Universidad Nacional Autónoma de México

Facultad de Ingeniería
ESTRUCTURAS DE DATOS Y ALGORITMOS II

Grupo: 07 - Semestre: 2024-1

PROYECTO 3 – Arboles Binarios

Fecha de entrega: 04/12/2023

Profesor:

Edgar Tista Garcia

Equipo 10 - Alumno(s):

Hernandez Gallardo Daniel Alonso
Garcia Riba Emilio
de la Cruz López, Oscar Abraham

Abstract

In this project, we will thoroughly review binary tree data structures such as Heap, AVL, and Arithmetic. We will focus on the algorithms and how to implement them in Java, with a special emphasis on object-oriented programming. Additionally, we will discuss potential issues or complications that may arise during implementation, exploring different approaches to address them. In summary, we will address how to design, implement, and execute binary trees efficiently in Java, considering the perspective of object-oriented programming and potential challenges that may arise along the way.

Resumen

En este proyecto, vamos a revisar en detalle las estructuras de datos de Árboles Binarios, como el Heap, AVL y Aritmético. Nos enfocaremos en los algoritmos y cómo implementarlos en Java, prestando especial atención a la programación orientada a objetos. Además, vamos a hablar sobre posibles problemas o complicaciones que podrían surgir durante la implementación, explorando diferentes enfoques para resolverlos. En resumen, abordaremos cómo diseñar, implementar y ejecutar eficientemente Árboles Binarios en Java, teniendo en cuenta la perspectiva de la programación orientada a objetos y las posibles dificultades que podríamos encontrar en el camino.

Índice general

1 Introducción

1.1. 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.

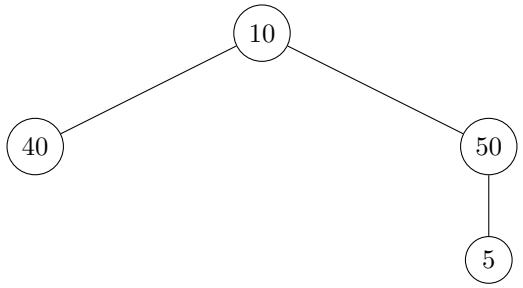
Árboles Binarios y Árboles Binarios Completos

En el mundo de las estructuras de datos, los árboles binarios juegan un papel esencial. Un árbol binario es una estructura jerárquica en la que cada nodo tiene, como máximo, dos hijos: uno izquierdo y uno derecho. Esta organización jerárquica facilita la representación y manipulación de datos.

Un caso especial de árboles binarios es el de los árboles binarios completos. En un árbol binario completo, todos los niveles están completamente llenos, excepto posiblemente el último, que se llena de izquierda a derecha. Esta propiedad asegura que el árbol tenga una estructura equilibrada y eficiente.

La propiedad de ser completo es crucial al hablar de Heaps, una estructura de datos basada en árboles binarios. Los Heaps aprovechan la estructura de árbol binario completo para su representación eficiente mediante arreglos, que es como comunmente se representan.

Ejemplo de Representación de Árbol Binario Completo



En este ejemplo, cada nivel se llena de izquierda a derecha, y el árbol resultante es un árbol binario completo. Esta estructura organizada facilita la implementación eficiente de Heaps, que exploraremos más adelante en la investigación.

1.2. Arbol AVL

Un árbol AVL es un árbol binario de búsqueda que cumple con la condición de que la diferencia entre las alturas de los subárboles de cada uno de sus nodos es, como mucho 1. Para medir este desequilibrio e indicar hacia que lado se hace uso de un factor de equilibrio que se consigo con la profundidad del subárbol derecho menos la profundidad del subárbol izquierdo, si la diferencia es mayor o igual a dos de signo positivo esta desequilibrado a la izquierda, y si es de signo contrario desequilibrado a la derecha. La denominación de árbol AVL viene dada por los creadores de tal estructura (Adelson-Velskii y Landis). La propiedad de equilibrio que debe cumplir un árbol para ser AVL asegura que la profundidad del árbol sea O(log(n)), por lo que las operaciones sobre estas estructuras no deberán recorrer mucho para hallar el elemento deseado. Como se verá, el tiempo de ejecución de las operaciones sobre estos árboles es, a lo sumo O(log(n)) en el peor caso, donde n es la cantidad de elementos del árbol. Sin embargo, y como era de esperarse, esta misma propiedad de equilibrio de los árboles AVL implica una dificultad a la hora de insertar o eliminar elementos: estas operaciones pueden no conservar dicha propiedad. A modo de ejemplificar esta dificultad,

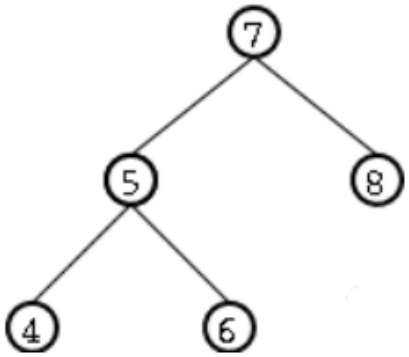


Figura 1.1: Árbol AVL de enteros

supongamos que al árbol AVL de enteros de Figura 1.1 le queremos agregar el entero 3. Si lo hacemos con el procedimiento normal de inserción de árboles binarios de búsqueda el resultado sería el árbol de Figura 1.2 el cual ya no cumple con la condición de equilibrio de los árboles AVL dado que la altura del subárbol izquierdo es 3 y la del subárbol derecho es 1.

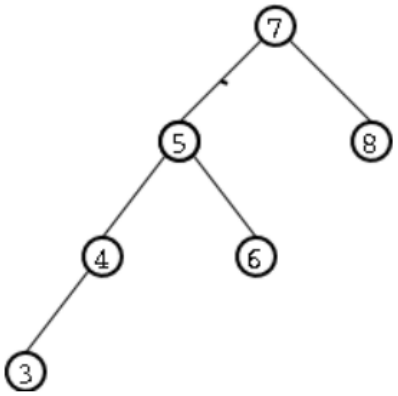


Figura 1.2: Árbol que no cumple con la condición de equilibrio de los árboles AVL.

Para evitar esto el arbol realiza una serie de operaciones que permiten rebalancearlo, quedando de la siguiente manera:

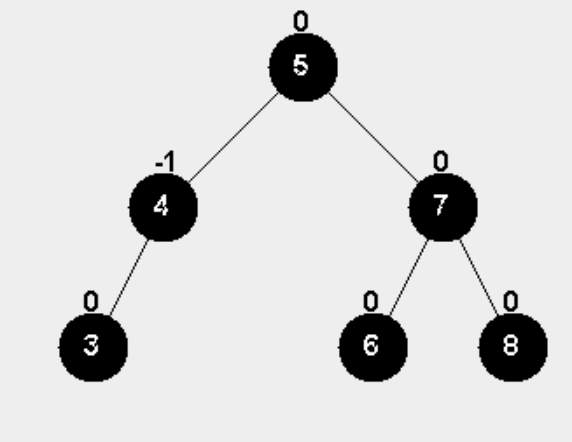


Figura 1.3: Árbol rebalanceado

1.3. Heap

La manipulación eficiente de datos es fundamental en el campo de la informática y la ciencia de la computación. Entre las diversas estructuras de datos que existen, los árboles binarios desempeñan un papel crucial. Estos árboles jerárquicos permiten organizar y acceder a datos de manera eficiente.

Una variante especial de los árboles binarios, conocida como árboles binarios completos, ha demostrado ser especialmente valiosa. Estos árboles completos brindan una estructura organizada que facilita la implementación de diversas operaciones.

En este contexto, nos enfocaremos en una estructura específica derivada de los árboles binarios: la estructura de Heap. Los Heaps son fundamentales para una variedad de aplicaciones, desde implementar colas de prioridad hasta facilitar operaciones eficientes en conjuntos de datos.

A lo largo de esta investigación, exploraremos en detalle los conceptos fundamentales de Heap, sus propiedades, operaciones clave como Heapify, y las aplicaciones prácticas de estas estructuras en el ámbito de las colas de prioridad. Este estudio proporcionará una comprensión integral de cómo los Heaps contribuyen a la eficiencia en el manejo de datos y la resolución de problemas computacionales.

1.3.1. Definición de Heap

Un Heap es esencialmente un árbol binario con propiedades específicas que lo hacen particularmente útil para ciertas operaciones eficientes. Existen dos tipos principales de Heaps: Max Heap y Min Heap.

- **Max Heap:** En un Max Heap, el valor de cada nodo es mayor o igual que el valor de sus hijos. El nodo con el valor máximo se encuentra en la raíz del árbol.
- **Min Heap:** En un Min Heap, el valor de cada nodo es menor o igual que el valor de sus hijos. El nodo con el valor mínimo se encuentra en la raíz del árbol.

Propiedades del Heap

La propiedad clave que distingue a los Heaps es la estructura de árbol binario completo. Además:

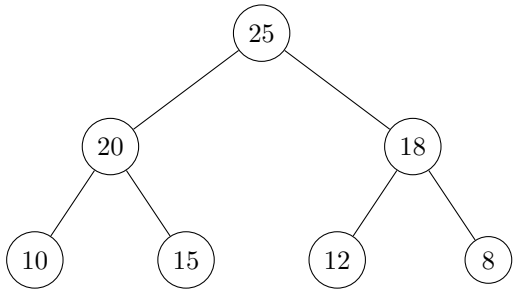
- En un Max Heap, cada nodo padre es mayor o igual que sus hijos.
- En un Min Heap, cada nodo padre es menor o igual que sus hijos.

Estas propiedades garantizan que el elemento más significativo (máximo o mínimo, según el tipo de Heap) esté siempre ubicado en la raíz, lo que es esencial para ciertas aplicaciones.

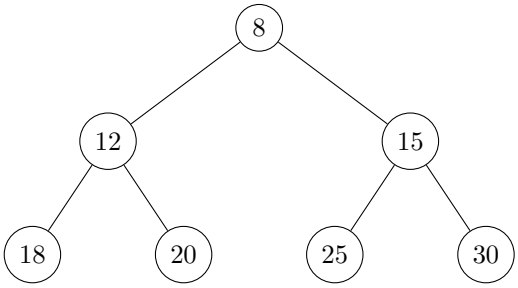
Tipos de Heap

Los Heaps se clasifican en dos tipos principales:

- **Max Heap:** Donde el elemento máximo está en la raíz.



- **Min Heap:** Donde el elemento mínimo está en la raíz.



Cada tipo de Heap tiene sus propias aplicaciones y ventajas, dependiendo de los requisitos del problema. En las secciones siguientes, exploraremos la estructura de los Heaps, cómo se representan, y las operaciones fundamentales que se pueden realizar sobre ellos, como Heapify y la construcción eficiente del Heap.

1.4. Árbol Aritmetico

El árbol de expresión aritmética es un árbol binario que sirve para poder representar una operación matemática en forma de un grafo. Con este grafo podemos resolver esta expresión considerando los paréntesis y el orden de operaciones.
En el árbol queremos que nuestros nodos hojas sean los números mientras que los nodos intermedios y la raíz sean operaciones. La profundidad de un nodo de operación estará indicada por dos criterios. Primero son los órdenes de operaciones, donde el orden de prioridad sería, raíz y potencias ¿ división y multiplicación ¿ suma y resta, entonces en teoría si hay una suma o resta en la expresión esta sería la raíz. En la realidad esto no es el caso ya que puede haber paréntesis en la expresión. Los paréntesis tienen mayor prioridad que cualquier operación entonces lo que esté dentro de estos se hará primero. Por ejemplo en la siguiente operación:

(2+3/4)*(5-7*(4-8))

Figura 1.4: Una expresión aritmética

Su forma en un árbol de expresión aritmética seria:

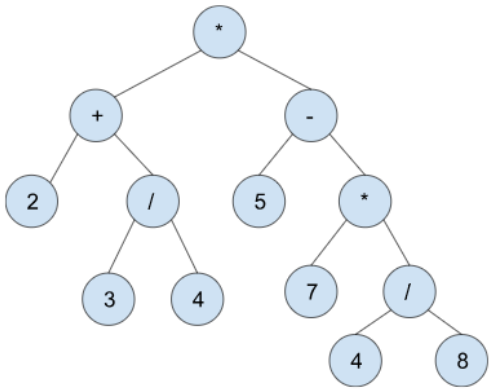


Figura 1.5: La forma de árbol de expresión aritmética de la anterior expresión.

Para resolver la expresión aritmética hacemos un recorrido posorder, haciendo una expresión en notación polaca inversa. En el ejemplo anterior el resultado seria: 2 3 4 / + 5 7 4 8 / * - *. Los positivos de la notación polaca inversa es que toma en cuenta el orden en las que las operaciones se tendrían que hacer, sin tener que utilizar paréntesis.

2 Desarrollo

2.1. Arbol AVL

2.1.1. Estructura del arbol

Definir una buena estructura del árbol es vital para realizar una buena implementación y más que nada para facilitarnos su creación, en especial en el lenguaje usado (Java) pues este al no manejar apuntadores y hacer todo mediante referencias, tenemos que definir ciertas variables que nos permitan el facil acceso a las distintas variables.

Clase AVL

Esta clase es la estructura principal del árbol; esta constituido por la raíz del árbol que es un objeto de tipo nodo, una variable estática de tipo Árbol AVL denominada MainTree pues este sera como la raíz principal del árbol, esto se hace asi ya que el arbol al recomendarse el nodo raíz con el que empezó puede moverse, así que se decidió porque solo exista una referencia al nodo que es la raiz y se vaya cambiando dinámicamente según los movimientos del árbol. También se tiene otra variable estática que es un contador de la cantidad de vertices totales del arbol, esto nos ayudara posteriormente para los algoritmos de recorrido, especialmente el BFS pues así se tiene la cantidad de nodos que hay por marcar en la lista de visitados.

Clase Nodo

Esta clase es de vital importancia pues aquí se tendrá el acceso a los subárboles que nos dará acceso a los subárboles, es importante que los nodos accedan a los subárboles porque así nos permiten abarcar un panorama más grande de la cantidad de referencias que se tienen y al tomar todo como subárboles es posible usar de manera más sencilla los métodos de clase de AVL, esta clase es más que nada para facilidad de acceso a las variables y generar más independencia. Tambien en este nodo se almacena el factor de equilibrio el cual nos indicara si el subarbol en el que nos encontramos esta en equilibrio, es decir, la cantidad de nodos derechos e izquierdos no difiere en 2 o más. Por lo tanto sus variables quedan como:

- AVL Padre: Es el subárbol padre que contiene al nodo en el que se esta.

- AVL Hizq: Es el subarbol hijo izquierdo del nodo en el que estamos.
- AVL Hder: Es el subarbol hijo derecho del nodo en el que estamos.
- int value: Es el valor o contenido del nodo.
- int eq: Es el factor de equilibrio que nos indica si el nodo esta en equilibrio o no.
- deepizq: Nos indica la profundidad total del subarbol hacia la izquierda.
- deepder: Nos indica la profundidad total del subarbol hacia la derecha.
- deep: Nos indica la profundidad del subarbol en el que estamos.

2.1.2. Algoritmos del árbol

El árbol AVL para mantener su balance posee 4 algoritmos.

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

Rotación simple a la izquierda

Este caso sucede si el árbol esta desequilibrado a la izquierda y su hijo derecho tiene el mismo signo (+)

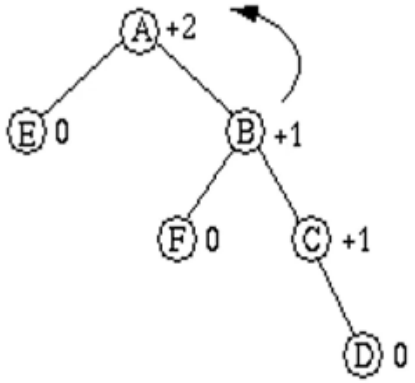


Figura 2.1: Árbol desequilibrado a la izquierda

Para esto el árbol se 'recorre' como si fuera una cadena hacia la izquierda quedando el nodo raiz como hijo izquierdo del que era su hijo derecho si este hijo derecho tenia un hijo izquierdo entonces se enlaza como hijo derecho de la raíz recorrida:

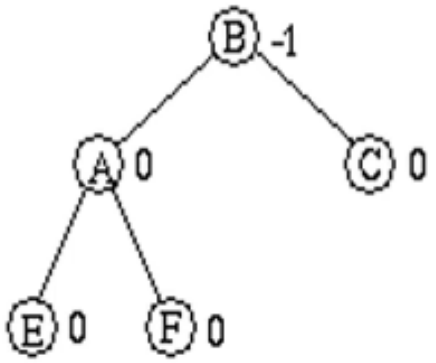


Figura 2.2: Árbol después de la rotación simple izquierda

Rotación simple a la derecha

Este caso sucede si el árbol esta desequilibrado a la derecha y su hijo izquierdo tiene el mismo signo(-)

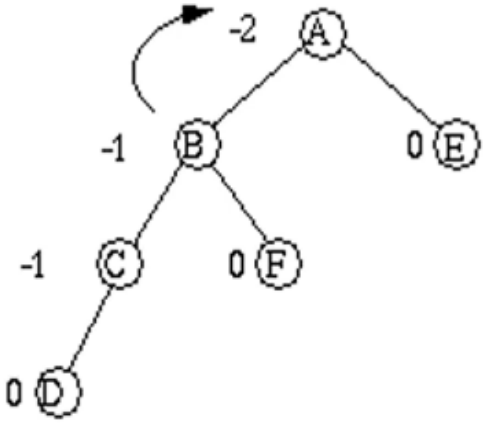


Figura 2.3: Arbol desequilibrado a la derecha

Para esto el árbol se 'recorre' como si fuera una cadena hacia la derecha quedando el nodo raíz como hijo derecho del que era su hijo izquierdo si este hijo izquierdo tenia un hijo derecho entonces se enlaza como hijo izquierdo de la raíz recorrida:

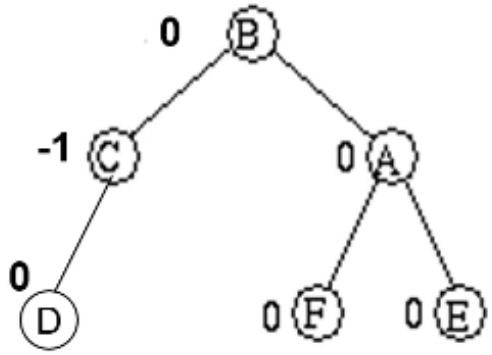


Figura 2.4: Árbol después de la rotación simple derecha

Rotación doble izquierda

Esta rotación es una combinación de las dos rotaciones simples solo que en un orden especifico; se da cuando el árbol esta desequilibrado a la izquierda (Factor de equilibrio menor que -1), y su hijo derecho tiene distinto signo (+) se realiza la rotación doble izquierda - derecha.

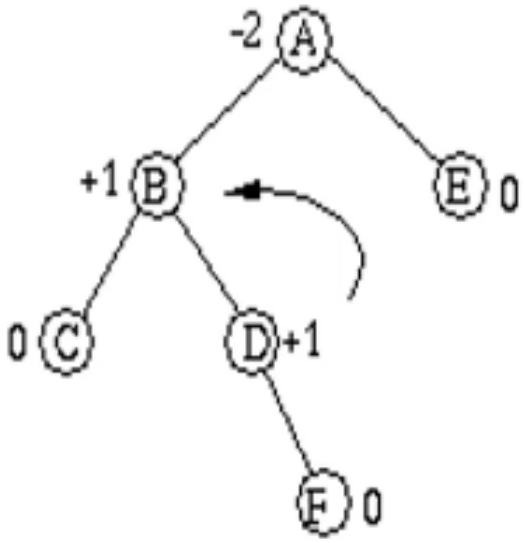


Figura 2.5: Árbol desequilibrado a la izquierda

En este caso se hace una rotación simple izquierda en el subárbol donde el signo cambia.

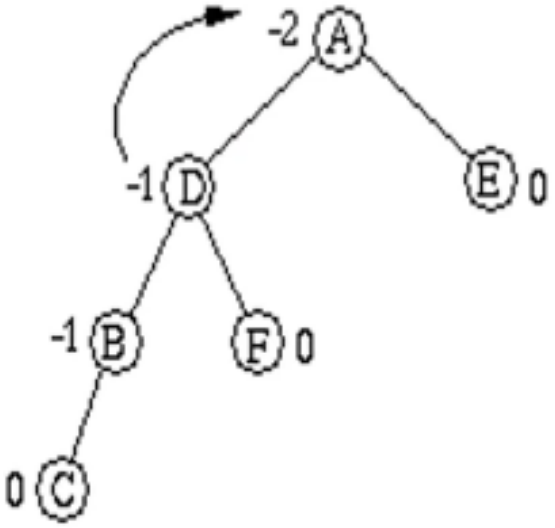


Figura 2.6: Árbol después de la rotación simple izquierda

Posteriormente se hace una rotación simple a la derecha en donde se encuentra el factor desequilibrado:

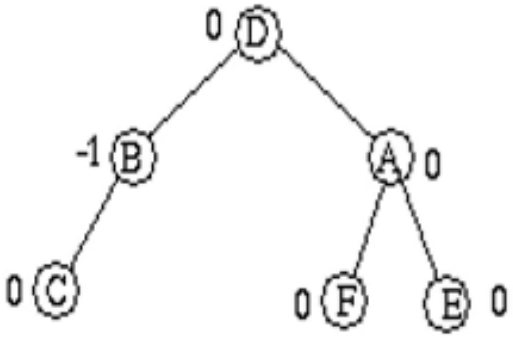


Figura 2.7: Árbol nuevamente equilibrado

Rotación doble derecha

Esta rotación es una combinación de las dos rotaciones simples solo que en un orden especifico; se da cuando el árbol esta desequilibrado a la derecha (Factor de equilibrio mayor que 1), y su hijo derecho tiene distinto signo (-) se realiza la rotación doble derecha - izquierda.

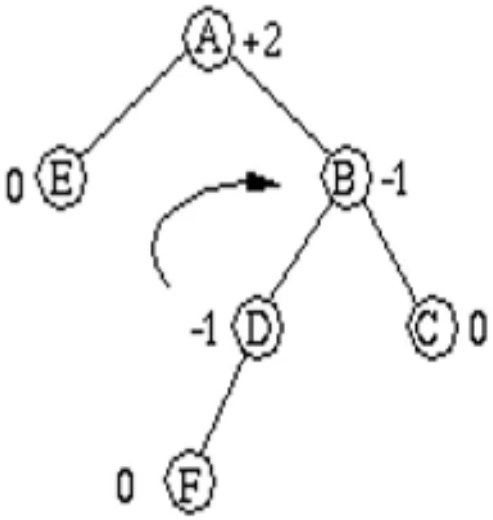


Figura 2.8: Árbol desequilibrado

En este caso se hace una rotación simple a la derecha en el subárbol donde el signo cambia:

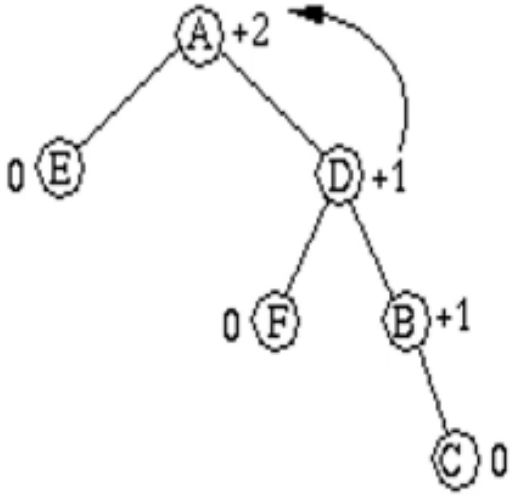


Figura 2.9: Árbol después de la rotación simple a la derecha

Y posteriormente una rotación simple a la izquierda en donde se encuentra el factor desequilibrado:

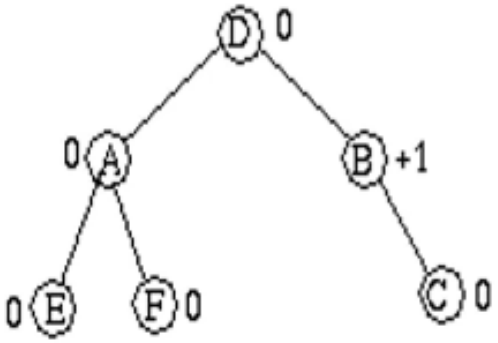


Figura 2.10: Árbol nuevamente equilibrado

Algoritmo de inserción

Al estar basado en un árbol binario, la inserción es en si la misma, es decir, si el nodo a insertar es mayor pasa al subárbol derecho o si es menor al subárbol izquierdo, así hasta encontrar su posición, una vez encontrado se inserta, se actualizan los factores de equilibrio en búsqueda de que el árbol este desequilibrado, si lo esta, se realiza alguno de los 4 algoritmos de balanceo, segun el caso en el que se encuentre.

Algoritmo de eliminación

Al estar basado en un árbol binario, la eliminación es en si la misma, es decir, si el nodo a eliminar es una hoja se elimina directamente, sino se intercambia con el nodo más a la izquierda del subárbol derecho o más a la derecha del subárbol izquierdo. se actualizan los factores de equilibrio en búsqueda de que el árbol este desequilibrado, si lo esta, se realiza alguno de los 4 algoritmos de balanceo, según el caso en el que se encuentre.

Algoritmo de búsqueda

Este es un simple DFS que según el valor a buscar va recorriendo el árbol, si es mayor al nodo en el que esta va al subárbol derecho sino al subárbol izquierdo.

2.1.3. Implementaciones

Una vez descritos los algoritmos y los objetos podemos hablar de sus implementaciones en el lenguaje Java, que terminan siendo más complicadas debido a su manejo de la memoria y a que los objetos se guían por referencias, pero si se realizan las clases según las especificaciones mencionadas no debería existir complicación alguna.

Clase Nodo

Esta clase solo contiene sus dos constructores y métodos de acceso.

```
1   public Nodo(AVL Padre, AVL Hizq, AVL Hder, int value, int eq, int deepizq, int
    deepder) {
2       this.Padre = Padre;
3       this.Hizq = Hizq;
4       this.Hder = Hder;
5       this.value = value;
6       this.eq = eq;
7       this.deepizq = deepizq;
8       this.deepder = deepder;
9       this.deep = 0;
10  }
11
12  public Nodo(int value){
13      this.value = value;
14  }
```

Figura 2.11: Constructores Clase Nodo

El primer constructor es para crear un nodo completo, es decir, un nodo al que se le pueden definir todos sus atributos, realmente su gran uso es para la creación del primer nodo. El segundo es para la creación de los nodos que se vayan añadiendo, pues los demás valores se van definiendo según el proceso de inserción.

Clase AVL

Esta clase contiene los siguientes constructores:

```
1     public AVL(Nodo Raiz) {
2         this.Raiz = Raiz;
3     }
4     public AVL(Nodo Raiz, boolean b){
5         this.Raiz = Raiz;
6         AVL.MainTree = this;
7         nv=0;
8     }
```

Figura 2.12: Constructores Clase AVL (Árbol AVL)

Son dos constructores porque la implementación se basa en un arbol que se divide en subárboles, por lo que el primer constructor es para la creación del primer nodo (nodo raiz) y el segundo constructor es para la construcción de los subárboles, es decir, arboles que si tienen padre y no son el Maintree o la raiz que origina todo. Se recibe un booleano únicamente para hacer uso de la sobrecarga de métodos y así tener disponible este constructor.

Método Equilibrar Este método recibe un Árbol AVL que es en si, el subárbol que esta desequilibrado y va a equilibrar.

```
1  public static void Equilibrar(AVL subtree){
2          if(subtree == null)
3          return;
4          System.out.println("A balancear:  " + subtree.getRaiz().getValue() + " " +
       subtree.getRaiz().getEq());
5          //subtree.bfs();
6          if(subtree.getRaiz().getEq()>1){
7                      if(subtree.getRaiz().getHder().getRaiz().getEq() >=0){
8                          AVL.RSI(subtree);
9                      }else{
10                         AVL.RDD(subtree);
11                     }
12                 }else if(subtree.getRaiz().getEq()<-1){
13                     if(subtree.getRaiz().getHizq().getRaiz().getEq() <=0){
14                         AVL.RSD(subtree);
15                     }else{
16                         AVL.RDI(subtree);
17                     }
18
19                 }
20             //actualizarAlturaTD(MainTree.Raiz);
21     }
```

Figura 2.13: Método Equilibrar

Si recibe un subárbol null pues realmente no hay nada que equilibrar, esto es más que nada para evitar ciclos y errores. Se imprime el nodo que se va a balancear y el valor para reconocer si es un menor que 2 o un mayor que 2, esto se imprime en la consola para proveer información al usuario de que se esta haciendo o que nodos se van a balancear. Finalmente cada condicional describe un caso de rotación, la linea 6 describe los casos de rotación simple izquierda y rotación simple derecha, el condicional anidado dice que si el nodo y su hijo derecho son del mismo signo (factor de equilibrio) entonces el una rotación simple izquierda sino una rotación doble derecha. El else if describe los casos de rotación simple derecha y rotación doble izquierda, en donde, si el nodo y su hijo izquierdo son del mismo signo (factor de equilibrio) entonces se da una rotacion simple derecha sino sea da una rotacion doble izquierda.

Método ActualizarAlturaTD Este método es el encargado de actualizar las alturas de los arboles, para si conseguir el factor de equilibrio, su manera de hacerlo es bottom up pues recorre el árbol desde la raíz y va hasta el fondo para posteriormente ir actualizando los datos de las alturas desde el ultimo nodo hasta el ultimo metodo de la Callstack actualizando asi las alturas correspondientes de los nodos, por lo que en cada eliminación, inserción y equilibrio se actualizan sus valores de profundidad y de factor de equilibrio.

```
1   public static void actualizarAlturaTD(Nodo nodo) {
2       if (nodo == null) {
3           return;
4       }
5       if(nodo.Hizq!=null)
6       actualizarAlturaTD(nodo.Hizq.getRaiz());
7       if(nodo.Hder != null)
8       actualizarAlturaTD(nodo.Hder.getRaiz());
9
10      int alturaIzquierda = (nodo.Hizq != null) ? nodo.Hizq.getRaiz().deep : 0;
11      int alturaDerecha = (nodo.Hder != null) ? nodo.Hder.getRaiz().deep : 0;
12      nodo.deep = 1 + Math.max(alturaIzquierda, alturaDerecha);
13      nodo.eq = alturaDerecha - alturaIzquierda;
14      nodo.deepder = alturaDerecha;
15      nodo.deepizq = alturaIzquierda;
16  }
```

Figura 2.14: Método actualizarAlturaTD

Cuando se llega a un nodo null, o un nodo hoja pues se deja la llamada recursiva y se van calculando las demas alturas según la callstack, es decir, a través del retroceso recursivo. Las lineas 10 y 11 son ternarias que nos sirven para extraer la profundidad, si el nodo tiene hijo izquierdo entonces se actualiza altura izquierda con la profundidad del subárbol izquierdo, si no hay nodo izquierdo significa que la profundidad es cero. La linea 11 es lo mismo pero para el subárbol derecho. Finalmente la altura del nodo es 1 más el máximo entre la profundidad del subárbol izquierdo y el derecho. El factor de equilibrio es la profundidad del árbol derecho menos la profundidad del árbol izquierdo y lo demás simplemente es rellenar los valores para el siguiente retroceso recursivo.

Rotación Simple a la Izquierda Este método es el que se encarga de realizar el algoritmo de Rotación simple a la izquierda.

```
1   public static void RSI(AVL subtree){
2       if(subtree == null)
3       return;
4
5       Nodo aux2 = subtree.Raiz;
6       AVL aux = null;
7       AVL Padre = null;
8       if(aux2.Hder.getRaiz().Hizq!= null)
9       aux = aux2.Hder.getRaiz().Hizq;
10
11      if(subtree.getRaiz().value == MainTree.getRaiz().value){
12          MainTree = subtree.getRaiz().getHder();
13      }else{
14          Padre = subtree.getRaiz().Padre;
15      }
16
17      subtree.setRaiz(subtree.getRaiz().getHder().getRaiz());
18      subtree.Raiz.Hizq = new AVL(aux2);
19      subtree.Raiz.Hizq.getRaiz().Padre = subtree;
20
21      if(subtree.getRaiz().value == MainTree.getRaiz().value){
22          subtree.getRaiz().Padre = null;
23      }else{
24          subtree.getRaiz().Padre = Padre;
25      }
26      subtree.Raiz.Hizq.Raiz.Hder = null;
27      if(subtree.Raiz.Hizq.Raiz.Hizq!=null)
28      subtree.Raiz.Hizq.Raiz.Hizq.Raiz.Padre = subtree.Raiz.Hizq;
29      if(aux != null)
30          subtree.Raiz.Hizq.insertar(aux.getRaiz());
31      actualizarAlturaTD(MainTree.getRaiz());
32  }
```

Figura 2.15: Método RSI

La rotación simple a la izquierda, inicia extrayendo la raíz del subárbol que va a equilibrar y declara dos subárboles y un nodo que serán auxiliares para proceder correctamente con el recorrido; si el subárbol que se va a mover a la izquierda tiene hijo izquierdo entonces este se guarda en una variable auxiliar, sino entonces la variable auxiliar queda nula; ahora, si el árbol que se va a equilibrar es el árbol Main, es decir, la raíz que lleva a todos los nodos entonces en este se define el que sera la nueva, raíz, sino esto significa que el subárbol a equilibrar tiene padre, entonces se guarda en la variable auxiliar Padre. Una vez esto se procede con el reacomodo, el árbol que se esta equilibrando ahora toma el valor de su hijo derecho a este arbol actualizado se le denominara 'nuevo arbol', el hijo izquierdo ahora pasa a tener el nodo que era del arbol original y el padre de ese subtree izquierdo ahora pasa a ser el nuevo subtree. Si el nuevo árbol es el a su vez el MainTree entonces se asigna su padre como null, sino, se le asigna el padre almacenado en la variable auxiliar. Ahora el árbol que paso a la izquierda se le asigna su hijo derecho como null y si este mismo tiene un hijo izquierdo entonces a ese hijo izquierdo se le asigna como padre el nodo que se recorrió. Retomando

la variable auxiliar que llenamos con el primer condicional, si no es nula, si ese hijo si existía, ahora se la encadenaremos al subárbol que se recorrió a la izquierda, es decir, a la raíz que paso a ser hijo izquierdo de la nueva raíz, se llama al método insertar(que explicaremos posteriormente) para que se inserte en la mejor posición disponible en ese subárbol. Finalmente se reacomoda la altura del árbol.

Rotación Simple a la Derecha Este método es el que se encarga de realizar el algoritmo de Rotación simple a la derecha.

```
1  public static void RSD(AVL subtree){
2         if(subtree == null)
3         return;
4
5         Nodo aux2 = subtree.Raiz;
6         AVL aux = null;
7         AVL Padre = null;
8         if(aux2.Hizq.getRaiz().Hder!= null)
9         aux = aux2.Hizq.getRaiz().Hder;
10
11        if(subtree.getRaiz().value == MainTree.getRaiz().value){
12            MainTree = subtree.getRaiz().getHizq();
13        }else{
14            Padre = subtree.getRaiz().Padre;
15        }
16
17        subtree.setRaiz(subtree.getRaiz().getHizq().getRaiz());
18        subtree.Raiz.Hder = new AVL(aux2);
19        subtree.Raiz.Hder.getRaiz().Padre = subtree;
20        if(subtree.getRaiz().value == MainTree.getRaiz().value){
21            subtree.getRaiz().Padre = null;
22        }else{
23            subtree.getRaiz().Padre = Padre;
24
25        }
26        subtree.Raiz.Hder.Raiz.Hizq =null;
27        if(subtree.Raiz.Hder.Raiz.Hder!=null)
28        subtree.Raiz.Hder.Raiz.Hder.Raiz.Padre = subtree.Raiz.Hder;
29        if(aux != null)
30            subtree.Raiz.Hder.insertar(aux.getRaiz());
31        actualizarAlturaTD(MainTree.getRaiz());
32    }
```

Figura 2.16: Método RSD

La rotación simple a la derecha es muy similar a la rotación simple izquierda pero se invierten los lados, prácticamente es lo mismo solo que cambiando los hijos izquierdos por derechos y los derechos por los izquierdos. Se empieza igual, 3 variables auxiliares, un Nodo y dos arboles que nos ayudaran para la correcta aplicación del algoritmo. Si el hijo derecho del hijo izquierdo del subárbol a equilibrar existe, este se guarda en la variable auxiliar para su posterior manejo. Si el árbol que se equilibra es el árbol Main, entonces se asigna el nuevo Maintree, sino el padre se asigna a la variable auxiliar Padre.

Ahora comienza el recorrido, el árbol ahora es el que era su hijo izquierdo y el hijo derecho de este árbol que se recorrió pasa a ser el que era el árbol a equilibrar. Este árbol que pasa a ser la raiz le denominaremos 'nuevo arbol' Se comprueba nuevamente si el nuevo árbol es el maintree, si lo es entonces su padre pasa a ser nulo sino, pasa a ser el padre almacenado en la variable auxiliar. Ahora el hijo izquierdo del hijo derecho del nuevo árbol sera null y si su hermano derecho no es null entonces su padre se asigna como el hijo derecho del nuevo árbol, esto es más que nada para que las variables tengan bien asignados las nuevas referencias que poseen. Si la variable auxiliar asignada en el primer condicional no es null significa que tenemos que reenlazarla en el hijo derecho del nuevo arbol. se llama al método insertar(que explicaremos posteriormente) para que se inserte en la mejor posición disponible en ese subárbol. Finalmente se reacomoda la altura del árbol.

Rotación Doble Izquierda Este método es el que se encarga de realizar el algoritmo de Rotación Doble izquierda.

```
1   public static void RDI(AVL subtree){
2        if(subtree ==null) return;
3        RSI(subtree.Raiz.Hizq);
4        RSD(subtree);
5
6    }
```

Figura 2.17: Método RDI

Como mencionamos,las rotaciones dobles son combinación de las rotaciones simples, por lo que en la rotación doble izquierda, primero se reacomoda el hijo donde cambia el signo (hijo izquierdo), posteriormente se reacomoda el arbol original con la rotacion simple derecha.

Rotación Doble Derecha Este método es el que se encarga de realizar el algoritmo de Rotación Doble Derecha.

```
1    public static void RDD(AVL subtree){
2        RSD(subtree.Raiz.Hder);
3        RSI(subtree);
4    }
```

Figura 2.18: Método RDD

La rotación doble derecha es lo mismo, una combinación de ambas pero ahora cambia el orden, se reacomoda el hijo donde cambia el signo con la rotación simple derecha y posteriormente se reacomoda el arbol original con la rotación simple izquierda.

Métodos de búsqueda de desequilibrio(dfsizqAVL y dfsderAVL) Para encontrar desequilibrio en alguno de los nodos se tienen dos dfs uno que recorre el árbol por los subárboles izquierdos y otro por los subárboles derechos. Procederemos a explicar la implementación de uno porque realmente funcionan igual solo cambian los subárboles que recorren.

```
1
2     public AVL dfsizqAVL(){
3         if(this.Raiz.Hizq != null)
4             if( this.getRaiz().Hizq.getRaiz().getEq() <2 && this.getRaiz().Hizq.
    getRaiz().getEq() >1 ||  this.getRaiz().Hizq.getRaiz().getEq() <2 && this.
    getRaiz().Hizq.getRaiz().getEq() <-1)
5                 return this.Raiz.Hizq.dfsizqAVL();
6         if(this.Raiz.Hder !=null)
7             if(this.getRaiz().Hder.getRaiz().getEq() <2 && this.getRaiz().Hder.
    getRaiz().getEq() >1 || this.getRaiz().Hder.getRaiz().getEq() <2 && this.getRaiz
    ().Hder.getRaiz().getEq() <-1)
8                 return this.Raiz.Hder.dfsderAVL();
9         if(this.Raiz.eq <=-2 || this.Raiz.eq >=2 )
10        return this;
11        if(this.Raiz.eq <=-2){
12            if(this.Raiz.Padre!=null)
13                return this.Raiz.Padre;
14            return MainTree;
15        }else{
16            if(this.Raiz.Hizq!= null){
17                return this.Raiz.Hizq.dfsizqAVL();
18            }
19        }
20        return null;
21    }
```

Figura 2.19: Método dfsizqAVL

El dfs primero procede a reconocer que si existe el subárbol que va a recorrer si existe y esta equilibrado procede a buscar desequilibrio en los subárboles que siguen, asi hasta llegar al fondo o que encuentre alguno que este desequilibrado, cuando encuentra uno desequilibrado retorna este árbol. En caso que no encuentre algún árbol desequilibrado entonces retorna null, indicando que el árbol sigue en equilibrio.

Inserción Una vez explicados los métodos esenciales para el balanceo del árbol y que nos ayudan a mantener su estructura podemos proceder a el método de inserción.

```
    public boolean insertar(Nodo node){
        if(node.value> this.Raiz.value){
            if(this.Raiz.Hder==null){
                this.Raiz.Hder = new AVL(node);
                this.Raiz.Hder.Raiz.Padre= this;
                AVL.nv++;
                actualizarAlturaTD(MainTree.getRaiz());
                AVL aux =MainTree.dfsderAVL();
                if(aux ==null){
                    AVL.Equilibrar(MainTree.dfsizqAVL());
                }else{
                    AVL.Equilibrar(aux);
                }

                return true;
            }
            return this.Raiz.Hder.insertar(node);
        }else if(node.value < this.Raiz.value){
            if(this.Raiz.Hizq==null){
                this.Raiz.Hizq = new AVL(node);
                this.Raiz.Hizq.Raiz.Padre= this;
                AVL.nv++;
                actualizarAlturaTD(MainTree.getRaiz());
                AVL aux =MainTree.dfsderAVL();
                if(aux ==null){
                    AVL.Equilibrar(MainTree.dfsizqAVL());
                }else{
                    AVL.Equilibrar(aux);
                }
                return true;
            }
            return this.Raiz.Hizq.insertar(node);
        }
        return false;
    }
    }
```

Figura 2.20: Método Insertar

El método de inserción funciona igual que el del árbol binario, se mueve por los nodos hasta encontrar su posición ideal, lo que cambia es que cada inserción se actualizan los valores del numero de vértices (la variable estática) y los factores de equilibrio de cada nodo, al no existir apuntadores, estos factores de equilibrio no se actualizan automáticamente y se tiene que hacer con la función ya explicada anteriormente.Ahora se usa el DFS derecho desde el MainTree en busca de desequilibrio, el subárbol desequilibrado se guarda en una variable auxiliar, si resulta null significa que en ese lado no hay desequilibrio y se realiza un dfs para el lado izquierdo; se pasa directamente a la función equilibrio porque como ya se hizo el otro dfs solo queda esta ultima comparación, el método equilibrar ya sabe como lidiar en caso de que también esa rama ya este en equilibrio o no; sino, entonces se llama a la función que equilibra el subárbol desequilibrado que encontró el dfs izquierdo, finalmente retorna true porque se agrego correctamente. en caso de que se inserta del lado izquierdo procede el mismo procedimiento. Retorna false únicamente si el nodo ya se encuentra en el árbol.

Eliminación El método de eliminación es realmente igual al de un árbol binario, únicamente que se llama a la función de equilibrio para que el árbol recupere su balance, para describirlo lo iremos seccionando en los casos que pueden existir.

```
public boolean Delete(Nodo nodo){

      if(nodo == null){
          System.out.println("No exite el nodo en el arbol");
          return false;
      }

      //Es una hoja?
      if(nodo.Hder==null && nodo.Hizq == null){
          //probablemente, esta hoja es la raiz?
          if(nodo.Padre == null){
              AVL.nv =0;
              AVL.MainTree =null;
              this.Raiz =null;
          }else{
              if(nodo.Padre.Raiz.Hder!=null && nodo.Padre.Raiz.Hizq !=null)
                  if(nodo.Padre.Raiz.Hder.Raiz.value == nodo.value){
                      nodo.Padre.Raiz.Hder=null;
                  }else{
                      nodo.Padre.Raiz.Hizq = null;
              }else if(nodo.Padre.Raiz.Hder==null)
                  nodo.Padre.Raiz.Hizq = null;
              else if(nodo.Padre.Raiz.Hizq==null)
                  nodo.Padre.Raiz.Hder = null;
          }
          AVL.actualizarAlturaTD(MainTree.Raiz);
              AVL aux2 =MainTree.dfsderAVL();
              if(aux2==null){
                  AVL.Equilibrar(MainTree.dfsizqAVL());
              }else{
                  AVL.Equilibrar(aux2);
              }
          return true;
      }
```

Figura 2.21: Método eliminación - ¿El nodo a eliminar es una hoja?

Si el nodo es null significa que no existe el nodo a eliminar retornando false. Ahora en esta parte del código se maneja el caso en el que el nodo sea una hoja, si lo es, se verifica si esta hoja es el maintree, osea que sea un árbol de un solo nodo, si lo es, entonces el árbol se 'reinicia', si no entonces se procede a eliminar directamente; esto despliega otros casos, que tenga un hermano o que no lo tenga, si tiene hermano entonces se procede a comprobar cual de los hermanos es el nodo a eliminar para eliminarlo directo desde la referencia del padre, si no tiene hermano entonces se busca cual de los nodos no es el nulo (que sera el nodo a eliminar) y se elimina directamente desde la referencia del padre. Ahora se actualizan los factores de equilibrio y se realizan los dfs para encontrar algún árbol desequilibrado, en caso de que si, se equilibra con las funciones de equilibrio.

Ahora pasaremos a explicar el caso en que no exista alguno de los subárboles es nulo.

```
        if(nodo.Hder == null){
            if(nodo.Padre ==null){
                AVL.MainTree = nodo.Hizq;
                AVL.MainTree.Raiz.Padre = null;
            }else{
                nodo.Hizq.Raiz.Padre = nodo.Padre;
                nodo.Padre.Raiz.Hizq = nodo.Hizq;
            }
            AVL.actualizarAlturaTD(MainTree.Raiz);
            AVL aux2 =MainTree.dfsderAVL();
                if(aux2==null){
                    AVL.Equilibrar(MainTree.dfsizqAVL());
                }else{
                    AVL.Equilibrar(aux2);
                }
            return true;

        }
```

Figura 2.22: Método eliminación - El nodo solo posee un subárbol

Si el nodo solo posee un subárbol entonces se busca que subárbol es nulo, en el ejemplo mostrado el subarbol nulo es el derecho, una vez idetificado, se verifica si el árbol a eliminar es el MainTree o no, esto se hace sabiendo si su padre es null o no, si lo es pues se actualiza el Maintree como si se recorriera el árbol hacia la derecha, pues como a su derecha no hay nada entonces es como si solo se recorrieran los nodos ocupando el lugar vació. En caso de que no sea el MainTree pues sucede el mismo proceso de recorrido únicamente que el padre se actualiza por el padre que tenia el nodo a borrar. Para el caso que el nodo izquierdo sea null se trata de la misma manera solo que ahora se 'recorre' hacia el otro lado.

Ahora pasaremos a explicar el caso en que no es una hoja y se tiene ambos subarboles.

```
        Nodo intecambar = null;
        intecambar = nodo.Hder.getRaiz();
        while (intecambar.Hizq!=null) {
            intecambar = intecambar.Hizq.getRaiz();
        }
        int aux;
        aux = nodo.value;
        nodo.value = intecambar.value;
        intecambar.value= aux;
        //System.out.println("Borrando: " + intecambar.value);
        this.Delete(intecambar);
            AVL aux2 =MainTree.dfsderAVL();
                if(aux2==null){
                    AVL.Equilibrar(MainTree.dfsizqAVL());
                }else{
                    AVL.Equilibrar(aux2);
                }
                return true;
    }
```

Figura 2.23: Método eliminación - El posee nodo ambos subárboles

Si el nodo posee ambos subárboles el algoritmo busca el nodo más a la izquierda del hijo derecho, intercambia los valores y lo elimina como si fuera una hoja llamando de nuevo a la función delete.

Cabe destacar que solo retorna false en caso de que el nodo recibido sea nulo, pero esto es porque se complementa con la función search que se explicara posteriormente. También algo a mencionar es que en cada caso sucede la actualización de alturas y posteriormente de equilibrio.

Método search Este es el método de búsqueda, se hace con un DFS para aprovechar las propiedades del Árbol AVL.

```
1         public Nodo search(Nodo ToSearch){
2         if (this.Raiz == null) {
3             return null;
4         }
5         if(this.Raiz.value == ToSearch.value){
6             return this.Raiz;
7         }else if(ToSearch.value > this.Raiz.value && this.Raiz.Hder!=null){
8
9                 return this.Raiz.Hder.search(ToSearch );
10        }else if(ToSearch.value < this.Raiz.value && this.Raiz.Hizq!=null ){
11                return this.Raiz.Hizq.search(ToSearch);
12        }
13        return null;
14    }
```

Figura 2.24: Método de Búsqueda

Este método es un DFS simple que compara los valores del nodo a buscar como el nodo en el que se encuentra, si es mayor pues prosigue al subárbol derecho o si es menor procede al subárbol izquierdo, si es igual retorna el nodo, esto es porque se complementa con el método de eliminación y así acceder a la referencia del nodo exacto a buscar, además nos sirve para así obtener más información acerca del nodo y pues si el nodo no se encuentra pues entonces retorna un nodo null.

Método BFS (para impresión) Este es el método de impresión, se hace con un BFS para imprimirlo por capaz.

```
public void bfs(){
        if (this.Raiz == null) {
            return;
        }
        Queue<Nodo> cola = new LinkedList<>();
        cola.add(this.Raiz);
        while (!cola.isEmpty()) {
            int nodosEnNivel = cola.size();
            for(int i = 0; i< nodosEnNivel; i++){
                Nodo actual = cola.poll();
                /* */
                System.out.print("||Node: " + actual.value  + " Eq: " + actual.eq);
                if(actual.Padre != null)
                    System.out.print(" Father: " + actual.Padre.Raiz.getValue());
                if(actual.Hder != null)
                    System.out.print(" Hder: " + actual.Hder.getRaiz().getValue());
                if(actual.Hizq != null)
                    System.out.print(" Hizq: " + actual.Hizq.getRaiz().getValue());
                System.out.print("||");
                //System.out.print( (char) actual.value + " ");
                if (actual.Hizq != null) {
                cola.add(actual.Hizq.getRaiz());
                }
                if (actual.Hder != null) {
                cola.add(actual.Hder.getRaiz());
                }
            }
            System.out.println();
        }
    }
```

Figura 2.25: Método BFS

Este método es un simple BFS, es la implementación más básica, solo que se manejan las salidas para que muestre información como el padre, los hijos y los factores de equilibrio, el BFS se usa para imprimir en el CMD pero con la interfaz gráfica implementada queda mejor y no hace uso de esto.

2.1.4. Ejecución del programa:

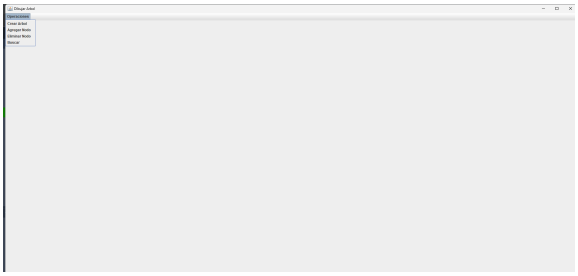


Figura 2.26: Interfaz y operaciones

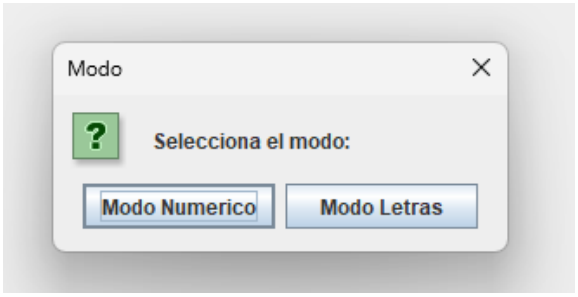


Figura 2.27: Selección de modo

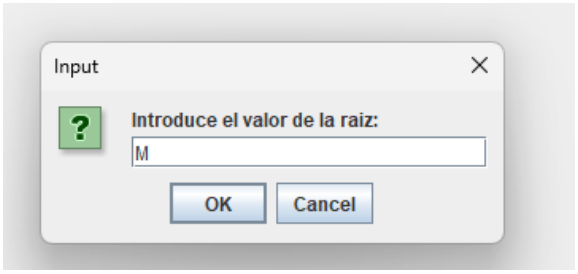


Figura 2.28: Introduccion de letra M

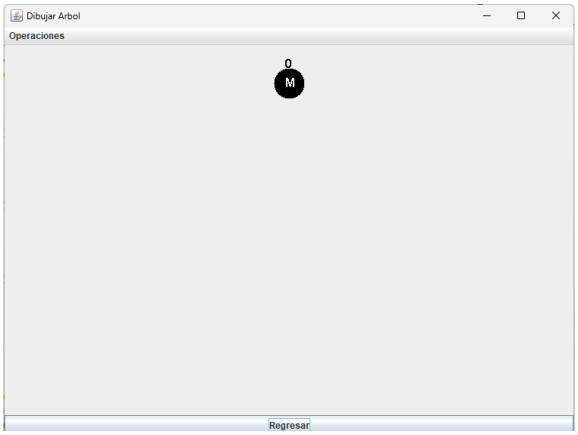


Figura 2.29: Nodo creado

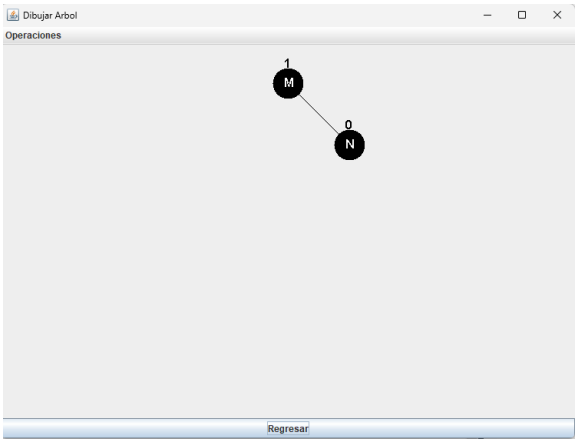


Figura 2.30: Añadiendo N

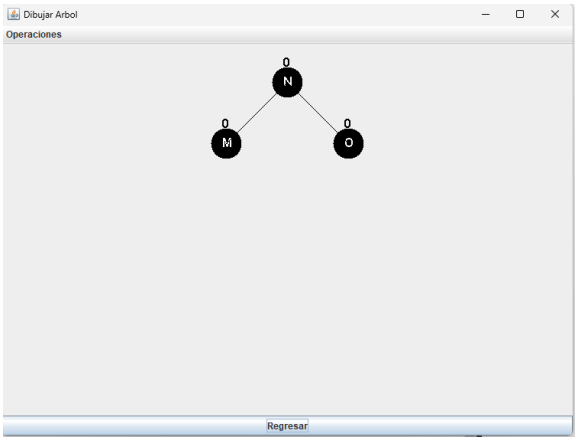


Figura 2.31: Añadiendo O, sucede rotación simple izquierda

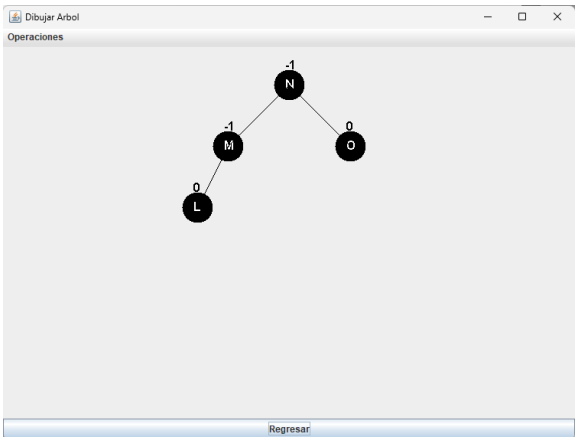


Figura 2.32: Se añade L

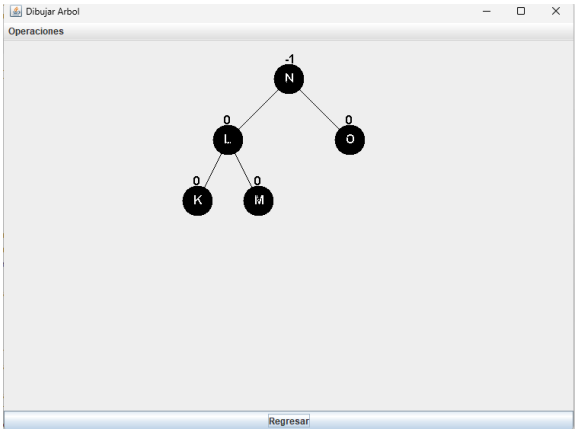


Figura 2.33: Se añade K, sucede rotacion simple a la derecha

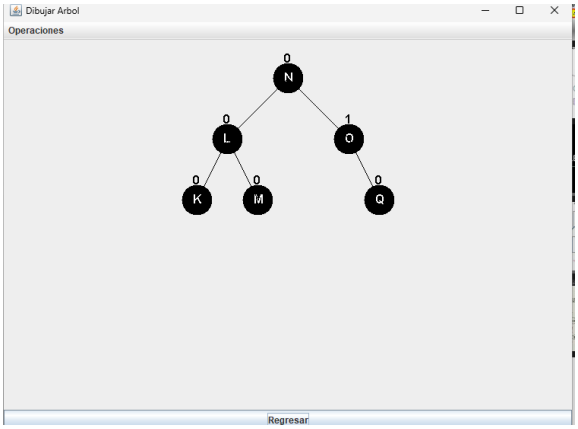


Figura 2.34: Se añade Q

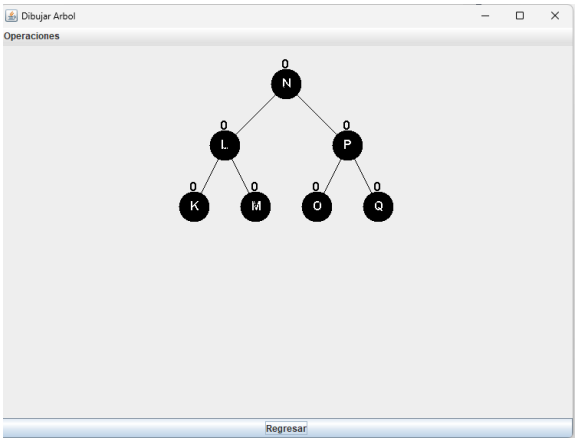


Figura 2.35: Se añade P, sucede rotación doble derecha
;

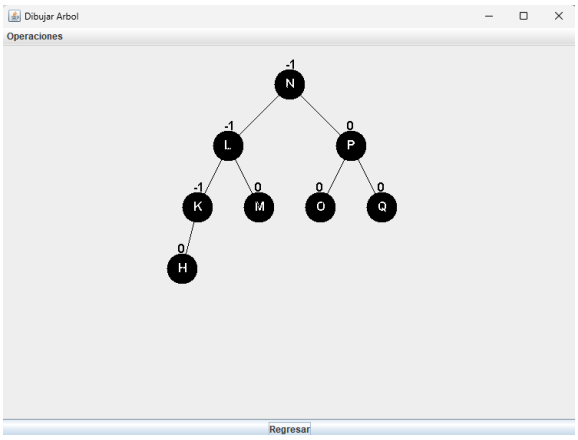


Figura 2.36: Se añade H

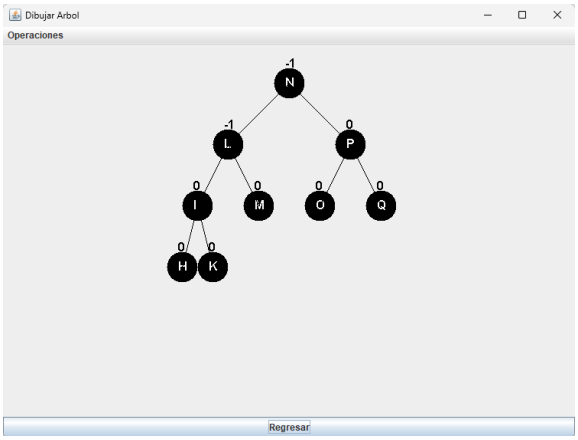


Figura 2.37: Se añade I, sucede rotación doble izquierda

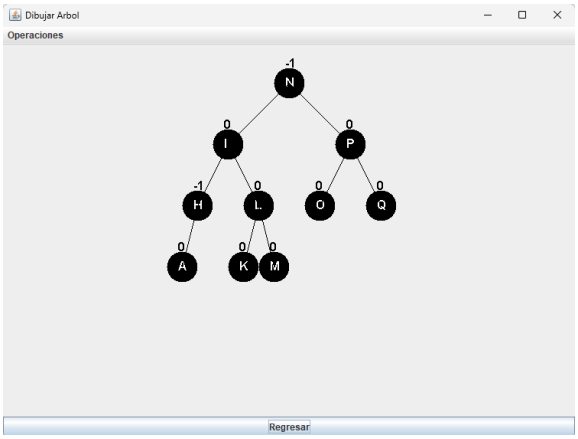


Figura 2.38: Se añade A, sucede una rotación simple a la derecha

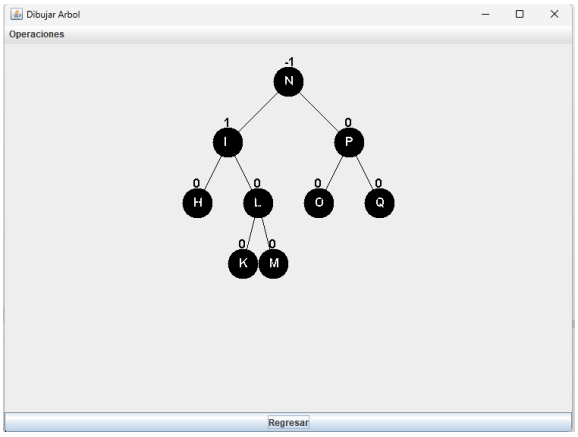


Figura 2.39: Eliminación A (Hoja)

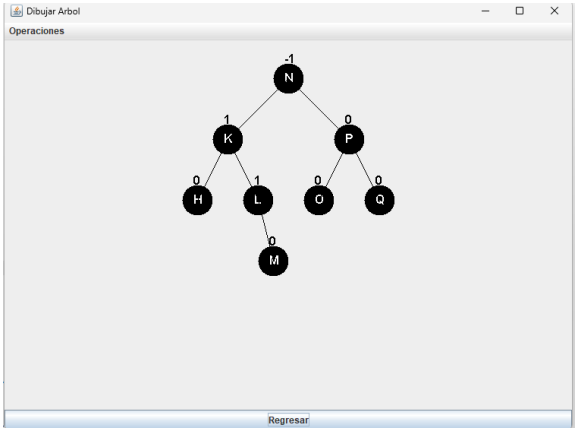


Figura 2.40: Eliminación I (Nodo intermedio)

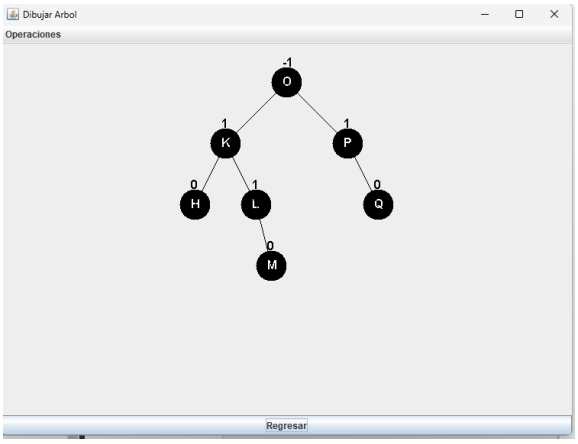


Figura 2.41: Eliminación N (Raíz)

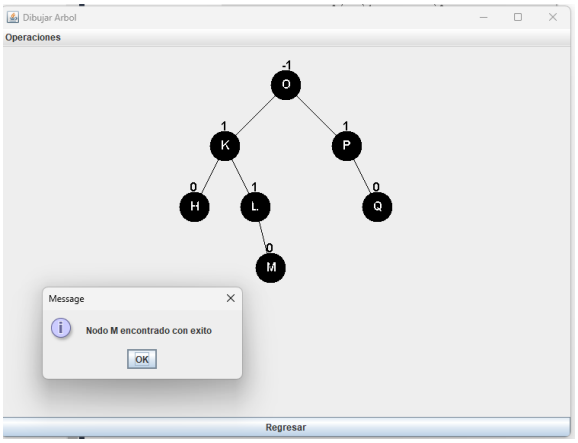


Figura 2.42: Búsqueda M

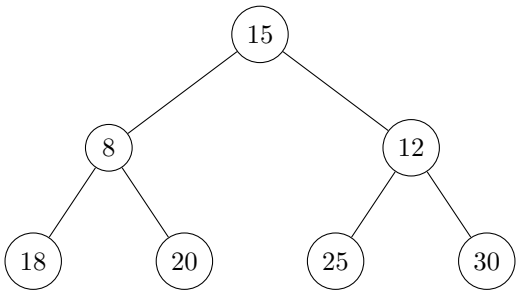
2.2. Heap

2.2.1. Heapify

Heapify es una operación fundamental en la manipulación de Heaps. Se aplica a un nodo de un Heap para mantener la propiedad del Heap cuando el nodo viola esta propiedad, es decir, cuando sus hijos cumplen con la propiedad del Heap, pero el nodo en sí mismo no. La operación Heapify juega un papel crucial en la construcción y manipulación eficiente de Heaps.

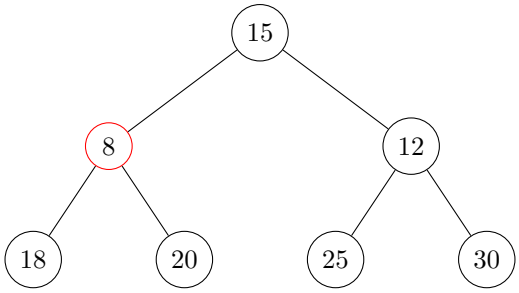
Operación Heapify

Consideremos el siguiente árbol que no cumple con la propiedad del Min Heap:

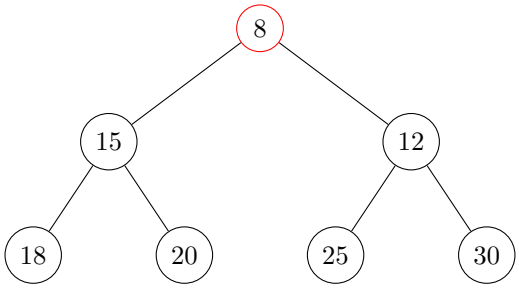


El nodo con el valor 8 no cumple con la propiedad del Min Heap ya que es menor que sus hijos. Aplicamos Heapify al nodo 8 para corregir esta violación de la propiedad.

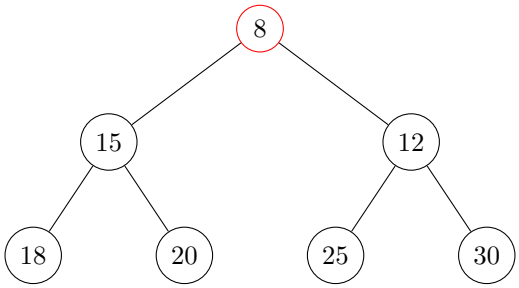
Paso 1: Identificación de la Violación Comparamos el nodo con sus hijos y encontramos que el hijo izquierdo (15) tiene un valor menor.



Paso 2: Intercambio de Valores Intercambiamos el valor del nodo con el valor del hijo más pequeño (15).



Paso 3: Recursión Aplicamos Heapify recursivamente en el subárbol afectado (nodo 8).



Después de aplicar Heapify, el árbol vuelve a cumplir con la propiedad del Min Heap. Este proceso se puede repetir en otros nodos para mantener la integridad de la estructura del Heap.

Análisis al Subdividir el Árbol

Al aplicar Heapify al nodo 8, hemos subdividido el árbol en subárboles y hemos analizado cada paso. Aquí hay algunos casos que podrían ocurrir durante el proceso:

- **Caso 1:** El nodo tiene dos hijos y uno de ellos es menor. Se intercambian los valores y se aplica Heapify en el subárbol afectado.
- **Caso 2:** El nodo tiene solo un hijo. Se intercambian los valores y no es necesario seguir con la recursión en el subárbol.
- **Caso 3:** El nodo no tiene hijos. No hay violación, y la operación Heapify termina en ese nodo.

Estos casos ayudan a entender cómo Heapify opera y mantiene la propiedad del Heap en diferentes situaciones. Este proceso se puede repetir en otros nodos para mantener la integridad de la estructura del Heap.

Análisis de la Operación Heapify

La operación Heapify se aplica en nodos cuyos hijos cumplen con la propiedad del Heap, pero el nodo en sí mismo no. Su complejidad se analiza en términos del tamaño del árbol y se realiza en tiempo $O(\log n)$.

Peor caso El peor caso ocurre cuando el último nivel del árbol está medio lleno. Esto maximiza el número de nodos en el subárbol afectado.

Optimización Aunque la complejidad es $O(\log n)$, se puede optimizar en la práctica para operaciones de construcción y manipulación eficientes de Heaps.

En las secciones siguientes, exploraremos más operaciones y aplicaciones de los Heaps.

2.2.2. Aplicaciones Prácticas de Heaps

Las estructuras de Heap encuentran aplicaciones prácticas en una variedad de escenarios en ciencia de la computación y programación. A continuación, se presentan algunas de las aplicaciones más comunes:

Colas de Prioridad

Las colas de prioridad son una aplicación clásica de los Heaps. Al utilizar un Min Heap o un Max Heap, es posible implementar una cola de prioridad eficiente para gestionar elementos en función de sus valores prioritarios. Esto resulta útil en algoritmos que requieren acceso rápido al elemento más prioritario.

Ordenamiento Eficiente

Heapsort es un algoritmo de ordenamiento basado en Heaps que aprovecha las propiedades de los Max Heaps o Min Heaps. Aunque no es tan eficiente como otros algoritmos de ordenamiento en muchos casos, Heapsort tiene la ventaja de ser in situ y tiene una complejidad de tiempo $O(n \log n)$.

Algoritmos de Grafos

En algoritmos de grafos, los Heaps son fundamentales para implementar algoritmos como Dijkstra y el algoritmo de Prim para encontrar caminos más cortos y árboles de expansión mínima, respectivamente. La prioridad en la selección de nodos se gestiona eficientemente utilizando colas de prioridad basadas en Heaps.

Programación Dinámica

En algunos problemas de programación dinámica, los Heaps se utilizan para gestionar conjuntos de soluciones parciales o para seleccionar la mejor solución entre varias alternativas. La capacidad de acceso rápido al elemento máximo o mínimo hace que los Heaps sean útiles en estos contextos.

Manejo de Eventos en Simulaciones

En simulaciones y sistemas de eventos discretos, los Heaps se pueden utilizar para gestionar eventos según sus tiempos de ocurrencia. Esto permite una ejecución eficiente de eventos en el orden correcto y es esencial en el diseño de simulaciones precisas.

Optimización de Algoritmos Greedy

Los algoritmos greedy a menudo toman decisiones en función de la información disponible en el momento. Los Heaps proporcionan una forma eficiente de acceder al elemento más prioritario, lo que es crucial en algoritmos greedy para tomar decisiones óptimas en cada paso.

Estas son solo algunas de las aplicaciones prácticas de las estructuras de Heap. Su versatilidad y eficiencia hacen que sean herramientas valiosas en diversos problemas y algoritmos en ciencia de la computación.

2.2.3. Implementación: Clase Min_Heap

(Nota*: Esta implementación es distinta a la que se usa en el programa, pero por cuestión de simplicidad procederemos explicar esta). La implementación de la clase `Min_Heap` se presenta a continuación, detallando los métodos y sus funciones específicas.

`Min_Heap` - Constructor

Descripción: Inicializa un nuevo Min Heap.
Código:

```
public Min_Heap() {
    this.raiz = null;
}
```

`insertar` - Método de Inserción

Descripción: Inserta un nuevo valor en el Min Heap y realiza la operación de *bubble-up* para mantener la propiedad del heap.
Código:

```
public void insertar(int valor) {
    Nodo nuevoNodo = new Nodo(valor);

    if (raiz == null) {
        raiz = nuevoNodo;
    } else {
        Nodo nodoInsertado = insertarAlFinal(nuevoNodo);
        bubbleUp(nodoInsertado);
    }
}
```

`insertarAlFinal` - Método Auxiliar para Inserción

Descripción: Encuentra la posición para insertar el nuevo nodo al final del árbol. En un árbol completo, el llenado se realiza de izquierda a derecha. Se utiliza una cola para realizar un recorrido por niveles y encontrar el primer lugar disponible en el último nivel.
Código:

```
private Nodo insertarAlFinal(Nodo nuevoNodo) {
   Queue<Nodo> cola = new LinkedList<>();
        cola.add(raiz);
        while (!cola.isEmpty()) {
            Nodo nodoActual = cola.poll();

            if (nodoActual.getHijoIzquierdo() == null) {
                nodoActual.setHijoIzquierdo(nuevoNodo);
                nuevoNodo.setPadre(nodoActual);
                return nuevoNodo;
            } else if (nodoActual.getHijoDerecho() == null) {
                nodoActual.setHijoDerecho(nuevoNodo);
                nuevoNodo.setPadre(nodoActual);
                return nuevoNodo;
            }

            cola.add(nodoActual.getHijoIzquierdo());
            cola.add(nodoActual.getHijoDerecho());
        }
        return null;
}
```

`bubbleUp` **- Método Auxiliar para Restaurar la Propiedad del Heap**

Descripción: El método `bubbleUp` se encarga de mantener la propiedad del heap después de una inserción, moviendo un nodo recién insertado hacia arriba del árbol hasta que se restablece el orden del heap. Esta operación es crucial para garantizar que el nuevo nodo esté en la posición correcta en relación con sus padres, preservando la propiedad del heap.

Parámetros:

- `nodo`: Nodo del árbol que se ha insertado recientemente y que podría violar la propiedad del heap.

Funcionamiento: El método `bubbleUp` toma como entrada el nodo recién insertado y lo compara con su padre. Si el valor del nodo es menor (en el caso de un Min Heap) que el valor del padre, intercambia los nodos y repite el proceso hasta que se restablece la propiedad del heap o hasta que el nodo alcanza la raíz del árbol.

Código:

```
private void bubbleUp(Nodo nodo) {
    Nodo nodoActual = nodo;

    while (nodoActual.getPadre() != null && nodoActual.getValor() < nodoActual.getPadre().getValor()
        Nodo padre = nodoActual.getPadre();
        intercambiarNodos(nodoActual, padre);
        nodoActual = padre;
    }
}
```

`intercambiarNodos` - Método Auxiliar para Intercambiar Valores entre Nodos

Descripción: Este método se encarga de intercambiar los valores entre dos nodos de un árbol. Su función es esencial en diversas operaciones dentro de la clase `Min_Heap`, como en los procesos de *bubble-up* y *bubble-down*, donde se requiere mantener la propiedad del heap al intercambiar nodos.

Funcionamiento: Este método toma como entrada dos nodos del árbol y realiza un intercambio directo de sus valores. Se utiliza comúnmente cuando se necesita ajustar la posición de un nodo dentro del árbol para mantener la propiedad del heap, donde el valor del nodo actual debe ser menor (o mayor, dependiendo del tipo de heap) que los valores de sus hijos.

Código:

```
private void intercambiarNodos(Nodo nodo1, Nodo nodo2) {
    int temp = nodo1.getValor();
    nodo1.setValor(nodo2.getValor());
    nodo2.setValor(temp);
}
```

`getMin` - Método para Obtener el Valor Mínimo del Heap

Descripción: El método `getMin` devuelve el valor mínimo almacenado en el heap sin realizar ninguna modificación en la estructura del árbol. Este método es útil para conocer el valor mínimo presente en el heap sin afectar su estado actual.

Parámetros: Ninguno.

Funcionamiento: El método `getMin` simplemente devuelve el valor almacenado en la raíz del heap, que es el valor mínimo en un Min Heap. No realiza cambios en la estructura del árbol y tiene una complejidad de tiempo constante, ya que el valor mínimo siempre se encuentra en la raíz.

Observaciones: Este método asume que el heap no está vacío. Antes de llamar a `getMin`, se debe verificar que el heap contenga al menos un elemento para evitar errores.

Código:

```
public int getMin() {
    return raiz != null ? raiz.getValor() : -1;
}
```

`eliminarMin` - Método para Eliminar el Valor Mínimo del Heap

Descripción: El método `eliminarMin` elimina el valor mínimo (la raíz) del heap y restaura la propiedad del heap mediante la operación de *bubble-down*. Este proceso es esencial en un Min Heap para mantener la estructura y asegurarse de que el siguiente valor mínimo esté en la raíz del heap.

Parámetros: Ninguno.

Funcionamiento: El método `eliminarMin` primero encuentra el último nodo del árbol y lo intercambia con la raíz. Luego, elimina el último nodo y realiza la operación de *bubble-down* para restaurar la propiedad del heap, asegurándose de que el nuevo valor mínimo se encuentre en la raíz.

Observaciones: Antes de llamar a `eliminarMin`, se debe verificar que el heap no esté vacío para evitar errores. Si el heap está vacío y se intenta eliminar el mínimo, no se realizarán cambios y el heap seguirá vacío.

Código:

```
public void eliminarMin() {
    if (raiz != null) {
        Nodo ultimoNodo = buscarUltimoNodo(raiz);
        intercambiarNodos(raiz, ultimoNodo);
        eliminarUltimoNodo(ultimoNodo);
        bubbleDown(raiz);
    }
}
```

`eliminar` - Método para Eliminar un Valor Específico del Heap

Descripción: El método `eliminar` elimina un nodo específico con un valor dado del heap y restaura la propiedad del heap mediante la operación de *bubble-down*. Este proceso es útil cuando se desea eliminar un valor específico del heap.

Parámetros:

- `valor`: El valor del nodo que se desea eliminar del heap.

Funcionamiento: El método `eliminar` busca el nodo con el valor especificado, lo intercambia con el último nodo del árbol, elimina el último nodo y realiza la operación de *bubble-down* para restaurar la propiedad del heap. Esto asegura que el heap mantenga su estructura y propiedad incluso después de la eliminación de un nodo específico.

Observaciones: Antes de llamar a `eliminar`, se debe verificar que el heap no esté vacío para evitar errores. Si el valor a eliminar no está presente en el heap, la estructura del heap permanecerá sin cambios.

Código:

```
public void eliminar(int valor) {
    Nodo nodoAEliminar = buscarNodo(raiz, valor);

    if (nodoAEliminar != null) {
        Nodo ultimoNodo = buscarUltimoNodo(raiz);
        intercambiarNodos(nodoAEliminar, ultimoNodo);
        eliminarUltimoNodo(ultimoNodo);
        bubbleDown(nodoAEliminar);
    }
}
```

`buscarNodo` - Método para Buscar un Nodo por Valor en el Heap

Descripción: El método `buscarNodo` busca un nodo específico en el heap que contiene un valor dado. Este método es útil para localizar nodos con valores específicos en el árbol.

Parámetros:

- `nodo`: Nodo del árbol donde se inicia la búsqueda.
- `valor`: Valor del nodo que se está buscando.

Funcionamiento: El método `buscarNodo` realiza una búsqueda en profundidad (DFS) en el árbol, comenzando desde el nodo proporcionado. Compara el valor del nodo actual con el valor buscado y continúa la búsqueda recursiva en los subárboles izquierdo y derecho según sea necesario. Si encuentra un nodo con el valor buscado, lo devuelve; de lo contrario, devuelve `null`.

Observaciones: Si el valor buscado no está presente en el heap, el método devuelve `null`. Además, este método asume que el nodo proporcionado como punto de partida pertenece al mismo árbol.

Código:

```
private Nodo buscarNodo(Nodo nodo, int valor) {
    if (nodo == null) {
        return null;
    }

    if (nodo.getValor() == valor) {
        return nodo;
    }

    Nodo nodoIzquierdo = buscarNodo(nodo.getHijoIzquierdo(), valor);
    Nodo nodoDerecho = buscarNodo(nodo.getHijoDerecho(), valor);

    return nodoIzquierdo != null ? nodoIzquierdo : nodoDerecho;
}
```

`buscarUltimoNodo` - Método para Encontrar el Último Nodo del Árbol

Descripción: El método `buscarUltimoNodo` busca y devuelve el último nodo (hoja más a la derecha en el último nivel) del árbol. Este método es útil para operaciones que involucran la eliminación del último nodo, como eliminar la raíz en un heap.

Parámetros:

- `nodo`: Nodo del árbol desde donde se inicia la búsqueda del último nodo.

Funcionamiento: El método `buscarUltimoNodo` utiliza un recorrido por niveles (BFS) para encontrar el último nodo en el árbol. Comienza desde el nodo proporcionado, agrega los nodos al nivel actual a una cola y continúa hasta encontrar el último nodo en el último nivel del árbol.

Observaciones: Este método asume que el nodo proporcionado pertenece al mismo árbol. Si el árbol está vacío, el método devuelve `null`.

Código:

```
private Nodo buscarUltimoNodo(Nodo nodo) {
    Queue<Nodo> cola = new LinkedList<>();
    cola.add(nodo);

    Nodo nodoActual = null;

    while (!cola.isEmpty()) {
        nodoActual = cola.poll();
```

```
        if (nodoActual.getHijoIzquierdo() != null) {
            cola.add(nodoActual.getHijoIzquierdo());
        }

        if (nodoActual.getHijoDerecho() != null) {
            cola.add(nodoActual.getHijoDerecho());
        }
    }

    return nodoActual;
}
```

`eliminarUltimoNodo` - Método para Eliminar el Último Nodo del Árbol

Descripción: El método `eliminarUltimoNodo` elimina el último nodo del árbol, lo cual es útil en situaciones como la eliminación de la raíz después de intercambiarla con el último nodo. Este método es parte del proceso de eliminación y mantenimiento de la propiedad del heap.

Parámetros:

- `nodo`: Último nodo a eliminar del árbol.

Funcionamiento: El método `eliminarUltimoNodo` verifica si el nodo proporcionado tiene un padre. Si tiene un padre, determina si es el hijo izquierdo o derecho de ese padre y actualiza la referencia del padre a `null` en la posición correspondiente. Si el nodo no tiene padre, significa que es la raíz del árbol, y en este caso, se establece la raíz en `null`.

Observaciones: Este método es utilizado en combinación con operaciones que intercambian nodos y luego requieren eliminar el nodo intercambiado.

Código:

```
private void eliminarUltimoNodo(Nodo nodo) {
    if (nodo.getPadre() != null) {
        if (nodo.getPadre().getHijoIzquierdo() == nodo) {
            nodo.getPadre().setHijoIzquierdo(null);
        } else if (nodo.getPadre().getHijoDerecho() == nodo) {
            nodo.getPadre().setHijoDerecho(null);
        }
    } else {
        // Si el nodo no tiene padre, entonces es la raíz.
        raiz = null;
    }
}
```

`bubbleDown` - Método para Restaurar la Propiedad del Heap hacia Abajo

Descripción: El método `bubbleDown` realiza la operación de *bubble-down* para restaurar la propiedad del heap después de realizar cambios en un nodo (como intercambiar nodos o eliminar la raíz). Esta operación asegura que el heap mantenga su estructura y propiedad.

Parámetros:

- `nodo`: Nodo del árbol desde donde se inicia la operación de *bubble-down*.

Funcionamiento: El método `bubbleDown` compara el valor del nodo actual con el valor de sus hijos (si tiene hijos). Si el valor de algún hijo es menor que el valor del nodo actual, intercambia el nodo con el hijo de menor valor y continúa la operación de *bubble-down* en el hijo. Este proceso se repite hasta que el nodo está en una posición donde su valor es menor o igual que el de ambos hijos, o hasta que se convierte en una hoja.

Observaciones: Este método es esencial para mantener la propiedad del heap después de ciertas operaciones, como la eliminación de la raíz. También se utiliza después de insertar un nuevo nodo en el último nivel del heap.

Código:

```
private void bubbleDown(Nodo nodo) {
   Nodo nodoActual = nodo;
        while (!nodoActual.isLeaf()) {
            Nodo hijoIzquierdo = nodoActual.getHijoIzquierdo();
            Nodo hijoDerecho = nodoActual.getHijoDerecho();
            Nodo hijoMenor = hijoIzquierdo;
            if (hijoDerecho != null && hijoDerecho.getValor() < hijoIzquierdo.getValor()) {
                hijoMenor = hijoDerecho;
            }
            if (hijoMenor.getValor() < nodoActual.getValor()) {
                intercambiarNodos(nodoActual, hijoMenor);
            } else {
                break;
            }
            nodoActual = hijoMenor;
        }
}
```

2.2.4. Ejecución del programa

Figura 2.43: Menú principal para seleccionar la opción Árbol Heap

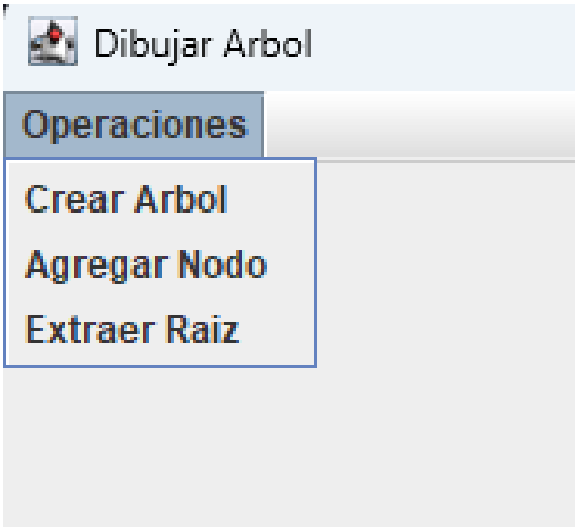


Figura 2.44: Interfaz y operaciones de Heap

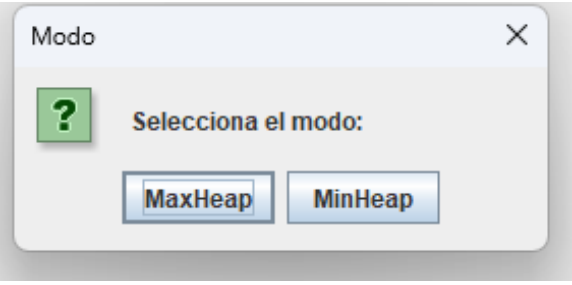


Figura 2.45: Seleccion de tipo de heap

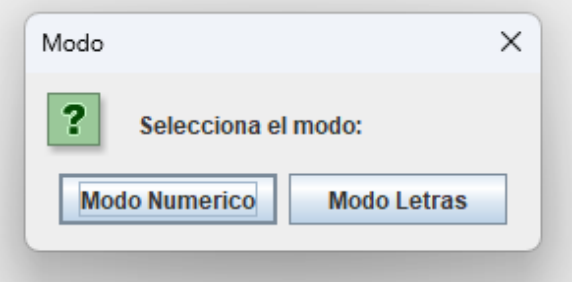


Figura 2.46: Seleccion de tipo de dato

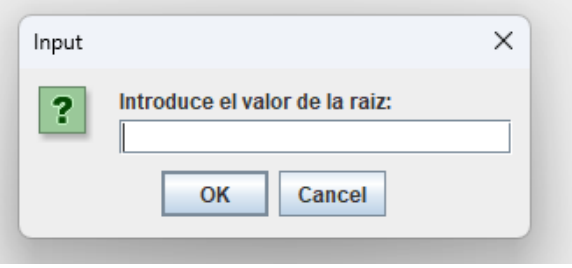


Figura 2.47: Crear el Arbol e iniciar raíz

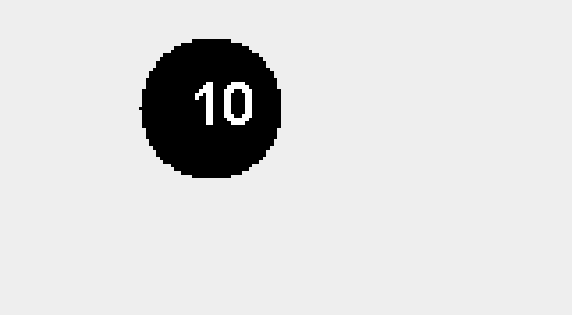


Figura 2.48: visualización del árbol creado

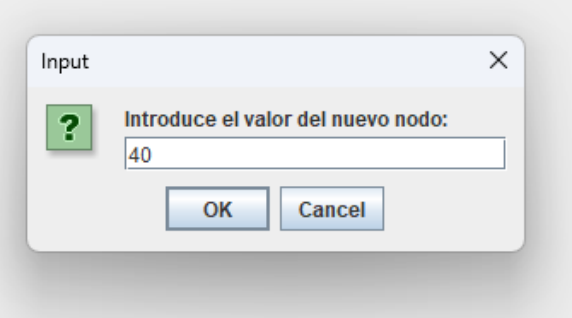


Figura 2.49: Insertar nuevo nodo

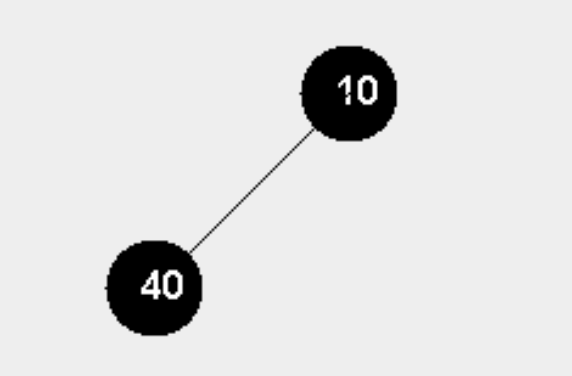


Figura 2.50: visualización inserción de nodo a árbol

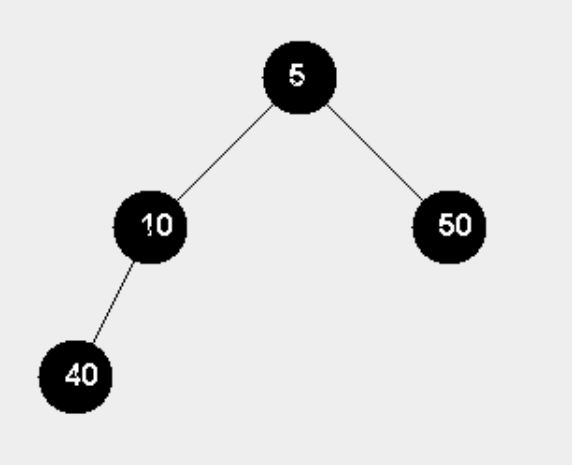


Figura 2.51: inserción de último nodo 5 y se realizan las redistribuciones para que quede como la raíz

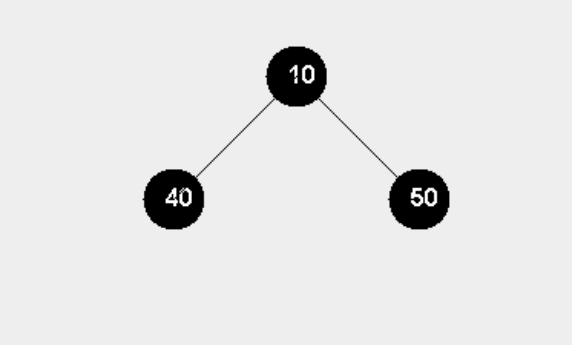


Figura 2.52: Arbol después de seleccionar eliminar raíz

2.3. Árbol Aritmético

2.3.1. Clases del árbol aritmético

Solo se utilizaron dos clases para la creación del árbol aritmético. La primera es el nodo y la segunda la del mismo árbol:

- **Clase Nodo:** Esta clase representa a los nodos del árbol. Contiene un valor para guardar a un numero, un cadena que guarda la operación y tres nodos. Dos para sus hijos y una para su padre.
- **Clase Aritmético:** Esta clase representa el árbol aritmético. Contiene una pila que se usara para guardar al árbol.

2.3.2. Implementación

Para la clase nodo no tienen ningún implementación de importancia. Solo servirá para guardar la información. En la clase aritmético es donde esta la implementación del árbol.

Método de creación: Para crear un árbol de expresión aritmética se necesita recibir una cadena donde esta esté representada. En la mayoría de las fuentes que encontramos utilizan la notación polaca para construir el árbol. Aunque este puede ser el método más sencillo, para el proyecto recibiremos una expresión aritmética con espacios entre cada carácter diferente del espacio. La razón es que nos ayudará a separar la cadena con un tokenizador con el espacio como separador.
Para formar el árbol utilizamos dos pilas. Una pila se usa para guardar los nodos finales y la otra para guardar operaciones y paréntesis en forma de cadenas. Hay cuatro casos a considerar, en cada caso se crea un nodo para guardar la información:

1. Cuando es un número solo lo metemos a la primera pila.

2. Cuando sea el principio de un paréntesis, (, solo lo empujamos a la segunda pila. Esto nos servirá para cuando nos encontremos con el fin del paréntesis,).

3. Con una operación queremos sacar los datos de la primera pila para poder crear nodos. Mientras que la pila no esté vacía y las operaciones guardadas en la pila sea menor o igual a la operación que estamos considerando, creamos un nodo para la operación guardada en la pila. En este nodo indicamos que sus hijos serán los dos nodos hasta arriba de la pila final y también el padre de los nodos hijos. Cuando ya queda el nodo lo metemos en la pila final.
   Acabado este proceso terminamos por meter la operación que estamos considerando a la segunda pila. De esta forma nos aseguramos que las operaciones de mayor prioridad se mantengan en los últimos niveles de la pila, si no hay un paréntesis.

4. Cuando llegamos al final del paréntesis, tenemos que sacar todos los datos guardados dentro de la segunda pila hasta llegar al inicio del paréntesis, “ (” . Seguimos el mismo proceso anterior indicado pero acabamos el while-loop cuando la segunda pila ya no tiene elementos, que teóricamente sería imposible, o cuando nos encontramos “ (“.
   Antes de salir del caso, sacamos el último elemento de la segunda pila, que sería “ (“, para que en futuras operaciones no nos afecte.

Después de este proceso todavía no acabamos ya que la segunda pila puede tener elementos restantes todavía. La razón es que la última operación siempre va a sobrar y si tuviéramos una expresión donde las operaciones tienen un orden descendente de importancia, E.i. AB̂*C-D, no se metería ninguna operación a la pila final. Se sigue el proceso descrito en el caso 3 hasta que la segunda pila ya no tenga nodos.
El nodo que quedó hasta arriba en la pila final lo sacamos para ser la raíz. En código se vería de la siguiente manera:

```
1 public boolean construir(String s){
2     Aritmetico.pila.clear();
3     String[] tokens = s.split(" ");
4     Stack<Nodo> pila = new Stack<>();
5     Stack<String> pila2 = new Stack<>();
6     for (String token : tokens) {
7        if(!esOperador(token)){
```

```
            pila.push(new Nodo(Integer.parseInt(token)));
        }else if( token.charAt(0) == '('){
            pila2.push(token);
        }else if(token.charAt(0) == ')'){
            while (!pila2.isEmpty() && pila2.peek().charAt(0) != '(') {
                Nodo node = new Nodo(pila2.pop());
                node.Hder = pila.pop();
                node.Hizq = pila.pop();
                node.Hder.Padre = node;
                node.Hizq.Padre = node;
                pila.push(node);
            }
                pila2.pop();
            }else if(esOperador(token)){
            while (!pila2.isEmpty() && prioridad(token) <= prioridad(pila2.peek())) {
                Nodo node = new Nodo(pila2.pop());
                node.Hder = pila.pop();
                node.Hizq = pila.pop();
                node.Hder.Padre = node;
                node.Hizq.Padre = node;
                pila.push(node);
            }
            pila2.push(token);
        }
    }
    while (!pila2.isEmpty()) {
        Nodo node = new Nodo(pila2.pop());
        node.Hder = pila.pop();
        node.Hizq = pila.pop();
        node.Hder.Padre = node;
        node.Hizq.Padre = node;
        pila.push(node);
    }
    this.MainRaiz = pila.pop();
     return true;
}
```

Método de resolución: Ya que tenemos el árbol echo podemos resolver la expresión aritmética. Para esto hacemos un recorrido posorden primero. Cada vez que nos encontremos con un nodo hoja lo metemos a una pila. Cuando tengamos dos números en la pila significa que podemos hacer una operación. Sacamos a los dos últimos valores guardados en la pila y basado en la operación que esta guardada en el nodo padre, hacemos la operación indicada y lo insertamos en la pila.
Como es un método recursivo esto se hará para cada uno de los nodos. Al final el resultado final estará hasta arriba de la pila. La implementación del código es la siguiente:

```
    public void NotacionPolacaInversa(Nodo node, Stack<Integer> pila){
      if(node != null){
        if(node.Hizq!=null)
        NotacionPolacaInversa(node.Hizq, pila);
        if(node.Hder!=null)
        NotacionPolacaInversa(node.Hder, pila);

        if(node.Hder == null && node.Hizq == null){
          pila.push(node.value);
        }else if(pila.size() >=2) {
```

```
          int derecha;
          int izquierda;
          derecha =pila.pop();
          izquierda = pila.pop();
          switch (node.op.charAt(0)) {
              case '+': pila.push(izquierda + derecha); break;
                    case '-': pila.push(izquierda - derecha); break;
                    case '*': pila.push(izquierda * derecha); break;
                    case '/': pila.push(izquierda / derecha); break;
                    case '\u005E': pila.push((float)Math.pow(izquierda, derecha));
    break;
                    default: pila.push((float)Math.pow(derecha, 1.0 / izquierda));
    break;
          }
        }
      }
    }
```

Los métodos restantes son de servicio y el entendimiento de el funcionamiento de los arboles aritméticos:

Método para verificación de operaciones: Sirve para determinar si un carácter es una operación o un paréntesis:

```
public static boolean esOperador(String token) {
  return token.equals("+") || token.equals("-") || token.equals("*") || token.
   equals("/") || token.equals("(") || token.equals(")" ) || token.equals("^" ) ||
   token.equals("√." );
}
```

Método para determinar la prioridad: Sirve para saber cual es la prioridad de las operaciones, un mayor numero significa mas significativo:

```
    public static int prioridad(String operador) {
      switch (operador) {
          case "+": return 1;
          case "-": return 1;
          case "*": return 2;
          case "/": return 3;
          case "^": return 3;
          case "√.": return 3;
          default: return 0;
      }
  }
```

Método de impresión: Sirve para poder imprimir el contenido guardado en el árbol. Se utilizo un BFS, donde se imprime el valor guardado en el nodo, su padre y sus hijos. El código es el siguiente:

```
    public void bfs(){
        if (this.MainRaiz == null) {
            return;
        }
        Queue<Nodo> cola = new LinkedList<>();
```

```
6         cola.add(this.MainRaiz);
7         while (!cola.isEmpty()) {
8             int nodosEnNivel = cola.size();
9             for(int i = 0; i< nodosEnNivel; i++){
10                Nodo actual = cola.poll();
11                /* */
12                if(actual.op != null){
13                System.out.print("||Node: " + actual.op);
14                }else{
15                     System.out.print("||Node: " + actual.value);
16                }
17
18                if(actual.Padre != null)
19                if(actual.Padre.op != null){
20                System.out.print(" Padre: " + actual.Padre.op);
21                }else{
22                     System.out.print(" Padre: " + actual.Padre.value);
23                }
24
25                if(actual.Hizq != null)
26                if(actual.Hizq.op != null){
27                System.out.print(" Hizq: " + actual.Hizq.op);
28                }else{
29                     System.out.print(" Hizq: " + actual.Hizq.value);
30                }
31                //System.out.print( (char) actual.value + " ");
32
33               if(actual.Hder != null)
34                if(actual.Hder.op != null){
35                System.out.print(" Hder: " + actual.Hder.op);
36                }else{
37                     System.out.print(" Hder: " + actual.Hder.value);
38                }
39               System.out.print("||");
40
41                if (actual.Hizq != null) {
42                cola.add(actual.Hizq);
43                }
44                if (actual.Hder != null) {
45                cola.add(actual.Hder);
46                }
47            }
48            System.out.println();
49        }
50    }
```

En el programa final este método no se usa, ya que la impresión del árbol va implícita cuando se representa visualmente. Aun con esto, el método sirve para propósitos de debugeo y pruebas rápidas.

2.3.3. Ejecución del programa

Utilizando el mismo ejemplo que se dio en la introducción:

Figura 2.53: Ventana para ingresar la expresión aritmética

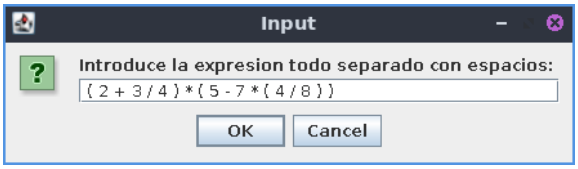


Figura 2.54: La expresión aritmética ingresada

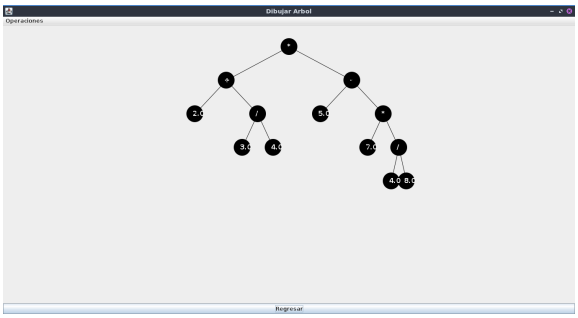


Figura 2.55: El árbol creado

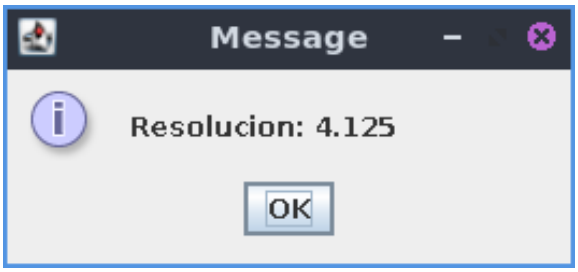


Figura 2.56: Solución de la expresión

2.4. Conclusiones

2.4.1. Hernandez Gallardo Daniel Alonso

Si el objetivo fuera establecer un conocimiento alto y significativo en el manejo de estas estructuras de datos podria decir que se concreto excelentemente, pues el realizar estas implementaciones me hizo entender de una manera increíble como funcionan estas estructuras de datos, sus métodos, en el caso del AVL sus maneras de balancearse, el como y porque se logra esto, este proyecto estuvo genial porque estas implementaciones parecen apantallantes pero resultan en un gran ejercicio mental para entender de gran manera estas abstracciones. El arbo AVL sin duda el más difícil de programar, pues al ser basado en un BST sus métodos de inserción pero especialmente de eliminación se tornan más complicados, fue de vital importancia reconocer el lenguaje en el que lo programaba pues, de la forma en que se usa Java y al ya haber programado un BST en este lenguaje, me facilito las herramientas pero sobretodo las ideas para empezar a crear este árbol, es por eso que empecé a crear el árbol y los nodos con las cosas que iba a necesitar para programar con mayor facilidad los algoritmos y métodos que fuese a requerir. Este árbol tomo unas 15 horas netas de programación para su lógica y métodos, pues los algoritmos de rotación si requieren mucho pensar e incluso estar dibujando para tener una imagen ilustrativa de como funciona, un gran ejemplo y que después de hacer este árbol, el Heap y el Aritmético fueron más que pan comido. El Heap tomo unas 5 horas programarlo, la parte complicada sin duda fue el mantener el árbol como un árbol completo y que fue de vital importancia una vez más el tener unas clases que me facilitaran la programación de este, es curioso como su forma de arreglo es realmente muy buena y que no requiere de tantas cosas como en este que se usaron recorridos para mantener su propiedad de arbol completo. El aritmético el árbol más fácil de los tres, la lógica no fue nada complicada una vez se analizo el funcionamiento del algoritmo, lo difícil fue entender las instrucciones del proyecto, porque se sienten incompletas, especialmente porque te dice nomas que armes el árbol pero pues el árbol se puede armar de distintas maneras según como recibas la expresión matemática, eso es otra cosa, no se menciona como recibir la expresión, por lo que hice lo que creí conveniente para mi como programador pero sobretodo para el usuario, pues introducir la expresión como çomúnmente se hace"me parecía más fácil. Este proyecto se hizieron muchas cosas tanto de estructuras de datos como de programación orientada a objetos, se siente como un cluster de ambos, pues en este proyecto se utilizaron:

- Clases y Objetos
- Metodos de instancia y de clase
- Variables de instancia y clase
- Manejo de excepciones
- Interfaz Gráfica
- Arboles
- Recorridos DFS y BFS
- Pilas y colas
- Manejo excepcional de paquetes

- UML, JavaDoc y Jar
- Patrón de diseño (Este ultimo no se si realmente tomarlo, porque se uso un estilo de 'memento' con una pila para en la interfaz gráfica regresara a un estado anterior.
- Casteo de variables
- Composición de objetos que fue vital en el heap y el AVL para que los nodos se manejen como subárboles.

Este proyecto fue un cluster increíble de muchas cosas y fue muy divertido de realizar. En la realización del proyecto pues no se puede notar una ventaja de los arboles, es realmente lo negativo de este proyecto, para conocimiento de las implementaciones y como funcionan esta increíble pero no cumple con conocer una aplicación real. Pero por decir algunas cosas es que la complejidad algoritmica de estos arboles es bastante util para el manejo rapido de datos, su desventaja es que son algo complicadas de implementar, ademas de que no son infalibles. Como ya menciones, la practica es increbile para terminos de conocimientos adquiridos, pero para aplicaciones pues realmente realizando el proyecto no se visualzian.
se concreto el proyecto al 100 %

2.4.2. Garcia Riba Emilio

El proyecto de árboles binarios en Java fue una experiencia muy enriquecedora para mí. El objetivo de implementar aplicaciones relacionadas con los árboles binarios se cumplió plenamente, ya que implementé tres tipos diferentes de árboles.
Por último, también pude desarrollar mis habilidades de programación orientada a objetos. El proyecto fue una oportunidad para aplicar los conceptos de programación orientada a objetos aprendidos en clase. Pude utilizar varios conceptos de la programacion orientada a objetos para implementar las diferentes funcionalidades del proyecto, de esta forma juntando las dos asignaturas de computacion de este semestre.
En cuanto a la conclusión del proyecto, estoy muy contento con el resultado. Se cumplieron todos los objetivos y pudimos dar más de lo que se nos pedía. El proyecto es un ejemplo de cómo el aprendizaje basado en proyectos puede ser una herramienta muy efectiva para el desarrollo de mis habilidades.
A continuación, se presentan algunos comentarios adicionales sobre la conclusión del proyecto:

- Pude ver el funcionamiento de los diferentes árboles binarios al implementarlos y ver su forma visual en el programa. El árbol aritmético fue el más interesante para mí, ya que muestra cómo se resuelve una expresión aritmética por la computadora.
- El proyecto me ayudó a entender el funcionamiento de los árboles binarios desde un punto de vista más profundo.
- El proyecto también me ayudó a desarrollar mi capacidad de resolución de problemas. Tuve que pensar detenidamente cómo implementar las diferentes funcionalidades del proyecto.

En general, estoy muy satisfecho con el resultado del proyecto. Fue una experiencia muy enriquecedora que me ayudó a aprender mucho sobre árboles binarios.

2.4.3. de la Cruz López, Oscar Abraham

En este proyecto, aplicamos todo lo visto durante el semestre, centrándonos en tres conceptos clave: árboles AVL, Heap y árboles de expresión aritmética. La combinación de investigación teórica y práctica implementación en Java nos proporcionó una comprensión más profunda de cómo estas estructuras de datos impactan en la eficiencia y la flexibilidad de los algoritmos.

La investigación inicial nos permitió comprender la importancia de las estructuras de datos en la optimización de algoritmos y la gestión eficiente de datos. Los árboles AVL emergieron como una poderosa herramienta para mantener el equilibrio en nuestras estructuras, garantizando un rendimiento eficiente en las operaciones de búsqueda, inserción y eliminación. Por otro lado, los Heaps demostraron su valía en situaciones en las que la prioridad es clave, como en colas de prioridad, así como otros (como se puede ver en la sección de `aplicaciones prácticas`).

La implementación práctica de estos conceptos en Java fue un desafío gratificante. Desarrollamos una interfaz gráfica que permitió visualizar de manera clara las operaciones realizadas en cada tipo de árbol. Este enfoque práctico no solo consolidó nuestro entendimiento de los conceptos teóricos, sino que también nos brindó la oportunidad de enfrentarnos a desafíos del mundo real en la programación de interfaces gráficas.

El árbol de expresión aritmética agregó otra dimensión interesante al proyecto, mostrándonos cómo las estructuras de datos pueden aplicarse de manera directa en la manipulación y evaluación de expresiones matemáticas complejas. La interfaz gráfica aquí proporcionó una forma intuitiva de observar la construcción y evaluación de expresiones aritméticas mediante la utilización de árboles.

Este proyecto no solo fortaleció nuestras habilidades técnicas en Java, sino que también nos proporcionó una apreciación más profunda de la importancia de elegir la estructura de datos adecuada para una tarea específica. El camino de la investigación y la implementación nos ha preparado para enfrentar desafíos más complejos en el futuro, equipándonos con las herramientas necesarias para diseñar algoritmos eficientes y robustas aplicaciones informáticas.

Referencias

Discretas, M. (2018, 3 de 4). Capítulo 12: Teoría de Árboles binarios - matemáticas discretas. Descargado de https://medium.com/@matematicasdiscretaslibro/cap%C3%ADtulo-12-teoria-de-arboles-binarios-f731baf470c0 (Medium)

GeeksforGeeks. (2023, 3 de 9). *Complete binary tree.* Descargado de https://www.geeksforgeeks.org/complete-binary-tree/

Gurin, S. (2004). *Árboles avl.* Descargado de http://es.tldp.org/Tutoriales/doc-programacion-arboles-avl/avl-trees.pdf (Recuperado el 30 de noviembre de 2023, de http://es.tldp.org/)

Gómez, C. N. S. (2020, marzo 18). *Ordenamiento por montículos (heapsort) con representación de árbol.* Descargado de https://www.youtube.com/watch?v=Hq3OD8dUTCM

martino. (2020, 19 de noviembre). *Arbol avl.* Descargado de https://www.youtube.com/watch?v=FO6pmLTgf9c (Vídeo)

y Estructuras de Datos II, A. (2021, mayo 1). *Arbol de expresión aritmética.* Descargado de https://www.youtube.com/watch?v=VzBCTIf2a8s