

	Carátula para entrega de prácticas	
Facultad de Ingeniería		Laboratorio de docencia

Laboratorios de computación

salas A y B

Profesor : Edgar Tista García

Asignatura: EDA 2

Grupo: 4

No. Práctica : 8-9

Integrante: Chagoya Gonzalez Leonardo

No. Equipo de Cómputo: N/A

No. de Lista o Brigada: #Lista 08

Semestre: 2022-2

Fecha de entrega: 26 de Marzo de 2022

Observaciones:

CALIFICACIÓN: _____

Objetivos:

Objetivo: el estudiante conocerá e identificará las características de la estructura no-lineal árbol (árbol binario, binario de búsqueda y Árbol-B)

Objetivo de clase: el alumno analizará las implementaciones proporcionadas y conocerá la forma en la que se pueden implementar estas estructuras de datos

Ejercicio 1

Clase Nodo

Dentro de esta clase se definen 3 atributos los cuales son

- *valor* es un atributo de tipo entero el cual tendrá el valor del nodo
- *izq* y *der* ambos son objetos de tipo nodo y son inicializados con null,
-recordemos que el árbol es binario y cada nodo debe de tener 2 referencias a otros 2 nodos hijos que podrán ser nulos -.

Dentro de esta clase también encontraremos 3 métodos constructores el primero no recibirá ningún parámetro e inmediatamente instancia los atributos *izq* y *der* con null. El tercer método recibe como parámetros un entero y dos objetos Nodo llamados *lt* y *rt*, asignando el entero al atributo *valor* y los valores Nodo *lt* es asignado al atributo *izq* así como *rt* es asignado al atributo *der*.

El segundo método constructor recibirá un entero llamado *data* y hará referencia al tercer método pasando como parámetros el valor entero recibido y dos valores null.

Clase ArbolBin

En esta clase se define un objeto de tipo Nodo llamado *root* este nodo será la raíz de nuestro árbol binario, dentro de esta clase se encontrarán 3 métodos constructores el primero no recibirá ningún parámetro e inmediatamente instancia el atributo *root* con null.

El segundo método constructor recibe un entero llamado *val* e instancia el objeto Nodo pasando como parámetro *val* - en otras palabras se hará uso del segundo método constructor de la clase nodo-.

El tercer método constructor recibirá un objeto de tipo Nodo llamado *root* y asigna este objeto al atributo *Nodo root* de nuestra clase - como ambos se llaman igual la asignación se hace por medio de *this.root* donde este valor será referido a nuestro atributo de la clase *ArbolBin*-.

El método *add* recibe dos objetos de tipo Nodo, padre e hijo, así como un valor entero llamado *lado*, según sea este valor pondrá el hijo en el atributo *izq* o *der* del nodo padre.

El método visit recibe como parámetro objeto de tipo Nodo llamado n imprimiendo en pantalla el valor que almacena este.

El método **breadthFrist** realiza el recorrido del árbol por medio de BFS

Para el recorrido por medio de BFS se modificó únicamente el formato de impresión con respecto al código proporcionado

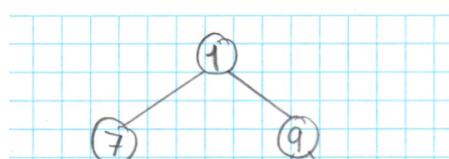
Ejecución

```
Nodo n7 = new Nodo(7);
Nodo n9 = new Nodo(9);
Nodo n1 = new Nodo(1,n7,n9); //nodo raiz
Nodo n15 = new Nodo(15);
Nodo n8 = new Nodo(8);
Nodo n4 = new Nodo(4);
Nodo n2 = new Nodo(2);
Nodo n16 = new Nodo(16);
Nodo n3 = new Nodo(3);

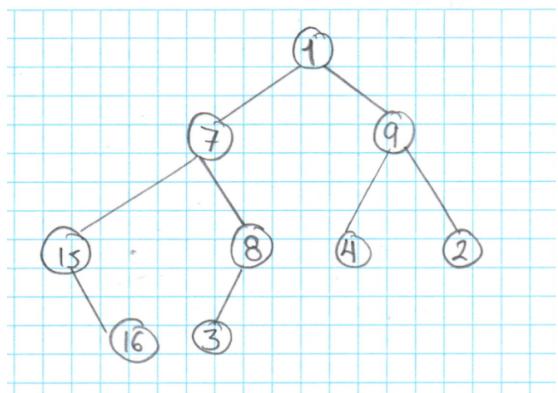
arbol = new ArbolBin(n1);
arbol.add(n7,n15,0);
arbol.add(n7,n8,1);
arbol.add(n9,n4,0);
arbol.add(n9,n2,1);
arbol.add(n15,n16,1);
arbol.add(n8,n3,0);
arbol.breadthFrist();
```

```
run:
Recorrido por BFS del arbol: 1 7 9 15 8 4 2 16 3 BUILD SUCCESSFUL (total time: 0 seconds)
```

Para la construcción de este árbol se puede observar que primero se definen todos los nodos y una vez definidos se empiezan a agregar al árbol binario, a excepción del nodo raíz ya que este ocupa dos nodos previamente definidos para agregarlos a sus referencias izq y der, de tal forma que cuando se instancie el árbol binario tengamos lo siguiente

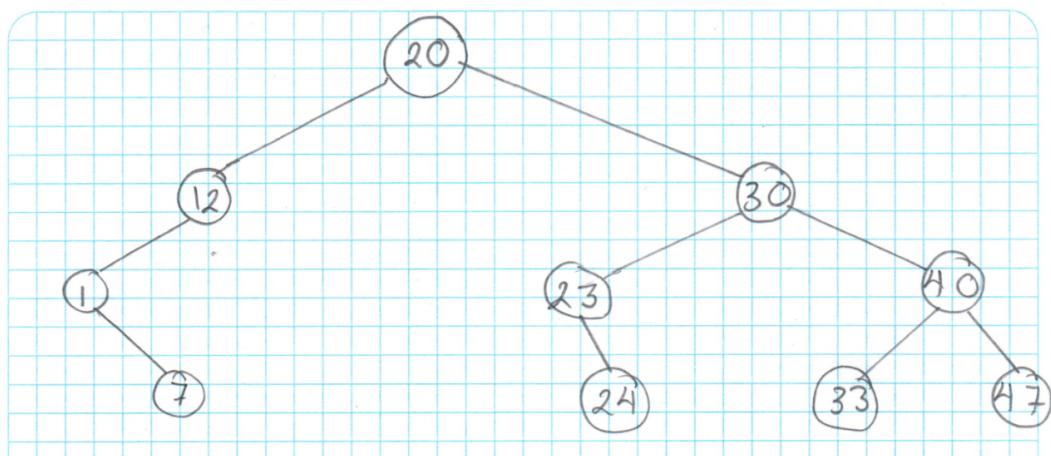


Árbol Construido



Se creó un segundo árbol a mano y posteriormente se realizó su implementación en el programa proporcionado

Árbol Construido



Ejecución

```
//Ejercicio c
Nodo n12 = new Nodo(12);
Nodo n30 = new Nodo(30);
Nodo n20 = new Nodo(20,n12,n30); //nodo raiz
Nodo n1 = new Nodo(1);
Nodo n23 = new Nodo(23);
Nodo n40 = new Nodo(40);
Nodo n7 = new Nodo(7);
Nodo n24 = new Nodo(24);
Nodo n33 = new Nodo(33);
Nodo n47 = new Nodo(47);

ArbolBin arbol_2 = new ArbolBin(n20);
arbol_2.add(n12, n1, 0);
arbol_2.add(n30, n23, 0);
arbol_2.add(n30, n40, 1);
arbol_2.add(n1, n7, 1);
arbol_2.add(n23, n24, 1);
arbol_2.add(n40, n33, 0);
arbol_2.add(n40, n47, 1);
arbol_2.breadthFrist();
```

run:
Recorrido por BFS del arbol: 20 12 30 1 23 40 7 24 33 47

La ventaja de la implementación proporcionada es que es intuitiva, por lo que su entendimiento es fácil, sin embargo a la hora de crear un árbol es muy fácil que por el indicador en el método add se sobreescriba un nodo dando como resultado un árbol diferente. Ejemplo

```
51     ArbolBin arbol_2 = new ArbolBin(n20);
52     arbol_2.add(n12, n1, 0);
53     arbol_2.add(n30, n23, 0);
54     arbol_2.add(n30, n40, 0);
55     arbol_2.add(n1, n7, 1);
56     arbol_2.add(n23, n24, 1);
57     arbol_2.add(n40, n33, 0);
58     arbol_2.add(n40, n47, 0);
59     arbol_2.breadthFrist();
```

Recorrido por BFS del arbol: 20 12 30 1 40 7 47

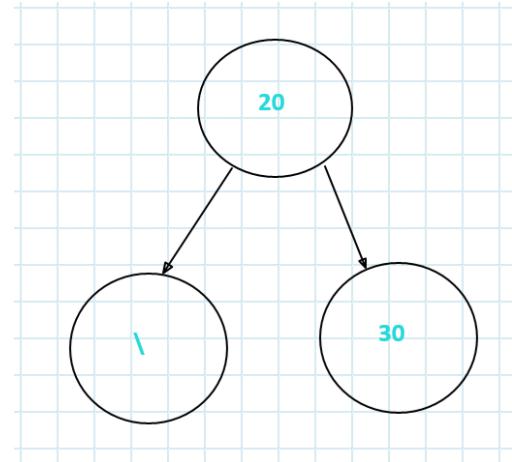
En esta implementación no aparecen los nodos 23 y 33, si bien es cierto, en la línea 53 es agregado como hijo izquierdo de 30 el nodo 23, en la línea 54 el nodo 40 se agrega como hijo izquierdo de 30 perdiendo así el nodo 23 dentro del árbol, mismo caso sucede con el nodo 33.

Además de esto la principal desventaja es que la impresión se realiza por BFS y en dicha impresión no indica si un nodo es hijo izquierdo o derecho esto lo entendemos hasta ver el código.

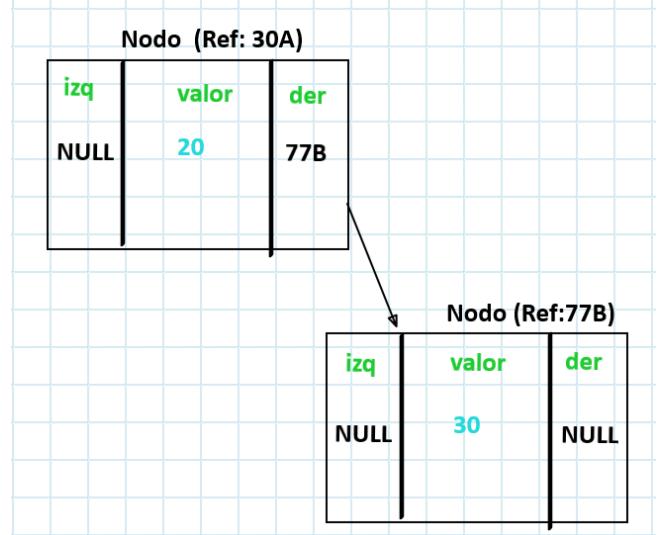
Proceso de Eliminación

Para la eliminación se decidió ocupar el criterio de identificar el nodo a buscar, determinar si este nodo es raíz o intermedio, convertir dicho nodo en una hoja y posteriormente eliminar dicha hoja. En el proceso de convertir una hoja se sustituye el nodo por toda la parte izquierda de los subárboles en caso de no haber parte izquierda se va por la parte derecha hasta que el nodo sea una hoja.

Si bien es cierto gráficamente la estructura de un nodo puede dibujarse de la forma en la que la conocemos.



Es importante ver que un Nodo dada la implementación proporcionada puede verse de la siguiente manera



Como una estructura la cual tiene 3 campos: valor (de tipo entero) , izq (guardará la referencia de un objeto de tipo nodo) y otro campo der (guardando la referencia de un objeto de tipo nodo). Es importante mencionar que podemos acceder al nodo que tiene como referencia 77B a través del nodo que tiene como referencia 30A ya que este último guarda la referencia del primero en su campo der , sin embargo el Nodo 77B no guarda la referencia del nodo 30A , por lo que es imposible que mediante el nodo 77B se logre acceder al nodo 30A.

Dicho estos razonamientos se explica el proceso de eliminación que se lleva a cabo mediante los siguientes métodos:

BusquedaPadre(int value): Mediante el recorrido de BFS determina el nodo padre del nodo que tiene por valor a *value* y retorna el valor del padre, en caso de que este método regrese null significará que el nodo que tiene por valor a *value* no tiene nodo padre y por lo tanto o no existe o es la raíz.

Swap(Nodo Padre, Nodo Hijo): Intercambia las referencias de izq y derecha del nodo Padre con nodo Hijo.

Lado(Padre, value): indica si el nodo correspondiente a *value* es hijo izquierdo o derecho del nodo Padre, para esto se ocupa el metodo BuscarPadre, en caso de que el nodo sea hijo izquierdo devuelve un 0,en caso de que el nodo sea hijo derecho devuelve un 1.

ConvertirHoja (Nodo Padre, Nodo Eliminar, int lado):

Este metodo recibe 3 parametros: un objeto de tipo nodo llamado Eliminar, el cual será nuestro nodo a eliminar, el objeto Padre el cual será el nodo padre del objeto a eliminar y un valor entero llamado lado este valor indicará si el nodo a eliminar es hijo izquierdo o derecho. Es un método recursivo el cual se encargará de colocar a Eliminar como una hoja del árbol, haciendo que eliminar “baje” por la parte izquierda del árbol en caso de que este exista, o por la parte derecha, esto se realizará gracias a la función Swap que será la encargada de cambiar las referencias del nodo Padre.izq y Padre.der con el nodo Hijo.izq e Hijo.der, el proceso recursivo termina una vez que nuestro nodo Eliminar tenga referencias null en sus campos izq y der.

ConvertirRaizHoja(): En este proceso lo que se realiza es verificar si raiz tiene subárbol izquierdo, en caso de que de sea cierta esta afirmación se le da a una variable llamada raizTemp el valor de la referencia de raiz.izq para posteriormente intercambiar los valores a los cuales apuntan raíz y raíz.izq haciendo uso del método swap, raíz será padre de raiz.izq.

Una vez hecho este cambio de referencias hacemos que el valor de la raíz ahora sea `raizTemp` de esta forma nuestra raíz original que era el nodo a eliminar ahora es un nodo intermedio pasando a ser un problema que se puede solucionar con el método `ConvertirHoja()`, en caso de que nuestra raíz no tenga subárbol izquierdo se verifica si tiene subárbol derecho en caso afirmativo procede a realizar el mismo procedimiento anteriormente explicado solo que con `raiz.der`.

`EliminarHoja(int value)`: Para este método es importante mencionar que nuestro nodo que tiene en su campo valor a `value` ya ha sido “convertido” en una hoja, tiene referencias en sus campos `izq` y `der` a `null` por cual solo basta con eliminar la referencia de su nodo padre hacia este nodo hoja.

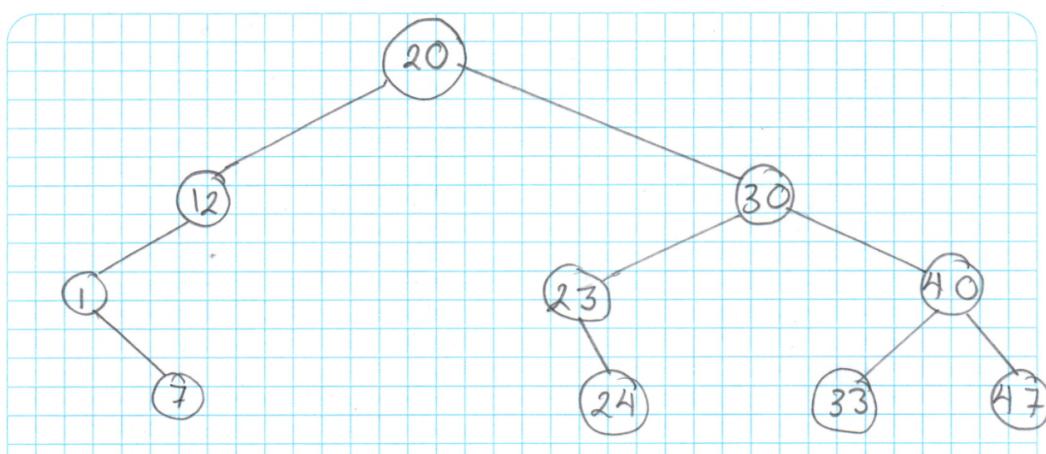
Eliminación (int value) : Este método tiene por objetivo principal determinar el nodo a eliminar por medio de `value`, si nuestro nodo a eliminar tiene un parent significativa que dicho nodo se encuentra dentro del árbol y es, o un nodo intermedio o una hoja, para ello determinamos si nuestro nodo a eliminar es hijo izquierdo o derecho del parent y procedemos a llamar al método `ConvertirHoja`

```
if (Padre.izq.valor == value) { //en caso de que el nodo a eliminar sea hijo izquierdo del parent
    System.out.println(" " + value + " es hijo izquierdo de " + " " + Padre.valor);
    ConvertirHoja(Padre, Padre.izq, 0); //proceso de convertir el nodo a eliminar en un nodo hoja
}
if (Padre.der.valor == value) { //en caso de que el nodo a eliminar sea hijo derecho del parent
    System.out.println(" " + value + " es hijo derecho de " + " " + Padre.valor);
    ConvertirHoja(Padre, Padre.der, 1); //proceso de convertir el nodo a eliminar en un nodo hoja
}
```

En caso de que nuestro nodo a eliminar no tenga parent verificamos si nuestro nodo a eliminar es una raíz, en caso de que esta afirmación se cumpla procedemos a llamar al método `ConvertirRaizHoja`, en caso contrario quiere decir que el nodo a eliminar no existe dentro del árbol.

Ejecución de la Eliminación

Para este proceso se decidió utilizar el árbol construido previamente



```
System.out.println("\n-----Eliminando el nodo 47-----");
arbol_2.Eliminacion(47);
arbol_2.breadthFrist();
System.out.println("\n-----Eliminando a 23-----");
arbol_2.Eliminacion(23);
arbol_2.breadthFrist();
arbol_2.Lado(n47, 0);
System.out.println("\n-----Eliminando el nodo 20-----");
arbol_2.Eliminacion(20);
arbol_2.breadthFrist();
```

Recorrido por BFS del arbol: 20 12 30 1 23 40 7 24 33 47

-----Eliminando el nodo 47-----

47 es hijo derecho de 40
El padre del nodo a eliminar es: 40
El nodo 47 ya es hoja
Eliminacion de hoja
Nodo padre 40 de la hoja 47
EL nodo eliminado es: 47

Recorrido por BFS del arbol: 20 12 30 1 23 40 7 24 33

-----Eliminando a 23-----

23 es hijo izquierdo de 30
El padre del nodo a eliminar es: 30
Intercambio
23 es hijo derecho de 24
El padre del nodo a eliminar es: 24
El nodo 23 ya es hoja
Eliminacion de hoja
Nodo padre 24 de la hoja 23
EL nodo eliminado es: 23

Recorrido por BFS del arbol: 20 12 30 1 24 40 7 33

-----Eliminando el nodo 20-----

Eliminacion de la raiz
Intercambio
Nueva raiz 12
El padre del nodo a eliminar es: 12
Intercambio
20 es hijo izquierdo de 1
El padre del nodo a eliminar es: 1
Intercambio
20 es hijo derecho de 7
El padre del nodo a eliminar es: 7
El nodo 20 ya es hoja
Eliminacion de hoja
Nodo padre 7 de la hoja 20
EL nodo eliminado es: 20

Recorrido por BFS del arbol: 12 1 30 7 24 40 33

Búsqueda

Para realizar la búsqueda se creó el método *Busqueda(int valor)* el cual verificará primero si nuestro nodo que tiene por valor el parámetro valor es la raíz, en caso de que sea cierto retorna una cadena “Si”, en caso de que nuestro nodo a buscar no sea la raíz nos apoyemos del método *BusquedaPadre(int value)* el cual retorna la referencia del nodo padre de nuestro nodo a buscar, si este método regresa un valor diferente de null es que se encontró un nodo en el cual su campo valor coincide con el valor pasado como parámetro y retorna un “Si”, en caso contrario, si regresa un null es que nuestro nodo a buscar no existe dentro del árbol regresando una cadena “No”.

Un aspecto importante para la búsqueda podría ser el no solo determinar el nodo que contiene al valor si no a su padre de forma directa como si el programa fuera dejando un rastro, como si cada nodo tuviera doble referencia.

Ejecución

Es importante mencionar que para este ejemplo la búsqueda se realiza posterior a la eliminación de nodos en el árbol por lo que si bien en nuestro árbol original el nodo 20 se encuentra después de la eliminación ya no está este mismo.

```
System.out.println("\n-----Busqueda-----");
System.out.println("El valor 20 se encuentra en el arbol? "+arbol_2.Busqueda(20) );
System.out.println("El valor 33| se encuentra en el arbol? "+arbol_2.Busqueda(33) );
```

```
-----Busqueda-----
El valor 20 se encuentra en el arbol? No
El valor 33 se encuentra en el arbol? Si
```

Ejercicio 2

Notación Prefija de un Árbol Binario (preOrden)

Para implementar este recorrido se creó el método *PreOrden(Nodo n)* el cual recibe como parámetro un objeto Nodo llamado n, - que en realidad desde la prueba este método recibe como parámetro el nodo raíz-. Dentro de este método se visita el nodo n, se examina si el subárbol izquierdo de n existe y si es así lo examina en PreOrden, si no existe subárbol izquierdo de n se examina el subárbol derecho en Preorden en caso de que este exista.

Notación Infija de un árbol binario (inorden)

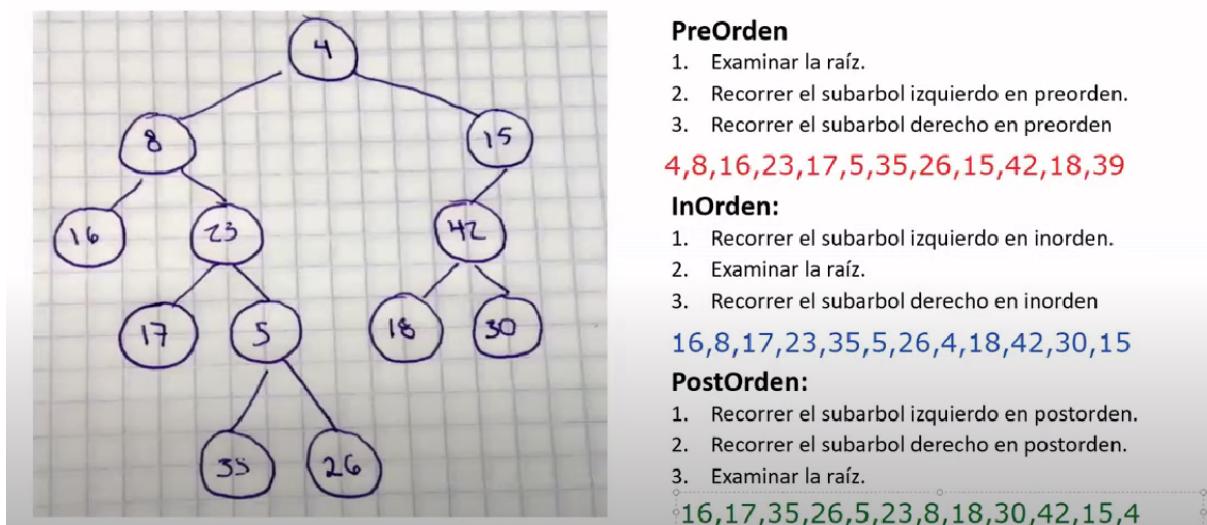
Para implementar este recorrido se creó el método InOrden(Nodo n) el cual recibe como parámetro un objeto Nodo llamado n, - que al igual que en Preorden desde la prueba este método recibe como parámetro el nodo raíz-. Haciendo el proceso de recorrer el subárbol izquierdo en caso de que este exista en InOrden, sino es así examinar el nodo n y recorrer el subárbol derecho en caso de que este exista.

Notación Posfija de un árbol binario (postorden)

Para implementar este recorrido se creó el método PostOrden(Nodo n) al igual que los otros dos recorridos en un inicio n será la raíz y se procederá a recorrer el subárbol izquierdo en PostOrden en caso de que este exista, en caso contrario se recorre el subárbol derecho en PostOrden en caso de que este exista y si ninguna de estas dos llega a ejecutarse se visita el nodo.

Ejecución

Para la prueba de ejecución de estos recorridos nos apoyamos en el ejemplo visto en clase, construyendo el mismo árbol y ejecutando los recorridos en Pruebas



Construcción del árbol

```
//EJERCICIO 2
//se construye otro arbol binario

Nodo n8 = new Nodo(8);
Nodo n15 = new Nodo(15);
Nodo n4 = new Nodo(4,n8,n15); //nodo raiz
Nodo n16 = new Nodo(16);
Nodo n23 = new Nodo(23);
Nodo n42 = new Nodo(42);
Nodo n17 = new Nodo(17);
Nodo n5 = new Nodo(5);
Nodo n18 = new Nodo(18);
Nodo n30 = new Nodo(30);
Nodo n35 = new Nodo(35);
Nodo n26 = new Nodo(26);

ArbolBin arbol3 = new ArbolBin(n4);
arbol3.add(n8, n16, 0);
arbol3.add(n8, n23, 1);
arbol3.add(n15, n42, 0);
arbol3.add(n23, n17, 0);
arbol3.add(n23, n5, 1);
arbol3.add(n42, n18, 0);
arbol3.add(n42, n30, 1);
arbol3.add(n5,n35,0);
arbol3.add(n5,n26,1);
arbol3.breadthFirst();
```

Recorridos

```
System.out.print("Recorrido en Preorden: ");
arbol3.PreOrden(n4);
System.out.println("");

System.out.print("Recorrido en InOrden: ");
arbol3.InOrden(n4);
System.out.println("");

System.out.print("Recorrido en PostOrden: ");
arbol3.PostOrden(n4);
System.out.println("");
```

Recorrido por BFS del arbol: 4 8 15 16 23 42 17 5 18 30 35 26
Recorrido en Preorden: 4 8 16 23 17 5 35 26 15 42 18 30
Recorrido en InOrden: 16 8 17 23 35 5 26 4 18 42 30 15
Recorrido en PostOrden: 16 17 35 26 5 23 8 18 30 42 15 4

Al final tanto los recorridos realizados en clase como los implementados en el código coinciden, por lo cual están completados de forma correcta.

Ejercicio 3

Árbol Binario de Búsqueda

Añadir

Para el proceso de insertar un nuevo elemento se realizó un método `add(Nodo turno, Nodo n, int value)` en el cual se ocupa la sobreescritura del método definido en `arbolBinBusq`, este nuevo método será un proceso recursivo el cual mediante el recorrido del árbol determina en donde se inserta el nodo `n`. El recorrido del árbol empezará desde la raíz, nuestro nodo a insertar será `n` y `value` servirá en un inicio como indicador de que se está comparando la raíz. El recorrido verificará si el valor del nodo en turno es mayor que el valor del nodo a insertar por lo que la ubicación del nodo a insertar tiene que estar a la derecha, en caso de que el valor del nodo a insertar sea mayor que el valor del nodo en turno su ubicación estará a la derecha, el proceso recursivo termina una vez que la ubicación del nodo a insertar sea nula, entonces es ahí donde se le añadirá al árbol.

Al ser una sobreescritura nuestro nuevo método `add` debía de tener los mismos parámetros que el `add` de un árbol binario, sin embargo creo que el parámetro `value` para este caso está de más.

Ejecución

```
//Ejercicio 3

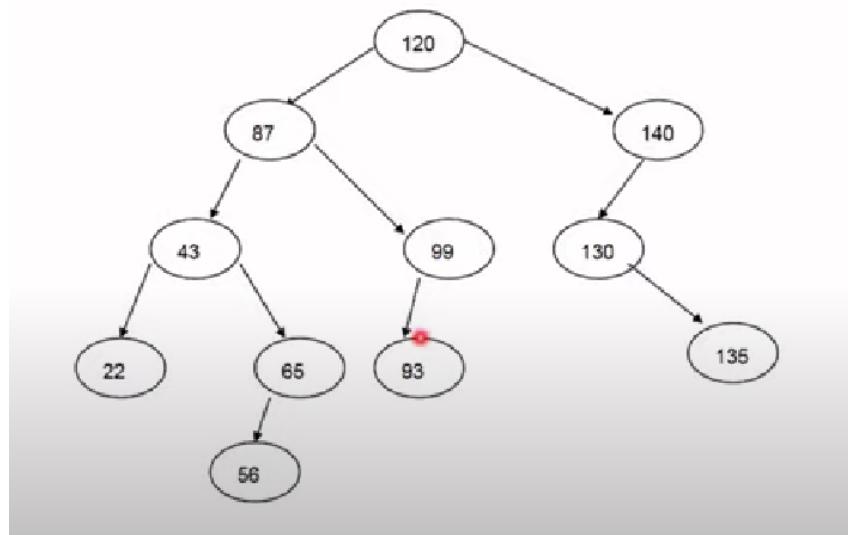
Nodo n120 = new Nodo(120); //nodo que sera raiz
Nodo n87 = new Nodo(87);
Nodo n140 = new Nodo(140);
Nodo n43 = new Nodo(43);
Nodo n99 = new Nodo(99);
Nodo n130= new Nodo(130);
Nodo n22 = new Nodo(22);
Nodo n65 = new Nodo(65);
Nodo n93 = new Nodo(93);
Nodo n135 = new Nodo(135);
Nodo n56 = new Nodo(56);

ArbolBinBusq arbol4 = new ArbolBinBusq();
arbol4.add(n120, n120, 2); //para añadir raiz

arbol4.add(n120, n87, 2);
arbol4.add(n120, n140, 2);
arbol4.add(n120, n43, 2);
arbol4.add(n120, n99, 2);
arbol4.add(n120, n130, 2);
arbol4.add(n120, n22, 2);
arbol4.add(n120, n65, 2);
arbol4.add(n120, n93, 2);
arbol4.add(n120, n135, 2);
arbol4.add(n120, n56, 2);

Recorrido por BFS del arbol: 120 87 140 43 99 130 22 65 93 135 56
```

El árbol creado , es el mismo árbol que se utilizó como ejemplo para explicar el proceso de eliminación en clase



Eliminación

Para el proceso de eliminación se sobreescribe el metodo Eliminacion(int value) definido previamente en arbolBin en este nuevo proceso lo que se realiza es que encuentra el nodo padre del nodo asociado a value, encuentra el nodo más a la derecha del subárbol izquierdo, encuentra el padre del nodo más a la derecha del subárbol izquierdo, realiza el intercambio de referencias de los hijos del nodo a eliminar con el nodo más a la derecha y actualiza las referencias que son apuntadas por el padre del nodo a eliminar y el padre más a la derecha, posteriormente si nuestro nodo a eliminar queda como una hoja se elimina la hoja, en caso contrario pasa al proceso de convertirHoja al nodo a eliminar. Para este proceso se definieron las siguientes funciones.

Busqueda(Nodo turno, int value) : Es un proceso recursivo para determinar el nodo padre del nodo que tiene por valor a value, haciendo uso de la lógica de búsqueda en la cual si el valor a buscar es menor que el valor del nodo en turno busca hacia el subárbol izquierdo y en caso de que nuestro valor a buscar es mayor que el valor del nodo en turno busca hacia el subárbol derecho es como se encuentra el padre. En caso de no encontrarse al nodo padre retorna un null y por lo tanto el nodo no existe dentro del árbol.

BuscarMasDer(Nodo n): Es un proceso recursivo el cual va a recibir el nodo a la izquierda de nuestro nodo a eliminar por lo que si este nodo es null el método retorna dicho valor e indicará que no hay subárbol izquierdo del nodo a eliminar. En caso contrario seguirá recorriendo hacia la derecha del nodo n y cuando ya no existan más elementos hacia la derecha retorna este valor encontrado.

DarNodoHijo(Nodo Padre, int n): Devuelve el nodo hijo ya sea izquierdo o derecho del nodo Padre pasado como parámetro la comparación se hace a través de los valores Padre.izq.valor , Padre.der.valor y n.

SwapBinBusq(Nodo Padre, Nodo PadreMasDer, Nodo eliminar, Nodo masDer): Es un método encargado de intercambiar las referencias hacia izq y der del nodo a eliminar y el nodo masDer , además de actualizar la referencia de los padres hacia estos dos. En caso de que nuestro nodo a eliminar sea una raíz se detecta mediante el padre ya que este será null haciendo un proceso parecido de intercambio de referencias entre nuestra raíz y el nodo más a la derecha solo que en esta ocasión debe de actualizar el nodo root con la referencia del nodo más a la derecha del subárbol izquierdo para que este nodo sea la nueva raíz del árbol.

EliminarRaiz(): Este proceso realiza un intercambio entre el nodo raíz y su nodo izquierdo mediante la función SwapBinBusq, -esto en caso de que exista nodo izquierdo-, posteriormente el nodo a eliminar (antigua raíz) pasa a ser un nodo intermedio el cual se convierte en hoja con el método ConvertirHoja() de la clase arbolBin, en caso de que nuestro árbol no tenga sub árbol izquierdo la raíz se reemplaza por la derecha hasta llegar a ser una hoja.

EliminarNodoIntermedio(int value): Este proceso determina el padre del nodo a eliminar mediante el método Busqueda empezando a buscar desde la raíz si el método devuelve un valor diferente de null y nuestro nodo a buscar no es una raíz, el nodo a eliminar existe, por lo cual mediante la función DarNodoHijo se determina el nodo a eliminar y posteriormente se busca el nodo más a la derecha del subárbol izquierdo de nuestro nodo a eliminar, esto último gracias al método BuscarMasDer, si nuestro nodo más a la derecha es diferente de null es que al menos el nodo a eliminar tiene árbol izquierdo, procediendo a buscar el padre del nodo más a la derecha si este padre es igual a nuestro nodo a eliminar significa que el sub árbol izq no tiene más nodos a la derecha por lo cual procedemos a convertir en hoja de forma directa el nodo a eliminar por la parte izquierda. Si el padre del nodo más a la derecha es diferente de nuestro nodo a eliminar se realiza un intercambio de referencias entre el nodo a eliminar, el nodo más a la derecha y se actualizan las referencias de sus respectivos padres, una vez realizado este proceso nuestro nodo a eliminar lo convertimos en Hoja y se procede a eliminar.

Eliminación(int value): Este último método determina si nuestro nodo a eliminar es la raíz en caso de que sea así llama al método EliminarRaiz() y procede a ejecutar, en caso contrario simplemente ejecuta el método EliminarNodoIntermedio()

Ejecución

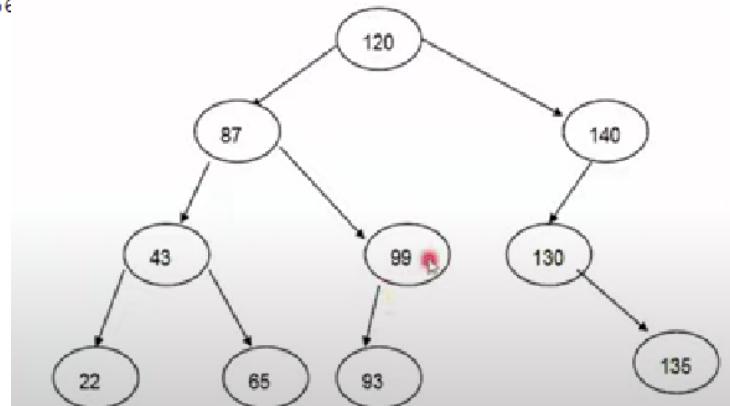
Eliminación de 56

Recorrido por BFS del arbol: 120 87 140 43 99 130 22 65 93 135 56

-----Eliminando 56-----
Haciendo el proceso de búsqueda

El padre de 56 es: 65
No hay sub arbol izquierdo
El padre del nodo a eliminar es: 65
El nodo 56 ya es hoja
Nodo padre 65 de la hoja 56
El nodo eliminado es: 56
Recorrido por BFS del arbol: 120 87 140 43 99 130 22 65 93 135

Árbol Esperado

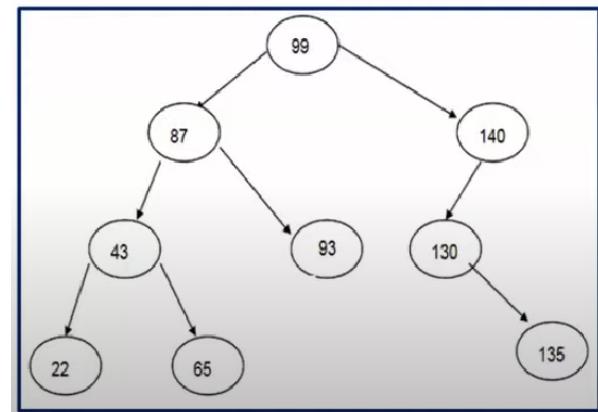


Eliminación de 120

-----Eliminando 120-----
No existe nodo padre de 120
Eliminacion de una raiz
El nodo mas a la derecha del sub arbol izquierdo de 120 es 99
El padre 99 es: 87
Eliminacion de: 120
****INTERCAMBIO DEL NODO MAS A LA DERECHA CON EL NODO A ELIMINAR**
Nueva raiz: 99
NODO A ELIMINAR:
Valor: 120
Hijo der nulo
Hijo izq: 93

Se entra a eliminar el intermedio
120 es hijo derecho de 87
El padre del nodo a eliminar es: 87
Intercambio
120 es hijo izquierdo de 93
El padre del nodo a eliminar es: 93
El nodo 120 ya es hoja
Eliminando hoja
Nodo padre 93 de la hoja 120
El nodo eliminado es: 120
Recorrido por BFS del arbol: 99 87 140 43 93 130 22 65 135

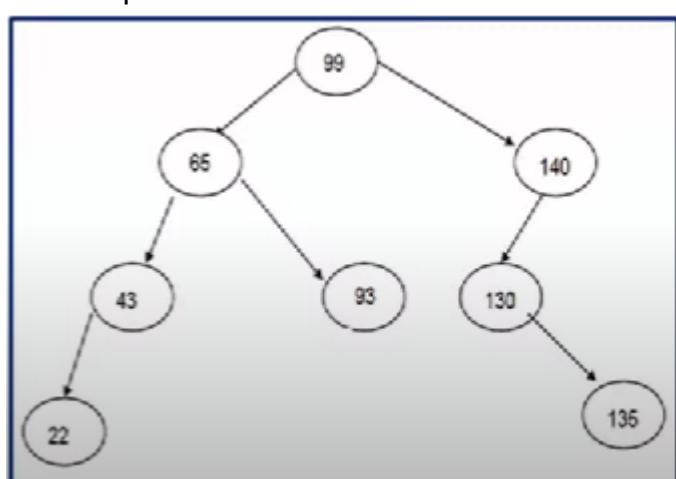
Árbol Esperado



Eliminando el nodo 87

```
-----Eliminando 87-----
Haciendo el proceso de busqueda
El padre de 87 es: 99
El nodo mas a la derecha del sub arbol izquierdo de 87 es 65
El padre 65 es: 43
Eliminacion de: 87
*****ANTES DEL INTERCAMBIO*****
Valor del parente
Valor: 99
Hijo izq: 87
Hijo der: 140
*****
Valor del parente mas a la derecha:
Valor: 43
Hijo izq: 22
Hijo der: 65
*****
Valor del nodo a eliminar
Valor: 87
Hijo izq nulo
Hijo der nulo
*****
Valor del nodo mas a la derecha
Valor: 65
Hijo izq: 43
Hijo der: 93
*****
87 es hijo derecho de 43
El parente del nodo a eliminar es: 43
El nodo 87 ya es hoja
Valor: 87
Hijo izq nulo
Hijo der nulo
*****
*****INTERCAMBIO DEL NODO MAS A LA DERECHA CON EL NODO A ELIMINAR**
*****Despues DEL INTERCAMBIO*****
Valor del parente
Valor: 99
Hijo izq: 65
Hijo der: 140
Nodo parente 43 de la hoja 87
EL nodo eliminado es: 87
Recorrido por BFS del arbol: 99 65 140 43 93 130 22 135
```

Árbol Esperado



Ejercicio 4

Implementación de Árboles binarios y Árboles binarios de búsqueda

Se creó un menú en el cual es usuario elegirá la implementación de arboles binarios o árboles binarios de búsqueda según sea el caso se mostrará un nuevo menú para utilizar las operaciones en un árbol binario y/o un árbol binario de búsqueda, la principal verificación para poder realizar una operación es que el árbol exista. Una vez implementado por separados los árboles binarios y árboles binarios de búsqueda su implementación en conjunto no requirió mayores detalles, por lo cual este ejercicio se completó al 100 de forma más sencilla, incluyendo una breve explicación de lo que sucede en cada caso

Menú de árbol Binario

Operación de Crear árbol y Agregar nodo

```
-----MENU PRINCIPAL-----
1.-Arboles Binarios
2.-Arboles Binarios de Busqueda
3.-Salir
Opcion: 1

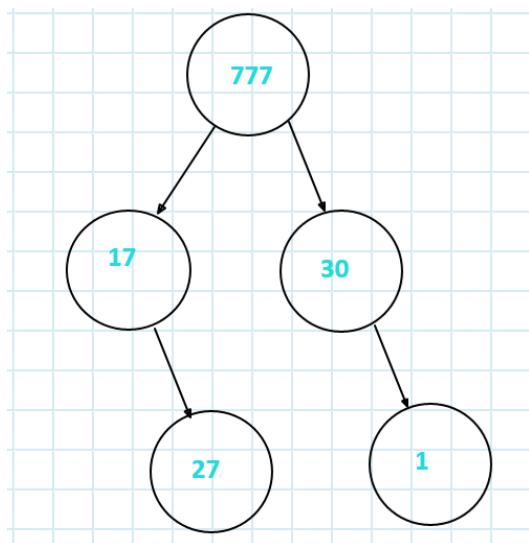
-----Menu Arbol Binario-----
1.-CrearArbol
2.-Agregar Dato
3.-Eliminar dato
4.-Imprimir Arbol
5.-Notacion Prefija
6.-Notacion Infija
7.-Notacion Posfija
8-Salir
Opcion: 1
Ingrese el valor del nodo raiz: 777
ARBOL CREADO CON EXITO

-----Menu Arbol Binario-----
1.-CrearArbol
2.-Agregar Dato
3.-Eliminar dato
4.-Imprimir Arbol
5.-Notacion Prefija
6.-Notacion Infija
7.-Notacion Posfija
8-Salir
Opcion: 2
Ingrese el valor del padre: 777
Ingrese el valor a agregar: 30
0.-Ingresar del lado izquierdo 1.-Ingresar del lado derecho      1
Recorrido por BFS del arbol: 777 30 1

-----Menu Arbol Binario-----
1.-CrearArbol
2.-Agregar Dato
3.-Eliminar dato
4.-Imprimir Arbol
5.-Notacion Prefija
6.-Notacion Infija
7.-Notacion Posfija
8-Salir
Opcion: 2
Ingrese el valor del padre: 777
Ingrese el valor a agregar: 17
0.-Ingresar del lado izquierdo 1.-Ingresar del lado derecho      0
Recorrido por BFS del arbol: 777 17 30 1
```

Árbol Construido

```
-----Menu Arbol Binario-----
1.-CrearArbol
2.-Agregar Dato
3.-Eliminar dato
4.-Imprimir Arbol
5.-Notacion Prefija
6.-Notacion Infija
7.-Notacion Posfija
8-Salir
Opcion: 2
Ingrese el valor del padre: 17
Ingrese el valor a agregar: 27
0.-Ingresar del lado izquierdo 1.-Ingresar del lado derecho      1
Valor Padre
Valor: 17
Hijo izq nulo
Hijo der nulo
*****
Recorrido por BFS del arbol: 777 17 30 27 1
```



Recorridos

```
-----Menu Arbol Binario-----
1.-CrearArbol
2.-Agregar Dato
3.-Eliminar dato
4.-Imprimir Arbol
5.-Notacion Prefija
6.-Notacion Infija
7.-Notacion Posfija
8-Salir
Opcion: 5
Recorrido Preorden
777 17 27 30 1
-----Menu Arbol Binario-----
1.-CrearArbol
2.-Agregar Dato
3.-Eliminar dato
4.-Imprimir Arbol
5.-Notacion Prefija
6.-Notacion Infija
7.-Notacion Posfija
8-Salir
Opcion: 6
Recorrido InOrden
17 27 777 30 1
```

```
-----Menu Arbol Binario-----
1.-CrearArbol
2.-Agregar Dato
3.-Eliminar dato
4.-Imprimir Arbol
5.-Notacion Prefija
6.-Notacion Infija
7.-Notacion Posfija
8-Salir
Opcion: 7
Recorrido PostOrden
27 17 1 30 777
```

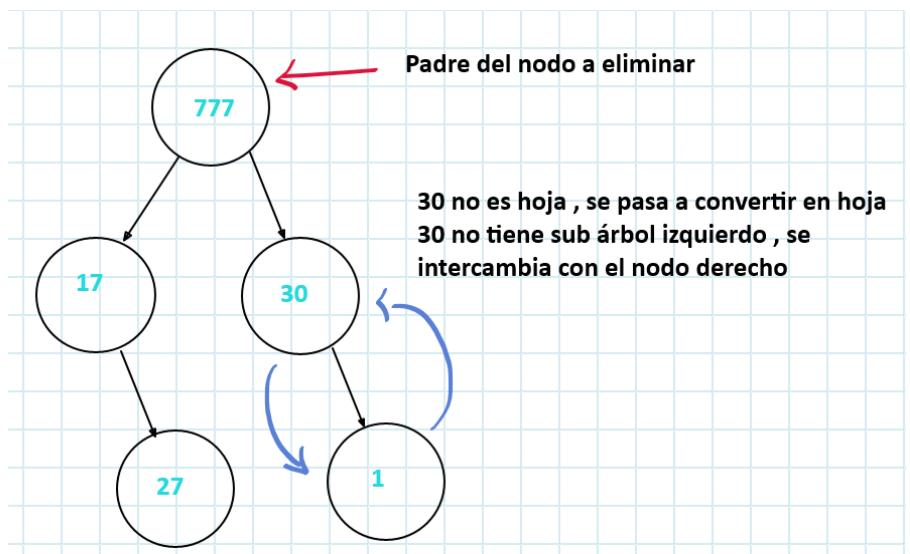
Eliminación

-----Menu Arbol Binario-----

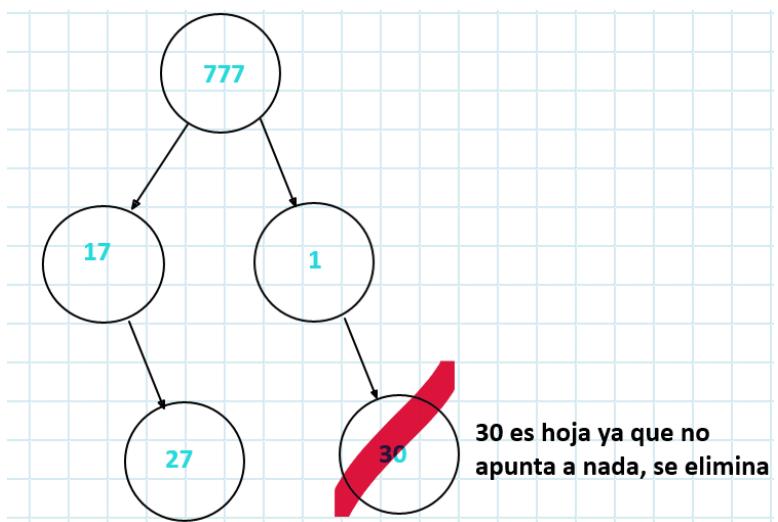
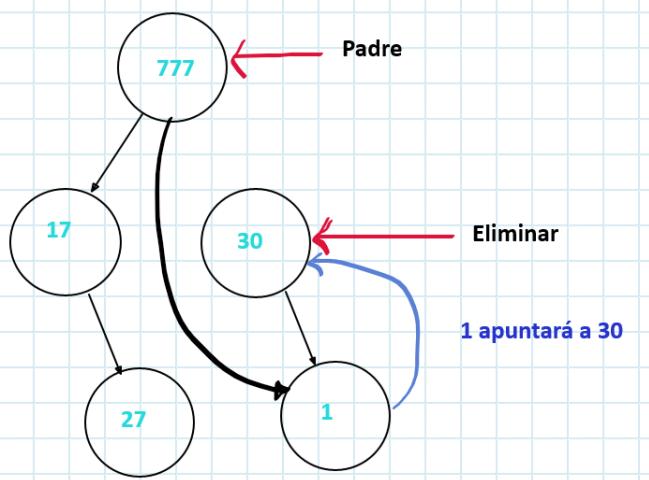
```

1.-CrearArbol
2.-Agregar Dato
3.-Eliminar dato
4.-Imprimir Arbol
5.-Notacion Prefija
6.-Notacion Infija
7.-Notacion Posfija
8-Salir
Opcion: 3
Ingrese el valor a eliminar: 30
30 es hijo derecho de 777
El padre del nodo a eliminar es: 777
**Intercambio**
30 es hijo derecho de 1
El padre del nodo a eliminar es: 1
El nodo 30 ya es hoja
Eliminacion de hoja
Nodo padre 1 de la hoja 30
EL nodo eliminado es: 30

```



Se hace la conexión del parent con el hijo del nodo a eliminar



Impresión

-----Menu Arbol Binario-----

```

1.-CrearArbol
2.-Agregar Dato
3.-Eliminar dato
4.-Imprimir Arbol
5.-Notacion Prefija
6.-Notacion Infija
7.-Notacion Posfija
8-Salir
Opcion: 4
Recorrido por BFS del arbol: 777 17 1 27

```

Menú Árbol Binario de Búsqueda

Crear Arbol e Insertar elemento

-----MENU PRINCIPAL-----

1.-Arboles Binarios
2.-Arboles Binarios de Busqueda
3.-Salir
Opcion: 2

-----MENU ARBOL BINARIO BUSQUEDA-----

1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 1
Ingrese el valor del nodo raiz: 20
ARBOL CREADO CON EXITO

-----MENU ARBOL BINARIO BUSQUEDA-----

1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 2
Ingresa el valor del nodo a agregar: 17
Ingreso exitoso

-----MENU ARBOL BINARIO BUSQUEDA-----

1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 2
Ingresa el valor del nodo a agregar: 23
Ingreso exitoso

-----MENU ARBOL BINARIO BUSQUEDA-----

1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 2
Ingresa el valor del nodo a agregar: 40
Ingreso exitoso

-----MENU ARBOL BINARIO BUSQUEDA-----

1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 2
Ingresa el valor del nodo a agregar: 24
Ingreso exitoso

-----MENU ARBOL BINARIO BUSQUEDA-----

1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 2
Ingresa el valor del nodo a agregar: 1
Ingreso exitoso

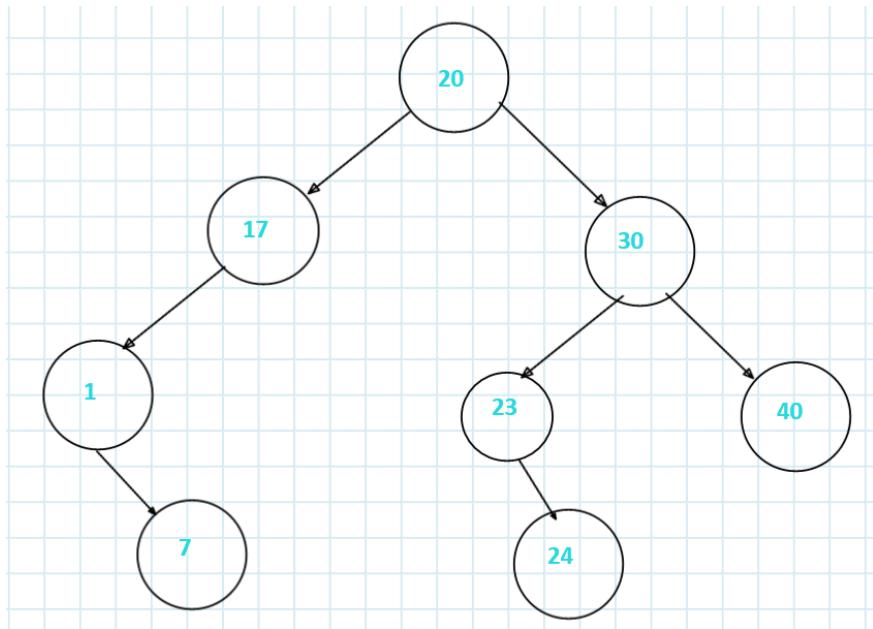
-----MENU ARBOL BINARIO BUSQUEDA-----

1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 2
Ingresa el valor del nodo a agregar: 7
Ingreso exitoso

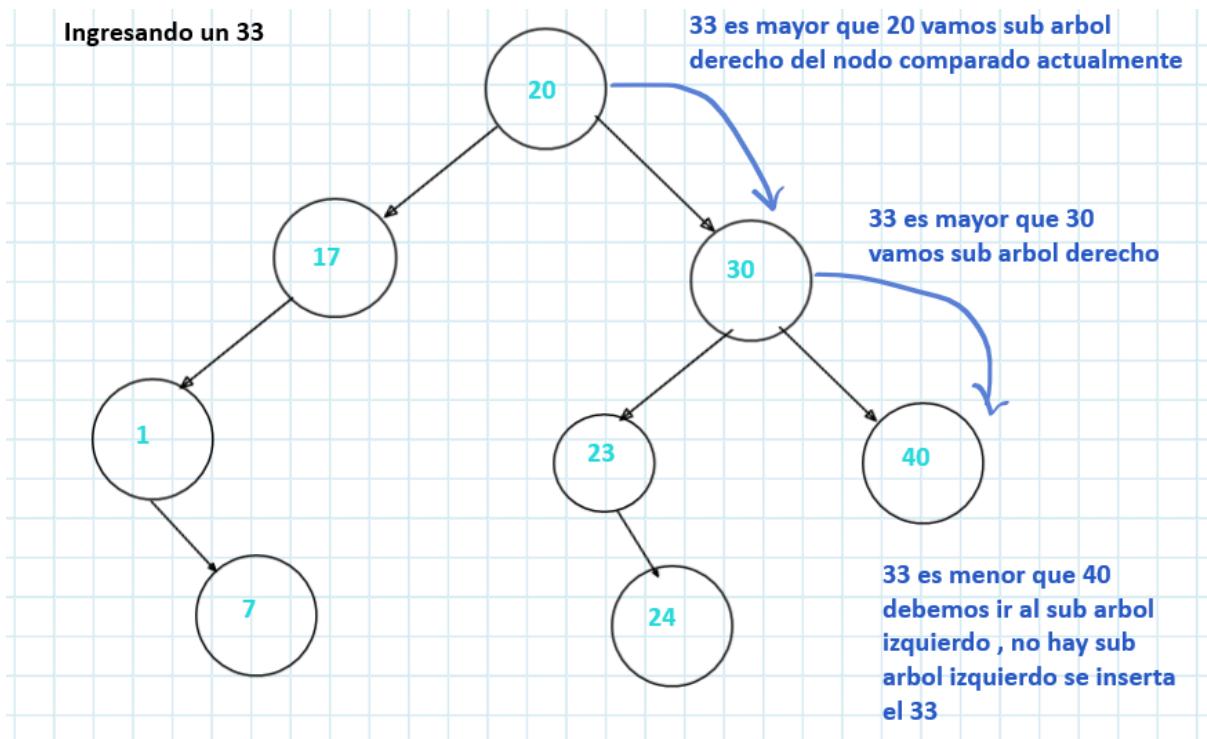
-----MENU ARBOL BINARIO BUSQUEDA-----

1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 2
Ingresa el valor del nodo a agregar: 30
Ingreso exitoso

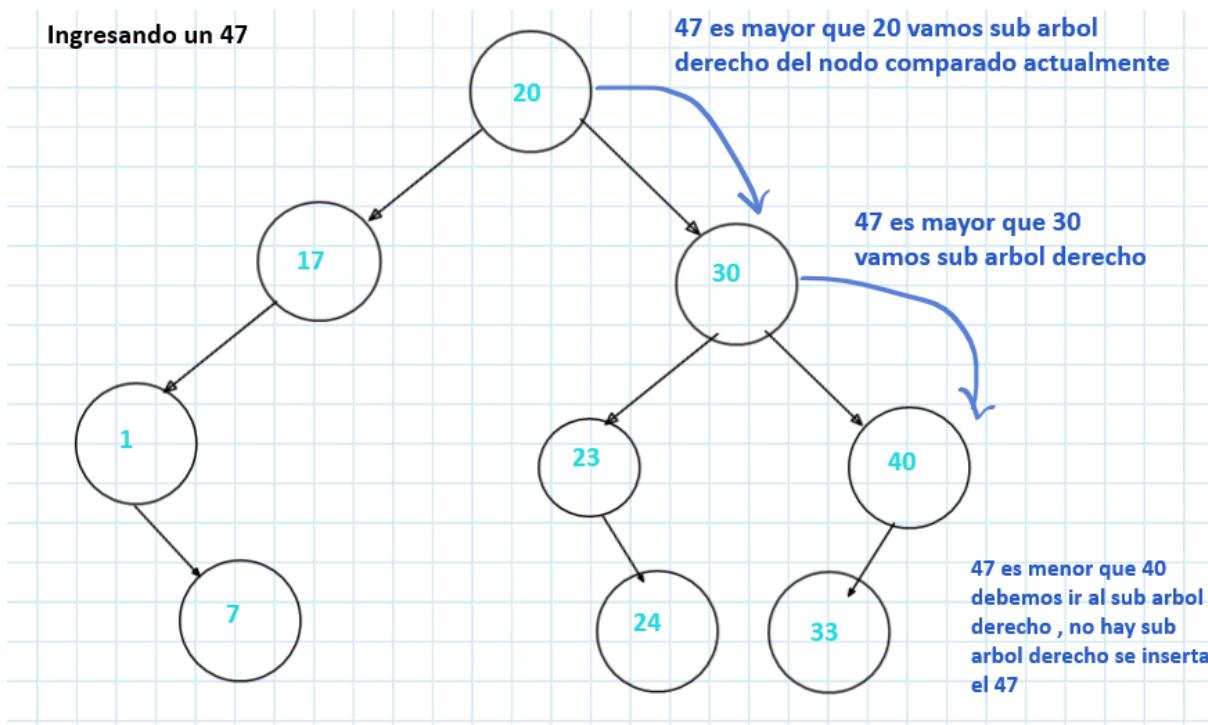
Hasta este momento lo que se lleva es el siguiente árbol



Si ingresamos un 33 internamente el proceso de insertar realiza lo siguiente



Si ahora ingresamos un 47



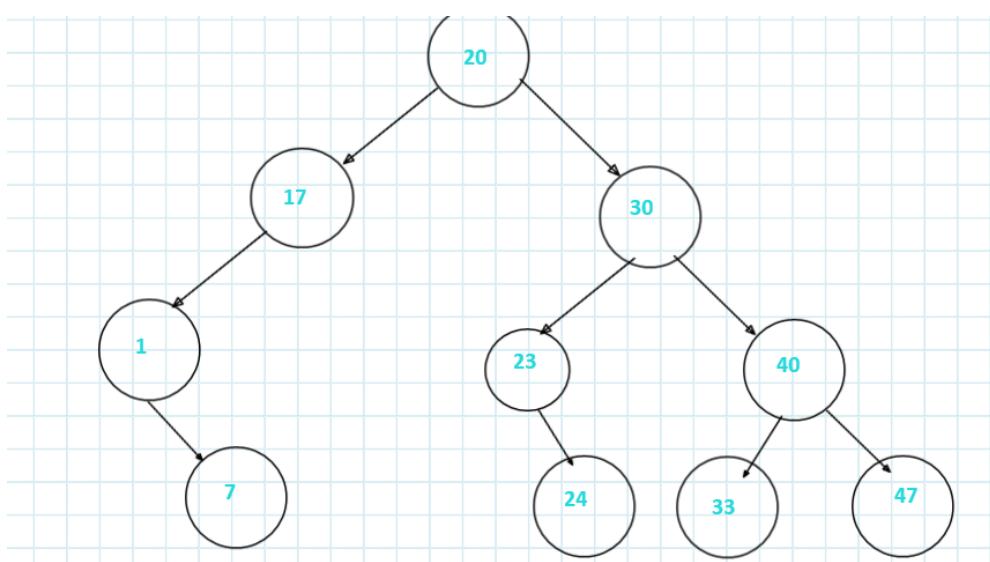
Impresión

De modo que al final tenemos el siguiente árbol

-----MENU ARBOL BINARIO BUSQUEDA-----

- 1.-Crear Arbol
 - 2.-Agregar dato
 - 3.-Eliminar dato
 - 4.-Buscar
 - 5.-Imprimir Arbol
 - 6.-Salir
- Opcion: 5

Recorrido por BFS del arbol: 20 17 30 1 23 40 7 24 33 47



Búsqueda

Para el proceso de búsqueda es una idea similar al de añadir en la forma en que se recorre el árbol ya que si se llega a una hoja y esta no es el valor a buscar significa que el nodo no existe dentro del árbol

```
-----MENU ARBOL BINARIO BUSQUEDA-----
1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 4
Ingresa el valor del nodo a buscar:
7
El nodo a buscar existe dentro del arbol
```

```
-----MENU ARBOL BINARIO BUSQUEDA-----
1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 4
Ingresa el valor del nodo a buscar:
333
El nodo no existe
```

Eliminación

Eliminación de 30

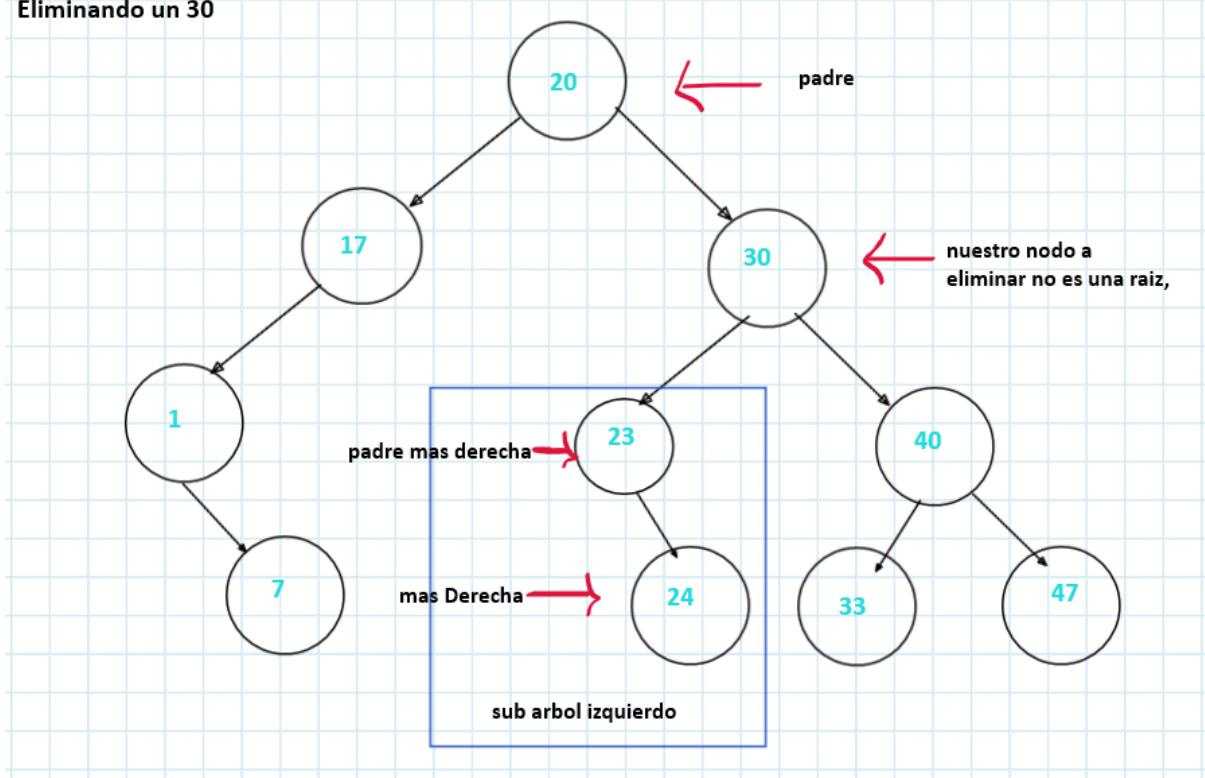
```
-----MENU ARBOL BINARIO BUSQUEDA-----
1.-Crear Arbol
2.-Agregar dato
3.-Eliminar dato
4.-Buscar
5.-Imprimir Arbol
6.-Salir
Opcion: 3
Ingresa el valor a eliminar: 30
Haciendo el proceso de busqueda
El padre de 30 es: 20
El nodo mas a la derecha del sub arbol izquierdo de 30 es 24
El padre 24 es: 23
Eliminacion de: 30
*****ANTES DEL INTERCAMBIO*****
Valor del padre
Valor: 20
Hijo izq: 17
Hijo der: 30
*****
Valor del padre mas a la derecha:
Valor: 23
Hijo izq nulo
Hijo der: 24
*****
Valor del nodo a eliminar
Valor: 30
Hijo izq: 23
Hijo der: 40
*****
```

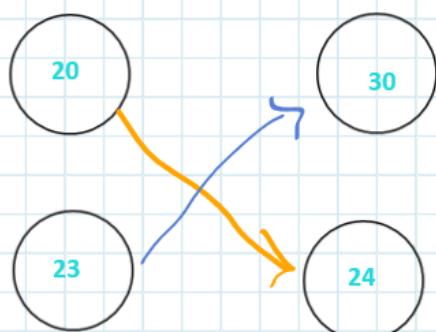
```

*****INTERCAMBIO DEL NODO MAS A LA DERECHA CON EL NODO A ELIMINAR*****
*****Despues DEL INTERCAMBIO*****
Valor del padre
Valor: 20
Hijo izq: 17
Hijo der: 24
*****
Valor del padre mas a la derecha:
Valor: 23
Hijo izq nulo
Hijo der: 30
*****
Valor del nodo a eliminar
Valor: 30
Hijo izq nulo
Hijo der nulo
*****
Valor del nodo mas a la derecha
Valor: 24
Hijo izq: 23
Hijo der: 40
*****
30 es hijo derecho de 23
El padre del nodo a eliminar es: 23
El nodo 30 ya es hoja
Valor: 30
Hijo izq nulo
Hijo der nulo
*****
Nodo padre 23 de la hoja 30
EL nodo eliminado es: 30

```

Eliminando un 30



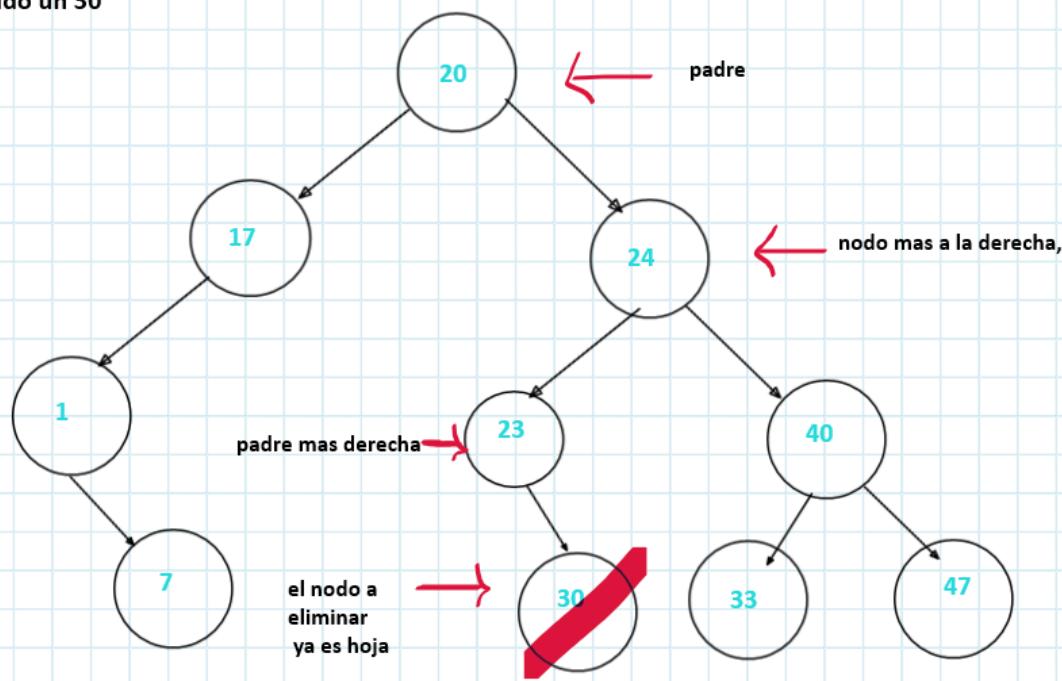


el padre apunta al nodo mas a la derecha

24 y 30 intercambian las referencias de los nodos a los que apuntan

el padre mas a la derecha apunta al nodo mas a eliminar

Eliminando un 30



-----MENU ARBOL BINARIO BUSQUEDA-----

- 1.-Crear Arbol
 - 2.-Agregar dato
 - 3.-Eliminar dato
 - 4.-Buscar
 - 5.-Imprimir Arbol
 - 6.-Salir
- Opcion: 5
Recorrido por BFS del arbol: 20 17 24 1 23 40 7 33 47

Ejercicio 5

Arboles B

Para la implementación de árboles proporcionada por el profesor se analizó primero la clase BNode en la cual se definen 5 atributos, m sera el numero de referencias, key es arreglo de valores enteros que serán las claves , child es un arreglo de referencias hacia otros nodos B, parent será el valor padre del nodo y por último leaf es el indicador de que nuestro nodo sea una hoja.

La clase BNode tiene un constructor el cual realiza las instancias de cada uno de los atributos anteriormente mencionados.

Contiene el método *getKey* que recibe como parámetro un entero llamado i para devolver el elemento del array list key encontrado en la posición i.

El método *getChild* devuelve un objeto BNode de acuerdo a la posición del valor pasado como parámetro

El método *setM* asigna al atributo m el valor pasado como parámetro, mismo caso sucede con *setChildren*

mostrarLlaves muestra las claves del nodo haciendo uso de la estructura repetitiva for

En la clase BTee tenemos dos atributos el primero es de tipo entero y se llama m el cual será el orden del árbol y un objeto BNode llamado root el cual será la raíz de todo el árbol. Dentro de esta clase tenemos los siguientes métodos:

Add: Recibe un entero que será la clave y busca si dicha clave no existe en el árbol, si esta afirmación es cierta consigue una hoja con el valor y es añadida al árbol.

leafNode(BNode nodo, int n) : Es un método recursivo que realiza el recorrido del arbol para encontrar un nodo hoja mediante el atributo leaf, cuando se encuentra este nodo es retornado y el proceso termina.

Insert(BNode nodo, int n): El método recibe como parámetros un objeto nodo que será en donde se inserte el valor entero n, dentro del método se realiza el proceso de búsqueda del índice en donde será colocado nuestro valor n.

addTreeNode(BNode nodo, int n): Este método nos ayuda a indicar si al querer ingresar una clave el árbol mantiene la estructura de un árbol B, es decir cada nodo tendrá como máximo un m referencias, en caso de que se cumpla se inserta el nodo

sin ningún problema en caso contrario, se procede a realizar el proceso de división celular.

divisionCelular: Realiza el proceso teórico revisado en clase, para que el árbol conserve la estructura de un Árbol B.

mostrarArbol: Realiza el proceso de impresión de las claves, una vez que se ha creado el árbol.

El método Find() tiene una sobrecarga el primer método recibe como parámetro un valor entero, su propósito es realizar la búsqueda de la clave pasada como parámetro para ello primero evalúa si los arraylist de referencias a otros nodos B y claves están vacíos, en caso de que esto suceda retorna un false, en caso contrario se hace uso del segundo método Find el cual es una función recursiva que recibe como parámetros un entero que será la clave y un objeto BNode llamado n, de esta forma busca si existe un nodo con el valor de la clave a buscar.

Ejercicio 6

Este ejercicio se completó al 100 sin ningún problema pues el nombre de los métodos es muy intuitivo para realizar cada una de las operaciones mencionadas, lo cual hace más sencilla su implementación que la misma interpretación realizada en el ejercicio 5.

Insertar e Imprimir

```
-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 1
Creacion del arbol de orden 5

-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 2
Ingrese el valor del nodo 20
nodo key size0

-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 2
Ingrese el valor del nodo 33
nodo key size1
```

```
-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 2
Ingrese el valor del nodo 12
nodo key size2

-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 2
Ingrese el valor del nodo 1
nodo key size3

-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 2
Ingrese el valor del nodo 14
```

```
-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 2
Ingrese el valor del nodo 9
nodo key size2
-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 4
do Raiz:
14
-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 2
Ingrese el valor del nodo 3
nodo key size3
do Padre: 14
Nodos:
1 3 9 12
20 33
```

Búsqueda

```
-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 3
Valor a buscar:
33
El valor existe dentro del arbol? true

-----Menu Arbol B-----
1.-Crear Arbol
2.-Agregar un valor
3.-Buscar un valor
4.-Imprimir Arbol
5.-Salir
Opcion: 3
Valor a buscar:
7777
El valor existe dentro del arbol? false
```

Conclusiones

Cada uno de los ejercicios se considera que se completaron de forma correcta , ya que en cada uno de ellos se desglosa un análisis ya sea del código proporcionado por el profesor o del código implementado con respecto a los árboles binarios,binarios de búsqueda y árboles , así como las muestras de ejecución y en algunos casos hasta comparaciones con respecto a ejercicios visto en clase, con lo cual se puede afirmar que se cumplieron los objetivos de la práctica.

Honestamente esta práctica ha sido aquella a la cual le he invertido más tiempo principalmente por el análisis del código de árboles B y por la implementación de la eliminación en árboles binarios y binarios de búsqueda, ya que como se mencionó previamente no sabemos quién es el padre de nuestro nodo a eliminar y esto es fundamental para el cambio de referencias. Con respecto a los árboles B la división celular es algo que honestamente de no ser que fue proporcionado el código dicha implementación por mi cuenta no hubiera sido posible.

La escala en la cual puede crecer esta práctica tiene un amplio margen ya que ahora que se está viendo el proyecto podemos agregar clases de los árboles AVL y Heaps heredando de la clase ArbolBin para así poder probar con más implementaciones de árboles.

PD: No me di cuenta sino hasta el final que para el menú de usuario pedía una clase :c una disculpa