Implementación de diccionarios sobre trie en C++

Algoritmos y Estructuras de Datos II

2^{do} cuatrimestre de 2022

TAD Diccionario

Interfaz

TAD Diccionario

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)
- ...

Estructura conocida

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

Estructura conocida

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

Profe, la clase pasada vimos conjunto, no diccionario.

Idea

Ideas:

Idea

Ideas:

- Claves con partes

Idea

Ideas:

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

Idea

Ideas:

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

Idea

Ideas:

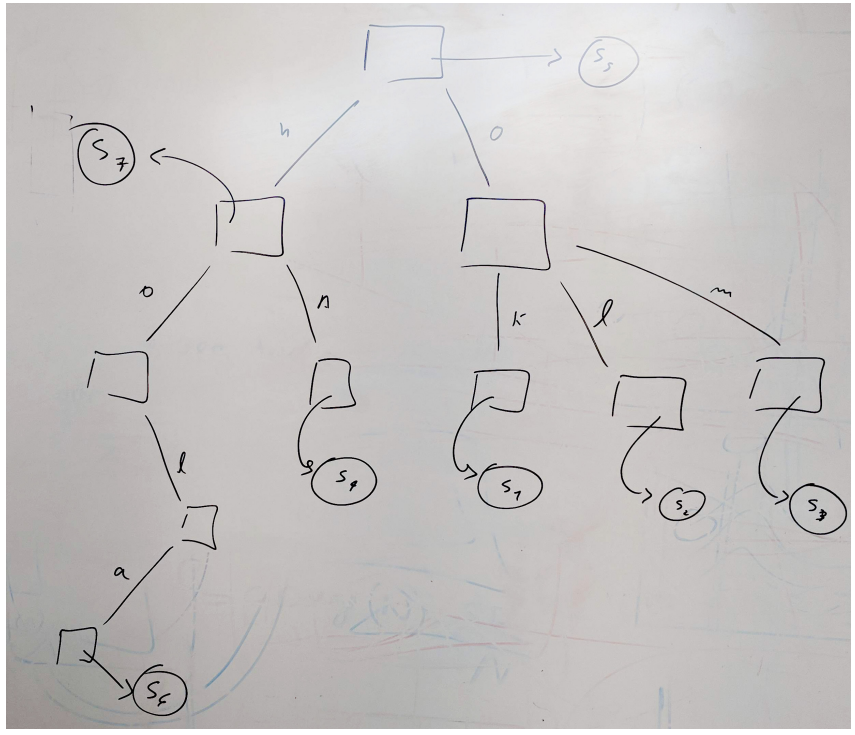
- Claves con partes
- Cadenas de elementos tomados de un alfabeto acotado
- `string`: cadenas de caracteres ASCII.
- ¿Qué pasa con los números?

Comparación de las complejidades

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

A continuación la estructura

Representación Trie



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

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?

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<Nodo*>` como un dicc(int, Nodo*)

Representación de los nodos

```
private:
    struct Nodo{
        vector<Nodo*> siguientes;
        T* definicion;
        Nodo(){
            /** Completar */
        }

    };
    Nodo* raiz;
    int size;
```

Ejercicio extra

Dar una estructura de datos que sirva para representar un conjunto de enteros que además de las operaciones usuales pueda responder si pertenece al conjunto algún número que sea divisible por 2^k.
Complejidad peor caso admitida: $O(k)$

Definido

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

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

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`.

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:

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 `nullptr`, 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*
- 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 `nullptr`, 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*
- 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 `nullptr`, 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*
- 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 `nullptr`, 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 `nullptr` 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 `nullptr` 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 `nullptr` 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 nodo: Tener en cuenta

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

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.

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 nullptr.
- 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*.

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 nullptr.
- 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.

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 nullptr.
- 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.

Bibliografía

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

Interfaz

- Queremos dotar a nuestra clase de una interfaz de Diccionario.

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

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:

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

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).

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

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).

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

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.

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

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.

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

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.

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

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.

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

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.

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

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);
```

Operador []

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

- Si la clave *key* está definida en el diccionario, la función devuelve una referencia a su significado.

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`?

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**.

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.

Fin

Fin

Fin

Fin... o quizás no

Extra 1: Operaciones con prefijos

El trie es útil para implementar algoritmos donde importan los prefijos.
Implementar de manera eficiente:

```
bool esPrefijoDeAlgunaClave(const string &str);
```

en $O(|str|)$

Extra 2: Extender una estructura de datos

El trie es útil para implementar algoritmos donde importan los prefijos.
Extender la estructura de datos trie para que dado un string s pueda en $O(|s|)$ decir cuantas claves tienen a s como prefijo.

Extra 3: std::optional[2]

std::optional

Defined in header `<optional>`

```
template< class T >
class optional;
```
(since C++17)

The class template `std::optional` manages an *optional* contained value, i.e. a value that may or may not be present.

A common use case for `optional` is the return value of a function that may fail. As opposed to other approaches, such as `std::pair<T,bool>`, `optional` handles expensive-to-construct objects well and is more readable, as the intent is expressed explicitly.

Any instance of `optional<T>` at any given point in time either *contains a value* or *does not contain a value*.

If an `optional<T>` *contains a value*, the value is guaranteed to be allocated as part of the `optional` object footprint, i.e. no dynamic memory allocation ever takes place. Thus, an `optional` object models an object, not a pointer, even though `operator*()` and `operator->()` are defined.

When an object of type `optional<T>` is contextually converted to `bool`, the conversion returns `true` if the object *contains a value* and `false` if it *does not contain a value*.

The `optional` object *contains a value* in the following conditions:

- The object is initialized with/assigned from a value of type T or another `optional` that *contains a value*.

The object *does not contain a value* in the following conditions:

- The object is default-initialized.
- The object is initialized with/assigned from a value of type `std::nullopt_t` or an `optional` object that *does not contain a value*.
- The member function `reset()` is called.

There are no optional references; a program is ill-formed if it instantiates an `optional` with a reference type. Alternatively, an `optional` of a `std::reference_wrapper` of type T may be used to hold a reference. In addition, a program is ill-formed if it instantiates an `optional` with the (possibly cv-qualified) tag types `std::nullopt_t` or `std::in_place_t`.

[2] https://en.cppreference.com/w/cpp/utility/optional

¡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`.