Ejercicios de clase

TEMA 4 – Map Ordenado y Árbol Binario de Búsqueda (ABB)

Ejercicio 1

Diseña un método que devuelva una ListaConPI con las entradas de un *MapOrdenado* ordenadas ascendentemente.

SOLUCIÓN:

```
public static <C extends Comparable<C>> ListaConPI<EntradaMap<C, V>>
       entradas(MapOrdenado<C, V> m) {
  ListaConPI<EntradaMap<C, V>> l = new LEGListaConPI<EntradaMap<C, V>>();
  EntradaMap<C, V> e = m.recuperarEntradaMin();
  l.insertar(e);
  for (int i = 1; i < m.talla(); i++) {
    e = m.sucesorEntrada(e.getClave());
    l.insertar(e);
  }
  return l;
}
```

Ejercicio 2

Diseña un método estático, genérico e iterativo *mapSort* que, con la ayuda de un *MapOrdenado*, ordene los elementos (*Comparable*) de un array *v*.

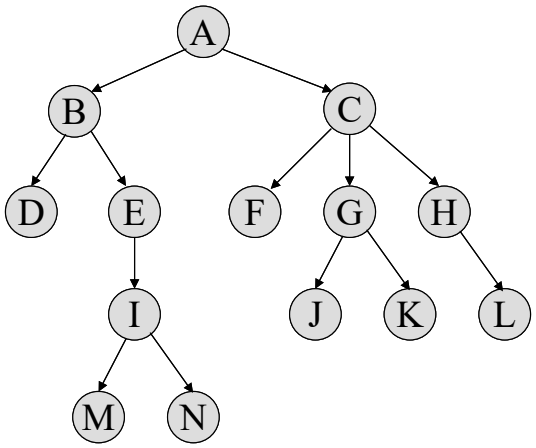
SOLUCIÓN:

```
public static <C extends Comparable <C>> void mapSort(C[] v) {
  MapOrdenado<C, C> m = new ABBMapOrdenado<C, C>();
  for (int i = 0; i < v.length; i++) {
    m.insertar(v[i], v[i]);
  }
  C x = m.recuperarMin();
  v[0] = x;
  for (int i = 1; i < v.length; i++) {
    x = m.sucesor(x);
    v[i] = x;
  }
}
```

Ejercicio 3

Responde a las siguientes cuestiones:

a) ¿Cuántas aristas tiene un árbol con N nodos?
b) ¿Longitud de A a D?
c) ¿Longitud de C a K?
d) ¿Longitud de B a N?
e) ¿Longitud de B a B?
f) ¿Profundidad de A, B, C y F?
g) ¿Altura de B, C, I, F y del árbol?

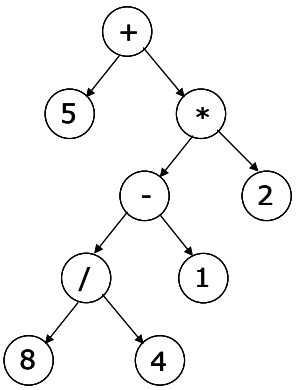


SOLUCIÓN:

a) A = N – 1
b) 2
c) 2
d) 3
e) 0
f) 0, 1, 1 y 2
g) 3, 2, 1, 0 y 4

Ejercicio 4

Muestra el resultado de recorrer en pre-orden, in-orden, post-orden y por niveles el siguiente árbol:



SOLUCIÓN:

Pre-orden: + 5 * – / 8 4 1 2
In-orden: 5 + 8 / 4 – 1 * 2
Post-orden: 5 8 4 / 1 – 2 * +
Por niveles: + 5 * – 2 / 1 8 4

Ejercicio 5

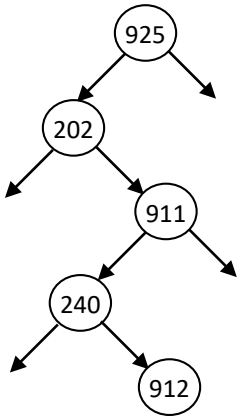
Si se busca el número 363 en un ABB que contiene números del 1 al 1000. ¿Cuál de las siguientes secuencias de nodos no puede ser la secuencia de nodos examinada?

a) 2, 252, 401, 398, 330, 344, 397, 363
b) 924, 220, 911, 244, 898, 258, 362, 363
c) 925, 202, 911, 240, 912, 245, 363
d) 2, 399, 387, 219, 266, 382, 381, 278, 363
e) 935, 278, 347, 621, 299, 392, 358, 363

SOLUCIÓN:

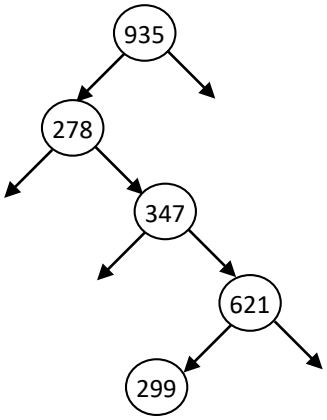
La **c)** y la **e)** no son posibles:

c)



Error: 912 es mayor que 911

e)



Error: 299 es menor que 347

Ejercicio 6

Diseñar un método que devuelva el dato que está en el nodo padre de un elemento dado. Indica el coste temporal del método.

SOLUCIÓN:

```
public E padre(E x) {
  if (raiz != null && raiz.dato.compareTo(x) == 0) return null;
  return padre(raiz, x);
}
```

```
protected E padre(NodoABB<E> n, E x) {
  if (n == null) return null;
  if (n.izq != null && n.izq.dato.compareTo(x)==0) return n.dato;
  if (n.der != null && n.der.dato.compareTo(x)==0) return n.dato;
  int resC = n.dato.compareTo(x);
  if (resC < 0) return padre(n.der, x);
  else return padre(n.izq, x);
}
```

Talla:

N = tamaño del nodo "n" (tamaño del árbol en la llamada más alta)

Instancias significativas:

- Mejor caso: el padre es el nodo "n" (el raíz en la llamada más alta)
- Peor caso: el dato *x* no está en el árbol

Ecuaciones de recurrencia:

$T_{padre}^{M}(N) = k1$

$T_{padre}^{P}(N = 0) = k2$

$$T_{padre}^{P}(N > 0) = \begin{cases} 1 * T_{padre}^{P}(N/2) + k3, \text{ si el árbol está equilibrado} \\ 1 * T_{padre}^{P}(N-1) + k3, \text{ para un árbol desequilibrado} \end{cases}$$

Coste asintótico:

$T_{padre}(N) \in \Omega(1)$

$T_{padre}(N) \in O(\log_2 N)$, si el árbol está equilibrado

$T_{padre}(N) \in O(N)$, si el árbol está desequilibrado

Ejercicio 7

Diseña un método que devuelva el nivel del nodo que contiene el dato *x* (se supone que no hay datos duplicados)

SOLUCIÓN:

```
public int buscarNivel(E x) {
  int nivel = 0;
  NodoABB<E> n = raiz;
  while (n != null) {
       int resC = n.dato.compareTo(x);
       if (resC < 0) n = n.der;
       else if (resC > 0) n = n.izq;
      else return nivel;
       nivel++;
  }
  return -1; // No encontrado
}
```

Ejercicio 8

Diseña un nuevo constructor para la clase ABB que, partiendo de un ABB vacío, inserte los datos de un vector de forma que el ABB resultante quede equilibrado.

SOLUCIÓN:

```
public ABB(E v[]) {
  Ordenacion.quickSort(v);
  raiz = insertarEquilibrado(v, 0, v.length – 1);
}

protected NodoABB<E> insertarEquilibrado(E v[], int izq, int der) {
  if (izq > der) return null;
  int mitad = (izq + der) / 2;
  return new NodoABB<E>(v[mitad],
            insertarEquilibrado(v, izq, mitad - 1),
            insertarEquilibrado(v, mitad + 1, der));
}
```

Ejercicio 9

Diseñar los siguientes métodos en la clase *ABB:*

- Obtener el número total de hojas del árbol
- Visualizar los datos de los nodos del nivel *k* del árbol
- Calcular la altura del árbol

SOLUCIÓN:

a) Obtener el número total de hojas del árbol

```
public int numHojas() {
  return numHojas(this.raiz);
}

protected int numHojas(NodoABB<E> n) {
  if (n == null) return 0;
  else if (n.izq == null && n.der == null) return 1;
  else return numHojas(n.izq) + numHojas(n.der);
}
```

b) Visualizar los datos de los nodos del nivel *k* del árbol

```
public void verNivel(int nivel) {
  String res = verNivel(this.raiz, nivel);
  System.out.println(res);
}

protected String verNivel(NodoABB<E> n, int nivel) {
  String res;
  if (n == null) res = "";
  else {
    if (nivel == 0) res = n.dato.toString() + "\n";
    else res = verNivel(n.izq, nivel-1) + verNivel(n.der, nivel-1);
  }
  return res;
}
```

c) Calcular la altura del árbol

```
public int altura() {
  return altura(this.raiz);
}

protected int altura(NodoABB<E> actual) {
  if (actual == null) return -1;
  return 1 + Math.max(altura(actual.izq), altura(actual.der));
}
```

Ejercicio 10

Diseña la clase *ABBInteger* como un *ABB* que trabaja con datos de tipo *Integer*, y añade los siguientes métodos:

- Un método que obtenga la suma de todos los elementos que sean mayores o iguales a un valor entero dado
- Un método que cambie el signo de todos los datos del árbol. El *ABB* debe seguir manteniendo la propiedad de orden.

SOLUCIÓN:

```
public class ABBInteger extends ABB<Integer> {

  public int sumaMayoresOIguales(int x) {
    return sumaMayoresOIguales(new Integer(x), this.raiz);
  }

  protected int sumaMayoresOIguales(Integer x, NodoABB<Integer> n) {
    if (n == null) return 0;
    int res = sumaMayoresOIguales(x, n.der);
    if (n.dato.compareTo(x) >= 0)
      res = n.dato.intValue() + sumaMayoresOIguales(x, n.izq);
    return res;
  }

  public void cambiarSigno() {
    this.raiz = cambiarSigno(this.raiz);
  }

  protected NodoABB<Integer> cambiarSigno(NodoABB<Integer> n) {
    if (n != null) {
      n.dato = new Integer(-n.dato.intValue());
      n.izq = cambiarSigno(n.der);
      n.der = cambiarSigno(n.izq);
    }
    return n;
  }
}
```

Ejercicio 11

Diseñar un método en un *ABB* para eliminar todos los elementos menores que uno dado.

SOLUCIÓN:

```
public void eliminarMenores(E x) {
  this.raiz = eliminarMenores(this.raiz, x);
}

protected NodoABB<E> eliminarMenores(NodoABB<E> actual, E x) {
  if (actual == null) return null;
  NodoABB<E> res = actual;
  int resC = actual.dato.compareTo(x);
  if (resC < 0) {          // Borramos el nodo actual y su subárbol izquierdo
    res = eliminarMenores(actual.der, x);
  } else {                 // Buscamos menores por la izquierda
    actual.izq = eliminarMenores(actual.izq, x);
  }
  if (res != null) res.talla = 1 + talla(res.izq) + talla(res.der);
  return res;
}
```

Ejercicio 12

Diseña en la clase *ABB* un método para obtener el predecesor de un dato *x* dado.

- El predecesor de un nodo es el máximo de su subárbol izquierdo (si tiene) o, en caso contrario, el ascendiente por la izquierda más cercano.

SOLUCIÓN:

```
public E predecesor(E x) {
  NodoABB<E> pred = predecesor(x, this.raiz, null);
  if (pred == null) return null;
  else return pred.dato;
}

protected NodoABB<E> predecesor(E x, NodoABB<E> actual, NodoABB<E> ascenIzq) {
  if (actual == null) return null;
  int resC = actual.dato.compareTo(x);
  if (resC == 0) {
    if (actual.izq != null) return buscarMax(actual.izq);
    else return ascenIzq;
  } else if (resC < 0) return predecesor(x, actual.der, actual);
  else return predecesor(x, actual.izq, ascenIzq);
}
```

```
protected NodoABB<E> buscarMax(NodoABB<E> actual) {
  while (actual.der != null) actual = actual.der;
  return actual;
}
```

Ejercicio 13

Diseña en la clase ABB un método que devuelva el número de elementos del árbol que están dentro de un intervalo dado [x, y]

SOLUCIÓN:

```
public int contarEnIntervalo(E x, E y) {
  return contarEnIntervalo(raiz, x, y);
}

private int contarEnIntervalo(NodoABB<E> actual, E x, E y) {
  if (actual == null) return 0;
  int c1 = actual.dato.compareTo(x);
  if (c1 < 0) return contarEnIntervalo(actual.der, x, y);
  int c2 = actual.dato.compareTo(y);
  if (c2 > 0) return contarEnIntervalo(actual.izq, x, y);
  return 1 + contarEnIntervalo(actual.izq, x, y) + contarEnIntervalo(actual.der, x, y);
}
```