Árboles Digitales: Trie y Patricia

Año 2026

Introducción

Los problemas de búsqueda son parte esencial en el estudio y aplicación de las Ciencias de la Computación. La mayoría de las aplicaciones, por simples que sean, necesitan en algún momento realizar una búsqueda sobre un conjunto de datos dado, obteniendo la información asociada a ellos para su posterior procesamiento. Existe una variedad de problemas de búsqueda, algunos más sencillos requieren generalmente soluciones sencillas, pero problemas más complejos necesitan soluciones más sofisticadas y creativas.

Un algoritmo de búsqueda es aquél que está diseñado para localizar un elemento concreto dentro de un conjunto de datos. Una búsqueda logra solucionar el problema de existencia o no de un elemento determinado en un conjunto finito de elementos, es decir, responder si el elemento en cuestión pertenece o no a dicho conjunto. Tradicionalmente la operación de búsqueda se ha aplicado a *datos estructurados*, es decir a un conjunto datos que se han debido organizar de alguna determinada manera con el objetivo de facilitar su manipulación, dando así origen a las ***estructuras de datos***. Éstas ocupan un lugar importante en la resolución de problemas de búsqueda, ya que generalmente de la elección de la estructura más adecuada depende la eficiencia de una solución.

Además, al momento de buscar en alguna estructura, se pueden realizar *búsquedas exactas* o *búsquedas aproximadas* sobre la información almacenada. Las búsquedas exactas son las más conocidas, en ellas, dado un elemento de consulta usualmente representado por una secuencia o cadena de símbolos (*clave*) se recupera, del conjunto de datos almacenados, la información en la cual la cadena de caracteres es ***exactamente igual*** al elemento de consulta. Por otro lado, enfocadas a otro tipo de problemas, se encuentran las búsquedas aproximadas o por similitud, en éstas se buscan elementos de la información almacenada en la estructura que sean ***similares o próximos*** al elemento de consulta dado, es decir que pueden *no ser iguales* a la clave buscada, pueden tener cierta cantidad de diferencias pero igual sirven como respuesta.

En el ámbito de esta materia hemos utilizado búsquedas exactas para resolver problemas como: *pertenencia*: se verifica si el elemento de consulta aparece, o no, entre los elementos del conjunto almacenado. Evocaciones *asociativas*: dado un valor del dominio privilegiado (el dominio por el que se organiza la estructura (clave)), se realiza la localización de la nupla almacenada en la estructura cuyo valor en dicho dominio coincida con el valor buscado; si éste se encuentra almacenado en la estructura se devuelve la información asociada a él. Por último evocaciones *extremales*: siempre se devuelve un extremo (el primero o el último) del conjunto almacenado, de acuerdo a algún orden definido sobre los datos (orden lexicográfico, orden de llegada, orden matemático, etc.). En todas ellas, se usaron métodos basados en la comparación *completa* del valor buscado con el valor almacenado en las nuplas de la estructura.

Un tipo de búsqueda distinto es la que se realiza cuando se busca en un gran diccionario: se lee la primera letra de la palabra buscada y a partir de ella se puede rápidamente localizar aquellas páginas que contienen todas las palabras que comienzan con esa letra gracias al índice alfabético que aparece en el borde de las hojas del diccionario; la segunda letra de la palabra reduce la cantidad de páginas que se mirarán, lo mismo que la tercera y así se irá considerando cada caracter, reduciendo el conjunto de páginas cada vez hasta encontrar lo que se busca.

Si se usa este concepto como esquema de búsqueda, considerando al elemento a buscar *no como un todo* sino como una *secuencia* de dígitos o caracteres alfanuméricos, según se esté trabajando con números o palabras, se usará cada uno de los elementos de la secuencia para determinar una de las múltiples bifurcaciones a seguir en cada paso. Si los elementos de búsqueda son secuencias alfabéticas,[1] entonces se comienza leyendo la *primera* letra de la secuencia y haciendo una bifurcación, de 27 caminos posibles, de acuerdo a ésta; para continuar el proceso se leerá el *segundo* carácter de la secuencia y otra ramificación, también de 27 posibilidades, será elegida de acuerdo a esta segunda letra, y así sucesivamente hasta que los caracteres de la cadena buscada se terminen. Aquí los caracteres son usados cómo ***índices*** para la búsqueda.

La implementación de este esquema de búsqueda permite obtener las estructuras conocidas como *Árboles Digitales*, que son árboles *n*-arios utilizados para almacenar *cadenas de caracteres o strings* y resultan muy eficientes al momento de buscar una secuencia. Éstos permiten el almacenamiento de grandes cantidades de texto y permiten buscar eficientemente en él. La principal característica de los *Árboles Digitales* es que descomponen la cadena de búsqueda en caracteres (letras, dígitos[2] o símbolos) y usan éstos como índices para moverse en la estructura. Cada carácter ayuda a determinar una rama, de múltiples posibles, en cada paso.

A diferencia de cómo se procede en un árbol binario de búsqueda (ABB), en el cual se realiza la comparación de claves completas para buscar un elemento en él, para encontrar una cadena en un Árbol Digital comenzamos desde la raíz, de la misma manera que en un ABB, sólo que en cada nodo *no se compara* la cadena completa para saber si es mayor, menor o igual al elemento de consulta, sino que consideramos sólo un carácter de la misma. Por lo tanto, en el primer nivel usamos el primer carácter de nuestra secuencia para saber qué rama se toma para seguir en el árbol, en el segundo nivel utilizamos el segundo carácter de la cadena para elegir el siguiente camino y así sucesivamente hasta llegar a un nodo hoja, en el que puede estar almacenada, o no (según sea su implementación), la cadena buscada.

Estas estructuras de datos son adecuadas para realizar búsquedas rápidas de cadenas en un texto grande, se utilizan en aplicaciones tales como búsquedas de patrones, indexación y compresión de texto, recuperación de secuencias de texto, biología computacional, etc.

Árboles Tries

Un caso particular de árboles digitales son los *Tries*, éstos son árboles que fueron propuestos independientemente por Rene de la Briandais en *Proceedings Western Joint Computer Conference* 1959 y por Edward Fredkin en 1960, a éste último le deben su nombre; el mismo proviene de la extracción de letras de la palabra *re**trie**val* (recuperación) pero como su pronunciación en inglés, "tri", se confundía con *tree* (árbol, que también se pronuncia "tri") se tomó como convención pronunciarlo como "try"que en inglés se pronuncia "trai".

El trie es un árbol M-ario, que permite el almacenamiento de cadenas de símbolos de un alfabeto finito de cardinalidad M. Cada nodo interno del árbol es un vector de M posiciones. Cada posición en el vector se corresponde con un carácter del alfabeto y contiene un puntero que es un posible camino a recorrer durante una búsqueda. Así mismo, para cada carácter de una cadena almacenada en el trie existe un nodo asociado al mismo, es decir, si la longitud de una palabra es de 5 caracteres entonces en el tie habrá 5 nodos para almacenar ese string. Por lo tanto, en un trie, un camino completo desde la raíz hasta una hoja se corresponde con una de las cadenas perteneciente al conjunto almacenado en él. La

[1] Considere las letras del alfabeto castellano como alfabeto para armar las secuencias. Es decir que cada secuencia está formada por una sucesión de caracteres de este alfabeto.

[2] De allí les viene el nombre.

altura del árbol está directamente relacionada con la longitud de las palabras ingresadas, así la palabra con mayor cantidad de caracteres será la que determine la correspondiente altura del trie que la contiene.

Un trie de orden M puede ser definido formalmente de manera recursiva como: un trie vacío o una secuencia ordenada de exactamente M tries de orden M.

Sea $\mathcal{T}_\Sigma$ el conjunto de todos los tries que se pueden definir a partir del alfabeto Σ, donde $|\Sigma| = M$.

- $\Lambda \in \mathcal{T}_\Sigma$
- Si $T_1, T_2, T_3, \ldots, T_M \in \mathcal{T}_\Sigma$ entonces $\langle T_1, T_2, T_3, \ldots, T_M \rangle \in \mathcal{T}_\Sigma$ con cada T_i asociado al $i-$ésimo símbolo de Σ

Mientras algunas implementaciones de los tries no almacenan explícitamente las cadenas del conjunto representado por él, como el de la definición anterior, otras sí lo hacen poniéndolas en las hojas, por lo que en estas implementaciones las hojas del trie tienen un formato totalmente diferente al de los nodos internos ya que sólo guardan un string en ellas.

A continuación se muestran algunos ejemplos de alfabetos:

- Alfabeto binario: $\{0, 1\}$
- Alfabeto ASCII: $\{$los 256 caracteres $ASCII\}$
- Alfabeto Castellano: {a, b, c, d, e,....., x, y, z}

Considerando estos ejemplos, si se generara un trie que represente un conjunto de cadenas del primer alfabeto la estructura obtenida sería un árbol binario, cada nodo tendría 2 hijos; un trie para cadenas del segundo alfabeto sería un árbol 256-ario, cada nodo sería un vector de 256 posiciones, y finalmente un trie que almacene secuencias del último alfabeto sería un árbol 27-ario.

La Figura 1 muestra un trie que almacena cadenas de A^*, siendo A el alfabeto utilizado para la codificación de las mismas: $A = \{a, b, c, d\}$; por lo tanto este trie es un árbol cuaternario, o de orden 4, en el que cada nodo, salvo las hojas, es un vector de 4 posiciones. Este trie representa al conjunto de cadenas $\{aacc, d, abd, cbb, aba\}$ y las almacena explícitamente. Estando en la raíz se ve que en el conjunto representado no hay cadenas que comienzan con ***b*** ya que el puntero que se corresponde con ese carácter es ***nil***. Por otro lado, siguiendo el puntero asociado al carácter ***a*** se encuentran todas las cadenas que comienzan con esa letra. Lo mismo pasa con el puntero que corresponde a la letra ***c***. Finalmente se ve que hay una única palabra con la letra ***d*** y está almacenada en la hoja apuntada por el puntero correspondiente a ese carácter. Si en cambio se considera el segundo nodo (de izquierda a derecha) del segundo nivel (la raíz tiene nivel 0), se puede notar que este nodo representa a todas las secuencias cuyo prefijo es **ab**. Como se dijo, cada carácter de las cadenas representadas en el trie da lugar a un nodo, así por ejemplo se necesitan 4 nodos, desde la raíz, para representar la secuencia ***aacc***, dado que la misma tiene 4 caracteres.

Búsqueda

Al momento de una búsqueda la raíz del trie funciona como un índice alfabético en un diccionario: utilizando el *primer* carácter de la cadena buscada como *subíndice* en la raíz, que es el nodo corriente en este momento, se decide cuál será la rama que se seguirá; por ejemplo, si el primer carácter de la secuencia a buscar es la i-ésima letra del alfabeto, entonces se deberá seguir el i-ésimo puntero, para alcanzar el próximo nodo en el árbol. Esa rama me conduce al $i-$ésimo subárbol que contiene a todas las palabras almacenadas que comienzan con el i-ésimo carácter, y todas las cadenas que comiencen con ese carácter del alfabeto deberán seguir este puntero.

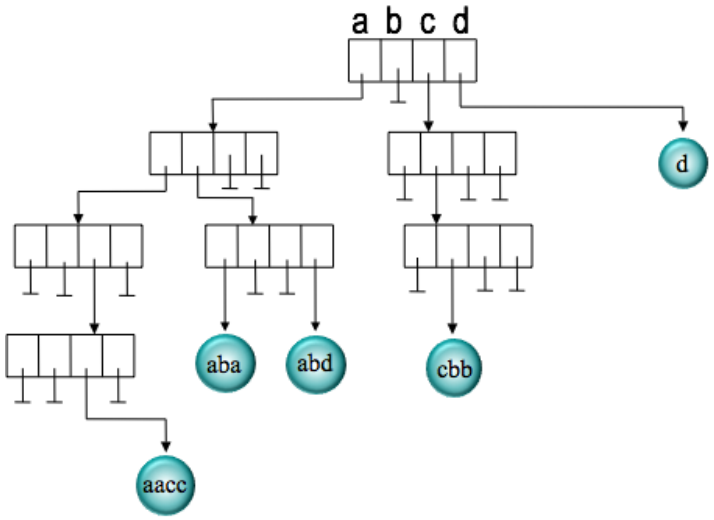


Figura 1: Ejemplo de un trie sobre el alfabeto $\{a, b, c, d\}$

En el próximo nivel la subindicación se realiza del mismo modo, pero de acuerdo al *segundo* carácter del elemento de consulta. Si la cadena a buscar tiene p caracteres el paso de búsqueda para esa cadena termina en un nodo de profundidad p. Cada nodo de nivel l en el árbol [3] representa el conjunto de todas las palabras que comienzan con una cierta secuencia de l caracteres; en ese punto el árbol permite seguir **una** de M ramas distintas dependiendo del carácter $l + 1$ de la cadena. En un nodo del trie los punteros que son nil corresponden a secuencias de caracteres que no aparecen en ninguna de las palabras almacenadas en el mismo.

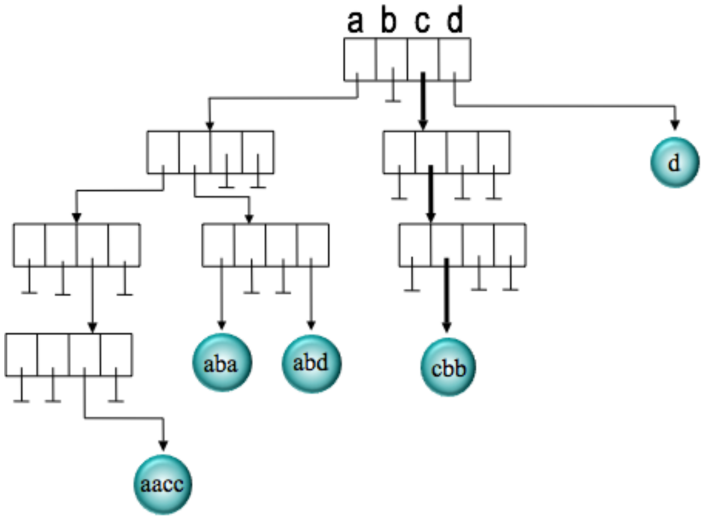


Figura 2: Búsqueda de '*cbb*' en el trie

Considerando el trie de la Figura 1, suponga que se necesita buscar la cadena ***cbb*** entonces el proceso a seguir sería el siguiente: comenzando en la raíz, se utiliza el primer carácter de la cadena buscada, una

[3] la raíz tiene nivel 0

c, como subíndice para decidir cuál será la rama a seguir. Así el puntero almacenado en *nodo[c]* nos permite avanzar en el árbol hasta un nodo en el próximo nivel. Ahora se toma el segundo carácter de la secuencia de consulta, una ***b***, y se utiliza como subíndice en el nodo corriente para encontrar el próximo puntero a seguir, *nodo[b]*. Este proceso se repite hasta terminar con todos los caracteres de la cadena buscada y llegar a una hoja en el trie. La Figura 2 muestra este proceso, en ella se señala el camino seguido en el trie resaltando los punteros utilizados con una línea más gruesa; como se ve en este ejemplo, el último puntero accede a una hoja en la que se encuentra la cadena buscada. Por lo tanto, la búsqueda fue exitosa.

Si ahora la cadena buscada fuera ***cbba***, el recorrido en el árbol sería igual al mostrado en la Figura 2, al menos para el prefijo ***cbb***, pero en este momento se ha alcanzado una hoja y en la cadena buscada todavía quedan caracteres, así que la búsqueda fracasa; esa cadena no se encuentra almacenada en el árbol por lo tanto no pertenece al conjunto representado por el trie. Otro caso se muestra en la Figura 3, aquí se ve el recorrido seguido en el trie durante la búsqueda de la secuencia ***cbc***: estando en la raíz seguimos el puntero que corresponde al carácter ***c***, luego atravesamos la rama correspondiente al carácter ***b***, y por último el puntero correspondiente al último carácter, ***c***, pero este puntero es nil, no se puede seguir, luego la cadena no se encuentra y la búsqueda fracasa.

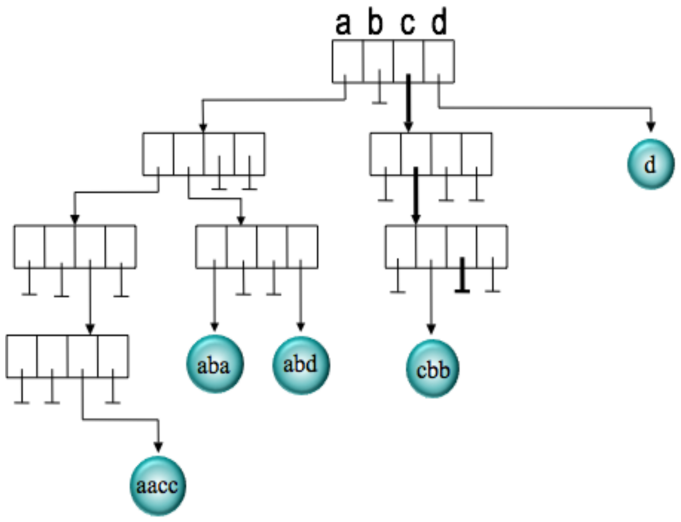


Figura 3: Búsqueda de '*cbc*' en el trie

Entonces, si durante una búsqueda se acaban los caracteres de la secuencia buscada y al seguir el puntero indicado por el último carácter se llega a una hoja, la *búsqueda tuvo éxito.* En cambio, si la cadena se termina antes de alcanzar una hoja, el puntero que se debe seguir es nil, o se llega a una hoja sin haber de terminado de recorrer completamente la cadena buscada, entonces la *búsqueda fracasó.* Notar que si la búsqueda de una secuencia fracasa, lo que se encuentra en el recorrido del trie hasta ese momento, es el substring más largo que coincide con la palabra buscada; esta propiedad resulta útil en algunas aplicaciones específicas como la búsqueda aproximada, búsqueda de la palabras que comienzan con..., etc.

Si analizamos los trie implementados según el ejemplo anterior, puede notarse que éstos no permiten almacenar cadenas que sean prefijos de otras ('*aa*' es prefijo de '*aacc*'), dado que este modelo sólo almacena las cadenas en las hojas del trie y un prefijo puede terminar en cualquier nivel del árbol. Esta característica, aunque útil para algunas aplicaciones, resultaba una limitación para otras más complejas; esto llevó a redefinir este modelo permitiendo el almacenamiento, y por lo tanto la búsqueda, de prefijos.

Una de las soluciones planteadas consiste en agregar un carácter especial que **no** pertenece al alfabeto y que indica '**fin de palabra**'; esta modificación transforma la estructura, ahora el trie será un árbol de aridad $M + 1$, siendo M la cardinalidad del alfabeto, y una posición más que se corresponde con el símbolo especial elegido para marcar el final de una secuencia. Toda secuencia del conjunto representado por el trie terminará con ese símbolo. Ahora en cada nodo del trie se tiene un conjunto de M punteros a nuevos nodos, si existen palabras más largas almacenadas, y un puntero a una palabra si alguna termina allí.

La Figura 4 muestra cómo se modifica el trie de la Figura 1 sobre el alfabeto $\{a, b, c, d\}$, cuando se agrega el símbolo especial λ (fin de cadena); también se ha agregado al conjunto de cadenas representadas el prefijo ***cb***λ, el cual se almacena en un nodo del segundo nivel siguiendo el puntero correspondiente, $nodo[\lambda]$. Esta nueva implementación del trie representa al conjunto de cadenas $\{aacc\lambda, d\lambda, abd\lambda, cbb\lambda,$
$cb\lambda, aba\lambda\}$ y como puede notarse, permite el almacenamiento de cadenas que son prefijos de otras. De aquí en más NO se escribirá el símbolo de fin de cadena (λ) en cada secuencia almacenada en las hojas del árbol por razones de legibilidad, ya que no se notan claramente las secuencias por su tamaño.

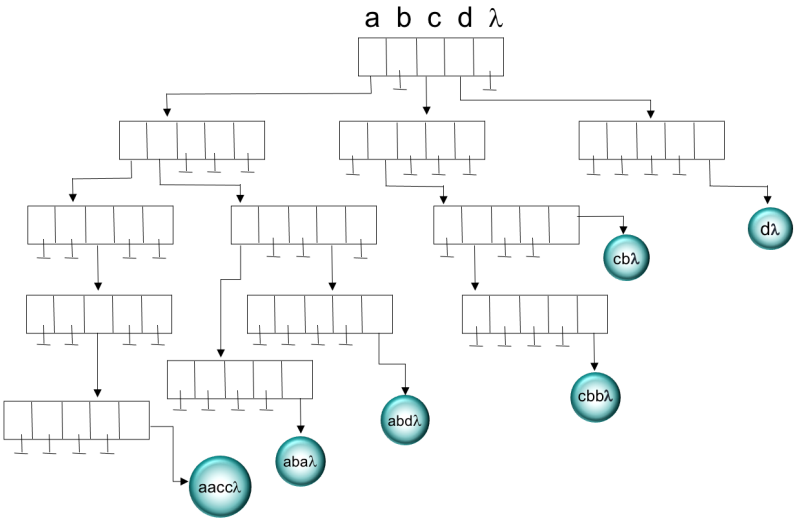


Figura 4: Ejemplo de un trie para el alfabeto $\{a, b, c, d\}$ con λ como fin de cadena

Las búsquedas, en esta nueva implementación del trie, se realizan de la misma forma que en la anterior, sólo que ahora el último carácter de las secuencias buscadas será λ, por lo que se deberá revisar el puntero asociado a su posición; si este puntero es nil entonces la cadena no pertenece al conjunto representado, en otro caso la cadena buscada debería estar almacenada allí.

Aquí se muestra, en pseudo-código, uno de los posibles algoritmos de búsqueda en un trie:

```
T1: p ⟵ raiz; Xb ⟵palabra a buscar          /* Inicializar */
T2: Si no terminó la palabra              /* Leer carácter */
          k ⟵ siguiente carácter
          Si k =fin de palabra
                  vaya a T4
          sino
                  vaya a T3
T3:Si ↑p[k] ≠ nil                        /* Avance */
                  p ⟵ p[k];
                  vaya a T2
          sino
                  Fracaso;
                  vaya a T5
T4: si ↑p[k] = Xb                         /* Verifica */
                  Éxito
          sino
                  Fracaso
T5: Termina el algoritmo
```

El poder utilizar un esquema de búsqueda basado en una subindicación repetida, es decir que los caracteres de una cadena sirvan como subíndices en cada nodo del árbol para decidir cómo seguir, logra que las búsquedas en un trie sean rápidas.

Altas

El trie es una estructura *dinámica*, es decir que permite altas y bajas. Cuando se quiere realizar un alta o insertar una nueva cadena en el trie, primero hay que buscarla en la estructura; para ello se debe atravesar el árbol siguiendo las ramas indicadas por los caracteres de la secuencia hasta llegar a un punto de fracaso, allí es donde deberá ser dada de alta la cadena. Si el camino seguido termina en un puntero a nil, entonces se deben insertar tantos nodos nuevos como caracteres falten de leer en la cadena (recordar que debe haber un nodo por cada carácter de la secuencia almacenada) para completar el camino correspondiente a la nueva cadena, e insertarla en el puntero correspondiente a λ del último nodo creado; si no quedan caracteres que leer en la cadena, sólo se insertará ésta en la posición $nodo[\lambda]$ del nodo corriente.

Por ejemplo, si se quiere insertar, en el trie de la Figura 4 la palabra ***dd***λ, la Figura 5 muestra el camino seguido durante su localización. Ésta fracasa en un puntero a nil correspondiente al segundo carácter de la cadena, una ***d***, en un nodo del primer nivel. Ese puntero a nil es el que se debería seguir si la secuencia estuviera almacenada en el trie, entonces ahí se deberá insertar la cadena. Pero si se analiza la secuencia a insertar, se ve que falta considerar un carácter (que aún no se ha leído) para terminar de recorrerla completamente, el carácter λ, entonces hay que insertar en la estructura un nodo nuevo que corresponde al fin de cadena λ y como ya se terminó la cadena, en la posición del arreglo $nodo[\lambda]$ se inserta la palabra ***dd***λ. Este proceso puede verse en la Figura 6, en la que se muestra, además del camino seguido durante la localización, cómo queda el trie luego del alta, con tres nodos que corresponden a los tres caracteres de la secuencia, y la palabra almacenada en el último de ellos.

Si ahora se quiere dar de alta la palabra ***ab***λ, que es prefijo de una cadena ya almacenada (***abd***λ), primero se debe localizar. Su localización terminará en un nodo del segundo nivel, en el puntero correspondiente al carácter de fin de palabra, que es nil, allí es donde debería estar almecenada si perteneciera al conjunto. Luego sólo se coloca allí la nueva palabra. La Figura 7 muestra tanto la localización, como la nueva forma del trie luego de esta inserción; notar que como esta secuencia es un prefijo de una ya

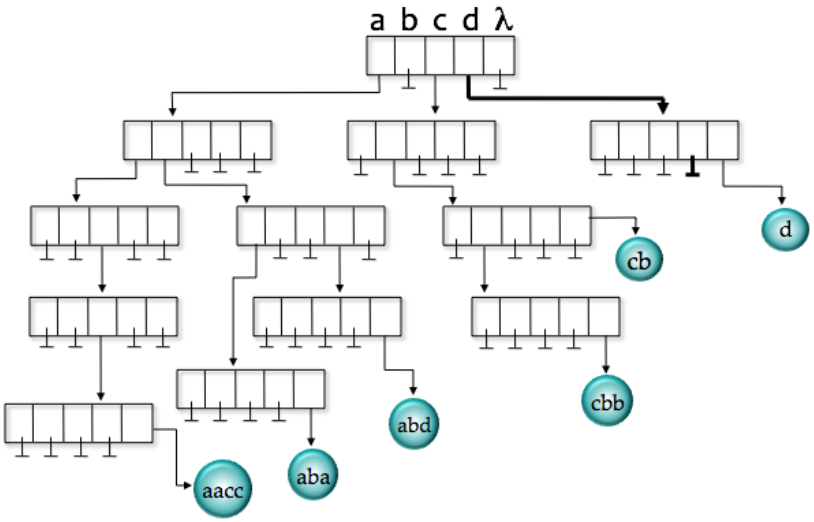


Figura 5: Buscando '$dd\lambda$' en el trie

existente en la estructura no se debe insertar ningún nodo nuevo. Además en el camino de la raíz a la palabra hay tres nodos, que es la longitud de la misma (***ab***λ).

Bajas

El mismo plan general utilizado para el alta también sirve para dar de baja o eliminar un elemento del trie. El proceso comienza localizando el elemento a dar de baja, es decir, recorriendo en el trie el camino indicado por la cadena que se eliminará, si ésta no se encuentra, entonces la baja fracasa; en otro caso, una vez alcanzado el nodo con la secuencia a dar de baja se procede a actualizar el puntero que lo señala asignándole nil y liberando su espacio. Éste es el procedimiento que se realiza si se quiere eliminar la palabra ***ab***λ: se busca la misma (en la Figura 7 se muestran los pasos seguidos en azul) y cuando se encuentra se le asigna nil al puntero $nodo[\lambda]$ que la apunta,demás de liberar el espacio asignado para la cadena.

Supongamos ahora que se desea eliminar la cadena ***aacc***λ, se comienza localizando la misma y cuando la secuencia es alcanzada, se actualiza el puntero correspondiente en su padre y se libera el espacio ocupado por la cadena. Pero al observar el nodo padre, puede advertirse que todos sus punteros tienen el mismo valor: nil, lo que significa que ese nodo sólo existía para alcanzar la cadena dada de baja, entonces ya no es necesario en la estructura y debe ser eliminado. Al dar de baja este nodo, el puntero en su padre por el cual se llegó a él debe ser actualizado con nil y en este momento se deberá chequear si el nodo padre tiene algún puntero distinto de nil. Si no lo tiene también se eliminará y el padre de éste será actualizado. Este proceso se repetirá hasta alcanzar algún nodo que tenga punteros distintos de nil, lo que significa que hay cadenas que compartían cierto prefijo con la dada de baja y en este nodo se distinguían.

Si se analiza la Figura 7, al dar de baja la palabra ***aacc***λ, los tres nodos por encima de ella sólo tienen un puntero distinto de nil, y es el que se utiliza para alcanzar la palabra dada de baja, por lo tanto estos nodos también serán eliminados. El nodo de más a la izquierda del primer nivel tiene a $nodo[b]$ distinto de nil por lo que aquí termina este proceso de actualización, luego de haber eliminado la cadena propiamente dicha y tres nodos que, como se dijo, sólo eran necesarios para alcanzar la secuencia dada de baja.

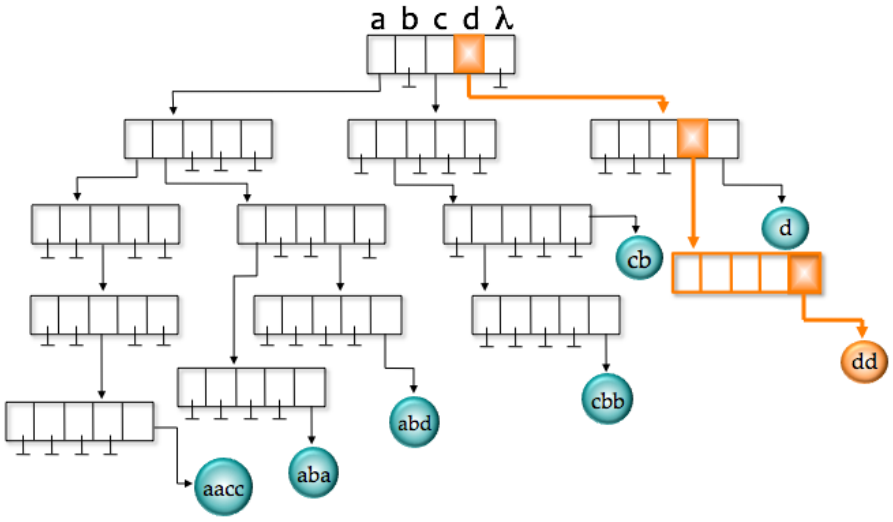


Figura 6: Alta de la cadena '*dd*' en el trie

Entonces para realizar una baja en el trie no sólo hace falta recordar el puntero al padre del nodo a eliminar, como en el *ABB*, sino que hay que mantener una pila (como estructura auxiliar) en la que, a medida que se avanza en el árbol, se vayan recordando los nodos por los que se fue pasando, desde la raíz hasta la hoja, para poder luego actualizar todos los que sean necesarios. La Figura 8 muestra la estructura luego de eliminar las secuencias ***aacc***λ y ***ab***λ.

Es importante observar que esta estructura no es sensible al *orden de la secuencia de entrada* (orden en que se ingresan las cadenas) como pasa con los *ABB*; dado un conjunto determinado de palabras distintas el trie que se obtendrá es único. En cambio sí se pueden producir agrupamientos en algunas ramas provocados por secuencias de caracteres que son prefijos de otras. No obstante, si se recorre el trie de izquierda a derecha recuperando cada secuencia almacenanda, éstas serán obtenidas en forma *ordenada*.

Análisis de costos del Trie

Al trabajar con estructuras siempre hay que tener en cuenta que una estructura es más eficiente cuanto más provecho se puede sacar del compromiso *tiempo-espacio* presente en cada problema planteado. Si al momento de decidir la mejor estructura a utilizar para resolver una situación particular sólo me interesa una de estas variables será mas sencillo tomar una decisión.

El poder utilizar un esquema de búsqueda en el cual los caracteres de una cadena sirvan como subíndices en cada nodo del árbol para decidir el camio a seguir, logra que las búsquedas en un trie sean muy rápidas. El número de caracteres examinados, en promedio, durante una búsqueda aleatoria con palabras igualmente probables de ser buscadas, es de aproximadamente $O(\log_M N)$; donde M es el tamaño del alfabeto y N la cantidad de palabras almacenadas en el trie. El peor caso es mucho mejor que el de los *ABB*s si: a) El número de cadenas es grande y b) Las cadenas no son largas.

Como se dijo, el tiempo de acceso en un trie es muy bueno y ésta es una de sus principales ventajas. Sea l la longitud (medida en cantidad de caracteres) de la cadena más larga, una secuencia puede ser encontrada en tiempo $O(l)$, independientemente de N, M y el número de nodos del trie. La desventaja más grande de esta estructura es su tamaño, son necesarios aproximadamente $\frac{N}{\ln M}$ nodos, para

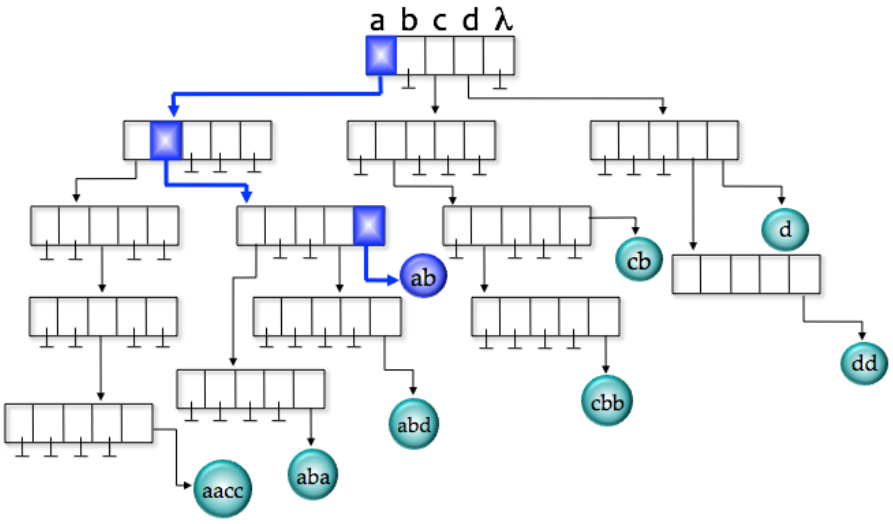


Figura 7: Alta de la cadena '$ab\lambda$' en el trie

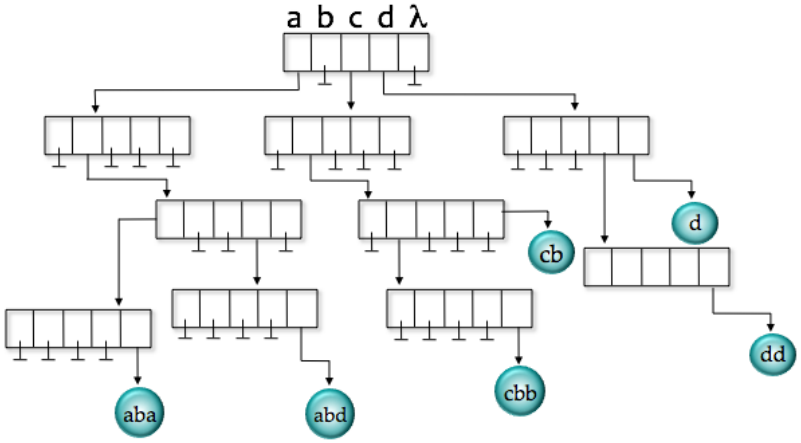


Figura 8: Trie luego de la baja de las cadenas '$ab\lambda$' y '$aacc\lambda$'

distinguir entre las N secuencias aleatorias almacenadas, luego el espacio total de almacenamiento es proporcional a $\frac{M*N}{\ln M}$ debido a que cada nodo es un vector de M posiciones.

El inconveniente del espacio ocupado por el trie no sería tal si se tiene acceso a una memoria ilimitada y sólo interesa el tiempo de respuesta ante una consulta ya que éste es muy bueno. Pero es bien sabido que las situaciones ideales no existen y si bien, como se dijo, el trie permite búsquedas muy eficientes, su principal desventaja es el espacio que ocupa.

En un trie que representa un conjunto de secuencias sobre un alfabeto de tamaño M, cada nodo en la estructura es un vector de $M + 1$ posiciones, muchas de las cuales pueden estar ocupadas con punteros a nil; además hay un nodo por cada carácter de las cadenas almacenadas. Entonces, si la cantidad de nodos del trie es s el espacio ocupado por el mismo es igual a $[\ (M + 1) * s *$ *tamaño de puntero* $]$ bits, más el espacio ocupado por las palabras, si es una implementación que las mantiene en sus hojas. Si bien en los primeros niveles del trie, los más cercanos a la raíz, los nodos tienen la mayoría de sus punteros con valores distintos de nil, a medida que nos acercamos a las hojas del árbol esta proporción

se invierte y la mayoría de los punteros de un nodo son nil.
Si se considera un trie que representa un conjunto de palabras del idioma castellano, el tamaño del alfabeto considerado sería de 27 letras, por lo que cada nodo del trie sería un vector de 28 punteros. Suponga que hasta el momento la única palabra almacenada en ese trie fuera ***imposibilidad***, se necesitarían 14 nodos, uno por cada carácter incluído el de fin de palabra, por lo que el espacio utilizado sería: $(28 * 14 *$ *tamaño de punteros*$) = (392 *$ *tamaño de punteros*) bits, de los cuales sólo 14 son distintos de nil, más el espacio que ocupa la palabra lo que significa un gran desperdicio del mismo. Es cierto que este es un ejemplo extremo y a medida que se inserten cadenas en el trie hay muchos nodos que serán compartidos por otras palabras y la cantidad de punteros a nil va a disminuir, pero aunque el trie del ejemplo almacene un diccionario, los nodos de los primeros niveles serán nodos con pocos punteros a nil y a medida que se baje hacia las hojas esto irá cambiando porque hay combinaciones de letras que nunca se dan en una palabra.

Utilizar sólo los Punteros Útiles

Teniendo en cuenta lo descripto anteriormente es interesante pensar en mejorar el espacio ocupado por el trie almacenando solamente aquellos punteros que tengan valores distintos a nil, siempre tratando de mantener los tiempos de búsqueda lo más cercanos posibles a los actuales. Así, en el ejemplo anterior si la única palabra guardada es ***imposibilidad***, sólo se necesitarían tener nodos con un solo puntero. La raíz debería tener una sola rama, el nodo del primer nivel también, y así siguiendo hasta llegar a la hoja que almacena la cadena. Si ahora se insertara otra palabra, por ejemplo ***imparcial***, el cuarto nodo dejará de tener un solo puntero para tener dos, ya que ambas secuencias comparten el prefijo ***imp*** pero el siguiente carácter es diferente en cada una, por lo que deben tomar diferentes ramas: una la correspondiente a la ***a*** y otra a la ***o***. Suponga que se da de alta a dos palabras más: ***implícito*** e ***importante***, en este punto el cuarto nodo tendrá tres punteros correspondientes a la ***a***, la ***o*** y la ***l***, el nodo que se alcanza al seguir el camino de la ***o*** tendrá dos punteros, uno por la ***s*** y otro por la ***r*** y todos los demás nodos sólo uno. Entonces ¿Cómo se puede saber cuantas ramas útiles (distintas de nil) tiene cada nodo si este valor puede variar según sea el caso? Además ya no todos los nodos tendrán la misma cantidad de punteros, sino que dependerá de la cantidad de cadenas distintas que pasen a través de él. Luego, si eliminamos todos los punteros nil del trie estamos en presencia de un árbol de *aridad variable*, entonces ahora el problema a resolver será cómo implementar esto.

Evidentemente se quiere disponer de un método de representación que emplee elementos de tamaño fijo, con un número limitado de campos de punteros, que no obstante puedan representar árboles de grado arbitrario. Una solución posible es utilizar un algoritmo conocido como *Transformada de Knuth* que representa un árbol n*-ario* como uno *binario*. La idea de éste algoritmo es ver los hijos de cada nodo como un conjunto de celdas que pueden encadenarse en una lista y ese nodo tendrá un puntero a ese conjunto. Ahora, la relación entre un nodo y sus hijos se puede representar por un simple puntero que accede al primer elemento en el conjunto de hijos del nodo considerado (Figura 9).

Como puede verse, el árbol obtenido luego de aplicar la transformada de Knuth es binario, cada nodo tiene cero, uno o dos hijos, y cada nodo está encadenado por la *izquierda*, con su lista de hijos, y por la *derecha* con elementos que tienen su mismo padre (hermanos). Como la lista de hermanos es considerada como un conjunto, el orden en el que los elementos de la misma son encadenados entre sí es indistinto, sin embargo al momento de una búsqueda esta lista será recorrida secuencialmente, por lo que conviene mantenerla ordenada para mejorar los tiempos de las búsquedas que fracasan, (ver Figura 9).

La Figura 10 muestra cómo queda, el trie que se está analizando como ejemplo, luego de aplicarle la transformada de Knuth; notar que en esta representación del trie se agregó un carácter que indica fin de palabra para poder almacenar prefijos, y en la estructura resultante los símbolos de fin de palabra

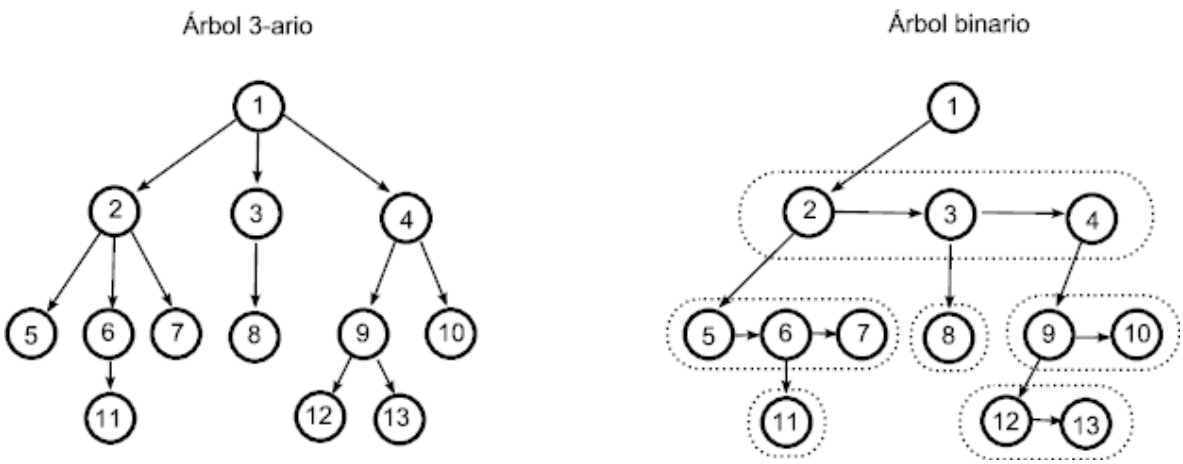


Figura 9: Transformada de Knuth

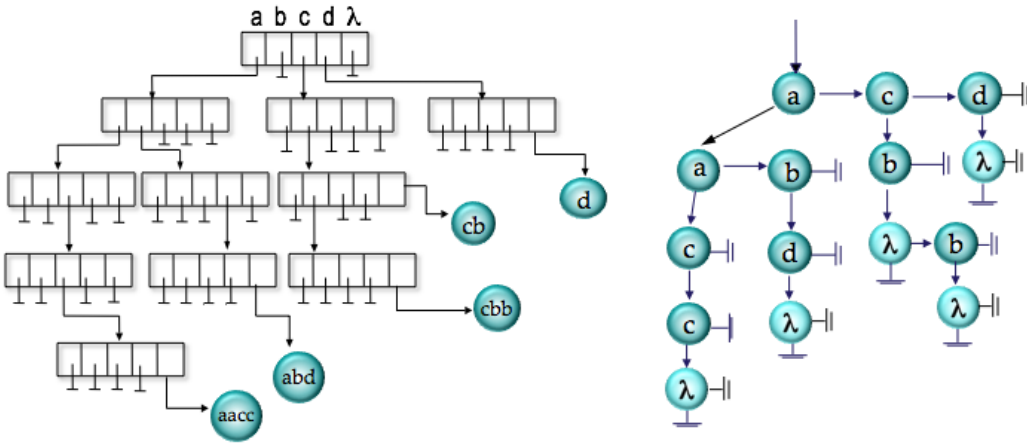


Figura 10: Transformada de Knuth aplicada al trie del ejemplo de la izquierda

aparecen como primer elemento de la lista correspondiente, esto se debe a que, por convención, son considerados como menores a cualquier carácter del alfabeto.

En esta implementación sólo están presentes los punteros que son distintos de nil en el trie, lo que disminuye notoriamente el espacio ocupado por el trie. Sin embargo, al momento de una búsqueda los tiempos se incrementan dado que los caracteres de la cadena buscada ya no pueden utilizarse como subíndices; los nodos ya no son vectores, y además si la secuencia buscada no se encuentra en el conjunto representado hay punteros que ni siquiera existen en el trie. Esto implica que el proceso de búsqueda será diferente al del trie original, al igual que la información que se almacene ahora en cada nodo. Dado que los punteros asociados a los caracteres que no aparecen en ninguna palabra almacenada no están presentes, esta representación debe guardar en cada nodo el carácter asociado al mismo para saber si ése es el camino que se debe seguir o no. Entonces ahora solamente están presentes los punteros que permiten avanzar en alguna palabra y los nodos están rotulados con el carácter correspondiente.

Para buscar en este árbol binario se toma el primer carácter de la secuencia buscada y con él se accede a la lista de elementos que contiene los primeros caracteres de las cadenas representadas, o sea, siguiendo la rama derecha del árbol buscamos por ella hasta que alguno de los caracteres almacenados

en ella coincida con el buscado o hasta llegar a un nodo cuyo puntero derecho sea nil, aquí la búsqueda fracasa porque no hay ninguna secuencia que comience con el carácter buscado. Si hubo coincidencia, se toma la rama izquierda del nodo con el que se emparejó y se lee el segundo carácter de la palabra de consulta, con este nuevo carácter en vista se repiten los pasos descriptos, pero esta vez sobre la lista accedida a través del hijo izquierdo del nodo con el que se coincidió antes.

Este proceso termina cuando se alcanza un punto de fracaso o cuando se lee el símbolo de fin de cadena en la secuencia buscada y el próximo puntero conduce a un nodo que lo contiene. La búsqueda de un carácter por la rama derecha es una búsqueda en una lista vinculada por lo que los costos de fracaso, medidos en cantidad de comparaciones, disminuyen si, como se dijo, la lista está ordenada, poniendo en primer lugar el símbolo de fin de palabra, si éste aparece en ese nivel del árbol.

Puede verse claramente que el ahorro de espacio conlleva un incremento de los tiempos de búsqueda provocados por la búsqueda del carácter correspondiente dentro de la lista de hermanos (acción que antes se realizaba solamente con una subindicación). En esta representación la búsqueda realizada es más por comparación, moviéndonos por igual o distinto (en lugar de por mayor o menor como en un *ABB*), y realiza al menos $\log_2 N$ comparaciones en promedio para distinguir una entre N secuencias distintas. Mientras que el trie tradicional, dado que en cada nodo puede hacer la elección de una de M ramas en un solo momento, realiza $\log_M N$ comparaciones para encontrar una cadena entre N secuencias (con N suficientemente grande) si todas las cadenas tienen la misma probabilidad de ser buscadas.

Estructuras Mixtas

El desperdicio de espacio del que parece padecer el trie no siempre es tan marcado, la mayoría de las veces depende de la aplicación para la que se utilice. Si el trie se necesita para representar un diccionario, por ejemplo, éste parece ser totalmente eficaz en los primeros niveles del árbol ya que aquí hay muchas palabras con diferentes letras y combinaciones de las mismas que logra que la mayoría de los nodos tengan punteros distintos de nil. Luego parece ir perdiendo efectividad a medida que avanza en la profundidad del mismo dado que la mayoría de los vectores tienen un gran porcentaje de sus punteros inútiles, posiblemente porque determinadas secuencias de letras no se encuentran en el idioma del conjunto representado. A además la cantidad de palabras va disminuyendo a medida que aumenta su longitud, por lo que el consumo de espacio se hace más notorio.

Una posible alternativa a un trie puro es una *estructura mixta* capaz de obtener un mejor desempeño. La idea es mezclar dos estrategias usando un trie para los primeros niveles, cambiando luego a alguna otra técnica. Por ejemplo E. H. Sussenguth, en 1963, sugirió usar un esquema carácter a carácter hasta alcanzar una parte del trie donde solamente, digamos, seis o menos cadenas del conjunto almacenado son posibles; entonces esas cadenas restantes podrían almacenarse secuencialmente una lista corta. Es claro que esta estrategia mixta disminuye la cantidad de nodos del trie en más o menos un factor de seis, sin cambiar substancialmente el tiempo de búsqueda.

Explotar los Conocimientos Adquiridos

Se han sugerido un gran número de variantes tanto del trie puro como de las estrategias de búsquedas por caracteres, y muchas de ellas derivan del conocimiento profundo de la aplicación en la que se utilizará la estructura en cuestión. Por ejemplo, si se quiere implementar un diccionario, es decir el conjunto a representar son todas las palabras del idioma castellano se necesitará, por supuesto, una gran cantidad de memoria para mantenerlo, entonces el objetivo será encontrar una forma compacta de representarlo que, sin embargo, permita mantener un tiempo de búsqueda razonablemente bajo. Un típico diccionario de escuela tiene alrededor de 100.000 palabras, pero al tratar de aplicar la aproximación de trie en

memoria es posible notar que es factible hacer importantes simplificaciones.

Se puede comenzar descartando los nombres propios y las abreviaturas; también utilizar el hecho de que si la primer letra de una palabra es ***b***, la segunda letra nunca podrá ser una j, o una c, y tampoco ninguno de los caracteres del conjunto $\{b, m, d, f, g, h, k, n, p, q, s, t, v, w, x, y, z\}$, una b sólo podrá ser seguida por una vocal o por consonantes como l y r; lo mismo pasa con palabras que comienzan con otras consonantes como d, k, f, g, que tienen estas 19 letras excluidas como segundo carácter de las mismas. Por supuesto existen excepciones como *psicología*, *pseudo*, *pterodáctilo* o *nylon*, entre otras. Si consideramos la letra b dentro de una palabra, ésta puede ser seguida por algunas letras más, por ejemplo *objeto*, *abstracto*, *obtuso*, y en aquellas palabras formadas por prefijos, por ejemplo *subconjunto*, *subterráneo*; lo mismo pasa con palabras que contienen otras consonantes juntas como *abducir*, *adjuntar*, *intermitente*, *inconsciente*, etc.

Una forma de aprovechar este hecho es codificar las letras que no pueden ser seguidas por cualquier otra (d, k, f, g, b, etc.), de forma tal de convertirlas en una representación especial con un código mayor al código numérico de las demás letras. De esta forma muchos de los nodos del trie se pueden acortar a vectores de 9 ramas, con alguna celda de *escape* que permita guardar aquellas palabras que son excepciones. La aplicación de esta estrategia en todos los niveles del trie disminuirá el espacio de memoria utilizado por éste.

Otra opción a tener en cuenta, que también surge del conocimiento del problema a resolver, es el tratamiento de los prefijos; por ejemplo si la palabra buscada comienza con alguna de las secuencias *in, des, a, pre, sub, anti*, etc. se podría asumir que esta subcadena es un prefijo y suprimirlo buscando sólo la secuencia restante. De esta forma al decidir representar sólo palabras sin prefijos la cantidad de secuencias almacenadas puede reducirse, removiendo aquellas cadenas que comiencen con prefijos, como *inmadura*, *subdesarrollada*, *incompleta*, *despoblado*, *atípico*, *desatado*, etc. Esto, además, evita almacenar cadenas largas que incremente la cantidad de nodos con pocos punteros diferentes de *nil*. Sin embargo hay que tener mucho cuidado con aquellas palabras que parezcan poseer prefijos pero que no los tienen, como es el caso de *deseado*, *inocuo*, *suburbio*, dado que su significado no puede deducirse si las subcadenas *des*, *in* o *sub*, son eliminadas, entonces ellas deberán almacenarse completas.

Con esta nueva estrategia de representación del conjunto de secuencias, al buscar una palabra se comenzará con la localización de la misma en forma completa y si ésta no se encuentra se suprimirá el prefijo y se buscará el resto de la palabra. Es claro que si se necesita el significado de una palabra no alcanza con sólo encontrarla sino que luego se debe ser capaz de rearmar el significado de la palabra buscada componiendo el significado de la secuencia localizada, y el significado del prefijo. Por ejemplo, para saber el significado de *desocupar*, primero se busca la palabra completa, como ésta no está se descompone: $des + ocupar$ y sacándole el prefijo se busca la palabra $ocupar$, en este caso, al encontrarla entre sus significados se encuentra *Emplearse en un trabajo, ejercicio o tarea*; entonces si rearmamos su significado: *sin* + *emplearse en un trabajo, ejercicio o tarea* = *sin actividades* o *sin tareas* o *sin trabajo*.

Si en cambio se necesita implementar un diccionario de palabras en inglés, una posibilidad importante a considerar para disminuir el tamaño del trie es a través de la supresión de los *sufijos*. En este idioma es posible obtener diferentes palabras a partir de la raíz de las mismas mediante el agregado de sufijos. Los diferentes tiempos verbales se arman a partir de esta forma agregando *ed, ing, es*, etc., lo mismo se aplica para obtener los plurales, transformar algunos verbos en sustantivos, adjetivos, adverbios o en una familia de palabras relacionadas, utilizando por ejemplo sufijos como: *ions, ionally, ability, able*, etc. Por ejemplo si tomamos el sustantivo *Success* (éxito) lo podemos transformar en un adjetivo si agregamos **ful**, *Successful* (exitoso), o en un adverbio agregando **fully**, *Successfully* (exitosamente). Para el idioma inglés esto probablemente sea más productivo que trabajar con prefijos y, aunque es cierto que según sea la parte de la palabra que se guarda (*comput-* o *computat-*), al agregar algunos sufijos se podría terminar armando una secuencia que no sea una palabra válida del idioma, esto no provocaría

daño alguno ya que esta secuencia nunca aparecería en la entrada para ser buscada.

Por supuesto, debe cuidarse que el significado de cada palabra sea determinado apropiadamente a partir de su raíz y del sufijo (o prefijo) que forme parte de ella. Si esto no fuera posible las excepciones deberán ser guardadas de forma tal que sea encontrada antes de tratar de descomponer la palabra en cuestión.

Árboles Patricia

Como se analizó anteriormente, la representación original del trie utiliza mucho espacio en punteros que no conducen a ninguna secuencia (nil), por lo que se plantearon posibles soluciones a este problema. Del mismo modo, el trie emplea buena cantidad de espacio cumpliendo con la exigencia de tener un nodo por cada carácter presente en las cadenas que almacena; el espacio malgastado no es tan notorio en nodos compartidos por diferentes secuencias del conjunto, en cambio toma relevancia en los recorridos hacia cadenas muy largas o aquellas que no comparten prefijo con otra. Los nodos que se atraviesan en este caso están presentes en el trie sólo para llegar a las mismas, si éstas no estuvieran almacenadas esos nodos no existirían; esto implica que cada uno de esos nodos tendrá un único puntero distinto de nil que es el que permite avanzar hasta estas secuencias.

Observando el trie de la Figura 10 puede verse un ejemplo de lo detallado anteriormente representado en la secuencia *aacc*λ, en el recorrido desde la raíz hasta esta cadena sólo dos nodos son compartidos por otra secuencia, el resto tiene un solo puntero útil cada uno, en este caso se desaprovecha el espacio de cuatro punteros por cada nodo del camino. Los caminos de este tipo se los conoce como *caminos unarios*, cada nodo tiene sólo un hijo. Evidentemente cuando los ejemplos que se analizan provienen de la realidad el espacio en desuso se vuelve un problema; retomando el ejemplo del diccionario castellano que, hasta el momento, sólo tiene almacenada la palabra *imposibilidad*, aunque se trata de un ejemplo extremo, puede verse que se necesitarían 14 nodos de 28 punteros cada uno, es decir, lugar para 392 punteros de los cuales sólo 13 son distintos de nil.

La búsqueda de posibles soluciones a este problema, teniendo como idea básica evitar los caminos unarios pero manteniendo los costos en las búsquedas, dio como resultado la definición de una nueva estructura, el árbol *Patricia*. Esta estructura que surgió durante el estudio de los problemas del trie, trata de mejorar el desempeño del mismo utilizando la reducción de los caminos unarios; su nombre *Patricia* es la sigla de **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric, en castellano: Algoritmo Práctico para Recuperar Información Codificada en Alfanumérico, y fue creado por Morrison en 1968 dentro del contexto de *indexación de strings*. Los árboles Patricia son árboles digitales cuya principal diferencia con los tries es la eliminación de los nodos unarios (con un solo hijo), los nodos que permanecen en el árbol son aquéllos que tienen una cantidad de hijos mayor o igual a dos (más de dos punteros distintos de nil); estos nodos son, igual que en los tries, vectores con tantas posiciones como símbolos tiene el alfabeto más una para el carácter de fin de palabra, y las búsquedas se continúan realizando por indexación. En los árboles Patricia las cadenas representadas son almacenadas en los nodos del árbol ***siempre***, esto no depende de la implementación que se elija sino que forma parte de la definición de la estructura. La Figura 11 muestra el trie analizado en la Figura 10 en el cual se han señalado (oscurecido) los nodos que forman parte de un camino unario.

Como puede verse todos los nodos marcados tienen un solo hijo, entonces éstos son los que desaparecen en el Patricia. La Figura 12 muestra el árbol Patricia obtenido a partir del trie de la Figura 11, los nodos unarios ya no forman parte del árbol y, como se dijo, la cadena completa es almacenada en el árbol. Además, dado que ya no hay un nodo por cada carácter de la secuencia, el simple hecho de ir leyendo los caracteres de entrada y avanzando en el árbol no siempre es posible, esto se debe a que

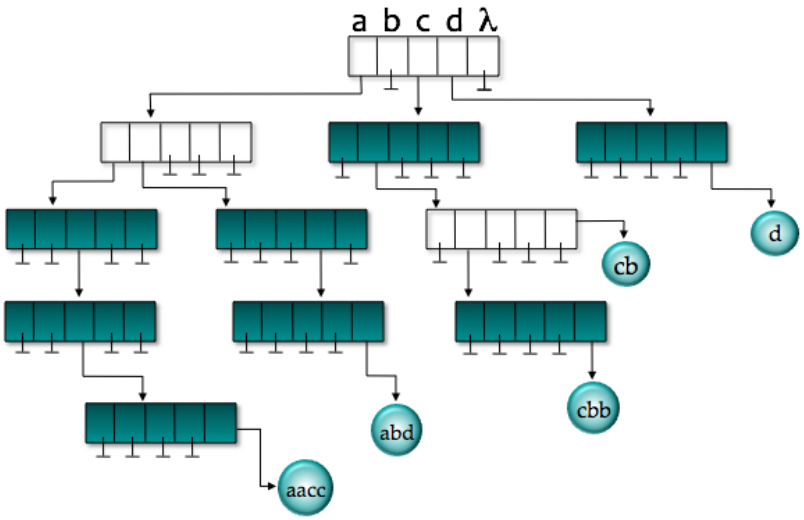


Figura 11: Nodos unarios del trie

habrá caracteres que no tengan un nodo asociado. Para saber qué carácter de la secuencia buscada hay que mirar para avanzar en el Patricia, se debe almacenar en cada nodo (en la figura aparece al lado del mismo) un número que indica la posición que ocupa, dentro de la secuencia, el carácter que se utilizará en ese nodo como índice para decidir por qué rama se debe seguir. Los caracteres que estén entre el último utilizado como índice y el que aparece en el nodo corriente serán *ignorados*.

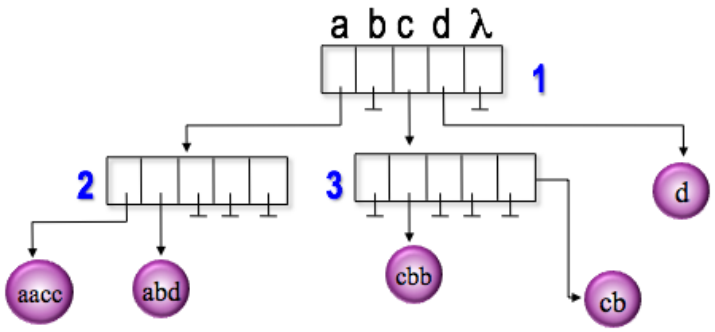


Figura 12: Árbol Patricia sobre el alfabeto $\{a, b, c, d\}$

En esta estructura las búsquedas se realizan de manera muy similar a la descripta para el trie, solo que ahora no siempre se comienza siguiendo el camino indicado por primer carácter de la cadena buscada. Entonces, dada una palabra a buscar se toma el carácter indicado por el número almacenado en el nodo corriente, en este caso la raíz, y ese carácter es utilizado como subíndice en este nodo para avanzar en el árbol. Al alcanzar el próximo nodo este proceso es repetido: leo el carácter de la secuencia de consulta que se indica en el nuevo nodo y lo uso como subíndice para seguir. Si todos los caracteres que se indican el los nodos visitados coinciden y llegamos a una hoja, se debe verificar que los caracteres *no comparados* son los que se buscan; es decir, si la cadena allí almacenada concuerda con la palabra que estamos buscando, y recién allí se confirmará si los caracteres que se saltaron son los correctos o no. Si la palabra coincide entonces la búsqueda fue exitosa, sino se fracasó.

Considerando un ejemplo se tratará de aclarar el proceso de búsqueda en un árbol Patricia. Sea ***aacc***λ

la secuencia buscada y el árbol de la Figura 12 la estructura que representa el conjunto de secuencias almacenadas. Entonces: se accede al nodo raíz del árbol para saber qué carácter de la cadena se debe leer, según la figura el primer carácter que permitirá avanzar en el árbol es el que está en la primera posición, como éste es una *a*, se toma el puntero correspondiente a esta letra; siguiendo ese puntero se alcanza un nuevo nodo cuyo valor indica que es el segundo carácter de la secuencia buscada el siguiente a considerar, usando éste como subíndice alcanzo una hoja del árbol, pero a pesar de que la cadena buscada tiene longitud cuatro sólo se compararon dos caracteres, por lo que no es posible asegurar que la palabra almacenada en esa hoja sea la buscada, entonces hay que comparar ambas secuencias carácter a carácter para determinar si la cadena fue encontrada o no, en este caso la consulta resultó exitosa ya que las secuencias coincidieron.

La Figura 13 muestra la localización de la secuencia ***aadc***λ, en ella puede verse, al lado de cada nodo, el carácter de la secuencia que fue leído en cada paso para establecer el puntero a seguir; el primer carácter leído fue una *a*, el primer símbolo de la cadena, y luego se consideró el segundo, otra *a*; también se muestra un camino más grueso que señala los punteros seguidos en cada momento, y como última acción se comparan las dos secuencias evidenciando que la cadena buscada no es la encontrada, luego la búsqueda fracasó. No siempre se compara el primer carácter de la cadena de consulta al localizarla. Por ejemplo, si en un Patricia sólo hubiera dos palabras almacenadas: '*acampar*' y '*acalorado*' el nodo de la raíz debería informar que el primer carácter que debe ser chequeado será el cuarto, porque recién en esa letra se diferencian las dos palabras.

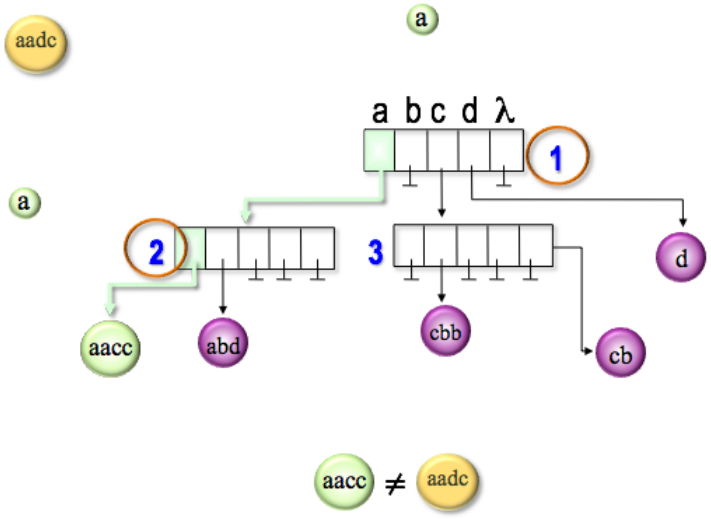


Figura 13: Búsqueda de la secuencia '*aadc*λ' en un árbol Patricia

El método de búsqueda en un Patricia difiere en dos puntos del utilizado en un trie: por un lado, se deben testear sólo los caracteres indicados en los nodos y no todos los caracteres de la cadena buscada, y por otro, la búsqueda siempre termina con la comparación completa de la secuencia buscada con la encontrada en la hoja.

Al momento de realizar un alta en un Patricia primero se debe buscar la secuencia a insertar para saber si pertenece al conjunto representado o no, cuando la localización fracasa el proceso de alta puede comenzar. Esta operación es más complicada que en un trie ya que en éste al llegar a un puntero a nil, se sabe si la cadena debe ser insertada en el nodo corriente o cuántos nodos nuevos (uno por cada carácter que falta leer para terminar la palabra) se necesitan agregar a la estructura para poder almacenar la secuencia. En cambio para los Patricia el trabajo necesario para saber dónde se debe insertar la nueva cadena es mayor debido a que los caracteres que se corresponden con nodos unarios, que ya no están,

son saltados y no se sabe cuáles son hasta llegar a una hoja.

Como se explicó, la localización en un árbol Patricia siempre finaliza con la comparación de la secuencia de búsqueda completa, y este proceso permite obtener información importante para el alta ya que, al comparar carácter a carácter, se puede encontrar la letra *de más a la izquierda* en la cual difieren la secuencia almacenada de la buscada. Con esta información se recorre el árbol por segunda vez, comparando esa posición con las posiciones guardadas en los nodos que están en el camino de búsqueda. Si se alcanza un nodo en el cual la posición del carácter a considerar es mayor que la posición en la que se diferencian la cadena a insertar de la encontrada, queda claro que durante la localización se saltó un carácter que deberá ser considerado para poder encontrar la nueva cadena a insertar; entonces debe agregarse a la estructura un nuevo nodo para testear ese carácter. Ese nodo se inserta como *padre* del que tiene una posición de carácter a considerar mayor a la posición en la que las dos cadenas se diferencian; la nueva cadena se cuelga de él en una hoja. Suponga que se quiere dar de alta la cadena '*cabcbλ*' en el árbol de la Figura 13, éste quedará como se muestra en la Figura 14 luego de esa inserción. Puede observarse que luego del alta el árbol tiene dos nodos nuevos: uno interno que informa que se debe comparar el caracter "2" para saber qué rama seguir; y otro que es una hoja conteniendo la cadena insertada.

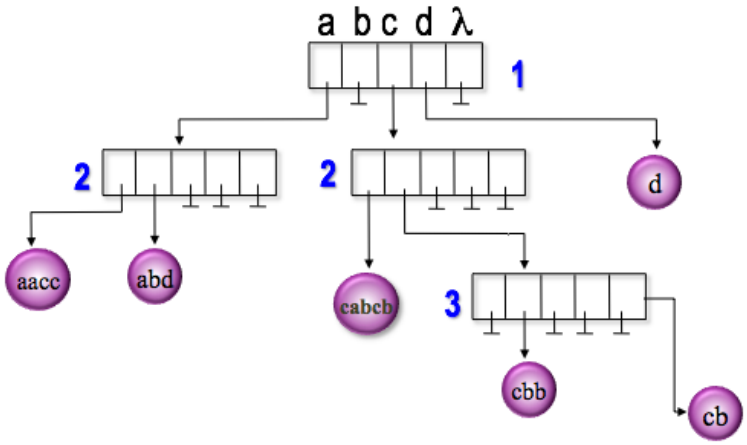


Figura 14: Inserción de la secuencia '*cabcbλ*' en un árbol Patricia

Si en cambio, llegamos a una hoja sin encontrar un nodo cuyo valor sea mayor que la posición en la que ambas secuencias difieren, entonces se agregará un nodo como *padre* de esta hoja y la nueva palabra será una hoja hijo de él; ver un ejemplo de este caso en la Figura 15, con el alta de la cadena '*aadcλ*'. Como puede verse en todos los casos el Patricia sólo agrega *un* nodo a la estructura durante un alta, mientras que en un trie los nodos agregados podían ser varios, dependiendo de la cantidad de caracteres de la cadena que no están en la estructura, y la mayoría unarios.

Como se dijo, la Figura 14 muestra la inserción de la cadena *cabcbλ* en un árbol Patricia; analicemos el proceso con más detalle: se busca *cabcbλ* y se encuentra *cbbλ* entonces se fracasó. Puede verse que la posición en la que las cadenas buscada (*cabcbλ*) y encontrada (*cbbλ*) difieren es la *dos*, por lo que al recorrer por segunda vez el árbol y alcanzar un nodo que almacena la posición tres, como la próxima a ser considerada, se determina que debe agregarse un nuevo nodo, que guardará como siguiente carácter a considerar el dos; de él colgarán tanto la nueva cadena como el nodo que ya existía. En la Figura 15 se puede analizar la inserción de la secuencia *aadcλ*, aquí luego de buscar, fracasar y conocer que ambas cadenas, encontrada (*aaccλ*) y a insertar (*aadcλ*) difieren en la posición *tres*, se vuelve a ingresar al árbol y en esta segunda recorrida se alcanza nuevamente la hoja porque no hay, en este camino, un nodo en el cual la posición a considerar sea mayor que tres, entonces el nuevo nodo se agrega como padre de las

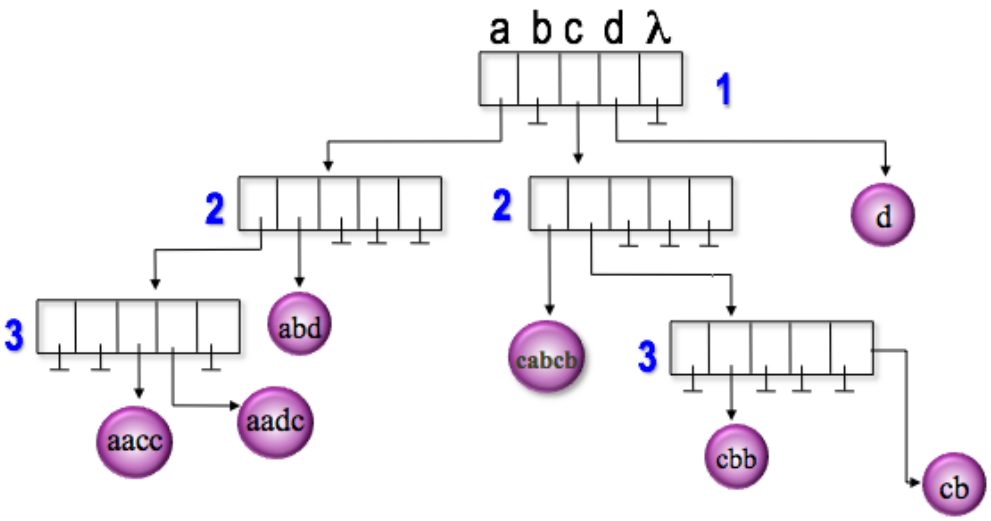


Figura 15: Alta de la secuencia '*aadcλ*' en un árbol Patricia

cadenas encontrada (*aaccλ*) e insertada (*aadcλ*), y guarda como próximo carácter a considerar el *tres*.

Las bajas en un árbol Patricia, también son parecidas a las de un trie: se busca la secuencia a dar de baja y al encontrarla se elimina la cadena y se actualiza el puntero del nodo padre poniéndolo en nil, una vez hecho esto hay que controlar los punteros distintos de nil que tiene este nodo. Como el Patricia elimina los nodos unarios, sólo aquellos que tengan dos punteros o más distintos de nil se conservarán; un nodo que tenga un solo puntero debe eliminarse porque éste forma parte de un camino unario. En este caso, al dar de baja este nodo, el puntero por el cual se llegó a él, el de su padre, también debe ser actualizado: si el nodo eliminado no tenía hijos su valor será nil; si el nodo eliminado tenía un hijo entonces el puntero accederá a la cadena que colgaba de ese nodo; en este momento se deberá volver a chequear si el nodo actualmente corriente tiene un solo puntero distinto de nil, si es así también se eliminará y su padre será actualizado. Este proceso se repetirá hasta alcanzar algún nodo que tenga dos o más punteros distintos de nil, en este momento habrá terminado la baja.

Considerando el Patricia de la Figura 15 se quiere dar de baja la secuencia *aacc* entonces se localiza la misma, se encuentra, se da de baja la cadena liberando la memoria utilizada y actualizando el puntero en el padre (se pone en nil) y luego se chequea la cantidad de punteros distintos de nil que quedaron en el nodo padre. En este caso vemos que un único puntero señalando la cadena *aadcλ* es el que está presente, por lo que este nodo será eliminado también, y la cadena colgará del padre del nodo eliminado (Figura 16). Ahora se confirma la cantidad de punteros útiles del nuevo nodo corriente, éste tiene dos punteros distintos de nil, entonces el proceso termina y la baja está completa.

Como puede observarse, el espacio ocupado por el árbol Patricia es menor que el que usa el trie, sin embargo mantiene el promedio de caracteres examinados durante la búsqueda próximo a $O(\log_M N)$ para un Patricia construido con N palabras aleatorias con igual probabilidad de ser consultadas; esto se debe a que aunque la cantidad de caracteres comparados mientras se atraviesa el árbol puede ser mucho menor que en un trie, al llegar a la hoja que contiene una cadena *siempre* se debe hacer una comparación completa por la secuencia buscada.

En muchos casos se utiliza el alfabeto binario para codificar los caracteres del alfabeto con el que se está trabajando, por lo que cada secuencia será una cadena de ceros y unos, lo que da lugar a un árbol Patricia binario, luego, si $M = 2$ entonces tenemos $\log_2 N$ comparaciones en promedio y alrededor de $2\ log_2 N$ caracteres comparados en el peor caso. Además, en todos los casos, estos costos son inde-

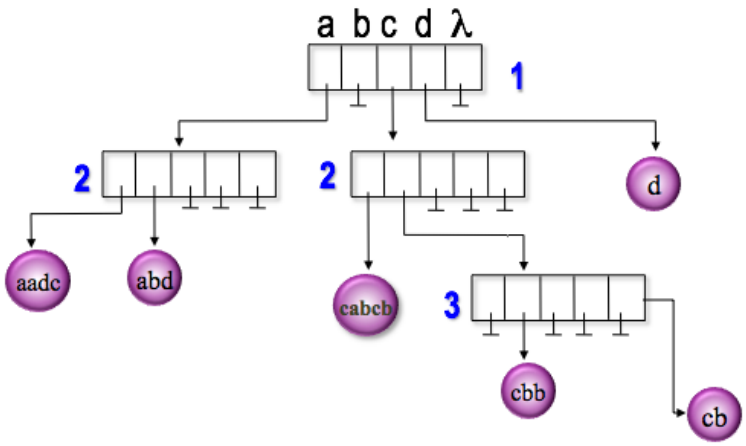


Figura 16: Baja de la secuencia '*aaccλ*' en un árbol Patricia

pendientes de la longitud de la cadena y permite secuencias de longitud variable, y cuanto más largas mejor.

Dado que los árboles Patricia son muy eficientes en el manejo de secuencias largas, una variante de los mismos al momento de mantener en el nodo el carácter que debe examinarse, guarda en cada nodo la *cantidad de caracteres a saltar* para alcanzar el próximo a comparar en lugar de la posición que ocupa este carácter en la palabra, esto permite que los números almacenados en cada nodo sean más chicos; por ejemplo si la cadena tuviera una longitud $l = 1525$ caracteres y el último es uno de los que decide que rama seguir, será más económico guardar que desde el penúltimo símbolo comparado tengo que avanzar 10 caracteres, por ejemplo, que almacenar la posición de ese carácter (1525). Para ello hay que establecer de antemano cuál será la convención para la raíz del árbol, si se almacena la posición del primer carácter a examinar, y de allí en más los saltos, o se supone que siempre se lee el primer símbolo de la cadena buscada y se almacena cuántos hay que saltar desde él para alcanzar el primer carácter a considerar.

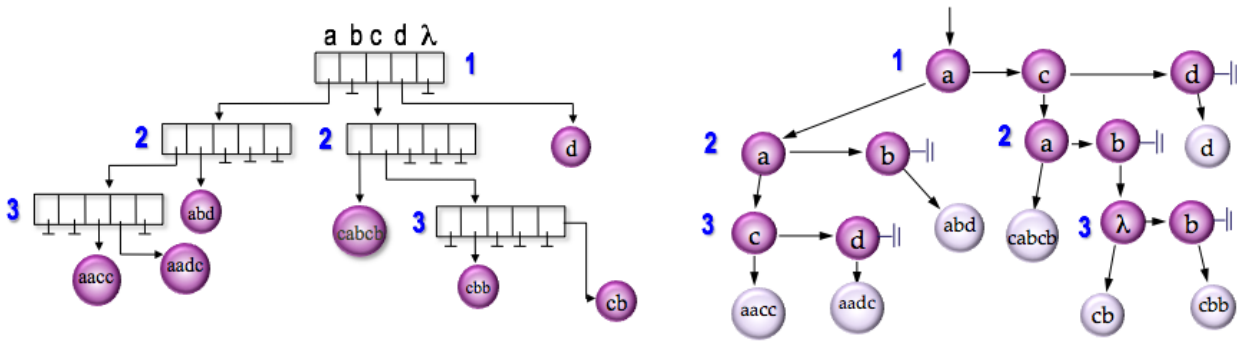


Figura 17: Transformada de Knuth sobre el árbol Patricia

Otra variante sobre el árbol Patricia es eliminar los punteros a nil obteniendo un árbol de aridad variable, y aplicarle la transformada de Knuth para obtener un árbol binario que sólo utiliza los nodos distintos de nil, sólo que ahora además de almacenar en el nodo el carácter correspondiente, se debe guardar la posición del símbolo a considerar. La Figura 17 muestra cómo quedaría el Patricia del ejemplo si aplicáramos la transformada de Knuth.

Como se dijo el Trie tiene muy buen tiempo de acceso durante la búsqueda de una secuencia y la cantidad de comparaciones durante la misma en promedio es cercana a $log_M N$, pero estos beneficios puede degradarse si el conjunto representado está formado por cadenas muy largas. Además están los altos costos de espacio, que es la principal desventaja del trie. En este apunte se mencionan algunas ideas que intentan mejorar estos costos, aunque algunas de ellas pierden eficiencia durante las búsquedas. Entre las sugerencias analizadas surge una manera elegante y eficiente de reducir espacio eliminando los caminos unarios, es el árbol Patricia. Esta estructura se adapta perfectamente a cadenas largas, es más, su rendimiento es mejor cuanto más largas son las cadenas, es particularmente eficiente cuando trabaja con secuencias de longitud variable y potencialmente largas. Con el árbol Patricia se espera que el número de caracteres inspeccionados durante una búsqueda, aún con cadenas largas, sea proporcional a $log_M N$ en promedio. O sea que mantiene la cantidad de comparaciones en promedio del trie pero utiliza mucho menos espacio.

Aplicaciones

Con muchas variaciones, los tries tienen un vasto ámbito de aplicaciones. Entre las más útiles, cómo se comentó anteriormente, se encuentra el almacenamiento de *diccionarios*, siempre considerando alguna de las alternativas mencionadas para tratar de mejorar el requerimiento de espacio, dado que, para una cantidad grande de palabras el tamaño del trie es muy importante si se necesita que éste resida en memoria principal. Estas aplicaciones, como las que se encuentran en los teléfonos móviles, se aprovechan de la capacidad de los tries para hacer búsquedas, inserciones y borrados rápidos.

Los tries también son útiles en la implementación de algoritmos de correspondencia aproximada, como los usados en el software de *corrección ortográfica*. Una herramienta indispensable para cualquier procesador de texto es un revisor ortográfico, el cual permite al usuario encontrar tantos errores de ortografía como sea posible. Dependiendo de la complejidad del revisor ortográfico, el usuario puede incluso ver las correcciones posibles. Los correctores ortográficos se utilizan principalmente en un entorno interactivo, el usuario puede invocarlos en cualquier momento mientras se utiliza el procesador de texto, hacer correcciones sobre la marcha, incluso antes de procesar el archivo completo. Esto requiere un programa de procesamiento de texto y además un módulo de revisión ortográfica. El núcleo de un revisor ortográfico es una estructura de datos que permite el acceso eficiente a las palabras en un diccionario. Este diccionario probablemente tiene miles de palabras, así que el acceso al mismo tiene que ser muy rápido para procesar un archivo de texto en un período razonable de tiempo. Entre muchas posibles estructuras de datos, el trie se elige para almacenar las del diccionario. Después de que el corrector ortográfico es invocado, se crea el trie primero y después la corrección ortográfica se realiza efectivamente.

Otra posible aplicación de los árboles digitales, el Patricia en particular, son las *tablas de enrutamiento o encaminamiento* de los routers. Una tabla de enrutamiento, también conocido como una tabla de encaminamiento, es un documento electrónico que almacena las rutas a los diferentes nodos en una red informática. Los nodos pueden ser cualquier tipo de dispositivo electrónico conectado a la red. La Tabla de enrutamiento generalmente se almacena en un router o en una red. Cuando los datos deben ser enviados desde un nodo a otro de la red, se hace referencia a la tabla de enrutamiento con el fin de encontrar la mejor ruta para la transferencia de datos. Así que cuando un paquete de datos llega a un nodo en particular, usa la tabla de rutas para encontrar la dirección del siguiente nodo. Una vez que llega a ese nodo, de nuevo usa la tabla de enrutamiento para la dirección del siguiente salto, y así sucesivamente, hasta llegar al destino final.

Bibliografía

Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching, Vol III* Addison–Wesley; ISBN 0-201-03803-X.

Harry R. Lewis, Larry Denenberg. *Data Structures & Their Algorithms.* HarperCollin Publishers; ISBN 0-673-39736-X.

Jeffrey Ullman, Alfred Aho, John Hopcroft. *Estructuras de Datos y Algoritmos, primera edición.* Addison–Wesley; ISBN 968-444-345-5.

Robert Kruse, CL Tondo, Bruce Leung. *Data Structure y Programing* ***Design****, segunda edición.* Prentice Hall; ISBN 013-288-366-X.

Robert Sedgewick. *Algorithms in C, 3rd. Edition. Parts 1-4: Fundamentals Data Structured, Sorting and Searching.* Addison–Wesley; ISBN 0-201-31452-5.

Drozdek, A. *Estructura de Datos Y Algoritmos Con Java* - 2^{da} edición. 768 páginas - 2007. Cengage Learning Latin America - ISBN 9789706866110