Apellido y Nombre: _____

Carrera: _____ DNI: _____

*[Llenar con letra mayúscula de imprenta GRANDE]*

Universidad Nacional del Litoral
Facultad de Ingeniera y Ciencias Hdricas
Departamento de Informtica
Algoritmos y Estructuras de Datos

# Algoritmos y Estructuras de Datos.
# Recuperatorio. [2012-11-29]

**ATENCIÓN (1):** Para aprobar deben obtener un **puntaje mínimo** de

- **50 %** en clases (Ej 1),
- **50 %** en programación (Ej 2),
- **50 %** en operativos (Ej 3) y
- **60 %** sobre las preguntas de teoría (Ej 4).

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

**[Ej. 1] [CLASES-P1 (30pt)]**

*a*) 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)***, ***list()***, ***begin()***, ***end()***.

*b*) 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.

**[Ej. 2] [CLASES-P2 (20pt)]**

*a*) **AOO:** 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)***

*b*) **AB:** declarar las clases ***btree***, ***cell***, ***iterator***, (preferentemente respetando el anidamiento), incluyendo las declaraciones de datos miembros. Implementar el método
***btree<T>::iterator btree<T>::erase(btree<T>::iterator n);***

**[Ej. 3] [CLASES-P3 (20pt)]**

*a*) Implementar ***bool bst_erase(btree<T> t,T x,bool (*less)(T,T))*** que elimina el elemento ***x*** del ABB T, retornando true si la eliminación fue exitosa, y false caso contrario.

*b*) Implementar una función ***pair<int,bool> oht_insert(vector<list<T> >& table, unsigned int (*hashfunc)(T),bool (*equal)(T,T),T x)*** que inserta el elemento ***x*** en la tabla de dispersión abierta ***table*** utilizando la función de dispersión ***hashfunc*** y la relación de equivalencia ***equal*** retornando un par de ***int*** (que indica la cubeta) y ***bool*** que indica si la inserción fue exitosa.

*c*) Implementar una función ***void vecbit_diffsym(vector<bool>&, A, vector<bool>& B, vector<bool>& C);*** que devuelve la **diferencia simétrica** definida como $C = (A - B) \cup (B - A) = (A \cup B) - (A \cap B)$ con ***A,B,C*** vectores de bits que representan conjuntos de un rango contiguo de enteros ***[0, N)***, donde ***N*** es el tamaño de los vectores ***A,B,C*** (asumir el mismo tamaño para los tres, es decir, haciendo ***int N=A.size();*** es suficiente).

**[Ej. 4] [PROG-P1 (50pt)]**

*a*) **[ascendente (25pt)]** En ciertas aplicaciones interesa detectar 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 $a_i$, $a_{i+1}$, ..., $a_{i+k}$ de modo tal que $a_j \leq a_{j+1}$ para $j = i, ..., i+k-1$, y es máxima en el sentido que si $i > 0$ entonces $a_{i-1} > a_i$, y si $i + k < n$ entonces $a_{i+k} > a_{i+k+1}$. Por ejemplo, si $n = 10$ y la lista es L = [0,5,5,9,4,3,9,6,5,5,2], entonces hay $h = 6$ corridas ascendentes, a saber, [0,5,5,9], [4], [3,9], [6], [5,5] y [2]. Usando las operaciones del TAD lista, escribir una función
***int ascendente(list<t> &l,vector< list<t> > &v)*** en la cual, dada una lista de enteros ***l***, almacene cada carrera ascendente como una componente del vector de listas ***vector<list <t> > &v***, devolviendo además el número $h$ de carreras ascendentes halladas.

*b*) **[concatmap (25pt)]** *Consigna:* Escribir una función
***`void concat_map(map<int,list<int> >& M, list<int>& L);`*** tal que reemplaza los elementos de ***`L`*** por su imagen en ***`M`***. Si un elemento de ***`L`*** no es clave de ***`M`*** entonces se asume que su imagen es la lista vacía.
*Por ejemplo:* Si
***`M={5->(3,2,5),2->(4,1),7->(2,1,1)}`*** y ***`L=(1,5,7,2,3)`***, entonces debe quedar
***`L=(3,2,5,2,1,1,4,1)`***.
*Restricciones:* El programa no debe usar contenedores auxiliares.

**[Ej. 5] [PROG-P2 (40pt)]**

*a*) **[ord-nodo (20pt)]**. Escribir una función predicado ***`bool ordnodo(tree<int> &A);`*** que verifica si cada secuencia de hermanos del subárbol del nodo ***`n`*** (perteneciente al árbol ordenado orientado ***`A`***) estan ordenadas entre sí, de izquierda a derecha. Por ejemplo, para el árbol ***`(3 5 (6 1 3) (7 4 5))`*** debería retornar ***`true`***, mientras que para ***`(3 9 (6 1 3) (7 4 2))`*** debería retornar ***`false`***, ya que las secuencias de hermanos ***`(9 6 7)`*** y ***`(4 2)`*** NO están ordenados. Se sugiere el siguiente algoritmo: para un dado nodo retornar false si: 1) sus hijos no estan ordenados o 2) algunos de sus hijos contiene en su subárbol una secuencia de hermanos no ordenada (recursividad).

*b*) **[depth-if (20pt)]**
Dado un AB ***`T`*** encontrar, la profundidad máxima de los nodos que satisfacen un cierto predicado (*profundidad condicionada*). Por ejemplo, si ***`T=(6 (7 9 (3 1)) 2)`***, entonces ***`depth_if(T,even)`*** debe retornar 1, ya que el nodo par de mayor profundidad es 2, mientras que ***`depth_if(T,odd)`*** debe retornar 3, ya que el nodo impar a máxima profundidad es 1.
*Consigna:* Escribir una función ***`int depth_if(btree<int> &T,bool (*pred)(int));`*** que realiza la tarea indicada.
*Ayuda:* La función recursiva auxiliar debe retornar la máxima profundidad de un nodo que satisface el predicado o -1 si el árbol está vacío o ningún nodo satisface el predicado. Entonces, dadas las profundidades condicionadas $d_l$, $d_r$ de los hijos la profundidad se puede expresar en forma recursiva como

$$\mathrm{depth}(n) = \begin{cases} -1, & \text{si } n \text{ es } \Lambda, \\ \max(d_r, d_l) + 1, & \text{si } \max(d_r, d_l) \geq 0, \\ 0, & \text{si } n \text{ satisface el predicado,} \\ -1, & \text{si } n \text{ no satisface el predicado.} \end{cases}$$

**[Ej. 6] [PROG-P3 (40pt)]**

*a*) **[is-hamilt (20pt)]** Dado un grafo ***`map<int, set<int> >G`*** y una lista de vértices ***`list<int> L`*** determinar si ***`L`*** es un **camino hamiltoniano** en ***`G`***.
**Ayuda:** Mantener un ***`set<int> visited`*** con todos los vértices visitados. Para dos enteros ***`x`***, ***`y`*** sucesivos en la lista ***`L`***, verificar si son vértices de ***`G`*** y si son adyacentes. Verificar que el nuevo vértice ***`y`*** no fuera visitado previamente e insertarlo en ***`visited`***. Al final, chequear que todos los vértices fueron visitados.
**Nota:** Recordamos que un camino Hamiltoniano en un grafo es un camino en el mismo que pasa por todos los nodos sin repetir ninguno. Por ejemplo en el grafo de arriba el camino ***`{0,1,2,3,4,5,6}`*** es Hamiltoniano.

*b*) **[en-todos (20pt)]**
Escribir una función predicado ***`bool en_todos(vector< set<int> > &v);`*** que retorna verdadero si existe al menos un elemento que pertenece a todos los conjuntos ***`v[j]`***. Por ejemplo, si

$$\mathrm{v[0]} = \{0,2,3,4,5\},\ \ \mathrm{v[1]} = \{0,1,5,7\},\ \ \mathrm{v[2]} = \{2,3,5,6,7\} \tag{1}$$

entonces ***`en_todos(v)`*** debe retornar ***`true`*** ya que 5 está en los tres conjuntos. Por el contrario, si

$$\mathrm{v[0]} = \{0,2,3,4,5\},\ \ \mathrm{v[1]} = \{0,1,7\},\ \ \mathrm{v[2]} = \{2,3,5,6,7\} \tag{2}$$

entonces ***`en_todos(v)`*** debe retornar ***`false`***.
*Sugerencia:* generar el conjunto que es la intersección de todos los ***`v[j]`*** y finalmente verificar si es vacío o no.

**[Ej. 7] [OPER-P2 (20pt)]**

*a*) **[rec-arbol (7pt)]** Dibujar el AOO cuyos nodos, listados en orden previo y posterior son
- ***ORD_PRE***=$\{A, R, B, O, L, I, T, O, \}$,
- ***ORD_POST***=$\{R, O, O, T, I, L, B, A, \}$,

*b*) **[huffman (7pt)]** Dados los caracteres siguientes con sus correspondientes probabilidades, contruir el código binario utilizando el algoritmo de Hufmann y encodar la palabra ***BUITRE***
$P(B) = 0.10, P(M) = 0.10, P(U) = 0.10, P(E) = 0.30, P(I) = 0.05, P(T) = 0.3, (R) = 0.05$ Calcular la longitud promedio del código obtenido.

*c*) **[hf-decode (6pt)]** Utilizando el código del siguiente árbol binario completo ***(F (L (* R N) (* D O)))*** a derecha desencodar el mensaje `10000001000010000`

**[Ej. 8] [OPER-P3 (20pt)]**

*a*) **[abb (5 pts)]** Dados los enteros $\{15, 9, 22, 4, 5, 12, 7, 6, 3, 14\}$ insertarlos, en ese orden, en un *"árbol binario de búsqueda"*. Mostrar las operaciones necesarias para eliminar los elementos 15, 7 y 4 en ese orden.
*b*) **[hash-dict (5 pts)]** Insertar los números $0, 13, 23, 6, 5, 33, 15, 2, 25$ en una tabla de dispersión cerrada con $B = 8$ cubetas, con función de dispersión $h(x) = x \bmod 9$ y estrategia de redispersión lineal.
*c*) **[heap-sort (5 pts)]** Dados los enteros $\{10, 13, 3, 2, 8, 5, 1\}$ ordenarlos por el método de *"montículos"* (*"heap-sort"*). Mostrar el montículo (minimal) antes y después de cada inserción/supresión.
*d*) **[quick-sort (5 pts)]** Dados los enteros $\{8, 12, 7, 4, 8, 13, 11, 6, 6, 5, 14, 9\}$ ordenarlos por el método de *"clasficación rápida"* (*"quick-sort"*). En cada iteración indicar el pivote y mostrar el resultado de la partición. Utilizar la estrategia de elección del pivote discutida en el curso, a saber el mayor de los dos primeros elementos distintos.

**[Ej. 9] [PREG-P1 (20pt)]**

*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 $O(\,\cdot\,)$).

$$T_1 = 5 \cdot 2^n + 2n^3 + 3\,n!+$$
$$T_2 = 2 \cdot 10^n + \sqrt{3} \cdot n + \log_8 n+$$
$$T_3 = n^2 + 5 \cdot 3^n + 4^{10}$$
$$T_4 = 2.3 \log_8 n + \sqrt{n} + 5\,n^2 + 2\,n^5$$
$$T_5 = 1000 + 3 \log_4 n + 8^2 + 5 \log_{10} n$$

*b*) ¿Cuáles son los tiempos de ejecución para los diferentes métodos de la clase ***map<>*** implementada con **listas desordenadas** en el caso promedio?
*Métodos:* ***find(key)***, ***M[key]***, ***erase(key)***, ***erase(p)***, ***begin()***, ***end()***, ***clear()***).

*c*) ¿Porqué decimos que $(n+1)^2 = O(n^2)$ si en realidad es siempre verdad que $(n+1)^2 > n^2$?
*d*) Discuta las ventajas y desventajas de utilizar listas doblemente enlazadas con respecto a las simplemente enlazadas.
*e*) ¿Qué ventajas o desventajas tendría implementar la clase *pila* en términos de **lista simplemente enlazada** poniendo el tope de la pila en el fin de la lista?

**[Ej. 10] [PREG-P2 (20pt)]**

*a*) Defina que es un **camino** en un árbol. Dado el AOO ***(a b (c e (f g)) d))***, cuáles de los siguientes son caminos?

1) ***(a,c,f,g)***,
2) ***(e,c,f,g)***,
3) ***(e,c,a)***,
4) ***(c,f,g)***.

*b*) Defina qué es un **AB lleno**. ¿Cuál es el **número de nodos** en el nivel $l$ de un árbol lleno? ¿Cuál es el **número total de nodos** en un árbol lleno de $n$ niveles? ¿Cuantas **hojas** contiene?

*c*) ¿Es verdad que si dos nodos están en el **mismo nivel** de un árbol, entonces son **hermanos**? ¿Y la recíproca? De ejemplos.

*d*) ¿Es posible **insertar** en una posición **no-dereferenciable** ($\Lambda$) en un AB? ¿Y en un AOO? Discuta y de ejemplos.

*e*) ¿Porqué el AB correspondiente a la **codificación binaria** de una serie de caracteres debe ser un **ABC (AB completo)**?

**[Ej. 11] [PREG-P3 (20pt)]**

*a*) 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.

*b*) Cual es el tiempo de ejecución en el caso **promedio** para el método ***insert(x)*** de la clase **diccionario** (***hash_set***) implementado por **tablas de dispersión cerradas**, en función de la ***tasa de llenado*** $\alpha = n/B$, para el caso de inserción exitosa y no exitosa.

*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 $N$.