

## Algoritmos y Estructuras de Datos. 3er Parcial. [2010-11-25]

**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 (20pt)] Insistimos: deben usar la interfaz STL.**

- a) Escribir la función `btree<int>::iterator abb_find(btree<int> &T, int x)`; que devuelve el iterator en el ABB T, donde se encuentra el elemento x, o caso contrario T.end().  
**Restricción:** La complejidad algorítmica debe ser  $O(l)$  donde  $l$  es la profundidad del árbol.
- b) Escribir la función `pair<int,int> minmax(btree<int> &T)`; que devuelve el mínimo y el máximo de el ABB T. Si el árbol está vacío debe devolver las constantes (INT\_MAX, INT\_MIN).  
**Restricción:** La complejidad algorítmica debe ser  $O(l)$  donde  $l$  es la profundidad del árbol.
- c) Escribir una función `void sort(vector<int> &v)`; que ordena los elementos de v utilizando el operador < como función de comparación. Puede elegir cualquier algoritmo de ordenamiento rápido o lento visto en el curso. **Especifique cuál es el algoritmo que está usando.**  
**Restricción:** El algoritmo debe ser *in-place*, es decir no debe usar contenedores auxiliares.
- d) 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.

**[Ej. 2] [Programación (total = 40pt)] Insistimos: deben usar la interfaz STL.**

- a) **[ident-set (20pt)]**  
Dado una serie de  $n$  conjuntos distintos  $S_j$  para  $j = 0, \dots, n-1$ , y un conjunto  $S$  que es alguno de los  $S_j$  queremos identificar cuál de ellos es, es decir encontrar  $k$  tal que  $S_j = S$ .  
**Consigna:** Escribir una función `int ident_set(vector<set<int>> vecset, set<int> s)`; que retorna el índice  $k$  tal que `vecset[k]=s`.  
Para ello proponemos el siguiente algoritmo:
  - 1) Construimos un `vector<set<int>> vecset` tal que `vecset[j]=Sj`, y un vector de enteros `vector<int> indx(n)`, que inicialmente es la identidad `indx[j]=j`.
  - 2) Escribir una función `void split_vecset(vector<set<int>> &vecset, vector<int> &indx, int x)`; que, dado un entero x deja en vecset los conjuntos que contengan a x y en indx los índices correspondientes. Así por ejemplo si `vecset[0]=(0,1,3,8)`, `vecset[1]=(1,3,5,9)`, `vecset[2]=(2,4,7,8)`, y `indx=(1,5,6)` entonces después de hacer `split_vecset(vecset,indx,8)`; debe quedar `vecset[0]=(0,1,3,8)`, `vecset[1]=(2,4,7,8)`, y `indx=(1,6)`. En todo momento el tamaño de vecset y indx debe ser el mismo.
  - 3) Para cada  $x \in S$ , aplicar la función anterior `split_vecset` para extraer los conjuntos que contienen a ese  $x$  en particular, hasta que quede un solo conjunto. El índice correspondiente en indx es el  $k$  buscado.
- b) **[contract-edge (20pt)]** Escribir una función `void contract_edge(map<int, set<int> > &G, pair<int,int> &e)`; que remueve del grafo simple  $G$  la arista  $e$  y uno de sus vértices, dejando en  $G$  el vértice de menor etiqueta.

Sea  $e = (u, v)$  con  $u < v$ , entonces los pasos a realizar por la función son:

- 1) **Actualizar la lista de adyacencia** de  $u$  incorporando la adyacencia de  $v$  (tener cuidado de no agregar  $u$  en la nueva adyacencia de  $u$  para no generar lazos),
- 2) Para cada vértice  $x$  de la adyacencia de  $v$ , **remover  $v$  de la adyacencia de  $x$**
- 3) Para cada vértice  $x$  de la adyacencia de  $v$ , si  $x \neq u$  entonces **agregar  $u$  a la adyacencia de  $x$** ,
- 4) **Remover el vértice  $v$**  y su adyacencia del grafo  $G$ .

Ejemplo: Sea

```
G=[1 -> {2,3,4},  
   2 -> {1,3,5,6},  
   3 -> {1,2},  
   4 -> {1,5},  
   5 -> {2,4},  
   6 -> {2}]
```

y  $e=(1,2)$ . Entonces después de hacer `remove_edge(G,e)`;  $G$  tiene que quedar:

```
G=[1 -> {3,4,5,6},  
   3 -> {1},  
   4 -> {1,5},  
   5 -> {1,4},  
   6 -> {1}]
```

[Ej. 3] [operativos (total 20pt)]

- a) [abb (5 pts)] Dados los enteros  $\{12, 6, 19, 1, 2, 9, 4, 3, 2, 11\}$  insertarlos, en ese orden, en un **árbol binario de búsqueda (ABB)**. Mostrar las operaciones necesarias para eliminar los elementos 17, 9 y 6 en ese orden.
- b) [hash-dict (5 pts)] Insertar los números 1, 17, 27, 10, 9, 37, 19, 6 en una **tabla de dispersión cerrada** con  $B = 8$  cubetas, con función de dispersión  $h(x) = x \% 8$  y estrategia de redistribución lineal.
- c) [heap-sort (5 pts)] Dados los enteros  $\{2, 6, 9, 3, 4, 14, 11\}$  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  $\{4, 8, 3, 0, 4, 9, 7, 2, 2, 1, 10, 5\}$  ordenarlos por el método de **clasificación 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. 4] [Preguntas (total = 20pt, 4pt por pregunta)]

- 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  $U = \{30, 33, 36, \dots, 99\}$ ) 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  $N$ .