Implementación de diccionarios sobre trie

Algoritmos y Estructuras de Datos

1^{er} cuatrimestre de 2024

TAD Diccionario

Interfaz

Recordemos la interfaz de Diccionario (del apunte de TADs básicos):

```
TAD Diccionario<K,V> {
   proc diccionarioVacío():  Diccionario<K,V>
   proc está(in d:  Diccionario<K,V>, in k:  K): bool
   proc definir(inout d:  Diccionario<K,V>, in k:  K,
in v:  V)
   proc obtener(in d:  Diccionario<K,V>, in k:  K): V
   proc borrar(inout d:  Diccionario<K,V>, in k:  K)
   proc definirRápido(inout d:  Diccionario<K,V>, in k:
K, in v:  V)
   proc tamaño(in d:  Diccionario<K,V>):  ℤ
}
```

Estructura conocida

	ABB balanceado	**Trie**
está?	$ln(n)$	??
obtener	$ln(n)$	??
definir	$ln(n)$	??

Estructura conocida

	ABB balanceado	**Trie**
está?	$ln(n)$	??
obtener	$ln(n)$	??
definir	$ln(n)$	??

Profe, la clase pasada vimos conjunto, no diccionario.

Conceptos principales

Ideas:

Conceptos principales

Ideas:

- Claves con partes

Conceptos principales

Ideas:

- Claves con partes
- Cadenas de elementos tomados de un alfabeto acotado

Conceptos principales

Ideas:

- Claves con partes
- Cadenas de elementos tomados de un alfabeto acotado
- `String`: cadenas de caracteres ASCII.

Conceptos principales

Ideas:

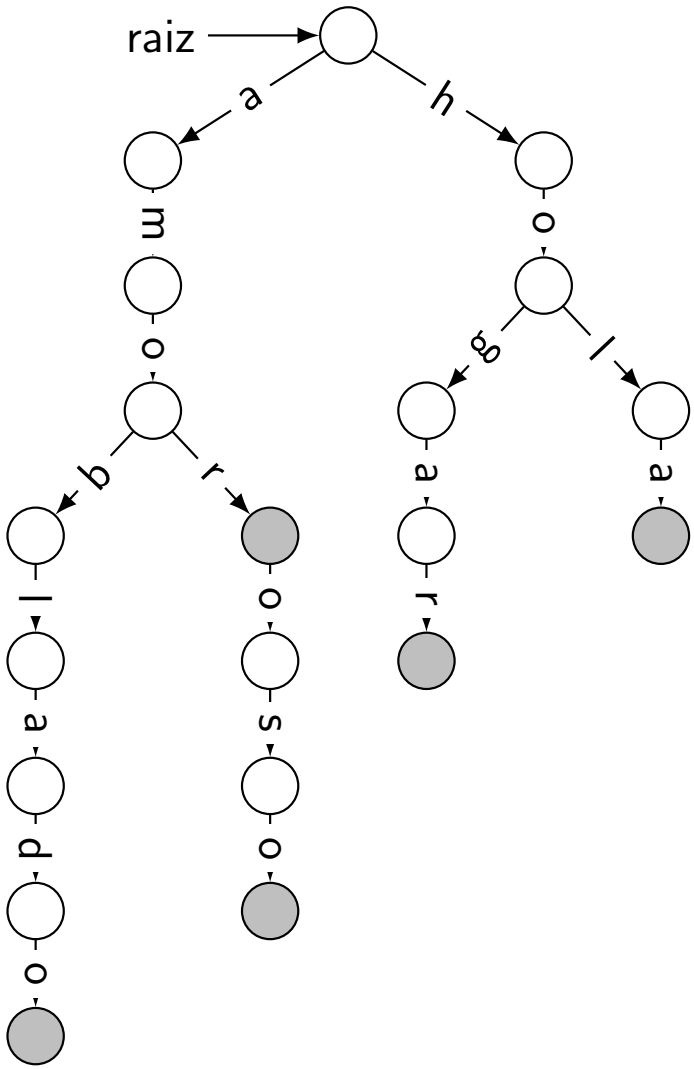
- Claves con partes
- Cadenas de elementos tomados de un alfabeto acotado
- `String`: cadenas de caracteres ASCII.
- ¿Qué pasa con los números?

Comparación de las complejidades

	ABB balanceado	**Trie**
está?	$ln(n).\|k\|$	$\|k\|$
obtener	$ln(n).\|k\|$	$\|k\|$
definir	$ln(n).\|k\|$	$\|k\|$

A continuación la estructura

Representación Trie para cadenas de un alfabeto Σ



- Arbol K-Ario (¿cuántos vale K?)
- Nodos: <significado: T, hijos: dicc(Σ, Nodo)>

Invariante de Representación

¿Qué cosas no pueden pasar en la estructura?

Invariante de Representación

¿Qué cosas no pueden pasar en la estructura?

- No llego por dos claves al mismo nodo / los nodos tienen un solo padre salvo la raíz (que no tiene padre) / es un árbol
- No hay nodos inútiles o (bien dicho) los nodos, si no tienen significado, tienen hijos.

Representación de los nodos

```
private class Nodo{
    dicc<char, Nodo> siguientes;
    T definicion;
    Nodo(){
        /*...*/
    }
};
Nodo raiz;
int tamanio;
```

Tenemos dos variables de instancia:

- `tamanio` contiene la cantidad de claves del diccionario.
- `raiz` apunta al nodo raíz del trie.

¿Siempre hay raíz?

Representación de los nodos

```
private class Nodo{
    dicc<char, Nodo> siguientes;
    T definicion;
    Nodo(){
        /*...*/
    }
};
Nodo raiz;
int tamanio;
```

Tenemos dos variables de instancia:

- `tamanio` contiene la cantidad de claves del diccionario.
- `raiz` apunta al nodo raíz del trie.

¿Siempre hay raíz?
Queremos un diccionario para recorrer el árbol k-ario: ¿Qué opciones tenemos?

- Las claves son cadenas de texto (`String`) y sus partes son caracteres (`char`)
- En este escenario, podemos asumir un abecedario acotado (ASCII: 256 caracteres)
- El código ASCII de un `char` x se puede obtener en Java con `(int) x`
- Podemos usar `ArrayList<Nodo>` como dicc(int, ref a Nodo)

Representación de los nodos

```
private class Nodo{
    ArrayList<Nodo> siguientes;
    T definicion;
    Nodo(){
        /** Constructor */
    }

};
Nodo raiz;
int size;
```

Definido

- Vamos a tener un *nodo actual* con el que recorremos y un índice de nuestra ubicación en la *clave* (si son cadenas de texto, en qué caracter estamos posicionados)

Definido

- Vamos a tener un *nodo actual* con el que recorremos y un índice de nuestra ubicación en la *clave* (si son cadenas de texto, en qué caracter estamos posicionados)
- Empezamos en la raíz como nodo actual

Definido

- Vamos a tener un *nodo actual* con el que recorremos y un índice de nuestra ubicación en la *clave* (si son cadenas de texto, en qué caracter estamos posicionados)
- Empezamos en la raíz como nodo actual, si no existe devolvemos `False`.

Definido

- Vamos a tener un *nodo actual* con el que recorremos y un índice de nuestra ubicación en la *clave* (si son cadenas de texto, en qué caracter estamos posicionados)
- Empezamos en la raíz como nodo actual, si no existe devolvemos `False`.
- Si existe la raíz, recorremos el trie mirando cada caracter de la clave. Esto significa:

Definido

- Vamos a tener un *nodo actual* con el que recorremos y un índice de nuestra ubicación en la *clave* (si son cadenas de texto, en qué caracter estamos posicionados)
- Empezamos en la raíz como nodo actual, si no existe devolvemos `False`.
- Si existe la raíz, recorremos el trie mirando cada caracter de la clave. Esto significa:
  - Si en siguientes del nodo actual ese caracter apunta a `null`, devolvemos `False`. No puedo llegar al nodo que representa la clave.

Definido

- Vamos a tener un *nodo actual* con el que recorremos y un índice de nuestra ubicación en la *clave* (si son cadenas de texto, en qué caracter estamos posicionados)
- Empezamos en la raíz como nodo actual, si no existe devolvemos `False`.
- Si existe la raíz, recorremos el trie mirando cada caracter de la clave. Esto significa:
  - Si en siguientes del nodo actual ese caracter apunta a `null`, devolvemos `False`. No puedo llegar al nodo que representa la clave.
  - Si no, pasamos a ver el nodo al que apunta siguientes del caracter actual y consideramos la clave desde el siguiente caracter.

Definido

- Vamos a tener un *nodo actual* con el que recorremos y un índice de nuestra ubicación en la *clave* (si son cadenas de texto, en qué caracter estamos posicionados)
- Empezamos en la raíz como nodo actual, si no existe devolvemos `False`.
- Si existe la raíz, recorremos el trie mirando cada caracter de la clave. Esto significa:
  - Si en siguientes del nodo actual ese caracter apunta a `null`, devolvemos `False`. No puedo llegar al nodo que representa la clave.
  - Si no, pasamos a ver el nodo al que apunta siguientes del caracter actual y consideramos la clave desde el siguiente caracter.
- Si llegamos al nodo que representa la clave, está definida si tiene un significado definido.

Definido

- Vamos a tener un *nodo actual* con el que recorremos y un índice de nuestra ubicación en la *clave* (si son cadenas de texto, en qué caracter estamos posicionados)
- Empezamos en la raíz como nodo actual, si no existe devolvemos `False`.
- Si existe la raíz, recorremos el trie mirando cada caracter de la clave. Esto significa:
  - Si en siguientes del nodo actual ese caracter apunta a `null`, devolvemos `False`. No puedo llegar al nodo que representa la clave.
  - Si no, pasamos a ver el nodo al que apunta siguientes del caracter actual y consideramos la clave desde el siguiente caracter.
- Si llegamos al nodo que representa la clave, está definida si tiene un significado definido.

¿Qué pasa con la clave que no tiene caracteres?: `""`

Definir un elemento

Definir un elemento

- Si el trie está vacío creamos un nuevo Nodo al que apunta la raíz.

Definir un elemento

- Si el trie está vacío creamos un nuevo Nodo al que apunta la raíz.
- Buscamos en qué lugar del trie debe ir la nueva clave. Para ello vamos recorriendo el trie como en Definido.

Definir un elemento

- Si el trie está vacío creamos un nuevo Nodo al que apunta la raíz.
- Buscamos en qué lugar del trie debe ir la nueva clave. Para ello vamos recorriendo el trie como en Definido.
- Cuando nos encontramos que un caracter en *siguientes* apunta a `null` creamos un nuevo Nodo al que apuntar.

Definir un elemento

- Si el trie está vacío creamos un nuevo Nodo al que apunta la raíz.
- Buscamos en qué lugar del trie debe ir la nueva clave. Para ello vamos recorriendo el trie como en Definido.
- Cuando nos encontramos que un caracter en *siguientes* apunta a `null` creamos un nuevo Nodo al que apuntar.
- Al terminar de recorrer todos los caracteres de la clave llegamos al lugar donde debe ir el significado.

Definir un elemento

- Si el trie está vacío creamos un nuevo Nodo al que apunta la raíz.
- Buscamos en qué lugar del trie debe ir la nueva clave. Para ello vamos recorriendo el trie como en Definido.
- Cuando nos encontramos que un caracter en *siguientes* apunta a `null` creamos un nuevo Nodo al que apuntar.
- Al terminar de recorrer todos los caracteres de la clave llegamos al lugar donde debe ir el significado.
- Asignamos como significado del nodo encontrado una referencia del significado recibido por parámetro.

Borrar nodo: Tener en cuenta

No hay nodos inútiles, es decir, los nodos, si no tienen significado tienen hijos útiles.

Borrar un elemento

Borrar un elemento

- Hay que borrar el significado asociado a la clave y todos los nodos intermedios que ya no tengan razón de ser. (Volver al *rep*)

Borrar un elemento

- Hay que borrar el significado asociado a la clave y todos los nodos intermedios que ya no tengan razón de ser. (Volver al *rep*)
- Para ello hay que encontrar el nodo que representa la clave y el último nodo que no hay que borrar.
- Dos casos posibles:

Borrar un elemento

- Hay que borrar el significado asociado a la clave y todos los nodos intermedios que ya no tengan razón de ser. (Volver al *rep*)
- Para ello hay que encontrar el nodo que representa la clave y el último nodo que no hay que borrar.
- Dos casos posibles:
  - El último nodo es el mismo que representa la clave. No se borran nodos. Esto pasa si el nodo de la clave tiene hijos.

Borrar un elemento

- Hay que borrar el significado asociado a la clave y todos los nodos intermedios que ya no tengan razón de ser. (Volver al *rep*)
- Para ello hay que encontrar el nodo que representa la clave y el último nodo que no hay que borrar.
- Dos casos posibles:
  - El último nodo es el mismo que representa la clave. No se borran nodos. Esto pasa si el nodo de la clave tiene hijos.
  - El último nodo no es el mismo que el que representa la clave. Todos los descendientes de este nodo tienen un solo hijo y ningún significado.

Borrar un elemento: encontrar la clave

Buscamos el nodo que representa la clave y en el camino guardamos el último nodo que no hay que borrar.

- Mientras tenga que ver elementos de la clave:
  - Si el *nodo actual* tiene más de un hijo o tiene significado, el *último nodo* es el *nodo actual* y *último índice* es nuestra posición en la clave.
  - Avanzamos el *nodo actual* bajando por el siguiente caracter de la clave y avanzamos nuestra posición en la clave.
- *nodo actual* es el nodo de la clave. Borramos su significado.
- *último nodo* es el último nodo en el camino de la clave que no hay que borrar y *último índice* la posición de la clave en la que estábamos.

Borrar un elemento: borrar lo innecesario

Si *nodo actual* no tiene hijos, borramos los descendientes de *último nodo*. Esto es similar a borrar una lista. ¿Cómo lo hacían?

Borrar un elemento: borrar lo innecesario

Si *nodo actual* no tiene hijos, borramos los descendientes de *último nodo*. Esto es similar a borrar una lista. ¿Cómo lo hacían? Eliminamos toda referencia a ella. Es decir, *último nodo*, en la posición *último índice*, ya no apunta a nadie.

Recorrer las claves

El recorrido de las claves es equivalente al recorrido inorder de un árbol binario de búsqueda.
Una posible implementación recursiva:

- Comienzo con la raíz, donde el prefijo es la cadena vacía.

Recorrer las claves

El recorrido de las claves es equivalente al recorrido inorder de un árbol binario de búsqueda.
Una posible implementación recursiva:

- Comienzo con la raíz, donde el prefijo es la cadena vacía.
- Si el nodo actual tiene significado, imprimo el prefijo (el cual es la clave).

Recorrer las claves

El recorrido de las claves es equivalente al recorrido inorder de un árbol binario de búsqueda.
Una posible implementación recursiva:

- Comienzo con la raíz, donde el prefijo es la cadena vacía.
- Si el nodo actual tiene significado, imprimo el prefijo (el cual es la clave).
- Continúo la recursión con cada hijo del nodo actual, pasando como prefijo la concatenación del prefijo actual con el caracter que mapea a ese hijo.

Bibliografía

- Sedgewick, R., Wayne, K. (2011). Algorithms, 4th Edition