Tries y Variaciones

Francisco Mamani[1]*

Abstract
Para procesamiento de cadenas, o ”String Processing”, una buena estructura con la cual trabajar es el denominado ”Trie”, el cual alcanza tiempos de búsqueda del orden **O(n)**. En este articulo se revisaron los conceptos y características de esta estructura, también así como de algunas de sus variaciones, las cuales varían en términos de espacio requerido, debido a que la estructura ”Trie” es muy ineficiente en términos de complejidad de memoria, alcanzando un orden **O(HN)**, **H** siendo la cantidad de llaves que contiene el alfabeto con el cual trabajan, debido a que esta estructura trabaja con un alfabeto finito, y **N** siendo la cantidad de nodos que existen dentro de la estructura. Por el lado de búsqueda, ya mencionamos que es muy eficiente, importándole solo el largo de la cadena o palabra la cual se este buscando, dándole cero importancia a la demás información que existe dentro de la estructura, se implemento la estructura y se corrieron pruebas de búsqueda, dándonos resultados de **3.211602E-6** con cadenas de largo 50, mientras que los resultados obtenidos en búsquedas con cadenas mas pequeñas, también disminuyen en tiempo.

Keywords
Tries — Cadenas — Strings

[1] *Facultad de Ingeniería y Arquitectura, Universidad Arturo Prat, Iquique, Chile*
***Francisco Mamani**: franciscomamani123@gmail.com

Contents

Introducción

Las cadenas de caracteres son un tema bastante importante para una variedad de problemas de programación, el ”String Processing” tiene una variedad de aplicaciones en el mundo real, tales como, **Motores de Búsqueda**, **Análisis del Genoma**, **Análisis de Datos**, entre otros. Se revisara una estructura la cual es eficiente para este tipo de problemas, junto a un par de variaciones de esta estructura.

En este articulo se revisara la estructura de datos ”Trie”, se explicara en que consiste y como funciona, veremos sus operaciones mas importantes, como la de inserción, eliminación y búsqueda, explicaremos como funcionan, mostraremos sus algoritmos, revisaremos su complejidad en estas diferentes operaciones, también, revisaremos variaciones de esta estructura, se verán sus ventajas y desventajas que tengan sobre esta. Se hizo una implementación de esta estructura, donde se usaron las diferentes operaciones, se realizaron pruebas, que sirvieron para obtener resultados, que finalmente, serán mostrados y que servirán para sacar las respectivas conclusiones. A continuación, se presenta la estructura ”Trie”.

1. Trie

Un ”Trie” es una estructura de datos de tipo árbol que fue primero descrita por Rene de la Briandais en 1959 y Edward Fredkin en 1960, en un articulo llamado ”Trie Memory”, y fue este último quien introdujo el termino ”Trie”, el cual viene de la palabra en ingles ”re**trie**val”, que significa recuperación, ”Trie” es pronunciado como ”try”, otra palabra en ingles. Los ”Tries” son arboles m-

arios, m siendo el largo del alfabeto con el cual trabajan, debido a que la estructura ”Trie” asume explicitamente que las llaves son una secuencia de valores de algún alfabeto finito. Esta estructura es útil para guardar palabras de diferentes largos o cantidad de caracteres que trabajan bajo algún alfabeto. La estructura ha sido usada para guardar grandes diccionarios que sirven para trabajar con programas de corrección de ortografía, programas de auto-completado, entre otros.

La forma en que trabaja esta estructura es guardando palabras, cada caracter de esta palabra sera considerado un nodo, generando así un camino desde la raíz hasta el ultimo caracter de la palabra guardada, marcando al ultimo nodo como termino de esta palabra.

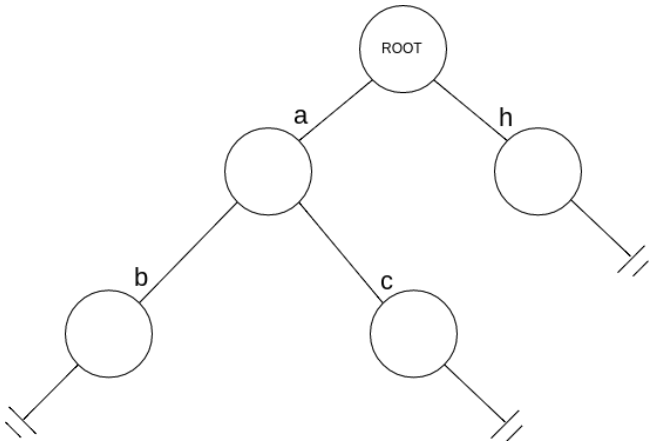

Figure 1

En esta representación básica de un ”Trie”(figura 1) podemos apreciar que contiene tres secuencias distintas de caracteres (en los casos que veremos para la explicación de un ”trie” usaremos cadenas pequeñas para ahorrar espacio), ”**ab**”, ”**ac**” y ”**h**”. ”b”, ”c” y ”h”, deben marcar de alguna forma que son el termino de la palabra, debido a que son el ultimo caracter de esta. También, podemos notar que las cadenas ”**ab**” y ”**ac**” comparten un mismo camino, esto es debido a que palabras con prefijos en común siguen un mismo camino para el llenado, búsqueda, eliminación e impresión de palabras, lo cual veremos mas adelante.

La representación que se mostró previamente es una representación bastante básica, pero que sirve para el entendimiento de esta estructura de datos. A continuación revisaremos las características de un ”Trie” y sus operaciones mas importantes.

1.1 Características de un Trie

Características Como acabamos de mencionar un ”Trie” es una estructura de tipo árbol, eficiente para el guardado de cadenas de caracteres, donde cada nodo es un caracter de esta cadena. Esta estructura trabaja con un alfabeto finito, esto quiere decir que cada nodo del árbol tendrá como mucho un numero de hijos igual al numero de llaves dentro de este alfabeto, por lo tanto, la estructura ”Trie” es conocida por usar mucha memoria, debido a que al insertar un nodo en el árbol, se le asigna inmediatamente la mayor cantidad de hijos que puede llegar a tener. Para el caso de este articulo, trabajamos con un alfabeto de 26 llaves, que van desde la letra ’a’ a la ’z’, debido a esto, si insertamos un nodo en el árbol, se le asignara un vector de 26 espacios vacíos, para llenar si llega el momento de insertar otro caracter que venga después de este. Otra característica de los ”Tries” es que al ser un árbol cuenta con una altura, la altura del árbol esta determinada por la palabra con mas caracteres dentro de la estructura. También, palabras que comparten prefijos siguen un mismo camino, ahorrando memoria al evitar crear nodos extras.

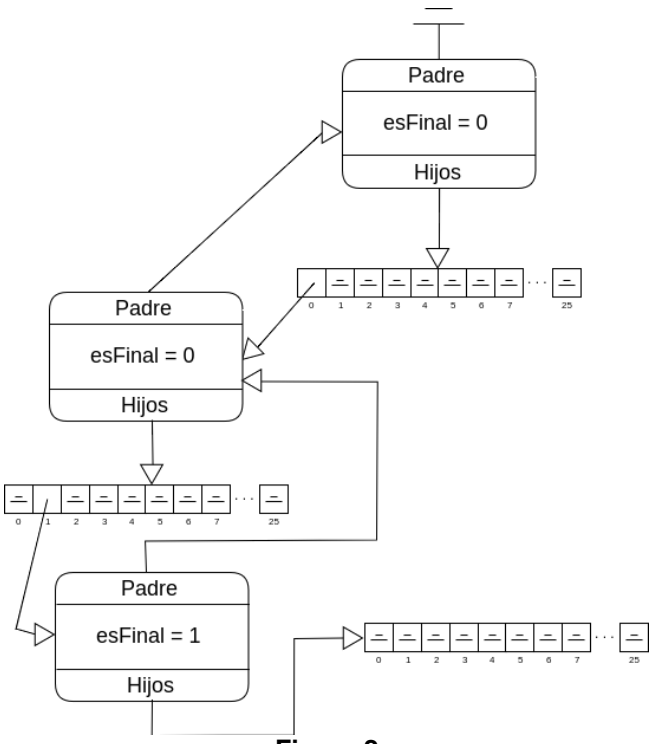

Figure 2

Como podemos ver en la figura 2, notamos que es una representación mas correcta de un ”Trie”, donde solo esta insertada la cadena ”**ab**”. Los nodos de esta estructura se componen de un puntero a un ”Padre”,

un dato de tipo entero, el cual marca si son el final de alguna palabra o cadena, y un puntero a un vector, el cual tiene el tamaño del alfabeto con el cual trabajan. En esta implementación, los nodos no cuentan con un tipo de dato que guarde que caracter son, ya que podemos obtener esa información debido a su posición en el vector. Para este ejemplo, se comienza con una raíz, la cual es el único nodo cuyo puntero a ”Padre” es nulo, y de esto podemos sacar provecho al momento implementar sus operaciones, que veremos mas adelante. Siguiendo, el nodo raíz tiene un vector, donde se busca la posición en donde debería ir el caracter ’a’, la manera en que encuentra esta posición es tomando el código ASCII del caracter de la palabra y lo resta con el código ASCII de otro caracter que colocamos como referencia para que los resultados de estas restas estén entre 0 y 25, que son los limites del vector.

A continuación, veremos las operaciones mas importantes de un ”Trie”.

1. Inserción
2. Búsqueda
3. Eliminación

1.2 Operaciones de un Trie

Inserción Para insertar una palabra dentro de la estructura, primero necesitamos un nodo raíz, luego, necesitamos la palabra a insertar y comenzar a recorrerla, caracter por caracter; El primer caracter de la palabra buscara su posición correspondiente dentro del vector que es apuntado por el nodo raíz, una vez que encuentre su posición se creara un nuevo nodo que ocupara dicho espacio en ese vector, tendrá un puntero ”padre” que apuntara al nodo raíz y un puntero que apuntara a un vector de 26 espacios vacíos, luego, se pasara al siguiente caracter de la palabra, en este caso, este nuevo caracter no buscara su posición en el vector que es apuntado por el nodo raíz, sino, por el nuevo vector que es apuntado por el nuevo nodo, generando un camino que finalmente es la palabra que queremos agregar. Se siguen los mismos pasos, se busca su posición, se coloca, su puntero a ”Padre” apuntara al nodo correspondiente y se verifica si es el ultimo caracter de la palabra o no, en el caso que si sea el ultimo caracter, se le suma 1 a su atributo *esFinal*.

Como podemos apreciar en la figura 3, que es pseudo código de nuestra función de inserción, podemos notar que hay una linea en donde verifica la posición de un caracter y revisa si ese espacio esta vacío, de no estarlo, crea un nodo como lo explicamos anteriormente, pero,

```
function Insertar(nodoRaiz, palabra)
begin
    auxNodo = nodoRaíz
    while (*palabra != '\0')     // recorre la palabra
        if (auxNodo->hijos_nodo[*palabra - 'a' == NULL])   // verifica de que no exista nodo
            auxNodo->hijos_nodo[*palabra - 'a'] == new Nodo()
            auxNodo->hijos_nodo[*palabra - 'a']->padre = auxNodo
        end if
        auxNodo = auxNodo->hijos_nodo[*palabra - 'a']
        ++palabra
    end while
    ++auxNodo->esFinal     // se marca final de la palabra
end
```

Figure 3

en el caso de que el espacio no este vació, osea, que exista un nodo, no creara otro, por que ya existe, eso quiere decir que ese caracter, en ese camino de la cadena ya esta colocado y no hace falta crear otro, simplemente se sigue al siguiente caracter. En el caso que exista ese nodo y sea el final de una palabra o cadena, solo se le suma uno a su atributo *esFinal*, incluso si es prefijo de alguna palabra o cadena mas larga, simplemente se denota que ahí termina una cadena.

Para esta estructura, los tiempos de inserción son bastante rápidos, están dentro del orden **O(k)**, **k** siendo el largo de la palabra a insertar. El tiempo de inserción y de las otras operaciones que revisaremos están dentro de un orden similar, donde son irrelevantes las demás cadenas o palabras que estén dentro de la estructura.

Búsqueda Como dijimos anteriormente, con respecto a la búsqueda esta estructura es muy eficiente, se encuentra dentro del orden **O(k)**, **k** siendo el largo de la palabra, esto quiere decir que si buscamos una cadena o palabra de largo 10, la estructura tendrá que visitar a lo mas 10 nodos, sin importar las demás cadenas o palabras que estén en la estructura, esto es por que al momento de la búsqueda, se toma el parámetro a buscar y se recorre caracter por caracter, como en el caso de la inserción, así, se busca por las posiciones dentro de los vectores que poseen cada nodo, por ejemplo, si buscamos como primero caracter una ’z’, revisamos directo el ultimo espacio del vector en cual nos encontramos, si este espacio se encuentra vacío, se termina la búsqueda inmediatamente, si por el contrario, el espacio tiene un nodo, quiere decir que si esta el caracter que buscamos y se sigue buscando el siguiente, hasta llegar al final de la palabra, o hasta que no encontremos lo que buscamos.

En esta figura, o código, notamos que también hacemos uso de el recorrido de la palabra que queremos buscar, por cada iteración de la palabra, el algoritmo

```
function Buscar(nodoRaiz, palabra)
begin
    while (*palabra != '\0')    // recorre la palabra
        if (nodoRaiz->hijos_nodo[*palabra - 'a'] != NULL)   // revisa si existe nodo
            nodoRaiz = nodoRaiz->hijos_nodo[*palabra - 'a']
            ++palabra
        else
            return NULL    // si no encontro nodo, no esta la palabra
        end if
    end while
    if (nodoRaiz->esFinal != 0)
        return nodoRaiz   // revisa el contador de la palabra
    else
        return NULL
    end if
end
```

Figure 4

revisa si la posición correspondiente del caracter esta vacía o esta llena, si esta llena, baja de nivel, y sigue con el siguiente caracter, en el caso de que este vacía, se termina la búsqueda, es por esto que para buscar una palabra solo se sigue un recorrido, que son los caracteres de la palabra, buscándolos por las ramas del árbol "Trie", y gracias a esta búsqueda no es necesario revisar ni un nodo extra. Cuando se termina de recorrer la palabra y se encontraron todos sus caracteres, solo queda revisar si ese ultimo nodo tiene su atributo *esFinal* mayor a cero, de ser así, si existe la palabra, en caso de que ese atributo tenga como valor, cero, significa que la cadena que se recorrió es prefijo de alguna otra palabra mas larga.

Eliminación En el caso de la eliminación, primero, necesitamos encontrar la palabra o cadena que se desea eliminar, si la palabra es encontrada se procede a eliminarla, en el caso de que la palabra no exista, la eliminación no tendría sentido.

Para la función de búsqueda, ademas de decirnos si la palabra esta o no dentro de la estructura, también, nos devuelve el ultimo nodo encontrado, y la función de eliminación necesita este ultimo nodo para comenzar a eliminar, por que la eliminación comienza desde abajo. Para poder eliminar un nodo, primero, necesita cumplir ciertos requisitos. El primer requisito es que su contador *esFinal* sea igual a 0, la función eliminar le restara 1 a dicho atributo de este nodo, si este atributo sigue siendo mayor que cero significa que la palabra estaba insertada mas de una vez, esto quiere decir que al restarle uno eliminamos una aparición de esta palabra, pero no podemos eliminar nodos, ya que aun se siguen necesitando. El siguiente requisito necesario es que sea un nodo hoja, esto quiere decir que no tenga ningún hijo que dependa de el, así que la función debe recorrer su vector y asegurarse de que no tenga hijos; Si este nodo llegase a tener hijos no se puede eliminar por que estaría afectando a otra palabra que comparte el mismo prefijo, y que necesita de ese nodo para existir dentro de la estructura. Como ultimo requisitos, necesitamos verificar si el "Padre" del nodo que estamos revisando sea distinto de Nulo, este requisito es muy importante, ya que el único nodo dentro del árbol que no debería tener padre es el nodo raíz, y el nodo raíz no lo queremos eliminar.

```
function Eliminar(nodoRaiz, palabra)
begin
    auxNodo = Buscar(nodoRaiz, palabra)  // la funcion de busqueda retorna el ultimo nodo de la palabra
    if (auxNodo != NULL)
        --auxNodo->esFinal
        auxPadre = NULL
        bool esHoja = true
        for (i = 0 ... i <ABECEDARIO)    // se recorre su vector para verificar si tiene hijos
            if (auxNodo->hijos_nodo[i] != NULL)
                esHoja = false
                break
            end if
        end for
        while (auxNodo->padre != NULL && esHoja && auxNodo->esFinal == 0)
            auxPadre = auxNodo->padre
            for (i = 0 ... i <ABECEDARIO)
                if (auxPadre->hijos_nodo[i] == auxNodo)
                    auxPadre->hijos_nodo[i] = NULL
                    delete auxNodo
                    auxNodo = auxPadre
                else if (auxPadre->hijos_nodo != NULL)
                    esHoja = false
                    break
                end if
            end for
        end while
    end if
end
```

Figure 5

Cuando se cumplen todos los requisitos, es seguro eliminar al nodo, y se sigue hasta llegar a la raíz, una vez que se llega a la raíz, la eliminación termina. Cabe destacar que la operación eliminación de esta estructura también se encuentra en el orden **O(k)**, **k** siendo el largo de la palabra o cadena.

1.3 Análisis

La estructura "Trie" es eficiente en términos de búsqueda, inserción y eliminación, estas tres operaciones caen dentro del mismo orden, un orden lineal, en donde solo dependen del largo de la palabra o cadena que se este usando como parámetro. Como se menciono anteriormente, a estas operaciones no les interesa los demás datos que pueden llegar a existir dentro de la estructura, estos algoritmos solo se concentran en la cadenas o palabra que se les entrega, alcanzando así, tiempos muy pequeños.

Table 1. Orden

Operación	Orden
Inserción	**O(k)**
Búsqueda	**O(k)**
Eliminación	**O(k)**

En la tabla que podemos ver, se denota la operación y su respectivo orden. Si bien es eficiente en estas operaciones, se ha mencionado reiteradas veces que no es eficiente en espacio, esto es debido a que los nodos mas alejados de la raíz tienden a tener muchos espacios vacíos en sus vectores, esto es una cantidad de memoria muy grande que no es usada. Dejemos a H como un alfabeto fijo, en este caso un alfabeto de 26. El espacio que ocupa un ”Trie” dependerá de los nodos que tenga, entonces, un ”Trie” con N nodos necesitara **O(HN)**.

Volviendo a lo mas importante, que es la búsqueda. El **Orden O(k)**, **k** siendo el largo de la cadena, se alcanza debido a que dentro de los ”Trie” se asume que al buscar una cadena, se obtienen los nodos en tiempo **O(1)**, esto es debido a que al buscar un nodo en un vector, este vector no se recorre, ya que se puede obtener la posición gracias al código del caracter, haciendo una sumatoria de los tiempos en que se obtiene cada nodo, la búsqueda finalmente es **O(k)**. Si llenamos el árbol con cadenas de un solo caracter, y realizamos una búsqueda, la respuesta sera inmediata, cualquier cadena que busquemos, debido a que la posición donde debería estar el nodo la podemos obtener inmediatamente.

Al revisar los casos de búsqueda, tenemos diferentes casos. Como mejor caso hay distintas variantes, si buscamos una cadena donde el primer caracter ya no esta en el árbol, inmediatamente termina la búsqueda, lo que nos da O(1), por otro lado, si buscamos una palabra, de largo 1, es decir, que contenga un caracter, el tiempo de respuesta también sera O(1). Ahora, revisando el peor caso, seria buscar la palabra con mas caracteres dentro del árbol, que en este caso estaría en el orden O(k), k siendo el largo de la cadena. Otro peor caso seria buscar la palabra mas larga dentro del árbol y que en el ultimo nodo no coincidiera, lo que también nos daría un peor caso de O(k).

Finalmente, quedando solo el caso promedio, en donde se puede realizar una búsqueda de una cadena de cualquier largo, que puede estar en el árbol, como también no puede estar en el árbol. Para este caso también cae en orden O(k), debido a que solo si buscara una cadena de largo uno pasaría a ser de orden constante, lo que llegaría a ser mejor caso.

2. Variaciones

2.0.1 Radix Tree

El ”Radix Tree”, también conocido como ”PATRICIA” por su acronimo (”Practical Algorithm To Retrieve Information Coded In Alphanumeric”), o ”Trie Comprimido”, es un simple variante de un ”Trie” donde todos los nodos interiores con un solo hijo son comprimidos, también, todos los nodos interiores del árbol tendrán al menos 2 hijos, por el contrario de los ”Trie”, donde sus nodos al menos 1 hijo. En esta estructura, en vez de que los bordes tengan una letra, se puede maximizar la eficiencia de memoria agrupando nodos que solo tengan un hijo, optimizando así la memoria usada por esta estructura.

La búsqueda en el Radix Tree es bastante similar a la búsqueda en un ”Trie”, la diferencia, es que en vez de recorrer nodos con un solo caracter con el cual comparar, recorre nodos que contienen cadenas de caracteres, pero la lógica es la misma, si se encuentra en un nodo con una cadena, recorre la cadena, si durante este recorrido encuentra que en la cadena hay un caracter no valido, distinto, etc, falla la búsqueda y se termina, por el contrario, si encuentra y valida todos los caracteres de la cadena, continua la búsqueda, o termina con la búsqueda, debido a haber encontrado la cadena. El orden de la búsqueda dentro de un Radix Tree es similar a la de ”Trie” común y corriente, **O(k)**, **k** siendo el largo de la cadena a buscar.

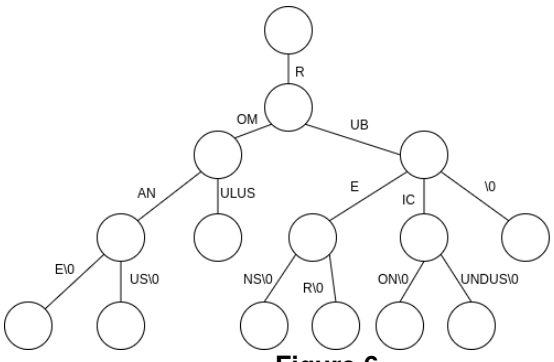

Figure 6

Como vemos en esta figura, se combinan los nodos, para crear cadenas de caracteres y ahorrar espacio, ademas de que en este caso, esta el nodo de termino de cadenas, pero solo en algunas cadenas, en otras cadenas esta combinado con la misma combinación de cadenas, también para ahorrar espacio. El orden en complejidad de espacio es **O(nM + N)**, donde **n** es el total de cadenas

guardadas dentro del árbol, **M** es la cantidad de llaves con las que trabaja el alfabeto y **N** es la suma del largo de todas las cadenas dentro de la estructura.

2.0.2 Ternary Tries

Esta estructura es otra estructura de tipo árbol, otra variación de un ”Trie”, para esta estructura cada nodo tendrá a los mas tres hijos, y cada uno de estos hijos representara algo distinto, los hijos izquierdos representan caracteres menores al padre, los hijos del medio cuentan como mismo valor y los hijos derechos representan caracteres menores al padre. Esta estructura guarda caracteres y valores, a diferencia de nuestra implementacion del ”Trie”, que solo guarda llaves.

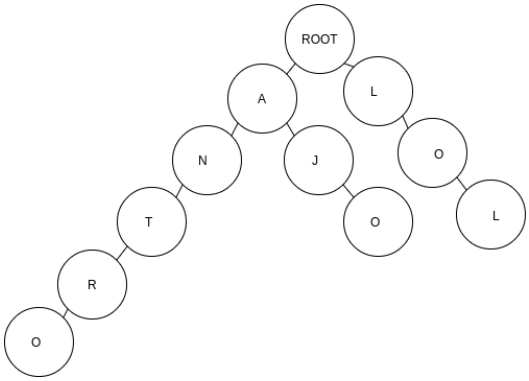

Figure 7

Como podemos ver en esta figura, la cual representa un ”Trie”, con las cadenas ”**antro**”, ”**ajo**” y ”**lol**”, podemos ver que su implementación es bastante simple, gracias a lo que hemos visto aquí. A continuación, se representara las mismas cadenas, pero en una estructura Ternary Trie.

En esta representación, el orden de inserción de las palabra juega un papel importante. La primera cadena que se inserto fue ”**ajo**”, que sigue un camino horizontal, la segunda cadena, ”**antro**”, viene después, en este caso se debe comparar el primer caracter, al ser el mismo caracter se avanza, ahora se toma el segundo caracter, el segundo caracter de ”antro” es ’n’, ’n’ es mayor que ’j’, es por esto que ’n’ es colocada como hijo derecho de ’j’, luego de eso, al no haber mas nodos que intervengan, simplemente se sigue con los demás caracteres. Finalmente se coloca la ultima cadena, ”lol”, para este caso, como el primer caracter ya no coincide, se ve inmediatamente si ’l’ es mayor o menor que ’a’, en este caso ’l’ es mayor, se coloca como hijo derecho de ’a’ y se procede a seguir la inserción de la cadena.

La búsqueda en esta estructura es bastante simple

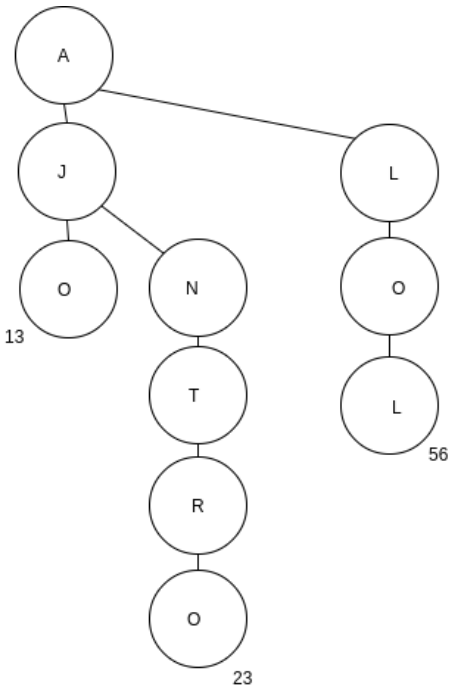

Figure 8

también, se sigue la cadena que se quiere buscar, cuando los caracteres no coinciden se verifica si son mayores o menores y se sigue la búsqueda, un ”hit” es la ocasión cuando se termina la cadena, y el valor que guardan las cadenas es distinto de NULL, un ”miss” seria el caso cuando efectivamente las cadenas si son NULL, queriendo decir que no tienen valor, significando que no son el termino de la palabra. El orden de búsqueda de esta estructura es **O(k + Lg(n))**, **k** siendo el largo de la cadena que estamos buscando, y **n** la cantidad de nodos que existen dentro de la estructura.

2.0.3 Conclusiones

Las variaciones del ”Trie” que pudimos revisar en esta sección, apuntan siempre a la optimización de memoria, debido a que es un tema muy importante, donde un ”Trie” común y corriente tiene muchas falencias y donde sufre desventajas en comparación con otras estructuras.

3. Resultados

Para poder demostrar los resultados que podemos obtener de un ”Trie” se realizo una implementación. Esta implementación era necesario para asegurar los ordenes en los que se encuentran sus operaciones básicas, es por esto que se construyo un ”Trie” en el lenguaje C++, donde lo sobrecargamos con cadenas de caracteres de distinto

largo y obtuvimos sus tiempos. A continuación, revisaremos los resultados que obtuvimos en las operaciones de inserción y búsqueda.

Inserción Primero revisaremos Inserción; Con inserción hubo muchas limitaciones, debido a la cantidad de memoria que requiere esta estructura, el programa era capaz de soportar como máximo doscientas cincuenta mil cadenas de cincuenta caracteres, ya que lo que ocupaba en memoria era absurdo, si sacamos cálculos, seria 26 * (cantidad de nodos); cantidad de nodos = 50 * 250000 = 12500000; Finalmente, se obtiene 26 * 12500000 = 325000000, lo que es demasiado. Debido a esto, estuvimos limitados a la cantidad de resultados que se pudieron obtener, no obstante, fueron suficientes para corroborar que el orden de inserción efectivamente es lineal.

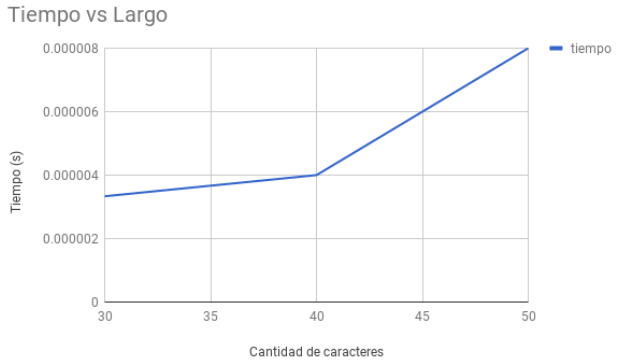

Figure 9

Como podemos apreciar en la figura 6, el crecimiento es debido a que se insertan cadenas con distintos largos, primero, se lleno la estructura con trescientas mil cadenas de largo treinta, y se midió el tiempo de inserción, lo que nos dio un resultado de 3.33333e-6, luego, se lleno con doscientas cincuenta mil cadenas de largo 40, etc. hasta 50.

Table 2. Resultados Inserción

Largo	Tiempo (s)
30	0.0000033333
40	0.000004
50	0.000008

Si bien, como muestra el gráfico de la figura 6, la linea es un poco pronunciada, es debido a la escaces de datos, que fue por las limitaciones de memoria, pero como se menciono anteriormente, fue lo suficiente para corroborar que la Inserción de esta estructura es de orden lineal.

Búsqueda Al seguir con los resultados de búsqueda, aquí, se tuvo mas libertad y una mayor cantidad de datos que se pudieron obtener. Para este caso, llenamos al árbol con trescientas mil cadenas de diferentes largos, primero solo cadenas de largo 5, luego de 10, luego 15, etc, también, hasta llegar a cadenas de largo 50, luego de llenar el árbol, hicimos búsquedas, por supuesto, realizar una sola búsqueda era absurdo, debido a que era muy rápido, incluso si buscábamos cadenas de largo 50. Una solución a esto, para poder obtener resultados con los que pudiéramos trabajar, fue realizar millones de búsquedas, obtener los tiempos y luego dividir los tiempos por las cantidades de búsquedas que hicimos. Aquí los resultados.

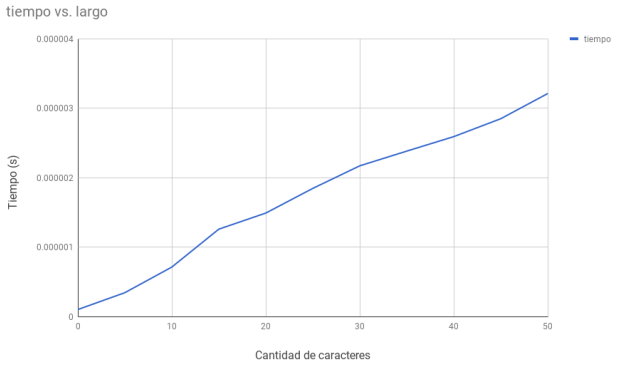

Figure 10

Aquí, en la figura 7, podemos notar resultados de manera creciente, que es lo que esperábamos, los tiempos suben cuando la cantidad de caracteres sube, lo que corrobora que el orden de búsqueda es **O(k)**, **k** siendo el largo de la cadena.

Table 3. Resultados Búsqueda

Largo	Tiempo (s)
5	0.0000003424960351
10	0.000000711539646
15	0.0000012575472
20	0.000001489519846
25	0.000001844992064
30	0.000002169956349
35	0.000002381829329
40	0.000002590623018
45	0.00000284855949
50	0.000003211602

Los resultados que podemos apreciar en esta tabla son mas exactos, estos resultados fueron obtenidos de un promedio de diferentes resultados, con cada largo de cadenas se realizo diferentes cantidades de búsquedas, estos resultados oscilaban dentro de un mismo rango muy pequeño, de estos resultados se obtuvieron los promedios. Los cambios mas notorios eran cuando se buscaban cadenas de diferente largo.

4. Conclusiones

Los resultados que obtuvimos gracias a la complementación de esta estructura fueron muy eficientes, recalcando, eficientes en tiempos de búsqueda, e inserción, por el contrario, en términos de espacio, hay muchas estructuras que son mejor opciones que esta, como las variaciones que vimos en este artículo. Cuando se trata de procesamiento de cadenas estas estructuras definitivamente son estructuras que no se deben obviar, el procesamiento de cadenas juega un rol importante en en problemas de programación. Esta estructuras, al ser eficientes en el guardado de cadenas y de acceso a estas, son usadas en diferentes programas, como programas de autocompletado, motores de búsqueda, y diferentes programas que necesitar guardar cadenas, que trabajen con alfabetos finitos, sea cual sea este. Si bien sus tiempos no son lo mas eficientes dentro de la lista, quedando por debajo de las estructuras con O(Lg N), estas estructuras están centradas para problemas en especifico, donde su implementación es la mas eficiente, incluso si requiere una disminución en velocidad.

References

1. de la Briandais, René (1959). File searching using variable length keys. Proc. Western J. Computer Conf. pp. 295–298. Cited by Brass.

2. Black, Paul E. (2009-11-16). ”trie”. Dictionary of Algorithms and Data Structures. National Institute of Standards and Technology. Archived from the original on 2010-05-19.

3. Franklin Mark Liang (1983). Word Hy-phen-a-tion By Com-put-er (Doctor of Philosophy thesis). Stanford University. Archived from the original (PDF) on 2010-05-19. Retrieved 2010-03-28.

4. Knuth, Donald (1997). ”6.3: Digital Searching”. The Art of Computer Programming Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley. p. 492.

5. Bentley, Jon; Sedgewick, Robert (1998-04-01). ”Ternary Search Trees”. Dr. Dobb’s Journal. Dr Dobb’s.

6. Edward Fredkin (1960). ”Trie Memory”. Communications of the ACM. 3 (9): 490–499.

7. ”Engineering Radix Sort for Strings”. Lecture Notes in Computer Science: 3–14.

8. Allison, Lloyd. ”Tries”. Retrieved 18 February 2014.

9. Sahni, Sartaj. ”Tries”. Data Structures, Algorithms, and Applications in Java. University of Florida. Retrieved 18 February 2014.

10. H. Cormen, Thomas. (2011). ”Introduction to algorithm”.