Showcasing genericity in C language, by writing a generic red-black tree

Red black trees are famous data structures, deriving from the simpler binary search trees. The latter can run most functions in O(N) time at worst. Red black trees are mostly balanced trees, so they improve on complexity, in the order of O(lgN).

In this project I created generic rbt libraries that provide functions to manipulate trees with nodes with integer keys, or string keys. THe same functions are used to operate on different types.
In the future one could possibly add more libraries to have trees with other data types as well, like, for example, double precision keys.

The project is meant to be a practise on the power of void pointers in C language, as well as the use of libraries and concepts of abstraction, in C programming.

What follows is the documentation of the current project's libraries and functions. All the functions of the libraries have proper contracts and comments written in their header files, that explain in detail what the iput arguments are (type, description etc) as well as the return values. Here their use will be briefly explained.

`RedBlackTree.h` library

This is the core library of the project providing the user with functions to manipulate red-black trees. It supports the manipulation of red-black trees that are structured from nodes that have keys of (theoretically) any data type, as long as it is used in conjunction with the corresponding key-type libraries (`IntegerKeys.h` and `StringKeys.h`). The structure of the red-black trees is based on the key values of the nodes, and their relationships (for example with integer keys they are sorted based on which is smaller and which is larger, while strings are sorted based on alphabetical order).
Along with the library goes a text file, **ErrorLogs.txt**, where the library writes to whenever it detects that some structure, that should be a red-black tree, is not a red black tree. That happens inside the source file `RedBlackTree.c`, using the functions `watchdog_rbt_selfcheck` and `watchdog_file_logger`. The type of mistake (the rule of the red-black tree structure that is broken) will be written in the ErrorLogs.txt file. Every time a function that modifies a tree is called the `watchdog_rbt_selfcheck` and `watchdog_file_logger` function ensure that what the function received is a red-black tree, and that it remains a red-black tree, after the modifications.

`handler`

This is a `typedef` of a pointer to an esoteric `struct` of the `RedBlackTree.c`, that the user will not have access to, but can declare variables of the type. It points to the so-called *sentinel* `struct` of the trees. These **sentinel structures** contain the basic information of each tree, and they are used by most functions inside `RedBlackTree.c` to operate on a specific tree.

`rbt_create`

This function is used to create a new tree. It allocates memory in the heap for the *sentinel* `struct` of the new tree returning a pointer to it as a type `handler`. The user is responsible of storing this pointer in an

apropriate `handler` variable, if they wish to operate on the specific tree. Else they will loose acces to it, resulting in memory leaks.

`rbt_destroy`

This function performs the opposite operation of `rbt_create`. It deallocated the memory of a tree's sentinel struct, using the `handler` to the tree's sentinel, effectively destroying the tree. In order to avoid memory leaks it is only allowed to destroy empty trees (trees that contain no nodes, but only have a sentinel `struct`). If the tree is not empty the function returns 1, without destroying the tree.

`rbt_nodefind`

This function finds a specific node inside a tree and returns a void pointer to its `struct`. This function is designed to work with other functions of the library, mainly to provide them with input arguments, when the user wishes to operate on a node (it is used with `rbt_nodeprint` and `rbt_delete`). The user must input the `handler` to the tree and a temporary proper key, of the same value of the key of the node they wish to find. That temporary key must be created with the key-type functions of the key-type libraries (`IntegerKeys.h` and `StringKeys.h`). The input argument will be a void pointer to a **key data-holder structure** created by the key-type functions. Pointers to comparison functions from the key-type libraries are added as input arguments of `rbt_nodefind` too.
If the function finds a specific node it returns a void pointer to it. Else it returns a NULL pointer. In any case the key inserted as argument is temporary, and the function is responsible to destroy it before it returns, to avoid memory leaks. Thus it takes as an argument a `destroykey` function, provided by the key-type libraries.

`rbt_nodeprint`

This function is designed to be used in conjunction with the `rbt_nodefind` function, to print the node's information - that is its key, its parent's and childrens' keys, and its color. The user must provide a pointer to the node directly, as a void pointer, as well as a proper `keyprinter` function provided by the key-type libraries (`IntegerKeys.h` and `StringKeys.h`). The only way for the user to acquire the pointer to a node in a tree is to run `rbt_nodefind` first, or even add it directly as argument. In the second case it will return 1 if `rbt_nodefind` found no proper node.

`rbt_insert`

This function is used to insert a node inside an existing red-black tree. The user must have first run `rbt_create` to create the tree, and use the returned `handler` as an input argument for `rbt_insert`. They must also have run a proper key creation function, provided by the key-type libraries (`IntegerKeys.h` and `StringKeys.h`), and add a pointer to the key's data-holder structure (that is returned by the key creation functions) as a second argument of `rbt_insert`. Also it must provide as arguments to `rbt_insert` pointers to functions from the key-type libraries, in order to perform comparisons of different key types.
The function has different return values to inform the user of proper execution, wether the insertion failed, or memory allocation in the heap failed. The latter is a fatal error and the user should terminate the program. The case where insertion failed is because a node with the inserted key already existed in the specified tree. In that case `rbt_insert` automatically destroys the inputed key data-holder structure

before it returns, to avoid memory leaks. To do that it takes as a last input argument a pointer to a `destroykey` function provided by the key-type libraries.

`rbt_delete`

This function is used to delete a node inside an existing red-black tree. The tree must have already been created using `rbt_create` and its `handler` should be known by the user, in order to be inputed as an argument. The user must also insert as an argument a pointer to the node they wish to delete. This is achived by running the `rbt_nodefind` function and using its return value as a second argument of `rbt_delete`. Alternatively they can call `rbt_nodefind` by adding it as an argument directly. Pointers to functions that operate on node's keys must also be isnerted as arguments of `rbt_delete`. These functions operate on key structures to compare them and destroy them, and they are provided by the key-type libraries (`IntegerKeys.h` and `StringKeys.h`).
The function has different return values to inform the user of the execution outcome. It informs wether the operation concluded with a succesful deletion, by returning 0. There is a specific case (return value 5) where it informs that the only node of the tree was deleted, because some users might like to run `rbt_destroy` directly after that. If the deletion did not conclude it informs the user of the possible errors, that are: the node they wish to delete does not exist, the tree is empty, some memory allocation failed *(esoterically in `RedBlackTree.c` some auxiliary memory allocations are performed, but before the function returns their memory is dealocated)*, or there are broken data types inside the structure. The latter is highly unlikely but the user must terminate the program in that case. If it terminates with a successful deletion that means that both the strucure of the node and the structure of the key of the node are deleted, and their memory in the heap is dealocated.

`rbt_print`

This function is used to print a specific tree. The tree must have been created using the `rbt_create` function and its `handler` should be known by the user, in order to be inserted as an argument. A `keyprinter` pointer to a function must also be provided as an argument. This function is provided by the key-type libraries (`IntegerKeys.h` and `StringKeys.h`). The function prints the tree in the terminal, or it returns 1 to inform the user that the inserted tree was empty.

Key-type `IntegerKey.h` and `StringKey.h` libraries

This libraries provide the user with functions to manipulate **key data-holder structures** that hold the key values of the nodes to be inserted in the red-black trees. These are vital because tree structure is based upon them and their relations.
Both libraries contain similar functions, that just support different data types (integers and strings) and in the future one can always create more libraries of similar structure with other data types, such as doubles.

It must be noted that in order to use the `StringKey.h` the user must add proper strings, created and manipulated according to the functions provided by the standard `<string.h>` C header file.
Also for the printing functions to be used, the user must also include the `<stdio.h>` header file.

`int_createkey` and `string_createkey`

These functions create a new key data-holder strucutre and they return a void pointer to it, that the user can insert as an argument to `RedBlackTree.h` functions, wherever it is needed. The user inputs as an

argumnent the value they wish to be contained in the key structure (a single integer or a string). If they return NULL it means that memory allocation in the heap failed and the user must terminate the program

`int_destroykey` and `string_destroykey`

These functions destroy the key data-holder structures, by dealocating their memory in the heap. The user must input as arguments a void pointer to the key structure they wish to destroy.
There is a small intricacy with the `string_destroykey`. Because string values inside the key data-holder structure are created by allocating memory in the heap, when `string_createkey` is called, these strings' memory is also dealocated when `string_destroykey` is called on them.

`int_compare` and `string_compare`

These functions take as arguments two pointers to two key data-holder structures and compare them to see if the first argument contains a greater value than the second. If it does it returns 1, else it returns 0. For integers this means arg1 > arg2 , while for strings it means that the first argument is of higher alphabetical order than the second.

`int_equal` and `string_equal`

These functions take as arguments two pointers to two key data-holder structures and compare them to see if their contained values are equal. If they are it returns 1, else it returns 0. For integers this means arg1 == arg2 , while for strings it means that the strings are identical, up to the null terminating character `'\0'`.

`int_print` and `string_print`

These functions provide the user the functionality to print the value contained inside the key data-holder struct, that they input as an argument to the functions.

Scaffold `main.c`

The main function of this project, `main.c`, is a scaffold to test the libraries' functions. It provides a user interface via terminal where the user has the ability to work on and modify up to 10 different intger type red-black trees and up to 10 different string type red-black trees. The different trees have IDs ranging from 0 to 9 (for integer and string types respectfully). These IDs are asked from the user everytime they wish to modify a certain tree.

The terminal menu provides the user options that are specifically designed to put in use the function of the libraries of the project. These options are:

insert

the user can insert a node in a specific tree. First the user must specify wether they want to insert a node in an integer type tree or a string type tree. Then they must provide the tree ID. If no tree with such ID exists then it is automatically created, and the node to be inserted will be the first node in the tree. Then they must input the value of the key they want the node to hold. Finally insertion in the tree is attempted.

If all goes well the program automatically resumes to the main menu. If something goes wrong then specific error messages are shown and then again the program returns to the main menu, except there is a fatal error.

Possible errors:

- inserting a tree ID outside the range 0 to 9
- inserting a node key that is equal with another node's key inside a tree

Functions from `RedBlackTree.h`, `IntegerNodes.h` and `StringNodes.h` used:

- type definition `handler` is used to declare pointers to sentinel structs of trees. These pointers are stored in tree ID arrays (the ID of the trees is basically the indexing of these arrays)
- if the tree ID inserted correspond to an empty position in the tree array then `rbt_create` is called and its value is passed to the proper ID in the array
- `int_createkey` or `string_createkey` based on what type of key the user wishes to input
- `rbt_insert` passing as arguments the proper `handler` from the trees ID arrays, the pointer to the created keys, and pointers to comparison,equality and destroykey functions from the key-type libraries

delete

The user can delete a node from a specific tree. Like insert they must specify tree type, tree ID and the key of the node they wish to delete. To find the node they wish `rbt_nodefind` is called, with `int_createkey`/ `string_createkey` as an input argument, to create a temporary key. The output of `rbt_nodefind` is inputed as an argument in `rbt_delete`.

If all goes well the program returns automatically to the main menu. If something goes wrong the user is propmpted by a specific error message before the program returns to the main menu.

Possible errors:

- inserting a tree ID outside the range 0 to 9
- invalid tree ID (empty spot in the ID array)
- inserting a node key that does not exist within the tree
- The tree is empty
- A serious mistake insidse the `RedBlackTree.c` source, in which case the program terminates
- Memory allocation in the heap failed, in which case the program terminates

There is also a special case, when the node deleted was the tree's only node in which case **the program will call `rbt_destroy` to destroy the sentinel of the specific tree**, completely erradicating from the propgram, and avoiding memory leaks.

Functions from `RedBlackTree.h`, `IntegerNodes.h` and `StringNodes.h` used:

- `rbt_nodefind` along with `int_createkey`/ `string_createkey` to find the node to be deleted
- `rbt_delete` with arguments the `handler` to the tree, containing the node to be deleted, a pointer to that node, and pointers to comparison,equality and destroykey functions from the key-type libraries
- in the case the node deleted was the only node of the tree, then `rbt_destroy` is called on that tree, using its `handler`

print node information

This operation prints the information of a specific node (that is its key value, its parent's and childrens' key value, and its color), from a specific tree. The user must specify the tree type, ID, and add the node's key.

If errors occur, possible error messages are shown:

- inserting a tree ID outside the range 0 to 9
- invalid tree ID (empty spot in the ID array)
- inserting a node key that does not exist within the tree

Functions from `RedBlackTree.h`, `IntegerNodes.h` and `StringNodes.h` used:

- `rbt_nodefind` along with `int_createkey`/ `string_createkey` to find the node to be printed
- `rbt_nodeprint` to print the node

print a tree

An entire tree can be printed, by inputing its type and ID. The nodes are shown in accending order and all their information (their key value, their parent's and childrens' key value, and their color) will be displayed. A special formatting is also used to show the levels of the tree. The rightmost nodes are in higher levels closer to the root, while the leftmost are lower, closer to the leaves.

Possible error messages:

- inserting a tree ID outside the range 0 to 9
- invalid tree ID (empty spot in the ID array)
- the tree is empty (basically impossible because empty trees are destroyed)

Functions from `RedBlackTree.h`, `IntegerNodes.h` and `StringNodes.h` used:

- `rbt_print` with pointers to `int_print` or `string_print` function, along with the `handler` to the tree's sentinel struct.

show all red-black trees

A very handy operation, that shows **all the existing red-black trees**, from the existing IDs in the tree ID arrays. It shows all their nodes, with their information. Also it shows each tree's ID.

Functions from `RedBlackTree.h`, `IntegerNodes.h` and `StringNodes.h` used:

- `rbt_print` with pointers to `int_print` or `string_print` function, along with the `handler` to the tree's sentinel struct.

quit

It terminates the program