



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Tista García Edgar Ing.

Asignatura: Estructura de Datos y Algoritmos II

Grupo: 5

No de Práctica(s): 8-9

Integrante(s): Téllez González Jorge Luis

*No. de Equipo de
cómputo empleado:* ---

No. de Lista o Brigada: X

Semestre: 2020-2

Fecha de entrega: 17/06/2020

Observaciones:

CALIFICACIÓN: _____



Índice

1. Introducción	2
2. Objetivos	4
3. Implementación del Árbol Binario	4
3.1. Análisis	4
3.1.1. Clase Nodo	5
3.1.2. Clase ArbolBin	5
3.2. Ejecución del ejemplo	7
3.3. Inserción	9
3.4. Búsqueda en el árbol	10
3.5. Ventajas y desventajas	11
4. Operaciones de un árbol binario	12
4.1. Notación Prefija	12
4.2. Notación Infija	13
4.3. Notación Posfija	13
4.4. Ejecución de los recorridos	14
5. Árboles binarios de búsqueda	15
5.1. Inserción	15
5.2. Eliminación	17
5.3. Impresión del árbol binario	19
6. Implementación de Árboles B	20
6.1. Análisis	20
6.1.1. Clase BNode	20
6.1.2. Clase BTree	21
6.2. Ejecución	25
7. Conclusiones	26

1. Introducción

Dentro del campo de la computación, una de las estructuras *no lineales* más importantes y con mayores implicaciones son los **árboles**. Su origen proviene de la *teoría de grafos*, ya que un árbol es un tipo especial de grafo con las siguientes características:

- Se trata de un grafo acíclico en el que cada nodo tiene 0 o más nodos hijos y como máximo un nodo padre.
- Únicamente la raíz (o también llamado *nodo superior*) no tiene *padre* y representa la base del árbol. Este nodo debe definirse previamente.

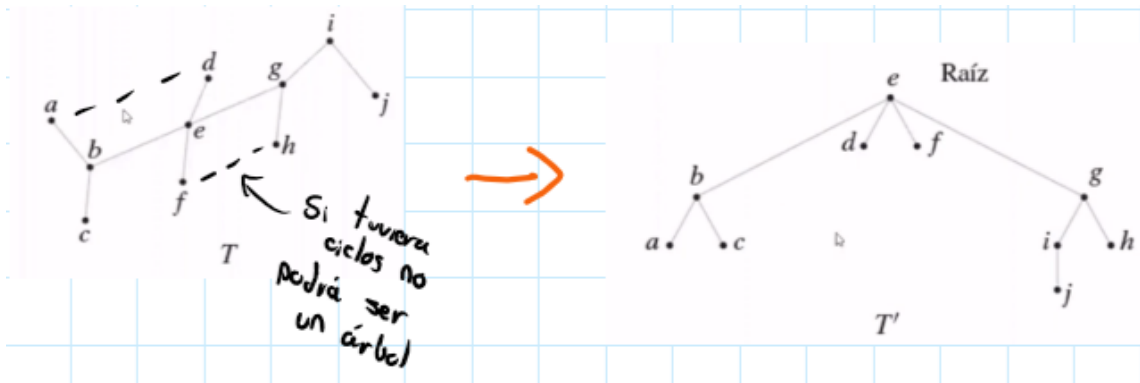


Figura 1: Comparación entre un grafo y su representación como árbol.

Una característica esencial de esta estructura es que, dependiendo del tipo de árbol, pueden tener una naturaleza *recursiva* en su interior. De esta forma, los nodos intermedios en un árbol son denominados como *sub-árboles* y los nodos inferiores (aquellos que no tienen hijos), se les conoce como *nodos hoja*.

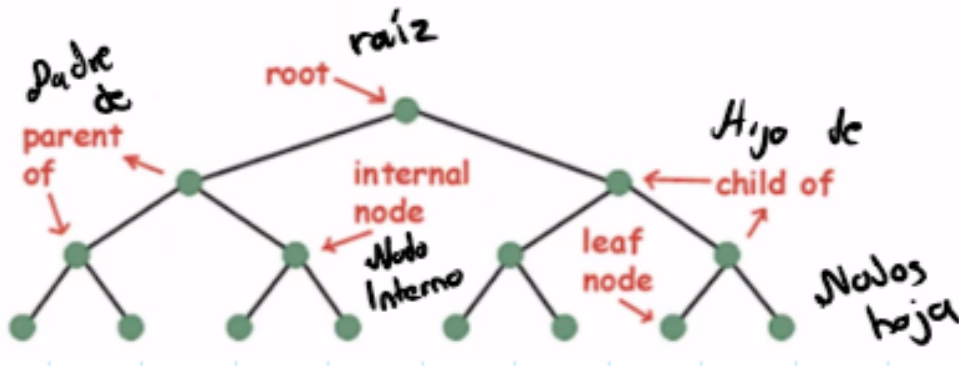


Figura 2: Elementos de un árbol cualquiera.

El uso de árboles generales o de n hijos puede resultar ser una idea árdua de trasladar a un entorno de programación. Por ello, surge el concepto de **árbol binario**, el cuál tiene las siguientes características:

- Todos los nodos tienen 2 hijos.
- Para cada nodo, uno de los hijos o ambos pueden ser de carácter nulo.

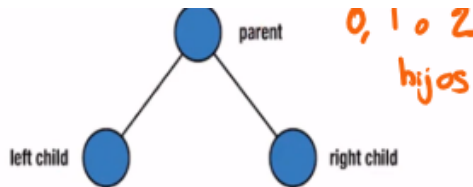


Figura 3: Árbol binario.

Un sub-tipo de esta clase de árbol corresponde al **binary search tree** o *árbol binario de búsqueda*. Este árbol tiene la característica de encontrarse *ordenado* y cada elemento contiene un identificador único. Siguiendo un orden ascendente, esta clase de árbol cumple con las siguientes condiciones:

- Los identificadores del subárbol izquierdo son **menores** al valor de la raíz.
- Los identificadores del subárbol derecho son **mayores** al valor de la raíz.
- Los subárboles izquierdo y derecho a su vez deben de ser árboles binarios de búsqueda y cumplir todas sus propiedades.

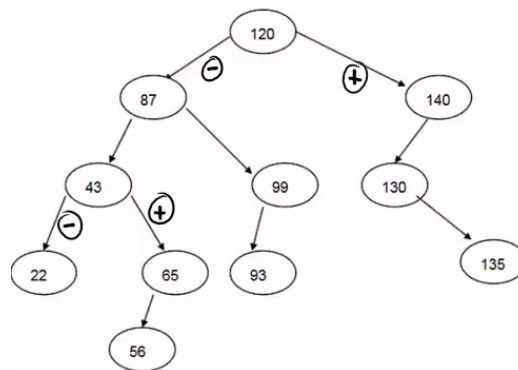


Figura 4: Para cada nodo, todo lo que está a su izquierda es menor y a su derecha, mayor.

Finalmente, otro tipo de árbol muy relevante por sus aplicaciones corresponde al **Árbol B**, definido por *Rudolf Bayer* y *EdMcCreight*. Estos árboles se caracterizan por tener nodos que pueden almacenar una o más claves al mismo tiempo. Entre sus características más relevantes se encuentran las siguientes:

- Conservan todas las propiedades definidas para un árbol binario.
- Es una extensión del concepto del *árbol binario de búsqueda*.

- No presentan claves repetidas.
- Representan una estructura de datos auto-balanceada.
- Permiten operaciones de inserción, eliminación y búsqueda con una complejidad de $O(\log(n))$.

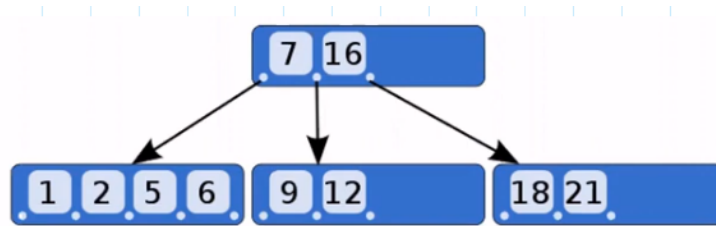


Figura 5: Árbol B de orden 5.

Los árboles B cumplen con la propiedad de ser una estructura *half-full*, es decir, dado un árbol de orden n , este tendrá como máximo n referencias, $n - 1$ claves en cada nodo y, además, cada nodo deberá de tener mínimo $n/2$ claves en su interior. Dicho de otra forma, todos los nodos deberán estar siempre a la mitad de su capacidad máxima de almacenamiento.

De forma general los árboles pueden tener las siguientes aplicaciones:

- Tienen un uso extendido en los *sistemas operativos* en la organización de carpetas y archivos.
- Pueden funcionar como estructuras de definición jerárquica para mostrar relaciones en registros de bases de datos.

En el siguiente reporte escrito se analizarán 2 implementaciones proporcionadas: árboles binarios y árboles B. Se escribirán las operaciones esenciales del árbol binario y, además, se implementará el uso de árboles binarios de búsqueda junto con sus operaciones esenciales.

2. Objetivos

- El estudiante conocerá e identificará las características de la estructura no-lineal Árbol (Árbol Binario, Binario de Búsqueda y Árbol B).

3. Implementación del Árbol Binario

3.1. Análisis

El código proporcionado que modela el concepto de árbol binario tiene 2 clases esenciales en su interior:

3.1.1. Clase Nodo

La clase *Nodo* se encarga de modelar los nodos y sus propiedades principales en un árbol binario. Contiene como atributos la variable entera **valor** que representa el valor contenido en el nodo creado y dos referencias nulas a objetos de tipo *Nodo*; llamándose estas referencias **izq** y **der**, respectivamente.

```
class Nodo {  
  
    private int valor;  
    private Nodo izq = null;  
    private Nodo der = null;  
}
```

Figura 6: Atributos de la clase *Nodo*.

En su interior contiene su respectivo método constructor, el cual se encuentra sobrecargado con otros 2 métodos más.

- El primer constructor no recibe ningún parámetro e inicializa de nuevo las referencias **izq** y **der** como **null**.
- El segundo constructor recibe como parámetro un entero llamado **data** y, tomando ese valor, lo envía al tercer constructor de la clase.
- El último constructor recibe como parámetro un entero **data**, un objeto de tipo *Nodo* **lt** que representará el hijo izquierdo del nodo creado y otro objeto de tipo *Nodo* **rt** que representará a su hijo derecho.

Con fines del desarrollo de las actividades de la práctica, los atributos han sido *encapsulados* previamente y se han establecido los correspondientes métodos **get & set**.

3.1.2. Clase ArbolBin

La clase *ArbolBin* es la encargada de realizar el modelado de los árboles binarios en el programa. Entre sus atributos se encuentra una referencia a un objeto *Nodo* denominado **root** que representa el nodo raíz del árbol en cuestión y un objeto *Scanner* declarado e inicializado. A continuación se detallarán los métodos que componen a esta clase:

- **Métodos constructores:** Esta clase cuenta con un total de tres constructores en su interior.
 1. El primero no recibe ningún parámetro e inicializa la referencia al objeto **root** como **null**.
 2. El segundo recibe un valor entero y establece la referencia al objeto **root** como un nuevo *Nodo* inicializado con el valor recibido como parámetro.

3. El tercero recibe directamente un *Nodo* inicializado previamente y lo asigna como el nodo **root** del árbol inicializado.

```
public ArbolBin() {  
    root = null;  
}  
  
public ArbolBin(int val) {  
    root = new Nodo(val);  
}  
  
public ArbolBin(Nodo root) {  
    this.root = root;  
}
```

Figura 7: Métodos constructores.

- **Métodos de Encapsulamiento:** se establecen métodos **get & set** para el nodo **root** del árbol binario.
- **add:** Este método es utilizado para añadir nuevos nodos a un árbol binario cualquiera. Recibe como parámetros el *Nodo padre* donde será insertado un nodo cualquiera, el *nodo hijo* que corresponde al nodo que se desea insertar y el lado, es decir, si se desea establecer como un hijo izquierdo o derecho del nodo padre al que se hace referencia.

```
if (lado == 0) {  
    padre.setIzq(hijo);  
}  
else {  
    padre.setDer(hijo);  
}
```

Figura 8: Condición de inserción como hijo izquierdo o derecho.

- **breadthFirst:** El método de búsqueda por capas utilizado para los grafos también puede ser utilizado para recorrer árboles binarios. De forma similar al algoritmo revisado en la práctica previa, se inicia obteniendo el nodo actual (es decir, el nodo **root** para iniciar a recorrer desde la raíz) por medio del método **getRoot**. Posteriormente, se inicializa una cola ligada de tipo *Nodo* como requisito previo para BF.

Una vez que se verifica que el nodo **root** no sea nulo, se agrega a la cola y, mientras esta cola no

```
protected void visit(Nodo n) {  
  
    System.out.println(n.getValor() + " ");  
}
```

Figura 9: Método *visit* para imprimir los nodos que han sido visitados y registrados por *BF*.

se encuentre vacía, se saca el nodo **root** de la cola y se marca como *visitado*, imprimiéndolo por medio del método **visit** y recuperando su valor. Posterior a esto, se verifica que los hijos izquierdo y derecho del nodo **root** no sean nulos, y de cumplirse lo anterior, son añadidos a la cola y el proceso de expansión **BFS** continuará con estos nodos hasta que el árbol sea recorrido por completo.

```
if (r.getIzq() != null) {  
    queue.add(r.getIzq());  
}  
  
if (r.getDer() != null) {  
    queue.add(r.getDer());  
}
```

Figura 10: Recuperación de los nodos hijos izquierdo y derecho del nodo actual.

3.2. Ejecución del ejemplo

Para realizar la ejecución del programa se ha creado un menú que permite seleccionar entre los diferentes tipos de árboles a operar.

```
Run:  
Practica #8: Arboles  
  
¿Sobre que tipo de arbol desea trabajar?  
1)Arboles binarios.  
2)Arboles binarios de busqueda.  
3)Ejemplo de la implementacion de arboles binarios.  
4)Arboles B  
5)Salir del programa.  
  
Opcion:
```

Figura 11: Menú principal.

El ejemplo proporcionado crea e inicializa manualmente un total de 9 nodos. Además, el nodo *n1* es inicializado directamente como el padre de los nodos *n7* (izquierdo) y *n9* (derecho). Posteriormente, se inicializa un árbol binario estableciendo como **root** a *n1*. Finalmente, se utiliza el método **add** en repetidas ocasiones para realizar las siguientes operaciones:

- Se inserta a *n15* como el hijo izquierdo de *n7*.
- Se inserta a *n8* como el hijo derecho de *n7*.

- Se inserta a $n4$ como el hijo izquierdo de $n9$.
- Se inserta a $n2$ como el hijo derecho de $n9$.
- Se inserta a $n16$ como el hijo derecho de $n15$.
- Se inserta a $n3$ como el hijo izquierdo de $n8$.

Finalmente, se realiza e imprime el recorrido del árbol creado por medio de **breadthFirst**.

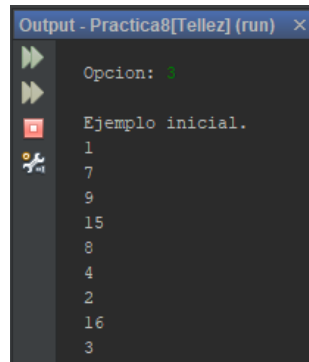


Figura 12: Salida obtenida con *breadthFirst*.

Con fines de utilidad, se han añadido a la práctica tres métodos denominados **traversePreOrder**, **traverseNodes** y **print** que han sido adaptados para funcionar con la implementación proporcionada. Su utilidad es imprimir gráficamente un árbol recibido como parámetro por medio del método **print** que se encuentra ligado directamente al mismo. Estos métodos fueron recuperados de *Baeldung*; el enlace a la fuente original y su construcción se encuentra en las referencias de la práctica actual.

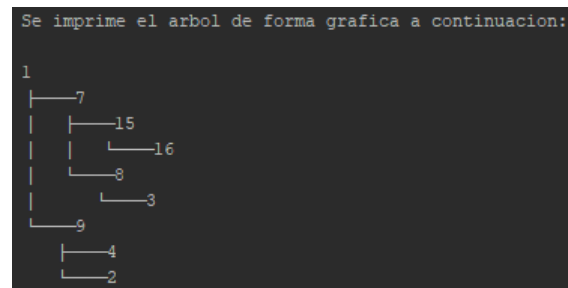


Figura 13: Impresión gráfica del árbol con los métodos adaptados.

A continuación, y tomando como referencia las operaciones descritas con anterioridad, se dibujará manualmente el árbol creado en el ejemplo proporcionado. El resultado se encuentra en concordancia con la salida gráfica mostrada por el programa realizado.

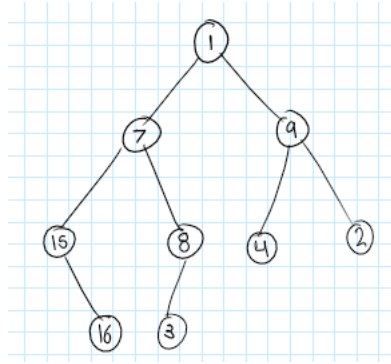


Figura 14: Dibujo del grafo creado.

3.3. Inserción

La inserción de nodos puede resultar problemática de implementar con el método proporcionado para la inserción de nodos. Por tanto, se ha optado por realizar un método recursivo que sea capaz de generar un árbol binario únicamente insertando valores como entradas. Este método sobrecarga al método **add** y, a diferencia del método presente en la implementación proporcionada, recibe al nodo padre o **root** del árbol y el valor que se desea insertar y realiza los siguientes pasos:

1. Verifica en primer lugar si el hijo izquierdo del nodo **root** es nulo. En ese caso, crea un nuevo nodo con el valor recibido, lo inserta como su hijo izquierdo y termina su ejecución. En caso de que esta referencia no sea nula, realiza el mismo procedimiento pero ahora tratando de insertar el nodo creado como el hijo derecho del **root**.
2. En caso de que el **root** ya tenga asignados hijos izquierdo y derecho, se hará una ponderación de los valores de los hijos del nodo **root**: si el valor del hijo izquierdo es menor al valor del hijo derecho, se hará una llamada recursiva del método **add** pero ahora utilizando como nodo *padre* al hijo izquierdo del root actual. En caso de que lo anterior no se cumpla, la llamada recursiva se hará con el hijo derecho.

```
if ((padre.getIzq()).getValor() < (padre.getDer()).getValor()) {  
    add(padre.getIzq(), dato);  
    return 0;  
} else {  
    add(padre.getDer(), dato);  
    return 0;  
}
```

Figura 15: Ponderación del valor almacenado entre los nodos izquierdo y derecho.

Este procedimiento es utilizado para asegurar que jamás se dará el caso de que un nodo tenga más de 2 hijos. Así, solo se podrán dar los tres casos de un árbol binario: que un nodo tenga 0, 1 o 2 hijos como máximo.

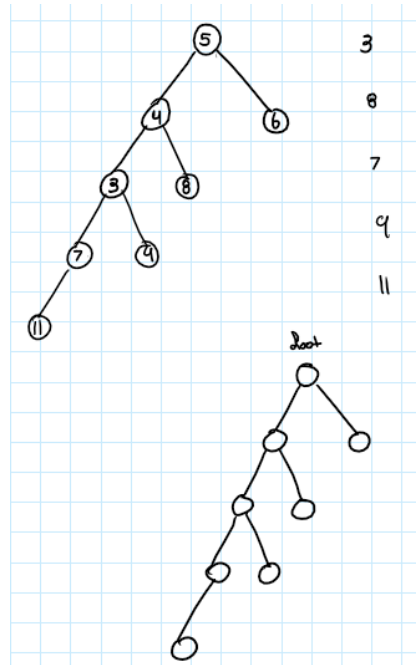


Figura 16: Estructura general de los árboles binarios creados con la inserción añadida.

3.4. Búsqueda en el árbol

Para realizar la búsqueda de un nodo en específico, se ha utilizado el recorrido por capas implementado por el método **breadthFirst**. El procedimiento a utilizar por el nuevo método denominado **breadthFirstSearch**, que recibe como parámetro un entero que representa el valor a buscar en el árbol, es el siguiente:

1. Se establece como el nodo inicial al **root** del árbol y se inicializa una cola ligada. Verificando que el **root** recibido no sea nulo, verifica en primer lugar si el nodo **root** tiene un valor idéntico al valor buscando en su atributo **valor** utilizando el método **get** establecido para el atributo. De darse lo anterior, se imprime que el nodo ha sido hallado con éxito y el método devuelve **true**.

```
if (r.getValor() == valorABuscar) {
    System.out.println("El nodo " + valorABuscar + " SI se ha encontrado en el arbol.\n");
    return true;
}
```

Figura 17: Caso inicial de búsqueda.

2. En caso de que lo anterior no se cumpla, se añade el nodo actual (en el caso inicial, el **root**), a la cola y, mientras la condición del ciclo **while** de que la cola no esté vacía se cumpla, se extrae el nodo actual y se compara su valor con el valor buscado. De ser los mismos, se imprime el éxito de la búsqueda y el método devuelve **true**.

3. Si la condición anterior tampoco se cumple, se verifica que los hijos izquierdo y derecho del nodo actual no sean nulos, y verificando eso, los añade a la cola usando los métodos **getIzq** y **getDer**.
4. Recordando que el ciclo **while** se repetirá para realizar la búsqueda por capas en el árbol, el procedimiento anterior se repetirá con los hijos que sean agregados sucesivamente a la cola. Si después de todo ese procedimiento se llega al caso donde se ha llegado a la última capa del árbol y no se ha hallado el valor buscado, se imprimirá un mensaje en pantalla indicando que el nodo no ha sido hallado en el árbol.

Para probar la efectividad de este método, se ha utilizado sobre el árbol creado en el ejemplo proporcionado, buscando los valores 15 y 19 en el árbol. Evidentemente, el resultado será positivo para el 15 y negativo para el 19.

```
Se imprime el arbol de forma grafica a continuacion:

1
├──7
│  ├──15
│  │  └──16
│  │  └──8
│  │  └──3
│  └──9
│     ├──4
│     └──2
└──

¿Se encuentre el nodo 15 en el arbol anterior?

El nodo 15 SI se ha encontrado en el arbol.

¿Y el nodo 19?

El nodo 19 NO se ha encontrado en el arbol.
```

Figura 18: *Búsqueda en el árbol de ejemplo.*

3.5. Ventajas y desventajas

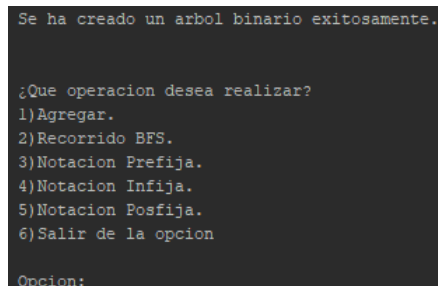
Esta implementación, aunque resulta relativamente sencilla de comprender y no está sobrecargada de elementos que dificulten su comprensión, me generó conflictos que no pude sortear del todo:

- No encontré una manera clara de utilizar el método **add** para introducir la inserción en el menú principal, por lo que opté por sobrecargar el método y realizar una inserción alternativa que cumpliera con los requisitos de un árbol binario.
- Tuve conflictos para realizar la eliminación exitosamente, y debido a otros factores ajenos a mi persona, no pude completar exitosamente una propuesta para implementar la eliminación; por lo que tuve que prescindir de ella en el menú.

El método BF, por su salida, puede resultar confuso de comprender. Por ello es que se agregaron los métodos gráficos para imprimir el árbol generado y tener una mejor idea del funcionamiento de la implementación. Considero que hay un gran margen de mejora para el funcionamiento de la implementación con el fin de hacerla todavía más sencilla de utilizar. Más allá de lo comentado, no encontré más detalles negativos que pueda destacar.

4. Operaciones de un árbol binario

Dentro del menú dedicado a los árboles binarios se han añadido tres opciones para permitir visualizar el recorrido del árbol en notación Prefija, Infija y Posfija; según desee el usuario.



```
Se ha creado un arbol binario exitosamente.

¿Que operacion desea realizar?
1)Agregar.
2)Recorrido BFS.
3)Notacion Prefija.
4)Notacion Infija.
5)Notacion Posfija.
6)Salir de la opcion

Opcion:
```

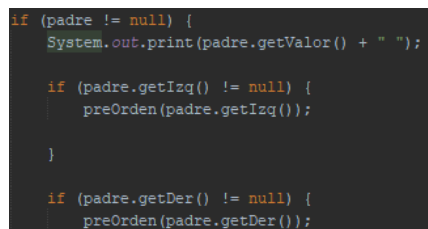
Figura 19: *Búsqueda en el árbol de ejemplo.*

Los métodos siguen una estructura similar y han sido integrados a la clase *ArbolBin*. A continuación, se detalla su funcionamiento.

4.1. Notación Prefija

Para realizar un recorrido con notación Prefija, se han implementado 2 métodos:

- **preOrden:** Este método recibe al nodo padre de un árbol cualquiera. Si el nodo padre recibido no es nulo, se imprime el valor del padre en pantalla y, posteriormente, se realiza una llamada recursiva del método pero ahora con los hijos izquierdo y derecho del nodo padre (o el nodo actual); siempre y cuando los hijos no tengan referencias nulas.



```
if (padre != null) {
    System.out.print(padre.getValor() + " ");

    if (padre.getIzq() != null) {
        preOrden(padre.getIzq());
    }

    if (padre.getDer() != null) {
        preOrden(padre.getDer());
    }
}
```

Figura 20: *Recorrido en Preorden.*

- **printPreOrden:** Este método que no recibe parámetros es utilizado para imprimir directamente el recorrido en preorden del árbol sobre el que se invoque este método. Para ello, verifica que el nodo **root** del árbol no sea nulo y, finalmente, llama al método **preOrden** enviando como parámetro el nodo **root** del árbol, así, asegurando que el recorrido se inicia en la raíz.

```
if (this.root != null) {  
    preOrden(this.root);  
}  
System.out.println("");
```

Figura 21: Método de impresión en Preorden.

4.2. Notación Infija

Para el recorrido en Inorden, se ha cambiado de lugar la línea de impresión para que ahora esté ubicada tras la finalización de la llamada recursiva hacia los nodos *izquierdos* del árbol, de esta forma, siguiendo las pautas teóricas estudiadas en clase.

```
if (padre.getIzq() != null) {  
    inOrden(padre.getIzq());  
}  
  
System.out.print(padre.getValor() + " ");  
  
if (padre.getDer() != null) {  
    inOrden(padre.getDer());  
}
```

Figura 22: Recorrido en Inorden.

El método para imprimir este recorrido en el árbol que lo invoque sigue exactamente la misma estructura que el mostrado en la notación Prefija.

4.3. Notación Posfija

Para este último caso, la línea de impresión ha sido cambiada de lugar hasta el final de las llamadas recursivas hacia los hijos izquierdos y derechos del árbol. De esta forma, se siguen las 3 reglas establecidas para el recorrido posfijo del árbol deseado.

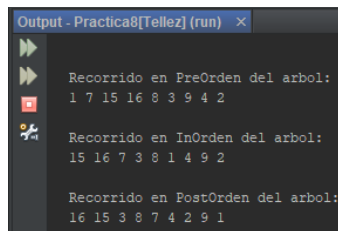
Al igual que el caso anterior, la impresión se realiza con otro método que inicializa el recorrido en Posorden desde el nodo **root** del árbol que lo invoca.

```
if (padre.getIzq() != null) {  
    postOrden(padre.getIzq());  
}  
  
if (padre.getDer() != null) {  
    postOrden(padre.getDer());  
}  
  
System.out.print(padre.getValor() + " ");
```

Figura 23: *Recorrido en Posorden.*

4.4. Ejecución de los recorridos

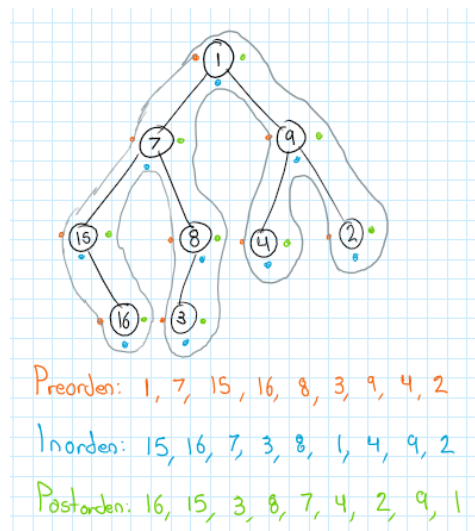
Para verificar el funcionamiento de los métodos implementados, se han utilizado los métodos sobre el árbol proporcionado como ejemplo, generando las siguientes salidas:



```
Output - Practica8[Tellez] (run) x  
Recorrido en PreOrden del arbol:  
1 7 15 16 8 3 9 4 2  
Recorrido en InOrden del arbol:  
15 16 7 3 8 1 4 9 2  
Recorrido en PostOrden del arbol:  
16 15 3 8 7 4 2 9 1
```

Figura 24: *Recorridos en Preorden, Inorden y Postorden sobre el árbol del ejemplo proporcionado.*

La salida obtenida coincide con el recorrido realizado manualmente del árbol:

Figura 25: *Recorridos en Preorden, Inorden y Postorden sobre el árbol manualmente.*

5. Árboles binarios de búsqueda

La implementación de árboles binarios de búsqueda propuesta para esta práctica contiene las siguientes opciones:

```
Se ha creado un arbol binario de busqueda exitosamente.

¿Qué operacion desea realizar?
1)Agregar.
2)Eliminar.
3)Buscar.
4)Imprimir arbol (BFS).
5)Salir de la opcion

Opcion:
```

Figura 26: Menú de opciones de los Binary Search Trees.

La clase que implementa esta clase de árboles se denomina *ArbolBinBus* y hereda de la clase *ArbolBin*. A continuación, se detallarán las operaciones implementadas.

5.1. Inserción

Para realizar la inserción en un BST se ha implementado un método llamado **addR**, el cual recibe al nodo **root** del árbol y el valor que se desea insertar en el mismo. Se denotan cuatro casos principales para la inserción con este método:

1. El caso base del método recursivo establece que, si la referencia al nodo *padre* es nula, se retorna *new Nodo(dato)*; creando el nodo de referencia.
2. De no cumplirse la condición del caso base, el procedimiento consistirá en verificar que si el dato es menor al valor del nodo padre actual. Si esto se cumple, se establecerá como hijo izquierdo **una llamada recursiva al método addR, enviándole al hijo izquierdo del padre actual y el mismo valor a insertar como parámetros**. Esta forma de insertar retoma la idea de los BST como estructuras recursivas llevándola a su máxima expresión, ya que permite recorrer el árbol sucesivamente hasta encontrar la posición correcta para insertar un nodo en el árbol.

```
//Si el valor del nuevo nodo es menor que el nodo actual, se procede a revisar a su hijo izquierdo.
if (dato < padre.getValor()) {
    padre.setIzq(addR(padre.getIzq(), dato));

    //Si el valor del nuevo nodo es mayor que el nodo actual, se procede a revisar al hijo derecho.
} else if (dato > padre.getValor()) {
    padre.setDer(addR(padre.getDer(), dato));
```

Figura 27: Inserción recursiva por la izquierda o derecha, según sea el caso.

3. Si la condición anterior no se cumple, es decir, el dato a insertar supera en valor al nodo *padre*, ahora la inserción recursiva será realizada sobre el hijo derecho del padre actual. Finalmente, si el valor ya existe en el árbol, se regresa al nodo *padre*.

Para utilizar esta operación se llaman dos métodos:

- **add**: Ejecuta el método **addR** asignando su salida al nodo **root** del árbol con el fin de que la inserción sobre el árbol sea exitosa.

```
public void add(int value) {  
    root = addR(root, value);  
}
```

Figura 28: Método *add* reimplementado.

- **agregarDato**: Por medio de este método se le solicita al usuario que ingrese el valor que desea introducir en el árbol actual, permitiéndole agregar más datos si así lo desea o salir de la operación. Cabe señalar que, al momento de seleccionar la operación de inserción, se le solicitará al usuario que introduzca un valor que representará al nodo raíz del árbol BST creado.

```
Opcion: |  
  
El arbol no contiene una raiz. A continuacion, introduce un valor para su raiz: |  
  
A continuacion, introduzca un valor para el nodo: |  
  
¿Deseas agregar más datos?  
1) Si  
2) No  
Opcion: |
```

Figura 29: Operación de inserción en el programa creado.

Como referencia para las siguientes operaciones, se ha creado el siguiente árbol utilizando la implementación descrita anteriormente:

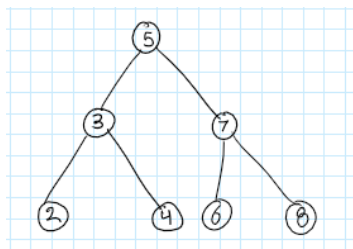


Figura 30: Árbol binario de búsqueda creado.

5.2. Eliminación

Al igual que la inserción, la eliminación debe de observarse como un proceso *recursivo* que requiere tener varias consideraciones adicionales (especialmente, en el caso de eliminación al tener un nodo con dos hijos) para mantener la integridad del árbol. El método **deleteR** implementado, que recibe al nodo *padre* y el valor a eliminar, está conformado por varios casos que serán detallados a continuación.

- El caso base de este método establece que, si la referencia al padre es nula, simplemente se devolverá al padre de salida del método.
- En caso contrario (el árbol no está vacío), se comenzará a realizar un recorrido recursivo sobre el árbol para hallar el valor a eliminar, comparando el valor a eliminar con el valor del nodo *padre* o el nodo actual. Dependiendo del caso (el valor es superior o es menor al valor del nodo padre), se realizará una asignación recursiva al nodo padre por la izquierda o derecha (según sea el caso) asignando la salida de la llamada recursiva del método **deleteR** pero ahora enviando como nodo *padre* al hijo izquierdo o derecho del nodo actual, según corresponda.

```
/* Caso contrario, se realiza un recorrido recursivo sobre el árbol. */  
if (valorAEliminar < padre.getValor()) {  
    padre.setIzq(deleteR(padre.getIzq(), valorAEliminar));  
  
} else if (valorAEliminar > padre.getValor()) {  
    padre.setDer(deleteR(padre.getDer(), valorAEliminar));  
}
```

Figura 31: *Recorrido recursivo.*

- Si resulta que, en una de las llamadas recursivas, el valor del nodo padre coincide con el valor a eliminar, implica que finalmente se ha hallado el valor a eliminar en el árbol. Entonces, se verificará el primer caso: un nodo a eliminar sin hijos. En caso de que las referencias a los padres izquierdo y derecho del nodo actual sean nulas, se devolverá null como resultado; sobrescribiendo y eliminando el nodo del árbol.

```
else {  
    // Nodo con un único hijo o sin hijo.  
  
    if (padre.getIzq()==null && padre.getDer()==null) {  
        return null;  
    }  
}
```

Figura 32: *Primer caso: nodo sin hijos.*

- El segundo caso involucra que el nodo tiene un único hijo: Si la referencia al hijo izquierdo del padre es nula, se devolverá al hijo derecho y será asignado en el lugar del padre para mantener la integridad

del árbol. Inversamente, si la referencia al hijo derecho es nula, el hijo izquierdo tomará su lugar en el árbol.

```
if (padre.getIzq() == null) {
    return padre.getDer();
} else if (padre.getDer() == null) {
    return padre.getIzq();
}
```

Figura 33: Segundo caso: nodo con un hijo.

- El último caso representa el caso cuando se tiene un nodo con 2 hijos. Para ello, se obtendrá al nodo sucesor de reemplazo recorriendo lo más que se pueda a la derecha a partir del subárbol izquierdo dado por el hijo izquierdo del nodo raíz. Para encontrar este nodo, se utiliza el método llamado **maxValor** que, recibiendo como parámetro al hijo izquierdo del árbol actual, empieza a recorrer sucesivamente por la derecha el subárbol izquierdo hasta hallar el máximo nodo por la derecha.

```
int maxv = root.getValor();
while (root.getDer() != null) {
    maxv = root.getDer().getValor();
    root = root.getDer();
}
System.out.println("Valor maximo del subarbol izq: " +maxv);
return maxv;
```

Figura 34: Búsqueda del sucesor.

Obtenido el valor máximo, se utiliza la salida del método para establecer el valor del padre actual y finalmente, se elimina el nodo por medio de otra llamada recursiva desde el suárbol izquierdo.

```
// Nodo con 2 hijos: se obtiene al sucesor recorriendo
// a la derecha a partir del subárbol izquierdo.
padre.setValor(maxValor(padre.getIzq()));

// Se reordena el árbol.
padre.setIzq(deleteR(padre.getIzq(), padre.getValor()));
```

Figura 35: Reemplazo y eliminación recursiva.

Al igual que el caso de la inserción, se han implementado otros dos métodos para acceder a la operación de eliminación desde el menú de usuario. Con motivos de prueba, en el árbol creado en la sección anterior se ha eliminado el 5 del árbol, quedando el resultado gráficamente de la siguiente forma:

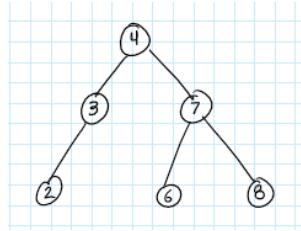


Figura 36: Eliminación del 5 en el árbol creado previamente.

5.3. Impresión del árbol binario

Finalmente, para verificar el funcionamiento de las operaciones implementadas, se ha optado por añadir en esta opción el recorrido por BF, la impresión gráfica del árbol e incluir los 3 recorridos sobre el árbol para validar su creación exitosa. La salida obtenida es la siguiente:

```
La impresión del árbol actual por BFS es la siguiente:
4
3
7
2
6
8

Impresión gráfica del árbol actual:

4
├── 3
│   ├── 2
│   └── 7
└── 7
    ├── 6
    └── 8

El recorrido en PreOrden es: 4 3 2 7 6 8

El recorrido en InOrden es: 2 3 4 6 7 8

El recorrido en PostOrden es: 2 3 6 8 7 4
```

Figura 37: Impresión de los datos del árbol creado.

El resultado obtenido permite validar las operaciones escritas para los árboles binarios de búsqueda exitosamente, pues la salida corresponde con los ejemplos gráficos mostrados en las secciones previas.

6. Implementación de Árboles B

La implementación proporcionada de árboles B consta de dos clases principales: **BNode** y **BTree**.

6.1. Análisis

6.1.1. Clase BNode

Esta clase contiene un total de cinco atributos:

1. **int m**: Representa el orden del nodo.
2. **ArrayList< Integer > key**: Contiene las claves del nodo.
3. **ArrayList< BNode > child**: Contiene una lista que hace referencia a los nodos hijos del nodo modelado.
4. **BNode parent**: Referencia al nodo padre del nodo modelado.
5. **boolean leaf**: Variable booleana que permite identificar si un nodo es una hoja en el árbol B.

A continuación, se tiene el método constructor de la clase, el cual inicializa las listas declaradas como atributos, establece el nodo como una hoja y la referencia a su padre como nula. Posteriormente, se tiene el método **getKey** que permite recuperar una de las claves almacenadas en el nodo por medio de la lista de enteros *key*, **getChild** que permite recuperar a uno de los hijos del nodo (Retornando null en caso de no encontrarse por medio de un bloque **try-catch**), los respectivos **sets** para el orden, la lista de claves e hijos y el método **getChildIndex** que permite recorrer la lista de hijos del nodo y obtener el índice de uno de ellos en la lista; retornando -1 en caso de no existir.

```
BNode padre = parent;
for (int i = 0; i < padre.child.size(); i++) {
    if (padre.child.get(i) == this) {
        return i;
    }
}
return -1;
```

Figura 38: Búsqueda del índice del hijo.

Finalmente, se tiene el método **mostrarLlaves** que imprime en pantalla la lista de claves que el nodo modelado almacena en su interior.

6.1.2. Clase BTree

Esta clase es la encargada de modelar los árboles B y únicamente tiene dos atributos: **int m** representando el orden del árbol y una referencia a un objeto *BNode* **root**; que representa al nodo raíz del árbol B a crear.

Esta clase contiene una extensa cantidad de métodos en su interior. Estos serán detallados y analizados a continuación:

- El método constructor de la clase inicializa el árbol B recibiendo el orden y creando un nodo **root** desde la inicialización del árbol.
- **add**: Este método recibe un entero y, si el método **find** devuelve **true**, imprimirá un mensaje indicando que la clave ya se encuentra en el árbol, caso contrario, recuperará una hoja en el árbol creando un objeto de tipo *BNode* e inicializándolo con el resultado del método **leafNode**; enviándole como parámetros el nodo **root** y el valor a insertar. Finalmente, la hoja recuperada se inserta en el nodo correspondiente usando el método **addToNode**.

```
if (find(n)) {
    System.out.println("La clave ya existe en el Árbol.");
} else {
    BNode hoja = leafNode(root, n);
    addToNode(hoja, n);
}
```

Figura 39: *Insertión de una clave en un nodo.*

- **leafNode**: Este método recibe un objeto *BNode* y un valor entero que representa una clave a insertar en el árbol. En primer lugar, si el nodo sobre el que se desea insertar tentativamente la clave es una hoja, el método devuelve directamente el nodo recibido. Caso contrario, por medio de un ciclo **for** se buscará el índice adecuado para realizar una inserción, es decir, se buscará el nodo hoja adecuado para insertar la clave y, finalmente, se devolverá una llamada recursiva del método peor ahora con el hijo *i* del nodo recibido con anterioridad.

```
for (; i < nodo.key.size(); i++) {
    if (n < nodo.getKey(i)) {
        i++;
        break;
    }
}

if (n < nodo.getKey(i - 1)) {
    i--;
}

return leafNode(nodo.getChild(i), n);
```

Figura 40: *Búsqueda de la hoja de inserción adecuada.*

- **addToNode**: Este método recibe un nodo y un valor *n* entero. Verificando que exista espacio en el

nodo actual (dado por $m - 1$), se imprime un mensaje del tamaño de la lista de claves del nodo y se inserta la clave por medio del método **insert**. Si esto no es posible, se realizará el proceso de **divisionCelular** siguiendo el procedimiento teórico establecido para los casos de inserción en un árbol B.

- **insert**: Recibiendo un nodo y una clave, se iterará con un ciclo **while** sobre la lista hasta llegar al índice tal que se cumpla que su valor sea menor al tamaño total de la lista de claves y el valor entero que se desea insertar sea superior a las otras claves almacenadas en la lista. Obtenido este índice, se inserta con el método **add** el valor en la hoja deseada.

```
while (i < nodo.key.size() && n > nodo.getKey(i)) {
    i++;
}
nodo.key.add(i, n);
```

Figura 41: Inserción en el caso en que existe espacio disponible en el nodo hoja adecuado.

- **divisionCelular**: El método encargado de realizar el proceso de división celular sigue una metodología un tanto *complicada*. En primer lugar, obtiene el entero representativo del valor mínimo que cada nodo puede tener de claves en el árbol. Si la operación $ordenArbolBmod2! = 0$, directamente se inserta el valor en el nodo recibido como parámetro (Por ejemplo, en un caso en el que se tenga un árbol de orden 4 que puede tener mínimo una clave en cada nodo).

De no cumplirse lo anterior, se obtendrá el valor medio obteniendo la clave intermedia del nodo usando el valor **h** renombrado a **halfMinIndex** que fue calculado previamente. Luego, se inicializarán un total de cuatro ArrayList, siendo las primeras dos listas las utilizadas para almacenar las claves de los nuevos nodos particionados y las últimas dos siendo sus respectivas referencias. Cabe señalar que la lista de claves para ambos nodos estará particionada de $[0, halfMinIndex]$ a $[halfMinIndex + 1, 2 * halfMinIndex + 1]$ y será repartida creando sublistas a partir de la original usando el método **subList**.

```
ArrayList<Integer> key1 = new ArrayList(nodo.key.subList(0, halfMinIndex));
ArrayList<Integer> key2 = new ArrayList(nodo.key.subList(halfMinIndex + 1, 2 * halfMinIndex + 1));
ArrayList<BNode> child1 = new ArrayList();
ArrayList<BNode> child2 = new ArrayList();
```

Figura 42: Proceso inicial de división.

Luego, si el nodo a particionar es el nodo **root** del árbol, se crearán dos nuevos nodos que serán establecidos como hojas y se les asignarán las sublistas de claves creadas previamente. Con respecto al nodo particionado, la clave intermedia subirá como un nuevo nodo, su lista de claves será limpiada y se le añadirá la clave *medio* obtenida previamente. Finalmente, se verifica si la operación $ordenArbolBmod2 == 0$. De ser cierto, si el valor a insertar es menor al valor medio, el valor será in-

gresado en el nodo izquierdo resultado de la partición, caso contrario, se insertará en el nodo derecho resultante.

```
if (ordenArbolB % 2 == 0) {
    if (n < medio) {
        insert(nuevoNodo1, n);
    } else {
        insert(nuevoNodo2, n);
    }
}
```

Figura 43: Asignación de la clave a insertar según su valor.

En el caso de que el nodo a particionar no sea una hoja, se realizará un procedimiento similar al seguido anteriormente pero verificando el valor de la operación *ordenArbolBmod2* para usar los índices para crear, establecer las referencias y particionar el nodo adecuadamente.

```
if (ordenArbolB % 2 != 0) {
    child1 = new ArrayList(nodo.child.subList(0, halfMinIndex + 1));
    child2 = new ArrayList(nodo.child.subList(halfMinIndex + 1, ordenArbolB + 1));
} else {
    if (n < medio) {
        child1 = new ArrayList(nodo.child.subList(0, halfMinIndex + 2));
        child2 = new ArrayList(nodo.child.subList(halfMinIndex + 2, ordenArbolB + 1));
    }
    if (n > medio) {
        child1 = new ArrayList(nodo.child.subList(0, halfMinIndex + 1));
        child2 = new ArrayList(nodo.child.subList(halfMinIndex + 1, ordenArbolB + 1));
    }
}
```

Figura 44: Índices de partición de las sublistas.

Hecho lo anterior, se asignarán a los nuevos nodos los arreglos de claves creados y, por medio de dos ciclos **for-each**, las referencias correspondientes de cada uno. Se limpia la lista de nodos hijo del nodo actual, se añaden esos nuevos hijos, se establece que los nodos creados son hijos del nodo **root** y, finalmente, el atributo **leaf** del nodo se establece como **false**.

Ahora, el segundo caso de todo este proceso largo se da cuando el nodo a particionar no es el nodo **root** del árbol B. En este caso, solo se crea un nuevo nodo y se establece a su padre como el mismo del nodo particionado y su condición de hoja como **true**. Luego, se obtiene el índice de los hijos contenidos en el nodo original, y se reparten las claves por medio de las sublistas creadas previamente asignando las sublistas con el método **setKeys**.

```
BNode nuevoNodo = new BNode();
nuevoNodo.leaf = nodo.leaf;
nuevoNodo.parent = nodo.parent;

int childIndex = nodo.getChildIndex();

nodo.setKeys(key1);
nuevoNodo.setKeys(key2);
```

Figura 45: Segundo caso representativo de la división celular.

Al igual que el caso anterior, se verifica la operación $\text{ordenArbolBmod2} == 0$ con el fin de saber en qué nodo se debe de insertar el valor (recordando que esta inserción debe ser exclusivamente ascendente para mantener la integridad del árbol B).

Luego, verificando que el nodo particionado no es una hoja, se realizará el proceso anterior de verificación de los índices para las sublistas de nodos hijos (es decir, las referencias) de forma similar al caso descrito previamente. Finalmente, se realizan las asignaciones de las sublistas de claves y las referencias correspondientes. Tras todo esto, finalmente, se inserta el nuevo nodo en la referencia correspondiente y se realiza el tratamiento del valor intermedio resultante.

```

nodo.setChildren(child1);
nuevoNodo.setChildren(child2);
for (BNode i : nodo.child) {
    i.parent = nodo;
}
for (BNode i : nuevoNodo.child) {
    i.parent = nuevoNodo;
}
nodo.parent.child.add(childIndex + 1, nuevoNodo);
addToNode(nodo.parent, medio);

```

Figura 46: Paso final de la división celular en su segundo caso.

- **mostrarArbol:** Este método se utiliza para imprimir el estado actual de un árbol B creado. En primer lugar, si resulta que el árbol no tiene hijos asignados y no contiene llaves en su nodo **root**, se imprimirá un mensaje indicando que no existen elementos.

Caso contrario, se procederá a realizar un recorrido por capas utilizando una cola ligada, añadiendo el nodo **root** a la lista y creando una referencia a un objeto **BNode parent** como nula. Luego, iterando por medio de un ciclo **while** y siempre y cuando la cola no esté vacía, se extraerá el nodo y si, el nodo extraído en su atributo *parent* es nulo, se imprimirá que el nodo actual representa al nodo **root** del árbol. Luego, si el nodo extraído contiene una referencia no nula a un nodo padre, se imprimirán las claves del nodo padre y los nodos hijos del nodo padre. Finalmente, se itera con un ciclo **for** sobre la lista de hijos para registrar los nodos hijos, añadirlos a la cola y repetir el proceso hasta que todos los nodos del árbol B sean visitados.

```

BNode v = nodos.poll();
if (v.parent == null) {
    System.out.print("Nodo Raiz: ");
}
if (parent != v.parent) {
    System.out.print("\n\nNodo Padre: ");
    v.parent.mostrarLlaves();
    parent = v.parent;
    System.out.print("\n\t\tNodos:");
}
System.out.print("\n\t\t");
v.mostrarLlaves();

```

Figura 47: Recorrido por capas para imprimir los nodos del árbol.

- **find**: Este método devuelve false si el nodo **root** de un árbol B no contiene referencias válidas hacia otros nodos (es decir, si no tiene hijos) y si se encuentra vacío. En caso contrario, devuelve el mismo método pero sobrecargado; enviando como parámetros al nodo **root** y el valor que fue introducido como parámetro originalmente en **find**.

Ahora, en el segundo método, si el nodo **root** enviado es nulo, el método devolverá **false**. De ser un árbol válido, se procederá a lo siguiente: si la clave en el índice 0 del nodo recibido es mayor al valor *v* buscado, se retornará el mismo método pero ahora enviando como nodo de búsqueda al hijo en el índice 0 del nodo actual.

```
if (n.getKey(0) > v) {
    return find(v, n.getChild(0));
}
```

Figura 48: Llamada recursiva del método *find*.

Si lo anterior no se cumple, se procederá a iterar con un ciclo **for** en la lista de claves del nodo actual, y si alguna de las claves contenidas en los índices del nodo coincide con el valor buscado, se retorna **true** o se enviará a uno de los hijos del nodo revisando el valor del nodo con respecto a las claves revisadas.

```
if (n.getKey(i) == v) {
    return true;
}

if (n.getKey(i) < v && n.getKey(i + 1) > v) {
    return find(v, n.getChild(i + 1));
}

}

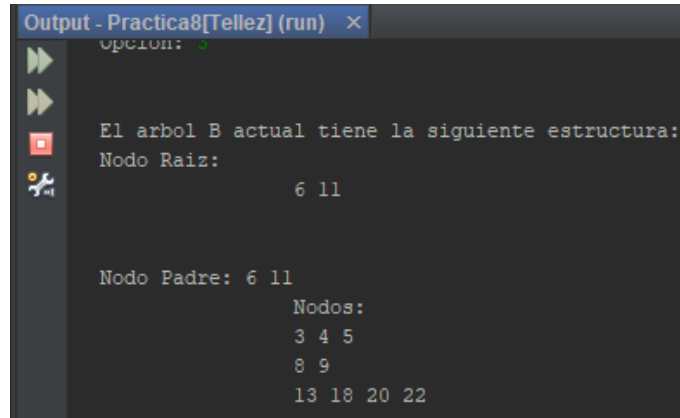
if (n.getKey(i) == v) {
    return true;
} else {
    if (n.getKey(i) < v) {
        return find(v, n.getChild(i + 1));
    } else {
        return find(v, n.getChild(i));
    }
}
```

Figura 49: Proceso recursivo de búsqueda en las referencias de los hijos del nodo según su valor.

6.2. Ejecución

El uso de las operaciones presentadas en la implementación han sido añadidas al programa permitiendo agregar un valor a un árbol B inicializado previamente, buscar un valor e imprimir el árbol. Con fines de prueba, se ha creado el siguiente árbol y se imprime en pantalla:

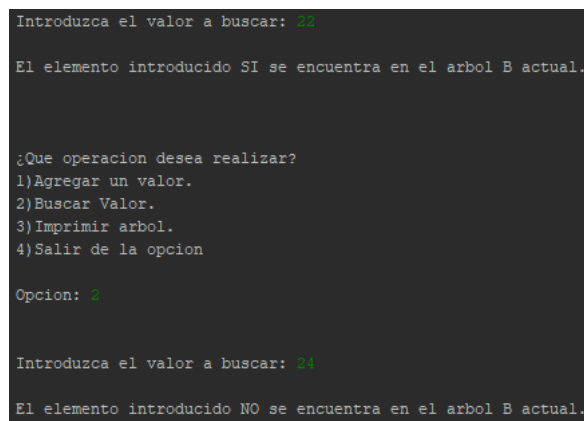
Si se realiza la búsqueda de la clave 22, el resultado será exitoso. Caso contrario, se imprimirá un mensaje de error. El mensaje de error aparecerá si se intentara buscar, por ejemplo, un 24 en el árbol.



```
Output - Practica8[Tellez] (run) x
opcion: 3

El arbol B actual tiene la siguiente estructura:
Nodo Raiz:      6 11

Nodo Padre: 6 11
Nodos:
  3 4 5
  8 9
 13 18 20 22
```

Figura 50: *Árbol B creado e impreso.*

```
Introduzca el valor a buscar: 22
El elemento introducido SI se encuentra en el arbol B actual.

¿Que operacion desea realizar?
1)Agregar un valor.
2)Buscar Valor.
3)Imprimir arbol.
4)Salir de la opcion

opcion: 2

Introduzca el valor a buscar: 34
El elemento introducido NO se encuentra en el arbol B actual.
```

Figura 51: *Árbol B creado e impreso.*

7. Conclusiones

Esta práctica ha representado la culminación de diversas técnicas de programación empleadas en el curso y ha resultado ser una de las experiencias más retadoras, si no es que la más retadora, de todas las prácticas del curso. Comprender el funcionamiento de las implementaciones, especialmente la de Árboles B, resultó ser una tarea árdua que llevó un buen tiempo. Quiero mencionar los siguientes puntos con respecto al desarrollo de la práctica:

- La creación de un método de eliminación para los árboles binarios no pudo ser completada con éxito debido a la aparición de múltiples errores al tratar de ligar las referencias creadas con el método de inserción propuesto. Esta parte, a pesar de ser la más *sencilla*, me causó problemas que no pude sortear del todo debido a dificultades en su implementación y problemas con los tiempos (tuve un corte repentino de luz en mi casa que me impidió trabajar por varios días). Por ello, el menú no incluye una eliminación para los árboles binarios.

- La creación de los BST, en cambio, resultó una tarea más complicada pero que pudo ser resuelta exitosamente. Cabe señalar que en gran parte de la bibliografía que consulté relacionada a la búsqueda del nodo sucesor para la eliminación se hacía en el sub árbol derecho, por lo que tuve que reimplementar desde 0 varios métodos para que funcionasen de acuerdo a la metodología seguida en las clases teóricas.
- El análisis de los Árboles ha sido uno de los más complicados que he enfrentado en todas las prácticas. Considero que este tiene un gran margen de mejora, sin embargo, me encuentro satisfecho con el trabajo realizado.
- La comprensión que tuve de las implementaciones, si bien considero que fue alto, me dejó con ciertas dudas existenciales especialmente en las partes concernientes a los índices en las operaciones de división celular en los Árboles B que resultan difíciles de seguir incluso teóricamente.

Esta práctica me ha dejado, a pesar de las dificultades que tuve, un buen sabor de boca y ha abierto mi interés de conocer más acerca del funcionamiento de estas estructuras. Considero que el trabajo desarrollado en esta práctica puede ser llevado más a fondo con el fin de que sea útil para próximas asignaturas; especialmente en lo concerniente a los *Sistemas Operativos*. También considero que existe un gran margen de mejora en la parte de los árboles binarios e, incluso, agregar un método de eliminación en los árboles B que se encuentra ausente en la implementación proporcionada.

La experiencia obtenida de todas las prácticas realizadas a lo largo del curso sin duda han marcado un antes y después en mi experiencia como Ingeniero en Computación. Con todas las herramientas adquiridas, me siento preparado para sortear futuros cursos y seguir mi formación académica con éxito; pese a las dificultades derivadas de la situación mundial debido a la pandemia del COVID-19.

Referencias

- [1] Binary Tree Implementation in Java - Insertion, Traversal And Search. Recuperado de: <https://www.netjstech.com/2019/03/binary-tree-implementation-in-java-insertion-traversal.html>. Fecha de consulta: 09/06/2020.
- [2] Deletion from BST (Binray Search Tree). Recuperado de: <https://www.techiedelight.com/deletion-from-bst/>. Fecha de consulta: 09/06/2020.
- [3] How to Print a Binary Tree Diagram. Recuperado de: <https://www.baeldung.com/java-print-binary-tree-diagram>. Fecha de consulta: 09/06/2020.
- [4] Implementing a Binary Tree in Java. Recuperado de: <https://www.baeldung.com/java-binary-tree>. Fecha de consulta: 09/06/2020.



[5] Program: How to delete a node from Binary Search Tree (BST)? Recuperado de: <https://www.java2novice.com/java-interview-programs/delete-node-binary-search-tree-bst/>. Fecha de consulta: 09/06/2020.

Los créditos de las fotografías pertenecen a sus respectivos autores. ©