Trabajo Práctico 1
Árboles

Organización del Computador 2

Primer Cuatrimestre 2013

1. Introducción

El objetivo de este trabajo práctico es implementar un conjunto de funciones sobre una estructura recursiva que representa árboles. Un árbol contiene una lista de hijos, un valor de 64 bits, y un tipo. Los hijos del árbol tienen valores del mismo tipo.

Las funciones a implementar permiten crear árboles, recorrerlos y borrarlos. Además de estas operaciones, se deben proveer dos funciones avanzadas, *prune* y *merge*. La primera permite podar el árbol, quitando ramas a partir de los nodos que no cumplan una cierta condición; la segunda se encarga de unir nodos del árbol.

Los ejercicios a realizar para este trabajo práctico estarán divididos en tres secciones. En la primera deberán completar las funciones que permiten manipular el tipo árbol. La segunda sección corresponde a un conjunto de funciones muy sencillas que operan sobre tipos de datos básicos. Estas funciones serán utilizadas por el tipo árbol para las operaciones avanzadas. En la última sección deben construir un programa en `C` que arme un árbol y ejecute algunas de las funciones ejemplo.

1.1. Tipo tree

La estructura del árbol está compuesta por un bloque de memoria que contiene un puntero a una lista enlazada de nodos que apuntan a los hijos, de un valor de 64 bits, y por último de un entero que indica su tipo. El valor puede usarse para almacenar los tipos `int`, `double` o `string` de C. La lista enlazada de nodos consta dos punteros, el primero de ellos apunta al siguiente nodo, y el segundo apunta al árbol hijo. En la figura 1 se puede ver un ejemplo para un árbol de ints que contiene 5 valores.

```
typedef enum tree_type { Integer, Double, String } tree_type;

typedef union tree_value {
    int i;
    double d;
    char *s;
} __attribute__((__packed__)) tree_value;

typedef struct tree {
    struct list_node *children;
    tree_value value;
    tree_type  type;
} __attribute__((__packed__)) tree;
```

```
typedef struct list_node {
    tree *element;
    struct list_node *next;
} __attribute__((__packed__)) list_node;
```

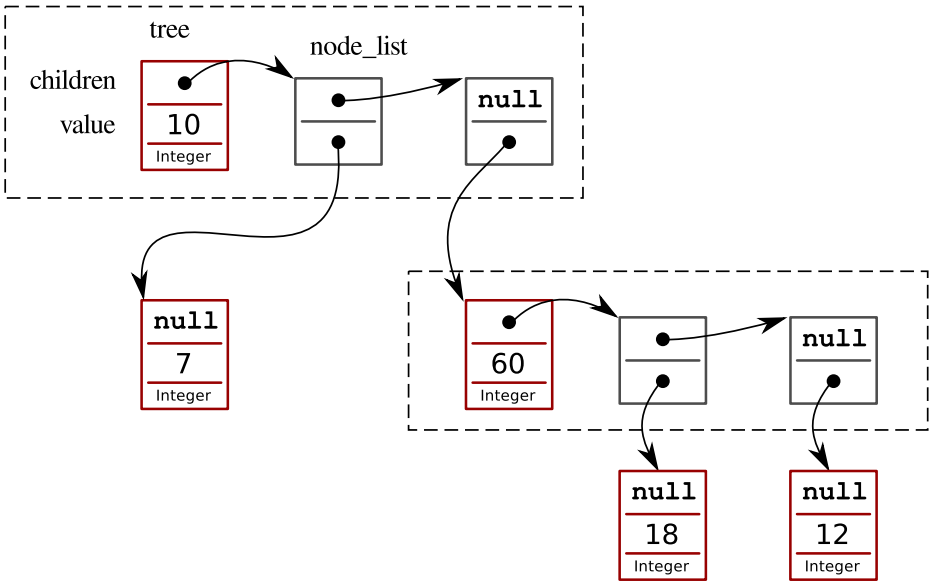


Figura 1: Ejemplo de una estructura árboles de `int`s de C.

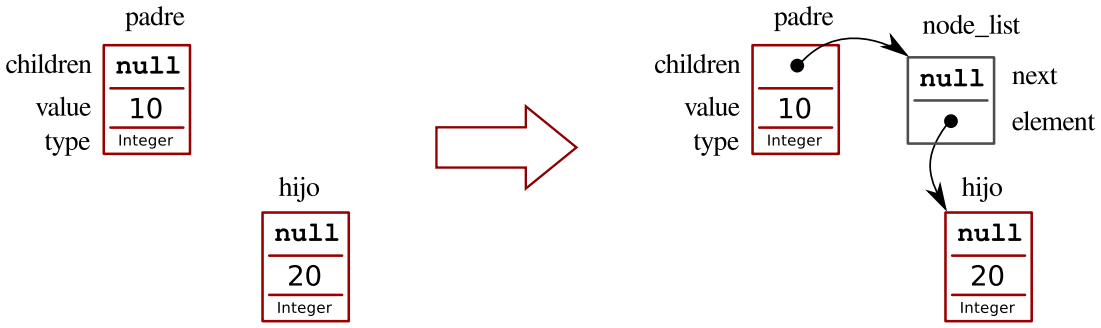
Funciones de tree

- `tree* tree_create(tree_value value, tree_type type);`
  Crea un árbol nuevo con valor value, del tipo type, y sin ningún hijo, es decir sin ningún nodo.
  Ej. `tree t = tree_create(ival(10), Integer);`

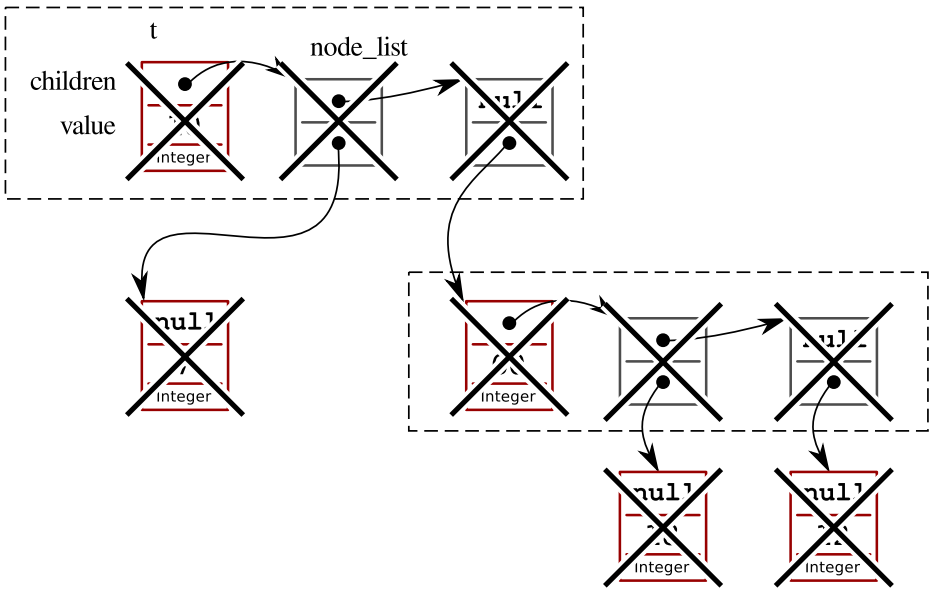
- Tres funciones *helpers*, que usan tree_create, para crear árboles de tipos específicos:
  - `tree* tree_create_int(integer value);`
    Para árboles de enteros. Ej. `tree *t = tree_create_int(10);`
  - `tree* tree_create_double(double value);`
    Para árboles de doubles. Ej. `tree *t = tree_create_double(2.5);`
  - `tree* tree_create_string(char *str);`
    Para árboles de strings. Ej. (es importante pasar al árbol solamente strings en el heap, alocados con malloc).

```
char *str = "banana";
int size = strsze(str) + 1;
str_en_heap = malloc(size);
strcpy(str_en_heap, str, size);
tree *t = tree_create_string(str_en_heap);
```

- `void tree_add_child(tree *self, tree *element);`
  Agrega el árbol element al final de la lista de hijos de self. Para esto, crea un nuevo tree_node que apunte al árbol hijo. Ej. padre es un árbol sin hijos de valor entero 10, hijo es un árbol sin hijos de valor entero 20. `tree_add_child(padre, element);`



- `void tree_deep_delete(tree *self);`
  Borra el árbol, eliminando todos los nodos que apuntan a sus hijos y también a sus hijos recursivamente, liberando toda la memoria correspondiente. En el caso de los árboles de strings, se debe liberar la memoria apuntada por los nodos
  Ej. `tree_deep_delete(t);`



- `void tree_print(tree *self, char* extra, char *archivo);`
  Debe agregar al archivo pasado por parámetro una línea por cada nodo del árbol self. El archivo se debe abrir en modo append, de modo que las nuevas líneas sean adicionadas

al archivo original.
Antes de imprimir el árbol se imprimirá el contenido de la variable `extra` seguido de un *enter*. El árbol se debe imprimir recursivamente desde la raíz hacia las hojas, de forma depth-first pre-order. El nodo inicial se imprime como `> node: <valor>`. Luego, para imprimir cada nodo se adicionarán dos espacios por nivel (la raíz tiene nivel 0, sus hijos estan en el nivel 1, los hijos de sus hijos en el 2 y así sucesivamente), luego el string `--> node:` y finalmente el valor. El valor impreso dependerá del `type` del árbol, ya sea, un `Integer`, un `Double` o una `String` de C.

Finalmente se imprimirá una línea con la cadena `--------`.

Ejemplo: `tree_print(t, "el arbol es:", "arbol.txt")` para el árbol de la figura 1 imprime en `arbol.txt`:

```
el arbol es:
> node: 10
--> node: 7
--> node: 60
  --> node: 18
  --> node: 12
--------
```

Funciones avanzadas sobre árboles

Previo a presentar el comportamiento de las funciones avanzadas cabe mencionar las siguientes definiciones de tipos.

```
typedef enum boolean { False=0, True=1 } boolean;
```

La misma sirve para indicar un valor de verdad.

```
typedef boolean (*tree_bool_method)(tree* self);
```

Representa al tipo de los punteros a funciones que reciben un árbol como argumento y devuelven un booleano.

```
typedef tree_value (*tree_value_method)(tree *parent, tree *child);
```

Representa al tipo de los punteros a funciones que reciben dos árboles (padre e hijo) como argumento y devuelven un valor. Las funciones avanzadas a implementar son las siguientes:

- `void tree_prune(tree *self, tree_bool_method *bool_method);`
  Toma un árbol y una función que devuelve booleanos. Evalúa la función para cada hijo del árbol. Si el resultado es "false" entonces conserva el elemento evaluado y evalúa recursivamente a sus hijos. En caso contrario borra el elemento, sus nodos, y a sus árboles hijos recursivamente.
  Ej. `tree_prune(t, &es_bisiesto )`,
  donde `t` es el árbol de la figura 1, y `es_bisiesto` es una función que dado un puntero a un árbol de enteros retorna si el valor que contiene el nodo corresponde a un año bisiesto

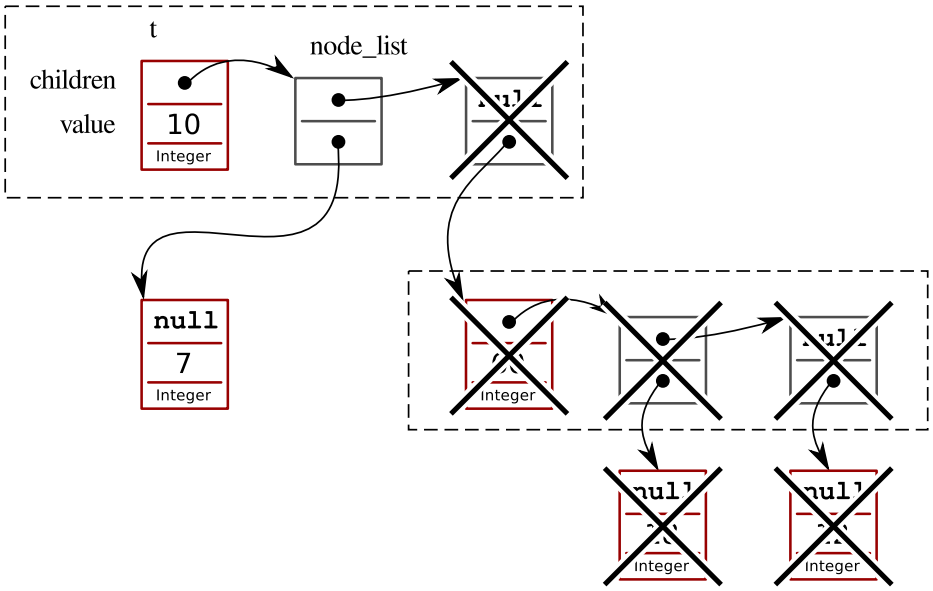


Figura 2: Operación de pruning

- `void tree_merge(tree *self, boolean_method merge_test, tree_value_method *value_method );`
  La función `tree_merge` toma un árbol y dos funciones, que aplicará a cada hijo del mismo. Para cada *hijo* primero aplica merge_test, y en caso de que devuelva "False" evalúa recursivamente a los hijos. En caso contrario, deberá *unir* al hijo con el padre. Esta unión se hará pasando todos los hijos del hijo al padre, y poniendo como valor del padre al resultado de evaluar value_method con el padre y el hijo como argumentos. En caso de unir padre con hijo **NO** se debe aplicar merge a los hijos del hijo. Además, la memoria correspondiente al nodo que desaparece al unirse con su padre **debe** ser liberada.
  Ej. `tree_merge(tree, &es_bisiesto, &sumar );` donde sumar devuelve la suma entera del valor del padre y del hijo.

Funciones adicionales

Las funciones avanzadas requieren como parámetros otras funciones. A continuación se enumeran las únicas 6 funciones que serán utilizadas como parámetros. Las tres primeras se usarán para las funciones prune y merge, mientras que las tres restantes sólo para la función merge.

- `enum boolean_e es_bisiesto(tree *self);`
  Toma un puntero a un árbol de *int*s y retorna un valor de verdad que indica si en ese nodo el número corresponde a un año bisiesto[1].

- `enum boolean_e es_mayor_que_sesenta(tree *self);`
  Toma un puntero a un árbol de *double*s y retorna un valor de verdad que indica si en ese nodo el número es mayor que sesenta.

[1] Un año es bisiesto si es múltiplo de 4, a menos que sea múltiplo de 100 y no de 400

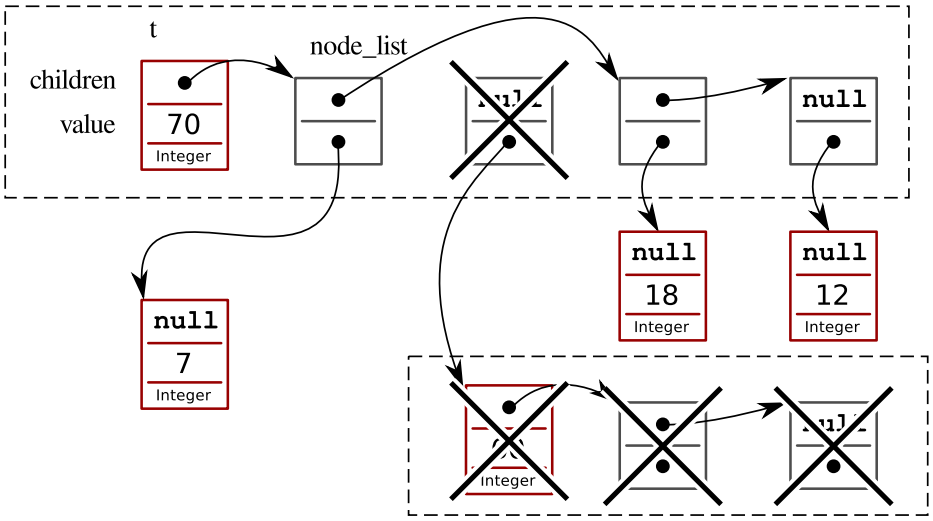


Figura 3: Operación merge

- `enum boolean_e tiene_vocales(tree *self);`
  Toma un árbol de *strings* de C y retorna un valor de verdad que indica si en ese nodo la cadena de caracteres contiene vocales.

- `int sumar(tree *parent, tree *child);`
  Toma dos punteros a tree y retorna un int que representa al resultado de sumar el valor de ambos nodos.

- `double multiplicar(tree *parent, tree *child);`
  Toma dos punteros a tree y retorna un double que representa a la multiplicacion del valor de ambos nodos.

- `char* intercalar(tree *parent, tree *child);`
  Toma dos punteros a tree y retorna una **nueva** string, que es el resultado de intercalar las letras del padre con el hijo de la forma detallada a continuación. En los indices pares se pone la letra correspondiente al padre en el mismo índice. En los indices impares lo mismo pero poniendo la letra correspondiente al hijo. Finalmente, si una cadena es mas larga que la otra, la cadena final se completa con las letras de la cadena de mayor tamaño. Ej. Intercalar de padre "hola" y e hijo "zaraza" da como resultado "halaza". Además debe liberar la memoria de las strings anteriores.

2. Enunciado

Ejercicio 1

Deberá implementar una serie de funciones en assembler que se enumeran a continuación.

- Funciones de tree
  - `tree* tree_create(tree_value value, tree_type type)`
    - `tree* tree_create_int(int value)`

    - `tree* tree_create_double(double value)`
    - `tree* tree_create_string(char *heap_str)`
  - `void tree_add_child(tree *self, tree *child)`
  - `void tree_deep_delete(tree *self)`
  - `void tree_print(tree *self, char* extra, char *archivo)`
- Funciones avanzadas de tree
  - `void tree_prune(tree *self, tree_bool_method method)`
  - `tree* tree_merge(tree *self, tree_bool_method *merge_test_method, tree_value_method *value_method)`
- Funciones auxiliares para tree_prune y tree_merge
  - `boolean es_bisiesto(tree *self);`
  - `boolean es_mayor_que_sesenta(tree *self);`
  - `boolean tiene_vocales(tree *self);`
- Funciones auxiliares para tree_merge
  - `int sumar(tree *parent, tree *child)`
  - `double multiplicar(tree *parent, tree *child)`
  - `char* intercalar(tree *parent, tree *child)`

Para implementar todo lo anterior de una manera concisa y fácil de entender, se recomienda *muy fuertemente* el uso de funciones auxiliares. Se enumeran aquí algunas recomendadas. Queda a criterio de cada uno si desea implementar lo anterior usando éstas.

- `list_node_p* tree_last_children_pointer(tree *self)`. Devuelve un *puntero al puntero* que apunta al último hijo.
- `int tree_children_count(tree *self)`. Devuelve el número de hijos que tiene ese árbol.
- `void tree_print_value_at_level(tree* self, FILE *file, int level)` Imprime el valor de `self` asumiendo que es un árbol en el nivel `level`
- `void tree_print_level(tree *self, FILE *file, int level)` Imprime el valor de `self` en el nivel `level` y se llama recursivamente con todos los hijos.
- `void list_node_deep_delete(list_node **node_pointer)` Borra un nodo que apunta a un arbol, el árbol apuntado y todos sus nodos e hijos recursivamente.
- `void tree_shallow_delete(tree *self)` Borra un árbol y los nodos que apuntan a sus hijos, pero sin borrar a los hijos del árbol.
- `void list_node_shallow_delete(list_node **node_pointer)` Recibe un *puntero al puntero* a un nodo y borra el nodo apuntado.
- `void merge(tree *parent, list_node **node_pointer, tree_value_method value_method)` Recibe un árbol padre, y un *puntero al puntero* a un nodo (hijo), que será mergeado al padre, guardando en el padre el valor que resulte de evaluar value_method con padre e hijo.

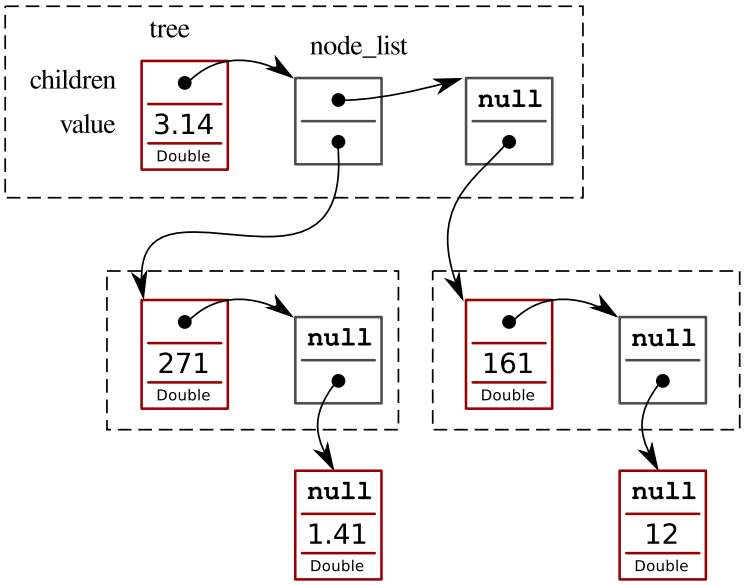


Figura 4: Árbol de doubles

Ejercicio 2

Construir un programa de prueba (`test.c`) que realice las siguientes acciones llamando a las funciones implementadas anteriormente:

1- Crear un árbol nuevo con el valor entero 10 (`tree *t = tree_create_int(10)`).

2- Crear un árbol como el de la figura 1 y aplicarle la función `tree_prune`, pasando como parámetro la función que verifica si es bisiesto.

3- Crear un árbol como el de la figura 1 y aplicarle la función `tree_merge`, pasando como parámetro la función que verifica si es bisiesto y la que suma al valor del padre con el del hijo.

4- Imprimir todos los árboles.

5- Destruir todos los árboles.

Realizar el mismo procedimiento para los siguientes árboles, usando `es_mayor_que_sesenta` / `multiplicar` para el árbol de doubles y `tiene_vocales` / `intercalar` para el de strings.

Testing

En un ataque de bondad, hemos decidido entregarle una serie de *tests* o pruebas para que usted mismo pueda verificar el buen funcionamiento de su código.

Luego de compilar, puede ejecutar `./tests.sh` y eso probará su código. Un test consiste en la creación, inserción, eliminación e impresión en archivo de una gran cantidad de árboles. Luego de cada test, el script comparará los archivos generados por su TP con las soluciones correctas provistas por la cátedra.

También será probada la correcta administración de la memoria dinámica.

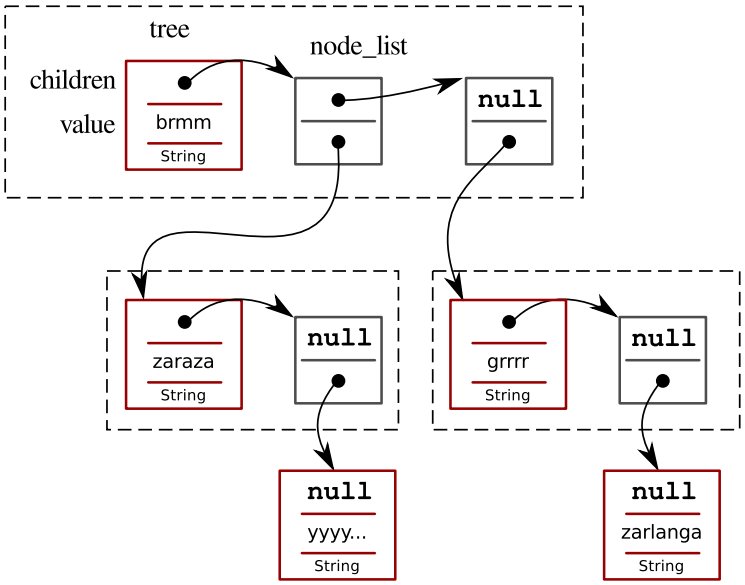


Figura 5: Árbol de strings

Archivos

Se entregan los siguientes archivos:

- `tp1.cpp`: Es el archivo principal, contiene el método `main`. No debe modificarlo.
- `tree.h`: Contiene la definición de la estructura de la lista, los elementos y las funciones que debe completar.
- `tree.asm`: Archivo a completar con su código.
- `Makefile`: Contiene las instrucciones para compilar el código.
- `tests.sh`: Es el script que realiza todos los test.

Notas:

a) Todas las funciones deben estar en lenguaje ensamblador. Cualquier función extra, también debe estar hecha en lenguaje ensamblador.

b) Toda la memoria dinámica reservada por la función `malloc` debe ser correctamente liberada, utilizando la función `free`.

c) Para el manejo de archivos se recomienda usar las funciones de C: `fopen`, `fread`, `fwrite`, `fclose`, `fseek`, `ftell`, `fprintf`, etc.

d) Para poder correr los test, se debe tener instalado *Valgrind* (En ubuntu: `sudo apt-get install valgrind`).

e) Para corregir los TPs usaremos los mismos tests que les fueron entregados. El criterio de aprobación es que el TP supere correctamente todos los tests.

3. Resolución informe y forma de entrega

Este trabajo práctico consta carácter de **individual**. No deberán entregar un informe. En cambio, deberán entregar un archivo comprimido con el mismo contenido que el dado para realizarlo, habiendo modificado solo archivo `tree.asm`, y agregado un archivo `main.c` con el segundo ejercicio resuelto. Además se deberá adjuntar un *Makefile* para compilar este último.

La fecha de entrega de este trabajo es Martes 23 de Abril. Deberá ser entregado a través de la página web. El sistema sólo aceptará entregas de trabajos hasta las 17:00 hs del día de entrega.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes.