



# C++ Pool - d02a

Koalab [koala@epitech.eu](mailto:koala@epitech.eu)

*Abstract: This document is the subject of d02a*

# Contents

I	General remarks	2
II	Exercice 0	3
III	Exercice 1	7
IV	Exercice 2	10
V	Exercice 3	14
VI	Exercice 4	17
VII	Exercice 5	20
VIII	Exercice 6	25


# Chapter I

## General remarks

- If you do half the exercises because you are having trouble with them, it's all right. However, if you do half the exercises because you're lazy and you leave at 2PM, you WILL be surprised. Do NOT try your luck.
- File names are to be respected TO THE LETTER, as well as function names.
- Turn-in directories are ex00, ex01, etc.
- Read the examples CAREFULLY. They might require things the subject doesn't tell...
- Read ENTIRELY the subject of an exercise before you start it !
- THINK. Please.
- THINK. By Odin !
- T.H.I.N.K ! For Pony !
- Notice none of your files should contain a "main" function, unless it is specifically requested. We will use our own "main" function to compile and try your code.

# Chapter II

## Exercise 0

	Exercise : 00	points : 2
Simple List - Create a simple list		
Turn-in directory: (piscine_cpp_d02a)/ex00		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror	
Makefile: No	Rules: n/a	
Files to turn in : simple_list.c		
Remarks : The 'simple_list.h' file is given to you, you have to use it without modifying it		
Forbidden functions : None		

The purpose of this exercise is to create some functions allowing you to handle a list. We consider that our list is defined like this:

```

1     typedef struct s_node
2     {
3         double value;
4         struct s_node *next;
5     } t_node;
6
7     typedef t_node *t_list;
```

An empty list is represented by a NULL pointer. We are also going to define the following type (representing a boolean):

```

1     typedef enum e_bool
2     {
3         FALSE,
4         TRUE
5     } t_bool;
```

Here are the functions you need to implement:

- Informative functions (File `simple_list.c` )
  - `unsigned int list_get_size(t_list list);`  
Takes a list as its parameter and returns the number of elements contained in the list.
  - `t_bool list_is_empty(t_list list);`  
Takes a list as its parameter, returns `TRUE` if the list is empty, `FALSE` otherwise.
  - `void list_dump(t_list list);`  
Takes a list as its parameter and displays every element of the list, separated by a new-line character. Use the default display of `printf` (`%f`) with no particular precision.
- Modification functions (File `simple_list.c` )
  - `t_bool list_add_elem_at_front(t_list *front_ptr, double elem);`  
Adds a new node at the beginning of the list with `'elem'` as a value. The function returns `FALSE` if it cannot allocate the new node, `TRUE` otherwise.
  - `t_bool list_add_elem_at_back(t_list *front_ptr, double elem);`  
Adds a new node at the end of the list with `'elem'` as a value. The function returns `FALSE` if it cannot allocate the new node, `TRUE` otherwise.
  - `t_bool list_add_elem_at_position(t_list * front_ptr, double elem, unsigned int position);`  
Adds a new node at the position `'position'` with `'elem'` as a value. If the value of `'position'` is 0, a call to this function is equivalent to a call to `'list_add_elem_at_front'`.  
The function returns `FALSE` if it cannot allocate the new node or if `'position'` is invalid, `TRUE` otherwise.
  - `t_bool list_del_elem_at_front(t_list *front_ptr);`  
Deletes the first node of the list.  
Returns `FALSE` if the list is empty, `TRUE` otherwise.
  - `t_bool list_del_elem_at_back(t_list *front_ptr);`  
Deletes the last node of the list.  
Returns `FALSE` if the list is empty, `TRUE` otherwise.
  - `t_bool list_del_elem_at_position(t_list *front_ptr, unsigned int position);`  
Deletes the node at position `'position'` . If the value of `'position'` is 0, a call to this function is equivalent to a call to `'list_del_elem_at_front'` . Returns `FALSE` if the list is empty or if `'position'` is invalid, `TRUE` otherwise.


- Value access functions (File `simple_list.c` )
  - `double list_get_elem_at_front(t_list list);` Returns the value of the first node in the list. Returns 0 if the list is empty.
  - `double list_get_elem_at_back(t_list list);` Return the value of the last node of the list. Returns 0 if the list is empty.
  - `double list_get_elem_at_position(t_list list, unsigned int position);` Returns the value of the node at the position `'position'` .  
If the value of `'position'` is 0, a call to this function is equivalent to a call to `'list_get_elem_at_front'` . Returns 0 if the list is empty or if `'position'` is invalid.
- Access functions (File `simple_list.c` )
  - `t_node *list_get_first_node_with_value(t_list list, double value);` Returns a pointer to the first node of the list `'list'` having the value `'value'` . If no node matches the value, the function returns `NULL` .

Here is an example of a main function with the expected output :

```
1 int main(void)
2 {
3     t_list list_head = NULL;
4     unsigned int size;
5     double i = 5.2;
6     double j = 42.5;
7     double k = 3.3;
8
9     list_add_elem_at_back(&list_head, i);
10    list_add_elem_at_back(&list_head, j);
11    list_add_elem_at_back(&list_head, k);
12
13    size = list_get_size(list_head);
14    printf("Il y a %u elements dans la liste\n", size);
15    list_dump(list_head);
16
17    list_del_elem_at_back(&list_head);
18
19    size = list_get_size(list_head);
20    printf("Il y a %u elements dans la liste\n", size);
21    list_dump(list_head);
22 return (0);
23 }
24
25 $> ./a.out
26 Il y a 3 elements dans la liste
27 5.200000
28 42.500000
29 3.300000
30 Il y a 2 elements dans la liste
31 5.200000
32 42.500000
33 $>
```

# Chapter III

## Exercice 1

	Exercise : 01	points : 3
Simple BTree - Create a simple tree		
Turn-in directory: (piscine_cpp_d02a)/ex01		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror	
Makefile: No	Rules: n/a	
Files to turn in : simple_btree.c		
Remarks : You have to use the provided file 'simple_btree.h' without any modification.		
Forbidden functions : None		

The purpose of this exercise is to create some functions that will allow you to use a binary tree.

We assume our binary tree is defined as follows:

```

1      typedef struct s_node
2      {
3          double value;
4          struct s_node *left;
5          struct s_node *right;
6      } t_node;
7
8      typedef t_node *t_tree;
```

An empty tree is represented by a `NULL` pointer.



Here are the functions you have to code:

- Informations functions (File `simple_btree.c` )

- `t_bool btree_is_empty(t_tree tree);`  
Returns `TRUE` if the tree `'tree'` is empty, `FALSE` otherwise.
- `unsigned int btree_get_size(t_tree tree);`  
Returns the number of nodes contained in the tree `'tree'`.
- `unsigned int btree_get_depth(t_tree tree);`  
Returns the depth of the tree `'tree'`.

- Modification functions (File `simple_btree.c` )

- `t_bool btree_create_node(t_tree *node_ptr, double value);`  
This function creates a new node and places it at the location pointed by `'node_ptr'` . The value of the node is `'value'` .  
The function returns `FALSE` if the node could not be added, `TRUE` otherwise.
- `t_bool btree_delete(t_tree *root_ptr);`  
This function deletes the `TREE` pointed by `'root_ptr'`. (In its entirety - children nodes included)  
This functions returns `FALSE` if the tree is empty, `TRUE` otherwise.

- Access functions (File `simple_btree.c` )


- `double btree_get_max_value(t_tree tree);`  
This function returns the maximal value contained in the tree `'tree'` . Returns 0 if the tree is empty.
- `double btree_get_min_value(t_tree tree)` This function returns the minimal value contained in the tree `'tree'` . Returns 0 if the tree is empty.

Here is an example of a main function with the expected output :

```
1 int main(void)
2 {
3     t_tree tree = NULL;
4     t_tree left_sub_tree;
5     unsigned int size;
6     unsigned int depth;
7     double max;
8     double min;
9
10    btree_create_node(&tree, 42.5);
11    btree_create_node(&(tree->right), 100);
12    btree_create_node(&(tree->left), 20);
13
14    left_sub_tree = tree->left;
15
16    btree_create_node(&(left_sub_tree->left), 30);
17    btree_create_node(&(left_sub_tree->right), 5);
18
19    size = btree_get_size(tree);
20    depth = btree_get_depth(tree);
21
22    printf("L'arbre a une taille de %u\n", size);
23    printf("L'arbre a une profondeur de %u\n", depth);
24
25    max = btree_get_max_value(tree);
26    min = btree_get_min_value(tree);
27
28    printf("Les valeurs de l'arbre vont de %f a %f\n", min, max);
29
30    return (0);
31 }
32
33
34 $> ./a.out
35 L'arbre a une taille de 5
36 L'arbre a une profondeur de 3
37 Les valeurs de l'arbre vont de 5.000000 a 100.000000
38 $>
```

# Chapter IV

## Exercice 2

	Exercise : 02	points : 3
Generic List - Create a generic list		
Turn-in directory: (piscine_cpp_d02a)/ex02		
Compiler: gcc	Compilation flags: -Wextra -Werror -Wall	
Makefile: No	Rules: n/a	
Files to turn in : generic_list.c		
Remarks : You have to use the provided file 'generic_list.h' without any modification.		
Forbidden functions : None		

The purpose of this exercise is to create a generic list.

The difference with the exercise 'Simple List' is that a node is defined like this:

```

1     typedef struct s_node
2     {
3         void *value;
4         struct s_node *next;
5     } t_node;
6
7     typedef t_node *t_list;
```

The functions you have to code are the same with small differences in their prototypes.

```

1     unsigned int list_get_size(t_list list);
2     t_bool list_is_empty(t_list list);
3
4     t_bool list_add_elem_at_front(t_list *front_ptr, void *elem);
5     t_bool list_add_elem_at_back(t_list *front_ptr, void *elem);
6     t_bool list_add_elem_at_position(t_list *front_ptr, void *elem,
7     unsigned int position);
8
```

```
9      t_bool list_del_elem_at_front(t_list *front_ptr);
10     t_bool list_del_elem_at_back(t_list *front_ptr);
11     t_bool list_del_elem_at_position(t_list *front_ptr,
12     unsigned int position);
13
14     void list_clear(t_list *front_ptr);
15
16     /*Cette fonction libere tout les noeuds de la liste,
17     et reinitialise la liste pointee par 'front_ptr' a une
18     liste vide.*/
19
20     void *list_get_elem_at_front(t_list list);
21     void *list_get_elem_at_back(t_list list);
22     void *list_get_elem_at_position(t_list list, unsigned int position);
```

Only two functions are really different:

- `typedef void (*t_value_displayer)(void *value);`  
`void list_dump(t_list list, t_value_displayer val_disp);`

The function 'list\_dump' now takes a function pointer of type 't\_value\_displayer' as its second parameter.

Using the function pointed by 'val\_disp', we can now display the value 'value' contained in a node followed by a newline.

- `typedef int (*t_value_comparator)(void *first, void *second);`  
`t_node *list_get_first_node_with_value(t_list list, void *value,`  
`t_value_comparator val_comp);`

The function 'list\_get\_first\_node\_with\_value' now takes a function pointer of type 't\_value\_comparator', which allows us to compare two values of the list.

The comparison function returns a positive value if 'first' is greater than 'second', a negative value if 'second' is greater than 'first', and 0 if 'first' equals 'second'.

Here is an example of a main function with the expected output :

```
1 void int_displayer(void *data)
2 {
3     int value;
4
5     value = *((int *)data);
6     printf('%d\n', value);
7 }
8 int int_comparator(void *first, void *second)
9 {
10    int val1;
11    int val2;
12
13    val1 = *((int *)first);
14    val2 = *((int *)second);
15    return (val1 - val2);
16 }
17 int main(void)
18 {
19     t_list list_head = NULL;
20     unsigned int size;
21     int i = 5;
22     int j = 42;
23     int k = 3;
24
25     list_add_elem_at_back(&list_head, &i);
26     list_add_elem_at_back(&list_head, &j);
27     list_add_elem_at_back(&list_head, &k);
28
29     size = list_get_size(list_head);
30     printf('Il y a %u elements dans la liste\n', size);
31     list_dump(list_head, &int_displayer);
32
33     list_del_elem_at_back(&list_head);
34
35     size = list_get_size(list_head);
36     printf('Il y a %u elements dans la liste\n', size);
37     list_dump(list_head, &int_displayer);
38
39     return (0);
40 }
41 $> ./a.out
42 Il y a 3 elements dans la liste
43 5
44 42
45 3
46 Il y a 2 elements dans la liste
47 5
48 42
49 $>
```




## C++ Pool - d02a

---



# Chapter V

## Exercice 3

	Exercise : 03	points : 2
Stack - Create a stack		
Turn-in directory: (piscine_cpp_d02a)/ex03		
Compiler: gcc	Compilation flags: -Wextra -Werror -Wall	
Makefile: No	Rules: n/a	
Files to turn in : stack.c, generic_list.c		
Remarks : You have to use the provided files 'stack.h' and 'generic_list.h' without any modification.		
Forbidden functions : None		

A code built around another code is called a Wrapper.

The purpose of this exercise is to create a stack based on the previously created generic list.

Use the previously coded 'generic\_list.c' file without any modification.

You may have guessed it, we will consider that a stack is a list which has smart features limitations, therefore we have :

```
1 typedef t_list t_stack;
```

Here is the list of functions to code :

- Information functions (File `stack.c` )
  - `unsigned int stack_get_size(t_stack stack);`  
Returns the number of elements in the stack.
  - `t_bool stack_is_empty(t_stack stack);`  
Returns `TRUE` if the stack is empty, `FALSE` otherwise.

- Modification functions (File `stack.c` )

- `t_bool stack_push(t_stack *stack_ptr, void *elem);`

Pushes the element `'elem'` on the top of the stack. Returns `FALSE` if the new element can not be pushed, `TRUE` otherwise.

- `t_bool stack_pop(t_stack *stack_ptr);`

Pops the element of the top of the stack. Returns `FALSE` if the stack is empty, `TRUE` otherwise.

- Access functions (File `stack.c` )

- `void *stack_top(t_stack stack);`

Returns the value of the element on the top of the stack.




Here is an example of a main function with the expected output :

```
1 int main(void)
2 {
3     t_stack stack = NULL;
4     int i = 5;
5     int j = 4;
6     int *data;
7
8     stack_push(&stack, &i);
9     stack_push(&stack, &j);
10
11     data = (int *)stack_top(stack);
12
13     printf('%d\n', *data);
14
15     return (0);
16 }
17
18 $> ./a.out
19 4
20 $>
```

# Chapter VI

## Exercise 4

	Exercise : 04	points : 2
Queue - Create a queue		
Turn-in directory: (piscine_cpp_d02a)/ex04		
Compiler: gcc	Compilation flags: -Wextra -Werror -Wall	
Makefile: No	Rules: n/a	
Files to turn in : queue.c, generic_list.c		
Remarks : You have to use the the provided files 'queue.h' and 'generic_list.h' without any modification.		
Forbidden functions : None		

A code built around another code is called a Wrapper.

The purpose of this exercise is to create a queue based on the previously created generic list.

Use the previously coded 'generic\_list.c' file without any modification.

You may have guessed it again, we will consider that a queue is a list which has smart features limitations, therefore we have :

```
1 typedef t_list t_queue;
```

Here is the list of functions to code :

- Information functions (File queue.c )
  - unsigned int queue\_get\_size(t\_queue queue);  
Returns the number of elements in the queue.
  - t\_bool queue\_is\_empty(t\_queue queue);  
Returns TRUE if the queue is empty, FALSE otherwise.


- Modification functions (File `queue.c` )
  - `t_bool queue_push(t_queue *queue_ptr, void *elem);`  
Pushes the element '`elem`' in the queue. Returns `FALSE` if the new element cannot be pushed, `TRUE` otherwise.
  - `t_bool queue_pop(t_queue *queue_ptr);`  
Pops the next element from the queue. Returns `FALSE` if the queue is empty, `TRUE` otherwise.
- Access functions (File `queue.c` )
  - `void *queue_front(t_queue queue);` Returns the value of the next element in the queue.

Here is an example of a main function with the expected output :

```
1 int main(void)
2 {
3     t_queue queue = NULL;
4     int i = 5;
5     int j = 4;
6     int *data;
7
8     queue_push(&queue, &i);
9     queue_push(&queue, &j);
10
11     data = (int *)queue_front(queue);
12
13     printf('%d\n', *data);
14
15     return (0);
16 }
17
18 $> ./a.out
19 5
20 $>
```

# Chapter VII

## Exercice 5

	Exercise : 05	points : 3
Map - Create a map		
Turn-in directory: (piscine_cpp_d02a)/ex05		
Compiler: gcc	Compilation flags: -Wextra -Werror -Wall	
Makefile: No	Rules: n/a	
Files to turn in : generic_list.c, map.c		
Remarks : You have to use the the provided files 'map.h' and 'generic_list.h' without any modification.		
Forbidden functions : None		

A code built around another code is called a Wrapper.

The purpose of this exercise is to create a map (associative array) based on the previously created generic list.

Use the previously coded 'generic\_list.c' file without any modification.

You may have guessed it again, we will consider that a map is a list which has smart features limitations, therefore we have :

```
1 typedef t_list t_map;
```

The main question is: "A map is a list of what ?!" The answer is :

```
1
2     typedef struct s_pair
3     {
4         void *key;
5         void *value;
6     } t_pair;
```



Think about it...

Here is the list of the functions you have to code :

- Informations functions (File `map.c` )
  - `unsigned int map_get_size(t_map map);` Returns the number of elements in the map.
  - `t_bool map_is_empty(t_map map);` Returns `TRUE` if the map is empty, `FALSE` otherwise.

Here comes the tricky part.

Because our map is generic, the key '`key`' may contain any data type. To be able to compare these data and to know if a key is equal to another one (among else), we need a function pointer pointing to a key comparator :

```
1 typedef int (*t_key_comparator)(void *first_key, void *second_key);
```

Returns 0 if the keys are equal, a positive number if '`first_key`' is greater than '`second_key`' and a negative number if '`second_key`' is greater than '`first_key`' .

The generic list uses the same function pointer system to find a node with a particular value.

So the question is now “how to make the function called by our list call the key comparison function, knowing that we cannot add new parameters ?”

Two solutions :

- A global variable
- A wrapper around a global variable ;)

Because we love pretty code, we will choose the second solution.

So you will have to code the following two functions (File `map.c` ) :

```
1 t_key_comparator key_cmp_container(t_bool store, t_key_comparator
2                                   new_key_cmp);
```

This function store a static variable of type `t_key_comparator` . If `'store'` has the value `TRUE` , the new value of the static variable is `'new_key_cmp'` . The function always returns the value contained in the static variable. This simulates the behaviour of a global variable : if you want to store a value, call this function with `TRUE` as its first argument and the value to store. If you want to access the value, call this function with `FALSE` as its first argument and `NULL` as its second.

```
1 int pair_comparator(void *first, void *second);
```

This function takes two values of our list (`void *`) , which are in reality pointers on pairs (`t_pair *`) as arguments. This function compares only the keys contained in these pairs. It returns `0` if the keys are equal, a positive value if the key of `'first'` is greater than the key of `'second'` , and a negative value if the key of `'second'` is greater than the key of `'first'`.

Before going back to our map, we will add a basic function to our generic list.

- Upgrading the generic list (File `generic_list.c` )
  - `t_bool list_del_node(t_list *front_ptr, t_node *node_ptr);` This function deletes the node pointed by `'node_ptr'` from the list. This function returns `FALSE` if the node is not in the list.

Now back to the map.

- Modification functions (File `map.c` )
  - `t_bool map_add_elem(t_map *map_ptr, void *key, void *value, t_key_comparator key_cmp);`  
This function adds the value `'value'` at the index `'key'` of the map. If a value already exists at the index `'key'` , this value is replaced by the new one.  
`'key_cmp'` is to be called to compare the keys of the map. Returns `FALSE` if the element could not be added, `TRUE` otherwise.

- `t_bool map_del_elem(t_map *map_ptr, void *key, t_key_comparator key_cmp);`

This function deletes the value at the index `'key'` . `'key_cmp'` is to be called to compare the keys of the map. Returns `FALSE` if there is no value at the index `'key'` , `TRUE` otherwise.

- Access functions (File `map.c` )

- `void *map_get_elem(t_map map, void *key, t_key_comparator key_cmp);`

This function returns the value contained at the index `'key'` of the map. If there is no value at the index `'key'` , this function returns `NULL` . `'key_cmp'` is to be called to compare the keys of the map.




Here is an example of a main function with the expected output :

```
1 int int_comparator(void *first, void *second)
2 {
3     int val1;
4     int val2;
5
6     val1 = *(int *)first;
7     val2 = *(int *)second;
8     return (val1 - val2);
9 }
10
11 int main(void)
12 {
13     t_map map = NULL;
14     int first_key = 1;
15     int second_key = 2;
16     int third_key = 3;
17     char *first_value = "first";
18     char *first_value_rw = "first_rw";
19     char *second_value = "second";
20     char *third_value = "third";
21     char **data;
22
23     map_add_elem(&map, &first_key, &first_value, &int_comparator);
24     map_add_elem(&map, &first_key, &first_value_rw, &int_comparator);
25     map_add_elem(&map, &second_key, &second_value, &int_comparator);
26     map_add_elem(&map, &third_key, &third_value, &int_comparator);
27
28     data = (char **)map_get_elem(map, &second_key, &int_comparator);
29     printf("A la clef [%d] se trouve la valeur [%s]\n", second_key, *data);
30
31     return (0);
32 }
33
34 $> ./a.out
35 A la clef [2] se trouve la valeur [second]
36 $>
```

# Chapter VIII

## Exercice 6

You have to use the provided files `'tree_traversal.h'` , `'stack.h'` , `'queue.h'` and `'generic_list.h'` without any modification

	Exercise : 06	points : 5
Tree Traversal - Iterating is human...		
Turn-in directory: (piscine_cpp_d02a)/ex06		
Compiler: gcc	Compilation flags: -Wextra -Werror -Wall	
Makefile: No	Rules: n/a	
Files to turn in : tree_traversal.c, stack.c, queue.c, generic_list.c		
Remarks : n/a		
Forbidden functions : None		

The purpose of this exercise is to iterate over a tree in a generic way using containers. This is the definition of our tree :

```

1      typedef struct s_tree_node
2      {
3          void *data;
4          struct s_tree_node *parent;
5          t_list children;
6      } t_tree_node;
7
8      typedef t_tree_node *t_tree;
```

Where `'data'` is the data contained in the node, `'parent'` is a pointer to the parent node and `'children'` is a generic list of child nodes.

An empty tree is represented by a `NULL` pointer.

Here is the list of functions to code :

- Information functions (File `tree_traversal.c` )
  - `t_bool tree_is_empty(t_tree tree);` This function returns `TRUE` if the tree is empty, `FALSE` otherwise.
  - `void tree_node_dump(t_tree_node *tree_node, t_dump_func dump_func);` This function is used to display the content of a node. To do this, the first argument is a pointer to a node and the second is a function pointer to a display function defined with the following type: `typedef void (*t_dump_func)(void *data);`
- Modification function (File `tree_traversal.c` )
  - `t_bool init_tree(t_tree *tree_ptr, void *data);` This function initializes the tree pointer by `'tree_ptr'` by creating a root node having for value `'data'` . Returns `FALSE` if the root node could not be allocated, `TRUE` otherwise.
  - `t_tree_node *tree_add_child(t_tree_node *tree_node, void *data)` This function adds a child node to the node pointed by `'tree_ptr'` . The child node will have the value `'data'` . The function returns `NULL` if the child node could not be added, otherwise it returns a pointer to the child node.
  - `t_bool tree_destroy(t_tree* tree_ptr);` This function deletes the tree pointed by `'tree_ptr'` . (Meaning the node pointed by `'tree_ptr'` and all the child nodes). It sets back the value pointed by `'tree_ptr'` to be an empty tree ( `NULL` ). It returns `FALSE` if it fails, `TRUE` otherwise.
- Tree traversal (File `tree_traversal.c` )

To code the ultimate function we need to define a generic container with the following definition:

```
1      typedef struct s_container
2      {
3          void *container;
4          t_push_func push_func;
5          t_pop_func pop_func;
6      } t_container;
```

with :

```
1 typedef t_bool (*t_push_func)(void *container, void *data);  
2 typedef void* (*t_pop_func)(void *container);
```

The structure `container` represents a generic container, the field `'container'` stores the adress of the real container. The field `'push_func'` is a function pointer allowing us to insert an element in the container. The field `'pop_func'` is a function pointer allowing us to extract an element from the container.

Now, here is the ultimate function to code :

```
void tree_traversal(t_tree tree, t_container *container, t_dump_func dump_func);
```

This function allows us to iterate over the tree `'tree'` and display its content using the container `'container'` . The function pointed by `'dump_func'` is to be called to display the data of the tree.

To do this, each node of the tree has to insert its child nodes in the container, displays itself, and starts this over with the next node being the one we extract from the container.



Your display will go from left to right with a FIFO container and from right to left with a LIFO container, it is normal.

```
1 Here is a main example with the expected output :
2
3 void dump_int(void *data)
4 {
5     printf("%d\n", *(int *)data);
6 }
7
8 t_bool generic_push_stack(void *container, void *data)
9 {
10     return (stack_push((t_stack *)container, data));
11 }
12
13 void *generic_pop_stack(void *container)
14 {
15     void *data;
16
17     data = stack_top(*(t_stack *)container);
18     stack_pop((t_stack *)container);
19     return (data);
20 }
21
22 t_bool generic_push_queue(void *container, void *data)
23 {
24     return (queue_push((t_queue *)container, data));
25 }
26
27 void *generic_pop_queue(void *container)
28 {
29     void *data;
30
31     data = queue_front(*(t_queue *)container);
32     queue_pop((t_queue *)container);
33     return (data);
34 }
35
36 int main(void)
37 {
38     t_tree tree = NULL;
39     t_tree_node *node;
40     int val_0 = 0;
41     int val_a = 1;
42     int val_b = 2;
43     int val_c = 3;
44     int val_aa = 11;
45     int val_ab = 12;
46     int val_ca = 31;
47     int val_cb = 32;
48     int val_cc = 33;
49
50     t_container container;
```

```
51  t_stack stack = NULL;
52  t_queue queue = NULL;
53
54  init_tree(&tree, &val_0);
55  node = tree_add_child(tree, &val_a);
56
57  tree_add_child(node, &val_aa);
58  tree_add_child(node, &val_ab);
59
60  tree_add_child(tree, &val_b);
61  node = tree_add_child(tree, &val_c);
62
63  tree_add_child(node, &val_ca);
64  tree_add_child(node, &val_cb);
65  tree_add_child(node, &val_cc);
66
67  printf("Parcours en Profondeur :\n");
68
69  container.container = &stack;
70  container.push_func = &generic_push_stack;
71  container.pop_func = &generic_pop_stack;
72
73  tree_traversal(tree, &container, &dump_int);
74
75  printf("Parcours en Largeur :\n");
76
77  container.container = &queue;
78  container.push_func = &generic_push_queue;
79  container.pop_func = &generic_pop_queue;
80
81  tree_traversal(tree, &container, &dump_int);
82
83  return (0);
84 }
85
86
87 $>./a.out
88 Parcours en Profondeur :
89 0
90 3
91 33
92 32
93 31
94 2
95 1
96 12
97 11
98 Parcours en Largeur :
99 0
100 1
101 2
```

```
102 3
103 11
104 12
105 31
106 32
107 33
108 $>
```