Algoritmos y Estructuras de Datos.
Trabajo Práctico de Laboratorio 2. [2019-10-10]

PASSWD PARA EL ZIP: `R4FC J3PR VQM6`

Ejercicios

[Ej. 1] [fill-oprev] (AOO)
Dado un AOO `T` y una lista `L`, escribir una función `void fill_ordprev(tree<int> &T,list<int> &L);` que reemplaza los valores de `T` por los valores en `L`, en orden previo. Si la lista `L` tiene menos elementos que nodos en `T` entonces los nodos restantes de `T` quedan en su valor original. Recíprocamente, si `L` tiene más valores que `T` entonces los valores restantes de `L` son descartados.
Ejemplos:

```
L:[6,7,8],T:(1 2 3 4 5)          =>      T:(6 7 8 4 5),
L:[6,7,8,9,10,11],T:(1 2 3 4 5) =>      T:(6 7 8 9 10),
L:[6,7,8],T:(1 (2 3 4) 5)        =>      T:(6 (7 8 4) 5),
L:[6,7,8,9,10,11,12],T:(1 (2 3 4) 5) => T:(6 (7 8 9) 10),
```

Nota: la función puede ser *destructiva* sobre `L`.
Ayuda:

- Considerar una función recursiva.
- La función debe chequear si el nodo no es Λ y si la lista no está vacía.
- Extraer un valor de la lista y ponerlo en la raíz.
- Aplicar recursivamente a los hijos del árbol.

[Ej. 2] [a-lo-ancho] (grafos, AOO) Implemente la función `void ancho(graph& G, tree<char>& T)` que reciba un grafo no dirigido `G` y genere un árbol de expansión `T` de la componente conexa a la que pertenece el primer vértice del grafo (que será la raíz del árbol), a partir del algoritmo de búsqueda a lo ancho. Recordar que en la búsqueda en anchura o a lo ancho:

- Se toma el primer vértice de `G` como raíz del árbol `T`.
- Añadimos al árbol todos los vértices adyacentes (forman el nivel 1 del árbol) en orden alfabético (se garantiza que la lista de adyacentes está ordenada de este modo)
- Por cada vértice del nivel 1, añadimos a `T` todos los vértices incidentes en él, siempre que no formen ciclo (no hayan sido previamente visitados). Se conforma el nivel 2.
- Repetimos el procedimiento hasta que no haya vértices en el nivel `j` actual.

[Ej. 3] [intersect-map] (correspondencia) Implemente una función `void intersect_map(map<int,list<int» &A, map<int, list<int» &B,map<int, list<int» &C)` que a partir de las correspondencias `A` y `B` construye una correspondencia `C` de manera que las claves de `C` son la intersección de las claves de `A` y `B` y para cada clave `k` en `C` la imagen `C[k]` es la intersección ordenada de las listas ordenadas `A[k]` y `B[k]`. Si ésta intersección de listas es nula, no debe incluirse la entrada de clave `k` en `C`. **Ejemplo:**

```
A:[0->{}, 3->{0,1}],
B:[0->{0,1,2}, 1->{0,1,2,3}, 3->{0,2}]
=>
C:[3->{0}]
```

Instrucciones generales

- El examen consiste en que escriban las funciones descriptas más arriba; 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.
- Salvo indicación contraria 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.
- Para cada ejercicio hay dos funciones de evaluación, por ejemplo si **`f`** es la función a evaluar tenemos

  ```
  ev.eval<j>(f,vrbs);
  hj = ev.evalr<j>(f,seed); // para SEED=123 debe dar Hj=170
  ```

  `j` es el número de ejercicio, por ejemplo para el ejercicio 1 tenemos las funciones (**`eval<1>`** y **`evalr<1>`**). La primera **`ev.eval<j>(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
  ```

- **`ucase`**: Además las funciones **`eval()`** tienen dos parámetros adicionales:
  `Eval::eval(func_t func,int vrbs,int ucase);`
  El tercer argumento 'ucase' (caso pedido por el usuario), permite que el usuario seleccione uno solo de todos los ejercicios para chequear. Por defecto está en **`ucase=-1`** que quiere "hacer todos". Por ejemplo **`ev.eval4(prune_to_level,1,51);`** corre sólo el caso 51.
- **Archivo con casos tests JSON:** Los casos test que corre la función **`eval<j>`** están almacenados en un archivo **`test1.json`** o similar. Es un archivo con un formato bastante legible. Abajo hay un ejemplo. **`datain`** son los datos pasados a la función y **`output`** la salida producida por la función de usuario. **`ucase`** es el número de caso.

  ```
  { "datain": {
    "T1": "( 0 (1 2) (3 4 5 6) )",
    "T2": "( 0 (2 4) (6 8 10 12) )",
    "func": "doble" },
   "output": { "retval": true },
   "ucase": 0 },
  ```

- La segunda función **`evalr<j>`** 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 función **`evalr<j>()`** de la clase evaluadora con un valor determinado de la semilla **`seed`** y se comprobará que genere el valor correcto del checksum **`H`**.
- En la clase evaluadora cuentan con funciones utilitarias como por ejemplo:
  `void Eval::dump(list <int> &L,string s="")`: Imprime una lista de enteros por **`stdout`**. Nota: Es

un método de la clase **`Eval`** es decir que hay que hacer **`Eval::dump(VX);`**. El string **`s`** es un label opcional.

- **`void Eval::dump(list <int> &L,string s="")`**

- Después del parcial deben entregar el programa fuente (sólo el **`program.cpp`**) renombrado con su apellido y nombre (por ejemplo **`messilionel.cpp`**). Primero el apellido.