B3 - C++ Pool

B-PAV-242

Day 02

Afternoon

KOALA

42.0

{EPITECH.}
L'ECOLE DE L'INNOVATION ET DE L'EXPERTISE INFORMATIQUE

Day 02

binary name:	no binary
group size:	1
repository name:	cpp_d02a
repository rights:	ramassage-tek
language:	C

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).

General Setpoints

READ THESE CAREFULLY
You will have no possible excuse if you end up with a 0 because you didn't follow one of these.

If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercices because you're lazy, and leave at 2PM, you **WILL** have problems. Do not tempt the devil.

Read the examples CAREFULLY. They might require things that weren't mentioned in the subject...

THINK. Please.

THINK

T.H.I.N.K.! For Pony!

To avoid compilation problems during automated tests, please include all necessary files within your headers.

Please note that none of your files must contain a `main` function, unless specified otherwise. We will use our own `main` functions to compile and test your code.

This subject may be modified up to one hour before turn-in time!

UNIT TESTS

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the **"How to write Unit Tests"** document on the intranet, available here.

Create a directory named `tests`. For each of the functions you turn in, create a file in that directory named `tests-FUNCTION_NAME.c` containing all the tests needed to cover all of the exercise's possible cases (regular or irregular).

Here is a sample set of unit tests for the **my_strlen** function:

```c
#include <criterion/criterion.h>

Test(my_strlen, positive_return_value)
{
    cr_assert_eq(my_strlen("toto"), 4);
}

Test(my_strlen, empty_string)
{
    cr_assert_eq(my_strlen(""), 0);
}
```

Exercise 0 - Simple List

	Exercise: 00	points : 2
Simple List - Create a simple list		
Turn-in directory: cpp_d02a/ex00		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror	
Makefile: No	Rules: n/a	
Files to turn in: simple_list.c		
Notes: The 'simple_list.h' file is provided. You must use it without modifying it		
Forbidden functions: None		

The purpose of this exercise is to create a set of functions that will let you manipulate a list. We will consider a list as the following:

```
typedef struct s_node
{
    double value;
    struct s_node *next;
} t_node;

typedef t_node * t_list;
```

An empty list is represented by a NULL pointer.
Let's define the following type, representing a boolean:

```
typedef enum e_bool
{
    FALSE,
    TRUE
} t_bool;
```

Implement the following functions:

- Informative functions:

```
unsigned int list_get_size(t_list list);
```

Returns the number of elements in the list.

```
t_bool list_is_empty(t_list list);
```

Returns TRUE if the list is empty, FALSE otherwise.

```
void list_dump(t_list list);
```

Displays every element in the list, separated by new-line characters. Use the default display of printf (%f) with no particular precision.

- Modification functions:

```
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 its value. The function returns `FALSE` if it cannot allocate memory for 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 its value. The function returns `FALSE` if it cannot allocate memory for 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 its 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 memory for the new node or if `position` is out of bounds, `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 the `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 out of bounds, `TRUE` otherwise.

- Value access functions:

```
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);
```

Returns the value of the last node in 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 out of bounds.

- Access functions

```
t_node *list_get_first_node_with_value(t_list list, double value);
```

Returns a pointer to the first node of `list` having `value` as its value. If no node matches `value`, the function returns `NULL`.

Here is a sample `main` function and its expected output:

```c
int main(void)
{
    t_list list_head = NULL;
    double i = 5.2;
    double j = 42.5;
    double k = 3.3;

    list_add_elem_at_back(&list_head, i);
    list_add_elem_at_back(&list_head, j);
    list_add_elem_at_back(&list_head, k);

    unsigned int size = list_get_size(list_head);
    printf("Il y a %u elements dans la liste\n", size);
    list_dump(list_head);

    list_del_elem_at_back(&list_head);

    size = list_get_size(list_head);
    printf("Il y a %u elements dans la liste\n", size);
    list_dump(list_head);
    return 0;
}
```

main.c

Terminal

```
~/B-PAV-242> ./a.out
Il y a 3 elements dans la liste
5.200000
42.500000
3.300000
Il y a 2 elements dans la liste
5.200000
42.500000
```

Exercise 1 - Simple BTree

KOALA	Exercise: 01	points : 3
Simple BTree - Create a simple tree		
Turn-in directory: cpp_d02a/ex01		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror	
Makefile: No	Rules: n/a	
Files to turn in: simple_btree.c		
Notes: You have to use the provided 'simple_btree.h' file without modifying it		
Forbidden functions: None		

The purpose of this exercise is to create a set of functions that will let you manipulate a binary tree. We will consider a binary tree as the following:

```
typedef struct s_node
{
    double value;
    struct s_node *left;
    struct s_node *right;
} t_node;

typedef t_node * t_tree;
```

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

Implement the following functions:

- Informative functions:

```
t_bool btree_is_empty(t_tree tree);
```

Returns `TRUE` if `tree` is empty, `FALSE` otherwise.

```
unsigned int btree_get_size(t_tree tree);
```

Returns the number of nodes in `tree`.

```
unsigned int btree_get_depth(t_tree tree);
```

Returns the depth of `tree`.

- Modification functions:

```
t_bool btree_create_node(t_tree *node_ptr, double value);
```

Creates a new node with `value` as its value and places it at the location pointed to by `node_ptr`. Returns `FALSE` if the node could not be added, `TRUE` otherwise.

```
t_bool btree_delete(t_tree *root_ptr);
```

Deletes the **TREE** pointed to by `root_ptr` in its entirety, including its children. The function returns `FALSE` if the tree is empty, `TRUE` otherwise.

- Access functions:

```
double btree_get_max_value(t_tree tree);
```

Returns the maximal value in `tree`. Returns 0 if the tree is empty.

```
double btree_get_min_value(t_tree tree);
```

Returns the minimal value in `tree`. Returns 0 if the tree is empty.

Here is a sample `main` function with its expected output:

```c
int main(void)
{
  t_tree tree = NULL;

  btree_create_node(&tree, 42.5);
  btree_create_node(&(tree->right), 100);
  btree_create_node(&(tree->left), 20);

  t_tree left_sub_tree = tree->left;

  btree_create_node(&(left_sub_tree->left), 30);
  btree_create_node(&(left_sub_tree->right), 5);

  unsigned int size = btree_get_size(tree);
  unsigned int depth = btree_get_depth(tree);

  printf("L'arbre a une taille de %u\n", size);
  printf("L'arbre a une profondeur de %u\n", depth);

  double max = btree_get_max_value(tree);
  double min = btree_get_min_value(tree);

  printf("Les valeurs de l'arbre vont de %f a %f\n", min, max);

  return (0);
}
```

main.c

Terminal

```
~/B-PAV-242> ./a.out
L'arbre a une taille de 5
L'arbre a une profondeur de 3
Les valeurs de l'arbre vont de 5.000000 a 100.000000
```

Exercise 2 - Generic List

KOALA	Exercise: 02	points : 3
Generic List - Create a generic list		
Turn-in directory: `cpp_d02a/ex02`		
Compiler: `gcc`	Compilation flags: `-Wall -Wextra -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `generic_list.c`		
Notes: `You have to use the provided 'generic_list.h' file without modifying it`		
Forbidden functions: `None`		

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

The difference between this and the `Simple List` exercise is that a node is defined like this:

```
typedef struct s_node
{
    void *value;
    struct s_node *next;
} t_node;

typedef t_node * t_list;
```

The functions you have to implement are similar, with some minor differences in their prototypes:

```
unsigned int list_get_size(t_list list);
t_bool list_is_empty(t_list list);

t_bool list_add_elem_at_front(t_list *front_ptr, void *elem);
t_bool list_add_elem_at_back(t_list *front_ptr, void *elem);
t_bool list_add_elem_at_position(t_list *front_ptr, void *elem,
                                 unsigned int position);

t_bool list_del_elem_at_front(t_list *front_ptr);
t_bool list_del_elem_at_back(t_list *front_ptr);
t_bool list_del_elem_at_position(t_list *front_ptr, unsigned int position);

void list_clear(t_list *front); // Releases all nodes in the list and
                                // makes 'front_ptr' point to an empty list

void *list_get_elem_at_front(t_list list);
void *list_get_elem_at_back(t_list list);
void *list_get_elem_at_position(t_list list, unsigned int position);
```

Only two functions truly differ:

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

`list_dump` now takes a `t_value_displayer` function pointer as its second parameter. Using the function pointed to by `val_disp`, it is now possible to display the `value` of each node, followed by a newline.

```c
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);
```

`list_get_first_node_with_value` now takes a `t_value_comparator` function pointer as its second parameter, which lets you 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` and `second` are equal.

Here is a sample `main` function with its expected output:

```c
void int_displayer(void *data)
{
    int value = *((int *)data);
    printf("%d\n", value);
}

int int_comparator(void *first, void *second)
{
    int val1 = *((int *)first);
    int val2 = *((int *)second);
    return (val1 - val2);
}

int main(void)
{
    t_list list_head = NULL;
    int i = 5;
    int j = 42;
    int k = 3;

    list_add_elem_at_back(&list_head, &i);
    list_add_elem_at_back(&list_head, &j);
    list_add_elem_at_back(&list_head, &k);

    unsigned int size = list_get_size(list_head);
    printf("Il y a %u elements dans la liste\n", size);
    list_dump(list_head, &int_displayer);

    list_del_elem_at_back(&list_head);

    size = list_get_size(list_head);
    printf("Il y a %u elements dans la liste\n", size);
    list_dump(list_head, &int_displayer);

    return 0;
}
```

main.c

Terminal

```
∼/B-PAV-242> ./a.out
Il y a 3 elements dans la liste
5
42
3
Il y a 2 elements dans la liste
5
42
```

Exercise 3 - Stack

	Exercise: 03	points : 2
Stack - Create a stack		
Turn-in directory: cpp_d02a/ex03		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror	
Makefile: No	Rules: n/a	
Files to turn in: stack.c, generic_list.c		
Notes: You have to use the provided ‘stack.h‘ and ‘generic_list.h‘ files without modifying them		
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 `generic_list.c` file from the previous exercises without modifying it.

As you may have guessed, we will consider a stack as a list which has smart feature limitations. Therefore:

```c
typedef t_list t_stack;
```

Implement the following functions:

- Informative functions:

```c
unsigned int stack_get_size(t_stack stack);
```

Returns the number of elements in the stack.

```c
t_bool stack_is_empty(t_stack stack);
```

Returns `TRUE` if the stack is empty, `FALSE` otherwise.

- Modification functions:

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

Pushes `elem` to the top of the stack. Returns `FALSE` if the new element could not be pushed, `TRUE` otherwise.

```c
t_bool stack_pop(t_stack *stack_ptr);
```

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

- Access functions:

```c
void *stack_top(t_stack stack);
```

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

Here is a sample `main` function and its expected output:

```c
int main(void)
{
    t_stack stack = NULL;
    int i = 5;
    int j = 4;

    stack_push(&stack, &i);
    stack_push(&stack, &j);

    int *data = (int *)stack_top(stack);

    printf("%d\n", *data);

    return (0);
}
```

main.c

Terminal

```
~/B-PAV-242> ./a.out
4
```

Exercise 4 - Queue

	Exercise: 04	points : 2
Queue - Create a queue		
Turn-in directory: `cpp_d02a/ex04`		
Compiler: `gcc`	Compilation flags: `-Wall -Wextra -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `queue.c, generic_list.c`		
Notes: `You have to use the provided 'queue.h' and 'generic_list.h' files without modifying them`		
Forbidden functions: `None`		

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

Use the `generic_list.c` file from the previous exercicses without modifying it.

As you may have guessed again, we will consider a queue as a list with some smart feature limitations. Therefore:

```
typedef t_list t_queue;
```

Implement the following functions:

- Informative functions:

```
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:

```
t_bool queue_push(t_queue *queue_ptr, void *elem);
```

Pushes `elem` into 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:

```
void *queue_front(t_queue queue);
```

Returns the value of the next element in the queue.

Here is a sample `main` function and its expected output:

```c
int main(void)
{
    t_queue queue = NULL;
    int i = 5;
    int j = 4;

    queue_push(&queue, &i);
    queue_push(&queue, &j);

    int *data = (int *)queue_front(queue);

    printf("%d\n", *data);

    return 0;
}
```

main.c

Terminal

```
~/B-PAV-242> ./a.out
5
```

Exercise 5 - Map

(Map - Create a map

	Exercise: 05	points : 3
Turn-in directory: `cpp_d02a/ex05`		
Compiler: `gcc`	Compilation flags: `-Wall -Wextra -Werror`	
Makefile: `No`	Rules: `n/a`	
Files to turn in: `map.c, generic_list.c`		
Notes: `You have to use the provided 'map.h' and 'generic_list.h' files without modifying them`		
Forbidden functions: `None`		

The purpose of this exercise is to create a map (which you may know as an associative array) based on the previously create generic list.

Use the `generic_list.c` file from the previous exercises without modifying it.

Once again, you may have guessed it: we will consider a map as a list with some smart feature limitations. Therefore:

```
typedef t_list t_map;
```

The remaining question you may have is: "What is a map a list of?". Well, here's the answer:

```
typedef struct s_pair
{
    void *key;
    void *value;
} t_pair;
```

Think about it...

Implement the following functions:

- Informative functions:

```c
unsigned int map_get_size(t_map map);
```

Returns the number of elements in the map.

```c
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` may contain any data type. To be able to compare these data and know whether two keys are equal (among other things), we need a key comparator:

```c
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`.

If you remember correctly, our generic list uses the same function pointer system to find a node with a particular value.

The question now is "How can we make the function called by our list call the key comparison function when we cannot add new parameters?".
There are two solutions to this problem:

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

Because we love nice and maintainable code, we will choose the second solution.
Implement the following functions:

```c
t_key_comparator key_cmp_container(t_bool store, t_key_comparator new_key_cmp);
```

Holds a static `t_key_comparator`. If `store` is set to `TRUE`, the value of the static variable must be set to `new_key_cmp`. The function always returns the value of the static variable. This simulates the behavior of a global variable: if you want to set its value, call this function with `TRUE` as its first parameter and the value as its second. If you want to access the value, call this function with `FALSE` as its argument and `NULL` as its second.

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

The two parameters are values from the list which point to `t_pairs`. Compares the keys in each pair. Returns 0 if the keys are equal, a positive value if the key of `first` is greater than that of `second`, and a negative value if the key of `second` is greater than that of `first`.

Before we go back to our map, add a basic function to the generic list.

Implement this function in `generic_list.c`:

```c
t_bool list_del_node(t_list *front_ptr, t_node *node_ptr);
```

Deletes `node_ptr` from the list. Returns `FALSE` if the node is not in the list, `TRUE` otherwise.
Now back to the map (in `map.c`):

- Modification functions:

```c
t_bool map_add_elem(t_map *map_ptr, void *key, void *value);
```

Adds `value` at the `key` index of the map. If a value already exists at the `key` index, it is replaced by `value`. `key_cmp` is to be called to compare the keys of the map. Returns `FALSE` if the element could not be added, `TRUE` otherwise.

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

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

- Access functions:

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

Returns the value held at the `key` index of the map. If there is no value at the `key` index, returns `NULL`. `key_cmp` is to be called to compare the keys of the map.

Here is a sample `main` function and its expected output:

```c
int int_comparator(void *first, void *second)
{
    int val1 = *(int *)first;
    int val2 = *(int *)second;
    return (val1 - val2);
}

int main(void)
{
    t_map map = NULL;
    int first_key = 1;
    int second_key = 2;
    int third_key = 3;
    char *first_value = "first";
    char *first_value_rw = "first_rw";
    char *second_value = "second";
    char *third_value = "third";

    map_add_elem(&map, &first_key, &first_value, &int_comparator);
    map_add_elem(&map, &first_key, &first_value_rw, &int_comparator);
    map_add_elem(&map, &second_key, &second_value, &int_comparator);
    map_add_elem(&map, &third_key, &third_value, &int_comparator);

    char **data = (char **)map_get_elem(map, &second_key, &int_comparator);
    printf("A la clef [%d] se trouve la valeur [%s]\n", second_key, *data);

    return 0;
}
```

Terminal

```
~/B-PAV-242> ./a.out
A la clef [2] se trouve la valeur [second]
```

Exercise 6 - Tree Traversal

	Exercise: 06	points : 5
Tree Traversal - Iterating is human...		
Turn-in directory: cpp_d02a/ex06		
Compiler: gcc	Compilation flags: -Wall -Wextra -Werror	
Makefile: No	Rules: n/a	
Files to turn in: tree_traversal.c, stack.c, queue.c, generic_list.c		
Notes: You have to use the provided 'tree_traversal.h'		
Forbidden functions: None		

The purpose of this exercise is to iterate over a tree in a generic way, using containers. Here is how we'll define a tree:

```
typedef struct s_tree_node
{
    void *data;
    struct s_tree_node *parent;
    t_list children;
} t_tree_node;

typedef t_tree_node * t_tree;
```

`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.

Implement the following functions:

- Informative functions:

```
t_bool tree_is_empty(t_tree tree);
```

Returns `TRUE` if the tree is empty, `FALSE` otherwise.

```c
void tree_node_dump(t_tree_node *t_tree_node, t_dump_func dump_func);
```

Displays the content of a node. The first argument is a pointer to a node, and the second is a function pointer to a display function defined like this:

```c
typedef void (*t_dump_func)(void *data);
```

- Modification functions:

```c
t_bool init_tree(t_tree *tree_ptr, void *data);
```

Initializes `tree_ptr` by creating a root node holding `data`. Returns `FALSE` if the root node could not be allocated, `TRUE` otherwise.

```c
t_tree_node *tree_add_child(t_tree_node *tree_node, void *data);
```

Adds a child node holding `data` to `tree_node`. Returns a pointer to the child node, or `NULL` if the child node could not be added.

```c
t_bool tree_destroy(t_tree *tree_ptr);
```

Deletes `tree_ptr`, including all its children. Resets `tree_ptr` to an empty tree. Returns `FALSE` if it fails, `TRUE` otherwise.

- Tree traversal:

To code the ultimate function, we need to define a generic container:

```c
typedef struct s_container
{
    void *container;
    t_push_func push_func;
    t_pop_func pop_func;
} t_container;

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

`t_container` is a generic container. The `container` field holds the adress of the actual container. `push_func` is a function pointer that inserts an element in the container. `pop_func` is a function pointer that extracts an element from the container.
Here is the ultimate function you must implement:

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

Iterates over `tree` and displays its content using `container` and `dump_func`.

To do this, each node of the tree has to insert its child nodes in the container, display itself, and start over with the next node, extrated from the container.

> Output must go from left to right with a FIFO container and from right to left with a LIFO container

Here is a sample `main` function and its expected output:

```c
void dump_int(void *data)
{
  printf("%d\n", *(int *)data);
}

t_bool generic_push_stack(void *container, void *data)
{
  return stack_push((t_stack *)container, data);
}

void *generic_pop_stack(void *container)
{
  void *data = stack_top(*(t_stack *)container);
  stack_pop((t_stack *)container);
  return data;
}

t_bool generic_push_queue(void *container, void *data)
{
  return queue_push((t_queue *)container, data);
}

void *generic_pop_queue(void *container)
{
  void *data = queue_front(*(t_queue *)container);
  queue_pop((t_queue *)container);
  return data;
}
```

funcs.c

```c
int main(void)
{
  int val_0 = 0;
  int val_a = 1;
  int val_b = 2;
  int val_c = 3;
  int val_aa = 11;
  int val_ab = 12;
  int val_ca = 31;
  int val_cb = 32;
  int val_cc = 33;

  t_tree tree = NULL;
  init_tree(&tree, &val_0);
  t_tree_node node = tree_add_child(tree, &val_a);

  tree_add_child(node, &val_aa);
  tree_add_child(node, &val_ab);
  tree_add_child(tree, &val_b);

  node = tree_add_child(tree, &val_c);
  tree_add_child(node, &val_ca);
  tree_add_child(node, &val_cb);
```

```c
	tree_add_child(node, &val_cc);

	t_container container;

	printf("Parcours␣en␣Profondeur␣:\n");

	t_stack stack = NULL;
	container.container = &stack;
	container.push_func = &generic_push_stack;
	container.pop_func = &generic_pop_stack;
	tree_traversal(tree, &container, &dump_int);

	printf("Parcours␣en␣Largeur␣:\n");

	t_queue queue = NULL;
	container.container = &queue;
	container.push_func = &generic_push_queue;
	container.pop_func = &generic_pop_queue;
	tree_traversal(tree, &container, &dump_int);

	return 0;
}
```

main.c

Terminal

```
~/B-PAV-242> ./a.out
Parcours en Profondeur :
0
3
33
32
31
2
1
12
11
Parcours en Largeur :
0
1
2
3
11
12
31
32
33
```