Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

PROYECTO SEMESTRAL DE LABORATORIO

Versión Preliminar al 01-08-2017

Durante el semestre se trabajará en distintos paradigmas de programación. Estos serán abordados a través de implementaciones de software en cuatro lenguajes de programación en cuatro entregas de laboratorio. La temática adoptada en el presente semestre es la implementación de un sistema de recuperación de información. Los aspectos funcionales y no funcionales a considerar en cada uno de los proyectos, tomarán como referencia sistemas de búsqueda de gran escala como son Google y Bing, sin embargo, las especificaciones de ambos tipos de aspectos serán simplificaciones de cómo operan este tipo de sistemas en la realidad.

Descripción general

Para este semestre se ha definido la realización de un sistema de recuperación de información. Al final del semestre los alumnos de Paradigmas de Programación habrán implementado distintas versiones del buscador en distintos lenguajes de programación, según los paradigmas que los rigen. A continuación se listan de forma general las características que deberá tener el sistema, las cuales serán abordadas de manera obligatoria o complementaria durante el semestre en los distintos proyectos. Los detalles para cada laboratorio se listan posteriormente.

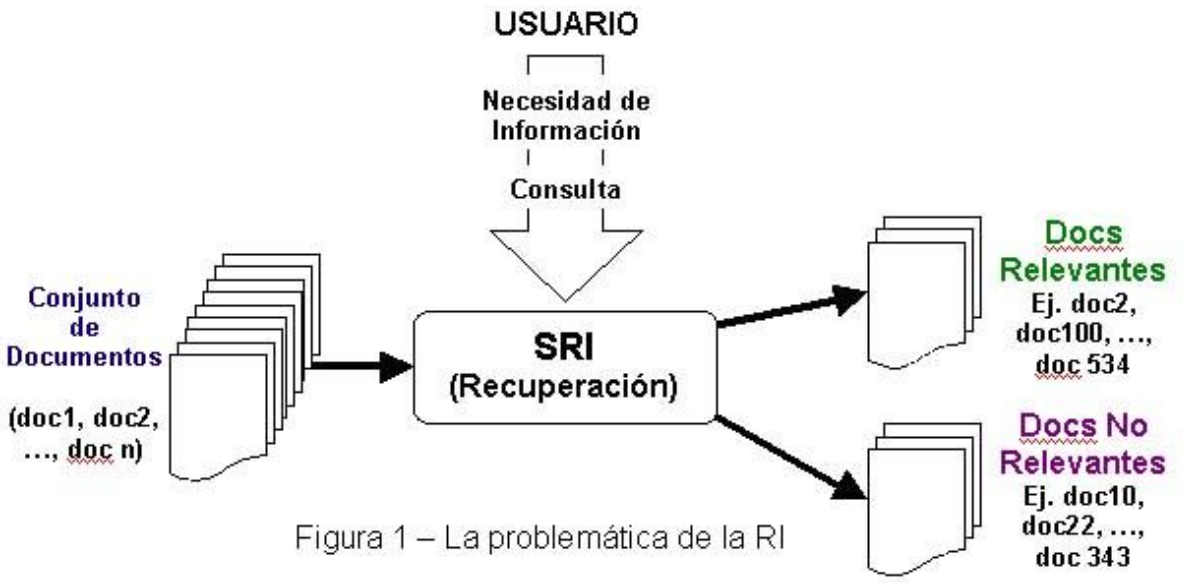

Figura 1 – La problemática de la RI

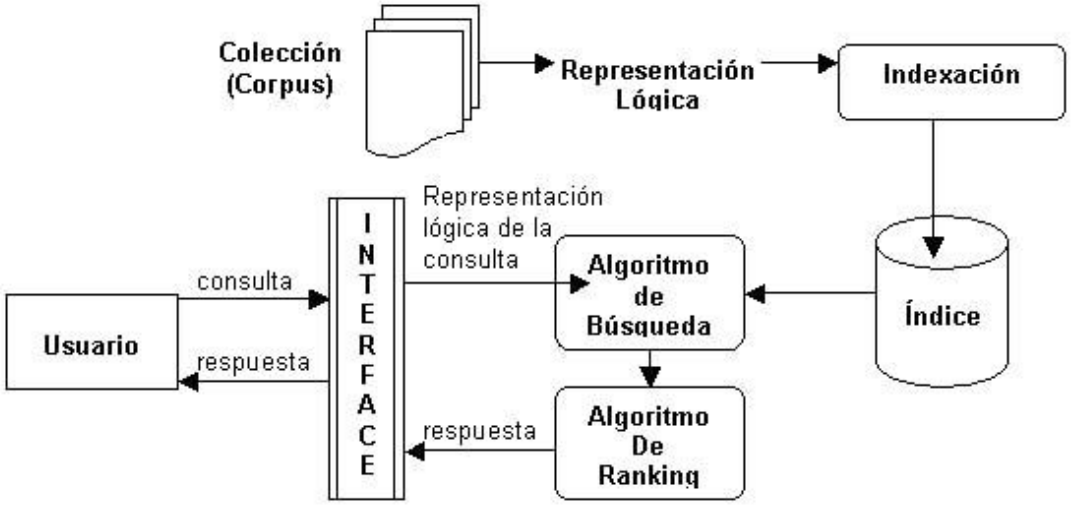

Figura 2 – Arquitectura básica de un SRI

Fuente: http://ferbor.blogspot.cl/2006/08/la-problemtica-de-la-recuperacin-de.html

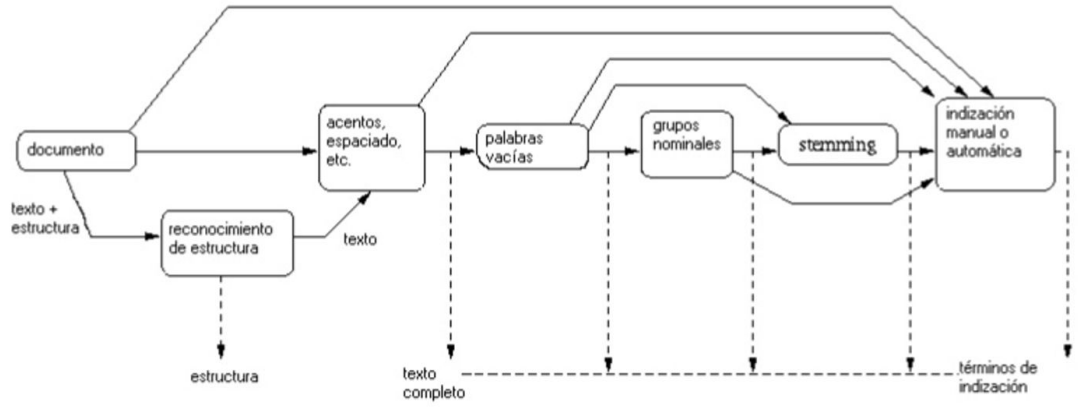

Fig. 1.2 Visión lógica de los documentos: desde el texto completo al conjunto de términos de indexación

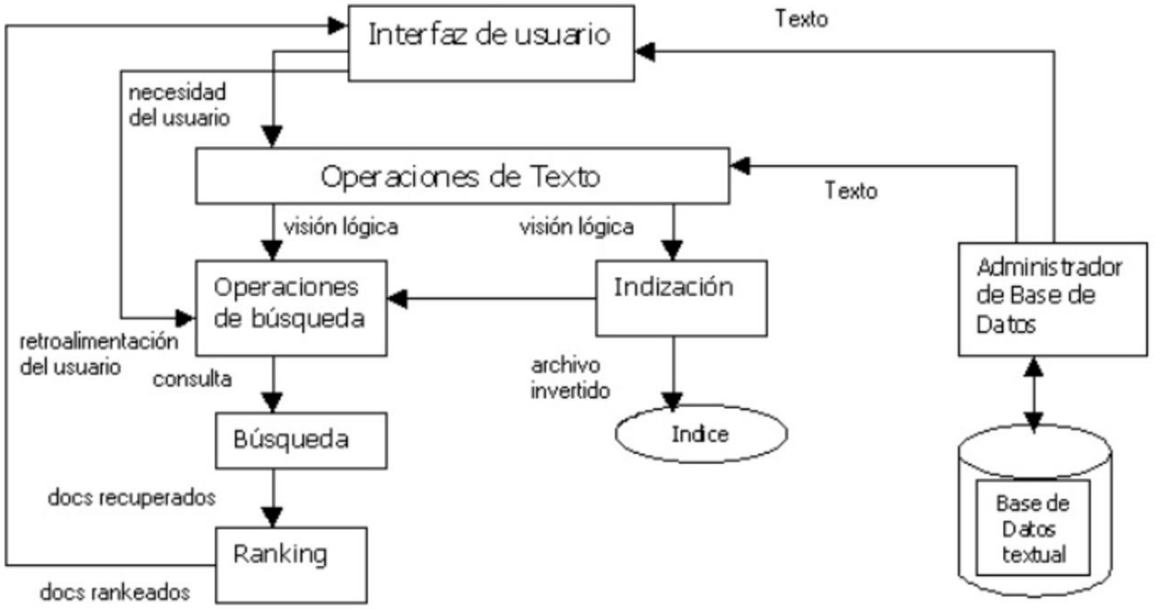

Fig. 1.3 El proceso de recuperación de información

Fuente: Ricardo Baeza, Capítulo 1.

- El programa debe ser capaz de leer un archivo de texto pasado como entrada y donde se alojan varios documentos en texto plano.
- Los archivos de texto solo contienen texto plano en inglés.

- El programa debe ser capaz de crear un índice invertido o una matriz de incidencia (según se especificará en cada proyecto) en base a los textos que se proporcione como entrada.
- Se debe poder consultar a través de un texto (consulta, frase, oración) compuesta de una o más palabras.
- Se deben poder mostrar los M primeros resultados, en base a un sistema de ranking. Este sistema de ranking se hará en base a la cantidad de incidencias de las palabras de la oración dentro de los distintos documentos leídos.
- Si dentro de una frase consultada, en algunos documentos sólo existen algunas palabras, éstas se deben incluir en los resultados pero con menor valoración en el ranking.
- El programa debe incluir una lista con *stopwords*, es decir, palabras que no se deben considerar al momento de construir el índice o al momento de la consultalo(ej.: el, la, los, en,...).
- El programa debe mostrar el título e identificador de los documentos en donde se encuentran los términos del texto utilizado como consulta.
- El programa debe mostrar el contenido de los documentos en donde se encuentran los términos del texto utilizado en la oración consultada.
- Se debe dar la posibilidad de agregar más archivos una vez creado el índice, actualizar el índice y posteriormente los resultados de las consultas.
- Se debe permitir guardar los resultados de las búsquedas en un archivo.
- Se debe permitir guardar el índice en un archivo.
- Se debe permitir cargar un índice desde un archivo.

Comodines: Durante el semestre cada alumno dispondrá de un total de **DOS comodines** que podrán utilizar **solo bajo las siguientes condiciones**:

- **Si su calificación** en los laboratorios 1, 2, 3 y parcialmente en el 4 (Ya sea por requerimientos funcionales, no funcionales, informe y/o manual de usuario) **es inferior a 4.0 pero mayor que 2.0**.

El comodín permitirá al alumno entregar lo necesario (requerimientos funcionales, no funcionales, informe y/o manual de usuario) para alcanzar la nota mínima requerida (4.0) en la correspondiente entrega de laboratorio para la aprobación al final del semestre. **IMPORTANTE:** Un comodín se asocia a un laboratorio como un todo, por lo que este permitirá someter a evaluación una nueva versión de cualquiera de sus partes (i.e., informe, manual, programa) de manera individual o conjunta en caso de que las calificaciones hayan sido inferior a 4.0 y superior o igual a 2.0. A modo de ejemplo, si en el primer laboratorio su calificación de manual, informe y programa son inferiores a 4.0 y superiores o iguales a 2.0, entonces podrá presentar una nueva versión de su manual, informe y programa para someterla a evaluación. Todo esto cuenta como un comodín.

Antes de proceder al uso de comodines, y si lo amerita, se recomienda a los alumnos que empleen las instancias de recorrecciones de laboratorio.

En caso de requerir el uso de comodín, podrá hacer entrega del correspondiente laboratorio corregido al final del semestre.

Finalmente, por temas de tiempo considerando el avance del semestre, en el laboratorio final se podrá hacer un uso limitado de comodines (en caso de que el estudiante aún disponga de alguno). Esto quedará a disposición de los profesores correctores dependiendo del nivel de los elementos que son calificados como insuficientes y la factibilidad de poder completarlos en los plazos para el cierre oficial del semestre.

Laboratorio 1 (Paradigma Imperativo Procedural - Lenguaje C)

Versión 1.0

Fecha de Entrega: 7 de Septiembre de 2017

Entregables: Archivo ZIP con el siguiente nombre de archivo: lab1_rut_ApellidoPaterno.zip (**ej: lab1_12123456_Perez**). Notar que no incluye dígito verificador. Dentro del archivo se debe incluir:

1. Informe en formato Word o PDF.
2. Manual de Usuario en formato Word o PDF.
3. Carpeta con código fuente y makefile para compilación.
4. Carpeta con ejecutable.
5. Archivo leeme.txt para cualquier instrucción especial para compilación u otro, según aplique.

Objetivo del laboratorio: Usar los conceptos abordados en cátedra con respecto al paradigma de programación imperativo-procedural usando el lenguaje de programación C en la solución a un problema específico.

Resultado esperado (entregable): Programa para la creación y posterior consulta de un índice de documentos, el cual permitirá generar un ranking de resultados similar a lo que realiza un buscador web en la actualidad.

Informe del proyecto: Extensión no debe superar las 10 planas y debe incluir: portada, índice, introducción, descripción del problema abordado con el paradigma y lenguaje, análisis del problema, diseño de la solución implementada, resultados obtenidos y conclusiones. **Contenido requerido y un formato tipo se encuentran disponibles en Usach Virtual.**

Manual de Usuario: Extensión no debe superar las 10 planas y debe incluir: portada, índice, introducción, contenido central del manual de usuario. **Contenido requerido y un formato tipo se encuentran disponible en Usach Virtual.**

Evaluación: Tanto el informe (*Inf*), Manual (*Man*), requerimientos funcionales (*RF*) y no funcionales (*RNF*) se evalúan por separado. La nota final de este laboratorio (*NL*) se calcula de la siguiente forma considerando sólo la calificación base a partir del cumplimiento de los *RF* y *RNF* obligatorios.

$$
\begin{aligned}
&\textit{if } (Inf >= 4.0 \;\&\&\; Man >= 4.0 \;\&\&\; (RF + 1.0) >= 4.0 \;\&\&\; (RNF + 1.0) >= 4.0) \textit{ then}\\
&\qquad NL = 0.05 \cdot Inf + 0.05 \cdot Man + 0.7 \cdot (RF + 1.0 + PtjeExtra) + 0.2 \cdot (RNF + 3.5)\\
&\textit{else}\\
&\qquad NL = min(Inf, Man, RF + 1.0, RNF + 1.0)
\end{aligned}
$$

Requerimientos No Funcionales obligatorios (0,2 pts. c/u. Requerimientos 10 al 14 son obligatorios. El no cumplimiento de estos implica la nota mínima para RNF)

1. La implementación debe ser en el lenguaje de programación C.
2. Se deben emplear sólo recursos de ANSI C.
3. Se debe garantizar la compilación y ejecución correcta del programa tanto en Windows como en Linux.
4. En caso de utilizar archivos de texto plano, se debe utilizar el formato de salto de línea de Unix, basado en '\n'.
5. Se debe trabajar con archivos de texto plano y/o en formato binario para las palabras usadas para generar el índice.
6. La implementación debe ser lo más genérica posible permitiendo el almacenamiento y lectura de nuevos datos los cuales se definirán a partir de los diseños específicos de cada implementación del buscador.
7. Se deben validar las entradas a fin de controlar errores durante la lectura de archivos.
8. Se debe documentar el diseño estructural de su solución, así como los principales aspectos de funcionamiento del mismo a través de diagramas ad hoc (ej.: diagramas de flujo).
9. Se debe documentar el código indicando una breve descripción del programa, funciones y procedimientos, parámetros de entrada y salida, tipo de paso de parámetros y ejemplo de llamado.
10. Organización del código (archivos, encabezados, etc.). Estructurar programa empleando archivos de cabecera (.h) y los códigos fuentes (.c)
11. Makefile para la correcta compilación del proyecto en Windows y Linux (compatible con MinGW o GCC). De ser necesario debe proveer dos makefile.
12. Se debe respetar las interfaces definidas en el enunciado, es decir: Tipo de retorno, nombre de función, orden y tipos de parámetros de entrada.
13. La estructuras centrales del buscador son:

```
/*Corresponde a la estructura donde se mantiene el índice de documentos*/
typedef struct Index {
	....;
	....;
} Index;

/*Representa un resultado individualizado a partir de una consulta. Esta estructura podría contener datos como: ID del documento, título del documento, autores, extracto del texto, detalles de la ubicación de los términos de la consulta en el documento, ranking asignado (un puntaje o similar)*/
typedef struct Result {
	....;
	....;
} Result;

/*Corresponde a un conjunto de resultados con un order determinado por una función de ranking, se señala la cantidad de resultados en la lista, la consulta que generó este ranking y otros datos que puedan ser de interés.*/
typedef struct Ranking {
	....;
	....;
} Ranking;
```

```
	/*Estructura que contiene el listado completo de stopwords detallando además
otros datos como la cantidad total de stopwords en el listado, el idioma de las
stopwords, etc.*/
	typedef struct StopWords {
		....;
		....;
	} StopWords;
```

14. Para efectos de reportar el estado de una función o procedimiento al término de su ejecución se emplea la siguiente enumeración:

    ```
    enum code {OK, ERR_FILE_NOT_FOUND, ERR_FILE_NOT_PERM};
    ```

    Si lo amerita, según sea el caso, puede incorporar otros valores con nombres adecuados a esta enumeración.

15. Construir un archivo separado llamado *buscador.c* donde se encuentre el procedimiento o función *main* a fin de demostrar el uso de sus funciones. Procure entregar resultados por pantalla sobre cada llamado. Para la entrega procure comentar estas líneas de código y solo habilite la o las líneas de código que permiten desplegar el menú del buscador.

Requerimientos Funcionales Obligatorios (0,571 pts. c/u)

1. loadStopWords: permite leer y almacenar las *stopwords* desde un archivo de texto ubicado en *pathStopWordsFile*.

   StopWords loadStopWords(char* pathStopWordsFile, code *statusCode);*

2. createIndex: permite la creación de un índice a partir de una archivo de entrada ubicado en *pathDocumentsFile.* En este archivo están contenidos N documentos. Para la creación del índice debe considerar la incorporación del conjunto de *stopwords*.

   Index createIndex(char* pathDocumentsFile, StopWords *sw, code *statusCode)*

3. saveIndex: permite almacenar el índice en un archivo de texto y autogenerar un id el cual es retornado a través del parámetro por referencia “id”. Dicho id permitirá posteriormente cargar un índice existente a través de la función loadIndex. Junto con el índice se debe almacenar la fecha y hora en que fue almacenado. En caso de error debe indicar la causa mediante el parámetro *statusCode*.

   *void saveIndex(Index *i, int *id, code *statusCode)*

4. loadIndex: permite recibir el “id” de un índice y lo retorna si es que éste se encuentra almacenado en un archivo, en caso de ocurrir un error debe retornar NULL e indicar el detalle del error en *statusCode*.

   Index loadIndex(int id, code *statusCode)*

Nota: Debe proveer dos o más archivos de índices de ejemplo para poder evaluar el funcionamiento de esta función.

5. query: permite consultar por una frase, la cual será evaluada dentro del índice y se generará una lista de resultados ordenadas por ranking del mayor al menor. La consulta toma en cuenta la lista de *stopwords* para el preprocesamiento de la consulta.

   Ranking query(Index *i, StopWords *sw, char* text, code *statusCode)*

6. displayResults: permite mostrar los resultados de la consulta almacenados en un Ranking. Debe permitir mostrar solamente los Top K resultados.

   *void displayResults(Ranking *r, int TopK, code *statusCode)*

7. Menú en main: Debe incluir un menú que permita probar cada una de las funciones/procedimientos descritos anteriormente. En caso de implementar requerimientos extras, estos se deben incluir en el menú.

Requerimientos extra (se consideran si y sólo si la evaluación de los requerimientos obligatorios es igual o superior a 4.0. La nota máxima que se puede alcanzar es un 7.0 en RF) (1pto c/u).

8. saveRanking: permite almacenar el ranking en un archivo de texto y autogenerar un id el cual es retornado a través del parámetro por referencia “id”. Dicho id permitirá posteriormente cargar un ranking existente a través de la función loadResults. Junto con el índice se debe almacenar la fecha y hora en que fue almacenado. En caso de error debe indicar la causa mediante el parámetro *statusCode*.

   *void saveRanking(Ranking *r, int *id, code *statusCode)*

9. loadRanking: permite recibir el “id” de un ranking y lo retorna si es que éste se encuentra almacenado en un archivo, en caso de ocurrir un error debe retornar NULL e indicar el detalle del error en *statusCode*.

   Ranking loadRanking(int id, code *statusCode)*

10. suggestQuery: A partir de una consulta, sugiere consultas similares usando ya sea un diccionario de consultas, consultas históricas, o consultas a partir de la ocurrencia contigua de términos en los documentos indexados..

    char suggestQuery(Index *i, char* query, code *statusCode)*

11. didYouMean: A partir de una consulta, sugiere una corrección de la consulta tomando en cuenta errores ortográficos (ej: didYouMean(“ligh”) => “light”). En este caso puede usar consultas históricas, diccionario de palabras o los mismos documentos.

```
char* didYouMean(Index *i,char* query, code *statusCode)
```