UNIVERSIDAD CATÓLICA DE SANTA MARÍA
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

LABORATORIO 07:

BINARY SEARCH TREE - BST

I
OBJETIVOS

- Analizar el comportamiento de los árboles binarios de búsqueda
- Analizar las diferentes formas de representar los árboles binarios de búsqueda
- Implementar recorridos en los árboles binarios: inorden, preorden y postorden
- Implementar las operaciones de los árboles binarios de búsqueda

II
TEMAS A TRATAR

- Definición de árbol binario.
- Representación de un árbol binario.
- Recorridos de un árbol binario.
- Árbol de búsqueda binario (BST).
- Operaciones de los BST.
- Aplicaciones de árboles binarios.
- Ejercicios.

III
DURACIÓN DE LA PRACTICA

- Dos sesiones (4 horas).

IV
RECURSOS

- Equipos de cómputo.
- Lenguaje de programación Java.
- IDE de desarrollo. Eclipse, VCode.
- Herramientas de seguimiento de versiones: Git & GitHub.

V

MARCO TEÓRICO

INTRODUCCIÓN

Los arreglos y las listas son ejemplos de estructuras de datos lineales. Cada elemento tiene:

- un predecesor único (excepto el primer elemento de la lista);
- un sucesor único (excepto el último elemento de la lista).

¿Existen otros tipos de estructuras?

Un grafo es una estructura de datos no lineal, ya que cada uno de los sus elementos, designados por nosotros, pueden tener más de un predecesor o más de un sucesor.

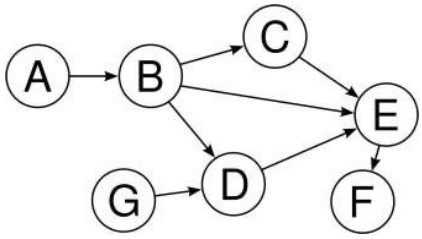


Figura 7.1. Grafo

ÁRBOLES

Un árbol es un tipo específico de grafo, cada elemento, designado por nodo, tiene cero o más sucesores, pero sólo un predecesor (excepto el primer nodo, que se llama raíz del árbol);
Un ejemplo de un árbol:

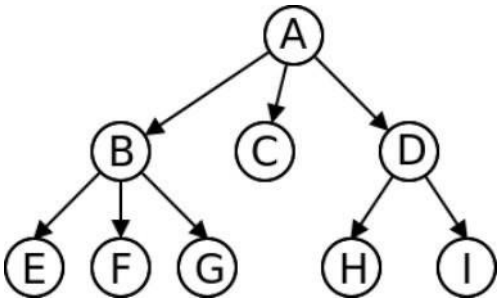


Figura 7.2. Árbol terciario

Los árboles son estructuras especialmente adecuadas para representar información organizada en jerarquías. Algunos ejemplos:

- La estructura de directorios (o carpetas) de un sistema de archivos
- Un árbol genealógico de una familia
- Un árbol de la vida

Figura 7.3. Estructuras jerárquicas

¿Qué es un Árbol Binario?

Un árbol binario es un tipo especial de árbol, en el cual ningún nodo de un árbol puede tener más de 2 nodos. Puede definirse como:

- Un árbol binario T, es una estructura vacía o
- T tiene un nodo especial llamado root (raiz)
- T tiene al menos uno de dos nodos, llamados subárbol izquierdo (Left Tree - LT) y subárbol derecho (Righ Tree - RT).
    - LT y RT son arboles binarios al mismo tiempo

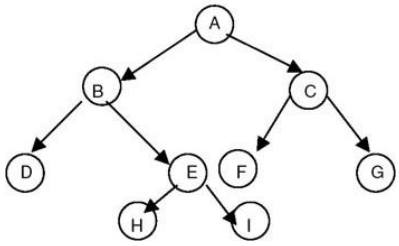


Figura 7.4. Árbol binario

Terminología

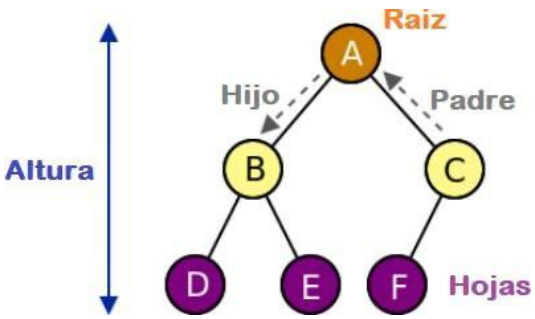


Figura 7.5. Terminología relacionada a árboles

- El predecesor (único) de un nodo se llama **padre**. Ejemplo: El padre de B esA; El padre de C también es A.
- Los sucesores de un nodo son sus **hijos**. Ejemplo: Los hijos de A son B y C.
- El **grado** de un nodo es su número de hijos. Ejemplo: A tiene 2 hijos, C tiene 1 hijo.
- Una **hoja** es un nodo sin hijos, es decir, de grado 0.Ejemplo: D, E y F son nodos hoja.
- La **raíz** es el único nodo sin padre.
- Un **subárbol** es un subconjunto de los nodos (vinculados) del árbol. Ejemplo: {B,D,E} son un subárbol.
- Los arcos que conectan los nodos se llaman **ramas**.
- La secuencia de ramas entre dos nodos se llama **camino**. Ejemplo: A-B-D es el camino entre A y D.
- La **longitud** de un camino es el número de ramas que contiene; Ejemplo: A- B-D tiene longitud 2.
- La **profundidad** de un nodo es la longitud del camino desde la raíz hasta este nodo (la profundidad de la raíz es cero);

  Ejemplo: B tiene profundidad 1, D tiene profundidad 2.
- La **altura** de un árbol es la profundidad máxima de un nodo en el árbol. Ejemplo: El árbol de la figura tiene una altura de 2.

Representación de un árbol binario

Los árboles binaries se pueden representar en memoria a través del:

- Uso de arreglos
- Uso de estructuras enlazadas

En el curso utilizaremos las estructuras enlazadas para representar los árboles, pues es la mejora forma y las más utilizada. Para definir la estructura de un nodo, se tendrá la clase **Node.**

```
class Node<T>{
   private T data;
   private Node<T> right;
   private Node<T> left;
    //otros métodos
};
```

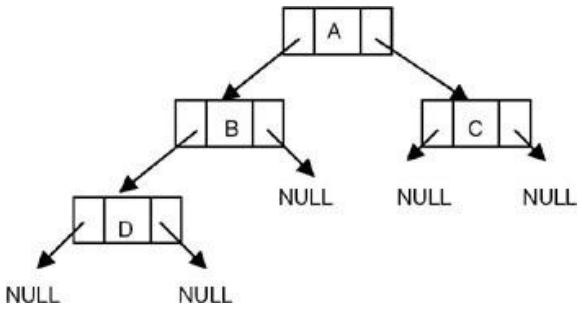


Figura 7.6. Estructura de un nodo

Recorridos de un árbol binario

- **InOrder**: Subárbol Izquierdo – Nodo (Raíz) – Subárbol Derecho
- **PostOrder**: Subárbol Izquierdo – Subárbol Derecho – Nodo (Raíz)
- **PreOrder:** Nodo (Raíz) – Subárbol Izquierdo – Subárbol Derecho

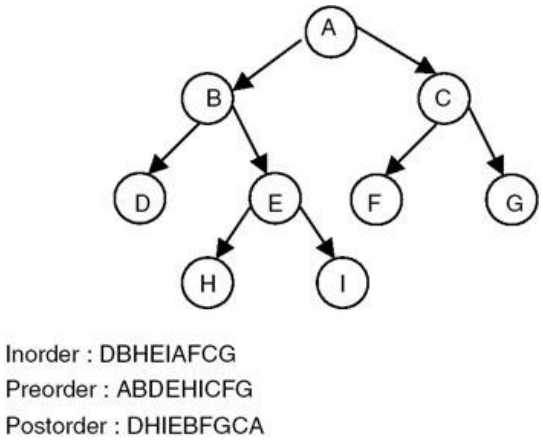


Figura 7.7. Recorridos de un árbol binario

BINARY SEARCH TREE (BST): Árbol de Búsqueda Binaria

Un BST es un árbol binario en el cual se debe cumple la siguiente regla:

- Cada dato de cualquier nodo del subárbol izquierdo es menor que el dato de la raíz.
- Un dato de cualquier nodo del subárbol derecho es mayor que el datos de la raíz.

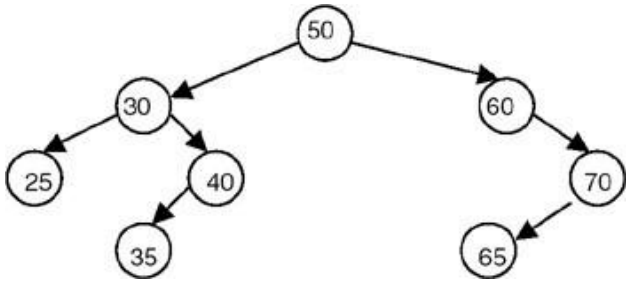


Figura 7.8. Árbol de búsqueda binaria

Operaciones de los BSTs

Las operaciones básicas son:

- **destroy()**: Elimina todos los elementos del BST dejándolo vacío
- **isEmpty()**: Verifica si el BST está vacío
- **insert(x):** Agrega el element '**x**' al BST.
- **remove(x):** Elimina el element '**x**' del BST.
- **search(x):** verifica la existencia del elemento '**x**' en el BST.
- **InOrder()**, **PostOrder(), PreOrder()**: recorren los elementos del árbol según corresponda al recorrido.

Adicionalmente se puede incluir otras operaciones que sean necesarias según sea el caso.

- **Búsqueda en un BST**

  Para buscar un elemento en el BST se debe de considerar la regla: **LEFT<Root<RIGHT**.

  En la figura 7.9, se observa el proceso de búsqueda de los elementos 4 y 18.

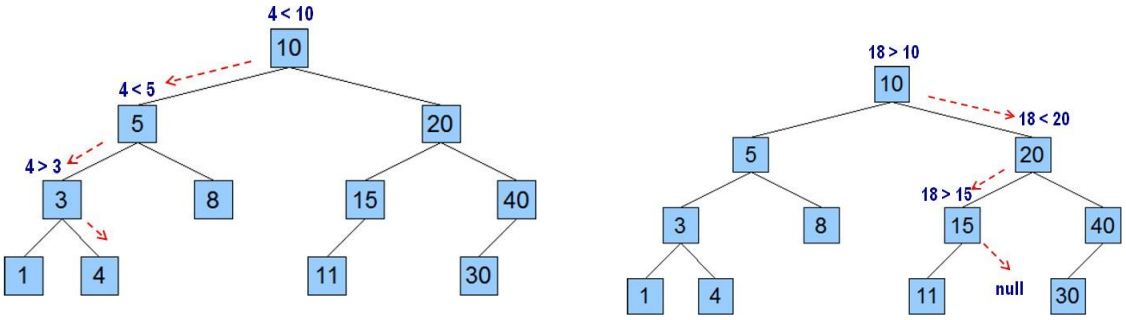


Figura 7.9. Proceso de búsqueda en un BST

- **Inserción en un BST**

  Para insertar un elemento se debe de considerar la regla del BST: **LEFT<Root<RIGHT**.

  Por ejemplo, se desea almacenar los números 8, 3, 1, 20, 10, 5 y 4 en un BST. La figura 7.10 nos muestra el proceso de inserción.

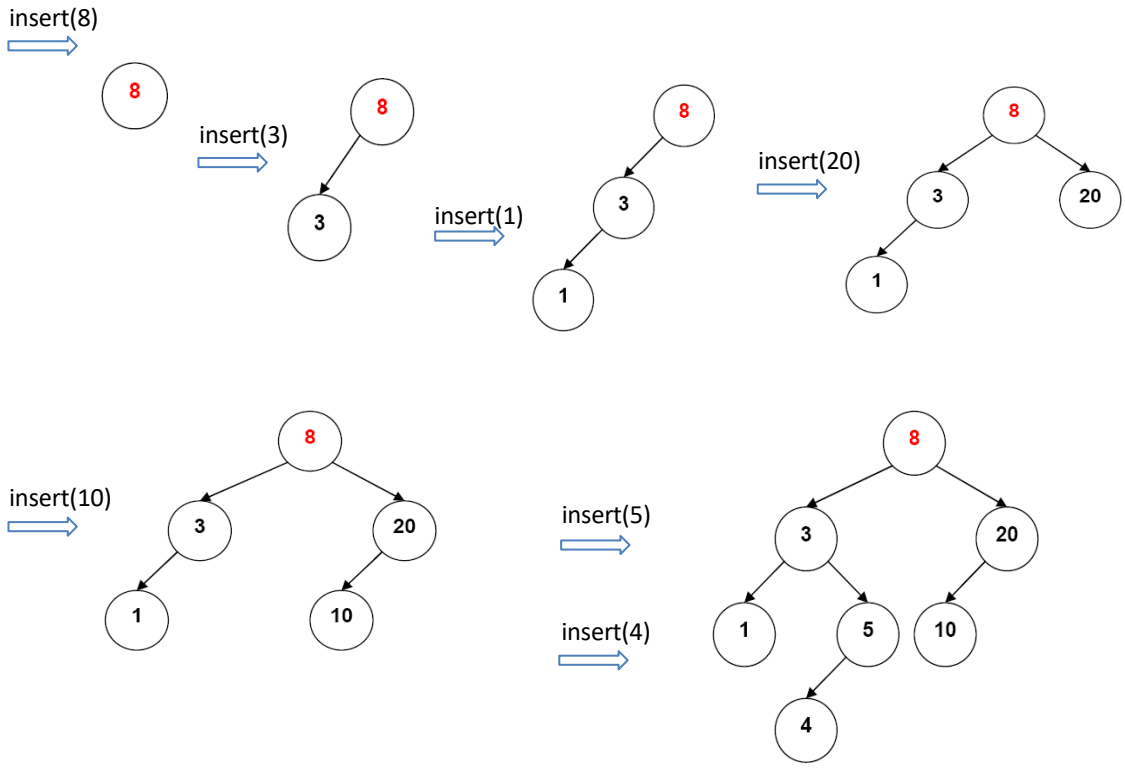


Figura 7.10. Proceso de inserción en un BST

- **Eliminación en un BST**

  En la eliminación de un elemento del BST, se identifican tres escenarios posibles:

  - **Caso 1**: Si se trata de una hoja se elimina directamente
  - **Caso 2**: Si tiene un único hijo se elimina el nodo haciendo que su nodo padre pase a referenciar a su nodo hijo
  - **Caso 3**: Si tiene dos hijos. Se puede proceder de alguna de las dos formas siguientes:
    a) Se sustituye el nodo por el menor elemento de su Subárbol Derecho (***sucesor en InOrden***)
    Ahora el objetivo es eliminar el nodo correspondiente a dicho menor elemento. Para esto se debe de volver a validar en que caso de los tres nos encontramos.
    b) O de forma simétrica, buscar el ***antecessor en Inorden***

  En la figura 7.11. podemos observar ejemplos de eliminación en un BST.

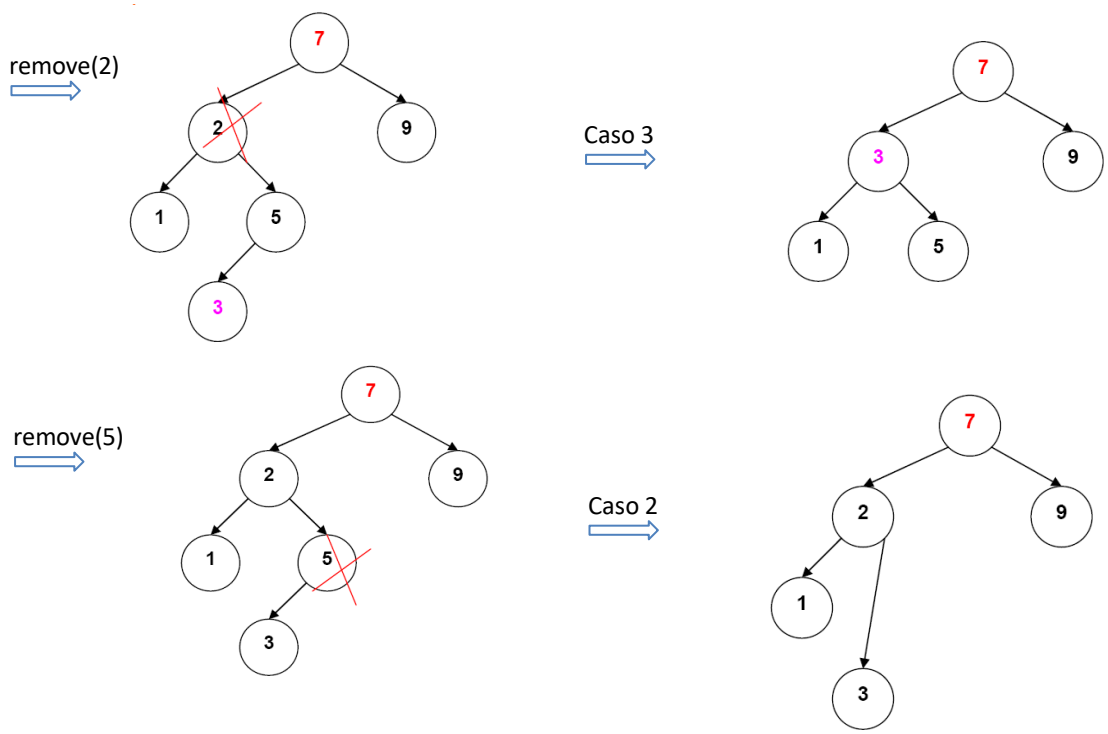


Figura 7.11. Eliminación del 2 y del 5 de un BST

VI

ACTIVIDADES

1. Aplicaciones de árboles binarios

Ingresar a los siguientes enlaces y probar cada una de las opciones de los applets y observe la forma como se realizan cada una de las operaciones en un BST.

a) **https://visualgo.net/en/bst**
b) **https://www.cs.usfca.edu/~galles/visualization/BST.html**

2. Representación Secuencial de un árbol binario en memoria

Revisar el contenido del siguiente enlace, analice y comprenda como se almacena de manera secuencial en un arreglo lineal un árbol binario en memoria y la manera como se recupera para construir dicho árbol binario.

https://www.youtube.com/watch?v=8UwJPGUXMsQ&t=370s

Luego de ver, analizar y comprender del video, **realice en una hoja la representación de manera secuencial en memoria utilizando un arreglo lineal y un puntero, de los siguientes arboles binarios:**

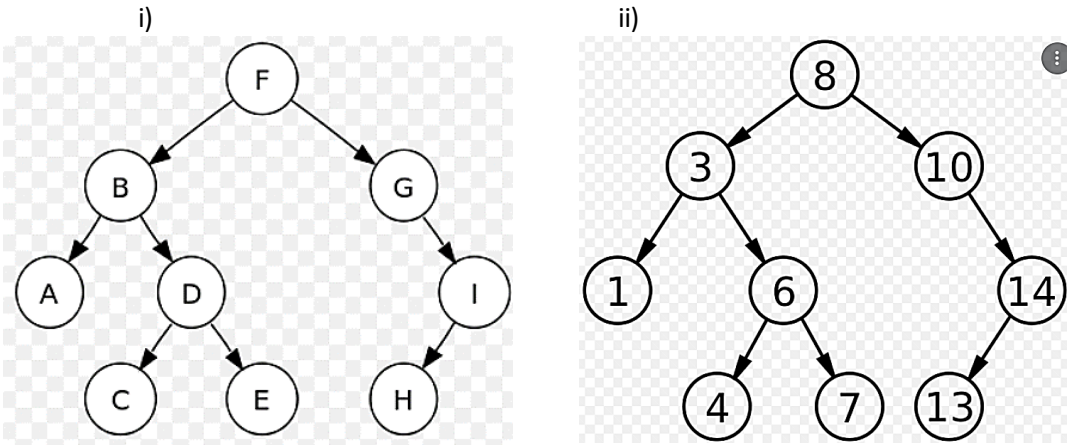


Figura 7.12. Árboles BST

3. Implementación de TAD: BinarySearchTree Generico.

- Cree un nuevo proyecto.

4. Implementación de Excepciones.

- Agregue un paquete **Exceptions**, realizada en el laboratorio anterior.
- En la clase **ItemDuplicated**, valida que el dato ya existe en la estructura de datos.

```
public class ItemDuplicated extends Exception {
    public ItemDuplicated(String msg) {
        super(message:msg);
    }
    public ItemDuplicated() {
        super();
    }
}
```

- En la clase **ItemNoFound**, valida que el dato no se encuentra en la estructura de datos.
- En la clase **ExceptionIsEmpty**, valida que este vacía la estructura de datos.
- s

5. Implementación de la Interfaz Genérica: BinarySearchTree

- Agregue en un paquete: **bstreeInterface**, implemente la interfaz: **BinarySearchTree**.

```
public interface BinarySearchTree<E> {
    void insert(E data) throws ItemDuplicated;
    E search(E data) throws ItemNotFound;
    void delete(E data) throws ExceptionIsEmpty;
    boolean isEmpty();
}
```

6. Implementación de operaciones del BST en TAD BST Genérico

- En un paquete: **bstreelinklistinterfgeneric**, cree la clase: **LinkedBST**, para implementar los atributos y métodos necesarios y de forma progresiva, para implementar la interfaz **BinarySearchTree**

```
public class LinkedBST<E> implements BinarySearchTree<E>{
    class Node{
        public E data;
        public Node left;
        public Node right;

        public Node(E data){
            this (data, null, null)
        }
        public Node(E data, Node left, Node right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
    }

    private Node root;
    public LinkedBST() {
        this.root = null;
    }

    public void insert(E data) throws ItemDuplicated{
        // Agregar codigo aqui
    }
    public void delete(E data) throws ExceptionIsEmpty{
        // Agregar codigo aqui
    }
    public E search(E data) throws ItemNotFound{
        // Agregar codigo aqui
    }
    public String toString(){
        // Agregar codigo aqui
    }
}
```

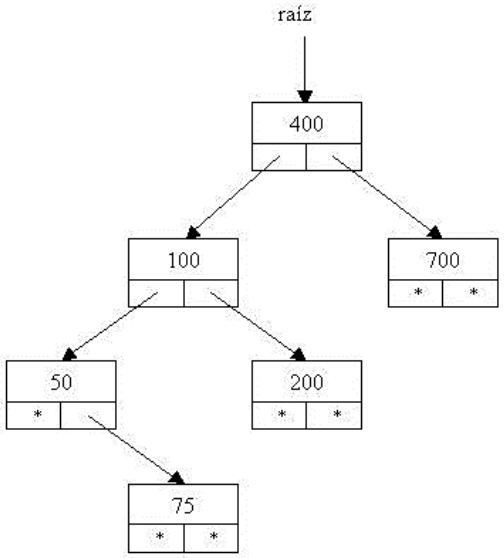
- En la clase: **LinkedBST**, implementar el método publico **insertar(E data)** para agregar un elemento en el **BST**, validando con la excepción **ItemDuplicated,** de existir duplicado.
- En la clase: **LinkedBST**, implementar el método publico **search(E data)** para buscar un elemento en el **BST**, validando con la excepción **ItemNoFound,** de NO ser encontrado.
- En la clase: **LinkedBST**, implementar el método publico **delete(E data)** para eliminar un elemento en el **BST**, validando con la excepción **ExceptionIsEmpty,** de estar vacío el BST.
- En la clase: **LinkedBST**, implementar el método publico **toString()** para mostrar la cadena que contiene la información en el **BST**, de usando **cadenas de texto formateadas**.

7. Implementación del recorrido In – Orden.

- Implementar en la clase **LinkedBST** un método privado **inOrder** para que realice el recorrido**: izquierda, cabecera, derecha**.
- Implementar el método de acuerdo al algoritmo y el ejemplo del árbol a continuación.
- Probar dicho recorrido en la clase de **Prueba**.

Recorrer el subárbol izquierdo en entreorden.
Recorrer el subárbol izquierdo en entreorden.
Recorrer el subárbol izquierdo en entreorden.
Vacío
Visitar la raiz: 50
Recorrer el subárbol derecho en entreorden.
Recorrer el subárbol izquierdo en entreorden.
Vacío
Visitar la raiz : 75
Recorrer el subárbol derecho en entreorden.
Vacío
Visitar la raiz: 100
Recorrer el subárbol derecho en entreorden.
Recorrer el subárbol izquierdo en entreorden.
Vacío
Visitar la raiz: 200
Recorrer el subárbol derecho en entreorden.
Vacío

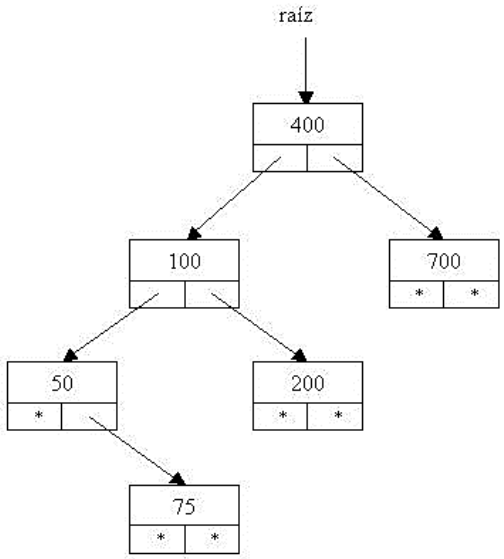
Visitar la raiz: 400
Recorrer el subárbol derecho en entreorden.
Recorrer el subárbol izquierdo en entreorden.
Vacío
Visitar la raiz: 700
Recorrer el subárbol derecho en entreorden.
Vacío



8. Implementación del recorrido Pre – Orden.

- Implementar en la clase **LinkedBST** un método privado **preOrder** para que realice el recorrido**: cabecera, izquierda, derecha**.
- Implementar el método de acuerdo al algoritmo y el ejemplo del árbol a continuación.
- Probar dicho recorrido en la clase de **Prueba**.

Visitar la raiz: 400
Recorrer el subárbol izquierdo en preorden.
Visitar la raiz: 100
Recorrer el subárbol izquierdo en preorden.
Visitar la raiz: 50
Recorrer el subárbol izquierdo en preorden.
Vacío
Recorrer el subárbol derecho en preorden.
Visitar la raiz: 75
Recorrer el subárbol izquierdo en preorden.
Vacío
Recorrer el subárbol derecho en preorden.
Vacío
Recorrer el subárbol derecho en preorden.
Visitar la raiz: 200
Recorrer el subárbol izquierdo en preorden.
Vacío
Recorrer el subárbol derecho en preorden.
Vacío
Recorrer el subárbol derecho en preorden.
Visitar la raiz: 700
Recorrer el subárbol izquierdo en preorden.
Vacío
Recorrer el subárbol derecho en preorden.
Vacío



9. Implementación del recorrido Post – Orden.

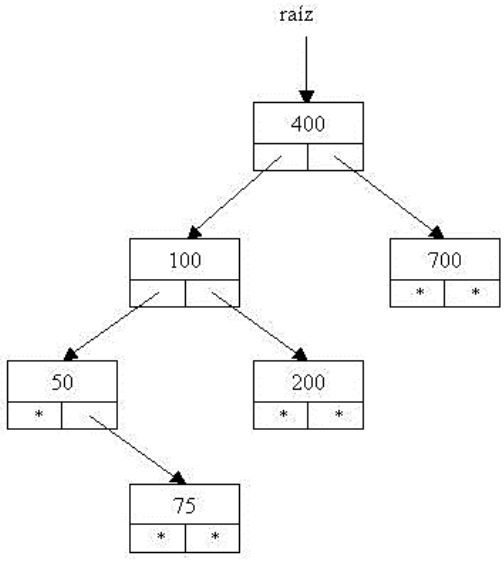
- Implementar en la clase **LinkedBST** un método privado **postOrder** para que realice el recorrido**: cabecera, izquierda, derecha**.
- Implementar el método de acuerdo al algoritmo y el ejemplo del árbol a continuación.
- Probar dicho recorrido en la clase de **Prueba**.

Recorrer el subárbol izquierdo en postorden.
Recorrer el subárbol izquierdo en postorden.
Recorrer el subárbol izquierdo en postorden.
Vacío
Recorrer el subárbol derecho en postorden.
Recorrer el subárbol izquierdo en postorden.
Vacío
Recorrer el subárbol derecho en postorden.
Vacío
Visitar la raiz: 75
Visitar la raiz: 50
Recorrer el subárbol derecho en postorden.
Recorrer el subárbol izquierdo en postorden.
Vacío
Recorrer el subárbol derecho en postorden.
Vacío
Visitar la raiz: 200
Visitar la raiz: 100

Recorrer el subárbol derecho en postorden.
Recorrer el subárbol izquierdo en postorden.
Vacío
Recorrer el subárbol derecho en postorden.

Visitar la raiz: 700

Visitar la raiz: 400



10. Implementación el mínimo y máximo de un arbol.

- Implementar en la clase **LinkedBST** un método privado **findMinNode** utilizando el método publico **search()**, encontrar el **menor** valor del subárbol con raíz node y devuelva el valor donde se encuentra el nodo con este valor en el **BST**, validando con la excepción **ItemNoFound,** de NO encontrarlo en el BST
- Implementar en la clase **LinkedBST** un método privado **findMaxNode** utilizando el método publico **search()**, encontrar el **mayor** valor del subárbol con raíz node y devuelva el valor donde se encuentra el nodo con este valor en el **BST**, validando con la excepción **ItemNoFound,** de NO encontrarlo en el BST
- Probar dicho recorrido en la clase de **Prueba**.

VII

EJERCICIOS

01. A la clase **LinkedBST** agregue los siguientes métodos que son parte de la interface del árbol:

a. Un método **destroyNodes()** que elimine todos los nodos de un BST, considerando las esceociones **ExceptionIsEmpty**.

b. Un método **countAllNodes()** que retorne el número de nodos no-hojas de un BST.

c. Un método **countNodes()** que retorne el número de nodos no-hojas de un BST.

d. Un método **height**(x) que retorne la altura del subárbol cuya raíz tenga como dato un valor igual a 'x'. Este método debe no debe ser recursivo, sino utilizar iteratividad. En caso no exista tal subárbol deberá retornar -1. **Nota**: en caso de necesitar utilizar estructuras adicionales, sólo podrá utilizar aquellas que haya implementado en laboratorios previos.

e. Un método **amplitude(Nivel)** que retorne la amplitud o la anchura de todo el árbol, utilizando el método height, es decir el número máximo de nodos que existen en un nivel determinado,

02. Agregue los siguientes métodos:

a. Se define el área de un árbol binario como el número de nodos hojas multiplicado por la altura del árbol. Agregue un método público **areaBST()** a la clase **LinkedBST** que retorne el área del árbol binario de búsqueda. Este método deberá utilizar iteratividad,

no recursividad. **Nota**: en caso de necesitar utilizar estructuras adicionales, sólo podrá utilizar aquellas que haya implementado en laboratorios previos.

b. Implementar un método **drawBST()** para dibujar o graficar el árbol y mostrar los nodos y las aristas que los vinculan utilizando el método **toString**.

c. En la clase **Prueba** implemente el método **sameArea()** que retorne si dos árboles binarios diferentes tienen la misma área.

03. La representación entre paréntesis con sangría de un árbol T cualquiera es una forma de visualizar su contenido, la misma que usa sangría y saltos de línea, similar a como se ilustra en la figura 7.13. cuya estructura se trata de un árbol cuaternario. Implemente un método público en la clase **LinkedBST** llamado **parenthesize()** que imprima esta forma de visualización del árbol BST. Si requiere utilizar alguna estructura adicional, sólo puede usted utilizar las estructuras (TAD) que haya usted implementado en laboratorios previos.

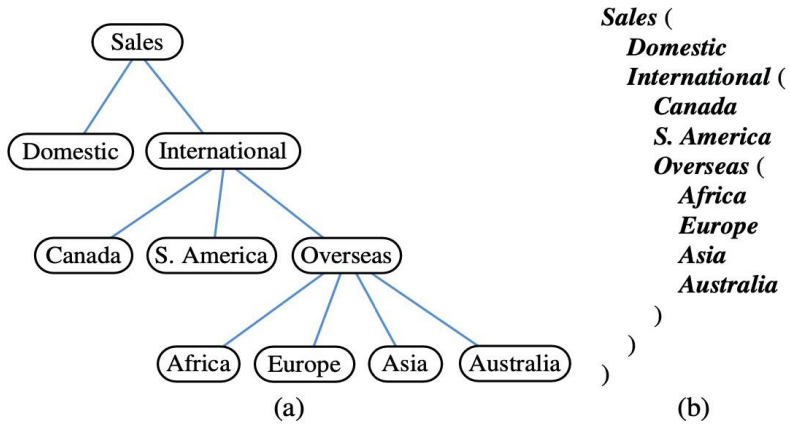


Figura 7.13. Árbol BST y su representación parenthetic

VIII

BIBLIOGRAFÍA

- Weiss M., Data Structures & Problem Solving Using Java, Addison-Wesley, 2010. Chapter 18.
- Cutajar J. Beginning Java Data Structures and Algorithms, Packt Publishing, 2018. Chapter 3.
- Goodrich M., Tamassia R. and Goldwasser M., Data Structures & Algorithms, Wiley, 2014. Chapter 8

RÚBRICA PARA LA CALIFICACIÓN DEL LABORATORIO

CRITERIO A SER EVALUADO		Nivel				
		A	B	C	D	NP
R1	Uso de estándares y buenas prácticas de programación	2.0	1.6	1.2	0.6	0
R2	Desarrollo de ejercicios propuestos en laboratorio, lo que se evidencia en los registros realizados en los repositorios.	4.0	3.2	2.4	1.2	0
R3	Solución de las actividades a través de la definición y uso adecuado de árboles binarios de búsqueda, lo que se evidencia en el registro oportuno de los commits en los repositorios correspondientes, así como de las pruebas realizadas para su verificación.	4.0	3.2	2.4	1.2	0
R4	Solución de los ejercicios a través de la definición y uso adecuado de árboles binarios de búsqueda, lo que se evidencia en el registro oportuno de los commits en los repositorios correspondientes, así como de las pruebas realizadas para su verificación.	5.0	4.0	3.0	1.5	0
R5	Informe bien redactado, ordenado, bien detallado de las actividades y ejercicios resueltos, y de acuerdo con las normas establecidas. Mostrar evidencia suficiente de lo efectuado.	5.0	4.0	3.0	1.5	0
	Total	**20**				

A: Muy destacado.
B: Destacado.
C: Satisfactorio.
D: Deficiente.
NP: Muy deficiente o No presenta