B1 - Unix & C Lab Seminar

B-CPE-100

Day 13 - ED

Binary Tree

1.0

Day 13 - ED

language: C

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- 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).

- Don't push your `main` nor `my_putchar` functions into your delivery directory, we will be adding our own. Your files will be compiled adding our `main.c` and `my_putchar.c`.

Allowed system function(s): `write`, `malloc`, `free`

For today's tasks, we will be using the following structure:

```
typedef struct btree
{
    struct btree *left;
    struct btree *right;
    void *item;
} btree_t;
```

You have to define this structure in file named `btree.h` placed in the your `include/` folder. But be careful: don't add attributes nor change their order, or your grade will lean toward 0.

We still encourage you to write unit tests for all your functions!
Check out Day06 if you need an example, and re-read the guide.

TASK 00 - LIBBTREE.A

Delivery: Makefile

You **must** have a Makefile at the root of your directory which will built a library called `libbtree.a` containing your tasks of the day.
The `btree.h` file must also contains the prototype of all the functions exposed in your `libbtree.a`.
Your Makefile must implement the following rules: all, clean, fclean and re.
For each of the following tasks we will build our main function with your library like so:

```
Terminal
~/B-CPE-100> make re
~/B-CPE-100> gcc main.c -I./include -L. -lbtree
```

TASK 01 - BTREE_CREATE_NODE

Delivery: btree_create_node.c

Write the `btree_create_node` function, which allocates a new node and initializes its item to the parameter value (and all the others to 0).
The newly-created node's memory address must be returned.
It must be prototyped as follows:

```c
btree_t *btree_create_node(void *item);
```

TASK 02 - BTREE_APPLY_PREFIX

Delivery: btree_apply_prefix.c

Write the `btree_apply_prefix` function, which executes the function given as parameter to each node while implementing a pre-order tree traversal. It must be prototyped as follows:

```c
void btree_apply_prefix(btree_t *root, int (*applyf)(void *));
```

TASK 03 - BTREE_APPLY_INFIX

Delivery: btree_apply_infix.c

Write the `btree_apply_infix` function, which executes the function given as parameter to each node, while implementing an in-order tree traversal. It must be prototyped as follows:

```
void btree_apply_infix(btree_t *root, int (*applyf)(void *));
```

TASK 04 - BTREE_APPLY_SUFFIX

Delivery: btree_apply_suffix.c

Write the `btree_apply_suffix` function, which executes the function given as parameter to each node, while implementing a post-order tree traversal. It must be prototyped as follows:

```
void btree_apply_suffix(btree_t *root, int (*applyf)(void *));
```

TASK 05 - BTREE_INSERT_DATA

Delivery: btree_insert_data.c

Write the `btree_insert_data` function that inserts the item element into a tree.
The tree given as parameter must be sorted, which means that for each **node**, all lower elements must be in the left subtree and all greater than/equal to elements must be in the right subtree.
You will give a comparative function as parameter, which acts the same way as `strcmp`.
It must be prototyped as follows:

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

TASK 06 - BTREE_SEARCH_ITEM

Delivery: btree_search_item.c

Write the `btree_search_item` function that returns the first element that corresponds to the reference data given as parameter. If the element is not found, the function must return `NULL`.
You must implement an infix tree search.
It must be prototyped as follows:

```
void *btree_search_item(btree_t const *root, void const *data_ref, int (*cmpf)());
```

TASK 07 - BTREE_LEVEL_COUNT

Delivery: btree_level_count.c

Write the `btree_level_count` function that returns the size of the biggest branch given as parameter. It must be prototyped as follows:

```
size_t btree_level_count(btree_t const *root);
```

For instance, in the following example, the size of the biggest branch is 5:

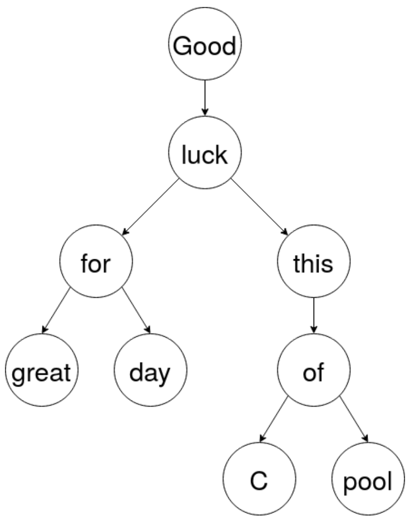


Figure 1: Example of B-Tree

The type `size_t` is defined in `stddef.h`.

TASK 08 - BTREE_APPLY_BY_LEVEL

Delivery: btree_apply_by_level.c

Write the `btree_apply_by_level` function, which executes the function given as parameter to each node in the tree. A level-by-level tree search should be implemented. The called function should have the three following parameters:

- the node's item (`void*`)
- the current position's level (`int`): 0 for root, 1 for children, 2 for subtrees
- 1 if it is the first level, 0 otherwise (`int`)

It must be prototyped as follows:

```
void btree_apply_by_level(btree_t *root, void (*applyf)(void *item, int level, int
    is_first_elem))
```

For instance, for a tree like the one represented above, the `applyf` function should be called separately, using the following parameters:

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

TASK 09 - RB_INSERT

Delivery: rb_insert.c

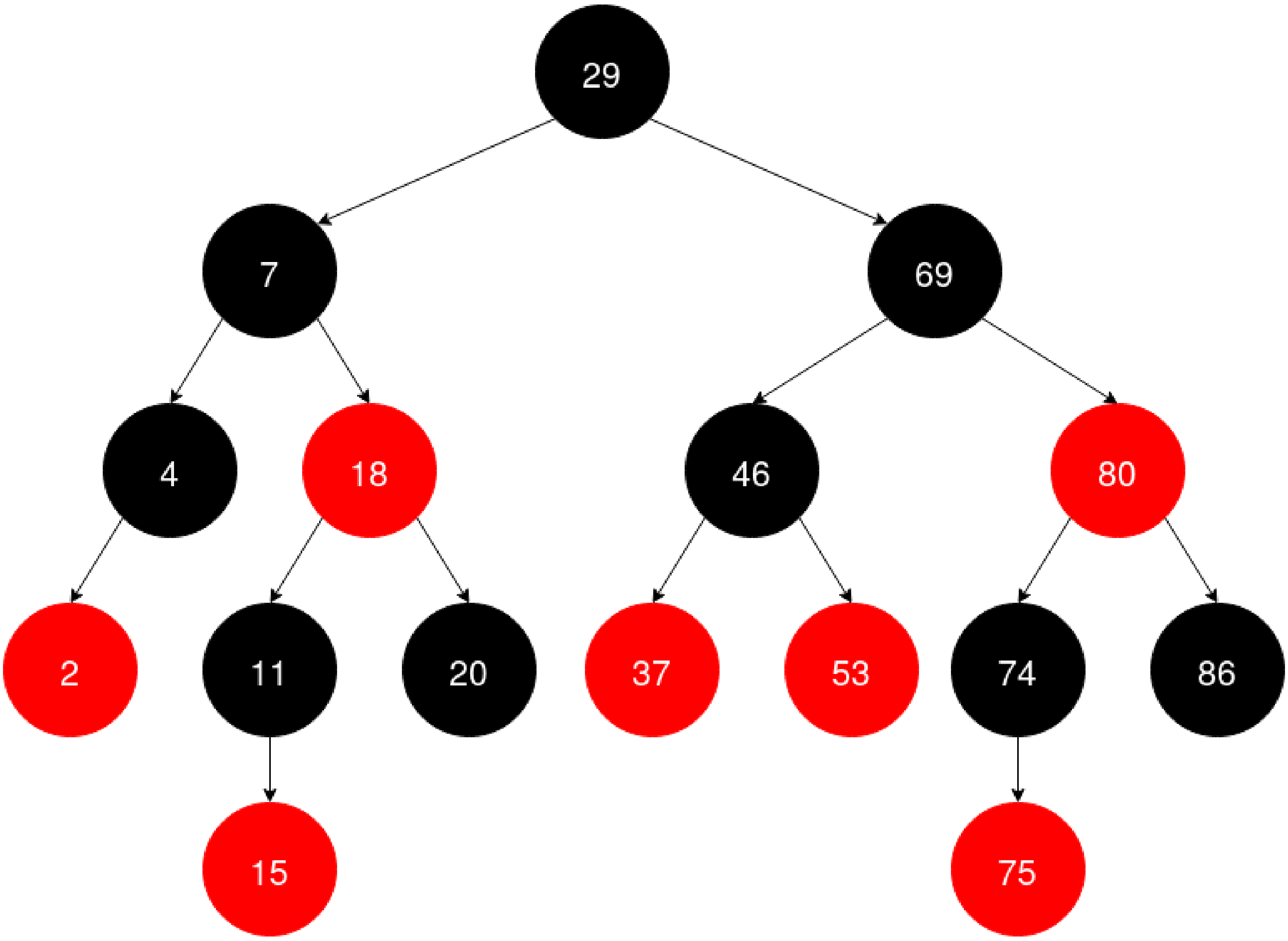
For the last two tasks, we are going to work with red-black trees:

```
typedef struct rb_node
{
    struct rb_node *left;
    struct rb_node *right;
    void *data;
    enum RB_COLOR {RB_BLACK, RB_RED} color;
} rb_node_t;
```

Add this structure in your `btree.h` file.
This structure has the same properties as the structure found at the beginning. It's possible to reuse the

functions that you have already written for red-black trees. Think of it as a basic form of polymorphism in C.

If you think it's necessary, you may add some properties at the end of the `rb_node_t` structure.



Write the `rb_insert` function, which adds new data to the tree while simultaneously keeping the red-black tree's restrictions.
The `root` parameter points to the tree's root node.
During the first call, it may be set to a `NULL` pointer.
You also need a comparative function that acts the same way as `strcmp`.
It must be prototyped as follows:

```
void rb_insert(rb_node_t **root, void *data, int (*cmpf)());
```

TASK 10 - RB_REMOVE

Delivery: rb_remove.c

Write the `rb_remove` function, which deletes data from the tree while simultaneously keeping the red-black tree's restrictions.
The `root` parameter points to the tree's root node.
You also need a comparative function that acts the same way as `strcmp`.
A function pointer called `f` must be called with the tree's elements that must be deleted.
It must be prototyped as follows:

```
void rb_remove(rb_node_t **root, void *data, int (*cmpf) (void *, void *), void (*f)
    (void *));
```