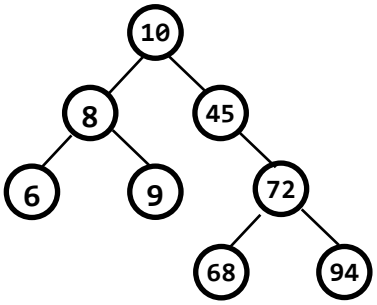
Recuperación Segundo Parcial de Estructuras de Datos y Algoritmos (EDA) – 25 de Junio de 2021

Duración: 1h. 30 m.

APELLIDOS, NOMBRE	GRUPO

1.- ***(3 puntos)*** Dado un árbol binario de búsqueda (de la clase **ABB**), se quiere devolver un **`String`** que contenga los caminos desde la raíz hasta las hojas que sean mayores que un valor determinado, **e**, dado como argumento.



Para el árbol del ejemplo y **e** = 21, se debe obtener el siguiente **`String`**:

```
10 45 72 68
10 45 72 94
```

Para el árbol del ejemplo y **e** = 6, se debe obtener el siguiente **`String`**:

```
10 8 9
10 45 72 68
10 45 72 94
```

Implementa el método de clase recursivo invocado en la siguiente lanzadera para que, del modo más eficiente, devuelva dicho **`String`.**

```
public String caminosAHojas(E e) {
    return caminosAHojas(raiz, e, "");
}
```

```
private String caminosAHojas(NodoABB<E> nodo, E e, String camino) {
    if (nodo == null) { return ""; }
    camino += nodo.dato + " ";
    if (nodo.dato.compareTo(e) > 0) { // El nodo es mayor que e
        if (nodo.izq == null && nodo.der == null) { return camino + "\n"; }
        return caminosAHojas(nodo.izq, e, camino) +
               caminosAHojas(nodo.der, e, camino);
    } else { // El nodo es menor o igual que e
        return caminosAHojas(nodo.der, e, camino);
    }
}
```

2.- ***(3 puntos)*** Implementa un método estático tal que, dados **a** y **b**, dos elementos de tipo genérico, y **cP**, una **ColaPrioridad** del mismo tipo, cambie todos los elementos de **cP** que coincidan con **a** por el elemento **b**, de la manera más eficiente posible. (Los demás elementos de **cP** deben quedar inalterados al final de la ejecución del método). Para ello, el método deberá usar una pila como estructura de datos auxiliar.

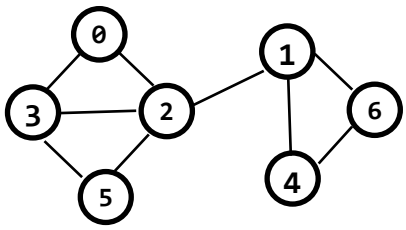
Usando la pila de la recursión:

```
public static <E extends Comparable<E>> void cambiar(ColaPrioridad<E> cP, E a, E b)
{
    if (!cP.esVacia()) {
        E min = cP.eliminarMin();
        int cmp = min.compareTo(a);
        if (cmp <= 0) { cambiar(cP, a, b); }
        if (cmp == 0) { cP.insertar(b); }
        else { cP.insertar(min); }
    }
}
```

Usando una pila auxiliar, tipo ArrayPila:

```
public static <E extends Comparable<E>> void cambiar(ColaPrioridad<E> cP, E a, E b)
{
        Pila<E> aux = new ArrayPila<E>();
        while (!cP.esVacia() && cP.recuperarMin().compareTo(a) <= 0) {
            E min = cP.eliminarMin();
            int cmp = min.compareTo(a);
            if (cmp == 0) { aux.apilar(b); }
            else { aux.apilar(min); }
        }
        while (!aux.esVacia()) { cP.insertar(aux.desapilar()); }
}
```

3.- ***(4 puntos)*** Una arista puente es una arista tal que si se elimina de un grafo se incrementa el número de componentes conexas. Un grafo conexo deja de serlo si se elimina una arista puente (en el grafo de la figura, la arista **(1, 2)** es un puente).



Se pide implementar en la clase **Grafo** un método **esAristaPuente** que, con el menor coste posible, compruebe si una arista **(i, j)** es una arista puente.

Ayuda. Una arista **(i, j)** es puente si no existe camino para llegar de **i** a **j** sin incluir dicha arista.

```
public boolean esAristaPuente(int i, int j) {
   visitados = new int[numVertices()];
   DFSsinIJ(i, i, j);
   return visitados[j] == 0;
}

protected void DFSsinIJ(int v, int i, int j) {
   visitados[v] = 1;
   ListaConPI<Adyacente> l = adyacentesDe(v);
   for (l.inicio(); !l.esFin(); l.siguiente()) {
     int w = l.recuperar().getDestino();
     if (visitados[w] == 0)
        if (!(v == i && w == j)) //si (v, w) NO es la arista (i, j) que se busca ...
          if (w == j) visitados[w] = 1; //si se alcanza j, (i, j) NO puede ser un puente
          else        DFSsinIJ(w, i, j); //sino, continúa la búsqueda
   }
}

// Una versión menos eficiente del DFS de un vértice, aunque también válida, sería:
protected void DFSsinIJ(int v, int i, int j) {
   visitados[v] = 1;
   ListaConPI<Adyacente> l = adyacentesDe(v);
   for (l.inicio(); !l.esFin(); l.siguiente()) {
      int w = l.recuperar().getDestino();
      if (visitados[w] == 0 && !(v == i && w == j)) { DFSsinIJ(w, i, j); }
   }
}
```

ANEXO

Las interfaces ListaConPI, ColaPrioridad y Pila del paquete modelos.

```
public interface ListaConPI<E> {
  void insertar(E e);
  void eliminar();
  void inicio();
  void siguiente();
  void fin();
  E recuperar();
  boolean esFin();
  boolean esVacia();
  int talla();
}
```

```
public interface ColaPrioridad<E extends
Comparable<E>> {
  void insertar(E e);
  /** SII !esVacia() */ E eliminarMin();
  /** SII !esVacia() */ E recuperarMin();
  boolean esVacia();
}
public interface Pila<E> {
  void apilar(E e);
  E desapilar();
  E tope();
  boolean esVacia();
}
```

Las clases NodoABB y ABB del paquete jerarquicos.

```
class NodoABB<E> {
  protected E dato;
  protected NodoABB<E> izq, der;
  protected int talla;
  public NodoABB(E dato) { … }
}
```

```
public class
ABB<E extends Comparable<E>> {
  protected NodoABB<E> raiz;
  public ABB() { … }
  …
}
```

Las clases Grafo y Adyacente del paquete grafos.

```
public abstract class Grafo {
  protected int[] visitados;
  protected int ordenVisita;
  protected Cola<Integer> q;
  ...
  public abstract int numVertices();
  public abstract int numAristas();
  public abstract ListaConPI<Adyacente> adyacentesDe(int i);
  ...
}
public class Adyacente {
  protected int destino;
  protected double peso;
  public Adyacente(int d, double p) {…}
  public int getDestino() { … }
  public double getPeso() { … }
}
```