Proyecto EDAII: Árboles Binarios

Arroyo Moreno Diego Alejandro
Rosario Vázquez José André

6 de diciembre del 2021

1. Objetivo

Que el alumno implemente aplicaciones relacionadas con los árboles binarios y que desarrolle sus habilidades de trabajo en equipo y programación orientada a objetos.

2. Introducción

En este proyecto se van trabajar las implementaciones sobre distintos tipos de árboles binarios. Para realizar este trabajo, hace falta tener un conocimiento sobre la programación orientada a objetos. Además, se tiene que conocer las operaciones básicas de los árboles binarios normales como de búsqueda. En este caso, se debe implementar un menú de usuario, donde se tratan tres tipos de árboles binarios que son los AVL, Heaps y los de expresión aritmética. Con estos árboles la operación que tienen común es mostrar el árbol y agregar elementos para formarlos, y cada uno tiene sus pequeñas diferencias. Los árboles a nivel conceptual son diferentes asimismo poseen propósitos diferentes, en este proyecto se dará a conocer su implementación y su metodología.

3. Desarrollo

3.1. Árboles Binarios de Busqueda Balanceados (AVL)

Los árboles binarios de búsqueda balanceados tratan de mantener la estructura del árbol de manera balanceada. Tratan de mantenerlo cuando se realizan operaciones de inserción y eliminación. Los creadores de estos árboles son matemáticos de nacionalidad rusa sus nombres son Adelson, Velskii y Landis. Con las iniciales de sus nombres se da el identificador principal de estos que es *AVL*.

Como este es un árbol binario este respeta la propiedad de orden en todos sus nodos. Este orden es donde todas la claves del subárbol izquierdo son menores al nodo y las claves derechas son mayores. Otra propiedad o característica que posee este árbol es la del balanceo. Consiste en que para cada nodo del árbol, la diferencia de altura entre el subárbol izquierdo y el subárbol derecho es a lo sumo 1.

Estas diferencias de alturas se definen mediante el factor de equilibrio. Este factor se calcula de la siguiente manera:

Fe (nodo) = Altura (subárbol izquierdo) - Altura (subárbol derecho)

Los valores entornan entre 1, 0, -1. Sí estos valores son mayores o menores no puede ser un árbol AVL, debido a que no está balanceado.

Para el momento de agregar o eliminar una clave, los ascendientes del nodo puede sufrir un cambio en sí factor de equilibrio, pero para todos los casos debería de ser de una sola unidad. Para equilibrar el árbol se necesita de las operaciones llamadas rotación para que regrese a ser un árbol AVL.

3.1.1. Rotaciones

Las rotaciones son movimientos que modifican la estructura del árbol. Estos movimientos se manejan para la inserción y eliminación en los árboles AVL. Estos árboles necesitan las modificaciones para mantenerse balanceados, para no afectar la estructura que poseen los árboles. Como ya se mencionó, el factor de equilibrio es una buena herramienta para identificar el balanceo del árbol. Si los valores son mayores al rango del árbol, esto significa que se debe usar estas rotaciones para mantenerlo balanceado. Las rotaciones son 4, están clasificadas de la siguiente manera:

- Rotación simple a la derecha
- Rotación simple a la izquierda
- Rotación doble a la derecha
- Rotación doble izquierda

Rotación simple a la derecha

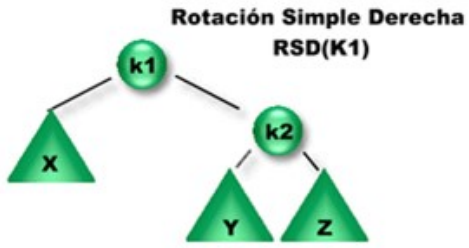

Figura 1: Ejemplo rotación simple a la derecha

Para entender cómo se realiza la rotación, se va mostrar un ejemplo donde se va a rotar el nodo k1. Mediante el siguiente pseudocódigo se realiza la rotación:

1. k2 = k1.hijoDerecho
2. k1.hijoDerecho = k2.hijoIzquierdo
3. k2.hijoIzquierdo = k1
4. Retomar k2 para que tome el lugar anterior de k1 en el árbol

Siguiendo los pasos el recorrido queda resuelto:

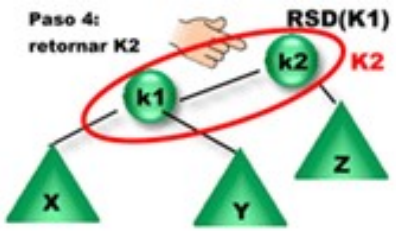

Figura 2: Fin de la rotación simple a la derecha

Rotación simple izquierda

Figura 3: Ejemplo de la rotación simple a la izquierda

La metodología con la rotación anterior es muy parecida. El pseudocódigo para realizar la rotación desde el nodo k1 se hace de la siguiente manera:

1. k1 = k2.hijoIzquierdo
2. k2.hijoIzquierdo = k1.hijoDerecho
3. k1.hijoDerecho = k2
4. Retomar k1 para que tome el lugar anterior de k2 en el árbol

Siguiendo los pasos la rotación queda finalizada.

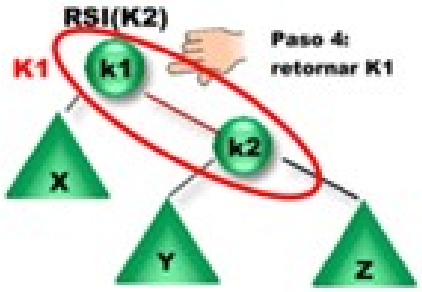

Figura 4: Fin de la rotación simple a la izquierda

Rotación doble a la derecha

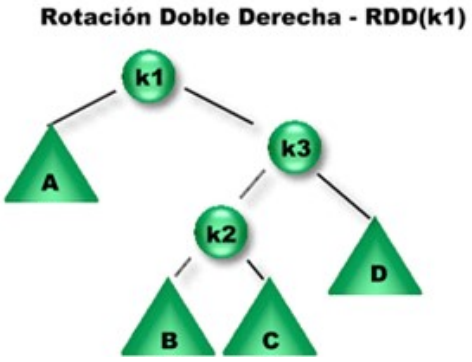

Figura 5: Ejemplo de la rotación doble a la derecha

Este tipo de rotaciones se realizan cuando los cambios involucran más nodos. Pero en realidad este tipo de rotaciones son combinaciones de las rotaciones simples. El pseudocódigo para realizar este tipo de rotación sobre el nodo k2 se realiza:

1. k2 = k1.hijoDerecho
2. Rotación simple izquierda (k2)
3. Rotación simple derecha (k1)

Completando la rotación, queda de la siguiente manera:

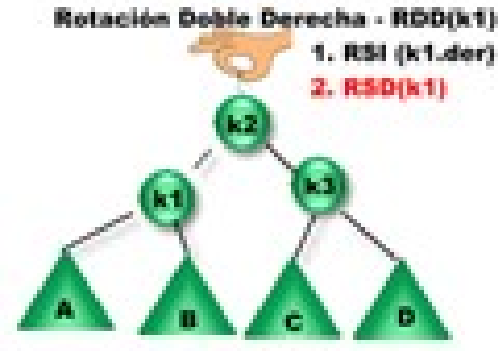

Figura 6: Fin de la rotación doble a la derecha

Rotación doble a la izquierda

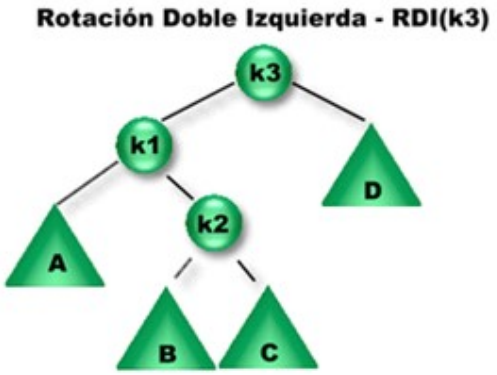

Figura 7: Ejemplo de la rotación doble a la izquierda

En este caso posee una forma muy parecida con respecto al anterior. En este caso se va a realizar la rotación sobre el nodo k2, se realiza de la siguiente manera:

1. k1 = k2.hijoIzquierdo
2. Rotación Simple Derecha (k1)
3. Rotación simple izquierda (k2)

Siguiendo los pasos la rotación termina de la siguiente manera:

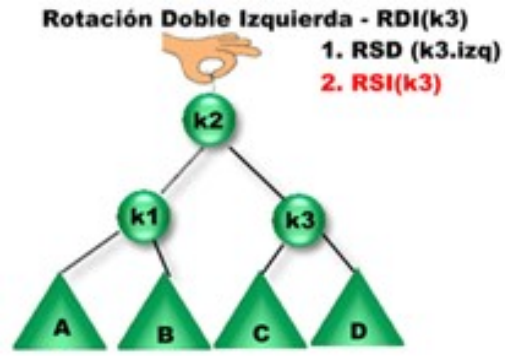

Figura 8: Fin de la rotación doble a la izquierda

3.1.2. Inserción

Para agregar claves al árbol, se necesitó plantear en premisa las rotaciones debido a que son fundamentales para esta operación. En esencia se agregan de la misma manera que un árbol binario de búsqueda, donde los valores mayores a la raíz se encuentran en el subárbol derecho y los menores en el izquierdo. Lo que cambia es que este árbol debe estar balanceado esta estructura. La modificación principal para mantenerlo balanceado es por medio de las rotaciones, estas ayudan a mantener de esta forma el árbol.

El proceso para la inserción es la siguiente:

1. Buscar hasta encontrar la posición de inserción o modificación (proceso idéntico a un árbol binario de búsqueda).

2. Insertar el nuevo nodo con factor de equilibrio (balanceado)

3. Recorrer el camino de búsqueda, verificando el equilibrio de los nodos, y re-equilibrando si es necesario para mantener el árbol equilibrado

3.1.3. Eliminación

La eliminación es un proceso complicado, debido a que es más común que el árbol pierda su equilibrio. Como con el proceso de inserción, su procedimiento es muy similar a un árbol binario de búsqueda. Por lo tanto, una vez eliminado un elemento, se debe deshacer el camino tomado verificando que los nodos del camino se encuentren balanceados. En caso de que no lo esté se debe operar por rotaciones.

3.2. Árbol Binario Semicompleto

Decimos que un árbol binario es semicompleto cuando todos sus niveles están completos a excepción del último cuyos nodos deben aparecer de izquierda a derecha. El siguiente árbol binario es semicompleto.

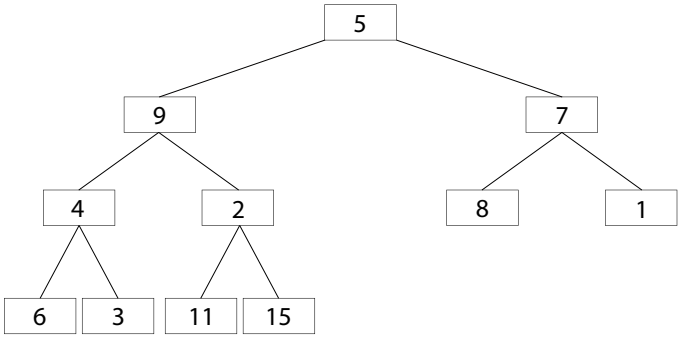

Figura 9: Árbol Binario Semicompleto

3.2.1. Heap (montículo)

Un montículo es un árbol binario semicompleto que tiene la característica de que el valor de cada nodo padre resulta ser mayor que el valor de cualquiera de sus hijos. En este caso, diremos que se trata de un "montículo de máxima". Análogamente, podemos hablar de un "montículo de mínima" cuando el valor de cada nodo padre es menor que los valores de sus hijos. Por ejemplo, el siguiente árbol binario es un montículo porque, además de ser semicompleto, se verifica que el valor de cada nodo padre es mayor al valor de cada uno de sus hijos.

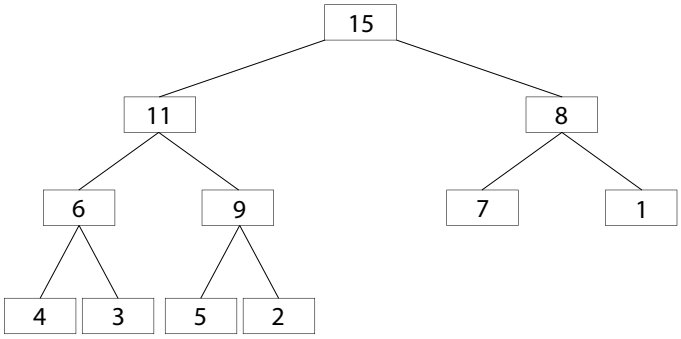

Figura 10: Árbol Binario Semicompleto donde cada padre es mayor que sus hijos (heap)

3.2.2. Transformar un árbol binario semicompleto en un heap

El algoritmo consiste en procesar cada terna (padre, hijo izquierdo, hijo derecho) para permutar el valor del padre por el valor de mayor de sus hijos; salvo, que el padre ya sea el mayor de los tres.

El proceso debe ser secuencial, comenzando desde el último padre, luego el anteúltimo y así, sucesivamente, hasta llegar a procesar la raíz.

Desde la **figura 9** que ilustra el árbol binario semicompleto vemos que el último padre es 2; luego, al permutarse por el mayor de sus hijos obtendremos la terna: [15, 11, 2]. El siguiente padre (avanzando hacia la izquierda) es 4 que, al permutarse por el mayor de sus hijos nos dará la terna: [6, 4, 3]. El próximo padre que debemos considerar es el 7 que, luego de procesar su valor y el de sus hijos formará la terna: [8, 7, 1].

El siguiente padre que corresponde procesar es 9 pero, antes de analizarlo veremos cómo quedó el árbol binario luego de haber aplicado todas estas permutaciones.

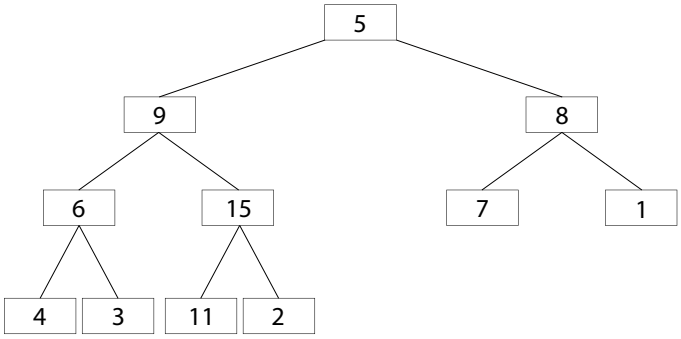

Figura 11: Árbol Binario luego de permutar 2 por 1, 4 por 6 y 7 por 8

Ahora podemos procesar el nodo 9 y obtener la terna: [15, 6, 9]. El problema es que, luego de esto, 9 quedará posicionado como padre de 11 y 2, haciendo que esta rama del árbol deje de ser montículo.

Para solucionarlo, cada vez que permutamos a un padre por alguno de sus hijos, repetiremos el proceso de cascada hacia abajo. Es decir, si permutamos 15 por 9, entonces consideraremos la *terna* que se forma entre el valor permutado (9) y sus (nuevos) hijos. En este caso debemos reconsiderar la *terna*: [9, 11, 2] y, obviamente, la permutamos así: [11, 9, 2]. Veamos el estado actual del árbol, donde aún queda pendiente el proceso del nodo raíz.

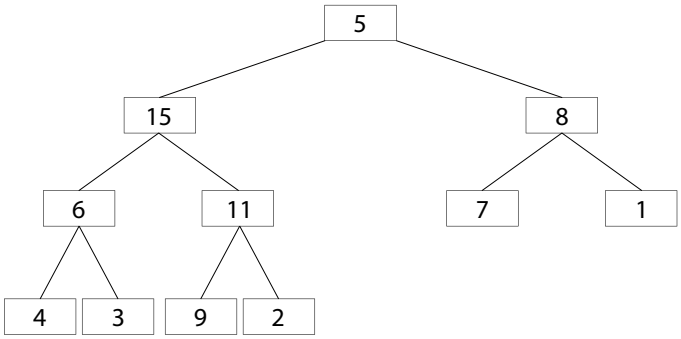

Figura 12: Árbol Binario con todos sus nodos procesados, excepto la raíz

Por último, queda por procesar la terna raíz: [5, 15, 8]. Al permutar 5 por

15, posicionamos a 5 como padre de 6 y 11. Esto nos obligará a reprocesar el elemento permutado, así que evaluaremos la terna: [5, 6, 11] y permutamos 5 por 11. Esta permutación, ahora, ubicó a 5 como padre de 9 y 2, obligándonos a procesar la terna: [5, 9, 2] para finalizar permutando 5 por 9. Con esto, el árbol quedó convertido en montículo.

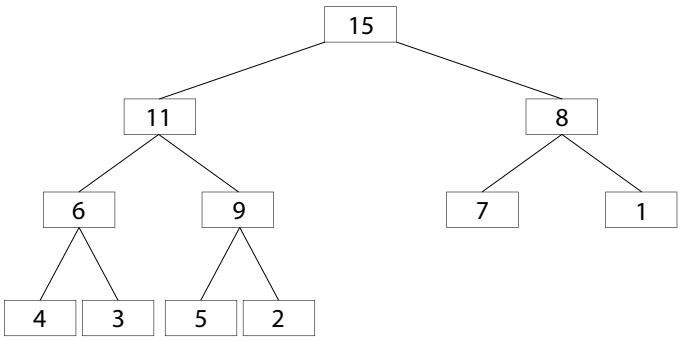

Figura 13: Heap con todos sus nodos procesados

3.3. Árbol de expresión aritmética

Para resolver expresiones matemáticas es un reto, y una de las mejores formas de resolver estas expresiones es mediante árboles binarios. Los valores numéricos van a ser nodos hojas, mientras que los operadores van a ser los nodos internos . Con esta pequeña explicación se muestra el siguiente algoritmo:

1. El primer nodo se convierte en la raíz del árbol de representación.
2. El segundo nodo se convierte en la raíz cuando se inserta, y el nodo raíz se convierte en el hijo izquierdo del nodo.
3. Cuando el nodo insertado sea un número, se inserta en el extremo derecho de la cadena derecha del nodo raíz.
4. Cuando el nodo insertado es un operador, primero se compara con la prioridad de operador del nodo raíz.

- Cuando la prioridad no es alta, el nuevo nodo se convierte en el nodo raíz y la expresión original se convierte en el subárbol izquierdo del nuevo nodo.
- Cuando la prioridad es alta, el nuevo nodo se convierte en el hijo derecho del nodo raíz original y se convierte en el subárbol izquierdo del nuevo nodo.

Teniendo el algoritmo de cómo se construye este árbol, es conocido que si se realiza un recorrido postorden sobre el árbol para la notación polaca inversa.

3.3.1. Notación polaca inversa

Es un método algebraico alternativo para la introducción de datos. Al ser inversa primero se encuentran los valores numéricos o operandos y después el operador que realiza los cálculos. Esta notación no necesita usar paréntesis para conocer el orden de las operaciones, mientras que el número de argumentos necesarios para que el operador funcione. Se muestra un pequeño ejemplo de notación polaca inversa.

$$345++ = 3+4+5$$

De esta manera se ve la nomenclatura de la notación polaca inversa.

4. Análisis del programa

El programa consta de dos tipos de árboles, el primer árbol que se va a explicar brevemente, es el árbol AVL. Posteriormente se explicará el árbol heap. No se va a explicar el árbol de expresión aritmética por términos de que no se realizó este árbol en el proyecto.

4.0.1. Clase NodoAVL

En esta clase se crea un objeto abstracto, donde se definen atributos para uno que representa la altura del nodo, otro su clave o valor; también se inicializa un atributo de la misma clase que representa el subárbol derecho e izquierdo del árbol. Con esto, se encuentran dos constructores, uno vacío para no tener problemas de instanciación dentro de clases y el último constructor que recibe una clave y se define su altura que es igual a uno.

4.0.2. Clase Avl

Esta clase es la más compleja debido a los métodos que se involucran y la lógica que poseen. Para empezar se crea un atributo de clase de la clase del nodo este atributo representa la raíz . Posteriormente se crean tres distintos constructores que representan las distintas formas de crear el árbol, dos pidiendo un dato como parámetro, mientras que el otro está vacío.

Posteriormente, se encuentran dos métodos que realizan la función de agregar, en esencia solo uno se debe usar para agregar, porque el otro método sirve para que funcione como una recursiva del método que realiza la inserción. En este método se agregan los nodos y se realizan las operaciones del factor de equilibrio y rotación para mantener balanceado el árbol. En este caso, fue de suma importancia que no recibiera datos duplicados debido a que este tipo de árboles no lo recibe por esas múltiples condiciones donde si es mayor se agrega al subárbol derecho y si es menor a la raíz se agrega a la izquierda en esencia esta operación funciona así. Cabe destacar que se obtienen las alturas para el correcto funcionamiento del factor de equilibrio, y actualizar las alturas.

Los métodos de buscar tienen una forma recursiva similar a los métodos. Donde una solo define e imprime si se encuentra o no el valor buscado, mientras que el otro método sólo mantiene la lógica para buscar el elemento por todo el árbol.

Los método elimina tiene una función más compleja debido a que se toman los principales casos en cuenta para que el funcionamiento de la eliminación sea la correcta. Evalúa casos de hojas, no tiene hijos el nodo, nodos con dos hijos con esta sintaxis de los casos se eliminan pero debe mantener el balanceado. Por este motivo se utiliza el factor de equilibrio para evaluar en qué caso se necesita que rotación de esta forma se trabaja de mejor manera la eliminación. Este posee otro método que utiliza de manera recursiva el método mencionado para que solo se pueda agregar la clave.

Por último, el método para imprimir fue tomado de una práctica anterior para que funcione lo más correcto posible debido a que es difícil de implementar una manera "gráfica del árbol", entonces en este caso se hace uso del recorrido BreathFirst para mostrar el árbol AVL.

Complementado se poseen como utilidades, la rotación a la derecha , a la izquierda; además de la obtención de las alturas, el factor de equilibrio, el máximo entre dos números, la altura y el nodo el valor máximo, estos método para que funcione cada una de las operaciones del árbol.

4.0.3. Clase Heap

Es la única clase que realiza un árbol heap. Para comenzar existe un método heapify que este recibe un arreglo y lo habilita para que se convierta en un heap. De esta manera mantiene el orden que establece un heap. Por otro lado, existe otro método que inserta un valor en el heap, donde se hace uso del método heapify para que mantenga el orden establecido por el árbol. De la misma manera que el primer método este recibe un arreglo para trabajar la inserción.

Por otro lado, la operación de eliminar, realiza la eliminación de un elemento del arreglo y este la integridad del heap se mantiene con el método heapify. Además, el método printArray, éste muestra en pantalla el heap en formato de arreglo y de la visualizaciones más comunes para el heap. Debido a que es más concurrente el uso de arreglos para los heap.

Por último, posee otro método alternativo para imprimir el heap, es cual tiene la peculiaridad que se ve de manera gráfica, ya no solo es un arreglo si no se ven sus ramificaciones. En este caso como puede ser más frecuente los errores se hace uso de excepciones, para tener un control dentro del error. Este método para mostrarlo en forma de árbol es complejo de implementar, aunque su debilidad son los valores de tres dígitos, ya que se pierden un poco los valores en las ramificaciones, pero sigue funcionando perfectamente.

5. Concluiones

Arroyo Moreno Diego Alejandro:
La falta de un compañero de equipo se vio directamente reflejada en el desarrollo de este proyecto. Entre mi compañero José André y yo nos repartimos las actividades de una buena manera, siendo así que el trabajo fue equitativamente resuelto. El desarrollo del proyecto desde el punto de la programación fue sencillo para la parte que me tocó implementar. Acudí a la primera práctica del curso y con ello fui capaz de desarrollar una estructura de Heap en el lenguaje de programación de Java. Con ello y un par de métodos que mi compañero me ayudó a implementar, el código del programa no fue la parte más difícil del proyecto. El objetivo fue medianamente cumplido, pues faltó una parte importante del programa: el árbol de expresión aritmética. Fue un algoritmo que ambos decidimos no programar, pues el tiempo se nos acababa y pensamos en mejorar los otros dos algoritmos y el comportamiento del

programa para compensar las cosas.

Rosario Vázquez José André:
Durante el proyecto se tuvieron bastantes retos, el primero fue empezar con uno menos para el desarrollo. Dejando esto de fuera, fue enriquecedor conocer más acerca de los árboles. Aunque es compleja la implementación de ambos, que se desarrollaron en este proyecto. El conocer características nuevas o implementaciones nuevas son una herramienta más. Lamentablemente no se pudo realizar la expresión aritmética, pero no cabe duda que el la más complicada de las tres. En general, considero que no se cumplieron todos los objetivos pero sí la mayor parte, volvería a intentar este proyecto para terminar de comprender el árbol de expresión aritmética.

Referencias

[1] Gonzalez, H. (8 de enero del 2019). ¿Qué es la notación polaca inversa?. De: https://es.quora.com/https://es.quora.com/Qué-es-la-notación-polaca-inversa

[2] Perez, G. (20 de septiembre de 2018). Definición de los árboles AVL. De: http://163.10.22.82/OAS/AVL_Definicion/definicin.html

[3] Programador Clic. (4 de abril de 2020). Usa un árbol para resolver cuatro expresiones aritméticas. De: https://programmerclick.com/article/17671056503/

[4] Sznajdleder, P. (2020). Algoritmos a fondo - Con implementaciones en C y Java (1.a ed.). Alfaomega.