ESTRUCTURA DE DATOS Y ALGORITMOS **4-Febrero-2010**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**
Nombre y apellidos: _____ Nota:

1. Ejercicio (3 puntos)

a) ¿Qué estructura de datos utilizarías para transformar la siguiente expresión a su equivalente expresión postfija? Argumenta tu decisión.

Ej. entrada: `(x-y)/(z+w)-(z+y)^x` Salida: `xy-zw+/zy+x^-`

Utilizaría una pila, porque se necesitan apilar los operadores según su precedencia o prioridad, para ello también habría que apilar los paréntesis de apertura. Los operadores se desapilarán cuando en la expresión de entrada se haya leído: 1) un operador que tiene menor o igual precedencia que el de la cima; o 2) un paréntesis de cierre, en ese caso se desapilan hasta el paréntesis de apertura.

b) La solución de un problema concreto requiere la ejecución de las dos siguientes operaciones una y otra vez sobre un conjunto de datos: 1) buscar un dato por contenido y 2) insertar más datos después del dato encontrado en 1). ¿Qué estructura de datos sería la más adecuada? Argumenta tu decisión.

Una lista enlazada (independiente del tipo de lista) sería la más adecuada, porque al no utilizar una estructura estática en memoria para la representación de los datos, las operaciones comentadas no requieren desplazamiento de los datos anteriores y además hace un uso justo de la memoria requerida. También podría ser valido un árbol con los mismos argumentos.

c) Cual es la complejidad del algoritmo de búsqueda de un elemento por posición en una lista? Es siempre igual? Argumenta tu respuesta.

Se podría particularizar y decir que: "para la lista vacía y primer y último elemento es O(1) y O(n) para otros casos". Pero hay que recordar que la complejidad de un algoritmo lo caracteriza el caso peor. **En este caso la complejidad depende de la implementación de la lista. Si es enlazada, es decir, dinámica entonces es O(n) y si fuera estática, se utilizaría un array o vector, entonces sería O(1).**

d) En las TablasHash abiertas o hashing enlazado, la operación de insertar un elemento se ha realizado como primer elemento de la lista. Si la inserción en la lista enlazada se hace de tal forma que esté ordenada respecto a la clave. ¿Qué ventajas y desventajas tiene respecto a la eficiencia de las operaciones *insertar*, *buscar* y *eliminar*?

Esta estructura En el caso de insertar es una desventaja, ya que en la alternativa que se propone se requiere insertar ordenado en la posición correspondiente, lo cuál dependerá del número de colisiones (misma clave) que se produzcan (tamaño de la lista) y dejará de ser constante.
En cuanto a buscar y eliminar, podemos decir que es ventaja en los casos en que la clave que se quiere buscar o eliminar no exista. Antes, siempre requería recorrer toda la lista y en cambio, con la nueva alternativa sólo se recorre hasta encontrarlo o encontrar una clave mayor o menor (dependiendo del criterio de ordenar) a la que se está buscando.

e) Muestra que es posible soportar simultáneamente las operaciones siguientes en tiempo constante: `apilar`, `desapilar`, `cima` y `buscarMin`. Observa que `eliminarMin` no forma parte del repertorio. ***Pista***: mantener dos pilas.

1. Explica que es lo que almacenarías en cada una de las pilas y la estrategia que sigues o el uso que haces de las dos pilas.

En la pila1 apilaremos todos los datos, según se vayan apilando. La cima indica el último dato apilado, el cuál será el primero en desapilar. Así, los métodos apilar, desapilar y cima se ejecutarán en tiempo constante.
En cambio, en la pila2 solamente apilaremos los datos que son mínimos según vayan apareciendo. La cima indica el valor mínimo actual. De está forma se garantiza que buscarMin se ejecute en tiempo constante. En está pila no tienen porque estar todos los datos. Apilaremos un dato en la pila2 cuando se de alguno de los siguientes casos: 1) el dato que apilamos es menor que el que está en la cima de la pila2; y 2) después de ejecutar desapilar o eliminarMin, el dato que está en la cima de la pila1 sea menor que el dato en la cima de la pila2.

2. representa el estado de las dos pilas, después de ejecutar aquellas instrucciones, que cambian el estado de las dos pilas (p.ej. `apilar`, `desapilar` y `eliminarMin`), del siguiente programa:

programa	1) PilaCP p = new PilaCP();	2) p. apilar(new Integer(6));
estado		
programa	3) p. apilar(new Integer(4));	4) p. apilar(new Integer(7));
estado		
programa	5) p. apilar(new Integer(5));	6) p. apilar(new Integer(3));
estado		
programa	7) p. cima();	8) p.buscarMin();
devuelve		
programa	9) p.desapilar();	**devuelve**
estado		
programa	10) p. cima();	11) p.buscarMin();
devuelve		
programa	12) p.eliminarMin();	**devuelve**
estado		
programa	13) p. cima();	14) p.buscarMin();
devuelve		
programa	15) p.eliminarMin();	**devuelve**
estado		
programa	16) p. cima();	17) p.buscarMin();
devuelve		
programa	18) p.eliminarMin();	**devuelve**
estado		

ESTRUCTURA DE DATOS Y ALGORITMOS **4-Febrero-2010**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**
Nombre y apellidos: _____

2. *Ejercicio (2 puntos)*

Se pide:

a) (0,75 punto)

Diseño

El diseño principal sería utilizar dos estructuras de datos iguales (p.ej *Vector*, *Array*, *VectorOrdenado* o *ArrayOrdenado*) con la característica de que los datos se almacenan ordenados (p.ej de menor a mayor). Sería mejor algo de tipo Vector que Array, porque no se sabe el número de contactos. Además se almacenarían los mismos datos en las dos estructuras de datos, en una de ellas ordenados por el nombre del contacto y en la otra por teléfono. El uso del Vector y el ordenar sería principalmente para luego realizar una búsqueda eficiente, tal como, la *búsqueda binaria*. Las dos funcionalidades o características requieren que los datos a almacenar sean comparables, bien implementando la interfaz *Comparable* o *Comparator*.

Principalmente, la dificultad reside en implementar las clases que van a representar los datos a almacenar. Tenemos dos opciones (con implementar una suficiente):

1) Utilizar la clase *Contacto* para representar los datos y dos clases para realizar las comparaciones, en una por nombre y en la otra por teléfono, implementando para ello la interfaz *Comparator*.
2) Una clase genérica (p.ej. *ElementoAgenda*), que represente los datos *clave-valor* e implemente la interfaz *Comparable* comparando la *clave*. Además de dos concretas (p.ej *NombreContacto* y *TelefonoContacto*) para realizar las correspondientes relaciones y recuperar datos.

En las dos opciones, cualquier método relacionado con insertar, buscar o eliminar requiere ser genérica y parametrizada con la correspondiente interfaz. Un ejemplo de ello lo veremos en el apartado b) de este ejercicio.

Falta figura

Implementación

```
public class Agenda {
        / * Por un lado almacenaremos los contactos ordenados por nombre. Dos opciones:
          * 1) se almacenan elementos de tipo Contacto ordenados según ContactosPorNombre
          * 2) se almacenan elementos de tipo NombreContacto ordenado por clave
           */
        VectorOrdenado agendaPorNombre = new VectorOrdenado();
        / * Por otro lado almacenaremos los contactos ordenados por teléfono. Dos opciones:
          * 1) se almacenan elementos de tipo Contacto ordenados según ContactosPorTelefono
          * 2) se almacenan elementos de tipo TelefonoContacto ordenado por clave
           */
        VectorOrdenado agendaPorTelefono = new VectorOrdenado();
}
```

Dos posibles implementaciones de datos serían las siguientes (puede que haya más):

```
1) public class ContactosPorNombre implements Comparator {
      public int compare (Object obj1, Object obj2){
        Contacto c1 = (Contacto) obj1;
        Contacto c2 = (Contacto) obj2;
        return c1.getNombre().compareTo(c2.getNombre());
      }
   }
   public class ContactosPorTelefono implements Comparator {
      public int compare (Object obj1, Object obj2){
        Contacto c1 = (Contacto) obj1;
        Contacto c2 = (Contacto) obj2;
        return c1.getTelefono().compareTo(c2. getTelefono());
      }
   }
```

```
2) public class ElementoAgenda implements Comparable{
      private String key;
      private ElementoAgenda valor;
      public int compareTo(Object obj){
            if (obj instanceof ElementoAgenda) {
                  ElementoAgenda elem = (ElementoAgenda) obj;
                  return this.clave.equals(elem.clave);
            }
            return false;
      }
      public abstract String getNombre();
      public abstract String getTelefono();
}

public class NombreContacto extends ElementoAgenda{
      public NombreContacto(String nombre){
            super(nombre);
      }
      public ElementoAgenda setTelefono(String telefono){
            if (this.valor == null)
                  this.valor = new TelefonoContacto(telefono);
            else
                  this.valor.clave = telefono;
            return this.valor;
      }
      public String getNombre(){ return key; }
      public String getTelefono(){ return valor.key; }
}

public class TelefonoContacto extends ElementoAgenda{
      public TelefonoContacto (String telefono){
            super(telefono);
      }
      public ElementoAgenda setNombre(String nombre){
            if (this.valor == null)
                  this.valor = new NombreContacto (nombre);
            else
                  this.valor.clave = nombre;
            return this.valor;
      }
      public String getNombre(){ return valor.key; }
      public String getTelefono(){ return key; }
}
```

b) (1 punto) **Implementar** los métodos *buscarTelefono* y *buscarNombre* de la clase *Agenda*. Ten en cuenta que se ejecutan con mucha frecuencia y tu respuesta en el apartado anterior.

```
1) public String buscarNombre(String telefono){
      Comparator porTelefono = new ContactosPorTelefono();
      Contacto buscar = new Contacto(null, telefono);
      Contacto contacto = (Contacto) buscarBinario(agendaPorTelefono,
                                                   porTelefono, buscar);
      if (contacto != null)
            return contacto.getNombre();
      else
            return null;

}
public String buscarTelefono(String nombre){
      Comparator porNombre = new ContactosPorNombre();
      Contacto aBuscar = new Contacto(nombre, null);
      Contacto contacto = (Contacto) buscarBinario(agendaPorNombre, porNombre,
                                                   aBuscar);
      if (contacto != null)
            return contacto.getTelefono();
      else
            return null;
}
public Object buscarBinario(VectorOrdenado v, Comparator comp, Object obj){
        int central;
        int inicio = 0;
        int fin = a.length -1;
        while (inicio <= fin) {
            central = (inicio + fin) / 2;
            if(comp.compare(obj,v.elementAt(central)) == 0)
               return v.elementAt(central);
            if(comp.compare(obj, v.elementAt(central)) > 0)
               inicio = central + 1;
            else // (comp.compare(obj, v.elementAt(central)) < 0)
               fin = central - 1;
        }
        return null;
}
```

c) (0,25 puntos) Calcula cual es la complejidad, en notación *O*, de los métodos implementados en b).

Los dos métodos, *buscarTelefono* y *buscarNombre*, dependen del método *buscarBinario*. Este método realiza una serie de comparaciones, que no es lineal, hasta encontrar el elemento buscado. Principalmente, tiene la característica que, al estar los datos de entrada ordenados, en cada iteración reduce el tamaño de los datos de entrada a la mitad que en el caso anterior (marcada por inicio y fin)

n, n/2, n/4, n/8 ..... esto es: $n/2^i$

Ultima vuelta (caso peor) es cuando $n/2^i = 1 \rightarrow 2^i = n \rightarrow$ Esto es cuando $i = \log_2 n$
Esto quiere decir que se darán $\log_2 n$ vueltas en el bucle (caso peor). Por lo tanto $\boldsymbol{O(\log_2 n)}$

Nota: Se valorarán los aspectos de reutilización.

3. *Ejercicio (1 punto)*

Se pide:

a) **Implementar** las clases necesarias (sin los métodos) para representar una lista enlazada con doble terminación.

```
class Node {
      Object element;
      Node next;
}
public class DoubleEndedLinkedList{
      Node top;
      Node bottom;
}
public class DoubleEndedLinkedListItr {
      DoubleEndedLinkedList theList;
      Node current;
      Node preceding;
}
```

b) **Implementar** la heurística *mover-al-frente* para las listas enlazadas con doble terminación.

```
public class DoubleEndedLinkedListItr {
      /* Este método mueve el nodo 'current' al frente de la Lista.
       * PRE: Este método es principalmente invocado desde el método buscar, el cual
       * ha posicionado la variable 'current' en el nodo que contiene el elemento
       * buscado.
       * POST: el nodo 'current' pasa a ser el primero de la lista.
       */
  private void moverAlFrente(){
      if (theList.top != current && current !=null) {
            if (theList.bottom == current)
                  theList.bottom = preceding;   // 0
            preceding.next = current.next;      // 1
            current.next = theList.top;         // 2
            theList.top = current;              // 3
            preceding = null;                   // 4
      }
  }
}
```

c) **Describir** gráficamente los pasos principales de la heurística *mover-al-frente*, mediante un ejemplo.

Falta figura

5. ***Ejercicio (3 puntos)***

1. Especificación

- Entrada: Un cuento dinámico y un entero, *i*, que representa la posición de un capitulo en un cuento normal.
- Salida: un entero, que representa el número de capítulos que son afirmativos, están sin terminar y además, están en la posición i de los respectivos cuentos normales.

```
public int numeroDeCapitulosEnIafirmativosYsinTerminar(int i)
```

Analizando la situación, vemos que sería más fácil resolver el problema por recursividad. Para ello, necesitamos especificar mejor el problema añadiendo más parámetros como datos de entrada:

1) Necesitamos una forma de representar subcuentos dinámicos, para poder crear subproblemas. Para ello, utilizaremos la definición recursiva del árbol binario.
2) Necesitamos saber cual es la posición del primer capítulo del subcuento en el cuento original.
3) Necesitamos saber también si el primer capítulo del subcuento es afirmativo o no. Este dato lo indica el capitulo anterior del cuento original.

Por lo tanto, añadimos 3 parámetros más al método y la especificación quedaría de la siguiente forma:

- Entrada: *A*, un árbol binario que representa un cuento dinámico; un entero, *i*, que representa la posición de un capitulo en un cuento normal; un entero, *pos*, que indica la posición del primer capítulo de A en el esquema inicial de cuentos dinámicos y un booleano, *esAfirmativo*, que indica si el primer capitulo de A es afirmativo o no.
- Salida: un entero, que representa el número de capítulos que son afirmativos, están sin terminar y además, están en la posición *i* de los respectivos cuentos normales que empiezan en la raíz de A.

```
private int numCapIafirmSinTerminar(BTNode A, int i, int pos, boolean esAfirmativo)
```

2. Diseño

a) casos triviales

1) Árbol vacío → devuelve 0.
2) Árbol no vacío y *i*<*pos* → devuelve 0.
3) Árbol no vacío, *i*=*pos* y no(*esAfirmativo*) → devuelve 0.
4) Árbol no vacío, *i*=*pos*, *sinTerminarPrimerCapitulo*(A) y *esAfirmativo* → devuelve 1.

b) casos generales

1) Árbol no vacío y *i* > *pos* → devuelve

```
numCapIafirmSinTerminar(A.sigCapAfirmativo, i, pos+1, true) +
numCapIafirmSinTerminar(A.sigCapNegativo, i, pos+1, false)
```

Notas:

1) *PrimerCapitulo*(BTNode A) ≈ devuelve el capitulo que está en el nodo raíz de A
2) *sinTerminarPrimerCapitulo*(BTNode A) ≈ devuelve True si *PrimerCapitulo*(A) tiene pregunta y además, no existe alguno (o los dos) de los siguientes capítulos del *PrimerCapitulo*(A), es decir, alguna de las respuestas posibles a la pregunta están sin asociar a los capítulos correspondientes.
3) `A.sigCapAfirmativo` ≈ devuelve el nodo del árbol que contiene el siguiente capitulo, en caso de que se conteste afirmativamente a la pregunta del primer capitulo de A
4) `A.sigCapNegativo` ≈ devuelve el nodo del árbol que contiene el siguiente capitulo, en caso de que se conteste negativamente a la pregunta del primer capitulo de A

3. Implementación

```
public class EsquemaCuentosDinamicos {
  ...
  public int numeroDeCapitulosEnIafirmativosYsinTerminar(int i){
    return itrCapitulos.numeroDeCapitulosEnIafirmativosYsinTerminar(i);
  }
}
public class CapitulosBinTreeItr {
  ...
  public int numeroDeCapitulosEnIafirmativosYsinTerminar(int i){
    return numCapIafirmSinTerminar(bTree.getRoot(), i, 1, false);
  }

  private int numCapIafirmSinTerminar(BTNode A, int i, int pos,boolean esAfirmativo){
    if ((A == null) || (i<pos) || ((i==pos) && !esAfirmativo)) )
        return 0;
    else if ((i==pos) && esAfirmativo && sinTerminarPrimerCapitulo(A))
        return 1;
    else
        return numCapIafirmSinTerminar(A.getLeft(), i, pos+1, true) +
               numCapIafirmSinTerminar(A.getRight(), i, pos+1, false);
  }

  /* PRE: A no es vacio
   * POST: Devuelve True si el capitulo de la ráiz de A tiene pregunta y además, está
   *       sin terminar.
   */
  private boolean sinTerminarPrimerCapitulo(BTNode A){
      Capitulo c = (Capitulo) A.getContent();
      return ((c.tienePregunta()) && ((A.getLeft()==null) || (A.getRight()==null)));
  }
}
```