Algoritmos y Estructuras de Datos.
TPL2R. Recup Trabajo Práctico de Laboratorio. [2015-10-22]

PASSWD PARA EL ZIP: `P3EJ KPW8 DW9H`

Ejercicios

ATENCION/ERRATA: En **`program.cpp`** el orden de los templates para **`tree2count`** y **`count2tree`** están invertidos.

[Ej. 1] **[tree2count (25pt)]** Escribir una función **`void tree2count(tree<int> &T,list<int> &L)`** que, dado un Arbol Ordenado Orientado (AOO) **`T`** retorna una lista que contiene para cada nodo **`n`**, en preorden, la cantidad de nodos del subárbol de **`n`**. Por ejemplo,

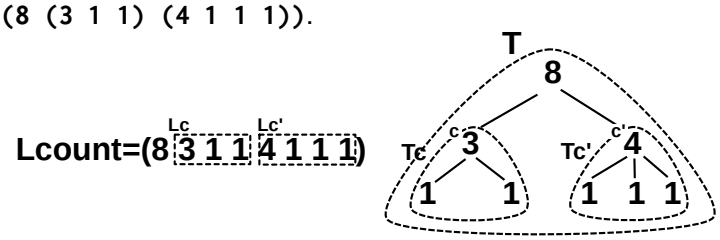
```
T=(4 5 3)  => L=(3 1 1);
T=(6 (7 8 9) (1 2 3 4))  => L=(8 3 1 1 4 1 1 1);
```

Ayuda:

- Conviene hacerlo en dos pasos **`mkcount(T)`** y aplicar luego **`preorder()`**.
- Escribir una función auxiliar recursiva **`mkcount(tree<int> &T,node_t n)`** que reemplaza los valores de los nodos de **`T`** por el conteo de nodos de su subárbol. Si el nodo es Λ no hace nada. Si no es Λ aplica primero **`mkcount`** a los hijos y después reemplaza el valor del nodo por 1+ los valores de los hijos (que a esa altura contendrán el conteo de nodos de sus subárboles).

ATENCION: En **`program.cpp`** la función recursiva auxiliar **`void tree2count(tree<int> &T, node_t n,list<int> &Ln);`** no coincide con la ayuda que les damos. Se podría hacer de esa forma pero no es como le estamos diciendo en la ayuda previa. Si van a seguir la ayuda indiciada aquí conviene eliminarla.

[Ej. 2] **[count2tree (35pt)]** Sea un árbol **`T`** tal que en sus nodos contiene el número de nodos que contiene su subárbol. Por ejemplo **`T=(8 (3 1 1) (4 1 1 1))`**.



Entonces, dada la lista **`Lcount=(8 3 1 1 4 1 1 1)`** de los valores (que son la cantidad de nodos en el subárbol) podemos reconstruir el árbol con el siguiente algoritmo. Sea **`n`** la raíz del árbol.

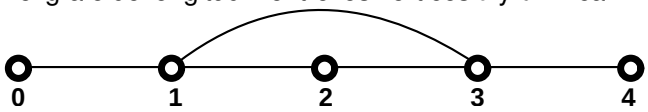
- El primer elemento **`count=8`** es la cantidad de nodos del árbol.
- El siguiente elemento **`3`** es la cantidad de nodos del subárbol del primer hijo **`c`** de **`n`**. Extrae **`3`** elementos **`Lc=(3 1 1)`** y esa es el **`Lcount`** del subárbol de **`c`**.
- El siguiente elemento **`4`** es la cantidad de elementos del segundo subárbol por lo tanto el **`Lcount`** del siguiente nodo **`c'`** es **`Lcount=(4 1 1 1)`**.
- El proceso se detiene cuando se extraen tantos elementos como **`count-1`** (el 1 corresponde a la raíz **`n`**).

Consigna: Escribir una función **`void count2tree(list<int> &L,tree<int> &T);`**

Ayuda: Escribir una función auxiliar recursiva
`node_t count2tree(list<int> &L,tree<int> &T,node_t n);`. Al entrar a la función **n** debe ser un Λ.
La función construye el subárbol en **n** y retorna el iterator refrescado.

- Extrae el primer elemento **count** de **L**, esa es la cantidad de nodos total del árbol, y lo inserta en **n**.
- Hace **count--** ya que insertó un nodo. El valor de **count** es ahora la cantidad total de nodos en los subárboles.
- Recursivamente va a aplicando el algoritmo a **Lcount** y va extrayendo subárboles en los hijos de **c**, **c'**... de **n** hasta que se hayan extraído **count** elementos de la lista.
- Prestar atención con retornar el nodo refrescado ya que la función **count2tree** toma un nodo vacío y inserta algo allí, por lo tanto el iterador cambia.
- Prestar atención a refrescar el iterador y avanzarlo al recorrer los hijos **c**, **c'** ...
- Mantener correctamente el conteo de los nodos que se van extrayendo de la lista. Cada vez que se aplica **count2tree** a **Lcount** la cantidad de nodos está en la primera posición; deben ir acumulando ese valor.

[Ej. 3] **[hasnpath (25pt)]** Dado un grafo simple `map<int,list<int>> G` y dos vértices **a** y **b**, implementar una función `bool hasnpath(graph&G, int a, int b, int n);` que retorne **true** si existe al menos un camino en el grafo de longitud **n** entre los vértices **a** y **b**. El camino puede repetir nodos.



Por ejemplo en este caso tenemos que para los nodos **a=0**, **b=4** debe dar:

```
a=0, b=4, n=1 -> 0
a=0, b=4, n=2 -> 0
a=0, b=4, n=3 -> 1
a=0, b=4, n=4 -> 1
a=0, b=4, n=5 -> 1
```

Ayuda: Utilizar una estrategia recursiva:

- Si la longitud **n** es 0, entonces debe ser **a==b**.
- Si no, verificar si existe un camino de longitud **n-1** entre alguno de los vecinos de **a** (la adyacencia de **a**) y **b** (o viceversa, la adyacencia de **b** con **a**).

[Ej. 4] **[key2abbrevs (35pt)]** Dada una lista de strings todos diferentes entre sí encontrar el menor prefijo de cada uno (abbrev) lo mas corto posible pero de tal forma que las abreviaciones sean diferentes entre si. La abreviación debe tener al menos longitud 1. Por ejemplo

```
# key -> abbrev
abcb -> a
acaa -> ac
badc -> b
bbad -> bb
bbbd -> bbb
bbcd -> bbc
bdca -> bd
```

Consigna: escribir una función `void key2abbrevs(map<string,string> &abbrevs);` que realiza la tarea indicada. Inicialmente **abbrevs** contiene como claves las keys y las imágenes son todas el string vacío. A la salida de la función la imagen debe ser la abreviación.

Ayuda: Utilizar un map auxiliar **`map<string,string> abb2key`** que contendrá la inversa de **`abbrevs`** (es decir **`abbrev->key`**). Inicialmente está vacío. Para cada key de **`abbrevs`**, probar con el prefijo de longitud 1. Si es clave de inverse **`abb2key`** ya tiene otra key asignada y esa abreviación no puede usarse. Probar con un prefijo de longitud 2 y así siguiendo hasta encontrar un prefijo que no tenga valor asignado en **`abb2key`**. Cuando encuentra el prefijo asignarlo en **`abbrevs`** y (la asignación inversa) en **`abb2key`**.
Ayuda memoria para strings: **`s,s1,s2`** son string:

- **`s.substr(pos,len)`** retorna el substring de longitud **`len`** que empieza en **`pos`**. Por ej. **`s="abcd"; s.susbtr(0,3) -> "abc"`**
- **`s.size()`** longitud del string.
- **`s[j]`** char en la posición **`j`**
- **`s=s1+s2`** concatenación.

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.evalj(f,vrbs);
  hj = ev.evalrj(f,seed); // para SEED=123 debe dar Hj=170
  ```

  `j` es el número de ejercicio, por ejemplo para el ejercicio 1 tenemos las funciones (**`eval1`** y **`evalr1`**). 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 **`evalrj`** 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 **`evalrj()`** 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="")**
- **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:** Notar que la suma de los puntos es 120. La nota del parcial **min(sum(Zj),100)** es decir que si haciendo los tres primeros ejercicios se obtienen 95/100 pts. Para aprobar basta hacer 50/100 pts de manera que basta con cualquiera dos de los cuatro.
- **usercase**: 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.
- **Torneo de programación:** Los ejercicios de este Recup **no participan** del Torneo.