Algoritmos y Estructuras de Datos.
TPLSR. Super Recup de TPLs. [2015-11-19]

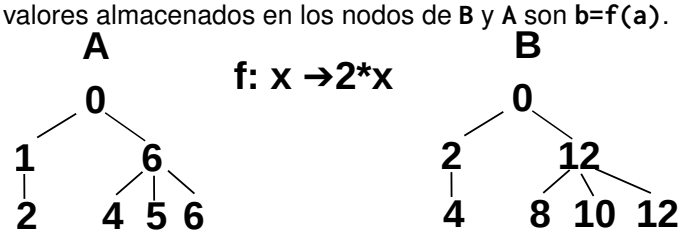
PASSWD PARA EL ZIP: 76DS YVK1 T8NH

Ejercicios

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

[Ej. 1] **[mkstats (25pt)]** Dado un vector de conjuntos `vector<set<int> > V`, escribir una función `void mkstats(vector<set<int> > &V, map<int,int> &count);` de tal forma que `counts[j]` es la cantidad de conjuntos que contienen a `j`. Si un entero no está en ningún conjunto NO debe estar como clave en `counts` o, dicho de otra manera, los valores en `counts` deben ser todos `count[j]>0`. Por ejemplo, si `V=[{1,2,3},{2,3,4},{3,4,5,9}]` entonces debe dar `M=(1->1,2->2,3->3,4->2,5->1,9->1)`.

[Ej. 2] **[ismaptree (35pt)]**: Dado dos árboles ordenados orientados (AOO) `tree<int> A,B`, y una función de mapeo `mapfunc_t f` (donde `typedef int (*mapfunc_t)(int);`) determinar si `B=f(A)`, es decir si `B` es **semejante** a `A` y los valores almacenados en los nodos de `B` y `A` son `b=f(a)`.



Es decir `B` debe ser semejante a `A` y cada valor de nodo en B surge de aplicar la función `f` al nodo de la posición equivalente en `A`.

Ejemplos: Asumiendo en todos los casos que la función de mapeo es `f(x)=2*x`

```
A=(0 (1 2) (3 4 5 6)), B=(0 (2 4) (6 8 10 12))   -> true
A=(0 (1 2) (3 4 5 6)), B=(0 (2 4) (6 8 10 13))   -> false (mal el valor 13)
A=(0 (1 2) (3 4 5 6)), B=(0 (2 4 5) (6 8 10 12)) -> false (el nodo 5 está de mas)
A=(0 (1 2) (3 4 5 6)), B=(0 (2 4) (6 8 12))      -> false (falta el nodo 10)
```

Ayuda (versión 1): Escribir una función recursiva auxiliar que itera sobre `A` y `B` (como en el `operator==()` de AOO) pero verificando que `*nb=f(*na)` para los nodos `na` en `A` y `nb` en `B`.
Ayuda (versión 2): Escribir una función `void apply(tree<int> &A,mapfunc_t f,tree<int> &fA);` que construye a `fA=f(A)` y después utilizar el operador de igualdad entre árboles.

[Ej. 3] **[foodplan (35pt)]** La princesa Leia se cansó de tener que pensar todas las noches la comida para ella y Hans Solo, así que le pidó a Hans que escriba un programa para que la computadora de a bordo del Halcón Milenario genere cronogramas de comidas para el resto del año. Hans escribió un programa que genera vectores con listas de 7 comidas (para cada día de la semana, para varias semanas). Pero su programa no respeta los caprichos de la princesa. A la princesa no le gusta, por ejemplo, comer pastas dos veces seguidas. Escriba una función `foodplan` que reciba el vector de listas con los cronogramas de varias semanas, una función `f` que dadas dos comidas retorna verdadero si la princesa acepta que se programen consecutivas en la misma semana, y un conjunto con comidas alternativas. Se debe buscar en cada lista pares de comidas consecutivas que no cumplan con el requisito, y corregirlo. Para corregirlo, cuando dos comidas no cumplen el requisito, debe intercambiar la segunda de ellas con la

primer opción compatible que encuentre en el conjunto de alternativas, y eliminar esa opción del conjunto. Las comidas se representan con `std::string`
Por ejemplo, si:

- El vector (para solo una semana) es `[("Pollo","Pollo","Vegetales","Pizza","Pastas","Pastas","Hamburguesas")]`,
- La función `f` retorna falso cuando recibe dos comidas iguales,
- y el conjunto de alternativas es `{"Empanadas", "Pescado"}`, la función debe corregirlo para que quede:
  `[("Pollo","Empanadas","Vegetales","Pizza","Pastas","Pescado","Hamburguesas")]`

Explicación: primero cambia la segunda aparición de `"Pollo"` por `"Empanadas"`, y alternativas queda `{"Pescado", "Pollo"}` ... Luego cambia la segunda aparición de `"Pastas"` por `"Pescado"`.
Consigna: escribir la función `bool foodplan(vector<list<string>> &cronograma, bool (*f)(string,string), set<string> &alternativas);`. Nota: la función retorna `true` si logra arreglar el cronograma, `false` si no encuentra una alternativa factible para alguna de las correcciones.
Ayuda:

- En `foodplan`:
  Para cada lista del vector
  Por cada par de elementos de la lista consecutivos `(c1,c2)`
  Aplicarles la función `F`, si retorna `false`
  Invocar a una función auxiliar `Corregir` pasandole `c1, c2, F` y `alternativas`
  Si `Corregir` retornó `false`, terminar retornando `false`
  Retornar true
- En la función auxiliar `Corregir:`
  Para cada elemento `(a)` del conjunto `alternativas`
  Invocar a `F` con `c1` y `a`.
  Si retorna true
  remover `a` del conjunto e insertar `c2`
  asignar `a` en `c2`, y retornar `true`
  Retornar false

Nota: no se debe considerar el caso en que la última comida de una semana coincide con la primera de la siguiente, ya que las restricciones aplican solo a comidas "de la misma semana"

[Ej. 4] **[minsublist (25pt)]** Escribir una función `int minsum(list<int> &L);` que dada una lista de enteros `L`, se debe retornar la suma total de la sublista de menor suma.
Ejemplos:

```
L=[1 -2 3 -3 -1 2 5 ]; Retorna: -4 (corresponde a sublista [-3 -1])
L=[1 -2 1 -3 -1 2 5 ]; Retorna: -5 (corresponde a sublista [-2 1 -3 -1])
L=[1 2];               Retorna = 0 (corresponde a la sublista vacía [])
```

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.

- 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. 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.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`**: Ahora las funciones **`eval()`** tienen dos parámetros adicionales:
  `Eval::eval(func_t func,int vrbx,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(set<int> S,string s="")`**.
  - **`void Eval::dump(list <int> &L,string s="")`**

  - `void Eval::dump(list< list<int> > &LL,string s="")`.
  - `tree<int>::lisp_print()`: Lisp print de un árbol ordenado orientado (AOO). Nota: esta pertenece a la clase `tree`. Uso: `tree<int> T; T.lisp_print();`
  - Idem para AB: `btree<int>::lisp_print()`: Lisp print de un árbol binario (AB). Nota: esta pertenece a la clase `btree`. Uso: `btree<int> T; T.lisp_print();`
- Después del parcial deben entregar el programa fuente (sólo el `program.cpp`) renombrado con su apellido y nombre (por ejemplo `messilionel.pdf`). Primero el apellido.
- **Puntos:** Los puntos de los ejercicios son variables (hay de 25pt y 35pt). Los puntos conseguidos son **comodines**, se pueden agregar a cualquiera de los TPLs hasta que sature en 100pt.
- **Torneo de programación:** Los ejercicios de este TPL **no participan** del Torneo.