Estructuras de Datos y Algoritmos I
Grado en Ingeniería Informática, Curso 2º

ACTIVIDADES OBLIGATORIAS (Examen Septiembre 2014)

Hay que desarrollar todas las actividades propuestas (con todos sus ejercicios) en el presente documento para poder presentarse al examen de Septiembre 2014 (tal y como indica la guía docente de la asignatura **Estructuras de Datos y Algoritmos I**). Cada actividad tiene asociado un test donde se validan todos los ejercicios planteados. Además, se proporcionará un nuevo `allTest.java` con todo lo anterior y lo nuevo, que se debe pasar adecuadamente (es decir, todos los test deben estar en ***verde***).

Actividad 1. En teoría hemos estudiado los *heap* (montículos) máximos y mínimos, y en prácticas hemos hecho uso de dicha implementación en la clase `class Heap<T>`. En dicha implementación hemos utilizado un `ArrayList<T> theHeap`, que tiene la raíz en la posición 0. Para obtener un heap mínimo de enteros (`Integer`) hemos declarado un *Comparator* de la siguiente forma `Less<Integer> less = new Less<Integer>()`, y dicho ***heap mínimo*** lo hemos declarado de la siguiente manera `Heap<Integer> heap = new Heap<Integer>(less)`. En esta actividad se pide implementar en Java los dos siguientes iteradores sobre dicho *heap*.

- **`public Iterator<T> breadthFirstIterator()`** sobre un *heap* mínimo, que ***realiza*** un recorrido en anchura (de arriba abajo, y de izquierda a derecha) sobre el *heap*, y para ello utiliza la clase `private class HeapBreadthFirstIterator implements Iterator<T>`.
- **`public Iterator<T> depthFirstIterator()`** sobre un *heap* mínimo, que ***realiza*** un recorrido en profundidad (siguiendo el esquema izquierdo-raíz-derecho) sobre el *heap*, y para ello utiliza la clase `private class HeapDepthFirstIterator implements Iterator<T>`.

Una vez implementado todo debe comprobar su correcto funcionamiento con el **test** correspondiente.

Actividad 2. Dada la estructura de datos `BSTree<T>` estudiada en clase e implementada en prácticas, que se corresponde con un ABB (árbol binario de búsqueda) que no contiene duplicados. Se pide, implementar en Java las siguientes funciones. Para el caso que sea necesario, se deben de implementar tanto la función pública como la privada.

- Implementar constructor copia para el BSTree **`public BSTree(BSTree<T> otherTree)`**, haciendo uso de una función privada `private BSTNode<T> copy(BSTNode<T> p)`. Comprobar que la copia es igual al original, haciendo uso del método `public boolean equals(Object obj)`.
- Implementar una función que elimine todos los elementos de un ABB menores que uno dado ***x***. **`public void removeSmaller(T x)`**.
- Implementar una función que elimine todos los nodos hoja del ABB. **`public void removeLeaves()`**.

- Implementar una función que obtenga la suma de todos los elementos que sean mayores o iguales a un valor entero dado ***x*** en un ABB. **`public int sumOfSmallerThanOrEqualTo(T x)`**.
- Sea un par de datos enteros ***x*** e ***y*** que definen un intervalo no vacío ***[x, y]***. Implementar una función que devuelva el número de elementos que no pertenecen a dicho intervalo en un ABB. **`public int countOutOfRange(T x, T y)`**.
- Implementar una función que muestre todos los nodos de un ABB que estén entre dos niveles dados, ***level1*** y ***level2***, ambos inclusive. **`public int countNodesBetweenLevels(int level1, int level2)`**.
- Implementar una función que devuelva el recorrido en anchura inverso, **`public String toStringReverseBreadthFirstTraversal()`**. Es decir, que devuelva los elementos del ABB de abajo-a-arriba (*bottom-up*) y de derecha-a-izquierda (*right-to-left*).

Una vez implementado todo debe comprobar su correcto funcionamiento con el **test** correspondiente.

Actividad 3. Hemos estudiado en teoría las tablas hash, y se ha proporcionado, en los archivos `HashTableQuadraticProbing.java` y su correspondiente `Program.java`, la implementación de una tabla hash para que se conozca su uso y su utilidad, y que utiliza la resolución de colisiones formando una secuencia de posiciones aplicando el método de ***prueba cuadrática*** (*quadratic probing*).

La resolución de colisiones se realizará formando una secuencia de posiciones aplicando el método de ***prueba según un exponente*** (*exponent probing*) caracterizada por la ecuación $\mathbf{p_k(x) = ((H(x) + k^{exp}) \bmod max)}$ y que dependiendo de valor del exponente la resolución de colisiones será prueba lineal (exponente = 1), prueba cuadrática (exponente = 2), prueba cúbica (exponente = 3), etc. La clase `HashTableExponentProbing<T>` constará de un array (`table`) tal y como se indica a continuación en la correspondiente declaración. Se ha tomado la decisión de que los elementos eliminados permanezcan en la tabla (para no perder información histórica), y por esta razón se añadirá el atributo `isActive` en la clase `HashEntry<T>`, que si está activo (`true`) indicará que se ha insertado en la tabla; mientras que si tiene un valor a `false` (inactivo) indicará que se ha eliminado dicho elemento. El número de elementos que hay en la tabla, incluyendo los eliminados, se almacena en la variable `hashTableSize`. Además, se añadirá la variable `tableThreshold` (asociado al concepto de factor de carga) de tal forma que cuando se alcance el **0.5** se pueda realizar un redimensionado de la tabla hash (`rehash()`) con el doble de elementos que tenga en ese momento.

Además, sabemos que la ***prueba cuadrática*** suele producir mejores resultados que la *prueba lineal* (*linear probing*), ya que la prueba cuadrática no tiene una tendencia tan grande a crear agrupaciones como le ocurre al método de prueba lineal. No obstante si no se elige convenientemente el tamaño de la tabla, no se puede asegurar que se prueben todas las posiciones de la tabla. Se puede demostrar que si el tamaño de la tabla es un número primo y el factor de carga no alcanza el 50%, todas las pruebas que se realicen con la secuencia $\mathbf{p_k(x) = ((H(x) + k^2) \bmod max)}$ se hacen sobre posiciones de la tabla distintas y siempre se podrá añadir.

En esta actividad se pide implementar una **tabla hash** para almacenar un conjunto de elementos genéricos (T), tal y como se especifica en la siguiente estructura de clase.

```
public class HashTableExponentProbing<T> {
  private static final int DEFAULT_TABLE_SIZE = 71;
  private final double MAX_LOAD_FACTOR = .5;

  private HashEntry<T> [] table;        // The table of elements
  private int occupiedEntries;          // The number of occupied entries
  private int hashTableSize;            // The current size
  private int tableThreshold;
  private int numberOfCollisions;
  private int exponent;

  public HashTableExponentProbing () {
    this(DEFAULT_TABLE_SIZE);
  }

  public HashTableExponentProbing (int exp) {
    allocateTable(DEFAULT_TABLE_SIZE);
    occupiedEntries = 0;
    hashTableSize = 0;
    tableThreshold = (int)(table.length * MAX_LOAD_FACTOR);
    numberOfCollisions = 0;
    exponent = exp;
    for (int i = 0; i < table.length; i++)
      table[i] = null;
  }

  private void rehash(int newTableSize) { … }
  private int findPos(T x) { … }
  public boolean add(T x) { … }
  public boolean remove(T x) { … }
  public int size() { return hashTableSize; }
  public int capacity() { return table.length; }
  public int numberOfOccupiedEntries() { return occupiedEntries; }
  public int getNumberOfCollisions() { return numberOfCollisions; }
  public boolean contains(T x) { … }
  private boolean isActive(int currentPos) { … }
  public void clear() { … }
  public boolean isEmpty() { return hashTableSize == 0; }
```

```
  public Object[] toArray() { … }
  public String toString() { … }
  private int myHash(T x) {
    int hashVal = (x.hashCode() & Integer.MAX_VALUE) % table.length;
    if (hashVal < 0)
      hashVal += table.length;
    return hashVal;
  }
  private void allocateTable(int tableSize)
    { table = new HashEntry[nextPrime(tableSize)]; }
  private static int nextPrime(int n) {
    if (n % 2 == 0)
      n++;
    for(; !isPrime(n); n += 2)
      ;
    return n;
  }
  private static boolean isPrime(int n) {
    if (n == 2 || n == 3)
      return true;
    if (n == 1 || n % 2 == 0)
      return false;
    for (int i = 3; i * i <= n; i += 2)
      if (n % i == 0)
        return false;
    return true;
  }
  private static class HashEntry<T> {
    public T element;           // The element
    public boolean isActive;    // true is marked added, false if marked deleted
    public HashEntry(T e) {
      this(e, true);
    }
    public HashEntry(T e, boolean i) {
      element  = e;
      isActive = i;
    }
  }
}
```

Una vez implementada la clase debe comprobar su correcto funcionamiento con el **test** correspondiente. Explique de forma razonada toda la implementación de código realizada.

Actividad 4.

Como ya sabemos muchas palabras son similares a otras a nivel letras. Por ejemplo, en inglés, cambiando la primera letra, la palabra **wine** se convierte en ***dine***, ***fine***, ***line***, ***mine***, ***nine***, ***pine*** o ***vine***. Cambiando la tercera letra, **wine** se puede convertir en ***wide***, ***wife***, ***wipe*** o ***wire***, entre otras. Cambiando la cuarta letra, **wine** puede convertirse en ***wind***, ***wing***, ***wink*** o ***wins***, entre otras. Como conclusión, podemos observar que se pueden obtener 15 palabras diferentes cambiando sólo una letra en la palabra **wine** (aunque en realidad hay unas 20 palabras diferentes, otras no consideradas aquí no son tan conocidas como las anteriores). Como objetivo final de esta actividad se desea implementar un programa lo más eficaz posible que encuentre todas las palabras que se pueden transformar en al menos otras 15 palabras sustituyendo un único carácter (letra). Suponemos que tenemos el diccionario en inglés, *dictionary.txt*, que tiene unas 58110 palabras diferentes y de longitud variable.

Una primera solución sería utilizar un **`Map`** en el que las claves (*keys*) son palabras y los valores (*values*) son listas que contienen las palabras que pueden transformarse a partir de la clave por la sustitución de un único carácter. Obviamente se necesita una función para comprobar si dos palabras son idénticas excepto por un carácter. En esta primera aproximación podemos implementar el algoritmo más sencillo para la construcción del **`Map`**, que es comprobar por fuerza bruta (brute-force) todos los posibles pares de palabras. Además, como ya sabemos para moverse (iterar) por una colección de palabras, podríamos utilizar un iterador, de no ser porque nos moveríamos en él con un bucle anidado (varias veces), y por ello copiamos la colección en un array mediante **toArray**. Con esto, entre otras cosas, evitamos repetidas llamadas para convertir el objeto a `String`. En su lugar, simplemente utilizaremos un `String []`.

```
// Devuelve true si word1 y word2 tienen la misma longitud y difieren en un único carácter
private static boolean oneCharOff(String word1, String word2)
// Actualiza el mapa
private static <KeyType> void update(Map<KeyType,List<String>> m, KeyType key,
  String value)
// Calcula un mapa en el que las claves son palabras y los valores listas de palabras que difieren en
// un único carácter de la clave correspondiente. Utiliza un algoritmo cuadrático de fuerza bruta
public static Map<String,List<String>> computeSimilarWordsBruteForce(
  List<String> theWords)
```

El problema con el algoritmo anterior es que es muy lento. Una mejora podría ser evitar la comparación de palabras de diferentes longitudes. Podemos hacer esto mediante la agrupación de las palabras por su longitud, y luego ejecutar el algoritmo anterior sobre cada uno de los grupos separados. Para ello, podemos utilizar un segundo **`Map`**. En este caso, la clave es un entero que representa una longitud de palabra, y el valor es una colección de todas las palabras de esa longitud (podemos utilizar una `List` para almacenar cada colección). En comparación con el primer algoritmo, este segundo algoritmo mejorado es un poco más difícil de implementar, pero bastante más rápido.

```
// Calcula un mapa en el que las claves son palabras y los valores listas de palabras que difieren en
// un único caracter de la clave correspondiente. Utiliza un algoritmo cuadrático, pero mejora y
// hace más rápido al anterior con un mapa adicional y agrupando palabras por su longitud
public static Map<String,List<String>> computeSimilarWordsImproved(
  List<String> theWords)
```

El tercer algoritmo es un poco más complejo, y utiliza mapas adicionales. Como antes, agrupamos las palabras por longitud de palabra, y luego trabajamos sobre cada grupo por separado. Para ver cómo funciona este algoritmo, supongamos que estamos trabajando en palabras de longitud 4. Entonces primero, queremos encontrar pares de palabras como **wine** y **nine** que son idénticas a excepción de la primera letra. Una forma de hacer esto, por cada palabra de longitud 4, es eliminar el primer carácter, dejando una *palabra representativa* de tres caracteres. Crear un **`Map`** en el que la clave es esa *palabra representativa*, y el valor es una **`List`** de todas las palabras que tienen esa *palabra representativa*. Por ejemplo, al examinar el primer carácter del grupo de palabras de cuatro letras, la *palabra representativa* **ine** corresponde a ***dine***, ***fine***, ***wine***, ***nine***, ***mine***, ***vine***, ***pine*** y ***line***. La *palabra representativa* **oot** corresponde a ***boot***, ***foot***, ***hoot***, ***loot*** y ***soot***. Cada **`List`** individual que es un valor en este último **`Map`** se forma un *clique* de palabras en el que cualquier palabra se puede cambiar a cualquier otra palabra por una sustitución de un carácter. Así que después de que se construye este último **`Map`**, es fácil de recorrerlo y agregar entradas al **`Map`** original que se está calculando. A continuación, se procedería con el segundo carácter del grupo de palabras de cuatro letras, con un nuevo **`Map`**. Y a continuación, el tercer carácter, y finalmente el cuarto carácter. Un esquema general del algoritmo sería el siguiente

```
for each grupo g, conteniendo palabras de longitud len
  for each posición p (en el rango desde 0 a len-1)
  {
    Crear un Map<String, List<String> > repsToWords vacío
    for each palabra w
    {
      Obtener la palabra representativa de w eliminando la posición p
      Actualizar repsToWords
    }
    Utilizar cliques en repsToWords para actualizar el mapa simWords
  }
```

```
// Calcula un mapa en el que las claves son palabras y los valores listas de palabras que difieren en
// un único caracter de la clave correspondiente. Utiliza un algoritmo eficiente que es O(N log N),
// con un TreeMap o O(N) si lo que se utiliza es un HashMap
public static Map<String,List<String>> computeSimilarWords(List<String> words)
```

Debemos aclarar que un *clique* es un subgrafo en que cada vértice está conectado a cada otro vértice del grafo, es decir, tendremos una relación de todos con todos.

Adicionalmente, una vez obtenido el mapa con las palabras más similares (cuya similitud es que varíe en un único carácter) se deberán implementar funciones adicionales que devuelvan el resultado requerido

```
// Dado el mapa que contiene palabras como clave y como valor una lista de palabras que difieren
// en un único carácter, devolver un mapa con las palabras que tienen minNumberOfSimilarWords
// o más palabras obtenidas por la sustitución de un carácter en ella
public static Map<String, Integer> findHighChangeables(Map<String,List<String>>
   similarWords, int minNumberOfSimilarWords)

// Encontrar la palabra más intercambiable, que es aquella palabra que difiere en sólo un carácter
// con la mayoría de las palabras. Devolvería una lista de esas palabras en caso de empate
public static List<String> findMostChangeable(Map<String,List<String>>
   similarWords)
```

Una vez implementado todo debe comprobar su correcto funcionamiento con el **test** correspondiente.

Actividad 5. En esta actividad vamos a complementar lo estudiado en las clases de teoría y prácticas, relativo a grafos. Dado un grafo orientado/no orientado valorado positivamente, implementado en Java utilizando un mapa de adyacencia en la `class Network<Vertex>`, tal y como se ha estudiado en clase. Implementar en Java la serie de funciones que se indican a continuación, dado un grafo orientado y declarado como `Network<String> net = new Network<String>()`, obtengan los resultados demandados. Se pueden añadir las variables globales que considere oportunas y necesarias a la clase para la correcta implementación de las funciones planteadas, todo debidamente justificado.

Es decir, para ello se deben implementar las siguientes funciones:

1) En la actividad 01, vista a principio de curso se estudiaba cómo realizar la persistencia de estructuras de datos mediante la serialización. En este primer ejercicio vamos a serializar el grafo con el que estamos trabajando en este curso, y para ello se deberá implementar los métodos **`readObject`** y **`writeObject`** en la clase `Network<Vertex>`.
2) Implementar en Java una función que, utilizando el algoritmo de *caminos simples* (estudiado e implementado en teoría y en prácticas), obtenga todos los caminos entre dos vértices del grafo (*fuente* y *destino*) y que pasen por un vértice *intermedio* que será proporcionado como parámetro. `public ArrayList<ArrayList<Vertex>> somePathWithSimplePaths(Vertex source, Vertex destination, Vertex intermediate)`, con su correspondiente función privada `private void somePathWithSimplePathsAux(Vertex current, Vertex destination, Vertex intermediate)`.

3) Hemos estudiado e implementado el algoritmo de *Dijkstra*, partiendo desde un vértice *fuente*, `public ArrayList<Object> Dijkstra(Vertex source, Vertex destination)`, sobre un grafo orientado/no orientado, conexo y valorado positivamente, utilizando mapas (`TreeMap`) y conjuntos (`TreeSet`). Se pide implementar en Java dicho algoritmo, y modificarlo para que además de calcular y mostrar los caminos mínimos entre un vértice *fuente* y el resto de vértices del grafo, también calcule y muestre otro `TreeMap<Vertex, Boolean> uniqueSortestPath` de valores booleanos (`true` o `false`), que indique para cada vértice si su camino mínimo desde el vértice *fuente* es único (`true`) o no (`false`). A esta nueva función la denominaremos **`public String modifiedDijkstraForUniqueSP(Vertex source)`**.
4) Hemos estudiado e implementado el algoritmo de *Dijkstra*, partiendo desde un vértice *fuente*, `public ArrayList<Object> Dijkstra(Vertex source, Vertex destination)`, sobre un grafo orientado/no orientado, conexo y valorado positivamente, utilizando mapas (`TreeMap`) y conjuntos (`TreeSet`). En este ejercicio se pide la implementación del algoritmo de *Dijkstra* utilizando una *cola de prioridad* (`PriorityQueue<VertexWeightPair> pQ`) en lugar de un `TreeSet` (`V_minus_S`) para determinar los vértices que se van procesando, `public ArrayList<Object>` **`DijkstraPQ`**`(Vertex source, Vertex destination)`. Como buena aproximación para acometer esta función es recomendable estudiar y comprobar el correcto funcionamiento de la función `public ArrayList<Object> PrimPQ(Vertex source)`. Nótese que hay que definir `protected class VertexWeightPair implements Comparable<VertexWeightPair> { Vertex vertex; double weight; // … }`.
5) Hemos estudiado tanto teoría como en prácticas el algoritmo de *Floyd* para obtener el camino mínimo entre todos los posibles pares de vértices del grafo. En este contexto, implementar una variante del algoritmo de Floyd que devuelva todos los caminos mínimos que hay partiendo de un vértice *fuente*, pasando por un vértice *intermedio*, hasta alcanzar cualquier otro vértice *destino* del grafo, diferentes de *fuente* e intermedio. `public ArrayList<ArrayList<Object>>` **`SomePathsWithFloyd`**`(Vertex source, Vertex intermediate)`