Algoritmos y Estructuras de Datos.
Recuperatorio de Trabajos Prácticos de Laboratorio. [2018-11-22]

PASSWD PARA EL ZIP: FRP5 2PF7 XR7J

Ejercicios

[Ej. 1] [tpl2 - tree-comp]

Queremos escribir una función predicado **`bool tree_less(tree<int> &T1,tree<int> &T2)`** que sea una relación de orden fuerte para AOO. Debe retornar **`true`** si **T1**<**T2**. Recordemos que una relación de orden fuerte debe ser transitiva y también debe satisfacer que si **T1**<**T2** y **T2**<**T1** son falsos debe ser **T1**=**T2**.

Recordemos cómo procedemos con objetos que tiene varios campos miembro **`class A { public: int j; double z; string s; }`**. Si tenemos dos objetos **a1** y **a2**, comparamos primero el campo **j**, es decir si **`a1.j<a2.j`** retornamos **`true`**, si **`a2.j<a1.j`** retornamos **`false`**, y si ninguno de los dos es cierto continuamos con el siguiente campo **z** y después **s**. Si todos los campos son iguales debemos retornar **`false`** ya que quiere decir que todos los campos son iguales.

En el caso de un AOO podemos pensar que es un objeto compuesto del elemento contenido en la raíz **r** y de una lista de objetos que son los subárboles de los hijos de la raíz, es decir **`T=(r (a...) (b...) (c...) ...)`** donde **a** es el hijo más izquierdo de **r**, **b** el segundo, y así siguiendo. Entonces dados dos árboles, **`T1=(r1 (a1...) (b1...) (c1...) ...)`**, y **`T2=(r2 (a2...) (b2...) (c2...) ...)`**.

Además el subárbol vacío Λ equivale a $-\infty$.

Entonces la función de comparación debe hacer lo siguiente:

- Si **T1**==Λ y **T2**==Λ retornar **`false`** (son iguales)
- Si **T1**!=Λ y **T2**==Λ retornar **`false`** (**T2** es $-\infty$ y nadie puede ser menor que $-\infty$)
- Si **T1**==Λ y **T2**!=Λ retornar **`true`** **T1** es $-\infty$ y **T2** no $\Rightarrow$ **T1**<**T2**)

Pasado este punto sabemos que ninguno de los dos es Λ y debemos comparar dos árboles que no son vacíos:

- Si **r1**<**r2** retornar **`true`**, si **r2**<**r1** retornar **`false`**.
- Si **`(a1...)`**<**`(a2...)`** retornar **`true`** si **`(a2...)`**<**`(a1...)`** retornar **`false`**.
- Idem para **`(b1...) (b2...)`**
- Idem para **`(c1...) y (c2...)`** hasta recorrer todos los hijos.

Tener en cuenta que para la raíz se deben comparar los valores almacenados en los nodos, mientras que para los hijos se deben comparar los subárboles, aplicando la función recursivamente.

Si al recorrer las listas de hijos, son todos iguales y algunas de ellas se acaba entonces hay que pensar como que los Λ equivalen a $-\infty$ es decir,

- Si se termina la lista de hijos de **T1** pero no la de **T2** retornar **`true`** (ya que **T1**<**T2**)
- Si termina la lista de hijos de **T2** pero no la de **T1** retornar **`false`** (ya que **T1**>**T2**)
- Si ambas listas se terminaron retornar **`false.`** (ya que **T1**=**T2**)

[Ej. 2] [tpl1 - xcommon]

Implemente una función `void xcommon(list<int> &L1,list<int> &L2,list<int> &common)` tal que dadas dos listas `L1` y `L2` extraiga la parte común del comienzo de `Lcommon`, de tal forma que `L2=(Lcommon,Ltail1)` y `L2=(Lcommon,Ltail2)`. Adicionalmente, los argumentos `L1`, `L2` deben quedar con las "colas" correspondientes `Ltail1` y `Ltail2`.

Ejemplo: si `L1=(0,3,2,1,5,6)` y `L2=(0,3,2,4)` ⇒ `Lcommon=(0,3,2)`, `L1=(1,5,6)`, `L2=(4)`

[Ej. 3] [tpl3 - xsubtrees]

Implemente la función `void xsubtrees(btree<int> &T, int depth, list<btree<int>> &Lout)` que dado un árbol binario `T` y un entero `depth` extraer del árbol `T` todos los subárboles de nodos que estén a profundidad `depth` (moverlos hacia la lista de subárboles `Lout`).

[Ej. 4] [tpl3 - maxsubK]

Escriba una función int `maxsubk(set<int> &S, int k)` que devuelva la máxima suma en valor absoluto de todos los subconjuntos posibles del conjunto `S` tomados de a `k`.

Ejemplos: Para `S={2,-3,5,-1,4}`:

- `maxsubk(S,1)` ⇒ `5`
- `maxsubk(S,2)` ⇒ `9`
- `maxsubk(S,3)` ⇒ `12`
- `maxsubk(S,4)` ⇒ `14`

Ayuda

- Implemente una función auxiliar que sume, en valor absoluto, todos los elementos de un conjunto
- Una propuesta de algoritmo recursivo para generar todos los k-subconjuntos realizaría lo siguiente:
  - Si el conjunto parcial ya tiene `k` elementos, retorna su suma
  - Si no hay más elementos en el conjunto total, retorna `-1`
  - Sino:
    - Toma el primer elemento del conjunto total y lo elimina
    - Recursión sin incluir el elemento en el conjunto parcial
    - Recursión incluyendo el elemento en el conjunto parcial
    - Retorna la mayor de las sumas absolutas obtenidas en recursión

[Ej. 5] [tpl2/tpl3 - num-path]

Escriba una función `int num_path(map<int,set<int>>& G, int i, int j)` que retorne el número de caminos (sin ciclos) en el grafo no dirigido `G`, partiendo desde el vértice `i` y finalizando en el vértice `j`. Se garantiza que los nodos `i` y `j` están en el grafo.

Ejemplos: Para `G` = `{ 1→{2,3,4}, 2→{1,3}, 3→{1,2}, 4→{1} }`:

- `num_path(G,1,4)` ⇒ `1`
- `num_path(G,1,3)` ⇒ `2`
- `num_path(G,1,2)` ⇒ `2`
- `num_path(G,3,4)` ⇒ `2`

[Ej. 6] [tpl1 - super-stable-partition]

Escribir una función **`super_stable_partition`** que reciba una lista con enteros **`L`** y dos listas vacías **`L_low`** y **`L_geq`**, y:

- determine si existe alguna posición para particionar la lista en dos sublistas según el valor de dicha posición, sin reordenarla. Es decir, debe encontrar una posición tal que todos los elementos previos a la misma sean menores al valor que hay en dicha posición, y todos los elementos posteriores sean mayores o iguales.
- si existe tal posición (si hay más de una, tome la primera en la secuencia), mueva (splice) ambas partes a las listas **`L_low`** (menores) y **`L_geq`** (mayores o iguales) y retorne **`true`**; si no existe retorne **`false`**.

Ejemplos

- `L = {5,3,2,7,9,7,10}` ⇒ `true, L = {} L_low = {5,3,2}, L_geq = {7,9,7,10}`
- `L = {1,3,2,7,9,7,1}` ⇒ `true, L = {}, L_low = {}, L_geq = {1,3,2,7,9,7,1}`
- `L = {10,3,2,7,9,7,11}` ⇒ `true, L = {}, L_low = {10,3,2,7,9,7}, L_geq = {11}`
- `L = {5,3,2,7,9,7,1}` ⇒ `false, L = {1,3,2,7,9,7,1}, L_low = {}, L_geq = {}`

Ayuda

- Escribir una función auxiliar que dada una lista y una posición, determine si dicha posición es váida para generar la partición
- Recorrer la lista **`L`** determinando en cada posición si es o no válida con la función auxiliar:
  - Si una posición es válida:
    - Mover (splice) a **`L_low`** desde el comienza hasta esa posición
    - Mover (splice) a **`L_geq`** todo lo que queda en **`L`**
    - Finalizar retornando **`true`**
- Si no se encontró posición válida, retornar **`false`**

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.