DEPARTAMENTO
INGENIERÍA INFORMÁTICA
Y CIENCIAS DE LA COMPUTACIÓN
FACULTAD DE INGENIERÍA UNIVERSIDAD DE CONCEPCIÓN

Estructuras de Datos (2022-1)

Proyecto 1: Trie

Profesor: Alexander Irribarra

Ayudantes: Leonardo Aravena, Diego Gatica, Vicente Lermanda

Objetivos

Los objetivos de este proyecto son:

- Mejorar la programación, compilación y ejecución de programas escritos en lenguaje *C++* u otros.
- Estudiar estructuras de datos basadas en árboles.
- Implementar múltiples variaciones de una misma estructura de datos con aplicación en problemas sobre cadenas de texto.

1. Descripción del problema

Un *trie* – pronunciado como la palabra *try* en inglés – es una estructura de datos basada en árboles, la cual tiene aplicaciones en la recuperación de información, conocida en inglés como *information re**trie**val*. Principalmente, se aplica en problemas relacionados con cadenas de texto, y por lo general es la estructura de datos que está detrás del auto completado predictivo.

La idea general de un *trie* es que cada camino que comienza en la raíz (el nodo que está más arriba del árbol) y termina en una nodo terminal (un nodo que está marcado) codifica una cadena, de tal manera que un *trie* almacena un conjunto de cadenas sobre las cuales podemos hacer diversas operaciones. La Figura 1 ilustra el comportamiento de esta estructura de datos.

El primer paso para desarrollar este proyecto, es leer acerca de esta estructura de datos. Se recomienda revisar el libro de referencia de la asignatura, que contiene un capítulo dedicado a esta estructura de datos, aunque también es cubierta por la mayoría de libros de estructuras de datos y algoritmos. Otra descripción se encuentra en `https://www.geeksforgeeks.org/trie-insert-and-search/` **(no se puede copiar el código fuente de este recurso ni de ningún otro, cualquier intento de plagio será penalizado con NCR)**.

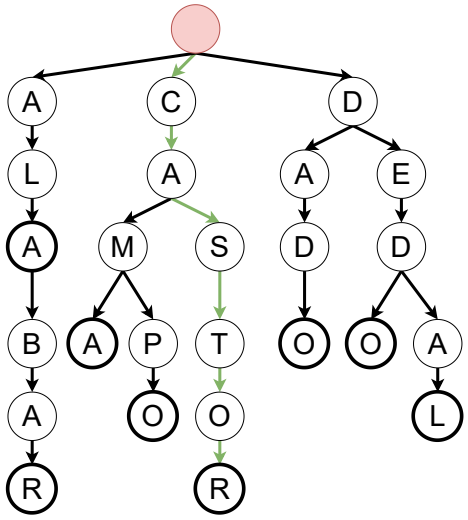

Figura 1: Ejemplo de un *trie* que contiene las cadenas "ALA", "ALABAR", "CAMA", "CAMPO", "CASTOR", "DADO", "DEDO" y "DEDAL". La raíz está destacada en rojo y los nodos terminales tienen su borde remarcado. El camino en color verde corresponde a la cadena "CASTOR".

Se implementarán diversas variantes de esta misma estructura de datos, variando la forma en que se almacenan los enlaces entre nodos. Se hará análisis teórico de las operaciones básicas considerando los siguientes parámetros:

- Número de elementos (cadenas) n.
- Tamaño del alfabeto ρ.
- Largo máximo de las cadenas m.

Además, se implementarán algoritmos que utilizan *tries* y se evaluará experimentalmente su desempeño, considerando tiempo y espacio.

2. Implementación

1. Se implementara una clase abstracta **Trie** que funcionará como interfaz para los métodos básicos, es decir:
   - `virtual void insert(const string &)=0`: Insertar una cadena al *trie*.
   - `virtual bool search(const string &)=0`: Retorna *true* si y solo si el string consultado está contenido en el *trie*.

- `virtual bool remove(const string &)=0`: Elimina del *trie* la cadena que recibe, en caso de existir. Retorna *true* si y solo si esta operación es válida (la cadena estaba contenida en el *trie*).
- `virtual vector<string> getAll()=0`: Retorna un vector que contiene todas las cadenas pertenecientes al *trie*.

2. Se implementarán dos variantes de implementación. En la primera, cada nodo contendrá un arreglo *children* de punteros a nodos, tan largo como el tamaño del alfabeto, donde $children[i]$ almacenará el puntero al hijo que codifique al i-ésimo caracter del alfabeto, o en caso de no existir, `NULL`. Para el ejemplo en la Figura 1, cuyo alfabeto son las letras mayúsculas, al mirar el nodo raíz, tenemos que $children[0]$ corresponde a su hijo que contiene la letra A, $children[1]$ contiene `NULL`, pues no tiene un hijo que corresponda a la letra B, $children[2]$ corresponde al hijo que contiene a la letra C, etc.

   En la segunda variante, en lugar de almacenar los punteros en un arreglo, se usará un *map*[1], cuyas claves serán caracteres (i.e. los símbolos del alfabeto) y los valores serán punteros a nodos. Notar que si se utilizan bien las operaciones de esta estructura de datos, solo quedan almacenadas entradas para los hijos existentes, es decir, no es necesario guardar punteros a `NULL`. Para el análisis teórico de esta solución, considerar que las operaciones de búsqueda, inserción y eliminación en un *map* son logarítmicas a la cantidad de elementos que contienen.

3. Se realizará un análisis experimental de las soluciones. Para esto, se pueden utilizar cadenas naturales o sintéticas. Ejemplos de diccionarios con cadenas naturales que se pueden utilizar: `https://www.winedt.org/dict.html`. El análisis experimental debe hacerse en base a la cantidad de elementos vs tiempo. Se deben incluir gráficos con los resultados. En el informe además se debe explicar como se realiza este análisis. Es decir, se debe explicar los datos de prueba utilizados (y como obtenerlos), los tamaños, el entorno de software y hardware en que se realizaron los experimentos y cualquier otro detalle relevante. La explicación debe ser suficiente para que alguien pueda tomar sus implementaciones y replicar los experimentos descritos, obteniendo los mismos resultados.

4. Como se mencionó anteriormente, los *tries* son usados para auto completado. Para este último ejercicio, se debe implementar la operación:

   `virtual vector<string> getKTopMatches(const string &, int)=0`

   Esta operación recibe una cadena *str* y un entero k. Esta operación retorna un vector que contiene las k cadenas más frecuentes contenidas en el *trie* que comienzan con la subcadena *str*. Notar que para implementar este comportamiento, ahora la operación de inserción tiene que además

[1] `https://es.cppreference.com/w/cpp/container/map`

recibir un entero que indique la frecuencia asociada a la palabra recibida, y asimismo los nodos terminales deben almacenar la frecuencia de la palabra correspondiente.

Para el ejemplo de la Figura 1, la operación $getKTopMatches(\text{“}CA'', 2)$ retornaría un vector con las cadenas “CAMA” y “CASTOR” (asumiendo que estas dos cadenas fueron insertadas con una frecuencia mayor a la de la cadena “CAMPO”).

Observación

Los estudiantes pertenecientes al minor son libres de implementar las soluciones en el lenguaje de programación *C++*, *Java* o *Python*. En el caso particular de Python, la segunda variante utilizará *dict*[2] en sustituto a *map* de C++[3]. En Java se utilizará *TreeMap*[4].

3. Evaluación

El proyecto se realizará en parejas. Se debe subir a Canvas lo siguiente:

1. Un informe que:
   a) Incluya portada, descripción de la tarea, descripción de las soluciones propuestas, detalles de implementación, análisis teórico y análisis experimental.
   b) Sea claro y esté bien escrito. Un informe difícil de entender será mal evaluado, aunque todo esté bien implementado. Quien revise el documento debe poder entender su solución solo mirando el informe.
   c) Esté en formato pdf.
2. Un archivo comprimido con todos los ficheros fuente implementados para solucionar la tarea. El informe debe hacer referencia a ellos y explicar en qué consiste cada uno.

Fecha de entrega: viernes 3 de junio de 2022 11:59PM

[2] `https://docs.python.org/3/tutorial/datastructures.html`

[3] Notar que el diccionario implementado en Python con la clase *dict* funciona a través de tablas hash, mientras que el map de C++ utiliza árboles de búsqueda. Para efectos del análisis teórico, considerar como si se utilizaran árboles de búsqueda, a pesar de que esto no sea cierto en la implementación.

[4] `https://www.geeksforgeeks.org/treemap-in-java/`