TEMA 3. DICCIONARIOS. BÚSQUEDA CON TOLERANCIA

Contenidos basados en los materiales de otros cursos como los de Manning, Baeza, Jurafsky.

Contenidos

En este tema se desarrollan:

- Estructuras de datos utilizadas para búsquedas de los términos del vocabulario en un fichero invertido.
- Los mecanismos de búsqueda de términos usando comodín “automat*”.
- Búsqueda de términos con errores tipográficos.

Bibliografía

A Introduction to Information Retrieval:
Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze.
Cambridge University Press, ***2009****.*
Capítulo 3

1.1 TIPOS DE DICCIONARIOS: TABLAS HASH

Tablas hash (o Tablas de dispersión)

- Estructura de datos especialmente diseñada para la implementación de DICCIONARIOS.
  - Se pueden conseguir los siguientes costes:
    - Búsqueda. O(1)
    - Inserción. O(1)
    - Borrado. O(1)
- Método: asociar una clave a cada elemento del dominio y asociarle un elemento de un vector.

La tabla Hash es una de las estructuras más útiles para representar un diccionario.

Cada término es asociado a un entero que representa una posición en la tabla de almacenamiento.

Como el espacio de términos es muy grande en comparación con el espacio de posiciones en la tabla, entonces

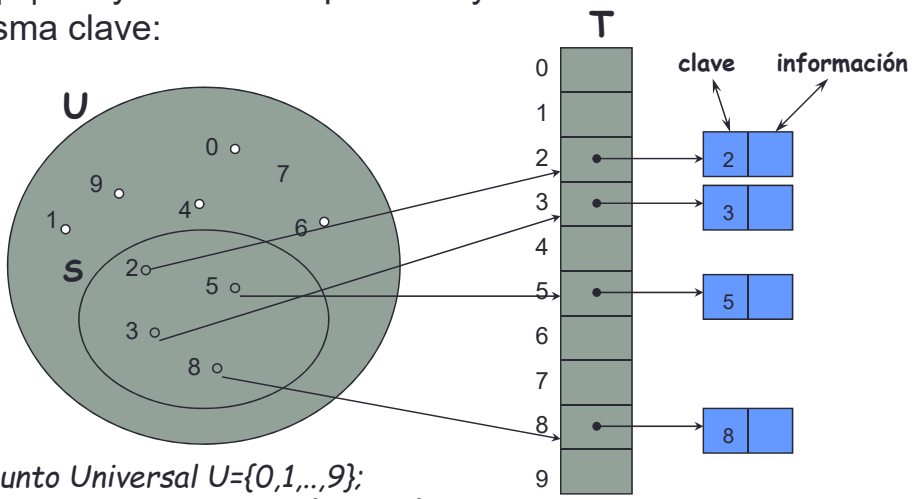
se producen colisiones, que deben resolverse mediante diversos mecanismos.

En el momento de la búsqueda se accede mediante esta tabla a la posición de la palabra que contiene el puntero

a su correspondiente posting list.

Tablas hash

- Caso sencillo: el universo de valores posibles es pequeño
- Si |U|<<< y asumimos que no hay dos elementos con la misma clave:



Conjunto Universal U={0,1,..,9};
Conjunto a representar S={2,3,5,8}

Es decir, se considera que cada elemento del universo de valores tiene asignado una posición en el vector T, cuya talla por tanto debe ser |U|

Tablas hash

Caso más complejo:

Sea |U| el tamaño del conjunto universal y |S| el tamaño del conjunto a representar. El espacio en memoria para representar S es $O(|U|)$.

- Si |U|>>>> puede ser que no tengamos memoria suficiente para representar todos los elementos posibles
- Si |S|<<<<|U| se utiliza mucho espacio cuando el conjunto que hay que representar es pequeño.

Para evitar estos problemas se van a representar los elementos del universo de valores en un vector T de talla menor que |U|, tal como se explica en la siguiente diapositiva

Tablas hash

Para construir una Tabla Hash:

- Dividir el conjunto **U** en un número finito **B** de clases
- Usar un vector **T[0..B-1], con B<<|U|.**
- Cada elemento de **U** se identifica por una clave **k**.
- Definir una función hash que asocie a cada elemento de **U** (cada clave k) un valor entre **[0..B-1].**
- Un elemento de clave k se almacena en la posición **h(k)**.
- **T** es una **Tabla hash** y **h** es la **función hash.**
- A cada una de las **T[j]** se les llama **cubeta.**

Ejemplo de función hash: h(x) = x MOD B

Características deseables:

- Debe ser fácil de calcular
- Debe minimizar el nº de colisiones
- Debe distribuir los elementos de forma aleatoria

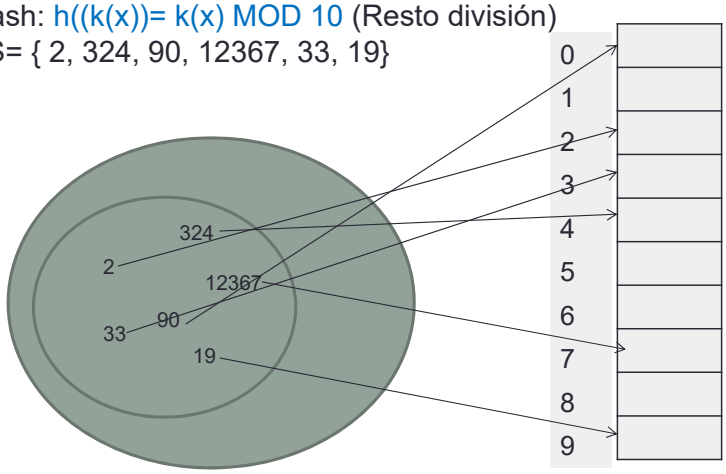
Es importante tener claro que por una parte hay un número que llamamos clave k, que identifica al elemento de U. Tened en cuenta que un elemento puede ser una palabra o una imagen,…, y se ha de representar mediante un número (en la mayoría de los casos ese número es único para cada elemento de U, pero no es imprescindible que eso ocurra).

Luego está la función de dispersión h, que asocia a esa clave una posición en el vector T. Aquí sí que es lógico que se produzcan colisiones, es decir, que dos o más claves vayan a parar a la misma posición del vector, puesto que estamos direccionando un número grande de elementos |U| en un número mucho más pequeño de posiciones del vector B.

Ejemplo: Representar elementos de Z^+

- U= Z^+, {enteros positivos}
- Clave del elemento x: k(x)=x (Es el mismo valor del número)
- Talla de la tabla Hash: B=10
- Función hash: h((k(x))= k(x) MOD 10 (Resto división)
- Conjunto S= { 2, 324, 90, 12367, 33, 19}

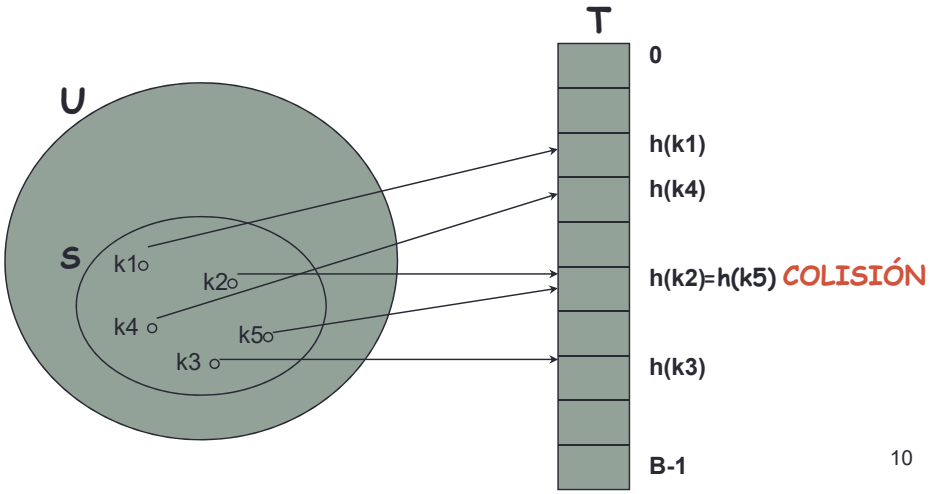
h(2)=2
h(324)=4
h(90)=0
h(12367)=7
h(33)=3
h(19)=9



Tablas hash

Función hash

h: U -> {0,1,2,..,B-1}, Coste O(1)



10

En este ejemplo las claves k2 y k5 van a parar a la misma posición de T.

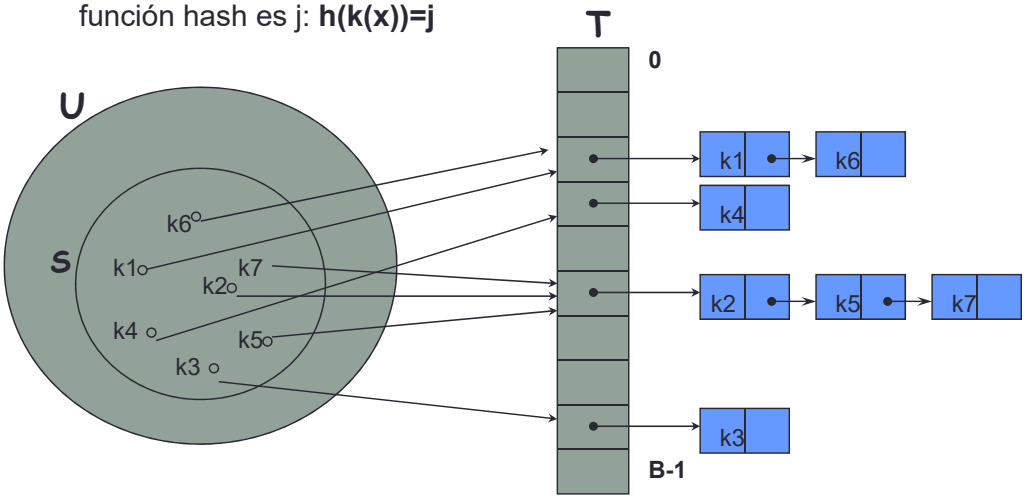
Tratamiento de colisiones

- Objetivo es definir una función hash que minimice el número de colisiones.
- Dos métodos para tratar las colisiones:
  - A. Por encadenamiento
  - B. Por direccionamiento abierto

A. Por encadenamiento

- Construir una lista de claves que tienen el mismo valor de la función hash

  T[j]: puntero a la cabeza de lista de aquellos elementos cuya función hash es j: **h(k(x))=j**



12

Es decir, si dos o más elementos van a parar a la misma posición i del vector T al aplicar la función de hashing h a sus claves, entonces se almacenan en una lista cuyo puntero inicial está en T[j].

B. Por direccionamiento abierto

Diferentes opciones de redirección (implementando circularidad):

A. Buscar secuencialmente a partir de *indiceHash* la siguiente posición libre de la tabla

B. Buscar sucesivamente las posiciones $indiceHash+1^2$, $indiceHash+2^2$, ..., $indiceHash+i^2$,.

C. Usar otras funciones: $h(x)=h1(x) + i\ h2(x)$

Donde i = número de intentos

En este tipo de direccionamiento se utiliza exclusivamente el vector T y cuando hay colisión, es decir queremos almacenar un elemento en una posición ya ocupada, lo que se hace es buscar una posición libre de T (por diversos mecanismos).

Por direccionamiento abierto….cont1

ESTRATEGIA DE REDISPERSIÓN LINEAL ("siguiente posición"):

- No eficiente. Larga secuencia de intentos

$h_i(x) = (h_{i-1}(x) + 1) \text{ MOD } B$

ESTRATEGIA DE REDISPERSIÓN ALEATORIA:

$h_i(x) = (h_{i-1}(x) + c) \text{ MOD } B$ con c>1

- Sigue produciendo AMONTONAMIENTO

(c y B no deben tener factores primos comunes mayores que 1)

ESTRATEGIA DE REDISPERSIÓN CON 2ª FUNCION HASH:

$h_i(x) = (h_{i-1}(x) + f(x)) \text{ MOD } B$

$f(x) = (x \text{ MOD } (B-1)) + 1$

(B debe ser primo)

Donde i = número de intentos

Estos son diversas estrategias de redispersión. Cada nueva posición del vector $h_i(x)$ se calcula a partir de la anteriormente calculada $h_{i-1}(x)$. Esto se hace hasta encontrar una posición vacía.

Ejercicio#1.

Completar la inserción en una tabla hash cerrada de tamaño *B=7*, con función hash *h(x)= x MOD B*, y con estrategia de redispersión 2ª función hash, los siguientes elementos: *23, 14, 9, 6, 30, 12, 18*

$h_i(x) = (h_{i-1}(x) + f(x)) \text{ MOD } B$ siendo $f(x) = (x \text{ MOD } (B-1)) + 1$

Ejercicio#1.

Completar la inserción en una tabla hash cerrada de tamaño *B=7*, con función hash *h(x)= x MOD B*, y con estrategia de redispersión 2ª función hash, los siguientes elementos: *23, 14, 9, 6, 30, 12, 18*

$h_i(x) = (h_{i-1}(x) + f(x)) \text{ MOD } B$ siendo $f(x) = (x \text{ MOD } (B-1)) + 1$

h(23)=23 MOD 7=***2***
h(14)=14 MOD 7=***0***
h(9)=9 MOD 7=2
f(9)=(9 MOD 6)+1=4
h_1(9)=(2+4) MOD 7=***6***

0	14
1	
2	23
3	
4	
5	
6	9

Cuando se calcula h(9) se detecta que ya está ocupada la posición 2, por lo que se vuelve a calcular con la fórmula de redispersión.

Ejercicio#1(Solución).

Insertar en una tabla hash los siguientes elementos: *23, 14, 9, 6, 30, 12, 18*

función hash $h(x)= x \text{ MOD } B$

Redispersión $h_i(x) = (h_{i-1}(x) + f(x)) \text{ MOD } B$ siendo $f(x) = (x \text{ MOD } (B-1)) + 1$

h(6)=6 MOD 7=6

f(6)=(6 MOD 6)+1=1

$h_1(6)=(6+1) \text{ MOD } 7=0$

$h_2(6)=(0+1) \text{ MOD } 7=\mathbf{1}$

0	14
1	
2	23
3	
4	
5	
6	9

0	14
1	6
2	23
3	
4	
5	
6	9

Ejercicio#1(Solución).

Insertar en una tabla hash los siguientes elementos: *23, 14, 9, 6, 30, 12, 18*

función hash $h(x)= x \text{ MOD } B$

Redispersión $h_i(x) = (h_{i-1}(x) + f(x)) \text{ MOD } B$ siendo $f(x) = (x \text{ MOD } (B-1)) + 1$

h(6)=6 MOD 7=6

f(6)=(6 MOD 6)+1=1

$h_1(6)=(6+1) \text{ MOD } 7=0$

$h_2(6)=(0+1) \text{ MOD } 7=\mathbf{1}$

h(30)=30 MOD 7=2

f(30)=(30 MOD 6)+1=1

$h_1(30)=(2+1) \text{ MOD } 7=\mathbf{3}$

0	14
1	6
2	23
3	
4	
5	
6	9

0	14
1	6
2	23
3	30
4	
5	
6	9

Ejercicio#1(Solución).

Insertar en una tabla hash los siguientes elementos: *23, 14, 9, 6, 30, 12, 18*

función hash $h(x)= x \text{ MOD } B$

Redispersión $h_i(x) = (h_{i-1}(x) + f(x)) \text{ MOD } B$ siendo $f(x) = (x \text{ MOD } (B-1)) + 1$

h(6)=6 MOD 7=6

f(6)=(6 MOD 6)+1=1

$h_1(6)=(6+1) \text{ MOD } 7=0$

$h_2(6)=(0+1) \text{ MOD } 7=\mathbf{1}$

h(30)=30 MOD 7=2

f(30)=(30 MOD 6)+1=1

$h_1(30)=(2+1) \text{ MOD } 7=\mathbf{3}$

h(12)=12 MOD 7=***5***

0	14
1	6
2	23
3	30
4	
5	
6	9

0	14
1	6
2	23
3	30
4	
5	12
6	9

Ejercicio#1(Solución).

Insertar en una tabla hash los siguientes elementos: *23, 14, 9, 6, 30, 12, 18*

función hash $h(x)= x \text{ MOD } B$

Redispersión $h_i(x) = (h_{i-1}(x) + f(x)) \text{ MOD } B$ siendo $f(x) = (x \text{ MOD } (B-1)) + 1$

h(6)=6 MOD 7=6

f(6)=(6 MOD 6)+1=1

$h_1(6)=(6+1) \text{ MOD } 7=0$

$h_2(6)=(0+1) \text{ MOD } 7=\mathbf{1}$

h(30)=30 MOD 7=2

f(30)=(30 MOD 6)+1=1

$h_1(30)=(2+1) \text{ MOD } 7=\mathbf{3}$

h(12)=12 MOD 7=***5***

h(18)=18 MOD 7=***4***

0	14
1	6
2	23
3	30
4	18
5	12
6	9

Nº TOTAL DE INTENTOS HASTA LA CLAVE 18:

11

Cómo obtener una clave de un elemento

(p.ej. de una cadena de caracteres)

1) Método sencillo: Sumar los códigos asociados a los caracteres

elemento		clave
casa	→	99 + 97 + 115 + 97 = 408
hola	→	104 + 111 + 108 + 97 = 420

Problema:

hola	→	104 + 111 + 108 + 97 = 420
teja	→	116 + 101 + 106 + 97 = 420

2) Funciones polinomiales: para mejorar la calidad de la función de dispersión se puede ponderar la posición de cada caracter dentro de la clave:

$$F(c) = c_0 \cdot a^{k-1} + c_1 \cdot a^{k-2} + \ldots + c_{k-2} \cdot a^1 + c_{k-1} \quad \text{, con } a>1.$$

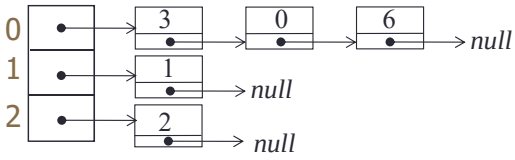
Ejemplo con a=2

hola	→	$104 \cdot 2^3 + 111 \cdot 2^2 + 108 \cdot 2 + 97 = 1589$
teja	→	$116 \cdot 2^3 + 101 \cdot 2^2 + 106 \cdot 2 + 97 = 1641$

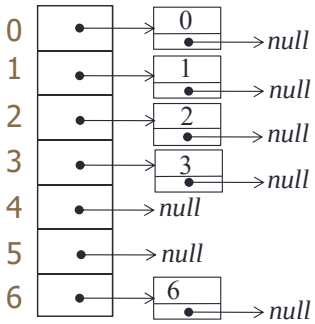
Problema del primer método (método sencillo): La suma de componentes no es una buena solución ya que es fácil que dos entradas distintas tengan la misma clave. El segundo método es mejor porque dificulta que dos palabras tengan la misma clave.

2.3. Rehashing.

- El número de colisiones puede crecer excesivamente si el Factor de Carga es demasiado alto. El Factor de Carga es el cociente entre el número de elementos que contiene la tabla y el número de cubetas.
- El ***rehashing*** consiste en incrementar el tamaño de la tabla hash, recolocando los elementos, reduciendo así su grado de ocupación



FC = 5 elementos/3 cubetas = 1.67



FC = 5 elementos/7 cubetas= 0.71

Hay que tener en cuenta que hacer un rehashing es costoso ya que hay que recalcular todos los elementos de la tabla.

Tabla hash. Pros y Contras

- Operaciones en tiempo de consulta (on-line): Calcular el valor del hashing del término y localizarlo en la tabla resolviendo colisiones.
- Pros:
  - La localización de un elemento es más rápida que en un árbol. Puede ser de tiempo constante O(1)
- Contras:
  - No se pueden encontrar pequeñas variantes, ya que el hashing las puede colocar en lugares muy distantes (p.e. *resume* vs *résumé*).
  - No se pueden hacer búsquedas de prefijos (p.e. todos los términos que empiezan por "*automat*")
  - Necesidad de hacer un rehashing de todo el vocabulario periódicamente si el vocabulario crece contínuamente.

Ejercicio#2.

Completar la inserción en una tabla hash cerrada de tamaño B=11, con función hash H(x)= x MOD B, y con estrategia de redispersión 2ª función hash, los siguientes elementos:

51, 14, 3, 7, 18, 30.

$h_i(x) = (h_{i-1}(x) + f(x)) \text{ MOD } B$ siendo $f(x) = (x \text{ MOD } (B-1)) + 1$

Ejercicio#2(Solución).

Completar la inserción en una tabla hash cerrada de tamaño B=11, con función hash **H(x)= x MOD B**, y con estrategia de redispersión 2ª función hash, los siguientes elementos:

51, 14, 3, 7, 18, 30.

$h_i(x) = (h_{i-1}(x) + f(x))$ MOD B siendo f(x) = (x MOD (B-1)) + 1

Solución:

x	H(x)	$h_1(x)$	$h_2(x)$
51	7		
14	3		
3	3	(3+3+1)MOD B=7	(7+3+1)MOD B=0
7	7	(7+7+1) MOD B=4	
18	7	(7+8+1) MOD B=5	
30	8		

La tabla queda como sigue:

0	1	2	3	4	5	6	7	8	9	10
3			14	7	18		51	30		

1.2. TIPOS DE DICCIONARIOS: ÁRBOLES

1.2.1. Árbol Binario de Búsqueda
1.2.2. B-Árbol

Árboles

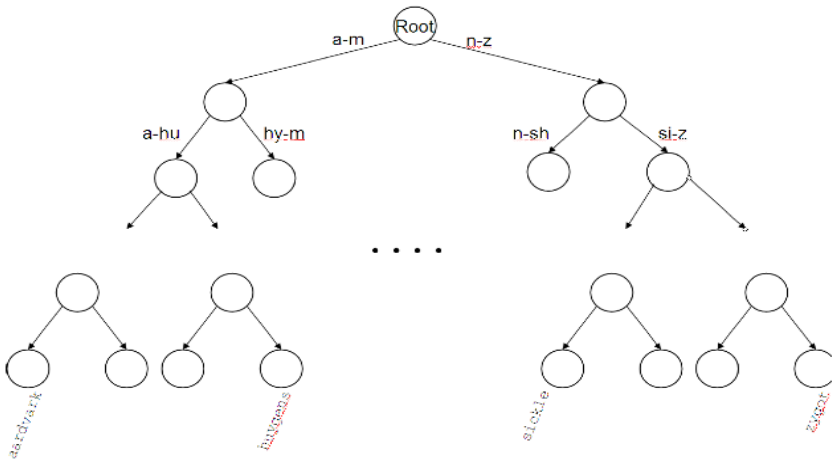
- En los árboles la búsqueda es dirigida por una comparación que se hace en cada uno de sus nodos
- Por ello se requiere un orden estándar de los caracteres para hacer la comparación (mayor, menor, igual). Aunque hay lenguas que no tienen este orden alfabético no es ningún problema definirlo.
- El más simple es el Árbol Binario de Búsqueda.
- El más habitual es el B-Árbol.

La búsqueda comienza en la raíz del árbol, y en cada nodo se realiza una comparación para decidir por qué rama se baja.

Las búsquedas tienen un coste logarítmico.

1.2.1. Árbol Binario de Búsqueda

Ejemplo, Árbol Binario de Búsqueda con cadenas de caracteres



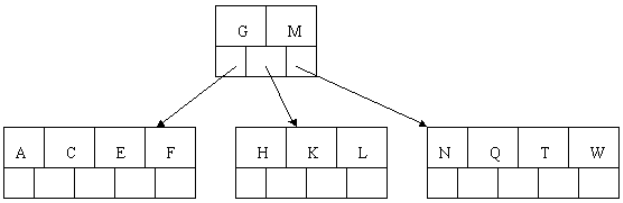
En cada nodo se toma una decisión en base al orden lexicográfico

Árboles binarios de búsqueda. Pros y contras

- Los árboles permiten resolver el problema de las búsquedas de prefijos (p.ej. encontrar todos los términos que comienzan por *automat*).
- La búsqueda es ligeramente más costosa que en la Tabla Hash: $O(\log M)$, donde M es la talla del vocabulario.
- El coste $O(\log M)$ sólo es cierto para árboles balanceados.
- Rebalancear árboles binarios es costoso.

1.2.2. B-Árbol

- Los B-árboles permiten que el número de subárboles de un nodo varíe entre un intervalo fijo, con ello mitigan el problema del rebalanceo.
- Búsquedas en un B-árbol: se realiza una (o varias) comparaciones en cada nodo para escoger el camino por el que descender.
- Ejemplo: B-Árbol de orden 5 (máx 5 hijos y 4 claves).

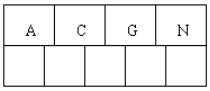


Para una búsqueda, en cada nodo interno se toma la decisión de descender por la única rama que es candidata a contener la palabra buscada. Cada rama admite un intervalo de palabras posibles.

Inserción en un B-Árbol

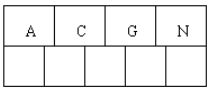
Supongamos que queremos insertar los siguientes caracteres en un B-árbol de orden 5 (máximo de 5 hijos y 4 claves): C N G A H E K Q M F W L T Z D P R X Y S

1. Insertar las cuatro primeras letras

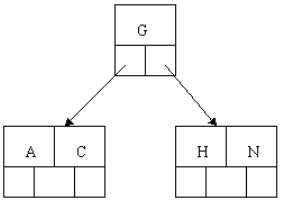


Las casillas de arriba contienen las letras insertadas, y las de debajo son punteros a los nodos que contendrán las letras comprendidas entre los intervalos definidos por esas letras de arriba. Es decir, el primer puntero apuntará a un nodo hijo con letras menores de A, el segundo puntero apuntará a un nodo hijo con letras comprendidas entre A y C, etc.

C N G A H E K Q M F W L T Z D P R X Y S

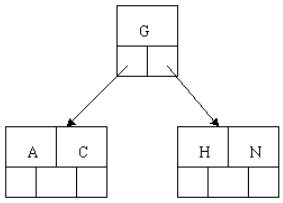


2. Como ya no hay sitio para la H, se desdobla este nodo en dos, que cuelgan de un nodo raíz, que es el valor de la mediana.

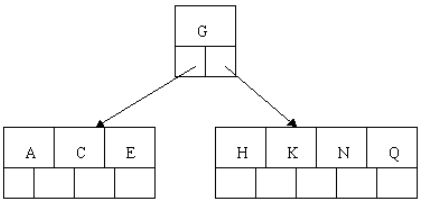


Al dividir por la G, el nodo superior se queda sólo con la G, y queda un hijo izquierdo, con valores menores que la G, y un nodo derecho con valores mayores de la G.

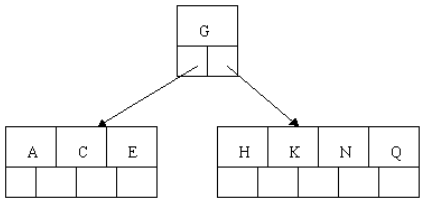
C N G A H E K Q M F W L T Z D P R X Y S



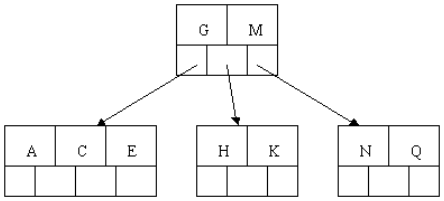
3. Para insertar E, K, y Q no hace falta dividir nodos.



C N G A H E K Q M F W L T Z D P R X Y S

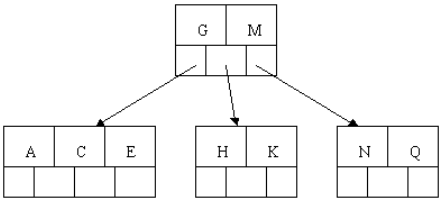


4. Insertar M requiere una división. En este caso la mediana es M, y es el que se sube al nodo padre.

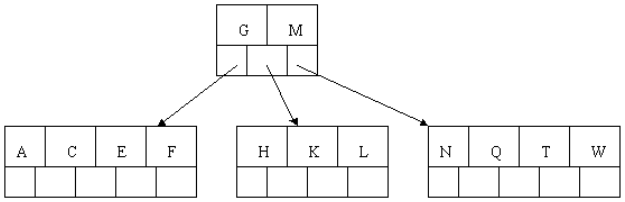


En este caso al insertar la M nos quedaría un nodo derecho con HKMNQ, que es demasiado grande. Por eso se divide en dos por la mediana M. Esa M sube al nodo de arriba, y genera un nuevo intervalo en este nodo superior.

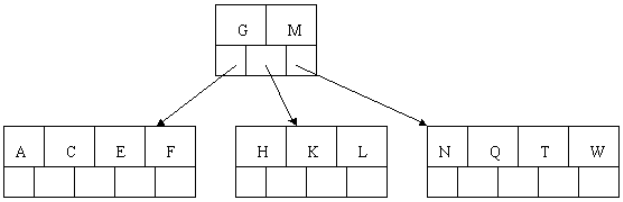
C N G A H E K Q M F W L T Z D P R X Y S



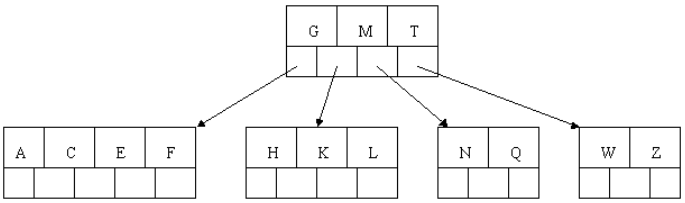
5. Las letras F, W, L, and T se añaden sin hacer divisiones.



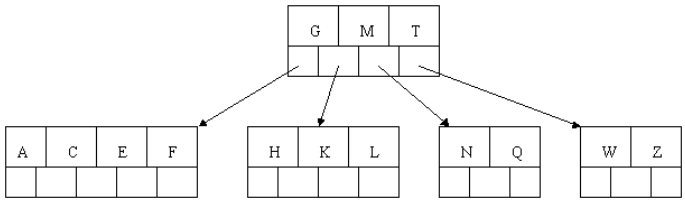
C N G A H E K Q M F W L T Z D P R X Y S



6. Al añadir la Z se produce una división.

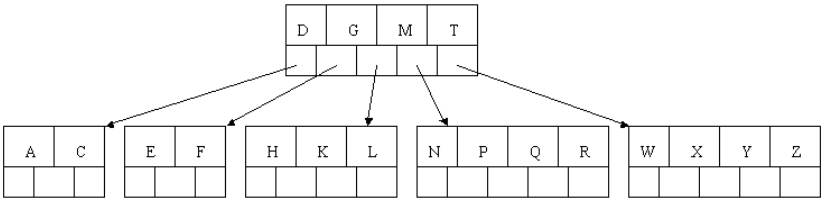


C N G A H E K Q M F W L T Z D P R X Y S

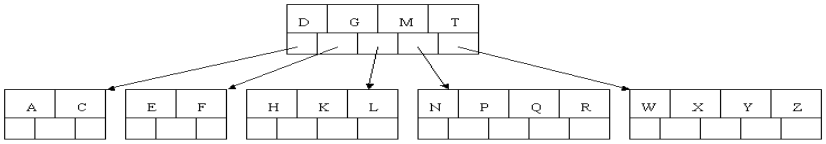


7. La inserción de D causa una división en la hoja de la izquierda.
D es la mediana y por tanto se sube al nodo padre.

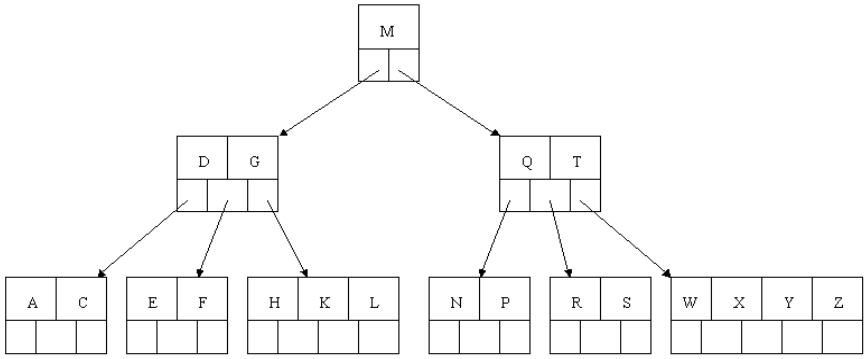
8. Las letras P, R, X,Y se añaden sin hacer divisiones.



C N G A H E K Q M F W L T Z D P R X Y S



9. Para añadir S, el nodo NPQR se divide enviando la mediana Q al padre. Sin embargo el padre está lleno, por tanto tiene que dividirse, enviado su mediana M hacia arriba creando un nuevo nodo raíz.



1.3. TIPOS DE DICCIONARIOS: TRIE

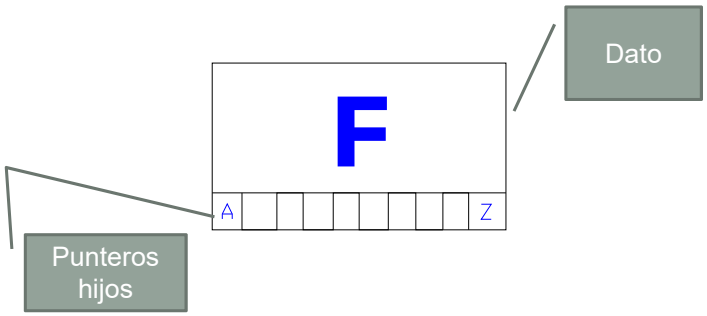
Trie

- Árbol para representar conjuntos de cadenas de caracteres u objetos (DICCIONARIO DE CADENAS):
- Cada nodo representa el prefijo de una palabra
- Los hijos de un nodo representan las cadenas que tiene a sus padres como prefijos
- Ventajas:
  - Búsquedas parciales (palabras que empiezan por “AR”)
  - No necesitan operación “redimensionar tabla”
  - Cada caracter se almacena una sola vez en los prefijos comunes
  - Complejidad en función de la longitud de la palabra y no en función del nº de palabras

Las desventajas es en cuanto al espacio de punteros y nodos “extra” que se verán en las siguientes implementaciones

Nodo:

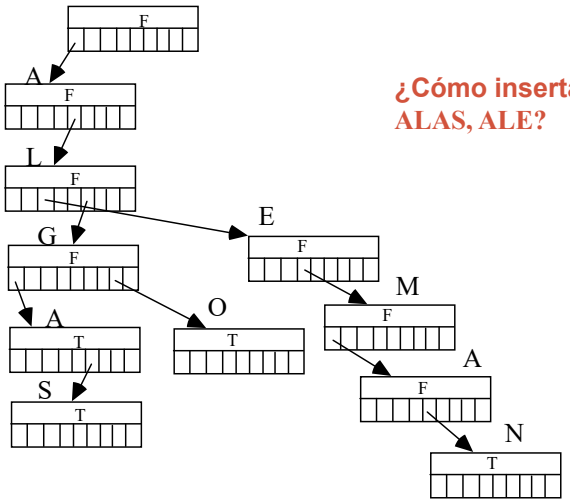
- **Dato**: campo booleano (**T** indica si es una palabra completa o **F** un prefijo)
- **Estructura de punteros** para todos los posibles hijos



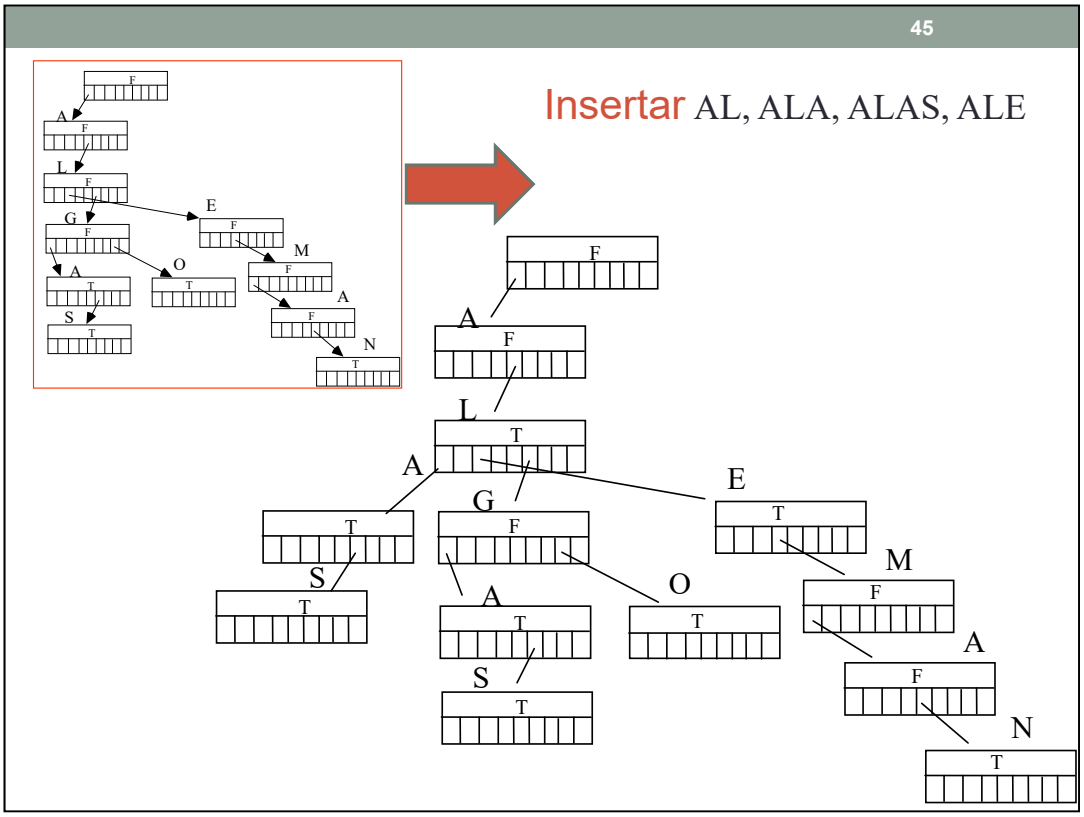
Hay que tener en cuenta que un camino en este árbol puede ser a la vez una palabra completa y un prefijo de palabras que cuelgan de ella. Por ejemplo, 'tela' y ´telar´.

Crear: Constructor que pondrá la variable booleana a FALSE y los punteros nulos

ALGA, ALGAS,
ALGO,
ALEMAN



¿Cómo insertar AL, ALA, ALAS, ALE?



Para insertar una palabra se va descendiendo por el árbol mientras se van consumiendo de izquierda a derecha los caracteres de la palabra. Si la llegar al final de la palabra se ha descendido por una camino existente, entonces se etiqueta el nodo al que se ha llegado con True, para indicar que ese camino representa una palabra. Si mientras se explora la palabra se llega a un nodo del que no se puede descender por el carácter actual que se está explorando en esa palabra, entonces se crea un nuevo nodo con puntero en ese carácter, y así sucesivamente hasta llegar al final de la palabra.

Pertenece la cadena *palabra* al diccionario?

```
ALGORITMO BÚSQUEDA_palabra_TRIE (palabra, T, Encontrado)
/*ENTRADA: palabra, T es el Trie
/*SALIDA: Encontrado: Boolean;
p ← T
Encontrado ← FALSE
mientras NOT EsVacio (p) AND NOT Encontrado
hacer   Si LONGITUD (palabra) = 0
        entonces   Encontrado ← Dato (p)
        sino       c ← OBTENER (palabra,1)
                   p ← p.HIJO (c)
                   Si NOT EsVacio (p)
                   entonces   palabra ← SUPRIMIR (palabra)
```

Donde:

Dato (p) : devuelve T (TRUE) o F (FALSE)
OBTENER (palabra,1) : devuelve primer carácter de ***palabra***
p.HIJO (c) : devuelve la posición del nodo hijo de **c**
SUPRIMIR (palabra): elimina el primer carácter de la *palabra*

2. BÚSQUEDA CON TOLERANCIA.

2.1 Índice Permuterm

2.2 Índice de k-gramas

Búsqueda con tolerancia

A menudo el usuario no está seguro de cómo se escribe una palabra o quiere buscar variantes de esa palabra.

Ejemplo1: se quiere buscar Sidney, pero no está seguro de si es ***Sidney*** o ***Sydney*** → buscar S*dney.

Ejemplo2: se quiere encontrar formas distintas de escribir la palabra ***color*** o ***colour*** → buscar col*r

Ejemplo3: se quiere buscar documentos que contienen ciertas variantes de un término, que podrían encontrarse si el sistema utilizara stemming, pero el usuario no lo sabe, p.ej. ***judicial, judiciary*** → buscar judicia*

*: representa cualquier cadena de caracteres

Wildcard query (consulta con comodín)

1) El * está al final de la palabra: mon*

Si el diccionario es un árbol es sencillo encontrar todos los términos W que tienen como prefijo *mon*. Luego se hacen |W| búsquedas de documentos que contienen los términos con ese prefijo.

2) Si * está al principio de la palabra: *mon

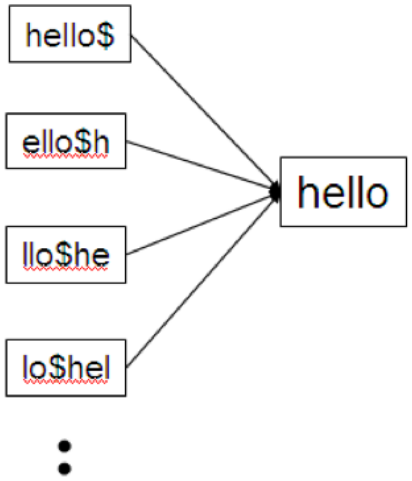
Se puede tener un B-árbol reverso que contiene el diccionario con las palabras escritas al revés.

3) Si el * está en medio de la palabra: se*mon

Podemos usar el árbol sencillo para encontrar las palabras cuyo prefijo es *se* y el B-árbol reverso para encontrar las palabras cuyo sufijo es *mon*. Luego se hace la intersección de los dos conjuntos obteniendo todas las palabras del diccionario que contienen el prefijo *se* y el sufijo *mon*.

2.1 Índice Permuterm

- Símbolo final del término: $
- Construye un Índice Permuterm con las diferentes rotaciones de cada término, todos enlazados al término original.
- Para la wildcard query h*o se rotaría apareciendo el * al final de la cadena buscando la cadena o$h* en el Índice Permuterm.
- A través de un árbol de búsqueda se dirigiría a términos como hero o hello, entre otros.

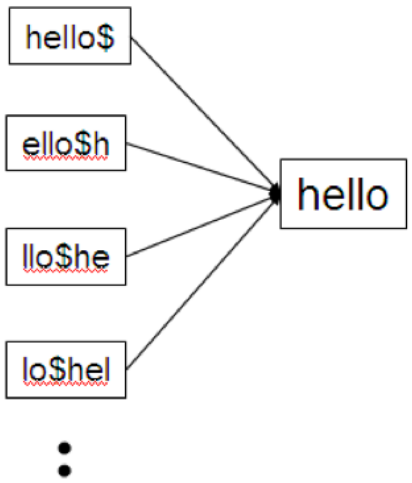


Problema: incrementa sensiblemente la talla del diccionario.

Es decir, por cada término se crean varios nuevos “términos”, que se incluyen en el diccionario.

Queries

- Para X → buscar X$
- Para X* → buscar $X*
- Para *X → buscar X$*
- Para *X* → buscar X*
- Para X*Y → buscar Y$X*



Query = hel*o (X=hel, Y=o) → buscar o$hel*

Ejemplos: Query = hello Hay que buscar hello$

Query = hello* Hay que buscar $hello*

Query = *hello Hay que buscar hello$*

Query = *hello* Hay que buscar hello*

Ejercicio: ¿Cómo se construiría el índice permuterm de la palabra “John”? Explica el mecanismo de búsqueda para la wilcard query “J*n”.

Ejercicio: ¿Cómo se construiría el índice permuterm de la palabra “John”? Explica el mecanismo de búsqueda para la wilcard query “J*n”.

El índice permuterm para el término John se construiría con las diferentes rotaciones del término:

John$

ohn$J

hn$Jo

n$Joh

$John

Y la búsqueda que se realiza es: n$J*

siguiendo la regla: Para buscar X*Y → buscar Y$X*

2.2 Índice de k-gramas

Enumerar todos los k-gramas (secuencias de k caracteres) que aparecen en cada término. Se añaden marcas de inicio y fin de palabra.

Ejemplo: "***April is the cruelest month***" se tienen los bigramas:

```
$a,ap,pr,ri,il,l$,$i,is,s$,$t,th,he,e$,$c,cr,ru,
ue,el,le,es,st,t$, $m,mo,on,nt,h$
```

El diccionario de un índice de k-gramas contiene todos los k-gramas que ocurren en los términos.

Se mantiene un segundo índice invertido desde los k-gramas a términos del diccionario que emparejan con ellos.

Ejemplo de posting list para el trigrama **etr**:

Procesar la wildcard query

Query ***re*ir*:** queremos buscar los documentos que contienen términos que empiecen por ***re*** y acaben en ***ir.***

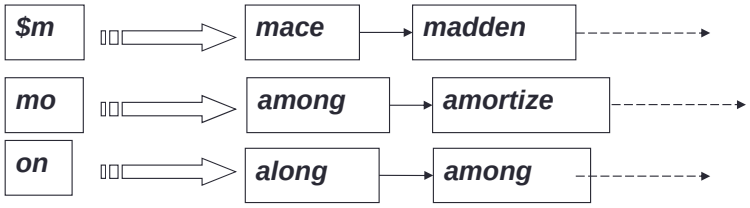
Ejecutar la consulta booleana: ***$re*** *AND* ***ir$***

1. En un Indice de 3-gramas se busca la lista de términos que emparejan, p.e. revivir y resistir.
2. En el Indice Invertido normal buscaremos los documentos que contienen estos términos.

Rápido, gestión del espacio eficiente (comparado con permuterm).

Procesar la wildcard query

- El índice *k*-gramas encuentra términos que contienen los *k*-gramas de la consulta (p.ej. *k*=2).



- Query ***mon**** se puede ejecutar como una expresión boleana
  - ***$m*** *AND* ***mo*** *AND* ***on***

- Podríamos haber obtenido ***moon***.
- Post-filtrar estos términos respecto a la query original.

Uno de los problemas de este método, tal como se indica en la trasparencia es que se pueden obtener términos que no correspondan con el query, es decir, se obtienen todos los correctos pero también se cuelan algunos incorrectos. Por ello es necesario un postproceso (que tiene un coste computacional) que compruebe uno a uno si el término obtenido es correcto.

3. CORRECCIÓN DE ERRORES

Corrección errores

Objetivo: corrección de errores en la consulta para obtener una respuesta correcta.

Ejemplo: el usuario teclea ***carot*** → corregir por ***carrot***

Aproximaciones:

1) Seleccionar la palabra más cercana: Distancia de edición.

2) Cuando hay más de una alternativa posible seleccionar la más común (mayor nº de ocurrencias en la colección o, en entornos web, la que mayor nº de usuarios han tecleado)

(pe. teclean carot que no se encuentra en el diccionario o ocurre en muy pocos documentos entonces se buscan correcciones posibles como carrot)

Ejemplo

El usuario teclea “graffe”

¿Cual es más cercana?

- graf
- graft
- grail
- giraffe

Necesitamos definir el concepto de “el más cercano”!!

Distancia de edición

La mínima distancia entre dos cadenas:

Es el número mínimo de operaciones de edición

- Inserción
- Borrado
- Substitución

necesario para transformar una cadena en otra.

Calcular la mínima distancia de edición:

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

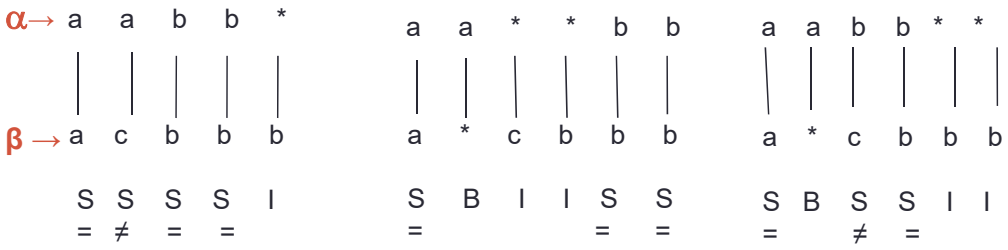
- La distancia será 5 si cada operación tiene coste 1.

Distancia: mínimo número de reglas de error para convertir la cadena **α** en **β**

$$D(\alpha, \beta) = \min_{\forall \Gamma}(N_s(\Gamma) + N_i(\Gamma) + N_B(\Gamma))$$

Como sólo se puede aplicar una regla de error a un carácter, una transformación corresponde con un "alineamiento" Γ entre las dos cadenas.

Ejemplo:



Se trata de realizar todos los alineamientos posibles entre las dos cadenas (siempre analizando las cadenas de izquierda a derecha, y no pueden cruzarse entre sí las líneas).

Hay tres tipos de alineamiento:

- Un carácter de $\alpha\lambda\phi\alpha$ con un carácter de beta: Corresponde con una Sustitución (que tiene coste 0 si son iguales y 1 si son diferentes)
- Un carácter de alfa con ningún carácter de beta (representado con un * en la transparencia): corresponde al Borrado, y tiene un coste de 1.
- Ningún carácter de alfa con un carácter de beta: corresponde con la Inserción, y tiene un coste de 1.

Una ilustración de este proceso de generar errores es la siguiente: imaginad que en vuestra mente está la idea de escribir una palabra, pero al ir tecleando carácter a carácter, os puede ocurrir que os equivoquéis de tecla, que no lleguéis a apretar bien la tecla y no se escriba, o que sin querer tecleéis una tecla cualquiera.

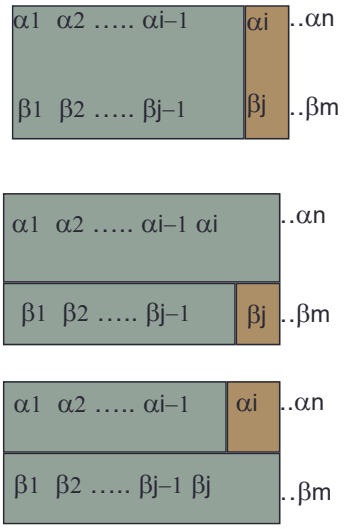
Fijaos que existen muchísimos alineamientos posibles, por ejemplo borrar todos los caracteres de alfa y luego insertar todos los de beta. O los tres ejemplos de la trasparencia.

El objetivo del algoritmo que se va a ver es: "de todas las formas que hay de realizar estos alineamientos, ¿cuál es la que menos errores contiene?", es decir buscamos el alineamiento más favorable, lo cual es una buena medida de lo parecidas que son entre

sí las dos palabras.

Solución recursiva:

¿Cuál es la mejor forma de haber alineado hasta el carácter *i* de la cadena α y el carácter *j* de la cadena β?.



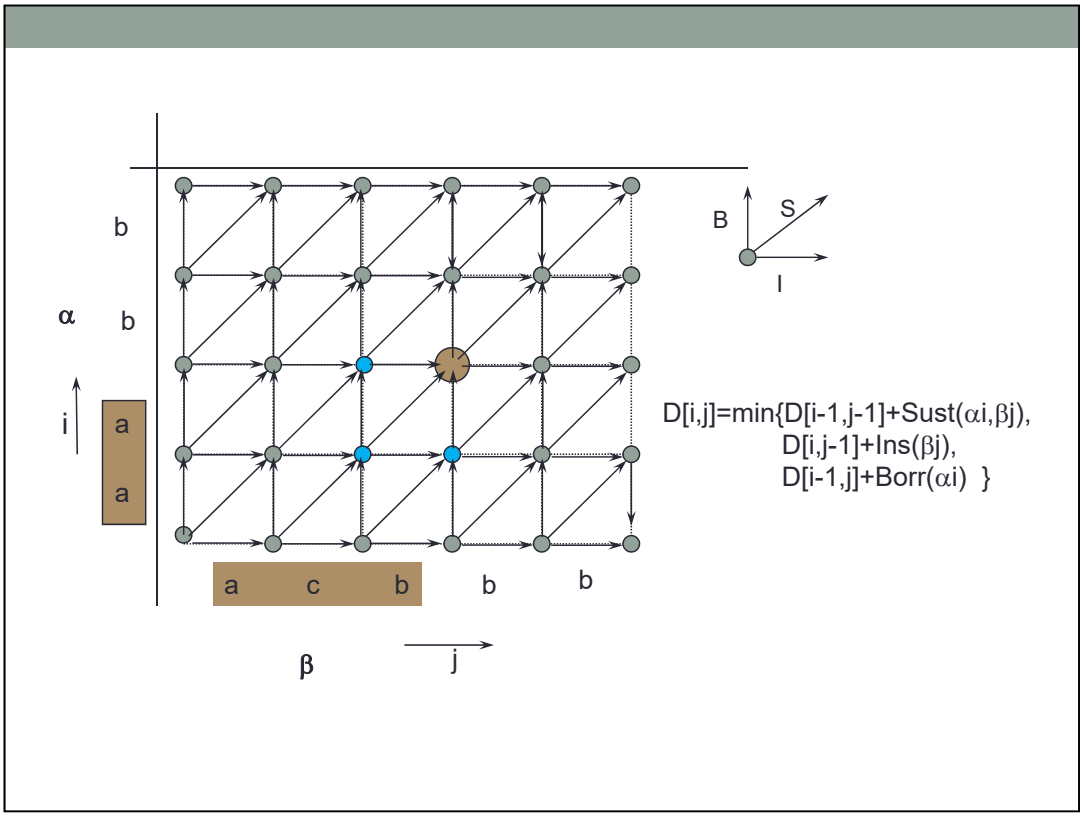
D(i-1,j-1)+ Sust (αi, βj)

D(i,j-1)+ Ins (βj)

D(i-1,j)+ Borr (αi)

Aquí se presenta la idea en que se basa el algoritmo: "¿Cuál es la mejor forma de haber llegado a analizar hasta el carácter i de la cadena alfa, y el carácter j de la cadena beta?". Sólo hay tres formas:

- Haber llegado al carácter i-1 de alfa, y al j-1 de beta, y entonces alinear el i de alfa con el j de beta (si son iguales el coste es cero y si son distintos es 1).
- Haber llegado al carácter i de alfa, y al j-1 de beta. Esto indica que el carácter j de beta es una Inserción
- Haber llegado al carácter i-1 de alfa, y al j de beta. Esto indica que el carácter i de alfa ha sido borrado.



Para resolver el problema se construye una matriz, en la que cada celda (i,j) almacenará la mejor forma de haber analizado hasta el carácter i de alfa y el carácter j de beta. Siguiendo el razonamiento de la trasparencia anterior, el valor de una celda (i,j) se basa en las tres celdas contiguas (i-1,j-1) (i-1,j) (i,j-1). De modo que, tal como expresa la formula:" El coste de la mejor forma de alinear hasta el carácter i de alfa y el carácter j de beta, es lo mejor de":

- el coste (ya calculado anteriormente y almacenado en la tabla) de haber alineado hasta el carácter i-1 de alfa y el carácter j-1 de beta más 1 o 0 (dependiendo si alfa(i) es diferente o no de beta(j))
- el coste (ya calculado anteriormente y almacenado en la tabla) de haber alineado hasta el carácter i de alfa y el carácter j-1de beta más 1 (Inserción)
- el coste (ya calculado anteriormente y almacenado en la tabla) de haber alineado hasta el carácter i-1 de alfa y el carácter j de beta más 1 (Borrado)

En el gráfico se ve que cada camino corresponde a uno de los posibles alineamientos (por ejemplo el camino que sigue la primera línea vertical y luego la horizontal superior corresponde a realizar primero todo Borrados y luego todo Inserciones, siendo su coste 9).

Levensthein prueba todos los caminos posibles y se queda con el de mínimo valor

Distancia de edición mínima (Levenshtein)

- Inicialización

  D(i,0) = i

  D(0,j) = j

- Cálculo de la matriz

  desde j←1 **hasta** |β| **hacer**

  desde i← 1 **hasta** |α| **hacer**

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 & /*\text{Borrado}*/ \\ D(i,j-1) + 1 & /*\text{Inserción}*/ \\ D(i-1,j-1) + \begin{cases} 1; \text{if } \alpha(i) \neq \beta(j) \\ 0; \text{if } \alpha(i) = \beta(j) \end{cases} & /*\text{Sustitución}*/ \end{cases}$$

- Terminación

  D(|α| , |β|) es la distancia

Levenshtein prueba todos los caminos posibles y se queda con el de mínimo valor.

Este algoritmo va rellenando la matriz representada en la figura anterior. Hay que fijarse en que es necesario calcularla en un orden. Antes de calcular un valor (i,j) deben estar calculados los valores (i-1,j-1) (i-1,j) (i,j-1). En este algoritmo el orden escogido consiste en calcular primero la primera fila y la primera columna, a la que solo se accede por la celda anterior (Inserción o Borrado). Luego se va de columna en columna, y dentro de una columna en orden ascendente de filas. La solución está en la celda superior derecha.

Tabla de distancia de edición

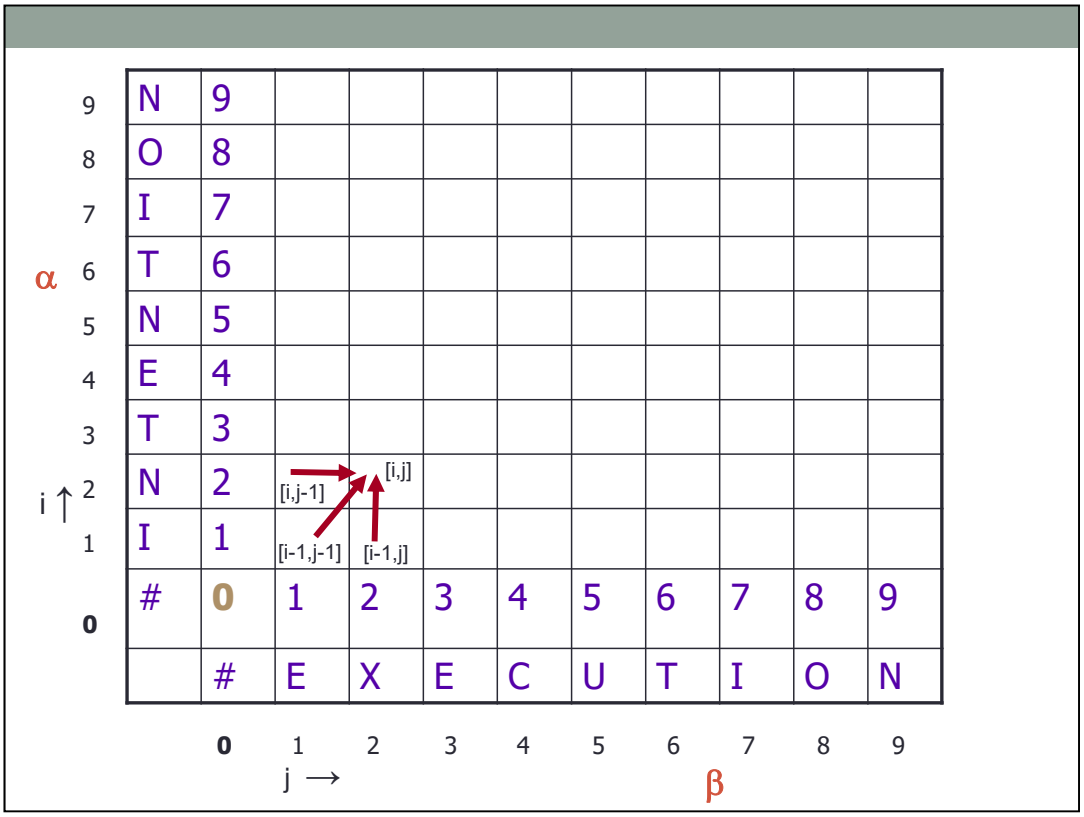
```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

α

i ↑		j →									
	N	9									
	O	8									
	I	7									
	T	6									
	N	5									
	E	4									
	T	3									
	N	2									
	I	1									
	#	0	1	2	3	4	5	6	7	8	9
		#	E	X	E	C	U	T	I	O	N

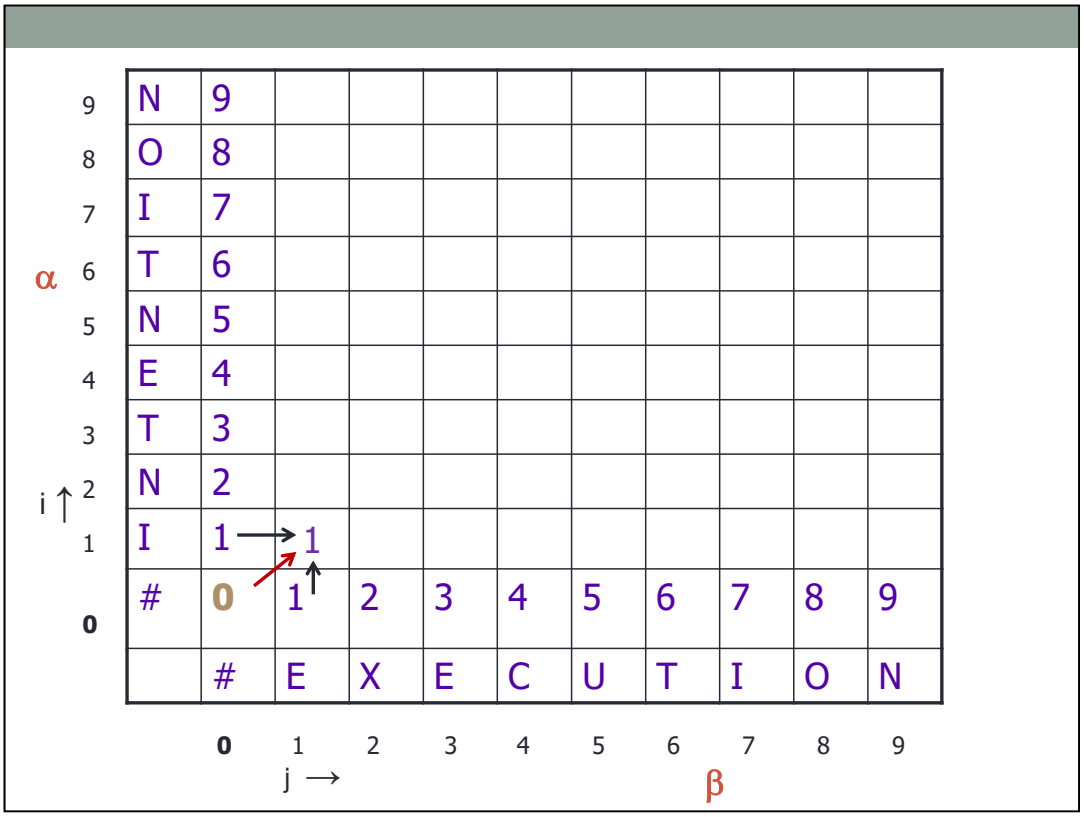
β

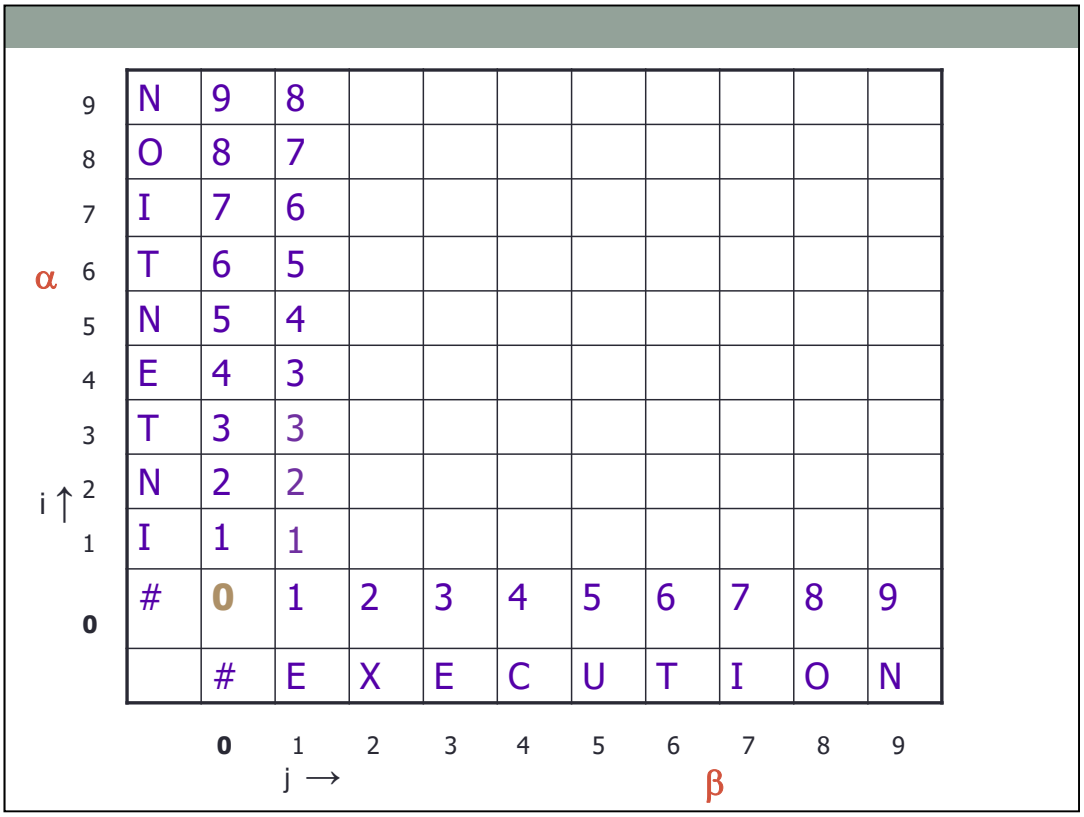
Esta tabla corresponde a la inicialización del algoritmo D(i,0)=i ; D(0,j)=j;



9	N	9								
8	O	8								
7	I	7								
6	T	6								
5	N	5								
4	E	4								
3	T	3								
2	N	2								
1	I	1								
0	#	**0**	1	2	3	4	5	6	7	8	9
		#	E	X	E	C	U	T	I	O	N
		0	1	2	3	4	5	6	7	8	9

Aquí se indica cómo se calcula una celda cualquiera en función de los valores almacenados en las celdas anteriores (y el coste de la transición utilizada)


N 9
O 8
I 7
T 6
N 5
E 4
T 3
N 2
I 1 1
0 1 2 3 4 5 6 7 8 9
E X E C U T I O N
α
β
i
j
0 1 2 3 4 5 6 7 8 9


α
i
j →
β
N 9 8
O 8 7
I 7 6
T 6 5
N 5 4
E 4 3
T 3 3
N 2 2
I 1 1
0 1 2 3 4 5 6 7 8 9
E X E C U T I O N
0 1 2 3 4 5 6 7 8 9

9	N	9	8	8	8	8	8	8	7	6	5
8	O	8	7	7	7	7	7	7	6	5	6
7	I	7	6	6	6	6	6	6	5	6	7
α 6	T	6	5	5	5	5	5	5	6	7	8
5	N	5	4	4	4	4	5	6	7	7	7
4	E	4	3	4	3	4	5	6	6	7	8
3	T	3	3	3	3	4	5	5	6	7	8
i ↑ 2	N	2	2	2	3	4	5	6	7	7	7
1	I	1	1	2	3	4	5	6	6	7	8
0	#	0	1	2	3	4	5	6	7	8	9
		#	E	X	E	C	U	T	I	O	N
		0	1	2	3	4	5	6	7	8	9
			j →						β		

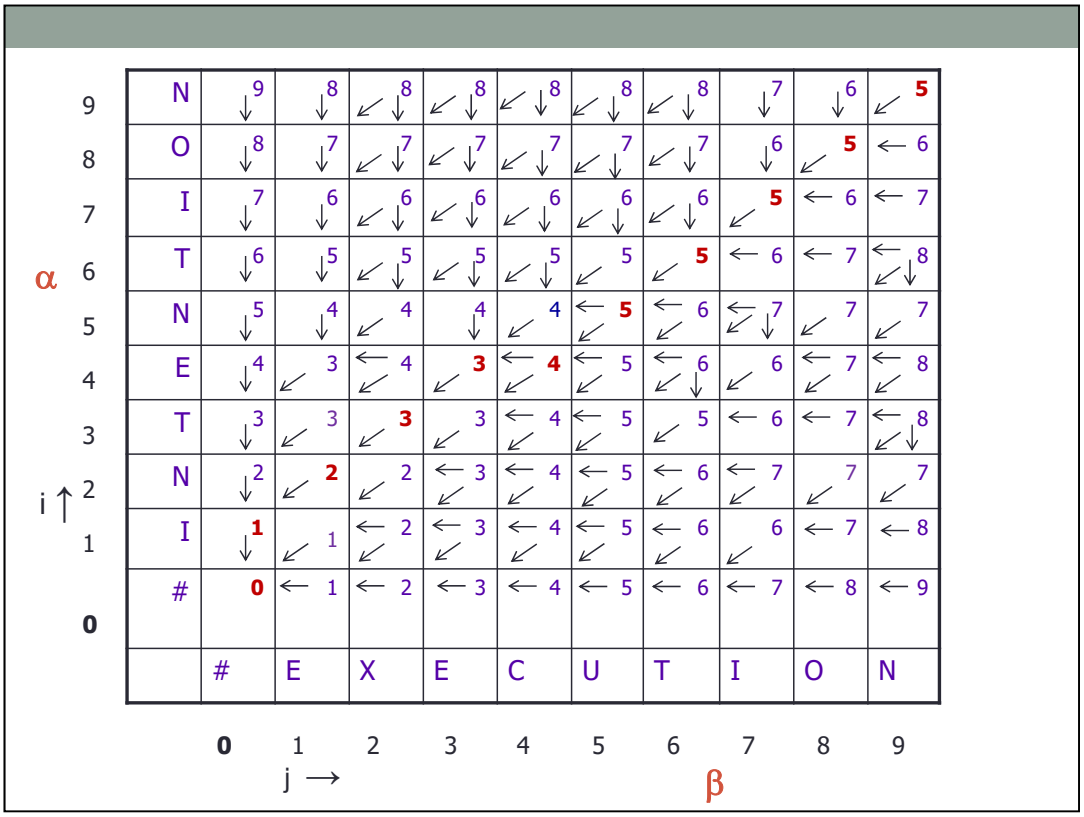
9	N	9	8	8	8	8	8	8	7	6	**5**
8	O	8	7	7	7	7	7	7	6	**5**	6
7	I	7	6	6	6	6	6	6	**5**	6	7
α 6	T	6	5	5	5	5	5	**5**	6	7	8
5	N	5	4	4	4	4	**5**	6	7	7	7
4	E	4	3	4	**3**	**4**	5	6	6	7	8
3	T	3	3	**3**	3	4	5	5	6	7	8
i ↑ 2	N	2	**2**	2	3	4	5	6	7	7	7
1	I	**1**	1	2	3	4	5	6	6	7	8
0	#	**0**	1	2	3	4	5	6	7	8	9
		#	E	X	E	C	U	T	I	O	N
		0	1	2	3	4	5	6	7	8	9

j → β

En rojo está marcado un camino ganador, que solamente puede conocerse al llegar al final del algoritmo.

Cálculo de alineamientos

- La distancia de edición no es suficiente
  - Si necesitamos alinear cada carácter de las dos cadenas
- Podemos hacerlo manteniendo un “backtrace”
- Cada vez que entramos en una celda, recordar desde dónde venimos
- Cuando se alcanza el final,
  - Rastrear el camino desde la esquina superior derecha para leer la alineación



Esta es una matriz “paralela” a la anterior de costes, en la que se almacenan los índices (o un código como S,I,B) de la celda anterior que ha ganado la minimización. En la figura se representa codificado con flechas, y además en caso de empate se han representado las dos transiciones que empatan.

Añadir “Backtrace” a la distancia de edición mínima

- Inicio:

  $D(i,0) = i \quad D(0,j) = j$

Terminación:

$D(|\alpha|, |\beta|)$ distancia

- Cálculo de las matrices:

  desde j←1 **hasta** $|\beta|$ **hacer**

  desde i←1 **hasta** $|\alpha|$ **hacer**

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \text{deletion} \\ D(i,j-1) + 1 & \text{insertion} \\ D(i-1,j-1) + \begin{cases} 1; \text{ if } \alpha(i) \neq \beta(j) \\ 0; \text{ if } \alpha(i) = \beta(j) \end{cases} & \text{substitution} \end{cases}$$

$$ptr(i,j) = \begin{cases} LEFT & \text{insertion} \\ DOWN & \text{deletion} \\ DIAG & \text{substitution} \end{cases}$$

Resultado del “Backtrace”

- Dos cadenas y su alineamiento:

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
```

Rendimiento

- Tiempo: O ($|\alpha| \times |\beta|$)
- Espacio: O ($|\alpha| \times |\beta|$)
- Backtrace: O ($|\alpha| + |\beta|$)

Los costes del algoritmo son muy buenos, ya que son polinómicos.