75.19 - ***Teoría de Comunicación***

1er Cuatrimestre 2009

Universidad de Buenos Aires

Facultad de Ingeniería

Primer Cuatrimestre del 2009

Grupo: 11

Alumnos:

Víctor Giordano 83862

Marina Romero 83841

Tema: Investigación sobre la eficiencia de los árboles trie

Paper: A compact static double-array keeping character codes

Autores: Susumu Yata , Masaki Oono, Kazuhiro Morita, Masao Fuketa, Toru Sumitomo, Jun-ichi Aoe

Índice

Metodología de la Investigación

Nuestra metodología consistió en hacer una lectura profunda con un posterior relevamiento del artículo científico, obteniendo claramente las **hipótesis** y **conclusiones** de los autores. Finalmente nuestro objetivo apuntó a comprobar y analizar algún aspecto de ellas, siguiendo las mismas hipótesis y elaborando alguna aplicación informática al respecto, sobre la cual **amparar nuestras conclusiones personales.**

Objetivo de la Investigación

La investigación se sitúa entorno a la implementación de árboles de recuperación de texto llamados Trie.

Los autores del paper proponen una estructura particular de este tipo de árbol, llamada "Trie Con Arreglo Doble"- en inglés "Double Array Trie"- del cual alegan una muy buena performance en términos de tiempo de búsqueda y eficiencia en términos de espacio.

Sobre esta estructura se basan para crear un nuevo Trie llamado "Reduced Double Trie" (abreviado "Reduced Trie") que disminuye aún más el uso de memoria, y finalmente sobre éste realizan operaciones de compactación aumentando su eficiencia en términos de espacio. La salvedad de este tipo de estructura, es que no es apropiada para actualizaciones dinámicas.

Nuestro objetivo, fue comprobar la eficiencia de dicho árbol y su posterior compactación. Para ello, acorde a las pautas del Trabajo, realizamos una aplicación gráfica con el objetivo de comprobar dicha eficiencia mediante una comparación con otras estructuras de datos comunmente usadas para almacenar cadenas de textos.

Aplicación Informática

Desarrollamos una aplicación que permite comparar distintas estructuras de datos para el propósito de almacenar cadenas de textos. Ésta muestra en pantalla los resultados de pruebas para que el usuario pueda verificar nuestras conclusiones.

La comparación se realiza entre el **"Double Array Trie"**, **"Reduced Double Array Trie"**, **"Descent Trie"** (Trie compactado), **"List Trie"** (Trie con listas) e **"Indexed Array"** (Arreglo estático indexado con búsquedas binarias). Siendo los dos primeros los propuestos en el paper.

Introducción a la Investigación

El "Double Array Trie" es una implementación de un árbol de búsqueda TRIE propuesta por un grupo de profesores Japoneses (los nombrados al comienzo) en 1992.

¿Qué es un Trie?

Es un árbol n-ario que se usa para almacenar palabras.

La estructura con la se fundan los árboles Trie consiste en tener cada "n" nivel, nodos que contengan las "n-ésimas" letras de las palabras almacenadas, pudiendo haber nodos compartidos que contengan una letra común para dos palabras o más sólo si las palabras comparten dicha letras hasta la misma posición.

Por ejemplo: Azar y Amargo, comparten la "A" hasta la primera posición. La "a" está en la tercera posición de ambas palabras, pero no comparten las letras anteriores.

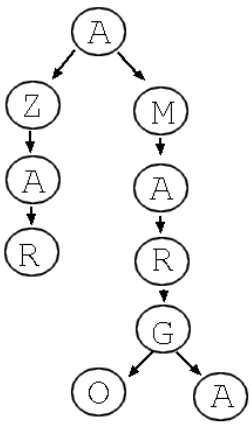
Desde la Raíz, usando cada letra de la palabra se puede ir navegando hasta el nodo que tiene la última letra de la palabra, y de esa forma determinar la existencia o no de una palabra.

A cada palabra la conocemos como una clave, esta clave puede tener asignados ciertos datos asociados.

En este caso el Trie se convertiría en una especie de mapa de claves (o diccionario).

Es decir, que dada una clave existente entonces puedo recuperar valores asociados. Cada clave puede existir a lo sumo una sola vez en el mapa.

Por ejemplo, dadas las palabras Amar, Amargo, Amarga y Azar.



Notar que nos estaríamos ahorrando espacio al poner letras que comparten las palabras en las mismas posiciones y que la navegación estaría dada por la misma palabra. Son estos, justamente, los dos fundamentos de un árbol Trie:

* Evitar guardar redundancias de datos

* Verificar rápidamente palabras almacenadas.

La descripción de un árbol Trie no basta por si sola para definir unívocamente la forma óptima de implementarlo. A lo largo del tiempo han surgido diversas formas y estructuras para implementarlo. Por este motivo nos pareció oportuno mostrar desde la más nativa hasta la planteada en este artículo, con motivo de compararlas.

Implementación con arreglos

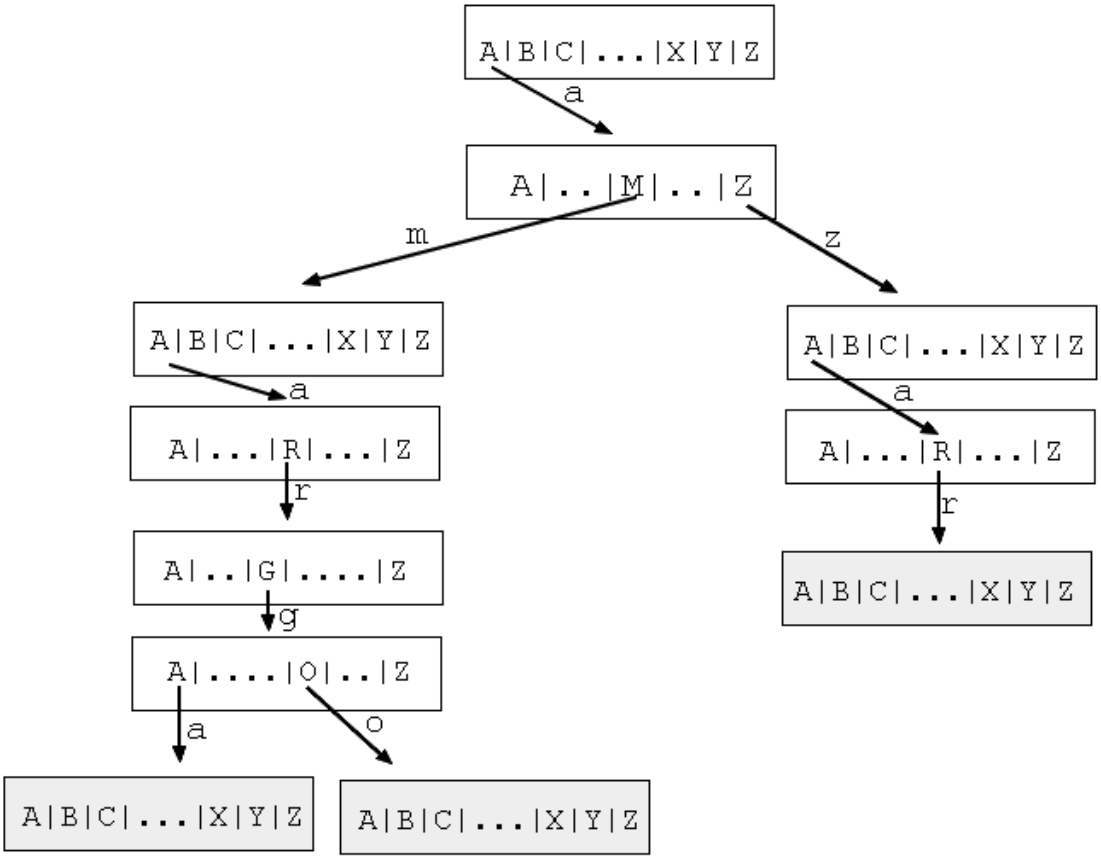
En este tipo de estructura, cada nodo contiene un arreglo de punteros a otros nodos de un tamaño equivalente a la cantidad de letras podría tener una palabra en cada posición. Podemos considerar a cada letra compuesta por un carácter ASCII (256 símbolos) y que de este se puede llegar a otro nodo según la letra que le siga. De esta forma el arreglo que tiene un nodo es de 256 punteros a próximos caracteres.

Dado que cada nodo procede de una letra y a esta letra le pueden suceder otros 256 caracteres, se define formalmente para cada nivel una cantidad máxima de $256^{(n-1)}$ nodos(n = [1,.., k]).

Para el nivel 1 existirá 1 nodo (con su arreglo de 256 punteros) pues cada palabra empieza con alguno de los 256 caracteres).

Para el nivel 2, se pueden tener 256 caracteres que sucedan a la “a”, otros 256 que le sucedan a la “b”, y así con todos los símbolos. Entonces tenemos 256 nodos (cada uno su arreglo de 256 punteros), es decir, todas la combinaciones posibles en cada nivel.

Naturalmente se opta por crear cada nodo de forma dinámica, evitando crear nodos de más.



El índice del arreglo de un nodo se corresponde con el código ASCII de un carácter. La idea es que con el n-ésimo carácter de cada palabra podemos ir directo (por ser un arreglo indexado) a la posición del puntero correspondiente a ese carácter. Si no es nula la referencia es porque esa letra está almacenada y le sucede a otro nodo. De esta manera se avanza un nivel de profundidad (A nivel n+1).

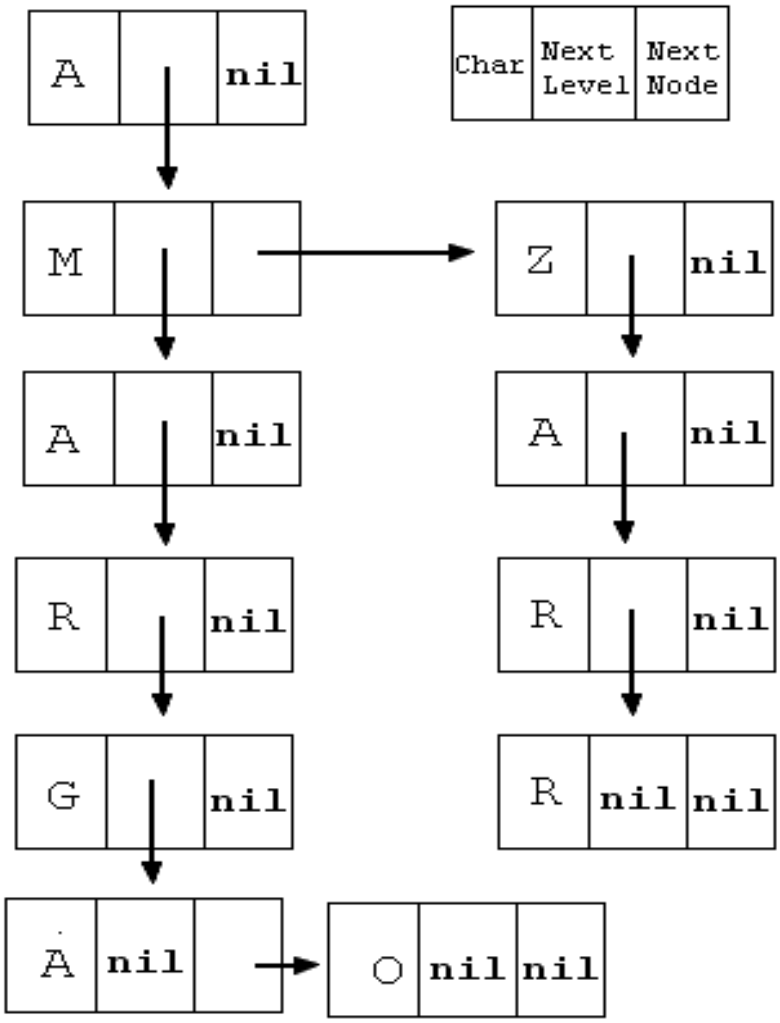
Lógicamente la búsqueda se hace empezando siempre por la raíz usando el primer carácter de la palabra. No se puede empezar una búsqueda por la mitad del arreglo pues claramente, no hay forma de llegar por medio de punteros hasta allí.

Los nodos hojas, son nodos con sus arreglos de punteros solamente inicializados en "null", deben estar para indicar que la palabra termina allí.

Esta implementación, es sencilla y muy rápida pero peca por el excesivo espacio que ocupa cada nodo.

Implementación con listas

En esta estructura cada nodo contiene una letra, un puntero al próximo nivel y un puntero a la próxima letra dentro de ese nivel.



Esta estructura permite ahorrar una cuantiosa cantidad de memoria comparada con la anterior, al no inicializar los 256 "posibles punteros" a próximos caracteres para los caracteres existentes.

Se evita crear nodos muy pesados como los anteriores pero la desventaja radica en que la búsqueda se vuelve considerablemente más lenta al tener que navegar entre listas. Esto provoca que la rapidez disminuya ampliamente en comparación con la de los arreglos indexados.

Implementación con arreglo doble (Double Array TRIE)

Descripción

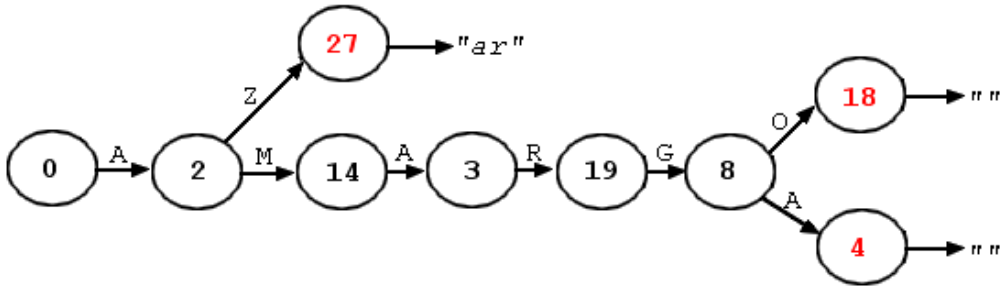
Esta es justamente la estructura propuesta por este grupo de investigadores de Japón. La idea es ir guardando los prefijos comunes de las palabras en un árbol de navegación implementado con un arreglo y a los sufijos excluyentes de las palabras guardadas en un buffer de memoria al que se llama "cola".

Esta estructura sólo mantiene la mínina cantidad de nodos necesarios correspondientes a los sufijos de las palabras o claves almacenadas, y los sufijos excluyentes de cada palabra los alberga en la cola. En particular ésta es una mejora del doble arreglo original, que su autor denominó "Reduced Trie".

En base a los lineamientos del paper desarrollamos una implementación acorde , la cual pasaremos a describir con un ejemplo .

Suponiendo que se ingresan en el arreglo las palabras : azar, amarga y amargo .

Este sería el resultado:



Arreglo de prefijos:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15...17	18	19	20....26	26	27
Base	1	0	1	1	-9	0	0	0	3	0	0	0	0	0	2	0	-4	1	0	0	0
Check	0	-	0	14	8	-	-	-	19	-	-	-	-	-	2	-	8	3	-	-	2
Terminal	f	f	f	f	t	f	f	f	f	f	f	f	f	f	f	f	t	f	f	f	f

Tabla de códigos:

Cáracter	a	g	m	o	r	z
Valor	1	7	13	15	18	26

Cola = "ar|?????|"

Posición de cola = 10

Máximo Nodo = 27

Cada nodo tiene una base, que en caso de ser mayor a cero, actúa como "offset" al lugar en el cual se encuentran los nodos hijos, de forma tal que los pueda acceder usando esa base + el código del carácter que tiene la tabla. En caso de ser menor o igual a cero, indica a partir de que posición se encuentra el sufijo en la cola, hasta el carácter especial de fin de palabra **"|"**.

El campo check, es un puntero al nodo padre . Éste sirve para verificar a medida que se recorre el árbol, que a tal nodo se llegó a partir de un nodo tal, y que ese camino realmente existe.

El campo terminal indica si un nodo es terminal o no. Un nodo terminal informa que el conjunto de letras que identifica a los arcos a los que se fue navegando conforman una palabra existente en la estructura. Si no es terminal, entonces la palabra que se "va formando" no esta contenida.

Notar que: (Entre paréntesis esta el código de carácter)

De Nodo: 0 (base = 1)→ Con 'a' (1) se va a nodo 2

De Nodo: 2 (base = 1)→ Con 'm' (13) se va a nodo 14, Con 'z' (26) se va a nodo 27

De Nodo: 14 (base = 2)→ Con 'a' (2) se va a nodo 3

De Nodo: 3 (base = 1)→ Con 'r' (18) se va a nodo 19

De Nodo: 19 (base = 1)→ Con 'g' (18) se va a nodo 8

De Nodo: 8 (base = 3)→ Con 'a' (1) se va a nodo 4, Con 'o' (15) se va a nodo 18

De Nodo: 4 (base = -9). Es Hoja.

De este nodo se obtiene: ""

De Nodo: 18 (base = 0). Es Hoja.

De este nodo se obtiene: ""

De Nodo: 27 (base = 0). Es Hoja.

De este nodo se obtiene: "ar"

La idea es que **un nodo no representa una letra**, sino que **la letra representa el arco entre dos nodos**, es decir aquel valor que permite moverse entre dos nodos.

Es muy importante tener la idea de que aunque dos nodos tengan la misma base, no significa que están apuntando a los mismos nodos hijos. La base por si sola no indica nada, hace falta tener el cuenta el carácter con el que se está trabajando, en particular el código numérico asociado al mismo.

Los nodos hijos de un nodo, se obtienen sumando **la base más el código del carácter**, y la idea es intentar usar todas las posiciones posibles del arreglo, sin dejar brechas. Si se da el caso de que dados dos nodos cualesquiera se los puede referenciar desde una misma base más el código de carácter (con el cual se puede acceder a ellos), entonces se usa esta misma base con la idea de ocupar los 256 nodos hijos posibles con las "más próximas" 256 posiciones libres del arreglo. Esto es algo totalmente dependiente del juego de palabras con el que se trabaje y se vaya agregando al Trie.

En este ejemplo, los nodos 0, 2, 3 y 19 comparten la base 1. De ellos con los caracteres que permiten navegación se da siempre que la base (1) + código de carácter va a referenciar una posición distinta del arreglo.

Nodo 0 (base = 1) : 'a' (1+1) → nodo 2

Nodo 2 (base = 1) : 'm' (13+1) → nodo 14, 'z' (26+1) → nodo 27

Nodo 3 (base = 1) : 'r' (18+1) → nodo 19

Nodo 19(base = 1) : 'g' (18+1) → nodo 8

De esta forma se maximiza el uso del espacio de los elementos del vector. Esta idea es fundamental y es lo que marca una gran diferencia con la implementación con arreglos. La razón es que si para cada nodo se reservaran los 256 elementos próximos adyacentes para los posibles nodos sucesivos, entonces se estaría realizando algo similar a la implementación con arreglos (ocupando nuevamente exceso de memoria).

En esta implementación la idea es usar los elementos del arreglo adyacentes; esto irá creando situaciones problemáticas e inconsistentes que se explicarán más adelante, así como también la forma de resolverlas.

Búsqueda

La búsqueda de una palabra es bastante lineal. Se parte del nodo inicial y el primer carácter de la letra.

Luego comienza un ciclo en el cual se obtiene la base del nodo actual, a la cual se le suma el código de carácter tomado y se verifica si existe una arista desde el nodo destino al nodo actual. Si es así se establece como nodo actual al nodo al que se llega con la suma y se toma el próximo carácter del arreglo.

La condición de fin se puede dar en dos oportunidades:

- Cuando se toma una base menor o igual a cero y se verifica que lo que está en la cola concuerda con lo que queda de la palabra buscada (la palabra menos los primeros caracteres con los cuales se hizo la navegación).

- Si ya se recorrieron todos los caracteres de la palabra, y aún se está en un nodo, entonces se verifica si este es un nodo terminal. Si es así, la palabra existe, sino no.

```
boolean SEARCH (String str)
{
    int curNode = ROOT_NODE;
    int i = 0,len = str.length();
    String strTail;
    boolean end = false;
    boolean find = false;

    while (!end)
    {
        int curBase = this.nodes[curNode].base;
        if (i == len)
        {
            find = this.nodes[curNode].isTerminal;
            end = true;
        }

        if (curBase <= 0 && !find)
        {
            int index = -curBase;
            strTail = tailRetrieve(index);
            find = strTail.equals(str.substring(i));
            end = true;
        }
        else if (i < len)
        {
            int nextNode = curBase + this.charCodes[str.charAt(i)];
            if (this.nodes[nextNode].check != curNode)
            {
                end = true;
            }
```

```
                else
                {
                        curNode = nextNode;
                        i++;
                }
        }
        else
        {
                end = true;
        }

    }
    return find;
}
```

Inserción

La inserción es uno de los temas más complejos. Se pueden dar cuatro casos de inserción:

- Inserción cuando esta vació el Trie
- Inserción sin colisiones
- Inserción con colisiones en la cola
- Inserción con conflicto en árbol
- Inserción implícita

Todos los casos tienen una parte inicial en común, en donde de manera muy similar a la búsqueda se recorre el árbol buscando el elemento a insertar.

```
void INSERT (String str)
{
    int curNode = ROOT_NODE, i = 0;
    boolean end = false;
    while (!end)
    {
        int curBase = this.nodes[curNode].base;
        if (curBase <= 0)
```

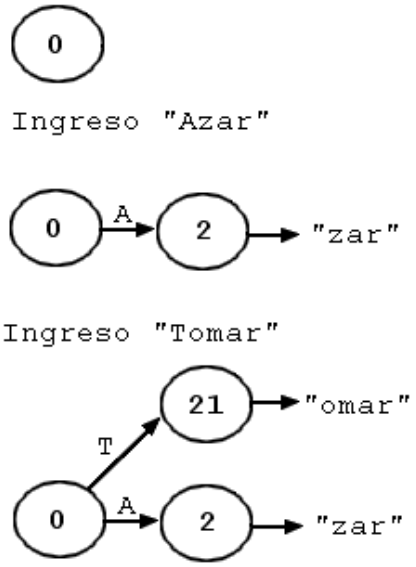
```
{
        // INSERCION CON COLISION EN COLA
        insert_2(str, i, curNode, curBase);
        end = true;
}
else if (str.length() == i)
{
        // INSERCION IMPLICITA
        this.nodes[curNode].isTerminal = true;
        end = true;
}
else
{
        int nextNode = curBase + this.charCodes[str.charAt(i)];
        if (this.nodes[nextNode].check != curNode)
        {

                if (this.nodes[nextNode].check == UNDEFINED)
                {
                        // INSERSION SIN COLISION
                        insert_1(str, i, curNode, nextNode);
                        end = true;
                }
                else
                {
                        // INSERCIÓN CON COLISION EN EL ARBOL
                        insert_3(str, i, curNode, nextNode);
                        end = true;
                }
        }
        else
        {
                i++;
                curNode = nextNode;
        }
```

```
            }
        }
    }
```

Donde insert_X, el "X" indica a que caso corresponde la inserción. El caso 1 y 2 son operacionalmente lo mismo, por eso se denota como 1_2.

Inserción 1 y 2.

Cuando el árbol esta vació o no hay colisiones de ningún tipo en la navegación, la inserción es bastante lineal, solamente se agrega un nodo al final, se guarda en el buffer de sufijos (la cola) los caracteres restantes del arreglo y se establece que la base de nodo "apunte" a esa posición de la cola.



Cola = "zar|omar|"

Posición Cola = 9

	0	1	2	3	.....	21
Base	1	0	0	0	0	-4
Check	0	-	0	-	-	-
Terminal	f	f	t	f	f	f

(Los códigos son los mismos que los puestos anteriormente), si verificamos:

De Nodo: 0 (base = 1)→. Con a(1) se va a nodo 2, con t(20) se va a nodo 21

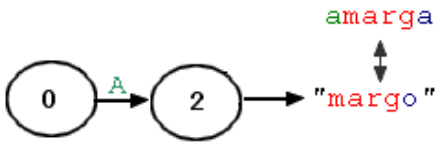
De Nodo: 2 (base = 0)→. Es Hoja y se obtiene "zar".

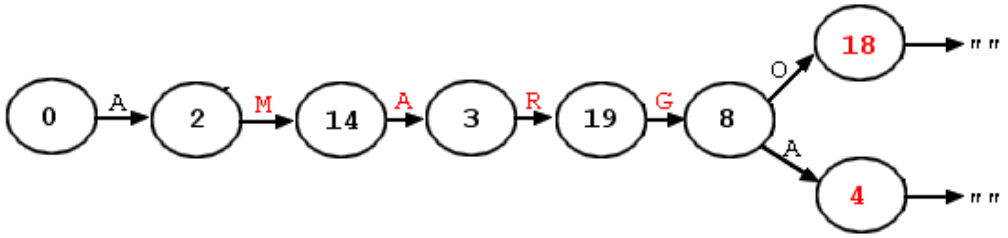
De Nodo: 21 (base = -4)→. Es Hoja y se obtiene: "omar".

Inserción 3

En el caso de la inserción 3, la colisión se da en la cola, es decir, al ir navegando por el árbol, se llega a un lugar donde se debería tener almacenado en la cola una porción de un string que no es coincidente con la porción del string que se está ingresando.

Se ingresan al árbol los nodos correspondientes a los caracteres del prefijo común a ambas, y luego se establecen los nodos hojas apuntado al lugar de la cola correspondiente donde se almacenan los sufijos exclusivos de cada uno.





	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Base	1	0	1	1	-6	0	0	0	3	0	0	0	0	0	2	0	0	0	0	1
Check	0	-	0	14	8	-	-	-	19	-	-	-	-	-	2	-	-	-	8	3
Terminal	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f

Cola = |argo| | ("argo" es basura)

Posición Cola = 7

De Nodo: 0 → Con a se va a nodo 2

De Nodo: 2 → Con m se va a nodo 14

De Nodo: 14 → Con a se va a nodo 3

De Nodo: 3 → Con r se va a nodo 19

De Nodo: 19 → Con g se va a nodo 8

De Nodo: 8 → Con a se va a nodo 4, Con o se va a nodo 18

De Nodo: 4 → Es Hoja y se obtiene: ""

De Nodo: 18 → Es Hoja y se obtiene: ""

Como se puede observar, entre amargo y amarga ya esta incorporada la primera "a" al árbol, pero se debe ingresar también la "m", la "a", la "r" y la "g". Por último los nodos hojas "o" y "a" apuntarán a un lugar en la cola de sufijos de forma tal que concatenando se forme la palabra completa. Siempre se toma el prefijo común a ambas, y las porciones de sufijos exclusivas de cada uno.

En este caso, se observa que los sufijos son "" (cadenas nulas) puesto que entre **amarg**o y **amarg**a, los sufijos excluyentes son "o" y "a" respectivamente y son letras que se usan para distinguir una palabra de otra (lo que se observa en el gráfico). No serán almacenadas en la cola de sufijos, porque la primera letra del sufijo de una palabra es la que se requiere para caer en un nodo hoja para acceder al resto del sufijo excluyente. Esto se debe a que son dos palabras cuyo sufijo exclusivo es la última letra. En estos casos, lo único que se puede hacer es guardar todas las letras, marcando el nodo correspondiente a la última letra como terminal y poniendo como sufijo una cadena nula (ya que no se le agrega nada más a la palabra).

Este caso tiene dos *subcasos* particulares que se explican brevemente a continuación:

- La inserción de palabras que <u>están implícitamente contenidas por otra existente</u>, (por ejemplo ingresar "cad" existiendo "cadena")
- La inserción de palabras <u>que contienen a otra ya existente,</u> (por ejemplo ingresar "cadena" existiendo "cad".

El algoritmo tiene sutiles diferencias operacionales.

De Nodo: 0 (base = 1)→. Con a(1) se va a nodo 2, con t(20) se va a nodo 21

De Nodo: 2 (base = 0)→. Es Hoja y se obtiene "zar".

De Nodo: 21 (base = -4)→. Es Hoja y obtiene: "omar".

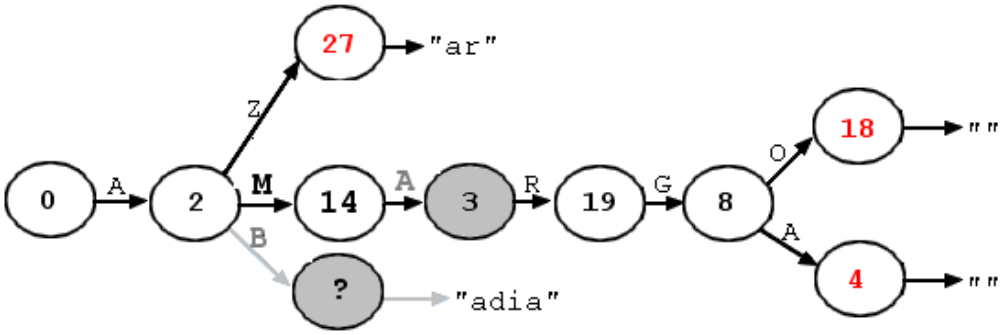
Inserción 4

Como se anticipó anteriormente, el tema de conflicto es para tener en cuenta. Se puede observar qué pasaría si sobre el trie que se tenía insertado "azar" "amargo" y "amarga" se intenta ingresar "abadía".

Con la primera "a", se va al nodo 2, y luego desde este se obtiene una "b" cuyo código es 2, y la base (2) + código ('b') = 1 + 2 = 3. Pero este nodo ya existe, es al que se

llega con la segunda "a" de am**a**rgo (o am**a**rga) y se sale con la respectiva **r.** Entonces en este punto efectivamente en el afán de querer utilizar los elementos adyacentes, ya no se tiene espacio para salir del nodo 2 con la letra hacia donde corresponda, porque se estaría yendo al nodo 3, del cual se sale con la letra r. Claramente no se está queriendo ingresar una palabra que comience con "abr.." sino una que comience con "aba…". Por lo tanto se deberá reestructurar el arreglo, consiguiéndole a los elementos en el conflicto nuevos lugares .

Graficando :

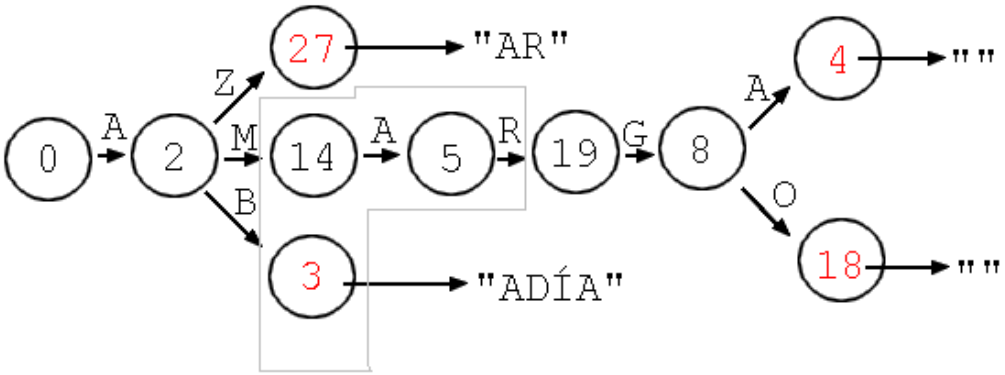


En estos casos el enfoque debe estar en el nodo anterior (14) al nodo donde ocurrió el conflicto (3), puesto que se debe alterar su base para que con su arco saliente identificado con la letra "A" no lleve al nodo (3) que es conflictivo.

Además ahora se debe lograr que el nodo 2 que tenía un arco saliente con la "M" hacia el nodo 14 y un arco saliente con la "Z" hacia el nodo 27, tenga además, un arco saliente nuevo: uno con la "B" hacia un nodo a asignar.

Es decir, del nodo 2, se tiene que construir un nuevo camino (para el string insertado "abadía") y conservar los dos anteriores (o sea los arcos de la "Z" y "M").

La resolución es la siguiente:



Notar que el a partir del nodo 2, navegando con la letra B ahora se va hacia el nodo el nodo correcto (3). Lo que era antes el nodo 3, ahora es el nodo 5. Esto se debe a que en nodo el 14 la base fue modificada, para que con la "A" no se llegue al nodo 3, sino justamente al nodo 5. Al cambiar la base del 14, entonces todos sus arcos salientes ahora van a apuntar a distintos nodos, y esa es justamente la idea para librarse de este conflicto.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20.. 26	27
Base	1	0	1	-10	6	1	0	0	3	0	0	0	0	0	4	0	0	0	0	1	0	7
Check	0	-	0	2	8	4	-	-	9	-	-	-	-	-	2	-	-	-	8	5	-	2
Leaf	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	t	f	f	f

Cola = |argo| |ar|adía|

Posición Cola = 15

De Nodo: 0 → Con a se va a nodo 2

De Nodo: 2 → Con b se va a nodo 3, m se va a nodo 14, z se va a nodo 27

De Nodo: 3 → Es Hoja y se obtiene: adía

De Nodo: 14 → Con a se va a nodo 5

De Nodo: 5 → Con r se va a nodo 19

De Nodo: 19 → Con g se va a nodo 8

De Nodo: 8 → Con a se va a nodo 4, o se va a nodo 18

De Nodo: 4 → Es Hoja y se obtiene: ""

De Nodo: 18 → Es Hoja y se obtiene: ""

De Nodo: 27 → Es Hoja y se obtiene: "ar"

Es oportuno esclarecer bien el concepto de **nodos terminal, nodos comunicantes, nodos hojas y arcos.**

No deben confundirse los **arcos** con los **nodos**. Las palabras se van formando con las **letras** de los **arcos** (transiciones), y los **nodos** (estados) indican en que **contexto** se está situado.

Contextos de los nodos:

Si (base > 0 y es nodo terminal) → indica que con las letras usadas para navegar hasta allí tenemos una palabra ingresada (la palabra se construye con las letras de los arcos navegados)

Si (base > 0 y no es nodo terminal) → indica el medio de una palabra. (las letras de los arcos por los cuales se viene navegando forman parte de una palabra insertada)

Si (base < 0 y no es nodo terminal) → indica un nodo del cual hay un sufijo en la cola de sufijos. (se puede construir una palabra con las letras de los arcos + el sufijo de este nodo)

Si (base < 0 y es nodo terminal) → indica un nodo del cual hay un sufijo en la cola de sufijos y además la palabra formada también esta agregada a la estructura. (Se puede construir una palabra con las letras de los arcos y otra palabra con las letras de los arcos + el sufijo de este nodo).

Los nodos **hojas** son los del 3er y 4to tipo. (No se puede navegar hacia delante, pues allí termina el árbol)

Los nodos **comunicantes** son los del 2do tipo. (Se puede navegar)

Los nodos **terminales y comunicantes** son los del 1er tipo (Se puede navegar, puesto que son comunicantes, pero también pero obtener una palabra, conformada por concatenar las letras identificadoras de los arcos que llevan a este nodo).

El secreto en el agregado está en la implementación de una sencilla función que busca la base que sirve para referenciar a un conjunto de nodos libres a partir de un conjunto de caracteres. Es decir, la base "X" de forma tal que "X" + código ("c") = elemento libre del arreglo (para todo "c" del conjunto de caracteres). Esta función es la llamada x_check y es gran parte a la que se debe el uso eficiente de los elementos del arreglo.

Inserción 5

La última inserción es realmente la más sencilla de todas. Ocurre cuando en el árbol hay nodos encadenamos de forma tal que la palabra a insertar no requiere agregar nuevos nodos a la estructura, sino simplemente marcar cual es el nodo en el cual se debe terminar de hacer el recorrido de búsqueda.

Este marcado de forma especial sirve para identificar cuales nodos son terminales y cuales no, aún no estando como nodos hojas.

<INSERTAR GRÁFICO>

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Base	1	0	1	1	-6	0	0	0	3	0	0	0	0	0	2	0	0	0	0	1
Check	0	-	0	14	8	-	-	-	19	-	-	-	-	-	2	-	-	-	8	3
Leaf	f	f	f	f	t	f	f	f	f	f	f	f	f	f	f	f	f	f	t	**T**

Cola = |argo| | (“argo” es basura)
Posición Cola = 7

De Nodo: 0 → Con a se va a nodo 2.
De Nodo: 2 → Con m se va a nodo 14.
De Nodo: 14 → Con a se va a nodo 3.
De Nodo: 3 → Con r se va a nodo 19.
De Nodo: 19 → Es **Terminal** y con g se va a nodo 8.
De Nodo: 8 → Con a se va a nodo 4, Con o se va a nodo 18.
De Nodo: 4 → Es Hoja y se obtiene: “”.
De Nodo: 18 → Es Hoja y se obtiene: “”.

Borrado

El borrado de elementos en el Trie se describe de esta forma. Esto no estaba planteado en el artículo original, por lo cual es una implementación propuesta.

```
boolean DELETE (String str)
{
        int curNode = ROOT_NODE;
        int i = 0,len = str.length();
        boolean end = false, find = false;
        String strTail;

        while (!end)
        {
                int curBase = this.nodes[curNode].base;
                int lastNode = this.nodes[curNode].check;
                if (i == len)
                {
                        find = this.nodes[curNode].isTerminal;
                        end = true;
                        if (find)
                                this.nodes[curNode].isTerminal = false;
```

```
}

if (curBase <= 0 && !find)
{
	int index = -curBase;
	strTail = tailRetrieve(index);
	find = strTail.equals(str.substring(i));
	end = true;
	if (find)
	{
		if (this.nodes[curNode].isTerminal)
		{
			this.nodes[curNode].base = -tailPos;
			this.tail.insert(this.tailPos, END_CHAR);
			this.tailPos += 1;
			this.nodes[curNode].isTerminal = false;
		}
		else
		{
			this.nodes[curNode].check = UNDEFINED;
			this.nodes[curNode].base = 0;
			do_back_delete(lastNode);
		}
	}
}
else if (!end)
{
	int nextNode = curBase + this.charCodes[str.charAt(i)];
	if (this.nodes[nextNode].check != curNode)
	{
		end = true;
	}
	else
	{
		curNode = nextNode;
		i++;
	}
```

```
                }
                else
                {
                        end = true;
                }

        }
}
```

El borrar hacia atrás lo que hace es ir recorriendo hacia atrás los nodos, cosa que se puede hacer mediante el campo check, e ir viendo sino tienen hijos, y en ese caso los va marcando libres.

```
// se va borrando hacia atrás
last = node;
Mientras (node != NODO_RAIZ && !isTerminal (node) &&!tieneHijos(node))
{
    last = getCheck(node);
    setBase(node,0);
    setCheck(node,UNDEFINED);
    node = last;
}
// si llege hasta el nodo raiz y no tiene hijos, entonces queda vacío
Si (node == NODO && !tieneHijos (node))
{
    setBase(node,1);
    setCheck(node,0);
}
```

Con esto se concluye la explicación general l funcionamiento de la búsqueda, inserción y borrado sobre un árbol Trie implementado con una estructura de doble arreglo.

Suffix Trie

Una Variante del DATrie se llama Suffix Trie, consiste en eliminar el buffer de caracteres para guardar los sufijos de las palabras e incluir en cada nodo una referencia (o puntero) a un tipo de dato que pueda guardar longitud arbitraria de caracteres

(generalmente: el tipo de dato "String"). Esta implementación más modesta tiene sus pros y sus contras.

Diferencias entre el Suffix Trie y el DATrie

A continuación nos enfocaremos en las diferencias, entre ambos tipos de estructuras, desde diferentes perspectivas.

Estructurales

En un trie de **sufijos** la única diferencia que se tiene es que no tiene una **cola de sufijos**, sino que en cada **nodo** tiene un **puntero** a su sufijo exclusivo.

Los nodos de ambas estructuras conservan los campos "**check**", "**isTerminal**" y "**base**".

El campo "**base**" tiene un valor que para los nodos **comunicantes (sea o no terminal) [1 y 2 contexto]** indica cual es el offset a los nodos que le siguen. Pero en el caso de que el nodo fuera **terminales solamente. [3 y 4 contexto]**

DATrie: Su valor será negativo, e indica en qué lugar de la cola esta el sufijo (o resto de la palabra).

SuffixTrie: Su valor no importa de ese nodo tenemos un puntero que no es nulo y referencia al sufijo exclusivo. (Para la implementación propuesta el campo base vale "-1", pero puede ser otro negativo arbitrario)

Si se observan los gráficos anteriores, basta con imaginarse que no hay cola y que los nodos hoja tienen una base igual a "-1".

Funcionales

No hay diferencias funcionales a nivel usuario, ambas estructuras de datos funcionan de forma similar para los usuarios.

A nivel implementación la diferencia es menos sutil, puesto que se debe ir llevando y actualizando la cola de sufijos (potencialmente un buffer de bytes que sirva para almacenar los caracteres).

De rendimiento

DATrie:

👍 Ocupa menos memoria.

👎 Hay que liberar la memoria no usada en la cola

👎 Presencia de Carácter delimitador:

👎 Incrementa complejidad en la búsqueda, al tener que escapearlo de alguna manera.

👎 Hace un poco más lenta la última parte de la búsqueda, pues debe extraer del arreglo "cola" la cadena en cuestión. Una operación mínima de tiempo, pero dado lo reducida que es la búsqueda puede tener cierta significación.

👍 Muy Fácil de serializar para guardar en el almacenamiento secundario, es prácticamente guardar los 2 arreglos tal como están.

SuffixTrie:

👎 Ocupa más memoria.

👍 No hay que liberar memoria de la cola (pues no existe)

👍 No existe Carácter delimitador

👍 Búsquedas un poco más rápidas.

👍 Se puede trabajar tranquilamente con cualquier tipo de cadena sin considerar la cuestión del carácter delimitador.

👎 Es levemente más difícil de serializar para guardar en el almacenamiento secundario.

(Notar que lo positivo en una estructura se muestra como negativo para la otra)

Consideraciones no planteadas en el Paper

(Y necesarias para efectivizar una implementación)

En el siguiente apartado se comentarán las cuestiones rozadas tangencialmente o ignoradas en el paper que estudiamos y en otros papers de los mismos autores a los cuales recurrimos para obtener más información. (Ver bibliografía.)

Inserciones implícitas

Agregar un campo de isTerminal y marcarlo cuando se de el caso de inserción 5.

Como marcar elementos vacíos

Mediante el campo check, se usa un valor definido arbitrariamente (cualquier número negativo basta).

Arreglo de códigos de caracteres

Se crea un arreglo de 256 posiciones para asignarle a cada posible carácter ASCII un código numérico. Se usa como índice el código número del carácter ASCII. Al leer los archivos con *stream* se tiene que tener cuidado de leer siempre de a un byte usando una codificación de 8 bits.

Borrados

Se implementó el algoritmo de borrado, tal como está explicado más arriba.

Carácter Delimitador

Si se usa un carácter delimitador, entonces se debe tener en cuenta que es necesaria alguna manera de "**escapearlo**" cuando se recibe una palabra de este tipo. Esto se logra haciendo un **escapeo** del carácter **escapeador**.

Tamaño del arreglo de nodos

El tamaño del arreglo inicial esta dado para ocupar aproximadamente unos 512kb de memoria, algo bastante razonable para almacenar de pocas y mediana cantidad de palabras claves.

Se creó una variable de control llamada "*max_node*" que indica cual es el último nodo que tiene datos. Al realizar inserciones se verifica que el *max_node* + *256* no supere a la cantidad de elementos en el arreglo, sino supera entonces no se incrementa. Si supera se incrementa en un factor de 50% su capacidad.

De esta forma se evitan los desbordes por derecha.

Compactación del DATrie

Existen dos maneras de compactar el DATrie. La primera consiste en una compactación de los elementos, es decir reducir el espacio que ocupa cada nodo. La segunda consiste en disminuir ya sea la cantidad de elementos del trie , o la cantidad de elementos de la cola.

Compactación de Elementos

Teniendo en cuenta que el campo CHECK es utilizado únicamente para actualizaciones dinámicas, si éstas no son requeridas puede dejar de almacenarse en este campo el puntero al padre. En su lugar se almacenará el código del carácter, de manera de compararlo con el código que se intenta conseguir, en el algoritmo de búsqueda. Se le llamará a este campo LCHECK.

Esta solución reduciría el espacio necesario para cada nodo considerablemente, ya que el puntero al padre ocupa 3 bytes y el código del carácter sólo uno.

Para lograr esto hay que tener en cuenta dos restricciones :

- Todos los elementos deben tener una base única. Excepto, claro está, en los que son hojas ya que la base no es utilizada para desplazarse al siguiente nivel.
- El campo LCHECK de los nodos que no son hojas debe almacenar un valor tal que no de aristas erróneas. Es decir un valor negativo servirá, ya que ningún código de carácter es negativo.

El algoritmo de búsqueda es modificado mínimamente , ya que ahora comparará el código que se está buscando con el campo LCHECK.

Compactación del Trie

Para compactar el Trie hay dos alternativas. Una es disminuir la cantidad de nodos o elementos del árbol, la otra es disminuir la cantidad de elementos de la cola.

Trie Descendiente

Para disminuir la cantidad de elementos del árbol se puede realizar un proceso de compactación que elimine aquellos nodos hojas que tienen un solo hermano y asigne como nuevo nodo hoja a su padre. Esto se logra haciendo que las aristas que llegan a dichos nodos hojas sean almacenadas como sufijos en la cola del Trie. A este tipo de árbol le llamaremos Trie descendiente.

Para construir este tipo de trie a partir de un trie reducido se desarrolló un algoritmo que realiza lo siguiente:

1. Recorrer el DATrie y obtener el subconjunto de hojas

2. Ordenar este subconjunto por el puntero al padre (campo CHECK)
3. Verificar cuáles son aquellas hojas que sólo tienen un hermano
4. Por cada par de hojas hermanas que cumpla estas condiciones:
    a. Encontrar la arista que llega a ellas
    b. Obtener los sufijos de las hojas hermanas
    c. Almacenar en la cola una tripla conformada de
        i. Prefijo común entre las hojas hermanas : se obtiene a partir de los dos sufijos a los que apuntan.
        ii. Longitud del primer sufijo, sin contar la longitud del prefijo común
        iii. Restante del primer y segundo sufijo
    d. Marcar como nodo hoja al padre de las hojas hermanas. Actualizar el link a la tripla de sufijos.
    e. Marcar como borradas las hojas hermanas

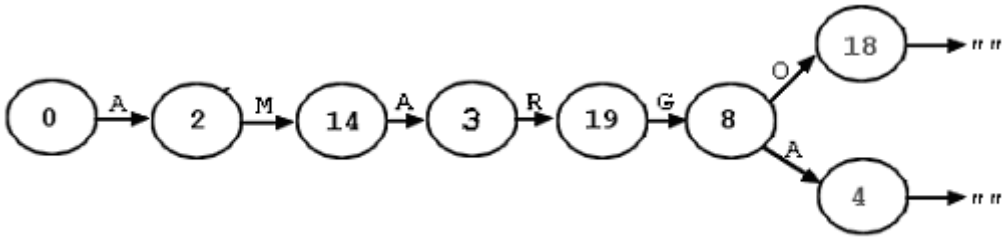
Además es necesario modificar mínimamente la estructura del nodo agregando un campo más llamado SPLIT que indique si el sufijo está o no dividido en la cola.

Las funciones de búsqueda son levemente modificadas ya que al comparar contra el sufijo debe verificarse si SPLIT es true, y de darse el caso comparar contra los dos sufijos almacenados en la cola.

A pesar de esta diferencia la eficiencia en la rapidez de la búsqueda no es alterada ya que el menor esfuerzo en la búsqueda se da justamente en los nodos hojas con un solo hermano.

Ejemplo :

Si se quisieran almacenar las palabras amarga y amargo, en el trie reducido existirían dos nodos hojas que tienen un solo hermano . Éstos son eliminados y el padre de ambos queda como nuevo nodo hoja apuntando a la tripla que contiene el prefijo común de ambos sufijos (vacío) , la longitud del primer sufijo (1) y la parte restante de ambos sufijos (A y O).



Sufijos Unificados

Otra manera de reducir espacio en el Trie es reduciendo espacio en la cola. Una manera sencilla de realizar esta tarea es eliminando los sufijos duplicados que estén almacenados en ésta. El algoritmo es tan sencillo como recorrer la cola en forma secuencial e ir agregando en una nueva cola cada sufijo, siempre y cuando éste no forme parte de la cola ya. Finalmente se hace que la nueva cola sea la cola actual.

Conclusiones

La conclusión que obtuvimos es que el "Reduced Double Array Trie" ciertamente es una estructura para almacenar cadenas de textos, **óptima tanto en términos de memoria como de tiempo de búsqueda.** Es mejor que su equivalente no reducido, el "Double Array Trie", en términos de memoria, pero en tiempos de búsqueda puede llegar a ser un poco menos eficiente.

Por otro lado las inserciones y actualizaciones se vuelven lentas y resulta ser una estructura con un tiempo de inserción muy elevado por sobre las otras.

La compactación final disminuye aún más el consumo de memoria pero el costo es que dicha estructura ya no puede admitir inserciones, borrados y/o actualizaciones de algún tipo.

También notamos que sería apropiado que el paper tratara en forma más detallada ciertos aspectos de la implementación de la compactación de árboles dinámicos.

Bibliografías Consultadas

An Efficient Implementation of Trie (Jun-ichi Aoe y Katsushi Morimoto)

An improvement key deletion method for double-arrays (Masaki Oono *, Masao Fuketa, Kazuhiro Morita, Shinkaku Kashiji, Jun-ichi Aoe)

A Method of Compressing Trie Structures. (Katsushi Morimoto, Hirokazu Iriguchi and Jun-ichi Aoe)

http://en.wikipedia.org/wiki/Trie