ESTRUCTURA DE DATOS Y ALGORITMOS **8-Septiembre-2006**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

Nombre-Apellidos:	**Nota:**

1. Ejercicio (1 punto)

Responder a las siguientes preguntas de tipo test, seleccionando sólo una de las respuestas. Se evaluarán sólo las respuestas contestadas con el siguiente criterio: respuesta acertada **positivo(+0,1)**, respuesta fallada **negativo(-0,025)**.

1. Un vector tiene la siguiente propiedad:
   a. Inserciones y borrados eficientes en medio de la estructura
   b. Inserciones y borrados eficientes en ambos extremos de la estructura
   c. Inserciones y borrados eficientes al final de la estructura
   d. Ninguna de las anteriores

2. Un vector se puede recorrer de la siguiente forma:
   a. Secuencial
   b. Bidireccional
   c. Saltos aleatorios
   d. Todas las anteriores

3. Una lista tiene la siguiente propiedad:
   a. Inserciones y borrados eficientes en medio de la estructura
   b. Inserciones y borrados eficientes al principio de la estructura
   c. Inserciones y borrados eficientes al final de la estructura
   d. Todas las anteriores

4. Una lista enlazada simple tiene la siguiente propiedad:
   a. Acceso secuencial
   b. Acceso bidireccional
   c. Acceso aleatorio
   d. Todas las anteriores

5. La probabilidad de no encontrar suficiente memoria disponible
   a. Es mayor para un vector que para una lista
   b. Es mayor para una lista que para un vector
   c. La probabilidad es la misma para el vector que para la lista
   d. Si la lista es doblemente enlazada B es cierto; si es simplemente enlazada A es cierto

6. Las pilas son conocidas porque realizan las operaciones de inserción y eliminación de acuerdo al principio de:
   a. FIFO
   b. FILO
   c. LIFO
   d. LILO

7. Las colas son conocidas porque realizan las operaciones de inserción y eliminación de acuerdo al principio de:
   a. FIFO
   b. FILO
   c. LIFO
   d. LILO

8. Las tablas hash son utilizadas porque:
   a. Se deben de recorrer los elementos en orden
   b. No se conoce el número de elementos a priori
   c. Se quiere utilizar un mismo índice para varios elementos
   d. Ninguna de las anteriores

9. Cual de las siguientes afirmaciones es cierta:
   a. Las estructuras de datos almacenan operaciones y las aplican sobre un objeto
   b. Las estructuras de datos restringen el uso de las variables
   c. Las estructuras de datos almacenan variables de objetos
   d. Las estructuras de datos almacenan referencias a objetos

10. Las principales diferencias entre las distintas estructuras son:
    a. La forma de organizar las operaciones y las restricciones que imponen sobre los datos
    b. La forma de organizar los datos y las restricciones que imponen sobre las operaciones
    c. La forma en que aplican la abstracción
    d. En principio, no existe ninguna diferencia sustancial, depende del problema planteado

ESTRUCTURA DE DATOS Y ALGORITMOS **8-Septiembre-2006**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

Nombre-Apellidos:	**Nota:**

1. Ejercicio (2,5 puntos)

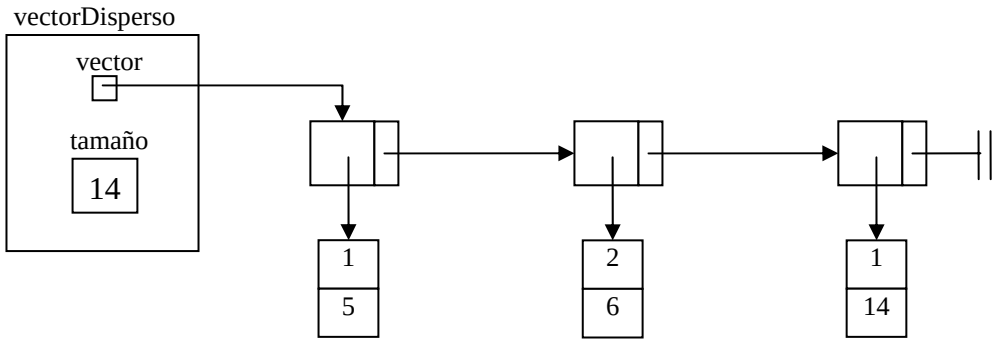
Un vector (matemático) disperso es aquel que tiene muchos elementos que son cero. Ej: v=(0,0,0,0,1,2,0,0,0,0,0,0,0,1). Realiza las siguientes tareas:

1. Escribe en Java las estructuras de datos para representar mediante listas un vector disperso.
2. Representa en memoria un ejemplo de un vector disperso.
3. Escribe en Java un método para sumar dos vectores dispersos.

1. Escribe en Java las estructuras de datos

```
public class VectorDisperso  {
   LinkedList vector;        // Para un mejor uso de la memoria, en
      // la lista se guardan sólo aquellos elementos del vector que no
      // tengan como valor el 0. Si las posiciones de los elementos
      // contiguos (ej: pos1 y pos2) no son contiguos (es decir,
      // pos1+1 < pos2) significa que entre esas dos posiciones hay
      // tantos ceros como pos2-pos1-1. En el caso del primer elemento,
      // pos1 es 0 y pos2 es la posición del primer elemento. Los
      // elementos están ordenados por su posición.
   LinkedListItr itrVector;
   int tamaño;    //indica el número de elementos en el vector
}
// Objeto a insertar en la lista, que representa un elemento del vector
public class Elemento {
   int valor;     // indica el valor de un elemento en el vector
   int posicion;  // indicar la posición del elemento en el vector
}

class Node {                        public class LinkedListItr {
  Object element;                     LinkedList theList;
  Node next;                          Node current;
}                                     Node preceding;
public class LinkedList{            }
  Node top;
}
```

2. Representa en memoria un ejemplo de un vector disperso: ver ejemplo del enunciado



3. Escribe en Java un método para sumar dos vectores dispersos:

Implementación:

```
class Elemento {
  ...
  public Elemento(Elemento e){
      Elemento(e.getValor(), e.getPosicion());
  }
```

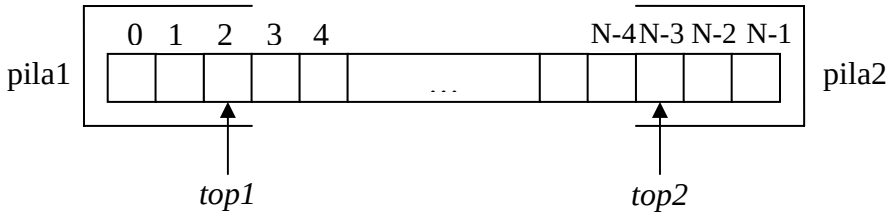
```
  public Elemento(int valor, int posicion){
    this.valor = valor;
    this.posicion = posicion;
  }
}

class VectorDisperso {
  ...
  public VectorDisperso(int tamaño){
    vector = new LinkedList();
    itrVector = new LinkedListItr(vector);
    this.tamaño = tamaño;
  }

  public VectorDisperso sumar(VectorDisperso v){
    // para sumar dos vectores, el numero de elementos de cada vector tiene que ser igual
    if (this.tamaño == v.tamaño){
          VectorDisperso vResultado = new VectorDisperso(this.numElems);
          Elemento e1, e2, e3;
          int pos1, pos2;
           LinkedListItr v1 = this.itrVector;
          LinkedListItr v2 = v.itrVector;
          v1.goFirst();
          v2.goFirst();
   // OJO: las posiciones de los elementos en el vector no coinciden con los de la lista,
   // porque los ceros no se guardan
          while ( v1.isOnList() && v2.isOnList() ) {
             e1 = (Elemento) v1.getCurrent();
             e2 = (Elemento) v2.getCurrent();
             pos1 = e1.getPosicion();
             pos2 = e2.getPosicion();
             if (pos1 == pos2){
                 e3 = new Elemento(e1.getValor()+e2.getValor(), pos1);
                 v1.advance();
                 v2.advance();
             }
             else if (pos1 < pos2){
                 e3 = new Elemento(e1);
                 v1.advance();
             }
             else {
                 e3 = new Elemento(e2);
                 v2.advance();
             }
             vResultado.addElement(e3);
          }
          // copiar los elementos que quedan en el primer vector
          while (v1.isOnList())
             e1 = (Elemento) v1.getCurrent();
             vResultado.addElement(new Elemento(e1));
             v1.advance();
          }
          // copiar los elementos que quedan en el segundo vector
          while (v2.isOnList())
             e2 = (Elemento) v2.getCurrent();
             vResultado.addElement(new Elemento(e2));
             v2.advance();
          }
    }
    else throws new Exception("Dos vectores no se pueden sumar si sus tamaños NO
                               son iguales")
  }

  public void addElement(Elemento e){
     itrVector.insert(e);
  }
}
```

ESTRUCTURA DE DATOS Y ALGORITMOS **8-Septiembre-2006**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

2. Ejercicio. (3 puntos)

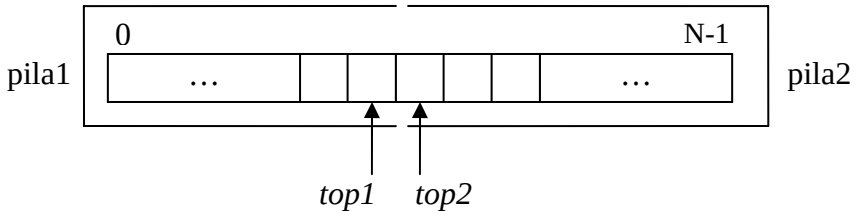
Supongamos que estamos trabajando en un lenguaje de programación que no tiene el tipo de dato referencia. Se plantea la resolución de un problema utilizando dos pilas de números reales. Las dos pilas se quieren implementar sobre un único array, de tal forma que siempre que haya sitio en el array se intentará utilizarla al máximo simulando que el tamaño de las pilas no es finita. Realiza las siguientes tareas:

a) Escribir la estrategia a seguir para la representación comentada anteriormente: colocación de las pilas, variables necesarias para…, etc.



Tal como se representa en la imagen de arriba, las pilas se sitúan en los extremos de un array de *N* elementos. La *pila1* se sitúa en el extremo izquierdo, siendo su primer elemento el de la posición 0 y creciendo hacia la parte derecha. Se necesitará de una variable (*top1*) para indicar cual es el elemento de la cima de la *pila1*. La *pila2* se sitúa en el extremo derecho, siendo su primer elemento el de la posición N-1 y creciendo hacia la parte izquierda. Del mismo modo que la *pila1*, también se necesitará de una variable (*top2*), el cual indicará el elemento de la cima de la *pila2*. Con estas variables (*top1* y *top2*) se podrá controlar también, si las pilas están vacías. Para este caso, los valores de las variables tendrán que indicar algún valor que no sea valido (p.ej *top1=-1 y top2=N*, respectivamente).

Por otro lado, aunque haya que simular que las pilas no tienen límite, al utilizar como implementación un array en algún momento se vuelven finitas. Este es el caso que se da cuando el array está completo. En ese caso, no se podrá insertar o meter ningún elemento en ninguna de las dos pilas. Este caso se dará cuando las cimas de las dos pilas se estén pegando, es decir, cuando se cumpla que *top1+1 es igual a top2*. La siguiente imagen muestra este caso:



b) Dada la representación anterior escribe en Java la siguiente clase con sus métodos correspondientes:

Clase **DosPilas** {
método DosPilas(): inicializa ambas pilas como pila vacía.
método Esvacia1() y Esvacia2(): determinan si las respectivas pilas están vacías.
método Meter1() y Meter2(): introduce un número real en la respectiva pila.
método Suprimir1() y Suprimir2(): extrae un número real de la respectiva pila.
}

Nota: En las últimas operaciones de meter y suprimir hay que tener en cuenta condiciones de error. Por otro lado, comentar que las cabeceras de los métodos no están completas.

Implementación:

```
class Dospilas{
      public static final int CAPACIDAD_MAX = 100;
      double[] valores;  // array en el que se guardan los valores de las pilas
      int top1;          // indica la cima de la pila1
      int top2;          // indica la cima de la pila2

      public Dospilas(){
            pospilas(CAPACIDAD_MAX);
      }

      public Dospilas(int capacidad){
            valores = new double[capacidad]; // indice∈[0..capacidad-1]
            top1 = -1;
            top2 = capacidad;
      }

      public boolean esVacia1(){
            return (top1 == -1);
      }

      public boolean esVacia2(){
            return (top2 == CAPACIDAD_MAX);
      }

      public void meter1(double valor) throws StackFullException {
            if (completo())
                  throw new StackFullException(“Pila1 está completo”);
            top1++;
            valores[top1]=valor;
      }

      public void meter2(double valor) throws StackFullException {
            if (completo())
                  throw new StackFullException(“Pila2 está completo”);
            top2--;
            valores[top2]=valor;
      }

      public double suprimir1() throws StackEmptyException {
            if (esVacia1())
                  throw new StackEmptyException(“Pila1 está vacía”);
            double valor = valores[top1];
            top1--;
            return valor;
      }
```

```
      public double suprimir2() throws StackEmptyException {
            if (esVacia2())
                  throw new StackEmptyException("Pila2 está vacía");
            double valor = valores[top2];
            top2++;
            return valor;
      }

      // comprueba si el array está completo o no. Devuelve true si lo está y
      // false, en el caso contrario
      private boolean completo(){
            return (top1+1 == top2);
   }
}
```

c) En la clase anterior faltan dos métodos que implementan una de las operaciones esenciales de las pilas. Cuales son? Impleméntalas.

```
class Dospilas{

      ...

      public double cima1() throws StackEmptyException {
            if (esVacia1())
                  throw new StackEmptyException ("Pila1 está vacía");
            return valores[top1];
      }

      public double cima2() throws StackEmptyException {
            if (esVacia2())
                  throw new StackEmptyException ("Pila2 está vacía");
            return valores[top2];
      }
}
```

3. Ejercicio (3,5 puntos)

a) ***Diseñar e implementar en Java*** un método que escriba por salida estándar los términos de un polinomio **ordenados de forma decreciente** por su grado (valor del exponente). El polinomio es representado mediante una tabla hash de N=2*T posiciones, donde T es el número medio de términos en el polinomio.

public static void imprimir(Hashtable polinomio);

El tipo de los elementos en la tabla hash será:

```
class Termino {
      int exp;
      int coef;
}
```

Ejemplo:

Si el polinomio que nos han pasado como parámetro está representado con la siguiente tabla hash (para N= 10):

Indice	Clave	Datos
1	450	16
2	550	24
3	vacía	
4	100	- 20
5	0	8
6	vacia	
7	200	- 48
8	vacía	
9	vacía	
10	vacía	

El resultado que se mostrará en pantalla será el siguiente: $\mathbf{24x^{550} + 16\ x^{450} - 48x^{200} - 20x^{100} + 8}$

b) **Indicar de forma razonada**, suponiendo que todas las operaciones del TAD tablahash son de *O*(1), el **orden** de la operación realizada en el apartado anterior.

c) Explica brevemente por qué en la tabla hash anterior la información no aparece contigua en las casillas correspondientes y no está ordenada.

Nota: cualquier método auxiliar que sea utilizado, es necesario implementarlo

ANEXO I: Especificación de clases

```
class HashTable {
void insert (Object key, Object value);
//pos: inserta el elemento de clave key y datos value. Si la clave ya estaba en la tabla no --
//     hace nada.
void modify (Object key, Object value)
//pos: reemplaza los datos asociados a la clave key de la tabla por value. Si la clave no está
//     en la tabla devuelve no hace nada
Object find (Object key);
//pos: Devuelve los datos asociados a la clave key. Si la clave no está en la tabla devuelve
//     null.
void remove (Object key);
//pos: Borra de la tabla hash el elemento  que coincide con el valor de la clave key. Si la
//     clave no está en la tabla no hace nada.
Boolean exist (Object key);
//pos: Devuelve cierto si la clave key está en la tabla y falso en caso contrario
Enumeration keys ();
///pos: Devuelve un objeto para iterar sobre las claves de la tabla
Enumeration values(); //pos: Devuelve un objeto para iterar sobre los valores de la tabla
}
```

a) ***Diseñar e implementar: public static void imprimir(Hashtable polinomio);***

Diseño

```
Algoritmo imprimir(Hashtable polinomio)

1. numeroTerminos = recuperar_numero_de_terminos(polinomio);
2. exponentes = recuperar_exponentes(polinomio);
3. arrayExponentes = pasarAarray(exponentes, numeroTerminos);
4. ordenarArray(arrayExponentes);
5. Para cada exponente en arrayExponentes
      5.1 exponente = recuperarExponente(arrayExponentes);
      5.2 coeficiente = recuperarCoeficiente(polinomio, exponente);
      5.3 imprimir_en_pantalla(coeficiente+"x"+exponente);
finPara
```

Implementación:

```
public static void imprimir(Hashtable polinomio){
    int numeroTerminos = polinomio.numpos();
    Enumeration exponentes = polinomio.keys();
    int[] arrayExponentes = pasarAarray(exponentes, numeroTerminos);
    ordernarArray(arrayExponentes);
    for(int i=0;i < arrayExponentes.length; i++){
      int exp = arrayExponentes[i];
      int coef = ((Integer) polinomio.find(new Integer(exp))).intValue();
      if (exp > 0)
            System.out.print(coef + "x" + exp);
      else
            System.out.print(coef);
      if ((i+1) < arrayExponents.length)
        System.out.print("+");
    }
}

public static int[] pasarAarray(Enumeration enum, int num){
  int[] enteros = new int[num];
  int i = 0;
  while ((i < num) && enum.hasMoreElements()){
    enteros[i] = ((Integer) enum.nextElement()).intValue();
    i++;
  }
  return enteros;
}

// implementar un algoritmo de ordenación vista en clase: insertionSort,
// selectionSort, MergeSort, QuickSort
```

b) El coste de ejecución del algoritmo 'imprimir' será la suma del coste de ejecución de cada acción, numeradas del 1 al 5. A continuación se detalla el coste de cada acción:

   a. *O*(1), constante: Operación sobre la tabla hash, cuyos métodos son de *O*(1)
   b. *O*(1), constante: Operación sobre la tabla hash, cuyos métodos son de *O*(1)
   c. *O*(n), lineal: depende del número de términos (n) guardados en la tabla hash
   d. *O*(n^2), cuadrática: es el coste de ejecutar los siguientes algoritmos de ordenación: insertionSort, selectionSort o Quicksort (aunque en la mayoría de veces será O(nlogn), siempre hay que tener en cuenta el peor de los casos)

      O(nlog n)*:* es el coste de ejecutar el algoritmo de ordenación MergeSort
   e. *O*(n), lineal: depende del número de términos (n) guardados en el array 'arrayExponentes', ya que el coste de todas sus subacciones 5.1, 5.2 y 5.3 es de *O*(1), es decir, constante.

Por lo tanto, el coste total se ve determinado por la acción 4, que será bien O(n^2) o O(nlog n), según implementación del algoritmo de ordenación.

c) No aparece contigua porque las tablas hash aplican una función hash sobre la clave de la información que se va a almacenar, el cual sirve para posicionarlo en el array interno**.**