Pregunta 1 *[1 + 0,5 ptos]*

```
public static int fib(int n) {
    int term = 1;
    int previous = 0;
    if (n == 0) {
        return previous;
    }
    if (n == 1) {
        return term;
    }
    int k = 2;
    while (k <= n) {
        term += previous;
        previous = term - previous;
        k++;
    }
    return term;
}
```

La función fib adjunta permite obtener el término enésimo de la sucesión de Fibonacci. Esta función se puede utilizar para dar un algoritmo que proporcione los n primeros términos de la sucesión, pero resultaría ineficiente. Se pide:

1. Proporcionar una clase, de nombre $Fibonacci$, que implemente la interfaz $Iterable<E>$. Una instancia de esta clase será un objeto iterable que permite obtener, uno a uno, los *n* primeros términos de la sucesión de Fibonacci. El número de términos deberá pasarse como argumento al constructor.
2. Utiliza la clase del apartado previo en la definición de la siguiente función:

```
/**
 * Muestra en consola los n primeros términos
 * de la sucesión de Fibonacci.
 * @param n número de términos a mostrar
 */
public static void print(int n);
```

Pregunta 2 *[2 ptos]*

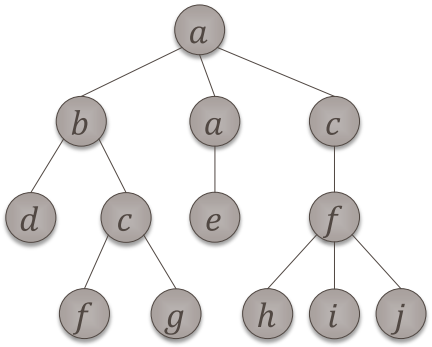
$C<E>$ es un tipo de dato que se define parcialmente como se adjunta. Sabiendo que sus instancias no podrán contener elementos nulos, ni elementos repetidos, se pide:

1. Implementar el constructor de conversión supuesto que **sus instancias son mutables**. Adicionalmente, deberá implementarse la operación add de la clase.
2. Implementar el constructor de conversión supuesto que **sus instancias son inmutables**.

Nota: en ambos casos se supondrá que la clase carece de constructor por defecto.

```
public class C<E>
        extends AbstractCollection<E> {
    List<E> data;  // área de datos
    // constructor de conversión
    public C(Collection <? extends E > c)
    { ... }
    ...
}
```

Pregunta 3 *[1,5 ptos]*



Dado el *árbol ordenado* adjunto, se pide:

1. Proporcionar la secuencia de etiquetas que se obtiene al recorrer éste en **postorden**.
2. Dar su representación mediante un árbol binario (representación $LCRS$) y la secuencia de etiquetas que se obtiene al recorrer su representación en **inorden.**
3. Dar el *árbol binario de búsqueda* que se obtendría al añadir a un árbol vacío las etiquetas de la secuencia solicitada en el primer apartado (éstas se insertarán, una a una, según su orden en la secuencia)

Pregunta 4 *[0,5+1+1 ptos]*

a) ¿Qué tipos de datos implementan las clases de JAVA ***HashSet<E> y HashMap<K, V>*** ?

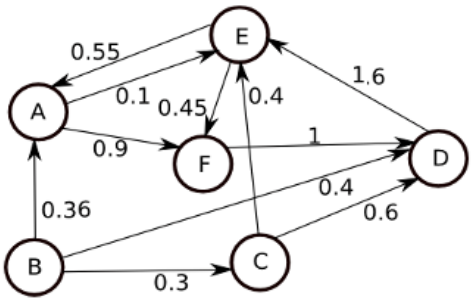
b) ¿Qué tipo de dato utilizan internamente para guardar la información? Descripción, características relevantes y coste de las operaciones básicas (inserción, borrado y pertenencia).

c) Dibujar el área de datos interna para cada uno de los dos programas adjuntos (el valor retornado por *hashCode* para *Integer* es el propio valor del entero)

```
HashSet<Integer> t1=new HashSet(11);
t1.add(25); t1.add(3);
t1.add(10); t1.add(21);
t1.add(11); t1.add(10);
t1.add(14);
```

```
HashMap<Integer,String> t2 =
    new HashMap(11);
t2.put(25,"Luis"); t2.put(3,"Pedro");
t2.put(10,"Juan"); t2.put(21,"Marta");
t2.put(10,"Ana"); t2.put(11,"Andrea");
t2.put(14,"Jorge");
```

Pregunta 5 *[0,5+1+1 ptos]*



a) Dado el grafo dirigido y etiquetado adjunto, dar su representación como **matriz** y como **lista** de adyacencias.

b) ¿Es posible usar **HashSet/HashMap** para soportar las representaciones pedidas en el apartado anterior (**matriz/lista** de adyacencias)? Dar y explicar la respuesta para las cuatro combinaciones posibles. En los casos afirmativos (si los hay), escribir el área de datos propuesta.

c) Dibujar el bosque de extensión en anchura del grafo de ejemplo explicando claramente los pasos seguidos (sin explicación no se puntúa). Se usará el orden alfabético para escoger los nodos del grafo mientras que para escoger los adyacentes se usará el criterio del mínimo coste del arco correspondiente.

Anexo

Interfaz *Iterable<E>*

default void	forEach (Consumer<? super E> action) Realiza la acción dada para cada elemento de la colección iterable hasta que todos se procesan o la acción lance una excepción.	
Iterator<E>	iterator()	Retorna un iterador sobre los elementos de tipo E de la colección iterable.

Interfaz *Iterator<E>*

boolean	hasNext()	Retorna true si la iteración tiene más elementos.
E	next()	Retorna el siguiente elemento en la iteración. Si no hay siguiente elemento se produce una excepción: *NoSuchElementException*
default void	remove()	Elimina de la colección el último elemento retornado por este iterador (*opcional*).

Interfaz *List<E>*

boolean	add(E e) Añade el elemento especificado al final de esta lista (*opcional*).
void	add(int index, E element) Inserta el elemento especificado en la posición de la lista especificada (*opcional*).
E	get(int index) Retorna el elemento en la posición especificada
ListIterator<E>	listIterator() Retorna un iterador sobre los elementos de esta lista según la propia secuencia.
E	remove(int index) Elimina de la lista el elemento que está en la posición especifiacda (*opcional*).
boolean	remove(Object o) Elimina de la lista la primera ocurrencia del elemento especificado si está presente (*opcional*).

Pregunta 1

```
public class Fibonacci implements Iterable<Integer> {
    private int numItems;

    public Fibonacci(int n) {
        numItems = n;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new FibonacciIterator();
    }

    private class FibonacciIterator implements Iterator<Integer> {
        private int term;
        private int previous;
        private int k;

        private FibonacciIterator() {
            term = 1;
            previous = 0;
            k = 0;
        }

        @Override
        public boolean hasNext() {
            return k < numItems;
        }

        @Override
        public Integer next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }

            int current = previous;
            term += previous;
            previous = term - previous;
            k++;

            return current;
        }

    }

}

public static void print(int n) {
    Fibonacci fib = new Fibonacci(n);

    for (int term: fib) {
        System.out.printf("%d ", term);
    }
    System.out.println();
}
```

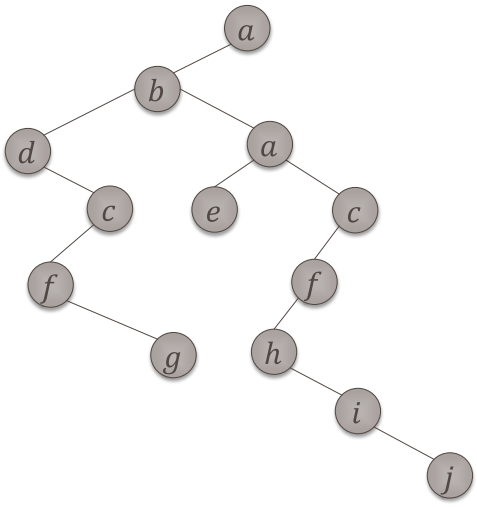
Pregunta 2

```
public CMutable(Collection<? extends E> c) {
    data = new LinkedList<>();  // o ArrayList<>()
    addAll(c);
}

@Override
public boolean add(E e) {
    if (e == null) {
        throw new NullPointerException();
    }

    return contains(e) ? false : data.add(e);  // o data.contains(e)
}


public CInmutable(Collection<? extends E> c) {
    CInmutable<E> other;

    if (c instanceof CInmutable<?>) {
        other = (CInmutable<E>)c;
        data = other.data;
    } else {
        other = new LinkedList<>();  // o ArrayList<>()
        for (E e: c) {
            if (e == null) {
                throw new NullPointerException();
            }
            if (!contains(e)) {     // o !data.contains(e)
                 data.add(e);
            }
        }
    }
}
```

Pregunta 3

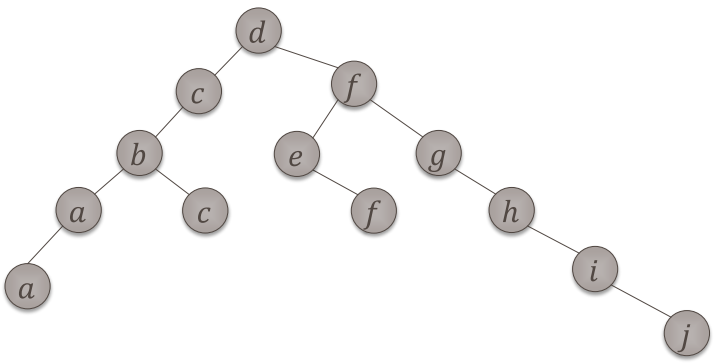
1. d f g c b e a h i j f c a

2.



La secuencia en inorden coincide con la anterior.

3.



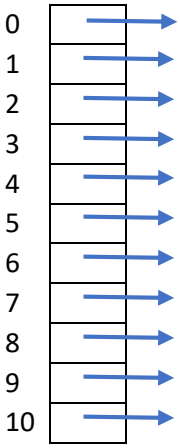
Ejercicio 4

a) HashSet implementa Conjuntos y HashMap implementa Diccionarios.

b) Todas las tablas hash en JAVA son de tipo abierto (tablas de colecciones). Hay que decir aquí las características de estas tablas: función hash, coste de las operaciones, …

c)

HashSet<Integer>

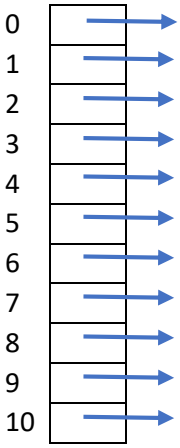


(11) h(11)=11%11=0

(25,3,14) h(25)=h(3)=h(14)=3

(10,21) h(10)=h(21)=10 (Como es un conjunto, el 10 no se mete dos veces)

HashMap<Integer,String>



((11,Andrea)) h(11)=11%11=0

((25,Luis) (3,Pedro) (14,Jorge)) h(25)=h(3)=h(14)=3

((10, Ana)(21,Marta)) h(10)=h(21)=10 (Como es un diccionario, "Ana" sustituye a "Juan" cuando se mete la clave 10 la segunda vez)

Ejercicio 5

a) Como matriz:

	A	B	C	D	E	F
A					0.1	0.9
B	0.36		0.3	0.4		
C				0.6	0.4	
D					1.6	
E	0.55					0.45
F				1		

También vale si se numeran los vértices: A=1, B=2, etc

Como lista de adyacencias:

(A, ((E, 0.1) (F, 0.9))

(B, ((A, 0.36) (C, 0.3) (D, 0.3))

(C, ((D, 0.6) (E, 0.4))

(D, ((E, 1.6))

(E, ((A, 0.55) (E, 0.45))

(F, ((D, 1))

b)

No se puede usar HashSet (de manera sencilla) ni para listas ni para matrices.

Si se puede usar HashMap tanto para listas como para matrices:

- Para listas es similar a la implementación de clase: HashMap<Vertex, HashMap<Vertex, Double>
- Para matrices estamos en el caso de matrices dispersas (sparse), por lo que hay que guardar pares de la forma (*fila*, *info_de_la_fila*) donde *info_de_la_fila* son pares (adyacente, peso).
    - Si no se numeran los vértices: HashMap<Vertex, HashMap<Vertex, Double>
    - Si se numeran los vértices: HashMap<Integer, HashMap<Integer, Double>

c)

Falta la explicación (uso de la cola y elección de los vértices). Faltan también los arcos que no forman parte de los árboles.