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. [2011-12-01]

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

1. **[CLASES-P1]**

   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.

2. **[CLASES-P2]**

   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);`

3. **[CLASES-P3]**

   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_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).

4. **[PROG-P1]**

   a) **[es-permutacion (25 puntos)]** 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}`. Para determinar si una correspondencia es una permutación (y sin usar contenedores de tipo *"conjunto"*, que serán vistos más adelante), se puede utilizar el siguiente procedimiento. 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`. Notar que si la correspondencia no es biunívoca, es decir si varias claves son asignadas a un mismo valor, entonces a `v` se le asignará alguna de esas claves. Cual de ellas es asignada dependerá del orden en que se recorren las asignaciones de `M`. Notar que si `M` es biunívoca, entonces `M2` es la inversa de `M`. Entonces, si la composición de `M` con `M2` es la identidad es decir para cada `k` en las claves de `M`, si `M2[M[k]] = k`, entonces `M` es una permutación. 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.
   b) **[list-sort (25 puntos)]** Escribir una función `void sort(list<int> &L);`, que ordena los elementos de `L` de menor a mayor. Para ello utilizar el siguiente algoritmo simple, utilizando una lista auxiliar `L2`: ir tomando el menor elemento de `L`, eliminarlo de `L` e insertarlo al final de `L2` hasta que `L` este vacía. Cual es el tiempo de ejecución en función de la cantidad de elementos `n` de `L`?

5. **[PROG-P2]**

   a) **[is-full (20pt)]** Recordemos que un árbol binario (AB) es *"lleno"* si todos sus *niveles están completos (atención, no confundir con árbol completo)*. Se puede definir en forma recursiva de la siguiente manera

   - El árbol vacío es lleno
   - Un árbol no vacío es lleno si sus dos hijos son llenos y tienen la misma altura.

   b) **[max-sons-count-node (20pt)]**
   Escribir una función `tree<int>::iterator max_sons_count_node(tree<int> &T, int &nsonmax);` que, dado un árbol ordenado orientado `T` retorna la posición (o nodo o iterator) del nodo cuya cantidad de hijos es la máxima y en `nsonmax` ese número máximo de hijos. Por ejemplo, si `T=(3 (4 2 10) (6 7 (8 9 5)) (11 12 13 14 15 16))`, entonces `max_sons_count_node` debe retornar la posición (o iterator) del nodo `11` y `nsonmax=5`. Si hay más de un nodo con la cantidad máxima de hijos, entonces puede retornar cualquiera de ellos.
   Sugerencia: dado un nodo debe ir recorriendo los hijos contándolos y al mismo tiempo ir reteniendo un iterator al nodo con mayor cantidad de hijos.

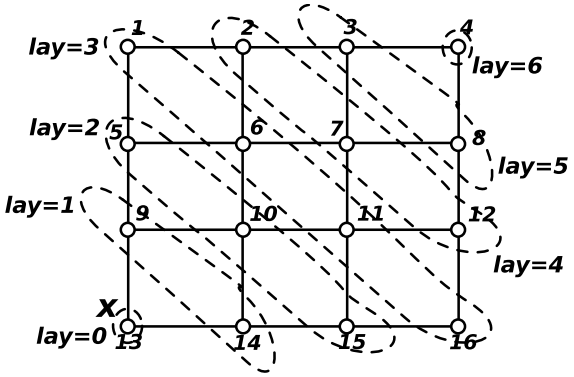
6. **[PROG-P3]**

   a) **[is-simple-graph (15 puntos)]**. Escribir una función `bool is_simple_graph(map<int,set<int> > &G)` que determinar si `G` es un grafo simple, es decir que no contiene blucles (`j->j`) y es no orientado (`i->j` $\iff$ `j->i`).

   b) **[mklayers (30 puntos)]** Escribir una función
   `void mklayers(vector<set<int> > &G, int x, vector<set<int> > &layers)` que dado un grafo `G` y un vértice de partida `x` determina la estructuras de capas de vecinos `layers` de `x` definida de la siguiente manera:

   - La capa 0 es el conjunto `layers[0]={x}` .
   - La capa 1 es el conjunto de los vecinos de `x`.
   - Para `l>1` la capa `l` es el conjunto de los vecinos de los nodos en la capa `l-1` que no están en capas anteriores (0 a `l-1`). Notar que en realidad sólo hace falta verificar que no estén en las capas `l-1` y `l-2`.

   Puede demostrarse que los vértices en la capa `l` son los que están a distancia `l` de `x`. Por ejemplo, dado el grafo de la figura, y partiendo del nodo `x=0` las capas son `layers[0]={0}`, `layers[1]={9,14}`, `layers[2]={5,10,15}`, `layers[3]={1,6,11,16}`, `layers[4]={2,7,12}`, `layers[5]={3,8}`, `layers[6]={4}`.



7. **[OPER-P2]**

   a) **[rec-arbol (10pt)]** Dibujar el AOO cuyos nodos, listados en orden previo y posterior son

   - `ORD_PRE` $= \{W, X, Y, A, B, C, D, F, G, H\}$;
   - `ORD_POST` $= \{X, Y, B, F, G, H, D, C, A, W\}$.

b) **[huffman (10pt)]** Dados los caracteres siguientes con sus correspondientes probabilidades, contruir el código binario utilizando el algoritmo de Hufmann y encodar la palabra `SUDAFRICA` $P(S) = 0.15, P(U) = 0.15, P(D) = 0.05, P(A) = 0.05, P(F) = 0.20, P(R) = 0.10, P(I) = 0.20, P(C) = 0.10$. Calcular la longitud promedio del código obtenido. Justificar si cumple o no la condición de prefijos.

8. **[OPER-P3]**

a) **[abb (5 pts)]** Dados los enteros $\{13, 7, 20, 2, 3, 10, 5, 4, 1, 12\}$ insertarlos, en ese orden, en un *"árbol binario de búsqueda"*. Mostrar las operaciones necesarias para eliminar los elementos 13, 5 y 2 en ese orden.
b) **[hash-dict (5 pts)]** Insertar los números $3, 16, 26, 9, 8, 36, 18, 5, 28$ 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 $\{1, 5, 8, 2, 3, 13, 10\}$ 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 $\{5, 9, 4, 1, 5, 10, 8, 3, 3, 2, 11, 6\}$ 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.

9. **[PREG-P1]**

a) Considere la función:

```
bool is_mapped(map<int,int> &M,int key,int val) {
  return /* Esta key -> val en M? ... */; }
```

que debe retornar `true` si el par de asignación `(key,val)` está en la correspondencia `M`. ¿Cuál es la expresión correcta que refina el seudocódigo?
b) ¿Cuál es el tiempo de ejecución de la función `find(key)` para correspondencias implementadas por vectores ordenados en función de n, el número de asignaciones en la correspondencia?
c) Dadas las funciones
- $T_1(n) = 3n^3 + 2n! + \log n$,
- $T_2(n) = 3 \cdot 2^3 + n^2 + n^{1.5}$,
- $T_3(n) = 5! + 6 \cdot 2^n + 20 \cdot n^2$ y
- $T_4(n) = 2^{10} + 20n + \log_2 40$

ordenarlas de menor a mayor orden de tiempo de ejecución.
d) Discuta si los algoritmos de búsqueda **exhaustiva** y **heurístico** para la coloración de grafos dan la solución **óptima** al problema. Señale la **complejidad algorítmica** de los mismos. ¿Cuál de los dos elegiría para un grafo con 100 vértices y 500 aristas? Justifique.
e) Si la correspondencia `M={(1->2),(5->8)}` y ejecutamos el código `int x= M[5]`. ¿Que ocurre? ¿Que valores toman `x` y `M`? ¿Y si hacemos `x = M[3]`?

10. **[PREG-P2]**

a) 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.
b) 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`.
c) 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`

d) 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.

e) 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);
```

11. **[PREG-P3]**

a) Defina la propiedad de **transitividad** para las relaciones de orden. Si $f(x)$ es una función sobre los reales y defino $a <_f b$ sii $f(a) < f(b)$, es $<_f$ transitiva?

b) Nombre diversas **relaciones de orden** que se pueden usar con los algoritmos de ordenamiento.

c) Discuta el valor de retorno de `insert(x)` para conjuntos.

d) Discuta el **número de intercambios** que requieren los algoritmos de ordenamiento lentos en el peor caso.

e) ¿Cuál es el costo de **inserción exitosa** en tablas de dispersión abiertas?