Programación II, 2016-2017

Escuela Politécnica Superior, UAM

Práctica 4: TAD Árbol

PARTE 2A: PREGUNTAS SOBRE LA PRÁCTICA

Ejercicio #1:

El árbol que se crea a partir de los ficheros de nodos (dict*.dat), ¿es completo o casi completo? Justifica tu respuesta.

Dependiendo del programa que usemos. Si usamos el programa *p4_e2* en estos ficheros no nos da un árbol ni completo ni casi completo. Sin embargo, si usamos el *p4_e2_bal,* nos dan árboles casi completos, y si el número de nodos fuese un número que cumpliese la condición $nNodos = 2^{n+1}$-1 para algún natural n, entonces nos generaría un árbol completo.

Ejercicio #2:

a) ¿Qué relación hay entre la "forma" de un árbol y sus recorridos?

Sus recorridos variarán dependiendo de la forma del árbol, y si nos encontramos ante un árbol creado por p4_e2_bal, el nodo raíz tenderá a estar en el medio del recorrido *tree_inOrder.*

b) ¿Se puede saber si un árbol binario de búsqueda está bien construido según sus recorridos?

Si, bastaría con recorrerlo en *tree_inOrder* y comprobar que la lista de números nos sale ordenada de menor a mayor. Si lo está, entonces estará bien construido, en caso contrario no.

Ejercicio #3:

Compara y describe las diferencias entre los árboles generados por los ejecutables p4_e2 y p4_e2_bal (número de nodos, profundidad, recorridos, etc.).

El generado por *p4_e2_bal* va a tener el mismo número de nodos que el generado por *p4_e2*, la profundidad del balanceado va a ser menor o igual que la del *p4_e2*, y siempre la mínima posible. Los recorridos son distintos, tendiendo el nodo del árbol balanceado a estar en el medio del recorrido *tree_inOrder.*

1. Decisiones de diseño

Ejercicio #1:

La implementación del TAD Tree (como árbol binario de búsqueda) ha resultado ser la parte más compleja de la práctica. La gran mayoría de las funciones requieren una implementación con una función recursiva, para lo que elegimos separar la recursión en una función aparte.

La nomenclatura elegida para estas funciones depende del nombre de la función original. La función recursiva de *tree_insert,* es *nodeBT_insert_recursive*. Al realizar la separación de la función recursiva y la original, tenemos que pasar como argumento las funciones de la generalización del árbol necesarias para ello.

Finalmente, tuvimos que crear dos estructuras de datos:

```
struct _Tree {
    NodeBT *root;
    destroy_elementtree_function_type destroy_element_function;
    copy_elementtree_function_type copy_element_function;
    print_elementtree_function_type print_element_function;
    cmp_elementtree_function_type cmp_element_function;
};
```

```
typedef struct _NodeBT {
    void* info;
    struct _NodeBT* left;
    struct _NodeBT* right;
} NodeBT;
```

Ejercicio #2:

En este ejercicio se nos pedía modificar el TAD Node para que el campo del nombre se guardara en memoria dinámica, para mejorar el uso de la misma.

Además, tuvimos que realizar cambios en la función *node_setName* y *node_cpy*, debido a la nueva implementación.

Para probarlo se nos pedía realizar un *main* que probara el TAD. Hemos repetido el ejercicio *p1_e1.c*, para ejecutarlo ha de escribirse en la consola: *./p4_e2_test*

Posteriormente se nos pidió que usáramos los programas y datos adjuntos para la comprobación del TAD Tree y del TAD Node. Para ello, creamos la función *node_cmp* y la añadimos a la interfaz. A su vez modificamos *cmp_node_function* de *functions.c* para que llamase a esta función.

También modificamos el programa dado y sustituimos *tree_free* por *tree_destroy,* para que se adecuase a la nomenclatura de nuestro TAD. Por último incorporamos parte del makefile proporcionado al nuestro.

Se adjunta la nueva estructura de datos de nodo, así como el pseudocódigo de la función de comparación que se usará en el *quickSort* de *p4_e2_bal.*

```
struct _Node {
    char *name;
    int id;
    int fatherId;
    Color color;
};
```

```
int node_cmp(Node n1, Node n2)
{
    vId ← id[n1]-id[n2];
    if vId ≠ 0 do
    {
        return vId;
    }
    vName ← strcmp(name[n1] - name[n2]);
    return vId + vName;
}
```

Ejercicio #3:

En esta parte de la práctica implementamos 3 funciones para recorrer el árbol, y otras 3 para añadir los elementos a una lista.

Implementamos las funciones separando la recursión de la función principal. En estas funciones simplemente teníamos que recorrer el árbol dependiendo del orden, *preOrder* (raíz – rama izquierda – rama derecha), *inOrder* (rama izquierda – raíz – rama derecha), y *postOrder* (rama izquierda – rama derecha – raíz).

El pseudocódigo de estas funciones es sencillo, simplemente recorre en el orden indicado hasta que llega a un nodo nulo, en cuyo caso vuelve al paso anterior.

Para la implementación de las funciones que añaden el elemento a la lista, el orden de adición a la lista es inverso al orden del recorrido, es decir, recorremos a la inversa. Esto se debe a que preferimos usar *list_insertFirst* ya que consume muchos menos recursos con el inconveniente de dejarnos una lista volteada, que hemos solucionado al recorrer de forma inversa.

Ejercicio #4:

Finalmente, en el cuarto ejercicio hemos implementado funciones en *functions.c* para que trabajase con el tipo de dato *char** o "*strings*" con almacenamiento en memoria dinámica. La mayoría de las funciones ya estaban creadas e implementadas en *string.h* y únicamente hemos añadido el control de errores y la hemos referenciado.

Por otro lado hemos modificado el *p4_e1* para que leyese elementos de tipo "*string*" y realizamos lo pedido en las instrucciones del PDF.

2. Informe del uso de memoria.

Se adjuntan los reportes de valgrind sin la salida del programa por motivos de extensión.

Ejercicio #1:

```
valgrind --leak-check=full ./p4_e1 numeros.txt
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 19 allocs, 19 frees, 6,932 bytes allocated

All heap blocks were freed -- no leaks are possible

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio #2:

```
valgrind --leak-check=full ./p4_e2_test
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 7 allocs, 7 frees, 1,115 bytes allocated

All heap blocks were freed -- no leaks are possible

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Se adjuntan los reportes de valgrind sin la salida del programa por motivos de extensión.

```
valgrind --leak-check=full ./p4_e2 dict10.dat
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 80 allocs, 80 frees, 13,101 bytes allocated

All heap blocks were freed -- no leaks are possible

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
valgrind --leak-check=full ./p4_e2_bal dict10.dat
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 80 allocs, 80 frees, 13,101 bytes allocated

All heap blocks were freed -- no leaks are possible

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio #3:

```
valgrind --leak-check=full ./p4_e3 numeros.txt
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 64 allocs, 64 frees, 7,472 bytes allocated

All heap blocks were freed -- no leaks are possible

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ejercicio #4:

```
valgrind --leak-check=full ./p4_e4 cadenas.txt
HEAP SUMMARY:
    in use at exit: 0 bytes in 0 blocks
  total heap usage: 21 allocs, 21 frees, 6,984 bytes allocated

All heap blocks were freed -- no leaks are possible

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

3. Conclusiones finales:

La práctica ha resultado beneficiosa para la comprensión de la abstracción de los TADs, y de cómo si queremos hacer un cambio, solo tendremos que modificar un fichero por lo general.

Se ha visto reforzado el uso de la memoria así como el control de errores y la referenciación de punteros, muy presente en las clases teóricas.

La parte más compleja de la práctica es la implementación del TAD Tree, aunque tampoco resultó demasiado difícil. El resto de la práctica ha resultado ser de un nivel bastante asequible.

Como aspectos nuevos de programación podemos recoger tanto el uso de un nuevo TAD, como el uso de Makefiles y control de entrada al programa.