Algoritmos y Estructuras de Datos.
TPLR. Recuperatorio de Trabajos Prácticos de Laboratorio. [2019-11-07]

PASSWD PARA EL ZIP: `4GW3 P267 4QCM`

Ejercicios

[Ej. 1] **[media-movil-entera] (TPL1)** Implemente la función `list<int> mediaMovilEntera(list<int>& L, int v)` que dada una lista de enteros `L`, retorne una lista con los valores de la media móvil con ventana fija de tamaño `v`.
El primer elemento de la lista resultado será el promedio (en división entera) de los primeros `v` elementos de `L`, el segundo será el promedio desde el 2do elemento al `v`+1. En general, el elemento en la posición `j` de la lista resultado será el promedio entre los elementos `[j,j+v)` de `L`
Ejemplos:

- Para `L=[1,2,6,5,8,3,4,6]` y `v=2` debe retornar `[1,4,5,6,5,3,5]`
- Para `L=[1,2,6,5,8,3,4,6]` y `v=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. 2] **[nSumaK] (TPL3)** Implementar la función `bool nSumaK(set<int> &S, int k)` que dado un conjunto `S` y un valor `k`, retorne el número de subconjuntos de `S` para los cuales la suma de sus elementos sea `k`.
Ejemplos:

- Para `S={-5,2,7}`, `k=4` debe retornar `1` (subconjunto `{-5,2,4}`)
- Para `S={-5,2,7}`, `k=2` debe retornar `2` (subconjuntos `{-5,7}, {2}`)
- Para `S={-5,2,7}`, `k=1` debe retornar `0`

Ayuda:

- Escribir primero un algoritmo recursivo para generar todos los subconjuntos de `S` posibles.
- Modificarlo para que cada vez que se genera un subconjunto, se verifique su suma, y se acumule uno si coincide con `k`.

[Ej. 3] **[nCaminosK] (TPL2,TPL3)** Dado un grafo simple `map<int,set<int>> G` y dos vértices `a` y `b`, implementar una función `int nCaminosK(graph& G, int a, int b, int k)` que retorne el número de caminos de longitud `k` entre los vértices `a` y `b`. El camino puede repetir nodos.
Ayuda. Utilizar una estrategia recursiva:

- Si la longitud `k=0`, se habrá encontrado un camino si `a==b`.
- Si no, contar la cantidad de caminos de longitud `k-1` entre alguno de los vecinos de `a` y `b`.

[Ej. 4] **[make-heap] (TPL3)** Escribir una función `void make_heap(btree<int> &T);` que convierte *in-place* el árbol binario en parcialmente ordenado, aplicando el algoritmo *make-heap*. (**Nota:** esto lo vemos en el curso para montículos representados como vectores, aquí pedimos hacerlo en un árbol binario, que puede no ser parcialmente completo). Por ejemplo,

```
(5 (3 2 1) (2 1 0))       => (0 (1 2 3) (1 5 2))
(4 (9 0 .) (4 8 .))       => (0 (4 9 .) (4 8 .))
(5 (5 . 5) (4 9 (0 . 7))) => (0 (5 . 5) (4 9 (5 . 7)))
```

El algoritmo se basa en escribir una función recursiva **`make_heap(T,n)`** de acuerdo a las siguientes instrucciones,

- Si el nodo es Λ entonces no hace nada.
- Caso contrario aplicar recursivamente **`make_heap`** a los hijos.
- Aplicar **`re_heap`** en el nodo, intercambiando el nodo con su hijo menor siempre que el hijo sea mayor que **`n`**. Repetir esto hasta llegar a una hoja o a un nodo tal que ambos hijos con mayores o iguales que el nodo.

[Ej. 5] **[sort-stack](TPL1)** Escribir un programa que ordene una pila **`S`** utilizando recursión de forma tal que los elementos de mayor valor se encuentren en el tope. No está permitido el uso de constructores iterativos (**`while`**, **`for`**, etc) ni tampoco el uso estructuras de datos adicionales. Sólo pueden utilizarse métodos de la pila, a saber:

- **`S.empty()`**
- **`S.top()`**
- **`S.push(s)`**
- **`S.pop()`**

Se propone el siguiente algoritmo recursivo:

- Si la pila está vacía finalizar
- Si no:
  - Guardar una copia del elemento al tope.
  - Quitarlo de la pila.
  - Ordenar el resto de la pila.
  - Insertar el elemento guardado al tope de la pila ordenada.

Para el último paso se requiere un método auxiliar, también recursivo, que reciba un elemento y una pila e inserte dicho elemento en forma ordenada, siguiendo el siguiente algoritmo:

- Si la pila está vacía o si el elemento a insertar es mayor al tope, insertar el elemento.
- Si no:
  - Almacenar una copia temporal del elemento al tope.
  - Quitarlo de la pila.
  - Insertar el elemento recibido en forma ordenada recursivamente.
  - Apilar el elemento temporal en la pila resultado.

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.

mstorti@galileo/tpl2-2019-61-g71f85448/Sat Nov 9 01:20:51 2019 -0300