



Piscine C++ - d02a

Mehdi “Drax” Ait-Bachir ait-ba_m@epitech.eu

Abstract: Ce document est le sujet du d02a

Table des matières

I	Remarques générales	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


Chapitre I

Remarques générales

- Si vous faites la moitié des exercices car vous avez du mal, c'est normal. Par contre, si vous faites la moitié des exercices par flemme et vous tirez à 14h, vous AUREZ des surprises. Ne tentez pas le diable.
- Les noms de fichiers qui vous sont imposés doivent être respectés A LA LETTRE, de même que les noms de fonctions.
- Les répertoires de rendus sont ex00, ex01, ...
- Lisez attentivement les exemples, ils peuvent requérir des choses que le sujet ne dit pas...
- Lisez ENTièrement le sujet d'un exercice avant de le commencer !
- RÉFLECHISSEZ. Par pitié.
- Notez bien qu'aucun de vos fichiers ne doit contenir de fonction "main" sauf si le contraire est explicite. Nous utiliserons notre propre fonction "main" pour compiler et tester votre code.

Chapitre II

Exercice 0

	Exercice : 00	points : 2
Simple List - Créer une liste simple		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d02a-promo-login_x)/ex00		
Compilateur : gcc	Flags de compilation: -Wall -Wextra -Werror	
Makefile : Non	Règles : n/a	
Fichiers a rendre : simple_list.c		
Remarques : Le fichier 'simple_list.h' vous est fourni, vous devez l'utiliser sans le modifier		
Fonctions Interdites : Aucune		

Le but de cet exercice est de créer une série de fonctions qui vous permettront de manipuler une liste.

On considère que notre liste est définie de la manière suivante :

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

Une liste vide est représentée par un pointeur NULL.

Nous allons également définir le type suivant (représentant un booléen) :

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

Voici la série de fonctions à coder :

- Fonctions d'informations (Fichier `simple_list.c`)
 - `unsigned int list_get_size(t_list list);`
Prend une liste en paramètre et retourne le nombre d'éléments contenus dans cette liste.
 - `t_bool list_is_empty(t_list list);`
Prend une liste en paramètre, retourne `TRUE` si la liste est vide, `FALSE` sinon.
 - `void list_dump(t_list list);`
Prend une liste en paramètre et affiche chacun des éléments de la liste, séparés par des retour à la ligne. Utilisez l'affichage par défaut de `printf (%f)` sans précision particulière.
- Fonctions de modification (Fichier `simple_list.c`)
 - `t_bool list_add_elem_at_front(t_list *front_ptr, double elem);`
Ajoute un nouveau noeud au début de la liste ayant pour valeur `'elem'` . La fonction renvoie `FALSE` si elle ne peut allouer le nouveau noeud, `TRUE` sinon.
 - `t_bool list_add_elem_at_back(t_list *front_ptr, double elem);`
Ajoute un nouveau noeud à la fin de la liste ayant pour valeur `'elem'` . La fonction renvoie `FALSE` si elle ne peut allouer le nouveau noeud, `TRUE` sinon.
 - `t_bool list_add_elem_at_position(t_list * front_ptr, double elem, unsigned int position);`
Ajoute un nouveau noeud à la position `'position'` ayant pour valeur `'elem'` . Si `'position'` vaut 0 cela revient à exécuter la fonction `'list_add_elem_at_front'` .

La fonction renvoie `FALSE` si elle ne peut allouer le nouveau noeud ou si `'position'` est invalide, sinon elle renvoie `TRUE` .
 - `t_bool list_del_elem_at_front(t_list *front_ptr);`
Supprime le premier noeud de la liste.
Renvoie `FALSE` si la liste est vide, `TRUE` sinon.
 - `t_bool list_del_elem_at_back(t_list *front_ptr);`
Supprime le dernier noeud de la liste.
Renvoie `FALSE` si la liste est vide, `TRUE` sinon.
 - `t_bool list_del_elem_at_position(t_list *front_ptr, unsigned int position);`
Supprime le noeud à la position `'position'` . Si `'position'` vaut 0 cela revient à exécuter la fonction `'list_del_elem_at_front'` . Renvoie `FALSE` si la liste est vide ou si `'position'` est invalide, sinon revoie `TRUE`.
- Fonctions d'accès aux valeurs (Fichier `simple_list.c`)


- `double list_get_elem_at_front(t_list list);` Renvoie la valeur du premier noeud de la liste. Renvoie 0 si la liste est vide.
 - `double list_get_elem_at_back(t_list list);` Renvoie la valeur du dernier noeud de la liste. Renvoie 0 si la liste est vide.
 - `double list_get_elem_at_position(t_list list, unsigned int position);` Renvoie la valeur du noeud a la position '`position`'. Si '`position`' vaut 0 cela revient à exécuter la fonction '`list_get_elem_at_front`'. Renvoie 0 si la liste est vide ou si '`position`' est invalide.
- Fonctions d'accès aux noeuds (Fichier `simple_list.c`)
 - `t_node *list_get_first_node_with_value(t_list list, double value);` Renvoie un pointeur sur le premier noeud de la liste '`list`' ayant pour valeur '`value`'. S'il n'y en a aucun renvoie `NULL`.

Voici un exemple de Main avec la sortie attendue :

```
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 $>
```

Chapitre III

Exercice 1

	Exercice : 01	points : 3
Simple BTree - Créer un arbre simple		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d02a-promo-login_x)/ex01		
Compilateur : gcc	Flags de compilation: -Wall -Wextra -Werror	
Makefile : Non	Règles : n/a	
Fichiers a rendre : simple_btree.c		
Remarques : Le fichier 'simple_btree.h' vous est fourni, vous devez l'utiliser sans le modifier.		
Fonctions Interdites : Aucune		

Le but de cet exercice est de créer une série de fonctions qui vous permettront de manipuler un arbre binaire.

On considère que notre arbre est défini de la manière suivante :

```

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;
```

Un arbre vide est représenté par un pointeur `NULL`.

Voici la série de fonctions à coder :


- Fonctions d'informations (Fichier `simple_btree.c`)
 - `t_bool btree_is_empty(t_tree tree);`
Renvoie `TRUE` si l'arbre `'tree'` est vide. `FALSE` sinon.
 - `unsigned int btree_get_size(t_tree tree);`
Renvoie le nombre de noeuds contenus dans l'arbre `'tree'`.
 - `unsigned int btree_get_depth(t_tree tree);`
Renvoie la profondeur de l'arbre `'tree'`.
- Fonctions de modification (Fichier `simple_btree.c`)
 - `t_bool btree_create_node(t_tree *node_ptr, double value);`
Cette fonction crée un nouveau noeud et le place à l'endroit pointé par `'node_ptr'` . La valeur du noeud est `'value'` .
Cette fonction renvoie `FALSE` si le noeud n'a pas pu être ajouté, `TRUE` sinon.
 - `t_bool btree_delete(t_tree *root_ptr);`
Cette fonction supprime l' ARBRE pointé par `'root_ptr'`. (Donc tous les noeuds enfants..)
Cette fonction renvoie `FALSE` si l'arbre est vide, `TRUE` sinon.
- Fonctions d'accès (Fichier `simple_btree.c`)
 - `double btree_get_max_value(t_tree tree);`
Cette fonction renvoie la valeur maximale contenue dans l'arbre `'tree'` . Renvoie 0 si l'arbre est vide.
 - `double btree_get_min_value(t_tree tree)` Cette fonction renvoie la valeur minimale contenue dans l'arbre `'tree'` . Renvoie 0 si l'arbre est vide.

Voici un exemple de main et la sortie attendue :

```
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 $>
```

Chapitre IV

Exercice 2

	Exercice : 02	points : 3
Generic List - Créer une liste générique		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d02a-promo-login_x)/ex02		
Compilateur : gcc	Flags de compilation: -Wextra -Werror -Wall	
Makefile : Non	Règles : n/a	
Fichiers a rendre : generic_list.c		
Remarques : Le fichier 'generic_list.h' vous est fourni, vous devez l'utiliser sans le modifier.		
Fonctions Interdites : Aucune		

Le but de cet exercice est de créer une liste générique.

La différence par rapport a l'exercice 'Simple List' est que le noeud d'une structure est définie de la manière suivante :

```

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

La liste des fonctions est la même avec un prototypage légèrement différent :

```

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);
```

Seules deux fonctions diffèrent réellement :

- `typedef void (*t_value_displayer)(void *value);`
`void list_dump(t_list list, t_value_displayer val_disp);`
La fonction 'list_dump' prend maintenant un pointeur sur fonction de type 't_value_displayer'. En utilisant la fonction pointée par 'val_disp' cela nous permet d'afficher la valeur 'value' contenue dans un noeud suivi d'un retour à la ligne.
- `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);`
La fonction 'list_get_first_node_with_value' prend maintenant un pointeur sur fonction de type 't_value_comparator' qui permet de comparer deux valeurs de la liste.
La fonction de comparaison renvoie une valeur positive si 'first' est supérieur à 'second', négative si 'second' est supérieur à 'first', et égale à 0 si 'first' est égal à 'second'.


Voici un exemple de main et la sortie attendue :

```
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 $>
```



Chapitre V

Exercice 3

	Exercice : 03	points : 2
Stack - Créer une stack		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d02a-promo-login_x)/ex03		
Compilateur : gcc	Flags de compilation: -Wextra -Werror -Wall	
Makefile : Non	Règles : n/a	
Fichiers a rendre : stack.c, generic_list.c		
Remarques : Les fichiers 'stack.h' et 'generic_list.h' vous sont fournis, vous devez les utiliser sans les modifier.		
Fonctions Interdites : Aucune		

Un code enveloppant un autre code s'appelle un Wrapper.

Le but de cet exercice est de créer une stack (pile) a partir de la liste générique créée précédemment.

Reprenez le fichier 'generic_list.c' fait précédemment, il sera utilisé mais aucune modification ne lui sera apportée.

Pour cela nous allons partir du fait que pour nous une stack est une liste dont on limite l'utilisation de manière intelligente. On a donc :

```
1 typedef t_list t_stack;
```

Voici la liste des fonctions à créer :

- Fonctions d'informations (Fichier stack.c)
 - unsigned int stack_get_size(t_stack stack);
Retourne le nombre d'éléments dans la stack.
 - t_bool stack_is_empty(t_stack stack);
Retourne TRUE si la stack est vide. FALSE sinon.


- Fonctions de modification (Fichier `stack.c`)
 - `t_bool stack_push(t_stack *stack_ptr, void *elem);`
Empile l'élément '`elem`' au dessus de la stack. Renvoie `FALSE` si le nouvel élément n'a pas pu être ajouté, `TRUE` sinon.
 - `t_bool stack_pop(t_stack *stack_ptr);`
Dépile l'élément au dessus de la stack. Renvoie `FALSE` si la stack est vide, `TRUE` sinon.
- Fonctions d'accès (Fichier `stack.c`)
 - `void *stack_top(t_stack stack);`
Renvoie la valeur de l'élément au dessus de la stack.

Voici un exemple de main avec la sortie attendue :

```
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 $>
```

Chapitre VI

Exercice 4

	Exercice : 04	points : 2
Queue - Créer une queue		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d02a-promo-login_x)/ex04		
Compilateur : gcc	Flags de compilation: -Wextra -Werror -Wall	
Makefile : Non	Règles : n/a	
Fichiers a rendre : queue.c, generic_list.c		
Remarques : Les fichiers 'queue.h' et 'generic_list.h' vous sont fournis, vous devez les utiliser sans les modifier.		
Fonctions Interdites : Aucune		

Un code enveloppant un autre code s'appelle un Wrapper.

Le but de cet exercice est de créer une queue (file) à partir de la liste générique créée précédemment.

Reprenez le fichier 'generic_list.c' fait précédemment, il sera utilisé mais aucune modification ne lui sera apportée.

Pour cela nous allons partir du fait que pour nous une `queue` est une liste dont on limite l'utilisation de manière intelligente. On a donc :

```
1 typedef t_list t_queue;
```

Voici la liste des fonctions à créer :

- Fonctions d'informations (Fichier `queue.c`)
 - `unsigned int queue_get_size(t_queue queue);`
Retourne le nombre d'éléments dans la queue.


- `t_bool queue_is_empty(t_queue queue);`
Retourne `TRUE` si la queue est vide. `FALSE` sinon.
- Fonctions de modification (Fichier `queue.c`)
 - `t_bool queue_push(t_queue *queue_ptr, void *elem);`
Ajoute l'élément `'elem'` a la queue. Renvoie `FALSE` si le nouvel élément n'a pas pu être ajouté, `TRUE` sinon.
 - `t_bool queue_pop(t_queue *queue_ptr);`
Retire l'élément suivant de la queue. Renvoie `FALSE` si la queue est vide, `TRUE` sinon.
- Fonctions d'accès (Fichier `queue.c`)
 - `void *queue_front(t_queue queue);` Renvoie la valeur de l'élément suivant de la queue.

Voici un exemple de main avec la sortie attendue :

```
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 $>
```

Chapitre VII

Exercice 5

	Exercice : 05	points : 3
Map - Créer une map		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d02a-promo-login_x)/ex05		
Compilateur : gcc	Flags de compilation: -Wextra -Werror -Wall	
Makefile : Non	Règles : n/a	
Fichiers a rendre : generic_list.c, map.c		
Remarques : Les fichiers 'map.h' et 'generic_list.h' vous sont fournis, vous devez les utiliser sans les modifier.		
Fonctions Interdites : Aucune		

Un code enveloppant un autre code s'appelle un Wrapper.

Le but de cet exercice est de créer une map (tableau associatif) a partir de la liste générique créée précédemment.

Reprenez le fichier 'generic_list.c' fait précédemment. Une seule fonction lui sera ajoutée.

Pour cela nous allons partir du fait que pour nous une map est une liste dont on limite l'utilisation de manière intelligente. On a donc :

```
1 typedef t_list t_map;
```

La question principale est : "Une map c'est une liste de quoi?!" Et la reponse est :

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



Méditez la-dessus...

Voici la liste des fonctions à créer :

- Fonctions d'informations (Fichier `map.c`)
 - `unsigned int map_get_size(t_map map);` Renvoie le nombre d'éléments dans la map.
 - `t_bool map_is_empty(t_map map);` Renvoie `TRUE` si la map est vide. `FALSE` sinon.

C'est là où ça se corse.

Comme notre map est générique, la clef `'key'` peut contenir n'importe quel type de données. Pour pouvoir les comparer et savoir si une clef est égale à une autre clef (entre autres), il nous faut un pointeur sur fonction contenant un comparateur de clefs :

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

Renvoie 0 si les clefs sont égales, un nombre strictement positif si `'first_key'` est supérieur à `'second_key'` et un nombre strictement négatif si `'second_key'` est supérieur à `'first_key'`.

La liste générique utilise elle-même ce système de pointeurs sur fonction pour trouver un noeud possédant une valeur particulière.

La question est donc “comment faire pour que la fonction appelée par notre liste appelle à son tour la fonction de comparaison de clefs, sachant que l'on ne peut pas lui passer d'autres paramètres ?”

Deux solutions :

- Une variable globale
- Un wrapper sur une variable globale ;)

Par souci d'esthétique nous opterons pour la deuxième solution. Vous allez donc coder les deux fonctions suivantes (Fichier `map.c`) :

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

Cette fonction stocke une variable static de type `t_key_comparator` . Si `'store'` vaut `TRUE` , la nouvelle valeur de la variable static est `'new_key_cmp'` . La fonction renvoie toujours la valeur contenue dans sa variable static. Cela émule le comportement d'une variable globale : si vous voulez stocker une valeur, appelez cette fonction avec `TRUE` en premier paramètre puis la valeur à stocker. Si vous voulez accéder à la valeur, appelez cette fonction avec `FALSE` en premier paramètre et `NULL` en deuxième.

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

Qui prend en paramètres deux valeurs de notre liste (`void *`) , qui sont en réalité des pointeurs sur pairs (`t_pair *`) . Cette fonction compare uniquement les clefs contenues dans ces pairs. Elle renvoie `0` si les clefs sont égales, un nombre strictement positif si la clef de `'first'` est supérieure à la clef de `'second'` , et un nombre strictement négatif si la clef de `'second'` est supérieure à la clef de `'first'`.

Avant de continuer sur les quelques fonctions restantes nous allons ajouter une fonction basique à notre liste générique.

- Amélioration de la liste générique (Fichier `generic_list.c`)
 - `t_bool list_del_node(t_list *front_ptr, t_node *node_ptr);` Cette fonction supprime le noeud pointé par `'node_ptr'` de la liste. Cette fonction renvoie `FALSE` si le noeud ne se trouve pas dans la liste.

Revenons enfin à la map.

- Fonctions de modification (Fichier `map.c`)
 - `t_bool map_add_elem(t_map *map_ptr, void *key, void *value, t_key_comparator key_cmp);`
Cette fonction ajoute la valeur `'value'` à l'index `'key'` de la map. S'il y a déjà une valeur à l'index `'key'` cette valeur est remplacée par la nouvelle. `'key_cmp'` permet de comparer les clef de la map. Renvoie `FALSE` si l'élément n'a pas pu être ajouté, `TRUE` sinon.
 - `t_bool map_del_elem(t_map *map_ptr, void *key, t_key_comparator key_cmp);`
Cette fonction supprime la valeur contenue à la clef `'key'` . `'key_cmp'` permet de comparer les clef de la map. Renvoie `FALSE` s'il n'y a aucune valeur à l'index `'key'` , `TRUE` sinon.

- Fonctions d'accès (Fichier `map.c`)

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

Cette fonction renvoie la valeur contenue à l'index `'key'` de la map. S'il n'y a pas de valeur à l'index `'key'`, cette fonction renvoie `NULL`. `'key_cmp'` permet de comparer les clefs de la map.


Voici un exemple de main avec la sortie attendue :

```
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 $>
```

Chapitre VIII

Exercice 6

Les fichiers 'tree_traversal.h' , 'stack.h' , 'queue.h' et 'generic_list.h' vous sont fournis, vous devez les utiliser sans les modifier.

	Exercice : 06	points : 5
Tree Traversal - Itérer est humain...		
Répertoire de rendu: (DÉPOT SVN - piscine_cpp_d02a-promo-login_x)/ex06		
Compilateur : gcc	Flags de compilation: -Wextra -Werror -Wall	
Makefile : Non	Règles : n/a	
Fichiers a rendre : tree_traversal.c, stack.c, queue.c, generic_list.c		
Remarques : n/a		
Fonctions Interdites : Aucune		

Le but de cet exercice est de parcourir un arbre de manière générique grâce à des conteneurs.

L'arbre est défini de la manière suivante :

```

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;
```

Où 'data' est la donnée contenue dans un noeud, 'parent' est un pointeur sur le noeud parent et 'children' est une liste générique de noeuds enfants.

Un arbre vide est représenté par un pointeur NULL .

Voici la série de fonctions à coder :

- Fonctions d'informations (Fichier `tree_traversal.c`)
 - `t_bool tree_is_empty(t_tree tree);` Cette fonction renvoie `TRUE` si l'arbre est vide. `FALSE` sinon.
 - `void tree_node_dump(t_tree_node *tree_node, t_dump_func dump_func);` Cette fonction permet d'afficher le contenu d'un noeud. Pour cela elle prend en paramètres un pointeur sur un noeud et un pointeur sur une fonction d'affichage définie de la manière suivante : `typedef void (*t_dump_func)(void *data);`
- Fonctions de Modifications (Fichier `tree_traversal.c`)
 - `t_bool init_tree(t_tree *tree_ptr, void *data);` Cette fonction initialise l'arbre pointé par `'tree_ptr'` en créant un noeud racine qui aura pour valeur `'data'` . Renvoie `FALSE` si le noeud racine n'a pas pu être alloué, `TRUE` sinon.
 - `t_tree_node *tree_add_child(t_tree_node *tree_node, void *data)` Cette fonction ajoute un noeud enfant au noeud pointé par `'tree_ptr'` . Ce noeud enfant aura pour valeur `'data'` . La fonction renvoie `NULL` si le noeud enfant n'a pas pu être ajouté, sinon elle renvoie un pointeur sur ce noeud enfant.
 - `t_bool tree_destroy(t_tree* tree_ptr);` Cette fonction supprime l'arbre pointé par `'tree_ptr'` . (Donc le noeud pointé par `'tree_ptr'` ainsi que tous les noeuds enfants). Elle réinitialise la valeur de l'arbre pointé par `'tree_ptr'` à `NULL` . Cette fonction renvoie `FALSE` si elle échoue, `TRUE` sinon.
- Tree traversal (Fichier `tree_traversal.c`)

Pour coder la fonction ultime, nous allons définir un conteneur générique de la manière suivante :

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

avec :

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

La structure `container` représente un conteneur générique, le champ `'container'` contient l'adresse du conteneur à proprement parler. Le champ `'push_func'` est un pointeur sur fonction permettant d'insérer un élément dans ce conteneur. Le champ `'pop_func'` est un pointeur sur fonction permettant de retirer un élément de ce conteneur.

Coder la fonction suivante :

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

Cette fonction permet de parcourir et d'afficher les données se trouvant dans l'arbre `'tree'` en utilisant le conteneur `'container'`. La fonction pointée par `'dump_func'` sert à afficher les données de l'arbre.

Pour cela il faut pour chaque noeud de l'arbre insérer ses noeuds enfants dans le conteneur, afficher le noeud, puis recommencer le processus en prenant comme prochain élément celui que l'on retirera du conteneur.



Votre affichage se fera de gauche à droite pour un conteneur FIFO et de droite à gauche pour un conteneur LIFO, c'est normal.

```
1 Voici un exemple de main et la sortie attendue :
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 $>
```