Estructuras de Datos y Algoritmos **18-06-2025**

Pauta Mesa de Estudio 2

Autor: *Diego Banda* (diego.banda@mail.udp.cl)

Esta es una guía de estudio para la Solemne 2-2025 de *Estructura de Datos y Algoritmos*, con los siguientes temas: *Binary Search*, Ordenamiento, Tablas de Hash, *Sets*, *Maps*, *Priority Queue* y *Binary Search (BST)*.

1. Binary Search

1.1. Conceptos

Es un tipo de búsqueda muy eficiente ya que trabaja bajo el concepto de *divide and conquer*, divide el arreglo en cada iteración y avanza con la búsqueda solo con la mitad del arreglo en el cual existen expectativas de encontrar lo buscado.

Funcionamiento: Inicia con *low* igual a 0 y *high* igual a la longitud del arreglo-1, compara *target* (lo buscado) con lo que está en el indice de en medio del arreglo, si el *target* es igual, retorna su indice (*mid*), si es menor que *target*, *low* se mueve a *mid*+1, y en caso que sea mayor que *target*, *high* se mueve a *mid*-1. *Mid* se calcula:

$$mid = low + (high - low)/2$$

Ejemplo: $target = 23$

1. **Inicio:**

Índice	0	1	2	3	4	5	6	7	8	9
Dato	2	5	8	12	16	23	38	56	72	91

   $low = 0$, $high = 9$, $mid = 4$

2. **Segunda Iteración:**

Índice	5	6	7	8	9
Dato	23	38	56	72	91

   $low = 5$, $high = 9$, $mid = 7$

3. **Tercera Iteración:**

Índice	5	6
Dato	23	38

   $low = 5$, $high = 6$, $mid = 5$

Utilizar *Binary Search* es **muy eficiente** ya que tiene un *Big O* de $O(\log(N))$, sin embargo, para poder hacer uso de esta búsqueda, el arreglo **debe** estar ordenado, en caso contrario no funcionará. Referencia (click aquí)

1.2. Ejercicios

1.2.1. Ejercicio 1: Compilador a Papel

Utilizando *Binary Search*, ejecute las siguientes configuraciones y diga el retorno obtenido:

1. $target = 4$

1	2	3	4	5	6	7	8	9	10

Cuadro 1: Arreglo 1

2. $target = 3$

-2	-1	3	11	44	100

Cuadro 2: Arreglo 2

3. $target = 91$

10	14	29	33	51	55	63	77	90

Cuadro 3: Arreglo 3

Solución:
1. ***Return*:** 3.
2. ***Return*:** 2.
3. ***Return*:** -1.

1.2.2. Ejercicio 2: Buscando Puntos

Utilizando la siguiente clase:

```
public class camion{
    String patente;
    float altura;
    float anchura;
    float profundidad;
    public camion(String patente, float altura, float anchura, float profundidad){
        this.patente = patente;
        this.altura = altura;
        this.anchura = anchura;
        this.profundidad = profundidad;
    }
    public float getCubicMeter(){
        return altura*anchura*profundidad;
    }
}
```

Listing 1: Clase camion

Cree un método que reciba los metros cúbicos necesarios para una carga y un arreglo **ordenado** con los metros cúbicos de cada camión disponible, debe retornar el índice del camión en el cual la carga cabe. Ejemplo:

- ***Input*:** *Target_altura*, *Target_anchura*, *Target_profundidad* y Arreglo con medidas.
- ***Output*:** Índice de camión que puede aceptar a carga.

  Ejemplo
- ***Input*:** 2 10 5 25 44 50 88 90 100
- ***Output*:** 5

Solución: Calcular metros cúbicos y utilizarlos como *target* con *binary search*.

```java
public static int busquedaBinariaCamiones(float altura, float anchura,
                                          float profundidad, camion[] arr){
    float target = altura*anchura*profundidad;
    int low = 0;
    int high = arr.length()-1;
    while(low <= high){
        int mid = low + (high-low)/2;
        if(arr[mid].getCubicMeter == target){
            return mid;
        }else if(arr[mid].getCubicMeter < target){
            high = mid - 1;
        }else{
            low = mid + 1;
        }
    }
    return -1;
}
```

Listing 2: Solución *binary search*

2. Ordenamiento

2.1. Conceptos

El ordenamiento siempre trata sobre recibir un arreglo y ordenarlo bajo cierto parámetro, y para ello existen distintas formas de lograrlo, pero tienen distintas características y puede que uno sea mejor que otro. Algunos conceptos importantes:

- **Estabilidad:** Mantiene el orden relativo de los elementos que son iguales en el parámetro por el cual están siendo ordenados.
- ***In-place*:** No utiliza memoria extra significativa.

Algoritmo	mejor caso *Big O*	promedio *Big O*	peor caso *Big O*	¿Es estable?	*link*
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Si	Click aquí
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	No	Click aquí
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Si	Click aquí
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Si	Click aquí
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	Click aquí
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No	Click aquí

Cuadro 4: Algoritmos de Ordenamiento

2.2. Ejercicio

2.2.1. Ordenamiento por distancia

Se tiene la siguiente definición de punto 2D en Java:

```java
public class point{
    float x;
    float y;
    public point(float x, float y){
        this.x = x;
        this.y = y;
    }
```

```
8       public float distanceToOrigin(){
9           return Math.sqrt(Math.pow(this.x, 2) + Math.pow(this.y, 2));
10      }
11  }
```

Listing 3: Clase point

Se tiene un arreglo con puntos 2D, y debes ordenarlo (con el algoritmo de ordenamiento de su gusto) según su distancia hacia el origen, sin embargo tu jefe es medio pesado y te pidió una cosita importante: Que no utilices memoria extra (más allá de la necesaria para el algoritmo escogido).

- ***Input*:** Arreglo de puntos a ordenar.
- ***Output*:** Arreglo ordenado de puntos.

  Ejemplo:
- ***Input*:** [(3,5), (4,4), (10,2), (1,1), (0,0), (-1,1)]
- ***Output*:** [(0,0), (1,1), (-1,1), (4,4), (3,5), (10,2)]

Responda también a las siguientes preguntas:

1. ¿Qué complejidad *Big O* tiene su algoritmo en un caso promedio?¿y en el peor caso?
2. ¿Con el algoritmo que implementó, qué pasará con los puntos que tengan la misma distancia al origen?

Solución: En esta pregunta se podía elegir cualquier algoritmo de ordenamiento, la siguiente implementación es con *mergeSort*.

```java
public static void mergeSort(point[] arr, int left, int right) {
    if (left < right){
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
public static void merge(point[] arr, int left, int mid, int right){
    int n1 = mid - left + 1;
    int n2 = right - mid;
    point[] leftArr = new point[n1];
    point[] rightArr = new point[n2];
    System.arraycopy(arr, left, leftArr, 0, n1);
    System.arraycopy(arr, mid + 1, rightArr, 0, n2);

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (leftArr[i].distanceToOrigin() <= rightArr[j].distanceToOrigin()) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
```

Listing 4: Solución problema de ordenamiento

1. El algoritmo escogido (*MergeSort*) en su caso promedio tiene una complejidad de $O(n \log n)$ y en el peor caso $O(n \log n)$.
2. Al ser un algoritmo estable, los puntos que tengan la misma distancia al origen, quedarán entre sí con el orden original del arreglo entregado inicialmente.

3. Tablas de Hash

3.1. Conceptos

Una tabla de *Hash* es aquella que a través de una función de *Hash* mapea las claves ingresadas a un indice del arreglo, lo cual conlleva:

- **Debe** ser determinista: misma clave -> mismo índice.

- Eficiente de calcular: $O(1)$.

Pueden existir **colisiones**, se generan cuando dos claves a través del cálculo, llegan al mismo índice.

Propiedades que se buscan a través de una tabla de *hash*:

1. **Uniformidad:** Distribuye las claves uniformemente.
2. **Determinismo:** Con la misma clave, siempre produce el mismo resultado.
3. **Eficiencia:** Rápida de calcular, eficiencia en sus operaciones.
4. ***Avalanche Effect*:** Pequeños cambios generan grandes cambios en hash.

Complejidad de operaciones:

Operación	***Big O***
Inserción	$O(1)$
Eliminación	$O(1)$
Búsqueda	$O(1)$

Cuadro 5: Tablas de *hash*: Complejidades temporales

Referencia (Click aquí)

4. Sets

4.1. Conceptos

Es un conjunto o colección de datos, con el objetivo de agrupar bajo el parámetro definido y realizar búsquedas eficientes de si un elemento se encuentra en el conjunto o no. Operaciones típicas de un *set* (o conjunto):

- Insertar(*type* x): Insertar un elemento en el conjunto.
- Eliminar(*type* x): Elimina el elemento del conjunto.
- Buscar(*type* x): Busca si un elemento se enuentra en el conjunto o no.

En java existen dos principales implementaciones de set; *HashSet* (click aquí) y *TreeSet* (click aquí), la principal diferencia es que *TreeSet* está implementado con una variante de los arboles de búsqueda binaria, los cual conlleva tener distintas funciones y complejidades, también mantiene los elementos ingresados ordenados de menor a mayor.

- ***HashSet*:**

Función	**Descripción**	**Complejidad temporal**
add(*type* x)	Añade el elemento al conjunto	$O(1)$
contains(*type* x)	Retorna *true* si el elemento está en el conjunto, en caso contrario, *false*.	$O(1)$
remove(*type* x)	Elimina el elemento del conjunto	$O(1)$

Cuadro 6: *HashSet* funciones principales

- ***TreeSet*:**

Función	Descripción	Complejidad temporal
add(*type* x)	Añade el elemento al conjunto	$O(\log n)$
contains(*type* x)	Retorna *true* si el elemento está en el conjunto, en caso contrario, *false*.	$O(\log n)$
remove(*type* x)	Elimina el elemento del conjunto	$O(\log n)$
lower(*type* x)	Retorna el antecesor del elemento ingresado	$O(\log n)$
higher(*type* x)	Retrona el sucesor del elemento ingresado	$O(\log n)$

Cuadro 7: *TreeSet* funciones principales

4.2. Ejercicios

4.2.1. Intersección

Dado dos arreglos de números enteros, se busca que retorne (o enseñe en pantalla) los números que están presentes en ambos arreglos. Ejemplo

- Arreglo 1: [4, 9, 5]
- Arreglo 2: [9, 4 ,9 8, 4]
- *Output*: [9, 4]

Solución: Crear un set con los números de un arreglo, luego iterando en segundo arreglo se agregan los números que estén en el set en un segundo set, finalmente se imprimen los números del set de intersección o se pasan a un arreglo para retornar.

```
public static int[] intersection(int[] arr1, int[] arr2){
        Set<Integer> set = new HashSet<>();
        Set<Integer> intersection = new HashSet<>();
        for(int i = 0 ; i < arr1.length ; i++){
            set.add(arr1[i]);
        }
        for(int i = 0 ; i < arr2.length ; i++){
            if(set.contains(arr2[i])){
                intersection.add(arr2[i]);
            }
        }
        int i = 0;
        int[] sol = new int[intersection.size()];
        for(int num : intersection){
            sol[i] = num;
            i++;
        }
    }
```

Listing 5: Solución *Set*

5. Maps

5.1. Conceptos

Es un conjunto similar a *set*, sin embargo trabaja bajo el concepto de 'llave-valor', lo cual significa que cada llave tiene un valor asociado, los valores pueden repetirse, sin embargo las llaves entre sí no. Operaciones típicas de un *map*:

- Insertar(*key*, *value*): Inserta un par llave-valor, si ya existe, se sobreescribe.
- ObtenerValor(*key*): Obtiene el valor asociado a la llave entregada.

- Eliminar(*key*): Elimina el par llave-valor con la llave asociada.
- Buscar(*key*): Busca y verifica si una llave se encuentra en el *map*.

En java al igual que con *set*, existen dos principales implementaciones de *map*; *HashMap* ((click aquí) y *TreeMap* (Click aquí), la principal diferencia es que *TreeMap* está implementado con una variable de los arboles de búsqueda binaria, lo cual conlleva a tener distintas funciones y complejidades, también mantiene los elementos ingresados ordenados de menor a mayor según su valor.

- ***HashMap*:**

Función	**Descripción**	**Complejidad Temporal**
put(*key*, *value*)	Añade el elemento al conjunto, en caso de existir la llave, lo sobreescribe	$O(1)$
get(*key*)	Retorna el valor asociado a la llave	$O(1)$
remove(*key*)	Elimina el elemento asociado a esa llave	$O(1)$
containsKey(*key*)	Verifica que exista un dato clave-valor para esta llave, si es así, retorna true, caso contrario *false*.	$O(1)$

Cuadro 8: *HashMap* funciones principales

- ***TreeMap*:**

Función	**Descripción**	**Complejidad Temporal**
put(*key*, *value*)	Añade el elemento al conjunto, en caso de existir la llave, lo sobreescribe	$O(\log n)$
get(*key*)	Retorna el valor asociado a la llave	$O(\log n)$
remove(*key*)	Elimina el elemento asociado a esa llave	$O(\log n)$
containsKey(*key*)	Verifica que exista un dato clave-valor para esta llave, si es así, retorna true, caso contrario *false*.	$O(\log n)$
lowerKey(*key*)	Retorna la llave antecesora de la llave ingresada	$O(\log n)$
higherKey(*key*)	Retorna la llave sucesora de la llave ingresada	$O(\log n)$

Cuadro 9: *TreeMap* funciones principales

5.2. Ejercicios

5.2.1. Verdadero y falso

Responda las siguientes afirmaciones con verdadero o false, en caso de ser falso, justifique.

1. _____ *HashMap* permite múltiples valores repetidos.
2. _____ *HashMap* garantiza que se tendrán los datos ordenados.
3. _____ Un *map* cuando hay una colisión en una llave, borra lo anterior y escribe lo nuevo para esa llave.
4. _____ La eficiencia entre utilizar un *HashMap* y un *TreeMap* es la misma.
5. _____ Gracias a *HashSet*, puedo realizar búsquedas del mínimo de un conjunto de números en $O(1)$.
6. _____ Un *set* es más eficiente que un *map*.
7. _____ Un *HashSet* es menos eficiente que un *TreeMap*.

Solución:
1. **Verdadero**, ya que las *keys* (llaves) son las que son únicas, los valores si pueden repetirse.
2. **Falso**, *TreeMap* o *TreeSet* mantienen los datos ordenados.
3. **Verdadero**.
4. **Falso**, *HashSet* trabaja con $O(1)$ mientras que *TreeMap* $O(\log n)$.
5. **Falso**, TDA *HashSet* no es para realizar búsquedas, y en caso de buscar, se tendría que hacer a través de un *iterator* y tendríamos un $O(n)$.
6. **Falso**, tienen la misma eficiencia $O(1)$, aunque también depende de la implementación.
7. **Falso**, *HashSet* utiliza funciones $O(1)$, mientras que *TreeMap* $O(\log n)$.

6. Priority Queue

6.1. Conceptos

Una *Priority Queue* funciona similar a su análogo sin prioridad (*Queue*), sin embargo, los elementos se mantienen ordenados de manera descendiente o ascendiente según la implementación requerida y un *comparator* entregado, por otro lado, en estas también se encolan y desencolan los elementos. Los principales métodos de una *Priority Queue* en *java* son:

Función	Descripción	Complejidad Temporal
add(*type* x)	Encola un elemento en la cola (internamente ordena según *comparator* utilizado)	$O(\log n)$
poll()	Desencola el elemento con menor prioridad (por defecto, puede modificarse a través de *comparator*), también reordena los elementos restantes según prioridad	$O(\log n)$
peek()	Retorna el próximo elemento a desencolar de la *queue* según prioridad, sin embargo, no lo saca de la cola	$O(1)$

Cuadro 10: *Priority Queue* funciones principales

6.2. Ejercicios

6.2.1. First Missing Positive

Dado un arreglo de números enteros entregado, debe retornar (o imprimir en pantalla) el número natural más pequeño que **no** esté presente en los arreglos. Ejemplo:

- Arreglo: [3, 4, -1, 1]
- *Output*: 2

Condición especial: Su código debe tener un *Big O* igual a $O(n)$.

Solución: Se agregan los datos del arreglo a la *Priority Queue* en orden ascendente, los números naturales empiezan en 1, si se encuentra en la cola, se aumenta en uno y sigue. La solución es $O(n)$ al agregar los números a la cola.

```java
public static int missing(int[] arr){
    PriorityQueue<Integer> nums = new PriorityQueue<>();
    for(int i = 0 ; i < arr.length ; i++){
        nums.add(arr[i]);
    }
    int min = 1;
    while(!nums.isEmpty()){
        int a = nums.poll();
        if(a == min){
            min++;
        }
    }
    return min;
}
```

Listing 6: Solución *Priority Queue*

7. Binary Search Tree (BST)

7.1. Conceptos

Los árboles de búsqueda binaria son una estructura de datos similar a una *Linked List*, sin embargo en vez de tener un nodo *next*, se tendrán dos 'hijos', un *left* y un *right*, los cuales siguen la siguiente lógica:

$$node.left < node.value$$

$$node.right > node.value$$

La implementación (referencia, click aquí) originalmente no admite duplicados, sin embargo, se puede modificar alguno de sus lados, permitiendo menor-igual o mayor-igual en alguno de los casos (*left* o *right*). En código es:

```java
public class node{
    int value;
    node left, right;
    public node(int value){
        this.value = value;
        left = right = null;
    }
}
```

Listing 7: implementación BST

Conceptos importantes que debes conocer:

- **Raíz:** Nodo inicial del árbol.
- **Ramas:** Nodos intermedios entre raíz y hojas.
- **Hojas:** Nodos finales, estos no tienen ningún hijo (ni *left* ni *right*).

Un árbol gráficamente:

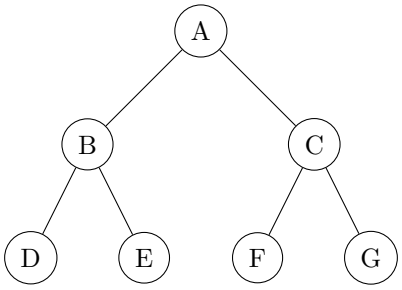

Figura 1: Gráfica de un árbol balanceado

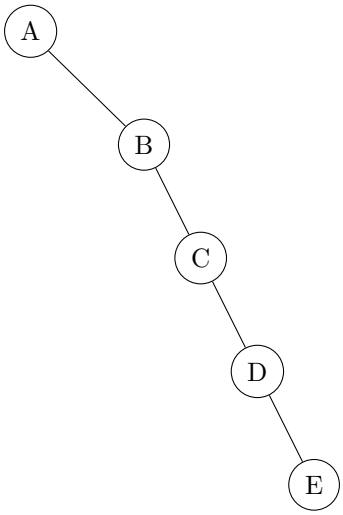

Figura 2: Gráfica de un árbol desbalanceado

A continuación se muestran las funciones básicas de todo árbol de búsqueda binaria, en el cual el caso promedio es que el árbol esté balanceado y el peor caso es que el árbol se encuentre desbalanceado.

Función	**Descripción**	**Complejidad temporal (caso promedio)**	**Complejidad temporal (peor caso)**
Insert(*type* x)	Insertar un nuevo nodo (con *value*) en árbol	$O(\log n)$	$O(n)$
Delete(*type* x)	Eliminar un nodo presente en el árbol	$O(\log n)$	$O(n)$
Search	Buscar un elemento en el árbol	$O(\log n)$	$O(n)$

Cuadro 11: *BST* funciones principales

Existen 3 formas de recorrer e imprimir un árbol, las cuales son:

1. ***preOrder*:** Imprimir, luego moverse al nodo izquierdo y luego al derecho.

```
preOrder(root){
    if(root == null){
        return;
    }
    print(root.value);
    preOrder(root.left);
    preOrder(root.right);
}
```

Listing 8: preOrder

2. ***inOrder*:** Se mueve al nodo izquierdo, imprime y luego al nodo derecho, este caso en partícular entrega los numeros ordenados de un *BST*.

```
inOrder(root){
    if(root == null){
        return;
    }
    inOrder(root.left);
    print(root.value);
    inOrder(root.right);
}
```

Listing 9: inOrder

3. ***postOrder*:** Se mueve al nodo izquierdo, luego al derecho y finalmente imprime.

```
postOrder(root){
    if(root == null){
        return;
    }
    postOrder(root.left);
    postOrder(root.right);
    print(root.value);
}
```

Listing 10: postOrder

7.2. Ejercicios

7.2.1. Imprimir...

Teniendo el siguiente árbol:

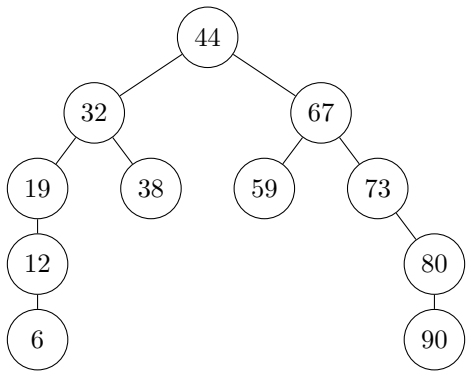

Figura 3: Árbol binario

Muestre el *output* que se obtiene al ejecutar *preOrder*, *inOrder*, *postOrder* con este árbol.

Solución:
- ***PreOrder*:** 44, 32, 19, 12, 6, 38, 67, 59, 73, 80, 90.
- ***InOrder*:** 6, 12, 19, 32, 38, 44, 59, 67, 73, 80, 90.
- ***PostOrder*:** 6, 12, 19, 38, 32, 59, 90, 80, 73, 67, 44.

7.2.2. Ejecutando...

Dada la siguiente secuencia: [-4, -1, 0, 3, 5, 6, 8, 10, 11], dibuje el árbol que resultará al insertar en tal orden y responda las siguientes preguntas:

1. ¿Qué problema se observa en su árbol?
2. ¿Cual es la complejidad que tendría realizar una inserción en el árbol?¿y una eliminación?¿y una búsqueda?

Ahora reordene la secuencia de tal manera que quede un árbol sin problemas y responda:

1. ¿Cual es la complejidad que tendría realizar una inserción en el árbol?¿y una eliminación?¿y una búsqueda?
2. Ejecute la búsqueda del número 12, enseñe todos los nodos por los que pasa la búsqueda.

Solución:

- Árbol que genera al insertarse bajo tal secuencia:

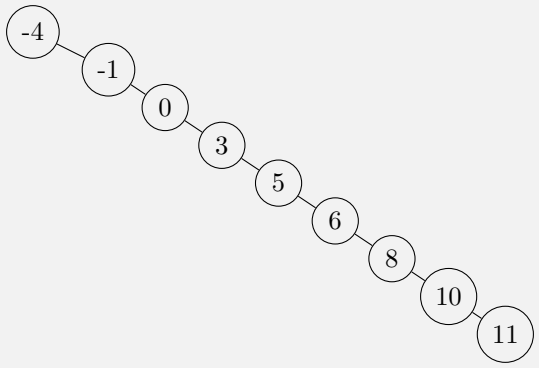

Figura 4: Gráfica de un árbol desbalanceado

1. El problema del árbol es que no está para nada balanceado, pareciendo una *Linked List*.
2. La complejidad para las 3 funciones para este árbol es $O(n)$, esto ya que no está balanceado, y para realizar cualquiera de estas acciones se recorrerán los nodos de manera secuencial.

- Para que resulte en un árbol sin problemas (balanceado) debe insertarse bajo el siguiente orden: [5, -1, 8, -4, 0, 3, 6, 10, 11], resultando en el siguiente árbol:

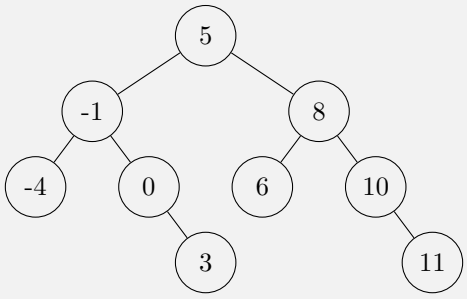

Figura 5: Árbol binario resultante de secuencia

1. Ahora que el árbol está balanceado, sus complejidades pasan a ser $O(\log n)$.
2. Nodos de búsqueda de 12: [Root: 5 -> 8 -> 10 -> 11 -> null], se llega a null ya que 12 no se encuentra en el árbol.