	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: M.I. Edgar Tista García

Asignatura: Estructura de datos y algoritmos II -1317

Grupo: 10

No. de práctica(s): Avance de la práctica 8-9

Integrante(s): Mendoza Hernández Carlos Emiliano

No. de lista o brigada: 26

Semestre: 2023-1

Fecha de entrega: 10 de noviembre del 2022

Observaciones:

CALIFICACIÓN: _____

Práctica 8-9.

Árboles

OBJETIVOS

- **Objetivo general:** El estudiante conocerá e identificará las características de la estructura no-lineal árbol (árbol binario, binario de búsqueda y árbol B).
- **Objetivo de la clase:** El alumno analizará las implementaciones proporcionadas y conocerá la forma en la que se pueden implementar estas estructuras de datos.

DESARROLLO

Ejercicio 1. Implementación de un árbol binario

Análisis de la implementación de *Nodo.java*

La clase *Nodo* está compuesta por tres atributos, tres constructores y dos métodos.

***Atributos*:**

- *int valor*: Es el contenido del nodo. Un valor numérico entero.
- *Nodo izq*: Referencia al hijo izquierdo del nodo. Se inicializa por defecto con *null*.
- *Nodo der*: Referencia al hijo derecho de nodo. Se inicializa por defecto con *null*.

***Constructores*:**

- *Nodo()*: Se crea un nodo sin un valor especificado. Las referencias a los hijos de este nodo apuntan a *null*.
- *Nodo(data, lt, rt)*: Se crea un nodo con el valor especificado *data*. La referencia al hijo izquierdo de este nodo es el nodo especificado *lt*, y de manera análoga con el hijo derecho y *rt*.
- *Nodo(data)*: Se crea un nodo con el valor especificado de data. Las referencias a los hijos de este nodo apuntan a *null*.

***Métodos*:**

- *setIzq(izq)*: Establece la referencia especificada *izq* como hijo izquierdo de este nodo.
- *setDer(izq)*: Establece la referencia especificada *der* como hijo derecho de este nodo.

Análisis de la implementación de *ArbolBin.java*

La clase *ArbolBin* está compuesta por un atributo, tres constructores y tres métodos.

***Atributos*:**

- *Nodo root*: Contiene una referencia a un objeto de clase *Nodo* que representa la raíz del árbol.

***Constructores*:**

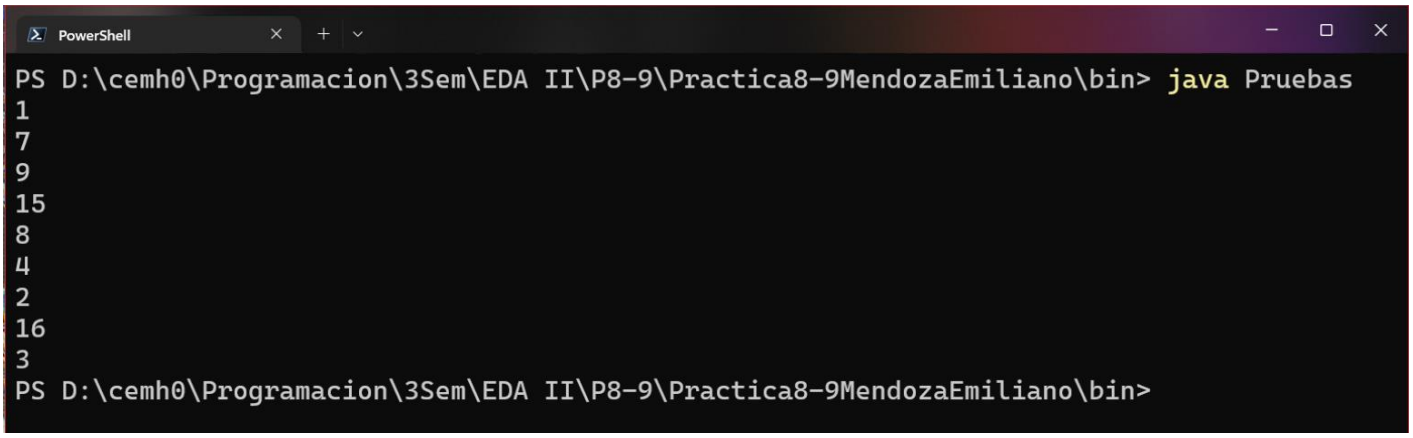
- *ArbolBin()*: Cuando el constructor no recibe parámetros, apunta la referencia de la raíz a *null*.
- *ArbolBin(val)*: Recibe un parámetro de tipo entero. Apunta la referencia de la raíz a una nueva instancia de un nodo con el valor especificado como parámetro.
- *ArbolBin(root)*: Recibe un nodo como parámetro. Apunta la referencia de la raíz al nodo especificado como parámetro.

***Métodos*:**

- *void add (padre, hijo, lado)*: Operación para agregar un nodo al árbol binario. Al nodo especificado como padre, se le asigna una referencia al nodo especificado como hijo. El parámetro *lado* funciona como una bandera para indicar de qué lado se agregará el nodo: si se pasa 0 como parámetro se coloca al nodo como hijo izquierdo del nodo especificado; de lo contrario se agrega como hijo derecho.
- *void visit (n)*: Este método funciona como auxiliar del siguiente (*breadthFirst*). Imprime el valor del nodo *n* especificado.

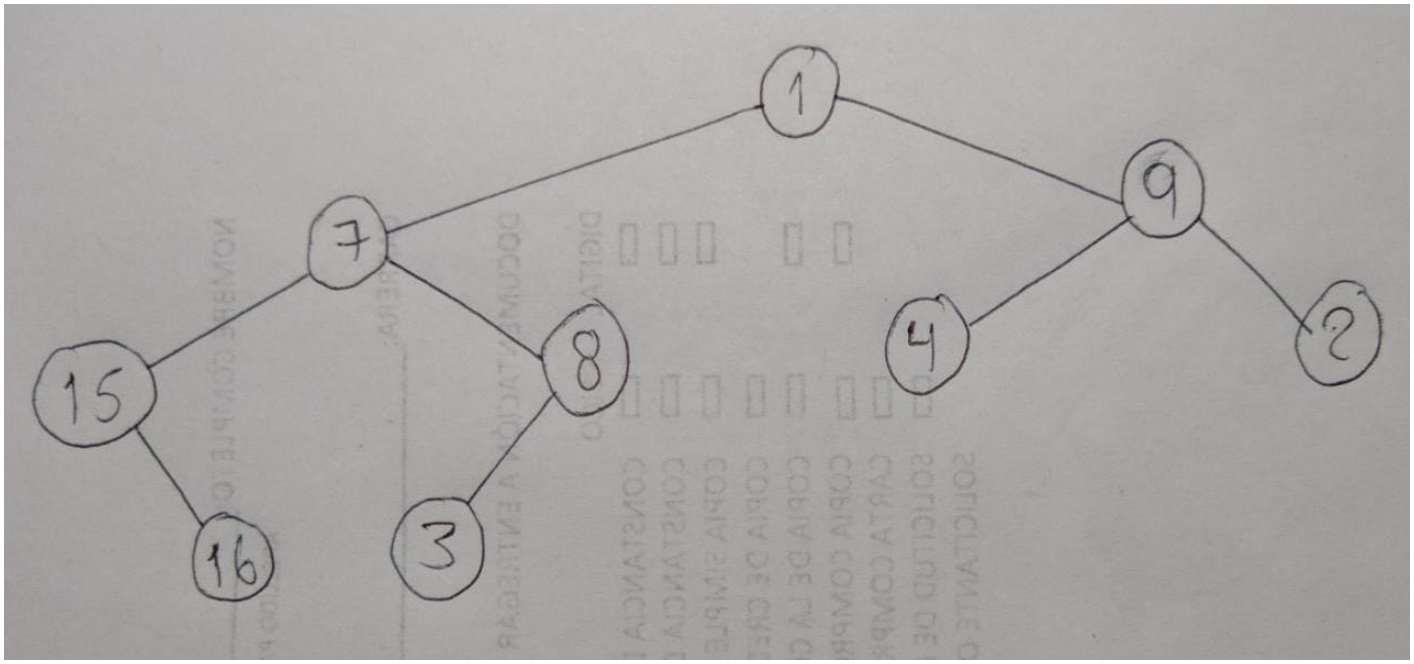
- *void breadthFirst()*: El método realiza lo siguiente:
  - Primero, se crea una nueva referencia *r* a la raíz del árbol y una cola de nodos *queue*.
  - Luego, se valida que el árbol no esté vacío.
  - A continuación, se encola la raíz. El recorrido BFS inicia siempre por la raíz.
  - Mientras la cola tenga nodos:
    - Se desencola y se actualiza *r* con el nodo desencolado.
    - Se marca el nodo *r* como visitado.
    - Si tiene un hijo izquierdo, se encola el hijo izquierdo de *r*.
    - Si tiene un hijo derecho, se encola el hijo derecho de *r*.

Ejecución de la clase *Pruebas.java* con el primer árbol de prueba

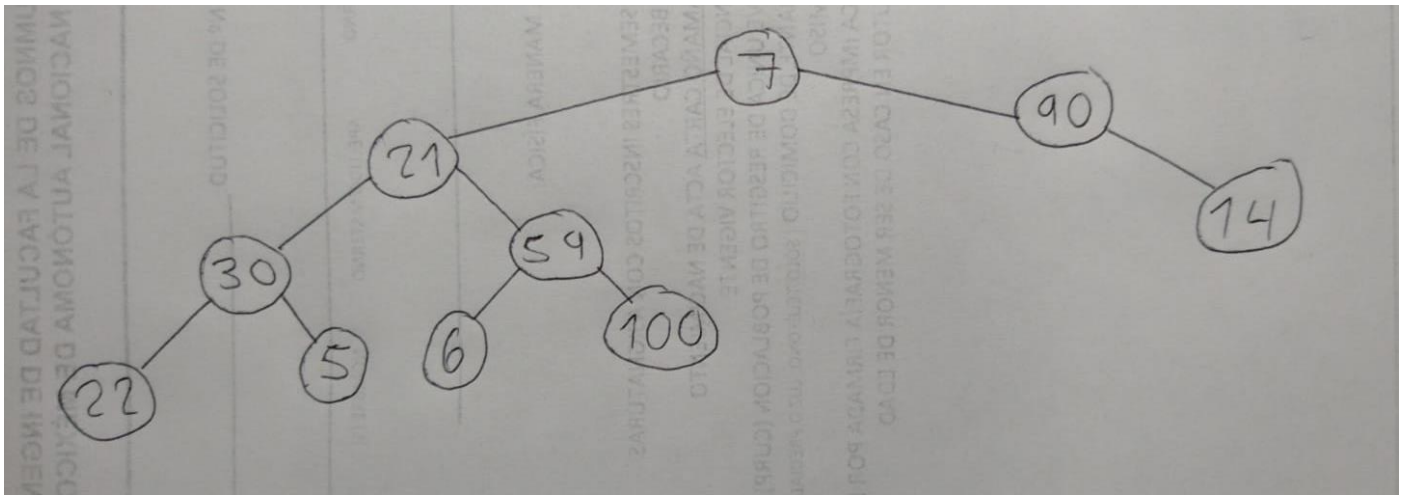


La salida del programa imprime el recorrido BFS para el primer árbol de prueba.

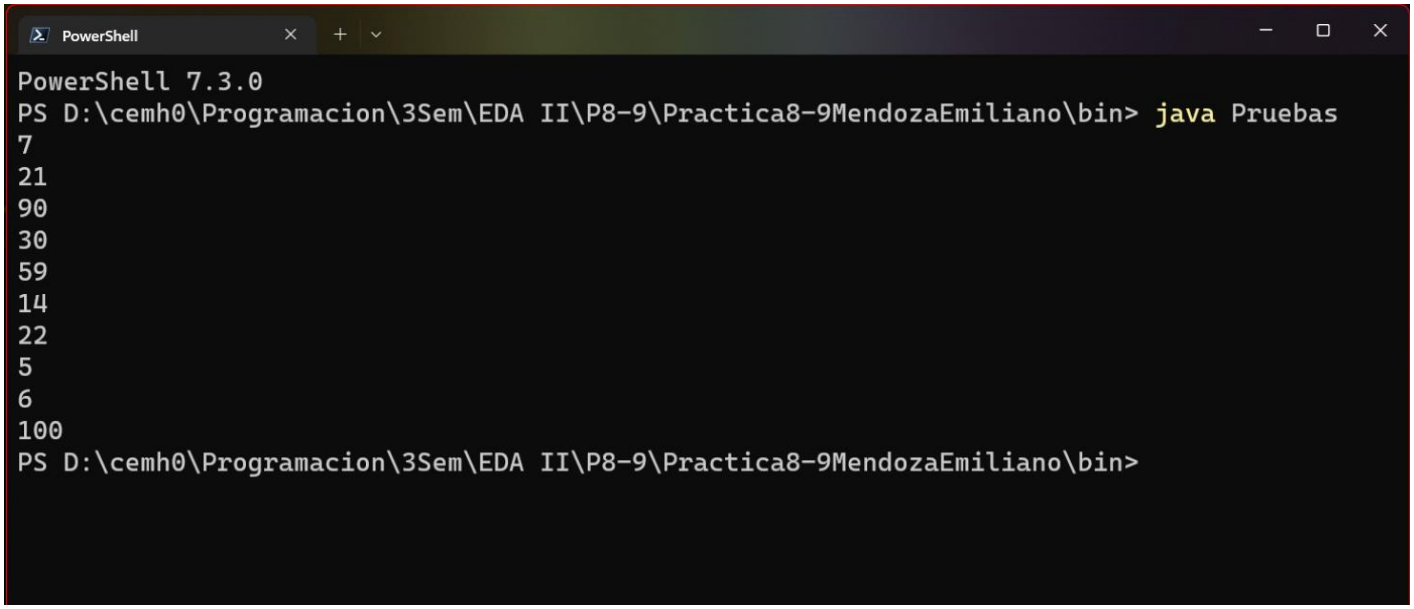
Dibujo del árbol generado en la clase *Pruebas*



Dibujo de un segundo árbol para la clase *Pruebas*



Ejecución de la clase *Pruebas.java* con el segundo árbol de prueba



La salida es correcta, imprimiendo el recorrido BFS del segundo árbol de ejemplo.

Análisis de ventajas y desventajas de la implementación de árbol binario

Al leer el código de prueba del primer árbol, noté que la manera de agregar nodos es muy similar a la manera en que se agregaban aristas en un grafo. La principal ventaja de esto es que se puede elegir con bastante libertad en qué parte del árbol se coloca un nuevo nodo. Otra ventaja es que es posible elegir en qué lado poner al hijo de un nodo (izquierdo o derecho), además de que es posible indicar la relación *padre-hijo* desde dos momentos diferentes: al instanciar el nodo o al añadir el nodo al árbol.

Sin embargo, una desventaja que noté es que se deben instanciar los nodos previamente a añadirlos al árbol, lo cual puede añadir demasiadas instrucciones al programa. Otra desventaja es que el recorrido BFS como método para imprimir el árbol no permite saber con exactitud la forma del árbol.

Implementación del método para realizar la eliminación de un nodo

- ***public void delete(int value)***

Este método funciona con un parámetro que especifica el valor que se desea eliminar. Se supondrá que el usuario ingresará un valor que existe en el árbol. A continuación, se describen los pasos que sigue el algoritmo:

***Caso 1*: El nodo que se quiere eliminar es la raíz**

***Subcaso 1.1:* La raíz no tiene ningún hijo.**

Para este caso específico, se alerta al usuario que no es posible eliminar la raíz porque se perdería por completo el árbol.

***Subcasos 1.2 y 1.3:* La raíz tiene solo un hijo izquierdo o solo un hijo derecho.**

La raíz pasa a ser el hijo izquierdo o el hijo derecho según sea el caso.

***Subcaso 1.4:* La raíz tiene sus dos hijos.**

La raíz pasa a ser el hijo izquierdo. Y para no perder el subárbol derecho, se guarda una referencia temporal de todo el subárbol derecho y se reinserta en el nodo más a la derecha del subárbol izquierdo.

***Caso 2:* El nodo que se quiere eliminar no es la raíz.**

***Subcaso 2.1:* El nodo es una hoja.**

El nodo se elimina directamente.

Subcaso 2.2 y 2.3: El nodo tiene solo un hijo izquierdo o solo un hijo derecho.

Se recorre el subárbol izquierdo o el subárbol derecho, según sea el caso, a la posición del nodo eliminado.

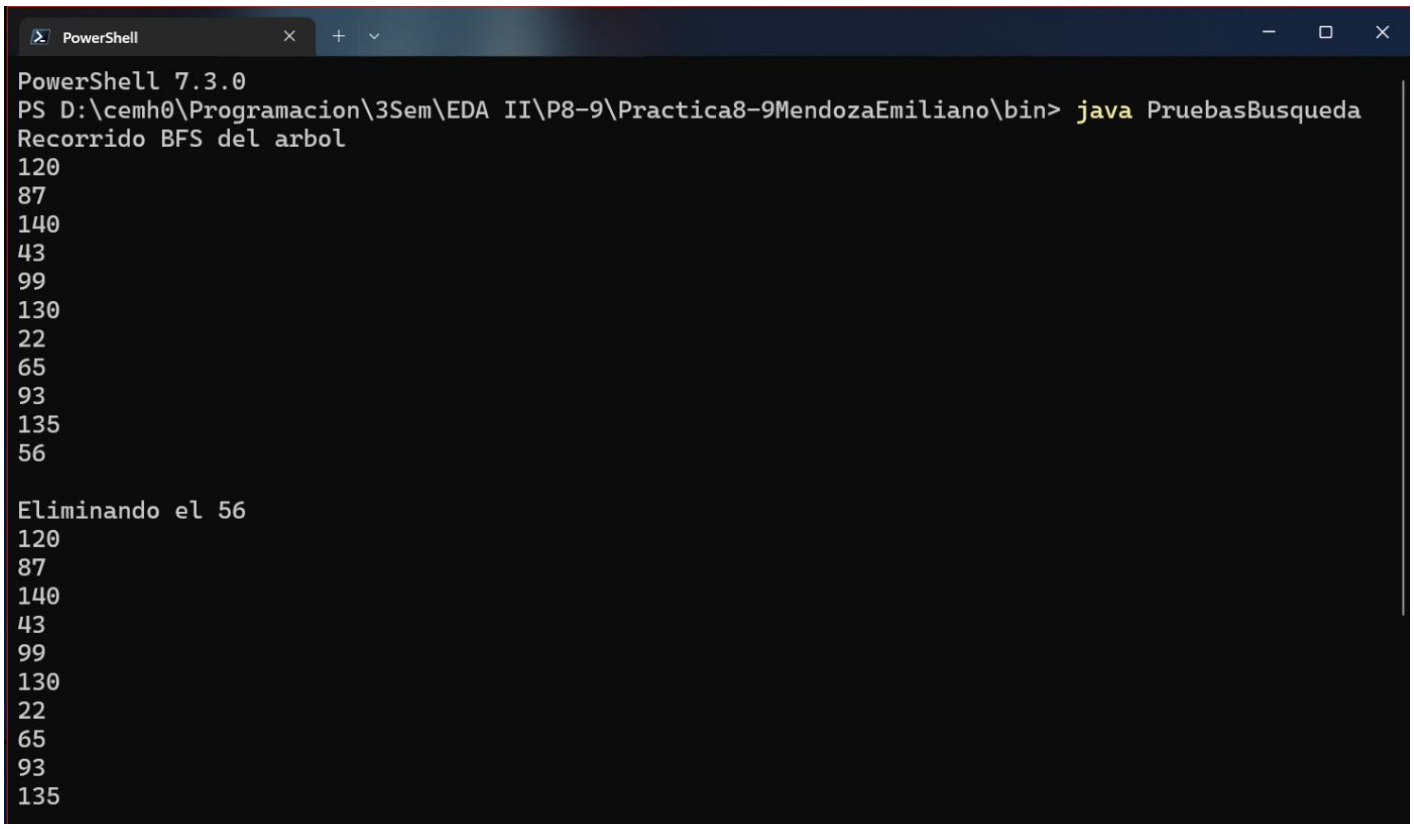
Subcaso 2.4: El nodo tiene dos hijos

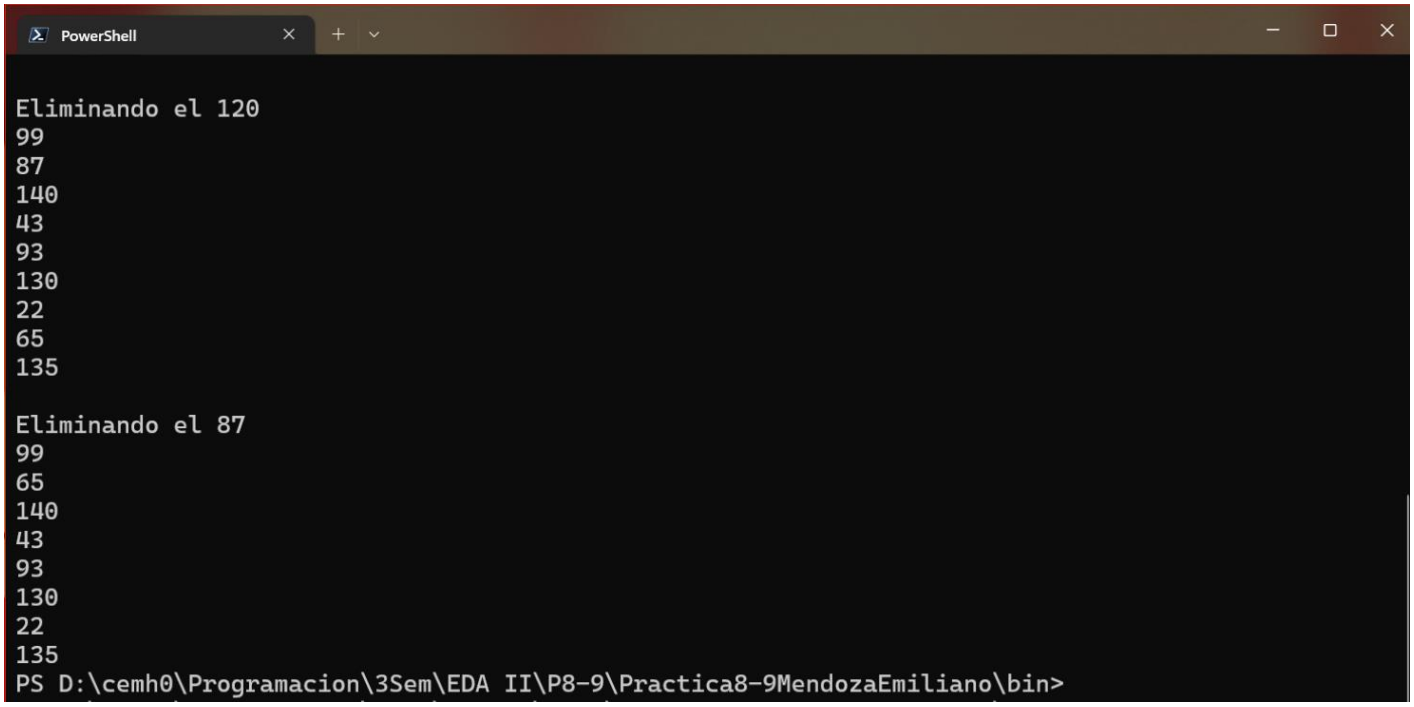
El nodo pasa a ser el hijo izquierdo. Y para no perder el subárbol derecho, se guarda una referencia temporal de todo el subárbol derecho y se reinserta en el nodo más a la derecha del subárbol izquierdo.

Nota: Se crearon dos métodos adicionales para el funcionamiento del algoritmo.

- *Nodo getParent (value)*: Operación para obtener el padre de un valor entero dado. Dentro de la función de eliminación funciona como referencia para borrar alguno de sus hijos.
- *Nodo search(x)*: Este método funciona para obtener una referencia de tipo *Nodo* de un valor entero *x* dado.

Ejecución de pruebas con el método *delete()*





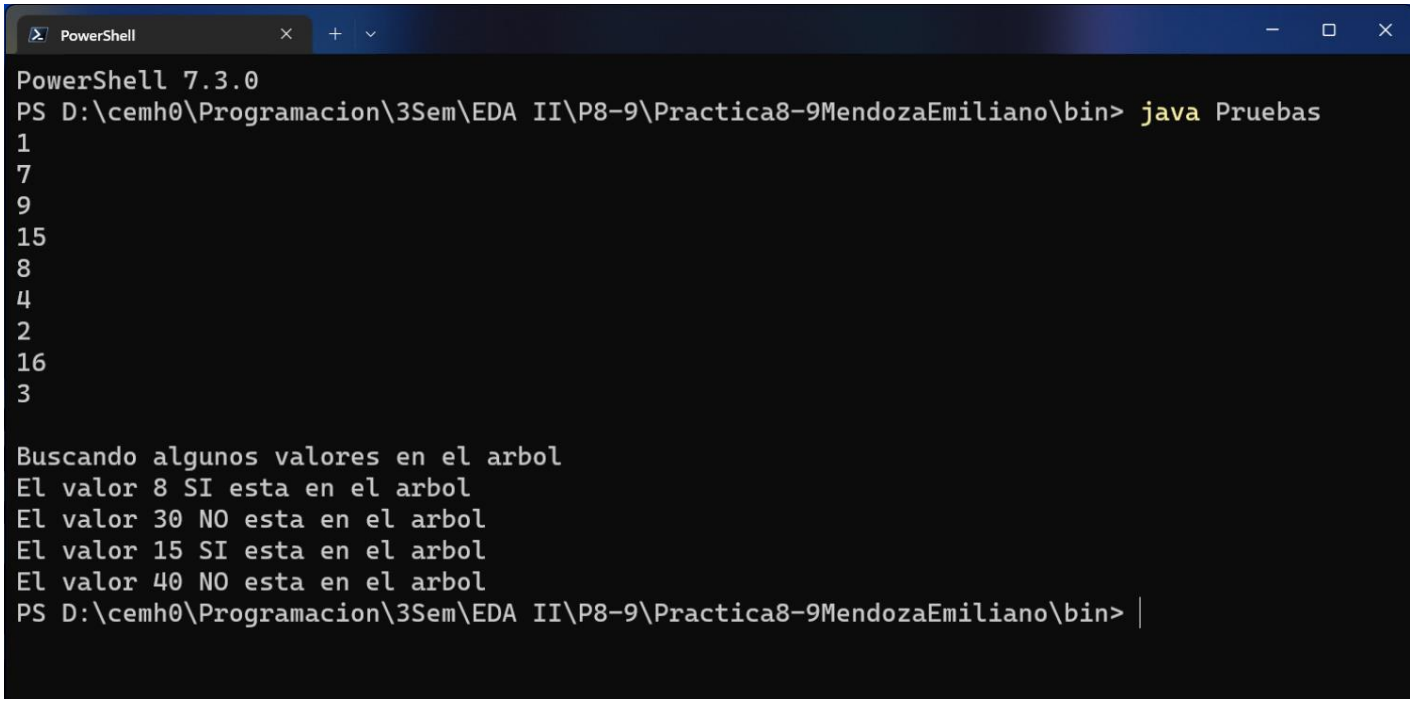
La eliminación de los nodos se realizó de manera correcta, incluyendo algunos de los casos de eliminación. Se eliminó la raíz (120), un nodo hoja (56) y un nodo con dos hijos (87).

Implementación del método para realizar la búsqueda de un valor en los nodos

- ***public void contains(int x)***

Para buscar un valor en los nodos, se recorre usando el algoritmo de recorrido BFS. La comparación con el valor especificado *x* se realiza cuando se desencola un nodo de *queue*. Si el valor coincide, imprime “SI”; caso contrario, cuando se recorre todo el árbol y no se encuentra ninguna coincidencia con el valor *x*, imprime “NO”.

Ejecución de pruebas con el método *contains()*



La salida del programa es correcta, podemos observar que efectivamente el árbol contiene los nodos de prueba con el valor 8 y 15, y que no contiene a los nodos 30 ni 40.

Ejercicio 2. Operaciones de un árbol binario

Implementación del método para obtener la notación prefija de un árbol binario

- ***public void preorder()***

Este método utiliza un método auxiliar para utilizar la recursividad. Al invocarlo imprime la notación prefija del árbol

- ***private void preorderUtil(Nodo n)***

Este método primero marca como visitado al nodo recibido como parámetro. Si este nodo tiene un hijo izquierdo, hace la llamada recursiva con él. Después de esa llamada recursiva, si el nodo tiene un hijo derecho, se hace la llamada recursiva con él. En resumen:

  - Examina la raíz
  - Recorre el subárbol izquierdo en preorden
  - Recorre el subárbol derecho en preorden

Implementación del método para obtener la notación infija de un árbol binario

- ***public void inorder()***

Este método utiliza un método auxiliar para utilizar la recursividad. Al invocarlo imprime la notación infija del árbol

- ***private void inorderUtil(Nodo n)***

Este método primero revisa si el nodo tiene un hijo izquierdo; si lo tiene, hace la llamada recursiva con él. Después, marca al nodo como revisado. Por último, si el nodo tiene un hijo derecho, hace la llamada recursiva con él. En resumen:

  - Recorre el subárbol izquierdo en inorden
  - Examina la raíz
  - Recorre el subárbol derecho en inorden

Implementación del método para obtener la notación posfija de un árbol binario

- ***public void postorder()***

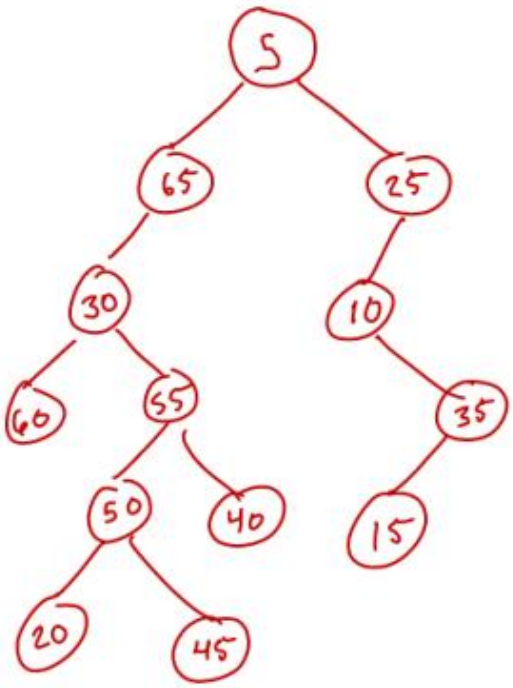
Este método utiliza un método auxiliar para utilizar la recursividad. Al invocarlo imprime la notación posfija del árbol.

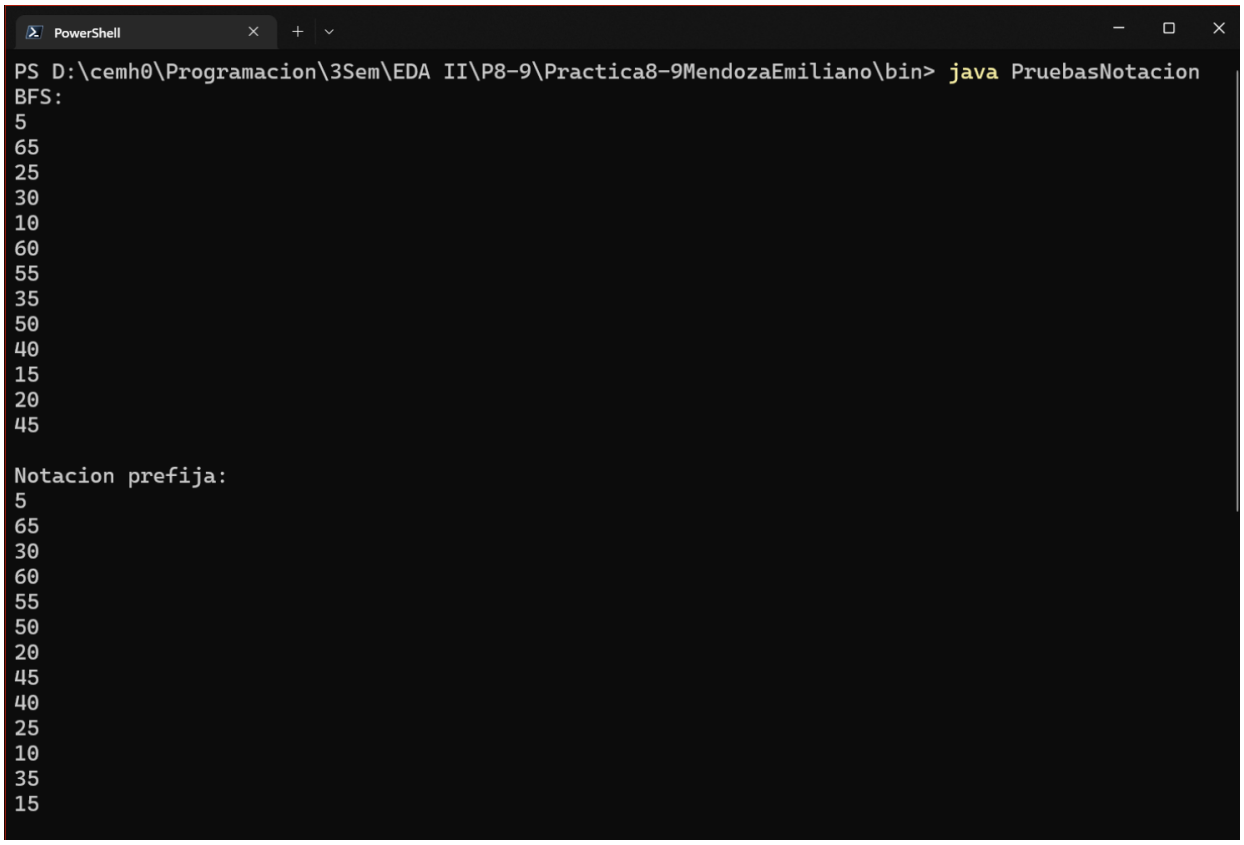
- ***private void postorderUtil(Nodo n)***

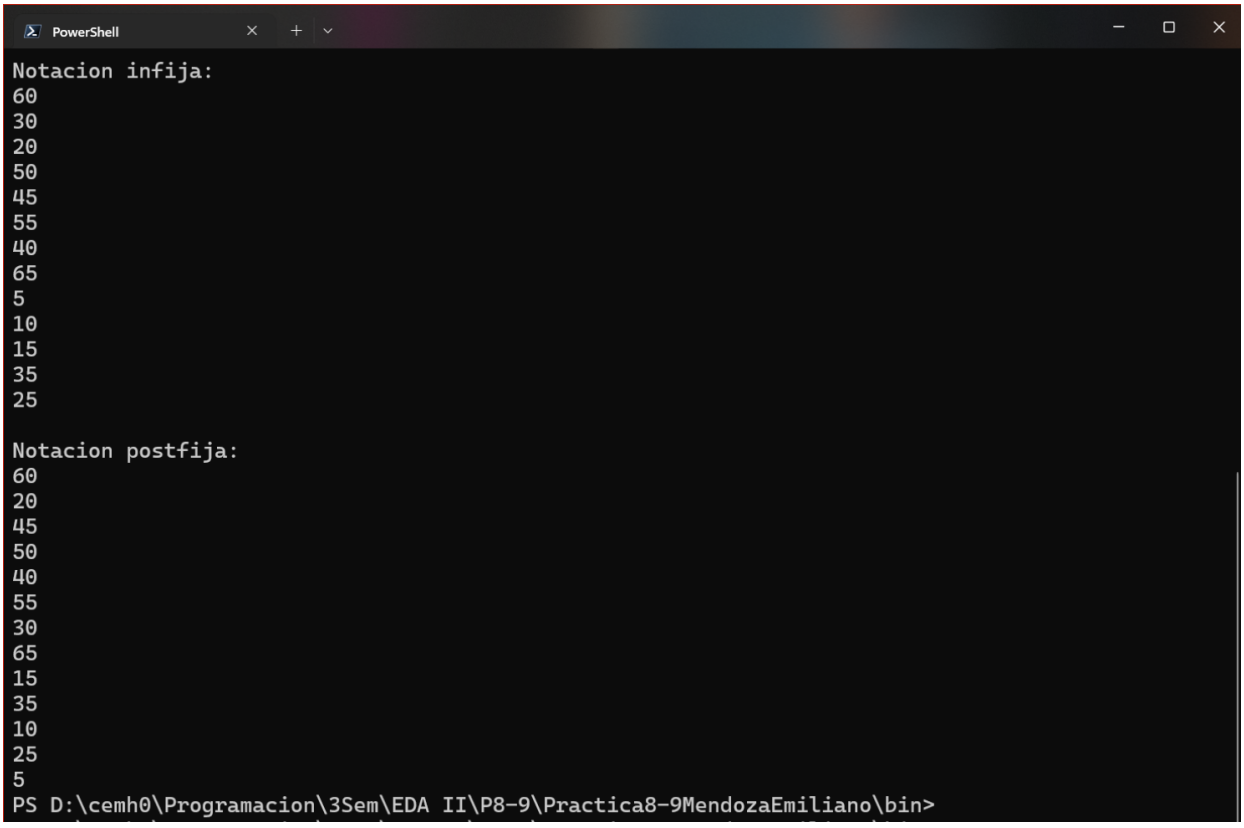
Este método primero revisa si el nodo tiene un hijo izquierdo; si lo tiene, hace la llamada recursiva con él. Después, si el nodo tiene un hijo derecho, hace la llamada recursiva con él. Finalmente, marca al nodo como revisado. En resumen:

- Recorre el subárbol izquierdo en posorden
- Recorre el subárbol derecho en inorden
- Examina la raíz

Prueba de los métodos para obtener las notaciones del árbol




PowerShell
PS D:\cemh0\Programacion\3Sem\EDA II\P8-9\Practica8-9MendozaEmiliano\bin> java PruebasNotacion
BFS:
5
65
25
30
10
60
55
35
50
40
15
20
45
Notacion prefija:
5
65
30
60
55
50
20
45
40
25
10
35
15


PowerShell
Notacion infija:
60
30
20
50
45
55
40
65
5
10
15
35
25
Notacion postfija:
60
20
45
50
40
55
30
65
15
35
10
25
5
PS D:\cemh0\Programacion\3Sem\EDA II\P8-9\Practica8-9MendozaEmiliano\bin>

Ejercicio 3. Árbol binario de búsqueda

Implementación de la clase *ArbolBinBusq.java*

La clase *ArbolBinBusq* establece una relación de herencia con la clase *ArbolBin*. Esto implica que *ArbolBinBusq* hereda los métodos y el atributo de *ArbolBin.*

.

***Atributos heredados de ArbolBin*:**

- *Nodo root*.

***Constructores*:**

- *ArbolBinBusq()*: Cuando el constructor no recibe parámetros, apunta la referencia de la raíz a *null*.
- *ArbolBin(val)*: Recibe un parámetro de tipo entero. Apunta la referencia de la raíz a una nueva instancia de un nodo con el valor especificado como parámetro.
- *ArbolBin(root)*: Recibe un nodo como parámetro. Apunta la referencia de la raíz al nodo especificado como parámetro.

Cabe mencionar que todos los constructores hacen un llamado al constructor de la clase padre. El funcionamiento y casos de uso de estos constructores son similares a los de la clase *ArbolBinBusq*.

***Métodos heredados de ArbolBin*:**

- *visit(n)*
- *breadthFirst()*
- *contains(x)*
- *search(x)*
- *getParent(x)*
- *preorder()*
- *inorder()*
- *postorder()*

***Método sobrescrito de la clase padre*:**

- *delete(val)*:

Este método sobrescribe al método de la clase padre. Funciona de manera similar: con un parámetro de tipo entero se especifica el valor que se desea eliminar. Se supondrá que el usuario ingresará un valor que existe en el árbol. A continuación, se describen los pasos que sigue el algoritmo:

***Caso 1*: El nodo que se quiere eliminar es la raíz**

***Subcaso 1.1:* La raíz no tiene ningún hijo.**

Para este caso específico, se alerta al usuario que no es posible eliminar la raíz porque se perdería por completo el árbol.

***Subcasos 1.2 y 1.3:* La raíz tiene solo un hijo izquierdo o solo un hijo derecho.**

La raíz pasa a ser el hijo izquierdo o el hijo derecho según sea el caso.

***Subcaso 1.4:* La raíz tiene sus dos hijos.**

Se busca al predecesor (nodo más a la derecha del subárbol izquierdo) del nodo. A continuación, se dividen otras dos posibilidades, para ambos subcasos se parte del hecho de que el subárbol derecho se guarda como respaldo en una referencia temporal:

Subcaso 1.4.1: El predecesor es una hoja

La raíz pasa a ser el predecesor y el subárbol derecho se reinserta como hijo derecho del predecesor.

Subcaso 1.4.2 El predecesor tiene un hijo

Se recorre el subárbol del predecesor (se sabe que son todos hijos izquierdos) a la raíz, se elimina el predecesor, y por último, se reinserta el subárbol derecho a la derecha del nodo que reemplaza al nodo eliminado.

***Caso 2:* El nodo que se quiere eliminar no es la raíz.**

***Subcaso 2.1:* El nodo es una hoja.**

El nodo se elimina directamente.

Subcaso 2.2 y 2.3: El nodo tiene solo un hijo izquierdo o solo un hijo derecho.

La referencia que apunta del nodo padre al nodo a eliminar pasa a apunar a su hijo. Se elimina el nodo.

Subcaso 2.4: El nodo tiene dos hijos

Primero se busca al predecesor del nodo, se intercambia el predecesor con el nodo que se desea eliminar. Para ello se deben validar dos casos más:

Subcaso 2.4.1 El predecesor es una hoja

Después de intercambiar el nodo con el predecesor, se elimina el nodo.

Subcaso 2.4.2 El predecesor tiene un hijo

El elemento se intercambia con el predecesor, es borrado y su hijo ahora será hijo del predecesor.

***Método específico de la clase*:**

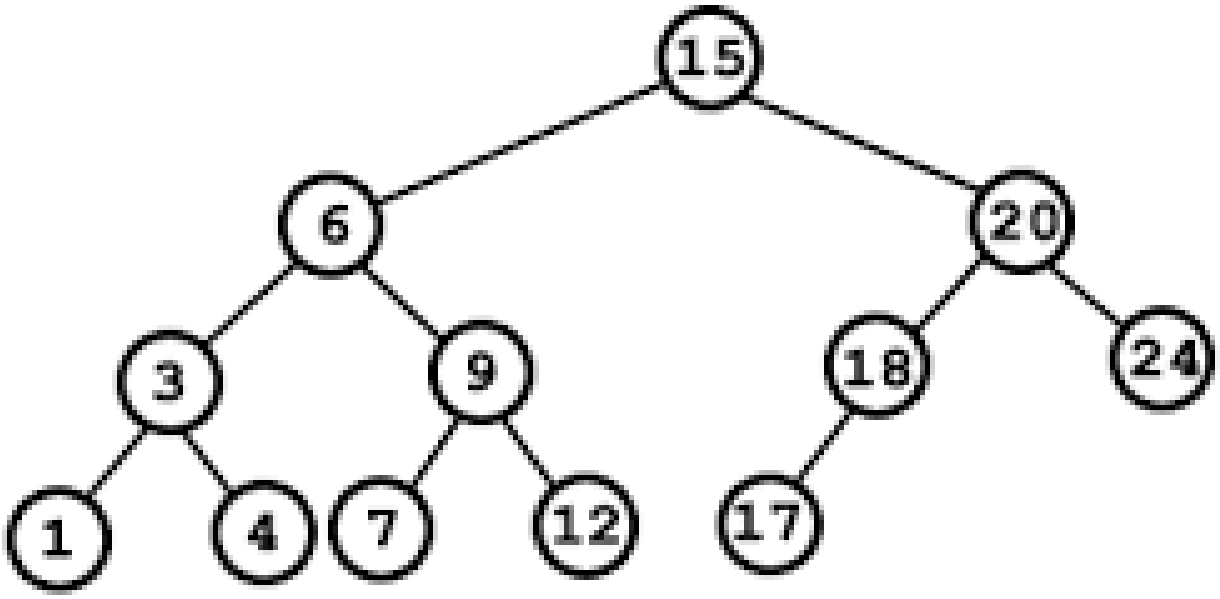
- *addBin(val)*:

Recibe un valor de tipo entero especificando el valor que se va a agregar al árbol.

Internamente funciona con un ciclo que recorre los nodos desde la raíz hasta llegar a la hoja que le corresponde y lo coloca de acuerdo con su valor (a la izquierda si es menor que su padre, a la derecha si es mayor).

Nota: Este método no sobreescribe el método *add(padre, hijo, lado)*. La principal razón de esto es porque en un árbol binario de búsqueda no podemos elegir, en teoría, quien será el padre del nodo que se está ingresando ni de qué lado ingresará; por lo tanto, los parámetros *padre*, *hijo* y *lado* no son necesarios. En su lugar, este método *addBin* solo recibe un parámetro con el valor del nodo especificado.

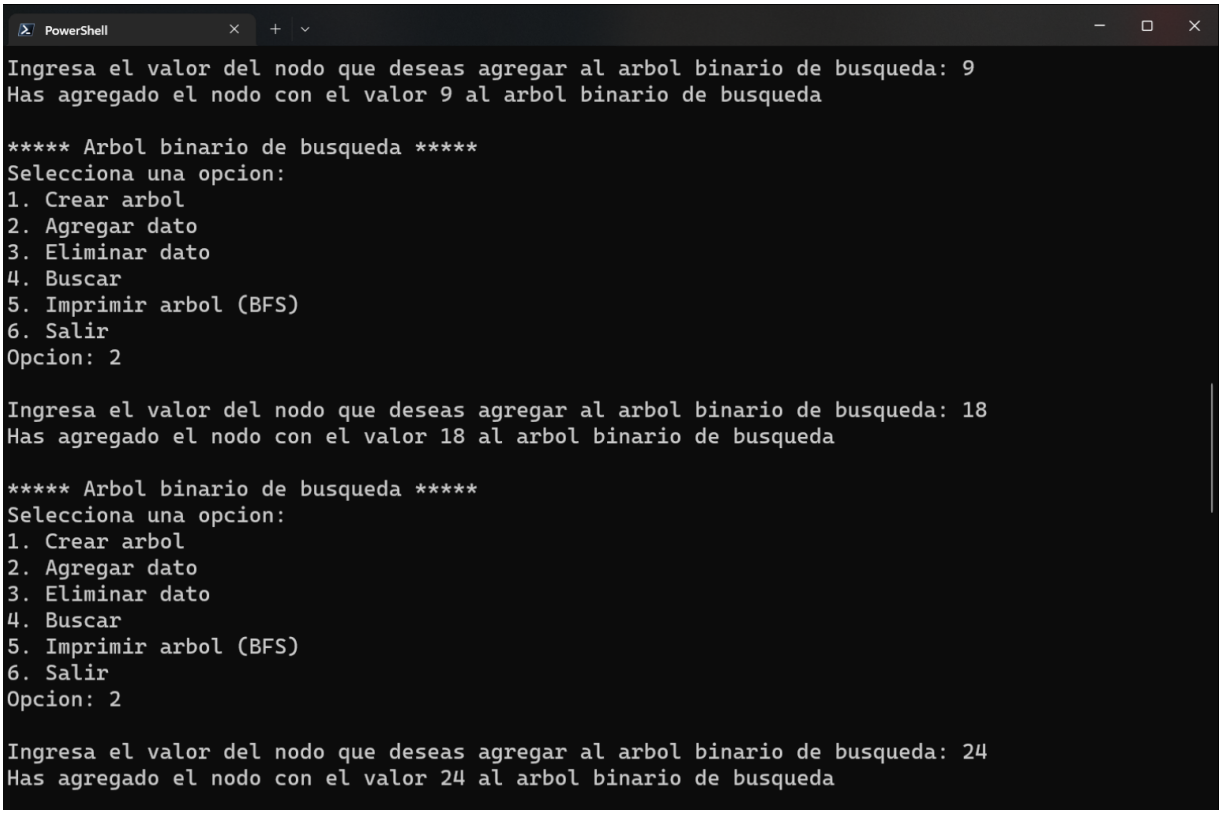
Prueba de la clase con el árbol

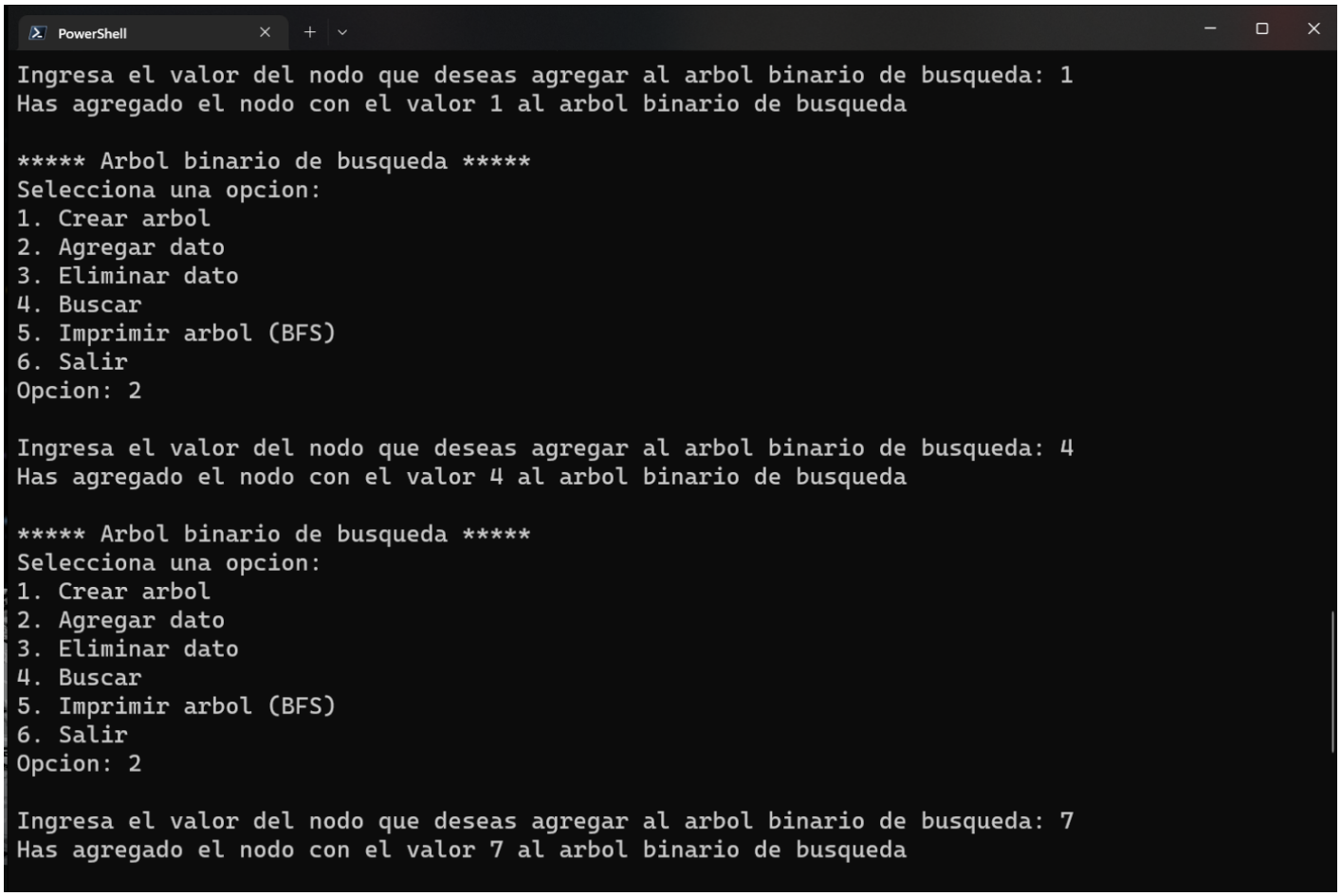


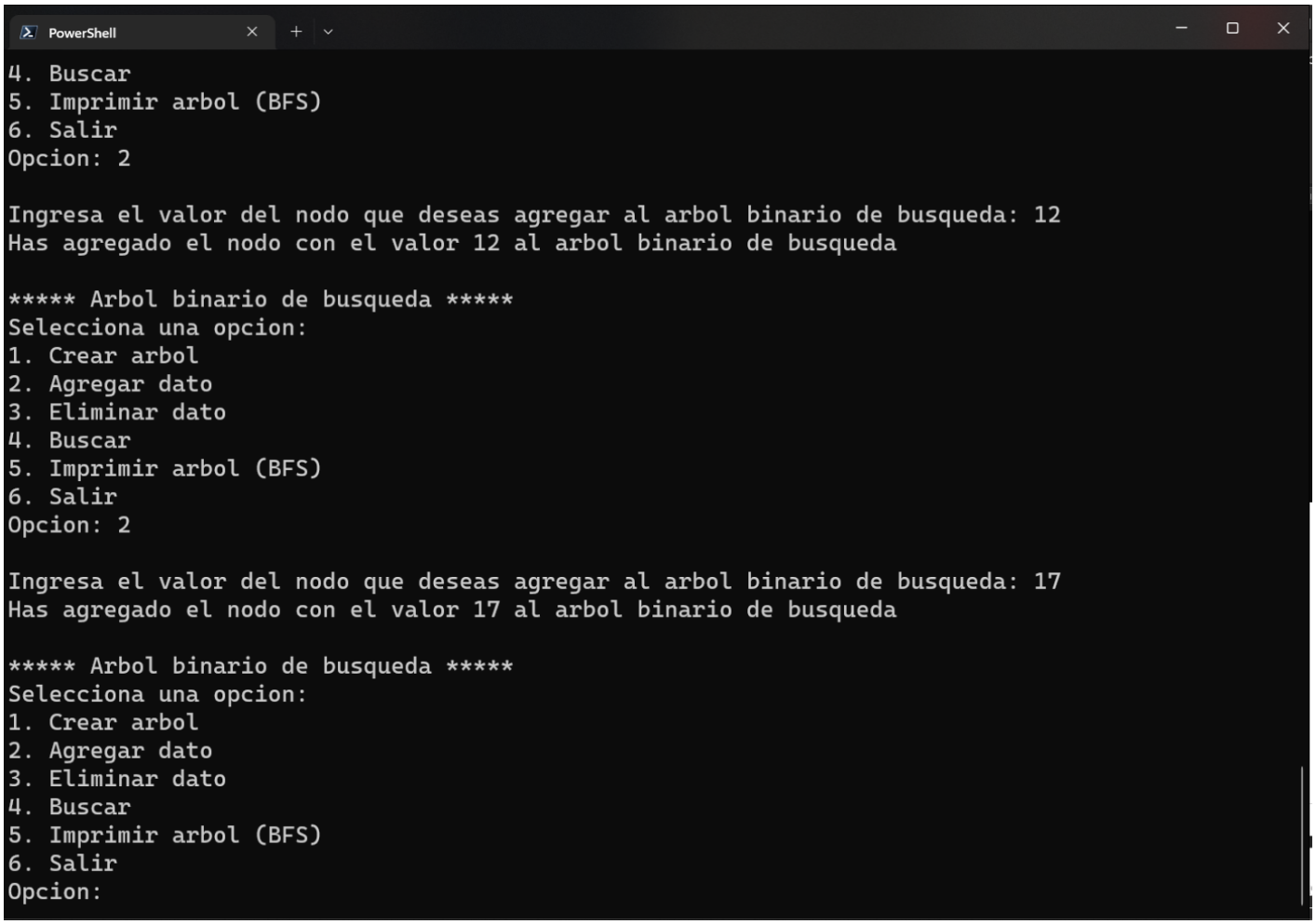
Creación del árbol en el programa

```
PowerShell
PS D:\cemh0\Programacion\3Sem\EDA II\P8-9\Practica8-9MendozaEmiliano\bin> java Main
***** Menu Practica 8-9 *****
Selecciona una opcion:
1. Arbol binario
2. Arbol binario de busqueda
3. Arbol B
Opcion: 2

***** Arbol binario de busqueda *****
Selecciona una opcion:
1. Crear arbol
2. Agregar dato
3. Eliminar dato
4. Buscar
5. Imprimir arbol (BFS)
6. Salir
Opcion: 1

Ingresa el valor de la raiz: 15
Has creado un nuevo arbol binario de busqueda con la raiz 15
```

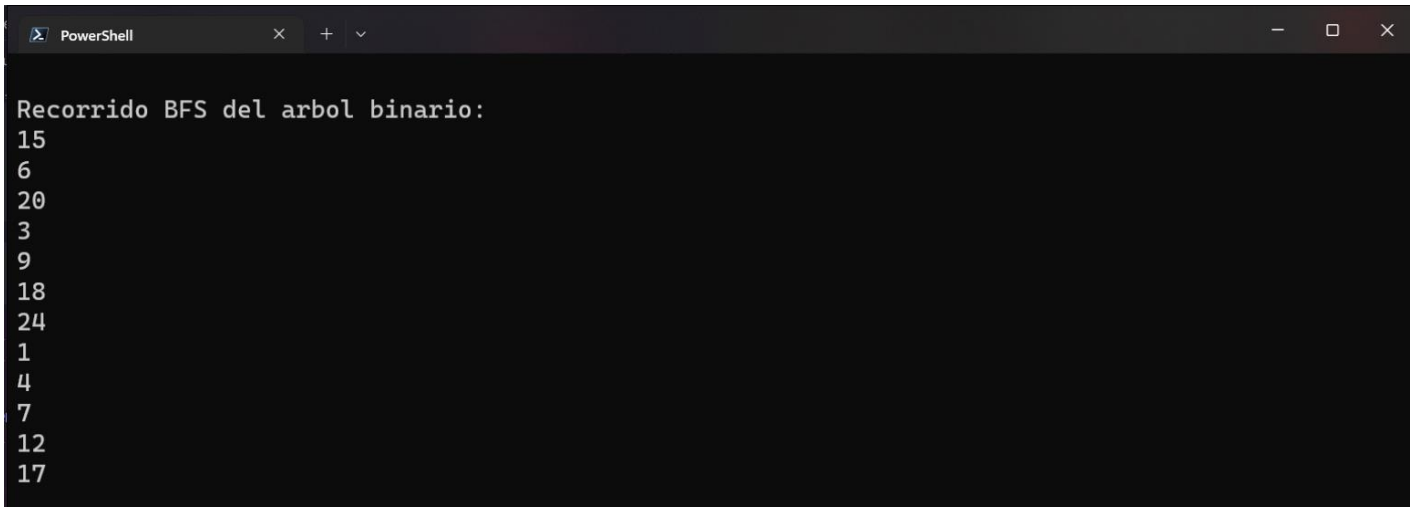
Agregando los nodos al árbol en el programa



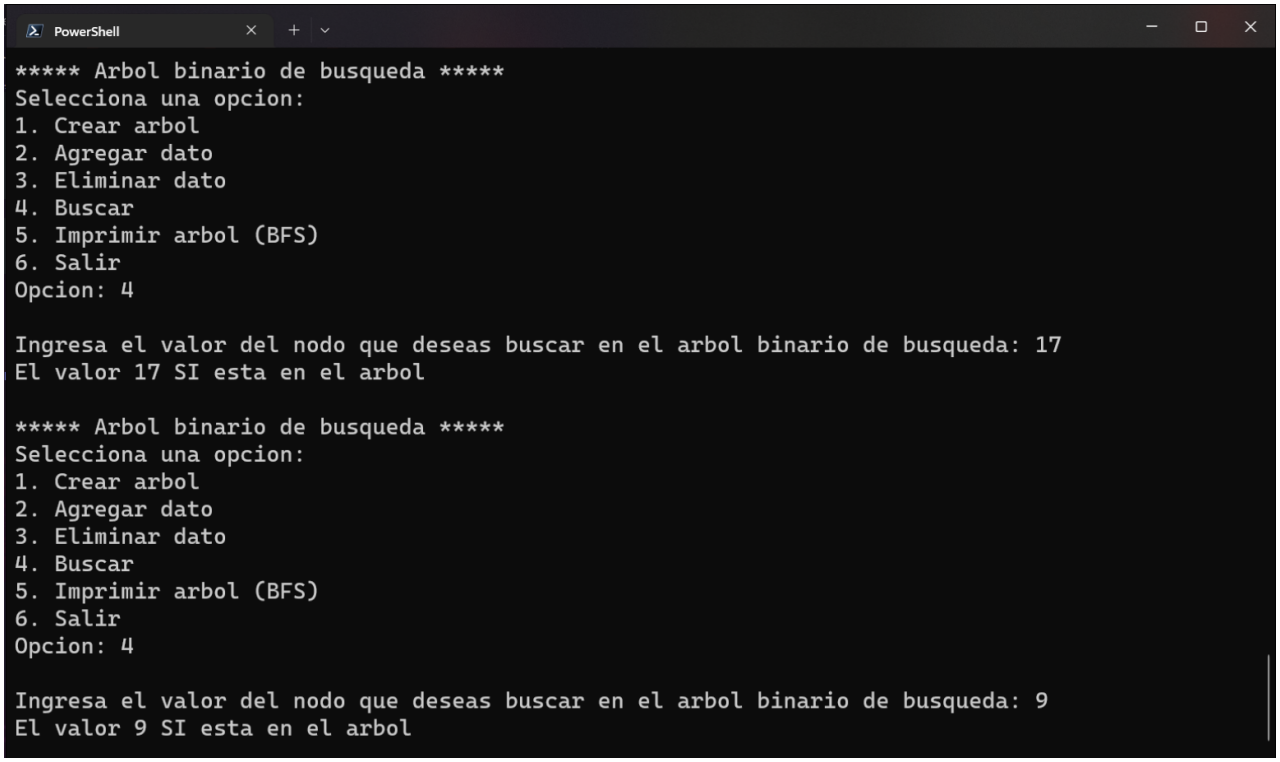

PowerShell
Ingresa el valor del nodo que deseas agregar al arbol binario de busqueda: 1
Has agregado el nodo con el valor 1 al arbol binario de busqueda
***** Arbol binario de busqueda *****
Selecciona una opcion:
1. Crear arbol
2. Agregar dato
3. Eliminar dato
4. Buscar
5. Imprimir arbol (BFS)
6. Salir
Opcion: 2
Ingresa el valor del nodo que deseas agregar al arbol binario de busqueda: 4
Has agregado el nodo con el valor 4 al arbol binario de busqueda
***** Arbol binario de busqueda *****
Selecciona una opcion:
1. Crear arbol
2. Agregar dato
3. Eliminar dato
4. Buscar
5. Imprimir arbol (BFS)
6. Salir
Opcion: 2
Ingresa el valor del nodo que deseas agregar al arbol binario de busqueda: 7
Has agregado el nodo con el valor 7 al arbol binario de busqueda

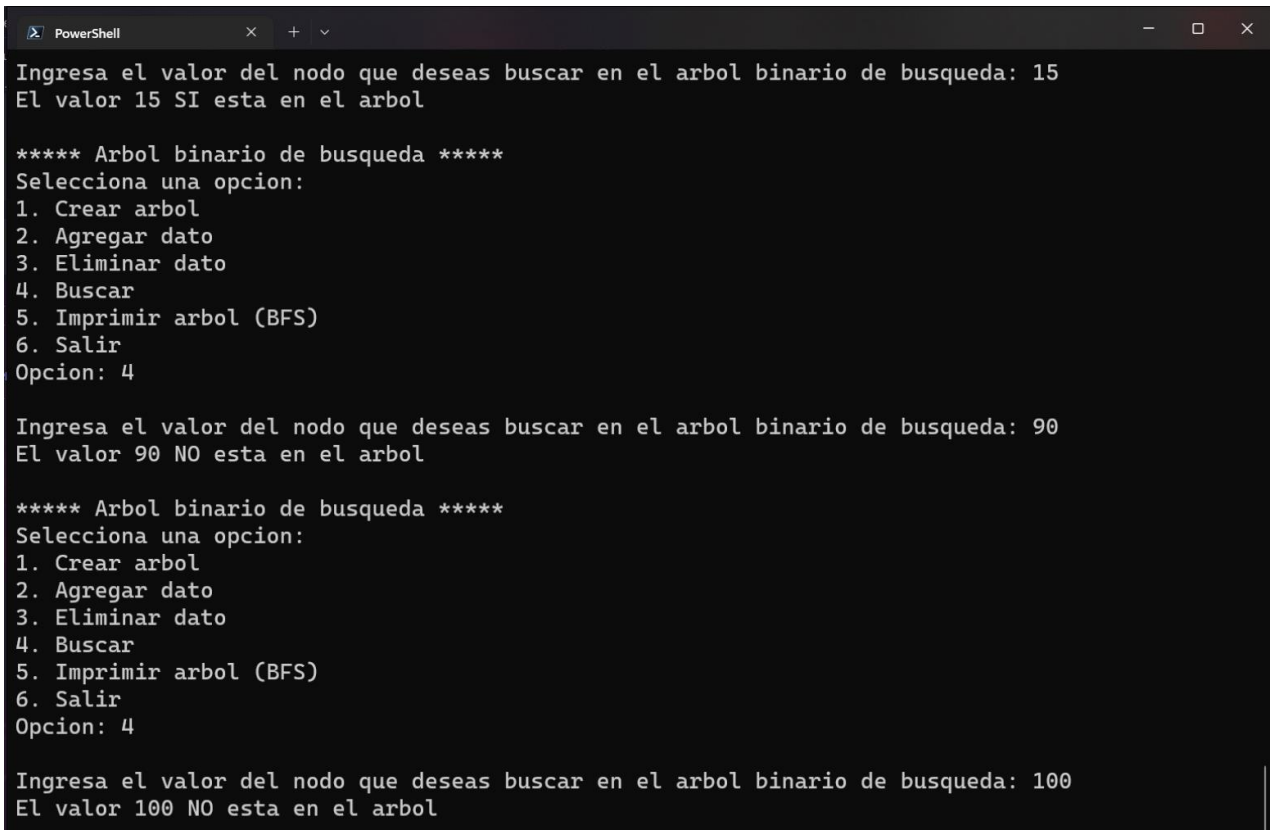


Recorrido BFS

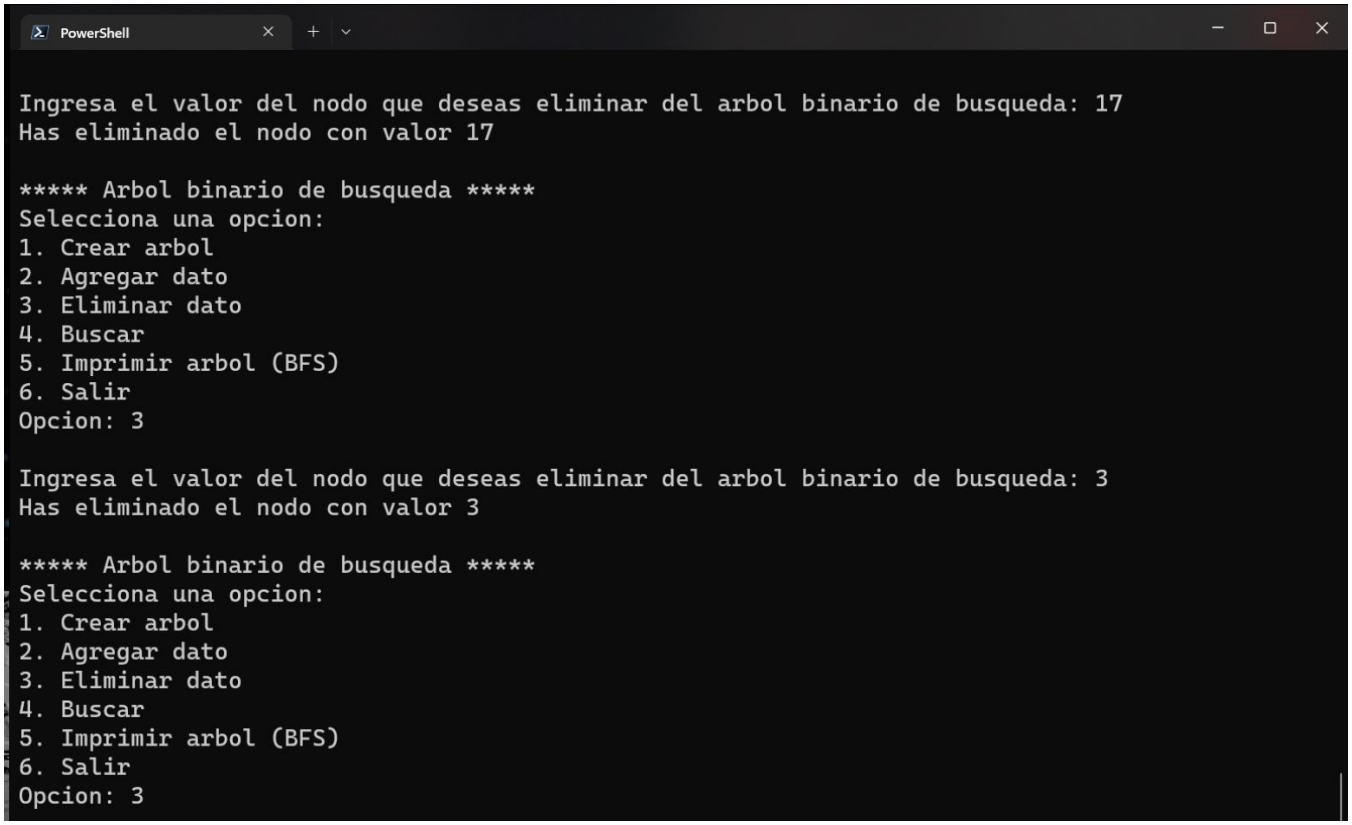


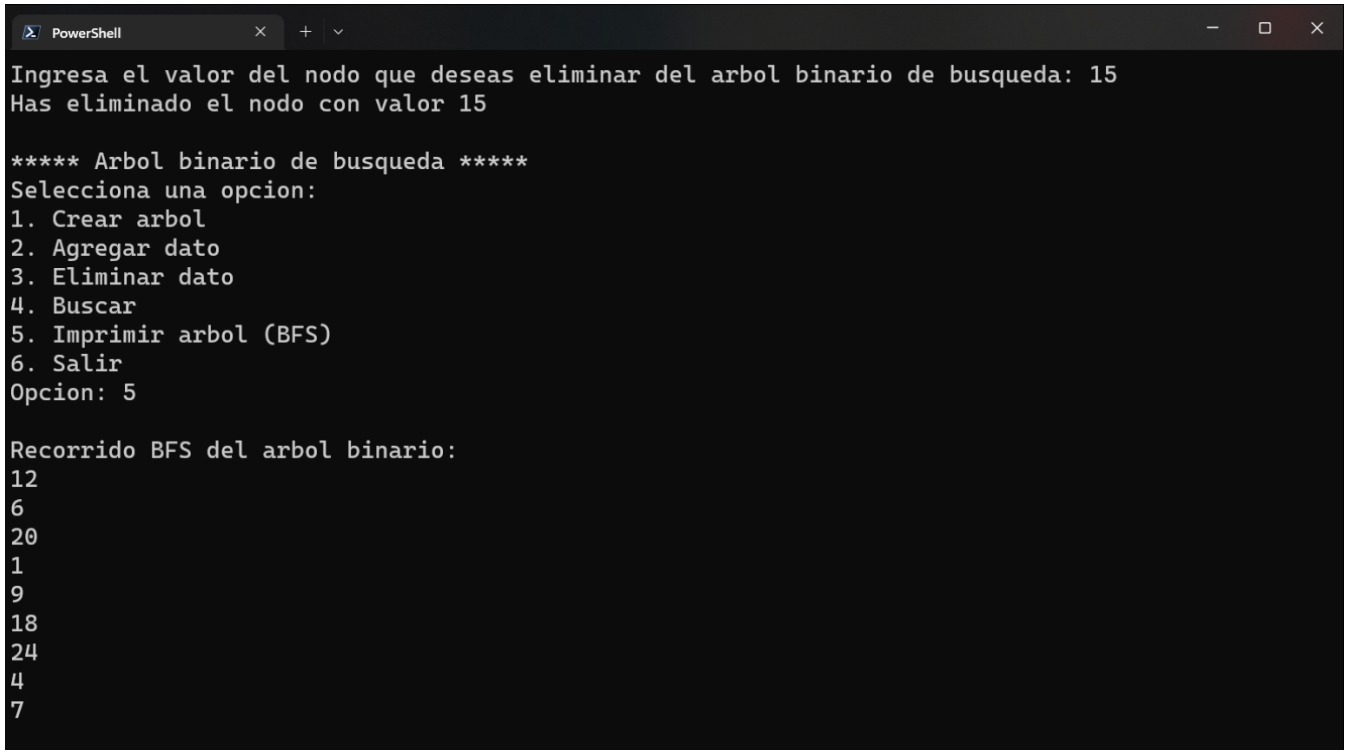
Buscando nodos (17, 9, 15, 90, 100)





Eliminando nodos (17, 3, 15)



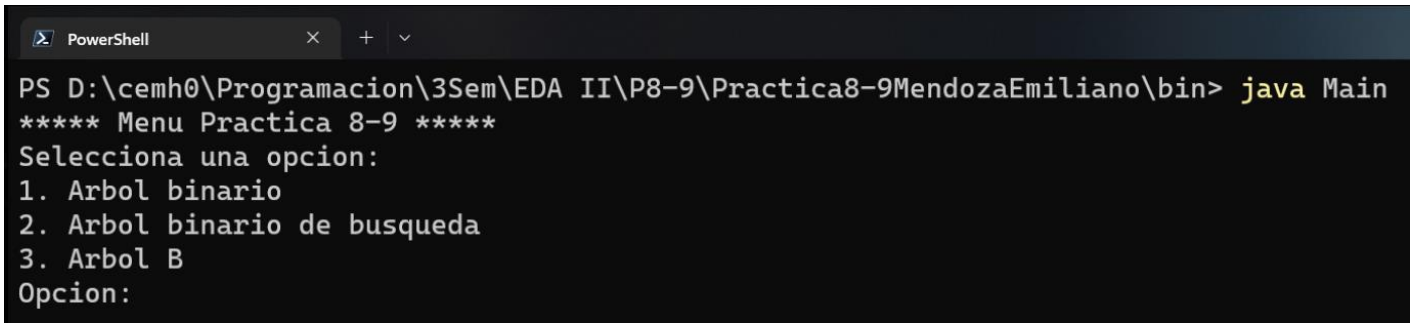


Se agregaron los nodos correctamente, se comprobó el funcionamiento del método de recorrido BFS y el de búsqueda. Además, se eliminaron correctamente los nodos de tres diferentes casos: un nodo hoja (17), un nodo con dos hijos (3) y el nodo raíz (15).

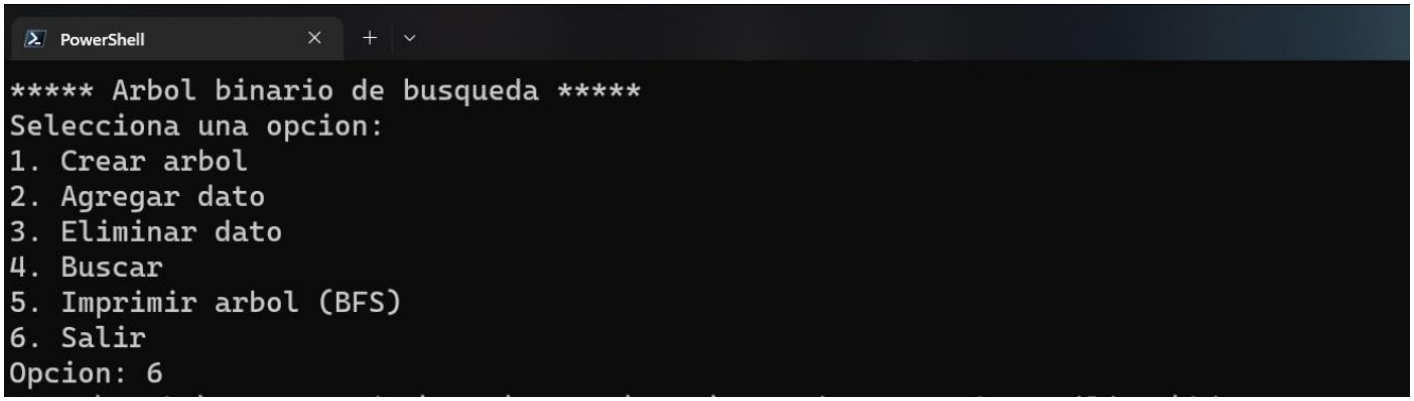
Ejercicio 4. Menú de usuario

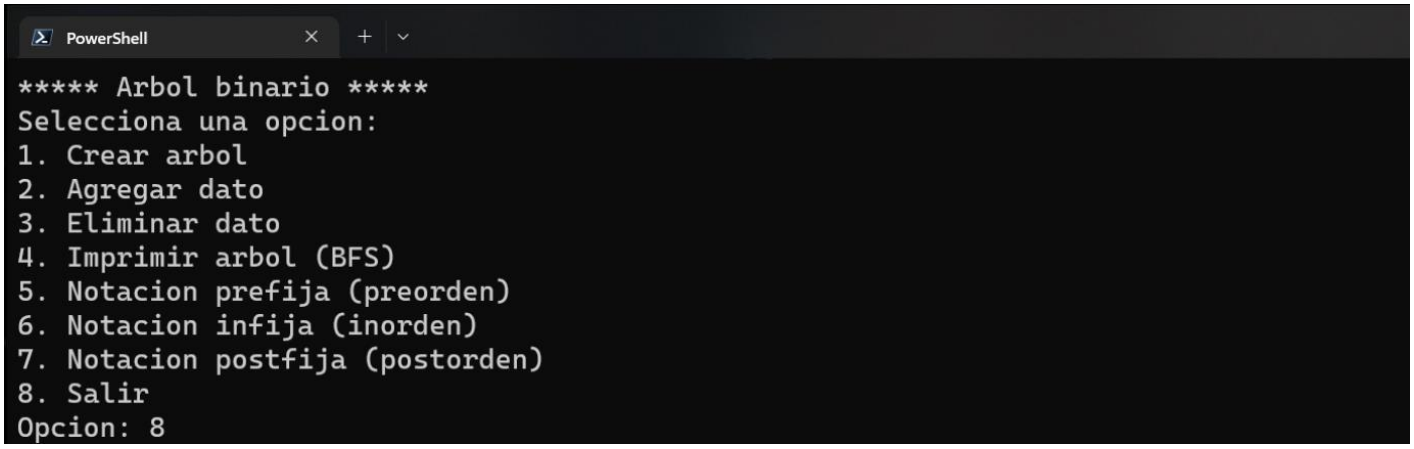
Implementación del menú de usuario

En una estructura switch case, se implementó un menú con las siguientes opciones:



Las opciones 1 y 2 tienen los siguientes submenús respectivamente:





Ejercicio 5. Árbol B

Análisis de la implementación de *BNode.java*

La clase *BNode* está compuesta por cinco atributos, un constructor y seis métodos.

***Atributos*:**

- *int m*: Representa el orden del árbol B, cada nodo debe tener un atributo para representarlo. Un valor numérico entero.
- *ArrayList<Integer> key*: Almacena las claves del nodo.
- *ArrayList<BNode> child*: Contiene referencias de los hijos del nodo.
- *boolean leaf*: Una bandera para indicar si el nodo es una hoja o no.

***Constructores*:**

- *BNode()*: Inicializa todos los atributos con valores por defecto.

***Métodos*:**

- *int getKey(i)*: Método de acceso para una clave en un índice especificado.
- *BNode getChild(i)*: Método de acceso para el hijo de un nodo en un índice especificado.
- *void setKeys(list)*: Método para modificar el arreglo de claves del nodo.
- *void setChildren(list)*: Método para modificar el arreglo de hijos del nodo.
- *int getChildIndex()*: Recorre el arreglo de nodos hermanos para encontrar el índice del nodo.
- *void mostrarLlaves()*: Imprime las claves del nodo.

Análisis de la implementación de *BTree.java*

La clase *BTree* está compuesta por dos atributos, un constructor y ocho métodos.

***Atributos*:**

- *int m*: Representa el orden del árbol B.
- *BNode root*: Referencia a la raíz del árbol.

***Constructores*:**

- *BTree(m)*: Inicializa el árbol B con un nuevo nodo por defecto en la raíz. Su grado es el parámetro especificado *m*.

***Métodos*:**

- *void add(n)*: Dado un parámetro *n* para ingresar una nueva clave al árbol, primero valida que la clave que se pretende ingresar no exista. Luego, encuentra la hoja a la que se debe ingresar y se llama a la función *addToNode* para agregar la clave al nodo.
- *BNode leafNode(nodo, n)*: Método recursivo para encontrar el nodo donde se insertaría una nueva clave.
- *void addToNode(nodo, n)*: Verifica que haya espacio en el nodo. De ser así, se inserta; si no, se hace la división celular.
- *void insert(nodo, n)*: Inserta la clave *n* en el nodo, respetando el ordenamiento de las claves.
- *void mostrarArbol()*: Imprime los nodos del árbol B por nivel, destacando los nodos padre e hijos.
- *void find(value)*: Método recursivo que recorre los nodos del árbol desde la raíz. Si encuentra el valor especificado *value* devuelve *true*.
- *void find(v, n)*: Método recursivo que recorre los nodos del árbol desde un nodo especificado *n*. Si encuentra el valor especificado *v* devuelve *true*.
- *void divisionCelular(nodo, n)*: Al principio verifica si el orden del árbol es impar: en este caso, primero se inserta la clave al nodo antes de elegir la clave de en medio. Luego, se toman en cuenta dos casos

  Caso 1: *Cuando el nodo es la raíz*

  Se crean dos nuevos nodos, el primero con la primera mitad de las claves y el segundo nodo con la segunda mitad (no se incluye al nodo intermedio). El nodo original que llama a la función se vacía y se queda únicamente con la clave intermedia. Si el orden del árbol es par, se agrega la clave en el nuevo nodo que le corresponde. Además, si la raíz deja de ser una hoja, se reacomodan sus hijos con la nueva distribución de las claves. Al final se indica que la raíz ya no es una hoja.

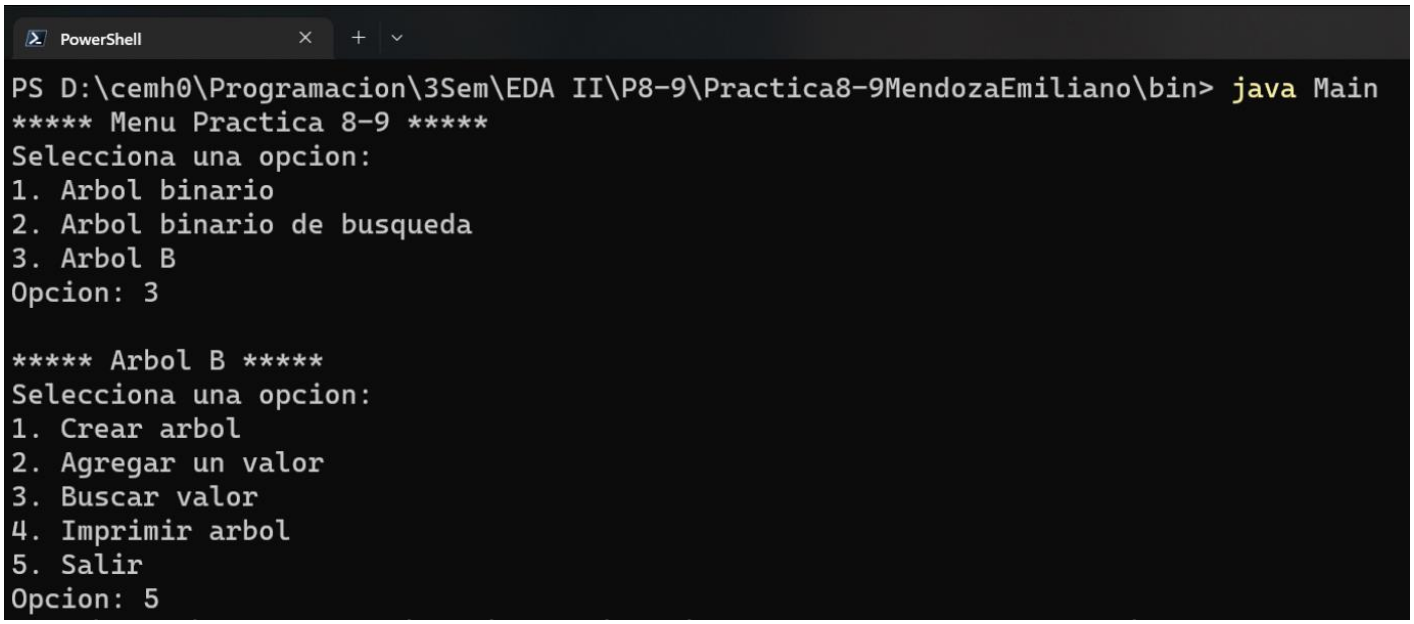
  Caso 2: *Cuando es cualquier otro nodo*

  El nodo actual se queda con la primera mitad de las claves, para la segunda mitad se crea un nuevo nodo. De igual manera, si el orden es par, se inserta la clave especificada en el nodo que le corresponde. Si el nodo deja de ser una hoja, se reacomodan sus hijos con la distribución de las claves.

Ejercicio 6. Creación de un árbol B

Modificación del menú de usuario

Se agregaron las siguientes opciones al submenú de la tercera opción del menú principal:



CONCLUSIONES

Algunos puntos que se concluyen de la primera actividad son los siguientes:

- Un árbol binario no sigue criterios tan estrictos para el ordenamiento de sus nodos, sin embargo, es una representación útil de árboles en general por su restricción de dos hijos máximo por nodo.
- A diferencia de los grafos, con esta implementación no hay una manera directa de obtener el dibujo de un árbol binario a través del recorrido BFS.

Para el segundo ejercicio se puede concluir que:

- Las operaciones para obtener las notaciones de un árbol binario (preorden, inorden y postorden) se pueden obtener de manera recursiva. En esencia son muy parecidas, pero se altera el orden de visita a los nodos.

Al realizar el ejercicio 3 se llegó a las siguientes conclusiones:

- Muchos de los métodos y los atributos para un árbol binario de búsqueda se pueden aprovechar de la implementación de árbol binario utilizando la herencia.
- Se pueden sobrescribir los métodos necesarios para adaptar la funcionalidad de ellos a la de un árbol binario de búsqueda.
- Para este tipo de árboles no podemos elegir en donde ubicar una nueva clave, sin embargo, el recorrido que se hace para hallar una clave en el árbol es bastante rápido.

Luego de realizar el ejercicio 5, se llegó a la siguiente conclusión:

- Un árbol B es muy eficiente en cuanto al uso de memoria por su estructura *half-full*. Los arreglos deben tener, cuando menos la mitad de su capacidad.
- Buscar un valor en un árbol B es también una tarea relativamente rápida.
- Sin embargo, los algoritmos de agregar y eliminar claves son muy complejos y se requiere de más esfuerzo para comprenderlos

Dicho todo lo anterior, considero que se ha cumplido con cada uno de los objetivos planteados para el desarrollo de esta práctica dado que se conocieron e identificaron las características necesarias para implementar y comprender los tipos de árboles, así como los algoritmos de agregación,

eliminación y búsqueda de claves. Además, todo lo anterior se realizó con un enfoque orientado a objetos.

Como comentario final, considero que no fue tan sencillo implementar algunos algoritmos como la eliminación de claves, al menos, en el enfoque que escogí. Al considerar muchos casos posibles se extendió mucho el código obtenido. Sin embargo, considero que hay mejores opciones como utilizar la recursividad para la eliminación, búsqueda y agregación de claves. Considero que, visual y conceptualmente los árboles no son tan complicados de comprender, sin embargo, a nivel de código se pueden complicar mucho como en el caso de los árboles B. Cabe destacar que considero que las aplicaciones de los árboles son diversas y que incluso esta implementación podría escalarse para manejar otro tipo de datos y algún otro tipo de árbol.