C - Pool - Tek1

Day 14 subject

C Pool Managers
looneytunes@epitech.eu

Contents

Instructions

The following exercises are only for the Tech3SI !

- The subject may change until one hour before turn-in.
- Respect the norm takes time, but is good for you. This way your code will respect the norm since the first written line.
- Ask yourself if it's relevant to let a `main()` function in your turn-in knowing we will add our own.
- For each exercise's directory, we are going to compile your files with the `cc -c *.c` command, so it will generate every single `.o` that we will link one by one with the addition of our `main.c` function:

```
$> cd ex_01
$> cc *.c -c
$> cc *.o ~moulinette/main_ex_01.o -I../include/ -L../lib/ -o ex01 -lmy
$> ./ex01
[...]
```

- This is a turn-in directory, of course you will only keep in it your final work revision. No temporary file should stand there!
  You shall leave in your directory no other files than those explicitly specified by the exercises.
  If one of your files prevents the compilation with `*.c`, the robot will not be able to do the correction and you will have a `0`. That is why it's in your interest to remove any file that doesn't work.
- Don't forget to discuss about it in the pool section of the forum !
- Turn-in:
  `Piscine_C_J14`

Hints Remember it is always better to create your repository at the beginning of the day and to turn-in your work on a regular basis

Hints On the instructions of each exercises, this directory is specified for every turn-in path

- Your library will be used during the 'linkage' phase.

Unit Tests

- It is highly recommended to test your functions when you are developing them.
- Usually, it is common to create a function named "`main`" (and a dedicated file to host it) to check the functions separately.
- Create a directory named "`tests`".
- Create a function "`int main()`" in a file named "`tests-exercise_name.c`", stored inside the directory "`tests`" previously created.
- According to you, this function must contains all the necessary call to "`exercise_name`" to cover all possible cases (special or regular) of the function.

Technical details

- For the exercises of today, we will use the following structure:

```
1   typedef struct s_btree
2   {
3       struct s_btree *left;
4       struct s_btree *right;
5       void *item;
6   } t_btree;
```

- You will have to define this structure in your files. Be careful : Don't add attributes and don't change the order. Otherwise your grade will tend towards `0`.

Exercise 00 : btree_create_node

- Write the `btree_create_node` function which allocates a new element, initialize its `item` to the parameter value and all the others to 0.
- The created node's memory adress is returned.
- It must be prototyped like this :

```
1   t_btree *btree_create_node(void *item);
```

- Turn-in directory:
`Piscine_C_J14/ex_00/`

Exercise 01 : btree_apply_prefix

- Write the `btree_apply_prefix` function which executes the function given as parameter to each node, looking down the tree in a `prefix` fashion.
- It must be prototyped like this :

```
1   void btree_apply_prefix(t_btree *root, int (*applyf)(void *));
```

- Turn-in directory:
  `Piscine_C_J14/ex_01/`

Exercise 02 : btree_apply_infix

- Write the `btree_apply_infix` function which executes the function given as parameter to each node, looking down the tree in a `infix` fashion.
- It must be prototyped like this :

```
1   void btree_apply_infix(t_btree *root, int (*applyf)(void *));
```

- Turn-in directory:
`Piscine_C_J14/ex_02/`

Exercise 03 : btree_apply_suffix

- Write the `btree_apply_suffix` function which executes the function given as parameter to each node, looking down the tree in a `suffix` fashion.
- It must be prototyped like this :

```
1   void btree_apply_suffix(t_btree *root, int (*applyf)(void *));
```

- Turn-in directory:
  `Piscine_C_J14/ex_03/`

Exercise 04 : btree_insert_data

- Write the `btree_insert_data` function which inserts the `item` element in a tree. The tree given as parameter will be sorted. It means that for each `node`, all lower elements are in the left subtree, and all greater or equal elements are in the right subtree. We'll give a comparative function as parameter, providing the same behavior as `strcmp`.
- It must be prototyped like this :

```
1   void btree_insert_data(t_btree **root, void *item, int (*cmpf)(void *, void *));
```

- Turn-in directory:
`Piscine_C_J14/ex_04/`

Exercise 05 : btree_search_item

- Write the `btree_search_item` function which returns the first element corresponding to the reference data given as parameter. The tree will have to be looked down in a `infix` fashion. If the element is not found, the function must returns `NULL`.
- It must be prototyped like this :

```
1   void *btree_search_item(t_btree *root, void *data_ref, int (*cmpf)(void *, void *));
```

- Turn-in directory:
  `Piscine_C_J14/ex_05/`

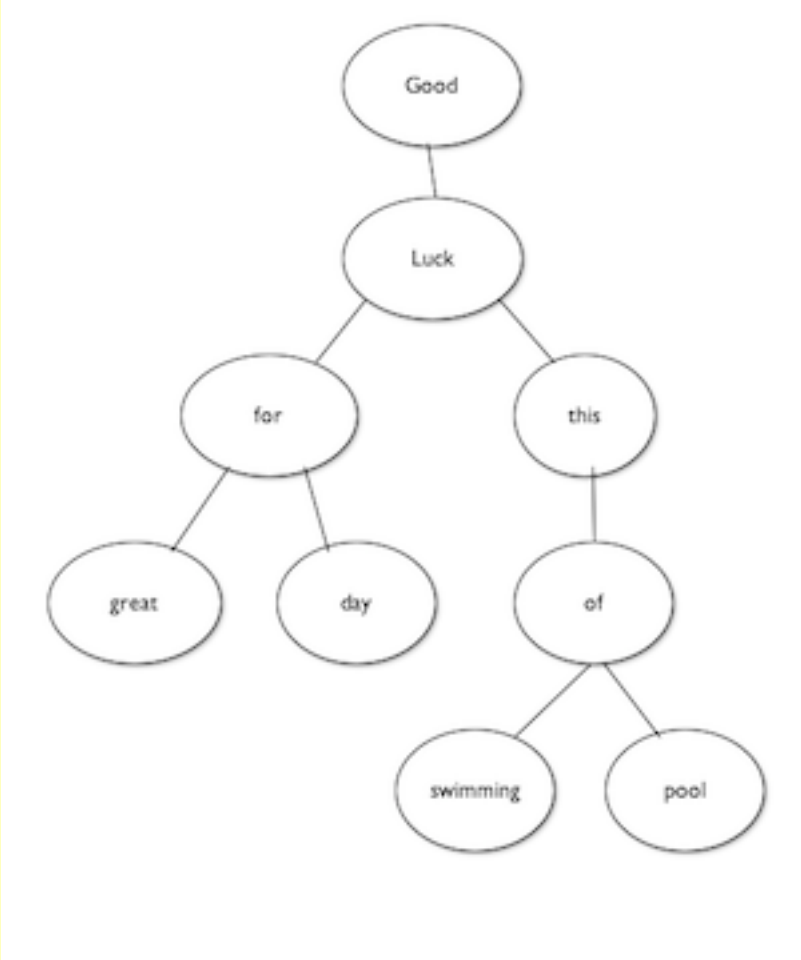
Exercise 06 : btree_level_count

- Write the `btree_level_count` function which returns the size of the biggest branch given as parameter.
- It must be prototyped like this :

```
1   int btree_level_count(t_btree *root);
```

- Turn-in directory:
  `Piscine_C_J14/ex_06/`

Hints



`In this exemple, we will return 5.`

Exercise 07 : btree_apply_by_level

- Write the `btree_apply_by_level` function which executes the function given as parameter to each node in the tree. The tree should be looked level per level. The called function will take three parameters :
  - The first parameter, of type void *, corresponds to the node's item.
  - The second parameter, of type `int`, corresponds to the level of the current position : 0 for root, 1 for children, 2 for his subtrees..
  - The third parameter, of type `int` is equal to 1 if it's the first level or else 0.
- It must be prototyped like this :

```
1  void btree_apply_by_level(t_btree *root, void (*applyf)(void *item, int current_level, int
       is_first_elem))
```

- Turn-in directory:
  `Piscine_C_J14/ex_07/`

Hints

For a tree like the one represented above, the applyf function should be called successively with the following parameters:

- "Good", 0, 1
- "Luck", 1, 1
- "for", 2, 1
- "this", 2, 0
- "great", 3, 1
- "day", 3, 0
- "of", 3, 0
- "swimming", 4, 1
- "pool", 4, 0

Technical details

- Now, we'll work with red-black trees.

```
1  typedef struct rb_node
2  {
3      struct rb_node *left;
4      struct rb_node *right;
5      void *data;
6      enum RB_COLOR { RB_BLACK, RB_RED } color;
7  } t_rb_node;
```

- This structure has the same attributes as the previous structure at the beginning. It's possible to re-use the functions already written with red-black trees. It's kind of a rudimentary form of polymorphism in C.

Hints You may add some attributes **at the end** of the structure if you think it's necessary.

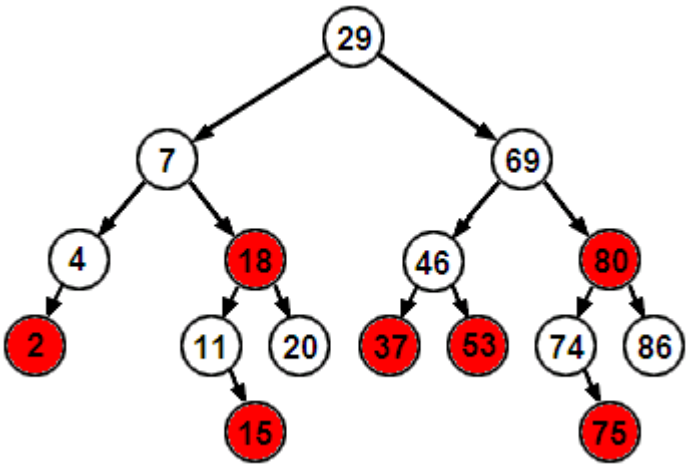
Exercise 08 : rb_insert

- Write the `rb_insert` function which add a new data into the tree while preserving the constraints of a red-black tree. The parameter `root` points to the root node of the tree. During the first call, it may be set to a `NULL` pointer. Also, we will send by parameter, a comparative function, providing the same behavior as `strcmp`.
- It must be prototyped like this :

```
1   void rb_insert(struct rb_node **root, void *data, int (*cmpf)(void *, void *));
```

- Turn-in directory:
`Piscine_C_J14/ex_08/`

Hints



red-black tree example

Exercise 09 : rb_remove

- Write the `rb_remove` function which deletes a data from the tree while preserving the constraints of a red-black tree. The parameter `root` points to the root node of the tree. Also, we will send a comparative function as parameter, providing the same behavior as `strcmp`, and a function pointer `freef` which will be called with the element of the tree which must be deleted as parameter.

- It must be prototyped like this :

```
1    void rb_remove(struct rb_node **root, void *data, int (*cmpf)(void *, void *), void (*freef)(
         void *));
```

- Turn-in directory:
`Piscine_C_J14/ex_09/`