KOALA

B3- C++ Pool

B-PAV-242

Day 02 - Afternoon

Pointers and Memory

v1.5

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

Day 02 - Afternoon

Pointers and Memory

repository name:	cpp_d02a
repository rights:	ramassage-tek
language:	C
group size:	1

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

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

Every function implemented in a header, or unprotected header, leads to 0 to the exercise.
Every class must possess a constructor and a destructor.

Every output goes to the standard output, and will be ended by a newline, unless specified otherwise.

Any use of "friend" will result in the grade -42 no questions asked.

To avoid compilation problems, please include necessary files within your headers.

Please note that none of your files must contain the main function except when explicitly asked. We will use our own main function to compile and test your code.

Exercise 00

Simple List - Create a Simple List

repository subdir:	/ex00
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	simple_list.c
forbidden functions:	none
points:	2

You must use the provided **"simple_list.h"** file, without any modification.

The purpose of this exercise is to create some functions that will allow you to manage a list.

Our list is defined as follows:

```
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.
We will also define the following type, which represents a boolean:

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

Here are the functions you should implement (in the `simple_list.c` file):

- **Informative functions**

  ```
  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 and returns TRUE if the list is empty. Otherwise, it returns FALSE.

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

  takes a list as its parameter and displays every element of the list, separated by a newline character. Use the **printf(%f)** default display, without any 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 a value. The function returns FALSE if it cannot allocate the new node. Otherwise, it returns TRUE.

```
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. Otherwise, it returns TRUE.

```
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. Otherwise, it returns TRUE.

```
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. Otherwise, it returns TRUE.

```
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. Otherwise, it returns TRUE.

```
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 invalid. Otherwise, it returns TRUE.

- **Value access functions**

```
double list_get_elem_at_front(t_list list);
```

returns the value of the first node of 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 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**

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

returns a pointer to the first node of **"list"** that has 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:

```c
int                main(void)
{
    t_list          list_head = NULL;
    unsigned int    size;
    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);

    size = list_get_size(list_head);
    printf("There are %u elements on the list\n", size);
    list_dump(list_head)

    list_del_elem_at_back(&list_head);

    size = list_get_size(list_head);
    printf("There are %u elements on the list\n", size);
    list_dump(list_head);

    return (0);
}
```

Terminal

```
~/B-PAV-242> ./a.out
There are 3 elements on the list
5.200000
42.500000
3.300000
There are 2 elements on the list
5.200000
42.500000
```

Exercise 01

Simple BTree - Create a Simple Tree

repository subdir:	/ex01
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	simple_btree.c
forbidden functions:	none
points:	2

You must use the provided **"simple_btree.h"** file, without any modification.

The purpose of this exercise is to create some functions that will allow you to manage a binary tree.
The binary tree is defined as follows:

```
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

Here are the functions you must implement (in the `simple_btree.c`):

- **Information functions**

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

  returns TRUE if **"tree"** is empty, and FALSE otherwise.

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

  returns the number of nodes contained 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 and places it at the location pointed to by **"node_ptr"**. The value of the node is **"value"**.
  The function returns FALSE if the node could not be added and TRUE otherwise.

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

deletes the tree pointed to by **"root_ptr"**, in its entirety - child nodes included. The function returns FALSE if the tree is empty and TRUE otherwise.

- **Access functions**

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

returns the maximum value contained in **"tree"**. Returns 0 if the tree is empty.

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

returns the minimum value contained in **"tree"**. Returns 0 if the tree is empty.

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

```c
int             main(void)
{
    t_tree          tree = NULL
    t_tree          left_sub_tree;
    unsigned int    size;
    unsigned int    depth;
    double          max;
    double          min;

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

    left_sub_tree = tree->left;

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

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

    printf("The size of the tree is: %u\n", size);
    printf("The depth of the tree is: %u\n", depth);

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

    printf("The values of the tree go from %f to %f\n", min, max);

    return (0);
}
```

Terminal

```
~/B-PAV-242> ./a.out
The size of the tree is:  5
The depth of the tree is:  3
The tree values go from 5.000000 to 100.000000
```

Exercise 02

Generic List - Create a Generic List

repository subdir:	/ex02
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	generic_list.c
forbidden functions:	none
points:	3

You must use the provided **"generic_list.h"** file without any modification.

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 as follows:

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

typedef t_node          *t_list;
```

The functions you have to code are the same, with only slight 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_ptr);
  /*This function frees up all of the list"s knots, and reinitializes the pointed list from
      "front_ptr" 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 are really different:

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

The **"list_dump"** function now takes a **"t_value_displayer"** function pointer as its second parameter.
Using the function pointed to 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 **"list_get_first_node_with_value"** function now takes a **"t_value_comparator"** type function pointer, which allows us to compare two of the list''s values.

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 main function with the expected output:

```
void    int_displayer(void *data)
{
    int   value;

    value = *((int *)data);
    printf("%d\n", value);
}
int     int_comparator(void *first, void *second)
{
    int   val1;
    int   val2;

    val1 = *((int *)first);
    val2 = *((int *)second);
    return (val1 - val2);
}
int             main(void)
{
    t_list      list_head = NULL;
    unsigned int size;
    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);

    size = list_get_size(list_head);
    printf("There are %u elements in the list\n", size);
    list_dump(list_head, &int_displayer);

    list_del_elem_at_back(&list_head);
    size = list_get_size(list_head);
    printf("There are %u elements in the list\n", size);
    list_dump(list_head, &int_displayer);

    return (0);
}
```

Terminal

```
~/B-PAV-242> ./a.out
There are 3 elements in the list
5
42
3
There are 2 elements in the list
5
42
```

Exercise 03

Stack - Create a Stack

repository subdir:	/ex03
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	stack.c, generic_list.c
forbidden functions:	none
points:	2

You must use the provided **"stack.h"** and **"generic_list.h"** files without any modification.

A code built around another code is called a Wrapper.
The purpose of this exercise is to create a stack based on the previously-generated generic list.
You may have guessed it: a stack is defined as a list with smart feature limitations. Therefore, we have:

```
typedef t_list t_stack;
```

Here is a list of functions to implement (in the `stack.c` file):

- **Information functions**

  ```
  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, and FALSE otherwise.

- **Modification functions**

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

  pushes **"elem"** to the top of the stack. Returns FALSE if the new element cannot be pushed, and TRUE otherwise.

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

  pops the element on top of the stack. Returns FALSE if the stack is empty, and TRUE otherwise.

- **Access functions**

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

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

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

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

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

    data = (int *)stack_top(stack);
    printf("%d\n", *data);

    return (0);
}
```

Terminal

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

Exercise 04

Queue - Create a Queue

repository subdir:	/ex04
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	queue.c generic_list.c
forbidden functions:	none
points:	2

You must use the provided **"queue.h"** and **"generic_list.h"** files without any modification.

As we saw in the previous exercise, a code built around another code is called a Wrapper.
The purpose of this exercise is to create a queue that is based on the previously-created generic list.
You may have guessed it again: a queue is defined as a list with smart feature limitations. Therefore, we have:

```
typedef t_list t_queue;
```

Here is a list of functions to implement (in the `queue.c` file):

- **Information 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, and 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, and TRUE otherwise.

  ```
  t_bool queue_pop(t_queue *queue_ptr);
  ```

  pops the next element from the queue. Returns FALSE if the queue is empty, and TRUE otherwise.

- **Access functions**

  ```
  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:

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

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

    data = (int *)queue_front(queue);

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

    return (0);
}
```

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

Exercise 05

Map - Create a Map

repository subdir:	/ex05
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	generic_list.c, map.c
forbidden functions:	none
points:	3

You must use the provided **"map.h"** and **"generic_list.h"** files without any modification.

You know by now that a code built around another code is called a Wrapper.
The purpose of this exercise is to create a map (associative array) that is based on the previously-created generic list.
Use the previously-coded **"generic_list.c"** file without any modfications.
I'm sure you've guessed again: a map is defined as a list with smart feature limitations. Therefore, we have:

```
typedef t_list t_map;
```

The real question is: "A map is a list of what?!".
Here's the answer:

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

Think about it...

Here is a list of the functions you have to implement (in the `map.c` file):

- **Information functions**

  ```
  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, and FALSE otherwise.

Here comes the tricky part...
Because our map is generic, the **"key"** key could contain any data type. In order to be able to compare the data and find out if one key is equal to another (among other things), we need a function pointer that points to a key comparator:

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

This 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 specific value.
So, now the question is: "how can we make the function that is called by our list call the key comparison function? Keeping in mind that we cannot add new parameters.

There are two solutions:

- **a global variable**,
- **a wrapper around a global variable**.

Since we love pretty code, we will choose the second solution.
So, now you have to code the two following functions (in the `map.c` file):

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

This function stores a static **t_key_comparator** variable.
If **"store"** has the TRUE value, the static variable's new value is set to **"new_key_cmp"**.
The function always returns the value that is contained in the static variable. This simulates the behavior of a global variable: if you want to store a value, call this function with TRUE as its first argument and the value to be stored.
If you want to access the value, call this function with FALSE as its first argument and NULL as its second.

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

This function takes two **void** * parameters, which, in reality, are pointers to **t_pair** *. This function only compares the keys contained in these pairs. It returns 0 if the keys are equal, a positive value if **"first"**'s key is greater than **"second"**'s, and a negative value if **"second"**'s' is greater than **"first"**'s'.

Before going back to our map, we will add a basic function to our generic list (in the `generic_list.c` file):

- **Upgrading the generic list**

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

  deletes the node pointed to by **"node_ptr"** from the list. It returns FALSE if the node is not in the list.

Now back to the map (in the `map.c` file):

- **Modification functions**

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

  adds **"value"** at the **"key"** index of the map. If a value already exists at the **"key"** index, it will be replaced by the new one. **"key_cmp"** should be called to compare the map's keys. It returns FALSE if the element could not be added, and TRUE otherwise.

  ```
  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"** should be called to compare the map's keys. It returns FALSE if there is no value at the **"key"** index, and TRUE otherwise.

- **Access functions**

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

  returns the value contained at the map's **"key"** index. If there is no value at the **"key"** index, this function returns NULL. **"key_cmp"** should be called to compare the map''s keys.

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

```c
int      int_comparator(void *first, void *second);
{
    int   val1;
    int   val2;

    val1 = *(int *)first;
    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"";
    char            **data;

    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);

    data = (char **)map_get_elem(map, &second_key, &int_comparator);
    printf("The [%d] key corresponds to the value [%s]\n", second_key, *data);

    return (0);
}
```

Terminal

```
~/B-PAV-242> ./a.out
The [2] key corresponds to the value [second]
```

Exercise 06

Tree Traversal - Iterating is human...

repository subdir:	/ex06
compilation:	gcc -Wall -Wextra -Werror -std=gnu99
files to turn in:	tree_traversal.c, stack.c, queue.c, generic_list.c
forbidden functions:	none
points:	5

The purpose of this exercise is to iterate over a tree, in a generic way, using containers.
Our tree is defined as follows:

```
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.
Here is a list of functions to implement (in the `tree_traversal.c` file):

- **Information functions**

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

  returns TRUE if the tree is empty, and FALSE otherwise.

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

  displays the content of a node. In order to do this, the first argument should be a pointer to a node and the second should be a pointer to a display function, with the following type: **typedef void (*t_dump_func)(void *data);**.

- **Modification functions**

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

  initializes the tree pointed to by **"tree_ptr"** by creating a root node that has **"data"** as a value. It returns FALSE if the root node cannot be allocated, and TRUE otherwise.

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

  adds a child node to the node pointed to by **"tree_node"**. The child node will have **"data"** as a value. The function returns NULL if the child node cannot be added, otherwise it returns a pointer to the new child node.

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

  deletes the tree pointed to by **"tree_ptr"**. (Meaning the node pointed to by **"tree_ptr"** and all its child nodes). It resets the value pointed to by **"tree_ptr"** to an empty tree (NULL). It returns FALSE if it fails and TRUE otherwise.

- **Tree Traversal**
  In order to code the ultimate function, we need to define a generic container as follows:

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

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

The **t_container** structure represents a generic container and the **"container"** field stores the real container's address. The **"push_func"** field is a function pointer that lets us insert an element into the container. The **"pop_func"** field is a function pointer that lets us 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);
```

iterates over **"tree"** and displays its content by using **"container"**. The function pointed to by **"dump_func"** should be called to display the tree's data.
To do this, each of the tree's nodes must insert its child nodes into the container, display itself and start the process again, with the next node being the one that is extracted from the container.

Your output will go from left to right with a FIFO container, and from right to left with a LIFO container. It's normal.

Here is an example main with the expected output:

```
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));}

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

void     *generic_pop_stack(void *container)
{
    void   *data;

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

void *generic_pop_queue(void *container)
{
    void *data;

    data = queue_front(*(t_queue *)container);
    queue_pop((t_queue *)container);
    return (data);
}

int             main(void)
{
    t_tree      tree = NULL;
    t_tree_node *node;
    int         val_0 = 0;
    int         val_a = 1;
    int         val_b = 2;
```

```c
    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_container container;
    t_stack     stack = NULL;
    t_queue     queue = NULL;

    init_tree(&tree, &val_0);
    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);
    tree_add_child(node, &val_cc);

    printf("Depth Scan :\n");
    container.container = &stack;
    container.push_func = &generic_push_stack;
    container.pop_func  = &generic_pop_stack;
    tree_traversal(tree, &container, &dump_int);

    printf("Width Scan :\n");
    container.container = &queue;
    container.push_func = &generic_push_queue;
    container.pop_func  = &generic_pop_queue;
    tree_traversal(tree, &container, &dump_int);

    return (0);
}
```

Terminal

```
~/B-PAV-242> ./a.out
Depth Scan:
0
3
33
32
31
2
1
12
11
Width Scan:
0
1
2
3
11
12
31
32
33
```