Algoritmos y Estructuras de Datos.
TPL3. Trabajo Práctico de Laboratorio 3. [2018-11-01]

PASSWD PARA EL ZIP: 8QGV JQHW XGP7

Ejercicios

ATENCION: Deben necesariamente usar la opción `-std=gnu++11` al compilador, **si no no va a compilar**.

[Ej. 1] **[make-full]** Implementar una función
`void make_full(btree<int> &T);` que elimina (in-place) los nodos interiores de un árbol binario que tienen un sólo hijo, de manera que el árbol resultante es un árbol binario lleno (Full Binary Tree). En el árbol resultante sólo deben quedar hojas o árboles interiores con dos hijos.
Ejemplos:

```
T(input): (5 . (7 . (3 2 4))) => T(output): (3 2 4)
T(input): (5 (6 8 9) .)       => T(output): (6 8 9)
T(input): (5 . (6 8 9))       => T(output): (6 8 9)
T(input): (1 (2 . 4) (3 . 5)) => T(output): (1 4 5)
```

Ayuda:
Escribir una función recursiva
`void make_full(btree<int> &T, btree<int>::iterator n);` que haga lleno el subárbol del nodo `n` de la siguiente manera:

- Sean `cl` y `cr` los hijos izquierdo y derecho del nodo `n`.
- Si solo uno de ellos (`cl` ó `cr`) es Λ:
  - Crear un árbol vacío `T2`.
  - Mover(`splice`) el subárbol de `cl`/`cr` a `T2`.
  - Eliminar el nodo `n` de `T`.
  - Mover(`splice`) `T2` de vuelta a `n`.
  - Aplicar recursivamente `make_full` al nuevo `it`
- Si no:
  - Si `cl` no es Λ, aplicar `make_full` recursivamente al subarbol izquierdo.
  - Si `cr` no es Λ, aplicar `make_full` recursivamente al subarbol derecho.

[Ej. 2] **[max-subtree]** Escribir una función,
`int max_subbtree(btree<int>&T);` que retorna la suma de etiquetas máxima de entre todos los posibles subárboles del árbol binario T.
Nota: Las etiquetas de los nodos pueden ser negativas y positivas (el subárbol vacío puede ser el de mayor suma, en este caso 0).
Ejemplos:

```
T=(1 2 3)                 => 6
T=(1 -1 1)                => 1
T=(1 -2 -4)               => 0
T=(-2 (2 2 -1) (2 3 -1))  => 5
T=(-2 (2 2 -1) (2 -3 -1)) => 3
```

Ayuda:

- Puede ser útil realizar una doble recursión.
- Escribir una función recursiva auxiliar que calcula la suma del subárbol de un nodo dato.
- En la función principal una primera recursión para recorrer todo el árbol, y una interna para obtener la suma de todo subárbol con raíz en el nodo que se está analizando.

[Ej. 3] **[most-connected]** Implementar una función `int most_connected(vector< set<int> > &VS);` que retorna el índice `j` tal que el conjunto `VS[j]` es el que está *conectado* con un mayor número de otros conjuntos de `VS`. Decimos que dos conjuntos están conectados si no son disjuntos, es decir, si tienen intersección común no vacía.
Nota: En el caso de haber varios conjuntos con el mismo número de conexiones debe retornar el primero de ellos.
Ejemplos:

```
VS=[{0},{1},{2},{0,1,2}],                       retorna 3
VS=[{0,1,2},{0},{1},{2}],                       retorna 0
VS=[{0,6,9},{5,6,9},{5},{1},{5,9},{5},{1,5,7}], retorna 1
```

Ayuda:

- Escribir una función auxiliar `bool connected(set<int> &s1,set<int> &s2);` que retorna `true` si los conjuntos no son disjuntos.
- Hacer un doble lazo sobre los conjuntos de `VS`.
- Para cada conjunto en la iteración exterior `VS[j]` determinar en el lazo interior cuantos conjuntos `VS[k]` están conectados.
- Mantener variables `jmax`, `nmax`, que contienen el índice correspondiente al máximo actual, y el número de conexiones del máximo actual.

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**.
  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 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.

mstorti@galileo/aed-3.1-302-gf1b8e1bc/Thu Nov 1 21:35:31 2018 -0300