Algoritmos y Estructuras de Datos.
TPL2R. Recup Trabajo Práctico de Laboratorio 2. [2014-10-16]

PASSWD PARA EL ZIP: `NPZ PZ4 ESB V51`

Ejercicios

[Ej. 1] **[mkmtree]** Escribir una función `void mkmtree(tree<int> &T,int m,int k);` que dados dos enteros positivos `m`, `k`, (con `0<k<m`) construye un **árbol ordenado orientado (AOO)** `T` de la siguiente forma:

- En la raíz tiene el valor `m`
- Los hijos de `m` son: `m-k`,..., `m-j*k`, mientras los elementos sean **no negativos** `m-j*k>=0`.
- La definición se propaga **recursivamente**, es decir si un nodo `n` tiene el valor `*n`, los hijos son `(*n)-k`,..., `(*n)-j*k`,... mientras `(*n)-j*k>=0`.

Ejemplos:

- `m=10,k=3, -> T=(10 (7 (4 1) 1) (4 1) 1)`
- `m=10,k=2, -> T=(10 (8 (6 (4 2) 2) (4 2) 2) (6 (4 2) 2) (4 2) 2)`
- `m=20,k=7, -> T=(20 (13 6) 6)`
- `m=23,k=6, -> T=(23 (17 (11 5) 5) (11 5) 5)`

[Ej. 2] **[has-equal-path]** Dado un arbol ordenado orientado (AOO) `T` escribir una funcion predicado que determina si contiene un camino de valores iguales que va desde la raíz a una hoja con todos elementos iguales. Por ejemplo

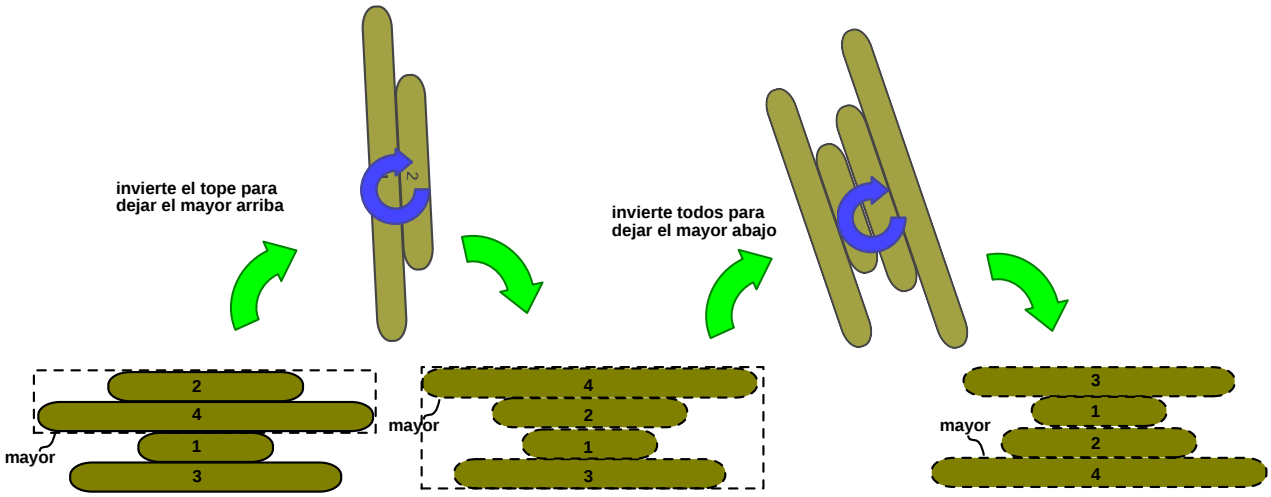
- `T=(1 (1 2 3) (1 2 4)) -> false`
- `T=(1 (1 2 3) (1 2 1)) -> true` (el camino es `(1,1,1)`)
- `T=(6 (6 2 6) (1 2 4)) -> true` (el camino es `(6,6,6)`)
- `T=(1 (2 2 1) (1 2 4)) -> false`

Consigna: Escribir una función `bool has_equal_path(tree<int> &T);` que realiza la tarea indicada.

Ayuda: Escribir una función recursiva
`bool has_equal_path(tree<int> &T,tree<int>::iterator n);` que

- Si `n` es Λ o una hoja retorna `true`.
- Caso contrario recorre los hijos `c` de `n` y retorna `true` sí y sólo si alguno de ellos es tal que su valor al del padre (es decir `*c==*n`) y tiene un camino de valores iguales a una hoja (**recursividad**) .

[Ej. 3] **[pancake-sort]:** Dada una pila de números `stack<int> &S`, implementar el algoritmo **Pancake Sort** para ordenar la misma. El algoritmo se basa en buscar el elemento mayor de la pila y colocarlo al fondo de la misma. Luego buscar el segundo mayor elemento y colocarlo en el fondo-1 de la pila. Y así sucesivamente hasta que quede completamente ordenada.



Sin embargo, hay una restricción para efectuar estas operaciones. Sólo se pueden realizar operaciones en las cuales se **invierte** una cierta cantidad de elementos en el tope de la pila, como si fuera una pila de panqueques (de ahí el nombre del algoritmo). Al encontrar el elemento mayor, se debe invertir toda la pila hasta la posición en que se encuentra el elemento mayor, y luego volver a invertir toda la pila hasta que el elemento mayor quede en el fondo de la pila. Luego repetir estos pasos con el resto de los elementos.

Consigna: Escribir la función `void pancake_sort(stack<int> &S);` que debe retornar por referencia, la pila ordenada.

Restricciones: Solo se deben utilizar como estructuras de datos auxiliares, pilas o colas.

Ayuda: Considerar el siguiente algoritmo:

Comenzar desde el tamaño actual de la pila e ir decrementando en 1 hasta llegar a 1. Llamar a este tamaño `POSITION`, y luego hacer lo siguiente para cada uno de los valores de `POSITION`:

- Encontrar el índice al máximo elemento `MAX_ELEM` en la pila desde el tope de la pila hasta la posición `POSITION`.
- Llamar a una función `FLIP(stack,i)` que se encargue de invertir la pila hasta la posición `i`-ésima con `i=MAX_ELEM`.
- Llamar a `FLIP(stack,i)` con `i=POSITION`.

[Ej. 4] **[count-cycles]** Dado un `map<int,int> &M` que representa una **permutación** (es decir tal que el conjunto de las claves es igual al conjunto de las imágenes) se deben contar cuantos **ciclos** tiene presente. Un ciclo se corresponde con una secuencia de pares `M[clave] = valor` tal que `M[X1] = X2`, `M[X2] = X3`, `M[X3] = ... , M[...] = X1`. Se debe considerar que `M[X1] = X1` es un ciclo.

Ejemplo:

Suponiendo el mapa siguiente:

```
M[1] = 2;
M[2] = 4;
M[3] = 6;
M[4] = 9;
M[6] = 3;
M[9] = 1;
M[10] = 10;
```

La función retornará 3. Los ciclos son

- `1->2->4->9->1`,

- `3->6->3`,
- `10->10`,

Consigna: Escribir la función `int count_cycles(map<int,int> &M);` que debe retornar la cantidad de ciclos que posee el mapa.

Restricción: los ciclos se deben contar una sola vez, ya que por ejemplo, el ciclo `1->2->4->9->1` es el mismo que `2->4->9->1->2` o `4->9->1->2->4`.

Ayuda:

- A partir de cualquier clave de la correspondencia `kini` hacer `k=kini` y aplicar repetidamente `k=M[k]` hasta volver a `k==kini`. Por ser una permutación está garantizado que a lo sumo en `M.size()` iteraciones esto va a ocurrir.
- Ir **eliminando** los pares de asignación por los cuales se va pasando, para no contar los ciclos dos veces (el algoritmo es **destructivo**).
- Mantener un **contador** de los ciclos.
- Prestar especial atención a los casos de ciclos de **un sólo elemento** (`k==M[k]`)

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.
- 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.evalj(f,vrbs);
h1 = ev.evaljr(f,seed); // para SEED=123 debe dar H1=170
```

`j` es el número de ejercicio, por ejemplo para el ejercicio 1 tenemos las funciones (`eval1` y `eval1r`). La primera `ev.evalj(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
```

- La segunda función `evaljr` 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 `X=123` entonces el checksum es `H=523`. Una vez que el alumno termina su tarea se le pedirá que corra la clase evaluadora con un valor determinado de la semilla `X` 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 `X` 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 las siguientes funciones utilitarias:
  - `void dump(map<int,int> &M,string s="")`: Imprime un mapa entero/entero. Nota: Es un método de la clase `Eval` es decir que hay que hacer `Eval ev; ev.dump(M);`. El string `s` es un label opcional.

- **`void dump(stack<int> S,string s="")`**: Imprime una lista de enteros (primero el tope). Uso: **`Eval ev; ev.dump(S);`**
- **`void dump(queue<int> S,string s="")`**: igual que **`dump(stack<int>)`**
- **`tree<int>::lisp_print()`**: Lisp print de un árbol. Nota: esta pertenece a la clase **`tree`**. Uso: **`tree<int> T; T.lisp_print();`**