Programación II
Trabajo Práctico Nro.2
01/06/2017
Entrega: 13/06/2017

La biblioteca y la carrera de biología de la UNGS nos solicitó ayuda para encontrar más rápido sus libros. Cada libro puede ser identificado por su ISBN.[1] (A efectos prácticos, un código de 13 dígitos).
Se desea, por tanto, una estructura de tipo diccionario que pueda asociar a cada ISBN su título correspondiente.

No obstante, se requiere una funcionalidad adicional: poder buscar, dado un prefijo, todas los valores cuya clave empieza por ese prefijo. Por ejemplo, si se tiene:

- ISBN "9785267006323" → "Cien años de soledad"
- ISBN "9788490093795" → "El amor de tu vida"
- ISBN "9785423113601" → "El amor en los tiempos del cólera"

Al hacer una búsqueda por el prefijo "9785", se debería retornar *Cien años de soledad* y *El amor en los tiempos del cólera,* pues ambos empiezan por 9785.

La solución que le ofreció Programación II fue el de una estructura *Trie,*[2] ya que permite una implementación eficiente de prefijos y, a la par, puede actuar como diccionario.

Tareas

Se pide:

- Implementación completa del TAD **TrieChar** con los siguientes *requisitos de complejidad:*
  - Insertar un significado debe ser O(|c|) donde c es la clave.
    Se asume el tamaño del alfabeto y del significado son acotados
  - Buscar(c) debe ser O(|c|*s) donde s es la cantidad de significados con prefijo c

- Implementación de dos alfabetos adicionales:
  - *Para la biblioteca:* ***Palabras***, donde los símbolos son las letras de la 'a' a la 'z' en minúsculas

[1] http://es.wikipedia.org/wiki/ISBN

- *Para la carrera de biología: AlfabetoADN, donde los símbolos son las cuatro posibles bases del ADN. Usar un enum para las bases:*
  public enum Base { A, C, G, T };

TrieChar<T>

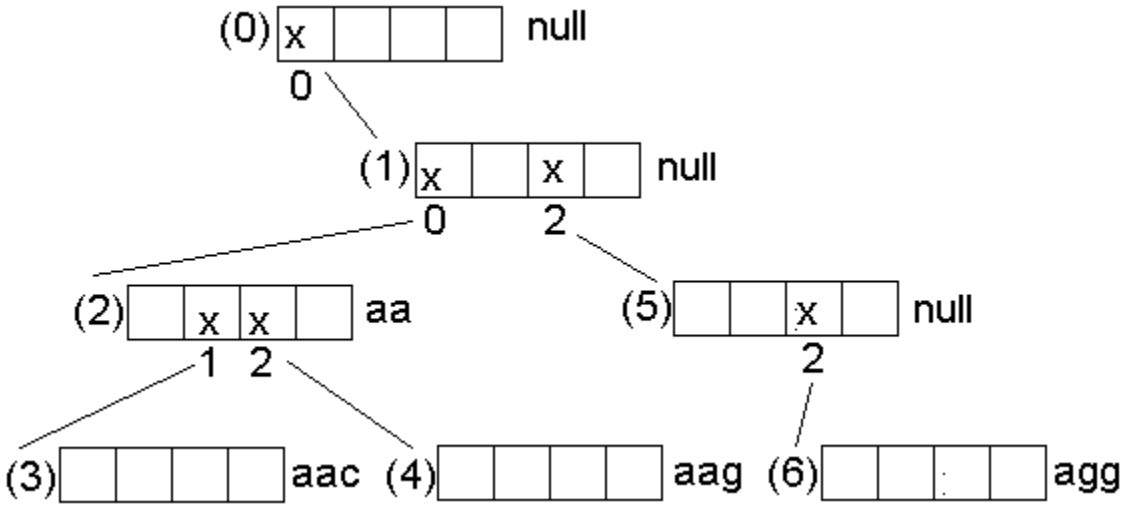
La clase *Trie* recibe en su constructor un objeto de tipo Alfabeto.
Las claves son siempre de tipo *String,* y el constructor recibe un **Alfabeto<Character>**.
Sí se debe, no obstante, parametrizar el valor que almacena el diccionario (como significado del diccionario que representa).

Ejemplo: para almacenar los títulos de los libros, se usaría *TrieChar<String>*

Método Trie.equals()

Dos Tries t1, t2 son iguales si para toda clave c1 de t1:
t1.significado(c1).equals(t2.significado(c1))

Como el Trie tiene que tener la capacidad de almacenar palabras con el mismo prefijo, cada nodo debe tener la posibilidad de almacenar la palabra (el significado del diccionario) o null en caso de que no tenga significado en ese nodo.



Trie con 7 nodos para el alfabeto **{a,c,g,t}**
que almacena las palabras {aa}, {aag}, {aac} y {agg}

Para este ejemplo insertaremos claves y significado iguales:

Agregar(“aa”,”aa”)
Agregar(“aac”,”aac”)
Agregar(“aag”,”aag”)
Agregar(“agg”,”agg”)

Agregamos “aa” en los nodos (0), (1) y (2). Notar que los nodos (0) y (1) son null porque no existe “a” como significado.
Agregamos “aac” en los nodos (0), (1), (2) y (3), y se almacena “aac” en el nodo (3).
Luego agregamos “aag” y “agg” respectivamente.
Las posiciones que no tienen “X” tienen null.

Notar que árbol no almacena las letras en sí, pero le pregunta a su alfabeto que índice utilizar para cada letra.
Por ejemplo, la letra “g” tiene asignado el índice 2.

El trie es una estructura de tipo árbol:

```
public class Trie {
        private Nodo raiz; …
}
```

donde los nodos albergan un número de hijos variable:

```
class Nodo {
        private Nodo []hijos;
}
```

La particularidad del *Trie* es la siguiente: cada nivel representa el carácter iesimo de la clave:

- El número de hijos de los nodos depende de cuántos símbolos posibles tenga el alfabeto (ver ejemplos abajo)
- El 1er carácter de la clave se usa para decidir por dónde realizar la búsqueda para pasar al siguiente nivel.
- En el siguiente nivel, se usa el segundo carácter de la clave, y así sucesivamente.
- A cada posible símbolo se le asigna un *índice;* dicho índice se usa para elegir en tiempo constante el hijo por el cual proseguir.

Ejemplos de Tries:

- En el caso de ISBN, como las claves son dígitos, cada nodo puede albergar hasta un máximo de 10 hijos, correspondientes a los posibles dígitos del ISBN.

- Un uso distinto sería utilizar un Trie para almacenar, por ejemplo, secuencias de ADN. En ese caso, serían 4 hijos solamente (correspondientes a las bases *A, C, G, T*).

- Un tercer caso podría ser almacenar un diccionario de palabras, en cuyo caso habría 26 hijos (en el caso del inglés) o 27 (en el caso del español).

Interfaz: Alfabeto

Para contemplar todos estos casos, se introduce la abstracción de **alfabeto.** Un alfabeto indicará al trie:

- El número de símbolos en el alfabeto.
- Dado un símbolo *S*, a qué índice corresponde.

Así, la clase Trie no tiene que preocuparse por conocer los símbolos exactos, y delega al *Alfabeto* esa responsabilidad.

Siguiendo con los ejemplos anteriores:

- En el caso de ISBN, la correspondencia carácter-índice es simplemente el valor numérico del dígito: '7' → 7.
- En el caso de las bases de ADN, se puede decidir: 'A' → 0, 'C' → 1, 'G' → 2, 'T' → 3
- En el caso del inglés, se seguiría el orden del abecedario; por ejemplo 'd' → 3, 'p' → 15. En español, por ejemplo, habría que decidir qué índice asignar a la *ñ*.

El único requisito que Trie impone a Alfabeto es que, para poder decidir qué hijo usar en tiempo constante, la consulta de la correspondencia símbolo → **índice sea O(1).**

En el código de apoyo se incluye una implementación del alfabeto *Digitos*, que se puede usar para implementar el trie de ISBN.

Alfabeto<T>

Podría ocurrir más adelante que queramos implementar un trie cuyos símbolos no sean caracteres. Por eso, desde ya introducimos un tipo paramétrico en la interfaz: **Alfabeto<T>,** el tipo del símbolo. Así, la interfaz queda:

```
public interface Alfabeto<T> {
        int tam();
        int indice(T elem);  // O(1). Devuelve valores de 0 a tam – 1.
        … // métodos adicionales
}
```

El alfabeto *Digitos*, entonces, es un alfabeto sobre *char:*

```
public class Digitos implements Alfabeto<Character> …
```

Código de apoyo

Como parte de la consigna se proporciona:

- La definición de la interfaz *Alfabeto*
- Un alfabeto de ejemplo, *Digitos*
- La clase Nodo<V>
- Un esqueleto de la clase TrieChar<V>
- Una clase de test

Todo el código debe estar en el paquete **prog2.tp2_2017a** y respetar las interfaces proporcionadas.

Test (ver archivo completo *TestTrie.java*):

```
      public void test() {
            TrieChar<String> libros = new TrieChar<>(new Digitos());
            libros.agregar("9785267006323", "Cien años de soledad");
            libros.agregar("9788490093795", "El amor de tu vida");
            libros.agregar("9785423113601", "El amor en los tiempos del
cólera");

            assertEquals("El amor de tu vida", libros.obtener("9788490093795"));

            List<String> busq = libros.busqueda("9785");
            assertEquals(2, busq.size());
            assertTrue(busq.contains("Cien años de soledad"));
            assertTrue(busq.contains("El amor en los tiempos del cólera"));
        }
```

Para aprobar el TP:
Se pueden armar grupos de hasta dos personas. Pueden ser los mismos que para el TP1 o pueden armar grupos nuevos.
La entrega se debe realizar conforme a las siguientes instrucciones:
https://ungs-prog2.github.io/tps/entrega/

El TP deberá ser entregado por mail antes del **13/06/2017**
El informe, deberá contener:
-El test funcionando, con el ejemplo del enunciado y otro test generado por el grupo para el alfabeto ADN.
- El código de agregar(), obtener() búsqueda() y (**equals y toString** de la clase Trie).

-Un informe de dos páginas explicando porque se cumple la complejidad de agregar() / **buscar**().

Apéndice I: Resumen del código proporcionado por la cátedra (ver archivo ZIP)

```
public class TrieChar<V> {
    private Nodo<V> raiz;
    private Alfabeto<Character> alf;

    public TrieChar(Alfabeto<Character> alf) { this.alf = alf; }
    /**
     * Agrega una cadena a la estructura, asociándole un determinado
     * valor.
     * Si la clave ya existía, se reemplaza su valor asociado.
     */
    public void agregar(String clave, V valor) {
    }

    /**
     * Devuelve el valor asociado a una clave, o null si no existe.
     */
    public V obtener(String clave) {
        return null;
    }

    /**
     * Devuelve una lista con todos los valores cuyas claves
     * empiezan por un determinado prefijo.
     */
    public List<V> busqueda(String prefijo) {
        return null;
    }
}

public interface Alfabeto<T> {
    /**
     * Devuelve k: número de símbolos en el alfabeto.
     */
    int tam();

    /**
     * Devuelve el índice correspondiente a un símbolo.
     *
     * Debe tener complejidad O(1). Lanza RuntimeException
     * si el símbolo no es válido.
     */
    int indice(T elem);
}
```

```
/**
* Nodo de un trie.
*
* El número de hijos se decide en el constructor.
*/
class Nodo<V>
{
    V val;
    private Nodo<V> hijos[];

    Nodo(int tam) {
        hijos = new Nodo[tam];
    }

    Nodo<V> hijo(int i) {
        return hijos[i];
    }

    void setHijo(int i, Nodo<V> hijo) {
        hijos[i] = hijo;
    }
}


/**
* Alfabeto de ejemplo.
*/
public class Digitos implements Alfabeto<Character>
{
    @Override
    public int tam() { return 10; }

    @Override
    public int indice(Character c) {
        // Se implementa basándose en el valor ASCII de char.
        // Los caracteres '0' a '9' son adyacentes en la tabla
        // ASCII, por lo que sus valores son contiguos.
        if (c >= '0' && c <= '9')
            return c - '0';

        throw new RuntimeException("digito no válido: " + c);
    }
}
```