Algoritmos y Estructuras de Datos.
Recuperatorio de Trabajos Prácticos de Laboratorio. [2017-11-09]

PASSWD PARA EL ZIP: T11A P23E L58D

Ejercicios

[Ej. 1] **[tpl1 - sum_sublist]**
Implemente la función `list<int> sum_sublist(list<int>& L,int S)` que dada una lista de enteros `L` y un entero `S`, encuentre y retorne una sublista cuya suma sea `S`. Si no existe ninguna sublista con dicha suma, retorne una lista vacía. En caso de haber varias listas que cumplan retorne la primera y la más corta.
Ejemplos:

- Para `L: [1,2,6,5,8,3,4,6]` y `S=13` debe retornar `[2,6,5]`
- Para `L: [1,2,6,5,8,3,4,6]` y `S=15` debe retornar `[8,3,4]`
- Para `L: [1,2,6,5,8,3,4,6]` y `S=17` debe retornar `[]`

[Ej. 2] **[tpl1 - discrete_moving_mean]** Implemente la función `list<int> discrete_moving_mean(list<int>& L, int w)` que dada una lista de enteros `L` y un entero `w`, retorne una lista con los valores de la media móvil con ventana fija de tamaño `w`. El primer elemento será el promedio (en división entera) de los primeros `w` elementos de `L`; el segundo será el promedio desde el `2` elemento al `w+1`; y en general, el elemento en la posición `N` de la lista resultado, será el promedio entre los elementos `[N,w+N)` de `L`.
Ejemplos:

- Para `L: [1,2,6,5,8,3,4,6]` y `w=2` debe retornar `[1,4,5,6,5,3,5]`
- Para `L: [1,2,6,5,8,3,4,6]` y `w=3` debe retornar `[3,4,6,5,5,4]`
- Para `L: [1,2,6,5,8,3,4,6]` y `w=4` debe retornar `[3,5,5,5,5]`

[Ej. 3] **[tpl2 - d10s]** Implemente la función `list<int> d10s(tree<int> &T)` que reciba un AOO `T` y retorne el camino (list<int>) desde la raíz hasta el nodo con la etiqueta `10`. Se garantiza que todo árbol tiene una sola etiqueta con el número `10`.
Ejemplos:

- Para `T = (1 (3 4 (5 10)) 2)` debe retornar `[1,3,5,10]`
- Para `T = (1 (3 4 5 (8 9 (15 10))) 2)`, debe retornar `[1,3,8,15,10]`

[Ej. 4] **[tpl2 - is_cycle]** Implemente una función bool `is_cycle(graph_t &G)` que determine si el grafo `G` es un ciclo dirigido o no. Un grafo ciclo dirigido cumple dos condiciones:

- cada vértice sea de grado 2 (una arista de entrada y otra de salida).
- es posible hallar un circuito Hamiltoniano.

Nota: `graph_t` es un `typedef` de `map<int,list<int>>`.
Ejemplos:

- Para `G = {0=>[1],1=>[2],2=>[3],3=>[4],4=>[5],5=>[0]}` debe retornar `true`.
- Para `G = {0=>[1],1=>[2],2=>[3],3=>[4],4=>[5],5=>[0],6=>[4]}` debe retornar `false`.

[Ej. 5] **[tpl3 - existe_subc]** Implementar una función `bool existe_subc(set<int> &S, int n)` que dado un conjunto `S` y un valor `n`, determine si existe un subconjunto de `S` para el cual la suma de sus elementos sea `n`.
Ejemplos:

- Para `S = {-1,3,5,8,13}, n=5` debe retornar `true`, el sub-conjunto sería `{5}`
- Para `S = {-1,3,5,8,13}, n=10` debe retornar `true`, el sub-conjunto sería `{-1,3,8}`
- Para `S = {-1,3,5,8,13}, n=6` debe retornar `false`
- Para `S = {-1,3,5,8,13}, n=0` debe retornar `true`, el sub-conjunto sería `{ }`

Ayuda:

- Escribir primero un algoritmo recursivo para generar todos los subconjuntos de `S` posibles.
- Modificarlo luego para que cada vez que se genera un subconjunto, se verifique su suma, y se corte la recursión si coincide con `n`.

[Ej. 6] **[tpl3 - replace_btree]** Implemente la función `void replace_btree(btree<int>&T, pt_fun_t pred)` que busque los nodos que cumplan con la condición `pred` y los reemplace. Cada nodo que cumple la condición debe ser reemplazado por el primero de sus descendientes (según el pre-orden) que no la cumpla. En caso de no encontrar ningún descendiente que no la cumpla, todo el nodo (junto con todo su subárbol) se elimina.
Ejemplos:

- Para `T = (2 (4 6 8) (10 12 14)), f = es_par,` debe retornar `T = ()`
- Para `T = (2 (4 6 8) (10 12 13)), f = es_par,` debe retornar `T = (13 . (13 . 13))`
- Para `T = (8 (5 . 7) (6 . 5)), f = es_par,` debe retornar `T = (5 (5 . 7) (5 . 5))`
- Para `T = (7 (5 3 (0 1 .)) 8), f = es_par,` debe retornar `T = (7 (5 3 (1 1 .)) .)`

Ayuda: Escribir una funcion recursiva `replace_btree(T,n,pred)`:

- Si `n` satisface el predicado entonces buscar en su subarbol un elemento `q` que no lo satisfaga.
  - Si `n` es hoja o no hay ningún `q` que no satisfaga el predicado entonces directamente eliminar todo el subárbol de `n`
  - En caso contrario, reemplaza el valor de `n` por el de `q`.
- Aplicar recursivamente a los hijos de `n`

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.