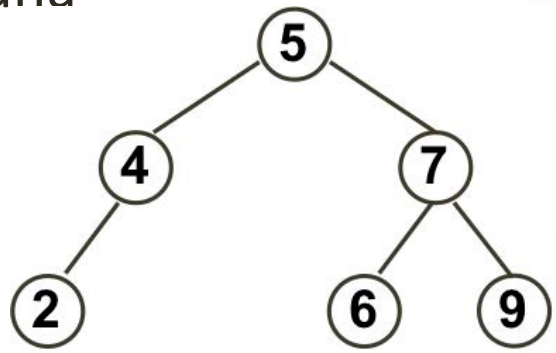
Ejercicio 1

Sea la clase ***ABBInteger*** una derivada de ***ABB***, cuyos elementos son todos enteros positivos. Diseña en ella un método *caminoQueSuma* que, con el menor coste posible y recursivamente, compruebe si existe una secuencia de nodos (camino) desde su raíz a uno de sus descendientes cuyos datos sumen ***s***, con ***s*** > 0.

Así, por ejemplo, dado el ABB de la derecha, el método devuelve ***true*** si el valor de ***s*** es 9, 11 o 12; sin embargo, devuelve ***false*** si el valor de ***s*** es 10 o 19.




```
public class ABBDeInteger extends ABB<Integer> {
  …
}
```

Solución

```
public boolean caminoQueSuma(int s) {
  return caminoQueSuma(s, raiz);
}

private boolean caminoQueSuma(int s, NodoABB<Integer> actual) {
  if (actual == null) return false;
  s -= actual.dato;
  if (s == 0) return true; // Encontrado
  if (s < 0) return false; // La suma del camino supera la buscada
  if (s <= actual.dato) return caminoQueSuma(s, actual.izq);
  return caminoQueSuma(s, actual.izq) ||
         caminoQueSuma(s, actual.der);
}
```

Ejercicio 2

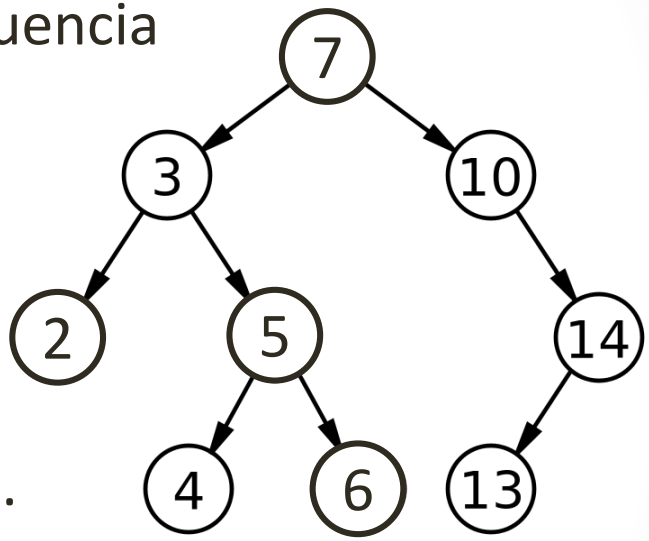
Escribe una función en la clase ABB que devuelva una Lista con PI con todos los elementos del árbol, ordenados de menor a mayor, que sean menores que uno dado.

Solución

```
public ListaConPI<E> menoresQueEnOrden(E x) {
  ListaConPI<E> res = new LEGListaConPI<E>();
  menoresQueEnOrden(raiz, x, res);
  return res;
}

private void menoresQueEnOrden(NodoABB<E> actual, E x,
                               ListaConPI<E> res) {
  if (actual != null) {
    menoresQueEnOrden(actual.izq, x, res);
    if (actual.dato.compareTo(x) < 0) {
      res.insertar(actual.dato);
      menoresQueEnOrden(actual.der, x, res);
    }
  }
}
```

Ejercicio 3

Escribe una función en la clase ABBInteger, que extiende de un ABB<Integer>, que devuelva el primer valor que falta en el árbol (suponiendo que la secuencia de valores empieza por el valor mínimo del árbol). En el árbol de la derecha, por ejemplo, el primer elemento que falta en la secuencia es el 8. La función devolverá ***null*** si el árbol está vacío.



Para que el coste en el peor caso sea lineal respecto al número de nodos, se podrá utilizar un Pila como estructura auxiliar donde almacenar los nodos conforme se van visitando.

Solución

```
public Integer faltaEnSecuencia() {
  Pila<Integer> pila = new ArrayPila<>();
  faltaEnSecuencia(raiz, pila);
  if (pila.esVacia()) return null;
  else return pila.tope() + 1;
}

private void faltaEnSecuencia(NodoABB<Integer> actual,
                              Pila<Integer> pila) {
  if (actual != null) {
    faltaEnSecuencia(actual.izq, pila);
    if (pila.esVacia() || { // Primer nodo, el mínimo
        pila.apilar(actual.dato);
        faltaEnSecuencia(actual.der, pila);
    } else if (actual.dato == pila.tope() + 1) { // Secuencia correcta
        pila.desapilar();          // Opcional, para consumir menos memoria
        pila.apilar(actual.dato);
        faltaEnSecuencia(actual.der, pila);
    }
  }
}
```

Ejercicio 4

En la clase ABB, se pide implementar un método público *contarEnRango* que, con el menor coste posible, devuelva el número de elementos del ABB comprendidos en el intervalo `[eI, eS]`, donde `eI` y `eS` son dos elementos dados de tipo genérico `E` tales que `eI` es menor o igual que `eS`.

Solución

```
public int contarEnRango(E eI, E eS) {
  return contarEnRango(eI, eS, raiz);
}

private int contarEnRango(E eI, E eS, NodoABB<E> actual) {
  if (actual == null) return 0;
  if (actual.dato.compareTo(eI) < 0)
    return contarEnRango(eI, eS, actual.der);
  if (actual.dato.compareTo(eS) > 0)
    return contarEnRango(eI, eS, actual.izq);
  return 1 + contarEnRango(eI, eS, actual.izq) +
         contarEnRango(eI, eS, actual.der);
}
```

Ejercicio 5

La sucesión de Fibonacci es una sucesión de números naturales que comienza con los números 0 y 1 y, a partir de estos, cada término es la suma de los dos anteriores:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, …

Diseña una función que compruebe si los elementos que contiene una Cola de Prioridad dada de valores Integer forman la secuencia de Fibonacci. Esta función debe dejar la Cola de Prioridad con los mismos elementos que contenía antes de la llamada.

Solución

```
public static boolean esFibonacci(ColaPrioridad<Integer> cp) {
    boolean res = true;
    if (!cp.esVacia()) {
        Integer n1 = cp.eliminarMin();
        if (n1 != 0) res = false;
        else if (!cp.esVacia()) {
            Integer n2 = cp.eliminarMin();
            if (n2 != 1) res = false;
            else res = esFibonacci(cp, n1, n2);
            cp.insertar(n2);
        }
        cp.insertar(n1);
    }
    return res;
}
private static boolean esFibonacci(ColaPrioridad<Integer> cp, int n1, int n2) {
    boolean res = true;
    if (!cp.esVacia()) {
        Integer n3 = cp.eliminarMin();
        if (n3 == n1 + n2) res = esFibonacci(cp, n2, n3);
        else res = false;
        cp.insertar(n3);
    }
    return res;
}
```

Ejercicio 6

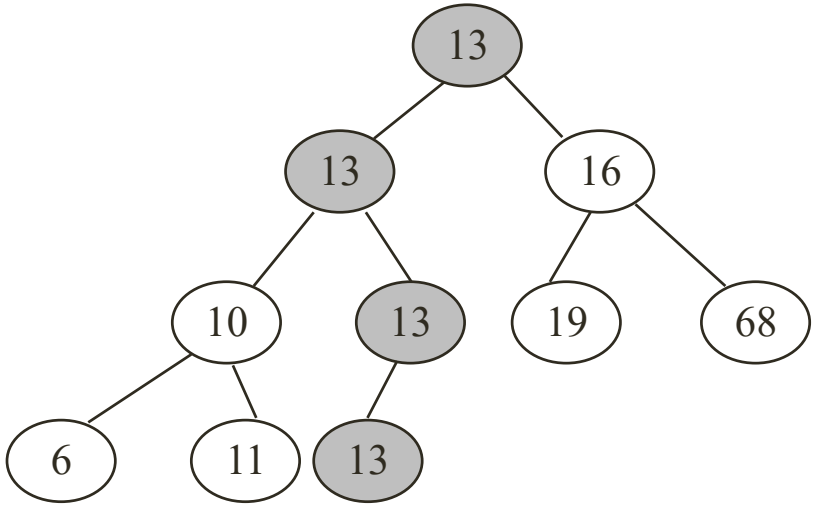
Diseña una función que, dado un array genérico de objetos comparables en el que todas sus posiciones están ocupadas, compruebe si dicho array cumple la propiedad de orden de un Montículo Minimal con raíz en la posición cero.

Solución

```
public static <E extends Comparable<E>>
  boolean cumplePO(E v[]) {
  // Comprobamos la propiedad de orden de cada elemento con su padre
  for (int i = 1; i < v.length; i++) {
    if (v[(i-1)/2].compareTo(v[i]) > 0)
       return false;
  }
  return true;
}
```

Ejercicio 7

Diseña una función en la clase MonticuloBinario que compruebe si hay un camino de la raíz a una de las hojas en el que todos nodos tengan el mismo valor.
En el siguiente Montículo, por ejemplo, sí que existiría un camino con todos sus elementos repetidos:

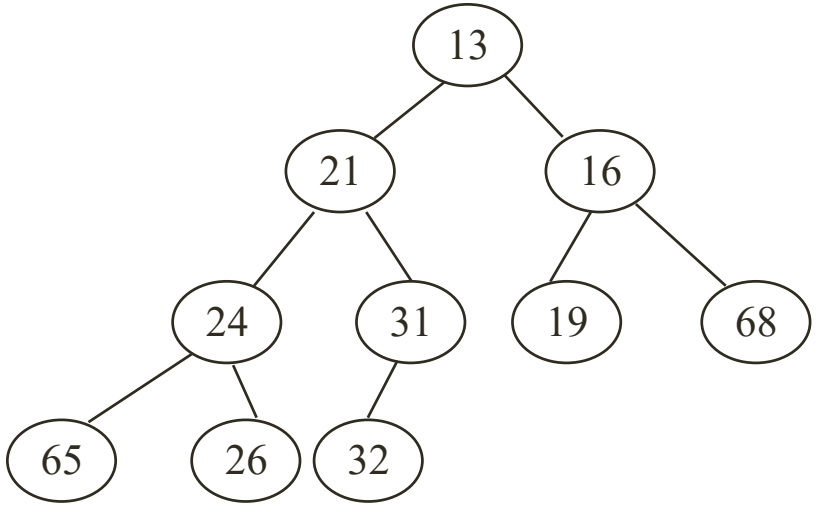


Solución

```
public boolean caminoDeRepetidos() {
  return caminoDeRepetidos(1);
}

private boolean caminoDeRepetidos(int pos) {
  if (pos > talla) return true;
  if (elArray[1].compareTo(elArray[pos]) != 0) return false;
  return caminoDeRepetidos(pos * 2) ||
         caminoDeRepetidos(pos * 2 + 1);
}
```

Ejercicio 8

Sea MonticuloBinarioInteger un montículo binario minimal con elementos de tipo Integer. Diseña en esta clase un método que elimine del montículo todas las hojas que sean pares.
Por ejemplo, en el siguiente montículo se eliminarán los nodos 68, 26 y 32:



Solución

```
public void eliminarHojasPares() {
  int i = talla / 2 + 1;  // Primera hoja
  while (i <= talla) {    // Recorremos las hojas
    if (elArray[i] % 2 == 0) { // Hoja par
      elArray[i] = elArray[talla]; // Copiamos el último en
                                   // la posición i
      talla--;                     // Borramos el último
      reflotar(i);     // Reflotamos por si hemos estropeado
                       // la propiedad de orden
    } else i++;
  }
}
```

Ejercicio 9

Diseñad en la clase Grafo un método *enCiclo* que compruebe si un vértice ***v*** dado forma parte de un ciclo en el grafo.

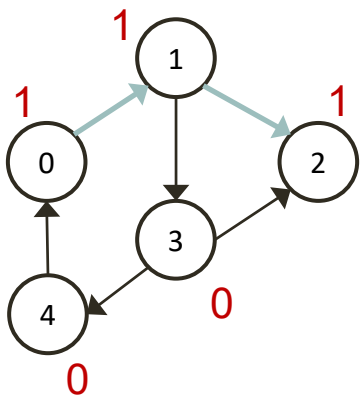
Solución

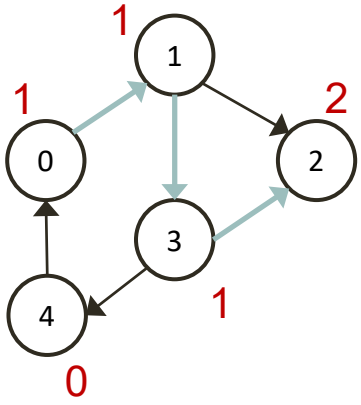
```
public boolean enCiclo(int v) {
  visitados = new int[numVertices()];
  return enCicloDFS(v, v);
}

protected boolean enCiclo(int actual, int v) {
  visitados[actual] = 1;
  ListaConPI<Adyacente> l = adyacentesDe(actual);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    int ady = l.recuperar().destino;
    if (ady == v ||                    // Llegamos a v -> ciclo
        (visitados[ady] == 0 && enCiclo(ady, v))
       return true;
  }
  return false;
}
```

Ejercicio 10

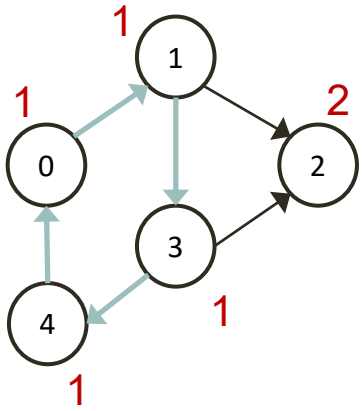
Diseña un método en la clase *Grafo* que compruebe si el grafo tiene ciclos.

Para ello no basta con saber si un vértice ha sido visitado o no, necesitamos tres posibles estados: **0** (no visitado), **1** (visitado en el camino actual), **2** (visitado por otro camino).





No es un ciclo, el vértice 2 fue visitado por otro camino



Ciclo detectado: una arista vuelve a un vértice visitado en el camino actual

Solución

```
public boolean tieneCiclos() {
  visitados = new int[numVertices()];
  for (int v = 0; v < numVertices(); v++)
    if (visitados[v] == 0 && tieneCiclos(v)) return true;
  return false;
}

protected boolean tieneCiclos(int v) {
  visitados[v] = 1;                    // En el camino actual
  ListaConPI<Adyacente> l = adyacentesDe(v);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    int w = l.recuperar().destino;
    if (visitados[w] == 1 ||                        // Ciclo
        (visitados[w] == 0 && tieneCiclos(w))
       return true;
  }
  visitados[v] = 2;              // Visitado por otro camino
  return false;
}
```

Ejercicio 11

Un grafo transpuesto *T* de un grafo *G* tiene el mismo conjunto de vértices pero con las direcciones de las aristas en sentido contrario, es decir, que una arista (u, v) en *G* se corresponde con una arista (v, u) en *T*.

Diseña un método en la clase *GrafoDirigido* que permita obtener su grafo transpuesto:

public GrafoDirigido grafoTranspuesto();

Solución

```
public GrafoDirigido grafoTranspuesto() {
  GrafoDirigido res = new GrafoDirigido(numVertices());
  for (int i = 0; i < numVertices(); i++) {
    ListaConPI<Adyacente> ady = adyacentesDe(i);
    for (ady.inicio(); !ady.esFin(); ady.siguiente()) {
      Adyacente a = ady.recuperar();
      res.insertarArista(a.destino, i, a.peso);
    }
  }
  return res;
}
```

Ejercicio 12

Diseña un método en la clase GrafoNoDirigido que devuelva el número de componentes conexas que tiene el grafo.

Solución

```
public int numeroComponentesConexas() {
  visitados = new int[numVertices()];
  int res = 0;
  for (int v = 0; v < numVertices(); v++) {
    if (visitados[v] == 0) {
      res++;
      recorridoDFS(v);
    }
  }
  return res;
}

protected void recorridoDFS(int v) {
  visitados[v] = 1;
  ListaConPI<Adyacente> l = adyacentesDe(v);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    int w = l.recuperar().destino;
    if (visitados[w] == 0) recorridoDFS(w);
  }
}
```

Ejercicio 13

Un vértice ***v*** es una raíz de un grafo acíclico si cada uno de los vértices del grafo es alcanzable desde ***v***.

Diseña una función en la clase *Grafo* que devuelva el primer vértice raíz de un grafo (o -1 si no existe tal vértice).

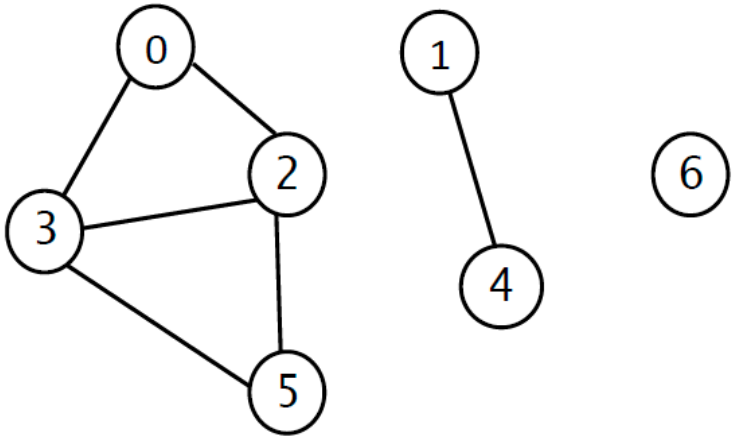
Solución

```
public int primeraRaiz() {
  for (int v = 0; v < numVertices(); v++) {
    visitados = new int[numVertices()];
    ordenVisita = 0;
    primeraRaiz(v);
    if (ordenVisita == numVertices()) return v;
  }
  return -1;
}

protected void primeraRaiz(int v) {
  visitados[v] = 1;
  ordenVisita++;
  ListaConPI<Adyacente> l = adyacentesDe(v);
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    int w = l.recuperar().destino;
    if (visitados[w] == 0) primeraRaiz(w);
  }
}
```

Ejercicio 14

En la clase ***Grafo***, implementa una función que, con el menor coste posible, devuelva el número máximo de aristas en las Componentes Conexas del grafo no dirigido.

Así, por ejemplo, para el siguiente grafo de tres componentes conexas, el método devolverá 5: en la componente de cuatro vértices hay 5 aristas, en la de dos 1 y en la de uno 0.



Solución

```
public int maxAristasEnCC() {
  int res = 0;
  visitados = new int[numVertices()];
  for (int v = 0; v < numVertices(); v++) {
    if (visitados[v] == 0) {
      int aristasEnCC = maxAristasEnCC(v);
      res = Math.max(res, aristasEnCC / 2);
    }
  }
  return res;
}
protected int maxAristasEnCC(int v) {
  visitados[v] = 1;
  ListaConPI<Adyacente> l = adyacentesDe(v);
  int res = l.talla();
  for (l.inicio(); !l.esFin(); l.siguiente()) {
    int w = l.recuperar().destino;
    if (visitados[w] == 0) res += maxAristasEnCC(w);
  }
  return res;
}
```