

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

ESTRUCTURA DE DATOS Y ALGORITMOS II

PROYECTO 2. ÁRBOLES BINARIOS

ALUMNOS:

MURRIETA VILLEGAS ALFONSO
REZA CHAVARRÍA SERGIO GABRIEL
VALDESPINO MENDIETA JOAQUÍN

PROFESOR:

EDGAR TISTA GARCÍA

CIUDAD UNIVERSITARIA, CD. MX., 5 DE NOVIEMBRE 2018

Índice

| | |
|---|-----------|
| 1. OBJETIVOS DE LA PRÁCTICA | 3 |
| 2. MARCO TEÓRICO | 3 |
| 3. ANTECEDENTES | 3 |
| 3.1. ÁRBOLES | 3 |
| 3.2. HEAP | 4 |
| 3.3. NOTACIÓN POLACA | 5 |
| 4. DESCRIPCIÓN GENERAL DEL PROYECTO | 5 |
| 5. ANÁLISIS DE CADA APARTADO Y ALGORITMO | 6 |
| 5.1. ÁRBOL BINARIO DE BÚSQUEDA | 6 |
| 5.2. HEAP | 8 |
| 5.2.1. CLASE “nodoH” | 8 |
| 5.2.2. CLASE “heap” | 8 |
| 5.3. HEAP | 11 |
| 5.3.1. CLASE “nodoE” | 11 |
| 5.3.2. CLASE “arbolExpression” | 12 |
| 6. EVIDENCIA DE EJECUCIÓN DEL PROGRAMA | 14 |
| 6.1. ÁRBOL BINARIO DE BÚSQUEDA | 14 |
| 6.2. HEAP | 16 |
| 6.3. ÁRBOL DE EXPRESIÓN ARITMÉTICA | 18 |
| 7. CONCLUSIONES | 18 |
| 8. REFERENCIAS | 19 |

1. OBJETIVOS DE LA PRÁCTICA

Que el alumno implemente árboles binarios y que desarrolle sus habilidades la programación orientada a objetos a través de la aplicación del concepto de árboles como estructuras de datos no lineales.

2. MARCO TEÓRICO

A continuación, se presentan algunos conceptos elementales para la mejor comprensión de este proyecto:

Grado: Es el número de subárboles o hijos que tienen como raíz ese nodo, mientras que el grado de un árbol hace referencia al grado máximo de los nodos de un árbol.

Nodos: Se le conoce como “Nodo terminal” o “Nodo hoja” a los nodos con grado 0 que no tienen subárboles, por otro lado, se les conoce como nodos hermanos a los nodos hijos del mismo nodo padre.

Hijos de un nodo: Nodos que dependen directamente de ese nodo, es decir, las raíces de sus subárboles. Si un nodo x es descendiente directo de un nodo y , se dice que x es hijo de y .

Padre de un nodo: Antecesor directo de un nodo, nodo del que depende directamente. Si un nodo x es antecesor directo de un nodo y , se dice que x es padre de y .

3. ANTECEDENTES

3.1. ÁRBOLES

En las estructuras de datos lineales como las pilas o las colas, los datos se estructuran en forma secuencial es decir cada elemento puede ir enlazado al siguiente o al anterior. En las estructuras de datos no lineales o estructuras multi-enlazadas se pueden presentar relaciones más complejas entre los elementos; cada elemento puede ir enlazado a cualquier otro, es decir, puede tener varios sucesores y/o varios predecesores. Ejemplos de estructuras de datos no lineales son los grafos y árboles.

Un árbol es una colección de elementos llamados nodos, uno de los cuales se distingue como raíz, junto con una relación(rama) que impone una estructura jerárquica entre los nodos. Los árboles genealógicos y los organigramas son ejemplos de árboles.

CARACTERÍSTICAS

La diferencia entre las alturas de los subárboles derecho e izquierdo no debe excederse en más de 1. Cada nodo tiene asignado un peso de acuerdo con las alturas de sus subárboles. Un nodo tiene un peso de 1 si su subárbol derecho es más alto, -1 si su subárbol izquierdo es más alto y 0 si las alturas son las mismas. Los árboles binarios balanceados siguen el concepto del equilibrio entre sus subárboles.

$$\text{Equilibrio} = (\text{altura derecha}) - (\text{altura izquierda})$$

Si alguno de los pesos de los nodos se modifica en un valor no válido (-2 o 2) el árbol debe de seguir el esquema de las rotaciones. Para el balanceo después de una inserción o eliminación se deben de checar si los pesos aún siguen correctos se siguen 4 casos de rotación:

- Rotación Simple Izquierda: Si esta desequilibrado a la izquierda (E_L+1) y su hijo derecho tiene el mismo signo (+) hacemos rotación sencilla izquierda

- Rotación Simple Derecha: Si esta desequilibrado a la derecha (E_i-1) y su hijo izquierdo tiene el mismo signo (-) hacemos rotación sencilla derecha.
- Rotación Doble Izquierda: Si está desequilibrado a la derecha (E_i-1), y su hijo izquierdo tiene distinto signo (+) hacemos rotación doble izquierda-derecha.
- Rotación Doble Derecha: Si esta desequilibrado a la izquierda (E_i+1), y su hijo derecho tiene distinto signo (-) hacemos rotación doble derecha-izquierda.

INSERCIÓN

Para la inserción de nodos lo que se debe realizar es lo siguiente:

- Trazamos una ruta desde el nodo raíz hasta un nodo hoja (donde hacemos la inserción).
- Insertamos el nodo nuevo.
- Volvemos a trazar la ruta de regreso al nodo raíz, ajustando el equilibrio a lo largo de ella.
- Si el equilibrio de un nodo llega a ser $+ - 2$, volvemos a ajustar los subárboles de los nodos para que su equilibrio se mantenga acorde con los lineamientos (que son ± 1).

ELIMINACIÓN

El proceso es similar a un árbol binario ordinario.

- Si el nodo es un nodo hoja, simplemente lo eliminamos.
- Si el nodo solo tiene un hijo, lo sustituimos con su hijo.
- Si el nodo eliminado tiene dos hijos, lo sustituimos por el nodo que se encuentra más a la derecha en el subárbol izquierdo o más a la izquierda en el subárbol derecho

Una vez que se ha eliminado el nodo, se tiene que equilibrar el árbol:

- Si el equilibrio del padre del nodo eliminado cambia de 0 a ± 1 termina el proceso
- Si el padre del nodo eliminado cambio de ± 1 a 0, la altura del árbol ha cambiado y se afecta el equilibrio de su abuelo.
- Si el equilibrio del padre del nodo eliminado cambia de ± 1 a ± 2 hay que hacer una rotación.

Después de concluirla, el equilibrio del padre podría cambiar, lo que, a su vez, podría forzarnos a hacer otros cambios (y probables rotaciones) en toda la ruta hacia arriba a medida que ascendemos hacia la raíz. Si encontramos en la ruta un nodo que cambie de 0 a ± 1 entonces se concluye.

3.2. HEAP

Estructura de datos del tipo árbol binario. La característica es que los montículos máximos tienen es que el nodo padre tiene un valor mayor al de cualquiera de sus nodos hijos. El ordenamiento de árbol cumple con la condición de montículos conectados por sus hijos, y el ordenamiento de los datos va de acuerdo entre más abajo de la ramificación y más a la izquierda, si se ve de manera gráfica.

INSERCIÓN

Se inserta el nuevo valor en la primera posición disponible, o sea en el último nivel del Heap y lo más a la izquierda posible. Después de la inserción en la última posición se verifica si es mayor que el nodo padre, en el tal caso que se cumpla la condición, intercambiarán de posición los elementos, en caso contrario la inserción finalizará.

ELIMINACIÓN

Para la eliminación de raíz se reemplaza la raíz del nodo actual con el elemento que se encuentra en la última posición del Heap respectivo. Al terminar con el reemplazo se verifica si el valor de la nueva raíz es menor que el valor más grande entre sus nodos hijos, para que siga la condición del Heap. Si la condición es verdadera se intercambiarán las referencias con la raíz y el nodo mayor, en caso contrario se terminará el proceso de acomodo.

3.3. NOTACIÓN POLACA

El esquema polaco inverso fue propuesto en 1954 por Burks, Warren y Wright¹ y reinventado independientemente por Friedrich L. Bauer y Edsger Dijkstra a principios de los años 1960, para reducir el acceso de la memoria de computadora y para usar una pila para evaluar expresiones. Posteriormente, Hewlett-Packard lo aplicó por primera vez en la calculadora de sobremesa HP-9100A en 1968 y luego en la primera calculadora científica de bolsillo, la HP-35. Su principio es el de evaluar los datos directamente cuando se introducen y manejarlos dentro de una estructura LIFO (Last In First Out), lo que optimiza los procesos a la hora de programar.

4. DESCRIPCIÓN GENERAL DEL PROYECTO

Se realizó un programa donde se utilizaron las principales aplicaciones de árboles binarios, como son:

- Árboles binarios de búsqueda balanceados
- Heaps
- Árboles de expresiones aritméticas

NOTA: En la misma ejecución del programa el usuario podrá utilizar indistintamente cualquiera de las implementaciones.

Al ejecutar el programa el usuario podrá ver un menú con las diferentes implementaciones de árbol binario, con las opciones para cada uno.

Árbol binario de búsqueda

- Agregar
 - Buscar
 - Eliminar
 - Mostrar árbol

Heap

- Agregar Nodo
 - Eliminar Raíz
 - Mostrar árbol

Árbol de Expresión Aritmética

- Ingresar expresión
- Mostrar árbol
- Resolver

Por otro lado, los datos de entrada del programa serán:

- Claves para construir el heap o el árbol binario de búsqueda

- Expresión Aritmética

Por obvias razones, los datos de salida que el programa mostrará serán las correspondientes a las seleccionadas

Respecto a cada uno de los apartados del proyecto se tuvieron que considerar los siguientes aspectos:

Se tuvo que resolver la expresión aritmética mediante una pila utilizando el algoritmo de notación polaca inversa. Para esto se realizó en 3 etapas del procedimiento o Conversión de la expresión aritmética ingresada en árbol de expresión Aritmética.

- Conversión de la expresión aritmética ingresada en árbol de expresión Aritmética
- Recorrido del árbol en PostOrden para generar la Notación Polaca inversa
- Resolución de la expresión con el uso de una pila

Por otro lado, se escogió Java tanto por su simpleza como por las grandes oportunidades y bondades que ofrece el lenguaje, la ventaja de este lenguaje sobre todo era que en verdad buscar algún método o encontrar información acerca de cómo llevar algo a cabo resulta muy fácil y además de páginas confiables o incluso de bastantes libros.

5. ANÁLISIS DE CADA APARTADO Y ALGORITMO

Para un mejor análisis, a continuación, se describirán las clases como de los métodos que se emplearon en cada uno de los apartados de este proyecto.

5.1. ÁRBOL BINARIO DE BÚSQUEDA

1) Método imprimir

Es un método sin parámetros, para la impresión de un árbol es exactamente el algoritmo de recorrido en PreOrden

2) Método búsqueda

Sus parámetros son un int valor y retorna un booleano. Es un método iterativo, se apoya de un nodo auxiliar “r” el cual iniciara en la raíz del árbol,

Además de una cola de Nodos, esta agregara primeramente a la raíz, después dentro de un ciclo que termina hasta que no haya más nodos, ira desencolando los nodos y verificando que si tiene hijos además de si es mayor o menor, encolar los respectivos hijos, agregando la verificación de si el valor del nodo es el valor buscado, si es así entonces mandara un mensaje de que fue hallado, retornando true, de otra forma no mandara mensaje y retornara false.

3) Método AlturaNODOS

Sus parámetros son un Nodo raíz, int altura y retorna un valor del tipo int. Este método devuelve la altura de un árbol/subárbol, esta solo verifica que no sea nula la raíz, para el proceso utiliza una función auxiliar mandándole la raíz, un entero (nivel) , y la altura actual. Cuando termine el proceso retornara la variable altura, si la raíz es nula entonces retornara 0.

4) Método AlturaAUX

Sus parámetros son Nodo raíz, int nivel, int altura y retorna un valor del tipo int. Esta función recursiva recorrerá tanto el sub árbol derecho como el izquierdo hasta que no haya nodos a donde viajar, en cada paso recursivo se mandara con un nivel más, este si es mayor que la altura actual, la actualizará, al final retornando

esa altura, es decir que la función retornara la mayor distancia.

5) Método BusquedaNODO

Su parámetro es int valor y retorna un Nodo. Este método retorna lo que devuelva la función BusquedaNUTIL

6) Método BusquedaNUTIL

Sus parámetros son Parámetro Nodo raíz, int valor y retorna un valor del tipo Nodo. Este método está basado en la función buscar, este hallara el nodo solicitado trayendo consigo todas las referencias de sus hijos y sus padres, el objetivo de esto es facilitar la eliminación del nodo.

7) Método RotarIzq

Su parámetro es Nodo raíz y retorna un Nodo. Actualiza las referencias de los nodos, mediante un nodo auxiliar, al final lo retorna promueve al aux a padre:

- El hijo derecho de aux será el hijo izquierdo del viejo padre
- El hijo derecho de aux será el viejo padre

8) Método RotarDer

Su parámetro es Nodo raíz y retorna un Nodo. Actualiza las referencias de los nodos, mediante un nodo auxiliar, al final lo retorna

- El aux se promueve a padre
- El hijo izquierdo de aux será el hijo derecho del viejo padre
- El hijo izquierdo de aux será el viejo padre

9) Método RotacionDobleI

Su parámetro es Nodo raíz y retorna un Nodo. Mediante un nodo auxiliar se aplicará rotación simple a la derecha del hijo izquierdo, del resultado se hará una rotación simple a la izquierda, actualizando al nodo auxiliar, retornándolo al final.

10) Método RotacionDobleD

Su parámetro es Nodo raíz y retorna un Nodo. Mediante un nodo auxiliar se aplicará una rotación simple a la izquierda del hijo derecho, del resultado se aplicará una rotación simple a la derecha actualizando al nodo auxiliar, retornándolo al final

11) Método agregar

Su parámetro es int valor. Para esta función solamente se requiere la entrada de un valor entero, esta creara un nodo y lo enviara a una función auxiliar que se encargara de manejar los índices al insertar el nodo en su posición.
correspondiente

12) Método add

Su parámetro es Nodo raíz y retorna un Nodo. Este es un algoritmo recursivo, el cual apoyándose de un nodo auxiliar, se verificara si el nodo a insertar es mayor o menor de la raíz valuada, dependiendo de ello verificara

que si el correspondiente hijo no está ocupado, si es así entonces solamente lo insertara, si no es así entonces lo insertara en alguno de los hijos, dependiendo si es mayor o menor, para ello se llama a la función de manera recursiva pasándole como parámetros n y el hijo, posteriormente se verifica el factor de equilibrio o balance, si este rompe la condición entonces aplicara rotaciones correspondientes a la teoría de árboles AVL, que van desde una rotación simple, a una doble rotación, esto dependiendo donde se localicen los nodos de mayor altura.

13) Método eliminar

Su parámetro int valor. El método principal de eliminación es el encargado de llevar a cabo primero la búsqueda del nodo (usando BusquedaNODO) y posteriormente de los 3 casos que existen al eliminar un nodo, cada uno de ellos con una función propia, lo único que realizan es el manejo de las referencias sobre los nodos.

El nodo que se quiere borrar no tiene ningún nodo hijo eliminarWITHOUT

Parámetro Nodo n

El nodo el cual se borra, el padre se tiene que eliminar la referencia solamente del hijo

El nodo que se quiere borrar solamente tiene un nodo hijo eliminarWITHONE

Parámetro Nodo n

Si el nodo eliminado tiene un hijo, el hijo se promueve a padre y la referencia de ese hijo se elimina

El nodo que se quiere borrar tiene tanto el nodo izquierdo y derecho eliminarWITHTWO

Parámetro Nodo n

En este caso al tener dos hijos se tiene que promover como nuevo padre el menor del subarbol derecho (usa el método Sucesor) la referencia de ese nodo suponiendo hoja, se elimina.

Posterior a este proceso, se tiene que reequilibrar el árbol para ello se tiene una función

14) Método reequilibrar

Como parámetro root (necesariamente la raíz del árbol) y devuelve un Nodo. Aplicando la lógica de inserción, apoyado de un auxiliar que actualizara los nodos se llega hasta los nodos hoja de manera recursiva se evalúa los factores de equilibrio, si rompe la condición de equilibrio se verifica hacia donde este mayor peso (o mayor altura). Dependiendo de ello hará rotaciones para reequilibrar, que van desde una rotación simple si la trayectoria es lineal o una doble rotación si no reequilibrará hasta llegar a la raíz

5.2. HEAP

5.2.1. CLASE “nodoH”

Para la realización del árbol binario Heap se hizo una clase llamada Nodos. Los atributos del nodo serán el valor de los nodos, el nivel en donde se encuentra en el Heap las referencias del nodo padre y sus nodos hijos.

El constructor de un nodo inicializará las referencias de sus conexiones en nulas y el valor y el índice se ingresarán.

5.2.2. CLASE “heap”

Para la clase del Heap los atributos principales son el nodo raíz, el nivel máximo en el que se encuentra el último nodo y la cantidad de nodos agregados a este. El constructor inicializa el Heap con la referencia nula de la raíz y a la cantidad junto con el nivel en 1.

AGREGADO DE NODOS

Para el agregado de un nodo se utilizaron 4 métodos:

1) Método add

Para el método de add obtiene, en entero, el nuevo dato a agregar en el Heap. Con la ayuda de la función find se checa si el dato a agregar se encuentra o no en el Heap, si no es así se llama a la función agregar. El método agregar devuelve la información del nodo creado junto con sus referencias. Este nodo será mandado a la función checar para poder acomodar el nuevo nodo en la lista siguiendo la lógica de un Heap, los nodos padres tienen un valor mayor al de sus nodos hijos.

2) Método find

Obtiene el valor a agregar. Con la lógica del recorrido por Breadth First Search o BFS se checa la información o el valor de cada nodo. Con ayuda de una cola se guarda primero la raíz. Ya al empezar el ciclo se verifica si el nodo que está al inicio de la cola es igual al nuevo valor, en el tal caso que el valor del nodo y el valor por agregar sean iguales, la función regresará verdadero. En caso contrario se agregan al nodo hijo izquierdo y al derecho en la lista, si es que no son nulos. Al revisar todos los nodos y no hallar coincidencia el método devolverá falso. El método ayuda para que no se registren valores repetidos, ya que en un Heap no se permite esto.

3) Método agregar

Se crea una cola y una lista para el guardado de nodos necesarios para el agregado del nodo en el Heap, junto con un booleano inicializado en falso y un nodo llamado regreso. Existirán 2 casos para el agregado, si es el primer nodo (nodo raíz) o si es un nuevo nodo.

Caso 1 (Agregado de raíz)

Para el caso del primer nodo el nivel del Heap aumentará, se inicializa en la raíz con el constructor del nodo con el valor y el nivel actual. Después de esto la cantidad de nodos incrementará y el nodo regreso obtendrá la información del nodo raíz.

Caso 2 (Nuevo nodo)

Para el segundo caso se agrega a la lista la raíz del Heap, siguiendo la lógica del recorrido BFS, se revisará el nodo inicial de la lista. La cola se utilizará para obtener la información de los nodos que se encuentran en el nivel anterior de los nodos hijos. Para que puedan ser agregados los datos de la cola a la lista el nivel de cada nodo será checado, si coincide con el nivel -1 actual del Heap entonces será guardado. Luego de esto se agregarán los nodos izquierdos y derechos a la lista.

Después de haber obtenido todos los nodos del penúltimo nivel se obtiene en una variable auxiliar la información del primer nodo de la lista. Se checa con un ciclo la información de todos los nodos. En el tal caso que el nodo del hijo izquierdo sea nulo se ingresará su información del nodo en esa posición y se asignarán las referencias a ambos nodos. Y si es el caso en el que el nodo izquierdo ya tenga referencia y el nodo derecho no, el nuevo nodo se agregará como referencia del hijo derecho. En ambos casos se cambia el estado del booleano a verdadero, se guardará la información del nodo en el nodo de apoyo regresar y se terminará el ciclo. El ciclo terminará cuando el nuevo nodo sea agregado o en donde todos los nodos de la lista ya tuvieran sus 2 referencias.

En el tal caso que se haya recorrido todo el ciclo terminará (el nivel máximo del nodo ya está lleno) y no se haya guardado el nuevo nodo se checará si el booleano es falso, si es así el nuevo nodo se guardará como referencia del nodo hijo izquierdo del primer nodo del nivel máximo actual del nodo, con esto el nivel del Heap aumentará. Ya al finalizar el método se regresará la información del nodo regresar, esto implica que ya tiene la información de las referencias de sus nodos. La información del nodo servirá para la función checar.

4) Método checar

Método que revisará si el nodo padre del nodo creado es diferente a nulo (si no es el nodo raíz del Heap). En ese caso se checarán copias de este nuevo nodo, tanto su información como la de sus referencias. Para continuar

con la función se revisará si el valor del nodo agregado es mayor a la de su padre, si es así entrará a la condición si el padre es el nodo raíz. En tal caso que la condición se cumpliera se determina si el nodo agregado es el nodo izquierdo o derecho del nodo padre, según sea el caso la información del nodo agregado y su padre actual cambian sus referencias, así toman la posición del otro. (Nota: existía la posibilidad de que sus referencias sean nulas, por eso se utilizaron las excepciones). Y en este caso se reasigna la información del nodo raíz, pasando a ser ahora el nodo agregado.

El otro caso será si el nodo agregado tiene un nodo hijo, si es así se reasignará la información del nodo padre y este nodo. Se seguirá una lógica similar a la vista en el primer caso con la diferencia de que la referencia del nodo abuelo (nodo arriba por 2 niveles) también se actualice la referencia de su respectivo hijo (padre del nodo agregado). Así se obtendrá el cambio, el nodo hijo tendrá la referencia del nodo padre como un hijo y al otro hijo respetivo con la referencia mutua del nodo abuelo y el nodo padre tendrá la posición del hijo y por último el caso en el que el nodo agregado tenga 2 referencias se seguirá la secuencia de pasos del caso anterior con la referencia de sus dos hijos. Cabe recalcar que la referencia de los nodos hijos con respecto a su padre será igualmente cambiada.

Para que el proceso se repita hasta que se encuentre el lugar adecuado en el heap se utilizará un try catch para hacer la recursión del método si es que se obtiene una referencia del nodo se realizará la recursividad, si no es así ahí terminará el proceso del método. Si resulta que la información del nodo agregado cumple con la especificación del heap entonces ya no se volverá a realizar el proceso.

ELIMINACIÓN DE LA RAÍZ

Para la eliminación del nodo raíz se utilizarán 5 funciones. La función deleteRoot servirá como inicializador del proceso al llamar a la función eliminar.

1) Método eliminar

Obtiene la información nodo en la última posición del Heap, este nodo se obtiene del método BuscarUltimo. El uso de esta función es remplazar al nodo raíz actual con la información del último nodo. Con esto se necesitó reasignar las referencias del último nodo con los hijos del nodo raíz y viceversa y eliminar las referencias del nodo raíz. Después de haber realizado esto el nivel del nodo se modificará como 1, la cantidad de nodos disminuirá y la referencia del nodo raíz será ahora el último nodo. Al finalizar el proceso de este método se llama a la función acomodo.

2) Método acomodo

Este método recibirá como parámetro al último nodo (ahora como el nodo raíz). Para esto se crearon 4 atributos, el nodo max servirá como referencia al nodo mayor de los hijos y los otros 3 servirán como copia de la información del nodo a cambiar. Existirán 2 casos de cambio de posición entre nodos.

Caso 1 (Primer acomodo entre la raíz y sus nodos hijos)

Como el intercambio entre el nodo raíz y uno de los nodos hijos, si este tiene 2 hijos. Se revisará si alguno de los datos de los nodos hijos es mayor al nodo padre, si es así se determinará que nodo es mayor con la función max que devuelve el nodo mayor. Ya determinado esto se modificará las referencias del nodo raíz con el nodo mayo de sus hijos, lo cual implica cambio entre los nodos hijos, el nodo padre y la referencia del nuevo nodo raíz. El proceso será similar si el nodo mayor es el nodo izquierdo o derecho. En este caso se checa si la referencia de los hijos del último nodo (antes raíz) tienen información, si es así se llamará nuevamente a la función acomodo con la información del último nodo. (Nota: Si se quiere eliminar un Heap con 2 nodos entrará a la función de acomodo, pero no entrará a ninguno de los 2 casos)

Caso 2 (Acomodo entre nodo intermedio e hijos)

El segundo caso se utiliza cuando la referencia del nodo obtenido por la función checar no sea la del nodo raíz.

Si es así se caerá una copia del nodo padre de este y se checará si las referencias del nodo derecho y nodo izquierdo no son nulas si es así se checará si los valores de los hijos son mayores al padre, si es así con el método máx se checará cuál es el mayor para realizar el cambio. En este caso se deberán modificar las direcciones de los hijos del nodo, del mismo nodo y del nodo padre. Si el nodo aún tiene referencias de hijos se llamará de nuevo a la función.

NOTA: En este caso existe una posibilidad, que el nodo nada más tenga una referencia, este podría considerarse como un caso base de la recursión debido a que si nada más tiene un nodo es seguro que el nodo checado está ubicado en el penúltimo nivel del Heap, con esto se checará si los valores están invertidos de posición, si es así el nodo hijo obtendrá la información del nodo padre y el nodo padre anulará la información de sus hijos y actualizará la información de su nuevo nodo padre.

3) Método buscarUltimo

Este método obtiene la información del nodo en la última posición con respecto al Heap. Se utiliza una cola y una lista. La cola guarda primero a la raíz. Esta saca al nodo inicial de la cola. Se revisa si el nodo no posee referencia alguna de nodos hijos, si es así se guarda ese nodo en la lista Leaf, en caso contrario se guardan en la cola las referencias de los hijos. Ya al tener a todos los hijos, el método devuelve el nodo en la última posición de la lista.

4) Método max

Método el cual devuelve el nodo que tenga el valor mayor entre los nodos obtenidos en sus parámetros. Para el uso en el código los parámetros son los nodos hijos del nodo a checar. El método revisa el valor de los nodos y dependiendo de los casos devuelve al nodo mayor.

5) Impresión de Heap

Por último, se realizó la impresión. El método de impresión recorrerá el Heap con el recorrido BFS. Esto se hizo para recorrer el Heap por niveles. La impresión del Heap mostrará la cantidad de nodos en el Heap, el valor del nodo a checar, el nivel en el que este se encuentra y las referencias que este tenga (tanto sus nodos hijos y su nodo padre)

5.3. HEAP

Para el desarrollo de este apartado del proyecto, se tuvo que partir de la idea más simple de cualquier árbol que es la creación de los nodos, además de la creación de un árbol donde se pudiera meter como datos expresiones aritméticas, es por ello, que continuación se presentarán y describirán las clases correspondientes y asociadas a estos 2 elementos principales:

5.3.1. CLASE “nodoE”

VARIABLES GLOBALES

Como variables globales se declararon 3 del tipo nodo, ya que en cualquier árbol binario siempre hay 3 casos, el nodo hijo izquierdo, el nodo hijo derecho y el nodo Padre, además y como aspecto más importante se declaró una variable del tipo String la cual es la encargada de guardar la expresión asociada al nodo.

MÉTODOS

Para el desarrollo de esta clase realmente no se realizaron muchos métodos, sin embargo, para mayor comprensión, a continuación, se explican cada uno de los métodos realizados:

1) Método constructor

Realmente es un método que se hizo por convención, no realiza, ni instancia ningún elemento

2) Método agregarNodo

Es un método del tipo void donde se le pasa como parámetro un valor de tipo cadena, dentro de este método el valor que se le pasa es asignado directamente a la variable global del tipo String del nodo que se ha creado.

3) Método agregarDer y agregarIzq

Son los dos métodos creados para poder definir qué tipo de nodo es el que se ha creado si el que se encuentra al lado derecho del nodo padre o el que se encuentra al lado izquierdo. Realmente estos 2 métodos solamente son de asignación ya que no realizan nada más.

4) Método toString

Es un método sobre-escrito el cual como comúnmente se ha utilizado en POO, se decidió nuevamente emplear en esta práctica sobre todo para tener una salida totalmente definida de esta clase.

Al tener una salida es por ello que es un método del tipo String el cual mediante los distintos casos de salida abordados en las condiciones if dentro del mismo método, es como realmente hace la asignación y salida de la variable “salida”.

5.3.2. CLASE “arbolExpression”

VARIABLES GLOBALES

Dentro del apartado de las variables globales se encuentran dos del tipo nodo donde un nodo fue dedicado al nodo raíz (Es por ello que se nombró así) y además un nodo auxiliar el cual servirá a lo largo de la ejecución del programa para ir asignando nodos.

Por otra parte, mediante un ArrayList de caracteres es como se guardaron y consideraron los distintos operadores que debía abarcar el programa para las distintas expresiones aritméticas. Como requisito y apartado encargado del guardado y asimilación de la notación polaca se declaró una estructura de datos de tipo pila – stack (Realmente, este apartado fue realmente fácil de realizar debido a la enorme cantidad de bibliotecas que tiene Java).

Por último, también se declararon dos valores extras que son un String dedicado al guardado y asimilación de la expresión general y un valor auxiliar del tipo booleano (Más adelante se explicará su uso).

MÉTODOS

Para mayor comprensión, a continuación, se explican cada uno de los métodos realizados:

1) Método constructor

Es el método dedicado a la asignación y declaración de las variables y métodos que son necesario necesariamente para poder llevar a cabo un árbol de expresión aritmética.

Lo primero que podemos ver es la asignación de la expresión al árbol, posteriormente la declaración del método creararbol el cual es un método de esta clase, posteriormente la impresión del nodo raíz, además y como parte importante de cualquier árbol, un método dedicado al recorrido e impresión del árbol es por ello que se hace la llamada del método “recorrido()”. Por último, una última impresión con la intención de mostrar lo que contiene la pila.

2) Método recorrido

Debe aclararse que este método fue sobre cargado debido a que hay dos casos generales del árbol de expresiones, el primero es cuando solamente hay un nodo el cual es el nodo raíz para ello solamente se imprime el nodo raíz, y el otro es donde realmente hay un árbol en toda su extensión, a continuación, se explicará brevemente:

Mediante la consideración de los casos de asignación de cada nodo hijo a través de condiciones if es como se pudo determinar el recorrido de forma recursivo.

NOTA: Para este método se consideró la notación posfija o PostOrden.

3) Método insertar

El método de insertar es un método de tipo void dedicado a el apartado de asignación de valores a la pila encargada de guardar los datos, es por ello que lo primero que requiere es pasarle mediante una cadena la expresión que se ha ingresado, posterior a ello mediante simplemente una condición a través de la extensión de la cadena (Esto con el fin de poder limitar el tipo de inserción).

NOTA: El método de inserción para el caso de la clase Stack es add

4) Método operaciones

Es el método encargado de realizar la operación asignada a cada símbolo que se declaró previamente en el arraylist de caracteres. Es un método del tipo String donde a través de un switch es como asigna cada operación en un caso (Dentro de cada caso a través del casteo es como asigna y realiza la operación correspondiente).

5) Crear árbol

Es un método de tipo void el cual es directamente llamado en el constructor de esta clase, dentro de este método a través de un string auxiliar es como se va realizando la asignación de cada elemento de la expresión ingresada en el árbol.

La primera condición que se realiza a través de la condición del paréntesis izquierdo donde en caso de que la cadena cumpla, directamente se realiza la asignación de los valores en un nodo auxiliar colocado del lado izquierdo, de ahí posteriormente se vuelve a llamar recursivamente el método creararbol.

La segunda condición a diferencia de la primera se realiza a través de la condición del paréntesis derecho, donde directamente a través del nodo auxiliar se asigna a un nodo padre.

Posterior a ello en caso de que no corresponda a ninguno de los casos anteriores, directamente se pasa a la parte directa de asignación de los valores de la expresión, es por ello que la primera línea directamente es la asignación de un valor al apartado de valor del nodo auxiliar, una vez realizada esa asignación a través de la condición de que solamente se tenga la extensión de un elemento es como nuevamente, mediante otra condición se hace el apartado de la asignación de los caracteres que serán empleados como operadores en la expresión y árbol.

En dado caso de que no tengan esa asignación (O sea que no sean los caracteres - operadores) simplemente se hace la asignación en algún nodo padre para posteriormente volver a llamar el método de manera recursiva.

Por último y como apartado más importante, la recursividad de este método está condicionada a que se realice hasta que la expresión auxiliar esté vacía.

6) eliminar()

Considerando las ventajas que ofrecían los 2 métodos que ofrecía la clase de las cadenas en Java (Checar Notas del método) es como se evaluó la cadena que contenía toda la expresión y además es como se pudo llevar

la evaluación y eliminación* de esta para así ir resolviendo nodo a nodo la expresión.

Lo primero que se realizó fue la declaración de una cadena dedicada a la salida del método (Es necesario denotar que el método es del tipo cadena) la cual fue instanciada directamente con los valores contenidos en el arraylist con los símbolos u operadores empleados para la parte aritmética, posterior a esto la expresión fue partida mediante el método substring.

Aquí es donde realmente entra la parte más importante de todo el algoritmo debido a que a través de una variable global (La variable booleana signo) es como realmente se va evaluando cada uno de los casos de que existe al evaluar e ir quitando los elementos del árbol para así realizar la expresión aritmética.

NOTAS:

Para el desarrollo de este método se tuvo que emplear 2 métodos realmente convenientes de la clase String de Java que son charAt y substring los cuales brevemente serán descritos a continuación:

charAt: es un método que a través de pasarle la posición es cómo evalúa el valor o el carácter de una cadena de caracteres.

Substring: es un método que sirve para poder partir o subdividir un string en varios strings esto con el propósito de poder facilitar la lectura o análisis de una cadena con muchos caracteres.

6. EVIDENCIA DE EJECUCIÓN DEL PROGRAMA

A continuación, se muestran las salidas de los programas con sus correspondientes capturas de pantalla:

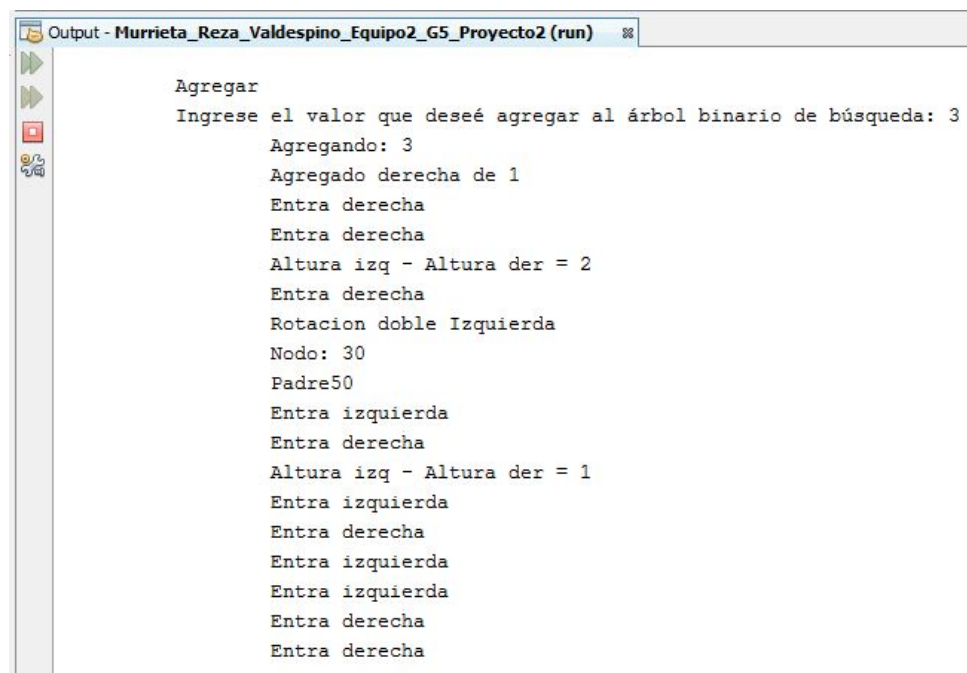
6.1. ÁRBOL BINARIO DE BÚSQUEDA

```
->Árboles Binarios de búsqueda

1)Agregar
2)Buscar
3)Eliminar
4)Mostrar árbol
5)Salir
Opción: 2

Buscar
Ingrese el valor que deseé buscar al árbol binario de búsqueda: 30
NODO 30 fue encontrado
```

Figura 1: Salida del método de búsqueda de un nodo en el árbol binario de búsqueda



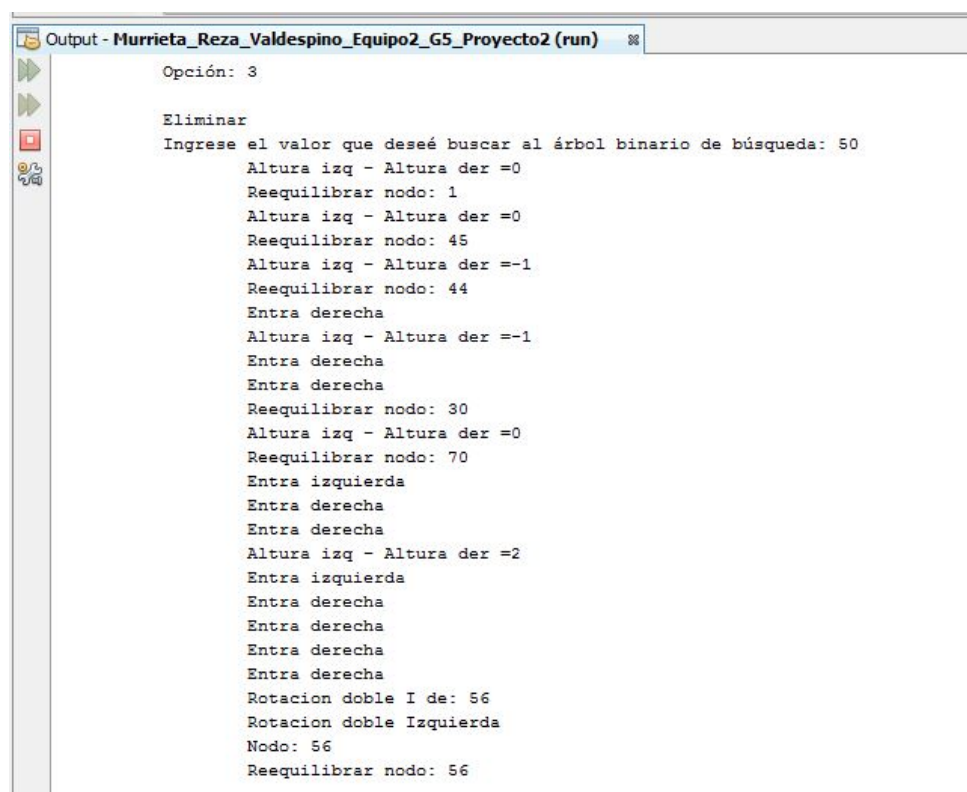
```

Output - Murrieta_Reza_Valdespino_Equipo2_G5_Proyecto2 (run)

Agregar
Ingrese el valor que desee agregar al árbol binario de búsqueda: 3
Agregando: 3
Agregado derecha de 1
Entra derecha
Entra derecha
Altura izq - Altura der = 2
Entra derecha
Rotacion doble Izquierda
Nodo: 30
Padre50
Entra izquierda
Entra derecha
Altura izq - Altura der = 1
Entra izquierda
Entra derecha
Entra izquierda
Entra izquierda
Entra derecha
Entra derecha

```

Figura 2: Salida del método de inserción de un nodo en el árbol binario de búsqueda, a su vez se puede ver cómo es que fueron colocados los demás elementos en el árbol



```

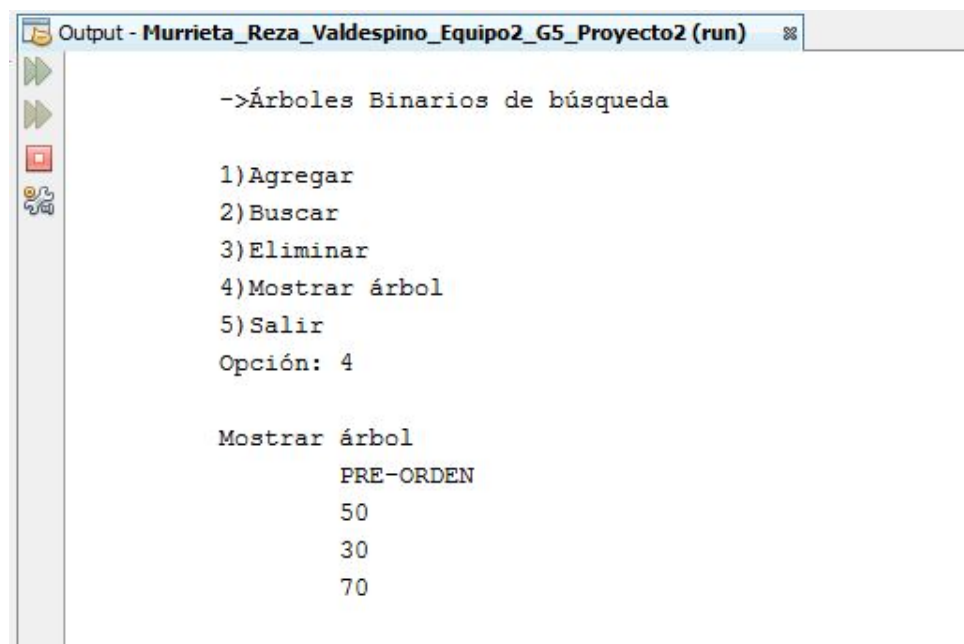
Output - Murrieta_Reza_Valdespino_Equipo2_G5_Proyecto2 (run)

Opción: 3

Eliminar
Ingrese el valor que desee buscar al árbol binario de búsqueda: 50
Altura izq - Altura der =0
Reequilibrar nodo: 1
Altura izq - Altura der =0
Reequilibrar nodo: 45
Altura izq - Altura der =-1
Reequilibrar nodo: 44
Entra derecha
Altura izq - Altura der =-1
Entra derecha
Entra derecha
Reequilibrar nodo: 30
Altura izq - Altura der =0
Reequilibrar nodo: 70
Entra izquierda
Entra derecha
Entra derecha
Altura izq - Altura der =2
Entra izquierda
Entra derecha
Entra derecha
Entra derecha
Entra derecha
Rotacion doble I de: 56
Rotacion doble Izquierda
Nodo: 56
Reequilibrar nodo: 56

```

Figura 3: Salida del método de eliminación de un nodo en el árbol binario de búsqueda, a su vez se puede ver cómo es que fueron reacomodando los demás elementos en el árbol.



```
Output - Murrieta_Reza_Valdespino_Equipo2_G5_Proyecto2 (run)

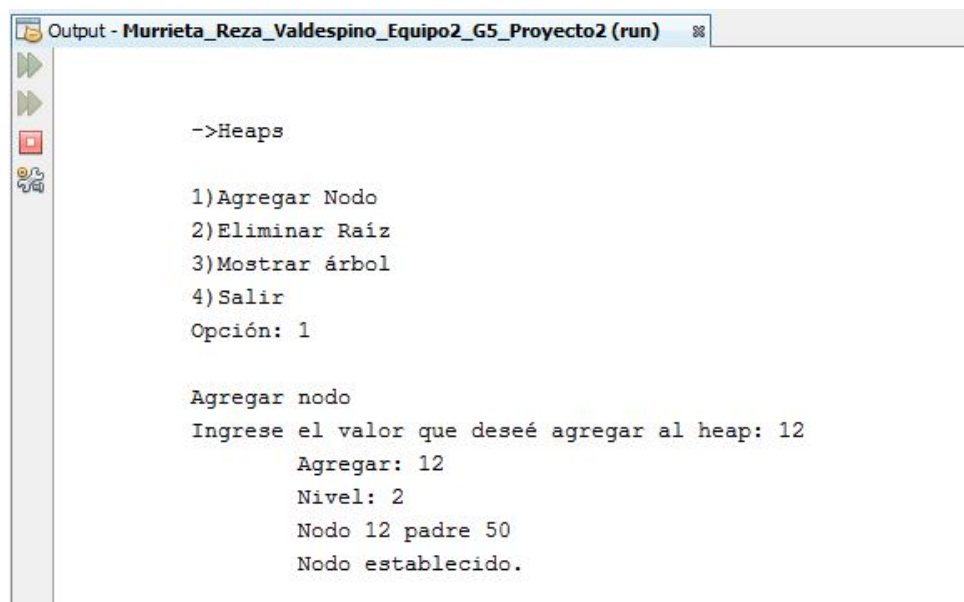
->Árboles Binarios de búsqueda

1)Agregar
2)Buscar
3)Eliminar
4)Mostrar árbol
5)Salir
Opción: 4

Mostrar árbol
PRE-ORDEN
50
30
70
```

Figura 4: Salida del método de mostrar árbol de diferentes árboles binarios de búsqueda.

6.2. HEAP



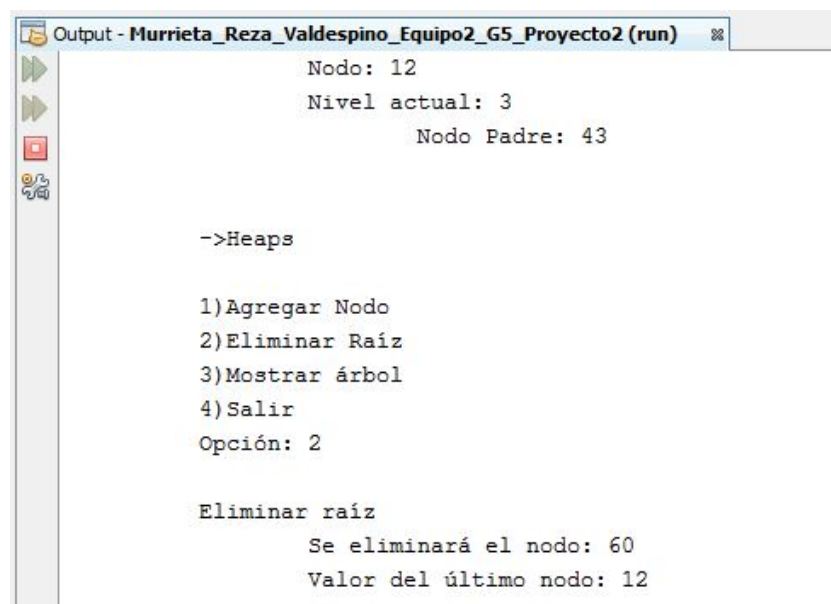
```
Output - Murrieta_Reza_Valdespino_Equipo2_G5_Proyecto2 (run)

->Heaps

1)Agregar Nodo
2)Eliminar Raíz
3)Mostrar árbol
4)Salir
Opción: 1

Agregar nodo
Ingrese el valor que desee agregar al heap: 12
Agregar: 12
Nivel: 2
Nodo 12 padre 50
Nodo establecido.
```

Figura 5: Salida del método de agregar nodo del apartado de Heap



```

Output - Murrieta_Reza_Valdespino_Equipo2_G5_Proyecto2 (run)

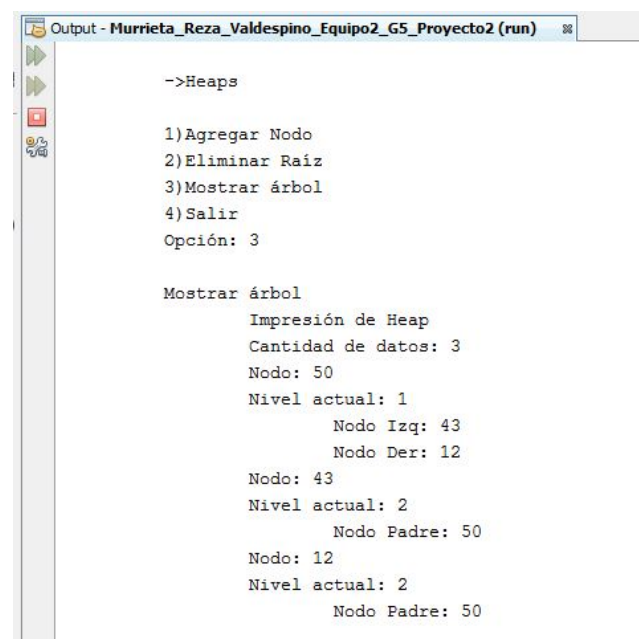
    Nodo: 12
    Nivel actual: 3
        Nodo Padre: 43

->Heaps

1) Agregar Nodo
2) Eliminar Raíz
3) Mostrar árbol
4) Salir
Opción: 2

Eliminar raíz
    Se eliminará el nodo: 60
    Valor del último nodo: 12
  
```

Figura 6: Salida del método de Eliminar Raíz del apartado de Heap



```

Output - Murrieta_Reza_Valdespino_Equipo2_G5_Proyecto2 (run)

->Heaps

1) Agregar Nodo
2) Eliminar Raíz
3) Mostrar árbol
4) Salir
Opción: 3

Mostrar árbol
    Impresión de Heap
    Cantidad de datos: 3
    Nodo: 50
    Nivel actual: 1
        Nodo Izq: 43
        Nodo Der: 12
    Nodo: 43
    Nivel actual: 2
        Nodo Padre: 50
    Nodo: 12
    Nivel actual: 2
        Nodo Padre: 50
  
```

Figura 7: Salida del método de Mostrar árbol del apartado de Heap

6.3. ÁRBOL DE EXPRESIÓN ARITMÉTICA

```

Output - Murrieta_Reza_Valdespino_Equipo2_G5_Proyecto2 (run) #2
->Árboles de expresiones aritméticas

1)Ingresar expresión y resolver
2)Mostrar árbol
3)Salir
Opción: 1

Ingresar expresión

Ingrese la cadena de la operación
Expresión: ((5*2)+((40-3)*(5-3)))
+ Nodo Izquierdo * Nodo Izquierdo 55    Nodo Izquierdo 22
  Nodo Izquierdo * Nodo Izquierdo - Nodo Izquierdo 4040    Nodo Izquierdo 33
    Nodo Izquierdo - Nodo Izquierdo 55    Nodo Izquierdo 33

[5, 2]
[10.0, 40, 3]
[10.0, 37.0, 5, 3]
[10.0, 37.0, 2.0]
[10.0, 74.0]

[84.0]

```

Figura 8: Salida del método ingresar expresión y resolver del apartado de Árbol de Expresión Aritmética., como se puede ver en la primera parte se muestra el árbol de la expresión, posteriormente se muestra cómo es que se va resolviendo elemento a elemento.

```

->Árboles de expresiones aritméticas

1)Ingresar expresión y resolver
2)Mostrar árbol
3)Salir
Opción: 2

Mostrar árbol[13.0, 3, 5]
[13.0, 15.0, 0, -2]
[13.0, 15.0, -2.0]

```

Figura 9: Salida del método Mostrar árbol del apartado de Árbol de Expresión Aritmética.

7. CONCLUSIONES

A lo largo de este proyecto se conocieron y usaron los conceptos previos de árboles binario, donde a través de la implementación de estos es como se pudieron abordar y resolver 3 casos distintos de sus aplicaciones, el primero que fue el caso de la creación de árboles binarios de búsqueda, el segundo que fue la implementación y uso de los HEAPS y el tercero que fue el caso de árboles de expresiones aritméticas a través del uso de la notación

polaca, los cuales sobre todo tuvieron una importante aplicación en el comienzo de las primeras calculadoras de finales de la década de los 70.

Algunos aspectos por resaltar respecto a los distintos árboles abordados en esta práctica podría ser que por ejemplo, respecto al Heap la lógica del agregado, eliminación y reordenamiento de los nodos se tuvo algunos problemas sobre todo por las múltiples referencias que se debían cambiar, es el caso de la reasignación de nodos hijos de los nodos “abuelos” o padres de padres, y las excepciones que podían ocasionar la asignación de posición de estos, como se vio en el código en la parte de la agregación. Respecto al apartado del árbol de expresión la forma en que se abordó el problema realmente simplificó y facilitó muchísimo la programación de este árbol, sin embargo, como desventaja se tuvo que lamentablemente no se pudo hacer directamente un método para poder sacar de manera directa la salida o el árbol en el que se escribía la expresión indicada.

Sin duda un proyecto que pese realmente nos llevó muchísimo esfuerzo poder llevarlo a cabo nos ha brindado una gran capacidad para poder abordar de manera directa diferentes aspectos de los árboles binarios.

8. REFERENCIAS

- Luis Joyones Aguilar. Estructuras de datos en Java. (2013). McGrawHill. Cuarta Edición.
- Bradley N. Miller y David L. Ranum, Franklin, Beedle and Associates. (2011). Problem Solving with Algorithms and Data Structures. Segunda Edición.
- Consultado el 21 de septiembre de 2018, de <https://docs.oracle.com/javase/7/docs/api/>