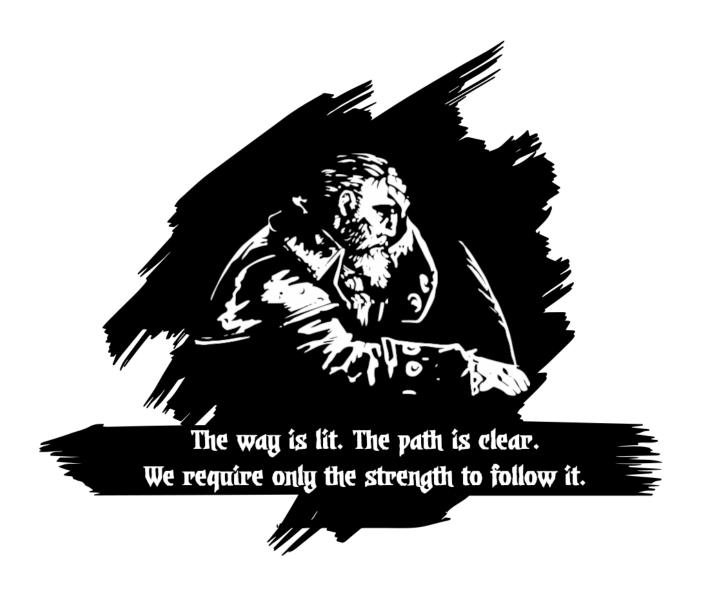


EXERCISES — BST

version #7be580532266ed398481e31366afcc24b1950c2a



Copyright

This document is for internal use at EPITA (website) only.

Copyright © 2022-2023 Assistants <assistants@tickets.assistants.epita.fr>

The use of this document must abide by the following rules:

- ▶ You downloaded it from the assistants' intranet.*
- ▷ This document is strictly personal and must **not** be passed onto someone else.
- ▷ Non-compliance with these rules can lead to severe sanctions.

Contents

1	Goal	3
2	Linked representation	3
	2.1 Creation	
	2.2 Insertion	
	2.3 Deletion	
	2.4 Search	
	2.5 Free	4
3	Sequential representation	5
	3.1 Initialisation	5
	3.2 Insertion	
	3.3 Search	6
	3./ Free	6

^{*}https://intra.assistants.epita.fr

File Tree

```
bst/
bst.c (to submit)
bst.h (to submit)
bst_static.c (to submit)
bst_static.h (to submit)
```

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

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

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

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

Compilation: Your code must compile with the following flags

• -std=c99 -pedantic -Werror -Wall -Wextra -Wvla

Main function: None

1 Goal

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

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.

2.1 Creation

```
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.

2.2 Insertion

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

This function creates a node that contains the value value and inserts 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.

2.3 Deletion

```
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.

2.4 Search

```
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.

2.5 Free

```
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.

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:

```
struct value
{
   int val;
};

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

The tree is stored in a dynamically allocated 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.

3.1 Initialisation

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

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

3.2 Insertion

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

This function creates a node that contains the value value and inserts 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).

3.3 Search

```
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.

3.4 Free

```
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.

The way is lit. The path is clear. We require only the strength to follow it.