Implementación de diccionarios sobre trie en C++

Algoritmos y Estructuras de Datos II

1.^{er} cuatrimestre de 2019

Historia

- 100k partidas de Rocket League
- Quiero saber cuantos goles hizo cada jugador
- Recorro las partidas, para cada jugador asocio su username con un número.
- Con cada partida, sumo al número asociado a un jugador la cantidad de goles que metió.

¿Solución?

Diccionario Lineal AED2

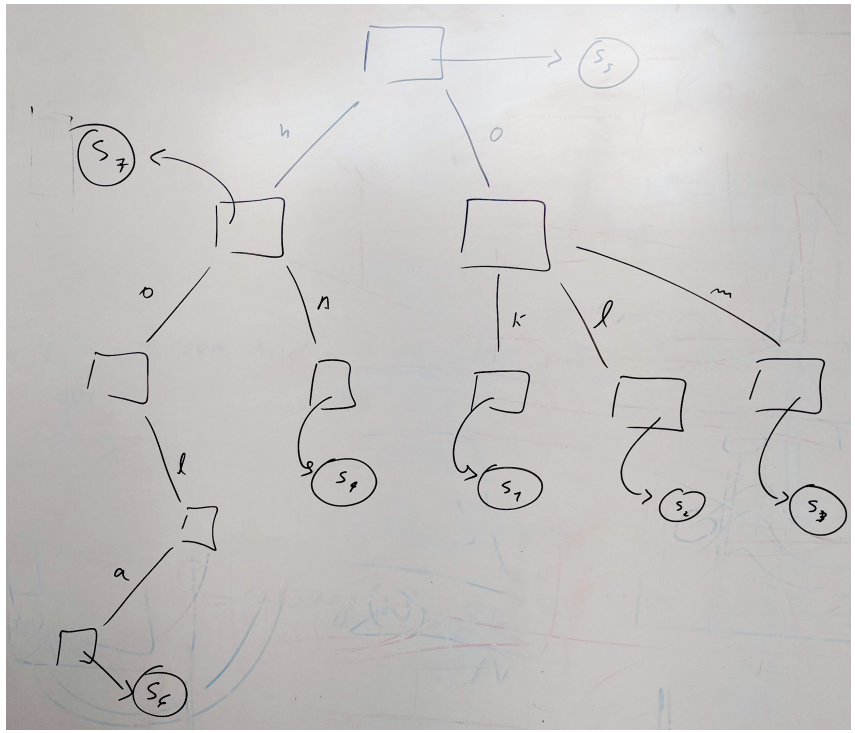
DEFINIDO?(**in** d: `dicc(`κ, σ`)`, **in** k: κ) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?(d, k)}
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K = \text{claves}(d)$
Descripción: devuelve `true` si y sólo k está definido en el diccionario.

SIGNIFICADO(**in** d: `dicc(`κ, σ`)`, **in** k: κ) $\rightarrow$ res : σ
Pre $\equiv$ {def?(d, k)}
Post $\equiv$ {alias($res =_{\text{obs}}$ Significado(d, k))}
Complejidad: $\Theta(\sum_{k' \in K} equal(k, k'))$, donde $K = \text{claves}(d)$
Descripción: devuelve el significado de la clave k en d.
Aliasing: res es modificable si y sólo si d es modificable.

Muuuuchas comparaciones... :(

Trie

- Claves con partes
- Búsqueda / inserción / borrado $\mathcal{O}(|k|)$



- Arbol K-Ario
- Significado en nodos
- Partes de clave en aristas

Resolver el problema

Interfaz

- definido: Dicc(K, S) $\times$ K $\rightarrow$ bool
- obtener: Dicc(K, S) d $\times$ K k $\rightarrow$ S $\{\neg$definido(d, k)$\}$
- definir: Dicc(K, S) $\times$ K $\times$ S $\rightarrow$ Dicc(K, S)

Resolver el problema

¿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 tengo ciclos (idem arriba)
- No hay nodos inútiles o (bien dicho) los nodos, si no tienen significado tienen hijos.

Esquema de la clase string_map<T>

```
#ifndef STRING_MAP_H_
#define STRING_MAP_H_

template <class T>
class string_map {
    public:
        /*...*/
    private:
        /*...*/
};

/* ... */
#endif //STRING_MAP_H_
```

Representación de los nodos

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

Tenemos dos variables de instancia:

- `size` contiene la cantidad de claves del `string_map`.
- `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 `string` y sus partes son `char`
- En este escenario podemos asumir un abecedario acotado (ASCII, 256 caracteres)
- Tenemos la función `int(char)` de C++ devuelve un número: el código ASCII de un caracter.
- Podemos usar `vector<int>` como un dicc(int, Nodo*)

Representación de los nodos

```
private:
    struct Nodo{
        vector<Nodo*> siguientes;
        T* definicion;
        Nodo() : siguientes(256, nullptr),
                 definicion(nullptr) { }
        Nodo(T* def) : siguientes(256, nullptr),
                       definicion(def) { }
    };
    Nodo* raiz;
    int size;
```

Definido

- Vamos a tener un *nodo actual* con el que recorremos y un índice de nuestra ubicación en la *clave*
- 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.

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

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 copia del significado recibido por parámetro.

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 ningun 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.

Borrar un elemento: borrar lo innecesario

Si *nodo actual* no tiene hijos, borramos los descendientes de *último nodo*.

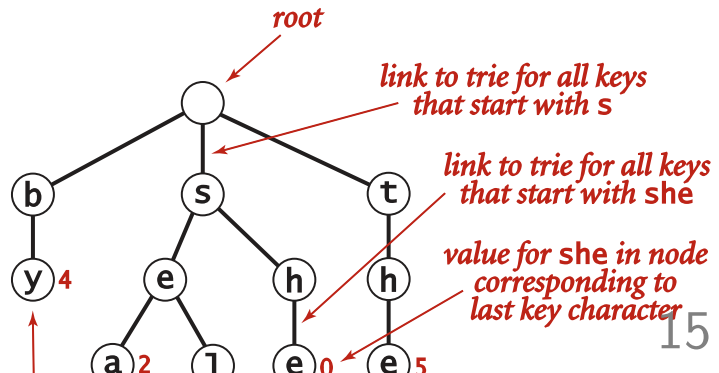
- Apunto *siguiente* al hijo de *último nodo* siguiendo la clave. Avanzo un lugar en la clave.
- Apunto la arista de *último nodo* a NULL.
- Apunto *último nodo* a *siguiente*
- Mientras tenga clave para mirar desde *último índice*:
  - Apunto *siguiente* al hijo de *último nodo* siguiendo la clave. Avanzo un lugar en la clave.
  - Borro *último nodo*.
  - Apunto *último nodo* a *siguiente*.
- Borro *último nodo*.

Esto último es como borrar una lista.

Caso base: trie vacío.

Tries In this section, we consider a search tree known as a *trie*, a data structure built from the characters of the string keys that allows us to use the characters of the search key to guide the search. The name "trie" is a bit of wordplay introduced by E. Fredkin in 1960 because the data structure is used for re*trie*val, but we pronounce it "try" to avoid confusion with "tree." We begin with a high-level description of the basic properties of tries, including search and insert algorithms, and then proceed to the details of the representation and Java implementation.

Basic properties. As with search trees, tries are data structures composed of *nodes* that contain *links* that are either *null* or references to other nodes. Each node is pointed to by just one other node, which is called its *parent* (except for one node, the *root*, which has no nodes pointing to it), and each node has R links, where R is the alphabet size. Often, tries have a substantial number of null links, so when we draw a trie, we typically omit null links. Although links point to nodes, we can view each link as pointing to a trie, the trie whose root is the referenced node. Each link corresponds to a character value—since each link points to exactly one node, we label each node with the character value corresponding to the link that points to it (except for the root, which has no link pointing to it). Each node also has a corresponding *value*, which may be null or the value associated with one of the string keys in



Insertion into a trie. As with binary search trees, we insert by first doing a search: in a trie that means using the characters of the key to guide us down the trie until reaching the last character of the key or a null link. At this point, one of the following two conditions holds:

- We encountered a null link before reaching the last character of the key. In this case, there is no trie node corresponding to the last character in the key, so we need to create nodes for each of the characters in the key not yet encountered and set the value in the last one to the value to be associated with the key.
- We encountered the last character of the key before reaching a null link. In this case, we set that node's value to the value to be associated with the key (whether or not that value is null), as usual with our associative array convention.

In all cases, we examine or create a node in the trie for each key character. The construction of the trie for our standard indexing client from CHAPTER 3 with the input

```
she sells sea shells by the sea shore
```

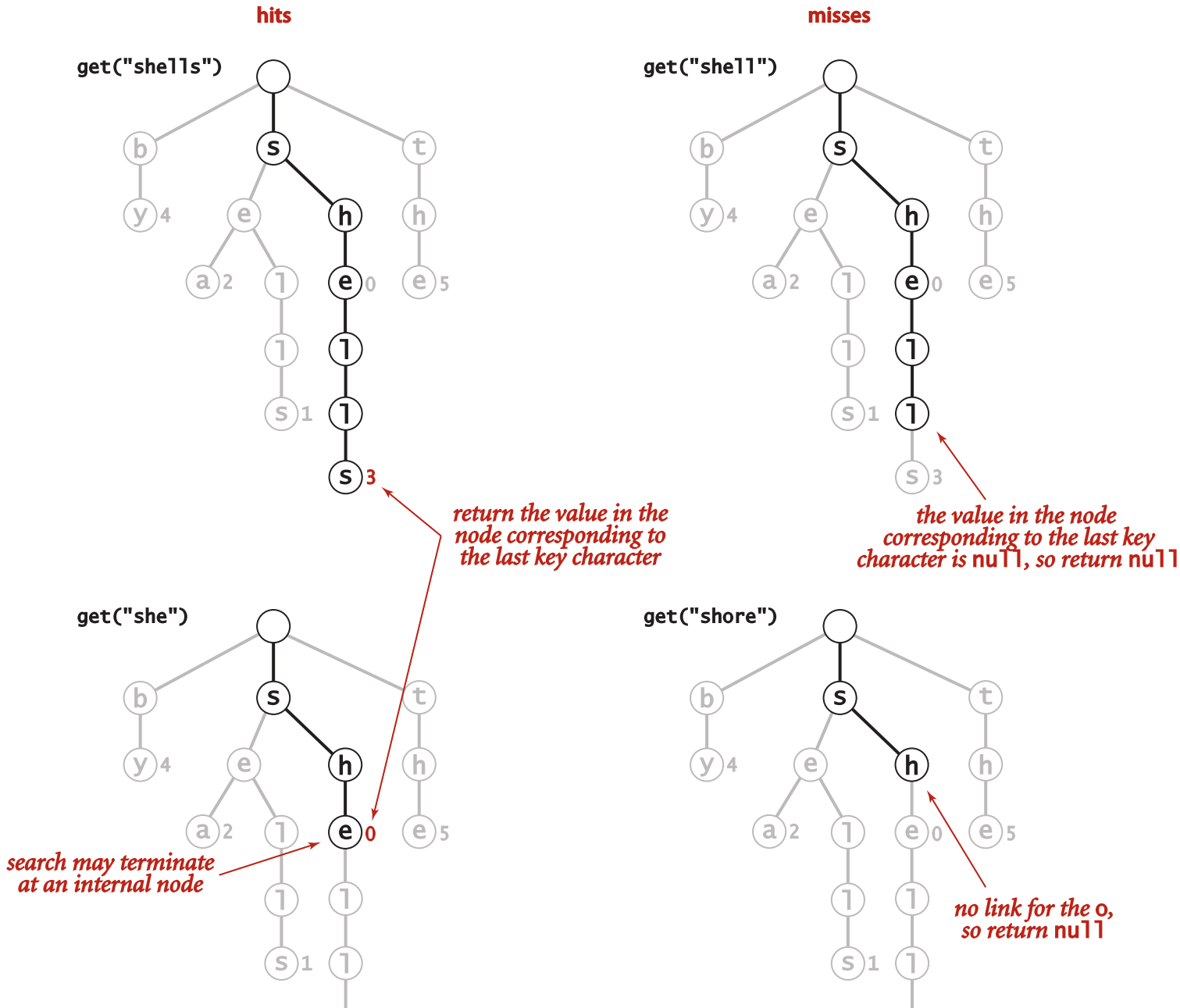
is shown on the facing page.

Node representation. As mentioned at the outset, our trie diagrams do not quite correspond to the data structures our programs will build, because we do not draw null links. Taking null links into account emphasizes the following important characteristics of tries:

- Every node has R links, one for each possible character.
- Characters and keys are *implicitly* stored in the data structure.

For example, the figure below depicts a trie for keys made up of lowercase letters, with

Bibliografía



Interfaz

- Queremos dotar a nuestra clase de una interfaz de Diccionario.
- Vamos a simular ser programadores de la `Standard Template Library` y programar un subconjunto de las funciones existentes[1]. En particular, para el taller, nos conformamos con:
  - Crear un diccionario nuevo (vacío).
  - Crear un diccionario a partir de otro (por copia).
  - El operador de asignación.
  - Definir un significado para una clave.
  - Decidir si una clave está definida en el diccionario.
  - Obtener el significado de una clave.
  - Borrar un elemento.
  - Tamaño del diccionario.

[1] http://www.cplusplus.com/reference/map/map/

Interfaz

```
template <class T>
class string_map {
    public:
        string_map();
        string_map(const string_map<T>& d);
        string_map& operator=(const string_map<T>& d);
        T& operator[](const string &key);
        int count(const string &key) const;
        T& at(const string& key);
        void erase(const string& key);
        int size() const;
        bool empty() const;
    private :
        /*...*/
};
```

¿Qué hacía el operador []?

Operador []

```
T& operator[](const string &key);
```

- Si la clave *key* está definida en el diccionario, la función devuelve una referencia a su significado.
- Si la clave *key* no está definida, la función crea un nuevo elemento con esa clave, y lo devuelve

¿Qué sucede cuando hacemos `dicc[key] = value`?

- Se busca la clave *key* en el diccionario y, en caso de no encontrarla, se crea un **valor por defecto**.
- Se llama al operador de asignación del tipo de *value* para reemplazar el valor almacenado en el diccionario.

¡A programar!

En `string_map.h` está la declaración de la clase, su parte pública y la declaración de `Nodo`.

En `string_map.hpp` está el esqueleto para que ustedes completen la definición de los métodos de `string_map`.