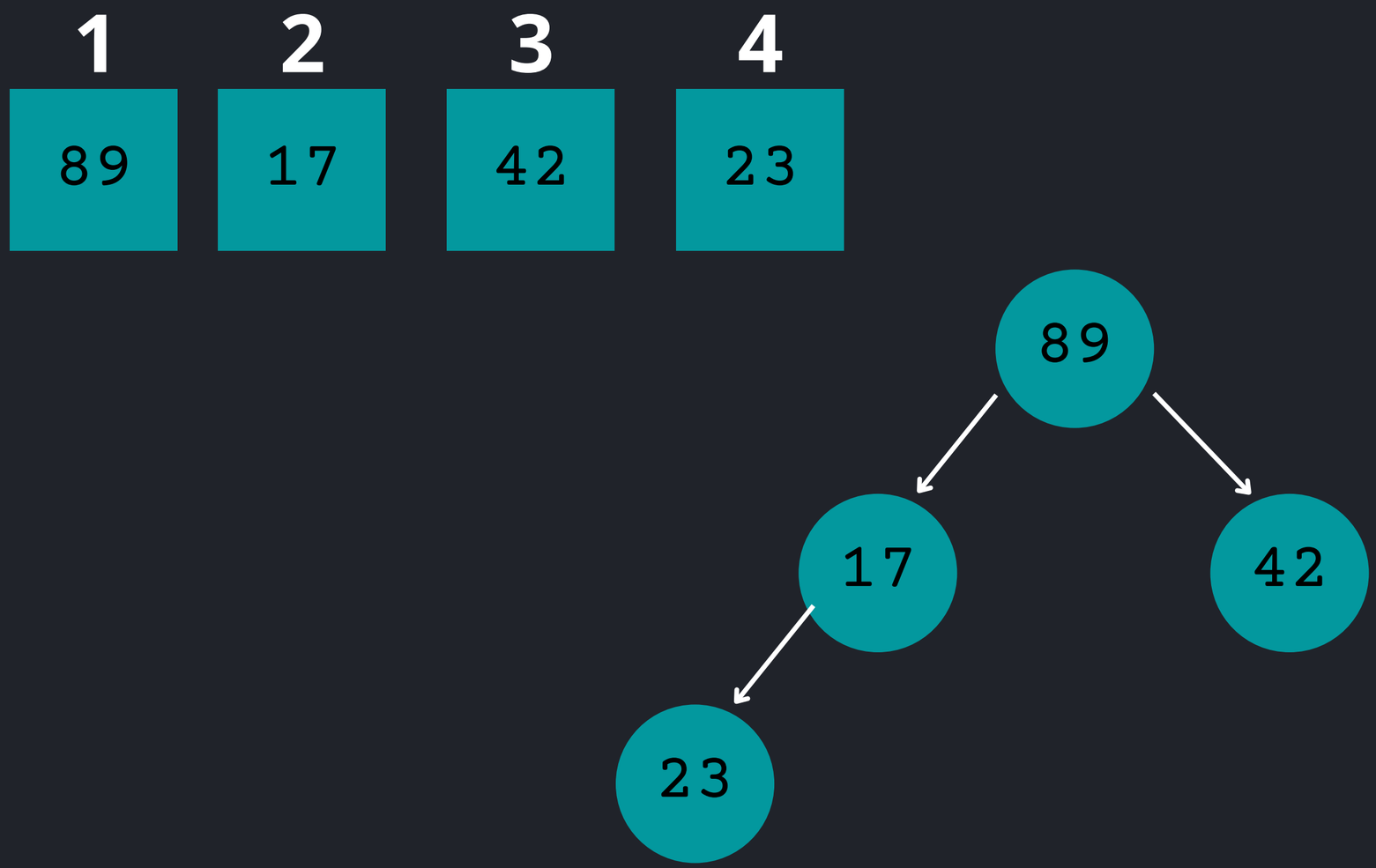

42
17

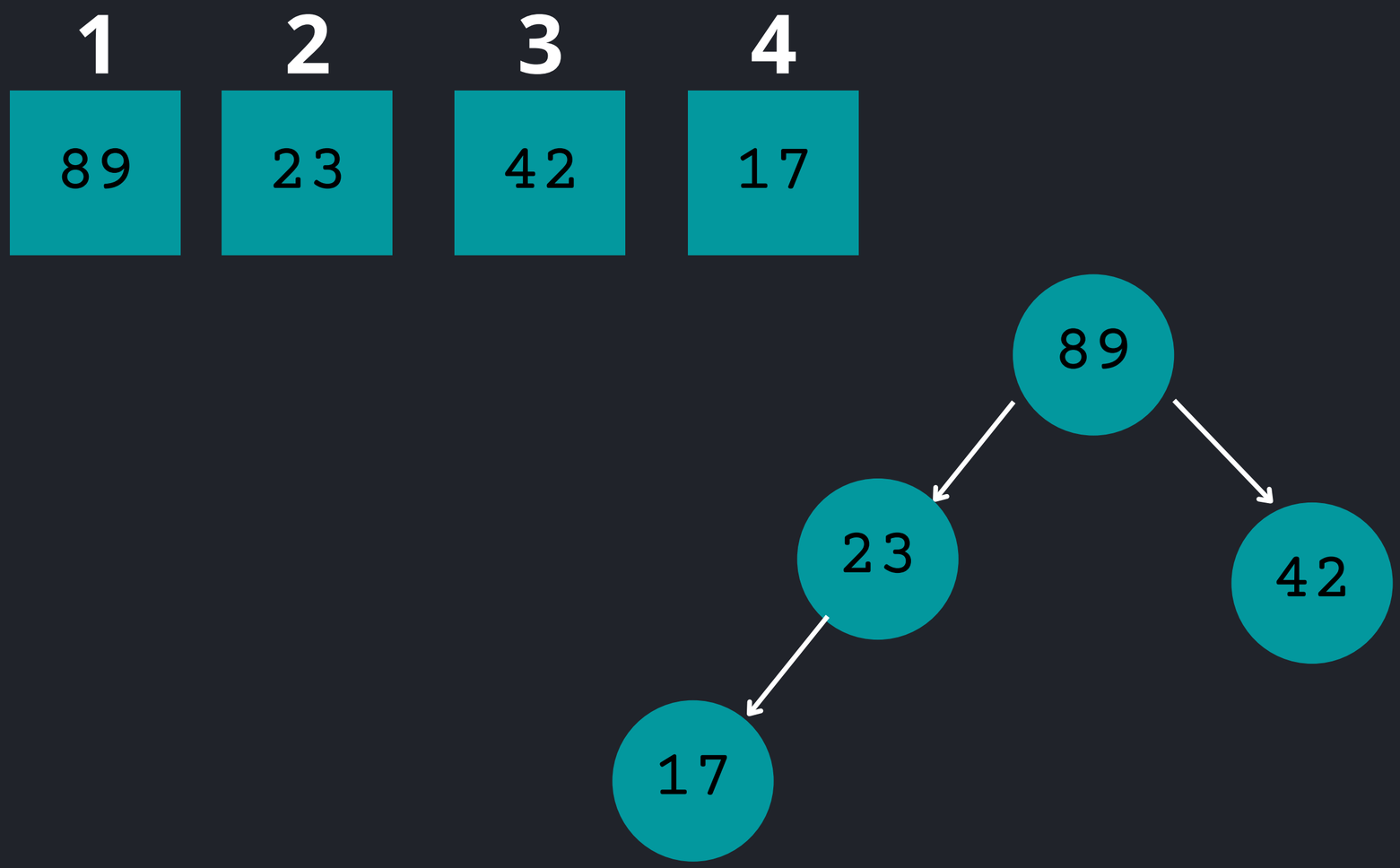

1
2
3
42
17
89


42
17
89

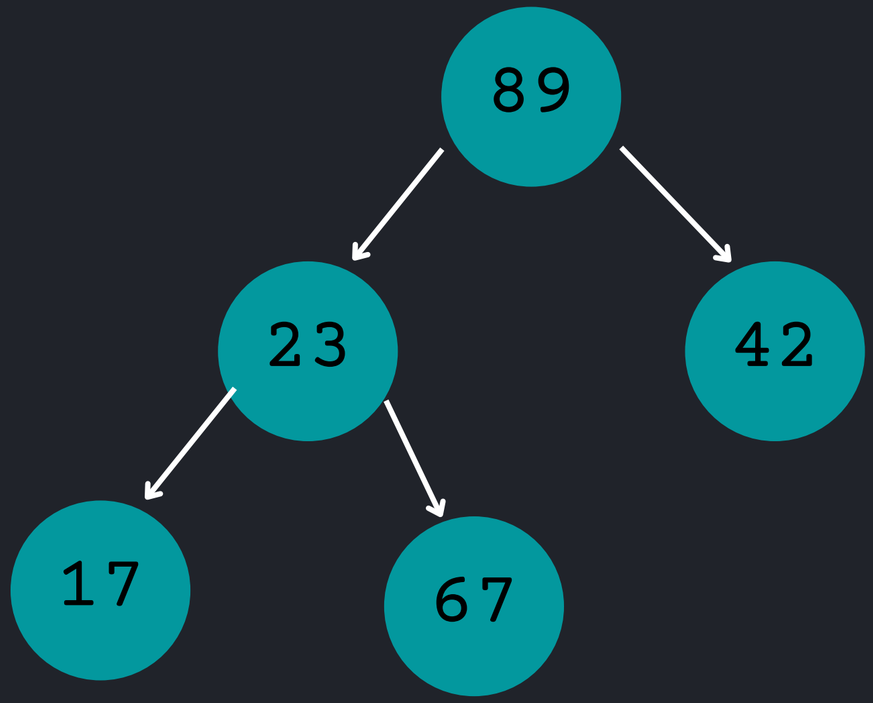

1
2
3
89
17
42


89
17
42

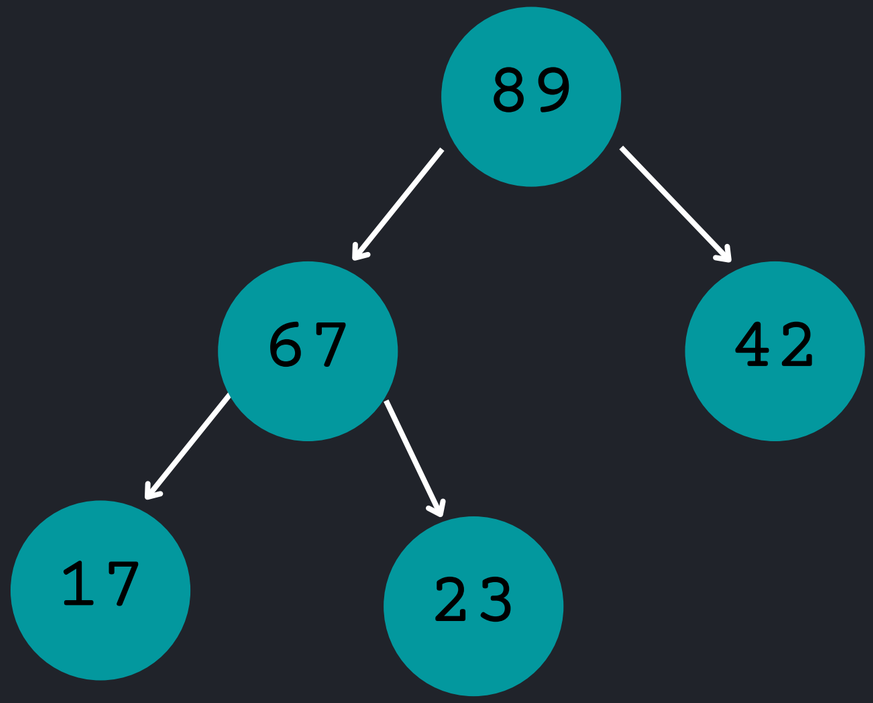

1
2
3
4
89
17
42
23
89
17
42
23


1
2
3
4
89
23
42
17
89
23
42
17

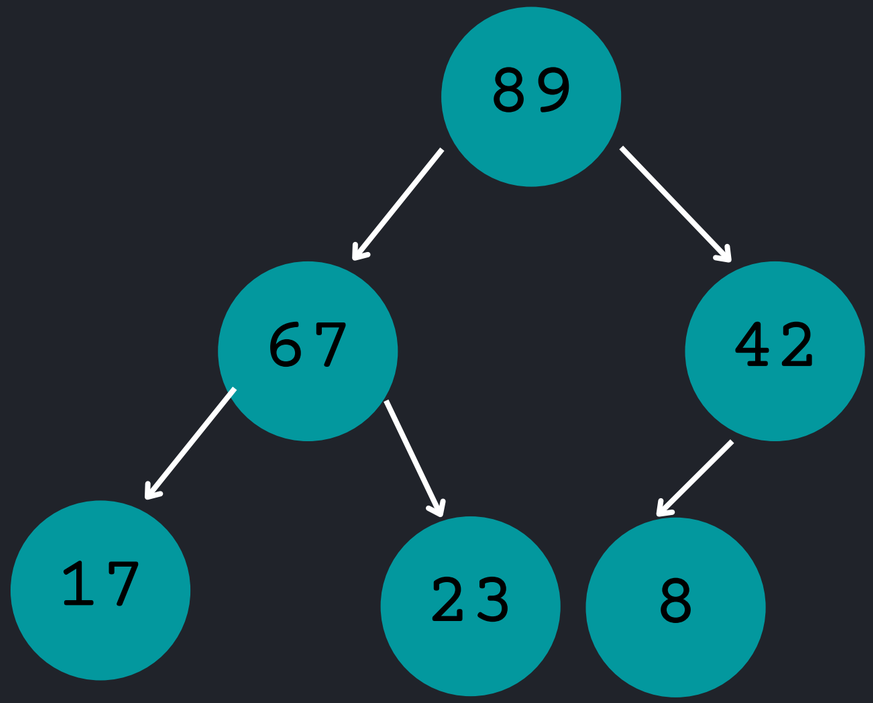
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 89 | 23 | 42 | 17 | 67 |



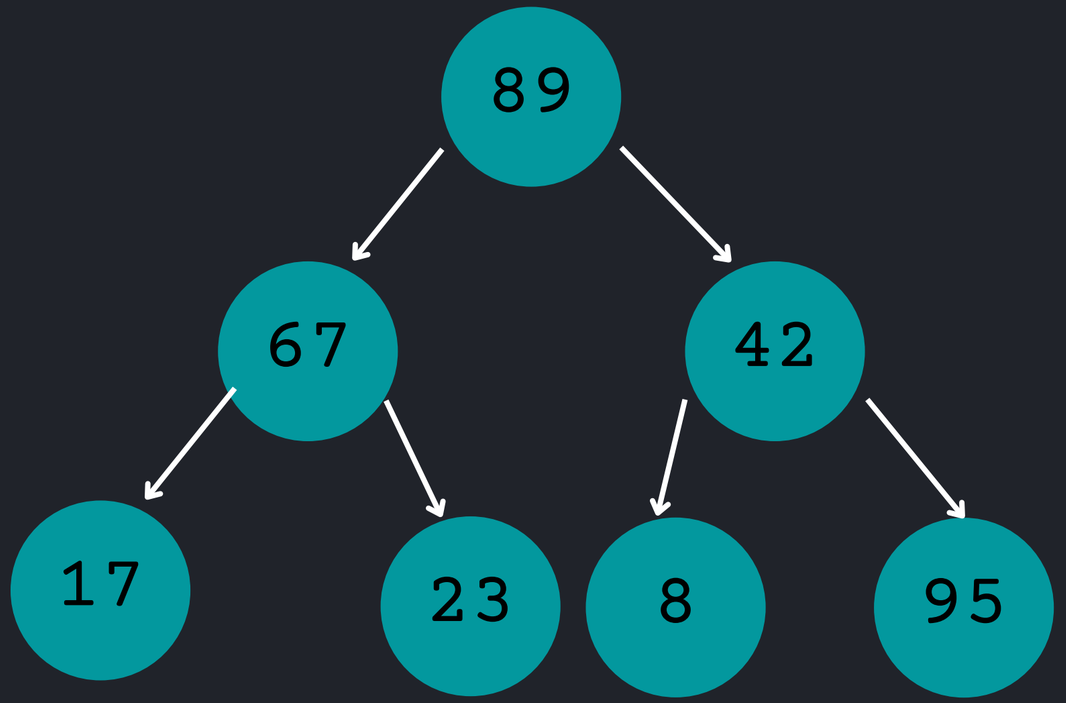
| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 89 | 67 | 42 | 17 | 23 |



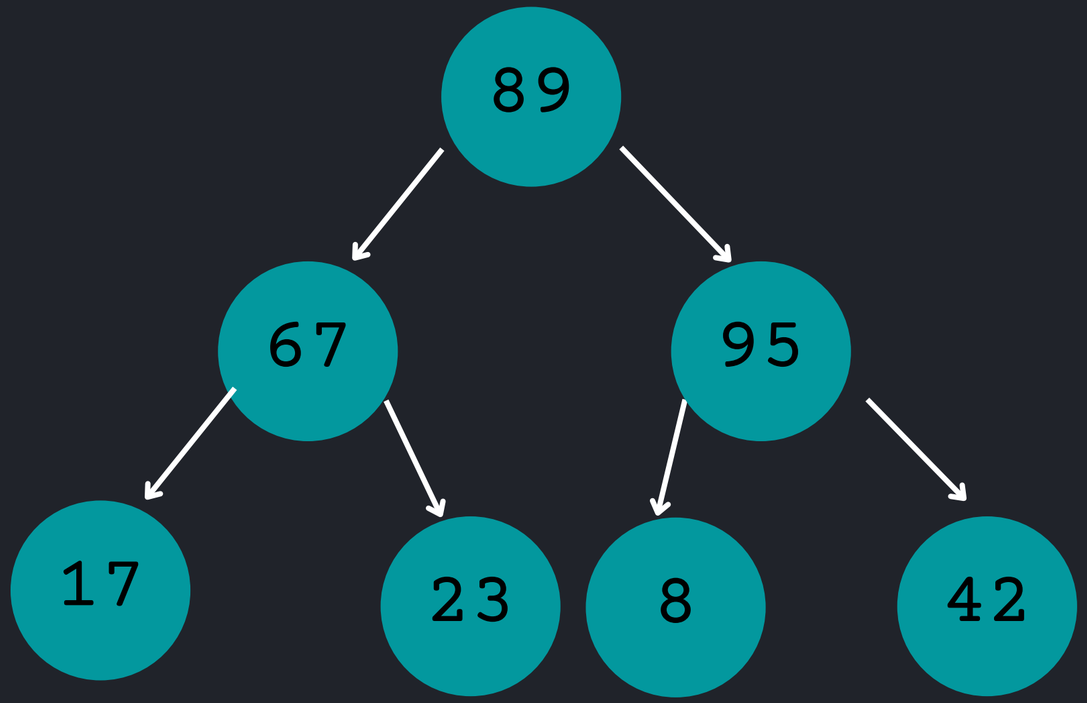
| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 89 | 67 | 42 | 17 | 23 | 8 |



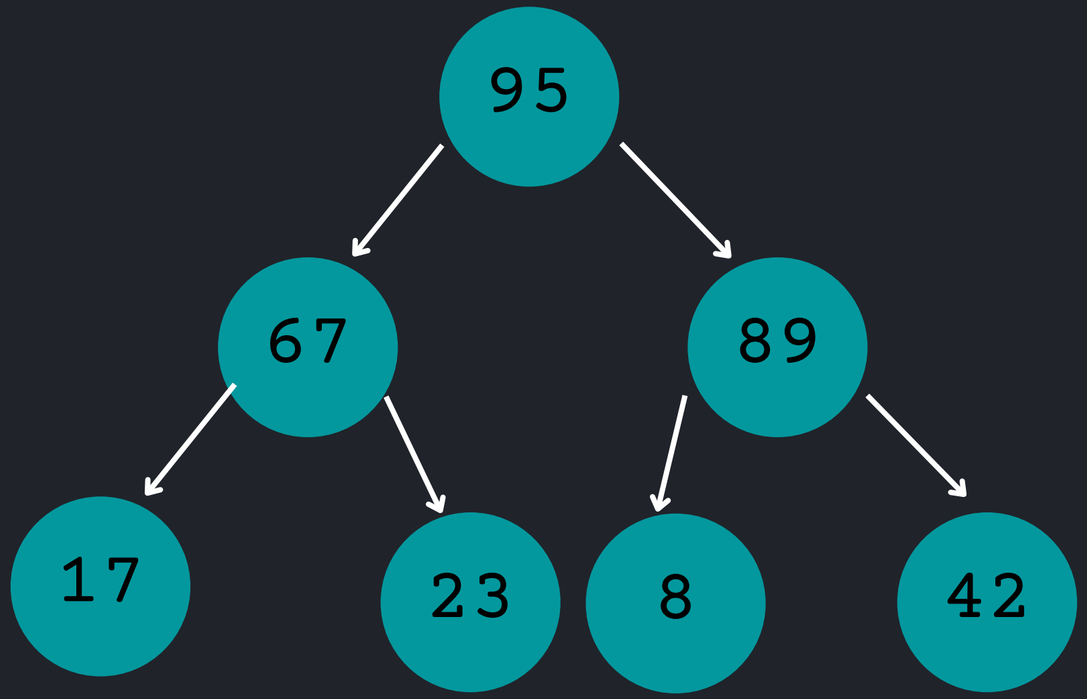
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 89 | 67 | 42 | 17 | 23 | 8 | 95 |



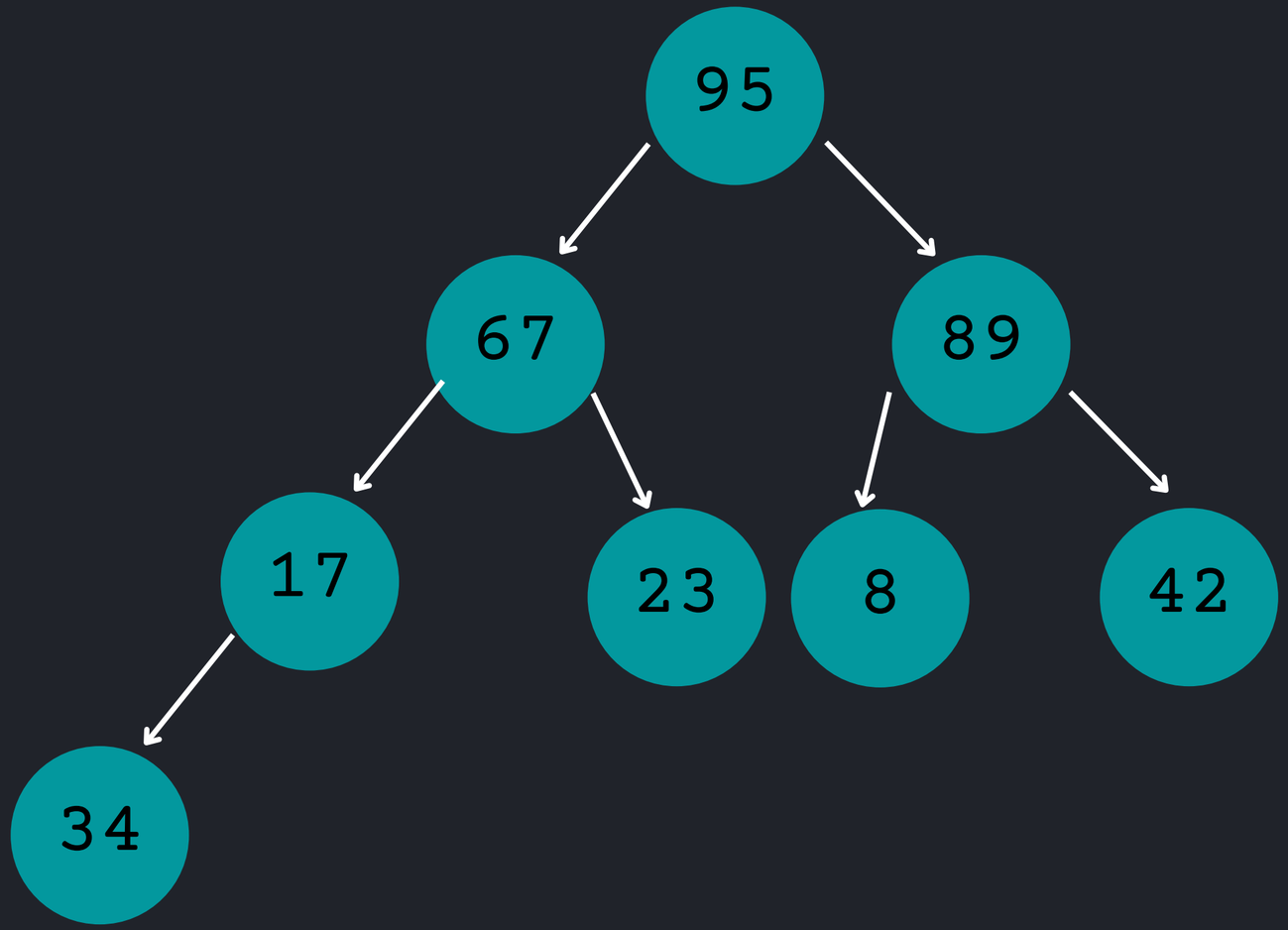
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 89 | 67 | 95 | 17 | 23 | 8 | 42 |



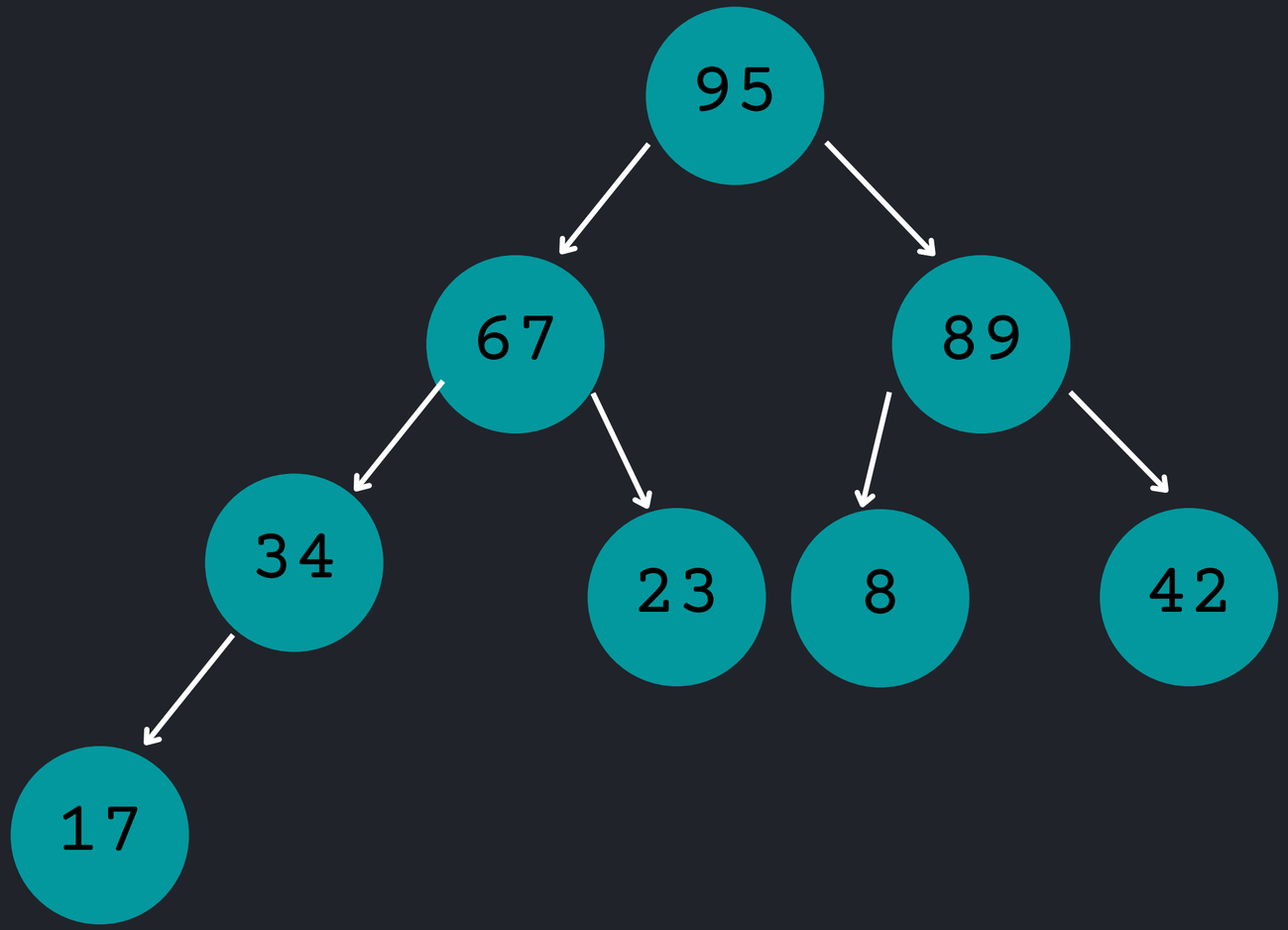
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 95 | 67 | 89 | 17 | 23 | 8 | 42 |



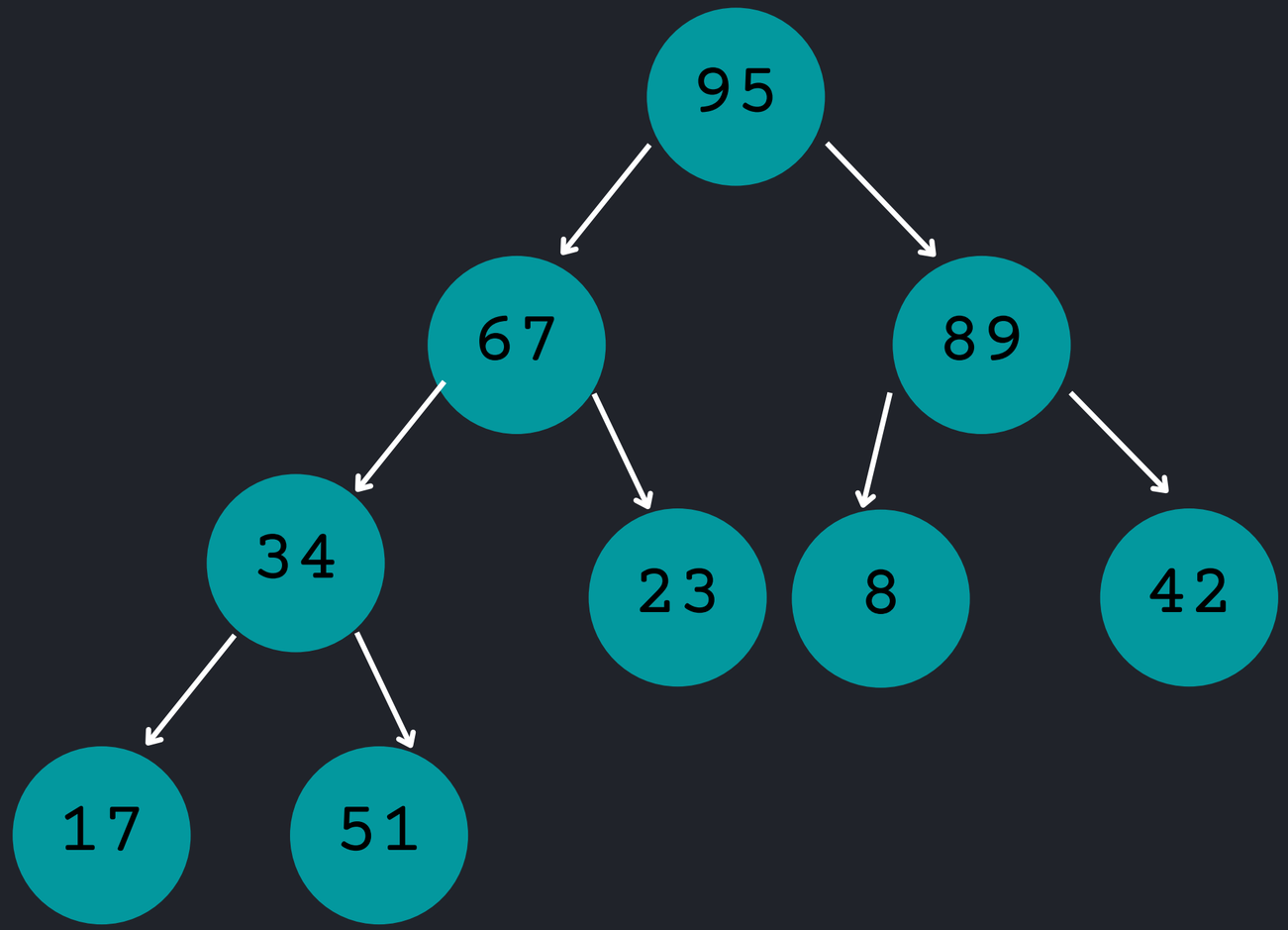
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 95 | 67 | 89 | 17 | 23 | 8 | 42 | 34 |



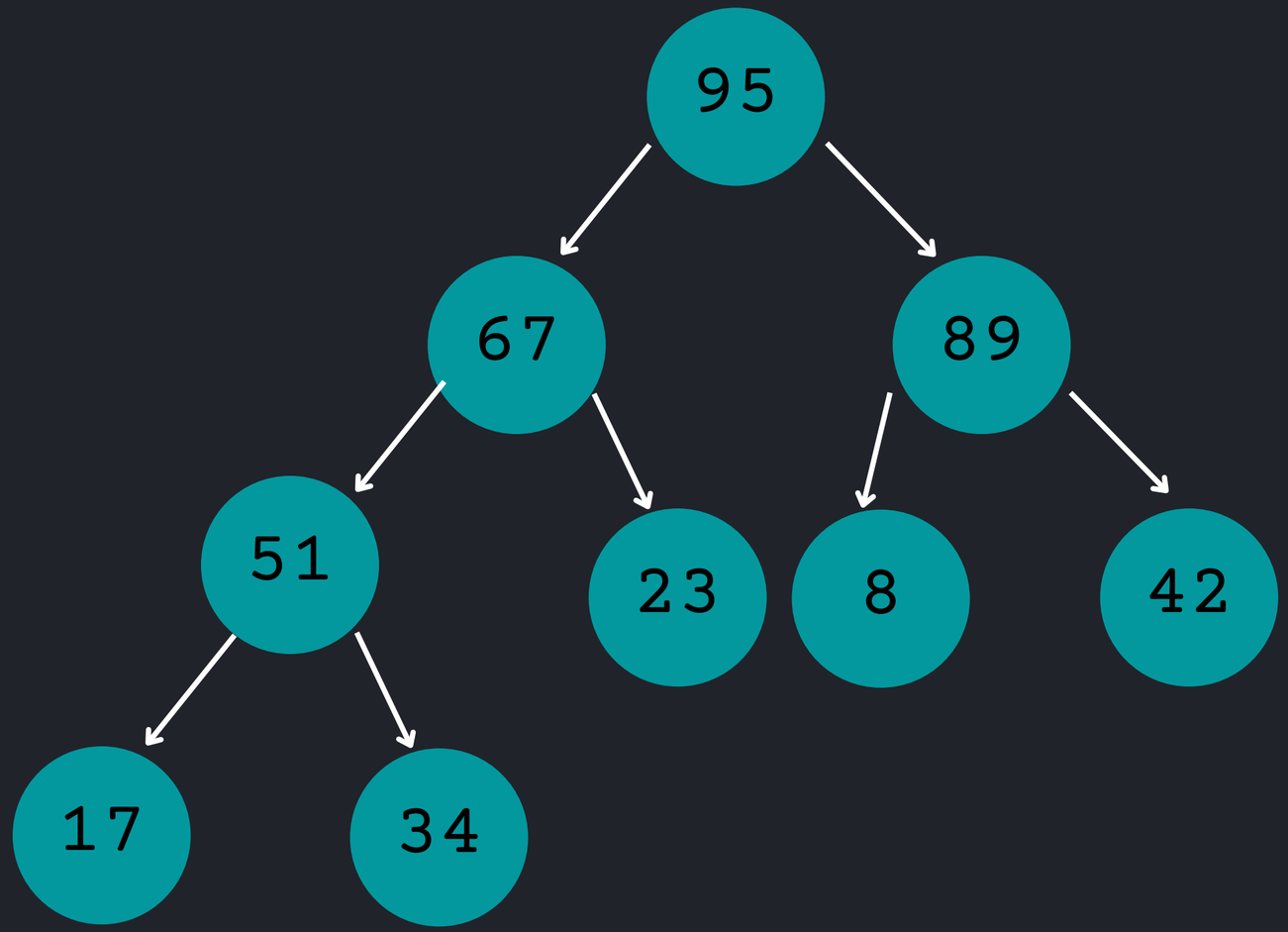
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 95 | 67 | 89 | 34 | 23 | 8 | 42 | 17 |



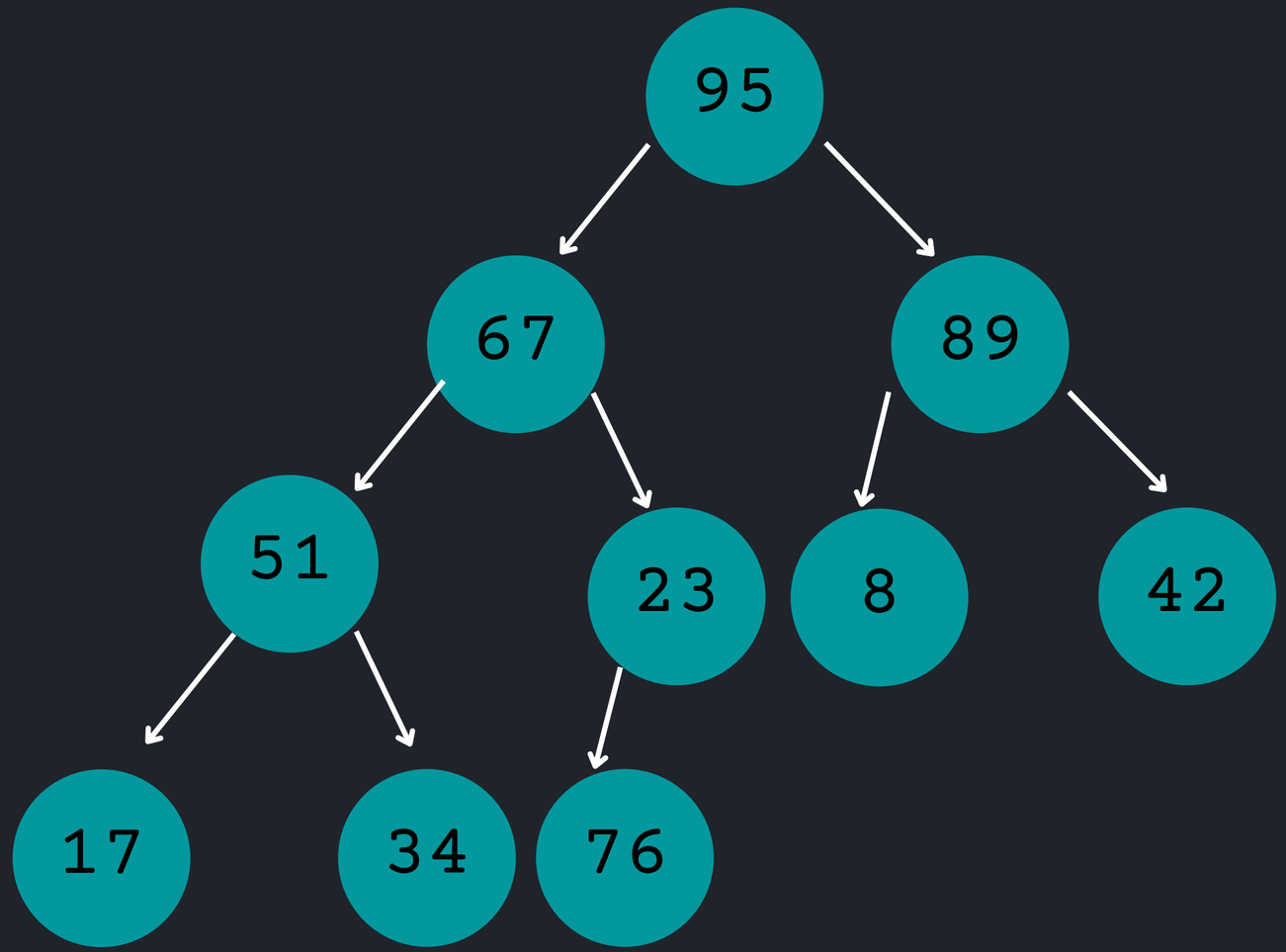
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 95 | 67 | 89 | 34 | 23 | 8 | 42 | 17 | 51 |



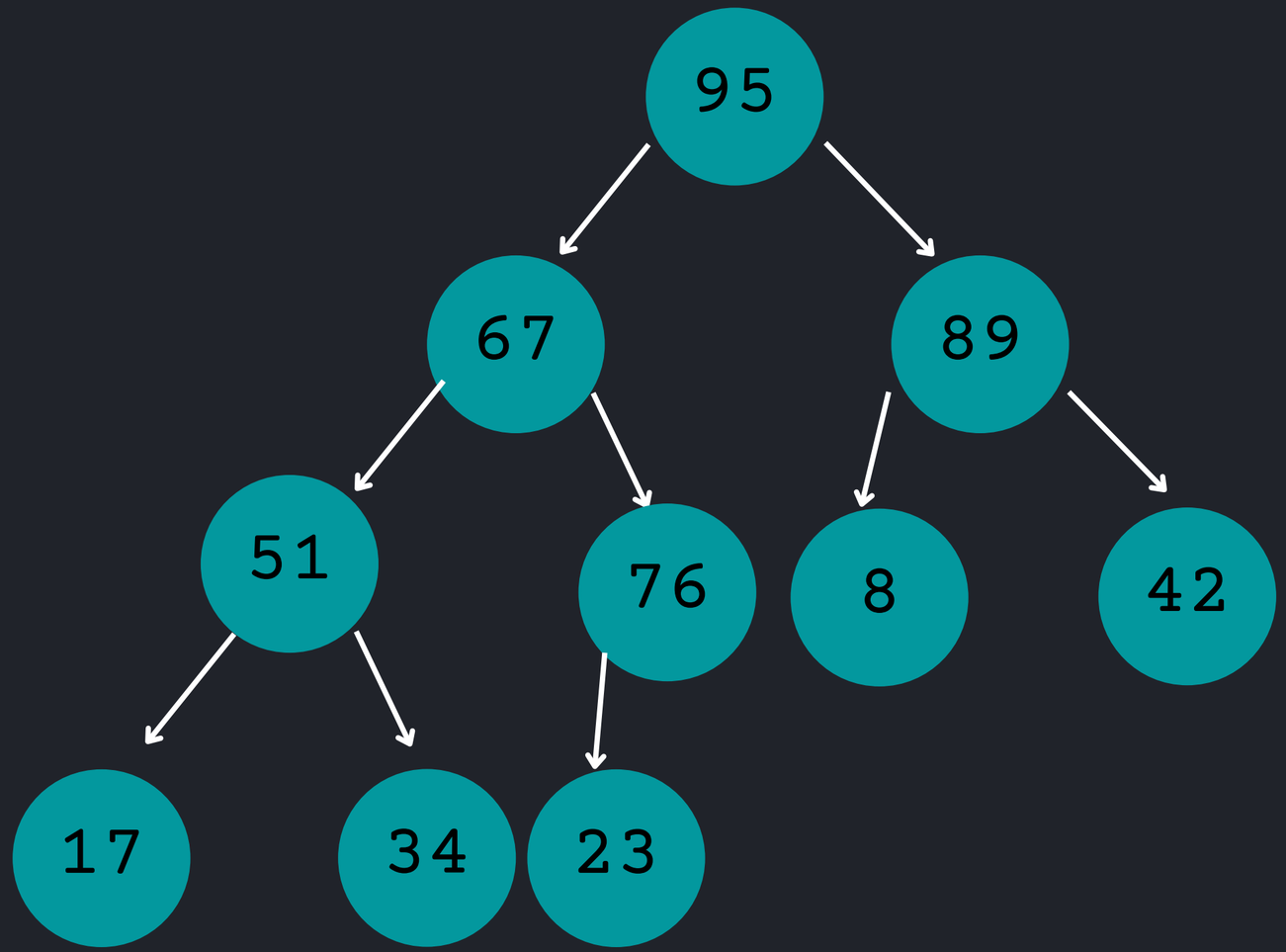
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 95 | 67 | 89 | 51 | 23 | 8 | 42 | 17 | 34 |



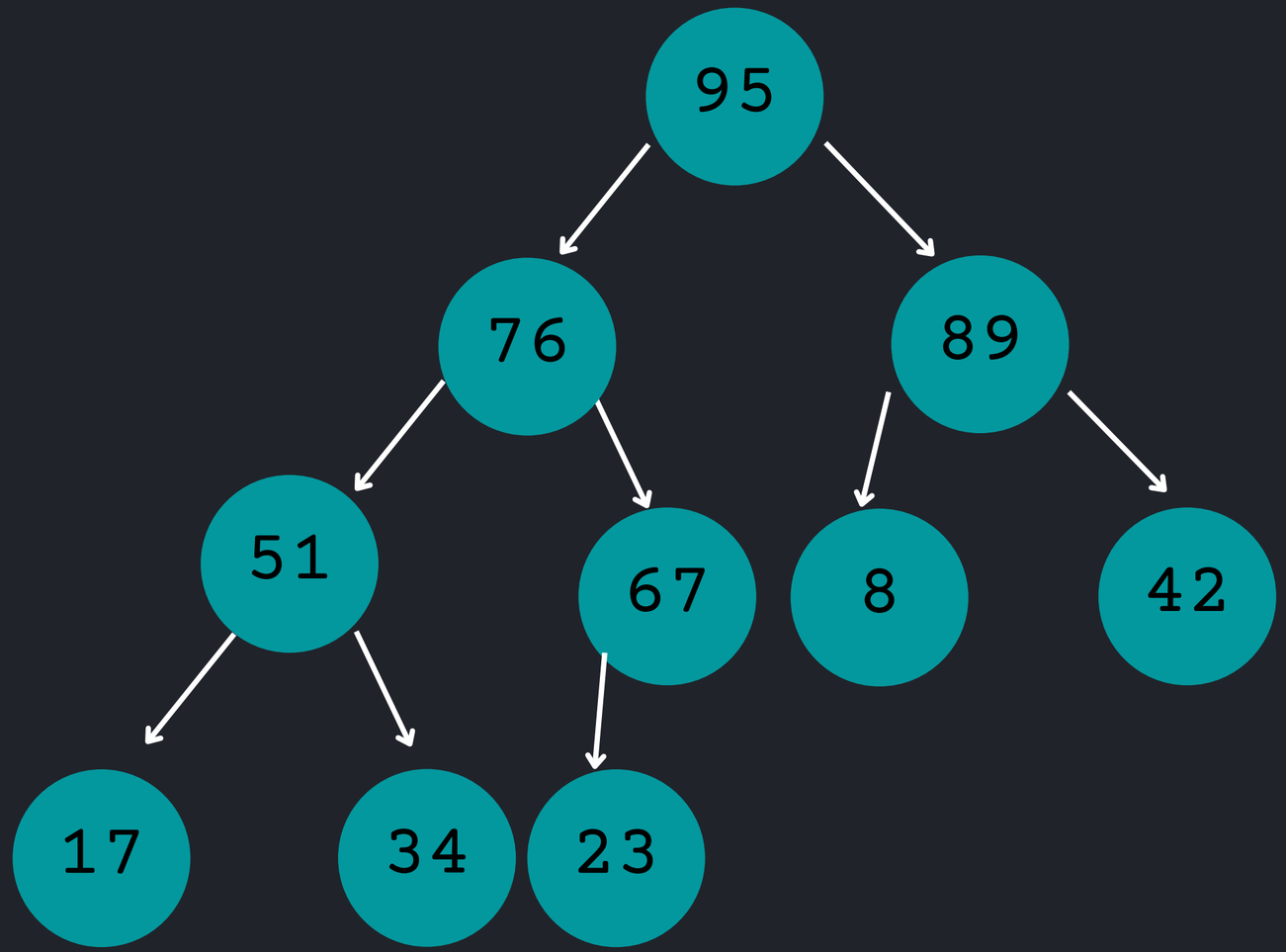
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 95 | 67 | 89 | 51 | 23 | 8 | 42 | 17 | 34 | 76 |



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 95 | 67 | 89 | 51 | 76 | 8 | 42 | 17 | 34 | 23 |



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 95 | 76 | 89 | 51 | 67 | 8 | 42 | 17 | 34 | 23 |



<!---->

# Extracción{

<Ejemplo"max-heap"/>

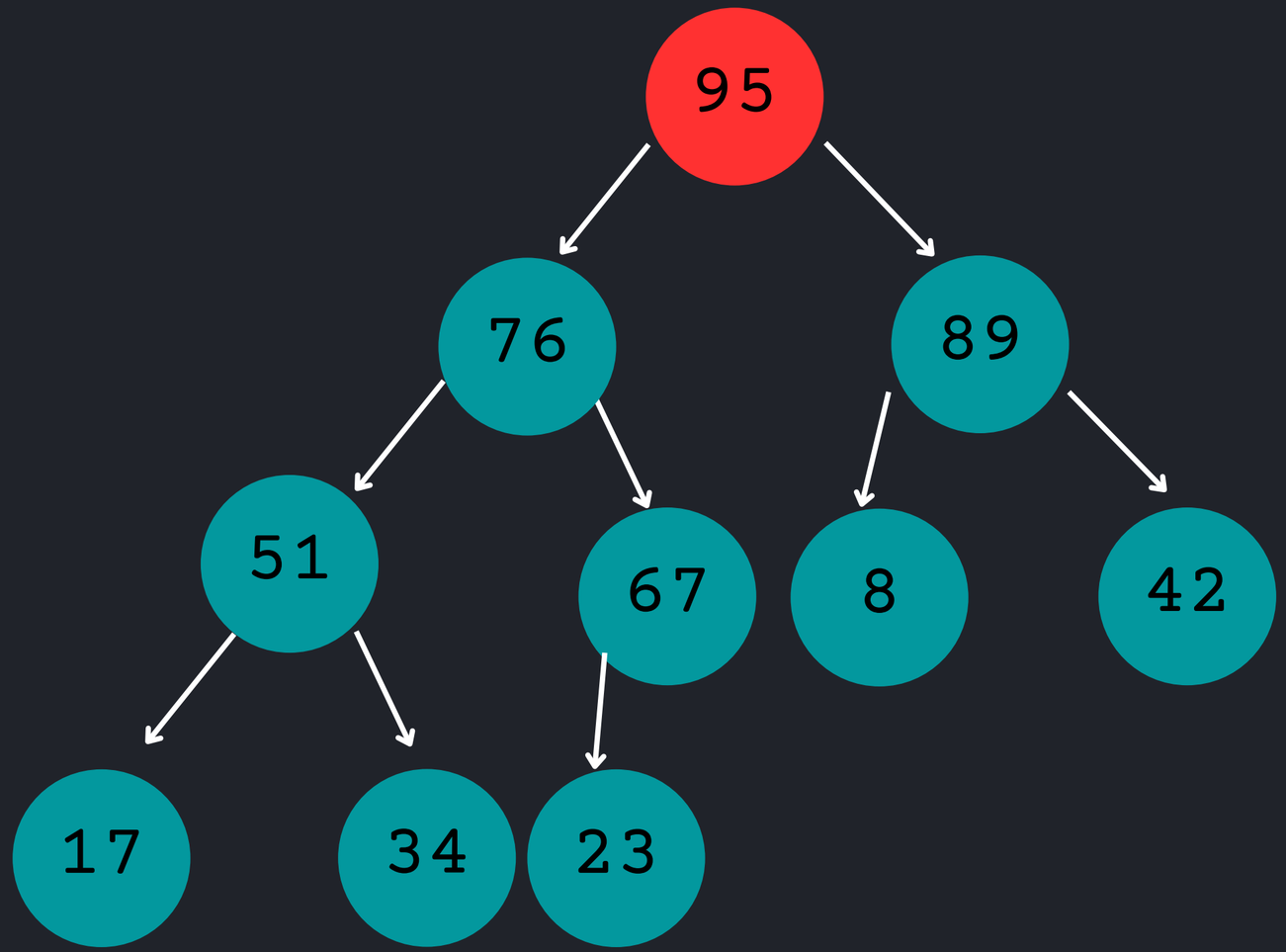
}

1.Extraemos siempre el elemento de mayor prioridad de nuestro Heap X[1].
2.Como debemos de dejar el Heap ordenado, ponemos el ultimo elemento del Heap, u = X[n], en la primera posición: X[1] = u, luego se decrementa n en uno y “hundimos” u intercambiándolo con el hijo de mayor prioridad hasta que quede bien ubicado.
La operación que “hunde” X[1] hasta dejarlo bien ubicado se llama heapify.

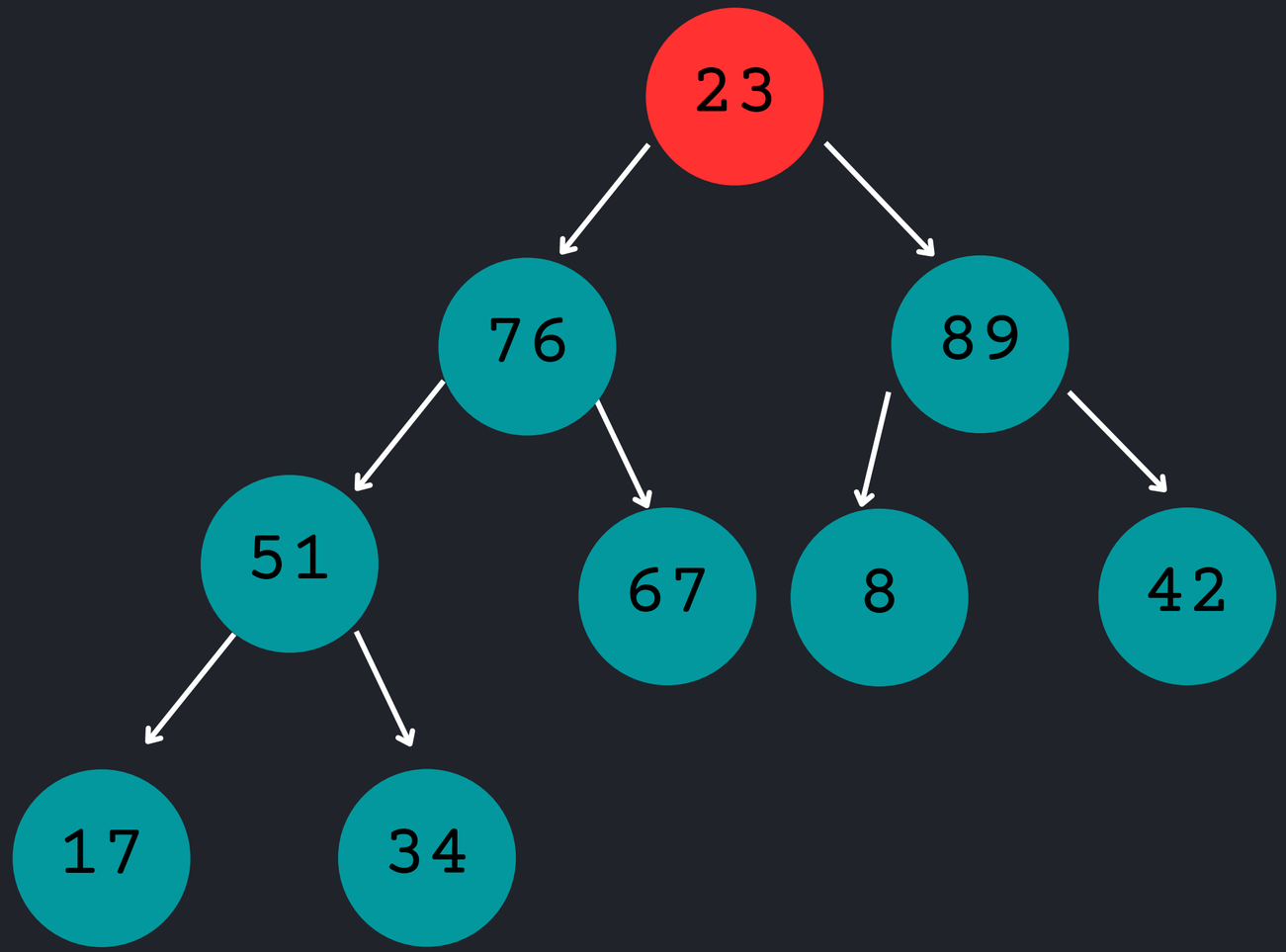
La extracción se hace en tiempo constante y siempre se extrae el elemento de mejor prioridad. Luego de extraer hay que dejar la estructura ordenada, lo cual toma nuevamente tiempo logarítmico, esto se denota como O(log n) y se lee: “el tiempo es orden log n”.

}

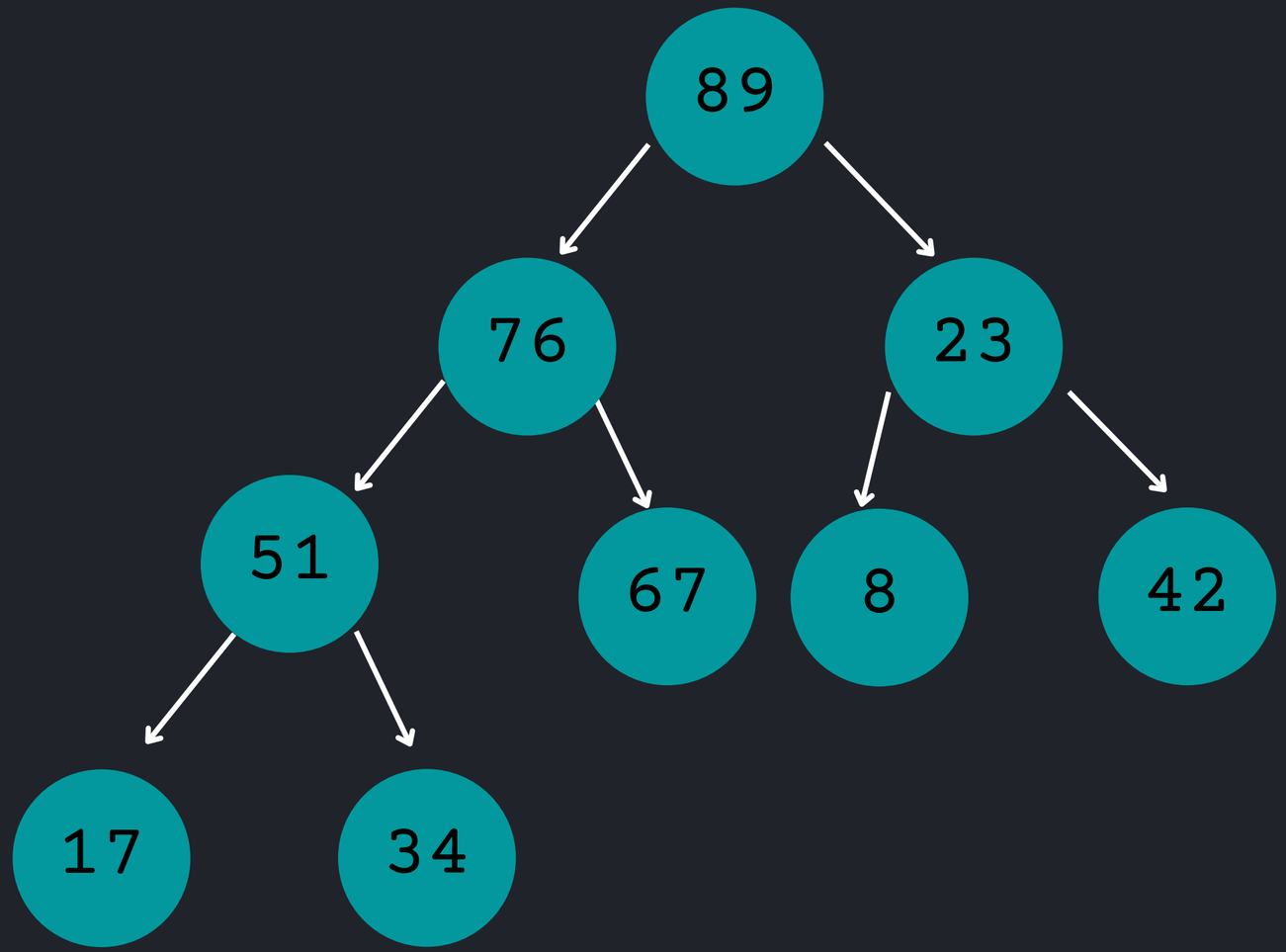
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 95 | 76 | 89 | 51 | 67 | 8 | 42 | 17 | 34 | 23 |



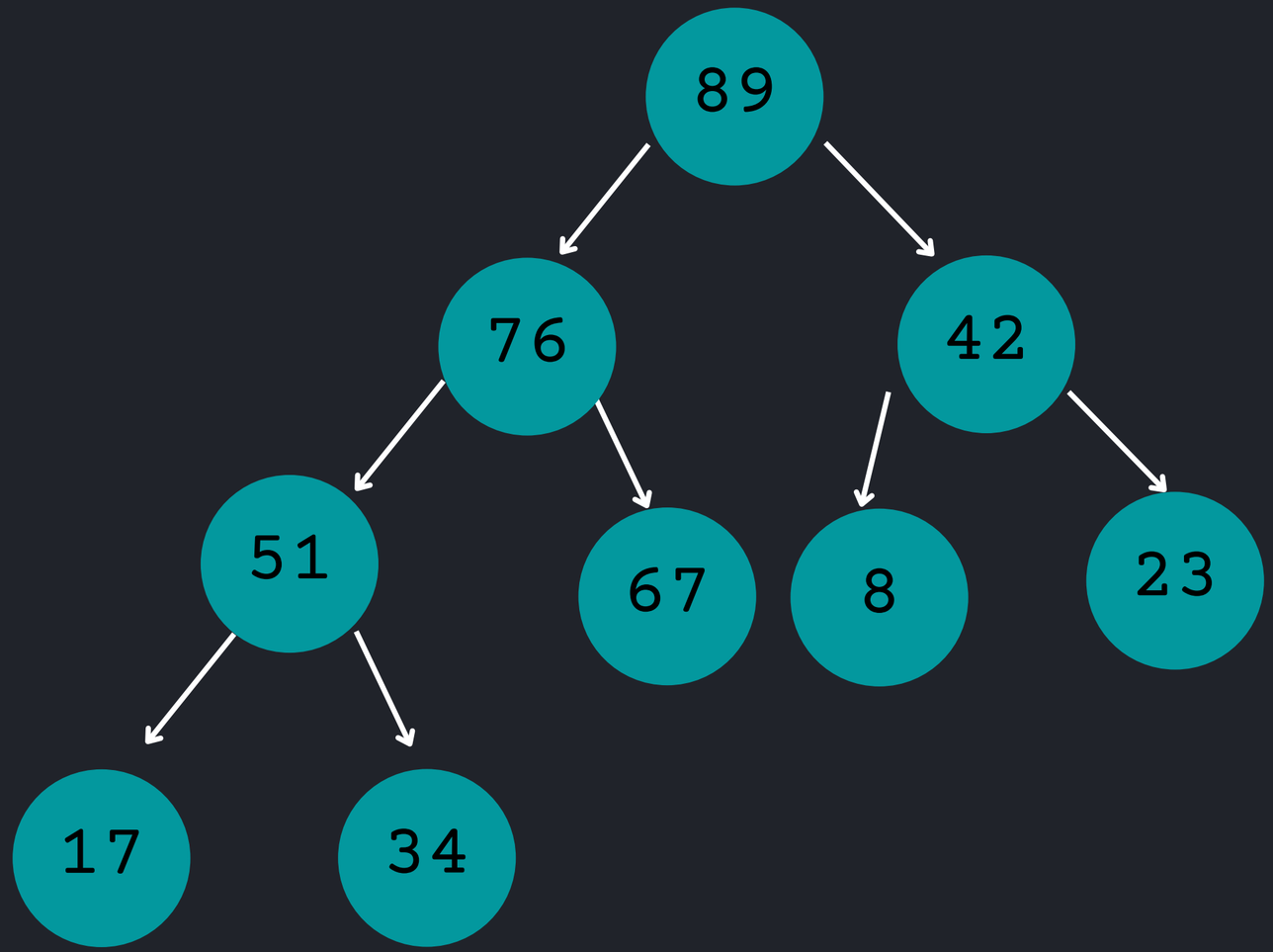
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 23 | 76 | 89 | 51 | 67 | 8 | 42 | 17 | 34 |



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 89 | 76 | 23 | 51 | 67 | 8 | 42 | 17 | 34 |



| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| 89 | 76 | 42 | 51 | 67 | 8 | 23 | 17 | 34 |



# AVL Teoría{

## ¿Qué es un árbol AVL?

Es una estructura de datos de tipo árbol binario auto-equilibrado, lo cual significa que siempre que se realice un cambio en los datos, este hará comparaciones entre nodos para nivelarse y que sus nodos no tengan mas de 1 nivel de diferencia.

## Implementaciones de árboles AVL

Actualmente los árboles AVL son de gran utilidad y son muy versátiles de implementar en:

- Búsquedas: Estas se facilitan gracias a la auto-nivelación de árbol, volviendo las búsquedas mas eficientes.
- Ordenamientos: Estos también pueden utilizarse para ordenar datos, ya que los nodos se colocan según su valor.

## ¿Como funciona un AVL?

Un valor ingresado por medio de la raíz (en caso de que exista), y por medio de comparaciones verifica si el nodo a insertar es de valor mayor o menor al actual, hasta llegar a su posición correcta. Las rotaciones son otra parte importante del auto-balanceo del árbol AVL, ya que estas son las que mantienen nivelado el árbol.

}

# AVL Rotaciones{

## ¿Qué es una rotación en un árbol AVL?

Una rotación es una modificación sencilla sobre la estructura de un árbol binario de busqueda, que permite mantener la propiedad de orden

## Rotaciones en un árbol AVL

Son cuatro correspondientes a las cuatro combinaciones para la inserción de una clave a partir de un nodo raíz del subárbol considerado:

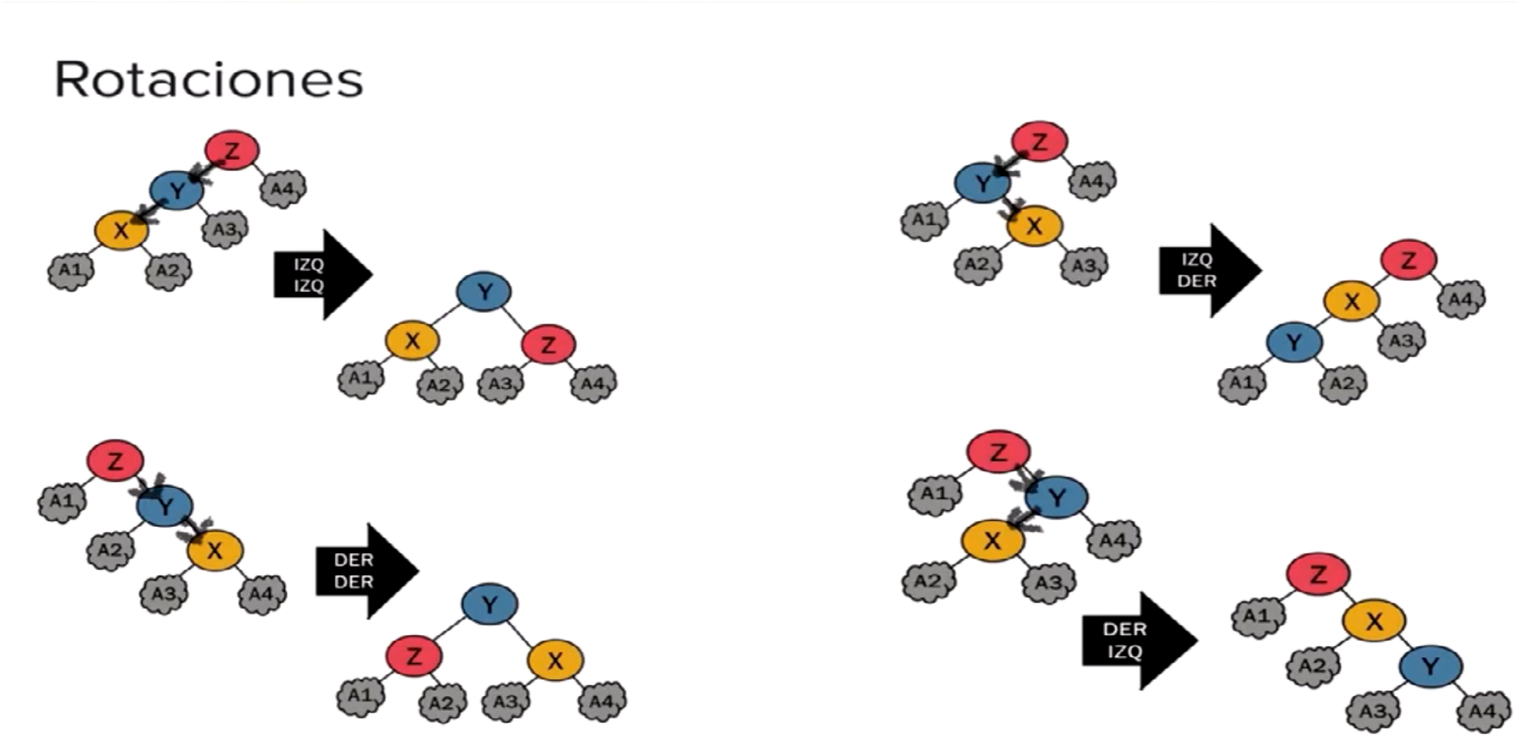
- Izquierda-izquierda: rotación simple de izquierda a derecha
- izquierda - derecha: Rotación doble de izquierda a derecha
- derecha - derecha: Rotación simple de derecha a izquierda
- derecha - izquierda: Rotación doble de derecha a izquierda

## ¿Para qué sirven las rotaciones?

- Se mantiene el arbol ordenado
- En los algoritmos de inserción y eliminación es posible buscar, insertar y eliminar un elemento en un árbol balanceado

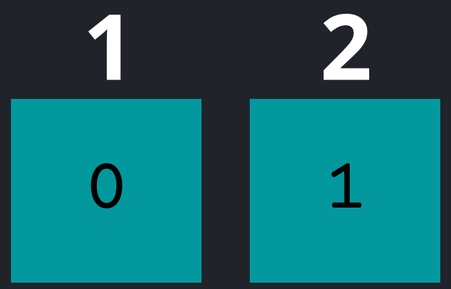
}

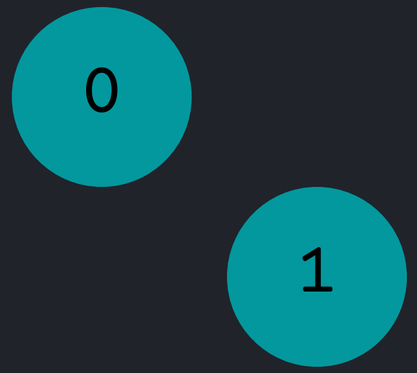
# EjemplificaciónAVL Rotaciones{




1
0


0


1
2
0
1

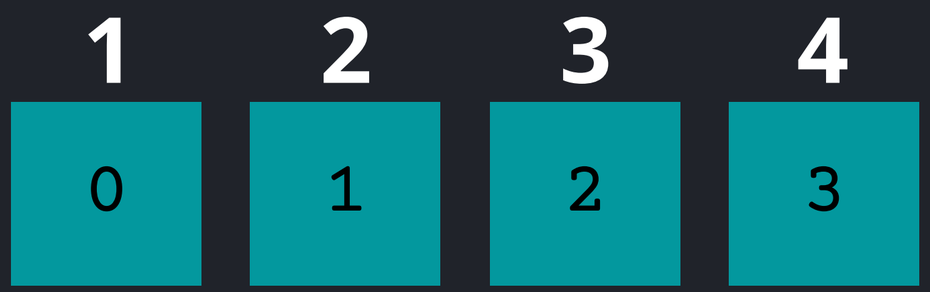

0
1


1
2
3
0
1
2

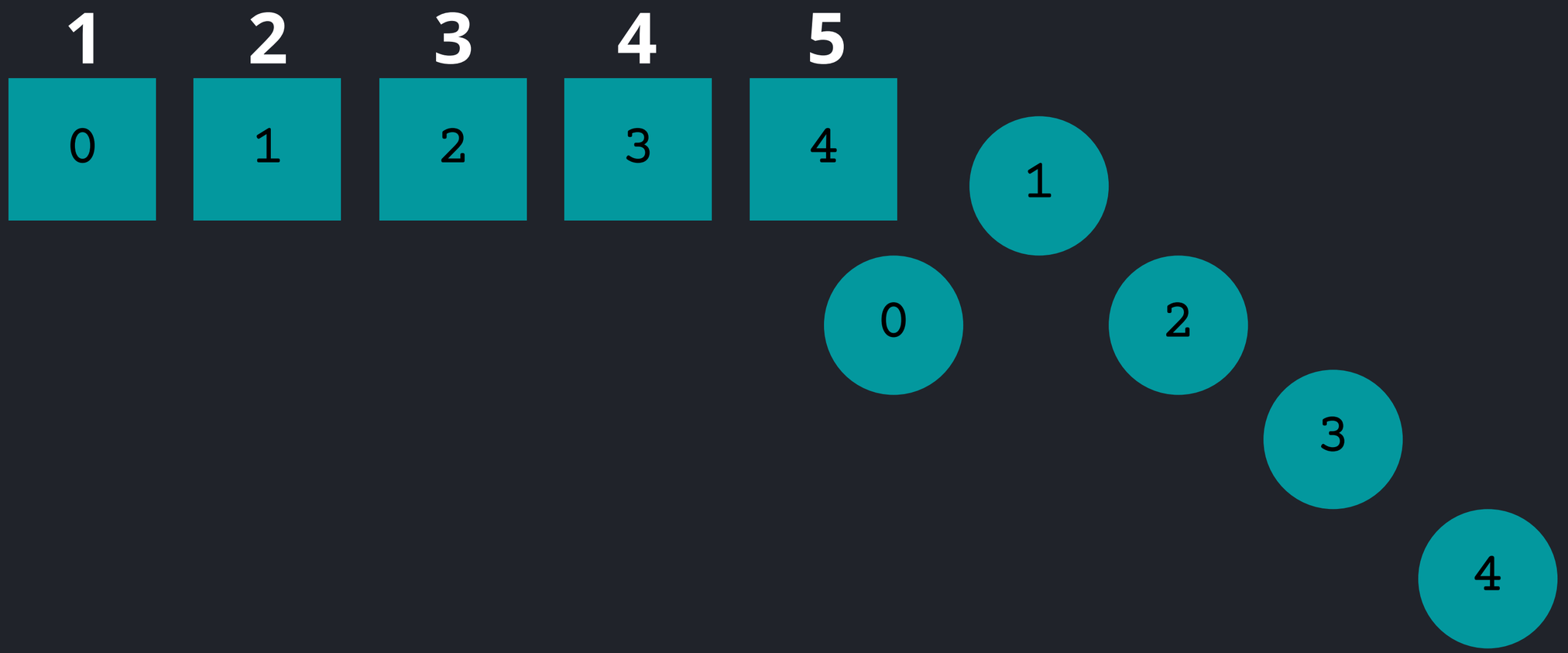

0
1
2

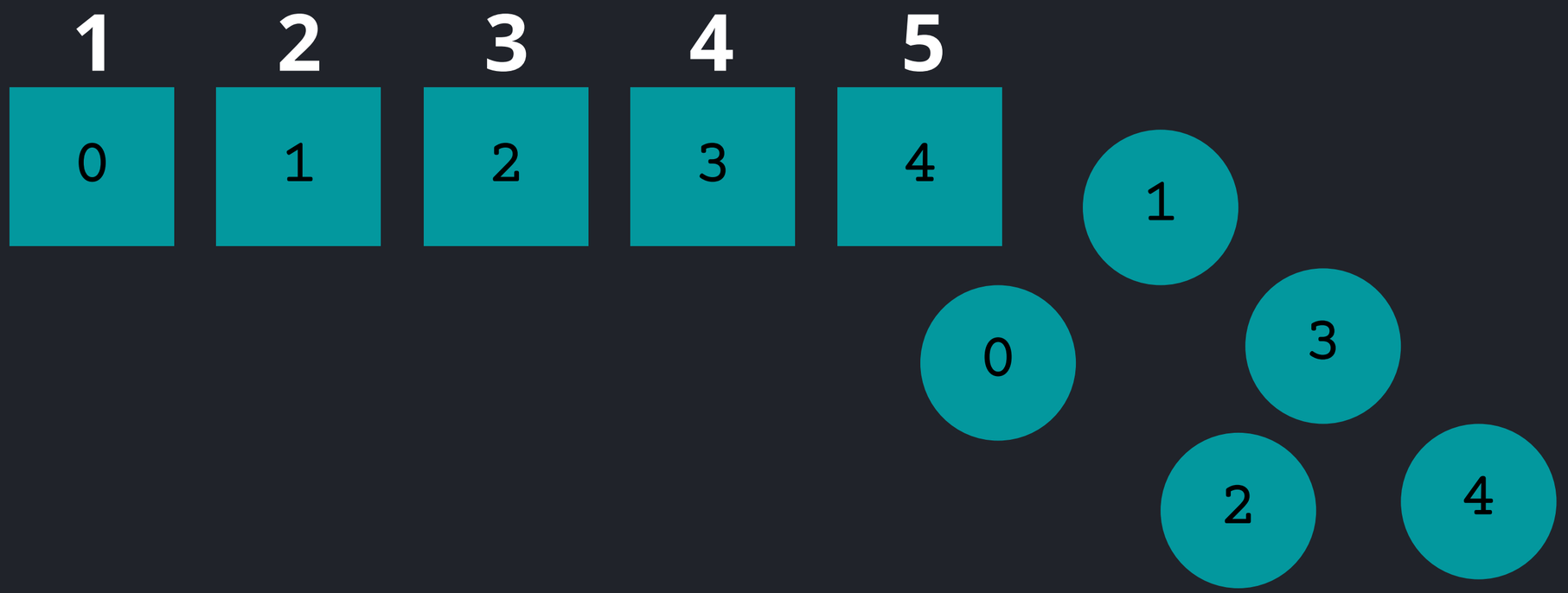

1
2
3
0
1
2

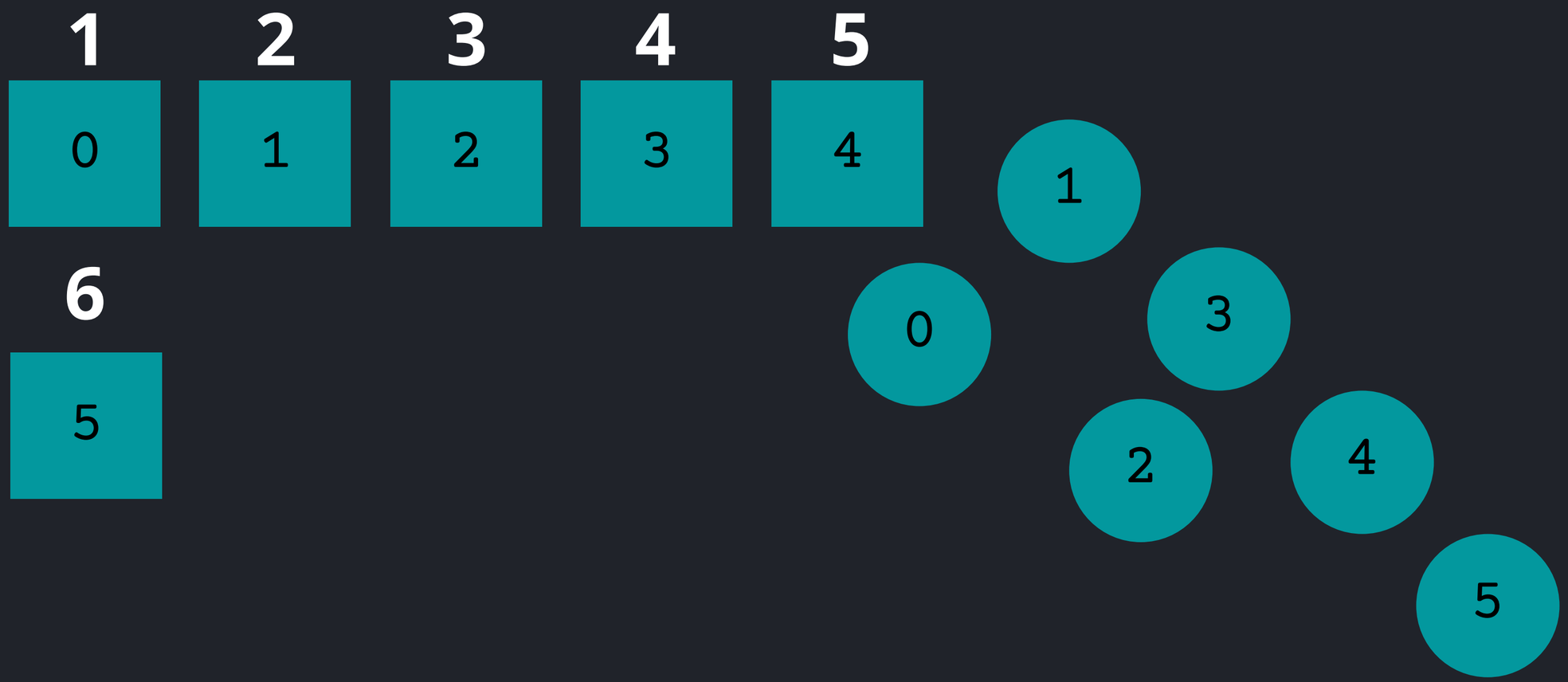

1
0
2

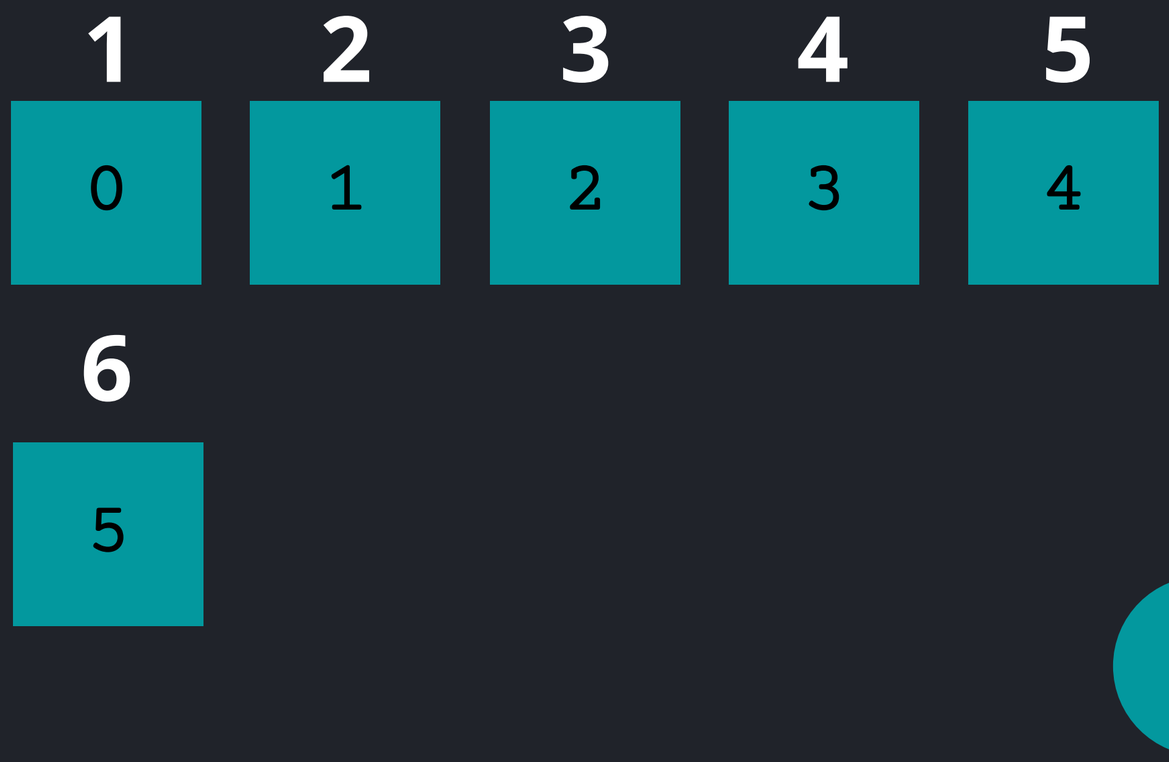

1
2
3
4
0
1
2
3


1
0
2
3


1
2
3
4
5
0
1
2
3
4
1
0
2
3
4


1
2
3
4
5
0
1
2
3
4
1
0
3
2
4


1
2
3
4
5
0
1
2
3
4
6
5
1
0
3
2
4
5


1
2
3
4
5
0
1
2
3
4
6
5
3
1
4
0
2
5

# AVL Eliminación{

## ¿Cómo funciona la eliminación en AVL?

Primero que nada, se comienza buscando el nodo con el valor que se desea eliminar, una vez ubicado se elimina y dependiendo el caso en el que se encuentre, se realizan rotaciones o no.

## Casos de eliminación AVL

1. Eliminación de hojas: Este es el caso mas sencillo, ya que únicamente se elimina el nodo y ya que se trata de una hoja, esta no apunta a ningún otro nodo.
2. Eliminación de raíz con un hijo: Cuando se elimina una raíz que posee un solo hijo únicamente su hijo precedente toma su lugar, posteriormente se realizan comparaciones para ver si se encuentra nivelado.
3. Eliminación de raíz con dos hijos: Este es el caso mas complicado, ya que se debe seleccionar un nodo que tome el lugar del nodo eliminado, pudiendo ser este el nodo mayor de su hijo de lado izquierdo, o el nodo menor de su hijo de lado derecho.

}

# Eliminación
# Caso 1

# Eliminación de hojas

Valores en árbol

```
10, 6, 15, 20, 0, 7
```

Valores a eliminar

```
20, 0
```

# Eliminación de hojas

Valores en árbol

```
10, 6, 15, 20, 0, 7
```

Valores eliminados

```
20, 0
```

# Eliminación
# Caso 2

# Eliminación padre 1 hijo

Valores en árbol

10, 6, 15, 20, 0, 7

Valores a eliminar

15

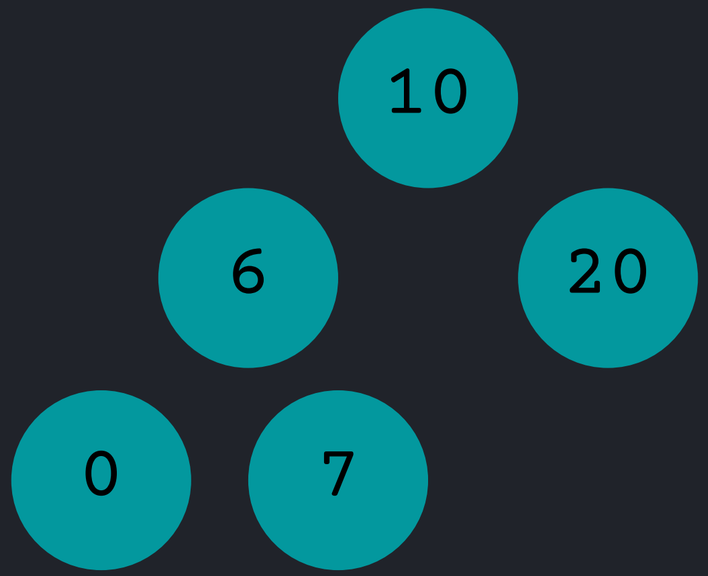
# Eliminación padre 1 hijo

Valores en árbol

10, 6, 15, 20, 0, 7

Valores eliminados

15



# Eliminación
# Caso 3

# Eliminación padre 2 hijos

Valores en árbol

```
10, 6, 15, 20, 0, 7
```

Valores a eliminar

```
6
```

# Eliminación padre 2 hijos

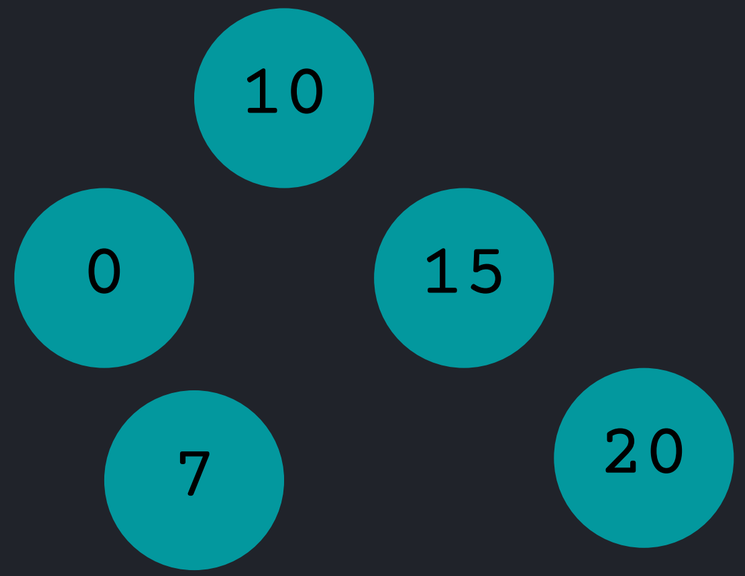
Valores en árbol

10, 6, 15, 20, 0, 7

Valores eliminados

6

Caso 1



# Eliminación padre 2 hijos

Valores en árbol

10, 6, 15, 20, 0, 7

Valores eliminados

6

Caso 2

# Eliminación raíz

Valores en árbol

```
10, 6, 15, 20, 0, 7
```

Valores a eliminar

```
10
```

# Eliminación raíz

Caso 1

Valores en árbol

10, 6, 15, 20, 0, 7

Valores eliminados

10

# Eliminación raíz

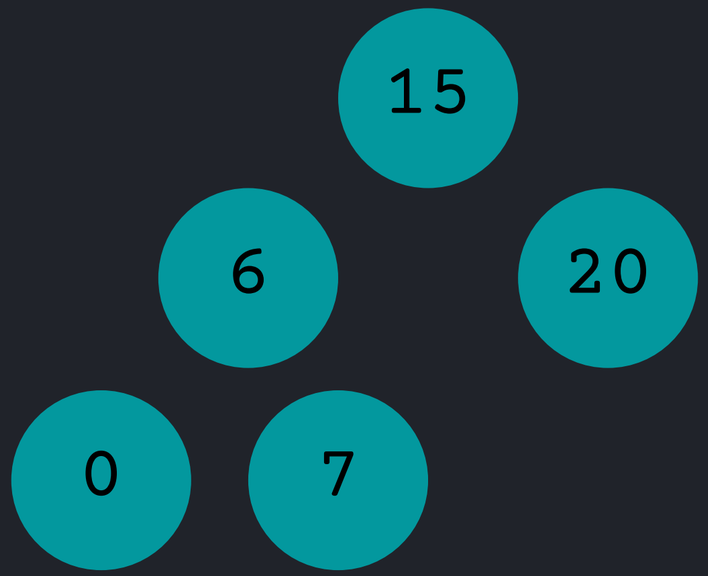
Caso 2

Valores en árbol

```
10, 6, 15, 20, 0, 7
```

Valores a eliminar

```
10
```



<!--Parcial II ED1->

# Gracias {

<Por="El Grupo"/>

}