Ejercicios de la asignatura Programación II

Grado en Ingeniería de Sistemas de Información

Primer curso

Profesor: Mariano Fernández López

Escuela Politécnica Superior, Universidad San Pablo CEU

8 de mayo de 2015

Índice

1. Métodos recursivos

Ejercicio 1. Programe las siguientes funciones mediante métodos recursivos:

1. El factorial de un número.
2. La suma $0 + 1 + 2 + ... + n$.
3. La potencia n-ésima de un número natural.
4. El elemento n-ésimo de la sucesión de Fibonacci.

Ejercicio 2. ¿En qué consiste el problema de las Torres de Hanoi? ¿Cómo puede resolverse?

2. Prueba de programas

Ejercicio 3. Diseñe pruebas para cada uno los programas del ejercicio 1.

Ejercicio 4. Implemente, utilizando JUnit, las pruebas del ejercicio 3.

Ejercicio 5. Explique los siguientes conceptos:

1. Prueba de caja blanca.
2. Prueba de caja negra.
3. Prueba unitaria.
4. Prueba de integración.
5. Prueba de sistema.
6. Prueba de aceptación.

3. Desarrollo dirigido por pruebas

Ejercicio 6. Realice los ejercicios 1 y 2 utilizando la técnica de desarrollo dirigido por pruebas (TDD).

4. Complejidad de algoritmos

Ejercicio 7. Se tiene el siguiente método:

```
public static int sumaNPrimeros(int n) {
    int suma = 0;
    for (int i = 1; i <= n; i++) {
        suma += i;
    }

    return suma;
}
```

Utilizando el *profiler* de Netbeans se han medido los tiempos de ejecución de diferentes llamadas al método (véase el cuadro 1). Explique los resultados.

Cuadro 1: Tiempos de ejecución del método del ejercicio 7

Llamada	Tiempo de ejecución
`sumaNPrimeros(100)`	0,003 ms
`sumaNPrimeros(1000)`	0,013 ms
`sumaNPrimeros(10000)`	0,131 ms
`sumaNPrimeros(100000)`	1,20 ms
`sumaNPrimeros(1000000)`	3,27 ms
`sumaNPrimeros(10000000)`	6,19 ms
`sumaNPrimeros(100000000)`	36 ms
`sumaNPrimeros(1000000000)`	325 ms

Ejercicio 8. Se tiene el siguiente método:

```
public static int sumaNMPrimeros(int n) {
    int suma = 0;
    for (int i = 1; i <= n; i++) {
        for(int j = 1; j <= i; j++)
        suma += j;
    }

    return suma;
}
```

Utilizando el *profiler* de Netbeans se han medido los tiempos de ejecución de diferentes llamadas al método (véase el cuadro 2). Explique los resultados.

Cuadro 2: Tiempos de ejecución del método del ejercicio 8

Llamada	Tiempo de ejecución
`sumaNPrimeros(100)`	0,085 ms
`sumaNPrimeros(1000)`	2,16 ms
`sumaNPrimeros(10000)`	18,1 ms
`sumaNPrimeros(100000)`	1531 ms

Ejercicio 9. Indique la complejidad asintótica (en términos de O grande) de los siguientes métodos:

1.
```
public static String primero(ArrayList<String> lista)
{
    return lista.get(0);
}
```

2.
```
public static double suma(double a, double b, double c)
{
    return a + b + c;
}
```

3.
```
public static int sumaNPrimeros(int n)
{
    int suma = 0;
    for(int i = 1; i <=n; i++)
    {
        suma+=i;
    }

    return suma;
}
```

4.
```
public static int sumaNMPrimeros(int n) {
    int suma = 0;
    for (int i = 1; i <= n; i++) {
        for(int j = 1; j <= i; j++)
        suma += j;
    }

    return suma;
}
```

5.
```
public static double sumaEltosMatriz(double matriz[][])
{
    double suma = 0;
    for(int i = 0; i < matriz.length; i++)
    {
        for(int j = 0; j < matriz[i].length; j++)
        {
            suma+=matriz[i][j];
        }
    }

    return suma;

}
```

6.
```
public static int buscar(int datoBuscado, int[] datos)
{
    int i = 0;
    boolean encontrado = false;
    while(!encontrado && i < datos.length)
    {
        encontrado = datos[i] == datoBuscado;
        i++;
    }

    return --i;
}
```

7. El siguiente código adaptado de `http://en.wikipedia.org/wiki/Binary_search_algorithm`

```
public static int binary_search(int sortedData[], int key) {
    int KEY_NOT_FOUND = -1;
    int imin = 0;
    int imax = sortedData.length-1;

    while (imax >= imin) {
        int imid = (imin + imax)/2;
        if (sortedData[imid] == key)
        {
            return imid;
        }
        else if (sortedData[imid] < key)
        {
            imin = imid + 1;
        } else
        {
            imax = imid - 1;
        }
    }

    return KEY_NOT_FOUND;
}
```

Ejercicio 10. Calcule la complejidad asintótica de los métodos que ha implementado en el ejercicio 1.

5. Recursividad, desarrollo dirigido por pruebas y complejidad

Ejercicio 11. El siguiente código calcula la suma de los elementos de una pila de *big decimals*:

```
package ejercicios;

import java.math.BigDecimal;
import java.util.Stack;

public class Ejercicio {

    static public BigDecimal sumaIterativaElementosPila(Stack pila)
    {
        BigDecimal suma = BigDecimal.ZERO;
        while(!pila.empty())
        {
            //pila.peek() devuelve la cima de la pila.
            suma = suma.add((BigDecimal) pila.peek());
            pila.pop();
        }
```

```
        return suma;
    }
}
```

Se pretende codificar, de acuerdo con la técnica de desarrollo dirigido por pruebas (TDD), el método *static public BigDecimal sumaRecursivaElementosPila(Stack pila)* de tal forma que realice la misma función que el anterior, pero de manera recursiva.

Se pide:

1. La secuencia de tests y de versiones del método a desarrollar según TDD que culmine con el método implementado que pase todos los tests. En la respuesta del estudiante debe quedar claro qué versión del software hace pasar cada test.

2. El análisis de la complejidad temporal asintótica tanto del método dado en el enunciado como del método que sea la versión final del estudiante.

6. Estructuras de datos

6.1. Pilas

Ejercicio 12. Las operaciones que se muestran a continuación son las típicas de una pila:

- Está vacía: devuelve *verdadero* si la pila está vacía, y *falso* en otro caso.
- Apilar: introduce un elemento como la cima de la pila.
- Desapilar: elimina la cima.
- Cima: devuelve la cima.
- Tamaño: devuelve el número de elementos que tiene la cola.
- Está llena (sólo para las implementaciones donde tenga sentido esta operacion): devuelve *verdadero* si la pila está llena, y *falso* en otro caso.

Se pide, utilizando TDD en la escritura del código, realizar los siguientes apartados:

1. Implementar la estructura anterior utilizando un *array*.

2. Implementar la estructura anterior utilizando una estructura enlazada.

3. Calcule la complejidad de las operaciones implementadas.

Basándose en la estructura anterior, realizar:

4. Un programa que evalúe una expresión en notación posfija.
5. Un programa que evalúe una expresión en notación prefija.
6. Un programa que transforme una expresión en notación infija en otra en notación posfija.
7. Un programa que evalúe una expresión aritmética en notación infija.
8. Basándose en la estructura anterior, realizar un programa que permita transformar un número decimal en número binario.

PISTA: puede inspirarse en `http://jcsites.juniata.edu/faculty/kruse/cs240/stackapps.htm`

¿Qué clase de Java SE implementa la estructura de datos de pila?

6.2. Colas

Ejercicio 13. De acuerdo con la técnica de desarrollo dirigido por pruebas, implemente una estructura de cola con las siguientes operaciones:

- Está vacía: devuelve *verdadero* si la cola está vacía, y *falso* en otro caso.
- Insertar: introduce un elemento al principio de la cola.
- Borrar: elimina un elemento del final de la cola.
- Primero: obtiene el primer elemento de la cola.
- Tamaño: devuelve el número de elementos que tiene la cola.

Para entender esta estructura de datos podemos pensar en la cola de un determinado servicio. Van llegando individuos a la cola (insertar en la cola), y se van atendiendo individuos (eliminar de la cola). La política de esta estructura de datos es FIFO: el primero que entra es el primero que sale, es decir, el primero que llega a la cola es el primero en ser atendido.

¿Qué clase de Java SE implementa la estructura de datos de cola?

6.3. Listas

Ejercicio 14. De acuerdo con la técnica de desarrollo dirigido por pruebas, y utilizando polimorfismo paramétrico, implemente la estructura *ArrayList* de Java, pero sólo con las siguientes operaciones `http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html` (a la clase, llámele *MiArrayList*):

- *isEmpty()*: devuelve *true* si la lista no contiene ningún elemento.
- *add(E e)*: añade el elemento *e* al final de la lista.
- *remove(E e)*: borra la primera aparición del elemento *e* en la lista.
- *get(int index)*: devuelve el elemento que está en la posición indicada por *index*.
- *contains(E e)*: devuelve *true* si el elemento *e* está en la lista.
- *size()*: devuelve el número de elementos de la lista.

Ejercicio 15. Utilizando el *ArrayList* de Java, implemente una libreta de contactos.

Ejercicio 16. De acuerdo con la técnica de desarrollo dirigido por pruebas, implemente una estructura de lista lineal clásica con las siguientes operaciones:

- Está vacía: devuelve *verdadero* si la lista está vacía, y *falso* en otro caso.
- Insertar: introduce un elemento al principio de la lista.
- Modificar: modifica el primer elemento de la lista.
- Borrar: en caso de que esté, elimina un elemento determinado dentro de la lista.
- Primero: obtiene el primer elemento de la lista.
- Resto: devuelve la lista formada por todos los elementos desde segundo (inclusive) en adelante.
- Tamaño: devuelve el número de elementos que tiene la lista.

Implemente métodos que permitan llevar a cabo las siguientes funciones:

1. Sumar los elementos de una lista.
2. Obtener el número de elementos que tiene la lista.
3. Multiplicar cada uno de los elementos de la lista por un número.
4. Sumar dos listas como si fueran dos vectores.

Ejercicio 17. Implemente un método en la clase 14 que obtenga el máximo de todos los elementos.

6.4. Ordenación de estructuras lineales

Ejercicio 18. Implemente los siguientes algoritmos de ordenación en la clase elaborada en el ejercicio 14 y realice su análisis de complejidad temporal asintótico:

1. Inserción directa.
2. Selección directa.
3. Intercambio directo (método de la burbuja).
4. *Mergesort*.
5. *Quicksort*.

Ejercicio 19. Analice y explique el código del *mergesort* que aparece en la siguiente URI: `http://www.vogella.com/tutorials/JavaAlgorithmsMergesort/article.html#mergesort_quicksort`

Ejercicio 20. Responda a las respuestas a las siguientes preguntas:

1. ¿Quién inventó el algoritmo *mergesort*?
2. ¿Quién inventó el algoritmo *quicksort*?

Ejercicio 21. Cree un *ArrayList* de números enteros (utilizando la clase original de Java) y ordénelo mediante el método estático *sort* de la clase *Collections*.

Ejercicio 22. El siguiente código muestra cómo se pueden comparar personas por nombre y por edad:

```
import comparadores.ComparadorLexicografico;
import comparadores.ComparadorPorEdad;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import personas.Persona;

public class DemoComparador {

    //Código inspirado en StackOverflow http://stackoverflow.com/questions/2839137/how-to
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        List<Persona> personas = new ArrayList();
        personas.add(new Persona("Juan", 24));
        personas.add(new Persona("Pedro", 18));
        personas.add(new Persona("Cristina", 21));

        Collections.sort(personas, new ComparadorLexicografico());
```

```
        System.out.println(personas);
        Collections.sort(personas, new ComparadorPorEdad());
        System.out.println(personas);
    }
}
```

El comparador por nombre es el que se muestra a continuación.

```
import java.util.Comparator;
import personas.Persona;

public class ComparadorLexicografico implements Comparator<Persona> {
    @Override
    public int compare(Persona a, Persona b) {
        return a.getNombre().compareToIgnoreCase(b.getNombre());
    }
}
```

Se pide implementar el comparador por edad para que el código completo funcione.

Ejercicio 23. Añada un método a la clase obtenida en el ejercicio 15 para ordenar los contactos según el nombre del contacto.

6.5. Árboles

Ejercicio 24. Defina los términos árbol y árbol binario.

Ejercicio 25. Sea el siguiente código que implementa un árbol binario (la mayor parte del código ha sido tomado de `http://cslibrary.stanford.edu/110/BinaryTrees.html#java`):

```
// BinaryTree.java
public class BinaryTree {

    // Root node pointer. Will be null for an empty tree.
    private Node root;


    /*
     --Node--
     The binary tree is built using this nested node class.
     Each node stores one data element, and has left and right
     sub-tree pointer which may be null.
     The node is a "dumb" nested class -- we just use it for
     storage; it does not have any methods.
     */
    private static class Node {

        Node left;
```

```
    Node right;
    int data;

    Node(int newData) {
        left = null;
        right = null;
        data = newData;
    }
}

/**
 * Creates an empty binary tree -- a null root pointer.
 */
public void BinaryTree() {
    root = null;
}

/**
 * Returns true if the given target is in the binary tree. Uses a recursive
 * helper.
 *
 * @param data
 * @return
 */
public boolean lookup(int data) {
    return (lookup(root, data));
}

/**
 * Recursive lookup -- given a node, recur down searching for the given
 * data.
 */
private boolean lookup(Node node, int data) {
    if (node == null) {
        return (false);
    }

    if (data == node.data) {
        return (true);
    } else if (data < node.data) {
        return (lookup(node.left, data));
    } else {
        return (lookup(node.right, data));
    }
}

/**
 * Inserts the given data into the binary tree. Uses a recursive helper.
 *
 * @param data
 */
public void insert(int data) {
    root = insert(root, data);
}
```

```
/**
 * Recursive insert -- given a node pointer, recur down and insert the given
 * data into the tree. Returns the new node pointer (the standard way to
 * communicate a changed pointer back to the caller).
 */
private Node insert(Node node, int data) {
    if (node == null) {
        node = new Node(data);
    } else {
        if (data <= node.data) {
            node.left = insert(node.left, data);
        } else {
            node.right = insert(node.right, data);
        }
    }

    return (node); // in any case, return the new pointer to the caller
}

public int size() {
    return (size(root));
}

private int size(Node node) {
    if (node == null) {
        return (0);
    } else {
        return (size(node.left) + 1 + size(node.right));
    }
}

/**
 * Returns the max root-to-leaf depth of the tree. Uses a recursive helper
 * that recurs down to find the max depth.
 *
 * @return
 */
public int maxDepth() {
    return (maxDepth(root));
}

private int maxDepth(Node node) {
    if (node == null) {
        return (0);
    } else {
        int lDepth = maxDepth(node.left);
        int rDepth = maxDepth(node.right);

        // use the larger + 1
        return (Math.max(lDepth, rDepth) + 1);
    }
}
```

```
/**
 * Returns the min value in a non-empty binary search tree. Uses a helper
 * method that iterates to the left to find the min value.
 *
 * @return
 */
public int minValue() {
    return (minValue(root));
}

/**
 * Finds the min value in a non-empty binary search tree.
 */
private int minValue(Node node) {
    Node current = node;
    while (current.left != null) {
        current = current.left;
    }

    return (current.data);
}

/**
 * Prints the node values in the "inorder" order. Uses a recursive helper to
 * do the traversal.
 *
 * @return
 */
@Override
public String toString() {
    return aStringBuilder(root).toString();
}

private StringBuilder aStringBuilder(Node node) {
    StringBuilder sb = new StringBuilder("");
    if (node == null) {
        return sb;
    }

    // left, node itself, right
    return aStringBuilder(node.left).
            append(new StringBuilder(" " + node.data + "  ")).
            append(aStringBuilder(node.right));
}

@Override
public boolean equals(Object other) {
    /*
     Compares the receiver to another tree to
     see if they are structurally identical.
     */
    return (sameTree(root, ((BinaryTree) other).root));
}
```

```
    /**
     * Recursive helper -- recurs down two trees in parallel, checking to see if
     * they are identical.
     */
    boolean sameTree(Node a, Node b) {
        // 1. both empty -> true
        if (a == null && b == null) {
            return (true);
        } // 2. both non-empty -> compare them
        else if (a != null && b != null) {
            return (a.data == b.data
                    && sameTree(a.left, b.left)
                    && sameTree(a.right, b.right));
        } // 3. one empty, one not -> false
        else {
            return (false);
        }
    }

    /**
     * Prints the node values in the "postorder" order. Uses a recursive helper
     * to do the traversal.
     */
    public void printPostorder() {
        printPostorder(root);
        System.out.println();
    }

    private void printPostorder(Node node) {
        if (node == null) {
            return;
        }

        // first recur on both subtrees
        printPostorder(node.left);
        printPostorder(node.right);

        // then deal with the node
        System.out.print(node.data + "  ");
    }
}
```

Se pide:

1. Mostrar el resultado del siguiente método y dibujar cómo está estructurado el árbol:

```
    static void mostrarFuncionamientoArbolesBinarios() {
        BinaryTree bt = new BinaryTree();
        bt.insert(2);
        bt.insert(1);
        bt.insert(3);
        bt.insert(8);
```

```
        bt.insert(5);
        bt.insert(2);

        System.out.println(bt);
        bt.printPostorder();
    }
```

2. El análisis de complejidad temporal asintótico de cada uno de los métodos públicos que aparecen en el ejercicio.

Ejercicio 26. Explique qué son los recorridos en orden simétrico, preorden y posorden.

Ejercicio 27. Explique cómo implementaría un árbol en que cada nudo puede tener más de dos hijos.

Ejercicio 28. Averigüe cómo JDOM estructuraría en forma de árbol el siguiente fragmento de HTML:

```
<TABLE>
<TBODY>
<TR>
<TD>Shady Grove</TD>
<TD>Aeolian</TD>
</TR>
<TR>
<TD>Over the River, Charlie</TD>
<TD>Dorian</TD>
</TR>
</TBODY>
</TABLE>
```

6.6. Conjuntos

Ejercicio 29. Con el propósito de posicionarnos en el mundo de los juegos, se ha decidido desarrollar una aplicación para jugar al bingo. Dado que tanto las líneas del cartón, como el propio cartón, como el bombo contienen elementos no repetidos, y no importa el orden entre tales elementos, se ha decido utilizar la estructura de datos *Conjunto*, inspirada en la noción de conjunto de las matemáticas. Se pide implementar, mediante desarrollo dirigido por pruebas, las siguientes operaciones típicas de la estructura *Conjunto*:

- Es vacío: devuelve *verdadero* si el conjunto está vacío, y *falso* en otro caso.
- Pertenece: devuelve *verdadero* si el elemento que se pasa como parámetro pertenece al conjunto, y *falso* en otro caso.
- Insertar: introduce en el conjunto el elemento que se pasa como parámetro. Si el elemento ya pertenece, la operación no tiene efecto ninguno.

- Borrar: elimina del conjunto el elemento que se pasa como parámetro. Si el elemento no pertenece al conjunto, la operación no tiene efecto ninguno. En la implementación de este método, téngase en cuenta que al borrar un elemento, se deben desplazar hacia la izquierda todos los elementos que estén después del elemento borrado en el *array*.
- Cardinal: obtiene el número de elementos del conjunto.
- Union: obtiene un conjunto con los elementos del conjunto que recibe el mensaje además de los elementos del conjunto que se pasa como parámetro.
- Intersección: obtiene un conjunto con los elementos comunes al conjunto que recibe el mensaje y al conjunto que se pasa como parámetro.
- Diferencia: obtiene un cojunto con los elementos pertenecientes al conjunto que recibe el mensaje que no pertenecen al conjunto que se pasa como parámetro.

Ejercicio 30. Realice los siguientes apartados:

1. Implemente un programa en Java que añada varios números enteros en un *TreeSet* y luego los muestre por pantalla. ¿Qué ocurre con el orden de los enteros? ¿Qué ocurre con los elementos repetidos?
2. Explique la diferencia de complejidad entre buscar en un *ArrayList* desordenado y buscar en un *TreeSet*.
3. Para implementar la aplicación del bingo mencionada en el ejercicio 29, ¿es preferible la estructura propuesta en dicho ejercicio o el *TreeSet*?. ¿por qué?

6.7. Tablas

Ejercicio 31. Realice los siguientes apartados:

1. Defina el término *tabla*.
2. ¿Cuál es la complejidad de buscar en un *HashMap* homogéneo?
3. ¿Cómo puede saber, para una aplicación particular, si es mejor utilizar la clase *HashMap* o *TreeMap*?
4. ¿Por qué el número de caracteres no es una función hash adecuada para cadenas de caracteres?
5. Explique la función hash que utiliza Java para Strings y por qué se utiliza el 31 como multiplicador.

6. Explique la función hash de Java para enteros.

7. Si se trabajara con un ordenador de bajas prestaciones, ¿cómo se debería modificar la función hash para enteros?

6.8. Grafos

Ejercicio 32. Realice los siguientes apartados:

1. Defina los términos: grafo, grafo dirigido y grafo con pesos.

2. Explique diferentes implementaciones que se pueden utilizar para la estructura de grafo.

3. Bájese la bilbioteca *JGraphT* y cree un grafo con pesos no dirigido con los vértices: "Albacete", "Almería", "Madrid", "Murcia", "Segovia" y "Toledo" y los arcos siguientes:

   - Madrid-Segovia: 92 km
   - Segovia-Toledo: 165 km
   - Madrid-Toledo: 72 km
   - Madrid-Albacete: 257 km
   - Albacete-Toledo: 245 km
   - Albacete-Murcia: 146 km
   - Murcia-Almería: 218 km
   - Toledo-Almería: 500 km

4. Ejecute el algoritmo de Dijkstra disponible en la biblioteca para obtener el camino más corto para ir de Madrid a Almería.

5. Ejecute usted en papel el algoritmo para comprobar el resultado.

6.9. Iteradores

Ejercicio 33. Realice los siguientes apartados:

1. ¿Por qué se necesitan objetos iteradores para algunas colecciones?

2. Cree una colección cualquiera y recórrala utilizando un iterador.

6.10. El *framework* de Java para estructuras de datos

Ejercicio 34. Realice los apartados que se muestran an continuación sobre el *framework* de Java:

1. Descríbalo.

2. ¿Por qué es útil hacer referencia en nuestros programas a interfaces como, por ejemplo, *List*?