

Algoritmos y Estructuras de Datos.

TPL3R. Recuperatorio Trabajo Práctico de Laboratorio 3. [2014-11-06]

Ejercicios

[Ej. 1] **[only1]** Dado un vector de conjuntos **VS**, determinar el conjunto **S1** de todos aquellos elementos que están en uno y sólo uno de ellos. Por ejemplo, si
VS = [{0, 1, 2, 3}, {2, 3, 4, 5}], entonces **S1** = {0, 1, 4, 5}.

Consigna: Escribir una función `void only1(vector<set<int>> &VS, set<int> &S1);` **Ayuda:** Recorrer los elementos de **VS**. Mantener un conjunto **X** que contiene todos los elementos de la union de los **VS[j]** y otro **Xdup** que contienen los elementos duplicados. Para esto realiza el siguiente pseudocódigo

```
for S in VS {
    for x in S {
        si x esta en X agregarlo a Xdup;
        agregar x a X;
    }
}
```

Al final de este recorrido debemos tener en **X** todos los elementos de los **VS[j]** y en **Xdup** los duplicados. Sólo basta con hacer una operación de diferencia.

[Ej. 2] **[included]** Dada un vector de conjuntos `vector< set<int> > VS`, escribir una función predicado `bool included(vector< set<int> > &VS)`, que determina si los conjuntos del vector **VS** son subconjuntos propios en forma consecutiva. Es decir, si $S_j \subset S_{j+1}$ para $j = 0, \dots, n - 2$. (Recordar que $A \subset B$ indica inclusión propia, es decir $A \subseteq B$ y $A \neq B$.)

[Ej. 3] **[diffh]** Un árbol binario (AB) es balanceado si

- Es el árbol vacío ó,
- Sus subárboles derecho e izquierdo son balanceados, y sus alturas difieren a lo sumo en 1, o sea $|h_L - h_R| \leq 1$.

Por ejemplo (1 (2 (3 (4 5 6) 7) (13 8 9)) (15 10 (16 11 12))) es un árbol balanceado (notar que no necesariamente las profundidades de las hojas difieren en ± 1).

Consigna: Escribir una función `bool diffh(btrees<int> &T)`; que retorna `true` si el árbol está balanceado.

Ayuda: En la función auxiliar retornar el resultado de si el árbol es balanceado o no, pero además retornar (con un valor pasado por referencia, o con un `pair`) la altura del árbol correspondiente.

[Ej. 4] **[onechild]** Dado una árbol binario (AB) **T**, determinar cuantos nodos de **T** tienen exactamente un solo hijo (*single child count*).

Ejemplos:

Para **T** = (8 (7 9 2) (3 . (9 . 1))) ; debe retornar 2

Para **T** = (2 8 (1 (2 . 6) .)) ; debe retornar 2

Para **T** = (4 (9 (2 6 .) .) 8) ; debe retornar 2

Para **T** = (8 (6 (5 . (4 3 (4 . (9 . 6))) .) 3) ; debe retornar 4

Definición recursiva:

$$\text{onec}(n) = \begin{cases} 0; & \text{si } n = \Lambda, \\ s + \text{onec}(l) + \text{onec}(r); & \text{caso contrario,} \end{cases} \quad (1)$$

donde $s = 1$, si n tiene exactamente un sólo hijo y 0 en caso contrario. l, r son los hijos izquierdo y derecho de n .

Consigna: Escribir una función `int onechild(btree<int> &T)`; que realiza la tarea descripta.

Curiosidad: Coincide con la cantidad de puntos que aparecen en notación Lisp, pero no recomendamos encarar esa estrategia.

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más abajo; impleméntandolas en C++ de tal forma que el código que escriban **compile y corra correctamente**, es decir, no se aceptará un código que de algún error de compilación o que tire alguna excepción/señal de interrupción en runtime. Básicamente se hace una evaluación de caja negra, aunque le daremos un rápido vistazo al código.
- Pueden utilizar todas las funciones y utilidades del estándar de C++ que por supuesto contiene a la librería STL.
- Se incluye un template llamado `program.cpp`. En principio sólo tienen que escribir el cuerpo de las funciones pedidas. El paquete ya incluye el header `tree.h`.
- Para cada ejercicio hay dos funciones de evaluación, por ejemplo si `f` es la función a evaluar tenemos

```
ev.evalj(f, vrbs);
h1 = ev.evaljr(f, seed); // para SEED=123 debe dar H1=170
```

`j` es el número de ejercicio, por ejemplo para el ejercicio 1 tenemos las funciones (`eval1` y `eval1r`). La primera `ev.evalj(f, vrbs)`; toma una serie de casos de prueba de entrada, le aplica la función del usuario `f` y compara la salida del usuario (`user`) con respecto a la esperada (`ref`). Si la verbosidad (el argumento `vrbs`) se pone en uno, entonces la función evaluadora reporta por consola los datos de entrada, la salida de la función de usuario y la salida esperada

```
m: 10, k: 3
T(ref): (10 (7 (4 1) 1) (4 1) 1)
T(user): (10 (7 (4 1) 1) (4 1) 1)
EJ1|Caso0. Estado: OK
```

- La segunda función `evaljr` es el chequeo que llamamos **SEED/HASH**. La clase evaluadora genera una serie de contenedores a partir de la semilla `seed`, se los pasa a la función del usuario `f()`. Las respuestas de la `f()` van siendo procesadas por la función interna de hash que genera un **checksum H** de las respuestas. Por ejemplo para el primer ejercicio si `seed=123` entonces el checksum es `H=523`. Una vez que el alumno termina su tarea se le pedirá que corra la clase evaluadora con un valor determinado de la semilla `seed` y se comprobará que genere el valor correcto del checksum `H`.
Desde el punto de vista del alumno esto no trae ninguna complicación adicional, simplemente debe llenar el parámetro `seed` con el valor indicado por la cátedra, recompilar el programa y correrlo. La cátedra verificará el valor de salida de `H`.
- En la clase evaluadora cuentan con las siguientes funciones utilitarias:
 - `void dump(vector<set<int> > &VX, string s="")`: Imprime un mapa entero/entero. Nota: Es un método de la clase `Eval` es decir que hay que hacer `Eval ev; ev.dump(VX)`; . El string `s` es un label opcional.
 - Análogamente está `void dump(set<int> S, string s="")`.
 - `btree<int>::lisp_print()`: Lisp print de un árbol. Nota: esta pertenece a la clase `tree`. Uso: `btree<int> T; T.lisp_print()`;