# Práctica 2: Sopa de Letras

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada


## Estructuras de Datos

Grado en Ingeniería Informática. Grupo C

# Índice de contenido

# 1. Introducción

Los objetivos de este guión de prácticas son los siguientes:

1. Asimilar los conceptos fundamentales de abstracción, aplicado al desarrollo de programas.
2. Documentar un tipo de dato abstracto (T.D.A)
3. Practicar con el uso de doxygen.
4. Profundizar en los conceptos relacionados especificación del T.D.A, representación del T.D.A., función de Abstracción e Invariante de la representación.
5. Entender como implementar funciones y clases plantilla.

Los requisitos para poder realizar esta práctica son:

1. Haber estudiado el Tema 1: Introducción a la eficiencia de los algoritmos
2. Haber estudiado el Tema 2: Abstracción de datos. Plantillas
3. Conocer el contenedor vector dinámico y listas como conjunto de celdas enlazadas

# 2. Tipos de datos abstractos.

Los tipos de datos abstractos son nuevos tipos de datos con un grupo de operaciones que proporcionan la única manera de manejarlos. De esta forma, debemos conocer las operaciones que se pueden usar, pero no necesitamos saber

- la forma en como se almacena los datos ni
- cómo se implementan las operaciones.

## 2.1. Selección de operaciones

Una tarea fundamental en el desarrollo de un T.D.A. es la selección del conjunto de operaciones que se usarán para manejar el nuevo tipo de dato. Para ello, el diseñador deberá considerar los problemas que quiere resolver en base a este tipo, y ofrecer el conjunto de operaciones que considere más adecuado. Las operaciones seleccionadas deben atender a las siguiente exigencias:

1. Debe existir un conjunto mínimo de operaciones para garantizar la abstracción. Este conjunto mínimo permite resolver cualquier problema en el que se necesite el T.D.A.
2. Las operaciones deben ser usadas con bastante frecuencia.
3. Debemos considerar que el tipo de dato sufra en el futuro modificaciones y conlleve también la modificación de las operaciones. Por lo tanto un número muy alto de operaciones puede conllevar un gran esfuerzo en la modificación.

Por otro lado las operaciones seleccionadas pueden clasificarse en dos conjuntos:

1. Fundamentales. Son aquellas necesarias para garantizar la abstracción. No es posible prescindir de ellas ya que habría problemas que no se podrían resolver sin acceder a la parte interna del tipo de dato. A estas funciones también se le denominan operaciones **primitivas.**
2. No fundamentales. Corresponden a las operaciones prescindibles ya que el usuario podría construirlas en base al resto de operaciones.

# 3. Documentación

El objetivo fundamental de un programador es que los T.D.A. que programe sean reutilizados en el futuro por él u otros programadores. Para que esta tarea pueda llevarse a cabo los módulos donde se materializa un T.D.A. deben de estar bien documentados. Para llevar a cabo una buena documentación de T.D.A. se deben crear dos documentos bien diferenciados:

1. **Especificación**. Es el documento donde se presentan las características sintácticas y semánticas que describen la parte pública ( la parte del T.D.A. visible a otros módulos). Con este documento cualquier otro módulo podría usar el módulo desarrollado y además es totalmente independiente de los detalle internos de construcción del mismo.
2. **Implementación.** Corresponden al documento que presenta las características internas del módulo. Para facilitar futuras mejoras o modificaciones de los detalles internos del módulo, es necesario que la parte de implementación sea documentada.

## 3.1. Especificación del T.D.A

En este caso vamos a especificar un tipo de dato junto con el conjunto de operaciones. Por lo tanto en la especificación de T.D.A. aparecerán dos partes:

1. **Definición.** En esta parte deberemos definir el nuevo tipo de dato abstracto, así como todos los términos relacionados que sean necesarios para comprender el resto de la especificación.
2. **Operaciones**. En esta parte se especifican las operaciones, tanto sintáctica como semánticamente.

### 3.1.1. Definición

Se dará una definición del T.D.A en lenguaje natural. Para ello el T.D.A se distinguirá como una nueva clase de objetos en la que cualquier instancia de esta nueva clase tomará valores (dominio) en un conjunto establecido. Por ejemplo si queremos definir un *Racional* diremos:

> ***Una instancia f del tipo de dato abstracto Racional es un objeto del conjunto de los números racionales, compuesto por dos valores enteros que representan, respectivamente, numerador y denominador. Lo representamos num/den.***

### 3.1.2. Operaciones

En esta parte se realiza una especificación de las operaciones que se usarán sobre el tipo de dato abstracto que se está construyendo. Cada una de estas operaciones representarán una función y por lo tanto se hará la especificación con los siguientes items:

1. Breve descripción. Que es lo que hace la función. Usando en doxygen la sentencia @brief.
2. Se especifican cada uno de los parámetros de la función. Por cada parámetro se especificará si el parámetros se modifica o no tras la ejecución de la función. Usando en doxygen el comando @param.
3. Las condiciones previas a la ejecución de la función ( precondiciones ) que deben cumplirse para un buen funcionamiento de la función. Usando en doxygen la sentencia @pre.
4. Que devuelve la función. En doxygen usaremos el comando @return
5. Las condiciones que deben cumplirse trás la ejecución de la función (postcondiciones). Usando en doxygen el comando @post.

Supongamos que queremos especificar en nuestra clase *Racional* la función *Comparar* que compara un Racional con el Racional que apunta this. Una posible especificación de esta

función sería.

```
/**
 * @brief Compara dos racionales
 * @param r racional a comparar
 * @return Devuelve 0 si este objeto es igual a r,
 *              <0 si este objeto es menor que r,
 *              >0 si este objeto es mayor que r
 */
 bool comparar(Racional r);
```

Un ejemplo donde se usa una precondición es en la función *asignar* que se le asigna al Racional apuntado por this unos valores concretos para el numerador y denominador. En esta especificación cabe resaltar que si el nuevo denominador es 0 se estará violando las propiedades que deben mantenerse para una correcta instanciación de un objeto de tipo Racional.

```
/**
 * @brief Asignación de un racional
 * @param n numerador del racional a asignar
 * @param d denominador del racional a asignar
 * @return Asigna al objeto implícito el numero racional n/d
 * @pre d debe ser distinto de cero
 */
 void asignar(int n, int d);
```

## 3.2. Implementación del T.D.A.

Para implementar un T.D.A., es necesario en primer lugar, escoger una representación interna adecuada, una forma de estructurar la información de manera que podamos representar todos los objetos de nuestro tipo de dato abstracto de una manera eficaz. Por lo tanto, debemos seleccionar una estructura de datos adecuada para la implementación, es decir, un tipo de dato que corresponda a esta representación interna y sobre el que implementamos las operaciones. A éste tipo escogido ( la estructura de datos seleccionada), se le denomina ***tipo rep.***

Para nuestro T.D.A *Racional* las estructuras de datos posibles para representar nuestro ***tipo rep*** podrían ser por ejemplo:

- Un vector de dos posiciones para almacenar el numerado y denominador

```
Class Racional{
private:
int r[2];
....
}
```

- Dos enteros que representen el numerador y denominador respectivamente

```
Class Racional{
private:
int numerador;
int denominador;
….
}
```

De entre las representaciones posibles en el documento debe aparecer la estructura de datos escogida para el ***tipo rep***. Esta elección debe formalizarse mediante la especificación de la función de abstracción. Esta función relaciona los objetos que se pueden representar con el ***tipo rep*** y los objetos del tipo de dato abstracto. Las propiedades de esta función son:

- Parcial, todos los valores de los objetos del ***tipo rep*** no se corresponden con un objeto del tipo abstracto. Por ejemplo valores de numerador=1 y denominador =0 no son valores válidos para un objeto del tipo Racional.
- Todos los elementos del tipo abstracto tienen que tener una representación.
- Varios valores de la representación podrían representar a un mismo valor abstracto. Por ejemplo {4,8} y {1,2} representan al mismo racional.

Por lo tanto en la documentación podemos incluir esta aplicación para indicar el significado de la representación. Esta aplicación tiene dos partes:

1. Indicar exactamente cual es el conjunto de valores de representación que son válidos, es decir, que representen a un tipo abstracto. Por tanto, será necesario establecer una condición sobre el conjunto de valores del tipo *rep* que nos indique si corresponden a un objeto válido. Esta condición se denomina invariante de la representación.

$$f_{inv}: rep \dashrightarrow booleanos$$

2. Indicar para cada representación válida cómo se obtiene el tipo abstracto correspondiente, es decir la función de abstracción.

$$f_{abs}: rep \dashrightarrow A$$

Un invariante de la representación es “invariante” porque siempre es cierto para la representación de cualquier objeto abstracto. Por tanto, cuando se llama a una función del tipo de dato se garantiza que la representación cumple dicha condición y cuando se devuelve el control de la llamada debemos asegurarnos que se sigue cumpliendo. En nuestro ejemplo el T.D.A Racional en la documentación de la implementación incluiríamos lo siguiente:

```
class Racional {
 private:
/**
  * @page repRacional Rep del TDA Racional
  * @section invRacional Invariante de la representación
  * El invariante es \e rep.den!=0
  *  * @section faRacional Función de abstracción
  * Un objeto válido @e rep del TDA Racional representa al valor
  * (rep.num,rep.den)
  */
  int num; /**< numerador */
  int den; /**< denominador */
 public:
””””
```

# 4. Ejercicio

El objetivo en este ejercicio es crear el clásico pasatiempo “Sopa de Letras”.

Con tal fin vamos a desarrollar varios tipos de datos abstractos:

- ***Matriz_Dispersa de caracteres***: Es una matriz que crece de forma dinámica conforme se requiere más espacio. Mantiene un valor por defecto ya que todas las casillas no se almacenan en memoria. Así podemos decir que es una array 1-d de tripletas formadas por una fila, columna y un valor asociado de tipo carácter. Las tripletas están ordenadas por fila, y a igualdad de fila por columna. Solamente se almacenan los valores con un valor diferente al valor por defecto. Por ejemplo:

  {(-1,-1,'H'),(-1,0,'0'),(-1,2,'L'),(-1,3,'A'),(*,0)}

  En este caso el objeto *Matriz_Dispersa* tiene una fila con índice -1 y cuatro columnas que van desde -1 a 3. El valor por defecto es 0. Así si se pregunta por el contenido de la casilla 10,10 devolverá 0, si se pregunta por la casilla (-1,3) se devuelve el carácter ***A***. El número de filas de esta matriz es 1 y el número de columnas es 4. Siguiendo con nuestro ejemplo podríamos insertar la palabra “ADIOS” en dirección vertical abajo desde la posición (-1,3) obteniendo:

  {(-1,-1,'H'),(-1,0,'0'),(-1,2,'L'),(-1,3,'A'),(0,3,'D'),(1,3,'I'),(2,3,'O'),(3,3,'S')(*,0)}

  Ahora el número de columnas sigue siendo 4 pero el número de filas 4. Si preguntamos por la casilla (1,1) devuelve el carácter 0. Pero si pregunta por la casilla (3,3) devuelve 'S'.

- ***Sopa_Letras***: Contiene un objeto de Matriz_Dispersa que almacena con una colección de palabras dispuestas en la matriz en dirección horizontal, vertical o diagonal. Además contiene una lista con las palabras aún no descubiertas por el jugador y la lista de palabras descubiertas por el jugador. Las palabras se pueden colocar en las direcciones vertical, horizontal y diagonal.

Se pide desarrollar los tipos de datos abstractos (TDA): ***Matriz_Dispersa*** y ***Sopa_Letras.*** Para cada uno de estos tipo de datos abstractos:

1. Dar la especificación. Establecer una definición y el conjunto de operaciones básicas.
2. Determinar la estructura de datos ***tipo rep.***
3. Para la estructura de datos del ***tipo rep*** establecer cual es el invariante de la representación y función de abstracción.
4. Fijado el ***tipo rep*** realizar la implementación de las operaciones.
5. Haciendo uso de prueba.cpp y prueba_sl.cpp probar los tipos de datos abstractos desarrollados. Este fichero viene en el material dado al alumno/a. Es importante que este fichero no se modifique. Por lo tanto debemos estudiar que funciones o métodos son necesarios para nuestros tipos de datos para que este fichero pueda compilarse.

Como guía para llevar a cabo estos puntos se puede observar el T.D.A Racional dado en el material. En la sección 4.3 se da las operaciones mínimas que debería tener cada uno de los TDAs que se pide que desarrolle el alumno.

Con respecto al desarrollo de T.D.A ***Matriz_Dispersa,*** el alumno usará para su representación dos T.D.A ya conocidos: ***vector dinámico y lista.*** En la sección 4.3.1 se describe con más detalle como llevar a cabo la implementación de ***Matriz_Dispersa.***

## 4.1. Programa prueba_sp

El alumno creará el programa prueba_sp que se ejecuta desde la línea de órdenes de la siguiente forma:

```
prompt>bin/prueba_sp datos/animales_sp_small.txt
```

Este programa recibe un fichero con las palabras con las que se debe intentar construir la sopa de letras. Para ver los detalles del formato de los ficheros con las palabras ver la sección 4.2.

La salida que produce la anterior  orden es la siguiente:

```
TITULO : ANIMALES  Numero de Palabras Ocultas: 3 Numero de Palabras Descubiertas: 0

*************************
          0    1    2    3    4    5    6    7    8    9    10
     0    V    I    F    G    Y    H    P    D    D    J    I
     1    A    Y    E    W    E    Y    U    E    C    J    N
     2    C    B    S    H    H    P    A    L    O    M    A
     3    A    I    A    C    T    F    U    B    O    C    I
     4    O    J    Z    T    T    O    R    T    U    G    A


 Dime una palabra:
```

Como se puede ver en el cuadro anterior se imprime el título de la sopa de letras, el número de palabras ocultas y el número de palabras descubiertas. A continuación se muestra la sopa de letras, con la primera fila se muestra los índices de las columnas (de 0 a 10) y como primera columna los índices de las filas (de 0 a 4)

```
Dime una palabra: TORTUGA
Dime la fila :4
Dime la columna :4
Direccion Arriba (vu), Abajo (vd), Izquierda (hi) , Derecha (hd), Diagonal abajo derecha (dd),Diagonal
abajo izquierda (di) :hd
TITULO : ANIMALES  Numero de Palabras Ocultas: 2 Numero de Palabras Descubiertas: 1

******************
          0    1    2    3    4    5    6    7    8    9    10
     0    V    R    U    L    A    J    O    H    X    N    G
     1    A    R    N    D    M    Y    M    T    H    H    I
     2    C    O    T    U    F    P    A    L    O    M    A
     3    A    F    A    N    G    X    R    J    O    L    V
     4    O    W    J    W    T    O    R    T    U    G    A


Dime una palabra:
```

El usuario inserta la palabra que cree haber descubierto, en que fila y columna se inicia y en que dirección. Como se puede observar las direcciones son vertical arriba y abajo (vu, vd respectivamente), horizontal izquierda y derecha (hi, hd respectivamente) y diagonales (abajo derecha dd y abajo izquierda di). En el ejemplo el usuario indica la palabra TORTUGA que se encuentra en la fila 4 columna 4 y dirección horizontal derecha (hd). Se puede ver que el programa muestra la sopa de letras ahora indicando que le quedan dos palabras ocultas y ha descubierto una palabra.  Se muestra la sopa de letras con la palabra en negrita.  Así el usuario debe insertar de nuevo otra palabra hasta que el número de palabras ocultas sea 0. Como se puede observar al imprimir la  Sopa de Letras consecutivamente los caracteres que se dibujan cuando no forma parte de ninguna palabra puede ser diferentes. Así por ejemplo en la fila 4 columna 1 se imprimió una 'J'  y  a continuación se imprimió una 'W'.

Para escribir en negrita en la salida estándar podemos usar las siguientes funciones

```
/*************************************************************************/
std::ostream& bold_on(std::ostream& os)
{
    return os << "\e[1m";
}

/*************************************************************************/

std::ostream& bold_off(std::ostream& os)
{
    return os << "\e[0m";
}
//Con estas funciones para escribir en negrita podemos hacer
…
cout<<bold_on<<"HOLA"<<bold_off<<endl;
```

## 4.2. Fichero con las palabras para construir la sopa de letras.

Para poder probar nuestro programa usaremos un fichero compuesto de una serie de líneas. Cada línea se corresponde con información de palabras candidatas con la que construir la sopa de letras. Un ejemplo se muestra a continuación:

```
ANIMALES
4       4       hd      TORTUGA
0       0       vd      VACA
2       5       hd      PALOMA
-1      0       vu      RATON
```

El fichero contiene:

1. En la primera línea el título de la sopa de letras.
2. A continuación viene la información de las palabras con las que se quiere construir la sopa de letras. Estas se describen como:
    - Filas donde se inician las palabras
    - Columna donde se inicias las palabras
    - Dirección y sentido para como poner la palabra (hd: horizontal derecha, hi:horizontal izquierda, vd: vertical abajo, vu: vertical arriba, dd:diagonal abajo derecha, di:diagonal abajo izquierda).
    - La palabra que queremos poner.

Hay que tener en cuenta que al intentar colocar una palabra en nuestra sopa de letras puede ocurrir que no se pueda colocar, ya que existe parte de otra palabra en las casillas que ocuparía la palabra que queremos colocar y las letras no coinciden. Por lo tanto se aborta el poner esta palabra y se pasa a la siguiente.

Como se puede ver en el ejemplo de fichero tanto las fila como la columna puede adoptar valores enteros negativos y positivos.

## 4.3. Módulos a desarrollar.

Habrá al menos dos módulos que deberéis desarrollar: 1) El módulo asociado a ***Matriz_Dispersa*** ( Matriz_Dispersa.cpp y Matriz_Dispersa.h), 2) ***Sopa_Letras*** (Sopa_Letras.cpp y Sopa_Letras.h). Estos módulos tienen que tener la documentación suficiente para que cualquier usuario pueda usarlo sin problemas. Para poder documentar los módulos usaremos doxygen.

### 4.3.1. Módulo Matriz_Dispersa

Este módulo está dedicado a la especificación e implementación del TDA Matriz_Dispersa. Este TDA es una matriz con un número de filas y columnas determinadas en el que no tiene que almacenar en memoria todas las casillas asociadas a las filas por las columnas. Para ello la Matriz_Dispersa se define como: Un objeto de tipo matriz_dispersa es un array 1-d de tripletas formadas por: una fila, columna y el valor asociado de tipo carácter. Solamente se almacenan los valores con un valor diferente a un valor por defecto (0):

$$\{(i_0, j_0, m_{i_0,j_0}),(i_1, j_1, m_{i_1,j_1}),\ldots,,(i_{n-1}, j_{n-1}, m_{i_{n-1},j_{n-1}}),(\ *, d)\}$$

Con (∗, d) se representa todas aquellas casillas en la matriz que no están almacenadas en la matriz dispersa y que tienen asociado el valor d (en nuestro caso d=0). Las tripletas se ordenan de menor a mayor valor de fila, y a igualdad de fila, de menor a mayor columna.

Las operaciones mas relevantes de este T.D.A son:

- Constructor por defecto
- Constructor por parámetros: se le indica el valor por defecto
- Operadores de consulta:
    - Obtener el elemento en la posición *(i,j)*.
    - Obtener la menor fila de la matriz
    - Obtener la mayor fila de la matriz
    - Obtener la menor columna de la matriz
    - Obtener la mayor columna de la matriz
    - Consultar el valor por defecto
    - Indicar el número de casillas de la matriz que no tienen el valor por defecto
    - Obtener el número de filas de la matriz
    - Obtener el número de columnas de la matriz
- Operadores de modificación: modificar el elemento de la posición **(i,j)** a un determinado valor. Este valor puede ser el valor por defecto, y en este caso se elimina dicha casilla de memoria si existe en la matriz dispersa.
- Además se podría sobrecargar los operadores de escritura y lectura en flujos.

Vamos a dar dos versiones de este módulo. Uno que viene usando como ***tipo rep*** un vector dinámico implementado como template. Y otra versión que viene dada usando como ***tipo rep*** lista. De forma que estos dos módulos deberán desarrollarse usando templates o plantillas. Como se muestra a continuación:

```
template <class T>
class VD{
    T *datos;
    int n;
    int reservados;
…
}
```

```
template <class T>
struct Celda{
   T*dato;
   Celda<T> * siguiente;
};

template <class T>
class Lista {
    Celda<T> * primera;
…
}
```

```
//Versio1: Matriz_Dispersa

class Matriz_Dispersa{
    VD<tripleta> m;
    char valor_defecto;
…..

…
}
```

```
//Versio2: Matriz_Dispersa

class Matriz_Dispersa{
    Lista<tripleta> m;
    char valor_defecto;
….
}
```

Siendo tripleta

```
struct tripleta{
       int fila,col;
       char d;
};
```

En función de que **tipo rep** estéis usando debéis implementar las anteriores operaciones comentadas. Puede incluirse otras si lo veis necesario. Hay que tener en cuenta que la clase ***Matriz_Dispersa*** no almacena en ningún sitio el número de filas y el número de columnas que tiene, ya que se deduce de los valores almacenados. Por ejemplo para saber el número de filas que tiene la matriz podemos obtener cual es la mayor fila, cual es la menor fila y deducir a partir de estos el número de filas de la matriz.

### 4.3.2. Módulo Sopa_Letras

Un objeto de tipo ***Sopa_Letras*** contiene un conjunto de palabras dispuesto en la direcciones verticales, horizontal o diagonal en un matriz dispersa. Además contiene una lista de palabras no descubiertas por el usuario y una lista de palabras descubiertas por el usuario. Además la sopa de letras contiene un título que recoge la temática de la misma. Para escribir en negrita las acertadas puede usarse otras matriz dispersa indicando si la casilla es parte de una descubierta o no.

Las operaciones mínimas que debería tener un objeto de tipo ***Sopa_Letras*** son las siguientes:

- Constructor por defecto.
- Operaciones de Consultas:
    - Comprobar si una palabra a partir de una posición (fila,columna) y dirección se encuentra en la sopa de letras.
    - Comprobar si una palabra esta en la lista de palabras no descubiertas

        - Comprobar si es posible poner una palabra en una posición (i,j) y dirección. Una palabra puede colocarse si en cada casilla que ocupa tiene el valor por defecto, o si no es así el carácter en la matriz debe coincidir con el carácter que corresponda de la palabra.
        - Obtener el número de palabras no descubiertas por el usuario
        - Obtener el número de palabras descubiertas por el usuario
    - Operaciones de Modificación:
        - Colocar una palabra en una posición y con una dirección concreta.
        - Modificar una palabra en la matriz dispersa como acertada. Quitarla de palabras no descubiertas y moverla a palabras descubiertas.
        - Eliminar una palabra de la sopa de letras.
    - Operaciones de lectura y escritura de un flujo.

Cuando se imprime la sopa de letras aquellas casillas que tienen un valor por defecto se escoge para imprimir un carácter aleatorio.

## 4.4. Fichero para probar la sopa de letras

En el directorio del material que os damos tenéis el fichero prueba_sp.cpp:

FICHERO prueba_sp.cpp

```
1.  #include "matriz_dispersa.h"
2.  #include <cstring>
3.  #include <fstream>
4.  #include "sopa_letras.h"
5.  int main(int argc, char * argv[]){
6.     if (argc!=2){
7.       cout<<"Dime el nombre del fichero con las palabras de la sopa de letras"<<endl;
8.       return 0;
9.     }
10.    ifstream f(argv[1]);
11.    if (!f){
12.          cout<<"No puedo abrir "<<argv[1]<<endl;
13.          return 0;
14.    }
15.    Sopa_letras Sl;
16.    f>>Sl;//Leemos las palabras y construimos la sopa de letras
17.    while (Sl.NumPalabras()!=0){
18.       cout<<Sl<<endl;
19.       cout<<"Dime una palabra: ";
20.       string word;
21.       cin>>word;
22.       cout<<"Dime la fila :";
23.       int row;
24.       cin>>row;
25.       cout<<"Dime la columna :";
```

```
26.      int col;
27.      cin>>col;
28.      cout<<"Direccion Arriba (vu), Abajo (vd), Izquierda (hi) , Derecha (hd), Diagonal abajo
    derecha          (dd),Diagonal abajo izquierda (di) :";
29.      dir direccion;
30.      cin>>direccion;
31.      if (!Sl.Comprobar_Palabra(word,row,col,direccion)){
32.           cout<<"La palabra "<<word << " no esta"<<endl;
33.      }
34.      else{
35.           Sl.Poner_Acertada(word,row,col,direccion);
36.      }
37.   }
38.
39.   cout<<Sl<<endl;
40.
41. }
```

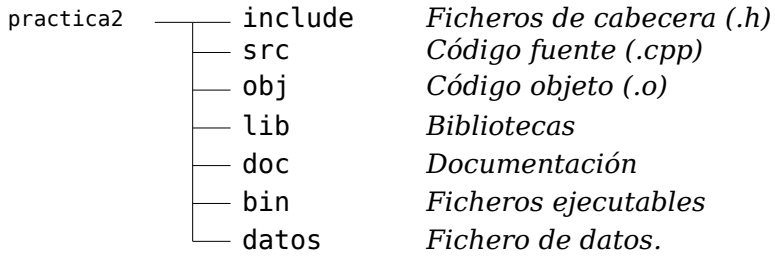
FIN de FICHERO prueba_sp.cpp

Este código debe funcionar con los TDA desarrollados.

# 5. Práctica a entregar

El alumno deberá empaquetar todos los archivos relacionados en el proyecto en un archivo con nombre "practica2.*tgz*" y entregarlo antes de la fecha que se publicará en la página web de la asignatura ( en PRADO). Tenga en cuenta que no se incluirán ficheros objeto, ni ejecutables, ni la carpeta datos. Es recomendable que haga una "limpieza" para eliminar los archivos temporales o que se puedan generar a partir de los fuentes.

El alumno debe incluir el archivo *Makefile* para realizar la compilación. Tenga en cuenta que los archivos deben estar distribuidos en directorios*:*




Para realizar la entrega, en primer lugar, realice la limpieza de archivos que no se incluirán en ella, y sitúese en la carpeta superior (en el mismo nivel de la carpeta "*practica2*") para ejecutar:

```
prompt% tar zcv practica2.tgz practica2
```

*tras lo cual, dispondrá de un nuevo archivo practica2.tgz que contiene la carpeta practica2 así como todas las carpetas y archivos que cuelgan de ella.*