EXERCISES — BST implementation

version #

ASSISTANTS C/UNIX 2022 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.assistants.epita.fr

1 BST implementation

Files to submit:

- bst/bst.c
- bst/bst.h
- bst/bst_static.c
- bst/bst_static.h

Provided files:

- bst/bst.h
- bst/bst_static.h

Authorized functions: You are only allowed to use the following functions:

- malloc(3)
- free(3)
- realloc(3)

Authorized headers: You are only allowed to use the functions defined in the following headers:

- err.h
- errno.h
- stddef.h
- assert.h

1.1 Goal

The goal of this exercise is to make you handle a BST in its linked representation and its sequential representation.

1.2 Linked representation

In this first part you will create the basic functions to create and use a BST in its linked representation. The structure of a node is given. Prototypes are available in `bst.h`.

1.2.1 Creation

```c
struct bst_node *create_node(int value);
```

This function creates a node that contains the value `value` and returns a pointer to this node. The children of this node are initialized to `NULL`.

1.2.2 Insertion

```c
struct bst_node *add_node(struct bst_node *tree, int value);
```

This function creates a node that contains the value `value` insert it in the BST `tree` at the right place. The resulting tree should still be a valid BST. This function returns a pointer to the root of the tree. Note that this function should also work if `NULL` is given as argument.

1.2.3 Deletion

```c
struct bst_node *delete_node(struct bst_node *tree, int value);
```

This function removes the first node of the `tree` containing the value `value`. It returns the root of the updated tree if the suppression is successful, `NULL` otherwise.

Be careful!

You have to conserve the order relation between the nodes of the BST.

If you have to delete a node with two children, you **must** replace the current value of the node with the maximum value of its left child.

1.2.4 Search

```c
const struct bst_node *find(const struct bst_node *tree, int value);
```

This function traverses the `tree` searching for the first node containing the value `value`. If this node is found, the function returns its pointer. Otherwise, it returns `NULL`.

1.2.5 Free

```c
void free_bst(struct bst_node *tree);
```

This function frees the `tree` **entirely**. Note that it should also work if `NULL` is given as argument. After this function, your `tree` **must not** be used.

1.3 Sequential representation

Now, you have to represent a BST in its sequential representation, using a static array. Remember: the left child is be found at index `2 * i + 1` and right child at index `2 * i + 2`.

For this exercise, you will use the following structure:

```c
struct value
{
    int val;
};

struct bst
{
    size_t capacity;
    size_t size;
    struct value **data;
};
```

The tree is stored in a static array, and its size is updated after each addition. There is also a `capacity` field that represents the size of the `data` array.

Here, we are working on integers, but keep in mind that it may be anything else.

1.3.1 Initialisation

```c
struct bst *init(size_t capacity);
```

This function creates a new tree with size 0 and initialise `data` with capacity `capacity`.

1.3.2 Insertion

```c
void add(struct bst *tree, int value);
```

This function creates a node that contains the value `value` insert it in the BST `tree` at the right place. The resulting tree should still be a valid BST. You have to consider the representation to check whether the array is big enough for another variable before addition.

If the size of `data` is lower than necessary, you will have to realloc it before performing the insertion.

Tips

Keep in mind that the worst case is the unbalanced tree (think about adding each value in order).

1.3.3 Search

```c
int search(struct bst *tree, int value);
```

This function traverses the `tree` searching for the first node containing the value `value` and returns its index if the value was found, `-1` otherwise.

1.3.4 Free

```c
void bst_free(struct bst *tree);
```

This function frees the `tree` **entirely**. Note that it should also work if `NULL` is given as argument. After this function, your `tree` **must not** be used.

It is my job to make sure you do yours.