Listas

- Algunos de los ejercicios contenidos en este documento se han de resolver en el Jutge (en la lista correspondiente del curso actual); aquí están señalados con la palabra *Jutge*.
- En general, los ejercicios contenidos en este documento se presentan por orden de dificultad. Por ello, recomendamos resolverlos en el orden en el que aparecen. No se supervisarán los problemas del Jutge si antes no se han resuelto los ejercicios previos.

Continuamos con los ejercicios de uso de las clases especiales vistas en clase de teoría, en este caso las listas. Para probar las listas usamos la Standard Template Library de C++ y su clase `list`. Para consultar detalles de su uso tenéis el fichero `list.pdf`. Recordad que en estas clases no se controla el cumplimiento de las precondiciones de las operaciones, por lo que interesará usar la opcion de compilación `-D_GLIBCXX_DEBUG` para obtener información adicional en caso de `segmentation fault`.

En la carpeta de la sesión se muestran ejemplos resueltos con listas de enteros. Además se incluyen los ficheros `listIOint.hh` y `listIOint.cc`, con operaciones de lectura y escritura para listas de enteros. Notad que dichas operaciones no pueden ser genéricas, pues dependen directamente del tipo del contenido de las estructuras. También hay ejemplos de como usar `tar` para hacer entregas en el *Jutge* y como codificar ficheros `Makefile`.

En las operaciones definidas específicamente como funciones, aseguraos de que al acabar el programa siempre queda una copia sin modificar de la estructura original.

1.1. Uso de la clase `list`

En los apuntes de teoría tenéis una serie de ejemplos sobre listas de enteros. Queremos obtener programas similares sobre listas de pares de enteros o de estudiantes. Elegid una de estas opciones y completad la secuencia de ejercicios que os proponemos.

Cada ejercicio que aparece a continuación requiere programar una o más operaciones de listas y un método `main` para probarlas.

1.1.1. Ejercicio: Búsqueda en una lista de pares de enteros

Ejercicio del Jutge X19134 de la lista *List & Map* En este ejercicio se ha de usar la clase `ParInt` que encontraréis en los archivos públicos del Jutge para representar pares de enteros. Dado un

número y una lista de pares de números enteros, comprobad si dicho número aparece como primer elemento de algún par de la lista y, en caso de éxito, escribid el par completo. Suponed que en la lista no existen repeticiones de los primeros elementos.

Necesitaréis producir los ficheros `LlistaIOParInt.hh` y `LlistaIOParInt.cc` con las cabeceras e implementación de operaciones para leer y escribir las listas de `ParInt`.

Probad varias situaciones distintas de búsquedas (que el elemento buscado sea el primero de la lista, que sea el último, que no esté, etc.)

Obtened una segunda versión de la solución, eliminando la restricción sobre las no repeticiones de elementos, que en lugar de buscar el par con el primer elemento dado, calcule el número de apariciones de dicho valor como primer elemento de los pares de la lista y la suma de sus compañeros. Si el elemento no existe, su número de apariciones será cero y definimos la suma de sus compañeros como cero.

Importante: El problema del Jutge corresponde a esta segunda versión.

1.1.2. Ejercicio: modificar los elementos de una lista

Producid dos soluciones, una en la que se obtenga una lista nueva a partir de la original y otra en la que las modificaciones se apliquen directamente sobre la original. Escribid un programa que dada una lista de pares de enteros le sume un valor *k* al segundo elemento de cada par.

1.1.3. Ejercicio: intersección de listas ordenadas

Ejercicio del Jutge X87360 de la lista *List & Map*. Especificad e implementad una operación eficiente que obtenga la intersección de dos listas ordenadas de enteros. Por ejemplo, con las listas [-7 1 5 12] y [1 3 4 5] debería obtener [1 5].

La versión del Jutge es `void` y sustituye una de las listas por la intersección de ambas. Otra posible versión consistiría en producir una tercera lista, pasada por referencia.

1.1.4. Ejercicio: Estadísticas de una secuencia de enteros con borrado

Ejercicio del Jutge X27494 de la lista *List & Map*. Consideremos una secuencia de pares de números enteros. El primer elemento de cada par es un código de operación denotado por un valor negativo: si es -1 significa que su compañero es relevante para las estadísticas; si es -2, significa que una de las apariciones de su compañero, si ha habido alguna, pasa a no ser relevante. Dicho de otra forma, es como si una de las apariciones de dicho número se hubiera borrado de la secuencia. Si el compañero no ha aparecido en la secuencia o se han borrado todas sus apariciones, no se borra nada. El par 0 0 marca el final de la secuencia.

Cada vez que se trate un par de números de la secuencia, se han de obtener las estadísticas relativas a los datos relevantes acumulados hasta el momento: el mínimo, el máximo y la media.

Si tras un -2 no queda ningún elemento relevante (tanto si es porque se ha borrado el único que había o porque previamente ya no había ninguno), solo se ha de escribir un cero.

Las procesos iterativos auxiliares de las estadísticas han de calcularse en operaciones aparte, así como el borrado. Organizad el programa de forma que el método `main` solamente lea los

datos, aplique el tratamiento oportuno tras cada lectura y escriba los resultados. Para cada nuevo par sólo se puede recorrer la secuencia una vez como máximo y solo si es estrictamente necesario.

En el fichero `estadisticas_lista_largo.dat` se incluye un ejemplo de entrada para este ejercicio y en `estadisticas_lista_largo.sal` se encuentra la salida correspondiente. Probad vuestro programa en otras situaciones, por ejemplo, una secuencia con elementos repetidos, uno de ellos el máximo o el mínimo.

1.1.5. Punto medio de una lista

Ejercicio X27494 de la lista *List & Map*

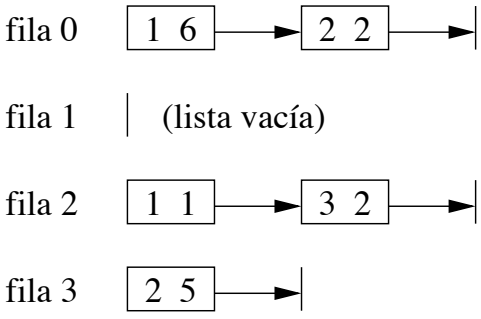
1.1.6. Ejercicio: implementación de matrices dispersas

Si sabemos que vamos a trabajar con matrices numéricas que contienen muchos ceros, podemos ahorrar espacio y tiempo si las representamos de forma compacta o dispersa, de manera que solo guardamos los valores distintos de cero. Para guardar una matriz m de tales características proponemos usar un vector de listas, donde cada posición representa una fila de la matriz. Dentro de la lista de la posición i se guardan los pares (j,x) tales que $m[i][j] = x$ y $x \neq 0$. Los pares de una misma fila estarán ordenados de forma creciente según su primera componente (es decir, el índice de la columna correspondiente).

Ejemplo: la matriz

$$m = \begin{pmatrix} 0 & 6 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 5 & 0 \end{pmatrix}$$

se representaría así:



Al implementar cualquier operación fuera de la clase sufriríamos una ineficiencia estructural, porque el acceso a la posición (i, j) de una matriz (es decir, el equivalente a $m[i][j]$) requiere recorrer la lista de la fila i hasta encontrar o superar la columna j. Para estar en condiciones de

evitar dicha ineficiencia, es conveniente implementar la mayor cantidad posible de operaciones dentro de la clase, ya que así podremos aprovechar el acceso a sus campos.

Notad que es posible implementar de forma eficiente las operaciones `suma` y `producto` dentro de la clase, adaptando el esquema de la soluciones normales que vimos en la sesión 1. `suma` solo necesita un iterador por cada una de las tres matrices implicadas (m_1, m_2 y el resultado). Sin embargo, para no perder eficiencia en comparación con la solución normal, `producto` requiere un iterador para m_1, otro para el resultado y *un vector de iteradores* para m_2.