Piscine — Tutorial D8 AM

version **#b4173ffc5996c36bf7f9730185ebc8357500bd86**

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Copyright

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

Contents

1 Unions **3**
- 1.1 Tagged unions . . . 3
- 1.2 Exercise: Variant . . . 4

2 Bitwise operators **5**
- 2.1 Introduction . . . 5
- 2.2 Operator descriptions . . . 6
- 2.3 Application . . . 8
- 2.4 Exercises . . . 9

3 Binary Trees **12**
- 3.1 Description . . . 12
- 3.2 Vocabulary . . . 12
- 3.3 Dynamic data structure: using `struct` . . . 13
- 3.4 Static data structure: hierarchically indexed array . . . 14
- 3.5 Traversal . . . 14
- 3.6 Exercise: Binary trees - static implementation . . . 16

4 Binary Search Trees **18**
- 4.1 Description . . . 18
- 4.2 Exercise: Binary Tree - Dynamic Implementation . . . 19
- 4.3 Exercise: BST . . . 20

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

1 Unions

This section will explain a way of writing generic data structures in *C*: unions.

Unions are used to represent a field that can be of many different types, but there can only be **one** stored inside it at a time. They are declared the same way as structures:

```
union int_or_string
{
  int int_t;
  char *str_t;
};
```

The previous union denotes a field that can either be an `int` or a `string`. It can then be used as follows:

```
union int_or_string value;

value.int_t = 12;
printf("The value is now: %i\n", value.int_t);

value.str_t = "acu";
printf("The value is now: %s\n", value.str_t);
```

The size of a union is the **maximum** of the size of its members. Consequently, the following union's size will be `sizeof(int64_t)` (the size of a pointer) which is 8 bytes.

```
#include <stdint.h>

union integer
{
  int8_t smallest;
  int16_t small;
  int32_t big;
  int64_t biggest;
};
```

Going further...

The header *stdint.h* provides typedefs for integers of many sizes. Whereas the size of an integer can vary depending on the computer's architecture, the headers garanties fixed width types. For more information, you can type *man stdint.h* in the terminal.

1.1 Tagged unions

Since there is no way to know which field was last assigned to a union, we have to add a **tag** to know the stored type.

Taking our previous union, we could write:

```
union int_or_string
{
    int int_t;
```

(continues on next page)

(continued from previous page)

```
    char *str_t;
};

enum age_or_name
{
    AGE,
    NAME
};

struct student
{
    enum age_or_name tag;
    union int_or_string data;
};
```

Using the `tag` field in the `student` structure, we can know the stored type in the union `int_or_string` and decide which behaviour we want.

```
int main(void)
{
    struct student ing1;
    ing1.tag = NAME;
    ing1.data.str_t = "Thibault";

    if (ing1.tag == NAME)
        printf("Hello ! My name is %s", ing1.data.str_t);
    else
        printf("Hello ! I am %d years old", ing1.data.int_t);
    return 0;
}
```

1.2 Exercise: Variant

1.2.1 Goal

In this exercise you will implement a variant using a tagged union. To do so, you need to write a structure named `variant` containing:

- An union, named `type_any`
- An enum representing the field currently held by the union, named `type`

The variant must be able to store the following types:

- `int`
- `float`
- `char`
- `const char *`

1.2.2 Display

Write a function that will print the content of a variant on the standard output, followed by a line feed (`\n`).

For instance, if the variant contains the following integer value: `12`, it should display `12\n`.

```
void variant_display(const struct variant *e);
```

1.2.3 Equal

Write a function that returns true if two variants have the same type **and** the same content.

> **Tips**
>
> Two NULL variants are equal.

```
bool variant_equal(const struct variant *left, const struct variant *right);
```

Note that you must include `stdbool.h` to have booleans.

1.2.4 Find

Write a function that will look for an element in a variant array. The function will return the index of the first matched element if found, otherwise it returns -1.

```
int variant_find(const struct variant *array, size_t len, enum type type,
                 union type_any value);
```

1.2.5 Sum

Write a function that returns the sum of all numeric elements in a variant array (`int` and `float`)

```
float variant_sum(const struct variant *array, size_t len);
```

2 Bitwise operators

2.1 Introduction

As you might have heard, all the data in a computer is stored in memory as a sequence of `1`s and `0`s or `bits`. The `bit` is the smallest possible unit of memory in a computer.

You have already seen how to write a number in binary.

For example, the number 42, when stored on 16 bits, will look like this in binary[1]:

[1] For readability and easier hexadecimal conversion, bits are often, if not always, written in groups of four.

```
0000 0000 0010 1010
```

Tips

There is no standard way to write a number in binary in *C*. However, there is a GNU libc extension which allows us to do so using the `0b` prefix.

```c
int binary = 0b101010; // binary == 42
int decimal = 101010;  // decimal == 101010
```

Be careful!

In this tutorial, we might write numbers in decimal, binary (prefixed with `0b`) or hexadecimal (prefixed with `0x`) notations.

Reminder: in hexadecimal notation, each digit corresponds to four bits.

```c
unsigned short dec = 4080;
unsigned short bit = 0b0000111111110000;
unsigned short hex = 0x0FF0;
```

Bear in mind that should you choose to represent a number in binary, decimal or hexadecimal, they will all contain the same value and its representation is interchangeable.

2.2 Operator descriptions

In addition to arithmetic operators, it is also possible to use another set of operators called bitwise operators. To better illustrate the operations we use the binary representations of numbers but as said previously, it is entirely possible to use these operators even if it is represented differently.

Furthermore, if the two numbers are not the same size (written on the same number of bits), the smaller is converted to the size of the biggest one.

```c
unsigned char hex_char = 0xF0;
unsigned short hex_short = 0xFF0F;

unsigned short xor_char_short = hex_char ^ hex_short;
/* 0x00F0 ^ 0xFF0F = 0xFFFF */
```

2.2.1 AND

The bitwise binary operator `&` (`AND`) executes the operation `AND` on each bit of the two arguments.

```c
unsigned char bin_9 = 0b1001;
unsigned char bin_12 = 0b1100;
unsigned char and_9_12 = bin_9 & bin_12; /* 0b1000 */
```

2.2.2 OR

The bitwise binary operator | (OR) executes the operation OR on each bit of the two arguments.

```
unsigned char bin_12 = 0b1100;
unsigned char bin_5 = 0b0101;
unsigned char or_12_5 = bin_12 | bin_5; /* 0b1101 */
```

2.2.3 XOR

The bitwise binary operator ^ (XOR) executes the operation XOR on each bit of the two arguments.

```
unsigned char bin_10 = 0b1010;
unsigned char bin_12 = 0b1100;
unsigned char xor_10_12 = bin_10 ^ bin_12; /* 0b0110 */
```

2.2.4 Left shift

The binary operator << executes a left shift on the bits of its operand: each bit is shifted to the left, and 0s are introduced on the right. The leftmost bits are discarded.

There are two operands: the left one is the one being shifted, and the right one is the (whole) number of shifts.

A left shift is equivalent to a **multiplication** by a power of 2 (watch out for the overflow).

```
unsigned short two = 2; /* 0b10 */
unsigned short two_mul_2 = two << 1; /* 0b100 */

unsigned short num = 183;      /* 0b10110111 */
unsigned short shift_3 = num << 3; /* 0b10111000 */
// res == 184
```

2.2.5 Right shift

The binary operator >> executes a right shift on the bits of its operand: each bit is shifted to the right, and 0s are introduced on the left, in the `unsigned` case. In the `signed` case, if the number is positive, it is shifted like an unsigned one, if not, the result is **implementation defined** and no specific behavior should be expected.

There are two operands: the left one is the one being shifted, and the right one is the (whole) number of shifts.

A right shift is equivalent to an integer **division** by a power of 2.

```
unsigned short four = 4; /* 0b100 */
unsigned short four_div_4 = four >> 2; /* 0b1 */

unsigned char a = 183;        /* 0b10110111 */
```

(continues on next page)

(continued from previous page)

```
unsigned char b = 3;
unsigned char res = a >> b;  /* 0b00010110 */
// res == 22
```

2.2.6 NOT

The bitwise unary operator ~ flips the bits of the number (1's become 0's and vice-versa).

```
unsigned char bin = 0xd5;  /* 1101 0101 */;
unsigned char flip = ~bin; /* 0010 1010 */
```

Tips

`~a` is equivalent to the one's complement of `a`, or the operation `a ^ -1`.

2.3 Application

Bitwise operators are mostly used for manipulating flags, where several booleans are stored on specific bits of a number. The number is not considered an integer anymore, but rather a pack of bits, each bit having a specific meaning.

Another use, often in conjunction with flags, is masks: they are used to select specific bits and ignore others.

For example, when manipulating pixels in RGB, the format is an `unsigned int`. Each color channel is encoded on 8 bits thus the three values are represented on different parts of the number:

```
0x0015A7B3
    RRGGBB
```

To extract the different parts, we can do this:

```
int color = 0x15A7B3;

int b_mask = 0x0000FF;
int g_mask = b_mask << 8;  // 0x00FF00
int r_mask = b_mask << 16; // 0xFF0000

int blue = color & b_mask;           // 0x0000B3
int green = (color & g_mask) >> 8;   // 0x0000A7
int red = (color & r_mask) >> 16;    // 0x000015
```

2.3.1 Set a bit

To set a bit to `1`, you need to create a mask (usually with a shift) with the value `1` for the bit you want to set. Then you can apply the `OR` operator (written `|`) to set the value.

```
unsigned short value = 0x000A;      /* 0000 0000 0000 1010 */
unsigned short mask = 1 << 4;       /* 0000 0000 0001 0000 */
unsigned short res = value | mask; /* 0000 0000 0001 1010 */
```

2.3.2 Unset a bit

To set a bit to `0`, you need a different kind of mask: the negation of the previous mask. That is, every bit must be `1` *except* for the one you want to unset. Then you can apply the `AND` operator (written `&`) to unset the value.

```
unsigned short value = 0x000A;      /* 0000 0000 0000 1010 */
unsigned short mask = ~(1 << 1);    /* 1111 1111 1111 1101 */
unsigned short res = value & mask; /* 0000 0000 0000 1000 */
```

2.3.3 Test a bit

To test a bit, you need the same mask as for setting it, but instead use `&`. If the bit is unset, the result will be `0`, otherwise the result is the value of the mask.

```
unsigned short value = 0x000A;       /* 0000 0000 0000 1010 */
unsigned short flag_unset = 1 << 4; /* 0000 0000 0001 0000 */
unsigned short flag_set = 1 << 3;    /* 0000 0000 0000 1000 */

unsigned short nope = value & flag_unset; /* 0000 0000 0000 0000 */
unsigned short yeay = value & flag_set;   /* 0000 0000 0000 1000 */
```

2.4 Exercises

2.4.1 Test a bit

Goal

Write the `is_set` function that returns `true` if the `n`th bit is set, `false` otherwise.

Be careful!

`n` is 1-based and will always be greater than zero.

```
unsigned int is_set(unsigned int value, unsigned char n);
```

Example

```c
#include <stdio.h>

#include "is_set.h"

int main(void)
{
    printf("%d\n", is_set(24, 4));
    printf("%d\n", is_set(24, 3));

    return 0;
}
```

```
42sh$ gcc -Wall -Werror -std=c99 -Wextra -Wvla -pedantic test.c is_set.c -o is_set_example
42sh$ ./is_set_example
1
0
```

2.4.2 Bit rotation

Goal

Rotate the bits of `value` by `roll` rolls to the left. Each roll will shift the bits by one bit in the left direction. The leftmost bits are placed on the right.

```
[1]1101010 -> 1101010[1]
```

The prototype:

```c
unsigned char rol(unsigned char value, unsigned char roll);
```

2.4.3 Binary cipher

Goal

In this exercise, you will be use bitwise operators to manipulate the given data. You will need to implement two functions :

- One to cipher texts using the xor operator.
- One to cipher texts by using bit rotations.

Where `data` is a pointer to the data that will be ciphered of length `data_len`. The key is given by the `key` pointer of length `key_len`. The length of the given key can be shorter than the given data. If such is the case, continue ciphering the text from the start of the key. The key is repeated as many times as necessary to have the same length as the data. Each byte of the data is combined with the corresponding byte of the key.

The data must be modified in-place.

my_xor_crypt

The prototype is the following:

```
/*
** XOR byte by byte the data of size data_len with the key of size
** key_len.
*/
void my_xor_crypt(void *data, size_t data_len, const void *key, size_t key_len);
```

Example:

```
char text[] = "Example";
my_xor_crypt(text, 7, "KEY", 3)
```

```
message :   E     x    a    m    p    l    e
key :       K     E    Y    K    E    Y    K
result :  E^K  x^E  a^Y  m^K  p^E  l^Y  e^K
```

my_rol_crypt

The prototype is the following:

```
/*
** Rotate byte by byte the data of size data_len with the key of size
** key_len.
** Rotating a byte by another means adding them together with potential
** overflow.
*/
void my_rol_crypt(void *data, size_t data_len, const void *key, size_t key_len);
```

Example:

```
message :   E     x    a    m    p    l    e
key :       K     E    Y    K    E    Y    K
result :  E+K  x+E  a+Y  m+K  p+E  l+Y  e+K
```

As the security of the data is not the objective here, you can leave the data and the key written in clear in the code.

You should work on the data using `unsigned char`.

3 Binary Trees

3.1 Description

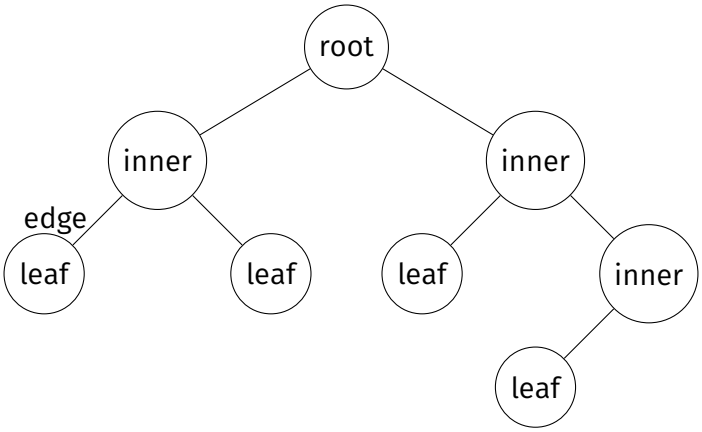
A tree is a **recursive** data structure with a set of **nodes**. The first node is called the **root**, and every node has children that are nodes themselves. The final nodes that do not have any children are called **leaves**.

In the case of **binary** trees, every node has at most two children: the *left* child and the *right* one. The recursive representation is: $B = \langle B_l, o, B_r \rangle$

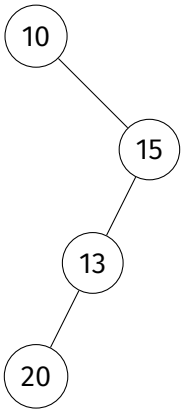
- o is the value contained in the node.
- B_l and B_r are the two children of B.
- B_l and B_r are either binary trees or the empty set: $\emptyset$.

3.2 Vocabulary

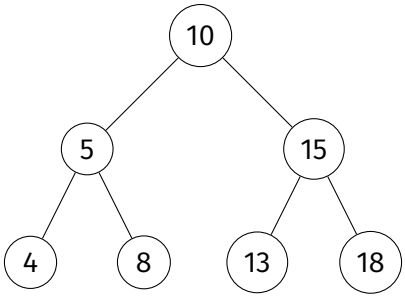
- **Root**: the top node in a tree
- **Internal node**: a node with at least one child
- **Leaf**: a node with no children (also called *external node*)
- **Edge**: a connection between one node and another



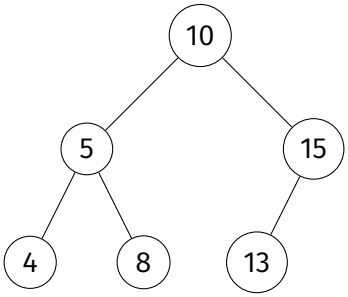
- **Size**: the size of a tree is equal to the number of nodes in it.
- **Level**: distance from the root to a node. The root is at level 0. Each node is thus on the `level of their parent node + 1`.
- **Height**: the height of a tree is the number of edges in the longest branch of a tree. An empty tree has a height of -1. A tree with a leaf root has a height of 0.
- **Degenerate tree**: a tree in which each node has at most one child.



- **Perfect tree**[1] : a tree in which all inner nodes have two children and all leaves are on the same level.



- **Complete tree**[1] : a tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left to right.



3.3 Dynamic data structure: using `struct`

In *C*, trees can be represented recursively using a structure:

```
struct binary_tree
{
    int                 key;
    struct binary_tree *left_child;
    struct binary_tree *right_child;
};
```

[1] In French, the names of the tree types are switched: complete tree are called *arbres parfaits* and perfect trees are *arbres complets*.

3.4 Static data structure: hierarchically indexed array

Another representation can be achieved using an array:

- The tree is contained in a contiguously allocated memory zone of fixed size.
- The element at position i is the value of the node.
- The children of the node at position i are located at the positions $2i + 1$ and $2i + 2$.

Be careful!

Unless the tree is *complete*, this data structure will waste space.

Going further...

As you may have noticed, the `struct` representation will also waste space because even `NULL` pointers have a size of 8.

Because optimization should be the very last part of a program, use whichever representation you are more comfortable with if and when we let you choose.

3.5 Traversal

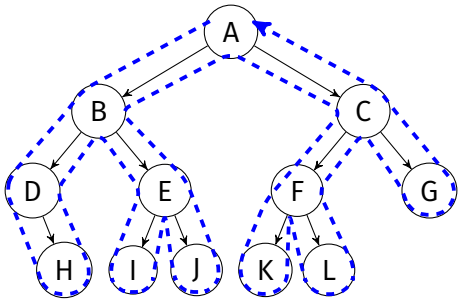
A traversal of a tree is a way of accessing each node of a given tree. There are two ways to do so:

- depth-first traversal
- breadth-first traversal

3.5.1 Depth-First Traversal

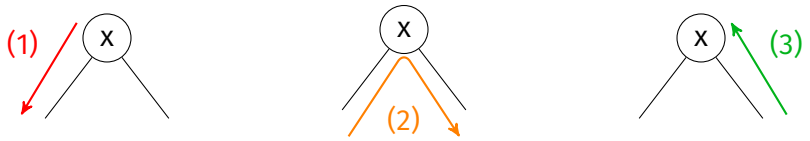
This traversal is recursive by nature. Here is the algorithm:

```
1:  function BIN_DEPTH_FIRST_TRAVERSAL(n)
2:      if n = NULL then
3:          return
4:      end if
5:      ▶ Pre-order operation (1)
6:      bin_depth_first_traversal(n.left_child)
7:      ▶ In-order operation (2)
8:      bin_depth_first_traversal(n.right_child)
9:      ▶ Post-order operation (3)
10: end function
```



When using this traversal, we encounter each node at most three times. This induces three orders to process those nodes:

- **Pre-order**: the order of the first encounter of a node.
- **In-order**: the order of the second encounter of a node. It happens when we are done visiting the subtree's first child.
- **Post-order**: the order of the last encounter of a node. It happens when we are done visiting the subtree's second child.



(a) Pre-order (b) In-order (c) Post-order

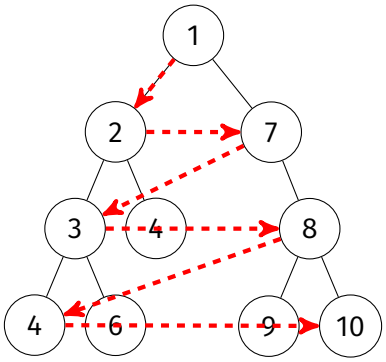
An operation can be anything here, like printing the value the node contains or evaluating a mathematical operation.

3.5.2 Breadth-First Traversal

This traversal is iterative. It begins at the root node. Then, for each child node in turn, it inspects their children nodes and so on. It is also called the level-order traversal.

Here is the algorithm:

```
1:  function BIN_BREADTH_FIRST_TRAVERSAL(root)
2:      queue ← empty_queue()
3:      enqueue(queue, root)
4:      while not empty(queue) do
5:          node ← dequeue(queue)
6:          ▶ Operations on the current node
7:          if node.left_child ≠ NULL then
8:              enqueue(queue, node.left_child)
9:          end if
10:         if node.right_child ≠ NULL then
11:             enqueue(queue, node.right_child)
12:         end if
13:     end while
14: end function
```



This algorithm uses a queue to store the two children of each visited node.

Tips

A queue is a first in first out structure, meaning that the first element inserted will be the first removed.

3.6 Exercise: Binary trees - static implementation

Now, you have to represent a binary tree using an array. As a reminder, the left child may 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
#ifndef BINARY_TREE_STATIC_H
#define BINARY_TREE_STATIC_H

#include <stddef.h>

#define CAPACITY 128

struct value
{
    int val;
};

struct binary_tree
{
    size_t max_index;
    struct value *data[CAPACITY];
};

size_t size(const struct binary_tree *tree);
int height(const struct binary_tree *tree);
void dfs_print_prefix(const struct binary_tree *tree);
void bfs_print(const struct binary_tree *tree);
int is_complete(const struct binary_tree *tree);
int is_perfect(const struct binary_tree *tree);

#endif /* !BINARY_TREE_STATIC_H */
```

The tree nodes are labeled with `struct value` as defined previously, but keep in mind that it may be anything else.

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

Be careful!

`max_index` represents the last possible node. It is the array size, the maximal index you would like to look at. If the tree is complete, it also represents the size.

3.6.1 Size

```
size_t size(const struct binary_tree *tree);
```

This function returns the size (number of nodes) of `tree`.

3.6.2 Height

```
int height(const struct binary_tree *tree);
```

This function returns the height of `tree`.

3.6.3 Depth-first traversal

```
void dfs_print_prefix(const struct binary_tree *tree);
```

This function displays the labels of the nodes of `tree` in the prefix order when doing a left-to-right traversal. This is what your functions should give when applied to the tree of the first example:

- `10 5 4 8 15 13 18 16`.

3.6.4 Breadth-first traversal

```
void bfs_print(const struct binary_tree *tree);
```

This function prints each node, level per level, from left-to-right. You should print a pipe (`|`) between each level. You should get the following output with the tree of the first example:

- `10 | 5 15 | 4 8 13 18 | 16`

3.6.5 Is complete

```
int is_complete(const struct binary_tree *tree);
```

This function returns `1` if the tree `tree` is complete, `0` otherwise.

3.6.6 Is perfect

```
int is_perfect(const struct binary_tree *tree);
```

This function returns `1` if the tree `tree` is perfect, `0` otherwise.

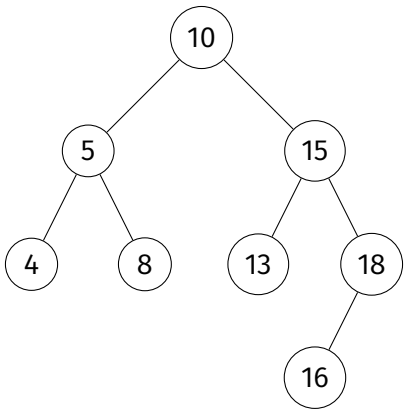
4 Binary Search Trees

4.1 Description

A binary search tree (*BST*) is a binary tree where the nodes are **labeled**, meaning they contain a value. The latter must belong to a set with a total order relation:

For every node x of the tree:

- All the labels of the nodes of the **left** subtree are **less than or equal** to the label of x.
- All the labels of the nodes of the **right** subtree are **greater** to the label of x.



A binary search tree is useful to quickly find an element in the structure. With a tree `T`, at most `height(T)` comparisons are needed to find an element. Therefore, if the tree is perfect, finding an element is proportional to $log_2(size(T))$, like dichotomy in an array. However, unlike arrays, we can efficiently add and remove elements. For the sake of simplicity, in this tutorial we will use integers as values but any set satisfying a total order relation is valid.

4.2 Exercise: Binary Tree - Dynamic Implementation

4.2.1 Goal

In this exercise you will implement some useful functions to work on binary trees. Your binary tree will have its nodes labeled with `int` values, and is implemented using the following structure:

```
struct binary_tree
{
    int data;
    struct binary_tree *left;
    struct binary_tree *right;
};
```

The `NULL` pointer represents an empty tree.

4.2.2 Size

```
int size(const struct binary_tree *tree);
```

This function returns the size (number of nodes) of `tree`.

4.2.3 Height

```
int height(const struct binary_tree *tree);
```

This function returns the height of `tree`.

4.2.4 Depth-first traversal

```
void dfs_print_prefix(const struct binary_tree *tree);
void dfs_print_infix(const struct binary_tree *tree);
void dfs_print_postfix(const struct binary_tree *tree);
```

These functions, also known as Depth-first traversal, display the labels of the nodes of `tree` in the specified order when doing a left-to-right traversal. They print the values separated with a space and a trailing white space and **NO** line feed.

For example, the output could look like this:

```
42sh$ ./example | cat -e
1 2 3 42sh$
```

If the tree is empty, you have to print an empty string.

```
42sh$ ./example | cat -e
42sh$
```

4.2.5 Is perfect

```
int is_perfect(const struct binary_tree *tree);
```

This function returns `1` if the tree `tree` is perfect, `0` otherwise.

4.2.6 Is complete

```
int is_complete(const struct binary_tree *tree);
```

This function returns `1` if the tree `tree` is complete, `0` otherwise.

4.2.7 Is degenerate

```
int is_degenerate(const struct binary_tree *tree);
```

This function returns `1` if the tree `tree` is a degenerate tree, `0` otherwise.

A degenerate tree is a tree where each node has at most one child.

4.2.8 Is full

```
int is_full(const struct binary_tree *tree);
```

This function returns `1` if the tree `tree` is full, `0` otherwise.

A full binary tree is a tree where each node is either a leaf or has two child.

4.2.9 Is BST

```
int is_bst(const struct binary_tree *tree);
```

This function returns `1` if all nodes in the binary tree `tree` are sorted according to a total order (the tree is a binary search tree), `0` otherwise.

4.3 Exercise: BST

4.3.1 Goal

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

4.3.2 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`.

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`.

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.

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.

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`.

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.

4.3.3 Sequential representation

Now, you have to represent a BST in its sequential representation, using a static array.

Tips

The left child is 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.

Initialisation

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

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

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 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).

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.

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.

I must not fear. Fear is the mind-killer.