ESTRUCTURA DE DATOS Y ALGORITMOS **5-Febrero-2004**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

1. Ejercicio (3 puntos)

1.1. Definir los tipos de datos:

```
class Node {
   Object element;
   Node next;
}
public class LinkedList{
   Node top;
}

public class DosListas
{
  LinkedList listaPares;
  LinkedListItr itrListaPares;
  LinkedList listaImpares;
  LinkedListItr itrListaImpares;
}
```

```
public class LinkedListItr {
   LinkedList theList;
   Node current;
   Node preceding;
}
```

1.2. Especificar los métodos que se van a utilizar

public **LinkedList**(); //Construye una lista enlazada vacia
public **LinkedListItr**(LinkedList l);
// Construye una clase iteradora y le asocia una lista enlazada
public void **advance**();
// avanza en la lista con la variable actual, pasando al siguiente nodo del nodo actual
public boolean **isOnList**();
// verifica si la variable actual referencia un nodo de la lista, en caso afirmativo devuelve true
// y en caso contrario false
public void **first**();
//actualiza la variable actual, el cuál referencia el primer nodo de la lista
public Object **getActual**();
// devuelve el objeto que esta referenciado en el nodo, el cuál es accesible mediante la
//variable actual
public void **insert**(Object obj);
// inserta la información obj en la lista, creando un nuevo nodo después del nodo referenciado
//por la variable actual. Finalmente, actual referencia el nuevo nodo creado.
public void **insertFirst**(Object obj);
// inserta la información en la primera posición de la lista, creando un nuevo nodo con la
//información obj. Finalmente, actual referencia el nuevo nodo creado.

Implementación:

```
public DosListas crearDosListas(){
  DosListas dosListas = new DosListas();
  this.primero();
  while (this.estaDentro()){
    Integer contenido = (Integer) this.getActual();
    if ((contenido.intValue()) % 2 == 0)
       dosListas.insertarPar(contenido);
    else
       dosListas.insertarImpar(contenido);
    this.avanzar();
  }
  return dosListas;
}

public class DosListas {
  LinkedList listaPares = new LinkedList ();
  LinkedListItr itrListaPares = new LinkedListItr(listaPares);
  LinkedList listaImpares = new LinkedList ();
  LinkedListItr itrListaImpares = new LinkedListItr(listaImpares);

  public void insertarPar(Object par){
    itrListaPares.insert(par);
  }

  public void insertarImpar(Object impar){
    itrListaImpares.insertFirst(impar);
  }

  public void imprimirPares(){
    itrListaPares.imprimir();
  }

  public void imprimirImpares() {
    itrListaImpares.imprimir();
  }
}

public void insert(Object obj){
  Node nuevoNodo=new Node(obj);
  if (theList.top == null){
    theList.top = nuevoNodo;
    current = theList.top;
  }
  else {
    nuevoNodo.next = current.next;
    current.next = nuevoNodo;
    current = nuevoNodo;
  }
}
public void insertFirst (Object obj) {
  Node nuevoNodo = new Node(obj);
  if (theList.top == null){
    theList.top = nuevoNodo;
    actual = theList.top;
  }
  else {
    nuevoNodo.next = theList.top;
    theList.top = nuevoNodo;
  }
}
```

ESTRUCTURA DE DATOS Y ALGORITMOS **5-Febrero-2004**
Titulación: Ingeniero Técnico en Informática de Gestión **Curso: 2º**

2. Ejercicio. (3 puntos)

1. **Especificar / Parametrizar**
   Entrada: A, un árbol binario de enteros
   Salida: Vacía
   Post: Se ha escrito en pantalla el peso de la rama de mayor longitud, y en caso de misma longitud el de mayor peso.

   public int pesoRamaMayorLongitud();

   A la hora de tratar un nodo debemos conocer su longitud y el peso de la rama de mayor longitud → INMERSION
   Entrada: A, un árbol binario de enteros,
   Salida: La longitud y el peso de la rama de mayor longitud →
   Se necesita una clase Contexto para poder devolver dos valores

```
public class Contexto {
        int longitud;
        int peso;
}
```

   public Contexto pesoRamaMayorLongitud (BTNode A)

2. **Diseño**
   Casos triviales
   Si Vacío(A) → devolver new Contexto(0, 0)
   Si Hoja(A) → devolver new Contexto(Raíz(A),1)

   Caso general

```
Si no Vacío(A) y no Hoja(A) →
    contextoIzq = pesoRamaMayorLongitud(Rama_izq(A));
    contextoDch = pesoRamaMayorLongitud(Rama_dch(A));
    Si (contextoIzq.longitud > contextoDch.longitud) entonces
      longitud = contextoIzq.longitud + 1;
      peso = contextoIzq.peso + Raíz(A);
    Si no si (contextoDch.longitud > contextoIzq.longitud) entonces
      longitud = contextoDch.longitud + 1;
      peso = contextoDch.peso + Raíz(A);
    Si no
      longitud = contextoIzq.longitud + 1;
      Si (contextoIzq.peso > contextoDch.peso) entonces
        peso = contextoIzq.peso;
      si no
        peso = contextoDch.peso;
      fin si
    fin si
    contexto = new Contexto(longitud, peso);
    devolver contexto;
```

3. Implementación

```
public int pesoRamaMayorLongitud() {
  Contexto contexto = pesoRamaMayorLongitud(bTree.root);
  Return contexto.longitud;
}

private Contexto pesoRamaMayorLongitud(BTNode nodo) {
  int longitud;
  int peso;
  int raiz;
  if (nodo==null) return
    new Contexto(0, 0);
  else if ((nodo.left==null) && (nodo.right==null) ) {
    raiz = ((Integer) nodo.content).intValue();
    return new Contexto(raiz,1);
  }
  else {
    raiz = ((Integer) nodo.content).intValue();
    Contexto contextoIzq = pesoRamaMayorLongitud(nodo.left);
    Contexto contextoDch = pesoRamaMayorLongitud(nodo.right);
    if (contextoIzq.longitud > contextoDch.longitud)
    {
      longitud = contextoIzq.longitud + 1;
      peso = contextoIzq.peso + raiz;
    }
    else if(contextoDch.longitud > contextoIzq.longitud) {
      longitud = contextoDch.longitud + 1;
      peso = contextoDch.peso + raiz;
    }
    else {
          longitud = contextoIzq.longitud + 1;
          if (contextoIzq.peso > contextoDch.peso)
        peso = contextoIzq.peso;
          else
        peso = contextoDch.peso;
    }
    Contexto contexto = new Contexto(longitud, peso);
    return contexto;
  }
}

public class Contexto {

  int longitud;
  int peso;

  public Contexto(int longitud, int peso)
  {
    this.longitud = longitud;
    this.peso = peso;
  }

  public void print()
  {
    System.out.println("longitud: "+longitud+"  peso: "+peso);
  }
}
```

3. Ejercicio (4 puntos)

1.

```
public class Solitario
{
  private static final int CUBIERTAS = 0;
  private static final int DESCUBIERTAS = 1;
  private static final int MAX_BARAJAS = 2;
  public static final int MAX_CASILLAS = 4;
  public static final int MAX_COLUMNAS = 7;
  public static final int MAX_CARTAS_A_MOSTRAR = 3;

  Stack[] baraja = new Stack[MAX_BARAJAS];
  Stack[] casillas = new Stack[MAX_CASILLAS+1];
  Stack[] columnas =  new Stack[MAX_COLUMNAS+1];
}
```

2.

```
public void iniciarJuego(String file) {
    crearPilas();
    Token1 token = new Token1(file);
    distribuirCartasEnColumnas(token);
    colocarCartasEnBaraja(token);
    mostrarCartas();
  }

  private void crearPilas() {
    for(int i = 0; i < MAX_BARAJAS; i++)
      baraja[i] = new Stack();
    for(int i = 1; i < MAX_CASILLAS + 1; i++)
      casillas[i] = new Stack();
    for(int i = 1; i < MAX_COLUMNAS + 1; i++)
      columnas[i] = new Stack();
  }

  private void distribuirCartasEnColumnas(Token1 token){
    Datos1 datos;
    int columna = 1;
    while (token.hasMoreTokens() && (columna < MAX_COLUMNAS + 1)){
      //colocar en cada columna cartas cubiertas
      for(int i = 1; i < columna; i++) {
        datos = token.getTokens();
        columnas[columna].push(new Carta(datos.getNumero(), datos.getPinta(), false));
      }
      //colocar la última carta descubierta
      datos = token.getTokens();
      columnas[columna].push(new Carta(datos.getNumero(), datos.getPinta(), true));
      columna++;
    }
  }

  private void colocarCartasEnBaraja(Token1 token){
    //dejarlas en el mismo ordenen que aparecen
    Datos1 datos;
    while (token.hasMoreTokens()) {
      datos = token.getTokens();
      baraja[DESCUBIERTAS].push(new Carta(datos.getNumero(), datos.getPinta(), false));
    }
    reiniciarBaraja();
  }
```

```
public void mostrarCartas()
{
  int cantidad = 0;
  while (!baraja[CUBIERTAS].isEmpty() && cantidad < MAX_CARTAS_A_MOSTRAR)
  {
    Carta carta = (Carta) baraja[CUBIERTAS].pop();
    carta.descubrir();
    System.out.print(carta.getInfo()+"\t");
    baraja[DESCUBIERTAS].push(carta);
    cantidad++;
  }
  System.out.println("");
}
```

```
public void moverDeColumnaAColumna(int origen, int destino, int cantidad)
{
  boolean seHaMovido = moverColumnaDePilaAColumna(columnas[origen], destino, cantidad);
  if (seHaMovido)
    descubrirUltimaCarta(columnas[origen]);
}

/**
 * Reglas para mover una serie de cartas (numero de cartas = cantidad)
 * de cualquier montón a una de las columnas
 */
private boolean moverColumnaDePilaAColumna(Stack pilaOrigen, int destino, int cantidad)
{
  boolean mover = false;
  if (cantidad > 0){
    Stack aux = new Stack();
    Stack pilaDestino = columnas[destino];

    int numCartas = moverCartasDescubiertas(pilaOrigen, aux, cantidad, false);

    //si hay cartas suficientes, verificar si se pueden mover
    if (numCartas == cantidad) {
      Carta cartaOrigen = (Carta) aux.peek();
      int numeroOrigen = cartaOrigen.getNumero();
      boolean colorOrigen = cartaOrigen.getColor();
      if (!pilaDestino.isEmpty()){
        Carta cartaDestino = (Carta) pilaDestino.peek();
        int numeroDestino = cartaDestino.getNumero();
        boolean colorDestino = cartaDestino.getColor();
        if ((numeroOrigen == numeroDestino -1) && //una menor y color distinto
            (colorOrigen != colorDestino))
          mover = true;
      }
      else if (numeroOrigen == Carta.K)
        mover = true;
    }

    if (mover)
      moverCartasDescubiertas(aux, pilaDestino, cantidad, true);
    else // no se pueden mover tantas cartas como 'cantidad'
    {
      if (!aux.isEmpty()){
        Carta cartaOrigen = (Carta) aux.peek();
        System.out.println("No se puede mover "+cartaOrigen.getInfo());
        //reponer la pila de origen
        moverCartasDescubiertas(aux, pilaOrigen, numCartas, false);
      }
    }
```

```
  }
  return mover;
}

/**
 *  mientras haya cartas descubiertas contar si hay tantas como cantidad
 */
private int moverCartasDescubiertas(Stack pilaOrigen, Stack pilaDestino, int cantidad, boolean imprimir)
{
  int numCartas = 0;
  while ((!pilaOrigen.isEmpty()) &&
         (((Carta)pilaOrigen.peek()).estaDescubierta()) &&
         (numCartas < cantidad)) {
    Carta carta = (Carta) pilaOrigen.pop();
    if (imprimir)
      System.out.print(carta.getInfo()+"\t");
    pilaDestino.push(carta);
    numCartas++;
  }
  if (imprimir)
    System.out.println("");
  return numCartas;
}

private void descubrirUltimaCarta(Stack pila) {
  if (!pila.isEmpty()) {
    Carta carta = (Carta) pila.peek();
    carta.descubrir();
  }
}
```

3.

```
public static void realizarMovimientos(Solitario juego, String file)
{
  System.out.println("Realizando movimientos...");
  Token2 token = new Token2(file);
  while (token.hasMoreTokens())
  {
    Datos2 datos = (Datos2) token.getTokens();
    String movimiento = datos.getMovimiento();
    if (movimiento.equals("M"))
      juego.mostrarCartas();
    else {
      if (movimiento.equals("BCA"))
        juego.moverDeBarajaACasilla();
      else {
        if (movimiento.equals("BCO"))
          juego.moverDeBarajaAColumna(datos.getColumna1());
        else {
          if (movimiento.equals("CCA"))
            juego.moverDeColumnaACasilla(datos.getColumna1());
          else {
            if (movimiento.equals("CCO"))
              juego.moverDeColumnaAColumna(datos.getColumna1(),
                                datos.getColumna2(), datos.getCantidad());
            else {
              if (movimiento.equals("R"))
                juego.reiniciarBaraja();
            } //end else
          } //end else
        } //end else
      } //end else
```

```
      } //end else
    } //en while
  }
```