Apellido y Nombre: _____

Carrera: _____ DNI: _____

[Llenar con letra mayúscula de imprenta GRANDE]

Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática
Algoritmos y Estructuras de Datos

Algoritmos y Estructuras de Datos. Recuperatorio. [2010-12-02]

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

- **[CLASES-P1]**

  1. 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()`.
  2. Escribir la implementación en `C++` de los métodos `push`, y `pop` de los **TAD pila y cola** (clases `stack` y `queue`).
  3. Escribir la implementación en `C++` del método `find(x)` para el **TAD correspondencia** (clase `map`) implementado con los listas ordenadas.

- **[CLASES-P2]**

  1. **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>::erase(tree<T>::iterator n);`
  2. **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>::insert(btree<T>::iterator n, const T& x)`

- **[CLASES-P3]**

  1. Implementar `set::iterator set::find(T x)` para conjuntos implementados por ABB. **No es necesario** escribir las declaraciones auxiliares de los miembros privados de la clase.
  2. Implementar `int set::erase(T x)` para conjuntos implementados por listas ordenadas. Si se utiliza algún método auxiliar, también implementarlo. **No es necesario** escribir las declaraciones auxiliares de los miembros privados de la clase.
  3. Escribir la función de ordenamiento por fusión `void merge_sort(list<int> &L)` para listas de enteros. Los enteros se comparan por el operador `<`.

- **[PROG-P1]**

  1. [max-dev-n] Dada una secuencia de números $\{a_1, a_2, ..., a_n\}$, vamos a decir que su *"máxima desviación"*, es la máxima diferencia (en valor absoluto) entre todos sus números: $\text{max\_dev}(a_1, a_2, ..., a_n) = (\text{máx}_{j=1}^{n} a_j) - (\text{mín}_{j=1}^{n} a_j)$. Escribir una función `int max_dev_m(list<int> &l, int m);` que retorna el máximo de las máximas desviaciones de las subsecuencias de L de longitud m, es decir

  $$\begin{aligned}\text{max\_dev\_m(L)} = \ &\text{máx}\{\text{max\_dev}(a_1, a_2, \ldots, a_m),\\ &\text{max\_dev}(a_2, a_3, \ldots, a_{m+1}), \text{max\_dev}(a_3, \ldots, a_{m+2}), \ldots, \text{max\_dev}(a_{n-m+1}, \ldots, a_n)\}\end{aligned} \quad (1)$$

  Por ejemplo, si `L=(1,3,5,4,3,5)`, entonces `max_dev_m(L,3)` debe retornar 4 ya que la máxima desviación se da en la primera subsecuencia `(1,3,5)` y es 4. Se sugiere el siguiente algoritmo, para cada posición `p` en la lista hallar la máxima desviación de los `m` elementos siguientes (incluyendo a `p`). Hallar la máxima de estas desviaciones.
  2. [chunk-revert] 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).

3. [cum-sum-pila] Escribir una función `void cum_sum(stack<int> &P)` que modifica a `P` dejando la suma acumulada de los elementos, es decir, si los elementos de `P` antes de llamar a `cum_sum(P)` son $(a_0, a_1, \ldots, a_{n-1})$, entonces después de llamar a `cum_sum(P)` debe quedar `P`=$(a_0, a_0 + a_1, \ldots, a_0 + a_1 + \ldots + a_n)$. Por ejemplo, si `P`=$(1, 3, 2, 4, 2)$ entonces después de hacer `cumsum(P)` debe quedar `P`=$(1, 4, 6, 10, 12)$. Usar una pila auxiliar.
Restricciones:
   - Usar la interfase STL para pilas (`clear()`, `top()`, `pop()`, `push(T x)`, `size()`, `empty()`).
   - No usar más estructuras auxiliares que la indicada ni otros algoritmos de STL.
   - El algoritmo debe ser $O(n)$.

- **[PROG-P2]**

1. **[check-ordprev]** (AOO)
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.

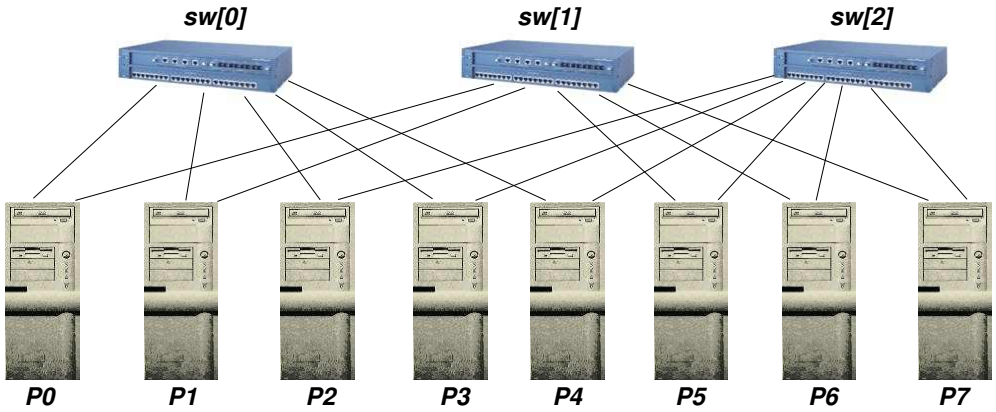
   Nota: se pueden usar todas las funciones de lista de STL sin restricciones.

2. **[depth-if]** (AB) 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 a 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

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

- **[PROG-P3]**

1. **[flat]** Se está diseñando una red inerconectada por switches y se desea, para reducir lo más posible la *latencia* entre nodos, que cada par de nodos esté conectado en forma directa por al menos un switch. Sabemos que el número de nodos es `n` y tenemos un `vector< set<int> > sw` que contiene para cada switch el conjunto de los nodos conectados por ese switch, es decir `sw[j]` es un conjunto de enteros que representa el conjunto de nodos inteconectados por el switch j.
Consigna: Escribir una función predicado `bool flat(vector< set<int> > &sw, int n);` que retorna verdadero si cada par de enteros (j, k) con $0 \leq j, k < n$ está contenido en al menos uno de los conjunto en `sw[]`.



En el ejemplo de la figura tenemos 8 nodos conectados via 3 switches y puede verficarse que cualquier par de nodos está conectado en forma directa a través de al menos un switch. Para este ejemplo el vector `sw` sería

$$\texttt{sw[0]} = \{0,1,2,3,4\}, \quad \texttt{sw[1]} = \{0,1,5,6,7\}, \quad \texttt{sw[2]} = \{2,3,4,5,6,7\} \tag{2}$$

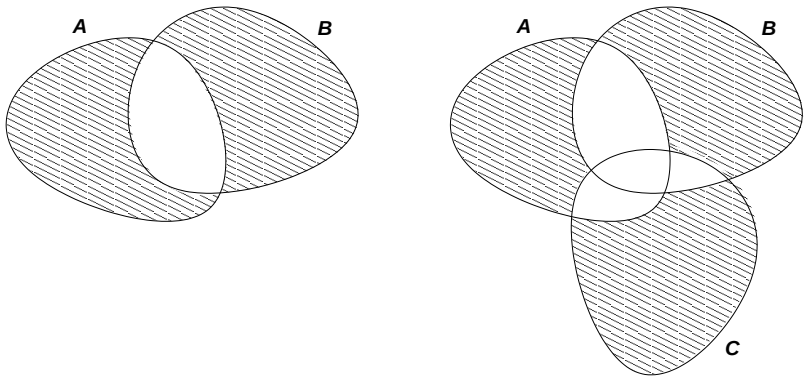
Por lo tanto `flat(sw,8)` debe retornar `true`. Por otra parte si tenemos

$$\texttt{sw[0]} = \{0,2,3,4\}, \quad \texttt{sw[1]} = \{0,1,5,7\}, \quad \texttt{sw[2]} = \{2,3,5,6,7\} \tag{3}$$

entonces los pares $(0,6)$, $(1,2)$, $(1,3)$, $(1,4)$, $(1,6)$, $(4,5)$, $(4,6)$ y $(4,7)$ no están conectados en forma directa y `flat(sw,8)` debe retornar `false`.
Sugerencia 1: Recorrer todos los pares de valores (j,k) y para cada par recorrer todos los conjuntos en `sw[]` hasta encontrar uno que contenga al par.
Sugerencia 2: Puede ser de ayuda el escribir una función auxiliar `bool estan_conectados(sw,j,k)`.

2. **[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<int> > &l,set<int>&s);` que retorna en `s` la diferencia simétrica de los conjuntos en `l`.
Ayuda: La solución se puede encarar con alguna de las dos estrategias siguientes:

a) Escribir una función `int cuenta(list<set<int> > &l,int x);` que retorna el número de conjuntos de `l` en los cuales x está incluido. Recorrer todos los elementos de todos los conjuntos de `l`, e insertar el elemento en `s` sólo si `cuenta` retorna exactamente 1.

b) Notar que en el caso de tres conjuntos si $S = \text{diff\_sym}(A, B)$ y $U = A \cup B$, entonces $\text{diff\_sym}(A, B, C) = (S - C) \cup (C - U)$. 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()));`

- [**OPER-P2**]

  1. [**huffman**] Dados los caracteres siguientes con sus correspondientes probabilidades, contruir el código binario y encodar la palabra `WIKILEAK`
     $P(W) = 0.1, P(I) = 0.1, P(K) = 0.3, P(L) = 0.1, P(E) = 0.2, P(A) = 0.05, P(Z) = 0.05, P(T) = 0.1$
     Calcular la longitud promedio del código obtenido. Justificar si cumple o no la condición de prefijos.
  2. [**rec-arbol**] Dibujar el árbol ordenado orientado cuyos nodos, listados en orden previo y posterior son
     - `ORD_PRE` $=\{Q, R, T, Y, L, W, A, B, C, Z, \}$,
     - `ORD_POST` $=\{R, L, A, B, C, W, Y, Z, T, Q, \}$,
  3. [**make-tree**] Escribir la secuencia de sentencias necesarias para construir el arbol binario `(5 (2 . 3) (7 6 8))`. (**Restricciones:** No usar `find()`).
  4. [**modif-tree**] Escribir la secuencia de secuencia necesarias para modificar el árbol ordenado orientado `A = (2 3 (5 6 7))` de manera que se convierta en `A = (2 (3 (5 6 7)))`, usando el método `splice()`.

- [**OPER-P3**]

  1. [**heap-sort**] Dados los enteros $\{2, 6, 9, 3, 4, 14, 5\}$ ordenarlos por el método de *"montículos"* (*"heap-sort"*). Mostrar el montículo (minimal) antes y después de **cada** inserción/supresión.
  2. [**quick-sort**] Dados los enteros $\{7, 11, 6, 3, 7, 12, 10, 5, 5, 4, 13, 8\}$ ordenarlos por el método de *"clasficación rápida"* (*"quick-sort"*). En cada iteración indicar el pivote y mostrar el resultado de la partición.
  3. [**abb**] Dados los enteros $\{12, 6, 19, 1, 2, 9, 4, 3, 0, 11\}$ insertarlos, en ese orden, en un *"árbol binario de búsqueda"*. Mostrar las operaciones necesarias para eliminar los elementos 12, 6 y 3 en ese orden.
  4. [**hash-dict**] Insertar los números $2, 15, 25, 8, 7, 35, 17, 4, 27$ en una tabla de dispersión cerrada con $B = 10$ cubetas, con función de dispersión $h(x) = x$ `mod` 10 y estrategia de redispersión lineal.

- **[PREG-P1]**

1. 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 $O(\cdot)$).

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

2. ¿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()`).

3. ¿Porqué decimos que $(n+1)^2 = O(n^2)$ si en realidad es siempre verdad que $(n+1)^2 > n^2$?
4. Discuta las ventajas y desventajas de utilizar listas doblemente enlazadas con respecto a las simplemente enlazadas.
5. ¿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?

- **[PREG-P2]**

1. Realice los pasos necesarios (sentencias de C/C++) para construir el siguiente árbol binario `T=(1 (3 . (2 10 20)) (5 7 .))`
2. Escriba la *"definición recursiva"* de la función ALTURA de un AOO.
3. Comente puntos a favor y puntos en contra del método de **Huffman** para la codificación de palabras/caracteres.
4. Explique porqué el método de Huffman cumple con la *"condición de prefijos"*.
5. Cual es el **orden del tiempo de ejecución** (en promedio) en función del número de elementos (n) que posee el **árbol ordenado orientado (AOO)** implementado con celdas encadenadas por punteros de las siguientes operaciones/funciones/métodos
   a) `*n`
   b) `begin()`
   c) `lchild()`
   d) `insert(p,x)`
   e) `operador++` (prefijo o postfijo)
   f) `erase(p)`
   g) `find(x)`

- **[PREG-P3]**

1. 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.
2. 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.
3. Comente ventajas y desventajas de las **tablas de dispersión abiertas y cerradas**.
4. 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.
5. 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.