Algoritmos y Estructuras de Datos.
Recuperatorio 2. [2013-12-19]

ATENCIÓN (1): Escribir cada sección en **hojas independientes**, poniendo claramente el nombre en la parte superior derecha de la misma.
ATENCIÓN (2): Para aprobar deben obtener un **puntaje mínimo** de

- **50 %** en clases.
- **50 %** en programación.
- **50 %** en operativos.
- **60 %** sobre las preguntas de teoría.

ATENCIÓN (3): Recordar que tanto en las clases (Ej. 1) como en los ejercicios de programación (Ej 2.) **deben usar la interfaz STL**.

[Ej. 1] [CLAS1 (W=20)]

a) **[list]** Escribir la implementación en `C++` del TAD lista (clase `list`) implementado por punteros **ó** cursores. Los métodos a implementar son `insert(p,x)`, `erase(p)`, `next()/iterator::operator++(int)` (postfijo), `list()`, `begin()`, `end()`.

b) **[stack]** Escribir la implementación en `C++` de los métodos `push`, `pop`,`front` y `top` de los TAD pila y cola (clases `stack` y `queue`), según corresponda.

c) **[AOO]** Para la clase `tree<>` (Arbol Ordenado Orientado) implementado por celdas enlazadas por punteros, declarar las clases `tree`, `cell`, `iterator`, (preferentemente respetando el anidamiento), incluyendo las declaraciones de datos miembros. Implementar el método
`tree<T>::iterator tree<T>::insert(tree<T>::iterator n, const T& x)`

[Ej. 2] [CLAS2 (W=20)]
Insistimos: **deben usar la interfaz avanzada (STL)**.

a) **[AB]** Par el TAD Arbol Binario (AB): declarar las clases `btree`, `cell`, `iterator`, incluyendo las declaraciones de datos miembros. Implementar el método
`btree<T>::iterator btree<T>::find(const T& x);`. Si utiliza alguna función auxiliar impleméntela.

b) **[set-list]** Escribir los siguientes métodos del **TAD conjunto** por listas ordenadas: `insert(x)`, `find(x)`, `clear()`.

c) **[set-vecbit]** Escribir las siguientes funciones para conjuntos representados con vectores de bit:
`void set_difference(vector<bool>&A, vector<bool>& B, vector<bool>& C);`
(Recordar que `set_difference(A,B,C)` hace `C=A-B`).

[Ej. 3] [PROG1 (W=40)]
Insistimos: **deben usar la interfaz avanzada (STL)**.

a) **[es-permutacion]** Una correspondencia es una *"permutacion"* si el conjunto de los elementos del dominio (las claves) es igual al del contradominio (los valores). De esta manera se puede interpretar a la correspondencia como una operación de permutación de las claves (de ahí su nombre). Por ejemplo
`M={1->5,3->7,9->9,7->1,5->3}` es una permutación ya que tanto las claves como los valores son el conjunto `{1,3,5,7,9}`.
Consigna: escribir una función predicado `bool es_permutacion(map<int,int> &M)` que retorna `true` si `M` es una permutación y `false` si no lo es.
Ayuda: Construir una correspondencia `M2` que mapea los valores del contradominio a las claves, es decir, para cada par de asignación `(k->v)` tal que `M[k] = v`, entonces se debe asignar el par `(v->k)` en `M2`.
Para que `M` sea una permutación el conjunto de las claves de `M` debe ser igual al conjunto de las claves de `M2`.

b) **[check-ordprev]**
Escribir una función **bool check_ordprev(tree<int> &T,list<int> &L);** que, dado un árbol ordenado orientado **T** retorna **true** si la lista **L** contiene al listado en orden previo de **T**. Por ejemplo, si **T=(3 (4 2 1) 0 (6 7 (8 9 5)))** y **L={3,4,2,1,0,6,7,8,9,5}** entonces **check_ordprev** debe retornar **true** y si por ejemplo **L={3,4,2,1,0,6,7,8,9,5,1000}** o **L={3,4,2,1,0,6,7000,8,9,5}** **check_ordprev** debe retornar **false**. Una manera simple de hacerlo es iterar en el árbol en la manera habitual e ir sacando los elementos de la lista **L** si coinciden con el contenido del nodo o posición actual. Una vez que se recorrió todo el árbol se verifica en la función *"wrapper"* que la lista haya quedado vacía. Se debe retornar **true** si este es el caso. La función **check_ordprev** está definida tal que:

- si el árbol es vacío y la lista no lo es (o viceversa) **check_ordprev** es **false**,
- si lo anterior no ocurre y si el árbol es vacío **check_ordprev** es **true**,
- si no ocurre lo anterior y los contenidos del nodo y la posición de la lista actual no coinciden **check_ordprev** es **false**,
- si no ocurre lo anterior, elimino el elemento actual de la lista,
- llamo recursivamente a la función verificando que no haya llegado al fin de lista.

[Ej. 4] [PROG2 (W=40)]
Insistimos: **deben usar la interfaz avanzada (STL)**.

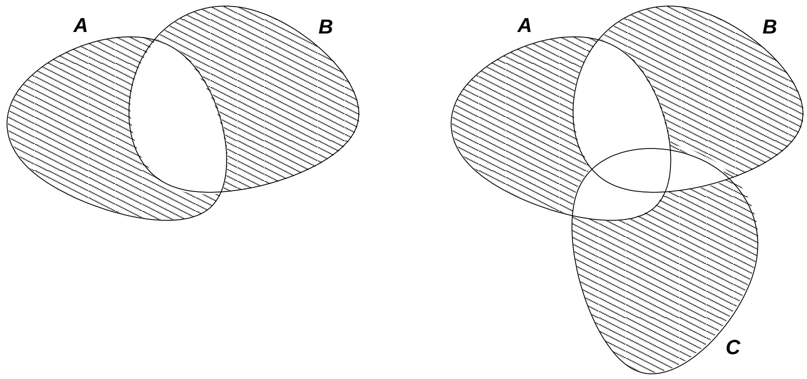
a) **[is-balanced]** Un árbol binario (AB) es balanceado si

- Es el árbol vacío ó,
- Sus subárboles derecho e izquierdo son balanceados, y sus alturas difieren a lo sumo en 1, o sea $|h_L - h_R| \leq 1$.

Por ejemplo **(1 (2 (3 (4 5 6) 7) (13 8 9)) (15 10 (16 11 12)))** es un árbol balanceado (notar que no necesariamente las profundidades de las hojas difieren en ± 1).
Consigna: Escribir una función **bool is_balanced(btree<int> &T);** que retorna **true** si el árbol está balanceado.
Ayuda: En la función auxiliar retornar el resultado de si el árbol es balanceado o no, pero además retornar (con un valor pasado por referencia, o con un **pair**) la altura del árbol correspondiente.

b) **[diff-sym]** Para dos conjuntos A, B, la *"diferencia simétrica"* se define como

$$\begin{aligned}\text{diff\_sym}(A, B) &= (A - B) \cup (B - A), \quad \text{o también} \\ &= (A \cup B) - (A \cap B)\end{aligned}$$



En general, definimos la diferencia simétrica de varios conjuntos como el conjunto de todos los elementos que pertenecen a uno y sólo uno de los conjuntos. En las figuras vemos en sombreado la diferencia simétrica para dos y tres conjuntos. Por ejemplo, si $A = \{1, 2, 5\}$, $B = \{2, 3, 6\}$ y $C = \{4, 6, 9\}$ entonces $\text{diff\_sym}(A, B, C) = \{1, 3, 4, 5, 9\}$.
Consigna: Escribir una función **void diff_sym(list<set<intd> > &l,set<int> &s);** que retorna en **s** la diferencia simétrica de los conjuntos en **l**.

Ayuda: Notar que en el caso de tres conjuntos si $S = \mathrm{diff\_sym}(A, B)$ y $U = A \cup B$, entonces $\mathrm{diff\_sym}(A, B, C) = (S - C) \cup (C - U)$. Esto vale en general para cualquier número de conjuntos, de manera que podemos utilizar el siguiente lazo

$l =$ lista de conjuntos, $S = \emptyset, U = \emptyset$;
for Q en la lista de conjuntos l **do**
  $S = (S - Q) \cup (Q - U)$;
  $U = U \cup Q$;
end for

Al terminar el lazo, S es la diferencia simétrica buscada.
Nota: Recordar que para las operaciones binarias (por ej. `set_union(A,B,C);`), los conjuntos deben ser distintos, es decir no se puede hacer `set_union(A,B,A);`, en ese caso debe hacerse `set_union(A,B,tmp); tmp=A;`
Nota: Para usar las operaciones binarias se puede usar tanto la versión simplificada que hemos utilizado en el curso, `void set_union(set &A,set &B,set &C);` como la versión de las STL `set_union(A.begin(),A.end(),B.begin(),B.end(),inserter(C,C.begin()));`

[Ej. 5] [OPER1 (W=20)]

a) **[rec-arbol (W=10)]** Dibujar el AOO cuyos nodos, listados en orden previo y posterior son
- `ORD_PRE={Y,W,Q,S,P,O,B,V}`,
- `ORD_POST={W,S,V,B,O,P,Q,Y}`.

b) **[particionar (W=10)]**. Considerando el árbol `(Y (S P Q) (B (A (C D (E X)))))` decir cuál son los nodos `descendientes(S)`, `antecesores(S)`, `izquierda(S)` y `derecha(S)`.
(**Nota:** se refiere a antecesores y descendientes **propios**).

[Ej. 6] [OPER2 (W=20)]

a) **[huffman (W=5)]** Dados los caracteres siguientes con sus correspondientes probabilidades, contruir el código binario utilizando el algoritmo de Hufmann y encodar la palabra `MANDELA`.
$P(M) = 0.10, P(N) = 0.10, P(L) = 0.10, P(D) = 0.30, P(A) = 0.05, P(E) = 0.3, (C) = 0.05$
Calcular la **longitud promedio** del código obtenido.

b) **[abb (W=5)]** Dados los enteros $\{14, 8, 21, 3, 4, 11, 6, 5, 13, 2\}$ insertarlos, en ese orden, en un *"árbol binario de búsqueda"*. Mostrar las operaciones necesarias para eliminar los elementos 14, 8 y 3 en ese orden.

c) **[hash-dict (W=5)]** Insertar los números $\{2, 15, 25, 8, 7, 35, 17, 4\}$ en una tabla de dispersión cerrada con $B = 10$ cubetas, con función de dispersión $h(x) = x$.

d) **[heap-sort (W=5)]** Dados los enteros $\{12, 15, 5, 4, 10, 7, 3\}$ ordenarlos por el método de *"montículos"* (*"heap-sort"*). Mostrar el montículo (minimal) antes y después de cada inserción/supresión.

[Ej. 7] [PREG1 (W=20)]

a) Ordenar las siguientes funciones por tiempo de ejecución. Además, para cada una de las funciones $T_1, \ldots, T_5$ determinar su velocidad de crecimiento (expresarlo con la notación $O(\,\cdot\,)$).

$$
\begin{aligned}
T_1 &= \sqrt{n} + 3\log_{10} n + 4\,n^2 + 2\,n^5,\\
T_2 &= \sqrt{5}\cdot n + 3\cdot 4^n + \log_2 n,\\
T_3 &= 3n^5 + 6\cdot 2^n + 5\,n!.\\
T_4 &= 3\log_2 n + 3^5 + 125.\\
T_5 &= 20^2 + 5n^2 + 2\cdot 3^n,
\end{aligned}
$$

b) ¿Porqué se dice que la pila es una estructura *FIFO ("First in, First Out")*? ¿Porqué se dice que la cola es una estructura *LIFO ("Last In, First Out")*?

c) Cual es la complejidad algorítmica (mejor/promedio/peor) de la función **`lower_bound()`** para correspondencias implementadas por
   1) listas ordenadas,
   2) vectores ordenados.

d) Considerando la implementación de pilas con **listas simplemente enlazadas**. ¿Cuál es la diferencia entre elegir como tope de la pila el comienzo o fin de la lista?

e) ¿Como se define la **altura** de un nodo en un árbol? ¿Cuál es la altura de los nodos **`A`**, **`B`**, y **`C`** en **`(W X Y (A B (C (D  F G H))))`**?

[Ej. 8] [PREG2 (W=20)]

a) Exprese como se calcula la longitud promedio de un código de Huffman en función de las probabilidades de cada unos de los caracteres P_i, de la longitud de cada caracter L_i para un número N_c de caracteres a codificar.

b) Escriba el código para ordenar un vector de enteros **`v`** por **valor absoluto**, es decir escriba la función de comparación correspondiente y la llamada a **`sort()`**.
Nota: recordar que la llamada a **`sort()`** es de la forma **`sort(p,q,comp)`** donde **`[p,q)`** es el rango de iteradores a ordenar y **`comp(x,y)`** es la función de comparación.

c) Comente ventajas y desventajas de las **tablas de dispersión abiertas y cerradas**.

d) Se quiere representar el conjunto de enteros múltiplos de 3 entre 30 y 99 (o sea $U = \{30, 33, 36, ..., 99\}$) por **vectores de bits**, escribir las funciones **`indx()`** y **`element()`** correspondientes.

e) Discuta la complejidad algorítmica de las operaciones binarias **`set_union(A,B,C)`**, **`set_intersection(A,B,C)`**, y **`set_difference(A,B,C)`** para conjuntos implementados por vectores de bits, donde **`A`**, **`B`**, y **`C`** son subconjuntos de tamaño n_A, n_B, y n_C respectivamente, de un conjunto universal **`U`** de tamaño N.