

## Programa tu propio Google

### El buscador

*Ahora que ya tenemos el indexador funcionando, podemos apoyarnos en esos flamantes índices recién creados para acelerar nuestras búsquedas. Tranquilos que esta es la parte más fácil de las bibliotecas digitales. Y es que precisamente eso es lo que pretendíamos: convertir la búsqueda en una tarea sencilla y rápida, a costa de preparar las estructuras necesarias con el proceso de indexación.*



- [Teoría sobre el buscador](#)
  - [Exclusión de términos](#)
  - [Búsqueda de conceptos](#)
  - [Búsquedas basadas en diccionarios de sinónimos](#)
  - [Búsquedas de fechas](#)
  - [Algoritmo de búsqueda](#)
  - [El orden de los resultados](#)
- [Implementando el buscador](#)
  - [Nuestro Toopo funcionando](#)
- [Conclusiones](#)
- [Los ejemplos](#)

### Teoría sobre el buscador

Las búsquedas que podemos realizar a una biblioteca digital pueden ser de dos tipos:

- De una sola palabra: en este caso sólo tenemos que localizar en el índice invertido la entrada correspondiente a esa palabra, y nuestros resultados serán los documentos asociados a esa entrada. Este algoritmo podemos verlo en el Listado 1. Utilizando el ejemplo de las frases que ya vimos en el [anterior artículo](#), y el índice invertido generado a partir de ellas, podemos buscar la palabra “cuidado” y veremos que aparece en los documentos 2 y 3.

```
Listado 1
.....

FOR "todas las palabras del indice"
  IF "palabra actual = palabra buscada" THEN
    "meter documentos asociados en la lista de encontrados"
    RETORNAR.
  END-IF
END-FOR
```

- De varias palabras: lo más habitual es que el usuario del buscador introduzca varios términos de búsqueda. En este caso tenemos que saber con qué operación relacionamos ambos términos. ¿Queremos buscar los documentos que contengan

*todas* las palabras? ¿O acaso queremos buscar los documentos que incluyan *alguna* de las palabras?

Normalmente, los buscadores dan el primer comportamiento por defecto, permitiendo modificarlo con algunas palabras especiales. Un ejemplo con las frases de nuestro ejemplo: si buscamos “coche rápido”, podemos considerar válidos dos conjuntos de resultados:

- Documento 1: esto será correcto si estamos buscando los documentos que contengan *todas las palabras buscadas*. Esta es la búsqueda por defecto que hacen la mayoría de los buscadores, como Google, y es la que da resultados más satisfactorios. El algoritmo para esta búsqueda lo encontraréis en el siguiente pseudocódigo:

#### Listado 2

```
.....

FOR "todas las palabras del indice"
  IF "palabra actual = primera palabra buscada" AND
    "palabra actual = segunda palabra buscada"
  THEN
    meter documentos asociados en la lista de encontrados
  END-IF
END-FOR

X = "numero de términos de búsqueda"

FOR "todos los documentos encontrados"
  IF "doc actual aparece menos de X veces en lista encontrados" THEN
    "eliminar documento actual de la lista de encontrados"
  END-IF
END-FOR
```

- Documentos 1 y 3: sí, por el contrario, estamos buscando los documentos que contengan *al menos una de las palabras*. En Google podemos realizar estas consultas intercalando el operador “OR” entre los distintos términos. Podéis ver el algoritmo para este caso en el siguiente listado:

#### Listado 3

```
.....

FOR "todas las palabras del indice"
  IF "palabra actual = primera palabra buscada" OR
    "palabra actual = segunda palabra buscada"
  THEN
    meter documentos asociados en la lista de encontrados
  END-IF
END-FOR

"eliminar repetidos de la lista de encontrados"
```

### Exclusión de términos

Además del modificador “OR”, es posible que se permitan otros modificadores, como por ejemplo el guión “-”, que Google utiliza para indicar que estamos buscando los documentos que no contengan cierta palabra. Por ejemplo, si hacemos la siguiente búsqueda en Google “cervantes –quijote”, nos retornará todas las páginas que incluyan el término “cervantes”, pero que no incluya “quijote”.

La implementación de esta característica no es difícil, pero sí un poco liosa. Debemos recorrer el índice completo buscando los términos que deben aparecer (los que no tienen el guión), tal y como vimos en el anterior Listado 2. Una vez que tenemos una lista de documentos encontrados, recorreremos de nuevo el índice invertido buscando los términos que no deben aparecer (los que tienen guión). Cada vez que encontremos uno de estos términos, buscaremos cada uno de los documentos de los resultados en los documentos asociados al término, y si el documento aparece, lo eliminaremos de los

resultados. Podéis verlo en forma de pseudocódigo en el siguiente listado Listado 4:

**Listado 4**

```
.....

FOR "todas las palabras del indice"
  IF "palabra actual = palabra buscada" THEN
    meter documentos asociados en la lista de encontrados
  END-IF
END-FOR

FOR "todas las palabras del indice"
  IF "palabra actual = palabra a excluir" THEN
    FOR "todos documentos encontrados"
      IF "aparece documento actual en documentos asociados" THEN
        "eliminar documento actual de documentos encontrados"
      END-IF
    END-FOR
  END-IF
END-FOR
```

### *Búsqueda de conceptos*

Todos sabemos que el castellano es un idioma muy rico tanto en vocabulario como en conjugación, pero esto, para la vida de los programadores de bibliotecas digitales, es un problema.

Por ejemplo, si tenemos una biblioteca digital legislativa, y queremos buscar las leyes que hablen sobre "multas", deberíamos buscar los documentos que contengan los términos "multa", "multas", "multado", "multados", "multada", "multadas", "multando", además de todas las conjugaciones verbales en cualquier número y tiempo: "multo", "multas", "multa", "multamos", "multáis", "multan", "multaba", "multé", "multó" y así hasta que os aburráis. Como veis, de un concepto muy concreto (esas dichosas multas), surgen muchísimos términos, que aunque se refieren a la misma idea, no son exactamente iguales. Así que aunque busquemos el término "multa", no tendremos la seguridad de haber encontrado todos los documentos que hagan referencia al concepto "multa" (ojo a la diferencia entre *término* y concepto).

Si hago memoria de mis tiempos de escolar, creo que había una parte de cada palabra que era la raíz, llamada "lexema", que permanecía invariable en todas las posibles formas. Si no me equivoco, el lexema de "multa" debe ser "mult", ya que es la parte que no varía de todas las formas que hemos visto antes.

Bien, aplicando esto, podemos indexar los textos con una pequeña variación: por cada palabra que nos encontremos, la indexaremos no como aparece en el texto, sino con su lexema. Es decir, si tenemos los siguientes textos:

1. "El policía nos multó a todos."
2. "Vaya cara que pusimos todos los multados"
3. "La multa fue injusta"
4. "¡Abajo las multas!"

Podemos indexarlos de forma normal, y en el índice invertido habrá entrada distintas para los términos "multó", "multados", "multa" y "multas", por lo que si buscamos el concepto "multa", solo encontraremos el documento 3. Pero también podemos indexarlos utilizando su lexema:

1. "El policía nos mult a todos."
2. "Vaya cara que pusimos todos los mult"
3. "La mult fue injusta"
4. "¡Abajo las mult!"

De este modo, el índice invertido solo tendrá una única entrada para el lexema, y no una entrada por cada variación. Cuando hagamos una consulta, sustituiremos cada término a buscar por su lexema y después haremos la búsqueda como hasta ahora. Por ejemplo, si buscamos el término “multado”, lo sustituiremos por “mult”, y si buscamos “multa”, también lo sustituiremos por su lexema. De este modo somos capaces de buscar “conceptos” y no solo “términos”.

Esta técnica, aunque muy interesante y útil, es difícil de implementar, ya que necesitamos un método para detectar los lexemas de cualquier palabra en castellano, y eso, con un algoritmo, es difícil de conseguir (pero no imposible).

Una solución más sencilla, o complementaria, es la búsqueda basada en diccionarios de sinónimos.

### *Búsquedas basadas en diccionarios de sinónimos*

Como las búsquedas de lexemas son complejas de implementar, existe otro método de búsqueda más sencillo, que aunque no nos ofrece la misma potencia, nos puede servir de complemento.

Básicamente consiste en lo mismo: a la hora de indexar los documentos, sustituiremos una serie de términos por uno solo, basándonos en un diccionario de sinónimos externo.

Por ejemplo, supongamos los siguientes textos:

1. “El coche se paró”
2. “Me mola el tuneado de tu buga”
3. “Ese automóvil es fantástico”
4. “Kit, ¡te necesito!”

Podemos aplicar el siguiente diccionario de sinónimos

- buga = coche
- automóvil = coche
- kit = coche

Así, a la hora de indexar, sustituiremos los sinónimos, dejando los siguientes documentos:

1. “El coche se paró”
2. “Me mola el tuneado de tu coche”
3. “Ese coche es fantástico”
4. “coche, ¡te necesito!”

Cuando vayamos a buscar, realizaremos el mismo proceso: sustituir los términos de búsqueda por sus sinónimos. La consulta “automóvil viejo” pasará a ser “coche viejo”. Así, utilizando cualquier término, llegaremos al mismo concepto.

Este método tiene una gran ventaja: es muy sencillo de implementar, pero un gran inconveniente: es difícil crear diccionarios de sinónimos completos, ya que nuestro idioma es muy rico en palabras.

El sistema de búsqueda perfecto sería aquel que combinase las búsquedas con lexemas (que cuando busques “multa” también se encuentre “multado”) y con diccionario, para los sinónimos (que cuando busques “coche” también se encuentre “automóvil”).

En nuestra implementación hemos utilizado un pequeño diccionario de sinónimos, gestionado a través de la clase TDiccionarioSinonimos.

### *Búsquedas de fechas*

Y otro problema habitual a la hora de buscar son las fechas. Nunca sabemos si la fecha está en formato día/mes/año, día-mes-año, con el año con 4 dígitos, con el nombre del mes en vez de su número...

Para solucionar esto podemos dar otra vuelta de tuerca a la misma técnica: normalizar las fechas tanto en indexación como en la consulta. Así, mientras estamos indexando, cada vez que encontremos una fecha en cualquier formato, se almacena en el índice siempre en el mismo formato. A la hora de buscar haremos lo mismo, cambiando las fechas todas al mismo formato en que están indexadas.

### Algoritmo de búsqueda

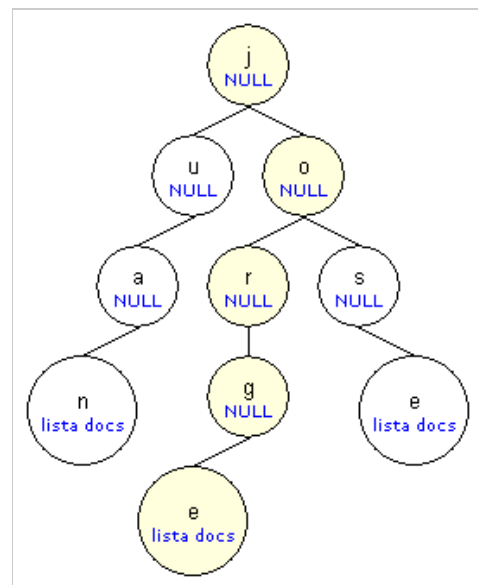
Ya sabemos que el índice invertido, al fin y al cabo, es una lista de palabras. Una de las decisiones más importantes que tenemos que tomar, es el método que utilizaremos para localizar una palabra en esa lista, ya que el rendimiento de nuestro buscador dependerá, en gran medida, de la forma en que hagamos esta búsqueda. Hay que tener en cuenta que el índice puede ser muy grande, por ejemplo: en el código fuente que acompaña al artículo, encontraréis una pequeña biblioteca digital. En ella se han indexado solo cuatro documentos, cuentos infantiles (y no tan infantiles) de toda la vida: Caperucita Roja, Pulgarcito, La cenicienta y El Principito. Como veis, esto es un ejemplo minúsculo de biblioteca digital, ya que ni son textos grandes, ni gran cantidad de documentos. Pues el índice invertido resultante contiene más de 3.500 palabras. Así que, para buscar una palabra en esta lista, puede que el rendimiento dependa mucho del algoritmo de búsqueda que utilicemos.

En nuestra implementación hemos utilizado una búsqueda secuencial, la más sencilla de programar pero la más ineficiente y lenta. Esto hace que el rendimiento de nuestras búsquedas no sea todo lo bueno que podría llegar a ser, y que los tiempos de búsqueda crezcan conforme el tamaño de la biblioteca vaya aumentando. Incluso con esta limitación, las búsquedas en la biblioteca de ejemplo son casi instantáneas, aunque si indexamos más documentos y más grandes (estoy hablando de “megas” o incluso “gigas” de información), el rendimiento dejará mucho que desear.

Otras posibilidades a la hora de localizar una palabra son utilizar algoritmos de búsqueda ordenada, binaria, basada en tablas *hash*, etc. Y por último, una implementación típica, y posiblemente la más óptima, suele ser un árbol *trie*, donde se va formando un árbol con las letras de cada palabra (Ver el siguiente cuadro)

Una de las mejores formas de resolver la búsqueda en un índice invertido es a través de un árbol *trie*. Se trata de una estructura de datos especial para búsquedas de cadenas de texto. Las búsquedas secuenciales o binarias tienen un gran problema: su tiempo de búsqueda dependen del tamaño del índice donde estemos buscando. Es decir: cuando más grande sea la lista de palabras (índice) de búsqueda, más tardaremos en encontrar nuestra palabra. Con esto conseguimos que los tiempos de búsqueda en una estructura *trie* dependan de la longitud de la palabra a buscar, y no de la cantidad de palabras contenidas en el índice.

Básicamente se implementa a través de un árbol n-ario, donde cada nodo intermedio contiene la letra de la palabra a buscar, y el





En nuestro buscador TooPo hemos obviado este tema para simplificarlo al máximo. De todas formas, para los valientes diremos que hay dos opciones:



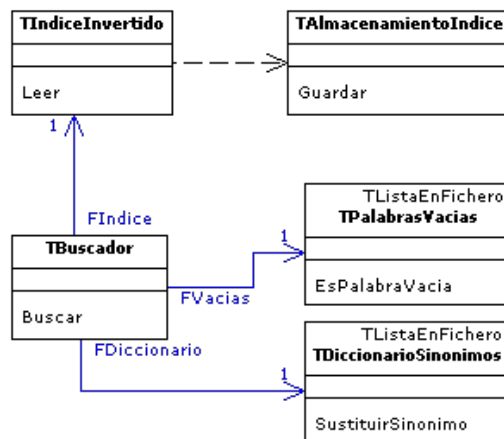
- Almacenar los documentos del índice invertido ya ordenados: con esto conseguimos rapidez en la búsqueda, ya que según recuperemos los documentos encontrados, podemos mostrarlos. Sin embargo este método es más rígido, ya que el orden que establezcamos en indexación será fijo para cualquier búsqueda que hagamos.
- Ordenar la lista de resultados en la búsqueda: una vez que ya tenemos la lista de resultados, podemos ordenarla dinámicamente dependiendo de distintos factores. Este método es más lento (tiene que ordenar una lista que puede ser muy grande) pero más flexible.

Por cierto, los responsables de Google-Desktop ha declarado que han simplificado enormemente el cálculo del *PageRank* en su herramienta de escritorio: los resultados obtenidos se ordenan “por fecha”, pudiendo cambiarlo a “por relevancia”.

## Implementando el buscador

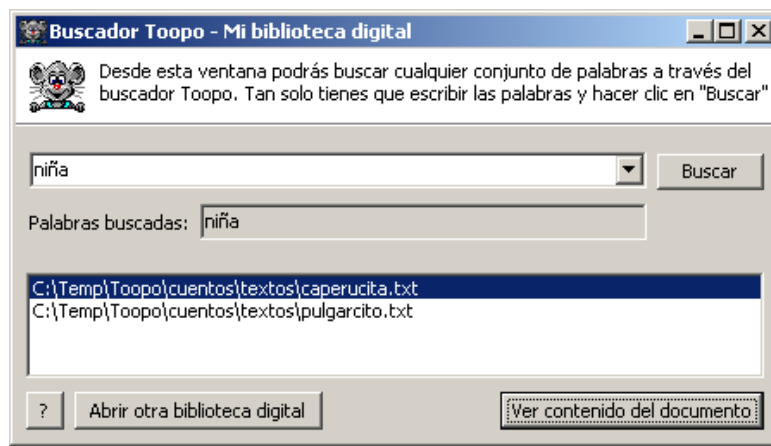
Por último, nos queda ver cómo hemos hecho la implementación de nuestro buscador. Aunque parezca increíble, hemos conseguido reutilizar algunas clases que ya habíamos creado para el indexador, así que la mitad del trabajo lo tenemos hecho. Concretamente, toda la lógica de almacenamiento está ya programada, a través de la clase *TAlmacenamientoIndice*, así como la representación en memoria del índice invertido, con la clase *TIndiceInvertido*. Tan solo nos falta añadir un método de búsqueda al índice (para que nos retorne la lista de documentos para una palabra dada), y crear una clase *TBuscador* que actúe como fachada (será la clase que utilizaremos desde el exterior).

En la siguiente imagen podéis ver un diagrama con el diseño de clases del buscador:



## Nuestro Toopo funcionando

Después de toda esta aventura, tenemos un flamante buscador al estilo Google-Desktop. En la siguiente imagen podéis verlo funcionando, tan mono él.



Podéis probar la indexación con vuestros propios textos, ya sean páginas web o archivos de texto plano.

Las búsquedas también podéis probarlas con vuestras propias colecciones indexadas, o bien utilizar alguna de las de ejemplo que encontraréis en el código fuente que acompaña al artículo (una con cuentos infantiles en texto plano, y otra con unas páginas web con la historia de la máquina Enigma).

## Conclusiones


Como ya hemos dicho, hemos dejado bastantes características en el tintero, como las búsquedas con el operador “OR”, el caché de los documentos indexados, la búsqueda en el índice utilizando un árbol *trie*, o la ordenación de los resultados por relevancia, aunque yo creo que como buscador funcional puede servir.

A partir de esta implementación podemos seguir trabajando, para hacer una indexación incremental de los documentos, indexar el disco duro completo, detectar cambios en los documentos y reindexarlos, almacenar los documentos indexados en un caché, comprimir los índices o los documentos en el caché, implementar el operador “OR”, almacenar las posiciones de los términos encontrados (para poder resaltar la palabra), implementar el árbol *trie* para mejorar el rendimiento (a la vez que hacemos un sistema más escalable), ordenar la lista de resultados, añadir nuevos formatos soportados (documentos PDF, Postscript, LaTeX, planos de Autocad), y un larguísimo etcétera. ¿Algún valiente que se atreva a ampliar el sistema de Toopo? Se me ocurren cientos de aplicaciones realmente útiles que se podrían construir con el motor de Toopo, pero eso ya lo dejo en vuestras manos. Si alguien está interesado en retomar el trabajo, que me avise y nos ponemos manos a la obra.

## Los ejemplos

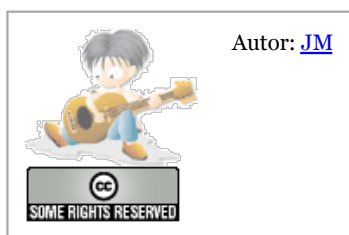
### Delphi 6

Nuestro buscador Toopo completo: las clases comunes de la arquitectura general (usadas tanto por el indexador y el buscador), y tanto el indexador como el buscador funcionando.

 [Fuentes en ZIP](#)

 [El ejecutable ya compilado](#)





Autor: [JM](#)



Este artículo apareció publicado por primera vez en el número 9 de la revista Todo Programación, editada por [Studio Press, S.L.](#) y se reproduce aquí con la debida autorización.



[Página principal](#)



[Programación](#)



[Delphi](#)



[Volver](#)



2005 by [JM](#)

