
	UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA	
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		
Aprobación: 2022/03/01	Código: GUIA-PRLD-001	Página: 1

GUÍA DE LABORATORIO

(formato docente)

INFORMACIÓN BÁSICA					
ASIGNATURA:	ESTRUCTURA DE DATOS Y ALGORITMOS				
TÍTULO DE LA PRÁCTICA:	ÁRBOL BINARIO SIMPLE Y DE BÚSQUEDA				
NÚMERO DE PRÁCTICA:	06	AÑO LECTIVO:	2025 – A	NRO. SEMESTRE:	TERCERO III
TIPO DE PRÁCTICA:	INDIVIDUAL	X			
	GRUPAL	—	MÁXIMO DE ESTUDIANTES	00	
FECHA INICIO:	09/06/2025	FECHA FIN:	13/06/2025	DURACIÓN:	90 minutos.
RECURSOS A UTILIZAR: <ul style="list-style-type: none"> • REPOSITORIO GITHUB: https://github.com/JosueClaudioQP/EDA-Lab • Lenguaje de Programación Java. • Ide Java Eclipse/Visual Studio Code. 					
DOCENTE(s): <ul style="list-style-type: none"> • Mg. Ing. Rene Alonso Nieto Valencia. 					
ALUMNO: <ul style="list-style-type: none"> • Josué Claudio Quispe Paucar 					

OBJETIVOS/TEMAS Y COMPETENCIAS	
OBJETIVOS: <ul style="list-style-type: none"> • Aprenda Pilas y Colas. • Aplicar conceptos elementales de programación a resolver utilizando POO en problemas de algoritmos. • Desarrollar pruebas. 	
TEMAS: <ul style="list-style-type: none"> • Árbol Binario Simple. • Árbol Binario de Búsqueda. • Operaciones BST. 	
COMPETENCIAS	C.a
	C.b
	C.c
	C.d

CONTENIDO DE LA GUÍA

I. EJERCICIO/PROBLEMA RESUELTO POR EL DOCENTE

En un editor Java, realizar la integración de los siguientes ejercicios, revisar y mostrar los resultados obtenidos y realizar una explicación del funcionamiento de forma concreta y clara.

Implementación Nodo Genérico.

```
public class Node<T> {  
    private T data;  
    private Node<T> left;  
    private Node<T> right;  
}
```

Se implementa la clase Nodo, el cual se usará para construir nuestros árboles.

```
1 package ProblemasResueltos;  
2  
3 public class Nodo<T>{  
4     private T data;  
5     private Nodo<T> left;  
6     private Nodo<T> righth;  
7  
8     public Nodo(T data, Nodo<T> left, Nodo<T> righth){  
9         this.left = left;  
10        this.righth = righth;  
11        this.data = data;  
12    }  
13  
14    public Nodo(T data){  
15        this(data, null, null);  
16    }  
17  
18    public T getData() {  
19        return data;  
20    }  
21  
22    public void setData(T data) {  
23        this.data = data;  
24    }  
25  
26    public Nodo<T> getLeft() {  
27        return left;  
28    }  
29  
30    public void setLeft(Nodo<T> left) {  
31        this.left = left;  
32    }  
33  
34    public Nodo<T> getRighth() {  
35        return righth;  
36    }  
37  
38    public void setRighth(Nodo<T> righth) {  
39        this.righth = righth;  
40    }  
41 }
```

Implementación Clase BST Genérico.

```
public class BST<T extends Comparable<T>> {  
    // Ingrese codigo aqui  
}
```

```
package ProblemasResueltos;  
  
public class BST <T extends Comparable<T>>{  
    public Nodo<T> root;  
  
    public BST(){  
        this.root = null;  
    }  
  
    /*  
    public void destroy(){}  
    public void isEmpty(){}  
    public T insert(T x){}  
    public T remove(T x){}  
    public T search(T x){}  
    public void Min(){}  
    public void Max(){}  
    public void Predecesor(){}  
    public void Sucesor(){}  
    public void InOrder(){}  
    public void PostOrder(){}  
    public void PreOrder(){}  
    */  
}
```

II. EJERCICIOS/PROBLEMAS PROPUESTOS

Usando la clase genérica: `public class BST<T>` y `public class Node<T>` usar métodos genéricos `public E Metodo()`

- Ejercicio 1:** Implementar un arbol binario de búsqueda ABB o BST agregando todas las operaciones: `destroy()`, `isEmpty()`, `insert(x)`, `remove(x)`, `search(x)`, `Min()`, `Max()`, `Predecesor()`, `Sucesor()`, `InOrder`, `PostOrder()`, `PreOrder()`.
Implementar una clase Test para probar los métodos y mostrar los resultados. utilizando clases y métodos genéricos.

Primero se implementa la clase Nodo para empezar nuestro BST genérico.

```
1 package ProblemasPropuestos.Ejercicio1;
2
3 public class Nodo<T> {
4     public T data;
5     public Nodo<T> left;
6     public Nodo<T> right;
7
8     public Nodo(T data, Nodo<T> left, Nodo<T> right){
9         this.left = left;
10        this.right = right;
11        this.data = data;
12    }
13
14    public Nodo(T data){
15        this(data, left:null, right:null);
16    }
17 }
18
```

Los atributos referencias al nodo, hijo derecho e izquierdo. El constructor almacena cada uno de estos datos y luego será instanciado en la clase BST.

Se continua con la clase BST, el cuál será genérico y hereda de la interfaz comparable, lo que nos servirá para comparar datos no primitivos.

```
3 public class BST<T extends Comparable<T>> {
4     public Nodo<T> root;
5
6     public BST(){
7         this.root = null;
8     }
9
```

Se crea un atributo de tipo Nodo el cual será la raíz del árbol, luego el constructor de la clase para iniciar un nuevo árbol, guardando la raíz como null.

Se implementa el método insert, este invoca a insertNode dándole como parámetros x de tipo T y el nodo en el que se encuentra.

```
10 public void insert(T x){
11     this.root = insertNode(x, root);
12 }
13
14 protected Nodo<T> insertNode(T x, Nodo<T> actual){
15     Nodo<T> nodo = actual;
16
17     if(actual == null){
18         System.out.println("Dato ingresado: " + x);
19         nodo = new Nodo<T>(x);
20     } else {
21         int resC = actual.data.compareTo(x);
22         if (resC == 0){
23             System.out.println(x:"Dato duplicado");
24             return actual;
25         }
26         if (resC < 0){
27             nodo.right = insertNode(x, actual.right);
28         } else {
29             nodo.left = insertNode(x, actual.left);
30         }
31     }
32     return nodo;
33 }
```

El método insert(T x) inicia la inserción desde la raíz.
El método insertNode(T x, Nodo<T> actual) recorre el árbol:
Si el nodo es null, crea un nuevo nodo con el dato.
Si el dato ya existe (compareTo == 0), muestra un mensaje de duplicado.
Si el dato es menor, se usa la recursividad para continuar hasta llegar al nodo null e insertar.
Si es mayor, usa la recursividad hasta llegar al nodo null e insertar.

Se implementa el método destroy para borrar el árbol.

```
35     public void destroy(){
36         destroyTree(root);
37         this.root = null;
38         System.out.println(x:"Arbol destruido");
39     }
40
41     protected void destroyTree(Nodo<T> nodo){
42         this.root = null;
43     }
```

Simplemente se referencia la raíz a null, ya que de esta manera no se podrá acceder a ningún nodo.

Creamos el isEmpty y también se implementa el método remove, mismos parámetros T y Nodo.

```
45     public boolean isEmpty(){
46         return root == null;
47     }
48
49     public void remove(T x){
50         this.root = removeNode(x, root);
51     }
52
53     public Nodo<T> removeNode(T x, Nodo<T> actual){
54         Nodo<T> nodo = actual;
55         if(actual == null){System.out.println(x + " no se encuentra");}
56         int resC = actual.data.compareTo(x);
57         if(resC < 0) nodo.right = removeNode(x, actual.right);
58         else if(resC > 0) nodo.left = removeNode(x, actual.left);
59         else if (actual.left != null && actual.right != null) {
60             Nodo<T> successor = minRecover(nodo.right);
61             nodo.data = successor.data;
62             nodo.right = removeNode(successor.data, nodo.right);
63         } else {
64             nodo = (actual.left != null) ? actual.left : actual.right;
65         }
66         return nodo;
67     }
68
69     public Nodo<T> minRecover(Nodo<T> dato){
70         while (dato.left != null) {
71             dato = dato.left;
72         }
73         return dato;
74     }
```

Llama a removeNode, empezando desde la raíz, para eliminar el valor x.
Busca y elimina el nodo con el valor x.

Si el nodo es null, el valor no está en el árbol.

Compara x con el valor actual:

Si x es mayor, busca a la derecha.

Si x es menor, busca a la izquierda.

Si x es igual:

Caso 2 hijos: reemplaza el valor con el mínimo del subárbol derecho (minRecover), y luego elimina ese mínimo.

Caso 1 hijo o ninguno: reemplaza el nodo con su hijo (si tiene).

MinRecover devuelve el nodo con el **valor mínimo** del subárbol (el más a la izquierda).

Ahora se implementan los ordenamientos InOrder, PreOrder y PostOrder. Todo esto se hace recursivamente.

```
76     public void InOrder(){
77         InOrderRec(root);
78     }
79
80     public void InOrderRec(Nodo<T> nodo){
81         if(nodo != null){
82             InOrderRec(nodo.left);
83             System.out.println(nodo.data + " ");
84             InOrderRec(nodo.right);
85         }
86     }
```

```
88     public void PostOrder(){
89         PostOrderRec(root);
90     }
91
92     public void PostOrderRec(Nodo<T> nodo){
93         if(nodo != null){
94             PostOrderRec(nodo.left);
95             PostOrderRec(nodo.right);
96             System.out.println(nodo.data + " ");
97         }
98     }
99 }
```

```
101    public void PreOrder(){
102        PreOrderRec(root);
103    }
104
105    public void PreOrderRec(Nodo<T> nodo){
106        if(nodo != null){
107            System.out.println(nodo.data + " ");
108            PreOrderRec(nodo.left);
109            PreOrderRec(nodo.right);
110        }
111    }
```

Continuamos con la implementación del método search, este invoca a searchNode para luego comprobar si el nodo resultante es null.

```
113     public T search(T x){
114         Nodo<T> res = searchNode(x, root);
115         if(res == null){
116             System.out.println(x:"No se encuentra el dato");
117             return null;
118         } else {
119             System.out.println(x:"Dato encontrado");
120             return res.data;
121         }
122     }
123
124     public Nodo<T> searchNode(T x, Nodo<T> actual){
125         if(actual == null) return null;
126         else {
127             int resC = actual.data.compareTo(x);
128             if(resC < 0) return searchNode(x, actual.right);
129             else if(resC > 0) return searchNode(x, actual.left);
130             else return actual;
131         }
132     }
```

Es una búsqueda recursiva:

Si el nodo actual es null, no se encontró.

Compara el dato:

Si x es menor, busca a la izquierda.

Si x es mayor, busca a la derecha.

Si es igual, retorna el nodo.

Se continúa implementando los métodos Min y Max, los cuales imprimen el menor y mayor valor del árbol.

```
134     public void Min(){
135         System.out.println(searchMin(root));
136     }
137
138     public T searchMin(Nodo<T> nodo){
139         while (nodo.left != null) {
140             nodo = nodo.left;
141         }
142         return nodo.data;
143     }
144
145     public void Max(){
146         System.out.println(searchMax(root));
147     }
148
149     public T searchMax(Nodo<T> nodo){
150         while (nodo.right != null){
151             nodo = nodo.right;
152         }
153         return nodo.data;
154     }
```

Al ser el mínimo y máximo, se busca hasta el limite de sus ramas tanto derecha como izquierda

Finalmente, en esta clase se implementan los métodos predecesor y sucesor, los cuales son los nodos que le suceden al nodo principal.

```
156 public void Predecesor(T x){
157     Nodo<T> pre = searchNode(x, root);
158     x = pre.left.data;
159     System.out.println(x);
160 }
161
```

Busca el nodo con valor x.

Luego va a su subárbol izquierdo y busca el nodo con el mayor valor (más a la derecha).
Ese es el predecesor inmediato (el valor justo menor que x).

```
162 public void Sucesor(T x){
163     Nodo<T> sucesor = searchNode(x, root);
164     x = sucesor.right.data;
165     System.out.println(x);
166 }
167 }
```

Busca el nodo con valor x.

Luego va a su subárbol derecho y busca el nodo con el menor valor (más a la izquierda).
Ese es el sucesor inmediato (el valor justo mayor que x).

Teniendo listo nuestro BST genérico, pasamos a las pruebas en la clase principal Test

```
3 import java.util.Scanner;
4
5 public class Test {
6     Run | Debug
7     public static void main(String[] args) {
8         Scanner sc = new Scanner(System.in);
9         BST<Integer> arbol = new BST<>();
10
11         arbol.insert(x:50);
12         arbol.insert(x:30);
13         arbol.insert(x:70);
14         arbol.insert(x:20);
15         arbol.insert(x:40);
16         arbol.insert(x:60);
17         arbol.insert(x:80);
18
19         System.out.println(arbol.isEmpty() + "\n");
20     }
21 }
```

Creamos el árbol


```
18      System.out.println(arbol.isEmpty() + "\n");
19
20      int opcion;
21      do {
22          System.out.println(x:"\n--- MENU BST ---");
23          System.out.println(x:"1. Mostrar Árbol (Inorden)");
24          System.out.println(x:"2. Mostrar Árbol (Postorden)");
25          System.out.println(x:"3. Mostrar Árbol (Preorden)");
26          System.out.println(x:"4. Eliminar un nodo");
27          System.out.println(x:"5. Encontrar un nodo");
28          System.out.println(x:"6. Encontrar nodo mínimo");
29          System.out.println(x:"7. Encontrar nodo máximo");
30          System.out.println(x:"8. Encontrar el sucesor de un nodo");
31          System.out.println(x:"9. Encontrar el predecesor de un nodo");
32          System.out.println(x:"10. Destruir árbol");
33          System.out.println(x:"11. Salir");
34          System.out.print(s:"Elige una opción: ");
35          opcion = sc.nextInt();
36      }
```

Creamos el menú

```
37      switch (opcion) {
38          case 1:
39              System.out.println(x:"Recorrido Inorden: ");
40              arbol.InOrder();
41              System.out.println();
42              break;
43          case 2:
44              System.err.println(x:"Recorrido PostOrden: ");
45              arbol.PostOrder();
46              break;
47          case 3:
48              System.err.println(x:"Recorrido PreOrden: ");
49              arbol.PreOrder();
50              break;
51          case 4:
52              System.out.print(s:"Ingresa el valor a eliminar: ");
53              int valor = sc.nextInt();
54              arbol.remove(valor);
55              System.out.println(x:"Nodo eliminado (si existía).");
56              break;
57          case 5:
58              System.out.print(s:"Ingresar el valor a buscar: ");
59              int valorIng = sc.nextInt();
60              arbol.search(valorIng);
61              break;
62          case 6:
63              System.out.print(s:"El nodo mínimo es: ");
64              arbol.Min();
65              break;
66          case 7:
67              System.out.print(s:"El nodo máximo es: ");
68              arbol.Max();
69              break;
70          case 8:
71              System.out.print(s:"Ingresa el nodo: ");
72              int sucess = sc.nextInt();
73              arbol.Sucesor(sucess);
74              break;
75      }
```

```

75         case 9:
76             System.out.print(s:"Ingresa el nodo: ");
77             int predecesor = sc.nextInt();
78             arbol.Predecesor(predecesor);
79             break;
80         case 10:
81             arbol.destroy();
82             break;
83         case 11:
84             System.out.println(x:"Saliendo...");
85             break;
86         default:
87             System.out.println(x:"Opción inválida.");
88     }
89     } while (opcion != 11);
90
91     sc.close();
92 }
93 }
94

```

Creamos el switch case para cada opción a usar

Resultado en consola:

```

Dato ingresado: 50
Dato ingresado: 30
Dato ingresado: 70
Dato ingresado: 20
Dato ingresado: 40
Dato ingresado: 60
Dato ingresado: 80
false

```

```

--- MENU BST ---
1. Mostrar Árbol (Inorden)
2. Mostrar Árbol (Postorden)
3. Mostrar Árbol (Preorden)
4. Eliminar un nodo
5. Encontrar un nodo
6. Encontrar nodo mínimo
7. Encontrar nodo máximo
8. Encontrar el sucesor de un nodo
9. Encontrar el predecesor de un nodo
10. Destruir árbol
11. Salir
Elige una opción: 

```

```

Elige una opción: 1
Recorrido Inorden:
20
30
40
50
60
70
80

```

```

Elige una opción: 2
Recorrido PostOrden:
20
40
30
60
80
70
50

```

```

Elige una opción: 3
Recorrido PreOrden:
50
30
20
40
70
60
80

```

```
Elige una opción: 4
Ingresa el valor a eliminar: 70
Nodo eliminado (si existía).
```

```
Elige una opción: 1
Recorrido Inorden:
20
30
40
50
60
80
```

```
Elige una opción: 5
Ingresar el valor a buscar: 10
No se encuentra el dato
```

```
Elige una opción: 5
Ingresar el valor a buscar: 30
Dato encontrado
```

```
Elige una opción: 6
El nodo mínimo es: 20
```

```
Elige una opción: 7
El nodo máximo es: 80
```

```
Elige una opción: 8
Ingresa el nodo: 30
Sucesor: 40
```

```
Elige una opción: 9
Ingresa el nodo: 30
Predecesor: 20
```

```
Elige una opción: 1
Recorrido Inorden:
20
30
40
50
60
80
```

```
Elige una opción: 10
Arbol destruido
```

2. **Ejercicio 2:** Implementar un árbol binario de búsqueda ABB o BST donde ingrese por teclado una palabra y como resultado debe mostrar el árbol de acuerdo al valor decimal correspondiente de su código ASCII. Utilizando los métodos del ejercicio anterior.

Para la clase TestASCII se usan los métodos usados en la clase BST, los cuales funcionan correctamente.

```

1  package ProblemasPropuestos.Ejercicio2;
2  import ProblemasPropuestos.Ejercicio1.BST;
3  import java.util.Scanner;
4
5  public class TestASCII {
6      Run | Debug
7      public static void main(String[] args) {
8          Scanner sc = new Scanner(System.in);
9          BST<Character> let = new BST<>();
10
11          System.out.print(s:"Ingresar Palabra: ");
12          String palabra = sc.nextLine();
13
14          for(int i = 0; i < palabra.length(); i++){
15              char p = palabra.charAt(i);
16              let.insert(p);
17          }
18
19          System.out.println(let.isEmpty() + "\n");

```

De esta manera ingresamos un texto String y se obtienen sus caracteres.

Se almacenan sus caracteres como char y se ingresan al árbol.

No se coloca más del código ya que el menú es igual que el test anterior.

Resultados en consola:

```

Ingresar Palabra: cavernicola
Dato ingresado: c
Dato ingresado: a
Dato ingresado: v
Dato ingresado: e
Dato ingresado: r
Dato ingresado: n
Dato ingresado: i
Dato duplicado
Dato ingresado: o
Dato ingresado: l
Dato duplicado
false

```

```

--- MENU BST ---
1. Mostrar Árbol (Inorden)
2. Mostrar Árbol (Postorden)
3. Mostrar Árbol (Preorden)
4. Eliminar un nodo
5. Encontrar un nodo
6. Encontrar nodo mínimo
7. Encontrar nodo máximo
8. Encontrar el sucesor de un nodo
8. Encontrar el predecesor de un nodo
9. Destruir árbol
10. Salir
Elige una opción: █

```

Elige una opción: 1
Recorrido Inorden:

a
c
e
i
l
n
o
r
v

Elige una opción: 2
Recorrido PostOrden:

a
l
i
o
n
r
e
v
c

Elige una opción: 3
Recorrido PreOrden:

c
a
v
e
r
n
i
l
o

Recorrido Inorden:

a
e
i
l
n
o
r
v

Elige una opción: 4
Ingresa el valor a eliminar: c
Nodo eliminado (si existía).

Elige una opción: 5
Ingresar el valor a buscar: o
Dato encontrado

Elige una opción: 5
Ingresar el valor a buscar: z
No se encuentra el dato

Elige una opción: 6
El nodo mínimo es: a

Elige una opción: 7
El nodo máximo es: v

Elige una opción: 8
Ingresa el nodo: e
Sucesor: i

Elige una opción: 9
Ingresa el nodo: e
Predecesor: a

- 3. Ejercicio 3:** Implementar un método para graficar el árbol binario de búsqueda ABB o BST resultante, mostrando todos sus nodos, sus aristas izquierda y derecha utilizando clases y métodos genéricos, utilizar la librería **Graph Stream**.

Para este ejercicio se inicia un proyecto java y se importa la librería Graph Stream. Para poder usarla y crear nuestro árbol creamos la clase VisualBST. SOLO se hará la inserción de datos y mostrar sus nodos con la librería.

```

1 package ProblemasPropuestos.Ejercicio3.src;
2
3 import org.graphstream.graph.*;
4 import org.graphstream.graph.implementations.*;
5
6 public class VisualBST<T extends Comparable<T>> {
7     private Nodo<T> root;
8
9     public void insert(T value) {
10         root = insertRec(root, value);
11     }
12
13     private Nodo<T> insertRec(Nodo<T> nodo, T value) {
14         if (nodo == null) {
15             return new Nodo<>(value);
16         }
17         int cmp = value.compareTo(nodo.data);
18         if (cmp < 0) nodo.left = insertRec(nodo.left, value);
19         else if (cmp > 0) nodo.right = insertRec(nodo.right, value);
20         return nodo;
21     }

```

Método que ya conocemos (Se está usando la clase Nodo)

```

23 public void draw() {
24     System.setProperty(key:"org.graphstream.ui", value:"swing");
25     Graph graph = new SingleGraph("BST");
26     graph.setAttribute("ui.stylesheet", "node { fill-color: skyblue; size: 30px; text-size: 16px; }");
27     addNodesEdges(graph, root, parent:null);
28     graph.display();
29 }

```

Usa GraphStream para visualizar gráficamente el árbol binario de búsqueda (BST).

Crea un grafo llamado "BST" con nodos de color celeste y etiquetas visibles.

Llama al método recursivo addNodesEdges para añadir los nodos y conexiones.

Muestra el grafo en una ventana (graph.display()).

```

31 private void addNodesEdges(Graph graph, Nodo<T> nodo, T parent) {
32     if (nodo == null) return;
33
34     String id = nodo.data.toString();
35     graph.addNode(id).setAttribute("ui.label", id);
36
37     if (parent != null) {
38         String parentId = parent.toString();
39         graph.addEdge(parentId + "-" + id, parentId, id, true);
40     }
41
42     addNodesEdges(graph, nodo.left, nodo.data);
43     addNodesEdges(graph, nodo.right, nodo.data);
44 }

```

Recorre el árbol recursivamente.

Por cada nodo:

Lo agrega al grafo (graph.addNode).

Si tiene un padre (parent), añade una arista (edge) entre el padre y el hijo.

Repite lo mismo para el hijo izquierdo y derecho.

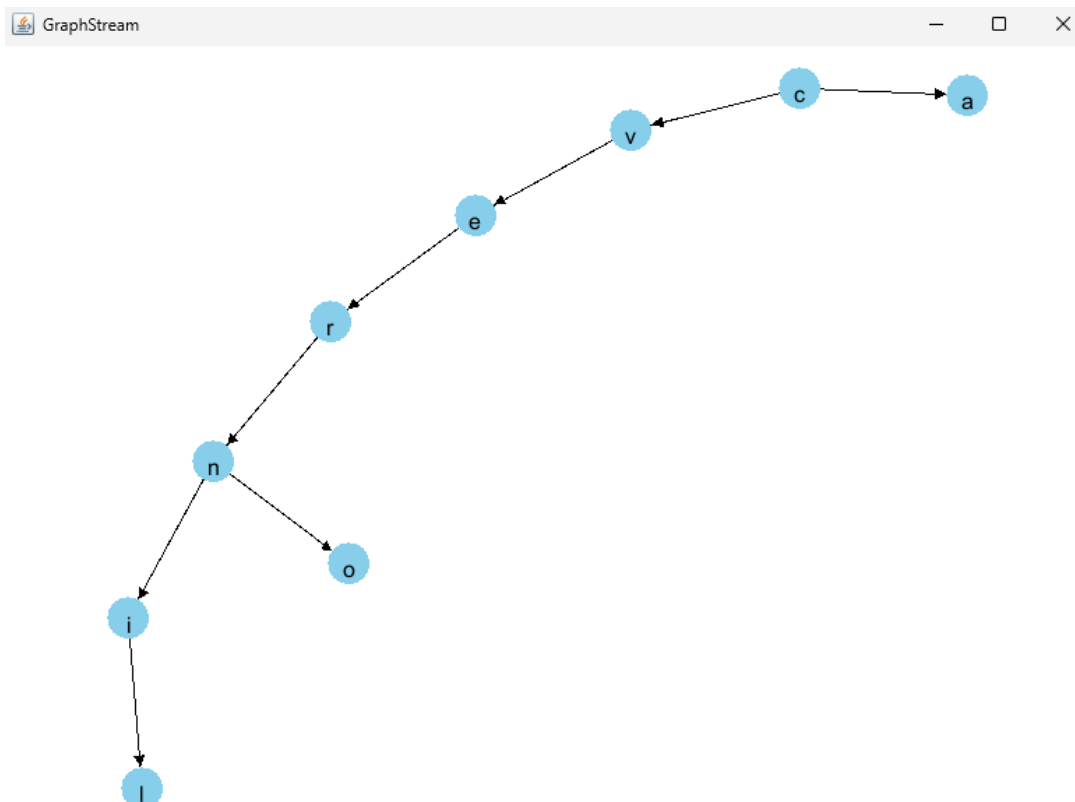
Para probar lo anterior, se crea la clase principal TestGraficar.



```

1  package ProblemasPropuestos.Ejercicio3.src;
2
3  import java.util.Scanner;
4
5  public class TestGraficar {
6      Run | Debug
7      public static void main(String[] args) throws Exception {
8          Scanner sc = new Scanner(System.in);
9          VisualBST<Character> bst = new VisualBST<>();
10
11          System.out.print(s:"Ingresa una palabra: ");
12          String palabra = sc.nextLine();
13
14          for (int i = 0; i < palabra.length(); i++) {
15              bst.insert(palabra.charAt(i));
16          }
17
18          bst.draw();
19          sc.close();
20      }
21  }

```

Ingresa una palabra: cavernicola



	<p align="center">UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		
Aprobación: 2022/03/01	Código: GUIA-PRLD-001	Página: 16

IV. CUESTIONARIO

- ¿Cuáles fueron las dificultades que encontraste al desarrollar los ejercicios propuestos? por ejemplo, poca documentación, complejidad del lenguaje, etc.
 En el último ejercicio, la documentación era algo complicada de entender y por el tiempo también se complicó pero nada que no se pueda resolver. En los anteriores problemas los algoritmos de búsqueda y eliminación fueron los fundamentales para poder crear todos los demás métodos solicitados.
- ¿Explique cómo es el algoritmo que implemento para obtener el árbol binario de búsqueda con la librería Graph Stream? Recuerde que puede agregar operaciones sobre la clase BST.
 El algoritmo para visualizar el árbol con GraphStream sigue estos pasos:
 - Se crea un grafo (SingleGraph) y se le aplica un estilo visual para los nodos.
 - Se utiliza un método recursivo llamado addNodesEdges que:
 - Agrega cada nodo del árbol al grafo.
 - Si el nodo tiene un padre, se crea una arista dirigida desde el padre hacia el hijo.
 - Luego se llama recursivamente al hijo izquierdo y derecho.
 - Finalmente, se llama a graph.display() para mostrar la visualización gráfica del árbol.
 Este algoritmo respeta la estructura del árbol binario de búsqueda y la traduce visualmente usando nodos (valores) y aristas (relaciones padre-hijo).

V. REFERENCIAS Y BIBLIOGRAFÍA RECOMENDADAS:

- Weiss M., Data Structures & Problem Solving Using Java, 2010, Addison-Wesley.
- Weiss M., Data Structures and Algorithms Analysis in Java, 2012, Addison-Wesley.
- Cormen T., Leiserson C., Rivest R., Stein C., Introduction to Algorithms, 2022, The MIT Press
- The Java™ Tutorials - <https://docs.oracle.com/javase/tutorial/>
- Sedgewick, R., Algorithms in Java, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching, Part 5: Graph Algorithms, Addison-Wesley.
- Malik D., Data Structures Using C++, 2003, Thomson Learning.
- Knuth D., The Art of Computer Programming, Vol. 1 y 3, Addison - Wesley.

TÉCNICAS E INSTRUMENTOS DE EVALUACIÓN

TÉCNICAS:

Actividades Resueltas
Ejercicios Propuestos

INSTRUMENTOS:

Rubricas

CRITERIOS DE EVALUACIÓN

Los criterios de evaluación se encuentran en el sílabo DUFA ANEXO en la sección EVOLUCIÓN CONTINUA