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. [3 de Diciembre de 2009]

- [**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`, `pop`,`front` y `top` de los TAD pila y cola (clases `stack` y `queue`), según corresponda.

- [**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>::insert(tree<T>::iterator n, const T& x)`
  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>::erase(btree<T>::iterator n);`

- [**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. Implementar una función `bool openhashtable_insert(vector<list<T> >& table, unsigned int (*hashfunc)(T), T x)` que inserta el elemento `x` en la tabla de dispersión abierta `table` utilizando la función de dispersión `hashfunc` y retorna un booleano indicando si la inserción fue o no exitosa.
  4. implementar una función `void vecbit_difference(vector<bool>&, A, vector<bool>& B, vector<bool>& C);` que devuelve `C=A-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).

- [**PROG-P1**]

  1. [**junta**] Escribir una función `void junta(list<int> &L,int n);` que dada una lista `L`, agrupa de a `n` elementos dejando su suma. 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).
     Restricciones: **N**o usar ninguna estructura auxiliar. Prestar atención a no usar posiciones inválidas después de una supresión. El algoritmo debe tener un tiempo de ejecución $O(n)$, donde n es el número de elementos en la lista original.
  2. [**nilpot**]. Dadas dos correspondencias M_1 y M_2 la *"composición"* de ambas es la correspondencia $M = M_2 \circ M_1$ tal que si $M_1[a] = b$ y $M_2[b] = c$, entonces $M[a] = c$. Por ejemplo, si `M1={(0,1),(1,2),(2,0),(3,4),(4,3)}`, y `M2={(0,1),(1,0),(2,3),(3,4),(4,2)}`, entonces $M = M_1 \circ M_2$ `={(0,0),(1,3),(2,1),(3,2),(4,4)}`. Notemos que para que sea posible componer las dos correspondencias es necesario que los valores del contradominio de M_1 estén incluidos en las claves de M_2. Si el conjunto de valores del contradominio de una correspondencia M está incluido en el conjunto de sus claves, entonces podemos componer a M consigo misma, es decir $M^2 = M \circ M$. Por ejemplo, $M_1^2 = M_1 \circ M_1$`= {(0,2),(1,0),(2,1),(3,3),(4,4)}`. De la misma manera puede definirse, $M^3, ..., M^n$, componiendo sucesivamente. Puede demostrarse que, para algún n debe ser $M^n = I$,

donde I es la *"correspondencia identidad"*, es decir aquella tal que $I[x] = x$. Por ejemplo, si `M = {(0,1),(1,2),(2,0)}`, entonces para $n = 3$, $M^n = M^3 = I$.
Consigna: Escribir una función `int nilpot(map<int,int> &M);` que dada una correspondencia `M` retorna el mínimo entero `n` tal que $M^n = I$.
Sugerencia: Escribir dos funciones auxiliares:

- `void compose(map<int,int> &M1,map<int,int> &M2,map<int,int> &M);` que dadas dos correspondencias `M1, M2`, calcula la composición $M = M_2 \circ M_1$, devolviéndola en el argumento `M`,
- `bool is_identity(map<int,int> &M);` que dada una correspondencia `M`, retorna `true` si `M` es la identidad, y `false` en caso contrario.

- **[PROG-P2]**

1. **[path-of-largest]**
   Escribir una función `void path_of_largest(tree<int> &A,list<int> &L);` que, dado un árbol ordenado orientado `A`, retorna en `L` el camino que se obtiene recorriendo el árbol hacia abajo, *siempre por el hijo más grande*. Por ejemplo, si `A=(3 (4 6 (7 8 9)) (5 2 4 6))` entonces `path_of_largest` debe retornar `L=(3 5 6)`. Si para un nodo el valor más grande de los hijos está repetido, entonces el camino puede seguir por cualquiera de ellos. Por ejemplo si `A=(3 (4 5 6) 4 2)` entonces `path_of_largest()` puede retornar `L=(3 4 6)` o `L=(3 4)`.
2. **[cumsum-down]**
   El `cumsum(v)` de un vector `v` es la suma acumulada, es decir en la posicion `v[j]` debe quedar la suma de los elementos de `v[0..j]`. Para un árbol lo podemos extender diciendo que en cada nodo del árbol queda la suma de los valores de los nodos desde la raíz hasta el nodo en cuestión ANTES de la operación. Por ejemplo si `T=(1 (3 2 6) (5 4 8))` entonces después de `cumsum(T)` debe quedar `T=(1 (4 6 10) (6 10 14))`.
   Consigna: Escribir una función `void cumsum(btree<int> &T);` que realiza la tarea indicada para un **árbol binario (AB).**

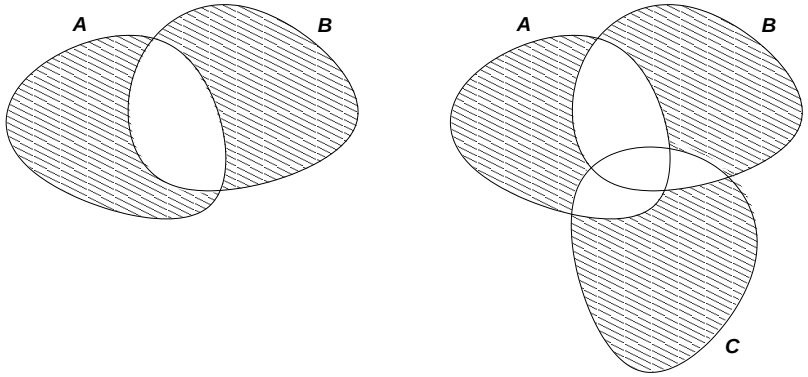
- **[PROG-P3]**

1. **[is-connected]**
   Se dice que un grafo `G` es **conexo** si para cada par de vértices `a`, `b` existe un camino que lleva desde un vértice al otro recorriendo las aristas del grafo.
   Consigna: Escribir una función bool `is_connected(map<int,set<int>> &G);` que determina si el grafo `G` es conexo o no.
   Ayuda: Escribir una función auxiliar `void connected_set(map<int,set<int>> &G,int x,set<int> &S);` que calcula la componente conexa `S` de `x`. Para ello recorre los vértices del grafo en forma *breadth-first* es decir usa una cola Q y ejecuta el siguiente algoritmo.

   `Q=` cola vacía, `S=`$\emptyset$;
   Pone `x` en `Q`
   while `Q` no esta vacía **do**
   Toma el primer elemento `y` de `Q`
   Pone `y` en `S`
   for `z` nodo vecino de `y` **do**
   Si `z` no fue visitado lo pone en `Q`.
   end for
   end while

   Cuando termina el lazo en `S` tenemos la componente conexa de `x` en `G`. Una vez escrita esta función sólo basta con tomar cualquier vértice de `G`, calcular su componente conexa y verificar si esta coincide con todo el conjunto de vértices del grafo o no.
2. **[diffsym]** Para dos conjuntos A, B, la *"diferencia simétrica"* se define como

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



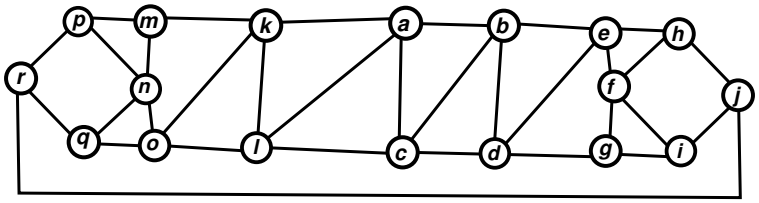
En general, definimos la diferencia simétrica de varios conjuntos como el conjunto de todos los elementos que pertenecen a uno y sólo uno de los conjuntos. En las figuras vemos en sombreado la diferencia simétrica para dos y tres conjuntos. Por ejemplo, si $A = \{1, 2, 5\}$, $B = \{2, 3, 6\}$ y $C = \{4, 6, 9\}$ 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: 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.

- [**OPER-P1**]

1. [**color-grafo**] Colorear el grafo de la figura usando el mínimo número de colores posible. Usar el algoritmo heurístico ávido siguiendo los nodos en el orden indicado $(a, b, c, ...)$. ¿La coloración obtenida es óptima? Justifique.



- [**OPER-P2**]

1. [**huffman**] Dados los caracteres siguientes con sus correspondientes probabilidades, contruir el código binario y encodar la palabra `PAPELERAS`
   $P(P) = 0.1, P(A) = 0.1, P(L) = 0.3, P(E) = 0.1, P(R) = 0.2, P(B) = 0.05, P(Q) = 0.05, P(S) = 0.1$
   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` $=\{Z, K, Y, W, Q, M, N, T, U, V\}$;
   - `ORD_POST` $=\{Y, W, K, M, V, U, T, N, Q, Z\}$.
3. [**make-tree**] Escribir la secuencia de sentencias necesarias para construir el arbol binario `(3 (4 . 2) (1 . 7))`. (*Restricciones:* No usar `find()`).
4. [**modif-tree**] Escribir la secuencia de secuencia necesarias para modificar el árbol ordenado orientado `A = (3 (4 5 6) (7 8 9))` de manera que se convierta en `A = (3 (4 5 6) (7 8 10 9))`. (Es decir, insertar el 10 entre el 8 y el 9). (*Restricciones:* No usar `find()`).

- [**OPER-P3**]

  - [**heap-sort**] Dados los enteros $\{0, 4, 7, 1, 2, 12, 9, 3\}$ ordenarlos por el método de *"montículos"* (*"heap-sort"*). Mostrar el montículo (minimal) antes y después de **cada** inserción/supresión.
  - [**quick-sort**] Dados los enteros $\{4, 8, 3, 0, 4, 9, 7, 2, 2, 1, 10, 5\}$ ordenarlos por el método de *"clasficación rápida"* (*"quick-sort"*). En cada iteración indicar el pivote y mostrar el resultado de la partición.
  - [**abb**] Dados los enteros $\{16, 10, 23, 5, 6, 13, 8, 7, 4, 15\}$ insertarlos, en ese orden, en un *"árbol binario de búsqueda"*. Mostrar las operaciones necesarias para eliminar los elementos 16, 10 y 7 en ese orden.
  - [**hash-dict**] Insertar los números $5, 18, 28, 11, 10, 38, 20, 7, 30$ 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. ¿Cuál es el tiempo de ejecución de `x=M[key]` para correspondencias implementadas con vectores ordenados?
2. ¿Cuál es el tiempo de ejecución para `L.insert(p,x)` en listas implementadas por arreglos?
3. Dado el siguiente fragmento de código

```
map<int,int> M;
M[0]=1;
int i = M[3];
```

   ¿Que valor obtiene `i`? ¿Se modifica `M`?
4. Dadas las funciones
   - $T_1(n) = 2n^2 + 1000\,n + \sqrt{5n} + 2^{20}$,
   - $T_2(n) = n! + 2n^5 + 5n^2 + 3n^{5/2}$,
   - $T_3(n) = 3n^2 + 2n^3 + n^{3/2} + 3\,\log n$,
   - $T_4(n) = 4n^{1.5} + 3n^{2.5} + n\log n + 6\sqrt{n}$.

   ordenarlas de menor a mayor.

$$T_{\square} < T_{\square} < T_{\square} < T_{\square}$$

- **[PREG-P2]**

1. Explique cual es la condición de códigos prefijos. De un ejemplo de códigos que cumplen con la condicion de prefijo y que no cumplen para un conjuntos de 3 caracteres.
2. Defina en forma recursiva el listado en orden previo y el listado en orden posterior de un arbol ordenado orientado con raiz `t` y sub-árboles hijos `h_1,h_2,...,h_n`.
3. Cual es el orden del tiempo de ejecución (en promedio) en función del número de elementos (n) que posee el árbol ordenado orientado implementado con celdas encadenadas por punteros de las siguientes operaciones/funciones/métodos
   - `insert(p,x)`
   - `begin()`
   - `lchild()`
   - `operador ++ (prefijo o postfijo)`
   - `find(x)`
   - `erase(p)`
   - `*n`
4. 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.
5. Para el arbol binario `(1 . (2 (3 5 .) 6))`
   - como queda el árbol y que sucede si hacemos

```
btree<int>::iterator p=T.begin();
p=T.erase(p.left());
```

   - y como queda el árbol asi hacemos por otro lado

```
btree<int>::iterator p=T.begin(),n;
n=p;
n=n.right();
p=T.splice(p.left(),n);
```

- **[PREG-P3]**

  1. ¿Cuál es el tiempo de ejecución de `find(x)` en el TAD diccionario por tablas de dispersión **abiertas**, en el caso promedio?
  2. ¿Cual es el tiempo de ejecución para `insert(x)` en el TAD conjunto por árbol binario de búsqueda, en el peor caso?
  3. ¿Cual es el tiempo de ejecución para `insert(x)` en el TAD conjunto por tabla de dispersión **cerrada**, en el caso promedio?
  4. ¿Cuál es el número de *intercambios* en el método de clasificación por selección?
  5. Recordemos que el procedimiento *"re-heap"* es áquel que, mediante una serie de intercambios restituye la propiedad de montículo a un árbol binario el cuál satisface la propiedad de parcialmente ordenado en todos sus nodos menos, eventualmente, en la raíz. ¿Cuál es el tiempo de ejecución de *re-heap*?