# lista

**Ejercicio 1** Escribir una función `void apply_map(list<int> &L, map<int,int> &M, list<int> &ML)` que, dada una lista `L` y una correspondencia `M` retorna por `ML` una lista con los resultados de aplicar `M` a los elementos de `L`. Si algún elemento de `L` no esta en el dominio de `M`, entonces el elemento correspondiente de `ML` no es incluido. Por ejemplo, si `L = (1,2,3,4,5,6,7,1,2,3)` y `M= (1,2),(2,3),(3,4),(4,5),(7,8)`, entonces, después de hacer `apply_map(L,M,ML)`, debe quedar `ML = (2,3,4,5,8,2,3,4)`. Restricciones: No usar estructuras auxiliares. El tiempo de ejecucion del algoritmo debe ser O(n), donde `n` es el numero de elementos en la lista, asumiendo que las operaciones usadaas de correspondencia son O(1). [Tomado en el 1er parcial del 20/4/2006]. Resuelto en el archivo `apply-map.cpp`

**Ejercicio 2** Dos corresponencias `M1` y `M2` son inversas una de la otra si tienen el mismo numero de asignaciones y para cada par de asignacion `x->y` en `M1` existe el par `y->x` en `M2`. Escribir una funcion predicado `bool areinverse(map<int,int> &M1,map<int,int> &M2);` que determina si las correspondencias `M1`, `M2` son una la inversa de la otra o no. [Tomado en Primer Parcial 17-SET-2009]. Resuelto en el archivo `areinverse.cpp`

**Ejercicio 3** En ciertas aplicaciones interesa separar las corridas ascendentes en una lista de números $L =$ ( $a\_1$, $a\_2$, ..., $a\_n$ ), donde cada corrida ascendente es una sublista de números consecutivos $a\_i$, $a\_i + 1$, ..., $a\_i + k$, la cual termina cuando $a\_i + k > a\_i + k + 1$, y es ascendente en el sentido de que $a\_i \leq a\_i + 1 \leq ... \leq a\_i + k$. Por ejemplo, si la lista es L = (0,5,6,9,4,3,9,6,5,5,2,3,7), entonces hay 6 corridas ascendentes, a saber: (0,5,6,9), (4), (3,9), (6), (5,5) y (2,3,7). *Consigna:* usando las operaciones de la clase lista, escribir una función `int ascendente (list <int> &L, list < list<int> > &LL)` en la cual, dada una lista de enteros `L`, almacena cada corrida ascendente como una sublista en la lista de listas `LL`, devolviendo además el número $z$ de corridas ascendentes halladas. *Restricciones:* a) El tiempo de ejecución del algoritmo debe ser $O(n)$, b) La lista de listas `LL` inicialmente está vacía, c) No usar otras estructuras auxiliares. [Tomado en Examen Final 29-JUL-2004]. Resuelto en el archivo `ascendente.cpp`

**Ejercicio 4** Escribir una función `void chunk_revert(list<int> &L,int n);` que dada una lista `L` y un entero `n`, invierte los elementos de la lista tomados de a `n`. Si la longitud de la lista no es múltiplo de `n` entonces se invierte el resto también. Por ejemplo, si `L=1,3,2,5,4,6,2,7` entonces después de hacer `chunk_revert(L,3)` debe quedar `L=2,3,1,6,4,5,7,2`. Restricciones: Usar a lo sumo una estructura auxiliar. (En tal caso debe ser lista, pila o cola). [Tomado en el 1er parcial 21/4/2005]. Resuelto en el archivo `chunk-revert.cpp`

**Ejercicio 5** Coloquemos $n$ números enteros positivos alrededor de una circunferencia inicial. Construyamos ahora sucesivas circunferencias concéntricas *hacia el exterior*, de igual cantidad de elementos, los cuales son obtenidos restando (en valor absoluto) pares consecutivos de la última circunferencia exterior. Entonces, puede verificarse que si $n = 2^k$ en alguna iteración $p$ apare-

cerán $n$ números iguales. En ese momento se detiene la iteración. Por ejemplo, supongamos $k = 2$, $(n = 4)$ y que la circunferencia inicial sea $C\_0 = (8, 2, 5, 7)$, entonces iteramos y obtendremos sucesivamente, $C\_1 = (6, 3, 2, 1)$, $C\_2 = (3, 1, 1, 5)$, $C\_3 = (2, 0, 4, 2)$, $C\_4 = (2, 4, 2, 0)$ y $C\_5 = (2, 2, 2, 2)$, por lo que el número de circunferencias iteradas es $p = 5$. Entonces, dada una lista $L = [x\_0, x\_1, ..., x\_n - 1]$ de $n$ números enteros que representan los valores iniciales alrededor de la circunferencia inicial, escribir una función `int circulo(list<int> & L);` que ejecuta esta tarea y devuelva además el número de circunferencias iteradas $p$. *Restricción:* el algoritmo debe ser *in place*. *Ayuda:* Pensar a la lista en un *"sentido circular"*. Tener cuidado al generar la diferencia correspondiente al extremo. [Tomado en el 1er parcial del 21/4/2005]. Resuelto en el archivo `circulo.cpp`

**Ejercicio 6** Escriba procedimientos para concatenanar: a) dos listas `L1` y `L2` usando `insert`; b) un vector `VL` de `n` listas usando `insert`; c) una lista `LL` de `n` sublistas usando `insert` "básico"; d) una lista `LL` de `n` sublistas usando una opción de `insert`; e) una lista `LL` de `n` sublistas usando `splice`. Resuelto en el archivo `concatena.cpp`

**Ejercicio 7** Escribir una función `void creciente(queue<int> &Q)` que elimina elementos de `Q` de tal manera de que los elementos que quedan estén ordenados en forma creciente. [Tomado en el 1er parcial 27-APR-2004] Resuelto en el archivo `creciente.cpp`

**Ejercicio 8** Implemente una función `encuentra(list<int> &L1, list<int> &L2, list<int> &indx)` que verifica si los elementos de 'L2' estan en 'L1' (en el mismo orden, pero no necesariamente en forma consecutiva). Si es asi, retorna true y en 'indx' retorna los indices de los elementos de 'L1' que corresponden a los elementos de 'L2'. Resuelto en el archivo `encuentra.cpp`

**Ejercicio 9** Escriba procedimientos para intercalar (*merge*): (i) dos listas ordenadas `L1` y `L2` en una nueva lista `L`; (ii) un vector `VL` de `n` listas ordenadas como nueva lista `L`. Notar que *intercalar* (*merge*) implica en ambos casos que la nueva lista `L` debe resultar también *ordenada*. Resuelto en el archivo `intercala.cpp`

**Ejercicio 10** Dada una correspondencia `M` y asumiendo que es invertible o biunivoca (esto es, todos los valores del contradominio son distintos), la correspondencia 'inversa' `N` es aquella tal que, si `y=M[x]`, entonces `x=N[y]`. Por ejemplo, si `M=(0,1),(1,2),(2,0)`, entonces la inversa es `N=(1,0),(2,1,(0,2))`. Consigna: Escribir una función `bool inverse(map<int,int> &M,map<int,int> &N)` tal que, si `M` es invertible, entonces retorna true y `N` es su inversa. En caso contrario retorna falso y `N` es la correspondencia 'vacia' (sin asignaciones) [Tomado en el 1er parcial del 20/4/2006]. Resuelto en el archivo `inverse.cpp`

**Ejercicio 11** Escribir una función `void junta (list <int> &L, int n)` que, dada una lista `L`, agrupa de a `n` elementos dejando su suma IN PLACE. Por ejemplo, si la lista `L` contiene L=(1,3,2,4,5,2,2,3,5,7,4,3,2,2), entonces depués de `junta (L,3)` debe quedar L=(6,11,10,14,4). Prestar atención a no usar posiciones inválidas después de una supresión. El algoritmo debe tener un tiempo de

ejecución `O (m)`, donde `m` es el número de elementos en la lista original. [Tomado en el examen final del 1/8/2002] Resuelto en el archivo `junta.cpp`

**Ejercicio 12** Escribir una función `void ordenag (list <int> &l, int m)` que, dada una lista `l`, va ordenando sus elementos de a grupos de `m` elementos. Por ejemplo si m=5, entonces `ordenag` ordena los primeros 5 elementos entre si, despues los siguientes 5 elementos, y asi siguiendo. Si la longitud `n` de la lista no es un múltiplo de `m`, entonces los últimos `n mod m` elementos también deben ser ordenados entre si. Por ejemplo, si l = (10 1 15 7 2 19 15 16 11 15 9 13 3 7 6 12 1), entonces después de `ordenag (5)` debemos tener l = (1 2 7 10 15 11 15 15 16 19 3 6 7 9 13 1 12). [Tomado en el examen final del 5-Dic-2002]. Resuelto en el archivo `ordenag.cpp`

**Ejercicio 13** Usando las operaciones del TAD lista, escribir una función `void particiona (list<int> &L, int a)` la cual, dada una lista de enteros `L`, reemplace aquellos que son mayores que `a` por una sucesión de elementos menores o iguales que `a` pero manteniendo la suma total constante. [Ejercicio tomado en el Exámen Final del 05/07/01] Resuelto en el archivo `particiona.cpp`

**Ejercicio 14** Escriba una función `void print_back (list<int> & L, list <int>::iterator p)` que, en forma *recursiva*, imprima una lista en sentido inverso, es decir, desde el final al principio de la lista. Se le da como dato el procedimiento a la primera posición de la lista. [Ejercicio 3 del final del 14/02/2002] Resuelto en el archivo `print_back.cpp`

**Ejercicio 15** Usando las operaciones del TAD lista, escribir una función `void random_shuffle (list <int> &L)` que, dada una lista de enteros `L`, reordena sus elementos en forma aleatoria. Se sugiere el siguiente algoritmo: usando una lista auxiliar `Q` se van generando números enteros desde 0 a `length (L) - 1`. Se extrae el elemento j-ésimo de `l` y se inserta en `Q`. Finalmente, se vuelven a pasar todos los elementos de la cola `Q` a la lista `L`. [Ejercicio tomado en el Exámen Final del 05/07/01] Resuelto en el archivo `random_shuffle.cpp`

**Ejercicio 16** Dada una lista de enteros `L` y dos listas `SEQ` y `REEMP` escribir una función `void reemplaza (list<int> &L, list<int> &SEQ, list<int> &REEMP)` que busca todas las secuencias de `SEQ` en `L` y las reemplaza por `REEMP`. Por ejemplo, si L=(1 2 3 4 5 1 2 3 4 5 1 2 3 4 5), SEQ=(4 5 1) y REEMP=(9 7 3), entonces despues de llamar a `reemplaza` debe quedar L=(1 2 3 9 7 3 2 3 9 7 3 2 3 4 5). Este procedimiento tiene un efecto equivalente a la función `reemplazar` de los editores de texto. [Tomado el 1er parcial, 16 abril 2002] Resuelto en el archivo `reemplaza.cpp`

**Ejercicio 17** (a) Escriba una función `void refina (list<double> &L, double delta)` tal que dada una lista inicial de reales clasificados de menor a mayor `L`, refina inserta elementos entre los de `L`, de tal modo que la diferencia máxima entre elementos de la lista final sea menor o igual que delta; (b) Escriba una función `void desrefina (list<double> &L, double delta)` tal que dada una lista inicial de reales clasificados de menor a mayor `L`, desrefina suprime elementos

de L, de tal modo que la diferencia minima entre elementos de la lista final sea mayor o igual que delta. Resuelto en el archivo `refina.cpp`

**Ejercicio 18** Usando las operaciones del TAD lista, escribir una función `void rejunta (list<int> &L, int A)` que, dada una lista de enteros `L`, agrupe elementos de tal manera que en la lista queden solo elementos mayores o iguales que `A`. El algoritmo recorre la lista y, cuando encuentra un elemento menor, empieza a agrupar el elemento con los siguientes hasta llegar a `A` o hasta que se acabe la lista. Por ejemplo, si L=[3,4,2,4,1,4,4,3,2,2,4,1,4,1,4,4,1,4,4,2], entonces rejunta (L,10) da L=[13,12,13,10,10]. En la lista final NO deben quedar elementos menores que `A` salvo, eventualmente, el último. [Ejercicio tomado en el Exámen Final del 05/07/01] Resuelto en el archivo `rejunta.cpp`

**Ejercicio 19** Escriba procedimientos para insertar, suprimir y buscar un elemento en una lista ordenada `L`. Versión **sin** `funciones genéricas` (comparar con `sorted_list2.cpp` y `sorted_list3.cpp`). Resuelto en el archivo `sorted_list1.cpp`

**Ejercicio 20** Escriba procedimientos para insertar, suprimir y buscar un elemento en una lista ordenada `L`. Versión **únicamente** con `funciones genéricas` (comparar con `sorted_list1.cpp` y `sorted_list3.cpp`). Resuelto en el archivo `sorted_list2.cpp`

**Ejercicio 21** Escriba procedimientos para insertar, suprimir y buscar un elemento en una lista ordenada `L`. Versión mediante una `clase genérica` (comparar con `sorted_list1.cpp` y `sorted_list2.cpp`). Resuelto en el archivo `sorted_list3.cpp`

## arbol orientado

**Ejercicio 1** Listado de árboles orientados en diferentes ordenes. Orden previo, posterior y simétrico. Resuelto en el archivo `listarbo.cpp`

**Ejercicio 2** El listado en orden de nivel de los nodos de un árbol lista primero la raiz, luego todos los nodos de profundidad 1, después todos los de profundidad 2, y asi sucesivamente. Los nodos que estén en la misma profundidad se listan en orden de izquierda a derecha. Escribir una función `void orden_de_nivel (tree <int> &t)` para listar los nodos de un árbol en orden de nivel. Resuelto en el archivo `orden_nivel.cpp`

## correspondencia

**Ejercicio 1** Escribir una función `void apply_map(list<int> &L, map<int,int> &M, list<int> &ML)` que, dada una lista `L` y una correspondencia `M` retorna por `ML` una lista con los resultados de aplicar `M` a los elementos de `L`. Si algún elemento de `L` no esta en el dominio de `M`, entonces el elemento correspondiente de `ML` no es incluido. Por ejemplo, si `L = (1,2,3,4,5,6,7,1,2,3)` y `M= (1,2),(2,3),(3,4),(4,5),(7,8)`, entonces, después de hacer `apply_map(L,M,ML)`,

debe quedar `ML = (2,3,4,5,8,2,3,4)`. Restricciones: No usar estructuras auxiliares. El tiempo de ejecucion del algoritmo debe ser O(n), donde `n` es el numero de elementos en la lista, asumiendo que las operaciones usadaas de correspondencia son O(1). [Tomado en el 1er parcial del 20/4/2006]. Resuelto en el archivo `apply-map.cpp`

**Ejercicio 2** Dos corresponencias `M1` y `M2` son inversas una de la otra si tienen el mismo numero de asignaciones y para cada par de asignacion `x->y` en `M1` existe el par `y->x` en `M2`. Escribir una funcion predicado `bool areinverse(map<int,int> &M1,map<int,int> &M2);` que determina si las correspondencias `M1`, `M2` son una la inversa de la otra o no. [Tomado en Primer Parcial 17-SET-2009]. Resuelto en el archivo `areinverse.cpp`

**Ejercicio 3** Dada una correspondencia `M` y asumiendo que es invertible o biunivoca (esto es, todos los valores del contradominio son distintos), la correspondencia 'inversa' `N` es aquella tal que, si `y=M[x]`, entonces `x=N[y]`. Por ejemplo, si `M=(0,1),(1,2),(2,0)`, entonces la inversa es `N=(1,0),(2,1,(0,2))`. Consigna: Escribir una función `bool inverse(map<int,int> &M,map<int,int> &N)` tal que, si `M` es invertible, entonces retorna true y `N` es su inversa. En caso contrario retorna falso y `N` es la correspondencia 'vacia' (sin asignaciones) [Tomado en el 1er parcial del 20/4/2006]. Resuelto en el archivo `inverse.cpp`