Sala común asignatura Estructura de Datos (2022-23)

Enunciado del examen

Importante Al final del examen deberás subir solamente los siguientes ficheros **no comprimidos**: `DictionaryStringTrie.hs` (que está en la carpeta DataStructures/Trie) y `DictionaryStringTrie.java` (que está en la carpeta dataStructures/trie). Asegúrate de que los ficheros subidos realmente corresponden con tus soluciones, ambos incluyen tu nombre y DNI en el comentario inicial y compilan sin error. Solo se corregirán ficheros sin errores de compilación. Además de la corrección del programa, se tendrán en cuenta la claridad, simplicidad y eficiencia de las soluciones.

Tries

Un Trie, o árbol de prefijos, es una estructura de datos que permite almacenar secuencias de elementos (las claves) y un valor asociado a cada una de las secuencias (los valores). Aunque las secuencias pueden ser de cualquier tipo de elementos, en el resto del enunciado consideraremos solamente Tries que almacenan secuencias de caracteres (Strings). Por ejemplo, podríamos usar un Trie para almacenar palabras en un texto de forma que cada una tenga asociada un valor entero que fuese el número de repeticiones de dicha palabra.

Un Trie es un árbol general, es decir, cada nodo del árbol puede tener un número de hijos `n` diferente ($n \geq 0$). Para almacenar eficientemente los hijos, cada nodo incluye un diccionario cuyas claves serán caracteres y cuyos valores serán los nodos hijos. Además del diccionario de hijos, cada nodo puede almacenar un valor o no almacenarlo. Si almacena un valor diremos que es un nodo final (marca el final de una cadena) y el valor almacenado será el asociado con dicha cadena.

La siguiente figura muestra un Trie con 12 nodos que almacena las siguientes cadenas y valores asociados:

Cadena	Valor
bat	0
be	1
bed	2
cat	3
to	4
toe	5

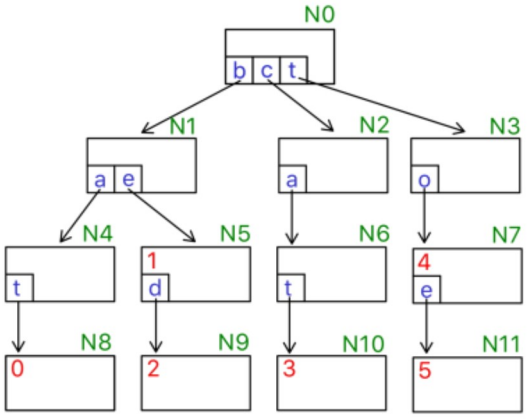

Figura 1. Trie ejemplo.

Para cada nodo se muestra:

- El valor almacenado (en rojo), si es un nodo final. Si no es final, no aparece ningún valor.
- El diccionario con los hijos, cuyas claves son los caracteres mostrados (en azul) y cuyos valores asociados son los nodos indicados por las flechas.

Hemos etiquetado (en verde) los distintos nodos de la figura para poder referirnos a ellos. Como puede verse, el nodo raíz N0 no es final (no incluye ningún valor en rojo) y su diccionario almacena los caracteres 'b', 'c' y 't'. Esto indica que todas las palabras almacenadas en este Trie comienzan por uno de estos tres caracteres. El nodo asociado con el carácter 'c' del nodo N0 es el nodo N2. Este nodo tampoco es final y su diccionario solo incluye el carácter 'a'. Esto indica que las palabras que empiezan por 'c' van seguidas de una 'a', es decir, tienen prefijo "ca". El nodo asociado con el carácter 'a' del nodo N2 es el nodo N6. Este nodo tampoco es final y su diccionario solo incluye el carácter 't'. Esto indica que tras el prefijo "ca" siempre aparece una 't', es decir, las palabras que empiezan por 'c' tienen prefijo "cat". Por último, el nodo asociado con la 't' del nodo N6 es el nodo N10. Este nodo es final (incluye el valor 3) y, por tanto, marca el final de una palabra almacenada en el Trie y su valor asociado (en este caso la palabra es "cat" y el valor asociado es 3).

Por otro lado, si volvemos al nodo raíz N0, vemos que el carácter 'b' tiene asociado el nodo N1, que incluye en su diccionario los caracteres 'a' y 'e', lo cual define los prefijos "ba" y "be". Si proseguimos desde la 'a' de N1 a N4 y desde la 't' de éste al nodo final N8, obtenemos una nueva palabra incluida en el Trie: "bat" con valor asociado 0. Si desde la 'e' de N1 vamos a N5, al ser éste un nodo final, tenemos la palabra "be" con valor asociado 1. Pero si, además, vamos desde la 'd' de N5 al nodo final N9, obtenemos la palabra "bed" y su valor asociado 2.

De forma similar, partiendo desde la 't' de N0 y descendiendo por la rama más a la derecha, obtendríamos la palabra "to" (con valor 4) y la palabra "toe" (con valor 5).

Por tanto, podemos observar que las distintas palabras almacenadas en el Trie se obtienen considerando todos los caminos que van desde el nodo raíz a un nodo final, concatenando los distintos caracteres claves de los diccionarios en cada nodo del camino.

Haskell

En Haskell, el tipo de datos para representar un Trie que almacene strings y valores de tipo `a` será el siguiente:

```
import qualified DataStructures.Dictionary.AVLDictionary as D

data Trie a = Empty | Node (Maybe a) (D.Dictionary Char (Trie a))
```

El tipo `D.Dictionary k v` representa un diccionario (o mapa) cuyas claves tienen tipo `k` y con valores asociados de tipo `v`. Puedes ver las distintas operaciones disponibles para diccionarios en el fichero `DataStructures.Dictionary.AVLDictionary.hs`.

Un Trie `Empty` será utilizado únicamente para representar un Trie vacío (que almacene cero cadenas).

Un Trie `Node` representará un Trie no vacío. Si el nodo es final, la primera componente almacenará un valor con `Just x`, siendo `x` el valor almacenado. Si el nodo no es final, la primera componente almacenará `Nothing`. La segunda componente del nodo es el diccionario cuyas claves son caracteres (tipo `Char`) y cuyos valores asociados son otros nodos (tipo `Trie a`).

a. (0.5 puntos) Define la función `empty` que devuelve un `Trie a` vacío.

```
empty :: Trie a
```

b. (0.5 puntos) Define la función `isEmpty` que, dado un Trie, devuelve `True` si éste está vacío o `False` en caso contrario.

```
isEmpty :: Trie a -> Bool
```

c. (0.5 puntos) Define la función `sizeValue` que, dado un valor de tipo `Maybe a`, devuelva 0 si es `Nothing` o 1 en caso contrario (si es un `Just`).

```
sizeValue :: Maybe a -> Int
```

Hay un apéndice al final del enunciado donde se recuerda el uso del tipo `Maybe`.

d. (1 punto) Define la función `size` que, dado un Trie, devuelve el número de cadenas que almacena. Observa que por cada cadena almacenada habrá un nodo final en el Trie. Por ejemplo, para el Trie de la figura 1, la función `size` tiene que devolver 6.

```
size :: Trie a -> Int
```

e. (0.5 puntos) Define la función `toTrie` que, dado un valor de tipo `Maybe (Trie a)`, devuelve `Empty` si el argumento es `Nothing` o que devuelve el Trie argumento si éste es de la forma `Just`.

```
toTrie :: Maybe (Trie a) -> Trie a
```

f. (0.5 puntos) Define la función `childOf` que, dados un carácter `c` y un Trie `t`, devuelve el Trie asociado con el carácter `c` en el diccionario del nodo `t`. Si `t` es un Trie vacío o si `c` no es una clave en el diccionario del nodo `t`, se devolverá `Empty`.

```
childOf :: Char -> Trie a -> Trie a
```

g. (1.5 puntos) Define la función `search` que, dados un `String` y un Trie `t`, devuelve el valor `v` asociado en el Trie a dicha cadena como `Just v`. Si el Trie `t` no contiene dicha cadena devuelve `Nothing`.

```
search :: String -> Trie a -> Maybe a
```

El algoritmo a aplicar es el siguiente:

- Si `t` es un Trie vacío, la cadena no está incluida y, por tanto, se devolverá `Nothing`.
- Si la cadena es vacía se devolverá el valor de tipo `Maybe` almacenado en el nodo `t` (un `Just` con el valor asociado si el nodo `t` es final o un `Nothing` si no lo es).
- Si la cadena no es vacía (tiene la forma `(c:cs)`) se buscará (recursivamente) la cola de la cadena (`cs`) en el Trie asociado con el carácter `c` en el diccionario del nodo `t` (función `childOf`).

h. (0.5 puntos) Define la función `update` que toma un Trie `t`, un carácter `c` y un Trie `child`. Si el Trie `t` es vacío devuelve un `Node` no final cuyo diccionario incluye solamente una asociación entre el carácter `c` y el Trie `child`. En otro caso devuelve un `Node` como `t` pero cuyo diccionario se habrá actualizado para que el carácter `c` quede asociado con el Trie `child`.

```
update :: Trie a -> Char -> Trie a -> Trie a
```

i. (1.5 puntos) Define la función `insert` que, dados una cadena `str`, un valor `v` a asociar con ésta y un Trie `t`, devuelve el Trie que se obtiene al insertar la cadena `str` y su valor asociado `v` en el Trie `t`. Si la cadena ya estaba incluida en el Trie, se devolverá una Trie en la que el valor asociado quede actualizado.

```
insert :: String -> a -> Trie a -> Trie a
```

El algoritmo a aplicar es el siguiente:

- Si la cadena es vacía y el Trie `t` es vacío se devuelve un `Node` final con valor `Just v` y diccionario vacío.
- Si la cadena es vacía y el Trie `t` no es vacío se devuelve un `Node` final con valor `Just v` y el mismo diccionario que hay en `t`.
- Si la cadena no es vacía (tiene la forma `(c:cs)`) se insertará (recursivamente) la cola de la cadena (`cs`) en el Trie asociado con el carácter `c` en el diccionario del nodo `t` (función `childOf`), obteniendo como resultado el nuevo Trie hijo (`child`) y se actualizará `t` de forma que el carácter `c` quede asociado con el nuevo hijo (`child`).

Apéndice Haskell. El tipo Maybe

El tipo `Maybe a` encapsula un valor opcional de tipo `a` y está predefinido en Haskell de la siguiente forma:

```
data Maybe a = Nothing | Just a
```

Un valor de tipo `Maybe a` puede almacenar un valor `x` de tipo `a` (representado como `Just x`) o puede estar vacío (representado como `Nothing`). `Maybe` es una manera de representar resultados que no siempre han de estar definidos. Por ejemplo, cuando se busca el valor asociado a una clave en una estructura de datos, si la clave no está está presente se puede devolver `Nothing`. Si la clave está presente y el valor asociado es `v`, se devolverá `Just v`. En Java, el caso en que la clave no está presente se puede implementar devolviendo `null`.

Puedes definir funciones que procesen valores de tipo `Maybe` usando patrones. Por ejemplo:

```
twiceMaybe :: Maybe Int -> Int
twiceMaybe Nothing  = 0    -- Devuelve 0 si el argumento es de la forma Nothing.
twiceMaybe (Just x) = 2*x  -- Devuelve el doble de x si el argumento es de la forma Just x.
```

También puedes usar algunas funciones disponibles en la biblioteca `Data.Maybe` para trabajar con el tipo de datos `Maybe` como:

```
isJust :: Maybe a -> Bool
isJust Nothing  = False
isJust (Just _) = True
-- Devuelve True si y solo si el argumento sigue el patrón Just _.

isNothing :: Maybe a -> Bool
isNothing Nothing  = True
isNothing (Just _) = False
-- Devuelve True si y solo si el argumento es Nothing.

fromJust :: Maybe a -> a
fromJust Nothing  = error "fromJust of Nothing"
fromJust (Just x) = x
-- Si el argumento sigue el patrón Just x, devuelve el valor x.
-- Si el argumento es Nothing eleva un error.

maybe :: b -> (a -> b) -> Maybe a -> b
maybe x _ Nothing  = x
maybe _ f (Just x) = f x
-- Si el tercer argumento es Nothing, devuelve como resultado el primer argumento.
-- Si el tercer argumento es de la forma Just x, devuelve el resultado de aplicar la función segundo argumento a x.

-- Por ejemplo, la función twiceMaybe puede ser implementada como:
twiceMaybe :: Maybe Int -> Int
twiceMaybe mb = maybe 0 (\x -> 2*x) mb
```

Java

En Java, la clase anidada para representar un nodo de un Trie que almacene strings y valores de tipo `V` será el siguiente:

```
  protected static class Node<V> {
    V value;
    Dictionary<Character, Node<V>> children;

    Node() {
      this.value = null;
      this.children = new AVLDictionary<>();
    }
  }
```

La interfaz `Dictionary<K,V>` representa un diccionario (o mapa) cuyas claves tienen tipo `K` y con valores asociados de tipo `V`. Puedes ver los distintos métodos soportados en la interfaz `dataStructures.dictionary.Dictionary` y en la clase `dataStructures.dictionary.AVLDictionary` que implementa dicha interfaz.

Un objeto de la clase `Node` representará un Trie no vacío. Si el nodo es final, la componente `value` almacenará un valor. Si el nodo no es final, la componente `value` almacenará `null`. La componente `children` es el diccionario cuyas claves son caracteres y cuyos valores asociados son los nodos hijos.

Un Trie se representará como un objeto de la clase `DictionaryStringTrie`, de forma que su atributo

```
protected Node<V> root;
```

almacenará `null` si el Trie es vacío (almacena cero cadenas) o una referencia al nodo raíz si el Trie no es vacío.

a. (0.5 puntos) Implementa el constructor de la clase `DictionaryStringTrie` que crea un Trie vacío.

```
      public DictionaryStringTrie()
```

b. (0.5 puntos) Implementa el método `isEmpty` que devuelve `true` si el trie está vacío o `false` en otro caso.

```
      public boolean isEmpty()
```

c. (0.5 puntos) Implementa el método auxiliar `sizeValue` que devuelve 0 si el valor que se pasa como argumento es `null` o 1 en caso contrario.

```
      protected static <V> int sizeValue(V value)
```

d. (1.5 puntos) Implementa el método `size` que devuelve el número de cadenas almacenadas el Trie. Observa que por cada cadena almacenada habrá un nodo final en el Trie. Por ejemplo, para el Trie de la figura 1, el método `size` tiene que devolver 6.

```
      public int size()
```

Para implementar este método, implementa además el método recursivo auxiliar

```
protected static <V> int size(Node<V> node)
```

que calcula el número de cadenas almacenadas en el Trie cuya raíz se pasa como argumento.

e. (0.5 puntos) Implementa el método auxiliar `childOf` que, dado un carácter `c` y un nodo `node`, devuelve el nodo asociado con `c` en el diccionario del nodo `node`. Si `node` es un Trie vacío o si `c` no es una clave en el diccionario, se devolverá `null`.

```
protected static <V> Node<V> childOf(char c, Node<V> node)
```

f. (1.5 puntos) Implementa el método `search` que, dado un String `str`, devuelve el valor asociado a `str` en el Trie.

```
public V search(String str)
```

Para implementar este método, implementa además el método recursivo auxiliar

```
protected static <V> V search(String str, Node<V> node)
```

que busca el valor asociado con la cadena `str` en el Trie cuya raíz es el nodo pasado como argumento.

El algoritmo a aplicar es el siguiente:

- Si `node` es un Trie vacío, la cadena no está incluida y, por tanto, se devolverá `null`.
- Si la cadena es vacía, se devolverá el valor almacenado en la componente `value` de `node` (el valor asociado si el nodo es final o `null` si no lo es).
- Si la cadena no es vacía, sea `c` su primer carácter y `suffix` el resto de la cadena. Entonces, se buscará (recursivamente) la cadena `suffix` en el `node` asociado con el carácter `c` en el diccionario de `node` (método `childOf`).

g. (2 puntos) Implementa el método `insert` que, dada una cadena `str` y su valor `value` asociado, los inserta en el Trie. Si la cadena ya estaba incluida en el Trie, simplemente se actualizará su valor asociado.

```
public void insert(String str, V value)
```

Para implementar este método, implementa además el método recursivo auxiliar

```
protected static <V> Node<V> insert(String str, V value, Node<V> node)
```

que inserta la cadena `str` y su valor asociado `value` en el Trie cuya raíz es el nodo pasado como argumento. El método devolverá el nodo modificado.

El algoritmo a aplicar es el siguiente:

- Si `node` es un Trie vacío, hay que crear primero un nuevo nodo no final con diccionario vacío. En otro caso usaremos el nodo argumento.
- Además,
  - Si la cadena es vacía, se almacena en la componente `value` del nodo el valor `value` recibido como argumento.
  - Si la cadena no es vacía, sea `c` su primer carácter y `suffix` el resto de la cadena. Entonces, se insertará (recursivamente) la cadena `suffix` en el nodo asociado con el carácter `c` en el diccionario del nodo (método `childOf`), obteniendo como resultado el nuevo nodo hijo (`child`) y se actualizará el diccionario del nodo de forma que el carácter `c` quede asociado con el nuevo hijo (`child`).
- Se debe devolver el nodo modificado.

Ejercicios extra para alumnos a tiempo parcial sin evaluación continua.

Haskell

e1) (1.5 puntos) Define la función `strings` que, dado un Trie, devuelve una lista con todas las palabras almacenadas en éste.

```
strings :: Trie a -> [String]
```

e2) (1.5 puntos) ***Usando una función de plegado,*** define la función `fromList` que, dada una lista de palabras, devuelve un Trie con ellas y donde los valores asociados sean el número de repeticiones de cada palabra en la lista.

```
fromList :: [String] -> Trie Int
```

Java

e1) (1.5 puntos) Define el método `strings` que devuelve una lista con todas las palabras almacenadas en un Trie.

```
public List<String> strings()
```

e2) (1.5 puntos) ***Usando un bucle foreach,*** define el método `fromList` que, dada una lista de palabras, devuelve un Trie con ellas y donde los valores asociados sean el

número de repeticiones de cada palabra en la lista.

```
public static DictionaryStringTrie<Integer> fromList(List<String> words)
```

Última modificación: jueves, 19 de enero de 2023, 08:56

◄ Códigos Haskell y Java del tema

Saltar a...

Códigos Java y Haskell ►

Universidad de Málaga · Avda. Cervantes, 2. 29071 MÁLAGA · Tel. 952131000 · info@uma.es

Todos los derechos reservados